Contents

- dotnet new --search
- dotnet new --install
- dotnet new --uninstall
- dotnet new --update-*
- .NET 默认模板
- 自定义模板

dotnet nuget

- dotnet nuget delete
- dotnet nuget locals
- dotnet nuget push
- dotnet nuget add source
- dotnet nuget disable source
- dotnet nuget enable source
- dotnet nuget list source
- dotnet nuget remove source
- dotnet nuget update source
- dotnet nuget verify
- dotnet nuget trust
- dotnet nuget sign

dotnet pack

dotnet publish

dotnet restore

dotnet run

dotnet sdk check

dotnet sln

dotnet store

dotnet test

dotnet tool

- dotnet tool install
- dotnet tool list
- dotnet tool restore
- dotnet tool run

CA1012
CA1014
CA1016
CA1017
CA1018
CA1019
CA1021
CA1024
CA1027
CA1028
CA1030
CA1031
CA1032
CA1033
CA1034
CA1036
CA1040
CA1041
CA1043
CA1044
CA1045
CA1046
CA1047
CA1050
CA1051
CA1052
CA1053
CA1054
CA1055
CA1056
CA1058
CA1060

CA2327
CA2328
CA2329
CA2330
CA2350
CA2351
CA2352
CA2353
CA2354
CA2355
CA2356
CA2361
CA2362
CA3001
CA3002
CA3003
CA3004
CA3005
CA3006
CA3007
CA3008
CA3009
CA3010
CA3011
CA3012
CA3061
CA3075
CA3076
CA3077
CA3147
CA5350
CA5351

CA5358

CA5359

CA5360

CA5361

CA5362

CA5363

CA5364

CA5365

CA5366

CA5367

CA5368

CA5369

CA5370

CA5371

CA5372

CA5373

CA5374

CA5375

CA5376

CA5377

CA5378

CA5379

CA5380

CA5381

CA5382

CA5383

CA5384

CA5385

CA5386

CA5387

CA5388

CA5389

IL2019

IL2022

IL2023

IL2024

IL2025

IL2026

IL2027

IL2028

IL2029

IL2030

IL2031

IL2032

IL2033

IL2034

IL2035

IL2036

IL2037

IL2038

IL2039

IL2040

IL2041

IL2042

IL2043

IL2044

IL2045

IL2046

IL2048

IL2049

IL2050

IL2051

IL2052

IL2053

.NET 入门

2021/11/16 •

本文介绍如何创建和运行“Hello World!” .NET 应用。

如果不确定什么是 .NET，请从 .NET 简介开始。

创建应用程序

首先，在计算机上下载并安装 .NET SDK。

然后，打开某一终端，如 PowerShell、命令提示符或 Bash 。输入以下 `dotnet` 命令，创建并运行 C# 应用程序：

```
dotnet new console --output sample1
dotnet run --project sample1
```

可以看到以下输出：

```
Hello World!
```

祝贺你！ 你现已创建了一个简单的 .NET 应用程序。

后续步骤

按照分步教程或观看 YouTube 上的 .NET 101 视频，开始开发 .NET 应用程序。

.NET 入门教程

2021/11/16 •

以下分布教程会在 Windows、Linux 或 macOS 上运行，除非有特别说明。

创建应用的教程

- 创建控制台应用
  - 使用 Visual Studio Code
  - 使用 Visual Studio (Windows)
  - 使用 Visual Studio for Mac (macOS)
- 创建 Web 应用
  - 使用服务器端 Web UI
  - 使用客户端 Web UI
- 创建 Web API
- 创建远程过程调用 Web 应用
- 创建实时 Web 应用
- 在云中创建无服务器函数
- 创建适用于 Android 和 iOS 的移动应用 (Windows)
- 创建 Windows 桌面应用
  - WPF
  - Windows 窗体
  - 通用 Windows 平台 (UWP)
- 使用 Unity 创建游戏
- 创建 Windows 服务

类库创建教程

- 创建类库
  - 使用 Visual Studio Code
  - 使用 Visual Studio (Windows)
  - 使用 Visual Studio for Mac (macOS)

.NET 语言学习资源

- C# 入门
- F# 入门
- Visual Basic 入门

其他入门资源

以下资源适用于 .NET 应用开发入门，但不是分步教程：

- 物联网 (IoT)
- 机器学习

后续步骤

若要详细了解 .NET，请参阅 .NET 简介。

在 Windows 上安装 .NET

2021/11/16 •

本文介绍如何在 Windows 上安装 .NET。.NET 由运行时和 SDK 组成。运行时用于运行 .NET 应用，应用可能包含也可能不包含它。SDK 用于创建 .NET 应用和库。.NET 运行时始终随 SDK 一起安装。

最新版本的 .NET 是 5.0。

支持的版本

下表列出了当前支持的 .NET 版本以及支持它们的 Windows 版本。这些版本在 .NET 版本达到支持终止日期或 Windows 版本达到生命周期之前仍受支持。

Windows 10 版本终止服务日期按版本分段。下表中仅考虑家庭版、专业版、专业教育版和专业工作站版。查看 Windows 生命周期事实表单，了解具体的详细信息。

TIP

+ 表示最低版本。

	.NET CORE 2.1	.NET CORE 3.1	.NET 5
Windows 11	✗	✓	✓
Windows Server 2022	✗	✓	✓
Windows 10 版本 21H1	✗	✓	✓
Windows 10/Windows Server 版本 20H2	✗	✓	✓
Windows 10/Windows Server 版本 2004	✗	✓	✓
Windows 10/Windows Server 版本 1909	✗	✓	✓
Windows 10/Windows Server 版本 1903	✗	✓	✓
Windows 10 版本 1809	✗	✓	✓
Windows 10 版本 1803	✗	✓	✓
Windows 10 版本 1709	✗	✓	✓
Windows 10 版本 1607	✗	✓	✓

[illegible]	.NET CORE 2.1	.NET CORE 3.1	.NET 5
Windows 8.1	✕	✔	✔
Windows 7 SP1 ESU	✕	✔	✔
Windows Server 2019 Windows Server 2016 Windows Server 2012 R2 Windows Server 2012	✕	✔	✔
Windows Server Core 2012 R2	✕	✔	✔
Windows Server Core 2012	✕	✔	✔
Nano Server 版本 1809+	✕	✔	✔
Nano Server 版本 1803	✕	✔	✕

不支持的版本

以下 .NET 版本 ✕ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

运行时信息

运行时用于运行使用 .NET 创建的应用。应用作者发布应用时，可以在其应用中包含运行时。如果作者未包含运行时，则由用户安装运行时。

可以在 Windows 上安装三个不同的运行时：

- *ASP.NET Core 运行时*
  运行 ASP.NET Core 应用。包括 .NET 运行时。
- *桌面运行时*
  运行适用于 Windows 的 .NET WPF 和 Windows 窗体桌面应用。包括 .NET 运行时。
- .NET 运行时
  此运行时是最简单的运行时，不包括任何其他运行时。强烈建议同时安装 ASP.NET Core 运行时和桌面运行时，以最大限度地提升与 .NET 应用的兼容性 。

[illegible] .NET [illegible]

SDK 信息

SDK 用于生成和发布 .NET 应用和库。安装 SDK 会包含三个运行时：ASP.NET Core、桌面和 .NET。

依赖项

- .NET 5
- .NET Core 3.1
- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1

.NET 5 支持下列 Windows 版本：

> **NOTE**
> + 表示最低版本。

(OS)	VERSION	IIII
Windows 11	21H2	x64、ARM64
Windows 10 客户端	1607+	x64、x86、ARM64
Windows 客户端	7 SP1+、8.1	x64、x86
Windows Server	2012+	x64、x86
Windows Server 核心	2012+	x64、x86
Nano Server	1809+	X64

有关 .NET 5 支持的操作系统、发行版和生命周期策略的详细信息，请参阅 .NET 5 支持的 OS 版本。

Windows 7 / Vista / 8.1 / Server 2008 R2 / Server 2012 R2

如果要在以下 Windows 版本上安装 .NET SDK 或运行时，则需要其他依赖项：

IIII	IIII
Windows 7 SP1 ESU	- Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位 - KB3063858 64 位 / 32 位 - Microsoft 根证书颁发机构 2011（仅限 .NET Core 2.1 脱机安装程序）
Windows Vista SP 2	Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位
Windows 8.1	Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位
Windows Server 2008 R2	Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位
Windows Server 2012	Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位
Windows Server 2012 R2	Microsoft Visual C++ 2015-2019 Redistributable 64 位 / 32 位

如果收到与以下 dll 之一相关的错误，也需要满足上述要求：

- api-ms-win-crt-runtime-l1-1-0.dll
- api-ms-win-cor-timezone-l1-1-0.dll
- hostfxr.dll

使用 PowerShell 自动化安装

dotnet-install 脚本用于运行时的 CI 自动化和非管理员安装。可从 dotnet-install 脚本引用页下载该脚本。

此脚本默认安装最新的长期支持 (LTS) 版本，即 .NET Core 3.1。可通过指定 `Channel` 开关以选择特定版本。包括 `Runtime` 开关以安装运行时。否则，该脚本安装 SDK。

```
dotnet-install.ps1 -Channel 5.0 -Runtime aspnetcore
```

通过省略 `-Runtime` 开关来安装 SDK。在此示例中将 `-Channel` 开关设置为 `Current`，这将安装受支持的最新版本。

```
dotnet-install.ps1 -Channel Current
```

使用 Visual Studio 安装

如果你要使用 Visual Studio 开发 .NET 应用，请参阅下表，了解不同目标 .NET SDK 版本所需的 Visual Studio 最低版本。

.NET SDK	VISUAL STUDIO
5.0	Visual Studio 2019 版本 16.8 或更高版本。
3.1	Visual Studio 2019 版本 16.4 或更高版本。
3.0	Visual Studio 2019 版本 16.3 或更高版本。
2.2	Visual Studio 2017 版本 15.9 或更高版本。
2.1	Visual Studio 2017 版本 15.7 或更高版本。

如果你已安装 Visual Studio，则可以使用以下步骤检查你的版本。

1. 打开 Visual Studio。
2. 选择“帮助” > “Microsoft Visual Studio”。
3. 从“关于”对话框中读取版本号。

Visual Studio 可安装最新的 .NET SDK 和运行时。

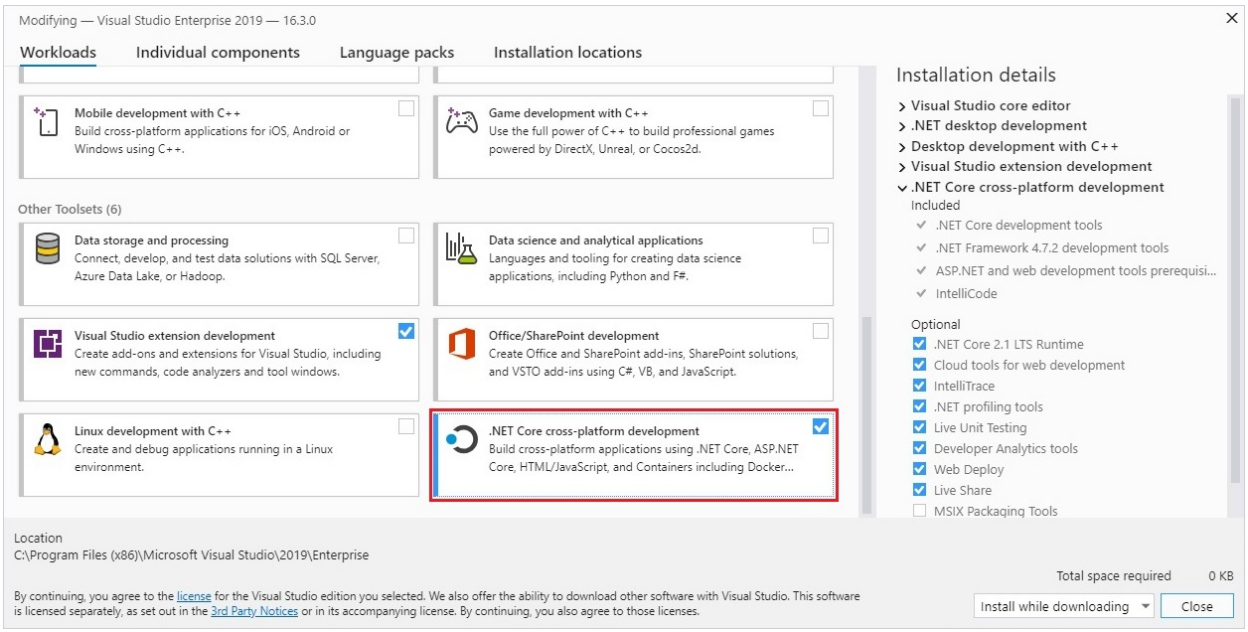
VISUAL STUDIO。

选择工作负载

安装或修改 Visual Studio 时，根据要生成的应用程序的类型，选择以下一个或多个工作负载：

- “其他工具集”部分中的“.NET Core 跨平台开发”工作负荷 。
- “Web 和云”部分中的“ASP.NET 和 Web 开发”工作负荷 。

- “Web 和云”部分中的“Azure 开发”工作负载 。
- “桌面和移动”部分中的“NET 桌面开发”工作负载 。



随 Visual Studio Code 一起安装

Visual Studio Code 是一个功能强大的轻量级源代码编辑器，可在桌面上运行。Visual Studio Code 适用于 Windows、macOS 和 Linux。

虽然 Visual Studio Code 不像 Visual Studio 一样附带自动的 .NET Core 安装程序，但添加 .NET Core 支持非常简单。

1. 下载并安装 Visual Studio Code。
2. 下载并安装 .NET SDK。
3. 从 Visual Studio Code 市场安装 C# 扩展。

Windows Installer

适用于 .NET 的下载页面提供了 Windows Installer 可执行文件。

使用 Windows 安装程序安装 .NET 时，可以通过设置 `DOTNETHOME_X64` 和 `DOTNETHOME_X86` 参数来自定义安装路径：

```
dotnet-sdk-3.1.301-win-x64.exe DOTNETHOME_X64="F:\dotnet\x64" DOTNETHOME_X86="F:\dotnet\x86"
```

如果要以无提示方式安装 .NET（例如在生产环境中）或要支持持续集成，请使用以下开关：

- `/install`
  安装 .NET。
- `/quiet`
  禁止显示任何 UI 和提示。
- `norestart`
  禁止任何重启尝试。

```
dotnet-sdk-3.1.301-win-x64.exe /install /quiet /norestart
```

有关详细信息，请参阅标准安装程序命令行选项。

> **TIP**
> 安装程序返回退出代码 0 以表示成功，返回退出代码 3010 以表示需要重启。任何其他值通常都是错误代码。

下载并手动安装

除了使用适用于 .NET 的 Windows 安装程序，还可以下载并手动安装 SDK 或运行时。手动安装通常作为持续集成测试的一部分执行。对于开发人员或用户，一般使用安装程序会更好。

在下载 .NET SDK 和 .NET 运行时后，可以手动安装它们。如果安装 .NET SDK，则无需安装相应的运行时。首先，从以下站点之一下载 SDK 或运行时的二进制版本：

- .NET 5 下载
- .NET Core 3.1 下载
- 所有 .NET Core 下载项

创建要将 .NET 提取到的目录，例如 `%USERPROFILE%\dotnet`。然后，将下载的 zip 文件提取到该目录中。

默认情况下，.NET CLI 命令和应用不会使用通过这种方式安装的 .NET，并且你必须显式选择才能使用它。为此，请更改用于启动应用程序的环境变量：

```
set DOTNET_ROOT=%USERPROFILE%\dotnet
set PATH=%USERPROFILE%\dotnet;%PATH%
set DOTNET_MULTILEVEL_LOOKUP=0
```

使用此方法可以将多个版本安装到不同的位置，然后通过使用指向安装位置的环境变量运行应用程序来明确选择应用程序应使用哪个安装位置。

将 `DOTNET_MULTILEVEL_LOOKUP` 设置为 `0` 时，.NET 将忽略任何全局安装的 .NET 版本。删除环境设置，让 .NET 在选择用于运行应用程序的最佳框架时考虑默认的全局安装位置。默认值通常为 `C:\Program Files\dotnet`，这是安装 .NET 的安装程序所在的位置。

Docker

容器提供了一种将应用程序与主机系统的其余部分隔离的轻量级方法。同一计算机上的容器只共享内核，并使用为应用程序提供的资源。

.NET 可在 Docker 容器中运行。官方 .NET Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可在 Microsoft.NET Docker Hub 存储库中找到这些映像。每个存储库包含 .NET（SDK 或运行时）和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，ASP.NET Core 存储库提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

有关在 Docker 容器中使用 .NET 的详细信息，请参阅 .NET 和 Docker 简介和示例。

后续步骤

- 如何检查是否已安装 .NET。
- 教程：Hello World 教程。
- 教程：使用 Visual Studio Code 创建一个新应用。
- 教程：使 .NET Core 应用容器化。

在 macOS 上安装 .NET

2021/11/16 •

在本文中，你将了解如何在 macOS 上安装 .NET。.NET 由运行时和 SDK 组成。运行时用于运行 .NET 应用，应用可能包含也可能不包含它。SDK 用于创建 .NET 应用和库。.NET 运行时始终随 SDK 一起安装。

最新版本的 .NET 是 5.0。

支持的版本

下表列出了当前支持的 .NET 版本以及支持它们的 macOS 版本。这些版本仍受支持，直到 .NET 版本达到支持终止日期。

- ✔ 指示 .NET Core 版本仍受支持。
- ✕ 指示 .NET Core 版本不受支持。

	.NET CORE 2.1	.NET CORE 3.1	.NET 5
macOS 11.0"Big Sur"	✕ 2.1(发行说明)	✔ 3.1(发行说明)	✔ 5.0(发行说明)
macOS 10.15"Catalina"	✕ 2.1(发行说明)	✔ 3.1(发行说明)	✔ 5.0(发行说明)
macOS 10.14"Mojave"	✕ 2.1(发行说明)	✔ 3.1(发行说明)	✔ 5.0(发行说明)
macOS 10.13"High Sierra"	✕ 2.1(发行说明)	✔ 3.1(发行说明)	✔ 5.0(发行说明)
macOS 10.12"Sierra"	✕ 2.1(发行说明)	✕ 3.1(发行说明)	✕ 5.0(发行说明)

不支持的版本

以下 .NET 版本 ✕ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

运行时信息

运行时用于运行使用 .NET 创建的应用。应用作者发布应用时，可以在其应用中包含运行时。如果作者未包含运行时，则由用户安装运行时。

macOS 上可以安装两种不同的运行时：

- *ASP.NET Core 运行时*
  运行 ASP.NET Core 应用。包括 .NET 运行时。
- .NET 运行时

此运行时是最简单的运行时，不包括任何其他运行时。强烈建议安装 ASP.NET Core 运行时，以最大限度地提升与 .NET 应用的兼容性。

.NET

SDK 信息

SDK 用于生成和发布 .NET 应用和库。安装 SDK 会包含两个运行时：ASP.NET Core 和 .NET。

依赖项

以下 macOS 版本支持 .NET：

NOTE

+ 表示最低版本。

.NET CORE	MACOS		
5.0	High Sierra (10.13+)	X64	详细信息
3.1	High Sierra (10.13+)	X64	详细信息
3.0	High Sierra (10.13+)	X64	详细信息
2.2	Sierra (10.12+)	X64	详细信息
2.1	Sierra (10.12+)	X64	详细信息

自 macOS Catalina（版本10.15）开始，所有在 2019 年 6 月 1 日之后生成并使用开发者 ID 扩散的软件都必须经过公证。此要求适用于 .NET 运行时、.NET SDK 以及使用 .NET 创建的软件。

自 2020 年 2 月 18 日起，.NET 5 和 .NET Core 3.1、3.0 和 2.1 的运行时和 SDK 安装程序都已经过公证。以前发布的版本没有经过公证。如果运行未经过公证的应用，将看到类似于下图的错误：

若要详细了解强制执行的公证要求对 .NET 和 .NET 应用的影响，请参阅处理 macOS Catalina 公证。

libgdiplus

使用 System.Drawing.Common 程序集的 .NET 应用程序要求安装 libgdiplus。

获取 libgdiplus 的一个简单方法是使用适用于 macOS 的 Homebrew ("brew") 包。在安装 brew 后，通过在终端（命令）提示符处执行以下命令来安装 libgdiplus：

```
brew update
brew install mono-libgdiplus
```

使用安装程序安装

macOS 具有独立的安装程序，可用于安装 .NET 5 SDK：

- x64（64 位）CPU

下载并手动安装

除了使用适用于 .NET 的 macOS 安装程序，还可以下载并手动安装 SDK 和运行时。手动安装通常作为持续集成测试的一部分执行。对于开发人员或用户，一般使用安装程序会更好。

首先，从以下站点之一下载 SDK 或运行时的二进制版本。如果安装 .NET SDK，则无需安装相应的运行时：

- ✔ .NET 5 下载
- ✔ .NET Core 3.1 下载
- ✔ .NET Core 2.1 下载
- 所有 .NET Core 下载项

接下来，提取已下载的文件并使用 `export` 命令将 `DOTNET_ROOT` 设置为提取文件夹的位置，然后确保 .NET 位于 PATH 中。这应会使 .NET CLI 命令在终端中可用。

或者，下载 .NET 二进制文件后，可以从保存文件的目录运行以下命令以提取运行时。这也会使 .NET CLI 命令在终端可用并设置所需的环境变量。请务必将 `DOTNET_FILE` 值更改为下载的二进制文件的名称：

```
DOTNET_FILE=dotnet-sdk-5.0.302-osx-x64.tar.gz
export DOTNET_ROOT=$(pwd)/dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT
```

TIP

前面的 `export` 命令只会使 .NET CLI 命令对运行它的终端会话可用。

你可以编辑 shell 配置文件，永久地添加这些命令。Linux 提供了许多不同的 shell，每个都有不同的配置文件。例如：

- **Bash Shell**：~/.bash_profile、~/.bashrc
- **Korn Shell**：~/.kshrc 或 .profile
- **Z Shell**：~/.zshrc 或 .zprofile

为 shell 编辑相应的源文件，并将 `:$HOME/dotnet` 添加到现有 `PATH` 语句的末尾。如果不包含 `PATH` 语句，则使用 `export PATH=$PATH:$HOME/dotnet` 添加新行。

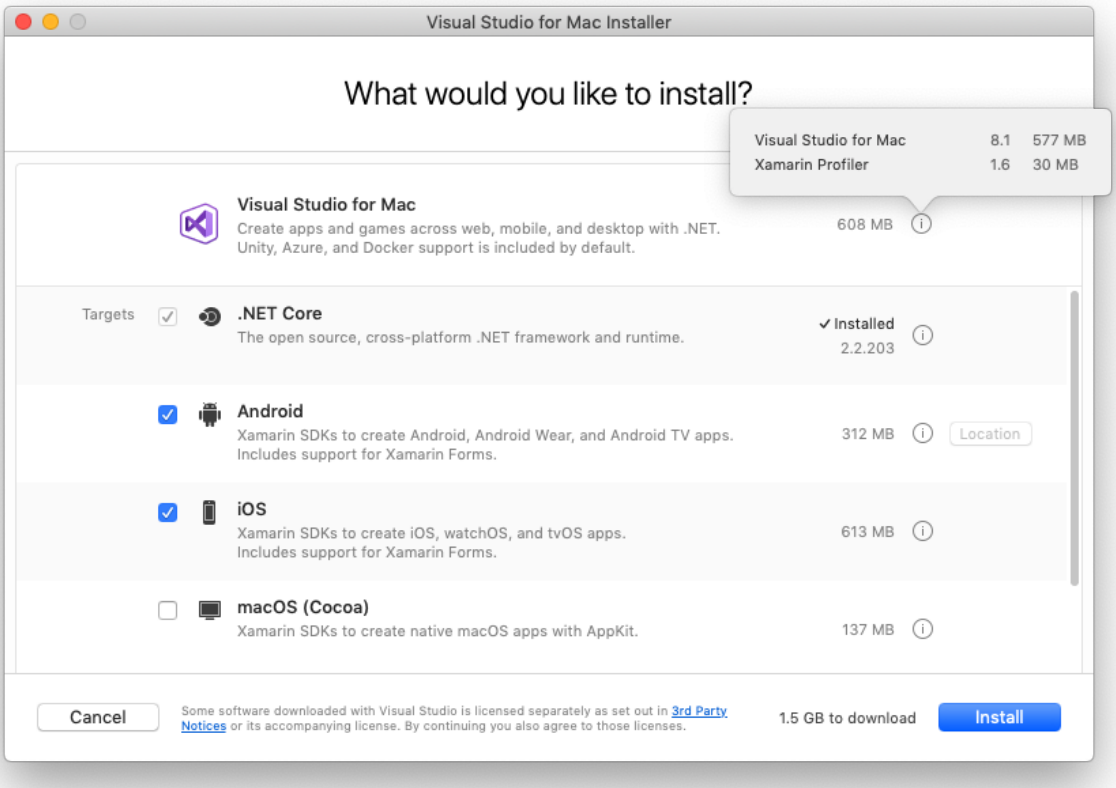
另外，将 `export DOTNET_ROOT=$HOME/dotnet` 添加至文件的末尾。

使用此方法可以将不同的版本安装到不同的位置，并明确选择应用程序要使用的对应版本。

使用 Visual Studio for Mac 安装

选定 .NET 工作负载后，可使用 Visual Studio for Mac 安装 .NET SDK。若要开始在 macOS 上进行 .NET 开发，请参阅安装 Visual Studio 2019 for Mac。

.NET SDK	VISUAL STUDIO
5.0	Visual Studio 2019 for Mac 版本 8.8 或更高版本。
3.1	Visual Studio 2019 for Mac 版本 8.4 或更高版本。
2.1	Visual Studio 2019 for Mac 版本 8.0 或更高版本。



随 Visual Studio Code 一起安装

Visual Studio Code 是一个功能强大的轻量级源代码编辑器，可在桌面上运行。Visual Studio Code 适用于 Windows、macOS 和 Linux。

虽然 Visual Studio Code 不像 Visual Studio 一样附带自动的 .NET 安装程序，但添加 .NET 支持非常简单。

1. 下载并安装 Visual Studio Code。
2. 下载并安装 .NET SDK。
3. 从 Visual Studio Code 市场安装 C# 扩展。

使用 Bash 自动化安装

dotnet-install 脚本用于运行时的自动化和非管理员安装。可从 dotnet-install 脚本引用页下载该脚本。

此脚本默认安装最新的长期支持 (LTS) 版本，即 .NET Core 3.1。可通过指定 `current` 开关以选择特定版本。包括 `runtime` 开关以安装运行时。否则，该脚本安装 SDK。

```
./dotnet-install.sh --channel 5.0 --runtime aspnetcore
```

> **NOTE**
> 可以使用前面的命令安装 ASP.NET Core 运行时，以实现最大的兼容性。ASP.NET Core 运行时还包括标准 .NET 运行时。

Docker

容器提供了一种将应用程序与主机系统的其余部分隔离的轻量级方法。同一计算机上的容器只共享内核，并使用为应用程序提供的资源。

.NET 可在 Docker 容器中运行。官方 .NET Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可在 Microsoft.NET Core Docker Hub 存储库中找到这些映像。每个存储库包含 .NET（SDK 或运行时）和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，ASP.NET Core 存储库提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

有关在 Docker 容器中使用 .NET Core 的详细信息，请参阅 .NET 和 Docker 简介和示例。

后续步骤

- 如何检查是否已安装 .NET Core。
- 处理 macOS Catalina 公证。
- 教程：开始使用 macOS。
- 教程：使用 Visual Studio Code 创建一个新应用。
- 教程：使 .NET Core 应用容器化。

在 Linux 上安装 .NET

2021/11/16 •

.NET 在不同的 Linux 发行版上可用。大多数 Linux 平台和发行版每年都有一个主要版本，并提供用于安装 .NET 的包管理器。本文介绍当前支持的版本以及使用的包管理器。

本文其余部分详细介绍了 .NET 支持的每个主要 Linux 发行版。所有 .NET 版本在 .NET Core 版本达到支持终止日期或 Linux 发行版达到生命周期之前仍受支持。

为了实现最佳兼容性，请选择长期支持版本 (LTS)。

不支持的版本

以下 .NET 版本 ✕ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

以下部分未详细介绍这些不受支持的版本，如果你尝试安装它们，则实际数据可能有所不同。

手动安装

如果你不想在 Linux 上使用包管理器来安装 .NET，可以通过下列方法之一来安装 .NET：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

请务必查看相应的发行页，以详细了解任何可能会在手动安装时缺失的必需依赖项。

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可以手动安装 .NET 的预览版和候选发布版本。

Alpine

下表列出了当前支持的 .NET 版本以及支持它们的 Alpine 版本。这些版本在 .NET 到达支持终止日期或 Alpine 的版本到达有效期之前仍受支持。

- ✔ 指示 Alpine 或 .NET 版本仍受支持。
- ✕ 指示 Alpine 或 .NET 版本在该 Alpine 发行版本上不受支持。
- 当 Alpine 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

ALPINE	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 3.14	✕ 2.1	✔ 3.1	✔ 5.0
✔ 3.13	✕ 2.1	✔ 3.1	✔ 5.0

ALPINE	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 3.12	✘ 2.1	✔ 3.1	✔ 5.0
✔ 3.11	✘ 2.1	✔ 3.1	✔ 5.0
✘ 3.10	✘ 2.1	✔ 3.1	✘ 5.0
✘ 3.9	✘ 2.1	✔ 3.1	✘ 5.0
✘ 3.8	✘ 2.1	✔ 3.1	✘ 5.0

有关详细信息，请参阅在 Alpine 上安装 .NET。

CentOS

CentOS 7 使用 Yum 作为包管理器，CentOS 8 使用 DNF。

下表列出了 CentOS 7 和 CentOS 8 上当前受支持的 .NET 版本。这些版本在 .NET 版本达到支持终止日期或 CentOS 版本不再受支持之前仍受支持。

CENTOS	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 8	✘ 2.1	✔ 3.1	✔ 5.0
✔ 7	✘ 2.1	✔ 3.1	✔ 5.0

有关详细信息，请参阅在 CentOS 上安装 .NET。

Debian

Debian 将 APT(高级包工具)用作包管理器。

下表列出了当前支持的 .NET 版本以及支持它们的 Debian 版本。这些版本在 .NET 版本达到支持终止日期或 Debian 的版本达到生命周期之前仍受支持。

- ✔ 指示 Debian 或 .NET 版本仍受支持。
- ✘ 指示 Debian 或 .NET 版本在该 Debian 版本上不受支持。
- 当 Debian 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

DEBIAN	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 11	✘ 2.1	✔ 3.1	✔ 5.0
✔ 10	✘ 2.1	✔ 3.1	✔ 5.0
✔ 9	✘ 2.1	✔ 3.1	✔ 5.0
✘ 8	✘ 2.1	✘ 3.1	✘ 5.0

有关详细信息，请参阅在 Debian 上安装 .NET。

Fedora

Fedora 将 DNF 用作其包管理器。

下表列出了当前支持的 .NET 版本以及支持它们的 Fedora 版本。这些版本在 .NET 版本达到支持终止日期或 Fedora 版本达到生命周期之前仍受支持。

- ✔ 指示 Fedora 或 .NET 版本仍受支持。
- ✘ 指示 Fedora 或 .NET 版本在该 Fedora 版本上不受支持。
- 当 Fedora 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

FEDORA	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 34	✘ 2.1	✔ 3.1	✔ 5.0
✔ 33	✘ 2.1	✔ 3.1	✔ 5.0
✘ 32	✘ 2.1	✔ 3.1	✔ 5.0
✘ 31	✘ 2.1	✔ 3.1	✘ 5.0
✘ 30	✘ 2.1	✔ 3.1	✘ 5.0
✘ 29	✘ 2.1	✔ 3.1	✘ 5.0
✘ 28	✘ 2.1	✘ 3.1	✘ 5.0
✘ 27	✘ 2.1	✘ 3.1	✘ 5.0

有关详细信息，请参阅在 Fedora 上安装 .NET。

openSUSE

openSUSE 将 zypper 用作包管理器。

下表列出了 openSUSE 15 上当前受支持的 .NET 版本。这些版本在 .NET 版本达到支持终止日期或 openSUSE 版本不再受支持之前仍受支持。

OPENSUSE	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 15	✘ 2.1	✔ 3.1	✔ 5.0

有关详细信息，请参阅在 openSUSE 上安装 .NET。

Red Hat

Red Hat Enterprise Linux (RHEL) 将 yum (RHEL 7) 和 DNF (RHEL 8) 用作包管理器。

下表列出了 RHEL 7 和 RHEL 8 上当前受支持的 .NET 版本。这些版本在 .NET 达到支持终止日期或 RHEL 版本不再受到支持之前仍受支持。

- ✔ 指示 RHEL 或 .NET 版本仍受支持。
- ✘ 指示 RHEL 或 .NET 版本在该 RHEL 版本上不受支持。
- 当 RHEL 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

RHEL	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 8	✘ 2.1	✔ 3.1	✔ 5.0
✔ 7	✘ 2.1	✔ 3.1	✔ 5.0

有关详细信息，请参阅在 RHEL 上安装 .NET。

SLES

SLES 将 zypper 用作包管理器。

下表列出了 SLES 12 SP2 和 SLES 15 上当前受支持的 .NET 版本。这些版本在 .NET 达到支持终止日期或 SLES 版本不再受到支持之前仍受支持。

- ✔ 指示 SLES 或 .NET 版本仍受支持。
- ✘ 指示 SLES 或 .NET 版本在该 SLES 版本上不受支持。
- 当 SLES 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

SLES	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 15	✘ 2.1	✔ 3.1	✔ 5.0
✔ 12 SP2	✘ 2.1	✔ 3.1	✔ 5.0

有关详细信息，请参阅在 SLES 上安装 .NET。

Ubuntu

Ubuntu 将 APT(高级包工具)用作包管理器。

下表表示 Ubuntu 和 .NET 的支持状态。

- ✔ 指示 Ubuntu 或 .NET 版本仍受支持。
- ✘ 指示 Ubuntu 或 .NET 版本在该 Ubuntu 版本上不受支持。
- 当 Ubuntu 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

UBUNTU	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 21.04	✘ 2.1	✔ 3.1	✔ 5.0
✘ 20.10	✘ 2.1	✔ 3.1	✔ 5.0
✔ 20.04 (LTS)	✘ 2.1	✔ 3.1	✔ 5.0
✘ 19.10	✘ 2.1	✔ 3.1	✔ 5.0
✘ 19.04	✘ 2.1	✔ 3.1	✘ 5.0
✘ 18.10	✘ 2.1	✘ 3.1	✘ 5.0
✔ 18.04 (LTS)	✘ 2.1	✔ 3.1	✔ 5.0

UBUNTU	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✕ 17.10	✕ 2.1	✕ 3.1	✕ 5.0
✕ 17.04	✕ 2.1	✕ 3.1	✕ 5.0
✕ 16.10	✕ 2.1	✕ 3.1	✕ 5.0
✔ 16.04 (LTS)	✕ 2.1	✔ 3.1	✔ 5.0

有关详细信息，请参阅在 Ubuntu 上安装 .NET。

后续步骤

- 如何检查是否已安装 .NET。
- 教程：使用 Visual Studio Code 创建一个新应用。
- 教程：使 .NET 应用容器化。

在 Ubuntu 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

Ubuntu 支持 .NET。本文介绍如何在 Ubuntu 上安装 .NET。如果 Ubuntu 版本不受支持，则该版本不再支持 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

支持的分发

下表列出了当前支持的 .NET 版本以及支持它们的 Ubuntu 版本。这些版本在 .NET 版本达到支持终止日期或 Ubuntu 的版本达到生命周期之前仍受支持。

- ✔ 指示 Ubuntu 或 .NET 版本仍受支持。
- ✘ 指示 Ubuntu 或 .NET 版本在该 Ubuntu 版本上不受支持。
- 当 Ubuntu 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

UBUNTU	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 21.04	✘ 2.1	✔ 3.1	✔ 5.0
✘ 20.10	✘ 2.1	✔ 3.1	✔ 5.0
✔ 20.04 (LTS)	✘ 2.1	✔ 3.1	✔ 5.0
✘ 19.10	✘ 2.1	✔ 3.1	✔ 5.0
✘ 19.04	✘ 2.1	✔ 3.1	✘ 5.0
✘ 18.10	✘ 2.1	✘ 3.1	✘ 5.0
✔ 18.04 (LTS)	✘ 2.1	✔ 3.1	✔ 5.0
✘ 17.10	✘ 2.1	✘ 3.1	✘ 5.0

UBUNTU	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✕ 17.04	✕ 2.1	✕ 3.1	✕ 5.0
✕ 16.10	✕ 2.1	✕ 3.1	✕ 5.0
✔ 16.04 (LTS)	✕ 2.1	✔ 3.1	✔ 5.0

以下 .NET 版本 ✕ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

21.04 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/21.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

20.10 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/20.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运

行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

20.04 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

19.10 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/19.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-3.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-3.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-3.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-3.1` 替换为 `dotnet-runtime-3.1`。

```
sudo apt-get install -y dotnet-runtime-3.1
```

19.04 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/19.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-3.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-3.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-3.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-3.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-3.1` 替换为 `dotnet-runtime-3.1`。

```
sudo apt-get install -y dotnet-runtime-3.1
```

18.10 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/18.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-2.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-2.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-2.1` 替换为 `dotnet-runtime-2.1`。

```
sudo apt-get install -y dotnet-runtime-2.1
```

18.04 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/18.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

17.10 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/17.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-2.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-2.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-2.1` 替换为 `dotnet-runtime-2.1`。

```
sudo apt-get install -y dotnet-runtime-2.1
```

17.04 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/17.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-2.1
```

IMPORTANT

如果收到类似于"找不到包 dotnet-sdk-2.1"的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-2.1
```

IMPORTANT

如果收到类似于"找不到包 aspnetcore-runtime-2.1"的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-2.1` 替换为 `dotnet-runtime-2.1`。

```
sudo apt-get install -y dotnet-runtime-2.1
```

16.10 ✕

✕ 请注意，此版本的 Ubuntu 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/16.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-2.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-2.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-2.1` 替换为 `dotnet-runtime-2.1`。

```
sudo apt-get install -y dotnet-runtime-2.1
```

16.04 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/ubuntu/16.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

IMPORTANT

如果收到类似于"找不到包 aspnetcore-runtime-5.0"的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：
  - SDK
  - Runtime — 运行时
- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
  - 5.0
  - 3.1
  - 3.0
  - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依

赖项和要求。

使用 APT 更新 .NET

当新的修补程序版本适用于 .NET 时，只需使用以下命令通过 APT 进行升级：

```
sudo apt-get update
sudo apt-get upgrade
```

如果安装 .NET 后已升级过 Linux 分发版，可能需要重新配置 Microsoft 包存储库。运行当前分发版本的安装说明，升级到相应的包存储库以进行 .NET 更新。

APT 疑难解答

本部分提供有关使用 APT 安装 .NET 时可能会遇到的常见错误的信息。

找不到包

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

找不到 \ 无法安装某些包

如果收到类似于“找不到包 {dotnet-package}”或“无法安装某些包”的错误消息，请运行以下命令 。

以下命令组中有两个占位符。

- `{dotnet-package}`
  此项表示要安装的 .NET 包，如 `aspnetcore-runtime-3.1`。它在以下 `sudo apt-get install` 命令中使用。
- `{os-version}`
  这表示你正在使用的发行版。此项在以下 `wget` 命令中使用。发行版是数值，如 Ubuntu 上的 `20.04` 或 Debian 上的 `10`。

首先，尝试清除包列表：

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

然后，再次尝试安装 .NET。如果这不起作用，可使用以下命令运行手动安装：

```
sudo apt-get install -y gpg
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/ubuntu/{os-version}/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

未能提取

安装 .NET 包时，可能会看到类似于 `Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 https://github.com/dotnet/core/issues 中提交问题。

依赖项

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET 或发布自包含的应用，则需要确保已安装以下库：

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu52（针对 14.x）
- libicu55（针对 16.x）
- libicu60（针对 18.x）
- libicu66（适用于 20. x）
- libssl1.0.0（适用于 14.x、16.x）
- libssl1.1（适用于18.x、20.x）
- libstdc++6
- zlib1g

对于使用 System.Drawing.Common 程序集的 .NET 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 Alpine 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

本文介绍如何在 Alpine 上安装 .NET。如果 Alpine 版本不再受到支持，则该版本不再支持 .NET。不过，可以按照这些说明在这些版本上运行 .NET，即使它不受支持。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

安装

安装程序不可用于 Alpine Linux。必须使用以下方式之一安装 .NET：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

支持的发行版

下表列出了当前支持的 .NET 版本以及支持它们的 Alpine 版本。这些版本在 .NET 到达支持终止日期或 Alpine 的版本到达有效期之前仍受支持。

- ✔ 指示 Alpine 或 .NET 版本仍受支持。
- ✘ 指示 Alpine 或 .NET 版本在该 Alpine 发行版本上不受支持。
- 当 Alpine 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

ALPINE	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 3.14	✘ 2.1	✔ 3.1	✔ 5.0
✔ 3.13	✘ 2.1	✔ 3.1	✔ 5.0
✔ 3.12	✘ 2.1	✔ 3.1	✔ 5.0
✔ 3.11	✘ 2.1	✔ 3.1	✔ 5.0
✘ 3.10	✘ 2.1	✔ 3.1	✘ 5.0
✘ 3.9	✘ 2.1	✔ 3.1	✘ 5.0
✘ 3.8	✘ 2.1	✔ 3.1	✘ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

依赖项

Alpine Linux 上的 .NET 要求安装以下依赖项：

- icu-libs
- krb5-libs
- libgcc
- libgdiplus（.NET 应用需要 System.Drawing.Common 程序集时）
- libintl
- libssl1.1（Alpine v3.9 或更高版本）
- libssl1.0（Alpine v3.8 或更低版本）
- libstdc++
- zlib

若要安装必需项，请运行以下命令：

```
apk add bash icu-libs krb5-libs libgcc libintl libssl1.1 libstdc++ zlib
```

若要安装 libgdiplus，可能需要指定一个存储库：

```
apk add libgdiplus --repository https://dl-3.alpinelinux.org/alpine/edge/testing/
```

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 CentOS 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

CentOS 支持 .NET。本文介绍如何在 CentOS 上安装 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

支持的分发

下表列出了 CentOS 7 和 CentOS 8 上当前受支持的 .NET 版本。这些版本在 .NET 版本达到支持终止日期或 CentOS 版本不再受支持之前仍受支持。

- ✔ 指示 CentOS 或 .NET 版本仍受支持。
- ✘ 指示 CentOS 或 .NET 版本在该 CentOS 版本上不受支持。
- 当 CentOS 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

CENTOS	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 8	✘ 2.1	✔ 3.1	✔ 5.0
✔ 7	✘ 2.1	✔ 3.1	✔ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

CentOS 8 ✔

.NET 5.0 在 CentOS 8 的默认包存储库中提供。

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo dnf install dotnet-sdk-5.0
```

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo dnf install aspnetcore-runtime-5.0
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo dnf install dotnet-runtime-5.0
```

CentOS 7 ✔

安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加 Microsoft 包存储库。打开终端并运行以下命令：

```
sudo rpm -Uvh https://packages.microsoft.com/config/centos/7/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
sudo yum install dotnet-sdk-5.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
sudo yum install aspnetcore-runtime-5.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-5.0` in the previous command with `dotnet-runtime-5.0`:

```
sudo yum install dotnet-runtime-5.0
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：
  - SDK
  - Runtime — 运行时
- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
  - 5.0
  - 3.1
  - 3.0
  - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依

赖项和要求。

包管理器疑难解答

本部分提供有关使用包管理器安装 .NET 时可能会遇到的常见错误的信息。

找不到包

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

未能提取

安装 .NET 包时，可能会看到类似于 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 https://github.com/dotnet/core/issues 中提交问题。

与缺少 `fxr`、`libhostfxr.so` 或 `FrameworkList.xml` 相关的错误

有关如何解决这些问题的详细信息，请参阅排查 `fxr`、`libhostfxr.so` 和 `FrameworkList.xml` 错误。

依赖关系

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET Core 或发布自包含的应用，则需要确保已安装以下库：

- krb5-libs
- libicu
- openssl-libs
- zlib

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

有关依赖项的详细信息，请参阅独立式 Linux 应用。

对于使用 System.Drawing.Common 程序集的 .NET Core 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 Debian 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

本文介绍如何在 Debian 上安装 .NET。如果 Debian 版本不受支持，则该版本不再支持 .NET。不过，可以按照这些说明在这些版本上运行 .NET，即使它不受支持。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

支持的分发

下表列出了当前支持的 .NET 版本以及支持它们的 Debian 版本。这些版本在 .NET 版本达到支持终止日期或 Debian 的版本达到生命周期之前仍受支持。

- ✔ 指示 Debian 或 .NET 版本仍受支持。
- ✘ 指示 Debian 或 .NET 版本在该 Debian 版本上不受支持。
- 当 Debian 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

DEBIAN	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 11	✘ 2.1	✔ 3.1	✔ 5.0
✔ 10	✘ 2.1	✔ 3.1	✔ 5.0
✔ 9	✘ 2.1	✔ 3.1	✔ 5.0
✘ 8	✘ 2.1	✘ 3.1	✘ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

Debian 11 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/debian/11/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中

的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

Debian 10 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget https://packages.microsoft.com/config/debian/10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

Debian 9 ✔

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到

受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/9/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-5.0”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-5.0
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-5.0”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo apt-get install -y dotnet-runtime-5.0
```

Debian 8 ✕

✕ 请注意，此版本的 Debian 不再受支持。

使用 APT 进行安装可通过几个命令来完成。安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加包存储库。

打开终端并运行以下命令：

```
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/8/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y dotnet-sdk-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 dotnet-sdk-2.1”的错误消息，请参阅 APT 疑难解答部分。

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y aspnetcore-runtime-2.1
```

> **IMPORTANT**
> 如果收到类似于“找不到包 aspnetcore-runtime-2.1”的错误消息，请参阅 APT 疑难解答部分。

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-2.1` 替换为 `dotnet-runtime-2.1`。

```
sudo apt-get install -y dotnet-runtime-2.1
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：

    - SDK
    - Runtime — 运行时
- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
    - 5.0
    - 3.1
    - 3.0
    - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依赖项和要求。

使用 APT 更新 .NET

当新的修补程序版本适用于 .NET 时，只需使用以下命令通过 APT 进行升级：

```
sudo apt-get update
sudo apt-get upgrade
```

如果安装 .NET 后已升级过 Linux 分发版，可能需要重新配置 Microsoft 包存储库。运行当前分发版本的安装说明，升级到相应的包存储库以进行 .NET 更新。

APT 疑难解答

本部分提供有关使用 APT 安装 .NET 时可能会遇到的常见错误的信息。

找不到包

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

找不到 \ 无法安装某些包

如果收到类似于“找不到包 {dotnet-package}”或“无法安装某些包”的错误消息，请运行以下命令 。

以下命令组中有两个占位符。

- `{dotnet-package}`
  此项表示要安装的 .NET 包，如 `aspnetcore-runtime-3.1`。它在以下 `sudo apt-get install` 命令中使用。
- `{os-version}`
  这表示你正在使用的发行版。此项在以下 `wget` 命令中使用。发行版是数值，如 Ubuntu 上的 `20.04` 或 Debian 上的 `10`。

首先，尝试清除包列表：

```
sudo dpkg --purge packages-microsoft-prod && sudo dpkg -i packages-microsoft-prod.deb
sudo apt-get update
```

然后，再次尝试安装 .NET。如果这不起作用，可使用以下命令运行手动安装：

```
sudo apt-get install -y gpg
wget -O - https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor -o microsoft.asc.gpg
sudo mv microsoft.asc.gpg /etc/apt/trusted.gpg.d/
wget https://packages.microsoft.com/config/debian/{os-version}/prod.list
sudo mv prod.list /etc/apt/sources.list.d/microsoft-prod.list
sudo chown root:root /etc/apt/trusted.gpg.d/microsoft.asc.gpg
sudo chown root:root /etc/apt/sources.list.d/microsoft-prod.list
sudo apt-get update; \
  sudo apt-get install -y apt-transport-https && \
  sudo apt-get update && \
  sudo apt-get install -y {dotnet-package}
```

未能提取

安装 .NET 包时，可能会看到类似于 `Failed to fetch ... File has unexpected size ... Mirror sync in progress?` 的错误。此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 30 分钟。如果持续收到此错误超过 30 分钟，请在 https://github.com/dotnet/core/issues 中提交问题。

依赖项

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET Core 或发布自包含的应用，则需要确保已安装以下库：

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu52（适用于 8.x）
- libicu57（适用于9.x）
- libicu63（适用于10.x）
- libicu67（适用于11.x）
- libssl1.0.0（适用于8.x）
- libssl1.1（适用于9.x-11.x）
- libstdc++6
- zlib1g

对于使用 System.Drawing.Common 程序集的 .NET Core 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
>
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 Fedora 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

.NET 在 Fedora 上受支持，本文就将介绍如何在 Fedora 上安装 .NET。如果 Fedora 版本不受支持，则该版本不再支持 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 通过 Snap 安装 .NET SDK 或 .NET Runtime。
- 使用脚本安装 .NET SDK 或 .NET Runtime。
- 手动安装 .NET SDK 或 .NET Runtime。

安装 .NET 5.0

Fedora 的默认包存储库中提供的 .NET 最新版是 .NET 5.0。

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo dnf install dotnet-sdk-5.0
```

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo dnf install aspnetcore-runtime-5.0
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo dnf install dotnet-runtime-5.0
```

安装 .NET Core 3.1

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo dnf install dotnet-sdk-3.1
```

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo dnf install aspnetcore-runtime-3.1
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-3.1` 替换为 `dotnet-runtime-3.1`。

```
sudo dnf install dotnet-runtime-3.1
```

支持的发行版

下表列出了当前支持的 .NET 版本以及支持它们的 Fedora 版本。这些版本在 .NET 版本达到支持终止日期或 Fedora 版本达到生命周期之前仍受支持。

- ✔ 指示 Fedora 或 .NET 版本仍受支持。
- ✘ 指示 Fedora 或 .NET 版本在该 Fedora 版本上不受支持。
- 当 Fedora 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

.NET ⇊	FEDORA 34 ✔	33 ✔	32 ✘	31 ✘	30 ✘	29 ✘	28 ✘	27 ✘
.NET 5	✔	✔	✔	✘	✘	✘	✘	✘
.NET Core 3.1	✔	✔	✔	✔	✔	✔	✘	✘
.NET Core 2.1	✘	✘	✘	✘	✘	✘	✘	✘

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

依赖项

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET Core 或发布自包含的应用，则需要确保已安装以下库：

- krb5-libs
- libicu
- openssl-libs
- zlib

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

有关依赖项的详细信息，请参阅独立式 Linux 应用。

对于使用 System.Drawing.Common 程序集的 .NET Core 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

在早期的发行版上进行安装

Fedora 的早期版本的默认包存储库中并不包含 .NET Core。可使用 snap 通过 dotnet-install.sh 脚本安装 .NET，或使用 Microsoft 的存储库安装 .NET：

1. 首先，将 Microsoft 签名密钥添加到受信任的密钥列表。

```
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
```

2. 接下来，添加 Microsoft 包存储库。存储库的源基于你的 Fedora 版本。

FEDORA 版	存储库
31	`https://packages.microsoft.com/config/fedora/31/prod.repo`
30	`https://packages.microsoft.com/config/fedora/30/prod.repo`
29	`https://packages.microsoft.com/config/fedora/29/prod.repo`
28	`https://packages.microsoft.com/config/fedora/28/prod.repo`
27	`https://packages.microsoft.com/config/fedora/27/prod.repo`

```
sudo wget -O /etc/yum.repos.d/microsoft-prod.repo
https://packages.microsoft.com/config/fedora/31/prod.repo
```

安装 SDK

.NET Core SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET Core SDK，则无需安装相应的运行时。若要安装 .NET Core SDK，请运行以下命令：

```
sudo dnf install dotnet-sdk-3.1
```

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
sudo dnf install aspnetcore-runtime-3.1
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上一命令中的 `aspnetcore-runtime-3.1` 替换为 `dotnet-runtime-3.1`。

```
sudo dnf install dotnet-runtime-3.1
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：
  - SDK
  - Runtime — 运行时
- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
  - 5.0
  - 3.1
  - 3.0
  - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依赖项和要求。

包管理器疑难解答

本部分提供有关使用包管理器安装 .NET 或 .NET Core 时可能会遇到的常见错误的信息。

找不到包

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 通过 Snap 安装 .NET SDK 或 .NET Runtime。
- 使用脚本安装 .NET SDK 或 .NET Runtime。
- 手动安装 .NET SDK 或 .NET Runtime。

未能提取

安装 .NET 包时，可能会看到类似于 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 https://github.com/dotnet/core/issues 中提交问题。

与缺少 `fxr`、`libhostfxr.so` 或 `FrameworkList.xml` 相关的错误

有关如何解决这些问题的详细信息，请参阅排查 `fxr`、`libhostfxr.so` 和 `FrameworkList.xml` 错误。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 openSUSE 上安装 .NET SDK 或 .NET Runtime

2021/11/16 •

openSUSE 支持 .NET。本文介绍如何在 openSUSE 上安装 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

支持的分发

下表列出了 openSUSE 15 上当前受支持的 .NET 版本。这些版本在 .NET 版本达到支持终止日期或 openSUSE 版本不再受支持之前仍受支持。

- ✔ 指示 openSUSE 或 .NET 版本仍受支持。
- ✘ 指示 openSUSE 或 .NET 版本在该 openSUSE 版本上不受支持。
- 当 openSUSE 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

OPENSUSE	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 15	✘ 2.1	✔ 3.1	✔ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

openSUSE 15 ✔

安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加 Microsoft 包存储库。打开终端并运行以下命令：

```
sudo zypper install libicu
sudo rpm --import https://packages.microsoft.com/keys/microsoft.asc
wget https://packages.microsoft.com/config/opensuse/15/prod.repo
sudo mv prod.repo /etc/zypp/repos.d/microsoft-prod.repo
sudo chown root:root /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
sudo zypper install dotnet-sdk-5.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
sudo zypper install aspnetcore-runtime-5.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-5.0` in the previous command with `dotnet-runtime-5.0`:

```
sudo zypper install dotnet-runtime-5.0
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：
  - SDK
  - Runtime — 运行时

- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
  - 5.0
  - 3.1
  - 3.0
  - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依赖项和要求。

包管理器疑难解答

本部分提供有关使用包管理器安装 .NET 时可能会遇到的常见错误的信息。

找不到包

> **IMPORTANT**
> 仅在 x64 体系结构上支持包管理器安装。对于 ARM 等其他体系结构，必须通过其他一些方式安装 .NET，例如，通过 Snap 和安装程序脚本进行安装，或通过手动提取二进制文件进行安装。

有关在不使用包管理器的情况下安装 .NET 的详细信息，请参阅以下文章之一：

- 或者，通过 Snap 安装 .NET。
- 或者，通过 `install-dotnet` 脚本安装 .NET。
- 手动安装 .NET

未能提取

安装 .NET 包时，可能会看到类似于 `signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 https://github.com/dotnet/core/issues 中提交问题。

依赖项

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET 或发布自包含的应用，则需要确保已安装以下库：

- krb5
- libicu
- libopenssl1_0_0

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

有关依赖项的详细信息，请参阅独立式 Linux 应用。

对于使用 System.Drawing.Common 程序集的 .NET 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 RHEL 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

Red Hat Enterprise Linux (RHEL) 支持 .NET。本文介绍如何在 RHEL 上安装 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

注册 Red Hat 订阅

若要在 RHEL 上从 Red Hat 安装 .NET，首先需要使用 Red Hat 订阅管理器进行注册。如果未在系统上完成此操作，或者不确定是否完成了此操作，请参阅适用于 .NET 的 Red Hat 产品文档。

支持的发行版

下表列出了 RHEL 7 和 RHEL 8 上当前受支持的 .NET 版本。这些版本在 .NET 达到支持终止日期或 RHEL 版本不再受到支持之前仍受支持。

- ✔ 指示 RHEL 或 .NET 版本仍受支持。
- ✘ 指示 RHEL 或 .NET 版本在该 RHEL 版本上不受支持。
- 当 RHEL 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

RHEL	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 8	✘ 2.1	✔ 3.1	✔ 5.0
✔ 7	✘ 2.1	✔ 3.1	✔ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参

阅如何删除 .NET 运行时和 SDK。

RHEL 8 ✔

.NET 包含在 RHEL 8 的 AppStream 存储库中。

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
sudo dnf install dotnet-sdk-5.0
```

安装运行时

通过 ASP.NET Core 运行时，可以运行使用 .NET 开发且未提供运行时的应用。以下命令将安装 ASP.NET Core 运行时，这是与 .NET 最兼容的运行时。在终端中，运行以下命令：

```
sudo dnf install aspnetcore-runtime-5.0
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上一命令中的 `aspnetcore-runtime-5.0` 替换为 `dotnet-runtime-5.0`：

```
sudo dnf install dotnet-runtime-5.0
```

RHEL 7 ✔ .NET 5.0

以下命令安装 `scl-utils` 包：

```
sudo yum install scl-utils
```

安装 SDK

.NET SDK 使你可以通过 .NET 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet50 -y
scl enable rh-dotnet50 bash
```

Red Hat 建议不要永久启用 `rh-dotnet50`，因为这可能会影响其他程序。如果要永久启用 `rh-dotnet`，请将以下行添加到 ~/.bashrc 文件中。

```
source scl_source enable rh-dotnet50
```

安装运行时

通过 .NET 运行时，可以运行使用 .NET 开发且未包含运行时的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet50-aspnetcore-runtime-5.0 -y
scl enable rh-dotnet50 bash
```

Red Hat 建议不要永久启用 `rh-dotnet50`，因为这可能会影响其他程序。如果要永久启用 `rh-dotnet50`，请将以下行添加到 ~/.bashrc 文件中。

```
source scl_source enable rh-dotnet50
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET 运行时：将上述命令中的 `rh-dotnet50-aspnetcore-runtime-5.0` 替换为 `rh-dotnet50-dotnet-runtime-5.0`。

RHEL 7 ✔ .NET Core 3.1

安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加 Microsoft 包存储库。打开终端并运行以下命令：

以下命令安装 `scl-utils` 包：

```
sudo yum install scl-utils
```

安装 SDK

.NET SDK 使你可以通过 .NET Core 开发应用。如果安装 .NET SDK，则无需安装相应的运行时。若要安装 .NET SDK，请运行以下命令：

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet31 -y
scl enable rh-dotnet31 bash
```

Red Hat 建议不要永久启用 `rh-dotnet31`，因为这可能会影响其他程序。例如，`rh-dotnet31` 包括与基本 RHEL 版本不同的 `libcurl` 版本。这可能会导致不需要不同版本的 `libcurl` 的程序出现问题。如果要永久启用 `rh-dotnet`，请将以下行添加到 ~/.bashrc 文件中。

```
source scl_source enable rh-dotnet31
```

安装运行时

.NET Core 运行时允许运行使用不随附运行时的 .NET Core 所开发的应用。以下命令安装 ASP.NET Core 运行时，这是与 .NET Core 最兼容的运行时。在终端中，运行以下命令。

```
subscription-manager repos --enable=rhel-7-server-dotnet-rpms
yum install rh-dotnet31-aspnetcore-runtime-3.1 -y
scl enable rh-dotnet31 bash
```

Red Hat 建议不要永久启用 `rh-dotnet31`，因为这可能会影响其他程序。例如，`rh-dotnet31` 包括与基本 RHEL 版本不同的 `libcurl` 版本。这可能会导致不需要不同版本的 `libcurl` 的程序出现问题。如果要永久启用 `rh-dotnet31`，请将以下行添加到 ~/.bashrc 文件中。

```
source scl_source enable rh-dotnet31
```

作为 ASP.NET Core 运行时的一种替代方法，你可以安装不包含 ASP.NET Core 支持的 .NET Core 运行时：将上述命令中的 `rh-dotnet31-aspnetcore-runtime-3.1` 替换为 `rh-dotnet31-dotnet-runtime-3.1`。

依赖关系

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET Core 或发布自包含的应用，则需要确保已安装以下库：

- krb5-libs
- libicu
- openssl-libs
- zlib

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

有关依赖项的详细信息，请参阅独立式 Linux 应用。

对于使用 System.Drawing.Common 程序集的 .NET Core 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

如何安装其他版本

有关安装其他版本的 .NET 所需的步骤，请参阅适用于 .NET 的 Red Hat 文档。

包管理器疑难解答

本部分提供有关使用包管理器安装 .NET 或 .NET Core 时可能会遇到的常见错误的信息。

与缺少 `fxr`、`libhostfxr.so` 或 `FrameworkList.xml` 相关的错误

有关如何解决这些问题的详细信息，请参阅排查 `fxr`、`libhostfxr.so` 和 `FrameworkList.xml` 错误。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

在 SLES 上安装 .NET SDK 或 .NET 运行时

2021/11/16 •

SLES 支持 .NET。本文介绍如何在 SLES 上安装 .NET。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

支持的发行版

下表列出了 SLES 12 SP2 和 SLES 15 上当前受支持的 .NET 版本。这些版本在 .NET 达到支持终止日期或 SLES 版本不再受到支持之前仍受支持。

- ✔ 指示 SLES 或 .NET 版本仍受支持。
- ✘ 指示 SLES 或 .NET 版本在该 SLES 版本上不受支持。
- 当 SLES 版本和 .NET 版本都有 ✔ 时，将支持该 OS 和 .NET 组合。

SLES	.NET CORE 2.1	.NET CORE 3.1	.NET 5
✔ 15	✘ 2.1	✔ 3.1	✔ 5.0
✔ 12 SP2	✘ 2.1	✔ 3.1	✔ 5.0

以下 .NET 版本 ✘ 不再受到支持。这些版本的下载仍保持发布状态：

- .NET Core 3.0
- .NET Core 2.2
- .NET Core 2.1
- .NET Core 2.0

安装预览版本

包管理器中未提供 .NET 的预览版和候选发布版本。可通过下面其中一种方式安装 .NET 的预览版和候选发布版本：

- Snap 包
- 使用 install-dotnet.sh 脚本安装
- 手动提取二进制文件

删除预览版本

使用包管理器管理 .NET 安装时，如果之前安装了预览版本，则可能会遇到冲突。包管理器可能会将非预览版本解释为 .NET 的较早版本。若要安装非预览版本，需要首先卸载预览版本。有关如何卸载 .NET 的详细信息，请参阅如何删除 .NET 运行时和 SDK。

SLES 15 ✔

安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加 Microsoft 包存储库。

打开终端并运行以下命令：

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/15/packages-microsoft-prod.rpm
```

目前，SLES 15 Microsoft 存储库安装包会将 microsoft-prod.repo 文件安装到错误的目录，从而导致 zypper 找不到 .NET 包。若要解决此问题，请在正确的目录中创建一个符号链接。

```
sudo ln -s /etc/yum.repos.d/microsoft-prod.repo /etc/zypp/repos.d/microsoft-prod.repo
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
sudo zypper install dotnet-sdk-5.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
sudo zypper install aspnetcore-runtime-5.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-5.0` in the previous command with `dotnet-runtime-5.0`:

```
sudo zypper install dotnet-runtime-5.0
```

SLES 12 ✔

SLES 12 系列的 .NET 需要至少为 SP2。

安装 .NET 之前，请运行以下命令，将 Microsoft 包签名密钥添加到受信任密钥列表，并添加 Microsoft 包存储库。打开终端并运行以下命令：

```
sudo rpm -Uvh https://packages.microsoft.com/config/sles/12/packages-microsoft-prod.rpm
```

Install the SDK

The .NET SDK allows you to develop apps with .NET. If you install the .NET SDK, you don't need to install the corresponding runtime. To install the .NET SDK, run the following command:

```
sudo zypper install dotnet-sdk-5.0
```

Install the runtime

The ASP.NET Core Runtime allows you to run apps that were made with .NET that didn't provide the runtime. The following command install the ASP.NET Core Runtime, which is the most compatible runtime for .NET. In your terminal, run the following command:

```
sudo zypper install aspnetcore-runtime-5.0
```

As an alternative to the ASP.NET Core Runtime, you can install the .NET Runtime, which doesn't include ASP.NET Core support: replace `aspnetcore-runtime-5.0` in the previous command with `dotnet-runtime-5.0`:

```
sudo zypper install dotnet-runtime-5.0
```

如何安装其他版本

.NET 的所有版本均可从 https://dotnet.microsoft.com/download/dotnet 下载，但需要手动安装。可尝试使用包管理器安装不同版本的 .NET。但请求的版本可能不可用。

添加到包管理器源的包以可改动的格式命名，例如：`{product}-{type}-{version}`。

- **product**
  要安装的 .NET 产品的类型。有效选项是：
  - dotnet
  - aspnetcore
- **type**
  选择 SDK 或运行时。有效选项是：
  - SDK
  - Runtime — 运行时
- **version**
  要安装的 SDK 或运行时的版本。本文始终提供最新支持的版本的说明。有效选项为任何已发布的版本，例如：
  - 5.0
  - 3.1
  - 3.0
  - 2.1

  尝试下载的 SDK/运行时可能不适用于 Linux 发行版。有关受支持的发行版列表，请参阅在 Linux 上安装 .NET。

示例

- 安装 ASP.NET Core 5.0 运行时：`aspnetcore-runtime-5.0`
- 安装 .NET Core 2.1 运行时：`dotnet-runtime-2.1`
- 安装 .NET 5 SDK：`dotnet-sdk-5.0`
- 安装 .NET Core 3.1 SDK：`dotnet-sdk-3.1`

缺少包

如果包版本组合无效，则它不可用。例如，未安装 ASP.NET Core SDK，所有 SDK 组件都包含在 .NET SDK 中。`aspnetcore-sdk-2.2` 的值不正确，应为 `dotnet-sdk-2.2`。有关 .NET 支持的 Linux 发行版的列表，请参阅 .NET 依赖项和要求。

包管理器疑难解答

本部分提供有关使用包管理器安装 .NET 时可能会遇到的常见错误的信息。

未能提取

安装 .NET 包时，可能会看到类似于

`signature verification failed for file 'repomd.xml' from repository 'packages-microsoft-com-prod'` 的错误。一般而言，此错误表示 .NET 的包源正在通过更新的包版本进行更新，应稍后重试。升级期间，包源的不可用时间不应超过 2 小时。如果持续收到此错误超过 2 小时，请在 https://github.com/dotnet/core/issues 中提交问题。

依赖项

使用包管理器进行安装时，将为你安装这些库。但是，如果手动安装 .NET 或发布自包含的应用，则需要确保已安装以下库：

- krb5
- libicu
- libopenssl1_1

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

有关依赖项的详细信息，请参阅独立式 Linux 应用。

对于使用 System.Drawing.Common 程序集的 .NET 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

通过 Snap 安装 .NET SDK 或 .NET Runtime

2021/11/16 •

使用 Snap 包安装 .NET SDK 或 .NET Runtime。对于内置于 Linux 发行版的包管理器而言，Snap 是一种很好的替代方法。本文介绍如何通过 Snap 安装 .NET。

Snap 是应用及其依赖项的捆绑包，无需修改即可在多个不同的 Linux 发行版中正常运行。可以从 Snap Store 中发现和安装 Snap。若要详细了解 Snap，请参阅开始使用 Snap。

Caution

Windows 10 上的 WSL2 不支持 Snap 包。作为替代方法，可使用 `dotnet-install` 脚本或特定 WSL2 发行版的包管理器。虽然可以尝试使用 Snapcraft 论坛中的替代方法启用 Snap，但该方法不受支持，因此不建议这样做。

.NET 版本

只有 ✔ 受支持的 .NET SDK 版本，才能通过 Snap 获取。从版本 2.1 开始，所有.NET Runtime 版本均可通过 Snap 获取。下表列出了 .NET（和 .NET Core）版本：

✔		✕	
5.0		3.0	
3.1 (LTS)		2.2	
		2.1	
		2.0	
		1.1	
		1.0	

有关 .NET 版本的生命周期的详细信息，请参阅 .NET Core 和 .NET 5 支持策略。

SDK 或 Runtime

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

安装 SDK

适用于 .NET SDK 的 Snap 包都是在同一标识符（即 `dotnet-sdk`）下发布的。可以通过指定通道来安装特定版本的 SDK。SDK 包括相应的运行时。下表列出了通道：

.NET		SNAP	
5.0		`5.0` 或 `latest/stable`	

.NET 版本	SNAP 通道
3.1 (LTS)	`3.1` 或 `lts/stable`

若要安装适用于 .NET SDK 的 Snap 包，请运行 `snap install` 命令。使用 `--channel` 参数来指明要安装哪个版本。如果省略此参数，则使用 `latest/stable`。在下面的示例中，指定的是 `5.0`：

```
sudo snap install dotnet-sdk --classic --channel=5.0
```

接下来，使用 `snap alias` 命令为系统注册 `dotnet` 命令：

```
sudo snap alias dotnet-sdk.dotnet dotnet
```

此命令的格式为 `sudo snap alias {package}.{command} {alias}`。可以选择所需的任何 `{alias}` 名称。例如，可以根据 Snap 安装的特定版本来命名此命令：`sudo snap alias dotnet-sdk.dotnet dotnet50`。运行命令 `dotnet50` 时，将调用这一特定版本的 .NET。但选择其他别名与大多数教程和示例不兼容，因为它们需要使用 `dotnet` 命令。

安装运行时

适用于 .NET Runtime 的 Snap 包都是在各自的包标识符下发布的。下表列出了这些包标识符：

.NET 版本	SNAP 包
5.0	`dotnet-runtime-50`
3.1 (LTS)	`dotnet-runtime-31`
3.0	`dotnet-runtime-30`
2.2	`dotnet-runtime-22`
2.1	`dotnet-runtime-21`

若要安装适用于 .NET Runtime 的 Snap 包，请运行 `snap install` 命令。在下面的示例中，安装的是 .NET 5：

```
sudo snap install dotnet-runtime-50 --classic
```

接下来，使用 `snap alias` 命令为系统注册 `dotnet` 命令：

```
sudo snap alias dotnet-runtime-50.dotnet dotnet
```

此命令的格式为 `sudo snap alias {package}.{command} {alias}`。可以选择所需的任何 `{alias}` 名称。例如，可以根据 Snap 安装的特定版本来命名此命令：`sudo snap alias dotnet-runtime-50.dotnet dotnet50`。运行命令 `dotnet50` 时，将调用特定版本的 .NET。但选择其他别名与大多数教程和示例不兼容，因为它们需要使用 `dotnet` 命令。

导出安装位置

`DOTNET_ROOT` 环境变量经常被工具用来确定 .NET 的安装位置。通过 Snap 安装 .NET 时，不配置此环境变量。应

在配置文件中配置 DOTNET_ROOT 环境变量。Snap 的路径采用以下格式：`/snap/{package}/current`。例如，如果你安装了 `dotnet-sdk` Snap，则使用以下命令将环境变量设置为 .NET 所在的位置：

```
export DOTNET_ROOT=/snap/dotnet-sdk/current
```

> **TIP**
> 前面的 `export` 命令只为运行它的终端会话设置环境变量。
>
> 你可以编辑 shell 配置文件，永久地添加这些命令。Linux 提供了许多不同的 shell，每个都有不同的配置文件。例如：
>
> - **Bash Shell**：~/.bash_profile、~/.bashrc
> - **Korn Shell**：~/.kshrc 或 .profile
> - **Z Shell**：~/.zshrc 或 .zprofile
>
> 为 shell 编辑相应的源文件并添加 `export DOTNET_ROOT=/snap/dotnet-sdk/current`。

TLS/SSL 证书错误

通过 Snap 安装 .NET 后，可能会在某些发行版上找不到 .NET TLS/SSL 证书，并且可能会在 `restore` 期间看到以下错误：

```
Processing post-creation actions...
Running 'dotnet restore' on /home/myhome/test/test.csproj...
  Restoring packages for /home/myhome/test/test.csproj...
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error : Unable to load the service index for source
https://api.nuget.org/v3/index.json. [/home/myhome/test/test.csproj]
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error :   The SSL connection could not be established,
see inner exception. [/home/myhome/test/test.csproj]
/snap/dotnet-sdk/27/sdk/2.2.103/NuGet.targets(114,5): error :   The remote certificate is invalid according
to the validation procedure. [/home/myhome/test/test.csproj]
```

若要解决此问题，请设置一些环境变量：

```
export SSL_CERT_FILE=[path-to-certificate-file]
export SSL_CERT_DIR=/dev/null
```

证书位置因发行版而异。下面是我们在发行版中遇到此问题的位置。

"	"
Fedora	`/etc/pki/ca-trust/extracted/pem/tls-ca-bundle.pem`
OpenSUSE	`/etc/ssl/ca-bundle.pem`
Solus	`/etc/ssl/certs/ca-certificates.crt`

解析 dotnet 时的问题

其他应用（例如适用于 Visual Studio Code 的 OmniSharp 扩展）通常会尝试解析 .NET SDK 的位置。一般情况下，这是通过确定可执行文件 `dotnet` 所在的位置完成的。Snap 安装的 .NET SDK 可能会混淆这些应用。当这些应用无法解析 .NET SDK 时，你将看到类似于以下消息之一的错误：

- 找不到指定的 SDK 'Microsoft.NET.Sdk'

- 找不到指定的 SDK 'Microsoft.NET.Sdk.Web'
- 找不到指定的 SDK 'Microsoft.NET.Sdk.Razor'

若要解决此问题，请将 Snap 可执行文件 `dotnet` 符号链接到程序正在查找的位置。`dotnet` 命令要查找的两个常见路径是 `/usr/local/bin/dotnet` 和 `/usr/share/dotnet`。例如，若要链接当前的 .NET SDK Snap 包，请使用以下命令：

```
ln -s /snap/dotnet-sdk/current/dotnet /usr/local/bin/dotnet
```

还可以查看这些 GitHub 问题，了解有关这些问题的信息：

- SDK 解析程序不能与 Linux 上的 SDK Snap 安装一起运行
- 找不到任何已安装的 .NET SDK

dotnet 别名

如果为 Snap 安装的 .NET 创建了别名 `dotnet`，则可能会发生冲突。使用 `snap unalias dotnet` 命令将其删除，然后根据需要添加其他别名。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

手动安装 .NET SDK 或 .NET Runtime

2021/11/16 •

.NET 在 Linux 上受支持，本文就将介绍如何使用安装脚本或通过提取二进制文件在 Linux 上安装 .NET。有关支持内置包管理器的发行版列表，请参阅在 Linux 上安装 .NET。

还可通过 Snap 安装 .NET。有关详细信息，请参阅通过 Snap 安装 .NET SDK 或 .NET Runtime。

如果要开发 .NET 应用，请安装 SDK（包括运行时）。或者，如果只需运行应用程序，请安装运行时。如果要安装该运行时，建议安装 ASP.NET Core 运行时，因为它同时包括 .NET 和 ASP.NET Core 运行时。

如果已安装 SDK 或运行时，请使用 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令查看安装了哪些版本。有关详细信息，请参阅如何检查是否已安装 .NET。

.NET 版本

下表列出了 .NET（和 .NET Core）版本：

✔ 𝍖	✕ 𝍖
5.0	3.0
3.1 (LTS)	2.2
	2.1
	2.0
	1.1
	1.0

有关 .NET 版本的生命周期的详细信息，请参阅 .NET Core 和 .NET 5 支持策略。

依赖项

安装 .NET 时，例如手动安装时，可能不会安装特定依赖项。下面的列表详细列出了 Microsoft 支持的 Linux 发行版以及可能需要安装的依赖项。更多信息，请查看发行版页面：

- Alpine
- Debian
- CentOS
- Fedora
- RHEL
- SLES
- Ubuntu

有关依赖项的一般信息，请参阅独立式 Linux 应用。

RPM 依赖项

如果之前未列出发行版，并且该版本基于 RPM，则可能需要以下依赖项：

- krb5-libs
- libicu
- openssl-libs

如果目标运行时环境的 OpenSSL 版本为1.1 或更高版本，则需要安装 compat-openssl10。

DEB 依赖项

如果之前未列出发行版，并且该版本基于 debian，则可能需要以下依赖项：

- libc6
- libgcc1
- libgssapi-krb5-2
- libicu67
- libssl1.1
- libstdc++6
- zlib1g

通用依赖项

对于使用 System.Drawing.Common 程序集的 .NET 应用，还需要以下依赖项：

- libgdiplus（版本 6.0.1 或更高版本）

> **WARNING**
> 可以通过将 Mono 存储库添加到系统来安装最新版 libgdiplus。有关详细信息，请参阅 https://www.mono-project.com/download/stable/。

脚本安装

dotnet-install 脚本用于 SDK 和 运行时 的自动化和非管理员安装。可通过 https://dot.net/v1/dotnet-install.sh 下载脚本。

> ！[重要说明]需要 Bash 才能运行该脚本。

此脚本默认安装最新的 SDK 长期支持 (LTS) 版本，即 .NET Core 3.1。若要安装当前版本（可能不是 (LTS) 版本），请使用 `-c Current` 参数。

```
./dotnet-install.sh -c Current
```

若要安装 .NET 运行时而非 SDK，请使用 `--runtime` 参数。

```
./dotnet-install.sh -c Current --runtime aspnetcore
```

可以通过更改 `-c` 参数以指示特定版本来安装特定版本。以下命令将安装 .NET SDK 5.0。

```
./dotnet-install.sh -c 5.0
```

有关详细信息，请参阅 dotnet-install 脚本参考。

手动安装

除了使用包管理器，还可以下载并手动安装 SDK 和运行时。手动安装通常作为持续集成测试的一部分执行，或在不支持的 Linux 发行版上执行。对于开发人员或用户，使用包管理器会更好。

首先，从以下站点之一下载 SDK 或运行时的二进制版本。如果安装 .NET SDK，则无需安装相应的运行时：

- ✔ .NET 5 下载
- ✔ .NET Core 3.1 下载
- ✔ .NET Core 2.1 下载
- 所有 .NET Core 下载项

接下来，提取已下载的文件并使用 `export` 命令将 `DOTNET_ROOT` 设置为提取文件夹的位置，然后确保 .NET 位于 PATH 中。这应会使 .NET CLI 命令在终端中可用。

或者，下载 .NET 二进制文件后，可以从保存文件的目录运行以下命令以提取运行时。这也会使 .NET CLI 命令在终端可用并设置所需的环境变量。请务必将 `DOTNET_FILE` 值更改为下载的二进制文件的名称：

```
DOTNET_FILE=dotnet-sdk-5.0.302-linux-x64.tar.gz
export DOTNET_ROOT=$(pwd)/dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT
```

TIP

前面的 `export` 命令只会使 .NET CLI 命令对运行它的终端会话可用。

你可以编辑 shell 配置文件，永久地添加这些命令。Linux 提供了许多不同的 shell，每个都有不同的配置文件。例如：

- **Bash Shell**：~/.bash_profile、~/.bashrc
- **Korn Shell**：~/.kshrc 或 .profile
- **Z Shell**：~/.zshrc 或 .zprofile

为 shell 编辑相应的源文件，并将 `:$HOME/dotnet` 添加到现有 `PATH` 语句的末尾。如果不包含 `PATH` 语句，则使用 `export PATH=$PATH:$HOME/dotnet` 添加新行。

另外，将 `export DOTNET_ROOT=$HOME/dotnet` 添加至文件的末尾。

使用此方法可以将不同的版本安装到不同的位置，并明确选择应用程序要使用的对应版本。

后续步骤

- 如何为 .NET CLI 启用 Tab 自动补全
- 教程：使用 Visual Studio Code 通过 .NET SDK 创建控制台应用程序

如何删除 .NET 运行时和 SDK

2021/11/16 •

经过一段时间后，在安装 .NET 运行时和 SDK 的更新版本时，你可能需要从计算机中删除过时的 .NET 版本。如 .NET 版本选择一文中详述，删除旧版运行时可能会更改为运行共享框架应用程序所选择的运行时。

是否应删除某个版本？

借助 .NET 版本选择行为和 .NET 各个更新之间的运行时兼容性，可安全地删除以前的版本。.NET 运行时更新在主版本"区段"（如 1.x 和 2.x）中兼容。此外，较新版本的 .NET SDK 通常能够兼容地生成面向运行时早期版本的应用程序。

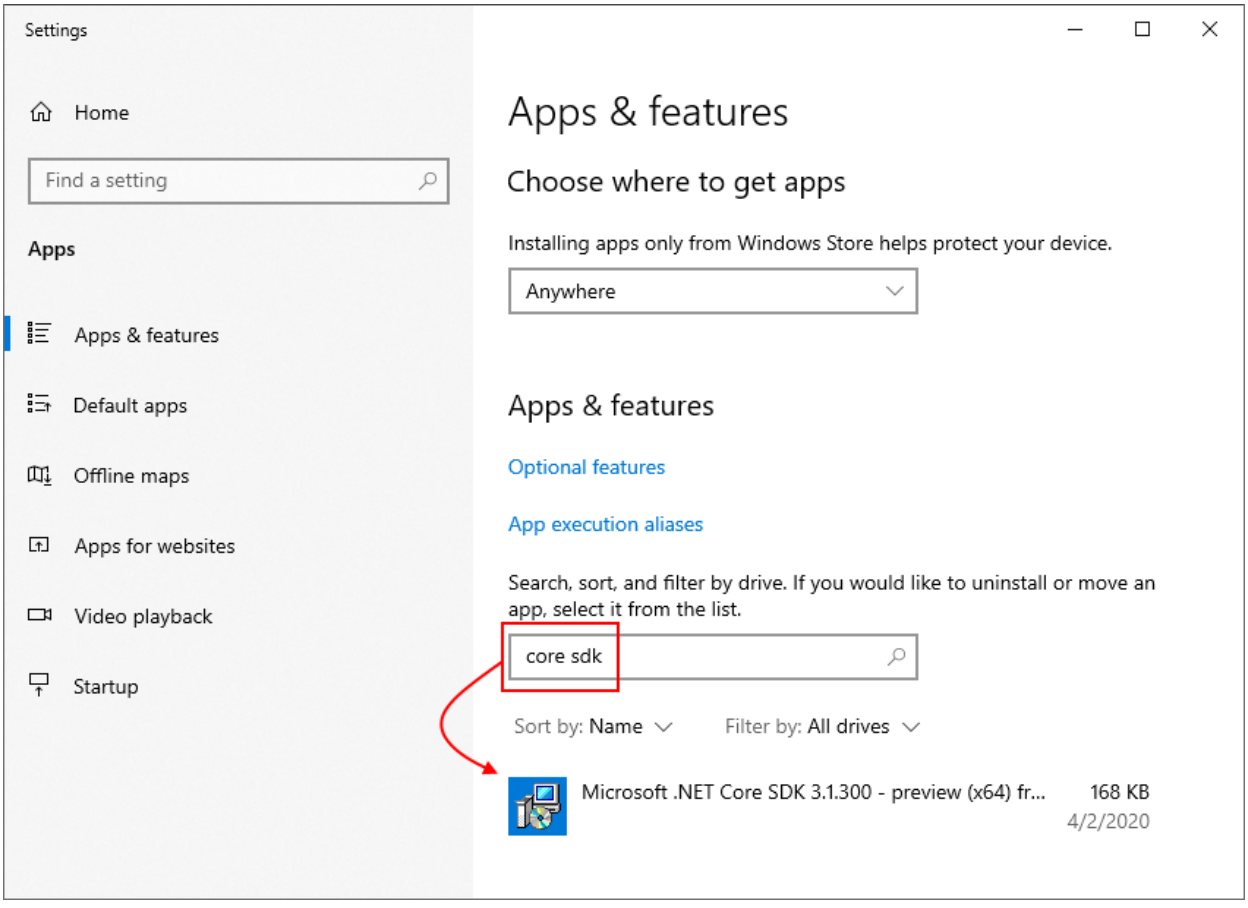
通常，只需要应用程序所需的最新 SDK 和运行时的最新补丁版本。需要保留旧版 SDK 或运行时版本的实例包括维护基于 project.json 的应用程序 。除非应用程序有需保留早期 SDK 或运行时的特定原因，否则可以安全地删除旧版本。

确定安装内容

.NET CLI 提供了相关的选项，你可以使用它们来列出计算机上安装的 SDK 和运行时。使用 `dotnet --list-sdks` 查看计算机上安装的 SDK 列表。使用 `dotnet --list-runtimes` 查看计算机上安装的运行时列表。有关详细信息，请参阅如何检查是否已安装 .NET。

卸载 .NET

.NET 使用 Windows"应用和功能"对话框来删除各版 .NET 运行时和 SDK。下图显示了"应用和功能"对话框 。你可以搜索 core sdk 或 .net sdk 来筛选和显示安装的 .NET 版本 。



选择要从计算机中删除的任何版本，然后单击“卸载”。

Linux 还提供其他选项来卸载 .NET（SDK 或运行时）。卸载 .NET 的最佳方法是镜像用来安装 .NET 的操作。具体取决于所选择的分发和安装方法。

> **IMPORTANT**
> 有关 Red Hat 安装，请参阅 Red Hat Product Documentation for .NET（Red Hat .NET 产品文档）。

除非从预览版本进行升级，否则使用包管理器升级时无需卸载 .NET SDK。包管理器 `update` 或 `refresh` 命令将在成功安装较新版本后自动删除旧版本。如果已安装预览版本，请卸载该版本。

如果使用包管理器安装 .NET，则使用同一包管理器来卸载 .NET SDK 或运行时。.NET 安装支持常用的包管理器。有关环境中的精确语法，请查阅分发的包管理器文档：

- apt-get(8) 由基于 Debian 的系统（包括 Ubuntu）使用。
- yum(8) 用于 Fedora、CentOS 和 Oracle Linux。
- zypper(8) 用于 openSUSE 和 SUSE Linux Enterprise System (SLES)。
- dnf(8) 用于 Fedora。

几乎在所有情况下，删除包的命令都是 `remove`。

大多数包管理器的 .NET SDK 安装包名称为 `dotnet-sdk`，后跟版本号。从 2.1.300 版 .NET SDK 和 `2.1` 版运行时开始，只需要主版本号和次版本号：例如，可将 .NET SDK 2.1.300 版引用为包 `dotnet-sdk-2.1`。以前的版本则需要整个版本字符串：例如，2.1.200 版 .NET SDK 需要 `dotnet-sdk-2.1.200`。

对于仅安装了运行时而未安装 SDK 的计算机，.NET 运行时的包名称为 `dotnet-runtime-<version>`，整个运行时堆栈的包名称为 `aspnetcore-runtime-<version>`。

> **TIP**
> 使用包管理器卸载 SDK 时，2.0 之前的 .NET Core 安装不会卸载主机应用程序。使用 `apt-get`，该命令为：
>
> ```
> apt-get remove dotnet-host
> ```
>
> 没有版本附加到 `dotnet-host`。

如果使用 tarball 安装，则必须手动删除 .NET。

在 Linux 上，必须通过删除进行版本控制的目录，分别删除 SDK 和运行时。这些目录可能因你的 Linux 分发版而异。删除它们会从磁盘中删除 SDK 和运行时。例如，要删除 1.0.1 SDK 和运行时，可使用以下 bash 命令：

```
version="1.0.1"
sudo rm -rf /usr/share/dotnet/sdk/$version
sudo rm -rf /usr/share/dotnet/shared/Microsoft.NETCore.App/$version
sudo rm -rf /usr/share/dotnet/shared/Microsoft.AspNetCore.All/$version
sudo rm -rf /usr/share/dotnet/shared/Microsoft.AspNetCore.App/$version
sudo rm -rf /usr/share/dotnet/host/fxr/$version
```

SDK 和运行时的父目录列在 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令的输出中，如上表所示。

在 Mac 上，必须通过删除进行版本控制的目录，分别删除 SDK 和运行时。删除它们会从磁盘中删除 SDK 和运行时。例如，要删除 1.0.1 SDK 和运行时，可使用以下 bash 命令：

```
version="1.0.1"
sudo rm -rf /usr/local/share/dotnet/sdk/$version
sudo rm -rf /usr/local/share/dotnet/shared/Microsoft.NETCore.App/$version
sudo rm -rf /usr/local/share/dotnet/shared/Microsoft.AspNetCore.All/$version
sudo rm -rf /usr/local/share/dotnet/shared/Microsoft.AspNetCore.App/$version
sudo rm -rf /usr/local/share/dotnet/host/fxr/$version
```

SDK 和运行时的父目录列在 `dotnet --list-sdks` 和 `dotnet --list-runtimes` 命令的输出中，如上表所示。

.NET 卸载工具

你可以使用 .NET 卸载工具 (`dotnet-core-uninstall`) 从系统中删除 .NET SDK 和运行时。可使用选项集合来指定应卸载的版本。

.NET Core SDK 版本的 Visual Studio 依赖项

在 Visual Studio 2019 版本 16.3 之前，Visual Studio 安装程序称为独立的 .NET Core SDK 安装程序。因此，SDK 版本显示在 Windows"应用和功能"对话框中 。使用独立安装程序删除 Visual Studio 安装的 .NET Core SDK 可能会破坏 Visual Studio。如果 Visual Studio 在卸载 SDK 之后出现问题，请在该特定版本的 Visual Studio 上运行修复。下表显示了 .NET Core SDK 版本的一些 Visual Studio 依赖项：

VISUAL STUDIO	.NET CORE SDK
Visual Studio 2019 版本 16.2	.NET Core SDK 2.2.4xx、2.1.8xx
Visual Studio 2019 版本 16.1	.NET Core SDK 2.2.3xx、2.1.7xx
Visual Studio 2019 版本 16.0	.NET Core SDK 2.2.2xx、2.1.6xx
Visual Studio 2017 版本 15.9	.NET Core SDK 2.2.1xx、2.1.5xx
Visual Studio 2017 版本 15.8	.NET Core SDK 2.1.4xx

从 Visual Studio 2019 16.3 版开始，Visual Studio 负责其自己的 .NET SDK 副本。因此，"应用和功能"对话框中将不再显示这些 SDK 版本 。

删除 NuGet 回退文件夹

在 .NET Core 3.0 SDK 之前，.NET Core SDK 安装程序使用名为 NuGetFallbackFolder 的文件夹存储 NuGet 包的缓存 。此缓存在操作期间（如 `dotnet restore` 或 `dotnet build /t:Restore`）使用。NuGetFallbackFolder 在 Windows 上位于 C:\Program Files\dotnet\sdk，在 macOS 上位于 /usr/local/share/dotnet/sdk 。

如果是以下情况，则可能需要删除此文件夹：

- 仅使用 .NET Core 3.0 SDK 或 .NET 5（或更高版本）进行开发。
- 你使用早于 3.0 的 .NET Core SDK 版本进行开发，但可以联机工作。

如果要删除 NuGet 回退文件夹，可以将其删除，但需要管理员权限才能执行此操作。

建议不要删除 dotnet 文件夹。这样做会删除以前安装的所有全局工具。此外，在 Windows 上：

- 你将中断 Visual Studio 2019 版本 16.3 及更高版本。可以运行"修复" 来恢复。
- 如果"应用和功能"对话框中存在 .NET Core SDK 条目，它们将是孤立的 。

管理 .NET 项目和项模板

2021/11/16 •

.NET Core 提供了一个模板系统，允许用户从 NuGet、NuGet 包文件或文件系统目录中安装或卸载模板。本文介绍如何通过 .NET SDK CLI 管理 .NET Core 模板。

若要详细了解如何创建模板，请参阅教程：创建模板。

安装模板

使用 `-i` 参数通过 dotnet new SDK 命令安装模板。可以提供模板的 NuGet 包标识符或包含模板文件的文件夹。

NuGet 托管包

.NET CLI 模板上传到 NuGet 以便进行广泛分发。还可以从专用源安装模板。如本地 NuGet 包部分所述，可以分发和手动安装 nupkg 模板文件，而不是将模板上传到 NuGet 源。

有关配置 NuGet 源的详细信息，请参阅 dotnet nuget add source。

若要从默认 NuGet 源安装模板包，请使用 `dotnet new -i {package-id}` 命令：

```
dotnet new -i Microsoft.DotNet.Web.Spa.ProjectTemplates
```

若要从默认 NuGet 源安装特定版本的模板包，请使用 `dotnet new -i {package-id}::{version}` 命令：

```
dotnet new -i Microsoft.DotNet.Web.Spa.ProjectTemplates::2.2.6
```

本地 NuGet 包

创建模板包后，将生成一个 nupkg 文件。如果有包含模板的 nupkg 文件，则可以使用 `dotnet new -i {path-to-package}` 命令进行安装：

```
dotnet new -i c:\code\nuget-packages\Some.Templates.1.0.0.nupkg
```

```
dotnet new -i ~/code/nuget-packages/Some.Templates.1.0.0.nupkg
```

文件夹

从 nupkg 文件安装模板的一个替代方法是使用 `dotnet new -i {folder-path}` 命令直接从文件夹安装模板。指定的文件夹将被视为找到的任何模板的模板包标识符。安装指定文件夹的层次结构中找到的任何模板。

```
dotnet new -i c:\code\nuget-packages\some-folder\
```

```
dotnet new -i ~/code/nuget-packages/some-folder/
```

在命令中指定的 `{folder-path}` 将成为找到的所有模板的模板包标识符。如模板列表部分中指定的一样，可以使用 `dotnet new -u` 命令获取已安装的模板列表。在此示例中，模板包标识符显示为用于安装的文件夹：

```
dotnet new -u
Template Instantiation Commands for .NET CLI

Currently installed items:

... cut to save space ...

  c:\code\nuget-packages\some-folder
    Templates:
      A Template Console Class (templateconsole) C#
      Project for some technology (contosoproject) C#
    Uninstall Command:
      dotnet new -u c:\code\nuget-packages\some-folder
```

```
dotnet new -u
Template Instantiation Commands for .NET CLI

Currently installed items:

... cut to save space ...

  /home/username/code/templates
    Templates:
      A Template Console Class (templateconsole) C#
      Project for some technology (contosoproject) C#
    Uninstall Command:
      dotnet new -u /home/username/code/templates
```

卸载模板

使用 `-u` 参数通过 dotnet new SDK 命令卸载模板。可以提供模板的 NuGet 包标识符或包含模板文件的文件夹。

NuGet 程序包

安装 NuGet 模板包后，无论是从 NuGet 源还是从 nupkg 文件安装的，都可以通过引用 NuGet 包标识符将其卸载。

若要卸载模板包，请使用 `dotnet new -u {package-id}` 命令：

```
dotnet new -u Microsoft.DotNet.Web.Spa.ProjectTemplates
```

文件夹

如果通过文件夹路径安装模板，文件夹路径将成为模板包标识符。

若要卸载模板包，请使用 `dotnet new -u {package-folder-path}` 命令：

```
dotnet new -u c:\code\nuget-packages\some-folder
```

```
dotnet new -u /home/username/code/templates
```

模板列表

通过使用不带包标识符的标准卸载命令，可以查看已安装模板的列表以及用于卸载每个模板的命令。

```
dotnet new -u
Template Instantiation Commands for .NET CLI

Currently installed items:

... cut to save space ...

  c:\code\nuget-packages\some-folder
    Templates:
      A Template Console Class (templateconsole) C#
      Project for some technology (contosoproject) C#
    Uninstall Command:
      dotnet new -u c:\code\nuget-packages\some-folder
```

从其他 SDK 安装模板

如果已按顺序安装了每个版本的 SDK(例如安装了 SDK 2.0、SDK 2.1 等)，则已安装每个 SDK 的模板。但是，如果从更高版本的 SDK 版本(如 3.1)开始，则将仅包括 LTS(长期支持)版本的模板(在发布 SDK 3.1 版本时，为 SDK 2.1 和 SDK 3.1)。不包含任何其他版本的模板。

NuGet 上提供了 .NET Core 模板，你可以像安装任何其他模板一样安装它们。有关详细信息，请参阅安装 NuGet 托管包。

SDK	NUGET
.NET Core 2.1	`Microsoft.DotNet.Common.ProjectTemplates.2.1`
.NET Core 2.2	`Microsoft.DotNet.Common.ProjectTemplates.2.2`
.NET Core 3.0	`Microsoft.DotNet.Common.ProjectTemplates.3.0`
.NET Core 3.1	`Microsoft.DotNet.Common.ProjectTemplates.3.1`
.NET Core 5.0	`Microsoft.DotNet.Common.ProjectTemplates.3.1`
ASP.NET Core 2.1	`Microsoft.DotNet.Web.ProjectTemplates.2.1`
ASP.NET Core 2.2	`Microsoft.DotNet.Web.ProjectTemplates.2.2`
ASP.NET Core 3.0	`Microsoft.DotNet.Web.ProjectTemplates.3.0`
ASP.NET Core 3.1	`Microsoft.DotNet.Web.ProjectTemplates.3.1`
ASP.NET Core 5.0	`Microsoft.DotNet.Web.ProjectTemplates.3.1`

例如，.NET Core SDK 包含面向 .NET Core 2.1 和 .NET Core 3.1 的控制台应用的模板。如果要面向 .NET Core 3.0，则需要安装 3.0 模板。

1. 尝试创建面向 .NET Core 3.0 的应用。

   ```
   dotnet new console --framework netcoreapp3.0
   ```

   如果看到错误消息，则需要安装模板。

> 找不到与输入匹配的已安装模板，正在联机搜索匹配的模板...

2. 安装 .NET Core 3.0 项目模板。

```
dotnet new -i Microsoft.DotNet.Common.ProjectTemplates.3.0
```

3. 再次尝试创建应用。

```
dotnet new console --framework netcoreapp3.0
```

应该会看到一条消息，指示项目已创建。

> “控制台应用程序”模板已成功创建。
>
> 正在处理创建后操作...正在 path-to-project-file.csproj 上运行“dotnet restore”...正在确定要还原的项目...path-to-project-file.csproj 的还原完成时间为 1.05 秒。
>
> 还原成功。

请参阅

- 教程：创建项模板
- dotnet new
- dotnet nuget add source

macOS Catalina 公证以及对 .NET Core 下载和项目的影响

2021/11/16 •

自 macOS Catalina（版本 10.15）开始，所有在 2019 年 6 月 1 日之后生成并使用开发者 ID 扩散的软件都必须经过公证。此要求适用于 .NET Core 运行时、.NET SDK 以及使用 .NET 创建的软件。本文介绍了 .NET Core 和 macOS 公证可能会遇到的常见情况。

安装 .NET Core

自 2020 年 2 月 18 日起，.NET Core（运行时和 SDK）版本 3.1、3.0 和 2.1 的安装程序都已经过公证。以前发布的版本没有经过公证。通过先下载安装程序，然后使用 `sudo installer` 命令，可以手动安装 .NET Core 的未公证版本。有关详细信息，请参阅下载并手动安装 macOS。

自以下版本开始，.NET Core 安装程序未经过公证：

- .NET Core 运行时
  - 2.1.16
  - 3.0.3
  - 3.1.2
- .NET Core SDK
  - 2.1.512
  - 3.0.103
  - 3.1.102

默认禁用 appHost

默认情况下，当项目编译、发布或运行时，.NET Core SDK 3.0 及更高版本的未公证版本会生成一个本机 Mach-O 可执行文件（即 appHost）。此可执行文件是运行应用的一种简便方法。否则必须通过运行 `dotnet <filename.dll>` 启动应用。启用 appHost 后，会在 appHost 的上下文中调用 `dotnet run` 命令。有关详细信息，请参阅 appHost 的上下文。

从 .NET Core SDK 3.0 及更高版本的已公证版本开始，默认情况下不生成 appHost 可执行文件。可以通过项目文件中的 `UseAppHost` 布尔值设置来启用 appHost 生成。还可以在运行的特定 `dotnet` 命令的命令行上通过 `-p:UseAppHost` 参数切换 appHost 的启用状态：

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

发布独立应用时，始终会创建 appHost。

有关 `UseAppHost` 设置的详细信息，请参阅 Microsoft.NET.Sdk 的 MSBuild 属性。

AppHost 的上下文

在项目中启用了 appHost 并使用 `dotnet run` 命令运行应用时，将在 appHost 的上下文中调用应用，而不是在默认主机（默认主机对应的是 `dotnet` 命令）的上下文中调用。如果在项目中禁用了 appHost，则 `dotnet run` 命令将在默认主机的上下文中运行应用。即使禁用了 appHost，如果发布独立应用，也会生成一份 appHost 可执行文件，且用户也可以使用该可执行文件运行应用。使用 `dotnet <filename.dll>` 运行应用会通过默认主机（共享运行时）调用应用。

调用使用 appHost 的应用时，应用访问的证书分区与已公证的默认主机不同。如果应用必须访问通过默认主机安装的证书，请使用 `dotnet run` 命令从应用的项目文件运行它，或使用 `dotnet <filename.dll>` 命令直接启动该应用。

有关此方案的详细信息，请参阅 ASP.NET Core 和 macOS 和证书部分。

ASP.NET Core 和 macOS 和证书

使用 .NET Core，可以使用 System.Security.Cryptography.X509Certificates 类在 macOS Keychain 中管理证书。在决定要考虑哪个分区时，对 macOS Keychain 的访问将使用应用程序标识作为主键。例如，未签名的应用程序会将机密存储在未签名分区中，而已签名应用程序会将其机密存储在仅能由它们访问的分区中。要使用的分区取决于调用应用的执行源。

.NET Core 提供三个执行源：appHost、默认主机（`dotnet` 命令）和自定义主机。每个执行模型都可能具有不同的标识（已签名或未签名），并可以访问 Keychain 中的不同分区。通过一种模式导入的证书可能不能通过另一种模式访问。例如，已公证版本的 .NET Core 具有已签名的默认主机。根据证书的标识将证书导入到安全分区中。由于 appHost 是未签名的，因此无法从生成的 appHost 访问这些证书。

再举一例，默认情况下，ASP.NET Core 通过默认主机导入默认 SSL 证书。使用 appHost 的 ASP.NET Core 应用程序将不能访问此证书，并且当 .NET Core 检测到无法访问该证书时，将收到错误消息。该错误消息中会提供关于如何解决此问题的说明。

如果证书共享是必需的，macOS 会提供带有 `security` 实用工具的配置选项。

如需详细了解如何解决 ASP.NET Core 证书问题，请参阅在 ASP.NET Core 中强制执行 HTTPS。

默认权利

.NET Core 的默认主机（`dotnet` 命令）具有一组默认权利。需要这些权利才能正常运行 .NET Core。你的应用程序可能需要其他权利，在这种情况下，需要生成并使用 appHost，然后在本地添加必要的权利。

.NET Core 的默认权利包括：

- `com.apple.security.cs.allow-jit`
- `com.apple.security.cs.allow-unsigned-executable-memory`
- `com.apple.security.cs.allow-dyld-environment-variables`
- `com.apple.security.cs.disable-library-validation`

对 .NET Core 应用进行公证

如果希望应用程序在 macOS Catalina（版本 10.15）或更高版本上运行，则需要对应用进行公证。为了进行公证而随应用程序提交的 appHost 应至少具备与 .NET Core 相同的默认权利。

后续步骤

- .NET Core 依赖项和要求。

- 安装 .NET Core 运行时和 SDK。

排查 *fxr* 、 *libhostfxr.so* 和 *FrameworkList.xml* 错误

2021/11/16 •

尝试使用 .NET 5+（和 .NET Core）时，诸如 `dotnet new`、`dotnet run` 之类的命令可能会失败，并显示与找不到某项内容相关的消息。某些错误消息可能如下所示：

- **System.IO.FileNotFoundException**

  > System.IO.FileNotFoundException: 找不到文件“/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml”。

- **出现错误。**

  > 出现错误。找不到所需的 libhostfxr.so 库。

  或

  > 出现错误。文件夹 [/usr/share/dotnet/host/fxr] 不存在。

  或

  > 出现错误，文件夹 [/usr/share/dotnet/host/fxr] 不包含任何以版本编号的子文件夹。

- **找不到有关 dotnet 的一般消息**

  可能会出现一条一般消息，指示找不到 SDK 或已安装包。

这些问题都有一个症状，即，`/usr/lib64/dotnet` 和 `/usr/share/dotnet` 文件夹都在你的系统上。

这是怎么回事

当两个 Linux 包存储库提供 .NET 包时，通常会发生这种情况。Microsoft 提供一个 Linux 包存储库来获取 .NET 包的源代码，而某些 Linux 发行版也提供 .NET 包，例如：

- Arch
- CentOS
- Fedora
- RHEL

混合使用来自两个不同源的 .NET 包很可能会导致问题，因为这些包可能会将内容放在不同的路径上，并且可能会以不同的方式进行编译。

解决方案

解决这些问题的方法是使用一个包存储库中的 .NET。选择哪个存储库，以及如何选择，皆因用例和 Linux 发行版而异。

如果发行版提供 .NET 包，则建议使用该包存储库，而不是 Microsoft 的包存储库。

1. **我仅使用 Microsoft 存储库中的 .NET 包，未使用其他包，而且我的发行版提供 .NET 包。**

如果仅使用 Microsoft 存储库中的 .NET 包，而未使用任何其他 Microsoft 包，例如 `mdatp`、`powershell` 或 `mssql`，则：

a. 删除 Microsoft 存储库
b. 从 OS 中删除与 .NET 相关的包
c. 安装发行版存储库中的 .NET 包

对于 Fedora、CentOS 8+、RHEL 8+，请使用以下 bash 命令：

```
sudo dnf remove packages-microsoft-prod
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
sudo dnf install dotnet-sdk-5.0
```

2. **我想使用发行版提供的 .NET 包，但我也使用 Microsoft 存储库中的其他包。**

   如果使用 Microsoft 存储库中的 Microsoft 包（例如 `mdatp`、`powershell` 或 `mssql`），但不想使用该存储库中的 .NET，则：

   a. 配置 Microsoft 存储库以排除任何 .NET 包
   b. 从 OS 中删除与 .NET 相关的包
   c. 安装发行版存储库中的 .NET 包

   对于 Fedora、CentOS 8+、RHEL 8+，请使用以下 bash 命令：

```
echo 'excludepkgs=dotnet*,aspnet*,netstandard*' | sudo tee -a /etc/yum.repos.d/microsoft-prod.repo
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
sudo dnf install dotnet-sdk-5.0
```

3. **我需要 Linux 发行版存储库未提供的最新版 .NET。**

   在这种情况下，请保留 Microsoft 存储库，但对其进行配置，以使 Microsoft 存储库中的 .NET 包具有更高的优先级。然后，删除已安装的 .NET 包，重新安装 Microsoft 存储库中的 .NET 包。

   对于 Fedora、CentOS 8+、RHEL 8+，请使用以下 bash 命令：

```
echo 'priority=50' | sudo tee -a /etc/yum.repos.d/microsoft-prod.repo
sudo dnf remove 'dotnet*' 'aspnet*' 'netstandard*'
sudo dnf install dotnet-sdk-5.0
```

4. **Linux 发行版的 .NET 出现了 bug，我需要最新的 Microsoft 版本。**

   按照解决方案 3 解决此问题。

在线参考资料

其中许多问题是由像你这样的用户报告的。下面列出了这些问题。你可以认真通读，了解可能会发生的情况：

- System.IO.FileNotFoundException 和"/usr/share/dotnet/packs/Microsoft.NETCore.App.Ref/5.0.0/data/FrameworkList.xml"
  - SDK #15785: unable to build brand new project after upgrading to 5.0.3（SDK #15785：升级到 5.0.3 后无法生成全新的项目）
  - SDK #15863: "MSB4018 ResolveTargetingPackAssets task failed unexpectedly" after updating to 5.0.103（SDK #15863：更新到 5.0.103 后，"MSB4018 ResolveTargetingPackAssets 任务意外失败"）
  - SDK #17411: dotnet build always throwing error（SDK #17411：dotnet 生成始终引发错误）
  - SDK #12075: dotnet 3.1.301 on Fedora 32 unable to find FrameworkList.xml because it doesn't exist（SDK #12075：Fedora 32 上的 dotnet 3.1.301 找不到 FrameworkList.xml，因为它不存在）

- 错误：找不到 libhostfxr.so
  - SDK #17570: After updating Fedora 33 to 34 and dotnet 5.0.5 to 5.0.6 I get error regarding libhostfxr.so（SDK #17570：将 Fedora 从 33 更新到 34，将 dotnet 从 5.0.5 更新到 5.0.6 后，我收到了有关 libhostfxr.so 的错误）
- 错误：文件夹 /host/fxr 不存在
  - Core #5746: The folder does not exist when installing 3.1 on CentOS 8 with packages.microsoft.com repo enabled（核心 #5746：在启用了 packages.microsoft.com 存储库的 CentOS 8 上安装 3.1 时，该文件夹不存在）
  - SDK #15476: A fatal error occurred. The folder [/usr/share/dotnet/host/fxr] does not exist（SDK #15476：出现错误。文件夹 [/usr/share/dotnet/host/fxr] 不存在）
- 错误：文件夹 /host/fxr 不包含任何以版本为编号的子文件夹
  - Installer #9254: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders（安装程序 #9254：在 CentOS 8 上安装 dotnet/core/aspnet:3.1 时出错 - 文件夹不包含任何以版本为编号的子文件夹）
  - StackOverflow: Error when install dotnet/core/aspnet:3.1 on CentOS 8 - Folder does not contain any version-numbered child folders（StackOverflow：在 CentOS 8 上安装 dotnet/core/aspnet:3.1 时出错 - 文件夹不包含任何以版本为编号的子文件夹）
- 没有明确消息的一般错误
  - Core #4605: cannot run "dotnet new console"（核心 #4605：无法运行“dotnet new console”）
  - Core #4644: Cannot install .NET Core SDK 2.1 on Fedora 32（核心 #4644：无法在 Fedora 32 上安装 .NET Core SDK 2.1）
  - Runtime #49375: After updating to 5.0.200-1 using package manager, it appears that no sdks are installed（运行时 #49375：使用包管理器更新到 5.0.200-1 后，似乎未安装任何 SDK）

后续步骤

- 在 Linux 上安装 .NET
- 如何删除 .NET 运行时和 SDK
- 教程：使用 Visual Studio Code 创建一个新应用。
- 教程：使 .NET 应用容器化。

如何检查是否已安装 .NET

2021/11/16 •

本文介绍如何检查计算机上安装的 .NET 运行时和 SDK 的版本。如果你拥有一个集成开发环境(如 Visual Studio 或 Visual Studio for Mac),则可能已安装 .NET。

安装 SDK 便会安装相应的运行时。

如果本文中的任何命令失败,则未安装运行时或 SDK。有关详细信息,请参阅 Windows、macOS 或 Linux 的安装文章。

检查 SDK 版本

可使用终端查看当前安装的 .NET SDK 版本。打开终端并运行以下命令。

```
dotnet --list-sdks
```

将获得类似于下面的输出。

```
2.1.500 [C:\program files\dotnet\sdk]
2.1.502 [C:\program files\dotnet\sdk]
2.1.504 [C:\program files\dotnet\sdk]
2.1.600 [C:\program files\dotnet\sdk]
2.1.602 [C:\program files\dotnet\sdk]
3.1.100 [C:\program files\dotnet\sdk]
5.0.100 [C:\program files\dotnet\sdk]
```

```
2.1.500 [/home/user/dotnet/sdk]
2.1.502 [/home/user/dotnet/sdk]
2.1.504 [/home/user/dotnet/sdk]
2.1.600 [/home/user/dotnet/sdk]
2.1.602 [/home/user/dotnet/sdk]
3.1.100 [/home/user/dotnet/sdk]
5.0.100 [/home/user/dotnet/sdk]
```

```
2.1.500 [/usr/local/share/dotnet/sdk]
2.1.502 [/usr/local/share/dotnet/sdk]
2.1.504 [/usr/local/share/dotnet/sdk]
2.1.600 [/usr/local/share/dotnet/sdk]
2.1.602 [/usr/local/share/dotnet/sdk]
3.1.100 [/usr/local/share/dotnet/sdk]
5.0.100 [/usr/local/share/dotnet/sdk]
```

检查运行时版本

可使用以下命令查看当前安装的 .NET 运行时版本。

```
dotnet --list-runtimes
```

将获得类似于下面的输出。

```
Microsoft.AspNetCore.All 2.1.7 [c:\program files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.13 [c:\program files\dotnet\shared\Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.7 [c:\program files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.13 [c:\program files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 3.1.0 [c:\program files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.0 [c:\program files\dotnet\shared\Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.1.7 [c:\program files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.13 [c:\program files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 3.1.0 [c:\program files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.0 [c:\program files\dotnet\shared\Microsoft.NETCore.App]
Microsoft.WindowsDesktop.App 3.0.0 [c:\program files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 3.1.0 [c:\program files\dotnet\shared\Microsoft.WindowsDesktop.App]
Microsoft.WindowsDesktop.App 5.0.0 [c:\program files\dotnet\shared\Microsoft.WindowsDesktop.App]
```

```
Microsoft.AspNetCore.All 2.1.7 [/home/user/dotnet/shared/Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.13 [/home/user/dotnet/shared/Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.7 [/home/user/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.13 [/home/user/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 3.1.0 [/home/user/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.0 [/home/user/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.1.7 [/home/user/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.13 [/home/user/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 3.1.0 [/home/user/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.0 [/home/user/dotnet/shared/Microsoft.NETCore.App]
```

```
Microsoft.AspNetCore.All 2.1.7 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.All]
Microsoft.AspNetCore.All 2.1.13 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.All]
Microsoft.AspNetCore.App 2.1.7 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 2.1.13 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 3.1.0 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.AspNetCore.App 5.0.0 [/usr/local/share/dotnet/shared/Microsoft.AspNetCore.App]
Microsoft.NETCore.App 2.1.7 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 2.1.13 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 3.1.0 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]
Microsoft.NETCore.App 5.0.0 [/usr/local/share/dotnet/shared/Microsoft.NETCore.App]
```

检查安装文件夹

可以安装 .NET，但不能将其添加到操作系统或用户配置文件的 `PATH` 变量。在这种情况下，前面几节中的命令可能不起作用。作为替代方法，可以检查 .NET 安装文件夹是否存在。

从安装程序或脚本安装 .NET 时，会将其安装到标准文件夹中。大多数时候，安装 .NET 时使用的安装程序或脚本会让你选择安装到另一个文件夹。如果选择安装到其他文件夹，请调整文件夹路径的开头。

- dotnet executable
  C:\program files\dotnet\dotnet.exe
- **.NET SDK**
  C:\program files\dotnet\sdk\{version}\
- .NET Runtime
  C:\program files\dotnet\shared\{runtime-type}\{version}\
- dotnet executable
  /home/user/share/dotnet/dotnet
- **.NET SDK**
  /home/user/share/dotnet/sdk/{version}/

- .NET Runtime
  /home/user/share/dotnet/shared/{runtime-type}/{version}/
- dotnet executable
  /usr/local/share/dotnet/dotnet
- **.NET SDK**
  /usr/local/share/dotnet/sdk/{version}/
- .NET Runtime
  /usr/local/share/dotnet/shared/{runtime-type}/{version}/

详细信息

可通过命令 `dotnet --info` 查看 SDK 版本和运行时版本。你还将获得其他环境相关信息，如操作系统版本和运行时标识符 (RID)。

后续步骤

- 安装 .NET 运行时和适用于 Windows 的 SDK。
- 安装 .NET 运行时和适用于 macOS 的 SDK。
- 安装 .NET 运行时和适用于 Linux 的 SDK。

请参阅

- 确定已安装的 .NET Framework 版本

如何为 .NET 安装本地化的 IntelliSense 文件

2021/11/16 •

IntelliSense 是一种代码完成辅助工具，可以在不同的集成开发环境 (IDE) 中使用，例如 Visual Studio。默认情况下，在开发 .NET 项目时，SDK 仅包含英语版本的 IntelliSense 文件。本文介绍：

- 如何安装这些文件的本地化版本。
- 如何修改 Visual Studio 安装以使用其他语言。

系统必备

- .NET Core 3.0 SDK 或更高版本，例如 .NET 5 SDK。
- Visual Studio 2019 版本 16.3 或更高版本。

下载并安装本地化的 IntelliSense 文件

> **IMPORTANT**
> 此过程需具有管理员权限，才能将 IntelliSense 文件复制到 .NET 安装文件夹中。

1. 转到下载 IntelliSense 文件页面。
2. 下载要使用的语言和版本的 IntelliSense 文件。
3. 提取 zip 文件的内容。
4. 导航到 .NET Intellisense 文件夹。
   a. 导航到 .NET 安装文件夹。默认情况下，它位于 %ProgramFiles%\dotnet\packs 下 。
   b. 选择要为其安装 IntelliSense 的 SDK，然后导航到关联的路径。有下列选项：

SDK	
.NET 5+ 和 .NET Core	Microsoft.NETCore.App.Ref
Windows 桌面	Microsoft.WindowsDesktop.App.Ref
.NET Standard	NETStandard.Library.Ref

   c. 导航到要为其安装本地化 IntelliSense 的版本。例如，5.0.0。
   d. 打开 ref 文件夹 。
   e. 打开 moniker 文件夹。例如 net5.0。

   因此，要导航到的完整路径看起来将类似于 C:\Program Files\dotnet\packs\Microsoft.NETCore.App.Ref\5.0.0\ref\net5.0。
5. 在刚打开的 moniker 文件夹中创建一个子文件夹。文件夹名称指示要使用的语言。下表指定了不同的选项：

[illegible]	[illegible]
巴西葡萄牙语	pt-br
中文(简体)	zh-hans
中文(繁体)	zh-hant
法语	fr
德语	de
意大利语	it
日语	ja
朝鲜语	ko
俄语	ru
西班牙语	es

6. 将在步骤 3 中提取的 .xml 文件复制到此新文件夹。.xml 文件按 SDK 文件夹细分，因此，请将它们复制到步骤 4 中选择的相应 SDK。

修改 Visual Studio 语言

要使 Visual Studio 使用其他语言的 IntelliSense，请安装适当的语言包。这可以在安装过程中完成，也可以之后通过修改 Visual Studio 安装来完成。如果已将 Visual Studio 配置为所需的语言，那么 IntelliSense 安装已准备就绪。

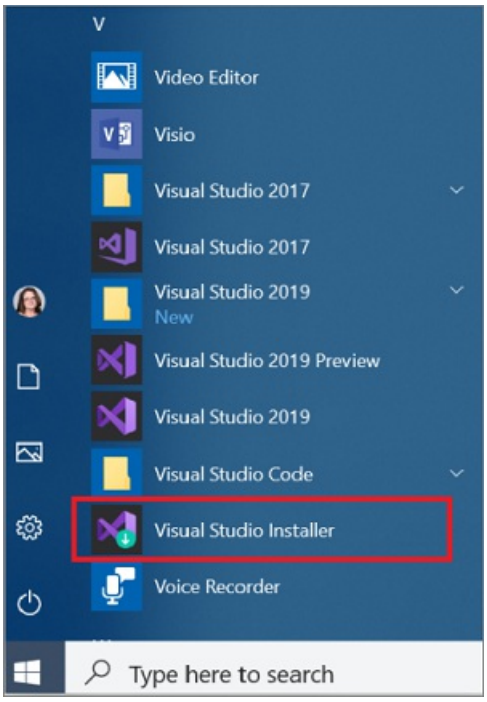
安装语言包

如果在安装过程中未安装所需的语言包，请按以下步骤更新 Visual Studio 来安装语言包：

> **IMPORTANT**
> 若要安装、更新或修改 Visual Studio，必须使用具有管理员权限的帐户登录。有关详细信息，请参阅用户权限与 Visual Studio。

1. 在计算机上找到 Visual Studio 安装程序。

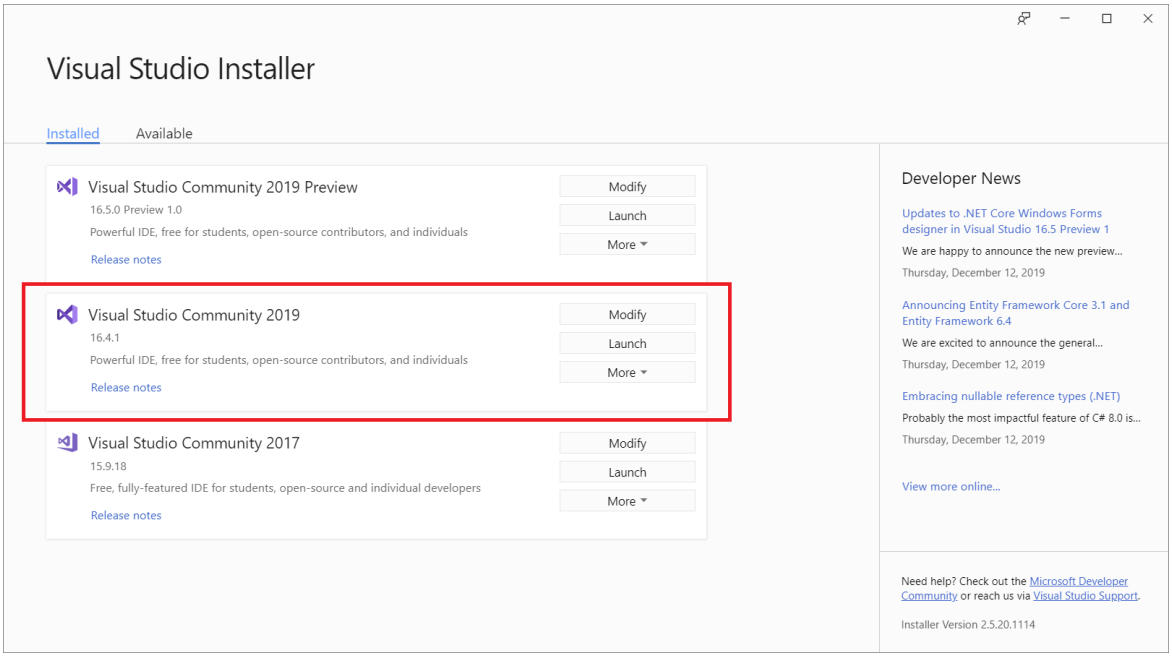
   例如，在运行 Windows 10 的计算机上，选择“开始”，然后滚动到字母“V”，它作为“Visual Studio 安装程序”在那里列出 。



> **NOTE**
>
> 还可以在以下位置中找到 Visual Studio 安装程序：
>
> ```
> C:\Program Files (x86)\Microsoft Visual Studio\Installer\vs_installer.exe
> ```

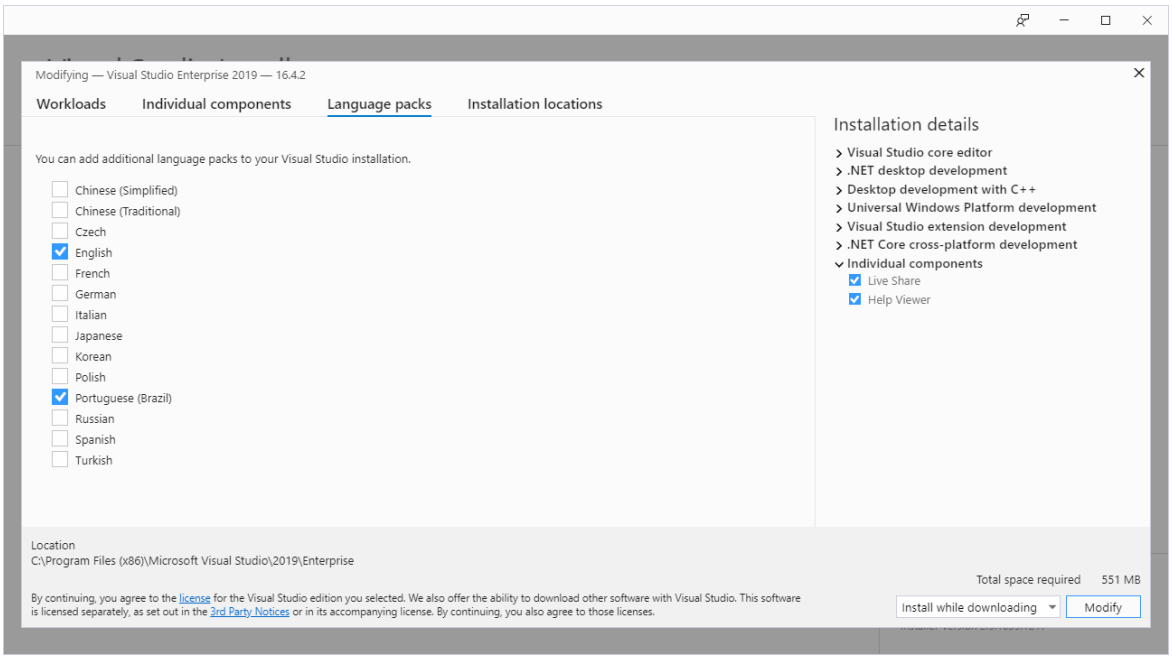
可能需要先更新安装程序，然后才能继续操作。如果是这样，请按照提示操作。

2. 在安装程序中，查找要为其添加语言包的 Visual Studio 版本，然后选择“修改”。



> **IMPORTANT**
>
> 如果未看到“修改”按钮，但看到了“更新”按钮，则需要先更新 Visual Studio，才能修改安装 。更新完成后，应会显示“修改”按钮 。

3. 在“语言包”选项卡中，选择或取消选择要安装或卸载的语言 。



4. 选择“修改”。更新开始。

修改 Visual Studio 中的语言设置

安装所需的语言包后，请修改 Visual Studio 设置以使用其他语言：

1. 打开 Visual Studio。
2. 在“启动”窗口中，选择“继续但无需代码”。
3. 在菜单栏上，选择“工具” > “选项”。“选项”对话框随即打开。
4. 在“环境”节点下，选择“国际设置”。
5. 在“语言”下拉列表中，选择所需的语言。选择 **“确定”**。
6. 随即会显示一个对话框，告知必须重启 Visual Studio 才能使所做的更改生效。选择 **“确定”**。
7. 重新启动 Visual Studio。

之后，当打开面向刚安装的 IntelliSense 文件版本的 .NET 项目时，IntelliSense 应会按预期方式工作。

另请参阅

- Visual Studio 中的 IntelliSense

.NET 简介

2021/11/16 •

.NET 是一种用于构建多种应用的免费开源开发平台，例如：

- Web 应用、Web API 和微服务
- 云中的无服务器函数
- 云原生应用
- 移动应用
- 桌面应用
  - Windows WPF
  - Windows 窗体
  - 通用 Windows 平台 (UWP)
- 游戏
- 物联网 (IoT)
- 机器学习
- 控制台应用
- Windows 服务

使用类库在不同应用和应用类型中共享功能。

使用 .NET 时，无论你正在构建哪种类型的应用，代码和项目文件看起来都一样。可以访问每个应用的相同运行时、API 和语言功能。

跨平台

可以为许多操作系统创建 .NET 应用，包括：

- Windows
- macOS
- Linux
- Android
- iOS
- tvOS
- watchOS

支持的处理器体系结构包括：

- X64
- x86
- ARM32
- ARM64

通过 .NET，可以使用特定于平台的功能，如操作系统 API。例如 Windows 上的 Windows 窗体和 WPF，以及从 Xamarin 到每个移动平台的原生绑定。

有关详细信息，请参阅支持的 OS 生命周期策略和 .NET RID 目录。

开源

.NET 是开放源代码，使用 MIT 和 Apache 2 许可证。.NET 是 .NET Foundation 的项目。

有关详细信息，请参阅 GitHub.com 上的项目存储库列表。

支持

Microsoft 支持在 Windows、macOS 和 Linux 上使用 .NET。它会定期更新以保证安全和质量（每月的第二个星期二）。

Microsoft 的 .NET 二进制发行版在 Azure 中的 Microsoft 维护服务器上进行生成和测试，并遵循 Microsoft 的工程和安全实践。

Red Hat 支持在 Red Hat Enterprise Linux (RHEL) 上使用 .NET。Red Hat 和 Microsoft 开展协作，共同确保 .NET Core 能够在 RHEL 上正常运行。

Tizen 支持在 Tizen 平台上使用 .NET。

有关详细信息，请参阅 .NET Core 和 .NET 5 的版本和支持。

工具与工作效率

.NET 为用户提供了各种语言、集成开发环境 (IDE) 和其他工具的选择。

编程语言

.NET 支持三种编程语言：

- C#

  C#（读作“See Sharp”）是一种新式编程语言，不仅面向对象，还类型安全。C# 源于 C 语言系列，C、C++、Java 和 JavaScript 程序员很快就可以上手使用。

- F#

  F# 语言支持函数式、命令式、面向对象的编程模式。

- Visual Basic

  在 .NET 语言中，Visual Basic 的语法最接近于人类的普通用语，因此更易于学习。不同于 C# 和 F#（Microsoft 正在积极为 C# 和 F# 开发新功能），Visual Basic 语言是稳定的。Visual Basic 不受 Web 应用支持，但受 Web API 支持。

下面是 .NET 语言支持的一些功能：

- 类型安全
- 类型推理 - C#、F#、Visual Basic
- 泛型类型
- 委托
- Lambda
- 事件
- 异常
- 特性
- 异步代码
- 并行编程
- 代码分析器

IDE

.NET 的集成开发环境包括：

- Visual Studio

  仅在 Windows 上运行。具有广泛的内置功能，设计为可以与 .NET 一起使用。社区版对学生、开放源代码贡献者和个人免费。

- Visual Studio Code

  在 Windows、macOS 和 Linux 上运行。免费且开源。扩展可用于使用 .NET 语言。

- Visual Studio for Mac

  仅在 macOS 上运行。用于开发适用于 iOS、Android 和 Web 的 .NET 应用和游戏。

- GitHub Codespaces

  联机 Visual Studio Code 环境，当前为 beta 版本。

SDK 和运行时

.NET SDK 是一组用于开发和运行 .NET 应用程序的库和工具。

下载 .NET 时，可以选择 SDK 或 *运行时*，例如 .NET 运行时或 ASP.NET Core 运行时。在要准备运行 .NET 应用的计算机上安装运行时。在要用于开发的计算机上安装 SDK。下载 SDK 时，将自动获取运行时。

SDK 下载包括以下组件：

- .NET CLI。可用于本地开发和持续集成脚本的命令行工具。
- `dotnet` 驱动程序。用于运行依赖于框架的应用的 CLI 命令。
- Roslyn 和 F# 编程语言编译器。
- MSBuild 生成引擎。
- .NET 运行时。提供类型系统、程序集加载、垃圾回收器、本机互操作和其他基本服务。
- 运行时库。提供基元数据类型和基本实用程序。
- ASP.NET Core 运行时。为连接 Internet 的应用（如 Web 应用、IoT 应用和移动后端）提供基本服务。
- 桌面运行时。为 Windows 桌面应用（包括 Windows 窗体和 WPF）提供基本服务。

运行时下载包括以下组件：

- （可选）桌面或 ASP.NET Core 运行时。
- .NET 运行时。提供类型系统、程序集加载、垃圾回收器、本机互操作和其他基本服务。
- 运行时库。提供基元数据类型和基本实用程序。
- `dotnet` 驱动程序。用于运行依赖于框架的应用的 CLI 命令。

有关详细信息，请参阅以下资源：

- .NET SDK 概述
- .NET CLI 概述
- dotnet 命令

项目系统和 MSBuild

.NET 应用是使用 MSBuild 从源代码中生成的。项目文件（.csproj、.fsproj 或 .vbproj）指定目标和负责编译、打包和发布代码的关联任务 。有引用目标和任务的标准集合的 SDK 标识符。使用这些标识符有助于使项目文件较小且易于使用。例如，下面是控制台应用的一个项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

下面是 Web 应用的一个项目文件：

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

在这些示例中，`Project` 元素的 `Sdk` 属性指定一组 MSBuild 目标和生成项目的任务。`TargetFramework` 元素指定应用所依赖的 .NET 版本。可以编辑项目文件以添加特定于项目的其他目标和任务。

有关详细信息，请参阅 .NET 项目 SDK 概述和目标框架。

CI/CD

MSBuild 和 .NET CLI 可与各种连续集成工具和环境一起使用，例如：

- GitHub Actions
- Azure DevOps
- CAKE
- FAKE

有关详细信息，请参阅在持续集成 (CI) 中使用 .NET SDK 和工具

NuGet

NuGet 是为 .NET 设计的开源包管理器。NuGet 包是具有 `.nupkg` 扩展的 .zip 文件，此扩展包含编译代码 (DLL)、与该代码相关的其他文件以及描述性清单（包含包版本号等信息）。使用代码的开发人员共享创建包，并将其发布到 nuget.org 或专用主机。希望使用共享代码的开发人员将包添加到其项目中，然后可以在项目代码中调用包公开的 API。

有关详细信息，请参阅 NuGet 文档。

.NET Interactive

.NET Interactive 是一组 CLI 工具和 API，使用户能够跨 Web、markdown 和笔记本创建交互式体验。

有关详细信息，请参阅以下资源：

- .NET 浏览器内教程
- 在计算机上将 .NET 笔记本与 Jupyter 配合使用
- .NET Interactive 文档

执行模型

.NET 应用在称为“公共语言运行时 (CLR)”的运行时环境中运行托管代码。

CLR

.NET CLR 是包含 Windows、macOS 和 Linux 支持的跨平台运行时。CLR 处理内存分配和管理。CLR 也是一个虚拟机，不仅可执行应用，还可使用实时 JIT 编译器生成和编译代码。

有关详细信息，请参阅公共语言运行时 (CLR) 概述。

JIT 编译器和 IL

C# 等较高级的 .NET 语言编译为称为中间语言 (IL) 的硬件无关性指令集。应用运行时，JIT 编译器将 IL 转换为处理器可理解的计算机代码。JIT 编译发生在要运行代码的同一台计算机上。

由于 JIT 编译在应用程序的执行过程中发生，因此编译时间是运行时的一部分。因此，JIT 编译器需要平衡优化代码所花费的时间与生成代码时可节约的时间。但 JIT 编译器知道实际硬件，这样开发人员就无需为不同平台提供不同的实现。

.NET JIT 编译器可以执行分层编译，这意味着它可以在运行时重新编译各个方法。通过此功能，它可以快速编译，同时仍然能够为常用方法生成高度优化的代码版本。

有关详细信息，请参阅托管执行过程和分层编译。

AOT 编译器

大多数 .NET 工作负载的默认体验是 JIT 编译器，但 .NET 提供两种形式的预先 (AOT) 编译：

- 某些场景需要 100% AOT 编译。例如 iOS。
- 在其他情况下，应用的大多数代码都是 AOT 编译的，但有些代码是 JIT 编译的。某些代码模式不适用于 AOT（如泛型）。这种形式的 AOT 编译的示例为准备运行发布选项。这种形式的 AOT 具有 AOT 的优点并且没有 AOT 的缺点。

自动内存管理

垃圾回收器 (GC) 管理应用程序的内存分配和释放。每当代码新建对象时，CLR 都会从托管堆为对象分配内存。只要托管堆中有地址空间，运行时就会继续为新对象分配空间。没有足够的可用地址空间时，GC 将检查托管堆中应用程序不再使用的对象。然后回收该内存。

GC 是一种有助于确保内存安全的 CLR 服务。如果某个程序仅访问分配的内存，则该程序就是内存安全的。例如，运行时可确保应用不会访问超过数组边界的未分配内存。

有关详细信息，请参阅自动内存管理和垃圾回收的基础知识。

处理未托管的资源

有时，代码需要引用 *非托管资源*。未托管的资源是指不由 .NET 运行时自动维护的资源。例如，文件句柄就是未托管的资源。FileStream 对象是一个托管对象，但它引用未托管的文件句柄。用完 FileStream 之后，需要显式释放文件句柄。

在 .NET 中，引用未托管资源的对象会实现 IDisposable 接口。用完对象后，需调用此对象的 Dispose() 方法，该方法会释放所有托管资源。.NET 语言提供一种方便的 `using` 语句（C#、F#、VB），确保调用 `Dispose` 方法。

有关详细信息，请参阅清理非托管资源。

部署模型

可以在两种不同模式下发布 .NET 应用：

- 将应用作为独立应用，生成的可执行文件将包含 .NET 运行时和库，以及该应用程序及其依赖项。应用程序的用户可以在未安装 .NET 运行时的计算机上运行该应用程序。独立应用是特定于平台的，可以使用 AOT 编译形式进行选择性发布。
- 将应用作为依赖于框架的应用发布会生成一个可执行文件和多个二进制文件（.dll 文件），其中仅包括应用程序本身及其依赖项 。应用程序的用户必须单独安装 .NET 运行时。可执行文件是特定于平台的，但依赖于框架的应用程序的 .dll 文件是跨平台的。

  可以并行安装多个版本的运行时，以运行针对不同运行时版本的依赖于框架的应用。有关详细信息，请参阅目标框架。

可执行文件是针对特定目标平台生成的，你可以通过运行时标识符 (RID) 指定。

有关详细信息，请参阅 .NET 应用程序发布概述和 .NET 和 Docker 简介。

运行时库

.NET 具有一组广泛的标准类库，称为运行时库、框架库或基类库 (BCL)。这些库为许多常规用途类型和特定于工作负载的类型和实用工具功能提供实现。

下面是在 .NET 运行时库中定义的一些类型示例：

- 基元类型，如 System.Boolean 和 System.Int32。
- 集合，例如 System.Collections.Generic.List<T> 和 System.Collections.Generic.Dictionary<TKey,TValue>。
- 数据类型，例如 System.Data.DataSet 和 System.Data.DataTable。
- 网络实用程序类型，如 System.Net.Http.HttpClient。
- 文件和流 I/O 实用程序类型，如 System.IO.FileStream 和 System.IO.TextWriter。
- 序列化实用程序类型，例如 System.Text.Json.JsonSerializer 和 System.Xml.Serialization.XmlSerializer。
- 高性能类型，例如 System.Span<T>、System.Numerics.Vector 和 Pipelines。

有关详细信息，请参阅运行时库概述。库的源代码位于 GitHub dotnet/运行时存储库中。

运行时库的扩展

某些常用应用程序功能的库没有包含在运行时库中，但在 NuGet 包中提供，如下所示：

NUGET 包	功能
Microsoft.Extensions.Hosting	应用程序生存期管理（通用主机）
Microsoft.Extensions.DependencyInjection	依赖关系注入 (DI)
Microsoft.Extensions.Configuration	配置
Microsoft.Extensions.Logging	Logging
Microsoft.Extensions.Options	选项模式

有关详细信息，请参阅 GitHub 上的 dotnet/extensions 存储库。

数据访问

.NET 提供了对象/关系映射器 (ORM) 和一种在代码中编写 SQL 查询的方法。

Entity Framework Core

Entity Framework (EF) Core 是一种可用作 ORM 的开源和跨平台的数据访问技术。借助 EF Core，可以通过引用代码中的 .NET 对象来处理数据库。它减少了需要编写和测试的数据访问代码的数量。EF Core 支持许多数据库引擎。

有关详细信息，可参阅 Entity Framework Core 和数据库提供程序。

LINQ

通过语言集成查询 (LINQ)，可以编写用于数据操作的声明性代码。数据可采用多种形式（例如，内存中对象、SQL 数据库或 XML 文档），但针对每个数据源编写的 LINQ 代码往往没有差别。

有关详细信息，请参阅 LINQ（语言集成查询）概述。

.NET 术语

了解一些术语在一段时期内的用法变化有利于理解 .NET 文档。

.NET Core 和 .NET 5

在 2002 年，Microsoft 发布了 .NET Framework，这是用于创建 Windows 应用的开发平台。目前 .NET Framework 的版本为 4.8，并且仍由 Microsoft 支持。

2014 年，Microsoft 开始编写 .NET Framework 的跨平台开源后续产品。.NET 的这个新实现被命名为 .NET Core，直到发展到版本 3.1。.NET Core 3.1 的下一个版本是 .NET 5。版本号 4 被跳过，以避免 .NET 的此实现和 .NET Framework 4.8 混淆。删除名称“Core”以表明这是现在 .NET 的主要实现。

本文是关于 .NET 5 的，但 .NET 5 文档的许多内容仍然引用“.NET Core”或“.NET Framework”。此外，“Core”在名称 ASP.NET Core 和 Entity Framework Core 中保留。

该文档还引用 .NET Standard。.NET Standard 是一种 API 规范，可让你为 .NET 的多个实现开发类库。

有关详细信息，请参阅 .NET 体系结构组件。

重载术语

.NET 的一些术语可能会令人困惑，因为同一个术语在不同的上下文中的用法不同。下面是一些较明显的例子：

- **运行时**

公共语言运行时 (CLR)	用于托管程序的执行环境。OS 属于运行时环境，但不属于 .NET 运行时。
.Net 下载页上的 .NET 运行时	CLR 和运行时库，它们一起提供了对运行依赖于框架的应用的支持。此页还提供 ASP.NET Core 服务器应用和 Windows 桌面应用的运行时选项。
运行时标识符 (RID)	运行 .NET 应用的 OS 平台和 CPU 体系结构。例如：Windows x64、Linux x64。

- **框架**

.NET framework	.NET 的原始、仅限 Windows 的实现。“框架”首字母大写。
Target Framework — 目标 Framework	.NET 应用或库依赖的 API 集合。示例：.NET Core 3.1、.NET Standard 2.0
目标框架名字对象 (TFM)	TFM 是一种标准化令牌格式，用于指定 .NET 应用或库的目标框架。示例：`net462`（对于 .NET Framework 4.6.2）。
依赖于框架的应用	只能在从 .NET 下载页安装了运行时的计算机上运行的应用。此用法中的“框架”与你从 .NET 下载页下载的“运行时”是相同的。
框架库	有时用作运行时库的同义词。

- **SDK**

□□□	"SDK"□□
.NET 下载页上的 SDK	工具和库的集合，你下载和安装它们以开发和运行 .NET 应用。包括 CLI、MSBuild、.NET 运行时和其他组件。
SDK 样式项目	一组 MSBuild 目标和任务，用于指定如何为特定应用类型生成项目。这种情况下的 SDK 是使用项目文件中 `Project` 元素的 `Sdk` 属性指定的。

- **平台**

□□□	"□□"□□
跨平台	在此术语中，"平台"表示操作系统以及运行它的硬件，例如 Windows、macOS、Linux、iOS 和 Android。
.NET 平台	用法有所不同。引用可以是针对 .NET（如 .NET Framework 或 .NET 5）的一种实现，也可以是针对包括所有实现的 .NET 的总体概念。

- **CLI**

□□□□□□□□□	□□
命令行接口	用于开发、生成、运行和发布 .NET 应用程序的跨平台工具链。
公共语言基础结构	CLR 实施的规范。

有关 .NET 术语的详细信息，请参阅 .NET 术语表。

高级方案

以下各部分介绍在一些高级场景中非常有用的 .NET 功能。

原生互操作

每个操作系统都有一个应用程序编程接口 (API)，用于提供系统服务。.NET 提供多种方式来调用这些 API。

与原生 API 进行互操作的主要方式是使用"平台调用"，简称 P/Invoke。跨 Linux 和 Windows 平台支持 P/Invoke。进行互操作的一种仅限 Windows 的方式称为"COM 互操作"，用于在托管代码中操作 COM 组件。这种方式建立在 P/Invoke 基础结构之上，但工作原理略有不同。

有关详细信息，请参阅原生互操作性。

不安全代码

根据语言支持，CLR 可通过 `unsafe` 代码访问本机内存和执行指针算术运算。某些算法和系统互操作性需要这些操作。尽管不安全代码的功能强大，但除非有必要与系统 API 互操作或实现最高效的算法，否则不建议使用。在不同的环境中，不安全代码的执行方式可能不同，使用它还会丧失垃圾回收器和类型安全带来的好处。建议尽可能地限制和集中化使用不安全代码，并全面测试该代码。

有关详细信息，请参阅不安全代码和指针。

后续步骤

选择 .NET 教程

在浏览器中试用 .NET

C# 导览

F# 导览

.NET 体系结构组件

2021/11/16 •

.NET 应用开发用于并运行于一个或多个 .NET 实现 。.NET 实现包括 .NET Framework、.NET 5（和 .NET Core）以及 Mono。对于多个 .NET 实现，有一个名为 .NET Standard 的通用 API 规范。本文简要介绍了每个概念。

.NET Standard

.NET Standard 是一组由 .NET 实现的基类库实现的 API。更正式地说，它是构成协定统一集（这些协定是编写代码的依据）的特定 .NET API 组。这些协定在多个 .NET 实现中实现。

.NET Standard 是一个目标框架。如果代码面向 .NET Standard 版本，则它可在支持该 .NET Standard 版本的任何 .NET 实现上运行。

创建 .NET Standard 是为了支持跨不同 .NET 实现的可移植性，但现在 .NET 5 提供了一种更好的方法来跨多个平台和工作负载共享代码。有关详细信息，请参阅 .NET 5 和 .NET Standard。

.NET 实现

.NET 的每个实现都具有以下组件：

- 一个或多个运行时。示例：.NET Framework CLR、.NET 5 CLR。
- 一个类库。示例：.NET Framework 基类库、.NET 5 基类库。
- 可选择包含一个或多个应用程序框架。示例：ASP.NET、Windows 窗体和 Windows Presentation Foundation (WPF) 包含在 .NET Framework 和 .NET 5 中。
- 可包含开发工具。某些开发工具在多个实现之间共享。

Microsoft 支持以下四种 .NET 实现：

- .NET 5（和 .NET Core）及更高版本
- .NET Framework
- Mono
- UWP

.NET 5 现在是主要的实现，是正在进行的开发的重点。.NET 5 是基于单个代码基底进行构建的，该代码基底支持多个平台和许多工作负载，例如 Windows 桌面应用和跨平台控制台应用、云服务和网站。

.NET 5

.NET 5 是 .NET 的跨平台实现，专门设计用于处理大规模的服务器和云工作负载。它还支持其他工作负载，包括桌面应用。可在 Windows、macOS 和 Linux 上运行。它可实现 .NET Standard，因此面向 .NET Standard 的代码都可在 .NET 5 上运行。ASP.NET Core、Windows 窗体和 Windows Presentation Foundation (WPF) 都在 .NET 5 上运行。

有关更多信息，请参见以下资源：

- .NET 简介
- 为服务器应用选择 .NET 5 或 .NET Framework
- .NET 5 和 .NET Standard

.NET Framework

.Net Framework 是自 2002 年起就已存在的原始 .NET 实现。4.5 版以及更高版本实现 .NET Standard，因此面向 .NET Standard 的代码都可在这些版本的 .NET Framework 上运行。它还包含一些特定于 Windows 的 API，如通

过 Windows 窗体和 WPF 进行 Windows 桌面开发的 API。.NET Framework 非常适合用于生成 Windows 桌面应用程序。

有关详细信息，请参阅 .NET Framework 指南。

Mono

Mono 是主要在需要小型运行时使用的 .NET 实现。它是在 Android、macOS、iOS、tvOS 和 watchOS 上驱动 Xamarin 应用程序的运行时，且主要针对小内存占用。Mono 还支持使用 Unity 引擎生成的游戏。

它支持所有当前已发布的 .NET Standard 版本。

以前，Mono 实现更大的 .NET Framework API 并模拟一些 Unix 上最常用的功能。有时使用它运行依赖 Unix 上的这些功能的 .NET 应用程序。

Mono 通常与实时编译器一起使用，但它也提供在 iOS 之类的平台使用的完整静态编译器（预先编译）。

有关详细信息，请参阅 Mono 文档。

通用 Windows 平台 (UWP)

UWP 是用于为物联网 (IoT) 生成新式触控 Windows 应用程序和软件的 .NET 实现。它旨在统一可能想要以其为目标的不同类型的设备，包括电脑、平板电脑、电话，甚至 Xbox。UWP 提供许多服务，如集中式应用商店、执行环境 (AppContainer) 和一组 Windows API（用于代替 Win32 (WinRT)）。应用可采用 C++、C#、Visual Basic 和 JavaScript 编写。

有关详细信息，请参阅通用 Windows 平台简介。

.NET 运行时

运行时是用于托管程序的执行环境。操作系统属于运行时环境，但不属于 .NET 运行时。下面是 .NET 运行时的一些示例：

- .NET Framework 公共语言运行时 (CLR)
- .NET 5 公共语言运行时 (CLR)
- 适用于通用 Windows 平台的 .NET Native
- 用于 Xamarin.iOS、Xamarin.Android、Xamarin.Mac 和 Mono 桌面框架的 Mono 运行时

.NET 工具和常见基础结构

可访问一整套适用于每种 .NET 实现的工具和基础结构组件。这些工具和组件包括：

- .NET 语言及其编译器
- .NET 项目系统（基于 .csproj .vbproj 和 .fsproj 文件）
- MSBuild（用于生成项目的生成引擎）
- NuGet（适用于.NET 的 Microsoft 程序包管理器）
- 开放源生成业务流程工具，例如 CAKE 和 FAKE

有关详细信息，请参阅工具与工作效率。

适用标准

C# 语言和公共语言基础结构 (CLI) 规范通过 Ecma International® 进行标准化。这些标准的第一版已于 2001 年 12 月由 Ecma 发布。

这些标准的后续版本由编程委员技术委员会 (TC49) 的 TC49-TG2 (C#) 和 TC49-TG3 (CLI) 任务组编制，被 Ecma General Assembly 采纳，随后通过 ISO 快速跟踪流程被 ISO/IEC JTC 1 采纳。

最新标准

以下官方 Ecma 文档可用于 C# 和 CLI (TR-84)：

- **C# 语言标准（版本 5.0）**：ECMA-334.pdf
- **公共语言基础结构**：ECMA-335.pdf。
- **派生自分区 IV XML 文件的信息**：ECMA-084.pdf 格式。

官方 ISO/IEC 文档可从 ISO/IEC 公开标准页获取。以下链接可从该页面直接获得：

- **信息技术 - 编程语言 - C#**：ISO/IEC 23270:2018
- **信息技术 - 公共语言基础结构 (CLI) 分区 I 到 VI**：ISO/IEC 23271:2012
- **信息技术 - 公共语言基础结构 (CLI) - 有关派生自分区 IV XML 文件的信息的技术报告**：ISO/IEC TR 23272:2011

请参阅

- .NET 简介
- .NET Standard 简介
- 为服务器应用选择 .NET 5 或 .NET Framework
- .NET Framework 指南
- C# 指南
- F# 指南
- Visual Basic 指南

.NET 类库

2021/11/16 •

类库是 .NET 的共享库概念。通过类库可将实用功能组件化为可供多个应用程序使用的模块。还可使用类库加载应用程序启动时不需要或未知的功能。类库通过 .NET 程序集文件格式进行描述。

有三种类型的类库可供使用：

- **平台特定** 的类库可访问给定平台（例如，.NET Framework、Xamarin、iOS）中的所有 API，但只有面向该平台的应用和库可使用该类库。
- **可移植** 类库可访问 API 的子集，并且可供面向多个平台的应用和库使用。
- .NET Standard 类库将平台专用库概念和可移植库概念合并到一个模型中，以同时获取两方面的优势。

平台特定的类库

平台特定的类库绑定到单个 .NET 实现（例如，Windows 上的 .NET Framework），因此它可以在已知的执行环境上接收重要的依赖项。此类环境会公开一组已知 API（.NET 和 OS API），维护并公开预期状态（例如，Windows 注册表）。

创建平台特定的库的开发人员可充分利用基础平台。该库只会在给定平台上运行，因此不需要平台检查和其他形式的条件代码（针对多个平台取模单个源代码）。

平台特定的库一直是 .NET Framework 的主要类库类型。即使出现了其他 .NET 实现，平台特定的库也仍然是最主要的类库类型。

可移植类库

可移植库支持多个 .NET 实现。该库仍可在已知执行环境上接收依赖项，不过该环境是一种合成环境，由一组具体 .NET 实现的交集生成。公开的 API 和平台假设是平台特定的库可用的子集。

创建可移植库时，需选择平台配置。平台配置是需要支持的平台集（例如，.NET Framework 4.5+、Windows Phone 8.0+）。要支持的平台越多，可生成的 API 和平台假设就越少，公分母越小。这一特性可能最初会令人感到疑惑，因为人们常认为“越多越好”，但却发现更多的支持平台带来的可用 API 更少。

许多库开发人员已经从由一个源开发多个平台特定的库（使用条件编译指令）转向开发可移植库。有多种方法可在可移植库中访问平台特定的功能，其中“诱饵替换”是目前最广为接受的方法。

.NET Standard 类库

.NET Standard 库是平台特定的库和可移植库概念的替代。.NET Core 库可从基础平台公开所有功能（无合成平台或平台交集），就此而言，它是平台特定的库。该库可在所有支持平台上运行，就此而言，它是可移植库。

.NET Standard 公开一组库协定。.NET 实现必须完全支持每个协定，否则就全都不支持。因此，每个实现都支持一组 .NET Standard 协定。得出的必然结果是，.NET Standard 类库在支持其协定依赖项的平台上受到支持。

.NET Standard 不公开整个 .NET Framework 的功能（也不将此作为目标），但相比可移植类库，其公开的 API 更多。随着时间推移，将添加更多的 API。

下列平台支持 .NET Standard 库：

- .NET Core
- .NET Framework
- Mono

- Xamarin.iOS、Xamarin.Mac、Xamarin.Android
- 通用 Windows 平台 (UWP)
- Windows
- Windows Phone
- Windows Phone Silverlight

有关详细信息，请参阅 .NET Standard。

Mono 类库

Mono 支持多种类库，包括上述三种类型的库。Mono 常被（正确地）视为 .NET Framework 的跨平台实现。部分原因是，平台特定的 .NET Framework 库可在 Mono 运行时上运行，而无需修改或重新编译。创建可移植库之前，此特性就已存在，因此在 .NET Framework 和 Mono 之间启用二进制可移植性是显而易见的选择（虽然它只能单向运行）。

.NET Standard

2021/11/16 •

.NET Standard 是针对多个 .NET 实现推出的一套正式的 .NET API 规范。推出 .NET Standard 的背后动机是要提高 .NET 生态系统中的一致性。但是，.NET 5 采用不同的方法来建立一致性，这种新方法在很多情况下都不需要 .NET Standard。有关详细信息，请参阅本文后续部分的 .NET 5 和 .NET Standard。

.NET 实现支持

各种 .NET 实现以特定版本的 .NET Standard 为目标。每个 .NET 实现版本都会公布它所支持的最高 .NET Standard 版本，这种声明意味着它也支持以前的版本。例如，.NET Framework 4.6 实现 .NET Standard 1.3。也就是说，它会公开在 .NET Standard 版本 1.0 到 1.3 中定义的所有 API。同样，.NET Framework 4.6.1 实现 .NET Standard 1.4，而 .NET 5 则实现 .NET Standard 2.1。

下表列出了支持每个 .NET Standard 版本的最低实现版本。这意味着所列实现的更高版本也支持相应的 .NET Standard 版本。例如，.NET Core 2.1 及更高版本支持 .NET Standard 2.0 及更早版本。

.NET STANDARD	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0	5.0
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework [1]	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1 [2]	4.6.1 [2]	4.6.1 [2]	N/A[3]
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
通用 Windows 平台	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	待定
Unity	2018 年 1 月	2018 年 1 月	2018 年 1 月	2018 年 1 月	2018 年 1 月	2018 年 1 月	2018 年 1 月	2018 年 1 月	2021.2.0b6

1 针对 .NET framework 列出的版本适用于 .NET Core 2.0 SDK 和更高版本的工具。旧版本对 .NET Standard 1.5 及更高版本使用了不同映射。如果无法升级到 Visual Studio 2017 或更高版本，可下载适用于 Visual Studio 2015 的 .NET Core 工具。

2 此处所列的版本表示 NuGet 用于确定给定 .NET Standard 库是否适用的规则。虽然 NuGet 将 .NET Framework 4.6.1 视为支持 .NET Standard 1.5 到 2.0，但使用为从 .NET Framework 4.6.1 项目构建的 .NET Standard 库存在一系列问题。对于需要使用此类库的 .NET Framework 项目，建议将项目升

级到面向 .NET Framework 4.7.2 或更高版本。

3 .NET Framework 不支持 .NET Standard 2.1。有关详细信息，请参阅 .NET Standard 2.1 公告。

- 列表示 .NET Standard 版本。每个标题单元格都是一个文档链接，其中介绍了相应版本的 .NET Standard 中新增了哪些 API。
- 行表示不同的 .NET 实现。
- 各单元格中的版本号指示要定向到此 .NET Standard 版本所需的最低实现版本。
- 有关交互式表的信息，请参阅 .NET Standard 版本。

若要查找可以定位的 .NET Standard 最高版本，请按照以下步骤操作：

1. 查找要运行的 .NET 实现所在的行。
2. 在这一行中从右向左查找可以定位的 .NET Standard 版本所在的列。
3. 列标题指示目标支持的 .NET Standard 版本。此外可以面向任何更低的 .NET Standard 版本。更高版本的 .NET Standard 还支持实现。
4. 对要定位的每个平台重复执行此过程。如果有多个目标平台，应选择它们都支持的最高版本。例如，如果要在 .NET Framework 4.8 和 .NET 5 上运行，可以使用的最高 .NET Standard 版本是 .NET Standard 2.0。

要定位哪个 .NET Standard 版本

建议定位 .NET Standard 2.0，除非你需要支持早期版本。最常规用途的库应该不需要除 .NET Standard 2.0 之外的其他 API。所有新式平台都支持 .NET Standard 2.0，并且它是支持具有一个目标的多个平台的推荐方法。

如果需要定位 .NET Standard 1.x，建议还定位 .NET Standard 2.0。.NET Standard 1.x 作为一组精细的 NuGet 包分发，它创建了一个大型的包依赖项关系图，并导致开发人员在构建时下载大量的包。有关详细信息，请参阅本文后续部分的跨平台定位和 .NET 5 和 .NET Standard。

.NET Standard 版本控制规则

版本控制规则主要有两个：

- 累加性：.NET Standard 版本在逻辑上形成同心圆。也就是说，较高的版本包含较低版本的所有 API。版本之间没有重大更改。
- 不可变：一旦发布，.NET Standard 版本就会冻结起来。

在 .NET Standard 2.1 版本之后，将不会有新版本。有关详细信息，请参阅本文后续部分的 .NET 5 和 .NET Standard。

规范

.NET Standard 规范是一组标准化的 API。此规范由 .NET 实现者（具体而言，由 Microsoft（包括 .NET Framework、NET Core 和 Mono）和 Unity）进行维护。

正式项目

正式规范是一组用于定义标准中包含的 API 的 .cs 文件。dotnet/standard 存储库中的 Ref 目录定义了 .NET Standard API。

NETStandard.Library 元包（源代码）描述用于部分定义一个或多个 .NET Standard 版本的库集。

给定的组件（如 `System.Runtime`）描述：

- .NET Standard 的一部分（即其范围）。
- .NET Standard 在此范围内的多个版本。

标准库提供派生项目以方便读取，并实现某些开发人员方案（例如，使用编译器）。

- Markdown 中的 API 列表。
- 引用程序集，以 NuGet 包的形式分发，由 NETStandard.Library 元包引用。

包表示形式

.NET Standard 引用程序集的主要分发载体是 NuGet 包。实现会以适用于每个 .NET 实现的各种方式提供。

NuGet 包面向一个或多个框架。.NET Standard 包面向".NET Standard"框架。可以使用 `netstandard` 精简 TFM（例如 `netstandard1.4`）来设定 .NET Standard 框架作为目标。计划在多个 .NET 实现上运行的库应将此框架作为目标。对于最广泛的 API 集，将 `netstandard2.0` 设定为目标，因为 .NET Standard 2.0 的可用 API 数量比 .NET Standard 1.6 的两倍还多。

`NETStandard.Library` 元包引用定义 .NET Standard 的一整套 NuGet 包。要指定 `netstandard` 作为目标，最常见的方法是引用此元包。它描述并提供了对大约 40 个 .NET 库及定义 .Net Standard 的相关 API 的访问权限。可以引用以 `netstandard` 为目标的其他包来使用其他 API。

版本管理

规范并不是单一的，而是一组以线性方式进行版本控制的 API。该标准的第一个版本建立了一组基准 API。后续版本将添加 API，并继承以前的版本定义的 API。在从 Standard 中移除 API 方面，并没有成文的规定。

.NET Standard 并不特定于任何一种 .NET 实现，也不与其中任一实现的版本控制方案匹配。

如前所述，在 .NET Standard 2.1 版本之后，将不会有新版本。

定位 .NET Standard

可以结合使用 `netstandard` 框架和 `NETStandard.Library` 元包来构建 .NET Standard 库。

.NET Framework 兼容性模式

从 .NET Standard 2.0 开始，引入了 .NET Framework 兼容性模式。此兼容性模式允许 .NET Standard 项目引用 .NET Framework 库，就像其针对 .NET Standard 编译一样。引用 .NET Framework 库并不适用于所有项目，，例如使用 Windows Presentation Foundation (WPF) API 的库。

有关详细信息，请参阅 .NET Framework兼容性模式。

.NET Standard 库和 Visual Studio

要在 Visual Studio 中生成 .NET Standard 库，请确保 Windows 上已安装 Visual Studio 2019 或 Visual Studio 2017 版本 15.3 或更高版本，或 macOS 上已安装 Visual Studio for Mac 版本 7.1 或更高版本。

如果项目中只需使用 .NET Standard 2.0 库，也可在 Visual Studio 2015 中执行此操作。但是需要安装 NuGet client 3.6 或更高版本。可从 NuGet 下载页面下载适用于 Visual Studio 2015 的 NuGet 客户端。

.NET 5 和 .NET Standard

.NET 5 是 Microsoft 正在积极开发的 .NET 实现。它是具有一组统一功能和 API 的单一产品，可用于 Windows 桌面应用和跨平台控制台应用、云服务和网站。.NET 5 TFM 反映了以下广泛的应用场景：

- `net5.0`

  此 TFM 适用于在任何位置运行的代码。它仅包括跨平台工作的技术（少数例外情况除外）。对于 .NET 5 代码，`net5.0` 替换了 `netcoreapp` 和 `netstandard` TFM。

- `net5.0-windows`

  这是一个特定于 OS 的 TFM 的示例，该 TFM 可向 `net5.0` 引用的所有内容添加特定于 OS 的功能。

以 net5.0 为目标与以 netstandard 为目标的情形

对于以 `netstandard` 为目标的现有代码，无需将 TFM 更改为 `net5.0`。.NET 5 可实现 .NET Standard 2.1 及更低版本。将目标从 .NET Standard 更改为 .NET 5 的唯一原因是获取对更多运行时功能、语言功能或 API 的访问权

限。例如，为了使用 C# 9，需要以 .NET 5 为目标。可同时以 .NET 5 和 .NET Standard 为目标来访问较新的功能，并仍然可将库供其他 .NET 实现使用。

下面是适用于 .NET 5 新代码的一些准则：

- 应用组件

  如果要使用库将应用程序分解成多个组件，建议以 `net5.x` 作为目标，其中 `5.x` 是应用程序可将其作为目标的最早的 .NET 5 版本。为简单起见，最好在同一 .NET 版本中保留构成应用程序的所有项目。然后，可以假设任何位置均具有相同的 BCL 功能。

- 可重用的库

  如果要构建计划在 NuGet 上发布的可重用的库，请考虑在覆盖范围和可用功能集之间进行权衡。.NET Standard 2.0 是 .NET Framework 支持的最新版本，因此它具有相当大的功能集，可提供良好的覆盖范围。我们建议你不要以 .NET Standard 1.x 作为目标，因为这样会限制可用的功能集，使覆盖范围的增幅降至最低。

  如果你不需要支持 .NET Framework，可以选择 .NET Standard 2.1 或 .NET 5。我们建议你跳过 .NET Standard 2.1，而直接选择 .NET 5。大多数广泛使用的库最终都将同时以 .NET Standard 2.0 和 .NET 5 作为目标。支持 .NET Standard 2.0 可提供最大的覆盖范围，而支持 .NET 5 可确保你可以为已使用 .NET 5 的客户利用最新的平台功能。

.NET Standard 的问题

下面是一些关于 .NET Standard 的问题，这些问题有助于解释为什么最好使用 .NET 5 来跨平台和工作负载共享代码：

- 添加新 API 的速度缓慢

  .NET Standard 是作为所有 .NET 实现都必须支持的 API 集创建的，因此会对添加新 API 的建议进行审核。目标是仅标准化可在所有当前和未来的 .NET 平台中实现的 API。因此，如果某个功能错过了特定版本，则你可能需要等待几年，该功能才会被添加到 Standard 版本中。然后，你需要等待更长的时间，新版本的 .NET Standard 才能受到广泛支持。

  .NET 5 中的解决方案：实现某项功能时，该功能便已可供所有 .NET 5 应用和库使用，因为代码基底是共享的。由于 API 规范与其实现之间没有区别，因此相较于使用 .NET Standard，你可以更快地利用新功能。

- 复杂的版本控制

  API 规范与其实现的分离导致 API 规范版本与实现版本之间出现复杂的映射。这种复杂性在本文前面显示的表以及其解释方式说明中显而易见。

  .NET 5 中的解决方案：.NET 5.x API 规范与其实现之间不存在任何分离。由此实现了一个简化的 TFM 方案。提供了一个适用于所有工作负载的 TFM 前缀：`net5.0` 用于库、控制台应用和 Web 应用。唯一的变化是针对特定平台指定特定于平台的 API 的后缀，例如 `net5.0-windows`。由于此 TFM 命名约定，你可以轻松判断给定应用是否可以使用给定的库。不需要版本号等效表（如 .NET Standard 的版本号等效表）。

- 运行时出现不受平台支持的异常

  .NET Standard 公开了特定于平台的 API。代码在编译时可能不会出错，并且看起来可以移植到任何平台（即该代码不可移植也是如此）。当它在不具有给定 API 实现的平台上运行时，会出现运行时错误。

  .NET 5 中的解决方案：.NET 5 SDK 包括默认启用的代码分析器。平台兼容性分析器会检测对你打算在其上运行的平台所不支持的 API 的意外使用情况。有关详细信息，请参阅平台兼容性分析器。

未弃用 .NET Standard

对于可由多个 .NET 实现使用的库，仍需要 .NET Standard。在以下情况下，建议以 .NET Standard 作为目标：

- 使用 `netstandard2.0` 在 .NET Framework 和 .NET 的所有其他实现之间共享代码。

- 使用 `netstandard2.1` 在 Mono、Xamarin 和 .NET Core 3.x 之间共享代码。

另请参阅

- .NET Standard 版本（源）
- .NET Standard 版本（交互式 UI）
- 生成 .NET Standard 库
- 跨平台定位

.NET Core 和 .NET 5 的版本和支持

2021/11/16 •

了解 .NET 5（包括 .NET Core）和更高版本的版本、补丁和支持。本文介绍版本类型、服务更新、SDK 功能区段、支持期限和支持选项。

版本类型

关于各版本类型的信息在版本号中以 major.minor.patch 格式进行编码。

例如：

- .NET Core 3.0 和 NET 5.0 是主要版本。
- .NET Core 3.1 是 .NET Core 3.0 主要版本后的第一个次要版本。
- .NET Core 3.1.7 是 .NET Core 3.1 的第七个补丁。

主要版本

主要版本包括新增功能、新的公共 API 图面和 bug 修复。示例包括 .NET Core 3.0 和 .NET 5。由于变更的性质，这些版本预计包含中断性变更。主要版本与先前的主要版本并行安装。

次要版本

次要版本也包括新功能、公共 API 图面和 bug 修复，还可能包含中断性变更。示例包括 .NET Core 2.1 和 .NET Core 3.1。次要版本与主要版本的区别在于变更量较小。应用程序从 .NET Core 3.0 升级到 3.1 所经历的变更较小。次要版本与先前的次要版本并行安装。

服务更新

几乎每个月都会发布服务更新（补丁），这些更新既包含安全性 bug 修复，又包含非安全性 bug 修复。例如，.NET Core 3.1.8 是 .NET Core 3.1 的第八个更新。如果这些更新包含安全性修复，则会在“星期二修补日”发布，该日期始终为每月的第二个星期二。服务更新应保持兼容性。从 .NET Core 3.1 开始，服务更新会删除之前的更新。例如，成功安装最新的 3.1 服务更新会删除之前的 3.1 更新。

功能区段（仅 SDK）

.NET SDK 的版本控制与 .NET 运行时略有不同。为了与新的 Visual Studio 版本保持一致，.NET SDK 更新有时包含新的功能或新版组件（如 MSBuild 和 NuGet）。这些新功能或组件可能与早期 SDK 更新中针对同一主要或次要版本发布的功能或组件版本不兼容。

为了区分此类更新，.NET SDK 使用功能区段的概念。例如，第一个 .NET Core 3.1 SDK 是 3.1.100。此版本对应于 3.1.1xx 功能区段。功能区段在版本号第三部分的百数组中定义。例如，3.1.101 和 3.1.201 属于两个不同功能区段的版本，而 3.1.101 和 3.1.199 则属于同一个功能区段。安装 .NET Core SDK 3.1.101 时，会从计算机中删除 .NET Core SDK 3.1.100（如果存在）。当在同一台计算机上安装 .NET Core SDK 3.1.200 时，不会删除 .NET Core SDK 3.1.101。

运行时前滚和兼容性

主要和次要更新与先前的版本并行安装。针对特定 major.minor 版本构建的应用程序会继续使用该目标运行时，即使安装了较新版本也是如此。应用不会自动前滚以使用运行时的较新 major.minor 版本，除非你选择启用此行为。针对 .NET Core 3.0 构建的应用程序不会在 .NET Core 3.1 上自动开始运行。建议在部署到生产环境之前，重新生成应用并针对较新的主要或次要运行时版本进行测试。有关详细信息，请参阅依赖于框架的应用前滚和独立部署运行时前滚。

服务更新的处理方式与主要和次要版本不同。默认情况下，针对 .NET Core 3.1 构建的应用程序在 3.1.0 运行时上运行。安装服务更新后，该应用程序会自动前滚，以使用较新的 3.1.1 运行时。此行为是默认行为，因为我们

希望在安装安全性修复后立即使用这些修复，而不需要执行任何其他操作。你可以选择禁用此默认前滚行为。

.NET Core 和 .NET 5 版本生命周期

.NET Core、.NET 5 及更高版本采用 新式生命周期，摒弃了 .NET Framework 版本所采用的固定生命周期。具有固定生命周期的产品提供较长的固定支持期，例如 5 年主要支持和 5 年外延支持。主要支持包括安全性修复和非安全性修复，而外延支持仅提供安全性修复。采用新式生命周期的产品具有更类似于服务的支持模型，支持周期更短，发布频率更高。

版本跟踪

版本有两种支持跟踪：

- 当前版本

  在下一个主要或次要版本发布后 3 个月内，支持这些版本。

  示例：

  - .NET Core 3.0 于 2019 年 9 月发布，随后于 2019 年 12 月发布 .NET Core 3.1。
  - .NET Core 3.0 支持于 3.1 发布后 3 个月（即 2020 年 3 月）结束。
- 长期支持 (LTS) 版本

  这些版本至少支持 3 年，或下一个 LTS 版本发布后 1 年（以较晚发生者为准）。

  示例：

  - .NET Core 2.1 于 2018 年 5 月发布，并于 2018 年 8 月被认定为 LTS 版本。
  - .NET Core 3.1 是下一个 LTS 版本，于 2019 年 12 月发布。
  - 由于 2021 年 8 月（3 年）晚于 2020 年 12 月（3.1 版本发布后一年），因此对 .NET Core 2.1 的支持将持续到 2021 年 8 月。

版本在 LTS 与当前之间交替，因此较早的版本可能比较晚的版本受支持的时间更长。例如，.NET Core 2.1 是支持时间持续到 2021 年 8 月的 LTS 版本。3.0 版本在一年多以后发布，但在 2019 年 12 月结束支持，这早于 2.1。

服务更新每月发布一次，包括安全性和非安全性（可靠性、兼容性和稳定性）修复。服务更新的支持时间持续到下一次发布服务更新。服务更新具有运行时前滚行为。这意味着应用程序默认在最新安装的运行时服务更新上运行。

如何选择版本

如果你要构建服务并希望继续定期更新服务，则当前版本（如 .NET 5）可能是最佳选择，它可让你持续获得 .NET 的最新功能。

如果你要构建将分发给使用者的客户端应用程序，稳定性可能比最新功能的可获得性更重要。在使用者可以升级到应用程序的下一个版本之前，可能需要在一段时间内支持你的应用程序。在这种情况下，LTS 版本（如 .NET Core 3.1）可能是正确的选择。

服务更新

.NET 服务更新的支持时间持续到下一次发布服务更新。服务更新每月发布一次。

你需要定期安装服务更新以确保应用处于安全且受支持的状态。例如，如果 .NET Core 3.1 的最新服务更新为 3.1.8，而我们发布了 3.1.9，则 3.1.8 不再是最新版本。3.1 的受支持服务级别为 3.1.9。

有关每个主要和次要版本的最新服务更新信息，请参阅 .NET 下载页面。

结束支持

结束支持日期是指 Microsoft 不再为产品版本提供修复、更新或技术协助的日期。在此日期之前，请确保你已升

级为使用受支持的版本。不受支持的版本不再能接收保护应用程序和数据的安全性更新。

支持的操作系统

.NET 5（和 .NET Core）及更高版本可在各种操作系统上运行。每个操作系统都有一个生命周期，由发起组织（例如 Microsoft、Red Hat 或 Apple）定义。在添加和删除操作系统版本支持时，我们将考虑这些生命周期计划。

当操作系统版本不受支持时，我们将停止测试该版本并停止对该版本提供支持。用户需要升级到受支持的操作系统版本才能获得支持。

有关详细信息，请参阅 .NET OS 生命周期策略。

获取支持

你可以选择 Microsoft 辅助支持或社区支持。

Microsoft 支持

若需要辅助支持，请与 Microsoft 支持专业人员联系。

你需要处于受支持的服务级别（最新可用的服务更新）才能获得支持。如果系统运行 3.1，而已发布 3.1.8 服务更新，那么第一步需要安装 3.1.8。

社区支持

有关社区支持，请参阅社区页面。

另请参阅

有关详细信息（包括各 .NET Core 版本和 .NET 5 受支持的日期范围），请参阅支持策略。

.NET 术语表

2021/11/16 •

此术语表的主要目的是阐明所选术语和缩写词的含义，这些词频繁出现在 .NET 文档中。

AOT

预编译器。

与 JIT 类似，此编译器还可将 IL 转换为机器代码。与 JIT 编译相比，AOT 编译在应用程序执行前进行并且通常在不同计算机上执行。AOT 工具不会在运行时进行编译，因此它们不需要最大程度地减少编译所花费的时间。这意味着它们可花更多的时间进行优化。由于 AOT 的上下文是整个应用程序，因此 AOT 编译器还会执行跨模块链接和全程序分析，这意味着之后会进行所有引用并会生成单个可执行文件。

请参阅 CoreRT 和 .NET Native。

应用模型

特定于工作负载的 API。下面是一些示例：

- ASP.NET
- ASP.NET Web API
- 实体框架 (EF)
- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- Windows 窗体 (WinForms)

ASP.NET

随 .NET Framework 一起提供的原始 ASP.NET 实现，也称为 ASP.NET 4.x。

有时，ASP.NET 是指原始 ASP.NET 和 ASP.NET Core 的涵盖性术语。该术语在任何给定实例中的含义取决于上下文。若要指明不要使用 ASP.NET 来表示这两种实现，请参考 ASP.NET 4.x。

请参阅 ASP.NET 文档。

ASP.NET Core

一种跨平台、高性能的开放源代码 ASP.NET 实现。

请参阅 ASP.NET Core 文档。

程序集 (assembly)

.dll 或 .exe 文件，其中包含一组可由应用程序或其他程序集调用的 API 。

程序集可以包括接口、类、结构、枚举和委托等类型。有时，项目的 bin 文件夹中的程序集被称为二进制文件。另请参阅库。

BCL

基类库。

一组构成 System.*（在一定的程度上构成 Microsoft.*）命名空间的库。BCL 是用于生成 ASP.NET Core 等较高级应用程序框架的较低级通用框架。

.NET 5（和 .NET Core）及更高版本的 BCL 的源代码包含在 .NET 运行时存储库中。这些 BCL API 中的大多数也可以在 .NET Framework 中获取，因此可将此源代码视为 .NET Framework BCL 源代码的一个分支。

以下术语通常指 BCL 引用的相同 API 集合：

- 核心 .NET 库
- 框架库
- 运行时库
- 共享框架

CLR

公共语言运行时。

具体含义依赖于上下文。公共语言运行时通常指 .NET Framework 或 .NET 5（和 .NET Core）及更高版本的运行时。

CLR 处理内存分配和管理。CLR 也是一种虚拟机，不仅可执行应用，还可使用 JIT 编译器快速生成和编译代码。

适用于 .NET Framework 的 CLR 实现仅限于 Windows。

.NET 5 及更高版本的 CLR 实现（也称为 Core CLR）是基于与 .NET Framework CLR 相同的代码库而构建的。Core CLR 最初是 Silverlight 的运行时，并且设计为在多个平台上运行，尤其是在 Windows 和 OS X 上运行。它仍是跨平台运行时，现在包括对许多 Linux 分发的支持。

另请参阅运行时。

Core CLR

.NET 5（和 .NET Core）及更高版本的公共语言运行时。

请参阅 CLR。

CoreRT

与 CLR 相比，CoreRT 不是虚拟机，这意味着它不包含用于快速生成并运行代码的功能，因为它不包括 JIT。但它包含 GC 以及运行时类型标识 (RTTI) 和反射功能。只是由于设计有类型系统，因此并不需要元数据反射功能。不需要元数据使它具有 AOT 工具链，该工具链可去除多余的元数据，更重要的是可识别应用不使用的代码。CoreRT 正在开发中。

请参阅 CoreRT 简介和 .NET 运行时实验室。

跨平台

能够开发并执行可在多个不同操作系统（如 Linux、Windows 和 iOS）上使用的应用程序，而无需专门针对每个操作系统进行重新编写。这样，可以在不同平台上的应用程序之间重复使用代码并保持一致性。

请参阅平台。

生态系统

所有针对给定技术生成和运行应用程序的运行时软件、开发工具和社区资源。

在所包含的第三方应用和库方面，术语“.NET 生态系统”不同于类似的“.NET 堆栈”等术语。请看以下这一句示例：

- “推出 .NET Standard 的背后动机是要提高 .NET 生态系统中的一致性。”

框架

一般指一个综合 API 集合，便于开发和部署基于特定技术的应用程序。从此常规意义上来说，ASP.NET Core 和 Windows 窗体都是示例应用程序框架。框架和库通常作同义词使用。

“框架”一词在以下术语中有不同的含义：

- 框架库
- .NET Framework
- 共享框架
- 目标框架
- TFM（目标框架名字对象）
- 依赖于框架的应用

有时，“框架”指的是 .NET 实现。例如，某文章可能会将 .NET 5 称为框架。

框架库

含义取决于上下文。可指适用于 .NET 5（和 .NET Core）及更高版本的框架库，在这种情况下，它指 BCL 引用的相同的库。它也可指在 BCL 上构建并为 Web 应用提供其他 API 的 ASP.NET Core 框架库。

GC

垃圾回收器。

垃圾回收器是自动内存管理的实现。GC 可释放不再使用的对象占用的内存。

请参阅垃圾回收。

IL

中间语言。

C# 等较高级的 .NET 语言编译为称为中间语言 (IL) 的硬件无关性指令集。IL 有时被称为 MSIL (Microsoft IL) 或 CIL（通用 IL）。

JIT

实时编译器。

与 AOT 类似，此编译器将 IL 转换为处理器可理解的计算机代码。与 AOT 不同，JIT 编译在需要运行代码的同一台计算机上按需执行。由于 JIT 编译在应用程序的执行过程中发生，因此编译时是运行时的一部分。因此，JIT 编译器需要平衡优化代码所花费的时间与生成代码时可节约的时间。但 JIT 知道实际硬件，这样开发人员就无需提供不同的实现。

.NET 实现

.NET 的实现包括：

- 一个或多个运行时。示例：CLR、CoreRT。
- 实现 .NET Standard 的某版本并且可能包含其他 API 的类库。示例：.NET Framework 和 .NET 5（和 .NET Core）及更高版本的 BCL。

- 可选择包含一个或多个应用程序框架。示例：ASP.NET、Windows 窗体及 WPF 包含在 .NET Framework 和 .NET 5 中。
- 可包含开发工具。某些开发工具在多个实现之间共享。

.NET 实现的示例：

- .NET Framework
- .NET 5（和 .NET Core）及更高版本
- 通用 Windows 平台 (UWP)
- Mono

库

可由应用或其他库调用的 API 集合。.NET 库由一个或多个程序集组成。

词库和框架通常作同义词使用。

Mono

Mono 是主要在需要小型运行时的时候使用的开放源、跨平台 .NET 实现。它是在 Android、Mac、iOS、tvOS 和 watchOS 上为 Xamarin 应用程序提供技术支持的运行时，主要针对内存占用少的应用程序。

它支持所有当前已发布的 .NET Standard 版本。

以前，Mono 实现更大的 .NET Framework API 并模拟一些 Unix 上最常用的功能。有时使用它运行依赖 Unix 上的这些功能的 .NET 应用程序。

Mono 通常与实时编译器一起使用，但它也提供在 iOS 之类的平台使用的完整静态编译器（预先编译）。

请参阅 Mono 文档。

.NET

- 一般情况下，.NET 是指 .NET Standard 和所有 .NET 实现及工作负载的涵盖性术语。
- 更具体地说，.NET 是指为所有新开发推荐的 .NET 实现：.NET 5（和 .NET Core）及更高版本）。

例如，第一个含义适用于“.NET 实现”等短语。第二个含义适用于 .NET SDK 和 .NET CLI 等名称。

.NET 始终采用全大写形式，请勿使用“.Net”。

请参阅 .NET 文档

.NET 5 及更高版本

一种跨平台、高性能的开放源 .NET 实现。包括公共语言运行时 (CLR)、AOT 运行时（正在开发中的 CoreRT）、基类库 (BCL) 以及 .NET SDK。

此 .NET 实现的早期版本称为 .NET Core。.Net 5 是继 .NET Core 3.1 之后的下一版本。跳过了版本 4，以避免将此较新的 .NET 实现与称为 .NET Framework 的旧实现混淆。.NET Framework 的当前版本为版本 4.8。

请参阅 .NET 文档。

.NET CLI

用于开发适用于 .NET 5（和 .NET Core）及更高版本的应用程序和库的跨平台工具链。也称为 .NET Core CLI。

请参阅 .NET CLI。

.NET Core

请参阅 .NET 5 及更高版本。

.NET Framework

仅在 Windows 上运行的 .NET 实现。包括公共语言运行时 (CLR)、基类库 (BCL) 以及应用程序框架库（例如 ASP.NET、Windows 窗体和 WPF）。

请查阅 .NET Framework 指南。

.NET Native

编译器工具链，可预先 (AOT) 生成，而非实时 (JIT) 生成本机代码。

编译采用与 C++ 编译器和链接器类似的工作方式在开发人员计算机上进行。它删除了未使用的代码，留出更多时间进行优化。它从库中提取代码，将它们合并到可执行文件中。结果是表示整个应用的单个模块。

UWP 是 .NET Native 支持的应用程序框架。

请参阅 .NET Native 文档。

.NET SDK

一组库和工具，开发人员可用其创建适用于 .NET 5（和 .NET Core）及更高版本的 .NET Core 应用程序和库。也称为 .NET Core SDK。

包括用于生成应用的 .NET CLI、用于生成和运行应用的 .NET 库以及用于运行 CLI 命令和运行应用程序的 dotnet 可执行文件 (dotnet.exe)。

请参阅 .NET SDK 概述。

.NET Standard

在每个 .NET 实现中都可用的 .NET API 正式规范。

.NET Standard 规范有时被称为库。由于库不仅包括规范（接口），还包括 API 实现，所以会误将 .NET Standard 称为“库”。

请参阅 .NET Standard。

NGEN

本机（映像）生成。

可将此方法视为永久性 JIT 编译器。它通常在执行代码的计算机上编译该代码，但通常在安装时进行编译。

包

NuGet 包 — 或只是一个包 — 是一个 .zip 文件，其中具有一个或多个名称相同的程序集以及作者姓名等其他元数据。

.zip 文件的扩展名为 .nupkg，且可以包含在多个目标框架和版本中使用的资产（如 .dll 文件和 .xml 文件）。在应用或库中安装时，会根据应用或库指定的目标框架选择相应的资产。定义接口的资产位于 ref 文件夹，而定义实现的资产位于 lib 文件夹。

平台

操作系统以及运行它的硬件，例如 Windows、macOS、Linux、iOS 和 Android。

下面是在句子中使用的示例：

- ".NET Core 是一个跨平台 .NET 实现。"
- "PCL 配置文件代表 Microsoft 平台，而 .NET Standard 与平台无关。"

旧的 .NET 文档有时使用".NET 平台"来表示一个 .NET 的实现或包括所有实现的 .NET 堆栈。这两种用法往往会与主（OS/硬件）含义混淆，因此我们要尽量避免这些用法。

"平台"在短语"开发人员平台"中有不同的含义，这里指的是提供用于生成和运行应用的工具和库的软件。.NET 是跨平台的开放源代码开发人员平台，用于构建多种不同类型的应用程序。

Runtime — 运行时

通常是指用于托管程序的执行环境。操作系统属于运行时环境，但不属于 .NET 运行时。下面是此情况下 .NET 运行时的一些示例：

- 公共语言运行时 (CLR)
- .NET Native（适用于 UWP）
- Mono 运行时

"运行时"一词在某些上下文中有不同的含义：

- .NET 5 下载页上的 .NET 运行时。

  可下载 .NET 运行时或其他运行时，如 ASP.NET Core 运行时 。在此用法中运行时是一组必须安装在计算机上的组件，这样才能在计算机上运行依赖于框架的应用。.NET 运行时包括 CLR 和 .NET 共享框架，后者提供 BCL。

- .NET 运行时库

  指 BCL 引用的相同的库。但是，其他运行时（例如 ASP.NET Core 运行时）具有不同的共享框架，并具有在 BCL 上构建的其他库。

- 运行时标识符 (RID)。

  此处的"运行时"表示运行 .NET 应用的 OS 平台和 CPU 体系结构，例如：`linux-x64`。

- 有时在 .NET 实现的意义上使用"运行时"，如以下示例中所示：

  - "各种 .NET 运行时实现特定版本的 .NET Standard。... 每个 .NET 运行时版本都将会公布它所支持的最高 .NET Standard 版本..."
  - "计划在多个运行时上运行的库应将此框架作为目标。"（参阅 .NET Standard）

共享框架

含义取决于上下文。.NET 共享框架指 .NET 运行时中包含的库。在这种情况下，适用于 .NET 5（和 .NET Core）及更高版本的共享框架指 BCL 引用的相同的库。

也存在其他共享框架。ASP.NET Core 共享框架指 ASP.NET Core 运行时中包含的库，其中包括 BCL 以及供 Web 应用使用的其他 API。

对于依赖于框架的应用，共享框架由库组成，这些库包含在运行应用的计算机上的文件夹中的程序集内。对于自包含应用，共享框架程序集包含在该应用中。

有关详细信息，请参阅深入了解 .NET Core 基元，第 2 部分：共享框架。

堆栈

一组编程方法，一起用于生成并运行应用程序。

".NET 堆栈"指 .NET Standard 和所有 .NET 实现。短语"一个 .NET 堆栈"可能指一种 .NET 实现。

Target Framework — 目标 Framework

.NET 应用或库依赖的 API 集合。

应用或库可将某版本的 .NET Standard（例如 .NET Standard 2.0）作为目标，这是所有 .NET 实现中一组标准化 API 的规范。应用或库还能以特定 .NET 的某版本实现为目标，这样便可获得特定于实现的 API 的访问权限。例如，面向 Xamarin.iOS 的应用有权访问 Xamarin 提供的 iOS API 包装器。

对于某些目标框架（例如 .NET Framework），可用 API 由 .NET 实现在系统上安装的程序集定义，其中可能包括应用程序框架 API（例如 ASP.NET、WinForms）。对于基于包的目标框架，框架 API 由安装在应用或库中的包定义。

请参阅目标框架。

TFM

目标框架名字对象。

一个标准化令牌格式，用于指定 .NET 应用或库的目标框架。目标框架通常由短名称（如 `net462`）引用。存在长格式的 TFM（如 .NETFramework,Version=4.6.2），但通常不用来指定目标框架。

请参阅目标框架。

UWP

通用 Windows 平台。

用于为物联网 (IoT) 生成触控 Windows 应用程序和软件的 .NET 实现。它旨在统一可能想要以其为目标的不同类型的设备，包括电脑、平板电脑、电话，甚至 Xbox。UWP 提供许多服务，如集中式应用商店、执行环境 (AppContainer) 和一组 Windows API（用于代替 Win32 (WinRT)）。应用可采用 C++、C#、Visual Basic 和 JavaScript 编写。使用 C# 和 Visual Basic 时，.NET API 由 .NET 5（和 .NET Core）及更高版本提供。

workload

某人正在构建的一种类型的应用。比应用模型更通用。例如，在每个 .NET 文档页（包括此页）的顶部都有一个"工作负载"下拉列表，你可以通过该列表切换到针对"Web"、"移动"、"云"、"桌面"和"机器学习 & 数据"的文档 。

在某些上下文中，工作负载是指 Visual Studio 功能的集合，可以选择安装这些功能以支持特定类型的应用。有关示例，请参阅选择工作负载。

请参阅

- .NET 基础知识
- .NET Framework 指南
- ASP.NET 概述
- 概述

新的 C# 模板生成顶级语句

2021/11/16 •

从 .NET 6 开始，使用 `console` 模板的新项目会生成与以前版本不同的代码：

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

新的输出使用最新的 C# 功能，这些功能简化了需要为程序编写的代码。通常，控制台应用模板生成以下代码：

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace MyApp // Note: actual namespace depends on the project name.
{
    public class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

这两种形式代表同一个程序。两者都适用于 C# 10.0。使用较新版本时，只需编写 `Main` 方法的主体。无需包含其他程序元素。可通过两个选项使用现有教程：

- 使用新的程序样式，添加功能时添加新的顶级语句。
- 使用 `Program` 类和 `Main` 方法将新程序样式转换为旧样式。

如果要使用旧模板，请参阅使用旧程序样式部分。

使用新的程序样式

使新程序更简单的功能是顶级语句、全局 `using` 指令和隐式 `using` 指令 。

顶级语句意味着编译器会为主程序生成命名空间、类和方法元素。你可以查看新应用程序的代码，并假设它包含早期模板所生成的 `Main` 方法内的语句。你可以向程序添加更多语句，就像用传统方式将更多语句添加到 `Main` 方法一样。甚至可以添加函数。它们被创建为嵌套在生成的 `Main` 方法中的本地函数。

顶级语句和隐式 `using` 指令都简化了构成应用程序的代码。若要按照现有教程操作，请将所有新语句添加到模板生成的 Program.cs 文件中。假设在本教程的说明中，你编写的语句位于 `Main` 方法中的左大括号和右大括号之间。

如果你更喜欢使用较旧的格式，则可以复制本文第二个示例中的代码，并像以前一样继续学习本教程。

有关顶级语句的详细信息，请参阅有关顶级语句的教程。

隐式 `using` 指令

隐式 `using` 指令意味着编译器会根据项目类型自动添加一组 `using` 指令。对于控制台应用程序，以下指令隐式包含在应用程序中：

- `using System;`
- `using System.IO;`
- `using System.Collections.Generic;`
- `using System.Linq;`
- `using System.Net.Http;`
- `using System.Threading;`
- `using System.Threading.Tasks;`

其他应用程序类型包括更多对这些应用程序类型通用的命名空间。

禁用隐式 `using` 语句

如果要删除该行为并手动控制项目中的所有命名空间，请在项目文件中添加 `<ImplicitUsings>disable</ImplicitUsings>`。

全局 `using` 指令

全局 `using` 指令导入整个应用程序的命名空间，而不是单个文件。可以通过向项目文件添加 `<Using>` 项或将 `global using` 指令添加到代码文件来添加这些全局指令。

还可以在项目文件中添加 `<Using>` 项以删除特定的隐式 `using` 指令。例如，如果通过 `<ImplicitUsings>enable</ImplicitUsings>` 打开隐式 usings 功能，则添加以下 `<Using>` 项会从隐式导入的 `System.Net.Http` 命名空间中删除命名空间：

```
<ItemGroup>
  <Using Remove="System.Net.Http" />
</ItemGroup>
```

使用旧程序样式

虽然 .NET 6 控制台应用模板将生成新样式的顶级语句程序，但使用 .NET 5 则不会。创建 .NET 5 项目时将获得旧程序样式。然后可以编辑项目文件以使其面向 .NET 6。

> **IMPORTANT**
> 创建面向 .NET 5 的项目需要 .NET 5 模板。可以使用 `dotnet new --install` 命令或通过安装 .NET 5 SDK来手动安装 .NET 5 模板。

1. 创建新项目

   ```
   dotnet new console --framework net5.0
   ```

2. 在文本编辑器中打开项目文件，并将 `<TargetFramework>net5.0</TargetFramework>` 更改为 `<TargetFramework>net6.0</TargetFramework>`。

   下面是说明更改的文件差异：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
-   <TargetFramework>net5.0</TargetFramework>
+   <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

</Project>
```

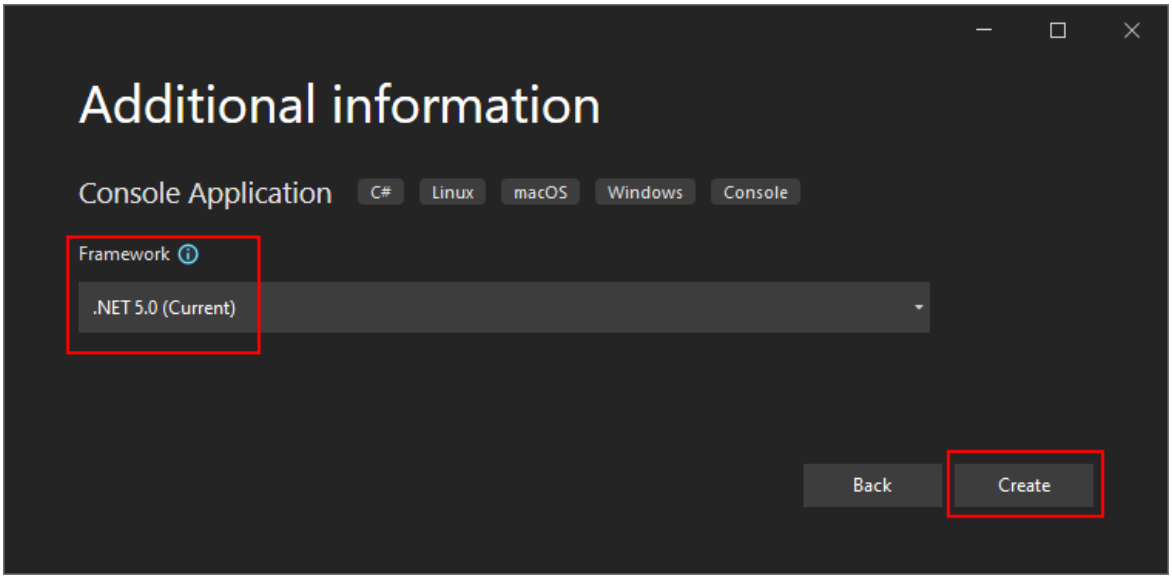
3. 可选步骤：仍可通过将隐式 `using` 指令和可为 null 上下文的属性添加到项目文件中，来使用某些较新的 .NET 6 和 C# 功能。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
+   <ImplicitUsings>enable</ImplicitUsings>
+   <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

在 Visual Studio 中使用旧程序样式

在 Visual Studio 中创建新的控制台项目时，系统会提示你使用一个下拉框来标识希望使用的目标框架。将该值更改为“5.0”。创建项目后，编辑项目文件以将其更改回 “6.0”。

1. 创建新项目时，设置步骤将导航到“其他信息”设置页。在此页上，将框架设置从“.NET 6.0 (长期支持)”更改为“.NET 5.0”，然后选择“创建”按钮 。

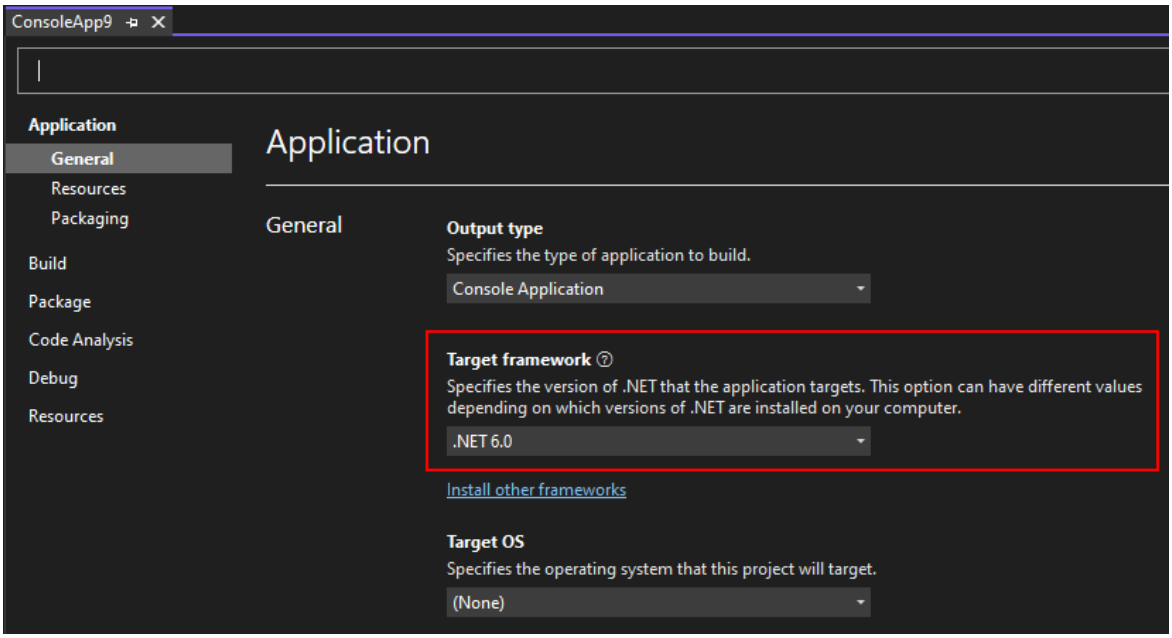


2. 创建项目后，找到“项目资源管理器”窗格。双击项目文件，并将 `<TargetFramework>net5.0</TargetFramework>` 更改为 `<TargetFramework>net6.0</TargetFramework>`。

   下面是说明更改的文件差异：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
-   <TargetFramework>net5.0</TargetFramework>
+   <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

</Project>
```

或者，在“解决方案资源管理器”窗格中右键单击项目，然后选择“属性”。这将打开一个设置页，可在其中更改“目标框架”。



3. 可选步骤：仍可通过将隐式 `using` 指令和可为 null 上下文的属性添加到项目文件中，来使用某些较新的 .NET 6 和 C# 功能。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
+   <ImplicitUsings>enable</ImplicitUsings>
+   <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

模板反馈

顶级模板是 .NET 6 中的新功能。对 GitHub 问题 #26313 添加赞成和反对票，以表达对此功能的支持。

教程：使用 Visual Studio 创建 .NET 控制台应用程序

2021/11/16 •

本教程演示如何在 Visual Studio 2022 中创建和运行 .NET 控制台应用程序。

先决条件

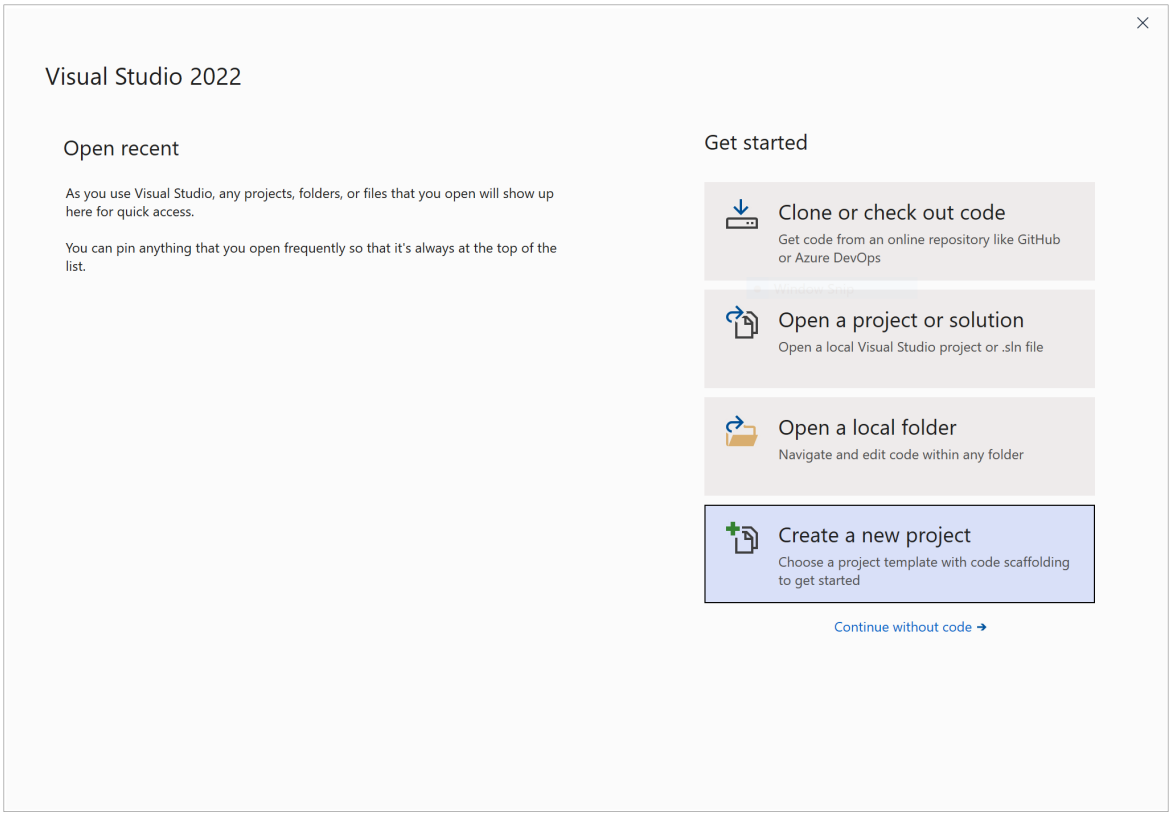
- 安装了具有 .NET 桌面开发工作负载的 Visual Studio 2022 版本 17.0.0 预览版。选择此工作负载时，将自动安装 .NET 6 SDK。

  有关详细信息，请参阅使用 Visual Studio 安装 .NET SDK。

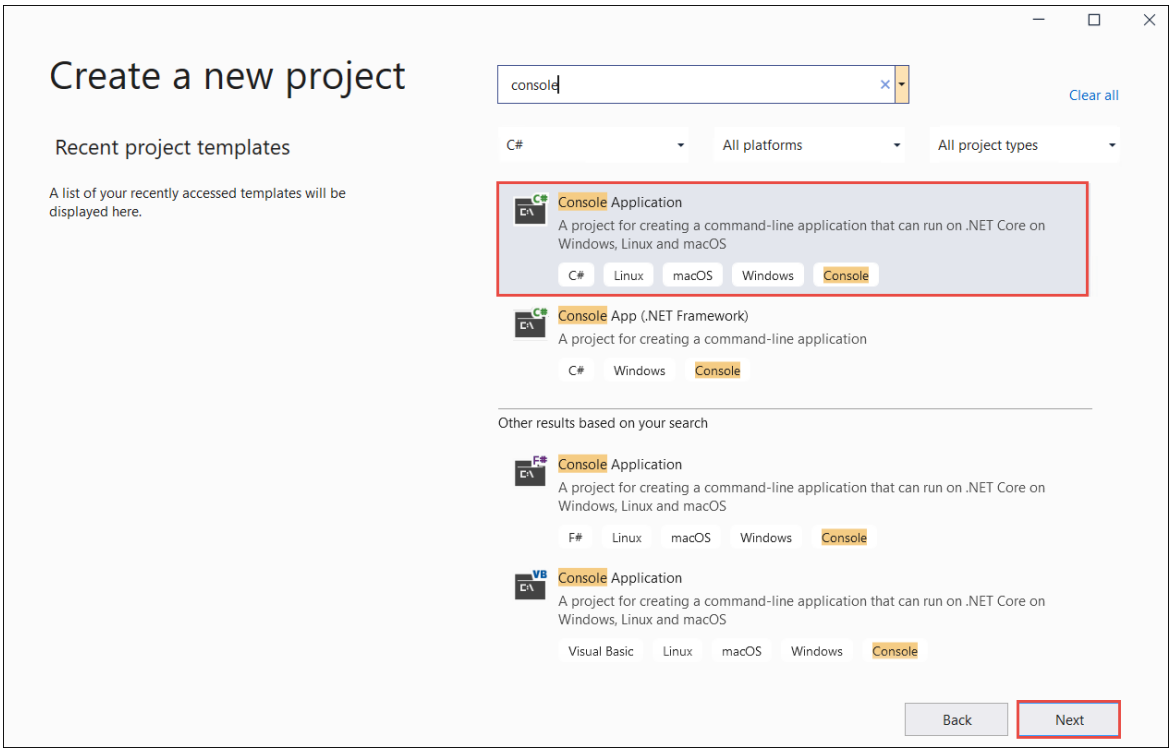
创建应用

创建一个名为“HelloWorld”的 .NET 控制台应用项目。

1. 启动 Visual Studio 2022。
2. 在“开始”页上，选择“创建新项目”。



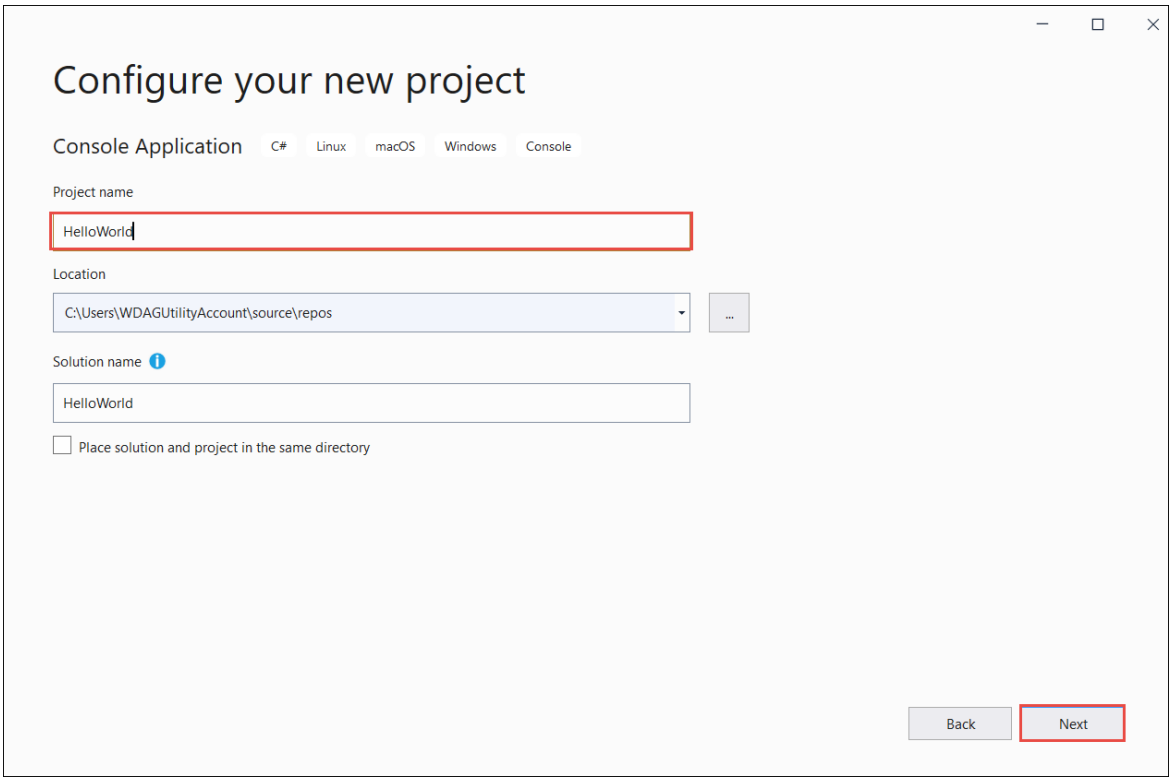
3. 在“创建新项目”页面，在搜索框中输入“控制台”。接下来，从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“控制台应用程序”模板，然后选择“下一步”。



TIP

如果看不到 .NET 模板，则可能缺少所需的工作负载。在“找不到所需内容?”消息下，选择“安装更多工具和功能”链接。Visual Studio 安装程序随即打开。确保安装了 .NET 桌面开发工作负载。

4. 在“配置新项目”对话框中，在“项目名称”框中输入“HelloWorld”。然后选择“下一步”。



5. 在“其他信息”对话框中，选择“.NET 6 (长期支持)”，然后选择“创建”。

该模板创建了一个在控制台窗口中显示“Hello World”的简单应用程序。代码位于 Program.cs 或 Program.vb 文件中：

```
Console.WriteLine("Hello, World!");
```

```
Imports System

Module Program
    Sub Main(args As String())
        Console.WriteLine("Hello World!")
    End Sub
End Module
```

如果未显示想要使用的语言，请更改页面顶部的语言选择器。

6. 对于 C#，代码只是一行，用于调用 Console.WriteLine(String) 方法以在控制台窗口中 显示“Hello World!”。将 Program.cs 的内容替换为以下代码：

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

```
' This step of the tutorial applies only to C#.
```

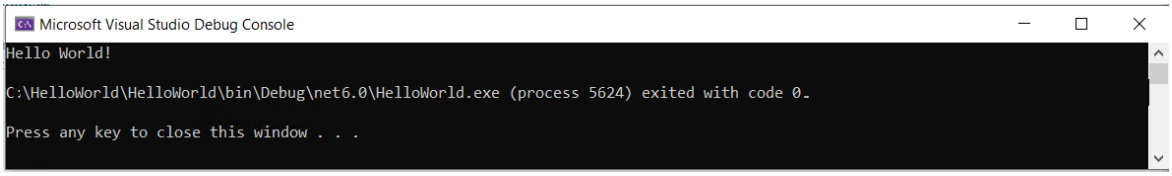
该代码将定义类 `Program`，其中包含一个将 String 数组用作参数的方法 `Main`。`Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。*args* 数组中包含在应用程序启动时提供的所有命令行自变量。

在最新版本的 C# 中，名为顶级语句的新功能允许你省略 `Program` 类和 `Main` 方法。大多数现有 C# 程序不使用顶级语句，因此本教程不使用此新功能。但它在 C# 10 中可用，是否在程序中使用它是样式首选项的问题。

运行应用

1. 按 `Ctrl+F5` 运行程序而不进行调试。

   此时将打开在屏幕上显示文本“Hello World!” 。



2. 按任意键关闭控制台窗口。

增强应用

改进应用程序，使其提示用户输入名字，并将其与日期和时间一同显示。

1. 在 Program.cs 或 Program.vb 中，将 `Main` 方法的内容（当前只是调用 `Console.WriteLine` 的行）替换为以下代码：

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
Console.Write($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

```
Console.WriteLine("What is your name?")
Dim name = Console.ReadLine()
Dim currentDate = DateTime.Now
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}")
Console.Write($"{Environment.NewLine}Press any key to exit...")
Console.ReadKey(True)
```

此代码会在控制台窗口中显示一条提示，然后等待用户输入字符串并按 Enter。它会将此字符串存储到名为 `name` 的变量中。它还会检索 DateTime.Now 属性的值（其中包含当前的本地时间），并将此值赋给 `currentDate` 变量。同时会在控制台窗口中显示这些值。最后会在控制台窗口中显示一条提示，并调用 Console.ReadKey(Boolean) 方法来等待用户输入。

Environment.NewLine 是一种独立于平台和语言的表示换行符的方式。替代方法是在 C# 中使用 `\n` 和在 Visual Basic 中使用 `vbCrLf`。

字符串前面的美元符号 (`$`) 使你可以将表达式（如变量名称）放入字符串中的大括号内。表达式值将代替表达式插入到字符串中。此语法称为内插字符串。

2. 按 `Ctrl+F5` 运行程序而不进行调试。
3. 出现提示时，输入名称并按 Enter 键。

```
What is your name?
Maira

Hello, Maira, on 12/3/2019 at 3:36 AM!

Press any key to exit...
```

4. 按任意键关闭控制台窗口。

其他资源

- 当前版本和长期支持版本

后续步骤

在本教程中，你创建了一个 .NET 控制台应用程序。在下一教程中，你将调试该应用。

使用 Visual Studio 调试 .NET 控制台应用程序

本教程演示如何在 Visual Studio 2019 中创建和运行 .NET 控制台应用程序。

先决条件

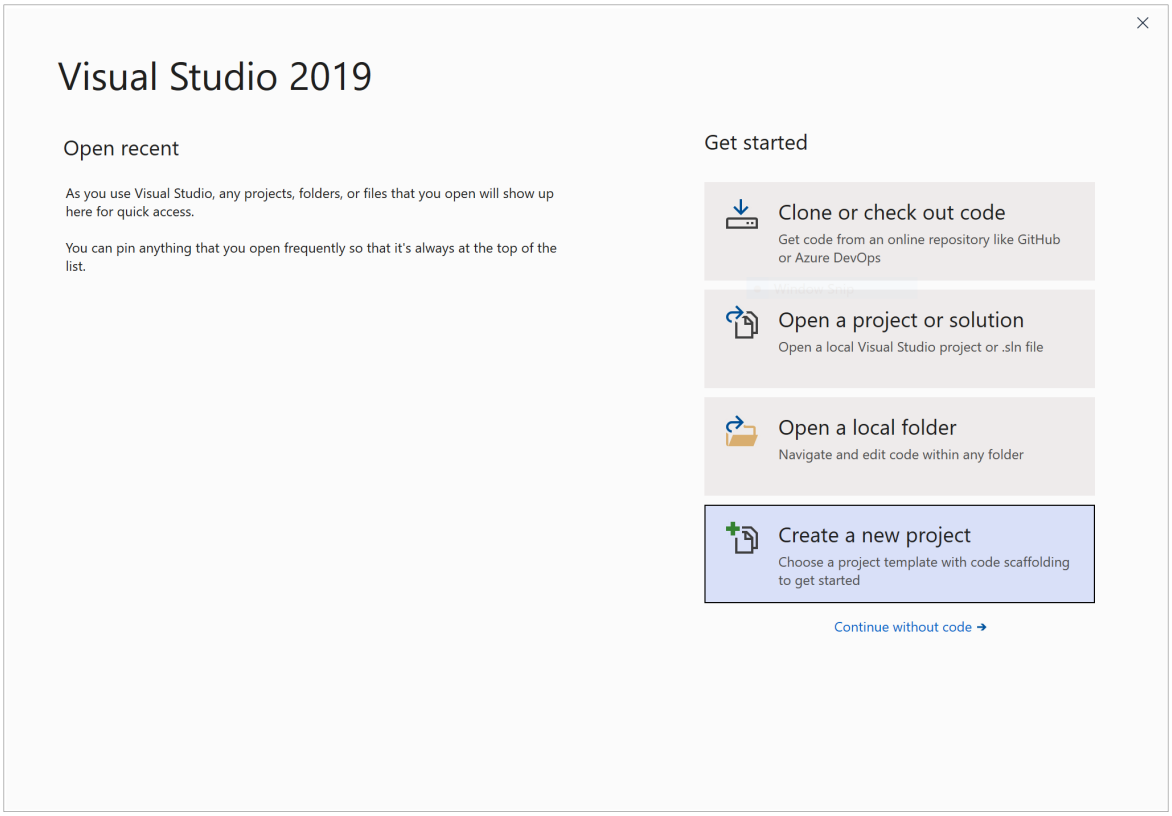
- 安装了“.NET Core 跨平台开发”工作负载的 Visual Studio 2019 版本 16.9.2 或更高版本。选择此工作负载时，将自动安装 .NET 5.0 SDK。

有关详细信息，请参阅使用 Visual Studio 安装 .NET SDK。

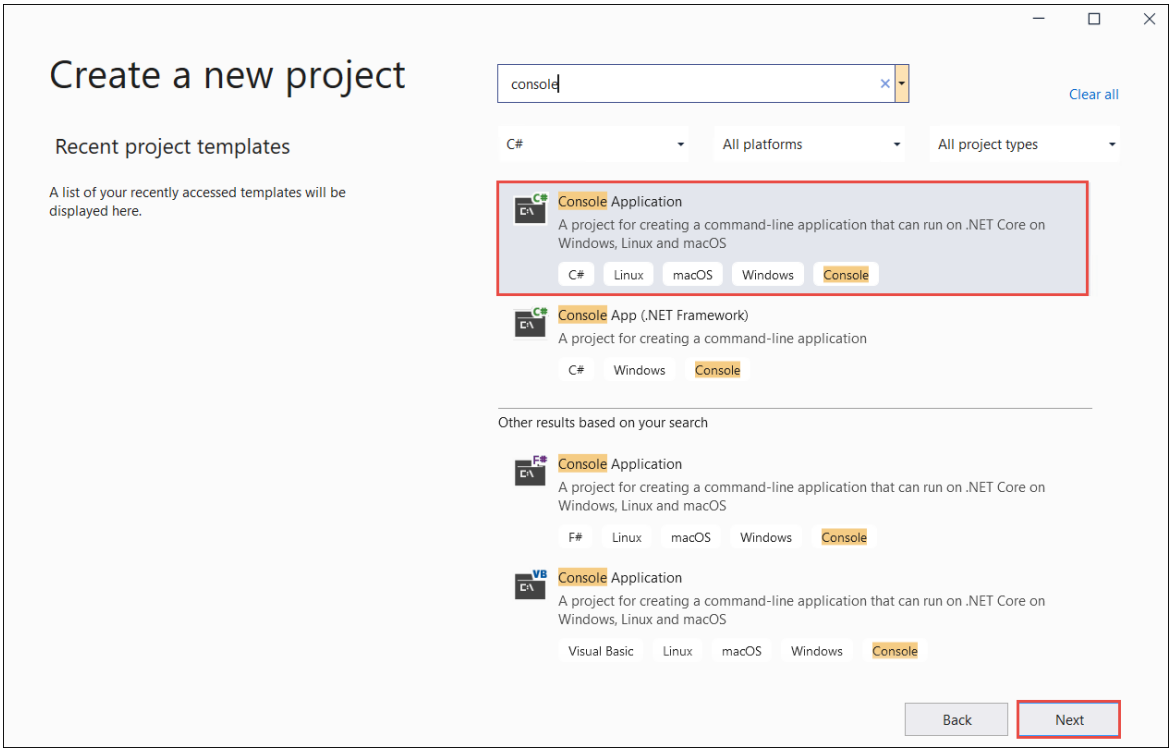
创建应用

创建一个名为“HelloWorld”的 .NET 控制台应用项目。

1. 启动 Visual Studio 2019。
2. 在“开始”页上，选择“创建新项目”。

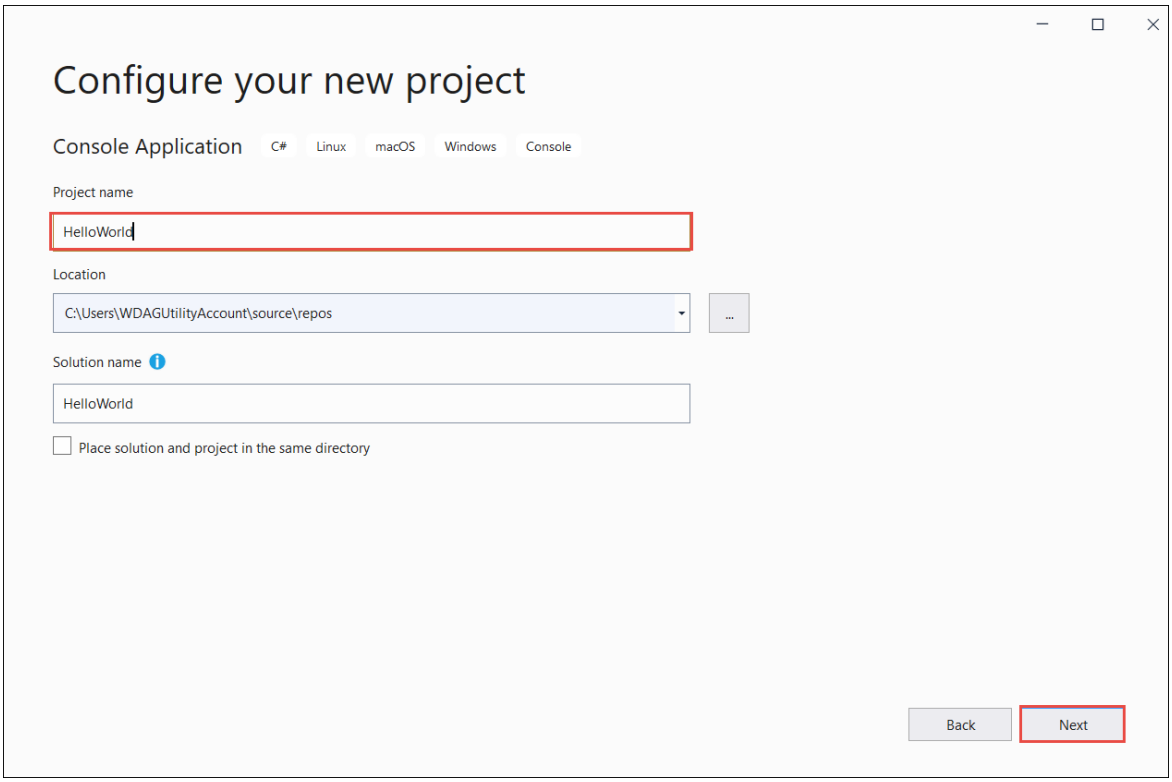


3. 在“创建新项目”页面，在搜索框中输入“控制台”。接下来，从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“控制台应用程序”模板，然后选择“下一步”。

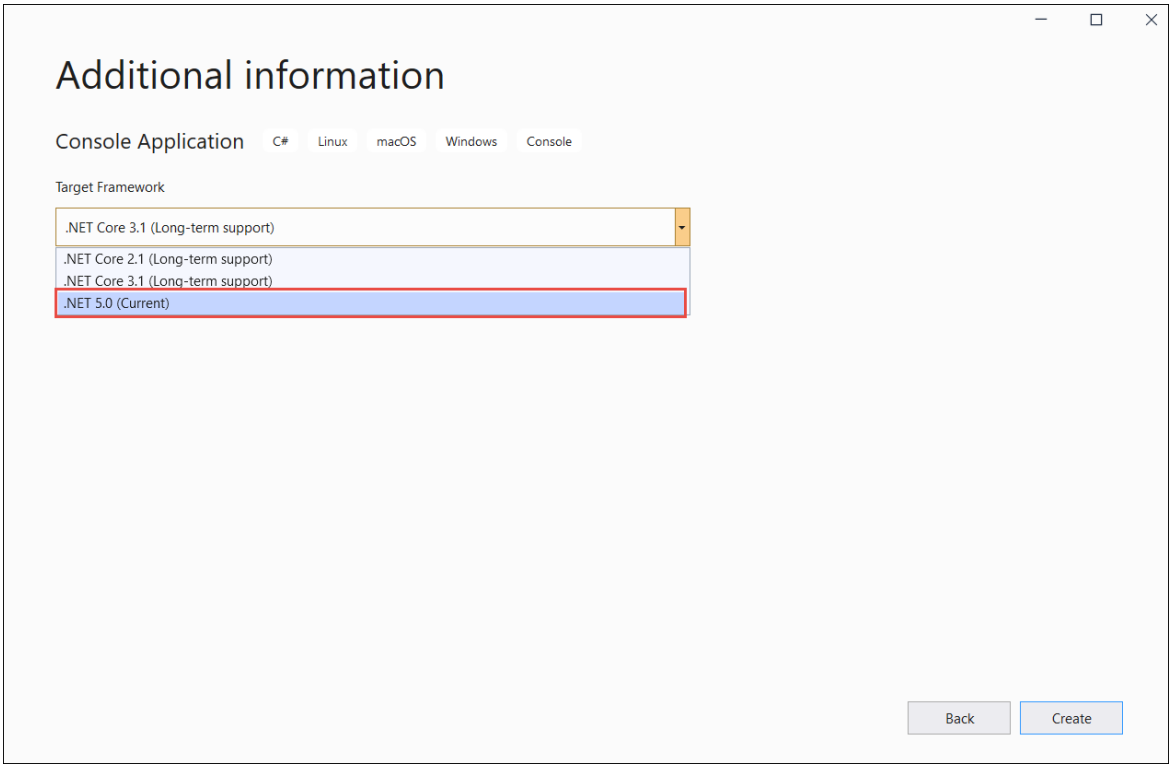


> **TIP**
> 如果看不到 .NET 模板，则可能缺少所需的工作负载。在“找不到所需内容?”消息下，选择“安装更多工具和功能”链接。Visual Studio 安装程序随即打开。确保安装了“.NET Core 跨平台开发”工作负载。

4. 在“配置新项目”对话框中，在“项目名称”框中输入“HelloWorld”。然后选择“下一步”。



5. 在“其他信息”对话框中，选择“.NET 5.0 (当前)”，然后选择“创建”。



用于创建简单的“Hello World”应用程序的模板。它会调用 Console.WriteLine(String) 方法来显示“Hello World!”显示文本字符串“Hello World!”。

模板代码将定义类 `Program`，其中包含一个需要将 String 数组用作参数的方法 `Main`：

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

```
Imports System

Module Program
    Sub Main(args As String())
        Console.WriteLine("Hello World!")
    End Sub
End Module
```

`Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。*args* 数组中包含在应用程序启动时提供的所有命令行自变量。

如果未显示想要使用的语言，请更改页面顶部的语言选择器。

运行应用

1. 按 Ctrl+F5 运行程序而不进行调试。

   此时将打开在屏幕上显示文本“Hello World!” 。

2. 按任意键关闭控制台窗口。

增强应用

改进应用程序，使其提示用户输入名字，并将其与日期和时间一同显示。

1. 在 Program.cs 或 Program.vb 中，将 `Main` 方法的内容（当前只是调用 `Console.WriteLine` 的行）替换为以下代码：

   ```
   Console.WriteLine("What is your name?");
   var name = Console.ReadLine();
   var currentDate = DateTime.Now;
   Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
   Console.Write($"{Environment.NewLine}Press any key to exit...");
   Console.ReadKey(true);
   ```

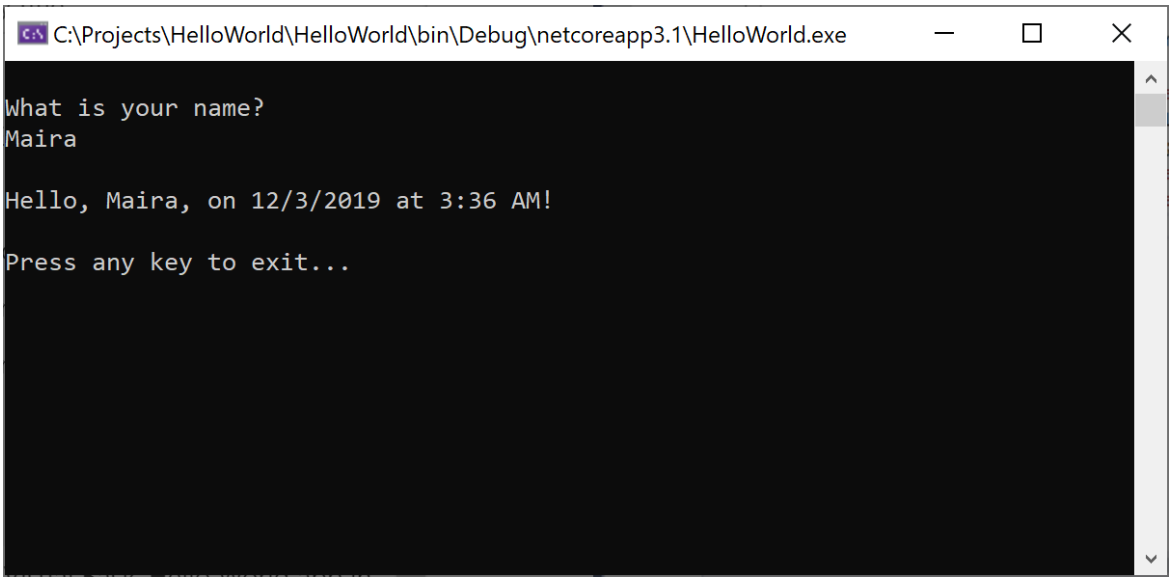
```
Console.WriteLine("What is your name?")
Dim name = Console.ReadLine()
Dim currentDate = DateTime.Now
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}")
Console.Write($"{Environment.NewLine}Press any key to exit...")
Console.ReadKey(True)
```

此代码会在控制台窗口中显示一条提示，然后等待用户输入字符串并按 Enter。它会将此字符串存储到名为 `name` 的变量中。它还会检索 DateTime.Now 属性的值（其中包含当前的本地时间），并将此值赋给 `currentDate` 变量。同时会在控制台窗口中显示这些值。最后会在控制台窗口中显示一条提示，并调用 Console.ReadKey(Boolean) 方法来等待用户输入。

Environment.NewLine 是一种独立于平台和语言的表示换行符的方式。替代方法是在 C# 中使用 `\n` 和在 Visual Basic 中使用 `vbCrLf`。

字符串前面的美元符号 (`$`) 使你可以将表达式（如变量名称）放入字符串中的大括号内。表达式值将代替表达式插入到字符串中。此语法称为内插字符串。

2. 按 `Ctrl+F5` 运行程序而不进行调试。
3. 出现提示时，输入名称并按 Enter 键。



4. 按任意键关闭控制台窗口。

其他资源

- 当前版本和长期支持版本

后续步骤

在本教程中，你创建了一个 .NET 控制台应用程序。在下一教程中，你将调试该应用。

使用 Visual Studio 调试 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio 调试 .NET 控制台应用程序

2021/11/16 •

本教程介绍了 Visual Studio 中提供的调试工具。

先决条件

- 本教程适用于在使用 Visual Studio 创建 .NET 控制台应用程序中创建的控制台应用。

使用“调试”生成配置

“调试”和“发布”是 Visual Studio 的内置生成配置 。可使用“调试”生成配置进行调试，使用“发布”配置进行最终版本分发。

在“调试”配置中，程序使用完整符号调试信息编译，且不进行优化。优化会使调试复杂化，因为源代码和生成的指令之间的关系更加复杂。程序的发布配置进行了完全优化，且不包含任何符号调试信息。

默认情况下，Visual Studio 使用“调试”生成配置，因此不需要在调试之前对其进行更改。

1. 启动 Visual Studio。
2. 打开在使用 Visual Studio 创建 .NET 控制台应用程序中创建的项目。

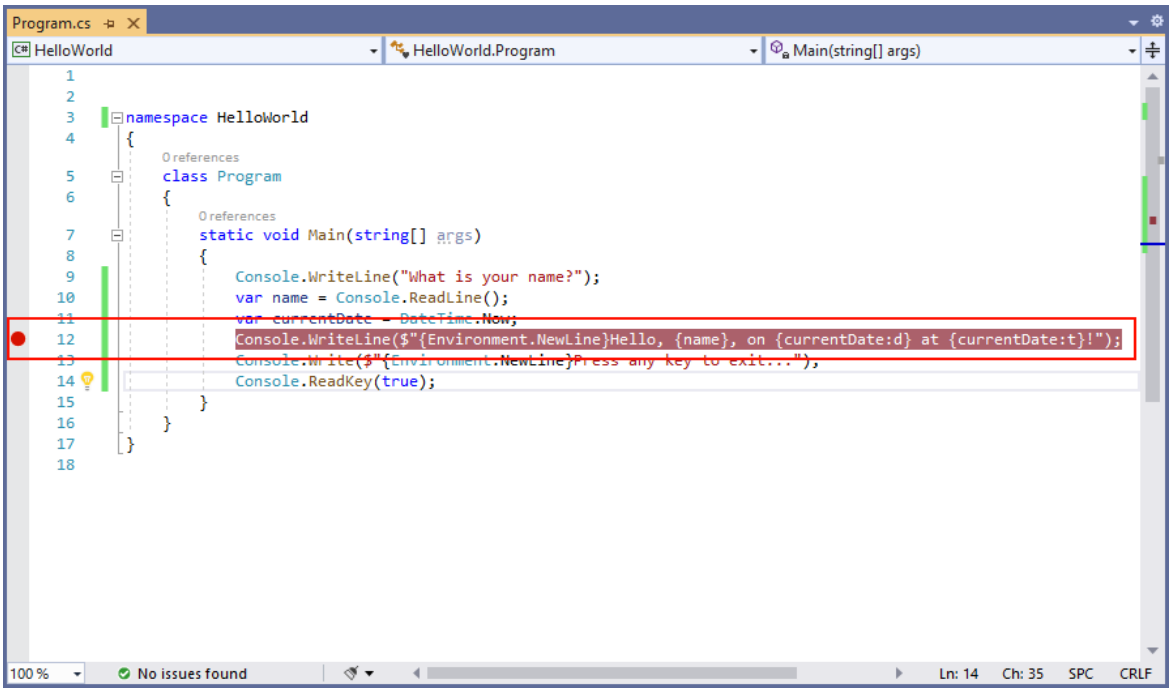
   当前的生成配置显示在工具栏上。下面的工具栏图像显示 Visual Studio 配置为编译应用的“调试”版本：

设置断点

断点会在执行包含断点的代码行之前暂时中断执行应用程序。

1. 单击该行代码窗口的左边缘，在显示名称、日期和时间的行上设置断点。左边缘在行号的左侧。设置断点的其他方法是，通过将光标置于代码行中，然后按 F9 或从菜单栏中选择“调试” > “切换断点”来进行设置。

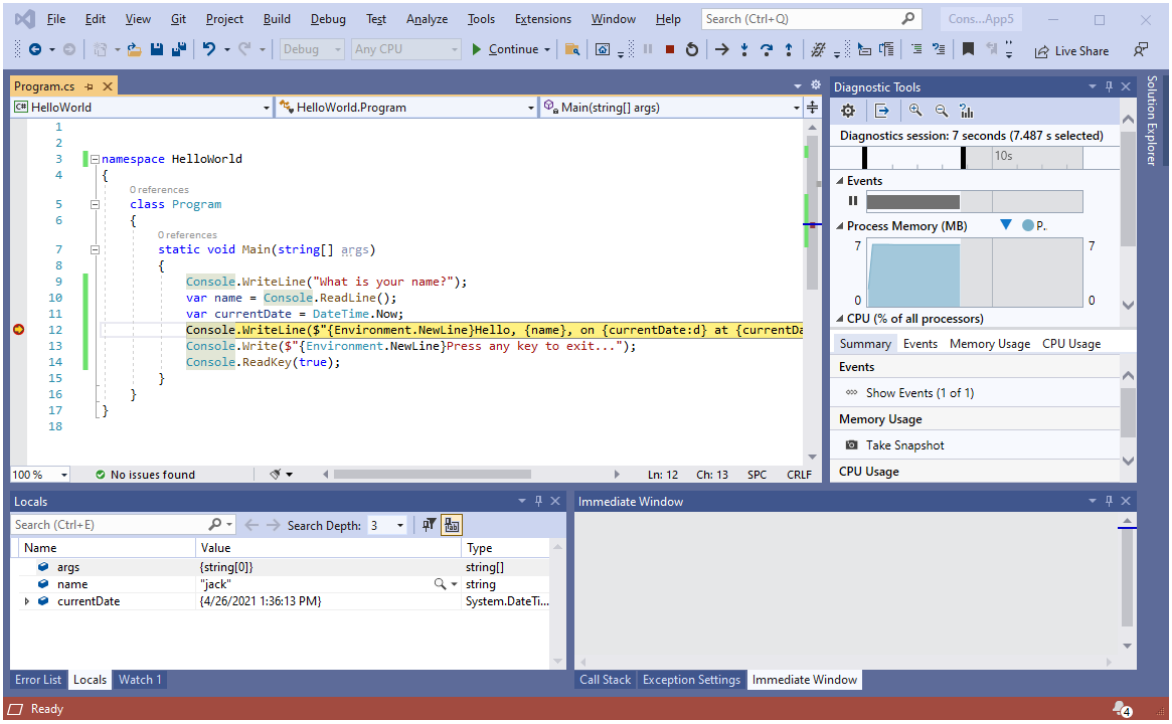
   如下图所示，Visual Studio 通过突出显示此代码行并在左边缘显示红点来指示设置了断点的行。



2. 按 F5，在调试模式下运行程序。启动调试的另一种方法是从菜单中选择“调试” > “启动调试”。

3. 当程序提示输入名称时，在控制台窗口中输入字符串，然后按 Enter。

4. 到达断点时，程序停止执行，然后执行 `Console.WriteLine` 方法。“局部变量”窗口显示当前正在执行的方法中定义的变量值。



使用“即时”窗口

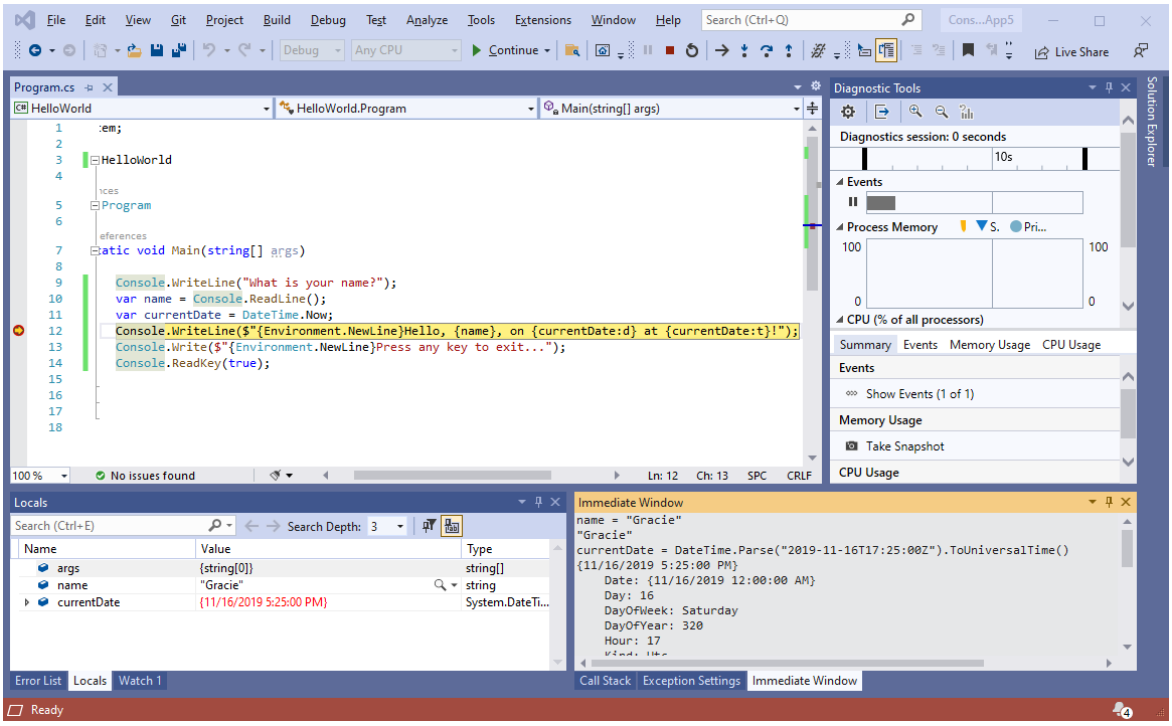
在“即时”窗口中，可以与正在调试的应用程序进行交互。可以通过交互方式更改变量值，看看这样会对程序产生哪些影响。

1. 如果“即时”窗口不可见，请选择“调试” > “Windows” > “即时”来显示它。

2. 在“即时”窗口中输入 `name = "Gracie"`，然后按 Enter 键。

3. 在“即时”窗口中输入 `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()`，然后按 Enter 键。

“即时”窗口显示字符串变量的值和 DateTime 值的属性。此外，“局部变量”窗口中也会更新变量值。



4. 按 F5 继续执行程序。继续操作的另一种方法是从菜单中选择“调试” > “继续”。

   控制台窗口中显示的值对应于在“即时”窗口中所做的更改。

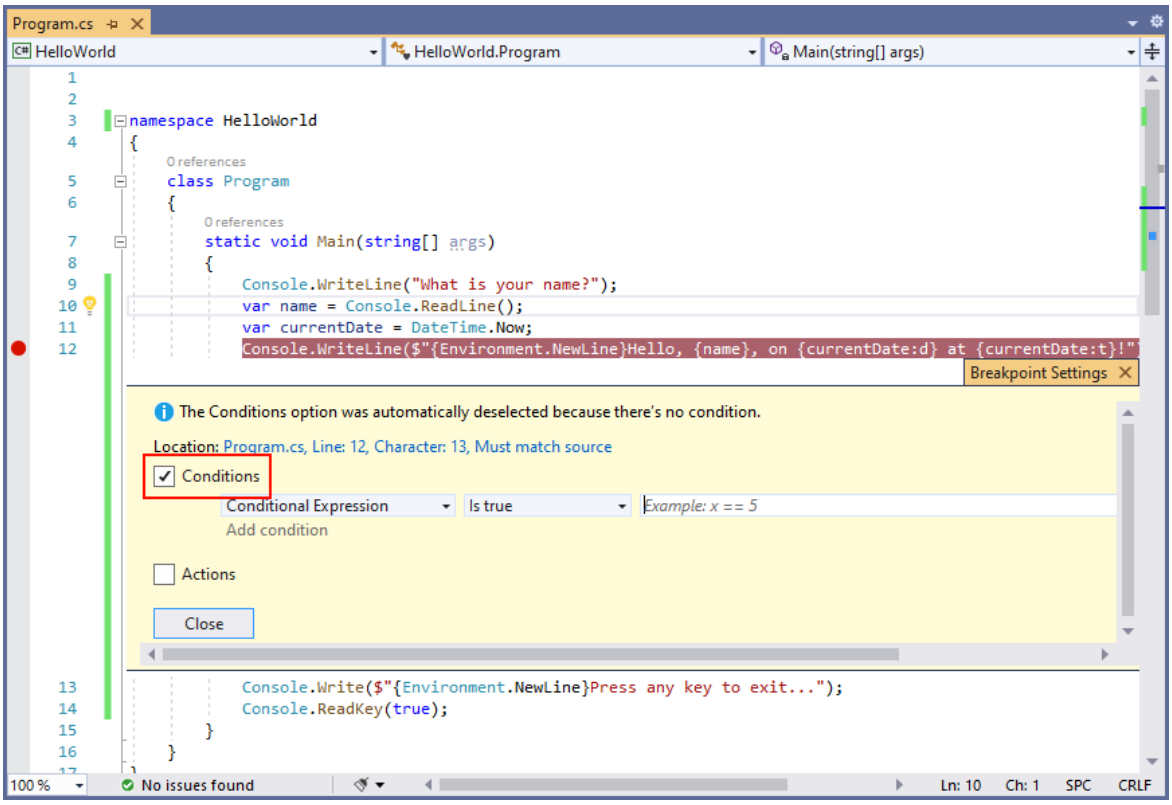
```
What is your name?
jack

Hello, Gracie, on 11/16/2019 at 5:25 PM!

Press any key to exit...
```

5. 按任意键，退出应用程序并停止调试。

设置条件断点

程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？ 可以使用名为“条件断点”的有用调试功能对此进行测试。

1. 右键单击表示断点的红点。在上下文菜单中，选择“条件”，打开“断点设置”对话框 。选择“条件”框（如果尚未选择）。



2. 对于条件表达式，在显示测试 `x` 是否为 5 的示例代码的字段中输入以下代码。

```
String.IsNullOrEmpty(name)
```


```
String.IsNullOrEmpty(name)
```


每次命中断点时，调试器都会调用 `String.IsNullOrEmpty(name)` 方法，仅当该方法调用返回 `true` 时，它才会在此行上中断。

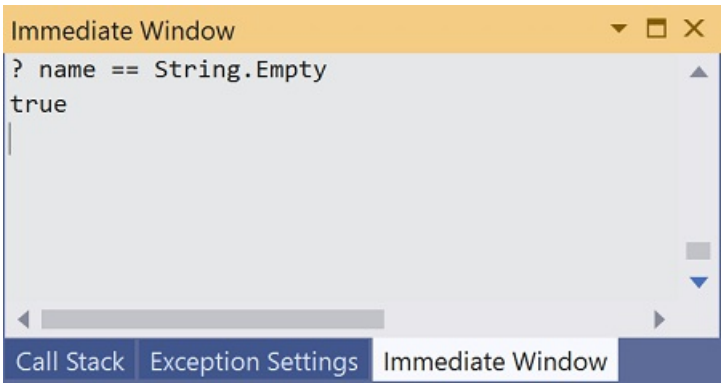
可以指定命中次数(而不是条件表达式)，这样程序就会在语句的执行次数达到指定值时中断执行。另一种方法是指定"筛选条件"，这样就可以根据诸如线程标识符、进程名称或线程名称之类的特性来中断程序执行。

3. 选择"关闭"以关闭对话框。
4. 通过按 F5 调试来启动程序。
5. 在控制台窗口中，在看到输入名称的提示时按 Enter 键。
6. 由于符合指定的条件(`name` 为 `null` 或 String.Empty)，因此程序会在到达断点时以及在 `Console.WriteLine` 方法执行之前停止执行。
7. 选择"局部变量"窗口，其中显示当前正在执行的方法的局部变量值。在这种情况下，`Main` 是当前正在执行的方法。请注意，`name` 变量的值为 `""` 或 String.Empty。
8. 在"即时"窗口中输入下面的语句并按 Enter，确认值为空字符串。结果为 `true`。

```
? name == String.Empty
```

```
? String.IsNullOrEmpty(name)
```

问号指示即时窗口计算表达式。



9. 按 F5 继续执行程序。
10. 按任意键，关闭控制台窗口并停止调试。
11. 单击代码窗口左边缘上的点，清除断点。清除断点的其他方法是在选中代码行时按 F9 或选择“调试”>“切换断点”。

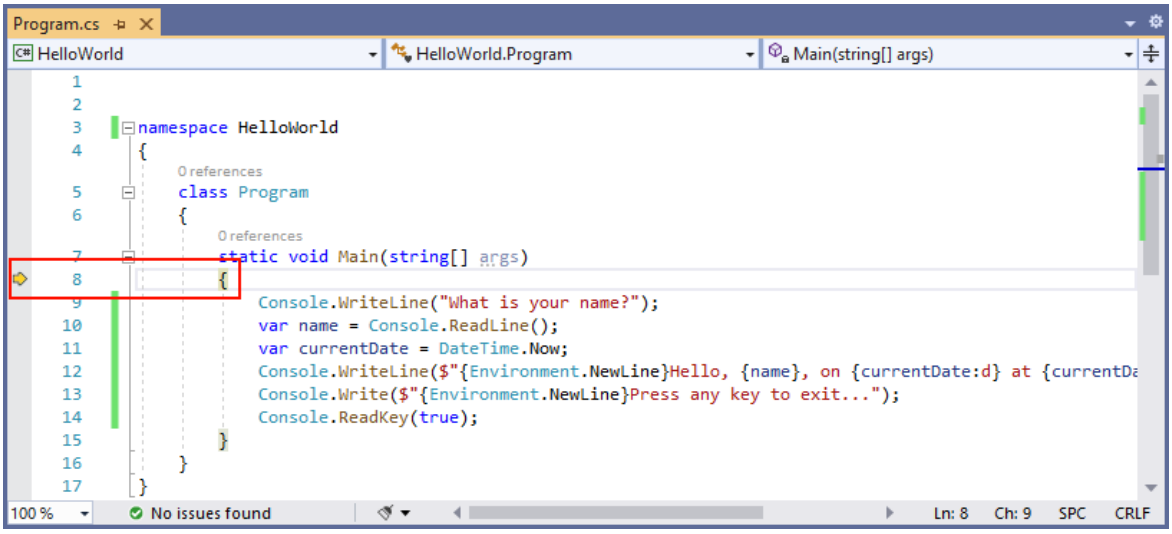
单步执行程序

使用 Visual Studio，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并通过程序代码的一小部分执行程序流。由于此程序很小，因此可以单步执行整个程序。

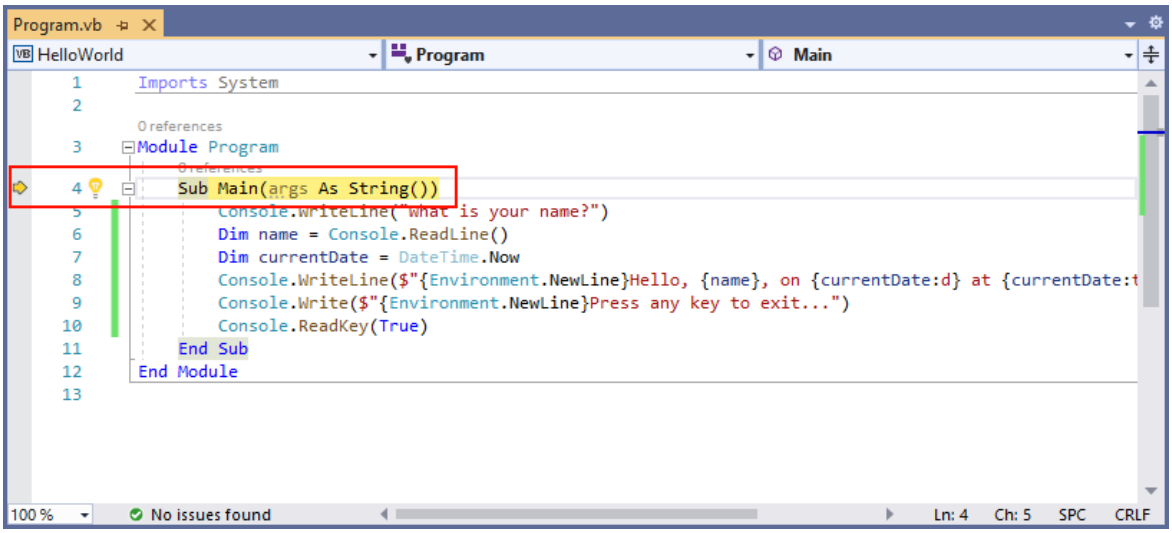
1. 选择“调试” > “单步执行” 。一次调试一个语句的另一种方法是按 F11。

   Visual Studio 会在要执行的下一行旁边突出显示一个箭头。

   C#



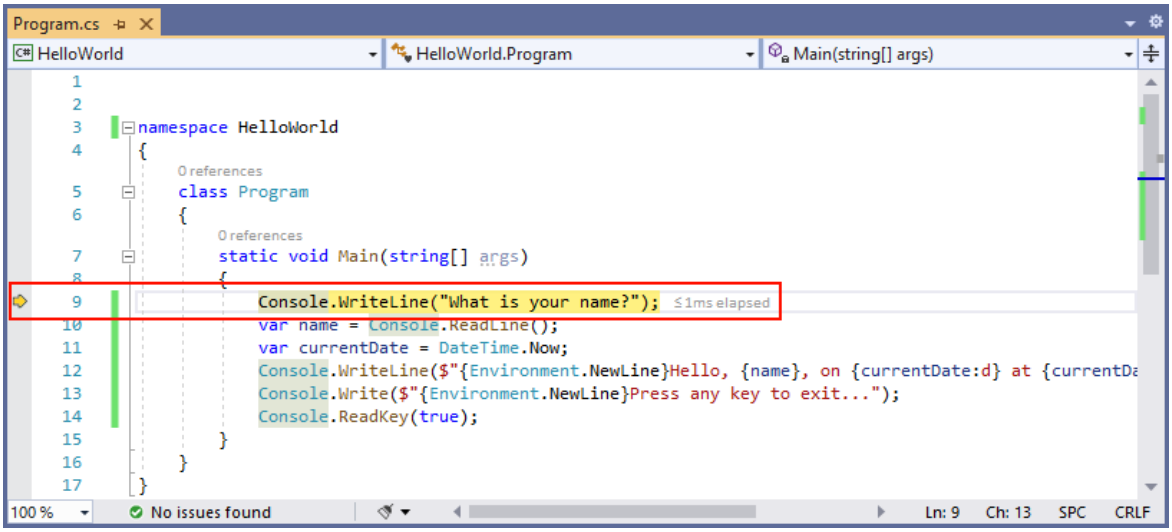
   Visual Basic



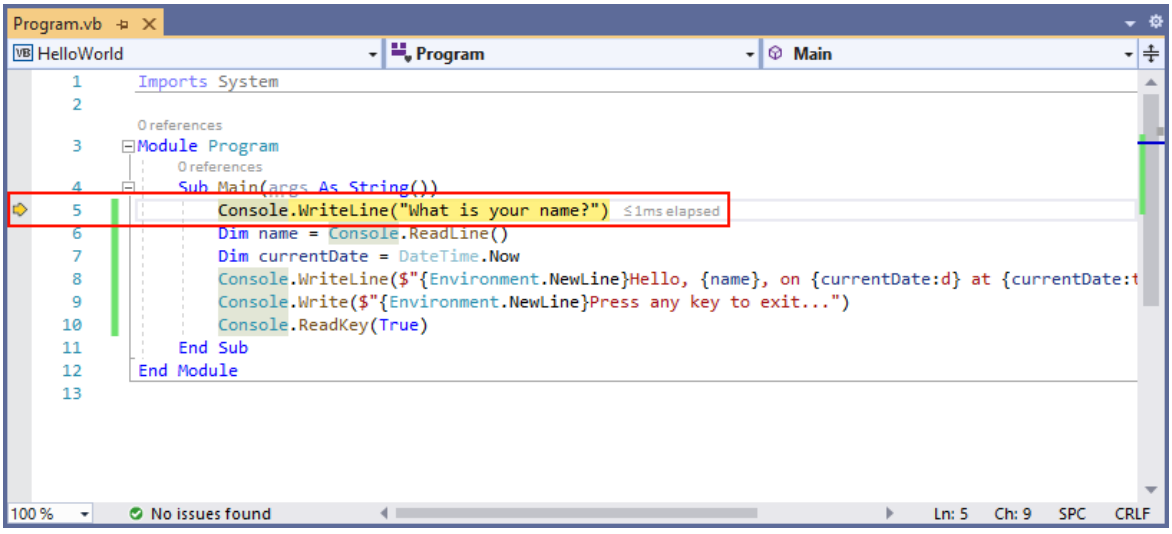
此时，“局部变量”窗口显示 `args` 数组为空，`name` 和 `currentDate` 具有默认值。此外，Visual Studio 还打开了一个空白控制台窗口。

2. 按下 F11。Visual Studio 现在突出显示要执行的下一行。“局部变量”窗口保持不变，控制台窗口仍为空白。

   C#



Visual Basic



3. 按下 F11。Visual Studio 突出显示包含 `name` 变量赋值的语句。“局部变量”窗口显示 `name` 为 `null`，控制台窗口显示字符串“What is your name?”。

4. 在控制台窗口中输入字符串，然后按 Enter，从而响应提示。控制台无响应，输入的字符串未显示在控制台窗口中，但 Console.ReadLine 方法将捕获输入。

5. 按下 F11。Visual Studio 突出显示包含 `currentDate` 变量赋值的语句。“局部变量”窗口显示 Console.ReadLine 方法调用返回的值。控制台窗口还显示在提示符处输入的字符串。

6. 按下 F11。“局部变量”窗口显示通过 DateTime.Now 属性赋值后的 `currentDate` 变量值。控制台窗口保持不变。

7. 按下 F11。Visual Studio 调用 Console.WriteLine(String, Object, Object) 方法。控制台窗口会显示格式化的字符串。

8. 选择“调试” > “跳出”。停止分步执行的另一种方法是按 Shift+F11。

   控制台窗口会显示一条消息，并等待用户按任意键。

9. 按任意键，关闭控制台窗口并停止调试。

使用“发布”生成配置

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，有时可能会对应用程序的行为产生不良影响。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要生成和测试控制台应用程序的发布版本，请将工具栏上的生成配置从“调试”更改为“发布”。

按 F5 或选择“生成”菜单中的“生成解决方案”后，Visual Studio 会编译应用程序的“发布”版本。可像测试“调试”版本一样测试“发布”版本。

后续步骤

在本教程中，你使用了 Visual Studio 调试工具。在下一教程中，你将发布应用的可部署版本。

使用 Visual Studio 发布 .NET 控制台应用程序

本教程介绍了 Visual Studio 中提供的调试工具。

先决条件

- 本教程适用于在使用 Visual Studio 创建 .NET 控制台应用程序中创建的控制台应用。

使用“调试”生成配置

“调试”和“发布”是 Visual Studio 的内置生成配置。可使用“调试”生成配置进行调试，使用“发布”配置进行最终版本分发。

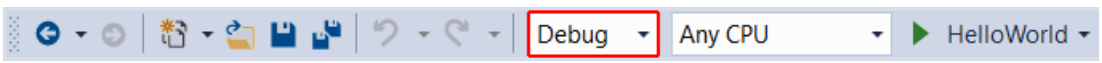
在“调试”配置中，程序使用完整符号调试信息编译，且不进行优化。优化会使调试复杂化，因为源代码和生成的指令之间的关系更加复杂。程序的发布配置进行了完全优化，且不包含任何符号调试信息。

默认情况下，Visual Studio 使用“调试”生成配置，因此不需要在调试之前对其进行更改。

1. 启动 Visual Studio。

2. 打开在使用 Visual Studio 创建 .NET 控制台应用程序中创建的项目。

   当前的生成配置显示在工具栏上。下面的工具栏图像显示 Visual Studio 配置为编译应用的“调试”版本：

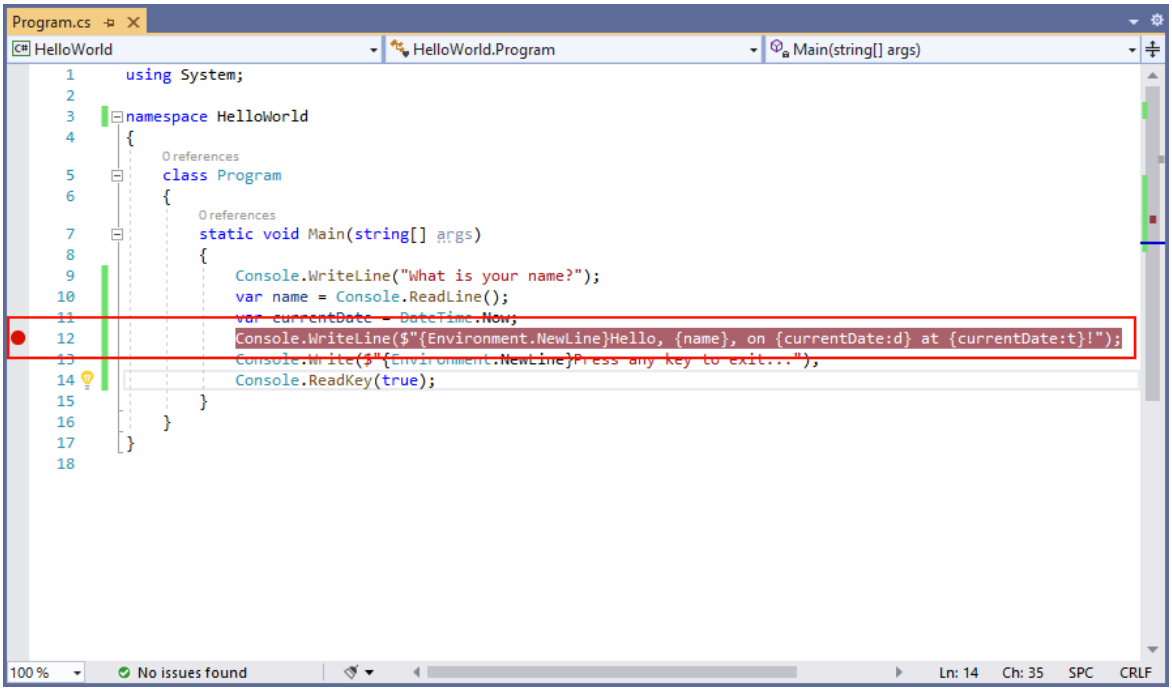


设置断点

断点会在执行包含断点的代码行之前暂时中断执行应用程序。

1. 单击该行代码窗口的左边缘，在显示名称、日期和时间的行上设置断点。左边缘在行号的左侧。设置断点的其他方法是，通过将光标置于代码行中，然后按 F9 或从菜单栏中选择“调试” > “切换断点”来进行设置。

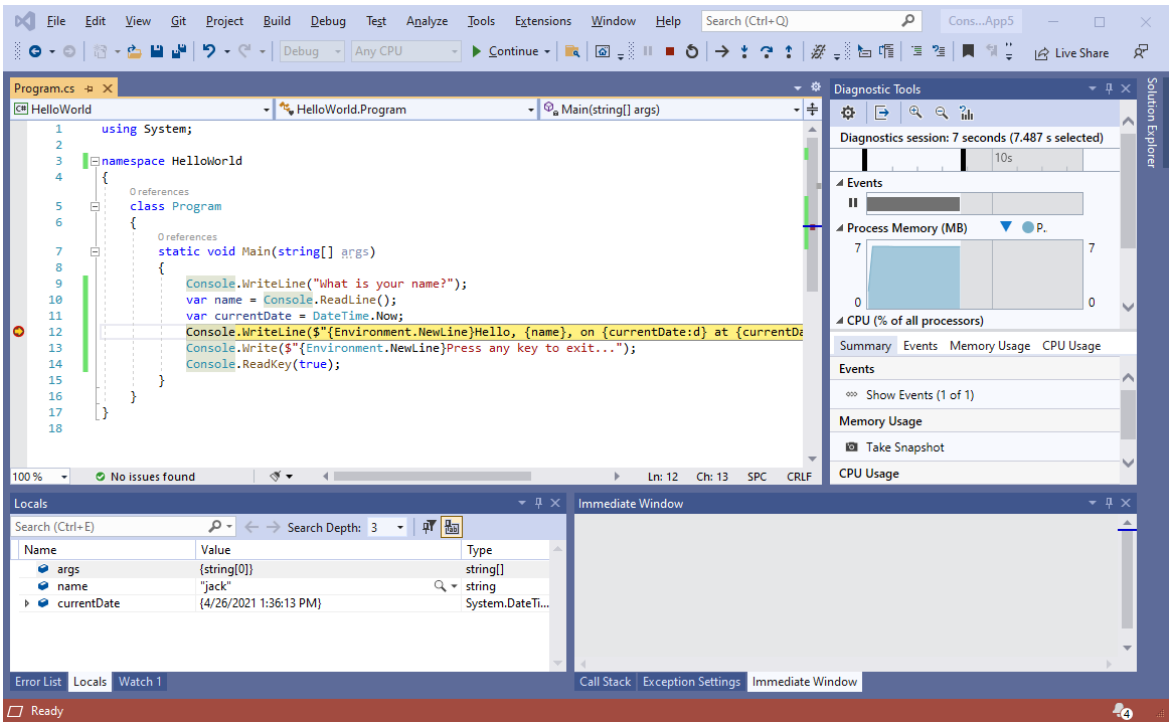
   如下图所示，Visual Studio 通过突出显示此代码行并在左边缘显示红点来指示设置了断点的行。



2. 按 F5，在调试模式下运行程序。启动调试的另一种方法是从菜单中选择“调试” > “启动调试”。

3. 当程序提示输入名称时，在控制台窗口中输入字符串，然后按 Enter。

4. 到达断点时，程序停止执行，然后执行 `Console.WriteLine` 方法。“局部变量”窗口显示当前正在执行的方法中定义的变量值。

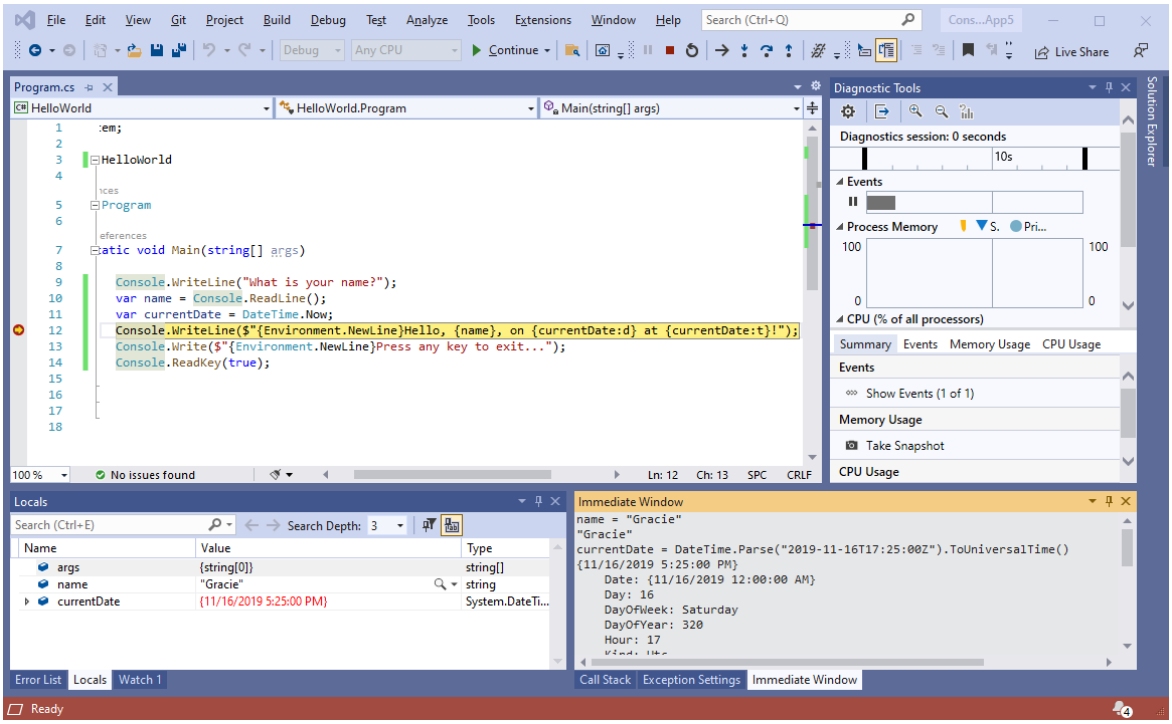


使用“即时”窗口

在“即时”窗口中，可以与正在调试的应用程序进行交互。可以通过交互方式更改变量值，看看这样会对程序产生哪些影响。

1. 如果“即时”窗口不可见，请选择“调试” > “Windows” > “即时”来显示它。
2. 在“即时”窗口中输入 `name = "Gracie"`，然后按 Enter 键。
3. 在“即时”窗口中输入 `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()`，然后按 Enter 键。

   “即时”窗口显示字符串变量的值和 DateTime 值的属性。此外，“局部变量”窗口中也会更新变量值。



4. 按 F5 继续执行程序。继续操作的另一种方法是从菜单中选择“调试” > “继续”。

   控制台窗口中显示的值对应于在“即时”窗口中所做的更改。

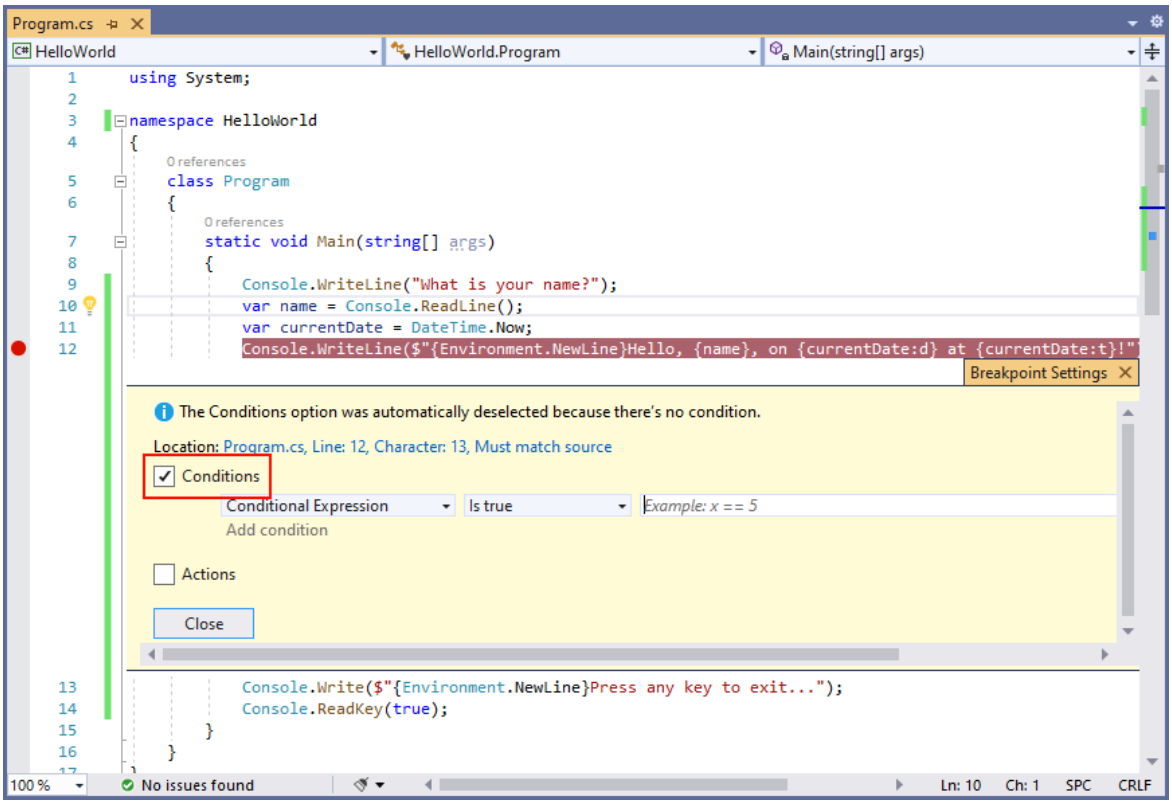
```
What is your name?
jack

Hello, Gracie, on 11/16/2019 at 5:25 PM!

Press any key to exit...
```

5. 按任意键，退出应用程序并停止调试。

设置条件断点

程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？ 可以使用名为“条件断点”的有用调试功能对此进行测试。

1. 右键单击表示断点的红点。在上下文菜单中，选择“条件”，打开“断点设置”对话框 。选择“条件”框（如果尚未选择）。



2. 对于条件表达式，在显示测试 `x` 是否为 5 的示例代码的字段中输入以下代码。如果未显示想要使用的语言，请更改页面顶部的语言选择器。

```
String.IsNullOrEmpty(name)
```


```
String.IsNullOrEmpty(name)
```


每次命中断点时，调试器都会调用 `String.IsNullOrEmpty(name)` 方法，仅当该方法调用返回 `true` 时，它才会在此行上中断。

可以指定命中次数(而不是条件表达式)，这样程序就会在语句的执行次数达到指定值时中断执行。另一种方法是指定"筛选条件"，这样就可以根据诸如线程标识符、进程名称或线程名称之类的特性来中断程序执行。

3. 选择"关闭"以关闭对话框。

4. 通过按 F5 调试来启动程序。

5. 在控制台窗口中，在看到输入名称的提示时按 Enter 键。

6. 由于符合指定的条件(`name` 为 `null` 或 String.Empty)，因此程序会在到达断点时以及在 `Console.WriteLine` 方法执行之前停止执行。

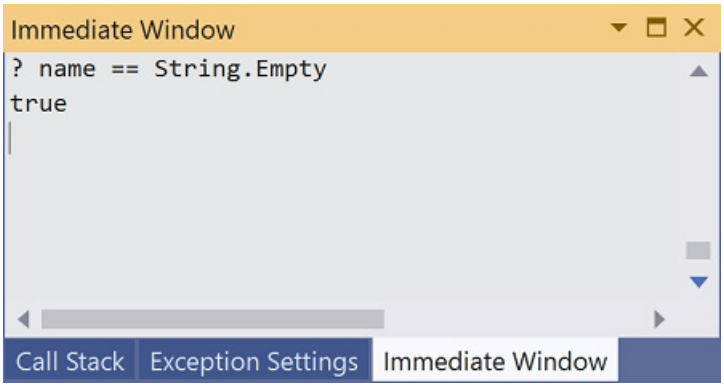
7. 选择"局部变量"窗口，其中显示当前正在执行的方法的局部变量值。在这种情况下，`Main` 是当前正在执行的方法。请注意，`name` 变量的值为 `""` 或 String.Empty。

8. 在"即时"窗口中输入下面的语句并按 Enter，确认值为空字符串。结果为 `true`。

```
? name == String.Empty
```

```
? String.IsNullOrEmpty(name)
```

问号指示即时窗口计算表达式。



9. 按 F5 继续执行程序。
10. 按任意键，关闭控制台窗口并停止调试。
11. 单击代码窗口左边缘上的点，清除断点。清除断点的其他方法是在选中代码行时按 F9 或选择“调试”>“切换断点”。

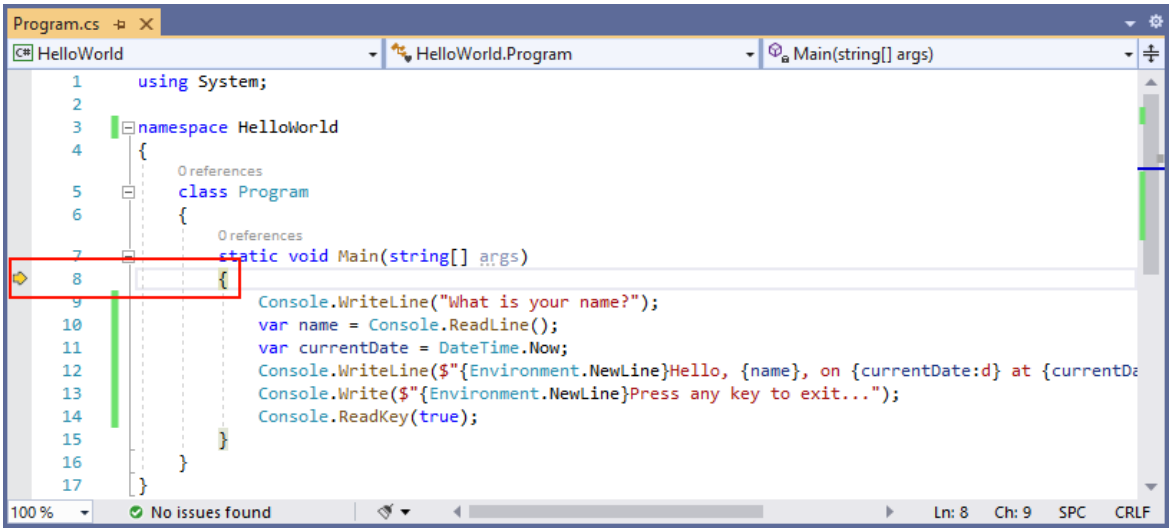
单步执行程序

使用 Visual Studio，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并通过程序代码的一小部分执行程序流。由于此程序很小，因此可以单步执行整个程序。

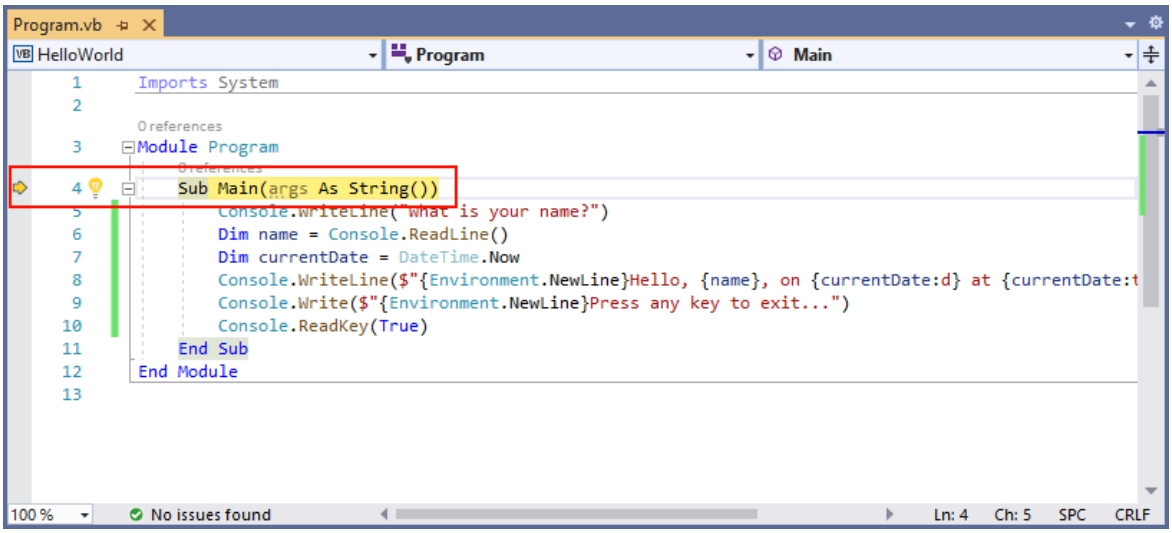
1. 选择“调试” > “单步执行”。一次调试一个语句的另一种方法是按 F11。

   Visual Studio 会在要执行的下一行旁边突出显示一个箭头。

   C#



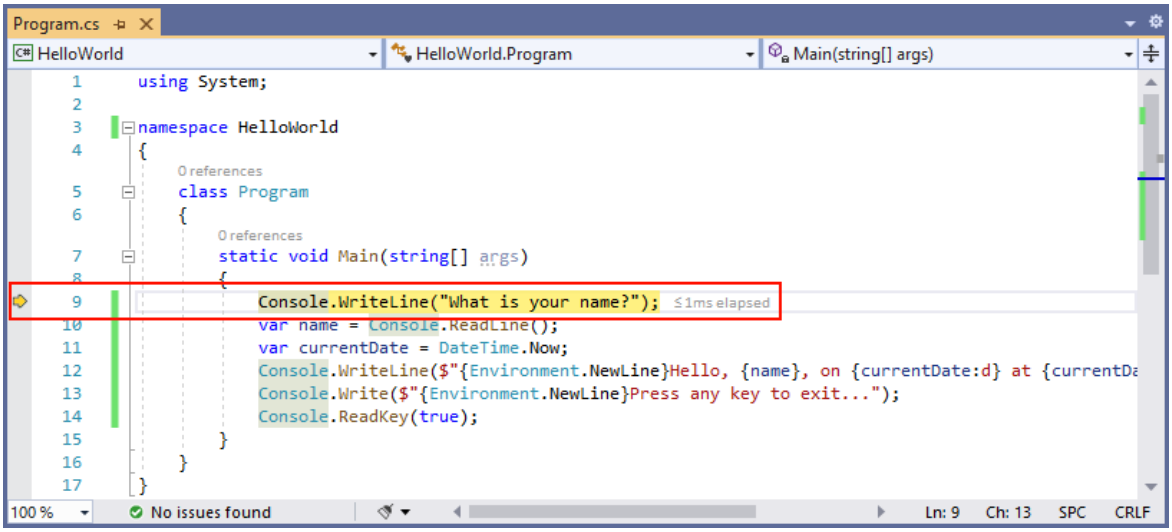
   Visual Basic



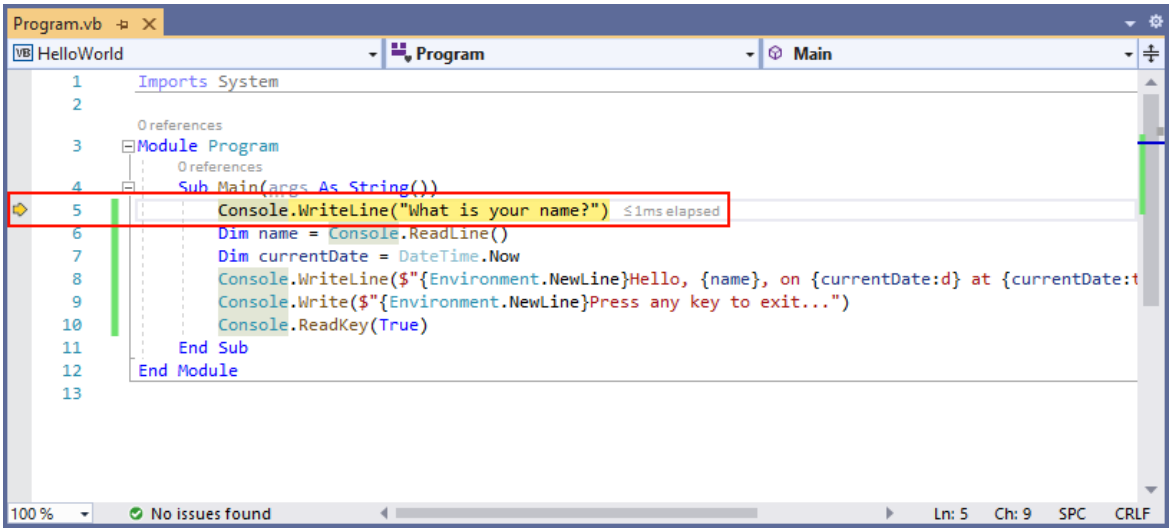
此时，“局部变量”窗口显示 `args` 数组为空，`name` 和 `currentDate` 具有默认值。此外，Visual Studio 还打开了一个空白控制台窗口。

2. 按下 F11。Visual Studio 现在突出显示要执行的下一行。“局部变量”窗口保持不变，控制台窗口仍为空白。

   C#



   Visual Basic



3. 按下 F11。Visual Studio 突出显示包含 `name` 变量赋值的语句。“局部变量”窗口显示 `name` 为 `null`，控制台窗口显示字符串“What is your name?”。

4. 在控制台窗口中输入字符串，然后按 Enter，从而响应提示。控制台无响应，输入的字符串未显示在控制台窗口中，但 Console.ReadLine 方法将捕获输入。

5. 按下 F11。Visual Studio 突出显示包含 `currentDate` 变量赋值的语句。“局部变量”窗口显示 Console.ReadLine 方法调用返回的值。控制台窗口还显示在提示符处输入的字符串。

6. 按下 F11。“局部变量”窗口显示通过 DateTime.Now 属性赋值后的 `currentDate` 变量值。控制台窗口保持不变。

7. 按下 F11。Visual Studio 调用 Console.WriteLine(String, Object, Object) 方法。控制台窗口会显示格式化的字符串。

8. 选择“调试” > “跳出”。停止分步执行的另一种方法是按 Shift+F11。

   控制台窗口会显示一条消息，并等待用户按任意键。

9. 按任意键，关闭控制台窗口并停止调试。

使用“发布”生成配置

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，有时可能会对应用程序的行为产生不良影响。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要生成和测试控制台应用程序的发布版本，请将工具栏上的生成配置从“调试”更改为“发布”。

按 F5 或选择“生成”菜单中的“生成解决方案”后，Visual Studio 会编译应用程序的“发布”版本。可像测试“调试”版本一样测试“发布”版本。

后续步骤

在本教程中，你使用了 Visual Studio 调试工具。在下一教程中，你将发布应用的可部署版本。

使用 Visual Studio 发布 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio 发布 .NET 控制台应用程序

2021/11/16 •

本教程演示如何发布控制台应用，以便其他用户可以运行它。发布应用程序会创建运行应用程序所需的一组文件。若要部署文件，请将文件复制到目标计算机。

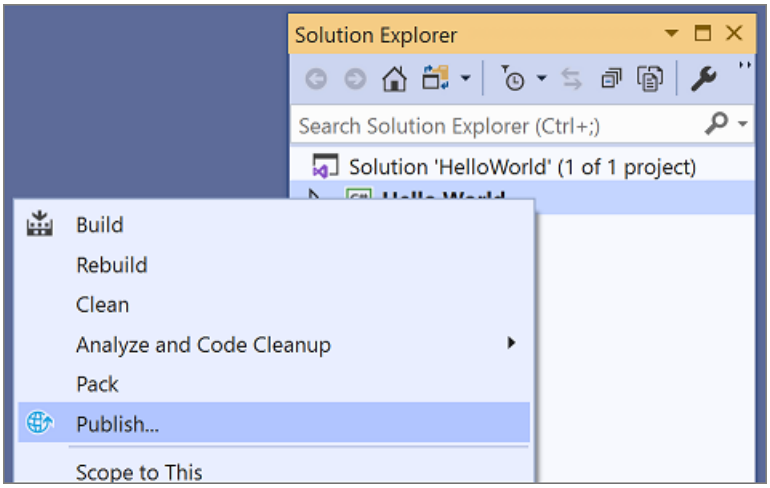
先决条件

- 本教程适用于在使用 Visual Studio 创建 .NET 控制台应用程序中创建的控制台应用。

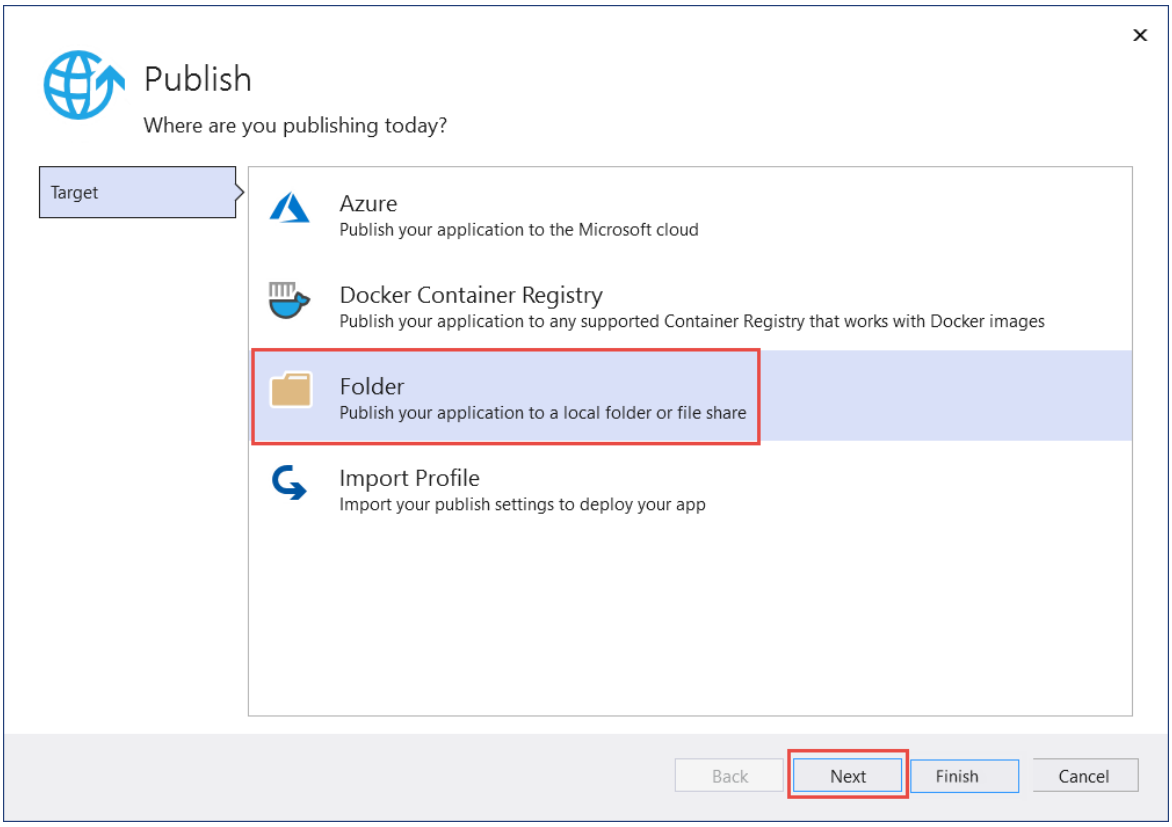
发布应用

1. 启动 Visual Studio。
2. 打开在使用 Visual Studio 创建 .NET 控制台应用程序中创建的 HelloWorld 项目。
3. 请确保 Visual Studio 正在使用“发布”生成配置。必要时，将工具栏上的生成配置设置从“调试”更改为“发布”。

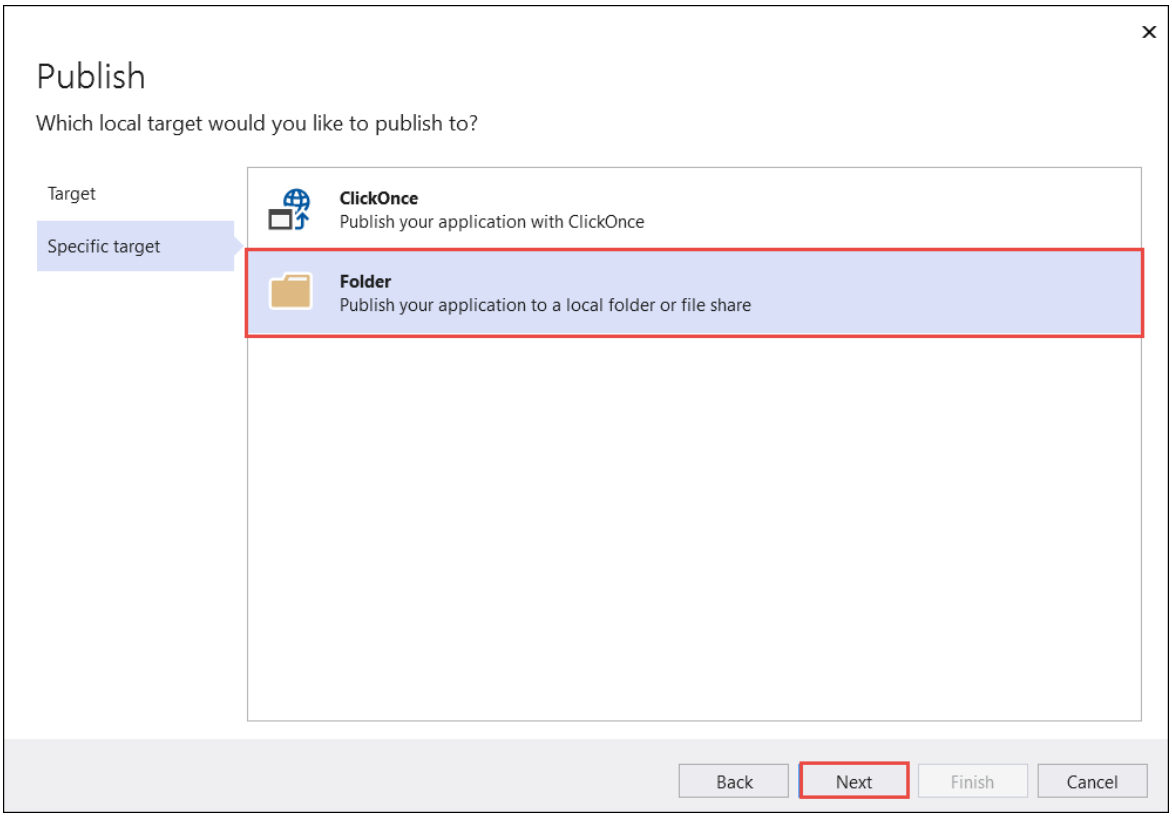
4. 右键单击“HelloWorld”项目（而不是 HelloWorld 解决方案），然后选择菜单中的“发布”。



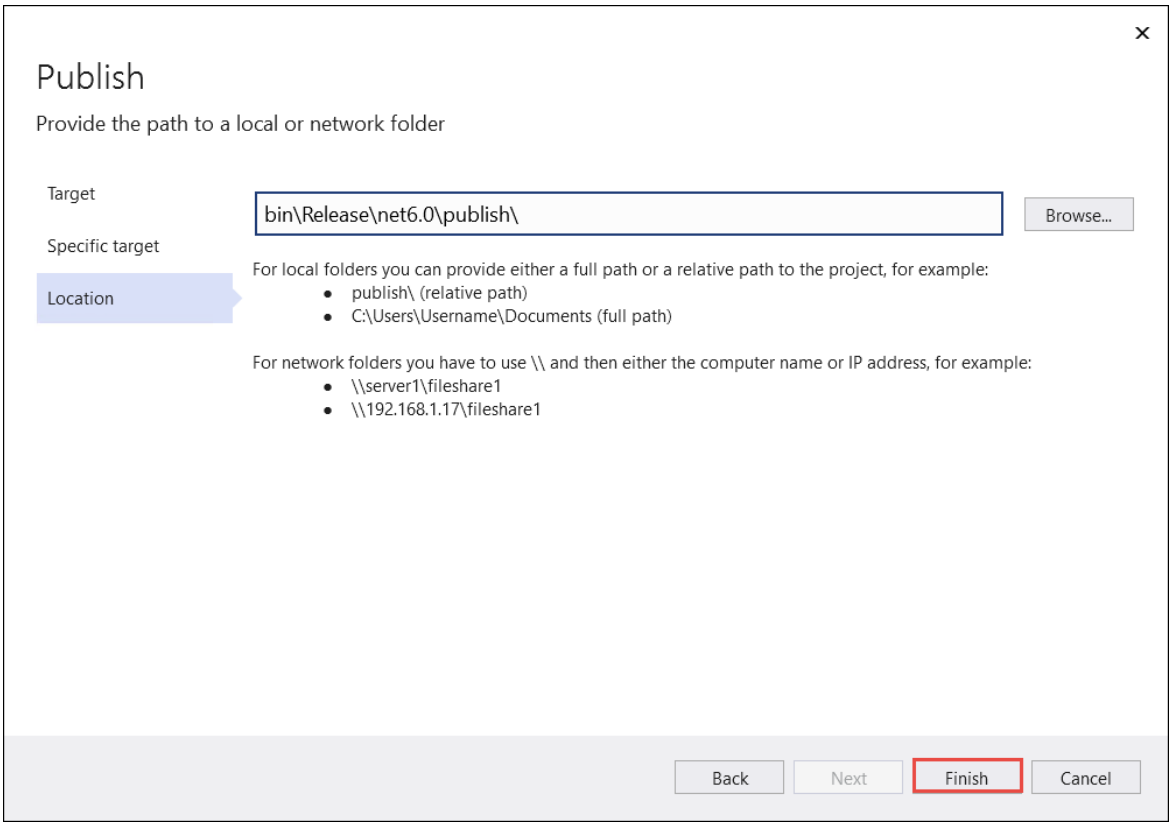
5. 在“发布”页的“目标”选项卡上，选择“文件夹”，然后选择“下一步”。



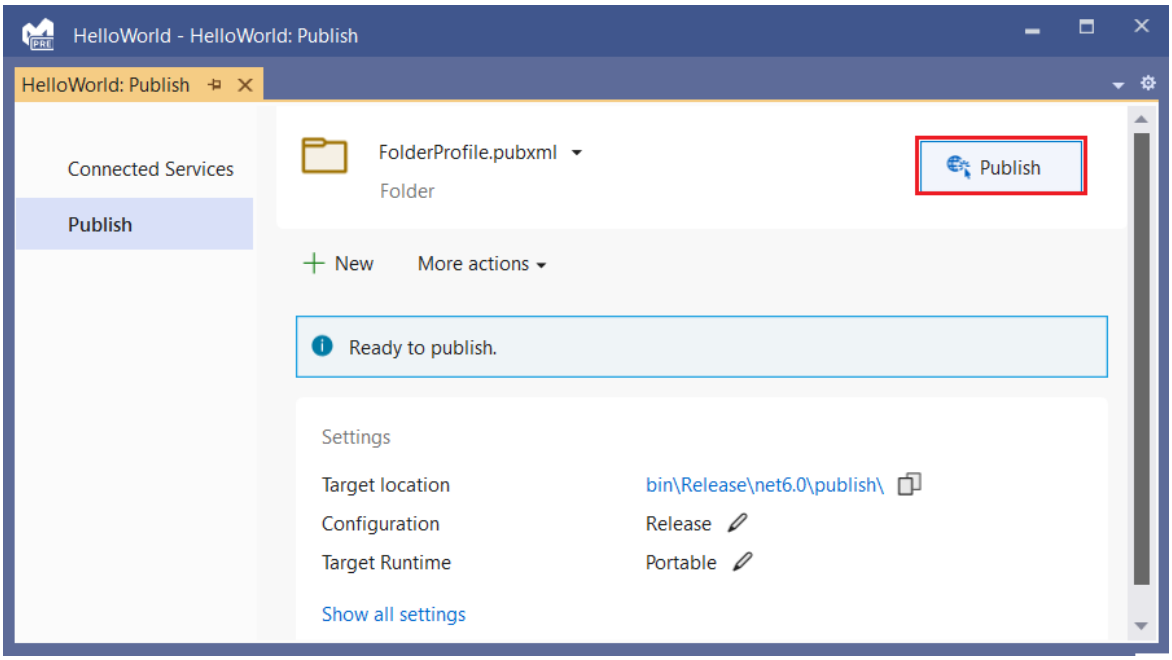
6. 在“发布”页的“特定目标”选项卡上，选择“文件夹”，然后选择“下一步”。



7. 在“发布”页的“位置”选项卡上，选择“完成”。



8. 在“发布”窗口的“发布”选项卡上，选择“发布”。



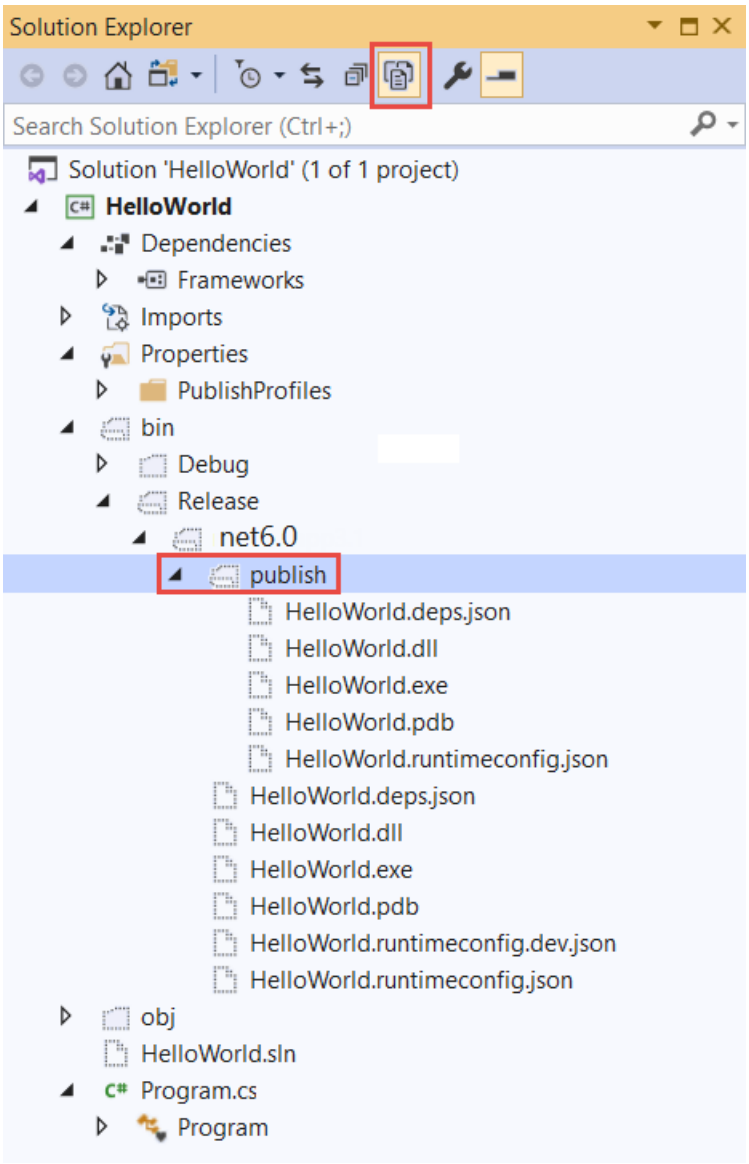
检查文件

默认情况下，发布过程会创建依赖于框架的部署，在此类部署中，已发布的应用程序在已安装 .NET 运行时的计算机上运行。用户可以通过双击可执行文件或从命令提示符发出 `dotnet HelloWorld.dll` 命令来运行发布的应用。

在下面的步骤中，查看由发布过程创建的文件。

1. 在“解决方案资源管理器”中，选择“显示所有文件”。

2. 在项目文件夹中，展开 bin/Release/net5.0/publish。



如下图所示，已发布的输出包括以下文件：

- HelloWorld.deps.json

  这是应用程序的运行时依赖项文件。该文件定义了运行应用所需的 .NET 组件和库（包括包含应用程序的动态链接库）。有关详细信息，请参阅运行时配置文件。

- HelloWorld.dll

  这是应用程序的依赖于框架的部署版本。若要执行此动态链接库，请在命令提示符处输入 `dotnet HelloWorld.dll`。这种运行应用的方法适用于安装了 .NET 运行时的任何平台。

- *HelloWorld.exe*

  这是应用程序的依赖于框架的可执行文件版本。若要运行该版本，请在命令提示符处输入 `HelloWorld.exe`。文件特定于操作系统。

- HelloWorld.pdb（对于部署是可选的）

  这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- HelloWorld.runtimeconfig.json

  这是应用程序的运行时配置文件。该文件标识用于运行应用程序的 .NET 版本。还可向其添加配置选项。有关详细信息，请参阅 .NET 运行时配置设置。

运行已发布的应用

1. 在“解决方案资源管理器”中，右键单击“模型”文件夹，然后选择“复制完整路径”。
2. 打开命令提示符，然后导航到“发布”文件夹。为此，请输入 `cd`，然后粘贴完整路径。例如：

   ```
   cd C:\Projects\HelloWorld\bin\Release\net6.0\publish\
   ```

3. 使用可执行文件运行应用：
   a. 输入 `HelloWorld.exe`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。
4. 使用 `dotnet` 命令运行应用：
   a. 输入 `dotnet HelloWorld.dll`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。

其他资源

- .NET 应用程序部署

后续步骤

在本教程中，你发布了一个控制台应用。在下一教程中，你将创建类库。

使用 Visual Studio 创建 .NET 类库

本教程演示如何发布控制台应用，以便其他用户可以运行它。发布应用程序会创建运行应用程序所需的一组文件。若要部署文件，请将文件复制到目标计算机。

先决条件

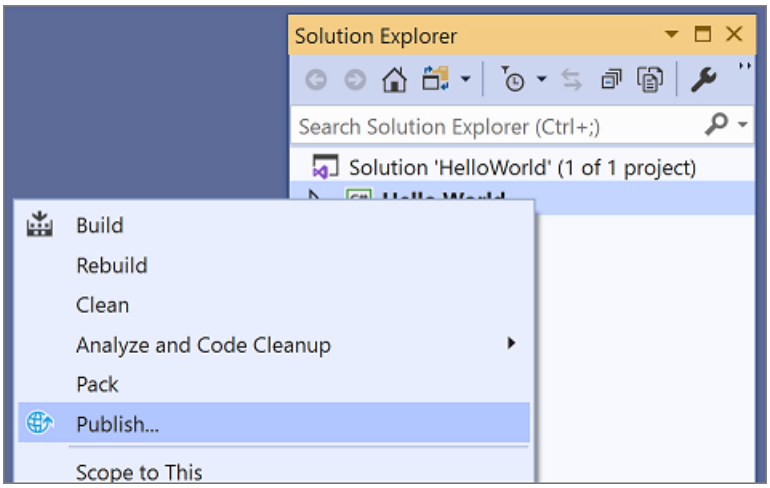
- 本教程适用于在使用 Visual Studio 创建 .NET 控制台应用程序中创建的控制台应用。

发布应用

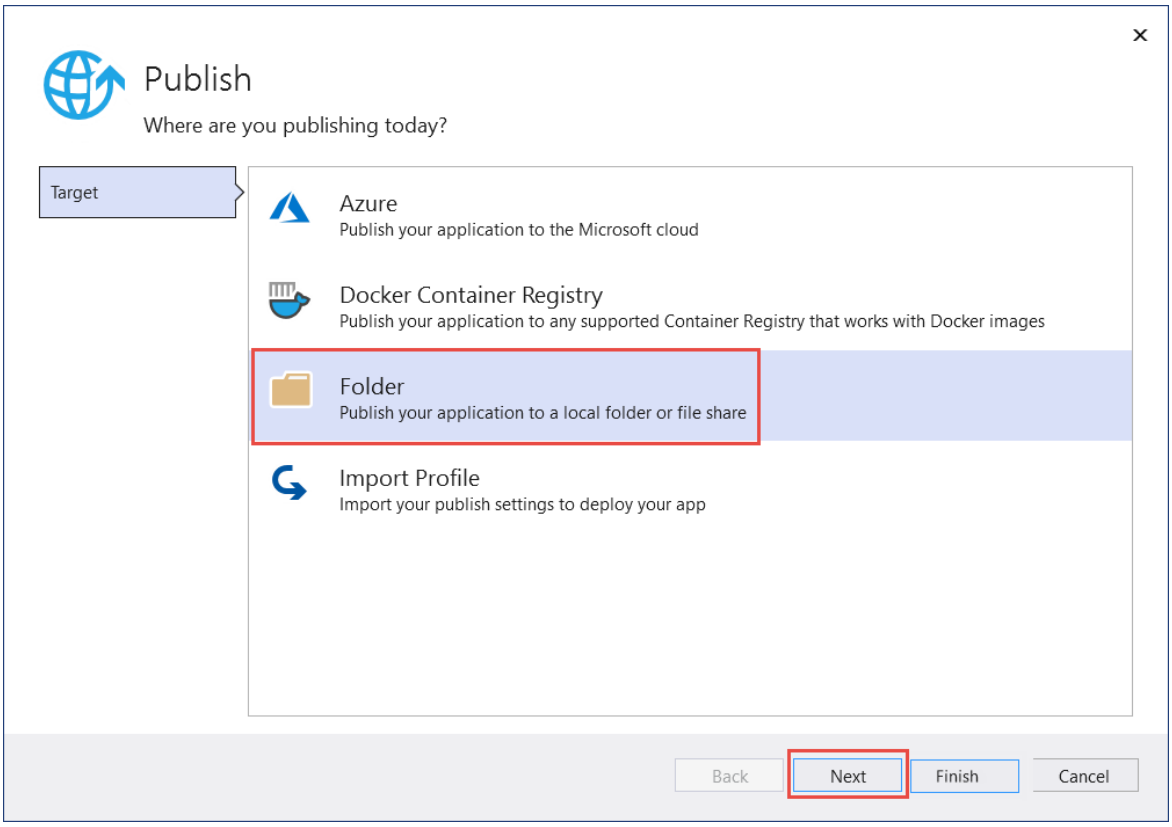
1. 启动 Visual Studio。
2. 打开在使用 Visual Studio 创建 .NET 控制台应用程序中创建的 HelloWorld 项目。
3. 请确保 Visual Studio 正在使用“发布”生成配置。必要时，将工具栏上的生成配置设置从“调试”更改为“发布”。

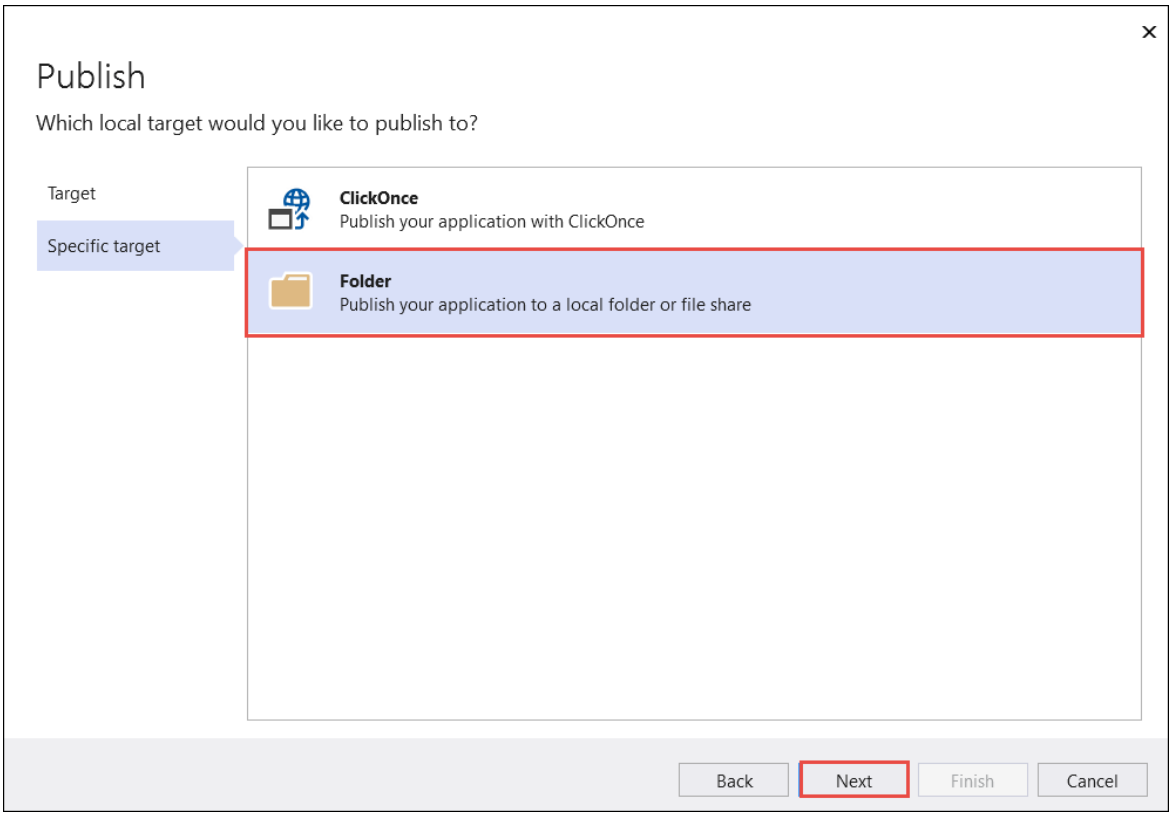
4. 右键单击“HelloWorld”项目（而不是 HelloWorld 解决方案），然后选择菜单中的“发布”。



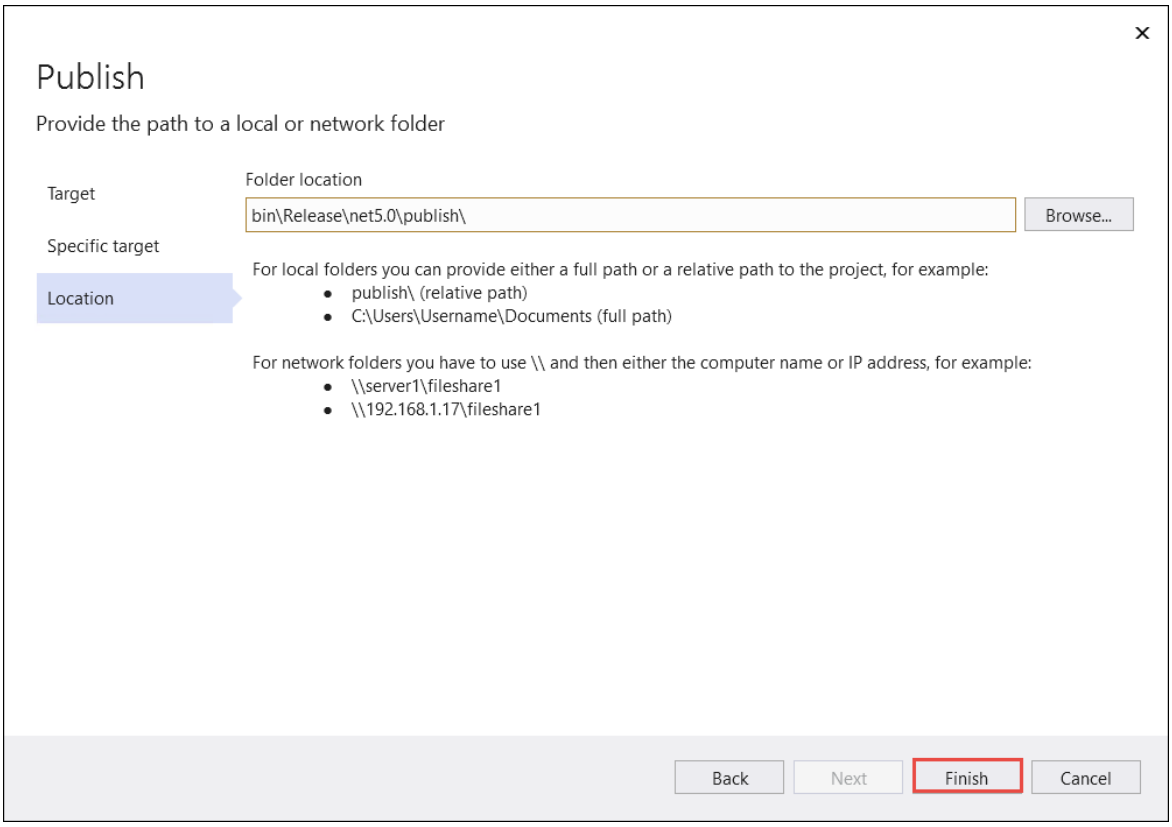
5. 在“发布”页的“目标”选项卡上，选择“文件夹”，然后选择“下一步”。



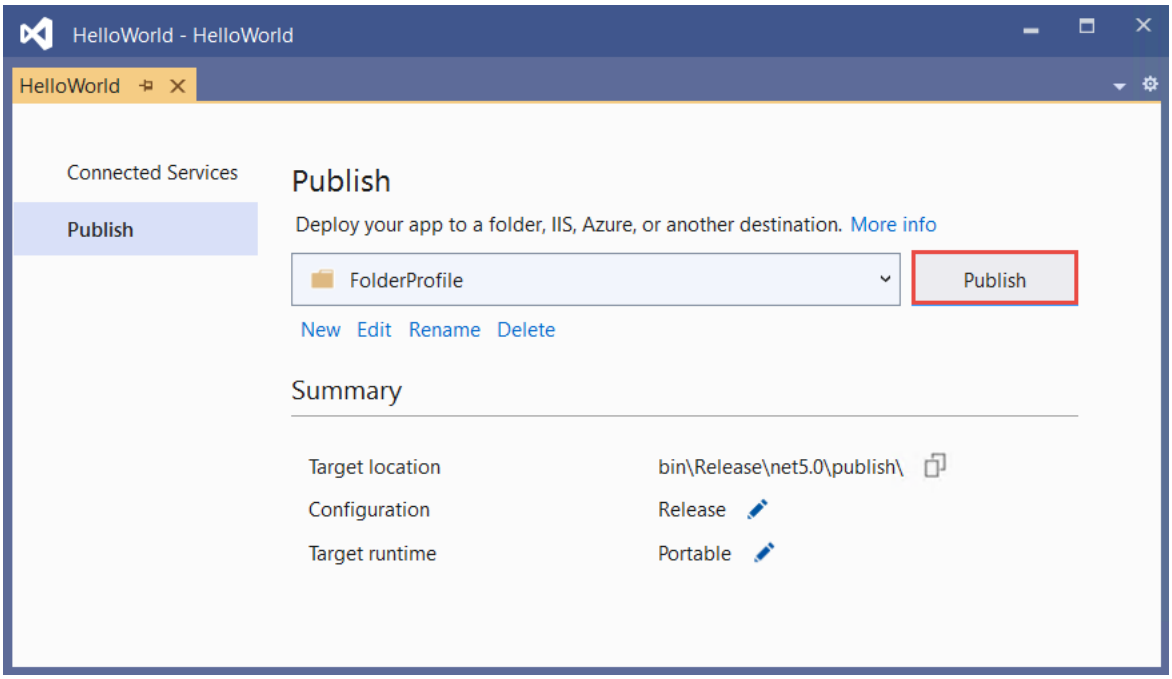
6. 在“发布”页的“特定目标”选项卡上，选择“文件夹”，然后选择“下一步”。



7. 在“发布”页的“位置”选项卡上，选择“完成”。



8. 在“发布”窗口的“发布”选项卡上，选择“发布”。

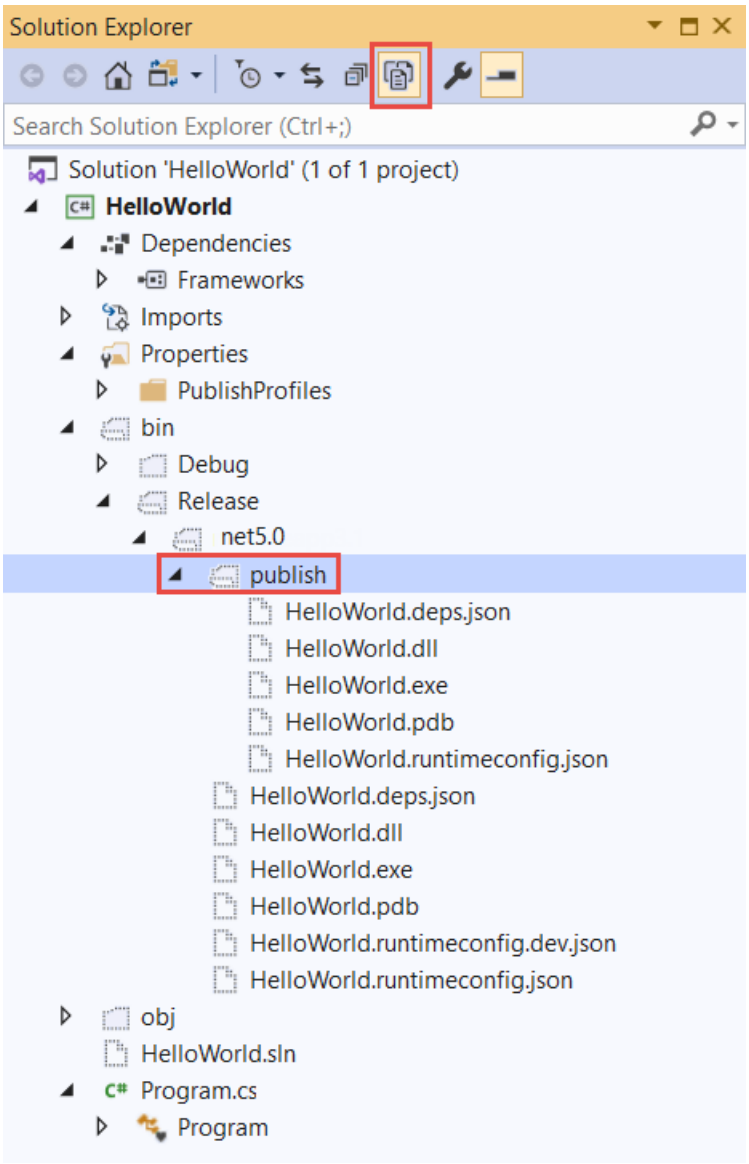


检查文件

默认情况下，发布过程会创建依赖于框架的部署，在此类部署中，已发布的应用程序在已安装 .NET 运行时的计算机上运行。用户可以通过双击可执行文件或从命令提示符发出 `dotnet HelloWorld.dll` 命令来运行发布的应用。

在下面的步骤中，查看由发布过程创建的文件。

1. 在“解决方案资源管理器”中，选择“显示所有文件”。
2. 在项目文件夹中，展开 bin/Release/net5.0/publish。



如下图所示，已发布的输出包括以下文件：

- HelloWorld.deps.json

  这是应用程序的运行时依赖项文件。该文件定义了运行应用所需的 .NET 组件和库（包括包含应用程序的动态链接库）。有关详细信息，请参阅运行时配置文件。

- HelloWorld.dll

  这是应用程序的依赖于框架的部署版本。若要执行此动态链接库，请在命令提示符处输入 `dotnet HelloWorld.dll`。这种运行应用的方法适用于安装了 .NET 运行时的任何平台。

- *HelloWorld.exe*

  这是应用程序的依赖于框架的可执行文件版本。若要运行该版本，请在命令提示符处输入 `HelloWorld.exe`。文件特定于操作系统。

- HelloWorld.pdb（对于部署是可选的）

  这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- HelloWorld.runtimeconfig.json

  这是应用程序的运行时配置文件。该文件标识用于运行应用程序的 .NET 版本。还可向其添加配置选项。有关详细信息，请参阅 .NET 运行时配置设置。

运行已发布的应用

1. 在“解决方案资源管理器”中，右键单击“模型”文件夹，然后选择“复制完整路径”。
2. 打开命令提示符，然后导航到“发布”文件夹。为此，请输入 `cd`，然后粘贴完整路径。例如：

   ```
   cd C:\Projects\HelloWorld\bin\Release\net5.0\publish\
   ```

3. 使用可执行文件运行应用：
   a. 输入 `HelloWorld.exe`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。
4. 使用 `dotnet` 命令运行应用：
   a. 输入 `dotnet HelloWorld.dll`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。

其他资源

- .NET 应用程序部署

后续步骤

在本教程中，你发布了一个控制台应用。在下一教程中，你将创建类库。

使用 Visual Studio 创建 .NET 类库

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio 创建 .NET 类库

2021/11/16 •

在本教程中，将创建包含一个字符串处理方法的简单类库。

类库定义的是可以由应用程序调用的类型和方法。如果库以 .NET Standard 2.0 为目标，则支持 .NET Standard 2.0 的任何 .NET 实现（包括 .NET Framework）均可调用该库。如果库以 .NET 6 为目标，则以 .NET 6 为目标的任何应用程序均可调用该库。本教程演示如何以 .NET 6 为目标。

创建类库后，可将其作为 NuGet 包或作为与使用该类库的应用程序捆绑在一起的组件进行分发。

必备条件

- 安装了具有 .NET 桌面开发工作负载的 Visual Studio 2022 版本 17.0.0 预览版。选择此工作负载时，将自动安装 .NET 6 SDK。

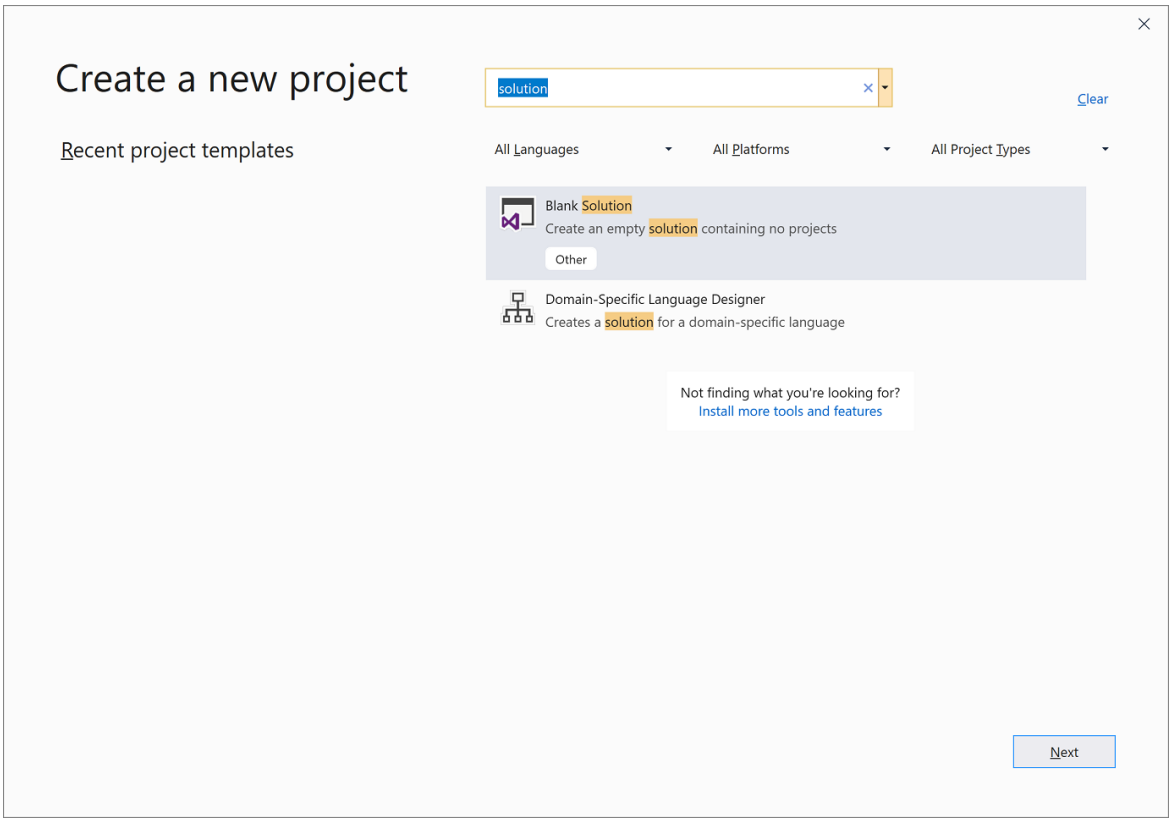
  有关详细信息，请参阅使用 Visual Studio 安装 .NET SDK。

创建解决方案

首先，创建一个空白解决方案来放置类库项目。Visual Studio 解决方案用作一个或多个项目的容器。将其他相关项目添加到同一个解决方案中。

创建空白解决方案：

1. 启动 Visual Studio。
2. 在“开始”窗口上，选择“创建新项目”。
3. 在“创建新项目”页面上，在搜索框中输入“解决方案”。选择“空白解决方案”模板，然后选择“下一步”。



4. 在“配置新项目”页面上，在“解决方案名称”框中输入“ClassLibraryProjects”。然后选择“创建”。

创建类库项目

1. 将名为“StringLibrary”的新 .NET 类库项目添加到解决方案。

   a. 在“解决方案资源管理器”中，右键单击解决方案并选择“添加” > “新建项目” 。

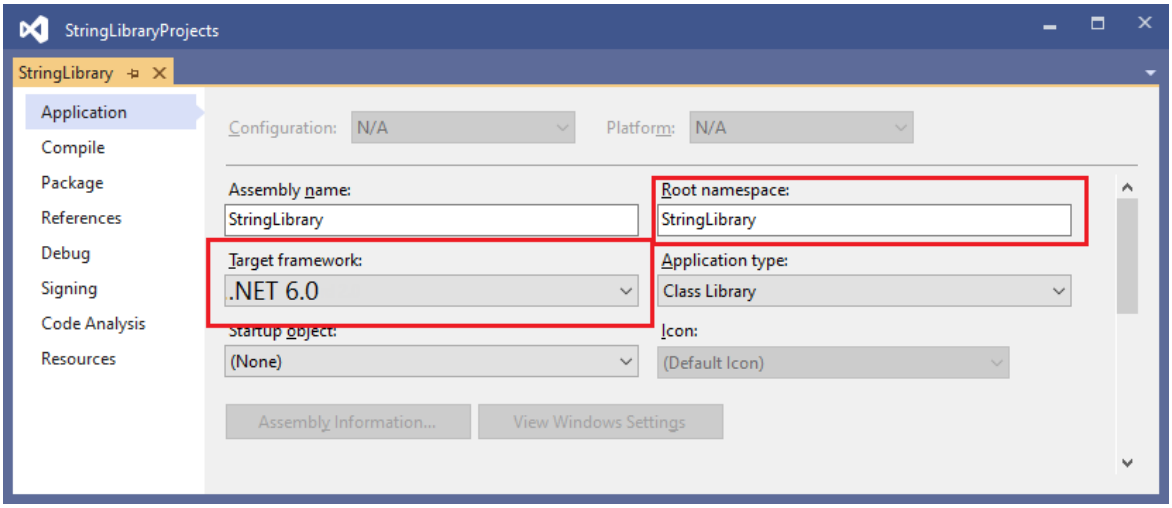
   b. 在“创建新项目”页面上，在搜索框中输入“库”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“类库”模板，然后选择“下一步” 。

   c. 在“配置新项目”页的“项目名称”框中，输入“StringLibrary”，然后选择“下一步” 。

   d. 在“其他信息”页上，选择“.NET 6 (长期支持)”，然后选择“创建” 。

2. 请检查以确保库以 .NET 的正确版本为目标。右键单击“解决方案资源管理器”中的库项目，然后选择“属性”。“目标框架”文本框显示项目以 .NET 6.0 为目标。

3. 如果使用 Visual Basic，请清除“根命名空间”文本框中的文本。



   对于每个项目，Visual Basic 会自动创建一个与项目名称对应的命名空间。在本教程中，通过使用代码文件中的 `namespace` 关键字定义顶级命名空间。

4. 将 Class1.cs 或 Class1.vb 代码窗口中的代码替换为以下代码，并保存文件 。如果未显示想要使用的语言，请更改页面顶部的语言选择器。

```
namespace UtilityLibraries;

public static class StringLibrary
{
    public static bool StartsWithUpper(this string? str)
    {
        if (string.IsNullOrWhiteSpace(str))
            return false;

        char ch = str[0];
        return char.IsUpper(ch);
    }
}
```

```
Imports System.Runtime.CompilerServices

Namespace UtilityLibraries
    Public Module StringLibrary
        <Extension>
        Public Function StartsWithUpper(str As String) As Boolean
            If String.IsNullOrWhiteSpace(str) Then
                Return False
            End If

            Dim ch As Char = str(0)
            Return Char.IsUpper(ch)
        End Function
    End Module
End Namespace
```

类库 `UtilityLibraries.StringLibrary` 包含一个名为 `StartsWithUpper` 的方法。此方法会返回 Boolean 值，以指明当前字符串实例是否以大写字符开头。Unicode 标准会区分大小写字符。如果为大写字符，Char.IsUpper(Char) 方法返回 `true`。

`StartsWithUpper` 以扩展方法的形式进行实现，这样就可以将其作为 String 类成员进行调用。`string` 后的问号 (`?`) 表示该字符串可能为 null。

5. 在菜单栏上，选择“生成” > “生成解决方案”或按 `Ctrl+Shift+B`，验证项目是否编译正确 。

向解决方案添加控制台应用

添加使用类库的控制台应用程序。应用将提示用户输入字符串，并报告字符串是否以大写字符开头。

1. 将名为“ShowCase”的新 .NET 控制台应用程序添加到解决方案。
   a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加” > “新建项目”。
   b. 在“创建新项目”页面，在搜索框中输入“控制台”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。
   c. 选择“控制台应用程序”模板，然后选择“下一步” 。
   d. 在“配置新项目”页面，在“项目名称”框中输入“ShowCase”。然后选择“下一步” 。
   e. 在“其他信息”页的“框架”框中选择“.NET 6 (长期支持)” 。然后选择“创建”。
2. 在“Program.cs”或“Program.vb”文件的代码窗口中，将所有代码替换为以下代码 。

```
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string? input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input}");
            Console.WriteLine("Begins with uppercase? " +
                 $"{(input.StartsWithUpper() ? "Yes" : "No")}");
            Console.WriteLine();
            row += 4;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}");
            row = 3;
        }
    }
}
```

```
Imports UtilityLibraries

Module Program
    Dim row As Integer = 0

    Sub Main()
        Do
            If row = 0 OrElse row >= 25 Then ResetConsole()

            Dim input As String = Console.ReadLine()
            If String.IsNullOrEmpty(input) Then Return

            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{If(input.StartsWithUpper(), "Yes", "No")} {Environment.NewLine}")
            row += 3
        Loop While True
    End Sub

    Private Sub ResetConsole()
        If row > 0 Then
            Console.WriteLine("Press any key to continue...")
            Console.ReadKey()
        End If
        Console.Clear()
        Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}")
        row = 3
    End Sub
End Module
```

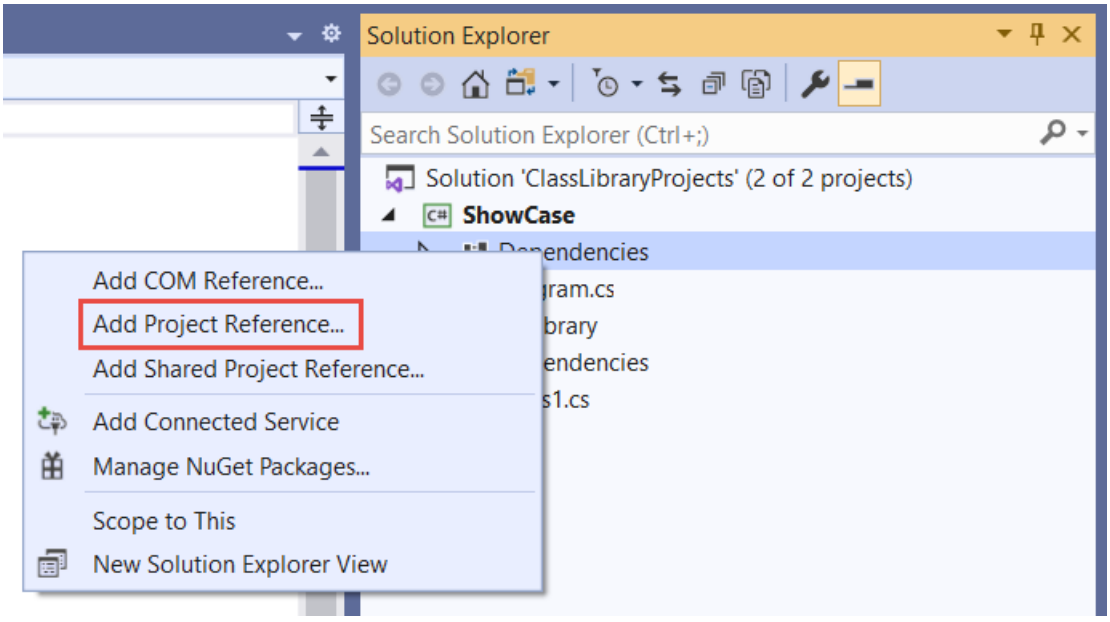
该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

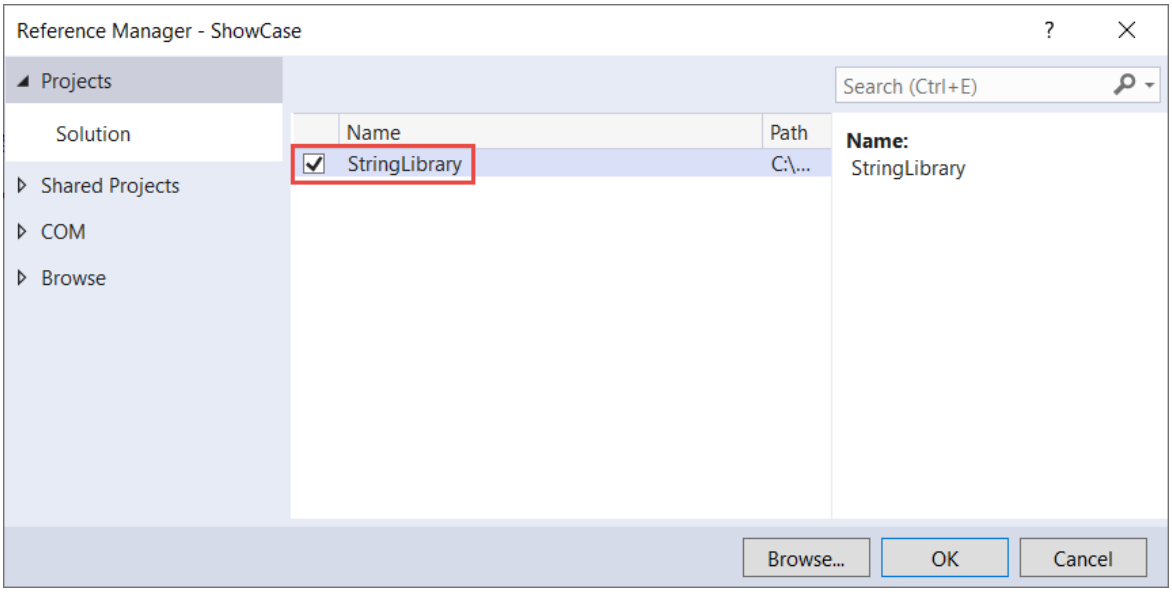
添加项目引用

最初，新的控制台应用项目无权访问类库。若要允许该项目调用类库中的方法，可以创建对类库项目的项目引用。

1. 在“解决方案资源管理器”中，右键单击 `ShowCase` 项目的“依赖项”节点，并选择“添加项目引用”。

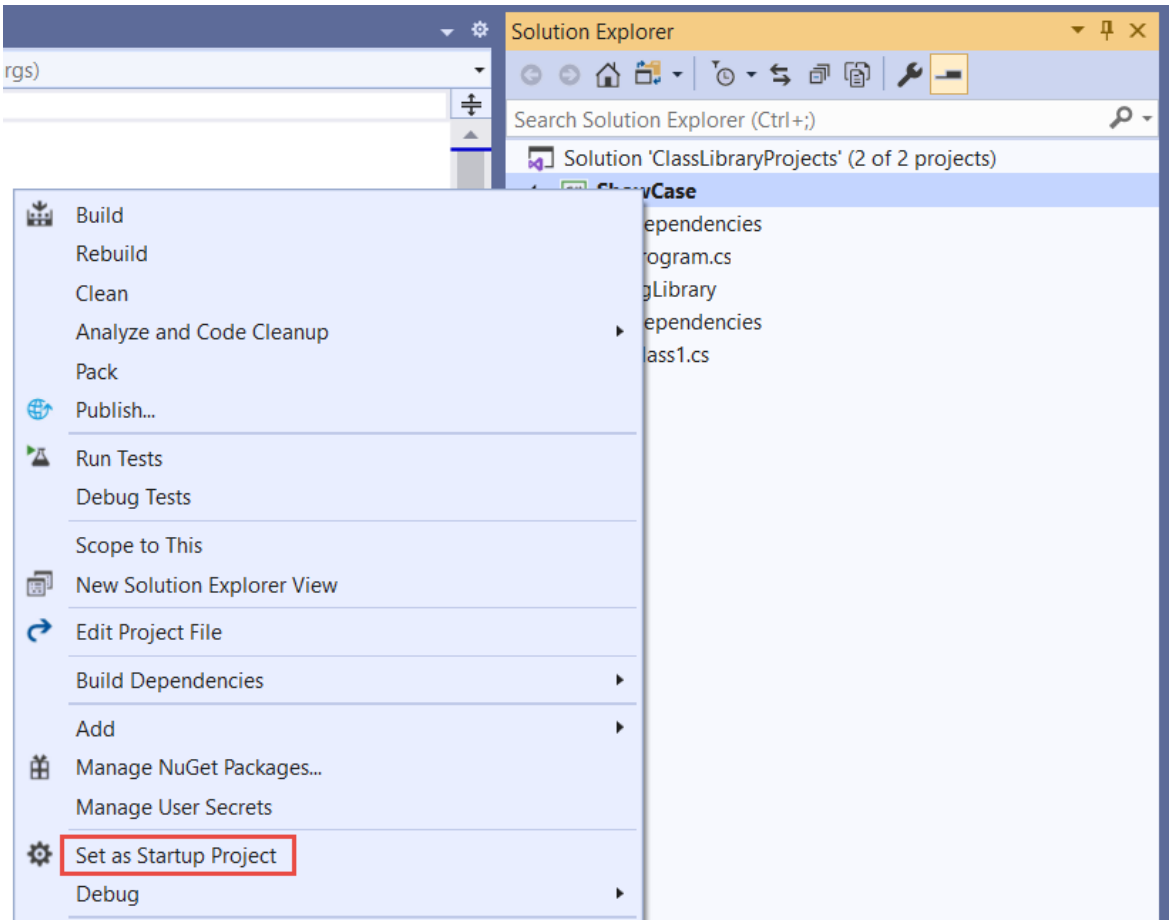


2. 在“引用管理器”对话框中，选择“StringLibrary”项目，然后选择“确定”按钮 。



运行应用

1. 在“解决方案资源管理器”中，右键单击“ShowCase”项目，在上下文菜单中选择“设为启动项目”。



2. 按 Ctrl+F5 编译并运行程序，而不进行调试。

3. 输入字符串并按 Enter 以试用程序，然后按 Enter 退出。

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:

Hello
Input: Hello
Begins with uppercase? Yes

hello
Input: hello
Begins with uppercase? No
```

其他资源

- 使用 .NET CLI 开发库
- .NET Standard 版本及其支持的平台。

后续步骤

在本教程中，你创建了一个类库。在下一教程中，你将了解如何对类库进行单元测试。

使用 Visual Studio 对 .NET 类库进行单元测试

或者，你可以跳过自动单元测试，并了解如何通过创建 NuGet 包来共享库：

使用 Visual Studio 创建和发布包

同时，还可以了解如何发布控制台应用。如果从本教程中创建的解决方案发布控制台应用，类库将以 .dll 文件的形式随附。

使用 Visual Studio 发布 .NET 控制台应用程序

在本教程中，将创建包含一个字符串处理方法的简单类库。

类库定义的是可以由应用程序调用的类型和方法。如果库以 .NET Standard 2.0 为目标，则支持 .NET Standard 2.0 的任何 .NET 实现（包括 .NET Framework）均可调用该库。如果库以 .NET 5 为目标，则以 .NET 5 为目标的任何应用程序均可调用该库。本教程演示如何以 .NET 5 为目标。

创建类库后，可将其作为 NuGet 包或作为与使用该类库的应用程序捆绑在一起的组件进行分发。

必备条件

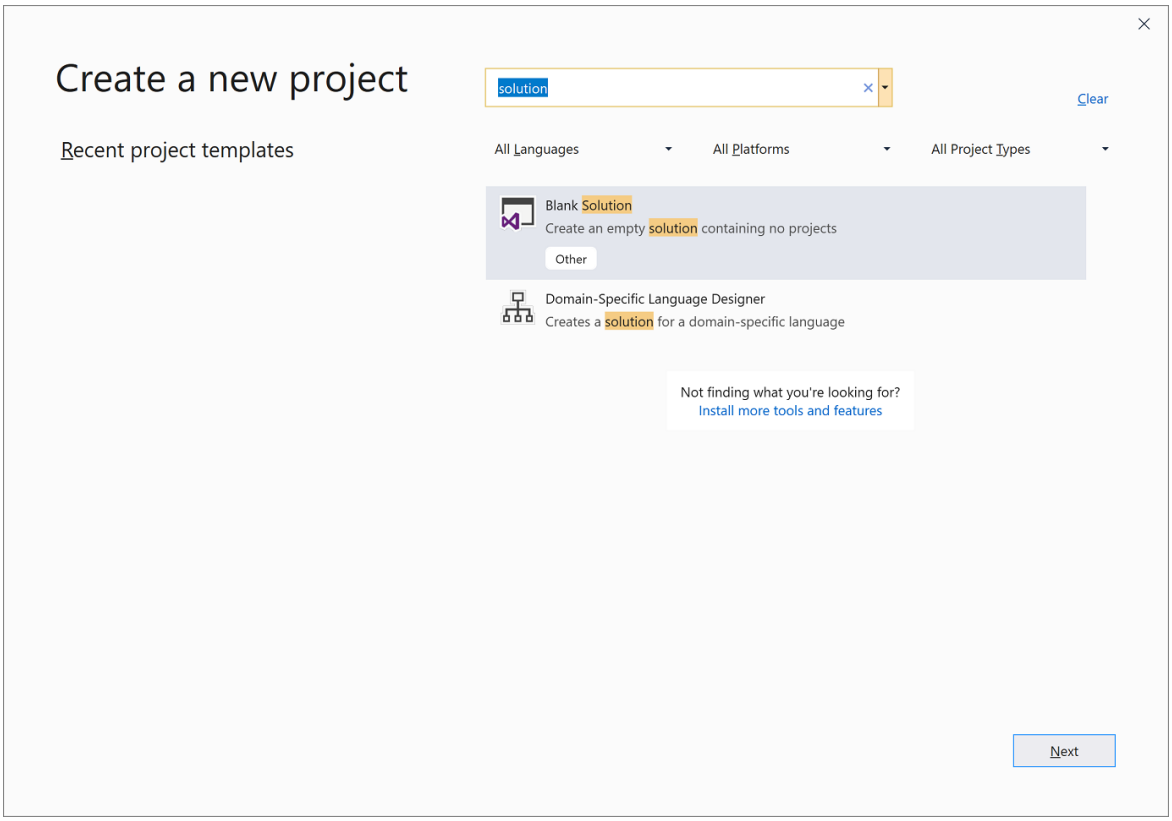
- 安装了具有“.NET Core 跨平台开发”工作负载的 Visual Studio 2019 版本 16.8 或更高版本。选择此工作负载时，将自动安装 .NET 5.0 SDK。本教程假设已启用“在‘新建项目’中显示所有 .NET Core 模板”，如教程：使用 Visual Studio 创建 .NET 控制台应用程序中所示。

创建解决方案

首先，创建一个空白解决方案来放置类库项目。Visual Studio 解决方案用作一个或多个项目的容器。将其他相关项目添加到同一个解决方案中。

创建空白解决方案：

1. 启动 Visual Studio。
2. 在“开始”窗口上，选择“创建新项目”。
3. 在“创建新项目”页面上，在搜索框中输入“解决方案”。选择“空白解决方案”模板，然后选择“下一步”。



4. 在“配置新项目”页面上，在“项目名称”框中输入“ClassLibraryProjects”。然后选择“创建”。

创建类库项目

1. 将名为“StringLibrary”的新 .NET 类库项目添加到解决方案。

   a. 在“解决方案资源管理器”中，右键单击解决方案并选择“添加” > “新建项目” 。

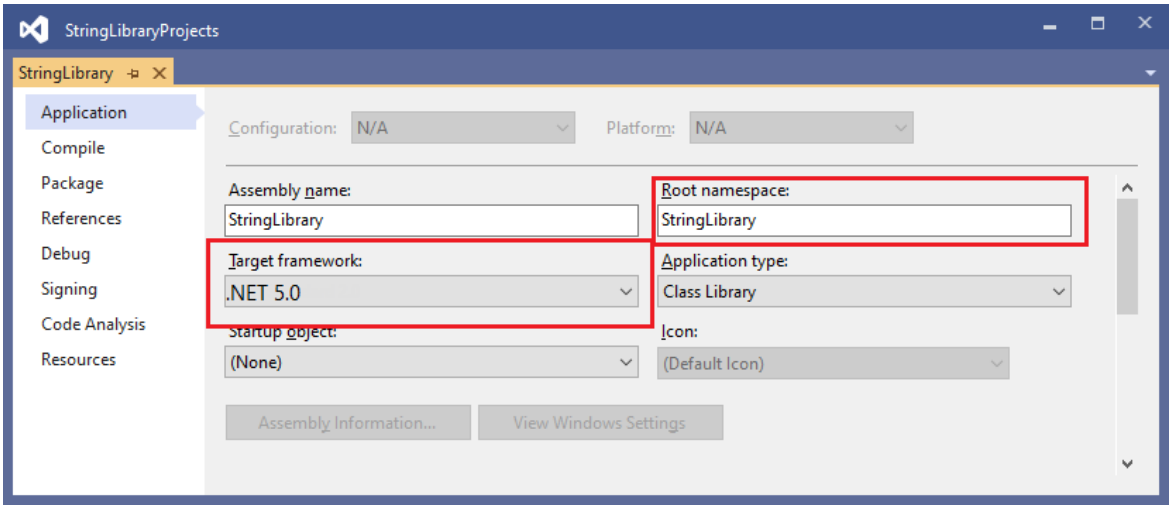
   b. 在“创建新项目”页面上，在搜索框中输入“库”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。选择“类库”模板，然后选择“下一步” 。

   c. 在“配置新项目”页的“项目名称”框中，输入“StringLibrary”，然后选择“下一步” 。

   d. 在“其他信息”页上，选择“.NET 5.0 (当前)”，然后选择“创建” 。

2. 请检查以确保库以 .NET 的正确版本为目标。右键单击“解决方案资源管理器”中的库项目，然后选择“属性”。“目标框架”文本框显示项目以 .NET 5.0 为目标。

3. 如果使用 Visual Basic，请清除“根命名空间”文本框中的文本。



对于每个项目，Visual Basic 会自动创建一个与项目名称对应的命名空间。在本教程中，通过使用代码文

件中的 `namespace` 关键字定义顶级命名空间。

4. 将 Class1.cs 或 Class1.vb 代码窗口中的代码替换为以下代码，并保存文件 。如果未显示想要使用的语言，请更改页面顶部的语言选择器。

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this string str)
        {
            if (string.IsNullOrWhiteSpace(str))
                return false;

            char ch = str[0];
            return char.IsUpper(ch);
        }
    }
}
```

```
Imports System.Runtime.CompilerServices

Namespace UtilityLibraries
    Public Module StringLibrary
        <Extension>
        Public Function StartsWithUpper(str As String) As Boolean
            If String.IsNullOrWhiteSpace(str) Then
                Return False
            End If

            Dim ch As Char = str(0)
            Return Char.IsUpper(ch)
        End Function
    End Module
End Namespace
```

类库 `UtilityLibraries.StringLibrary` 包含一个名为 `StartsWithUpper` 的方法。此方法会返回 Boolean 值，以指明当前字符串实例是否以大写字符开头。Unicode 标准会区分大小写字符。如果为大写字符，Char.IsUpper(Char) 方法返回 `true` 。

`StartsWithUpper` 以扩展方法的形式进行实现，这样就可以将其作为 String 类成员进行调用。

5. 在菜单栏上，选择“生成” > “生成解决方案”或按 `Ctrl+Shift+B`，验证项目是否编译正确 。

向解决方案添加控制台应用

添加使用类库的控制台应用程序。应用将提示用户输入字符串，并报告字符串是否以大写字符开头。

1. 将名为“ShowCase”的新 .NET 控制台应用程序添加到解决方案。

   a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加” > “新建项目”。

   b. 在“创建新项目”页面，在搜索框中输入“控制台”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。

   c. 选择“控制台应用程序”模板，然后选择“下一步” 。

   d. 在“配置新项目”页面，在“项目名称”框中输入“ShowCase”。然后选择“下一步” 。

   e. 在“其他信息”页上，选择“目标框架”框中的“.NET 5.0 (当前)” 。然后选择“创建”。

2. 在“Program.cs”或“Program.vb”文件的代码窗口中，将所有代码替换为以下代码 。

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{(input.StartsWithUpper() ? "Yes" : "No")}{Environment.NewLine}");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}");
            row = 3;
        }
    }
}
```

```
Imports UtilityLibraries

Module Program
    Dim row As Integer = 0

    Sub Main()
        Do
            If row = 0 OrElse row >= 25 Then ResetConsole()

            Dim input As String = Console.ReadLine()
            If String.IsNullOrEmpty(input) Then Return

            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{If(input.StartsWithUpper(), "Yes", "No")} {Environment.NewLine}")
            row += 3
        Loop While True
    End Sub

    Private Sub ResetConsole()
        If row > 0 Then
            Console.WriteLine("Press any key to continue...")
            Console.ReadKey()
        End If
        Console.Clear()
        Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}")
        row = 3
    End Sub
End Module
```

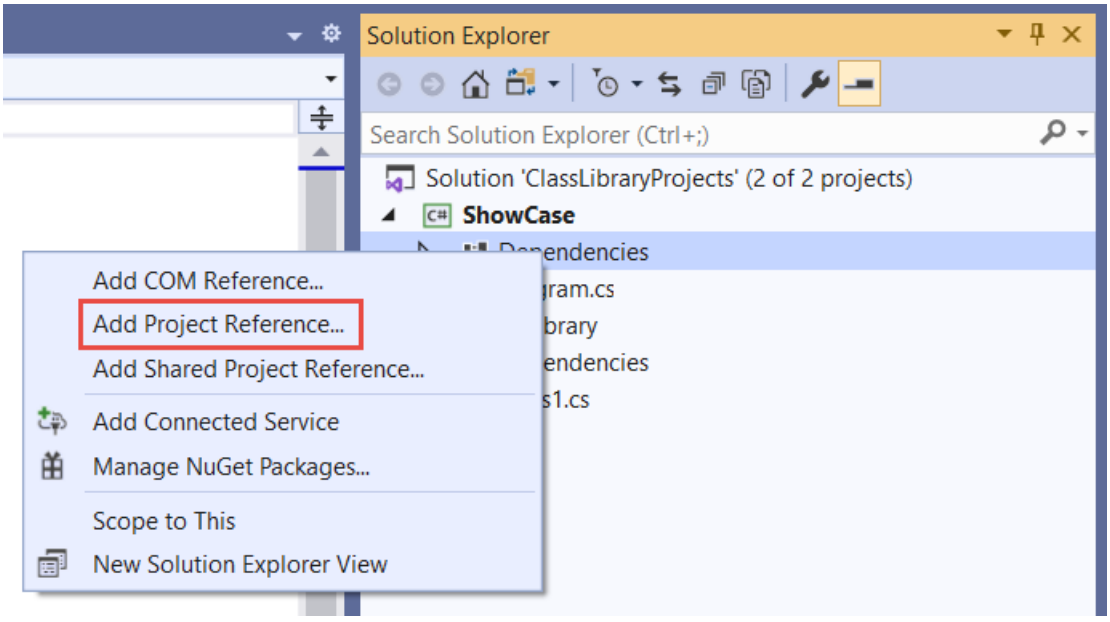
该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

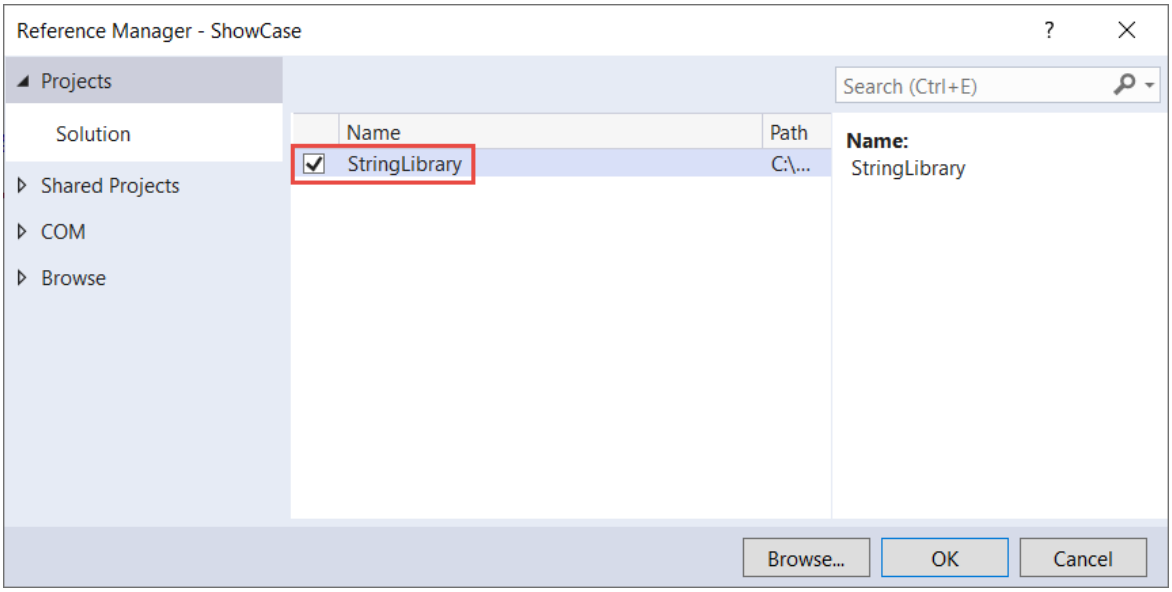
添加项目引用

最初，新的控制台应用项目无权访问类库。若要允许该项目调用类库中的方法，可以创建对类库项目的项目引用。

1. 在“解决方案资源管理器”中，右键单击 `ShowCase` 项目的“依赖项”节点，并选择“添加项目引用”。

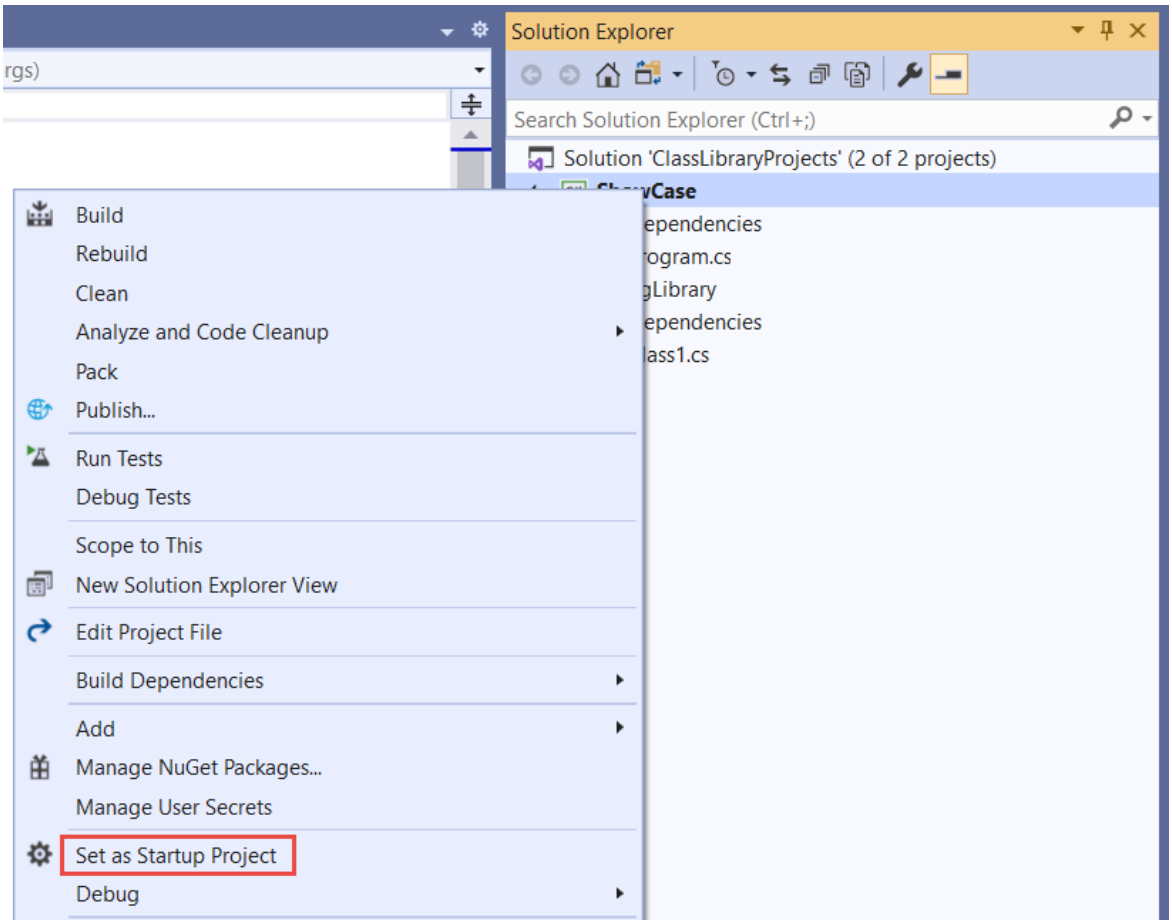


2. 在“引用管理器”对话框中，选择“StringLibrary”项目，然后选择“确定”按钮 。



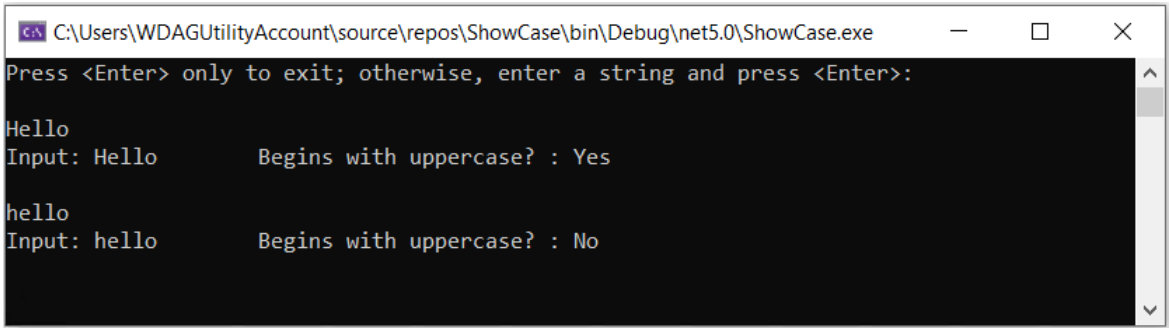
运行应用

1. 在“解决方案资源管理器”中，右键单击“ShowCase”项目，在上下文菜单中选择“设为启动项目”。



2. 按 Ctrl+F5 编译并运行程序，而不进行调试。

3. 输入字符串并按 Enter 以试用程序，然后按 Enter 退出。



其他资源

- 使用 .NET CLI 开发库
- .NET Standard 版本及其支持的平台。

后续步骤

在本教程中，你创建了一个类库。在下一教程中，你将了解如何对类库进行单元测试。

使用 Visual Studio 对 .NET 类库进行单元测试

或者，你可以跳过自动单元测试，并了解如何通过创建 NuGet 包来共享库：

使用 Visual Studio 创建和发布包

同时，还可以了解如何发布控制台应用。如果从本教程中创建的解决方案发布控制台应用，类库将以 .dll 文件的形式随附。

使用 Visual Studio 发布 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio 通过 .NET 测试 .NET 类库

2021/11/16 •

本教程演示如何通过将测试项目添加到解决方案来自动执行单元测试。

先决条件

- 本教程适用于在使用 Visual Studio 创建 .NET 类库中创建的解决方案。

创建单元测试项目

单元测试在开发和发布期间提供自动化的软件测试。MSTest 是可供选择的三个测试框架之一。其他两个是 xUnit 和 nUnit。

1. 启动 Visual Studio。
2. 打开在使用 Visual Studio 创建 .NET 类库中创建的 `ClassLibraryProjects` 解决方案。
3. 将名为“StringLibraryTest”的新单元测试项目添加到解决方案。
   a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加” > “新建项目”。
   b. 在“添加新项目”页面，在搜索框中输入“mstest”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。
   c. 选择“MSTest 测试项目”模板，然后选择“下一步” 。
   d. 在“配置新项目”页面，在“项目名称”框中输入“StringLibraryTest”。然后选择“下一步” 。
   e. 在“其他信息”页的“框架”框中选择“.NET 6 (长期支持)” 。然后选择“创建”。
4. 此时，Visual Studio 会创建项目，并在具有以下代码的代码窗口中打开类文件。如果未显示想要使用的语言，请更改页面顶部的语言选择器。

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Sub TestSub()

        End Sub
    End Class
End Namespace
```

单元测试模板创建的源代码负责执行以下操作：

- 它会导入 Microsoft.VisualStudio.TestTools.UnitTesting 命名空间，其中包含用于单元测试的类型。
- 向 `UnitTest1` 类应用 TestClassAttribute 特性。
- 它应用 TestMethodAttribute 特性来定义 C# 中的 `TestMethod1` 或 Visual Basic 中的 `TestSub`。

使用 [TestClass] 标记的测试类中标记有 [TestMethod] 的所有测试方法都会在单元测试运行时自动执行。

添加项目引用

对于要使用 `StringLibrary` 类的测试库，请在 StringLibraryTest 项目中添加对 `StringLibrary` 项目的引用。

1. 在“解决方案资源管理器”中，右键单击“StringLibraryTest”项目的“依赖项”节点，并从上下文菜单中选择“添加项目引用”。
2. 在“引用管理器”对话框中，展开“项目”节点，并选择“StringLibrary”旁边的框。添加对 `StringLibrary` 程序集的引用后，编译器可以在编译 StringLibraryTest 项目时查找 StringLibrary 方法。
3. 选择“确定”。

添加并运行单元测试方法

运行单元测试时，Visual Studio 执行使用 TestClassAttribute 特性标记的类中标记有 TestMethodAttribute 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 Assert 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

[illegible]	[illegible]
`Assert.AreEqual`	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
`Assert.AreSame`	验证两个对象变量引用的是否是同一个对象。如果这些变量引用不同的对象，则断言失败。
`Assert.IsFalse`	验证条件是否为 `false`。如果条件为 `true`，则断言失败。
`Assert.IsNotNull`	验证对象是否不为 `null`。如果对象为 `null`，则断言失败。

还可以在测试方法中使用 Assert.ThrowsException 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 Assert.IsTrue 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况

下，此方法应返回 `false`，以便可以调用 Assert.IsFalse 方法。

由于库方法处理的是字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`)（不含字符且 Length 为 0 的有效字符串）和 `null` 字符串（尚未初始化的字符串）。可以直接将 `StartsWithUpper` 作为静态方法进行调用，并向其传递一个 String 自变量。或者，可以对分配给 `null` 的 `string` 变量将 `StartsWithUpper` 作为扩展方法进行调用。

将定义三个方法，每个方法都会对字符串数组中的各个元素调用它的 Assert 方法。你将调用方法重载，以便指定在测试失败时要显示的错误消息。消息标识导致失败的字符串。

创建测试方法：

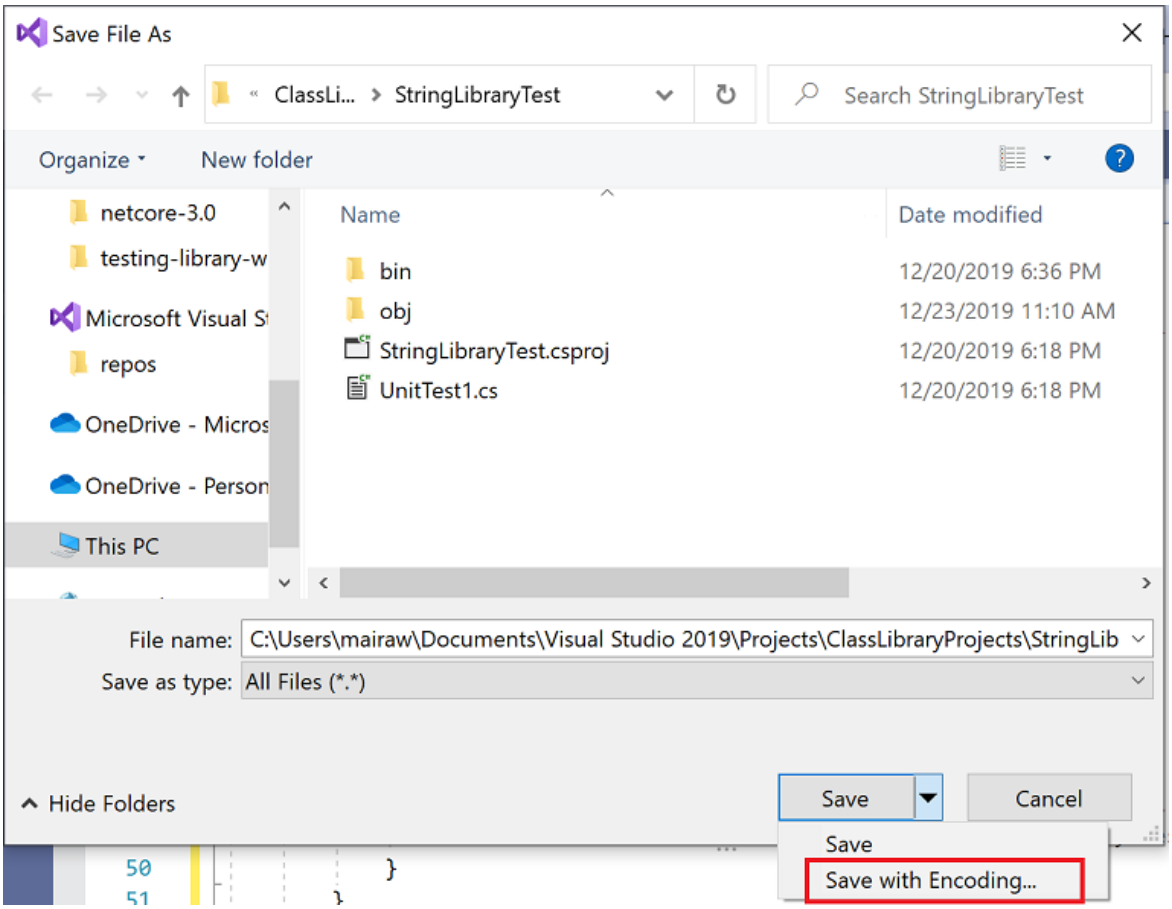
1. 将 UnitTest1.cs 或 UnitTest1.vb 代码窗口中的代码替换为以下代码：

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports UtilityLibraries

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Public Sub TestStartsWithUpper()
            ' Tests that we expect to return true.
            Dim words() As String = {"Alphabet", "Zebra", "ABC", "Αθήνα", "Москва"}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsTrue(result,
                       $"Expected for '{word}': true; Actual: {result}")
            Next
        End Sub

        <TestMethod>
        Public Sub TestDoesNotStartWithUpper()
            ' Tests that we expect to return false.
            Dim words() As String = {"alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία",
"государство",
                                     "1234", ".", ";", " "}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsFalse(result,
                       $"Expected for '{word}': false; Actual: {result}")
            Next
        End Sub

        <TestMethod>
        Public Sub DirectCallWithNullOrEmpty()
            ' Tests that we expect to return false.
            Dim words() As String = {String.Empty, Nothing}
            For Each word In words
                Dim result As Boolean = StringLibrary.StartsWithUpper(word)
                Assert.IsFalse(result,
                       $"Expected for '{If(word Is Nothing, "<null>", word)}': false; Actual:
{result}")
            Next
        End Sub
    End Class
End Namespace
```

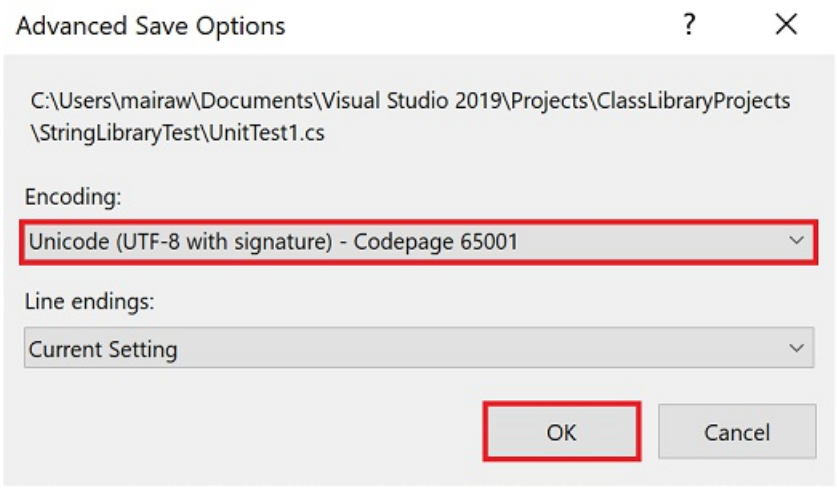
`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

2. 在菜单栏上，选择“文件” > “将 UnitTest1.cs 另存为”或“文件” > “将 UnitTest1.vb 另存为”。在“文件另存为”对话框中，选择“保存”按钮旁边的箭头，然后选择“保存时使用编码”。



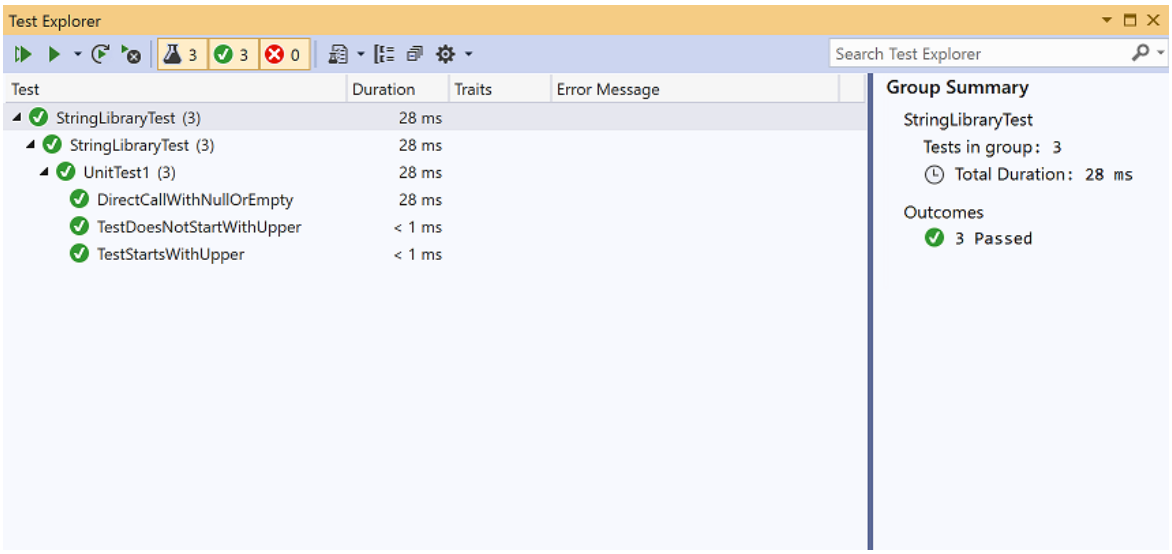
3. 在“确认另存为”对话框中，选择“是”按钮，保存文件。

4. 在“高级保存选项”对话框的“编码”下拉列表中，选择“Unicode (UTF-8 带签名) - 代码页 65001”，然后选择“确定”。



如果无法将源代码保存为 UTF8 编码文件，Visual Studio 可能会将其另存为 ASCII 文件。在这种情况下，运行时将无法准确解码 ASCII 范围以外的 UTF8 字符，且测试结果也会不正确。

5. 在菜单栏上，选择“测试” > “运行所有测试” 。如果“测试资源管理器”窗口未打开，请选择“测试” > “测试资源管理器”来将其打开 。“通过的测试”部分列出了三个测试，“摘要”部分报告了测试运行结果。



处理测试失败

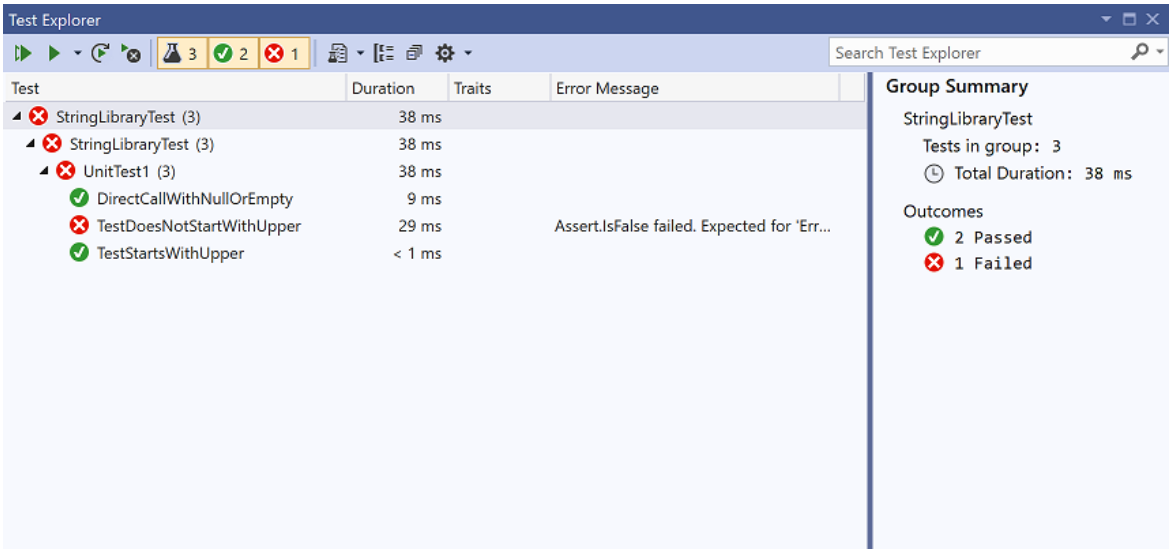
如果进行的是测试驱动开发 (TDD)，请先编写测试，然后测试会在第一次运行时失败。接着将可以使测试成功的代码添加到应用。在本教程中，先编写了测试要验证的应用代码然后才创建测试，所以没有看到测试失败。若要验证测试是否在预期失败时失败，请在测试输入中添加无效值。

1. 通过修改 `TestDoesNotStartWithUpper` 方法中的 `words` 数组来包含字符串“Error”。由于 Visual Studio 将在生成运行测试的解决方案时自动保存打开的文件，因此无需手动保存。

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

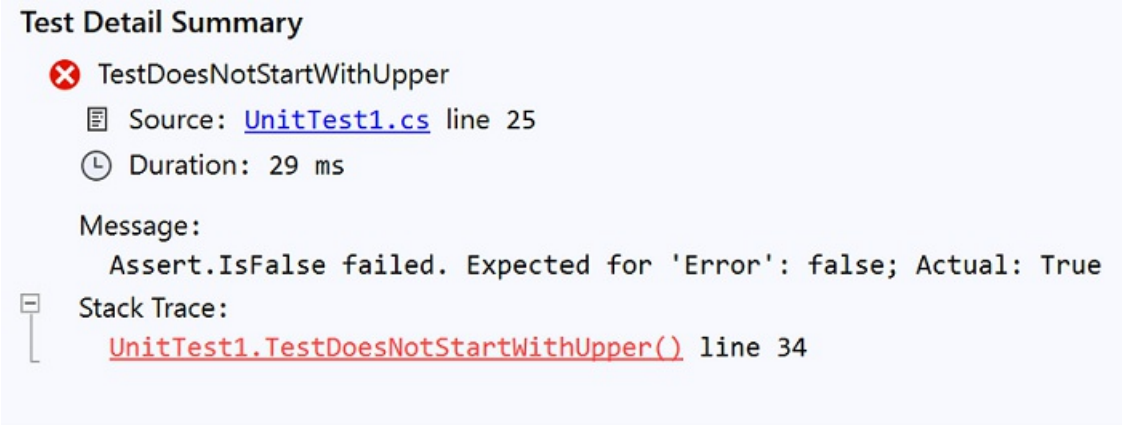
```
Dim words() As String = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " }
```

2. 从菜单栏中选择“测试” > “运行所有测试”，运行测试 。“测试资源管理器”窗口指示有两个测试成功，还有一个失败。



3. 选择失败的测试，`TestDoesNotStartWith`。

“测试资源管理器”窗口显示断言生成的消息：“Assert.IsFalse 失败。“Error”应返回 false；实际返回 True”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。



4. 删除在步骤 1 中添加的字符串"Error"。重新运行测试，测试将通过。

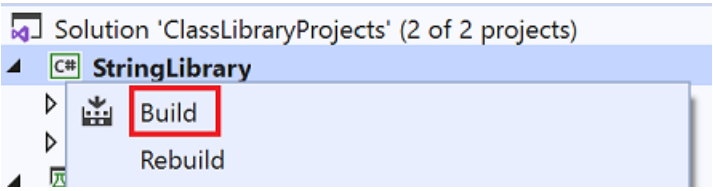
测试库的发行版本

至此，在运行库的调试版本时，测试已全部通过，接下来应对库的发行版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

若要测试发行版本，请执行以下操作：

1. 在 Visual Studio 工具栏中，将生成配置从"调试"更改为"发行"。

2. 在"解决方案资源管理器"中，右键单击"StringLibrary"项目，从上下文菜单中选择"生成"，重新编译库。



3. 从菜单栏中选择"测试运行" > "所有测试"，运行单元测试。测试通过。

调试测试

如果使用 Visual Studio 作为 IDE，则可以使用教程：使用 Visual Studio 调试 .NET 控制台应用程序中所示的相同过程，使用单元测试项目来调试代码。右键单击"StringLibraryTests"项目，然后从上下文菜单中选择"调试测试"，而不是启动 ShowCase 应用项目。

Visual Studio 启动附有调试器的测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

其他资源

- 单元测试基础知识 - Visual Studio
- .NET 中的单元测试

后续步骤

在本教程中，你对类库进行了单元测试。你可以将库作为包发布到 NuGet，使其可供其他人使用。若要了解如何操作，请遵循 NuGet 教程：

使用 Visual Studio 创建和发布 NuGet 包

如果将库作为 NuGet 包发布，其他人可以安装并使用它。若要了解如何操作，请遵循 NuGet 教程：

在 Visual Studio 中安装和使用包

库并非必须作为包进行分发。它还可与使用它的控制台应用捆绑在一起。若要了解如何发布控制台应用，请参阅本系列中前面的教程：

使用 Visual Studio 发布 .NET 控制台应用程序

本教程演示如何通过将测试项目添加到解决方案来自动执行单元测试。

先决条件

- 本教程适用于在使用 Visual Studio 创建 .NET 类库中创建的解决方案。

创建单元测试项目

单元测试在开发和发布期间提供自动化的软件测试。MSTest 是可供选择的三个测试框架之一。其他两个是 xUnit 和 nUnit。

1. 启动 Visual Studio。
2. 打开在使用 Visual Studio 创建 .NET 类库中创建的 `ClassLibraryProjects` 解决方案。
3. 将名为“StringLibraryTest”的新单元测试项目添加到解决方案。
   a. 在“解决方案资源管理器”中右键单击解决方案并选择“添加” > “新建项目”。
   b. 在“添加新项目”页面，在搜索框中输入“mstest”。从“语言”列表中选择“C#”或“Visual Basic”，然后从“平台”列表中选择“所有平台”。
   c. 选择“MSTest 测试项目”模板，然后选择“下一步” 。
   d. 在“配置新项目”页面，在“项目名称”框中输入“StringLibraryTest”。然后选择“下一步” 。
   e. 在“其他信息”页上，选择“目标框架”框中的“.NET 5.0 (当前)” 。然后选择“创建”。
4. 此时，Visual Studio 会创建项目，并在具有以下代码的代码窗口中打开类文件。如果未显示想要使用的语言，请更改页面顶部的语言选择器。

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Sub TestSub()

        End Sub
    End Class
End Namespace
```

单元测试模板创建的源代码负责执行以下操作：

- 它会导入 Microsoft.VisualStudio.TestTools.UnitTesting 命名空间，其中包含用于单元测试的类型。
- 向 `UnitTest1` 类应用 TestClassAttribute 特性。
- 它应用 TestMethodAttribute 特性来定义 C# 中的 `TestMethod1` 或 Visual Basic 中的 `TestSub`。

使用 [TestClass] 标记的测试类中标记有 [TestMethod] 的所有测试方法都会在单元测试运行时自动执行。

添加项目引用

对于要使用 `StringLibrary` 类的测试库，请在 StringLibraryTest 项目中添加对 `StringLibrary` 项目的引用。

1. 在“解决方案资源管理器”中，右键单击“StringLibraryTest”项目的“依赖项”节点，并从上下文菜单中选择“添加项目引用”。
2. 在“引用管理器”对话框中，展开“项目”节点，并选择“StringLibrary”旁边的框。添加对 `StringLibrary` 程序集的引用后，编译器可以在编译 StringLibraryTest 项目时查找 StringLibrary 方法。
3. 选择“确定”。

添加并运行单元测试方法

运行单元测试时，Visual Studio 执行使用 TestClassAttribute 特性标记的类中标记有 TestMethodAttribute 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 Assert 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

[illegible]	[illegible]
`Assert.AreEqual`	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
`Assert.AreSame`	验证两个对象变量引用的是否是同一个对象。如果这些变量引用不同的对象，则断言失败。
`Assert.IsFalse`	验证条件是否为 `false`。如果条件为 `true`，则断言失败。
`Assert.IsNotNull`	验证对象是否不为 `null`。如果对象为 `null`，则断言失败。

还可以在测试方法中使用 Assert.ThrowsException 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 Assert.IsTrue 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况

下，此方法应返回 `false`，以便可以调用 Assert.IsFalse 方法。

由于库方法处理的是字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`)（不含字符且 Length 为 0 的有效字符串）和 `null` 字符串（尚未初始化的字符串）。可以直接将 `StartsWithUpper` 作为静态方法进行调用，并向其传递一个 String 自变量。或者，可以对分配给 `null` 的 `string` 变量将 `StartsWithUpper` 作为扩展方法进行调用。

将定义三个方法，每个方法都会对字符串数组中的各个元素调用它的 Assert 方法。你将调用方法重载，以便指定在测试失败时要显示的错误消息。消息标识导致失败的字符串。

创建测试方法：

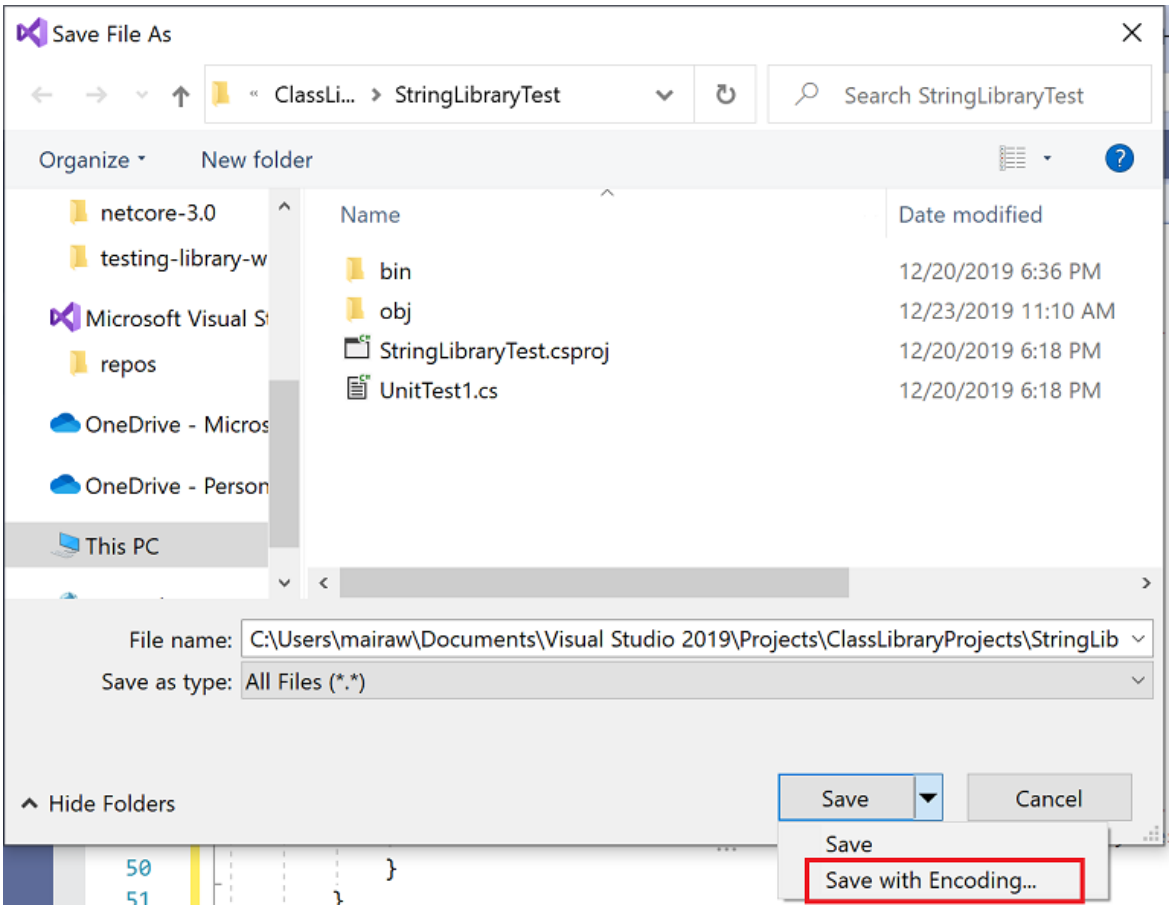
1. 将 UnitTest1.cs 或 UnitTest1.vb 代码窗口中的代码替换为以下代码：

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting
Imports UtilityLibraries

Namespace StringLibraryTest
    <TestClass>
    Public Class UnitTest1
        <TestMethod>
        Public Sub TestStartsWithUpper()
            ' Tests that we expect to return true.
            Dim words() As String = {"Alphabet", "Zebra", "ABC", "Αθήνα", "Москва"}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsTrue(result,
                       $"Expected for '{word}': true; Actual: {result}")
            Next
        End Sub

        <TestMethod>
        Public Sub TestDoesNotStartWithUpper()
            ' Tests that we expect to return false.
            Dim words() As String = {"alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία",
"государство",
                                     "1234", ".", ";", " "}
            For Each word In words
                Dim result As Boolean = word.StartsWithUpper()
                Assert.IsFalse(result,
                       $"Expected for '{word}': false; Actual: {result}")
            Next
        End Sub

        <TestMethod>
        Public Sub DirectCallWithNullOrEmpty()
            ' Tests that we expect to return false.
            Dim words() As String = {String.Empty, Nothing}
            For Each word In words
                Dim result As Boolean = StringLibrary.StartsWithUpper(word)
                Assert.IsFalse(result,
                       $"Expected for '{If(word Is Nothing, "<null>", word)}': false; Actual:
{result}")
            Next
        End Sub
    End Class
End Namespace
```

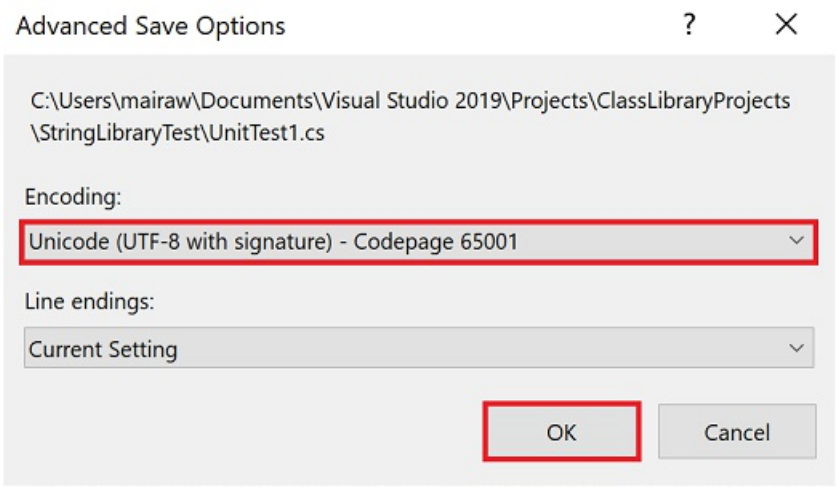
`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

2. 在菜单栏上，选择“文件” > “将 UnitTest1.cs 另存为”或“文件” > “将 UnitTest1.vb 另存为”。在“文件另存为”对话框中，选择“保存”按钮旁边的箭头，然后选择“保存时使用编码”。



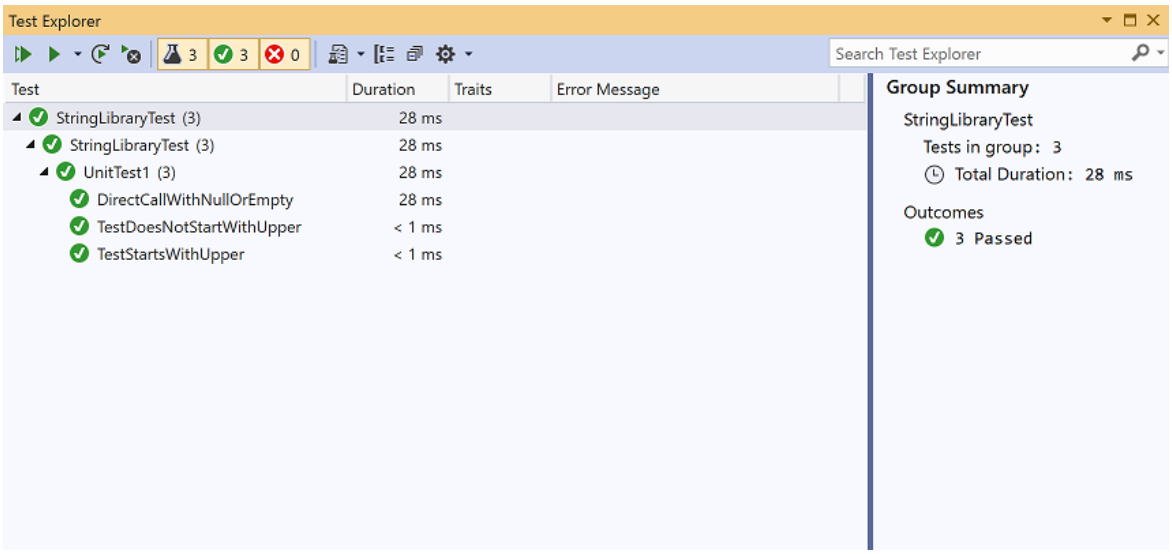
3. 在“确认另存为”对话框中，选择“是”按钮，保存文件。

4. 在“高级保存选项”对话框的“编码”下拉列表中，选择“Unicode (UTF-8 带签名) - 代码页 65001”，然后选择“确定”。



如果无法将源代码保存为 UTF8 编码文件，Visual Studio 可能会将其另存为 ASCII 文件。在这种情况下，运行时将无法准确解码 ASCII 范围以外的 UTF8 字符，且测试结果也会不正确。

5. 在菜单栏上，选择“测试” > “运行所有测试” 。如果“测试资源管理器”窗口未打开，请选择“测试” > “测试资源管理器”来将其打开 。“通过的测试”部分列出了三个测试，“摘要”部分报告了测试运行结果。



处理测试失败

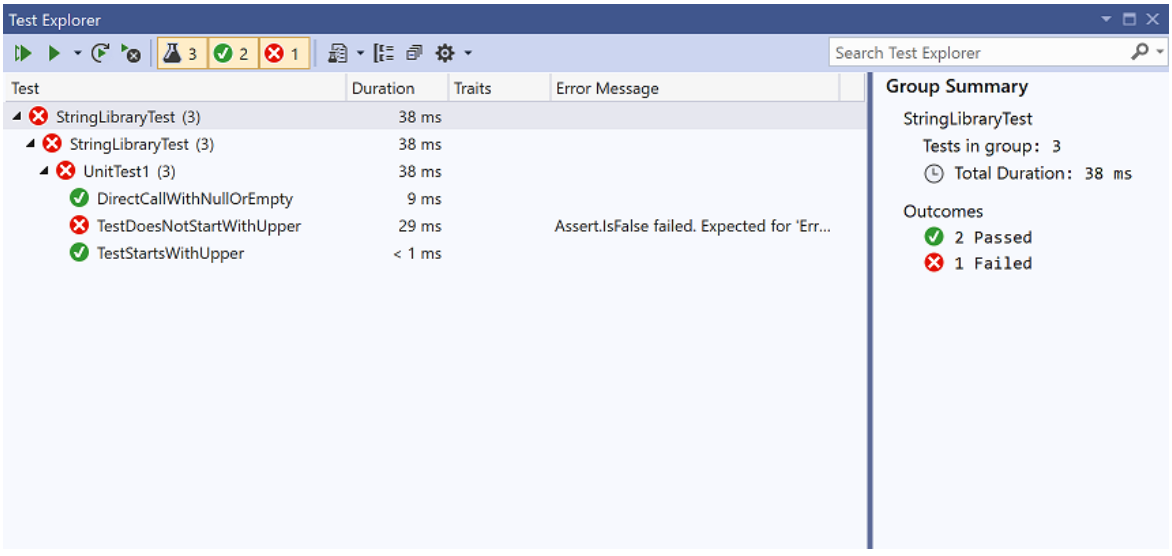
如果进行的是测试驱动开发 (TDD)，请先编写测试，然后测试会在第一次运行时失败。接着将可以使测试成功的代码添加到应用。在本教程中，先编写了测试要验证的应用代码然后才创建测试，所以没有看到测试失败。若要验证测试是否在预期失败时失败，请在测试输入中添加无效值。

1. 通过修改 `TestDoesNotStartWithUpper` 方法中的 `words` 数组来包含字符串“Error”。由于 Visual Studio 将在生成运行测试的解决方案时自动保存打开的文件，因此无需手动保存。

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

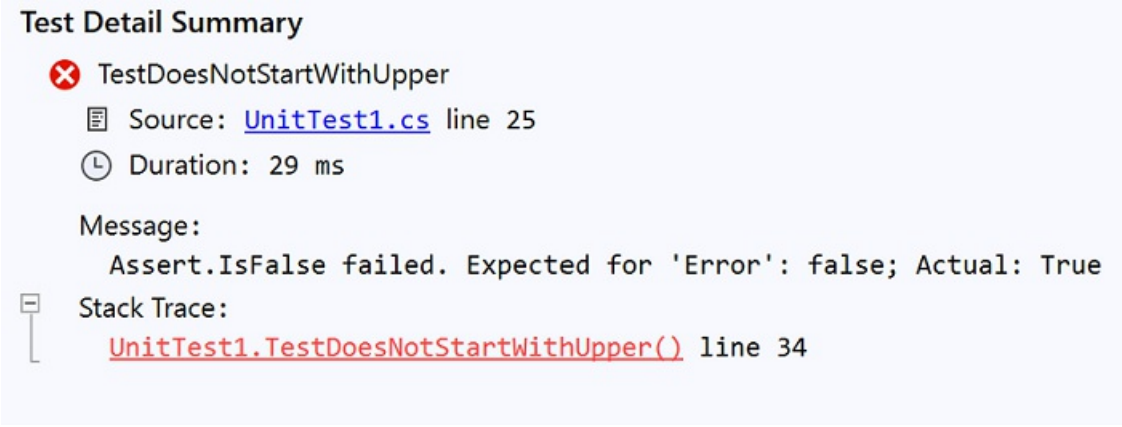
```
Dim words() As String = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " }
```

2. 从菜单栏中选择“测试” > “运行所有测试”，运行测试 。“测试资源管理器”窗口指示有两个测试成功，还有一个失败。



3. 选择失败的测试，`TestDoesNotStartWith`。

“测试资源管理器”窗口显示断言生成的消息：“Assert.IsFalse 失败。“Error”应返回 false；实际返回 True”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。



4. 删除在步骤 1 中添加的字符串"Error"。重新运行测试，测试将通过。

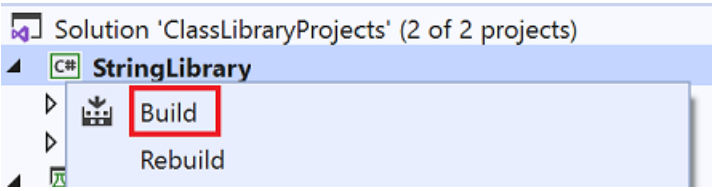
测试库的发行版本

至此，在运行库的调试版本时，测试已全部通过，接下来应对库的发行版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

若要测试发行版本，请执行以下操作：

1. 在 Visual Studio 工具栏中，将生成配置从"调试"更改为"发行"。

2. 在"解决方案资源管理器"中，右键单击"StringLibrary"项目，从上下文菜单中选择"生成"，重新编译库。



3. 从菜单栏中选择"测试运行" > "所有测试"，运行单元测试。测试通过。

调试测试

如果使用 Visual Studio 作为 IDE，则可以使用教程：使用 Visual Studio 调试 .NET 控制台应用程序中所示的相同过程，使用单元测试项目来调试代码。右键单击"StringLibraryTests"项目，然后从上下文菜单中选择"调试测试"，而不是启动 ShowCase 应用项目。

Visual Studio 启动附有调试器的测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

其他资源

- 单元测试基础知识 - Visual Studio
- .NET 中的单元测试

后续步骤

在本教程中，你对类库进行了单元测试。你可以将库作为包发布到 NuGet，使其可供其他人使用。若要了解如何操作，请遵循 NuGet 教程：

使用 Visual Studio 创建和发布 NuGet 包

如果将库作为 NuGet 包发布，其他人可以安装并使用它。若要了解如何操作，请遵循 NuGet 教程：

在 Visual Studio 中安装和使用包

库并非必须作为包进行分发。它还可与使用它的控制台应用捆绑在一起。若要了解如何发布控制台应用，请参阅本系列中前面的教程：

使用 Visual Studio 发布 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio Code 创建 .NET 控制台应用程序

2021/11/16 •

本教程演示如何使用 Visual Studio Code 和 .NET CLI 创建并运行 .NET 控制台应用程序。项目任务（例如创建、编译和运行项目）通过使用 .NET CLI 来完成。你可以遵循本教程中的步骤使用其他代码编辑器，然后在终端中运行命令（如果你愿意）。

先决条件

- 已安装 C# 扩展 的 Visual Studio Code。有关如何在 Visual Studio Code 上安装扩展的信息，请访问 VS Code 扩展市场。
- .NET 6 SDK。

创建应用

创建一个名为“HelloWorld”的 .NET 控制台应用项目。

1. 启动 Visual Studio Code。
2. 从主菜单中选择“文件” > “打开文件夹”（在 macOS 上为“文件” > “打开...”）。
3. 在“打开文件夹”对话框中，创建“HelloWorld”文件夹并将其选中。然后单击“选择文件夹”（在 macOS 上为“打开”）。

   默认情况下，文件夹名称将是项目名称和命名空间名称。稍后将在本教程中添加代码，假定项目命名空间为 `HelloWorld`。
4. 在“是否信任此文件夹中的文件作者”对话框中，选择“是，我信任此作者”。
5. 在主菜单中选择“视图” > “终端”，从 Visual Studio Code 中打开“终端”。

   “终端”在“HelloWorld”文件夹中连同命令提示符一起打开。
6. 在“终端”中输入以下命令：

   ```
   dotnet new console --framework net6.0
   ```

   该项目模板通过在 Program.cs 中调用 Console.WriteLine(String) 方法，创建了一个在控制台窗口中显示“Hello World”的简单应用程序。

   ```
   Console.WriteLine("Hello, World!");
   ```
7. 将 Program.cs 的内容替换为以下代码：

```
namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

如果 Visual Studio Code 提示添加缺少的资产，请选择“是”，以生成和调试应用。

该代码将定义类 `Program`，其中包含一个将 String 数组用作参数的方法 `Main`。 `Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。*args* 数组中包含在应用程序启动时提供的所有命令行自变量。

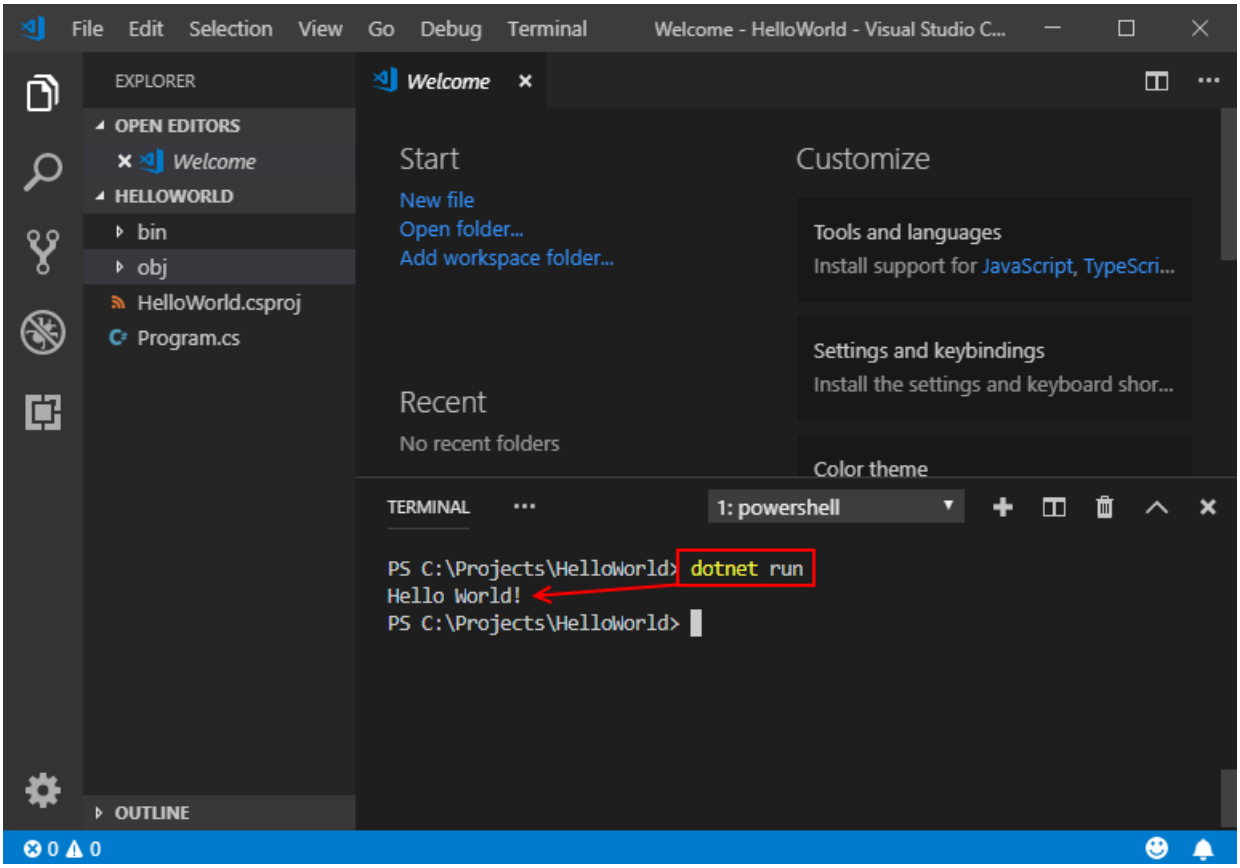
在最新版本的 C# 中，名为顶级语句的新功能允许你省略 `Program` 类和 `Main` 方法。大多数现有 C# 程序不使用顶级语句，因此本教程不使用此新功能。但它在 C# 10 中可用，是否在程序中使用它是样式首选项的问题。

运行应用

在“终端”中运行以下命令：

```
dotnet run
```

程序显示“Hello World!” 然后结束。



增强应用

改进应用程序，使其提示用户输入名字，并将其与日期和时间一同显示。

1. 打开 Program.cs。

2. 将 Program.cs 中 `Main` 方法的内容(当前只是调用 `Console.WriteLine` 的行)替换为以下代码:

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
Console.Write($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

此代码会在控制台窗口中显示一条提示，然后等待用户输入字符串并按 Enter。它会将此字符串存储到名为 `name` 的变量中。它还会检索 DateTime.Now 属性的值(其中包含当前的本地时间)，并将此值赋给 `currentDate` 变量。同时会在控制台窗口中显示这些值。最后会在控制台窗口中显示一条提示，并调用 Console.ReadKey(Boolean) 方法来等待用户输入。

NewLine 是一种独立于平台和语言的表示换行符的方式。替代方法是在 C# 中使用 `\n` 和在 Visual Basic 中使用 `vbCrLf`。

字符串前面的美元符号 (`$`) 使你可以将表达式(如变量名称)放入字符串中的大括号内。表达式值将代替表达式插入到字符串中。此语法称为内插字符串。

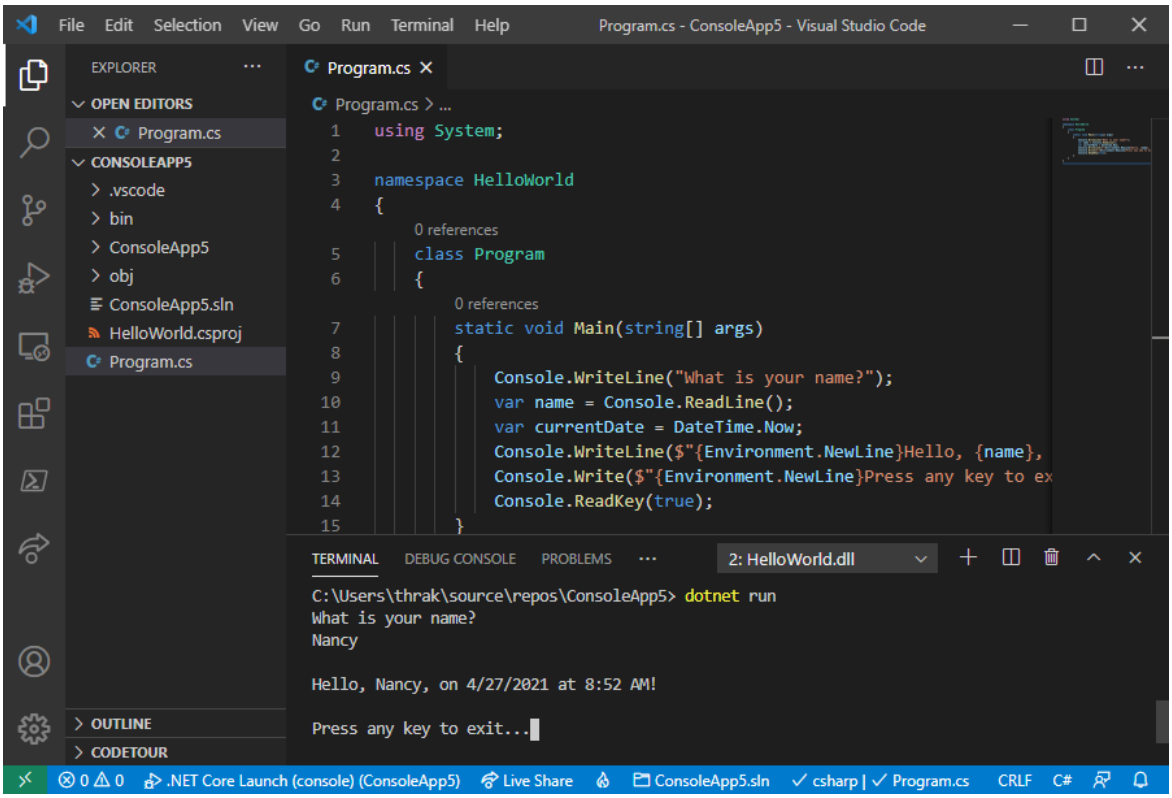
3. 保存更改。

> **IMPORTANT**
> 在 Visual Studio Code 中，必须显式保存更改。与 Visual Studio 不同，生成和运行应用时不会自动保存文件更改。

4. 再次运行程序:

```
dotnet run
```

5. 出现提示时，输入名称并按 Enter 键。



6. 按任意键退出程序。

其他资源

- 设置 Visual Studio Code

后续步骤

在本教程中，你创建了一个 .NET 控制台应用程序。在下一教程中，你将调试该应用。

使用 Visual Studio Code 调试 .NET 控制台应用程序

本教程演示如何使用 Visual Studio Code 和 .NET CLI 创建并运行 .NET 控制台应用程序。项目任务（例如创建、编译和运行项目）通过使用 .NET CLI 来完成。你可以遵循本教程中的步骤使用其他代码编辑器，然后在终端中运行命令（如果你愿意）。

先决条件

1. 已安装 C# 扩展 的 Visual Studio Code。有关如何在 Visual Studio Code 上安装扩展的信息，请访问 VS Code 扩展市场。
2. .NET 5 SDK。如果安装 .NET 6 SDK，请同时安装 .NET 5 SDK，否则某些教程说明将不适用。有关详细信息，请参阅新 C# 模板生成顶级语句。

创建应用

创建一个名为“HelloWorld”的 .NET 控制台应用项目。

1. 启动 Visual Studio Code。
2. 从主菜单中选择“文件” > “打开文件夹”（在 macOS 上为“文件” > “打开...”）。
3. 在“打开文件夹”对话框中，创建“HelloWorld”文件夹，然后单击“选择文件夹”（在 macOS 上为“打开”）。

   默认情况下，文件夹名称将是项目名称和命名空间名称。稍后将在本教程中添加代码，假定项目命名空间为 `HelloWorld`。
4. 在主菜单中选择“视图” > “终端”，从 Visual Studio Code 中打开“终端” 。

   “终端”在“HelloWorld”文件夹中连同命令提示符一起打开。
5. 在“终端”中输入以下命令：

   ```
   dotnet new console --framework net5.0
   ```

用于创建简单的“Hello World”应用程序的模板。调用 Console.WriteLine(String) 方法以在控制台窗口中显示“Hello World!”。

模板代码将定义类 `Program`，其中包含一个需要将 String 数组用作参数的方法 `Main`：

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

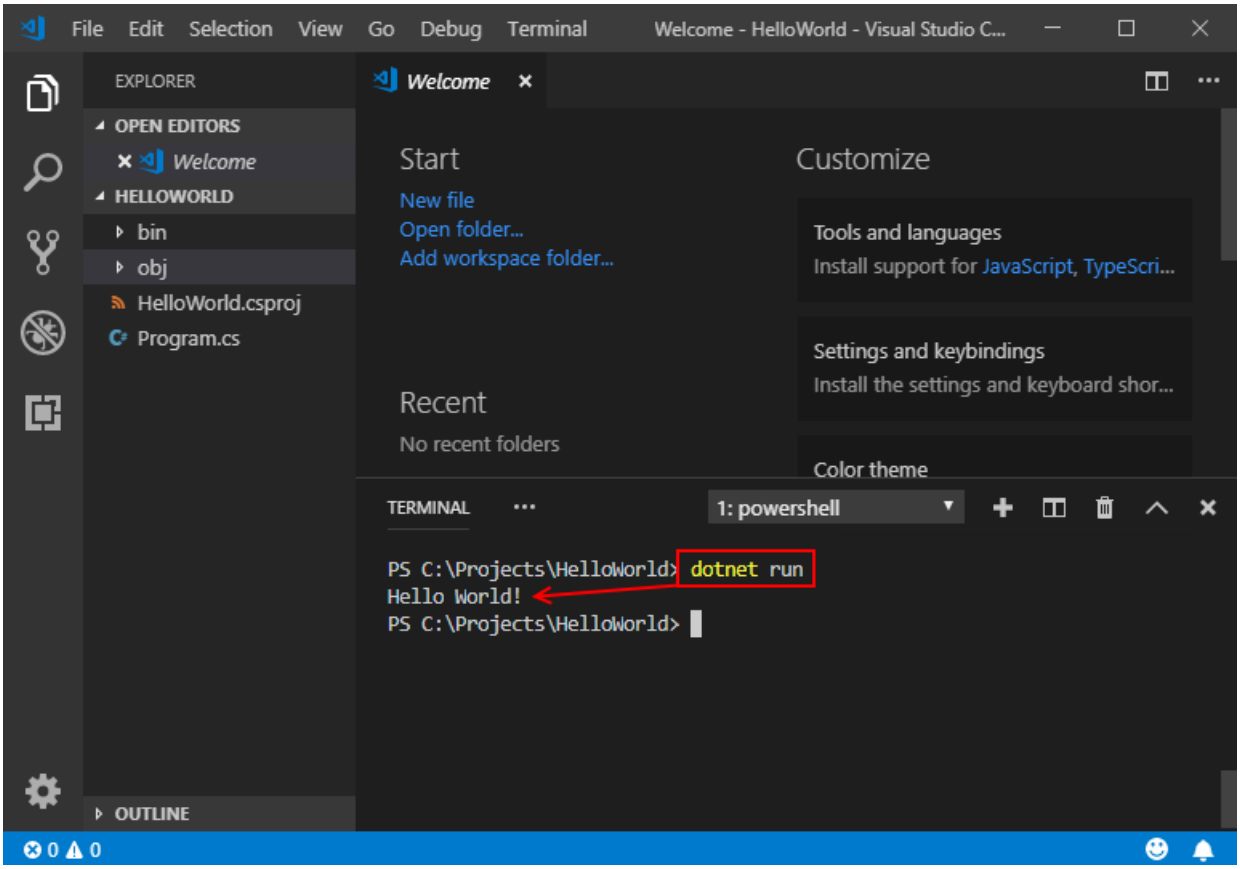
`Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。*args* 数组中包含在应用程序启动时提供的所有命令行自变量。

运行应用

在"终端"中运行以下命令：

```
dotnet run
```

程序显示"Hello World!" 然后结束。

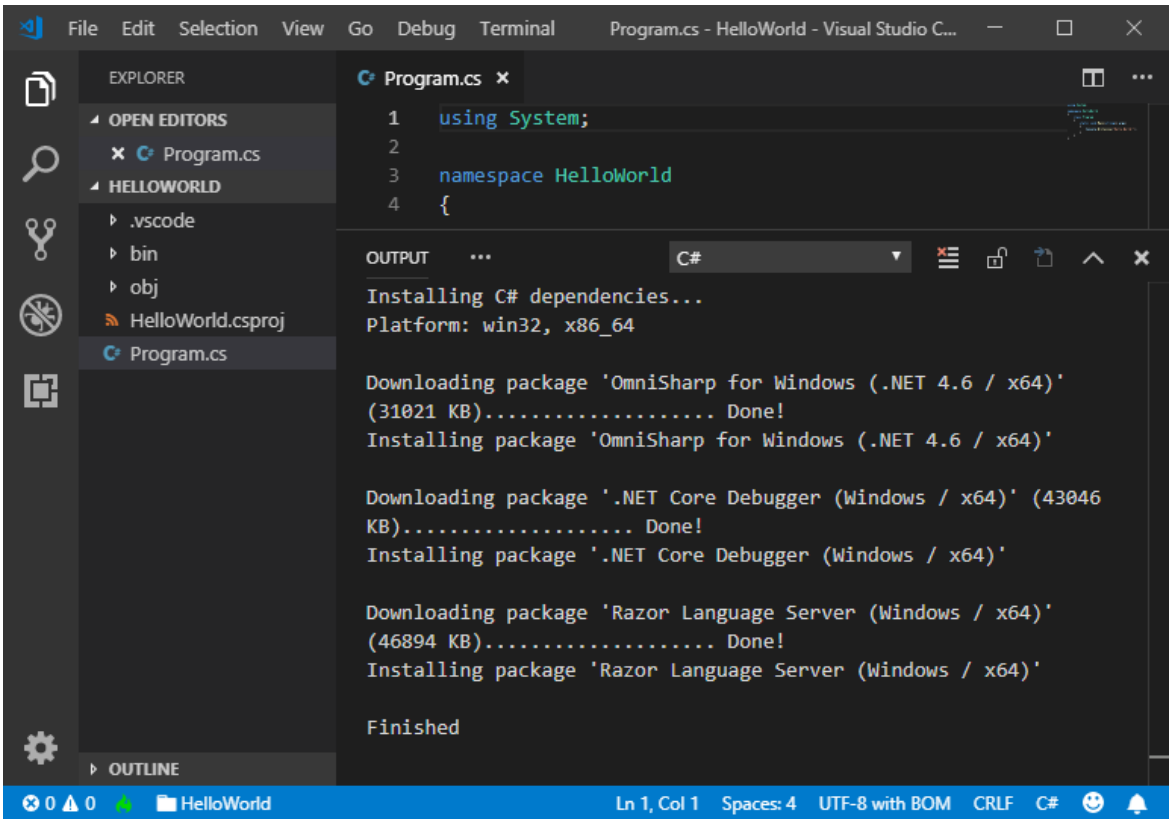


增强应用

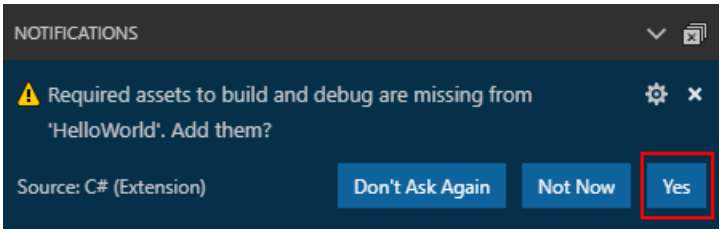
改进应用程序，使其提示用户输入名字，并将其与日期和时间一同显示。

1. 单击打开 *Program.cs*。

   在 Visual Studio Code 中首次打开 C# 文件时，会在编辑器中加载 OmniSharp。



2. Visual Studio Code 提示添加缺少的资产时选择“是”，以生成和调试应用。



3. 将 Program.cs 中 `Main` 方法的内容（当前只是调用 `Console.WriteLine` 的行）替换为以下代码：

```
Console.WriteLine("What is your name?");
var name = Console.ReadLine();
var currentDate = DateTime.Now;
Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
Console.Write($"{Environment.NewLine}Press any key to exit...");
Console.ReadKey(true);
```

此代码会在控制台窗口中显示一条提示，然后等待用户输入字符串并按 Enter。它会将此字符串存储到名为 `name` 的变量中。它还会检索 DateTime.Now 属性的值（其中包含当前的本地时间），并将此值赋给 `currentDate` 变量。同时会在控制台窗口中显示这些值。最后会在控制台窗口中显示一条提示，并调用 Console.ReadKey(Boolean) 方法来等待用户输入。

NewLine 是一种独立于平台和语言的表示换行符的方式。替代方法是在 C# 中使用 `\n` 和在 Visual Basic 中使用 `vbCrLf`。

字符串前面的美元符号 (`$`) 使你可以将表达式（如变量名称）放入字符串中的大括号内。表达式值将代替表达式插入到字符串中。此语法称为内插字符串。

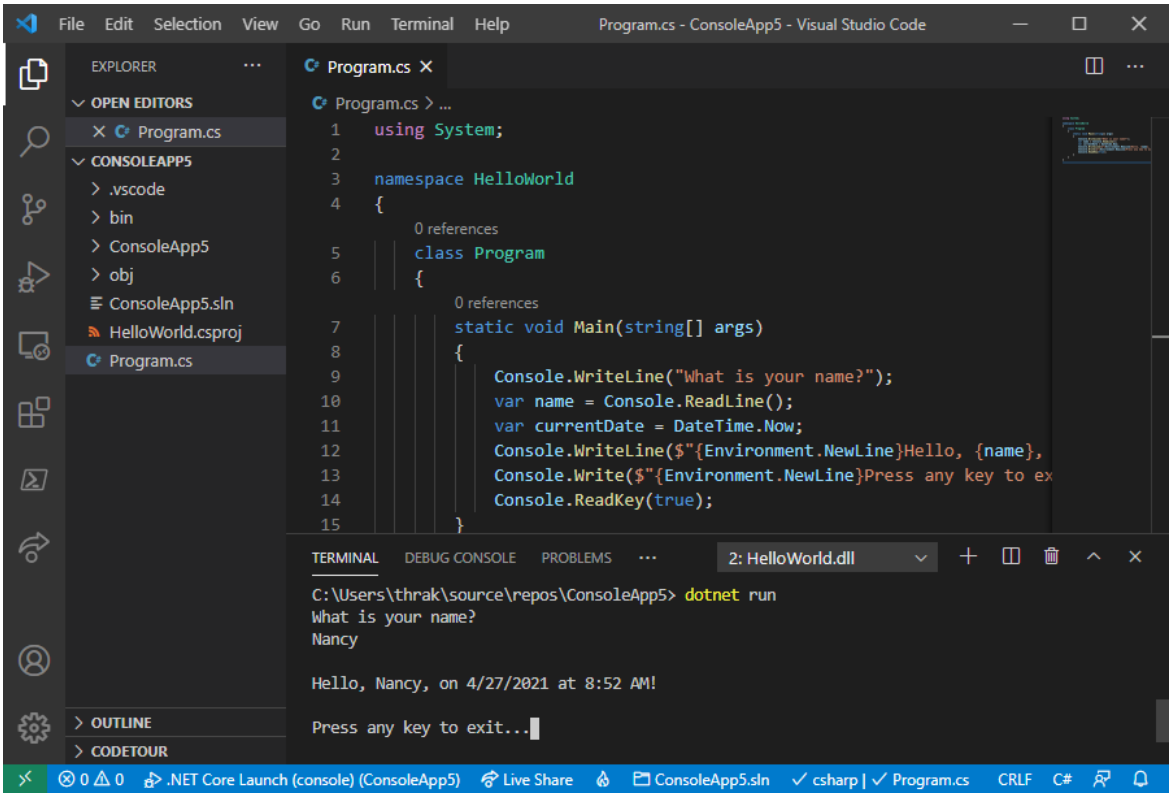
4. 保存更改。

> **IMPORTANT**
> 在 Visual Studio Code 中，必须显式保存更改。与 Visual Studio 不同，生成和运行应用时不会自动保存文件更改。

5. 再次运行程序：

```
dotnet run
```

6. 出现提示时，输入名称并按 Enter 键。



7. 按任意键退出程序。

其他资源

- 设置 Visual Studio Code

后续步骤

在本教程中，你创建了一个 .NET 控制台应用程序。在下一教程中，你将调试该应用。

使用 Visual Studio Code 调试 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio Code 调试 .NET 控制台应用程序

2021/11/16 •

本教程介绍了 Visual Studio Code 中可用于处理 .NET 应用的调试工具。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的控制台应用。

使用“调试”生成配置

“调试”和“发布”是 .NET 的内置生成配置 。可使用“调试”生成配置进行调试，使用“发布”配置进行最终版本分发。

在“调试”配置中，程序使用完整符号调试信息编译，且不进行优化。优化会使调试复杂化，因为源代码和生成的指令之间的关系更加复杂。程序的发布配置进行了完全优化，且不包含任何符号调试信息。

默认情况下，Visual Studio Code 启动设置使用“调试”生成配置，因此不需要在调试之前对其进行更改。

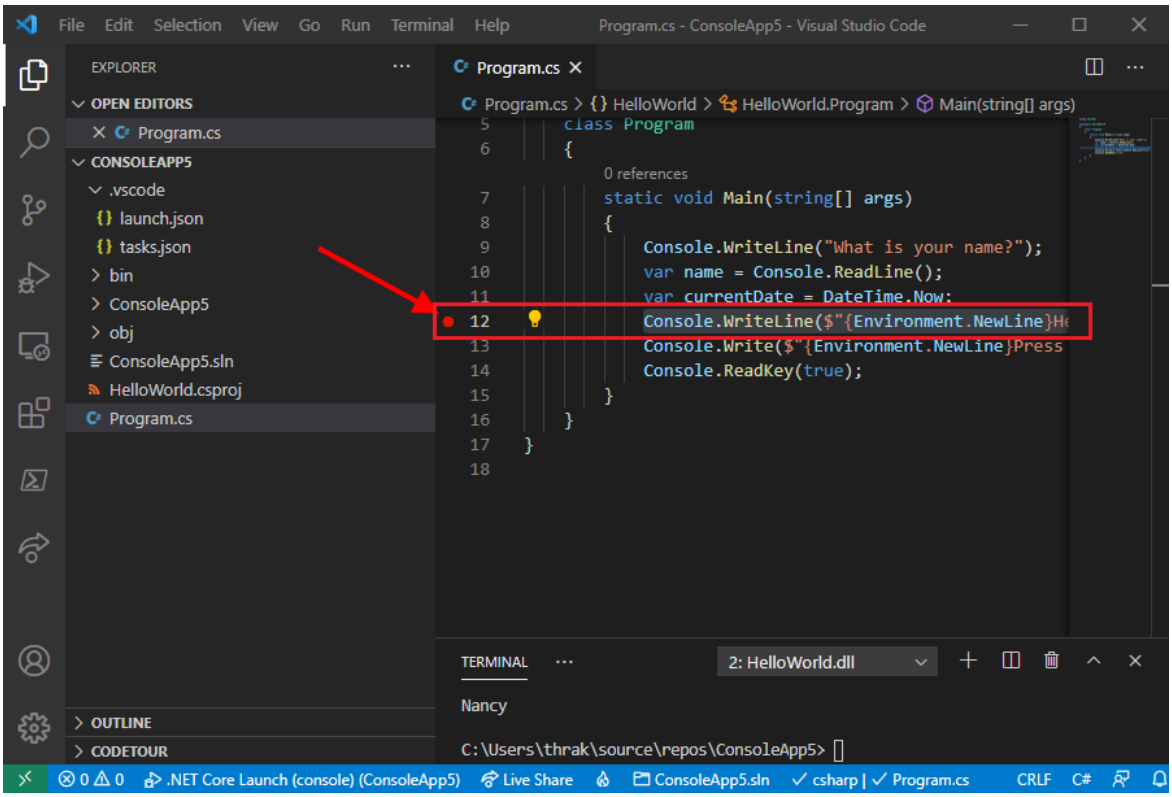
1. 启动 Visual Studio Code。
2. 打开在使用 Visual Studio Code 中创建 .NET 控制台应用程序中创建的项目的文件夹。

设置断点

断点会在运行包含断点的代码行之前暂时中断执行应用程序。

1. 打开 *Program.cs* 文件。
2. 单击代码窗口的左边缘，在显示名称、日期和时间的行上设置断点。左边缘在行号的左侧。设置断点的其他方法是在选中代码行时按 F9 或从菜单中选择“运行” > “切换断点” 。

   Visual Studio Code 通过在左边缘显示红点来指示设置了断点的行。



设置终端输入

断点位于 `Console.ReadLine` 方法调用之后。调试控制台不接受正在运行的程序的终端输入。若要在调试时处理终端输入，可以使用集成终端(Visual Studio Code 窗口之一)或外部终端。本教程中使用集成终端。

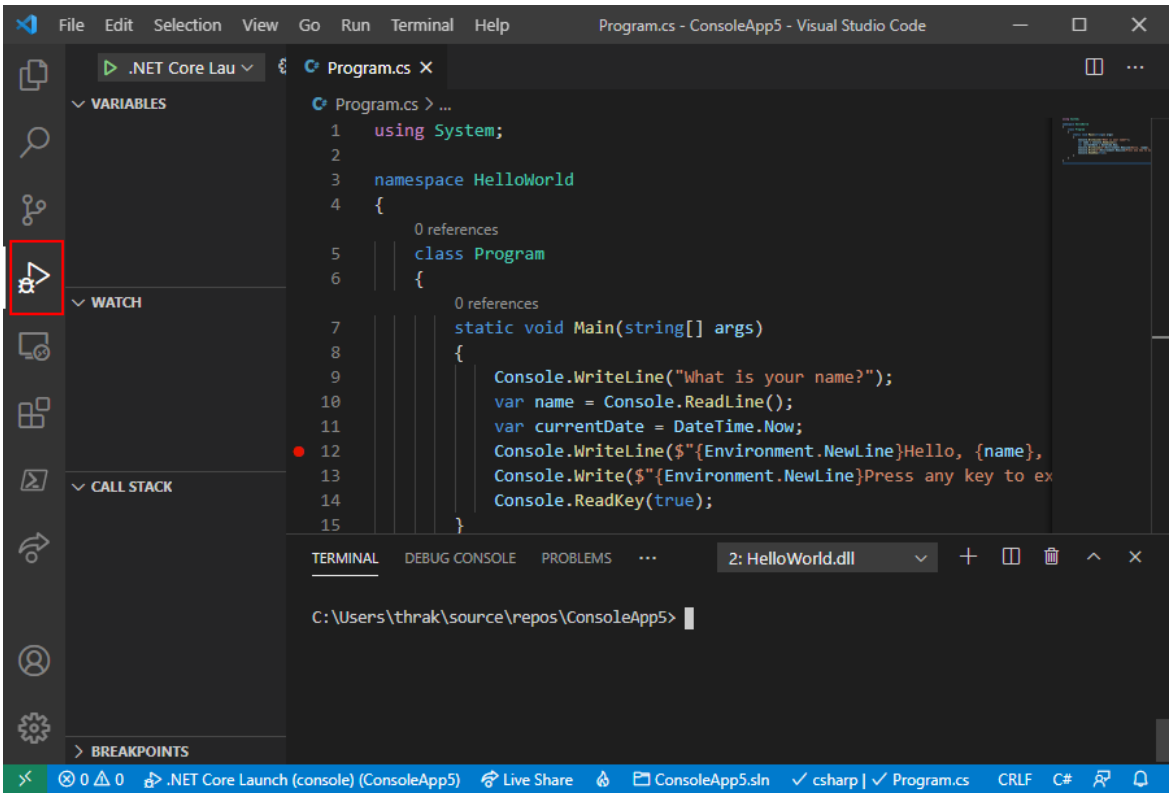
1. 打开 .vscode/launch.json。
2. 将 `console` 设置从 `internalConsole` 更改为 `integratedTerminal`：

```
"console": "integratedTerminal",
```

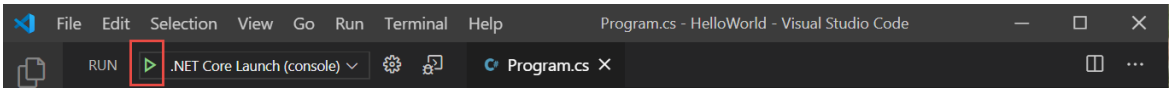
3. 保存更改。

“启动调试”

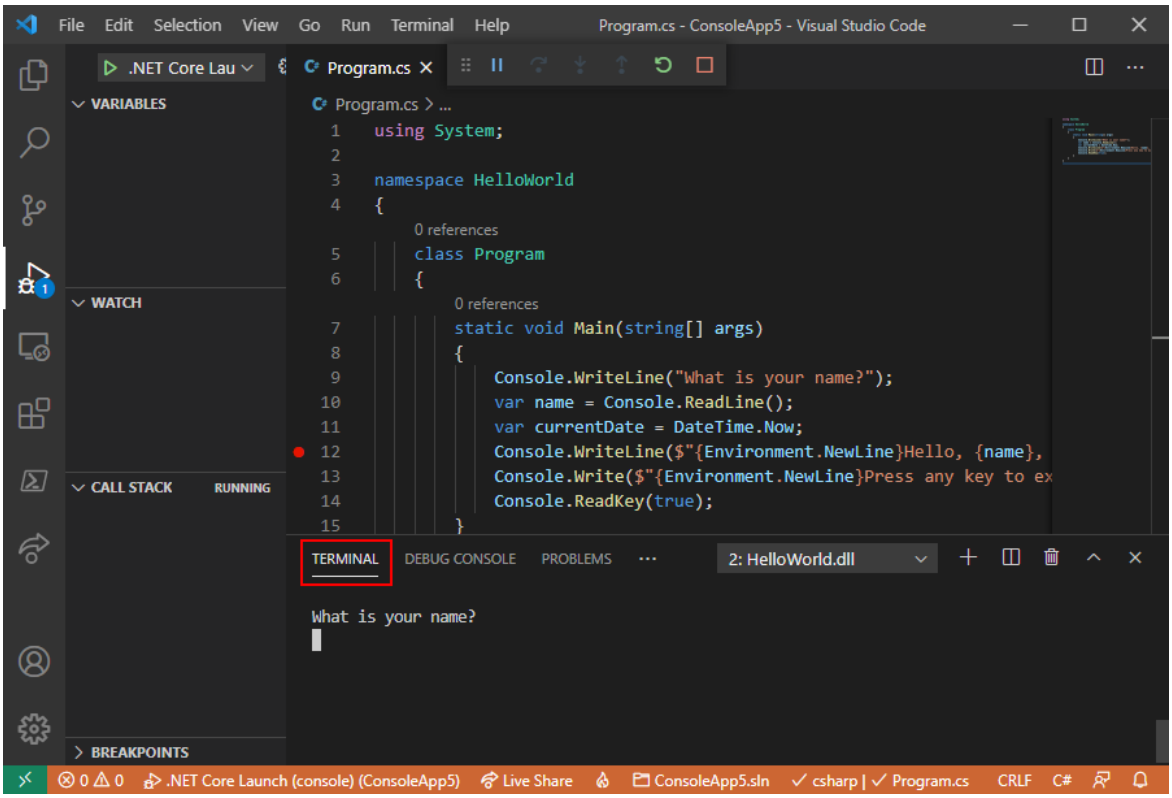
1. 选择左侧菜单上的“调试”图标，打开“调试”视图。



2. 选择窗格顶部 .NET Core Launch(控制台)旁边的绿色箭头。在调试模式下启动程序的其他方法是，按 F5 或从菜单中选择"运行" > "启动调试" 。

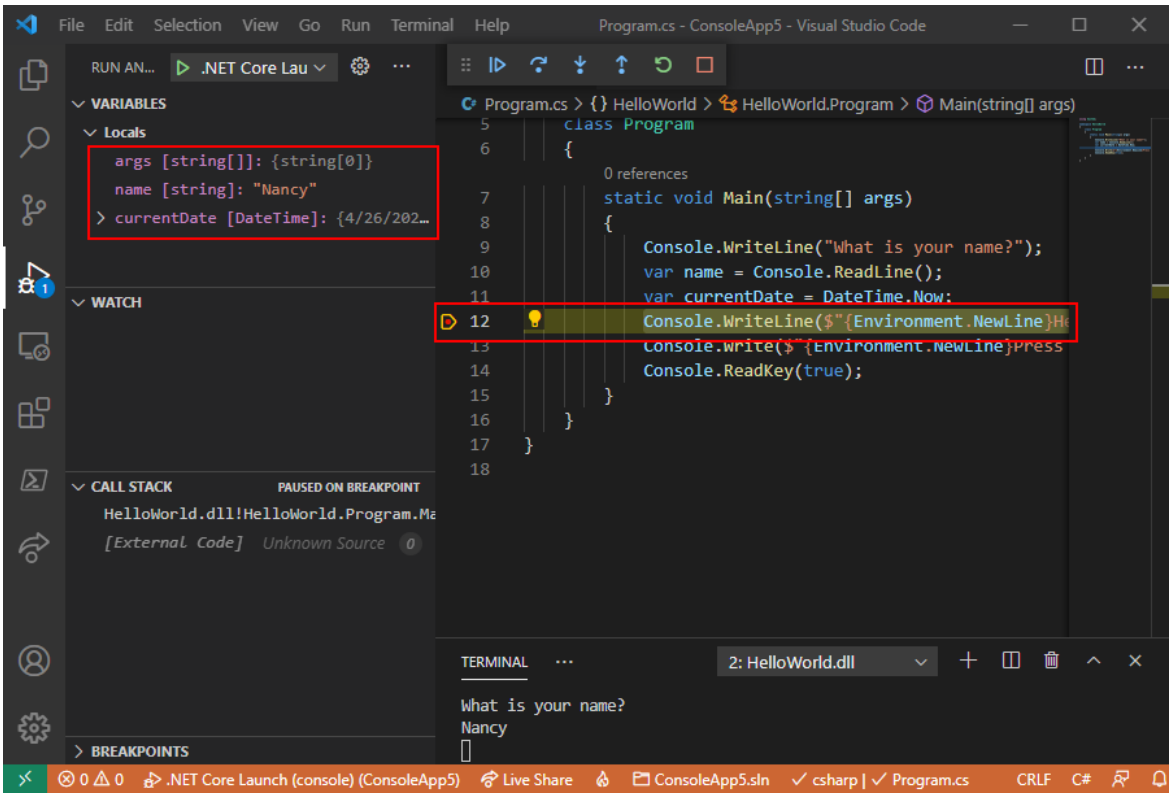


3. 选择"终端"选项卡以查看程序在等待响应之前 显示的"What is your name?"提示。



4. 在"终端"窗口中输入字符串以响应输入姓名提示，然后按 Enter。

   到达断点时，程序停止执行，然后运行 `Console.WriteLine` 方法。"变量"窗口的"局部变量"部分显示当前正在运行的方法中定义的变量值 。

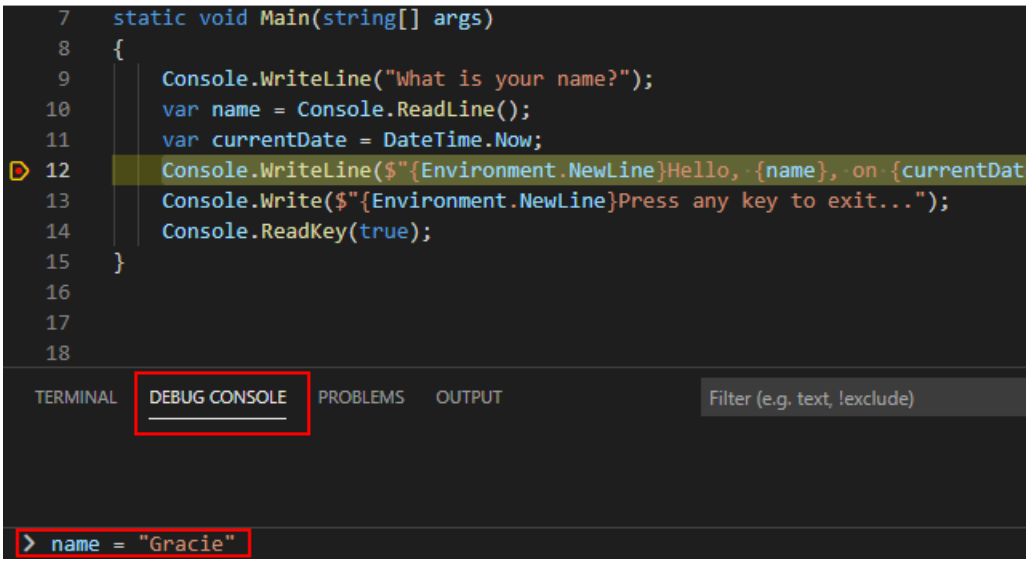


使用“调试控制台”

在“调试控制台”窗口中，可以与正在调试的应用程序进行交互。可更改变量值，看看这样会对程序产生哪些影响。

1. 选择“调试控制台”选项卡。

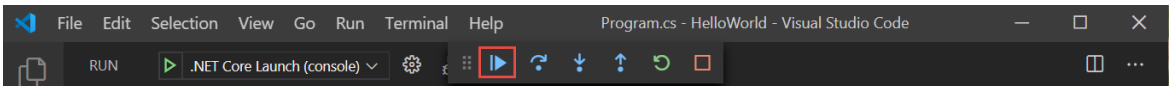
2. 在“调试控制台”窗口底部的提示符处输入 `name = "Gracie"`，然后按 Enter。



3. 在“调试控制台”窗口底部输入 `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()`，然后按 Enter。

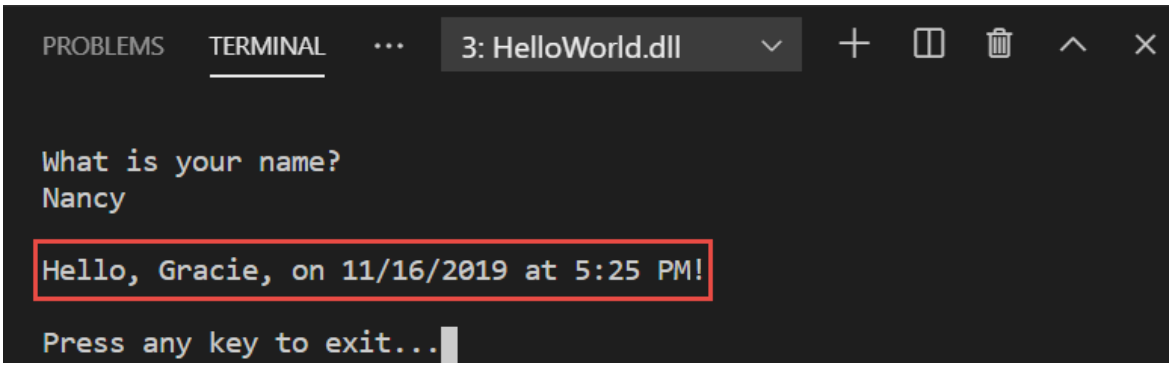
   “变量”窗口显示 `name` 和 `currentDate` 变量的新值。

4. 选择工具栏中的“继续”按钮继续执行程序。继续操作的另一种方法是按 F5。



5. 再次选择"终端"选项卡。

   控制台窗口中显示的值对应于在"调试控制台"窗口中所做的更改。

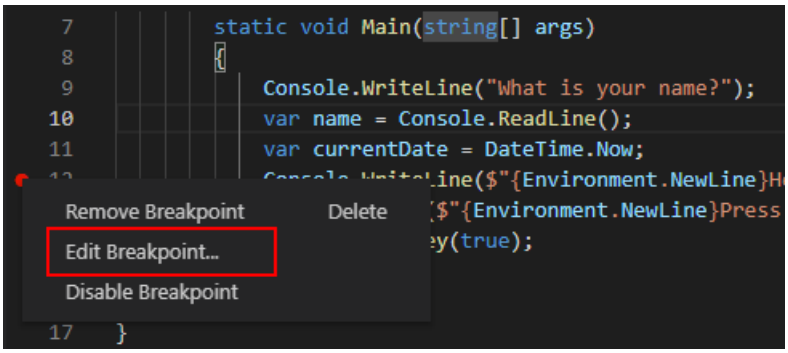


6. 按任意键，退出应用程序并停止调试。

设置条件断点

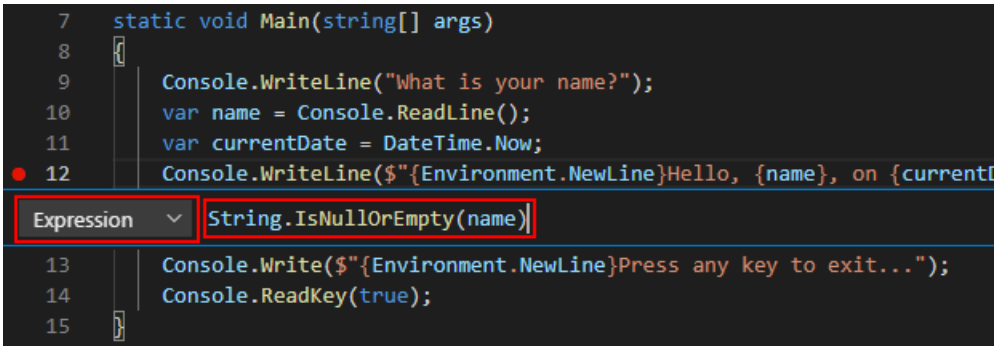
程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？ 可以使用名为"条件断点"的有用调试功能对此进行测试。

1. 右键单击(在 macOS 上按住 Ctrl 并单击)表示断点的红点。在上下文菜单中，选择"编辑断点"，打开可输入条件表达式的对话框。



2. 在下拉菜单中选择 `Expression`，输入以下条件表达式，并按 Enter。

```
String.IsNullOrEmpty(name)
```



   每次命中断点时，调试器都会调用 `String.IsNullOrEmpty(name)` 方法，仅当该方法调用返回 `true` 时，它才会在此行上中断。

   可以指定"命中次数"(而不是条件表达式)，这样程序就会在语句的运行次数达到指定值时中断执行。另一种方法是指定筛选条件，这样就可以根据诸如线程标识符、进程名称或线程名称之类的特性来中断程序执行。

3. 通过按 F5 调试来启动程序。

4. 在“终端”选项卡中，在系统提示输入姓名时按 Enter。

   由于符合指定的条件（`name` 为 `null` 或 String.Empty），因此程序会在到达断点时以及在 `Console.WriteLine` 方法运行之前停止执行。

   “变量”窗口显示 `name` 变量的值为 `""` 或 String.Empty。

5. 在“调试控制台”提示符中输入下面的语句并按 Enter，确认值为空字符串。结果为 `true`。

   ```
   name == String.Empty
   ```

6. 选择工具栏上的“继续”按钮，继续执行程序。

7. 选择“终端”选项卡，然后按任意键退出程序，停止调试。

8. 单击代码窗口左边缘上的点，清除断点。清除断点的其他方法是在选中代码行时按 F9 或从菜单中选择“运行”>“切换断点”。

9. 如果收到断点条件将丢失的警告，请选择“删除断点”。

单步执行程序

使用 Visual Studio Code，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并通过程序代码的一小部分执行程序流。由于此程序很小，因此可以单步执行整个程序。

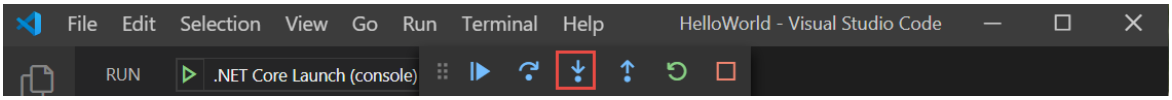
1. 在 `Main` 方法的左大括号处设置一个断点。

2. 按 F5 启动调试。

   Visual Studio Code 突出显示断点行。

   此时，“变量”窗口显示 `args` 数组为空，`name` 和 `currentDate` 具有默认值。

3. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 突出显示下一行。

4. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 运行名称提示的 `Console.WriteLine` 并突出显示下一执行行。下一行是 `name` 的 `Console.ReadLine`。“变量”窗口保持不变，“终端”选项卡显示“What is your name? ” 提示。

5. 选择“运行” > “单步执行”或按 F11。

   Visual Studio 突出显示 `name` 变量赋值。“变量”窗口显示 `name` 仍为 `null`。

6. 在“终端”选项卡中输入字符串，然后按 Enter，响应提示。

   输入字符串时，“终端”选项卡可能无法显示输入的字符串，但 Console.ReadLine 方法将捕获你的输入。

7. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 突出显示 `currentDate` 变量赋值。“变量”窗口显示 Console.ReadLine 方法调用返回的值。“终端”选项卡显示在提示符处输入的字符串。

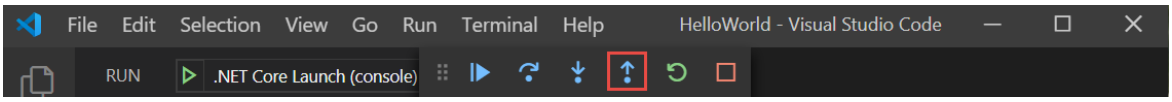
8. 选择“运行” > “单步执行”或按 F11。

   “变量”窗口显示通过 DateTime.Now 属性赋值后的 `currentDate` 变量值。

9. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 调用 Console.WriteLine(String, Object, Object) 方法。控制台窗口会显示格式化的字符串。

10. 选择“运行” > “跳出”或按 Shift+F11。



11. 选择“终端”选项卡。

    终端显示“按任意键退出...”

12. 按任意键退出程序。

使用“发布”生成配置

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，这些优化可能会影响应用程序的行为。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要构建和测试控制台应用程序的发布版本，请打开终端，并运行以下命令：

```
dotnet run --configuration Release
```

其他资源

- 在 Visual Studio Code 中进行调试

后续步骤

在本教程中，使用了 Visual Studio Code 调试工具。在下一教程中，你将发布应用的可部署版本。

使用 Visual Studio Code 发布 .NET 控制台应用程序

本教程介绍了 Visual Studio Code 中可用于处理 .NET 应用的调试工具。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的控制台应用。

使用“调试”生成配置

“调试”和“发布”是 .NET Core 的内置生成配置 。可使用“调试”生成配置进行调试，使用“发布”配置进行最终版本分发。

在“调试”配置中，程序使用完整符号调试信息编译，且不进行优化。优化会使调试复杂化，因为源代码和生成的指令之间的关系更加复杂。程序的发布配置进行了完全优化，且不包含任何符号调试信息。

默认情况下，Visual Studio Code 启动设置使用“调试”生成配置，因此不需要在调试之前对其进行更改。

1. 启动 Visual Studio Code。

2. 打开在使用 Visual Studio Code 中创建 .NET 控制台应用程序中创建的项目的文件夹。

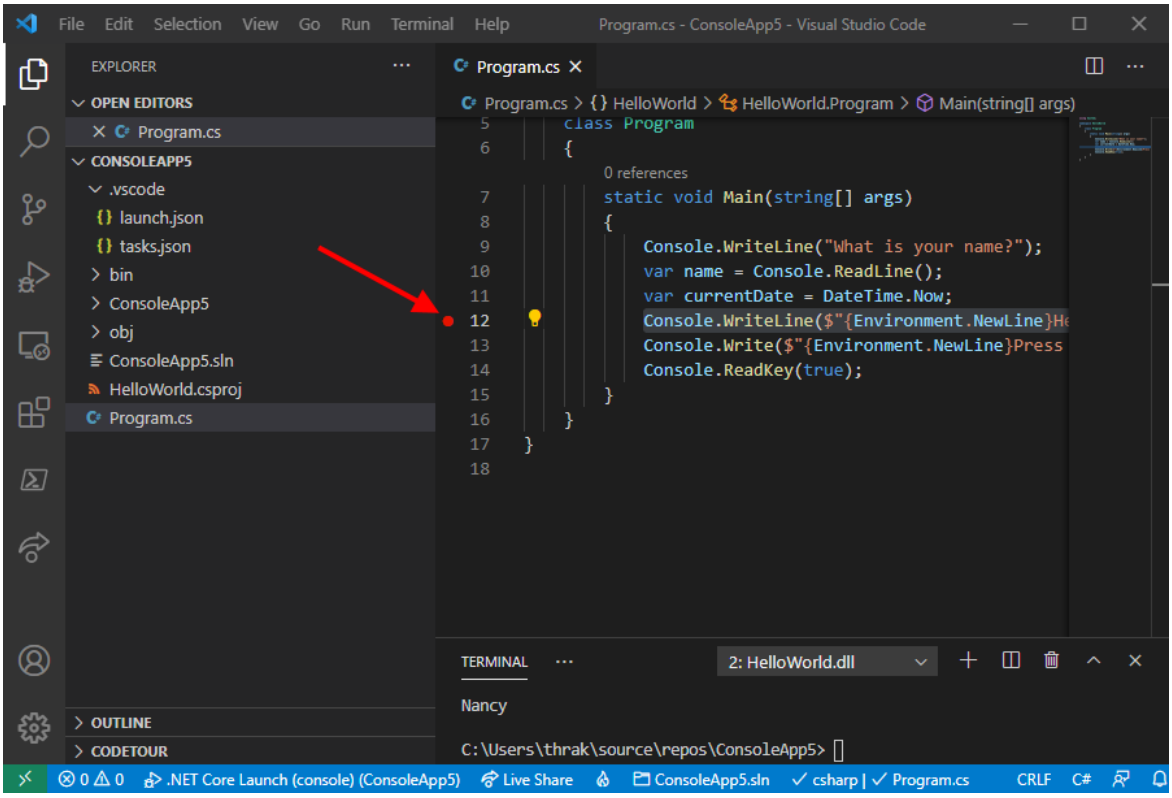
设置断点

断点会在执行包含断点的代码行之前暂时中断执行应用程序。

1. 打开 *Program.cs* 文件。

2. 单击代码窗口的左边缘，在显示名称、日期和时间的行上设置断点。左边缘在行号的左侧。设置断点的其他方法是在选中代码行时按 F9 或从菜单中选择“运行” > “切换断点”。

   Visual Studio Code 通过在左边缘显示红点来指示设置了断点的行。



设置终端输入

断点位于 `Console.ReadLine` 方法调用之后。调试控制台不接受正在运行的程序的终端输入。若要在调试时处理终端输入，可以使用集成终端(Visual Studio Code 窗口之一)或外部终端。本教程中使用集成终端。

1. 打开 .vscode/launch.json。

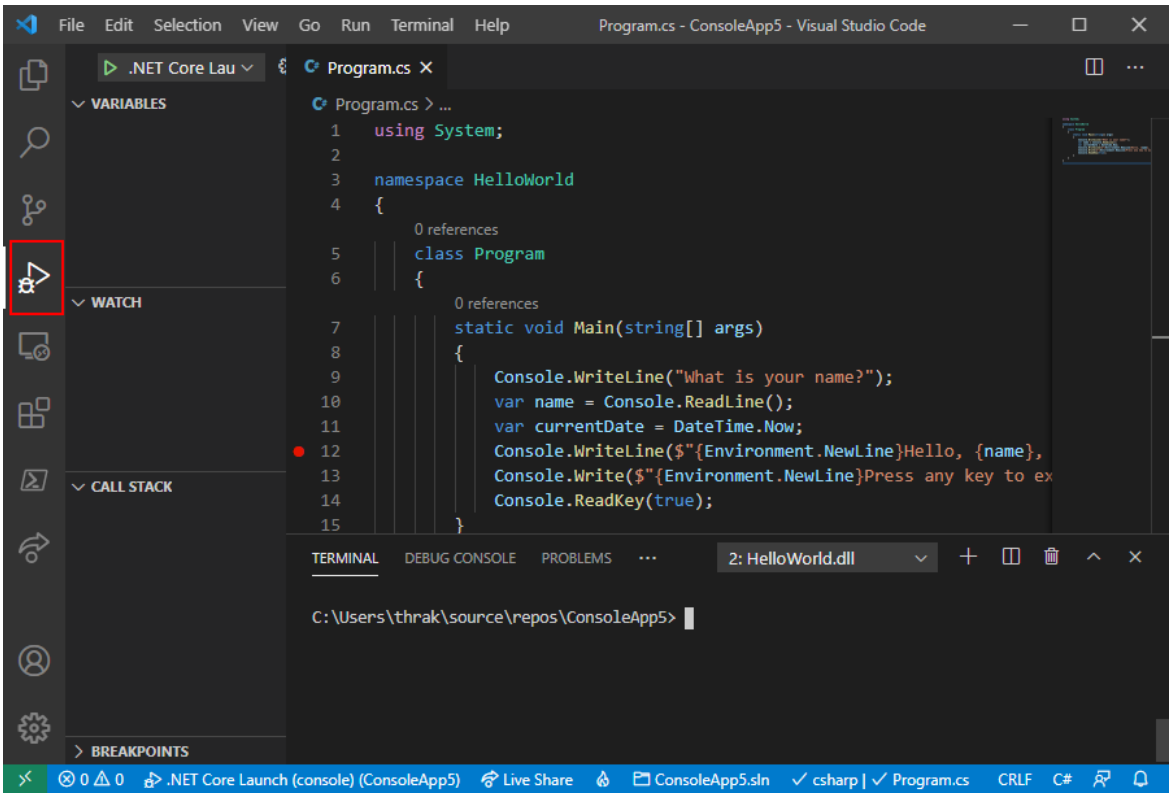
2. 将 `console` 设置从 `internalConsole` 更改为 `integratedTerminal`：

```
"console": "integratedTerminal",
```

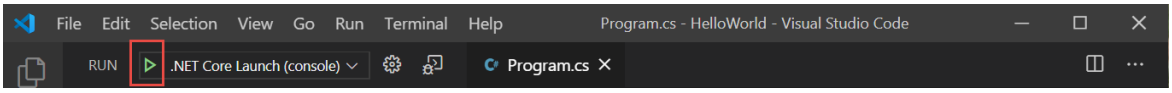
3. 保存更改。

“启动调试”

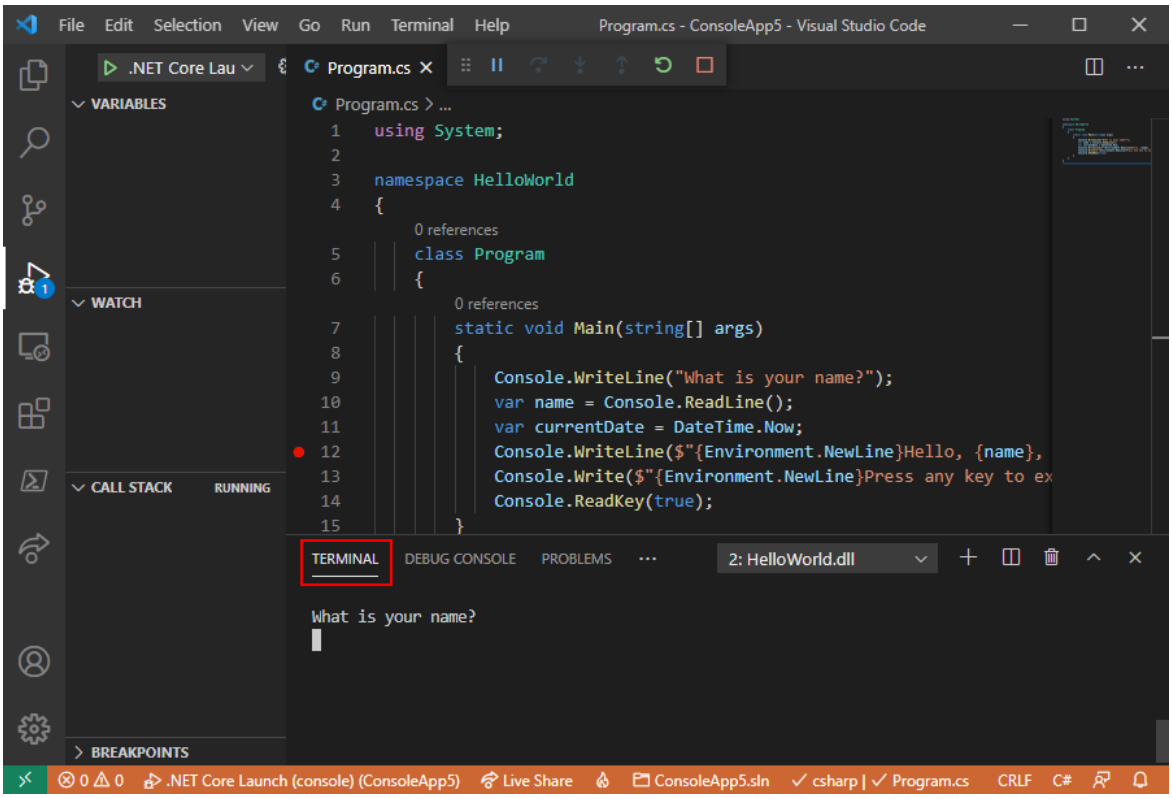
1. 选择左侧菜单上的“调试”图标，打开“调试”视图。



2. 选择窗格顶部 .NET Core Launch(控制台)旁边的绿色箭头。在调试模式下启动程序的其他方法是，按 F5 或从菜单中选择"运行" > "启动调试" 。

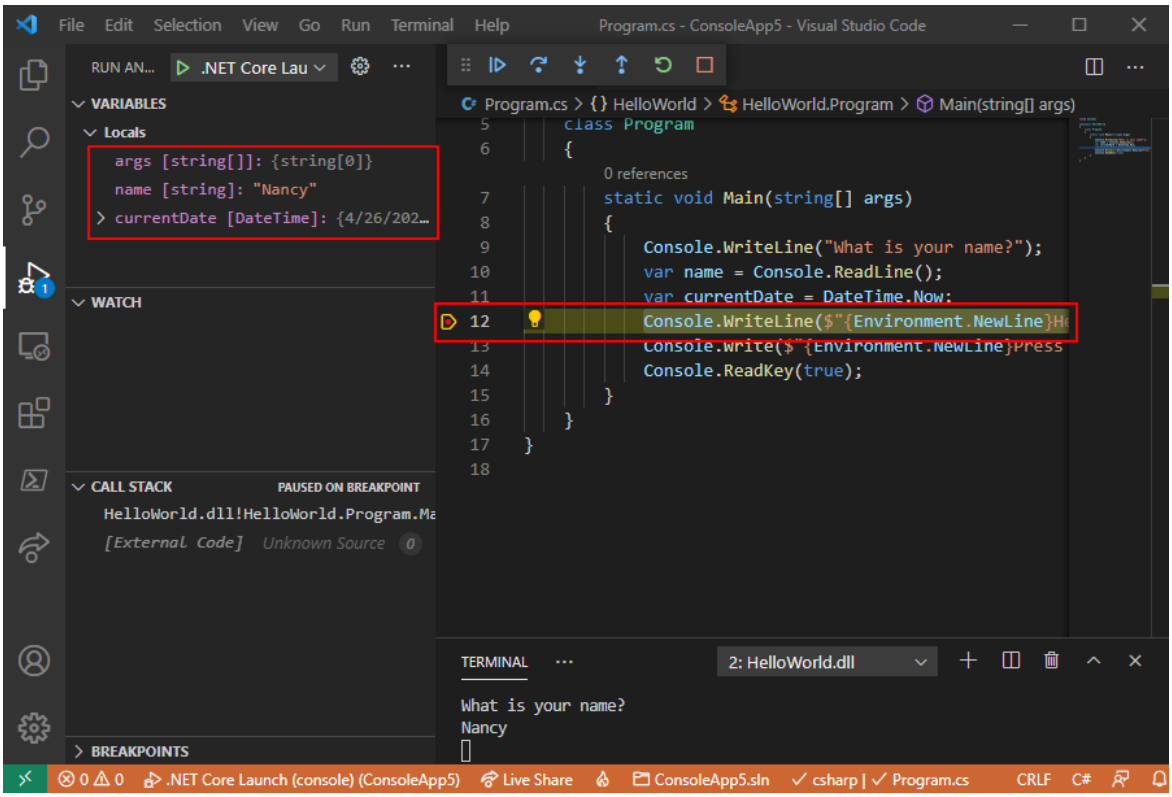


3. 选择"终端"选项卡以查看程序在等待响应之前 显示的"What is your name?"提示。



4. 在"终端"窗口中输入字符串以响应输入姓名提示，然后按 Enter。

   到达断点时，程序停止执行，然后执行 `Console.WriteLine` 方法。"变量"窗口的"局部变量"部分显示当前正在执行的方法中定义的变量值 。

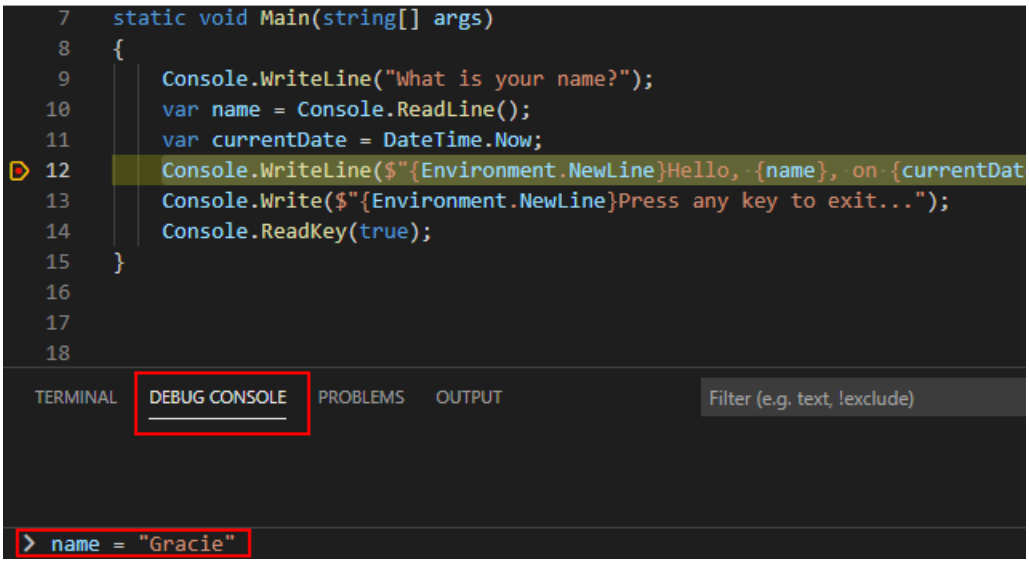


使用“调试控制台”

在“调试控制台”窗口中，可以与正在调试的应用程序进行交互。可更改变量值，看看这样会对程序产生哪些影响。

1. 选择“调试控制台”选项卡。

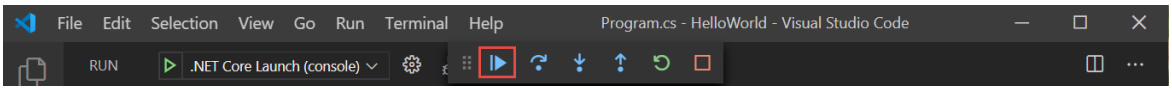
2. 在“调试控制台”窗口底部的提示符处输入 `name = "Gracie"`，然后按 Enter。



3. 在“调试控制台”窗口底部输入 `currentDate = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()`，然后按 Enter。

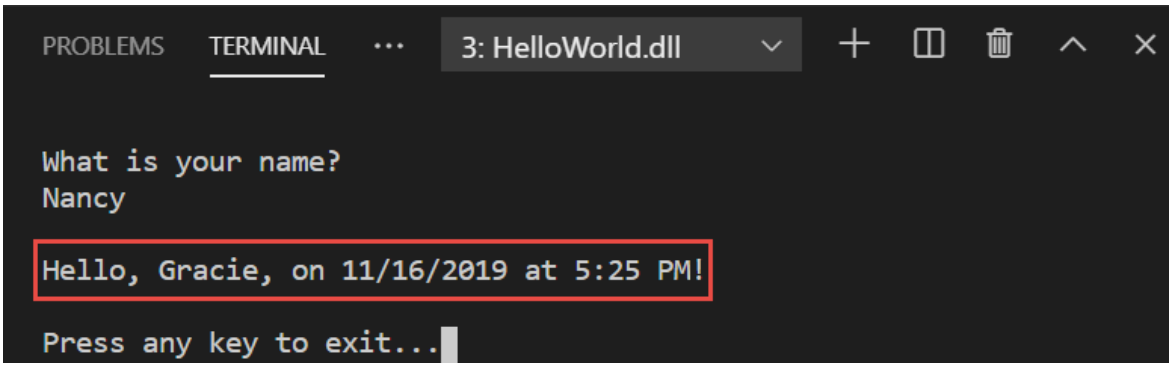
   “变量”窗口显示 `name` 和 `currentDate` 变量的新值。

4. 选择工具栏中的“继续”按钮继续执行程序。继续操作的另一种方法是按 F5。



5. 再次选择“终端”选项卡。

   控制台窗口中显示的值对应于在“调试控制台”窗口中所做的更改。

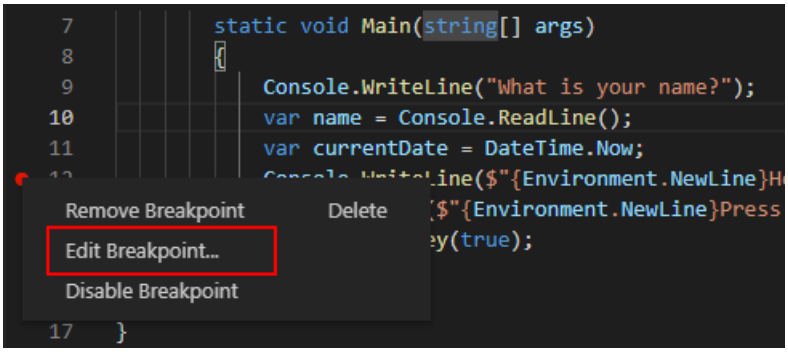


6. 按任意键，退出应用程序并停止调试。

设置条件断点

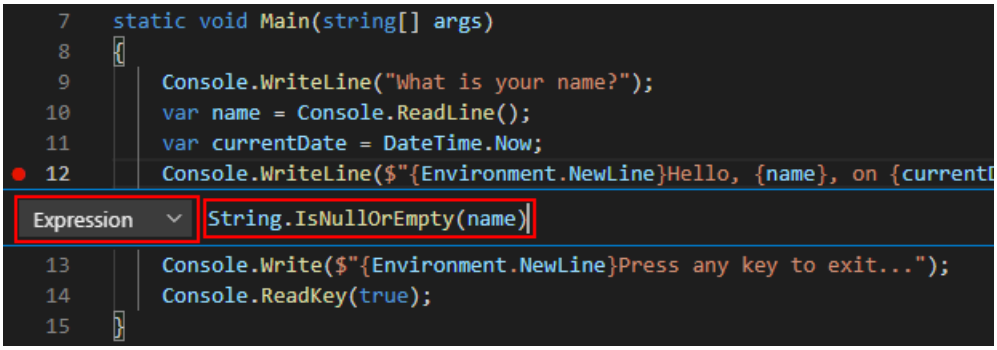
程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？ 可以使用名为“条件断点”的有用调试功能对此进行测试。

1. 右键单击(在 macOS 上按住 Ctrl 并单击)表示断点的红点。在上下文菜单中，选择“编辑断点”，打开可输入条件表达式的对话框。



2. 在下拉菜单中选择 `Expression`，输入以下条件表达式，并按 Enter。

```
String.IsNullOrEmpty(name)
```



   每次命中断点时，调试器都会调用 `String.IsNullOrEmpty(name)` 方法，仅当该方法调用返回 `true` 时，它才会在此行上中断。

   可以指定命中次数(而不是条件表达式)，这样程序就会在语句的执行次数达到指定值时中断执行。另一种方法是指定筛选条件，这样就可以根据诸如线程标识符、进程名称或线程名称之类的特性来中断程序执行。

3. 通过按 F5 调试来启动程序。

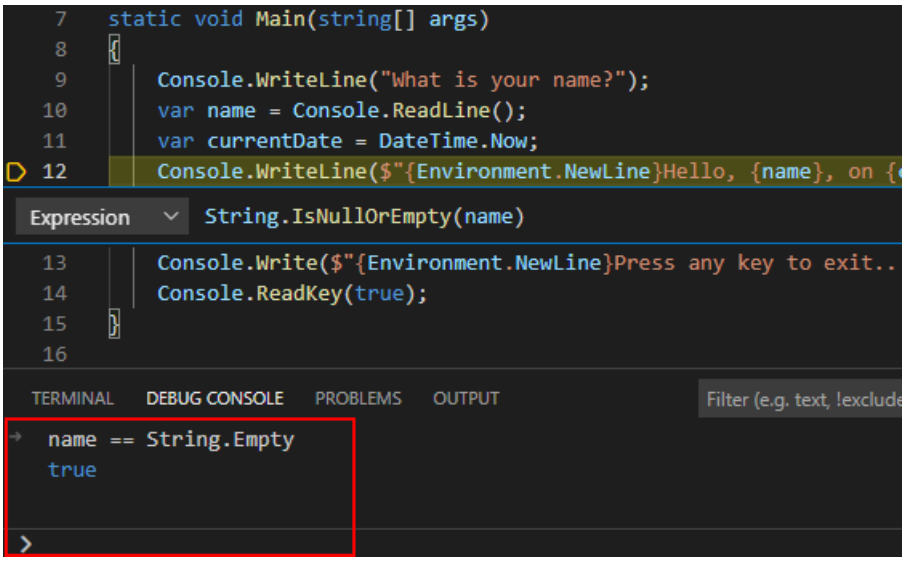
4. 在"终端"选项卡中，在系统提示输入姓名时按 Enter。

   由于符合指定的条件（`name` 为 `null` 或 String.Empty），因此程序会在到达断点时以及在 `Console.WriteLine` 方法执行之前停止执行。

   "变量"窗口显示 `name` 变量的值为 `""` 或 String.Empty。

5. 在"调试控制台"提示符中输入下面的语句并按 Enter，确认值为空字符串。结果为 `true`。

   ```
   name == String.Empty
   ```

6. 选择工具栏上的"继续"按钮，继续执行程序。

7. 选择"终端"选项卡，然后按任意键退出程序，停止调试。

8. 单击代码窗口左边缘上的点，清除断点。清除断点的其他方法是在选中代码行时按 F9 或从菜单中选择"运行">"切换断点"。

9. 如果收到断点条件将丢失的警告，请选择"删除断点"。

单步执行程序

使用 Visual Studio Code，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并通过程序代码的一小部分执行程序流。由于此程序很小，因此可以单步执行整个程序。

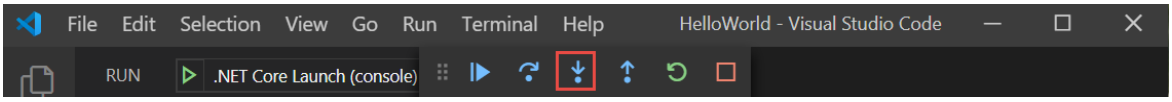
1. 在 `Main` 方法的左大括号处设置一个断点。

2. 按 F5 启动调试。

   Visual Studio Code 突出显示断点行。

   此时，"变量"窗口显示 `args` 数组为空，`name` 和 `currentDate` 具有默认值。

3. 选择"运行" > "单步执行"或按 F11。

   Visual Studio Code 突出显示下一行。

4. 选择"运行" > "单步执行"或按 F11。

   Visual Studio Code 执行名称提示的 `Console.WriteLine` 并突出显示下一执行行。下一行是 `name` 的

`Console.ReadLine`。“变量”窗口保持不变，“终端”选项卡显示“What is your name? ” 提示。

5. 选择“运行” > “单步执行”或按 F11。

   Visual Studio 突出显示 `name` 变量赋值。“变量”窗口显示 `name` 仍为 `null`。

6. 在“终端”选项卡中输入字符串，然后按 Enter，响应提示。

   输入字符串时，“终端”选项卡可能无法显示输入的字符串，但 Console.ReadLine 方法将捕获你的输入。

7. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 突出显示 `currentDate` 变量赋值。“变量”窗口显示 Console.ReadLine 方法调用返回的值。“终端”选项卡显示在提示符处输入的字符串。

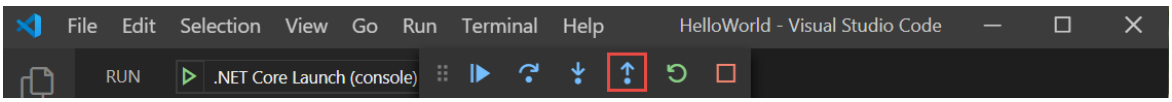
8. 选择“运行” > “单步执行”或按 F11。

   “变量”窗口显示通过 DateTime.Now 属性赋值后的 `currentDate` 变量值。

9. 选择“运行” > “单步执行”或按 F11。

   Visual Studio Code 调用 Console.WriteLine(String, Object, Object) 方法。控制台窗口会显示格式化的字符串。

10. 选择“运行” > “跳出”或按 Shift+F11。

11. 选择“终端”选项卡。

    终端显示“按任意键退出...”

12. 按任意键退出程序。

使用“发布”生成配置

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，这些优化可能会影响应用程序的行为。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要构建和测试控制台应用程序的发布版本，请打开终端，并运行以下命令：

```
dotnet run --configuration Release
```

其他资源

- 在 Visual Studio Code 中进行调试

后续步骤

在本教程中，使用了 Visual Studio Code 调试工具。在下一教程中，你将发布应用的可部署版本。

使用 Visual Studio Code 发布 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio Code 发布 .NET 控制台应用程序

2021/11/16 •

本教程演示如何发布控制台应用，以便其他用户可以运行它。发布应用程序会创建运行应用程序所需的一组文件。若要部署文件，请将文件复制到目标计算机。

.NET CLI 用于发布应用，因此可以根据需要使用 Visual Studio Code 以外的代码编辑器来学习本教程。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的控制台应用。

发布应用

1. 启动 Visual Studio Code。
2. 打开在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的 HelloWorld 项目文件夹。
3. 从主菜单中选择“视图” > “终端” 。

   终端在 HelloWorld 文件夹中打开。
4. 运行下面的命令：

   ```
   dotnet publish --configuration Release
   ```

   默认生成配置为“调试”，因此此命令指定“版本”生成配置 。版本生成配置的输出进行了完全优化，且具有最低限度的符号调试信息。

   该命令的输出类似于以下示例：

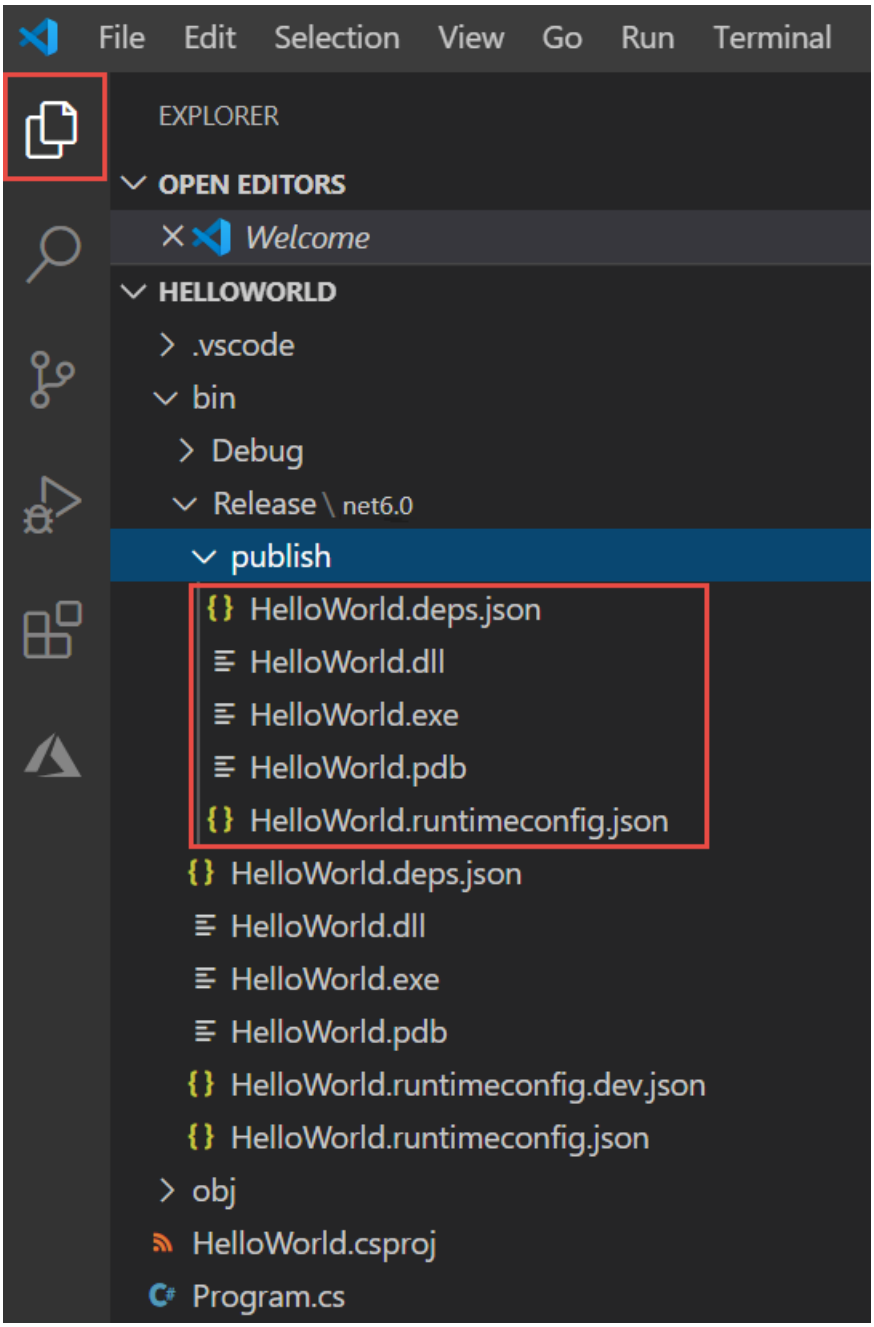
   ```
   Microsoft (R) Build Engine version 16.7.0+b89cb5fde for .NET
   Copyright (C) Microsoft Corporation. All rights reserved.
     Determining projects to restore...
     All projects are up-to-date for restore.
     HelloWorld -> C:\Projects\HelloWorld\bin\Release\net6.0\HelloWorld.dll
     HelloWorld -> C:\Projects\HelloWorld\bin\Release\net6.0\publish\
   ```

检查文件

默认情况下，发布过程中会创建依赖于框架的部署，在此类部署中，已发布的应用程序在已安装 .NET 运行时的计算机上运行。若要运行已发布的应用，可以使用可执行文件，或从命令提示符中运行 `dotnet HelloWorld.dll` 命令。

在下面的步骤中，查看由发布过程创建的文件。

1. 在左侧导航栏中选择“资源管理器”。
2. 展开 bin/Release/net6.0/publish。



如下图所示，已发布的输出包括以下文件：

- HelloWorld.deps.json

  这是应用程序的运行时依赖项文件。该文件定义了运行应用所需的 .NET 组件和库（包括包含应用程序的动态链接库）。有关详细信息，请参阅运行时配置文件。

- HelloWorld.dll

  这是应用程序的依赖于框架的部署版本。若要运行此动态链接库，请在命令提示符处输入 `dotnet HelloWorld.dll`。这种运行应用的方法适用于安装了 .NET 运行时的任何平台。

- HelloWorld.exe（在 Linux 上而不是在 macOS 上创建的 HelloWorld）

  这是应用程序的依赖于框架的可执行文件版本。文件特定于操作系统。

- HelloWorld.pdb（对于部署是可选的）

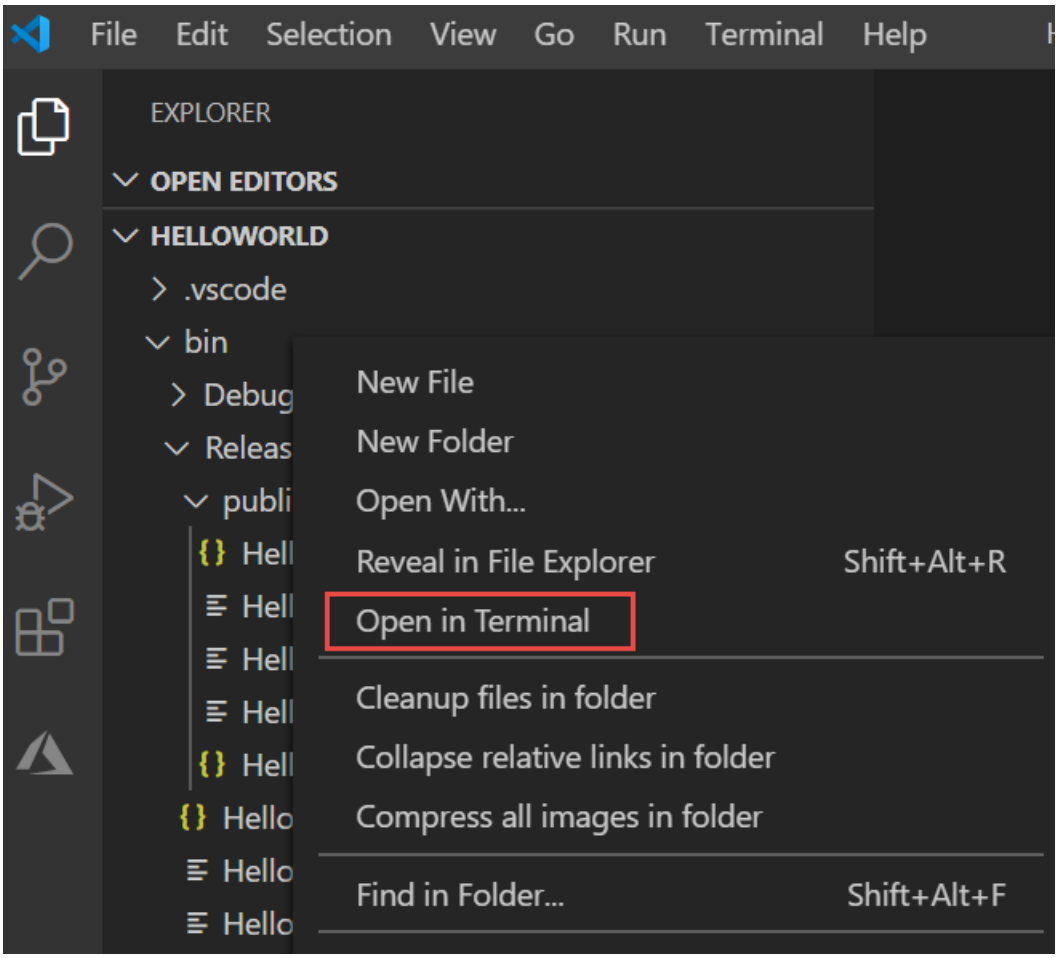
  这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- HelloWorld.runtimeconfig.json

这是应用程序的运行时配置文件。该文件标识用于运行应用程序的 .NET 版本。还可向其添加配置选项。有关详细信息，请参阅 .NET 运行时配置设置。

运行已发布的应用

1. 在“资源管理器”中，右键单击“发布”文件夹(在 macOS 上按住 Ctrl 单击)，然后选择“在终端中打开”。



2. 在 Windows 或 Linux 上，使用可执行文件运行应用。
   a. 在 Windows 上，输入 `.\HelloWorld.exe`，然后按 Enter。
   b. 在 Linux 上，输入 `./HelloWorld`，然后按 Enter。
   c. 输入一个名字以响应提示，并按任意键退出。
3. 在任何平台上，使用 `dotnet` 命令运行应用：
   a. 输入 `dotnet HelloWorld.dll`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。

其他资源

- .NET 应用程序部署

后续步骤

在本教程中，你发布了一个控制台应用。在下一教程中，你将创建类库。

使用 Visual Studio Code 创建 .NET 类库

本教程演示如何发布控制台应用，以便其他用户可以运行它。发布应用程序会创建运行应用程序所需的一组文

件。若要部署文件，请将文件复制到目标计算机。

.NET CLI 用于发布应用，因此可以根据需要使用 Visual Studio Code 以外的代码编辑器来学习本教程。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的控制台应用。

发布应用

1. 启动 Visual Studio Code。
2. 打开在使用 Visual Studio Code 创建 .NET 控制台应用程序中创建的 HelloWorld 项目文件夹。
3. 从主菜单中选择“视图” > “终端” 。

   终端在 HelloWorld 文件夹中打开。
4. 运行下面的命令：

   ```
   dotnet publish --configuration Release
   ```

   默认生成配置为“调试”，因此此命令指定“版本”生成配置 。版本生成配置的输出进行了完全优化，且具有最低限度的符号调试信息。

   该命令的输出类似于以下示例：

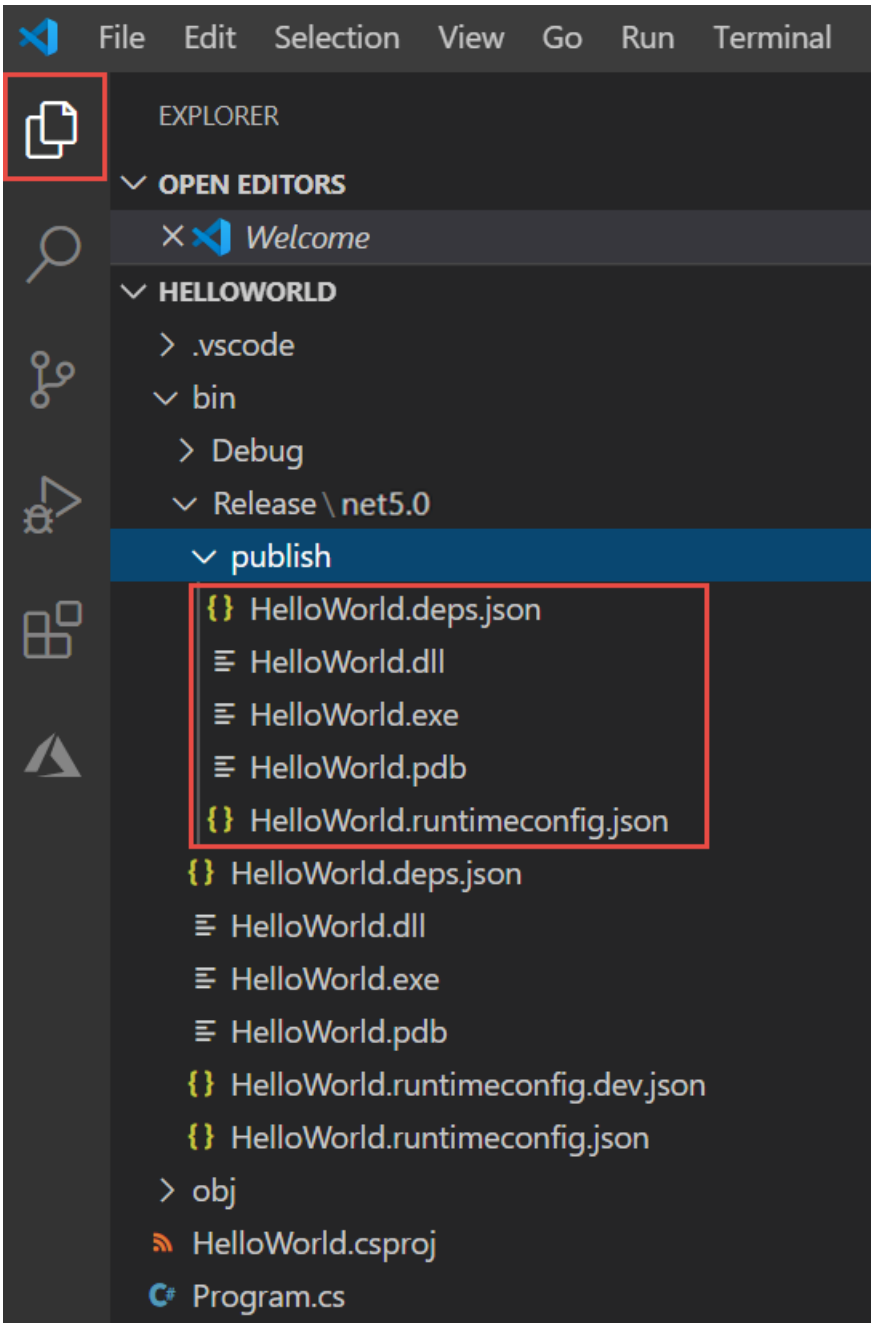
   ```
   Microsoft (R) Build Engine version 16.7.0+b89cb5fde for .NET
   Copyright (C) Microsoft Corporation. All rights reserved.
     Determining projects to restore...
     All projects are up-to-date for restore.
     HelloWorld -> C:\Projects\HelloWorld\bin\Release\net5.0\HelloWorld.dll
     HelloWorld -> C:\Projects\HelloWorld\bin\Release\net5.0\publish\
   ```

检查文件

默认情况下，发布过程中会创建依赖于框架的部署，在此类部署中，已发布的应用程序在已安装 .NET 运行时的计算机上运行。若要运行已发布的应用，可以使用可执行文件，或从命令提示符中运行 `dotnet HelloWorld.dll` 命令。

在下面的步骤中，查看由发布过程创建的文件。

1. 在左侧导航栏中选择“资源管理器”。
2. 展开 bin/Release/net5.0/publish。



如下图所示，已发布的输出包括以下文件：

- HelloWorld.deps.json

  这是应用程序的运行时依赖项文件。该文件定义了运行应用所需的 .NET 组件和库（包括包含应用程序的动态链接库）。有关详细信息，请参阅运行时配置文件。

- HelloWorld.dll

  这是应用程序的依赖于框架的部署版本。若要执行此动态链接库，请在命令提示符处输入 `dotnet HelloWorld.dll`。这种运行应用的方法适用于安装了 .NET 运行时的任何平台。

- HelloWorld.exe（在 Linux 上而不是在 macOS 上创建的 HelloWorld）

  这是应用程序的依赖于框架的可执行文件版本。文件特定于操作系统。

- HelloWorld.pdb（对于部署是可选的）

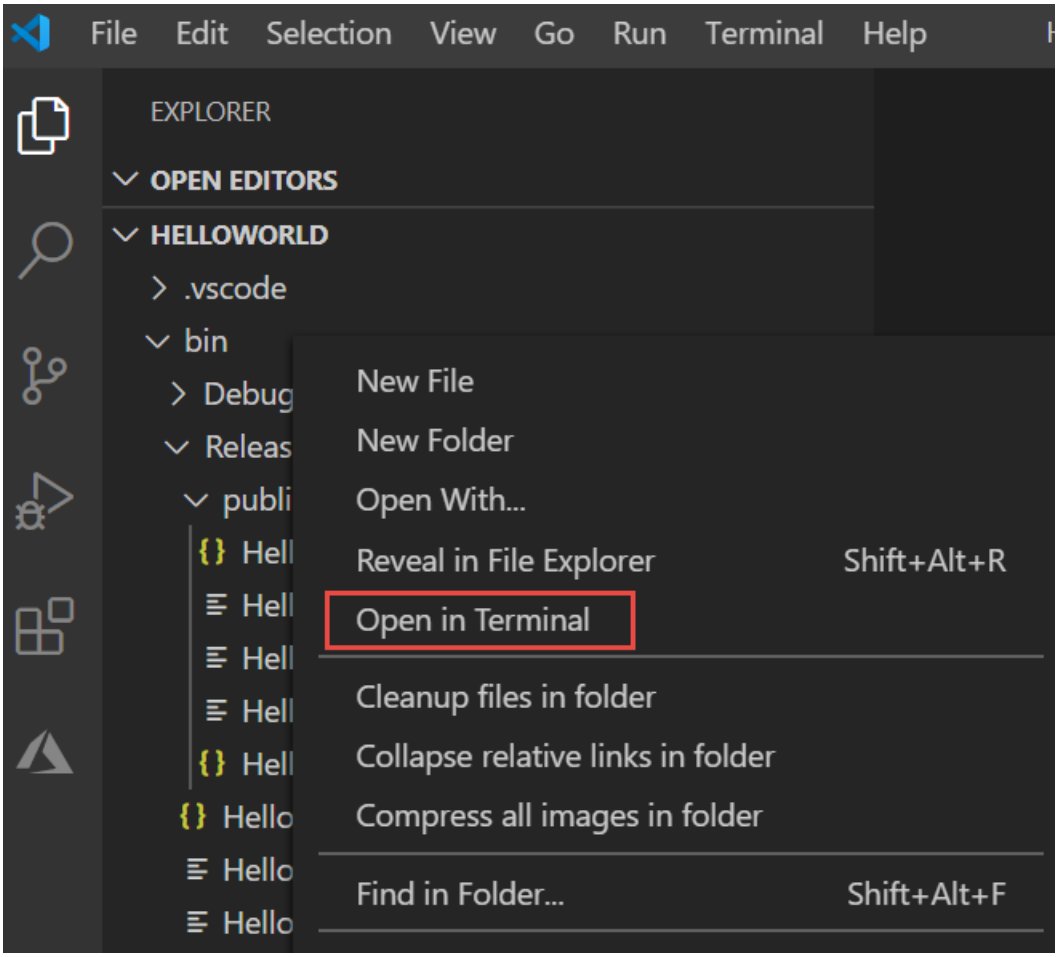
  这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- HelloWorld.runtimeconfig.json

这是应用程序的运行时配置文件。该文件标识用于运行应用程序的 .NET 版本。还可向其添加配置选项。有关详细信息，请参阅 .NET 运行时配置设置。

运行已发布的应用

1. 在“资源管理器”中，右键单击“发布”文件夹（在 macOS 上按住 Ctrl 单击），然后选择“在集成终端中打开”。



2. 在 Windows 或 Linux 上，使用可执行文件运行应用。
   a. 在 Windows 上，输入 `.\HelloWorld.exe`，然后按 Enter。
   b. 在 Linux 上，输入 `./HelloWorld`，然后按 Enter。
   c. 输入一个名字以响应提示，并按任意键退出。
3. 在任何平台上，使用 `dotnet` 命令运行应用：
   a. 输入 `dotnet HelloWorld.dll`，然后按 Enter。
   b. 输入一个名字以响应提示，并按任意键退出。

其他资源

- .NET 应用程序部署

后续步骤

在本教程中，你发布了一个控制台应用。在下一教程中，你将创建类库。

使用 Visual Studio Code 创建 .NET 类库

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio Code 创建 .NET 类库

2021/11/16 •

在本教程中，将创建包含一个字符串处理方法的简单实用工具库。

类库定义的是可以由应用程序调用的类型和方法。如果库以 .NET Standard 2.0 为目标，则支持 .NET Standard 2.0 的任何 .NET 实现（包括 .NET Framework）均可调用该库。如果库以 .NET 6 为目标，则以 .NET 6 为目标的任何应用程序均可调用该库。本教程演示如何以 .NET 6 为目标。

创建类库后，可将其作为第三方组件进行分发，也可将其作为与一个或多个应用程序捆绑在一起的组件进行分发。

先决条件

- 已安装 C# 扩展 的 Visual Studio Code。有关如何在 Visual Studio Code 上安装扩展的信息，请访问 VS Code 扩展市场。
- .NET 6 SDK。

创建解决方案

首先，创建一个空白解决方案来放置类库项目。解决方案用作一个或多个项目的容器。将其他相关项目添加到同一个解决方案中。

1. 启动 Visual Studio Code。
2. 从主菜单中选择“文件” > “打开文件夹”（在 macOS 上为“打开...”）
3. 在“打开文件夹”对话框中，创建“ClassLibraryProjects”文件夹，然后单击“选择文件夹”（在 macOS 上为“打开”）。
4. 在主菜单中选择“视图” > “终端”，从 Visual Studio Code 中打开“终端” 。

   “终端”在“ClassLibraryProjects”文件夹中连同命令提示符一起打开。
5. 在“终端”中输入以下命令：

   ```
   dotnet new sln
   ```

   终端输出如以下示例所示：

   ```
   The template "Solution File" was created successfully.
   ```

创建类库项目

将名为“StringLibrary”的新 .NET 类库项目添加到解决方案。

1. 在终端中，运行以下命令创建库项目：

   ```
   dotnet new classlib -o StringLibrary
   ```

   `-o` 或 `--output` 命令指定用于放置生成的输出的位置。

终端输出如以下示例所示：

```
The template "Class library" was created successfully.
Processing post-creation actions...
Running 'dotnet restore' on StringLibrary\StringLibrary.csproj...
  Determining projects to restore...
  Restored C:\Projects\ClassLibraryProjects\StringLibrary\StringLibrary.csproj (in 328 ms).
Restore succeeded.
```

2. 运行以下命令，向解决方案添加库项目：

```
dotnet sln add StringLibrary/StringLibrary.csproj
```

终端输出如以下示例所示：

```
Project `StringLibrary\StringLibrary.csproj` added to the solution.
```

3. 检查以确保该库以 .NET 6 为目标。在资源管理器中，打开 StringLibrary/StringLibrary.csproj。

`TargetFramework` 元素表明项目以 .NET 6.0 为目标。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
  </PropertyGroup>

</Project>
```

4. 打开 Class1.cs 并将代码替换为以下代码。

```
namespace UtilityLibraries;

public static class StringLibrary
{
    public static bool StartsWithUpper(this string? str)
    {
        if (string.IsNullOrWhiteSpace(str))
            return false;

        char ch = str[0];
        return char.IsUpper(ch);
    }
}
```

类库 `UtilityLibraries.StringLibrary` 包含一个名为 `StartsWithUpper` 的方法。此方法会返回 Boolean 值，以指明当前字符串实例是否以大写字符开头。Unicode 标准会区分大小写字符。如果为大写字符，Char.IsUpper(Char) 方法返回 `true`。

`StartsWithUpper` 以扩展方法的形式进行实现，这样就可以将其作为 String 类成员进行调用。

5. 保存该文件。

6. 运行以下命令以生成解决方案，并验证项目是否正确编译。

```
dotnet build
```

终端输出如以下示例所示：

```
Microsoft (R) Build Engine version 16.7.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.
  Determining projects to restore...
  All projects are up-to-date for restore.
  StringLibrary -> C:\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\net6.0\StringLibrary.dll
Build succeeded.
    0 Warning(s)
    0 Error(s)
Time Elapsed 00:00:02.78
```

向解决方案添加控制台应用

添加使用类库的控制台应用程序。应用将提示用户输入字符串，并报告字符串是否以大写字符开头。

1. 在终端中，运行以下命令创建控制台应用项目：

   ```
   dotnet new console -o ShowCase
   ```

   终端输出如以下示例所示：

   ```
   The template "Console Application" was created successfully.
   Processing post-creation actions...
   Running 'dotnet restore' on ShowCase\ShowCase.csproj...
     Determining projects to restore...
     Restored C:\Projects\ClassLibraryProjects\ShowCase\ShowCase.csproj (in 210 ms).
   Restore succeeded.
   ```

2. 运行以下命令，向解决方案添加控制台应用项目：

   ```
   dotnet sln add ShowCase/ShowCase.csproj
   ```

   终端输出如以下示例所示：

   ```
   Project `ShowCase\ShowCase.csproj` added to the solution.
   ```

3. 打开 ShowCase/Program.cs 并将所有代码替换为以下代码。

```
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string? input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input}");
            Console.WriteLine("Begins with uppercase? " +
                 $"{(input.StartsWithUpper() ? "Yes" : "No")}");
            Console.WriteLine();
            row += 4;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}");
            row = 3;
        }
    }
}
```

该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

4. 保存更改。

添加项目引用

最初，新的控制台应用项目无权访问类库。若要允许该项目调用类库中的方法，可以创建对类库项目的项目引用。

1. 运行下面的命令：

```
dotnet add ShowCase/ShowCase.csproj reference StringLibrary/StringLibrary.csproj
```

终端输出如以下示例所示：

```
Reference `..\StringLibrary\StringLibrary.csproj` added to the project.
```

运行应用

1. 在终端中运行以下命令：

```
dotnet run --project ShowCase/ShowCase.csproj
```

2. 输入字符串并按 Enter 以试用程序，然后按 Enter 退出。

   终端输出如以下示例所示：

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:

A string that starts with an uppercase letter
Input: A string that starts with an uppercase letter
Begins with uppercase? : Yes

a string that starts with a lowercase letter
Input: a string that starts with a lowercase letter
Begins with uppercase? : No
```

其他资源

- 使用 .NET CLI 开发库
- .NET Standard 版本及其支持的平台。

后续步骤

在本教程中，你创建了一个解决方案，添加了一个库项目，并添加了一个使用该库的控制台应用项目。在下一教程中，将向解决方案中添加单元测试项目。

使用 Visual Studio Code 通过 .NET 测试 .NET 类库

在本教程中，将创建包含一个字符串处理方法的简单实用工具库。

类库定义的是可以由应用程序调用的类型和方法。如果库以 .NET Standard 2.0 为目标，则支持 .NET Standard 2.0 的任何 .NET 实现（包括 .NET Framework）均可调用该库。如果库以 .NET 5 为目标，则以 .NET 5 为目标的任何应用程序均可调用该库。本教程演示如何以 .NET 5 为目标。

创建类库后，可将其作为第三方组件进行分发，也可将其作为与一个或多个应用程序捆绑在一起的组件进行分发。

先决条件

1. 已安装 C# 扩展 的 Visual Studio Code。有关如何在 Visual Studio Code 上安装扩展的信息，请访问 VS Code 扩展市场。
2. .NET 5.0 SDK 或更高版本

创建解决方案

首先，创建一个空白解决方案来放置类库项目。解决方案用作一个或多个项目的容器。将其他相关项目添加到同一个解决方案中。

1. 启动 Visual Studio Code。

2. 从主菜单中选择“文件” > “打开文件夹”（在 macOS 上为“打开...”）

3. 在“打开文件夹”对话框中，创建“ClassLibraryProjects”文件夹，然后单击“选择文件夹”（在 macOS 上为“打

开”）。

4. 在主菜单中选择“视图” > “终端”，从 Visual Studio Code 中打开“终端” 。

   “终端”在“ClassLibraryProjects”文件夹中连同命令提示符一起打开。

5. 在”终端”中输入以下命令：

   ```
   dotnet new sln
   ```

   终端输出如以下示例所示：

   ```
   The template "Solution File" was created successfully.
   ```

创建类库项目

将名为“StringLibrary”的新 .NET 类库项目添加到解决方案。

1. 在终端中，运行以下命令创建库项目：

   ```
   dotnet new classlib -o StringLibrary
   ```

   `-o` 或 `--output` 命令指定用于放置生成的输出的位置。

   终端输出如以下示例所示：

   ```
   The template "Class library" was created successfully.
   Processing post-creation actions...
   Running 'dotnet restore' on StringLibrary\StringLibrary.csproj...
     Determining projects to restore...
     Restored C:\Projects\ClassLibraryProjects\StringLibrary\StringLibrary.csproj (in 328 ms).
   Restore succeeded.
   ```

2. 运行以下命令，向解决方案添加库项目：

   ```
   dotnet sln add StringLibrary/StringLibrary.csproj
   ```

   终端输出如以下示例所示：

   ```
   Project `StringLibrary\StringLibrary.csproj` added to the solution.
   ```

3. 检查以确保该库以 .NET 5 为目标。在资源管理器中，打开 StringLibrary/StringLibrary.csproj。

   `TargetFramework` 元素表明项目以 .NET 5.0 为目标。

   ```
   <Project Sdk="Microsoft.NET.Sdk">

     <PropertyGroup>
       <TargetFramework>net5.0</TargetFramework>
     </PropertyGroup>

   </Project>
   ```

4. 打开 Class1.cs 并将代码替换为以下代码。

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this string str)
        {
            if (string.IsNullOrWhiteSpace(str))
                return false;

            char ch = str[0];
            return char.IsUpper(ch);
        }
    }
}
```

类库 `UtilityLibraries.StringLibrary` 包含一个名为 `StartsWithUpper` 的方法。此方法会返回 Boolean 值，以指明当前字符串实例是否以大写字符开头。Unicode 标准会区分大小写字符。如果为大写字符，Char.IsUpper(Char) 方法返回 `true`。

`StartsWithUpper` 以扩展方法的形式进行实现，这样就可以将其作为 String 类成员进行调用。`string` 后的问号 (`?`) 表示该字符串可能为 null。

5. 保存该文件。

6. 运行以下命令以生成解决方案，并验证项目是否正确编译。

```
dotnet build
```

终端输出如以下示例所示：

```
Microsoft (R) Build Engine version 16.7.0+b89cb5fde for .NET
Copyright (C) Microsoft Corporation. All rights reserved.
  Determining projects to restore...
  All projects are up-to-date for restore.
  StringLibrary -> C:\Projects\ClassLibraryProjects\StringLibrary\bin\Debug\net5.0\StringLibrary.dll
Build succeeded.
    0 Warning(s)
    0 Error(s)
Time Elapsed 00:00:02.78
```

向解决方案添加控制台应用

添加使用类库的控制台应用程序。应用将提示用户输入字符串，并报告字符串是否以大写字符开头。

1. 在终端中，运行以下命令创建控制台应用项目：

```
dotnet new console -o ShowCase
```

终端输出如以下示例所示：

```
The template "Console Application" was created successfully.
Processing post-creation actions...
Running 'dotnet restore' on ShowCase\ShowCase.csproj...
  Determining projects to restore...
  Restored C:\Projects\ClassLibraryProjects\ShowCase\ShowCase.csproj (in 210 ms).
Restore succeeded.
```

2. 运行以下命令，向解决方案添加控制台应用项目：

```
dotnet sln add ShowCase/ShowCase.csproj
```

终端输出如以下示例所示：

```
Project `ShowCase\ShowCase.csproj` added to the solution.
```

3. 打开 ShowCase/Program.cs 并将所有代码替换为以下代码。

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{(input.StartsWithUpper() ? "Yes" : "No")}{Environment.NewLine}");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}");
            row = 3;
        }
    }
}
```

该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

4. 保存更改。

添加项目引用

最初，新的控制台应用项目无权访问类库。若要允许该项目调用类库中的方法，可以创建对类库项目的项目引用。

1. 运行下面的命令：

```
dotnet add ShowCase/ShowCase.csproj reference StringLibrary/StringLibrary.csproj
```

终端输出如以下示例所示：

```
Reference `..\StringLibrary\StringLibrary.csproj` added to the project.
```

运行应用

1. 在终端中运行以下命令：

```
dotnet run --project ShowCase/ShowCase.csproj
```

2. 输入字符串并按 Enter 以试用程序，然后按 Enter 退出。

终端输出如以下示例所示：

```
Press <Enter> only to exit; otherwise, enter a string and press <Enter>:

A string that starts with an uppercase letter
Input: A string that starts with an uppercase letter
Begins with uppercase? : Yes

a string that starts with a lowercase letter
Input: a string that starts with a lowercase letter
Begins with uppercase? : No
```

其他资源

- 使用 .NET CLI 开发库
- .NET Standard 版本及其支持的平台。

后续步骤

在本教程中，你创建了一个解决方案，添加了一个库项目，并添加了一个使用该库的控制台应用项目。在下一教程中，将向解决方案中添加单元测试项目。

使用 Visual Studio Code 通过 .NET 测试 .NET 类库

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio Code 测试 .NET 类库

2021/11/16 •

本教程演示如何通过将测试项目添加到解决方案来自动执行单元测试。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 类库中创建的解决方案。

创建单元测试项目

单元测试在开发和发布期间提供自动化的软件测试。本教程中使用的测试框架是 MSTest。MSTest 是可供选择的三个测试框架之一。其他两个是 xUnit 和 nUnit。

1. 启动 Visual Studio Code。
2. 打开在使用 Visual Studio Code 创建 .NET 类库中创建的 `ClassLibraryProjects` 解决方案。
3. 创建名为“StringLibraryTest”的单元测试项目。

   ```
   dotnet new mstest -o StringLibraryTest
   ```

   项目模板创建包含以下代码的 UnitTest1.cs 文件：

   ```
   using Microsoft.VisualStudio.TestTools.UnitTesting;

   namespace StringLibraryTest
   {
       [TestClass]
       public class UnitTest1
       {
           [TestMethod]
           public void TestMethod1()
           {
           }
       }
   }
   ```

   单元测试模板创建的源代码负责执行以下操作：

   - 它会导入 Microsoft.VisualStudio.TestTools.UnitTesting 命名空间，其中包含用于单元测试的类型。
   - 向 `UnitTest1` 类应用 TestClassAttribute 特性。
   - 它应用 TestMethodAttribute 特性来定义 `TestMethod1`。

   使用 [TestClass] 标记的测试类中标记有 [TestMethod] 的所有测试方法都会在调用单元测试时自动运行。

4. 向解决方案添加测试项目。

   ```
   dotnet sln add StringLibraryTest/StringLibraryTest.csproj
   ```

添加项目引用

对于要使用 `StringLibrary` 类的测试库，请在 `StringLibraryTest` 项目中添加对 `StringLibrary` 项目的引用。

1. 运行下面的命令：

```
dotnet add StringLibraryTest/StringLibraryTest.csproj reference StringLibrary/StringLibrary.csproj
```

添加并运行单元测试方法

调用单元测试时，Visual Studio 运行使用 TestClassAttribute 特性标记的类中标记有 TestMethodAttribute 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 Assert 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

`Assert.AreEqual`	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
`Assert.AreSame`	验证两个对象变量引用的是否是同一个对象。如果这些变量引用不同的对象，则断言失败。
`Assert.IsFalse`	验证条件是否为 `false`。如果条件为 `true`，则断言失败。
`Assert.IsNotNull`	验证对象是否不为 `null`。如果对象为 `null`，则断言失败。

还可以在测试方法中使用 Assert.ThrowsException 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 Assert.IsTrue 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况下，此方法应返回 `false`，以便可以调用 Assert.IsFalse 方法。

由于库方法可以处理字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`) 和 `null` 字符串。空字符串不包含任何字符，且 Length 为 0。`null` 字符串是尚未初始化的字符串。可以直接将 `StartsWithUpper` 作为静态方法进行调用，并向其传递一个 String 自变量。或者，可以对分配给 `null` 的 `string` 变量将 `StartsWithUpper` 作为扩展方法进行调用。

将定义三个方法，每个方法都会对字符串数组中的各个元素调用它的 Assert 方法。你将调用方法重载，以便指定在测试失败时要显示的错误消息。消息标识导致失败的字符串。

创建测试方法：

1. 打开 StringLibraryTest/UnitTest1.cs 并将所有代码替换为以下代码。

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest;

[TestClass]
public class UnitTest1
{
    [TestMethod]
    public void TestStartsWithUpper()
    {
        // Tests that we expect to return true.
        string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
        foreach (var word in words)
        {
            bool result = word.StartsWithUpper();
            Assert.IsTrue(result,
                   String.Format("Expected for '{0}': true; Actual: {1}",
                                 word, result));
        }
    }

    [TestMethod]
    public void TestDoesNotStartWithUpper()
    {
        // Tests that we expect to return false.
        string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                           "1234", ".", ";", " " };
        foreach (var word in words)
        {
            bool result = word.StartsWithUpper();
            Assert.IsFalse(result,
                   String.Format("Expected for '{0}': false; Actual: {1}",
                                 word, result));
        }
    }

    [TestMethod]
    public void DirectCallWithNullOrEmpty()
    {
        // Tests that we expect to return false.
        string?[] words = { string.Empty, null };
        foreach (var word in words)
        {
            bool result = StringLibrary.StartsWithUpper(word);
            Assert.IsFalse(result,
                   String.Format("Expected for '{0}': false; Actual: {1}",
                                 word == null ? "<null>" : word, result));
        }
    }
}
```

`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

2. 保存更改。

3. 运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

终端输出显示所有测试都已通过。

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     3, Skipped:     0, Total:     3, Duration: 3 ms -
StringLibraryTest.dll (net6.0)
```

处理测试失败

如果进行的是测试驱动开发 (TDD)，请先编写测试，然后测试会在第一次运行时失败。接着将可以使测试成功的代码添加到应用。在本教程中，先编写了测试要验证的应用代码然后才创建测试，所以没有看到测试失败。若要验证测试是否在预期失败时失败，请在测试输入中添加无效值。

1. 通过修改 `TestDoesNotStartWithUpper` 方法中的 `words` 数组来包含字符串“Error”。

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

2. 运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

终端输出显示一个测试失败，并提供关于失败测试的错误消息：“Assert.IsFalse 失败。“Error”应返回 false；实际返回 True”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
  Failed TestDoesNotStartWithUpper [28 ms]
  Error Message:
   Assert.IsFalse failed. Expected for 'Error': false; Actual: True
  Stack Trace:
     at StringLibraryTest.UnitTest1.TestDoesNotStartWithUpper() in
C:\ClassLibraryProjects\StringLibraryTest\UnitTest1.cs:line 33

Failed!  - Failed:     1, Passed:     2, Skipped:     0, Total:     3, Duration: 31 ms -
StringLibraryTest.dll (net5.0)
```

3. 删除在步骤 1 中添加的字符串“Error”。重新运行测试，测试将通过。

测试库的发行版本

至此，在运行库的调试版本时，测试已全部通过，接下来应对库的发行版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

1. 使用版本生成配置运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj --configuration Release
```

测试通过。

调试测试

如果使用 Visual Studio 作为 IDE，则可以使用与使用 Visual Studio Code 调试 .NET Core 控制台应用程序中所示的相同过程，来通过使用单元测试项目调试代码。打开 StringLibraryTest/UnitTest1.cs，然后在第 7 行和第 8 行之间选择“调试所有测试”，而不是启动 ShowCase 应用项目。如果找不到该位置，请按下 `Ctrl+Shift+P` 打开命

令面板，然后输入“重载窗口”。

Visual Studio Code 启动附有调试器的测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

其他资源

- .NET 中的单元测试

后续步骤

在本教程中，你对类库进行了单元测试。你可以将库作为包发布到 NuGet，使其可供其他人使用。若要了解如何操作，请遵循 NuGet 教程：

使用 dotnet CLI 创建和发布包

如果将库作为 NuGet 包发布，其他人可以安装并使用它。若要了解如何操作，请遵循 NuGet 教程：

使用 dotnet CLI 安装并使用包

库并非必须作为包进行分发。它还可与使用它的控制台应用捆绑在一起。若要了解如何发布控制台应用，请参阅本系列中前面的教程：

使用 Visual Studio Code 发布 .NET 控制台应用程序

本教程演示如何通过将测试项目添加到解决方案来自动执行单元测试。

先决条件

- 本教程适用于在使用 Visual Studio Code 创建 .NET 类库中创建的解决方案。

创建单元测试项目

单元测试在开发和发布期间提供自动化的软件测试。本教程中使用的测试框架是 MSTest。MSTest 是可供选择的三个测试框架之一。其他两个是 xUnit 和 nUnit。

1. 启动 Visual Studio Code。
2. 打开在使用 Visual Studio Code 创建 .NET 类库中创建的 `ClassLibraryProjects` 解决方案。
3. 创建名为“StringLibraryTest”的单元测试项目。

```
dotnet new mstest -o StringLibraryTest
```

项目模板创建包含以下代码的 UnitTest1.cs 文件：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

单元测试模板创建的源代码负责执行以下操作：

- 它会导入 Microsoft.VisualStudio.TestTools.UnitTesting 命名空间，其中包含用于单元测试的类型。
- 向 `UnitTest1` 类应用 TestClassAttribute 特性。
- 它应用 TestMethodAttribute 特性来定义 `TestMethod1`。

使用 [TestClass] 标记的测试类中标记有 [TestMethod] 的所有测试方法都会在单元测试运行时自动执行。

4. 向解决方案添加测试项目。

```
dotnet sln add StringLibraryTest/StringLibraryTest.csproj
```

添加项目引用

对于要使用 `StringLibrary` 类的测试库，请在 `StringLibraryTest` 项目中添加对 `StringLibrary` 项目的引用。

1. 运行下面的命令：

```
dotnet add StringLibraryTest/StringLibraryTest.csproj reference StringLibrary/StringLibrary.csproj
```

添加并运行单元测试方法

运行单元测试时，Visual Studio 执行使用 TestClassAttribute 特性标记的类中标记有 TestMethodAttribute 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 Assert 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

`Assert.AreEqual`	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
`Assert.AreSame`	验证两个对象变量引用的是否是同一个对象。如果这些变量引用不同的对象，则断言失败。
`Assert.IsFalse`	验证条件是否为 `false`。如果条件为 `true`，则断言失败。
`Assert.IsNotNull`	验证对象是否不为 `null`。如果对象为 `null`，则断言失败。

还可以在测试方法中使用 Assert.ThrowsException 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 Assert.IsTrue 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况下，此方法应返回 `false`，以便可以调用 Assert.IsFalse 方法。

由于库方法可以处理字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`) 和 `null` 字符串。空字符串不包含任何字符，且 Length 为 0。`null` 字符串是尚未初始化的字符串。可以直接将 `StartsWithUpper` 作为静态方法进行调用，并向其传递一个 String 自变量。或者，可以对分配给 `null` 的 `string` 变量将 `StartsWithUpper` 作为扩展方法进行调用。

将定义三个方法，每个方法都会对字符串数组中的各个元素调用它的 Assert 方法。你将调用方法重载，以便指定在测试失败时要显示的错误消息。消息标识导致失败的字符串。

创建测试方法：

1. 打开 StringLibraryTest/UnitTest1.cs 并将所有代码替换为以下代码。

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

2. 保存更改。

3. 运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

终端输出显示所有测试都已通过。

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     3, Skipped:     0, Total:     3, Duration: 3 ms -
StringLibraryTest.dll (net5.0)
```

处理测试失败

如果进行的是测试驱动开发 (TDD)，请先编写测试，然后测试会在第一次运行时失败。接着将可以使测试成功的代码添加到应用。在本教程中，先编写了测试要验证的应用代码然后才创建测试，所以没有看到测试失败。若要验证测试是否在预期失败时失败，请在测试输入中添加无效值。

1. 通过修改 `TestDoesNotStartWithUpper` 方法中的 `words` 数组来包含字符串“Error”。

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

2. 运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj
```

终端输出显示一个测试失败，并提供关于失败测试的错误消息：“Assert.IsFalse 失败。“Error”应返回 false；实际返回 True”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。

```
Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
  Failed TestDoesNotStartWithUpper [28 ms]
  Error Message:
   Assert.IsFalse failed. Expected for 'Error': false; Actual: True
  Stack Trace:
     at StringLibraryTest.UnitTest1.TestDoesNotStartWithUpper() in
C:\ClassLibraryProjects\StringLibraryTest\UnitTest1.cs:line 33

Failed!  - Failed:     1, Passed:     2, Skipped:     0, Total:     3, Duration: 31 ms -
StringLibraryTest.dll (net5.0)
```

3. 删除在步骤 1 中添加的字符串“Error”。重新运行测试，测试将通过。

测试库的发行版本

至此，在运行库的调试版本时，测试已全部通过，接下来应对库的发行版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

1. 使用版本生成配置运行测试：

```
dotnet test StringLibraryTest/StringLibraryTest.csproj --configuration Release
```

测试通过。

调试测试

如果使用 Visual Studio 作为 IDE，则可以使用与使用 Visual Studio Code 调试 .NET Core 控制台应用程序中所示的相同过程，来通过使用单元测试项目调试代码。打开 StringLibraryTest/UnitTest1.cs，然后在第 7 行和第 8 行之间选择“调试所有测试”，而不是启动 ShowCase 应用项目。如果找不到该位置，请按下 `Ctrl+Shift+P` 打开命令面板，然后输入“重载窗口”。

Visual Studio Code 启动附有调试器的测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

其他资源

- .NET 中的单元测试

后续步骤

在本教程中，你对类库进行了单元测试。你可以将库作为包发布到 NuGet，使其可供其他人使用。若要了解如何操作，请遵循 NuGet 教程：

使用 dotnet CLI 创建和发布包

如果将库作为 NuGet 包发布，其他人可以安装并使用它。若要了解如何操作，请遵循 NuGet 教程：

使用 dotnet CLI 安装并使用包

库并非必须作为包进行分发。它还可与使用它的控制台应用捆绑在一起。若要了解如何发布控制台应用，请参阅本系列中前面的教程：

使用 Visual Studio Code 发布 .NET 控制台应用程序

本教程仅适用于 .NET 5 和 .NET 6。在页面顶部选择其中一个选项。

教程：使用 Visual Studio for Mac 创建 .NET 控制台应用程序

2021/11/16 •

本教程演示如何使用 Visual Studio for Mac 创建和运行 .NET 控制台应用程序。

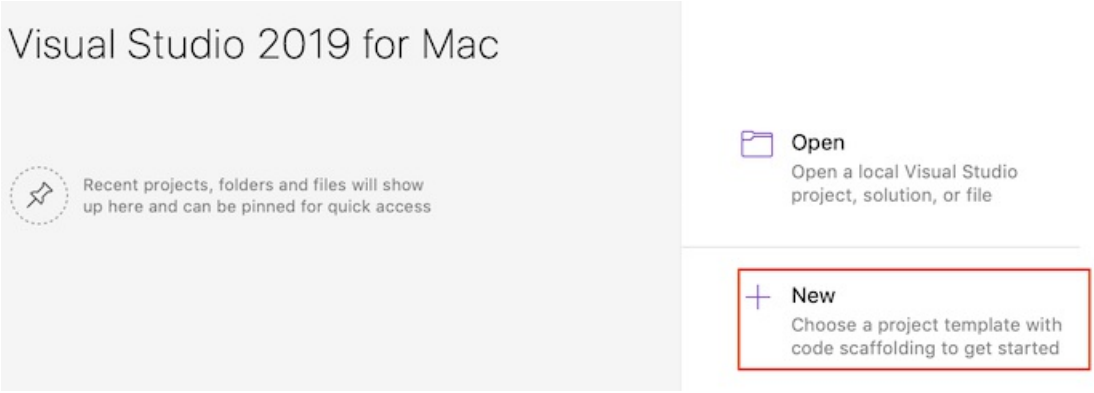
> **NOTE**
>
> 你的反馈非常有价值。有两种方法可以向开发团队提供有关 Visual Studio for Mac 的反馈：
>
> - 在 Visual Studio for Mac 中，从菜单中选择“帮助” > “报告问题”，或从欢迎屏幕中选择“报告问题”，将打开一个窗口，以供填写 bug 报告。可在开发人员社区门户中跟踪自己的反馈。
> - 若要提出建议，从菜单中选择“帮助” > “提供建议”，或从欢迎屏幕中选择“提供建议”，转到 Visual Studio for Mac 开发人员社区网页。

先决条件

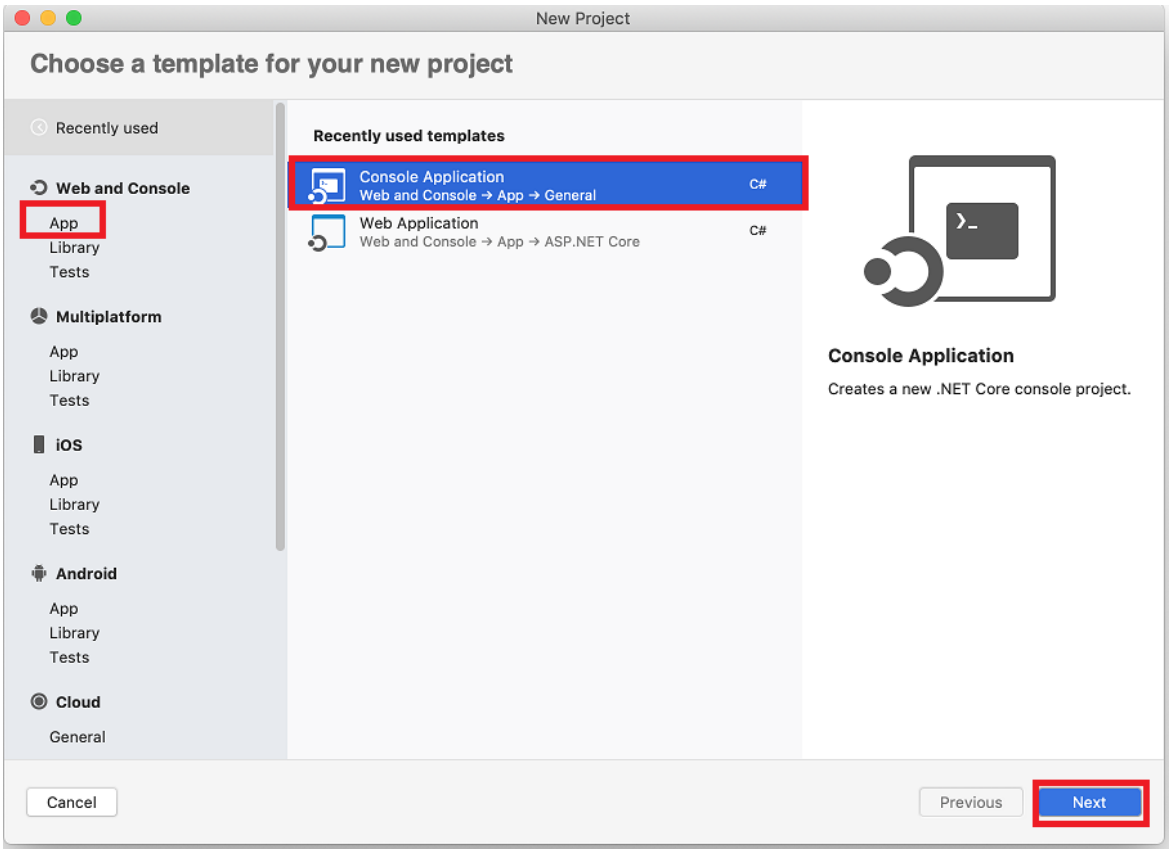
- Visual Studio for Mac 8.8 或更高版本。选择用于安装 .NET Core 的选项。安装 Xamarin 对于 .NET 开发而言是可选项。有关更多信息，请参见以下资源：
  - 教程：安装 Visual Studio for Mac。
  - 支持的 macOS 版本。
  - Visual Studio for Mac 支持的 .NET 版本。

创建应用

1. 启动 Visual Studio for Mac。
2. 在“开始”窗口中，选择“新建”。

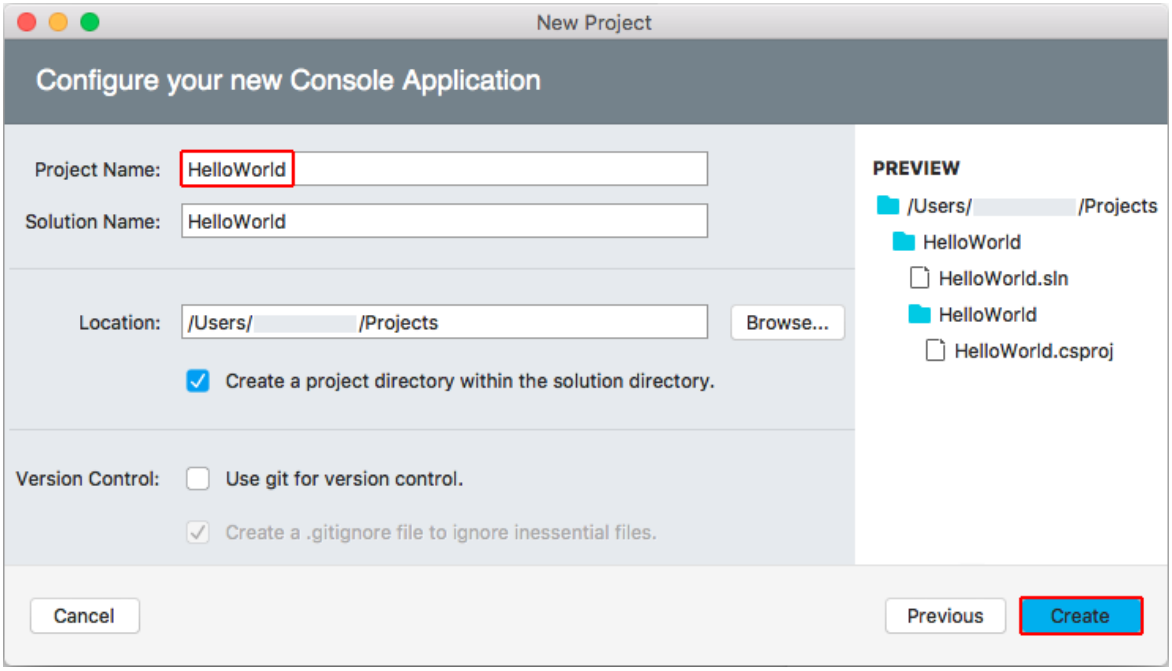


3. 在“新建项目”对话框中，选择“Web 和控制台”节点下的“应用”。选择“控制台应用程序”模板，然后选择“下一步”。



4. 在“配置新的控制台应用程序”对话框的“目标框架”下拉列表中，选择“.NET 5.0”，然后选择“下一步”。

5. 键入“HelloWorld”作为“项目名称”，然后选择“创建”。



用于创建简单的“Hello World”应用程序的模板。它会调用 Console.WriteLine(String) 方法来显示“Hello World!”（在终端窗口中）。

模板代码将定义类 `Program`，其中包含一个将 String 数组用作参数的方法 `Main`：

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

`Main` 是应用程序入口点，同时也是在应用程序启动时由运行时自动调用的方法。`args` 数组中包含在应用程序启动时提供的所有命令行参数。

运行应用

1. 按 ⌥⌘↵ (option+command+enter) 以运行应用而不进行调试。

2. 关闭终端窗口。

增强应用

改进应用程序，使其提示用户输入名字，并将其与日期和时间一同显示。

1. 在 Program.cs 中，将 `Main` 方法的内容（是调用 `Console.WriteLine` 的行）替换为以下代码：

   ```
   Console.WriteLine("What is your name?");
   var name = Console.ReadLine();
   var currentDate = DateTime.Now;
   Console.WriteLine($"{Environment.NewLine}Hello, {name}, on {currentDate:d} at {currentDate:t}!");
   Console.Write($"{Environment.NewLine}Press any key to exit...");
   Console.ReadKey(true);
   ```

   此代码会在控制台窗口中显示一条提示，然后等待用户输入字符串并按 Enter。它会将此字符串存储到名为 `name` 的变量中。它还会检索 DateTime.Now 属性的值（其中包含当前的本地时间），并将此值赋给 `currentDate` 变量。同时会在控制台窗口中显示这些值。最后会在控制台窗口中显示一条提示，并调用 Console.ReadKey(Boolean) 方法来等待用户输入。

   NewLine 是一种独立于平台和语言的表示换行符的方式。替代方法是在 C# 中使用 `\n` 和在 Visual Basic 中使用 `vbCrLf`。

   字符串前面的美元符号 (`$`) 使你可以将表达式（如变量名称）放入字符串中的大括号内。表达式值将代替表达式插入到字符串中。此语法称为内插字符串。

2. 按 ⌥⌘↵ (option+command+enter) 以运行应用。

3. 输入名称并按 Enter，从而响应提示。

   ```
   Terminal – HelloWorld

   What is your name?
   Nancy

   Hello, Nancy, on 6/4/2020 at 3:42 PM!

   Press any key to exit...
   ```

4. 关闭终端。

后续步骤

在本教程中，你创建了一个 .NET 控制台应用程序。在下一教程中，你将调试该应用。

使用 Visual Studio for Mac 调试 .NET 控制台应用程序

教程：使用 Visual Studio for Mac 调试 .NET 控制台应用程序

2021/11/16 •

本教程介绍了 Visual Studio for Mac 中提供的调试工具。

先决条件

- 本教程适用于在使用 Visual Studio for Mac 创建 .NET 控制台应用程序中创建的控制台应用。

使用“调试”生成配置

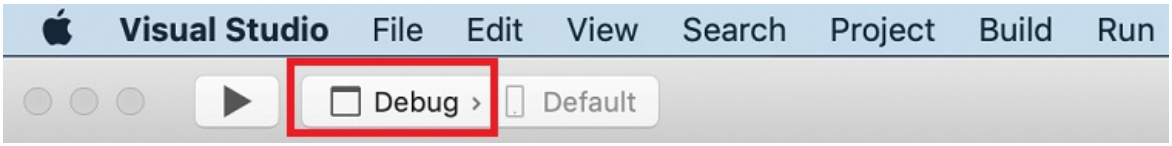
“调试”和“发布”是 Visual Studio 的内置生成配置 。可使用“调试”生成配置进行调试，使用“发布”配置进行最终版本分发。

在“调试”配置中，程序使用完整符号调试信息编译，且不进行优化。优化会使调试复杂化，因为源代码和生成的指令之间的关系更加复杂。程序的发布配置进行了完全优化，且不包含任何符号调试信息。

默认情况下，Visual Studio for Mac 使用“调试”生成配置，因此不需要在调试之前对其进行更改。

1. 启动 Visual Studio for Mac。
2. 打开在使用 Visual Studio for Mac 创建 .NET 控制台应用程序中创建的项目。

   当前的生成配置显示在工具栏上。下面的工具栏图像显示 Visual Studio 配置为编译应用的“调试”版本：

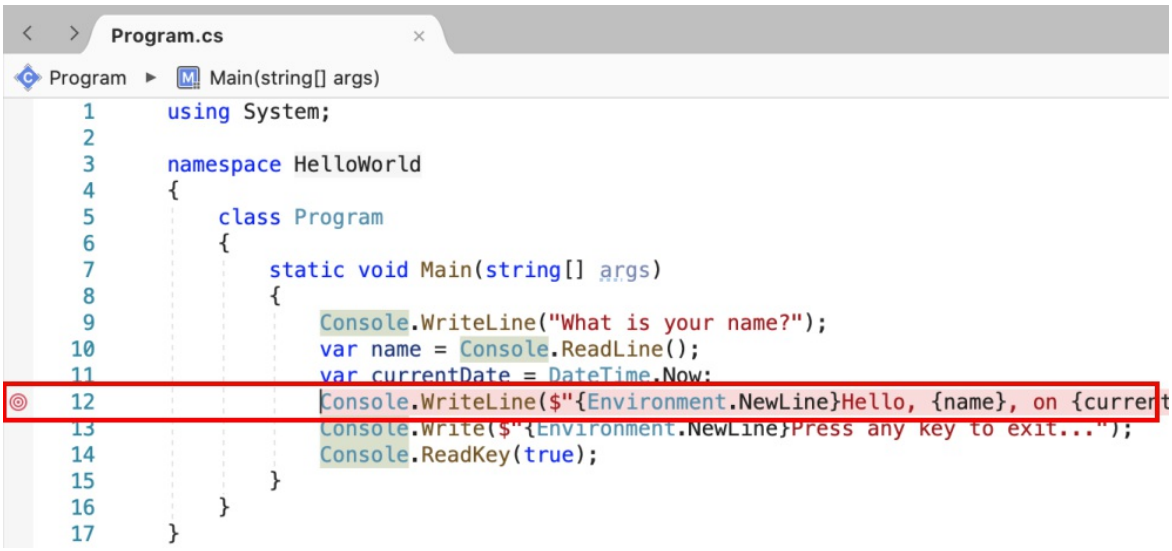


设置断点

断点会在执行包含断点的代码行之前暂时中断执行应用程序。

1. 在显示名称、日期和时间的行上设置断点。为此，请将光标放在代码行中，然后按 ⌘\ (command+\)。设置断点的另一种方法是从菜单中选择“运行” > “切换断点”。

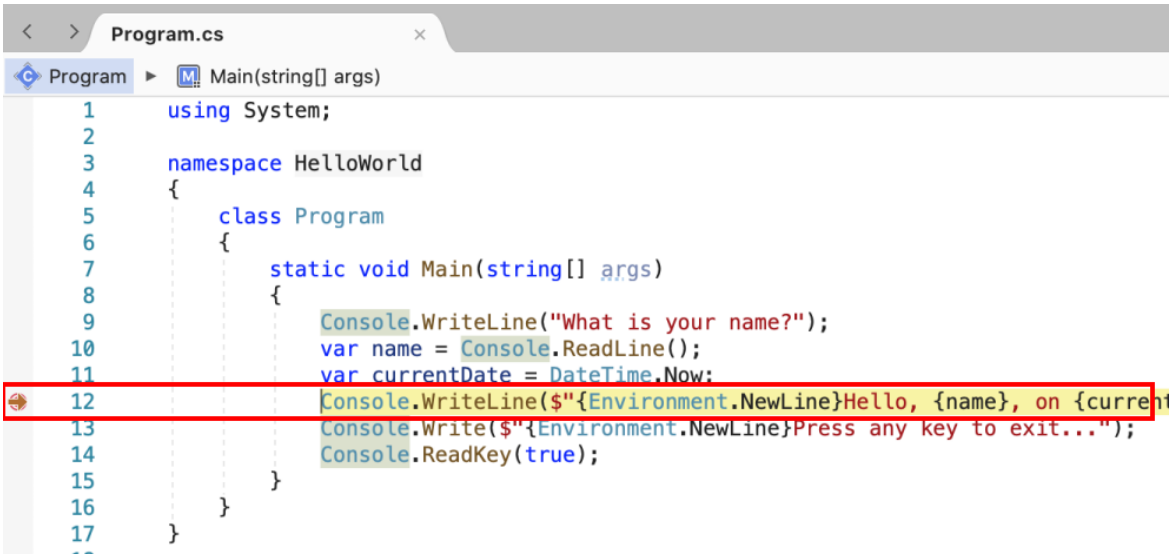
   Visual Studio 通过突出显示此代码行并在左边缘显示红点来指示设置了断点的行。



2. 按 ⌘↵ (command+enter) 以在调试模式下启动程序。启动调试的另一种方法是从菜单中选择“运行” > “启动调试”。

3. 当程序提示输入名称时，在终端窗口中输入字符串，然后按 Enter。

4. 到达断点时，程序停止执行，然后执行 `Console.WriteLine` 方法。



使用“即时”窗口

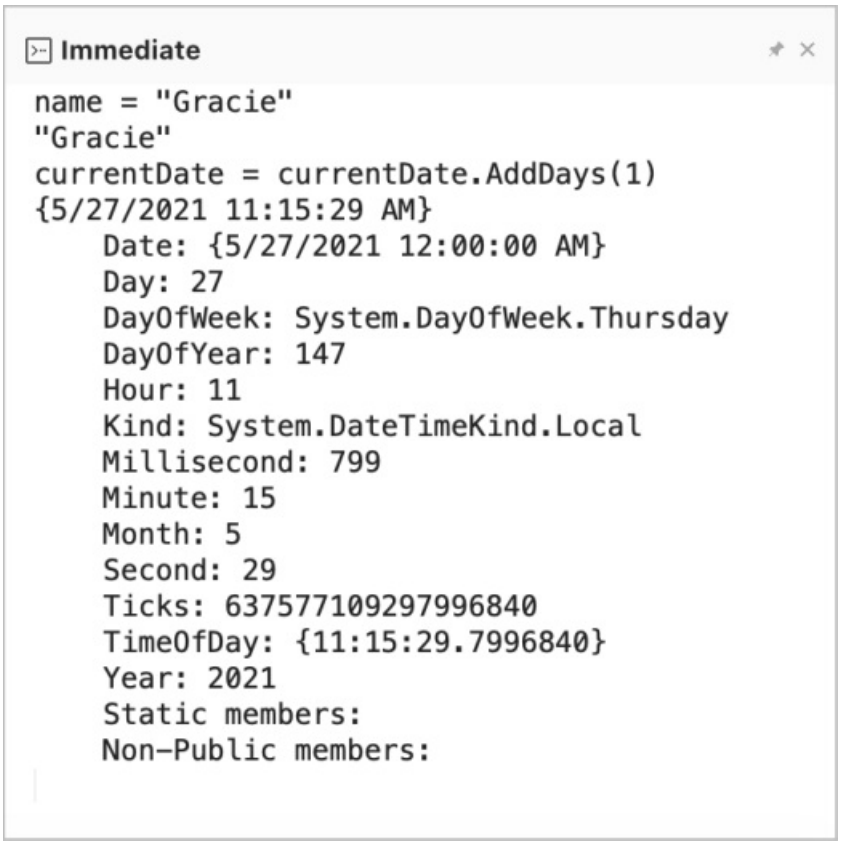
在“即时”窗口中，可以与正在调试的应用程序进行交互。可以通过交互方式更改变量值，看看这样会对程序产生哪些影响。

1. 如果“即时”窗口不可见，请选择“视图” > “调试边栏” > “即时”来显示它。

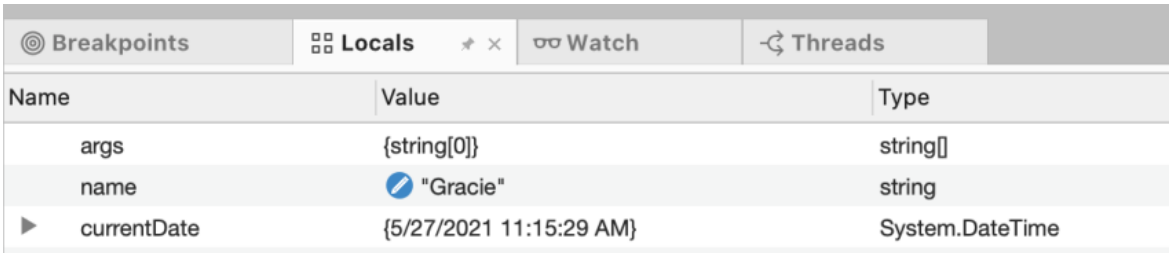
2. 在“即时”窗口中输入 `name = "Gracie"`，然后按 Enter。

3. 在“即时”窗口中输入 `currentDate = currentDate.AddDays(1)`，然后按 Enter。

   “即时”窗口显示字符串变量的新值和 DateTime 值的属性。



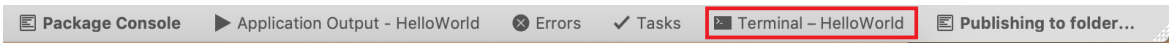
"局部变量"窗口显示当前正在执行的方法中定义的变量值。"局部变量"窗口中会更新刚更改的变量的值。



4. 按 ⌘↵ (command+enter) 以继续调试。

   终端中显示的值对应于在"即时"窗口中所做的更改。

   如果看不到终端，请选择底部导航栏中的"终端 - HelloWorld"。



5. 按任意键退出程序。

6. 关闭终端窗口。

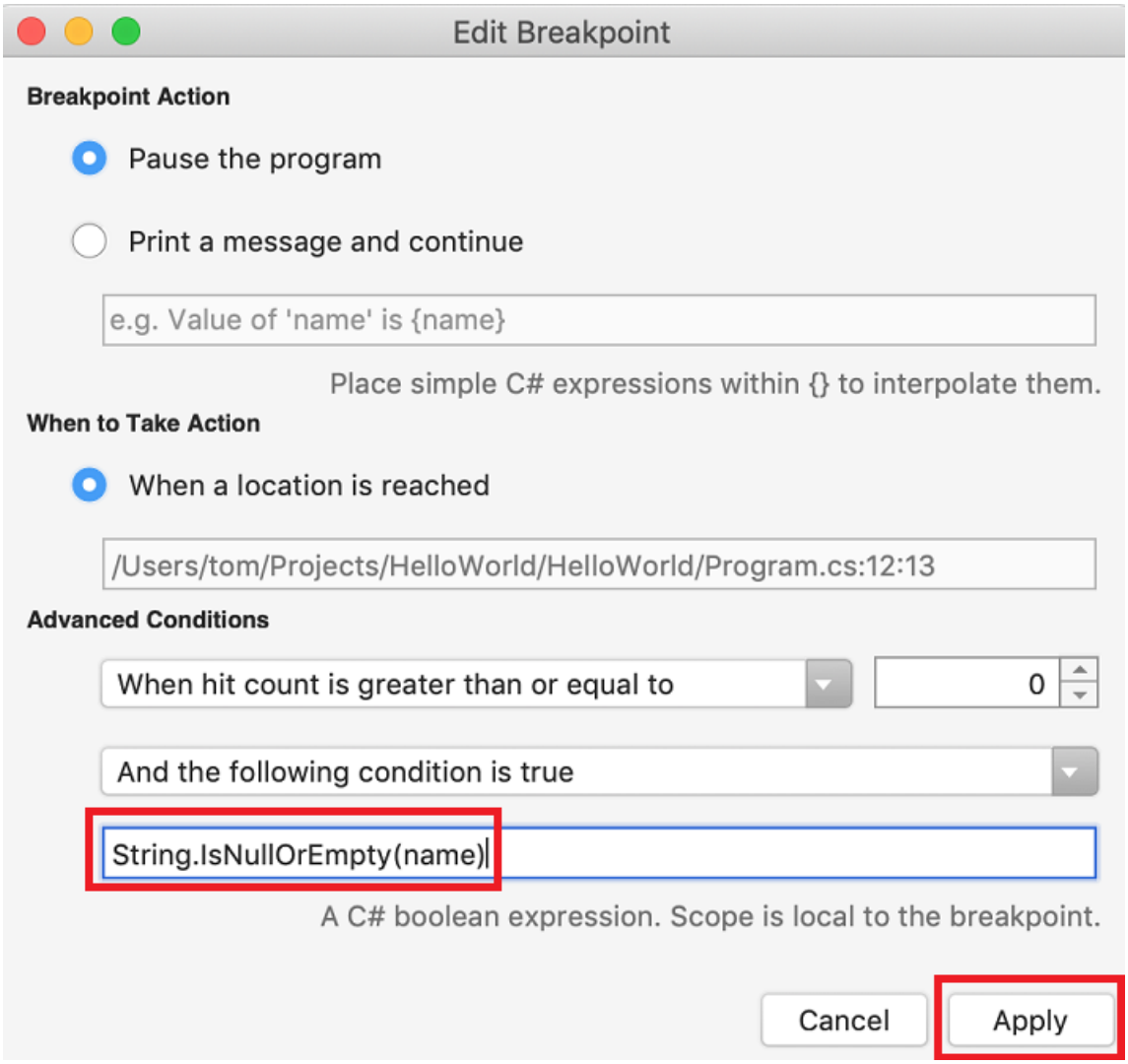
设置条件断点

程序显示用户输入的字符串。如果用户没有输入任何内容，情况又如何呢？ 可以使用名为"条件断点"的有用调试功能对此进行测试。

1. 按 Ctrl 并单击表示断点的红点。在上下文菜单中，选择"编辑断点"。

2. 在"编辑断点"对话框中，在遵循"且以下条件为 true"的字段中输入以下代码，然后选择"应用" 。

```
String.IsNullOrEmpty(name)
```



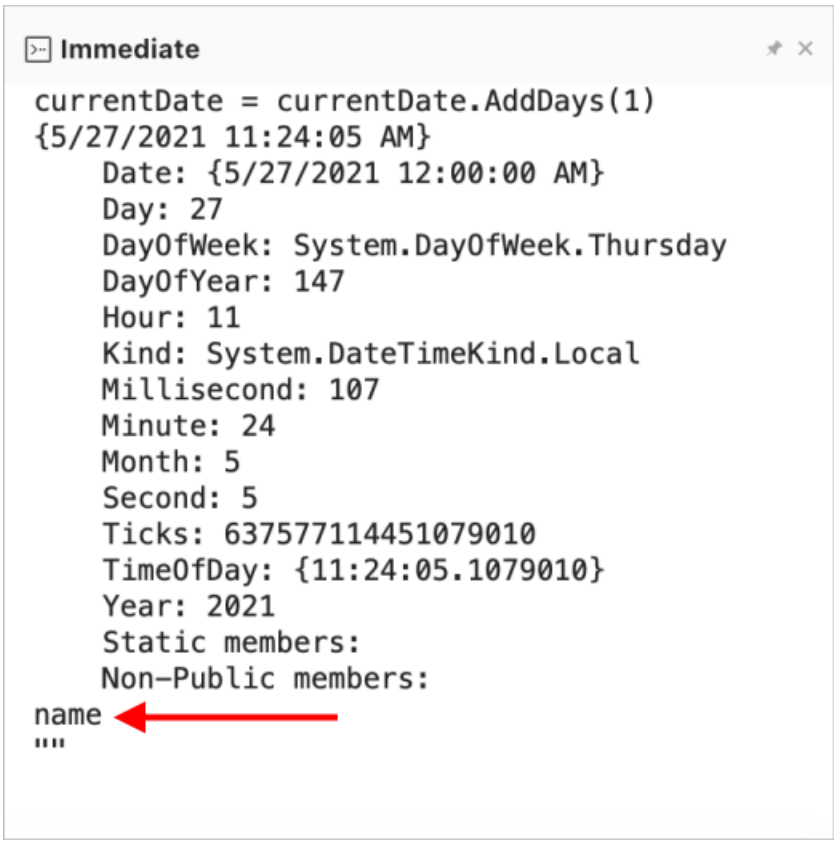
每次命中断点时，调试器都会调用 `String.IsNullOrEmpty(name)` 方法，仅当该方法调用返回 `true` 时，它才会在此行上中断。

可以指定"命中次数"（而不是条件表达式），这样程序就会在语句的执行次数达到指定值时中断执行。

3. 按 ⌘↵ (command+enter) 以启动调试。
4. 在终端窗口中，在系统提示输入姓名时按 `Enter`。

   由于符合指定的条件（`name` 为 `null` 或 String.Empty），因此程序会在到达断点时停止执行。

5. 选择"局部变量"窗口，其中显示当前正在执行的方法的局部变量值。在这种情况下，`Main` 是当前正在执行的方法。请注意，`name` 变量的值为 `""`，即 String.Empty。
6. 还可以通过在"即时"窗口中输入 `name` 变量名称并按 `Enter` 来看到值为空字符串。



7. 按 ⌘↵ (command+enter) 以继续调试。
8. 在终端窗口中，按任意键退出程序。
9. 关闭终端窗口。
10. 单击代码窗口左边缘上的红点，清除断点。清除断点的另一种方法是在选中代码行时选择"运行">"切换断点"。

单步执行程序

使用 Visual Studio，还可以单步执行程序，并监视其执行情况。通常可以设置断点，并通过程序代码的一小部分执行程序流。由于此程序很小，因此可以单步执行整个程序。

1. 在标记 `Main` 方法的开头的大括号处设置断点（按 command+\）。
2. 按 ⌘↵ (command+enter) 以启动调试。

   Visual Studio 在包含断点的行上停止。
3. 按 ⇧⌘I (shift+command+I) 或选择"运行" > "单步执行"以推进一行。

   Visual Studio 会在要执行的下一行旁边突出显示一个箭头。

此时，“局部变量”窗口显示 `args` 数组为空，`name` 和 `currentDate` 具有默认值。此外，Visual Studio 还打开了一个空白终端。

4. 按 ⇧⌘I (`shift+command+I`)。

   Visual Studio 突出显示包含 `name` 变量赋值的语句。“局部变量”窗口显示 `name` 为 `null`，终端显示字符串“What is your name?”。

5. 在控制台窗口中输入字符串，然后按 `Enter`，从而响应提示。

6. 按 ⇧⌘I (`shift+command+I`)。

   Visual Studio 突出显示包含 `currentDate` 变量赋值的语句。“局部变量”窗口显示 Console.ReadLine 方法调用返回的值。终端显示在提示符处输入的字符串。

7. 按 ⇧⌘I (`shift+command+I`)。

   “局部变量”窗口显示通过 DateTime.Now 属性赋值后的 `currentDate` 变量值。终端无变化。

8. 按 ⇧⌘I (`shift+command+I`)。

   Visual Studio 调用 Console.WriteLine(String, Object, Object) 方法。终端会显示格式化的字符串。

9. 按 ⇧⌘U (`shift+command+U`) 或选择“运行” > “单步跳出”。

   终端会显示一条消息，并等待用户按任意键。

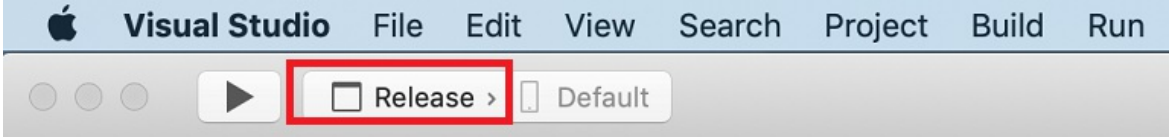
10. 按任意键退出程序。

使用“发布”生成配置

测试应用程序的“调试”版本后，还应该编译并测试“发布”版本。发布版本包含编译器优化，可能会对应用程序的行为产生不良影响。例如，旨在提升性能的编译器优化可能会在多线程应用程序中创建争用条件。

若要生成和测试控制台应用程序的发布版本，请执行以下步骤：

1. 将工具栏上的生成配置从“调试”更改为“发布”。



2. 按 ⌥⌘↵ (`option+command+enter`) 以运行但不调试。

后续步骤

在本教程中，你使用了 Visual Studio 调试工具。在下一教程中，你将发布应用的可部署版本。

使用 Visual Studio for Mac 发布 .NET 控制台应用程序

教程：使用 Visual Studio for Mac 发布 .NET 控制台应用程序

2021/11/16 •

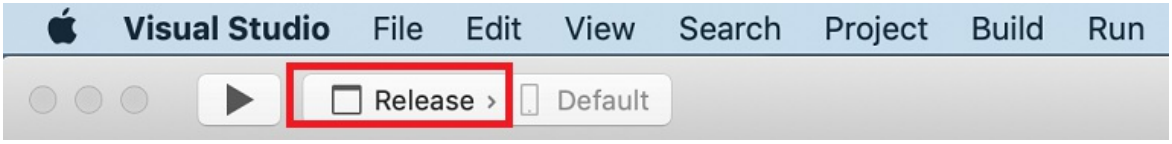
本教程演示如何发布控制台应用，以便其他用户可以运行它。发布应用程序会创建运行应用程序所需的一组文件。若要部署文件，请将文件复制到目标计算机。

先决条件

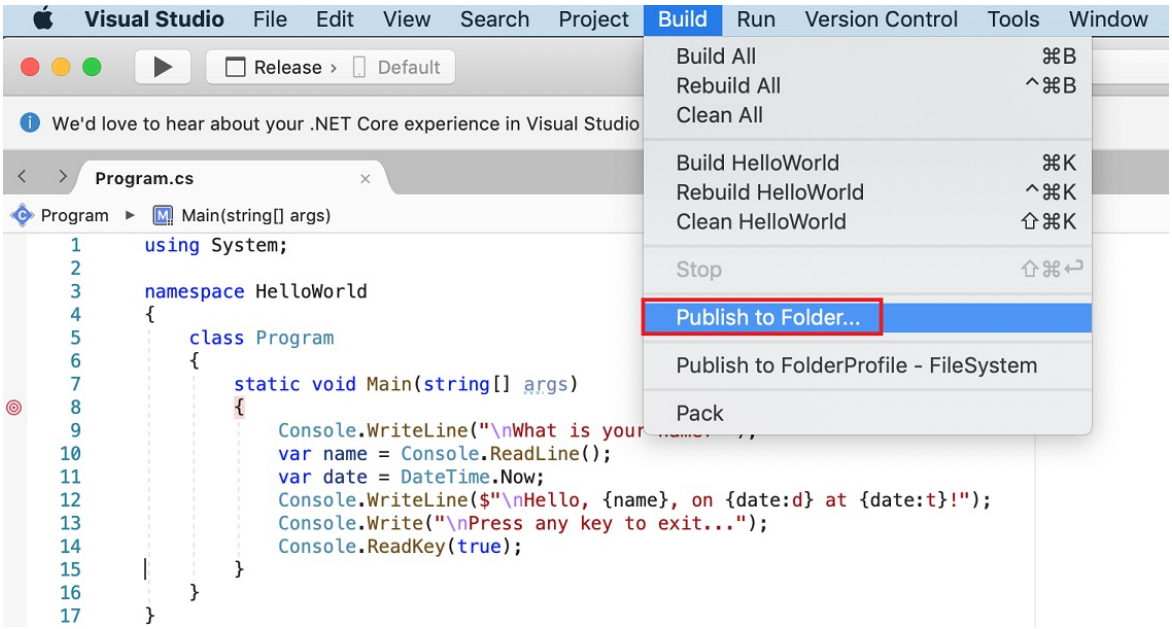
- 本教程适用于在使用 Visual Studio for Mac 创建 .NET 控制台应用程序中创建的控制台应用。

发布应用

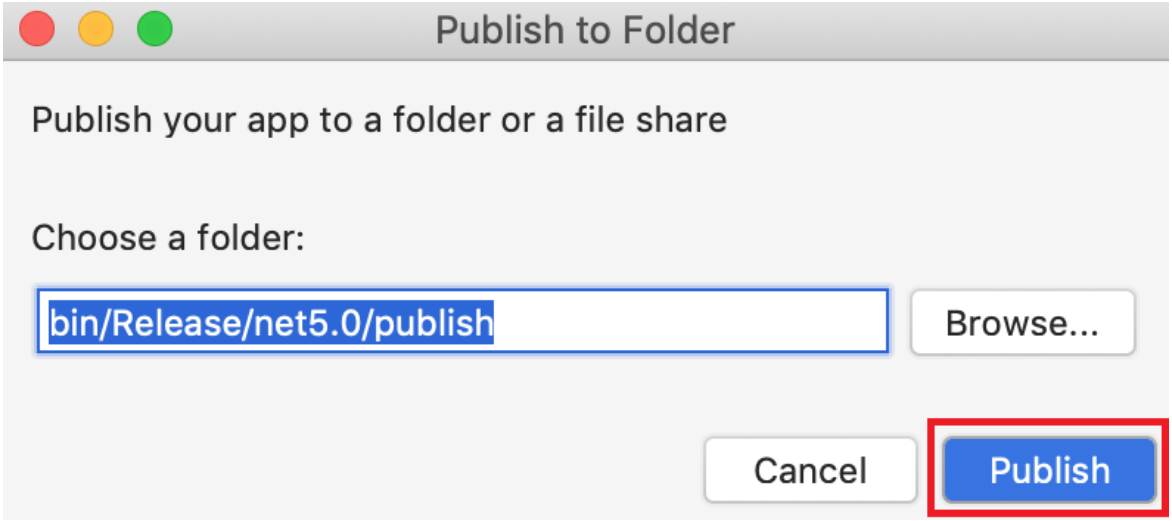
1. 启动 Visual Studio for Mac。
2. 打开在使用 Visual Studio for Mac 创建 .NET 控制台应用程序中创建的 HelloWorld 项目。
3. 请确保 Visual Studio 生成的是应用程序的发布版本。必要时，将工具栏上的生成配置设置从“调试”更改为“发布”。



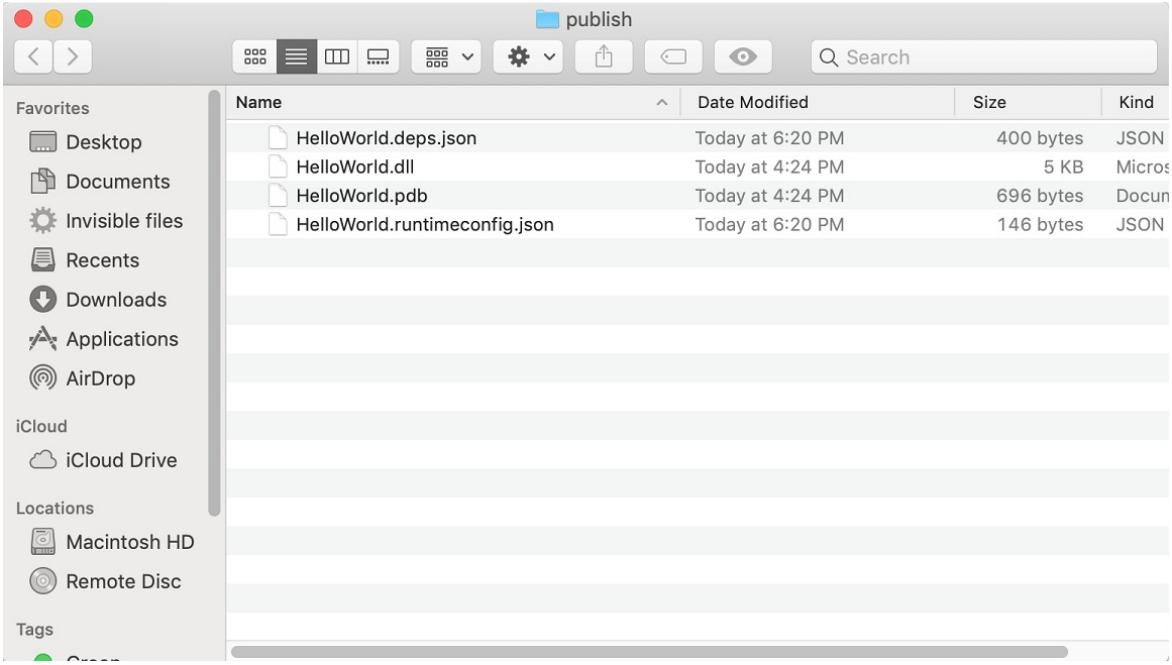
4. 从主菜单中，选择“生成” > “发布到文件夹...”。



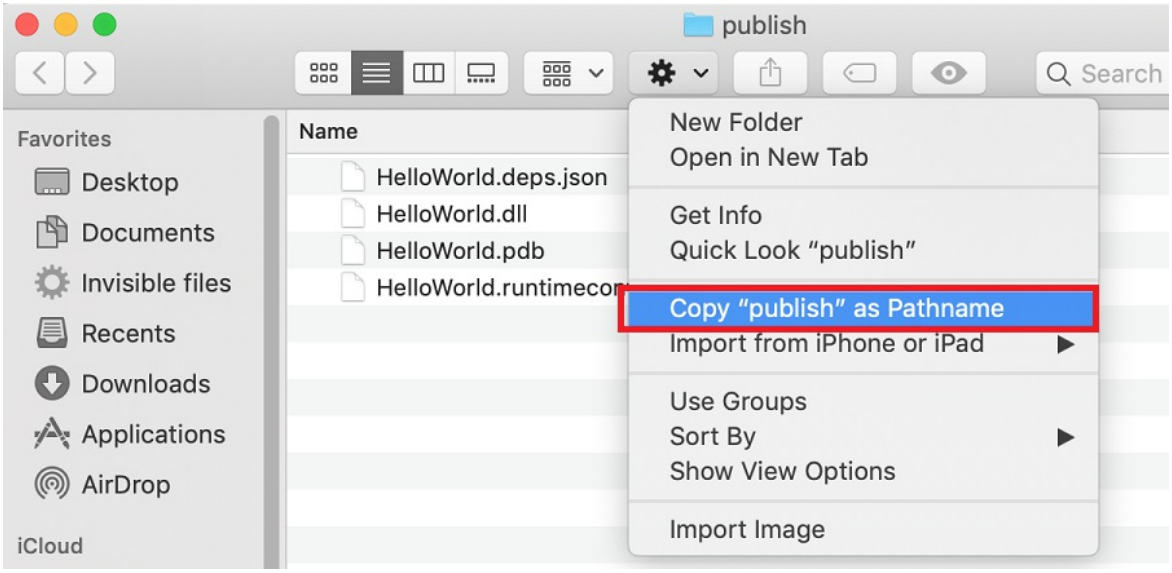
5. 在“发布到文件夹”对话框中，选择“发布” 。



此时会打开发布文件夹，其中显示已创建的文件。



6. 选择齿轮图标，并从上下文菜单中选择“将“发布”复制为路径名称”。



检查文件

发布过程中会创建依赖于框架的部署，在此类部署中，已发布的应用程序在已安装 .NET 运行时的计算机上运

行。用户可以通过从命令提示符运行 `dotnet HelloWorld.dll` 命令来运行发布的应用。

如上图所示，已发布的输出包括以下文件：

- HelloWorld.deps.json

  这是应用程序的运行时依赖项文件。该文件定义了运行应用所需的 .NET 组件和库（包括包含应用程序的动态链接库）。有关详细信息，请参阅运行时配置文件。

- HelloWorld.dll

  这是应用程序的依赖于框架的部署版本。若要执行此动态链接库，请在命令提示符处输入 `dotnet HelloWorld.dll`。这种运行应用的方法适用于安装了 .NET 运行时的任何平台。

- HelloWorld.pdb（对于部署是可选的）

  这是调试符号文件。尽管应在需要调试应用程序的已发布版本时保存此文件，但无需将此文件与应用程序一起部署。

- HelloWorld.runtimeconfig.json

  这是应用程序的运行时配置文件。该文件标识用于运行应用程序的 .NET 版本。还可向其添加配置选项。有关详细信息，请参阅 .NET 运行时配置设置。

运行已发布的应用

1. 打开终端并导航到发布文件夹。为此，请输入 `cd`，然后粘贴前面复制的路径。例如：

   ```
   cd ~/Projects/HelloWorld/HelloWorld/bin/Release/net5.0/publish/
   ```

2. 使用 `dotnet` 命令运行应用：

   a. 输入 `dotnet HelloWorld.dll`，然后按 Enter。

   b. 输入一个名字以响应提示，并按任意键退出。

其他资源

- .NET 应用程序部署

后续步骤

在本教程中，你发布了一个控制台应用。在下一教程中，你将创建类库。

使用 Visual Studio for Mac 创建 .NET 库

教程：使用 Visual Studio for Mac 创建 .NET 类库

2021/11/16 •

在本教程中，将创建包含一个字符串处理方法的类库。

类库定义的是可以由应用程序调用的类型和方法。如果库以 .NET Standard 2.0 为目标，则支持 .NET Standard 2.0 的任何 .NET 实现（包括 .NET Framework）均可调用该库。如果库以 .NET 5 为目标，则以 .NET 5 为目标的任何应用程序均可调用该库。本教程演示如何以 .NET 5 为目标。

> **NOTE**
> 你的反馈非常有价值。有两种方法可以向开发团队提供有关 Visual Studio for Mac 的反馈：
>
> - 在 Visual Studio for Mac 中，从菜单选择“帮助” > “报告问题”，或从欢迎屏幕中选择“报告问题”，将打开一个窗口，以供填写 bug 报告。可在开发人员社区门户中跟踪自己的反馈。
> - 若要提出建议，从菜单中选择“帮助” > “提供建议”，或从欢迎屏幕中选择“提供建议”，转到 Visual Studio for Mac 开发人员社区网页。

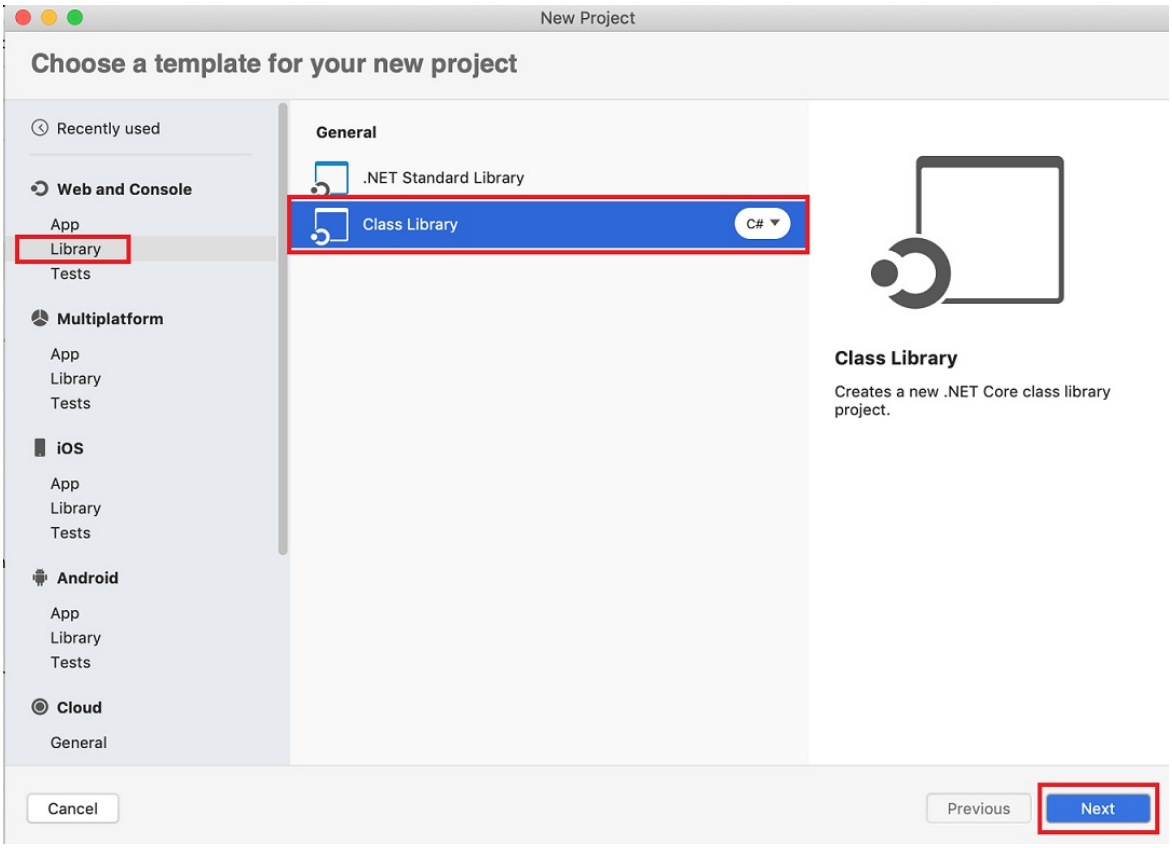
先决条件

- 安装 Visual Studio for Mac 版本 8.8 或更高版本。选择用于安装 .NET Core 的选项。安装 Xamarin 对于 .NET 开发而言是可选项。有关更多信息，请参见以下资源：
  - 教程：安装 Visual Studio for Mac。
  - 支持的 macOS 版本。
  - Visual Studio for Mac 支持的 .NET 版本。

创建包含类库项目的解决方案

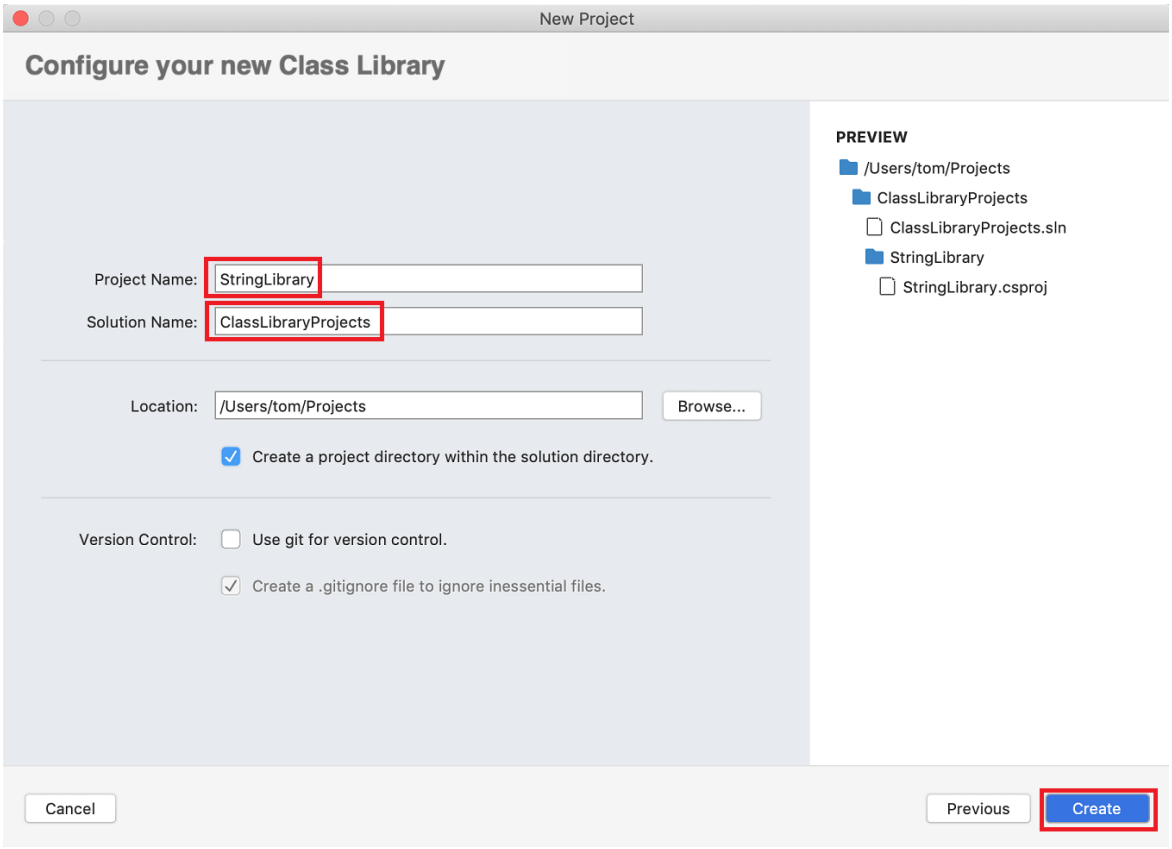
Visual Studio 解决方案用作一个或多个项目的容器。创建解决方案和解决方案中的类库项目。稍后将其他相关项目添加到同一个解决方案中。

1. 启动 Visual Studio for Mac。
2. 在“开始”窗口中，选择“新建项目”。
3. 在“为新项目选择一个模板”对话框中选择“Web 和控制台” > “库” > “类库”然后选择“下一步”。

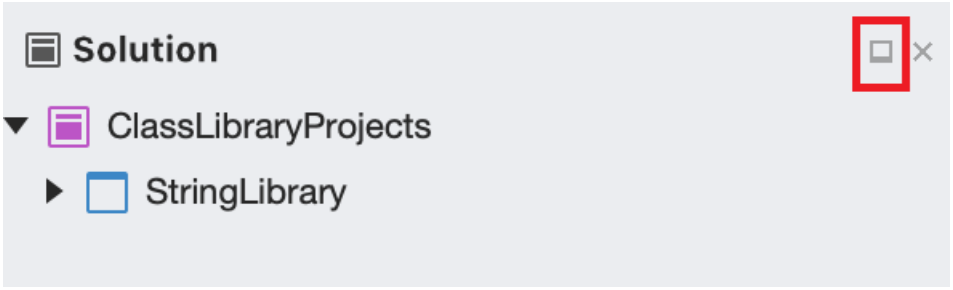


4. 在"配置新的类库"对话框中，选择".NET 5.0"，然后选择"下一步"。

5. 将项目命名为"StringLibrary"，并将解决方案命名为"ClassLibraryProjects"。使"在解决方案目录中创建项目目录"保持选中状态。选择"创建"。



6. 从主菜单中，选择"视图" > "解决方案"，然后选择"停靠"图标使边栏保持打开状态。

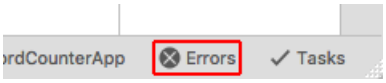


7. 在“解决方案”边栏中，展开 `StringLibrary` 节点以显示模板提供的类文件 *Class1.cs*。按住 Ctrl 并单击该文件，从上下文菜单中选择“重命名”，然后将该文件重命名为“StringLibrary.cs”。打开文件并将内容替换为以下代码：

```
using System;

namespace UtilityLibraries
{
    public static class StringLibrary
    {
        public static bool StartsWithUpper(this string str)
        {
            if (string.IsNullOrWhiteSpace(str))
                return false;

            char ch = str[0];
            return char.IsUpper(ch);
        }
    }
}
```

8. 按 ⌘S (command+S) 以保存文件。

9. 在 IDE 窗口底部边距处选择“错误”，打开“错误”面板。选择“生成输出”按钮。



10. 从菜单中选择“生成” > “生成所有”。

    生成解决方案。生成输出面板显示生成成功。

```
Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.50

========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========

Build successful.
```

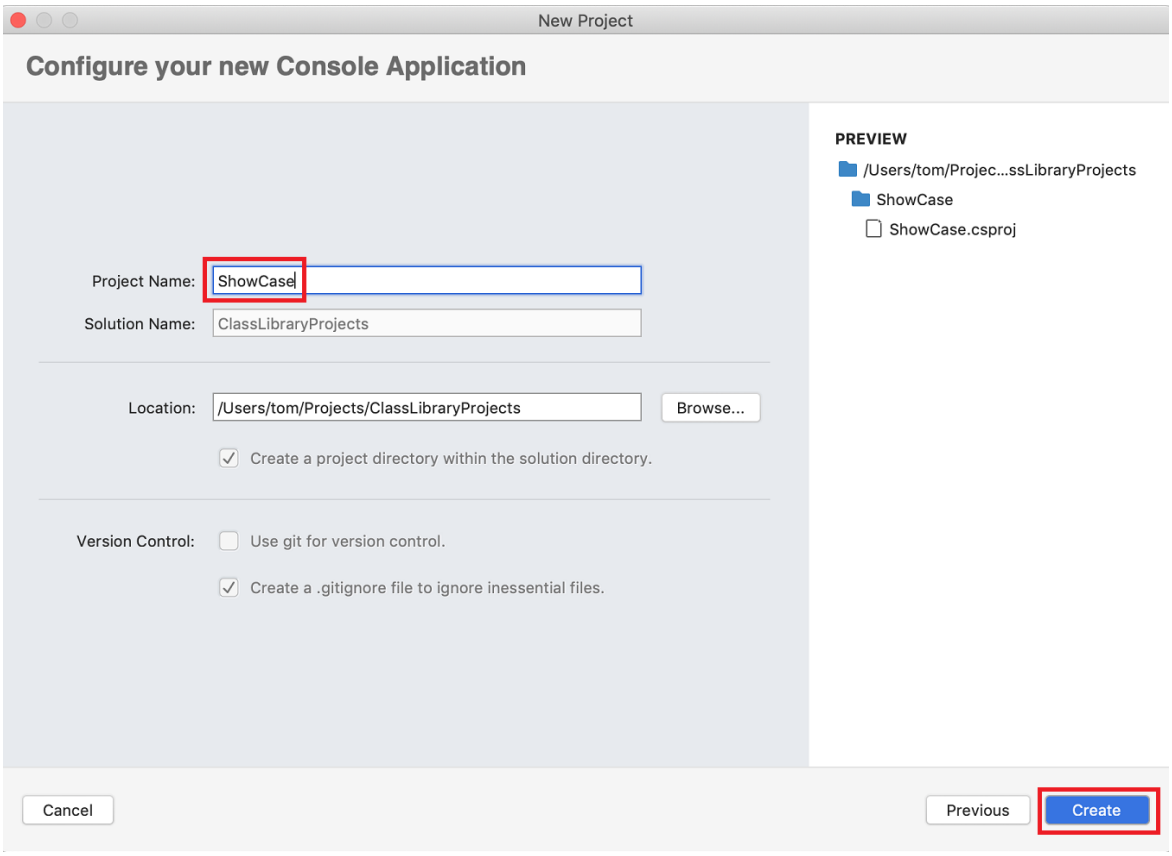
向解决方案添加控制台应用

添加使用类库的控制台应用程序。应用将提示用户输入字符串，并报告字符串是否以大写字符开头。

1. 在“解决方案”边栏中，按住 Ctrl 并单击 `ClassLibraryProjects` 解决方案。通过从“Web 和控制台” > “应用”模板中选择模板来添加新的“控制台应用程序”项目，然后选择“下一步”。

2. 选择“.NET Core 5.0”作为“目标框架”，然后选择“下一步”。

3. 将项目命名为“ShowCase”。选择“创建”以在解决方案中创建项目。



4. 打开 *Program.cs* 文件。将代码替换为以下代码：

```
using System;
using UtilityLibraries;

class Program
{
    static void Main(string[] args)
    {
        int row = 0;

        do
        {
            if (row == 0 || row >= 25)
                ResetConsole();

            string input = Console.ReadLine();
            if (string.IsNullOrEmpty(input)) break;
            Console.WriteLine($"Input: {input} {"Begins with uppercase? ",30}: " +
                              $"{(input.StartsWithUpper() ? "Yes" : "No")}{Environment.NewLine}");
            row += 3;
        } while (true);
        return;

        // Declare a ResetConsole local method
        void ResetConsole()
        {
            if (row > 0)
            {
                Console.WriteLine("Press any key to continue...");
                Console.ReadKey();
            }
            Console.Clear();
            Console.WriteLine($"{Environment.NewLine}Press <Enter> only to exit; otherwise, enter a
string and press <Enter>:{Environment.NewLine}");
            row = 3;
        }
    }
}
```

该程序会提示用户输入字符串。它会指明字符串是否以大写字符开头。如果用户没有输入字符串就按 Enter 键，那么应用程序会终止，控制台窗口会关闭。

该代码使用 `row` 变量来维护写入到控制台窗口的数据行数计数。如果大于或等于 25，该代码将清除控制台窗口，并向用户显示一条消息。

添加项目引用

最初，新的控制台应用项目无权访问类库。若要允许该项目调用类库中的方法，可以创建对类库项目的项目引用。

1. 在“解决方案”边栏中，按住 Ctrl 并单击新“ShowCase”项目的“依赖项”节点。在上下文菜单中，选择“添加引用”。
2. 在“引用”对话框中，选择“StringLibrary”，然后选择“确定”。

运行应用

1. 按住 Ctrl 并单击 ShowCase 项目，然后从上下文菜单中选择“运行项目”。
2. 输入字符串并按 Enter 以试用程序，然后按 Enter 退出。

```
Terminal – ShowCase

Press <Enter> only to exit; otherwise, enter a string and press <Enter>:

Begins with uppercase
Input: Begins with uppercase        Begins with uppercase? : Yes

begins with lowercase
Input: begins with lowercase        Begins with uppercase? : No
```

其他资源

- 使用 .NET CLI 开发库
- Visual Studio 2019 for Mac 发行说明
- .NET Standard 版本及其支持的平台。

后续步骤

在本教程中，你创建了一个解决方案和一个库项目，并添加了一个使用该库的控制台应用项目。在下一教程中，将向解决方案中添加单元测试项目。

使用 Visual Studio for Mac 测试 .NET 类库

使用 Visual Studio 测试 .NET 类库

2021/11/16 •

本教程演示如何通过将测试项目添加到解决方案来自动执行单元测试。

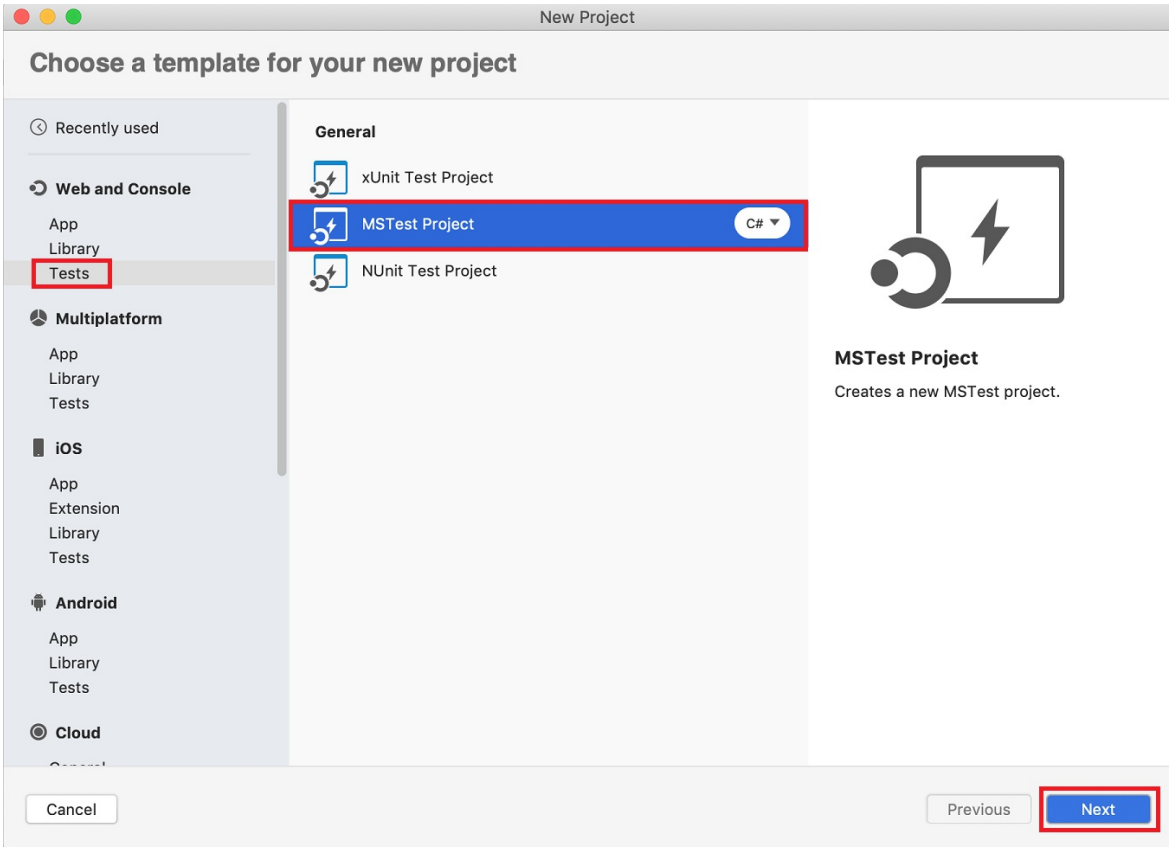
先决条件

- 本教程适用于在使用 Visual Studio for Mac 创建 .NET 类库中创建的解决方案。

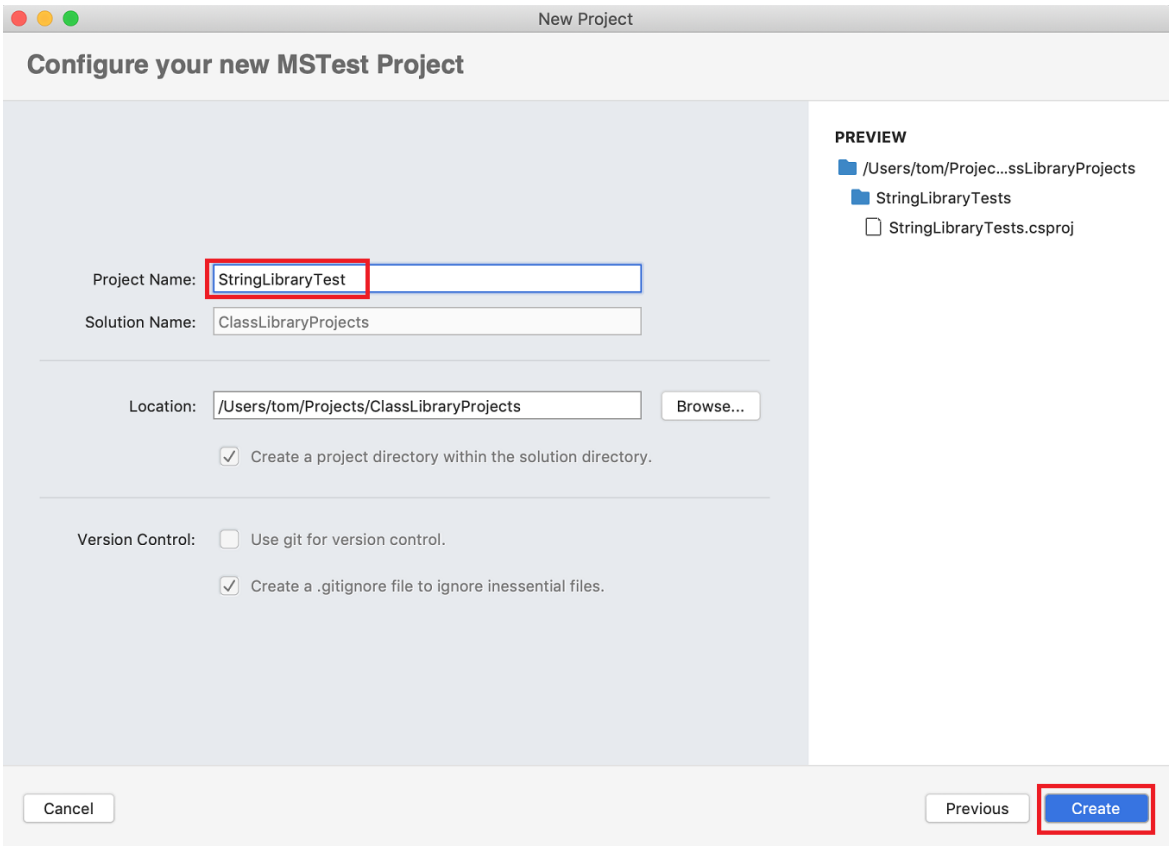
创建单元测试项目

单元测试在开发和发布期间提供自动化的软件测试。MSTest 是可供选择的三个测试框架之一。其他两个是 xUnit 和 nUnit。

1. 启动 Visual Studio for Mac。
2. 打开在使用 Visual Studio for Mac 创建 .NET 类库中创建的 `ClassLibraryProjects` 解决方案。
3. 在“解决方案”边栏中，按住 Ctrl 并单击 `ClassLibraryProjects` 解决方案，然后选择“添加” > “新建项目”。
4. 在“新建项目”对话框中，选择“Web 和控制台”节点中的“测试”。依次选择“MSTest 项目”、“下一步”。



5. 选择“.NET Core 5.0”作为“目标框架”，然后选择“下一步”。
6. 将新项目命名为"StringLibraryTest"，然后选择“创建”。



Visual Studio 使用以下代码创建类文件：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

单元测试模板创建的源代码负责执行以下操作：

- 它会导入 Microsoft.VisualStudio.TestTools.UnitTesting 命名空间，其中包含用于单元测试的类型。
- 向 `UnitTest1` 类应用 TestClassAttribute 特性。
- 它向 `TestMethod1` 应用 TestMethodAttribute 特性。

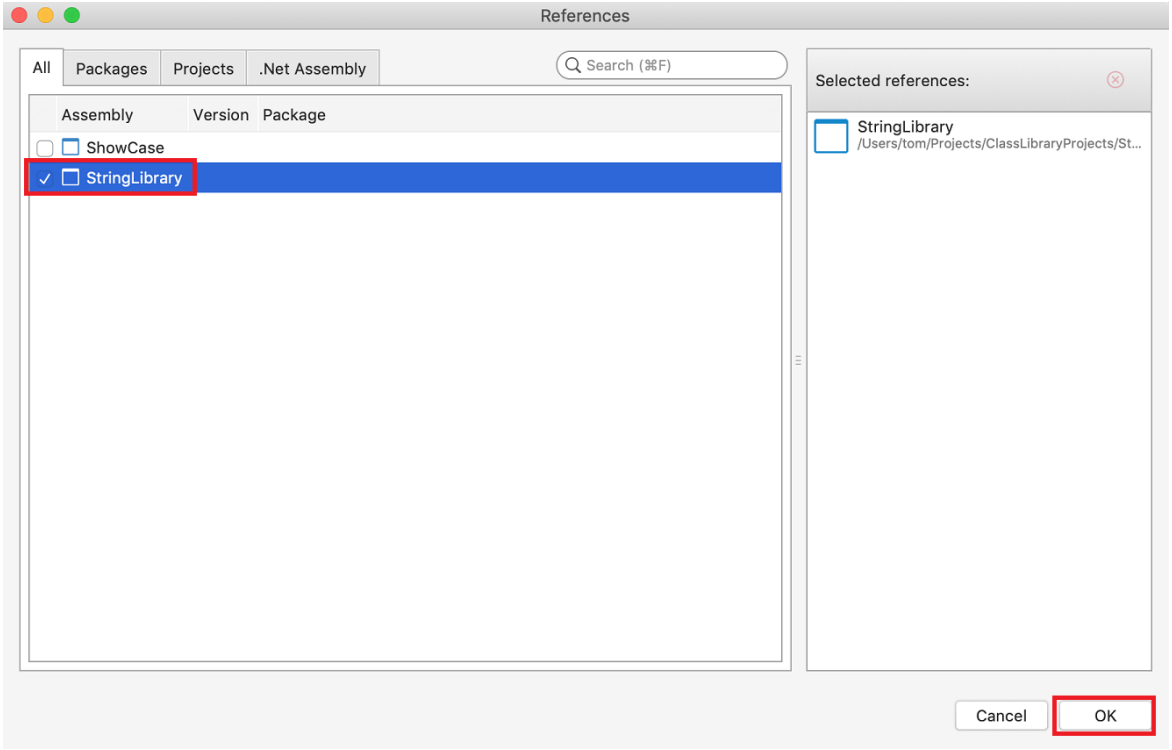
使用 [TestClass] 标记的测试类中标记有 [TestMethod] 的所有测试方法都会在单元测试运行时自动执行。

添加项目引用

对于要使用 `StringLibrary` 类的测试库，请将引用添加到 `StringLibrary` 项目中。

1. 在“解决方案”边栏中，按住 Ctrl 并单击“StringLibraryTest”下的“依赖项”。从上下文菜单中选择“添加引用”。

2. 在“引用”对话框中，选择“StringLibrary”项目。选择“确定”。



添加并运行单元测试方法

运行单元测试时，Visual Studio 执行使用 TestClassAttribute 特性标记的类中标记有 TestMethodAttribute 特性的所有方法。当第一次遇到测试不通过或测试方法中的所有测试均已成功通过时，测试方法终止。

最常见的测试调用 Assert 类的成员。许多断言方法至少包含两个参数，其中一个是预期的测试结果，另一个是实际的测试结果。下表显示了 `Assert` 类最常调用的一些方法：

`Assert.AreEqual`	验证两个值或对象是否相等。如果值或对象不相等，则断言失败。
`Assert.AreSame`	验证两个对象变量引用的是否是同一个对象。如果这些变量引用不同的对象，则断言失败。
`Assert.IsFalse`	验证条件是否为 `false`。如果条件为 `true`，则断言失败。
`Assert.IsNotNull`	验证对象是否不为 `null`。如果对象为 `null`，则断言失败。

还可以在测试方法中使用 Assert.ThrowsException 方法来指示它应引发的异常的类型。如果未引发指定异常，则测试不通过。

测试 `StringLibrary.StartsWithUpper` 方法时，需要提供许多以大写字符开头的字符串。在这种情况下，此方法应返回 `true`，以便可以调用 Assert.IsTrue 方法。同样，需要提供许多以非大写字符开头的字符串。在这种情况下，此方法应返回 `false`，以便可以调用 Assert.IsFalse 方法。

由于库方法处理的是字符串，因此还需要确保它能够成功处理空字符串 (`String.Empty`)(不含字符且 Length 为 0 的有效字符串)和 `null` 字符串(尚未初始化的字符串)。可以直接将 `StartsWithUpper` 作为静态方法进行调用，并向其传递一个 String 自变量。或者，可以对分配给 `null` 的 `string` 变量将 `StartsWithUpper` 作为扩展方法进行调用。

将定义三个方法，每个方法都会对字符串数组中的各个元素调用它的 Assert 方法。你将调用方法重载，以便指定在测试失败时要显示的错误消息。消息标识导致失败的字符串。

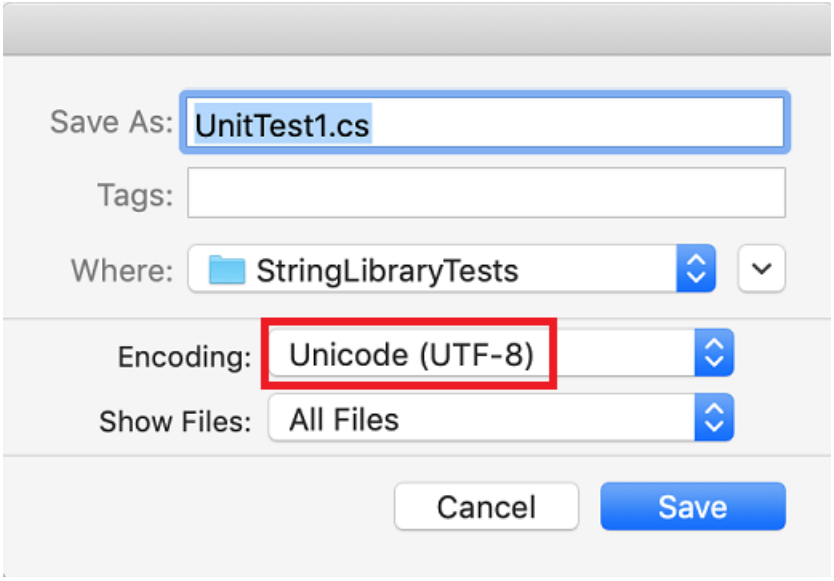
创建测试方法：

1. 打开 UnitTest1.cs 文件，并将代码替换为以下代码：

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using UtilityLibraries;

namespace StringLibraryTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestStartsWithUpper()
        {
            // Tests that we expect to return true.
            string[] words = { "Alphabet", "Zebra", "ABC", "Αθήνα", "Москва" };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsTrue(result,
                       String.Format("Expected for '{0}': true; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void TestDoesNotStartWithUpper()
        {
            // Tests that we expect to return false.
            string[] words = { "alphabet", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                               "1234", ".", ";", " " };
            foreach (var word in words)
            {
                bool result = word.StartsWithUpper();
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word, result));
            }
        }

        [TestMethod]
        public void DirectCallWithNullOrEmpty()
        {
            // Tests that we expect to return false.
            string[] words = { string.Empty, null };
            foreach (var word in words)
            {
                bool result = StringLibrary.StartsWithUpper(word);
                Assert.IsFalse(result,
                       String.Format("Expected for '{0}': false; Actual: {1}",
                                     word == null ? "<null>" : word, result));
            }
        }
    }
}
```

`TestStartsWithUpper` 方法中的大写字符的测试包括希腊文大写字母 alpha (U+0391) 和西里尔文大写字母 EM (U+041C)。`TestDoesNotStartWithUpper` 方法中的小写字符的测试包括希腊文小写字母 alpha (U+03B1) 和西里尔文小写字母 Ghe (U+0433)。

2. 在菜单栏中，选择“文件” > “另存为” 。在对话框中，确保“编码”设置为“Unicode (UTF-8)” 。

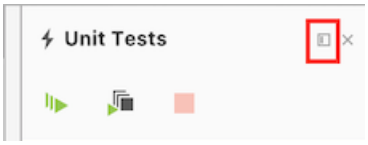


3. 当系统询问你是否要替换现有文件时，请选择“替换”。

   如果无法将源代码保存为 UTF8 编码文件，Visual Studio 可能会将其另存为 ASCII 文件。在这种情况下，运行时将无法准确解码 ASCII 范围以外的 UTF8 字符，且测试结果也会不正确。

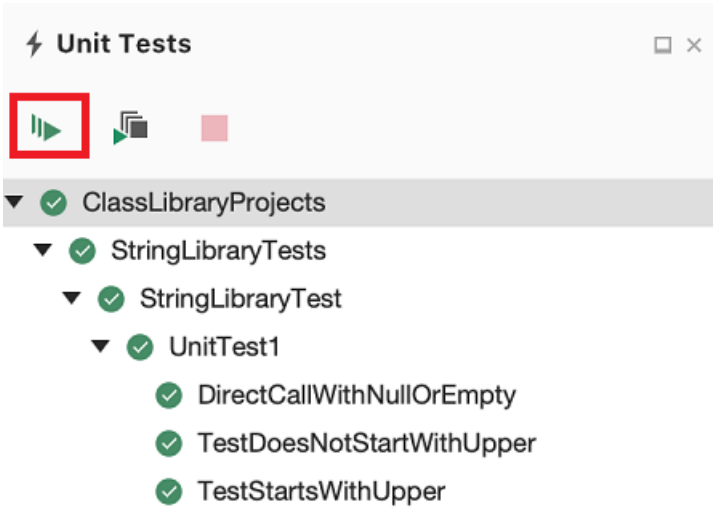
4. 打开屏幕右侧的“单元测试”面板。从菜单中选择“查看” > “测试”。

5. 单击“停靠”图标使此面板保持打开状态。



6. 单击“全部运行”按钮。

   所有测试通过。



处理测试失败

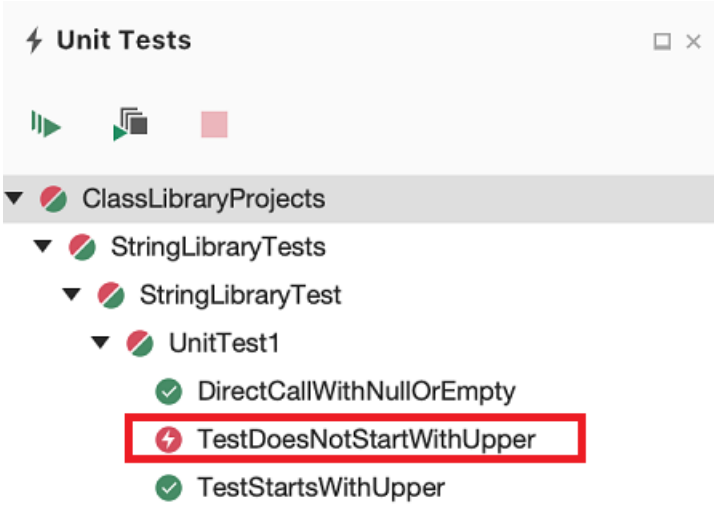
如果进行的是测试驱动开发 (TDD)，请先编写测试，然后测试会在第一次运行时失败。接着将可以使测试成功的代码添加到应用。在本教程中，先编写了测试要验证的应用代码然后才创建测试，所以没有看到测试失败。若要验证测试是否在预期失败时失败，请在测试输入中添加无效值。

1. 通过修改 `TestDoesNotStartWithUpper` 方法中的 `words` 数组来包含字符串“Error”。由于 Visual Studio 将在生成运行测试的解决方案时自动保存打开的文件，因此无需手动保存。

```
string[] words = { "alphabet", "Error", "zebra", "abc", "αυτοκινητοβιομηχανία", "государство",
                   "1234", ".", ";", " " };
```

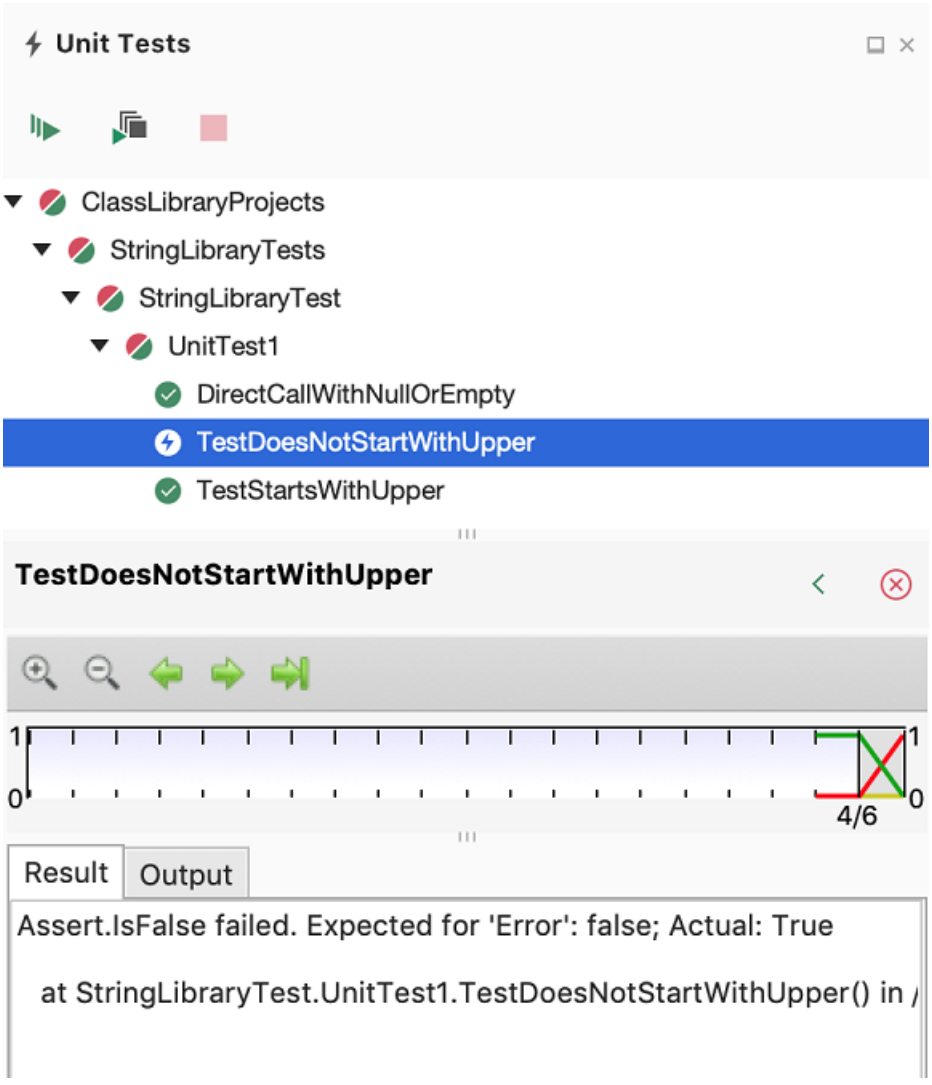
2. 重新运行测试。

   这一次，“测试资源管理器”窗口指示有两个测试成功，还有一个失败。



3. 按 Ctrl 并单击失败的测试 `TestDoesNotStartWithUpper`，然后从上下文菜单中选择“显示结果边栏”。

   “结果”边栏显示断言生成的消息：“Assert.IsFalse 失败。“Error”应返回 false；实际返回 True”。由于此次失败，数组中“Error”之后的所有字符串都未进行测试。



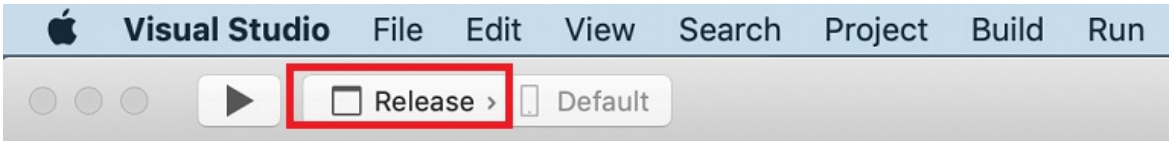
4. 删除在步骤 1 中添加的字符串"Error"。重新运行测试，测试将通过。

测试库的发行版本

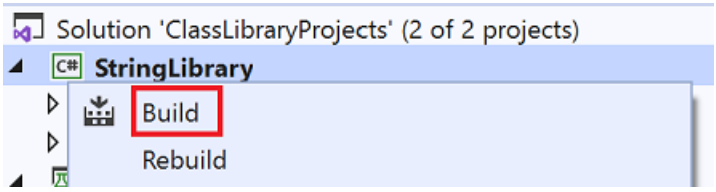
至此，在运行库的调试版本时，测试已全部通过，接下来应对库的发行版本再运行一次这些测试。许多因素（包括编译器优化）有时可能会导致调试版本和发行版本出现行为差异。

若要测试发行版本，请执行以下操作：

1. 在 Visual Studio 工具栏中，将生成配置从"调试"更改为"发行"。



2. 在"解决方案"边栏中，按 Ctrl 并单击"StringLibrary"项目，然后从上下文菜单中选择"生成"，重新编译库。



3. 再次运行单元测试。

   测试通过。

调试测试

如果使用 Visual Studio for Mac 作为 IDE，可以使用教程：使用 Visual Studio for Mac 调试 .NET 控制台应用程序中所示的相同过程，使用单元测试项目调试代码。按住 Ctrl 并单击"StringLibraryTests"项目，然后从上下文菜单中选择"启动调试项目"，而不是启动 ShowCase 应用项目。

Visual Studio 启动附有调试器的测试项目。执行将在添加到测试项目的任何断点或基础库代码处停止。

其他资源

- .NET 中的单元测试

后续步骤

在本教程中，你对类库进行了单元测试。你可以将库作为包发布到 NuGet，使其可供其他人使用。若要了解如何操作，请遵循 NuGet 教程：

创建并发布包 (dotnet CLI)

如果将库作为 NuGet 包发布，其他人可以安装并使用它。若要了解如何操作，请遵循 NuGet 教程：

在 Visual Studio for Mac 中安装和使用包

库并非必须作为包进行分发。它还可与使用它的控制台应用捆绑在一起。若要了解如何发布控制台应用，请参阅本系列中前面的教程：

使用 Visual Studio for Mac 发布 .NET 控制台应用程序

探讨这些教程，学习 .NET 和 .NET Core SDK 工具

2021/11/16 •

以下教程演示了如何为 .NET Core、.NET 5 及更高版本开发控制台应用和库。有关其他类型的应用程序，请参阅 .NET 入门教程。

使用 Visual Studio

- 创建控制台应用
- 调试应用
- 发布应用
- 创建类库
- 对类库进行单元测试
- 安装并使用包
- 创建和发布包
- 创建 F# 控制台应用

使用 Visual Studio Code

如果要使用 Visual Studio Code 或其他代码编辑器，请选择这些教程。全都使用 CLI 来处理 .NET Core 开发任务，因此全都可用于任意代码编辑器（调试教程除外）。

- 创建控制台应用
- 调试应用
- 发布应用
- 创建类库
- 对类库进行单元测试
- 安装并使用包
- 创建和发布包
- 创建 F# 控制台应用

使用 Visual Studio for Mac

- 创建控制台应用
- 调试应用
- 发布应用
- 创建类库
- 对类库进行单元测试
- 安装并使用包
- 创建 F# 控制台应用

高级主题

- 如何创建库
- 使用 xUnit 对应用进行单元测试
- 结合使用 C#/VB/F# 和 NUnit/xUnit/MSTest 进行单元测试
- 使用 Visual Studio 进行实时单元测试

- 创建 CLI 的模板
- 创建和使用适用于 CLI 的工具
- 使用插件创建应用

.NET 6 中的新增功能

2021/11/16 •

.NET 6 提供从 .NET 5开始的 .NET 统一计划的最后部分。.NET 6 跨移动、桌面、IoT 和云应用统一了 SDK、基本库和运行时。除了这种统一，.NET 6 生态系统还提供：

- **简化的开发**：入门很简单。C# 10中的新语言功能减少了需要编写的代码量。通过 Web 堆栈和最小 API 方面的投资，可以轻松快速编写更小、速度更快的微服务。
- **更好的性能**：.NET 6 是最快的完整堆栈 Web 框架，如果是在云中运行，则它可以降低计算成本。
- **终极工作效率**：.NET 6 和 Visual Studio 2022提供热重载、新的 git 工具、智能代码编辑、可靠的诊断和测试工具，以及更好的团队协作。

.NET 6 将支持 三年，作为 LTS (长期) 支持。

预览 功能默认处于禁用状态。它们也不支持在生产环境中使用，并且可能会在将来的版本中删除。新的 用于批注预览 API，如果使用这些预览 API，则相应的 RequiresPreviewFeaturesAttribute 分析器会发出警报。

.NET 6 受 Visual Studio 2022 和 Visual Studio 2022 for Mac (及更高版本) 。

本文未介绍 .NET 6 的所有新功能。若要查看所有新功能，以及有关本文中列出的功能的进一步信息，请参阅宣布 推出 .NET 6 博客文章。

性能

.NET 6 包括许多性能改进。本部分列出了一些改进。有关详细信息，请参阅 .NET 6 中的 性能改进博客文章。

FileStream

System.IO.FileStream为 .NET 6 重写了 类型，以提供更好的性能和可靠性，Windows。现在，FileStream 为异步 I/O 创建时，切勿阻止Windows。有关详细信息，请参阅 .NET 6 中的文件 IO 改进博客文章。

按配置优化

PGO (按) 优化是 JIT 编译器根据最常用的类型和代码路径生成优化代码的地方。.NET 6 引入了 *动态* PGO。动态 PGO 与分层编译一起工作，以基于第 0 层期间放置的其他检测进一步优化代码。动态 PGO 默认处于禁用状态，但可以使用环境变量 `DOTNET_TieredPGO` 启用它。有关详细信息，请参阅 JIT 性能改进。

Crossgen2

.NET 6 引入了 Crossgen2（Crossgen 的后继者，已删除）。Crossgen 和 Crossgen2 是一些工具，可提前 (AOT) 以改进应用的启动时间。Crossgen2 是使用 C# 而不是 C++ 编写的，可以执行先前版本中无法进行分析和优化。有关详细信息，请参阅关于 Crossgen2 的对话。

Arm64 支持

.NET 6 版本包括对本机 Arm64 执行和 x64 仿真的 macOS Arm64 (或"Apple Silicon") 和 Windows Arm64 操作系统的支持。此外，x64 和 Arm64 .NET 安装程序现在并行安装。有关详细信息，请参阅适用于arm64 和 x64 的 macOS 11 Windows 11 .NET 支持。

热重载

热 重载是一项功能，可用于修改应用的源代码，并立即将这些更改应用到正在运行的应用。该功能的目的是避免在编辑之间重启应用，从而提高工作效率。2022 Visual Studio命令行工具中提供了 `dotnet watch` 热重载。热重载适用于大多数类型的 .NET 应用，适用于 C#、Visual Basic 和 C++ 源代码。有关详细信息，请参阅热重载 博

客文章。

.NET MAUI

.NET 多平台应用 UI (.NET MAUI) 仍处于预览状态，候选发布于 2022 年第一季度推出，2022 年第二季度发布 (GA) 。.NET MAUI一个代码库，为桌面和移动操作系统生成本机客户端应用。有关详细信息，请参阅 .NET 多 平台应用 UI 上的更新博客文章。

C# 10 和模板

C# 10 包括指令、文件范围命名空间声明和 `global using` 记录结构等创新。有关详细信息，请参阅 C# 10 中的新增功能。

随着该工作，适用于 C# 的 .NET SDK 项目模板已现代化，以使用一些新的语言功能：

- `async Main` 方法
- 顶级语句
- 目标类型的新表达式
- 隐式 `global using` 指令
- 文件范围的命名空间
- 可为空引用类型

通过将这些新语言功能添加到项目模板，新代码从启用的功能开始。但是，升级到 .NET 6 时，现有代码不受影响。有关这些模板更改的信息，请参阅 .NET SDK：C# 项目模板现代化 博客文章。

F# 和 Visual Basic

F# 6 对 F# 语言和语言F# 交互窗口。有关详细信息，请参阅 F# 6 中的新增功能。

Visual Basic窗体项目启动Visual Studio体验Windows改进。

SDK 工作负载

为了减小 .NET SDK 的大小，某些组件已放置在新的可选 *SDK 工作负荷中*。这些组件包括 .NET MAUI 和 Blazor WebAssembly AOT。如果使用 Visual Studio，它将负责安装所需的任何 SDK 工作负载。如果使用 .NET CLI，可以使用新命令管理 `dotnet workload` 工作负荷：

COMMAND
dotnet workload search	搜索可用的工作负荷。
dotnet workload install	安装指定的工作负荷。
dotnet workload uninstall	删除指定的工作负荷。
dotnet workload update	更新已安装的工作负载。
dotnet workload repair	重新安装所有已安装的工作负载以修复损坏的安装。
dotnet workload list	列出已安装的工作负载。

有关详细信息，请参阅可选的 SDK 工作负载。

System.Text.Json API

在 .NET 6 中进行了许多改进，因此它现在是" System.Text.Json 工业强度"序列化解决方案。

源生成器

.NET 6 为 添加了 新的源 生成器 System.Text.Json 。源生成适用于 JsonSerializer ，可通过多种方式进行配置。它可以提高性能、减少内存使用量，以及简化程序集修整。有关详细信息，请参阅如何在 System.Text.Json 中选择反射或源生成和如何在 System.Text.Json中使用源生成。

可写 DOM

添加了一个新的可写文档对象模型 (DOM)，它补充了预先存在的只读 DOM。当无法将普通旧 CLR 对象用于 POCO (时，) API 提供了一种轻型序列化替代方法。它还允许你有效地导航到大型 JSON 树的子节，然后从该子部分读取数组或反化 POCO。添加了以下新类型以支持可写 DOM：

- JsonNode
- JsonArray
- JsonObject
- JsonValue

有关详细信息，请参阅 JSON DOM 选项。

IAsyncEnumerable 序列化

System.Text.Json 现在支持使用 实例进行序列化和 IAsyncEnumerable<T> 反序列化。异步序列化方法枚举 IAsyncEnumerable<T> 对象图中的任何实例，然后将这些实例序列化为 JSON 数组。对于反序列化，添加了 JsonSerializer.DeserializeAsyncEnumerable<TValue>(Stream, JsonSerializerOptions, CancellationToken) 新方法。有关详细信息，请参阅 IAsyncEnumerable 序列化。

其他新 API

用于验证和默认值的新序列化接口：

- IJsonOnDeserialized
- IJsonOnDeserializing
- IJsonOnSerialized
- IJsonOnSerializing

有关详细信息，请参阅 回调。

新的属性排序属性：

- JsonPropertyOrderAttribute

  有关详细信息，请参阅 配置序列化属性 的顺序。

编写"原始"JSON 的新方法：

- Utf8JsonWriter.WriteRawValue

  有关详细信息，请参阅写入 原始 JSON。

同步序列化和反序列化到流：

- JsonSerializer.Deserialize(Stream, Type, JsonSerializerOptions)
- JsonSerializer.Deserialize(Stream, Type, JsonSerializerContext)
- JsonSerializer.Deserialize<TValue>(Stream, JsonSerializerOptions)
- JsonSerializer.Deserialize<TValue>(Stream, JsonTypeInfo<TValue>)
- JsonSerializer.Serialize(Stream, Object, Type, JsonSerializerOptions)
- JsonSerializer.Serialize(Stream, Object, Type, JsonSerializerContext)
- JsonSerializer.Serialize<TValue>(Stream, TValue, JsonSerializerOptions)

- JsonSerializer.Serialize<TValue>(Stream, TValue, JsonTypeInfo<TValue>)

在序列化期间检测到引用循环时忽略对象的新选项：

- ReferenceHandler.IgnoreCycles

  有关详细信息，请参阅 忽略循环引用。

有关使用 进行序列化和反序列化详细信息，请参阅 .NET 中的 JSON 序列化和 `System.Text.Json` 反序列化。

HTTP/3

.NET 6 包括对 HTTP/3（HTTP 的新版本）的预览支持。HTTP/3 通过使用名为 QUIC 的新基础连接协议解决了一些现有的功能和性能难题。QUIC 可以更快速地建立连接，并且连接独立于 IP 地址，使移动客户端能够漫游到 Wi-fi 和移动电话网络。有关详细信息，请参阅将 HTTP/3 与 HttpClient 一起使用。

ASP.NET Core

ASP.NET Core包括针对 Blazor WebAssembly 应用和单页 (AOT) 编译的最少 API、提前执行 AOT 编译的改进。此外，Blazor 组件现在可以从 JavaScript 呈现并集成到现有的基于 JavaScript 的应用。

OpenTelemetry

.NET 6 改进了 对 OpenTelemetry 的支持，OpenTelemetry是工具、API 和 SDK 的集合，可帮助你分析软件的性能和行为。命名空间中的 System.Diagnostics.Metrics API 实现 OpenTelemetry 指标 API 规范。例如，有四个检测类支持不同的指标方案。检测类包括：

- Counter<T>
- Histogram<T>
- ObservableCounter<T>
- ObservableGauge<T>

安全性

.NET 6 添加了对两个关键安全缓解措施的预览支持：控制流强制技术 (CET) 和"写入独占执行" (W^X) 。

CET 是一种 Intel 技术，在某些较新的 Intel 和 AMD 处理器中可用。它将功能添加到硬件，防止某些控制流劫持攻击。.NET 6 为 x64 Windows CET 提供支持，必须显式启用它。有关详细信息，请参阅 .NET 6 与 Intel CET 卷影堆栈的兼容性。

W^X 适用于 .NET 6 的所有操作系统，但仅在 Apple Silicon 上默认启用。W^X 通过禁止内存页同时可写入和可执行来阻止最简单的攻击路径。

IL 修整

改进了独立部署的剪裁。在 .NET 5 中，仅剪裁未使用的程序集。.NET 6 还添加了对未使用的类型和成员的剪裁。此外，现在默认启用剪裁警告，警告提醒你修整可能会删除运行时 *使用的代码的位置* 。有关详细信息，请参阅剪裁自包含部署和可执行文件。

代码分析

.NET 6 SDK 包括一些新的代码分析器，这些分析器涉及 API 兼容性、平台兼容性、剪裁安全性、在字符串串联和拆分中使用范围、更快的字符串 API 和更快的集合 API。有关分析器中 (和) 的完整列表，请参阅分析器版本 - .NET 6。

自定义平台防护

平台 兼容性分析器 将 类中的方法(例如)识别 `Is<Platform>` OperatingSystem OperatingSystem.IsWindows() 为平台保护。为了允许自定义平台防护，.NET 6 引入了两个新属性，可用于使用受支持的或不受支持的平台名称对字段、属性或方法进行批注：

- SupportedOSPlatformGuardAttribute
- UnsupportedOSPlatformGuardAttribute

Windows 窗体

Application.SetDefaultFont(Font) 是 .NET 6 中的一种新方法，可在整个应用程序中设置默认字体。

已更新 C# Windows窗体应用的模板，以支持指令、文件范围的命名空间和 `global using` 可为空引用类型。此外，它们还包括应用程序启动代码，它减少了样板代码，并允许 Windows 窗体设计器以首选字体呈现设计图面。启动代码是对 的调用，这是一种源生成的方法，可发出对 `ApplicationConfiguration.Initialize()` 其他配置方法(如)的调用 Application.EnableVisualStyles() 。此外，如果通过ApplicationDefaultFont MSBuild设置非默认字体，将 `ApplicationConfiguration.Initialize()` 发出对 的调用 SetDefaultFont(Font) 。

有关详细信息，请参阅 Windows窗体中的新增功能博客文章。

源生成

源 *tarball* 包含 .NET SDK 的所有源，现在是 .NET SDK 内部版本的产品。其他组织(如 Red Hat)可以使用此源 tarball 生成自己的 SDK 版本。

目标框架名字对象

为 .NET 6 添加了其他特定于 OS (框架名字对象) 为 .NET 6 添加了 TFM，例如 `net6.0-android` `net6.0-ios` 、和 `net6.0-macos` 。有关详细信息，请参阅.NET 5+ 特定于 OS 的 TFM。

泛型数学

在 *预览* 版中，可以在 .NET 6 中的泛型类型上使用运算符。.NET 6 引入了许多接口，这些接口利用 C# 10 的新预览功能 `static abstract` ，即接口成员。这些接口对应于不同的运算符，例如 `IAdditionOperators` ，表示 `+` 运算符。这些接口在System.Runtime.Experimental NuGet包中提供。有关详细信息，请参阅通用 数学 博客文章。

NuGet包验证

如果你是一名NuGet开发人员，新的包验证工具可用于验证包是否一致且格式良好。可以确定是否：

- 包版本之间存在任何中断性变更。
- 包具有一组相同的公共 API，用于所有特定于运行时的实现。
- 目标框架或运行时适用性存在任何差距。

有关详细信息，请参阅包 验证 博客文章。

反射 API

.NET 6 引入了以下新 API，用于检查代码并提供可为 null 性的信息：

- System.Reflection.NullabilityInfo
- System.Reflection.NullabilityInfoContext
- System.Reflection.NullabilityState

这些 API 可用于基于反射的工具和序列化程序。

Microsoft.Extensions API

多个扩展命名空间在 .NET 6 中进行了改进，如下表所示。

Microsoft.Extensions.DependencyInjection	CreateAsyncScope 使你能够安全地 `using` 对注册服务的服务提供商使用 IAsyncDisposable 语句。
Microsoft.Extensions.Hosting	新方法 ConfigureHostOptions 简化了应用程序设置。
Microsoft.Extensions.Logging	Microsoft.Extensions.Logging 具有用于性能日志记录 API 的新源生成器。如果将新的 添加到日志记录方法，则触发 LoggerMessageAttribute 源 `partial` 生成器。在编译时，生成器生成 方法的实现，这通常比现有日志记录解决方案运行时 `partial` 更快。有关详细信息，请参阅编译 时日志记录源生成。

新的 LINQ API

.NET 6 中添加了许多 LINQ 方法。下表中列出的大多数新方法在 类型中具有等效 System.Linq.Queryable 的方法。

Enumerable.TryGetNonEnumeratedCount<TSource>(IEnumerable<TSource>, Int32)	尝试在不强制枚举的情况下确定序列中的元素数。
Enumerable.Chunk<TSource>(IEnumerable<TSource>, Int32)	将序列的元素拆分为指定大小的区块。
Enumerable.MaxBy 和 Enumerable.MinBy	使用键选择器查找最大或最小元素。
Enumerable.DistinctBy、Enumerable.ExceptBy、Enumerable.IntersectBy 和 Enumerable.UnionBy	执行基于集的操作的方法的这些新变体允许使用键选择器函数指定相等性。
Enumerable.ElementAt<TSource>(IEnumerable<TSource>, Index) 和 Enumerable.ElementAtOrDefault<TSource>(IEnumerable<TSource>, Index)	接受从序列的开头或结尾计算的索引 — ，例如，`Enumerable.Range(1, 10).ElementAt(^2)` 返回 `9` 。
Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, TSource) 和 Enumerable.FirstOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource) Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, TSource) 和 Enumerable.LastOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource) Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, TSource) 和 Enumerable.SingleOrDefault<TSource>(IEnumerable<TSource>, Func<TSource,Boolean>, TSource)	如果序列为空，则新重载允许您指定要使用的默认值。
Enumerable.Max<TSource>(IEnumerable<TSource>, IComparer<TSource>) 和 Enumerable.Min<TSource>(IEnumerable<TSource>, IComparer<TSource>)	新重载允许您指定比较器。

"	"
Enumerable.Take<TSource>(IEnumerable<TSource>, Range)	接受 Range 参数以简化序列的切片 —，例如，可以使用 `source.Take(2..7)` 而不是 `source.Take(7).Skip(2)` 。
Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>)	使用 三个 指定序列的元素生成元组序列。

日期、时间和时区改进

.NET 6 中添加了以下两个结构：System.DateOnly 和 System.TimeOnly 。它们分别表示的日期部分和时间部分 DateTime 。DateOnly 适用于生日和周年纪念，适用 TimeOnly 于每日警报和每周工作时间。

现在，可以在安装了时区数据的任何操作系统上使用 (IANA) 或 Windows 时区 id 的 Internet 分配的号码颁发机构。已 TimeZoneInfo.FindSystemTimeZoneById(String) 更新方法，以便自动将其输入从 Windows 时区转换为 IANA 时区 (，反之亦然) 如果系统上找不到请求的时区，则为。此外， TryConvertIanaIdToWindowsId(String, String) TryConvertWindowsIdToIanaId 当你仍需要从一种时区格式手动转换到另一种时区格式时，还为方案添加了新方法。

还有其他一些时区改进功能。有关详细信息，请参阅 .net 6 中的日期、时间和时区增强功能。

PriorityQueue 类

新 PriorityQueue<TElement,TPriority> 类表示同时具有值和优先级的项的集合。项按递增优先级顺序取消排队 —，即优先级值最低的项先取消排队。此类实现 最小堆 数据结构。

另请参阅

- C# 10 中的新增功能
- F# 6 中的新增功能
- EF Core 6 中的新增功能
- .NET 6 发行说明
- Visual Studio 2022 的发行说明
- 博客：宣布推出 .NET 6
- 博客：尝试新的 System.web 源生成器

.NET 6 中的中断性变更

2021/11/16 •

如果要将应用迁移到 .NET 6，则此处列出的中断性变更可能会影响到你。变更按技术领域分组，例如 ASP.NET Core 或 Windows 窗体。

> **NOTE**
> 本文是当前正在进行的工作。这并不是 .NET 6 中的中断性变更的完整列表。若要查询仍处于待发布状态的中断性变更，请查看 .NET 问题。

ASP.NET Core

AddDataAnnotationsValidation 方法已过时	✔	✘	
从 Microsoft.AspNetCore.App 共享框架中删除了程序集	✘	✔	
Blazor：RequestImageFileAsync 方法中的参数名称已更改	✔	✘	预览版 1
Blazor：已替换 WebEventDescriptor.EventArgsType 属性	✘	✘	
Blazor：字节数组互操作	✔	✘	预览版 6
更改了 @microsoft/signalr-protocol-msgpack 中的 MessagePack 库	✘	✔	
ClientCertificate 属性不会对 HttpSys 触发重新协商	✔	✘	
EndpointName 元数据不自动设置	✔	✘	RC 2
Kestrel：日志消息属性已更改	✔	✘	
Microsoft.AspNetCore.Http.Features 拆分	✘	✔	
中间件：HTTPS 重定向中间件会在 HTTPS 端口不明确时引发异常	✔	✘	

[illegible]	[illegible]	[illegible]	[illegible]
中间件：新 Use 重载	✔	✘	预览版 4
RC 1 中的最小 API 重命名	✘	✘	RC 1
RC 2 中的最小 API 重命名	✘	✘	RC 2
使用 System.Text.Json 时，MVC 不缓冲 IAsyncEnumerable 类型	✔	✘	预览版 4
可以为 Null 的引用类型注释已更改	✔	✘	
已过时和已删除的 API	✔	✘	预览版 1
PreserveCompilationContext 不默认配置	✘	✔	
Razor：编译器不再生成 Views 程序集	✔	✘	预览版 3
Razor：日志记录 ID 更改	✘	✔	RC1
Razor：RazorEngine API 标记为已过时	✔	✘	预览版 1
SignalR：Java 客户端已更新到 RxJava3	✘	✔	预览版 4
验证 TryParse 和 BindAsync 方法	✘	✘	RC 2

Core .NET 库

[illegible]	[illegible]	[illegible]	[illegible]
API 已过时并带有非默认诊断 ID	✔	✘	预览版 1
对可以为 Null 的引用类型注释的更改	✔	✘	预览版 1-2
“调试”方法中的条件字符串计算	✔	✘	RC 1
Windows 上的 Environment.ProcessorCount 行为	✔	✘	预览版 2
Unix 上的 File.Replace 引发匹配 Windows 的异常	✔	✘	预览版 7

□□	□□□□□	□□□	□□□
FileStream 使用 Unix 上的共享锁来锁定文件	✗	✓	预览版 1
FileStream 不再将文件偏移量与操作系统同步	✗	✗	预览版 4
FileStream.Position 在 ReadAsync 或 WriteAsync 完成后更新	✗	✗	预览版 4
过时 API 的新诊断 ID	✓	✗	预览版 5
AssociatedMetadataTypeTypeDescriptionProvider 中新的可为 Null 的注释	✓	✗	RC 2
新的 System.Linq.Queryable 方法重载	✓	✗	预览版 3-4
已从包中删除较旧的框架版本	✗	✓	预览版 5
参数名称已更改	✓	✗	预览版 1
数据流派生类型中的参数名称	✓	✗	预览版 1
DeflateStream、GZipStream 和 CryptoStream 中的部分和零字节读取	✓	✗	预览版 6
标准数字格式分析精度	✓	✗	预览版 2
接口中的静态抽象成员	✗	✓	预览版 7
StringBuilder.Append 重载和计算顺序	✗	✓	RC 1
仅在 Windows 上支持 System.Drawing.Common	✗	✗	预览版 7
System.Security.SecurityContext 被标记为过时	✓	✗	RC 1
Task.FromResult 可能返回单一实例	✗	✓	预览版 1
来自 BackgroundService 的未经处理的异常	✓	✗	预览版 4
XmlDocument.XmlResolver 为 Null 性更改	✗	✓	RC 1

针对无效索引的 XNodeReader.GetAttribute 行为	✔	✕	预览版 2

密码

CreateEncryptor 方法针对不正确的反馈大小引发异常	✕	✔	预览版 7

Entity Framework Core

EF Core 6 中的中断性变更

Extensions

AddProvider 检查非 null 提供程序	✔	✕	RC 1
FileConfigurationProvider.Load 引发 InvalidDataException	✔	✕	RC 1
解析已释放的 ServiceProvider 引发异常	✔	✕	RC 1

全球化

全球化固定模式下的区域性创建和大小写映射			预览版 7

Interop

接口中的静态抽象成员	✕	✔	预览版 7

JIT 编译器

根据 ECMA-335 强制调用参数	✔	✔	预览版 1

网络

□□	□□□□□	□□□	□□□
从 SPN 中删除了用于 Kerberos 和协商的端口	✗	✓	RC 1
WebRequest、WebClient 和 ServicePoint 已过时	✓	✗	预览版 1

SDK 中 IsInRole 中的声明

□□	□□□□□	□□□	□□□
`dotnet run` 的 `-p` 选项已弃用	✓	✗	预览版 6
早期版本不支持模板中的 C# 代码	✓	✓	预览版 7
生成适用于 macOS 的 apphost	✓	✗	预览版 6
针对发布输出中的重复文件生成错误	✗	✓	预览版 1
从 ProjectReference 协议中删除了 GetTargetFrameworkProperties 和 GetNearestTargetFramework	✗	✓	预览版 1
C# 项目中的隐式全局 using 指令	✓	✗	RC 1
ARM64 上模拟 x64 的安装位置	✓	✗	RC 2
MSBuild 不再支持调用 GetType()			RC 1
OutputType 不会自动设置为 WinExe	✓	✗	RC 1
未指定自包含时出现 RuntimeIdentifier 警告	✓	✗	RC 1

序列化

□□	□□□□□	□□□	□□□
IAsyncEnumerable 序列化	✓	✗	预览版 4

[illegible]	[illegible]	[illegible]	[illegible]
JSON 源-生成 API 重构	✗	✓	RC 2
JsonNode 不再支持 C# `dynamic` 类型	✗	✓	预览版 7
集合属性上的 JsonNumberHandlingAttribute	✗	✓	RC 1
新的 JsonSerializer 源生成器重载	✗	✓	预览版 6

Windows 窗体

[illegible]	[illegible]	[illegible]	[illegible]
C# 模板使用应用程序启动	✓	✗	RC 1
所选的 TableLayoutSettings 属性会引发 InvalidEnumArgumentException	✗	✓	预览版 1
与 DataGridView 相关的 API 现在引发 InvalidOperationException	✗	✓	预览版 4
ListViewGroupCollection 方法引发新的 InvalidOperationException	✗	✓	RC 2
增加了 NotifyIcon.Text 最大文本长度	✗	✓	预览版 1
一些 API 引发 ArgumentNullException	✗	✓	预览版 1-4
如果节点被分配到其他地方，则 TreeNodeCollection.Item 抛出异常	✗	✓	预览版 1

.NET 5 的新变化

2021/11/16 •

.NET 5 是 .NET Core 的下一个主要版本，3.1。我们出于以下两个原因将此新版本命名为 .NET 5，而不是 .NET Core 4：

- 我们跳过了版本1-4，以避免与 .NET Framework 4.x 混淆。
- 我们从名称中删除了 "核心" 以强调，这是 .NET 的主要实现。.NET 5 支持比 .NET Core 或 .NET Framework 更多的应用程序和平台。

ASP.NET Core 5.0 基于 .net 5，但保留名称 "核心" 以避免与 ASP.NET MVC 5 混淆。同样，Entity Framework Core 5.0 保留名称 "核心"，以避免将其与实体框架5和6混淆。

与 .NET Core 3.1 相比，.NET 5 包含以下改进和新功能：

- C # 更新
- F # 更新
- Visual Basic 更新
- System.object 新功能
- 单个文件应用
- 应用修整
- WindowsARM64 和 ARM64 内部函数
- 转储调试的工具支持
- 对于可以为 null 的引用类型，运行时库的批注百分比为80%
- 性能改进：
  - 垃圾回收 (GC)
  - System.Text.Json
  - System.Text.RegularExpressions
  - Async ValueTask pooling
  - 容器大小优化
  - 多个区域

.NET 5 不会替换 .NET Framework

.net 5 是 .net 的主要实现，但仍支持 .NET Framework 4.x。

没有将以下技术从 .NET Framework 移植到 .net 5 的计划，但 .net 5 中有一些替代方法：

tt	tttttt
Web 窗体	ASP.NET Core Blazor或Razor Pages
WindowsWorkflow (WF)	开源 CoreWF 或 Elsa-工作流

Windows Communication Foundation

Windows Communication Foundation (WCF) 的原始实现仅在 Windows 上受支持。但是，.NET Foundation 中提供了一个客户端端口。它完全是 开放源代码、跨平台并受 Microsoft 支持。下面列出了核心 NuGet 包：

- System.ServiceModel.Duplex

- System.servicemodel. 联合
- System.ServiceModel.Http
- System.ServiceModel.NetTcp
- System.ServiceModel.Primitives
- System.servicemodel. Security

社区维护补充前述的客户端库的服务器组件。可以在CoreWCF中找到 GitHub 存储库。Microsoft 不正式支持这些服务器组件。有关 WCF 的替代方法，请考虑使用 gRPC。

.NET 5 不会替换 .NET Standard

新的应用程序开发可以 `net5.0` 为所有项目类型（包括类库）(TFM) 指定目标框架名字对象。在 .NET 5 工作负载之间共享代码，只需 `net5.0` TFM 即可。

对于 .NET 5 应用和库，`net5.0` 目标框架名字对象 (TFM) 组合并替换 `netcoreapp` 和 `netstandard` tfm。但是，如果你计划在 .NET Framework、.net Core 和 .net 5 工作负载之间共享代码，则可以通过将指定为 TFM 来实现此目的 `netstandard2.0` 。有关详细信息，请参阅 .NET Standard。

C # 更新

编写 .NET 5 应用程序的开发人员将有权访问最新的 c # 版本和功能。.NET 5 与 c # 9 配对，这为语言带来了许多新功能。下面是几个要点：

- 记录：具有基于值的相等语义的引用类型和新表达式支持的非破坏性变化 `with` 。
- 关系模式匹配：将模式匹配功能扩展为关系运算符以用于比较计算和表达式，包括逻辑模式-新关键字 `and` 、`or` 和 `not` 。
- 顶级语句：作为加速采用和学习 c # 的一种 `Main` 方法，可以省略方法，并使应用程序尽可能简单，如下所示：

  ```
  System.Console.Write("Hello world!");
  ```

- 函数指针：语言构造，它公开以下中间语言 (IL) 操作码：`ldftn` 和 `calli` 。

有关可用 c # 9 功能的详细信息，请参阅 c # 9 中的新增功能。

源生成器

除了一些突出显示的新 c # 功能，源生成器也是开发人员项目的方式。源生成器允许在编译过程中运行的代码检查程序，并生成与其余代码一起编译的其他文件。

有关源生成器的详细信息，请参阅 c # 源生成器 和 c # 源生成器示例简介。

F # 更新

F # 是 .NET 函数编程语言，在 .NET 5 中，开发人员可以访问 F # 5。其中一个新功能是内插字符串，类似于 c # 中的内插字符串，甚至是 JavaScript。

```
let name = "David"
let age = 36
let message = $"{name} is {age} years old."
```

除了基本的字符串内插之外，还存在类型化的内插。对于类型化内插，给定类型必须匹配格式说明符。

```
let name = "David"
let age = 36
let message = $"%s{name} is %d{age} years old."
```

这类似于 `sprintf` 根据类型安全输入来设置字符串格式的函数。

有关详细信息，请参阅 F # 5 中的新增功能。

Visual Basic 更新

.net 5 中没有适用于 Visual Basic 的新语言功能。但对于 .net 5，Visual Basic 支持扩展到：

“	`DOTNET NEW` “
控制台应用程序	`console`
类库	`classlib`
WPF 应用程序	`wpf`
WPF 类库	`wpflib`
WPF 自定义控件库	`wpfcustomcontrollib`
WPF 用户控件库	`wpfusercontrollib`
Windows 窗体 (WinForms) 应用程序	`winforms`
Windows 窗体 (WinForms) 类库	`winformslib`
单元测试项目	`mstest`
NUnit 3 测试项目	`nunit`
NUnit 3 测试项	`nunit-test`
xUnit 测试项目	`xunit`

有关 .NET CLI 中的项目模板的详细信息，请参阅 `dotnet new` 。

System.object 新功能

和中有一些新功能：

- 保留引用并处理循环引用
- HttpClient 和 HttpContent 扩展方法
- 允许或写入带引号的数字
- 支持不可变类型和 C# 9 记录
- 支持非公共属性访问器
- 支持字段
- 有条件地忽略属性
- 支持非字符串键字典

- 允许自定义转换器处理 null
- 复制 JsonSerializerOptions
- 使用 Web 默认值创建 JsonSerializerOptions

另请参阅

- 一个 .NET 之旅
- .NET 5 中的性能改进
- 下载 .NET SDK

.NET 5 中的中断性变更

2021/11/16 •

如果要将应用迁移到 .NET 5，则此处列出的中断性变更可能会影响到你。变更按技术领域分组，例如 ASP.NET Core 或加密。

ASP.NET Core

更改	源兼容	二进制兼容
ASP.NET Core 应用反序列化带引号的数字	✔	✕
标记为已过时的 AzureAD.UI 和 AzureADB2C.UI API	✔	✕
BinaryFormatter 序列化方法已过时	✔	✕
终结点路由中的资源为 HttpContext	✔	✕
Microsoft 预先指定的 Azure 集成包已删除	✕	✔
Blazor：Blazor 应用中已更改路由优先逻辑	✔	✕
Blazor：更新的浏览器支持	✔	✔
Blazor：编译器剪裁掉的无意义空格	✔	✕
Blazor：JSObjectReference 和 JSInProcessObjectReference 类型是内部类型	✔	✕
Blazor：NuGet 包的目标框架已更改	✕	✔
Blazor：ProtectedBrowserStorage 功能已移动到共享框架	✔	✕
Blazor：RenderTreeFrame 只读公共字段现在是属性	✕	✔
Blazor：更新的静态 Web 资产的验证逻辑	✕	✔
浏览器不支持的加密 API	✕	✔
扩展：包引用更改	✕	✔

□□	□□□□	□□□
Kestrel 和 IIS BadHttpRequestException 类型已过时	✔	✕
IHttpClientFactory 创建的 HttpClient 实例记录整数状态代码	✔	✕
HttpSys：默认情况下禁用客户端证书重新协商	✔	✕
IIS：保留 UrlRewrite 中间件查询字符串	✔	✕
Kestrel：默认检测的配置更改	✔	✕
Kestrel：默认支持的 TLS 协议版本已更改	✔	✕
Kestrel：在不兼容的 Windows 版本上通过 TLS 禁用 HTTP/2	✔	✔
Kestrel：Libuv 传输标记为已过时	✔	✕
ConsoleLoggerOptions 上已过时的属性	✔	✕
ResourceManagerWithCultureStringLocalizer 类和 WithCulture 接口成员已删除	✔	✕
已删除 Pubternal API	✔	✕
请求本地化中间件中删除了已过时的构造函数	✔	✕
中间件：数据库错误页标记为已过时	✔	✕
异常处理程序中间件会引发原始异常	✔	✔
ObjectModelValidator 调用验证的新重载	✔	✕
Cookie 名称编码已删除	✔	✕
IdentityModel NuGet 包版本已更新	✕	✔
SignalR：MessagePack 中心协议选项类型已更改	✔	✕
SignalR：MessagePack 中心协议已变动	✔	✕
UseSignalR 和 UseConnections 方法已删除	✔	✕

[illegible]	[illegible]	[illegible]
CSV 内容类型已更改为符合标准	✔	✘

代码分析

[illegible]	[illegible]	[illegible]
CA1416 警告	✔	✘
CA1417 警告	✔	✘
CA1831 警告	✔	✘
CA2013 警告	✔	✘
CA2014 警告	✔	✘
CA2015 警告	✔	✘
CA2200 警告	✔	✘
CA2247 警告	✔	✘

Core .NET 库

[illegible]	[illegible]	[illegible]
适用于单文件发布的与程序集相关的 API 更改	✘	✔
BinaryFormatter 序列化方法已过时	✔	✘
代码访问安全性 API 已过时	✔	✘
CreateCounterSetInstance 会引发 InvalidOperationException	✔	✘
默认 ActivityIdFormat 为 W3C	✘	✔
Environment.OSVersion 返回正确的版本	✘	✔
FrameworkDescription 的值是 .NET 而不是 .NET Core	✔	✘
GAC API 已过时	✔	✘
硬件内在 IsSupported 检查	✘	✔
IntPtr 和 UIntPtr 实现 IFormattable	✔	✘

[illegible]	[illegible]	[illegible]
LastIndexOf 处理空搜索字符串	✗	✓
Unix 上包含非 ASCII 字符的 URI 路径	✗	✓
API 已过时并带有非默认诊断 ID	✓	✗
ConsoleLoggerOptions 上已过时的属性	✓	✗
LINQ OrderBy.First 的复杂性	✗	✓
已重命名或已删除 OSPlatform 属性	✓	✗
Microsoft.DotNet.PlatformAbstractions 包已删除	✗	✓
PrincipalPermissionAttribute 已过时	✓	✗
来自预览版本的参数名称更改	✓	✗
引用程序集中的参数名称更改	✓	✗
远程处理 API 已过时	✗	✓
Activity.Tags 列表的顺序是相反的	✓	✗
SSE 和 SSE2 比较方法处理 NaN	✓	✗
Thread.Abort 已过时	✓	✗
Unix 上 UNC 路径的 URI 识别	✗	✓
UTF-7 代码路径已过时	✓	✗
Vector2.Lerp 和 Vector4.Lerp 的行为变更	✓	✗
向量<T> 引发 NotSupportedException	✗	✓

密码

[illegible]	[illegible]	[illegible]
浏览器不支持的加密 API	✗	✓
Cryptography.Oid 仅限 init	✓	✗
Linux 上的默认 TLS 密码套件	✗	✓
对加密抽象的 Create() 重载已过时	✓	✗

默认 FeedbackSize 值已更改	✔	✖

Entity Framework Core

EF Core 5.0 中的中断性变更

全球化

在 Windows 上使用 ICU 库	✖	✔
StringInfo 和 TextElementEnumerator 与 UAX29 兼容	✖	✔
Latin-1 字符的 Unicode 类别已更改	✔	✖
TextInfo.ListSeparator 值已更改	✔	✖

Interop

已删除对 WinRT 的支持	✖	✔
将 RCW 强制转换为 InterfaceIsIInspectable 会引发异常	✖	✔
不在非 Windows 平台上探测 A/W 后缀	✖	✔

网络

Cookie 路径处理符合 RFC 6265	✔	✖
调用 SendToAsync 后更新 LocalEndPoint	✔	✖
MulticastOption.Group 不接受 NULL	✔	✖
流允许后续开始操作	✖	✔
已从 .NET 运行时中删除 WinHttpHandler	✖	✔

SDK 中 IsInRole 中的声明

[illegible]	[illegible]	[illegible]
默认已导入 Directory.Packages.props 文件	✗	✓
可执行项目引用不匹配的可执行文件时生成错误		✓
FrameworkReference 替换为适用于 Windows SDK 的 WindowsSdkPackageVersion	✓	✗
未定义 NETCOREAPP3_1 预处理器符号	✓	✗
OutputType 已设置为 WinExe	✗	✓
PublishDepsFilePath 行为变更	✗	✓
TargetFramework 从 netcoreapp 更改为 net	✗	✓
WinForms 和 WPF 应用使用 Microsoft.NET.Sdk	✗	✓

安全性

[illegible]	[illegible]	[illegible]
代码访问安全性 API 已过时	✓	✗
PrincipalPermissionAttribute 已过时	✓	✗
UTF-7 代码路径已过时	✓	✗

序列化

[illegible]	[illegible]	[illegible]
BinaryFormatter.Deserialize 重新包装异常	✓	✗
JsonSerializer.Deserialize 需要单字符的字符串	✓	✗
ASP.NET Core 应用反序列化带引号的数字	✓	✗
JsonSerializer.Serialize 引发 ArgumentNullException	✓	✗
非公共的无参数构造函数不用于反序列化	✓	✗

序列化键值对时可以采用选项	✔	✘

Windows 窗体

本机代码无法访问 Windows 窗体对象	✔	✘
OutputType 已设置为 WinExe	✘	✔
DataGridView 未重置自定义字体	✔	✘
方法引发 ArgumentException	✔	✘
方法引发 ArgumentNullException	✔	✘
属性引发 ArgumentOutOfRangeException	✔	✘
TextFormatFlags.ModifyString 已过时	✔	✘
DataGridView API 引发 InvalidOperationException	✔	✘
WinForms 应用使用 Microsoft.NET.Sdk	✘	✔
已删除的状态栏控件	✔	✘

WPF

OutputType 已设置为 WinExe	✘	✔
WPF 应用使用 Microsoft.NET.Sdk	✘	✔

.NET Core 3.1 的新增功能

2021/11/16 •

本文介绍了 .NET Core 3.1 中的新增功能。此版本包含对 .NET Core 3.0 的细微改进，重点介绍小型但重要的修复。.NET Core 3.1 中最重要的特性为，它是长期支持 (LTS) 版本。

如果使用的是 Visual Studio 2019，则必须更新到 Visual Studio 2019 版本 16.4 或更高版本才能使用 .NET Core 3.1 项目。有关 Visual Studio 版本 16.4 中新增功能的详细信息，请参阅 Visual Studio 2019 版本 16.4 中的新增功能。

Visual Studio for Mac 也支持 .NET Core 3.1，并且 Visual Studio for Mac 8.4 中就包括 .NET Core 3.1。

有关版本的详细信息，请参阅 .NET Core 3.1 公告。

- 在 Windows、macOS 或 Linux 上下载并开始使用 .NET Core 3.1。

长期支持

.NET Core 3.1 是未来三年包含来自 Microsoft 的支持的 LTS 版本。强烈建议将应用移到 .NET Core 3.1。其他主要版本的当前生命周期如下所示：

RELEASE	ıı
.NET Core 3.0	生命周期终结于 2020 年 3 月 3 日。
.NET Core 2.2	生命周期终结于 2019 年 12 月 23 日。
.NET Core 2.1	生命周期终结于 2021 年 8 月 21 日。

有关详细信息，请参阅 .NET Core 支持策略。

macOS appHost 和公证

仅 macOS

从已公证的适用于 macOS 的 .NET Core SDK 3.1 开始，默认已禁用 appHost 设置。有关详细信息，请参阅 macOS Catalina 公证以及对 .NET Core 下载和项目的影响。

启用 appHost 设置后，.NET Core 在生成或发布时将生成本机 Mach-O 可执行文件。如果使用 `dotnet run` 命令从源代码中运行应用，或通过启动 Mach-O 可执行文件直接运行应用，则应用会在 appHost 的上下文中运行。

如果没有 appHost，用户就只能使用 `dotnet <filename.dll>` 命令启动依赖框架的应用。发布独立应用时，始终会创建 appHost。

可以在项目级别配置 appHost，或通过 `-p:UseAppHost` 参数切换特定 `dotnet` 命令的 appHost：

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

有关 `UseAppHost` 设置的详细信息，请参阅 Microsoft.NET.Sdk 的 MSBuild 属性。

Windows 窗体

仅限 Windows

> **WARNING**
> Windows 窗体中发生重大变更。

旧版控件包含在 Windows 窗体中，这些窗体在一段时间内无法在 Visual Studio 设计器工具箱中使用。它们已替换为 .NET Framework 2.0 中的新控件。它们已从适用于 .NET Core 3.1 的桌面 SDK 中删除。

[illegible]	[illegible]	[illegible] API
DataGrid	DataGridView	DataGridCell DataGridRow DataGridTableCollection DataGridColumnCollection DataGridTableStyle DataGridColumnStyle DataGridLineStyle DataGridParentRowsLabel DataGridParentRowsLabelStyle DataGridBoolColumn DataGridTextBox GridColumnStylesCollection GridTableStylesCollection HitTestType
ToolBar	ToolStrip	ToolBarAppearance
ToolBarButton	ToolStripButton	ToolBarButtonClickEventArgs ToolBarButtonClickEventHandler ToolBarButtonStyle ToolBarTextAlign
ContextMenu	ContextMenuStrip	
Menu	ToolStripDropDown ToolStripDropDownMenu	MenuItemCollection
MainMenu	MenuStrip	
MenuItem	ToolStripMenuItem	

我们建议你将应用程序更新到 .NET Core 3.1 并移动到替换控件。替换控件是一个简单的过程，本质上属于“查找和替换”类型。

C++/CLI

仅限 Windows

已添加对创建 C++/CLI（也称为“托管 C++”）项目的支持。从这些项目生成的二进制文件与 .NET Core 3.0 及更高版本兼容。

若要添加对 Visual Studio 2019 版本 16.4 中的 C++/CLI 的支持，请安装“使用 C++ 的桌面开发”工作负荷。此工作负载将两个模板添加到 Visual Studio：

- CLR 类库(.NET Core)
- CLR 空项目(.NET Core)

后续步骤

- 查看 .NET Core 3.0 和 3.1 之间的重大变更。
- 查看用于 Windows 窗体应用的 .NET Core 3.1 中的中断性变更。

.NET Core 3.1 中的中断性变更

2021/11/16 •

若要迁移到 3.1 版 .NET Core 或 ASP.NET Core，本文中列出的中断性变更可能会影响到你的应用。

ASP.NET Core

- HTTP：浏览器的 SameSite 更改会影响身份验证

HTTP：浏览器的 SameSite 更改会影响身份验证

某些浏览器（如 Chrome 和 Firefox）对 Cookie 的 `SameSite` 实现进行了中断性变更。这些变更会影响 OpenID Connect 和 WS 联合身份验证等远程身份验证方案，必须通过发送 `SameSite=None` 来选择退出。但是，`SameSite=None` 会在 iOS 12 和其他浏览器的某些较早版本上中断运行。应用需探查这些版本，并忽略 `SameSite`。

有关此问题的讨论，请参阅 dotnet/aspnetcore#14996。

引入的版本

3.1 预览版 1

旧行为

`SameSite` 是对 HTTP Cookie 的 2016 草案标准扩展。它旨在减少跨站点请求伪造 (CSRF)。它最初设计成一项功能，服务器可通过添加新参数选择加入该功能。ASP.NET Core 2.0 添加了对 `SameSite` 的初始支持。

新行为

Google 提出了一项不向后兼容的新草案标准。该标准将默认模式更改为 `Lax` 并添加了用于选择退出的新条目 `None`。`Lax` 可满足大多数应用 Cookie；但是，它会造成 OpenID Connect 和 WS 联合身份验证登录等跨站点方案中断。由于请求流程不同，大多数 OAuth 登录不受影响。新的 `None` 参数会导致实现先前草案标准的客户端（例如 iOS 12）出现兼容性问题。Chrome 80 将包含这些更改。有关 Chrome 产品发布日程表，请查看 SameSite 更新。

已更新 ASP.NET Core 3.1 来实现新的 `SameSite` 行为。该更新重新定义了 `SameSiteMode.None` 的行为以发出 `SameSite=None`，并添加了一个新值 `SameSiteMode.Unspecified` 以忽略 `SameSite` 属性。现在，所有 Cookie API 都默认为 `Unspecified`，但某些使用 Cookie 的组件设置了更特定于其方案的值，例如 OpenID Connect 相关性和 nonce Cookie。

有关此方面的其他最新更改，请参阅 HTTP：某些 Cookie SameSite 默认值已更改为“None”。在 ASP.NET Core 3.0 中，大多数默认值已从 SameSiteMode.Lax 更改为 SameSiteMode.None（但仍使用之前的标准）。

更改原因

浏览器和规范更改如前文所述。

建议操作

与远程站点交互（例如通过第三方登录）的应用需要：

- 在多个浏览器中测试这些方案。
- 应用支持旧版浏览器中讨论的 Cookie 策略浏览器探查缓解措施。

有关测试和浏览器探查说明，请参阅下一部分。

确定你是否受到影响

使用可选择采用新行为的客户端版本测试 Web 应用。Chrome、Firefox 和 Microsoft Edge Chromium 都具有可用于测试的新的“选择加入”功能标志。在应用修补程序（特别是 Safari）后，验证应用是否与较旧的客户端版本兼容。有关详细信息，请参阅支持旧版浏览器。

Chrome

Chrome 78 及更高版本生成误导性的测试结果。这些版本具有临时缓解措施，允许 Cookie 的使用时间小于两分钟。启用合适的测试标志后，Chrome 76 和 77 会生成更准确的结果。要测试新行为，请将 `chrome://flags/#same-site-by-default-cookies` 切换为"已启用"。据报告，Chrome 75 及更早版本使用新的 `None` 设置时失败。有关详细信息，请参阅支持旧版浏览器。

Google 不提供较旧的 Chrome 版本。但是，你可下载较旧版本的 Chromium，这将足以用于测试。按照下载 Chromium 的说明进行操作。

- Chromium 76 Win64
- Chromium 74 Win64

Safari

Safari 12 严格执行了先前的草案，如果它在 Cookie 中检测到新的 `None` 值，则将失败。必须通过支持旧版浏览器中所示的浏览器探查代码来避免这种情况。请确保使用 Microsoft 身份验证库 (MSAL)、Active Directory 身份验证库 (ADAL) 或所使用的任何库来测试 Safari 12 和 13 以及基于 WebKit 的 OS 样式的登录。问题取决于基础 OS 版本。已知 OSX Mojave 10.14 和 iOS 12 存在与新行为相关的兼容性问题。升级到 OSX Catalina 10.15 或 iOS 13 会解决此问题。Safari 当前没有用于测试新规范行为的选择加入标志。

Firefox

通过在具有功能标志 `network.cookie.sameSite.laxByDefault` 的 `about:config` 页面上选择加入，可在版本 68 及更高版本上测试 Firefox 对新标准的支持。Firefox 旧版本未报告兼容性问题。

Microsoft Edge

虽然 Microsoft Edge 支持旧的 `SameSite` 标准，但从版本 44 开始，它与新标准不存在任何兼容性问题。

Microsoft Edge Chromium

功能标志为 `edge://flags/#same-site-by-default-cookies`。使用 Microsoft Edge Chromium 78 进行测试时，未发现兼容性问题。

Electron

Electron 的版本包括较早版本的 Chromium。例如，Microsoft Teams 使用的 Electron 版本为 Chromium 66，该版本呈现了较旧的行为。使用你的产品所用的 Electron 版本执行你自己的兼容性测试。有关详细信息，请参阅支持旧版浏览器。

支持旧版浏览器

2016 `SameSite` 标准要求将未知值视为 `SameSite=Strict` 值。因此，任何支持原始标准的旧版浏览器都可能在检测到 `SameSite` 属性具有 `None` 值时中断。如果 Web 应用要支持这些旧版浏览器，它们必须实现浏览器探查。ASP.NET Core 不会为你实现浏览器探查，因为 `User-Agent` 请求标头值非常不稳定，每周都会更改。相反，Cookie 策略中的扩展点允许添加特定于 `User-Agent` 的逻辑。

在 Startup.cs 中，添加以下代码：

```
private void CheckSameSite(HttpContext httpContext, CookieOptions options)
{
    if (options.SameSite == SameSiteMode.None)
    {
        var userAgent = httpContext.Request.Headers["User-Agent"].ToString();
        // TODO: Use your User Agent library of choice here.
        if (/* UserAgent doesn't support new behavior */)
        {
            options.SameSite = SameSiteMode.Unspecified;
        }
    }
}

public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CookiePolicyOptions>(options =>
    {
        options.MinimumSameSitePolicy = SameSiteMode.Unspecified;
        options.OnAppendCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
        options.OnDeleteCookie = cookieContext =>
            CheckSameSite(cookieContext.Context, cookieContext.CookieOptions);
    });
}

public void Configure(IApplicationBuilder app)
{
    // Before UseAuthentication or anything else that writes cookies.
    app.UseCookiePolicy();

    app.UseAuthentication();
    // code omitted for brevity
}
```

“选择退出”开关

通过 `Microsoft.AspNetCore.SuppressSameSiteNone` 兼容性开关，可暂时选择退出新的 ASP.NET Core Cookie 行为。 将以下 JSON 添加到项目的 runtimeconfig.template.json 文件中：

```
{
  "configProperties": {
    "Microsoft.AspNetCore.SuppressSameSiteNone": "true"
  }
}
```

其他版本

下述项的相关 `SameSite` 修补程序即将发布：

- ASP.NET Core 2.1、2.2 和 3.0
- `Microsoft.Owin` 4.1
- `System.Web`（用于 .NET Framework 4.7.2 及更高版本）

类别

ASP.NET

受影响的 API

- Microsoft.AspNetCore.Builder.CookiePolicyOptions.MinimumSameSitePolicy
- Microsoft.AspNetCore.Http.CookieBuilder.SameSite
- Microsoft.AspNetCore.Http.CookieOptions.SameSite
- Microsoft.AspNetCore.Http.SameSiteMode
- Microsoft.Net.Http.Headers.SameSiteMode

- Microsoft.Net.Http.Headers.SetCookieHeaderValue.SameSite

MSBuild

- 设计时生成仅返回最上层包引用

设计时生成仅返回最上层包引用

从 .NET Core SDK 3.1.400 开始，`RunResolvePackageDependencies` 目标仅返回最上层包引用。

引入的版本

.NET Core SDK 3.1.400

更改描述

在早期版本的 .NET Core SDK 中，`RunResolvePackageDependencies` 目标创建了以下 MSBuild 项，其中包含 NuGet 资产文件中的信息：

- `PackageDefinitions`
- `PackageDependencies`
- `TargetDefinitions`
- `FileDefinitions`
- `FileDependencies`

Visual Studio 使用这些数据来填充解决方案资源管理器中的依赖项节点。但这些数据可能很大，除非对依赖项节点进行了扩展，否则不需要这些数据。

从 .NET Core SDK 3.1.400 版开始，默认不会生成这些项中的大多数。仅返回 `Package` 类型的项。如果 Visual Studio 需要这些项来填充依赖项节点，它会直接从资产文件中读取信息。

更改原因

引入此更改是为了改进 Visual Studio 内的解决方案加载性能。之前系统会加载所有的包引用，包括加载多数用户永远不会查看的许多引用。

建议的操作

如果你有依赖于所创建项的 MSBuild 逻辑，请在项目文件中将 `EmitLegacyAssetsFileItems` 属性设置为 `true`。此设置会启用以前的行为(这种行为将创建所有项)。

类别

MSBuild

受影响的 API

不可用

Windows 窗体

- 已删除的控件
- 如果显示工具提示，则不引发 CellFormatting 事件

已删除的控件

从 .NET Core 3.1 开始，某些 Windows 窗体控件不再可用。

更改描述

从 .NET Core 3.1 开始，各种 Windows 窗体控件不再可用。.NET Framework 2.0 中引入改进了设计和支持的替换控件。弃用的控件之前已从设计器工具箱中删除，但仍可供使用。

以下类型不再可用：

- ContextMenu

- DataGrid
- DataGrid.HitTestType
- DataGridBoolColumn
- DataGridCell
- DataGridColumnStyle
- DataGridLineStyle
- DataGridParentRowsLabelStyle
- DataGridPreferredColumnWidthTypeConverter
- DataGridTableStyle
- DataGridTextBox
- DataGridTextBoxColumn
- GridColumnStylesCollection
- GridTablesFactory
- GridTableStylesCollection
- IDataGridEditingService
- IMenuEditorService
- MainMenu
- Menu
- Menu.MenuItemCollection
- MenuItem
- ToolBar
- ToolBarAppearance
- ToolBarButton
- ToolBar.ToolBarButtonCollection
- ToolBarButtonClickEventArgs
- ToolBarButtonStyle
- ToolBarTextAlign

引入的版本

3.1

建议操作

每个已删除的控件都有一个推荐的替换控件。请参阅以下表：

[illegible] (API)	[illegible]	[illegible] API
ContextMenu	ContextMenuStrip	
DataGrid	DataGridView	DataGridCell、DataGridRow、DataGridTableCollection、DataGridColumnCollection、DataGridTableStyle、DataGridColumnStyle、DataGridLineStyle、DataGridParentRowsLabel、DataGridParentRowsLabelStyle、DataGridBoolColumn、DataGridTextBox、GridColumnStylesCollection、GridTableStylesCollection、HitTestType
MainMenu	MenuStrip	

[illegible] (API)	[illegible]	[illegible] API
菜单	ToolStripDropDown、ToolStripDropDownMenu	MenuItemCollection
MenuItem	ToolStripMenuItem	
ToolBar	ToolStrip	ToolBarAppearance
ToolBarButton	ToolStripButton	ToolBarButtonClickEventArgs、ToolBarButtonClickEventHandler、ToolBarButtonStyle、ToolBarTextAlign

类别

Windows 窗体

受影响的 API

- System.Windows.Forms.ContextMenu
- System.Windows.Forms.GridColumnStylesCollection
- System.Windows.Forms.GridTablesFactory
- System.Windows.Forms.GridTableStylesCollection
- System.Windows.Forms.IDataGridEditingService
- System.Windows.Forms.MainMenu
- System.Windows.Forms.Menu
- System.Windows.Forms.Menu.MenuItemCollection
- System.Windows.Forms.MenuItem
- System.Windows.Forms.ToolBar
- System.Windows.Forms.ToolBar.ToolBarButtonCollection
- System.Windows.Forms.ToolBarAppearance
- System.Windows.Forms.ToolBarButton
- System.Windows.Forms.ToolBarButtonClickEventArgs
- System.Windows.Forms.ToolBarButtonStyle
- System.Windows.Forms.ToolBarTextAlign
- System.Windows.Forms.DataGrid
- System.Windows.Forms.DataGrid.HitTestType
- System.Windows.Forms.DataGridBoolColumn
- System.Windows.Forms.DataGridCell
- System.Windows.Forms.DataGridColumnStyle
- System.Windows.Forms.DataGridLineStyle
- System.Windows.Forms.DataGridParentRowsLabelStyle
- System.Windows.Forms.DataGridPreferredColumnWidthTypeConverter
- System.Windows.Forms.DataGridTableStyle
- System.Windows.Forms.DataGridTextBox
- System.Windows.Forms.DataGridTextBoxColumn
- System.Windows.Forms.Design.IMenuEditorService

如果显示工具提示，则不引发 CellFormatting 事件

现在，当鼠标悬停和通过键盘选择时，DataGridView 将显示单元格的文本和错误工具提示。如果显示工具提示，则不会引发 DataGridView.CellFormatting 事件。

更改描述

在 .NET Core 3.1 之前，将 ShowCellToolTips 属性设置为 `true` 的 DataGridView 会在鼠标悬停在单元格上方时显示单元格文本和错误的工具提示。之前，通过键盘选择单元格时（例如通过使用 Tab 键、快捷键或箭头导航），不显示工具提示。如果用户编辑了单元格，然后在 DataGridView 仍处于编辑模式时将鼠标悬停在未设置 ToolTipText 属性的单元格上，则会引发 CellFormatting 事件，对要在单元格中显示的单元格文本进行格式化。

为满足辅助功能标准，自 .NET Core 3.1 起，将 ShowCellToolTips 属性设置为 `true` 的 DataGridView 不仅在鼠标悬停在单元格上时会显示单元格文本和错误的工具提示，而且在通过键盘选择单元格时也会显示。由于这一变更，如果鼠标在 DataGridView 处于编辑模式时悬停在未设置 ToolTipText 属性的单元格上，不会引发 CellFormatting 事件 。不引发该事件的原因是鼠标悬停的单元格的内容显示为工具提示，而不是显示在单元格中。

引入的版本

3.1

建议操作

当 DataGridView 处于编辑模式时，对依赖 CellFormatting 事件的所有代码进行重构。

类别

Windows 窗体

受影响的 API

None

.NET Core 3.0 的新增功能

2021/11/16 •

本文介绍了 .NET Core 3.0 中的新增功能。最大的增强功能之一是对 Windows 桌面应用程序的支持(仅限 Windows)。通过使用 .NET Core 3.0 SDK Windows 桌面组件，可移植 Windows 窗体和 Windows Presentation Foundation (WPF) 应用程序。明确地说，只有在 Windows 上才支持和包含 Windows 桌面组件。有关详细信息，请参阅本文后面的 Windows 桌面部分。

.NET Core 3.0 添加了对 C#8.0 的支持。强烈建议使用 Visual Studio 2019 版本 16.3 或更高版本、Visual Studio for Mac 8.3 或更高版本，或具有最新 C# 扩展的 Visual Studio Code。

立即在 Windows、macOS 或 Linux 上下载并开始使用 .NET Core 3.0。

有关版本的详细信息，请参阅 .NET Core 3.0 公告。

Microsoft 认为 .NET Core 3.0 RC 1 可用于生产环境，且该软件完全受支持。如果使用的是预览版本，则必须转换为 RTM 版本才能继续获得支持。

语言改进 C# 8.0

C# 8.0 也是该发布的一部分，包含可为空引用类型功能、异步流和更多模式。有关 C# 8.0 功能的详细信息，请参阅 C# 8.0 中的新增功能。

与 C# 8.0 语言功能相关的教程：

- 教程：使用可为空和不可为空引用类型更清晰地表达设计意图
- 教程：使用 C# 8.0 和 .NET Core 3.0 生成和使用异步流
- 教程：使用模式匹配来构建类型驱动和数据驱动的算法

添加了语言增强功能，以支持下面详细说明的 API 功能：

- 范围和索引
- 异步流

.NET Standard 2.1

.NET Core 3.0 已实现 .NET Standard 2.1。但是，默认的 `dotnet new classlib` 模板还是会生成一个面向 .NET Standard 2.0 的项目。若要面向 .NET Standard 2.1，请编辑项目文件并将 `TargetFramework` 属性更改为 `netstandard2.1`：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netstandard2.1</TargetFramework>
  </PropertyGroup>

</Project>
```

如果使用 Visual Studio，则需要 Visual Studio 2019，这是因为 Visual Studio 2017 不支持 .NET Standard 2.1 或 .NET Core 3.0 。

编译/部署

默认可执行文件

.NET Core 现在默认生成依赖于框架的可执行文件。对于使用全局安装的 .NET Core 版本的应用程序而言，这是一种新行为。以前，仅独立部署会生成可执行文件。

在 `dotnet build` 或 `dotnet publish` 期间，将创建一个与你使用的 SDK 的环境和平台相匹配的可执行文件(即 appHost)。和其他本机可执行文件一样，可以使用这些可执行文件执行相同操作，例如：

- 可以双击可执行文件。
- 可以直接从命令提示符启用应用程序，如 Windows 上的 `myapp.exe`，以及 Linux 和 macOS 上的 `./myapp`。

macOS appHost 和公证

仅 macOS

从已公证的适用于 macOS 的 .NET Core SDK 3.0 开始，默认已禁用用于生成默认可执行文件(即 appHost)的设置。有关详细信息，请参阅 macOS Catalina 公证以及对 .NET Core 下载和项目的影响。

启用 appHost 设置后，.NET Core 在生成或发布时将生成本机 Mach-O 可执行文件。如果使用 `dotnet run` 命令从源代码中运行应用，或通过启动 Mach-O 可执行文件直接运行应用，则应用会在 appHost 的上下文中运行。

如果没有 appHost，用户就只能使用 `dotnet <filename.dll>` 命令启动依赖框架的应用。发布独立应用时，始终会创建 appHost。

可以在项目级别配置 appHost，或通过 `-p:UseAppHost` 参数切换特定 `dotnet` 命令的 appHost：

- 项目文件

```
<PropertyGroup>
  <UseAppHost>true</UseAppHost>
</PropertyGroup>
```

- 命令行参数

```
dotnet run -p:UseAppHost=true
```

有关 `UseAppHost` 设置的详细信息，请参阅 Microsoft.NET.Sdk 的 MSBuild 属性。

单文件可执行文件

`dotnet publish` 命令支持将应用打包为特定于平台的单文件可执行文件。该可执行文件是自解压缩文件，包含运行应用所需的所有依赖项(包括本机依赖项)。首次运行应用时，应用程序将根据应用名称和生成标识符自解压缩到一个目录中。再次运行应用程序时，启动速度将变快。除非使用了新版本，否则应用程序无需再次进行自解压缩。

若要发布单文件可执行文件，请使用 `dotnet publish` 命令在项目或命令行中设置 `PublishSingleFile`：

```
<PropertyGroup>
  <RuntimeIdentifier>win10-x64</RuntimeIdentifier>
  <PublishSingleFile>true</PublishSingleFile>
</PropertyGroup>
```

- 或 -

```
dotnet publish -r win10-x64 -p:PublishSingleFile=true
```

有关单文件发布的详细信息，请参阅单文件捆绑程序设计文档。

程序集剪裁

.NET core 3.0 SDK 随附了一种工具，可以通过分析 IL 并剪裁未使用的程序集来减小应用的大小。

自包含应用包括运行代码所需的所有内容，而无需在主计算机上安装 .NET。但是，很多时候应用只需要一小部分框架即可运行，并且可以删除其他未使用的库。

.NET Core 现在包含一项将会使用 IL 剪裁器工具来扫描应用 IL 的设置。此工具将检测哪些代码是必需的，然后剪裁未使用的库。此工具可以显著减少某些应用的部署大小。

要启用此工具，请使用项目中的 `<PublishTrimmed>` 设置并发布自包含应用：

```
<PropertyGroup>
  <PublishTrimmed>true</PublishTrimmed>
</PropertyGroup>
```

```
dotnet publish -r <rid> -c Release
```

例如，包含的基本“hello world”新控制台项目模板在发布时命中大小约为 70 MB。通过使用 `<PublishTrimmed>`，其大小将减少到约 30 MB。

请务必考虑到使用反射或相关动态功能的应用程序或框架（包括 ASP.NET Core 和 WPF）通常会在剪裁时损坏。发生此损坏是因为剪裁器不了解此动态行为，并且不能确定反射需要哪些框架类型。可将 IL 剪裁器工具配置为感知这种情况。

最重要的是，剪裁后务必对应用进行测试。

有关 IL 剪裁器工具的详细信息，请参阅文档，或访问 mono/linker 存储库。

分层编译

.NET Core 3.0 中默认启用了分层编译 (TC)。此功能使运行时能够更适应地使用实时 (JIT) 编译器来实现更好的性能。

分层编译的主要优势是提供两种实现实时的方法，可在低质量快速层或高质量慢速层中编译。质量是指方法的优化程度。这有助于提高应用程序在从启动到稳定状态的各个执行阶段的性能。禁用分层编译后，每种方法都以同一种方式进行编译，这种方式倾向于牺牲启动性能来保证稳定状态性能。

启用 TC 后，以下行为适用于应用启动时的方法编译：

- 如果方法具有预先编译的代码 (ReadyToRun)，将使用预生成的代码。
- 否则，将实时编译该方法。一般来说，这些方法是泛型而不是值类型。
  - 快速 JIT 可以更快地生成较低质量（优化程度较低）的代码。在 .NET Core 3.0 中，默认为不包含循环的方法启用了快速 JIT，并且启动过程中首选快速 JIT。
  - 完全优化的 JIT 可生成更高质量（优化程度更高）的代码，但速度更慢。对于不使用快速 JIT 的方法（例如，如果该方法具有 MethodImplOptions.AggressiveOptimization 特性），则使用完全优化的 JIT。

对于频繁调用的方法，实时编译器最终会在后台创建完全优化的代码。然后，优化后的代码将替换该方法的预编译代码。

通过快速 JIT 生成的代码可能会运行较慢、分配更多内存或使用更多堆栈空间。如果出现问题，可以在项目文件中使用此 MSBuild 属性禁用快速 JIT：

```
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

若要完全禁用 TC，请在项目文件中使用此 MSBuild 属性：

```
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

> **TIP**
> 如果在项目文件中更改这些设置，则可能需要执行干净的生成以反映新的设置（删除 `obj` 和 `bin` 目录并重新生成）。

有关在运行时配置编译的详细信息，请参阅用于编译的运行时配置选项。

ReadyToRun 映像

可以通过将应用程序集编译为 ReadyToRun (R2R) 格式来改进.NET Core 应用程序的启动时间。R2R 是一种预先 (AOT) 编译形式。

R2R 二进制文件通过减少应用程序加载时实时 (JIT) 编译器需要执行的工作量来改进启动性能。二进制文件包含与 JIT 将生成的内容类似的本机代码。但是，R2R 二进制文件更大，因为它们包含中间语言 (IL) 代码（某些情况下仍需要此代码）和相同代码的本机版本。仅当发布面向特定运行时环境 (RID)（如 Linux x64 或 Windows x64）的自包含应用时 R2R 才可用。

若要将项目编译为 ReadyToRun，请执行以下操作：

1. 向项目中添加 `<PublishReadyToRun>` 设置：

   ```
   <PropertyGroup>
     <PublishReadyToRun>true</PublishReadyToRun>
   </PropertyGroup>
   ```

2. 发布自包含应用。例如，此命令将创建适用于 Windows 64 位版本的自包含应用：

   ```
   dotnet publish -c Release -r win-x64 --self-contained
   ```

跨平台/体系结构限制

ReadyToRun 编译器当前不支持跨目标。必须在给定的目标上编译。例如，如果想要 Windows x64 R2R 映像，需要在该环境中运行发布命令。

跨目标的例外情况：

- 可以使用 Windows x64 编译 Windows ARM32、ARM64 和 x86 映像。
- 可以使用 Windows x86 编译 Windows ARM32 映像。
- 可以使用 Linux x64 编译 Linux ARM32 和 ARM64 映像。

有关详细信息，请参阅准备好运行。

运行时/SDK

主要版本运行时前滚

.NET Core 3.0 引入了一项选择加入功能，该功能允许应用前滚到 .NET Core 最新的主要版本。此外，还添加了一项新设置来控制如何将前滚应用于应用。这可以通过以下方式配置：

- 项目文件属性：`RollForward`
- 运行时配置文件属性：`rollForward`
- 环境变量：`DOTNET_ROLL_FORWARD`
- 命令行参数：`--roll-forward`

必须指定以下值之一。如果省略该设置，则默认值为“Minor”。

- LatestPatch
  前滚到最高补丁版本。这会禁用次要版本前滚。
- Minor
  如果缺少所请求的次要版本，则前滚到最低的较高次要版本。如果存在所请求的次要版本，则使用 LatestPatch 策略。
- Major
  如果缺少所请求的主要版本，则前滚到最低的较高主要版本和最低的次要版本。如果存在所请求的主要版本，则使用 Minor 策略。
- LatestMinor
  即使存在所请求的次要版本，仍前滚到最高次要版本。适用于组件托管方案。
- LatestMajor
  即使存在所请求的主要版本，仍前滚到最高主要版本和最高次要版本。适用于组件托管方案。
- Disable
  不前滚。仅绑定到指定的版本。建议不要将此策略用于一般用途，因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

除“Disable”设置外，所有设置都将使用可用的最高补丁版本。

默认情况下，如果请求的版本（`.runtimeconfig.json` 中为应用程序指定的版本）为发布版本，则前滚时只考虑发布版本。将忽略所有预发布版本。如果没有匹配的发布版本，则会考虑使用预发布版本。可以通过设置 `DOTNET_ROLL_FORWARD_TO_PRERELEASE=1` 来更改此行为，在这种情况下，将始终考虑所有版本。

生成会复制依赖项

`dotnet build` 命令现在将应用程序的 NuGet 依赖项从 NuGet 缓存复制到生成输出文件夹。此前，依赖项仅作为 `dotnet publish` 的一部分复制。

有些操作（例如剪裁和 razor 页面发布）仍需要发布。

本地工具

.NET Core 3.0 引入了本地工具。本地工具类似于全局工具，但与磁盘上的特定位置相关联。本地工具在全局范围内不可用，并作为 NuGet 包进行分发。

> **WARNING**
> 如果尝试使用过 .NET Core 3.0 预览版 1 中的本地工具，例如运行 `dotnet tool restore` 或 `dotnet tool install`，请删除本地工具缓存文件夹。否则，本地工具将无法在任何较新的版本上运行。此文件夹位于：
>
> 在 macOS、Linux 上：`rm -r $HOME/.dotnet/toolResolverCache`
>
> 在 Windows 上：`rmdir /s %USERPROFILE%\.dotnet\toolResolverCache`

本地工具依赖于当前目录中名为 `dotnet-tools.json` 的清单文件。此清单文件定义在该文件夹和以下文件夹中可用的工具。你可以随代码一起分发清单文件，以确保使用代码的任何人都可以还原和使用相同的工具。

对于全局工具和本地工具，需要一个兼容的运行时版本。目前，NuGet.org 上的许多工具都面向 .NET Core Runtime 2.1。若要在全局范围或本地安装这些工具，仍需要安装 .NET Core 2.1 运行时。

新 global.json 选项

global.json 文件包含新选项，当你尝试定义所使用的 .NET Core SDK 版本时，这些选项可提供更大的灵活性。新选项包括：

- `allowPrerelease`：指示在选择要使用的 SDK 版本时，SDK 解析程序是否应考虑预发布版本。
- `rollForward`：指示选择 SDK 版本时要使用的前滚策略，可作为特定 SDK 版本缺失时的回退，或者作为使用更

高版本的指令。

有关这些更改的详细信息(包括默认值、支持的值和新的匹配规则),请参阅 global.json 概述。

垃圾回收堆大小减小

垃圾回收器的默认堆大小已减小,以使 .NET Core 使用更少的内存。此更改更符合具有现代处理器缓存大小的第 0 代分配预算。

垃圾回收大型页面支持

大型页面(也称为 Linux 上的巨型页面)是一项功能,其中操作系统能够建立大于本机页面大小(通常为 4K)的内存区域,以提高请求这些大型页面的应用程序的性能。

现在可以使用 **GCLargePages** 设置将垃圾回收器配置为一项选择加入功能,以选择在 Windows 上分配大型页面。

Windows 桌面和 COM

.NET Core SDK Windows Installer

用于 Windows 的 MSI 安装程序已从 .NET Core 3.0 开始更改。SDK 安装程序现在将对 SDK 功能区段版本进行就地升级。功能区段在版本号的 *补丁* 部分中的 *百数* 组中定义。例如,**3.0.101** 和 **3.0.201** 是两个不同功能区段中的版本,而 **3.0.101** 和 **3.0.199** 则属于同一个功能区段。并且,当安装 .NET Core SDK **3.0.101** 时,将从计算机中删除 .NET Core SDK **3.0.100**(如果存在)。当 .NET Core SDK **3.0.200** 安装在同一台计算机上时,不会删除 .NET Core SDK **3.0.101** 。

有关版本控制的详细信息,请参阅 .NET Core 的版本控制方式概述。

Windows 桌面

.NET Core 3.0 支持使用 Windows Presentation Foundation (WPF) 和 Windows 窗体的 Windows 桌面应用程序。这些框架还支持通过 XAML 岛从 Windows UI XAML 库 (WinUI) 使用新式控件和 Fluent 样式。

Windows 桌面部件是 Windows .NET Core 3.0 SDK 的一部分。

可以使用以下 `dotnet` 命令创建新的 WPF 或 Windows 窗体应用:

```
dotnet new wpf
dotnet new winforms
```

Visual Studio 2019 添加了适用于 .NET Core 3.0 Windows 窗体和 WPF 的"新建项目"模板。

有关如何移植现有 .NET Framework 应用程序的详细信息,请参阅移植 WPF 项目和移植 Windows 窗体项目。

WinForms 高 DPI

.NET Core Windows 窗体应用程序可以使用 Application.SetHighDpiMode(HighDpiMode) 设置高 DPI 模式。`SetHighDpiMode` 方法可设置相应的高 DPI 模式,除非该设置已通过其他方式(例如使用 `App.Manifest` 或在 `Application.Run` 前面使用 P/Invoke)进行设置。

由 System.Windows.Forms.HighDpiMode 枚举表示的可能的 `highDpiMode` 值包括:

- `DpiUnaware`
- `SystemAware`
- `PerMonitor`
- `PerMonitorV2`
- `DpiUnawareGdiScaled`

有关高 DPI 模式的详细信息,请参阅在 Windows 上开发高 DPI 桌面应用程序。

创建 COM 组件

在 Windows 上，现在可以创建可调用 COM 的托管组件。在将 .NET Core 与 COM 加载项模型结合使用，以及使用 .NET Framework 提供奇偶校验时，此功能至关重要。

与将 *mscoree.dll* 用作 COM 服务器的 .NET Framework 不同，.NET Core 将在生成 COM 组件时向 *bin* 目录添加本机启动程序 dll。

有关如何创建 COM 组件并使用它的示例，请参阅 COM 演示。

Windows 本机互操作

Windows 提供丰富的本机 API，包括平面 C API、COM 和 WinRT 的形式。.NET Core 支持 **P/Invoke**，.NET Core 3.0 则增加了 **CoCreate COM API** 和 **Activate WinRT API** 的功能。有关代码示例，请参阅 Excel 演示。

MSIX 部署

MSIX 是新的 Windows 应用程序包格式。可以使用它将 .NET Core 3.0 桌面应用程序部署到 Windows 10。

使用 Visual Studio 2019 中的 Windows 应用打包项目，可以创建包含独立式 .NET Core 应用的 MSIX 包。

.NET Core 项目文件必须在 `<RuntimeIdentifiers>` 属性中指定支持的运行时：

```
<RuntimeIdentifiers>win-x86;win-x64</RuntimeIdentifiers>
```

Linux 改进

适用于 Linux 的 SerialPort

.NET Core 3.0 提供对 Linux 上 System.IO.Ports.SerialPort 的基本支持。

以前，.NET Core 只支持在 Windows 上使用 `SerialPort`。

有关对 Linux 上串行端口有限支持的详细信息，请参阅 GitHub 问题 #33146。

Docker 和 cgroup 内存限制

在 Linux 上使用 Docker 运行 .NET Core 3.0 时，可更好地应对 cgroup 内存限制。运行具有内存限制的 Docker 容器（例如使用 `docker run -m`）会更改 .NET Core 的行为方式。

- 默认垃圾回收器 (GC) 堆大小：最大为 20 MB 或容器内存限制的 75%。
- 可以将显式大小设置为绝对数或 cgroup 限制的百分比。
- 每个 GC 堆的最小保留段大小为 16 MB。此大小可减少在计算机上创建的堆数量。

对 Raspberry Pi 的 GPIO 支持

已向 NuGet 发布了两个可用于 GPIO 编程的包：

- System.Device.Gpio
- Iot.Device.Bindings

GPIO 包包括用于 *GPIO*、*SPI*、*I2C* 和 *PWM* 设备的 API。IoT 绑定包包括设备绑定。有关详细信息，请参阅设备 GitHub 存储库。

ARM64 Linux 支持

.NET Core 3.0 增加了对 ARM64 for Linux 的支持。ARM64 的主要用例是当前的 IoT 场景。有关详细信息，请参阅 .NET Core ARM64 状态。

ARM64 上适用于 .NET Core 的 Docker 映像可用于 Alpine、Debian 和 Ubuntu。

> **NOTE**
> ARM64 尚未提供 Windows 支持。

安全性

Linux 上的 TLS 1.3 和 OpenSSL 1.1.1

.NET Core 现在可以在给定环境中使用 OpenSSL 1.1.1 中的 TLS 1.3 支持。使用 TLS 1.3：

- 通过减少客户端和服务器之间所需的往返次数，提高了连接时间。
- 由于删除了各种过时和不安全的加密算法，提高了安全性。

.NET Core 3.0 在 Linux 系统上使用 OpenSSL 1.1.1、OpenSSL 1.1.0 或 OpenSSL 1.0.2（如果可用）。当 OpenSSL 1.1.1 可用时，System.Net.Security.SslStream 和 System.Net.Http.HttpClient 类型都将使用 TLS 1.3（假定客户端和服务器都支持 TLS 1.3）。

> **IMPORTANT**
> Windows 和 macOS 尚不支持 TLS 1.3。

下面的 C# 8.0 示例演示在 Ubuntu 18.10 上 .NET Core 3.0 如何连接到 https://www.cloudflare.com：

```
using System;
using System.Net.Security;
using System.Net.Sockets;
using System.Threading.Tasks;

namespace whats_new
{
    public static class TLS
    {
        public static async Task ConnectCloudFlare()
        {
            var targetHost = "www.cloudflare.com";

            using TcpClient tcpClient = new TcpClient();

            await tcpClient.ConnectAsync(targetHost, 443);

            using SslStream sslStream = new SslStream(tcpClient.GetStream());

            await sslStream.AuthenticateAsClientAsync(targetHost);
            await Console.Out.WriteLineAsync($"Connected to {targetHost} with {sslStream.SslProtocol}");
        }
    }
}
```

加密密码

.NET Core 3.0 增加了对 AES-GCM 和 AES-CCM 密码的支持（分别使用 System.Security.Cryptography.AesGcm 和 System.Security.Cryptography.AesCcm 实现）。这些算法都是用于关联数据的认证加密 (AEAD) 算法。

下面的代码演示了如何使用 `AesGcm` 密码加密和解密随机数据。

```
using System;
using System.Linq;
using System.Security.Cryptography;

namespace whats_new
{
    public static class Cipher
    {
        public static void Run()
        {
            // key should be: pre-known, derived, or transported via another channel, such as RSA encryption
            byte[] key = new byte[16];
            RandomNumberGenerator.Fill(key);

            byte[] nonce = new byte[12];
            RandomNumberGenerator.Fill(nonce);

            // normally this would be your data
            byte[] dataToEncrypt = new byte[1234];
            byte[] associatedData = new byte[333];
            RandomNumberGenerator.Fill(dataToEncrypt);
            RandomNumberGenerator.Fill(associatedData);

            // these will be filled during the encryption
            byte[] tag = new byte[16];
            byte[] ciphertext = new byte[dataToEncrypt.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Encrypt(nonce, dataToEncrypt, ciphertext, tag, associatedData);
            }

            // tag, nonce, ciphertext, associatedData should be sent to the other part

            byte[] decryptedData = new byte[ciphertext.Length];

            using (AesGcm aesGcm = new AesGcm(key))
            {
                aesGcm.Decrypt(nonce, ciphertext, tag, decryptedData, associatedData);
            }

            // do something with the data
            // this should always print that data is the same
            Console.WriteLine($"AES-GCM: Decrypted data is {(dataToEncrypt.SequenceEqual(decryptedData) ?
"the same as" : "different than")} original data.");
        }
    }
}
```

加密密钥导入/导出

.NET Core 3.0 支持从标准格式导入和导出非对称公钥和私钥。你不需要使用 X.509 证书。

所有密钥类型（例如 *RSA*、*DSA*、*ECDsa* 和 *ECDiffieHellman*）都支持以下格式：

- **公钥**
  - X.509 SubjectPublicKeyInfo
- **私钥**
  - PKCS#8 PrivateKeyInfo
  - PKCS#8 EncryptedPrivateKeyInfo

RSA 密钥还支持：

- **公钥**
  - PKCS#1 RSAPublicKey
- **私钥**
  - PKCS#1 RSAPrivateKey

导出方法生成 DER 编码的二进制数据，导入方法也是如此。如果密钥以文本友好的 PEM 格式存储，调用方需要在调用导入方法之前对内容进行 base64 解码。

```
using System;
using System.Security.Cryptography;

namespace whats_new
{
    public static class RSATest
    {
        public static void Run(string keyFile)
        {
            using var rsa = RSA.Create();

            byte[] keyBytes = System.IO.File.ReadAllBytes(keyFile);
            rsa.ImportRSAPrivateKey(keyBytes, out int bytesRead);

            Console.WriteLine($"Read {bytesRead} bytes, {keyBytes.Length - bytesRead} extra byte(s) in
file.");
            RSAParameters rsaParameters = rsa.ExportParameters(true);
            Console.WriteLine(BitConverter.ToString(rsaParameters.D));
        }
    }
}
```

可以使用 System.Security.Cryptography.Pkcs.Pkcs8PrivateKeyInfo 检查 PKCS#8 文件，使用 System.Security.Cryptography.Pkcs.Pkcs12Info 检查 PFX/PKCS#12 文件。可以使用 System.Security.Cryptography.Pkcs.Pkcs12Builder 操作 PFX/PKCS#12 文件。

.NET Core 3.0 API 改动

范围和索引

新 System.Index 类型可用于编制索引。可从 `int` 创建一个从开头开始计数的索引，也可使用前缀 `^` 运算符 (C#) 创建一个从末尾开始计数的索引：

```
Index i1 = 3;  // number 3 from beginning
Index i2 = ^4; // number 4 from end
int[] a = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
Console.WriteLine($"{a[i1]}, {a[i2]}"); // "3, 6"
```

此外，还有 System.Range 类型，它包含两个 `Index` 值，一个用于开头一个用于结尾，可以使用 `x..y` 范围表达式 (C#) 进行编写。然后可以使用 `Range` 编制索引，以便生成一个切片：

```
var slice = a[i1..i2]; // { 3, 4, 5 }
```

有关详细信息，请参阅范围和索引教程。

异步流

IAsyncEnumerable<T> 类型是 IEnumerable<T> 的新异步版本。通过该语言，可通过 `IAsyncEnumerable<T>` 执行 `await foreach` 操作来使用其元素，并对其使用 `yield return` 操作来生成元素。

下面的示例演示如何生成和使用异步流。`foreach` 语句为异步语句，它本身使用 `yield return` 为调用方生成异步流。此模式（使用 `yield return`）是生成异步流的建议模型。

```
async IAsyncEnumerable<int> GetBigResultsAsync()
{
    await foreach (var result in GetResultsAsync())
    {
        if (result > 20) yield return result;
    }
}
```

除了能够 `await foreach`，还可以创建异步迭代器，例如，一个返回 `IAsyncEnumerable/IAsyncEnumerator` 的迭代器，可以在其中进行 `await` 和 `yield` 操作。对于需要处理的对象，可以使用各种 BCL 类型（如 `Stream` 和 `Timer`）实现的 `IAsyncDisposable`。

有关详细信息，请参阅异步流教程。

IEEE 浮点

正在更新浮点 API，以符合 IEEE 754-2008 修订。这些更改旨在公开所有 **必需** 操作并确保这些操作在行为上符合 IEEE 规范。有关浮点改进的详细信息，请参阅 .NET Core 3.0 中的浮点分析和格式化改进博客文章。

分析和格式化修复包括：

- 正确分析并舍入任何输入长度。
- 正确分析并格式化负零。
- 通过执行不区分大小写的检查并允许在前面使用可选的 `+`（如果适用），正确分析 `Infinity` 和 `NaN`。

新的 System.Math API 包括：

- BitIncrement(Double) 和 BitDecrement(Double)
  相当于 `nextUp` 和 `nextDown` IEEE 运算。它们将返回最小的浮点数，该数字大于或小于输入值（分别）。例如，`Math.BitIncrement(0.0)` 将返回 `double.Epsilon`。
- MaxMagnitude(Double, Double) 和 MinMagnitude(Double, Double)
  相当于 `maxNumMag` 和 `minNumMag` IEEE 运算，它们将（分别）返回大于或小于两个输入的量值的值。例如，`Math.MaxMagnitude(2.0, -3.0)` 将返回 `-3.0`。
- ILogB(Double)
  相当于返回整数值的 `logB` IEEE 运算，它将返回输入参数的整数对数（以 2 为底）。此方法实际上与 `floor(log2(x))` 相同，但完成后出现最小舍入错误。
- ScaleB(Double, Int32)
  相当于采用整数值的 `scaleB` IEEE 运算，它实际返回 `x * pow(2, n)`，但完成后出现最小舍入错误。
- Log2(Double)
  相当于返回（以 2 为底）对数的 `log2` IEEE 运算。它会最小化舍入错误。
- FusedMultiplyAdd(Double, Double, Double)
  相当于执行乘法加法混合的 `fma` IEEE 运算。也就是说，它以单个运算的形式执行 `(x * y) + z`，从而最小化舍入错误。例如，`FusedMultiplyAdd(1e308, 2.0, -1e308)` 返回 `1e308`。常规 `(1e308 * 2.0) - 1e308` 返回 `double.PositiveInfinity`。
- CopySign(Double, Double)
  相当于 `copySign` IEEE 运算，它返回 `x` 的值但带有符号 `y`。

.NET 平台相关内部函数

已添加 API，允许访问某些性能导向的 CPU 指令，例如 SIMD 或位操作指令集 。这些指令有助于在某些情况下

实现显著的性能改进，例如高效地并行处理数据。

在适当的情况下，.NET 库已开始使用这些指令来改进性能。

有关详细信息，请参阅 .NET Platform Dependent Intrinsics（.NET 平台相关内部函数）。

改进的 .NET Core 版本 API

从 .NET Core 3.0 开始，.NET Core 提供的版本 API 现在可以返回你预期的信息。例如：

```
System.Console.WriteLine($"Environment.Version: {System.Environment.Version}");

// Old result
//   Environment.Version: 4.0.30319.42000
//
// New result
//   Environment.Version: 3.0.0
```

```
System.Console.WriteLine($"RuntimeInformation.FrameworkDescription:
{System.Runtime.InteropServices.RuntimeInformation.FrameworkDescription}");

// Old result
//   RuntimeInformation.FrameworkDescription: .NET Core 4.6.27415.71
//
// New result (notice the value includes any preview release information)
//   RuntimeInformation.FrameworkDescription: .NET Core 3.0.0-preview4-27615-11
```

> **WARNING**
> 重大变更。这在技术上是一个中断性变更，因为版本控制方案已发生变化。

内置的快速 JSON 支持

.NET 用户在很大程度上依赖于 Newtonsoft.Json 和其他常用的 JSON 库，它们仍是很好的选择。`Newtonsoft.Json` 使用 .NET 字符串作为其基本数据类型，它实际上是 UTF-16。

新的内置 JSON 支持具有高性能、低分配的特点，并且可用于 UTF-8 编码的 JSON 文本。有关 System.Text.Json 命名空间和类型的详细信息，请参阅以下文章：

- .NET 中的 JSON 序列化 - 概述
- 如何在 .NET 中对 JSON 数据进行序列化和反序列化。
- 如何从 Newtonsoft.Json 迁移到 System.Text.Json

HTTP/2 支持

System.Net.Http.HttpClient 类型支持 HTTP/2 协议。如果启用 HTTP/2，则将通过 TLS/ALPN 协商 HTTP 协议版本，并在服务器选择使用 HTTP/2 时使用。

默认协议将保留 HTTP/1.1，但可以在两种不同方法中启用 HTTP/2。首先，可以将 HTTP 请求消息设置为使用 HTTP/2：

```
var client = new HttpClient() { BaseAddress = new Uri("https://localhost:5001") };

// HTTP/1.1 request
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);

// HTTP/2 request
using (var request = new HttpRequestMessage(HttpMethod.Get, "/") { Version = new Version(2, 0) })
using (var response = await client.SendAsync(request))
    Console.WriteLine(response.Content);
```

其次，可以更改 HttpClient 以默认使用 HTTP/2：

```
var client = new HttpClient()
{
    BaseAddress = new Uri("https://localhost:5001"),
    DefaultRequestVersion = new Version(2, 0)
};

// HTTP/2 is default
using (var response = await client.GetAsync("/"))
    Console.WriteLine(response.Content);
```

很多时候，在你开发应用程序时要使用未加密的连接。如果你知道目标终结点将使用 HTTP/2，你可以为 HTTP/2 打开未加密的连接。可以通过将 `DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANDLER_HTTP2UNENCRYPTEDSUPPORT` 环境变量设置为 `1` 或通过在应用上下文中启用它来将其打开：

```
AppContext.SetSwitch("System.Net.Http.SocketsHttpHandler.Http2UnencryptedSupport", true);
```

后续步骤

- 查看 .NET Core 2.2 和 3.0 之间的重大变更。
- 查看用于 Windows 窗体应用的 .NET Core 3.0 中的中断性变更。

.NET Core 3.0 中的中断性变更

2021/11/16 •

若要迁移到 3.0 版 .NET Core、ASP.NET Core 或 EF Core，本文中列出的中断性变更可能会影响到你的应用。

ASP.NET Core

- 已删除过时防伪、CORS、诊断、MVC 和路由 API
- 已删除“Pubternal”API
- 身份验证：Google+ 弃用
- 身份验证：已删除 HttpContext.Authentication 属性
- 身份验证：已替换 Newtonsoft.json 类型
- 身份验证：已更改 OAuthHandler ExchangeCodeAsync 签名
- 授权：AddAuthorization 重载已移到不同的程序集
- 授权：已从 AuthorizationFilterContext.Filters 中删除 IAllowAnonymous
- 授权：IAuthorizationPolicyProvider 实现需要新方法
- 缓存：已删除 CompactOnMemoryPressure 属性
- 缓存：Microsoft.Extensions.Caching.SqlServer 使用新的 SqlClient 包
- 缓存：ResponseCaching“Pubternal”类型已更改为内部类型
- 数据保护：DataProtection.Blobs 使用新的 Azure 存储 API
- 托管：已从 Windows 托管捆绑包中删除 AspNetCoreModule V1
- 托管：通用主机限制 Startup 构造函数注入
- 托管：已为 IIS 进程外应用启用 HTTPS 重定向
- 托管：已替换 IHostingEnvironment 和 IApplicationLifetime 类型
- 托管：已从 WebHostBuilder 依赖项中删除 ObjectPoolProvider
- HTTP：已删除 DefaultHttpContext 扩展性
- HTTP：HeaderNames 字段已更改为静态只读
- HTTP：响应正文基础结构更改
- HTTP：已更改某些 Cookie SameSite 默认值
- HTTP：已默认禁用同步 IO
- 标识：已删除 AddDefaultUI 方法重载
- 标识：UI 启动版本更改
- 标识：对于未经身份验证的标识，SignInAsync 会引发异常
- 标识：SignInManager 构造函数接受新参数
- 标识：UI 使用静态 Web 资产功能
- Kestrel：已删除连接适配器
- Kestrel：已删除空 HTTPS 程序集
- Kestrel：请求尾部标头已移到新集合
- Kestrel：传输抽象层更改
- 本地化：API 已标记为已过时
- 日志记录：已将 DebugLogger 类设为内部类
- MVC：已删除控制器操作 Async 后缀
- MVC：JsonResult 已移至 Microsoft.AspNetCore.Mvc.Core
- MVC：已弃用预编译工具
- MVC：类型已更改为内部
- MVC：已删除 Web API 兼容性填充码
- Razor：已删除 RazorTemplateEngine API
- Razor：运行时编译已移到包
- 会话状态：已删除过时的 API
- 共享框架：已从 Microsoft.AspNetCore.App 中删除程序集
- 共享框架：已删除 Microsoft.AspNetCore.All
- SignalR：已替换 HandshakeProtocol.SuccessHandshakeData
- SignalR：已删除 HubConnection 方法
- SignalR：已更改 HubConnectionContext 构造函数
- SignalR：JavaScript 客户端包名称更改
- SignalR：过时的 API
- SPA：SpaServices 和 NodeServices 已标记为过时
- SPA：SpaServices 和 NodeServices 控制台记录器回退默认更改
- 目标框架：不支持 .NET Framework

已删除过时防伪、CORS、诊断、MVC 和路由 API

删除了 ASP.NET Core 2.2 中的过时成员和兼容性开关。

引入的版本

3.0

更改原因

随着时间的推移，API 图面会得到改进。

建议操作

针对 .NET Core 2.2 时，请遵循过时生成消息中的指导来改为采用新 API。

类别

ASP.NET Core

受影响的 API

以下类型和成员标记为对 ASP.NET Core 2.1 和 2.2 过时：

类型

- `Microsoft.AspNetCore.Diagnostics.Views.WelcomePage`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.AttributeValue`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.BaseView`
- `Microsoft.AspNetCore.DiagnosticsViewPage.Views.HelperResult`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.ValidationProblemDetails21Wrapper`
- `Microsoft.AspNetCore.Mvc.Razor.Compilation.ViewsFeatureProvider`
- `Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageArgumentBinder`
- `Microsoft.AspNetCore.Routing.IRouteValuesAddressMetadata`
- `Microsoft.AspNetCore.Routing.RouteValuesAddressMetadata`

构造函数

- `Microsoft.AspNetCore.Cors.Infrastructure.CorsService(IOptions{CorsOptions})`
- `Microsoft.AspNetCore.Routing.Tree.TreeRouteBuilder(ILoggerFactory,UrlEncoder,ObjectPool{UriBuildingContext},IInlineConstraintResolver)`
- `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatterCanWriteContext`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions},IInlineConstraintResolver,IModelMetadataProvider)`
- `Microsoft.AspNetCore.Mvc.ApiExplorer.DefaultApiDescriptionProvider(IOptions{MvcOptions},IInlineConstraintResolver,IModelMetadataProvider,IAc`
- `Microsoft.AspNetCore.Mvc.Formatters.FormatFilter(IOptions{MvcOptions})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ArrayModelBinder`1(IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ByteArrayModelBinder`
- Microsoft.AspNetCore.Mvc.ModelBinding.Binders.CollectionModelBinder`1(IModelBinder)
- Microsoft.AspNetCore.Mvc.ModelBinding.Binders.ComplexTypeModelBinder(IDictionary`2)
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DictionaryModelBinder`2(IModelBinder,IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.DoubleModelBinder(System.Globalization.NumberStyles)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FloatModelBinder(System.Globalization.NumberStyles)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormCollectionModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.FormFileModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.HeaderModelBinder`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.KeyValuePairModelBinder`2(IModelBinder,IModelBinder)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Binders.SimpleTypeModelBinder(System.Type)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes(IEnumerable{System.Object})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelAttributes(IEnumerable{System.Object},IEnumerable{System.Object})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ModelBinderFactory(IModelMetadataProvider,IOptions{MvcOptions})`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder(IModelMetadataProvider,IModelBinderFactory,IObjectModelValidator)`
- Microsoft.AspNetCore.Mvc.Routing.KnownRouteValueConstraint()
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter(System.Boolean)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlDataContractSerializerInputFormatter(MvcOptions)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter(System.Boolean)`
- `Microsoft.AspNetCore.Mvc.Formatters.XmlSerializerInputFormatter(MvcOptions)`
- Microsoft.AspNetCore.Mvc.TagHelpers.ImageTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,IUrlHelperFactory)

- `Microsoft.AspNetCore.Mvc.TagHelpers.LinkTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,JavaScriptEncoder,IUrlHelperFactory)`
- `Microsoft.AspNetCore.Mvc.TagHelpers.ScriptTagHelper(IHostingEnvironment,IMemoryCache,HtmlEncoder,JavaScriptEncoder,IUrlHelperFactory)`
- `Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.RazorPageAdapter(RazorPageBase)`

属性

- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieDomain`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookieName`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.CookiePath`
- `Microsoft.AspNetCore.Antiforgery.AntiforgeryOptions.RequireSsl`
- `Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.AllowInferringBindingSourceForCollectionTypesAsFromQuery`
- `Microsoft.AspNetCore.Mvc.ApiBehaviorOptions.SuppressUseValidationProblemDetailsForInvalidModelStateResponses`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.CookieName`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Domain`
- `Microsoft.AspNetCore.Mvc.CookieTempDataProviderOptions.Path`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.MvcDataAnnotationsLocalizationOptions.AllowDataAnnotationsLocalizationForEnumDisplayAttributes`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.MvcXmlOptions.AllowRfc7807CompliantProblemDetailsFormat`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowBindingHeaderValuesToNonStringModelTypes`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowCombiningAuthorizeFilters`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowShortCircuitingValidationWhenNoValidatorsArePresent`
- `Microsoft.AspNetCore.Mvc.MvcOptions.AllowValidatingTopLevelNodes`
- `Microsoft.AspNetCore.Mvc.MvcOptions.InputFormatterExceptionPolicy`
- `Microsoft.AspNetCore.Mvc.MvcOptions.SuppressBindingUndefinedValueToEnumType`
- `Microsoft.AspNetCore.Mvc.MvcViewOptions.AllowRenderingMaxLengthAttribute`
- `Microsoft.AspNetCore.Mvc.MvcViewOptions.SuppressTempDataAttributePrefix`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowAreas`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowDefaultHandlingForOptionsRequests`
- `Microsoft.AspNetCore.Mvc.RazorPages.RazorPagesOptions.AllowMappingHeadRequestsToGetHandler`

方法

- `Microsoft.AspNetCore.Mvc.LocalRedirectResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToActionResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToPageResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.RedirectToRouteResult.ExecuteResult(ActionContext)`
- `Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext,IValueProvider,ParameterDescriptor)`
- Microsoft.AspNetCore.Mvc.ModelBinding.ParameterBinder.BindModelAsync(ActionContext,IValueProvider,ParameterDescriptor,Object)

已删除“Pubternal”API

为了更好地维护 ASP.NET Core 面向公众的 API，`*.Internal` 命名空间中的大多数类型（称为 "pubternal" API）已成为真正的内部类型。这些命名空间中的成员永远不会作为面向公众的 API 得到支持。API 可能在次要版本中中断（这种情况经常会发生）。在更新到 ASP.NET Core 3.0 时，依赖于这些 API 的代码会中断。

有关详细信息，请参阅 dotnet/aspnetcore#4932 和 dotnet/aspnetcore#11312。

引入的版本

3.0

旧行为

受影响的 API 使用 `public` 访问修饰符进行标记，并存在于 `*.Internal` 命名空间中。

新行为

受影响的 API 使用 internal 访问修饰符进行标记，不能再供该程序集外部的代码使用。

更改原因

对于这些 "pubternal" API，指导原则是：

- 它们可能会更改，恕不另行通知。
- 它们不遵从 .NET 策略来防止中断性变更。

保留这些 API `public`（即使保留在 `*.Internal` 命名空间中）会对客户造成混淆。

建议操作

停止使用这些 "pubternal" API。如果对其他 API 有任何疑问，请在 dotnet/aspnetcore 存储库中提问。

例如，请考虑 ASP.NET Core 2.2 项目中的以下 HTTP 请求缓冲代码。`EnableRewind` 扩展方法存在于

`Microsoft.AspNetCore.Http.Internal` 命名空间中。

```
HttpContext.Request.EnableRewind();
```

在 ASP.NET Core 3.0 项目中，将 `EnableRewind` 调用替换为对 `EnableBuffering` 扩展方法的调用。请求缓冲功能的工作方式与过去相同。`EnableBuffering` 立即调用 `internal` API。

```
HttpContext.Request.EnableBuffering();
```

类别

ASP.NET Core

受影响的 API

名称中具有 `Internal` 段的 `Microsoft.AspNetCore.*` 和 `Microsoft.Extensions.*` 命名空间中的所有 API。例如：

- `Microsoft.AspNetCore.Authentication.Internal`
- `Microsoft.AspNetCore.Builder.Internal`
- `Microsoft.AspNetCore.DataProtection.Cng.Internal`
- `Microsoft.AspNetCore.DataProtection.Internal`
- `Microsoft.AspNetCore.Hosting.Internal`
- `Microsoft.AspNetCore.Http.Internal`
- `Microsoft.AspNetCore.Mvc.Core.Infrastructure`
- `Microsoft.AspNetCore.Mvc.Core.Internal`
- `Microsoft.AspNetCore.Mvc.Cors.Internal`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Json.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal`
- `Microsoft.AspNetCore.Mvc.Internal`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Internal`
- `Microsoft.AspNetCore.Mvc.Razor.Internal`
- `Microsoft.AspNetCore.Mvc.RazorPages.Internal`
- `Microsoft.AspNetCore.Mvc.TagHelpers.Internal`
- `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal`
- `Microsoft.AspNetCore.Rewrite.Internal`
- `Microsoft.AspNetCore.Routing.Internal`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Http`
- `Microsoft.AspNetCore.Server.Kestrel.Core.Internal.Infrastructure`
- `Microsoft.AspNetCore.Server.Kestrel.Https.Internal`

身份验证：Google+ 已弃用并被替换

Google 早在 2019 年 1 月 28 日就开始关闭对应用使用 Google + 登录。

更改描述

ASP.NET 4.x 和 ASP.NET Core 已使用 Google + 登录 API 在 Web 应用中对 Google 帐户用户进行身份验证。受影响的 NuGet 包为 Microsoft.AspNetCore.Authentication.Google（对于 ASP.NET Core）和 Microsoft.Owin.Security.Google（对于具有 ASP.NET Web Forms 和 MVC 的 `Microsoft.Owin`）。

Google 的替代 API 使用不同的数据源和格式。下面提供的缓解措施和解决方案说明存在结构性更改。应用应验证数据本身是否仍然满足其需求。例如，名称、电子邮件地址、配置文件链接和个人资料照片的值可能与以前稍有不同。

引入的版本

所有版本。此更改是 ASP.NET Core 的外部更改。

建议操作

包含 ASP.NET Web Forms 和 MVC 的 Owin

针对 `Microsoft.Owin` 3.1.0 和更高版本，本文概述了临时的缓解措施。应用应通过缓解措施来完成测试，以检查数据格式的更改。计划发布 `Microsoft.Owin` 4.0.1，并提供一个修补程序。使用任何较低版本的应用都应更新到版本 4.0.1。

ASP.NET Core 1.x

包含 ASP.NET Web Form 和MVC 的 Owin的缓解措施可应用于 ASP.NET Core 1.x。未计划 NuGet 包修补程序，因为 1.x 已处于生命周期结束状态。

ASP.NET Core 2.x

对于 `Microsoft.AspNetCore.Authentication.Google` 版本 2.x，请将对 `Startup.ConfigureServices` 中 `AddGoogle` 的现有调用替换为以下代码：

```
.AddGoogle(o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.UserInformationEndpoint = "https://www.googleapis.com/oauth2/v2/userinfo";
    o.ClaimActions.Clear();
    o.ClaimActions.MapJsonKey(ClaimTypes.NameIdentifier, "id");
    o.ClaimActions.MapJsonKey(ClaimTypes.Name, "name");
    o.ClaimActions.MapJsonKey(ClaimTypes.GivenName, "given_name");
    o.ClaimActions.MapJsonKey(ClaimTypes.Surname, "family_name");
    o.ClaimActions.MapJsonKey("urn:google:profile", "link");
    o.ClaimActions.MapJsonKey(ClaimTypes.Email, "email");
});
```

2 月 2.1 和 2.2 修补程序将前面的重新配置合并为新的默认配置。不会为 ASP.NET Core 2.0 计划任何修补程序，因为它的生命周期已经结束。

ASP.NET Core 3.0

为 ASP.NET Core 2.x 提供的缓解措施也可用于 ASP.NET Core 3.0。未来的 3.0 预览版中，可能会删除 `Microsoft.AspNetCore.Authentication.Google` 包。改为将用户定向到 `Microsoft.AspNetCore.Authentication.OpenIdConnect`。以下代码演示了如何在 `Startup.ConfigureServices` 中将 `AddGoogle` 替换为 `AddOpenIdConnect`。此替换可以用于 ASP.NET Core 2.0 及更高版本，并且可以根据需要应用于 ASP.NET Core 1.x。

```
.AddOpenIdConnect("Google", o =>
{
    o.ClientId = Configuration["Authentication:Google:ClientId"];
    o.ClientSecret = Configuration["Authentication:Google:ClientSecret"];
    o.Authority = "https://accounts.google.com";
    o.ResponseType = OpenIdConnectResponseType.Code;
    o.CallbackPath = "/signin-google"; // Or register the default "/signin-oidc"
    o.Scope.Add("email");
});
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();
```

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Authentication.Google

身份验证：已删除 HttpContext.Authentication 属性

已删除 `HttpContext` 上弃用的 `Authentication` 属性。

更改描述

作为 dotnet/aspnetcore#6504 的一部分，已删除 `HttpContext` 上弃用的 `Authentication` 属性。从 2.0 开始，`Authentication` 属性已弃用。迁移指南 已发布，可使用此弃用属性将代码迁移到新的替换 API。在 commit dotnet/aspnetcore@d7a7c65 中移除了与旧 ASP.NET Core 1.x 身份验证堆栈相关的其余未使用的类/API。

有关讨论，请参阅 dotnet/aspnetcore#6533。

引入的版本

3.0

更改原因

ASP.NET Core 1.0 API 已被 Microsoft.AspNetCore.Authentication.AuthenticationHttpContextExtensions 中的扩展方法替换。

建议的操作

请参阅迁移指南。

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Http.Authentication.AuthenticateInfo
- Microsoft.AspNetCore.Http.Authentication.AuthenticationManager
- Microsoft.AspNetCore.Http.Authentication.AuthenticationProperties
- Microsoft.AspNetCore.Http.Features.Authentication.AuthenticateContext
- Microsoft.AspNetCore.Http.Features.Authentication.ChallengeBehavior
- Microsoft.AspNetCore.Http.Features.Authentication.ChallengeContext
- Microsoft.AspNetCore.Http.Features.Authentication.DescribeSchemesContext
- Microsoft.AspNetCore.Http.Features.Authentication.IAuthenticationHandler
- Microsoft.AspNetCore.Http.Features.Authentication.IHttpAuthenticationFeature.Handler
- Microsoft.AspNetCore.Http.Features.Authentication.SignInContext
- Microsoft.AspNetCore.Http.Features.Authentication.SignOutContext
- Microsoft.AspNetCore.Http.HttpContext.Authentication

身份验证：Newtonsoft.json 类型已替换

在 ASP.NET Core 3.0 中，身份验证 API 中使用的 `Newtonsoft.Json` 类型已替换为 `System.Text.Json` 类型。除以下情况外，身份验证包的基本使用不受影响：

- 派生自 OAuth 提供程序的类，例如来自 aspnet-contrib 的类。
- 高级声明操作实现。

有关详细信息，请参阅 dotnet/aspnetcore#7105。有关讨论，请参阅 dotnet/aspnetcore#7289。

引入的版本

3.0

建议操作

对于派生的 OAuth 实现，最常见的更改是将 `JObject.Parse` 替换为 `CreateTicketAsync` 重写中的 `JsonDocument.Parse`，如本文所示。`JsonDocument` 可实现 `IDisposable`。

以下列表概述了已知更改：

- ClaimAction.Run(JObject, ClaimsIdentity, String) 变为 `ClaimAction.Run(JsonElement userData, ClaimsIdentity identity, string issuer)`。`ClaimAction` 的所有派生的实现都受到类似的影响。
- ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, Func<JObject,String>) 变为
  ```
  MapCustomJson(this ClaimActionCollection collection, string claimType, Func<JsonElement, string>
  resolver)
  ```
- ClaimActionCollectionMapExtensions.MapCustomJson(ClaimActionCollection, String, String, Func<JObject,String>) 变为
  ```
  MapCustomJson(this ClaimActionCollection collection, string claimType, string valueType,
  Func<JsonElement, string> resolver)
  ```
- OAuthCreatingTicketContext 已移除一个旧构造函数，另一个将 `JObject` 替换为 `JsonElement`。已更新 `User` 属性和 `RunClaimActions` 方法以使其匹配。
- Success(JObject) 现在接受类型为 `JsonDocument` 而非 `JObject` 的参数。已更新 `Response` 属性以使其匹配。`OAuthTokenResponse` 现在可处置，并将由 `OAuthHandler` 处置。替代 `ExchangeCodeAsync` 的派生的 OAuth 实现无需处置 `JsonDocument` 或 `OAuthTokenResponse`。
- UserInformationReceivedContext.User 从 `JObject` 更改为 `JsonDocument`。
- TwitterCreatingTicketContext.User 从 `JObject` 更改为 `JsonElement`。
- TwitterHandler.CreateTicketAsync(ClaimsIdentity,AuthenticationProperties,AccessToken,JObject) 的最后一个参数从 `JObject` 更改为 `JsonElement`。替换方法为 TwitterHandler.CreateTicketAsync(ClaimsIdentity, AuthenticationProperties, AccessToken, JsonElement)。

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Authentication.Facebook
- Microsoft.AspNetCore.Authentication.Google
- Microsoft.AspNetCore.Authentication.MicrosoftAccount
- Microsoft.AspNetCore.Authentication.OAuth
- Microsoft.AspNetCore.Authentication.OpenIdConnect
- Microsoft.AspNetCore.Authentication.Twitter

身份验证：已更改 OAuthHandler ExchangeCodeAsync 签名

在 ASP.NET Core 3.0 中，`OAuthHandler.ExchangeCodeAsync` 的签名已从以下位置更改：

```
protected virtual System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthTokenResponse>
ExchangeCodeAsync(string code, string redirectUri) { throw null; }
```

到：

```
protected virtual System.Threading.Tasks.Task<Microsoft.AspNetCore.Authentication.OAuth.OAuthTokenResponse>
ExchangeCodeAsync(Microsoft.AspNetCore.Authentication.OAuth.OAuthCodeExchangeContext context) { throw null;
}
```

引入的版本

3.0

旧行为

`code` 和 `redirectUri` 字符串作为单独的参数传递。

新行为

`Code` 和 `RedirectUri` 是 `OAuthCodeExchangeContext` 上的属性，可通过 `OAuthCodeExchangeContext` 构造函数进行设置。新的 `OAuthCodeExchangeContext` 类型是传递到 `OAuthHandler.ExchangeCodeAsync` 的唯一参数。

更改原因

此更改允许以非中断方式提供其他参数。无需创建新的 `ExchangeCodeAsync` 重载。

建议操作

使用适当的 `code` 和 `redirectUri` 值构造 `OAuthCodeExchangeContext`。必须提供 AuthenticationProperties 实例。此单个 `OAuthCodeExchangeContext` 实例可传递到 `OAuthHandler.ExchangeCodeAsync` 而不是多个参数。

类别

ASP.NET Core

受影响的 API

OAuthHandler<TOptions>.ExchangeCodeAsync(String, String)

授权：AddAuthorization 重载已移动到不同的程序集

用于驻留在 `Microsoft.AspNetCore.Authorization` 中的核心 `AddAuthorization` 方法已重命名为 `AddAuthorizationCore`。旧的 `AddAuthorization` 方法仍然存在，但却在 `Microsoft.AspNetCore.Authorization.Policy` 程序集中。使用这两种方法的应用应该不会受到任何影响。请注意，`Microsoft.AspNetCore.Authorization.Policy` 现随附于共享框架而不是独立的包中，如共享框架：从 Microsoft.AspNetCore.App 中删除了程序集中所述。

引入的版本

3.0

旧行为

`Microsoft.AspNetCore.Authorization` 中已存在 `AddAuthorization` 方法。

新行为

`Microsoft.AspNetCore.Authorization.Policy` 中存在 `AddAuthorization` 方法。`AddAuthorizationCore` 是旧方法的新名称。

更改原因

`AddAuthorization` 是一个更好的方法名称，用于添加授权所需的所有常用服务。

建议操作

添加对 `Microsoft.AspNetCore.Authorization.Policy` 的引用或改用 `AddAuthorizationCore`。

类别

ASP.NET Core

受影响的 API

Microsoft.Extensions.DependencyInjection.AuthorizationServiceCollectionExtensions.AddAuthorization(IServiceCollection, Action<AuthorizationOptions>)

授权：从 AuthorizationFilterContext.Filters 中删除 IAllowAnonymous

从 ASP.NET Core 3.0 起，MVC 不会为在控制器和操作方法上发现的 [AllowAnonymous] 属性添加 AllowAnonymousFilters。此更改针对 AuthorizeAttribute 派生在本地进行处理，但对于 IAsyncAuthorizationFilter 和 IAuthorizationFilter 实现来说是一个重大变更。包装在 [TypeFilter] 属性中的此类实现是在需要配置和依赖项注入时实现强类型的基于属性的授权的常见的支持方式。

引入的版本

3.0

旧行为

IAllowAnonymous 出现在 AuthorizationFilterContext.Filters 集合中。对接口的状态进行测试是对单个控制器方法上的筛选器进行重写或禁用的一种有效方法。

新行为

`IAllowAnonymous` 不再出现在 `AuthorizationFilterContext.Filters` 集合中。依赖于旧行为的 `IAsyncAuthorizationFilter` 实现通常导致间歇性 HTTP 401 未授权或 HTTP 403 禁止响应。

更改原因

ASP.NET Core 3.0 中引入了新的终结点路由策略。

建议操作

在终结点元数据中搜索 `IAllowAnonymous`。例如：

```
var endpoint = context.HttpContext.GetEndpoint();
if (endpoint?.Metadata?.GetMetadata<IAllowAnonymous>() != null)
{
}
```

此 HasAllowAnonymous 方法中演示了此方法。

类别

ASP.NET Core

受影响的 API

None

授权：IAuthorizationPolicyProvider 实现需要新方法

在 ASP.NET Core 3.0 中，已将一个新 `GetFallbackPolicyAsync` 方法添加到 `IAuthorizationPolicyProvider`。当未指定策略时，授权中间件会使用此回退策略。

有关详细信息，请参阅 dotnet/aspnetcore#9759。

引入的版本

3.0

旧行为

`IAuthorizationPolicyProvider` 的实现不需要 `GetFallbackPolicyAsync` 方法。

新行为

`IAuthorizationPolicyProvider` 的实现需要 `GetFallbackPolicyAsync` 方法。

更改原因

如果未指定策略，则新 `AuthorizationMiddleware` 需要使用新方法。

建议操作

将 `GetFallbackPolicyAsync` 方法添加到 `IAuthorizationPolicyProvider` 的实现。

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Authorization.IAuthorizationPolicyProvider

---

缓存：已删除 CompactOnMemoryPressure 属性

ASP.NET Core 3.0 版本已删除过时的 MemoryCacheOptions API。

更改描述

此更改是对 aspnet/Caching#221 的后续补充。有关讨论，请参阅 dotnet/extensions#1062。

引入的版本

3.0

旧行为

`MemoryCacheOptions.CompactOnMemoryPressure` 属性可用。

新行为

`MemoryCacheOptions.CompactOnMemoryPressure` 属性已被删除。

更改原因

自动压缩缓存会引起问题。若要避免意外行为，只应在需要时压缩缓存。

建议操作

若要压缩缓存，请向下转换到 `MemoryCache` 并在需要时调用 `Compact`。

类别

ASP.NET Core

受影响的 API

MemoryCacheOptions.CompactOnMemoryPressure

缓存：Microsoft.Extensions.Caching.SqlServer 使用新 SqlClient 包

`Microsoft.Extensions.Caching.SqlServer` 包将使用新的 `Microsoft.Data.SqlClient` 包，而不是 `System.Data.SqlClient` 包。此更改可能会导致轻微的行为中断性变更。有关详细信息，请参阅引入新的 Microsoft.Data.SqlClient。

引入的版本

3.0

旧行为

`Microsoft.Extensions.Caching.SqlServer` 包已使用过 `System.Data.SqlClient` 包。

新行为

`Microsoft.Extensions.Caching.SqlServer` 现在使用 `Microsoft.Data.SqlClient` 包。

更改原因

`Microsoft.Data.SqlClient` 是从 `System.Data.SqlClient` 生成的新包。从现在开始，所有新功能的工作都在该包中进行。

建议操作

客户无需担心此中断性变更，除非他们使用 `Microsoft.Extensions.Caching.SqlServer` 包返回的类型，并将它们强制转换为 `System.Data.SqlClient` 类型。例如，如果有人将 `DbConnection` 强制转换为旧的 SqlConnection 类型，则需要将转换更改为新的 `Microsoft.Data.SqlClient.SqlConnection` 类型。

类别

ASP.NET Core

受影响的 API

None

缓存：ResponseCaching“Pubternal”类型已更改为内部

在 ASP.NET Core 3.0 中，`ResponseCaching` 中的“pubternal”类型已更改为 `internal`。

此外，`IResponseCachingPolicyProvider` 和 `IResponseCachingKeyProvider` 的默认实现不再作为 `AddResponseCaching` 方法的一部分添加到服务。

更改描述

在 ASP.NET Core 中，“pubternal”类型声明为 `public`，但驻留在后缀为 `.Internal` 的命名空间中。尽管这些类型为 public，但它们没有支持策略，可能会发生中断性变更。遗憾的是，经常会意外使用这些类型，导致对这些项目做出中断性变更，并使维护框架的能力受到限制。

引入的版本

3.0

旧行为

这些类型公开可见，但不受支持。

新行为

这些类型现在为 `internal`。

更改原因

`internal` 范围更好地反映了不受支持的策略。

建议操作

复制应用或库使用的类型。

类别

ASP.NET Core

受影响的 API

- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedResponse`
- `Microsoft.AspNetCore.ResponseCaching.Internal.CachedVaryByRules`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCacheEntry`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.IResponseCachingPolicyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.MemoryResponseCache`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingContext`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingKeyProvider`
- `Microsoft.AspNetCore.ResponseCaching.Internal.ResponseCachingPolicyProvider`
- Microsoft.AspNetCore.ResponseCaching.ResponseCachingMiddleware.ResponseCachingMiddleware(RequestDelegate, IOptions<ResponseCachingOptions>, ILoggerFactory, IResponseCachingPolicyProvider, IResponseCache, IResponseCachingKeyProvider)

数据保护：DataProtection.Blobs 使用新的 Azure 存储 API

`Azure.Extensions.AspNetCore.DataProtection.Blobs` 取决于 Azure 存储库。这些库已重命名其程序集、包和命名空间。从 ASP.NET Core 3.0 开始，`Azure.Extensions.AspNetCore.DataProtection.Blobs` 使用新的带有 `Azure.Storage.` 前缀的 API 和包。

有关 Azure 存储 API 的问题，请使用 https://github.com/Azure/azure-storage-net。有关此问题的讨论，请参阅 dotnet/aspnetcore#19570。

引入的版本

3.0

旧行为

该包引用了 `WindowsAzure.Storage` NuGet 包。该包引用 `Microsoft.Azure.Storage.Blob` NuGet 包。

新行为

该包引用 `Azure.Storage.Blob` NuGet 包。

更改原因

此更改允许 `Azure.Extensions.AspNetCore.DataProtection.Blobs` 迁移到建议的 Azure 存储包。

建议操作

如果仍需要将较旧的 Azure 存储 API 与 ASP.NET Core 3.0 一起使用，请将直接依赖项添加到包 WindowsAzure.Storage 或 Microsoft.Azure.Storage。此包可以与新的 `Azure.Storage` API 一起安装。

在许多情况下，升级仅涉及更改 `using` 语句以使用新的命名空间：

```
- using Microsoft.WindowsAzure.Storage;
- using Microsoft.WindowsAzure.Storage.Blob;
- using Microsoft.Azure.Storage;
- using Microsoft.Azure.Storage.Blob;
+ using Azure.Storage;
+ using Azure.Storage.Blobs;
```

类别

ASP.NET Core

受影响的 API

无

托管：从 Windows 托管捆绑包中删除了 AspNetCoreModule V1

从 ASP.NET Core 3.0 开始，Windows 托管捆绑不包含 AspNetCoreModule (ANCM) V1。

ANCM V2 向后兼容 ANCM OutOfProcess，建议与 ASP.NET Core 3.0 应用一起使用。

有关讨论，请参阅 dotnet/aspnetcore#7095。

引入的版本

3.0

旧行为

ANCM V1 包含在 Windows 托管捆绑包中。

新行为

ANCM V1 不包含在 Windows 托管捆绑包中。

更改原因

ANCM V2 向后兼容 ANCM OutOfProcess，建议与 ASP.NET Core 3.0 应用一起使用。

建议操作

将 ANCM V2 与 ASP.NET Core 3.0 应用一起使用。

如果需要 ANCM V1，则可以使用 ASP.NET Core 2.1 或 2.2 Windows 托管捆绑包进行安装。

此更改将中断以下 ASP.NET Core 3.0 应用：

- 已明确选择将 ANCM V1 与 `<AspNetCoreModuleName>AspNetCoreModule</AspNetCoreModuleName>` 结合使用。
- 具有 `<add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModule" resourceType="Unspecified" />` 的自定义 web.config 文件。

类别

ASP.NET Core

受影响的 API

None

托管：通用主机限制 Startup 构造函数注入

通用主机支持的 `Startup` 类构造函数注入的唯一类型为 `IHostEnvironment`、`IWebHostEnvironment` 和 `IConfiguration`。使用 `WebHost` 的应用不受影响。

更改描述

在低于 ASP.NET Core 3.0 的版本中，构造函数注入可用于 `Startup` 类的构造函数中的任意类型。在 ASP.NET Core 3.0 中，Web 堆栈将重新平台化到通用主机库上。可以在模板的 Program .cs 文件中查看更改：

ASP.NET Core 2.x：

https://github.com/dotnet/aspnetcore/blob/5cb615fcbe8559e49042e93394008077e30454c0/src/Templating/src/Microsoft.DotNet.Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L20-L22

ASP.NET Core 3.0：

https://github.com/dotnet/aspnetcore/blob/b1ca2c1155da3920f0df5108b9fedbe82efaa11c/src/ProjectTemplates/Web.ProjectTemplates/content/EmptyWeb-CSharp/Program.cs#L19-L24

`Host` 使用一个依赖关系注入 (DI) 容器生成应用。`WebHost` 使用两个容器：一个用于主机，另一个用于应用。因此，`Startup` 构造函数不再支持自定义服务注入。只能注入 `IHostEnvironment`、`IWebHostEnvironment` 和 `IConfiguration`。此更改可防止出现 DI 问题，例如重复创建单一实例服务。

引入的版本

3.0

更改原因

此更改是将 Web 堆栈重新平台化到通用主机库的结果。

建议操作

将服务注入 `Startup.Configure` 方法签名。例如：

```
public void Configure(IApplicationBuilder app, IOptions<MyOptions> options)
```

类别

ASP.NET Core

受影响的 API

None

托管：已为 IIS 进程外应用启用 HTTPS 重定向

用于通过 IIS 进程外应用进行托管的 ASP.NET Core 模块 (ANCM) 的 13.0.19218.0 版本可为 ASP.NET Core 3.0 和 2.2 应用启用现有 HTTPS 重定向功能。

有关讨论，请参阅 dotnet/AspNetCore#15243。

引入的版本

3.0

旧行为

ASP.NET Core 2.1 项目模板首先引入了对 HTTPS 中间件方法的支持，例如 UseHttpsRedirection 和 UseHsts。启用 HTTPS 重定向需要添加配置，因为开发中的应用不使用默认端口 443。仅当请求已使用 HTTPS 时，HTTP 严格传输安全 (HSTS) 才处于活动状态。默认跳过 localhost。

新行为

在 ASP.NET Core 3.0 中，IIS HTTPS 方案已增强。利用此增强功能，应用可以发现服务器的 HTTPS 端口并默认使 `UseHttpsRedirection` 工作。进程内组件通过 `IServerAddresses` 功能完成端口发现，该功能仅影响 ASP.NET Core 3.0 应用，因为进程内库通过框架进行版本控制。进程外组件已更改为自动添加 `ASPNETCORE_HTTPS_PORT` 环境变量。由于进程外组件是全局共享组件，因此此更改会同时影响 ASP.NET Core 2.2 和 3.0 应用。ASP.NET Core 2.1 应用不会受到影响，因为它们默认使用 ANCM 的先前版本。

前面的行为已在 ASP.NET Core 3.0.1 和 3.1.0 预览版 3 中进行修改，以反转 ASP.NET Core 2.x 中的行为更改。这些更改仅影响 IIS 进程外应用。

如上所述，安装 ASP.NET Core 3.0.0 还具有在 ASP.NET Core 2.x 应用中激活 `UseHttpsRedirection` 中间件的副作用。已对 ASP.NET Core 3.0.1 和 3.1.0 预览版 3 中的 ANCM 进行更改，以确保安装它们时不会再对 ASP.NET Core 2.x 应用产生此影响。ANCM 在 ASP.NET Core 3.0.0 中填充的 `ASPNETCORE_HTTPS_PORT` 环境变量在 ASP.NET Core 3.0.1 和 3.1.0 预览版 3 中已更改为 `ASPNETCORE_ANCM_HTTPS_PORT`。`UseHttpsRedirection` 在这些版本中也已更新，可同时理解新旧变量。不会更新 ASP.NET Core 2.x。因此，它会还原为默认禁用的先前行为。

更改原因

已改进 ASP.NET Core 3.0 功能。

建议操作

如果希望所有客户端都使用 HTTPS，则无需执行任何操作。要允许部分客户端使用 HTTP，请执行以下步骤之一：

- 从项目的 `Startup.Configure` 方法中删除对 `UseHttpsRedirection` 和 `UseHsts` 的调用，然后重新部署应用。
- 在 web.config 文件中，将 `ASPNETCORE_HTTPS_PORT` 环境变量设置为空字符串 。无需重新部署应用即可直接在服务器上进行此更改。例如：

```
<aspNetCore processPath="dotnet" arguments=".\WebApplication3.dll" stdoutLogEnabled="false"
stdoutLogFile="\\?\%home%\LogFiles\stdout" >
    <environmentVariables>
    <environmentVariable name="ASPNETCORE_HTTPS_PORT" value="" />
    </environmentVariables>
</aspNetCore>
```

`UseHttpsRedirection` 仍可：

- 在 ASP.NET Core 2.x 中手动激活，方法是将 `ASPNETCORE_HTTPS_PORT` 环境变量设置为相应的端口号(在大多数生产方案中为 443)。
- 在 ASP.NET Core 3.x 中停用，方法是使用空字符串值定义 `ASPNETCORE_ANCM_HTTPS_PORT`。此值的设置方式与前面的 `ASPNETCORE_HTTPS_PORT` 示例相同。

运行 ASP.NET Core 3.0.0 应用的计算机应先安装 ASP.NET Core 3.0.1 运行时，然后再安装 ASP.NET Core 3.1.0 预览版 3 ANCM。这样做可以确保继续按预期对 ASP.NET Core 3.0 应用运行 `UseHttpsRedirection`。

在 Azure 应用服务中，由于 ANCM 具有全局性，其部署时间与运行时不同。部署 ASP.NET Core 3.0.1 和 3.1.0 后，才会将 ANCM 以及相应的更改部署到 Azure。

类别

ASP.NET Core

受影响的 API

HttpsPolicyBuilderExtensions.UseHttpsRedirection(IApplicationBuilder)

托管：IHostingEnvironment 和 IApplicationLifetime 类型标记为过时并被替换

引入了新类型以替换现有的 `IHostingEnvironment` 和 `IApplicationLifetime` 类型。

引入的版本

3.0

旧行为

`Microsoft.Extensions.Hosting` 和 `Microsoft.AspNetCore.Hosting` 有两个不同的 `IHostingEnvironment` 和 `IApplicationLifetime` 类型。

新行为

旧类型已标记为过时，并已替换为新类型。

更改原因

在 ASP.NET Core 2.1 中引入 `Microsoft.Extensions.Hosting` 时，从 `Microsoft.AspNetCore.Hosting` 复制了某些类型（如 `IHostingEnvironment` 和 `IApplicationLifetime`）。某些 ASP.NET Core 3.0 更改会导致应用包括 `Microsoft.Extensions.Hosting` 和 `Microsoft.AspNetCore.Hosting` 命名空间。如果同时引用了两个命名空间，则使用这些重复类型会导致“引用不明确”编译器错误。

建议操作

将旧类型的所有使用替换为新引入的类型，如下所示：

过时类型（警告）：

- Microsoft.Extensions.Hosting.IHostingEnvironment
- Microsoft.AspNetCore.Hosting.IHostingEnvironment
- Microsoft.Extensions.Hosting.IApplicationLifetime
- Microsoft.AspNetCore.Hosting.IApplicationLifetime
- Microsoft.Extensions.Hosting.EnvironmentName
- Microsoft.AspNetCore.Hosting.EnvironmentName

新类型：

- Microsoft.Extensions.Hosting.IHostEnvironment
- `Microsoft.AspNetCore.Hosting.IWebHostEnvironment : IHostEnvironment`
- Microsoft.Extensions.Hosting.IHostApplicationLifetime
- Microsoft.Extensions.Hosting.Environments

新的 `IHostEnvironment` `IsDevelopment` 和 `IsProduction` 扩展方法位于 `Microsoft.Extensions.Hosting` 命名空间中。可能需要将该命名空间添加到项目中。

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Hosting.EnvironmentName
- Microsoft.AspNetCore.Hosting.IApplicationLifetime
- Microsoft.AspNetCore.Hosting.IHostingEnvironment
- Microsoft.Extensions.Hosting.EnvironmentName
- Microsoft.Extensions.Hosting.IApplicationLifetime
- Microsoft.Extensions.Hosting.IHostingEnvironment

托管：ObjectPoolProvider 已从 WebHostBuilder 依赖项中删除

作为使 ASP.NET Core 更具成本效益计划的一部分，`ObjectPoolProvider` 已从主要依赖项集中删除。依赖于 `ObjectPoolProvider` 的特定组件现在会自行添加。

有关讨论，请参阅 dotnet/aspnetcore#5944。

引入的版本

3.0

旧行为

`WebHostBuilder` 默认在 DI 容器中提供 `ObjectPoolProvider`。

新行为

`WebHostBuilder` 默认不再在 DI 容器中提供 `ObjectPoolProvider`。

更改原因

进行此更改是为了使 ASP.NET Core 更具成本效益。

建议操作

如果组件需要 `ObjectPoolProvider`，则需要通过 `IServiceCollection` 将其添加到依赖项。

类别

ASP.NET Core

受影响的 API

None

HTTP：已删除 DefaultHttpContext 扩展性

作为 ASP.NET Core 3.0 性能改进的一部分，已删除 `DefaultHttpContext` 的扩展性。此类现在为 `sealed`。有关详细信息，请参阅 dotnet/aspnetcore#6504。

如果单元测试使用 `Mock<DefaultHttpContext>`，请改用 `Mock<HttpContext>` 或 `new DefaultHttpContext()`。

有关讨论，请参阅 dotnet/aspnetcore#6534。

引入的版本

3.0

旧行为

类可从 `DefaultHttpContext` 派生。

新行为

类不可从 `DefaultHttpContext` 派生。

更改原因

最初提供扩展性是为了允许 `HttpContext` 合并，但它引入了不必要的复杂性并妨碍了其他优化。

建议操作

如果在单元测试中使用 `Mock<DefaultHttpContext>`，请改为开始使用 `Mock<HttpContext>`。

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Http.DefaultHttpContext

HTTP：HeaderNames 常量已更改为静态只读

从 ASP.NET Core 3.0 预览版 5 开始，Microsoft.Net.Http.Headers.HeaderNames 中的字段已从 `const` 更改为 `static readonly`。

有关讨论，请参阅 dotnet/aspnetcore#9514。

引入的版本

3.0

旧行为

这些字段以前是 `const`。

新行为

这些字段现在是 `static readonly`。

更改原因

更改：

- 防止将值嵌入到程序集边界内，允许根据需要更正值。
- 实现更快的引用同等性检查。

建议操作

针对 3.0 重新编译。通过以下方式使用这些字段的源代码将无法再执行此项操作：

- 作为特性参数
- 作为 `switch` 语句中的 `case`
- 定义其他 `const` 时

若要解决中断性变更，请切换到使用自定义标头名称常量或字符串文本。

类别

ASP.NET Core

受影响的 API

Microsoft.Net.Http.Headers.HeaderNames

HTTP：响应正文基础结构更改

支持 HTTP 响应正文的基础结构已发生更改。如果直接使用 `HttpResponse`，则不需要进行任何代码更改。如果要包装或替换 `HttpResponse.Body` 或访问 `HttpContext.Features`，请进行进一步了解。

引入的版本

3.0

旧行为

有三个 API 与 HTTP 响应正文关联：

- `IHttpResponseFeature.Body`
- `IHttpSendFileFeature.SendFileAsync`
- `IHttpBufferingFeature.DisableResponseBuffering`

新行为

如果替换 `HttpResponse.Body`，则它通过使用 `StreamResponseBodyFeature` 为所有预期的 API 提供默认实现，将整个 `IHttpResponseBodyFeature` 替换为给定流周围的包装器。重新设置回原始流会还原此更改。

更改原因

动机是将响应正文 API 合并为单一新功能接口。

建议操作

使用之前在其中使用 `IHttpResponseFeature.Body`、`IHttpSendFileFeature` 或 `IHttpBufferingFeature` 的 `IHttpResponseBodyFeature`。

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Http.Features.IHttpBufferingFeature
- Microsoft.AspNetCore.Http.Features.IHttpResponseFeature.Body
- Microsoft.AspNetCore.Http.Features.IHttpSendFileFeature

HTTP：某些 cookie SameSite 默认值更改为“None”

`SameSite` 是 cookie 的一个选项，可以帮助减轻某些跨站点请求伪造 (CSRF) 攻击。最初引入此选项时，各种 ASP.NET Core API 中使用了不一致的默认值。不一致会导致结果混乱。从 ASP.NET Core 3.0 开始，这些默认值更一致了。必须为每个组件选择启用此功能。

引入的版本

3.0

旧行为

类似的 ASP.NET Core API 使用不同的默认值 SameSiteMode。在 `HttpResponse.Cookies.Append(String, String)` 和 `HttpResponse.Cookies.Append(String, String, CookieOptions)` 中可以看到不一致的示例，它们分别默认为 `SameSiteMode.None` 和 `SameSiteMode.Lax`。

新行为

所有受影响的 API 都默认为 `SameSiteMode.None`。

更改原因

更改了默认值，使 `SameSite` 成为可选功能。

建议操作

发出 cookie 的每个组件都需要决定 `SameSite` 是否适用于其方案。检查受影响 API 的使用情况，并根据需要重新配置 `SameSite`。

类别

ASP.NET Core

受影响的 API

- IResponseCookies.Append(String, String, CookieOptions)
- CookiePolicyOptions.MinimumSameSitePolicy

HTTP：所有服务器均禁用同步 IO

从 ASP.NET Core 3.0 开始，默认将禁用同步服务器操作。

更改描述

`AllowSynchronousIO` 是每台服务器中包含的一个选项，用于启用或禁用同步 IO API，如 `HttpRequest.Body.Read`、`HttpResponse.Body.Write` 和 `Stream.Flush`。这些 API 长期以来一直是线程不足和应用挂起的根源。从 ASP.NET Core 3.0 预览版 3 开始，默认将禁用这些同步操作。

受影响的服务器：

- Kestrel
- HttpSys
- IIS（进程内）
- TestServer

错误应如下所示：

- `Synchronous operations are disallowed. Call ReadAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call WriteAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call FlushAsync or set AllowSynchronousIO to true instead.`

每台服务器都有一个 `AllowSynchronousIO` 选项，该选项控制此行为，且它们的默认值当前为 `false`。

作为一种临时缓解措施，还可以根据每个请求重写该行为。例如：

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();
if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

如果调用 `Dispose` 中同步 API 的 `TextWriter` 或另一个流出现问题，请改为调用新的 `DisposeAsync` API。

有关讨论，请参阅 dotnet/aspnetcore#7644。

引入的版本

3.0

旧行为

默认情况下允许 `HttpRequest.Body.Read`、`HttpResponse.Body.Write` 和 `Stream.Flush`。

新行为

默认情况下，不允许使用这些同步 API：

错误应如下所示：

- `Synchronous operations are disallowed. Call ReadAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call WriteAsync or set AllowSynchronousIO to true instead.`
- `Synchronous operations are disallowed. Call FlushAsync or set AllowSynchronousIO to true instead.`

更改原因

这些同步 API 长期以来一直是线程不足和应用挂起的根源。从 ASP.NET Core 3.0 预览版 3 开始，默认将禁用同步操作。

建议操作

使用这些方法的异步版本。作为一种临时缓解措施，还可以根据每个请求重写该行为。

```
var syncIOFeature = HttpContext.Features.Get<IHttpBodyControlFeature>();
if (syncIOFeature != null)
{
    syncIOFeature.AllowSynchronousIO = true;
}
```

类别

ASP.NET Core

受影响的 API

- Stream.Flush
- Stream.Read
- Stream.Write

标识：**已删除 AddDefaultUI 方法重载**

从 ASP.NET Core 3.0 开始，IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder,UIFramework) 方法重载不再存在。

引入的版本

3.0

更改原因

此更改是采用静态 Web 资产功能的结果。

建议操作

调用 IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder) 而不是使用两个参数的重载。如果使用的是 Bootstrap 3，还应将以下行添加到项目文件中的 `<PropertyGroup>` 元素：

```
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

类别

ASP.NET Core

受影响的 API

IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder,UIFramework)

标识：**已更改 UI 的默认 Bootstrap 版本**

从 ASP.NET Core 3.0 开始，标识 UI 默认为使用 Bootstrap 版本 4。

引入的版本

3.0

旧行为

`services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` 方法调用以前与 `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap3);` 相同

新行为

`services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();` 方法调用现在与 `services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap4);` 相同

更改原因

在 ASP.NET Core 3.0 时间范围内发布了 Bootstrap 4。

建议操作

如果使用默认标识用户界面并将其添加到 `Startup.ConfigureServices` 中，则会受到此更改的影响，如以下示例中所示：

```
services.AddDefaultIdentity<IdentityUser>().AddDefaultUI();
```

请执行以下一项操作：

- 迁移应用，以使用其迁移指南来使用 Bootstrap 4。
- 更新 `Startup.ConfigureServices` 以强制使用 Bootstrap 3。例如：

```
services.AddDefaultIdentity<IdentityUser>().AddDefaultUI(UIFramework.Bootstrap3);
```

类别

ASP.NET Core

受影响的 API

None

标识：对于未经身份验证的标识，SignInAsync 会引发异常

默认情况下，`SignInAsync` 会为其中 `IsAuthenticated` 为 `false` 的主体/标识引发异常。

引入的版本

3.0

旧行为

`SignInAsync` 接受任何主体/标识，包括其中 `IsAuthenticated` 为 `false` 的标识。

新行为

默认情况下，`SignInAsync` 会为其中 `IsAuthenticated` 为 `false` 的主体/标识引发异常。有一个新的标志可禁止显示此行为，但默认行为已更改。

更改原因

旧行为是有问题的，因为在默认情况下，`[Authorize]` / `RequireAuthenticatedUser()` 拒绝了这些主体。

建议操作

在 ASP.NET Core 3.0 预览版 6 中，`AuthenticationOptions` 上有 `RequireAuthenticatedSignIn` 标记，默认为 `true`。将此标志设置为 `false` 以还原旧行为。

类别

ASP.NET Core

受影响的 API

None

标识：SignInManager 构造函数接受新参数

从 ASP.NET Core 3.0 开始，新的 `IUserConfirmation<TUser>` 参数添加到了 `SignInManager` 构造函数中。有关详细信息，请参阅 dotnet/aspnetcore#8356。

引入的版本

3.0

更改原因

更改的动机是为了添加对标识中的新电子邮件/确认流的支持。

建议操作

如果手动构造 `SignInManager`，请提供 `IUserConfirmation` 的实现，或从依赖项注入获取一个实现来提供。

类别

ASP.NET Core

受影响的 API

SignInManager<TUser>

标识：UI 使用静态 Web 资产功能

ASP.NET Core 3.0 引入了静态 Web 资产功能，标识 UI 已采用此功能。

更改描述

由于标识 UI 采用静态 Web 资产功能，因此：

- 可通过使用项目文件中的 `IdentityUIFrameworkVersion` 属性来完成框架选择。
- Bootstrap 4 是标识 UI 的默认 UI 框架。Bootstrap 3 的生命周期已经结束，应考虑迁移到受支持的版本。

引入的版本

3.0

旧行为

标识 UI 的默认 UI 框架为 Bootstrap 3 。可使用 `Startup.ConfigureServices` 中 `AddDefaultUI` 方法调用的参数配置 UI 框架。

新行为

标识 UI 的默认 UI 框架为 Bootstrap 4 。UI 框架必须在项目文件中进行配置，而不是在 `AddDefaultUI` 方法调用中配置。

更改原因

采用静态 Web 资产功能要求 UI 框架配置迁移到 MSBuild。要在哪个框架上进行嵌入的决策是生成时决策，而非

运行时决策。

建议操作

查看站点 UI，以确保新的 Bootstrap 4 组件兼容。如有必要，请使用 `IdentityUIFrameworkVersion` MSBuild 属性还原为 Bootstrap 3。将属性添加到项目文件中的 `<PropertyGroup>` 元素：

```
<IdentityUIFrameworkVersion>Bootstrap3</IdentityUIFrameworkVersion>
```

类别

ASP.NET Core

受影响的 API

IdentityBuilderUIExtensions.AddDefaultUI(IdentityBuilder, UIFramework)

Kestrel：已删除连接适配器

作为将“pubternal”API 移动到 `public` 的移动的一部分，已从 Kestrel 中删除 `IConnectionAdapter` 的概念。正在将连接适配器替换为连接中间件（这与 ASP.NET Core 管道中的 HTTP 中间件类似，但适用于较低级别的连接）。HTTPS 和连接日志记录已从连接适配器转移到连接中间件。这些扩展方法应能继续无缝运行，但实现详细信息发生了改变。

有关详细信息，请参阅 dotnet/aspnetcore#11412。有关讨论，请参阅 dotnet/aspnetcore#11475。

引入的版本

3.0

旧行为

Kestrel 扩展性组件是使用 `IConnectionAdapter` 创建的。

新行为

Kestrel 扩展性组件被创建为中间件。

更改原因

此更改旨在提供更灵活的扩展性体系结构。

建议操作

转换 `IConnectionAdapter` 的任何实现以使用新的中间件模式，如此处所示。

类别

ASP.NET Core

受影响的 API

`Microsoft.AspNetCore.Server.Kestrel.Core.Adapter.Internal.IConnectionAdapter`

Kestrel：删除了空 HTTPS 程序集

已删除程序集 Microsoft.AspNetCore.Server.Kestrel.Https。

引入的版本

3.0

更改原因

在 ASP.NET Core 2.1 中，`Microsoft.AspNetCore.Server.Kestrel.Https` 的内容已移动到 Microsoft.AspNetCore.Server.Kestrel.Core 中。此更改是使用 `[TypeForwardedTo]` 属性以非中断方式进行的。

建议操作

- 引用 `Microsoft.AspNetCore.Server.Kestrel.Https` 2.0 的库应将所有 ASP.NET Core 依赖项更新为 2.1 或更高版本。否则，它们可能会在加载到 ASP.NET Core 3.0 应用时中断。
- 面向 ASP.NET Core 2.1 和更高版本的应用和库应删除对 `Microsoft.AspNetCore.Server.Kestrel.Https` NuGet 包的任何直接引用。

类别

ASP.NET Core

受影响的 API

None

Kestrel：请求尾部标头已移动到新集合

在以前的版本中，当读取请求正文直到结尾时，Kestrel 会将 HTTP/1.1 分块尾部标头添加到请求标头集合中。此行为将引发对标头和尾部之间存在不明确性的担忧。因此，做出了将尾部移动到新集合的决定。

HTTP/2 请求尾部在 ASP.NET Core 2.2 中不可用，但现在可用于 ASP.NET Core 3.0 的此新集合。

添加了新的请求扩展方法以访问这些尾部。

读取整个请求正文后，HTTP/1.1 尾部即可用。

从客户端收到 HTTP/2 尾部后，这些尾部即可用。客户端将不会发生尾部，直到整个请求正文至少已由服务器缓冲。可能需要读取请求正文，以释放缓冲区空间。如果读取请求正文直到结尾，则尾部始终可用。尾部标记正文的结尾。

引入的版本

3.0

旧行为

请求尾部标头将添加到 `HttpRequest.Headers` 集合中。

新行为

请求尾部标头在 `HttpRequest.Headers` 集合中不存在。在 `HttpRequest` 上使用以下扩展方法来访问它们：

- `GetDeclaredTrailers()` - 获取列出了正文后应具有的尾部的请求"尾部"标头。
- `SupportsTrailers()` - 指示请求是否支持接收尾部标头。
- `CheckTrailersAvailable()` - 确定请求是否支持尾部以及是否可读取。
- `GetTrailer(string trailerName)` - 从响应获取请求的尾随标头。

更改原因

在 gRPC 等方案中，尾部是关键功能。将尾部合并到请求标头会使用户感到困惑。

建议操作

在 `HttpRequest` 上使用尾部相关扩展方法访问尾部。

类别

ASP.NET Core

受影响的 API

HttpRequest.Headers

Kestrel：删除并公开传输抽象

作为远离"pubternal"API 的一部分，Kestrel 传输层 API 被公开为 `Microsoft.AspNetCore.Connections.Abstractions` 库中的公共接口。

引入的版本

3.0

旧行为

- `Microsoft.AspNetCore.Server.Kestrel.Transport.Abstractions` 库中提供传输相关的抽象。
- `ListenOptions.NoDelay` 属性可用。

新行为

- 在 `Microsoft.AspNetCore.Connections.Abstractions` 库中引入 `IConnectionListener` 接口以公开 `...Transport.Abstractions` 库最常用的功能。
- 现在提供传输选项 `NoDelay`（`LibuvTransportOptions` 和 `SocketTransportOptions`）。
- `SchedulingMode` 不再可用。

更改原因

ASP.NET Core 3.0 已从"pubternal"API 移出。

建议操作

类别

ASP.NET Core

受影响的 API

None

本地化：ResourceManagerWithCultureStringLocalizer 和 WithCulture 被标记为过时

ResourceManagerWithCultureStringLocalizer 类和 WithCulture 接口成员通常是造成本地化用户混淆的原因，尤其是在创建自己的 `IStringLocalizer` 实现时。这些项给用户留下了 `IStringLocalizer` 实例"按语言、按资源"的印象。实际上，实例应该仅"按资源"。搜索的语言由 `CultureInfo.CurrentUICulture` 在执行时确定。为了消除混淆来源，这些 API 在 ASP.NET Core 3.0 Preview 3 中被标记为过时。将从未来版本中删除这些 API。

有关上下文，请参阅 dotnet/aspnetcore#3324。有关讨论，请参阅 dotnet/aspnetcore#7756。

引入的版本

3.0

旧行为

方法未标记为 `Obsolete`。

新行为

方法被标记为 `Obsolete`。

更改原因

API 表示一个不推荐使用的用例。本地化设计存在混淆。

建议操作

建议改用 `ResourceManagerStringLocalizer`。允许 `CurrentCulture` 设置区域性。如果没有这个选项，请创建并使用 ResourceManagerWithCultureStringLocalizer 的副本。

类别

ASP.NET Core

受影响的 API

- ResourceManagerWithCultureStringLocalizer
- ResourceManagerStringLocalizer.WithCulture

日志记录：将 DebugLogger 类设为内部类

在 ASP.NET Core 3.0 之前，`DebugLogger` 的访问修饰符为 `public`。在 ASP.NET Core 3.0 中，访问修饰符更改为 `internal`。

引入的版本

3.0

更改原因

正在进行更改以便：

- 强制执行与其他记录器（如 `ConsoleLogger`）实现的一致性。
- 减少 API 图面。

建议操作

使用 AddDebug `ILoggingBuilder` 扩展方法来启用调试日志记录。如果需要手动注册服务，DebugLoggerProvider 也仍为 `public`。

类别

ASP.NET Core

受影响的 API

Microsoft.Extensions.Logging.Debug.DebugLogger

MVC：从控制器操作名称中剪裁的异步后缀

作为寻址 dotnet/aspnetcore#4849 的一部分，ASP.NET Core MVC 默认从操作名称中剪裁后缀 `Async`。从 ASP.NET Core 3.0 开始，这一更改会影响路由和链接生成。

引入的版本

3.0

旧行为

考虑以下 ASP.NET Core MVC 控制器：

```
public class ProductController : Controller
{
    public async IActionResult ListAsync()
    {
        var model = await DbContext.Products.ToListAsync();
        return View(model);
    }
}
```

此操作可通过 `Product/ListAsync` 进行路由。链接生成需要指定 `Async` 后缀。例如：

```
<a asp-controller="Product" asp-action="ListAsync">List</a>
```

新行为

在 ASP.NET Core 3.0 中，操作可通过 `Product/List` 进行路由。链接生成代码应省略 `Async` 后缀。例如：

```
<a asp-controller="Product" asp-action="List">List</a>
```

此更改不会影响使用 `[ActionName]` 属性指定的名称。可以通过在 `Startup.ConfigureServices` 中将 `MvcOptions.SuppressAsyncSuffixInActionNames` 设置为 `false` 来禁用新行为：

```
services.AddMvc(options =>
{
   options.SuppressAsyncSuffixInActionNames = false;
});
```

更改原因

按照约定，异步 .NET 方法以 `Async` 为后缀。但是，当方法定义 MVC 操作时，不需要使用 `Async` 后缀。

建议操作

如果应用依赖于保留名称的 `Async` 后缀的 MVC 操作，请选择下列缓解措施之一：

- 使用 `[ActionName]` 属性以保留原始名称。
- 通过在 `Startup.ConfigureServices` 中将 `MvcOptions.SuppressAsyncSuffixInActionNames` 设置为 `false` 来完全禁用重命名：

```
services.AddMvc(options =>
{
   options.SuppressAsyncSuffixInActionNames = false;
});
```

类别

ASP.NET Core

受影响的 API

None

MVC：JsonResult 已移至 Microsoft.AspNetCore.Mvc.Core

`JsonResult` 已移至 `Microsoft.AspNetCore.Mvc.Core` 程序集。此类型用于在 Microsoft.AspNetCore.Mvc.Formatters.Json 中进行定义。已将程序集级别 [TypeForwardedTo] 属性添加到 `Microsoft.AspNetCore.Mvc.Formatters.Json`，以便为大多数用户解决此问题。使用第三方库的应用可能会遇到问题。

引入的版本

3.0 预览版 6

旧行为

已成功生成使用基于 2.2 的库的应用。

新行为

使用基于 2.2 的库的应用无法进行编译。提供了包含以下文本变体的错误：

```
The type 'JsonResult' exists in both 'Microsoft.AspNetCore.Mvc.Core, Version=3.0.0.0, Culture=neutral,
PublicKeyToken=adb9793829ddae60' and 'Microsoft.AspNetCore.Mvc.Formatters.Json, Version=2.0.0.0,
Culture=neutral, PublicKeyToken=adb9793829ddae60'
```

有关此类问题的示例，请参阅 dotnet/aspnetcore#7220。

更改原因

按 aspnet/Announcements#325 所述，对 ASP.NET Core 组成进行平台级别的更改。

建议操作

根据 2.2 版本的 `Microsoft.AspNetCore.Mvc.Formatters.Json` 编译的库可能需要重新编译才能解决所有使用者的问题。如果受到影响，请与库作者联系。请求重新编译库以面向 ASP.NET Core 3.0。

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Mvc.JsonResult

MVC：已弃用预编译工具

在 ASP.NET Core 1.1 中，引入了 `Microsoft.AspNetCore.Mvc.Razor.ViewCompilation`（MVC 预编译工具）包以添加对发布时进行 Razor 文件（.cshtml 文件）编译的支持。在 ASP.NET Core 2.1 中，引入了 Razor SDK，以扩展预编译工具的功能。Razor SDK 添加了对生成时和发布时进行 Razor 文件编译的支持。SDK 在生成时验证 .cshtml 文件的正确性，同时缩短应用的启动时间。默认情况下，Razor SDK 处于启用状态，并且不需要任何手势即可开始使用。

在 ASP.NET Core 3.0 中，已删除 ASP.NET Core 1.1 时代的 MVC 预编译工具。早期的包版本将继续收到修补版本中的重要 Bug 和安全修补程序。

引入的版本

3.0

旧行为

`Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` 包用于预编译 MVC Razor 视图。

新行为

Razor SDK 本机支持此功能。`Microsoft.AspNetCore.Mvc.Razor.ViewCompilation` 包不再更新。

更改原因

Razor SDK 提供更多的功能并在生成时验证 .cshtml 文件的正确性。SDK 还会缩短应用的启动时间。

建议的操作

对于 ASP.NET Core 2.1 或更高版本的用户，更新以在 Razor SDK 中使用本机支持进行预编译。如果 Bug 或缺失的功能阻止迁移到 Razor SDK，请在 dotnet/aspnetcore 中提出问题。

类别

ASP.NET Core

受影响的 API

无

MVC："Pubternal"类型更改为内部

在 ASP.NET Core 3.0 中，MVC 中的所有“pubternal”类型都已更新为 `public`（在受支持的命名空间中）或 `internal`（根据需要）。

更改描述

在 ASP.NET Core 中，“pubternal”类型声明为 `public`，但驻留在后缀为 `.Internal` 的命名空间中。尽管这些类型为 `public`，但它们没有支持策略，并且可能会发生中断性变更。遗憾的是，经常会意外使用这些类型，导致对这些项目做出中断性变更，并使维护框架的能力受到限制。

引入的版本

3.0

旧行为

MVC 中的某些类型为 `public`，但在 `.Internal` 命名空间中。这些类型没有支持策略，并且可能会发生中断性变更。

新行为

所有此类类型都将更新为 `public`（在受支持的命名空间中）或标记为 `internal`。

更改原因

经常会意外使用“pubternal”类型，导致对这些项目做出中断性变更，并使维护框架的能力受到限制。

建议操作

如果使用的类型已真正成为 `public` 且已移动到新的受支持的命名空间中，请更新引用以匹配新命名空间。

如果使用的类型已标记为 `internal`，则需要查找替代方法。永远不支持将以前的“pubternal”类型用于公共用途。如果这些命名空间中的特定类型对应用至关重要，请在 dotnet/aspnetcore 中提出问题。可以考虑将请求的类型设为 `public`。

类别

ASP.NET Core

受影响的 API

此更改包括以下命名空间中的类型：

- `Microsoft.AspNetCore.Mvc.Cors.Internal`
- `Microsoft.AspNetCore.Mvc.DataAnnotations.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Json.Internal`
- `Microsoft.AspNetCore.Mvc.Formatters.Xml.Internal`
- `Microsoft.AspNetCore.Mvc.Internal`
- `Microsoft.AspNetCore.Mvc.ModelBinding.Internal`
- `Microsoft.AspNetCore.Mvc.Razor.Internal`
- `Microsoft.AspNetCore.Mvc.RazorPages.Internal`
- `Microsoft.AspNetCore.Mvc.TagHelpers.Internal`
- `Microsoft.AspNetCore.Mvc.ViewFeatures.Internal`

MVC：已删除 Web API 兼容性填充码

从 ASP.NET Core 3.0 开始，不再提供 `Microsoft.AspNetCore.Mvc.WebApiCompatShim` 包。

更改描述

`Microsoft.AspNetCore.Mvc.WebApiCompatShim` (WebApiCompatShim) 包在 ASP.NET Core 中提供了与 ASP.NET 4.x Web API 2 的部分兼容性，以简化将现有 Web API 实现迁移到 ASP.NET Core 的工作。但是，使用 WebApiCompatShim 的应用不会从最近 ASP.NET Core 版本中提供的与 API 相关的功能中获益。此类功能包括改进的开放 API 规范生成、标准化错误处理和客户端代码生成。为了更好地专注于 3.0 中的 API 工作，已删除 WebApiCompatShim。使用 WebApiCompatShim 的现有应用应迁移到较新的 `[ApiController]` 模型。

引入的版本

3.0

更改原因

Web API 兼容性填充码是一种迁移工具。它限制用户对 ASP.NET Core 中添加的新功能的访问。

建议操作

删除此填充码的使用，直接迁移到 ASP.NET Core 本身中的类似功能。

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Mvc.WebApiCompatShim

Razor：已删除 RazorTemplateEngine API

已删除 RazorTemplateEngine API，该内容已替换为 RazorProjectEngine。

有关讨论，请参阅 GitHub 问题 dotnet/aspnetcore#25215。

引入的版本

3.0

旧行为

可创建模板引擎并将其用于分析和生成 Razor 文件的代码。

新行为

可创建 `RazorProjectEngine`，使其提供与 `RazorTemplateEngine` 相同类型的信息，以分析和生成 Razor 文件的代码。 `RazorProjectEngine` 也可提供额外的配置级别。

更改原因

`RazorTemplateEngine` 与现有实现过于紧密耦合。在尝试正确配置 Razor 分析/生成管道时，这种紧密耦合会导致更多问题。

建议操作

请使用 `RazorProjectEngine`，而不是 `RazorTemplateEngine`。请考虑以下示例。

创建和配置 `RazorProjectEngine`

```
RazorProjectEngine projectEngine =
    RazorProjectEngine.Create(RazorConfiguration.Default,
        RazorProjectFileSystem.Create(@"C:\source\repos\ConsoleApp4\ConsoleApp4"),
        builder =>
        {
            builder.ConfigureClass((document, classNode) =>
            {
                classNode.ClassName = "MyClassName";

                // Can also configure other aspects of the class here.
            });

            // More configuration can go here
        });
```

生成 Razor 文件的代码

```
RazorProjectItem item = projectEngine.FileSystem.GetItem(
    @"C:\source\repos\ConsoleApp4\ConsoleApp4\Example.cshtml",
    FileKinds.Legacy);
RazorCodeDocument output = projectEngine.Process(item);

// Things available
RazorSyntaxTree syntaxTree = output.GetSyntaxTree();
DocumentIntermediateNode intermediateDocument =
    output.GetDocumentIntermediateNode();
RazorCSharpDocument csharpDocument = output.GetCSharpDocument();
```

类别

ASP.NET Core

受影响的 API

- RazorTemplateEngine
- RazorTemplateEngineOptions

Razor：运行时编译已移动到包

支持 Razor 视图的运行时编译，已将 Razor Pages 移动到单独的包中。

引入的版本

3.0

旧行为

无需额外的包，即可使用运行时编译。

新行为

此功能已移至 Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation 包中。

以下 API 以前在 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions` 中提供，用于支持运行时编译。现在可通过 `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation.MvcRazorRuntimeCompilationOptions` 获取 API。

- `RazorViewEngineOptions.FileProviders` 现在为 `MvcRazorRuntimeCompilationOptions.FileProviders`
- `RazorViewEngineOptions.AdditionalCompilationReferences` 现在为 `MvcRazorRuntimeCompilationOptions.AdditionalReferencePaths`

此外，已删除 `Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions.AllowRecompilingViewsOnFileChange`。默认情况下，通过引用 `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` 包来启用对文件更改的重新编译。

更改原因

此更改是删除 Roslyn 上的 ASP.NET Core 共享框架依赖项所必需的。

建议操作

需要对 Razor 文件进行运行时编译或重新编译的应用应执行以下步骤：

1. 添加对 `Microsoft.AspNetCore.Mvc.Razor.RuntimeCompilation` 包的引用。

2. 更新项目的 `Startup.ConfigureServices` 方法以包含对 `AddRazorRuntimeCompilation` 的调用。例如：

```
services.AddMvc()
    .AddRazorRuntimeCompilation();
```

类别

ASP.NET Core

受影响的 API

Microsoft.AspNetCore.Mvc.Razor.RazorViewEngineOptions

会话状态：已删除过时的 API

已删除用于配置会话 cookie 的过时 API。有关详细信息，请参阅 aspnet/Announcements#257。

引入的版本

3.0

更改原因

此更改强制实施跨 API 的一致性来配置使用 cookie 的功能。

建议操作

将已删除的 API 的使用迁移到其更新的替换项。请看下面 `Startup.ConfigureServices` 中的示例：

```
public void ConfigureServices(ServiceCollection services)
{
    services.AddSession(options =>
    {
        // Removed obsolete APIs
        options.CookieName = "SessionCookie";
        options.CookieDomain = "contoso.com";
        options.CookiePath = "/";
        options.CookieHttpOnly = true;
        options.CookieSecure = CookieSecurePolicy.Always;

        // new API
        options.Cookie.Name = "SessionCookie";
        options.Cookie.Domain = "contoso.com";
        options.Cookie.Path = "/";
        options.Cookie.HttpOnly = true;
        options.Cookie.SecurePolicy = CookieSecurePolicy.Always;
    });
}
```

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Builder.SessionOptions.CookieDomain
- Microsoft.AspNetCore.Builder.SessionOptions.CookieHttpOnly
- Microsoft.AspNetCore.Builder.SessionOptions.CookieName
- Microsoft.AspNetCore.Builder.SessionOptions.CookiePath
- Microsoft.AspNetCore.Builder.SessionOptions.CookieSecure

共享框架：从 Microsoft.AspNetCore.App 中删除了程序集

从 ASP.NET Core 3.0 开始，ASP.NET Core 共享框架（`Microsoft.AspNetCore.App`）只包含 Microsoft 完全开发、支持和可服务的第一方程序集。

更改描述

将这一更改视为重新定义 ASP.NET Core“平台”的边界。任何人都可通过 GitHub 生成共享框架的源代码，共享框架将继续为应用提供 .NET Core 共享框架的现有优势。一些优势包括部署大小更小、集中修补和启动时间更快。

作为此更改的一部分，`Microsoft.AspNetCore.App` 中引入了一些值得注意的重大更改。

引入的版本

3.0

旧行为

项目通过项目文件中的 `<PackageReference>` 元素引用了 `Microsoft.AspNetCore.App`。

此外，`Microsoft.AspNetCore.App` 包含以下子组件：

- Json.NET（`Newtonsoft.Json`）
- Entity Framework Core（以 `Microsoft.EntityFrameworkCore.` 为前缀的程序集）
- Roslyn（`Microsoft.CodeAnalysis`）

新行为

对 `Microsoft.AspNetCore.App` 的引用不再需要项目文件中的 `<PackageReference>` 元素。.NET Core SDK 支持名为 `<FrameworkReference>` 的新元素，该元素将替代对 `<PackageReference>` 的使用。

有关详细信息，请参阅 dotnet/aspnetcore#3612。

Entity Framework Core 作为 NuGet 包提供。此更改将使随附模型与 .NET 上的所有其他数据访问库保持一致。它在支持各种 .NET 平台的同时，为 Entity Framework Core 提供最简单的途径来继续创新。将 Entity Framework Core 移出共享框架不会影响其作为 Microsoft 开发、支持和可服务的库的状态。.NET Core 支持策略继续支持此功能。

Json.NET 和 Entity Framework Core 可继续使用 ASP.NET Core。但是，它们不会包含在共享框架中。

有关详细信息，请参阅 .NET Core 3.0 中 JSON 的未来。另请参阅从共享框架中删除的二进制文件的完整列表。

更改原因

此更改简化了 `Microsoft.AspNetCore.App` 的使用，并减少了 NuGet 包与共享框架之间的重复。

有关此更改动机的详细信息，请参阅博客文章。

建议操作

从 ASP.NET Core 3.0 开始，项目不需要使用 `Microsoft.AspNetCore.App` 中的程序集作为 NuGet 包。为了简化 ASP.NET Core 共享框架的定位和使用，已不再生成自 ASP.NET Core 1.0 以来提供的许多 NuGet 包。通过将 `<FrameworkReference>` 用于 `Microsoft.AspNetCore.App`，应用仍可使用这些包提供的 API。常见的 API 示例包括 Kestrel、MVC 和 Razor。

此更改不适用于通过 ASP.NET Core 2.x 中的 `Microsoft.AspNetCore.App` 引用的所有二进制文件。值得注意的例外包括：

- 继续以 .NET Standard 为目标的 `Microsoft.Extensions` 库将以 NuGet 包的形式提供（请参阅 https://github.com/dotnet/extensions）。
- 不属于 `Microsoft.AspNetCore.App` 的 ASP.NET Core 团队生成的 API。例如，以下组件以 NuGet 包的形式提供：
  - Entity Framework Core
  - 提供第三方集成的 API
  - 实验性功能
  - 包含无法满足共享框架中的要求的依赖项的 API
- 用于保留对 Json.NET 的支持的 MVC 扩展。API 作为 NuGet 包提供，以支持使用 Json.NET 和 MVC。有关更多详细信息，请参阅 ASP.NET Core 迁移指南。
- SignalR .NET 客户端继续支持 .NET Standard 并作为 NuGet 包提供。它可用于许多 .NET 运行时，例如 Xamarin 和 UWP。

有关详细信息，请参阅停止为 3.0 中的共享框架程序集生成包。有关讨论，请参阅 dotnet/aspnetcore#3757。

类别

ASP.NET Core

受影响的 API

- Microsoft.CodeAnalysis
- Microsoft.EntityFrameworkCore

共享框架：已删除 Microsoft.AspNetCore.All

从 ASP.NET Core 3.0 开始，不再生成 `Microsoft.AspNetCore.All` 元包和匹配的 `Microsoft.AspNetCore.All` 共享框架。此包在 ASP.NET Core 2.2 中提供，并将继续接收 ASP.NET Core 2.1 中的服务更新。

引入的版本

3.0

旧行为

应用可以使用 `Microsoft.AspNetCore.All` 元包来针对 .NET Core 上的 `Microsoft.AspNetCore.All` 共享框架。

新行为

.NET Core 3.0 不包含 `Microsoft.AspNetCore.All` 共享框架。

更改原因

`Microsoft.AspNetCore.All` 元包包含了大量外部依赖项。

建议操作

迁移项目以使用 `Microsoft.AspNetCore.App` 框架。以前在 `Microsoft.AspNetCore.All` 中可用的组件在 NuGet 上仍然可用。这些组件现可与应用一起部署，而不是包含在共享框架中。

类别

ASP.NET Core

受影响的 API

None

SignalR：HandshakeProtocol.SuccessHandshakeData 已替换

已删除 HandshakeProtocol.SuccessHandshakeData 字段，并将其替换为一个 Helper 方法，该方法根据特定 `IHubProtocol` 生成成功的握手响应。

引入的版本

3.0

旧行为

`HandshakeProtocol.SuccessHandshakeData` 是 `public static ReadOnlyMemory<byte>` 字段。

新行为

`HandshakeProtocol.SuccessHandshakeData` 已被 `static` `GetSuccessfulHandshake(IHubProtocol protocol)` 方法替换，该方法根据指定的协议返回 `ReadOnlyMemory<byte>`。

更改原因

握手响应 中添加了其他字段，这些字段是非常量，并会根据所选协议的不同而变化。

建议操作

无。此类型不适用于从用户代码使用。它是 `public` 类型，因此可以在 SignalR 服务器和客户端之间共享。它还可以由以 .NET 编写的客户 SignalR 客户端使用。SignalR 用户不应受此更改的影响 。

类别

ASP.NET Core

受影响的 API

HandshakeProtocol.SuccessHandshakeData

SignalR：已删除 HubConnection ResetSendPing 和 ResetTimeout 方法

已从 SignalR `HubConnection` API 中删除了 `ResetSendPing` 和 `ResetTimeout` 方法。这些方法最初仅供内部使用，但已在 ASP.NET Core 2.2 中公开。从 ASP.NET Core 3.0 预览版 4 开始，这些方法将不可用。有关讨论，请参阅 dotnet/aspnetcore#8543。

引入的版本

3.0

旧行为

API 可用。

新行为

API 已删除。

更改原因

这些方法最初仅供内部使用，但已在 ASP.NET Core 2.2 中公开。

建议操作

不要使用这些方法。

类别

ASP.NET Core

受影响的 API

- HubConnection.ResetSendPing()
- HubConnection.ResetTimeout()

SignalR：已更改 HubConnectionContext 构造函数

SignalR 的 `HubConnectionContext` 构造函数已更改为接受选项类型（而不是多个参数），以接受经得起未来考验的添加选项。此更改使用单个接受选项类型的构造函数替换两个构造函数。

引入的版本

3.0

旧行为

`HubConnectionContext` 有两个构造函数：

```
public HubConnectionContext(ConnectionContext connectionContext, TimeSpan keepAliveInterval, ILoggerFactory
loggerFactory);
public HubConnectionContext(ConnectionContext connectionContext, TimeSpan keepAliveInterval, ILoggerFactory
loggerFactory, TimeSpan clientTimeoutInterval);
```

新行为

这两个构造函数已被删除并已替换为一个构造函数：

```
public HubConnectionContext(ConnectionContext connectionContext, HubConnectionContextOptions contextOptions,
ILoggerFactory loggerFactory)
```

更改原因

新的构造函数使用新的选项对象。因此，可以在以后展开 `HubConnectionContext` 的功能，而无需执行更多的构造函数和中断性变更。

建议操作

而不是使用以下构造函数：

```
HubConnectionContext connectionContext = new HubConnectionContext(
    connectionContext,
    keepAliveInterval: TimeSpan.FromSeconds(15),
    loggerFactory,
    clientTimeoutInterval: TimeSpan.FromSeconds(15));
```

使用以下构造函数：

```
HubConnectionContextOptions contextOptions = new HubConnectionContextOptions()
{
    KeepAliveInterval = TimeSpan.FromSeconds(15),
    ClientTimeoutInterval = TimeSpan.FromSeconds(15)
};
HubConnectionContext connectionContext = new HubConnectionContext(connectionContext, contextOptions,
loggerFactory);
```

类别

ASP.NET Core

受影响的 API

- HubConnectionContext(ConnectionContext, TimeSpan, ILoggerFactory)
- HubConnectionContext(ConnectionContext, TimeSpan, ILoggerFactory, TimeSpan)

SignalR：已更改 JavaScript 客户端包名称

在 ASP.NET Core 3.0 预览版 7 中，SignalR JavaScript 客户端包名称从 `@aspnet/signalr` 更改为 `@microsoft/signalr`。由于 Azure SignalR 服务，名称更改反映了 SignalR 不只是在 ASP.NET Core 应用中有用这一事实。

若要对此更改做出反应，请更改 package.json 文件、`require` 语句和 ECMAScript `import` 语句中的引用。在此重命名过程中，不会更改 API。

有关讨论，请参阅 dotnet/aspnetcore#11637。

引入的版本

3.0

旧行为

客户端包以前命名为 `@aspnet/signalr`。

新行为

客户端包现在命名为 `@microsoft/signalr`。

更改原因

由于 Azure SignalR 服务，名称更改阐明了 SignalR 在 ASP.NET Core 应用之外也很有用。

建议操作

切换到新包 `@microsoft/signalr`。

类别

ASP.NET Core

受影响的 API

None

SignalR：UseSignalR 和 UseConnections 方法被标记为过时

在 ASP.NET Core 3.0 中，方法 `UseConnections` 和 `UseSignalR` 以及类 `ConnectionsRouteBuilder` 和 `HubRouteBuilder` 被标记为过时。

引入的版本

3.0

旧行为

SignalR 中心路由是使用 `UseSignalR` 或 `UseConnections` 配置的。

新行为

配置路由的旧方法已弃用，并已替换为终结点路由。

更改原因

正在将中间件移动到新的终结点路由系统。添加中间件的旧方法即将过时。

建议操作

将 `UseSignalR` 替换为 `UseEndpoints`：

旧代码：

```
app.UseSignalR(routes =>
{
    routes.MapHub<SomeHub>("/path");
});
```

新代码：

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapHub<SomeHub>("/path");
});
```

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Builder.ConnectionsAppBuilderExtensions.UseConnections(IApplicationBuilder, Action<ConnectionsRouteBuilder>)
- Microsoft.AspNetCore.Builder.SignalRAppBuilderExtensions.UseSignalR(IApplicationBuilder, Action<HubRouteBuilder>)
- Microsoft.AspNetCore.Http.Connections.ConnectionsRouteBuilder
- Microsoft.AspNetCore.SignalR.HubRouteBuilder

SPA：SpaServices 和 NodeServices 被标记为过时

自 ASP.NET Core 2.1 起，以下 NuGet 包的内容都是不必要的。因此，下列包将被标记为过时：

- Microsoft.AspNetCore.SpaServices
- Microsoft.AspNetCore.NodeServices

出于相同的原因，以下 npm 模块将被标记为已弃用：

- aspnet-angular
- aspnet-prerendering
- aspnet-webpack
- aspnet-webpack-react
- domain-task

上述包和 npm 模块稍后将从 .NET 5 中删除。

引入的版本

3.0

旧行为

已弃用的包和 npm 模块旨在将 ASP.NET Core 与各种单页应用 (SPA) 框架集成。此类框架包括 Angular、React 和 React Redux。

新行为

Microsoft.AspNetCore.SpaServices.Extensions NuGet 包中存在新的集成机制。自 ASP.NET Core 2.1 起，包仍然是 Angular 和 React 项目模板的基础。

更改原因

ASP.NET Core 支持与各种单页应用 (SPA) 框架（包括 Angular、React 和 React Redux）集成。最初，与这些框架的集成是通过 ASP.NET Core 特定组件实现的，这些组件用于处理服务器端预呈现和与 Webpack 集成等情况。随着时间的推移，行业标准也发生了变化。每个 SPA 框架都发布了其自己的标准命令行接口。例如，Angular CLI 和 create-react-app。

当 ASP.NET Core 2.1 于 2018 年 5 月发布时，团队对标准的变化做出了响应。提供了一种更新且更简单的方法来与 SPA 框架自身的工具链集成。自 ASP.NET Core 2.1 起，包 `Microsoft.AspNetCore.SpaServices.Extensions` 中存在新的集成机制，且仍是 Angular 和 React 项目模板的基础。

为了阐明较低版本的 ASP.NET Core 特定组件不相关且不建议使用，请执行以下操作：

- 2.1 之前的集成机制被标记为已过时。
- 支持 npm 包被标记为已弃用。

建议操作

如果正在使用这些包，请更新应用以使用以下功能：

- 在 `Microsoft.AspNetCore.SpaServices.Extensions` 包中。
- 由使用的 SPA 框架提供

若要启用服务器端预呈现和热模块重载等功能，请参阅相应的 SPA 框架的文档。
`Microsoft.AspNetCore.SpaServices.Extensions` 中的功能未过时，将继续受到支持。

类别

ASP.NET Core

受影响的 API

- Microsoft.AspNetCore.Builder.SpaRouteExtensions
- Microsoft.AspNetCore.Builder.WebpackDevMiddleware
- Microsoft.AspNetCore.NodeServices.EmbeddedResourceReader
- Microsoft.AspNetCore.NodeServices.INodeServices
- Microsoft.AspNetCore.NodeServices.NodeServicesFactory
- Microsoft.AspNetCore.NodeServices.NodeServicesOptions
- Microsoft.AspNetCore.NodeServices.StringAsTempFile
- Microsoft.AspNetCore.NodeServices.HostingModels.INodeInstance
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationException
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeInvocationInfo
- Microsoft.AspNetCore.NodeServices.HostingModels.NodeServicesOptionsExtensions
- Microsoft.AspNetCore.NodeServices.HostingModels.OutOfProcessNodeInstance
- Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerenderer
- Microsoft.AspNetCore.SpaServices.Prerendering.ISpaPrerendererBuilder
- Microsoft.AspNetCore.SpaServices.Prerendering.JavaScriptModuleExport
- Microsoft.AspNetCore.SpaServices.Prerendering.Prerenderer
- Microsoft.AspNetCore.SpaServices.Prerendering.PrerenderTagHelper
- Microsoft.AspNetCore.SpaServices.Prerendering.RenderToStringResult
- Microsoft.AspNetCore.SpaServices.Webpack.WebpackDevMiddlewareOptions
- Microsoft.Extensions.DependencyInjection.NodeServicesServiceCollectionExtensions
- Microsoft.Extensions.DependencyInjection.PrerenderingServiceCollectionExtensions

---

SPA：SpaServices 和 NodeServices 不再回退到控制台记录器

除非配置了日志记录，否则 Microsoft.AspNetCore.SpaServices 和 Microsoft.AspNetCore.NodeServices 不会显示控制台日志。

引入的版本

3.0

旧行为

`Microsoft.AspNetCore.SpaServices` 和 `Microsoft.AspNetCore.NodeServices` 用于在未配置日志记录时自动创建控制台记录器。

新行为

除非配置了日志记录，否则 `Microsoft.AspNetCore.SpaServices` 和 `Microsoft.AspNetCore.NodeServices` 不会显示控制台日志。

更改原因

需要与其他 ASP.NET Core 包实现日志记录的方式保持一致。

建议操作

如果需要旧行为，若要配置控制台日志记录，请将 `services.AddLogging(builder => builder.AddConsole())` 添加到 `Setup.ConfigureServices` 方法。

类别

ASP.NET Core

受影响的 API

None

目标框架：删除了 .NET Framework 支持

从 ASP.NET Core 3.0 开始，.NET Framework 不再是受支持的目标框架。

更改描述

.NET Framework 4.8 是 .NET Framework 的上一个主要版本。新的 ASP.NET Core 应用应基于 .NET Core 构建。从 .NET Core 3.0 版开始，可将 ASP.NET Core 3.0 视为 .NET Core 的一部分。

将 .NET Framework 和 ASP.NET Core 结合使用的客户可以使用 2.1 LTS 版本以完全受支持的方式继续。至少在 2021 年 8 月 21 日之前，将继续提供对 2.1 的支持和服务。根据 .NET 支持策略，此日期为声明 LTS 版本后三年。.NET Framework 对 ASP.NET Core 2.1 包的支持将无限延期，类似于其他基于包的 ASP.NET 框架的服务策略。

有关从 .NET Framework 移植到 .NET Core 的详细信息，请参阅移植到 .NET Core。

`Microsoft.Extensions` 包(如日志记录、依赖项注入和配置)和 Entity Framework Core 不受影响。它们将继续支持 .NET Standard。

有关此更改动机的详细信息，请参阅原始博客文章。

引入的版本

3.0

旧行为

ASP.NET Core 应用可在 .NET Core 或 .NET Framework 上运行。

新行为

ASP.NET Core 应用只能在 .NET Core 上运行。

建议的操作

请执行以下一项操作：

- 将应用保留在 ASP.NET Core 2.1 上。
- 将应用和依赖项迁移到 .NET Core。

类别

ASP.NET Core

受影响的 API

无

Core .NET 库

- 报告版本的 API 现在报告产品版本而不是文件版本
- 自定义 EncoderFallbackBuffer 实例无法递归回退
- 浮点格式设置和分析行为变更
- 浮点分析操作不再失败或引发 OverflowException
- InvalidAsynchronousStateException 已移到另一个程序集
- 替换格式错误的 UTF-8 字节序列将遵循 Unicode 准则
- TypeDescriptionProviderAttribute 已移到另一个程序集
- ZipArchiveEntry 不再处理条目大小不一致的存档
- FieldInfo.SetValue 将对静态、仅初始化字段引发异常
- 将 GroupCollection 传递到采用 IEnumerable<T> 的扩展方法需要消除歧义

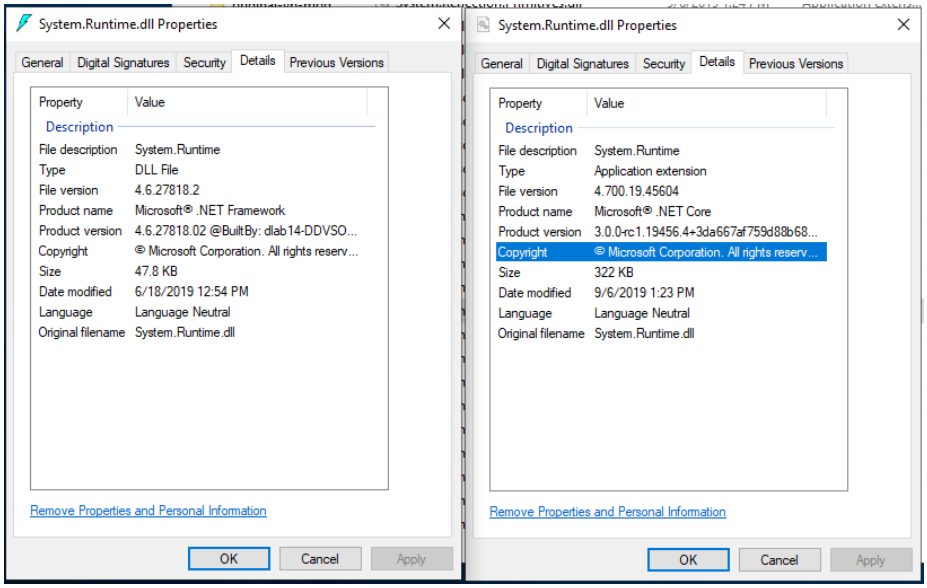
报告版本的 API 现报告的是产品版本而不是文件版本

返回 .NET Core 中的版本的很多 API 现在返回的是产品版本，而不是文件版本。

更改描述

在 .NET Core 2.2 及更低版本中，Environment.Version 和 RuntimeInformation.FrameworkDescription 等方法以及 .NET Core 程序集的“文件属性”对话框反映的都是文件版本。自 .NET Core 3.0 起，它们反映的是产品版本。

下图展示了由“Windows 资源管理器”文件属性对话框显示的 .NET Core 2.2(左侧)与 .NET Core 3.0(右侧)的 System.Runtime.dll 程序集版本信息区别 。



引入的版本

3.0

建议操作

无。此更改会使版本检测变得直观而非退化。

类别

Core .NET 库

受影响的 API

- Environment.Version
- RuntimeInformation.FrameworkDescription

自定义 EncoderFallbackBuffer 实例无法递归回退

自定义 EncoderFallbackBuffer 实例无法以递归方式回退。EncoderFallbackBuffer.GetNextChar() 的实现必须生成一个可转换为目标编码的字符序列。否则会发生异常。

更改描述

在字符到字节的转码操作期间，运行时将检测格式不正确或不可转换的 UTF-16 序列，并将这些字符提供给 EncoderFallbackBuffer.Fallback 方法。`Fallback` 方法确定应将哪些字符替换为原始不可转换数据，并通过在循环中调用 EncoderFallbackBuffer.GetNextChar 来释放这些字符。

然后，运行时尝试将这些替换字符转码为目标编码。如果此操作成功，则运行时继续从原始输入字符串中的中断位置进行转码。

之前，EncoderFallbackBuffer.GetNextChar() 的自定义实现可以返回无法转换为目标编码的字符序列。如果替换字符无法转码为目标编码，则运行时将使用替换字符再调用一次 EncoderFallbackBuffer.Fallback 方法，并要求 EncoderFallbackBuffer.GetNextChar() 方法返回新的替换序列。此过程将一直继续，直到运行时最终看到格式正确的、可转换的替换，或直到达到最大递归计数。

从 .NET Core 3.0 开始，EncoderFallbackBuffer.GetNextChar() 的自定义实现必须返回可转换为目标编码的字符序列。如果替换字符无法转码为目标编码，则引发 ArgumentException。运行时将不再对 EncoderFallbackBuffer 实例进行递归调用。

仅当满足以下所有三个条件时，此行为才适用：

- 运行时检测格式不正确的 UTF-16 序列或无法转换为目标编码的 UTF-16 序列。
- 已指定自定义 EncoderFallback。
- 自定义 EncoderFallback 尝试替换新的格式不正确的或无法转换的 UTF-16 序列。

引入的版本

3.0

建议操作

大多数开发人员都不需要执行任何操作。

如果应用程序使用自定义 EncoderFallback 和 EncoderFallbackBuffer 类，请确保 EncoderFallbackBuffer.Fallback 的实现使用格式正确的 UTF-16 数据（该数据在运行时第一次调用 Fallback 方法时可直接转换为目标编码）填充回退缓冲区。

类别

Core .NET 库

受影响的 API

- EncoderFallbackBuffer.Fallback
- EncoderFallbackBuffer.GetNextChar()

浮点格式设置和分析行为已更改

浮点分析和格式设置行为（由 Double 和 Single 类型实现）现符合 IEEE 标准。这可确保 .NET 中浮点类型的行为与符合 IEEE 标准的语言的行为一致。例如，`double.Parse("SomeLiteral")` 应始终与 C# 为 `double x = SomeLiteral` 生成的结果匹配。

更改描述

在 .NET Core 2.2 及更低版本中，通过 Double.ToString 和 Single.ToString 进行格式设置的行为，以及使用 Double.Parse、Double.TryParse、Single.Parse 和 Single.TryParse 进行分析的行为不符合 IEEE 标准。因此，没法保证值在没有任何受支持的标准或自定义格式字符串的情况下双向传输。对于某些输入，尝试分析已设置格式的值可能会失败；而对于其他输入，分析后的值与原始值不相等。

自 .NET Core 3.0 起，浮点分析和格式设置操作均符合 IEEE 754 标准。

下表显示了两个代码片段，以及 .NET Core 2.2 和 .NET Core 3.1 之间的输出更改情况。

代码	.NET Core 2.2 结果	.NET Core 3.1 结果
`Console.WriteLine((-0.0).ToString());`	0	-0
`var value = -3.123456789123456789;` `Console.WriteLine(value == double.Parse(value.ToString()));`	`False`	`True`

有关浮点改进的详细信息，请参阅 Floating-point parsing and formatting improvements in .NET Core 3.0（.NET Core 3.0 中浮点分析和格式设置改进）博客文章。

引入的版本

3.0

建议操作

.NET Core 3.0 中浮点分析和格式设置改进博客文章的"对现有代码的潜在影响"部分建议，如果要维护以前的行为，可以对代码进行一些更改。

- 对于格式设置中的一些差异，可以通过指定不同的格式字符串获得与以前行为等效的行为。
- 对于分析方面的差异，没有回退到以前行为的机制。

类别

Core .NET 库

受影响的 API

- Double.ToString
- Single.ToString
- Double.Parse
- Double.TryParse
- Single.Parse
- Single.TryParse

浮点分析操作不再失败或引发 OverflowException

浮点分析方法在分析数值超出 Single 或 Double 浮点类型范围的字符串时，不再引发 OverflowException 或返回 `false`。

更改描述

在 .NET Core2.2 和早期版本中，Double.Parse 和 Single.Parse 方法对超出其各自类型范围的值会引发 OverflowException。对于超出范围的数值的字符串表示形式，Double.TryParse 和 Single.TryParse 方法返回 `false`。

从 .NET Core 3.0 开始，分析超出范围的数值字符串时，Double.Parse、Double.TryParse、Single.Parse 和 Single.TryParse 方法不再失败。相反，Double 分析方法对于超过 Double.MaxValue 的值返回 Double.PositiveInfinity，对于小于 Double.MinValue 的值返回 Double.NegativeInfinity。同样，Single 分析方法对于超过 Single.MaxValue 的值返回 Single.PositiveInfinity，对于小于 Single.MinValue 的值返回 Single.NegativeInfinity。

此更改是为了改进 IEEE 754:2008 的符合性。

引入的版本

3.0

建议操作

此更改会以两种方式之一影响你的代码：

- 你的代码依赖于 OverflowException 在发生溢出时执行的处理程序。在这种情况下，应删除 `catch` 语句，并在 `If` 语句中放置必要的代码，以测试 Double.IsInfinity 或 Single.IsInfinity 是否为 `true`。
- 代码假设浮点值不是 `Infinity`。在这种情况下，应添加必要的代码来检查 `PositiveInfinity` 和 `NegativeInfinity` 的浮点值。

类别

Core .NET 库

受影响的 API

- Double.Parse
- Double.TryParse
- Single.Parse
- Single.TryParse

InvalidAsynchronousStateException 已移到另一程序集

InvalidAsynchronousStateException 类已移动。

更改描述

在 .NET Core 2.2 及更低版本中，InvalidAsynchronousStateException 位于 System.ComponentModel.TypeConverter 程序集中 。

而自 .NET Core 3.0 起，它位于 System.ComponentModel.Primitives 程序集中 。

引入的版本

3.0

建议操作

此更改仅影响这样的应用程序，它们通过调用 Assembly.GetType 等方法或调用假设类型位于特定程序集中的 Activator.CreateInstance 的重载，使用反射过程来加载 InvalidAsynchronousStateException。如果是这种情况，可以更新在方法调用中引用的程序集，以反映出类型的新程序集位置。

类别

Core .NET 库

受影响的 API

无。

替换格式错误的 UTF-8 字节序列将遵循 Unicode 准则

UTF8Encoding 类在字节到字符转码操作期间遇到格式错误的 UTF-8 字节序列时，它将在输出字符串中用“&#xFFFD;”(U+FFFD 替换字符)替换该序列。.NET Core 3.0 与以前版本的 .NET Core 和 .NET Framework 的不同之处在于，在转码操作期间按照 Unicode 最佳做法执行此替换。

这是在整个 .NET 中改进 UTF-8 处理的较大工作量(包括通过新的 System.Text.Unicode.Utf8 和 System.Text.Rune 类型)的一部分。为 UTF8Encoding 类型提供了改进的错误处理机制，以便生成与新引入的类型一致的输出。

更改描述

从 .NET Core 3.0 开始，当将字节转码为字符时，UTF8Encoding 类会根据 Unicode 最佳做法执行字符替换。Unicode 标准 12.0 版第 3.9 节 (PDF) 的 “U+FFFD 替换最大子部分”标题中描述了使用的替换机制。

仅当 输入字节序列包含格式错误的 UTF-8 数据时，此行为才适用。此外，如果已通过 `throwOnInvalidBytes: true` 构造了 UTF8Encoding 实例，则 `UTF8Encoding` 实例将继续对无效输入引发，而不是执行 U+FFFD 替换。有关 `UTF8Encoding` 构造函数的详细信息，请参阅 UTF8Encoding(Boolean, Boolean)。

下表通过无效的 3 字节输入说明了此更改的影响：

ㄱㄱㄱㄱ 3 ㄱㄱㄱㄱ	.NET CORE 3.0 ㄱㄱㄱㄱ	ㄷ .NET CORE 3.0 ㄱㄱㄱㄱ
`[ ED A0 90 ]`	`[ FFFD FFFD ]`(2 字符输出)	`[ FFFD FFFD FFFD ]`(3 字符输出)

根据以前链接的 Unicode 标准 PDF 的表 3-9，此 3 字符输出为首选输出。

引入的版本

3.0

建议操作

开发人员一方不需要执行任何操作。

类别

Core .NET 库

受影响的 API

- UTF8Encoding.GetCharCount
- UTF8Encoding.GetChars
- UTF8Encoding.GetString(Byte[], Int32, Int32)

TypeDescriptionProviderAttribute 已移到另一程序集

TypeDescriptionProviderAttribute 类已移动。

更改描述

在 .NET Core 2.2 及更低版本中，TypeDescriptionProviderAttribute 位于 System.ComponentModel.TypeConverter 程序集中 。

而自 .NET Core 3.0 起，它位于 System.ObjectModel 程序集中 。

引入的版本

3.0

建议操作

此更改仅影响这样的应用程序，它们通过调用 Assembly.GetType 等方法或调用假设类型位于特定程序集中的 Activator.CreateInstance 的重载，使用反射过程来加载 TypeDescriptionProviderAttribute 类型。若是如此，应更新在方法调用反射的程序集，以反射出类型的新程序集位置。

类别

Windows 窗体

受影响的 API

无。

ZipArchiveEntry 不再处理具有不一致条目大小的存档

Zip 存档在中央目录和本地标头中列出压缩的大小和未压缩的大小。条目数据本身还指示其大小。在 .NET Core 2.2 及更早版本中，永远不会对这些值进行一致性检查。从 .NET Core 3.0 开始，对它们进行一致性检查。

更改描述

在 .NET Core 2.2 及更早版本中，即使本地标头与 zip 文件的中央标头不一致，ZipArchiveEntry.Open() 也会成功。即使数据长度超出了中央目录/本地标头中列出的未压缩文件大小，数据也会一直进行解压缩，直到达到压缩流的末尾。

从 .NET Core 3.0 开始，ZipArchiveEntry.Open() 方法会检查本地标头和中央标头是否在条目的压缩大小和未压缩大小方面保持一致。如果不是这样，则该方法在存档的本地标头和/或数据描述符列表大小与 zip 文件的中央目录不一致时会引发 InvalidDataException。当读取条目时，解压缩的数据将被截断为标头中列出的未压缩文件大

小。

进行此更改是为了确保 ZipArchiveEntry 正确表示其数据的大小并且只读取该数据量。

引入的版本

3.0

建议操作

对出现这些问题的所有 zip 存档重新打包。

类别

Core .NET 库

受影响的 API

- ZipArchiveEntry.Open()
- ZipFileExtensions.ExtractToDirectory
- ZipFileExtensions.ExtractToFile
- ZipFile.ExtractToDirectory

FieldInfo.SetValue 将对静态、仅初始化字段引发异常

从 .NET Core 3.0 开始，当你尝试通过调用 System.Reflection.FieldInfo.SetValue 在静态 InitOnly 字段上设置值时，将引发异常。

更改描述

在 .NET Framework 和 3.0 之前的 .NET Core 版本中，你可以通过调用 System.Reflection.FieldInfo.SetValue 来设置初始化后为常量的静态字段的值(C# 中的 readonly)。但是，以这种方式设置此类字段导致了不可预测的行为，具体取决于目标框架和优化设置。

在 .NET Core 3.0 及更高版本中，当对静态 InitOnly 字段调用 SetValue 时，将引发 System.FieldAccessException 异常。

> **TIP**
> InitOnly 字段是只能在声明它时或位于包含类的构造函数中时设置的字段。换句话说，它在初始化后为常量。

引入的版本

3.0

建议操作

初始化静态构造函数中的静态 InitOnly 字段。这同时适用于动态和非动态类型。

或者，可以从字段中删除 FieldAttributes.InitOnly 属性，然后调用 FieldInfo.SetValue。

类别

Core .NET 库

受影响的 API

- FieldInfo.SetValue(Object, Object)
- FieldInfo.SetValue(Object, Object, BindingFlags, Binder, CultureInfo)

将 GroupCollection 传递到采用 IEnumerable<T> 的扩展方法需要消除歧义

调用在 GroupCollection 上采用 `IEnumerable<T>` 的扩展方法时，必须使用强制转换来消除该类型的歧义。

更改说明

从 .NET Core 3.0 开始，System.Text.RegularExpressions.GroupCollection 除了实现 `IEnumerable<Group>` 等类型之外，还会实现 `IEnumerable<KeyValuePair<String,Group>>`。调用采用 IEnumerable<T> 的扩展方法时，这会导致歧义。如果在 GroupCollection 实例上调用此类扩展方法，例如 Enumerable.Count，你将看到以下编译器错误：

CS1061："GroupCollection"未包含"Count"的定义，并且找不到可接受第一个"GroupCollection"类型参数的可访问扩展方法"Count"(是否缺少 using 指令或程序集引用？)

在早期版本的 .NET 中，没有歧义，也没有编译器错误。

引入的版本

3.0

更改原因

这是意外的中断性变更。由于此更改已经有一段时间了，所以我们不打算将其还原。此外，此类更改本身就会造成中断。

建议操作

对于 GroupCollection 实例，使用强制转换对接受 `IEnumerable<T>` 的扩展方法的调用消除歧义。

```
// Without a cast - causes CS1061.
match.Groups.Count(_ => true)

// With a disambiguating cast.
((IEnumerable<Group>)m.Groups).Count(_ => true);
```

类别

Core .NET 库

受影响的 API

任何接受 IEnumerable<T> 的扩展方法都会受到影响。例如：

- System.Collections.Immutable.ImmutableArray.ToImmutableArray<TSource>(IEnumerable<TSource>)
- System.Collections.Immutable.ImmutableDictionary.ToImmutableDictionary
- System.Collections.Immutable.ImmutableHashSet.ToImmutableHashSet
- System.Collections.Immutable.ImmutableList.ToImmutableList<TSource>(IEnumerable<TSource>)
- System.Collections.Immutable.ImmutableSortedDictionary.ToImmutableSortedDictionary
- System.Collections.Immutable.ImmutableSortedSet.ToImmutableSortedSet
- System.Data.DataTableExtensions.CopyToDataTable
- 大多数 `System.Linq.Enumerable` 方法，例如 System.Linq.Enumerable.Count
- System.Linq.ParallelEnumerable.AsParallel
- System.Linq.Queryable.AsQueryable

密码

- Linux 不再支持 BEGIN TRUSTED CERTIFICATE 语法
- EnvelopedCms 默认为 AES-256 加密
- RSAOpenSsl 密钥生成的最小大小已增加
- .NET Core 3.0 倾向于使用 OpenSSL 1.1.x 而不是 OpenSSL 1.0.x
- CryptoStream.Dispose 仅在写入时转换最终块

Linux 上的根证书不再支持“BEGIN TRUSTED CERTIFICATE”语法

Linux 和其他类似 Unix 的系统（但不是 macOS）上的根证书可以采用两种形式显示：标准 `BEGIN CERTIFICATE` PEM 标头和特定于 OpenSSL 的 `BEGIN TRUSTED CERTIFICATE` PEM 标头。后一种语法允许进行其他配置，这些配置已导致与 .NET Core 的 System.Security.Cryptography.X509Certificates.X509Chain 类之间的兼容性问题。从 .NET Core 3.0 开始，链引擎不再加载 `BEGIN TRUSTED CERTIFICATE` 根证书内容。

更改描述

以前，`BEGIN CERTIFICATE` 和 `BEGIN TRUSTED CERTIFICATE` 语法均用于填充根信任列表。如果使用了 `BEGIN TRUSTED CERTIFICATE` 语法，并且在文件中指定了其他选项，则 X509Chain 可能报告已明确禁止链信任 (X509ChainStatusFlags.ExplicitDistrust)。但是，如果在以前加载的文件中同时使用 `BEGIN CERTIFICATE` 语法指定了证书，则允许链信任。

从 .NET Core 3.0 开始，不再读取 `BEGIN TRUSTED CERTIFICATE` 内容。如果还没有通过标准 `BEGIN CERTIFICATE` 语法指定证书，则 X509Chain 会报告根不受信任 (X509ChainStatusFlags.UntrustedRoot)。

引入的版本

3.0

建议操作

大多数应用程序不受此更改的影响，但是由于权限问题而无法同时看到两个根证书源的应用程序可能会在升级后遇到意外的 `UntrustedRoot` 错误。

许多 Linux 分发版（或发行版）将根证书写入两个位置：每个文件一个证书目录和一个文件串联。在某些发行版上，每个文件一个证书目录使用 `BEGIN TRUSTED CERTIFICATE` 语法，而一个文件串联则使用标准 `BEGIN CERTIFICATE` 语法。请确保将任何自定义根证书作为 `BEGIN CERTIFICATE` 添加到这些位置中的至少一个位置，并且你的应用程序可以读取这两个位置。

典型的目录是 /etc/ssl/certs/ /，典型的串联文件是 /etc/ssl/cert.pem。使用命令 `openssl version -d` 来确定特定于平台的根目录，这可能不同于 /etc/ssl/。例如，在 Ubuntu 18.04 上，目录是 /usr/lib/ssl/certs/，文件是 /usr/lib/ssl/cert.pem。不过，/usr/lib/ssl/certs/ 是 /etc/ssl/certs/ 的符号链接，并且 /usr/lib/ssl/cert.pem 不存在。

```
$ openssl version -d
OPENSSLDIR: "/usr/lib/ssl"
$ ls -al /usr/lib/ssl
total 12
drwxr-xr-x  3 root root 4096 Dec 12 17:10 .
drwxr-xr-x 73 root root 4096 Feb 20 15:18 ..
lrwxrwxrwx  1 root root   14 Mar 27  2018 certs -> /etc/ssl/certs
drwxr-xr-x  2 root root 4096 Dec 12 17:10 misc
lrwxrwxrwx  1 root root   20 Nov 12 16:58 openssl.cnf -> /etc/ssl/openssl.cnf
lrwxrwxrwx  1 root root   16 Mar 27  2018 private -> /etc/ssl/private
```

类别

密码

受影响的 API

- System.Security.Cryptography.X509Certificates.X509Chain

EnvelopedCms 默认为 AES-256 加密

`EnvelopedCms` 使用的默认对称加密算法已由 TripleDES 改为 AES-256。

更改描述

在以前的版本中，如果 EnvelopedCms 用于对数据加密但未通过构造函数重载指定对称加密算法，则使用 TripleDES/3DES/3DEA/DES3-EDE 算法对数据加密。

从 .NET Core 3.0 开始（通过 System.Security.Cryptography.Pkcs NuGet 包的 4.6.0 版本），默认算法已改为 AES-256 以实现算法现代化并提高默认选项的安全性。如果消息收件人证书具有（非 EC）Diffie-Hellman 公钥，则由于底层平台的限制，加密操作可能会失败并显示 CryptographicException。

在下面的示例代码中，如果在 .NET Core 2.2 或更早版本上运行，则使用 TripleDES 对数据加密。如果在.NET Core 3.0 或更高版本上运行，则使用 AES-256 对数据加密。

```
EnvelopedCms cms = new EnvelopedCms(content);
cms.Encrypt(recipient);
return cms.Encode();
```

引入的版本

3.0

建议操作

如果该更改有负面影响，则可以通过在包含 AlgorithmIdentifier 类型参数的 EnvelopedCms 构造函数中显式指定加密算法标识符来还原 TripleDES 加密，例如：

```
Oid tripleDesOid = new Oid("1.2.840.113549.3.7", null);
AlgorithmIdentifier tripleDesIdentifier = new AlgorithmIdentifier(tripleDesOid);
EnvelopedCms cms = new EnvelopedCms(content, tripleDesIdentifier);

cms.Encrypt(recipient);
return cms.Encode()l
```

类别

密码

受影响的 API

- EnvelopedCms()
- EnvelopedCms(ContentInfo)
- EnvelopedCms(SubjectIdentifierType, ContentInfo)

RSAOpenSsl 密钥生成的最小大小已增加

在 Linux 上生成新 RSA 密钥的最小大小已从 384 位提高到 512 位。

更改描述

自 .NET Core 3.0 起，Linux 上 RSA.CreateRSAOpenSsl 和 RSACryptoServiceProvider 中 RSA 实例上的 `LegalKeySizes` 属性报告的最低合法密钥大小已从 384 增加到 512。

因此，在 .NET Core 2.2 及更早版本中，方法调用（如 `RSA.Create(384)`）会成功。在 .NET Core 3.0 及更高版本中，方法调用 `RSA.Create(384)` 会引发异常，指示大小太小。

此更改是因为在 Linux 上执行加密操作的 OpenSSL 在版本 1.0.2 与 1.1.0 之间提高了其最小值。.NET Core 3.0 倾向于使用 OpenSSL 1.1.x 而不是 1.0.x，并提高了最小报告版本来反映这一新的更高的依赖项限制。

引入的版本

3.0

建议操作

如果调用任何受影响的 API，请确保所有生成的密钥的大小不小于提供程序的最小值。

> **NOTE**
>
> 384 位 RSA 已被视为不安全（512 位 RSA 也是如此）。新式建议（例如 NIST 特别出版物 800-57 第 1 部分修订版 4）建议将 2048 位作为新生成的密钥的最小大小。

类别

密码

受影响的 API

- AsymmetricAlgorithm.LegalKeySizes
- RSA.Create
- RSAOpenSsl
- RSACryptoServiceProvider

.NET Core 3.0 倾向于使用 OpenSSL 1.1.x 而不是 OpenSSL 1.0.x

跨多个 Linux 分发工作的适用于 Linux 的 .NET Core 可同时支持 OpenSSL 1.0.x 和 OpenSSL 1.1.x。.NET Core 2.1

和 .NET Core 2.2 首先查找 1.0.x，然后回退到 1.1.x；.NET Core 3.0 首先查找 1.1.x。进行此更改是为了增加对新加密标准的支持。

此更改可能会影响库或应用程序，这些库或应用程序与 .NET Core 中的 OpenSSL 特定互操作类型进行平台互操作。

更改描述

在 .NET Core 2.2 及更早版本中，运行时倾向于加载 OpenSSL 1.0.x 而不是加载 1.1.x。这意味着，与 OpenSSL 互操作的 IntPtr 和 SafeHandle 类型倾向于与 libcrypto.so.1.0.0 / libcrypto.so.1.0 / libcrypto.so.10 一起使用。

从 .NET Core 3.0 开始，运行时倾向于加载 OpenSSL 1.1.x 而不是 OpenSSL 1.0.x，因此与 OpenSSL 互操作的 IntPtr 和 SafeHandle 类型倾向于与 libcrypto.so.1.1 / libcrypto.so.11 / libcrypto.so.1.1.0 / libcrypto.so.1.1.1 一起使用。因此，从 .NET Core 2.1 或 .NET Core 2.2 升级时，与 OpenSSL 直接互操作的库和应用程序的指针可能与 .NET Core 公开的值不兼容。

引入的版本

3.0

建议操作

需要小心使用直接与 OpenSSL 操作的库和应用程序，确保它们使用的 OpenSSL 版本与 .NET Core 运行时相同。

所有将 .NET Core 加密类型中的 IntPtr 或 SafeHandle 值直接用于 OpenSSL 的库或应用程序都应当将其使用的库的版本与新的 SafeEvpPKeyHandle.OpenSslVersion 属性进行比较，以确保指针兼容。

类别

密码

受影响的 API

- SafeEvpPKeyHandle
- RSAOpenSsl(IntPtr)
- RSAOpenSsl(SafeEvpPKeyHandle)
- RSAOpenSsl.DuplicateKeyHandle()
- DSAOpenSsl(IntPtr)
- DSAOpenSsl(SafeEvpPKeyHandle)
- DSAOpenSsl.DuplicateKeyHandle()
- ECDsaOpenSsl(IntPtr)
- ECDsaOpenSsl(SafeEvpPKeyHandle)
- ECDsaOpenSsl.DuplicateKeyHandle()
- ECDiffieHellmanOpenSsl(IntPtr)
- ECDiffieHellmanOpenSsl(SafeEvpPKeyHandle)
- ECDiffieHellmanOpenSsl.DuplicateKeyHandle()
- X509Certificate.Handle

CryptoStream.Dispose 仅在写入时转换最终块

用于完成 `CryptoStream` 操作的 CryptoStream.Dispose 方法不再尝试在读取时转换最终块。

更改说明

在以前的 .NET 版本中，如果用户在 Read 模式下使用 CryptoStream 时执行了不完整的读取操作，Dispose 方法可能会引发异常（例如，使用带有填充的 AES 时）。引发异常是因为尝试转换最终块，但数据不完整。

在 .NET Core 3.0 及更高版本中，Dispose 不再尝试在读取时转换最终块，这会允许执行不完整的读取操作。

更改原因

由于此更改，当取消网络操作后，将允许从加密流中进行不完整的读取操作，而无需捕获异常。

引入的版本

3.0

建议操作

大多数应用都不会受到此更改的影响。

如果应用程序先前在读取不完整的情况下捕获到异常，你可以删除相应 `catch` 块。如果应用在哈希方案中使用了最终块的转换，则可能需要确保在释放所有流之前先对其进行读取。

类别

密码

受影响的 API

- System.Security.Cryptography.CryptoStream.Dispose

Entity Framework Core

Entity Framework Core 重大变更

全球化

- "C"区域设置映射到固定区域设置

"C"区域设置映射到固定区域设置

.NET Core 2.2 及更早版本依赖于默认 ICU 行为，该行为将"C"区域设置映射到 en_US_POSIX 区域设置。En_US_POSIX 区域设置的排序行为并不可取，因为它不支持不区分大小写的字符串比较。用户会遇到意外行为是因为部分 Linux 分发版将"C"区域设置设置为了默认区域设置。

更改描述

从 .NET Core 3.0 开始，"C"区域设置映射已更改为使用固定区域设置，而不是 en_US_POSIX。"C"区域设置到固定的映射还应用于 Windows，以实现一致性。

将"C"映射到 en_US_POSIX 区域性会导致客户混乱，因为 en_US_POSIX 不支持不区分大小写的字符串排序/搜索操作。由于"C"区域设置用作部分 Linux 分发版中的默认区域设置，因此客户在这些操作系统上会遭遇此类不良行为。

引入的版本

3.0

建议操作

除了解此变更外，没有什么特别之处需要注意。此变更仅影响使用"C"本地映射的应用程序。

类别

全球化

受影响的 API

所有排序规则和区域性 API 都会受到此变更的影响。

MSBuild

- 资源清单文件名更改

资源清单文件名更改

从 .NET Core 3.0 开始，默认情况下，MSBuild 会为资源文件生成不同的清单文件名。

引入的版本

3.0

更改描述

在 .NET Core 3.0 之前，如果没有为项目文件中的 `EmbeddedResource` 项指定 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据，则 MSBuild 会在 `<RootNamespace>.<ResourceFilePathFromProjectRoot>.resources` 模式中生成清单文件名。如果未在项目文件中定义 `RootNamespace`，则其默认为项目名称。例如，根项目目录中名为"Form1.resx"的资源文件的生成清单名称是"MyProject.Form1.resources"。

从 .NET Core 3.0 开始，如果资源文件与同名的源文件（例如 Form1.resx 和 Form1.cs）并置，则 MSBuild 将使用源文件中的类型信息在 `<Namespace>.<ClassName>.resources` 模式中生成清单文件名。命名空间和类名称是从并置源文件的第一个类型中提取的。例如，与名为"Form1.cs"的源文件并置的、名为"Form1.resx"的资源文件的生成清单名称是"MyNamespace.Form1.resources"。需要注意的一点是，文件名的第一部分不同于早期版本的 .NET Core（是 MyNamespace，而不是 MyProject）。

> **NOTE**
> 如果已在项目文件中的 `EmbeddedResource` 项上指定 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据，则此更改不会影响该资源文件。

此重大更改是在 .NET Core 项目中添加 `EmbeddedResourceUseDependentUponConvention` 属性时引入的。默认情况下，不会在 .NET Core 项目文件中显式列出资源文件，因此它们没有 `DependentUpon` 元数据来指定如何命名生成的 .resources 文件。如果 `EmbeddedResourceUseDependentUponConvention` 设置为 `true`（默认值），则 MSBuild 将查找并置的源文件，并从该文件中提取命名空间和类名。如果将 `EmbeddedResourceUseDependentUponConvention` 设置为 `false`，则 MSBuild 将根据之前的行为生成清单名称，将 `RootNamespace` 和相对文件路径组合在一起。

建议操作

在大多数情况下，开发人员不需要执行任何操作，应用应可以继续工作。但是，如果此更改造成应用中断运行，你可以：

- 将代码更改为需要新的清单名称。
- 在项目文件中将 `EmbeddedResourceUseDependentUponConvention` 设置为 `false`，以选择退出新命名约定。

```
<PropertyGroup>
  <EmbeddedResourceUseDependentUponConvention>false</EmbeddedResourceUseDependentUponConvention>
</PropertyGroup>
```

类别

MSBuild

受影响的 API

不可用

网络

- HttpRequestMessage.Version 的默认值已更改为 1.1

HttpRequestMessage.Version 的默认值已更改为 1.1

System.Net.Http.HttpRequestMessage.Version 属性的默认值已从 2.0 更改为 1.1。

引入的版本

3.0

更改描述

在 .NET Core 1.0 至 2.0 中，System.Net.Http.HttpRequestMessage.Version 属性的默认值为 1.1。从 .NET Core 2.1 开始，该值已更改为 2.1。

从 .NET Core 3.0 开始，System.Net.Http.HttpRequestMessage.Version 属性返回的默认版本号再次为 1.1。

建议操作

如果代码依赖于 System.Net.Http.HttpRequestMessage.Version 属性，则返回默认值 2.0，以更新代码。

类别

网络

受影响的 API

- System.Net.Http.HttpRequestMessage.Version

Visual Basic

- Microsoft.VisualBasic.Constants.vbNewLine 已过时

Microsoft.VisualBasic.Constants.vbNewLine 已过时

自 .NET Core 3.0 起，Microsoft.VisualBasic.Constants.vbNewLine 常量标记为[已过时]。

引入的版本

3.0

更改描述

自 .NET Core 3.0 起，已向 Microsoft.VisualBasic.Constants.vbNewLine 常量应用 Obsolete 属性。使用常量会引发编辑器警告。在 .NET Framework 和之前的 .NET Core 版本中，它未被标记为已过时。

此项更改旨在支持将 Visual Basic 用作多平台开发的一种语言。vbNewLine 常量与 Windows 上的换行符序列 `\r\n` 等效。在基于 Unix 的系统上，换行符为 `\n`。

建议操作

vbNewLine 的已过时属性消息包含以下建议：

对于回车符和换行符，请使用 vbCrLf。对于当前平台的新行，请使用 Environment.NewLine。

类别

Visual Basic

受影响的 API

- Microsoft.VisualBasic.Constants.vbNewLine

.NET Core 2.2 的新增功能

2021/11/16 •

.NET Core 2.2 包括在应用程序部署、运行时服务的事件处理、Azure SQL 数据库的身份验证、JIT 编译器性能，以及执行 `Main` 方法之前的代码注入方面的增强功能。

新部署模式

从 .NET Core 2.2 开始，可以部署依赖于框架的可执行文件，这是".exe" 文件而不是".dll" 文件。与依赖框架的部署在功能上类似，依赖框架的可执行文件 (FDE) 仍然依赖于存在的 .NET Core 的共享系统级版本来运行。应用程序只包含代码和任何第三方依赖项。与依赖框架的部署不同，FDE 特定于平台。

这种新的部署模式在构建可执行文件（而不是库）方面具有独特优势，这意味着你可以直接运行应用程序，而无需首先调用 `dotnet`。

核心

在运行时服务中处理事件

你可能经常希望监视应用程序的运行时服务（如 GC、JIT 和 ThreadPool）的使用情况，以了解它们如何影响应用程序。 在 Windows 系统上，这通常通过监视当前进程的 ETW 事件来完成。 虽然这仍然可以很好地工作，但是如果你在低特权环境中或者在 Linux 或 macOS 上运行，那么并不总是能够使用 ETW。

从 .NET Core 2.2 开始，现在可以使用 System.Diagnostics.Tracing.EventListener 类来使用 CoreCLR 事件。这些事件描述了诸如 GC、JIT、ThreadPool 和 interop 等运行时服务的行为。这些事件与作为 CoreCLR ETW 提供程序的一部分公开的事件相同。 这允许应用程序使用这些事件或使用传输机制将它们发送到遥测聚合服务。可以在以下代码示例中看到如何订阅事件：

```
internal sealed class SimpleEventListener : EventListener
{
    // Called whenever an EventSource is created.
    protected override void OnEventSourceCreated(EventSource eventSource)
    {
        // Watch for the .NET runtime EventSource and enable all of its events.
        if (eventSource.Name.Equals("Microsoft-Windows-DotNETRuntime"))
        {
            EnableEvents(eventSource, EventLevel.Verbose, (EventKeywords)(-1));
        }
    }

    // Called whenever an event is written.
    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        // Write the contents of the event to the console.
        Console.WriteLine($"ThreadID = {eventData.OSThreadId} ID = {eventData.EventId} Name =
{eventData.EventName}");
        for (int i = 0; i < eventData.Payload.Count; i++)
        {
            string payloadString = eventData.Payload[i]?.ToString() ?? string.Empty;
            Console.WriteLine($"\tName = \"{eventData.PayloadNames[i]}\" Value = \"{payloadString}\"");
        }
        Console.WriteLine("\n");
    }
}
```

此外，.NET Core 2.2 向 EventWrittenEventArgs 类添加了以下两个属性来提供有关 ETW 事件的其他信息：

- EventWrittenEventArgs.OSThreadId
- EventWrittenEventArgs.TimeStamp

数据

使用 SqlConnection.AccessToken 属性对 Azure SQL 数据库进行 AAD 身份验证

从 .NET Core 2.2 开始，由 Azure Active Directory 颁发的访问令牌可用于对 Azure SQL 数据库进行身份验证。若要支持访问令牌，必须将 AccessToken 属性添加到 SqlConnection 类。若要利用 AAD 身份验证，请下载 System.Data.SqlClient NuGet 包的版本 4.6。若要使用功能，可以使用包含在 `Microsoft.IdentityModel.Clients.ActiveDirectory` NuGet 包中的适用于 .NET 的 Active Directory 身份验证库获取访问令牌值。

JIT 编译器改进

分层编译仍然是一项可选功能

在 .NET Core 2.1，JIT 编译器实现了一项新的编译器技术，即“分层编译”，作为可选功能。分层编译旨在提高性能。由 JIT 编译器执行的重要任务之一是优化代码执行。然而，对于很少使用的代码路径，相比执行未优化代码所花费的运行时，编译器可能需要更多的时间来优化代码。分层编译介绍了 JIT 编译中的两个阶段：

- 第一层，将尽可能快地生成代码。
- 第二层，将为那些频繁执行的方法生成优化代码。为了增强性能，第二层编译并行执行。

有关分层编译可能带来的性能改进的信息，请参阅宣布发布 .NET Core 2.2 预览版 2。

在 .NET Core 2.2 预览版 2 中，默认情况下已启用分层编译。但是，我们仍然没有准备好在默认情况下启用分层编译。因此在 .NET Core 2.2 中，分层编译仍是一项可选功能。有关选择加入分层编译的信息，请参阅 .NET Core 2.1 中的新增功能中的 Jit 编译器改进。

运行时

在执行 Main 方法之前注入代码

从 .NET Core 2.2 开始，可以使用启动挂钩注入代码，然后再运行应用程序的 Main 方法。启动挂钩使主机可以在部署应用程序之后自定义其行为，而不需要重新编译或更改应用程序。

我们希望托管提供商定义自定义配置和策略，包括可能会影响主入口点加载行为的设置，如 System.Runtime.Loader.AssemblyLoadContext 行为。挂钩可用于设置跟踪或遥测注入，以设置回调进行处理，或定义其他环境相关的行为。挂钩独立于入口点，因此不需要修改用户代码。

有关详细信息，请参阅主机启动挂钩。

请参阅

- .NET Core 3.1 的新增功能
- ASP.NET Core 2.2 的新增功能
- EF Core 2.2 中的新增功能

.NET Core 2.1 的新增功能

2021/11/16 •

.NET Core 2.1 提供以下几个方面的增强功能和新功能：

工具

.NET Core 2.1 SDK (v 2.1.300)，该工具与 .NET Core 2.1 一起提供，包括以下更改和增强功能：

生成性能改进

.NET Core 2.1 的主要关注点是改进生成时性能，特别是增量生成。这些性能改进适用于使用 `dotnet build` 的两个命令行生成和 Visual Studio 中的生成。一些个别的改进领域包括：

- 对于包资产解决方法，只解决由生成使用的资产而不是所有资产。
- 程序集引用缓存。
- 使用长时间运行的 SDK 生成服务器，这些是跨各个 `dotnet build` 调用的过程。每次 `dotnet build` 运行时不再需要 JIT 编译大量代码块。生成服务器进程可以使用以下命令自动终止：

  ```
  dotnet buildserver shutdown
  ```

新的 CLI 命令

许多使用 `DotnetCliToolReference` 的仅在每个项目的基础上可用的工具现作为 .NET Core SDK 的一部分提供。这些工具包括：

- `dotnet watch` 提供文件系统观察程序，该程序在执行指定的命令集之前会首先等待文件更改。例如，下面的命令将自动重新生成当前项目，并在其中的文件发生更改时生成详细输出：

  ```
  dotnet watch -- --verbose build
  ```

  请注意 `--verbose` 选项前面的 `--` 选项。它分隔从传递给子 `dotnet` 进程的参数直接传递到 `dotnet watch` 命令的选项。如果没有该选项，`--verbose` 选项将适用于 `dotnet watch` 命令，而非 `dotnet build` 命令。

  有关详细信息，请参阅使用 dotnet watch 开发 ASP.NET Core 应用。

- `dotnet dev-certs` 生成和管理在 ASP.NET Core 应用程序开发期间使用的证书。
- `dotnet user-secrets` 管理 ASP.NET Core 应用程序中用户机密库的机密。
- `dotnet sql-cache` 在 Microsoft SQL Server 数据库中创建表和索引以用于分布式缓存。
- `dotnet ef` 是用于管理 Entity Framework Core 应用程序中数据库、DbContext 对象和迁移的工具。有关

详细信息，请参阅 EF Core .NET 命令行工具。

全局工具

.NET Core 2.1 支持全局工具，即，可通过命令行在全局范围内使用的自定义工具。以前版本的 .NET Core 中的扩展性模型只能通过使用 `DotnetCliToolReference` 在每个项目的基础上提供自定义工具。

若要安装全局工具，请使用 dotnet tool install 命令。例如：

```
dotnet tool install -g dotnetsay
```

完成安装后，可以通过指定工具名称从命令行运行该工具。有关详细信息，请参阅 .NET Core 工具概述。

使用 `dotnet tool` 命令管理工具

在 .NET Core 2.1 SDK 中，所有工具操作都使用 `dotnet tool` 命令。可用选项如下：

- `dotnet tool install` 安装工具。
- `dotnet tool update` 卸载并重新安装工具，它将高效地对其进行更新。
- `dotnet tool list` 列出当前安装的工具。
- `dotnet tool uninstall` 卸载当前安装的工具。

前滚

从 .NET Core 2.0 开始，所有 .NET Core 应用程序都将自动前滚到系统上安装的最新次要版本。

从 .NET Core 2.0 开始，如果在其中构建应用程序的 .NET Core 版本在运行时不存在，应用程序将针对最新安装的次要版本的 .NET Core 自动运行。换而言之，如果应用程序在 .NET Core 2.0 中生成，而主机系统未安装 .NET Core 2.0 但安装了 .NET Core 2.1，则应用程序将通过 .NET Core 2.1 运行。

> **IMPORTANT**
> 此前滚行为不适用于预览版本。默认情况下，它也不适用于主要版本，但可以通过以下设置进行更改。

可以通过在没有候选共享框架的情况下更改前滚设置来修改此行为。可用设置如下：

- `0` - 禁用次要版本前滚行为。使用此设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 2.0.1，但不会前滚到 .NET Core 2.2.0 或 .NET Core 3.0.0。
- `1` - 启用次要版本前滚行为。这是设置的默认值。使用此设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 2.0.1 或 .NET Core 2.2.0，具体取决于安装的版本，但它不会前滚到 .NET Core3.0.0。
- `2` - 启用次要和主要版本前滚行为。即使考虑不同的主要版本，如果这样设置，为 .NET Core 2.0.0 构建的应用程序将前滚到 .NET Core 3.0.0。

可以通过以下三种方式之一修改此设置：

- 将 `DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX` 环境变量设置为所需的值。
- 使用所需的值将下列行添加到 .runtimeconfig.json 文件：

  ```
  "rollForwardOnNoCandidateFx" : 0
  ```

- 使用 .NET Core CLI 时，请使用所需的值将下列选项添加到 .NET Core 命令，例如 `run`：

```
dotnet run --rollForwardOnNoCandidateFx=0
```

修补程序版本前滚与此设置无关，并且在应用任何潜在的次要或主要版本前滚之后完成。

部署

自包含应用程序服务

`dotnet publish` 现发布服务运行时版本的自包含应用程序。当你使用 .NET Core 2.1 SDK (v 2.1.300) 发布自包含应用程序时，你的应用程序将包括此 SDK 已知的最新服务运行时版本。升级到最新的 SDK 时，你将发布最新的 .NET Core 运行时版本。这适用于 .NET Core 1.0 运行时以及更高版本。

自包含发布依赖于 NuGet.org 上的运行时版本。计算机上不需要有服务运行时。

使用 .NET Core 2.0 SDK，自包含应用程序将通过 .NET Core 2.0.0 运行时发布，除非通过 `RuntimeFrameworkVersion` 属性指定不同版本。借助此新行为，你将不再需要设置此属性便可为自包含的应用程序选择更高版本的运行时。最简单的方法是始终通过 .NET Core 2.1 SDK (v 2.1.300) 发布。

有关详细信息，请参阅独立部署运行时前滚。

Windows 兼容包

当你将现有代码从 .NET Framework 转移到 .NET Core 时，可以使用 Windows 兼容包。除了 .NET Core 中提供的 API，它使你还能够访问额外的 20,000 多个 API。这些 API 包括 System.Drawing 命名空间中的类型、EventLog 类、WMI、性能计数器、Windows 服务以及 Windows 注册表类型和成员。

JIT 编译器改进

.NET Core 包含新的 JIT 编译器技术，称为“分层编译”（也称为“自适应优化”），可以显著提高性能。分层编译是一个可选设置。

由 JIT 编译器执行的重要任务之一是优化代码执行。然而，对于很少使用的代码路径，相比运行未优化代码所花费的运行时，编译器可能需要更多的时间来优化代码。分层编译介绍了 JIT 编译中的两个阶段：

- 第一层，将尽可能快地生成代码。
- 第二层，将为那些频繁执行的方法生成优化代码。为了增强性能，第二层编译并行执行。

可以通过这两种方法之一选择加入分层编译。

- 若要在所有使用 .NET Core 2.1 SDK 的项目中使用分层编译，请设置以下环境变量：

  ```
  COMPlus_TieredCompilation="1"
  ```

- 若要在每个项目的基础上使用分层编译，将 `<TieredCompilation>` 属性添加到 MSBuild 项目文件的 `<PropertyGroup>` 部分，如以下示例所示：

  ```
  <PropertyGroup>
      <!-- other property definitions -->

      <TieredCompilation>true</TieredCompilation>
  </PropertyGroup>
  ```

API 更改

`Span<T>` 和 `Memory<T>`

.NET Core 2.1 包括一些新类型，使得在使用数组和其他类型内存方面要高效得多。新类型包括：

- System.Span<T> 和 System.ReadOnlySpan<T>。
- System.Memory<T> 和 System.ReadOnlyMemory<T>。

如果没有这些类型，那么在作为数组的一部分或内存缓冲区的一部分传递此类项时，必须在将数据的某些部分传递给方法之前复制该数据部分。这些类型提供了该数据的虚拟视图，无需额外的内存分配和复制操作。

下面的示例使用 Span<T> 和 Memory<T> 实例来提供一个数组 10 个元素的虚拟视图。

```
using System;

class Program
{
    static void Main()
    {
        int[] numbers = new int[100];
        for (int i = 0; i < 100; i++)
        {
            numbers[i] = i * 2;
        }

        var part = new Span<int>(numbers, start: 10, length: 10);
        foreach (var value in part)
            Console.Write($"{value}  ");
    }
}
// The example displays the following output:
//     20  22  24  26  28  30  32  34  36  38
```

```
Module Program
    Sub Main()
        Dim numbers As Integer() = New Integer(99) {}

        For i As Integer = 0 To 99
            numbers(i) = i * 2
        Next

        Dim part = New Memory(Of Integer)(numbers, start:=10, length:=10)

        For Each value In part.Span
            Console.Write($"{value}  ")
        Next
    End Sub
End Module
' The example displays the following output:
'     20  22  24  26  28  30  32  34  36  38
```

Brotli 压缩

.NET Core 2.1 添加了对 Brotli 压缩和解压缩的支持。Brotli 是在 RFC 7932 中定义的通用无损压缩算法，并且大多数 Web 浏览器和主 Web 服务器都提供支持。可以使用基于流的 System.IO.Compression.BrotliStream 类或基于范围的高性能 System.IO.Compression.BrotliEncoder 和 System.IO.Compression.BrotliDecoder 类。下面的示例用 BrotliStream 类演示压缩：

```
public static Stream DecompressWithBrotli(Stream toDecompress)
{
    MemoryStream decompressedStream = new MemoryStream();
    using (BrotliStream decompressionStream = new BrotliStream(toDecompress, CompressionMode.Decompress))
    {
        decompressionStream.CopyTo(decompressedStream);
    }
    decompressedStream.Position = 0;
    return decompressedStream;
}
```

```
Public Function DecompressWithBrotli(toDecompress As Stream) As Stream
    Dim decompressedStream As New MemoryStream()
    Using decompressionStream As New BrotliStream(toDecompress, CompressionMode.Decompress)
        decompressionStream.CopyTo(decompressedStream)
    End Using
    decompressedStream.Position = 0
    Return decompressedStream
End Function
```

BrotliStream 行为等同于 DeflateStream 和 GZipStream，这样就可以轻松地将调用这些 API 的代码转换为 BrotliStream。

新加密 API 和加密改进

.NET Core 2.1 包括加密 API 的许多增强功能：

- System.Security.Cryptography.Pkcs.SignedCms 在 System.Security.Cryptography.Pkcs 包中提供。其实现与 .NET Framework 中的 SignedCms 类相同。
- X509Certificate.GetCertHash 和 X509Certificate.GetCertHashString 方法的新重载接受一个哈希算法标识符，使调用方能够使用除 SHA-1 以外的算法获得证书指纹值。
- 新的基于 Span<T> 的加密 API 可用于哈希、HMAC、加密随机数生成、非对称签名生成、非对称签名处理和 RSA 加密。
- 通过使用基于 Span<T> 的实现，System.Security.Cryptography.Rfc2898DeriveBytes 的性能提高了大约 15%。
- 新 System.Security.Cryptography.CryptographicOperations 类包括两个新方法：
  - FixedTimeEquals 需要固定时间来返回任意两个长度相同的输入，这使得它适用于加密验证，从而避免提供计时旁道信息。
  - ZeroMemory 是不能进行优化的内存清理例程。
- 静态 RandomNumberGenerator.Fill 方法用随机值填充 Span<T>。
- System.Security.Cryptography.Pkcs.EnvelopedCms 现在在 Linux 和 macOS 上受支持。
- System.Security.Cryptography.ECDiffieHellman 类系列现提供椭圆曲线 Diffie-Hellman (ECDH)。外围应用与 .NET Framework 中相同。
- RSA.Create 返回的实例可以使用 SHA-2 摘要对 OAEP 进行加密或解密，并使用 RSA-PSS 生成或验证签名。

套接字改进

.NET Core 包括一个新类型 System.Net.Http.SocketsHttpHandler 和重写的 System.Net.Http.HttpMessageHandler，两者构成了更高级别网络 API 的基础。例如，System.Net.Http.SocketsHttpHandler 是 HttpClient 实现的基础。在以前版本的 .NET Core 中，更高级别的

API 基于本机网络实现。

.NET Core 2.1 中引入的套接字实现具有很多优点：

- 对照以前的实现，可以看到显著的性能改进。
- 消除平台依赖项，从而简化部署和维护。
- 在所有 .NET Core 平台之间保持行为一致。

SocketsHttpHandler 是 .NET Core 2.1 中的默认实现。但是，你可以通过调用 AppContext.SetSwitch 方法配置应用程序以使用较旧的 HttpClientHandler 类：

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", false);
```

```
AppContext.SetSwitch("System.Net.Http.UseSocketsHttpHandler", False)
```

还可以使用环境变量选择退出使用基于 SocketsHttpHandler 的套接字实现。为此，需要将 `DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPHANDLER` 设置为 `false` 或 0。

在 Windows 上，还可以选择使用 System.Net.Http.WinHttpHandler后者依赖于本机实现），或者通过将类实例传递到 HttpClient 构造函数来使用 SocketsHttpHandler 类。

在 Linux 和 macOS 上，可以在每个进程的基础上仅配置 HttpClient。在 Linux 上，如果想要使用旧的 HttpClient 实现，则需要部署 libcurl。（它随 .NET Core 2.0 一起安装。）

重大更改

有关中断性变更的信息，请参阅从版本 2.0 迁移到 2.1 的中断性变更。

请参阅

- .NET Core 3.1 的新增功能
- EF Core 2.1 中的新增功能
- ASP.NET Core 2.1 的新增功能

.NET Core 2.1 中的中断性变更

2021/11/16 •

若要迁移到 2.1 版 .NET Core，本文中列出的中断性变更可能会影响到你的应用。

Core .NET 库

- 路径 API 对无效字符不引发异常
- 添加到内置结构类型的私有字段
- macOS 上的 OpenSSL 版本

路径 API 对无效字符不引发异常

如果找到无效字符，涉及文件路径的 API 将不再验证路径字符或引发 ArgumentException。

更改说明

在 .NET Framework 和 .NET Core 1.0-2.0 中，如果路径参数包含无效路径字符，受影响的 API 部分中列出的方法便会引发 ArgumentException。从 .NET Core 2.1 开始，如果找到无效字符，这些方法就不再检查无效路径字符或引发异常。

更改原因

主动验证路径字符会对某些跨平台场景形成阻碍。引入此更改是为了使 .NET 不尝试复制或预测操作系统 API 调用的结果。有关详细信息，请参阅博客文章 .Net Core 2.1 中的 System.IO 速览。

引入的版本

.NET Core 2.1

建议的操作

如果代码依赖这些 API 来检查无效字符，则可以添加对 Path.GetInvalidPathChars 的调用。

受影响的 API

- System.IO.Directory.CreateDirectory
- System.IO.Directory.Delete
- System.IO.Directory.EnumerateDirectories
- System.IO.Directory.EnumerateFiles
- System.IO.Directory.EnumerateFileSystemEntries
- System.IO.Directory.GetCreationTime(String)
- System.IO.Directory.GetCreationTimeUtc(String)
- System.IO.Directory.GetDirectories
- System.IO.Directory.GetDirectoryRoot(String)
- System.IO.Directory.GetFiles
- System.IO.Directory.GetFileSystemEntries
- System.IO.Directory.GetLastAccessTime(String)
- System.IO.Directory.GetLastAccessTimeUtc(String)
- System.IO.Directory.GetLastWriteTime(String)
- System.IO.Directory.GetLastWriteTimeUtc(String)
- System.IO.Directory.GetParent(String)
- System.IO.Directory.Move(String, String)
- System.IO.Directory.SetCreationTime(String, DateTime)
- System.IO.Directory.SetCreationTimeUtc(String, DateTime)

- System.IO.Directory.SetCurrentDirectory(String)
- System.IO.Directory.SetLastAccessTime(String, DateTime)
- System.IO.Directory.SetLastAccessTimeUtc(String, DateTime)
- System.IO.Directory.SetLastWriteTime(String, DateTime)
- System.IO.Directory.SetLastWriteTimeUtc(String, DateTime)
- System.IO.DirectoryInfo ctor
- System.IO.Directory.GetDirectories
- System.IO.Directory.GetFiles
- System.IO.DirectoryInfo.GetFileSystemInfos
- System.IO.File.AppendAllText
- System.IO.File.AppendAllTextAsync
- System.IO.File.Copy
- System.IO.File.Create
- System.IO.File.CreateText
- System.IO.File.Decrypt
- System.IO.File.Delete
- System.IO.File.Encrypt
- System.IO.File.GetAttributes(String)
- System.IO.File.GetCreationTime(String)
- System.IO.File.GetCreationTimeUtc(String)
- System.IO.File.GetLastAccessTime(String)
- System.IO.File.GetLastAccessTimeUtc(String)
- System.IO.File.GetLastWriteTime(String)
- System.IO.File.GetLastWriteTimeUtc(String)
- System.IO.File.Move
- System.IO.File.Open
- System.IO.File.OpenRead(String)
- System.IO.File.OpenText(String)
- System.IO.File.OpenWrite(String)
- System.IO.File.ReadAllBytes(String)
- System.IO.File.ReadAllBytesAsync(String, CancellationToken)
- System.IO.File.ReadAllLines(String)
- System.IO.File.ReadAllLinesAsync(String, CancellationToken)
- System.IO.File.ReadAllText(String)
- System.IO.File.ReadAllTextAsync(String, CancellationToken)
- System.IO.File.SetAttributes(String, FileAttributes)
- System.IO.File.SetCreationTime(String, DateTime)
- System.IO.File.SetCreationTimeUtc(String, DateTime)
- System.IO.File.SetLastAccessTime(String, DateTime)
- System.IO.File.SetLastAccessTimeUtc(String, DateTime)
- System.IO.File.SetLastWriteTime(String, DateTime)
- System.IO.File.SetLastWriteTimeUtc(String, DateTime)
- System.IO.File.WriteAllBytes(String, Byte[])
- System.IO.File.WriteAllBytesAsync(String, Byte[], CancellationToken)
- System.IO.File.WriteAllLines
- System.IO.File.WriteAllLinesAsync

- System.IO.File.WriteAllText
- System.IO.FileInfo ctor
- System.IO.FileInfo.CopyTo
- System.IO.FileInfo.MoveTo
- System.IO.FileStream ctor
- System.IO.Path.GetFullPath(String)
- System.IO.Path.IsPathRooted(String)
- System.IO.Path.GetPathRoot(String)
- System.IO.Path.ChangeExtension(String, String)
- System.IO.Path.GetDirectoryName(String)
- System.IO.Path.GetExtension(String)
- System.IO.Path.HasExtension(String)
- System.IO.Path.Combine

另请参阅

- .NET Core 2.1 中的 System.IO 速览

添加到内置结构类型的私有字段

私有字段已添加到引用程序集中的特定结构类型。因此，在 C# 中，必须始终使用 new 运算符或默认文本来实例化结构类型。

更改描述

在 .NET Core 2.0 和早期版本中，某些提供的结构类型（例如 ConsoleKeyInfo）可以在不使用 `new` 运算符或默认文本的情况下在 C# 中实例化。这是因为 C# 编译器使用的引用程序集不包含结构的私有字段。从 .NET Core 2.1 开始，.NET 结构类型的所有私有字段都将添加到引用程序集。

例如，下面的 C# 代码在 .NET Core 2.0 中编译，但不在 .Net core 2.1 中编译：

```
ConsoleKeyInfo key;    // Struct type

if (key.ToString() == "y")
{
    Console.WriteLine("Yes!");
}
```

在 .NET Core 2.1 中，之前的代码会导致以下编译器错误：**CS0165 - 使用了未赋值的局部变量“key”**

引入的版本

2.1

建议操作

使用 `new` 运算符或默认文本实例化结构类型。

例如：

```
ConsoleKeyInfo key = new ConsoleKeyInfo();    // Struct type.

if (key.ToString() == "y")
    Console.WriteLine("Yes!");
```

```
ConsoleKeyInfo key = default;    // Struct type.

if (key.ToString() == "y")
    Console.WriteLine("Yes!");
```

类别

Core .NET 库

受影响的 API

- System.ArraySegment<T>.Enumerator
- System.ArraySegment<T>
- System.Boolean
- System.Buffers.MemoryHandle
- System.Buffers.StandardFormat
- System.Byte
- System.Char
- System.Collections.DictionaryEntry
- System.Collections.Generic.Dictionary<TKey,TValue>.Enumerator
- System.Collections.Generic.Dictionary<TKey,TValue>.KeyCollection.Enumerator
- System.Collections.Generic.Dictionary<TKey,TValue>.ValueCollection.Enumerator
- System.Collections.Generic.HashSet<T>.Enumerator
- System.Collections.Generic.KeyValuePair<TKey,TValue>
- System.Collections.Generic.LinkedList<T>.Enumerator
- System.Collections.Generic.List<T>.Enumerator
- System.Collections.Generic.Queue<T>.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.KeyCollection.Enumerator
- System.Collections.Generic.SortedDictionary<TKey,TValue>.ValueCollection.Enumerator
- System.Collections.Generic.SortedSet<T>.Enumerator
- System.Collections.Generic.Stack<T>.Enumerator
- System.Collections.Immutable.ImmutableArray<T>.Enumerator
- System.Collections.Immutable.ImmutableArray<T>
- System.Collections.Immutable.ImmutableDictionary<TKey,TValue>.Enumerator
- System.Collections.Immutable.ImmutableHashSet<T>.Enumerator
- System.Collections.Immutable.ImmutableList<T>.Enumerator
- System.Collections.Immutable.ImmutableQueue<T>.Enumerator
- System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>.Enumerator
- System.Collections.Immutable.ImmutableSortedSet<T>.Enumerator
- System.Collections.Immutable.ImmutableStack<T>.Enumerator
- System.Collections.Specialized.BitVector32.Section
- System.Collections.Specialized.BitVector32
- LazyMemberInfo
- System.ComponentModel.Design.Serialization.MemberRelationship
- System.ConsoleKeyInfo
- System.Data.SqlTypes.SqlBinary
- System.Data.SqlTypes.SqlBoolean
- System.Data.SqlTypes.SqlByte
- System.Data.SqlTypes.SqlDateTime

- System.Data.SqlTypes.SqlDecimal
- System.Data.SqlTypes.SqlDouble
- System.Data.SqlTypes.SqlGuid
- System.Data.SqlTypes.SqlInt16
- System.Data.SqlTypes.SqlInt32
- System.Data.SqlTypes.SqlInt64
- System.Data.SqlTypes.SqlMoney
- System.Data.SqlTypes.SqlSingle
- System.Data.SqlTypes.SqlString
- System.DateTime
- System.DateTimeOffset
- System.Decimal
- System.Diagnostics.CounterSample
- System.Diagnostics.SymbolStore.SymbolToken
- System.Diagnostics.Tracing.EventSource.EventData
- System.Diagnostics.Tracing.EventSourceOptions
- System.Double
- System.Drawing.CharacterRange
- System.Drawing.Point
- System.Drawing.PointF
- System.Drawing.Rectangle
- System.Drawing.RectangleF
- System.Drawing.Size
- System.Drawing.SizeF
- System.Guid
- System.HashCode
- System.Int16
- System.Int32
- System.Int64
- System.IntPtr
- System.IO.Pipelines.FlushResult
- System.IO.Pipelines.ReadResult
- System.IO.WaitForChangedResult
- System.Memory<T>
- System.ModuleHandle
- System.Net.Security.SslApplicationProtocol
- System.Net.Sockets.IPPacketInformation
- System.Net.Sockets.SocketInformation
- System.Net.Sockets.UdpReceiveResult
- System.Net.WebSockets.ValueWebSocketReceiveResult
- System.Nullable<T>
- System.Numerics.BigInteger
- System.Numerics.Complex
- System.Numerics.Vector<T>
- System.ReadOnlyMemory<T>
- System.ReadOnlySpan<T>.Enumerator

- System.ReadOnlySpan<T>
- System.Reflection.CustomAttributeNamedArgument
- System.Reflection.CustomAttributeTypedArgument
- System.Reflection.Emit.Label
- System.Reflection.Emit.OpCode
- System.Reflection.Metadata.ArrayShape
- System.Reflection.Metadata.AssemblyDefinition
- System.Reflection.Metadata.AssemblyDefinitionHandle
- System.Reflection.Metadata.AssemblyFile
- System.Reflection.Metadata.AssemblyFileHandle
- System.Reflection.Metadata.AssemblyFileHandleCollection.Enumerator
- System.Reflection.Metadata.AssemblyFileHandleCollection
- System.Reflection.Metadata.AssemblyReference
- System.Reflection.Metadata.AssemblyReferenceHandle
- System.Reflection.Metadata.AssemblyReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.AssemblyReferenceHandleCollection
- System.Reflection.Metadata.Blob
- System.Reflection.Metadata.BlobBuilder.Blobs
- System.Reflection.Metadata.BlobContentId
- System.Reflection.Metadata.BlobHandle
- System.Reflection.Metadata.BlobReader
- System.Reflection.Metadata.BlobWriter
- System.Reflection.Metadata.Constant
- System.Reflection.Metadata.ConstantHandle
- System.Reflection.Metadata.CustomAttribute
- System.Reflection.Metadata.CustomAttributeHandle
- System.Reflection.Metadata.CustomAttributeHandleCollection.Enumerator
- System.Reflection.Metadata.CustomAttributeHandleCollection
- System.Reflection.Metadata.CustomAttributeNamedArgument<TType>
- System.Reflection.Metadata.CustomAttributeTypedArgument<TType>
- System.Reflection.Metadata.CustomAttributeValue<TType>
- System.Reflection.Metadata.CustomDebugInformation
- System.Reflection.Metadata.CustomDebugInformationHandle
- System.Reflection.Metadata.CustomDebugInformationHandleCollection.Enumerator
- System.Reflection.Metadata.CustomDebugInformationHandleCollection
- System.Reflection.Metadata.DeclarativeSecurityAttribute
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandle
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection.Enumerator
- System.Reflection.Metadata.DeclarativeSecurityAttributeHandleCollection
- System.Reflection.Metadata.Document
- System.Reflection.Metadata.DocumentHandle
- System.Reflection.Metadata.DocumentHandleCollection.Enumerator
- System.Reflection.Metadata.DocumentHandleCollection
- System.Reflection.Metadata.DocumentNameBlobHandle
- System.Reflection.Metadata.Ecma335.ArrayShapeEncoder
- System.Reflection.Metadata.Ecma335.BlobEncoder

- System.Reflection.Metadata.Ecma335.CustomAttributeArrayTypeEncoder
- System.Reflection.Metadata.Ecma335.CustomAttributeElementTypeEncoder
- System.Reflection.Metadata.Ecma335.CustomAttributeNamedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.CustomModifiersEncoder
- System.Reflection.Metadata.Ecma335.EditAndContinueLogEntry
- System.Reflection.Metadata.Ecma335.ExceptionRegionEncoder
- System.Reflection.Metadata.Ecma335.FixedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.GenericTypeArgumentsEncoder
- System.Reflection.Metadata.Ecma335.InstructionEncoder
- System.Reflection.Metadata.Ecma335.LabelHandle
- System.Reflection.Metadata.Ecma335.LiteralEncoder
- System.Reflection.Metadata.Ecma335.LiteralsEncoder
- System.Reflection.Metadata.Ecma335.LocalVariablesEncoder
- System.Reflection.Metadata.Ecma335.LocalVariableTypeEncoder
- System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder.MethodBody
- System.Reflection.Metadata.Ecma335.MethodBodyStreamEncoder
- System.Reflection.Metadata.Ecma335.MethodSignatureEncoder
- System.Reflection.Metadata.Ecma335.NamedArgumentsEncoder
- System.Reflection.Metadata.Ecma335.NamedArgumentTypeEncoder
- System.Reflection.Metadata.Ecma335.NameEncoder
- System.Reflection.Metadata.Ecma335.ParametersEncoder
- System.Reflection.Metadata.Ecma335.ParameterTypeEncoder
- System.Reflection.Metadata.Ecma335.PermissionSetEncoder
- System.Reflection.Metadata.Ecma335.ReturnTypeEncoder
- System.Reflection.Metadata.Ecma335.ScalarEncoder
- System.Reflection.Metadata.Ecma335.SignatureDecoder<TType,TGenericContext>
- System.Reflection.Metadata.Ecma335.SignatureTypeEncoder
- System.Reflection.Metadata.Ecma335.VectorEncoder
- System.Reflection.Metadata.EntityHandle
- System.Reflection.Metadata.EventAccessors
- System.Reflection.Metadata.EventDefinition
- System.Reflection.Metadata.EventDefinitionHandle
- System.Reflection.Metadata.EventDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.EventDefinitionHandleCollection
- System.Reflection.Metadata.ExceptionRegion
- System.Reflection.Metadata.ExportedType
- System.Reflection.Metadata.ExportedTypeHandle
- System.Reflection.Metadata.ExportedTypeHandleCollection.Enumerator
- System.Reflection.Metadata.ExportedTypeHandleCollection
- System.Reflection.Metadata.FieldDefinition
- System.Reflection.Metadata.FieldDefinitionHandle
- System.Reflection.Metadata.FieldDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.FieldDefinitionHandleCollection
- System.Reflection.Metadata.GenericParameter
- System.Reflection.Metadata.GenericParameterConstraint
- System.Reflection.Metadata.GenericParameterConstraintHandle

- System.Reflection.Metadata.GenericParameterConstraintHandleCollection.Enumerator
- System.Reflection.Metadata.GenericParameterConstraintHandleCollection
- System.Reflection.Metadata.GenericParameterHandle
- System.Reflection.Metadata.GenericParameterHandleCollection.Enumerator
- System.Reflection.Metadata.GenericParameterHandleCollection
- System.Reflection.Metadata.GuidHandle
- System.Reflection.Metadata.Handle
- System.Reflection.Metadata.ImportDefinition
- System.Reflection.Metadata.ImportDefinitionCollection.Enumerator
- System.Reflection.Metadata.ImportDefinitionCollection
- System.Reflection.Metadata.ImportScope
- System.Reflection.Metadata.ImportScopeCollection.Enumerator
- System.Reflection.Metadata.ImportScopeCollection
- System.Reflection.Metadata.ImportScopeHandle
- System.Reflection.Metadata.InterfaceImplementation
- System.Reflection.Metadata.InterfaceImplementationHandle
- System.Reflection.Metadata.InterfaceImplementationHandleCollection.Enumerator
- System.Reflection.Metadata.InterfaceImplementationHandleCollection
- System.Reflection.Metadata.LocalConstant
- System.Reflection.Metadata.LocalConstantHandle
- System.Reflection.Metadata.LocalConstantHandleCollection.Enumerator
- System.Reflection.Metadata.LocalConstantHandleCollection
- System.Reflection.Metadata.LocalScope
- System.Reflection.Metadata.LocalScopeHandle
- System.Reflection.Metadata.LocalScopeHandleCollection.ChildrenEnumerator
- System.Reflection.Metadata.LocalScopeHandleCollection.Enumerator
- System.Reflection.Metadata.LocalScopeHandleCollection
- System.Reflection.Metadata.LocalVariable
- System.Reflection.Metadata.LocalVariableHandle
- System.Reflection.Metadata.LocalVariableHandleCollection.Enumerator
- System.Reflection.Metadata.LocalVariableHandleCollection
- System.Reflection.Metadata.ManifestResource
- System.Reflection.Metadata.ManifestResourceHandle
- System.Reflection.Metadata.ManifestResourceHandleCollection.Enumerator
- System.Reflection.Metadata.ManifestResourceHandleCollection
- System.Reflection.Metadata.MemberReference
- System.Reflection.Metadata.MemberReferenceHandle
- System.Reflection.Metadata.MemberReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.MemberReferenceHandleCollection
- System.Reflection.Metadata.MetadataStringComparer
- System.Reflection.Metadata.MethodDebugInformation
- System.Reflection.Metadata.MethodDebugInformationHandle
- System.Reflection.Metadata.MethodDebugInformationHandleCollection.Enumerator
- System.Reflection.Metadata.MethodDebugInformationHandleCollection
- System.Reflection.Metadata.MethodDefinition
- System.Reflection.Metadata.MethodDefinitionHandle

- System.Reflection.Metadata.MethodDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.MethodDefinitionHandleCollection
- System.Reflection.Metadata.MethodImplementation
- System.Reflection.Metadata.MethodImplementationHandle
- System.Reflection.Metadata.MethodImplementationHandleCollection.Enumerator
- System.Reflection.Metadata.MethodImplementationHandleCollection
- System.Reflection.Metadata.MethodImport
- System.Reflection.Metadata.MethodSignature<TType>
- System.Reflection.Metadata.MethodSpecification
- System.Reflection.Metadata.MethodSpecificationHandle
- System.Reflection.Metadata.ModuleDefinition
- System.Reflection.Metadata.ModuleDefinitionHandle
- System.Reflection.Metadata.ModuleReference
- System.Reflection.Metadata.ModuleReferenceHandle
- System.Reflection.Metadata.NamespaceDefinition
- System.Reflection.Metadata.NamespaceDefinitionHandle
- System.Reflection.Metadata.Parameter
- System.Reflection.Metadata.ParameterHandle
- System.Reflection.Metadata.ParameterHandleCollection.Enumerator
- System.Reflection.Metadata.ParameterHandleCollection
- System.Reflection.Metadata.PropertyAccessors
- System.Reflection.Metadata.PropertyDefinition
- System.Reflection.Metadata.PropertyDefinitionHandle
- System.Reflection.Metadata.PropertyDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.PropertyDefinitionHandleCollection
- System.Reflection.Metadata.ReservedBlob<THandle>
- System.Reflection.Metadata.SequencePoint
- System.Reflection.Metadata.SequencePointCollection.Enumerator
- System.Reflection.Metadata.SequencePointCollection
- System.Reflection.Metadata.SignatureHeader
- System.Reflection.Metadata.StandaloneSignature
- System.Reflection.Metadata.StandaloneSignatureHandle
- System.Reflection.Metadata.StringHandle
- System.Reflection.Metadata.TypeDefinition
- System.Reflection.Metadata.TypeDefinitionHandle
- System.Reflection.Metadata.TypeDefinitionHandleCollection.Enumerator
- System.Reflection.Metadata.TypeDefinitionHandleCollection
- System.Reflection.Metadata.TypeLayout
- System.Reflection.Metadata.TypeReference
- System.Reflection.Metadata.TypeReferenceHandle
- System.Reflection.Metadata.TypeReferenceHandleCollection.Enumerator
- System.Reflection.Metadata.TypeReferenceHandleCollection
- System.Reflection.Metadata.TypeSpecification
- System.Reflection.Metadata.TypeSpecificationHandle
- System.Reflection.Metadata.UserStringHandle
- System.Reflection.ParameterModifier

- System.Reflection.PortableExecutable.CodeViewDebugDirectoryData
- System.Reflection.PortableExecutable.DebugDirectoryEntry
- System.Reflection.PortableExecutable.PEMemoryBlock
- System.Reflection.PortableExecutable.SectionHeader
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncVoidMethodBuilder
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>.ConfiguredTaskAwaiter
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable<TResult>
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable.ConfiguredTaskAwaiter
- System.Runtime.CompilerServices.ConfiguredTaskAwaitable
- System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>
- System.Runtime.CompilerServices.ConfiguredValueTaskAwaitable<TResult>.ConfiguredValueTaskAwaiter
- System.Runtime.CompilerServices.TaskAwaiter<TResult>
- System.Runtime.CompilerServices.TaskAwaiter
- System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
- System.Runtime.CompilerServices.ValueTaskAwaiter<TResult>
- System.Runtime.InteropServices.ArrayWithOffset
- System.Runtime.InteropServices.GCHandle
- System.Runtime.InteropServices.HandleRef
- System.Runtime.InteropServices.OSPlatform
- System.Runtime.InteropServices.WindowsRuntime.EventRegistrationToken
- System.Runtime.Serialization.SerializationEntry
- System.Runtime.Serialization.StreamingContext
- System.RuntimeArgumentHandle
- System.RuntimeFieldHandle
- System.RuntimeMethodHandle
- System.RuntimeTypeHandle
- System.SByte
- System.Security.Cryptography.CngProperty
- System.Security.Cryptography.ECCurve
- System.Security.Cryptography.HashAlgorithmName
- System.Security.Cryptography.X509Certificates.X509ChainStatus
- System.Security.Cryptography.Xml.X509IssuerSerial
- System.ServiceProcess.SessionChangeDescription
- System.Single
- System.Span<T>.Enumerator
- System.Span<T>
- System.Threading.AsyncFlowControl
- System.Threading.AsyncLocalValueChangedArgs<T>
- System.Threading.CancellationToken
- System.Threading.CancellationTokenRegistration
- System.Threading.LockCookie
- System.Threading.SpinLock

- System.Threading.SpinWait
- System.Threading.Tasks.Dataflow.DataflowMessageHeader
- System.Threading.Tasks.ParallelLoopResult
- System.Threading.Tasks.ValueTask<TResult>
- System.TimeSpan
- System.TimeZoneInfo.TransitionTime
- System.Transactions.TransactionOptions
- System.TypedReference
- System.TypedReference
- System.UInt16
- System.UInt32
- System.UInt64
- System.UIntPtr
- System.Windows.Forms.ColorDialog.Color
- System.Windows.Media.Animation.KeyTime
- System.Windows.Media.Animation.RepeatBehavior
- System.Xml.Serialization.XmlDeserializationEvents
- Windows.Foundation.Point
- Windows.Foundation.Rect
- Windows.Foundation.Size
- Windows.UI.Color
- Windows.UI.Xaml.Controls.Primitives.GeneratorPosition
- Windows.UI.Xaml.CornerRadius
- Windows.UI.Xaml.Duration
- Windows.UI.Xaml.GridLength
- Windows.UI.Xaml.Media.Matrix
- Windows.UI.Xaml.Media.Media3D.Matrix3D
- Windows.UI.Xaml.Thickness

---

macOS 上的 OpenSSL 版本

对于 AesCcm、AesGcm、DSAOpenSsl、ECDiffieHellmanOpenSsl、ECDsaOpenSsl、RSAOpenSsl 和 SafeEvpPKeyHandle 类型，macOS 上的 .NET Core 3.0 及更高版本的运行时现在首选 OpenSSL 1.1.x 版而非 OpenSSL 1.0.x 版。

.NET Core 2.1 运行时现在支持 OpenSSL 1.1.x 版本，但仍首选 OpenSSL 1.0.x 版。

更改描述

以前，.NET Core 运行时在 macOS 上使用 OpenSSL 1.0.x 版处理与 OpenSSL 交互的类型。最新的 OpenSSL 1.0.x 版 OpenSSL 1.0.2 现已不受支持。若要在支持的 OpenSSL 版本上保留使用 OpenSSL 的类型，.NET Core 3.0 及更高版本的运行时现需在 macOS 上使用较新版本的 OpenSSL。

通过此更改，macOS 上的 .NET Core 运行时的行为如下所示：

- .NET Core 3.0 及更高版本运行时使用 OpenSSL 1.1.x（如果可用），并且仅在没有 1.1.x 版本可用的情况下才回退到 OpenSSL 1.0.x。

  对于将 OpenSSL 互操作类型与自定义 P/Invoke 一起使用的调用方，请按照 SafeEvpPKeyHandle.OpenSslVersion 注释中的指南进行操作。如果不检查 OpenSslVersion 值，你的应用可能会出现故障。

- .NET Core 2.1 运行时使用 OpenSSL 1.0.x（如果可用），并且仅在没有 1.0.x 版本可用的情况下才回退到

OpenSSL 1.1.x。

2.1 运行时首选早期版本的 OpenSSL，因为 .NET Core 2.1 中不存在 SafeEvpPKeyHandle.OpenSslVersion 属性，因此无法在运行时可靠地确定 OpenSSL 版本。

引入的版本

- .NET Core 2.1.16
- .NET Core 3.0.3
- .NET Core 3.1.2

建议操作

- 如果不再需要 OpenSSL 版本 1.0.2，请将其卸载。
- 如果你使用 AesCcm、AesGcm、DSAOpenSsl、ECDiffieHellmanOpenSsl、ECDsaOpenSsl、RSAOpenSsl 或 SafeEvpPKeyHandle 类型，请安装 OpenSSL 1.1.x。
- 如果你将 OpenSSL 互操作类型与自定义 P/Invoke 一起使用，请按照 SafeEvpPKeyHandle.OpenSslVersion 注释中的指南进行操作。

类别

Core .NET 库

受影响的 API

- System.Security.Cryptography.AesCcm
- System.Security.Cryptography.AesGcm
- System.Security.Cryptography.DSAOpenSsl
- System.Security.Cryptography.ECDiffieHellmanOpenSsl
- System.Security.Cryptography.ECDsaOpenSsl
- System.Security.Cryptography.RSAOpenSsl
- System.Security.Cryptography.SafeEvpPKeyHandle

MSBuild

- SDK 现已包含项目工具

SDK 现已包含项目工具

.NET Core 2.1 SDK 现包括常见 CLI 工具，你无需再从项目中引用这些工具。

更改描述

在 .NET Core 2.0 中，项目通过 `<DotNetCliToolReference>` 项目设置引用外部 .NET 工具。在 .NET Core 2.1 中，其中的某些工具包含在 .NET Core SDK 中，不再需要设置。如果在项目中包含对这些工具的引用，则会收到如下所示的错误：“Microsoft.EntityFrameworkCore.Tools.DotNet”工具现包含在 .NET Core SDK 中。

.NET Core 2.1 SDK 中现在包含的工具：

<DOTNETCLITOOLREFERENCE> [	[[
`Microsoft.DotNet.Watcher.Tools`	`dotnet-watch`
`Microsoft.Extensions.SecretManager.Tools`	dotnet-user-secrets
`Microsoft.Extensions.Caching.SqlConfig.Tools`	dotnet-sql-cache
`Microsoft.EntityFrameworkCore.Tools.DotNet`	dotnet-ef

引入的版本

.NET Core SDK 2.1.300

建议的操作

从项目中删除 `<DotNetCliToolReference>` 设置。

类别

MSBuild

受影响的 API

不可用

.NET Core 2.0 的新增功能

2021/11/16 •

.NET Core 2.0 提供以下几个方面的增强功能和新功能：

- 工具
- 语言支持
- 平台改进
- API 更改
- Visual Studio 集成
- 文档改进

工具

dotnet restore 隐式运行

在旧版 .NET Core 中，在使用 dotnet new 命令新建项目后，以及每当向项目添加新的依赖项时，都必须立即运行 dotnet restore 命令，以便下载依赖项。

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

还可以将 `--no-restore` 开关传递到 `new`、`run`、`build`、`publish`、`pack` 和 `test` 命令，从而禁用自动调用 `dotnet restore`。

重定目标到 .NET Core 2.0

如果已安装 .NET Core 2.0 SDK，那么定目标到 .NET Core 1.x 的项目可以重定目标到 .NET Core 2.0。

若要重定目标到 .NET Core 2.0，请将 `<TargetFramework>` 元素（或 `<TargetFrameworks>` 元素，如果项目文件中有多个目标的话）值从 1.x 更改为 2.0，从而编辑项目文件：

```
<PropertyGroup>
    <TargetFramework>netcoreapp2.0</TargetFramework>
 </PropertyGroup>
```

还可以用同样的方式，将 .NET Standard 库重定目标到 .NET Standard 2.0：

```
<PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
 </PropertyGroup>
```

若要详细了解如何将项目迁移到 .NET Core 2.0，请参阅从 ASP.NET Core 1.x 迁移到 ASP.NET Core 2.0。

语言支持

除了支持 C# 和 F# 外，.NET Core 2.0 还支持 Visual Basic。

Visual Basic

在版本 2.0 发布后，.NET Core 现在支持 Visual Basic 2017。可使用 Visual Basic 创建以下类型项目：

- .NET Core 控制台应用程序
- .NET Core 类库
- .NET Standard 类库
- .NET Core 单元测试项目
- .NET Core xUnit 测试项目

例如，若要创建 Visual Basic"Hello World"应用程序，请通过命令行按照以下步骤操作：

1. 打开控制台窗口，创建项目目录，并将它设为当前目录。
2. 输入命令 `dotnet new console -lang vb`。

   此命令创建文件扩展名为 `.vbproj` 的项目文件，以及名为 Program.vb 的 Visual Basic 源代码文件。此文件包含用于将字符串"Hello World!"写入控制台窗口的源代码 。
3. 输入命令 `dotnet run`。.NET Core CLI 自动编译并执行应用程序，在控制台窗口中 显示文本字符串"Hello World!"。

支持 C# 7.1

.NET Core 2.0 支持 C# 7.1，其中新增了大量功能，具体包括：

- 可以使用 async 关键字标记 `Main` 方法（应用程序入口点）。
- 推断的元组名称。
- 默认表达式。

平台改进

.NET Core 2.0 包括许多功能，可便于用户更轻松地在支持的操作系统上安装并使用 .NET Core。

.NET Core for Linux 是一个实现代码

.NET Core 2.0 提供一个 Linux 实现代码，适用于多个 Linux 发行版本。.NET Core 1.x 要求下载发行版本专属的 Linux 实现代码。

还可以开发定目标到 Linux 一个操作系统的应用程序。.NET Core 1.x 要求分别定目标到每个 Linux 发行版本。

支持 Apple 加密库

macOS 上的 .NET Core 1.x 要求使用 OpenSSL 工具包的加密库。.NET Core 2.0 使用 Apple 加密库，不要求使用 OpenSSL，因此不再需要安装它。

API 更改和库支持

支持 .NET Standard 2.0

.NET Standard 定义了一组版本化 API，这些 API 必须可用于符合相应 Standard 版本要求的 .NET 实现代码。.NET Standard 面向库开发者。旨在保证功能对每个 .NET 实现代码中以 .NET Standard 版本为目标的库可用。.NET Core 1.x 支持 .NET Standard 版本 1.6；.NET Core 2.0 支持最新版 .NET Core 2.0。有关详细信息，请参阅 .NET Standard。

.NET Standard 2.0 比 .NET Standard 1.6 多包含 20,000 多个 API。此扩展的外围应用的大部分来自于，将 .NET Framework 和 Xamarin 的通用 API 合并到 .NET Standard。

.NET Standard 2.0 类库还可以引用 .NET Framework 类库，但前提是它们调用 .NET Standard 2.0 中的 API。不需要重新编译 .NET Framework 库。

有关自上一版本 (.NET Standard 1.6) 起添加到 .NET Standard 的 API 列表，请参阅 .NET Standard 2.0 与 1.6。

扩展的外围应用

与 .NET Core 1.1 相比，.NET Core 2.0 中的可用 API 总数增加了一倍以上。

借助 Windows 兼容包，从 .NET Framework 移植也简单了很多。

支持 .NET Framework 库

.NET Core 代码可以引用现有的 .NET Framework 库，包括现有的 NuGet 包。请注意，库必须使用 .NET Standard 中的 API。

Visual Studio 集成

Visual Studio 2017 版本 15.3 和（在某些情况下）Visual Studio for Mac 提供了大量面向 .NET Core 开发者的重大增强功能。

重定目标 .NET Core 应用程序和 .NET Standard 库

如果已安装 .NET Core 2.0 SDK，可以将 .NET Core 1.x 项目重定目标到 .NET Core 2.0，并将 .NET Standard 1.x 库重定目标到 .NET Standard 2.0。

若要在 Visual Studio 中重定目标项目，可以打开项目属性对话框的“应用程序”选项卡，再将“目标框架”值更改为“.NET Core 2.0”或“.NET Standard 2.0”。还可以通过右键单击项目并选择“编辑 *.csproj 文件”选项进行更改。有关详细信息，请参阅本主题前面的工具部分。

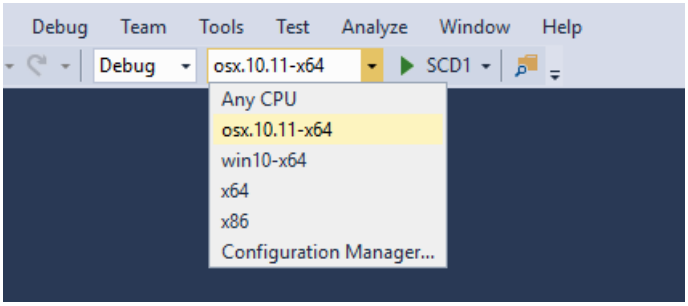
.NET Core 的 Live Unit Testing 支持

修改代码时，Live Unit Testing 在后台自动运行任何受影响的单元测试，并在 Visual Studio 环境中实时显示结果和代码覆盖率。.NET Core 2.0 现在支持 Live Unit Testing。以前，Live Unit Testing 仅适用于 .NET Framework 应用程序。

有关详细信息，请参阅使用 Visual Studio 执行 Live Unit Testing 和 Live Unit Testing 常见问题解答。

更好地支持多个目标框架

若要为多个目标框架生成项目，现在可以从顶级菜单中选择目标平台。在下图中，名为 SCD1 的项目将目标定为 64 位 macOS X 10.11 (`osx.10.11-x64`) 和 64 位 Windows 10/Windows Server 2016 (`win10-x64`)。可以先选择目标框架，再选择项目按钮（在此示例中是为了要运行调试版本）。



对 .NET Core SDK 的并行支持

现在可以安装 .NET Core SDK，与 Visual Studio 互不影响。这样，单版本 Visual Studio 可以生成定目标到不同 .NET Core 版本的项目。以前，Visual Studio 和 .NET Core SDK 紧密结合在一起；特定版本的 SDK 附带了特定版本的 Visual Studio。

文档改进

.NET 应用程序体系结构

通过 .NET 应用程序体系结构，可以查看一系列电子图书，其中提供了有关如何使用 .NET 生成内容的指导、最佳做法和示例应用程序：

- 微服务和 Docker 容器
- 使用 ASP.NET 的 Web 应用程序
- 使用 Xamarin 的移动应用
- 使用 Azure 部署到云的应用程序

请参阅

- ASP.NET Core 2.0 的新增功能

.NET Standard 中的新增功能

2021/11/16 •

.NET Standard 是一种正式规范，它定义了一组版本化 API。这些 API 必须可用于符合相应 Standard 版本要求的 .NET 实现。.NET Standard 面向库开发者。定目标到 .NET Standard 版本的库可用于任意 .NET Framework、.NET Core 或支持 Standard 版本的 Xamarin 实现。

.NET Core SDK 及选择 .NET Core 工作负载时的 Visual Studio 包含 .NET Standard。

支持的 .NET 实现

以下 .NET 实现支持 .NET Standard 2.0：

- .NET Core 2.0 或更高版本
- .NET Framework 4.6.1 或更高版本
- Mono 5.4 或更高版本
- Xamarin.iOS 10.14 或更高版本
- Xamarin.Mac 3.8 或更高版本
- Xamarin.Android 8.0 或更高版本
- 通用 Windows 平台 10.0.16299 或更高版本

.NET Standard 2.0 中的新增功能

.NET Standard 2.0 新增了以下功能：

大幅扩展了 API 集

.NET Standard 版本 1.6 中包含了相对较小的一部分 API。不包含的 API 许多都是 .NET Framework 或 Xamarin 中的常用 API。这样一来，开发变得更为棘手，因为开发人员必须在开发定目标到多个 .NET 实现的应用和库时，寻找常用 API 的合适替代项。为了消除此限制，.NET Standard 2.0 向 Standard 旧版本 .NET Standard 1.6 中的可用 API 补充了 20,000 多个 API。有关添加到 .NET Standard 2.0 的 API 列表，请参阅 .NET Standard 2.0 与 1.6。

.NET Standard 2.0 的 System 命名空间中新增的一些功能包括：

- 支持 AppDomain 类。
- 更好地支持通过 Array 类中的附加成员处理数组。
- 更好地支持通过 Attribute 类中的附加成员处理属性。
- 改进了日历支持，并附加了 DateTime 值的格式设置选项。
- 附加了 Decimal 舍入功能。
- 在 Environment 类中附加了功能。
- 增强了通过 GC 类控制垃圾回收器。
- 增强了 String 类中的字符串比较、枚举和规范化支持。
- TimeZoneInfo.AdjustmentRule 和 TimeZoneInfo.TransitionTime 类支持夏令时调整和时间转换。
- 显著改进了 Type 类中的功能。
- 通过添加包含 SerializationInfo 和 StreamingContext 参数的异常构造函数，改进了对异常对象反序列化的支持。

支持 .NET Framework 库

许多库定目标到 .NET Framework，而不是 .NET Standard。不过，这些库大多调用的是 .NET Standard 2.0 中的 API。自 .NET Standard 2.0 起，可以使用兼容性填充码从 .NET Standard 库访问 .NET Framework 库。此兼容性层

对开发人员透明;无需执行任何操作，即可使用 .NET Framework 库。

只有一项要求就是，.NET Framework 类库调用的 API 必须是 .NET Standard 2.0 中的 API。

支持 Visual Basic

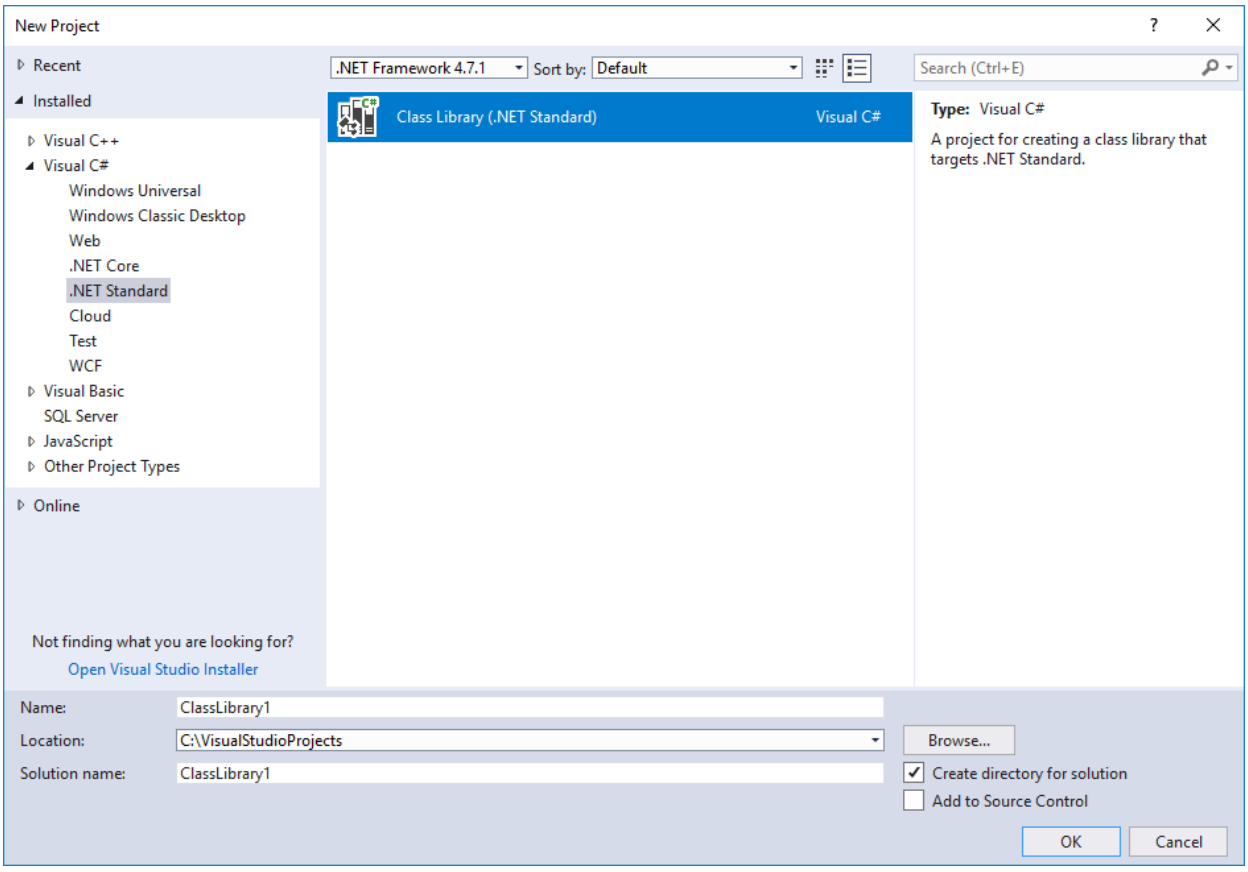
现在可以使用 Visual Basic 开发 .NET Standard 库。安装了 .NET Core 工作负载的 Visual Studio 2019 和 Visual Studio 2017 版本 15.3 或更高版本包含 .NET Standard 类库模板。对于使用其他开发工具和环境的 Visual Basic 开发人员，可以使用 dotnet new 命令创建 .NET Standard 库项目。有关详细信息，请参阅 .NET Standard 库的工具支持。

.NET Standard 库的工具支持

随着 .NET Core 2.0 和 .NET Standard 2.0 发布，Visual Studio 2017 和 .NET Core CLI 均包含创建 .NET Standard 库所需的工具支持。

如果安装含 .NET Core 跨平台开发工作负荷 的 Visual Studio，可以使用项目模板创建 .NET Standard 2.0 库项目，如下图所示：

- C#
- Visual Basic



如果使用的是 .NET Core CLI，可以运行下面的 dotnet new 命令，以创建以 .NET Standard 2.0 为目标的类库项目：

```
dotnet new classlib
```

请参阅

- .NET Standard
- .NET Standard 简介
- 下载 .NET SDK

.NET 中的工具和诊断

2021/11/16 •

在本文中，你将了解可供 .NET 开发人员使用的各种工具。使用 .NET，将获得可靠的软件开发工具包 (SDK)，其中包含命令行接口 (CLI)。.NET CLI 支持 .NET 集成开发环境 (IDE) 中的许多功能。本文还提供用于提高生产率的资源，例如用于诊断性能问题、内存泄漏、高 CPU、死锁的 .NET CLI 工具，以及用于代码分析的工具支持。

.NET SDK

.NET SDK 同时包括 .NET 运行时和 .NET CLI。可下载适用于 Windows、Linux、macOS 或 Docker 的 .NET SDK。有关详细信息，请参阅 .NET SDK 概述。

.NET CLI

.NET CLI 是用于开发、生成、运行和发布 .NET 应用程序的跨平台工具链。.NET CLI 附带了 .NET SDK。有关详细信息，请参阅 .NET CLI 概述。

IDE

可在 Visual Studio Code、Visual Studio或 Visual Studio for Mac 中写入 .NET 应用程序。

其他工具

除了更常用的工具外，.NET 还提供适用于特定方案的工具。部分用例，包括卸载 .NET SDK 或 .NET 运行时、检索 Windows Communication Foundation (WCF) 元数据、生成代理源代码以及序列化 XML。有关详细信息，请参阅 .NET 其他工具概述。

诊断和检测

作为 .NET 开发人员，你可使用常见的性能诊断工具来监视应用性能、使用跟踪分析应用、收集性能指标以及分析转储文件。可使用事件计数器收集性能指标，并使用分析工具深入了解应用的执行方式。有关详细信息，请参阅 .NET 诊断工具。

代码分析

.NET Compiler Platform (Roslyn) 分析器会检查 C# 或 Visual Basic 代码的代码质量和代码样式问题。有关详细信息，请参阅 .NET 源代码分析概述。

程序包验证

使用 .NET SDK，库开发人员验证其包是否一致且格式良好。有关详细信息，请参阅 .NET SDK 包验证。

.NET SDK 概述

2021/11/16 •

.NET SDK 是一组库和工具，开发人员可用其创建 .NET 应用程序和库。它包含以下用于构建和运行应用程序的组件：

- .NET CLI。
- .NET 库和运行时。
- `dotnet` 驱动程序。

获取 .NET SDK

与任何工具一样，首先应将工具安装到计算机上。根据场景，可以使用以下某个方法安装 SDK：

- 使用本机安装程序。
- 使用安装 shell 脚本。

本机安装程序主要用于开发人员的计算机。SDK 通过每个受支持平台的本机安装机制进行分发，例如 Ubuntu 上的 DEB 包或 Windows 上的 MSI 程序包。这些安装程序将根据需要为用户安装并设置环境，以便在安装完成后可立即使用 SDK。但是，这些安装程序也需要对计算机的管理权限。可以在 .NET 下载页面上找到要安装的 SDK。

另一方面，安装脚本不需要使用管理权限。但是，它们也不会在计算机上安装任何系统必备组件；需要手动安装所有系统必备组件。这些脚本主要用于设置生成服务器或希望安装工具但没有管理权限的情况（请务必注意上述系统必备组件注意事项）。可以在安装脚本引用一文中找到详细信息。如果对如何在 CI 构建服务器上设置 SDK 感兴趣，请参阅在持续集成 (CI) 中使用 .NET SDK 和工具一文。

默认情况下，SDK 以“并排”(SxS) 方式安装，这意味着多个版本可以随时在一台计算机上共存。选择要使用的 .NET 版本一文中详细说明了如何在运行 CLI 命令时选择版本。

另请参阅

- .NET CLI 概述
- .NET 版本控制概述
- 如何删除 .NET 运行时和 SDK
- 选择要使用的 .NET 版本

.NET SDK、.NET CLI 和 .NET 运行时使用的环境变量

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

使用以下环境变量配置 .NET SDK、.NET CLI 和 .NET 运行时。

`DOTNET_ROOT`, `DOTNET_ROOT(x86)`

指定 .NET 运行时的位置（如果运行时未安装在默认位置）。Windows 上的默认位置为 `C:\Program Files\dotnet`。Linux 和 macOS 上的默认位置为 `/usr/share/dotnet`。此环境变量仅在通过生成的可执行文件 (apphosts) 运行应用时使用。在 64 位 OS 上运行 32 位可执行文件时，改用 `DOTNET_ROOT(x86)`。

`NUGET_PACKAGES`

全局包文件夹。如果未设置，则默认为 Unix 上的 `~/.nuget/packages` 或 Windows 上的 `%userprofile%\.nuget\packages`。

`DOTNET_SERVICING`

指定加载运行时期间共享主机要使用的服务索引的位置。

`DOTNET_NOLOGO`

指定是否在首次运行时显示 .NET 欢迎消息和遥测消息。设置为 `true` 可将这些消息静音（接受 `true`、`1` 或 `yes` 值），或者，设置为 `false` 可允许显示消息（接受 `false`、`0` 或 `no` 值）。如果未设置，则默认值为 `false`，表示在首次运行时将显示消息。此标志对遥测不起作用（请参阅 `DOTNET_CLI_TELEMETRY_OPTOUT` 中关于如何选择不发送遥测数据的信息）。

`DOTNET_CLI_TELEMETRY_OPTOUT`

指定是否收集并向 Microsoft 发送 .NET 工具使用情况的相关数据。设置为 `true` 以选择退出遥测功能（接受的值为 `true`、`1` 或 `yes`）。否则，设置为 `false` 以选择加入遥测功能（接受的值为 `false`、`0` 或 `no`）。如果未设置，则默认为 `false` 且遥测功能为活动状态。

`DOTNET_SKIP_FIRST_TIME_EXPERIENCE`

如果 `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` 设置为 `true`，`NuGetFallbackFolder` 将不会扩展到磁盘，并将显示一条较短的欢迎消息和遥测通知。

`DOTNET_MULTILEVEL_LOOKUP`

指定是否从全局位置解析 .NET 运行时、共享框架或 SDK。如果未设置，则默认为 1（逻辑 `true`）。设置为 0（逻辑 `false`），不从全局位置解析，并且具有独立的 .NET 安装。有关多级别查找的详细信息，请参阅 Multi-level SharedFX Lookup（多级别 SharedFX 查找）。

DOTNET_ROLL_FORWARD

确定前滚行为。有关详细信息，请参阅本文章前面介绍的 `--roll-forward` 选项。自 .NET Core 3.x 起可用。

DOTNET_ROLL_FORWARD_TO_PRERELEASE

如果设置为 `1`（已启用），则允许从发布版本前滚到预发行版本。默认情况下（`0` - 禁用），请求 .NET 运行时的发行版本时，前滚将仅考虑已安装的发行版本。自 .NET Core 3.x 起可用。

有关详细信息，请参阅前滚。

DOTNET_ROLL_FORWARD_ON_NO_CANDIDATE_FX

如果设置为 `0`，则禁用次要版本前滚。有关详细信息，请参阅前滚。

此设置在 .NET Core 3.0 中被 `DOTNET_ROLL_FORWARD` 取代。应改为使用新设置。

DOTNET_CLI_UI_LANGUAGE

使用区域设置值（如 `en-us`）设置 CLI UI 的语言。支持的值与 Visual Studio 中的值相同。有关详细信息，请参阅 Visual Studio 安装文档中有关更改安装程序语言一节。.NET 资源管理器规则适用，因此你无需选取精确匹配项 — 你还可以在 `CultureInfo` 树中选取后代。例如，如果将其设置为 `fr-CA`，CLI 将查找并使用 `fr` 翻译。如果你将其设置为不受支持的语言，CLI 会退回到英语。

DOTNET_DISABLE_GUI_ERRORS

对于启用了 GUI 的已生成可执行文件 - 禁用对话框弹出窗口，此窗口通常显示某些类错误。在这些情况下，它仅写入到 `stderr` 并退出。

DOTNET_ADDITIONAL_DEPS

等效于 CLI 选项 `--additional-deps`。

DOTNET_RUNTIME_ID

替代检测到的 RID。

DOTNET_SHARED_STORE

程序集解析在某些情况下将回退到的“共享存储”的位置。

DOTNET_STARTUP_HOOKS

要从中加载和执行启动挂钩的程序集列表。

DOTNET_BUNDLE_EXTRACT_BASE_DIR

指定在执行单文件应用程序之前将其提取到的目录。自 .NET Core 3.x 起可用。

有关详细信息，请参阅单文件可执行文件。

COREHOST_TRACE

控制来自托管组件(例如 `dotnet.exe`、`hostfxr` 和 `hostpolicy`)的诊断跟踪。

- `COREHOST_TRACE=[0/1]` -默认值为 `0` - 禁用跟踪。如果设置为 `1`，则启用诊断跟踪。
- `COREHOST_TRACEFILE=<file path>` - 仅当通过 `COREHOST_TRACE=1` 启用跟踪时才会生效。设置后，跟踪信息将写入指定的文件，否则会将跟踪信息写入 `stderr`。自 .NET Core 3.x 起可用。
- `COREHOST_TRACE_VERBOSITY=[1/2/3/4]` - 默认值为 `4`。此设置仅在通过 `COREHOST_TRACE=1` 启用跟踪时使用。自 .NET Core 3.x 起可用。
  - `4` - 写入所有跟踪信息
  - `3` - 仅写入信息性、警告和错误消息
  - `2` - 仅写入警告和错误消息
  - `1` - 仅写入错误消息

获取有关应用程序启动的详细跟踪信息的典型方法是设置 `COREHOST_TRACE=1` 和 `COREHOST_TRACEFILE=host_trace.txt`，然后运行该应用程序。将在当前目录中创建一个新文件 `host_trace.txt`，其中包含详细信息。

`DOTNET_CLI_WORKLOAD_UPDATE_NOTIFY_DISABLE`

禁用工作负载的播发清单后台下载。默认值为 `false` - 未禁用。如果设置为 `true`，则禁用下载。有关详细信息，请参阅播发清单。

`DOTNET_CLI_WORKLOAD_UPDATE_NOTIFY_INTERVAL_HOURS`

指定工作负载的播发清单后台下载之间间隔的最短小时数。默认值为 `24` - 每天不超过一次。有关详细信息，请参阅播发清单。

`SuppressNETCoreSdkPreviewMessage`

如果设置为 `true`，则调用 `dotnet` 将不会在使用预览版 SDK 时发出警告。

另请参阅

- dotnet 命令
- 运行时配置文件
- .NET 运行时配置设置

dotnet-install 脚本引用

2021/11/16 •

“属性”

`dotnet-install.ps1` | `dotnet-install.sh` - 用于安装 .NET SDK 和共享运行时的脚本。

摘要

Windows：

```
dotnet-install.ps1 [-Architecture <ARCHITECTURE>] [-AzureFeed]
    [-Channel <CHANNEL>] [-DryRun] [-FeedCredential]
    [-InstallDir <DIRECTORY>] [-JSonFile <JSONFILE>]
    [-NoCdn] [-NoPath] [-ProxyAddress] [-ProxyBypassList <LIST_OF_URLS>]
    [-ProxyUseDefaultCredentials] [-Quality <QUALITY>] [-Runtime <RUNTIME>]
    [-SkipNonVersionedFiles] [-UncachedFeed] [-Verbose]
    [-Version <VERSION>]

Get-Help ./dotnet-install.ps1
```

Linux/macOS：

```
dotnet-install.sh  [--architecture <ARCHITECTURE>] [--azure-feed]
    [--channel <CHANNEL>] [--dry-run] [--feed-credential]
    [--install-dir <DIRECTORY>] [--jsonfile <JSONFILE>]
    [--no-cdn] [--no-path] [--quality <QUALITY>]
    [--runtime <RUNTIME>] [--runtime-id <RID>]
    [--skip-non-versioned-files] [--uncached-feed] [--verbose]
    [--version <VERSION>]

dotnet-install.sh --help
```

bash 脚本也读取 PowerShell 开关。因此，可以在 Linux/macOS 系统上将 PowerShell 开关与脚本结合使用。

描述

`dotnet-install` 脚本执行 .NET SDK 的非管理员安装，其中包含 .NET CLI 和共享运行时。有两个脚本：

- 在 Windows 上运行的 PowerShell 脚本。
- 在 Linux/macOS 上运行的 bash 脚本。

> **NOTE**
> .NET 会收集遥测数据。若要了解详细信息和退出方式，请查看 .NET SDK 遥测。

目标

脚本的预期用途是用于持续集成 (CI) 方案，其中：

- 需要在无需用户交互和管理员权限的情况下安装 SDK。
- 无需在多个 CI 运行中保留 SDK 安装。

典型的事件序列：

- 触发 CI。
- CI 使用其中一个脚本安装 SDK。
- CI 完成其工作并清除临时数据（包括 SDK 安装）。

要设置开发环境或运行应用，请使用安装程序而不是这些脚本。

建议的版本

建议使用脚本的稳定版本：

- Bash (Linux/macOS)：https://dot.net/v1/dotnet-install.sh
- PowerShell (Windows)：https://dot.net/v1/dotnet-install.ps1

脚本行为

这两个脚本的行为相同。它们从 CLI 生成放置下载 ZIP/tarball 文件，并将其安装在默认位置或 `-InstallDir|--install-dir` 所指定的位置。

默认情况下，安装脚本下载 SDK 并安装它。如果只想获取共享的运行时，请指定 `-Runtime|--runtime` 参数。

默认情况下，该脚本会将安装位置添加到当前会话的 $PATH。通过指定 `-NoPath|--no-path` 参数覆盖此默认行为。脚本未设置 `DOTNET_ROOT` 环境变量。

运行脚本前，请安装所需的依赖项。

可以使用 `-Version|--version` 参数安装特定版本。必须将版本指定为由 3 部分构成的版本号，例如 `2.1.0`。如果未指定版本，则脚本将安装 `latest` 版本。

安装脚本不会更新 Windows 上的注册表。它们只是下载压缩的二进制文件并将其复制到文件夹。如果要更新注册表项值，请使用 .NET 安装程序。

选项

- `-Architecture|--architecture <ARCHITECTURE>`

  要安装的 .NET 二进制文件的体系结构。可能值为 `<auto>`、`amd64`、`x64`、`x86`、`arm64` 以及 `arm`。默认值为 `<auto>`，它表示当前正在运行的操作系统体系结构。

- `-AzureFeed|--azure-feed`

  指定此安装程序的 Azure 源的 URL。建议不要更改该值。默认值为 `https://dotnetcli.azureedge.net/dotnet`。

- `-Channel|--channel <CHANNEL>`

  指定安装的源通道。可能的值为：

  - `Current` - 最新版本。
  - `LTS` - 长期支持频道（最新受支持版本）。
  - 表示特定版本的由两部分构成的 A.B 格式版本（例如 `2.1` 或 `3.0`）。
  - 表示特定 SDK 版本的由三部分构成的 A.B.Cxx 格式版本（例如 5.0.1xx 或 5.0.2xx）。自 5.0 版本起可用。

  使用 `latest` 以外的任何版本时，`version` 参数将替代 `channel` 参数。

  默认值为 `LTS`。有关 .NET 支持频道的详细信息，请参阅 .NET 支持策略页。

- `-DryRun|--dry-run`

  如果设置，脚本将不会执行安装。而会显示要使用哪个命令行来持续安装当前请求的 .NET CLI 版本。例

如，如果指定版本 `latest`，它将显示特定版本的链接，以便可在生成脚本中明确地使用此命令。如果想要自行安装或下载，它还会显示二进制文件位置。

- `-FeedCredential|--feed-credential`

  用作追加到 Azure 源的查询字符串。这允许更改 URL 以使用非公共 blob 存储帐户。

- `--help`

  打印脚本帮助。仅适用于 bash 脚本。对于 PowerShell，请使用 `Get-Help ./dotnet-install.ps1`。

- `-InstallDir|--install-dir <DIRECTORY>`

  指定安装路径。如果不存在，则会创建该目录。默认值为“%LocalAppData%\Microsoft\dotnet”（在 Windows 上）和“$HOME/.dotnet”（在 Linux/macOS 上）。会将二进制文件直接放入目录中。

- `-JSonFile|--jsonfile <JSONFILE>`

  指定将用于确定 SDK 版本的 global.json 文件的路径。global.json 文件必须具有 `sdk:version` 的值。

- `-NoCdn|--no-cdn`

  禁止从 Azure 内容分发网络 (CDN) 进行下载，并直接使用未缓存源。

- `-NoPath|--no-path`

  如果设定，不会将安装文件夹导出到当前会话的路径。默认情况下，该脚本会修改 PATH，这会使 .NET CLI 在安装后立即可用。

- `-ProxyAddress`

  如果设置，安装程序发出 Web 请求时将使用该代理。（仅适用于 Windows。）

- `-ProxyBypassList <LIST_OF_URLS>`

  如果设置了 `ProxyAddress`，它会提供一个以逗号分隔的 URL 列表，表中的 URL 将绕过代理。（仅适用于 Windows。）

- `ProxyUseDefaultCredentials`

  如果设置，在使用代理地址时，安装程序会使用当前用户的凭据。（仅适用于 Windows。）

- `-Quality|--quality <QUALITY>`

  在通道中下载指定质量的最新版本。可能的值为：`daily`、`signed`、`validated`、`preview`、`GA`。只能与 `channel` 结合使用。不适用于当前通道和 LTS 通道；如果使用了其中一个通道，则将被忽略。

  对于 SDK 安装，请使用 `A.B` 或 `A.B.Cxx` 格式的 `channel`。对于运行时安装，请使用 `A.B` 格式的 `channel`。

  使用 `latest` 以外的任何 `version` 时，`version` 参数将替代 `channel` 和 `quality` 参数。

  自 5.0 版本起可用。

- `-Runtime|--runtime <RUNTIME>`

  仅安装共享运行时，而非整个 SDK。可能的值为：

  - `dotnet` - `Microsoft.NETCore.App` 共享运行时。
  - `aspnetcore` - `Microsoft.AspNetCore.App` 共享运行时。
  - `windowsdesktop` - `Microsoft.WindowsDesktop.App` 共享运行时。

- `--runtime-id <RID>` **[已弃用]**

指定要为其安装工具的运行时标识符。使用适用于可移植 Linux 的 `linux-x64`。（仅适用于 Linux/macOS 和低于 .NET Core 2.1 的版本。）

`--os <OPERATING_SYSTEM>`

指定要为其安装工具的操作系统。可能的值包括：`osx`、`linux`、`linux-musl`、`freebsd`、`rhel.6`。（适用于 .NET Core 2.1 及更高版本。）

此参数是可选的，只应在需要替代脚本检测到的操作系统时使用。

- `-SharedRuntime|--shared-runtime`

  > **NOTE**
  > 此参数已过时，可能会在将来版本的脚本中删除。建议的替代项为 `-Runtime|--runtime` 选项。

  仅安装共享运行时位，而非整个 SDK。此选项等效于指定 `-Runtime|--runtime dotnet`。

- `-SkipNonVersionedFiles|--skip-non-versioned-files`

  跳过安装未添加版本的文件，例如 dotnet.exe（如果它们已经存在）。

- `-UncachedFeed|--uncached-feed`

  允许更改此安装程序使用的未缓存源的 URL。建议不要更改该值。

- `-Verbose|--verbose`

  显示诊断信息。

- `-Version|--version <VERSION>`

  表示特定的内部版本。可能的值为：

  - `latest` - 频道上的最新内部版本（与 `-Channel` 选项结合使用）。
  - 由三部分组成的版本，采用 X.Y.Z 格式，表示特定的内部版本；取代 `-Channel` 选项。例如：`2.0.0-preview2-006120`。

  如果没有指定，`-Version` 默认值为 `latest`。

示例

- 将最新的长期支持 (LTS) 版本安装到默认位置：

  Windows：

  ```
  ./dotnet-install.ps1 -Channel LTS
  ```

  macOS/Linux：

  ```
  ./dotnet-install.sh --channel LTS
  ```

- 将 6.0.1xx SDK 的最新预览版本安装到指定位置：

  Windows：

  ```
  ./dotnet-install.ps1 -Channel 6.0.1xx -Quality preview -InstallDir C:\cli
  ```

macOS/Linux：

```
./dotnet-install.sh --channel 6.0.1xx --quality preview --install-dir ~/cli
```

- 安装 3.0.0 版共享运行时：

  Windows：

  ```
  ./dotnet-install.ps1 -Runtime dotnet -Version 3.0.0
  ```

  macOS/Linux：

  ```
  ./dotnet-install.sh --runtime dotnet --version 3.0.0
  ```

- 获取脚本并在公司代理后面安装 2.1.2 版本（仅限 Windows）：

  ```
  Invoke-WebRequest 'https://dot.net/v1/dotnet-install.ps1' -Proxy $env:HTTP_PROXY -
  ProxyUseDefaultCredentials -OutFile 'dotnet-install.ps1';
  ./dotnet-install.ps1 -InstallDir '~/.dotnet' -Version '2.1.2' -ProxyAddress $env:HTTP_PROXY -
  ProxyUseDefaultCredentials;
  ```

- 获取脚本并安装 .NET CLI 单行式命令示例：

  Windows：

  ```
  # Run a separate PowerShell process because the script calls exit, so it will end the current
  PowerShell session.
  &powershell -NoProfile -ExecutionPolicy unrestricted -Command "
  [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; &
  ([scriptblock]::Create((Invoke-WebRequest -UseBasicParsing 'https://dot.net/v1/dotnet-install.ps1')))
  <additional install-script args>"
  ```

  macOS/Linux：

  ```
  curl -sSL https://dot.net/v1/dotnet-install.sh | bash /dev/stdin <additional install-script args>
  ```

请参阅

- .NET 版本
- .NET 运行时和 SDK 下载存档

global.json 概述

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

通过 global.json 文件，可定义在运行 .NET CLI 命令时使用的 .NET SDK 版本。选择 .NET SDK 与指定项目所面向的运行时无关。.NET SDK 版本指示使用哪个版本的 .NET CLI。

通常会使用最新版 SDK 工具，因此不需要 global.json 文件。在某些高级方案中，你可能需要控制 SDK 工具的版本，本文对如何完成此操作进行了介绍。

有关指定运行时的详细信息，请参阅目标框架。

.NET SDK 在当前工作目录(不必与项目目录相同)或其某个父目录中查找 global.json 文件。

global.json 架构

SDK

类型：`object`

指定要选择的 .NET SDK 的相关信息。

version

- 类型：`string`
- 自 .NET Core 1.0 SDK 起可用。

要使用的 .NET SDK 版本。

此字段：

- 不支持通配符，也就是说，必须指定完整版本号。
- 不支持版本范围。

allowPrerelease

- 类型：`boolean`
- 自 .NET Core 3.0 SDK 起可用。

指示在选择要使用的 SDK 版本时，SDK 解析程序是否应考虑预发布版本。

如果未显式设置此值，则默认值将取决于是否从 Visual Studio 运行：

- 如果未使用 Visual Studio，则默认值为 `true`。
- 如果使用 Visual Studio，它将使用请求的预发布状态。也就是说，如果使用 Visual Studio 的预览版本，或者设置了“使用 .NET Core SDK 的预览版”选项(在“工具” > “选项” > “环境” > “预览功能”下方)，则默认值为 `true`，否则为 `false`。

rollForward

- 类型：`string`
- 自 .NET Core 3.0 SDK 起可用。

选择 SDK 版本时要使用的前滚策略，可作为特定 SDK 版本缺失时的回退，或者作为使用更高版本的指令。必须使用 `rollForward` 值指定版本，除非将其设置为 `latestMajor`。默认的前滚行为由匹配规则确定。

要了解可用策略及其行为，请考虑以下格式为 `x.y.znn` 的 SDK 版本定义：

- `x` 是主版本。
- `y` 是次版本。
- `z` 是功能区段。
- `nn` 是修补程序版本。

下表列出了 `rollForward` 键的可能值：

“[”	[[
`patch`	使用指定的版本。 如果找不到，则前滚到最新的修补程序级别。 如果找不到，则失败。 此值是早期 SDK 版本中的旧行为。
`feature`	对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到，则前滚到同一主/次版本中的下一个较高功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则失败。
`minor`	对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到，则前滚到同一主/次版本中的下一个较高功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则前滚到同一主版本中的下一个较高次版本和功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则失败。
`major`	对指定的主版本、次版本和功能区段使用最新的修补程序级别。 如果找不到，则前滚到同一主/次版本中的下一个较高功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则前滚到同一主版本中的下一个较高次版本和功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则前滚到下一个较高主版本、次版本和功能区段，并使用该功能区段的最新修补程序级别。 如果找不到，则失败。
`latestPatch`	使用安装的最新修补程序级别，以与请求的主版本、次版本和功能区段匹配，并且该修补程序级别大于或等于指定的值。 如果找不到，则失败。
`latestFeature`	使用安装的最高功能区段和修补程序级别，以与请求的主版本和次版本匹配，并且该功能区段和修补级别大于或等于指定的值。 如果找不到，则失败。
`latestMinor`	使用安装的最高次版本、功能区段和修补程序级别，以与请求的主版本匹配，并且该次版本、功能区段和修补级别大于或等于指定的值。 如果找不到，则失败。
`latestMajor`	使用安装的最高版本的 .NET SDK，并且该版本大于或等于指定的值。 如果找不到，则失败。

"["	[[
`disable`	不前滚。需要完全匹配。

msbuild-sdks

类型：`object`

可便于在一个位置(而不是在各个项目中)控制项目 SDK 版本。有关详细信息，请参阅如何解析项目 SDK。

示例

下面的示例演示如何不使用预发布版本：

```
{
  "sdk": {
    "allowPrerelease": false
  }
}
```

下面的示例展示了如何使用安装的最高版本，此版本大于或等于指定的版本。所示的 JSON 不允许任何低于 2.2.200 的 SDK 版本，而允许 2.2.200 或任何更高版本(包括 3.0.xxx 和 3.1.xxx)。

```
{
  "sdk": {
    "version": "2.2.200",
    "rollForward": "latestMajor"
  }
}
```

下面的示例演示如何使用完全指定的版本：

```
{
  "sdk": {
    "version": "3.1.100",
    "rollForward": "disable"
  }
}
```

下面的示例展示了如何使用特定主要版本和次要版本的已安装最新功能区段和修补程序版本。所示的 JSON 不允许任何低于 3.1.102 的 SDK 版本，而允许 3.1.102 或任何更高的 3.1.xxx 版本(如 3.1.103 或 3.1.200)。

```
{
  "sdk": {
    "version": "3.1.102",
    "rollForward": "latestFeature"
  }
}
```

下面的示例展示了如何使用特定版本的已安装最高修补程序版本。所示的 JSON 不允许任何低于 3.1.102 的 SDK 版本，而允许 3.1.102 或任何更高的 3.1.1xx 版本(如 3.1.103 或 3.1.199)。

```
{
  "sdk": {
    "version": "3.1.102",
    "rollForward": "latestPatch"
  }
}
```

global.json 和 .NET CLI

最好能知道计算机上安装了哪些 SDK 版本，以便在 global.json 文件中设置相应版本。有关如何执行此操作的详细信息，请参阅如何检查是否已安装 .NET。

若要在计算机上安装其他 .NET SDK 版本，请访问下载 .NET 页面。

可执行 dotnet new 命令，在当前目录中创建一个新的 global.json 文件，如下例所示：

```
dotnet new globaljson --sdk-version 3.0.100
```

匹配规则

> **NOTE**
> 匹配规则由 `dotnet.exe` 入口点控制，该入口点在所有已安装的 .NET 运行时中很常见。如果并行安装了多个运行时，或者正在使用 global.json 文件，则使用已安装的最新版 .NET 运行时的匹配规则。

- .NET Core 3.x
- .NET Core 2.x

从 .NET Core 3.0 开始，在确定要使用的 SDK 版本时，适用以下规则：

- 如果未找到 global.json 文件，或者 global.json 未指定 SDK 版本和 `allowPrerelease` 值，则使用安装的最高 SDK 版本（相当于将 `rollForward` 设置为 `latestMajor`）。是否考虑 SDK 预发布版本取决于 `dotnet` 的调用方式。
  - 如果未使用 Visual Studio，则考虑预发布版本。
  - 如果使用 Visual Studio，它将使用请求的预发布状态。也就是说，如果使用 Visual Studio 的预览版本，或者设置了“使用 .NET Core SDK 的预览版”选项（在“工具” > “选项” > “环境” > “预览功能”下方），则考虑预发布版本；否则仅考虑发布版本 。
- 如果找到了未指定 SDK 版本但指定了 `allowPrerelease` 值的 global.json 文件，则使用安装的最高 SDK 版本（相当于将 `rollForward` 设置为 `latestMajor`）。最新 SDK 版本是发布版本还是预发布版本取决于 `allowPrerelease` 的值。`true` 指示考虑预发布版本；`false` 指示仅考虑发布版本。
- 如果找到 global.json 文件，并且该文件指定了 SDK 版本：
  - 如果未设置 `rollForward` 值，它将使用 `latestPatch` 作为默认 `rollForward` 策略。否则，请在 rollForward 部分中检查每个值及其行为。
  - 有关是否考虑预发布版本以及未设置 `allowPrerelease` 时的默认行为的信息，请参阅 allowPrerelease 部分。

针对生成警告的疑难解答

- 以下警告指示你的项目使用 .NET Core SDK 的预发布版本进行编译：

> 使用的是 .NET Core SDK 的预览版本，可通过当前项目中的 global.json 文件定义 SDK 版本。有关详细信息，请访问 https://go.microsoft.com/fwlink/?linkid=869452。

.NET Core SDK 的各种版本均品质优良稳定，在业内口碑良好。但是，如果不希望使用预发布版本，请在 allowPrerelease 部分中查看可用于 .NET Core 3.0 SDK 或更高版本的各种策略。对于从未安装 .NET Core 3.0 或更高版本运行时或 SDK 的计算机，需要创建一个 global.json 文件，并指定要使用的确切版本。

- 以下警告指示项目面向 EF Core 1.0 或 1.1，后者与 .NET Core 2.1 SDK 及更高版本不兼容：

> 启动项目 '{startupProject}' 面向框架 '.NETCoreApp' 的版本 '{targetFrameworkVersion}'。此版本的 Entity Framework Core .NET 命令行工具仅支持 2.0 或更高版本。有关使用旧版工具的信息，请参阅 https://go.microsoft.com/fwlink/?linkid=871254。

从 .NET Core 2.1 SDK（版本 2.1.300）开始，SDK 中包含 `dotnet ef` 命令。若要编译项目，请在计算机上安装 .NET Core 2.0 SDK（版本 2.1.201）或更早版本，并使用 global.json 文件定义所需 SDK 版本。有关 `dotnet ef` 命令的详细信息，请参阅 EF Core .NET 命令行工具。

请参阅

- 如何解析 SDK 项目

.NET SDK 遥测

2021/11/16 •

.NET SDK 包含遥测功能，可在 .NET CLI 崩溃时收集使用情况数据和异常信息。.NET CLI 附带 .NET SDK，是一组用于生成、测试和发布 .NET 应用的谓词。请务必让 .NET 团队了解到工具使用情况，以便我们对其做出改进。有关故障的信息可帮助团队解决问题并修复 bug。

收集的数据根据 Creative Commons Attribution 许可证以汇总形式发布。

范围

`dotnet` 具有两个功能：运行应用程序和执行 CLI 命令。按以下格式使用 `dotnet` 来启动应用程序时，不会收集遥测数据：

- `dotnet [path-to-app].dll`

使用任何 .NET CLI 命令时，都会收集遥测数据，如：

- `dotnet build`
- `dotnet pack`
- `dotnet run`

如何选择退出

.NET SDK 遥测功能默认处于启用状态。要选择退出遥测功能，请将 `DOTNET_CLI_TELEMETRY_OPTOUT` 环境变量设置为 `1` 或 `true`。

如果安装成功，.NET SDK 安装程序也会发送一个遥测条目。若要选择退出，请在安装 .NET SDK 之前设置 `DOTNET_CLI_TELEMETRY_OPTOUT` 环境变量。

> **IMPORTANT**
> 要在启动安装程序后选择退出，请执行以下操作：关闭安装程序，设置环境变量，然后使用该值集再次运行安装程序。

公开

首次运行其中一个 .NET CLI 命令（如 `dotnet build`）时，.NET SDK 显示以下类似文本。文本可能会因运行的 SDK 版本而略有不同。此"首次运行"体验是 Microsoft 通知用户有关数据收集信息的方式。

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your experience. The data is collected by
Microsoft and shared with the community. You can opt-out of telemetry by setting the
DOTNET_CLI_TELEMETRY_OPTOUT environment variable to '1' or 'true' using your favorite shell.

Read more about .NET CLI Tools telemetry: https://aka.ms/dotnet-cli-telemetry
```

若要禁用此消息和 .NET 欢迎消息，请将 `DOTNET_NOLOGO` 环境变量设置为 `true`。请注意，此变量在遥测选择退出时不起作用。

数据点

遥测功能不收集用户名或电子邮件地址等个人数据。也不会扫描代码，更不会提取项目级敏感数据，如名称、存储库或作者。数据通过 Azure Monitor 技术安全地发送到 Microsoft 服务器，提供对保留数据的受限访问权限，并在严格的安全控制下从安全的 Azure 存储系统发布。

保护你的隐私对我们很重要。如果你怀疑遥测在收集敏感数据，或认为处理数据的方式不安全或不恰当，请在 dotnet/sdk 存储库中记录问题或发送电子邮件至 dotnet@microsoft.com 以供我们展开调查。

遥测功能收集以下数据：

SDK	
全部	调用时间戳。
全部	调用的命令（例如，“build”），从 2.1 开始进行哈希处理。
全部	用于确定地理位置的三个八进制数 IP 地址。
全部	操作系统和版本。
全部	运行 SDK 的运行时 ID (RID)。
全部	.NET SDK 版本。
全部	遥测配置文件：一个可选值，仅在用户显式选择加入时可用，并在 Microsoft 内部使用。
>=2.0	命令参数和选项：收集若干参数和选项（非任意字符串）。请参阅收集的选项。从 2.1.300 后进行哈希处理。
>=2.0	SDK 是否在容器中运行。
>=2.0	目标框架（来自 `TargetFramework` 事件），从 2.1 开始进行哈希处理。
>=2.0	经过哈希处理的媒体访问控制 (MAC) 地址 (SHA256)。
>=2.0	经过哈希处理的当前工作目录。
>=2.0	安装成功报告，包含进行了哈希处理的安装程序 exe 文件名。
>=2.1.300	内核版本。
>=2.1.300	Libc 发行/版本。
>=3.0.100	是否已重定向输出（true 或 false）。
>=3.0.100	CLI/SDK 故障时的异常类型及其堆栈跟踪（发送的堆栈跟踪中仅包含 CLI/SDK 代码）。有关详细信息，请参阅收集的 .NET CLI/SDK 故障异常遥测。
>=5.0.100	用于生成的经过哈希处理的 TargetFrameworkVersion（MSBuild 属性）

SDK 𝌆	𝌆
>=5.0.100	用于生成的经过哈希处理的 RuntimeIdentifier(MSBuild 属性)
>=5.0.100	用于生成的经过哈希处理的 SelfContained(MSBuild 属性)
>=5.0.100	用于生成的经过哈希处理的 UseApphost(MSBuild 属性)
>=5.0.100	用于生成的经过哈希处理的 OutputType(MSBuild 属性)
>=5.0.202	从进程开始到进入 CLI 程序的 main 方法为止的运行时间，可衡量主机和运行时的启动情况。
>=5.0.202	在首次运行时将 .NET 工具添加到路径这一步的运行时间。
>=5.0.202	首次运行时要显示首次使用时间通知的运行时间。
>=5.0.202	首次运行时生成 ASP.NET 证书的运行时间。
>=5.0.202	分析 CLI 输入的运行时间。

收集的选项

某些命令发送其他数据。小部分命令发送第一个参数：

𝌆	𝌆𝌆𝌆𝌆𝌆𝌆𝌆𝌆𝌆
`dotnet help <arg>`	正在查询命令帮助。
`dotnet new <arg>`	模板名称(进行哈希处理)。
`dotnet add <arg>`	单词 `package` 或 `reference`。
`dotnet remove <arg>`	单词 `package` 或 `reference`。
`dotnet list <arg>`	单词 `package` 或 `reference`。
`dotnet sln <arg>`	单词 `add`、`list` 或 `remove`。
`dotnet nuget <arg>`	单词 `delete`、`locals` 或 `push`。
`dotnet workload <subcommand> <arg>`	单词 `install`、`update`、`list`、`search`、`uninstall`、`repair`、`restore` 和工作负荷名称(进行哈希处理)。
`dotnet tool <subcommand> <arg>`	单词 `install`、`update`、`list`、`search`、`uninstall`、`run` 和 dotnet tool 名称(进行哈希处理)。

一小部分命令发送所选项目(如果使用)及其值：

𝌆	𝌆
`--verbosity`	所有命令

“	“
`--language`	`dotnet new`
`--configuration`	`dotnet build`, `dotnet clean`, `dotnet publish`, `dotnet run`, `dotnet test`
`--framework`	`dotnet build`, `dotnet clean`, `dotnet publish`, `dotnet run`, `dotnet test`, `dotnet vstest`
`--runtime`	`dotnet build`, `dotnet publish`
`--platform`	`dotnet vstest`
`--logger`	`dotnet vstest`
`--sdk-package-version`	`dotnet migrate`

除 `--verbosity` 和 `--sdk-package-version` 外，从 .NET Core 2.1.100 SDK 开始，所有其他值都会进行哈希处理。

收集的 .NET CLI/SDK 故障异常遥测

如果 .NET CLI/SDK 崩溃，则会收集 CLI/SDK 代码的异常和堆栈跟踪名称。收集此信息是为了评估问题并改善 .NET SDK 和 CLI 的质量。本文提供了所收集数据的信息。本文还提供了有关生成自己的 .NET SDK 版本的用户如何避免无意泄露个人或敏感信息的提示。

收集的数据类型

.NET CLI 只收集有关 CLI/SDK 异常的信息，不收集应用程序中的异常信息。收集的数据包含异常和堆栈跟踪的名称。此堆栈跟踪为 CLI/SDK 代码。

下面的示例显示所收集的数据类型：

```
System.IO.IOException
at System.ConsolePal.WindowsConsoleStream.Write(Byte[] buffer, Int32 offset, Int32 count)
at System.IO.StreamWriter.Flush(Boolean flushStream, Boolean flushEncoder)
at System.IO.StreamWriter.Write(Char[] buffer)
at System.IO.TextWriter.WriteLine()
at System.IO.TextWriter.SyncTextWriter.WriteLine()
at Microsoft.DotNet.Cli.Utils.Reporter.WriteLine()
at Microsoft.DotNet.Tools.Run.RunCommand.EnsureProjectIsBuilt()
at Microsoft.DotNet.Tools.Run.RunCommand.Execute()
at Microsoft.DotNet.Tools.Run.RunCommand.Run(String[] args)
at Microsoft.DotNet.Cli.Program.ProcessArgs(String[] args, ITelemetry telemetryClient)
at Microsoft.DotNet.Cli.Program.Main(String[] args)
```

避免意外泄露信息

.NET 参与者以及运行自己生成的 .NET SDK 版本的任何其他人都应考虑其 SDK 源代码的路径。如果在使用属于自定义调试生成或者使用自定义生成符号文件配置的 .NET SDK 时出现故障，则生成计算机的 SDK 源文件路径将作为堆栈跟踪的一部分收集，并且不会进行哈希处理。

因此，.NET SDK 的自定义生成不应位于路径名公开个人或敏感信息的目录中。

请参阅

- .NET CLI 遥测数据

- 遥测参考源（dotnet/sdk 存储库）

NETSDK1004：找不到资产文件

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

NuGet 在“obj”文件夹中写入名为 project.assets.json 的文件，.NET SDK 使用该文件来获取有关要传递到编译器的包的信息 。如果在生成过程中找不到资产文件 project.assets.json，则会发生此错误。完整的错误消息类似于以下示例：

NETSDK1004：找不到资产文件“C:\path\to\project.assets.json”。运行 NuGet 包还原以生成此文件。

下面是一些可能导致此错误的原因：

- 你正在从包含 `%` 字符的目录路径运行 `dotnet build` 命令。若要解决此错误，请从文件夹名称中删除 `%`，然后重新运行 `dotnet build`。
- 项目系统不会自动检测和还原对项目文件的更改。若要解决此错误，请打开命令提示符并对项目运行 `dotnet restore`。
- 一个项目由更早版本的 Nuget.exe 单独还原。若要解决此错误，请打开命令提示符并对项目运行 `dotnet restore`。
- 之前的错误，如 NETSDK1045（正在使用的 SDK 版本不支持项目的目标框架），会阻止 NuGet 创建项目资产文件。若要解决 NETSDK1004 错误，请解决以前的错误，然后对该项目运行 `dotnet restore`。
- App Center CI 正在生成一个项目，该项目具有不在 NuGet 中的外部程序集。若要解决此错误，请对程序集使用 NuGet 包。
- 你在 Visual Studio 中添加了一个名称开头是句点的解决方案文件夹。若要解决此错误，请删除文件夹名称中的前导句点。

NETSDK1005 和 NETSDK1047：资产文件缺少目标

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

当 .NET SDK 发出错误 NETSDK1005 或 NETSDK1047 时，项目的资产文件缺失某个目标框架的相关信息。NuGet 在“obj”文件夹中写入名为 project.assets.json 的文件，.NET SDK 使用该文件来获取有关要传递到编译器的包的信息 。在 .NET 5 中，NuGet 添加了名为 `TargetFrameworkAlias` 的新字段，以便早期版本的 MSBuild 或 NuGet 在没有新字段的情况下生成资产文件。有关详细信息，请参阅错误 NETSDK1005。

可以采取下面的一些操作来解决错误：

- 确保使用的是 MSBuild 版本 16.8 或更高版本以及 NuGet 版本 5.8 或更高版本，并在更新工具后还原项目。使用 NuGet 版本 5.8 或更高版本时，应使用 Visual Studio 2019 版本 16.8 或更高版本、MSBuild 版本 16.8 或更高版本以及 .NET 5 SDK 或更高版本。
- 如果在安装版本 16.8 或在更改项目的目标框架后首次在 Visual Studio 2019 中生成项目时出现错误，请再次生成项目。
- 在生成项目之前删除“obj”文件夹。
- 请确保项目的 `TargetFrameworks` 属性中包含缺少的目标值。

NETSDK1013：无法识别 TargetFramework 值

2021/11/16 •

本文适用于：✔ .NET Core 3.1.100 SDK 及更高版本

SDK 尝试将 `<TargetFramework>` 或 `<TargetFrameworks>` 项目文件中提供的值分析为已知值。如果无法识别该值，则 `TargetFrameworkIdentifier` 或 `TargetFrameworkVersion` 值可能会设置为空字符串或 `Unsupported`。

为了解决此问题，请检查支持的框架列表中 `TargetFramework` 值的拼写。也可以直接在项目文件中设置 `TargetFrameworkIdentifier` 和 `TargetFrameworkVersion` 属性。

```
<PropertyGroup Condition="'$(TargetFrameworkIdentifier)' == ''">
  <TargetFrameworkIdentifier>.NETCOREAPP</TargetFrameworkIdentifier>
  <TargetFrameworkVersion>3.1</TargetFrameworkVersion>
</PropertyGroup>
```

NETSDK1022：包含重复的项。

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

从 Visual Studio 2017 / MSBuild 版本 15.3 开始，默认情况下 .NET SDK 会自动包括项目目录中的项。这包括 `Compile` 和 `Content` 目标。这应该能够很好地清理你的项目文件，并降低其中的复杂性。

通过将正确的属性设置为 false，可以还原到以前的行为。

`Compile` 项的示例：

```
<PropertyGroup>
    <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
</PropertyGroup>
```

NETSDK1045：当前的 .NET SDK 不支持将“更新的版本”作为目标。

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

当生成工具找不到生成项目所需的 .NET SDK 版本时，会发生此错误。这通常是由于 .NET SDK 安装或配置问题导致的。完整的错误消息类似于以下示例：

> NETSDK1045：当前的 .NET SDK 不支持将“更新的版本”作为目标。将“更旧的版本”或更低版本作为目标，或者使用支持“更新的版本”的 .NET SDK 版本。

以下部分介绍此错误的一些可能原因。查看每个原因并查看哪一项适用于你。请记住，在对环境或配置文件进行更改时，可能需要重新启动命令窗口、重新启动 Visual Studio 或重新启动计算机，才能使所做的更改生效。

.NET SDK 版本

打开项目文件（.csproj、.vbproj 或 .fsproj），并检查目标框架。这是应用尝试使用的框架版本。

```
<TargetFramework>netcoreapp3.0</TargetFramework>
```

确保计算机上已安装列出的 .NET 版本。可以使用以下命令（打开开发人员命令提示并运行此命令）来列出已安装的版本：

```
dotnet --list-sdks
```

x86 或 x64 体系结构

.NET SDK 的每个版本均可用于 x86 和 x64 体系结构。项目可能会尝试查找适用于错误体系结构的 .NET SDK，或者适用于项目所需体系结构的 .NET SDK 可能未安装。检查所需体系结构的安装文件夹。例如，在 Windows 上，x86 版本的 .NET SDK 安装在 C:\Program Files (x86)\dotnet 中，而 x64 版本安装在 C:\Program Files\dotnet 中。请参阅如何检查是否已安装 .NET 并选择操作系统，查明如何检测计算机上安装的内容。

如果未安装所需的版本，请在 .NET 下载页面找到你需要的版本。

未启用预览版

如果已安装所请求的 .NET SDK 版本的预览版，还需要设置用于在 Visual Studio 中启用预览版的选项。请转到“工具” > “选项” > “环境” > “预览功能”并确保选中了“使用 .NET Core SDK 的预览版”。

Visual Studio 版本

例如，.NET Core 3.0 和更高版本需要 Visual Studio 2019。升级到 Visual Studio 2019 版本 16.3 或更高版本以生成项目。

PATH 环境变量

生成工具使用 PATH 环境变量查找 .NET 生成工具的正确版本。如果 PATH 环境变量包含指向较旧生成工具的直接路径，则可能出现此错误消息。请确保 PATH 环境变量中 .NET 工具的唯一路径为指向顶级 dotnet 文件夹的路

径，例如 C:\Program Files\dotnet。错误 PATH 的示例如下所示：C:\Program Files\dotnet\2.1.0\sdks。

MSBuildSDKPath 环境变量

检查 MSBuildSDKPath 环境变量。这一可选的环境变量由 MSBuild 识别，如果已设置，则会覆盖默认值。它可能会设置为 .NET SDK 的特定版本。如果已设置它，请尝试删除它并重新生成项目。

global.json 文件

在项目的根文件夹中查找并沿着目录链向上一直到该卷的根目录进行查找，看看是否有 global.json 文件，因为它可存在于此文件夹结构中的任何位置。如果它包含 SDK 版本，请删除 `sdk` 节点及其所有子节点，或将其更新为所需的较新 .NET Core 版本。

```
{
  "sdk": {
    "version": "2.1.0"
  }
}
```

global.json 文件不是必需的，因此，如果除 `sdk` 节点以外，该文件未包含任何内容，则可以删除整个文件。

Directory.build.props 文件

Directory.build.props 文件是可选的 MSBuild 文件，它可以设置全局属性。在项目的根文件夹中查找并沿着目录链向上一直到该卷的根目录进行查找，看看是否有这些文件，因为它们可存在于此文件夹结构中的任何位置。查找 `TargetFramework` 元素或 `MSBuildSDKPath` 的设置，这些可能会覆盖所需的设置。

请参阅

- The Current .NET SDK does not support targeting .NET Core 3.0 – Fix（当前的 .NET SDK 不支持将 .NET Core 3.0 作为目标 – 解决）

NETSDK1059：项目包含已过时的 .NET CLI 工具

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

当 .NET SDK 发出警告 NETSDK1059 时，表示项目包含已过时的 .NET CLI 工具。自 .NET Core 2.1 起，这些工具包含在 .NET SDK 中，无需由项目显式引用。有关详细信息，请参阅 SDK 中现已包含的项目工具。

NETSDK1064：找不到包

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

当生成工具找不到生成项目所需的 NuGet 包时，会发生此错误。这通常是由于包还原问题导致的。完整的错误消息类似于以下示例：

> NETSDK1064：找不到版本 x.x.x 的包“PackageName”。自 NuGet 还原以来，它可能已被删除。否则，NuGet 还原可能仅部分完成（可能由于最大路径长度限制所致）。

可以执行以下操作来解决此错误：

- 将 `/restore` 选项添加到 MSBuild.exe 命令。不要使用 `/t:Restore;Build`，因为这可能会导致难以察觉的 bug。一种替代方法是使用 `dotnet build` 命令，因为它会自动执行包还原。
- 如果使用 Visual Studio 2019 或 MSBuild.exe 运行包还原，则该错误可能是由最大路径长度限制所导致的。有关详细信息，请参阅长路径支持 (NuGet CLI) 和 NuGet/Home 问题 #3324。
- 如果使用 x86 nuget.exe 进行还原，并使用 x64 MSBuild.exe 进行生成，则不匹配的位数可能会导致此错误。该生成找不到还原过程声明它所获得的包，因为 project.assets.json 中的路径在不同位数的进程中无法起作用。若要解决此错误，请使用相同位数的工具进行还原和生成，或将 NuGet 配置为将包还原到未在 x86 和 x64 之间进行虚拟化的文件夹。有关详细信息，请参阅 dotnet/core 问题 #4332。
- 如果要生成 Docker 映像，请确保 .dockerginore 文件忽略 bin 和 obj 目录。有关详细信息，请参阅 NETSDK1064：找不到包 DnsClient 1.2.0。

NETSDK1071：对框架中将包含的元包的显式版本化 PackageReference。

2021/11/16 •

本文适用于： ✔ .NET 5.0.100 SDK 及更高版本

当 .NET SDK 发出警告 NETSDK1071 时，它表明将来在 PackageReference 中指定的元包版本与通过 TargetFramework 属性隐式引用的元包版本之间可能会存在版本冲突：

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>
```

由于 TargetFramework 会自动引入元包的版本，因此如果版本不同，它们将发生冲突。

若要解决此问题：

1. 在面向 .NET Core 或 .NET Standard 时，请考虑避免显式引用项目文件中的 `Microsoft.NETCore.App` 或 `NETStandard.Library`。
2. 面向 .NET Core 时，如果需要特定版本的运行时，请使用 `<RuntimeFrameworkVersion>` 属性，而不是直接引用元包。例如，如果使用的是 self-contained deployments 并需要 1.0.0 LTS 运行时的特定修补程序，则可能会发生这种情况。
3. 面向 .NET Standard 时，如果需要特定版本的 `NetStandard.Library`，可以使用 `<NetStandardImplicitPackageVersion>` 属性并将其设置为所需版本。
4. 请勿在 .NET Framework 项目中显式添加或更新对 `Microsoft.NETCore.App` 或 `NETSTandard.Library` 的引用。使用基于 .NET Standard 的 NuGet 包时，NuGet 会自动安装所需的任何版本的 `NETStandard.Library`。
5. 使用 .NET Core 2.1+ 时，请勿指定 `Microsoft.AspNetCore.App` 或 `Microsoft.AspNetCore.All` 版本，因为 .NET SDK 会自动选择相应的版本。（注意：如果该项目还使用 `Microsoft.NET.Sdk.Web`，则此操作仅适用于面向 .NET Core 2.1 的情况。.NET Core 2.2 SDK 中解决了此问题。）
6. 如果希望警告消失，还可以禁用它：

   ```
   <PackageReference Include="Microsoft.NetCore.App" Version="2.2.8" >
     <AllowExplicitVersion>true</AllowExplicitVersion>
   </PackageReference>
   ```

NETSDK1073：未识别 FrameworkReference

2021/11/16 •

本文适用于：✔ .NET Core 2.1.100 SDK 及更高版本

此错误通常表示存在 SDK 无法找到的特定 FrameworkReference 的版本。尝试删除 obj 和 bin 文件夹，并运行 `dotnet restore` 以重新下载最新的目标包 。

或者，由于安装时可能存在问题，因此请确保使用最新版本的 .NET 和 Visual Studio

NETSDK1079：面向 .NET Core 3.0 或更高版本时，不支持 Microsoft.AspNetCore.All 包。

2021/11/16 •

本文适用于：✔ .NET Core SDK 3.1.100 及更高版本

在以下情况下，你可能会收到此错误消息：

- 将 ASP.NET Core 项目从 .NET Core 2.2 或更早版本重定向到 .NET Core 3.0 或更高版本。
- 该项目使用 Microsoft.AspNetCore.All 包。

已删除 Microsoft.AspNetCore.All 的 `PackageReference`。你可能还需要为从 Microsoft.AspNetCore.All 引用但不包含在 ASP.NET Core 共享框架中的包添加包引用。此处列出了这些包：从 Microsoft.AspNetCore.All 迁移到 Microsoft.AspNetCore.App。

另请参阅从 ASP.NET Core 2.2 迁移到 3.0

NETSDK1080：针对 Microsoft.AspNetCore.App 的 PackageReference 不是必需的

2021/11/16 •

NETSDK1080 警告你注意你的项目文件中 `Microsoft.AspNetCore.App` 的 `PackageReference` 元素不是必需的。完整的错误消息类似于以下示例：

> 警告 NETSDK1080：当面向 .NET Core 3.0 或更高版本时，针对 Microsoft.AspNetCore.App 的 PackageReference 不是必需的。如果使用的是 Microsoft.NET.Sdk.Web，则会自动引用共享框架。否则，应将 PackageReference 替换为 FrameworkReference。

通常，从在项目文件中需要 `PackageReference` 条目的更低版本将项目升级到 .NET Core 3.0 或更高版本后，会出现此错误。

ASP.NET Core 项目文件

例如，原始项目文件可能如下例所示：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp2.2</TargetFramework>
    <AspNetCoreHostingModel>InProcess</AspNetCoreHostingModel>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.App"/>
    <PackageReference Include="Microsoft.AspNetCore.Razor.Design" Version="2.2.0" PrivateAssets="All" />
  </ItemGroup>

</Project>
```

更新到 .NET Core 3.1 后，同一项目的项目文件应如下例所示：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

</Project>
```

若要消除此警告，请进行这些更改，尤其是删除 `PackageReference` 元素。有关详细信息，请查看删除过时的包引用。

类库项目

在使用 ASP.NET Core API 的类库项目中，将 `PackageReference` 替换为 `FrameworkReference`，如下例中所示：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <FrameworkReference Include="Microsoft.AspNetCore.App" />
  </ItemGroup>

</Project>
```

有关详细信息，请查看在类库中使用 ASP.NET Core API。

NETSDK1130：无法直接引用 Windows 元数据组件

2021/11/16 •

NETSDK1130 指示你正在尝试直接从以 .NET 5 或更高版本为目标的应用引用 Windows 元数据组件。完整的错误消息类似于以下示例：

> 无法引用 *<Component name>* 。不支持以 .NET 5 或更高版本为目标时，请直接引用 Windows 元数据组件。

解决此问题的方法是：

- 删除对 Microsoft.Windows.SDK.Contracts 包的引用。改为通过项目的 `TargetFramework` 属性指定要访问的 Windows API 版本。例如：。

  ```
  <TargetFramework>net5.0-windows10.0.19041.0</TargetFramework>
  ```

- 使用 C#/WinRT 工具链生成或自定义针对 .NET 5 及更高版本的 WinRT API 和类型。

有关详细信息，请参阅已从 .NET 中删除对 WinRT 的内置支持和在桌面应用中调用 Windows 运行时 API。

NETSDK1141：无法解析 global.json 中指定的 .NET SDK 版本

2021/11/16 •

本文适用于：✔ .NET Core 5.0 SDK 及更高版本

global.json 文件中指定的 SDK 版本有问题。

NETSDK1141：无法解析位于 C:\path\global.json 的 global.json 中指定的 .NET SDK 版本。

原因

- global.json 文件中的 SDK 版本未正确指定。
- 未安装 global.json 文件中指定的 SDK 版本。
- 由于路径不正确，找不到 global.json 中指定的 SDK 版本。

如何修复错误

- 安装 global.json 中请求的 SDK 版本。
- 在 global.json 中指定不同的 SDK 版本。
- 检查 global.json 中的拼写错误或其他问题。有关该文件的正确结构，请参阅 global.json。
- 删除 global.json。在这种情况下，将使用最新安装的 SDK 版本。

当你使用共享项目时，开发人员需要同意将该 SDK 版本用于此项目。在没有 global.json 的情况下，如果使用不同开发计算机的开发人员未使用相同的 SDK 版本，则开发团队中的生成环境可能不一致。若要解决此情况，可以在 global.json 中指定 SDK 版本，并将其签入到源代码管理中作为常用文件，该文件对于所有开发人员来说都是相同的，可确保在所有开发环境中使用相同的 SDK 版本。因此，若要在共享项目中解决此问题，团队可能需要一致同意使用特定 SDK 版本，并更新所有代码以使用此版本。

另请参阅

global.json 如何检查是否安装了 .NET SDK

NETSDK1145：缺少目标包或 apphost 包

2021/11/16 •

本文适用于： ✔ .NET 5.0.100 SDK 及更高版本

当 .NET SDK 出现错误 NETSDK1145 时，将不会安装目标包或 apphost 包，也不支持 NuGet 包还原。这通常是因为 SDK 比 Visual Studio for C++/CLI 项目中包含的 SDK 更新。升级 Visual Studio，删除 global.json（如果它指定了某个 SDK 版本），并卸载较新的 SDK。或者，可以替代目标或 apphost 版本。从错误消息中查找包目录下存在的版本，并匹配项目的目标框架。将以下内容添加到项目中：

对于 apphost 包

```
<ItemGroup>
  <KnownAppHostPack Update="@(KnownAppHostPack)">
    <AppHostPackVersion Condition="'%(TargetFramework)' ==
'TARGETFRAMEWORK'">EXISTINGVERSION</AppHostPackVersion>
  </KnownAppHostPack>
</ItemGroup>
```

对于目标包

```
<ItemGroup>
  <KnownFrameworkReference Update="@(KnownFrameworkReference)">
    <TargetingPackVersion Condition="'%(TargetFramework)' ==
'TARGETFRAMEWORK'">EXISTINGVERSION</TargetingPackVersion>
  </KnownFrameworkReference>
</ItemGroup>
```

NETSDK1147：指定的目标框架缺少工作负载

2021/11/16 •

出现此错误是因为尝试编译的项目需要可选工作负载，但却没有安装此工作负载。完整的错误消息类似于以下示例：

> NETSDK1147：若要生成此项目，必须安装以下工作负载：<workload ID>
>
> 若要安装这些工作负载，请运行以下命令：dotnet workload install <workload ID>

例如，如果你的项目面向 `net6.0-android`，则可能必须运行 `dotnet workload install` 命令并指定工作负载 ID `android`：

```
dotnet workload install android
```

有关详细信息，请参阅 `dotnet workload install`。

NETSDK1149：.NET 5 和更高版本中未提供对 WinRT 的内置支持

2021/11/16 •

NETSDK1149 指示你正在尝试引用的组件需要在面向 .NET 5 或更高版本的应用程序中使用 WinRT。这些 .NET 版本未内置对 WinRT 的支持。完整的错误消息类似于以下示例：

> 无法引用 <Component name>，因为它使用了对 WinRT 的内置支持，而此功能在 .NET 5 和更高版本中不再受支持。需要支持 .NET 5 的组件的更新版本。

如果应用程序调用了 Windows 运行时 API，解决此错误的方法是将应用程序的目标框架名字对象 (TFM) 更改为面向 Windows 10 的值。有关详细信息，请参阅在桌面应用中调用 Windows 运行时 API。

如果你的应用程序调用了第三方 WinRT 组件，请获取支持 .NET 5 的组件的更新版本。可以使用 C#/WinRT 生成更新的版本。

有关详细信息，请参阅已从 .NET 中删除对 WinRT 的内置支持。

NETSDK1174：dotnet run 中 --project 的 -p 缩写已弃用

2021/11/16 •

完整的错误消息类似于以下示例：

> --project 的 -p 缩写已弃用。请使用 --project。

已在 `dotnet run` 中弃用 `-p`，因为 `dotnet run` 与 `dotnet build` 和 `dotnet publish` 有紧密关系。在 `dotnet build` 和 `dotnet publish` 中，`p` 可用于设置 MSBuild 属性。此弃用是对齐这三个命令的缩写的第一步。

有关详细信息，请参阅 `dotnet run` 的 `-p` 选项已弃用。

.NET CLI 概述

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

.NET 命令行接口 (CLI) 工具是用于开发、生成、运行和发布 .NET 应用程序的跨平台工具链。

.NET CLI 附带了 .NET SDK。若要了解如何安装 .NET SDK，请参阅安装 .NET Core。

CLI 命令

默认安装以下命令：

基本命令

- `new`
- `restore`
- `build`
- `publish`
- `run`
- `test`
- `vstest`
- `pack`
- `migrate`
- `clean`
- `sln`
- `help`
- `store`

项目修改命令

- `add package`
- `add reference`
- `remove package`
- `remove reference`
- `list reference`

高级命令

- `nuget delete`
- `nuget locals`
- `nuget push`
- `msbuild`
- `dotnet install script`

工具管理命令

- `tool install`
- `tool list`
- `tool update`
- `tool restore` 自 .NET Core SDK 3.0 起可用。

- `tool run` 自 .NET Core SDK 3.0 起可用。
- `tool uninstall`

工具是控制台应用程序，它们从 NuGet 包中安装并从命令提示符处进行调用。你可自行编写工具，也可安装由第三方编写的工具。工具也称为全局工具、工具路径工具和本地工具。有关详细信息，请参阅 .NET 工具概述。

命令结构

CLI 命令结构包含驱动程序（"dotnet"）和命令，还可能包含命令参数和选项。在大部分 CLI 操作中可看到此模式，例如创建新控制台应用并从命令行运行该应用，因为从名为 *my_app* 的目录中执行时，显示以下命令：

```
dotnet new console
dotnet build --output ./build_output
dotnet ./build_output/my_app.dll
```

驱动程序

驱动程序名为 dotnet，并具有两项职责，即运行依赖于框架的应用或执行命令。

若要运行依赖于框架的应用，请在驱动程序后指定应用，例如，`dotnet /path/to/my_app.dll`。从应用的 DLL 驻留的文件夹执行命令时，只需执行 `dotnet my_app.dll` 即可。如果要使用特定版本的 .NET 运行时，请使用 `--fx-version <VERSION>` 选项（请参阅 dotnet 命令参考）。

为驱动程序提供命令时，`dotnet.exe` 启动 CLI 命令执行过程。例如：

```
dotnet build
```

首先，驱动程序确定要使用的 SDK 版本。如果没有 global.json 文件，则使用可用的最新版本 SDK。这有可能是预览版或稳定版，具体取决于计算机上的最新版本。确定 SDK 版本后，它便会执行命令。

命令

由命令执行操作。例如，`dotnet build` 生成代码。`dotnet publish` 发布代码。使用 `dotnet {command}` 约定将命令作为控制台应用程序实现。

自变量

在命令行上传递的参数是被调用的命令的参数。例如，执行 `dotnet publish my_app.csproj` 时，`my_app.csproj` 参数指示要发布的项目，并被传递到 `publish` 命令。

选项

在命令行上传递的选项是被调用的命令的选项。例如，执行 `dotnet publish --output /build_output` 时，`--output` 选项及其值被传递到 `publish` 命令。

请参阅

- dotnet/sdk GitHub 存储库
- .NET 安装指南

dotnet 命令

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet` - .NET CLI 的通用驱动程序。

摘要

获取有关可用命令和环境的信息：

```
dotnet [--version] [--info] [--list-runtimes] [--list-sdks]

dotnet -h|--help
```

运行命令（需要 SDK 安装）：

```
dotnet <COMMAND> [-d|--diagnostics] [-h|--help] [--verbosity <LEVEL>]
    [command-options] [arguments]
```

运行应用程序：

```
dotnet [--additionalprobingpath <PATH>] [--additional-deps <PATH>]
    [--fx-version <VERSION>]  [--roll-forward <SETTING>]
    <PATH_TO_APPLICATION> [arguments]

dotnet exec [--additionalprobingpath] [--additional-deps <PATH>]
    [--fx-version <VERSION>]  [--roll-forward <SETTING>]
    <PATH_TO_APPLICATION> [arguments]
```

`--roll-forward` 自 .NET Core 3.x 起可用。使用 .NET Core 2.x 的 `--roll-forward-on-no-candidate-fx`。

描述

`dotnet` 命令有两个函数：

- 它提供了用于处理 .NET 项目的命令。

  例如，`dotnet build` 生成项目。每个命令定义自己的选项和参数。所有命令都支持 `--help` 选项，用于打印有关如何使用命令的简短文档。

- 它运行 .NET 应用程序。

  指定应用程序 `.dll` 文件的路径以运行应用程序。运行应用程序即意味着找到并执行入口点，对于控制台应用，入口点是 `Main` 方法。例如，`dotnet myapp.dll` 运行 `myapp` 应用程序。若要了解部署选项，请参阅 .NET 应用程序部署。

选项

`dotnet` 本身有不同的选项，可用于运行命令和运行应用程序。

dotnet 本身的选项

以下是 `dotnet` 本身的选项。例如 `dotnet --info`。这些选项打印出有关环境的信息。

- `--info`

  打印出有关 .NET 安装和计算机环境(如当前操作系统)的详细信息，并提交 .NET 版本的 SHA。

- `--version`

打印出 dotnet 命令使用的 .NET SDK 版本。包括任何 global.json 的影响

- `--list-runtimes`

  打印出已安装的 .NET 运行时的列表。x86 版本的 SDK 只列出 x86 运行时，而 x64 版本的 SDK 只列出 x64 运行时。

- `--list-sdks`

  打印出已安装的 .NET SDK 的列表。

- `-?|-h|--help`

  打印可用命令列表。

用于运行命令的 SDK 选项

以下选项适用于使用命令的 `dotnet`。例如 `dotnet build --help`。

- `-d|--diagnostics`

  启用诊断输出。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。并非在每个命令中均受支持。请参阅特定的命令页，确定此选项是否可用。

- `-?|-h|--help`

  打印出给定命令的文档，如 `dotnet build --help`。

- `command options`

  每个命令定义特定于该命令的选项。有关可用选项的列表，请参阅特定命令页。

运行时选项

`dotnet` 运行应用程序时，可以使用以下选项。例如 `dotnet myapp.dll --roll-forward Major`。

- `--additionalprobingpath <PATH>`

  包含要进行探测的探测策略和程序集的路径。

- `--additional-deps <PATH>`

  附加 .deps.json 文件的路径。deps.json 文件包含依赖项、编译依赖项和用于解决程序集冲突的版本信息列表。有关详细信息，请参阅 GitHub 上的运行时配置文件。

- `--depsfile <PATH_TO_DEPSFILE>`

  deps.json 文件的路径。.deps.json 文件是一个配置文件，其中包含有关运行应用程序所需的依赖项的信息。此文件由 .NET SDK 生成。

- `--runtimeconfig`

runtimeconfig.template.json 文件的路径。runtimeconfig.template.json 文件是包含运行时设置的配置文件。有关详细信息，请参阅 .NET 运行时配置设置。

- `--roll-forward <SETTING>` 自 .NET Core SDK 3.0 起可用 。

  控制将前滚操作应用于应用的方式。`SETTING` 可以为下列值之一。如果未指定，则 `Minor` 为默认类型。

  - `LatestPatch` - 前滚到最高补丁版本。这会禁用次要版本前滚。
  - `Minor` - 如果缺少所请求的次要版本，则前滚到最低的较高次要版本。如果存在所请求的次要版本，则使用 LatestPatch 策略。
  - `Major` - 如果缺少所请求的主要版本，则前滚到最低的较高主要版本和最低的次要版本。如果存在所请求的主要版本，则使用 Minor 策略。
  - `LatestMinor` - 即使存在所请求的次要版本，仍前滚到最高次要版本。适用于组件托管方案。
  - `LatestMajor` - 即使存在所请求的主要版本，仍前滚到最高主要版本和最高次要版本。适用于组件托管方案。
  - `Disable` - 不前滚。仅绑定到指定的版本。建议不要将此策略用于一般用途，因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

  除 `Disable` 外，所有设置都将使用可用的最高补丁版本。

  前滚行为还可以在项目文件属性、运行时配置文件属性和环境变量中进行配置。有关详细信息，请参阅主版本运行时前滚。

- `--roll-forward-on-no-candidate-fx <N>` 在 .NET Core 2.x SDK 中可用 。

  所需的共享框架不可用时，请定义行为。`N` 可以是：

  - `0` - 禁用次要版本前滚。
  - `1` - 前滚次要版本，但不前滚主版本。这是默认行为。
  - `2` - 前滚次要和主版本。

  有关详细信息，请参阅前滚。

  从 .NET Core 3.0 开始，此选项被 `--roll-forward` 取代，应改为使用此取代项。

- `--fx-version <VERSION>`

  用于运行应用程序的 .NET 运行时版本。

  此选项将重写应用程序 `.runtimeconfig.json` 文件中第一个框架引用的版本。这意味着，仅当只有一个框架引用时，它才会按预期方式工作。如果应用程序具有多个框架引用，则使用此选项可能会导致错误。

dotnet 命令

常规

"	"
dotnet build	生成 .NET 应用程序。
dotnet build-server	与通过生成启动的服务器进行交互。
dotnet clean	清除生成输出。
dotnet help	显示命令更详细的在线文档。
dotnet migrate	将有效的预览版 2 项目迁移到 .NET Core SDK 1.0 项目。

"	"
dotnet msbuild	提供对 MSBuild 命令行的访问权限。
dotnet new	为给定的模板初始化 C# 或 F# 项目。
dotnet pack	创建代码的 NuGet 包。
dotnet publish	发布 .NET 依赖于框架或独立应用程序。
dotnet restore	还原给定应用程序的依赖项。
dotnet run	从源运行应用程序。
dotnet sdk check	显示已安装 SDK 和运行时版本的最新状态。
dotnet sln	用于添加、删除和列出解决方案文件中项目的选项。
dotnet store	将程序集存储到运行时包存储区。
dotnet test	使用测试运行程序运行测试。

项目引用

"	"
dotnet add reference	添加项目引用。
dotnet list reference	列出项目引用。
dotnet remove reference	删除项目引用。

NuGet 包

"	"
dotnet add package	添加 NuGet 包。
dotnet remove package	删除 NuGet 包。

NuGet 命令

"	"
dotnet nuget delete	从服务器删除或取消列出包。
dotnet nuget push	将包推送到服务器，并将其发布。
dotnet nuget locals	清除或列出本地 NuGet 资源，例如 http 请求缓存、临时缓存或计算机范围的全局包文件夹。
dotnet nuget add source	添加 NuGet 源。

dotnet nuget disable source	禁用 NuGet 源。
dotnet nuget enable source	启用 NuGet 源。
dotnet nuget list source	列出所有已配置的 NuGet 源。
dotnet nuget remove source	删除 NuGet 源。
dotnet nuget update source	更新 NuGet 源。

工作负载命令

dotnet workload install	安装可选的工作负载。
dotnet workload list	列出已安装的所有工作负载。
dotnet workload repair	修复所有已安装的工作负载。
dotnet workload search	列出所选工作负载或所有可用的工作负载。
dotnet workload uninstall	卸载工作负载。
dotnet workload update	重新安装所有已安装的工作负载。

全局、工具路径和本地工具命令

工具是控制台应用程序，它们从 NuGet 包中安装并从命令提示符处进行调用。你可自行编写工具，也可安装由第三方编写的工具。工具也称为全局工具、工具路径工具和本地工具。有关详细信息，请参阅 .NET 工具概述。全局和工具路径工具从 .NET Core SDK 2.1 开始可用。本地工具从 .NET Core SDK 3.0 开始可用。

dotnet tool install	在计算机上安装工具。
dotnet tool list	列出计算机上当前安装的所有全局、工具路径或本地工具。
dotnet tool search	在 NuGet.org 中搜索其名称或元数据中具有指定搜索词的工具。
dotnet tool uninstall	从计算机中卸载工具。
dotnet tool update	更新计算机上安装的工具。

其他工具

自 .NET Core SDK 2.1.300 开始，许多使用 `DotnetCliToolReference` 且仅在每个项目的基础上可用的工具现作为 .NET SDK 的一部分提供。下表中列出了这些工具：

dev-certs	创建和管理开发证书。
ef	Entity Framework Core 命令行工具。
user-secrets	管理开发用户机密。
watch	启动文件观察程序，以在更改文件时运行命令。

有关每个工具的详细信息，请键入 `dotnet <tool-name> --help`。

示例

创建新的 .NET 控制台应用程序：

```
dotnet new console
```

生成给定目录中的项目及其依赖项：

```
dotnet build
```

运行应用程序：

```
dotnet myapp.dll
```

另请参阅

- .NET SDK、.NET CLI 和 .NET 运行时使用的环境变量
- 运行时配置文件
- .NET 运行时配置设置

dotnet add package

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet add package` - 向项目文件添加包引用。

摘要

```
dotnet add [<PROJECT>] package <PACKAGE_NAME>
    [-f|--framework <FRAMEWORK>] [--interactive]
    [-n|--no-restore] [--package-directory <PACKAGE_DIRECTORY>]
    [--prerelease] [-s|--source <SOURCE>] [-v|--version <VERSION>]

dotnet add package -h|--help
```

描述

使用 `dotnet add package` 命令可方便地向项目文件添加包引用。运行该命令后，还有一个兼容性检查，确保包与项目中的框架兼容。如果通过了该检查，则将 `<PackageReference>` 元素添加到项目文件并运行 dotnet 还原。

例如，将 `Newtonsoft.Json` 添加到 ToDo.csproj 后的输出如以下示例所示：

```
Writing C:\Users\me\AppData\Local\Temp\tmp95A8.tmp
info : Adding PackageReference for package 'Newtonsoft.Json' into project 'C:\projects\ToDo\ToDo.csproj'.
log  : Restoring packages for C:\Temp\projects\consoleproj\consoleproj.csproj...
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/index.json 79ms
info :   GET https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg
info :   OK https://api.nuget.org/v3-flatcontainer/newtonsoft.json/12.0.1/newtonsoft.json.12.0.1.nupkg 232ms
log  : Installing Newtonsoft.Json 12.0.1.
info : Package 'Newtonsoft.Json' is compatible with all the specified frameworks in project
'C:\projects\ToDo\ToDo.csproj'.
info : PackageReference for package 'Newtonsoft.Json' version '12.0.1' added to file
'C:\projects\ToDo\ToDo.csproj'.
```

ToDo.csproj 文件现包含用于引用的包的 `<PackageReference>` 元素。

```
<PackageReference Include="Newtonsoft.Json" Version="12.0.1" />
```

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

自变量

- `PROJECT`

  指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

- `PACKAGE_NAME`

  要添加的包引用。

选项

- `-f|--framework <FRAMEWORK>`

  仅在以特定框架为目标时添加包引用。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `-n|--no-restore`

  在不执行还原预览和兼容性检查的情况下添加包引用。

- `--package-directory <PACKAGE_DIRECTORY>`

  要在其中还原包的目录。Windows 上的默认包还原位置为 `%userprofile%\.nuget\packages`，macOS 和 Linux 上的默认包还原位置为 `~/.nuget/packages`。有关详细信息，请参阅在 NuGet 中管理全局包、缓存和临时文件夹。

- `--prerelease`

  允许安装预发行包。自 .NET Core 5 SDK 起可用

- `-s|--source <SOURCE>`

  要在还原操作期间使用的 NuGet 包源的 URI。

- `-v|--version <VERSION>`

  包的版本。请参阅 NuGet 包版本控制。

示例

- 将 `Newtonsoft.Json` NuGet 包添加到项目：

```
dotnet add package Newtonsoft.Json
```

- 向项目添加特定版本的包：

```
dotnet add ToDo.csproj package Microsoft.Azure.DocumentDB.Core -v 1.0.0
```

- 使用特定的 NuGet 源添加包：

```
dotnet add package Microsoft.AspNetCore.StaticFiles -s https://dotnet.myget.org/F/dotnet-
core/api/v3/index.json
```

请参阅

- 在 NuGet 中管理全局包、缓存和临时文件夹
- NuGet 包版本控制

dotnet list package

2021/11/16 •

本文适用于：✔ .NET Core 2.2 SDK 及更高版本

“属性”

`dotnet list package` - 列出项目或解决方案的包引用。

摘要

```
dotnet list [<PROJECT>|<SOLUTION>] package [--config <SOURCE>]
    [--deprecated]
    [--framework <FRAMEWORK>] [--highest-minor] [--highest-patch]
    [--include-prerelease] [--include-transitive] [--interactive]
    [--outdated] [--source <SOURCE>] [-v|--verbosity <LEVEL>]
    [--vulnerable]

dotnet list package -h|--help
```

描述

使用 `dotnet list package` 命令，可以方便地列出特定项目或解决方案的所有 NuGet 包引用。首先，需要生成项目，以提供必需资产以供此命令处理。下面的示例展示了 SentimentAnalysis 项目的 `dotnet list package` 命令输出：

```
Project 'SentimentAnalysis' has the following package references
   [netcoreapp2.1]:
   Top-level Package               Requested   Resolved
   > Microsoft.ML                  1.4.0       1.4.0
   > Microsoft.NETCore.App   (A)   [2.1.0, )   2.1.0

(A) : Auto-referenced package.
```

“已请求”列是指项目文件中指定的包版本，可以是一个范围。“已解析”列列出了项目当前使用的版本，始终都是一个值。紧靠名称旁边显示 `(A)` 的包表示从项目设置（`Sdk` 类型、`<TargetFramework>` 或 `<TargetFrameworks>` 属性）推断出的隐式包引用。

使用 `--outdated` 选项，可以确定项目中正在使用的包是否有更高版本。默认情况下，`--outdated` 列出最新稳定包，除非已解析版本也是预发行版本。若要在列出更高版本时包含预发行版本，还请指定 `--include-prerelease` 选项。下面的示例展示了上一个示例中相同项目的 `dotnet list package --outdated --include-prerelease` 命令输出：

```
The following sources were used:
   https://api.nuget.org/v3/index.json
   C:\Program Files (x86)\Microsoft SDKs\NuGetPackages\

Project `SentimentAnalysis` has the following updates to its packages
   [netcoreapp2.1]:
   Top-level Package      Requested   Resolved   Latest
   > Microsoft.ML         1.4.0       1.4.0      1.5.0-preview
```

如果需要确定项目是否有可传递依赖关系，请使用 `--include-transitive` 选项。如果在项目中添加包，它转而又依赖另一个包，就会出现可传递依赖关系。下面的示例展示了 HelloPlugin 项目的 `dotnet list package --include-transitive` 命令运行输出，其中显示顶级包及其依赖的包：

```
Project 'HelloPlugin' has the following package references
   [netcoreapp3.0]:
   Transitive Package      Resolved
   > PluginBase            1.0.0
```

自变量

`PROJECT | SOLUTION`

要对其运行命令的项目或解决方案文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。如果找到多个解决方案或项目，便会抛出错误。

选项

- `--config <SOURCE>`

  在搜索版本更高的包时，要使用的 NuGet 源。需要使用 `--outdated` 选项。

- `--deprecated`

  显示已弃用的包。

- `--framework <FRAMEWORK>`

  只显示适用于指定目标框架的包。若要指定多个框架，请多次重复此选项。例如：`--framework netcoreapp2.2 --framework netstandard2.0`。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--highest-minor`

  在搜索版本更高的包时，仅考虑有匹配的主版本号的包。需要使用 `--outdated` 或 `--deprecated` 选项。

- `--highest-patch`

  在搜索版本更高的包时，仅考虑有匹配的主版本号和次要版本号的包。需要使用 `--outdated` 或 `--deprecated` 选项。

- `--include-prerelease`

  在搜索版本更高的包时，考虑有预发行版本的包。需要使用 `--outdated` 或 `--deprecated` 选项。

- `--include-transitive`

  除了顶级包之外，还列出可传递包。如果指定此选项，可以获取顶级包所依赖的包列表。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--outdated`

  列出版本更高的包。

- `-s|--source <SOURCE>`

  在搜索版本更高的包时，要使用的 NuGet 源。需要使用 `--outdated` 或 `--deprecated` 选项。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

- `--vulnerable`

  列出有已知漏洞的包。无法与 `--deprecated` 或 `--outdated` 选项合并。

示例

- 列出特定项目的包引用：

```
dotnet list SentimentAnalysis.csproj package
```

- 列出有更高版本（包括预发行版本）的包引用：

```
dotnet list package --outdated --include-prerelease
```

- 列出特定目标框架的包引用：

```
dotnet list package --framework netcoreapp3.0
```

dotnet remove package

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

名称

`dotnet remove package` - 从项目文件删除包引用。

摘要

```
dotnet remove [<PROJECT>] package <PACKAGE_NAME>

dotnet remove package -h|--help
```

说明

使用 `dotnet remove package` 命令可方便地从项目删除 NuGet 包引用。

自变量

`PROJECT`

指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

`PACKAGE_NAME`

要删除的包引用。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 从当前目录中的项目删除 `Newtonsoft.Json` NuGet 包：

  ```
  dotnet remove package Newtonsoft.Json
  ```

dotnet add reference

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet add reference` - 添加项目到项目 (P2P) 引用。

摘要

```
dotnet add [<PROJECT>] reference [-f|--framework <FRAMEWORK>]
    [--interactive] <PROJECT_REFERENCES>

dotnet add reference -h|--help
```

描述

使用 `dotnet add reference` 命令可方便地向项目添加项目引用。运行该命令后，会将 `<ProjectReference>` 元素添加到项目文件。

```
<ItemGroup>
  <ProjectReference Include="app.csproj" />
  <ProjectReference Include="..\lib2\lib2.csproj" />
  <ProjectReference Include="..\lib1\lib1.csproj" />
</ItemGroup>
```

自变量

- `PROJECT`

  指定项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

- `PROJECT_REFERENCES`

  要添加的项目到项目 (P2P) 引用。指定一个或多个项目。基于 Unix/Linux 的系统支持 glob 模式。

选项

- `-f|--framework <FRAMEWORK>`

  仅在以特定框架为目标时使用 TFM 格式添加项目引用。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

示例

- 添加项目引用：

```
dotnet add app/app.csproj reference lib/lib.csproj
```

- 向当前目录中的项目添加多个项目引用：

```
dotnet add reference lib1/lib1.csproj lib2/lib2.csproj
```

- 使用 glob 模式在 Linux/Unix 上添加多个项目引用：

```
dotnet add app/app.csproj reference **/*.csproj
```

dotnet list reference

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet list reference` - 列出项目到项目引用。

摘要

```
dotnet list [<PROJECT>] reference

dotnet list -h|--help
```

描述

使用 `dotnet list reference` 命令可方便地列出给定项目的项目引用。

自变量

- `PROJECT`

  要操作的项目文件。如果未指定文件，此命令会搜索当前目录来获取文件。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 列出指定项目的项目引用：

  ```
  dotnet list app/app.csproj reference
  ```

- 列出当前目录中的项目的项目引用：

  ```
  dotnet list reference
  ```

dotnet remove reference

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet remove reference` - 删除项目到项目 (P2P) 引用。

摘要

```
dotnet remove [<PROJECT>] reference [-f|--framework <FRAMEWORK>]
    <PROJECT_REFERENCES>

dotnet remove reference -h|--help
```

描述

使用 `dotnet remove reference` 命令可方便地从项目删除项目引用。

自变量

`PROJECT`

目标项目文件。如果未指定，此命令会搜索当前目录来获取一个项目文件。

`PROJECT_REFERENCES`

要删除的项目到项目 (P2P) 引用。可指定一个或多个项目。基于 Unix/Linux 的终端支持 Glob 模式。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-f|--framework <FRAMEWORK>`

  仅在以特定框架为目标时使用 TFM 格式删除引用。

示例

- 从指定项目删除项目引用：

  ```
  dotnet remove app/app.csproj reference lib/lib.csproj
  ```

- 从当前目录中的项目删除多个项目引用：

  ```
  dotnet remove reference lib1/lib1.csproj lib2/lib2.csproj
  ```

- 使用 Unix/Linux 的 glob 模式删除多个项目引用：

```
dotnet remove app/app.csproj reference **/*.csproj`
```

dotnet build

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet build` - 生成项目及其所有依赖项。

摘要

```
dotnet build [<PROJECT>|<SOLUTION>] [-a|--arch <ARCHITECTURE>]
    [-c|--configuration <CONFIGURATION>] [-f|--framework <FRAMEWORK>]
    [--force] [--interactive] [--no-dependencies] [--no-incremental]
    [--no-restore] [--nologo] [--no-self-contained] [--os <OS>]
    [-o|--output <OUTPUT_DIRECTORY>] [-r|--runtime <RUNTIME_IDENTIFIER>]
    [--self-contained [true|false]] [--source <SOURCE>]
    [-v|--verbosity <LEVEL>] [--version-suffix <VERSION_SUFFIX>]

dotnet build -h|--help
```

描述

`dotnet build` 命令将项目及其依赖项生成为一组二进制文件。二进制文件包括扩展名为 .dll 的中间语言 (IL) 文件中的项目代码。根据项目类型和设置，可能会包含其他文件，例如：

- 可用于运行应用程序的可执行文件（如果项目类型是面向 .NET Core 3.0 或更高版本的可执行文件）。
- 用于调试的扩展名为 .pdb 的符号文件。
- 列出了应用程序或库的依赖项的 .deps.json 文件。
- 用于指定应用程序的共享运行时及其版本的 .runtimeconfig.json 文件。
- 项目通过项目引用或 NuGet 包引用所依赖的其他库。

对于目标版本低于 .NET Core 3.0 的可执行项目，通常不会将 NuGet 中的库依赖项复制到输出文件夹。而是在运行时从 NuGet 全局包文件夹中对其进行解析。考虑到这一点，`dotnet build` 的产品还未准备好转移到另一台计算机进行运行。要创建可部署的应用程序版本，需要发布该应用程序（例如，使用 dotnet publish 命令）。有关详细信息，请参阅 .NET 应用程序部署。

对于面向 .NET Core 3.0 及更高版本的可执行项目，库依赖项会被复制到输出文件夹。这意味着如果没有其他任何特定于发布的逻辑（例如，Web 项目具有的逻辑），则应可部署生成输出。

隐式还原

构建需要 *project.assets.json* 文件，该文件列出了你的应用程序的依赖项。此文件在 `dotnet restore` 执行时创建。如果资产文件未就位，那么工具将无法解析引用程序集，进而导致错误生成。

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

以长格式传入时，此命令支持 `dotnet restore` 选项（例如，`--source`）。不支持缩写选项，例如 `-s`。

可执行文件或库输出

项目是否可执行由项目文件中的 `<OutputType>` 属性决定。以下示例显示生成可执行代码的项目：

```
<PropertyGroup>
  <OutputType>Exe</OutputType>
</PropertyGroup>
```

要生成库，请省略 `<OutputType>` 属性或将其值更改为 `Library`。库的 IL DLL 不包含入口点，因此无法执行。

MSBuild

`dotnet build` 使用 MSBuild 生成项目，因此它支持并行生成和增量生成。有关详细信息，请参阅增量生成。

除其自己的选项外，`dotnet build` 命令也接受 MSBuild 选项，如用来设置属性的 `-p` 或用来定义记录器的 `-l`。有关这些选项的详细信息，请参阅 MSBuild 命令行参考。或者也可以使用 dotnet msbuild 命令。

> **NOTE**
> 如果 `dotnet build` 由 `dotnet run` 自动运行，则不遵守 `-property:property=value` 等参数。

运行 `dotnet build` 等同于运行 `dotnet msbuild -restore`；但是，输出的默认详细程度不同。

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

自变量

`PROJECT | SOLUTION`

要生成的项目或解决方案文件。如果未指定项目或解决方案文件，MSBuild 会在当前工作目录中搜索文件扩展名以 *proj* 或 *sln* 结尾的文件并使用该文件。

选项

- `-a|--arch <ARCHITECTURE>`

  指定目标体系结构。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--arch x86` 会将 RID 设置为 `win-x86`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

  编译特定框架。必须在项目文件中定义该框架。

- `--force`

  强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--no-dependencies`

  忽略项目到项目 (P2P) 引用，并仅生成指定的根项目。

- `--no-incremental`

  将生成标记为对增量生成不安全。此标记关闭增量编译，并强制完全重新生成项目依赖项关系图。

- `--no-restore`

  在生成期间不执行隐式还原。

- `--nologo`

  不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `--no-self-contained`

  将应用程序发布为与框架相关的应用程序。必须在目标计算机上安装兼容的 .NET 运行时才能运行应用程序。自 .NET 6 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

  放置生成二进制文件的目录。如果未指定，则默认路径为 `./bin/<configuration>/<framework>/`。对于具有多个目标框架的项目（通过 `TargetFrameworks` 属性），在指定此选项时还需要定义 `--framework`。

- `--os <OS>`

  指定目标操作系统 (OS)。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--os os` 会将 RID 设置为 `os-x64`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  指定目标运行时。有关运行时标识符 (RID) 的列表，请参阅 RID 目录。如果将此选项与 .NET 6 SDK 结合使用，则还要使用 `--self-contained` 或 `--no-self-contained`。

- `--self-contained [true|false]`

  .NET 运行时随应用程序一同发布，因此无需在目标计算机上安装运行时。如果指定了运行时标识符，则默认值为 `true`。自 .NET 6 SDK 起可用。

- `--source <SOURCE>`

  要在还原操作期间使用的 NuGet 包源的 URI。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

- `--version-suffix <VERSION_SUFFIX>`

  设置生成项目时使用的 `$(VersionSuffix)` 属性的值。这仅在未设置 `$(Version)` 属性时有效。然后，`$(Version)` 设置为 `$(VersionPrefix)` 与 `$(VersionSuffix)` 组合，并用短划线分隔。

示例

- 生成项目及其依赖项：

```
dotnet build
```

- 使用“发布”配置生成项目及其依赖项：

```
dotnet build --configuration Release
```

- 针对特定运行时（本例中为 Ubuntu 18.04）生成项目及其依赖项：

```
dotnet build --runtime ubuntu.18.04-x64
```

- 生成项目，并在还原操作过程中使用指定的 NuGet 包源：

```
dotnet build --source c:\packages\mypackages
```

- 生成项目并设置版本 1.2.3.4 作为使用 `-p` MSBuild 选项的生成参数：

```
dotnet build -p:Version=1.2.3.4
```

dotnet build-server

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet build-server` -与通过生成启动的服务器进行交互。

摘要

```
dotnet build-server shutdown [--msbuild] [--razor] [--vbcscompiler]

dotnet build-server shutdown -h|--help

dotnet build-server -h|--help
```

命令

- `shutdown`

  关闭从 dotnet 启动的生成服务器。默认情况下会关闭所有服务器。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--msbuild`

  关闭 MSBuild 生成服务器。

- `--razor`

  关闭 Razor 生成服务器。

- `--vbcscompiler`

  关闭 VB/C# 编译器生成服务器。

dotnet clean

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

名称

`dotnet clean` - 清除项目输出。

摘要

```
dotnet clean [<PROJECT>|<SOLUTION>] [-c|--configuration <CONFIGURATION>]
    [-f|--framework <FRAMEWORK>] [--interactive]
    [--nologo] [-o|--output <OUTPUT_DIRECTORY>]
    [-r|--runtime <RUNTIME_IDENTIFIER>] [-v|--verbosity <LEVEL>]

dotnet clean -h|--help
```

说明

`dotnet clean` 命令可清除上一个生成的输出。它以 MSBuild 目标 的形式实现，以便在运行命令时对项目进行评估。只会清除在生成过程中创建的输出。中间 (*obj*) 和最终输出 (*bin*) 文件夹都会被清除。

参数

`PROJECT | SOLUTION`

要清理的 MSBuild 项目或解决方案。如果未指定项目或解决方案文件，MSBuild 会在当前工作目录中搜索文件扩展名以 *proj* 或 *sln* 结尾的文件并使用该文件。

选项

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。只有在生成期间指定了此选项，才必须在清除时使用此选项。

- `-f|--framework <FRAMEWORK>`

  在生成时指定的框架。必须在项目文件中定义该框架。如果在生成时指定了框架，则必须在清除时指定框架。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--nologo`

  不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

  包含要清理的生成项目的目录。如果在生成项目时指定了框架，则使用输出目录开关指定 `-f|--framework <FRAMEWORK>` 开关。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  清除指定运行时的输出文件夹。在创建独立部署 (SCD) 时使用此选项。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 清除项目的默认生成：

```
dotnet clean
```

- 清除使用版本配置生成的项目：

```
dotnet clean --configuration Release
```

dotnet format

2021/11/16 •

本文适用于：✔ .NET 6.x SDK 及更高版本

名称

`dotnet format` - 设置代码格式以匹配 `editorconfig` 设置。

摘要

```
dotnet format [options] [<PROJECT | SOLUTION>]

dotnet format -h|--help
```

说明

`dotnet format` 是一种代码格式化程序，它将样式首选项应用于项目或解决方案。将从 .editorconfig 文件中读取首选项（如果存在），否则将使用一组默认首选项。有关详细信息，请查看 EditorConfig 文档。

参数

`PROJECT | SOLUTION`

用于运行代码格式化的 MSBuild 项目或解决方案。如果未指定项目或解决方案文件，MSBuild 会在当前工作目录中搜索文件扩展名以 *proj* 或 *sln* 结尾的文件并使用该文件。

选项

要成功执行 `dotnet format` 命令，下面的选项都不是必需的，但是可以用来进一步自定义格式化的内容和需要遵循的规则。

- `--diagnostics <DIAGNOSTICS>`

  以空格分隔的诊断 ID 列表，在修复代码样式或第三方问题时用作筛选器。默认值为 .editorconfig 文件中列出的 ID。有关可以指定的内置分析器规则 ID 的列表，请参阅用于代码分析样式规则的 ID 列表。

- `--severity`

  要修复的诊断的最低严重性。允许使用的值为 `info`、`warn` 和 `error`。默认值为 `warn`。

- `--no-restore`

  请勿在设置格式之前执行隐式还原。默认设置是执行隐式还原。

- `--verify-no-changes`

  验证不会执行任何格式更改。如果任何文件已设置格式，则以非零退出代码终止。

- `--include <INCLUDE>`

  要包含在格式设置中的以空格分隔的相关文件或文件夹路径列表。默认为解决方案或项目中的所有文件。

- `--exclude <EXCLUDE>`

  要从格式设置中排除的以空格分隔的相关文件或文件夹路径列表。默认值为 none。

- `--include-generated`

  设置 SDK 生成的文件的格式。

- `-v|--verbosity <LEVEL>`

  设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值是 `m[inimal]`。

- `--binarylog <BINARY-LOG-PATH>`

  将所有项目或解决方案加载信息记录到二进制日志文件中。

- `--report <REPORT-PATH>`

  在 `<REPORT_PATH>` 指定的目录中生成 JSON 报告。

- `-h|--help`

  显示帮助和使用情况信息

子命令

空格

`dotnet format whitespace` - 设置代码格式以匹配空白的 `editorconfig` 设置。

说明

`dotnet format whitespace` 子命令将只运行与空白格式设置相关的格式设置规则。有关可以在 .editorconfig 文件中指定的可能格式设置选项的完整列表，请参阅 C# 格式设置规则。

选项

- `--folder`

  将 `<PROJECT | SOLUTION>` 参数视为代码文件的简单文件夹的路径。

Style

`dotnet format style` - 设置代码格式以匹配代码样式的 EditorConfig 设置。

说明

`dotnet format style` 子命令将只运行与代码样式格式设置相关的格式设置规则。有关可以在 `editorconfig` 文件中指定的格式设置选项的完整列表，请参阅代码样式规则。

选项

- `--diagnostics <DIAGNOSTICS>`

  以空格分隔的诊断 ID 列表，在修复代码样式或第三方问题时用作筛选器。默认值为 .editorconfig 文件中列出的 ID。有关可以指定的内置分析器规则 ID 的列表，请参阅用于代码分析样式规则的 ID 列表。

- `--severity`

  要修复的诊断的最低严重性。允许使用的值为 `info`、`warn` 和 `error`。默认值为 `warn`

分析器

`dotnet format analyzers` - 设置代码格式以匹配分析器的 `editorconfig` 设置。

说明

`dotnet format analyzers` 子命令将只运行与分析器相关的格式设置规则。有关可在 `editorconfig` 文件中指定

的分析器规则的列表，请参阅代码样式规则。

选项

- `--diagnostics <DIAGNOSTICS>`

  以空格分隔的诊断 ID 列表，在修复代码样式或第三方问题时用作筛选器。默认值为 .editorconfig 文件中列出的 ID。有关可以指定的内置分析器规则 ID 的列表，请参阅用于代码分析样式规则的 ID 列表。

- `--severity`

  要修复的诊断的最低严重性。允许使用的值为 `info`、`warn` 和 `error`。默认值为 `warn`。

示例

- 设置解决方案中所有代码的格式：

```
dotnet format ./solution.sln
```

- 清理应用程序项目的所有代码：

```
dotnet format ./src/application.csproj
```

- 验证所有代码的格式是否正确：

```
dotnet format --verify-no-changes
```

- 清理 src 和 tests 目录中的所有代码，但不清理 src/submodule-a 中的代码：

```
dotnet format --include ./src/ ./tests/ --exclude ./src/submodule-a/
```

dotnet help 参考

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

名称

`dotnet help` - 显示指定命令更详细的在线文档。

摘要

```
dotnet help <COMMAND_NAME> [-h|--help]
```

说明

`dotnet help` 命令打开 docs.microsoft.com 参考页，以提供指定命令的更多详细信息。

参数

- `COMMAND_NAME`

  .NET CLI 命令名称。有关有效 CLI 命令的列表，请参阅 CLI 命令。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 打开 dotnet new 命令的文档页：

  ```
  dotnet help new
  ```

dotnet migrate

2021/11/16 •

本文适用于：✔ .NET Core 2.x SDK

“属性”

`dotnet migrate` - 将预览版 2 .NET Core 项目迁移到 .NET Core SDK 样式的项目中。

摘要

```
dotnet migrate [<SOLUTION_FILE|PROJECT_DIR>] [--format-report-file-json <REPORT_FILE>]
    [-r|--report-file <REPORT_FILE>] [-s|--skip-project-references [Debug|Release]]
    [--skip-backup] [-t|--template-file <TEMPLATE_FILE>] [-v|--sdk-package-version]
    [-x|--xproj-file]

dotnet migrate -h|--help
```

描述

此命令已弃用。自 .NET Core 3.0 SDK 起，`dotnet migrate` 命令不再可用。它只能将预览版 2 .NET Core 项目迁移到不支持的 1.x .NET Core 项目。

默认情况下，命令迁移根项目和根项目包含的任何项目引用。在运行时使用 `--skip-project-references` 选项禁用此行为。

可在下列资产上执行迁移：

- 通过指定 *project.json* 文件进行迁移的单个项目。
- 通过将路径传递到 *global.json* 文件在 *global.json* 文件中指定的所有目录。
- 一个 *solution.sln* 文件，它迁移在解决方案中引用的项目。
- 以递归方式迁移给定目录的所有子目录。

`dotnet migrate` 命令将迁移的 *project.json* 文件保存在 `backup` 目录中，如果该目录不存在，将创建一个。使用 `--skip-backup` 选项重写此行为。

默认情况下，迁移操作会将迁移过程的状态输出到标准输出 (STDOUT)。如果使用 `--report-file <REPORT_FILE>` 选项，输出将保存到指定的文件中。

`dotnet migrate` 命令仅支持有效的预览版 2 基于 *project.json* 的项目。这意味着，不能使用它将 DNX 或预览版 1 基于 *project.json* 的项目直接迁移到 MSBuild/csproj 项目。首先需要将项目手动迁移到预览版 2 基于 *project.json* 的项目，然后使用 `dotnet migrate` 命令迁移该项目。

自变量

`PROJECT_JSON/GLOBAL_JSON/SOLUTION_FILE/PROJECT_DIR`

下列路径之一：

- 要迁移的 *project.json* 文件。
- global.json 文件：迁移在 global.json 中指定的文件夹 。
- solution.sln 文件：迁移该解决方案中引用的项目。

- 要迁移的目录：在指定的目录中以递归方式搜索要迁移的 project.json 文件。

如未指定，则默认为当前目录。

选项

`--format-report-file-json <REPORT_FILE>`

将迁移报告文件（而非用户消息）作为 JSON 输出。

`-h|--help`

打印出有关命令的简短帮助。

`-r|--report-file <REPORT_FILE>`

除控制台外，还将迁移报告输出到文件。

`-s|--skip-project-references [Debug|Release]`

跳过迁移项目引用。默认情况下，以递归方式迁移项目引用。

`--skip-backup`

在成功迁移后，跳过将 *project.json*、*global.json* 和 **.xproj* 移动到 `backup` 目录的步骤。

`-t|--template-file <TEMPLATE_FILE>`

用于迁移的模板 csproj 文件。默认情况下，使用与被 `dotnet new console` 删除的模板相同的模板。

`-v|--sdk-package-version <VERSION>`

在已迁移应用中将被引用的 sdk 包的版本。默认为 `dotnet new` 中 SDK 的版本。

`-x|--xproj-file <FILE>`

要使用的 xproj 文件的路径。当项目目录中有多个 xproj 时需要。

示例

将当前目录中的项目及其所有项目迁移到项目依赖项：

```
dotnet migrate
```

迁移 *global.json* 文件所包含的所有项目：

```
dotnet migrate path/to/global.json
```

仅迁移当前项目，不迁移项目到项目 (P2P) 的依赖项。此外，使用特定的 SDK 版本：

```
dotnet migrate -s -v 1.0.0-preview4
```

dotnet msbuild

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet msbuild` - 生成项目及其所有依赖项。注意：如果有多个解决方案或项目文件，可能需要指定一个。

摘要

```
dotnet msbuild <MSBUILD_ARGUMENTS>

dotnet msbuild -h
```

描述

`dotnet msbuild` 命令允许访问功能完备的 MSBuild。

该命令与仅适用于 SDK 样式项目的现有 MSBuild 命令行客户端具有完全相同的功能。选项一致。有关可用选项的详细信息，请参阅 MSBuild 命令行参考。

dotnet build 命令相当于 `dotnet msbuild -restore`。如果不想生成项目，并且拥有要运行的特定目标，请使用 `dotnet build` 或 `dotnet msbuild` 并指定目标。

示例

- 生成项目及其依赖项：

  ```
  dotnet msbuild
  ```

- 使用“发布”配置生成项目及其依赖项：

  ```
  dotnet msbuild -property:Configuration=Release
  ```

- 运行发布目标并发布 `osx.10.11-x64` RID：

  ```
  dotnet msbuild -target:Publish -property:RuntimeIdentifiers=osx.10.11-x64
  ```

- 请参阅包含 SDK 添加的所有目标的整个项目：

  ```
  dotnet msbuild -preprocess
  dotnet msbuild -preprocess:<fileName>.xml
  ```

dotnet new

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

“属性”

`dotnet new` - 根据指定的模板，创建新的项目、配置文件或解决方案。

摘要

```
dotnet new <TEMPLATE> [--dry-run] [--force] [-lang|--language {"C#"|"F#"|VB}]
    [-n|--name <OUTPUT_NAME>] [--no-update-check] [-o|--output <OUTPUT_DIRECTORY>] [Template options]

dotnet new -h|--help
```

描述

`dotnet new` 命令基于模板创建 .NET 项目或其他项目。

命令调用模板引擎，以根据指定的模板和选项在磁盘上创建项目。

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

自变量

- `TEMPLATE`

  调用命令时要实例化的模板。每个模板可能具有可传递的特定选项。有关详细信息，请参阅模板选项。

  可以运行 `dotnet new --list` 以查看所有已安装模板的列表。

  从 .NET Core 3.0 SDK 开始并以 .NET Core 5.0.300 SDK 结束，当你在以下情况下调用 `dotnet new` 命令时，CLI 将在 NuGet.org 中搜索模板：

  - 如果在调用 `dotnet new` 时 CLI 找不到模板匹配项，即使是部分匹配也不行。
  - 如果有较新版本的模板可用。在这种情况下，将创建项目或工件，但 CLI 会就模板的更新版本发出警告。

  从 .NET Core 5.0.300 SDK 开始，应使用 `--search` 选项在 NuGet.org. 中搜索模板。

  下表显示随 .NET SDK 一起预安装的模板。模板的默认语言显示在括号内。单击短名称链接可查看特定的模板选项。

模板	短名称	语言	TAGS	创建选项
控制台应用程序	`console`	[C#]、F#、VB	常用/控制台	1.0
类库	`classlib`	[C#]、F#、VB	常用/库	1.0
WPF 应用程序	`wpf`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 类库	`wpflib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 自定义控件库	`wpfcustomcontrollib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 用户控件库	`wpfusercontrollib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
Windows 窗体 (WinForms) 应用程序	`winforms`	[C#]、VB	常用/WinForms	3.0（对于 VB，则为 5.0）
Windows 窗体 (WinForms) 类库	`winformslib`	[C#]、VB	常用/WinForms	3.0（对于 VB，则为 5.0）
Worker Service	`worker`	[C#]	常用/Worker/Web	3.0
单元测试项目	`mstest`	[C#]、F#、VB	测试/MSTest	1.0
NUnit 3 测试项目	`nunit`	[C#]、F#、VB	测试/NUnit	2.1.400
NUnit 3 测试项	`nunit-test`	[C#]、F#、VB	测试/NUnit	2.2
xUnit 测试项目	`xunit`	[C#]、F#、VB	测试/xUnit	1.0
Razor 组件	`razorcomponent`	[C#]	Web/ASP.NET	3.0
Razor 页	`page`	[C#]	Web/ASP.NET	2.0
MVC ViewImports	`viewimports`	[C#]	Web/ASP.NET	2.0
MVC ViewStart	`viewstart`	[C#]	Web/ASP.NET	2.0
Blazor 服务器应用	`blazorserver`	[C#]	Web/Blazor	3.0
Blazor WebAssembly 应用	`blazorwasm`	[C#]	Web/Blazor/WebAssembly	3.1.300
ASP.NET Core 空	`web`	[C#]，F#	Web/空	1.0
ASP.NET Core Web 应用程序 (Model-View-Controller)	`mvc`	[C#]，F#	Web/MVC	1.0

模板名称	短名称	语言	TAGS	创建者
ASP.NET Core Web 应用程序	`webapp, razor`	[C#]	Web/MVC/Razor Pages	2.2、2.0
含 Angular 的 ASP.NET Core	`angular`	[C#]	Web/MVC/SPA	2.0
含 React.js 的 ASP.NET Core	`react`	[C#]	Web/MVC/SPA	2.0
含 React.js 和 Redux 的 ASP.NET Core	`reactredux`	[C#]	Web/MVC/SPA	2.0
Razor 类库	`razorclasslib`	[C#]	Web/Razor/库/Razor 类库	2.1
ASP.NET Core Web API	`webapi`	[C#], F#	Web/WebAPI	1.0
ASP.NET Core gRPC 服务	`grpc`	[C#]	Web/gRPC	3.0
dotnet gitignore 文件	`gitignore`		配置	3.0
global.json 文件	`globaljson`		配置	2.0
NuGet 配置	`nugetconfig`		配置	1.0
Dotnet 本地工具清单文件	`tool-manifest`		配置	3.0
Web 配置	`webconfig`		配置	1.0
解决方案文件	`sln`		解决方案	1.0
协议缓冲区文件	`proto`		Web/gRPC	3.0
EditorConfig 文件	`editorconfig` (dotnet-new-sdk-templates.md#editorconfig)		Config	6.0

选项

- `--dry-run`

  显示有关以下内容的摘要：给定命令运行导致模板创建时发生的情况。自 .NET Core 2.2 SDK 起可用。

- `--force`

  强制生成内容，即使会更改现有文件，也不例外。当选择的模板将覆盖输出目录中的现有文件时，需要执行此操作。

- `-?|-h|--help`

打印命令帮助。可针对 `dotnet new` 命令本身或任何模板调用它。例如 `dotnet new mvc --help`。

- `-lang|--language {C#|F#|VB}`

  要创建的模板的语言。接受的语言因模板而异(请参阅参数部分中的默认值)。对于某些模板无效。

  > **NOTE**
  > 某些 shell 将 `#` 解释为特殊字符。在这些情况下，请将语言参数值括在引号中。例如 `dotnet new console -lang "F#"`。

- `-n|--name <OUTPUT_NAME>`

  所创建的输出的名称。如果未指定名称，使用的是当前目录的名称。

- `-no-update-check`

  禁止在实例化模板时检查模板包更新。自 .NET 6.0.100 SDK 起可用。从使用 `dotnet new --install` 安装的模板包实例化模板时，`dotnet new` 会检查模板是否有更新。从 .NET 6 开始，不对 .NET 默认模板进行更新检查。若要更新 .NET 默认模板，请安装 .NET SDK 的修补程序版本。

- `-o|--output <OUTPUT_DIRECTORY>`

  用于放置生成的输出的位置。默认为当前目录。

模板选项

每个模板都可能定义了附加选项。有关详细信息，请参阅适用于 `dotnet new` 的 .NET 默认模板。

示例

- 创建 C# 控制台应用程序项目：

```
dotnet new console
```

- 在当前目录中创建 F# 控制台应用程序项目：

```
dotnet new console --language "F#"
```

- 在指定的目录中创建 .NET Standard 2.0 类库项目：

```
dotnet new classlib --framework "netstandard2.0" -o MyLibrary
```

- 在当前目录中新建没有设置身份验证的 ASP.NET Core C# MVC 项目：

```
dotnet new mvc -au None
```

- 创建新的 xUnit 项目：

```
dotnet new xunit
```

- 在当前目录中创建 global.json，将 SDK 版本设置为 3.1.101：

```
dotnet new globaljson --sdk-version 3.1.101
```

- 显示 C# 控制台应用程序模板的帮助：

```
dotnet new console -h
```

- 显示 F# 控制台应用程序模板的帮助：

```
dotnet new console --language "F#" -h
```

请参阅

- dotnet new --list 选项
- dotnet new --search 选项
- dotnet new --install 选项
- 适用于 dotnet new 的 .NET 默认模板
- dotnet new 自定义模板
- 创建 dotnet new 自定义模板
- dotnet/dotnet-template-samples GitHub 存储库

dotnet new--list 选项

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

名称

`dotnet new --list` - 列出要使用 `dotnet new` 运行的可用模板。

摘要

```
dotnet new [<TEMPLATE_NAME>] -l|--list [--author <AUTHOR>] [-lang|--language {"C#"|"F#"|VB}]
    [--tag <TAG>] [--type <TYPE>] [--columns <COLUMNS>] [--columns-all]
```

说明

`dotnet new --list` 选项列出了要与 `dotnet new` 配合使用的可用模板。如果指定了 <TEMPLATE_NAME>，则列出包含指定名称的模板。此选项仅列出默认的和已安装的模板。若要在 NuGet 中查找可在本地安装的模板，请使用 `--search` 选项。

自变量

- `TEMPLATE_NAME`

  如果指定了参数，将只显示模板名称或短名称中包含 `<TEMPLATE_NAME>` 的模板。

  > **NOTE**
  > 从 .NET SDK 6.0.100 开始，可以将 `<TEMPLATE_NAME>` 参数放在 `--list` 选项后面。例如，`dotnet new --list web` 与 `dotnet new web --list` 的结果相同。不允许使用多个参数。

选项

- `--author <AUTHOR>`

  基于模板作者筛选模板。支持部分匹配。自 .NET Core 5.0.300 SDK 起可用。

- `--columns <COLUMNS>`

  要在输出中显示的列的以逗号分隔的列表。支持的列包括：

  - `language` - 模板支持的语言的以逗号分隔的列表。
  - `tags` - 模板标记列表。
  - `author` - 模板作者。
  - `type` - 模板类型：项目或项。

  始终显示模板名称和短名称。默认的列列表是模板名称、短名称、语言和标记。该列表等效于指定 `--columns=language,tags`。自 .NET Core 5.0.300 SDK 起可用。

- `--columns-all`

在输出中显示所有列。自 .NET Core 5.0.300 SDK 起可用。

- `-lang|--language {C#|F#|VB}`

  根据模板支持的语言筛选模板。接受的语言因模板而异。对于某些模板无效。

  > **NOTE**
  > 某些 shell 将 `#` 解释为特殊字符。在这些情况下，请将语言参数值括在引号中。例如 `dotnet new --list --language "F#"`。

- `--tag <TAG>`

  基于模板标记筛选模板。若要选择，模板必须至少具有一个与条件完全匹配的标记。自 .NET Core 5.0.300 SDK 起可用。

- `--type <TYPE>`

  基于模板类型筛选模板。预定义的值为 `project`、`item` 和 `solution`。

示例

- 列出所有模板

```
dotnet new --list
```

- 列出单页应用程序 (SPA) 模板：
  - 自 .NET SDK 6.0.100 起

```
dotnet new --list spa
```

  - .NET SDK 6.0.100 之前

```
dotnet new spa --list
```

- 列出与“we”子字符串匹配的所有模板。
  - 自 .NET SDK 6.0.100 起

```
dotnet new --list we
```

  - .NET SDK 6.0.100 之前

```
dotnet new we --list
```

- 列出与支持 F# 语言的“we”子字符串匹配的所有模板。

```
dotnet new --list we --language "F#"
```

- 列出所有项模板。

```
dotnet new --list --type item
```

- 列出所有 C# 模板，从而在输出中显示作者和类型。

```
dotnet new --list --language "C#" --columns "author,type"
```

另请参阅

- dotnet new 命令
- dotnet new --search 选项
- dotnet new 自定义模板

dotnet new --search 选项

2021/11/16 •

本文适用于：✔ .NET Core 5.0.300 SDK 及更高版本

名称

`dotnet new --search` - 在 NuGet.org 上搜索 `dotnet new` 支持的模板。

摘要

```
dotnet new <TEMPLATE_NAME> --search

dotnet new [<TEMPLATE_NAME>] --search [--author <AUTHOR>] [-lang|--language {"C#"|"F#"|VB}]
    [--package <PACKAGE>] [--tag <TAG>] [--type <TYPE>]
    [--columns <COLUMNS>] [--columns-all]
```

说明

`dotnet new --search` 选项在 NuGet.org 上搜索 `dotnet new` 支持的模板。指定 <TEMPLATE_NAME> 时，将搜索包含指定名称的模板。

自变量

- `TEMPLATE_NAME`

  如果指定了参数，将只显示模板名称或短名称中包含 `<TEMPLATE_NAME>` 的模板。如果未指定 `--author`、`--language`、`--package`、`--tag` 或 `--type` 选项，则该参数是必需的。

  > **NOTE**
  > 从 .NET SDK 6.0.100 开始，可以将 `<TEMPLATE_NAME>` 参数放在 `--search` 选项后面。例如，`dotnet new --search web` 与 `dotnet new web --search` 的结果相同。不允许使用多个参数。

选项

- `--author <AUTHOR>`

  基于模板作者筛选模板。支持部分匹配。

- `--columns <COLUMNS>`

  要在输出中显示的列的以逗号分隔的列表。支持的列包括：

  - `language` - 模板支持的语言的以逗号分隔的列表。
  - `tags` - 模板标记列表。
  - `author` - 模板作者。
  - `type` - 模板类型：项目或项。

  模板名称、短名称、包名称和下载总数将始终显示。列的默认列表为模板名称、短名称、作者、语言、包和下载总数。该列表等效于指定 `--columns=author,language`。

- `--columns-all`

  在输出中显示所有列。

- `-lang|--language {C#|F#|VB}`

  根据模板支持的语言筛选模板。接受的语言因模板而异。对于某些模板无效。

  > **NOTE**
  > 某些 shell 将 `#` 解释为特殊字符。在这些情况下，请将语言参数值括在引号中。例如 `dotnet new --search --language "F#"`。

- `--package <PACKAGE>`

  基于 NuGet 包 ID 筛选模板。支持部分匹配。

- `--tag <TAG>`

  基于模板标记筛选模板。若要选择，模板必须至少具有一个与条件完全匹配的标记。

- `--type <TYPE>`

  基于模板类型筛选模板。预定义的值为 `project`、`item` 和 `solution`。

  > **NOTE**
  > 若要确保模板包显示在 `dotnet new --search` 结果中，请将 NuGet 包类型设置为 `Template`。

示例

- 搜索 NuGet.org 上提供的与 spa 子字符串匹配的所有模板。
  - 自 .NET SDK 6.0.100 起

```
dotnet new --search spa
```

  - .NET SDK 6.0.100 之前

```
dotnet new spa --search
```

- 搜索 NuGet.org 上提供的与 we 子字符串匹配且支持 F# 语言的所有模板。
  - 自 .NET SDK 6.0.100 起

```
dotnet new --search we --language "F#"
```

  - .NET SDK 6.0.100 之前

```
dotnet new we --search --language "F#"
```

- 搜索项模板。

```
dotnet new --search --type item
```

- 搜索所有 C# 模板，并在输出中显示类型和标记。

```
dotnet new --search --language "C#" --columns "type,tags"
```

另请参阅

- dotnet new 命令
- dotnet new --list 选项
- dotnet new 自定义模板

dotnet new --install 选项

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

名称

`dotnet new --install` - 安装模板包。

摘要

```
dotnet new --install <PATH|NUGET_ID>  [--interactive] [--nuget-source <SOURCE>]
```

说明

`dotnet new --install` 命令用于从提供的 `PATH` 或 `NUGET_ID` 安装模板包。若要安装模板包的特定版本或预发布版本，请以 `<package-name>::<package-version>` 格式指定该版本。默认情况下，`dotnet new` 为该版本传递 *，它表示最新的稳定包版本。有关详细信息，请参阅示例部分。

如果在运行此命令时已经安装了模板包的某个版本，则该模板包将更新到指定版本，如果没有指定版本，则将更新到最新的稳定版本。从 .NET SDK 6.0.100 开始，如果 `--install` 选项参数指定了版本，并且该版本的 NuGet 已安装，则不会重新安装。如果参数是 `PATH` 并且它已安装，则不会重新安装。若要了解如何创建自定义模板，请参阅 dotnet new 自定义模板。

在 .NET SDK 6.0.100 之前的版本中，每个.NET SDK 版本(包括修补程序版本)的模板包都是单独管理的。例如，如果在 .NET SDK 5.0.100 中使用 `dotnet new --install` 安装模板包，则只会为 .NET SDK 5.0.100 安装该模板包。包中的模板将不可用于计算机上安装的其他 .NET SDK 版本。

从 .NET SDK 6.0.100 开始，安装的模板包将可用于计算机上安装的更高版本的 .NET SDK。.NET SDK 6.0.100 中安装的模板包也可用于 .NET SDK 6.0.101、.NET SDK 6.0.200 等。不过，这些模板包不可用于 .NET SDK 6.0.100 之前的 .NET SDK 版本。若要在早期的 .NET SDK 版本中使用 .NET SDK 6.0.100 或更高版本安装的模板包，则需通过该 .NET SDK 版本使用 `dotnet new --install` 安装该模板包。

选项

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET 5.0 SDK 起可用。

- `--nuget-source <SOURCE>`

  默认情况下，`dotnet new --install` 使用当前目录中的 NuGet 配置文件的层次结构来确定可从中安装包的 NuGet 源。如果指定了 `--nuget-source`，则会将源添加到要检查的源的列表中。
  若要检查当前目录的配置源，请使用 `dotnet nuget list source`。有关详细信息，请参阅常见的 NuGet 配置

示例

- 安装 ASP.NET Core 的 SPA 模板的最新版本：

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates
```

- 安装 ASP.NET Core 的 SPA 模板 2.0 版：

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0
```

- 使用交互模式从自定义 NuGet 源安装 ASP.NET Core 的 SPA 模板 2.0 版：

```
dotnet new --install Microsoft.DotNet.Web.Spa.ProjectTemplates::2.0.0 --nuget-source "https://api.my-custom-nuget.com/v3/index.json" --interactive
```

另请参阅

- dotnet new 命令
- dotnet new --search 选项
- dotnet new 自定义模板

dotnet new --uninstall 选项

2021/11/16 •

本文适用于：✔ .NET Core 2.0 SDK 及更高版本

名称

`dotnet new --uninstall` - 卸载模板包。

摘要

```
dotnet new --uninstall <PATH|NUGET_ID>
```

说明

`dotnet new --uninstall` 命令在提供的 `PATH` 或 `NUGET_ID` 中卸载模板包。如果未指定 `<PATH|NUGET_ID>` 值，将显示所有当前安装的模板包及其关联的模板。指定 `NUGET_ID` 时，请勿包含版本号。

示例

- 列出已安装的模板及其详细信息，包括如何卸载它们：

  ```
  dotnet new --uninstall
  ```

- 卸载 ASP.NET Core 的 SPA 模板：

  ```
  dotnet new --uninstall Microsoft.DotNet.Web.Spa.ProjectTemplates
  ```

另请参阅

- dotnet new 命令
- dotnet new --list 选项
- dotnet new --search 选项
- dotnet new 自定义模板

dotnet new --update-check 和 --update-apply 选项

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

“属性”

`dotnet new --update-check` 检查已安装的模板包的可用更新。

`dotnet new --update-apply` 将更新应用于已安装的模板包。

摘要

```
dotnet new --update-check

dotnet new --update-apply
```

说明

`dotnet new --update-check` 选项用于检查是否有可用于当前安装的模板包的更新。`dotnet new --update-apply` 选项用于检查是否有可用于当前安装的模板包的更新并安装这些更新。

另请参阅

- dotnet new 命令
- dotnet new --search 选项
- dotnet new --install 选项
- dotnet new 自定义模板

适用于 dotnet new 的 .NET 默认模板

2021/11/16 •

安装 .NET SDK 时，将收到十多个用于创建项目和文件的内置模板，包括控制台应用、类库、单元测试项目、ASP.NET Core 应用（包括 Angular 和 React 项目）和配置文件。若要列出内置模板，请运行带有 `-l|--list` 选项的 `dotnet new` 命令：

```
dotnet new --list
```

下表显示随 .NET SDK 一起预安装的模板。模板的默认语言显示在括号内。单击短名称链接可查看特定的模板选项。

控制台应用程序	`console`	[C#]、F#、VB	常用/控制台	1.0
类库	`classlib`	[C#]、F#、VB	常用/库	1.0
WPF 应用程序	`wpf`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 类库	`wpflib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 自定义控件库	`wpfcustomcontrollib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
WPF 用户控件库	`wpfusercontrollib`	[C#]、VB	常用/WPF	3.0（对于 VB，则为 5.0）
Windows 窗体 (WinForms) 应用程序	`winforms`	[C#]、VB	常用/WinForms	3.0（对于 VB，则为 5.0）
Windows 窗体 (WinForms) 类库	`winformslib`	[C#]、VB	常用/WinForms	3.0（对于 VB，则为 5.0）
Worker Service	`worker`	[C#]	常用/Worker/Web	3.0
单元测试项目	`mstest`	[C#]、F#、VB	测试/MSTest	1.0
NUnit 3 测试项目	`nunit`	[C#]、F#、VB	测试/NUnit	2.1.400
NUnit 3 测试项	`nunit-test`	[C#]、F#、VB	测试/NUnit	2.2
xUnit 测试项目	`xunit`	[C#]、F#、VB	测试/xUnit	1.0
Razor 组件	`razorcomponent`	[C#]	Web/ASP.NET	3.0
Razor 页	`page`	[C#]	Web/ASP.NET	2.0

[illegible]	[illegible]	[illegible]	TAGS	[illegible]
MVC ViewImports	`viewimports`	[C#]	Web/ASP.NET	2.0
MVC ViewStart	`viewstart`	[C#]	Web/ASP.NET	2.0
Blazor 服务器应用	`blazorserver`	[C#]	Web/Blazor	3.0
Blazor WebAssembly 应用	`blazorwasm`	[C#]	Web/Blazor/WebAssembly	3.1.300
ASP.NET Core 空	`web`	[C#], F#	Web/空	1.0
ASP.NET Core Web 应用程序 (Model-View-Controller)	`mvc`	[C#], F#	Web/MVC	1.0
ASP.NET Core Web 应用程序	`webapp, razor`	[C#]	Web/MVC/Razor Pages	2.2、2.0
含 Angular 的 ASP.NET Core	`angular`	[C#]	Web/MVC/SPA	2.0
含 React.js 的 ASP.NET Core	`react`	[C#]	Web/MVC/SPA	2.0
含 React.js 和 Redux 的 ASP.NET Core	`reactredux`	[C#]	Web/MVC/SPA	2.0
Razor 类库	`razorclasslib`	[C#]	Web/Razor/库/Razor 类库	2.1
ASP.NET Core Web API	`webapi`	[C#], F#	Web/WebAPI	1.0
ASP.NET Core gRPC 服务	`grpc`	[C#]	Web/gRPC	3.0
dotnet gitignore 文件	`gitignore`		配置	3.0
global.json 文件	`globaljson`		配置	2.0
NuGet 配置	`nugetconfig`		配置	1.0
Dotnet 本地工具清单文件	`tool-manifest`		配置	3.0
Web 配置	`webconfig`		配置	1.0
解决方案文件	`sln`		解决方案	1.0
协议缓冲区文件	`proto`		Web/gRPC	3.0

EditorConfig 文件	`editorconfig` (#editorconfig)		Config	6.0

模板选项

每个模板都可能有附加选项。核心模板有以下附加选项：

`console`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。自 .NET Core 3.0 SDK 起可用。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK	
6.0	`net6.0`
5.0	`net5.0`
3.1	`netcoreapp3.1`

  能否为较早的 TFM 创建项目取决于是否安装了该版本的 SDK。例如，如果仅安装了 SDK 5.0，则 `--framework` 唯一可用的值为 `net5.0`。如果安装了 SDK 3.1，值 `netcoreapp3.1` 将可用于 `--framework`。如果安装了 SDK 2.1，`netcoreapp2.1` 将可用，以此类推。因此，通过指定 `--framework netcoreapp2.1`，即使在 SDK 5.0 中运行 `dotnet new` 时，也可使用 SDK 2.1。

  或者，若要创建面向早于你使用的 SDK 的框架的项目，可以通过为模板安装 NuGet 包来实现。Common、web 和 SPA 项目类型根据目标框架名字对象 (TFM) 使用不同的包。例如，若要创建面向 `netcoreapp1.0` 的 `console` 项目，请在 `Microsoft.DotNet.Common.ProjectTemplates.1.x` 上运行 `dotnet new --install`。

- `--langVersion <VERSION_NUMBER>`

  在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。不支持 F#。自 .NET Core 2.2 SDK 起可用。

  有关默认的 C# 版本列表，请参阅默认。

- `--no-restore`

  如已指定，则在项目创建期间不执行隐式还原。自 .NET Core 2.2 SDK 起可用。

`classlib`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。值：`net6.0`、`net5.0` 或 `netcoreapp3.1`（若要创建 .NET 类库），或 `netstandard<version>`（若要创建 .NET Standard 类库）。.NET 6 SDK 的默认值是 `net6.0`。

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--langVersion <VERSION_NUMBER>`

  在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。不支持 F#。自 .NET Core 2.2 SDK 起可用。

  有关默认的 C# 版本列表，请参阅默认。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`wpf`, `wpflib`, `wpfcustomcontrollib`, `wpfusercontrollib`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。默认值为 `net5.0`。自 .NET Core 3.1 SDK 起可用。

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--langVersion <VERSION_NUMBER>`

  在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。

  有关默认的 C# 版本列表，请参阅默认。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`winforms`, `winformslib`

- `--langVersion <VERSION_NUMBER>`

  在已创建的项目文件中设置 `LangVersion` 属性。例如，使用 `--langVersion 7.3` 以使用 C# 7.3。

  有关默认的 C# 版本列表，请参阅默认。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`worker`, `grpc`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。默认值为 `netcoreapp3.1`。自 .NET Core 3.1 SDK 起可用。

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`mstest`, `xunit`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。自 .NET Core 3.0 SDK 起可用的选项。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `-p|--enable-pack`

  允许使用 dotnet pack 为项目打包。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`nunit`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`
2.2	`netcoreapp2.2`
2.1	`netcoreapp2.1`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `-p|--enable-pack`

  允许使用 dotnet pack 为项目打包。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`page`

- `-na|--namespace <NAMESPACE_NAME>`

  已生成代码的命名空间。默认值为 `MyApp.Namespace`。

- `-np|--no-pagemodel`

  创建不含 PageModel 的页。

`viewimports`, `proto`

- `-na|--namespace <NAMESPACE_NAME>`

  已生成代码的命名空间。默认值为 `MyApp.Namespace`。

`blazorserver`

- `-au|--auth <AUTHENTICATION_TYPE>`

  要使用的身份验证类型。可能的值为:

  - `None` - 不进行身份验证(默认)。
  - `Individual` - 个人身份验证。
  - `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
  - `SingleOrg` - 对一个租户进行组织身份验证。
  - `MultiOrg` - 对多个租户进行组织身份验证。
  - `Windows` - Windows 身份验证。

- `--aad-b2c-instance <INSTANCE>`

  要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。

- `-ssp|--susi-policy-id <ID>`

  此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-rp|--reset-password-policy-id <ID>`

  此项目的重置密码策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `-ep|--edit-profile-policy-id <ID>`

  此项目的编辑配置文件策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `--aad-instance <INSTANCE>`

  要连接到的 Azure Active Directory 实例。与 `SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--client-id <ID>`

  此项目的客户端 ID。与 `IndividualB2C`、`SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-11111111111111111`。

- `--domain <DOMAIN>`

目录租户的域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `qualified.domain.name`。

- `--tenant-id <ID>`

  要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。

- `--callback-path <PATH>`

  重定向 URI 的应用程序基路径中的请求路径。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `/signin-oidc`。

- `-r|--org-read-access`

  允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 或 `MultiOrg` 身份验证。

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `--no-https`

  关闭 HTTPS。此选项仅适用于 `Individual`、`IndividualB2C`、`SingleOrg` 和 `MultiOrg` 未用于 `--auth` 的情况。

- `-uld|--use-local-db`

  指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`blazorwasm`

- `-f|--framework <FRAMEWORK>`

  指定目标框架。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `-ho|--hosted`

  包括 Blazor WebAssembly 应用的 ASP.NET Core 主机。

- `-au|--auth <AUTHENTICATION_TYPE>`

  要使用的身份验证类型。可能的值为：

  - `None` - 不进行身份验证(默认)。
  - `Individual` - 个人身份验证。
  - `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
  - `SingleOrg` - 对一个租户进行组织身份验证。
- `--authority <AUTHORITY>`

  OIDC 提供程序所属的机构。与 `Individual` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--aad-b2c-instance <INSTANCE>`

  要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://aadB2CInstance.b2clogin.com/`。

- `-ssp|--susi-policy-id <ID>`

  此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `--aad-instance <INSTANCE>`

  要连接到的 Azure Active Directory 实例。与 `SingleOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--client-id <ID>`

  此项目的客户端 ID。在独立方案中与 `IndividualB2C`、`SingleOrg` 或 `Individual` 身份验证一起使用。默认值为 `33333333-3333-3333-33333333333333333`。

- `--domain <DOMAIN>`

  目录租户的域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `qualified.domain.name`。

- `--app-id-uri <URI>`

  要调用的服务器 API 的应用 ID URI。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `api.id.uri`。

- `--api-client-id <ID>`

  服务器承载的 API 的客户端 ID。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `11111111-1111-1111-11111111111111111`。

- `-s|--default-scope <SCOPE>`

  客户端为预配访问令牌所需请求的 API 作用域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `user_impersonation`。

- `--tenant-id <ID>`

  要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。

- `-r|--org-read-access`

  允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 身份验证。

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `-p|--pwa`

  生成支持安装和脱机使用的渐进式 Web 应用程序 (PWA)。

- `--no-https`

  关闭 HTTPS。此选项仅适用于 `Individual`、`IndividualB2C` 和 `SingleOrg` 未用于 `--auth` 的情况。

- `-uld|--use-local-db`

  指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。

- `--called-api-url <URL>`

  要从 Web 应用调用的 API 的 URL。仅适用于未指定 ASP.NET Core 主机的 `SingleOrg` 或 `IndividualB2C` 身份验证。默认值为 `https://graph.microsoft.com/v1.0/me`。

- `--calls-graph`

  指定 Web 应用是否调用 Microsoft Graph。仅适用于 `SingleOrg` 身份验证。

- `--called-api-scopes <SCOPES>`

  为从 Web 应用调用 API 而请求的作用域。仅适用于未指定 ASP.NET Core 主机的 `SingleOrg` 或 `IndividualB2C` 身份验证。默认值为 `user.read`。

---

web

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `-f|--framework <FRAMEWORK>`

  指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`
2.1	`netcoreapp2.1`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `--no-https`

  关闭 HTTPS。

`mvc`, `webapp`

- `-au|--auth <AUTHENTICATION_TYPE>`

  要使用的身份验证类型。可能的值为:

  - `None` - 不进行身份验证(默认)。
  - `Individual` - 个人身份验证。
  - `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
  - `SingleOrg` - 对一个租户进行组织身份验证。
  - `MultiOrg` - 对多个租户进行组织身份验证。
  - `Windows` - Windows 身份验证。
- `--aad-b2c-instance <INSTANCE>`

  要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。
- `-ssp|--susi-policy-id <ID>`

  此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。
- `-rp|--reset-password-policy-id <ID>`

  此项目的重置密码策略 ID。与 `IndividualB2C` 身份验证结合使用。
- `-ep|--edit-profile-policy-id <ID>`

  此项目的编辑配置文件策略 ID。与 `IndividualB2C` 身份验证结合使用。
- `--aad-instance <INSTANCE>`

  要连接到的 Azure Active Directory 实例。与 `SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。
- `--client-id <ID>`

  此项目的客户端 ID。与 `IndividualB2C`、`SingleOrg` 或 `MultiOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-11111111111111111`。
- `--domain <DOMAIN>`

  目录租户的域。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `qualified.domain.name`。
- `--tenant-id <ID>`

  要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。
- `--callback-path <PATH>`

  重定向 URI 的应用程序基路径中的请求路径。与 `SingleOrg` 或 `IndividualB2C` 身份验证结合使用。默认值为 `/signin-oidc`。
- `-r|--org-read-access`

  允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 或 `MultiOrg` 身份验证。
- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `--no-https`

  关闭 HTTPS。此选项仅适用于未使用 `Individual`、`IndividualB2C`、`SingleOrg` 和 `MultiOrg` 的情况。

- `-uld|--use-local-db`

  指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。

- `-f|--framework <FRAMEWORK>`

  指定目标框架。自 .NET Core 3.0 SDK 起可用的选项。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `--use-browserlink`

  在项目中添加 BrowserLink。选项在 .NET Core 2.2 和 3.1 SDK 中不可用。

- `-rrc|--razor-runtime-compilation`

  确定项目是否配置为在调试生成中使用 Razor 运行时编译。自 .NET Core 3.1.201 SDK 起可用的选项。

`angular`, `react`

- `-au|--auth <AUTHENTICATION_TYPE>`

  要使用的身份验证类型。自 .NET Core 3.0 SDK 起可用。

  可能的值为：

  - `None` - 不进行身份验证（默认）。
  - `Individual` - 个人身份验证。

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `--no-https`

  关闭 HTTPS。仅当身份验证为 `None` 时，此选项才适用。

- `-uld|--use-local-db`

指定应使用 LocalDB，而不使用 SQLite。仅适用于 `Individual` 或 `IndividualB2C` 身份验证。自 .NET Core 3.0 SDK 起可用。

- `-f|--framework <FRAMEWORK>`

  指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`
2.1	`netcoreapp2.0`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

`reactredux`

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `-f|--framework <FRAMEWORK>`

  指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`
2.1	`netcoreapp2.0`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `--no-https`

  关闭 HTTPS。

`razorclasslib`

- `--no-restore`

  在项目创建期间不执行隐式还原。

- `-s|--support-pages-and-views`

  除了将组件添加到此库以外，还支持添加传统的 Razor 页面和视图。自 .NET Core 3.0 SDK 起可用。

`webapi`

- `-au|--auth <AUTHENTICATION_TYPE>`

  要使用的身份验证类型。可能的值为：

  - `None` - 不进行身份验证（默认）。
  - `IndividualB2C` - 使用 Azure AD B2C 进行个人身份验证。
  - `SingleOrg` - 对一个租户进行组织身份验证。
  - `Windows` - Windows 身份验证。

- `--aad-b2c-instance <INSTANCE>`

  要连接到的 Azure Active Directory B2C 实例。与 `IndividualB2C` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/tfp/`。

- `-ssp|--susi-policy-id <ID>`

  此项目的登录和注册策略 ID。与 `IndividualB2C` 身份验证结合使用。

- `--aad-instance <INSTANCE>`

  要连接到的 Azure Active Directory 实例。与 `SingleOrg` 身份验证结合使用。默认值为 `https://login.microsoftonline.com/`。

- `--client-id <ID>`

  此项目的客户端 ID。与 `IndividualB2C` 或 `SingleOrg` 身份验证结合使用。默认值为 `11111111-1111-1111-11111111111111111`。

- `--domain <DOMAIN>`

  目录租户的域。与 `IndividualB2C` 或 `SingleOrg` 身份验证结合使用。默认值为 `qualified.domain.name`。

- `--tenant-id <ID>`

  要连接到的目录的 TenantId ID。与 `SingleOrg` 身份验证结合使用。默认值为 `22222222-2222-2222-2222-222222222222`。

- `-r|--org-read-access`

  允许此应用程序对目录进行读取访问。仅适用于 `SingleOrg` 身份验证。

- `--exclude-launch-settings`

  从生成的模板中排除 launchSettings.json。

- `--no-https`

  关闭 HTTPS。`app.UseHsts` 和 `app.UseHttpsRedirection` 未添加到 `Startup.Configure` 中。此选项仅适用于 `IndividualB2C` 或 `SingleOrg` 未用于身份验证的情况。

- `-uld|--use-local-db`

指定应使用 LocalDB，而不使用 SQLite。仅适用于 `IndividualB2C` 身份验证。

- `-f|--framework <FRAMEWORK>`

  指定目标框架。选项在 .NET Core 2.2 SDK 中不可用。

  下表根据所使用的 SDK 版本号列出了默认值：

SDK 版本	框架
5.0	`net5.0`
3.1	`netcoreapp3.1`
3.0	`netcoreapp3.0`
2.1	`netcoreapp2.1`

  要创建一个面向你使用的 SDK 之前的框架的项目，请参阅本文前面部分 `console` 项目的 `--framework`。

- `--no-restore`

  在项目创建期间不执行隐式还原。

`globaljson`

- `--sdk-version <VERSION_NUMBER>`

  指定要在 global.json 文件中使用的 .NET SDK 版本。

`editorconfig`

创建用于配置代码样式首选项的 .editorconfig 文件。

- `--empty`

  创建空的 .editorconfig 而非 .NET 的默认值。

另请参阅

- dotnet new 命令
- dotnet new --list 选项
- dotnet new 自定义模板
- 创建 dotnet new 自定义模板
- 隐式 using 指令

dotnet new 自定义模板

2021/11/16 •

.NET SDK 随附了许多已安装且可供使用的模板。`dotnet new` 命令不仅用于使用模板，还用于说明如何安装和卸载模板。可以为任何类型的项目（如应用、服务、工具或类库）创建自己的自定义模板。甚至可以创建输出一个或多个独立文件（如配置文件）的模板。

可以从任何 NuGet 源上的 NuGet 包安装自定义模板，具体方法是直接引用 NuGet .nupkg 文件，或指定包含模板的文件系统目录。借助模板引擎提供的功能，可以替换值、包括和排除文件，并在使用模板时执行自定义处理操作。

模板引擎是开放源代码，在线代码存储库位于 GitHub 上的 dotnet/templating。可以使用 `dotnet new --search` 找到更多模板，包括第三方模板。若要详细了解如何创建和使用自定义模板，请参阅如何创建自己的 dotnet new 模板和 dotnet/templating GitHub 存储库 Wiki。

> **NOTE**
>
> dotnet/dotnet-template-samples GitHub 存储库提供了模板示例。尽管这些示例是了解模板工作方式的良好资源，但该存储库已存档且不再维护。这些例子可能已过期且不再有效。

若要按照演示步骤操作并创建模板，请参阅创建 dotnet new 自定义模板教程。

.NET 默认模板

安装 .NET SDK 时，将收到十多个用于创建项目和文件的内置模板，包括控制台应用、类库、单元测试项目、ASP.NET Core 应用（包括 Angular 和 React 项目）和配置文件。若要列出内置模板，请运行带有 `-l|--list` 选项的 `dotnet new` 命令：

```
dotnet new --list
```

Configuration

模板由以下部分组成：

- 源文件和文件夹。
- 配置文件 (template.json)。

源文件和文件夹

源文件和文件夹包含运行 `dotnet new <TEMPLATE>` 命令时用户希望模板引擎使用的任何文件和文件夹。模板引擎旨在将可运行项目用作源代码，以生成项目。这样做有以下几个好处：

- 模板引擎不要求用户将特殊令牌注入项目的源代码。
- 代码文件不必是特殊文件，也不必以任何方式进行修改，即可与模板引擎配合使用。因此，处理项目时通常使用的工具也适用于模板内容。
- 生成、运行和调试模板项目，就像生成、运行和调试其他任何项目一样。
- 只需将 ./.template.config/template.json 配置文件添加到项目，即可通过现有项目快速创建模板。

模板中存储的文件和文件夹并不限于正式的 .NET 项目类型。源文件和文件夹可能包含用户希望在使用模板时创建的任何内容，即使模板引擎仅生成一个文件作为输出也不例外。

可以基于在 template.json 配置文件中提供的逻辑和设置对模板生成的文件进行修改。用户可以将选项传递到

`dotnet new <TEMPLATE>` 命令以覆盖这些设置。自定义逻辑的一个常见示例为模板部署的代码文件中的类或变量提供名称。

template.json

template.json 文件位于模板根目录中的 .template.config 文件夹。此文件向模板引擎提供配置信息。最低配置必须包含下表中列出的成员，这足以创建功能模板。

`$schema`	URI	template.json 文件的 JSON 架构。如果指定架构，支持 JSON 架构的编辑器启用 JSON 编辑功能。例如，Visual Studio Code 要求此成员启用 IntelliSense。使用值 `http://json.schemastore.org/template`。
`author`	string	模板创建者。
`classifications`	array(string)	为了找到模板，用户可能会在搜索模板时使用的 0 个或多个模板特征。如果出现在使用 `dotnet new -l|--list` 命令生成的模板列表中，classifications 还会出现在“Tags”列中。
`identity`	string	此模板的唯一名称。
`name`	string	用户应看到的模板名称。
`shortName`	string	方便用户选择模板的默认速记名称，适用于模板名称由用户指定(而不是通过 GUI 选择)的环境。例如，通过命令提示符和 CLI 命令使用模板时，短名称非常有用。
`sourceName`	字符串	源树中的名称，它即将替换为用户指定的名称。模板引擎将查找配置文件中提及并出现的任何 `sourceName`，并将其替换为文件名和文件内容。可以在运行模板时使用 `-n` 或 `--name` 选项提供要替换的值。如果未指定名称，则使用的是当前目录。
`preferNameDirectory`	boolean	(如果指定了名称，但未设置输出目录)指示是否为模板创建目录(而不是直接在当前目录中创建内容)。默认值是 False。

template.json 文件的完整架构位于 JSON 架构存储。有关 template.json 文件的详细信息，请参阅 dotnet 创建模板 wiki。

示例

例如，下面是包含两个内容文件的模板文件夹：console.cs 和 readme.txt。请注意，其中有包含 template.json 文件的名为 .template.config 的所需文件夹。

```
└──mytemplate
    │   console.cs
    │   readme.txt
    │
    └──.template.config
           template.json
```

template.json 文件如下所示：

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Travis Chau",
  "classifications": [ "Common", "Console" ],
  "identity": "AdatumCorporation.ConsoleTemplate.CSharp",
  "name": "Adatum Corporation Console Application",
  "shortName": "adatumconsole"
}
```

mytemplate 文件夹是可安装的模板包。安装此包后，`shortName` 可与 `dotnet new` 命令结合使用。例如，`dotnet new adatumconsole` 会将 `console.cs` 和 `readme.txt` 文件输出到当前文件夹。

将模板打包到 NuGet 包（nupkg 文件）

自定义模板与 dotnet pack 命令和 .csproj 文件一起打包。或者，NuGet 可与 nuget pack 命令以及 .nuspec 文件一起使用。但是，NuGet 在 Windows 上需要 .NET Framework，在 Linux 和 macOS 上需要 Mono。

该 .csproj 文件与传统代码项目 .csproj 文件略有不同。请注意以下设置：

1. 添加 `<PackageType>` 设置并将其设为 `Template`。
2. 添加 `<PackageVersion>` 设置并将其设为有效的 NuGet 版本号。
3. 添加 `<PackageId>` 设置并将其设为唯一标识符。此标识符用于卸载模板包，NuGet 源用它来注册你的模板包。
4. 应设置泛型元数据设置：`<Title>`、`<Authors>`、`<Description>` 和 `<PackageTags>`。
5. 必须设置 `<TargetFramework>` 设置，即使未使用模板过程生成的二进制文件也必须设置。在下面的示例中，它设置为 `netstandard2.0`。

.nupkg NuGet 包形式的模板包要求所有模板都存储在包中的 content 文件夹中 。还有几个设置将添加到 .csproj 文件以确保生成的 .nupkg 作为模板包安装：

1. `<IncludeContentInPack>` 设置设为 `true` 以包含项目在 NuGet 包中设为“内容”的任何文件。
2. `<IncludeBuildOutput>` 设置设为 `false` 以从 NuGet 包排除编译器生成的所有二进制文件。
3. `<ContentTargetFolders>` 设置设为 `content`。这可确保设为“内容”的文件存储在 NuGet 包的 content 文件夹中。NuGet 包中的此文件夹由 dotnet 模板系统解析。

使所有代码文件不被模板项目编译的一个简单的方法是使用 `<ItemGroup>` 元素内项目文件中的 `<Compile Remove="**\*" />` 项。

设置模板包结构的一个简单方法是将所有模板放在单独的文件夹中，然后放到位于 .csproj 文件所在目录的 templates 文件夹的每个模板文件夹中 。这样，你可以使用单个项目项包括 templates 中的所有文件和文件夹作为“内容”。在 `<ItemGroup>` 元素中创建 `<Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />` 项。

下面是一个遵循上述所有准则的示例 .csproj 文件。它将 templates 子文件夹打包为“内容”包文件夹，并使所有代码文件都不被编译。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>
    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

以下示例演示使用 .csproj 创建模板包的文件和文件夹结构。MyDotnetTemplates.csproj 文件和 templates文件夹都位于名为 project_folder 的根目录中。templates 文件夹包含两个模板 mytemplate1 和 mytemplate2。每个模板具有内容文件和包含 template.json 配置文件的 .template.config 文件夹。

```
project_folder
|   MyDotnetTemplates.csproj
|
└───templates
    ├───mytemplate1
    |   |   console.cs
    |   |   readme.txt
    |   |
    |   └───.template.config
    |           template.json
    |
    └───mytemplate2
        |   otherfile.cs
        |
        └───.template.config
                template.json
```

> **NOTE**
> 若要确保模板包显示在 `dotnet new --search` 结果中，请将 NuGet 包类型设置为 `Template`。

安装模板包

使用 dotnet new --install 命令安装模板包。

从 nuget.org 中存储的 NuGet 包安装模板包的具体步骤

使用 NuGet 包标识符安装模板包。

```
dotnet new --install <NUGET_PACKAGE_ID>
```

从本地 nupkg 文件安装模板包的具体步骤

提供指向 .nupkg NuGet 包文件的路径。

```
dotnet new --install <PATH_TO_NUPKG_FILE>
```

从文件系统目录安装模板包的具体步骤

模板可从模板文件夹安装，如以上示例中的 mytemplate1 文件夹。指定 .template.config 文件夹的文件夹路径。模板目录的路径不需要是绝对路径。

```
dotnet new --install <FILE_SYSTEM_DIRECTORY>
```

获取已安装模板包的列表

在没有任何其他参数的情况下，卸载命令将列出所有已安装的模板包和包含的模板。

```
dotnet new --uninstall
```

该命令返回如下所示的输出：

```
Template Instantiation Commands for .NET CLI

Currently installed items:
  Microsoft.DotNet.Common.ItemTemplates
    Templates:
      global.json file (globaljson)
      NuGet Config (nugetconfig)
      Solution File (sln)
      Dotnet local tool manifest file (tool-manifest)
      Web Config (webconfig)
  Microsoft.DotNet.Common.ProjectTemplates.3.0
    Templates:
      Class library (classlib) C#
      Class library (classlib) F#
      Class library (classlib) VB
      Console Application (console) C#
      Console Application (console) F#
      Console Application (console) VB
...
```

`Currently installed items:` 后面的第一级项是用于卸载模板包的标识符。在上述示例中，列出了 `Microsoft.DotNet.Common.ItemTemplates` 和 `Microsoft.DotNet.Common.ProjectTemplates.3.0`。如果使用文件系统路径安装模板包，此标识符将是 .template.config 文件夹的文件夹路径。

卸载模板包

使用 dotnet new -u|--uninstall 命令卸载模板包。

如果通过 NuGet 源或直接通过 .nupkg 文件安装包，请提供标识符。

```
dotnet new --uninstall <NUGET_PACKAGE_ID>
```

如果通过指定 .template.config 文件夹的路径安装包，请使用该路径卸载包。你可以在 `dotnet new --uninstall` 命令提供的输出中看到模板包的绝对路径。有关详细信息，请参阅上文的获取已安装的模板列表部分。

```
dotnet new --uninstall <FILE_SYSTEM_DIRECTORY>
```

使用自定义模板创建项目

安装模板后，通过执行 `dotnet new <TEMPLATE>` 命令来使用模板，就像使用其他任何预安装模板一样。还可以为 `dotnet new` 命令指定选项，包括在模板设置中配置的模板专用选项。直接向命令提供模板的短名称：

```
dotnet new <TEMPLATE>
```

请参阅

- 创建 dotnet new 自定义模板（教程）
- dotnet/templating GitHub 存储库 Wiki
- dotnet/dotnet-template-samples GitHub 存储库
- 如何创建自己的 dotnet new 模板
- JSON 架构存储中的 template.json 架构

dotnet nuget delete

2021/11/16 •

本文适用于：✔ .NET Core 1.x SDK 及更高版本

名称

`dotnet nuget delete` - 从服务器删除或取消列出包。

摘要

```
dotnet nuget delete [<PACKAGE_NAME> <PACKAGE_VERSION>] [--force-english-output]
    [--interactive] [-k|--api-key <API_KEY>] [--no-service-endpoint]
    [--non-interactive] [-s|--source <SOURCE>]

dotnet nuget delete -h|--help
```

说明

`dotnet nuget delete` 命令从服务器删除或取消列出包。对于 NuGet.org，该操作将取消列出包。

参数

- `PACKAGE_NAME`

  要删除的包的名称/ID。

- `PACKAGE_VERSION`

  要删除的包的版本。

选项

- `--force-english-output`

  使用固定的、基于英语的区域性强制运行应用程序。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `-k|--api-key <API_KEY>`

  服务器的 API 密钥。

- `--no-service-endpoint`

  不将“api/v2/package”追加至源 URL。自 .NET Core 2.1 SDK 起可用的选项。

- `--non-interactive`

不提示用户输入或确认。

- `-s|--source <SOURCE>`

  指定服务器 URL。Nuget.org 的支持 URL 包括 `https://www.nuget.org`、`https://www.nuget.org/api/v3` 和 `https://www.nuget.org/api/v2/package`。对于专用源，请替换主机名（例如，`%hostname%/api/v3`）。

示例

- 删除包 `Microsoft.AspNetCore.Mvc` 的 1.0 版：

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0
```

- 删除包 `Microsoft.AspNetCore.Mvc` 的 1.0 版（不提示用户需要凭据或其他输入）：

```
dotnet nuget delete Microsoft.AspNetCore.Mvc 1.0 --non-interactive
```

dotnet nuget locals

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

名称

`dotnet nuget locals` -清除或列出本地 NuGet 资源。

摘要

```
dotnet nuget locals <CACHE_LOCATION> [(-c|--clear)|(-l|--list)] [--force-english-output]

dotnet nuget locals -h|--help
```

说明

`dotnet nuget locals` 命令清除或列出 http 请求缓存中的本地 NuGet 资源，临时缓存或计算机范围的全局包文件夹。

参数

- `CACHE_LOCATION`

  要列出或清除的缓存位置。可以接受以下值之一：

  - `all` - 表示指定的操作应用于所有缓存类型，即 http 请求缓存、全局包缓存和临时缓存。
  - `http-cache` - 表示指定的操作仅应用于 http 请求缓存。其他缓存位置不受影响。
  - `global-packages` - 表示指定的操作仅应用于全局包缓存。其他缓存位置不受影响。
  - `temp` - 表示指定的操作仅应用于临时缓存。其他缓存位置不受影响。

选项

- `--force-english-output`

  使用固定的、基于英语的区域性强制运行应用程序。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-c|--clear`

  清除选项对指定的缓存类型执行清除操作。缓存目录的内容被以递归方式删除。正在执行的用户/组必须具有对缓存目录中的文件的相关权限。反之，则显示错误，指示未清除的文件/文件夹。

- `-l|--list`

  列表选项用于显示指定缓存类型的位置。

示例

- 显示所有本地缓存目录的路径(http 缓存目录、全局包缓存目录和临时缓存目录):

```
dotnet nuget locals all -l
```

- 显示本地 http 缓存录的路径:

```
dotnet nuget locals http-cache --list
```

- 清除所有本地缓存目录的文件(http 缓存目录、全局包缓存目录和临时缓存目录):

```
dotnet nuget locals all --clear
```

- 清除本地全局包缓存目录中的所有文件:

```
dotnet nuget locals global-packages -c
```

- 清除本地临时缓存目录中的所有文件:

```
dotnet nuget locals temp -c
```

疑难解答

有关使用 `dotnet nuget locals` 命令时的常见问题和错误的信息，请参阅管理 NuGet 缓存。

dotnet nuget push

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet nuget push` - 将包推送到服务器，并将其发布。

摘要

```
dotnet nuget push [<ROOT>] [-d|--disable-buffering] [--force-english-output]
    [--interactive] [-k|--api-key <API_KEY>] [-n|--no-symbols]
    [--no-service-endpoint] [-s|--source <SOURCE>] [--skip-duplicate]
    [-sk|--symbol-api-key <API_KEY>] [-ss|--symbol-source <SOURCE>]
    [-t|--timeout <TIMEOUT>]

dotnet nuget push -h|--help
```

描述

`dotnet nuget push` 将包推送到服务器，并将其发布。push 命令使用在系统的 NuGet 配置文件或配置文件链中找到的服务器和凭据详细信息。有关配置文件的详细信息，请参阅 Configuring NuGet Behavior（配置 NuGet 行为）。通过加载 %AppData%\NuGet\NuGet.config (Windows) 或 $HOME/.nuget/NuGet/NuGet.Config (Linux/macOS) 获得 NuGet 的默认配置，然后加载任意 nuget.config 或 .nuget\nuget.config，从驱动器的根目录开始，并在当前目录中结束 。

命令推送现有包。它不会创建包。若要创建包，请使用 `dotnet pack` 。

自变量

- `ROOT`

  指定要推送的包的文件路径。

选项

- `-d|--disable-buffering`

  当推送到 HTTP(S) 服务器以减少内存使用率时，禁用缓冲。

- `--force-english-output`

  使用固定的、基于英语的区域性强制运行应用程序。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `-k|--api-key <API_KEY>`

  服务器的 API 密钥。

- `-n|--no-symbols`

  不推送符号(即使存在)。

- `--no-service-endpoint`

  不将"api/v2/package"追加至源 URL。自 .NET Core 2.1 SDK 起可用的选项。

- `-s|--source <SOURCE>`

  指定服务器 URL。NuGet 标识 UNC 或本地文件夹源，只在其中复制文件，而不会使用 HTTP 进行推送。

  > **IMPORTANT**
  > 从 NuGet 3.4.2 开始，此参数为必需，除非 NuGet 配置文件指定了 `DefaultPushSource` 值。有关详细信息，请参阅配置 NuGet 行为。

- `--skip-duplicate`

  将多个包推送到 HTTP(S) 服务器时，将任何 409 冲突响应视为警告，以便可以继续推送。自 .NET Core 3.1 SDK 起可用。

- `-sk|--symbol-api-key <API_KEY>`

  符号服务器的 API 密钥。

- `-ss|--symbol-source <SOURCE>`

  指定符号服务器 URL。

- `-t|--timeout <TIMEOUT>`

  指定推送到服务器的超时(秒)。默认值为 300 秒(5 分钟)。指定 0 时应用默认值。

示例

- 将 foo.nupkg 推送到 NuGet 配置文件中指定的默认推送源(使用 API 密钥):

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a
```

- 将 foo.nupkg 推送到官方 NuGet 服务器，以指定 API 密钥:

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s
https://api.nuget.org/v3/index.json
```

  - 将 foo.nupkg 推送到自定义推送源 `https://customsource`(指定 API 密钥):

```
dotnet nuget push foo.nupkg -k 4003d786-cc37-4004-bfdf-c4f3e8ef9b3a -s https://customsource/
```

- 将 foo.nupkg 推送到 NuGet 配置文件中指定的默认推送源:

```
dotnet nuget push foo.nupkg
```

- 将 foo.symbols.nupkg 推送到默认符号源：

```
dotnet nuget push foo.symbols.nupkg
```

- 将 foo.nupkg 推送到 NuGet 配置文件中指定的默认推送源（超时 360 秒）：

```
dotnet nuget push foo.nupkg --timeout 360
```

- 将当前目录中的所有 .nupkg 文件推送到 NuGet 配置文件中指定的默认推送源：

```
dotnet nuget push "*.nupkg"
```

> **NOTE**
> 如果此命令不起作用，则可能是较旧版本的 SDK（.NET Core 2.1 SDK 及更早版本）中的 bug 导致的。要解决此问题，请升级 SDK 版本或改为运行以下命令：`dotnet nuget push "**/*.nupkg"`

> **NOTE**
> 用于执行文件组合的 bash 等 shell 需要用引号括起来。有关详细信息，请参阅 NuGet/Home#4393。

- 将所有 .nupkg 文件推送到 NuGet 配置文件中指定的默认推送源，即使 HTTP(S) 服务器返回“409 Conflict”响应也是如此：

```
dotnet nuget push "*.nupkg" --skip-duplicate
```

- 将当前目录中的所有 .nupkg 文件推送到本地源目录：

```
dotnet nuget push "*.nupkg" -s c:\mydir
```

此命令不会将包存储在分层文件夹结构中，因此建议优化性能。有关详细信息，请参阅本地源。

dotnet nuget add source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget add source` - 添加 NuGet 源。

摘要

```
dotnet nuget add source <PACKAGE_SOURCE_PATH> [--name <SOURCE_NAME>] [--username <USER>]
    [--password <PASSWORD>] [--store-password-in-clear-text]
    [--valid-authentication-types <TYPES>] [--configfile <FILE>]

dotnet nuget add source -h|--help
```

描述

`dotnet nuget add source` 命令将新的包源添加到 NuGet 配置文件中。

> **WARNING**
> 添加多个包源时，请注意不要引入依赖关系混乱漏洞。

自变量

- `PACKAGE_SOURCE_PATH`

  包源的路径。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-n|--name <SOURCE_NAME>`

  源的名称。

- `-p|--password <PASSWORD>`

  连接到已验证源时要使用的密码。

- `--store-password-in-clear-text`

  通过禁用密码加密允许存储可移植包源凭据。

- `-u|--username <USER>`

  连接到已经过身份验证的源时要使用的用户名。

- `--valid-authentication-types <TYPES>`

  此源的有效身份验证类型的逗号分隔列表。如果服务器公布 NTLM 或协商，并且你必须使用基本机制发送凭据（例如，在本地 Azure DevOps Server 中使用 PAT 时），则将此项设置为 `basic`。其他有效值包括 `negotiate`、`kerberos`、`ntlm` 和 `digest`，但这些值不太可能有用。

示例

- 将 `nuget.org` 添加为源：

```
dotnet nuget add source https://api.nuget.org/v3/index.json -n nuget.org
```

- 将 `c:\packages` 添加为本地源：

```
dotnet nuget add source c:\packages
```

- 添加需要身份验证的源：

```
dotnet nuget add source https://someServer/myTeam -n myTeam -u myUsername -p myPassword --store-password-in-clear-text
```

- 添加需要身份验证的源（然后继续安装凭据提供程序）：

```
dotnet nuget add source https://azureartifacts.microsoft.com/myTeam -n myTeam
```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget disable source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget disable source` - 禁用 NuGet 源。

摘要

```
dotnet nuget disable source <NAME> [--configfile <FILE>]

dotnet nuget disable source -h|--help
```

描述

`dotnet nuget disable source` 命令在 NuGet 配置文件中禁用现有源。

自变量

- `NAME`

  源的名称。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

示例

- 禁用名为 `mySource` 的源：

  ```
  dotnet nuget disable source mySource
  ```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget enable source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget enable source` - 启用 NuGet 源。

摘要

```
dotnet nuget enable source <NAME> [--configfile <FILE>]

dotnet nuget enable source -h|--help
```

描述

`dotnet nuget enable source` 命令在 NuGet 配置文件中启用现有源。

自变量

- `NAME`

  源的名称。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

示例

- 启用名为 `mySource` 的源：

  ```
  dotnet nuget enable source mySource
  ```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget list source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget list source` - 列出所有配置的 NuGet 源。

摘要

```
dotnet nuget list source [--format [Detailed|Short]] [--configfile <FILE>]

dotnet nuget list source -h|--help
```

描述

`dotnet nuget list source` 命令列出 NuGet 配置文件中的所有现有源。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--format [Detailed|Short]`

  list 命令输出的格式为：`Detailed`（默认值）和 `Short`。

示例

- 列出当前目录中的已配置源：

  ```
  dotnet nuget list source
  ```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget remove source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget remove source` - 删除 NuGet 源。

摘要

```
dotnet nuget remove source <NAME> [--configfile <FILE>]

dotnet nuget remove source -h|--help
```

描述

`dotnet nuget remove source` 命令从 NuGet 配置文件中删除现有源。

自变量

- `NAME`

  源的名称。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

示例

- 删除名为 `mySource` 的源：

  ```
  dotnet nuget remove source mySource
  ```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget update source

2021/11/16 •

本文适用于： ✔ .NET Core 3.1.200 SDK 及更高版本

“属性”

`dotnet nuget update source` - 更新 NuGet 源。

摘要

```
dotnet nuget update source <NAME> [--source <SOURCE>] [--username <USER>]
    [--password <PASSWORD>] [--store-password-in-clear-text]
    [--valid-authentication-types <TYPES>] [--configfile <FILE>]

dotnet nuget update source -h|--help
```

描述

`dotnet nuget update source` 命令在 NuGet 配置文件中更新现有源。

自变量

- `NAME`

  源的名称。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-p|--password <PASSWORD>`

  连接到已验证源时要使用的密码。

- `-s|--source <SOURCE>`

  包源的路径。

- `--store-password-in-clear-text`

  通过禁用密码加密允许存储可移植包源凭据。

- `-u|--username <USER>`

  连接到已经过身份验证的源时要使用的用户名。

- `--valid-authentication-types <TYPES>`

  此源的有效身份验证类型的逗号分隔列表。如果服务器公布 NTLM 或协商，并且你必须使用基本机制发送凭据（例如，在本地 Azure DevOps Server 中使用 PAT 时），则将此项设置为 `basic`。其他有效值包括

`negotiate`、`kerberos`、`ntlm` 和 `digest`，但这些值不太可能有用。

示例

- 更新名为 `mySource` 的源：

```
dotnet nuget update source mySource --source c:\packages
```

请参阅

- NuGet.config 文件中的包源部分
- sources 命令 (nuget.exe)

dotnet nuget verify

2021/11/16 •

本文适用于： ✔ .NET 5.0.100-rc.2.x SDK 及更高版本

名称

`dotnet nuget verify` -验证已签名的 NuGet 包。

摘要

```
dotnet nuget verify [<package-path(s)>]
    [--all]
    [--certificate-fingerprint <FINGERPRINT>]
    [-v|--verbosity <LEVEL>]

dotnet nuget verify -h|--help
```

描述

`dotnet nuget verify` 命令验证已签名的 NuGet 包。

参数

- `package-path(s)`

  指定要验证的包的文件路径。可以传入多个位置参数以验证多个包。

选项

- `--all`

  指定应对包执行的所有可能的验证。默认情况下，只验证 `signatures`。

> **NOTE**
> 此命令当前仅支持 `signature` 验证。

- `--certificate-fingerprint <FINGERPRINT>`

  验证签名者证书是否与指定的其中一个 `SHA256` 指纹匹配。可以多次提供此选项以提供多个指纹。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

  下表显示了针对每种详细级别显示的内容。

```
                                | `q[uiet]` | `m[inimal]` | `n[ormal]` | `d[etailed]` | `diag[nostic]`
```

---|---|---|---|---|---
`Certificate chain Information` | ✗ | ✗ | ✗ | ✓ | ✓ `Path to package being verified` | ✗ | ✗ | ✓ | ✓ | ✓ `Hashing algorithm used for signature` | ✗ | ✗ | ✓ | ✓ | ✓
`Author/Repository Certificate -> SHA1 hash` | ✗ | ✗ | ✓ | ✓ | ✓
`Author/Repository Certificate -> Issued By` | ✗ | ✗ | ✓ | ✓ | ✓ `Timestamp Certificate -> Issued By` | ✗ | ✗ | ✓ | ✓ | ✓ `Timestamp Certificate -> SHA-256 hash` | ✗ | ✗ | ✓ | ✓ | ✓
`Timestamp Certificate -> Validity period` | ✗ | ✗ | ✓ | ✓ | ✓ `Timestamp Certificate -> SHA1 hash` | ✗ | ✗ | ✓ | ✓ | ✓ `Timestamp Certificate -> Subject name` | ✗ | ✗ | ✓ | ✓ | ✓
`Author/Repository Certificate -> Subject name` | ✗ | ✓ | ✓ | ✓ | ✓
`Author/Repository Certificate -> SHA-256 hash` | ✗ | ✓ | ✓ | ✓ | ✓
`Author/Repository Certificate -> Validity period` | ✗ | ✓ | ✓ | ✓ | ✓
`Author/Repository Certificate -> Service index URL (If applicable)` | ✗ | ✓ | ✓ | ✓ | ✓
`Package name being verified` | ✗ | ✓ | ✓ | ✓ | ✓ `Type of signature (author or repository)` | ✗ | ✓ | ✓ | ✓ | ✓

✗ 表示未显示的详细信息。✓ 表示已显示的详细信息。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 验证 foo.nupkg：

```
dotnet nuget verify foo.nupkg
```

- 验证多个 NuGet 包 - foo.nupkg 和指定目录中的所有 .nupkg 文件：

```
dotnet nuget verify foo.nupkg c:\mydir\*.nupkg
```

- 验证 foo.nupkg 签名是否与指定的证书指纹匹配：

```
dotnet nuget verify foo.nupkg --certificate-fingerprint
CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039
```

- 验证 foo.nupkg 签名是否与指定的其中一个证书指纹匹配：

```
dotnet nuget verify foo.nupkg --certificate-fingerprint
CE40881FF5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E039 --certificate-fingerprint
EC10992GG5F0AD3E58965DA20A9F571EF1651A56933748E1BF1C99E537C4E027
```

dotnet nuget trust

2021/11/16 •

本文适用于：✔ .NET 5.0.300 SDK 及更高版本

名称

`dotnet nuget trust` - 获取受信任的签名者或将受信任的签名者设置为 NuGet 配置。

摘要

```
dotnet nuget trust [command] [Options]

dotnet nuget trust -h|--help
```

说明

`dotnet nuget trust` 命令用于管理受信任的签名者。默认情况下，NuGet 接受所有作者和存储库。通过这些命令，可仅指定将接受其签名的特定子集的签名者，同时拒绝所有其他签名者。有关详细信息，请参阅常见的 NuGet 配置。有关 nuget.config 架构外观的详细信息，请参阅 NuGet 配置文件引用。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

命令

如果未指定命令，则命令默认为 `list`。

`list`

列出配置中所有受信任的签名者。此选项将包括每个签名者具有的所有证书（带有指纹和指纹算法）。如果证书具有前导 [U]，则意味着证书条目的 allowUntrustedRoot 设置为 true。

摘要：

```
dotnet nuget trust list [--configfile <PATH>] [-h|--help] [-v, --verbosity <LEVEL>]
```

选项：

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`sync`

删除证书的当前列表，并将其替换为存储库中的最新列表。

摘要

```
dotnet nuget trust sync <NAME> [--configfile <PATH>] [-h|--help] [-v, --verbosity <LEVEL>]
```

自变量

- `NAME`

  要同步的现有受信任签名者的名称。

选项：

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`remove`

删除与给定名称匹配的任何受信任的签名者。

摘要

```
dotnet nuget trust remove <NAME> [--configfile <PATH>] [-h|--help] [-v, --verbosity <LEVEL>]
```

自变量

- `NAME`

  要删除的现有受信任签名者的名称。

选项：

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`author`

根据包的作者签名，添加具有给定名称的受信任签名者。

摘要

```
dotnet nuget trust author <NAME> <PACKAGE> [--allow-untrusted-root] [--configfile <PATH>] [-h|--help] [-v, --verbosity <LEVEL>]
```

自变量

- `NAME`

  要添加的受信任签名者的名称。如果配置中已存在 `NAME`，则追加签名。

- `PACKAGE`

  给定的 `PACKAGE` 应为已签名的 .nupkg 文件的本地路径。

选项：

- `--allow-untrusted-root`

  指定是否应允许受信任的签名者的证书链接到不受信任的根。不建议这样做。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`repository`

根据已签名包的存储库签名或副署，添加具有给定名称的受信任签名者。

摘要

```
dotnet nuget trust repository <NAME> <PACKAGE> [--allow-untrusted-root] [--configfile <PATH>] [-h|--help] [--owners <LIST>] [-v, --verbosity <LEVEL>]
```

自变量

- `NAME`

  要添加的受信任签名者的名称。如果配置中已存在 `NAME`，则追加签名。

- `PACKAGE`

  给定的 `PACKAGE` 应为已签名的 .nupkg 文件的本地路径。

选项：

- `--allow-untrusted-root`

  指定是否应允许受信任的签名者的证书链接到不受信任的根。不建议这样做。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

打印出有关如何使用命令的说明。

- `--owners <LIST>`

  受信任所有者的分号分隔列表，用来进一步限制存储库的信任。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`certificate`

基于证书指纹添加具有给定名称的受信任签名者。

摘要

```
dotnet nuget trust certificate <NAME> <FINGERPRINT> [--algorithm <ALGORITHM>] [--allow-untrusted-root] [--
configfile <PATH>] [-h|--help] [-v, --verbosity <LEVEL>]
```

自变量

- `NAME`

  要添加的受信任签名者的名称。如果已存在具有给定名称的受信任签名者，则证书项将添加到该签名者。否则，将使用给定证书信息中的证书项创建受信任作者。

- `FINGERPRINT`

  证书的指纹。

选项：

- `--algorithm <ALGORITHM>`

  指定用于计算证书指纹的哈希算法。默认为 SHA256。支持的值为 SHA256、SHA384 和 SHA512。

- `--allow-untrusted-root`

  指定是否应允许受信任的签名者的证书链接到不受信任的根。不建议这样做。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

`source`

基于给定的包源添加受信任的签名者。

摘要

```
dotnet nuget trust source <NAME> [--configfile <PATH>] [-h|--help] [--owners <LIST>] [--source-url] [-v, --
verbosity <LEVEL>]
```

自变量

- `NAME`

  要添加的受信任签名者的名称。如果仅提供 `<NAME>` 而不提供 `--<source-url>`，则来自 NuGet 配置文件的同名包源将添加到受信任列表中。如果配置中已存在 `<NAME>`，则将包源追加到其中。

选项：

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--owners <LIST>`

  受信任所有者的分号分隔列表，用来进一步限制存储库的信任。

- `--source-url`

  如果提供了 `source-url`，则它必须是 v3 包源 URL（如 `https://api.nuget.org/v3/index.json`）。不支持其他包源类型。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 列出受信任的签名者：

```
dotnet nuget trust list
```

- 信任指定的 nuget.config 文件中的源 NuGet：

```
dotnet nuget trust source NuGet --configfile ..\nuget.config
```

- 信任已签名的 nupkg 包文件 foo.nupkg 中的作者：

```
dotnet nuget trust author PackageAuthor .\foo.nupkg
```

- 信任已签名的 nupkg 包文件 foo.nupkg 中的存储库：

```
dotnet nuget trust repository PackageRepository .\foo.nupkg
```

- 信任使用 SHA256 指纹的包签名证书：

```
  dotnet nuget trust certificate MyCert
F99EC8CDCE5642B380296A19E22FA8EB3AEF1C70079541A2B3D6E4A93F5E1AFD --algorithm SHA256
```

- 信任存储库 `https://api.nuget.org/v3/index.json` 中的所有者 Nuget 和 Microsoft：

```
dotnet nuget trust source NuGetTrust https://api.nuget.org/v3/index.json --owners "Nuget;Microsoft"
```

- 从指定的 nuget.config 文件中删除名为 NuGet 的受信任签名者：

```
dotnet nuget trust remove NuGet --configfile ..\nuget.config
```

dotnet nuget sign

2021/11/16 •

本文适用于：✔ .NET 6 预览版 5 SDK 及更高版本

“属性”

`dotnet nuget sign` - 使用证书对匹配第一个参数的所有 NuGet 包进行签名。

摘要

```
dotnet nuget sign [<package-path(s)>]
    [--certificate-path <PATH>]
    [--certificate-store-name <STORENAME>]
    [--certificate-store-location <STORELOCATION>]
    [--certificate-subject-name <SUBJECTNAME>]
    [--certificate-fingerprint <FINGERPRINT>]
    [--certificate-password <PASSWORD>]
    [--hash-algorithm <HASHALGORITHM>]
    [-o|--output <OUTPUT DIRECTORY>]
    [--overwrite]
    [--timestamp-hash-algorithm <HASHALGORITHM>]
    [--timestamper <TIMESTAMPINGSERVER>]
    [-v|--verbosity <LEVEL>]

dotnet nuget sign -h|--help
```

说明

`dotnet nuget sign` 命令使用证书对匹配第一个参数的所有包进行签名。通过提供使用者名称或 SHA-1 指纹，可以从文件中或证书存储中安装的证书中获取带私钥的证书。

自变量

- `package-path(s)`

  指定要签名的包的文件路径。可以传入多个参数来对多个包签名。

选项

- `--certificate-path <PATH>`

  指定对包进行签名时要使用的证书的文件路径。

  > **NOTE**
  > 此选项当前仅支持包含证书私钥的 `PKCS12 (PFX)` 文件。

- `--certificate-store-name <STORENAME>`

  指定用于搜索证书的 x.509 证书存储的名称。默认为 "My"，即个人证书的 x.509 证书存储。当通过 `--certificate-subject-name` 或 `--certificate-fingerprint` 选项指定证书时，应使用此选项。

- `--certificate-store-location <STORELOCATION>`

  指定用于搜索证书的 x.509 证书存储的名称。默认为 "CurrentUser"，即当前用户使用的 x.509 证书存储。当通过 `--certificate-subject-name` 或 `--certificate-fingerprint` 选项指定证书时，应使用此选项。

- `--certificate-subject-name <SUBJECTNAME>`

  指定用于在本地证书存储中搜索证书的证书使用者名称。该搜索是使用提供的值进行的区分大小写的字符串比较，它将查找使用者名称中包含该字符串的所有证书，且不考虑其他使用者值。可通过 `--certificate-store-name` 和 `--certificate-store-location` 选项指定证书存储。

  > **NOTE**
  > 此选项目前仅支持在结果中显示一个匹配的证书。如果结果中有多个匹配的证书，或者结果中没有匹配的证书，sign 命令将失败。

- `--certificate-fingerprint <FINGERPRINT>`

  指定用于在本地证书存储中搜索证书的证书 SHA-1 指纹。

- `--certificate-password <PASSWORD>`

  如果需要，指定证书密码。如果证书受密码保护，但未提供密码，sign 命令将失败。

  > **NOTE**
  > `sign` 命令仅支持非交互模式。在运行时不会提示输入密码。

- `--hash-algorithm <HASHALGORITHM>`

  用于对包进行签名的哈希算法。默认为 SHA256。可能的值有 SHA256、SHA384 和 SHA512。

- `-o|--output`

  指定用于保存已签名的包的目录。如果未指定此选项，默认情况下，已签名的包将覆盖原始包。

- `--overwrite`

  指示应覆盖当前签名。默认情况下，如果包已有签名，该命令将失败。

- `--timestamp-hash-algorithm <HASHALGORITHM>`

  RFC 3161 时间戳服务器要使用的哈希算法。默认为 SHA256。

- `--timestamper <TIMESTAMPINGSERVER>`

  RFC 3161 时间戳加盖服务器的 URL。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 使用证书 cert.pfx 对 foo.nupkg 进行签名（无密码保护）：

```
dotnet nuget sign foo.nupkg --certificate-path cert.pfx
```

- 使用证书 cert.pfx 对 foo.nupkg 进行签名（有密码保护）：

```
dotnet nuget sign foo.nupkg --certificate-path cert.pfx --certificate-password password
```

- 使用与默认证书存储 (CurrentUser\My) 中指定的 SHA-1 指纹匹配的证书对 foo.nupkg 进行签名（有密码保护）：

```
dotnet nuget sign foo.nupkg --certificate-fingerprint 89967D1DD995010B6C66AE24FF8E66885E6E03A8 --
certificate-password password
```

- 使用与默认证书存储 (CurrentUser\My) 中指定的使用者名称 "Test certificate for testing signing" 匹配的证书对 foo.nupkg 进行签名（有密码保护）：

```
dotnet nuget sign foo.nupkg --certificate-subject-name "Test certificate for testing signing" --
certificate-password password
```

- 使用与证书存储 CurrentUser\Root 中指定的 SHA-1 指纹匹配的证书对 foo.nupkg 进行签名（有密码保护）：

```
dotnet nuget sign foo.nupkg --certificate-fingerprint 89967D1DD995010B6C66AE24FF8E66885E6E03A8 --
certificate-password password --certificate-store-location CurrentUser --certificate-store-name Root
```

- 使用证书 cert.pfx 对多个 NuGet 包进行签名 - foo.nupkg 和指定的目录中所有的 .nupkg 文件（无密码保护）：

```
dotnet nuget sign foo.nupkg c:\mydir\*.nupkg --certificate-path cert.pfx
```

- 使用证书 cert.pfx 对 foo.nupkg 进行签名（有密码保护），并用 `http://timestamp.test` 加盖时间戳：

```
dotnet nuget sign foo.nupkg --certificate-path cert.pfx --certificate-password password --timestamper
http://timestamp.test
```

- 使用证书 cert.pfx 对 foo.nupkg 进行签名（无密码保护），并将签名的包保存在指定目录下：

```
dotnet nuget sign foo.nupkg --certificate-path cert.pfx --output c:\signed\
```

- 使用证书 cert.pfx 对 foo.nupkg 进行签名（无密码保护），如果该包已签名，则覆盖当前签名：

```
dotnet nuget sign foo.nupkg --certificate-path cert.pfx --overwrite
```

dotnet pack

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet pack` - 将代码打包到 NuGet 包。

摘要

```
dotnet pack [<PROJECT>|<SOLUTION>] [-c|--configuration <CONFIGURATION>]
    [--force] [--include-source] [--include-symbols] [--interactive]
    [--no-build] [--no-dependencies] [--no-restore] [--nologo]
    [-o|--output <OUTPUT_DIRECTORY>] [--runtime <RUNTIME_IDENTIFIER>]
    [-s|--serviceable] [-v|--verbosity <LEVEL>]
    [--version-suffix <VERSION_SUFFIX>]

dotnet pack -h|--help
```

描述

`dotnet pack` 命令生成项目并创建 NuGet 包。该命令的结果是一个 NuGet 包，也就是一个 .nupkg 文件。

如果要生成包含调试符号的包，可以使用以下两个选项：

- `--include-symbols`：该选项用于创建符号包。
- `--include-source`：该选项用于创建带有 `src` 文件夹的符号包，该文件夹包含源文件。

将被打包项目的 NuGet 依赖项添加到 *.nuspec* 文件，以便在安装包时可以进行正确解析。如果打包的项目具有对其他项目的引用，则不会将其他项目包含在包中。目前，如果具有项目到项目的依赖项，则每个项目均必须包含一个包。

默认情况下，`dotnet pack` 先构建项目。如果希望避免此行为，则传递 `--no-build` 选项。此选项在持续集成 (CI) 生成方案中通常非常有用，你可以知道代码是之前生成的。

> **NOTE**
> 在某些情况下，无法执行隐式生成。设置 `GeneratePackageOnBuild` 以避免生成目标和包目标之间的循环依赖关系时可能会发生这种情况。如果存在锁定文件或其他问题，生成也可能失败。

可向 `dotnet pack` 命令提供 MSBuild 属性，用于打包进程。有关详细信息，请参阅 NuGet 包目标属性和 MSBuild 命令行引用。示例部分介绍了如何在不同的情况下使用 MSBuild `-p` 开关。

默认情况下，Web 项目不可打包。若要覆盖默认行为，请将以下属性添加到 .csproj 文件中：

```
<PropertyGroup>
   <IsPackable>true</IsPackable>
</PropertyGroup>
```

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

以长格式传入时，此命令支持 `dotnet restore` 选项(例如，`--source`)。不支持缩写选项，例如 `-s`。

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

自变量

`PROJECT | SOLUTION`

要打包的项目或解决方案。它可能是 csproj 文件、vbproj 文件、fsproj 文件、解决方案文件或目录的路径。如果未指定，此命令会搜索当前目录，以获取项目文件或解决方案文件。

选项

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `--force`

  强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--include-source`

  除输出目录中的常规 NuGet 包外，还包括调试符号 NuGet 包。源文件包括在符号包内的 `src` 文件夹中。

- `--include-symbols`

  除输出目录中的常规 NuGet 包外，还包括调试符号 NuGet 包。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--no-build`

  打包前不生成项目。还将隐式设置 `--no-restore` 标记。

- `--no-dependencies`

  忽略项目间引用，仅还原根项目。

- `--no-restore`

  运行此命令时不执行隐式还原。

- `--nologo`

  不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `-o|--output <OUTPUT_DIRECTORY>`

  将生成的包放置在指定目录。

- `--runtime <RUNTIME_IDENTIFIER>`

  指定要为其还原包的目标运行时。有关运行时标识符 (RID) 的列表，请参阅 RID 目录。

- `-s|--serviceable`

  设置包中可用的标志。有关详细信息，请参阅 .NET 博客：.NET Framework 4.5.1 支持 .NET NuGet 库的 Microsoft 安全更新。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。有关详细信息，请参阅 LoggerVerbosity。

- `--version-suffix <VERSION_SUFFIX>`

  定义 `VersionSuffix` MSBuild 属性的值。此属性对包版本的影响取决于 `Version` 和 `VersionPrefix` 属性的值，如下表所示：

[illegible]	[illegible]
无	`1.0.0`
`Version`	`$(Version)`
仅 `VersionPrefix`	`$(VersionPrefix)`
仅 `VersionSuffix`	`1.0.0-$(VersionSuffix)`
`VersionPrefix` 和 `VersionSuffix`	`$(VersionPrefix)-$(VersionSuffix)`

  如果要使用 `--version-suffix`，请在项目文件中指定 `VersionPrefix` 而不是 `Version`。例如，如果 `VersionPrefix` 是 `0.1.2` 并且你将 `--version-suffix rc.1` 传递给 `dotnet pack`，则包版本将是 `0.1.2-rc.1`。

  如果 `Version` 具有值并且你将 `--version-suffix` 传递到 `dotnet pack`，则忽略为 `--version-suffix` 指定的值。

示例

- 打包当前目录中的项目：

```
dotnet pack
```

- 打包 `app1` 项目：

```
dotnet pack ~/projects/app1/project.csproj
```

- 打包当前目录中的项目并将生成的包放置到 `nupkgs` 文件夹：

```
dotnet pack --output nupkgs
```

- 将当前目录中的项目打包到 `nupkgs` 文件夹并跳过生成步骤：

```
dotnet pack --no-build --output nupkgs
```

- 将项目的版本后缀配置为 *.csproj* 文件中的 `<VersionSuffix>$(VersionSuffix)</VersionSuffix>`，使用给定的后缀打包当前项目，并更新生成的程序包版本：

```
dotnet pack --version-suffix "ci-1234"
```

- 使用 `PackageVersion` MSBuild 属性将包版本设置为 `2.1.0`：

```
dotnet pack -p:PackageVersion=2.1.0
```

- 打包特定目标框架的项目：

```
dotnet pack -p:TargetFrameworks=net45
```

- 打包项目，并使用特定运行时 (Windows 10) 进行还原操作：

```
dotnet pack --runtime win10-x64
```

- 使用 .nuspec 文件打包项目：

```
dotnet pack ~/projects/app1/project.csproj -p:NuspecFile=~/projects/app1/project.nuspec -
p:NuspecBasePath=~/projects/app1/nuget
```

  要了解如何使用 `NuspecFile`、`NuspecBasePath` 和 `NuspecProperties`，请参阅以下资源：

  - 使用 .nuspec 打包
  - 用于创建自定义包的高级扩展点
  - 全局属性

dotnet publish

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet publish` - 将应用程序及其依赖项发布到文件夹以部署到托管系统。

摘要

```
dotnet publish [<PROJECT>|<SOLUTION>] [-a|--arch <ARCHITECTURE>]
    [-c|--configuration <CONFIGURATION>]
    [-f|--framework <FRAMEWORK>] [--force] [--interactive]
    [--manifest <PATH_TO_MANIFEST_FILE>] [--no-build] [--no-dependencies]
    [--no-restore] [--nologo] [-o|--output <OUTPUT_DIRECTORY>]
    [--os <OS>] [-r|--runtime <RUNTIME_IDENTIFIER>]
    [--self-contained [true|false]]
    [--no-self-contained] [-v|--verbosity <LEVEL>]
    [--version-suffix <VERSION_SUFFIX>]

dotnet publish -h|--help
```

描述

`dotnet publish` 编译应用程序、读取 project 文件中指定的所有依赖项并将生成的文件集发布到目录。输出包括以下资产：

- 扩展名为 dll 的程序集中的中间语言 (IL) 代码。
- 包含项目所有依赖项的 .deps.json 文件。
- .runtimeconfig.json 文件，其中指定了应用程序所需的共享运行时，以及运行时的其他配置选项（例如垃圾回收类型）。
- 应用程序的依赖项，将这些依赖项从 NuGet 缓存复制到输出文件夹。

`dotnet publish` 命令的输出可供部署至托管系统（例如服务器、电脑、Mac、笔记本电脑）以便执行。若要准备用于部署的应用程序，这是唯一正式受支持的方法。根据项目指定的部署类型，托管系统不一定已在其上安装 .NET 共享运行时。有关详细信息，请参阅使用 .NET CLI 发布 .NET 应用。

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

MSBuild

`dotnet publish` 命令调用 MSBuild，后者会调用 `Publish` 目标。如果特定项目的 `IsPublishable` 属性设置为 `false`，则无法调用 `Publish` 目标，并且 `dotnet publish` 命令仅在项目上运行隐式 dotnet restore。

任何传递给 `dotnet publish` 的参数都将传递给 MSBuild。`-c` 和 `-o` 参数分别映射到 MSBuild 的

`Configuration` 和 `PublishDir` 属性。

`dotnet publish` 命令接受 MSBuild 选项，如用来设置属性的 `-p` 和用来定义记录器的 `-l`。例如，可以使用以下格式设置 MSBuild 属性：`-p:<NAME>=<VALUE>`。

还可通过引用 .pubxml 文件（自 .NET Core 3.1 SDK 起可用）设置与发布相关的属性。例如：

```
dotnet publish -p:PublishProfile=FolderProfile
```

前面的示例使用 <project_folder>/Properties/PublishProfiles 文件夹中的 FolderProfile.pubxml 文件。如果在设置 `PublishProfile` 属性时指定路径和文件扩展名，则它们会被忽略。默认情况下，MSBuild 会在 Properties/PublishProfiles 文件夹中查找，并假定 .pubxml 文件扩展名。若要指定包含扩展名的路径和文件名，请设置 `PublishProfileFullPath` 属性，而不是 `PublishProfile` 属性。

以下 MSBuild 属性更改 `dotnet publish` 的输出。

- `PublishReadyToRun`

  以 ReadyToRun (R2R) 格式编译应用程序集。R2R 是一种预先 (AOT) 编译形式。有关详细信息，请参阅 ReadyToRun 图像。自 .NET Core 3.0 SDK 起可用。

  若要查看有关缺少的依赖项可能导致运行时失败的警告，请使用 `PublishReadyToRunShowWarnings=true`。

  建议在发布配置文件中而不是在命令行中指定 `PublishReadyToRun`。

- `PublishSingleFile`

  将应用打包到特定于平台的单个文件可执行文件中。有关单文件发布的详细信息，请参阅单文件捆绑程序设计文档。自 .NET Core 3.0 SDK 起可用。

  建议在项目文件中而不是在命令行中指定此选项。

- `PublishTrimmed`

  在发布自包含的可执行文件时，剪裁未使用的库以减小应用的部署大小。有关详细信息，请参阅剪裁自包含部署和可执行文件。自 .NET 6 SDK 起可用。

  建议在项目文件中而不是在命令行中指定此选项。

有关更多信息，请参见以下资源：

- MSBuild 命令行参考
- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件 (.pubxml)
- dotnet msbuild

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

自变量

- `PROJECT|SOLUTION`

  要发布的项目或解决方案。

  - `PROJECT` 是 C#、F# 或 Visual Basic 项目文件的路径和文件名，或包含 C#、F# 或 Visual Basic 项目文件的目录的路径。如果未指定目录，则默认为当前目录。
  - `SOLUTION` 是解决方案文件（扩展名为 .sln）的路径和文件名，或包含解决方案文件的目录的路径。

如果未指定目录，则默认为当前目录。自 .NET Core 3.0 SDK 起可用。

选项

- `-a|--arch <ARCHITECTURE>`

  指定目标体系结构。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--arch x86` 会将 RID 设置为 `win-x86`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

  为指定的目标框架发布应用程序。必须在项目文件中指定目标框架。

- `--force`

  强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--manifest <PATH_TO_MANIFEST_FILE>`

  指定一个或多个目标清单，用于剪裁与应用程序一同发布的一组包。清单文件是 `dotnet store` 命令输出的一部分。若要指定多个清单，请为每个清单添加一个 `--manifest` 选项。

- `--no-build`

  发布前不生成项目。还将隐式设置 `--no-restore` 标记。

- `--no-dependencies`

  忽略项目间引用，仅还原根项目。

- `--nologo`

  不显示启动版权标志或版权消息。自 .NET Core 3.0 SDK 起可用。

- `--no-restore`

  运行此命令时不执行隐式还原。

- `-o|--output <OUTPUT_DIRECTORY>`

  指定输出目录的路径。

  如果未指定，则默认为依赖框架的可执行文件和跨平台二进制文件的路径 [project_file_folder]/bin/[configuration]/[framework]/publish/。默认为独立的可执行文件路径 [project_file_folder]/bin/[configuration]/[framework]/[runtime]/publish/。

  在 Web 项目中，如果输出文件夹位于项目文件夹，则连续的 `dotnet publish` 命令将产生嵌套的输出文件夹。例如，如果项目文件夹是“myproject”，发布输出文件夹是“myproject/publish”，并且运行 `dotnet publish` 两次，则第二次运行会将“.config”和“.json”等内容文件放入“myproject/publish/publish”。

若要避免嵌套发布文件夹，请指定一个不在项目文件夹正下方的发布文件夹，或从项目中排除发布文件夹。若要排除名为“publishoutput”的发布文件夹，请将以下元素添加到“.csproj”文件中的 `PropertyGroup` 元素中：

```
<DefaultItemExcludes>$(DefaultItemExcludes);publishoutput**</DefaultItemExcludes>
```

  - .NET Core 3.x SDK 和更高版本

    如果在发布项目时指定相对路径，则生成的输出目录相对于当前工作目录，而不是项目文件位置。

    如果在发布解决方案时指定相对路径，则所有项目的所有输出都会进入相对于当前工作目录的指定文件夹中。若要使发布输出进入每个项目的单独文件夹，请使用 msbuild `PublishDir` 属性（而不是 `--output` 选项）指定相对路径。例如，`dotnet publish -p:PublishDir=.\publish` 将每个项目的发布输出发送到包含项目文件的文件夹下的 `publish` 文件夹中。

  - .NET Core 2.x SDK

    如果在发布项目时指定相对路径，则生成的输出目录相对于项目文件位置，而不是当前工作目录。

    如果在发布解决方案时指定相对路径，则每个项目的输出会进入相对于项目文件位置的单独文件夹中。如果在发布解决方案时指定绝对路径，则所有项目的所有发布输出都会进入指定文件夹中。

- `--os <OS>`

  指定目标操作系统 (OS)。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--os os` 会将 RID 设置为 `os-x64`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `--self-contained [true|false]`

  .NET 运行时随应用程序一同发布，因此无需在目标计算机上安装运行时。如果指定了运行时标识符，并且项目是可执行项目（而不是库项目），则默认值为 `true`。有关详细信息，请参阅 .NET 应用程序发布和使用 .NET CLI 发布 .NET 应用。

  如果在未指定 `true` 或 `false` 的情况下使用此选项，则默认值为 `true`。在这种情况下，请不要紧接在 `--self-contained` 后放置解决方案或项目参数，因为该位置需要 `true` 或 `false`。

- `--no-self-contained`

  等效于 `--self-contained false`。自 .NET Core 3.0 SDK 起可用。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  发布针对给定运行时的应用程序。有关运行时标识符 (RID) 的列表，请参阅 RID 目录。有关详细信息，请参阅 .NET 应用程序发布和使用 .NET CLI 发布 .NET 应用。如果使用此选项，则还要使用 `--self-contained` 或 `--no-self-contained`。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

- `--version-suffix <VERSION_SUFFIX>`

  定义版本后缀来替换项目文件的版本字段中的星号 (`*`)。

示例

- 为当前目录中的项目创建一个依赖框架的跨平台二进制文件：

```
dotnet publish
```

自 .NET Core 3.0 SDK 起，此示例还为当前平台创建依赖框架的可执行文件。

- 针对特定运行时，为当前目录中的项目创建独立可执行文件：

```
dotnet publish --runtime osx.10.11-x64
```

项目文件中必须包含 RID。

- 针对特定平台，为当前目录中的项目创建依赖框架的可执行文件：

```
dotnet publish --runtime osx.10.11-x64 --self-contained false
```

项目文件中必须包含 RID。此示例适用于 .NET Core 3.0 SDK 及更高版本。

- 针对特定运行时和目标框架，在当前目录中发布项目：

```
dotnet publish --framework netcoreapp3.1 --runtime osx.10.11-x64
```

- 发布指定的项目文件：

```
dotnet publish ~/projects/app1/app1.csproj
```

- 发布当前应用程序，但在还原操作期间不还原项目到项目 (P2P) 引用，只还原根项目：

```
dotnet publish --no-dependencies
```

请参阅

- .NET 应用程序发布概述
- 使用 .NET CLI 发布 .NET 应用
- 目标框架
- 运行时标识符 (RID) 目录
- 处理 macOS Catalina 公证
- 已发布应用程序的目录结构
- MSBuild 命令行参考
- 用于 ASP.NET Core 应用部署的 Visual Studio 发布配置文件 (.pubxml)
- dotnet msbuild
- ILLInk.Tasks

dotnet restore

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet restore` - 恢复项目的依赖项和工具。

摘要

```
dotnet restore [<ROOT>] [--configfile <FILE>] [--disable-parallel]
    [-f|--force] [--force-evaluate] [--ignore-failed-sources]
    [--interactive] [--lock-file-path <LOCK_FILE_PATH>] [--locked-mode]
    [--no-cache] [--no-dependencies] [--packages <PACKAGES_DIRECTORY>]
    [-r|--runtime <RUNTIME_IDENTIFIER>] [-s|--source <SOURCE>]
    [--use-lock-file] [-v|--verbosity <LEVEL>]

dotnet restore -h|--help
```

描述

`dotnet restore` 命令使用 NuGet 还原依赖项以及在 project 文件中指定的特定于项目的工具。在大多数情况下，不需要显式使用 `dotnet restore` 命令，因为在运行以下命令时，将会在必要时隐式运行 NuGet 还原：

- `dotnet new`
- `dotnet build`
- `dotnet build-server`
- `dotnet run`
- `dotnet test`
- `dotnet publish`
- `dotnet pack`

有时，通过这些命令运行隐式 NuGet 还原可能不方便。例如，某些自动化系统（如生成系统）需要显式调用 `dotnet restore`，以控制还原发生的时间，以便可以控制网络使用量。为了防止运行隐式 NuGet 还原，可以通过上述任意命令使用 `--no-restore` 标记禁用隐式还原。

指定源

为了还原依赖项，NuGet 需要包所在的源。通常通过“nuGet.config”配置文件提供源。安装 .NET SDK 时提供一个默认的配置文件。若要指定其他源，请执行以下任一项操作：

- 在项目目录中创建自己的 nuget.config 文件。有关详细信息，请参阅本文后面介绍的常见 NuGet 配置和 nuget.config 差异。
- 使用诸如 `dotnet nuget add source` 等 `dotnet nuget` 命令。

可以使用 `-s` 选项替代 nuget.config 源。

有关如何使用经过身份验证的源的信息，请参阅使用经过身份验证的源中的包。

全局包文件夹

对于依赖项，可以使用 `--packages` 参数指定还原操作期间放置还原包的位置。如未指定，将使用默认的 NuGet 包缓存，可在所有操作系统上的用户主目录中的 `.nuget/packages` 目录找到它。例如 Linux 上的 /home/user1 或 Windows 上的 C:\Users\user1 。

特定于项目的工具

对于特定于项目的工具，`dotnet restore` 首先还原打包工具所在的包，然后继续还原 project 文件中指定的工具依赖项。

nuget.config 差异

`dotnet restore` 命令的行为会受 Nuget.Config 文件（如果有）中某些设置的影响。例如，在 NuGet.Config 中设置 `globalPackagesFolder` 会将还原的 NuGet 包置于指定的文件夹中。这是在 `dotnet restore` 命令中指定 `--packages` 选项的替代方法。有关详细信息，请参阅 nuget.config 参考。

有三个 `dotnet restore` 可忽略的特定设置：

- bindingRedirects

  绑定重定向不适用于 `<PackageReference>` 元素，并且 .NET 仅支持 NuGet 包的 `<PackageReference>` 元素。

- 解决方案

  此设置特定于 Visual Studio，不适用于 .NET。.NET 不使用 `packages.config` 文件，而是使用 NuGet 包的 `<PackageReference>` 元素。

- trustedSigners

  .NET 5.0.100 SDK 中添加了对跨平台包签名验证的支持。

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

自变量

- `ROOT`

  要还原的项目文件的可选路径。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  禁用并行还原多个项目。

- `--force`

  强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `--force-evaluate`

  即使锁定文件已存在，也会强制还原以重新评估所有依赖项。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  如果存在符合版本要求的包，则源失败时警告。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--lock-file-path <LOCK_FILE_PATH>`

  写入项目锁定文件的输出位置。默认情况下，此位置为 PROJECT_ROOT \packages.lock.json。

- `--locked-mode`

  不允许更新项目锁定文件。

- `--no-cache`

  指定不缓存 HTTP 请求。

- `--no-dependencies`

  当使用项目到项目 (P2P) 引用还原项目时，还原根项目，不还原引用。

- `--packages <PACKAGES_DIRECTORY>`

  指定还原包的目录。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  指定程序包还原的运行时。这用于还原 *.csproj* 文件中的 `<RuntimeIdentifiers>` 标记中未显式列出的运行时的程序包。有关运行时标识符 (RID) 的列表，请参阅 RID 目录。通过多次指定此选项提供多个 RID。

- `-s|--source <SOURCE>`

  指定要在还原操作期间使用的 NuGet 包源的 URI。此设置会替代 nuget.config 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--use-lock-file`

  允许生成项目锁定文件并与还原一起使用。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 还原当前目录中项目的依赖项和工具：

  ```
  dotnet restore
  ```

- 还原在给定路径中找到的 `app1` 项目的依赖项和工具：

  ```
  dotnet restore ./projects/app1/app1.csproj
  ```

- 通过将提供的文件路径用作源，在当前目录中还原项目的依赖项和工具：

```
dotnet restore -s c:\packages\mypackages
```

- 通过将提供的两个文件路径用作源，在当前目录中还原项目的依赖项和工具：

```
dotnet restore -s c:\packages\mypackages -s c:\packages\myotherpackages
```

- 还原当前目录中项目的依赖项和工具，并显示详细的输出：

```
dotnet restore --verbosity detailed
```

dotnet run

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet run` - 无需任何显式编译或启动命令即可运行源代码。

摘要

```
dotnet run [-a|--arch <ARCHITECTURE>] [-c|--configuration <CONFIGURATION>]
    [-f|--framework <FRAMEWORK>] [--force] [--interactive]
    [--launch-profile <NAME>] [--no-build]
    [--no-dependencies] [--no-launch-profile] [--no-restore]
    [--os <OS>] [--project <PATH>] [-r|--runtime <RUNTIME_IDENTIFIER>]
    [-v|--verbosity <LEVEL>] [[--] [application arguments]]

dotnet run -h|--help
```

描述

`dotnet run` 命令为从源代码使用一个命令运行应用程序提供了一个方便的选项。这对从命令行中进行快速迭代开发很有帮助。命令取决于生成代码的 `dotnet build` 命令。对于此生成的任何要求，例如项目必须首先还原，同样适用于 `dotnet run`。

> **NOTE**
> `dotnet run` 不遵守 `/property:property=value` 等参数，`dotnet build` 遵守这些参数。

输出文件会写入到默认位置，即 `bin/<configuration>/<target>`。例如，如果具有 `netcoreapp2.1` 应用程序并且运行 `dotnet run`，则输出置于 `bin/Debug/netcoreapp2.1`。将根据需要覆盖文件。临时文件将置于 `obj` 目录。

如果该项目指定多个框架，在不使用 `-f|--framework <FRAMEWORK>` 选项指定框架时，执行 `dotnet run` 将导致错误。

在项目上下文，而不是生成程序集中使用 `dotnet run` 命令。如果尝试改为运行依赖于框架的应用程序 DLL，则必须在不使用命令的情况下使用 dotnet。例如，若要运行 `myapp.dll`，请使用：

```
dotnet myapp.dll
```

有关 `dotnet` 驱动程序的详细信息，请参阅 .NET 命令行工具 (CLI) 主题。

若要运行应用程序，`dotnet run` 命令需从 NuGet 缓存解析共享运行时之外的应用程序依赖项。因为它使用缓存的依赖项，因此，不推荐在生产中使用 `dotnet run` 来运行应用程序。相反，使用 `dotnet publish` 命令创建部署，并部署已发布的输出。

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

以长格式传入时，此命令支持 `dotnet restore` 选项（例如，`--source`）。不支持缩写选项，例如 `-s`。

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

选项

- `--`

  将参数分隔到正在运行的应用程序的参数的 `dotnet run`。在此分隔符后的所有参数均传递给已运行的应用程序。

- `-a|--arch <ARCHITECTURE>`

  指定目标体系结构。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--arch x86` 会将 RID 设置为 `win-x86`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `-f|--framework <FRAMEWORK>`

  使用指定框架生成并运行应用。框架必须在项目文件中进行指定。

- `--force`

  强制解析所有依赖项，即使上次还原已成功，也不例外。指定此标记等同于删除 project.assets.json 文件。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `--launch-profile <NAME>`

  启动应用程序时要使用的启动配置文件（若有）的名称。启动配置文件在 launchSettings.json 文件中进行定义，通常称为 `Development`、`Staging` 和 `Production`。有关详细信息，请参阅使用多个环境。

- `--no-build`

  运行前不生成项目。还隐式设置 `--no-restore` 标记。

- `--no-dependencies`

  当使用项目到项目 (P2P) 引用还原项目时，还原根项目，不还原引用。

- `--no-launch-profile`

  不尝试使用 launchSettings.json 配置应用程序。

- `--no-restore`

运行此命令时不执行隐式还原。

- `--os <OS>`

  指定目标操作系统 (OS)。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--os os` 会将 RID 设置为 `os-x64`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `--project <PATH>`

  指定要运行的项目文件的路径（文件夹名称或完整路径）。如果未指定，则默认为当前目录。

  从 .NET 6 SDK 开始，`--project` 的缩写 `-p` 已弃用。在从 .NET 6 RC1 SDK 发布后的有限时段内，仍可对 `--project` 使用 `-p`，不过会显示弃用警告。如果为选项提供的参数不包含 `=`，则命令将接受 `--project` 的短格式 `-p`。否则，命令会假设 `-p` 是 `--property` 的短格式。在 .NET 7 中将逐渐淘汰这种灵活使用 `-p` 来表示 `--project` 的做法。

- `--property:<NAME>=<VALUE>`

  设置一个或多个 MSBuild 属性。指定以分号分隔的多个属性，或通过重复该选项指定多个属性：

  ```
  --property:<NAME1>=<VALUE1>;<NAME2>=<VALUE2>
  --property:<NAME1>=<VALUE1> --property:<NAME2>=<VALUE2>
  ```

  短格式 `-p` 可用于 `--property`。如果为选项提供的参数包含 `=`，则接受 `-p` 作为 `--property` 的短格式。否则，命令会假设 `-p` 是 `--project` 的短格式。

  若要将 `--property` 传递给应用程序而不是设置 MSBuild 属性，请在 `--` 语法分隔符后面提供该选项，例如：

  ```
  dotnet run -- --property name=value
  ```

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  指定要为其还原包的目标运行时。有关运行时标识符 (RID) 的列表，请参阅 RID 目录。自 .NET Core 3.0 SDK 起可用的 `-r` 简短选项。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 运行当前目录中的项目：

  ```
  dotnet run
  ```

- 运行指定的项目：

  ```
  dotnet run --project ./projects/proj1/proj1.csproj
  ```

- 运行当前目录中的项目，并指定 Release 配置：

```
dotnet run --property:Configuration=Release
```

- 运行当前目录中的项目(在本例中，`--help` 参数被传递到应用程序，因为使用了空白的 `--` 选项)：

```
dotnet run --configuration Release -- --help
```

- 在仅显示最小输出的当前目录中还原项目的依赖项和工具，然后运行项目：

```
dotnet run --verbosity m
```

dotnet sdk check

2021/11/16 •

本文适用于：✔ .NET 6 及更高版本

名称

`dotnet sdk check` - 列出每个功能区段最新可用的 .NET SDK 和 .NET 运行时版本。

摘要

```
dotnet sdk check

dotnet sdk check -h|--help
```

说明

使用 `dotnet sdk check` 命令，可以更轻松地跟踪新版 SDK 和运行时何时可用。在每个功能区段中，它会告知：

- .NET SDK 和 .NET 运行时的最新可用版本。
- 安装的版本是最新的还是不受支持的版本。

下面是该命令的输出示例：

```
.NET SDKs:
Version                         Status
-----------------------------------------------------------
2.1.816                         Up to date.
2.2.401                         .NET 2.2 is out of support.
3.1.410                         Up to date.
5.0.204                         Up to date.
5.0.301                         Up to date.

.NET Runtimes:
Name                              Version                     Status
-----------------------------------------------------------------------------------------
Microsoft.AspNetCore.All          2.1.28                      Up to date.
Microsoft.AspNetCore.App          2.1.28                      Up to date.
Microsoft.NETCore.App             2.1.28                      Up to date.
Microsoft.AspNetCore.All          2.2.6                       .NET 2.2 is out of support.
Microsoft.AspNetCore.App          2.2.6                       .NET 2.2 is out of support.
Microsoft.NETCore.App             2.2.6                       .NET 2.2 is out of support.
Microsoft.AspNetCore.App          3.1.16                      Up to date.
Microsoft.NETCore.App             3.1.16                      Up to date.
Microsoft.WindowsDesktop.App      3.1.16                      Up to date.
Microsoft.AspNetCore.App          5.0.7                       Up to date.
Microsoft.NETCore.App             5.0.7                       Up to date.
Microsoft.WindowsDesktop.App      5.0.7                       Up to date.
```

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 显示已安装的 .NET SDK 和 .NET 运行时的最新状态。

```
dotnet sdk check
```

dotnet sln

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

“属性”

`dotnet sln` - 在 .NET 解决方案文件中列出或修改项目。

摘要

```
dotnet sln [<SOLUTION_FILE>] [command]

dotnet sln [command] -h|--help
```

描述

使用 `dotnet sln` 命令，可以便捷地在解决方案文件中列出和修改项目。

若要使用 `dotnet sln` 命令，必须存在解决方案文件。如果需要创建一个解决方案文件，请使用 dotnet new 命令，如下例所示：

```
dotnet new sln
```

自变量

- `SOLUTION_FILE`

  要使用的解决方案文件。如果省略此参数，此命令会搜索当前目录来获取一个解决方案文件。如果未找到解决方案文件或找到多个解决方案文件，则该命令将失败。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

命令

`list`

列出解决方案文件中的所有项目。

摘要

```
dotnet sln list [-h|--help]
```

自变量

- `SOLUTION_FILE`

要使用的解决方案文件。如果省略此参数，此命令会搜索当前目录来获取一个解决方案文件。如果未找到解决方案文件或找到多个解决方案文件，则该命令将失败。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

`add`

将一个或多个项目添加到解决方案文件。

摘要

```
dotnet sln [<SOLUTION_FILE>] add [--in-root] [-s|--solution-folder <PATH>] <PROJECT_PATH>
[<PROJECT_PATH>...]
dotnet sln add [-h|--help]
```

自变量

- `SOLUTION_FILE`

  要使用的解决方案文件。如果未指定，此命令会搜索当前目录以获取一个解决方案文件，如果找到多个解决方案文件，则该命令将失败。

- `PROJECT_PATH`

  要添加到解决方案的一个或多个项目的路径。Unix/Linux shell glob 模式扩展由 `dotnet sln` 命令正确处理。

  如果 `PROJECT_PATH` 包括含项目文件夹的文件夹，则路径的该部分将用于创建解决方案文件夹。例如，以下命令在解决方案文件夹 `folder1/folder2` 中使用 `myapp` 创建解决方案：

  ```
  dotnet new sln
  dotnet new console --output folder1/folder2/myapp
  dotnet sln add folder1/folder2/myapp
  ```

  使用 `--in-root` 或 `-s|--solution-folder <PATH>` 选项可重写此默认行为。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--in-root`

  将项目放在解决方案的根目录下，而不是创建解决方案文件夹。无法与 `-s|--solution-folder` 一起使用。自 .NET Core 3.0 SDK 起可用。

- `-s|--solution-folder <PATH>`

  要将项目添加到的目标解决方案文件夹路径。无法与 `--in-root` 一起使用。自 .NET Core 3.0 SDK 起可用。

`remove`

从解决方案文件中删除一个或多个项目。

摘要

```
dotnet sln [<SOLUTION_FILE>] remove <PROJECT_PATH> [<PROJECT_PATH>...]
dotnet sln [<SOLUTION_FILE>] remove [-h|--help]
```

自变量

- `SOLUTION_FILE`

  要使用的解决方案文件。如果保留未指定，此命令会搜索当前目录以获取一个解决方案文件，如果找到多个解决方案文件，则该命令将失败。

- `PROJECT_PATH`

  要从解决方案中删除的一个或多个项目的路径。Unix/Linux shell glob 模式扩展由 `dotnet sln` 命令正确处理。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 在解决方案中列出项目：

  ```
  dotnet sln todo.sln list
  ```

- 将一个 C# 项目添加到解决方案中：

  ```
  dotnet sln add todo-app/todo-app.csproj
  ```

- 从解决方案中删除一个 C# 项目：

  ```
  dotnet sln remove todo-app/todo-app.csproj
  ```

- 将多个 C# 项目添加到解决方案的根目录中：

  ```
  dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj --in-root
  ```

- 将多个 C# 项目添加到解决方案中：

  ```
  dotnet sln todo.sln add todo-app/todo-app.csproj back-end/back-end.csproj
  ```

- 从解决方案中删除多个 C# 项目：

  ```
  dotnet sln todo.sln remove todo-app/todo-app.csproj back-end/back-end.csproj
  ```

- 使用 glob 模式（仅限 Unix/Linux）将多个 C# 项目添加到解决方案中：

  ```
  dotnet sln todo.sln add **/*.csproj
  ```

- 使用 globbing 模式（仅限 Windows PowerShell）将多个 C# 项目添加到解决方案中：

```
dotnet sln todo.sln add (ls -r **/*.csproj)
```

- 使用 glob 模式(仅限 Unix/Linux)将多个 C# 项目从解决方案中删除:

```
dotnet sln todo.sln remove **/*.csproj
```

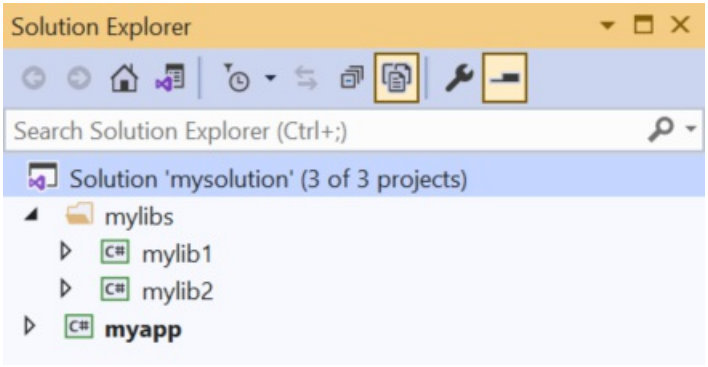
- 使用 globbing 模式(仅限 Windows PowerShell)将多个 C# 项目从解决方案中删除:

```
dotnet sln todo.sln remove (ls -r **/*.csproj)
```

- 创建解决方案、控制台应用和两个类库。将项目添加到解决方案，并使用 `dotnet sln` 的 `--solution-folder` 选项将类库组织到一个解决方案文件夹中。

```
dotnet new sln -n mysolution
dotnet new console -o myapp
dotnet new classlib -o mylib1
dotnet new classlib -o mylib2
dotnet sln mysolution.sln add myapp\myapp.csproj
dotnet sln mysolution.sln add mylib1\mylib1.csproj --solution-folder mylibs
dotnet sln mysolution.sln add mylib2\mylib2.csproj --solution-folder mylibs
```

以下屏幕截图显示了 Visual Studio 2019"解决方案资源管理器"中的结果:



另请参阅

- dotnet/sdk GitHub 存储库(.NET CLI 源)

dotnet store

2021/11/16 •

本文适用于： ✔ .NET Core 2.x SDK 及更高版本

名称

`dotnet store` - 将指定的程序集存储到运行时包存储区。

摘要

```
dotnet store -m|--manifest <PATH_TO_MANIFEST_FILE>
    -f|--framework <FRAMEWORK_VERSION> -r|--runtime <RUNTIME_IDENTIFIER>
    [--framework-version <FRAMEWORK_VERSION>] [--output <OUTPUT_DIRECTORY>]
    [--skip-optimization] [--skip-symbols] [-v|--verbosity <LEVEL>]
    [--working-dir <WORKING_DIRECTORY>]

dotnet store -h|--help
```

说明

`dotnet store` 将指定的程序集存储到运行时包存储区。默认情况下，程序集更适用于目标运行时和框架。有关详细信息，请参阅运行时包存储区主题。

必需选项

- `-f|--framework <FRAMEWORK>`

  指定目标框架。目标框架必须在项目文件中进行指定。

- `-m|--manifest <PATH_TO_MANIFEST_FILE>`

  包存储区清单文件 是包含要存储的包列表的 XML 文件。清单文件的格式与 SDK 样式项目格式兼容。因此，引用所需的包的项目文件能够与 `-m|--manifest` 选项结合使用，以便于将程序集存储到运行时包存储区。若要指定多个清单文件，请为各个文件重复指定选项和路径。例如：`--manifest packages1.csproj --manifest packages2.csproj`。

- `-r|--runtime <RUNTIME_IDENTIFIER>`

  目标运行时标识符。

可选选项

- `--framework-version <FRAMEWORK_VERSION>`

  指定 .NET SDK 版本。使用此选项，可以选择特定的框架版本，不再局限于 `-f|--framework` 选项指定的框架。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-o|--output <OUTPUT_DIRECTORY>`

指定运行时包存储区的路径。如果未指定，默认路径为用户配置文件 .NET 安装目录的 store 子目录。

- `--skip-optimization`

  跳过优化阶段。

- `--skip-symbols`

  跳过符号生成。目前，只能在 Windows 和 Linux 上生成符号。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。有关详细信息，请参阅 LoggerVerbosity。

- `-w|--working-dir <WORKING_DIRECTORY>`

  此命令使用的工作目录。如果未指定，使用当前目录的 obj 子目录。

示例

- 存储 packages.csproj 项目文件中为 .NET Core 2.0.0 指定的包：

```
dotnet store --manifest packages.csproj --framework-version 2.0.0
```

- 存储 packages.csproj 中指定的包，但不进行优化：

```
dotnet store --manifest packages.csproj --skip-optimization
```

另请参阅

- 运行时包存储

dotnet test

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet test` - 用于执行单元测试的 .NET 测试驱动程序。

摘要

```
dotnet test [<PROJECT> | <SOLUTION> | <DIRECTORY> | <DLL>]
    [-a|--test-adapter-path <ADAPTER_PATH>] [--arch <ARCHITECTURE>]
    [--blame] [--blame-crash]
    [--blame-crash-dump-type <DUMP_TYPE>] [--blame-crash-collect-always]
    [--blame-hang] [--blame-hang-dump-type <DUMP_TYPE>]
    [--blame-hang-timeout <TIMESPAN>]
    [-c|--configuration <CONFIGURATION>]
    [--collect <DATA_COLLECTOR_NAME>]
    [-d|--diag <LOG_FILE>] [-f|--framework <FRAMEWORK>]
    [--filter <EXPRESSION>] [--interactive]
    [-l|--logger <LOGGER>] [--no-build]
    [--nologo] [--no-restore] [-o|--output <OUTPUT_DIRECTORY>] [--os <OS>]
    [-r|--results-directory <RESULTS_DIR>] [--runtime <RUNTIME_IDENTIFIER>]
    [-s|--settings <SETTINGS_FILE>] [-t|--list-tests]
    [-v|--verbosity <LEVEL>] [[--] <RunSettings arguments>]

dotnet test -h|--help
```

描述

`dotnet test` 命令用于在给定的解决方案中执行单元测试。 `dotnet test` 命令生成解决方案，并为解决方案中的每个测试项目运行测试主机应用程序。测试主机使用测试框架（例如，MSTest、NUnit 或 xUnit）在给定项目中执行测试，并报告每个测试成功与否。如果所有测试均成功，测试运行程序将返回 0 作为退出代码；否则将返回 1。

对于多目标项目，将为每个目标框架运行测试。测试主机和单元测试框架打包为 NuGet 包，并还原为项目的普通依赖项。

测试项目使用普通 `<PackageReference>` 元素指定测试运行程序，如下方示例项目文件所示：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3" />
  </ItemGroup>

</Project>
```

如果 `Microsoft.NET.Test.Sdk` 是测试主机，则 `xunit` 是测试框架。另外， `xunit.runner.visualstudio` 是测试适

配器，可便于 xUnit 框架与测试主机一起运行。

隐式还原

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

工作负载清单下载

运行此命令时，它将为工作负载启动播发清单的异步后台下载。如果此命令完成后，下载仍在运行，则将停止下载。有关详细信息，请参阅播发清单。

自变量

- `PROJECT | SOLUTION | DIRECTORY | DLL`

  - 指向测试项目的路径。
  - 解决方案的路径。
  - 包含项目或解决方案的目录的路径。
  - 测试项目 .dll 文件的路径。

  如果未指定，则会在当前目录中搜索项目或解决方案。

选项

- `-a|--test-adapter-path <ADAPTER_PATH>`

  要在其中搜索其他测试适配器的目录的路径。只检查后缀为 `.TestAdapter.dll` 的 .dll 文件。如果未指定，则会搜索测试 .dll 的目录。

- `--arch <ARCHITECTURE>`

  指定目标体系结构。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--arch x86` 会将 RID 设置为 `win-x86`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `--blame`

  在意见模式中运行测试。此选项有助于隔离导致测试主机出现故障的有问题的测试。检测到故障时，它会在 `TestResults/<Guid>/<Guid>_Sequence.xml` 中创建一个序列文件，用于捕获在出现故障之前运行的测试的顺序。

- `--blame-crash`（自 .NET 5.0 SDK 起可用）

  在追责模式下运行测试，并在测试主机意外退出时收集故障转储。此选项取决于所使用的 .NET 版本、错误的类型和操作系统。

  对于托管代码中的异常，将在 .NET 5.0 及更高版本上自动收集转储。对于 testhost 或也在 .NET 5.0 上运行并且出现故障的任何子进程，它将生成转储。本机代码中的故障将不会生成转储。此选项适用于 Windows、macOS 和 Linux。

  本机代码中的故障转储（或者当使用 .NET Core 3.1 或更早版本时）只能使用 Procdump 在 Windows 上进行收集。包含 procdump.exe 和 procdump64.exe 的目录必须位于 PATH 或 PROCDUMP_PATH 环境变量中。下载工具。意味着 `--blame`。

若要从 .NET 5.0 或更高版本上运行的本机应用程序收集故障转储，可以通过将 `VSTEST_DUMP_FORCEPROCDUMP` 环境变量设置为 `1` 来强制执行 Procdump 的使用。

- `--blame-crash-dump-type <DUMP_TYPE>`（自 .NET 5.0 SDK 起可用）

  要收集的故障转储的类型。意味着 `--blame-crash`。

- `--blame-crash-collect-always`（自 .NET 5.0 SDK 起可用）

  在预期和意外的测试主机退出时收集故障转储。

- `--blame-hang`（自 .NET 5.0 SDK 起可用）

  在追责模式下运行测试，并在测试超过给定超时时长时收集挂起转储。

- `--blame-hang-dump-type <DUMP_TYPE>`（自 .NET 5.0 SDK 起可用）

  要收集的故障转储的类型。它应为 `full`、`mini` 或 `none`。指定 `none` 时，测试主机将在超时时终止，但不会收集任何转储。意味着 `--blame-hang`。

- `--blame-hang-timeout <TIMESPAN>`（自 .NET 5.0 SDK 起可用）

  每个测试超时时间，在此时间后，将触发挂起转储，并转储和终止测试主机进程及其所有子进程。超时值是采用以下格式之一指定的：

  - 1.5h、1.5hour、1.5hours
  - 90m、90min、90minute、90minutes
  - 5400s、5400sec、5400second、5400seconds
  - 5400000ms、5400000mil、5400000millisecond、5400000milliseconds

  如果未使用单位（例如，5400000），则假定该值以毫秒为单位。与数据驱动的测试一起使用时，超时行为取决于所使用的测试适配器。对于 xUnit 和 NUnit，会在每个测试用例后更新超时。对于 MSTest，超时用于所有测试用例。此选项在使用 netcoreapp 2.1 和更高版本的 Windows 上，在使用 netcoreapp 3.1 和更高版本的 Linux 上以及在使用 net5.0 或更高版本的 macOS 上受支持。意味着 `--blame` 和 `--blame-hang`。

- `-c|--configuration <CONFIGURATION>`

  定义生成配置。大多数项目的默认配置为 `Debug`，但你可以覆盖项目中的生成配置设置。

- `--collect <DATA_COLLECTOR_NAME>`

  为测试运行启用数据收集器。有关详细信息，请参阅监视和分析测试运行。

  若要在 .NET Core 支持的任何平台上收集代码覆盖率，请安装 Coverlet 并使用 `--collect:"XPlat Code Coverage"` 选项。

  在 Windows 上，可以使用 `--collect "Code Coverage"` 选项收集代码覆盖率。此选项将生成“.coverage”文件，该文件可在 Visual Studio 2019 Enterprise 中打开。有关详细信息，请参阅使用代码覆盖率和自定义代码覆盖率分析。

- `-d|--diag <LOG_FILE>`

  启用测试平台的诊断模式，并将诊断消息写入到指定文件及其旁边的文件。正在记录消息的进程可确定创建了哪些文件，如测试主机日志的 `*.host_<date>.txt`，以及数据收集器日志的 `*.datacollector_<date>.txt`。

- `-f|--framework <FRAMEWORK>`

  强制将 `dotnet` 或 .NET Framework 测试主机用于测试二进制文件。此选项只确定要使用哪种类型的主机。要使用的实际框架版本由测试项目的 runtimeconfig.json 决定。如果未指定，则 TargetFramework 程

序集特性用于确定主机的类型。如果已从 .dll 中去除此特性，则使用的是 .NET Framework 主机。

- `--filter <EXPRESSION>`

  使用给定表达式筛选掉当前项目中的测试。有关详细信息，请参阅筛选选项详细信息部分。若要获取使用选择性单元测试筛选的其他信息和示例，请参阅运行选择性单元测试。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。自 .NET Core 3.0 SDK 起可用。

- `-l|--logger <LOGGER>`

  指定测试结果记录器。与 MSBuild 不同，dotnet 测试不接受缩写，应使用 `-l "console;verbosity=detailed"`，而不使用 `-l "console;v=d"`。多次指定参数，以启用多个记录器。

- `--no-build`

  不在运行测试项目之前生成它。还将隐式设置 - `--no-restore` 标记。

- `--nologo`

  运行测试，而不显示 Microsoft TestPlatform 横幅。自 .NET Core 3.0 SDK 起可用。

- `--no-restore`

  运行此命令时不执行隐式还原。

- `-o|--output <OUTPUT_DIRECTORY>`

  查找要运行的二进制文件的目录。如果未指定，则默认路径为 `./bin/<configuration>/<framework>/`。对于具有多个目标框架的项目（通过 `TargetFrameworks` 属性），在指定此选项时还需要定义 `--framework`。`dotnet test` 始终从输出目录运行测试。可以使用 AppDomain.BaseDirectory 以使用输出目录中的测试资产。

- `--os <OS>`

  指定目标操作系统 (OS)。这是用于设置运行时标识符 (RID) 的简写语法，其中提供的值与默认 RID 相结合。例如，在 `win-x64` 计算机上，指定 `--os os` 会将 RID 设置为 `os-x64`。如果使用此选项，请不要使用 `-r|--runtime` 选项。从 .NET 6 Preview 7 开始提供。

- `-r|--results-directory <RESULTS_DIR>`

  用于放置测试结果的目录。如果指定的目录不存在，则会创建该目录。默认值为包含项目文件的目录中的 `TestResults`。

- `--runtime <RUNTIME_IDENTIFIER>`

  要针对其测试的目标运行时。

- `-s|--settings <SETTINGS_FILE>`

  `.runsettings` 文件用于运行测试。`TargetPlatform` 元素 (x86|x64) 对 `dotnet test` 不起作用。若要运行面向 x86 的测试，请安装 .NET Core 的 x86 版本。路径上 dotnet.exe 的位数是用于运行测试的内容。有关更多信息，请参见以下资源：

  - 使用 `.runsettings` 文件配置单元测试。
  - 配置测试运行

- `-t|--list-tests`

  列出已发现的测试，而不是运行测试。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

- `RunSettings` 参数

内联的 `RunSettings` 作为"-- "(请注意 -- 后面有空格)后的最后一个命令行参数传递。内联的 `RunSettings` 被指定为 `[name]=[value]` 对。空格用于分隔多个 `[name]=[value]` 对。

示例：`dotnet test -- MSTest.DeploymentEnabled=false MSTest.MapInconclusiveToFailed=True`

有关详细信息，请参阅通过命令行传递 RunSettings 参数。

示例

- 运行当前目录所含项目中的测试：

```
dotnet test
```

- 运行 `test1` 项目中的测试：

```
dotnet test ~/projects/test1/test1.csproj
```

- 在当前目录运行项目中的测试，并以 trx 格式生成测试结果文件：

```
dotnet test --logger trx
```

- 在当前目录运行项目中的测试，并生成代码覆盖率文件(安装 Coverlet 收集器集成后)：

```
dotnet test --collect:"XPlat Code Coverage"
```

- 在当前目录运行项目中的测试，并生成代码覆盖率文件(仅限 Windows)：

```
dotnet test --collect "Code Coverage"
```

- 在当前目录中运行项目中的测试，并将详细的测试结果记录到控制台：

```
dotnet test --logger "console;verbosity=detailed"
```

- 在当前目录下的项目中运行测试，并报告在测试主机发生故障时正在进行的测试：

```
dotnet test --blame
```

筛选选项详细信息

`--filter <EXPRESSION>`

`<Expression>` 格式为 `<property><operator><value>[|&<Expression>]`。

`<property>` 是 `Test Case` 的特性。下面介绍了常用单元测试框架支持的属性：

[illegible]	[illegible]
MSTest	• FullyQualifiedName • "属性" • ClassName • Priority • TestCategory
xUnit	• FullyQualifiedName • DisplayName • 类别
NUnit	• FullyQualifiedName • "属性" • TestCategory • Priority

`<operator>` 说明了属性和值之间的关系：

[illegible]	[illegible]
`=`	完全匹配
`!=`	非完全匹配
`~`	包含
`!~`	不包含

`<value>` 是字符串。所有查找都不区分大小写。

不含 `<operator>` 的表达式自动被视为 `FullyQualifiedName` 属性上的 `contains`（例如，`dotnet test --filter xyz` 与 `dotnet test --filter FullyQualifiedName~xyz` 相同）。

表达式可与条件运算符结合使用：

[illegible]	[illegible]
`|`	或
`&`	AND

使用条件运算符时，可以用括号将表达式括起来（例如，`(Name~TestMethod1) | (Name~TestMethod2)`）。

若要获取使用选择性单元测试筛选的其他信息和示例，请参阅运行选择性单元测试。

请参阅

- 框架和目标

- .NET 运行时标识符 (RID) 目录
- 通过命令行传递 runsettings 参数

dotnet tool install

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool install` - 在计算机上安装指定的 .NET 工具。

摘要

```
dotnet tool install <PACKAGE_NAME> -g|--global
    [--add-source <SOURCE>] [--configfile <FILE>] [--disable-parallel]
    [--framework <FRAMEWORK>] [--ignore-failed-sources] [--interactive]
    [--no-cache] [--tool-manifest <PATH>] [-v|--verbosity <LEVEL>]
    [--version <VERSION_NUMBER>]

dotnet tool install <PACKAGE_NAME> --tool-path <PATH>
    [--add-source <SOURCE>] [--configfile <FILE>] [--disable-parallel]
    [--framework <FRAMEWORK>] [--ignore-failed-sources] [--interactive]
    [--no-cache] [--tool-manifest <PATH>] [-v|--verbosity <LEVEL>]
    [--version <VERSION_NUMBER>]

dotnet tool install <PACKAGE_NAME> [--local]
    [--add-source <SOURCE>] [--configfile <FILE>] [--disable-parallel]
    [--framework <FRAMEWORK>] [--ignore-failed-sources] [--interactive]
    [--no-cache] [--tool-manifest <PATH>] [-v|--verbosity <LEVEL>]
    [--version <VERSION_NUMBER>]

dotnet tool install -h|--help
```

描述

`dotnet tool install` 命令提供一种在计算机上安装 .NET 工具的方法。若要使用命令，请指定以下安装选项之一：

- 若要在默认位置中安装全局工具，请使用 `--global` 选项。
- 若要在自定义位置中安装全局工具，请使用 `--tool-path` 选项。
- 若要安装本地工具，请省略 `--global` 和 `--tool-path` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

指定 `-g` 或 `--global` 选项时，全局工具默认安装在以下目录中：

(OS)
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

本地工具将添加到当前目录下 .config 目录中的 dotnet-tools.json 文件中。如果清单文件尚不存在，请通过运行以下命令来创建它：

```
dotnet new tool-manifest
```

有关详细信息，请参阅安装本地工具。

自变量

- `PACKAGE_NAME`

  包含要安装的 .NET 工具的 NuGet 包的名称/ID。

选项

- `--add-source <SOURCE>`

  添加安装过程中要使用的其他 NuGet 包源。系统会并行访问这些源，而不是按某种优先级顺序依次访问。如果同一个包和版本在多个源中，则选取速度最快的源。有关详细信息，请查看安装 NuGet 包时会发生什么情况？。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  防止并行还原多个项目。

- `--framework <FRAMEWORK>`

  指定要安装工具的目标框架。默认情况下，.NET SDK 尝试选择最合适的目标框架。

- `-g|--global`

  指定安装是用户范围的。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 指定本地工具安装。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--local`

  更新工具和本地工具清单。不能与 `--global` 选项或 `--tool-path` 选项一起使用。

- `--no-cache`

  不要缓存包和 HTTP 请求。

- `--tool-manifest <PATH>`

  清单文件的路径。

- `--tool-path <PATH>`

  指定全局工具的安装位置。路径可以是绝对的，也可以是相对的。如果路径不存在，命令会尝试创建它。省略 `--global` 和 `--tool-path` 指定本地工具安装。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。有关详细信息，请参阅 LoggerVerbosity。

- `--version <VERSION_NUMBER>`

  要安装的工具版本。默认情况下，安装最新的稳定包版本。使用此选项安装工具的预览版或较旧版本。

示例

- `dotnet tool install -g dotnetsay`

  在默认位置中安装 dotnetsay 全局工具。

- `dotnet tool install dotnetsay --tool-path c:\global-tools`

  在特定 Windows 目录中安装 dotnetsay 全局工具。

- `dotnet tool install dotnetsay --tool-path ~/bin`

  在特定 Linux/macOS 目录中安装 dotnetsay 全局工具。

- `dotnet tool install -g dotnetsay --version 2.0.0`

  安装 2.0.0 版的 dotnetsay 全局工具。

- `dotnet tool install dotnetsay`

  在当前目录中安装 dotnetsay 本地工具。

请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool list

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool list` - 列出计算机上当前安装的所有指定类型的 .NET 工具。

摘要

```
dotnet tool list -g|--global

dotnet tool list --tool-path <PATH>

dotnet tool list --local

dotnet tool list

dotnet tool list -h|--help
```

描述

`dotnet tool list` 命令提供一种方法，用于在计算机上安装所有 .NET 全局、工具路径或本地工具。此命令列出包名称、安装的版本以及工具命令。若要使用命令，请指定以下项之一：

- 若要列出在默认位置安装的全局工具，请使用 `--global` 选项
- 若要列出在自定义位置安装的全局工具，请使用 `--tool-path` 选项。
- 若要列出本地工具，请使用 `--local` 选项或省略 `--global`、`--tool-path` 和 `--local` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

选项

- `-g|--global`

  列出用户范围的全局工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 将列出本地工具。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--local`

  列出当前目录的本地工具。不能与 `--global` 或 `--tool-path` 选项组合。即使未指定 `--local`，同时省略 `--global` 和 `--tool-path` 也会列出本地工具。

- `--tool-path <PATH>`

  指定用于查找全局工具的自定义位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 将列出本地工具。

示例

- `dotnet tool list -g`

  列出计算机上安装的所有用户范围的全局工具（当前用户配置文件）。

- `dotnet tool list --tool-path c:\global-tools`

  列出特定 Windows 目录中的全局工具。

- `dotnet tool list --tool-path ~/bin`

  列出特定 Linux/macOS 目录中的全局工具。

- `dotnet tool list` 或 `dotnet tool list --local`

  列出当前目录中所有可用的本地工具。

请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool restore

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

“属性”

`dotnet tool restore` - 安装当前目录范围内的 .NET 本地工具。

摘要

```
dotnet tool restore
    [--configfile <FILE>] [--add-source <SOURCE>]
    [--tool-manifest <PATH_TO_MANIFEST_FILE>] [--disable-parallel]
    [--ignore-failed-sources] [--no-cache] [--interactive]
    [-v|--verbosity <LEVEL>]

dotnet tool restore -h|--help
```

描述

`dotnet tool restore` 命令查找当前目录范围内的工具清单文件，并安装其中列出的工具。有关清单文件的信息，请参阅安装本地工具和调用本地工具。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--add-source <SOURCE>`

  添加安装过程中要使用的其他 NuGet 包源。系统会并行访问这些源，而不是按某种优先级顺序依次访问。如果同一个包和版本在多个源中，则选取速度最快的源。有关详细信息，请查看安装 NuGet 包时会发生什么情况？。

- `--tool-manifest <PATH>`

  清单文件的路径。

- `--disable-parallel`

  防止并行还原多个项目。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--no-cache`

  不要缓存包和 HTTP 请求。

- `--interactive`

允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。有关详细信息，请参阅 LoggerVerbosity。

示例

- `dotnet tool restore`

  还原当前目录的本地工具。

请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool run

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

“属性”

`dotnet tool run` -调用本地工具。

摘要

```
dotnet tool run <COMMAND NAME>

dotnet tool run -h|--help
```

描述

`dotnet tool run` 命令将搜索当前目录范围内的工具清单文件。当找到对指定工具的引用时，它会运行该工具。有关详细信息，请参阅 调用本地工具。

自变量

- `COMMAND_NAME`

  要运行的工具的命令名称。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- `dotnet tool run dotnetsay`

  运行 `dotnetsay` 本地工具。

请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool search

2021/11/16 •

本文适用于：✔ .NET 5 SDK 及更高版本

名称

`dotnet tool search` - 搜索已发布到 NuGet 的 .NET 工具。

摘要

```
dotnet tool search [--detail]  [--prerelease]
    [--skip <NUMBER>] [--take <NUMBER>] <SEARCH TERM>

dotnet tool search -h|--help
```

描述

`dotnet tool search` 命令提供了一种方法，使你可以在 NuGet 中搜索可用作 .NET 全局、工具路径或本地工具的工具。该命令搜索工具名称和元数据（如标题、说明和标记）。

该命令使用 NuGet 搜索 API。它筛选 `packageType=dotnettool` 以仅选择 .NET 工具包。

选项

- `--detail`

  显示查询的详细结果。

- `--prerelease`

  包含预发行包。

- `--skip <NUMBER>`

  指定要跳过的查询结果数。用于分页。

- `--take <NUMBER>`

  指定要显示的查询结果数。用于分页。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 在 NuGet.org 中搜索其包名称或描述中具有“格式”的 .NET 工具：

  ```
  dotnet tool search format
  ```

  输出如下所示：

```
Package ID                              Latest Version      Authors
Downloads      Verified
-------------------------------------------------------------------------------------------------------------
-----------------------------------------------------------
dotnet-format                           4.1.131201          Microsoft
496746
bsoa.generator                          1.0.0               Microsoft
533
```

- 在 NuGet.org 中搜索其包名称或元数据中具有“格式”的 .NET 工具，仅显示第一条结果并显示详细视图。

  ```
  dotnet tool search format --take 1 --detail
  ```

  输出如下所示：

  ```
  ----------------
  dotnet-format
  Latest Version: 4.1.131201
  Authors: Microsoft
  Tags:
  Downloads: 496746
  Verified: False
  Description: Command line tool for formatting C# and Visual Basic code files based on .editorconfig
  settings.
  Versions:
          3.0.2 Downloads: 1973
          3.0.4 Downloads: 9064
          3.1.37601 Downloads: 114730
          3.2.107702 Downloads: 13423
          3.3.111304 Downloads: 131195
          4.0.130203 Downloads: 78610
          4.1.131201 Downloads: 145927
  ```

另请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool uninstall

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool uninstall` - 从计算机上卸载指定的 .NET 工具。

摘要

```
dotnet tool uninstall <PACKAGE_NAME> -g|--global

dotnet tool uninstall <PACKAGE_NAME> --tool-path <PATH>

dotnet tool uninstall <PACKAGE_NAME>

dotnet tool uninstall -h|--help
```

描述

`dotnet tool uninstall` 提供一种从计算机上卸载 .NET 工具的方法。若要使用命令，请指定以下选项之一：

- 若要卸载安装在默认位置的全局工具，请使用 `--global` 选项。
- 若要卸载安装在自定义位置的全局工具，请使用 `--tool-path` 选项。
- 若要卸载本地工具，请省略 `--global` 和 `--tool-path` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

自变量

- `PACKAGE_NAME`

  包含要卸载的 .NET 工具的 NuGet 包的名称/ID。你可以使用 dotnet tool list 命令查找包名称。

选项

- `-g|--global`

  指定要从用户范围的安装中删除工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 指定要删除的工具是本地工具。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--tool-path <PATH>`

  指定卸载工具的位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 指定要删除的工具是本地工具。

示例

- `dotnet tool uninstall -g dotnetsay`

  卸载 dotnetsay 全局工具。

- `dotnet tool uninstall dotnetsay --tool-path c:\global-tools`

  从特定 Windows 目录卸载 dotnetsay 全局工具。

- `dotnet tool uninstall dotnetsay --tool-path ~/bin`

  从特定 Linux/macOS 目录卸载 dotnetsay 全局工具。

- `dotnet tool uninstall dotnetsay`

  从当前目录卸载 dotnetsay 全局工具。

请参阅

- .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet tool update

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

“属性”

`dotnet tool update` - 在计算机上更新指定的 .NET 工具。

摘要

```
dotnet tool update <PACKAGE_ID> -g|--global
    [--add-source <SOURCE>] [--configfile <FILE>]
    [--disable-parallel] [--framework <FRAMEWORK>]
    [--ignore-failed-sources] [--interactive] [--no-cache]
    [-v|--verbosity <LEVEL>] [--version <VERSION>]

dotnet tool update <PACKAGE_ID> --tool-path <PATH>
    [--add-source <SOURCE>] [--configfile <FILE>]
    [--disable-parallel] [--framework <FRAMEWORK>]
    [--ignore-failed-sources] [--interactive] [--no-cache]
    [-v|--verbosity <LEVEL>] [--version <VERSION>]

dotnet tool update <PACKAGE_ID> --local
    [--add-source <SOURCE>] [--configfile <FILE>]
    [--disable-parallel] [--framework <FRAMEWORK>]
    [--ignore-failed-sources] [--interactive] [--no-cache]
    [--tool-manifest <PATH>]
    [-v|--verbosity <LEVEL>] [--version <VERSION>]

dotnet tool update -h|--help
```

描述

`dotnet tool update` 命令让你可以将计算机上的 .NET Core 工具更新为包的最新稳定版。此命令卸载并重新安装工具，有效地对工具进行更新。若要使用命令，请指定以下选项之一：

- 若要更新安装在默认位置的全局工具，请使用 `--global` 选项
- 若要更新安装在自定义位置的全局工具，请使用 `--tool-path` 选项。
- 若要更新本地工具，请使用 `--local` 选项。

本地工具从 .NET Core SDK 3.0 开始可用。

自变量

- `PACKAGE_ID`

  包含要更新的 .NET 全局工具的 NuGet 包的名称/ID。你可以使用 dotnet tool list 命令查找包名称。

选项

- `--add-source <SOURCE>`

  添加安装过程中要使用的其他 NuGet 包源。系统会并行访问这些源，而不是按某种优先级顺序依次访

问。如果同一个包和版本在多个源中，则选取速度最快的源。有关详细信息，请查看安装 NuGet 包时会发生什么情况？。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  防止并行还原多个项目。

- `--framework <FRAMEWORK>`

  指定要更新工具的目标框架。

- `-g|--global`

  指定此更新用于用户范围的工具。不能与 `--tool-path` 选项一起使用。省略 `--global` 和 `--tool-path` 均指定要更新的工具是本地工具。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--local`

  更新工具和本地工具清单。不能与 `--global` 选项或 `--tool-path` 选项一起使用。

- `--no-cache`

  不要缓存包和 HTTP 请求。

- `--tool-manifest <PATH>`

  清单文件的路径。

- `--tool-path <PATH>`

  指定全局工具的安装位置。路径可以是绝对的，也可以是相对的。不能与 `--global` 选项一起使用。省略 `--global` 和 `--tool-path` 均指定要更新的工具是本地工具。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。有关详细信息，请参阅 LoggerVerbosity。

- `--version <VERSION>`

  要将工具包更新到的版本范围。这不能用于降级版本，必须首先 `uninstall` 较新的版本。

示例

- `dotnet tool update -g dotnetsay`

更新 dotnetsay 全局工具。

- `dotnet tool update dotnetsay --tool-path c:\global-tools`

  更新位于特定 Windows 目录的 dotnetsay 全局工具。

- `dotnet tool update dotnetsay --tool-path ~/bin`

  更新位于特定 Linux/macOS 目录的 dotnetsay 全局工具。

- `dotnet tool update dotnetsay`

  更新为当前目录安装的 dotnetsay 本地工具。

- `dotnet tool update -g dotnetsay --version 2.0.*`

  将 dotnetsay 全局工具更新到最新的修补程序版本，主版本为 `2`，次要版本为 `0`。

- `dotnet tool update -g dotnetsay --version (2.0.*,2.1.4)`

  将 dotnetsay 全局工具更新到指定范围 `(> 2.0.0 && < 2.1.4)` 内的最低版本，因此将安装版本 `2.1.0`。
  有关语义化版本控制范围的详细信息，请参阅 NuGet 打包版本范围。

请参阅

- .NET 工具
- 语义化版本控制
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

dotnet vstest

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

> **IMPORTANT**
> `dotnet vstest` 命令被 `dotnet test` 取代，后者现在可用于运行程序集。请参阅 `dotnet test`。

“属性”

`dotnet-vstest` - 从指定的程序集运行测试。

摘要

```
dotnet vstest [<TEST_FILE_NAMES>] [--Blame] [--Diag <PATH_TO_LOG_FILE>]
    [--Framework <FRAMEWORK>] [--InIsolation] [-lt|--ListTests <FILE_NAME>]
    [--logger <LOGGER_URI/FRIENDLY_NAME>] [--Parallel]
    [--ParentProcessId <PROCESS_ID>] [--Platform] <PLATFORM_TYPE>
    [--Port <PORT>] [--ResultsDirectory<PATH>] [--Settings <SETTINGS_FILE>]
    [--TestAdapterPath <PATH>] [--TestCaseFilter <EXPRESSION>]
    [--Tests <TEST_NAMES>] [[--] <args>...]]

dotnet vstest -?|--Help
```

描述

`dotnet-vstest` 命令运行 `VSTest.Console` 命令行应用程序以运行自动化单元测试。

自变量

- `TEST_FILE_NAMES`

  从指定的程序集运行测试。用空格分隔多个测试程序集名称。支持通配符。

选项

- `--Blame`

  在意见模式中运行测试。此选项有助于隔离导致测试主机出现故障的有问题的测试。它会在当前目录中创建一个输出文件 (Sequence.xml)，其中捕获了故障前的测试执行顺序 。

- `--Diag <PATH_TO_LOG_FILE>`

  为测试平台启用详细日志。日志被写入到所提供的文件。

- `--Framework <FRAMEWORK>`

  用于测试执行的目标 .NET Framework 版本。有效值的示例为 `.NETFramework,Version=v4.6` 或 `.NETCoreApp,Version=v1.0`。其他支持的值为 `Framework40`、`Framework45`、`FrameworkCore10` 和 `FrameworkUap10`。

- `--InIsolation`

  在隔离的进程中运行测试。虽然这使得 vstest.console.exe 进程不太可能在测试出错时停止，但测试的运行速度会较慢 。

- `-lt|--ListTests <FILE_NAME>`

  列出给定测试容器中所有已发现的测试。

- `--logger <LOGGER_URI/FRIENDLY_NAME>`

  为测试结果指定一个记录器。

  - 若要将测试结果发布到 Team Foundation Server，请使用 `TfsPublisher` 记录器提供程序：

    ```
    /logger:TfsPublisher;
        Collection=<team project collection url>;
        BuildName=<build name>;
        TeamProject=<team project name>
        [;Platform=<Defaults to "Any CPU">]
        [;Flavor=<Defaults to "Debug">]
        [;RunTitle=<title>]
    ```

  - 若要将结果记录到 Visual Studio 测试结果文件 (TRX)，请使用 `trx` 记录器提供程序。此开关使用给定的日志文件名在测试结果目录中创建一个文件。如果未提供 `LogFileName`，将创建唯一的文件名以保留测试结果。

    ```
    /logger:trx [;LogFileName=<Defaults to unique file name>]
    ```

- `--Parallel`

  并行运行测试。默认情况下，计算机上的所有可用内核都可供使用。通过在 runsettings 文件的 `RunConfiguration` 节点下设置 `MaxCpuCount` 属性来指定显式内核数 。

- `--ParentProcessId <PROCESS_ID>`

  父进程的进程 ID 负责启动当前进程。

- `--Platform <PLATFORM_TYPE>`

  用于执行测试的目标平台体系结构。有效值为 `x86`、`x64` 和 `ARM`。

- `--Port <PORT>`

  指定套接字连接和接收事件消息的端口。

- `--ResultsDirectory:<PATH>`

  如果不存在，则将在指定路径中创建测试结果目录。

- `--Settings <SETTINGS_FILE>`

  运行测试时要使用的设置。

- `--TestAdapterPath <PATH>`

  在测试运行中使用来自给定路径(如果有)的自定义测试适配器。

- `--TestCaseFilter <EXPRESSION>`

  运行与给定表达式匹配的测试。`<EXPRESSION>` 的格式为 `<property>Operator<value>[|&<EXPRESSION>]`，其

中运算符是 `=`、`!=` 或 `~` 之一。运算符 `~` 具有"包含"语义，并适用于字符串属性，如 `DisplayName`。括号 `()` 用于组的子表达式。有关详细信息，请参阅 TestCase 筛选器

- `--Tests <TEST_NAMES>`

  运行具有与提供的值匹配的名称的测试。用逗号分隔多个值。

- `-?|--Help`

  打印出有关命令的简短帮助。

- `@<file>`

  有关更多选项，请阅读响应文件。

- `args`

  指定要传递到适配器的额外参数。参数被指定为 `<n>=<v>` 格式的名称值对，其中 `<n>` 是参数名称，`<v>` 是参数值。使用空格分隔多个参数。

示例

在 mytestproject.dll 中运行测试：

```
dotnet vstest mytestproject.dll
```

在 mytestproject.dll 中运行测试，并使用自定义名称导出到自定义文件夹：

```
dotnet vstest mytestproject.dll --logger:"trx;LogFileName=custom_file_name.trx" --
ResultsDirectory:custom/file/path
```

在 mytestproject.dll 和 myothertestproject.exe 中运行测试：

```
dotnet vstest mytestproject.dll myothertestproject.exe
```

运行 `TestMethod1` 测试：

```
dotnet vstest /Tests:TestMethod1
```

运行 `TestMethod1` 和 `TestMethod2` 测试：

```
dotnet vstest /Tests:TestMethod1,TestMethod2
```

请参阅

- VSTest.Console.exe 命令行选项

dotnet workload install

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload install` - 安装可选的工作负载。

摘要

```
dotnet workload install <WORKLOAD_ID>...
    [--configfile <FILE>] [--disable-parallel]
    [--ignore-failed-sources] [--include-previews] [--interactive]
    [--no-cache] [--skip-manifest-update]
    [--source <SOURCE>] [--temp-dir <PATH>] [-v|--verbosity <LEVEL>]

dotnet workload install -?|-h|--help
```

说明

`dotnet workload install` 命令安装一个或多个可选工作负载。可选工作负载可以在 .NET SDK 的基础上安装，以提供对各种应用程序类型（如 .NET MAUI 和 Blazor WebAssembly AOT）的支持。

使用 dotnet workload search 了解可安装的工作负载。

何时在特权模式下运行

对于已安装到受保护目录的 macOS 和 Linux SDK 安装，该命令需要在特权模式下运行（使用 `sudo` 命令）。在 Windows 上，即使 SDK 已安装到 Program Files 目录，也无需在特权模式下运行该命令。对于 Windows，该命令使用该位置的 MSI 安装程序。

结果因 SDK 版本而异

`dotnet workload` 命令在特定 SDK 版本的上下文中运行。假设同时安装了 .NET 6.0.100 SDK 和 .NET 6.0.200 SDK。`dotnet workload` 命令将给出不同的结果，具体取决于你选择的 SDK 版本。此行为适用于主要版本和次要版本以及功能区段差异，而不适用于修补版本差异。例如，.NET SDK 6.0.101 和 6.0.102 给出相同结果，而 6.0.100 和 6.0.200 则给出不同结果。可使用 global.json 文件或 `dotnet workload` 命令的 `--sdk-version` 选项来指定 SDK 版本。

播发清单

工作负载安装所需的资产的名称和版本在清单中进行维护。默认情况下，`dotnet workload install` 命令在安装工作负载之前下载最新的可用清单。然后，清单的本地副本提供查找和下载工作负载的资产所需的信息。

`dotnet workload list` 命令将已安装的工作负载版本与当前可用版本进行比较。当发现有比已安装版本更新的版本可用时，它会在命令输出中播发该事实。从 .NET 6 开始，`dotnet workload list` 中的这些新版本通知将可用。

若要启用这些通知，请下载清单的最新可用版本，并将其存储为播发清单。运行以下任一命令时，这些下载将在后台异步进行。

- dotnet build
- dotnet pack

- dotnet publish
- dotnet restore
- dotnet run
- dotnet test

如果命令在清单下载完成之前完成，则下载将停止。下次运行这些命令时，会再次尝试下载。可设置环境变量来禁用这些后台下载或控制其频率。默认情况下，下载频率每天不会超过一次。

可使用 `--skip-manifest-update` 选项阻止 `dotnet workload install` 命令执行清单下载。

`dotnet workload update` 命令还会下载播发清单。必须通过下载才能了解是否有更新可用，因此没有阻止它们运行的选项。但是，可使用 `--advertising-manifests-only` 选项跳过工作负载更新，仅执行清单下载。此选项从 .NET 6 开始可用。

参数

- `WORKLOAD_ID` ...

  要安装的工作负载 ID 或多个 ID。使用 dotnet workload search 了解可用的工作负载。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  阻止并行还原多个项目。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--include-previews`

  允许预发布工作负载清单。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--no-cache`

  阻止缓存包和 http 请求。

- `--skip-manifest-update`

  跳过更新工作负载清单。工作负载清单定义需要为每个工作负载安装的资产和版本。

- `-s|--source <SOURCE>`

  指定要使用的 NuGet 包源的 URI。此设置会替代 nuget.config 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--temp-dir <PATH>`

  指定用于下载和提取 NuGet 包的临时目录(必须是安全的)。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。如果指定 `detailed` 或 `diagnostic` 详细程度，该命令将显示有关它下载的 Nuget 包的信息。

示例

- 安装 `maui` 工作负载:

```
dotnet workload install maui
```

- 安装 `maui-android` 和 `maui-ios` 工作负载:

```
dotnet workload install maui-android maui-ios
```

dotnet workload list

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload list` - 列出已安装的工作负载。

摘要

```
dotnet workload list [-v|--verbosity <LEVEL>]

dotnet workload list [-?|-h|--help]
```

说明

`dotnet workload list` 命令将列出已安装的所有工作负载。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 列出已安装的工作负载：

  ```
  dotnet workload list
  ```

dotnet workload repair

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload repair` - 修复工作负载安装。

摘要

```
dotnet workload repair
    [--configfile] [--disable-parallel] [--ignore-failed-sources]
    [--interactive] [--no-cache]
    [-s|--source <SOURCE>] [--temp-dir <PATH>]
    [-v|--verbosity <LEVEL>]

dotnet workload repair -?|-h|--help
```

说明

`dotnet workload repair` 命令会重新安装所有已安装的工作负载。工作负载由多个工作负载包组成，它可能出现部分安装成功，其他安装失败的情况。例如，`dotnet workload install` 命令可能因 Internet 连接中断而无法完成安装。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

参数

- `WORKLOAD_ID`

  要修复的工作负载的 ID。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  阻止并行还原多个项目。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--include-previews`

允许预发布工作负载清单。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--no-cache`

  阻止缓存包和 http 请求。

- `-s|--source <SOURCE>`

  指定要使用的 NuGet 包源的 URI。此设置会替代 nuget.config 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--temp-dir <PATH>`

  指定用于下载和提取 NuGet 包的临时目录（必须是安全的）。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 修复所有已安装的工作负载：

```
dotnet workload repair
```

dotnet workload restore

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload restore` - 安装项目或解决方案所需的工作负载。

摘要

```
dotnet workload restore [<PROJECT | SOLUTION>]
    [--configfile <FILE>] [--disable-parallel]
    [--ignore-failed-sources] [--include-previews] [--interactive]
    [--no-cache] [--skip-manifest-update]
    [-s|--source <SOURCE>] [--temp-dir <PATH>] [-v|--verbosity <LEVEL>]

dotnet workload restore -?|-h|--help
```

说明

`dotnet workload restore` 命令会分析项目或解决方案，来确定其所需的工作负载，然后安装所有缺少的工作负载。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

参数

- `PROJECT | SOLUTION`

  要为其安装工作负载的项目或解决方案文件。如果未指定文件，则该命令会在当前目录中搜索一个文件。

选项

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  阻止并行还原多个项目。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--include-previews`

  允许预发布工作负载清单。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--no-cache`

  阻止缓存包和 http 请求。

- `--skip-manifest-update`

  跳过更新工作负载清单。工作负载清单定义需要为每个工作负载安装的资产和版本。

- `-s|--source <SOURCE>`

  指定要使用的 NuGet 包源的 URI。此设置会替代 nuget.config 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--temp-dir <PATH>`

  指定用于下载和提取 NuGet 包的临时目录(必须是安全的)。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 还原 MyApp.csproj 所需的工作负载：

```
dotnet workload restore MyApp.csproj
```

dotnet workload search

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload search` - 搜索可选工作负载。

摘要

```
dotnet workload search [<SEARCH_STRING>] [-v|--verbosity <LEVEL>]

dotnet workload search -?|-h|--help
```

说明

`dotnet workload search` 命令会列出可用工作负载。可通过指定要查找的全部或部分工作负载 ID 来筛选列表。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

参数

- `SEARCH_STRING`

  要搜索的全部或部分工作负载的 ID。例如，如果指定 `maui`，则该命令会列出其中包含 `maui` 的所有工作负载 ID。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。有关详细信息，请参阅 LoggerVerbosity。

示例

- 列出所有可用工作负载：

  ```
  dotnet workload search
  ```

- 列出工作负载 ID 中包含“maui”的所有可用工作负载：

  ```
  dotnet workload search maui
  ```

dotnet workload uninstall

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload uninstall` - 卸载指定的工作负载。

摘要

```
dotnet workload uninstall <WORKLOAD_ID...>

dotnet workload uninstall -?|-h|--help
```

说明

`dotnet workload uninstall` 命令卸载一个或多个工作负载。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

参数

- `WORKLOAD_ID...`

  要卸载的工作负载 ID 或多个 ID。

选项

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

示例

- 卸载 `maui` 工作负载：

  ```
  dotnet workload uninstall maui
  ```

- 卸载 `maui-android` 和 `maui-ios` 工作负载：

  ```
  dotnet workload uninstall maui-android maui-ios
  ```

dotnet workload update

2021/11/16 •

本文适用于：✔ .NET 6 SDK 及更高版本

名称

`dotnet workload update` - 更新已安装的工作负载。

摘要

```
dotnet workload update
    [--advertising-manifests-only]
    [--configfile <FILE>] [--disable-parallel]
    [--from-previous-sdk] [--ignore-failed-sources]
    [--include-previews] [--interactive] [--no-cache]
    [-s|--source <SOURCE>] [--temp-dir <PATH>]
    [-v|--verbosity <LEVEL>]

dotnet workload update -?|-h|--help
```

说明

`dotnet workload update` 命令会将所有已安装的工作负载更新到最新可用版本。针对已更新的工作负载清单查询 Nuget.org。随后更新本地清单，下载已安装的工作负载的新版本，然后删除每个工作负载的所有旧版本。

有关 `dotnet workload` 命令的详细信息，请参阅 dotnet workload install 命令。

选项

- `--advertising-manifests-only`

  下载播发清单，但不更新任何工作负载。

- `--configfile <FILE>`

  要使用的 NuGet 配置文件 (nuget.config)。如果指定，则只使用此文件中的设置。如果不指定，将使用当前目录中的配置文件的层次结构。有关详细信息，请参阅常见的 NuGet 配置。

- `--disable-parallel`

  阻止并行还原多个项目。

- `--from-previous-sdk`

  在更新中包含随以前的 SDK 版本一起安装的工作负载。

- `-?|-h|--help`

  打印出有关如何使用命令的说明。

- `--ignore-failed-sources`

  将包源失败视为警告。

- `--include-previews`

  允许预发布工作负载清单。

- `--interactive`

  允许命令停止并等待用户输入或操作。例如，完成身份验证。

- `--no-cache`

  阻止缓存包和 http 请求。

- `-s|--source <SOURCE>`

  指定要使用的 NuGet 包源的 URI。此设置会替代 nuget.config 文件中指定的所有源。多次指定此选项可以提供多个源。

- `--temp-dir <PATH>`

  指定用于下载和提取 NuGet 包的临时目录(必须是安全的)。

- `-v|--verbosity <LEVEL>`

  设置命令的详细级别。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `minimal`。如果指定 `detailed` 或 `diagnostic` 详细程度，该命令将显示有关它下载的 Nuget 包的信息。

示例

- 更新已安装的工作负载:

  ```
  dotnet workload update
  ```

- 将更新已安装的工作负载所需的资产下载到位于当前目录下 workload-cache 目录中的缓存中。然后从该缓存位置更新已安装的工作负载:

  ```
  dotnet workload update --download-to-cache ./workload-cache
  dotnet workload update --from-cache ./workload-cache
  ```

提升的 Dotnet 命令访问权限

2021/11/16 •

开发人员可根据软件开发最佳做法来编写需要最少权限的软件。但是，某些软件（如性能监视工具）由于操作系统规则，需要管理员权限。以下指南介绍使用 .NET Core 编写此类软件的适用方案。

可以运行以下提升的命令：

- `dotnet tool` 命令，如 dotnet tool install。
- `dotnet run --no-build`
- `dotnet-core-uninstall`

不建议运行其他提升的命令。具体而言，不建议为使用 MSBuild（例如，dotnet restore、dotnet build 和 dotnet run）的命令提升访问权限。主要问题是用户在发出 dotnet 命令后在根帐户和受限帐户之间来回切换时存在权限管理问题。受限用户可能会发现自己无法访问根用户构建的文件。有办法可以解决这种情况，但不一定要使用这些方法。

只要不在根帐户和受限帐户之间来回切换，就能够以根帐户的身份运行命令。例如，Docker 容器默认以根帐户身份运行，因此它们具有此特性。

全局工具安装

以下说明展示了执行下述操作的推荐方法：安装、运行和卸载需要提升权限才能执行的 .NET 工具。

- Windows
- Linux
- macOS

安装工具

如果文件夹 `%ProgramFiles%\dotnet-tools` 已存在，请执行以下操作以检查“用户”组是否有写入或修改该目录的权限：

- 右键单击 `%ProgramFiles%\dotnet-tools` 文件夹并选择“属性”。随即打开“常用属性”对话框。
- 选择“安全性”选项卡。在“组或用户名”下，检查“用户”组是否具有写入或修改目录的权限。
- 如果“用户”组可以写入或修改目录，则在安装工具时使用其他目录名，而不使用 dotnet-tools。

要安装工具，请在提升的提示符下运行以下命令。此操作将在安装期间创建 dotnet-tools 文件夹。

```
dotnet tool install PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools".
```

运行全局工具

选项 1 在提升的提示符中使用完整路径：

```
"%ProgramFiles%\dotnet-tools\TOOLCOMMAND"
```

选项 2 将新创建的文件夹添加到 `%Path%`。只需执行此操作一次。

```
setx Path "%Path%;%ProgramFiles%\dotnet-tools\"
```

然后使用以下命令运行：

```
TOOLCOMMAND
```

卸载全局工具

在提升的提示符处，键入下列命令：

```
dotnet tool uninstall PACKAGEID --tool-path "%ProgramFiles%\dotnet-tools"
```

本地工具

本地工具的作用域按用户和个子目录树来限定。执行特权运行后，本地工具将受限的用户环境共享给提升的环境。在 Linux 和 macOS 中，这会导致将文件设置为仅限根用户访问。如果用户切换回受限帐户，则用户无法再访问或写入文件。因此，不建议将必须提升的工具安装为本地工具。建议使用 `--tool-path` 选项和上述全局工具指南。

开发过程中的提升

在开发过程中，可能需要提升访问权限才能测试应用程序。（例如）IoT 应用就常存在这种情况。建议在构建应用程序时不要进行提升，而是在运行时使用提升。有几种模式，如下所示：

- 使用生成的可执行文件（它提供最佳的启动性能）：

  ```
  dotnet build
  sudo ./bin/Debug/netcoreapp3.0/APPLICATIONNAME
  ```

- 使用 dotnet run 命令与 `–no-build` 标志，以避免生成新的二进制文件：

  ```
  dotnet build
  sudo dotnet run --no-build
  ```

请参阅

- .NET 工具概述

如何为 .NET CLI 启用 Tab 自动补全

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

本文介绍了如何为四种 shell（PowerShell、Bash、zsh 和 fish）配置 Tab 自动补全。对于其他 shell，请参阅相关文档，了解如何配置 tab 自动补全。

设置完成后，通过在 shell 中键入 `dotnet` 命令，然后按下 Tab 键即可触发 .NET CLI 的 Tab 自动补全。当前命令行将发送到 `dotnet complete` 命令，结果将由 shell 处理。可以通过直接向 `dotnet complete` 命令发送内容来测试结果而无需启用 tab 自动补全。例如：

```
> dotnet complete "dotnet a"
add
clean
--diagnostics
migrate
pack
```

如果该命令不起作用，请确保已安装 .NET Core 2.0 SDK 或更高版本。如果已安装，但该命令仍不起作用，请确保 `dotnet` 命令解析为 .NET Core 2.0 SDK 及更高版本。使用 `dotnet --version` 命令查看当前路径解析为的 `dotnet` 的版本。有关详细信息，请参阅选择要使用的 .NET 版本。

示例

下面是 tab 自动补全提供的一些示例：

“	“”	“
`dotnet a⇥`	`dotnet add`	`add` 是第一项子命令，按字母排序。
`dotnet add p⇥`	`dotnet add --help`	Tab 自动补全匹配子字符串，`--help` 首先按字母顺序排列。
`dotnet add p⇥⇥`	`dotnet add package`	第二次按 Tab 将显示下一条建议。
`dotnet add package Microsoft⇥`	`dotnet add package Microsoft.ApplicationInsights.Web`	结果按字母顺序返回。
`dotnet remove reference ⇥`	`dotnet remove reference ..\..\src\OmniSharp.DotNet\OmniSharp.DotNet.csproj`	Tab 自动补全是可识别的项目文件。

PowerShell

若要将 Tab 自动补全添加到适用于 .NET CLI 的 PowerShell，请创建或编辑存储在变量 `$PROFILE` 中的配置文件。有关详细信息，请参阅如何创建配置文件和配置文件和执行策略。

将以下代码添加到配置文件中：

```
# PowerShell parameter completion shim for the dotnet CLI
Register-ArgumentCompleter -Native -CommandName dotnet -ScriptBlock {
     param($commandName, $wordToComplete, $cursorPosition)
         dotnet complete --position $cursorPosition "$wordToComplete" | ForEach-Object {
            [System.Management.Automation.CompletionResult]::new($_, $_, 'ParameterValue', $_)
         }
 }
```

bash

若要将 Tab 自动补全添加到适用于 .NET CLI 的 bash shell，请将以下代码添加到 `.bashrc` 文件：

```
# bash parameter completion for the dotnet CLI

_dotnet_bash_complete()
{
  local word=${COMP_WORDS[COMP_CWORD]}

  local completions
  completions="$(dotnet complete --position "${COMP_POINT}" "${COMP_LINE}" 2>/dev/null)"
  if [ $? -ne 0 ]; then
    completions=""
  fi

  COMPREPLY=( $(compgen -W "$completions" -- "$word") )
}

complete -f -F _dotnet_bash_complete dotnet
```

zsh

若要将 Tab 自动补全添加到适用于 .NET CLI 的 zsh shell，请将以下代码添加到 `.zshrc` 文件：

```
# zsh parameter completion for the dotnet CLI

_dotnet_zsh_complete()
{
  local completions=("$(dotnet complete "$words")")

  reply=( "${(ps:\n:)completions}" )
}

compctl -K _dotnet_zsh_complete dotnet
```

fish

若要向 .NET CLI 的 fish shell 添加 Tab 自动补全，请将以下代码添加到 `config.fish` 文件中：

```
complete -f -c dotnet -a "(dotnet complete)"
```

在持续集成 (CI) 中使用 .NET SDK 和工具

2021/11/16 •

本文档概述了如何在生成服务器上使用 .NET SDK 及其工具。.NET 工具集既能以交互方式运行（当开发人员在命令提示符处键入命令时），也可以自动运行（当持续集成 (CI) 服务器运行生成脚本时）。命令、选项、输入和输出都相同，可通过提供的唯一内容来获取用于生成应用的工具和系统。本文档重点介绍了 CI 工具获取方案，并提供了有关如何设计和构建生成脚本的建议。

CI 生成服务器的安装选项

使用本机安装程序

本机安装程序适用于 macOS、Linux 和 Windows。安装程序需要拥有对生成服务器的管理员 (sudo) 访问权限。使用本机安装程序的优势在于，可以安装运行工具所需的全部本机依赖项。本机安装程序还可以在整个系统内安装 SDK。

macOS 用户应使用 PKG 安装程序。在 Linux 上，可选择使用基于源的包管理器（如用于 Ubuntu 的 apt-get 或用于 CentOS 的 yum），也可以选择使用包本身（即 DEB 或 RPM）。在 Windows 上，使用 MSI 安装程序。

有关最新的稳定二进制文件，请参阅 .NET 下载。若要使用最新（但可能不稳定）的预览版工具，请使用 dotnet/core-sdk GitHub 存储库中提供的链接。对于 Linux 发行版本，可以使用 `tar.gz` 存档（亦称为 `tarballs`）；使用存档中的安装脚本来安装 .NET Core。

使用安装程序脚本

使用安装程序脚本，可以在生成服务器上执行非管理员安装，并能轻松实现自动化，以便获取工具。安装程序脚本负责下载并将工具提取到默认或指定位置，以供使用。还可以指定要安装的工具版本，以及是要安装整个 SDK，还是仅安装共享运行时。

安装程序脚本在开始生成时自动运行，以提取和安装相应版本的 SDK。相应版本 是指生成项目所需的任意 SDK 版本。使用安装程序脚本，可以在服务器的本地目录中安装 SDK，并能从安装位置运行工具，还可以在生成后进行清理（或让 CI 服务进行清理）。这样，可以封装和隔离整个生成进程。有关安装脚本参考，请参阅 dotnet-install 一文。

> **NOTE**
> Azure DevOps Services
>
> 使用安装程序脚本时，不会自动安装本机依赖项。如果操作系统没有本机依赖项，必须手动安装。有关详细信息，请参阅 .NET 依赖项和要求。

CI 安装示例

此部分介绍了如何使用 PowerShell 或 bash 脚本进行手动安装，同时还介绍了多个服务型软件 (SaaS) CI 解决方案。涵盖的 SaaS CI 解决方案包括 Travis CI、AppVeyor 和 Azure Pipelines。

手动安装

每个 SaaS 服务都有自己的生成进程创建和配置方法。如果使用与所列不同的 SaaS 解决方案，或需要超越预封装支持范围的自定义设置，至少必须执行一些手动配置。

一般来说，手动安装需要获取一个版本的工具（或最新每日版工具），再运行生成脚本。可以使用 PowerShell 或 bash 脚本安排 .NET 命令，也可以使用概述生成进程的项目文件。业务流程部分详细介绍了这些选项。

创建执行手动 CI 生成服务器安装的脚本后，在开发计算机上使用它来生成本地代码以供测试。确认此脚本可以在本地正常运行后，将它部署到 CI 生成服务器。下面是一相对简单的 PowerShell 脚本，说明了如何获取 .NET SDK，以及如何将它安装到 Windows 生成服务器上：

```
$ErrorActionPreference="Stop"
$ProgressPreference="SilentlyContinue"

# $LocalDotnet is the path to the locally-installed SDK to ensure the
#   correct version of the tools are executed.
$LocalDotnet=""
# $InstallDir and $CliVersion variables can come from options to the
#   script.
$InstallDir = "./cli-tools"
$CliVersion = "1.0.1"

# Test the path provided by $InstallDir to confirm it exists. If it
#   does, it's removed. This is not strictly required, but it's a
#   good way to reset the environment.
if (Test-Path $InstallDir)
{
    rm -Recurse $InstallDir
}
New-Item -Type "directory" -Path $InstallDir

Write-Host "Downloading the CLI installer..."

# Use the Invoke-WebRequest PowerShell cmdlet to obtain the
#   installation script and save it into the installation directory.
Invoke-WebRequest `
    -Uri "https://dot.net/v1/dotnet-install.ps1" `
    -OutFile "$InstallDir/dotnet-install.ps1"

Write-Host "Installing the CLI requested version ($CliVersion) ..."

# Install the SDK of the version specified in $CliVersion into the
#   specified location ($InstallDir).
& $InstallDir/dotnet-install.ps1 -Version $CliVersion `
    -InstallDir $InstallDir

Write-Host "Downloading and installation of the SDK is complete."

# $LocalDotnet holds the path to dotnet.exe for future use by the
#   script.
$LocalDotnet = "$InstallDir/dotnet"

# Run the build process now. Implement your build script here.
```

在此脚本末尾提供生成进程的实现代码。此脚本先获取工具，再执行生成进程。对于 UNIX 计算机，下面的 bash 脚本以类似方式执行 PowerShell 脚本中所述的操作：

```
#!/bin/bash
INSTALLDIR="cli-tools"
CLI_VERSION=1.0.1
DOWNLOADER=$(which curl)
if [ -d "$INSTALLDIR" ]
then
    rm -rf "$INSTALLDIR"
fi
mkdir -p "$INSTALLDIR"
echo Downloading the CLI installer.
$DOWNLOADER https://dot.net/v1/dotnet-install.sh > "$INSTALLDIR/dotnet-install.sh"
chmod +x "$INSTALLDIR/dotnet-install.sh"
echo Installing the CLI requested version $CLI_VERSION. Please wait, installation may take a few minutes.
"$INSTALLDIR/dotnet-install.sh" --install-dir "$INSTALLDIR" --version $CLI_VERSION
if [ $? -ne 0 ]
then
    echo Download of $CLI_VERSION version of the CLI failed. Exiting now.
    exit 0
fi
echo The CLI has been installed.
LOCALDOTNET="$INSTALLDIR/dotnet"
# Run the build process now. Implement your build script here.
```

Travis CI

可以将 Travis CI 配置为使用 `csharp` 语言和 `dotnet` 键安装 .NET SDK。有关详细信息，请参阅 Travis CI 官方文档生成 C#、F# 或 Visual Basic 项目。请注意，访问 Travis CI 信息时，社区维护的 `language: csharp` 语言标识符适用于所有 .NET 语言，包括 F# 和 Mono。

Travis CI 可同时在生成矩阵 中运行 macOS 和 Linux 作业。在生成矩阵中，可以指定运行时、环境和排除项/包含项的组合，从而涵盖应用的生成组合。有关详细信息，请参阅 Travis CI 文档中的自定义生成一文。基于 MSBuild 的工具在包中添加 LTS (1.0.x) 和最新 (1.1.x) 运行时；因此，通过安装 SDK，可以收到执行生成所需的一切。

AppVeyor

AppVeyor 使用 `Visual Studio 2017` 生成辅助角色映像安装 .NET Core 1.0.1 SDK。提供具有不同版本的 .NET SDK 的其他生成映像。有关详细信息，请参阅 AppVeyor 文档中的 appveyor.yml 示例和生成辅助角色映像一文。

.NET SDK 二进制文件通过安装脚本下载并解压缩到子目录，再添加到 `PATH` 环境变量。添加生成矩阵可以运行包含多个版本 .NET SDK 的集成测试：

```
environment:
  matrix:
    - CLI_VERSION: 1.0.1
    - CLI_VERSION: Latest

install:
  # See appveyor.yml example for install script
```

Azure DevOps Services

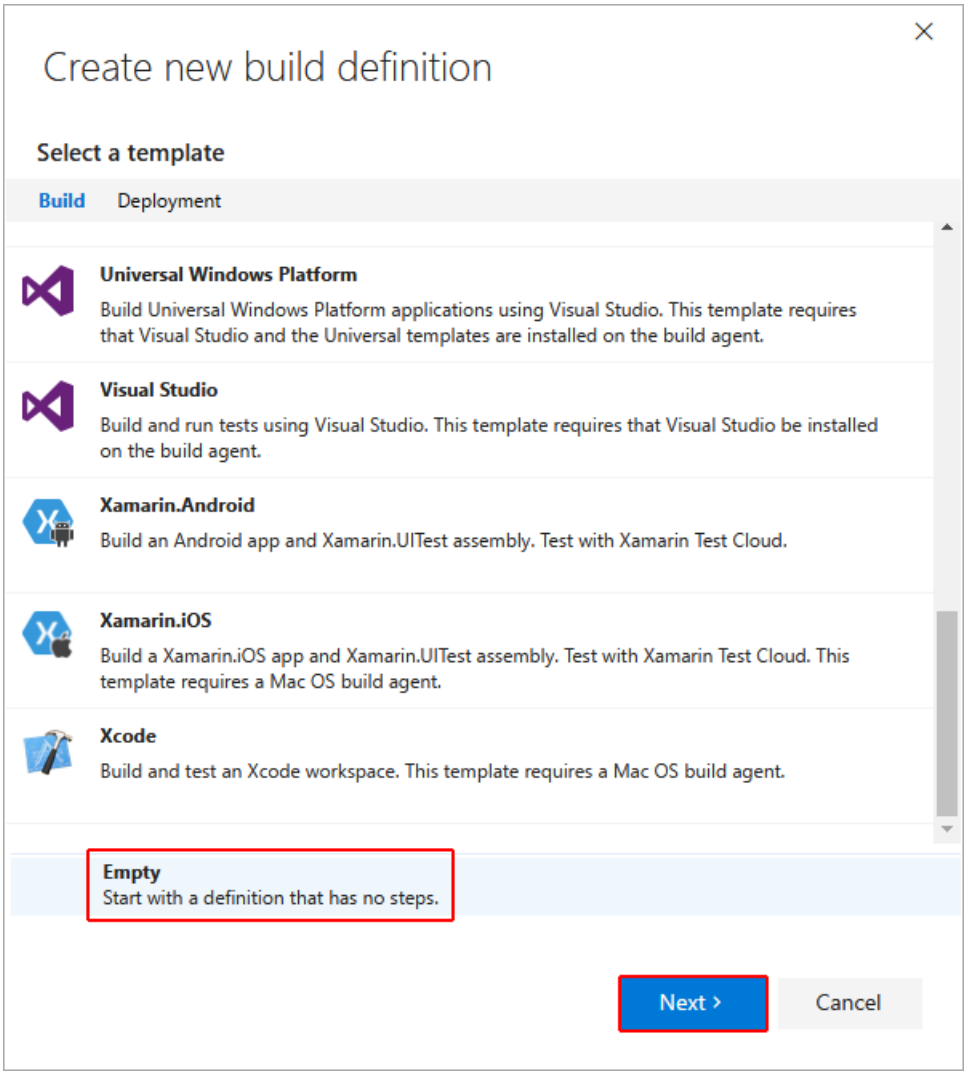
将 Azure DevOps Services 配置为使用以下方法之一生成 .NET 项目：

1. 使用命令运行手动安装步骤中的脚本。
2. 创建包含多个 Azure DevOps Services 内置生成任务（配置为使用 .NET 工具）的生成。

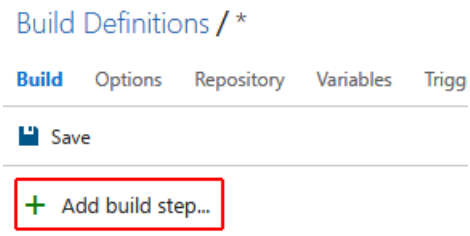
这两种均为有效解决方案。使用手动安装脚本，可以控制收到的工具版本，因为工具是作为生成的一部分进行下载。此生成是通过必须创建的脚本进行运行。本文仅涉及手动选项。有关使用 Azure DevOps Services 生成任务撰写生成的详细信息，请参阅 Azure Pipelines 一文。

若要在 Azure DevOps Services 中使用手动安装脚本，请新建生成定义，并指定要对生成步骤运行的脚本。为此，请使用 Azure DevOps Services 用户界面：

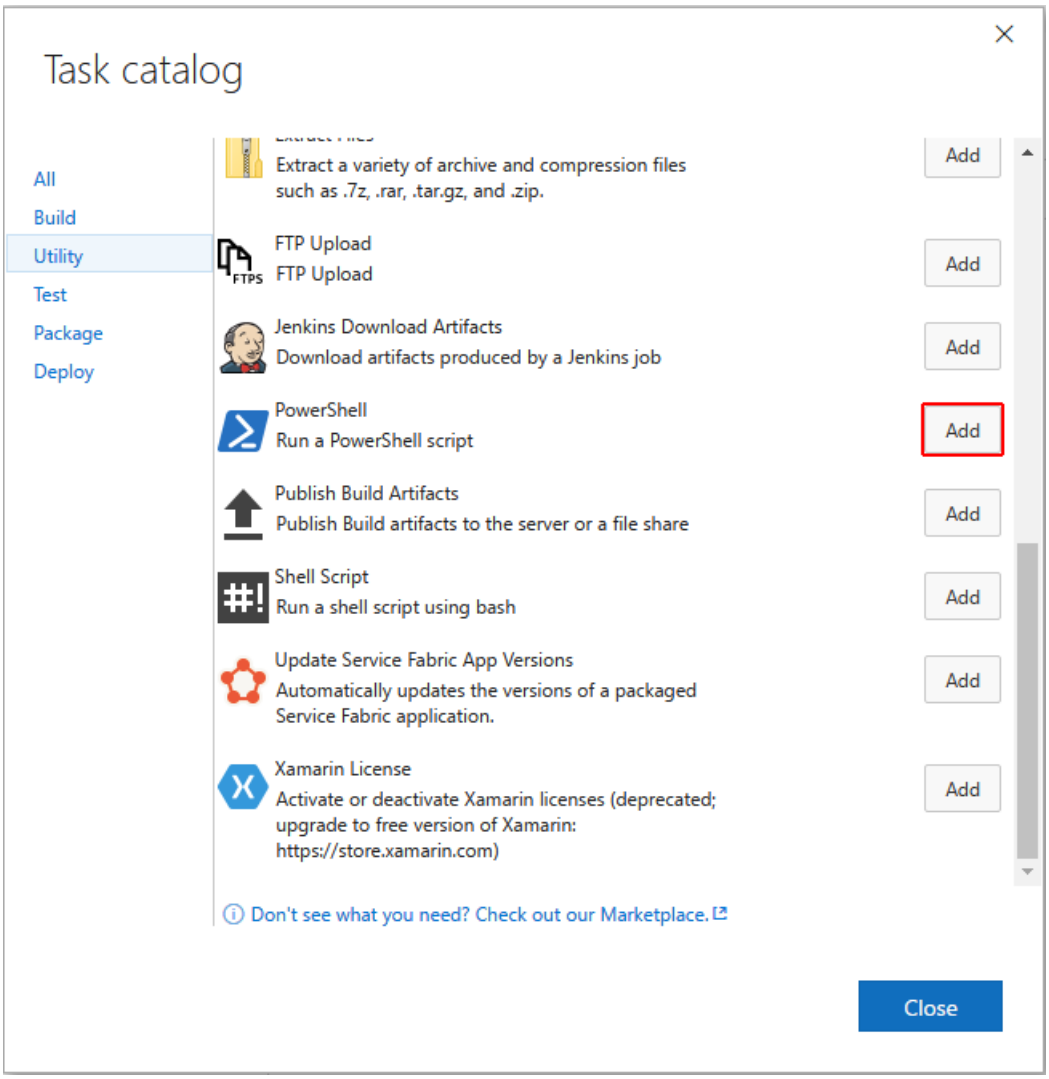
1. 首先，新建生成定义。到达可以定义要创建的生成类型的屏幕后，选择“空”选项。



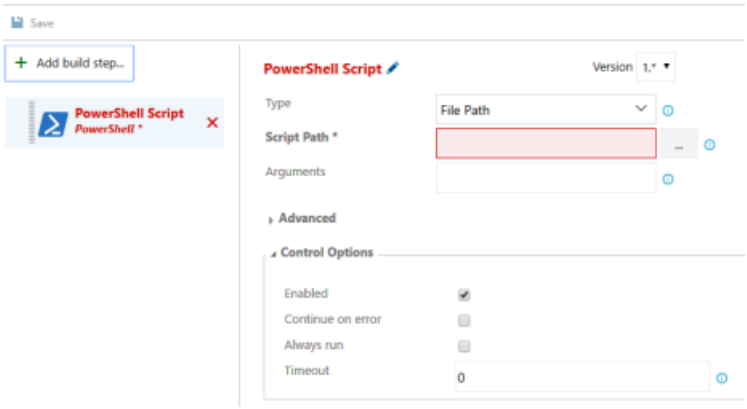
2. 配置要生成的存储库后，将转到生成定义。选择“添加生成步骤”：



3. 此时，系统会显示“任务目录”。此目录包含在生成中使用的任务。由于已有脚本，因此选择“PowerShell: 运行 PowerShell 脚本”旁边的“添加”按钮。



4. 配置生成步骤。从要生成的存储库中添加脚本：



安排生成

本文档的大部分内容介绍了如何获取 .NET Core 工具和配置各种 CI 服务，并未介绍如何安排或实际生成 .NET Core 代码。具体如何构建生成进程取决于许多因素，我们无法在本文中笼统概述。有关使用每种技术安排生成的详细信息，请浏览 Travis CI、AppVeyor、和 Azure Pipelines 文档集中提供的资源和示例。

使用 .NET 工具构建 .NET 代码生成进程的两种常规方法是，直接使用 MSBuild 或使用 .NET 命令行命令。应采用哪种方法取决于对方法的熟悉程度和复杂性取舍。使用 MSBuild，可以将生成进程表达为任务和目标，但需要学习 MSBuild 项目文件语法，这增加了复杂性。使用 .NET 命令行工具可能更为简单，但需要在 `bash` 或 PowerShell 等脚本语言中编写业务流程逻辑。

另请参阅

- .NET 下载 - Linux

使用 .NET CLI 开发库

2021/11/16 •

本文介绍如何使用 .NET CLI 编写 .NET 的库。CLI 提供可跨任何支持的 OS 工作的高效低级别体验。仍可使用 Visual Studio 生成库，如果你首选这种体验，请参阅 Visual Studio 指南。

先决条件

需要在计算机上安装 .NET SDK。

对于本文档中处理 .NET Framework 版本的部分，需要在 Windows 计算机上安装 .NET Framework。

此外，如果想要支持较旧的 .NET Framework 目标，需要从 .NET Framework 下载页安装目标包或开发人员工具包。请参阅此表：

.NET FRAMEWORK 版	说明
4.6.1	.NET Framework 4.6.1 目标包
4.6	.NET Framework 4.6 目标包
4.5.2	.NET Framework 4.5.2 开发人员工具包
4.5.1	.NET Framework 4.5.1 开发人员工具包
4.5	适用于 Windows 8 的 Windows 软件开发工具包
4.0	Windows SDK for Windows 7 和 .NET Framework 4
2.0、3.0 和 3.5	.NET Framework 3.5 SP1 运行时（或 Windows 8+ 版本）

如何以 .NET 5.0 或 .NET Standard 为目标

你可以通过将项目的目标框架添加到项目文件（.csproj 或 .fsproj）来控制项目的目标框架 。有关如何选择以 .NET 5.0 还是 .NET Standard 为目标的指导，请参阅 .NET 5 和 .NET Standard。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>netstandard2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

如果希望面向 .NET Framework 版本 4.0 或更低版本，或者要使用 .NET Framework 中提供但 .NET Standard 中不提供的 API（例如 `System.Drawing`），请阅读以下部分，了解如何设定多目标。

如何面向 .NET framework

> **NOTE**
> 这些说明假定计算机上安装有 .NET Framework。请参阅先决条件 获取安装的依赖项。

请记住，此处使用的某些 .NET Framework 版本不再受支持。有关不受支持的版本信息，请参阅 .NET Framework 支持生命周期策略常见问题。

如果要达到最大数量的开发人员和项目，可将 .NET Framework 4.0 用作基线目标。若要以 .NET Framework 为目标，首先使用与要支持的 .NET Framework 版本相对应的正确目标框架名字对象 (TFM)。

.NET FRAMEWORK	TFM
.NET Framework 2.0	`net20`
.NET Framework 3.0	`net30`
.NET Framework 3.5	`net35`
.NET Framework 4.0	`net40`
.NET Framework 4.5	`net45`
.NET Framework 4.5.1	`net451`
.NET Framework 4.5.2	`net452`
.NET Framework 4.6	`net46`
.NET Framework 4.6.1	`net461`
.NET Framework 4.6.2	`net462`
.NET Framework 4.7	`net47`
.NET Framework 4.8	`net48`

然后将此 TFM 插入项目文件的 `TargetFramework` 部分。例如，下面展示了如何编写面向 .NET Framework 4.0 的库：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net40</TargetFramework>
  </PropertyGroup>
</Project>
```

大功告成！ 虽然此库仅针对 .NET Framework 4 编译，但可在较新版本的 .NET Framework 上使用此库。

如何设定多目标

> **NOTE**
> 以下说明假定计算机上安装有 .NET Framework。请参阅先决条件部分，了解需要安装哪些依赖项以及在何处下载。

如果项目同时支持 .NET Framework 和 .NET，可能需要以较旧版本的 .NET Framework 为目标。在此方案中，如果要为较新目标使用较新的 API 和语言构造，请在代码中使用 `#if` 指令。可能还需要为要面向的每个平台添加不同的包和依赖项，以包含每种情况所需的不同 API。

例如，假设有一个库，它通过 HTTP 执行联网操作。对于 .NET Standard 和 .NET Framework 版本 4.5 或更高版本，可从 `System.Net.Http` 命名空间使用 `HttpClient` 类。但是，.NET Framework 的早期版本没有 `HttpClient` 类，因此可对早期版本使用 `System.Net` 命名空间中的 `WebClient` 类。

项目文件可能如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.0 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net40'">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Need to conditionally bring in references for the .NET Framework 4.5 target -->
  <ItemGroup Condition="'$(TargetFramework)' == 'net45'">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>
</Project>
```

在此处可看到三项主要更改：

1. `TargetFramework` 节点已替换为 `TargetFrameworks`，其中表示了三个 TFM。
2. `net40` 目标有一个 `<ItemGroup>` 节点，拉取一个 .NET Framework 引用。
3. `net45` 目标中有一个 `<ItemGroup>` 节点，拉取两个 .NET Framework 引用。

生成系统可识别以下用在 `#if` 指令中的处理器符号：

[illegible]	[illegible]	.NET 5 [illegible] SDK [illegible]
.NET Framework	`NETFRAMEWORK`, `NET48`, `NET472`, `NET471`, `NET47`, `NET462`, `NET461`, `NET46`, `NET452`, `NET451`, `NET45`, `NET40`, `NET35`, `NET20`	`NET48_OR_GREATER`, `NET472_OR_GREATER`, `NET471_OR_GREATER`, `NET47_OR_GREATER`, `NET462_OR_GREATER`, `NET461_OR_GREATER`, `NET46_OR_GREATER`, `NET452_OR_GREATER`, `NET451_OR_GREATER`, `NET45_OR_GREATER`, `NET40_OR_GREATER`, `NET35_OR_GREATER`, `NET20_OR_GREATER`

[illegible]	[illegible]	.NET 5 [illegible] SDK [illegible]
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`, `NETSTANDARD1_1`, `NETSTANDARD1_0`	`NETSTANDARD2_1_OR_GREATER`, `NETSTANDARD2_0_OR_GREATER`, `NETSTANDARD1_6_OR_GREATER`, `NETSTANDARD1_5_OR_GREATER`, `NETSTANDARD1_4_OR_GREATER`, `NETSTANDARD1_3_OR_GREATER`, `NETSTANDARD1_2_OR_GREATER`, `NETSTANDARD1_1_OR_GREATER`, `NETSTANDARD1_0_OR_GREATER`
.NET 5 及更高版本(和 .NET Core)	`NET`, `NET6_0`, `NET6_0_ANDROID`, `NET6_0_IOS`, `NET6_0_MACOS`, `NET6_0_MACCATALYST`, `NET6_0_TVOS`, `NET6_0_WINDOWS`, `NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`	`NET6_0_OR_GREATER`, `NET6_0_ANDROID_OR_GREATER`, `NET6_0_IOS_OR_GREATER`, `NET6_0_MACOS_OR_GREATER`, `NET6_0_MACCATALYST_OR_GREATER`, `NET6_0_TVOS_OR_GREATER`, `NET6_0_WINDOWS_OR_GREATER`, `NET5_0_OR_GREATER`, `NETCOREAPP_OR_GREATER`, `NETCOREAPP3_1_OR_GREATER`, `NETCOREAPP3_0_OR_GREATER`, `NETCOREAPP2_2_OR_GREATER`, `NETCOREAPP2_1_OR_GREATER`, `NETCOREAPP2_0_OR_GREATER`, `NETCOREAPP1_1_OR_GREATER`, `NETCOREAPP1_0_OR_GREATER`

NOTE

- 无论目标版本是什么，都将定义无版本符号。
- 仅针对目标版本定义特定于版本的符号。
- 为目标版本和所有早期版本定义 `<framework>_OR_GREATER` 符号。例如，如果针对 .NET Framework 2.0，则会定义以下符号：`NET_2_0`、`NET_2_0_OR_GREATER`、`NET_1_1_OR_GREATER` 和 `NET_1_0_OR_GREATER`。

以下是使用每目标条件编译的示例：

```
using System;
using System.Text.RegularExpressions;
#if NET40
// This only compiles for the .NET Framework 4 targets
using System.Net;
#else
 // This compiles for all other targets
using System.Net.Http;
using System.Threading.Tasks;
#endif

namespace MultitargetLib
{
    public class Library
    {
#if NET40
        private readonly WebClient _client = new WebClient();
        private readonly object _locker = new object();
#else
        private readonly HttpClient _client = new HttpClient();
#endif

#if NET40
        // .NET Framework 4.0 does not have async/await
        public string GetDotNetCount()
        {
            string url = "https://www.dotnetfoundation.org/";

            var uri = new Uri(url);

            string result = "";

            // Lock here to provide thread-safety.
            lock(_locker)
            {
                result = _client.DownloadString(uri);
            }

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"Dotnet Foundation mentions .NET {dotNetCount} times!";
        }
#else
        // .NET Framework 4.5+ can use async/await!
        public async Task<string> GetDotNetCountAsync()
        {
            string url = "https://www.dotnetfoundation.org/";

            // HttpClient is thread-safe, so no need to explicitly lock here
            var result = await _client.GetStringAsync(url);

            int dotNetCount = Regex.Matches(result, ".NET").Count;

            return $"dotnetfoundation.org mentions .NET {dotNetCount} times in its HTML!";
        }
#endif
    }
}
```

如果使用 `dotnet build` 生成此项目，则在 `bin/` 文件夹下有三个目录：

```
net40/
net45/
netstandard2.0/
```

其中每个目录都包含每个目标的 `.dll` 文件。

如何在 .NET 上测试库

能够跨平台进行测试至关重要。可使用现成的 xUnit 或 MSTest。它们都非常适合在 .NET 上对库进行单元测试。如何使用测试项目设置解决方案取决于解决方案的结构。下面的示例假设测试和源目录位于同一顶级目录下。

> **NOTE**
> 此示例将使用某些 .NET CLI 命令。有关详细信息，请参阅 dotnet new 和 dotnet sln。

1. 设置解决方案。可使用以下命令实现此目的：

   ```
   mkdir SolutionWithSrcAndTest
   cd SolutionWithSrcAndTest
   dotnet new sln
   dotnet new classlib -o MyProject
   dotnet new xunit -o MyProject.Test
   dotnet sln add MyProject/MyProject.csproj
   dotnet sln add MyProject.Test/MyProject.Test.csproj
   ```

   这将创建多个项目，并一个解决方案中将这些项目链接在一起。`SolutionWithSrcAndTest` 的目录应如下所示：

   ```
   /SolutionWithSrcAndTest
   |__SolutionWithSrcAndTest.sln
   |__MyProject/
   |__MyProject.Test/
   ```

2. 导航到测试项目的目录，然后添加对 `MyProject` 中的 `MyProject.Test` 的引用。

   ```
   cd MyProject.Test
   dotnet add reference ../MyProject/MyProject.csproj
   ```

3. 还原包和生成项目：

   ```
   dotnet restore
   dotnet build
   ```

4. 执行 `dotnet test` 命令，验证 xUnit 是否在运行。如果选择使用 MSTest，则应改为运行 MSTest 控制台运行程序。

大功告成！ 现在可以使用命令行工具跨所有平台测试库。若要继续测试，现已设置好了所有内容，测试库将非常简单：

1. 对库进行更改。
2. 使用 `dotnet test` 命令在测试目录中从命令行运行测试。

调用 `dotnet test` 命令时，将自动重新生成代码。

如何使用多个项目

对于较大的库，通常需要将功能置于不同项目中。

假设要生成一个可以惯用的 C# 和 F# 使用的库。这意味着库的使用者可通过对 C# 或 F# 来说很自然的方式来使用它。例如，在 C# 中，了能会这样使用库：

```
using AwesomeLibrary.CSharp;

public Task DoThings(Data data)
{
    var convertResult = await AwesomeLibrary.ConvertAsync(data);
    var result = AwesomeLibrary.Process(convertResult);
    // do something with result
}
```

在 F# 中可能是这样：

```
open AwesomeLibrary.FSharp

let doWork data = async {
    let! result = AwesomeLibrary.AsyncConvert data // Uses an F# async function rather than C# async method
    // do something with result
}
```

这样的使用方案意味着被访问的 API 必须具有用于 C# 和 F# 的不同结构。通常的方法是将库的所有逻辑因子转化到核心项目中，C# 和 F# 项目定义调用到核心项目的 API 层。该部分的其余部分将使用以下名称：

- **AwesomeLibrary.Core** - 核心项目，其中包含库的所有逻辑
- **AwesomeLibrary.CSharp** - 具有打算在 C# 中使用的公共 API 的项目
- **AwesomeLibrary.FSharp** - 具有打算在 F# 中使用的公共 API 的项目

可在终端运行下列命令，生成与下列指南相同的结构：

```
mkdir AwesomeLibrary && cd AwesomeLibrary
dotnet new sln
mkdir AwesomeLibrary.Core && cd AwesomeLibrary.Core && dotnet new classlib
cd ..
mkdir AwesomeLibrary.CSharp && cd AwesomeLibrary.CSharp && dotnet new classlib
cd ..
mkdir AwesomeLibrary.FSharp && cd AwesomeLibrary.FSharp && dotnet new classlib -lang "F#"
cd ..
dotnet sln add AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
dotnet sln add AwesomeLibrary.CSharp/AwesomeLibrary.CSharp.csproj
dotnet sln add AwesomeLibrary.FSharp/AwesomeLibrary.FSharp.fsproj
```

这将添加上述三个项目和将它们链接在一起的解决方案文件。创建解决方案文件并链接项目后，可从顶级还原和生成项目。

项目到项目的引用

引用项目的最佳方式是使用 .NET CLI 添加项目引用。在 AwesomeLibrary.CSharp 和 AwesomeLibrary.FSharp 项目目录中，可运行下列命令：

```
dotnet add reference ../AwesomeLibrary.Core/AwesomeLibrary.Core.csproj
```

AwesomeLibrary.CSharp 和 AwesomeLibrary.FSharp 的项目文件现在需要将 AwesomeLibrary.Core 作为 `ProjectReference` 目标引用。可通过检查项目文件和查看其中的下列内容来进行验证：

```
<ItemGroup>
  <ProjectReference Include="..\AwesomeLibrary.Core\AwesomeLibrary.Core.csproj" />
</ItemGroup>
```

如果不想使用 .NET CLI，可手动将此部分添加到每个项目文件。

结构化解决方案

多项目解决方案的另一个重要方面是建立良好的整体项目结构。可根据自己的喜好随意组织代码，只要使用 `dotnet sln add` 将每个项目链接到解决方案文件，就可在解决方案级别运行 `dotnet restore` 和 `dotnet build`。

教程：创建项模板

2021/11/16 •

使用 .NET，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第一部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 为项模板创建类
- 创建模板配置文件夹和文件
- 从文件路径安装模板
- 测试项模板
- 卸载项模板

先决条件

- .NET 5.0 SDK 或更高版本。
- 阅读参考文章为 dotnet new 自定义模板。

  参考文章介绍了有关模板的基础知识，以及如何将它们组合在一起。其中一些信息将在本文中重复出现。
- 打开终端并导航到 working\templates 文件夹。

创建所需的文件夹

本系列使用包含模板源的“working 文件夹”和用于测试模板的“testing 文件夹”。working 文件夹和 testing 文件夹应位于同一父文件夹下。

首先，创建父文件夹，名称无关紧要。然后，创建一个名为“working” 的子文件夹。在 working 文件夹内，创建一个名为“templates” 的子文件夹。

接下来，在名为“test” 的父文件夹下创建一个文件夹。文件夹结构应如下所示。

```
parent_folder
├──test
└──working
    └──templates
```

创建项模板

项模板是包含一个或多个文件的特定类型的模板。当你想要生成类似于配置、代码或解决方案文件的内容时，这些类型的模板非常有用。在本例中，你将创建一个类，该类将扩展方法添加到字符串类型中。

在终端中，导航到 working\templates 文件夹，并创建一个名为“extensions” 的新子文件夹。进入文件夹。

```
working
└──templates
    └──extensions
```

创建一个名为“CommonExtensions.cs” 的新文件，并使用你喜爱的文本编辑器打开它。此类将提供一个用于反转字符串内容的名为 `Reverse` 的扩展方法。粘贴以下代码并保存文件：

```
using System;

namespace System
{
    public static class StringExtensions
    {
        public static string Reverse(this string value)
        {
            var tempArray = value.ToCharArray();
            Array.Reverse(tempArray);
            return new string(tempArray);
        }
    }
}
```

现在你已经创建了模板的内容，需要在模板的根文件夹中创建模板配置。

创建模板配置

模板通过模板根目录中的特殊文件夹和配置文件进行识别。在本教程中，你的模板文件夹位于 working\templates\extensions 。

创建模板时，除特殊配置文件夹外，模板文件夹中的所有文件和文件夹都作为模板的一部分包含在内。此配置文件夹名为".template.config" 。

首先，创建一个名为".template.config" 的新子文件夹，然后进入该文件夹。然后，创建一个名为"template.json" 的新文件。文件夹结构应如下所示：

```
working
└──templates
    └──extensions
        └──.template.config
                template.json
```

使用你喜爱的文本编辑器打开 template.json 并粘贴以下 JSON 代码，然后保存。

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Code" ],
  "identity": "ExampleTemplate.StringExtensions",
  "name": "Example templates: string extensions",
  "shortName": "stringext",
  "tags": {
    "language": "C#",
    "type": "item"
  }
}
```

此配置文件包含模板的所有设置。可以看到基本设置，例如 `name` 和 `shortName`，除此之外，还有一个设置为 `item` 的 `tags/type` 值。这会将你的模板归类为项模板。你创建的模板类型不存在限制。`item` 和 `project` 值是 .NET 建议使用的通用名称，便于用户轻松筛选正在搜索的模板类型。

`classifications` 项表示你在运行 `dotnet new` 并获取模板列表时看到的"标记" 列。用户还可以根据分类标记进行搜索。不要将 *.json 文件中的 `tags` 属性与 `classifications` 标记列表混淆。它们虽然具有类似的名称，但截然不同。template.json 文件的完整架构位于 JSON 架构存储。有关 template.json 文件的详细信息，请参阅 dotnet 创建模板 wiki。

现在你已有一个有效的 .template.config/template.json 文件，可以安装模板了。在终端中，导航到 extensions 文

件夹，并运行以下命令以安装位于当前文件夹的模板：

- **在 Windows 上**：`dotnet new --install .\`
- **在 Linux 或 macOS 上**：`dotnet new --install ./`

此命令输出安装的模板列表，其中应包括你的模板。

```
The following template packages will be installed:
   <root path>\working\templates\extensions

Success: <root path>\working\templates\extensions installed the following templates:
Templates                                       Short Name               Language          Tags
----------------------------------------------  -------------------      ------------      ---------------
-------
Example templates: string extensions            stringext                [C#]              Common/Code
```

测试项模板

现在你已安装了项模板，可对其进行测试。导航到 test/ 文件夹，使用 `dotnet new console` 创建新的控制台应用程序。这将生成一个可以使用 `dotnet run` 命令轻松测试的工作项目。

```
dotnet new console
```

将获得类似于下面的输出。

```
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on C:\test\test.csproj...
  Restore completed in 54.82 ms for C:\test\test.csproj.

Restore succeeded.
```

运行该项目。

```
dotnet run
```

将获得以下输出。

```
Hello World!
```

接下来，运行 `dotnet new stringext` 以从模板生成 CommonExtensions.cs 。

```
dotnet new stringext
```

将获得以下输出。

```
The template "Example templates: string extensions" was created successfully.
```

更改 Program.cs 中的代码以使用模板提供的扩展方法反转 `"Hello World"` 字符串。

```
Console.WriteLine("Hello World!".Reverse());
```

再次运行程序，将看到结果已反转。

```
dotnet run
```

将获得以下输出。

```
!dlroW olleH
```

祝贺你！ 你已使用 .NET 创建并部署了项模板。为准备学习本系列教程的下一部分，必须卸载已创建的模板。确保同时删除 test 文件夹中的所有文件。这将回到干净状态，为本教程的下一个主要部分做好准备。

卸载模板

在终端中，导航到 extensions 文件夹，并运行以下命令以卸载位于当前文件夹的模板：

- **在 Windows 上**：`dotnet new --uninstall .\`
- **在 Linux 或 macOS 上**：`dotnet new --uninstall ./`

此命令输出已卸载的模板列表，其中应包括你的模板。

```
Success: <root path>\working\templates\extensions was uninstalled.
```

随时可以使用 `dotnet new --uninstall` 查看已安装的模板包列表，包括每个模板包的卸载命令。

后续步骤

在本教程中，你创建了一个项模板。若要了解如何创建项目模板，请继续学习本系列教程。

创建项目模板

教程：创建项目模板

2021/11/16 •

使用 .NET，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第二部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 创建项目模板的资源
- 创建模板配置文件夹和文件
- 从文件路径安装模板
- 测试项模板
- 卸载项模板

先决条件

- 完成本系列教程的第 1 部分。
- 打开终端并导航到 working\templates 文件夹。

创建项目模板

项目模板生成可立即运行的项目，使用户可以轻松地使用一组有效的代码。.NET 包含一些项目模板，例如控制台应用程序或类库。在本例中，你将创建一个启用 C# 10.0 并生成 `async main` 入口点的新控制台项目。

在终端中，导航到 working\templates 文件夹，并创建一个名为“consoleasync”的新子文件夹。进入子文件夹，并运行 `dotnet new console` 以生成标准控制台应用程序。将编辑此模板生成的文件以创建新模板。

```
working
└──templates
    └──consoleasync
            consoleasync.csproj
            Program.cs
```

修改 Program.cs

打开 program.cs 文件。控制台项目不使用异步入口点，我们来添加它。将代码更改为以下内容并保存文件。

```
await Console.Out.WriteAsync("Hello World with C# 10.0!");
```

修改 consoleasync.csproj

将项目使用的 C# 语言版本更新到 10.0 版。编辑 consoleasync.csproj 文件并将 `<LangVersion>` 设置添加到 `<PropertyGroup>` 节点。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>

    <LangVersion>10.0</LangVersion>

  </PropertyGroup>

</Project>
```

生成项目

在完成项目模板之前，应对其进行测试，确保它能够正确编译和运行。

在终端中，运行以下命令。

```
dotnet run
```

将获得以下输出。

```
Hello World with C# 10.0!
```

可以使用 `dotnet run` 删除已创建的 obj 和 bin 文件夹。删除这些文件可确保你的模板仅包含与模板相关的文件，而不包含生成操作产生的任何文件。

现在你已经创建了模板的内容，需要在模板的根文件夹中创建模板配置。

创建模板配置

模板在 .NET 中通过模板根目录中的特殊文件夹和配置文件进行识别。在本教程中，你的模板文件夹位于 working\templates\consoleasync 。

创建模板时，除特殊配置文件夹外，模板文件夹中的所有文件和文件夹都作为模板的一部分包含在内。此配置文件夹名为“.template.config” 。

首先，创建一个名为“.template.config” 的新子文件夹，然后进入该文件夹。然后，创建一个名为“template.json” 的新文件。文件夹结构应如下所示。

```
working
└───templates
    └───consoleasync
        └───.template.config
                template.json
```

使用你喜爱的文本编辑器打开 template.json 并粘贴以下 json 代码，然后保存。

```
{
  "$schema": "http://json.schemastore.org/template",
  "author": "Me",
  "classifications": [ "Common", "Console", "C#9" ],
  "identity": "ExampleTemplate.AsyncProject",
  "name": "Example templates: async project",
  "shortName": "consoleasync",
  "tags": {
    "language": "C#",
    "type": "project"
  }
}
```

此配置文件包含模板的所有设置。可以看到基本设置，例如 `name` 和 `shortName`，除此之外，还有一个设置为 `project` 的 `tags/type` 值。这会将模板指定为项目模板。你创建的模板类型不存在限制。`item` 和 `project` 值是 .NET 建议使用的通用名称，便于用户轻松筛选正在搜索的模板类型。

`classifications` 项表示你在运行 `dotnet new` 并获取模板列表时看到的"标记" 列。用户还可以根据分类标记进行搜索。不要将 json 文件中的 `tags` 属性与 `classifications` 标记列表混淆。它们虽然具有类似的名称，但截然不同。template.json 文件的完整架构位于 JSON 架构存储。有关 template.json 文件的详细信息，请参阅 dotnet 创建模板 wiki。

现在你已有一个有效的 .template.config/template.json 文件，可以安装模板了。在安装模板之前，请务必删除无需在模板中包含的任何额外文件夹和文件，例如 bin 或 obj 文件夹。在终端中，导航到 consoleasync 文件夹，并运行 `dotnet new --install .\` 以安装位于当前文件夹的模板。如果使用的是 Linux 或 macOS 操作系统，请使用正斜杠：`dotnet new --install ./`。

```
dotnet new --install .\
```

此命令输出安装的模板列表，其中应包括你的模板。

```
The following template packages will be installed:
   <root path>\working\templates\consoleasync

Success: <root path>\working\templates\consoleasync installed the following templates:
Templates                                       Short Name               Language          Tags
----------------------------------------------  -------------------      ------------      ---------------
-------
Example templates: async project                consoleasync             [C#]
Common/Console/C#9
```

测试项目模板

现在你已安装了项目模板，可对其进行测试。

1. 导航到 test 文件夹
2. 使用以下命令创建一个新的控制台应用程序，该命令生成可使用 `dotnet run` 命令轻松测试的工作项目。

   ```
   dotnet new consoleasync
   ```

   将获得以下输出。

   ```
   The template "Example templates: async project" was created successfully.
   ```

3. 请使用以下命令运行项目。

```
dotnet run
```

将获得以下输出。

```
Hello World with C# 10.0!
```

祝贺你！ 你已使用 .NET 创建并部署了项目模板。为准备学习本系列教程的下一部分，必须卸载已创建的模板。确保同时删除 test 文件夹中的所有文件。这将回到干净状态，为本教程的下一个主要部分做好准备。

卸载模板

在终端中，导航到 consoleasync 文件夹，并运行以下命令以卸载位于当前文件夹的模板：

- **在 Windows 上**：`dotnet new --uninstall .\`
- **在 Linux 或 macOS 上**：`dotnet new --uninstall ./`

此命令输出已卸载的模板列表，其中应包括你的模板。

```
Success: <root path>\working\templates\consoleasync was uninstalled.
```

随时可以使用 `dotnet new --uninstall` 查看已安装的模板包列表，包括每个模板包的卸载命令。

后续步骤

在本教程中，你创建了一个项目模板。若要了解如何将项模板和项目模板打包为易于使用的文件，请继续学习本教程系列。

创建模板包

教程：创建模板包

2021/11/16 •

使用 .NET，可以创建和部署可生成项目、文件甚至资源的模板。本教程是系列教程的第三部分，介绍如何创建、安装和卸载用于 `dotnet new` 命令的模板。

在本系列的这一部分中，你将了解如何：

- 创建一个 *.csproj 项目以生成模板包
- 配置项目文件以进行打包
- 从 NuGet 包文件安装模板包
- 按包 ID 卸载模板包

先决条件

- 完成本系列教程的第 1 部分和第 2 部分。

  本教程使用本教程前两部分中创建的两个模板。只要将不同的模板作为文件夹复制到 working\templates\ 文件夹中，就可以使用该模板。

- 打开终端并导航到 working\ 文件夹。

创建模板包项目

模板包是打包到 NuGet 包中的一个或多个模板。安装或卸载模板包时，将分别添加或删除包中包含的所有模板。本系列教程的前几部分仅适用于各自的模板。若要共享非打包模板，必须复制模板文件夹并通过该文件夹进行安装。由于模板包中可以包含多个模板，并且是单个文件，因此共享更容易。

模板包由 NuGet 包 (.nupkg) 文件表示。并且，与任何 NuGet 包一样，可以将模板包上传到 NuGet 源。`dotnet new --install` 命令支持从 NuGet 包源安装模板包。此外，可以直接从 .nupkg 文件安装模板包。

通常情况下，使用 C# 项目文件来编译代码并生成二进制文件。但是，该项目也可用于生成模板包。通过更改 .csproj 的设置，可以阻止它编译任何代码，而是将模板的所有资产都包含在内作为资源。生成此项目后，它会生成模板包 NuGet 包。

将要创建的包将包含先前创建的项模板和包模板。由于我们将两个模板分组到 working\templates\ 文件夹中，因此可以使用 .csproj 文件的 working 文件夹。

在终端中，导航到 working 文件夹。创建一个新项目，将名称设置为 `templatepack`，并将输出文件夹设置为当前文件夹。

```
dotnet new console -n templatepack -o .
```

`-n` 参数将 .csproj 文件名设置为 templatepack.csproj 。`-o` 参数将在当前目录中创建文件。应看到类似于以下输出的结果。

```
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on .\templatepack.csproj...
  Restore completed in 52.38 ms for C:\working\templatepack.csproj.

Restore succeeded.
```

新项目模板生成 Program.cs 文件。模板不使用此文件，你可以安全删除。

接下来，在你喜爱的编辑器中打开 templatepack.csproj 文件，并将内容替换为以下 XML：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <PackageType>Template</PackageType>
    <PackageVersion>1.0</PackageVersion>
    <PackageId>AdatumCorporation.Utility.Templates</PackageId>
    <Title>AdatumCorporation Templates</Title>
    <Authors>Me</Authors>
    <Description>Templates to use when creating an application for Adatum Corporation.</Description>
    <PackageTags>dotnet-new;templates;contoso</PackageTags>

    <TargetFramework>netstandard2.0</TargetFramework>

    <IncludeContentInPack>true</IncludeContentInPack>
    <IncludeBuildOutput>false</IncludeBuildOutput>
    <ContentTargetFolders>content</ContentTargetFolders>
    <NoWarn>$(NoWarn);NU5128</NoWarn>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="templates\**\*" Exclude="templates\**\bin\**;templates\**\obj\**" />
    <Compile Remove="**\*" />
  </ItemGroup>

</Project>
```

上面的 XML 中的 `<PropertyGroup>` 设置分为三组。第一组处理 NuGet 包所需的属性。三个 `<Package*>` 设置与 NuGet 包属性相关，用于在 NuGet 源上标识你的包。具体来说，`<PackageId>` 值用于通过单个名称而不是目录路径卸载模板包。它还可用于从 NuGet 源安装模板包。其余的设置，如 `<Title>` 和 `<PackageTags>`，它们与 NuGet 源上显示的元数据相关。有关 NuGet 设置的详细信息，请参阅 NuGet 和 MSBuild 属性。

> **NOTE**
> 若要确保模板包显示在 `dotnet new --search` 结果中，请将 `<PackageType>` 设置为 `Template`。

必须设置 `<TargetFramework>` 设置，以便在运行 pack 命令编译和打包项目时 MSBuild 正常运行。

接下来的三项设置与正确配置项目有关，以便在创建 NuGet 包时将模板包含在该包内的相应文件夹中。

最后一项设置可用于取消不适用于模板包项目的警告消息。

`<ItemGroup>` 包含两个设置。首先，`<Content>` 设置的内容包含 templates 文件夹中的所有内容。它还设置为排除任何 bin 文件夹或 obj 文件夹，以防止包含任何已编译的代码(如果已测试和编译模板)。其次，`<Compile>` 设置将所有代码文件排除在编译范围之外，无论它们位于何处都是如此。此设置可阻止用于创建模板包的项目尝试编译 templates 文件夹层次结构中的代码。

生成和安装

保存项目文件。在生成模板包之前，请验证文件夹结构是否正确。要打包的任何模板都应放置在自己的文件夹中的 templates 文件夹中。文件夹结构应如下所示：

```
working
│   templatepack.csproj
└───templates
    ├───extensions
    │   └───.template.config
    │           template.json
    └───consoleasync
        └───.template.config
                template.json
```

templates 文件夹中有两个文件夹：extensions 和 consoleasync 。

在终端中的 working 文件夹中，运行 `dotnet pack` 命令。此命令会生成项目，并在 working\bin\Debug 文件夹中创建一个 NuGet 包，如以下输出所示：

```
C:\working> dotnet pack

Microsoft (R) Build Engine version 16.8.0+126527ff1 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Restore completed in 123.86 ms for C:\working\templatepack.csproj.

  templatepack -> C:\working\bin\Debug\netstandard2.0\templatepack.dll
  Successfully created package 'C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg'.
```

接下来，使用 `dotnet new --install PATH_TO_NUPKG_FILE` 命令安装模板包。

```
C:\working> dotnet new -i C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg
The following template packages will be installed:
   C:\working\bin\Debug\AdatumCorporation.Utility.Templates.1.0.0.nupkg

Success: AdatumCorporation.Utility.Templates::1.0.0 installed the following templates:
Templates                                       Short Name               Language          Tags
----------------------------------------------      -------------------      ------------      ---------------
-------
Example templates: string extensions            stringext                [C#]              Common/Code
Example templates: async project                consoleasync             [C#]
Common/Console/C#9
```

如果将 NuGet 包上传到 NuGet 源，可以使用 `dotnet new --install PACKAGEID` 命令，其中，`PACKAGEID` 与 .csproj 文件中的 `<PackageId>` 设置相同。此包 ID 与 NuGet 包标识符相同。

卸载模板包

无论如何安装模板包，即无论是直接使用 .nupkg 文件还是通过 NuGet 源安装，删除模板包的操作都是一样的。使用要卸载的模板的 `<PackageId>`。可以通过运行 `dotnet new --uninstall` 命令获取已安装的模板列表。

```
C:\working> dotnet new --uninstall

Template Instantiation Commands for .NET CLI

Currently installed items:
  Microsoft.DotNet.Common.ProjectTemplates.2.2
    Details:
      NuGetPackageId: Microsoft.DotNet.Common.ProjectTemplates.2.2
      Version: 1.0.2-beta4
      Author: Microsoft
    Templates:
      Class library (classlib) C#
      Class library (classlib) F#
      Class library (classlib) VB
      Console Application (console) C#
      Console Application (console) F#
      Console Application (console) VB
    Uninstall Command:
      dotnet new --uninstall  Microsoft.DotNet.Common.ProjectTemplates.2.2

... cut to save space ...

  AdatumCorporation.Utility.Templates
    Details:
      NuGetPackageId: AdatumCorporation.Utility.Templates
      Version: 1.0.0
      Author: Me
    Templates:
      Example templates: async project (consoleasync) C#
      Example templates: string extensions (stringext) C#
    Uninstall Command:
      dotnet new --uninstall AdatumCorporation.Utility.Templates
```

运行 `dotnet new --uninstall AdatumCorporation.Utility.Templates` 以卸载模板包。该命令将输出有关卸载了哪些模板包的信息。

祝贺！ 你已完成模板包的安装和卸载操作。

后续步骤

若要了解有关模板的详细信息(你已经了解了大部分相关信息)，请参阅为 dotnet new 自定义模板一文。

- dotnet/templating GitHub 存储库 Wiki
- dotnet/dotnet-template-samples GitHub 存储库
- JSON 架构存储中的 template.json 架构

.NET 5+ 中已过时的功能

2021/11/16 •

从 .NET 5 开始，一些新标记为已过时的 API 使用 ObsoleteAttribute 上的两个新属性。

- ObsoleteAttribute.DiagnosticId 属性指示编译器使用自定义诊断 ID 产生生成警告。通过自定义 ID 可专门、单独地取消过时警告。对于 .NET 5+ 过时，自定义诊断 ID 的格式为 `SYSLIB0XXX`。
- ObsoleteAttribute.UrlFormat 属性指示编译器包含一个 URL 链接，可了解有关过时的详细信息。

如果由于使用过时的 API 而遇到生成警告或错误，请遵循参考部分中列出的诊断 ID 所提供的特定指导。不能使用过时类型或成员的标准诊断 ID (CS0618) 取消有关这些过时类型或成员的警告或错误；请改用自定义 `SYSLIB0XXX` 诊断 ID 值。有关详细信息，请参阅取消警告。

参考

下表提供了 .NET 5+ 中 `SYSLIB0XXX` 过时的索引。

ID		
SYSLIB0001	警告	UTF-7 编码不安全，因此不应使用。请考虑改用 UTF-8。
SYSLIB0002	错误	PrincipalPermissionAttribute 不受运行时支持，不得使用。
SYSLIB0003	警告	运行时不支持或不接受代码访问安全性 (CAS)。
SYSLIB0004	警告	不支持受约束的执行区域 (CER) 功能。
SYSLIB0005	警告	不支持全局程序集缓存 (GAC)。
SYSLIB0006	警告	Thread.Abort() 不受支持并会引发 PlatformNotSupportedException。
SYSLIB0007	警告	不支持此加密算法的默认实现。
SYSLIB0008	警告	CreatePdbGenerator() API 不受支持并会引发 PlatformNotSupportedException。
SYSLIB0009	警告	AuthenticationManager.Authenticate 和 AuthenticationManager.PreAuthenticate 方法都不受支持并会引发 PlatformNotSupportedException。
SYSLIB0010	警告	某些远程处理 API 不受支持并会引发 PlatformNotSupportedException。

ID	级别	说明
SYSLIB0011	警告	BinaryFormatter 序列化已过时，不应使用。
SYSLIB0012	警告	包含 Assembly.CodeBase 和 Assembly.EscapedCodeBase 只是为了实现 .NET Framework 兼容性。请改用 Assembly.Location。
SYSLIB0013	警告	在某些情况下，Uri.EscapeUriString(String) 可能会导致 Uri 字符串损坏。请考虑改用 Uri.EscapeDataString(String) 查询字符串组件。
SYSLIB0014	错误	WebRequest、HttpWebRequest、ServicePoint 与 WebClient 已过时。请改用 HttpClient。
SYSLIB0015	警告	DisablePrivateReflectionAttribute 在 .NET 6+ 中不起作用。
SYSLIB0016	警告	为改善性能并减少分配，使用接受参数的 Graphics.GetContextInfo 重载。
SYSLIB0017	警告	强名称签名不受支持并引发 PlatformNotSupportedException。
SYSLIB0018	警告	仅反射加载不受支持并引发 PlatformNotSupportedException。
SYSLIB0019	警告	不再支持 System.Runtime.InteropServices.RuntimeEnvironment 成员 SystemConfigurationFile、GetRuntimeInterfaceAsIntPtr(Guid, Guid) 与 GetRuntimeInterfaceAsObject(Guid, Guid) 并引发 PlatformNotSupportedException。
SYSLIB0020	警告	JsonSerializerOptions.IgnoreNullValues 已过时。若要在序列化时忽略 null 值，请将 DefaultIgnoreCondition 设为 JsonIgnoreCondition.WhenWritingNull。
SYSLIB0021	警告	派生的加密类型已过时。请改为对基类型使用 `Create` 方法。
SYSLIB0022	警告	Rijndael 和 RijndaelManaged 类型已过时。请改用 Aes。
SYSLIB0023	警告	RNGCryptoServiceProvider 已过时。若要生成随机数，请改为使用 RandomNumberGenerator 静态方法之一。

ID		
SYSLIB0024	警告	不支持创建和卸载 AppDomains，并引发异常。
SYSLIB0025	警告	SuppressIldasmAttribute 在 .NET 6+ 中不起作用。
SYSLIB0026	警告	X509Certificate 和 X509Certificate2 是不可变的。使用适当的构造函数创建新证书。
SYSLIB0027	警告	PublicKey.Key 已过时。使用适当的方法获取公钥，如 GetRSAPublicKey()。
SYSLIB0028	警告	X509Certificate2.PrivateKey 已过时。使用适当的方法获取私钥（例如 RSACertificateExtensions.GetRSAPrivateKey(X509Certificate2)），或使用 X509Certificate2.CopyWithPrivateKey(ECDiffieHellman) 方法创建一个具有私钥的新实例。
SYSLIB0029	警告	`ProduceLegacyHmacValues` 已过时。不再支持生成旧的 HMAC 值。
SYSLIB0030	警告	`HMACSHA1` 始终使用平台提供的算法实现。使用不带 `useManagedSha1` 参数的构造函数。
SYSLIB0031	警告	CryptoConfig.EncodeOID(String) 已过时。使用 System.Formats.Asn1 中提供的 ASN.1 功能。
SYSLIB0032	警告	不支持从进程状态已损坏异常中恢复；将忽略 HandleProcessCorruptedStateExceptionsAttribute。
SYSLIB0033	警告	Rfc2898DeriveBytes.CryptDeriveKey(String, String, Int32, Byte[]) 已过时，不受支持。请改用 PasswordDeriveBytes.CryptDeriveKey(String, String, Int32, Byte[])。
SYSLIB0034	警告	CmsSigner(CspParameters) 已过时。请改用替代的构造函数。
SYSLIB0035	警告	SignerInfo.ComputeCounterSignature() 已过时。请改用接受 CmsSigner 的重载。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0001：UTF-7 编码不安全

2021/11/16 •

UTF-7 编码在应用程序中不再广泛使用，并且许多规范现在在交换中禁止其使用。它偶尔还会在不期望遇到 UTF-7 编码数据的应用程序中用作攻击途径。Microsoft 警告不要使用 System.Text.UTF7Encoding，因为它不提供错误检测。

因此从 .NET 5 开始，以下 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0001`。

- Encoding.UTF7 属性
- UTF7Encoding 构造函数

工作区

- 如果在你自己的协议或文件格式中使用的是 Encoding.UTF7 或 UTF7Encoding：

  切换到使用 Encoding.UTF8 或 UTF8Encoding。UTF-8 是一种行业标准，并且受到语言、操作系统和运行时的广泛支持。使用 UTF-8 简化了代码的将来维护，并使其与生态系统的其余部分更具互操作性。

- 如果要将 Encoding 实例与 Encoding.UTF7 进行比较：

  可转为考虑针对众所周知的 UTF-7 代码页（即 `65000`）执行检查。通过与代码页进行比较，可以避免出现警告，还可以处理一些边缘情况，例如，如果有人调用 `new UTF7Encoding()` 或子类化类型。

```
void DoSomething(Encoding enc)
{
    // Don't perform the check this way.
    // It produces a warning and misses some edge cases.
    if (enc == Encoding.UTF7)
    {
        // Encoding is UTF-7.
    }

    // Instead, perform the check this way.
    if (enc != null && enc.CodePage == 65000)
    {
        // Encoding is UTF-7.
    }
}
```

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- UTF-7 代码路径已过时

SYSLIB0002：PrincipalPermissionAttribute 已过时

2021/11/16 •

从 .NET 5 开始，PrincipalPermissionAttribute 构造函数已过时并生成编译时错误 `SYSLIB0002`。不能实例化此属性或将其应用于方法。

与其他过时警告不同，无法禁止显示此错误。

工作区

- 如果要将属性应用于 ASP.NET MVC 操作方法，请执行以下操作：

  考虑使用 ASP.NET 的内置授权基础结构。以下代码演示如何使用 AuthorizeAttribute 属性来为控制器添加批注。ASP.NET 运行时将在执行操作之前向用户授权。

  ```
  using Microsoft.AspNetCore.Authorization;

  namespace MySampleApp
  {
      [Authorize(Roles = "Administrator")]
      public class AdministrationController : Controller
      {
          public ActionResult MyAction()
          {
              // This code won't run unless the current user
              // is in the 'Administrator' role.
          }
      }
  }
  ```

  有关详细信息，请参阅 ASP.NET Core 中基于角色的授权和 ASP.NET Core 中的授权简介。

- 如果要将属性应用到 Web 应用上下文之外的库代码，请执行以下操作：

  通过调用 IPrincipal.IsInRole(String) 方法，在方法开始时手动执行检查。

  ```
  using System.Threading;

  void DoSomething()
  {
      if (Thread.CurrentPrincipal == null
          || !Thread.CurrentPrincipal.IsInRole("Administrators"))
      {
          throw new Exception("User is anonymous or isn't an admin.");
      }

      // Code that should run only when user is an administrator.
  }
  ```

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

PrincipalPermissionAttribute 已过时，报告为错误

SYSLIB0003：不支持代码访问安全性

2021/11/16 •

代码访问安全性 (CAS) 是一项不受支持的传统技术。用于启用 CAS 的基础结构（仅存在于 .NET Framework 2.x - 4.x）已弃用，并且不接受服务或安全修补。

因此，从 .NET 5 开始，.NET 中大多数与代码访问安全性 (CAS) 相关的类型均已过时。这包含 CAS 属性（如 SecurityPermissionAttribute）、CAS 权限对象（如 SocketPermission）、EvidenceBase 派生类型和其他支持 API。使用这些 API 会在编译时生成警告 `SYSLIB0003`。

已过时 CAS API 的完整列表如下所示：

- System.AppDomain.ExecuteAssembly(String, String[], Byte[], AssemblyHashAlgorithm)
- System.AppDomain.PermissionSet
- System.Configuration.ConfigurationPermission
- System.Configuration.ConfigurationPermissionAttribute
- System.Data.Common.DBDataPermission
- System.Data.Common.DBDataPermissionAttribute
- System.Data.Odbc.OdbcPermission
- System.Data.Odbc.OdbcPermissionAttribute
- System.Data.OleDb.OleDbPermission
- System.Data.OleDb.OleDbPermissionAttribute
- System.Data.OracleClient.OraclePermission
- System.Data.OracleClient.OraclePermissionAttribute
- System.Data.SqlClient.SqlClientPermission
- System.Data.SqlClient.SqlClientPermissionAttribute
- System.Diagnostics.EventLogPermission
- System.Diagnostics.EventLogPermissionAttribute
- System.Diagnostics.PerformanceCounterPermission
- System.Diagnostics.PerformanceCounterPermissionAttribute
- System.DirectoryServices.DirectoryServicesPermission
- System.DirectoryServices.DirectoryServicesPermissionAttribute
- System.Drawing.Printing.PrintingPermission
- System.Drawing.Printing.PrintingPermissionAttribute
- System.Net.DnsPermission
- System.Net.DnsPermissionAttribute
- System.Net.Mail.SmtpPermission
- System.Net.Mail.SmtpPermissionAttribute
- System.Net.NetworkInformation.NetworkInformationPermission
- System.Net.NetworkInformation.NetworkInformationPermissionAttribute
- System.Net.PeerToPeer.Collaboration.PeerCollaborationPermission
- System.Net.PeerToPeer.Collaboration.PeerCollaborationPermissionAttribute
- System.Net.PeerToPeer.PnrpPermission
- System.Net.PeerToPeer.PnrpPermissionAttribute
- System.Net.SocketPermission
- System.Net.SocketPermissionAttribute

- System.Net.WebPermission
- System.Net.WebPermissionAttribute
- System.Runtime.InteropServices.AllowReversePInvokeCallsAttribute
- System.Security.CodeAccessPermission
- System.Security.HostProtectionException
- System.Security.IPermission
- System.Security.IStackWalk
- System.Security.NamedPermissionSet
- System.Security.PermissionSet
- System.Security.Permissions.CodeAccessSecurityAttribute
- System.Security.Permissions.DataProtectionPermission
- System.Security.Permissions.DataProtectionPermissionAttribute
- System.Security.Permissions.DataProtectionPermissionFlags
- System.Security.Permissions.EnvironmentPermission
- System.Security.Permissions.EnvironmentPermissionAccess
- System.Security.Permissions.EnvironmentPermissionAttribute
- System.Security.Permissions.FileDialogPermission
- System.Security.Permissions.FileDialogPermissionAccess
- System.Security.Permissions.FileDialogPermissionAttribute
- System.Security.Permissions.FileIOPermission
- System.Security.Permissions.FileIOPermissionAccess
- System.Security.Permissions.FileIOPermissionAttribute
- System.Security.Permissions.GacIdentityPermission
- System.Security.Permissions.GacIdentityPermissionAttribute
- System.Security.Permissions.HostProtectionAttribute
- System.Security.Permissions.HostProtectionResource
- System.Security.Permissions.IUnrestrictedPermission
- System.Security.Permissions.IsolatedStorageContainment
- System.Security.Permissions.IsolatedStorageFilePermission
- System.Security.Permissions.IsolatedStorageFilePermissionAttribute
- System.Security.Permissions.IsolatedStoragePermission
- System.Security.Permissions.IsolatedStoragePermissionAttribute
- System.Security.Permissions.KeyContainerPermission
- System.Security.Permissions.KeyContainerPermissionAccessEntry
- System.Security.Permissions.KeyContainerPermissionAccessEntryCollection
- System.Security.Permissions.KeyContainerPermissionAccessEntryEnumerator
- System.Security.Permissions.KeyContainerPermissionAttribute
- System.Security.Permissions.KeyContainerPermissionFlags
- System.Security.Permissions.MediaPermission
- System.Security.Permissions.MediaPermissionAttribute
- System.Security.Permissions.MediaPermissionAudio
- System.Security.Permissions.MediaPermissionImage
- System.Security.Permissions.MediaPermissionVideo
- System.Security.Permissions.PermissionSetAttribute
- System.Security.Permissions.PermissionState
- System.Security.Permissions.PrincipalPermission

- System.Security.Permissions.PrincipalPermissionAttribute
- System.Security.Permissions.PublisherIdentityPermission
- System.Security.Permissions.PublisherIdentityPermissionAttribute
- System.Security.Permissions.ReflectionPermission
- System.Security.Permissions.ReflectionPermissionAttribute
- System.Security.Permissions.ReflectionPermissionFlag
- System.Security.Permissions.RegistryPermission
- System.Security.Permissions.RegistryPermissionAccess
- System.Security.Permissions.RegistryPermissionAttribute
- System.Security.Permissions.ResourcePermissionBase
- System.Security.Permissions.ResourcePermissionBaseEntry
- System.Security.Permissions.SecurityAction
- System.Security.Permissions.SecurityAttribute
- System.Security.Permissions.SecurityPermission
- System.Security.Permissions.SecurityPermissionAttribute
- System.Security.Permissions.SecurityPermissionFlag
- System.Security.Permissions.SiteIdentityPermission
- System.Security.Permissions.SiteIdentityPermissionAttribute
- System.Security.Permissions.StorePermission
- System.Security.Permissions.StorePermissionAttribute
- System.Security.Permissions.StorePermissionFlags
- System.Security.Permissions.StrongNameIdentityPermission
- System.Security.Permissions.StrongNameIdentityPermissionAttribute
- System.Security.Permissions.StrongNamePublicKeyBlob
- System.Security.Permissions.TypeDescriptorPermission
- System.Security.Permissions.TypeDescriptorPermissionAttribute
- System.Security.Permissions.TypeDescriptorPermissionFlags
- System.Security.Permissions.UIPermission
- System.Security.Permissions.UIPermissionAttribute
- System.Security.Permissions.UIPermissionClipboard
- System.Security.Permissions.UIPermissionWindow
- System.Security.Permissions.UrlIdentityPermission
- System.Security.Permissions.UrlIdentityPermissionAttribute
- System.Security.Permissions.WebBrowserPermission
- System.Security.Permissions.WebBrowserPermissionAttribute
- System.Security.Permissions.WebBrowserPermissionLevel
- System.Security.Permissions.ZoneIdentityPermission
- System.Security.Permissions.ZoneIdentityPermissionAttribute
- System.Security.Policy.ApplicationTrust.ApplicationTrust(PermissionSet, IEnumerable<StrongName>)
- System.Security.Policy.ApplicationTrust.FullTrustAssemblies
- System.Security.Policy.FileCodeGroup
- System.Security.Policy.GacInstalled
- System.Security.Policy.IIdentityPermissionFactory
- System.Security.Policy.PolicyLevel.AddNamedPermissionSet(NamedPermissionSet)
- System.Security.Policy.PolicyLevel.ChangeNamedPermissionSet(String, PermissionSet)
- System.Security.Policy.PolicyLevel.GetNamedPermissionSet(String)

- System.Security.Policy.PolicyLevel.RemoveNamedPermissionSet
- System.Security.Policy.PolicyStatement.PermissionSet
- System.Security.Policy.PolicyStatement.PolicyStatement
- System.Security.Policy.Publisher
- System.Security.Policy.Site
- System.Security.Policy.StrongName
- System.Security.Policy.StrongNameMembershipCondition
- System.Security.Policy.Url
- System.Security.Policy.Zone
- System.Security.SecurityContext
- System.Security.SecurityManager
- System.ServiceProcess.ServiceControllerPermission
- System.ServiceProcess.ServiceControllerPermissionAttribute
- System.Threading.Thread.GetCompressedStack()
- System.Threading.Thread.SetCompressedStack(CompressedStack)
- System.Transactions.DistributedTransactionPermission
- System.Transactions.DistributedTransactionPermissionAttribute
- System.Web.AspNetHostingPermission
- System.Web.AspNetHostingPermissionAttribute
- System.Xaml.Permissions.XamlLoadPermission

工作区

- 如果要断言任何安全权限，请删除断言该权限的属性或调用。

```
// REMOVE the attribute below.
[SecurityPermission(SecurityAction.Assert, ControlThread = true)]
public void DoSomething()
{
}
public void DoAssert()
{
    // REMOVE the line below.
    new SecurityPermission(SecurityPermissionFlag.ControlThread).Assert();
}
```

- 如果要拒绝或限制（通过 `PermitOnly`）任何权限，请与安全顾问联系。由于 .NET 5 及更高版本的运行时不支持 CAS 属性，因此如果应用程序错误地依赖于 CAS 基础结构来限制对这些方法的访问，则它可能存在安全漏洞。

```
// REVIEW the attribute below; could indicate security vulnerability.
[SecurityPermission(SecurityAction.Deny, ControlThread = true)]
public void DoSomething()
{
}
public void DoPermitOnly()
{
    // REVIEW the line below; could indicate security vulnerability.
    new SecurityPermission(SecurityPermissionFlag.ControlThread).PermitOnly();
}
```

- 如果要求任何权限（除 PrincipalPermission 外），请删除该请求。所有请求都将在运行时成功。

```
// REMOVE the attribute below; it will always succeed.
[SecurityPermission(SecurityAction.Demand, ControlThread = true)]
public void DoSomething()
{
}
public void DoDemand()
{
    // REMOVE the line below; it will always succeed.
    new SecurityPermission(SecurityPermissionFlag.ControlThread).Demand();
}
```

- 如果要求 PrincipalPermission，请参阅 SYSLIB0002：PrincipalPermissionAttribute 已过时指南。本指南适用于 PrincipalPermission 和 PrincipalPermissionAttribute。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- SYSLIB0002：PrincipalPermissionAttribute 已过时

SYSLIB0004：不支持受约束的执行区域 (CER) 功能

2021/11/16 •

受约束的执行区域 (CER) 功能仅在 .NET Framework 中受支持。因此从 .NET 5 开始，与 CER 相关的各种 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0004`。

以下与 CER 相关的 API 已过时：

- RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup(RuntimeHelpers+TryCode, RuntimeHelpers+CleanupCode, Object)
- RuntimeHelpers.PrepareConstrainedRegions()
- RuntimeHelpers.PrepareConstrainedRegionsNoOP()
- RuntimeHelpers.PrepareContractedDelegate(Delegate)
- RuntimeHelpers.ProbeForSufficientStack()
- System.Runtime.ConstrainedExecution.Cer
- System.Runtime.ConstrainedExecution.Consistency
- System.Runtime.ConstrainedExecution.PrePrepareMethodAttribute
- System.Runtime.ConstrainedExecution.ReliabilityContractAttribute

工作区

- 如果已将 CER 属性应用于方法，请删除该属性。这些属性在 .NET 5 及更高版本中无效。

```
// REMOVE the attribute below.
[ReliabilityContract(Consistency.WillNotCorruptState, Cer.Success)]
public void DoSomething()
{
}

// REMOVE the attribute below.
[PrePrepareMethod]
public void DoSomething()
{
}
```

- 如果调用 `RuntimeHelpers.ProbeForSufficientStack` 或 `RuntimeHelpers.PrepareContractedDelegate`，请删除该调用。这些调用在 .NET 5 及更高版本中无效。

```
public void DoSomething()
{
    // REMOVE the call below.
    RuntimeHelpers.ProbeForSufficientStack();

    // (Remainder of your method logic here.)
}
```

- 如果要调用 `RuntimeHelpers.PrepareConstrainedRegions`，请删除该调用。该调用在 .NET 5 及更高版本中无效。

```
public void DoSomething_Old()
{
    // REMOVE the call below.
    RuntimeHelpers.PrepareConstrainedRegions();
    try
    {
        // try code
    }
    finally
    {
        // cleanup code
    }
}

public void DoSomething_Corrected()
{
    // There is no call to PrepareConstrainedRegions. It's a normal try / finally block.

    try
    {
        // try code
    }
    finally
    {
        // cleanup code
    }
}
```

- 如果调用 `RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup`，请将调用替换为标准 try / catch / finall 块。

```
// The sample below produces warning SYSLIB0004.
public void DoSomething_Old()
{
    RuntimeHelpers.ExecuteCodeWithGuaranteedCleanup(MyTryCode, MyCleanupCode, null);
}
public void MyTryCode(object state) { /* try code */ }
public void MyCleanupCode(object state, bool exceptionThrown) { /* cleanup code */ }

// The corrected sample below does not produce warning SYSLIB0004.
public void DoSomething_Corrected()
{
    try
    {
        // try code
    }
    catch (Exception ex)
    {
        // exception handling code
    }
    finally
    {
        // cleanup code
    }
}
```

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- 受约束的执行区域

SYSLIB0005：不支持全局程序集缓存 (GAC)

2021/11/16 •

.NET Core 和 .NET 5 及更高版本消除了 .NET Framework 中存在的全局程序集缓存 (GAC) 这一概念。为帮助开发人员摒弃这些 API，从 .NET 5 开始，一些 GAC 相关的 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0005`。

以下与 GAC 相关的 API 标记为已过时：

- Assembly.GlobalAssemblyCache

  库和应用不应使用 GlobalAssemblyCache API 来确定运行时行为，因为它在 .NET Core 和 .NET 5+ 中始终返回 `false`。

解决方法

如果你的应用程序查询 GlobalAssemblyCache 属性，请考虑删除该调用。如果在运行时使用 GlobalAssemblyCache 值在“GAC 中的程序集”流与“不在 GAC 中的程序集”流之间进行选择，请重新考虑流对于 .NET 5+ 应用程序是否仍然有意义。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

请参阅

- 全局程序集缓存

SYSLIB0006：不支持 Thread.Abort

2021/11/16 •

从 .NET 5 开始，以下 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0006`，并在运行时生成 PlatformNotSupportedException。

- Thread.Abort()
- Thread.Abort(Object)

解决方法

使用 CancellationToken 中止对工作单元的处理，而不是调用 Thread.Abort。以下示例说明了 CancellationToken 的用法。

```
void ProcessPendingWorkItemsNew(CancellationToken cancellationToken)
{
    if (QueryIsMoreWorkPending())
    {
        // If the CancellationToken is marked as "needs to cancel",
        // this will throw the appropriate exception.
        cancellationToken.ThrowIfCancellationRequested();

        WorkItem work = DequeueWorkItem();
        ProcessWorkItem(work);
    }
}
```

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- Thread.Abort 已过时
- 托管线程中的取消

SYSLIB0007：不支持加密算法的默认实现

2021/11/16 •

.NET Framework 中的加密配置系统不允许适当的加密灵活性，且不存在于 .NET Core 和 .NET 5+ 中。.NET 的后向兼容性要求也禁止框架更新某些加密 API 以跟上加密技术的发展。因此从 .NET 5 开始，以下 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0007`，并在运行时生成 PlatformNotSupportedException。

- System.Security.Cryptography.AsymmetricAlgorithm.Create()
- System.Security.Cryptography.HashAlgorithm.Create()
- System.Security.Cryptography.HMAC.Create()
- System.Security.Cryptography.KeyedHashAlgorithm.Create()
- System.Security.Cryptography.SymmetricAlgorithm.Create()

解决方法

- 建议采取的操作是用对特定算法（例如 Aes.Create()）的工厂方法的调用替换对现已过时的 API 的调用。这样，便可以完全控制要实例化哪些算法。
- 如果需要保持与使用现已过时的 API 的 .NET Framework 应用生成的现有有效负载的兼容性，请使用下表中建议的替换项。该表提供了从 .NET Framework 默认算法到其 .NET 5+ 等效项的映射。

.NET FRAMEWORK	.NET CORE/.NET 5 IIIIIIIIII	II
AsymmetricAlgorithm.Create()	RSA.Create()	
HashAlgorithm.Create()	SHA1.Create()	SHA-1 算法被认为已无效。如果可能，请考虑使用更强大的算法。请咨询安全顾问以获取进一步的指导。
HMAC.Create()	HMACSHA1()	对于大多数新式应用程序，不建议使用 HMACSHA1 算法。如果可能，请考虑使用更强大的算法。请咨询安全顾问以获取进一步的指导。
KeyedHashAlgorithm.Create()	HMACSHA1()	对于大多数新式应用程序，不建议使用 HMACSHA1 算法。如果可能，请考虑使用更强大的算法。请咨询安全顾问以获取进一步的指导。
SymmetricAlgorithm.Create()	Aes.Create()	

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- 不支持对加密抽象的默认实现进行实例化

SYSLIB0008：不支持 CreatePdbGenerator

2021/11/16 •

从 .NET 5 开始，DebugInfoGenerator.CreatePdbGenerator() API 标记为已过时。使用此 API 将在编译时生成警告 `SYSLIB0008`，并在运行时引发 PlatformNotSupportedException。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0009：AuthenticationManager 身份验证和预身份验证方法不受支持

2021/11/16 •

从 .NET 5 开始，以下 API 标记为已过时。使用这些 API 会在编译时生成警告 `SYSLIB0009`，并在运行时引发 PlatformNotSupportedException。

- AuthenticationManager.Authenticate
- AuthenticationManager.PreAuthenticate

解决方法

实现 IAuthenticationModule，其中具有先前由 AuthenticationManager.Authenticate 调用的方法。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0010：不支持的远程处理 API

2021/11/16 •

.NET 远程处理是一项传统技术，基础结构仅存在于 .NET Framework 中。从 .NET 5 开始，以下与远程处理相关的 API 标记为已过时。在代码中使用这些方法会在编译时生成警告 `SYSLIB0010`，并在运行时引发 PlatformNotSupportedException。

- MarshalByRefObject.GetLifetimeService()
- MarshalByRefObject.InitializeLifetimeService()

解决方法

请考虑使用 WCF 或基于 HTTP 的 REST 服务与其他应用程序的对象或跨计算机进行通信。有关详细信息，请参阅 .NET Framework 技术在 .NET Core 上不可用。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- .NET 远程处理

SYSLIB0011：BinaryFormatter 序列化已过时

2021/11/16 •

由于 BinaryFormatter 存在安全漏洞，从 .NET 5.开始，以下 API 标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0011`。

- System.Exception.SerializeObjectState
- BinaryFormatter.Serialize
- BinaryFormatter.Deserialize
- Formatter.Serialize(Stream, Object)
- Formatter.Deserialize(Stream)
- IFormatter.Serialize(Stream, Object)
- IFormatter.Deserialize(Stream)

解决方法

请考虑使用 JsonSerializer 或 XmlSerializer，而不是 BinaryFormatter。

若要详细了解建议的操作，请参阅修复 BinaryFormatter 过时和禁用错误。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- 修复 BinaryFormatter 过时和禁用错误
- BinaryFormatter 序列化方法已过时，并且已在 ASP.NET 应用中禁用

SYSLIB0012：Assembly.CodeBase 和 Assembly.EscapedCodeBase 已过时

2021/11/16 •

从 .NET 5 开始，以下 API 标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0012`。

- Assembly.CodeBase
- Assembly.EscapedCodeBase

解决方法

请改用 Assembly.Location。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0013：EscapeUriString 已过时

2021/11/16 •

从 .NET 6 开始，Uri.EscapeUriString(String) API 标记为已过时。在代码中使用会在编译时生成警告 `SYSLIB0013`。

在某些情况下，Uri.EscapeUriString(String) 可能会导致 Uri 字符串损坏。

有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/31387。

解决方法

将 Uri.EscapeDataString(String) 用于查询字符串组件。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0014：WebRequest、HttpWebRequest、ServicePoint 与 WebClient 已过时

2021/11/16 •

从 .NET 6 开始，将以下 API 标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0014`。

- WebRequest()
- System.Net.WebRequest.Create
- System.Net.WebRequest.CreateHttp
- System.Net.WebRequest.CreateDefault(Uri)
- HttpWebRequest(SerializationInfo, StreamingContext)
- System.Net.ServicePointManager.FindServicePoint
- WebClient()

解决方法

请改用 HttpClient。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

另请参阅

- WebRequest、WebClient 和 ServicePoint 已过时

SYSLIB0015：DisablePrivateReflectionAttribute 已过时

2021/11/16 •

从 .NET 6 开始，System.Runtime.CompilerServices.DisablePrivateReflectionAttribute 类型标记为已过时。此属性在 .NET Core 2.1 和更高版本应用程序中不起作用。对于 .NET 6 和更高版本的应用程序，在代码中使用会在编译时生成警告 `SYSLIB0015`。

有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/11811。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0016：GetContextInfo() 已过时

2021/11/16 •

从 .NET 6 开始，将不带参数的 Graphics.GetContextInfo() 方法标记为已过时。在代码中使用会在编译时生成警告 `SYSLIB0016`。

有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/47880。

解决方法

为改善性能并减少分配，使用接受参数的 Graphics.GetContextInfo 重载。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0017：强名称签名不受支持，并引发 PlatformNotSupportedException

2021/11/16 •

从 .NET 6 开始，以下 API 标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0017`。这些 API 在运行时引发 PlatformNotSupportedException。

- AssemblyName.KeyPair
- StrongNameKeyPair

有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/50529。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0018：仅反射加载不受支持，并引发 PlatformNotSupportedException

2021/11/16 •

从 .NET 6 开始，以下方法标记为已过时。在代码中调用这些方法会在编译时生成警告 `SYSLIB0018`。这些方法在运行时引发 PlatformNotSupportedException。

- Assembly.ReflectionOnlyLoad
- Assembly.ReflectionOnlyLoadFrom(String)
- Type.ReflectionOnlyGetType(String, Boolean, Boolean)

有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/50529。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0019：某些 RuntimeEnvironment API 已过时

2021/11/16 •

从 .NET 6 开始，将以下 API 标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0019`。

- RuntimeEnvironment.SystemConfigurationFile 属性
- RuntimeEnvironment.GetRuntimeInterfaceAsIntPtr(Guid, Guid) 方法
- RuntimeEnvironment.GetRuntimeInterfaceAsObject(Guid, Guid) 方法

这些 API 在运行时始终引发 PlatformNotSupportedException。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0020：IgnoreNullValues 已过时

2021/11/16 •

从 .NET 6 开始，JsonSerializerOptions.IgnoreNullValues 属性标记为已过时。在代码中使用会在编译时生成警告 `SYSLIB0020`。

解决方法

若要在序列化时忽略 null 值，请将 DefaultIgnoreCondition 设为 JsonIgnoreCondition.WhenWritingNull。有关详细信息，请参阅 https://github.com/dotnet/runtime/issues/39152。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0021：派生加密类型已过时

2021/11/16 •

从 .NET 6 开始，以下派生加密类型标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0021`。

- System.Security.Cryptography.AesCryptoServiceProvider
- System.Security.Cryptography.AesManaged
- System.Security.Cryptography.DESCryptoServiceProvider
- System.Security.Cryptography.MD5CryptoServiceProvider
- System.Security.Cryptography.RC2CryptoServiceProvider
- System.Security.Cryptography.SHA1CryptoServiceProvider
- System.Security.Cryptography.SHA1Managed
- System.Security.Cryptography.SHA256Managed
- System.Security.Cryptography.SHA256CryptoServiceProvider
- System.Security.Cryptography.SHA384Managed
- System.Security.Cryptography.SHA384CryptoServiceProvider
- System.Security.Cryptography.SHA512Managed
- System.Security.Cryptography.SHA512CryptoServiceProvider
- System.Security.Cryptography.TripleDESCryptoServiceProvider

解决方法

请改为对基类型使用 `Create` 方法。例如，请使用 TripleDES.Create 而不是 TripleDESCryptoServiceProvider。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0022：Rijndael 和 RijndaelManaged 类型已过时

2021/11/16 •

从 .NET 6 开始，Rijndael 和 RijndaelManaged 类型标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0022`。

解决方法

请改用 System.Security.Cryptography.Aes。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0023：RNGCryptoServiceProvider 已过时

2021/11/16 •

从 .NET 6 开始，RNGCryptoServiceProvider 标记为已过时。在代码中使用会在编译时生成警告 `SYSLIB0023`。

解决方法

若要生成随机数，请改为使用 RandomNumberGenerator 方法之一，例如 RandomNumberGenerator.GetInt32(Int32)。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0024：不支持创建和卸载 AppDomains，并引发异常

2021/11/16 •

从 .NET 6 开始，AppDomain.CreateDomain(String) 和 AppDomain.Unload(AppDomain) 方法标记为已过时。在代码中使用这些方法会在编译时生成警告 `SYSLIB0024`，并在运行时引发异常。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0025：SuppressIldasmAttribute 已过时

2021/11/16 •

从 .NET 6 开始，SuppressIldasmAttribute 类型标记为已过时。在代码中使用会在编译时生成警告 `SYSLIB0025`。 IL 反汇编程序 (ildasm.exe) 不再支持此属性。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0026：X509Certificate 和 X509Certificate2 是不可变的

2021/11/16 •

从 .NET 6 开始，以下可变 x509 证书 API 将标记为已过时。在代码中使用这些 API 会在编译时生成警告 `SYSLIB0026`。

- X509Certificate()
- X509Certificate.Import
- X509Certificate2()
- X509Certificate2.Import

解决方法

使用接受证书作为输入的构造函数重载创建 `X509Certificate` 和 `X509Certificate2` 的新实例。例如：

```
// Change this:
cert.Import("/path/to/certficate.crt");

// To this:
cert.Dispose();
cert = new X509Certificate2("/path/to/certificate.crt");
```

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0027：PublicKey.Key 已过时

2021/11/16 •

从 .NET 6 开始，PublicKey.Key 属性标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0027`。

解决方法

使用适当的方法获取公钥，如 GetRSAPublicKey()。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0028：X509Certificate2.PrivateKey 已过时

2021/11/16 •

从 .NET 6 开始，X509Certificate2.PrivateKey 属性标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0028`。

解决方法

使用适当的方法获取私钥（例如 RSACertificateExtensions.GetRSAPrivateKey(X509Certificate2)），或使用 X509Certificate2.CopyWithPrivateKey(ECDiffieHellman) 方法创建一个具有私钥的新实例。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0029：ProduceLegacyHmacValues 已过时

2021/11/16 •

从 .NET 6 开始，以下属性标记为已过时：

- HMACSHA384.ProduceLegacyHmacValues
- HMACSHA512.ProduceLegacyHmacValues

在代码中使用这些 API 会在编译时生成警告 `SYSLIB0029`。不再支持生成旧的 HMAC 值。

解决方法

无。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

> **NOTE**
> 以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0030：HMACSHA1 始终使用平台提供的算法实现

2021/11/16 •

从 .NET 6 开始，HMACSHA1(Byte[], Boolean) 构造函数标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0030`。

解决方法

使用不带 `useManagedSha1` 参数的构造函数。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0031：EncodeOID 已过时

2021/11/16 •

从 .NET 6 开始，CryptoConfig.EncodeOID(String) 方法标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0031`，并在运行时引发 PlatformNotSupportedException 异常。

解决方法

使用 System.Formats.Asn1 中提供的 ASN.1 功能。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0032：不支持从进程状态已损坏异常中恢复

2021/11/16 •

不支持从进程状态已损坏异常中恢复。从 .NET 6 开始，HandleProcessCorruptedStateExceptionsAttribute 类型标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0032`。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0033：Rfc2898DeriveBytes.CryptDeriveKey 已过时

2021/11/16 •

从 .NET 6 开始，Rfc2898DeriveBytes.CryptDeriveKey(String, String, Int32, Byte[]) 方法标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0033`。

解决方法

请改用 PasswordDeriveBytes.CryptDeriveKey(String, String, Int32, Byte[])。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0034：CmsSigner(CspParameters) 构造函数已过时

2021/11/16 •

从 .NET 6 开始，CmsSigner(CspParameters) 构造函数标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0034`。

解决方法

使用替代的构造函数。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

SYSLIB0035 ：不指定 CmsSigner 的 ComputeCounterSignature 已过时

2021/11/16 •

从 .NET 6 开始，SignerInfo.ComputeCounterSignature() 方法标记为已过时。在代码中使用此 API 会在编译时生成警告 `SYSLIB0035` 。

解决方法

使用接受 CmsSigner 的重载，即 SignerInfo.ComputeCounterSignature(CmsSigner)。

禁止显示警告

建议尽可能使用可用的解决方法。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果必须使用过时 API，并且 `SYSLIB0XXX` 诊断没有显示为错误，则可以在代码或项目文件中取消该警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB0001

// Code that uses obsolete API.
//...

// Re-enable the warning.
#pragma warning restore SYSLIB0001
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net5.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB0001 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB0001</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB0002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB0003</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB0001;SYSLIB0002;SYSLIB0003</NoWarn>
  </PropertyGroup>
</Project>
```

NOTE

以这种方式取消警告只会禁用指定的过时警告。它不会禁用任何其他警告，包括具有不同诊断 ID 的其他过时警告。

.NET 6+ 中的源生成器诊断

2021/11/16 •

如果你的 .NET 6+ 项目引用一个启用源生成代码的包(例如日志记录解决方案),则特定于源生成的分析器将在编译时运行。本文列出了与源生成代码相关的编译器诊断。

如果你遇到其中一个生成警告或错误,请按照为参考部分列出的诊断 ID 提供的具体指导进行操作。还可以使用特定的 `SYSLIB1XXX` 诊断 ID 值来取消警告。有关详细信息,请参阅取消警告。

分析器警告

为源生成代码分析器警告保留的诊断 ID 值为 `SYSLIB1001` 至 `SYSLIB1999`。

参考

下表提供了 .NET 6 及更高版本中 `SYSLIB1XXX` 诊断的索引。

ID	
SYSLIB1001	日志记录方法名称不能以 `_` 开头
SYSLIB1002	不要将日志级别参数作为模板包含在日志记录消息中
SYSLIB1003	`InvalidLoggingMethodParameterNameTitle`
SYSLIB1004	日志记录类不能位于嵌套类型中
SYSLIB1005	找不到所需的类型定义
SYSLIB1006	多个日志记录方法不能在类中使用相同的事件 ID
SYSLIB1007	日志记录方法必须返回 `void`
SYSLIB1008	日志记录方法的参数之一必须实现 `Microsoft.Extensions.Logging.ILogger` 接口
SYSLIB1009	日志记录方法必须为 `static`
SYSLIB1010	日志记录方法必须为 `partial`
SYSLIB1011	日志记录方法不能是泛型
SYSLIB1012	日志记录消息中的多余限定符
SYSLIB1013	不要将异常参数作为模板包含在日志记录消息中
SYSLIB1014	日志记录模板无相应的方法参数
SYSLIB1015	未从日志记录消息中引用参数

ID	
SYSLIB1016	日志记录方法不能有主体
SYSLIB1017	必须在 `LoggerMessage` 属性中提供 LogLevel 值或将其用作日志记录方法的参数
SYSLIB1018	不要将记录器参数作为模板包含在日志记录消息中
SYSLIB1019	找不到 `Microsoft.Extensions.Logging.ILogger` 类型的字段
SYSLIB1020	找到 `Microsoft.Extensions.Logging.ILogger` 类型的多个字段
SYSLIB1021	多个消息模板项名称只是大小写不同
SYSLIB1022	不能使用格式错误的格式字符串(例如不成对的大括号)
SYSLIB1023	不支持生成六个以上的参数

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1001：日志记录方法名称不能以下划线开头

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法的名称以下划线字符开头。不允许这种做法，因为这可能会导致符号名称与自动生成的代码相冲突。

解决方法

选择不以下划线开头的其他方法名称。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1002：不要将日志级别参数作为模板包含在日志记录消息中

2021/11/16 •

日志记录方法的第一个日志级别参数在日志记录消息中作为模板引用。不必要这样做，因为第一个日志级别会显式传递给日志记录基础结构。不需要在日志记录消息中重复它。

解决方法

从日志记录消息中删除引用日志级别参数的模板。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1003：日志记录方法参数名称不能以下划线开头

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法的参数名称以下划线字符开头。不允许这种做法，因为这可能会导致符号名称与自动生成的代码相冲突。

解决方法

选择不以下划线开头的其他参数名称。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1004：日志记录类不能位于嵌套类型中

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法包含在嵌套类型中。.NET 6 Preview 4 和 5 中的日志记录模型不支持这种做法。

> **NOTE**
> 从 .NET 6 Preview 6 开始，支持嵌套类中的记录器方法。

解决方法

在名为 `Log` 的顶级静态类中声明日志记录方法是一种惯例。此模式消除了嵌套类型的问题。

例如：

```
namespace MyService
{
    internal static partial class Log
    {
        [LoggerMessage(
            EventId = 0,
            Level = LogLevel.Critical,
            Message = "Could not open socket to `{hostName}`")]
        public static partial void CouldNotOpenSocket(
            ILogger logger, string hostName);
    }
}
```

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1005：找不到所需的类型定义

2021/11/16 •

代码生成器无法找到对已知类型的必要引用。

解决方法

此错误不太可能发生。如果确实发生这种错误，可以考虑将 `<PackageReference>` 添加到项目文件以包含所需的类型定义。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1006：多个日志记录方法不能使用相同的事件 ID

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的多个方法正在使用相同的事件 ID 值。事件 ID 值在每个程序集的范围内必须独一无二。

解决方法

查看程序集中所有日志记录方法使用的事件 ID 值，确保它们独一无二。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1007：日志记录方法必须返回 void

2021/11/16 •

使用 `LoggerMessageAttribute` 属性进行注释的方法返回了一个值。

解决方法

所有日志记录方法必须返回 void。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1008：日志记录方法的参数之一必须实现 `ILogger` 接口

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法的参数之一必须是 `ILogger` 类型或实现 `ILogger` 的类型。

解决方法

确保所有日志记录方法的参数的类型是 `ILogger`，或者是实现 `ILogger` 的类型。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1009：日志记录方法必须为静态方法

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法不是静态方法。

解决方法

所有日志记录方法必须声明为静态方法。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1010：日志记录方法必须为分部方法

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法未标记为分部方法。

解决方法

所有日志记录方法必须声明为分部方法。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1011：日志记录方法不能是泛型

2021/11/16 •

使用 `LoggerMessageAttribute` 进行注释的方法包含泛型类型的参数。

解决方法

日志记录方法不能包含任何泛型类型化参数。请改用完全解析的类型。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1012：日志记录消息中的多余限定符

2021/11/16 •

`LoggerMessageAttribute` 属性的消息字符串包含一个前缀（例如 `INFO:` 或 `ERROR:`），这是多余的，因为每个日志消息都有相应的日志级别。

解决方法

从消息字符串中删除前缀。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1013：不要将异常参数作为模板包含在日志记录消息中

2021/11/16 •

日志记录方法的第一个异常参数在日志记录消息中作为模板引用。不必要这样做，因为第一个异常将显式传递给日志记录基础结构。不需要在日志记录消息中重复它。

解决方法

从日志记录消息中删除引用异常参数的模板。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1014：日志记录模板无相应的方法参数

2021/11/16 •

日志记录消息中的模板在日志记录方法定义中没有匹配的参数。

解决方法

确保日志记录消息的所有模板在日志记录方法定义中都有相应的参数。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1015：未从日志记录消息中引用参数

2021/11/16 •

日志记录方法中的参数在日志记录消息中没有相应的模板。

解决方法

确保日志记录方法的所有参数在关联的日志记录消息中具有相应的模板。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1016：日志记录方法不能有主体

2021/11/16 •

`LoggerMessageAttribute` 属性已应用于具有方法主体的方法。

解决方法

删除方法主体。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1017：必须在 `LoggerMessage` 属性中提供 `LogLevel` 值或将其用作日志记录方法的参数

2021/11/16 •

`LoggerMessageAttribute` 属性应用于未指定 `LogLevel` 值的方法。执行此操作时，日志记录方法的一个参数必须属于该类型，以便在调用日志记录方法时最终明确指定 `LogLevel` 值。

解决方法

在 `LoggerMessage` 属性中指定 `LogLevel` 值，或将日志记录方法的参数之一设为 `LogLevel` 值。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1018：不要将记录器参数作为模板包含在日志记录消息中

2021/11/16 •

日志记录方法的第一个记录器参数在日志记录消息中作为模板引用。第一个记录器显式传递给日志记录基础结构，因此不需要在日志记录消息中重复它。

解决方法

从日志记录消息中删除引用记录器参数的模板。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1019：找不到 `ILogger` 类型的字段

2021/11/16 •

如果日志记录方法定义未显式包含 `ILogger` 类型的参数，则包含日志记录方法的类型必须有一个（且只有一个）类型为 `ILogger` 的字段。 `ILogger` 将用作日志消息的目标。

解决方法

确保包含日志记录方法的类型包含 `ILogger` 类型的字段，或者在日志记录方法签名中包含 `ILogger` 类型的参数。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1020：找到 `ILogger` 类型的多个字段

2021/11/16 •

如果日志记录方法定义未显式包含 `ILogger` 类型的参数，则包含日志记录方法的类型必须有一个（且只有一个）类型为 `ILogger` 的字段，该字段将用作日志消息的目标。

解决方法

确保包含日志记录方法的类型仅包含类型为 `ILogger` 的单个字段。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1021：多个消息模板项名称只是大小写不同

2021/11/16 •

使用 `LoggerMessageAttribute` 属性进行注释的方法具有多个模板项名称，而这些名称只是大小写不同。

解决方法

确保在使用 `LoggerMessageAttribute` 进行注释的方法中的消息模板项名称不是只有大小写不同的重复。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1022：不能有格式错误的格式字符串

2021/11/16 •

使用 `LoggerMessageAttribute` 属性注释的方法具有格式不正确的消息模板。例如，模板具有不匹配的大括号 (`}`)。

解决方法

请确保在消息模板中正确使用大括号。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

SYSLIB1023：不支持生成六个以上的参数

2021/11/16 •

`LoggerMessageAttribute` 限制为 `LoggerMessage.Define` 支持的参数数目，即 6。

解决方法

考虑使用结构实现自定义属性，而不要使用 `LoggerMessageAttribute`。

禁止显示警告

建议尽量使用解决方法之一。但是，如果无法更改代码，可以通过 `#pragma` 指令或 `<NoWarn>` 项目设置来禁止显示警告。如果 `SYSLIB1XXX` 源生成器诊断未显示为错误，则可以在代码或项目文件中禁止警告。

若要禁止显示代码中的警告，请执行以下操作：

```
// Disable the warning.
#pragma warning disable SYSLIB1006

// Code that generates compiler diagnostic.
// ...

// Re-enable the warning.
#pragma warning restore SYSLIB1006
```

若要禁止显示项目文件中的警告，请执行以下操作：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
   <TargetFramework>net6.0</TargetFramework>
   <!-- NoWarn below suppresses SYSLIB1002 project-wide -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <!-- To suppress multiple warnings, you can use multiple NoWarn elements -->
   <NoWarn>$(NoWarn);SYSLIB1002</NoWarn>
   <NoWarn>$(NoWarn);SYSLIB1006</NoWarn>
   <!-- Alternatively, you can suppress multiple warnings by using a semicolon-delimited list -->
   <NoWarn>$(NoWarn);SYSLIB1002;SYSLIB1006;SYSLIB1007</NoWarn>
  </PropertyGroup>
</Project>
```

.NET 项目 SDK

2021/11/16 •

.NET Core 和 .NET 5 及更高版本项目与软件开发工具包 (SDK) 关联。每个项目 SDK 都是一组 MSBuild 目标和相关的任务，它们负责编译、打包和发布代码。引用项目 SDK 的项目有时称为“SDK 样式的项目”。

可用的 SDK

有以下 SDK 可用：

ID		
`Microsoft.NET.Sdk`	.NET SDK	https://github.com/dotnet/sdk
`Microsoft.NET.Sdk.Web`	.NET Web SDK	https://github.com/dotnet/sdk
`Microsoft.NET.Sdk.BlazorWebAssembly`	The .NET Blazor WebAssembly SDK	
`Microsoft.NET.Sdk.Razor`	.NET Razor SDK	
`Microsoft.NET.Sdk.Worker`	.NET 辅助角色服务 SDK	
`Microsoft.NET.Sdk.WindowsDesktop`	.NET 桌面 SDK，其中包括 Windows 窗体 (WinForms) 和 Windows Presentation Foundation (WPF)。*	https://github.com/dotnet/winforms 和 https://github.com/dotnet/wpf

.NET SDK 是 .NET 的基本 SDK。其他 SDK 引用 .NET SDK，与其他 SDK 关联的项目具有所有可用的 .NET SDK 属性。例如，Web SDK 依赖于 .NET SDK 和 Razor SDK。

你还可以创建自己的 SDK，并通过 NuGet 进行分发。

* 从 .NET 5 开始，Windows 窗体和 Windows Presentation Foundation (WPF) 项目应指定 .NET SDK (`Microsoft.NET.Sdk`)，而不是 `Microsoft.NET.Sdk.WindowsDesktop`。对于这些项目，将 `TargetFramework` 设置为 `net5.0-windows` 并将 `UseWPF` 或 `UseWindowsForms` 设置为 `true` 的操作会自动导入 Windows 桌面 SDK。如果你的项目面向 .NET 5 或更高版本，并指定 `Microsoft.NET.Sdk.WindowsDesktop` SDK，则会收到生成警告 NETSDK1137。

项目文件

.NET 项目基于 MSBuild 格式。具有扩展名（如用于 C# 项目的 .csproj 和用于 F# 项目的 .fsproj）的项目文件都是 XML 格式的 。MSBuild 项目文件的根元素是 Project 元素。`Project` 元素有一个可选的 `Sdk` 属性，该属性指定要使用的 SDK（和版本）。若要使用 .NET 工具并构建你的代码，请将 `Sdk` 属性设置为可用 SDK 表中的其中一个 ID。

```
<Project Sdk="Microsoft.NET.Sdk">
  ...
</Project>
```

若要指定来自 NuGet 的 SDK，请在名称末尾包含版本，或者在 global.json 文件中指定名称和版本。

```
<Project Sdk="MSBuild.Sdk.Extras/2.0.54">
  ...
</Project>
```

另一种指定 SDK 的方法是使用顶层 Sdk 元素：

```
<Project>
  <Sdk Name="Microsoft.NET.Sdk" />
  ...
</Project>
```

以这些方式之一引用 SDK 可以极大地简化 .NET 的项目文件。在评估项目时，MSBuild 在项目文件的顶部和底部分别为 `Sdk.props` 和 `Sdk.targets` 添加隐式导入。

```
<Project>
  <!-- Implicit top import -->
  <Import Project="Sdk.props" Sdk="Microsoft.NET.Sdk" />
  ...
  <!-- Implicit bottom import -->
  <Import Project="Sdk.targets" Sdk="Microsoft.NET.Sdk" />
</Project>
```

> **TIP**
> 在 Windows 计算机上，Sdk.props 和 Sdk.targets 文件位于 %ProgramFiles%\dotnet\sdk\[version]\Sdks\Microsoft.NET.Sdk\Sdk 文件夹中 。

预处理项目文件

使用 `dotnet msbuild -preprocess` 命令，可以看到 MSBuild 在包含 SDK 及其目标之后所显示的完全扩展的项目。`dotnet msbuild` 命令的预处理开关显示导入的文件、文件源及其在生成中的参与情况，而无需实际生成项目。

如果项目有多个目标框架，请将命令的结果指定为 MSBuild 属性，使其仅侧重于框架之一。例如：

```
dotnet msbuild -property:TargetFramework=netcoreapp2.0 -preprocess:output.xml
```

默认包含和排除的内容

SDK 中定义了 `Compile` 项、嵌入的资源和 `None` 项默认包含和排除的内容。与非 SDK .NET 框架项目不同，你无需在项目文件中指定这些项，因为默认设置涵盖了最常见的用例。此行为使得项目文件更小、更易于理解和手动编辑（如需要）。

下表显示在 .NET SDK 中包含和排除的元素和 glob：

项	项 GLOB	项 GLOB	项 GLOB
Compile	**/*.cs（或其他语言扩展名）	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	不可用
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	不可用
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	**/*.cs; **/*.resx

> **NOTE**
> 默认情况下，由 `$(BaseOutputPath)` 和 `$(BaseIntermediateOutputPath)` MSBuild 属性表示的 `./bin` 和 `./obj` 文件夹不包含在 glob 中。排除由 DefaultItemExcludes 属性表示。

.NET 桌面 SDK 对于 WPF 有更多包含和排除的内容。有关详细信息，请参阅 WPF 默认包含和排除的内容。

生成错误

如果在项目文件中显式定义这些项中的任何项，可能会出现类似于以下内容的“NETSDK1022”生成错误：

> 包含重复的“Compile”项。默认情况下，.NET SDK 包括项目目录中的“Compile”项。可从项目文件中删除这些项，或如果想要在项目文件中显式包括它们，则将“EnableDefaultCompileItems”属性设为“false”。

> 包含重复的“EmbeddedResource”项。默认情况下，.NET SDK 包括项目目录中的“EmbeddedResource”项。可从项目文件中删除这些项，或如果想要在项目文件中显式包括它们，则将“EnableDefaultEmbeddedResourceItems”属性设为“false”。

若要解决此错误，请执行以下操作之一：

- 删除与上表中列出的隐式项匹配的显式 `Compile`、`EmbeddedResource` 或 `None` 项。
- 若要禁用所有隐式文件包含，请将 EnableDefaultItems 属性设置为 `false`：

  ```
  <PropertyGroup>
    <EnableDefaultItems>false</EnableDefaultItems>
  </PropertyGroup>
  ```

  若要指定某些文件通过应用发布，仍可以使用相应的已知 MSBuild 机制来实现（例如 `Content` 元素）。

- 可选择仅禁用 `Compile`、`EmbeddedResource` 或 `None` glob，方法是将 EnableDefaultCompileItems、EnableDefaultEmbeddedResourceItems 或 EnableDefaultNoneItems 属性设置为 `false`：

  ```
  <PropertyGroup>
    <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
    <EnableDefaultEmbeddedResourceItems>false</EnableDefaultEmbeddedResourceItems>
    <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
  </PropertyGroup>
  ```

  如果仅禁用 `Compile` glob，则 Visual Studio 中的解决方案资源管理器仍将 *.cs 项显示为项目的一部分，并作为 `None` 项包含在内。若要禁用隐式 `None` glob，请将 `EnableDefaultNoneItems` 也设置为 `false`。

隐式 using 指令

从 .NET 6 开始，隐式 `global using` 指令将添加到新的 C# 项目中。这意味着可以使用这些命名空间中定义的类型，而无需指定完全限定的名称或手动添加 `using` 指令。隐式方面是指向项目的 obj 目录中生成的文件添加 `global using` 指令这一事实。

为使用以下 SDK 之一的项目添加隐式 `global using` 指令：

- Microsoft.NET.Sdk
- Microsoft.NET.Sdk.Web
- Microsoft.NET.Sdk.Worker
- Microsoft.NET.Sdk.WindowsDesktop

为基于项目 SDK 的一组默认命名空间中的每个命名空间添加 `global using` 指令。下表显示了这些默认命名空间。

SDK ISINROLE	
Microsoft.NET.Sdk	System System.Collections.Generic System.IO System.Linq System.Net.Http System.Threading System.Threading.Tasks
Microsoft.NET.Sdk.Web	System.Net.Http.Json Microsoft.AspNetCore.Builder Microsoft.AspNetCore.Hosting Microsoft.AspNetCore.Http Microsoft.AspNetCore.Routing Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging
Microsoft.NET.Sdk.Worker	Microsoft.Extensions.Configuration Microsoft.Extensions.DependencyInjection Microsoft.Extensions.Hosting Microsoft.Extensions.Logging
Microsoft.NET.Sdk.WindowsDesktop（Windows 窗体）	Microsoft.NET.Sdk 命名空间 System.Drawing System.Windows.Forms
Microsoft.NET.Sdk.WindowsDesktop (WPF)	Microsoft.NET.Sdk 命名空间 已删除 System.IO 已删除 System.Net.Http

若要禁用此功能，或要在现有的 C# 项目中启用隐式 `global using` 指令，可通过 `ImplicitUsings` MSBuild 属性实现。如果需要，可以通过向项目文件添加 `Using` 项来添加其他隐式 `global using` 指令，例如：

```
<ItemGroup>
  <Using Include="System.IO.Pipes" />
</ItemGroup>
```

隐式包引用

如果以 .NET Core 1.0 - 2.2 或 .NET Standard 1.0 - 2.0 为目标，则 .NET SDK 会添加对某些元包的隐式引用。元包是一种基于框架的包，其中只包含对其他包的依赖项。元包根据项目文件的 TargetFramework 或 TargetFrameworks 属性中指定的目标框架被隐式引用。

```
<PropertyGroup>
  <TargetFramework>netcoreapp2.1</TargetFramework>
</PropertyGroup>
```

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp2.1;net462</TargetFrameworks>
</PropertyGroup>
```

如果需要，可以使用 DisableImplicitFrameworkReferences 属性来禁用隐式包引用，并只添加对所需的框架或包的显式引用。

建议：

- 如果以 .NET Framework、.NET Core 1.0 - 2.2 或 .NET Standard 1.0 - 2.0 为目标，不要通过项目文件中的 `<PackageReference>` 项添加对 `Microsoft.NETCore.App` 或 `NETStandard.Library` 元包的显式引用。对于 .NET Core 1.0 - 2.2 和 .NET Standard 1.0 - 2.0 项目，这些元包被隐式引用。对于 .NET Framework 项目，如果在使用基于 .NET Standard 的 NuGet 包时需要任何版本的 `NETStandard.Library`，则 NuGet 会自动安装相应版本。
- 如果在以 .NET Core 1.0 - 2.2 为目标时需要特定版本的运行时，请在项目中使用 `<RuntimeFrameworkVersion>` 属性（例如，`1.0.4`），而不是引用元包。例如，如果在使用独立式部署，则可能需要特定补丁版本的 1.0.0 LTS 运行时。
- 如果在以 .NET Standard 1.0 - 2.0 为目标时需要特定版本的 `NETStandard.Library` 元包，则可以使用 `<NetStandardImplicitPackageVersion>` 属性，并设置所需的版本。

生成事件

在 SDK 样式的项目中，请使用名为 `PreBuild` 或 `PostBuild` 的 MSBuild 目标，并设置 `PreBuild` 的 `BeforeTargets` 属性或 `PostBuild` 的 `AfterTargets` 属性。

```
<Target Name="PreBuild" BeforeTargets="PreBuildEvent">
    <Exec Command="&quot;$(ProjectDir)PreBuildEvent.bat&quot; &quot;$(ProjectDir)..\&quot;
&quot;$(ProjectDir)&quot; &quot;$(TargetDir)&quot;" />
</Target>

<Target Name="PostBuild" AfterTargets="PostBuildEvent">
   <Exec Command="echo Output written to $(TargetDir)" />
</Target>
```

NOTE

- 可以为 MSBuild 目标使用任何名称。但是，Visual Studio IDE 会识别 `PreBuild` 和 `PostBuild` 目标，因此通过使用这些名称，可以在 IDE 中编辑命令。
- 不建议在 SDK 样式的项目中使用属性 `PreBuildEvent` 和 `PostBuildEvent`，因为无法解析 `$(ProjectDir)` 这样的宏。例如，以下代码是不受支持的：

```
<PropertyGroup>
  <PreBuildEvent>"$(ProjectDir)PreBuildEvent.bat" "$(ProjectDir)..\" "$(ProjectDir)" "$(TargetDir)"
</PreBuildEvent>
</PropertyGroup>
```

自定义生成

可以通过多种方式自定义生成。建议通过将属性作为参数传递给 msbuild 或 dotnet 命令来重写该属性。还可以将属性添加到项目文件或 Directory.Build.props 文件中。有关 .NET 项目的有用属性列表，请参见 .NET SDK 项目的 MSBuild 参考。

自定义目标

.NET 项目可以打包自定义的 MSBuild 目标和属性，以供使用该包的项目使用。如果要执行以下操作，请使用此

类型的可扩展性：

- 扩展生成过程。
- 访问生成过程的工件，如生成的文件。
- 检查调用生成的配置。

通过在项目的生成文件夹中以 `<package_id>.targets` 或 `<package_id>.props`（例如 `Contoso.Utility.UsefulStuff.targets`）的形式放置文件，可以添加自定义生成目标或属性。

以下 XML 是 .csproj 文件中的一个片段，该文件指示 `dotnet pack` 命令打包的内容。 `<ItemGroup Label="dotnet pack instructions">` 元素将目标文件放入包内的生成文件夹中。 `<Target Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">` 元素将程序集和 .json 文件放入生成文件夹 。

```
<Project Sdk="Microsoft.NET.Sdk">

  ...
  <ItemGroup Label="dotnet pack instructions">
    <Content Include="build\*.targets">
      <Pack>true</Pack>
      <PackagePath>build\</PackagePath>
    </Content>
  </ItemGroup>
  <Target Name="CollectRuntimeOutputs" BeforeTargets="_GetPackageFiles">
    <!-- Collect these items inside a target that runs after build but before packaging. -->
    <ItemGroup>
      <Content Include="$(OutputPath)\*.dll;$(OutputPath)\*.json">
        <Pack>true</Pack>
        <PackagePath>build\</PackagePath>
      </Content>
    </ItemGroup>
  </Target>
  ...

</Project>
```

若要在项目中使用自定义目标，请添加指向包及其版本的 `PackageReference` 元素。与工具不同，自定义目标包包含在消费项目的依赖项闭包中。

你可以配置自定义目标的使用方式。由于它是 MSBuild 目标，因此会依赖于给定的目标并在另一个目标后运行，也可使用 `dotnet msbuild -t:<target-name>` 命令手动调用。若要提供更好的用户体验，可以合并基于项目的工具和自定义目标。在此方案中，每个项目工具接受所需的任何参数，并将其转换为执行目标所需的 `dotnet msbuild` 调用。有关此类协同作用的示例，请访问 `dotnet-packer` 项目中的 2016 年编程马拉松 MVP 峰会示例存储库。

请参阅

- 安装 .NET Core
- 如何使用 MSBuild 项目 SDK
- 使用 NuGet 打包自定义 MSBuild 目标和属性

.NET SDK 项目的 MSBuild 引用

2021/11/16 •

此页是对可用于配置 .NET 项目的 MSBuild 属性和项的引用。

> **NOTE**
> 此页面正在运行中，未列出 .NET SDK 的所有有用的 MSBuild 属性。有关通用 MSBuild 属性的列表，请参阅通用 MSBuild 属性。

框架属性

本节收录了以下 MSBuild 属性：

- TargetFramework
- TargetFrameworks
- NetStandardImplicitPackageVersion

TargetFramework

`TargetFramework` 属性指定应用的目标框架版本。有关有效的目标框架名字对象的列表，请参阅 SDK 样式项目中的目标框架。

```
<PropertyGroup>
  <TargetFramework>netcoreapp3.1</TargetFramework>
</PropertyGroup>
```

有关详细信息，请参阅 SDK 样式项目中的目标框架。

TargetFrameworks

如果希望应用面向多个平台，请使用 `TargetFrameworks` 属性。有关有效的目标框架名字对象的列表，请参阅 SDK 样式项目中的目标框架。

> **NOTE**
> 如果指定了 `TargetFramework`（单数），则忽略此属性。

```
<PropertyGroup>
  <TargetFrameworks>netcoreapp3.1;net462</TargetFrameworks>
</PropertyGroup>
```

有关详细信息，请参阅 SDK 样式项目中的目标框架。

NetStandardImplicitPackageVersion

> **NOTE**
> 此属性仅适用于使用 `netstandard1.x` 的项目。它不适用于使用 `netstandard2.x` 的项目。

如果要指定低于元包版本的框架版本，请使用 `NetStandardImplicitPackageVersion` 属性。以下示例中的项目文件以 `netstandard1.3` 为目标，但使用 `NETStandard.Library` 的 1.6.0 版本。

```
<PropertyGroup>
  <TargetFramework>netstandard1.3</TargetFramework>
  <NetStandardImplicitPackageVersion>1.6.0</NetStandardImplicitPackageVersion>
</PropertyGroup>
```

程序集特性属性

- GenerateAssemblyInfo
- GenerateAssemblyInfoFile

GenerateAssemblyInfo

`GenerateAssemblyInfo` 属性控制项目的 `AssemblyInfo` 特性生成。默认值为 `true`。使用 `false` 禁用文件生成：

```
<PropertyGroup>
  <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
</PropertyGroup>
```

GeneratedAssemblyInfoFile 设置控制生成的文件的名称。

当 `GenerateAssemblyInfo` 值为 `true` 时，与包相关的项目属性将转换为程序集特性。下表列出了用于生成特性的项目属性。还列出了可用于基于每个特性禁用该生成的属性，例如：

```
<PropertyGroup>
  <GenerateNeutralResourcesLanguageAttribute>false</GenerateNeutralResourcesLanguageAttribute>
</PropertyGroup>
```

MSBUILD □□	□□□□	□□□□□□□□
`Company`	AssemblyCompanyAttribute	`GenerateAssemblyCompanyAttribute`
`Configuration`	AssemblyConfigurationAttribute	`GenerateAssemblyConfigurationAttribute`
`Copyright`	AssemblyCopyrightAttribute	`GenerateAssemblyCopyrightAttribute`
`Description`	AssemblyDescriptionAttribute	`GenerateAssemblyDescriptionAttribute`
`FileVersion`	AssemblyFileVersionAttribute	`GenerateAssemblyFileVersionAttribute`
`InformationalVersion`	AssemblyInformationalVersionAttribute	`GenerateAssemblyInformationalVersionAttribut`
`Product`	AssemblyProductAttribute	`GenerateAssemblyProductAttribute`
`AssemblyTitle`	AssemblyTitleAttribute	`GenerateAssemblyTitleAttribute`
`AssemblyVersion`	AssemblyVersionAttribute	`GenerateAssemblyVersionAttribute`
`NeutralLanguage`	NeutralResourcesLanguageAttribute	`GenerateNeutralResourcesLanguageAttribute`

有关这些设置的说明：

- `AssemblyVersion` 和 `FileVersion` 默认采用 `$(Version)` 的值而不带后缀。例如，如果 `$(Version)` 为 `1.2.3-beta.4`，则值将为 `1.2.3`。

- `InformationalVersion` 默认是 `$(Version)` 的值。
- 如果存在 `$(SourceRevisionId)` 属性，则该属性会附加到 `InformationalVersion`。可以使用 `IncludeSourceRevisionInInformationalVersion` 禁用此行为。
- `Copyright` 和 `Description` 属性也可用于 NuGet 元数据。
- `Configuration`（默认值为 `Debug`）由所有 MSBuild 目标共享。可以通过 `dotnet` 命令的 `--configuration` 选项对其进行设置，例如 dotnet pack。
- 创建 NuGet 包时会使用某些属性。有关详细信息，请参阅包属性。

从 .NET Framework 迁移

.NET Framework 项目模板会创建一个包含这些程序集信息属性集的代码文件。该文件通常位于 .\Properties\AssemblyInfo.cs 或 .\Properties\AssemblyInfo.vb 。SDK 样式项目基于项目设置为你生成此文件。你不能同时具有这两者。将代码移植到 .NET 5（或 .NET Core 3.1）或更高版本时，请执行以下操作之一：

- 通过在项目文件中将 `GenerateAssemblyInfo` 设置为 `false` 来禁用包含程序集信息特性的临时代码文件的生成。这使你可以保留 AssemblyInfo 文件。
- 将 `AssemblyInfo` 文件中的设置迁移到项目文件，然后删除该 `AssemblyInfo` 文件。

GeneratedAssemblyInfoFile

`GeneratedAssemblyInfoFile` 属性定义生成的程序集信息文件的相对或绝对路径。默认为 `$(IntermediateOutputPath)`（通常为 obj）目录中名为 [project-name].AssemblyInfo.[cs|vb] 的文件 。

```
<PropertyGroup>
  <GenerateAssemblyInfoFile>assemblyinfo.cs</GenerateAssemblyInfoFile>
</PropertyGroup>
```

包属性

可以指定 `PackageId`、`PackageVersion`、`PackageIcon`、`Title` 和 `Description` 等属性来描述通过项目创建的包。若要了解这些属性和其他属性，请参阅包目标。

```
<PropertyGroup>
  ...
  <PackageId>ClassLibDotNetStandard</PackageId>
  <Version>1.0.0</Version>
  <Authors>John Doe</Authors>
  <Company>Contoso</Company>
</PropertyGroup>
```

与发布相关的属性

本节收录了以下 MSBuild 属性：

- AppendRuntimeIdentifierToOutputPath
- AppendTargetFrameworkToOutputPath
- CopyLocalLockFileAssemblies
- EnablePackageValidation
- ErrorOnDuplicatePublishOutputFiles
- GenerateRuntimeConfigurationFiles
- IsPublishable
- PreserveCompilationContext
- PreserveCompilationReferences
- RollForward
- RuntimeFrameworkVersion

- RuntimeIdentifier
- RuntimeIdentifiers
- SatelliteResourceLanguages
- UseAppHost

AppendTargetFrameworkToOutputPath

`AppendTargetFrameworkToOutputPath` 属性控制是否将目标框架名字对象 (TFM) 追加到输出路径(由 OutputPath 定义)。.NET SDK 会自动将目标框架以及运行时标识符(如果有)追加到输出路径。将 `AppendTargetFrameworkToOutputPath` 设置为 `false` 可防止将 TFM 追加到输出路径。但是，如果输出路径中没有 TFM，则可能会发生多个生成项目相互覆盖的情况。

例如，对于 .NET 5 应用，输出路径将从 `bin\Debug\net5.0` 更改为 `bin\Debug`，并具有以下设置：

```
<PropertyGroup>
  <AppendTargetFrameworkToOutputPath>false</AppendTargetFrameworkToOutputPath>
</PropertyGroup>
```

AppendRuntimeIdentifierToOutputPath

`AppendRuntimeIdentifierToOutputPath` 属性控制是否将运行时标识符 (RID) 追加到输出路径。.NET SDK 会自动将目标框架以及运行时标识符(如果有)追加到输出路径。将 `AppendRuntimeIdentifierToOutputPath` 设置为 `false` 可防止将 RID 追加到输出路径。

例如，对于 .NET 5 应用和 RID `win10-x64`，输出路径将从 `bin\Debug\net5.0\win10-x64` 更改为 `bin\Debug\net5.0`，并具有以下设置：

```
<PropertyGroup>
  <AppendRuntimeIdentifierToOutputPath>false</AppendRuntimeIdentifierToOutputPath>
</PropertyGroup>
```

CopyLocalLockFileAssemblies

对于依赖于其他库的插件项目，`CopyLocalLockFileAssemblies` 属性非常有用。如果将此属性设置为 `true`，系统会将所有 NuGet 包依赖项复制到输出目录。这意味着，可以使用 `dotnet build` 的输出在任何计算机上运行插件。

```
<PropertyGroup>
  <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
</PropertyGroup>
```

> **TIP**
>
> 或者，可以使用 `dotnet publish` 发布类库。有关详细信息，请查看 dotnet publish。

ErrorOnDuplicatePublishOutputFiles

`ErrorOnDuplicatePublishOutputFiles` 属性与当 MSBuild 在发布输出中检测到重复文件时 SDK 是否生成错误 NETSDK1148 有关，但无法确定要删除的文件。如果不希望生成错误，请将 `ErrorOnDuplicatePublishOutputFiles` 属性设置为 `false`。

```
<PropertyGroup>
  <ErrorOnDuplicatePublishOutputFiles>false</ErrorOnDuplicatePublishOutputFiles>
</PropertyGroup>
```

此属性是在 .NET 6 中引入的。

EnablePackageValidation

`EnablePackageValidation` 属性在 `pack` 任务后对包启用一系列验证。有关详细信息，请参阅包验证。

```
<PropertyGroup>
  <EnablePackageValidation>true</EnablePackageValidation>
</PropertyGroup>
```

此属性是在 .NET 6 中引入的。

GenerateRuntimeConfigurationFiles

`GenerateRuntimeConfigurationFiles` 属性控制运行时配置选项是否从 runtimeconfig.template.json 文件复制到 [appname].runtimeconfig.json 文件。对于需要 runtimeconfig.json 文件的应用（即，其 `OutputType` 为 `Exe` 的应用），此属性默认设置为 `true`。

```
<PropertyGroup>
  <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
</PropertyGroup>
```

IsPublishable

`IsPublishable` 属性允许 `Publish` 目标运行。此属性仅影响使用 .*proj 文件和 `Publish` 目标的进程，例如 dotnet publish 命令。它不会影响 Visual Studio 中使用 `PublishOnly` 目标的发布。默认值为 `true`。

如果在解决方案文件上运行 `dotnet publish`，则此属性很有用，因为它允许自动选择应发布的项目。

```
<PropertyGroup>
  <IsPublishable>false</IsPublishable>
</PropertyGroup>
```

PreserveCompilationContext

`PreserveCompilationContext` 属性允许已构建或已发布的应用程序在运行时使用与构建时使用的相同设置来编译更多代码。在构建时引用的程序集将被复制到输出目录的 ref 子目录中。引用程序集的名称与传递给编译器的选项一起存储在应用程序的 .deps.json 文件中。可使用 DependencyContext.CompileLibraries 和 DependencyContext.CompilationOptions 属性检索此信息。

此功能主要由 ASP.NET Core MVC 和 Razor Pages 在内部使用，以支持 Razor文件的运行时编译。

```
<PropertyGroup>
  <PreserveCompilationContext>true</PreserveCompilationContext>
</PropertyGroup>
```

PreserveCompilationReferences

`PreserveCompilationReferences` 属性与 PreserveCompilationContext 属性类似，只不过它仅将引用的程序集复制到发布目录，而不是 .deps.json 文件。

```
<PropertyGroup>
  <PreserveCompilationReferences>true</PreserveCompilationReferences>
</PropertyGroup>
```

有关详细信息，请参阅 Razor SDK 属性。

RollForward

`RollForward` 属性控制应用程序在有多个运行时版本可用时如何选择运行时。此值作为 `rollForward` 设置输出到 .runtimeconfig.js。

```
<PropertyGroup>
  <RollForward>LatestMinor</RollForward>
</PropertyGroup>
```

将 `RollForward` 设置为以下值之一：

“[”	“[”
`Minor`	如果未指定，则为默认值。 如果缺少所请求的次要版本，则前滚到最低的较高次要版本。 如果存在所请求的次要版本，则使用 `LatestPatch` 策略。
`Major`	如果缺少所请求的主要版本，则前滚到下一个可用的更高主要版本和最低的次要版本。如果存在所请求的主要版本，则使用 `Minor` 策略。
`LatestPatch`	前滚到最高补丁版本。此值会禁用次要版本前滚。
`LatestMinor`	即使存在所请求的次要版本，仍前滚到最高次要版本。
`LatestMajor`	即使存在所请求的主要版本，仍前滚到最高主要版本和最高次要版本。
`Disable`	不要前滚，仅绑定到指定的版本。建议不要将此策略用于一般用途，因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

有关详细信息，请参阅控制前滚行为。

RuntimeFrameworkVersion

`RuntimeFrameworkVersion` 属性指定发布时要使用的运行时版本。指定运行时版本：

```
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

当发布依赖于框架的应用程序时，此值指定所需的最低版本。当发布自包含的应用程序时，此值指定所需的确切版本。

RuntimeIdentifier

`RuntimeIdentifier` 属性可用于指定项目的单个运行时标识符 (RID)。RID 支持发布独立部署。

```
<PropertyGroup>
  <RuntimeIdentifier>ubuntu.16.04-x64</RuntimeIdentifier>
</PropertyGroup>
```

RuntimeIdentifiers

`RuntimeIdentifiers` 属性可用于指定项目的运行时标识符 (RID) 的列表（以分号分隔）。如果需要为多个运行时发布，请使用此属性。`RuntimeIdentifiers` 在还原时使用，以确保图中有适当的资产。

TIP

`RuntimeIdentifier`（单数）可以在只需要一个运行时时提供更快的生成。

```
<PropertyGroup>
  <RuntimeIdentifiers>win10-x64;osx.10.11-x64;ubuntu.16.04-x64</RuntimeIdentifiers>
</PropertyGroup>
```

SatelliteResourceLanguages

`SatelliteResourceLanguages` 属性允许你指定要在生成和发布过程中为哪些语言保留附属资源程序集。许多 NuGet 包将本地化资源附属程序集包含在主包中。对于引用不需要本地化资源的 NuGet 包的项目，本地化程序集可能会不必要地增加生成和发布输出大小。通过将 `SatelliteResourceLanguages` 属性添加到项目文件，只会将你指定的语言的本地化程序集包含在生成和发布输出中。例如，在以下项目文件中，将只保留美国英语的资源附属程序集。

```
<PropertyGroup>
  <SatelliteResourceLanguages>en-US</SatelliteResourceLanguages>
</PropertyGroup>
```

> **NOTE**
> 必须在引用包含本地化资源附属程序集的 NuGet 包的项目中指定此属性。

UseAppHost

`UseAppHost` 属性控制是否为部署创建本机可执行文件。自包含部署需要本机可执行文件。

在 .NET Core3.0 及更高版本中，默认情况下会创建依赖于框架的可执行文件。将 `UseAppHost` 属性设置为 `false` 可禁用可执行文件的生成。

```
<PropertyGroup>
  <UseAppHost>false</UseAppHost>
</PropertyGroup>
```

有关部署的详细信息，请参阅 .NET 应用程序部署。

与编译相关的属性

本节收录了以下 MSBuild 属性：

- EmbeddedResourceUseDependentUponConvention
- EnablePreviewFeatures
- GenerateRequiresPreviewFeaturesAttribute
- OptimizeImplicitlyTriggeredBuild

还可在项目文件中将 C# 编译器选项指定为 MSBuild 属性。有关详细信息，请参阅 C# 编译器选项。

EmbeddedResourceUseDependentUponConvention

`EmbeddedResourceUseDependentUponConvention` 属性定义了是否从与资源文件并置的源文件中的类型信息生成资源清单文件名。例如，如果 Form1.resx 与 Form1.cs 位于同一文件夹中，并且 `EmbeddedResourceUseDependentUponConvention` 设置为 `true`，则生成的 .resources 文件将采用 Form1.cs 中定义的第一个类型的名称作为其文件名。例如，如果 Form1.cs 中定义的第一个类型为 `MyNamespace.Form1`，则生成的文件名为 MyNamespace.Form1.resources。

> **NOTE**
> 如果为 `EmbeddedResource` 项指定 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据，则为该资源文件生成的清单文件名将改为基于该元数据。

默认情况下，在新的 .NET 项目中，此属性设置为 `true`。如果设置为 `false`，并且没有为项目文件中的 `EmbeddedResource` 项指定 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据，则资源清单文件名将基于项目的根命名空间和 .resx 文件的相对文件路径。有关详细信息，请参阅资源清单文件的命名。

```
<PropertyGroup>
  <EmbeddedResourceUseDependentUponConvention>true</EmbeddedResourceUseDependentUponConvention>
</PropertyGroup>
```

EnablePreviewFeatures

`EnablePreviewFeatures` 属性定义项目是否依赖于使用 RequiresPreviewFeaturesAttribute 特性修饰的任何 API 或程序集。此特性用于表示 API 或程序集使用的功能被视为是你正使用的 SDK 版本的预览功能。不支持预览功能，并且可能会在将来的版本中删除这些功能。若要启用预览功能，请将属性设置为 `True`。

```
<PropertyGroup>
  <EnablePreviewFeatures>True</EnablePreviewFeatures>
</PropertyGroup>
```

当项目包含设置为 `True` 的属性时，以下程序集级别特性将添加到 AssemblyInfo.cs 文件中：

```
[assembly: RequiresPreviewFeatures]
```

当 `EnablePreviewFeatures` 未设置为 `True` 时，如果此特性存在于项目的依赖项中，分析器会发出警告。

要发运预览程序集的库作者应将此属性设置为 `True`。如果程序集需要随预览 API 和非预览 API 一起提供，请参阅下面的 GenerateRequiresPreviewFeaturesAttribute 部分。

GenerateRequiresPreviewFeaturesAttribute

`GenerateRequiresPreviewFeaturesAttribute` 属性与 EnablePreviewFeatures 属性密切相关。如果库使用预览功能，但不希望使用 RequiresPreviewFeaturesAttribute 特性标记整个程序集（需要任何使用者启用预览功能），请将此属性设置为 `False`。

```
<PropertyGroup>
    <EnablePreviewFeatures>True</EnablePreviewFeatures>
    <GenerateRequiresPreviewFeaturesAttribute>False</GenerateRequiresPreviewFeaturesAttribute>
</PropertyGroup>
```

> **IMPORTANT**
> 如果将 `GenerateRequiresPreviewFeaturesAttribute` 属性设置 `False`，则一定要使用 RequiresPreviewFeaturesAttribute 修饰依赖于预览功能的所有公共 API。

OptimizeImplicitlyTriggeredBuild

为了加快生成时间，Visual Studio 隐式触发的生成将跳过代码分析（包括可为 null 的分析）。例如，当运行测试时，Visual Studio 触发隐式生成。但是，仅当 `TreatWarningsAsErrors` 不是 `true` 时，才会优化隐式生成。如果将 `TreatWarningsAsErrors` 设置为 `true`，但仍希望优化已隐式触发的生成，则可以将 `OptimizeImplicitlyTriggeredBuild` 设置为 `True`。若要关闭隐式触发的生成的生成优化，请将 `OptimizeImplicitlyTriggeredBuild` 设置为 `False`。

```
<PropertyGroup>
    <OptimizeImplicitlyTriggeredBuild>True</OptimizeImplicitlyTriggeredBuild>
</PropertyGroup>
```

默认项包含属性

本节收录了以下 MSBuild 属性：

- DefaultExcludesInProjectFolder
- DefaultItemExcludes
- EnableDefaultCompileItems
- EnableDefaultEmbeddedResourceItems
- EnableDefaultItems
- EnableDefaultNoneItems

有关详细信息，请参阅默认的包括和排除。

DefaultItemExcludes

使用 `DefaultItemExcludes` 属性定义需从“包括”、“排除”和“删除”glob 中排除的文件和文件夹的 glob 模式。默认情况下从 glob 模式中排除 ./bin 和 ./obj 文件夹 。

```
<PropertyGroup>
  <DefaultItemExcludes>$(DefaultItemExcludes);**/*.myextension</DefaultItemExcludes>
</PropertyGroup>
```

DefaultExcludesInProjectFolder

使用 `DefaultExcludesInProjectFolder` 属性定义项目文件夹中需要从“包括”、“排除”和“删除”glob 中排除的文件和文件夹的 glob 模式。默认情况下，从 glob 模式中排除以句点 (`.`) 开头的文件夹，如 .git 和 .vs。

此属性与 `DefaultItemExcludes` 属性非常相似，不同之处在于它只涉及项目文件夹中的文件和文件夹。如果 glob 模式会无意中将项目文件夹外部的项与相对路径进行匹配，请使用 `DefaultExcludesInProjectFolder` 属性，而不是 `DefaultItemExcludes` 属性。

```
<PropertyGroup>
  <DefaultExcludesInProjectFolder>$(DefaultExcludesInProjectFolder);**/myprefix*/**
</DefaultExcludesInProjectFolder>
</PropertyGroup>
```

EnableDefaultItems

`EnableDefaultItems` 属性控制是否在项目中隐式包含编译项、嵌入的资源项和 `None` 项。默认值为 `true`。若要禁用所有隐式文件包含，请将 `EnableDefaultItems` 属性设置为 `false`。

```
<PropertyGroup>
  <EnableDefaultItems>false</EnableDefaultItems>
</PropertyGroup>
```

EnableDefaultCompileItems

`EnableDefaultCompileItems` 属性控制是否在项目中隐式包含编译项。默认值为 `true`。将 `EnableDefaultCompileItems` 属性设置为 `false` 以禁用 * .cs 和其他语言扩展文件的隐式包含。

```
<PropertyGroup>
  <EnableDefaultCompileItems>false</EnableDefaultCompileItems>
</PropertyGroup>
```

EnableDefaultEmbeddedResourceItems

`EnableDefaultEmbeddedResourceItems` 属性控制是否在项目中隐式包含嵌入的资源项。默认值为 `true`。将 `EnableDefaultEmbeddedResourceItems` 属性设置为 `false` 以禁用嵌入的资源文件的隐式包含。

```
<PropertyGroup>
  <EnableDefaultEmbeddedResourceItems>false</EnableDefaultEmbeddedResourceItems>
</PropertyGroup>
```

EnableDefaultNoneItems

`EnableDefaultNoneItems` 属性控制是否在项目中隐式包含 `None` 项(生成过程中未赋予角色的文件)。默认值为 `true`。将 `EnableDefaultNoneItems` 属性设置为 `false` 以禁用 `None` 项的隐式包含。

```
<PropertyGroup>
  <EnableDefaultNoneItems>false</EnableDefaultNoneItems>
</PropertyGroup>
```

代码分析属性

本节收录了以下 MSBuild 属性:

- AnalysisLevel
- AnalysisLevel<Category>
- AnalysisMode
- AnalysisMode<Category>
- CodeAnalysisTreatWarningsAsErrors
- EnableNETAnalyzers
- EnforceCodeStyleInBuild

AnalysisLevel

`AnalysisLevel` 属性使你可以指定一组代码分析器，以根据 .NET 版本进行运行。从 .NET 5 开始，每个 .NET 版本都有一组代码分析规则。在这组规则中，默认为该版本启用的规则将分析代码。

例如，如果升级到 .NET 6，但不希望更改默认的这组代码分析规则，请将 `AnalysisLevel` 设置为 `5`。

```
<PropertyGroup>
  <AnalysisLevel>preview</AnalysisLevel>
</PropertyGroup>
```

(可选)从 .NET 6 开始，可以为此属性指定复合值，该值还指定启用规则的积极程度。复合值采用形式 `<version>-<mode>`，其中 `<mode>` 值是 AnalysisMode 值之一。以下示例使用代码分析器预览版，并启用建议的规则集。

```
<PropertyGroup>
  <AnalysisLevel>preview-recommended</AnalysisLevel>
</PropertyGroup>
```

> **NOTE**
> 如果将 `AnalysisLevel` 设置为 `5-<mode>` 或 `5.0-<mode>`，然后安装 .NET 6 SDK 并重新编译项目，可能会看到意外的新生成警告。有关详细信息，请参阅 dotnet/roslyn-analyzers#5679。

默认值：

- 如果你的项目以 .NET 5 或更高版本为目标，或你添加了 AnalysisMode 属性，则默认值为 `latest`。
- 否则，此属性被省略，除非你将它明确添加到项目文件中。

下表显示可以指定的值。

`latest`	使用已发布的最新版代码分析器。这是默认值。
`latest-<mode>`	使用已发布的最新版代码分析器。`<mode>` 值确定启用哪些规则。
`preview`	使用最新的代码分析器（即使它们处于预览状态）。
`preview-<mode>`	使用最新的代码分析器（即使它们处于预览状态）。`<mode>` 值确定启用哪些规则。
`6.0`	即使有较新的规则可用，也会使用可用于 .NET 6 版本的规则集。
`6.0-<mode>`	即使有较新的规则可用，也会使用可用于 .NET 6 版本的规则集。`<mode>` 值确定启用哪些规则。
`6`	即使有较新的规则可用，也会使用可用于 .NET 6 版本的规则集。
`6-<mode>`	即使有较新的规则可用，也会使用可用于 .NET 6 版本的规则集。`<mode>` 值确定启用哪些规则。
`5.0`	即使有较新的规则可用，也会使用可用于 .NET 5 版本的规则集。
`5.0-<mode>`	即使有较新的规则可用，也会使用可用于 .NET 5 版本的规则集。`<mode>` 值确定启用哪些规则。
`5`	即使有较新的规则可用，也会使用可用于 .NET 5 版本的规则集。
`5-<mode>`	即使有较新的规则可用，也会使用可用于 .NET 5 版本的规则集。`<mode>` 值确定启用哪些规则。

NOTE

- 在 .NET 5 以及更早版本中，此属性仅影响代码质量 (CAXXXX) 规则。从 .NET 6 开始，如果将 EnforceCodeStyleInBuild 设置为 `true`，则此属性也会影响代码样式 (IDEXXXX) 规则。
- 如果为 `AnalysisLevel` 设置复合值，则无需指定 AnalysisMode。但是，如果这样做，`AnalysisLevel` 会优先于 `AnalysisMode`。
- 此属性对未引用项目 SDK 的项目（例如，引用 Microsoft.CodeAnalysis.NetAnalyzers NuGet 包的旧版 .NET Framework 项目）中的代码分析没有影响。

AnalysisLevel<Category>

.NET 6 中引入的此属性与 AnalysisLevel 相同，但仅适用于特定的代码分析规则类别。此属性使你能够为特定类别使用不同版本的代码分析器，或在其他规则类别的不同级别启用或禁用规则。如果为特定规则类别省略此属性，则其默认为 AnalysisLevel 值。可用值与 AnalysisLevel 相同。

```
<PropertyGroup>
  <AnalysisLevelSecurity>preview</AnalysisLevelSecurity>
</PropertyGroup>
```

```
<PropertyGroup>
  <AnalysisLevelSecurity>preview-recommended</AnalysisLevelSecurity>
</PropertyGroup>
```

下表列出了每个规则类别的属性名称。

`<AnalysisLevelDesign>`	设计规则
`<AnalysisLevelDocumentation>`	文档规则
`<AnalysisLevelGlobalization>`	全球化规则
`<AnalysisLevelInteroperability>`	可移植性和互操作性规则
`<AnalysisLevelMaintainability>`	可维护性规则
`<AnalysisLevelNaming>`	命名规则
`<AnalysisLevelPerformance>`	性能规则
`<AnalysisLevelSingleFile>`	单文件应用程序规则
`<AnalysisLevelReliability>`	可靠性规则
`<AnalysisLevelSecurity>`	安全规则
`<AnalysisLevelStyle>`	所有代码样式 (IDEXXXX) 规则
`<AnalysisLevelUsage>`	用法规则

AnalysisMode

从 .NET 5 开始，.NET SDK 附带了所有“CA”代码质量规则。默认情况下，只有一些规则作为生成警告启用。

`AnalysisMode` 属性允许自定义默认启用的一组规则。可以切换到更主动的(选择退出)分析模式，也可以切换到更保守的(选择加入)分析模式。例如，如果要作为生成警告默认启用所有规则，请将值设置为 `All` 或 `AllEnabledByDefault` 。

```
<PropertyGroup>
  <AnalysisMode>All</AnalysisMode>
</PropertyGroup>
```

下表显示了 .NET 5 和 .NET 6 中可用的选项值。它们按其启用的规则数的增加顺序列出。

.NET 5	.NET 6	
`AllDisabledByDefault`	`None`	默认情况下，禁用所有规则。可以选择选择加入各条规则，以启用它们。
`Default`	`Default`	默认模式，其中某些规则作为生成警告启用，某些规则作为 Visual Studio IDE 建议启用，其余规则被禁用。
	`Minimum`	比 `Default` 模式更主动的模式。强烈建议生成实施的某些建议作为生成警告启用。
	`Recommended`	比 `Minimum` 模式更主动的模式，其中启用了更多规则作为生成警告。
`AllEnabledByDefault`	`All`	所有规则默认作为生成警告启用。可以选择选择退出各条规则，以禁用它们。

NOTE

- 在 .NET 5 中，此属性仅影响代码质量 (CAXXXX) 规则。从 .NET 6 开始，如果将 EnforceCodeStyleInBuild 设置为 `true` ，则此属性也会影响代码样式 (IDEXXXX) 规则。
- 如果使用 AnalysisLevel 的复合值(如 `<AnalysisLevel>5-recommended</AnalysisLevel>`)，则可以完全省略此属性。但是，如果同时指定这两个属性，则 `AnalysisLevel` 会优先于 `AnalysisMode` 。
- 如果将 `AnalysisMode` 设置为 `AllEnabledByDefault` ，并将 `AnalysisLevel` 设置为 `5` 或 `5.0` ，然后安装 .NET 6 SDK 并重新编译项目，可能会看到意外的新生成警告。有关详细信息，请参阅 dotnet/roslyn-analyzers#5679。
- 此属性对未引用项目 SDK 的项目(例如，引用 Microsoft.CodeAnalysis.NetAnalyzers NuGet 包的旧版 .NET Framework 项目)中的代码分析没有影响。

AnalysisMode<Category>

.NET 6 中引入的此属性与 AnalysisMode 相同，但仅适用于特定的代码分析规则类别。此属性使你能够在其他规则类别的不同级别启用或禁用规则。如果为特定规则类别省略此属性，则其默认为 AnalysisMode 值。可用值与 AnalysisMode 相同。

```
<PropertyGroup>
  <AnalysisModeSecurity>All</AnalysisModeSecurity>
</PropertyGroup>
```

下表列出了每个规则类别的属性名称。

`<AnalysisModeDesign>`	设计规则

tttt	tttt
`<AnalysisModeDocumentation>`	文档规则
`<AnalysisModeGlobalization>`	全球化规则
`<AnalysisModeInteroperability>`	可移植性和互操作性规则
`<AnalysisModeMaintainability>`	可维护性规则
`<AnalysisModeNaming>`	命名规则
`<AnalysisModePerformance>`	性能规则
`<AnalysisModeSingleFile>`	单文件应用程序规则
`<AnalysisModeReliability>`	可靠性规则
`<AnalysisModeSecurity>`	安全规则
`<AnalysisModeStyle>`	所有代码样式 (IDEXXXX) 规则
`<AnalysisModeUsage>`	用法规则

CodeAnalysisTreatWarningsAsErrors

`CodeAnalysisTreatWarningsAsErrors` 属性可配置是否应将代码质量分析警告 (CAxxxx) 视为警告并中断生成。如果在生成项目时使用 `-warnaserror` 标志，则 .NET 代码质量分析警告也会被视为错误。如果不希望将代码质量分析警告视为错误，可以在项目文件中将 `CodeAnalysisTreatWarningsAsErrors` MSBuild 属性设置为 `false`。

```
<PropertyGroup>
  <CodeAnalysisTreatWarningsAsErrors>false</CodeAnalysisTreatWarningsAsErrors>
</PropertyGroup>
```

EnableNETAnalyzers

默认情况下，为面向 .NET 5 或更高版本的项目启用 .NET 代码质量分析。如果你是使用 .NET 5+ SDK 进行开发，可通过将 `EnableNETAnalyzers` 属性设置为 `true`，来为面向 .NET 早期版本的 SDK 样式项目启用 .NET 代码分析。若要禁用任何项目中的代码分析，可将此属性设置为 `false`。

```
<PropertyGroup>
  <EnableNETAnalyzers>true</EnableNETAnalyzers>
</PropertyGroup>
```

> **NOTE**
> 此属性专门用于 .NET 5 及更高版本 SDK 中的内置分析器。安装 NuGet 代码分析包时不应使用此属性。

EnforceCodeStyleInBuild

> **NOTE**
> 此功能当前为实验性功能，可能会在 .NET 5 和 .NET 6 版本之间发生更改。

对于所有 .NET 项目的版本，.NET 代码样式分析默认处于禁用状态。通过将 `EnforceCodeStyleInBuild` 属性设置为 `true`，可以为 .NET 项目启用代码样式分析。

```
<PropertyGroup>
  <EnforceCodeStyleInBuild>true</EnforceCodeStyleInBuild>
</PropertyGroup>
```

生成和报告违规时将执行配置为警告或错误的所有代码样式规则。

运行时配置属性

可以通过在应用的项目文件中指定 MSBuild 属性来配置某些运行时行为。有关配置运行时行为的其他方法的信息，请参阅运行时配置设置。

- ConcurrentGarbageCollection
- InvariantGlobalization
- PredefinedCulturesOnly
- RetainVMGarbageCollection
- ServerGarbageCollection
- ThreadPoolMaxThreads
- ThreadPoolMinThreads
- TieredCompilation
- TieredCompilationQuickJit
- TieredCompilationQuickJitForLoops

ConcurrentGarbageCollection

`ConcurrentGarbageCollection` 属性配置是否启用 后台(并发)垃圾回收。将值设置为 `false` 以禁用后台垃圾回收。有关详细信息，请参阅后台 GC。

```
<PropertyGroup>
  <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>
```

InvariantGlobalization

`InvariantGlobalization` 属性配置应用是否在全球化固定模式下运行，这意味着它无权访问特定于区域性的数据。将值设置为 `true` 以在全球化固定模式下运行。有关详细信息，请参阅固定模式。

```
<PropertyGroup>
  <InvariantGlobalization>true</InvariantGlobalization>
</PropertyGroup>
```

PredefinedCulturesOnly

在 .NET 6 及更高版本中，`PredefinedCulturesOnly` 属性配置应用在启用全球化固定模式时，是否可以创建除固定区域性之外的区域性。默认为 `true`。将值设置为 `false` 可在全球化固定模式下创建任何新区域性。

```
<PropertyGroup>
  <PredefinedCulturesOnly>false</PredefinedCulturesOnly>
</PropertyGroup>
```

有关详细信息，请参阅全球化固定模式下的区域性创建和大小写映射。

RetainVMGarbageCollection

`RetainVMGarbageCollection` 属性配置垃圾回收器，以将已删除的内存段放置在备用列表上供将来使用或释放它

们。将值设置为 `true` 会告知垃圾回收器将段放在备用列表上。有关详细信息，请参阅保留 VM。

```
<PropertyGroup>
  <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
</PropertyGroup>
```

ServerGarbageCollection

`ServerGarbageCollection` 属性配置应用程序是使用工作站垃圾回收还是服务器垃圾回收。将值设置为 `true` 以使用服务器垃圾回收。有关详细信息，请参阅工作站与服务器。

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

ThreadPoolMaxThreads

`ThreadPoolMaxThreads` 属性配置工作线程池的最大线程数。有关详细信息，请参阅最大线程数。

```
<PropertyGroup>
  <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>
</PropertyGroup>
```

ThreadPoolMinThreads

`ThreadPoolMinThreads` 属性配置工作线程池的最小线程数。有关详细信息，请参阅最小线程数。

```
<PropertyGroup>
  <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
</PropertyGroup>
```

TieredCompilation

`TieredCompilation` 属性配置实时 (JIT) 编译器是否使用分层编译。将值设置为 `false` 以禁用分层编译。有关详细信息，请参阅分层编译。

```
<PropertyGroup>
  <TieredCompilation>false</TieredCompilation>
</PropertyGroup>
```

TieredCompilationQuickJit

`TieredCompilationQuickJit` 属性配置 JIT 编译器是否使用快速 JIT。将值设置为 `false` 以禁用快速 JIT。有关详细信息，请参阅快速 JIT。

```
<PropertyGroup>
  <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
</PropertyGroup>
```

TieredCompilationQuickJitForLoops

`TieredCompilationQuickJitForLoops` 配置 JIT 编译器是否对包含循环的方法使用快速 JIT。将值设置为 `true` 以对包含循环的方法启用快速 JIT。有关详细信息，请参阅适用于循环的快速 JIT。

```
<PropertyGroup>
  <TieredCompilationQuickJitForLoops>true</TieredCompilationQuickJitForLoops>
</PropertyGroup>
```

引用属性

本节收录了以下 MSBuild 属性：

- AssetTargetFallback
- DisableImplicitFrameworkReferences
- 与还原相关的属性
- ValidateExecutableReferencesMatchSelfContained

AssetTargetFallback

使用 `AssetTargetFallback` 属性，可以为项目引用和 NuGet 包指定其他兼容的框架版本。例如，如果使用 `PackageReference` 指定包依赖项，但该包不包含与项目的 `TargetFramework` 兼容的资源，则可使用 `AssetTargetFallback` 属性。使用 `AssetTargetFallback` 中指定的每个目标框架重新检查引用包的兼容性。此属性替换已弃用的属性 `PackageTargetFallback`。

可以将 `AssetTargetFallback` 属性设置为一个或多个目标框架版本。

```
<PropertyGroup>
  <AssetTargetFallback>net461</AssetTargetFallback>
</PropertyGroup>
```

DisableImplicitFrameworkReferences

面向 .NET Core 3.0 及更高版本时，`DisableImplicitFrameworkReferences` 属性会控制隐式 `FrameworkReference` 项。面向 .NET Core 2.1 或 .NET Standard 2.0 及早期版本时，该属性会控制元包中包的隐式 PackageReference 项。（元包是一种基于框架的包，其中仅包含对其他包的依赖项。）在面向 .NET Framework 时，此属性还控制隐式引用，如 `System` 和 `System.Core`。

将此属性设置为 `true` 以禁用隐式 `FrameworkReference` 或 PackageReference 项。如果将此属性设置为 `true`，则可以仅添加对所需框架或包的显式引用。

```
<PropertyGroup>
  <DisableImplicitFrameworkReferences>true</DisableImplicitFrameworkReferences>
</PropertyGroup>
```

与还原相关的属性

还原被引用的包会安装它的所有直接依赖项，以及这些依赖项的全部依赖项。可以通过指定 `RestorePackagesPath` 和 `RestoreIgnoreFailedSources` 等属性来自定义包还原。若要详细了解这些属性和其他属性，请参阅还原目标。

```
<PropertyGroup>
  <RestoreIgnoreFailedSource>true</RestoreIgnoreFailedSource>
</PropertyGroup>
```

ValidateExecutableReferencesMatchSelfContained

`ValidateExecutableReferencesMatchSelfContained` 属性可用于禁用与可执行项目引用相关的错误。如果 .NET 检测到独立可执行文件项目引用依赖于框架的可执行项目，或相反，则会分别生成错误 NETSDK1150 和 NETSDK1151。若要避免在引用是有意而为之时出现这些错误，请将 `ValidateExecutableReferencesMatchSelfContained` 属性设置为 `false`。

```
<PropertyGroup>
  <ValidateExecutableReferencesMatchSelfContained>false</ValidateExecutableReferencesMatchSelfContained>
</PropertyGroup>
```

WindowsSdkPackageVersion

`WindowsSdkPackageVersion` 属性可用于替代 Windows SDK 目标包的版本。此属性是在 .NET 5 中引入的，并出于此目的替换了 `FrameworkReference` 项的使用。

```
<PropertyGroup>
  <WindowsSdkPackageVersion>10.0.19041.18</WindowsSdkPackageVersion>
</PropertyGroup>
```

> **NOTE**
> 不建议替代 Windows SDK 版本，因为 .NET 5+SDK 中包含 Windows SDK 目标包。若要引用最新的 Windows SDK 包，请更新 .NET SDK 的版本。此属性仅应在极少数情况下使用，例如使用预览包或需要替代 C#/WinRT 的版本。

与运行相关的属性

以下属性用于使用 `dotnet run` 命令启动应用：

- RunArguments
- RunWorkingDirectory

RunArguments

`RunArguments` 属性定义了在应用运行时向其传递的参数。

```
<PropertyGroup>
  <RunArguments>-mode dryrun</RunArguments>
</PropertyGroup>
```

> **TIP**
> 可以使用 `dotnet run` 的 `--` 选项来指定要传递到应用的其他参数。

RunWorkingDirectory

`RunWorkingDirectory` 属性定义要用于启动应用程序进程的工作目录。它可以是绝对路径，也可以是相对于项目目录的路径。如果未指定目录，`OutDir` 将用作工作目录。

```
<PropertyGroup>
  <RunWorkingDirectory>c:\temp</RunWorkingDirectory>
</PropertyGroup>
```

与托管相关的属性

本节收录了以下 MSBuild 属性：

- EnableComHosting
- EnableDynamicLoading

EnableComHosting

`EnableComHosting` 属性表示程序集提供了 COM 服务器。将 `EnableComHosting` 设置为 `true` 也表明 EnableDynamicLoading 为 `true`。

```
<PropertyGroup>
  <EnableComHosting>True</EnableComHosting>
</PropertyGroup>
```

有关详细信息，请参阅向 COM 公开 .NET 组件。

EnableDynamicLoading

`EnableDynamicLoading` 属性指示程序集是动态加载的组件。组件可以是 COM 库，也可以是在本机主机中使用或用作插件的非 COM 库。将此属性设置为 `true` 会产生以下结果：

- 生成 runtimeconfig.json 文件。
- RollForward 设置为 `LatestMinor`。
- 在本地复制 NuGet 引用。

```
<PropertyGroup>
  <EnableDynamicLoading>true</EnableDynamicLoading>
</PropertyGroup>
```

生成的文件属性

以下属性与生成的文件中的代码有关：

- DisableImplicitNamespaceImports
- ImplicitUsings

DisableImplicitNamespaceImports

`DisableImplicitNamespaceImports` 属性可用于在面向 .NET 6 或更高版本的 Visual Basic 项目中禁用隐式命名空间导入。隐式命名空间是 Visual Basic 项目中全局导入的默认命名空间。将此属性设置为 `true` 可禁用隐式命名空间导入。

```
<PropertyGroup>
  <DisableImplicitNamespaceImports>true</DisableImplicitNamespaceImports>
</PropertyGroup>
```

ImplicitUsings

`ImplicitUsings` 属性可用于在面向 .NET 6（或更高版本）和 C# 10.0（或更高版本）的 C# 项目中禁用隐式 `global using` 指令。启用该功能后，.NET SDK 会根据项目 SDK 的类型为一组默认命名空间添加 `global using` 指令。将此属性设置为 `true` 或 `enable` 可启用隐式 `global using` 指令。若要禁用隐式 `global using` 指令，请删除该属性或将其设置为 `false` 或 `disable`。

```
<PropertyGroup>
  <ImplicitUsings>enable</ImplicitUsings>
</PropertyGroup>
```

> **NOTE**
> 面向 .NET 6 或更高版本的 C# 新项目的模板将 `ImplicitUsings` 默认设置为 `enable`。

若要定义显式 `global using` 指令，请添加 Using 项。

项

MSBuild 项是生成系统的输入。根据项的类型（即元素名称）指定项。例如，`Compile` 和 `Reference` 是两个常见项类型。.NET SDK 提供了以下附加项类型：

- AssemblyMetadata
- InternalsVisibleTo

- PackageReference
- TrimmerRootAssembly
- Using

你可以在这些项上使用任何标准的项目属性，例如 `Include` 和 `Update`。使用 `Include` 添加新项，使用 `Update` 修改现有项。例如，`Update` 通常用于修改由 .NET SDK 隐式添加的项。

AssemblyMetadata

`AssemblyMetadata` 项指定键值对 AssemblyMetadataAttribute 程序集特性。`Include` 元数据将成为密钥，`Value` 元数据将成为值。

```
<ItemGroup>
  <AssemblyMetadata Include="Serviceable" Value="True" />
</ItemGroup>
```

InternalsVisibleTo

`InternalsVisibleTo` 项为指定的友元程序集生成 InternalsVisibleToAttribute 程序集特性。

```
<ItemGroup>
  <InternalsVisibleTo Include="MyProject.Tests" />
</ItemGroup>
```

如果友元程序集已签名，则可以指定可选的 `Key` 元数据来指定其完整公钥。如果未指定 `Key` 元数据，并且 `$(PublicKey)` 可用，则使用该密钥。否则，不会向该特性添加公钥。

PackageReference

`PackageReference` 项定义了对 NuGet 包的引用。

`Include` 属性指定包 ID。`Version` 特性指定版本或版本范围。若要了解如何指定最低版本、最高版本、范围或完全匹配，请参阅版本范围。

以下示例中的项目文件片段引用 System.Runtime 包。

```
<ItemGroup>
  <PackageReference Include="System.Runtime" Version="4.3.0" />
</ItemGroup>
```

你还可以使用元数据（例如 `PrivateAssets`）来控制依赖项资产。

```
<ItemGroup>
  <PackageReference Include="Contoso.Utility.UsefulStuff" Version="3.6.0">
    <PrivateAssets>all</PrivateAssets>
  </PackageReference>
</ItemGroup>
```

有关详细信息，请参阅项目文件中的包引用。

TrimmerRootAssembly

`TrimmerRootAssembly` 项允许通过修整排除程序集。修整是从打包的应用程序中删除运行时未使用部分的过程。在某些情况下，修整可能会错误地删除所需的引用。

以下 XML 通过修整排除 `System.Security` 程序集。

```
<ItemGroup>
  <TrimmerRootAssembly Include="System.Security" />
</ItemGroup>
```

使用

`Using` 项可在整个 C# 项目中全局添加命名空间，因此不必在源文件顶部为命名空间添加 `using` 指令。此项类似于可在 Visual Basic 项目中实现相同目的的 `Import` 项。从 .NET 6 开始，就可使用此属性。

```
<ItemGroup>
  <Using Include="My.Awesome.Namespace" />
</ItemGroup>
```

还可使用 `Using` 项来定义全局 `using <alias>` 和 `using static <type>` 指令。

```
<ItemGroup>
  <Using Include="My.Awesome.Namespace" Alias="Awesome" />
</ItemGroup>
```

例如：

- `<Using Include="Microsoft.AspNetCore.Http.Results" Alias="Results" />` 发出 `global using Results = global::Microsoft.AspNetCore.Http.Results;`
- `<Using Include="Microsoft.AspNetCore.Http.Results" Static="True" />` 发出 `global using static global::Microsoft.AspNetCore.Http.Results;`

若要详细了解别名 `using` 指令和 `using static <type>` 指令，请参阅 using 指令。

项元数据

除了标准的 MSBuild 项目属性之外，.net SDK 还提供以下项元数据标记：

- CopyToPublishDirectory
- LinkBase

CopyToPublishDirectory

MSBuild 项上的 `CopyToPublishDirectory` 元数据控制何时将项复制到发布目录。允许的值为 `PreserveNewest`（仅在项已更改时复制项）、`Always`（始终复制项）和 `Never`（从不复制项）。从性能角度来看，`PreserveNewest` 更为可取，因为它可实现增量生成。

```
<ItemGroup>
  <None Update="appsettings.Development.json" CopyToOutputDirectory="PreserveNewest"
CopyToPublishDirectory="PreserveNewest" />
</ItemGroup>
```

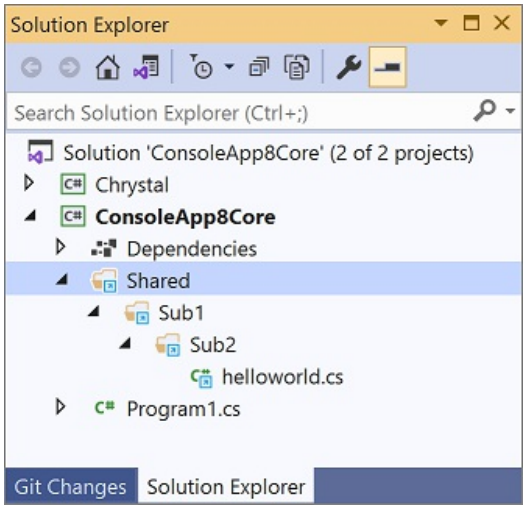
LinkBase

对于项目目录及其子目录之外的项，发布目标使用项的链接元数据来确定要将项复制到的位置。`Link` 还将确定项目树外的项在 Visual Studio 的“解决方案资源管理器”窗口中的显示方式。

如果没有为项目圆锥之外的项指定 `Link`，则默认为 `%(LinkBase)\%(RecursiveDir)%(Filename)%(Extension)`。通过 `LinkBase`，可以为项目圆锥之外的项指定一个合理的基础文件夹。基础文件夹下的文件夹层次结构通过 `RecursiveDir` 保留。如果未指定 `LinkBase`，则将从 `Link` 路径中省略它。

```
<ItemGroup>
  <Content Include="..\Extras\**\*.cs" LinkBase="Shared"/>
</ItemGroup>
```

下图显示通过上一个项 `Include` glob 包含的文件在解决方案资源管理器中的显示方式。



请参阅

- MSBuild 架构引用
- 通用 MSBuild 属性
- NuGet 包的 MSBuild 属性
- NuGet 还原的 MSBuild 属性
- 自定义生成

.NET 桌面 SDK 项目的 MSBuild 参考

2021/11/16 •

此页是有关用于通过 .NET 桌面 SDK 配置 Windows 窗体 (WinForms) 和 Windows Presentation Foundation (WPF) 项目的 MSBuild 属性和项的参考。

> **NOTE**
> 本文介绍了 .NET SDK 的 MSBuild 属性的子集，因为它与桌面应用相关。有关常用 .NET SDK 特定 MSBuild 属性的列表，请参见 .NET SDK 项目的 MSBuild 参考。有关通用 MSBuild 属性的列表，请参阅通用 MSBuild 属性。

启用 .NET 桌面 SDK

若要使用 WinForms 或 WPF，请配置你的项目文件。

.NET 5.0

在 WinForms 或 WPF 项目的项目文件中指定以下设置：

- 将 .NET SDK `Microsoft.NET.Sdk` 作为目标。有关详细信息，请参阅项目文件。
- 将 `TargetFramework` 设置为 `net5.0-windows`。
- 添加 UI 框架属性（或两者，如果必要）：
  - 将 `UseWPF` 设置为 `true` 以导入并使用 WPF。
  - 将 `UseWindowsForms` 设置为 `true` 以导入并使用 WinForms。
- （可选）将 `OutputType` 设置为 `WinExe`。这将生成应用，而不是库。若要生成库，请省略此属性。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>

    <UseWPF>true</UseWPF>
    <!-- and/or -->
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

.NET Core 3.1

在 WinForms 或 WPF 项目的项目文件中指定以下设置：

- 将 .NET SDK `Microsoft.NET.Sdk.WindowsDesktop` 作为目标。有关详细信息，请参阅项目文件。
- 将 `TargetFramework` 设置为 `netcoreapp3.1`。
- 添加 UI 框架属性（或两者，如果必要）：
  - 将 `UseWPF` 设置为 `true` 以导入并使用 WPF。
  - 将 `UseWindowsForms` 设置为 `true` 以导入并使用 WinForms。
- （可选）将 `OutputType` 设置为 `WinExe`。这将生成应用，而不是库。若要生成库，请省略此属性。

```
<Project Sdk="Microsoft.NET.Sdk.WindowsDesktop">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>

    <UseWPF>true</UseWPF>
    <!-- and/or -->
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>

</Project>
```

WPF 默认包含和排除的内容

SDK 项目定义一组规则，用于在项目中隐式包含或排除文件。这些规则还自动设置文件的生成操作。这与较旧的非 SDK .NET Framework 项目不同，后者没有默认的包含或排除规则。.NET Framework 项目需要你显式声明要将哪些文件包含在项目中。

.NET 项目文件包括一组标准规则，用于自动处理文件。WPF 项目添加了更多规则。

下表显示当 `UseWPF` 项目属性设置为 `true` 时在 .NET 桌面 SDK 中包含和排除哪些元素和 glob：

ApplicationDefinition	App.xaml 或 Application.xaml	不可用	不可用
Page	**/*.xaml	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc *ApplicationDefinition* 定义的任何 XAML	不可用
None	空值	不可用	**/*.xaml

下面是所有项目类型的默认包含和排除设置。有关详细信息，请参阅默认的包括和排除。

Compile	**/*.cs;**/*.vb（或其他语言扩展名）	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	不可用
EmbeddedResource	**/*.resx	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	不可用
None	**/*	**/*.user; **/*.*proj; **/*.sln; **/*.vssscc	**/*.cs; **/*.resx

与“重复”项相关的错误

如果已将文件显式添加到项目，或让 XAML glob 自动将文件包含在项目中，则可能会收到以下错误之一：

- 包含重复的“ApplicationDefinition”项。
- 包含重复的“Page”项。

这些错误是隐式包含 glob 与你的设置冲突的结果。要解决此问题，请将 `EnableDefaultApplicationDefinition` 或 `EnableDefaultPageItems` 设置为 `false`。将这些值设置为 `false` 会恢复到以前 SDK 的行为，其中你必须在项目中显式定义默认 glob，或者显式定义要包括在项目中的文件。

可以通过将 `EnableDefaultItems` 属性设置为 `false` 来完全禁用所有隐式包含。

WPF 设置

- UseWPF
- EnableDefaultApplicationDefinition
- EnableDefaultPageItems

有关非特定于 WPF 的项目设置的信息，请参阅 .NET SDK 项目的 MSBuild 参考。

UseWPF

`UseWPF` 属性控制是否包含对 WPF 库的引用。这还会更改 MSBuild 管道，以便正确处理 WPF 项目和相关文件。默认值为 `false`。将 `UseWPF` 属性设置为 `true` 以启用 WPF 支持。仅当启用了此属性时，才能将 Windows 平台作为目标。

```
<PropertyGroup>
  <UseWPF>true</UseWPF>
</PropertyGroup>
```

如果将此属性设置为 `true`，则 .NET 5+ 项目会自动导入 .NET 桌面 SDK。

.NET Core 3.1 项目需要将 .NET 桌面 SDK 显式设为目标才能使用此属性。

EnableDefaultApplicationDefinition

`EnableDefaultApplicationDefinition` 属性控制是否在项目中隐式包含 `ApplicationDefinition` 项。默认值为 `true`。若要禁用隐式文件包含，请将 `EnableDefaultApplicationDefinition` 属性设置为 `false`。

```
<PropertyGroup>
  <EnableDefaultApplicationDefinition>false</EnableDefaultApplicationDefinition>
</PropertyGroup>
```

此属性要求将 `EnableDefaultItems` 属性设置为 `true`，这是默认设置。

EnableDefaultPageItems

`EnableDefaultPageItems` 属性控制是否在项目中隐式包含 `Page` 项(即 *.xaml* 文件)。默认值为 `true`。若要禁用隐式文件包含，请将 `EnableDefaultPageItems` 属性设置为 `false`。

```
<PropertyGroup>
  <EnableDefaultPageItems>false</EnableDefaultPageItems>
</PropertyGroup>
```

此属性要求将 `EnableDefaultItems` 属性设置为 `true`，这是默认设置。

Windows 窗体设置

- ApplicationDefaultFont
- ApplicationHighDpiMode
- ApplicationUseCompatibleTextRendering
- ApplicationVisualStyles
- UseWindowsForms

有关非特定于 WinForms 的项目属性的信息，请参阅 .NET SDK 项目的 MSBuild 参考。

ApplicationDefaultFont

`ApplicationDefaultFont` 属性指定要在应用程序范围内应用的自定义字体信息。它控制源生成的 `ApplicationConfiguration.Initialize()` API 是否发出对 Application.SetDefaultFont(Font) 方法的调用。默认值为空字符串，这意味着应用程序默认字体来源于 Control.DefaultFont 属性。非空值必须符合与 `FontConverter.ConvertTo` 方法的输出等效的格式，即：`name, size[units[, style=style1[, style2, ...]]]`。

```
<PropertyGroup>
  <ApplicationDefaultFont>Calibri, 11pt, style=regular</ApplicationDefaultFont>
</PropertyGroup>
```

.NET 6 及更高版本和 Visual Studio 2022 及更高版本支持此属性。

ApplicationHighDpiMode

`ApplicationHighDpiMode` 属性为高 DPI 模式指定应用程序范围内的默认值。它控制由源生成的 `ApplicationConfiguration.Initialize()` API 发出的 Application.SetHighDpiMode(HighDpiMode) 方法的参数。默认值是 `SystemAware`。

```
<PropertyGroup>
  <ApplicationHighDpiMode>PerMonitorV2</ApplicationHighDpiMode>
</PropertyGroup>
```

`ApplicationHighDpiMode` 可设置为 HighDpiMode 枚举值之一：

“[”	[[
`DpiUnaware`	应用程序窗口不会随着 DPI 更改而缩放，始终假定缩放比例为 100%。
`DpiUnawareGdiScaled`	类似于 `DpiUnaware`，但提高了基于 GDI/GDI+ 的内容的质量。
`PerMonitor`	此窗口会在创建 DPI 时对其进行检查，并在 DPI 更改时调整缩放比例。
`PerMonitorV2`	类似于 `PerMonitor`，但启用了子窗口 DPI 更改通知、comctl32 控件的改进缩放和对话框缩放。
`SystemAware`	如果未指定，则为默认值。 此窗口会查询一次主监视器的 DPI，并将其用于所有监视器上的应用程序。

.NET 6 及更高版本支持此属性。

ApplicationUseCompatibleTextRendering

`ApplicationUseCompatibleTextRendering` 属性为某些控件上定义的 `UseCompatibleTextRendering` 属性指定应用程序范围内的默认值。它控制由源生成的 `ApplicationConfiguration.Initialize()` API 发出的 Application.SetCompatibleTextRenderingDefault(Boolean) 方法的参数。默认值是 `false`。

```
<PropertyGroup>
  <ApplicationUseCompatibleTextRendering>true</ApplicationUseCompatibleTextRendering>
</PropertyGroup>
```

.NET 6 及更高版本支持此属性。

ApplicationVisualStyles

`ApplicationVisualStyles` 属性指定应用程序范围内的默认值来启用视觉样式。它控制源生成的 `ApplicationConfiguration.Initialize()` API 是否发出对 Application.EnableVisualStyles() 的调用。默认值是 `true`。

```
<PropertyGroup>
  <ApplicationVisualStyles>true</ApplicationVisualStyles>
</PropertyGroup>
```

.NET 6 及更高版本支持此属性。

UseWindowsForms

`UseWindowsForms` 属性控制应用程序是否构建为以 Windows 窗体作为目标。此属性会更改 MSBuild 管道，以便正确处理 Windows 窗体项目和相关文件。默认值为 `false`。将 `UseWindowsForms` 属性设置为 `true` 可启用 Windows 窗体支持。仅当启用了此设置时，才能将 Windows 平台作为目标。

```
<PropertyGroup>
  <UseWindowsForms>true</UseWindowsForms>
</PropertyGroup>
```

如果将此属性设置为 `true`，则 .NET 5+ 项目会自动导入 .NET 桌面 SDK。

.NET Core 3.1 项目需要将 .NET 桌面 SDK 显式设为目标才能使用此属性。

共享设置

- DisableWinExeOutputInference

DisableWinExeOutputInference

适用于 .NET 5 SDK 及更高版本。

当应用的 `OutputType` 属性设置为 `Exe` 值时，如果该应用未从控制台运行，则将创建一个控制台窗口。这通常不是所需的 Windows 桌面应用行为。如果使用 `WinExe` 值，则不会创建控制台窗口。从 .NET 5 SDK 开始，`Exe` 值会自动转换为 `WinExe`。

`DisableWinExeOutputInference` 属性会还原将 `Exe` 视为 `WinExe` 的行为。将此值设置为 `true` 可还原 `OutputType` 属性值 `Exe` 的行为。默认值为 `false`。

```
<PropertyGroup>
  <DisableWinExeOutputInference>true</DisableWinExeOutputInference>
</PropertyGroup>
```

请参阅

- .NET 项目 SDK
- .NET SDK 项目的 MSBuild 引用
- MSBuild 架构引用
- 通用 MSBuild 属性
- 自定义生成

SDK 样式项目中的目标框架

2021/11/16 •

以应用或库中的框架为目标时，需要指定想要向应用或库提供的 API 集。使用目标框架名字对象 (TFM) 在项目文件中指定目标框架。

应用或库可以使用 .NET Standard 版本作为目标。.NET Standard 版本表示所有 .NET 实现中的标准化 API 集。例如，库可以使用 .NET Standard 1.6 作为目标，并获得对可使用相同基本代码跨 .NET Core 和 .NET Framework 工作的 API 的访问权限。

应用或库还能以一个特定 .NET 实现为目标，获得特定于实现的 API 的访问权限。例如，面向 Xamarin.iOS 的应用（如 `Xamarin.iOS10`）有权访问 Xamarin 提供的适用于 iOS 10 的 iOS API 包装器；面向通用 Windows 平台 (UWP) 的应用（如 `uap10.0`）有权访问为运行 Windows 10 的设备编译的 API。

对于某些目标框架（例如 .NET Framework），API 由框架在系统上安装的程序集定义，并且可能包括应用程序框架 API（例如 ASP.NET）。

对于基于包的目标框架（例如 .NET 5、.NET Core 和 .NET Standard），API 由应用或库中包含的 NuGet 包定义。

最新版本

下表定义了最常见的目标框架、如何引用这些框架，以及它们实现的 .NET Standard 版本。这些目标框架版本是最新的稳定版本。预览版不会显示。目标框架名字对象 (TFM) 是一个标准化令牌格式，用于指定 .NET 应用或库的目标框架。

[illegible]	[illegible]	[illegible] (TFM)	[illegible] .NET STANDARD [illegible]
.NET 5	5.0	net5.0	空值
.NET Standard	2.1	netstandard2.1	空值
.NET Core	3.1	netcoreapp3.1	2.1
.NET Framework	4.8	net48	2.0

支持的目标框架

目标框架通常由 TFM 引用。下表显示 .NET SDK 和 NuGet 客户端支持的目标框架。等效项显示在括号内。例如，`win81` 对于 `netcore451` 来说等效于 TFM。

[illegible] FRAMEWORK	TFM
.NET 5 及更高版本（和 .NET Core）	netcoreapp1.0 netcoreapp1.1 netcoreapp2.0 netcoreapp2.1 netcoreapp2.2 netcoreapp3.0 netcoreapp3.1 net5.0 *net6.0*

FRAMEWORK	TFM
.NET Standard	netstandard1.0 netstandard1.1 netstandard1.2 netstandard1.3 netstandard1.4 netstandard1.5 netstandard1.6 netstandard2.0 netstandard2.1
.NET Framework	net11 net20 net35 net40 net403 net45 net451 net452 net46 net461 net462 net47 net471 net472 net48
Windows 应用商店	netcore [netcore45] netcore45 [win] [win8] netcore451 [win81]
.NET Micro Framework	netmf
Silverlight	sl4 sl5
Windows Phone	wp [wp7] wp7 wp75 wp8 wp81 wpa81
通用 Windows 平台	uap [uap10.0] uap10.0 [win10] [netcore50]

* .NET 5 及更高版本的 TFM 包含一些特定于操作系统的变体。有关详细信息，请参阅下一节：.NET 5 及更高版本特定于 OS 的 TFM。

.NET 5 及更高版本特定于 OS 的 TFM

`net5.0` 和 `net6.0` TFM 包括可在不同平台中使用的技术。指定特定于 OS 的 TFM 使特定于操作系统的 API 可供你的应用（例如 Windows 窗体或 iOS 绑定）使用。特定于 OS 的 TFM 还会继承其基础 TFM（例如 `net5.0` TFM）可用的每个 API。

.NET 5 引入了 `net5.0-windows` 特定于 OS 的 TFM，其中包括适用于 WinForms、WPF 和 UWP API 的特定于 Windows 的绑定。.NET 6 引入了更多特定于 OS 的 TFM。

下表说明了 .NET 5 及更高版本 TFM 的兼容性。

TFM	描述
net5.0	net1..4（带有 NU1701 警告） netcoreapp1..3.1（引用 WinForms 或 WPF 时出现警告） netstandard1..2.1
net5.0-windows	netcoreapp1..3.1（以及从 `net5.0` 继承的所有其他内容）
net6.0	（后续版本的 `net5.0`）
net6.0-android	`xamarin.android`（以及从 `net6.0` 继承的所有其他内容）
net6.0-ios	`xamarin.ios`（以及从 `net6.0` 继承的所有其他内容）
net6.0-macos	`xamarin.mac`（以及从 `net6.0` 继承的所有其他内容）
net6.0-maccatalyst	`xamarin.ios`（以及从 `net6.0` 继承的所有其他内容）
net6.0-tvos	`xamarin.tvos`（以及从 `net6.0` 继承的所有其他内容）
net6.0-windows	（后续版本的 `net5.0-windows`）

若要使应用可跨不同平台移植，但仍有权访问特定于 OS 的 API，你可以定位多个特定于 OS 的 TFM，并使用 `#if` 预处理器指令围绕特定于 OS 的 API 调用增加平台防护。

建议的目标

使用以下准则确定在应用中使用哪种 TFM：

- 可移植到多个平台的应用应面向基础 TFM，例如 `net5.0`。这包括大多数库，但也包含 ASP.NET Core 和实体框架。
- 特定于平台的库应面向特定于平台的风格。例如，WinForms 和 WPF 项目应面向 `net5.0-windows` 或 `net6.0-windows`。
- 跨平台应用模型（Xamarin Forms、ASP.NET Core）和网桥包 (Xamarin Essentials) 应至少面向基础 TFM（例如 `net6.0`），但也可以面向其他特定于平台的风格来支持更多的 API 或功能。

TFM 中的 OS 版本

你还可以在 OS 特定的 TFM 的末尾指定可选的 OS 版本，例如，`net6.0-ios15.0`。版本指示应用或库可用的 API。它不控制应用或库在运行时支持的 OS 版本。它用于选择项目编译的引用程序集，并用于从 NuGet 包中选择资产。将此版本视为“平台版本”或“OS API 版本”，可以与运行时 OS 版本进行区分。

当特定于 OS 的 TFM 不显式指定平台版本时，它具有可从基础 TFM 和平台名称推断的隐含值。例如，.NET 6 中 iOS 的默认平台值为 `15.0`，这意味着 `net6.0-ios` 是规范 `net6.0-ios15.0` TFM 的简写形式。较新的基础 TFM 的隐含平台版本可能更高，例如，将来的 `net7.0-ios` TFM 可以映射到 `net7.0-ios16.0`。简写形式仅用于项目文件，在传递给其他工具（如 NuGet）之前，.NET SDK 的 MSBuild 目标将其扩展为规范格式。

.NET SDK 设计为能够支持单个平台的新发布的 API，而无需使用新版本的基础 TFM。这样，你无需等待 .NET 的主要版本就能访问特定于平台的功能。可以通过在 TFM 中增加平台版本来访问这些新发布的 API。例如，如果 iOS 平台在 .NET 6.0.x SDK 更新中添加了 iOS 15.1 API，你可以使用 TFM `net6.0-ios15.1` 访问它们。

支持旧版 OS

虽然特定于平台的应用或库是针对特定版本的 OS 中的 API 编译的，但你可以通过将 `SupportedOSPlatformVersion` 属性添加到项目文件，使其与早期版本的 OS 兼容。`SupportedOSPlatformVersion` 属性指示运行应用或库所需的最低 OS 版本。如果不在项目中显式指定此最低运行时 OS 版本，则默认为 TFM 中

的平台版本。

若要使应用在较旧的 OS 版本上正常运行，它不能调用该 OS 版本上不存在的 API。但是，可以增加对较新 API 的调用的防护，以便只有在支持这些 API 的 OS 版本上运行时，才能调用它们。使用此模式可以设计应用或库，以支持在较旧的 OS 版本上运行，同时在较新的 OS 版本上运行时利用较新的 OS 功能。

`SupportedOSPlatformVersion` 值（无论是显式还是默认）由平台兼容性分析器使用，用于检测并警告对较新 API 的无防御调用。它作为 UnsupportedOSPlatformAttribute 程序集属性记录到项目的编译程序集中，使平台兼容性分析器可以从具有较低值 `SupportedOSPlatformVersion` 的项目中检测对该程序集 API 的无防御调用。在一些平台上，`SupportedOSPlatformVersion` 值会影响特定于平台的应用打包和生成过程，这些平台的文档对此有所介绍。

下面是一个项目文件的示例摘录，该文件使用 `TargetFramework` 和 `SupportedOSPlatformVersion` MSBuild 属性指定应用或库有权访问 iOS 15.0 API，但支持在 iOS 13.0 及更高版本上运行：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0-ios15.0</TargetFramework>
    <SupportedOSPlatformVersion>13.0</SupportedOSPlatformVersion>
  </PropertyGroup>

  ...

</Project>
```

如何指定目标框架

在项目文件中指定目标框架。指定单个目标框架时，使用 TargetFramework 元素。以下控制台应用项目文件演示了如何面向 .NET 5：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

指定多个目标框架时，可有条件地为每个目标框架引用程序集。在代码中，可使用具有 -if-then-else 逻辑的预处理器符号，有条件地针对这些程序集进行编译。

以下库项目面向 .NET Standard (`netstandard1.4`) 和 .NET Framework（`net40` 和 `net45`）的 API。将复数形式的 TargetFrameworks 元素与多个目标框架一起使用。为两个 .NET Framework TFM 编译库时，`Condition` 属性包括特定于实现的包：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard1.4;net40;net45</TargetFrameworks>
  </PropertyGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.0 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net40' ">
    <Reference Include="System.Net" />
  </ItemGroup>

  <!-- Conditionally obtain references for the .NET Framework 4.5 target -->
  <ItemGroup Condition=" '$(TargetFramework)' == 'net45' ">
    <Reference Include="System.Net.Http" />
    <Reference Include="System.Threading.Tasks" />
  </ItemGroup>

</Project>
```

在库或应用中，使用预处理器指令编写条件代码，针对每个目标框架进行编译：

```
public class MyClass
{
    static void Main()
    {
#if NET40
        Console.WriteLine("Target framework: .NET Framework 4.0");
#elif NET45
        Console.WriteLine("Target framework: .NET Framework 4.5");
#else
        Console.WriteLine("Target framework: .NET Standard 1.4");
#endif
    }
}
```

使用 SDK 样式项目时，生成系统可识别预处理器符号，这些符号表示支持的目标框架版本表中所示的目标框架。使用表示 .NET Standard、.NET Core 或 .NET 5 TFM 的符号时，请用下划线替换点和连字符，并将小写字母更改为大写字母（例如，`netstandard1.4` 的符号为 `NETSTANDARD1_4`）。

.NET 目标框架的预处理器符号的完整列表如下：

□□□□	□□	.NET 5 □□□□ SDK □□□□□□□
.NET Framework	`NETFRAMEWORK`, `NET48`, `NET472`, `NET471`, `NET47`, `NET462`, `NET461`, `NET46`, `NET452`, `NET451`, `NET45`, `NET40`, `NET35`, `NET20`	`NET48_OR_GREATER`, `NET472_OR_GREATER`, `NET471_OR_GREATER`, `NET47_OR_GREATER`, `NET462_OR_GREATER`, `NET461_OR_GREATER`, `NET46_OR_GREATER`, `NET452_OR_GREATER`, `NET451_OR_GREATER`, `NET45_OR_GREATER`, `NET40_OR_GREATER`, `NET35_OR_GREATER`, `NET20_OR_GREATER`

[illegible]	[illegible]	.NET 5 [illegible] SDK [illegible]
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`, `NETSTANDARD1_1`, `NETSTANDARD1_0`	`NETSTANDARD2_1_OR_GREATER`, `NETSTANDARD2_0_OR_GREATER`, `NETSTANDARD1_6_OR_GREATER`, `NETSTANDARD1_5_OR_GREATER`, `NETSTANDARD1_4_OR_GREATER`, `NETSTANDARD1_3_OR_GREATER`, `NETSTANDARD1_2_OR_GREATER`, `NETSTANDARD1_1_OR_GREATER`, `NETSTANDARD1_0_OR_GREATER`
.NET 5 及更高版本(和 .NET Core)	`NET`, `NET6_0`, `NET6_0_ANDROID`, `NET6_0_IOS`, `NET6_0_MACOS`, `NET6_0_MACCATALYST`, `NET6_0_TVOS`, `NET6_0_WINDOWS`, `NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`	`NET6_0_OR_GREATER`, `NET6_0_ANDROID_OR_GREATER`, `NET6_0_IOS_OR_GREATER`, `NET6_0_MACOS_OR_GREATER`, `NET6_0_MACCATALYST_OR_GREATER`, `NET6_0_TVOS_OR_GREATER`, `NET6_0_WINDOWS_OR_GREATER`, `NET5_0_OR_GREATER`, `NETCOREAPP_OR_GREATER`, `NETCOREAPP3_1_OR_GREATER`, `NETCOREAPP3_0_OR_GREATER`, `NETCOREAPP2_2_OR_GREATER`, `NETCOREAPP2_1_OR_GREATER`, `NETCOREAPP2_0_OR_GREATER`, `NETCOREAPP1_1_OR_GREATER`, `NETCOREAPP1_0_OR_GREATER`

NOTE

- 无论目标版本是什么，都将定义无版本符号。
- 仅针对目标版本定义特定于版本的符号。
- 为目标版本和所有早期版本定义 `<framework>_OR_GREATER` 符号。例如，如果针对 .NET Framework 2.0，则会定义以下符号：`NET_2_0`、`NET_2_0_OR_GREATER`、`NET_1_1_OR_GREATER` 和 `NET_1_0_OR_GREATER`。

已弃用的目标框架

以下目标框架已弃用。面向这些目标框架的包应迁移到指定的替代框架。

[illegible] TFM	REPLACEMENT
aspnet50 aspnetcore50 dnxcore50 dnx dnx45 dnx451 dnx452	netcoreapp

TFM	REPLACEMENT
dotnet dotnet50 dotnet51 dotnet52 dotnet53 dotnet54 dotnet55 dotnet56	netstandard
netcore50	uap10.0
win	netcore45
win8	netcore45
win81	netcore451
win10	uap10.0
winrt	netcore45

另请参阅

- .NET 5 中的目标框架名称
- 在桌面应用中调用 Windows 运行时 API
- 使用跨平台工具开发库
- .NET Standard
- .NET Core 版本控制
- dotnet/standard GitHub 存储库
- NuGet 工具 GitHub 存储库
- .NET 中的框架配置文件
- 平台兼容性分析器

管理 .NET 应用程序中的包依赖项

2021/11/16 •

本文介绍如何通过编辑项目文件或使用 CLI 来添加和删除包依赖项。

<PackageReference> 元素

`<PackageReference>` 项目文件元素具有以下结构：

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" />
```

`Include` 特性指定要添加到项目的包的 ID。`Version` 特性指定要获取的版本。版本根据 NuGet 版本规则进行指定。

> **NOTE**
> 如果不熟悉“项目-文件”语法，可参阅 MSBuild 项目参考文档了解详细信息。

使用条件来添加仅在特定目标中可用的依赖项，如以下示例所示：

```
<PackageReference Include="PACKAGE_ID" Version="PACKAGE_VERSION" Condition="'$(TargetFramework)' ==
'netcoreapp2.1'" />
```

上述示例中的依赖项只有在对给定目标生成时才有效。条件中的 `$(TargetFramework)` 是将在项目中设置的 MSBuild 属性。对于大多数常见的 .NET 应用程序，无需这样做。

通过编辑项目文件添加依赖项

若要添加依赖项，请在 `<ItemGroup>` 元素内添加 `<PackageReference>` 元素。可以添加到现有 `<ItemGroup>`，也可以新建一个元素。下面的示例使用 `dotnet new console` 创建的默认控制台应用程序项目：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.EntityFrameworkCore" Version="3.1.2" />
  </ItemGroup>

</Project>
```

使用 CLI 添加依赖项

若要添加依赖项，请运行 dotnet add package 命令，如以下示例中所示：

```
dotnet add package Microsoft.EntityFrameworkCore
```

通过编辑项目文件删除依赖项

若要删除依赖项，请从项目文件中删除其 `<PackageReference>` 元素。

使用 CLI 删除依赖项

若要删除依赖项，请运行 dotnet remove package 命令，如以下示例中所示：

```
dotnet remove package Microsoft.EntityFrameworkCore
```

请参阅

- 项目文件中的包引用
- dotnet list package 命令

如何管理 .NET 工具

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

.NET 工具是一种特殊的 NuGet 包，其中包含控制台应用程序。可以通过以下方式在计算机上安装该工具：

- 作为全局工具。

  工具二进制文件安装在添加到 PATH 环境变量的默认目录中。无需指定工具位置即可从计算机上的任何目录调用该工具。工具的一个版本用于计算机上的所有目录。

- 作为自定义位置中的全局工具（也称为工具路径工具）。

  工具二进制文件安装在你指定的位置中。可以从安装目录调用该工具，也可以通过提供具有命令名称的目录或将目录添加到 PATH 环境变量来调用该工具。工具的一个版本用于计算机上的所有目录。

- 作为本地工具（适用于 .NET Core SDK 3.0 及更高版本）。

  工具二进制文件安装在默认目录中。可以从安装目录或其任何子目录调用该工具。不同目录可以使用同一工具的不同版本。

  .NET CLI 使用清单文件跟踪哪些工具作为本地工具安装到目录。将清单文件保存到源代码存储库的根目录中后，参与者可以克隆存储库并调用用于安装清单文件中列出的所有工具的单个 .NET CLI 命令。

> **IMPORTANT**
> .NET 工具在完全信任环境中运行。除非你信任工具作者，否则请勿安装 .NET 工具。

查找工具

以下是查找工具的一些方法：

- 使用 dotnet tool search 命令查找发布到 NuGet.org 的工具。
- 使用“.NET 工具”包类型筛选器搜索 NuGet 网站。有关详细信息，请参阅查找和选择包。
- 在 dotnet/aspnetcore GitHub 存储库的工具目录中查看 ASP.NET Core 团队创建的工具的源代码。
- 在 .NET 诊断工具中了解诊断工具。

查看作者和统计信息

由于 .NET 工具在完全信任环境中运行，并且全局工具已添加到 PATH 环境变量，因此它们的功能非常强大。请勿下载不信任的人提供的工具。

如果该工具在 NuGet 中托管，可以通过搜索该工具来查看作者和统计信息。

安装全局工具

若要将工具作为全局工具安装，请使用 dotnet tool install 的 `-g` 或 `--global` 选项，如以下示例中所示：

```
dotnet tool install -g dotnetsay
```

输出显示用于调用该工具和已安装的版本的命令，类似于以下示例：

```
You can invoke the tool using the following command: dotnetsay
Tool 'dotnetsay' (version '2.1.4') was successfully installed.
```

工具二进制文件的默认位置取决于操作系统：

(OS)	此
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

首次运行 SDK 时，会将此位置添加到用户的路径，因此，无需指定工具位置即可从任何目录调用全局工具。

工具访问特定于用户，而不针对计算机全局。全局工具仅适用于安装了该工具的用户。

安装自定义位置中的全局工具

若要将工具作为自定义位置中的全局工具安装，请使用 dotnet tool install 的 `--tool-path` 选项，如以下示例中所示。

在 Windows 上：

```
dotnet tool install dotnetsay --tool-path c:\dotnet-tools
```

在 Linux 或 macOS 上：

```
dotnet tool install dotnetsay --tool-path ~/bin
```

.NET SDK 不将此位置自动添加至 PATH 环境变量。若要调用工具路径工具，必须确保可使用以下方法之一来调用命令：

- 将安装目录添加到 PATH 环境变量。
- 调用该工具时，指定该工具的完整路径。
- 从安装目录调用该工具。

安装本地工具

适用于 .NET Core 3.0 SDK 及更高版本。

若要安装仅用于本地访问的工具(对于当前目录和子目录)，必须将其添加到工具清单文件。若要创建工具清单文件，请运行 `dotnet new tool-manifest` 命令：

```
dotnet new tool-manifest
```

此命令在“.config” 目录下创建一个名为“dotnet-tools.json” 的清单文件。若要将本地工具添加到清单文件，请使用 dotnet tool install 命令并省略 `--global` 和 `--tool-path` 选项，如以下示例中所示：

```
dotnet tool install dotnetsay
```

命令输出显示新安装的工具所在的清单文件，类似于以下示例：

```
You can invoke the tool from this directory using the following command:
dotnet tool run dotnetsay
Tool 'dotnetsay' (version '2.1.4') was successfully installed.
Entry is added to the manifest file /home/name/botsay/.config/dotnet-tools.json.
```

以下示例显示安装了两个本地工具的清单文件：

```
{
  "version": 1,
  "isRoot": true,
  "tools": {
    "botsay": {
      "version": "1.0.0",
      "commands": [
        "botsay"
      ]
    },
    "dotnetsay": {
      "version": "2.1.3",
      "commands": [
        "dotnetsay"
      ]
    }
  }
}
```

通常将本地工具添加到存储库的根目录。将清单文件签入到存储库后，从存储库中签出代码的开发人员会获得最新的清单文件。若要安装清单文件中列出的所有工具，请运行 `dotnet tool restore` 命令：

```
dotnet tool restore
```

输出表明还原了哪些工具：

```
Tool 'botsay' (version '1.0.0') was restored. Available commands: botsay
Tool 'dotnetsay' (version '2.1.3') was restored. Available commands: dotnetsay
Restore was successful.
```

安装特定工具版本

若要安装工具的预发布版本或特定版本，请使用 `--version` 选项指定版本号，如以下示例中所示：

```
dotnet tool install dotnetsay --version 2.1.3
```

若要安装工具的预发布版本而不指定确切的版本号，请使用 `--version` 选项并提供通配符，如以下示例所示：

```
dotnet tool install --global dotnetsay --version "*-rc*"
```

使用工具

用于调用工具的命令可能不同于安装的包的名称。若要显示计算机上目前安装的所有工具，请使用 dotnet tool list 命令：

```
dotnet tool list
```

输出显示每个工具的版本和命令，类似于以下示例：

```
Package Id      Version      Commands       Manifest
-------------------------------------------------------------------------------------
botsay          1.0.0        botsay         /home/name/repository/.config/dotnet-tools.json
dotnetsay       2.1.3        dotnetsay      /home/name/repository/.config/dotnet-tools.json
```

如本示例中所示，列表显示了本地工具。若要查看全局工具，请使用 `--global` 选项，若要查看工具路径工具，请使用 `--tool-path` 选项。

调用全局工具

对于全局工具，请单独使用工具命令。例如，如果命令为 `dotnetsay` 或 `dotnet-doc`，则可以使用以下命令调用该工具：

```
dotnetsay
dotnet-doc
```

如果命令以前缀 `dotnet-` 开头，则调用该工具的另一种方法是使用 `dotnet` 命令并省略工具命令前缀。例如，如果命令为 `dotnet-doc`，则可以使用以下命令调用该工具：

```
dotnet doc
```

但是，在以下情况下，不能使用 `dotnet` 命令来调用全局工具：

- 全局工具和本地工具具有以 `dotnet-` 为前缀的相同命令。
- 你希望从本地工具范围内的目录调用全局工具。

在这种情况下，`dotnet doc` 和 `dotnet dotnet-doc` 调用本地工具。若要调用全局工具，请单独使用命令：

```
dotnet-doc
```

调用工具路径工具

若要调用使用 `tool-path` 选项安装的全局工具，请确保该命令可用，如本文前面所述。

调用本地工具

若要调用本地工具，必须从安装目录使用 `dotnet` 命令。可以使用长格式 (`dotnet tool run <COMMAND_NAME>`) 或短格式 (`dotnet <COMMAND_NAME>`)，如以下示例中所示：

```
dotnet tool run dotnetsay
dotnet dotnetsay
```

如果命令以 `dotnet-` 为前缀，则可以在调用该工具时包括或省略前缀。例如，如果命令为 `dotnet-doc`，则可以使用以下任何示例调用本地工具：

```
dotnet tool run dotnet-doc
dotnet dotnet-doc
dotnet doc
```

更新工具

更新工具涉及卸载该工具并重新安装它的最新稳定版。若要更新工具，请使用具有用于安装该工具的相同选项

的 dotnet tool update 命令：

```
dotnet tool update --global <packagename>
dotnet tool update --tool-path <packagename>
dotnet tool update <packagename>
```

对于本地工具，SDK 通过在当前目录和父目录中查找来查找包含包 ID 的第一个清单文件。如果任何清单文件中都没有此类包 ID，SDK 会将新条目添加到最近的清单文件。

卸载工具

使用具有用于安装该工具的相同选项的 dotnet tool uninstall 命令删除工具：

```
dotnet tool uninstall --global <packagename>
dotnet tool uninstall --tool-path <packagename>
dotnet tool uninstall <packagename>
```

对于本地工具，SDK 通过在当前目录和父目录中查找来查找包含包 ID 的第一个清单文件。

获取帮助和疑难解答

如果工具无法安装或运行，请参阅排查 .NET 工具使用问题。可以使用 `--help` 参数获取可用 `dotnet tool` 命令和参数的列表：

```
dotnet tool --help
```

若要获取工具使用说明，请输入以下命令之一，或访问工具的网站：

```
<command> --help
dotnet <command> --help
```

请参阅

- 教程：使用 .NET CLI 创建 .NET 工具
- 教程：使用 .NET CLI 安装和使用 .NET 全局工具
- 教程：使用 .NET CLI 安装和使用 .NET 本地工具

排查 .NET 工具使用问题

2021/11/16 •

在尝试安装或运行 .NET 工具(可能是全局工具，也可能是本地工具)时，可能会遇到问题。本文介绍了常见的根本原因及一些可能的解决方案。

安装的 .NET 工具无法运行

当 .NET 工具无法运行时，你最可能遇到下述问题之一：

- 找不到工具的可执行文件。
- 找不到 .NET 运行时的正确版本。

找不到可执行文件

如果找不到可执行文件，则将看到如下所示的消息：

```
Could not execute because the specified command or file was not found.
Possible reasons for this include:
  * You misspelled a built-in dotnet command.
  * You intended to execute a .NET program, but dotnet-xyz does not exist.
  * You intended to run a global tool, but a dotnet-prefixed executable with this name could not be found on
the PATH.
```

可执行文件的名称决定了调用工具的方式。相关格式请参见下表：

[illegible]	[illegible]
`dotnet-<toolName>.exe`	`dotnet <toolName>`
`<toolName>.exe`	`<toolName>`

全局工具

全局工具可安装在默认目录中，也可安装在特定位置中。默认目录为：

(OS)	[illegible]
Linux/macOS	`$HOME/.dotnet/tools`
Windows	`%USERPROFILE%\.dotnet\tools`

如果要尝试运行全局工具，请检查计算机上的 `PATH` 环境变量是否包含安装该全局工具的路径且可执行文件是否位于该路径中。

.NET CLI 在首次使用时尝试将默认位置添加到 PATH 环境变量。但是，在某些情况下，位置不会自动添加至 PATH：

- 如果使用的 Linux，并且已使用 .tar.gz 文件(而非 apt-get 或 rpm)安装 .NET SDK。
- 如果使用的是 macOS 10.15"Catalina"或更高版本。
- 如果使用的是 macOS 10.14"Mojave"或更低版本，并且已使用 .tar.gz 文件(而非 .pkg)安装 .NET SDK 。
- 如果已安装 .NET Core 3.0 SDK，并且已将 `DOTNET_ADD_GLOBAL_TOOLS_TO_PATH` 环境变量设置为 `false` 。
- 如果已安装 .NET Core 2.2 SDK 或更低版本，并且已将 `DOTNET_SKIP_FIRST_TIME_EXPERIENCE` 环境变量设置为

`true`。

在这些情况下，或者如果指定了 `--tool-path` 选项，则计算机上的 `PATH` 环境变量不会自动包含安装全局工具的路径。在这种情况下，使用 shell 提供的用于更新环境变量的任何方法，将工具位置（例如 `$HOME/.dotnet/tools`）追加到 `PATH` 环境变量。有关详细信息，请参阅 .NET 工具。

本地工具

如果要尝试运行本地工具，请验证当前目录或其任何父目录中是否存在一个名为 dotnet-tools.json 的清单文件。此文件还可位于项目文件夹层次结构中任意位置的 .config 文件夹下，而不是位于根文件夹中。如果存在 dotnet-tools.json，请将其打开，检查是否存在你要尝试运行的工具。如果该文件不包含 `"isRoot": true` 的条目，则还要进一步检查文件层次结构中是否存在其他工具清单文件。

如果要尝试运行已通过指定的路径安装的 .NET 工具，则需要在使用该工具时包含该路径。下面是使用安装了工具路径的工具的示例：

```
..\<toolDirectory>\dotnet-<toolName>
```

找不到运行时

.NET 工具是依赖框架的应用程序，也就是说它们依赖于计算机上安装的 .NET 运行时。如果找不到所需的运行时，则遵循常规的 .NET 运行时前滚规则，例如：

- 应用程序前滚至指定的主要版本和次要版本的最高修补程序版本。
- 如果主要版本号和次要版本号没有匹配的运行时，则使用下一个较高的次要版本。
- 前滚不会发生在运行时的预览版之间，也不会发生在预览版和发行版之间。因此，使用预览版创建的 .NET 工具必须由作者重新生成和重新发布，再重新安装。

在下面两种常见场景中，默认不进行前滚：

- 只有运行时的较低版本可用。前滚仅选择运行时的较高版本。
- 只有运行时的较高版本可用。前滚不跨越主要版本边界。

如果应用程序找不到合适的运行时，则它无法运行并报告错误。

若要查看计算机上安装了哪些 .NET 运行时，可使用以下命令之一：

```
dotnet --list-runtimes
dotnet --info
```

如果你认为此工具应支持你当前安装的运行时版本，可联系工具作者，询问他们是否可更新版本号或实现多目标。在工具作者编译其工具包并使用更新后的版本号将其重新发布到 NuGet 之后，你可更新你的副本。虽然这种情况不会发生，但最快捷的解决方案是安装适合你要尝试运行的工具的运行时版本。若要下载特定的 .NET 运行时版本，请访问 .NET 下载页。

如果将 .NET SDK 安装到非默认位置，则需要将环境变量 `DOTNET_ROOT` 设置到包含 `dotnet` 可执行文件的目录。

.NET 工具安装失败

.NET 全局或本地工具安装失败的原因有很多。当工具安装失败时，你将看到如下所示的消息：

```
Tool '{0}' failed to install. This failure may have been caused by:

* You are attempting to install a preview release and did not use the --version option to specify the
version.
* A package by this name was found, but it was not a .NET tool.
* The required NuGet feed cannot be accessed, perhaps because of an Internet connection problem.
* You mistyped the name of the tool.

For more reasons, including package naming enforcement, visit https://aka.ms/failure-installing-tool
```

为帮助诊断这些失败情况，会随同之前的消息直接向用户显示 NuGet 消息。NuGet 消息可帮助你确定问题。

- 包命名强制
- 预览版本
- 包不是 .NET 工具
- 无法访问 NuGet 源
- 包 ID 错误
- 401（未授权）

包命名强制

Microsoft 针对工具的包 ID 更改了相关指导，导致没法用预测出的名称找到很多工具。新的指导是所有 Microsoft 工具均附上“Microsoft”这一前缀。此前缀已预留，仅用于使用 Microsoft 授权证书签名的包。

在过渡期间，一些 Microsoft 工具将采用包 ID 的旧格式，而另外一些将采用新格式：

```
dotnet tool install -g Microsoft.<toolName>
dotnet tool install -g <toolName>
```

更新包 ID 后，你将需要更改到新的包 ID 以获取最新更新。带简化工具名称的包将遭到弃用。

预览版本

- 你正在尝试安装预览版本，但未使用 `--version` 选项来指定该版本。

必须用名称的一部分指定处于预览状态的 .NET 工具，这样才能指示它们现为预览版。不需要包含整个预览版。假定版本号都采用预期的格式，则你可使用与下例类似的内容：

```
dotnet tool install -g --version 1.1.0-pre <toolName>
```

包不是 .NET 工具

- 已按此名称找到 NuGet 包，但它不是 .NET 工具。

如果尝试安装的 NuGet 包是常规 NuGet 包而非 .NET 工具，你将看到如下所示的错误：

> NU1212：`<toolName>` 的项目包组合无效。DotnetToolReference 项目类型仅可包含 DotnetTool 类型的引用。

无法访问 NuGet 源

- 无法访问必需的 NuGet 源，这可能是由 Internet 连接问题造成的。

需要访问包含工具包的 NuGet 源才能安装工具。如果源不可用，则安装将失败。可使用 `nuget.config` 修改源、请求特定的 `nuget.config` 文件，也可使用 `--add-source` 开关指定其他源。默认情况下，对于无法连接的任何源，NuGet 都将引发错误。标志 `--ignore-failed-sources` 可跳过这些不可访问的源。

包 ID 错误

- 工具的名称拼写错误。

常见的失败原因是工具名称不正确。原因可能是拼写错误，或者工具已移动或已被弃用。对于 NuGet.org 上的工具，确保名称正确的一种方式是在 NuGet.org 处搜索该工具并复制安装命令。

401（未授权）

您很可能已指定了备用 NuGet 源，并且该源要求进行身份验证。解决此问题有多种不同的方式：

- 添加 `--ignore-failed-sources` 参数以忽略专用源中的错误并使用公共 Microsoft 源。

  如果要从 Microsoft NuGet 源安装工具，自定义源将在 Microsoft NuGet 源返回结果之前返回此错误。该错误将终止请求，从而取消任何其他等待中的源请求，其中可能包括 Microsoft 的 NuGet 源。添加 `--ignore-failed-sources` 选项会导致命令将此错误视为警告，并允许其他源处理该请求。

  ```
  dotnet tool install -g --ignore-failed-sources <toolName>
  ```

- 利用 `--add-source` 参数强制使用 Microsoft NuGet 源。

  全局或本地 NuGet 配置文件可能缺少公共 Microsoft NuGet 源。使用 `--add-source` 和 `--ignore-failed-sources` 参数组合避免错误源，并依赖公共 Microsoft 源。

  ```
  dotnet tool install -g --add-source 'https://api.nuget.org/v3/index.json' --ignore-failed-sources
  <toolName>
  ```

- 使用自定义 NuGet 配置（`--configfile <FILE>` 参数）。

  仅使用公共 Microsoft NuGet 源创建本地 nuget.config 文件，并使用 `--configfile` 参数引用该文件：

  ```
  dotnet tool install -g --configfile "./nuget.config" <toolName>
  ```

  下面是一个配置文件示例：

  ```
  <?xml version="1.0" encoding="utf-8"?>
  <configuration>
    <packageSources>
      <add key="nuget.org" value="https://api.nuget.org/v3/index.json" protocolVersion="3" />
    </packageSources>
  </configuration>
  ```

  有关详细信息，请参阅 nuget.config 参考

- 向配置文件添加所需的凭据。

  如果你知道配置的源中存在包，请在 NuGet 配置文件中提供登录凭据。有关 NuGet 配置文件中凭据的详细信息，请参阅 nuget.config 引用的 packageSourceCredentials 部分。

请参阅

- .NET 工具

教程：使用 .NET CLI 创建 .NET 工具

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

本教程介绍如何创建和打包 .NET 工具。使用 .NET CLI，你可以创建一个控制台应用程序作为工具，便于其他人安装并运行。.NET 工具是从 .NET CLI 安装的 NuGet 包。有关工具的详细信息，请参阅 .NET 工具概述。

将创建的工具是一个控制台应用程序，它将消息作为输入，并显示消息以及用于创建机器人图像的文本行。

这是一系列教程（包含三个教程）中的第一个。在本教程中，将创建并打包工具。在接下来的两个教程中，使用该工具作为全局工具并使用该工具作为本地工具。无论你是将工具用作全局工具还是用作本地工具，创建工具的过程都是相同的。

必备知识

- .NET SDK 6.0.100 或更高版本。

  本教程使用 .NET SDK 6.0，但从 .NET Core SDK 2.1 开始，提供全局工具。本地工具从 .NET Core SDK 3.0 开始可用。

- 按需选择的文本编辑器或代码编辑器。

创建项目

1. 打开命令提示符，创建一个名为“repository” 的文件夹。

2. 导航到“repository”文件夹并输入以下命令：

   ```
   dotnet new console -n microsoft.botsay -f net6.0
   ```

   此命令将在“repository”文件夹下创建一个名为“microsoft.botsay”的新文件夹。

   > **NOTE**
   > 在本教程中，你将创建一个面向 .NET 6.0 的工具。若要以其他框架为目标，请更改 `-f|--framework` 选项。若要以多个框架为目标，请将 `TargetFramework` 元素更改为项目文件中的 `TargetFrameworks` 元素，如以下示例中所示：
   >
   > ```
   > <Project Sdk="Microsoft.NET.Sdk">
   >   <PropertyGroup>
   >     <OutputType>Exe</OutputType>
   >     <TargetFrameworks>netcoreapp3.1;net5.0;net6.0</TargetFrameworks>
   >   </PropertyGroup>
   > </Project>
   > ```

3. 导航到“microsoft.botsay”文件夹。

   ```
   cd microsoft.botsay
   ```

添加代码

1. 使用代码编辑器打开 Program.cs 文件。
2. 将 Program.cs 中的代码替换为以下代码：

```
using System.Reflection;

namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

3. 将 `Main` 方法替换为以下代码，以便处理应用程序的命令行参数。

```
static void Main(string[] args)
{
    if (args.Length == 0)
    {
        var versionString = Assembly.GetEntryAssembly()?
                                .GetCustomAttribute<AssemblyInformationalVersionAttribute>()?
                                .InformationalVersion
                                .ToString();

        Console.WriteLine($"botsay v{versionString}");
        Console.WriteLine("-------------");
        Console.WriteLine("\nUsage:");
        Console.WriteLine("  botsay <message>");
        return;
    }

    ShowBot(string.Join(' ', args));
}
```

   如果未传递任何参数，将显示简短的帮助消息。否则，所有参数都将连接到单个字符串中，并通过调用在下一步中创建的 `ShowBot` 方法进行打印。

4. 添加一个名为 `ShowBot` 的新方法，该方法采用一个字符串参数。该方法使用文本行打印出消息和机器人图像。

```
static void ShowBot(string message)
{
    string bot = $"\n        {message}";
    bot += @"
    ___________________
                      \
                       \
                          ....
                          ....'
                           ....
                        ..........
                     ............'..'..
                  .................'..'.....
                ........'..........'..'..'....
               ........'..........'..'..'.....
              .'....'..'..........'..'.......'.
              .'...................'...   ......
              .  ......'.........         .....
              .    _             __        ......
             ..    #             ##        ......
            ....       .                  .......
            ......  .......          ............
             ................  ......................
             ........................'.................
            ......................'..'......    .......
         ........................'..'.....       .......
      ........    ..'.............'..'....      ..........
   ..'..'...      ...............'.......      ..........
  ...'......     ...... ..........  ......         .......
 ...........   .......              ........        ......
.......        '...'.'.              '.'.'.'         ....
.......       .....'..               ..'.....
   ..       ..........               ..'........
          ............               ..............
         ..............              '..............
        ............'..             .'.'............
       ................             .'.'..............
      ..............'..              ..'..'............
      ................                 .'...............
       .........                        ..............
        .....
";
    Console.WriteLine(bot);
}
```

5. 保存更改。

测试应用程序

运行项目并观察输出。尝试使用命令行处的这些变体来查看不同的结果：

```
dotnet run
dotnet run -- "Hello from the bot"
dotnet run -- Hello from the bot
```

位于 `--` 分隔符后的所有参数均会传递给应用程序。

打包工具

在将应用程序作为工具打包并分发之前，你需要修改项目文件。

1. 打开“microsoft.botsay.csproj”文件，然后将三个新的 XML 节点添加到 `<PropertyGroup>` 节点的末尾：

```
<PackAsTool>true</PackAsTool>
<ToolCommandName>botsay</ToolCommandName>
<PackageOutputPath>./nupkg</PackageOutputPath>
```

`<ToolCommandName>` 是一个可选元素，用于指定在安装工具后将调用该工具的命令。如果未提供此元素，则该工具的命令名称为没有“.csproj”扩展名的项目文件名。

`<PackageOutputPath>` 是一个可选元素，用于确定将在何处生成 NuGet 包。NuGet 包是 .NET CLI 用于安装你的工具的包。

项目文件现在类似于以下示例：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>

    <PackAsTool>true</PackAsTool>
    <ToolCommandName>botsay</ToolCommandName>
    <PackageOutputPath>./nupkg</PackageOutputPath>

  </PropertyGroup>

</Project>
```

2. 通过运行 dotnet pack 命令创建 NuGet 包：

```
dotnet pack
```

“microsoft.botsay.1.0.0.nupkg”文件在由 microsoft.botsay.csproj 文件的 `<PackageOutputPath>` 值标识的文件夹中创建，在本示例中为“./nupkg”文件夹 。

如果想要公开发布一个工具，你可以将其上传到 `https://www.nuget.org`。该工具在 NuGet 上可用后，开发人员就可以使用 dotnet tool install 命令安装该工具。在本教程中，你将直接从本地“nupkg”文件夹安装包，因此无需将包上传到 NuGet。

疑难解答

如果在学习本教程时收到错误消息，请参阅排查 .NET 工具使用问题。

后续步骤

在本教程中，你创建了一个控制台应用程序并将其打包为工具。若要了解如何使用该工具作为全局工具，请转到下一教程。

安装和使用全局工具

如果需要，可以跳过全局工具教程，直接转到本地工具教程。

安装和使用本地工具

教程：使用 .NET CLI 安装和使用 .NET 全局工具

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

本教程介绍如何安装和使用全局工具。使用在本系列的第一个教程中创建的工具。

先决条件

- 完成本系列的第一个教程。

使用该工具作为全局工具

1. 通过运行 microsoft.botsay 项目文件夹中的 dotnet tool install 命令，从包中安装该工具：

   ```
   dotnet tool install --global --add-source ./nupkg microsoft.botsay
   ```

   `--global` 参数指示 .NET CLI 将工具二进制文件安装在自动添加到 PATH 环境变量的默认位置中。

   `--add-source` 参数指示 .NET CLI 临时使用 ./nupkg 目录作为 NuGet 包的附加源数据源。为包提供了唯一名称，以确保它仅位于 ./nupkg 目录中，而不是在 Nuget.org 站点上。

   输出显示用于调用该工具和已安装的版本的命令：

   ```
   You can invoke the tool using the following command: botsay
   Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.
   ```

2. 调用该工具：

   ```
   botsay hello from the bot
   ```

   > **NOTE**
   > 如果此命令失败，则可能需要打开新终端来刷新 PATH。

3. 通过运行 dotnet tool uninstall 命令来删除该工具：

   ```
   dotnet tool uninstall -g microsoft.botsay
   ```

使用该工具作为自定义位置中安装的全局工具

1. 从包中安装该工具。

   在 Windows 上：

   ```
   dotnet tool install --tool-path c:\dotnet-tools --add-source ./nupkg microsoft.botsay
   ```

   在 Linux 或 macOS 上：

```
dotnet tool install --tool-path ~/bin --add-source ./nupkg microsoft.botsay
```

`--tool-path` 参数指示 .NET CLI 将工具二进制文件安装在指定位置中。如果目录不存在，则会创建该目录。此目录不会自动添加到 PATH 环境变量中。

输出显示用于调用该工具和已安装的版本的命令：

```
You can invoke the tool using the following command: botsay
Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.
```

2. 调用该工具：

   在 Windows 上：

   ```
   c:\dotnet-tools\botsay hello from the bot
   ```

   在 Linux 或 macOS 上：

   ```
   ~/bin/botsay hello from the bot
   ```

3. 通过运行 dotnet tool uninstall 命令来删除该工具：

   在 Windows 上：

   ```
   dotnet tool uninstall --tool-path c:\dotnet-tools microsoft.botsay
   ```

   在 Linux 或 macOS 上：

   ```
   dotnet tool uninstall --tool-path ~/bin microsoft.botsay
   ```

疑难解答

如果在学习本教程时收到错误消息，请参阅排查 .NET 工具使用问题。

后续步骤

在本教程中，已将工具作为全局工具安装和使用。有关如何安装和使用全局工具的详细信息，请参阅管理全局工具。若要安装和使用与本地工具相同的工具，请转到下一教程。

安装和使用本地工具

教程：使用 .NET CLI 安装和使用 .NET 本地工具

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

本教程介绍如何安装和使用本地工具。使用在本系列的第一个教程中创建的工具。

先决条件

- 完成本系列的第一个教程。
- 安装 .NET Core 2.1 运行时。

  在本教程中，安装和使用面向 .NET Core 2.1 的工具，因此需要在计算机上安装该运行时。若要安装 2.1 运行时，请转到 .NET Core 2.1 下载页面并在“运行应用 - 运行时”列中查找运行时安装链接。

创建清单文件

若要安装仅用于本地访问的工具(对于当前目录和子目录)，必须将其添加到清单文件。

在“microsoft.botsay”文件夹中，向上导航一个级别到“repository”文件夹：

```
cd ..
```

通过运行 dotnet new 命令来创建清单文件：

```
dotnet new tool-manifest
```

输出指示文件创建成功。

```
The template "Dotnet local tool manifest file" was created successfully.
```

.config/dotnet-tools.json 文件中尚无工具：

```
{
  "version": 1,
  "isRoot": true,
  "tools": {}
}
```

清单文件中列出的工具可用于当前目录和子目录。当前目录是包含具有清单文件的 .config 目录的目录。

使用引用本地工具的 CLI 命令时，SDK 会在当前目录和父目录中搜索清单文件。如果它找到清单文件，但该文件不包含所引用的工具，则会通过父目录继续向上搜索。搜索在找到所引用的工具或找到将 `isRoot` 设置为 `true` 的清单文件时结束。

将 botsay 作为本地工具安装

从在第一个教程中创建的包中安装该工具：

```
dotnet tool install --add-source ./microsoft.botsay/nupkg microsoft.botsay
```

此命令将该工具添加到在上一步中创建的清单文件。命令输出显示新安装的工具所在的清单文件：

```
You can invoke the tool from this directory using the following command:
'dotnet tool run botsay' or 'dotnet botsay'
Tool 'microsoft.botsay' (version '1.0.0') was successfully installed.
Entry is added to the manifest file /home/name/repository/.config/dotnet-tools.json
```

.config/dotnet-tools.json 文件现在有一个工具：

```
{
  "version": 1,
  "isRoot": true,
  "tools": {
    "microsoft.botsay": {
      "version": "1.0.0",
      "commands": [
        "botsay"
      ]
    }
  }
}
```

使用该工具

通过运行“repository”文件夹中的 `dotnet tool run` 命令来调用该工具：

```
dotnet tool run botsay hello from the bot
```

还原其他人安装的本地工具

通常将本地工具安装在存储库的根目录中。将清单文件签入到存储库后，其他开发人员可以获得最新的清单文件。若要安装清单文件中列出的所有工具，他们可以运行单个 `dotnet tool restore` 命令。

1. 打开 .config/dotnet-tools.json 文件并将内容替换为以下 JSON：

   ```
   {
     "version": 1,
     "isRoot": true,
     "tools": {
       "microsoft.botsay": {
         "version": "1.0.0",
         "commands": [
           "botsay"
         ]
       },
       "dotnetsay": {
         "version": "2.1.3",
         "commands": [
           "dotnetsay"
         ]
       }
     }
   }
   ```

2. 将 `<name>` 替换为用于创建项目的名称。

3. 保存更改。

   进行此更改等同于在其他人安装项目目录的包 `dotnetsay` 后从存储库获取最新版本。

4. 运行 `dotnet tool restore` 命令。

   ```
   dotnet tool restore
   ```

   该命令生成的输出如以下示例所示：

   ```
   Tool 'microsoft.botsay' (version '1.0.0') was restored. Available commands: botsay
   Tool 'dotnetsay' (version '2.1.3') was restored. Available commands: dotnetsay
   Restore was successful.
   ```

5. 验证工具是否可用：

   ```
   dotnet tool list
   ```

   输出是包和命令的列表，类似于以下示例：

   ```
   Package Id      Version      Commands       Manifest
   -------------------------------------------------------------------------------------
   microsoft.botsay 1.0.0        botsay         /home/name/repository/.config/dotnet-tools.json
   dotnetsay        2.1.3        dotnetsay      /home/name/repository/.config/dotnet-tools.json
   ```

6. 测试工具：

   ```
   dotnet tool run dotnetsay hello from dotnetsay
   dotnet tool run botsay hello from botsay
   ```

更新本地工具

本地工具 `dotnetsay` 的已安装版本为 2.1.3。使用 dotnet tool update 命令将工具更新到最新版本。

```
dotnet tool update dotnetsay
```

输出指示新的版本号：

```
Tool 'dotnetsay' was successfully updated from version '2.1.3' to version '2.1.7'
(manifest file /home/name/repository/.config/dotnet-tools.json).
```

update 命令查找包含包 ID 的第一个清单文件并对其进行更新。如果搜索范围内的任何清单文件中都没有此类包 ID，SDK 会将新条目添加到最近的清单文件。搜索范围上至父目录，直到找到具有 `isRoot = true` 的清单文件。

删除本地工具

通过运行 dotnet tool uninstall 命令来删除已安装的工具：

```
dotnet tool uninstall microsoft.botsay
```

```
dotnet tool uninstall dotnetsay
```

疑难解答

如果在学习本教程时收到错误消息，请参阅排查 .NET 工具使用问题。

请参阅

有关详细信息，请参阅 .NET 工具

.NET 附加工具概述

2021/11/16 •

本节除了 .NET CLI 外，还编译了可支持和扩展 .NET 功能的工具列表。

.NET 卸载工具

使用 .NET 卸载工具 (`dotnet-core-uninstall`)，可清理系统上的 .NET SDK 和运行时，以便仅保留指定的版本。可使用选项集合来指定要卸载的版本。

.NET 诊断工具

dotnet-counters 是一个性能监视工具，用于初级运行状况监视和性能调查。

通过 dotnet-dump，可在不使用本机调试器的情况下收集和分析 Windows 和 Linux 核心转储。

dotnet-gcdump 提供为活动 .NET 进程收集 GC（垃圾回收器）转储的方式。

dotnet-trace 会从你的应用收集分析数据，这些数据可帮助你了解应用运行速度缓慢的原因。

适用于扩展创建者的 .NET 安装工具

适用于扩展创建者的 .NET 安装工具是一种专门供 VS Code 扩展创建者获取 .NET 运行时的 Visual Studio Code 扩展。此工具专门供采用 .NET 编写并且需要 .NET 启动其各个部分的扩展（例如语言服务器）使用。此扩展并非直接供用户用来安装用于开发的 .NET。

WCF Web Service Reference 工具

WCF (Windows Communication Foundation) Web ervice Reference 工具是一个 Visual Studio 连接服务提供程序，首次推出是在 Visual Studio 2017 版本 15.5 中。此工具可从网络位置上当前解决方案的 Web 服务中，或从 WSDL 文件中检索元数据。还可生成与 .NET 兼容的源文件并使用可用于访问 Web 服务操作的方法定义 WCF 代理类。

WCF dotnet-svcutil 工具

WCF dotnet-svcutil 工具是一个 .NET 工具，可从网络位置上的 Web 服务中或从 WSDL 文件中检索元数据。还可生成与 .NET 兼容的源文件并使用可用于访问 Web 服务操作的方法定义 WCF 代理类。

dotnet-svcutil 工具是 WCF Web Service Reference Visual Studio 连接服务提供程序（随 Visual Studio 2017 版本 15.5 首次推出）的替代产品。dotnet-svcutil 工具作为一种 .NET 工具，可用于 Linux、macOS 和 Windows。

WCF dotnet-svcutil.xmlserializer 工具

在 .NET Framework 中，可以使用 svcutil 工具预生成序列化程序集。WCF dotnet-svcutil.xmlserializer 工具在 .NET 5（和 .NET Core）以及更高版本上提供类似的功能。它为客户端应用程序中 WCF 服务协定使用且可由 XmlSerializer 序列化的类型预生成 C# 序列化代码。当序列化或反序列化这些类型的对象时，这会提高 XML 序列化的启动性能。

XML 序列化程序生成器

正如 XML 序列化程序生成器 (sgen.exe) 适用于 .NET Framework，Microsoft.XmlSerializer.Generator NuGet 包 是适用于面向 .NET 5（和 .NET Core）以及更高版本的库的解决方案。它为程序集中包含的类型创建 XML 序列化程

序集，从而提高使用 XmlSerializer 序列化或反序列化这些类型对象时，XML 序列化的启动性能。

生成自签名证书

可以使用 dotnet dev-certs 创建用于开发和测试方案的自签名证书。

.NET 代码覆盖率工具

可使用 dotnet-coverage 从任何 .NET 进程收集代码覆盖率。

.NET 卸载工具

2021/11/16 •

你可以使用 .NET 卸载工具 (`dotnet-core-uninstall`) 从系统中删除 .NET SDK 和运行时。可使用选项集合来指定要卸载的版本。

该工具支持 Windows 和 macOS。目前不支持 Linux。

在 Windows 上，该工具只能卸载使用以下安装程序之一安装的 SDK 和运行时：

- .NET SDK 和运行时安装程序。
- Visual studio 安装程序的版本早于 Visual Studio 2019 版本 16.3。

在 macOS 上，该工具只能卸载位于 /usr/local/share/dotnet 文件夹中的 SDK 和运行时。

由于这些限制，该工具可能无法卸载计算机上的所有 .NET SDK 和运行时。可以使用 `dotnet --info` 命令来查找所有安装的 .NET SDK 和运行时，包括此工具无法删除的 SDK 和运行时。 `dotnet-core-uninstall list` 命令显示可以通过该工具卸载的 SDK。版本 1.2 及更高版本可以卸载版本 5.0 或更早版本的 SDK 和运行时，而以前版本的工具可以卸载 3.1 及更早版本。

安装工具

可以从工具的发布页面下载 .NET 卸载工具，然后在 dotnet/cli-lab GitHub 存储库中找到源代码。

> **NOTE**
> 此工具需要提升才能卸载 .NET SDK 和运行时。因此，应将其安装在写入保护的目录中，如 Windows 上的 C:\Program Files 或 macOS 上的 /usr/local/bin。另请参阅提升的 Dotnet 命令访问权限。有关详细信息，请参阅详细安装说明。

运行该工具

以下步骤说明了运行卸载工具的建议方法：

- 步骤 1 - 显示安装的 .NET SDK 和运行时
- 步骤 2 - 执行试运行
- 步骤 3 - 卸载 .NET SDK 和运行时
- 步骤 4 - 删除 NuGet 回退文件夹（可选）

步骤 1 - 显示安装的 .NET SDK 和运行时

`dotnet-core-uninstall list` 命令列出了已安装的 .NET SDK 和运行时，可以通过此工具将其删除。Visual Studio 可能需要某些 SDK 和运行时，它们将显示出来，并说明为何不建议将其卸载。

> **NOTE**
> 在大多数情况下， `dotnet-core-uninstall list` 命令的输出将与 `dotnet --info` 输出中的已安装版本列表不匹配。具体而言，此工具将不会显示通过 zip 文件安装的版本，也不会显示由 Visual Studio（Visual Studio 2019 16.3 或更高版本）托管的版本。检查某个版本是否由 Visual Studio 托管的一种方法是在 `Add or Remove Programs` 中查看该版本，由 Visual Studio 托管的版本在显示名称中会以这种方式标记。

dotnet-core-uninstall list

摘要

```
dotnet-core-uninstall list [options]
```

选项

- Windows
- macOS

- `--aspnet-runtime`

  列出可通过此工具卸载的所有 ASP.NET 运行时。

- `--hosting-bundle`

  列出可通过此工具卸载的所有 .NET 托管捆绑包。

- `--runtime`

  列出可通过此工具卸载的所有 .NET 运行时。

- `--sdk`

  列出可通过此工具卸载的所有 .NET SDK。

- `-v, --verbosity <LEVEL>`

  设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。

- `--x64`

  列出可通过此工具卸载的所有 x64 .NET SDK 和运行时。

- `--x86`

  列出可通过此工具卸载的所有 x86 .NET SDK 和运行时。

示例

- 列出可通过此工具删除的所有 .NET SDK 和运行时：

  ```
  dotnet-core-uninstall list
  ```

- 列出所有 x64 .NET SDK 和运行时：

  ```
  dotnet-core-uninstall list --x64
  ```

- 列出所有 x86 .NET SDK：

  ```
  dotnet-core-uninstall list --sdk --x86
  ```

步骤 2 - 执行试运行

`dotnet-core-uninstall dry-run` 和 `dotnet-core-uninstall whatif` 命令显示将根据提供的选项删除的 .NET SDK 和运行时，而无需执行卸载。这些命令是同义词。

dotnet-core-uninstall dry-run 和 dotnet-core-uninstall whatif

摘要

```
dotnet-core-uninstall dry-run [options] [<VERSION>...]

dotnet-core-uninstall whatif [options] [<VERSION>...]
```

自变量

- `VERSION`

  要卸载的指定版本。可以逐一列出多个版本，用空格分隔。此外还支持响应文件。

  > **TIP**
  > 响应文件是在命令行上放置所有版本的替代方法。它们是文本文件，通常具有 *.rsp 扩展名，每个版本都在单独的行上列出。若要为 `VERSION` 参数指定响应文件，请使用后面紧跟响应文件名的 @ 字符。

选项

- Windows
- macOS

- `--all`

  删除所有 .NET SDK 和运行时。

- `--all-below <VERSION>[ <VERSION>...]`

  仅删除版本小于指定版本的 .NET SDK 和运行时。仍安装指定版本。

- `--all-but <VERSIONS>[ <VERSION>...]`

  除了那些指定版本外，删除所有 .NET SDK 和运行时。

- `--all-but-latest`

  删除 .NET SDK 和运行时（最高版本除外）。

- `--all-lower-patches`

  删除由较高版本的修补程序取代的 .NET SDK 和运行时。此选项保护 global.json。

- `--all-previews`

  删除标记为预览的 .NET SDK 和运行时。

- `--all-previews-but-latest`

  删除标记为预览的 .NET SDK 和运行时（最高预览版除外）。

- `--aspnet-runtime`

  仅删除 ASP.NET 运行时。

- `--hosting-bundle`

  仅删除 .NET 运行时和托管绑定。

- `--major-minor <MAJOR_MINOR>`

  删除与指定 `major.minor` 版本相匹配的 .NET SDK 和运行时。

- `--runtime`

  仅删除 .NET 运行时。

- `--sdk`

  仅删除 .NET SDK。

- `-v, --verbosity <LEVEL>`

  设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。

- `--x64`

  必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x64 SDK 或运行时。

- `--x86`

  必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x86 SDK 或运行时。

- `--force` 强制删除可能由 Visual Studio 使用的版本。

注意：

1. 只需 `--sdk`、`--runtime`、`--aspnet-runtime` 和 `--hosting-bundle` 中的一个。
2. `--all`、`--all-below`、`--all-but`、`--all-but-latest`、`--all-lower-patches`、`--all-previews`、`--all-previews-but-latest`、`--major-minor` 和 `[<VERSION>...]` 除外。
3. 如果未指定 `--x64` 或 `--x86`，则同时删除 x64 和 x86。

示例

> **NOTE**
> 默认情况下，Visual Studio 或其他 SDK 可能需要的 .NET SDK 和运行时不会包含在 `dotnet-core-uninstall dry-run` 输出中。在下面的示例中，某些指定的 SDK 和运行时可能不会包含在输出中，具体取决于计算机的状态。若要包括所有 SDK 和运行时，请将它们显式列出为参数或使用 `--force` 选项。

- 试运行删除已被较高版本的修补程序取代的所有 .NET 运行时：

```
dotnet-core-uninstall dry-run --all-lower-patches --runtime
```

- 试运行删除低于版本 `2.2.301` 的所有 .NET SDK：

```
dotnet-core-uninstall whatif --all-below 2.2.301 --sdk
```

步骤 3 - 卸载 .NET SDK 和运行时

`dotnet-core-uninstall remove` 卸载由选项集合指定的 .NET SDK 和运行时。版本 1.2 及更高版本可以卸载版本 5.0 或更早版本的 SDK 和运行时，而以前版本的工具可以卸载 3.1 及更早版本。

由于此工具具有破坏性行为，因此强烈建议在运行 remove 命令之前执行试运行。使用 `remove` 命令时，试运行将显示要删除的 .NET SDK 和运行时。请参阅是否应删除版本？了解哪些 SDK 和运行时可以安全删除。

Caution

请记住以下注意事项：

- 此工具可以卸载计算机上 `global.json` 文件所需的 .NET SDK 版本。可以从下载 .NET 页重新安装 .NET SDK。
- 此工具可以卸载计算机上依赖于框架的应用程序所需的 .NET 运行时版本。可以从下载 .NET 页重新安装 .NET 运行时。
- 此工具可以卸载 Visual Studio 所依赖的 .NET SDK 和运行时版本。如果中断 Visual Studio 安装，请在 Visual Studio 安装程序中运行“修复”以返回到工作状态。

默认情况下，所有命令都将保留 Visual Studio 或其他 SDK 可能需要的 .NET SDK 和运行时。可以通过将这些 SDK 和运行时显式列出为参数或使用 `--force` 选项来卸载这些 SDK 和运行时。

此工具需要提升才能卸载 .NET SDK 和运行时。在 Windows 上的管理员命令提示符中运行此工具，在 macOS 上则通过 `sudo` 运行。`dry-run` 和 `whatif` 命令不需要提升。

dotnet-core-uninstall remove

摘要

```
dotnet-core-uninstall remove [options] [<VERSION>...]
```

自变量

- `VERSION`

  要卸载的指定版本。可以逐一列出多个版本，用空格分隔。此外还支持响应文件。

  > **TIP**
  > 响应文件是在命令行上放置所有版本的替代方法。它们是文本文件，通常具有 *.rsp 扩展名，每个版本都在单独的行上列出。若要为 `VERSION` 参数指定响应文件，请使用后面紧跟响应文件名的 @ 字符。

选项

- Windows
- macOS

- `--all`

  删除所有 .NET SDK 和运行时。

- `--all-below <VERSION>[ <VERSION>...]`

  仅删除版本小于指定版本的 .NET SDK 和运行时。仍安装指定版本。

- `--all-but <VERSIONS>[ <VERSION>...]`

  除了那些指定版本外，删除所有 .NET SDK 和运行时。

- `--all-but-latest`

  删除 .NET SDK 和运行时（最高版本除外）。

- `--all-lower-patches`

  删除由较高版本的修补程序取代的 .NET SDK 和运行时。此选项保护 global.json。

- `--all-previews`

  删除标记为预览的 .NET SDK 和运行时。

- `--all-previews-but-latest`

  删除标记为预览的 .NET SDK 和运行时（最高预览版除外）。

- `--aspnet-runtime`

  仅删除 ASP.NET 运行时。

- `--hosting-bundle`

仅删除 .NET 托管绑定。

- `--major-minor <MAJOR_MINOR>`

  删除与指定 `major.minor` 版本相匹配的 .NET SDK 和运行时。

- `--runtime`

  仅删除 .NET 运行时。

- `--sdk`

  仅删除 .NET SDK。

- `-v, --verbosity <LEVEL>`

  设置详细程度。允许使用的值为 `q[uiet]`、`m[inimal]`、`n[ormal]`、`d[etailed]` 和 `diag[nostic]`。默认值为 `normal`。

- `--x64`

  必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x64 SDK 或运行时。

- `--x86`

  必须与 `--sdk`、`--runtime` 和 `--aspnet-runtime` 结合使用才能删除 x86 SDK 或运行时。

- `-y, --yes` 执行命令而不需要进行是或否确认。

- `--force` 强制删除可能由 Visual Studio 使用的版本。

注意：

1. 只需 `--sdk`、`--runtime`、`--aspnet-runtime` 和 `--hosting-bundle` 中的一个。
2. `--all`、`--all-below`、`--all-but`、`--all-but-latest`、`--all-lower-patches`、`--all-previews`、`--all-previews-but-latest`、`--major-minor` 和 `[<VERSION>...]` 除外。
3. 如果未指定 `--x64` 或 `--x86`，则同时删除 x64 和 x86。

示例

> **NOTE**
> 默认情况下，将保留 Visual Studio 或其他 SDK 可能需要的 .NET SDK 和运行时。在下面的示例中，可能保留某些指定的 SDK 和运行时，具体取决于计算机的状态。若要删除所有 SDK 和运行时，请将它们显式列出为参数或使用 `--force` 选项。

- 删除除版本 `3.0.0-preview6-27804-01` 之外的所有 .NET 运行时，无需进行 Y/N 确认：

```
dotnet-core-uninstall remove --all-but 3.0.0-preview6-27804-01 --runtime --yes
```

- 删除所有 .NET Core 1.1 SDK，无需进行 Y/N 确认：

```
dotnet-core-uninstall remove --sdk --major-minor 1.1 -y
```

- 删除没有控制台输出的 .NET Core 1.1.11 SDK：

```
dotnet-core-uninstall remove 1.1.11 --sdk --yes --verbosity q
```

- 删除可由此工具安全删除的所有 .NET SDK：

```
dotnet-core-uninstall remove --all --sdk
```

- 删除此工具可删除的所有 .NET SDK，包括 Visual Studio 可能需要的 SDK（不推荐）：

```
dotnet-core-uninstall remove --all --sdk --force
```

- 删除响应文件 `versions.rsp` 中指定的所有 .NET SDK

```
dotnet-core-uninstall remove --sdk @versions.rsp
```

versions.rsp 的内容如下所示：

```
2.2.300
2.1.700
```

步骤 4 - 删除 NuGet 回退文件夹（可选）

在某些情况下，你不再需要 `NuGetFallbackFolder`，可能希望将其删除。有关删除此文件夹的详细信息，请参阅删除 NuGetFallbackFolder。

卸载工具

- Windows
- macOS

1. 打开“添加或删除程序”。
2. 搜索 `Microsoft .NET SDK Uninstall Tool`。
3. 选择“卸载”。

适用于扩展创建者的 .NET 安装工具

2021/11/16 •

适用于扩展创建者的 .NET 安装工具是一种专门供 VS Code 扩展创建者获取 .NET 运行时的 Visual Studio Code 扩展。此工具专门供采用 .NET 编写并且需要 .NET 启动其各个部分的扩展(例如语言服务器)使用。此扩展并非直接供用户用来安装用于开发的 .NET。

入门指南:扩展创建者

为确保适用于扩展创建者的 .NET 安装工具适合你的方案,请先从 GitHub 页查看此扩展的目标。

> **NOTE**
> 此工具只可用于安装 .NET 运行时,当前无法安装 .NET SDK。

验证适用于扩展创建者的 .NET 安装工具符合你的需求后,即可在扩展清单中利用对它的依赖关系,并开始通过 VS Code API 使用我们所公开的命令。你可找到此扩展在 GitHub 上公开的命令列表。

请查看此扩展示例,了解相关操作步骤。

若要获取更多示例,请查看当前利用了此工具的以下开源扩展:

- 适用于 Visual Studio Code 的 Azure 资源管理器 (ARM) 工具
- .NET 交互式笔记本

入门指南:最终用户

最终用户通常完全不需要与适用于扩展创建者的 .NET 安装工具交互。如果你在使用此扩展时遇到问题,请查看"故障排除"页或通过 GitHub 提交问题。

通过 .NET CLI 生成自签名证书

2021/11/16 •

使用自签名证书时，可通过不同的方式创建自签名证书，并将它们用于开发和测试场景。本指南将介绍如何通过 `dotnet dev-certs` 以及 `PowerShell` 和 `OpenSSL` 等其他选项使用自签名证书。

然后，可以使用容器中托管的 ASP.NET Core 应用等示例来验证是否将加载证书。

先决条件

可在示例中使用 .NET Core 3.1 或 .NET 5。

对于 `dotnet dev-certs`，请确保已安装适当版本的 .NET：

- 在 Windows 上安装 .NET
- 在 Linux 上安装 .NET
- 在 macOS 上安装 .NET

此示例需要 Docker 17.06 或更高版本的 Docker 客户端。

准备示例应用

你需要根据要用于测试的运行时（.NET Core 3.1 或 .NET 5）来准备示例应用。

对于本指南，你将使用示例应用并进行适当的更改。

.NET Core 3.1 示例应用

获取示例应用。

```
git clone https://github.com/dotnet/dotnet-docker/
```

在本地导航到存储库，并在编辑器中打开工作区。

> **NOTE**
> 如果要使用 dotnet publish 参数对部署进行剪裁，则应确保包含适当的依赖项以便支持 SSL 证书。更新 dotnet-docker\samples\aspnetapp\aspnetapp.csproj 以确保容器中包含适当的程序集。有关参考，请查看如何更新 .csproj 文件以便在对自包含部署应用剪裁时支持 SSL 证书。

确保 `aspnetapp.csproj` 包含适当的目标框架：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>.netcoreapp3.1</TargetFramework>
    <!--Other Properties-->
  </PropertyGroup>

</Project>
```

修改 Dockerfile，确保运行时指向 .NET Core 3.1：

```
# https://hub.docker.com/_/microsoft-dotnet-core
FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app --no-restore

# final stage/image
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
WORKDIR /app
COPY --from=build /app ./
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

确保指向示例应用。

```
cd .\dotnet-docker\samples\aspnetapp
```

构建用于本地测试的容器。

```
docker build -t aspnetapp:my-sample -f Dockerfile .
```

.NET 5 示例应用

对于本指南，应针对 .NET 5 检查示例 aspnetapp。

检查示例应用 Dockerfile 是否使用 .NET 5。

根据主机 OS，可能需要更新 ASP.NET 运行时。例如，在 Dockerfile 中从 `mcr.microsoft.com/dotnet/aspnet:5.0-nanoservercore-2009 AS runtime` 更改为 `mcr.microsoft.com/dotnet/aspnet:5.0-windowsservercore-ltsc2019 AS runtime` 将有助于面向适当的 Windows 运行时。

例如，这将有助于在 Windows 上测试证书：

```
# https://hub.docker.com/_/microsoft-dotnet
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /source

# copy csproj and restore as distinct layers
COPY *.sln .
COPY aspnetapp/*.csproj ./aspnetapp/
RUN dotnet restore -r win-x64

# copy everything else and build app
COPY aspnetapp/. ./aspnetapp/
WORKDIR /source/aspnetapp
RUN dotnet publish -c release -o /app -r win-x64 --self-contained false --no-restore

# final stage/image
# Uses the 2009 release; 2004, 1909, and 1809 are other choices
FROM mcr.microsoft.com/dotnet/aspnet:5.0-windowsservercore-ltsc2019 AS runtime
WORKDIR /app
COPY --from=build /app ./
ENTRYPOINT ["aspnetapp"]
```

如果要在 Linux 上测试证书，可以使用现有的 Dockerfile。

确保 `aspnetapp.csproj` 包含适当的目标框架：

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <!--Other Properties-->
  </PropertyGroup>

</Project>
```

> **NOTE**
> 如果要使用 `dotnet publish` 参数对部署进行剪裁，请确保包含适当的依赖项以便支持 SSL 证书。更新 dotnet-docker\samples\aspnetapp\aspnetapp.csproj 以确保容器中包含适当的程序集。有关参考，请查看如何更新 .csproj 文件以便在对自包含部署应用剪裁时支持 SSL 证书。

确保指向示例应用。

```
cd .\dotnet-docker\samples\aspnetapp
```

构建用于本地测试的容器。

```
docker build -t aspnetapp:my-sample -f Dockerfile .
```

创建自签名证书

可以通过以下方法创建自签名证书：

- 使用 dotnet dev-certs
- 使用 PowerShell
- 使用 OpenSSL

使用 dotnet dev-certs

可以使用 `dotnet dev-certs` 来处理自签名证书。此示例使用 PowerShell 控制台。

```
dotnet dev-certs https -ep $env:USERPROFILE\.aspnet\https\aspnetapp.pfx -p crypticpassword
dotnet dev-certs https --trust
```

> **NOTE**
> 证书名称(在本例中为 aspnetapp.pfx)必须与项目程序集名称一致。`crypticpassword` 用作你所选密码的替代对象。如果控制台返回"已存在有效 HTTPS 证书",则表示存储中已存在受信任的证书。可使用 MMC 控制台导出证书。

为证书配置应用程序机密:

```
dotnet user-secrets -p aspnetapp\aspnetapp.csproj set "Kestrel:Certificates:Development:Password"
"crypticpassword"
```

> **NOTE**
> 注意:密码必须与证书所用的密码一致。

使用为 HTTPS 配置的 ASP.NET Core 运行容器映像:

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -v
$env:APPDATA\microsoft\UserSecrets\:C:\Users\ContainerUser\AppData\Roaming\microsoft\UserSecrets -v
$env:USERPROFILE\.aspnet\https:C:\Users\ContainerUser\AppData\Roaming\ASP.NET\Https
mcr.microsoft.com/dotnet/samples:aspnetapp
```

应用程序启动后,在 Web 浏览器中导航到 `https://localhost:8001`。

清理

如果未使用机密和证书,请务必将它们清除。

```
dotnet user-secrets remove "Kestrel:Certificates:Development:Password" -p aspnetapp\aspnetapp.csproj
dotnet dev-certs https --clean
```

使用 PowerShell

可以使用 PowerShell 来生成自签名证书。可以使用 PKI 客户端来生成自签名证书。

```
$cert = New-SelfSignedCertificate -DnsName @("contoso.com", "www.contoso.com") -CertStoreLocation
"cert:\LocalMachine\My"
```

将生成证书,但出于测试目的,应将证书置于证书存储中,以便在浏览器中进行测试。

```
$certKeyPath = "c:\certs\contoso.com.pfx"
$password = ConvertTo-SecureString 'password' -AsPlainText -Force
$cert | Export-PfxCertificate -FilePath $certKeyPath -Password $password
$rootCert = $(Import-PfxCertificate -FilePath $certKeyPath -CertStoreLocation 'Cert:\LocalMachine\Root' -
Password $password)
```

此时,应该可以从 MMC 管理单元查看证书。

可以在适用于 Linux 的 Windows 子系统 (WSL) 中运行示例容器：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/contoso.com.pfx -v /c/certs:/https/
mcr.microsoft.com/dotnet/samples:aspnetapp
```

> **NOTE**
> 请注意，装载卷后，根据主机的不同，文件路径的处理方式可能有所不同。例如，在 WSL 中，可以将 /c/certs 替换为 /mnt/c/certs 。

如果使用的是之前为 Windows 构建的容器，运行命令将如下所示：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=c:\https\contoso.com.pfx -v c:\certs:C:\https aspnetapp:my-
sample
```

应用程序启动后，在浏览器中导航到 contoso.com:8001。

确保已更新主机条目，以便 `contoso.com` 在适当的 IP 地址（例如 127.0.0.1）上进行应答。如果无法识别证书，请确保随容器一起加载的证书在主机上也受信任，并确保存在 `contoso.com` 的适当 SAN/DNS 条目。

清理

```
$cert | Remove-Item
Get-ChildItem $certFilePath | Remove-Item
$rootCert | Remove-item
```

使用 OpenSSL

可以使用 OpenSSL 来创建自签名证书。此示例将使用 WSL/Ubuntu 以及带有 `OpenSSL` 的 Bash Shell。

这将生成一个 .crt 和一个 .key 文件。

```
PARENT="contoso.com"
openssl req \
-x509 \
-newkey rsa:4096 \
-sha256 \
-days 365 \
-nodes \
-keyout $PARENT.key \
-out $PARENT.crt \
-subj "/CN=${PARENT}" \
-extensions v3_ca \
-extensions v3_req \
-config <( \
  echo '[req]'; \
  echo 'default_bits= 4096'; \
  echo 'distinguished_name=req'; \
  echo 'x509_extension = v3_ca'; \
  echo 'req_extensions = v3_req'; \
  echo '[v3_req]'; \
  echo 'basicConstraints = CA:FALSE'; \
  echo 'keyUsage = nonRepudiation, digitalSignature, keyEncipherment'; \
  echo 'subjectAltName = @alt_names'; \
  echo '[ alt_names ]'; \
  echo "DNS.1 = www.${PARENT}"; \
  echo "DNS.2 = ${PARENT}"; \
  echo '[ v3_ca ]'; \
  echo 'subjectKeyIdentifier=hash'; \
  echo 'authorityKeyIdentifier=keyid:always,issuer'; \
  echo 'basicConstraints = critical, CA:TRUE, pathlen:0'; \
  echo 'keyUsage = critical, cRLSign, keyCertSign'; \
  echo 'extendedKeyUsage = serverAuth, clientAuth')

openssl x509 -noout -text -in $PARENT.crt
```

若要获取 .pfx 文件，请使用以下命令：

```
openssl pkcs12 -export -out $PARENT.pfx -inkey $PARENT.key -in $PARENT.crt
```

NOTE

.aspnetcore 3.1 示例将使用 `.pfx` 和密码。从 `.net 5` 运行时开始，Kestrel 还可以采用 `.crt` 和 PEM 编码的 `.key` 文件。

根据主机 OS，需要信任证书。在 Linux 主机上，“信任”证书有所不同，并且依赖于发行版。

对于本指南，在 Windows 中使用 PowerShell 的示例如下：

```
Import-Certificate -FilePath $certFilePath -CertStoreLocation 'Cert:\LocalMachine\Root'
```

对于 .NET Core 3.1，在 WSL 中运行以下命令：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/contoso.com.pfx -v /c/path/to/certs/:/https/
mcr.microsoft.com/dotnet/samples:aspnetapp
```

从 .NET 5 开始，Kestrel 可以采用 `.crt` 和 PEM 编码的 `.key` 文件。可以通过以下命令针对 .NET 5 运行该示例：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/contoso.com.crt -e
ASPNETCORE_Kestrel__Certificates__Default__KeyPath=/https/contoso.com.key -v /c/path/to/certs:/https/
mcr.microsoft.com/dotnet/samples:aspnetapp
```

> **NOTE**
> 请注意，在 WSL 中，卷装载路径可能会根据配置而发生变化。

对于 Windows 中的 .NET Core 3.1，请在 `Powershell` 中运行以下命令：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Password="password" -e
ASPNETCORE_Kestrel__Certificates__Default__Path=c:\https\contoso.com.pfx -v c:\certs:C:\https aspnetapp:my-
sample
```

对于 Windows 中的 .NET 5，请在 PowerShell 中运行以下命令：

```
docker run --rm -it -p 8000:80 -p 8001:443 -e ASPNETCORE_URLS="https://+;http://+" -e
ASPNETCORE_HTTPS_PORT=8001 -e ASPNETCORE_ENVIRONMENT=Development -e
ASPNETCORE_Kestrel__Certificates__Default__Path=c:\https\contoso.com.crt -e
ASPNETCORE_Kestrel__Certificates__Default__KeyPath=c:\https\contoso.com.key -v c:\certs:C:\https
aspnetapp:my-sample
```

应用程序启动后，在浏览器中导航到 contoso.com:8001。

确保已更新主机条目，以便 `contoso.com` 在适当的 IP 地址（例如 127.0.0.1）上进行应答。如果无法识别证书，请确保随容器一起加载的证书在主机上也受信任，并确保存在 `contoso.com` 的适当 SAN/DNS 条目。

清理

确保在完成测试后清除自签名证书。

```
Get-ChildItem $certFilePath | Remove-Item
```

使用 WCF Web Service Reference Provider 工具

2021/11/16 •

多年来，许多 Visual Studio 开发者在其 .NET Framework 项目需要访问 Web 服务时，都享受到了添加服务引用工具所带来的工作效率。WCF Web 服务引用工具是 Visual Studio 连接服务的扩展，提供了类似于 .NET Core 和 ASP.NET Core 项目的"添加服务引用"功能的体验 。此工具可从网络位置的当前解决方案的 web 服务中或从 WSDL 文件中检索元数据，并生成包含可用于访问 web 服务的 Windows Communication Foundation (WCF) 客户端代理代码的可兼容 .NET Core 的源文件。

> **IMPORTANT**
> 应仅从受信任源引用服务。从不受信任的源添加引用可能会危及安全性。

系统必备

- Visual Studio 2017 版本 15.5 或更高版本

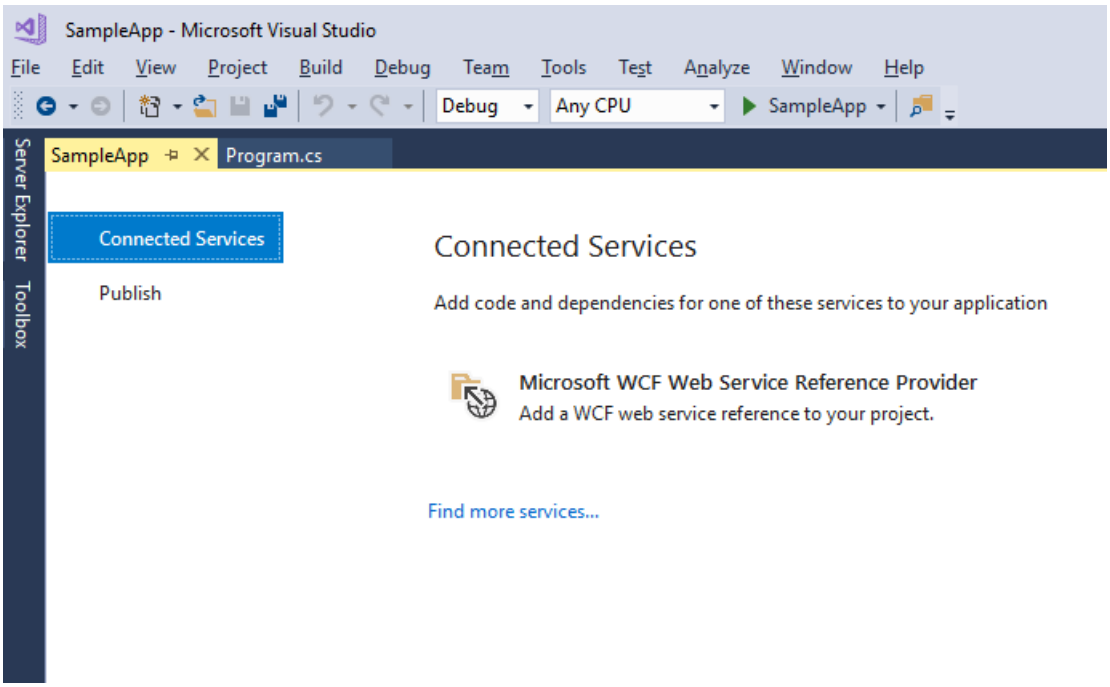
如何使用扩展

> **NOTE**
> "WCF Web 服务引用"选项适用于使用以下项目模板创建的项目：
>
> - Visual C# > .NET Core
> - Visual C# > .NET Standard
> - Visual C# > Web > ASP.NET Core Web 应用程序

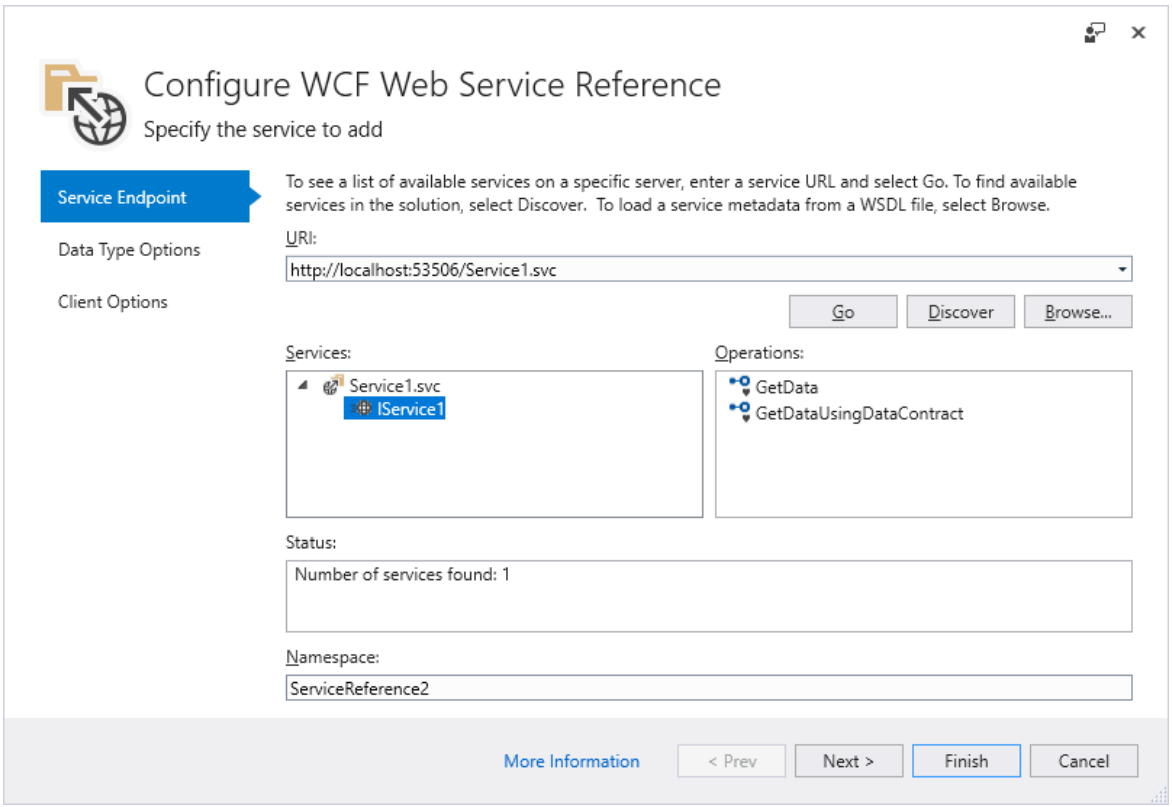
以"ASP.NET Core Web 应用程序"项目模板为例，本文将介绍如何向该项目中添加 WCF 服务引用：

1. 在解决方案资源管理器中，双击项目的"连接的服务"节点（对于 .NET Core 或 .NET Standard 项目，当在解决方案资源管理器中右键单击项目的"依赖项"节点时，该选项可用）。

   随即显示"连接的服务"页，如下图所示：



2. 在“连接的服务”页上，单击“Microsoft WCF Web Service Reference Provider”。此操作将显示“配置 WCF Web 服务引用”向导：



3. 选择服务。

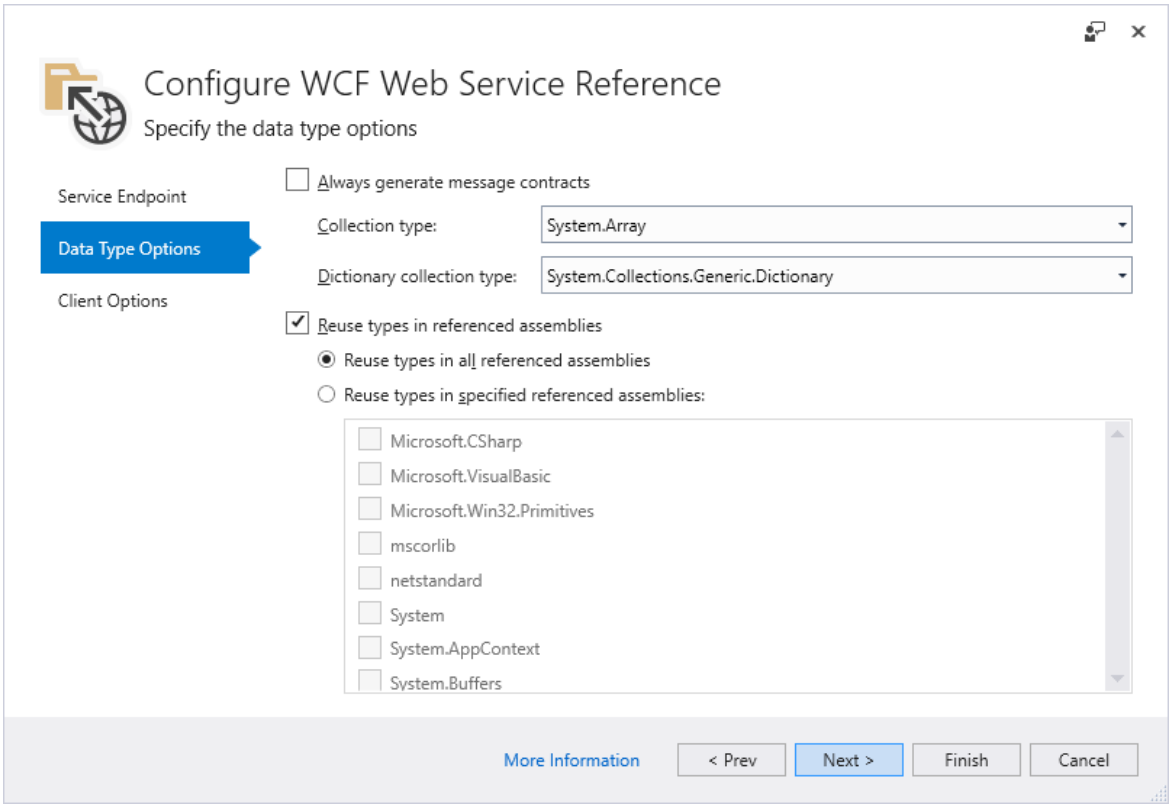
   3a. “配置 WCF Web 服务引用”向导中提供了多个服务搜索选项：

   - 要搜索当前解决方案中定义的服务，请单击“发现”按钮。
   - 要搜索在指定地址托管的服务，请在“地址”框中输入服务 URL，然后单击“转到”按钮。
   - 要选择包含 Web 服务元数据信息的 WSDL 文件，请单击“浏览”按钮。

   3b. 从“服务”框内的搜索结果列表中选择服务。如果需要，请在相应的“名称空间”文本框中为生成的代码输入命名空间。

   3c. 单击“下一步”按钮，打开“数据类型选项”页和“客户端选项”页。或者，单击“完成”按钮，使用默认选项。

4. “数据类型选项”窗体可用于优化生成的服务引用配置设置：



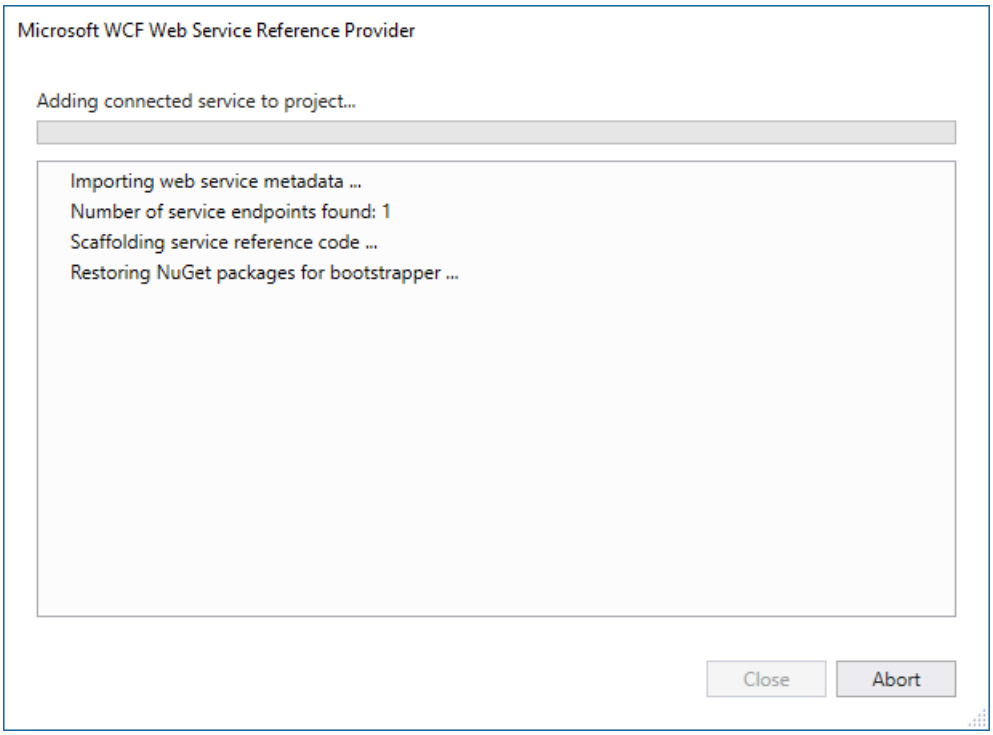
> **NOTE**
> 如果在项目的引用程序集中定义了服务引用代码生成所需的数据类型，则“重新使用引用程序集中的类型”复选框选项将非常有用 。重新使用这些现有数据类型，从而避免编译时类型冲突或运行时问题，这是非常重要的。

加载类型信息时可能会有延迟，具体取决于项目依赖项和其他系统性能因素的数量。加载过程中，“完成”按钮被禁用，除非未选中“重新使用引用程序集中的类型”复选框 。

5. 完成后，单击“完成” 。

在显示进度的同时，工具：

- 从 WCF 服务下载元数据。
- 在名为“reference.cs”的文件中生成服务引用代码，并将其添加到“连接的服务”节点下的项目 。
- 使用在目标平台上编译和运行所需的 NuGet 包引用更新项目文件 (.csproj)。



进度完成后，可创建生成的 WCF 客户端类型的实例并调用服务操作。

另请参阅

- Windows Communication Foundation 应用程序入门
- Visual Studio 中的 Windows Communication Foundation 服务和 WCF 数据服务
- .NET Core 上 WCF 支持的功能

反馈和问题

如果你有任何产品反馈，请使用报告问题工具在开发者社区进行报告。

发行说明

- 请参阅发行说明，了解更新的版本信息（包括已知问题）。

.NET Core 的 WCF dotnet-svcutil 工具

2021/11/16 •

Windows Communication Foundation (WCF) dotnet-svcutil 工具是一种 .NET 工具，此工具从网络位置上的 Web 服务中或从 WSDL 文件中检索元数据，并生成包含访问 Web 服务操作的客户端代理方法的 WCF 类。

类似于 .NET Framework 项目的服务模型元数据 - svcutil 工具，dotnet svcutil 是用于生成 Web 服务引用的命令行工具，与 .NET Core 和 .NET Standard 项目兼容。

dotnet-svcutil 工具是 WCF Web 服务引用 Visual Studio 连接服务提供程序（随 Visual Studio 2017 版本 15.5 首次推出）的替代选项 。dotnet-svcutil 工具作为一种 .NET 工具，可跨平台用于 Linux、macOS 和 Windows。

> **IMPORTANT**
> 应仅从受信任源引用服务。从不受信任的源添加引用可能会危及安全性。

先决条件

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

- .NET Core 2.1 SDK 或更高版本
- 你最喜欢的代码编辑器

入门

下面的示例将指导你完成将 Web 服务引用添加到 .NET Core Web 项目并调用该服务所需的步骤。将创建名为“HelloSvcutil”的 .NET Core Web 应用程序，并将引用添加到实现以下协定的 Web 服务：

```
[ServiceContract]
public interface ISayHello
{
    [OperationContract]
    string Hello(string name);
}
```

在此示例中，我们假定 Web 服务托管在以下地址中：`http://contoso.com/SayHello.svc`

从 Windows、macOS 或 Linux 命令窗口执行以下步骤：

1. 为项目创建一个名为“HelloSvcutil”的目录，并将其设置为当前目录，如以下示例所示：

   ```
   mkdir HelloSvcutil
   cd HelloSvcutil
   ```

2. 在该目录中使用 `dotnet new` 命令创建新的 C# Web 项目，如下所示：

   ```
   dotnet new web
   ```

3. 安装 `dotnet-svcutil` NuGet 包作为 CLI 工具：

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

```
dotnet tool install --global dotnet-svcutil
```

4. 运行 dotnet-svcutil 命令生成 Web 服务引用文件，如下所示：

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

```
dotnet-svcutil http://contoso.com/SayHello.svc
```

生成的文件保存为 *HelloSvcutil/ServiceReference/Reference.cs*。dotnet-svcutil 工具还向项目添加代理代码所需的适当 WCF 包作为包引用。

使用服务引用

1. 使用 `dotnet restore` 命令还原 WCF 包，如下所示：

```
dotnet restore
```

2. 找到要使用的客户端类和操作的名称。`Reference.cs` 将包含一个继承自 `System.ServiceModel.ClientBase` 的类，其方法可用于调用服务上的操作。在本例中，想要调用 SayHello 服务的 Hello 操作。`ServiceReference.SayHelloClient` 是客户端类的名称，它有一个名为 `HelloAsync` 的方法，可用于调用该操作。

3. 在编辑器中打开 `Startup.cs` 文件，并在顶部为服务引用命名空间添加一个 `using` 指令：

```
using ServiceReference;
```

4. 编辑 `Configure` 方法来调用 Web 服务。为此，可以创建一个继承自 `ClientBase` 的类的实例，并在客户端对象上调用此方法：

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
        var client = new SayHelloClient();
        var response = await client.HelloAsync();
        await context.Response.WriteAsync(response);
    });
}
```

5. 使用 `dotnet run` 命令运行应用程序，如下所示：

```
dotnet run
```

6. 导航到在 Web 浏览器的控制台中列出的 URL（例如，`http://localhost:5000`）。

您应看到以下输出："Hello dotnet-svcutil!"

有关 `dotnet-svcutil` 工具参数的详细说明，请调用传递帮助参数的工具，如下所示：

- dotnet-svcutil 2.x
- dotnet-svcutil 1.x

```
dotnet-svcutil --help
```

反馈和问题

如果有任何问题或反馈，请在 GitHub 上提问。也可以在 GitHub 上的 WCF 存储库中查看任何现有问题。

发行说明

- 请参阅发行说明，了解更新的版本信息（包括已知问题）。

信息

- dotnet-svcutil NuGet 包

在 .NET Core 上使用 dotnet-svcutil.xmlserializer

2021/11/16 •

`dotnet-svcutil.xmlserializer` NuGet 包可以为 .NET Core 项目预生成序列化程序集。它为客户端应用程序中由 WCF 服务协定使用的且可由 XmlSerializer 序列化的类型预生成 C# 序列化代码。当序列化或反序列化这些类型的对象时，这会提高 XML 序列化的启动性能。

先决条件

- .NET Core 2.1 SDK 或更高版本
- 你最喜欢的代码编辑器

可以使用命令 `dotnet --info` 检查已安装哪些版本的 .NET SDK 和运行时。

入门

在 .NET Core 控制台应用程序中使用 `dotnet-svcutil.xmlserializer`：

1. 在 .NET Framework 中使用默认模板“WCF 服务应用程序”创建一个名为“MyWCFService”的 WCF 服务。在服务方法上添加 `[XmlSerializerFormat]` 属性，如下所示：

   ```
   [ServiceContract]
   public interface IService1
   {
       [XmlSerializerFormat]
       [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
   "http://tempuri.org/IService1/GetDataResponse")]
       string GetData(int value);
   }
   ```

2. 创建 .NET Core 控制台应用程序作为面向 .NET Core 2.1 或更高版本的 WCF 客户端应用程序。例如，使用以下命令创建名为“MyWCFClient”的应用：

   ```
   dotnet new console --name MyWCFClient
   ```

   要确保项目面向 .NET Core 2.1 或更高版本，请检查项目文件中的 `TargetFramework` XML 元素：

   ```
   <TargetFramework>netcoreapp2.1</TargetFramework>
   ```

3. 通过运行以下命令将包引用添加到 `System.ServiceModel.Http`：

   ```
   dotnet add package System.ServiceModel.Http
   ```

4. 添加 WCF 客户端代码：

```
using System.ServiceModel;

    class Program
    {
        static void Main(string[] args)
        {
            var myBinding = new BasicHttpBinding();
            var myEndpoint = new EndpointAddress("http://localhost:2561/Service1.svc"); //Fill your
service url here
            var myChannelFactory = new ChannelFactory<IService1>(myBinding, myEndpoint);
            IService1 client = myChannelFactory.CreateChannel();
            string s = client.GetData(1);
            ((ICommunicationObject)client).Close();
        }
    }

[ServiceContract]
public interface IService1
{
    [XmlSerializerFormat]
    [OperationContract(Action = "http://tempuri.org/IService1/GetData", ReplyAction =
"http://tempuri.org/IService1/GetDataResponse")]
    string GetData(int value);
}
```

5. 通过运行以下命令将引用添加到 `dotnet-svcutil.xmlserializer` 包：

   ```
   dotnet add package dotnet-svcutil.xmlserializer
   ```

   运行该命令应向项目文件中添加一个类似于以下内容的条目：

   ```
   <ItemGroup>
     <DotNetCliToolReference Include="dotnet-svcutil.xmlserializer" Version="1.0.0" />
   </ItemGroup>
   ```

6. 通过运行 `dotnet build` 生成应用程序。如果一切顺利，则会在输出文件夹中生成名为“MyWCFClient.XmlSerializers.dll”的程序集。如果该工具无法生成程序集，将在生成输出中看到警告。

7. 例如，通过在浏览器中运行 `http://localhost:2561/Service1.svc` 来启动 WCF 服务。然后启动客户端应用程序，它将在运行时自动加载和使用预生成的序列化程序。

在 .NET Core 上使用 Microsoft XML 序列化程序生成器

2021/11/16 •

本教程介绍如何在 C# .NET Core 应用程序中使用 Microsoft XML 序列化程序生成器。在本教程中可学习：

- 如何创建 .NET Core 应用
- 如何添加 Microsoft.XmlSerializer.Generator 包引用
- 如何编辑 MyApp.csproj，以添加依赖项
- 如何添加类和 XmlSerializer
- 如何生成并运行应用程序

正如适用于 .NET Framework 的 Xml Serializer Generator (sgen.exe)，Microsoft.XmlSerializer.Generator NuGet 包是适用于 .NET Core 和 .NET 标准项目的等效项。它为程序集中包含的类型创建 XML 序列化程序集，从而提高使用 XmlSerializer 序列化或反序列化这些类型对象时，XML 序列化的启动性能。

先决条件

完成本教程：

- .NET Core 2.1 SDK 或更高版本。
- 最喜爱的代码编辑器。

> **TIP**
> 需要安装代码编辑器？ 试用 Visual Studio！

在 .NET Core 控制台应用程序中使用 Microsoft XML 序列化程序生成器

以下说明将展示如何在 .NET Core 控制台应用程序中使用 XML 序列化程序生成器。

创建 .NET Core 控制台应用程序

打开命令提示符，创建一个名为“MyApp”的文件夹。导航到创建的文件夹，并键入以下命令：

```
dotnet new console
```

在 MyApp 项目中向 Microsoft.XmlSerializer.Generator 包添加引用

使用 `dotnet add package` 命令在项目中添加引用。

类型：

```
dotnet add package Microsoft.XmlSerializer.Generator -v 1.0.0
```

添加包后，验证对 MyApp.csproj 的更改

打开代码编辑器并开始操作！ 仍从生成了应用的 MyApp 目录中进行操作。

在文本编辑器中打开 MyApp.csproj。

运行 `dotnet add package` 命令后，会将以下行添加到 MyApp.csproj 项目文件中：

```
<ItemGroup>
   <PackageReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

为 .NET CLI 工具支持添加其他 ItemGroup 部分

在已检查的 `ItemGroup` 部分后添加以下行：

```
<ItemGroup>
   <DotNetCliToolReference Include="Microsoft.XmlSerializer.Generator" Version="1.0.0" />
</ItemGroup>
```

在应用程序中添加类

在文本编辑器中打开 Program.cs。在 Program.cs 中添加名为“MyClass”的类 。

```
public class MyClass
{
   public int Value;
}
```

为 MyClass 创建 `XmlSerializer`

在 Main 中添加以下行，为 MyClass 创建 `XmlSerializer`：

```
var serializer = new System.Xml.Serialization.XmlSerializer(typeof(MyClass));
```

编译和运行应用程序

还是在 MyApp 文件夹中，通过 `dotnet run` 运行应用程序，它会在运行时自动加载和使用预生成的序列化程序。

在控制台窗口中键入以下命令：

```
dotnet run
```

> **NOTE**
> `dotnet run` 调用 `dotnet build` 来确保已生成要生成的目标，然后调用 `dotnet <assembly.dll>` 运行目标应用程序。

> **IMPORTANT**
> 本教程中用来运行应用程序的命令和步骤仅用于开发过程。准备好部署应用后，查看适用于 .NET Core 应用的不同部署策略和 `dotnet publish` 命令。

如果一切顺利，则会在输出文件夹中生成名为“MyApp.XmlSerializers.dll”的程序集。

祝贺你！ 你刚才已完成：

- 创建 .NET Core 应用。
- 向 Microsoft.XmlSerializer.Generator 包中添加引用。
- 编辑 MyApp.csproj 以添加依赖项。
- 添加类和 XmlSerializer。

- 生成和运行应用程序。

相关资源

- XML 序列化简介
- 如何使用 XmlSerializer 进行序列化 (C#)
- 如何：使用 XmlSerializer (Visual Basic) 进行序列化

.NET Core 中提供哪些诊断工具？

2021/11/16 •

软件并非始终按预计方式运行，但 .NET Core 具有可帮助用户快速有效地诊断这些问题的工具和 API。

本文可帮助用户查找各种所需的工具。

托管调试器

借助托管调试器，用户可以与程序进行交互。暂停、增量执行、检查和恢复操作可让用户深入了解代码的行为。调试器是诊断易于重现的功能问题的首选。

日志记录和跟踪

日志记录和跟踪是相关技术。它们指的是用于创建日志文件的检测代码。这些文件记录了程序操作的详细信息。这些细节可用于诊断最复杂的问题。当与时间戳结合使用时，这些技术在性能调查中也非常有用。

指标

指标是在一段时间内记录的数值度量，用于监视应用程序性能和运行状况。指标通常用于在检测到潜在问题时生成警报。在正常使用中，指标的性能开销非常低，并且配置为"始终启用"遥测。.NET 运行时和库发布了许多内置的指标，你可以使用指标 API 创建新指标。

单元测试

单元测试是持续集成和部署高质量软件的关键组件。单元测试的目的在于，在用户操作导致系统出现问题时提前向其发出警告。

转储

转储是一个文件，其中包含创建时进程的快照。它们可用于检查应用程序的状态，以便进行调试。

符号

符号是源代码和编译器生成的二进制代码之间的映射。这些通常被 .NET 调试器用来解析源行号、局部变量名称以及其他类型的诊断信息。

收集容器中的诊断

也可使用非容器化 Linux 环境中使用的相同诊断工具收集容器中的诊断。只需进行几次使用更改就可使这些工具在 Docker 容器中运行。

.NET Core 诊断全局工具

dotnet-counters

dotnet-counters 是一个性能监视工具，用于初级运行状况监视和性能调查。它通过 EventCounter API 观察已发布的性能计数器值。例如，可以快速监视 CPU 使用情况或 .NET Core 应用程序中的异常率等指标。

dotnet-dump

通过 dotnet-dump 工具，可在不使用本机调试器的情况下收集和分析 Windows 和 Linux 核心转储。

dotnet-gcdump

dotnet-gcdump 工具可用于为活动 .NET 进程收集 GC（垃圾回收器）转储。

dotnet-trace

分析数据通过 .NET Core 中的 `EventPipe` 公开。通过 dotnet-trace 工具，可以使用来自应用的有意思的分析数据，这些数据可帮助你分析应用运行缓慢的根本原因。

dotnet-stack

使用 dotnet 工具可以快速打印正在运行的 .net 进程中的所有线程的托管堆栈。

dotnet-symbol

dotnet-symbol 用于下载打开核心转储或小型转储所需的文件（符号、DAC/DBI、主机文件等）。如果需要使用符号和模块来调试在其他计算机上捕获的转储文件，请使用此工具。

dotnet-sos

dotnet-sos 在 Linux 和 macOS（如果使用的是 Windbg/cdb，则在 Windows 上）安装 SOS 调试扩展。

PerfCollect

PerfCollect 是一个 bash 脚本，可用于收集包含 `perf` 和 `LTTng` 的跟踪，以便更深入地分析在 Linux 分发版上运行的 .NET 应用的性能。

.NET Core 诊断教程

编写自己的诊断工具

使用诊断客户端库可以编写最适合诊断场景的自定义诊断工具。在 Microsoft.Diagnostics.NETCore.Client API 参考中查找信息。

调试内存泄露

教程：调试内存泄漏演示了如何查找内存泄漏。dotnet-counters 工具用于确认泄露，dotnet-dump 工具用于诊断泄露。

调试高 CPU 使用率

教程：调试高 CPU 使用率逐步介绍了如何调查高 CPU 使用率。它使用 dotnet-counters 工具来确认高 CPU 使用率。然后，它逐步介绍了如何使用性能分析实用工具（`dotnet-trace`）跟踪或 Linux `perf` 来收集和查看 CPU 使用率配置文件。

调试死锁

教程：调试死锁介绍了如何使用 dotnet-dump 工具来调查线程和锁。

调试 StackOverflow

教程：调试 StackOverflow 演示了如何在 Linux 上调试 StackOverflowException。

调试 Linux 转储

调试 Linux 转储说明了如何收集和分析 Linux 上的转储。

使用 EventCounters 衡量性能

教程：使用 .NET 中的 EventCounters 度量性能演示了如何使用 EventCounter API 来衡量 .NET 应用的性能。

.NET Core 托管调试器

2021/11/16 •

调试器允许逐步骤暂停或执行程序。暂停后，可以查看进程的当前状态。通过逐步完成关键部分，你可以了解代码以及产生这一结果的原因。

Microsoft 在 Visual Studio 和 Visual Studio Code 中提供托管代码的调试器。

Visual Studio 托管调试器

Visual Studio 是一个集成开发环境，其中提供了最全面的调试器。Visual Studio 是 Windows 开发人员的最佳选择。

- 教程 - 使用 Visual Studio 在 Windows 上调试 .NET Core 应用程序

尽管 Visual Studio 是一个 Windows 应用程序，但仍可用于远程调试 Linux 和 macOS 应用。

- 使用 Visual Studio 在 Linux/OSX 上调试 .NET Core 应用程序

调试 ASP.NET Core 应用需要略微不同的说明。

- 在 Visual Studio 中调试 ASP.NET Core 应用

Visual Studio Code 托管调试器

Visual Studio Code 是轻量型跨平台代码编辑器。它使用与 Visual Studio 相同的 .NET Core 调试器实现，但使用的是简化的用户界面。

- 教程 - 使用 Visual Studio Code 调试 .NET Core 应用程序
- 在 Visual Studio Code 中进行调试

转储

2021/11/16 •

转储是一种文件，其中包含创建进程时该进程的快照，可用于检查应用程序的状态。当很难将调试程序附加到 .NET 应用程序（如生产或 CI 环境）时，可使用转储来调试该应用程序。使用转储可以捕获有问题进程的状态，并且可以直接检查状态而无需停止应用程序。

收集转储

可以通过多种方式收集转储，具体取决于运行应用的平台。

> **NOTE**
> 在容器内收集转储需要 PTRACE 功能，可通过 `--cap-add=SYS_PTRACE` 或 `--privileged` 添加该功能。

> **NOTE**
> 转储可能包含敏感信息，因为它们可以包含正在运行进程的全部内存。处理它们时请考虑所有安全限制和指导。

在发生故障时收集转储

可以使用环境变量将应用程序配置为在发生故障时收集转储。如果想要了解故障原因，这将很有帮助。例如，在引发异常时捕获转储有助于通过在应用发生故障时检查应用状态来确定问题。

下表显示了可用于将应用程序配置为在发生故障时收集转储的环境变量。

[illegible]	[illegible]	[illegible]
`COMPlus_DbgEnableMiniDump` 或 `DOTNET_DbgEnableMiniDump`	如果设置为 1，则启用核心转储生成。	0
`COMPlus_DbgMiniDumpType` 或 `DOTNET_DbgMiniDumpType`	要收集的转储类型。有关详细信息，请参阅下表	2 (`MiniDumpWithPrivateReadWriteMemory`)
`COMPlus_DbgMiniDumpName` 或 `DOTNET_DbgMiniDumpName`	写入转储的文件路径。	`/tmp/coredump.<pid>`
`COMPlus_CreateDumpDiagnostics` 或 `DOTNET_CreateDumpDiagnostics`	如果设置为 1，则启用转储进程的诊断日志记录。	0

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_` 。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

下表显示了可用于 `DOTNET_DbgMiniDumpType` 的所有值。例如，将 `DOTNET_DbgMiniDumpType` 设置为 1 意味着在发生故障时将收集 `MiniDumpNormal` 类型的转储。

[	[[	[[
1	`MiniDumpNormal`	只包含为进程中的所有现有线程捕获堆栈跟踪所需的信息。有限的 GC 堆内存和信息。
2	`MiniDumpWithPrivateReadWriteMemory`	包含 GC 堆以及为进程中的所有现有线程捕获堆栈跟踪所需的信息。
3	`MiniDumpFilterTriage`	只包含为进程中的所有现有线程捕获堆栈跟踪所需的信息。有限的 GC 堆内存和信息。
4	`MiniDumpWithFullMemory`	包含进程中的所有可访问内存。原始内存数据包含在末尾，因此无需原始内存信息即可直接映射初始结构。此选项可能会导致文件非常大。

在特定时间点收集转储

你可能需要在应用尚未发生故障时收集转储。例如，如果想要检查似乎处于死锁状态的应用程序的状态，则配置环境变量以在发生故障时收集转储将不起作用，因为应用仍在运行。

若要在自己的请求时收集转储，可以使用 `dotnet-dump`，这是一种用于收集和分析转储的 CLI 工具。若要详细了解如何使用 `dotnet-dump` 来收集转储，请参阅转储收集和分析实用工具。

分析转储

可以使用 `dotnet-dump` CLI 工具或 Visual Studio 分析转储。

> **NOTE**
> Visual Studio 16.8 及更高版本允许打开在 .NET Core 3.1.7 或更高版本上生成的 Linux 转储。

> **NOTE**
> 如果需要进行原生调试，则可以对 Linux 和 macOS 上的 LLDB 使用 SOS 调试器扩展。尽管建议使用 Visual Studio，但 Windows 上的 Windbg/cdb 也支持 SOS。

请参阅

详细了解如何利用转储来帮助诊断 .NET 应用程序中的问题。

- 调试 Linux 转储这一教程分步演示了如何调试在 Linux 中收集的转储。
- 调试死锁这一教程分步演示了如何使用转储来调试 .NET 应用程序中的死锁。

调试 Linux 转储

2021/11/16 •

本文适用于： ✔ .NET Core 3.0 SDK 及更高版本

在 Linux 上收集转储

在 Linux 上收集转储的两种建议方法是：

- `dotnet-dump` CLI 工具
- 用于在故障时收集转储的环境变量

使用 `dotnet-dump` 的托管转储

`dotnet-dump` 工具简单易用，并且不依赖于任何本机调试器。`dotnet-dump` 可用于各种 Linux 平台（例如 Alpine 或 ARM32/ARM64），在这些平台上，传统调试工具可能不适用。但是，`dotnet-dump` 只捕获托管状态，因此不能将其用于调试本机代码中的问题。`dotnet-dump` 收集的转储在具有创建转储的相同 OS 和体系结构的环境中进行分析。`dotnet-gcdump` 工具可用作仅捕获 GC 堆信息但生成可在 Windows 上分析的转储的替代方法。

使用 `createdump` 的核心转储

作为创建仅托管转储的 `dotnet-dump` 的替代方法，建议使用 `createdump` 这一工具在包含本机和托管信息的 Linux 上创建核心转储。其他工具（例如 gdb 或 gcore）也可以用于创建核心转储，但可能会缺失托管调试所需的状态，从而导致在分析过程中出现"未知"类型或函数名称。

`createdump` 工具与 .NET Core 运行时一起安装，可以在 libcoreclr.so（通常位于"/usr/share/dotnet/shared/Microsoft.NETCore.App/[version]"中）旁边找到。该工具采用进程 ID 将转储作为其主要参数收集，还可以采用可选参数来指定要收集的转储类型（带有堆的小型转储是默认类型）。选项包括：

- `<input-filename>`

  要转换的输入跟踪文件。默认为 trace.nettrace 。

选项

- `-f|--name <output-filename>`

  要将转储写入到其中的文件。默认值为"/tmp/coredump.%p"，其中 %p 是目标进程的进程 ID。

- `-n|--normal`

  创建小型转储。

- `-h|--withheap`

  创建带有堆的小型转储（默认）。

- `-t|--triage`

  创建会审小型转储。

- `-u|--full`

  创建完整的核心转储。

- `-d|--diag`

  启用诊断消息。

收集核心转储需要 `SYS_PTRACE` 功能，或者需要 `createdump` 与 sudo 或 su 一起运行。

在 Linux 上分析转储

通过使用 `dotnet-dump analyze` 命令，可以借助 `dotnet-dump` 工具分析使用 `dotnet-dump` 收集的托管转储和使用 `createdump` 收集的核心转储。`dotnet dump` 要求分析转储的环境与捕获转储的环境具有相同的 OS 和体系结构。

另外，LLDB 可用于分析 Linux 上的核心转储，这允许分析托管帧和本机帧。LLDB 使用 SOS 扩展调试托管代码。`dotnet-sos` CLI 工具可用于安装 SOS，它具有许多用于调试托管代码的有用命令。若要分析 .NET Core 转储、LLDB 和 SOS，要求在创建转储的环境中使用以下 .NET Core 二进制文件：

1. libmscordaccore.so
2. libcoreclr.so
3. dotnet（用于启动应用的主机）

在大多数情况下，可以使用 `dotnet-symbol` 工具下载这些二进制文件。如果无法使用 `dotnet-symbol` 下载所需的二进制文件（例如，如果正在使用从源构建的 .NET Core 专用版本），则可能需要从创建转储的环境复制上面列出的文件。如果文件不位于转储文件旁边，则可以使用 LLDB/SOS 命令 `setclrpath <path>` 设置应从中加载文件的路径，并使用 `setsymbolserver -directory <path>` 设置用于查找符号文件的路径。

所需文件可用后，就可以通过将 dotnet 主机指定为要调试的可执行文件来将转储加载到 LLDB 中：

```
lldb --core <dump-file> <host-program>
```

在上面的命令行中，`<dump-file>` 是要分析的转储路径，而 `<host-program>` 是已启动 .NET Core 应用程序的本机程序。除非应用是独立应用，否则通常为 `dotnet` 二进制文件，在本例中，它是不包含 dll 扩展名的应用程序的名称。

LLDB 启动后，可能需要使用 `setsymbolserver` 命令指向正确的符号位置（`setsymbolserver -ms` 用于使用 Microsoft 的符号服务器或 `setsymbolserver -directory <path>` 用于指定本地路径）。可以通过运行 `loadsymbols` 来加载本机符号。此时，SOS 命令可用于分析转储。

另请参阅

- 若要了解有关安装 SOS 扩展的更多详细信息，请参阅 dotnet-sos。
- 若要了解有关安装和使用符号下载工具的更多详细信息，请参阅 dotnet-symbol。
- 若要了解有关调试（包括有用的常见问题解答）的更多详细信息，请参阅 .NET Core 诊断库。
- 若要获取有关在 Linux 或 Mac 上安装 LLDB 的说明，请参阅安装 LLDB。

SOS 调试扩展

2021/11/16 •

SOS 调试扩展让你可查看有关在在 .NET Core 运行时(包括实时进程和转储)内运行的代码的信息。该扩展已预安装 dotnet-dump 和 Windbg/dbg，并可供下载，以便与 LLDB 配合使用。SOS 调试扩展可用于收集有关托管堆的信息、查找堆损坏情况、显示运行时所使用的内部数据类型以及查看有关在运行时内运行的所有托管代码的信息。

语法

Windows

```
![command] [options]
```

Linux 和 macOS

```
sos [command] [options]
```

许多命令在 lldb 下都有别名或快捷方式：

```
clrstack [options]
```

命令

下表的命令还可在 Help 或 soshelp 下使用 。使用 `soshelp <command>` 可提供单个命令帮助。

COMMAND	
bpmd [**-nofuturemodule**] [<*module name*> <*method name*>] [**-md** < `MethodDesc` >] **-list** **-clear** <*pending breakpoint number*> **-clearall**	在指定模块中的指定方法处创建断点。 如果尚未加载指定的模块和方法，则此命令将在创建断点之前等待已加载并进行实时 (JIT) 编译的模块的通知。 可以通过使用 **-list**、**-clear** 和 **-clearall** 选项来管理挂起断点的列表： 该 **-list** 选项生成所有挂起断点的列表。如果挂起断点有一个非零模块 ID，则该断点特定于该特定已加载模块中的函数。如果挂起断点有一个零模块 ID，则该断点适用于尚未加载的模块。 使用 **-clear** 或 **-clearall** 选项可从该列表中移除挂起断点。

COMMAND	此
CLRStack [**-a**] [**-l**] [**-p**] [**-n**] [**-f**] [**-r**] [**-all**]	仅提供托管代码的堆栈跟踪。 **-p** 选项显示托管函数的自变量。 **-l** 选项显示有关帧中的局部变量的信息。SOS 调试扩展无法检索本地名称，因此本地名称的输出采用的格式为 <*local address*> = <*value*>。 “-a”选项是 -l 和 -p 组合的快捷方式 。 **-n** 选项禁止显示源文件名和行号。如果调试器已指定选项 SYMOPT_LOAD_LINES，则 SOS 将查找每个托管帧的符号，如果成功，则将显示对应的源文件名和行号。可以指定 **-n**（无行号）参数来禁用此行为。 “-f”选项（完整模式）显示原生帧，将其与托管帧混合在一起，并显示托管帧的程序集名称和函数偏移量。与 `dotnet-dump` 一起使用时，此选项不显示原生帧。 “-r”选项转储每个堆栈帧的寄存器。 “-all”选项转储所有托管线程的堆栈。
COMState	列出每个线程的 COM 单元模型和 `Context` 指针（如果可用）。此命令仅可用于 Windows。
DumpArray [**-start** <*startIndex*>] [**-length** <*length*>] [**-details**] [**-nofields**] <*array object address*> - 或 - **DA** [**-start** <*startIndex*>] [**-length** <*length*>] [**-detail**] [**-nofields**] *array object address*>	检查数组对象的元素。 **-start** 选项指定开始显示元素的起始索引。 **-length** 选项指定要显示的元素数量。 **-details** 选项使用 **DumpObj** 和 **DumpVC** 格式显示元素的详细信息。 **-nofields** 选项可阻止显示数组。此选项仅在指定 **-detail** 选项后可用。
DumpAsync (**dumpasync**) [**-mt** <*MethodTable address*>] [**-type** <*partial type name*>] [**-waiting**] [**-roots**]	DumpAsync 遍历垃圾回收堆，查找表示异步状态机的对象（在将异步方法的状态传输到堆时创建）。此命令识别定义为 `async void`、`async Task`、`async Task<T>`、`async ValueTask` 和 `async ValueTask<T>` 的异步状态机。 输出包括找到的每个异步状态机对象的详细信息块。这些详细信息包括： - 一行为异步状态机对象的类型，其中包括其 MethodTable 地址、对象地址、大小和类型名称。 - 一行为对象中包含的状态机类型名称。 - 状态机上每个字段的列表。 - 一行为此状态机对象的延续（如果已注册一个或多个）。 - 为此异步状态机对象发现的 GC 根。
DumpAssembly <*assembly address*>	显示有关程序集的信息。 **DumpAssembly** 命令将列出多个模块（如果存在）。 可以通过使用 **DumpDomain** 命令获取程序集地址。

COMMAND	此
DumpClass <*EEClass address*>	显示有关与类型关联的 `EEClass` 结构的信息。 **DumpClass** 命令显示静态字段值，但不显示非静态字段值。 使用 **DumpMT**、**DumpObj**、**Name2EE** 或 **Token2EE** 命令获取 `EEClass` 结构地址。
DumpDomain [<*domain address*>]	枚举在指定的 Assembly 对象地址内加载的每个 AppDomain 对象。若在调用 DumpDomain 命令时不提供任何参数，则将列出过程中的所有 AppDomain 对象。由于 .NET Core 仅有一个 AppDomain，因此 DumpDomain 将仅返回一个对象。
DumpHeap [**-stat**] [**-strings**] [**-short**] [**-min** <*size*>] [**-max** <*size*>] [**-thinlock**] [**-startAtLowerBound**] [**-mt** <*MethodTable address*>] [**-type** <*partial type name*>] [*start* [*end*]]	显示有关垃圾回收堆的信息和有关对象的收集统计信息。 如果 **DumpHeap** 命令在垃圾回收器堆中检测到过多碎片，将会显示警告。 **-stat** 选项将输出限制为统计类型摘要。 **-strings** 选项将输出限制为统计字符串值摘要。 **-short** 选项将输出限制为只是每个对象的地址。这使你能够轻松地以管道方式将输出从该命令转移到另一个调试器命令以实现自动化。 **-min** 选项忽略小于 `size` 参数指定的大小（以字节为单位）的对象。 **-max** 选项忽略大于 `size` 参数指定的大小（以字节为单位）的对象。 **-thinlock** 选项报告 ThinLocks。有关详细信息，请参阅 **SyncBlk** 命令。 `-startAtLowerBound` 选项将强制在提供的地址范围的下限开始堆审核。在计划阶段，堆通常不可移动，因为对象正在移动。此选项将强制 **DumpHeap** 在指定的下限开始其审核。你必须提供有效的对象的地址作为使此选项工作的下限。你可以显示错误对象的地址的内存来手动寻找下一个方法表。如果垃圾回收当前处于对 `memcopy` 的调用中，则你还可以通过将大小添加到作为参数提供的起始地址中来查找下一个对象的地址。 **-mt** 选项仅列出与指定的 `MethodTable` 结构对应的那些对象。 **-type** 选项仅列出其类型名称为指定字符串的子字符串匹配项的那些对象。 `start` 参数从指定的地址处开始列出。 `end` 参数在指定的地址处停止列出。
DumpIL <*Managed DynamicMethod object*> | <*DynamicMethodDesc pointer*> | <*MethodDesc pointer*>	显示与托管方法关联的 Microsoft 中间语言 (MSIL)。 发出的动态 MSIL 与从程序集加载的 MSIL 不同。动态 MSIL 引用托管对象数组中的对象而不是引用元数据标记。

COMMAND	此
DumpLog [**-addr** <*addressOfStressLog*>] [<*Filename*>]	将内存中压力日志的内容写入到指定文件。如果你不指定名称，则此命令将在当前目录中创建一个名为 StressLog.txt 的文件。 内存中压力日志可帮助你在不使用锁或 I/O 的情况下诊断压力故障。若要启用压力日志，请在 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework 下设置以下注册表项： (DWORD) StressLog = 1 (DWORD) LogFacility = 0xffffffff (DWORD) StressLogSize = 65536 利用可选的 `-addr` 选项，你可以指定压力日志而非默认日志。
DumpMD <*MethodDesc address*>	显示有关指定地址处的 `MethodDesc` 结构的信息。 可以使用 **IP2MD** 命令从托管函数中获取 `MethodDesc` 结构地址。
DumpMT [**-MD**] <*MethodTable address*>	显示有关指定地址处的方法表的信息。指定 **-MD** 选项将显示与对象一起定义的所有方法的列表。 每个托管对象均包含一个方法表指针。
DumpModule [**-mt**] <*Module address*>	显示有关指定地址处的模块的信息。**-mt** 选项显示模块中定义的类型和模块所引用的类型 可以使用 **DumpDomain** 或 **DumpAssembly** 命令检索模块的地址。
DumpObj [**-nofields**] <*object address*> - 或 - **DO** <*object address*>	显示有关指定地址处的对象的信息。**DumpObj** 命令显示对象的字段、`EEClass` 结构信息、方法表和大小。 可以使用 **DumpStackObjects** 命令检索对象的地址。 可以对 `CLASS` 类型的字段运行 DumpObj 命令，因为这些字段也是对象。 `-` **nofields** 选项可阻止显示对象的字段，它对 String 这样的对象很有用。
DumpRuntimeTypes	显示垃圾回收器堆中的运行时类型对象并列出其关联的类型名称和方法表。
DumpStack [**-EE**] [**-n**] [`top` *stack* [`bottom` *stac* `k`]]	显示堆栈跟踪。 **-EE** 选项使 **DumpStack** 命令仅显示托管函数。使用 `top` 和 `bottom` 参数可限制 x86 平台上显示的堆栈帧。 **-n** 选项禁止显示源文件名和行号。如果调试器已指定选项 SYMOPT_LOAD_LINES，则 SOS 将查找每个托管帧的符号，如果成功，则将显示对应的源文件名和行号。可以指定 **-n**（无行号）参数来禁用此行为。

COMMAND	ㄖ
DumpSig <*sigaddr*> <*moduleaddr*>	显示有关指定地址处的 `Sig` 结构的信息。
DumpSigElem <*sigaddr*> <*moduleaddr*>	显示签名对象的单个元素。大多数情况下，应使用 **DumpSig** 查看单个签名对象。但是，如果签名已在某种程度上被损坏，则可使用 **DumpSigElem** 读取它的有效部分。
DumpStackObjects [**-verify**] [`top` *stack* [`bottom` *stack*]] - 或 - **DSO** [**-verify**] [`top` *stack* [`bottom` *stack*]]	显示在当前堆栈的边界内找到的所有托管对象。 **-verify** 选项验证对象字段的每个非静态 `CLASS` 字段。 将 DumpStackObject 命令与堆栈跟踪命令（如 K (windbg) 或 bt (lldb)）及 clrstack 命令一起使用，以确定局部变量和参数的值 。
DumpVC <*MethodTable address*> <*Address*>	显示有关指定地址处的值类字段的信息。 **MethodTable** 参数使 **DumpVC** 命令可以正确解释字段。值类不将方法表作为其第一个字段。
EEHeap [**-gc**] [**-loader**]	显示有关内部运行时数据结构所使用的进程内存的信息。 **-gc** 和 **-loader** 选项将此命令的输出限制为垃圾回收器或加载程序数据结构。 有关垃圾回收器的信息列出了托管堆中每个段的范围。如果指针落在由 **-gc** 给定的段范围内，则该指针是一个对象指针。
EEStack [**-short**] [**-EE**]	对一个进程中的所有线程运行 **DumpStack** 命令。 将 **-EE** 选项直接传递给 **DumpStack** 命令。**-short** 参数将输出限制为以下类型的线程： 已获取锁的线程。 已停止运行以允许垃圾回收的线程。 当前在托管代码中的线程。
EHInfo [<*MethodDesc address*>] [<*Code address*>]	显示指定方法中的异常处理块。此命令显示子句块（`try` 块）和处理程序块（`catch` 块）的代码地址和偏移量。
▬	显示常见问题。`dotnet-dump` 不支持。

COMMAND	此
FinalizeQueue [**-detail**] | [**-allReady**] [**-short**]	显示所有已进行终结注册的对象。 **-detail** 选项显示有关需要清理的任何 `SyncBlocks` 的额外信息，以及有关等待清理的任何 `RuntimeCallableWrappers` (RCW) 的额外信息。这两种数据结构都由终结器线程在运行时进行缓存和清理。 `-allReady` 选项显示所有准备终止的对象，无论它们已被垃圾回收标记成这样，还是将被下一个垃圾回收标记。“准备终止”列表中的对象为不再为根的可终止对象。此选项可能耗费大量资源，因为它验证可终止队列中的所有对象是否仍然为根对象。 `-short` 选项将输出限制为每个对象的地址。如果将此选项与 **-allReady** 一起使用，则将枚举具有不再为根的终结器的所有对象。如果单独使用此选项，则将列出终结和“准备终结”队列中的所有对象。
FindAppDomain <*Object address*>	确定指定地址处的对象的应用程序域。
FindRoots -gen <*N*> | **-gen any** |<*object address*>	导致调试器在指定生成的下次回收时在调试对象中中断。在中断发生时立即重置该效果。若要在下一个集合中断，你必须重新发出该命令。此命令的 *<object address>* 形式在 -gen 或 -gen any 引起的中断发生后使用。此时，调试对象处于正确的状态，以便 **FindRoots** 从当前已报废的生成中识别出对象的根。只有在 Windows 上才受支持。
GCHandles [**-perdomain**]	显示有关进程中的垃圾回收器句柄的统计信息。 **-perdomain** 选项将按应用程序域排列统计信息。 使用 **GCHandles** 命令可查找由垃圾回收器句柄泄漏导致的内存泄漏。例如，当代码由于强垃圾回收器句柄仍指向一个大型数组而保留该数组时，若不释放句柄就将其放弃，则会发生内存泄漏。 只有在 Windows 上才受支持。
GCHandleLeaks	在内存中搜索进程中的对强垃圾回收器句柄和固定垃圾回收器句柄的任何引用并显示结果。如果找到某个句柄，**GCHandleLeaks** 命令将显示相应的引用地址。如果在内存中找不到句柄，此命令将显示一个通知。只有在 Windows 上才受支持。
GCInfo <*MethodDesc address*> <*Code address*>	显示指示寄存器或堆栈位置何时包含托管对象的数据。如果发生垃圾回收，回收器必须知道对象引用的位置，以便可以使用新的对象指针值更新相应的对象引用。

COMMAND	𝐈𝐈
GCRoot [**-nostacks**] [**-all**] <*Object address*>	显示有关对指定地址处的对象的引用(或根)的信息。 **GCRoot** 命令将检查整个托管堆和句柄表以查找其他对象内的句柄和堆栈上的句柄。然后，在每个堆栈中搜索对象的指针，同时还搜索终结器队列。 此命令无法确定堆栈根是有效的还是已丢弃。使用 clrstack 和 U 命令可对本地或自变量值所属的帧进行反汇编，以便确定堆栈根是否仍在使用中 。 -nostacks 选项将搜索限制为垃圾回收器句柄和 reachable 对象。 "-all"选项强制显示所有根，而不仅仅是唯一的根。
GCWhere *<object address>*	显示垃圾回收堆中自变量传入的位置和大小。如果自变量位于托管堆中，但不是有效的对象地址，则大小显示为 0(零)。
Help (**soshelp**) [<*command*>] [`faq`]	在未指定参数时显示所有可用命令，或者显示有关指定命令的详细帮助信息。 `faq` 参数显示常见问题的答案。
HeapStat [**-inclUnrooted** | **-iu**]	显示每个堆的生成大小及每个堆上各生成的可用空间总量。如果指定 **-inclUnrooted** 选项，则报告将包括有关不再为根的垃圾回收堆中的托管对象的信息。只有在 Windows 上才受支持。
HistClear	释放由 `Hist` 命令系列使用的任何资源。 通常，你不必显式调用 `HistClear`，因为每个 `HistInit` 都会清理以前的资源。
HistInit	从保存在调试对象中的压力日志初始化 SOS 结构。
HistObj *<obj_address>*	检查所有压力日志的重定位记录，并显示可能已将地址作为自变量传入的垃圾回收重定位链。
HistObjFind *<obj_address>*	显示在指定地址处引用对象的所有日志项。
HistRoot *<root>*	显示与指定根的提升和重定位相关的信息。 根值可用于通过垃圾回收来跟踪对象的移动。
IP2MD (**ip2md**) <*Code address*>	显示已 JIT 编译的代码中指定地址处的 `MethodDesc` 结构。
ListNearObj (**lno**) *<obj_address>*	显示指定地址之前和之后的对象。该命令在垃圾回收堆和自变量地址之后的对象中寻找地址，而该垃圾回收堆看上去像托管对象(基于有效的方法表)的有效开头。只有在 Windows 上才受支持。

COMMAND	此
MinidumpMode [**0**] [**1**]	防止在使用小型转储时运行不安全的命令。 传递 **0** 禁用此功能，或传递 **1** 启用此功能。默认情况下，**MinidumpMode** 值设置为 **0**。 使用 **.dump /m** 命令或 **.dump** 命令创建的小型转储具有有限的 CLR 特定数据，允许只正确地运行一小部分 SOS 命令。有些命令可能会因错误而失败，因为所需的内存区域未被映射或仅被部分映射。此选项可防止你对小型转储运行不安全的命令。 仅在 Windbg 中受支持。
Name2EE (**name2ee**) <*module name*> <*type or method name*> - 或 - **Name2EE** <*module name*> **!** <*type or method name*>	显示指定模块中的指定类型或方法的 `MethodTable` 结构和 `EEClass` 结构。 在进程中必须加载指定的模块。 若要获取正确的类型名称，请通过使用 Ildasm.exe（IL 反汇编程序）浏览模块。也可以将 `*` 作为模块名参数传递以搜索所有已加载的托管模块。*module name* 参数也可以是模块的调试器名称，如 `mscorlib` 或 `image00400000`。 此命令支持 <`module`>`!`<`type`> 的 Windows 调试器语法。类型必须是完全限定的。
ObjSize [<*Object address*>]	[**-aggregate**] [**-stat**]	显示指定对象的大小。如果未指定任何参数，则 **ObjSize** 命令将显示在托管线程上找到的所有对象的大小，显示进程中的所有垃圾回收器句柄，并对这些句柄指向的任何对象的大小进行合计。除父对象之外，**ObjSize** 命令还包括所有子对象的大小。 **-aggregate** 选项能够与 **-stat** 参数一起使用，以获得仍为根类型的类型的详细视图。通过使用 **!dumpheap -stat** 和 **!objsize -aggregate -stat**，可以确定不再为根的对象并诊断各种内存问题。 只有在 Windows 上才受支持。
PrintException [**-nested**] [**-lines**] [<*Exception object address*>] - 或 - **PE** [**-nested**] [<*Exception object address*>]	显示从指定地址处的 Exception 类派生的任何对象的字段并设置这些字段的格式。如果不指定地址，**PrintException** 命令将显示在当前线程上引发的最后一个异常。 **-nested** 选项显示有关嵌套异常对象的详细信息。 **-lines** 选项显示源信息（如果可用）。 可以使用此命令设置 `_stackTrace` 字段的格式并查看该字段（它是一个二进制数组）。
ProcInfo [**-env**] [**-time**] [**-mem**]	显示进程的环境变量、内核 CPU 时间和内存使用统计信息。仅在 Windbg 中受支持。
RCWCleanupList <*RCWCleanupList address*>	显示在指定地址处等待清理的运行时可调用包装器的列表。仅在 Windbg 中受支持。
SaveModule <*Base address*> <*Filename*>	将加载到内存中指定地址的图像写入指定文件。仅在 Windbg 中受支持。

COMMAND	此

SetHostRuntime [<runtime-directory>]	此命令会设置 .NET Core 运行时的路径，以用于托管在调试器 (lldb) 中作为 SOS 的一部分运行的托管代码。运行时至少需要为 2.1.0 或更高版本。如果目录中有空格，则需要使用单引号 (') 引起来。
	
	通常，SOS 会尝试查找已安装的 .NET Core 运行时来自动运行其托管代码，但如果失败，则可用此命令。默认情况下使用正进行调试的同一运行时 (libcoreclr)。如果正进行调试的默认运行时无法有效运行 SOS 代码或其版本低于 2.1.0，请使用此命令。
	
	如果在运行 SOS 命令时收到以下错误消息，请使用此命令将路径设置为 2.1.0 或更高版本的 .NET Core 运行时。
	
	`(lldb) clrstack`
	`Error: Fail to initialize CoreCLR 80004005 ClrStack failed`
	
	`(lldb) sethostruntime /usr/share/dotnet/shared/Microsoft.NETCore.App/2.1.6`
	
	可以在命令行界面中使用“dotnet --info”来查找已安装 .NET Core 运行时的路径。

SetSymbolServer [**-ms**] [**-disable**] [**-log**] [**-loadsymbols**] [**-cache** <cache-path>] [**-directory** <search-directory>] [**-sympath** <windows-symbol-path>] [<symbol-server-URL>]	启用符号服务器下载支持。 “-ms”选项允许从公共 Microsoft 符号服务器下载。 “-disable”选项启用符号下载支持。 -cache <cache-path> 选项指定符号缓存目录。如果未指定，则默认为 $HOME/.dotnet/symbolcache。 “-directory”选项添加路径来搜索符号。可以有多个。 “-sympath”选项以 Windows 符号路径格式添加服务器、缓存和目录路径。 “-log”选项启用符号下载日志记录。 “-loadsymbols”选项尝试下载运行时的原生 .NET Core 符号。lldb 和 dotnet-dump 上受支持
SOSFlush	刷新内部 SOS 缓存。
SOSStatus [**-reset**]	显示内部 SOS 状态或重置内部缓存状态。
StopOnException [**-derived**] [**-create** | **-create2**] *<Exception> <Pseudo-register number>*	使调试器在引发指定异常时停止，但在引发其他异常时继续运行。 **-derived** 选项用于捕获指定异常以及从指定异常派生的每个异常。 仅在 Windbg 中受支持。

COMMAND	此
SyncBlk [**-all** | <*syncblk number*>]	显示指定的 `SyncBlock` 结构或所有 `SyncBlock` 结构。如果不传递任何自变量，**SyncBlk** 命令将显示与一个线程拥有的对象对应的 `SyncBlock` 结构。 `SyncBlock` 结构是一个容器，用于存放无需为每个对象创建的额外信息。它可以存放 COM 互操作数据、哈希代码和用于线程安全操作的锁定信息。
ThreadPool	显示有关托管线程池的信息，包括队列中工作请求的数目、完成端口线程的数目和计时器的数目。
Threads (**clrthreads**) [**-live**] [**-special**]	显示进程中的所有托管线程。 Threads 命令显示调试程序速记 ID、CLR 线程 ID 以及操作系统线程 ID。此外，**Threads** 命令还会显示“Domain”列、“APT”列和“Exception”列，这三列分别用于指示正在执行某线程的应用程序域、显示 COM 单元模式以及显示该线程中引发的上一个异常。 **-live** 选项显示与活动线程关联的线程。 选项显示由 CLR 创建的所有特殊线程。特殊线程包括垃圾回收线程(在并发垃圾回收和服务器垃圾回收中)、调试器帮助程序线程、终结器线程、AppDomain 卸载线程和线程池计时器线程。
ThreadState < *State value field* >	显示线程的状态。`value` 参数为 **Threads** 报告输出中的 `State` 字段的值。
Token2EE <*module name*> <*token*>	将指定模块中的指定元数据标记转换成 `MethodTable` 结构或 `MethodDesc` 结构。 可以传递 `*` 作为模块名参数，以便在每个已加载的托管模块中查找该标记映射到的内容。也可以传递某个模块的调试器名称，如 `mscorlib` 或 `image00400000`。
U [**-gcinfo**] [**-ehinfo**] [**-n**] <*MethodDesc address*> | <*Code address*>	显示由方法的 `MethodDesc` 结构指针或方法体内的代码地址指定的托管方法的反汇编(带有批注)。**U** 命令将从开始到结束显示整个方法，并带有将元数据标记转换为名称的批注。 **-gcinfo** 选项使 **U** 命令显示方法的 `GCInfo` 结构。 **-ehinfo** 选项显示方法的异常信息。也可以使用 **EHInfo** 命令获取此信息。 **-n** 选项禁止显示源文件名和行号。如果调试器已指定选项 SYMOPT_LOAD_LINES，则 SOS 将查找每个托管帧的符号，如果成功，则将显示对应的源文件名和行号。可以指定 **-n** 选项来禁用此行为。
VerifyHeap	检查垃圾回收器堆中是否有损坏迹象，并显示找到的任何错误。 错误构造的平台调用可能导致堆损坏。
VerifyObj <*object address*>	检查作为自变量传递的对象是否有损坏迹象。只有在 Windows 上才受支持。

COMMAND	说
VMMap	遍历虚拟地址空间并显示应用于每个区域的保护类型。仅在 Windbg 中受支持。
VMStat	提供虚拟地址空间的摘要视图，并按应用于该内存的每种保护类型（可用、已保留、已提交、私有、已映射和映像）排序。“TOTAL”列显示“AVERAGE”列乘以“BLK COUNT”列的结果。仅在 Windbg 中受支持。

Dotnet-Dump

有关 `dotnet-dump analyze` 的可用 SOS 命令的列表，请参阅 dotnet-dump。

Windows 调试器

还可以通过将 SOS 调试扩展加载到 WinDbg/dbg 调试程序中并在 Windows 调试程序中执行命令来使用此扩展。可对实时进程或转储使用 SOS 命令。

若要将 SOS 调试扩展加载到 Windows 调试程序中，请在工具中运行以下命令：

```
.loadby sos clr
```

WinDbg.exe 和 Visual Studio 使用与当前使用的 Mscorwks.dll 版本对应的 SOS.dll 版本。默认情况下，应使用与当前版本的 Mscorwks.dll 匹配的 SOS.dll 版本。

若要使用在其他计算机上创建的转储文件，请确保该安装所附带的 Mscorwks.dll 文件存在于符号路径中，并加载相应的 SOS.dll 版本。

若要加载特定版本的 SOS.dll，请在 Windows 调试程序中输入以下命令：

```
.load <full path to sos.dll>
```

LLDB 调试程序

有关针对 LLDB 配置 SOS 的说明，请参阅 dotnet-sos。可对实时进程或转储使用 SOS 命令。

默认情况下，可以通过输入以下内容来使用所有 SOS 命令：`sos [command\_name]`。但是，常见命令已有别名，因此你不需要 `sos` 前缀：

说	说
`bpmd`	在指定模块中的指定托管方法处创建断点。
`clrstack`	仅提供托管代码的堆栈跟踪。
`clrthreads`	列出正在运行的托管线程。
`clru`	显示托管方法的批注反汇编。
`dso`	显示在当前堆栈的边界内找到的所有托管对象。
`dumpasync`	显示有关垃圾回收堆上异步状态机的信息。
`dumpclass`	显示有关指定地址处的 `EEClass` 结构的信息。

“	“
`dumpdomain`	显示所有 AppDomain 和指定域中的所有程序集的信息。
`dumpheap`	显示有关垃圾回收堆的信息和有关对象的收集统计信息。
`dumpil`	显示与托管方法关联的 Microsoft 中间语言 (MSIL)。
`dumplog`	将内存中压力日志的内容写入到指定文件。
`dumpmd`	显示有关指定地址处的 `MethodDesc` 结构的信息。
`dumpmodule`	显示有关指定地址处的模块的信息。
`dumpmt`	显示有关指定地址处的方法表的信息。
`dumpobj`	显示有关指定地址处的对象的信息。
`dumpstack`	显示原生和托管堆栈跟踪。
`eeheap`	显示有关内部运行时数据结构所使用的进程内存的信息。
`eestack`	对进程中的所有线程运行 `dumpstack` 命令。
`gcroot`	显示有关对指定地址处的对象的引用(或根)的信息。
`histclear`	释放由 Hist 命令系列使用的任何资源。
`histinit`	从保存在调试对象中的压力日志初始化 SOS 结构。
`histobj`	检查所有压力日志的重定位记录，并显示可能已将地址作为自变量传入的垃圾回收重定位链。
`histobjfind`	显示在指定地址处引用对象的所有日志项。
`histroot`	显示与指定根的提升和重定位相关的信息。
`ip2md`	显示已 JIT 编译的代码中指定地址处的 `MethodDesc` 结构。
`loadsymbols`	加载 .NET Core 原生模块符号。
`name2ee`	显示指定模块中的指定类型或方法的 `MethodTable` 和 `EEClass` 结构。
`pe`	显示从指定地址处的 Exception 类派生的任何对象的字段并设置这些字段的格式。
`setclrpath`	设置用于加载 coreclr dac/dbi 文件的路径。 `setclrpath <path>`

`sethostruntime`	设置或显示用于在 SOS 中运行托管代码的 .NET Core 运行时目录。
`setsymbolserver`	启用符号服务器支持。
`setsostid`	设置当前 OS tid/thread 索引，而不是使用 lldb 提供的索引。 `setsostid <tid> <index>`
`sos`	各种 coreclr 调试命令。有关详细信息，请参阅“soshelp”。 `sos <command-name> <args>`
`soshelp`	在未指定参数时显示所有可用命令，或者显示有关指定命令的详细帮助信息：`soshelp <command>`
`syncblk`	显示 SyncBlock 持有者信息。

Windbg/cdb 示例用法

COMMAND	
`!dumparray -start 2 -length 5 -detail 00ad28d0`	显示地址 `00ad28d0` 处的数组内容。显示从第二个元素开始，连续显示五个元素。
`!dumpassembly 1ca248`	显示地址 `1ca248` 处的程序集内容。
`!dumpheap`	显示有关垃圾回收器堆的信息。
`!DumpLog`	将内存中压力日志的内容写入到当前目录中名为 StressLog.txt 的（默认）文件中。
`!dumpmd 902f40`	显示在地址 `MethodDesc` 处的 `902f40` 结构。
`!dumpmodule 1caa50`	显示有关地址 `1caa50` 处的模块的信息。
`!DumpObj a79d40`	显示有关地址 `a79d40` 处的对象的信息。
`!DumpVC 0090320c 00a79d9c`	使用地址 `00a79d9c` 处的方法表显示地址 `0090320c` 处的值类的字段。
`!eeheap` -gc	显示垃圾回收器所使用的进程内存。
`!finalizequeue`	显示所有已做好终结计划的对象。
`!findappdomain 00a79d98`	确定地址 `00a79d98` 处的对象的应用程序域。
`!gcinfo 5b68dbb8`	显示当前进程中的所有垃圾回收器句柄。
`!name2ee unittest.exe MainClass.Main`	显示模块 `unittest.exe` 的类 `MainClass` 中的 `Main` 方法的 `MethodTable` 和 `EEClass` 结构。

COMMAND	说明
`!token2ee unittest.exe 02000003`	显示有关模块 `unittest.exe` 中的地址 `02000003` 处的元数据标记的信息。

LLDB 示例用法

COMMAND	说明
`sos DumpArray -start 2 -length 5 -detail 00ad28d0`	显示地址 `00ad28d0` 处的数组内容。显示从第二个元素开始，连续显示五个元素。
`sos DumpAssembly 1ca248`	显示地址 `1ca248` 处的程序集内容。
`dumpheap`	显示有关垃圾回收器堆的信息。
`dumplog`	将内存中压力日志的内容写入到当前目录中名为 StressLog.txt 的(默认)文件中。
`dumpmd 902f40`	显示在地址 `MethodDesc` 处的 `902f40` 结构。
`dumpmodule 1caa50`	显示有关地址 `1caa50` 处的模块的信息。
`dumpobj a79d40`	显示有关地址 `a79d40` 处的对象的信息。
`sos DumpVC 0090320c 00a79d9c`	使用地址 `00a79d9c` 处的方法表显示地址 `0090320c` 处的值类的字段。
`eeheap -gc`	显示垃圾回收器所使用的进程内存。
`sos FindAppDomain 00a79d98`	确定地址 `00a79d98` 处的对象的应用程序域。
`sos GCInfo 5b68dbb8`	显示当前进程中的所有垃圾回收器句柄。
`name2ee unittest.exe MainClass.Main`	显示模块 `unittest.exe` 的类 `MainClass` 中的 `Main` 方法的 `MethodTable` 和 `EEClass` 结构。
`sos Token2EE unittest.exe 02000003`	显示有关模块 `unittest.exe` 中的地址 `02000003` 处的元数据标记的信息。
`clrthreads`	显示托管线程。

另请参阅

- .NET 中的转储简介
- 了解如何调试 .NET Core 中的内存泄漏
- 收集和分析内存转储博客
- 转储分析工具 (dotnet-dump)
- SOS 安装工具 (dotnet-sos)
- 堆分析工具 (dotnet-gcdump)

EventPipe

2021/11/16 •

EventPipe 是类似于 ETW 或 LTTng 的运行时组件，可用于收集跟踪数据。EventPipe 的目标是使 .NET 开发人员能够轻松地跟踪其 .NET 应用程序，而无需依赖于平台特定的 OS 本机组件（如 ETW 或 LTTng）。

EventPipe 是众多诊断工具背后的一种机制，可用于接收运行时发出的事件以及通过 EventSource 编写的自定义事件。

本文是对 EventPipe 的简要概述。文章说明何时以及如何使用 EventPipe，以及如何对其进行配置以最大程度地满足你的需求。

EventPipe 基础知识

EventPipe 聚合由运行时组件发出的事件（例如实时编译器或垃圾回收器）以及从库和用户代码中的 EventSource 实例编写的事件。

然后，将这些事件序列化，并直接写入文件或通过诊断端口在进程外使用。在 Windows 上，诊断端口作为 `NamedPipe` 实现。在非 Windows 平台（如 Linux 或 macOS）上，该端口可使用 Unix 域套接字实现。有关诊断端口以及如何通过其自定义进程内通信协议与之交互的详细信息，请参阅诊断 IPC 协议文档。

然后，EventPipe 以 `.nettrace` 文件格式写入序列化事件，可作为流通过诊断端口写入，也可直接写入文件。若要了解有关 EventPipe 序列化格式的详细信息，请参阅 EventPipe 格式文档。

EventPipe 与ETW/LTTng

EventPipe 是 .NET 运行时 (CoreCLR) 的一部分，旨在跨 .NET Core 支持的所有平台以相同的方式工作。这将允许基于 EventPipe 的跟踪工具（例如 `dotnet-counters`、`dotnet-gcdump` 和 `dotnet-trace`）无缝地跨平台工作。

但是，因为 EventPipe 是一个运行时内置组件，所以它的作用域仅限于托管代码和运行时本身。EventPipe 不能用于跟踪一些较低级别的事件，例如，解析本机代码堆栈或获取各种内核事件。如果在应用中使用 C/C++ 互操作或者要跟踪运行时本身（使用 C++ 编写的），或者要更深入地诊断需要内核事件（即本机线程上下文切换事件）的应用的行为，则应使用 ETW 或 Perf/LTTng。

EventPipe 和 ETW/LTTng 之间的另一个主要区别是管理员/根用户权限要求。若要使用 ETW 或 LTTng 跟踪应用程序，你需要是管理员/根用户。可使用 EventPipe 跟踪应用程序，前提是跟踪器（如 `dotnet-trace`）是由启动该应用程序的同一用户运行的。

下表汇总了 EventPipe 和 ETW/LTTng 之间的差异。

🔗	EVENTPIPE	ETW	LTTNG
跨平台	是	否（仅在 Windows 上）	否（仅在受支持的 Linux 发行版上）
需要管理员/根用户权限	否	是	是
可获取 OS/内核事件	否	是	是
可解析本机调用堆栈	否	是	是

使用 EventPipe 跟踪 .NET 应用程序

可通过多种方式使用 EventPipe 跟踪 .NET 应用程序：

- 使用基于 EventPipe 构建的诊断工具之一。
- 使用 Microsoft.Diagnostics.NETCore.Client 库来编写自己的工具，以配置和启动 EventPipe 会话。
- 使用环境变量来启动 EventPipe。

生成包含 EventPipe 事件的 `nettrace` 文件之后，可在 `PerfView` 或 Visual Studio 中查看该文件。在非 Windows 平台上，可使用 `dotnet-trace convert` 命令将 `nettrace` 文件转换为 `speedscope` 或 `Chromium` 跟踪格式，并使用 `speedscope` 或 Chrome DevTools 查看该文件。

还以通过 TraceEvent 以编程方式分析 EventPipe 跟踪。

使用 EventPipe 的工具

这是使用 EventPipe 跟踪应用的最简单方法。若要详细了解如何使用这些工具，请参阅每个工具的文档。

- dotnet-counters 使你能够监视和收集由 .NET 运行时和核心库发出的各种指标，以及可以编写的自定义指标。
- dotnet-gcdump 使你能够收集实时进程的 GC 堆转储以分析应用程序的托管堆。
- dotnet-trace 使你能够收集应用程序的跟踪以进行性能分析。

使用环境变量进行跟踪

使用 EventPipe 的首选机制是使用 `dotnet-trace` 或 `Microsoft.Diagnostics.NETCore.Client` 库。

但是，可使用以下环境变量在应用上设置 EventPipe 会话，并使其将跟踪直接写入到文件。若要停止跟踪，请退出应用程序。

- `DOTNET_EnableEventPipe`：将此值设置为 `1` 以启动直接写入到文件的 EventPipe 会话。默认值为 `0`。
- `DOTNET_EventPipeOutputPath`：输出 EventPipe 跟踪文件（配置为通过 `DOTNET_EnableEventPipe` 运行时）的路径。默认值为 `trace.nettrace`，将在运行应用的同一目录中创建该默认值。

  > **NOTE**
  > 自 .NET 6 起，`DOTNET_EventPipeOutputPath` 中的 `{pid}` 字符串实例将替换为所跟踪的进程的进程 ID。

- `DOTNET_EventPipeCircularMB`：一个十六进制值，它表示 EventPipe 的内部缓冲区大小（以 MB 为单位）。仅当 EventPipe 配置为通过 `DOTNET_EnableEventPipe` 运行时，才使用此配置值。默认缓冲区大小为 1024 MB，而由于 `0x400` == `1024`，它转换为设置成 `400` 的环境变量。

  > **NOTE**
  > 如果目标进程过于频繁地写入事件，则它可能会溢出此缓冲区，并且某些事件可能会被丢弃。如果丢弃的事件过多，请增加缓冲区大小，查看丢弃的事件数是否减少。如果丢弃的事件数未随缓冲区大小的增加而减少，则可能是因为读取器的速度较慢，导致无法刷新目标进程的缓冲区。

- `DOTNET_EventPipeProcNumbers`：将此项设置为 `1`，以在 EventPipe 事件标头中捕获处理器数。默认值为 `0`。
- `DOTNET_EventPipeConfig`：使用 `DOTNET_EnableEventPipe` 启动 EventPipe 会话时设置 EventPipe 会话配置。语法如下：

`<provider>:<keyword>:<level>`

还可通过使用逗号连接多个提供程序来指定它们：

`<provider1>:<keyword1>:<level1>,<provider2>:<keyword2>:<level2>`

如果未设置此环境变量但通过 `DOTNET_EnableEventPipe` 启用了 EventPipe，则会通过使用以下关键字和级别启用以下提供程序来启动跟踪：

- `Microsoft-Windows-DotNETRuntime:4c14fccbd:5`
- `Microsoft-Windows-DotNETRuntimePrivate:4002000b:5`
- `Microsoft-DotNETCore-SampleProfiler:0:5`

若要详细了解 .NET 中的一些已知提供程序，请参阅已知事件提供程序。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

.NET Core 日志记录和跟踪

2021/11/16 •

日志记录和跟踪实际上是同一技术的两个名称。这种简单的技术从计算机早期就开始使用了。它只涉及检测应用程序以写入稍后要使用的输出。

使用日志记录和跟踪的原因

这一简单技术功能非常强大。可以在调试器失败的情况下使用它：

- 很难使用传统调试器来调试在很长一段时间内发生的问题。借助日志，可以跨在较长时间进行详细的事后检查。与此相反，调试器限制为只能进行实时分析。
- 多线程应用程序和分布式应用程序通常难以调试。附加调试器往往会修改行为。可以根据需要分析详细日志，以了解复杂的系统。
- 分布式应用程序中的问题可能源于许多组件之间的复杂交互，将调试器连接到系统的每个部分可能是不合理的。
- 许多服务不应停止。附加调试器往往会导致超时失败。
- 问题并非总是可预见的。日志记录和跟踪旨在降低开销，以便在出现问题的情况下可以始终记录程序。

.NET Core API

打印样式 API

每个 System.Console、System.Diagnostics.Trace 和 System.Diagnostics.Debug 类都提供类似的打印样式 API，便于日志记录。

选择使用哪种打印样式 API 由用户自己决定。主要区别包括：

- System.Console
  - 始终启用，并始终写入控制台。
  - 这对于客户可能需要在版本中查看的信息非常有用。
  - 由于这是最简单的方法，所以常常用于临时调试。此调试代码通常不会签入到源代码管理中。
- System.Diagnostics.Trace
  - 仅在通过向源中添加 `#define TRACE` 或在编译时指定选项 `/d:TRACE` 来定义 `TRACE` 时启用。
  - 写入到附加 Listeners，默认情况下为 DefaultTraceListener。
  - 创建将在大多数生成中启用的日志时使用此 API。
- System.Diagnostics.Debug
  - 仅在通过向源中添加 `#define DEBUG` 或在编译时指定选项 `/d:DEBUG` 来定义 `DEBUG` 时启用。
  - 写入附加调试器。
  - 在 `*nix` 上，如果设置了 `DOTNET_DebugWriteToStdErr` 或 `COMPlus_DebugWriteToStdErr`，则写入 stderr。
  - 创建将仅在调试生成中启用的日志时使用此 API。

记录事件

以下 API 更面向于事件。它们记录事件对象，而不是记录简单字符串。

- System.Diagnostics.Tracing.EventSource
  - EventSource 是主根 .NET Core 跟踪 API。
  - 在所有 .NET Standard 版本中提供。
  - 仅允许跟踪可序列化的对象。

    - 可以通过配置为使用 EventSource 的任何 EventListener 实例在进程中使用。
    - 可通过以下方式在进程外使用：
        - 所有平台上的 .NET Core EventPipe
        - Windows 事件跟踪 (ETW)
        - 适用于 Linux 的 LTTng 跟踪框架
            - 演练：使用 PerfCollect 收集 LTTng 跟踪。
- System.Diagnostics.DiagnosticSource
    - 包含在 .NET Core 中，用作 .NET Framework 的 NuGet 包。
    - 允许对非序列化对象进行进程内跟踪。
    - 包括一个桥，以允许将已记录对象的所选字段写入 EventSource。
- System.Diagnostics.EventLog
    - 仅限 Windows。
    - 将消息写入 Windows 事件日志。
    - 系统管理员希望严重的应用程序错误消息会在 Windows 事件日志中出现。

分布式跟踪

分布式跟踪是一种诊断技术，可帮助工程师找出应用程序中的故障和性能问题，尤其是那些可能跨多个计算机或进程分布的问题。此技术通过应用程序跟踪请求，将不同应用程序组件完成的工作关联在一起，并将其与应用程序可能为并发请求所做的其他工作分开。

ILogger 和日志记录框架

低级别 API 可能不符合你的日志记录需求。可能需要考虑使用日志记录框架。

ILogger 接口已用于创建一个公共日志记录接口，可在其中通过依赖项注入插入记录器。

例如，为了使你能够为应用程序做出最佳选择，.NET 提供了对选择的内置和第三方框架的支持：

- .NET 内置日志记录提供程序
- .NET 第三方日志记录提供程序

与日志记录相关的引用

- 如何：使用跟踪和调试进行条件编译
- 如何：向应用程序代码添加跟踪语句
- .NET 中的日志记录提供它所支持的日志记录技术的概述。
- C# 字符串内插可以简化日志记录代码的编写。
- 运行时提供程序事件列表
- .NET 中的已知事件提供程序
- Exception.Message 属性对日志记录异常很有用。
- System.Diagnostics.StackTrace 类可用于在日志中提供堆栈信息。

性能注意事项

字符串格式设置可能会占用大量的 CPU 处理时间。

在性能关键应用程序中，建议执行以下操作：

- 如果未侦听，则避免进行大量日志记录。通过检查是否首先启用了日志记录，避免构造开销较大的日志记录消息。
- 仅记录有用的内容。
- 将复杂的格式设置推迟到分析阶段。

.NET 中的已知事件提供程序

2021/11/16 •

.NET 运行时和库通过许多不同的事件提供程序编写诊断事件。根据诊断需求，可以选择要启用的相应提供程序。本文介绍了 .NET 运行时和库中最常用的一些事件提供程序。

CoreCLR

“Microsoft-Windows-DotNETRuntime”提供程序

此提供程序从 .NET 运行时发出各种事件，包括 GC、加载程序、JIT、异常和其他事件。请在运行时提供程序事件列表中详细了解此提供程序中的各种事件。

“Microsoft-DotNETCore-SampleProfiler”提供程序

此提供程序是 .NET 运行时事件提供程序，用于对托管调用堆栈进行 CPU 采样。启用后，它会每隔 10 毫秒捕获一次每个线程的托管调用堆栈的快照。若要启用此捕获功能，必须将 EventLevel 指定为 `Informational` 或更高级别。

框架库

“Microsoft-Extensions-DependencyInjection”提供程序

此提供程序记录来自 DependencyInjection 的信息。下表显示 `Microsoft-Extensions-DependencyInjection` 提供程序记录的事件：

ıııı	ııı	LEVEL	ıı
`CallSiteBuilt`		详细级别 (5)	调用站点已生成。
`ServiceResolved`		详细级别 (5)	服务已解决。
`ExpressionTreeGenerated`		详细级别 (5)	表达式树已生成。
`DynamicMethodBuilt`		详细级别 (5)	DynamicMethod 已生成。
`ScopeDisposed`		详细级别 (5)	范围已释放。
`ServiceRealizationFailed`		详细级别 (5)	服务实现已失败。
`ServiceProviderBuilt`	`ServiceProviderInitialized(0x1)`	详细级别 (5)	ServiceProvider 已生成。
`ServiceProviderDescriptors`	`ServiceProviderInitialized(0x1)`	详细级别 (5)	在 ServiceProvider 生成过程中使用的 ServiceDescriptor 列表。

“System.Buffers.ArrayPoolEventSource”提供程序

此提供程序记录来自 ArrayPool 的信息。下表显示了 `ArrayPoolEventSource` 记录的事件：

[illegible]	LEVEL	[illegible]
`BufferRented`	详细级别 (5)	缓冲区已成功租用。
`BufferAllocated`	信息性 (4)	缓冲区由池分配。
`BufferReturned`	详细级别 (5)	缓冲区返回到池中。
`BufferTrimmed`	信息性 (4)	由于内存不足或不活动，试图释放缓冲区。
`BufferTrimPoll`	信息性 (4)	正在检查以剪裁缓冲区。

“System.Net.Http”提供程序

此提供程序记录来自 HTTP 堆栈的信息。下表显示了 `System.Net.Http` 提供程序记录的事件：

[illegible]	LEVEL	[illegible]
RequestStart	信息性 (4)	HTTP 请求已启动。
RequestStop	信息性 (4)	HTTP 请求已完成。
RequestFailed	错误 (2)	HTTP 请求失败。
ConnectionEstablished	信息性 (4)	HTTP 连接已建立。
ConnectionClosed	信息性 (4)	HTTP 连接已关闭。
RequestLeftQueue	信息性 (4)	HTTP 请求已离开请求队列。
RequestHeadersStart	信息性 (4)	针对标头的 HTTP 请求已启动。
RequestHeaderStop	信息性 (4)	针对标头的 HTTP 请求已完成。
RequestContentStart	信息性 (4)	针对内容的 HTTP 请求已启动。
RequestContentStop	信息性 (4)	针对内容的 HTTP 请求已完成。
ResponseHeadersStart	信息性 (4)	针对标头的 HTTP 响应已启动。
ResponseHeaderStop	信息性 (4)	针对标头的 HTTP 响应已完成。
ResponseContentStart	信息性 (4)	针对内容的 HTTP 响应已启动。
ResponseContentStop	信息性 (4)	针对内容的 HTTP 响应已完成。

“System.Net.NameResolution”提供程序

此提供程序记录与域名解析有关的信息。下表显示了 `System.Net.NameResolution` 记录的事件：

[illegible]	LEVEL	[illegible]
`ResolutionStart`	信息性 (4)	域名解析已启动。
`ResolutionStop`	信息性 (4)	域名解析已完成。
`ResolutionFailed`	信息性 (4)	域名解析失败。

“System.Net.Sockets”提供程序

此提供程序记录来自 Socket 的信息。下表显示了 `System.Net.Sockets` 提供程序记录的事件：

[illegible]	LEVEL	[illegible]
`ConnectStart`	信息性 (4)	尝试开始套接字连接已启动。
`ConnectStop`	信息性 (4)	尝试开始套接字连接已完成。
`ConnectFailed`	信息性 (4)	尝试开始套接字连接失败。
`AcceptStart`	信息性 (4)	尝试接受套接字连接已启动。
`AcceptStop`	信息性 (4)	尝试接受套接字连接已完成。
`AcceptFailed`	信息性 (4)	尝试接受套接字连接失败。

“System.Threading.Tasks.TplEventSource”提供程序

此提供程序记录有关任务并行库的信息，例如任务计划程序事件。下表显示了 `TplEventSource` 记录的事件：

[illegible]	[illegible]	LEVEL	[illegible]
`TaskScheduled`	`TaskTransfer` (`0x1`) `Tasks` (`0x2`)	信息性 (4)	Task 已加入任务计划程序的队列中。
`TaskStarted`	`Tasks` (`0x2`)	信息性 (4)	Task 已开始执行。
`TaskCompleted`	`TaskStops` (`0x40`)	信息性 (4)	Task 已完成执行。
`TaskWaitBegin`	`TaskTransfer` (`0x1`) `TaskWait` (`0x2`)	信息性 (4)	当对 Task 完成的隐式或显式等待启动时触发。
`TaskWaitEnd`	`Tasks` (`0x2`)	详细级别 (5)	等待 Task 完成返回时触发。
`TaskWaitContinuationStarted`	`Tasks` (`0x2`)	详细级别 (5)	当与 `TaskWaitEnd` 关联的工作(方法)启动时触发。
`TaskWaitContinuationComplete`	`TaskStops` (`0x40`)	详细级别 (5)	当与 `TaskWaitEnd` 关联的工作(方法)完成时触发。

[[[[	[[[	LEVEL	[[
`AwaitTaskContinuationSchedul`	`TaskTransfer` (`0x1`) `Tasks` (`0x2`)	信息性 (4)	在安排 Task 的异步持续时触发。

ASP.NET Core

ASP.NET Core 还提供了数种事件来帮助诊断 ASP.NET Core 堆栈中的问题。

若要详细了解 ASP.NET Core 中的事件以及如何使用这些事件，请参阅登录 .NET Core 和 ASP.NET Core。

Entity Framework Core

EF Core 也提供了事件来帮助你诊断 EF Core 中的问题。

若要详细了解 EF Core 中的事件及其使用方式，请查看 EF Core 中的 .NET 事件。

.NET 分布式跟踪

2021/11/16 •

分布式跟踪是一种诊断技术，可帮助工程师找出应用程序中的故障和性能问题，尤其是那些可能跨多个计算机或进程分布的问题。此技术通过应用程序跟踪请求，将不同应用程序组件完成的工作关联在一起，并将其与应用程序可能为并发请求所做的其他工作分开。例如，对典型 Web 服务的请求可能首先由负载均衡器接收，然后转发到 Web 服务器进程，后者随后会对数据库进行多次查询。使用分布式跟踪，工程师可以区分这些步骤中的任何一项是否失败、每个步骤所用的时间，并有可能记录每个步骤运行时生成的消息。

.NET 应用开发人员入门

关键 .NET 库经过检测，可自动生成分布式跟踪信息。但是，需要收集并存储这些信息，以供日后查看。通常，应用开发人员会选择使用遥测服务来为其存储这些跟踪信息，然后使用相应的库将分布式跟踪遥测传输到所选的服务：

- OpenTelemetry 是一个与供应商无关的库，支持多种服务。有关详细信息，请参阅使用 OpenTelemetry 收集分布式跟踪。
- Application Insights 是由 Microsoft 提供的功能齐全的服务。有关详细信息，请参阅使用 Application Insights 收集分布式跟踪。
- 有许多高质量的第三方应用程序性能监视 (APM) 供应商提供集成的 .NET 解决方案。

有关详细信息，请参阅了解分布式跟踪概念和以下指南：

- 使用自定义逻辑收集分布式跟踪
- 添加自定义分布式跟踪检测

对于第三方遥测收集服务，请按照供应商提供的设置说明进行操作。

.NET 库开发人员入门

对于 .NET 库，我们不需要关心遥测数据最终是如何收集的，而只需要关心它是如何产生的。如果希望库的使用者能够在分布式跟踪中看到库所做的详细工作，请添加分布式跟踪检测以提供支持。

有关详细信息，请参阅了解分布式跟踪概念和添加自定义分布式跟踪检测指南。

.NET 分布式跟踪概念

2021/11/16 •

分布式跟踪是一种诊断技术，可帮助工程师找出应用程序中的故障和性能问题，尤其是那些可能跨多个计算机或进程分布的问题。如需深入了解分布式跟踪的使用场景以及入门示例代码，请参阅分布式跟踪概述。

跟踪和活动

每次应用程序收到新请求时，该请求都可以与跟踪相关联。在用 .NET 编写的应用程序组件中，跟踪中的工作单元由 System.Diagnostics.Activity 实例表示，并且跟踪整体上构成了这些活动的树，可能跨越了许多不同的进程。为新请求创建的第一个活动形成跟踪树的根，并跟踪处理请求的总体持续时间和成功/失败。可以选择创建子 Activity，以将工作细分为可单独跟踪的不同步骤。例如，假设某个 Activity 跟踪了 Web 服务器中的特定入站 HTTP 请求，则可以创建子 Activity 来跟踪完成请求所需的每个数据库查询。这样做可以单独记录每个查询的持续时间和成功情况。活动可以记录每个工作单元的其他信息，例如 OperationName、称为 Tags 的名称-值对和 Events。名称标识所执行的工作类型；标记可以记录工作的描述性参数；事件是一种简单的日志记录机制，用于记录带时间戳的诊断消息。

> **NOTE**
> 分布式跟踪中工作单元的另一个常见的行业名称为“Span”。.NET 在很多年前就采用了“Activity(活动)”这个术语，当时“Span”这个名称的概念还不为人们所了解。

活动 ID

使用唯一 ID 建立分布式跟踪树中活动之间的父-子关系。.NET 分布式跟踪的实现支持两种 ID 方案：W3C 标准 TraceContext（这是 .NET 5+ 中的默认设置）和一个较早的名为“分层”的 .NET 约定（可用于向后兼容）。Activity.DefaultIdFormat 控制使用的 ID 方案。在 W3C TraceContext 标准中，每个跟踪都分配有一个全局唯一的 16 字节 trace-id (Activity.TraceId)，且该跟踪内的每个 Activity 都分配有唯一的 8 字节 span-id (Activity.SpanId)。每个活动都记录跟踪 ID、其自身的 Span ID 以及其父 (Activity.ParentSpanId) 的 Span ID。由于分布式跟踪可以跟踪跨进程边界的工作，因此父 Activity 和子 Activity 可能不在同一进程中。跟踪 ID 和父 Span ID 的组合可以在全局范围内仅标识父活动，无论该活动驻留在哪个进程中。

Activity.DefaultIdFormat 控制使用哪种 ID 格式来启动新的跟踪，但默认情况下，如果将新活动添加到现有跟踪，则会使用任何父活动所使用的格式。将 Activity.ForceDefaultIdFormat 设置为 true 会覆盖此行为，并使用 DefaultIdFormat 创建所有新的活动（即使父级使用其他 ID 格式也是如此）。

启动和停止 Activity

进程中的每个线程都有一个对应的 Activity 对象，该对象跟踪该线程上发生的工作，可通过 Activity.Current 进行访问。当前活动会自动流过线程上的所有同步调用，以及在不同线程上处理的异步调用。如果 Activity A 是线程上的当前 Activity，并且代码启动新的 Activity B，则 B 将成为该线程上新的当前 Activity。默认情况下，Activity B 还会将 Activity A 视为其父项。之后，当 Activity B 停止时，Activity A 将还原为该线程上的当前 Activity。Activity 启动时，它会将当前时间捕获为 Activity.StartTimeUtc。停止时，会将 Activity.Duration 计算为当前时间与开始时间的差值。

跨进程边界进行协调

为了跟踪跨进程边界的工作，需要在网络中传输 Activity 的父 ID，以便接收进程可以创建引用它们的 Activity。使用 W3C TraceContext ID 格式时，.NET 还将使用标准推荐的 HTTP 标头来传输此信息。使用 Hierarchical ID 格

式时，.NET 会使用自定义 request-id HTTP 标头来传输该 ID。与许多其他语言运行时不同，.NET 内置库（如 ASP.NET Web 服务器和 System.Net.Http）本身理解如何对 HTTP 消息上的 Activity ID 进行解码和编码。运行时还理解如何通过同步和异步调用流式传输 ID。这意味着，接收和发出 HTTP 消息的 .NET 应用程序会自动参与分布式跟踪 ID 的流式传输工作，应用开发人员无需进行特殊编码，也无需依赖第三方库。第三方库可以添加对通过非 HTTP 消息协议传输 ID 的支持，或者支持 HTTP 的自定义编码约定。

收集跟踪

检测代码可以创建 Activity 对象作为分布式跟踪的一部分，但需将这些对象中的信息传输到集中持久性存储中并在其中进行序列化，以便以后可以对整个跟踪进行有用的查看。有几个遥测收集库可以执行此任务，例如 Application Insights、OpenTelemetry 或第三方遥测或 APM 供应商提供的库。或者，开发人员可以使用 System.Diagnostics.ActivityListener 或 System.Diagnostics.DiagnosticListener 创作自己的自定义活动遥测收集逻辑。ActivityListener 支持观察任何 Activity，无论开发人员是否对此有先验知识。因此，ActivityListener 是一种简单而灵活的常规用途解决方案。与此相反，使用 DiagnosticListener 是一个更复杂的方案，该方案需要检测代码通过调用 DiagnosticSource.StartActivity 来选择加入，而且收集库需要知道检测代码在启动时所使用的确切命名信息。使用 DiagnosticSource 和 DiagnosticListener，创建者和侦听器可以交换任意 .NET 对象并建立自定义的信息传递约定。

采样

为了改进高吞吐量应用程序的性能，.NET 上的分布式跟踪支持仅采样跟踪的一部分，而不是记录所有跟踪。对于使用推荐的 ActivitySource.StartActivity API 创建的 Activity，遥测收集库可以使用 ActivityListener.Sample 回调来控制采样。日志记录库可以选择完全不创建活动，使用传播分布式跟踪 ID 所需的最小信息创建活动，或者用完整的诊断信息填充活动。这些选择进行了权衡，增加了性能开销以提高诊断效用。使用较旧的调用方式 Activity.Activity 和 DiagnosticSource.StartActivity 启动的活动，可以通过首先调用 DiagnosticSource.IsEnabled 来支持 DiagnosticListener 采样。即使在捕获完整的诊断信息时，.NET 实现也可以快速与高效的收集器耦合，同时，活动可以在新式硬件上以大约一微秒的时间创建、填充和传输。采样可以将每个未记录的活动的检测成本降低到小于 100 毫微秒。

后续步骤

有关在 .NET 应用程序中开始使用分布式跟踪的示例代码，请参阅分布式跟踪检测。

添加分布式跟踪检测

2021/11/16 •

本文适用范围：✔ .NET Core 2.1 及更高版本 ✔ .NET Framework 4.5 及更高版本

可以使用 System.Diagnostics.Activity API 检测 .NET 应用程序，以生成分布式跟踪遥测。标准 .NET 库内置了一些检测，但你可能想要添加更多检测，使代码更易于诊断。在本教程中，你将添加新的自定义分布式跟踪检测。请参阅集合教程，详细了解如何记录此检测生成的遥测。

先决条件

- .NET Core 2.1 SDK 或更高版本

创建初始应用

首先，将创建使用 OpenTelemetry 收集遥测，但尚未安装任何检测的示例应用。

```
dotnet new console
```

面向 .NET 5 及更高版本的应用程序已包含必要的分布式跟踪 API。对于面向早期 .NET 版本的应用，请添加 System.Diagnostics.DiagnosticSource NuGet 包版本 5 或更高版本。

```
dotnet add package System.Diagnostics.DiagnosticSource
```

添加将用于收集遥测的 OpenTelemetry 和 OpenTelemetry.Exporter.Console NuGet 包。

```
dotnet add package OpenTelemetry
dotnet add package OpenTelemetry.Exporter.Console
```

将生成的 Program.cs 内容替换为此示例源：

```
using OpenTelemetry;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using System;
using System.Threading.Tasks;

namespace Sample.DistributedTracing
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using var tracerProvider = Sdk.CreateTracerProviderBuilder()
                .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("MySample"))
                .AddSource("Sample.DistributedTracing")
                .AddConsoleExporter()
                .Build();

            await DoSomeWork("banana", 8);
            Console.WriteLine("Example work done");
        }

        // All the functions below simulate doing some arbitrary work
        static async Task DoSomeWork(string foo, int bar)
        {
            await StepOne();
            await StepTwo();
        }

        static async Task StepOne()
        {
            await Task.Delay(500);
        }

        static async Task StepTwo()
        {
            await Task.Delay(1000);
        }
    }
}
```

该应用尚无检测，因此没有要显示的跟踪信息：

```
> dotnet run
Example work done
```

最佳做法

只有应用开发人员才需要引用可选的第三方库来收集分布式跟踪遥测，例如本示例中的 OpenTelemetry。.NET 库创建者可以完全依赖于 System.Diagnostics.DiagnosticSource（.NET 运行时的一部分）中的 API。这样可确保库在各种 .NET 应用中运行，而无论应用开发人员在用于收集遥测的库或供应商方面有何偏好。

添加基本检测

应用程序和库使用 System.Diagnostics.ActivitySource 和 System.Diagnostics.Activity 类添加分布式跟踪检测。

ActivitySource

首先创建 ActivitySource 实例。ActivitySource 提供用于创建和启动 Activity 对象的 API。将 Main() 和 `using System.Diagnostics;` 上方的静态 ActivitySource 变量添加到 using 语句。

```
using OpenTelemetry;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Sample.DistributedTracing
{
    class Program
    {
        private static ActivitySource source = new ActivitySource("Sample.DistributedTracing", "1.0.0");

        static async Task Main(string[] args)
        {
            ...
```

最佳做法

- 创建一次 ActivitySource，将它存储在静态变量中，并根据需要使用相应实例。每个库或库子组件都可以（并且通常应该）创建自己的源。如果预期应用开发人员想要能够独立启用和禁用源中的 Activity 遥测，请考虑创建新源，而不是重复使用现有源。
- 传递给构造函数的源名称必须是唯一的，以免与其他任何源发生冲突。如果同一程序集内有多个源，请使用包含程序集名称和（可选）组件名称的层次结构名称，例如 `Microsoft.AspNetCore.Hosting`。如果程序集在第二个独立程序集中添加代码检测，则名称应基于定义 ActivitySource 的程序集，而不是要检测其代码的程序集。
- version 是可选参数。建议在发布库的多个版本时提供 version 并更改已检测的遥测。

> **NOTE**
>
> OpenTelemetry 使用替代术语“Tracer”和“Span”。在 .NET 中，“ActivitySource”是 Tracer 的实现，而 Activity 则是“Span”的实现。.NET 的 Activity 类型远早于 OpenTelemetry 规范，并且已保留原始 .NET 命名，以确保 .NET 生态系统内的一致性和 .NET 应用程序兼容性。

活动

使用 ActivitySource 对象，根据有意义的工作单元启动和停止 Activity 对象。使用下面显示的代码更新 DoSomeWork()：

```
        static async Task DoSomeWork(string foo, int bar)
        {
            using (Activity activity = source.StartActivity("SomeWork"))
            {
                await StepOne();
                await StepTwo();
            }
        }
```

现在运行应用会显示正在记录的新 Activity：

```
> dotnet run
Activity.Id:          00-f443e487a4998c41a6fd6fe88bae644e-5b7253de08ed474f-01
Activity.DisplayName: SomeWork
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:36:51.4720202Z
Activity.Duration:    00:00:01.5025842
Resource associated with Activity:
    service.name: MySample
    service.instance.id: 067f4bb5-a5a8-4898-a288-dec569d6dbef
```

说明

- ActivitySource.StartActivity 同时创建和启动 Activity。列出的代码模式使用的是 `using` 块，它会在执行此块后自动释放创建的 Activity 对象。释放 Activity 对象将停止它，因此代码无需显式调用 Activity.Stop()。这简化了编码模式。
- ActivitySource.StartActivity 在内部确定是否有任何侦听器记录 Activity。如果没有已注册的侦听器，或有不关注此类事件的侦听器，那么 `StartActivity()` 会返回 `null`，并避免创建 Activity 对象。这是一项性能优化，以便代码模式仍可用于频繁调用的函数。

可选：填充标记

Activity 支持名为标记的键值数据，后者通常用于存储可能对诊断有用的工作的所有参数。更新 DoSomeWork() 以包括它们：

```
        static async Task DoSomeWork(string foo, int bar)
        {
            using (Activity activity = source.StartActivity("SomeWork"))
            {
                activity?.SetTag("foo", foo);
                activity?.SetTag("bar", bar);
                await StepOne();
                await StepTwo();
            }
        }
```

```
> dotnet run
Activity.Id:          00-2b56072db8cb5a4496a4bfb69f46aa06-7bc4acda3b9cce4d-01
Activity.DisplayName: SomeWork
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:37:31.4949570Z
Activity.Duration:    00:00:01.5417719
Activity.TagObjects:
    foo: banana
    bar: 8
Resource associated with Activity:
    service.name: MySample
    service.instance.id: 25bbc1c3-2de5-48d9-9333-062377fea49c

Example work done
```

最佳做法

- 如上所述，ActivitySource.StartActivity 返回的 `activity` 可能为 null。C# 中的 Null 合并操作符 `?.` 是一个方便的快捷方式，它仅在 `activity` 不为 null 时调用 Activity.SetTag。该行为等同于写入：

```
if(activity != null)
{
    activity.SetTag("foo", foo);
}
```

- OpenTelemetry 提供一组建议的约定，用于在 Activity 上设置代表常见应用程序工作类型的标记。
- 如果要检测具有高性能要求的函数，则 Activity.IsAllDataRequested 是一个提示，可指示侦听 Activity 的任何代码是否打算读取辅助信息，例如标记。如果没有侦听器要进行读取，则检测代码无需耗费 CPU 周期来填充它。为简单起见，此示例未应用该优化。

可选：添加事件

事件是带有时间戳的消息，可以将任意附加诊断数据流附加到 Activity。向 Activity 添加一些事件：

```
        static async Task DoSomeWork(string foo, int bar)
        {
            using (Activity activity = source.StartActivity("SomeWork"))
            {
                activity?.SetTag("foo", foo);
                activity?.SetTag("bar", bar);
                await StepOne();
                activity?.AddEvent(new ActivityEvent("Part way there"));
                await StepTwo();
                activity?.AddEvent(new ActivityEvent("Done now"));
            }
        }
```

```
> dotnet run
Activity.Id:          00-82cf6ea92661b84d9fd881731741d04e-33fff2835a03c041-01
Activity.DisplayName: SomeWork
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:39:10.6902609Z
Activity.Duration:    00:00:01.5147582
Activity.TagObjects:
    foo: banana
    bar: 8
Activity.Events:
    Part way there [3/18/2021 10:39:11 AM +00:00]
    Done now [3/18/2021 10:39:12 AM +00:00]
Resource associated with Activity:
    service.name: MySample
    service.instance.id: ea7f0fcb-3673-48e0-b6ce-e4af5a86ce4f

Example work done
```

最佳做法

- 事件存储在内存中列表中，直到可以传输，这使得此机制仅适合记录适量的事件。对于大量或不限量的事件，最好使用专注于此任务的日志记录 API（例如 ILogger）。ILogger 还可确保无论应用开发人员是否选择使用分布式跟踪，日志记录信息都将可用。ILogger 支持自动捕获活动 Activity ID，因此仍可以将通过该 API 记录的消息与分布式跟踪关联。

可选：添加状态

OpenTelemetry 允许每个 Activity 报告代表工作的通过/失败结果的状态。.NET 目前没有用于此目的的强类型 API，但存在有关使用标记的既有约定：

- `otel.status_code` 是用于存储 `StatusCode` 的标记名称。StatusCode 标记的值必须是字符串“UNSET”、

"OK"或"ERROR"之一，其分别对应于 StatusCode 枚举 `Unset`、`Ok` 和 `Error`。

- `otel.status_description` 是用于存储可选 `Description` 的标记名称

更新 DoSomeWork() 以设置状态：

```
        static async Task DoSomeWork(string foo, int bar)
        {
            using (Activity activity = source.StartActivity("SomeWork"))
            {
                activity?.SetTag("foo", foo);
                activity?.SetTag("bar", bar);
                await StepOne();
                activity?.AddEvent(new ActivityEvent("Part way there"));
                await StepTwo();
                activity?.AddEvent(new ActivityEvent("Done now"));

                // Pretend something went wrong
                activity?.SetTag("otel.status_code", "ERROR");
                activity?.SetTag("otel.status_description", "Use this text give more information about the
error");
            }
        }
```

可选：添加其他 Activity

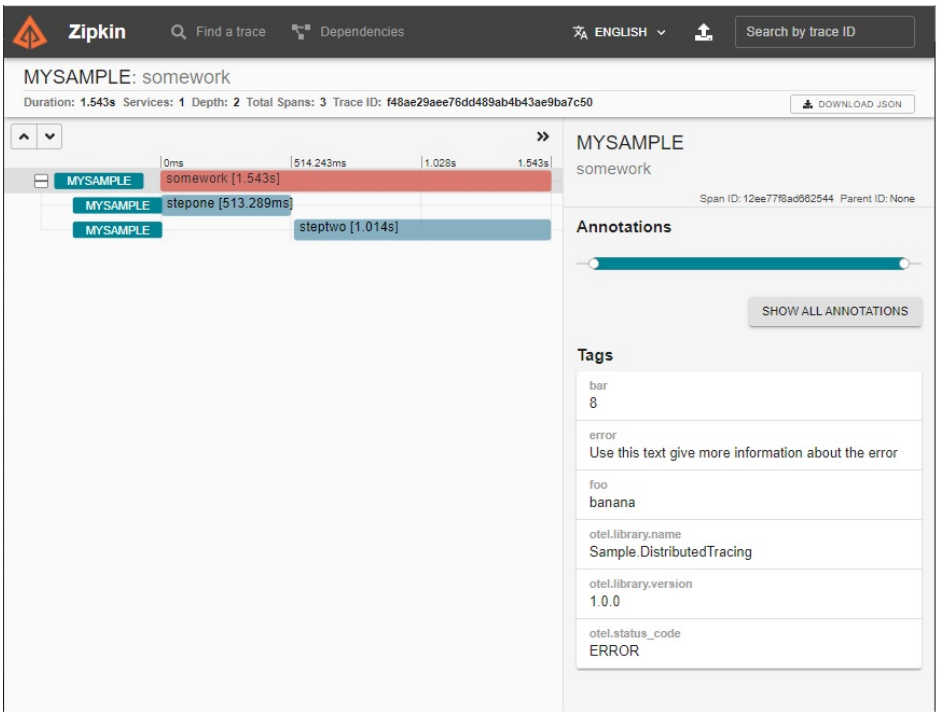
可以嵌套 Activity 用于描述较大工作单元的各个部分。这对于可能不会快速执行的代码部分或更好地找到来自特定外部依赖项的故障而言很有价值。尽管此示例在每种方法中都使用 Activity，但这仅仅是因为已最大限度地减少了额外的代码。在更大、更真实的项目中，在每种方法中都使用 Activity 会产生极其详细的跟踪，因此不建议这样做。

更新 StepOne 和 StepTwo，以围绕这些单独的步骤添加更多跟踪：

```
        static async Task StepOne()
        {
            using (Activity activity = source.StartActivity("StepOne"))
            {
                await Task.Delay(500);
            }
        }

        static async Task StepTwo()
        {
            using (Activity activity = source.StartActivity("StepTwo"))
            {
                await Task.Delay(1000);
            }
        }
```

```
> dotnet run
Activity.Id:          00-9d5aa439e0df7e49b4abff8d2d5329a9-39cac574e8fda44b-01
Activity.ParentId:    00-9d5aa439e0df7e49b4abff8d2d5329a9-f16529d0b7c49e44-01
Activity.DisplayName: StepOne
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:40:51.4278822Z
Activity.Duration:    00:00:00.5051364
Resource associated with Activity:
    service.name: MySample
    service.instance.id: e0a8c12c-249d-4bdd-8180-8931b9b6e8d0

Activity.Id:          00-9d5aa439e0df7e49b4abff8d2d5329a9-4ccccb6efdc59546-01
Activity.ParentId:    00-9d5aa439e0df7e49b4abff8d2d5329a9-f16529d0b7c49e44-01
Activity.DisplayName: StepTwo
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:40:51.9441095Z
Activity.Duration:    00:00:01.0052729
Resource associated with Activity:
    service.name: MySample
    service.instance.id: e0a8c12c-249d-4bdd-8180-8931b9b6e8d0

Activity.Id:          00-9d5aa439e0df7e49b4abff8d2d5329a9-f16529d0b7c49e44-01
Activity.DisplayName: SomeWork
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:40:51.4256627Z
Activity.Duration:    00:00:01.5286408
Activity.TagObjects:
    foo: banana
    bar: 8
    otel.status_code: ERROR
    otel.status_description: Use this text give more information about the error
Activity.Events:
    Part way there [3/18/2021 10:40:51 AM +00:00]
    Done now [3/18/2021 10:40:52 AM +00:00]
Resource associated with Activity:
    service.name: MySample
    service.instance.id: e0a8c12c-249d-4bdd-8180-8931b9b6e8d0

Example work done
```

请注意，StepOne 和 StepTwo 都包含引用 SomeWork 的 ParentId。控制台不能很好地呈现嵌套工作树，但许多 GUI 查看器（例如 Zipkin）都可以将其显示为甘特图：



可选：ActivityKind

Activity 包含描述 Activity、其父项和子项之间关系的 Activity.Kind 属性。默认情况下，所有新 Activity 都设置为 Internal，这适用于属于应用程序中的内部操作且没有远程父项或子项的 Activity。可以使用 ActivitySource.StartActivity 上的 kind 参数设置其他类型。有关其他选项，请参阅 System.Diagnostics.ActivityKind。

可选：链接

当工作在批处理系统中发生时，单个 Activity 可能表示同时代表许多不同请求的工作，且其中每个请求都有自己的 trace-id。尽管 Activity 被限制为具有单个父项，但它可以使用 System.Diagnostics.ActivityLink 链接到其他 trace-id。每个 ActivityLink 都填充有 ActivityContext，后者存储有关要链接到的 Activity 的 ID 信息。可以使用 Activity.Context 从进程内 Activity 对象中检索 ActivityContext，也可以使用 ActivityContext.Parse(String, String) 从序列化 ID 信息中分析它。

```
void DoBatchWork(ActivityContext[] requestContexts)
{
    // Assume each context in requestContexts encodes the trace-id that was sent with a request
    using(Activity activity = s_source.StartActivity(name: "BigBatchOfWork",
                                                     kind: ActivityKind.Internal,
                                                     parentContext: null,
                                                     links: requestContexts.Select(ctx => new
ActivityLink(ctx))
    {
        // do the batch of work here
    }
}
```

与可以按需添加的事件和标记不同，链接必须在 StartActivity() 期间添加且此后不可变。

收集分布式跟踪

2021/11/16 •

本文适用范围：✔ .NET Core 2.1 及更高版本 ✔ .NET Framework 4.5 及更高版本

检测代码可以创建 Activity 对象作为分布式跟踪的一部分，但需要将这些对象中的信息收集到集中存储中，以便稍后可以查看整个跟踪。本教程将以不同的方式收集分布式跟踪遥测，以便在需要时可用于诊断应用程序问题。如果需要添加新的检测，请参阅检测教程。

使用 OpenTelemetry 收集跟踪

先决条件

- .NET Core 2.1 SDK 或更高版本

创建一个示例应用程序

我们需要先生成分布式跟踪遥测，然后才能进行收集。通常，此检测可能位于库中，但为简单起见，将使用 StartActivity 创建一个小型应用并在其中包含一些示例检测。此时，尚未进行任何收集，StartActivity() 没有副作用，并且返回 null。有关详细信息，请参阅检测教程。

```
dotnet new console
```

面向 .NET 5 及更高版本的应用程序已包含必要的分布式跟踪 API。对于面向早期 .NET 版本的应用，请添加 System.Diagnostics.DiagnosticSource NuGet 包版本 5 或更高版本。

```
dotnet add package System.Diagnostics.DiagnosticSource
```

将生成的 Program.cs 内容替换为此示例源：

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Sample.DistributedTracing
{
    class Program
    {
        static ActivitySource s_source = new ActivitySource("Sample.DistributedTracing");

        static async Task Main(string[] args)
        {
            await DoSomeWork();
            Console.WriteLine("Example work done");
        }

        static async Task DoSomeWork()
        {
            using (Activity a = s_source.StartActivity("SomeWork"))
            {
                await StepOne();
                await StepTwo();
            }
        }

        static async Task StepOne()
        {
            using (Activity a = s_source.StartActivity("StepOne"))
            {
                await Task.Delay(500);
            }
        }

        static async Task StepTwo()
        {
            using (Activity a = s_source.StartActivity("StepTwo"))
            {
                await Task.Delay(1000);
            }
        }
    }
}
```

运行应用时暂时不会收集任何跟踪数据：

```
> dotnet run
Example work done
```

使用 OpenTelemetry 收集

OpenTelemetry 是由云本机计算基础支持的与供应商无关的开源项目，旨在标准化为云原生软件生成和收集遥测的过程。此示例将收集和显示控制台上的分布式跟踪信息，但可以重新配置 OpenTelemetry 以将其发送到其他位置。有关详细信息，请参阅 OpenTelemetry 入门指南。

添加 OpenTelemetry.Exporter.Console NuGet 包。

```
dotnet add package OpenTelemetry.Exporter.Console
```

使用其他 OpenTelemetry `using` 指令更新 Program.cs：

```
using OpenTelemetry;
using OpenTelemetry.Resources;
using OpenTelemetry.Trace;
using System;
using System.Diagnostics;
using System.Threading.Tasks;
```

更新 `Main()` 以创建 OpenTelemetry TracerProvider：

```
        public static async Task Main()
        {
            using var tracerProvider = Sdk.CreateTracerProviderBuilder()
                .SetResourceBuilder(ResourceBuilder.CreateDefault().AddService("MySample"))
                .AddSource("Sample.DistributedTracing")
                .AddConsoleExporter()
                .Build();

            await DoSomeWork();
            Console.WriteLine("Example work done");
        }
```

现在，应用将收集分布式跟踪信息，并将其显示到控制台：

```
> dotnet run
Activity.Id:          00-7759221f2c5599489d455b84fa0f90f4-6081a9b8041cd840-01
Activity.ParentId:    00-7759221f2c5599489d455b84fa0f90f4-9a52f72c08a9d447-01
Activity.DisplayName: StepOne
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:46:46.8649754Z
Activity.Duration:    00:00:00.5069226
Resource associated with Activity:
    service.name: MySample
    service.instance.id: 909a4624-3b2e-40e4-a86b-4a2c8003219e

Activity.Id:          00-7759221f2c5599489d455b84fa0f90f4-d2b283db91cf774c-01
Activity.ParentId:    00-7759221f2c5599489d455b84fa0f90f4-9a52f72c08a9d447-01
Activity.DisplayName: StepTwo
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:46:47.3838737Z
Activity.Duration:    00:00:01.0142278
Resource associated with Activity:
    service.name: MySample
    service.instance.id: 909a4624-3b2e-40e4-a86b-4a2c8003219e

Activity.Id:          00-7759221f2c5599489d455b84fa0f90f4-9a52f72c08a9d447-01
Activity.DisplayName: SomeWork
Activity.Kind:        Internal
Activity.StartTime:   2021-03-18T10:46:46.8634510Z
Activity.Duration:    00:00:01.5402045
Resource associated with Activity:
    service.name: MySample
    service.instance.id: 909a4624-3b2e-40e4-a86b-4a2c8003219e

Example work done
```

源

此示例代码调用了 `AddSource("Sample.DistributedTracing")`，这样 OpenTelemetry 就可以捕获代码中已经存在的 ActivitySource 所生成的 Activity：

```
static ActivitySource s_source = new ActivitySource("Sample.DistributedTracing");
```

可以通过使用源名称调用 `AddSource()` 来捕获任何 ActivitySource 中的遥测。

导出工具

控制台导出工具对简单的示例或本地开发非常有用，但在生产部署中，可能需要将跟踪发送到集中存储。OpenTelemetry 支持使用不同导出工具的各种目标。有关如何配置 OpenTelemetry 的详细信息，请参阅 OpenTelemetry 入门指南。

使用 Application Insights 收集跟踪

为 ASP.NET 或 ASP.NET Core 应用配置 Application Insights SDK 或者启用无代码检测后，系统会自动捕获分布式跟踪遥测。

有关详细信息，请参阅 Application Insights 分布式跟踪文档。

> **NOTE**
> 目前，Application Insights 仅支持收集特定的已知 Activity 检测，并忽略新用户添加的 Activity。Application Insights 提供 TrackDependency 作为供应商特定的 API，用于添加自定义分布式跟踪信息。

使用自定义逻辑收集跟踪

开发人员可随意为活动跟踪数据创建自定义收集逻辑。此示例使用 .NET 提供的 System.Diagnostics.ActivityListener API 收集遥测数据，并将其输出到控制台。

先决条件

- .NET Core 2.1 SDK 或更高版本

创建一个示例应用程序

首先将创建一个示例应用程序，并在其中包含一些分布式跟踪检测，但未收集任何跟踪数据。

```
dotnet new console
```

面向 .NET 5 及更高版本的应用程序已包含必要的分布式跟踪 API。对于面向早期 .NET 版本的应用，请添加 System.Diagnostics.DiagnosticSource NuGet 包版本 5 或更高版本。

```
dotnet add package System.Diagnostics.DiagnosticSource
```

将生成的 Program.cs 内容替换为此示例源：

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

namespace Sample.DistributedTracing
{
    class Program
    {
        static ActivitySource s_source = new ActivitySource("Sample.DistributedTracing");

        static async Task Main(string[] args)
        {
            await DoSomeWork();
            Console.WriteLine("Example work done");
        }

        static async Task DoSomeWork()
        {
            using (Activity a = s_source.StartActivity("SomeWork"))
            {
                await StepOne();
                await StepTwo();
            }
        }

        static async Task StepOne()
        {
            using (Activity a = s_source.StartActivity("StepOne"))
            {
                await Task.Delay(500);
            }
        }

        static async Task StepTwo()
        {
            using (Activity a = s_source.StartActivity("StepTwo"))
            {
                await Task.Delay(1000);
            }
        }
    }
}
```

运行应用时暂时不会收集任何跟踪数据：

```
> dotnet run
Example work done
```

添加代码以收集跟踪数据

使用下面的代码更新 Main()：

```
        static async Task Main(string[] args)
        {
            Activity.DefaultIdFormat = ActivityIdFormat.W3C;
            Activity.ForceDefaultIdFormat = true;

            Console.WriteLine("         {0,-15} {1,-60} {2,-15}", "OperationName", "Id", "Duration");
            ActivitySource.AddActivityListener(new ActivityListener()
            {
                ShouldListenTo = (source) => true,
                Sample = (ref ActivityCreationOptions<ActivityContext> options) =>
ActivitySamplingResult.AllDataAndRecorded,
                ActivityStarted = activity => Console.WriteLine("Started: {0,-15} {1,-60}",
activity.OperationName, activity.Id),
                ActivityStopped = activity => Console.WriteLine("Stopped: {0,-15} {1,-60} {2,-15}",
activity.OperationName, activity.Id, activity.Duration)
            });

            await DoSomeWork();
            Console.WriteLine("Example work done");
        }
```

现在，输出包括日志记录：

```
> dotnet run
         OperationName   Id                                                          Duration
Started: SomeWork        00-bdb5faffc2fc1548b6ba49a31c4a0ae0-c447fb302059784f-01
Started: StepOne         00-bdb5faffc2fc1548b6ba49a31c4a0ae0-a7c77a4e9a02dc4a-01
Stopped: StepOne         00-bdb5faffc2fc1548b6ba49a31c4a0ae0-a7c77a4e9a02dc4a-01      00:00:00.5093849
Started: StepTwo         00-bdb5faffc2fc1548b6ba49a31c4a0ae0-9210ad536cae9e4e-01
Stopped: StepTwo         00-bdb5faffc2fc1548b6ba49a31c4a0ae0-9210ad536cae9e4e-01      00:00:01.0111847
Stopped: SomeWork        00-bdb5faffc2fc1548b6ba49a31c4a0ae0-c447fb302059784f-01      00:00:01.5236391
Example work done
```

虽然设置 DefaultIdFormat 和 ForceDefaultIdFormat 是可选操作，但这样做有助于确保示例在不同的 .NET 运行时版本上生成类似的输出。.NET 5 默认使用 W3C TraceContext ID 格式，但早期的 .NET 版本默认使用 Hierarchical ID 格式。有关详细信息，请参阅 Activity ID。

System.Diagnostics.ActivityListener 用于在活动的生存期内接收回调。

- ShouldListenTo - 每个 Activity 都与 ActivitySource 关联，后者可充当 Activity 的命名空间和生成方。对于进程中的每个 ActivitySource，都会调用一次此回调。如果你有兴趣执行采样或收到有关此源产生的活动的启动/停止事件的通知，则返回 true。
- Sample - 默认情况下，StartActivity 不会创建 Activity 对象，除非某个 ActivityListener 指示应对其进行采用。返回 AllDataAndRecorded 表示应创建活动，IsAllDataRequested 应设置为 true，并且 ActivityTraceFlags 将设置 Recorded 标志。IsAllDataRequested 可被检测代码观测到，这提示侦听器需要确保填充辅助 Activity 信息（如标记和事件）。记录的标志在 W3C TraceContext ID 中进行编码，暗示分布式跟踪中涉及的其他进程应对此跟踪进行采样。
- ActivityStarted 和 ActivityStopped 分别在启动和停止活动时调用。可以通过这些回调记录 Activity 的相关信息或对其进行修改。当 Activity 刚启动时，许多数据可能仍然不完整，在 Activity 停止之前，系统会对这些数据进行填充。

创建 ActivityListener 并填充回调之后，即可通过调用 ActivitySource.AddActivityListener(ActivityListener) 启动调用回调操作。调用 ActivityListener.Dispose() 可停止回调流。请注意，在多线程代码中，当 `Dispose()` 运行时，甚至在它返回后不久，都可能会收到正在进行的回调通知。

.NET Core 中的 EventCounters

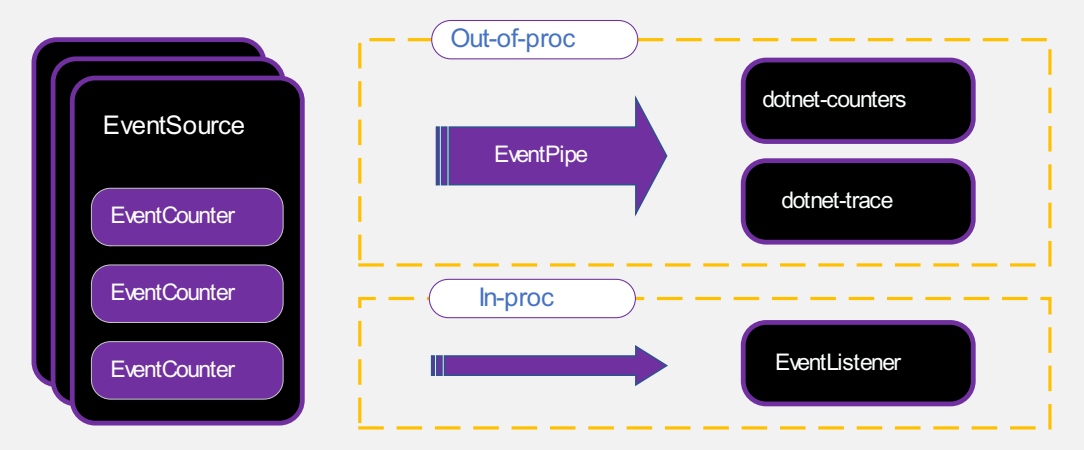
2021/11/16 •

本文适用于： ✔ .NET Core 3.0 SDK 及更高版本

EventCounters 是一些 .NET Core API，用于轻量级、跨平台、准实时性能指标收集。EventCounters 添加为 Windows 上的 .NET 框架的"性能计数器"的跨平台替代。本文将介绍什么是 EventCounters，如何实现它们，以及如何使用它们。

.NET Core 运行时和几个 .NET 库使用从 .NET Core 3.0 开始引入的 EventCounters 发布基本诊断信息。除了 .NET 运行时提供的 EventCounters 外，你还可以选择实现自己的 EventCounters。可使用 EventCounters 跟踪各种指标。在 .NET 中的已知 EventCounters 中详细了解其信息

EventCounters 作为 EventSource 的一部分实时自动定期推送到侦听器工具。与 EventSource 上所有其他事件一样，可以通过 EventListener 和 EventPipe 在进程内和进程外使用它们。本文重点介绍 EventCounters 的跨平台功能，并特意排除 PerfView 和 ETW（Windows 事件跟踪）- 尽管两者都可用于 EventCounters。



EventCounter API 概述

有两种主要类别的计数器。某些计数器用于计算"比率"的值，例如异常总数、GC 总数和请求总数。其他计数器是"快照"值，例如堆使用情况、CPU 使用率和工作集大小。在这两个类别的计数器中，各有两种类型的计数器，由获取值的方式区分。轮询计数器通过回调检索其值，非轮询计数器直接在计数器实例上设置其值。

计数器由以下实现表示：

- EventCounter
- IncrementingEventCounter
- IncrementingPollingCounter
- PollingCounter

事件侦听器指定测量间隔的时长。在每个间隔结束时，每个计数器的值将传输到侦听器。计数器的实现确定使用哪些 API 和计算来生成每个间隔的值。

1. EventCounter 记录一组值。EventCounter.WriteMetric 方法将新值添加到集。在每个间隔中，将计算集的统计摘要，如最小值、最大值和平均值。dotnet-counters 工具将始终显示平均值。EventCounter 用于描述一组离散的操作。常见用法包括监视最近 IO 操作的平均大小（以字节为单位）或一组金融交易的平均货币价值。

2. IncrementingEventCounter 记录每个时间间隔的运行总计。IncrementingEventCounter.Increment 方法添加到总计。例如，如果在一段间隔内调用三次 `Increment()`，其值分别为 `1`、`2` 和 `5`，则此间隔的计数器值将报告运行总计 `8`。dotnet-counters 工具将比率显示为记录的总计/时间。
   IncrementingEventCounter 用于测量操作发生的频率，例如每秒处理的请求数。

3. PollingCounter 使用回调来确定报告的值。在每个时间间隔中，调用用户提供的回调函数，然后返回值用作计数器值。可以使用 PollingCounter 从外部源查询指标，例如获取磁盘上的当前可用字节。它还可用于报告应用程序可按需计算的自定义统计信息。示例包括报告最近请求延迟的第 95 个百分位，或缓存的当前命中或错过比率。

4. IncrementingPollingCounter 使用回调来确定报告的增量值。对于每个时间间隔，调用回调，然后当前调用与最后一个调用之间的差值是报告的值。dotnet-counters 工具始终将比率显示为报告的值/时间。如果不可在每次发生事件时调用 API，但可以查询事件总数，则此计数器很有用。例如，可以报告每秒写入文件的字节数，即使每次写入字节时没有通知。

实现 EventSource

下面的代码实现作为命名 `"Sample.EventCounter.Minimal"` 提供程序公开的示例 EventSource。此源包含表示请求处理时间的 EventCounter。此类计数器具有名称(即其在源中的唯一 ID)和显示名称，这两个名称都可由侦听器工具(如 dotnet-counter)使用。

```
using System.Diagnostics.Tracing;

[EventSource(Name = "Sample.EventCounter.Minimal")]
public sealed class MinimalEventCounterSource : EventSource
{
    public static readonly MinimalEventCounterSource Log = new MinimalEventCounterSource();

    private EventCounter _requestCounter;

    private MinimalEventCounterSource() =>
        _requestCounter = new EventCounter("request-time", this)
        {
            DisplayName = "Request Processing Time",
            DisplayUnits = "ms"
        };

    public void Request(string url, long elapsedMilliseconds)
    {
        WriteEvent(1, url, elapsedMilliseconds);
        _requestCounter?.WriteMetric(elapsedMilliseconds);
    }

    protected override void Dispose(bool disposing)
    {
        _requestCounter?.Dispose();
        _requestCounter = null;

        base.Dispose(disposing);
    }
}
```

可以使用 `dotnet-counters ps` 来显示可监视的 .NET 进程的列表：

```
dotnet-counters ps
   1398652 dotnet     C:\Program Files\dotnet\dotnet.exe
   1399072 dotnet     C:\Program Files\dotnet\dotnet.exe
   1399112 dotnet     C:\Program Files\dotnet\dotnet.exe
   1401880 dotnet     C:\Program Files\dotnet\dotnet.exe
   1400180 sample-counters C:\sample-counters\bin\Debug\netcoreapp3.1\sample-counters.exe
```

将 EventSource 名称传递到 `--counters` 选项，以开始监视计数器：

```
dotnet-counters monitor --process-id 1400180 --counters Sample.EventCounter.Minimal
```

以下示例显示监视器输出：

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Samples-EventCounterDemos-Minimal]
    Request Processing Time (ms)                              0.445
```

按 q 停止监视命令。

条件计数器

实现 EventSource 时，通过 Command 值 `EventCommand.Enable` 调用 EventSource.OnEventCommand 方法时，可以有条件地实例化包含计数器。要仅在计数器实例为 `null` 时将其安全地实例化，请使用 null 合并赋值运算符。此外，自定义方法可以计算 IsEnabled 方法，以确定是否启用了当前事件源。

```
using System.Diagnostics.Tracing;

[EventSource(Name = "Sample.EventCounter.Conditional")]
public sealed class ConditionalEventCounterSource : EventSource
{
    public static readonly ConditionalEventCounterSource Log = new ConditionalEventCounterSource();

    private EventCounter _requestCounter;

    private ConditionalEventCounterSource() { }

    protected override void OnEventCommand(EventCommandEventArgs args)
    {
        if (args.Command == EventCommand.Enable)
        {
            _requestCounter ??= new EventCounter("request-time", this)
            {
                DisplayName = "Request Processing Time",
                DisplayUnits = "ms"
            };
        }
    }

    public void Request(string url, float elapsedMilliseconds)
    {
        if (IsEnabled())
        {
            _requestCounter?.WriteMetric(elapsedMilliseconds);
        }
    }

    protected override void Dispose(bool disposing)
    {
        _requestCounter?.Dispose();
        _requestCounter = null;

        base.Dispose(disposing);
    }
}
```

> **TIP**
> 条件计数器是有条件地实例化的计数器，即微优化。对于通常不使用计数器的场景，运行时采用此模式来节省不到一毫秒的时间。

.NET Core 运行时示例计数器

在 .NET Core 运行时中有许多很好的示例实现。下面是跟踪应用程序工作集大小的计数器的运行时实现。

```
var workingSetCounter = new PollingCounter(
    "working-set",
    this,
    () => (double)(Environment.WorkingSet / 1_000_000))
{
    DisplayName = "Working Set",
    DisplayUnits = "MB"
};
```

PollingCounter 报告映射到应用的进程(工作集)的当前物理内存量，因为它在一个时刻捕获一个指标。轮询值的回调是提供的 lambda 表达式，这只是对 System.Environment.WorkingSet API 的调用。DisplayName 和 DisplayUnits 是可选属性，可以设置它们，帮助计数器的使用者方更清楚地显示值。例如，dotnet-counters 使用这些属性来显示计数器名称的更具有显示友好性的版本。

> **IMPORTANT**
> `DisplayName` 属性未本地化。

对于 PollingCounter 和 IncrementingPollingCounter，无需执行任何其他操作。它们本身都按使用者请求的时间间隔轮询值。

下面是使用 IncrementingPollingCounter 实现的运行时计数器的示例。

```
var monitorContentionCounter = new IncrementingPollingCounter(
    "monitor-lock-contention-count",
    this,
    () => Monitor.LockContentionCount
)
{
    DisplayName = "Monitor Lock Contention Count",
    DisplayRateTimeScale = TimeSpan.FromSeconds(1)
};
```

IncrementingPollingCounter 使用 Monitor.LockContentionCount API 报告锁争用数总计的增量。DisplayRateTimeScale 属性可选，但使用它时，它可以提供有关计数器最佳显示时间间隔的提示。例如，锁争用计数最好显示为"每秒计数"，因此其 DisplayRateTimeScale 设置为一秒。可为不同类型的比率计数器调整显示比率。

> **NOTE**
> DisplayRateTimeScale 不由 dotnet-counters 使用，不需要事件侦听器即可使用它。

在 .NET 运行时存储库中，有更多的计数器实现可用作参考。

并发

> **TIP**
> EventCounters API 不能保证线程安全性。当传递到 PollingCounter 或 IncrementingPollingCounter 实例的委托由多个线程调用时，你有责任保证委托的线程安全性。

例如，请考虑使用以下 EventSource 来跟踪请求。

```
using System;
using System.Diagnostics.Tracing;

public class RequestEventSource : EventSource
{
    public static readonly RequestEventSource Log = new RequestEventSource();

    private IncrementingPollingCounter _requestRateCounter;
    private long _requestCount = 0;

    private RequestEventSource() =>
        _requestRateCounter = new IncrementingPollingCounter("request-rate", this, () => _requestCount)
        {
            DisplayName = "Request Rate",
            DisplayRateTimeScale = TimeSpan.FromSeconds(1)
        };

    public void AddRequest() => ++ _requestCount;

    protected override void Dispose(bool disposing)
    {
        _requestRateCounter?.Dispose();
        _requestRateCounter = null;

        base.Dispose(disposing);
    }
}
```

可以从请求处理程序调用 `AddRequest()` 方法，并且 `RequestRateCounter` 按计数器使用者指定的间隔轮询值。但是，`AddRequest()` 方法可以同时由多个线程调用，将争用条件置于 `_requestCount`。增加 `_requestCount` 的线程安全替代方法是使用 Interlocked.Increment。

```
public void AddRequest() => Interlocked.Increment(ref _requestCount);
```

若要防止破坏(在 32 位体系结构上)对 `long` 字段 `_requestCount` 的读取，请使用 Interlocked.Read。

```
_requestRateCounter = new IncrementingPollingCounter("request-rate", this, () => Interlocked.Read(ref
_requestCount))
{
    DisplayName = "Request Rate",
    DisplayRateTimeScale = TimeSpan.FromSeconds(1)
};
```

使用 EventCounters

使用 EventCounters 主要方式有两种：进程内或进程外。EventCounters 的使用可以分为三层不同的使用技术。

- 通过 ETW 或 EventPipe 在原始流中传输事件：
  - ETW API 附带 Windows OS，EventPipe 可作为 .NET API 或诊断 IPC 协议进行访问。
- 将二进制事件流解码为事件：

- TraceEvent 库可处理 ETW 和 EventPipe 流格式。
- 命令行和 GUI 工具：
  - PerfView（ETW 或 EventPipe）、dotnet-counters（仅 EventPipe）和 dotnet-monitor（仅 EventPipe）等工具。

进程外使用

进程外使用 EventCounters 是一种非常常见的方法。你可以使用 dotnet-counters 通过 EventPipe 以跨平台方式使用它们。`dotnet-counters` 工具是一个跨平台 dotnet CLI 全局工具，可用于监视计数器值。要了解如何使用 `dotnet-counters` 监视计数器，请参阅 dotnet-counters 或浏览使用 EventCounters 衡量性能教程。

dotnet-trace

`dotnet-trace` 工具可用于通过 EventPipe 使用计数器数据。下面是使用 `dotnet-trace` 收集计数器数据的一个示例。

```
dotnet-trace collect --process-id <pid> Sample.EventCounter.Minimal:0:0:EventCounterIntervalSec=1
```

有关如何随着时间的推移收集计数器值的详细信息，请参阅 dotnet-trace 文档。

Azure Application Insights

EventCounters 可由 Azure Monitor 使用，特别是 Azure Application Insights。可以添加和删除计数器，并且可以自由指定自定义计数器或已知计数器。有关详细信息，请参阅自定义要收集的计数器。

dotnet-monitor

`dotnet-monitor` 工具是一个实验性工具，通过它可以更轻松地访问 .NET 进程中的诊断信息。该工具用作所有诊断工具的超集。除跟踪外，它还可以监视指标、收集内存转储和收集 GC 转储。它以 CLI 工具和 docker 映像的形式发布。它公开了 REST API，以及通过 REST 调用发生的诊断项目集合。

有关详细信息，请参阅实验性工具 dotnet-monitor 简介。

进程内使用

可以通过 EventListener API 使用计数器值。EventListener 是使用由应用程序中 EventSource 的所有实例编写的任何事件的一种进程内方法。有关如何使用 `EventListener` API 的详细信息，请参阅 EventListener。

首先，需要启用生成计数器值的 EventSource。替代 EventListener.OnEventSourceCreated 方法以在创建 EventSource 时获取通知，如果对于 EventCounters 这是正确的 EventSource，则可在其上调用 EventListener.EnableEvents。下面是示例替代：

```
protected override void OnEventSourceCreated(EventSource source)
{
    if (!source.Name.Equals("System.Runtime"))
    {
        return;
    }

    EnableEvents(source, EventLevel.Verbose, EventKeywords.All, new Dictionary<string, string>()
    {
        ["EventCounterIntervalSec"] = "1"
    });
}
```

代码示例

下面是一个示例 EventListener 类，它打印出 .NET 运行时的 EventSource 的所有计数器名称和值，用于每秒发布其内部计数器（`System.Runtime`）。

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;

public class SimpleEventListener : EventListener
{
    public SimpleEventListener()
    {
    }

    protected override void OnEventSourceCreated(EventSource source)
    {
        if (!source.Name.Equals("System.Runtime"))
        {
            return;
        }

        EnableEvents(source, EventLevel.Verbose, EventKeywords.All, new Dictionary<string, string>()
        {
            ["EventCounterIntervalSec"] = "1"
        });
    }

    protected override void OnEventWritten(EventWrittenEventArgs eventData)
    {
        if (!eventData.EventName.Equals("EventCounters"))
        {
            return;
        }

        for (int i = 0; i < eventData.Payload.Count; ++ i)
        {
            if (eventData.Payload[i] is IDictionary<string, object> eventPayload)
            {
                var (counterName, counterValue) = GetRelevantMetric(eventPayload);
                Console.WriteLine($"{counterName} : {counterValue}");
            }
        }
    }

    private static (string counterName, string counterValue) GetRelevantMetric(
        IDictionary<string, object> eventPayload)
    {
        var counterName = "";
        var counterValue = "";

        if (eventPayload.TryGetValue("DisplayName", out object displayValue))
        {
            counterName = displayValue.ToString();
        }
        if (eventPayload.TryGetValue("Mean", out object value) ||
            eventPayload.TryGetValue("Increment", out value))
        {
            counterValue = value.ToString();
        }

        return (counterName, counterValue);
    }
}
```

如下所示，调用 EnableEvents 时必须确保在 `filterPayload` 参数中设置 `"EventCounterIntervalSec"` 参数。否则，计数器将无法清空值，因为它不知道应清空哪个时间间隔。

另请参阅

- dotnet-counters
- dotnet-trace
- EventCounter
- EventListener
- EventSource

.NET 中的已知 EventCounters

2021/11/16 •

.NET 运行时和库实现并发布几个 `EventCounter`，可用于识别和诊断各种性能问题。通过本文档，你可了解可用于监视这些 `EventCounters` 的提供程序及其描述。

System.Runtime 计数器

以下计数器作为 .NET 运行时 (CoreCLR) 的一部分发布，并在 `RuntimeEventSource.cs` 中进行维护。

% Time in GC since last GC (`time-in-gc`)	自上次 GC 以来 GC 的时间百分比	.NET Core 3.1
Allocation Rate (`alloc-rate`)	每个更新间隔分配的字节数	.NET Core 3.1
CPU Usage (`cpu-usage`)	相对于所有系统 CPU 资源进程的 CPU 使用率百分比	.NET Core 3.1
Exception Count (`exception-count`)	已发生的异常数	.NET Core 3.1
GC Heap Size (`gc-heap-size`)	根据 GC.GetTotalMemory(Boolean) 认为要分配的字节数	.NET Core 3.1
Gen 0 GC Count (`gen-0-gc-count`)	每个更新间隔发生的第 0 代 GC 次数	.NET Core 3.1
Gen 0 Size (`gen-0-size`)	第 0 代 GC 的字节数	.NET Core 3.1
Gen 1 GC Count (`gen-1-gc-count`)	每个更新间隔发生的第 1 代 GC 次数	.NET Core 3.1
Gen 1 Size (`gen-1-size`)	第 1 代 GC 的字节数	.NET Core 3.1
Gen 2 GC Count (`gen-2-gc-count`)	每个更新间隔发生的第 2 代 GC 次数	.NET Core 3.1
Gen 2 Size (`gen-2-size`)	第 2 代 GC 的字节数	.NET Core 3.1
LOH Size (`loh-size`)	大型对象堆的字节数	.NET Core 3.1
POH Size (`poh-size`)	已固定对象堆的字节数(在 .NET 5 及更高版本中可用)	.NET Core 3.1
GC Fragmentation (`gc-fragmentation`)	GC 堆碎片(在 .NET 5 及更高版本中可用)	.NET Core 3.1
Monitor Lock Contention Count (`monitor-lock-contention-count`)	尝试锁定监视器时出现争用的次数，基于 Monitor.LockContentionCount	.NET Core 3.1

[illegible]	[illegible]	[illegible]
Number of Active Timers (`active-timer-count`)	当前活动的 Timer 实例的计数，基于 Timer.ActiveCount	.NET Core 3.1
Number of Assemblies Loaded (`assembly-count`)	在某个时间点加载到进程中的 Assembly 实例的计数	.NET Core 3.1
ThreadPool Completed Work Item Count (`threadpool-completed-items-count`)	迄今为止 ThreadPool 中已处理的工作项数	.NET Core 3.1
ThreadPool Queue Length (`threadpool-queue-length`)	ThreadPool 中当前已加入处理队列的工作项数	.NET Core 3.1
ThreadPool Thread Count (`threadpool-thread-count`)	ThreadPool 中当前存在的线程池线程数，基于 ThreadPool.ThreadCount	.NET Core 3.1
Working Set (`working-set`)	某个时间点映射到进程上下文的物理内存量，基于 Environment.WorkingSet	.NET Core 3.1
IL Bytes Jitted (`il-bytes-jitted`)	JIT 编译的 IL 的总大小，以字节为单位	.NET 5
Method Jitted Count (`method-jitted-count`)	JIT 编译的方法数	.NET 5
GC Committed Bytes (`gc-committed-bytes`)	由 GC 所提交的字节数	.NET 6

“Microsoft.AspNetCore.Hosting”计数器

以下计数器作为 ASP.NET Core 的一部分发布，并在 `HostingEventSource.cs` 中进行维护。

[illegible]	[illegible]	[illegible]
Current Requests (`current-requests`)	已启动但尚未停止的请求总数	.NET Core 3.1
Failed Requests (`failed-requests`)	应用生命周期内发生的失败请求总数	.NET Core 3.1
Request Rate (`requests-per-second`)	每个更新间隔发生的请求数	.NET Core 3.1
Total Requests (`total-requests`)	应用生命周期内发生的请求总数	.NET Core 3.1

“Microsoft.AspNetCore.Http.Connections”计数器

以下计数器作为 ASP.NET Core SignalR 的一部分发布，并在 `HttpConnectionsEventSource.cs` 中进行维护。

[illegible]	[illegible]	[illegible]
Average Connection Duration (`connections-duration`)	连接的平均持续时间(毫秒)	.NET Core 3.1

[illegible]	[illegible]	[illegible]
Current Connections (`current-connections`)	已启动但尚未停止的活动连接数	.NET Core 3.1
Total Connections Started (`connections-started`)	已启动的连接总数	.NET Core 3.1
Total Connections Stopped (`connections-stopped`)	已停止的连接总数	.NET Core 3.1
Total Connections Timed Out (`connections-timed-out`)	已超时的连接总数	.NET Core 3.1

“Microsoft-AspNetCore-Server-Kestrel”计数器

以下计数器作为 ASP.NET Core Kestrel Web 服务器的一部分发布，并在 `KestrelEventSource.cs` 中进行维护。

[illegible]	[illegible]	[illegible]
Connection Queue Length (`connection-queue-length`)	连接队列的当前长度	.NET 5
Connection Rate (`connections-per-second`)	每个更新间隔与 Web 服务器的连接数	.NET 5
Current Connections (`current-connections`)	到 Web 服务器的当前活动连接数	.NET 5
Current TLS Handshakes (`current-tls-handshakes`)	当前 TLS 握手数	.NET 5
Current Upgraded Requests (WebSockets) (`current-upgraded-requests`)	当前升级请求数 (WebSockets)	.NET 5
Failed TLS Handshakes (`failed-tls-handshakes`)	失败的 TLS 握手总数	.NET 5
Request Queue Length (`request-queue-length`)	请求队列的当前长度	.NET 5
TLS Handshake Rate (`tls-handshakes-per-second`)	每个更新间隔的 TLS 握手数	.NET 5
Total Connections (`total-connections`)	到 Web 服务器的连接总数	.NET 5
Total TLS Handshakes (`total-tls-handshakes`)	Web 服务器的 TLS 握手总数	.NET 5

“System.Net.Http”计数器

以下计数器由 HTTP 堆栈发布。

[illegible]	[illegible]	[illegible]
Requests Started (`requests-started`)	自进程启动以来启动的请求数	.NET 5
Requests Started Rate (`requests-started-rate`)	每个更新间隔启动的请求数	.NET 5
Requests Failed (`requests-failed`)	自进程启动以来失败的请求数	.NET 5
Requests Failed Rate (`requests-failed-rate`)	每个更新间隔的失败请求数	.NET 5
Current Requests (`current-requests`)	当前已启动但尚未完成或失败的活动 HTTP 请求数	.NET 5
Current HTTP 1.1 Connections (`http11-connections-current-total`)	当前已启动但尚未完成或失败的 HTTP 1.1 连接数	.NET 5
Current HTTP 2.0 Connections (`http20-connections-current-total`)	当前已启动但尚未完成或失败的 HTTP 2.0 连接数	.NET 5
HTTP 1.1 Requests Queue Duration (`http11-requests-queue-duration`)	HTTP 1.1 请求在请求队列中花费的平均持续时间	.NET 5
HTTP 2.0 Requests Queue Duration (`http20-requests-queue-duration`)	HTTP 2.0 请求在请求队列中花费的平均持续时间	.NET 5

“System.Net.NameResolution”计数器

以下计数器跟踪与 DNS 查找相关的指标。

[illegible]	[illegible]	[illegible]
DNS Lookups Requested (`dns-lookups-requested`)	自进程启动以来请求的 DNS 查找数	.NET 5
Average DNS Lookup Duration (`dns-lookups-duration`)	DNS 查找所用的平均时间	.NET 5

“System.Net.Security”计数器

以下计数器跟踪与传输层安全协议相关的指标。

[illegible]	[illegible]	[illegible]
TLS handshakes completed (`tls-handshake-rate`)	每个更新间隔完成的 TLS 握手数	.NET 5
Total TLS handshakes completed (`total-tls-handshakes`)	自进程启动以来完成的 TLS 握手总数	.NET 5

[illegible]	[illegible]	[illegible]
Current TLS handshakes (`current-tls-handshakes`)	当前已启动但尚未完成的 TLS 握手数	.NET 5
Total TLS handshakes failed (`failed-tls-handshakes`)	自进程启动以来失败的 TLS 握手总数	.NET 5
All TLS Sessions Active (`all-tls-sessions-open`)	任何版本的活动 TLS 会话数	.NET 5
TLS 1.0 Sessions Active (`tls10-sessions-open`)	活动的 TLS 1.0 会话数	.NET 5
TLS 1.1 Sessions Active (`tls11-sessions-open`)	活动的 TLS 1.1 会话数	.NET 5
TLS 1.2 Sessions Active (`tls12-sessions-open`)	活动的 TLS 1.2 会话数	.NET 5
TLS 1.3 Sessions Active (`tls13-sessions-open`)	活动的 TLS 1.3 会话数	.NET 5
TLS Handshake Duration (`all-tls-handshake-duration`)	所有 TLS 握手的平均持续时间	.NET 5
TLS 1.0 Handshake Duration (`tls10-handshake-duration`)	TLS 1.0 握手的平均持续时间	.NET 5
TLS 1.1 Handshake Duration (`tls11-handshake-duration`)	TLS 1.1 握手的平均持续时间	.NET 5
TLS 1.2 Handshake Duration (`tls12-handshake-duration`)	TLS 1.2 握手的平均持续时间	.NET 5
TLS 1.3 Handshake Duration (`tls13-handshake-duration`)	TLS 1.3 握手的平均持续时间	.NET 5

“System.Net.Sockets”计数器

以下计数器跟踪与 Socket 相关的指标。

[illegible]	[illegible]	[illegible]
Outgoing Connections Established (`outgoing-connections-established`)	自进程启动以来建立的传出连接总数	.NET 5
Incoming Connections Established (`incoming-connections-established`)	自进程启动以来建立的传入连接总数	.NET 5
Bytes Received (`bytes-received`)	自进程启动以来收到的字节总数	.NET 5

[illegible]	[illegible]	[illegible]
Bytes Sent (`bytes-sent`)	自进程启动以来发送的字节总数	.NET 5
Datagrams Received (`datagrams-received`)	自流程启动以来收到的数据报总数	.NET 5
Datagrams Sent (`datagrams-sent`)	自流程启动以来发送的数据报总数	.NET 5

教程：使用 .NET Core 中的 EventCounters 衡量性能

2021/11/16 •

本文适用于： ✔ .NET Core 3.0 SDK 及更高版本

本教程将介绍如何使用 EventCounter 衡量高频率事件的性能。可以使用由各种官方 .NET Core 包或第三方提供者发布的可用的计数器，或创建自己的监视指标。

在本教程中，将：

- 实现 EventSource。
- 利用 dotnet-counters 监视计数器。

先决条件

本教程使用：

- .NET Core 3.1 SDK 或更高版本。
- dotnet-counters 监视事件计数器。
- 要诊断的示例调试目标应用。

获取源

示例应用程序将用作监视的基础。示例浏览器中提供了示例 ASP.NET Core 存储库。下载 zip 文件，下载后提取它，并在你喜欢的 IDE 中打开它。生成并运行应用程序以确保它正常工作，然后停止应用程序。

实现 EventSource

对于每隔几毫秒发生的事件，最好使每个事件的开销较低（小于一毫秒）。否则，对性能的影响将很大。记录事件意味着你将向磁盘写入内容。如果磁盘不够快，你将丢失事件。你需要一个解决方案，而不是记录事件本身。

在处理大量事件时，了解每个事件的度量值也无济于事。大多数时候，你只需要一些统计信息。因此，你可以在进程本身中获取统计信息，然后偶尔编写一个事件来报告统计信息，这是 EventCounter 将执行的操作。

下面是有关如何实现 System.Diagnostics.Tracing.EventSource 的示例。创建名为 MinimalEventCounterSource.cs 的新文件，并使用代码片段作为其源：

```
using System.Diagnostics.Tracing;

[EventSource(Name = "Sample.EventCounter.Minimal")]
public sealed class MinimalEventCounterSource : EventSource
{
    public static readonly MinimalEventCounterSource Log = new MinimalEventCounterSource();

    private EventCounter _requestCounter;

    private MinimalEventCounterSource() =>
        _requestCounter = new EventCounter("request-time", this)
        {
            DisplayName = "Request Processing Time",
            DisplayUnits = "ms"
        };

    public void Request(string url, long elapsedMilliseconds)
    {
        WriteEvent(1, url, elapsedMilliseconds);
        _requestCounter?.WriteMetric(elapsedMilliseconds);
    }

    protected override void Dispose(bool disposing)
    {
        _requestCounter?.Dispose();
        _requestCounter = null;

        base.Dispose(disposing);
    }
}
```

EventSource.WriteEvent 行是 EventSource 部分，而不是 EventCounter 的一部分，编写它是为了表明你可以一起记录消息和事件计数器。

添加操作筛选器

示例源代码是 ASP.NET Core 项目。可以全局添加将记录总请求时间的操作筛选器。创建名为 LogRequestTimeFilterAttribute.cs 的新文件，并使用以下代码：

```
using System.Diagnostics;
using Microsoft.AspNetCore.Http.Extensions;
using Microsoft.AspNetCore.Mvc.Filters;

namespace DiagnosticScenarios
{
    public class LogRequestTimeFilterAttribute : ActionFilterAttribute
    {
        readonly Stopwatch _stopwatch = new Stopwatch();

        public override void OnActionExecuting(ActionExecutingContext context) => _stopwatch.Start();

        public override void OnActionExecuted(ActionExecutedContext context)
        {
            _stopwatch.Stop();

            MinimalEventCounterSource.Log.Request(
                context.HttpContext.Request.GetDisplayUrl(), _stopwatch.ElapsedMilliseconds);
        }
    }
}
```

操作筛选器在请求开始时启动 Stopwatch，并在其完成后停止，捕获运行时间。总毫秒数记录到 `MinimalEventCounterSource` 单一实例。为了应用此筛选器，需要将其添加到筛选器集合。在 Startup.cs 文件中，

更新包含此筛选器的 `ConfigureServices` 方法。

```
public void ConfigureServices(IServiceCollection services) =>
    services.AddControllers(options => options.Filters.Add<LogRequestTimeFilterAttribute>())
            .AddNewtonsoftJson();
```

监视事件计数器

通过 EventSource 上的实现和自定义操作筛选器，生成和启动应用程序。你已将指标记录到 EventCounter 中，但除非你从其中访问统计信息，否则它将不起作用。要获取统计信息，需要通过创建以所需事件频率触发的计时器来启用 EventCounter，并启用侦听器来捕获事件。为此，可以使用 dotnet-counters。

使用 dotnet-counters ps 命令来显示可监视的 .NET 进程的列表。

```
dotnet-counters ps
```

通过使用 `dotnet-counters ps` 命令的输出中的进程标识符，你可以使用以下 `dotnet-counters monitor` 命令开始监视事件计数器：

```
dotnet-counters monitor --process-id 2196 --counters
Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-requests,requests-per-
second],System.Runtime[cpu-usage]
```

当 `dotnet-counters monitor` 命令正在运行时，请在浏览器上按住 F5，以开始向 `https://localhost:5001/api/values` 终结点发出连续请求。几秒后按 q 以停止

```
Press p to pause, r to resume, q to quit.
    Status: Running

[Microsoft.AspNetCore.Hosting]
    Request Rate / 1 sec                               9
    Total Requests                                   134
[System.Runtime]
    CPU Usage (%)                                     13
[Sample.EventCounter.Minimal]
    Request Processing Time (ms)                    34.5
```

`dotnet-counters monitor` 命令非常适合主动监视。不过，可以收集这些诊断指标以便进行后续处理和分析。为此，请使用 `dotnet-counters collect` 命令。`collect` 开关命令类似于 `monitor` 命令，但接受几个其他参数。你可以指定所需的输出文件名和格式。对于名为 diagnostics.json 的 JSON 文件，请使用以下命令：

```
dotnet-counters collect --process-id 2196 --format json -o diagnostics.json --counters
Sample.EventCounter.Minimal,Microsoft.AspNetCore.Hosting[total-requests,requests-per-
second],System.Runtime[cpu-usage]
```

再一次，当命令正在运行时，在浏览器上按住 F5，以开始向 `https://localhost:5001/api/values` 终结点发出连续请求。几秒后按 q 以停止。写入 diagnostics.json 文件。写入的 JSON 文件不会缩进；但为了提升可读性，在这里进行了缩进。

```
{
  "TargetProcess": "DiagnosticScenarios",
  "StartTime": "8/5/2020 3:02:45 PM",
  "Events": [
    {
      "timestamp": "2020-08-05 15:02:47Z",
```

```
  "provider": "System.Runtime",
  "name": "CPU Usage (%)",
  "counterType": "Metric",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:47Z",
  "provider": "Microsoft.AspNetCore.Hosting",
  "name": "Request Rate / 1 sec",
  "counterType": "Rate",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:47Z",
  "provider": "Microsoft.AspNetCore.Hosting",
  "name": "Total Requests",
  "counterType": "Metric",
  "value": 134
},
{
  "timestamp": "2020-08-05 15:02:47Z",
  "provider": "Sample.EventCounter.Minimal",
  "name": "Request Processing Time (ms)",
  "counterType": "Metric",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:47Z",
  "provider": "System.Runtime",
  "name": "CPU Usage (%)",
  "counterType": "Metric",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:48Z",
  "provider": "Microsoft.AspNetCore.Hosting",
  "name": "Request Rate / 1 sec",
  "counterType": "Rate",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:48Z",
  "provider": "Microsoft.AspNetCore.Hosting",
  "name": "Total Requests",
  "counterType": "Metric",
  "value": 134
},
{
  "timestamp": "2020-08-05 15:02:48Z",
  "provider": "Sample.EventCounter.Minimal",
  "name": "Request Processing Time (ms)",
  "counterType": "Metric",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:48Z",
  "provider": "System.Runtime",
  "name": "CPU Usage (%)",
  "counterType": "Metric",
  "value": 0
},
{
  "timestamp": "2020-08-05 15:02:50Z",
  "provider": "Microsoft.AspNetCore.Hosting",
  "name": "Request Rate / 1 sec",
  "counterType": "Rate",
  "value": 0
},
{
```

```
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Microsoft.AspNetCore.Hosting",
      "name": "Total Requests",
      "counterType": "Metric",
      "value": 134
    },
    {
      "timestamp": "2020-08-05 15:02:50Z",
      "provider": "Sample.EventCounter.Minimal",
      "name": "Request Processing Time (ms)",
      "counterType": "Metric",
      "value": 0
    }
  ]
}
```


后续步骤

EventCounters

符号

2021/11/16 •

符号可用于调试和其他诊断工具。符号文件的内容在语言、编译器和平台之间各有不同。以非常概要的角度来看，符号是源代码和编译器生成的二进制文件之间的映射。Visual Studio 和 Visual Studio Code 等工具会使用这些映射来解析源行号信息或本地变量名称。

可在有关符号的 Windows 文档中更详细地了解适用于 Windows 的符号，不过其中很多概念也不用于其他平台。

了解 .NET 的可移植 PDB 格式

.NET Core 引入了一种新的符号文件 (PDB) 格式，即可移植 PDB。与仅限 Windows 的传统 PDB 不同，可在任意平台上创建和读取可移植 PDB。

什么是 PDB？

PDB 文件是编译器生成的辅助文件，目的是向其他工具（尤其是调试程序）提供主可执行文件中的内容及其生成方式的相关信息。例如，调试程序读取 PDB 来将 foo.cs 第 12 行映射到适当的可执行文件位置，以便它可设置断点。Windows PDB 格式已存在很长时间，它是从甚至更久远的其他本机调试符号格式演变而来的。它最初是用作本机 (C/C++) 程序的一种格式。针对 .NET Framework 的首次发布，Windows PDB 格式进行了扩展以支持 .NET。

使用适合你的方案的 PDB 格式

任何位置都不支持可移植 PDB 和 Windows PDB，因此你需要思考要在哪里使用和调试你的项目来确定要使用的具体格式。如果你有一个项目，而你希望它能在这两种格式中使用和调试，那么可使用不同的生成配置，并生成项目两次以同时支持这两种类型的使用者。

支持可移植 PDB

可在任意操作系统上读取可移植 PDB，建议对托管代码使用这种符号格式，但存在一些旧版工具和应用程序不支持这种格式：

- 面向 .NET Framework 4.7.1 或更低版本的应用程序：将带有映射的堆栈跟踪打印回行号（例如在 ASP.NET 错误页面中）。方法的名称不受影响，只有源文件名和行号不受支持。
- 使用 .NET 反编译程序（例如 ildasm 或 .NET 反射器），且预期看到源行映射或本地参数名称。
- 最新版本的 DIA 和工具用它来读取符号（例如 WinDBG 支持可移植 PDB），但更低的版本不这样做。
- 可能存在不支持可移植 PDB 的更低版本的探查器。

若要在不支持可移植 PDB 的工具上使用这些格式，可尝试使用 Pdb2Pdb，它会在可移植 PDB 和 Windows PDB 之间进行转换。

支持 Windows PDB

仅可在 Windows 上编写或读取 Windows PDB。对托管代码使用 Windows PDB 的操作已过时，且只有旧版工具需要此操作。建议使用可移植 PDB 而不是 Windows PDB，原因是一些更新的编译器功能仅支持可移植 PDB。

另请参阅

- dotnet-symbol 可用于下载框架二进制文件的符号文件
- 有关符号的 Windows 文档

诊断客户端库

2021/11/16 •

本文适用于：✔ 针对目标应用的 .NET Core 3.0 SDK 和更高版本及要使用该库的 .NET Standard 2.0。

Microsoft.Diagnostics.NETCore.Client（也称为诊断客户端库）是一种托管库，该库支持你与 .NET Core 运行时 (CoreCLR) 交互，以实现各种与诊断相关的任务，如通过 EventPipe 进行跟踪、请求转储或附加 ICorProfiler。此库是 `dotnet-counters`、`dotnet-trace`、`dotnet-gcdump` 和 `dotnet-dump` 等许多诊断工具背后的支持库。使用此库，可以为自己编写针对特定方案自定义的诊断工具。

你可以通过将 `PackageReference` 添加到你的项目来获取 Microsoft.Diagnostics.NETCore.Client。该包托管在 `NuGet.org` 之上。

以下部分中的示例演示了如何使用 Microsoft.Diagnostics.NETCore.Client 库。其中一些示例还演示了如何使用 TraceEvent 库来分析事件负有效负载。

附加到进程并打印出所有 GC 事件

此代码片段演示了如何在信息级别使用带有 GC 关键字的 .NET 运行时提供程序启动 EventPipe 会话。它还演示了如何使用 TraceEvent 库提供的 `EventPipeEventSource` 类来分析传入的事件，并将其名称实时打印到控制台。

```
using Microsoft.Diagnostics.NETCore.Client;
using Microsoft.Diagnostics.Tracing;
using Microsoft.Diagnostics.Tracing.EventPipe;
using Microsoft.Diagnostics.Tracing.Parsers;
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;

public class RuntimeGCEventsPrinter
{
    public static void PrintRuntimeGCEvents(int processId)
    {
        var providers = new List<EventPipeProvider>()
        {
            new EventPipeProvider("Microsoft-Windows-DotNETRuntime",
                EventLevel.Informational, (long)ClrTraceEventParser.Keywords.GC)
        };

        var client = new DiagnosticsClient(processId);
        using (EventPipeSession session = client.StartEventPipeSession(providers, false))
        {
            var source = new EventPipeEventSource(session.EventStream);

            source.Clr.All += (TraceEvent obj) => Console.WriteLine(obj.ToString());

            try
            {
                source.Process();
            }
            catch (Exception e)
            {
                Console.WriteLine("Error encountered while processing events");
                Console.WriteLine(e.ToString());
            }
        }
    }
}
```

编写核心转储

此示例演示了如何使用 `DiagnosticsClient` 触发核心转储的集合。

```
using Microsoft.Diagnostics.NETCore.Client;

public partial class Dumper
{
    public static void TriggerCoreDump(int processId)
    {
        var client = new DiagnosticsClient(processId);
        client.WriteDump(DumpType.Normal, "/tmp/minidump.dmp");
    }
}
```

当 CPU 使用率超过阈值时触发核心转储

此示例演示了如何监视 .NET 运行时发布的 `cpu-usage` 计数器，并在 CPU 使用率超出特定阈值时请求转储。

```
using Microsoft.Diagnostics.NETCore.Client;
using Microsoft.Diagnostics.Tracing;
using Microsoft.Diagnostics.Tracing.EventPipe;
using Microsoft.Diagnostics.Tracing.Parsers;
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;

public partial class Dumper
{
    public static void TriggerDumpOnCpuUsage(int processId, int threshold)
    {
        var providers = new List<EventPipeProvider>()
        {
            new EventPipeProvider(
                "System.Runtime",
                EventLevel.Informational,
                (long)ClrTraceEventParser.Keywords.None,
                new Dictionary<string, string>
                {
                    ["EventCounterIntervalSec"] = "1"
                }
            )
        };
        var client = new DiagnosticsClient(processId);
        using (var session = client.StartEventPipeSession(providers))
        {
            var source = new EventPipeEventSource(session.EventStream);
            source.Dynamic.All += (TraceEvent obj) =>
            {
                if (obj.EventName.Equals("EventCounters"))
                {
                    var payloadVal = (IDictionary<string, object>)(obj.PayloadValue(0));
                    var payloadFields = (IDictionary<string, object>)(payloadVal["Payload"]);
                    if (payloadFields["Name"].ToString().Equals("cpu-usage"))
                    {
                        double cpuUsage = Double.Parse(payloadFields["Mean"].ToString());
                        if (cpuUsage > (double)threshold)
                        {
                            client.WriteDump(DumpType.Normal, "/tmp/minidump.dmp");
                        }
                    }
                }
            };
            try
            {
                source.Process();
            }
            catch (Exception) {}
        }
    }
}
```

在给定的秒数内触发 CPU 跟踪

此示例演示了如何使用默认 CLR 跟踪关键字以及示例探查器触发特定时间段的 EventPipe 会话。此后，它会读取输出流，并将这些字节写入文件。本质上讲，这就是 `dotnet-trace` 在内部用于编写跟踪文件的内容。

```
using Microsoft.Diagnostics.Tracing;
using Microsoft.Diagnostics.Tracing.Parsers;
using Microsoft.Diagnostics.NETCore.Client;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Diagnostics.Tracing;
using System.IO;
using System.Threading.Tasks;

public partial class Tracer
{
    public void TraceProcessForDuration(int processId, int duration, string traceName)
    {
        var cpuProviders = new List<EventPipeProvider>()
        {
            new EventPipeProvider("Microsoft-Windows-DotNETRuntime", EventLevel.Informational,
(long)ClrTraceEventParser.Keywords.Default),
            new EventPipeProvider("Microsoft-DotNETCore-SampleProfiler", EventLevel.Informational,
(long)ClrTraceEventParser.Keywords.None)
        };
        var client = new DiagnosticsClient(processId);
        using (var traceSession = client.StartEventPipeSession(cpuProviders))
        {
            Task copyTask = Task.Run(async () =>
            {
                using (FileStream fs = new FileStream(traceName, FileMode.Create, FileAccess.Write))
                {
                    await traceSession.EventStream.CopyToAsync(fs);
                }
            });

            Task.WhenAny(copyTask, Task.Delay(TimeSpan.FromMilliseconds(duration * 1000)));
            traceSession.Stop();
        }
    }
}
```

打印已发布诊断通道的进程名称

此示例演示了如何使用 `DiagnosticsClient.GetPublishedProcesses` API 来打印发布了诊断 IPC 通道的 .NET 进程名称。

```
using Microsoft.Diagnostics.NETCore.Client;
using System;
using System.Diagnostics;
using System.Linq;

public class ProcessTracker
{
    public static void PrintProcessStatus()
    {
        var processes = DiagnosticsClient.GetPublishedProcesses()
            .Select(Process.GetProcessById)
            .Where(process => process != null);

        foreach (var process in processes)
        {
            Console.WriteLine($"{process.ProcessName}");
        }
    }
}
```

实时分析事件

此示例演示了一个示例，在该示例中，我们创建了两个任务，一个任务用于通过 `EventPipeEventSource` 分析实时传入的事件，另一个任务用于读取用户输入的控制台输入信号，以使程序结束。如果在用户按 Enter 之前目标应用已经存在，则应用将正常存在。否则，`inputTask` 会将 Stop 命令发送到管道并正常退出。

```
using Microsoft.Diagnostics.NETCore.Client;
using Microsoft.Diagnostics.Tracing;
using Microsoft.Diagnostics.Tracing.EventPipe;
using Microsoft.Diagnostics.Tracing.Parsers;
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using System.Threading.Tasks;

public partial class Tracer
{
    public static void PrintEventsLive(int processId)
    {
        var providers = new List<EventPipeProvider>()
        {
            new EventPipeProvider("Microsoft-Windows-DotNETRuntime",
                EventLevel.Informational, (long)ClrTraceEventParser.Keywords.Default)
        };
        var client = new DiagnosticsClient(processId);
        using (var session = client.StartEventPipeSession(providers, false))
        {

            Task streamTask = Task.Run(() =>
            {
                var source = new EventPipeEventSource(session.EventStream);
                source.Clr.All += (TraceEvent obj) => Console.WriteLine(obj.EventName);
                try
                {
                    source.Process();
                }
                // NOTE: This exception does not currently exist. It is something that needs to be added to
TraceEvent.
                catch (Exception e)
                {
                    Console.WriteLine("Error encountered while processing events");
                    Console.WriteLine(e.ToString());
                }
            });

            Task inputTask = Task.Run(() =>
            {
                Console.WriteLine("Press Enter to exit");
                while (Console.ReadKey().Key != ConsoleKey.Enter)
                {
                    Task.Delay(TimeSpan.FromMilliseconds(100));
                }
                session.Stop();
            });

            Task.WaitAny(streamTask, inputTask);
        }
    }
}
```

附加 ICorProfiler 探查器

此示例演示了如何通过探查器附加将 ICorProfiler 附加到进程。

```
using System;
using Microsoft.Diagnostics.NETCore.Client;

public class Profiler
{
    public static void AttachProfiler(int processId, Guid profilerGuid, string profilerPath)
    {
        var client = new DiagnosticsClient(processId);
        client.AttachProfiler(TimeSpan.FromSeconds(10), profilerGuid, profilerPath);
    }
}
```

Microsoft.Diagnostics.NETCore.Client API

2021/11/16 •

本部分介绍诊断客户端库 API。

DiagnosticsClient 类

```
public DiagnosticsClient
{
    public DiagnosticsClient(int processId);

    public EventPipeSession StartEventPipeSession(
        IEnumerable<EventPipeProvider> providers,
        bool requestRundown = true,
        int circularBufferMB = 256);

    public void WriteDump(
        DumpType dumpType,
        string dumpPath,
        bool logDumpGeneration = false);

    public void AttachProfiler(
        TimeSpan attachTimeout,
        Guid profilerGuid,
        string profilerPath,
        byte[] additionalData = null);

    public static IEnumerable<int> GetPublishedProcesses();
}
```

构造函数

```
public DiagnosticsClient(int processId);
```

为使用进程 ID `processId` 运行的兼容 .NET 进程创建新实例 `DiagnosticsClient`。

`processID`：目标应用程序的进程 ID。

StartEventPipeSession 方法

```
public EventPipeSession StartEventPipeSession(
    IEnumerable<EventPipeProvider> providers,
    bool requestRundown = true,
    int circularBufferMB = 256)
```

使用给定提供程序和设置启动 EventPipe 跟踪会话。

- `providers`：要开始跟踪的 `EventPipeProvider` 的 `IEnumerable`。
- `requestRundown`：一个 `bool`，用于指定是否应请求来自目标应用运行时的断开提供程序事件。
- `circularBufferMB`：一个 `int`，用于指定目标应用运行时在收集事件时使用的循环缓冲区的总大小。

```
public EventPipeSession StartEventPipeSession(EventPipeProvider providers, bool requestRundown=true, int
circularBufferMB=256)
```

- `providers`：要开始跟踪的 `EventPipeProvider`。
- `requestRundown`：一个 `bool`，用于指定是否应请求来自目标应用运行时的断开提供程序事件。
- `circularBufferMB`：一个 `int`，用于指定目标应用运行时在收集事件时使用的循环缓冲区的总大小。

> **NOTE**
> 断开事件包含可能需要用于后期分析的有效负载，如解析线程示例的方法名称。建议将此值设置为 true，除非你不希望出现此行为。在大型应用程序中，这可能需要一段时间。

- `circularBufferMB`：要在运行时用作写入事件的缓冲区的循环缓冲区的大小。

WriteDump 方法

```
public void WriteDump(DumpType dumpType, string dumpPath, bool logDumpGeneration=false);
```

请求转储以事后调试目标应用程序。可以使用 `DumpType` 枚举指定转储类型。

- `dumpType`：要请求的转储类型。
- `dumpPath`：要写出到其中的转储的路径。
- `logDumpGeneration`：如果设置为 `true`，则目标应用程序将在转储生成期间写出诊断日志。

AttachProfiler 方法

```
public void AttachProfiler(TimeSpan attachTimeout, Guid profilerGuid, string profilerPath, byte[]
additionalData=null);
```

请求将 ICorProfiler 附加到目标应用程序。

- `attachTimeout`：将在其后中止附加的 `TimeSpan`。
- `profilerGuid`：要附加的 ICorProfiler 的 `Guid`。
- `profilerPath`：要附加的 ICorProfiler dll 的路径。
- `additionalData`：在探查器附加过程中可传递给运行时的可选其他数据。

GetPublishedProcesses 方法

```
public static IEnumerable<int> GetPublishedProcesses();
```

获取可附加到 `IEnumerable` 所有活动 .NET 进程的进程 ID。

EventPipeProvider 类

```
public class EventPipeProvider
{
    public EventPipeProvider(
        string name,
        EventLevel eventLevel,
        long keywords = 0,
        IDictionary<string, string> arguments = null)

    public string Name { get; }

    public EventLevel EventLevel { get; }

    public long Keywords { get; }

    public IDictionary<string, string> Arguments { get; }

    public override string ToString();

    public override bool Equals(object obj);

    public override int GetHashCode();

    public static bool operator ==(Provider left, Provider right);

    public static bool operator !=(Provider left, Provider right);
}
```

构造函数

```
public EventPipeProvider(
    string name,
    EventLevel eventLevel,
    long keywords = 0,
    IDictionary<string, string> arguments = null)
```

创建具有给定提供程序名称、EventLevel、关键字和参数的新 `EventPipeProvider` 实例。

Name 属性

```
public string Name { get; }
```

获取提供程序的名称。

EventLevel 属性

```
public EventLevel EventLevel { get; }
```

获取 `EventPipeProvider` 的给定实例的 `EventLevel`。

Keywords 属性

```
public long Keywords { get; }
```

获取一个代表 `EventSource` 的关键字的位掩码的值。

Arguments 属性

```
public IDictionary<string, string> Arguments { get; }
```

获取键/值对字符串的 `IDictionary`，这些字符串代表要传递给代表给定 `EventPipeProvider` 的 `EventSource` 的可选参数。

备注

此类不可变，因为 EventPipe 不允许在截至 .NET Core 3.1 的 EventPipe 会话过程中修改提供程序的配置。

EventPipeSession 类

```
public class EventPipeSession : IDisposable
{
    public Stream EventStream { get; }
    public void Stop();
}
```

此类表示正在进行的 EventPipe 会话。它是不可变的，充当给定运行时的 EventPipe 会话的句柄。

EventStream 属性

```
public Stream EventStream { get; }
```

获取可用于读取事件流的 `Stream`。

Stop 方法

```
public void Stop();
```

停止给定的 `EventPipe` 会话。

DumpType 枚举

```
public enum DumpType
{
    Normal = 1,
    WithHeap = 2,
    Triage = 3,
    Full = 4
}
```

代表可以请求的转储类型。

- `Normal`：只包含为进程中的所有现有线程的所有现有跟踪捕获堆栈跟踪所需的信息。有限的 GC 堆内存和信息。
- `WithHeap`：包含 GC 堆以及为进程中的所有现有线程捕获堆栈跟踪所需的信息。
- `Triage`：只包含为进程中的所有现有线程的所有现有跟踪捕获堆栈跟踪所需的信息。有限的 GC 堆内存和信息。
- `Full`：包含进程中的所有可访问内存。原始内存数据包含在末尾，因此无需原始内存信息即可直接映射初始结构。此选项可能会导致转储文件非常大。

异常

从库中引发的异常的类型为 `DiagnosticsClientException` 或派生类型。

```
public class DiagnosticsClientException : Exception
```

UnsupportedCommandException

```
public class UnsupportedCommandException : DiagnosticsClientException
```

当库或目标进程的运行时不支持该命令时，可能会引发此异常。

UnsupportedProtocolException

```
public class UnsupportedProtocolException : DiagnosticsClientException
```

当目标进程的运行时与库使用的诊断 IPC 协议不兼容时，可能会引发此异常。

ServerNotAvailableException

```
public class ServerNotAvailableException : DiagnosticsClientException
```

当运行时不可用于诊断 IPC 命令（例如在运行时尚未准备好执行诊断命令的运行时启动早期）或运行时关闭时，可能会引发此异常。

ServerErrorException

```
public class ServerErrorException : DiagnosticsClientException
```

当运行时错误响应给定命令时，可能会引发此异常。

.NET 运行时事件

2021/11/16 •

.NET 运行时 (CoreCLR) 发出各种事件，这些事件可用于诊断 .NET 应用程序的问题，并且可通过各种机制（例如 `ETW`、`LTTng` 和 `EventPipe`）来使用。

本文档作为对 .NET Core 运行时触发的事件的参考。

有关 .NET Framework 中的运行时事件，请参阅 CLR ETW 事件。

在本节中

争用事件

这些事件收集有关监视器锁争用的信息。

垃圾回收事件

这些事件可收集有关垃圾回收的信息。它们可帮助进行诊断和调试，包括确定垃圾回收执行的次数、垃圾回收期间释放的内存量等。

异常事件

这些运行时事件捕获有关引发的异常的信息。

互操作事件

这些运行时事件捕获有关公共中间语言 (CIL) 存根生成的信息。

加载器和绑定器事件

这些事件收集有关加载和卸载程序集和模块的信息。

方法事件

这些事件收集特定于方法的信息。符号解析需要这些事件的负载。此外，这些事件还提供如调用方法的次数等有用信息。

线程事件

这些事件收集有关工作线程和 I/O 线程的信息。

类型事件

这些事件收集有关类型系统的信息。

.NET 运行时争用事件

2021/11/16 •

这些运行时事件使用 `Monitor.Enter` 或 C# 锁关键字等捕获有关监视器锁争用的信息。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

ContentionStart_V1 事件

在监视器锁争用开始时发出此事件。

[illegible]	LEVEL
`ContentionKeyword` (0x4000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ContentionStart_V1`	81	监视器锁争用开始。

[illegible]	[illegible]	[illegible]
`Flags`	`win:UInt8`	`0` 表示托管；`1` 表示本机。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ContentionStop_V1 事件

在监视器锁争用结束时发出此事件。

[illegible]	LEVEL
`ContentionKeyword` (0x4000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ContentionStop_V1`	91	监视器锁争用结束。

[illegible]	[illegible]	[illegible]
`Flags`	`win:UInt8`	`0` 表示托管；`1` 表示本机。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。
`DurationNs`	`win:Double`	争用的持续时间（以纳秒为单位）。

.NET 运行时异常事件

2021/11/16 •

这些运行时事件捕获有关引发的异常的信息。有关如何将这些事件用于诊断的详细信息，请参阅 .NET 应用程序日志记录和跟踪

ExceptionThrown_V1 事件

关键字	LEVEL
`ExceptionKeyword` (0x8000)	错误 (1)

下表显示了事件信息。

版本	事件 ID	说明
`ExceptionThrown_V1`	80	引发托管异常。

属性	类型	说明
`ExceptionType`	`win:UnicodeString`	异常的类型，例如，`System.NullReferenceException`。
`ExceptionMessage`	`win:UnicodeString`	实际的异常消息。
`EIPCodeThrow`	`win:Pointer`	指向异常发生位置的指令指针。
`ExceptionHR`	`win:UInt32`	异常 HRESULT。
`ExceptionFlags`	`win:UInt16`	`0x01`：HasInnerException。 `0x02`：IsNestedException。 `0x04`：IsRethrownException。 `0x08`：IsCorruptedStateException（指示进程状态已损坏，请参阅处理损坏状态异常）。 `0x10`：IsCLSCompliant（从 Exception 派生的异常符合 CLS，此外的其他异常均不符合 CLS）。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ExceptionCatchStart 事件

当托管异常 catch 处理程序开始时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionCatchStart`	250	托管异常由运行时处理。

[illegible]	[illegible]	[illegible]
`EIPCodeThrow`	`win:Pointer`	指向异常发生位置的指令指针。
`MethodID`	`win:Pointer`	指向发生异常的方法上的方法描述符的指针。
`MethodName`	`win:UnicodeString`	发生异常的方法的名称。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ExceptionCatchStop 事件

当托管异常 catch 处理程序结束时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionCatchStop`	251	托管异常 catch 处理程序已完成。

ExceptionFinallyStart 事件

当托管异常 finally 处理程序开始时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionFinallyStart`	252	托管异常由运行时处理。

[illegible]	[illegible]	[illegible]
`EIPCodeThrow`	`win:Pointer`	指向异常发生位置的指令指针。
`MethodID`	`win:Pointer`	指向发生异常的方法上的方法描述符的指针。
`MethodName`	`win:UnicodeString`	发生异常的方法的名称。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ExceptionFinallyStop 事件

当托管异常 catch 处理程序结束时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionFinallyStop`	253	托管异常 finally 处理程序已完成。

ExceptionFilterStart 事件

当托管异常筛选开始时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionFilterStart`	254	托管异常筛选开始。

[illegible]	[illegible]	[illegible]
`EIPCodeThrow`	`win:Pointer`	指向异常发生位置的指令指针。
`MethodID`	`win:Pointer`	指向发生异常的方法上的方法描述符的指针。
`MethodName`	`win:UnicodeString`	发生异常的方法的名称。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ExceptionFilterStop 事件

当托管异常筛选结束时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionFilteringStart`	255	托管异常筛选结束。

ExceptionThrownStop 事件

当运行时完成处理引发的托管异常时，将发出此事件。

[illegible]	LEVEL
`ExceptionKeyword` (0x8000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ExceptionThrownStop`	256	托管异常筛选结束。

.NET 运行时垃圾回收事件

2021/11/16 •

这些事件可收集有关垃圾回收的信息。它们有助于诊断和调试，包括确定执行垃圾回收的次数、垃圾回收期间释放的内存量等。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

GCStart_V2 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCStart_V1`	1	垃圾回收已启动。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt32`	第 *n* 代垃圾回收。
`Depth`	`win:UInt32`	正在回收的代。
`Reason`	`win:UInt32`	垃圾回收的触发原因： `0x0` - 小型对象堆分配。 `0x1` - 已引发。 `0x2` - 内存不足。 `0x3` - 空。 `0x4` - 大型对象堆分配。 `0x5` - 空间外（针对小型对象堆）。 `0x6` - 空间外（针对大型对象堆）。 `0x7` - 已引发，但不是强制作为阻塞。

[illegible]	[illegible]	[illegible]
`Type`	`win:UInt32`	`0x0` - 后台垃圾回收外发生了阻止垃圾回收。 `0x1` - 后台垃圾回收。 `0x2` - 后台垃圾回收时发生了阻止垃圾回收。
`ClrInstanceID`	win:UInt16	CoreCLR 实例的唯一 ID。

GCEnd_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCEnd_V1`	2	垃圾回收已结束。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt32`	第 *n* 代垃圾回收。
`Depth`	`win:UInt32`	已回收的代。
`ClrInstanceID`	win:UInt16	CoreCLR 实例的唯一 ID。

GCHeapStats_V2 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCHeapStats_V2`	4	在每次垃圾回收结束时显示堆统计信息。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`GenerationSize0`	`win:UInt64`	第 0 代内存的大小(以字节为单位)。
`TotalPromotedSize0`	`win:UInt64`	从第 0 代提升到第 1 代的字节数。
`GenerationSize1`	`win:UInt64`	第 1 代内存的大小(以字节为单位)。
`TotalPromotedSize1`	`win:UInt64`	从第 1 代提升到第 2 代的字节数。
`GenerationSize2`	`win:UInt64`	第 2 代内存的大小(以字节为单位)。
`TotalPromotedSize2`	`win:UInt64`	上次回收后仍存在于第 2 代中的字节数。
`GenerationSize3`	`win:UInt64`	大型对象堆的大小(以字节为单位)。
`TotalPromotedSize3`	`win:UInt64`	上次回收后仍存在于大型对象堆中的字节数。
`FinalizationPromotedSize`	`win:UInt64`	准备终结的对象的总大小(以字节为单位)。
`FinalizationPromotedCount`	`win:UInt64`	已准备好进行终结的对象的数目。
`PinnedObjectCount`	`win:UInt32`	固定(不可移动)对象的数目。
`SinkBlockCount`	`win:UInt32`	正在使用的同步块的数目。
`GCHandleCount`	`win:UInt32`	使用中的垃圾回收句柄的数目。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。
`GenerationSize4`	`win:UInt64`	已固定对象堆的大小(以字节为单位)。
`TotalPromotedSize4`	`win:UInt64`	上次回收后仍存在于固定对象堆中的字节数。

GCCreateSegment_V1 事件

下表显示了关键字和级别:

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息:

[illegible]	[illegible] ID	[illegible]

[illegible]	[illegible] ID	[illegible]
`GCCreateSegment_V1`	5	已创建的新的垃圾回收段。此外，当在已运行的进程中启用跟踪时，将为每个现有段引发此事件。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Address`	`win:UInt64`	段的地址。
`Size`	`win:UInt64`	段的大小。
`Type`	`win:UInt32`	0x0 - 小型对象堆。 0x0 - 大型对象堆。 0x2 - 只读堆。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

请注意，由垃圾回收器分配的段的大小特定于实现，且随时可能更改（包括定期更新）。应用程序不应假设特定段的大小或依赖于此大小，也不应尝试配置段分配可用的内存量。

GCFreeSegment_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCFreeSegment_V1`	6	已释放垃圾回收段。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Address`	`win:UInt64`	段的地址。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

GCRestartEEBegin_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCRestartEEBegin_V1`	7	已开始从公共语言运行时挂起进行恢复。

此事件没有任何事件数据。

GCRestartEEEnd_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCRestartEEEnd_V1`	3	从公共语言运行时挂起进行的恢复已结束。

此事件没有任何事件数据。

GCSuspendEEEnd_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCSuspendEEEnd_V1`	8	结束垃圾回收执行引擎的挂起。

此事件没有任何事件数据。

GCSuspendEEBegin_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

□□	□□ ID	□□□□□□□□□
`GCSuspendEEBegin_V1`	8	开始挂起垃圾回收的执行引擎。

□□□	□□□□	□□
`Count`	`win:UInt32`	第 *n* 代垃圾回收。
`Reason`	`win:UInt32`	EE 挂起的原因。 `0x0`：因其他事件而暂停 `0x1`：因 GC 而暂停。 `0x2`：因 AppDomain 关闭而暂停。 `0x3`：因代码间距调整而暂停。 `0x4`：因关闭而暂停。 `0x5`：因调试器而暂停。 `0x6`：因 GC 准备而暂停。 `0x7`：因调试程序扫描而暂停

GCAllocationTick_V3 事件

下表显示了关键字和级别：

□□□□□□□	LEVEL
`GCKeyword` (0x1)	详细 (4)

下表显示了事件信息：

□□	□□ ID	□□□□□□□□□
`GCAllocationTick_V3`	10	每次大约分配 100 KB。

下表显示了事件数据：

□□□	□□□□	□□
`AllocationAmount`	`win:UInt32`	分配大小（以字节为单位）。对于小于 ULONG（4,294,967,295 字节）长度的分配，此值为精确值。如果分配长度更大，则此字段包含了截断的值。对于非常大的分配使用 `AllocationAmount64`。

[illegible]	[illegible]	[illegible]
`AllocationKind`	`win:UInt32`	`0x0` - 小型对象分配(小型对象堆中的分配)。 `0x1` - 大型对象分配(大型对象堆中的分配)。
`AllocationAmount64`	`win:UInt64`	分配大小(以字节为单位)。对于非常大的分配，此值为精确值。
`TypeId`	`win:Pointer`	MethodTable 的地址。如果在此事件期间分配了几种类型的对象，则此地址为对应于分配的最后一个对象(导致超过 100 KB 阈值的对象)的 MethodTable 地址。
`TypeName`	`win:UnicodeString`	已分配的类型的名称。如果在此事件期间分配了几种类型的对象，则此地址为对应于分配的最后一个类型(导致超过 100 KB 阈值的对象)。
`HeapIndex`	`win:UInt32`	此对象所分配到的堆。与工作站垃圾回收一起运行时，此值为 0(零)。
`Address`	`win:Pointer`	上次分配的对象的地址。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

GCCreateConcurrentThread_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCCreateConcurrentThread_V1`	11	已创建并发垃圾回收线程。

此事件没有任何事件数据。

GCTerminateConcurrentThread_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCTerminateConcurrentThread_V1`	12	已终止并发垃圾回收线程。

此事件没有任何事件数据。

GCFinalizersBegin_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCFinalizersBegin_V1`	14	开始运行终结器。

此事件没有任何事件数据。

GCFinalizersEnd_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCFinalizersEnd_V1`	13	结束运行终结器。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt32`	运行的终结器数。
`ClrInstanceID`	win:UInt16	CLR 或 CoreCLR 的实例的唯一 ID。

SetGCHandle 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCHandleKeyword` (0x2)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`SetGCHandle`	30	已设置 GC 句柄。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`HandleID`	`win:Pointer`	已分配的句柄的地址。
`ObjectID`	`win:Pointer`	创建了句柄的对象的地址。
`Kind`	`win:UInt32`	已设置的 GC 句柄的类型。 `0x0`：WeakShort `0x1`：WeakLong `0x2`：Strong `0x3`：Pinned `0x4`：Variable `0x5`：RefCounted `0x6`：Dependent `0x7`：AsyncPinned `0x8`：SizedRef
`Generation`	`win:UInt32`	创建了其句柄的对象的生成。
`AppDomainID`	`win:UInt64`	AppDomain ID。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

DestroyGCHandle 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCHandleKeyword` (0x2)	信息性 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`DestroyGCHandle`	31	已销毁 GC 句柄。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`HandleID`	`win:Pointer`	已销毁的句柄的地址。
`ClrInstanceID`	win:UInt16	CoreCLR 实例的唯一 ID。

PinObjectAtGCTime 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	详细级别 (5)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`PinObjectAtGCTime`	33	对象在 GC 期间已固定。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`HandleID`	`win:Pointer`	句柄的地址。
`ObjectID`	`win:Pointer`	已固定对象的地址。
`ObjectSize`	`win:UInt64`	已固定对象的大小。
`TypeName`	`win:UnicodeString`	已固定对象的类型的名称。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

GCTriggered 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	详细级别 (5)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCTriggered`	33	已触发 GC。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Reason`	`win:UInt32`	触发 GC 的原因。 `0x0`：AllocSmall `0x1`：Induced `0x2`：LowMemory `0x3`：Empty `0x4`：AllocLarge `0x5`：OutOfSpaceSmallObjectHeap `0x6`：OutOfSpaceLargeObjectHeap `0x7`：InducedNoForce `0x8`：Stress `0x9`：InducedLowMemory
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

IncreaseMemoryPressure 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`IncreaseMemoryPressure`	200	内存压力增加。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`ClrInstanceID`	win:UInt16	CoreCLR 实例的唯一 ID。

DecreaseMemoryPressure 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`DecreaseMemoryPressure`	201	内存压力降低。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`BytesFreed`	`win:UInt32`	已释放字节。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

GCMarkWithType 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	信息 (4)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCMarkWithType`	202	GC 标记阶段期间已标记了一个 GC 根。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`HeapNum`	`win:UInt32`	堆号。
`ClrInstanceID`	win:UInt16	CoreCLR 实例的唯一 ID。

[illegible]	[illegible]	[illegible]
`Type`	`win:UInt32`	GC 根类型。 `0x0` : Stack `0x1` : Finalizer `0x2` : Handle `0x3` : Older `0x4` : SizedRef `0x5` : Overflow
`Bytes`	`win:UInt64`	已标记的字节数。

GCJoin_V2 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`GCKeyword` (0x1)	详细级别 (5)

下表显示了事件信息：

[illegible]	[illegible] ID	[illegible]
`GCJoin_V2`	203	已联接的 GC 线程。

下表显示了事件数据：

[illegible]	[illegible]	[illegible]
`Heap`	`win:UInt32`	堆号
`JoinTime`	`win:UInt32`	指示在联接开始或结束时是否触发此事件（`0x0` 为联接开始，`0x1` 为联接结束）
`JoinType`	`win:UInt32`	联接类型。 `0x0` : Last Join `0x1` : Join `0x2` : Restart `0x3` : First Reverse Join `0x4` : Reverse Join

[illegible]	[illegible]	[illegible]
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

.NET 运行时互操作事件

2021/11/16 •

这些运行时事件捕获有关公共中间语言 (CIL) 存根生成的信息。有关如何将这些事件用于诊断的详细信息，请参阅 .NET 应用程序日志记录和跟踪

ILStubGenerated 事件

[illegible]	LEVEL
`InteropKeyword` (0x2000)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ILStubGenerated`	88	生成 IL 存根。

[illegible]	[illegible]	[illegible]
`ModuleID`	`win:UInt16`	模块标识符。
`StubMethodID`	`win:UInt64`	存根方法标识符。
`StubFlags`	`win:UInt32`	存根标志： `0x1` - 反向互操作。 `0x2` - COM 互操作。 `0x4` - 由 NGen.exe 生成的存根。 `0x8` - 委托。 `0x10` - 变量参数。 `0x20` - 非托管被调用方。 `0x40` - 结构封送
`ManagedInteropMethodToken`	`win:UInt32`	托管互操作方法的标记。
`ManagedInteropMethodNameSpace`	`win:UnicodeString`	托管互操作方法的命名空间和封闭类型。
`ManagedInteropMethodName`	`win:UnicodeString`	托管互操作方法的名称。
`ManagedInteropMethodSignature`	`win:UnicodeString`	托管互操作方法的签名。
`NativeMethodSignature`	`win:UnicodeString`	本机方法签名。

[illegible]	[illegible]	[illegible]
`StubMethodSignature`	`win:UnicodeString`	存根方法签名。
`StubMethodILCode`	`win:UnicodeString`	存根方法的公共中间语言 (CIL) 代码。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

.NET 运行时加载器和绑定器事件

2021/11/16 •

这些事件收集有关加载和卸载程序集和模块的信息。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`DomainModuleLoad_V1`	151	在为应用程序域加载模块时引发。

ModuleLoad_V2 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ModuleLoad_V2`	152	在进程的生存期内加载模块时引发。

[illegible]	[illegible]	[illegible]
`ModuleID`	`win:UInt64`	模块的唯一 ID。
`AssemblyID`	`win:UInt64`	此模块所驻留的程序集的 ID。
`ModuleFlags`	`win:UInt32`	0x1：非特定于域的模块。 0x2：模块具有本机映像。 0x4：动态模块。 0x8：清单模块。
`Reserved1`	`win:UInt32`	保留的字段。
`ModuleILPath`	`win:UnicodeString`	模块的公共中间语言 (CIL) 映像的路径；如果是动态程序集（以 null 结尾），则为动态模块名。
`ModuleNativePath`	`win:UnicodeString`	如果存在（以 null 结尾），则为模块本机映像的路径。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

[illegible]	[illegible]	[illegible]
`ManagedPdbSignature`	`win:GUID`	匹配此模块的托管程序数据库 (PDB) 的 GUID 签名。
`ManagedPdbAge`	`win:UInt32`	写入匹配此模块的托管 PDB 的年限数。
`ManagedPdbBuildPath`	`win:UnicodeString`	生成匹配此模块的托管 PDB 的位置的路径。在某些情况下，这可能只是一个文件名。
`NativePdbSignature`	`win:GUID`	匹配此模块的本机映像生成器 (NGen) PDB 的 GUID 签名（如果适用）。
`NativePdbAge`	`win:UInt32`	写入匹配此模块的 NGen PDB 的年限数（如果适用）。
`NativePdbBuildPath`	`win:UnicodeString`	生成匹配此模块的 NGen PDB 的位置的路径（如果适用）。在某些情况下，这可能只是一个文件名。

ModuleUnload_V2 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ModuleUnload_V2`	153	在进程的生存期内卸载模块时引发。

[illegible]	[illegible]	[illegible]
`ModuleID`	`win:UInt64`	模块的唯一 ID。
`AssemblyID`	`win:UInt64`	此模块所驻留的程序集的 ID。
`ModuleFlags`	`win:UInt32`	0x1：非特定于域的模块。 0x2：模块具有本机映像。 0x4：动态模块。 0x8：清单模块。
`Reserved1`	`win:UInt32`	保留的字段。
`ModuleILPath`	`win:UnicodeString`	模块的公共中间语言 (CIL) 映像的路径；如果是动态程序集（以 null 结尾），则为动态模块名。
`ModuleNativePath`	`win:UnicodeString`	如果存在（以 null 结尾），则为模块本机映像的路径。

[illegible]	[illegible]	[illegible]
ClrInstanceID	win:UInt16	CLR 或 CoreCLR 的实例的唯一 ID。
ManagedPdbSignature	win:GUID	匹配此模块的托管程序数据库 (PDB) 的 GUID 签名。
ManagedPdbAge	win:UInt32	写入匹配此模块的托管 PDB 的年限数。
ManagedPdbBuildPath	win:UnicodeString	生成匹配此模块的托管 PDB 的位置的路径。在某些情况下，这可能只是一个文件名。
NativePdbSignature	win:GUID	匹配此模块的本机映像生成器 (NGen) PDB 的 GUID 签名（如果适用）。
NativePdbAge	win:UInt32	写入匹配此模块的 NGen PDB 的年限数（如果适用）。
NativePdbBuildPath	win:UnicodeString	生成匹配此模块的 NGen PDB 的位置的路径（如果适用）。在某些情况下，这可能只是一个文件名。

ModuleDCStart_V2 事件

[illegible]	[illegible]	LEVEL
LoaderKeyword (0x8)	DomainModuleLoad_V1	信息性 (4)

[illegible]	[illegible] ID	[illegible]
ModuleDCStart_V2	153	在启动断开期间枚举模块。

[illegible]	[illegible]	[illegible]
ModuleID	win:UInt64	模块的唯一 ID。
AssemblyID	win:UInt64	此模块所驻留的程序集的 ID。
ModuleFlags	win:UInt32	0x1：非特定于域的模块。 0x2：模块具有本机映像。 0x4：动态模块。 0x8：清单模块。
Reserved1	win:UInt32	保留的字段。
ModuleILPath	win:UnicodeString	模块的公共中间语言 (CIL) 映像的路径；如果是动态程序集（以 null 结尾），则为动态模块名。

[illegible]	[illegible]	[illegible]
`ModuleNativePath`	`win:UnicodeString`	如果存在(以 null 结尾)，则为模块本机映像的路径。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。
`ManagedPdbSignature`	`win:GUID`	匹配此模块的托管程序数据库 (PDB) 的 GUID 签名。
`ManagedPdbAge`	`win:UInt32`	写入匹配此模块的托管 PDB 的年限数。
`ManagedPdbBuildPath`	`win:UnicodeString`	生成匹配此模块的托管 PDB 的位置的路径。在某些情况下，这可能只是一个文件名。
`NativePdbSignature`	`win:GUID`	匹配此模块的本机映像生成器 (NGen) PDB 的 GUID 签名(如果适用)。
`NativePdbAge`	`win:UInt32`	写入匹配此模块的 NGen PDB 的年限数(如果适用)。
`NativePdbBuildPath`	`win:UnicodeString`	生成匹配此模块的 NGen PDB 的位置的路径(如果适用)。在某些情况下，这可能只是一个文件名。

ModuleDCEnd_V2 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ModuleDCEnd_V2`	154	在结束断开期间枚举模块。

[illegible]	[illegible]	[illegible]
`ModuleID`	`win:UInt64`	模块的唯一 ID。
`AssemblyID`	`win:UInt64`	此模块所驻留的程序集的 ID。
`ModuleFlags`	`win:UInt32`	0x1：非特定于域的模块。 0x2：模块具有本机映像。 0x4：动态模块。 0x8：清单模块。
`Reserved1`	`win:UInt32`	保留的字段。

[illegible]	[illegible]	[illegible]
`ModuleILPath`	`win:UnicodeString`	模块的公共中间语言 (CIL) 映像的路径；如果是动态程序集（以 null 结尾），则为动态模块名。
`ModuleNativePath`	`win:UnicodeString`	如果存在（以 null 结尾），则为模块本机映像的路径。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。
`ManagedPdbSignature`	`win:GUID`	匹配此模块的托管程序数据库 (PDB) 的 GUID 签名。
`ManagedPdbAge`	`win:UInt32`	写入匹配此模块的托管 PDB 的年限数。
`ManagedPdbBuildPath`	`win:UnicodeString`	生成匹配此模块的托管 PDB 的位置的路径。在某些情况下，这可能只是一个文件名。
`NativePdbSignature`	`win:GUID`	匹配此模块的本机映像生成器 (NGen) PDB 的 GUID 签名（如果适用）。
`NativePdbAge`	`win:UInt32`	写入匹配此模块的 NGen PDB 的年限数（如果适用）。
`NativePdbBuildPath`	`win:UnicodeString`	生成匹配此模块的 NGen PDB 的位置的路径（如果适用）。在某些情况下，这可能只是一个文件名。

AssemblyLoad_V1 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`AssemblyLoad_V1`	154	在加载程序集时引发。

[illegible]	[illegible]	[illegible]
`AssemblyID`	`win:UInt64`	程序集的唯一 ID。
`AppDomainID`	`win:UInt64`	此程序集的域的 ID。
`BindingID`	`win:UInt64`	唯一地标识程序集绑定的 ID。

[illegible]	[illegible]	[illegible]
`AssemblyFlags`	`win:UInt32`	0x1:非特定于域的程序集。 0x2:动态程序集。 0x4:程序集具有本机映像。 0x8:可回收程序集。
`AssemblyName`	`win:UnicodeString`	完全限定程序集名称。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyUnload_V1 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`FireAssemblyUnload_V1`	155	在加载程序集时引发。

[illegible]	[illegible]	[illegible]
`AssemblyID`	`win:UInt64`	程序集的唯一 ID。
`AppDomainID`	`win:UInt64`	此程序集的域的 ID。
`BindingID`	`win:UInt64`	唯一地标识程序集绑定的 ID。
`AssemblyFlags`	`win:UInt32`	0x1:非特定于域的程序集。 0x2:动态程序集。 0x4:程序集具有本机映像。 0x8:可回收程序集。
`AssemblyName`	`win:UnicodeString`	完全限定程序集名称。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyDCStart_V1 事件

[illegible]	[illegible]	LEVEL
`LoaderKeyword` (0x8)	`DomainModuleLoad_V1`	信息性 (4)

□□	□□ ID	□□
`AssemblyDCStart_V1`	155	在启动断开期间枚举程序集。

□□□	□□□□	□□
`AssemblyID`	`win:UInt64`	程序集的唯一 ID。
`AppDomainID`	`win:UInt64`	此程序集的域的 ID。
`BindingID`	`win:UInt64`	唯一地标识程序集绑定的 ID。
`AssemblyFlags`	`win:UInt32`	0x1：非特定于域的程序集。 0x2：动态程序集。 0x4：程序集具有本机映像。 0x8：可回收程序集。
`AssemblyName`	`win:UnicodeString`	完全限定程序集名称。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyLoadStart 事件

□□□□□□□	□□	LEVEL
`Binder` (0x4)	`AssemblyLoadStart`	信息性 (4)

□□	□□ ID	□□
`AssemblyLoadStart`	290	已请求程序集加载。

□□□	□□□□	□□
`AssemblyName`	`win:UnicodeString`	程序集的名称。
`AssemblyPath`	`win:UnicodeString`	程序集名称的路径。
`RequestingAssembly`	`win:UnicodeString`	正在请求的(“父”)程序集的名称。
`AssemblyLoadContext`	`win:UnicodeString`	程序集的加载上下文。
`RequestingAssemblyLoadContext`	`win:UnicodeString`	正在请求的(“父”)程序集的加载上下文。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyLoadStop 事件

[illegible]	[illegible]	LEVEL
`Binder` (0x4)	`AssemblyLoadStart`	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`AssemblyLoadStart`	291	已请求程序集加载。

[illegible]	[illegible]	[illegible]
`AssemblyName`	`win:UnicodeString`	程序集的名称。
`AssemblyPath`	`win:UnicodeString`	程序集名称的路径。
`RequestingAssembly`	`win:UnicodeString`	正在请求的(“父”)程序集的名称。
`AssemblyLoadContext`	`win:UnicodeString`	程序集的加载上下文。
`RequestingAssemblyLoadContext`	`win:UnicodeString`	正在请求的(“父”)程序集的加载上下文。
`Success`	`win:Boolean`	程序集加载是否成功。
`ResultAssemblyName`	`win:UnicodeString`	已加载的程序集的名称。
`ResultAssemblyPath`	`win:UnicodeString`	从中加载的程序集的路径。
`Cached`	`win:UnicodeString`	是否缓存了加载。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ResolutionAttempted 事件

[illegible]	LEVEL
`Binder` (0x4)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ResolutionAttempted`	292	已请求程序集加载。

[illegible]	[illegible]	[illegible]
`AssemblyName`	`win:UnicodeString`	程序集的名称。

[illegible]	[illegible]	[illegible]
`Stage`	`win:UInt16`	解析阶段。 0:在加载中查找。 1:程序集加载上下文 2:部署应用程序集。 3:默认程序集加载上下文回退。 4:解析附属程序集。 5:正在解析程序集加载上下文。 6:正在解析 AppDomain 程序集。
`AssemblyLoadContext`	`win:UnicodeString`	程序集的加载上下文。
`Result`	`win:UInt16`	解析尝试的结果。 0:成功 1:未找到程序集 2:版本不兼容 3:程序集名称不匹配 4:失败 5:异常
`ResultAssemblyName`	`win:UnicodeString`	已解析的程序集的名称。
`ResultAssemblyPath`	`win:UnicodeString`	从中解析的程序集的路径。
`ErrorMessage`	`win:UnicodeString`	发生异常时的错误消息。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyLoadContextResolvingHandlerInvoked 事件

[illegible]	LEVEL
`Binder` (0x4)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`AssemblyLoadContextResolvingHandlerInvoked`	293	已调用 AssemblyLoadContext.Resolving 处理程序。

[illegible]	[illegible]	[illegible]
`AssemblyName`	`win:UnicodeString`	程序集的名称。
`HandlerName`	`win:UnicodeString`	已调用的处理程序的名称。
`AssemblyLoadContext`	`win:UnicodeString`	程序集的加载上下文。
`ResultAssemblyName`	`win:UnicodeString`	已解析的程序集的名称。
`ResultAssemblyPath`	`win:UnicodeString`	从中解析的程序集的路径。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AppDomainAssemblyResolveHandlerInvoked 事件

[illegible]	LEVEL
`Binder` (0x4)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`AppDomainAssemblyResolveHandlerInvoked`	294	已调用 AppDomain.AssemblyResolve 处理程序。

[illegible]	[illegible]	[illegible]
`AssemblyName`	`win:UnicodeString`	程序集的名称。
`HandlerName`	`win:UnicodeString`	已调用的处理程序的名称。
`ResultAssemblyName`	`win:UnicodeString`	已解析的程序集的名称。
`ResultAssemblyPath`	`win:UnicodeString`	从中解析的程序集的路径。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

AssemblyLoadFromResolveHandlerInvoked 事件

[illegible]	LEVEL
`Binder` (0x4)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`AssemblyLoadFromResolveHandlerInvoked`	295	已调用 Assembly.LoadFrom 处理程序。

[illegible]	[illegible]	[illegible]
`AssemblyName`	`win:UnicodeString`	程序集的名称。
`IsTrackedLoad`	`win:Boolean`	是否跟踪程序集加载。
`RequestingAssemblyPath`	`win:UnicodeString`	正在请求的程序集的路径。
`ComputedRequestedAssemblyPath`	`win:UnicodeString`	已请求的程序集的路径。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

KnownPathProbed 事件

[illegible]	LEVEL
`Binder` (0x4)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`KnownPathProbed`	296	探测到了程序集的一个已知路径。

[illegible]	[illegible]	[illegible]
`FilePath`	`win:UnicodeString`	已探测的路径。
`Source`	`win:UInt16`	已探测的路径的源。 0x0：应用程序集。 0x1：应用本机映像路径。 0x2：应用路径。 0x3：平台资源根。 0x4：附属子目录。
`Result`	`win:UInt32`	探测的 HRESULT。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

.NET 运行时方法事件

2021/11/16 •

这些事件收集特定于方法的信息。符号解析需要这些事件的负载。此外，这些事件还提供了一些有用的信息，例如已加载和已卸载的方法。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

所有方法事件都具有“信息性 (4)”级别。所有方法的详细事件都具有“详细级别 (5)”级别。

所有方法事件都是由运行时提供程序下的 `JITKeyword` (0x10) 关键字或 `NGenKeyword` (0x20) 关键字引发的，或是由断开提供程序下的 `JitRundownKeyword` (0x10) 或 `NGENRundownKeyword` (0x20) 引发的。

这些事件的 V2 版本包括 ReJITID（V1 版本不包含）。

MethodLoad_V1 事件

下表显示了事件信息：

`MethodLoad_V1`	141	在实时加载（JIT 加载）方法或者加载 NGEN 映像时引发。动态和泛型方法不使用此版本进行方法加载。JIT 帮助器从不使用此版本。

`JITKeyword` (0x10) 运行时提供程序	信息性 (4)
`NGenKeyword` (0x20) 运行时提供程序	信息性 (4)

`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	方法的起始地址。
`MethodSize`	`win:UInt32`	方法的大小。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。

[illegible]	[illegible]	[illegible]
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的代码方法（否则为 NGEN 本机映像代码）。 0x8：帮助器方法。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodLoad_V2 事件

[illegible]	[illegible] ID	[illegible]
`MethodLoad_V2`	141	在实时加载（JIT 加载）方法或者加载 NGEN 映像时引发。动态和泛型方法不使用此版本进行方法加载。JIT 帮助器从不使用此版本。

[illegible]	LEVEL
`JITKeyword` (0x10) 运行时提供程序	信息性 (4)
`NGenKeyword` (0x20) 运行时提供程序	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	方法的起始地址。
`MethodSize`	`win:UInt32`	方法的大小。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的代码方法（否则为 NGEN 本机映像代码）。 0x8：帮助器方法。
`ReJITID`	`win:UInt64`	方法的 ReJIT ID。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodUnLoad_V1 事件

名称	事件 ID	说明
`MethodUnLoad_V1`	142	在卸载模块或销毁应用程序域时引发。动态方法从不使用此版本进行方法卸载。

关键字	级别
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

项	类型	说明
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	方法的起始地址。
`MethodSize`	`win:UInt32`	方法的大小。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的代码方法（否则为 NGEN 本机映像代码）。 0x8：帮助器方法。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodUnLoad_V2 事件

名称	事件 ID	说明
`MethodUnLoad_V2`	142	在卸载模块或销毁应用程序域时引发。动态方法从不使用此版本进行方法卸载。

关键字	级别
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符(0 表示 JIT 帮助器)。
`MethodStartAddress`	`win:UInt64`	方法的起始地址。
`MethodSize`	`win:UInt32`	方法的大小。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的代码方法(否则为 NGEN 本机映像代码)。 0x8：帮助器方法。
`ReJITID`	`win:UInt64`	方法的 ReJIT ID。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

R2RGetEntryPoint 事件

[illegible]	[illegible] ID	[illegible]
`R2RGetEntryPoint`	159	R2R 入口点查找结束时引发。

[illegible]	LEVEL
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	R2R 方法的唯一标识符。
`MethodNamespace`	`win:UnicodeString`	要查找的方法的命名空间。
`MethodName`	`win:UnicodeString`	要查找的方法的名称。
`MethodSignature`	`win:UnicodeString`	方法的签名(以逗号分隔的类型名称列表)。
`EntryPoint`	`win:UInt64`	指向 R2R 方法入口点的指针

字段	类型	说明
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

R2RGetEntryPointStart 事件

名称	事件 ID	说明
`R2RGetEntryPointStart`	160	R2R 入口点查找开始时引发。

关键字	级别
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

字段	类型	说明
`MethodID`	`win:UInt64`	R2R 方法的唯一标识符。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodLoadVerbose_V1 事件

名称	事件 ID	说明
`MethodLoadVerbose_V1`	143	当方法为 JIT 加载的或加载 NGEN 映像时引发。动态和泛型方法始终使用此版本进行方法加载。JIT 帮助器始终使用此版本。

关键字	级别
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

字段	类型	说明
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符(0 表示 JIT 帮助器)。
`MethodStartAddress`	`win:UInt64`	起始地址。
`MethodSize`	`win:UInt32`	方法长度。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。

[illegible]	[illegible]	[illegible]
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的方法（否则由 NGen.exe 生成） 0x8：帮助器方法。
`MethodNameSpace`	`win:UnicodeString`	与该方法关联的完整命名空间名称。
`MethodName`	`win:UnicodeString`	与该方法关联的完整类名称。
`MethodSignature`	`win:UnicodeString`	方法的签名（以逗号分隔的类型名称列表）。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodLoadVerbose_V2 事件

[illegible]	[illegible] ID	[illegible]
`MethodLoadVerbose_V1`	143	当方法为 JIT 加载的或加载 NGEN 映像时引发。动态和泛型方法始终使用此版本进行方法加载。JIT 帮助器始终使用此版本。

[illegible]	LEVEL
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	起始地址。
`MethodSize`	`win:UInt32`	方法长度。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。

[illegible]	[illegible]	[illegible]
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的方法（否则由 NGen.exe 生成） 0x8：帮助器方法。
`MethodNameSpace`	`win:UnicodeString`	与该方法关联的完整命名空间名称。
`MethodName`	`win:UnicodeString`	与该方法关联的完整类名称。
`MethodSignature`	`win:UnicodeString`	方法的签名（以逗号分隔的类型名称列表）。
`ReJITID`	`win:UInt64`	方法的 ReJIT ID。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodUnLoadVerbose_V1 事件

[illegible]	[illegible] ID	[illegible]
`MethodUnLoadVerbose_V1`	144	在销毁动态方法、卸载模块或销毁应用程序域时引发。动态方法始终使用此版本进行方法卸载。

[illegible]	LEVEL
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	起始地址。
`MethodSize`	`win:UInt32`	方法长度。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。

[illegible]	[illegible]	[illegible]
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的方法（否则由 NGen.exe 生成） 0x8：帮助器方法。
`MethodNameSpace`	`win:UnicodeString`	与该方法关联的完整命名空间名称。
`MethodName`	`win:UnicodeString`	与该方法关联的完整类名称。
`MethodSignature`	`win:UnicodeString`	方法的签名（以逗号分隔的类型名称列表）。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodUnLoadVerbose_V2 事件

[illegible]	[illegible] ID	[illegible]
`MethodUnLoadVerbose_V2`	144	在销毁动态方法、卸载模块或销毁应用程序域时引发。动态方法始终使用此版本进行方法卸载。

[illegible]	LEVEL
`JITKeyword` (0x10)	信息性 (4)
`NGenKeyword` (0x20)	信息性 (4)

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。对于 JIT 帮助器方法，将设置为该方法的起始地址。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符（0 表示 JIT 帮助器）。
`MethodStartAddress`	`win:UInt64`	起始地址。
`MethodSize`	`win:UInt32`	方法长度。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。

[illegible]	[illegible]	[illegible]
`MethodFlags`	`win:UInt32`	0x1：动态方法。 0x2：泛型方法。 0x4：JIT 编译的方法（否则由 NGen.exe 生成） 0x8：帮助器方法。
`MethodNameSpace`	`win:UnicodeString`	与该方法关联的完整命名空间名称。
`MethodName`	`win:UnicodeString`	与该方法关联的完整类名称。
`MethodSignature`	`win:UnicodeString`	方法的签名（以逗号分隔的类型名称列表）。
`ReJITID`	`win:UInt64`	方法的 ReJIT ID。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodJittingStarted_V1 事件

下表显示了关键字和级别：

[illegible]	LEVEL
`JITKeyword` (0x10)	详细级别 (5)
`NGenKeyword` (0x20)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodJittingStarted_V1`	145	在方法由 JIT 编译时引发。

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。
`ModuleID`	`win:UInt64`	此方法所属的模块标识符。
`MethodToken`	`win:UInt32`	0 代表动态方法和 JIT 帮助器。
`MethodILSize`	`win:UInt32`	要进行 JIT 编译的方法的公共中间语言 (CIL) 的大小。
`MethodNameSpace`	`win:UnicodeString`	与该方法关联的完整类名称。
`MethodName`	`win:UnicodeString`	方法的名称。

[illegible]	[illegible]	[illegible]
`MethodSignature`	`win:UnicodeString`	方法的签名(以逗号分隔的类型名称列表)。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodJitInliningSucceeded 事件

[illegible]	LEVEL
`JITTracingKeyword` (0x1000)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodJitInliningSucceeded`	185	当方法由 JIT 编译器成功内联时引发。

[illegible]	[illegible]	[illegible]
`MethodBeingCompiledNamespace`	`win:UnicodeString`	要编译的方法的命名空间。
`MethodBeingCompiledName`	`win:UnicodeString`	要编译的方法的名称。
`MethodBeingCompiledNameSignature`	`win:UnicodeString`	要编译的方法的签名(以逗号分隔的类型名称列表)。
`InlinerNamespace`	`win:UnicodeString`	内联方("父级")方法的命名空间。
`InlinerName`	`win:UnicodeString`	内联方("父级")方法的名称。
`InlinerNameSignature`	`win:UnicodeString`	内联方("父级")方法的签名(以逗号分隔的类型名称列表)。
`InlineeNamespace`	`win:UnicodeString`	被内联方("子级")方法的命名空间。
`InlineeName`	`win:UnicodeString`	被内联方("子级")方法的名称。
`InlineeNameSignature`	`win:UnicodeString`	被内联方("子级")方法的签名(以逗号分隔的类型名称列表)。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodJitInliningFailed 事件

[illegible]	LEVEL
`JITTracingKeyword` (0x1000)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodJitInliningFailed`	192	当方法无法由 JIT 编译器内联时引发。

[illegible]	[illegible]	[illegible]
`MethodBeingCompiledNamespace`	`win:UnicodeString`	要编译的方法的命名空间。
`MethodBeingCompiledName`	`win:UnicodeString`	要编译的方法的名称。
`MethodBeingCompiledNameSignature`	`win:UnicodeString`	要编译的方法的签名(以逗号分隔的类型名称列表)。
`InlinerNamespace`	`win:UnicodeString`	内联方("父级")方法的命名空间。
`InlinerName`	`win:UnicodeString`	内联方("父级")方法的名称。
`InlinerNameSignature`	`win:UnicodeString`	内联方("父级")方法的签名(以逗号分隔的类型名称列表)。
`InlineeNamespace`	`win:UnicodeString`	被内联方("子级")方法的命名空间。
`InlineeName`	`win:UnicodeString`	被内联方("子级")方法的名称。
`InlineeNameSignature`	`win:UnicodeString`	被内联方("子级")方法的签名(以逗号分隔的类型名称列表)。
`FailAlways`	`win:Boolean`	方法是否被标记为不可内联。
`FailReason`	`win:UnicodeString`	内联失败的原因。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodJITTailCallSucceeded 事件

[illegible]	LEVEL
`JITTracingKeyword` (0x1000)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodJitTailCallSucceeded`	192	当方法可成功被尾调用时由 JIT 编译器引发。

[illegible]	[illegible]	[illegible]
`MethodBeingCompiledNamespace`	`win:UnicodeString`	要编译的方法的命名空间。
`MethodBeingCompiledName`	`win:UnicodeString`	要编译的方法的名称。

[illegible]	[illegible]	[illegible]
`MethodBeingCompiledNameSignature`	`win:UnicodeString`	要编译的方法的签名(以逗号分隔的类型名称列表)。
`CallerNamespace`	`win:UnicodeString`	调用方方法的命名空间。
`CallerName`	`win:UnicodeString`	调用方方法的名称。
`CallerNameSignature`	`win:UnicodeString`	调用方方法的签名(以逗号分隔的类型名称列表)。
`CalleeNamespace`	`win:UnicodeString`	被调用方方法的命名空间。
`CalleeName`	`win:UnicodeString`	被调用方方法的名称。
`CalleeNameSignature`	`win:UnicodeString`	被调用方方法的签名(以逗号分隔的类型名称列表)。
`TailPrefix`	`win:Boolean`	它是否为尾前缀指令。
`TailCallType`	`win:UInt32`	尾调用的类型。 0:优化的尾调用 (epilog + jmp) 1:递归的尾调用(方法尾调用自身) 2:帮助程序辅助的尾调用
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodJITTailCallFailed 事件

[illegible]	LEVEL
`JITTracingKeyword` (0x1000)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodJitTailCallFailed`	191	当方法无法被尾调用时由 JIT 编译器引发。

[illegible]	[illegible]	[illegible]
`MethodBeingCompiledNamespace`	`win:UnicodeString`	要编译的方法的命名空间。
`MethodBeingCompiledName`	`win:UnicodeString`	要编译的方法的名称。
`MethodBeingCompiledNameSignature`	`win:UnicodeString`	要编译的方法的签名(以逗号分隔的类型名称列表)。
`CallerNamespace`	`win:UnicodeString`	调用方方法的命名空间。

[illegible]	[illegible]	[illegible]
`CallerNam` e	`win:UnicodeString`	调用方方法的名称。
`CallerNameSignature`	`win:UnicodeString`	调用方方法的签名(以逗号分隔的类型名称列表)。
`CalleeNamespace`	`win:UnicodeString`	被调用方方法的命名空间。
`CalleeName`	`win:UnicodeString`	被调用方方法的名称。
`CalleeNameSignature`	`win:UnicodeString`	被调用方方法的签名(以逗号分隔的类型名称列表)。
`TailPrefix`	`win:Boolean`	它是否为尾前缀指令。
`FailReason`	`win:UnicodeString`	尾调用失败的原因。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

MethodILToNativeMap 事件

[illegible]	LEVEL
`JittedMethodILToNativeMapKeyword` (0x20000)	详细级别 (5)

[illegible]	[illegible] ID	[illegible]
`MethodILToNativeMap`	190	为 JIT 编译的方法映射 IL 到本机映射事件。

[illegible]	[illegible]	[illegible]
`MethodID`	`win:UInt64`	方法的唯一标识符。
`ReJITID`	`win:UInt64`	方法的 ReJIT ID。
`MethodExtent`	`win:UInt8`	已执行 JIT 的方法的范围。
`CountOfMapEntries`	`win:UInt8`	映射条目数
`ILOffsets`	`win:UInt32`	IL 偏移量。
`NativeOffsets`	`win:UInt32`	本机代码偏移量。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

.NET 运行时线程池事件

2021/11/16 •

这些事件收集线程池中有关工作线程和 I/O 线程的信息。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

IOThreadCreate_V1 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`IOThreadCreate_V1`	44	在线程池中创建 I/O 线程。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt64`	I/O 线程数，包括新创建的线程。
`NumRetired`	`win:UInt64`	已停用的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

IOThreadTerminate_V1 事件

下表显示了关键字和级别

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`IOThreadTerminate`	45	在线程池中终止 I/O 线程。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt64`	线程池中剩余的 I/O 线程数。

[illegible]	[illegible]	[illegible]
`NumRetired`	`win:UInt64`	已停用的 I/O 线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

IOThreadRetire_V1 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`IOThreadRetire_V1`	46	I/O 线程变为停用候选项。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt64`	线程池中剩余的 I/O 线程数。
`NumRetired`	`win:UInt64`	已停用的 I/O 线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

IOThreadUnretire_V1 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`IOThreadUnretire_V1`	47	I/O 线程因在该线程变为停用候选项后的等待期间内达到 I/O 而恢复使用。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`Count`	`win:UInt64`	线程池中的 I/O 线程数，包括这一个。

[illegible]	[illegible]	[illegible]
`NumRetired`	`win:UInt64`	已停用的 I/O 线程数。
`ClrInstanceID`	`Win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadStart 事件

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ThreadPoolWorkerThreadStart`	50	创建工作线程。

[illegible]	[illegible]	[illegible]
`ActiveWorkerThreadCount`	`win:UInt32`	可用于处理工作的工作线程数，包括已在处理工作的工作线程。
`RetiredWorkerThreadCount`	`win:UInt32`	不能用于处理工作但被保留以防之后需要更多线程的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadStop 事件

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`ThreadPoolWorkerThreadStop`	51	停止工作线程。

[illegible]	[illegible]	[illegible]
`ActiveWorkerThreadCount`	`win:UInt32`	可用于处理工作的工作线程数，包括已在处理工作的工作线程。
`RetiredWorkerThreadCount`	`win:UInt32`	不能用于处理工作但被保留以防之后需要更多线程的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadWait 事件

□□□□□□□	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

□□	□□ ID	□□
`ThreadPoolWorkerThreadWait`	57	工作线程开始等待运行。

□□□	□□□□	□□
`ActiveWorkerThreadCount`	`win:UInt32`	可用于处理工作的工作线程数，包括已在处理工作的工作线程。
`RetiredWorkerThreadCount`	`win:UInt32`	不能用于处理工作但被保留以防之后需要更多线程的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadRetirementStart 事件

□□□□□□□	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

□□	□□ ID	□□
`ThreadPoolWorkerThreadRetirementStart`	52	停用工作线程。

□□□	□□□□	□□
`ActiveWorkerThreadCount`	`win:UInt32`	可用于处理工作的工作线程数，包括已在处理工作的工作线程。
`RetiredWorkerThreadCount`	`win:UInt32`	不能用于处理工作但被保留以防之后需要更多线程的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadRetirementStop 事件

□□□□□□□	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

□□	□□ ID	□□
`ThreadPoolWorkerThreadRetirementStop`	53	停用的工作线程再次变为活动状态。

[illegible]	[illegible]	[illegible]
`ActiveWorkerThreadCount`	`win:UInt32`	可用于处理工作的工作线程数，包括已在处理工作的工作线程。
`RetiredWorkerThreadCount`	`win:UInt32`	不能用于处理工作但被保留以防之后需要更多线程的工作线程数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadAdjustmentSample 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolWorkerThreadAdjustmentSample`	54	指一个示例的信息收集，即具有一定并发级别的即时吞吐量测量。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`Throughput`	`win:Double`	每个时间单位的完成数。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadAdjustmentAdjustment 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolWorkerThreadAdjustmentAdjustment`	55	当线程注入（爬山）算法确定并发级别发生更改时，在控件中记录更改。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`AverageThroughput`	`win:Double`	测量示例的平均吞吐量。
`NewWorkerThreadCount`	`win:UInt32`	新的活动工作线程数。
`Reason`	`win:UInt32`	调整的原因。 `0x0` - 预热。 `0x1` - 正在初始化。 `0x2` - 随机移动。 `0x3` - 攀移。 `0x4` - 更改点。 `0x5` - 正在稳定。 `0x6` - 匮乏。 `0x7` - 线程已超时。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolWorkerThreadAdjustmentStats 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolWorkerThreadAdjustmentStats`	56	在线程池上收集数据。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`Duration`	`win:Double`	收集这些统计信息的时间量(以秒为单位)。
`Throughput`	`win:Double`	在此间隔期间每秒完成的平均数量。
`ThreadWave`	`win:Double`	保留以供内部使用。
`ThroughputWave`	`win:Double`	保留以供内部使用。
`ThroughputErrorEstimate`	`win:Double`	保留以供内部使用。

[illegible]	[illegible]	[illegible]
`AverageThroughputErrorEstimate`	`win:Double`	保留以供内部使用。
`ThroughputRatio`	`win:Double`	在此间隔期间因活动工作线程计数变化导致的相对吞吐量变化。
`Confidence`	`win:Double`	ThroughputRatio 字段的有效性测量。
`NewcontrolSetting`	`win:Double`	活动工作线程数，充当活动线程数未来变化的基线。
`NewThreadWaveMagnitude`	`win:UInt16`	活动线程计数的未来变化量值。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

ThreadPoolEnqueue 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolEnqueue`	61	工作项已排入线程池队列。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`WorkID`	`win:Pointer`	指向工作请求的指针。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadPoolDequeue 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolDequeue`	62	工作项已从线程池队列中取消排队。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`WorkID`	`win:Pointer`	指向工作请求的指针。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadPoolIOEnqueue 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolIOEnqueue`	63	异步 IO 完成后，线程对 IO 完成通知进行排队。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`NativeOverlapped`	`win:Pointer`	保留以供内部使用。
`Overlapped`	`win:Pointer`	保留以供内部使用。
`MultiDequeues`	`win:Boolean`	保留以供内部使用。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadPoolIODequeue 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolIODequeue`	64	线程对 IO 完成通知取消排队。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`NativeOverlapped`	`win:Pointer`	保留以供内部使用。
`Overlapped`	`win:Pointer`	保留以供内部使用。
`MultiDequeues`	`win:Boolean`	保留以供内部使用。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadPoolIOPack 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	详细级别 (5)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadPoolIOPack`	65	调用了 ThreadPool 重叠的 IO 包。

下表显示了事件数据

[illegible]	[illegible]	[illegible]
`NativeOverlapped`	`win:Pointer`	保留以供内部使用。
`Overlapped`	`win:Pointer`	保留以供内部使用。
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadCreating 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadCreating`	70	已创建线程。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`ID`	`win:Pointer`	线程 ID
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

ThreadRunning 事件

下表显示了关键字和级别。

[illegible]	LEVEL
`ThreadingKeyword` (0x10000)	信息性 (4)

下表显示了事件信息。

[illegible]	[illegible] ID	[illegible]
`ThreadRunning`	71	线程已开始运行。

下表显示了事件数据。

[illegible]	[illegible]	[illegible]
`ID`	`win:Pointer`	线程 ID
`ClrInstanceID`	`win:UInt16`	CoreCLR 实例的唯一 ID。

.NET 运行时类型事件

2021/11/16 •

这些事件收集与加载类型相关的信息。有关如何将这些事件用于诊断的详细信息，请参阅对 .NET 应用程序进行日志记录和跟踪

TypeLoadStart 事件

[illegible]	[illegible]	LEVEL
`TypeDiagnosticKeyword` (0x8000000000)	信息性 (4)	

[illegible]	[illegible] ID	[illegible]
`TypeLoadStart`	73	类型加载已开始。

[illegible]	[illegible]	[illegible]
`TypeLoadStartID`	`win:UInt32`	类型加载操作的 ID。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

TypeLoadStop 事件

[illegible]	LEVEL
`TypeDiagnosticKeyword` (0x8000000000)	信息性 (4)

[illegible]	[illegible] ID	[illegible]
`TypeLoadStop`	74	类型加载已完成。

[illegible]	[illegible]	[illegible]
`TypeLoadStartID`	`win:UInt32`	类型加载操作的 ID（与对应的 TypeLoadStart 事件的 TypeLoadStartID 相匹配）。
`LoadLevel`	`win:UInt16`	类型加载级别。
`TypeID`	`win:UInt64`	指向该类型句柄的指针。
`TypeName`	`win:UnicodeString`	类型的名称。
`ClrInstanceID`	`win:UInt16`	CLR 或 CoreCLR 的实例的唯一 ID。

收集容器中的诊断

2021/11/16 •

其他场景中用于诊断 .NET Core 问题的相同诊断工具也适用于 Docker 容器。但是，某些工具需要特殊步骤才能在容器中运行。本文介绍如何在 Docker 容器中使用收集性能跟踪和收集转储的工具。

在容器中使用 .NET CLI 工具

这些工具适用于： ✔ .NET Core 3.0 SDK 及更高版本

.NET Core 全局 CLI 诊断工具（`dotnet-counters`、`dotnet-dump`、`dotnet-gcdump` 和 `dotnet-trace`）设计为可在多种环境中工作，应该都可以直接用于 Docker 容器。因此，这些工具是在容器中收集面向 .NET Core 3.0 或更高版本（对于 `dotnet-gcdump` 则为 3.1 或更高版本）的 .NET Core 方案的诊断信息的首选方法。

在容器中使用这些工具的唯一复杂化的因素是，它们与 .NET SDK 一起安装，而许多 Docker 容器在 .NET SDK 不存在时也能运行。解决此问题的一种简单方法是在初始 Docker 映像中安装这些工具。这些工具不需要 .NET SDK 运行，只需安装它即可。因此，可以通过多阶段生成来创建 Dockerfile，在生成阶段（.NET SDK 存在时）安装工具，然后将二进制文件复制到最终映像中。此方法的唯一缺点是增加了 Docker 映像大小。

```
# In build stage
# Install desired .NET CLI diagnostics tools
RUN dotnet tool install --tool-path /tools dotnet-trace
RUN dotnet tool install --tool-path /tools dotnet-counters
RUN dotnet tool install --tool-path /tools dotnet-dump

...

# In final stage
# Copy diagnostics tools
WORKDIR /tools
COPY --from=build /tools .
```

另外，还可以在需要时在容器中安装 .NET SDK，以便安装 CLI 工具。请注意，安装 .NET SDK 有重新安装 .NET 运行时的副作用。因此，请确保安装与容器中存在的运行时匹配的 SDK 版本。

在挎斗容器中使用 .NET Core 全局 CLI 工具

如果要使用 .NET Core 全局 CLI 诊断工具来诊断其他容器中的进程，请记住以下附加要求：

1. 容器必须共享进程命名空间（以便挎斗容器中的工具可以访问目标容器中的进程）。
2. .NET Core 全局 CLI 诊断工具需要访问 .NET Core 运行时写入到 /tmp 目录的文件，因此必须通过卷装载在目标容器和挎斗容器之间共享 /tmp 目录。例如，可以通过让容器共享公用卷或 Kubernetes emptyDir 卷来完成此操作。如果在不共享 /tmp 目录的情况下尝试使用挎斗容器中的诊断工具，将收到有关进程的错误"未运行兼容的 .NET 运行时"。

在容器中使用 `PerfCollect`

此工具适用于： ✔ .NET Core 2.1 及更高版本

`PerfCollect` 脚本对于收集性能跟踪非常有用，.NET Core 3.0 之前建议使用此工具收集跟踪。如果在容器中使用 `PerfCollect`，请记住以下要求：

- `PerfCollect` 需要 `SYS_ADMIN` 功能（以运行 `perf` 工具），因此请确保通过该功能启动容器。

- `PerfCollect` 需要在其将分析的应用启动之前设置某些环境变量。可以在 Dockerfile 中设置这些变量，也可以在启动容器时设置。由于在正常生产环境中不应设置这些变量，因此，在启动要分析的容器时添加它们是常见做法。PerfCollect 需要的两个变量是：
  - `DOTNET_PerfMapEnabled=1`
  - `DOTNET_EnableEventLog=1`

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

在挎斗容器中使用 `PerfCollect`

如果要在一个容器中运行 `PerfCollect` 来分析另一容器中的 .NET Core 进程，体验几乎相同，只是有以下差异：

- 必须为目标容器（不是运行 `PerfCollect` 的容器）设置前面所述的环境变量（`DOTNET_PerfMapEnabled` 和 `DOTNET_EnableEventLog`）。
- 运行 `PerfCollect` 的容器必须具有 `SYS_ADMIN` 功能（而目标容器则不必）。
- 这两个容器必须共享进程命名空间。

在容器中使用 `createdump`

此工具适用于： ✔ .NET Core 2.1 及更高版本

`createdump` 是 `dotnet-dump` 的一种替代方法，可用于在包含本机和托管信息的 Linux 上创建核心转储。`createdump` 工具与 .NET Core 运行时一起安装，可以在 libcoreclr.so（通常位于“/usr/share/dotnet/shared/Microsoft.NETCore.App/[version]”中）旁边找到。该工具在容器中的使用效果与在非容器化 Linux 环境中相同，唯一的差异是该工具需要 `SYS_PTRACE` 功能，因此必须通过该功能启动 Docker 容器。

在挎斗容器中使用 `createdump`

如果要使用 `createdump` 由另一容器中的进程创建转储，体验几乎相同，只是有以下差异：

- 运行 `createdump` 的容器必须具有 `SYS_PTRACE` 功能（而目标容器则不必）。
- 这两个容器必须共享进程命名空间。

调查性能计数器 (dotnet-counters)

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

安装

可采用两种方法来下载和安装 `dotnet-counters`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-counters` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-counters
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	ıı
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

> **NOTE**
> 若要在 x86 应用上使用 `dotnet-counters`，需要使用相应的 x86 版本的工具。

摘要

```
dotnet-counters [-h|--help] [--version] <command>
```

描述

`dotnet-counters` 是一个性能监视工具，用于临时运行状况监视和初级性能调查。它可以观察通过 EventCounter API 发布的性能计数器值。例如，可以快速监视 CPU 使用情况或 .NET Core 应用程序中引发的异常率，以了解在使用 `PerfView` 或 `dotnet-trace` 深入调查更严重的性能问题之前是否有任何可疑操作。

选项

- `--version`

  显示 dotnet-counters 实用工具的版本。

- `-h|--help`

显示命令行帮助。

命令

dotnet-counters collect
dotnet-counters list
dotnet-counters monitor
dotnet-counters ps

dotnet-counters collect

定期收集所选计数器的值，并将它们导出为指定的文件格式以进行后续处理。

摘要

```
dotnet-counters collect [-h|--help] [-p|--process-id] [-n|--name] [--diagnostic-port] [--refresh-interval]
[--counters <COUNTERS>] [--format] [-o|--output] [-- <command>]
```

选项

- `-p|--process-id <PID>`

  要从中收集计数器数据的进程的 ID。

- `-n|--name <name>`

  要从中收集计数器数据的进程的名称。

- `--diagnostic-port`

  要创建的诊断端口的名称。请参阅使用诊断端口，了解如何使用此选项从应用启动时开始监视计数器。

- `--refresh-interval <SECONDS>`

  更新显示的计数器之间延迟的秒数

- `--counters <COUNTERS>`

  计数器的逗号分隔列表。计数器可以指定为 `provider_name[:counter_name]`。如果使用 `provider_name` 时没有限定的计数器列表，则显示来自提供程序的所有计数器。若要发现提供程序和计数器名称，请使用 dotnet-counters list 命令。

- `--format <csv|json>`

  要导出的格式。当前可用的格式：csv 和 json。

- `-o|--output <output>`

  输出文件的名称。

- `-- <command>`（仅适用于运行 .NET 5 或更高版本的目标应用程序）

  在集合配置参数之后，用户可以追加 `--`，后跟一个命令，以启动至少具有 5.0 运行时的 .NET 应用程序。`dotnet-counters` 将启动一个进程，并收集请求的指标。这通常用于收集应用程序的启动路径的指标，并

可用于诊断或监视在主入口点前后发生的问题。

> **NOTE**
> 使用此选项监视第一个 .NET 5 进程，该进程与该工具通信，这意味着如果命令启动多个 .NET 应用程序，它将仅收集第一个应用。因此，建议在自包含应用程序上使用此选项，或使用 `dotnet exec <app.dll>` 选项。

> **NOTE**
> 通过 dotnet-counters 启动 .NET 可执行文件将重定向其输入/输出，你将无法与其 stdin/stdout 进行交互。通过 CTRL+C 或 SIGTERM 退出工具将安全地结束该工具和子进程。如果子进程在工具之前退出，工具也将退出，应可安全查看跟踪。如果需要使用 stdin/stdout，可以使用 `--diagnostic-port` 选项。有关详细信息，请参阅使用诊断端口。

> **NOTE**
> 在 Linux 和 macOS 上，此命令需要目标应用程序和 `dotnet-counters` 使用同一 `TMPDIR` 环境变量。否则，该命令将超时。

> **NOTE**
> 若要使用 `dotnet-counters` 收集指标，需要以与运行目标进程的用户相同的用户身份或以根身份运行。否则，该工具将无法与目标进程建立连接。

示例

- 以 3 秒的刷新间隔时间收集所有计数器的值，并生成 csv 输出文件：

```
> dotnet-counters collect --process-id 1902 --refresh-interval 3 --format csv

counter_list is unspecified. Monitoring all counters by default.
Starting a counter session. Press Q to quit.
```

- 将 `dotnet mvc.dll` 作为子进程启动，开始从启动中收集运行时计算器和 ASP.NET Core Hosting 计算器，并将其另存为 JSON 输出：

```
> dotnet-counters collect --format json --counters System.Runtime,Microsoft.AspNetCore.Hosting --
dotnet mvc.dll
Starting a counter session. Press Q to quit.
File saved to counter.json
```

dotnet-counters list

显示按提供程序分组的计数器名称和说明的列表。

摘要

```
dotnet-counters list [-h|--help]
```

示例

```
> dotnet-counters list
Showing well-known counters only. Specific processes may support additional counters.

System.Runtime
    cpu-usage                                  Amount of time the process has utilized the CPU (ms)
    working-set                                Amount of working set used by the process (MB)
    gc-heap-size                               Total heap size reported by the GC (MB)
    gen-0-gc-count                             Number of Gen 0 GCs per interval
    gen-1-gc-count                             Number of Gen 1 GCs per interval
    gen-2-gc-count                             Number of Gen 2 GCs per interval
    time-in-gc                                 % time in GC since the last GC
    gen-0-size                                 Gen 0 Heap Size
    gen-1-size                                 Gen 1 Heap Size
    gen-2-size                                 Gen 2 Heap Size
    loh-size                                   LOH Heap Size
    alloc-rate                                 Allocation Rate
    assembly-count                             Number of Assemblies Loaded
    exception-count                            Number of Exceptions per interval
    threadpool-thread-count                    Number of ThreadPool Threads
    monitor-lock-contention-count              Monitor Lock Contention Count
    threadpool-queue-length                    ThreadPool Work Items Queue Length
    threadpool-completed-items-count           ThreadPool Completed Work Items Count
    active-timer-count                         Active Timers Count

Microsoft.AspNetCore.Hosting
    requests-per-second                 Request rate
    total-requests                      Total number of requests
    current-requests                    Current number of requests
    failed-requests                     Failed number of requests
```

NOTE

当有已标记的进程支持 `Microsoft.AspNetCore.Hosting` 计数器时(例如当 ASP.NET Core 应用程序在主机上运行时)，将显示这些计数器。

dotnet-counters monitor

显示所选计数器的定期刷新值。

摘要

```
dotnet-counters monitor [-h|--help] [-p|--process-id] [-n|--name] [--diagnostic-port] [--refresh-interval]
[--counters] [-- <command>]
```

选项

- `-p|--process-id <PID>`

  要监视的进程的 ID。

- `-n|--name <name>`

  要监视的进程的名称。

- `--diagnostic-port`

  要创建的诊断端口的名称。请参阅使用诊断端口，了解如何使用此选项从应用启动时开始监视计数器。

- `--refresh-interval <SECONDS>`

  更新显示的计数器之间延迟的秒数

- `--counters <COUNTERS>`

  计数器的逗号分隔列表。计数器可以指定为 `provider_name[:counter_name]`。如果使用 `provider_name` 时没有限定的计数器列表，则显示来自提供程序的所有计数器。若要发现提供程序和计数器名称，请使用 dotnet-counters list 命令。

`-- <command>`（仅适用于运行 .NET 5 或更高版本的目标应用程序）

在集合配置参数之后，用户可以追加 `--`，后跟一个命令，以启动至少具有 5.0 运行时的 .NET 应用程序。`dotnet-counters` 将启动一个进程，并监视请求的指标。这通常用于收集应用程序的启动路径的指标，并可用于诊断或监视在主入口点前后发生的问题。

> **NOTE**
> 使用此选项监视第一个 .NET 5 进程，该进程与该工具通信，这意味着如果命令启动多个 .NET 应用程序，它将仅收集第一个应用。因此，建议在自包含应用程序上使用此选项，或使用 `dotnet exec <app.dll>` 选项。

> **NOTE**
> 通过 dotnet-counters 启动 .NET 可执行文件将重定向其输入/输出，你将无法与其 stdin/stdout 进行交互。通过 CTRL+C 或 SIGTERM 退出工具将安全地结束该工具和子进程。如果子进程在工具之前退出，工具也将退出。如果需要使用 stdin/stdout，可以使用 `--diagnostic-port` 选项。有关详细信息，请参阅使用诊断端口。

> **NOTE**
> 在 Linux 和 macOS 上，此命令需要目标应用程序和 `dotnet-counters` 使用同一 `TMPDIR` 环境变量。

> **NOTE**
> 若要使用 `dotnet-counters` 监视指标，需要以与运行目标进程的用户相同的用户身份或以根身份运行。

> **NOTE**
> 如果你看到一条类似于以下内容的错误消息：
> `[ERROR] System.ComponentModel.Win32Exception (299): A 32 bit processes cannot access modules of a 64 bit process.`
> ，你正在尝试使用的 `dotnet-counters` 存在与目标进程不一致的位数。请务必在安装链接中下载工具的正确位数。

示例

- 以 3 秒的刷新间隔监视 `System.Runtime` 中的所有计数器：

```
> dotnet-counters monitor --process-id 1902  --refresh-interval 3 --counters System.Runtime
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    % Time in GC since last GC (%)                                 0
    Allocation Rate (B / 1 sec)                                5,376
    CPU Usage (%)                                                  0
    Exception Count (Count / 1 sec)                                0
    GC Fragmentation (%)                                      48.467
    GC Heap Size (MB)                                              0
    Gen 0 GC Count (Count / 1 sec)                                 1
    Gen 0 Size (B)                                                24
    Gen 1 GC Count (Count / 1 sec)                                 1
    Gen 1 Size (B)                                                24
    Gen 2 GC Count (Count / 1 sec)                                 1
    Gen 2 Size (B)                                           272,000
    IL Bytes Jitted (B)                                       19,449
    LOH Size (B)                                              19,640
    Monitor Lock Contention Count (Count / 1 sec)                  0
    Number of Active Timers                                        0
    Number of Assemblies Loaded                                    7
    Number of Methods Jitted                                     166
    POH (Pinned Object Heap) Size (B)                             24
    ThreadPool Completed Work Item Count (Count / 1 sec)           0
    ThreadPool Queue Length                                        0
    ThreadPool Thread Count                                        2
    Working Set (MB)                                              19
```

- 仅监视 `System.Runtime` 中的 CPU 使用情况和 GC 堆大小：

```
> dotnet-counters monitor --process-id 1902 --counters System.Runtime[cpu-usage,gc-heap-size]

Press p to pause, r to resume, q to quit.
  Status: Running

[System.Runtime]
    CPU Usage (%)                                  24
    GC Heap Size (MB)                             811
```

- 监视用户定义的 `EventSource` 中的 `EventCounter` 值。有关详细信息，请参阅教程：使用 .NET Core 中的 EventCounters 衡量性能。

```
> dotnet-counters monitor --process-id 1902 --counters Samples-EventCounterDemos-Minimal

Press p to pause, r to resume, q to quit.
    request                                       100
```

- 查看 `dotnet-counters` 中可用的所有已知计数器：

```
> dotnet-counters list

Showing well-known counters for .NET (Core) version 3.1 only. Specific processes may support
additional counters.
System.Runtime
    cpu-usage                           The percent of process' CPU usage relative to all of the
system CPU resources [0-100]
    working-set                         Amount of working set used by the process (MB)
    gc-heap-size                        Total heap size reported by the GC (MB)
    gen-0-gc-count                      Number of Gen 0 GCs between update intervals
    gen-1-gc-count                      Number of Gen 1 GCs between update intervals
    gen-2-gc-count                      Number of Gen 2 GCs between update intervals
    time-in-gc                          % time in GC since the last GC
    gen-0-size                          Gen 0 Heap Size
    gen-1-size                          Gen 1 Heap Size
    gen-2-size                          Gen 2 Heap Size
    loh-size                            LOH Size
    alloc-rate                          Number of bytes allocated in the managed heap between update
intervals
    assembly-count                      Number of Assemblies Loaded
    exception-count                     Number of Exceptions / sec
    threadpool-thread-count             Number of ThreadPool Threads
    monitor-lock-contention-count       Number of times there were contention when trying to take the
monitor lock between update intervals
    threadpool-queue-length             ThreadPool Work Items Queue Length
    threadpool-completed-items-count    ThreadPool Completed Work Items Count
    active-timer-count                  Number of timers that are currently active

Microsoft.AspNetCore.Hosting
    requests-per-second                 Number of requests between update intervals
    total-requests                      Total number of requests
    current-requests                    Current number of requests
    failed-requests                     Failed number of requests
```

- 查看 `dotnet-counters` 中可用于 .NET 5 应用的所有已知计数器：

```
> dotnet-counters list --runtime-version 5.0

Showing well-known counters for .NET (Core) version 5.0 only. Specific processes may support
additional counters.
System.Runtime
    cpu-usage                           The percent of process' CPU usage relative to all of the
system CPU resources [0-100]
    working-set                         Amount of working set used by the process (MB)
    gc-heap-size                        Total heap size reported by the GC (MB)
    gen-0-gc-count                      Number of Gen 0 GCs between update intervals
    gen-1-gc-count                      Number of Gen 1 GCs between update intervals
    gen-2-gc-count                      Number of Gen 2 GCs between update intervals
    time-in-gc                          % time in GC since the last GC
    gen-0-size                          Gen 0 Heap Size
    gen-1-size                          Gen 1 Heap Size
    gen-2-size                          Gen 2 Heap Size
    loh-size                            LOH Size
    poh-size                            POH (Pinned Object Heap) Size
    alloc-rate                          Number of bytes allocated in the managed heap between update
intervals
    gc-fragmentation                    GC Heap Fragmentation
    assembly-count                      Number of Assemblies Loaded
    exception-count                     Number of Exceptions / sec
    threadpool-thread-count             Number of ThreadPool Threads
    monitor-lock-contention-count       Number of times there were contention when trying to take the
monitor lock between update intervals
    threadpool-queue-length             ThreadPool Work Items Queue Length
    threadpool-completed-items-count    ThreadPool Completed Work Items Count
    active-timer-count                  Number of timers that are currently active
    il-bytes-jitted                     Total IL bytes jitted
    methods-jitted-count                Number of methods jitted

Microsoft.AspNetCore.Hosting
    requests-per-second   Number of requests between update intervals
    total-requests        Total number of requests
    current-requests      Current number of requests
    failed-requests       Failed number of requests

Microsoft-AspNetCore-Server-Kestrel
    connections-per-second      Number of connections between update intervals
    total-connections           Total Connections
    tls-handshakes-per-second   Number of TLS Handshakes made between update intervals
    total-tls-handshakes        Total number of TLS handshakes made
    current-tls-handshakes      Number of currently active TLS handshakes
    failed-tls-handshakes       Total number of failed TLS handshakes
    current-connections         Number of current connections
    connection-queue-length     Length of Kestrel Connection Queue
    request-queue-length        Length total HTTP request queue

System.Net.Http
    requests-started        Total Requests Started
    requests-started-rate   Number of Requests Started between update intervals
    requests-aborted        Total Requests Aborted
    requests-aborted-rate   Number of Requests Aborted between update intervals
    current-requests        Current Requests
```

- 启动 `my-aspnet-server.exe` 并从其启动监视加载的程序集的数量（仅限 .NET 5 或更高版本）：

> **IMPORTANT**
>
> 这仅适用于运行 .NET 5 或更高版本的应用。

```
> dotnet-counters monitor --counters System.Runtime[assembly-count] -- my-aspnet-server.exe

Press p to pause, r to resume, q to quit.
  Status: Running

[System.Runtime]
    Number of Assemblies Loaded                  24
```

- 启动 `my-aspnet-server.exe`，以 `arg1` 及 `arg2` 作为命令行参数，并从其启动监视其工作集和 GC 堆大小（仅限 .NET 5 或更高版本）：

  > **IMPORTANT**
  > 这仅适用于运行 .NET 5 或更高版本的应用。

```
> dotnet-counters monitor --counters System.Runtime[working-set,gc-heap-size] -- my-aspnet-server.exe
arg1 arg2
```

```
Press p to pause, r to resume, q to quit.
  Status: Running

[System.Runtime]
    GC Heap Size (MB)                                39
    Working Set (MB)                                 59
```

dotnet-counters ps

显示可监视的 dotnet 进程的列表。

摘要

```
dotnet-counters ps [-h|--help]
```

示例

```
> dotnet-counters ps

  15683 WebApi     /home/user/repos/WebApi/WebApi
  16324 dotnet     /usr/local/share/dotnet/dotnet
```

使用诊断端口

> **IMPORTANT**
> 这仅适用于运行 .NET 5 或更高版本的应用。

诊断端口是 .NET 5 中新增的运行时功能，你可以通过它从应用启动时开始监视或收集计数器。若要使用 `dotnet-counters` 执行此操作，可以使用以上示例中所述的 `dotnet-counters <collect|monitor> -- <command>`，也可以使用 `--diagnostic-port` 选项。

使用 `dotnet-counters <collect|monitor> -- <command>` 以子进程的形式启动应用程序，是从启动时开始对其进行快速监视的最简单方法。

但是，如果想要更好地控制所监视应用的生存期(例如，仅在前 10 分钟内监视应用并继续执行)，或者如果需要使用 CLI 与应用进行交互，则使用 `--diagnostic-port` 选项可以同时控制要监视的目标应用和 `dotnet-counters`。

1. 以下命令使 dotnet-counters 创建一个名为 `myport.sock` 的诊断套接字并等待连接。

   ```
   dotnet-counters collect --diagnostic-port myport.sock
   ```

   输出：

   ```
   Waiting for connection on myport.sock
   Start an application with the following environment variable:
   DOTNET_DiagnosticPorts=/home/user/myport.sock
   ```

2. 在单独的控制台中，通过将环境变量 `DOTNET_DiagnosticPorts` 设置为 `dotnet-counters` 输出中的值，启动目标应用程序。

   ```
   export DOTNET_DiagnosticPorts=/home/user/myport.sock
   ./my-dotnet-app arg1 arg2
   ```

   这应该会使 `dotnet-counters` 开始在 `my-dotnet-app` 上收集计数器：

   ```
   Waiting for connection on myport.sock
   Start an application with the following environment variable: DOTNET_DiagnosticPorts=myport.sock
   Starting a counter session. Press Q to quit.
   ```

> **IMPORTANT**
>
> 通过 `dotnet run` 启动应用可能会产生问题，因为 dotnet CLI 可能会生成许多子进程，这些子程序不是应用，并且可以在应用之前连接到 `dotnet-counters`，从而导致应用在运行时挂起。建议直接使用应用的独立版本或使用 `dotnet exec` 来启动应用程序。

dotnet-coverage 代码覆盖率实用工具

2021/11/16 •

本文适用于： ✔ .NET Core 3.1 SDK 及更高版本

安装

若要安装最新版 `dotnet-coverage` NuGet 包，请使用 dotnet tool install 命令：

```
dotnet tool install --global dotnet-coverage
```

摘要

```
dotnet-coverage [-h, --help] [--version] <command>
```

描述

`dotnet-coverage` 工具：

- 启用在 Windows 和 Linux x64 上收集正在运行的进程的代码覆盖率数据。
- 提供代码覆盖率报表的跨平台合并。

选项

- `-h|--help`

  显示命令行帮助。

- `--version`

  显示 dotnet-coverage 实用工具的版本。

命令

dotnet-coverage collect
dotnet-coverage merge
dotnet-coverage shutdown

dotnet-coverage collect

`collect` 命令用于收集任何 .NET 进程及其子进程的代码覆盖率数据。例如，可以收集控制台应用程序或 Blazor 应用程序的代码覆盖率数据。此命令适用于 Windows（x86 和 x64）和 Linux (x64)。该命令仅支持 .NET 模块。不支持本机模块。

摘要

```
dotnet-coverage collect [-?|-h|--help] [-l|--log-file <log-file>] [-ll|--log-level <log-level>]
    [-o|--output <output>] [-f|--output-format <output-format>]
    [-s|--settings <settings>] [-id|--session-id <session-id>]
    <command>
```

自变量

- `<command>`

  用于收集其代码覆盖率数据的命令。

选项

- `-l|--log-file <log-file>`

  设置日志文件路径。如果提供目录(末尾有路径分隔符)，则将为分析下的每个进程生成新的日志文件。

- `-ll|--log-level <log-level>`

  设置日志级别。受支持的值为: `Error`、`Info` 和 `Verbose`。

- `-o|--output <output>`

  设置代码覆盖率报表输出文件。

- `-f|--output-format <output-format>`

  输出文件格式。受支持的值为: `coverage`、`xml` 和 `cobertura`。默认值为 `coverage`(可在 Visual Studio 中打开的二进制格式)。

- `-id|--session-id <session-id>`

  指定代码覆盖率会话 ID。如果未提供，该工具将生成随机 GUID。

- `-s|--settings <settings>`

  设置 XML 代码覆盖率设置的路径。

dotnet-coverage merge

`merge` 命令用于将多个代码覆盖率报表合并为一个。此命令在所有平台上均可用。此命令支持以下代码覆盖率报表格式:

- `coverage`
- `cobertura`
- `xml`

摘要

```
dotnet-coverage merge [-?|-h|--help] [-l|--log-file <log-file>] [-ll|--log-level <log-level>]
    [-o|--output <output>] [-f|--output-format <output-format>]
    [-r|--recursive] [--remove-input-files]
    <files>
```

自变量

- `<files>`

  输入代码覆盖率报表。

选项

- `-l|--log-file <log-file>`

  设置日志文件路径。如果提供目录(末尾有路径分隔符)，则将为分析下的每个进程生成新的日志文件。

- `-ll|--log-level <log-level>`

  设置日志级别。受支持的值为: `Error`、`Info` 和 `Verbose`。

- `-o|--output <output>`

  设置代码覆盖率报表输出文件。

- `-f|--output-format <output-format>`

  输出文件格式。受支持的值为: `coverage`、`xml` 和 `cobertura`。默认值为 `coverage`(可在 Visual Studio 中打开的二进制格式)。

- `-r, --recursive`

  搜索子条目中的覆盖率报表。

- `--remove-input-files`

  删除所有已合并的输入覆盖率报表。

dotnet-coverage shutdown

关闭现有代码覆盖率集合。

摘要

```
dotnet-coverage shutdown [-?|-h|--help] [-l|--log-file <log-file>] [-ll|--log-level <log-level>] <session>
```

自变量

- `<session>`

  要关闭的集合的会话 ID。

选项

- `-l|--log-file <log-file>`

  设置日志文件路径。如果提供目录(末尾有路径分隔符)，则将为分析下的每个进程生成新的日志文件。

- `-ll|--log-level <log-level>`

  设置日志级别。受支持的值为: `Error`、`Info` 和 `Verbose`。

收集代码覆盖率

使用以下命令收集任何 .NET 应用程序(如控制台或 Blazor)的代码覆盖率数据:

```
dotnet-coverage collect "dotnet run"
```

对于需要终止信号的应用程序，可以使用 `Ctrl+C`，这样仍可收集代码覆盖率数据。对于参数，可以提供最终启动 .NET 应用的任何命令。例如，它可以是 PowerShell 脚本。

会话

在仅等待消息并发送响应的 .NET 服务器上运行代码覆盖率分析时，需要一种方法来停止该服务器，以获得最终

的代码覆盖率结果。可以在本地使用 Ctrl+C，但不能在 Azure Pipelines 中使用。对于这些情况，可以使用会话。可以在启动收集时指定会话 ID，然后使用 `shutdown` 命令停止收集和服务器。

例如，假设你在 D:\serverexample\server 目录中有一个服务器，在 D:\serverexample\tests 目录中有一个测试项目。测试正在通过网络与服务器通信。可以启动服务器的代码覆盖率收集，如下所示：

```
D:\serverexample\server> dotnet-coverage collect --session-id serverdemo "dotnet run"
```

会话 ID 指定为 `serverdemo`。然后，可以按如下所示运行测试：

```
D:\serverexample\tests> dotnet test
```

最后，`serverdemo` 会话和服务器可以关闭，如下所示：

```
dotnet-coverage shutdown serverdemo
```

下面是服务器端的完整输出示例：

```
D:\serverexample\server> dotnet-coverage collect --session-id serverdemo "dotnet run"
SessionId: serverdemo
Waiting for a connection... Connected!
Received: Hello!
Sent: HELLO!
Waiting for a connection... Code coverage results: output.coverage.
D:\serverexample\server>
```

设置

使用 `collect` 命令时，可以指定具有设置的文件。设置文件可用于从代码覆盖率分析中排除某些模块或方法。格式与 runsettings 文件中数据收集器配置的格式相同。有关详细信息，请参阅自定义代码覆盖率分析。下面是一个示例：

```
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
    <CodeCoverage>
        <!--
        Additional paths to search for .pdb (symbol) files. Symbols must be found for modules to be
instrumented.
        If .pdb files are in the same folder as the .dll or .exe files, they are automatically found.
Otherwise, specify them here.
        Note that searching for symbols increases code coverage run time. So keep this small and local.
        -->
        <SymbolSearchPaths>
            <Path>C:\Users\User\Documents\Visual Studio 2012\Projects\ProjectX\bin\Debug</Path>
            <Path>\\mybuildshare\builds\ProjectX</Path>
        </SymbolSearchPaths>

        <!--
        About include/exclude lists:
        Empty "Include" clauses imply all; empty "Exclude" clauses imply none.
        Each element in the list is a regular expression (ECMAScript syntax). See /visualstudio/ide/using-
regular-expressions-in-visual-studio.
        An item must first match at least one entry in the include list to be included.
        Included items must then not match any entries in the exclude list to remain included.
        -->

        <!-- Match assembly file paths: -->
        <ModulePaths>
            <Include>
                <ModulePath>.*\.dll$</ModulePath>
```

```
                <ModulePath>.*\.dll$</ModulePath>
                <ModulePath>.*\.exe$</ModulePath>
            </Include>
            <Exclude>
                <ModulePath>.*CPPUnitTestFramework.*</ModulePath>
            </Exclude>
        </ModulePaths>

        <!-- Match fully qualified names of functions: -->
        <!-- (Use "\." to delimit namespaces in C# or Visual Basic, "::" in C++.)  -->
        <Functions>
            <Exclude>
                <Function>^Fabrikam\.UnitTest\..*</Function>
                <Function>^std::.*</Function>
                <Function>^ATL::.*</Function>
                <Function>.*::__GetTestMethodInfo.*</Function>
                <Function>^Microsoft::VisualStudio::CppCodeCoverageFramework::.*</Function>
                <Function>^Microsoft::VisualStudio::CppUnitTestFramework::.*</Function>
            </Exclude>
        </Functions>

        <!-- Match attributes on any code element: -->
        <Attributes>
            <Exclude>
            <!-- Don't forget "Attribute" at the end of the name -->
                <Attribute>^System\.Diagnostics\.DebuggerHiddenAttribute$</Attribute>
                <Attribute>^System\.Diagnostics\.DebuggerNonUserCodeAttribute$</Attribute>
                <Attribute>^System\.CodeDom\.Compiler\.GeneratedCodeAttribute$</Attribute>
                <Attribute>^System\.Diagnostics\.CodeAnalysis\.ExcludeFromCodeCoverageAttribute$</Attribute>
            </Exclude>
        </Attributes>

        <!-- Match the path of the source files in which each method is defined: -->
        <Sources>
            <Exclude>
                <Source>.*\\atlmfc\\.*</Source>
                <Source>.*\\vctools\\.*</Source>
                <Source>.*\\public\\sdk\\.*</Source>
                <Source>.*\\microsoft sdks\\.*</Source>
                <Source>.*\\vc\\include\\.*</Source>
            </Exclude>
        </Sources>

        <!-- Match the company name property in the assembly: -->
        <CompanyNames>
            <Exclude>
                <CompanyName>.*microsoft.*</CompanyName>
            </Exclude>
        </CompanyNames>

        <!-- Match the public key token of a signed assembly: -->
        <PublicKeyTokens>
            <!-- Exclude Visual Studio extensions: -->
            <Exclude>
                <PublicKeyToken>^B77A5C561934E089$</PublicKeyToken>
                <PublicKeyToken>^B03F5F7F11D50A3A$</PublicKeyToken>
                <PublicKeyToken>^31BF3856AD364E35$</PublicKeyToken>
                <PublicKeyToken>^89845DCD8080CC91$</PublicKeyToken>
                <PublicKeyToken>^71E9BCE111E9429C$</PublicKeyToken>
                <PublicKeyToken>^8F50407C4E9E73B6$</PublicKeyToken>
                <PublicKeyToken>^E361AF139669C375$</PublicKeyToken>
            </Exclude>
        </PublicKeyTokens>

    </CodeCoverage>
</Configuration>
```

合并代码覆盖率报表

可以合并 `a.coverage` 和 `b.coverage` 并将数据存储在 `merged.coverage` 中，如下所示：

```
dotnet-coverage merge -o merged.coverage a.coverage b.coverage
```

例如，如果运行类似 `dotnet test --collect "Code Coverage"` 的命令，则覆盖率报表将存储到命名为随机 GUID 的文件夹中。此类文件夹难以查找和合并。使用此工具可以合并所有项目的所有代码覆盖率报表，如下所示：

```
dotnet-coverage merge -o merged.cobertura.xml -f cobertura -r *.coverage
```

上述命令将合并当前目录和所有子目录中的所有覆盖率报表，并将结果存储到 cobertura 文件中。在 Azure Pipelines 中，可以使用“发布代码覆盖率结果”任务发布合并的 cobertura 报表。

可以使用 `merge` 命令将代码覆盖率报表转换为其他格式。例如，以下命令将二进制代码覆盖率报表转换为 XML 格式。

```
dotnet-coverage merge -o output.xml -f xml input.coverage
```

请参阅

- 自定义代码覆盖率分析
- “发布代码覆盖率结果”任务

转储收集和分析实用工具 (dotnet-dump)

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

> **NOTE**
> macOS 的 `dotnet-dump` 仅在 .NET 5 及更高版本中受支持。

安装

可采用两种方法来下载和安装 `dotnet-dump`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-dump` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-dump
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

> **NOTE**
> 若要在 x86 应用上使用 `dotnet-dump`，需要使用相应的 x86 版本的工具。

摘要

```
dotnet-dump [-h|--help] [--version] <command>
```

描述

`dotnet-dump` 全局工具是在未涉及任何本机调试器（如 Linux 上的 `lldb`）的情况下收集和分析 Windows 和 Linux 转储的方法。在 `lldb` 无法正常运行的平台（如 Alpine Linux）上，此工具非常重要。借助 `dotnet-dump` 工具，可以运行 SOS 命令来分析崩溃和垃圾回收器 (GC)，但它不是本机调试器，因此不支持显示本机堆栈帧之类的操作。

选项

- `--version`

  显示 dotnet-dump 实用工具的版本。

- `-h|--help`

  显示命令行帮助。

命令

dotnet-dump collect
dotnet-dump analyze

dotnet-dump collect

从进程捕获转储。

摘要

```
dotnet-dump collect [-h|--help] [-p|--process-id] [-n|--name] [--type] [-o|--output] [--diag]
```

选项

- `-h|--help`

  显示命令行帮助。

- `-p|--process-id <PID>`

  指定从中收集转储的进程的 ID 号。

- `-n|--name <name>`

  指定从中收集转储的进程的名称。

- `--type <Full|Heap|Mini>`

  指定转储类型，它确定从进程收集的信息的类型。有三种类型：

  - `Full` - 最大的转储，包含所有内存（包括模块映像）。
  - `Heap` - 大型且相对全面的转储，其中包含模块列表、线程列表、所有堆栈、异常信息、句柄信息和除映射图像以外的所有内存。
  - `Mini` - 小型转储，其中包含模块列表、线程列表、异常信息和所有堆栈。

  如果未指定，则 `Full` 为默认类型。

- `-o|--output <output_dump_path>`

  应在其中写入收集的转储的完整路径和文件名。

  如果未指定：

  - 在 Windows 上默认为 .\dump_YYYYMMDD_HHMMSS.dmp 。
  - 在 Linux 上默认为 ./core_YYYYMMDD_HHMMSS 。

  YYYYMMDD 为年/月/日，HHMMSS 为小时/分钟/秒。

- `--diag`

  启用转储收集诊断日志记录。

> **NOTE**
> 在 Linux 和 macOS 上，此命令需要目标应用程序和 `dotnet-dump` 使用同一 `TMPDIR` 环境变量。否则，该命令将超时。

> **NOTE**
> 若要使用 `dotnet-dump` 收集转储，需要以与运行目标进程的用户相同的用户身份或以根身份运行。否则，该工具将无法与目标进程建立连接。

dotnet-dump analyze

启动交互式 shell 以了解转储。shell 接受各种 SOS 命令。

摘要

```
dotnet-dump analyze <dump_path> [-h|--help] [-c|--command]
```

自变量

- `<dump_path>`

  指定要分析的转储文件的路径。

选项

- `-c|--command <debug_command>`

  指定要在启动时在 shell 中运行的命令。

分析 SOS 命令

"	"
`soshelp|help`	显示所有可用命令
`soshelp|help <command>`	执行指定的命令。
`exit|quit`	退出交互模式。
`clrstack <arguments>`	仅提供托管代码的堆栈跟踪。
`clrthreads <arguments>`	列出正在运行的托管线程。
`dumpasync <arguments>`	显示有关垃圾回收堆上异步状态机的信息。
`dumpassembly <arguments>`	显示有关指定地址处程序集的详细信息。
`dumpclass <arguments>`	显示有关指定地址处的 `EEClass` 结构的信息。
`dumpdelegate <arguments>`	显示有关指定地址处的委托的信息。

“	“
`dumpdomain <arguments>`	显示所有 AppDomain 和指定域中的所有程序集的信息。
`dumpheap <arguments>`	显示有关垃圾回收堆的信息和有关对象的收集统计信息。
`dumpil <arguments>`	显示与托管方法关联的 Microsoft 中间语言 (MSIL)。
`dumplog <arguments>`	将内存中压力日志的内容写入到指定文件。
`dumpmd <arguments>`	显示有关指定地址处的 `MethodDesc` 结构的信息。
`dumpmodule <arguments>`	显示有关指定地址处的模块的信息。
`dumpmt <arguments>`	显示有关指定地址处的 `MethodTable` 的信息。
`dumpobj <arguments>`	显示有关位于指定地址处的对象的信息。
`dso|dumpstackobjects <arguments>`	显示在当前堆栈的边界内找到的所有托管对象。
`eeheap <arguments>`	显示有关内部运行时数据结构所使用的进程内存的信息。
`finalizequeue <arguments>`	显示所有已进行终结注册的对象。
`gcroot <arguments>`	显示有关对指定地址处的对象的引用(或根)的信息。
`gcwhere <arguments>`	显示传入参数在 GC 堆中的位置。
`ip2md <arguments>`	显示 JIT 代码中指定地址处的 `MethodDesc` 结构。
`histclear <arguments>`	释放由 `hist*` 命令系列使用的任何资源。
`histinit <arguments>`	从保存在调试对象中的压力日志初始化 SOS 结构。
`histobj <arguments>`	显示与 `<arguments>` 相关的垃圾回收压力日志重定位。
`histobjfind <arguments>`	显示在指定地址处引用对象的所有日志项。
`histroot <arguments>`	显示与指定根的提升和重定位相关的信息。
`lm|modules`	显示进程中的本机模块。
`name2ee <arguments>`	显示 `<argument>` 的 `MethodTable` 和 `EEClass` 结构。
`pe|printexception <arguments>`	显示从 Exception 类派生的 `<argument>` 的任何对象。
`setsymbolserver <arguments>`	启用符号服务器支持
`syncblk <arguments>`	显示 SyncBlock 持有者信息。

〃	〃
`threads|setthread <threadid>`	设置或显示 SOS 命令的当前线程 ID。

> **NOTE**
> 可以在适用于 .NET 的 SOS 调试扩展中找到其他详细信息。

使用 `dotnet-dump`

第一步是收集转储。如果已生成核心转储，则可以跳过此步骤。操作系统或 .NET Core 运行时的内置转储生成功能均可以创建核心转储。

```
$ dotnet-dump collect --process-id 1902
Writing minidump to file ./core_20190226_135837
Written 98983936 bytes (24166 pages) to core file
Complete
```

现在，使用 `analyze` 命令分析核心转储：

```
$ dotnet-dump analyze ./core_20190226_135850
Loading core dump: ./core_20190226_135850
Ready to process analysis commands. Type 'help' to list available commands or 'help [command]' to get
detailed help on a command.
Type 'quit' or 'exit' to exit the session.
>
```

此操作会显示一个交互式会话，该会话接受以下类似命令：

```
> clrstack
OS Thread Id: 0x573d (0)
    Child SP               IP Call Site
00007FFD28B42C58 00007fb22c1a8ed9 [HelperMethodFrame_PROTECTOBJ: 00007ffd28b42c58]
System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[], System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67 System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[], System.Globalization.CultureInfo)
[/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs @ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.Program.Foo4(System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.Program.Foo2(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.Program.Foo1(Int32, System.String)
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.Program.Main(System.String[])
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]
00007FFD28B43210 00007fb22aa9cedf [GCFrame: 00007ffd28b43210]
00007FFD28B43610 00007fb22aa9cedf [GCFrame: 00007ffd28b43610]
```

查看已终止应用的未经处理的异常：

```
> pe -lines
Exception object: 00007fb18c038590
Exception type:   System.Reflection.TargetInvocationException
Message:          Exception has been thrown by the target of an invocation.
InnerException:   System.Exception, Use !PrintException 00007FB18C038368 to see more.
StackTrace (generated):
SP               IP               Function
00007FFD28B42DD0 0000000000000000
System.Private.CoreLib.dll!System.RuntimeMethodHandle.InvokeMethod(System.Object, System.Object[],
System.Signature, Boolean, Boolean)
00007FFD28B42DD0 00007FB1B1334F67
System.Private.CoreLib.dll!System.Reflection.RuntimeMethodInfo.Invoke(System.Object,
System.Reflection.BindingFlags, System.Reflection.Binder, System.Object[],
System.Globalization.CultureInfo)+0xa7
[/root/coreclr/src/mscorlib/src/System/Reflection/RuntimeMethodInfo.cs @ 472]
00007FFD28B42E20 00007FB1B18D33ED SymbolTestApp.dll!SymbolTestApp.Program.Foo4(System.String)+0x15d
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 54]
00007FFD28B42ED0 00007FB1B18D2FC4 SymbolTestApp.dll!SymbolTestApp.Program.Foo2(Int32, System.String)+0x34
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 29]
00007FFD28B42F00 00007FB1B18D2F5A SymbolTestApp.dll!SymbolTestApp.Program.Foo1(Int32, System.String)+0x3a
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 24]
00007FFD28B42F30 00007FB1B18D168E SymbolTestApp.dll!SymbolTestApp.Program.Main(System.String[])+0x6e
[/home/mikem/builds/SymbolTestApp/SymbolTestApp/SymbolTestApp.cs @ 19]

StackTraceString: <none>
HResult: 80131604
```

Docker 的特殊说明

如果在 Docker 下运行，则转储集合需要 `SYS_PTRACE` 功能（`--cap-add=SYS_PTRACE` 或 `--privileged`）。

在 Microsoft .NET SDK Linux Docker 映像上，某些 `dotnet-dump` 命令可能会引发以下异常：

> 未经处理的异常：System.DllNotFoundException:无法加载共享库“libdl.so”或其依赖项之一的异常。

若要解决此问题，请安装“libc6-dev”包。

另请参阅

- 收集和分析内存转储博客
- 堆分析工具 (dotnet-gcdump)

堆分析工具 (dotnet-gcdump)

2021/11/16 •

本文适用于： ✔ .NET Core 3.1 SDK 及更高版本

安装

可采用两种方法来下载和安装 `dotnet-gcdump`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-gcdump` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-gcdump
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	ıı
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

> **NOTE**
> 若要在 x86 应用上使用 `dotnet-gcdump`，需要使用相应的 x86 版本的工具。

摘要

```
dotnet-gcdump [-h|--help] [--version] <command>
```

说明

`dotnet-gcdump` 全局工具使用 EventPipe 收集实时 .NET 进程的 GC（垃圾回收器）转储。创建 GC 转储时需要在目标进程中触发 GC、开启特殊事件并从事件流中重新生成对象根图。此过程允许在进程运行时以最小的开销收集 GC 转储。这些转储对于以下几种情况非常有用：

- 比较多个时间点堆上的对象数。
- 分析对象的根（回答诸如“还有哪些引用此类型的内容？”等问题）。
- 收集有关堆上的对象计数的常规统计信息。

查看从 dotnet-gcdump 捕获的 GC 转储

在 Windows 上，可以在 PerfView 中查看 `.gcdump` 文件，以便进行分析，也可在 Visual Studio 中查看该文件。目前，无法在非 Windows 平台上打开 `.gcdump`。

可以收集多个 `.gcdump`，并在 Visual Studio 中同时打开它们以获取比较体验。

选项

- `--version`

  显示 `dotnet-gcdump` 实用工具的版本。

- `-h|--help`

  显示命令行帮助。

`dotnet-gcdump collect`

从当前正在运行的进程中收集 GC 转储。

> **WARNING**
> 为了遍历 GC 堆，此命令将触发第 2 代(完整)垃圾回收，这可能会使运行时长时间挂起，尤其是在 GC 堆很大的情况下。如果 GC 堆很大，请不要在对性能要求高的环境中使用此命令。

摘要

```
dotnet-gcdump collect [-h|--help] [-p|--process-id <pid>] [-o|--output <gcdump-file-path>] [-v|--verbose] [-t|--timeout <timeout>] [-n|--name <name>]
```

选项

- `-h|--help`

  显示命令行帮助。

- `-p|--process-id <pid>`

  可从中收集 GC 转储的进程 ID。

- `-o|--output <gcdump-file-path>`

  应写入收集 GC 转储的路径。默认为 .\YYYYMMDD_HHMMSS_<pid>.gcdump。

- `-v|--verbose`

  收集 GC 转储时输出日志。

- `-t|--timeout <timeout>`

  如果收集 GC 转储的时间超过了此秒数，则放弃收集。默认值为 30。

- `-n|--name <name>`

  可从中收集 GC 转储的进程的名称。

> **NOTE**
> 在 Linux 和 macOS 上，此命令需要目标应用程序和 `dotnet-gcdump` 使用同一 `TMPDIR` 环境变量。否则，该命令将超时。

> **NOTE**
> 若要使用 `dotnet-gcdump` 收集 GC 转储，需要以与运行目标进程的用户相同的用户身份或以根身份运行。否则，该工具将无法与目标进程建立连接。

`dotnet-gcdump ps`

列出可为其收集 GC 转储的 dotnet 进程。

摘要

```
dotnet-gcdump ps
```

`dotnet-gcdump report <gcdump_filename>`

从以前生成的 GC 转储或从正在运行的进程生成报表，并将其写入 `stdout`。

摘要

```
dotnet-gcdump report [-h|--help] [-p|--process-id <pid>] [-t|--report-type <HeapStat>]
```

选项

- `-h|--help`

  显示命令行帮助。

- `-p|--process-id <pid>`

  可从中收集 GC 转储的进程 ID。

- `-t|--report-type <HeapStat>`

  可生成报表的类型。可用选项：heapstat（默认）。

疑难解答

- gcdump 中没有类型信息。

  在 .NET Core 3.1 之前，存在一个问题，即使用 EventPipe 调用 gcdump 时，gcdump 之间的类型缓存没有清除。这导致确定类型信息所需的事件未发送给第二个和后续 gcdump。此问题已在 .NET Core 3.1-preview2 中得以修复。

- COM 和静态类型不在 GC 转储中。

  在 .NET Core 3.1-preview2 之前，存在一个问题，即通过 EventPipe 调用 GC 转储时，不会发送静态和 COM 类型。此问题已在 .NET Core 3.1-preview2 中得以修复。

dotnet-trace 性能分析实用工具

2021/11/16 •

本文适用于：✔ .NET Core 3.0 SDK 及更高版本

安装

可采用两种方法来下载和安装 `dotnet-trace`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-trace` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-trace
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	体
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

> **NOTE**
> 若要在 x86 应用上使用 `dotnet-trace`，需要使用相应的 x86 版本的工具。

摘要

```
dotnet-trace [-h, --help] [--version] <command>
```

描述

`dotnet-trace` 工具：

- 是一个跨平台的 .NET Core 工具。
- 在不使用本机探查器的情况下启用正在运行的进程的 .NET Core 跟踪集合。
- 是基于 .NET Core 运行时的 `EventPipe` 构建的。
- 在 Windows、Linux 或 macOS 上提供相同体验。

选项

- `-h|--help`

显示命令行帮助。

- `--version`

  显示 dotnet-dump 实用工具的版本。

命令

"
dotnet-trace collect
dotnet-trace convert
dotnet-trace ps
dotnet-trace list-profiles

dotnet-trace collect

从正在运行的进程中收集诊断跟踪，或者启动子进程并对其进行跟踪(仅限 .NET 5+)。若要让工具运行子进程并自其启动时对其进行跟踪，请将 `--` 追加到 collect 命令。

摘要

```
dotnet-trace collect [--buffersize <size>] [--clreventlevel <clreventlevel>] [--clrevents <clrevents>]
    [--format <Chromium|NetTrace|Speedscope>] [-h|--help]
    [-n, --name <name>] [--diagnostic-port] [-o|--output <trace-file-path>] [-p|--process-id <pid>]
    [--profile <profile-name>] [--providers <list-of-comma-separated-providers>]
    [--show-child-io]
    [-- <command>] (for target applications running .NET 5 or later)
```

选项

- `--buffersize <size>`

  设置内存中循环缓冲区的大小(以 MB 表示)。默认值为 256 MB。

  > **NOTE**
  > 如果目标进程过于频繁地写入事件，则它可能会溢出此缓冲区，并且某些事件可能会被丢弃。如果丢弃的事件过多，请增加缓冲区大小，查看丢弃的事件数是否减少。如果丢弃的事件数未随缓冲区大小的增加而减少，则可能是因为读取器的速度较慢，导致无法刷新目标进程的缓冲区。

- `--clreventlevel <clreventlevel>`

  要发出的 CLR 事件的详细级别。

- `--clrevents <clrevents>`

  要启用的 CLR 运行时提供程序关键字列表，以 `+` 符号分隔。这是一个简单映射，支持通过字符串别名而不是其十六进制值指定事件关键字。例如，`dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:3:4` 请求与 `dotnet-trace collect --clrevents gc+gchandle --clreventlevel informational` 相同的事件集。下表显示可用关键字的列表：

[illegible]	[illegible]
`gc`	`0x1`
`gchandle`	`0x2`
`fusion`	`0x4`
`loader`	`0x8`
`jit`	`0x10`
`ngen`	`0x20`
`startenumeration`	`0x40`
`endenumeration`	`0x80`
`security`	`0x400`
`appdomainresourcemanagement`	`0x800`
`jittracing`	`0x1000`
`interop`	`0x2000`
`contention`	`0x4000`
`exception`	`0x8000`
`threading`	`0x10000`
`jittedmethodiltonativemap`	`0x20000`
`overrideandsuppressngenevents`	`0x40000`
`type`	`0x80000`
`gcheapdump`	`0x100000`
`gcsampledobjectallocationhigh`	`0x200000`
`gcheapsurvivalandmovement`	`0x400000`
`gcheapcollect`	`0x800000`
`gcheapandtypenames`	`0x1000000`
`gcsampledobjectallocationlow`	`0x2000000`

[illegible]	[illegible]
`perftrack`	`0x20000000`
`stack`	`0x40000000`
`threadtransfer`	`0x80000000`
`debugger`	`0x100000000`
`monitoring`	`0x200000000`
`codesymbols`	`0x400000000`
`eventsource`	`0x800000000`
`compilation`	`0x1000000000`
`compilationdiagnostic`	`0x2000000000`
`methoddiagnostic`	`0x4000000000`
`typediagnostic`	`0x8000000000`

有关 CLR 提供程序的详细信息，请参阅 .NET 运行时提供程序参考文档。

- `--format {Chromium|NetTrace|Speedscope}`

  设置跟踪文件转换的输出格式。默认值为 `NetTrace`。

- `-n, --name <name>`

  从中收集跟踪的进程的名称。

- `--diagnostic-port <path-to-port>`

  要创建的诊断端口的名称。请参阅使用诊断端口从应用启动时开始收集跟踪，以了解如何使用此选项从应用启动时开始收集跟踪。

- `-o|--output <trace-file-path>`

  收集的跟踪数据的输出路径。如果未指定，则默认为 `trace.nettrace`。

- `-p|--process-id <PID>`

  从中收集跟踪的进程 ID。

- `--profile <profile-name>`

  一组命名的预定义提供程序配置，允许简明地指定常见跟踪方案。可用配置文件如下：

[illegible]	[illegible]
`cpu-sampling`	可用于跟踪 CPU 使用情况和一般 .NET 运行时信息。如果未指定配置文件或提供程序，则这是默认选项。

[[[[	[[
`gc-verbose`	跟踪 GC 集合并示例对象分配。
`gc-collect`	仅以极低的开销跟踪 GC 集合。

- `--providers <list-of-comma-separated-providers>`

  要启用的 `EventPipe` 提供程序的以逗号分隔的列表。这些提供程序会补充 `--profile <profile-name>` 隐含的任何提供程序。如果特定提供程序存在任何不一致的情况，此配置将优先于配置文件中的隐式配置。

  此提供程序列表的格式为：

  - `Provider[,Provider]`
  - `Provider` 的格式为：`KnownProviderName[:Flags[:Level][:KeyValueArgs]]`。
  - `KeyValueArgs` 的格式为：`[key1=value1][;key2=value2]`。

  若要详细了解 .NET 中的一些已知提供程序，请参阅已知事件提供程序。

- `-- <command>`（仅适用于运行 .NET 5 的目标应用程序）

  在集合配置参数之后，用户可以追加 `--`，后跟一个命令，以启动至少具有 5.0 运行时的 .NET 应用程序。这在过程早期发生诊断问题（如启动性能问题或程序集加载程序和绑定器错误）时可能会有所帮助。

  > **NOTE**
  > 使用此选项监视第一个 .NET 5 进程，该进程与该工具通信，这意味着如果命令启动多个 .NET 应用程序，它将仅收集第一个应用。因此，建议在自包含应用程序上使用此选项，或使用 `dotnet exec <app.dll>` 选项。

- `--show-child-io`

  显示当前控制台中已启动的子进程的输入和输出流。

> **NOTE**
> 对于大型应用程序，停止跟踪可能需要较长时间（可达数分钟）。运行时需要为跟踪中捕获的所有托管代码发送类型缓存。

> **NOTE**
> 在 Linux 和 macOS 上，此命令需要目标应用程序和 `dotnet-trace` 使用同一 `TMPDIR` 环境变量。否则，该命令将超时。

> **NOTE**
> 若要使用 `dotnet-trace` 收集跟踪，需要以与运行目标进程的用户相同的用户身份或以根身份运行。否则，该工具将无法与目标进程建立连接。

> **NOTE**
> 如果你看到一条类似于以下内容的错误消息：
> `[ERROR] System.ComponentModel.Win32Exception (299): A 32 bit processes cannot access modules of a 64 bit process.`
> ，你正在尝试使用的 `dotnet-trace` 存在与目标进程不一致的位数。请务必在安装链接中下载工具的正确位数。

dotnet-trace convert

将 `nettrace` 跟踪转换为备用格式，以便用于备用跟踪分析工具。

摘要

```
dotnet-trace convert [<input-filename>] [--format <Chromium|NetTrace|Speedscope>] [-h|--help] [-o|--output
<output-filename>]
```

自变量

- `<input-filename>`

  要转换的输入跟踪文件。默认为 trace.nettrace 。

选项

- `--format <Chromium|NetTrace|Speedscope>`

  设置跟踪文件转换的输出格式。

- `-o|--output <output-filename>`

  输出文件名。将添加目标格式的扩展。

> **NOTE**
> 将 `nettrace` 文件转换为 `chromium` 或 `speedscope` 文件是不可逆操作。`speedscope` 和 `chromium` 文件不具备重新构造 `nettrace` 文件所需的全部信息。但是，`convert` 命令保留了原始 `nettrace` 文件，因此，如果打算将来打开该文件，请不要将其删除。

dotnet-trace ps

列出可从中收集跟踪的 dotnet 进程。

摘要

```
dotnet-trace ps [-h|--help]
```

dotnet-trace list-profiles

列出预生成的跟踪配置文件，并描述每个配置文件中包含的提供程序和筛选器。

摘要

```
dotnet-trace list-profiles [-h|--help]
```

使用 dotnet-trace 收集跟踪

若要使用 `dotnet-trace` 收集跟踪，请执行以下操作：

- 需要首先查找要从中收集跟踪的 .NET Core 应用程序的进程标识符 (PID)。
  - 例如，在 Windows 上，可以使用任务管理器或 `tasklist` 命令。
  - 在 Linux 上，使用 `ps` 命令。
  - dotnet-trace ps

- 运行下面的命令：

```
dotnet-trace collect --process-id <PID>
```

前面的命令生成类似于以下内容的输出：

```
Press <Enter> to exit...
Connecting to process: <Full-Path-To-Process-Being-Profiled>/dotnet.exe
Collecting to file: <Full-Path-To-Trace>/trace.nettrace
Session Id: <SessionId>
Recording trace 721.025 (KB)
```

- 按 `<Enter>` 键停止收集。`dotnet-trace` 会将完成将事件记录到 trace.nettrace 文件中 。

启动子应用程序，并使用 dotnet-trace 从启动中收集跟踪

> **IMPORTANT**
> 这仅适用于运行 .NET 5 或更高版本的应用。

有时，从进程启动中收集进程的跟踪可能很有用。对于运行 .NET 5 或更高版本的应用，可以使用 dotnet-trace 来做到这一点。

这将启动 `hello.exe` 并以 `arg1` 和 `arg2` 作为其命令行参数，从其运行时启动中收集跟踪：

```
dotnet-trace collect -- hello.exe arg1 arg2
```

前面的命令生成类似于以下内容的输出：

```
No profile or providers specified, defaulting to trace profile 'cpu-sampling'

Provider Name                           Keywords            Level               Enabled By
Microsoft-DotNETCore-SampleProfiler     0x0000F00000000000  Informational(4)    --profile
Microsoft-Windows-DotNETRuntime         0x00000014C14FCCBD  Informational(4)    --profile

Process        : E:\temp\gcperfsim\bin\Debug\net5.0\gcperfsim.exe
Output File    : E:\temp\gcperfsim\trace.nettrace


[00:00:00:05]   Recording trace 122.244  (KB)
Press <Enter> or <Ctrl+C> to exit...
```

可以通过按 `<Enter>` 或 `<Ctrl + C>` 键来停止收集跟踪。此操作还将退出 `hello.exe`。

> **NOTE**
> 通过 dotnet-trace 启动 `hello.exe` 会重定向其输入/输出；默认情况下，你将无法在控制台上与其交互。使用 `--show-child-io` 开关与其 stdin/stdout 交互。通过 CTRL+C 或 SIGTERM 退出工具将安全地结束该工具和子进程。如果子进程在工具之前退出，工具也将退出，应可安全查看跟踪。

使用诊断端口从应用启动时开始收集跟踪

> **IMPORTANT**
> 这仅适用于运行 .NET 5 或更高版本的应用。

诊断端口是 .NET 5 中新增的运行时功能，你可以通过它从应用启动时开始跟踪。若要使用 `dotnet-trace` 执行此操作，可以使用以上示例中所述的 `dotnet-trace collect -- <command>`，也可以使用 `--diagnostic-port` 选项。

使用 `dotnet-trace <collect|monitor> -- <command>` 以子进程的形式启动应用程序，是从启动时开始对该应用程序进行快速跟踪的最简单方法。

但是，如果想要更好地控制所跟踪应用的生存期(例如，仅在前 10 分钟内监视应用并继续执行)，或者如果需要使用 CLI 与应用进行交互，则使用 `--diagnostic-port` 选项可以同时控制要监视的目标应用和 `dotnet-trace`。

1. 以下命令使 `dotnet-trace` 创建一个名为 `myport.sock` 的诊断套接字并等待连接。

```
dotnet-trace collect --diagnostic-port myport.sock
```

   输出：

```
Waiting for connection on myport.sock
Start an application with the following environment variable:
DOTNET_DiagnosticPorts=/home/user/myport.sock
```

2. 在单独的控制台中，通过将环境变量 `DOTNET_DiagnosticPorts` 设置为 `dotnet-trace` 输出中的值，启动目标应用程序。

```
export DOTNET_DiagnosticPorts=/home/user/myport.sock
./my-dotnet-app arg1 arg2
```

   这应该会使 `dotnet-trace` 开始跟踪 `my-dotnet-app`：

```
Waiting for connection on myport.sock
Start an application with the following environment variable: DOTNET_DiagnosticPorts=myport.sock
Starting a counter session. Press Q to quit.
```

> **IMPORTANT**
> 通过 `dotnet run` 启动应用可能会产生问题，因为 dotnet CLI 可能会生成许多子进程，这些子程序不是应用，并且可以在应用之前连接到 `dotnet-trace`，从而导致应用在运行时挂起。建议直接使用应用的独立版本或使用 `dotnet exec` 来启动应用程序。

查看由 dotnet-trace 捕获的跟踪

在 Windows 上，可以使用 PerfView 查看 .nettrace 文件以进行分析：对于其他平台上收集的跟踪，可以将跟踪文件移动到 Windows 计算机上，以在 PerfView 上进行查看。

在 Linux 上，可以通过将 `dotnet-trace` 的输出格式更改为 `speedscope` 来查看跟踪。可以使用 `-f|--format` 选项更改输出文件格式 - `-f speedscope` 会使 `dotnet-trace` 生成 `speedscope` 文件。可以在 `nettrace`(默认选项)

和 `speedscope` 之间进行选择。可以在 https://www.speedscope.app 打开 `Speedscope` 文件。

> **NOTE**
> .NET Core 运行时以 `nettrace` 格式生成跟踪。跟踪完成后，跟踪将转换为 speedscope（如果指定）。由于某些转换可能会导致数据丢失，因此，原始 `nettrace` 文件将保留在转换后的文件旁边。

使用 dotnet-trace 收集随时间变化的计数器值

`dotnet-trace` 可以：

- 在性能敏感的环境中使用 `EventCounter` 进行基本运行状况监视。例如，在生产环境中。
- 收集跟踪，这样就不需要实时查看。

例如，若要收集运行时性能计数器值，请使用以下命令：

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1
```

上述命令通知运行时计数器每秒报告一次，以便进行轻量型运行状况监视。通过使用较大的值（例如 60）替换 `EventCounterIntervalSec=1`，可以收集计数器数据中粒度较小的跟踪。

以下命令比以上命令产生更小的开销和跟踪大小：

```
dotnet-trace collect --process-id <PID> --providers System.Runtime:0:1:EventCounterIntervalSec=1,Microsoft-Windows-DotNETRuntime:0:1,Microsoft-DotNETCore-SampleProfiler:0:1
```

以上命令会禁用运行时事件和托管堆栈探查器。

使用 .rsp 文件来避免键入长命令

可以使用包含要传递的参数的 `.rsp` 文件启动 `dotnet-trace`。当启用需要较长参数的提供程序时，或在使用可去除字符的 shell 环境时，这很有用。

例如，以下提供程序在每次要跟踪时都可能要繁琐地键入内容：

```
dotnet-trace collect --providers Microsoft-Diagnostics-DiagnosticSource:0x3:5:FilterAndPayloadSpecs="SqlClientDiagnosticListener/System.Data.SqlClient.WriteCommandBefore@Activity1Start:-Command;Command.CommandText;ConnectionId;Operation;Command.Connection.ServerVersion;Command.CommandTimeout;Command.CommandType;Command.Connection.ConnectionString;Command.Connection.Database;Command.Connection.DataSource;Command.Connection.PacketSize\r\nSqlClientDiagnosticListener/System.Data.SqlClient.WriteCommandAfter@Activity1Stop:\r\nMicrosoft.EntityFrameworkCore/Microsoft.EntityFrameworkCore.Database.Command.CommandExecuting@Activity2Start:-Command;Command.CommandText;ConnectionId;IsAsync;Command.Connection.ClientConnectionId;Command.Connection.ServerVersion;Command.CommandTimeout;Command.CommandType;Command.Connection.ConnectionString;Command.Connection.Database;Command.Connection.DataSource;Command.Connection.PacketSize\r\nMicrosoft.EntityFrameworkCore/Microsoft.EntityFrameworkCore.Database.Command.CommandExecuted@Activity2Stop:",OtherProvider,AnotherProvider
```

此外，前一个示例包含 `"` 作为参数的一部分。由于每个 shell 对引号的处理不同，因此在使用不同的 shell 时可能会遇到各种问题。例如，在 `zsh` 中输入的命令与 `cmd` 中的命令不同。

可以将以下文本保存到名为 `myprofile.rsp` 的文件中，而不必每次都键入此内容。

```
--providers
Microsoft-Diagnostics-
DiagnosticSource:0x3:5:FilterAndPayloadSpecs="SqlClientDiagnosticListener/System.Data.SqlClient.WriteCommand
Before@Activity1Start:-
Command;Command.CommandText;ConnectionId;Operation;Command.Connection.ServerVersion;Command.CommandTimeout;C
ommand.CommandType;Command.Connection.ConnectionString;Command.Connection.Database;Command.Connection.DataSo
urce;Command.Connection.PacketSize\r\nSqlClientDiagnosticListener/System.Data.SqlClient.WriteCommandAfter@Ac
tivity1Stop:\r\nMicrosoft.EntityFrameworkCore/Microsoft.EntityFrameworkCore.Database.Command.CommandExecutin
g@Activity2Start:-
Command;Command.CommandText;ConnectionId;IsAsync;Command.Connection.ClientConnectionId;Command.Connection.Se
rverVersion;Command.CommandTimeout;Command.CommandType;Command.Connection.ConnectionString;Command.Connectio
n.Database;Command.Connection.DataSource;Command.Connection.PacketSize\r\nMicrosoft.EntityFrameworkCore/Micr
osoft.EntityFrameworkCore.Database.Command.CommandExecuted@Activity2Stop:",OtherProvider,AnotherProvider
```

保存 `myprofile.rsp` 后，可以使用以下命令通过此配置启动 `dotnet-trace`：

```
dotnet-trace @myprofile.rsp
```

另请参阅

- .NET 的已知事件提供程序

检查托管堆栈跟踪 (dotnet-stack)

2021/11/16 •

本文适用于：✔ .NET Core 3.0 及更高版本

安装

可采用两种方法来下载和安装 `dotnet-stack`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-stack` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-stack
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	ıı
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

摘要

```
dotnet-stack [-h, --help] [--version] <command>
```

描述

`dotnet-stack` 工具：

- 是一个跨平台的 .NET Core 工具。
- 为目标 .NET 进程中的所有线程捕获和打印托管堆栈。
- 利用 .NET Core 运行时提供的 `EventPipe` 跟踪。

选项

- `-h|--help`

  显示命令行帮助。

- `--version`

  显示 dotnet-stack 实用工具的版本。

命令

“	“
dotnet-stack 报告	打印目标进程中每个线程的堆栈跟踪。
dotnet-stack ps	列出可从中收集跟踪的 dotnet 进程。

dotnet-stack 报告

打印目标进程中每个线程的堆栈跟踪。

摘要

```
dotnet-stack report -p|--process-id <pid>
                    -n|--name <process-name>
                    [-h|--help]
```

选项

- `-n, --name <name>`

  从中收集跟踪的进程的名称。

- `-p|--process-id <PID>`

  从中收集跟踪的进程 ID。

dotnet-stack ps

列出可从中收集跟踪的 dotnet 进程。

摘要

```
dotnet-stack ps [-h|--help]
```

使用 dotnet-stack 报告托管堆栈

使用 `dotnet-stack` 报告托管堆栈：

- 获取要从中报告堆栈的 .NET Core 应用程序的进程标识符 (PID)。
  - 例如，在 Windows 上，可以使用任务管理器或 `tasklist` 命令。
  - 在 Linux 上，使用 `ps` 命令。
  - dotnet-stack ps
- 运行下面的命令：

  ```
  dotnet-stack report --process-id <PID>
  ```

  前面的命令生成类似于以下内容的输出：

```
Thread (0x48839B):
  [Native Frames]
  System.Console!System.IO.StdInReader.ReadKey(bool&)
  System.Console!System.IO.SyncTextReader.ReadKey(bool&)
  System.Console!System.ConsolePal.ReadKey(bool)
  System.Console!System.Console.ReadKey()
  StackTracee!Tracee.Program.Main(class System.String[])
```

`dotnet-stack` 的输出遵循以下格式：

- 输出中的注释以 `#` 为前缀。
- 每个线程都有一个标头，其中包含本机线程 ID：`Thread (<thread-id>):`。
- 堆栈帧遵循格式 `Module!Method`。
- 转换为非托管代码在输出中表示为 `[Native Frames]`。

```
# comment
Thread (0x1234):
  module!Method
  module!Method

Thread (0x5678):
  [Native Frames]
  Module!Method
  Module!Method
```

后续步骤

- 使用 dotnet-trace 收集 .NET 应用程序的 CPU 示例
- 使用 dotnet-dump 收集 .NET 应用程序的转储

符号下载器 (dotnet-symbol)

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

安装

若要安装最新版 `dotnet-symbol` NuGet 包，请使用 dotnet tool install 命令：

```
dotnet tool install --global dotnet-symbol
```

摘要

```
dotnet-symbol [-h|--help] [options] <FILES>
```

描述

`dotnet-symbol` 全局工具下载调试核心转储和小型转储所需的文件（符号、DAC、模块等）。当调试其他计算机上捕获的转储时，这很有用。`dotnet-symbol` 可用于下载分析转储所需的模块和符号。

选项

- `--microsoft-symbol-server`

  添加“http://msdl.microsoft.com/download/symbols”符号服务器路径（默认）。

- `--server-path <symbol server path>`

  将符号服务器添加到服务器路径。

- `authenticated-server-path <pat> <server path>`

  使用个人访问令牌 (PAT) 将经过身份验证的符号服务器添加到服务器路径。

- `--cache-directory <file cache directory>`

  添加缓存目录。

- `--recurse-subdirectories`

  处理所有子目录中的输入文件。

- `--host-only`

  仅下载 lldb 加载核心转储所需的主机程序（即 dotnet）。

- `--symbols`

  下载符号文件（.pdb、.dbg 和 .dwarf）。

- `--modules`

  下载模块文件（.dll、.so 和 .dylib）。

- `--debugging`

  下载特殊的调试模块(DAC、DBI 和 SOS)。

- `--windows-pdbs`

  当可移植的 PDB 也可用时，会强制下载 Windows PDB。

- `-o, --output <output directory>`

  设置输出目录。否则，请在输入文件旁边写入(默认)。

- `-d, --diagnostics`

  启用诊断输出。

- `-h|--help`

  显示命令行帮助。

下载符号

默认情况下，针对转储文件运行 `dotnet-symbol` 将下载调试转储所需的所有模块、符号和 DAC/DBI 文件，包括托管程序集。由于 SOS 现在可以按需下载符号，因此可以使用仅带主机 (dotnet) 和调试模块的 lldb 分析大多数 Linux 核心转储。若要获取使用 lldb 诊断核心转储所需的这些文件，请运行以下内容：

```
dotnet-symbol --host-only --debugging <dump file path>
```

故障排除

- 下载符号时出现“404 未找到”。

  只有通过官方渠道(例如官方网站和 dotnet 安装脚本中的默认源)获得的官方 .NET Core 运行时版本才支持符号下载。下载调试文件时出现 404 错误，这可能表示转储是使用来自其他源的 .NET Core 运行时创建的，例如，从本地源、特定 Linux 发行版或从社区站点(例如 archlinux)构建的转储。在此类情况下，应从这些源或创建转储文件的环境复制调试所需的文件(dotnet、libcoreclr.so 和 libmscordaccore.so)。

另请参阅

- 使用符号进行调试
- 符号和可移植 PDB

SOS 安装程序 (dotnet-sos)

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

安装

可采用两种方法来下载和安装 `dotnet-sos`：

- **dotnet 全局工具**：

  若要安装最新版 `dotnet-sos` NuGet 包，请使用 dotnet tool install 命令：

  ```
  dotnet tool install --global dotnet-sos
  ```

- **直接下载**：

  下载与平台相匹配的工具可执行文件：

(OS)	
Windows	x86 | x64 | arm | arm-x64
macOS	x64
Linux	x64 | arm | arm64 | musl-x64 | musl-arm64

摘要

```
dotnet-sos [-h|--help] [options] [command]]
```

说明

`dotnet-sos` 全局工具将安装 SOS 调试程序扩展。借助此扩展，你可以从本机调试器（如 lldb 和 windbg）检查托管 .NET Core 状态。

> **NOTE**
> 只有 Linux 或 macOS 需要通过 `dotnet-sos` 工具安装 SOS。如果使用的是旧版调试工具，则 Windows 也可能需要使用此工具。Windows 调试程序的最新版本（WinDbg 或 cdb 的版本 10.0.18317.1001 及更高版本）会从 Microsoft 扩展程序库自动加载 SOS。

选项

- `--version`

  显示版本信息。

- `-h|--help`

显示命令行帮助。

安装 dotnet-sos

在本地安装用于调试 .NET Core 进程的 SOS 扩展。在 macOS 和 Linux 上，将更新 .lldbinit 文件，以便扩展在 lldb 启动时自动加载。如果要使用较旧的调试工具（低于版本 10.0.18317.1001）在 Windows 上安装 SOS，则需要通过在调试程序中运行 `.load %USERPROFILE%\.dotnet\sos\sos.dll` 以在 WinDbg 或 cdb 中手动加载扩展。

摘要

```
dotnet-sos install [--architecture <arch>]
```

选项

- `--architecture <arch>`

  指定要安装的 SOS 二进制文件的处理器体系结构。默认情况下，`dotnet-sos` 安装主机的体系结构。当你要为与 dotnet 主机体系结构不同的体系结构安装 SOS 时，请使用此选项。例如，如果要从 Arm64 主机运行 Arm32 二进制文件，则需要使用 `dotnet-sos install --architecture Arm` 安装 SOS。

  可以使用以下体系结构：

  - `Arm`
  - `Arm64`
  - `X86`
  - `X64`

卸载 dotnet-sos

卸载 SOS 扩展名，并在 Linux 和 macOS 上将其从 lldb 配置中删除。

摘要

```
dotnet-sos uninstall
```

使用 PerfCollect 跟踪 .NET 应用程序

2021/11/16 •

本文适用于：✔ .NET Core 2.1 SDK 及更高版本

在 Linux 上遇到性能问题时，可使用 `perfcollect` 收集跟踪，以便收集有关出现性能问题时计算机上发生的状况的详细信息。

`perfcollect` 是一个 bash 脚本，它利用 Linux 跟踪工具包: 下一代 (LTTng) 收集从运行时或任何 EventSource 写入的事件，并利用 perf 收集目标进程的 CPU 示例。

准备计算机

按照以下步骤准备你的计算机以使用 `perfcollect` 收集性能跟踪。

> **NOTE**
> 如果你处于容器环境中，则容器需要具有 `SYS_ADMIN` 功能。有关使用 PerfCollect 跟踪容器内应用程序的详细信息，请参阅在容器中收集诊断信息。

1. 下载 `perfcollect`。

```
curl -OL https://aka.ms/perfcollect
```

2. 使脚本可执行。

```
chmod +x perfcollect
```

3. 安装跟踪必备组件 - 这些是实际的跟踪库。

```
sudo ./perfcollect install
```

   这将在你的计算机上安装以下必备组件：

   a. `perf`：Linux 性能事件子系统和配套的用户模式收集/查看器应用程序。`perf` 是 Linux 内核源的一部分，但是默认情况下通常不安装。

   b. `LTTng`：用于捕获 CoreCLR 在运行时发出的事件数据。然后使用这些数据分析各种运行时组件（如 GC、JIT 和线程池）的行为。

最新版本的 .NET Core 和 Linux 性能工具支持自动解析框架代码的方法名称。如果使用的是 .NET Core 3.1 或更低版本，则需要执行额外的步骤。有关详细信息，请参阅解析框架符号。

若要解析本机运行时 DLL 的方法名称（例如 libcoreclr.so），`perfcollect` 将在转换数据时为其解析符号，但前提是存在这些二进制文件的符号。有关详细信息，请参阅获取本机运行时的符号部分。

收集跟踪

1. 有两个可用的 shell - 一个用于控制跟踪，称为 [Trace]，另一个用于运行应用程序，称为 [App] 。

2. [Trace]：启动收集。

```
sudo ./perfcollect collect sampleTrace
```

   预期输出：

```
Collection started.  Press CTRL+C to stop.
```

3. [App]：使用以下环境变量设置应用程序 shell - 这将启用 CoreCLR 的跟踪配置。

```
export DOTNET_PerfMapEnabled=1
export DOTNET_EnableEventLog=1
```

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_` 。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

4. [App]：运行应用 - 使其运行捕获性能问题所需的时间。确切时间可以是所需的最短时间，只要足以捕获要调查的性能问题发生的时间窗口。

```
dotnet run
```

5. [Trace]：停止收集 - 按 CTRL+C。

```
^C
...STOPPED.

Starting post-processing. This may take some time.

Generating native image symbol files
...SKIPPED
Saving native symbols
...FINISHED
Exporting perf.data file
...FINISHED
Compressing trace files
...FINISHED
Cleaning up artifacts
...FINISHED

Trace saved to sampleTrace.trace.zip
```

   压缩的跟踪文件现存储在当前工作目录中。

查看跟踪

有许多选项可用于查看收集的跟踪。在 Windows 上，最好使用 PerfView 查看跟踪；但在 Linux 上，可以使用 `PerfCollect` 本身或 `TraceCompass` 直接进行查看。

使用 PerfCollect 查看跟踪文件

你可以使用 perfcollect 本身来查看收集的跟踪。为此，请使用以下命令：

```
./perfcollect view sampleTrace.trace.zip
```

默认情况下，这将使用 `perf` 显示应用程序的 CPU 跟踪。

要查看通过 `LTTng` 收集的事件，可以传入标志 `-viewer lttng` 以查看各个事件：

```
./perfcollect view sampleTrace.trace.zip -viewer lttng
```

这将使用 `babeltrace` 查看器打印事件有效负载：

```
# [01:02:18.189217659] (+0.020132603) ubuntu-xenial DotNETRuntime:ExceptionThrown_V1: { cpu_id = 0 }, {
ExceptionType = "System.Exception", ExceptionMessage = "An exception happened", ExceptionEIP =
139875671834775, ExceptionHRESULT = 2148734208, ExceptionFlags = 16, ClrInstanceID = 0 }
# [01:02:18.189250227] (+0.020165171) ubuntu-xenial DotNETRuntime:ExceptionCatchStart: { cpu_id = 0 }, {
EntryEIP = 139873639728404, MethodID = 139873626968120, MethodName = "void [helloworld]
helloworld.Program::Main(string[])", ClrInstanceID = 0 }
```

使用 PerfView 打开跟踪文件

要查看 CPU 示例和事件的聚合视图，可以在 Windows 计算机上使用 `PerfView`。

1. 将 trace.zip 文件从 Linux 复制到 Windows 计算机。
2. 从 https://aka.ms/perfview 下载 PerfView。
3. 运行 PerfView.exe

    ```
    PerfView.exe <path to trace.zip file>
    ```

PerfView 将基于跟踪文件中包含的数据显示受支持的视图列表。

- 对于 CPU 调查，请选择“CPU 堆栈”。
- 有关 GC 的详细信息，请选择“GCStats”。
- 有关每个进程/模块/方法的 JIT 信息，请选择“JITStats”。
- 如果没有所需信息的视图，可以尝试在原始事件视图中查找事件。选择“事件”。

有关如何在 PerfView 中解释视图的详细信息，请参见视图本身的帮助链接，或者从 PerfView 的主窗口中，选择“帮助”->“用户指南”。

> **NOTE**
>
> 通过 System.Diagnostics.Tracing.EventSource API 编写的事件（包括 Framework 中的事件）不会显示在其提供程序名称下。相反，它们被编写为 `Microsoft-Windows-DotNETRuntime` 提供程序下的 `EventSourceEvent` 事件，其有效负载是经过 JSON 序列化的。

使用 TraceCompass 打开跟踪文件

Eclipse TraceCompass 是另一个可用于查看跟踪的选项。`TraceCompass` 也可以在 Linux 计算机上工作，因此不需要将跟踪移到 Windows 计算机上。要使用 `TraceCompass` 打开跟踪文件，需要解压缩该文件。

```
unzip myTrace.trace.zip
```

`perfcollect` 将它收集的 LTTng 跟踪保存为 CTF 文件格式，位于 `lttngTrace` 的子目录中。具体来说，CTF 文件将位于类似于 `lttngTrace/auto-20201025-101230\ust\uid\1000\64-bit\` 的目录中。

可以通过选择 `File -> Open Trace` 打开 `TraceCompass` 中的 CTF 跟踪文件，然后选择 `metadata` 文件。

有关详细信息，请参阅 `TraceCompass` 文档。

解析框架符号

收集跟踪时，需要手动生成框架符号。它们不同于应用级别符号，因为框架是预编译的，而应用代码是即时编译的。对于预编译为本机代码的框架代码，需要调用 `crossgen`，它知道如何生成从本机代码到方法名称的映射。

`perfcollect` 可以处理大部分细节，但需要 `crossgen` 可用。默认情况下，它不随 .NET 分发版一起安装。如果 `crossgen` 不存在，`perfcollect` 会向你发出警告，并让你参考这些说明。要修复这些问题，需要为正在使用的运行时获取正确版本的 crossgen。如果将 crossgen 工具置于 .NET 运行时 DLL 的同一目录中（例如 libcoreclr.so），则 `perfcollect` 可以找到该工具并将框架符号添加到跟踪文件中。

通常，当你创建 .NET 应用程序时，它只为你编写的代码生成 DLL，对其余代码使用运行时的共享副本。但是，你也可以生成应用程序所谓的“自包含”版本，其中包含所有运行时 DLL。`crossgen` 是用于创建自包含应用的 NuGet 包的一部分，因此获取正确版本的 `crossgen` 的一种方法是创建应用程序的自包含包。

例如：

```
mkdir helloWorld
cd helloWorld
dotnet new console
dotnet publish --self-contained -r linux-x64
```

这将创建一个新的 Hello World 应用程序并将其生成为自包含应用。

创建自包含应用程序的副作用是 dotnet 工具会下载名为 runtime.linux-x64.microsoft.netcore.app 的 NuGet 包，并将其置于目录 ~/.nuget/packages/runtime.linux-x64.microsoft.netcore.app/VERSION 中，其中 VERSION 是 .NET Core 运行时的版本号（例如 2.1.0）。在这个目录下是一个工具目录，其中有你需要的 crossgen 工具。从 .NET Core 3.0 开始，包位置为 ~/.nuget/packages/microsoft.netcore.app.runtime.linux-x64/VERSION。

需要将 `crossgen` 工具放在应用程序实际使用的运行时旁边。通常，你的应用程序使用安装在 /usr/share/dotnet/shared/Microsoft.NETCore.App/VERSION 上的 .NET Core 共享版本，其中 VERSION 是 .NET 运行时的版本号。这是一个共享位置，因此你需要成为超级用户才能对其进行修改。如果 VERSION 为 2.1.0，则更新 `crossgen` 的命令为：

```
sudo bash
cp ~/.nuget/packages/runtime.linux-x64.microsoft.netcore.app/2.1.0/tools/crossgen
/usr/share/dotnet/shared/Microsoft.NETCore.App/2.1.0
```

完成此操作后，`perfcollect` 将使用 crossgen 包含框架符号。`perfcollect` 以前发出的警告应会消失。这在每台计算机上只需要执行一次（直到更新运行时为止）。

替代项：禁用预编译代码

如果无法更新 .NET 运行时（以添加 `crossgen`），或者如果上述过程出于某种原因而无效，可以使用另一种方法来获取框架符号。你可以指示运行时不要使用预编译的框架代码。代码将即时编译，不需要 `crossgen`。

> **NOTE**
> 选择此方法可能会增加应用程序的启动时间。

为此，可以添加以下环境变量：

```
export DOTNET_ZapDisable=1
```

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

通过此更改，你应该会获得所有 .NET 代码的符号。

获取本机运行时的符号

大多数情况下，你感兴趣的是自己的代码，`perfcollect` 默认解析这些代码。有时查看 .NET DLL 内部的情况很有用（这是上一节讨论的内容），但有时查看本机运行时 dll 中的情况（通常为 libcoreclr.so）也很有趣。`perfcollect` 在转换其数据时将解析这些符号，但前提是存在这些本机 DLL 的符号（并且位于它们所对应的库的旁边）。

有一个名为 dotnet-symbol 的全局命令可以执行此操作。使用 dotnet-symbol 获取本机运行时符号：

1. 安装 `dotnet-symbol`：

   ```
   dotnet tool install -g dotnet-symbol
   ```

2. 下载符号。如果你安装的 .NET Core 运行时版本是 2.1.0，则执行此操作的命令是：

   ```
   mkdir mySymbols
   dotnet symbol --symbols --output mySymbols
   /usr/share/dotnet/shared/Microsoft.NETCore.App/2.1.0/lib*.so
   ```

3. 将符号复制到正确的位置。

   ```
   sudo cp mySymbols/* /usr/share/dotnet/shared/Microsoft.NETCore.App/2.1.0
   ```

   如果由于没有对相应目录的写入访问权限而无法完成此操作，可以使用 `perf buildid-cache` 添加符号。

此后，当你运行 `perfcollect` 时，应获取本机 dll 的符号名称。

在 Docker 容器中收集信息

有关如何在容器环境中使用 `perfcollect` 的详细信息，请参阅在容器中收集诊断信息。

了解有关集合选项的详细信息

你可以使用 `perfcollect` 指定以下可选标志，以更好地满足诊断需求。

在特定的时间内收集

如果要收集特定时间内的跟踪，可以使用 `-collectsec` 选项后跟一个数字，该数字指定收集跟踪的总秒数。

收集线程时间跟踪

使用 `perfcollect` 指定 `-threadtime` 可让你收集每个线程的 CPU 使用率数据。从而分析每个线程将 CPU 时间用在何处。

收集托管内存和垃圾回收器性能的跟踪

以下选项可让你专门收集运行时中的 GC 事件。

- `perfcollect collect -gccollectonly`

仅收集一组最少的 GC 收集事件。这是最不详细的 GC 事件收集配置文件，对目标应用性能的影响最小。此命令类似于 PerfView 中的 `PerfView.exe /GCCollectOnly collect` 命令。

- `perfcollect collect -gconly`

收集更详细的 GC 收集事件，包括 JIT、加载程序和异常事件。这会请求更详细的事件（例如分配信息和 GC 联接信息），对目标应用性能产生的影响比 `-gccollectonly` 选项产生的影响更大。此命令类似于 PerfView 中的 `PerfView.exe /GCOnly collect` 命令。

- `perfcollect collect -gcwithheap`

收集最详细的 GC 收集事件（用于跟踪堆的存活和移动情况）。这会对 GC 行为进行深入分析，但会对性能产生较大的影响，因为每个 GC 都可能需要两倍的时间。建议在生产环境中进行跟踪时，了解使用此跟踪选项的性能影响。

调试 .NET Core 中的内存泄漏

2021/11/16 •

本文适用于： ✔ .NET Core 3.1 SDK 及更高版本

当应用引用不再需要执行所需任务的对象时，可能会发生内存泄漏。引用上述对象会使垃圾回收器无法回收所使用的内存，这通常会导致性能降低，并可能最终引发 OutOfMemoryException。

本教程演示如何使用 .NET 诊断 CLI 工具分析 .NET Core 应用中的内存泄漏。如果所在的操作系统是 Windows，则可以使用 Visual Studio 的内存诊断工具调试内存泄漏。

本教程使用一个示例应用程序，它设计为有意泄漏内存。本示例作为练习提供。还可以分析无意中泄漏内存的应用程序。

在本教程中，你将：

- 使用 dotnet-counters 检查托管内存的使用情况。
- 生成转储文件。
- 使用转储文件分析内存使用情况。

先决条件

本教程使用：

- .NET Core 3.1 SDK 或更高版本。
- dotnet-counters 检查托管内存的使用情况。
- dotnet-dump 收集和分析转储文件。
- 要诊断的示例调试目标应用。

本教程假设已安装示例和工具并可供使用。

检查托管内存的使用情况

在开始收集诊断数据以帮助分析本案例的根本原因时，需要确保实际看到的是内存泄漏（内存增加）。可以使用 dotnet-counters 工具进行确认。

打开控制台窗口并导航到下载并解压缩示例调试目标的目录。运行目标：

```
dotnet run
```

在单独的控制台中，找到处理 ID：

```
dotnet-counters ps
```

输出应如下所示：

```
4807 DiagnosticScena
/home/user/git/samples/core/diagnostics/DiagnosticScenarios/bin/Debug/netcoreapp3.0/DiagnosticScenarios
```

现使用 dotnet-counters 工具检查托管内存的使用情况。`--refresh-interval` 指定两次刷新之间的秒数：

```
dotnet-counters monitor --refresh-interval 1 -p 4807
```

实时输出应如下所示：

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    # of Assemblies Loaded                          118
    % Time in GC (since last GC)                      0
    Allocation Rate (Bytes / sec)                37,896
    CPU Usage (%)                                     0
    Exceptions / sec                                  0
    GC Heap Size (MB)                                 4
    Gen 0 GC / sec                                    0
    Gen 0 Size (B)                                    0
    Gen 1 GC / sec                                    0
    Gen 1 Size (B)                                    0
    Gen 2 GC / sec                                    0
    Gen 2 Size (B)                                    0
    LOH Size (B)                                      0
    Monitor Lock Contention Count / sec               0
    Number of Active Timers                           1
    ThreadPool Completed Work Items / sec            10
    ThreadPool Queue Length                           0
    ThreadPool Threads Count                          1
    Working Set (MB)                                 83
```

重点介绍此行：

```
    GC Heap Size (MB)                                 4
```

启动后，可以看到托管堆内存为 4 MB。

现在，点击 URL `https://localhost:5001/api/diagscenario/memleak/20000`。

请注意，内存使用量已增加到 30 MB。

```
    GC Heap Size (MB)                                30
```

通过监视内存使用情况，可以确定内存正在增长或泄漏。下一步是收集内存分析的适当数据。

生成内存转储

分析可能的内存泄漏时，需要访问应用的内存堆。然后可以分析内存内容。查看对象之间的关系，可以创建理论说明内存未释放的原因。常见的诊断数据源是 Windows 上的内存转储或 Linux 上的等效核心转储。若要生成 .NET Core 应用程序转储，可使用 dotnet-dump 工具。

使用之前启动的示例调试目标，运行以下命令以生成 Linux 核心转储：

```
dotnet-dump collect -p 4807
```

结果是位于同一文件夹中的核心转储。

```
Writing minidump with heap to ./core_20190430_185145
Complete
```

重新启动失败的进程

收集转储后，你应该有足够的信息来诊断失败的进程。如果失败的进程在生产服务器上运行，现在是通过重新启动进程进行短期修正的理想时机。

在本教程中，你已经完成了示例调试目标，现在可以将其关闭。导航到启动服务器的终端并按 Ctrl+C。

分析核心转储

生成核心转储后，请使用 dotnet-dump 工具分析转储：

```
dotnet-dump analyze core_20190430_185145
```

其中 `core_20190430_185145` 是要分析的核心转储的名称。

> **NOTE**
> 如果你看到报错"找不到 libdl.so "，则可能需要安装 libc6-dev 包。有关详细信息，请参阅 Linux 上 .NET Core 的先决条件。

此时会显示一个提示，可在其中输入 SOS 命令。通常，首先要查看的是托管堆的整体状态：

```
> dumpheap -stat

Statistics:
              MT    Count    TotalSize Class Name
...
00007f6c1eeefba8      576        59904 System.Reflection.RuntimeMethodInfo
00007f6c1dc021c8     1749        95696 System.SByte[]
00000000008c9db0     3847       116080      Free
00007f6c1e784a18      175       128640 System.Char[]
00007f6c1dbf5510      217       133504 System.Object[]
00007f6c1dc014c0      467       416464 System.Byte[]
00007f6c21625038        6      4063376 testwebapi.Controllers.Customer[]
00007f6c20a67498   200000      4800000 testwebapi.Controllers.Customer
00007f6c1dc00f90   206770     19494060 System.String
Total 428516 objects
```

可在此处看到大多数对象是 `String` 或 `Customer` 对象。

可以使用方法表 (MT) 再次使用 `dumpheap` 命令来获取所有 `String` 实例的列表：

```
> dumpheap -mt 00007faddaa50f90

         Address               MT     Size
...
00007f6ad09421f8 00007faddaa50f90       94
...
00007f6ad0965b20 00007f6c1dc00f90       80
00007f6ad0965c10 00007f6c1dc00f90       80
00007f6ad0965d00 00007f6c1dc00f90       80
00007f6ad0965df0 00007f6c1dc00f90       80
00007f6ad0965ee0 00007f6c1dc00f90       80

Statistics:
              MT    Count    TotalSize Class Name
00007f6c1dc00f90   206770     19494060 System.String
Total 206770 objects
```

现在可以对 `System.String` 实例使用 `gcroot` 命令，以查看对象的根方式和原因。请耐心等待，因为对于 30 MB 的堆，此命令需要几分钟的时间：

```
> gcroot -all 00007f6ad09421f8

Thread 3f68:
    00007F6795BB58A0 00007F6C1D7D0745 System.Diagnostics.Tracing.CounterGroup.PollForValues()
[/_/src/System.Private.CoreLib/shared/System/Diagnostics/Tracing/CounterGroup.cs @ 260]
        rbx:  (interior)
            ->  00007F6BDFFFF038 System.Object[]
            ->  00007F69D0033570 testwebapi.Controllers.Processor
            ->  00007F69D0033588 testwebapi.Controllers.CustomerCache
            ->  00007F69D00335A0 System.Collections.Generic.List`1[[testwebapi.Controllers.Customer,
DiagnosticScenarios]]
            ->  00007F6C000148A0 testwebapi.Controllers.Customer[]
            ->  00007F6AD0942258 testwebapi.Controllers.Customer
            ->  00007F6AD09421F8 System.String

HandleTable:
    00007F6C98BB15F8 (pinned handle)
    -> 00007F6BDFFFF038 System.Object[]
    -> 00007F69D0033570 testwebapi.Controllers.Processor
    -> 00007F69D0033588 testwebapi.Controllers.CustomerCache
    -> 00007F69D00335A0 System.Collections.Generic.List`1[[testwebapi.Controllers.Customer,
DiagnosticScenarios]]
    -> 00007F6C000148A0 testwebapi.Controllers.Customer[]
    -> 00007F6AD0942258 testwebapi.Controllers.Customer
    -> 00007F6AD09421F8 System.String

Found 2 roots.
```

可以看到 `String` 由 `Customer` 对象直接保存，并由 `CustomerCache` 对象间接保存。

可以继续转储对象，以查看大多数 `String` 对象是否遵循类似的模式。此时，调查会提供足够的信息来确定代码中的根本原因。

可通过此常规过程确定主要内存泄漏源。

清理资源

在本教程中，你已启动一个示例 Web 服务器。此服务器应已关闭，如重新启动失败的进程部分所述。

还可以删除已创建的转储文件。

请参阅

- 用于列出进程的 dotnet-trace
- 用于检查托管内存使用情况的 dotnet-counters
- 用于收集和分析转储文件的 dotnet-dump
- dotnet/diagnostics
- 使用 Visual Studio 调试内存泄漏

后续步骤

调试 .NET Core 中的高 CPU

调试 .NET Core 中的高 CPU 使用率

2021/11/16 •

本文适用于： ✔ .NET Core 3.1 SDK 及更高版本

本教程将介绍如何调试 CPU 使用率过高的情况。使用提供的示例 ASP.NET Core Web 应用 源代码存储库，可以故意造成死锁。终结点将停止响应并遇到线程累积问题。你将了解如何使用各种工具，通过几条关键的诊断数据诊断此情况。

在本教程中，你将：

- 调查 CPU 使用率是否过高
- 使用 dotnet-counters 确定 CPU 使用率
- 使用 dotnet-trace 进行跟踪生成
- PerfView 中的配置文件性能
- 诊断并解决 CPU 使用率过高的问题

先决条件

本教程使用：

- .NET Core 3.1 SDK 或更高版本。
- 示例调试目标以触发场景。
- dotnet-trace 以列出进程并生成配置文件。
- dotnet-counters 以监视 CPU 使用率。

CPU 计数器

在尝试收集诊断数据之前，需要观察 CPU 状况是否过高。使用以下命令从项目根目录运行示例应用程序。

```
dotnet run
```

若要查找该进程 ID，请使用以下命令：

```
dotnet-trace ps
```

注意命令输出中的进程 ID。我们的进程 ID 是 `22884`，你的进程 ID 将不同。若要检查当前的 CPU 使用率，请使用 dotnet counters 工具命令：

```
dotnet-counters monitor --refresh-interval 1 -p 22884
```

`refresh-interval` 是计数器轮询 CPU 值间隔的秒数。输出应与以下内容类似：

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    % Time in GC since last GC (%)                      0
    Allocation Rate / 1 sec (B)                         0
    CPU Usage (%)                                       0
    Exception Count / 1 sec                             0
    GC Heap Size (MB)                                   4
    Gen 0 GC Count / 60 sec                             0
    Gen 0 Size (B)                                      0
    Gen 1 GC Count / 60 sec                             0
    Gen 1 Size (B)                                      0
    Gen 2 GC Count / 60 sec                             0
    Gen 2 Size (B)                                      0
    LOH Size (B)                                        0
    Monitor Lock Contention Count / 1 sec               0
    Number of Active Timers                             1
    Number of Assemblies Loaded                       140
    ThreadPool Completed Work Item Count / 1 sec        3
    ThreadPool Queue Length                             0
    ThreadPool Thread Count                             7
    Working Set (MB)                                   63
```

在 Web 应用运行的情况下，CPU 根本不会在启动后就立即被消耗，且会在 `0%` 进行报告。使用 `60000` 作为路由参数导航到 `api/diagscenario/highcpu` 路由：

`https://localhost:5001/api/diagscenario/highcpu/60000`

现在，重新运行 dotnet-counters 命令。若要只监视 `cpu-usage`，请在命令中指定 `System.Runtime[cpu-usage]`。

```
dotnet-counters monitor --counters System.Runtime[cpu-usage] -p 22884 --refresh-interval 1
```

你将看到 CPU 使用率已增加，如下所示：

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    CPU Usage (%)                                      25
```

在整个请求期间，CPU 使用率将徘徊在 25% 左右。根据主机的不同，预期 CPU 使用率会有所不同。

> **TIP**
> 若要可视化更高的 CPU 使用率，可以在多个浏览器选项卡中同时使用此终结点。

此时，你可以放心地说 CPU 运行的速度比预期的要高。

跟踪生成

当分析速度较慢的请求时，需要一个诊断工具来提供代码正在执行的操作的见解。常见的选择是探查器，并且有不同的探查器选项可供选择。

- Linux
- Windows

`perf` 工具可用于生成 .NET Core 应用配置文件。退出示例调试目标的上一个实例。

设置 `DOTNET_PerfMapEnabled` 环境变量，使 .NET Core 应用在 `/tmp` 目录中创建 `map` 文件。`perf` 使用此 `map` 文件按名称将 CPU 地址映射到 JIT 生成的函数。有关详细信息，请参阅写入 Perf 映射。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

在同一终端会话中运行示例调试目标。

```
export DOTNET_PerfMapEnabled=1
dotnet run
```

再次使用高 CPU API (`https://localhost:5001/api/diagscenario/highcpu/60000`) 终结点。当它在 1 分钟请求内运行时，对进程 ID 运行 `perf` 命令：

```
sudo perf record -p 2266 -g
```

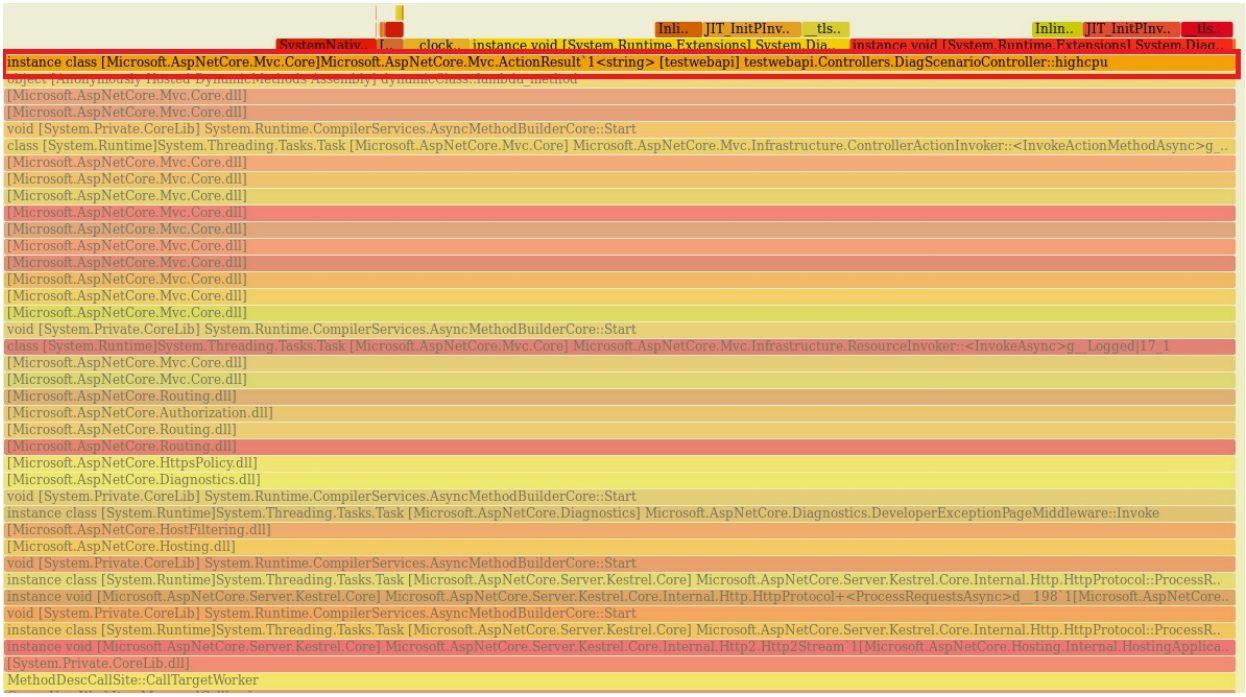
`perf` 命令将启动性能收集过程。让它运行大约 20-30 秒，然后按 `Ctrl+C` 退出收集过程。可以使用相同的 `perf` 命令来查看跟踪的输出。

```
sudo perf report -f
```

还可以使用以下命令生成 flame-graph：

```
git clone --depth=1 https://github.com/BrendanGregg/FlameGraph
sudo perf script | FlameGraph/stackcollapse-perf.pl | FlameGraph/flamegraph.pl > flamegraph.svg
```

此命令生成 `flamegraph.svg`，你可以在浏览器中查看 `flamegraph.svg` 以调查性能问题：



请参阅

- 用于列出进程的 dotnet-trace

- 用于检查托管内存使用情况的 dotnet-counters
- 用于收集和分析转储文件的 dotnet-dump
- dotnet/diagnostics

后续步骤

调试 .NET Core 中的死锁

调试 .NET Core 中的死锁

2021/11/16 •

本文适用于： ✔ .NET Core 3.1 SDK 及更高版本

本教程将介绍如何调试死锁情况。使用提供的示例 ASP.NET Core Web 应用 源代码存储库，可以故意造成死锁。终结点将停止响应并遇到线程累积问题。你将了解如何使用各种工具来分析问题，例如核心转储、核心转储分析和进程跟踪。

在本教程中，你将：

- 调查已停止响应的应用
- 生成核心转储文件
- 分析转储文件中的进程线程
- 分析调用堆栈和同步块
- 诊断并解决死锁

先决条件

本教程使用：

- .NET Core 3.1 SDK 或更高版本
- 用于触发场景的示例调试目标 - Web 应用
- 用于列出进程的 dotnet-trace
- 收集和分析转储文件的 dotnet-dump

核心转储生成

为了调查应用程序无响应问题，核心转储或内存转储允许你检查其线程的状态以及任何可能存在争用问题的锁定状态。使用以下命令从示例根目录运行示例调试应用程序：

```
dotnet run
```

若要查找进程 ID，请使用以下命令：

```
dotnet-trace ps
```

注意命令输出中的进程 ID。我们的进程 ID 是 `4807`，你的进程 ID 将不同。导航到以下 URL，该 URL 是示例站点上的 API 终结点：

`https://localhost:5001/api/diagscenario/deadlock`

向站点发出的 API 请求将停止响应。让请求运行大约 10-15 秒。然后使用以下命令创建核心转储：

- Linux
- Windows

```
sudo dotnet-dump collect -p 4807
```

分析核心转储

若要启动核心转储分析，请使用以下 `dotnet-dump analyze` 命令打开核心转储。参数是先前收集的核心转储文件的路径。

```
dotnet-dump analyze  ~/.dotnet/tools/core_20190513_143916
```

由于要查看可能无响应的应用程序，因此需要对进程中的线程活动有一个总体了解。可以按如下所示使用 `threads` 命令：

```
> threads
*0 0x1DBFF (121855)
 1 0x1DC01 (121857)
 2 0x1DC02 (121858)
 3 0x1DC03 (121859)
 4 0x1DC04 (121860)
 5 0x1DC05 (121861)
 6 0x1DC06 (121862)
 7 0x1DC07 (121863)
 8 0x1DC08 (121864)
 9 0x1DC09 (121865)
 10 0x1DC0A (121866)
 11 0x1DC0D (121869)
 12 0x1DC0E (121870)
 13 0x1DC10 (121872)
 14 0x1DC11 (121873)
 15 0x1DC12 (121874)
 16 0x1DC13 (121875)
 17 0x1DC14 (121876)
 18 0x1DC15 (121877)
 19 0x1DC1C (121884)
 20 0x1DC1D (121885)
 21 0x1DC1E (121886)
 22 0x1DC21 (121889)
 23 0x1DC22 (121890)
 24 0x1DC23 (121891)
 25 0x1DC24 (121892)
 26 0x1DC25 (121893)
...
...
 317 0x1DD48 (122184)
 318 0x1DD49 (122185)
 319 0x1DD4A (122186)
 320 0x1DD4B (122187)
 321 0x1DD4C (122188)
```

该输出显示进程中当前运行的所有线程及其关联的调试器线程 ID 和操作系统线程 ID。基于输出，有超过 300 个线程。

下一步是通过获取每个线程的调用堆栈来更好地了解线程当前正在执行的操作。`clrstack` 命令可用于输出调用堆栈。它既可以输出单个调用堆栈，也可以输出所有调用堆栈。使用以下命令输出进程中所有线程的所有调用堆栈：

```
clrstack -all
```

输出的典型部分如下所示：

```
  ...
  ...
  ...
```

```
        Child SP               IP Call Site
00007F2AE37B5680 00007f305abc6360 [GCFrame: 00007f2ae37b5680]
00007F2AE37B5770 00007f305abc6360 [GCFrame: 00007f2ae37b5770]
00007F2AE37B57D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae37b57d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE37B5920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE37B5950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE37B5CA0 00007f30593044af [GCFrame: 00007f2ae37b5ca0]
00007F2AE37B5D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae37b5d70]
OS Thread Id: 0x1dc82
        Child SP               IP Call Site
00007F2AE2FB4680 00007f305abc6360 [GCFrame: 00007f2ae2fb4680]
00007F2AE2FB4770 00007f305abc6360 [GCFrame: 00007f2ae2fb4770]
00007F2AE2FB47D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae2fb47d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE2FB4920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE2FB4950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE2FB4CA0 00007f30593044af [GCFrame: 00007f2ae2fb4ca0]
00007F2AE2FB4D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae2fb4d70]
OS Thread Id: 0x1dc83
        Child SP               IP Call Site
00007F2AE27B3680 00007f305abc6360 [GCFrame: 00007f2ae27b3680]
00007F2AE27B3770 00007f305abc6360 [GCFrame: 00007f2ae27b3770]
00007F2AE27B37D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae27b37d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE27B3920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE27B3950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE27B3CA0 00007f30593044af [GCFrame: 00007f2ae27b3ca0]
00007F2AE27B3D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae27b3d70]
OS Thread Id: 0x1dc84
        Child SP               IP Call Site
00007F2AE1FB2680 00007f305abc6360 [GCFrame: 00007f2ae1fb2680]
00007F2AE1FB2770 00007f305abc6360 [GCFrame: 00007f2ae1fb2770]
00007F2AE1FB27D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae1fb27d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE1FB2920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE1FB2950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE1FB2CA0 00007f30593044af [GCFrame: 00007f2ae1fb2ca0]
00007F2AE1FB2D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae1fb2d70]
OS Thread Id: 0x1dc85
        Child SP               IP Call Site
00007F2AE17B1680 00007f305abc6360 [GCFrame: 00007f2ae17b1680]
00007F2AE17B1770 00007f305abc6360 [GCFrame: 00007f2ae17b1770]
00007F2AE17B17D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae17b17d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE17B1920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE17B1950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE17B1CA0 00007f30593044af [GCFrame: 00007f2ae17b1ca0]
00007F2AE17B1D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae17b1d70]
```

```
OS Thread Id: 0x1dc86
        Child SP               IP Call Site
00007F2AE0FB0680 00007f305abc6360 [GCFrame: 00007f2ae0fb0680]
00007F2AE0FB0770 00007f305abc6360 [GCFrame: 00007f2ae0fb0770]
00007F2AE0FB07D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae0fb07d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE0FB0920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE0FB0950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE0FB0CA0 00007f30593044af [GCFrame: 00007f2ae0fb0ca0]
00007F2AE0FB0D70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae0fb0d70]
OS Thread Id: 0x1dc87
        Child SP               IP Call Site
00007F2AE07AF680 00007f305abc6360 [GCFrame: 00007f2ae07af680]
00007F2AE07AF770 00007f305abc6360 [GCFrame: 00007f2ae07af770]
00007F2AE07AF7D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2ae07af7d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2AE07AF920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2AE07AF950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2AE07AFCA0 00007f30593044af [GCFrame: 00007f2ae07afca0]
00007F2AE07AFD70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2ae07afd70]
OS Thread Id: 0x1dc88
        Child SP               IP Call Site
00007F2ADFFAE680 00007f305abc6360 [GCFrame: 00007f2adffae680]
00007F2ADFFAE770 00007f305abc6360 [GCFrame: 00007f2adffae770]
00007F2ADFFAE7D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2adffae7d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2ADFFAE920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2ADFFAE950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2ADFFAECA0 00007f30593044af [GCFrame: 00007f2adffaeca0]
00007F2ADFFAED70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2adffaed70]
...
...
```

观察所有 300 多个线程的调用堆栈可以发现一个模式，其中大多数线程共享一个公共调用堆栈：

```
OS Thread Id: 0x1dc88
        Child SP               IP Call Site
00007F2ADFFAE680 00007f305abc6360 [GCFrame: 00007f2adffae680]
00007F2ADFFAE770 00007f305abc6360 [GCFrame: 00007f2adffae770]
00007F2ADFFAE7D0 00007f305abc6360 [HelperMethodFrame_1OBJ: 00007f2adffae7d0]
System.Threading.Monitor.ReliableEnter(System.Object, Boolean ByRef)
00007F2ADFFAE920 00007F2FE392B31F testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_1()
[/home/marioh/webapi/Controllers/diagscenario.cs @ 36]
00007F2ADFFAE950 00007F2FE392B46D
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/__w/3/s/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
00007F2ADFFAECA0 00007f30593044af [GCFrame: 00007f2adffaeca0]
00007F2ADFFAED70 00007f30593044af [DebuggerU2MCatchHandlerFrame: 00007f2adffaed70]
```

该调用堆栈似乎显示请求传入了死锁方法，而死锁方法继而又调用了 `Monitor.ReliableEnter` 。此方法表示这些线程正试图进入监视器锁定。它们正在等待该锁定的可用性。它可能已被其他线程锁定。

下一步是找出实际持有监视器锁定的线程。由于监视器通常将锁定信息存储在同步块表中，因此我们可以使用 `syncblk` 命令来获取更多信息：

```
> syncblk
Index         SyncBlock MonitorHeld Recursion Owning Thread Info          SyncBlock Owner
   43 00000246E51268B8          603         1 0000024B713F4E30 5634  28   00000249654b14c0 System.Object
   44 00000246E5126908            3         1 0000024B713F47E0 51d4  29   00000249654b14d8 System.Object
-----------------------------
Total           344
CCW             1
RCW             2
ComClassFactory 0
Free            0
```

两个有趣的列是“MonitorHeld”和“Owning Thread Info” 。“MonitorHeld”列显示线程是否获取了监视器锁定以及正在等待的线程的数量。“Owning Thread Info”列显示当前拥有监视器锁定的线程。线程信息有三个不同的子列。第二个子列显示操作系统线程 ID。

此时，我们知道两个不同的线程(0x5634 和 0x51d4)持有监视器锁定。下一步是查看这些线程正在执行的操作。我们需要检查它们是否无限期陷入持有锁定。让我们使用 `setthread` 和 `clrstack` 命令切换到每个线程并显示调用堆栈。

若要查看第一个线程，请运行 `setthread` 命令，并找到 0x5634 线程的索引(我们的索引是 28)。死锁函数正在等待获取某个锁定，但它已拥有该锁定。该函数处于正在等待它已经持有的锁定的死锁状态。

```
> setthread 28
> clrstack
OS Thread Id: 0x5634 (28)
        Child SP               IP Call Site
0000004E46AFEAA8 00007fff43a5cbc4 [GCFrame: 0000004e46afeaa8]
0000004E46AFEC18 00007fff43a5cbc4 [GCFrame: 0000004e46afec18]
0000004E46AFEC68 00007fff43a5cbc4 [HelperMethodFrame_1OBJ: 0000004e46afec68]
System.Threading.Monitor.Enter(System.Object)
0000004E46AFEDC0 00007FFE5EAF9C80 testwebapi.Controllers.DiagScenarioController.DeadlockFunc()
[C:\Users\dapine\Downloads\Diagnostic_scenarios_sample_debug_target\Controllers\DiagnosticScenarios.cs @ 58]
0000004E46AFEE30 00007FFE5EAF9B8C testwebapi.Controllers.DiagScenarioController.<deadlock>b__3_0()
[C:\Users\dapine\Downloads\Diagnostic_scenarios_sample_debug_target\Controllers\DiagnosticScenarios.cs @ 26]
0000004E46AFEE80 00007FFEBB251A5B System.Threading.ThreadHelper.ThreadStart_Context(System.Object)
[/_/src/System.Private.CoreLib/src/System/Threading/Thread.CoreCLR.cs @ 44]
0000004E46AFEEB0 00007FFE5EAEEEEC
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
[/_/src/System.Private.CoreLib/shared/System/Threading/ExecutionContext.cs @ 201]
0000004E46AFEF20 00007FFEBB234EAB System.Threading.ThreadHelper.ThreadStart()
[/_/src/System.Private.CoreLib/src/System/Threading/Thread.CoreCLR.cs @ 93]
0000004E46AFF138 00007ffebdcc6b63 [GCFrame: 0000004e46aff138]
0000004E46AFF3A0 00007ffebdcc6b63 [DebuggerU2MCatchHandlerFrame: 0000004e46aff3a0]
```

第二个线程类似。它还尝试获取已拥有的锁定。其余 300 多个正在等待的线程很可能也在等待导致死锁的锁定之一。

请参阅

- 用于列出进程的 dotnet-trace
- 用于检查托管内存使用情况的 dotnet-counters
- 用于收集和分析转储文件的 dotnet-dump
- dotnet/diagnostics

后续步骤

.NET Core 中提供哪些诊断工具

调试 StackOverflow 错误

2021/11/16 •

当执行堆栈溢出时，会引发 StackOverflowException，因为它包含太多的嵌套方法调用。

例如，假设你有一款应用，如下所示：

```
using System;

namespace temp
{
    class Program
    {
        static void Main(string[] args)
        {
            Main(args); // Oops, this recursion won't stop.
        }
    }
}
```

`Main` 方法将持续调用自身，直到没有更多堆栈空间为止。没有更多堆栈空间后，执行将无法继续，于是将引发 StackOverflowException。

```
> dotnet run
Stack overflow.
```

> **NOTE**
> 在 .NET 5 及更高版本上，调用栈将输出到控制台。

> **NOTE**
> 本文介绍如何使用 lldb 调试堆栈溢出。如果在 Windows 上运行，建议使用 Visual Studio 或 Visual Studio Code 调试应用。

示例

1. 运行配置为在发生故障时收集转储的应用

```
> export DOTNET_DbgEnableMiniDump=1
> dotnet run
Stack overflow.
Writing minidump with heap to file /tmp/coredump.6412
Written 58191872 bytes (14207 pages) to core file
```

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

2. 使用 dotnet-sos 安装 SOS 扩展

```
dotnet-sos install
```

3. 在 lldb 中调试转储以查看失败的堆栈

```
lldb --core /temp/coredump.6412
(lldb) bt
...
    frame #261930: 0x00007f59b40900cc
    frame #261931: 0x00007f59b40900cc
    frame #261932: 0x00007f59b40900cc
    frame #261933: 0x00007f59b40900cc
    frame #261934: 0x00007f59b40900cc
    frame #261935: 0x00007f5a2d4a080f libcoreclr.so`CallDescrWorkerInternal at
unixasmmacrosamd64.inc:867
    frame #261936: 0x00007f5a2d3cc4c3 libcoreclr.so`MethodDescCallSite::CallTargetWorker(unsigned
long const*, unsigned long*, int) at callhelpers.cpp:70
    frame #261937: 0x00007f5a2d3cc468 libcoreclr.so`MethodDescCallSite::CallTargetWorker(this=
<unavailable>, pArguments=0x00007ffe8222e7b0, pReturnValue=0x0000000000000000, cbReturnValue=0) at
callhelpers.cpp:604
    frame #261938: 0x00007f5a2d4b6182 libcoreclr.so`RunMain(MethodDesc*, short, int*, PtrArray**)
[inlined] MethodDescCallSite::Call(this=<unavailable>, pArguments=<unavailable>) at callhelpers.h:468
...
```

4. 顶部框架 `0x00007f59b40900cc` 重复了几次。使用 SOS `ip2md` 命令找出 `0x00007f59b40900cc` 地址处的方法

```
(lldb) ip2md 0x00007f59b40900cc
MethodDesc:   00007f59b413ffa8
Method Name:          temp.Program.Main(System.String[])
Class:                00007f59b4181d40
MethodTable:          00007f59b4190020
mdToken:              0000000006000001
Module:               00007f59b413dbf8
IsJitted:             yes
Current CodeAddr:     00007f59b40900a0
Version History:
  ILCodeVersion:      0000000000000000
  ReJIT ID:           0
  IL Addr:            0000000000000000
     CodeAddr:           00007f59b40900a0  (MinOptJitted)
     NativeCodeVersion:  0000000000000000
Source file:  /temp/Program.cs @ 9
```

5. 查看指定方法 temp.Program.Main(System.String[]) 和源代码“/temp/Program.cs @ 9”，了解能否找出错误。如果仍不清楚，则可以在该代码区域中添加日志记录。

另请参阅

- .NET 中的转储简介
- 调试 Linux 转储
- 适用于 .NET 的 SOS 调试扩展

.NET 源代码分析概述

2021/11/16 •

.NET Compiler Platform (Roslyn) 分析器会检查 C# 或 Visual Basic 代码的代码质量和样式问题。从 .NET 5 开始，这些分析器包含在 .NET SDK 中，无需单独安装。如果项目面向 .NET 5 或更高版本，则默认启用代码分析。如果项目面向不同的 .NET 实现（例如 .NET Core、.NET Standard 或 .NET Framework），则必须通过将 EnableNETAnalyzers 属性设置为 `true` 以手动启用代码分析。

如果你不想移动到 .NET 5+ SDK、具有非 SDK 样式的 .NET Framework 项目或更倾向于使用基于 NuGet 包的模型，则也可以在 Microsoft.CodeAnalysis.NetAnalyzers NuGet 包中使用该分析器。对于按需版本更新，你可能更倾向于使用基于包的模型。

> **NOTE**
> .NET 分析器与目标框架无关。即，你的项目不需要面向特定的 .NET 实现。分析器适用于面向 `net5.0` 及早期 .NET 版本（如 `netcoreapp3.1` 和 `net472`）的项目。但是，若要使用 EnableNETAnalyzers 属性启用代码分析，则项目必须引用项目 SDK。

如果分析器发现规则冲突，则这些冲突会被报告为建议、警告或错误，具体取决于每个规则的配置方式。代码分析冲突以前缀“CA”或“IDE”显示，以便将它们与编译器错误区分开来。

代码质量分析

代码质量分析（“CAxxxx”）规则检查 C# 或 Visual Basic 代码的安全性、性能、设计及其他问题。分析功能针对面向 .NET 5 或更高版本的项目默认启用。可通过将 EnableNETAnalyzers 属性设置为 `true`，在面向 .NET 早期版本的项目上启用代码分析。你也可通过将 `EnableNETAnalyzers` 设置为 `false`，对项目禁用代码分析。

> **TIP**
> 如果使用的是 Visual Studio，则许多分析器规则都有相关的代码修补程序，可以应用它们来纠正问题。代码修补程序显示在灯泡图标菜单中。

已启用的规则

在 .NET 5 中，以下规则默认启用。

ID			
CA1416	互操作性	警告	平台兼容性分析器
CA1417	互操作性	警告	请勿对 P/Invokes 的字符串参数使用 `OutAttribute`
CA1831	性能	警告	在合适的情况下，对字符串使用 `AsSpan` 而不是基于范围的索引器
CA2013	可靠性	警告	请勿将 `ReferenceEquals` 与值类型结合使用

ID			
CA2014	可靠性	警告	请勿在循环中使用 `stackalloc`
CA2015	可靠性	警告	请勿为派生自 MemoryManager<T> 的类型定义终结器
CA2200	使用情况	警告	再次引发以保留堆栈详细信息
CA2247	使用情况	警告	传递到 TaskCompletionSource 构造函数的参数应为 TaskCreationOptions 枚举，而不是 TaskContinuationOptions

可更改这些规则的严重性，以禁用这些规则或将它们提升为错误。也可启用更多规则。

- 有关每个 .NET SDK 版本附带的规则的列表，请参阅分析器版本。
- 有关所有代码质量规则的列表，请参阅代码质量规则。

启用其他规则

分析模式指预定义的代码分析配置，在此配置下，未启用任何规则、启用某些规则或启用所有规则。在默认分析模式下，只有少量规则作为生成警告启用。可通过在项目文件中设置 <AnalysisMode> 属性来更改项目的分析模式。允许的值为：

`AllDisabledByDefault`	这是最保守的模式。默认情况下，禁用所有规则。可以选择选择加入各条规则，以启用它们。 `<AnalysisMode>AllDisabledByDefault</AnalysisMode>`
`AllEnabledByDefault`	这是最激进的模式。所有规则作为生成警告启用。可选择性地选择退出各条规则以禁用它们。 `<AnalysisMode>AllEnabledByDefault</AnalysisMode>`
`Default`	在默认模式下，少量规则作为警告启用、其他规则仅作为带有相应代码修补程序的 Visual Studio IDE 建议启用，而其余规则被完全禁用。可选择性地选择加入或退出各条规则以禁用它们。 `<AnalysisMode>Default</AnalysisMode>`

若要查找每个可用规则的默认严重性以及了解规则是否在默认分析模式下启用，请参阅规则列表。

视警告为错误

如果在生成项目时使用 `-warnaserror` 标志，则所有代码分析警告也会被视为错误。如果不希望在出现 `-warnaserror` 时将代码质量警告 (CAxxxx) 视为错误，可在项目文件中将 `CodeAnalysisTreatWarningsAsErrors` MSBuild 属性设置为 `false`。

```
<PropertyGroup>
  <CodeAnalysisTreatWarningsAsErrors>false</CodeAnalysisTreatWarningsAsErrors>
</PropertyGroup>
```

你仍会看到任何代码分析警告，但它们不会中断生成。

最新更新

默认情况下，在升级到较新版本的 .NET SDK 时，你将获得最新的代码分析规则和默认规则严重性。如果你不希望出现此行为(例如，如果你想要确保未启用或禁用任何新规则)，可通过以下方式之一来替代此行为：

- 将 `AnalysisLevel` MSBuild 属性设置为特定值，以将警告锁定到相应的集。在升级到较新的 SDK 时，你仍会获得针对这些警告的 bug 修补程序，但系统不会启用新的警告，也不会禁用现有的警告。例如，若要将规则集锁定为随 .NET SDK 5.0 版本一起提供的规则集，请向项目文件添加以下条目。

  ```
  <PropertyGroup>
    <AnalysisLevel>5.0</AnalysisLevel>
  </PropertyGroup>
  ```

  > **TIP**
  > `AnalysisLevel` 属性的默认值为 `latest`，这意味着在你移动到较新版本的 .NET SDK 时，你始终会获得最新的代码分析规则。

  如需了解详细信息，以及查看可能值的列表，请参阅AnalysisLevel。

- 安装 Microsoft.CodeAnalysis.NetAnalyzers NuGet 包，将规则更新与 .NET SDK 更新分离。对于面向 .NET 5+ 的项目，安装该包将关闭内置 SDK 分析器。如果 SDK 所含的分析器程序集版本比 NuGet 包所含的版本更新，你会收到生成警告。若要禁用该警告，请将 `_SkipUpgradeNetAnalyzersNuGetWarning` 属性设置为 `true`。

  > **NOTE**
  > 如果你安装了 Microsoft.CodeAnalysis.NetAnalyzers NuGet 包，则不应将 EnableNETAnalyzers 属性添加到项目文件或 Directory.Build.props 文件。在安装了 NuGet 包并将 `EnableNETAnalyzers` 属性设置为 `true` 时，一个生成警告随即生成。

代码样式分析

通过代码样式分析("IDExxxx")规则，可在代码库中定义和维护一致的代码样式。默认的启用设置为：

- 命令行生成：默认情况下，对命令行生成上的所有 .NET 项目禁用代码样式分析。

  从 .NET 5 开始，无论是在命令行还是在 Visual Studio 内，你都可以在生成时启用代码样式分析。代码样式冲突显示为带有"IDE"前缀的警告或错误。这使你能够在生成时强制执行一致的代码样式。

- Visual Studio：默认情况下，代码样式分析作为代码重构快速操作对 Visual Studio 中的所有 .NET 项目启用。

有关代码样式分析规则的完整列表，请参阅代码样式规则。

生成时启用

通过 .NET 5 SDK 及更高版本，可在从命令行和 Visual Studio 生成时启用代码样式分析。(然而，出于性能方面的原因，一些代码样式规则仍仅适用于 Visual Studio IDE。)

执行以下步骤，在生成时启用代码样式分析：

1. 将 MSBuild 属性 EnforceCodeStyleInBuild 设置为 `true`。
2. 在 .editorconfig 文件中，配置你希望在生成时作为警告或错误运行的每个“IDE”代码样式规则。例如：

```
[*.{cs,vb}]
# IDE0040: Accessibility modifiers required (escalated to a build warning)
dotnet_diagnostic.IDE0040.severity = warning
```

或者，可将整个类别默认配置为警告或错误，然后选择性地禁用该类别中你不希望在生成时运行的规则。例如：

```
[*.{cs,vb}]

# Default severity for analyzer diagnostics with category 'Style' (escalated to build warnings)
dotnet_analyzer_diagnostic.category-Style.severity = warning

# IDE0040: Accessibility modifiers required (disabled on build)
dotnet_diagnostic.IDE0040.severity = silent
```

> **NOTE**
> 代码样式分析功能为实验性功能，可能会在 .NET 5 和 .NET 6 版本之间发生更改。

抑制警告

一种抑制规则冲突的方法是在 EditorConfig 文件中将该规则 ID 的严重性选项设置为 `none`。例如：

```
dotnet_diagnostic.CA1822.severity = none
```

有关抑制警告的详细信息和其他方式，请参阅如何抑制代码分析警告。

作为 GitHub 操作运行代码分析

dotnet/code-analysis GitHub 操作可用于在脱机模式下作为持续集成 (CI) 的一部分运行 .NET 代码分析器。有关详细信息，请参阅 .NET 代码分析 GitHub 操作。

第三方分析器

除了官方 .NET 分析器外，你也可以安装第三方分析器，如 StyleCop、Roslynator、XUnit Analyzers 和 Sonar Analyzer。

另请参阅

- 代码质量分析规则引用
- 代码样式分析规则引用
- Visual Studio 中的代码分析
- .NET 编译器平台 SDK
- 教程：编写第一个分析器和代码修补程序

代码分析的配置选项

2021/11/16 •

代码分析规则具有多种配置选项。这些选项是在分析器配置文件中使用 `<option key> = <option value>` 语法以键值对形式指定的。

最常见的配置选项是规则的严重性。你可以为所有分析器规则(包括代码质量规则和代码样式规则)配置严重性级别。例如，若要启用某个规则作为警告，可以向 EditorConfig 文件添加以下键值对。

```
dotnet_diagnostic.<rule ID>.severity = warning
```

你还可以配置其他选项，来自定义规则行为：

- 代码质量规则具有用于配置行为的其他选项，例如规则适用的方法名称。
- 代码样式规则具有自定义代码样式选项。
- 第三方分析器规则可以使用自定义键名和值格式定义各自的配置选项。

要在分析器配置文件中配置特定规则的严重性，请使用下面的语法：

```
dotnet_diagnostic.<rule ID>.severity = <severity>
```

常规选项

这些选项适用于整个代码分析。不能只将它们应用于一个规则或一组规则。

排除生成的代码

通过将 `generated_code = true | false` 条目添加到配置文件，可以配置额外的文件和文件夹以作为生成的代码来处理。.NET 代码分析器警告对生成的代码文件不起作用，比如对于设计器生成的文件，用户无法通过编辑这些文件来修复任何违规行为。在大多数情况下，代码分析器会跳过生成的代码文件，并且不会报告这些文件上的违规行为。

默认情况下，具有特定文件扩展名或自动生成的文件头的文件会被视为生成的代码文件。例如，以 `.designer.cs` 或 `.generated.cs` 结尾的文件名被视为生成的代码。使用这个配置选项可以指定更多命名模式。

例如，若要将名称以 `.MyGenerated.cs` 结尾的所有文件视为生成的代码，请添加以下条目：

```
[*.MyGenerated.cs]
generated_code = true
```

特定于规则的选项

特定于规则的选项可应用于一个规则、一组规则或所有规则。特定于规则的选项包括：

- 规则严重性级别
- 特定于代码质量规则的选项

严重性级别

下表显示了可为所有分析器规则(包括代码质量和代码样式规则)配置的各种规则严重性。

值	说明
`error`	违规行为以生成错误形式出现，并会导致生成失败。
`warning`	违规行为以生成警告形式出现，但不会导致生成失败（除非你已设置将警告视为错误的选项）。
`suggestion`	违规行为以生成消息形式出现，在 Visual Studio IDE 中以建议形式出现。
`silent`	违规行为对用户不可见。
`none`	完全禁止显示规则。
`default`	使用规则的默认严重性。Roslyn 分析器存储库列出了每个 .NET 版本的默认严重性。在该表中，“禁用”与 `none` 对应，“隐藏”与 `silent` 对应，“信息”与 `suggestion` 对应。

> **TIP**
> 若要了解规则严重性在 Visual Studio 中的显示方式，请参阅严重性级别。

范围

- **单一规则**

  若要为单个规则设置规则严重性，请使用以下语法。

  ```
  dotnet_diagnostic.<rule ID>.severity = <severity value>
  ```

- **规则类别**

  若要为某个规则类别设置默认规则严重性，请使用以下语法。

  ```
  dotnet_analyzer_diagnostic.category-<rule category>.severity = <severity value>
  ```

  在规则类别中列出并描述了不同的类别。此外，可以在其参考页上找到特定规则的类别，例如 CA1000。

- **所有规则**

  若要为所有分析器规则设置默认规则严重性，请使用以下语法。

  ```
  dotnet_analyzer_diagnostic.severity = <severity value>
  ```

> **IMPORTANT**
> 当你使用一个条目为多个规则配置严重性级别时，无论是为一个规则类别还是为所有规则配置，严重性都只适用于默认情况下启用的规则 。若要启用默认情况下已禁用的规则，必须执行以下任一操作：
>
> - 为每个规则添加一个显式 `dotnet_diagnostic.<rule ID>.severity = <severity>` 配置条目。
> - 在 .NET 6 及更高版本中，通过将 <AnalysisMode<Category>> 设置为 `All` 启用一种类别的规则。
> - 通过将 <AnalysisMode> 设置为 `AllEnabledByDefault` 来启用所有规则。

优先级

如果你有多个严重性配置条目可应用于同一个规则 ID，将按以下顺序选择优先级：

- 基于 ID 的单个规则的条目优先于一个类别的条目。
- 一个类别的条目优先于所有分析器规则的条目。

请考虑以下示例，其中 CA1822 属于“性能”类别：

```
[*.cs]
dotnet_diagnostic.CA1822.severity = error
dotnet_analyzer_diagnostic.category-performance.severity = warning
dotnet_analyzer_diagnostic.severity = suggestion
```

在前面的示例中，三个严重性条目都适用于 CA1822。但是，按照指定的优先级规则，第一个基于规则 ID 的条目优先于后续条目。在此示例中，CA1822 的有效严重性为 `error`。“性能”类别内的所有其他规则的严重性为 `warning`。

若要了解如何确定文件间的优先级，请参阅“配置文件”一文的“优先级”部分。

代码分析规则的配置文件

2021/11/16 •

代码分析规则具有多种配置选项。可以在下列任一分析器配置文件中将这些选项指定为键值对：

- EditorConfig 文件：基于文件或基于文件夹的配置选项。
- 全局 AnalyzerConfig 文件：项目级别配置选项。当某些项目文件位于项目文件夹外时，它非常有用。

EditorConfig

EditorConfig 文件用于提供适用于特定资源文件或文件夹的选项。选项位于节标头下，用于标识适用的文件和文件夹。为要配置的每个规则添加一个条目，并将其放置在相应的文件扩展名节下，例如 `[*.cs]`。

```
[*.cs]
<option_name> = <option_value>
```

在上面的示例中，`[*.cs]` 是一个 editorconfig 节标头，用于选择当前文件夹（包括子文件夹）中带有 `.cs` 文件扩展名的所有 C# 文件。接下来 `<option_name> = <option_value>` 这一条目是一个分析器选项，将应用于所有 C# 文件。

可将文件放在相应的目录中，将 EditorConfig 文件约定应用于文件夹、项目或整个存储库。可在生成时执行分析时以及在 Visual Studio 中编辑代码时应用这些选项。

> **NOTE**
> EditorConfig 选项仅应用于项目或目录中的源文件。作为 AdditionalFiles 包含在项目中的文件不被视为源文件，EditorConfig 选项不会应用于这些文件。若要将规则选项应用于非源文件，请在全局配置文件中指定该选项。

如果有一个现有的 .editorconfig 文件可用于编辑器设置（如缩进大小或是否剪裁尾随空格），可将代码分析配置选项放在同一文件中。

> **TIP**
> Visual Studio 提供 .editorconfig 项模板，通过该模板可轻松地将其中一个文件添加到项目中。有关详细信息，请参阅将 EditorConfig 文件添加到项目。

示例

下面是一个示例 EditorConfig 文件，用于配置选项和规则严重性：

```
# Remove the line below if you want to inherit .editorconfig settings from higher directories
root = true

# C# files
[*.cs]

#### Core EditorConfig Options ####

# Indentation and spacing
indent_size = 4
indent_style = space
tab_width = 4

#### .NET Coding Conventions ####

# this. and Me. preferences
dotnet_style_qualification_for_method = true

#### Diagnostic configuration ####

# CA1000: Do not declare static members on generic types
dotnet_diagnostic.CA1000.severity = warning
```

全局 AnalyzerConfig

从 .NET 5 SDK(在 Visual Studio 2019 版本 16.8 和更高版本中受支持)开始，还可配置包含全局 AnalyzerConfig 文件的分析器选项。这些文件用于提供适用于项目中所有源文件的选项，不考虑其文件名和文件路径。

与 EditorConfig 文件不同，全局配置文件不能用于为 IDE 配置编辑器样式设置，如缩进大小或是否剪裁尾随空格。而是专用于指定项目级别分析器配置选项。

格式

EditorConfig 文件必须包含节标头(如 `[*.cs]`)，以标识适用的文件和文件夹，但全局 AnalyzerConfig 文件没有节标头。相反，它们需要 `is_global = true` 格式的顶级条目，以便与常规 EditorConfig 文件区分开来。这表示文件中的所有选项都适用于整个项目。例如：

```
is_global = true
<option_name> = <option_value>
```

命名

EditorConfig 文件必须命名为 `.editorconfig`，而全局配置文件不需要有特定的名称或文件扩展名。但是，如果将这些文件命名为 `.globalconfig`，它们会隐式应用于当前文件夹(包括子文件夹)中的所有 C# 和 Visual Basic 项目。否则，必须将 `GlobalAnalyzerConfigFiles` 项显式添加到 MSBuild 项目文件中：

```
<ItemGroup>
  <GlobalAnalyzerConfigFiles Include="<path_to_global_analyzer_config>" />
</ItemGroup>
```

考虑以下命名建议：

- 最终用户应将其全局配置文件命名为 .globalconfig。
- NuGet 包创建者应将其全局配置文件命名为 <%Package_Name%>.globalconfig。
- MSBuild 生成工具的全局配置文件应命名为 <%Target_Name%>_Generated.globalconfig 或类似的名称。

> **NOTE**
> 即使文件命名为 `.globalconfig`，也需要顶级条目 `is_global = true`。

示例

下面是一个示例全局 AnalyzerConfig 文件，用于在项目级别配置选项和规则严重性：

```
# Top level entry required to mark this as a global AnalyzerConfig file
is_global = true

# NOTE: No section headers for configuration entries

#### .NET Coding Conventions ####

# this. and Me. preferences
dotnet_style_qualification_for_method = true:warning

#### Diagnostic configuration ####

# CA1000: Do not declare static members on generic types
dotnet_diagnostic.CA1000.severity = warning
```

优先级

EditorConfig 文件和全局 AnalyzerConfig 文件都为每个选项指定键值对。如果有多个条目具有相同键但值不同，则会发生冲突。以下优先规则用于解决冲突。

tttttt	tttt
在相同配置文件中	文件中后出现的条目优先。这适用于在单个 EditorConfig 文件中和单个全局 AnalyzerConfig 文件中的冲突条目。
在两个 EditorConfig 文件中	EditorConfig 文件位于文件系统更深层的条目（因此文件路径较长）优先。
在两个全局 AnalyzerConfig 文件中	.NET 5：系统会报告编译器警告并忽略这两个条目。 .NET 6 及更高版本：具有更高 `global_level` 值的文件中的条目优先。如果 `global_level` 未明确定义并且文件名为 .globalconfig，则 `global_level` 值默认为 `100`；对于所有其他全局 AnalyzerConfig 文件，`global_level` 默认为 `0`。如果具有冲突条目的配置文件的 `global_level` 值相等，则系统会报告编译器警告并忽略这两个条目。
在 EditorConfig 文件和全局 AnalyzerConfig 文件中	EditorConfig 文件中的条目优先。

严重性选项

严重性配置选项适用于下列其他优先规则：

- 在命令行上作为编译器选项（`-nowarn` 或 `-warnaserror`）指定的严重性选项始终会重写 EditorConfig 和全局 AnalyzerConfig 文件中指定的严重性配置选项。
- 规则集文件和 EditorConfig 或全局 AnalyzerConfig 文件中的严重性冲突条目的优先规则未定义。

> 规则集文件已弃用，改用 EditorConfig 和全局 AnalyzerConfig 文件。建议将规则集文件转换为等效的 EditorConfig 文件。

- 有关具有不同键的相关严重性选项的优先级规则的信息（例如，为单个规则和为规则所属的类别指定不同的严重性），请参阅代码分析的配置选项。

另请参阅

- EditorConfig 与全局 AnalyzerConfig 设计问题

如何禁止显示代码分析警告

2021/11/16 •

本文介绍了在开发 .NET 应用时抑制代码分析警告的不同方法。

> **TIP**
> 如果使用 Visual Studio 作为开发环境，灯泡菜单可提供一些选项来生成用于抑制警告的代码。有关详细信息，请参阅抑制冲突。

禁用规则

禁用导致警告的代码分析规则后，将对整个文件或项目禁用规则（具体取决于使用的配置文件的作用域）。若要禁用规则，请在配置文件中将其严重性设置为 `none`。

```
[*.{cs,vb}]
dotnet_diagnostic.<rule-ID>.severity = none
```

有关规则严重性的详细信息，请参阅配置规则严重性。

使用预处理器指令

使用 #pragma 警告 (C#) 或禁用 (Visual Basic) 指令来仅抑制特定代码行的警告。

```
    try { ... }
    catch (Exception e)
    {
#pragma warning disable CA2200 // Rethrow to preserve stack details
        throw e;
#pragma warning restore CA2200 // Rethrow to preserve stack details
    }
```

```
    Try
        ...
    Catch e As Exception
#Disable Warning CA2200 ' Rethrow to preserve stack details
        Throw e
#Enable Warning CA2200 ' Rethrow to preserve stack details
    End Try
```

使用 SuppressMessageAttribute

可以使用 SuppressMessageAttribute 在源文件中或项目的全局抑制文件（GlobalSuppressions.cs 或 GlobalSuppressions.vb）中抑制警告 。此特性提供了一种仅在项目或文件的特定部分抑制警告的方法。

SuppressMessageAttribute 特性的两个必需的位置参数 是：规则的类别和规则 ID 。下面的代码片段传递这些参数的 `"Usage"` 和 `"CA2200:Rethrow to preserve stack details"`。

```
[System.Diagnostics.CodeAnalysis.SuppressMessage("Usage", "CA2200:Rethrow to preserve stack details",
Justification = "Not production code.")]
private static void IngorableCharacters()
{
    try
    {
        ...
    }
    catch (Exception e)
    {
        throw e;
    }
}
```

如果将该特性添加到全局抑制文件中，则会将抑制的作用域设置到所需的级别，例如 `"member"`。使用 Target 属性指定应抑制其警告的 API。

```
[assembly: SuppressMessage("Usage", "CA2200:Rethrow to preserve stack details", Justification = "Not
production code.", Scope = "member", Target = "~M:MyApp.Program.IngorableCharacters")]
```

将文档 ID 用于要在 `Target` 属性中引用的 API。有关文档 ID 的信息，请参阅文档 ID 格式。

若要对未映射到显式提供的用户源的编译器生成代码抑制警告，必须将抑制特性放置在全局抑制文件中。例如，下面的代码将抑制针对编译器发出的构造函数的冲突：

```
[module: SuppressMessage("Design", "CA1055:AbstractTypesDoNotHavePublicConstructors", Scope="member",
Target="MyTools.Type..ctor()")]
```

另请参阅

- 抑制冲突 (Visual Studio)

代码质量规则配置选项

2021/11/16 •

除了配置严重性外，代码质量规则还有其他配置选项。例如，可以将每个代码质量分析器配置为仅应用于代码库的特定部分。通过向指定规则严重性和常规编辑器首选项的同一个 EditorConfig 文件添加键值对，可指定这些选项。

选项作用域

每个优化选项都可以针对所有规则、某个规则类别（例如“安全性”或“设计”）或某个特定规则进行配置。

┬T矜玥

若要为所有规则配置选项，请使用下面的语法：

"	"
dotnet_code_quality.<OptionName> = <OptionValue>	`dotnet_code_quality.api_surface = public`

`<OptionName>` 的值列在选项下。

规则类别

若要为某个规则类别配置选项，请使用下面的语法：

"	"
dotnet_code_quality.<RuleCategory>.<OptionName> = OptionValue	`dotnet_code_quality.Naming.api_surface = public`

下表列出了 `<RuleCategory>` 的可用值。

Design Documentation Globalization Interoperability

Maintainability Naming Performance SingleFile

Reliability Security Usage

特定规则

若要为某个特定规则配置选项，请使用下面的语法：

"	"
dotnet_code_quality.<RuleId>.<OptionName> = <OptionValue>	`dotnet_code_quality.CA1040.api_surface = public`

选项

本节列出了一部分可用选项。若要查看可用选项的完整列表，请参阅分析器配置。

- api_surface
- exclude_async_void_methods
- exclude_single_letter_type_parameters

- output_kind
- required_modifiers
- exclude_extension_method_this_parameter
- null_check_validation_methods
- additional_string_formatting_methods
- excluded_type_names_with_derived_types
- excluded_symbol_names
- disallowed_symbol_names

api_surface

[illegible]	[illegible]	[illegible]	[illegible]

要分析 API 图面的哪个部分	`public`
	`internal` 或 `friend`
	`private`
	`all`
	
	用逗号 (,) 分隔多个值
		`public`	CA1000 CA1003 CA1008 CA1010
			CA1012 CA1024 CA1027 CA1028
			CA1030 CA1036 CA1040 CA1041
			CA1043 CA1044 CA1051 CA1052
			CA1054 CA1055 CA1056 CA1058
			CA1063 CA1708 CA1710 CA1711
			CA1714 CA1715 CA1716 CA1717
			CA1720 CA1721 CA1725 CA1801
			CA1802 CA1815 CA1819 CA2217
			CA2225 CA2226 CA2231 CA2234

exclude_async_void_methods

[illegible]	[illegible]	[illegible]	[illegible]
是否忽略不返回值的异步方法	`true` `false`	`false`	CA2007

> **NOTE**
> 早期版本中将此选项命名为 `skip_async_void_methods`。

exclude_single_letter_type_parameters

[illegible]	[illegible]	[illegible]	[illegible]
是否从规则中排除单字符的类型参数，例如，`Collection<S>` 中的 `S`	`true` `false`	`false`	CA1715

> **NOTE**
> 早期版本中将此选项命名为 `allow_single_letter_type_parameters`。

output_kind

□□	□□□□	□□□	□□□□□□
指定应分析项目中生成此程序集类型的代码	OutputKind 枚举的一个或多个字段 用逗号 (,) 分隔多个值	所有输出种类	CA2007

required_modifiers

□□	□□□□	□□□	□□□□□□
指定应分析的 API 所需的修饰符	以下允许的修饰符表中的一个或多个值 用逗号 (,) 分隔多个值	取决于每个规则	CA1802

□□□□□□	□□
`none`	无修饰符要求
`static` 或 `Shared`	必须声明为 `static`（在 Visual Basic 中为 `Shared`）
`const`	必须声明为 `const`
`readonly`	必须声明为 `readonly`
`abstract`	必须声明为 `abstract`
`virtual`	必须声明为 `virtual`
`override`	必须声明为 `override`
`sealed`	必须声明为 `sealed`
`extern`	必须声明为 `extern`
`async`	必须声明为 `async`

exclude_extension_method_this_parameter

□□	□□□□	□□□	□□□□□□
是否跳过对扩展方法的 `this` 参数的分析	`true` `false`	`false`	CA1062

null_check_validation_methods

[illegible]	[illegible]	[illegible]	[illegible]
null 检查验证方法的名称，这些方法用于确定传递给方法的参数不是 null	允许的方法名称格式（以 `|` 分隔）： - 仅方法名称（包括具有相应名称的所有方法，不考虑包含的类型或命名空间） - 完全限定的名称，使用符号的文档 ID 格式，前缀为 `M:`（可选）	无	CA1062

additional_string_formatting_methods

[illegible]	[illegible]	[illegible]	[illegible]
其他字符串格式设置方法的名称	允许的方法名称格式（以 `|` 分隔）： - 仅方法名称（包括具有相应名称的所有方法，不考虑包含的类型或命名空间） - 完全限定的名称，使用符号的文档 ID 格式，前缀为 `M:`（可选）	无	CA2241

excluded_type_names_with_derived_types

[illegible]	[illegible]	[illegible]	[illegible]
类型的名称，用于将类型及其所有派生类型从分析范围内排除	允许的符号名称格式（以 `|` 分隔）： - 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间） - 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）	无	CA1303

excluded_symbol_names

[illegible]	[illegible]	[illegible]	[illegible]
从分析范围排除的符号的名称	允许的符号名称格式（以 `|` 分隔）： - 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间） - 完全限定的名称，使用符号的文档 ID 格式 每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:` 前缀、表示类型的 `T:` 前缀，以及表示命名空间的 `N:` 前缀。 - `.ctor` 表示构造函数，`.cctor` 表示静态构造函数	无	CA1062 CA1303 CA2000 CA2100 CA2301 CA2302 CA2311 CA2312 CA2321 CA2322 CA2327 CA2328 CA2329 CA2330 CA3001 CA3002 CA3003 CA3004 CA3005 CA3006 CA3007 CA3008 CA3009 CA3010 CA3011 CA3012 CA5361 CA5376 CA5377 CA5378 CA5380 CA5381 CA5382 CA5383 CA5384 CA5387 CA5388 CA5389 CA5390

disallowed_symbol_names

[illegible]	[illegible]	[illegible]	[illegible]
不允许出现在分析上下文中的符号的名称	允许的符号名称格式(以 `|` 分隔): - 仅符号名称(包括具有相应名称的所有符号,不考虑包含的类型或命名空间) - 完全限定的名称,使用符号的文档 ID 格式 每个符号名称都需要带有一个符号类型前缀,例如表示方法的 `M:` 前缀、表示类型的 `T:` 前缀,以及表示命名空间的 `N:` 前缀。 - `.ctor` 表示构造函数,`.cctor` 表示静态构造函数	无	CA1031

预定义的配置文件

2021/11/16 •

使用预定义的 EditorConfig 和规则集文件，可以快速轻松地启用某一类别的代码质量规则，如安全性或设计规则。通过启用特定类别的规则，可以确定目标问题和特定情况。若要访问这些预定义的文件，请安装 Microsoft.CodeAnalysis.NetAnalyzers NuGet 分析器包。

Microsoft.CodeAnalysis.NetAnalyzers 包括用于以下规则类别的预定义 EditorConfig 文件和规则集：

- ┬T矽玥
- 数据流
- 设计
- 文档
- 全球化
- 互操作性
- 可维护性
- 命名
- 性能
- 从 FxCop 移植
- 可靠性
- 安全性
- 使用情况

每类规则都有一个 EditorConfig 或规则集文件，用于：

- 启用相应类别中的所有规则（并禁用所有其他规则）
- 使用每个规则由默认设置启用的默认严重性（并禁用所有其他规则）

> **TIP**
> “所有规则”类别具有一个额外的 EditorConfig 或规则集文件，用于禁用所有规则。可使用此文件快速清除项目中的任何分析器警告或错误。

预定义的 EditorConfig 文件

Microsoft.CodeAnalysis.NetAnalyzers 分析器包的预定义 EditorConfig 文件位于 NuGet 包安装位置的“editorconfig”子目录中。例如，用于启用所有安全规则的 EditorConfig 文件位于 editorconfig/SecurityRulesEnabled/.editorconfig。

请将所选的 .editorconfig 文件复制到项目的根目录中。

预定义规则集

Microsoft.CodeAnalysis.NetAnalyzers 分析器包的预定义规则集文件位于 NuGet 包安装位置的“rulesets”子目录中。例如，用于启用所有安全规则的规则集文件位于 rulesets/SecurityRulesEnabled.ruleset。请复制一个或多个规则集，并将其粘贴到包含你的项目的目录中。

请参阅

- 分析器配置
- EditorConfig 的 .NET 代码样式规则选项

代码样式规则选项

2021/11/16 •

通过在 EditorConfig 文件中定义 .NET 代码样式规则选项，可以在代码库中定义和保持一致的代码样式。在你编辑代码时，Visual Studio 等各种开发 IDE 会实施这些规则。对于 .NET 项目，还可以在生成时强制执行这些规则。你可以启用或禁用单个规则，并可通过严重性级别配置强制执行每个规则的程度。

> **TIP**
>
> - 在 EditorConfig 文件中定义代码样式选项，就是在配置代码样式分析器分析代码的方式。EditorConfig 文件是适用于这些分析器的配置文件。
> - 在 Visual Studio 中，代码样式选项还可以在文本编辑器选项对话框中进行设置。这些是按用户选项，只有在 Visual Studio 中进行编辑时才会采用这些选项。在生成时不会采用这些选项，也不会由其他 IDE 采用。此外，如果在 Visual Studio 中打开的项目或解决方案包含 EditorConfig 文件，则优先采用 EditorConfig 文件中的选项。

代码样式规则分为以下子类别：

- 语言规则
- 不必要的代码规则
- 格式设置规则
- 命名规则

其中每个子类别都定义了各自的语法来指定选项。有关这些规则和相应选项的详细信息，请参阅代码样式规则引用。

EditorConfig 文件示例

下面是具有默认选项的示例 .editorconfig 文件，可帮助你入门。

> **TIP**
>
> 在 Visual Studio 2019 及更高版本中（在 Windows 上），你可以生成此文件并将其保存到项目中，步骤为“工具” > “选项” > “文本编辑器”> [“C#”或“基本”] >“代码样式” > “常规” 。然后，单击“从设置生成 .editorconfig 文件”按钮。有关详细信息，请参阅代码样式首选项。

```
###############################
# Core EditorConfig Options   #
###############################
root = true
# All files
[*]
indent_style = space

# XML project files
[*.{csproj,vbproj,vcxproj,vcxproj.filters,proj,projitems,shproj}]
indent_size = 2

# XML config files
[*.{props,targets,ruleset,config,nuspec,resx,vsixmanifest,vsct}]
indent_size = 2
```

```
# Code files
[*.{cs,csx,vb,vbx}]
indent_size = 4
insert_final_newline = true
charset = utf-8-bom
###############################
# .NET Coding Conventions     #
###############################
[*.{cs,vb}]
# Organize usings
dotnet_sort_system_directives_first = true
# this. preferences
dotnet_style_qualification_for_field = false:silent
dotnet_style_qualification_for_property = false:silent
dotnet_style_qualification_for_method = false:silent
dotnet_style_qualification_for_event = false:silent
# Language keywords vs BCL types preferences
dotnet_style_predefined_type_for_locals_parameters_members = true:silent
dotnet_style_predefined_type_for_member_access = true:silent
# Parentheses preferences
dotnet_style_parentheses_in_arithmetic_binary_operators = always_for_clarity:silent
dotnet_style_parentheses_in_relational_binary_operators = always_for_clarity:silent
dotnet_style_parentheses_in_other_binary_operators = always_for_clarity:silent
dotnet_style_parentheses_in_other_operators = never_if_unnecessary:silent
# Modifier preferences
dotnet_style_require_accessibility_modifiers = for_non_interface_members:silent
dotnet_style_readonly_field = true:suggestion
# Expression-level preferences
dotnet_style_object_initializer = true:suggestion
dotnet_style_collection_initializer = true:suggestion
dotnet_style_explicit_tuple_names = true:suggestion
dotnet_style_null_propagation = true:suggestion
dotnet_style_coalesce_expression = true:suggestion
dotnet_style_prefer_is_null_check_over_reference_equality_method = true:silent
dotnet_style_prefer_inferred_tuple_names = true:suggestion
dotnet_style_prefer_inferred_anonymous_type_member_names = true:suggestion
dotnet_style_prefer_auto_properties = true:silent
dotnet_style_prefer_conditional_expression_over_assignment = true:silent
dotnet_style_prefer_conditional_expression_over_return = true:silent
###############################
# Naming Conventions          #
###############################
# Style Definitions
dotnet_naming_style.pascal_case_style.capitalization             = pascal_case
# Use PascalCase for constant fields
dotnet_naming_rule.constant_fields_should_be_pascal_case.severity = suggestion
dotnet_naming_rule.constant_fields_should_be_pascal_case.symbols  = constant_fields
dotnet_naming_rule.constant_fields_should_be_pascal_case.style    = pascal_case_style
dotnet_naming_symbols.constant_fields.applicable_kinds            = field
dotnet_naming_symbols.constant_fields.applicable_accessibilities  = *
dotnet_naming_symbols.constant_fields.required_modifiers          = const
###############################
# C# Coding Conventions       #
###############################
[*.cs]
# var preferences
csharp_style_var_for_built_in_types = true:silent
csharp_style_var_when_type_is_apparent = true:silent
csharp_style_var_elsewhere = true:silent
# Expression-bodied members
csharp_style_expression_bodied_methods = false:silent
csharp_style_expression_bodied_constructors = false:silent
csharp_style_expression_bodied_operators = false:silent
csharp_style_expression_bodied_properties = true:silent
csharp_style_expression_bodied_indexers = true:silent
csharp_style_expression_bodied_accessors = true:silent
# Pattern matching preferences
csharp_style_pattern_matching_over_is_with_cast_check = true:suggestion
csharp_style_pattern_matching_over_as_with_null_check = true:suggestion
```

```
csharp_style_pattern_matching_over_as_with_null_check = true:suggestion
# Null-checking preferences
csharp_style_throw_expression = true:suggestion
csharp_style_conditional_delegate_call = true:suggestion
# Modifier preferences
csharp_preferred_modifier_order =
public,private,protected,internal,static,extern,new,virtual,abstract,sealed,override,readonly,unsafe,volatil
e,async:suggestion
# Expression-level preferences
csharp_prefer_braces = true:silent
csharp_style_deconstructed_variable_declaration = true:suggestion
csharp_prefer_simple_default_expression = true:suggestion
csharp_style_pattern_local_over_anonymous_function = true:suggestion
csharp_style_inlined_variable_declaration = true:suggestion
###############################
# C# Formatting Rules         #
###############################
# New line preferences
csharp_new_line_before_open_brace = all
csharp_new_line_before_else = true
csharp_new_line_before_catch = true
csharp_new_line_before_finally = true
csharp_new_line_before_members_in_object_initializers = true
csharp_new_line_before_members_in_anonymous_types = true
csharp_new_line_between_query_expression_clauses = true
# Indentation preferences
csharp_indent_case_contents = true
csharp_indent_switch_labels = true
csharp_indent_labels = flush_left
# Space preferences
csharp_space_after_cast = false
csharp_space_after_keywords_in_control_flow_statements = true
csharp_space_between_method_call_parameter_list_parentheses = false
csharp_space_between_method_declaration_parameter_list_parentheses = false
csharp_space_between_parentheses = false
csharp_space_before_colon_in_inheritance_clause = true
csharp_space_after_colon_in_inheritance_clause = true
csharp_space_around_binary_operators = before_and_after
csharp_space_between_method_declaration_empty_parameter_list_parentheses = false
csharp_space_between_method_call_name_and_opening_parenthesis = false
csharp_space_between_method_call_empty_parameter_list_parentheses = false
# Wrapping preferences
csharp_preserve_single_line_statements = true
csharp_preserve_single_line_blocks = true
###############################
# VB Coding Conventions       #
###############################
[*.vb]
# Modifier preferences
visual_basic_preferred_modifier_order =
Partial,Default,Private,Protected,Public,Friend,NotOverridable,Overridable,MustOverride,Overloads,Overrides,
MustInherit,NotInheritable,Static,Shared,Shadows,ReadOnly,WriteOnly,Dim,Const,WithEvents,Widening,Narrowing,
Custom,Async:suggestion
```

请参阅

- 代码样式分析规则引用
- 在生成时强制执行代码样式
- Visual Studio 中的快速操作
- 在 Visual Studio 中创建可移植的自定义编辑器选项
- .NET Compiler Platform"Roslyn".editorconfig 文件
- .NET 运行时 .editorconfig 文件

规则类别

2021/11/16 •

每个代码分析规则都属于某种规则类别。例如，设计规则支持遵从 .NET 设计准则，而安全规则可帮助防止出现安全漏洞。你可为整个规则类别配置严重性级别。还可以按类别配置其他选项。

下表显示了不同的代码分析规则类别，并提供指向每个类别中的规则的链接。它还列出了 EditorConfig 文件中要使用的配置值，以按类别批量配置规则严重性。例如，若要将安全规则冲突的严重性设置为错误，则 EditorConfig 条目将为 `dotnet_analyzer_diagnostic.category-Security.severity = error`。

> **TIP**
>
> 使用 `dotnet_analyzer_diagnostic.category-<category>.severity` 语法设置一类规则的严重性并不适用于默认禁用的规则。但是，从 .NET 6 开始，可以使用 AnalysisMode<Category> 项目属性启用某一类别中的所有规则。

		EDITORCONFIG
代码质量规则	此类别包含以下其他规则：IDE0051、IDE0064、IDE0076。	`dotnet_analyzer_diagnostic.category-CodeQuality.severity`
设计规则	设计规则支持遵从 .NET Framework 设计准则。	`dotnet_analyzer_diagnostic.category-Design.severity`
文档规则	文档规则支持通过对外部可见的 API 正确使用 XML 文档注释来编写记录详尽的库。	`dotnet_analyzer_diagnostic.category-Documentation.severity`
全球化规则	全球化规则支持世界通用库和应用程序。	`dotnet_analyzer_diagnostic.category-Globalization.severity`
可移植性和互操作性规则	可移植性规则支持跨不同平台的可移植性。互操作性规则支持与 COM 客户端交互。	`dotnet_analyzer_diagnostic.category-Interoperability.severity`
可维护性规则	可维护性规则支持库和应用程序维护。	`dotnet_analyzer_diagnostic.category-Maintainability.severity`
命名规则	命名规则支持遵从 .NET 设计准则的命名约定。	`dotnet_analyzer_diagnostic.category-Naming.severity`
性能规则	性能规则支持高性能库和应用程序。	`dotnet_analyzer_diagnostic.category-Performance.severity`
单文件规则	单文件规则支持单文件应用程序。	`dotnet_analyzer_diagnostic.category-SingleFile.severity`
可靠性规则	可靠性规则支持库和应用程序可靠性（例如正确使用内存和线程）。	`dotnet_analyzer_diagnostic.category-Reliability.severity`

[	[	EDITORCONFIG [
安全规则	安全规则支持更安全的库和应用程序。这些规则有助于防止程序出现安全漏洞。	`dotnet_analyzer_diagnostic.category-Security.severity`
样式规则	样式规则支持代码库中的代码样式保持一致。这些规则以“IDE”前缀开头。	`dotnet_analyzer_diagnostic.category-Style.severity`
用法规则	用法规则支持正确使用 .NET。	`dotnet_analyzer_diagnostic.category-Usage.severity`

代码质量规则

2021/11/16 •

.NET 代码分析提供旨在提高代码质量的规则。这些规则分为设计、全球化、性能和安全性等领域。某些规则特定于 .NET API 用法，而其他规则与通用代码质量相关。

规则索引

下表列出了代码质量分析规则。

ID	
CA1000:不要在泛型类型中声明静态成员	调用泛型类型的静态成员时，必须指定该类型的类型参数。当调用不支持推理的泛型实例成员时，必须指定该成员的类型参数。在上述两种情况下，用于指定类型自变量的语法不同，但很容易混淆。
CA1001:具有可释放字段的类型应该是可释放的	一个类声明并实现 System.IDisposable 类型的实例字段，但该类不实现 IDisposable。声明 IDisposable 字段的类间接拥有非托管资源，并且应该实现 IDisposable 接口。
CA1002:不要公开泛型列表	Collections.Generic.List<(Of <(T>)>) 是针对性能(而非继承)设计的泛型集合。因此，List 不包含任何虚拟成员。应改为公开针对继承设计的泛型集合。
CA1003:使用泛型事件处理程序实例	某个类型包含的委托返回 void，该委托的签名包含两个参数(第一个参数是对象，第二个参数是可以分配给 EventArgs 的类型)，而且包含程序集针对的是 Microsoft .NET Framework 2.0。
CA1005:避免泛型类型的参数过多	泛型类型包含的类型参数越多，越难以知道并记住每个类型参数各代表什么。它通常有一个类型参数，如在 List<T> 中，而在某些情况下有两个类型参数，如在 Dictionary<TKey, TValue> 中。但是，如果存在两个以上的类型参数，则大多数用户都会感到过于困难。
CA1008:枚举应具有零值	像其他值类型一样，未初始化枚举的默认值为零。无标志特性的枚举应通过使用零值来定义成员，这样默认值即为该枚举的有效值。如果应用了 FlagsAttribute 特性的枚举定义值为零成员，则该成员的名称应为“None”，以指示枚举中尚未设置值。
CA1010:集合应实现泛型接口	若要扩大集合的用途，应实现某个泛型集合接口。然后，可以使用该集合来填充泛型集合类型。
CA1012:抽象类型不应具有构造函数	抽象类型的构造函数只能由派生类型调用。由于公共构造函数用于创建类型的实例，但无法为抽象类型创建实例，因此具有公共构造函数的抽象类在设计上是错误的。
CA1014:用 CLSCompliantAttribute 标记程序集	公共语言规范 (CLS) 定义了程序集在跨编程语言使用时必须符合的命名限制、数据类型和规则。好的设计要求所有程序集用 CLSCompliantAttribute 显式指示 CLS 合规性。如果程序集没有此特性，则该程序集即不合规。
CA1016:用 AssemblyVersionAttribute 标记程序集	.NET 使用版本号来唯一标识程序集，并绑定到强名称程序集中的类型。版本号与版本和发行者策略一起使用。默认情况下，仅使用用于生成应用程序的程序集版本运行应用程序。

ID	
CA1017:用 ComVisibleAttribute 标记程序集	ComVisibleAttribute 决定 COM 客户端如何访问托管代码。合理的设计指出程序集将显式指示 COM 可见性。可以设置整个程序集的 COM 可见性，然后重写各个类型和类型成员的 COM 可见性。如果此特性不存在，则程序集的内容对 COM 客户端可见。
CA1018:用 AttributeUsageAttribute 标记特性	当定义自定义特性时，用 AttributeUsageAttribute 标记该特性，以指示源代码中可以应用自定义特性的位置。特性的含义和预定用法将决定它在代码中的有效位置。
CA1019:定义特性参数的访问器	特性可以定义强制自变量，在对目标应用该特性时必须指定这些自变量。这些实参也称为位置实参，因为它们将作为位置形参提供给特性构造函数。对于每一个强制变量，特性还必须提供一个相应的只读属性，以便可以在执行时检索该变量的值。特性还可以定义可选实参，可选实参也称为命名实参。这些变量按名称提供给特性构造函数，并且必须具有相应的读/写属性。
CA1021:避免使用 out 参数	通过引用(使用 out 或 ref)传递类型要求具有使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。另外，out 和 ref 参数之间的差异没有得到广泛了解。
CA1024:在适用处使用属性	公共或受保护方法的名称以“Get”开头，没有采用任何参数或返回的值不是数组。该方法可能很适于成为属性。
CA1027:用 FlagsAttribute 标记枚举	枚举是一种值类型，它定义一组相关的已命名常数。如果可以按照有意义的方式组合一个枚举的已命名常数，则对该枚举应用 FlagsAttribute。
CA1028:枚举存储应为 Int32	枚举是一种值类型，它定义一组相关的已命名常数。默认情况下，System.Int32 数据类型用于存储常量值。尽管您可以更改此基础类型，然而对于大多数情况，既不需要，也不建议您这样做。
CA1030:在适用处使用事件	该规则检测名称通常用于事件的方法。如果为响应明确定义的状态更改而调用一个方法，则应由事件处理程序调用该方法。调用该方法的对象应引发事件而不是直接调用该方法。
CA1031:不要捕捉一般异常类型	不应捕捉一般异常。捕捉更具体的异常，或者在执行 catch 块中的最后一条语句时重新引发一般异常。
CA1032:实现标准异常构造函数	如果不能提供完整的构造函数集，要正确处理异常将变得比较困难。
CA1033:接口方法应可由子类型调用	未密封的外部可见类型提供了显式实现公共接口的方法，但没有提供具有相同名称的其他外部可见方法。
CA1034:嵌套类型不应是可见的	嵌套类型是在另一个类型的范围中声明的类型。嵌套类型用于封装包含类型的私有实现详细信息。如果用于此用途，则嵌套类型不应是外部可见的。
CA1036:重写可比较类型中的方法	公共或受保护类型实现 System.IComparable 接口。它不重写 Object.Equals，也不重载表示相等、不等、小于或大于的语言特定运算符。
CA1040:避免使用空接口	接口定义提供某个行为或使用协定的成员。接口所描述的功能可以被任何类型采用，而不管该类型出现在继承层次结构中的哪个位置。类型通过实现接口的成员来实现接口。空接口无法定义任何成员；因此，它无法定义可以实现的协定。

ID	
CA1041:提供 ObsoleteAttribute 消息	用未指定其 ObsoleteAttribute.Message 属性的 System.ObsoleteAttribute 特性来标记类型或成员。当编译用 ObsoleteAttribute 标记的类型或成员时，将显示该特性的 Message 属性。这将为用户提供有关已过时的类型或成员的信息。
CA1043:将整型或字符串参数用于索引器	索引器(即索引属性)应将整型或字符串类型用于索引。这些类型一般用于为数据结构编制索引，并且提高库的可用性。应仅限于在设计时无法指定特定整型或字符串类型的情况下使用 Object 类型。
CA1044:属性不应是只写的	虽然可以接受且经常需要使用只读属性，但设计准则禁止使用只写属性。这是因为允许用户设置值但又禁止该用户查看这个值不能提供任何安全性。而且，如果没有读访问，将无法查看共享对象的状态，使其用处受到限制。
CA1045:不要通过引用来传递类型	通过引用(使用 out 或 ref)传递类型要求具有以下能力：使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。为一般用户进行设计的库架构师不应指望用户能熟练运用 `out` 或 `ref` 参数。
CA1046:不要对引用类型重载相等运算符	对于引用类型，相等运算符的默认实现几乎始终是正确的。默认情况下，仅当两个引用指向同一对象时，它们才相等。
CA1047:不要在密封类型中声明受保护的成员	类型声明受保护的成员，使继承类型可以访问或重写该成员。按照定义，不能继承密封类型，这表示不能调用密封类型上的受保护方法。
CA1050:在命名空间中声明类型	应在命名空间内声明类型以避免名称冲突，并作为一种在对象层次结构中组织相关类型的方式。
CA1051:不要声明可见实例字段	字段的主要用途应是作为实现的详细信息。字段应为 private 或 internal，并应通过使用属性公开这些字段。
CA1052:应密封静态容器类型	公共或受保护类型仅包含静态成员，而且没有用 sealed(C# 参考)(NotInheritable) 修饰符声明该类型。应使用 sealed 修饰符标记不希望被继承的类型，以免将其用作基类型。
CA1053:静态容器类型不应具有构造函数	公共或嵌套公共类型只声明了静态成员，但具有公共或受保护的默认构造函数。由于调用静态成员不需要类型的示例，因此没必要使用构造函数。为安全起见，字符串重载应使用字符串自变量调用统一资源标识符 (URI) 重载。
CA1054:URI 参数不应为字符串	如果某方法采用 URI 的字符串表示形式，则应提供采用 URI 类的实例的相应重载，该重载以安全的方式提供这些服务。
CA1055:URI 返回值不应是字符串	此规则假定该方法返回 URI。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。
CA1056:URI 属性不应是字符串	此规则假定属性表示统一资源标识符 (URI)。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。
CA1058:类型不应扩展某些基类型	外部可见的类型扩展某些基类型。请使用某个备选项。
CA1060：将 P/Invoke 移动到 NativeMethods 类	平台调用方法(例如标以 System.Runtime.InteropServices.DllImportAttribute 特性的那些方法，或在 Visual Basic 中使用 `Declare` 关键字定义的方法)可以访问非托管代码。这些方法应属于 NativeMethods、SafeNativeMethods 或 UnsafeNativeMethods 类。

ID	
CA1061:不要隐藏基类方法	如果派生方法的参数签名只是在类型方面有所不同，而且与基方法的参数签名中的对应类型相比，这些类型的派生方式更弱，则基类型中的方法由派生类型中的同名方法隐藏。
CA1062:验证公共方法的参数	对于传递给外部可见方法的所有引用自变量，都应检查其是否为 null。
CA1063:正确实现 IDisposable	所有的 IDisposable 类型都应当正确实现 Dispose 模式。
CA1064:异常应该是公共的	内部异常仅在其自己的内部范围内可见。当异常超出内部范围后，只能使用基异常来捕获该异常。如果内部异常继承自 Exception、SystemException 或 ApplicationException，则外部代码将没有足够的信息来了解如何处理该异常。
CA1065:不要在意外的位置引发异常	不应引发异常的方法引发了异常。
CA1066：重写 Equals 时实现 IEquatable	值类型替代 Equals 方法，但不实现 IEquatable<T>。
CA1067：实现 IEquatable 时重写 Equals	类型实现 IEquatable<T>，但不替代 Equals 方法。
CA1068:CancellationToken 参数必须最后出现	方法具有 CancellationToken 参数，但它不是最后一个参数。
CA1069:枚举不得具有重复值	枚举具有多个成员，这些成员显式分配有相同常数值。
CA1070:不要将事件字段声明为“虚拟”	类字段事件被声明为“虚拟”。
CA1200:不要使用带前缀的 cref 标记	XML 文档标记中的 cref 属性是指“代码引用”。它指定标记的内部文本是一个代码元素，例如类型、方法或属性。避免使用带有前缀的 `cref` 标记，因为它会阻止编译器验证引用。它还会阻止 Visual Studio 集成开发环境 (IDE) 在重构过程中查找和更新这些符号引用。
CA1303:请不要将文本作为本地化参数传递	某外部可见的方法将一个字符串字面量作为参数传递给 .NET 构造函数或方法，该字符串应该是可本地化的字符串。
CA1304:指定 CultureInfo	某方法或构造函数调用的成员有一个接受 System.Globalization.CultureInfo 参数的重载，但该方法或构造函数没有调用接受 CultureInfo 参数的重载。如果未提供 CultureInfo 或 System.IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生您想要的效果。
CA1305:指定 IFormatProvider	某方法或构造函数调用的一个或多个成员有接受 System.IFormatProvider 参数的重载，但该方法或构造函数没有调用接受 IFormatProvider 参数的重载。如果未提供 System.Globalization.CultureInfo 或 IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生您想要的效果。
CA1307:为了清晰起见，请指定 StringComparison	字符串比较运算使用不设置 StringComparison 参数的方法重载。
CA1308:将字符串规范化为大写	字符串应正常化为大写字母。少量字符转换为小写字母后不能再转换回来。
CA1309:使用按顺序的 StringComparison	非语义的字符串比较运算不会将 StringComparison 参数设置为 Ordinal 或 OrdinalIgnoreCase。因此，通过将参数显式设置为 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase，通常可以提高代码的速度、正确性和可靠性。

ID	
CA1310：为了确保正确，请指定 StringComparison	字符串比较操作使用未设置 StringComparison 参数的方法重载，并默认使用区域性特定的字符串比较。
CA1401：P/Invokes 应为不可见	公共类型中的公共或受保护方法具有 System.Runtime.InteropServices.DllImportAttribute 属性（在 Visual Basic 中由 Declare 关键字实现）。这些方法不能公开。
CA1416：验证平台兼容性	在组件上使用依赖于平台的 API 会使代码无法用于所有平台。
CA1417：请勿对 P/Invokes 的字符串参数使用 `OutAttribute`	如果该字符串为暂存的字符串，则通过包含 `OutAttribute` 的值传递的字符串参数可能使运行时变得不稳定。
CA1418：使用有效的平台字符串	平台兼容性分析器需要有效的平台名称和版本。
CA1501:避免过度继承	类型在继承层次结构中的深度超过四级。深度嵌套的类型层次结构可能很难遵循、理解和维护。
CA1502:避免过度复杂	此规则通过方法来测量线性独立的路径的数量，该数量是由条件分支的数量和复杂度决定的。
CA1505:避免使用无法维护的代码	类型或方法具有较低的可维护性索引值。如果可维护性指数较低，则表示类型或方法可能难以维护，最好重新进行设计。
CA1506:避免过度类耦合度	此规则通过计算类型或方法包含的唯一类型引用的个数来衡量类耦合。
CA1507：使用 nameof 代替字符串	字符串字面量用作参数，可在其中使用 `nameof` 表达式。
CA1508：避免死条件代码	方法具有在运行时始终计算为 `true` 或 `false` 的条件代码。这会导致条件的 `false` 分支中出现死代码。
CA1509：代码度量配置文件中的条目无效	代码度量规则（如 CA1501、CA1502、CA1505 和 CA1506）提供了具有无效条目的名为 `CodeMetricsConfig.txt` 的配置文件。
CA1700:不要命名“Reserved”枚举值	此规则假定当前不使用名称中包含“reserved”的枚举成员，而是将其作为一个占位符，以在将来的版本中重命名或移除它。重命名或移除成员是一项重大更改。
CA1707:标识符不应包含下划线	按照约定，标识符名称不包含下划线 (_) 字符。该规则将检查命名空间、类型、成员和参数。
CA1708:标识符应以大小写之外的差别进行区分	不能仅通过大小写区分命名空间、类型、成员和参数的标识符，因为针对公共语言运行时的语言不需要区分大小写。
CA1710:标识符应具有正确的后缀	按照约定，扩展某些基类型或实现某些接口的类型的名称，或者由这些类型派生的类型的名称应具有与相应基类型或接口关联的后缀。
CA1711:标识符应采用正确的后缀	按照约定，只有扩展某些基类型或实现某些接口的类型的名称或者从这些类型派生的类型的名称，应该以特定的保留后缀结尾。其他类型名称不应使用这些保留的后缀。
CA1712:不要将类型名用作枚举值的前缀	枚举成员的名称不能使用类型名称作为前缀，因为类型信息将由开发工具提供。
CA1713:事件不应具有 before 或 after 前缀	事件的名称以“Before”或“After”开头。若要命名按特定顺序引发的相关事件，请使用现在时或过去时指示一系列操作中的相对位置。

ID	
CA1714:Flags 枚举应采用复数形式的名称	公共枚举具有 System.FlagsAttribute 特性并且其名称不是以“s”结尾。用 FlagsAttribute 标记的类型具有复数形式的名称，因为该特性指明可以指定多个值。
CA1715:标识符应具有正确的前缀	外部可见的接口的名称不以大写的“I”开头。外部可见的类型或方法上的泛型类型参数的名称不以大写的“T”开头。
CA1716:标识符不应与关键字冲突	某个命名空间名称或类型名称与编程语言中的保留关键字相同。命名空间和类型的标识符不应与针对公共语言运行时的语言所定义的关键字冲突。
CA1717:只有 FlagsAttribute 枚举应采用复数形式的名称	命名约定规定，复数形式的枚举名称表示可以同时指定多个枚举值。
CA1720:标识符不应包含类型名称	外部可见成员中的某个参数的名称包含一个数据类型名称，或者外部可见成员的名称包含一个语言特定的数据类型名称。
CA1721:属性名不应与 get 方法冲突	公共或受保护成员的名称以“Get”开头，且其余部分与公共或受保护属性的名称匹配。“Get”方法和属性的名称应能够明确区分其功能上的差异。
CA1724:类型名不应与命名空间冲突	类型名不应与 .NET 命名空间的名称匹配。与该规则冲突将使库的可用性下降。
CA1725:参数名应与基方法中的声明保持一致	以一致的方式命名重写层次结构中的参数可以提高方法重写的可用性。如果派生方法中的参数名与基声明中的名称不同，可能会导致无法区分出该方法是基方法的重写还是该方法的新重载。
CA1801:检查未使用的参数	方法签名包含一个没有在方法体中使用的参数。
CA1802:在合适的位置使用文本	某个字段被声明为 static 和 read-only（在 Visual Basic 中为 Shared 和 ReadOnly），并使用可在编译时计算的值初始化。因为赋给目标字段的值可在编译时计算，因此请将声明更改为 const（在 Visual Basic 中为 Const）字段，以便在编译时而非运行时计算值。
CA1805：避免进行不必要的初始化	在运行构造函数之前，.NET 运行时将引用类型的所有字段初始化为其默认值。在大多数情况下，将字段显式初始化为其默认值是多余的，这会增加维护成本，并可能会降低性能（例如随着程序集大小的增加）。
CA1806:不要忽略方法结果	创建一个新对象，但从不使用该对象；或者调用会创建并返回一个新字符串的方法，但从不使用这个新字符串；或者 COM 或 P/Invoke 方法返回一个从不使用的 HRESULT 或错误代码。
CA1810:以内联方式初始化引用类型的静态字段	当一个类型声明显式静态构造函数时，实时 (JIT) 编译器会向该类型的每个静态方法和实例构造函数中添加一项检查，以确保之前已调用该静态构造函数。静态构造函数检查会降低性能。
CA1812:避免未实例化的内部类	程序集级别类型的实例不是由程序集中的代码创建的。
CA1813:避免使用非密封特性	.NET 提供用于检索自定义属性的方法。默认情况下，这些方法搜索特性继承层次结构。通过密封特性，将无需搜索继承层次结构，且能够提高性能。
CA1814:与多维数组相比，首选使用交错数组	交错数组是元素为数组的数组。构成元素的数组可以是不同的大小，以减少某些数据集的浪费空间。

ID	
CA1815:重写值类型上的 Equals 和相等运算符	对于值类型，Equals 的继承的实现使用反射库，并比较所有字段的内容。反射需要消耗大量计算资源，可能没有必要比较每一个字段是否相等。如果希望用户对实例进行比较或排序，或者希望用户将实例用作哈希表键，则值类型应实现 Equals。
CA1816:正确调用 GC.SuppressFinalize	作为 Dispose 的实现的某个方法未调用 GC.SuppressFinalize；或者不是 Dispose 的实现的某个方法调用了 GC.SuppressFinalize；或者某个方法调用了 GC.SuppressFinalize 并传递 this（在 Visual Basic 中是 Me）以外的某个值。
CA1819:属性不应返回数组	即使属性是只读的，该属性返回的数组也不是写保护的。若要使数组不会被更改，属性必须返回数组的副本。通常，用户不能理解调用这种属性的负面性能影响。
CA1820:使用字符串长度测试是否有空字符串	使用 String.Length 属性或 String.IsNullOrEmpty 方法比较字符串要比使用 Equals 的速度快得多。
CA1821:移除空终结器	应尽可能避免终结器，因为跟踪对象生存期会产生额外的性能系统开销。空的终结器只会徒增系统开销，没有一点好处。
CA1822:将成员标记为 static	可以将不访问实例数据或不调用实例方法的成员标记为 static（在 Visual Basic 中为 Shared）。在将这些方法标记为 static 之后，编译器将向这些成员发出非虚拟调用站点。这会使性能敏感的代码的性能得到显著提高。
CA1823:避免未使用的私有字段	检测到程序集内有似乎未访问过的私有字段。
CA1824:用 NeutralResourcesLanguageAttribute 标记程序集	NeutralResourcesLanguage 特性通知资源管理器用于显示程序集的非特定区域性资源的语言。这将改进所加载的第一个资源的查找性能，并缩小工作集。
CA1825:避免数组分配长度为零	初始化长度为零的数组将导致不必要的内存分配。相反，请通过调用 Array.Empty 来使用静态分配的空数组实例。内存分配在此方法的所有调用之间共享。
CA1826:使用属性，而不是 Linq Enumerable 方法	对支持等效且更有效的属性的类型使用了 Enumerable LINQ 方法。
CA1827:如果可以使用 Any，请勿使用 Count/LongCount	在使用 Any 方法会更有效的情况下使用了 Count 或 LongCount 方法。
CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync	在使用 AnyAsync 方法会更有效的情况下使用了 CountAsync 或 LongCountAsync 方法。
CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法	对支持等效且更有效的 `Length` 或 `Count` 属性的类型使用了 Count LINQ 方法。
CA1830:在 StringBuilder 上优先使用强类型“追加和插入”方法重载	Append 和 Insert 为除 String 之外的多种类型提供重载。如果可能，首选强类型重载，而非 ToString () 和基于字符串的重载。
CA1831:在合适的情况下，为字符串使用 AsSpan 而不是基于范围的索引器	对字符串使用范围索引器并向 ReadOnlySpan<char> 类型隐式赋值时，将使用方法 Substring 而非 Slice，这会生成字符串请求部分的副本。
CA1832:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 ReadOnlySpan 或 ReadOnlyMemory 部分	对字符串使用范围索引器并向 ReadOnlySpan<T> 或 ReadOnlyMemory<T> 类型隐式赋值时，将使用方法 GetSubArray 而非 Slice，这会生成数组请求部分的副本。

ID	
CA1833:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 Span 或 Memory 部分	对字符串使用范围索引器并向 Span<T> 或 Memory<T> 类型隐式赋值时，将使用方法 GetSubArray 而非 Slice，这会生成数组请求部分的副本。
CA1834:对单字符字符串使用 StringBuilder.Append(char)	StringBuilder 具有将 `char` 用作其参数的 `Append` 重载。优先选择调用 `char` 重载以提高性能。
CA1835：对于“ReadAsync”和“WriteAsync”，首选基于“Memory”的重载	“Stream”有一个将“Memory<byte>”用作第一个参数的“ReadAsync”重载和一个将“ReadOnlyMemory<Byte>”用作第一个参数的“WriteAsync”重载。优先选择调用基于内存的重载，它们更有效。
CA1836：如可用，首选 `IsEmpty` 而不是 `Count`	首选比 `Count`、`Length`、Count<TSource>(IEnumerable<TSource>) 或 LongCount<TSource>(IEnumerable<TSource>) 更有效的 `IsEmpty` 属性，以确定对象是否包含任何项目。
CA1837：使用 `Environment.ProcessId` 而不是 `Process.GetCurrentProcess().Id`	`Environment.ProcessId` 比 `Process.GetCurrentProcess().Id` 更简单、更快速。
CA1838：避免对 P/Invokes 使用 `StringBuilder` 参数	“StringBuilder”的封送处理总是会创建一个本机缓冲区副本，这导致一个封送处理操作出现多次分配。
CA1841：首选字典包含方法	对 `Keys` 或 `Values` 集合调用 `Contains` 通常比对字典本身调用 `ContainsKey` 或 `ContainsValue` 开销更高。
CA1844：对“流”进行子分类时，提供异步方法的基于内存的重写	若要提高性能，请在对“流”进行子分类时重写基于内存的异步方法。然后，在基于内存的方法中实现基于数组的方法。
CA1845：使用基于跨度的“string.Concat”	使用 `AsSpan` 和 `string.Concat` 比使用 `Substring` 和串联运算符更高效。
CA1846：首选 `AsSpan`，次选 `Substring`	`AsSpan` 比 `Substring` 更高效。`Substring` 执行 O(n) 字符串复制，而 `AsSpan` 不会执行此操作且具有固定成本。`AsSpan` 也不执行任何堆分配。
CA1847：对单个字符查找使用 char 文本	搜索单个字符时使用 `string.Contains(char)` 而不是 `string.Contains(string)`。
CA1849：当在异步方法中时，调用异步方法	在已属于异步的方法中，对其他方法的调用应指向其存在的异步版本。
CA1850：首选静态 `HashData` 方法，而非 `ComputeHash`	相比创建并管理 `HashAlgorithm` 实例来调用 `ComputeHash`，使用静态 `HashData` 方法更高效。
CA2000:丢失范围之前释放对象	由于可能发生异常事件，导致对象的终结器无法运行，因此，应显式释放对象，以避免对该对象的所有引用超出范围。
CA2002:不要锁定具有弱标识的对象	当可以跨应用程序域边界直接进行访问对象时，则认为该对象具有弱标识。对于尝试获取对具有弱标识的对象的锁的线程，该线程可能会被其他应用程序域中持有对同一对象的锁的另一线程所阻止。
CA2007：不直接等待任务	异步方法会直接等待 Task。异步方法直接等待 Task 时，延续任务出现在创建任务的同一线程中。此行为可能会降低性能，并且可能会导致 UI 线程发生死锁。请考虑调用 Task.ConfigureAwait(Boolean) 以表示延续任务意图。
CA2008：不要在未传递 TaskScheduler 的情况下创建任务	任务创建或延续操作使用未指定 TaskScheduler 参数的方法重载。

ID	
CA2009：请勿对 ImmutableCollection 值调用 ToImmutableCollection	没有必要在 System.Collections.Immutable 命名空间的不可变集合上调用 `ToImmutable` 方法。
CA2011:请勿在其资源库中分配属性	属性在自身的 set 访问器中被意外赋值。
CA2012:正确使用 ValueTask	从成员调用中返回的 ValueTasks 旨在直接等待。多次尝试使用 ValueTask 或在已知完成之前直接访问其结果可能会导致异常或损坏。忽略此类 ValueTask 可能指示出现功能 Bug，还可能降低性能。
CA2013:请勿将 ReferenceEquals 与值类型结合使用	使用 System.Object.ReferenceEquals 比较值时，如果 objA 和 objB 是值类型，则在将其传递给 ReferenceEquals 方法之前将它们装箱。这意味着，即使 objA 和 objB 都表示值类型的同一个实例，ReferenceEquals 方法也会返回 false。
CA2014：请勿在循环中使用 stackalloc。	仅在当前方法调用结束时，Stackalloc 分配的堆栈空间才会释放。在循环中使用此方法可能导致无限堆栈增长，最终出现堆栈溢出的情况。
CA2015：请勿为派生自 MemoryManager<T> 的类型定义终结器	将终结器添加到派生自 MemoryManager<T> 的类型可能使内存在仍被 Span<T> 使用时得到释放。
CA2016：将 CancellationToken 参数转发到采用一个该参数的方法	将 `CancellationToken` 参数转发给方法来确保操作取消通知得到正确传播，或者在 `CancellationToken.None` 中显式传递，以指示有意不传播令牌。
CA2018：`Buffer.BlockCopy` 的 `count` 参数应指定要复制的字节数	使用 `Buffer.BlockCopy` 时，`count` 参数指定要复制的字节数。应仅对元素大小正好为一个字节的数组将 `Array.Length` 用于 `count` 参数。`byte`、`sbyte` 和 `bool` 数组具有大小为一个字节的元素。
CA2100:检查 SQL 查询是否存在安全漏洞	一个方法使用按该方法的字符串参数生成的字符串设置 System.Data.IDbCommand.CommandText 属性。此规则假定字符串参数中包含用户输入。基于用户输入生成的 SQL 命令字符串易于受到 SQL 注入式攻击。
CA2101：指定对 P/Invoke 字符串参数进行封送处理	某平台调用成员允许部分受信任的调用方，具有一个字符串参数，并且不显式封送该字符串。这可能导致潜在的安全漏洞。
CA2109:检查可见的事件处理程序	检测到公共事件处理方法或受保护事件处理方法。除非绝对必要，否则不应公开事件处理方法。
CA2119:密封满足私有接口的方法	可继承的公共类型为 internal（在 Visual Basic 中为 Friend）接口提供可重写的方法实现。若要修复与此规则的冲突，请禁止方法在程序集外重写。
CA2153：避免处理损坏状态异常	损坏状态异常 (CSE) 指示进程中存在内存损坏。如果攻击者可以将攻击放置到损坏的内存区域，则捕获它们（而非允许进程崩溃）可能导致安全漏洞。
CA2200:再次引发以保留堆栈详细信息	再次引发某个异常，在 throw 语句中显式指定了该异常。如果通过在 throw 语句中指定异常来重新引发该异常，则引发该异常的原始方法与当前方法之间的方法调用的列表将丢失。
CA2201:不要引发保留的异常类型	这使得很难检测和调试原始错误。
CA2207:以内联方式初始化值类型的静态字段	某值类型声明了显式静态构造函数。要修复与该规则的冲突，请在声明它时初始化所有静态数据并移除静态构造函数。

ID	
CA2208:正确实例化参数异常	调用了异常类型 ArgumentException 或其派生类型的默认(无参数)构造函数，或者向异常类型 ArgumentException 或其派生类型的参数化构造函数传递了错误的字符串参数。
CA2211:非常量字段不应是可见的	不是常数也不是只读字段的静态字段不是线程安全的。必须严格控制对这类字段的访问，并需要高级编程技术来同步对类对象的访问。
CA2213:应释放可释放的字段	实现 System.IDisposable 的类型声明了同样实现 IDisposable 的类型的字段。字段的 Dispose 方法不由声明类型的 Dispose 方法调用。
CA2214:不要在构造函数中调用可重写的方法	构造函数调用虚方法时，可能尚未执行调用该方法的实例的构造函数。
CA2215:Dispose 方法应调用基类释放	如果类型继承自可释放类型，则必须从它自己的 Dispose 方法中调用基类型的 Dispose 方法。
CA2216:可释放类型应声明终结器	实现 System.IDisposable 并包含建议使用非托管资源的字段的类型未实现 Object.Finalize 所描述的终结器。
CA2218:重写 Equals 时重写 GetHashCode	公共类型重写 System.Object.Equals，但不重写 System.Object.GetHashCode。
CA2217:不要使用 FlagsAttribute 标记枚举	外部可见的枚举使用 FlagsAttribute 标记，并且它包含的一个或多个值不是 2 的幂或不是为该枚举定义的其他值的组合。
CA2219:在异常子句中不引发异常	如果在 finally 或 fault 子句中引发异常，新异常将隐藏活动异常。当在 filter 子句中引发异常时，运行时会在不提示的情况下捕捉异常。这使得很难检测和调试原始错误。
CA2224:重载相等运算符时重写 Equals 方法	公共类型会实现相等运算符，但不重写 System.Object.Equals。
CA2225:运算符重载具有命名的备用项	检测到运算符重载，但未找到预期的指定备用方法。命名的备用成员提供了对与运算符相同的功能的访问，它提供给开发人员，在用不支持重载运算符的语言进行编程时使用。
CA2226:运算符应有对称重载	某个类型实现了相等运算符或不等运算符，却未实现相反运算符。
CA2227:集合属性应为只读	使用可写的集合属性，用户可以将该集合替换为不同的集合。只读属性禁止替换该集合，但仍允许设置单个成员。
CA2229:实现序列化构造函数	要修复与该规则的冲突，请实现序列化构造函数。对于密封类，请使构造函数成为私有；否则，请使构造函数成为受保护。
CA2231:重写 ValueType.Equals 时应重载相等运算符	值类型重写 Object.Equals，但未实现相等运算符。
CA2234:传递 System.Uri 对象，而不传递字符串	调用了带有一个字符串参数的方法，该参数的名称中包含“uri”、“URI”、“urn”、“URN”、“url”或“URL”。此方法的声明类型包含具有 System.Uri 参数的对应方法重载。
CA2235:标记所有不可序列化的字段	在可以序列化的类型中声明了类型不可序列化的实例字段。
CA2237:用 SerializableAttribute 标记 ISerializable 类型	若要被公共语言运行时识别为可序列化，类型必须用 SerializableAttribute 特性标记，即使该类型通过实现 ISerializable 接口使用了自定义的序列化例程也是如此。
CA2241:为格式化方法提供正确的参数	传递给 System.String.Format 的 format 自变量不包含对应于每个对象自变量的格式项，反之亦然。

ID	
CA2242:正确测试 NaN	此表达式对照 Single.Nan 或 Double.Nan 测试某个值。使用 Single.IsNan(Single) 或 Double.IsNan(Double) 测试该值。
CA2243:特性字符串文本应正确分析	特性的字符串文本参数不能正确解析为 URL、GUID 或版本。
CA2244:不要复制已索引的元素初始值设定项	对象初始值设定项有多个具有相同常量索引的索引元素初始值设定项。除最后一个初始值设定项之外，其余都是冗余的。
CA2245:请勿将属性分配给其自身	属性意外赋值给了其自身。
CA2246:请勿在同一语句中分配符号及其成员	不建议在同一语句中分配符号及其成员(即字段或属性)。目前尚不清楚成员访问是打算在赋值之前使用符号的旧值还是打算使用此语句中赋值的新值。
CA2247：传递给 TaskCompletionSource 构造函数的参数应为 TaskCreationOptions 枚举，而不是 TaskContinuationOptions 枚举。	TaskCompletionSource 既有采用控制基础任务的 TaskCreationOptions 的构造函数，也有采用任务中存储的对象状态的构造函数。如果意外传递 TaskContinuationOptions 而不是 TaskCreationOptions，则将导致调用将选项视为状态。
CA2248：向 Enum.HasFlag 提供正确的 enum 实参	作为实参传递给 `HasFlag` 方法调用的枚举类型不同于调用枚举类型。
CA2249：请考虑使用 String.Contains 而不是 String.IndexOf	对 `string.IndexOf` 的调用(其结果用于检查是否存在子字符串)可以用 `string.Contains` 替换。
CA2250：使用 `ThrowIfCancellationRequested`	`ThrowIfCancellationRequested` 自动检查令牌是否已取消，如果已取消，则引发 `OperationCanceledException`。
CA2251：使用 `String.Equals` 代替 `String.Compare`	与其将 `String.Compare` 的结果与零进行比较，不如使用 `String.Equals`，这样更清晰且速度可能更快。
CA2252：选择预览功能	使用预览 API 之前选择预览功能。
CA2300：请勿使用不安全的反序列化程序 BinaryFormatte	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2301：在未先设置 BinaryFormatter.Binder 的情况下，请不要调用 BinaryFormatter.Deserialize	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2302：在调用 BinaryFormatter.Deserialize 之前，确保设置 BinaryFormatter.Binder	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2305：请勿使用不安全的反序列化程序 LosFormatter	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2310：请勿使用不安全的反序列化程序 NetDataContractSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2311：在未先设置 NetDataContractSerializer.Binder 的情况下，请不要反序列化	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2312：确保在反序列化之前设置 NetDataContractSerializer.Binder	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。

ID	
CA2315：请勿使用不安全的反序列化程序 ObjectStateFormatter	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2321：请勿使用 SimpleTypeResolver 对 JavaScriptSerializer 进行反序列化	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2322：确保在反序列化之前没有使用 SimpleTypeResolver 初始化 JavaScriptSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2326：请勿使用 None 以外的 TypeNameHandling 值	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2327：不要使用不安全的 JsonSerializerSettings	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2328：确保 JsonSerializerSettings 是安全的	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2329：不要使用不安全的配置反序列化 JsonSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2330：在反序列化时确保 JsonSerializer 具有安全配置	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2350:确保 DataTable.ReadXml() 的输入受信任	对包含不受信任的输入的 DataTable 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。
CA2351:确保 DataSet.ReadXml() 的输入受信任	对包含不受信任的输入的 DataSet 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。
CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击	带有 SerializableAttribute 标记的类或结构包含 DataSet 或 DataTable 字段或属性，但不具有 GeneratedCodeAttribute。
CA2353:可序列化类型中的不安全 DataSet 或 DataTable	使用 XML 序列化特性或数据协定特性进行了标记的类或结构包含 DataSet 或 DataTable 字段或属性。
CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击	当使用序列化的 System.Runtime.Serialization.IFormatter 进行反序列化时，且强制转换的类型的对象图可能包含 DataSet 或 DataTable 时。
CA2355:反序列化对象图中的不安全 DataSet 或 DataTable	当强制转换的或指定的类型的对象图可能包含 DataSet 或 DataTable 类时，进行反序列化。
CA2356：Web 反序列化的对象图中不安全的 DataSet 或 DataTable	带有 System.Web.Services.WebMethodAttribute 或 System.ServiceModel.OperationContractAttribute 的方法具有可能引用 DataSet 或 DataTable 的参数。
CA2361：请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用	对包含不受信任的输入的 DataSet 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

ID	
CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击	当反序列化具有 BinaryFormatter 的不受信任的输入且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可能创建执行远程代码执行攻击的恶意有效负载。
CA3001：查看 SQL 注入漏洞的代码	使用不受信任的输入和 SQL 命令时，请注意防范 SQL 注入攻击。SQL 注入攻击可以执行恶意的 SQL 命令，从而降低应用程序的安全性和完整性。
CA3002：查看 XSS 漏洞的代码	在处理来自 Web 请求的不受信任的输入时，请注意防范跨站脚本 (XSS) 攻击。XSS 攻击会将不受信任的输入注入原始 HTML 输出，使攻击者可以执行恶意脚本或恶意修改网页中的内容。
CA3003:查看文件路径注入漏洞的代码	在处理来自 Web 请求的不受信任的输入时，请谨慎使用用户控制的输入指定文件路径。
CA3004：查看信息泄露漏洞的代码	泄漏异常信息可让攻击者深入了解应用程序的内部机制，从而帮助攻击者找到其他漏洞并利用这些漏洞。
CA3006：查看进程命令注入漏洞的代码	处理不受信任的输入时，请注意防范命令注入攻击。命令注入攻击可在基础操作系统上执行恶意命令，从而降低服务器的安全和完整性。
CA3007：查看公开重定向漏洞的代码	处理不受信任的输入时，请注意防范开放重定向漏洞。攻击者可以利用开放重定向漏洞，使用你的网站提供合法 URL 的外观，但将毫不知情的访客重定向到钓鱼网页或其他恶意网页。
CA3008：查看 XPath 注入漏洞的代码	处理不受信任的输入时，请注意防范 XPath 注入攻击。使用不受信任的输入构造 XPath 查询可能会允许攻击者恶意控制查询，使其返回一个意外的结果，并可能泄漏查询的 XML 的内容。
CA3009：查看 XML 注入漏洞的代码	处理不受信任的输入时，请注意防范 XML 注入攻击。
CA3010：查看 XAML 注入漏洞的代码	处理不受信任的输入时，请注意防范 XAML 注入攻击。XAML 是一种直接表示对象实例化和执行的标记语言。这意味着 XAML 中创建的元素可以与系统资源（例如，网络访问和文件系统 IO）交互。
CA3011：查看 DLL 注入漏洞的代码	处理不受信任的输入时，请谨慎加载不受信任的代码。如果你的 Web 应用加载不受信任的代码，攻击者可能能够将恶意 DLL 注入到你的进程中，并执行恶意代码。
CA3012：查看正则表达式注入漏洞的代码	处理不受信任的输入时，请注意防范正则表达式注入攻击。攻击者可以使用正则表达式注入恶意修改正则表达式，让正则表达式匹配非预期结果，或者让正则表达式占用过多 CPU，从而形成拒绝服务攻击。
CA3061：请勿按 URL 添加架构	请勿使用不安全的“添加”方法重载，因为这可能会导致危险的外部引用。
CA3075:不安全的 DTD 处理	如果使用不安全的 DTDProcessing 实例或引用外部实体源，分析器可能会接受不受信任的输入并将敏感信息泄露给攻击者。
CA3076:不安全的 XSLT 脚本执行	如果在 .NET 应用程序中不安全地执行可扩展样式表语言转换 (XSLT)，处理器可能会解析不受信任的 URI 引用，这种引用会把敏感信息泄露给攻击者，从而导致拒绝服务和跨站点攻击。

ID	
CA3077:API 设计、XML 文档和 XML 文本读取器中的不安全处理	当设计派生自 XMLDocument 和 XMLTextReader 的 API 时，请注意 DtdProcessing。当引用或解析外部实体源或设置 XML 中的不安全值时，使用不安全的 DTDProcessing 实例可能会导致信息泄露。
CA3147:使用 ValidateAntiForgeryToken 标记谓词处理程序	设计 ASP.NET MVC 控制器时，请注意防范跨网站请求伪造攻击。跨网站请求伪造攻击可将来自经过身份验证的用户的恶意请求发送到 ASP.NET MVC 控制器。
CA5350:请勿使用弱加密算法	出于多种原因，现今使用弱加密算法和哈希函数，但不应使用它们来保证保密性或它们所保护的数据的完整性。当此规则在代码中找到 TripleDES、SHA1、或 RIPEMD160 算法时，此规则将触发。
CA5351 不使用损坏的加密算法	损坏的加密算法不安全，强烈建议不要使用。当此规则在代码中找到 MD5 哈希算法，或者 DES 或 RC2 加密算法时，此规则将触发。
CA5358：请勿使用不安全的密码模式	请勿使用不安全的密码模式
CA5359：请勿禁用证书验证	证书有助于对服务器的身份进行验证。客户端应验证服务器证书，确保将请求发送到目标服务器。如果 ServerCertificateValidationCallback 始终返回 `true`，那么任何证书都将通过验证。
CA5360：在反序列化中不要调用危险的方法	不安全的反序列化是一种漏洞。当使用不受信任的数据来损害应用程序的逻辑，造成拒绝服务 (DoS) 攻击，或甚至在反序列化时任意执行代码，就会出现该漏洞。应用程序对受其控制的不受信任数据进行反序列化时，恶意用户很可能会滥用这些反序列化功能。具体来说，就是在反序列化过程中调用危险方法。如果攻击者成功执行不安全的反序列化攻击，就能实施更多攻击，如 DoS 攻击、绕过身份验证和执行远程代码。
CA5361：不禁用强加密的 SChannel 使用	将 `Switch.System.Net.DontEnableSchUseStrongCrypto` 设置为 `true` 会减弱传出的传输层安全性连接中使用的加密性。较弱的加密性会泄露应用程序与服务器之间通信的机密性，使攻击者更易于窃听敏感数据。
CA5362：反序列化对象图中存在潜在引用循环	反序列化不受信任的数据时，处理反序列化对象图的任何代码都需要在处理引用循环时不进入无限循环。这包括反序列化回叫中的一部分代码和在反序列化完成后处理对象图的代码。否则攻击者可能会利用带有包含引用循环的恶意数据执行拒绝服务攻击。
CA5363：请勿禁用请求验证	请求验证是 ASP.NET 中的一项功能，可检查 HTTP 请求并确定这些请求是否包含可能导致跨站点脚本编写等注入攻击的潜在危险内容。
CA5364：不使用已弃用的安全协议	传输层安全性 (TLS) 通常使用超文本传输协议安全 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。
CA5365：请勿禁用 HTTP 头检查	通过 HTTP 标头检查，可对在响应头中找到的回车符和换行符 (\r 和 \n) 进行编码。此编码有助于避免注入攻击，这些注入攻击会攻击对标头包含的不受信数据进行回显的应用程序。
CA5366：将 XmlReader 用于数据集读取 XML	使用 DataSet 读取包含不受信数据的 XML，可能会加载危险的外部引用，应使用具有安全解析程序或禁用了 DTD 处理的 XmlReader 来限制这种行为。

ID	
CA5367：请勿序列化具有 Pointer 字段的类型	此规则检查是否存在带有指针字段或属性的可序列化类。无法进行序列化的成员可能是指针，例如使用 NonSerializedAttribute 进行标记的静态成员或字段。
CA5368：针对派生自 Page 的类设置 ViewStateUserKey	设置 ViewStateUserKey 属性有助于防止对应用程序的攻击，方法是允许你为各个用户的视图状态变量分配标识符，这样攻击者就无法使用变量生成攻击。否则会出现“跨网站请求伪造”漏洞。
CA5369：将 XmlReader 用于反序列化	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用，应使用具有安全解析程序或禁用了 DTD 和 XML 内联架构处理的 XmlReader 来限制这种行为。
CA5370：将 XmlReader 用于验证读取器	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用。此危险的加载行为可使用具有安全解析程序或者禁用了 DTD 和 XML 内联架构处理的 XmlReader 来进行限制。
CA5371：将 XmlReader 用于架构读取	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用。请使用具有安全解析程序或者禁用了 DTD 和 XML 内联架构处理的 XmlReader 对其进行限制。
CA5372：将 XmlReader 用于 XPathDocument	处理来自不受信任的数据的 XML 时可能会加载危险的外部引用，可使用具有安全解析程序或禁用了 DTD 处理的 XmlReader 对其进行限制。
CA5373：请勿使用已过时的密钥派生功能	此规则会检测对弱密钥派生方法 System.Security.Cryptography.PasswordDeriveBytes 和 `Rfc2898DeriveBytes.CryptDeriveKey` 的调用。System.Security.Cryptography.PasswordDeriveBytes 使用了弱算法 PBKDF1。
CA5374：请勿使用 XslTransform	此规则检查 System.Xml.Xsl.XslTransform 是否在代码中进行了实例化。System.Xml.Xsl.XslTransform 现已过时且不应使用。
CA5375：请勿使用帐户共享访问签名	帐户 SAS 可以委派对 blob 容器、表、队列和文件共享执行读取、写入和删除操作的访问权限，而这是服务 SAS 所不允许的。但是它不支持容器级别的策略，并且其灵活性和对授予的权限的控制力更低。一旦恶意用户获取它后，存储帐户的信息很容易泄露。
CA5376：使用 SharedAccessProtocol HttpsOnly	SAS 是无法在 HTTP 上以纯文本形式传输的敏感数据。
CA5377：使用容器级别访问策略	容器级别的访问策略可以随时修改或撤销。它具有更高的灵活性，对授予的权限的控制力更强。
CA5378：不禁用 ServicePointManagerSecurityProtocols	将 `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityPro` 设置为 `true` 会将 Windows Communication Framework (WCF) 的传输层安全性 (TLS) 连接限制为使用 TLS 1.0。该版本的 TLS 将被弃用。
CA5379：请勿使用弱密钥派生功能算法	Rfc2898DeriveBytes 类默认使用 SHA1 算法。应指定在 SHA256 或更高版本的构造函数的某些重载中使用哈希算法。请注意，HashAlgorithm 属性只具有 `get` 访问器，而没有 `overriden` 修饰符。
CA5380：请勿将证书添加到根存储中	此规则会对将证书添加到“受信任的根证书颁发机构”证书存储的代码进行检测。默认情况下，“受信任的根证书颁发机构”证书存储配置有一组符合 Microsoft 根证书计划要求的公共 CA。

ID	
CA5381：请确保证书未添加到根存储中	此规则会对可能将证书添加到“受信任的根证书颁发机构”证书存储的代码进行检测。默认情况下，“受信任的根证书颁发机构”证书存储配置有一组符合 Microsoft 根证书计划要求的公共证书颁发机构 (CA)。
CA5382：在 ASP.NET Core 中使用安全 Cookie	HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用安全套接字层 (SSL) 进行传输。
CA5383：确保在 ASP.NET Core 中使用安全 Cookie	HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用安全套接字层 (SSL) 进行传输。
CA5384：不使用数字签名算法 (DSA)	DSA 是一种弱非对称加密算法。
CA5385：设置具有足够密钥大小的 Rivest–Shamir–Adleman (RSA) 算法	小于 2048 位的 RSA 密钥更容易受到暴力攻击。
CA5386：避免对 SecurityProtocolType 值进行硬编码	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。TLS 1.2 和 TLS 1.3 将来可能也会弃用。要确保应用程序的安全性，请避免对协议版本进行硬编码，并且至少以 .NET Framework v4.7.1 为目标。
CA5387：请勿使用迭代计数不足的弱密钥派生功能	此规则检查加密密钥是否由迭代计数小于 100,000 的 Rfc2898DeriveBytes 生成。迭代计数较高有助于缓解尝试猜测已生成的加密密钥的字典攻击。
CA5388：使用弱密钥派生功能时，请确保迭代计数足够大	此规则检查加密密钥是否由迭代计数可能小于 100,000 的 Rfc2898DeriveBytes 生成。迭代计数较高有助于缓解尝试猜测已生成的加密密钥的字典攻击。
CA5389：请勿将存档项的路径添加到目标文件系统路径中	文件路径可以是相对的，并且可能导致文件系统访问预期文件系统目标路径以外的内容，从而导致攻击者通过“布局和等待”技术恶意更改配置和执行远程代码。
CA5390：请勿硬编码加密密钥	要成功使用对称算法，密钥必须只有发送方和接收方知道。如果密钥是硬编码的，就容易被发现。即使使用编译的二进制文件，恶意用户也容易将其提取出来。私钥泄露后，密码文本可直接被解密并且不再受保护。
CA5391：在 ASP.NET Core MVC 控制器中使用防伪造令牌	处理 `POST`、`PUT`、`PATCH` 或 `DELETE` 请求而不验证防伪造令牌可能易受到跨网站请求伪造攻击。跨网站请求伪造攻击可将经过身份验证的用户的恶意请求发送到 ASP.NET Core MVC 控制器。
CA5392：对 P/Invoke 使用 DefaultDllImportSearchPaths 特性	默认情况下，使用 DllImportAttribute 的 P/Invoke 函数会探测大量目录，包括要加载的库的当前工作目录。这对于某些应用程序来说是一个安全隐患，会导致 DLL 劫持。
CA5393：请勿使用不安全的 DllImportSearchPath 值	默认的 DLL 搜索目录和程序集目录中可能存在恶意 DLL。或者根据应用程序运行的位置，应用程序的目录中可能存在恶意 DLL。
CA5394：请勿使用不安全的随机性	如果使用加密较弱的伪随机数生成器，攻击者可以预测将要生成的安全敏感值。
CA5395：缺少操作方法的 HttpVerb 特性	创建、编辑或以其它方式修改数据等所有操作方法都需要使用防伪特性来保护，以避免受跨网站请求伪造攻击的影响。执行 GET 操作应是没有副作用且不会修改持久数据的安全操作。

ID	
CA5396：将 HttpCookie 的 HttpOnly 设置为 true	请确保将安全敏感的 HTTP Cookie 标记为 HttpOnly，这是一个深度防御措施。这表明 Web 浏览器应禁止脚本访问 Cookie。注入恶意脚本是常见的窃取 Cookie 的方式。
CA5397：不使用已弃用的 SslProtocols 值	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。
CA5398：避免硬编码的 SslProtocols 值	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。将来可能也会弃用 TLS 1.2 和 TLS 1.3。要确保应用程序的安全性，请避免对协议版本进行硬编码。
CA5399：绝对禁用 HttpClient 证书吊销列表检查	撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。
CA5400：确保未禁用 HttpClient 证书吊销列表检查	撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。
CA5401：不要将 CreateEncryptor 与非默认 IV 结合使用	对称加密应始终使用非可重复的初始化向量，以防止字典攻击。
CA5402：将 CreateEncryptor 与默认 IV 结合使用	对称加密应始终使用非可重复的初始化向量，以防止字典攻击。
CA5403：请勿硬编码证书	X509Certificate 或 X509Certificate2 构造函数的 `data` 或 `rawData` 参数是硬编码的。
CA5404：不要禁用令牌验证检查	用于控制令牌验证的 TokenValidationParameters 属性不应设置为 `false`。
CA5405：不要始终跳过委托中的令牌验证	分配给 AudienceValidator 或 LifetimeValidator 的回调始终返回 `true`。
IL3000 当发布为单个文件时，避免访问程序集文件路径	当发布为单个文件时，避免访问程序集文件路径。
IL3001 当发布为单个文件时，避免访问程序集文件路径	当发布为单个文件时，避免访问程序集文件路径。
IL3002 当发布为单个文件时，避免调用使用“RequiresAssemblyFilesAttribute”批注的成员	当发布为单个文件时，避免调用使用“RequiresAssemblyFilesAttribute”批注的成员

图例

下表显示了为参考文档中每个规则提供的信息类型。

类型	规则的 TypeName。
■ ID	规则的唯一标识符。RuleId 和类别用于源代码中禁止显示警告。
■	规则的类别，例如安全性。

[	[[
▬▬▬	规则冲突的修复是否是一项重大更改。重大更改意味着，在导致冲突的目标上具有依赖关系的程序集不会使用新修复的版本重新编译，或者可能会由于此更改在运行时失败。当具有多个修复可用且至少有一个修复是一项重大更改，有一个不是时，将同时指定“重大”和“非重大”。
原因	导致规则生成警告的特定托管代码。
说明	讨论警告背后的问题。
如何解决冲突	说明如何更改源代码以满足规则并防止它生成警告。
何时禁止显示警告	描述何时可以安全地禁止显示此规则警告。
示例代码	规则冲突示例和满足该规则的已更正示例。
相关规则	相关规则。

设计规则

2021/11/16 •

设计规则**支持遵从** .NET Framework 设计准则。

本节内容

"	"
CA1000:不要在泛型类型中声明静态成员	调用泛型类型的静态成员时，必须指定该类型的类型参数。当调用不支持推理的泛型实例成员时，必须指定该成员的类型参数。在上述两种情况下，用于指定类型自变量的语法不同，但很容易混淆。
CA1001:具有可释放字段的类型应该是可释放的	某个类声明并实现 System.IDisposable 类型的实例字段，但该类不实现 IDisposable。声明 IDisposable 字段的类间接拥有非托管资源，并且应该实现 IDisposable 接口。
CA1002:不要公开泛型列表	Collections.Generic.List<(Of <(T>)>) 是针对性能(而非继承)设计的泛型集合。因此，List 不包含任何虚拟成员。应改为公开针对继承设计的泛型集合。
CA1003:使用泛型事件处理程序实例	某个类型包含的委托返回 void，该委托的签名包含两个参数(第一个参数是对象，第二个参数是可以分配给 EventArgs 的类型)，而且包含程序集针对的是 .NET Framework 2.0。
CA1005:避免泛型类型的参数过多	泛型类型包含的类型参数越多，越难以知道并记住每个类型参数各代表什么。它通常有一个类型参数，如在 List<T> 中，而在某些情况下有两个类型参数，如在 Dictionary<TKey, TValue> 中。但是，如果存在两个以上的类型参数，则大多数用户都会感到过于困难。
CA1008:枚举应具有零值	像其他值类型一样，未初始化枚举的默认值为零。无标志特性的枚举应通过使用零值来定义成员，这样默认值即为该枚举的有效值。如果应用了 FlagsAttribute 特性的枚举定义值为零成员，则该成员的名称应为“None”，以指示枚举中尚未设置值。
CA1010:集合应实现泛型接口	若要扩大集合的用途，应实现某个泛型集合接口。然后，可以使用该集合来填充泛型集合类型。
CA1012:抽象类型不应具有构造函数	抽象类型的构造函数只能由派生类型调用。由于公共构造函数用于创建类型的实例，但无法为抽象类型创建实例，因此具有公共构造函数的抽象类在设计上是错误的。
CA1014:用 CLSCompliantAttribute 标记程序集	公共语言规范 (CLS) 定义了程序集在跨编程语言使用时必须符合的命名限制、数据类型和规则。好的设计要求所有程序集用 CLSCompliantAttribute 显式指示 CLS 合规性。如果程序集没有此特性，则该程序集即不合规。

CA1016:用 AssemblyVersionAttribute 标记程序集	.NET 使用版本号唯一地标识程序集，并绑定到具有强名称的程序集中的类型。版本号与版本和发行者策略一起使用。默认情况下，仅使用用于生成应用程序的程序集版本运行应用程序。
CA1017:用 ComVisibleAttribute 标记程序集	ComVisibleAttribute 决定 COM 客户端如何访问托管代码。合理的设计指出程序集将显式指示 COM 可见性。可以设置整个程序集的 COM 可见性，然后重写各个类型和类型成员的 COM 可见性。如果此特性不存在，则程序集的内容对 COM 客户端可见。
CA1018:用 AttributeUsageAttribute 标记特性	当定义自定义特性时，用 AttributeUsageAttribute 标记该特性，以指示源代码中可以应用自定义特性的位置。特性的含义和预定用法将决定它在代码中的有效位置。
CA1019:定义特性参数的访问器	特性可以定义强制自变量，在对目标应用该特性时必须指定这些自变量。这些实参也称为位置实参，因为它们将作为位置形参提供给特性构造函数。对于每一个强制变量，特性还必须提供一个相应的只读属性，以便可以在执行时检索该变量的值。特性还可以定义可选实参，可选实参也称为命名实参。这些变量按名称提供给特性构造函数，并且必须具有相应的读/写属性。
CA1021:避免使用 out 参数	通过引用(使用 out 或 ref)传递类型要求具有使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。另外，out 和 ref 参数之间的差异没有得到广泛了解。
CA1024:在适用处使用属性	公共或受保护方法的名称以“Get”开头，没有采用任何参数或返回的值不是数组。该方法可能很适于成为属性。
CA1027:用 FlagsAttribute 标记枚举	枚举是一种值类型，它定义一组相关的已命名常数。如果可以按照有意义的方式组合一个枚举的已命名常数，则对该枚举应用 FlagsAttribute。
CA1028:枚举存储应为 Int32	枚举是一种值类型，它定义一组相关的已命名常数。默认情况下，System.Int32 数据类型用于存储常量值。虽然你可以更改此基础类型，但对于大多数情况，既不需要，也不建议你这样做。
CA1030:在适用处使用事件	该规则检测名称通常用于事件的方法。如果为响应明确定义的状态更改而调用一个方法，则应由事件处理程序调用该方法。调用该方法的对象应引发事件而不是直接调用该方法。
CA1031:不要捕捉一般异常类型	不应捕捉一般异常。捕捉更具体的异常，或者在执行 catch 块中的最后一条语句时重新引发一般异常。
CA1032:实现标准异常构造函数	如果不能提供完整的构造函数集，要正确处理异常将变得比较困难。
CA1033:接口方法应可由子类型调用	未密封的外部可见类型提供了显式实现公共接口的方法，但没有提供具有相同名称的其他外部可见方法。
CA1034:嵌套类型不应是可见的	嵌套类型是在另一个类型的范围中声明的类型。嵌套类型用于封装包含类型的私有实现详细信息。如果用于此用途，则嵌套类型不应是外部可见的。

CA1036:重写可比较类型中的方法	公共或受保护类型实现 System.IComparable 接口。它不重写 Object.Equals，也不重载表示相等、不等、小于或大于的语言特定运算符。
CA1040:避免使用空接口	接口定义提供某个行为或使用协定的成员。接口所描述的功能可以被任何类型采用，而不管该类型出现在继承层次结构中的哪个位置。类型通过实现接口的成员来实现接口。空接口无法定义任何成员；因此，它无法定义可以实现的协定。
CA1041:提供 ObsoleteAttribute 消息	用未指定其 ObsoleteAttribute.Message 属性的 System.ObsoleteAttribute 特性来标记类型或成员。当编译使用 ObsoleteAttribute 标记的类型或成员时，将显示该特性的 Message 属性，这为用户提供有关过时的类型或成员的信息。
CA1043:将整型或字符串参数用于索引器	索引器（即索引属性）应将整型或字符串类型用于索引。这些类型一般用于为数据结构编制索引，并且提高库的可用性。应仅限于在设计时无法指定特定整型或字符串类型的情况下使用 Object 类型。
CA1044:属性不应是只写的	虽然可以接受且经常需要使用只读属性，但设计准则禁止使用只写属性。这是因为允许用户设置值但又禁止该用户查看这个值不能提供任何安全性。而且，如果没有读访问，将无法查看共享对象的状态，使其用处受到限制。
CA1045:不要通过引用来传递类型	通过引用（使用 out 或 ref）传递类型要求具有使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。为一般用户进行设计的库架构师不应指望用户能熟练运用 out 或 ref 参数。
CA1046:不要对引用类型重载相等运算符	对于引用类型，相等运算符的默认实现几乎始终是正确的。默认情况下，仅当两个引用指向同一对象时，它们才相等。
CA1047:不要在密封类型中声明受保护的成员	类型声明受保护的成员，使继承类型可以访问或重写该成员。按照定义，不能继承密封类型，这表示不能调用密封类型上的受保护方法。
CA1050:在命名空间中声明类型	应在命名空间内声明类型以避免名称冲突，并作为一种在对象层次结构中组织相关类型的方式。
CA1051:不要声明可见实例字段	字段的主要用途应是作为实现的详细信息。字段应为 private 或 internal，并应通过使用属性公开这些字段。
CA1052:应密封静态容器类型	公共或受保护类型仅包含静态成员，而且没有用 sealed (C#) 或 NotInheritable (Visual Basic) 修饰符声明该类型。应使用 sealed 修饰符标记不希望被继承的类型，以免将其用作基类型。
CA1053:静态容器类型不应具有构造函数	公共或嵌套公共类型只声明了静态成员，但具有公共或受保护的默认构造函数。由于调用静态成员不需要类型的示例，因此没必要使用构造函数。为安全起见，字符串重载应使用字符串自变量调用统一资源标识符 (URI) 重载。
CA1054:URI 参数不应为字符串	如果某方法采用 URI 的字符串表示形式，则应提供采用 URI 类的实例的相应重载，该重载以安全的方式提供这些服务。

“	“
CA1055:URI 返回值不应是字符串	此规则假定该方法返回 URI。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。
CA1056:URI 属性不应是字符串	此规则假定属性表示 URI。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。
CA1058:类型不应扩展某些基类型	外部可见的类型扩展某些基类型。请使用某个备选项。
CA1060：将 P/Invoke 移动到 NativeMethods 类	平台调用方法（例如标以 System.Runtime.InteropServices.DllImportAttribute 的方法或在 Visual Basic 中使用 Declare 关键字定义的方法）访问非托管代码。这些方法应属于 NativeMethods、SafeNativeMethods 或 UnsafeNativeMethods 类。
CA1061:不要隐藏基类方法	如果派生方法的参数签名只是在类型方面有所不同，而且与基方法的参数签名中的对应类型相比，这些类型的派生方式更弱，则基类型中的方法由派生类型中的同名方法隐藏。
CA1062:验证公共方法的参数	对于传递给外部可见方法的所有引用自变量，都应检查其是否为 null。
CA1063:正确实现 IDisposable	所有的 IDisposable 类型都应当正确实现 Dispose 模式。
CA1064:异常应该是公共的	内部异常仅在其自己的内部范围内可见。当异常超出内部范围后，只能使用基异常来捕获该异常。如果内部异常继承自 System.Exception、System.SystemException 或 System.ApplicationException，则外部代码将没有足够的信息来了解如何处理该异常。
CA1065:不要在意外的位置引发异常	不应引发异常的方法引发了异常。
CA1066：重写 Equals 时实现 IEquatable	值类型替代 Equals 方法，但不实现 IEquatable<T>。
CA1067：实现 IEquatable 时重写 Equals	类型实现 IEquatable<T>，但不替代 Equals 方法。
CA1068:CancellationToken 参数必须最后出现	方法具有 CancellationToken 参数，但它不是最后一个参数。
CA1069:枚举不得具有重复值	枚举具有多个成员，这些成员显式分配有相同常数值。
CA1070:不要将事件字段声明为“虚拟”	类字段事件被声明为“虚拟”。

CA1000:不要在泛型类型中声明静态成员

2021/11/16 •

	[
■ ID	CA1000
■	设计
■■■■■	重大

原因

泛型类型包含 `static`（在 Visual Basic 中为 `Shared`）成员。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

调用泛型类型的 `static` 成员时，必须指定该类型的类型参数。当调用不支持推理的泛型实例成员时，必须指定该成员的类型参数。在上述两种情况下，用于指定类型参数的语法不同且容易混淆，如以下调用所示：

```
' Shared method in a generic type.
GenericType(Of Integer).SharedMethod()

' Generic instance method that does not support inference.
someObject.GenericMethod(Of Integer)()
```

```
// Static method in a generic type.
GenericType<int>.StaticMethod();

// Generic instance method that does not support inference.
someObject.GenericMethod<int>();
```

通常，应避免前两个声明，以便在调用成员时不必指定类型参数。这导致用于调用泛型中的成员的语法与用于非泛型的语法没有区别。

如何解决冲突

若要解决此规则的冲突，请删除静态成员或将其更改为实例成员。

何时禁止显示警告

不禁止显示此规则发出的警告。以易于理解和使用的语法提供泛型，可减少学习所需的时间，并增加新库的采用率。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1005:避免泛型类型的参数过多
- CA1010:集合应实现泛型接口
- CA1002:不要公开泛型列表
- CA1003:使用泛型事件处理程序实例

另请参阅

- 泛型

CA1001:具有可释放字段的类型应该是可释放的

2021/11/16 •

	‹
■ ID	CA1001
■	设计
修复是中断修复还是非中断修复	非中断 - 如果类型在程序集外部不可见，则为非中断修复。 中断 - 如果类型在程序集外部可见，则为中断修复。

原因

一个类声明并实现 System.IDisposable 类型的实例字段，但该类不实现 IDisposable。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

声明 IDisposable 字段的类间接拥有非托管资源。类应实现 IDisposable 接口，在资源不再使用时释放它拥有的非托管资源。如果类不直接拥有任何非托管资源，它不应实现终结器。

此规则将实现 System.IAsyncDisposable 的类型视为可释放类型。

如何解决冲突

若要解决此规则的冲突，请实现 IDisposable 接口。在 IDisposable.Dispose 方法中，调用字段类型的 Dispose 方法。

何时禁止显示警告

通常，不要禁止显示此规则发出的警告。当包含类型不包含字段的释放所有权时，可以禁止显示该警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

可以仅为此规则、为所有规则或为此类别（设计）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

示例

下面的示例演示一个与规则冲突的类和一个通过实现 IDisposable 来符合规则的类。类不实现终结器，因为该类不直接拥有任何非托管资源。

```
Imports System
Imports System.IO

Namespace ca1001

    ' This class violates the rule.
    Public Class NoDisposeMethod

        Dim newFile As FileStream

        Sub New()
            newFile = New FileStream("c:\temp.txt", FileMode.Open)
        End Sub

    End Class

    ' This class satisfies the rule.
    Public Class HasDisposeMethod
        Implements IDisposable

        Dim newFile As FileStream

        Sub New()
            newFile = New FileStream("c:\temp.txt", FileMode.Open)
        End Sub

        Protected Overridable Overloads Sub Dispose(disposing As Boolean)

            If disposing Then
                ' dispose managed resources
                newFile.Close()
            End If

            ' free native resources

        End Sub 'Dispose


        Public Overloads Sub Dispose() Implements IDisposable.Dispose

            Dispose(True)
            GC.SuppressFinalize(Me)

        End Sub 'Dispose

    End Class

End Namespace
```

```
// This class violates the rule.
public class NoDisposeMethod
{
    FileStream _newFile;

    public NoDisposeMethod()
    {
        _newFile = new FileStream(@"c:\temp.txt", FileMode.Open);
    }
}

// This class satisfies the rule.
public class HasDisposeMethod : IDisposable
{
    FileStream _newFile;

    public HasDisposeMethod()
    {
        _newFile = new FileStream(@"c:\temp.txt", FileMode.Open);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Dispose managed resources.
            _newFile.Close();
        }
        // Free native resources.
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

相关规则

- CA2213:应释放可释放的字段
- CA2216:可释放类型应声明终结器
- CA2215:Dispose 方法应调用基类释放

另请参阅

- 实现 Dispose 方法

CA1002:不要公开泛型列表

2021/11/16 •

	[
■ ID	CA1002
■	设计
▬▬	重大

原因

类型包含外部可见的成员，该成员是 System.Collections.Generic.List<T> 类型、返回 List<T> 类型，或其签名包含 List<T> 参数。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

System.Collections.Generic.List<T> 是针对性能(而非继承)设计的泛型集合。List<T> 不包含可使更改继承类的行为变得更容易的虚拟成员。应改为公开以下针对继承设计的泛型集合，而不是公开 List<T>。

- System.Collections.ObjectModel.Collection<T>
- System.Collections.ObjectModel.ReadOnlyCollection<T>
- System.Collections.ObjectModel.KeyedCollection<TKey,TItem>
- System.Collections.Generic.IList<T>
- System.Collections.Generic.ICollection<T>

如何解决冲突

若要解决与此规则的冲突，请将 System.Collections.Generic.List<T> 类型更改为针对继承而设计的泛型集合之一。

何时禁止显示警告

除非引发此警告的程序集不是可重复使用的库，否则请勿禁止显示此规则的警告。例如，在性能优化的应用程序中可禁止显示此警告，因为使用泛型列表可以提高应用程序性能。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

CA1005:避免泛型类型的参数过多

CA1010:集合应实现泛型接口

CA1000:不要在泛型类型中声明静态成员

CA1003:使用泛型事件处理程序实例

另请参阅

泛型

CA1003:使用泛型事件处理程序实例

2021/11/16 •

[	"["
RuleId	CA1003
类别	设计
修复是中断修复还是非中断修复	重大

原因

某个类型包含的委托返回 void，且该委托的签名包含两个参数（第一个参数是对象，第二个参数是可以分配给 EventArgs 的类型），而且包含程序集面向的是 .NET。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

在 .NET Framework 2.0 之前，为了将自定义信息传递到事件处理程序，必须将新委托声明为指定派生自 System.EventArgs 类的类。在 .NET Framework 2.0 及更高版本中，泛型 System.EventHandler<TEventArgs> 委托允许将所有派生自 EventArgs 的类与事件处理程序一起使用。

如何解决冲突

若要解决此规则的冲突，请删除委托并使用 System.EventHandler<TEventArgs> 委托替换其使用。

如果委托由 Visual Basic 编译器自动生成，请更改事件声明的语法以使用 System.EventHandler<TEventArgs> 委托。

何时禁止显示警告

不禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示了与此规则冲突的委托。在 Visual Basic 示例中，注释说明了如何修改示例以符合规则。对于 C# 示例，下面的示例演示了修改后的代码。

```
Imports System

Namespace ca1003

    Public Class CustomEventArgs
        Inherits EventArgs

        Public info As String = "data"

    End Class

    Public Class ClassThatRaisesEvent

        ' This statement creates a new delegate, which violates the rule.
        Event SomeEvent(sender As Object, e As CustomEventArgs)

        ' To satisfy the rule, comment out the previous line
        ' and uncomment the following line.
        'Event SomeEvent As EventHandler(Of CustomEventArgs)

        Protected Overridable Sub OnSomeEvent(e As CustomEventArgs)
            RaiseEvent SomeEvent(Me, e)
        End Sub

        Sub SimulateEvent()
            OnSomeEvent(New CustomEventArgs())
        End Sub

    End Class

    Public Class ClassThatHandlesEvent

        Sub New(eventRaiser As ClassThatRaisesEvent)
            AddHandler eventRaiser.SomeEvent, AddressOf HandleEvent
        End Sub

        Private Sub HandleEvent(sender As Object, e As CustomEventArgs)
            Console.WriteLine("Event handled: {0}", e.info)
        End Sub

    End Class

    Class Test

        Shared Sub Main()

            Dim eventRaiser As New ClassThatRaisesEvent()
            Dim eventHandler As New ClassThatHandlesEvent(eventRaiser)

            eventRaiser.SimulateEvent()

        End Sub

    End Class

End Namespace
```

```
// This delegate violates the rule.
public delegate void CustomEventHandler(object sender, CustomEventArgs e);

public class CustomEventArgs : EventArgs
{
    public string info = "data";
}

public class ClassThatRaisesEvent
{
    public event CustomEventHandler SomeEvent;

    protected virtual void OnSomeEvent(CustomEventArgs e)
    {
        SomeEvent?.Invoke(this, e);
    }

    public void SimulateEvent()
    {
        OnSomeEvent(new CustomEventArgs());
    }
}

public class ClassThatHandlesEvent
{
    public ClassThatHandlesEvent(ClassThatRaisesEvent eventRaiser)
    {
        eventRaiser.SomeEvent += new CustomEventHandler(HandleEvent);
    }

    private void HandleEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine("Event handled: {0}", e.info);
    }
}

class Test
{
    static void MainEvent()
    {
        var eventRaiser = new ClassThatRaisesEvent();
        var eventHandler = new ClassThatHandlesEvent(eventRaiser);

        eventRaiser.SimulateEvent();
    }
}
```

下面的代码片段会从上一个示例中删除符合规则的委托声明。它使用 System.EventHandler<TEventArgs> 委托替换其在 `ClassThatHandlesEvent` 和 `ClassThatRaisesEvent` 方法中的使用。

```
public class CustomEventArgs : EventArgs
{
    public string info = "data";
}

public class ClassThatRaisesEvent
{
    public event EventHandler<CustomEventArgs> SomeEvent;

    protected virtual void OnSomeEvent(CustomEventArgs e)
    {
        SomeEvent?.Invoke(this, e);
    }

    public void SimulateEvent()
    {
        OnSomeEvent(new CustomEventArgs());
    }
}

public class ClassThatHandlesEvent
{
    public ClassThatHandlesEvent(ClassThatRaisesEvent eventRaiser)
    {
        eventRaiser.SomeEvent += new EventHandler<CustomEventArgs>(HandleEvent);
    }

    private void HandleEvent(object sender, CustomEventArgs e)
    {
        Console.WriteLine("Event handled: {0}", e.info);
    }
}

class Test
{
    static void MainEvent()
    {
        var eventRaiser = new ClassThatRaisesEvent();
        var eventHandler = new ClassThatHandlesEvent(eventRaiser);

        eventRaiser.SimulateEvent();
    }
}
```

相关规则

- CA1005:避免泛型类型的参数过多
- CA1010:集合应实现泛型接口
- CA1000:不要在泛型类型中声明静态成员
- CA1002:不要公开泛型列表

另请参阅

- 泛型

CA1005:避免泛型类型的参数过多

2021/11/16 •

■ ID	CA1005
■	设计
■■■■■	重大

原因

外部可见的泛型类型具有两个以上的类型参数。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

泛型类型包含的类型参数越多，越难以知道并记住每个类型参数各代表什么。它通常有一个类型参数，如在 `List<T>` 中，而在某些情况下有两个类型参数，如在 `Dictionary<TKey, TValue>` 中。如果存在两个以上的类型参数，则大多数用户都会感到过于困难(例如 C# 中的 `TooManyTypeParameters<T, K, V>` 或 Visual Basic 中的 `TooManyTypeParameters(Of T, K, V)`)。

如何解决冲突

若要解决此规则的冲突，请将设计更改为使用不超过两个类型参数。

何时禁止显示警告

除非设计确实需要两个以上的类型参数，否则不要禁止显示此规则的警告。以易于理解和使用的语法提供泛型，可减少学习所需的时间，并增加新库的采用率。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

CA1010:集合应实现泛型接口

CA1000:不要在泛型类型中声明静态成员

CA1002:不要公开泛型列表

CA1003:使用泛型事件处理程序实例

另请参阅

- 泛型

CA1008:枚举应具有零值

2021/11/16 •

	[
■ ID	CA1008
■	设计
▬▬▬	非中断 - 如果系统提示你向无标志枚举添加 `None` 值。中断 - 如果系统提示你重命名或删除任何枚举值。

原因

没有应用 System.FlagsAttribute 的枚举不定义值为零的成员。或者，已应用 FlagsAttribute 的枚举定义值为零但其名称不为“None”的成员。或者，枚举定义多个零值成员。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

像其他值类型一样，未初始化枚举的默认值为零。无标志特性的枚举应定义值为零的成员，这样默认值即为该枚举的有效值。如果可行，请将成员命名为“None”。否则，将零赋给最常使用的成员。默认情况下，如果未在声明中设置第一个枚举成员的值，则其值为零。

如果应用了 FlagsAttribute 的枚举定义值为零成员，则该成员的名称应为“None”，以指示枚举中尚未设置值。将值为零的成员用于任何其他目的与使用 FlagsAttribute 存在冲突，因为 AND 和 OR 位运算符对成员没有意义。这意味着，只应为一个成员分配零值。如果有多个零值成员在标志特性的枚举中出现，对于不为零的成员，`Enum.ToString()` 将返回不正确的结果。

如何解决冲突

若要解决无标志特性枚举与此规则的冲突，请定义值为零的成员，这是一项非中断性变更。对于定义零值成员的标志特性枚举，请将此成员命名为“None”，并删除值为零的任何其他成员，这是一项中断性变更。

何时禁止显示警告

不要禁止显示此规则发出的警告，但之前已发布的标志特性枚举除外。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规

则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示两个满足规则的枚举和一个违反规则的枚举 `BadTraceOptions`。

```
using System;

namespace ca1008
{
    public enum TraceLevel
    {
        Off = 0,
        Error = 1,
        Warning = 2,
        Info = 3,
        Verbose = 4
    }

    [Flags]
    public enum TraceOptions
    {
        None = 0,
        CallStack = 0x01,
        LogicalStack = 0x02,
        DateTime = 0x04,
        Timestamp = 0x08,
    }

    [Flags]
    public enum BadTraceOptions
    {
        CallStack = 0,
        LogicalStack = 0x01,
        DateTime = 0x02,
        Timestamp = 0x04,
    }

    class UseBadTraceOptions
    {
        static void MainTrace()
        {
            // Set the flags.
            BadTraceOptions badOptions =
               BadTraceOptions.LogicalStack | BadTraceOptions.Timestamp;

            // Check whether CallStack is set.
            if ((badOptions & BadTraceOptions.CallStack) ==
                BadTraceOptions.CallStack)
            {
                // This 'if' statement is always true.
            }
        }
    }
}
```

```
Imports System

Namespace ca1008

    Public Enum TraceLevel
        Off = 0
        AnError = 1
        Warning = 2
        Info = 3
        Verbose = 4
    End Enum

    <Flags>
    Public Enum TraceOptions
        None = 0
        CallStack = &H1
        LogicalStack = &H2
        DateTime = &H4
        Timestamp = &H8
    End Enum

    <Flags>
    Public Enum BadTraceOptions
        CallStack = 0
        LogicalStack = &H1
        DateTime = &H2
        Timestamp = &H4
    End Enum

    Class UseBadTraceOptions

        Shared Sub Main1008()

            ' Set the flags.
            Dim badOptions As BadTraceOptions =
            BadTraceOptions.LogicalStack Or BadTraceOptions.Timestamp

            ' Check whether CallStack is set.
            If ((badOptions And BadTraceOptions.CallStack) =
             BadTraceOptions.CallStack) Then
                ' This 'If' statement is always true.
            End If

        End Sub

    End Class

End Namespace
```

相关规则

- CA2217:不要使用 FlagsAttribute 标记枚举
- CA1700:不要命名“Reserved”枚举值
- CA1712:不要将类型名用作枚举值的前缀
- CA1028:枚举存储应为 Int32
- CA1027:用 FlagsAttribute 标记枚举

另请参阅

- System.Enum

CA1010:集合应实现泛型接口

2021/11/16 •

[	"["
RuleId	CA1010
类别	设计
修复是中断修复还是非中断修复	非中断

原因

类型实现 System.Collections.IEnumerable 接口，但不能实现 System.Collections.Generic.IEnumerable<T> 接口和包含程序集的目标 .NET。此规则会忽略能够实现 System.Collections.IDictionary 的类型。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。还可配置其他接口以要求实现泛型接口。

规则说明

若要扩大集合的用途，应实现某个泛型集合接口。然后，可以使用该集合来填充泛型集合类型，如下所示：

- System.Collections.Generic.List<T>
- System.Collections.Generic.Queue<T>
- System.Collections.Generic.Stack<T>

如何解决冲突

若要解决此规则的冲突，请实现某个泛型集合接口：

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.ICollection<T>
- System.Collections.Generic.IList<T>

何时禁止显示警告

禁止显示此规则的警告是安全的；但是，集合的使用将受到更多限制。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 其他所需的泛型接口

你可以仅为此规则、为所有规则或为此类别(设计)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

其他所需的泛型接口

你可以配置接口名称列表(用 `|` 进行分隔)及其所需的通用完全限定接口(用 `->` 进行分隔)。

允许的接口格式：

- 仅接口名称(包括具有相应名称的所有接口，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CA1010.additional_required_generic_interfaces = ISomething->System.Collections.Generic.IEnumerable`1`	所有实现 `ISomething` 的类型，无论其名称空间如何，都应该实现 System.Collections.Generic.IEnumerable<T>。
`dotnet_code_quality.CA1010.additional_required_generic_interfaces = T:System.Collections.IDictionary->T:System.Collections.Generic.IDictionary`2`	所有实现 System.Collections.IDictionary 的类型都应该实现 System.Collections.Generic.IDictionary<TKey,TValue>。

示例

以下示例显示从非泛型 `CollectionBase` 类派生并与此规则产生冲突的类。

```
public class Book
{
    public Book()
    {
    }
}

public class BookCollection : CollectionBase
{
    public BookCollection()
    {
    }

    public void Add(Book value)
    {
        InnerList.Add(value);
    }

    public void Remove(Book value)
    {
        InnerList.Remove(value);
    }

    public void Insert(int index, Book value)
    {
        InnerList.Insert(index, value);
    }

    public Book this[int index]
    {
        get { return (Book)InnerList[index]; }
        set { InnerList[index] = value; }
    }

    public bool Contains(Book value)
    {
        return InnerList.Contains(value);
    }

    public int IndexOf(Book value)
    {
        return InnerList.IndexOf(value);
    }

    public void CopyTo(Book[] array, int arrayIndex)
    {
        InnerList.CopyTo(array, arrayIndex);
    }
}
```

若要解决此规则的冲突，请执行以下某个操作：

- 实现泛型接口。
- 将基类更改为已同时实现泛型和非泛型接口的类型（如 `Collection<T>` 类）。

通过接口实现来解决

以下示例通过实现 IEnumerable<T>、ICollection<T> 和 IList<T> 等泛型接口来解决冲突。

```
public class Book
{
    public Book()
    {
    }
}
```

```
public class BookCollection : CollectionBase, IList<Book>
{
    public BookCollection()
    {
    }

    int IList<Book>.IndexOf(Book item)
    {
        return this.List.IndexOf(item);
    }

    void IList<Book>.Insert(int location, Book item)
    {
    }

    Book IList<Book>.this[int index]
    {
        get { return (Book)this.List[index]; }
        set { }
    }

    void ICollection<Book>.Add(Book item)
    {
    }

    bool ICollection<Book>.Contains(Book item)
    {
        return true;
    }

    void ICollection<Book>.CopyTo(Book[] array, int arrayIndex)
    {
    }

    bool ICollection<Book>.IsReadOnly
    {
        get { return false; }
    }

    bool ICollection<Book>.Remove(Book item)
    {
        if (InnerList.Contains(item))
        {
            InnerList.Remove(item);
            return true;
        }
        return false;
    }

    IEnumerator<Book> IEnumerable<Book>.GetEnumerator()
    {
        return new BookCollectionEnumerator(InnerList.GetEnumerator());
    }

    private class BookCollectionEnumerator : IEnumerator<Book>
    {
        private IEnumerator _Enumerator;

        public BookCollectionEnumerator(IEnumerator enumerator)
        {
            _Enumerator = enumerator;
        }

        public Book Current
        {
            get { return (Book)_Enumerator.Current; }
        }

        object IEnumerator.Current
```

```
        {
            get { return _Enumerator.Current; }
        }

        public bool MoveNext()
        {
            return _Enumerator.MoveNext();
        }

        public void Reset()
        {
            _Enumerator.Reset();
        }

        public void Dispose()
        {
        }
    }
}
```

通过基类更改来解决

以下示例通过将集合的基类从非泛型 `CollectionBase` 类更改为泛型 `Collection<T>`（在 Visual Basic 为 `Collection(Of T)`）类来解决冲突。

```
public class Book
{
    public Book()
    {
    }
}

public class BookCollection : Collection<Book>
{
    public BookCollection()
    {
    }
}
```

更改已发布的类的基类，这是对现有使用者的突破性更改。

相关规则

- CA1005:避免泛型类型的参数过多
- CA1000:不要在泛型类型中声明静态成员
- CA1002:不要公开泛型列表
- CA1003:使用泛型事件处理程序实例

另请参阅

- 泛型

CA1012：抽象类型不应具有公共构造函数

2021/11/16 •

[	“[”
RuleId	CA1012
类别	设计
修复是中断修复还是非中断修复	非中断

原因

类型为抽象类型并且具有公共构造函数。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

抽象类型的构造函数只能由派生类型调用。由于公共构造函数可创建类型的实例，但无法创建抽象类型的实例，因此具有公共构造函数的抽象类型在设计上是错误的。

如何解决冲突

若要解决此规则的冲突，请将构造函数设置为受保护的函数，或者不将该类型声明为抽象类型。

何时禁止显示警告

不禁止显示此规则发出的警告。抽象类型具有公共构造函数。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的代码片段包含与此规则冲突的抽象类型。

```
' Violates this rule
Public MustInherit Class Book

    Public Sub New()
    End Sub

End Class
```

```
// Violates this rule
public abstract class Book
{
    public Book()
    {
    }
}
```

下面的代码片段将构造函数的可访问性从 `public` 更改为 `protected` 来解决以前的冲突。

```
// Does not violate this rule
public abstract class Book
{
    protected Book()
    {
    }
}
```

```
' Violates this rule
Public MustInherit Class Book

    Protected Sub New()
    End Sub

End Class
```

CA1014:用 CLSCompliantAttribute 标记程序集

2021/11/16 •

	“[”
RuleId	CA1014
类别	设计
修复是中断修复还是非中断修复	非中断

原因

程序集没有已应用的 System.CLSCompliantAttribute 属性。

规则说明

公共语言规范 (CLS) 定义了程序集在跨编程语言使用时必须符合的命名限制、数据类型和规则。好的设计要求所有程序集用 CLSCompliantAttribute 显式指示 CLS 合规性。如果程序集没有此属性，则该程序集即不合规。

符合 CLS 的程序集可能包含不合规的类型或类型成员。

如何解决冲突

若要解决此规则的冲突，请将属性添加到程序集。应确定不合规的类型或类型成员，并将这些元素标记为不合规，而不是将整个程序集标记为不相容。如果可能，应为不合规的成员提供符合 CLS 的替代方法，让尽可能多的用户能够访问程序集的所有功能。

何时禁止显示警告

不禁止显示此规则发出的警告。如果不希望程序集符合 CLS，请应用属性并将其值设置为 `false`。

如果必须禁止显示此警告，请向 .globalconfig 文件添加 `dotnet_diagnostic.CA1014.severity = none`。或者，如果项目中没有代码文件，请将 `<NoWarn>CA1041</NoWarn>` 添加到项目文件。

示例

下面的示例演示应用了 System.CLSCompliantAttribute 属性的程序集，该属性声明此程序集符合 CLS。

```
[assembly:CLSCompliant(true)]
namespace DesignLibrary {}
```

```
<assembly:CLSCompliant(true)>
Namespace DesignLibrary
End Namespace
```

另请参阅

- System.CLSCompliantAttribute

- 语言独立性和与语言无关的组件

CA1016:用 AssemblyVersionAttribute 标记程序集

2021/11/16 •

[	"["
RuleId	CA1016
类别	设计
修复是中断修复还是非中断修复	非中断

原因

程序集没有版本号。

规则说明

程序集的标识由以下信息组成:

- 程序集名称
- 版本号
- 环境
- 公钥(用于强名称程序集)。

.NET 使用版本号来唯一标识程序集,并绑定到强名称程序集中的类型。版本号与版本和发行者策略一起使用。默认情况下,仅使用用于生成应用程序的程序集版本运行应用程序。

如何解决冲突

若要解决此规则的冲突,请使用 System.Reflection.AssemblyVersionAttribute 属性将版本号添加到程序集。

何时禁止显示警告

对于第三方或生产环境中使用的程序集,请勿禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告,包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息,请参阅如何禁止显示代码分析警告。

示例

下面的示例演示应用了 AssemblyVersionAttribute 属性的程序集。

```
using System;
using System.Reflection;

[assembly: AssemblyVersionAttribute("4.3.2.1")]
namespace DesignLibrary {}
```

```
<Assembly: AssemblyVersionAttribute("4.3.2.1")>
Namespace DesignLibrary
End Namespace
```

请参阅

- 程序集版本控制
- 如何：创建发布者策略

CA1017:用 ComVisibleAttribute 标记程序集

2021/11/16 •

[	"["
RuleId	CA1017
类别	设计
修复是中断修复还是非中断修复	非中断

原因

程序集没有已应用的 System.Runtime.InteropServices.ComVisibleAttribute 属性。

规则说明

ComVisibleAttribute 属性确定 COM 客户端如何访问托管代码。合理的设计指出程序集将显式指示 COM 可见性。可以针对整个程序集设置 COM 可见性，然后重写各个类型和类型成员的 COM 可见性。如果此属性不存在，则程序集的内容对 COM 客户端可见。

如何解决冲突

若要解决此规则的冲突，请将该属性添加到程序集。如果你不希望程序集对 COM 客户端可见，请应用该属性并将其值设置为 `false`。

何时禁止显示警告

不禁止显示此规则发出的警告。如果希望程序集可见，请应用该属性并将其值设置为 `true`。

示例

下面的示例演示一个应用了 ComVisibleAttribute 属性的程序集，以防止其对 COM 客户端可见。

```
<Assembly: System.Runtime.InteropServices.ComVisible(False)>
Namespace DesignLibrary
End Namespace
```

```
[assembly: System.Runtime.InteropServices.ComVisible(false)]
namespace DesignLibrary {}
```

请参阅

- 与非托管代码交互操作
- 为互操作限定 .NET 类型

CA1018:用 AttributeUsageAttribute 标记特性

2021/11/16 •

	“”
RuleId	CA1018
类别	设计
修复是中断修复还是非中断修复	重大

原因

自定义特性上不存在 System.AttributeUsageAttribute 特性。

规则说明

当定义自定义特性时，用 AttributeUsageAttribute 标记该特性，以指示源代码中可以应用自定义特性的位置。特性的含义和预定用法将决定它在代码中的有效位置。例如，你可以定义一个特性，该特性标识负责维护和增强库中的每个类型的人员，并且此责任始终在类型级别上分配。在这种情况下，编译器应在类、枚举和接口上启用该特性，但不应在方法、事件或属性上启用它。组织策略和过程将规定是否应在程序集上启用该特性。

System.AttributeTargets 枚举定义可为自定义特性指定的目标。如果省略 AttributeUsageAttribute，则自定义特性将对所有目标有效，如 AttributeTargets 枚举的 `All` 值所定义。

如何解决冲突

若要解决此规则的冲突，请使用 AttributeUsageAttribute 指定特性的目标。请参阅以下示例。

何时禁止显示警告

应解决此规则的冲突，而不是排除消息。即使该特性继承 AttributeUsageAttribute，也应该提供该特性以简化代码维护。

示例

下面的示例定义了两个特性。`BadCodeMaintainerAttribute` 错误地省略了 AttributeUsageAttribute 语句，但 `GoodCodeMaintainerAttribute` 正确实现了本部分前面所述的特性。（设计规则 CA1019：定义特性参数的访问器要求属性 `DeveloperName`，出于完整性考虑，此属性包含在内。）

```
using System;

namespace ca1018
{
    // Violates rule: MarkAttributesWithAttributeUsage.
    public sealed class BadCodeMaintainerAttribute : Attribute
    {
        public BadCodeMaintainerAttribute(string developerName)
        {
            DeveloperName = developerName;
        }
        public string DeveloperName { get; }
    }

    // Satisfies rule: Attributes specify AttributeUsage.
    // This attribute is valid for type-level targets.
    [AttributeUsage(AttributeTargets.Class | AttributeTargets.Enum | AttributeTargets.Interface |
AttributeTargets.Delegate)]
    public sealed class GoodCodeMaintainerAttribute : Attribute
    {
        public GoodCodeMaintainerAttribute(string developerName)
        {
            DeveloperName = developerName;
        }
        public string DeveloperName { get; }
    }
}
```

```
Imports System

Namespace ca1018

    ' Violates rule: MarkAttributesWithAttributeUsage.
    Public NotInheritable Class BadCodeMaintainerAttribute
        Inherits Attribute

        Public Sub New(developerName As String)
            Me.DeveloperName = developerName
        End Sub 'New

        Public ReadOnly Property DeveloperName() As String
    End Class

    ' Satisfies rule: Attributes specify AttributeUsage.
    ' The attribute is valid for type-level targets.
    <AttributeUsage(AttributeTargets.Class Or AttributeTargets.Enum Or
   AttributeTargets.Interface Or AttributeTargets.Delegate)>
    Public NotInheritable Class GoodCodeMaintainerAttribute
        Inherits Attribute

        Public Sub New(developerName As String)
            Me.DeveloperName = developerName
        End Sub 'New

        Public ReadOnly Property DeveloperName() As String
    End Class

End Namespace
```

相关规则

- CA1019:定义特性参数的访问器
- CA1813:避免使用非密封特性

请参阅

- 特性

CA1019:定义特性参数的访问器

2021/11/16 •

	“
RuleId	CA1019
类别	设计
修复是中断修复还是非中断修复	非中断

原因

在其构造函数中，特性定义了没有相应属性的参数。

规则说明

特性可以定义强制自变量，在对目标应用该特性时必须指定这些自变量。这些实参也称为位置实参，因为它们将作为位置形参提供给特性构造函数。对于每一个强制变量，特性还必须提供一个相应的只读属性，以便可以在执行时检索该变量的值。此规则检查是否已为每个构造函数参数定义了相应属性。

特性还可以定义可选实参，可选实参也称为命名实参。这些变量按名称提供给特性构造函数，并且必须具有相应的读/写属性。

对于强制参数和可选参数，相应属性和构造函数参数应使用相同的名称，但大小写不同。属性使用 Pascal 大小写，参数使用 Camel 大小写。

如何解决冲突

若要解决此规则的冲突，请为每个没有只读属性的构造函数参数添加一个只读属性。

何时禁止显示警告

如果不希望强制参数的值可检索，则禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

示例

自定义特性

下面的示例演示了定义强制(位置)参数的两个特性。未正确定义特性的首个实现。第二个实现正确。

```
// Violates rule: DefineAccessorsForAttributeArguments.
[AttributeUsage(AttributeTargets.All)]
public sealed class BadCustomAttribute : Attribute
{
    string _data;

    // Missing the property that corresponds to
    // the someStringData constructor parameter.

    public BadCustomAttribute(string someStringData)
    {
        _data = someStringData;
    }
}

// Satisfies rule: Attributes should have accessors for all arguments.
[AttributeUsage(AttributeTargets.All)]
public sealed class GoodCustomAttribute : Attribute
{
    public GoodCustomAttribute(string someStringData)
    {
        SomeStringData = someStringData;
    }

    //The constructor parameter and property
    //name are the same except for case.

    public string SomeStringData { get; }
}
```

```
Imports System

Namespace ca1019

    ' Violates rule: DefineAccessorsForAttributeArguments.
    <AttributeUsage(AttributeTargets.All)>
    Public NotInheritable Class BadCustomAttribute
        Inherits Attribute
        Private data As String

        ' Missing the property that corresponds to
        ' the someStringData parameter.
        Public Sub New(someStringData As String)
            data = someStringData
        End Sub 'New
    End Class 'BadCustomAttribute

    ' Satisfies rule: Attributes should have accessors for all arguments.
    <AttributeUsage(AttributeTargets.All)>
    Public NotInheritable Class GoodCustomAttribute
        Inherits Attribute

        Public Sub New(someStringData As String)
            Me.SomeStringData = someStringData
        End Sub 'New

        'The constructor parameter and property
        'name are the same except for case.

        Public ReadOnly Property SomeStringData() As String
    End Class

End Namespace
```

位置参数和命名参数

位置参数和命名参数使库的使用者清楚地了解对于特性而言，哪些参数是强制的，哪些参数是可选的。

下面的示例演示了具有位置参数和命名参数的特性的实现：

```
[AttributeUsage(AttributeTargets.All)]
public sealed class GoodCustomAttribute : Attribute
{
    public GoodCustomAttribute(string mandatoryData)
    {
        MandatoryData = mandatoryData;
    }

    public string MandatoryData { get; }

    public string OptionalData { get; set; }
}
```

下面的示例演示了如何将自定义特性应用于两个属性：

```
[GoodCustomAttribute("ThisIsSomeMandatoryData", OptionalData = "ThisIsSomeOptionalData")]
public string MyProperty { get; set; }

[GoodCustomAttribute("ThisIsSomeMoreMandatoryData")]
public string MyOtherProperty { get; set; }
```

相关规则

CA1813:避免使用非密封特性

请参阅

- 特性

CA1021:避免使用 out 参数

2021/11/16 •

[	"["
RuleId	CA1021
类别	设计
修复是中断修复还是非中断修复	重大

原因

公共类型中的公共或受保护方法具有 `out` 参数。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

按引用(使用 `out` 或 `ref`)传递类型要求具有使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。另外，`out` 和 `ref` 参数之间的区别并未得到广泛了解。

如果引用类型"按引用"传递，则该方法会使用参数来返回对象的不同实例。按引用传递引用类型也称为使用双指针、指向指针的指针或双间接。通过使用"按值"传递这一默认调用约定，采用引用类型的参数已经收到指向对象的指针。指针(而不是它指向的对象)按值传递。按值传递表示方法不能更改指针以使其指向引用类型的新实例。但是，它可以更改它所指向的对象的内容。对于大多数应用程序，这就足够了，还生成了所需的行为。

如果方法必须返回不同的实例，请使用该方法的返回值来实现此目的。有关对字符串执行操作并返回字符串的新实例的各种方法，请参阅 System.String 类。使用此模型时，调用方必须决定是否保留原始对象。

尽管返回值很常见且被大量使用，但正确应用 `out` 和 `ref` 参数需要中间设计和编码技能。为一般用户进行设计的库架构师不应指望用户能熟练运用 `out` 或 `ref` 参数。

如何解决冲突

要修复由值类型引起的此规则的冲突，需使方法返回对象作为其返回值。如果该方法必须返回多个值，请重新设计它以返回保存值的对象的单个实例。

要修复由引用类型引起的此规则的冲突，需确保所需的行为是否为返回引用的新实例。如果是，则该方法应使用其返回值来执行此操作。

何时禁止显示警告

可禁止显示此规则发出的警告。但这种设计可能会引发可用性问题。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例 1

以下库显示生成对用户反馈响应的类的两个实现。第一个实现 (`BadRefAndOut`) 强制库用户管理三个返回值。第二个实现 (`RedesignedRefAndOut`) 通过返回容器类 (`ReplyData`) 的实例来简化用户体验，该容器类将数据作为单个单元进行管理。

```
public enum Actions
{
    Unknown,
    Discard,
    ForwardToManagement,
    ForwardToDeveloper
}

public enum TypeOfFeedback
{
    Complaint,
    Praise,
    Suggestion,
    Incomprehensible
}

public class BadRefAndOut
{
    // Violates rule: DoNotPassTypesByReference.

    public static bool ReplyInformation(TypeOfFeedback input,
       out string reply, ref Actions action)
    {
        bool returnReply = false;
        string replyText = "Your feedback has been forwarded " +
                           "to the product manager.";

        reply = String.Empty;
        switch (input)
        {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise:
                action = Actions.ForwardToManagement;
                reply = "Thank you. " + replyText;
                returnReply = true;
                break;
            case TypeOfFeedback.Suggestion:
                action = Actions.ForwardToDeveloper;
                reply = replyText;
                returnReply = true;
                break;
```

```
            case TypeOfFeedback.Incomprehensible:
            default:
                action = Actions.Discard;
                returnReply = false;
                break;
        }
        return returnReply;
    }
}

// Redesigned version does not use out or ref parameters.
// Instead, it returns this container type.

public class ReplyData
{
    bool _returnReply;

    // Constructors.
    public ReplyData()
    {
        this.Reply = String.Empty;
        this.Action = Actions.Discard;
        this._returnReply = false;
    }

    public ReplyData(Actions action, string reply, bool returnReply)
    {
        this.Reply = reply;
        this.Action = action;
        this._returnReply = returnReply;
    }

    // Properties.
    public string Reply { get; }
    public Actions Action { get; }

    public override string ToString()
    {
        return String.Format("Reply: {0} Action: {1} return? {2}",
           Reply, Action.ToString(), _returnReply.ToString());
    }
}

public class RedesignedRefAndOut
{
    public static ReplyData ReplyInformation(TypeOfFeedback input)
    {
        ReplyData answer;
        string replyText = "Your feedback has been forwarded " +
           "to the product manager.";

        switch (input)
        {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise:
                answer = new ReplyData(
                   Actions.ForwardToManagement,
                   "Thank you. " + replyText,
                   true);
                break;
            case TypeOfFeedback.Suggestion:
                answer = new ReplyData(
                   Actions.ForwardToDeveloper,
                   replyText,
                   true);
                break;
            case TypeOfFeedback.Incomprehensible:
            default:
                answer = new ReplyData();
```

```
                break;
        }
        return answer;
    }
}
```

示例 2

以下应用程序说明了用户的体验。对重新设计的库的调用(`UseTheSimplifiedClass` 方法)更简单，并且由方法返回的信息非常易于管理。这两个方法的输出是相同的。

```
public class UseComplexMethod
{
    static void UseTheComplicatedClass()
    {
        // Using the version with the ref and out parameters.
        // You do not have to initialize an out parameter.

        string[] reply = new string[5];

        // You must initialize a ref parameter.
        Actions[] action = {Actions.Unknown,Actions.Unknown,
                            Actions.Unknown,Actions.Unknown,
                            Actions.Unknown,Actions.Unknown};
        bool[] disposition = new bool[5];
        int i = 0;

        foreach (TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
        {
            // The call to the library.
            disposition[i] = BadRefAndOut.ReplyInformation(
               t, out reply[i], ref action[i]);
            Console.WriteLine("Reply: {0} Action: {1}  return? {2} ",
               reply[i], action[i], disposition[i]);
            i++;
        }
    }

    static void UseTheSimplifiedClass()
    {
        ReplyData[] answer = new ReplyData[5];
        int i = 0;
        foreach (TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
        {
            // The call to the library.
            answer[i] = RedesignedRefAndOut.ReplyInformation(t);
            Console.WriteLine(answer[i++]);
        }
    }

    public static void UseClasses()
    {
        UseTheComplicatedClass();

        // Print a blank line in output.
        Console.WriteLine("");

        UseTheSimplifiedClass();
    }
}
```

示例 3

下面的示例库说明了如何使用引用类型的 `ref` 参数，并演示了实现此功能的更好方法。

```
public class ReferenceTypesAndParameters
{
    // The following syntax will not work. You cannot make a
    // reference type that is passed by value point to a new
    // instance. This needs the ref keyword.

    public static void BadPassTheObject(string argument)
    {
        argument += " ABCDE";
    }

    // The following syntax works, but is considered bad design.
    // It reassigns the argument to point to a new instance of string.
    // Violates rule DoNotPassTypesByReference.

    public static void PassTheReference(ref string argument)
    {
        argument += " ABCDE";
    }

    // The following syntax works and is a better design.
    // It returns the altered argument as a new instance of string.

    public static string BetterThanPassTheReference(string argument)
    {
        return argument + " ABCDE";
    }
}
```

示例 4

下面的应用程序调用库中的每个方法来演示行为。

```
public class Test
{
    public static void MainTest()
    {
        string s1 = "12345";
        string s2 = "12345";
        string s3 = "12345";

        Console.WriteLine("Changing pointer - passed by value:");
        Console.WriteLine(s1);
        ReferenceTypesAndParameters.BadPassTheObject(s1);
        Console.WriteLine(s1);

        Console.WriteLine("Changing pointer - passed by reference:");
        Console.WriteLine(s2);
        ReferenceTypesAndParameters.PassTheReference(ref s2);
        Console.WriteLine(s2);

        Console.WriteLine("Passing by return value:");
        s3 = ReferenceTypesAndParameters.BetterThanPassTheReference(s3);
        Console.WriteLine(s3);
    }
}
```

该示例产生下面的输出：

```
Changing pointer - passed by value:
12345
12345
Changing pointer - passed by reference:
12345
12345 ABCDE
Passing by return value:
12345 ABCDE
```

Try 模式方法

实现 Try<Something> 模式的方法(如 System.Int32.TryParse)不会引发此冲突。下面的示例演示了实现 System.Int32.TryParse 方法的结构(值类型)。

```
public struct Point
{
    public Point(int axisX, int axisY)
    {
        X = axisX;
        Y = axisY;
    }

    public int X { get; }

    public int Y { get; }

    public override int GetHashCode()
    {
        return X ^ Y;
    }

    public override bool Equals(object obj)
    {
        if (!(obj is Point))
            return false;

        return Equals((Point)obj);
    }

    public bool Equals(Point other)
    {
        if (X != other.X)
            return false;

        return Y == other.Y;
    }

    public static bool operator ==(Point point1, Point point2)
    {
        return point1.Equals(point2);
    }

    public static bool operator !=(Point point1, Point point2)
    {
        return !point1.Equals(point2);
    }

    // Does not violate this rule
    public static bool TryParse(string value, out Point result)
    {
        // TryParse Implementation
        result = new Point(0, 0);
        return false;
    }
}
```

相关规则

CA1045:不要通过引用来传递类型

CA1024:在适用处使用属性

2021/11/16 •

[	"["
RuleId	CA1024
类别	设计
修复是中断修复还是非中断修复	重大

原因

一个方法的名称以 `Get` 开头，不采用任何参数，并返回一个非数组的值。

默认情况下，此规则仅查看外部可见的方法，但这是可配置的。

规则说明

在大多数情况下，属性表示数据，方法执行操作。访问属性的方式类似于访问字段，这使得它们更易于使用。如果一个方法具备以下条件之一，则该方法可能很适合成为属性：

- 方法不采用任何自变量，并返回对象的状态信息。
- 方法接受单个自变量，以设置对象的部分状态。

如何解决冲突

若要解决此规则的冲突，请将方法更改为属性。

何时禁止显示警告

如果方法满足以下条件之一，则禁止显示此规则发出的警告。在下面的情形下，方法比属性更可取。

- 方法表现的行为不像字段。
- 方法执行耗时的操作。方法设置或获取字段值所需的时间明显更长。
- 方法执行了一个转换。访问一个字段不会返回它所存储的数据的转换版本。
- `Get` 方法有一个明显的副作用。检索字段的值不会产生任何副作用。
- 执行的顺序很重要。设置字段的值不依赖于其他操作的发生。
- 连续调用方法两次会产生不同的结果。
- 方法是 `static`，但返回一个可由调用方更改的对象。检索字段的值不允许调用方更改由字段存储的数据。
- 方法返回一个数组。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例包含了几个应转换为属性的方法，和几个不应转换为属性的方法(因为它们的行为不像字段)。

```
public class Appointment
{
    static long nextAppointmentID;
    static double[] discountScale = { 5.0, 10.0, 33.0 };
    string customerName;
    long customerID;
    DateTime when;

    // Static constructor.
    static Appointment()
    {
        // Initializes the static variable for Next appointment ID.
    }

    // This method violates the rule, but should not be a property.
    // This method has an observable side effect.
    // Calling the method twice in succession creates different results.
    public static long GetNextAvailableID()
    {
        nextAppointmentID++;
        return nextAppointmentID - 1;
    }

    // This method violates the rule, but should not be a property.
    // This method performs a time-consuming operation.
    // This method returns an array.
    public Appointment[] GetCustomerHistory()
    {
        // Connect to a database to get the customer's appointment history.
        return LoadHistoryFromDB(customerID);
    }

    // This method violates the rule, but should not be a property.
    // This method is static but returns a mutable object.
    public static double[] GetDiscountScaleForUpdate()
    {
        return discountScale;
    }

    // This method violates the rule, but should not be a property.
    // This method performs a conversion.
    public string GetWeekDayString()
    {
        return DateTimeFormatInfo.CurrentInfo.GetDayName(when.DayOfWeek);
    }

    // These methods violate the rule and should be properties.
```

```
    // These methods violate the rule and should be properties.
    // They each set or return a piece of the current object's state.

    public DayOfWeek GetWeekDay()
    {
        return when.DayOfWeek;
    }

    public void SetCustomerName(string customerName)
    {
        this.customerName = customerName;
    }

    public string GetCustomerName()
    {
        return customerName;
    }

    public void SetCustomerID(long customerID)
    {
        this.customerID = customerID;
    }

    public long GetCustomerID()
    {
        return customerID;
    }

    public void SetScheduleTime(DateTime when)
    {
        this.when = when;
    }

    public DateTime GetScheduleTime()
    {
        return when;
    }

    // Time-consuming method that is called by GetCustomerHistory.
    Appointment[] LoadHistoryFromDB(long customerID)
    {
        ArrayList records = new ArrayList();
        // Load from database.
        return (Appointment[])records.ToArray();
    }
}
```

```
Public Class Appointment
    Shared nextAppointmentID As Long
    Shared discountScale As Double() = {5.0, 10.0, 33.0}
    Private customerName As String
    Private customerID As Long
    Private [when] As Date

    ' Static constructor.
    Shared Sub New()
        ' Initializes the static variable for Next appointment ID.
    End Sub

    ' This method violates the rule, but should not be a property.
    ' This method has an observable side effect.
    ' Calling the method twice in succession creates different results.
    Public Shared Function GetNextAvailableID() As Long
        nextAppointmentID += 1
        Return nextAppointmentID - 1
    End Function

    ' This method violates the rule, but should not be a property.
```

```
    ' This method violates the rule, but should not be a property.
    ' This method performs a time-consuming operation.
    ' This method returns an array.
    Public Function GetCustomerHistory() As Appointment()
        ' Connect to a database to get the customer's appointment history.
        Return LoadHistoryFromDB(customerID)
    End Function

    ' This method violates the rule, but should not be a property.
    ' This method is static but returns a mutable object.
    Public Shared Function GetDiscountScaleForUpdate() As Double()
        Return discountScale
    End Function

    ' This method violates the rule, but should not be a property.
    ' This method performs a conversion.
    Public Function GetWeekDayString() As String
        Return DateTimeFormatInfo.CurrentInfo.GetDayName([when].DayOfWeek)
    End Function

    ' These methods violate the rule and should be properties.
    ' They each set or return a piece of the current object's state.

    Public Function GetWeekDay() As DayOfWeek
        Return [when].DayOfWeek
    End Function

    Public Sub SetCustomerName(customerName As String)
        Me.customerName = customerName
    End Sub

    Public Function GetCustomerName() As String
        Return customerName
    End Function

    Public Sub SetCustomerID(customerID As Long)
        Me.customerID = customerID
    End Sub

    Public Function GetCustomerID() As Long
        Return customerID
    End Function

    Public Sub SetScheduleTime([when] As Date)
        Me.[when] = [when]
    End Sub

    Public Function GetScheduleTime() As Date
        Return [when]
    End Function

    ' Time-consuming method that is called by GetCustomerHistory.
    Private Function LoadHistoryFromDB(customerID As Long) As Appointment()
        Dim records As ArrayList = New ArrayList()
        Return CType(records.ToArray(), Appointment())
    End Function
End Class
```

控制调试器中的属性扩展

编程人员避免使用属性的一个原因是，它们不希望调试器自动扩展它。例如，属性可能涉及到分配一个大型对象或调用一个 P/Invoke，但它实际上可能没有任何明显的副作用。

可以通过应用 System.Diagnostics.DebuggerBrowsableAttribute 来阻止调试器自动扩展属性。下面的示例展示了如何将此特性应用于实例属性。

```
Imports System.Diagnostics

Namespace Microsoft.Samples
    Public Class TestClass
        ' [...]

        <DebuggerBrowsable(DebuggerBrowsableState.Never)> _
        Public ReadOnly Property LargeObject() As LargeObject
            Get
                ' Allocate large object
                ' [...]
            End Get
        End Property
    End Class
End Namespace
```

```
using System.Diagnostics;

namespace Microsoft.Samples
{
    class TestClass
    {
        // [...]

        [DebuggerBrowsable(DebuggerBrowsableState.Never)]
        public LargeObject LargeObject
        {
            get
            {
                // Allocate large object
                // [...]
            }
        }
    }
}
```

CA1027:用 FlagsAttribute 标记枚举

2021/11/16 •

	“”
RuleId	CA1027
类别	设计
修复是中断修复还是非中断修复	非中断

原因

枚举的值是两个值的幂或枚举中定义的其他值的组合，且不存在 System.FlagsAttribute 属性。为了减少误报，此规则不对具有连续值的枚举报告冲突。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

枚举是一种值类型，它定义一组相关的已命名常数。如果可以按照有意义的方式组合一个枚举的已命名常数，则对该枚举应用 FlagsAttribute。例如，考虑应用程序中一周中各天的枚举，该枚举会跟踪可用的日期。如果使用包含 FlagsAttribute 的枚举对每个资源的可用性进行编码，则可以表示天数的任意组合。如果没有该属性，则只能表示一周中的某一天。

对于存储可组合枚举的字段，可将单个枚举值视为字段中的位组。因此，有时称此类字段为“位字段”。若要组合枚举值，以存储在位字段中，请使用布尔条件运算符。若要测试位字段，以确定是否存在特定的枚举值，请使用布尔逻辑运算符。若要正确存储位字段并检索组合枚举值，那么枚举中定义的每个值必须是两个值的幂。若非如此，布尔逻辑运算符将无法提取存储在字段中的各个枚举值。

如何解决冲突

若要解决此规则的冲突，请向枚举添加 FlagsAttribute。

何时禁止显示警告

如果不希望组合枚举值，请禁止显示此规则发出的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

在以下示例中，`DaysEnumNeedsFlags` 这一枚举，虽不包含 FlagsAttribute，但满足其使用要求。`ColorEnumShouldNotHaveFlag` 枚举不包括两个值的幂，还错误地指定 FlagsAttribute。这与规则 CA2217：不要使用 FlagsAttribute 标记枚举相冲突。

```
// Violates rule: MarkEnumsWithFlags.
public enum DaysEnumNeedsFlags
{
    None = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    All = Monday | Tuesday | Wednesday | Thursday | Friday
}

// Violates rule: DoNotMarkEnumsWithFlags.
[FlagsAttribute]
public enum ColorEnumShouldNotHaveFlag
{
    None = 0,
    Red = 1,
    Orange = 3,
    Yellow = 4
}
```

相关规则

- CA2217:不要使用 FlagsAttribute 标记枚举

另请参阅

- System.FlagsAttribute

CA1028:枚举存储应为 Int32

2021/11/16 •

RuleId	CA1028
类别	设计
修复是中断修复还是非中断修复	重大

原因

枚举的基础类型不是 System.Int32。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

枚举是一种值类型，它定义一组相关的已命名常数。默认情况下，System.Int32 数据类型用于存储常量值。虽然你可以更改此基础类型，但对于大多数情况，既不需要，也不建议你这样做。使用小于 Int32 的数据类型不会显著提高性能。如果无法使用默认数据类型，则应使用某种符合公共语言规范 (CLS) 的整型类型，例如 Byte、Int16、Int32 或 Int64，以确保枚举的所有值都可以用符合 CLS 的编程语言表示。

如何解决冲突

若要解决此规则的冲突，除非存在大小或兼容性问题，否则请使用 Int32。对于 Int32 不够大而无法保存值的情况，请使用 Int64。如果向后兼容性要求较小的数据类型，请使用 Byte 或 Int16。

何时禁止显示警告

仅当后向兼容性问题需要时，才禁止显示此规则的警告。在应用程序中，未能遵守此规则通常不会导致问题。在需要语言互操作性的库中，未能遵守此规则可能会对用户造成不利影响。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面

运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示了两个不使用建议的基础数据类型的枚举。

```
[Flags]
public enum Days : uint
{
    None = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    All = Monday | Tuesday | Wednesday | Thursday | Friday
}

public enum Color : sbyte
{
    None = 0,
    Red = 1,
    Orange = 3,
    Yellow = 4
}
```

```
<Flags()>
Public Enum Days As UInteger
    None = 0
    Monday = 1
    Tuesday = 2
    Wednesday = 4
    Thursday = 8
    Friday = 16
    All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday
End Enum

Public Enum Color As SByte
    None = 0
    Red = 1
    Orange = 3
    Yellow = 4
End Enum
```

下面的示例将基础数据类型更改为 Int32 来解决先前的冲突。

```
[Flags]
public enum Days : int
{
    None = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    All = Monday | Tuesday | Wednesday | Thursday | Friday
}

public enum Color : int
{
    None = 0,
    Red = 1,
    Orange = 3,
    Yellow = 4
}
```

```
<Flags()>
Public Enum Days As Integer
    None = 0
    Monday = 1
    Tuesday = 2
    Wednesday = 4
    Thursday = 8
    Friday = 16
    All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday
End Enum

Public Enum Color As Integer
    None = 0
    Red = 1
    Orange = 3
    Yellow = 4
End Enum
```

相关规则

- CA1008:枚举应具有零值
- CA1027:用 FlagsAttribute 标记枚举
- CA2217:不要使用 FlagsAttribute 标记枚举
- CA1700:不要命名“Reserved”枚举值
- CA1712:不要将类型名用作枚举值的前缀

另请参阅

- System.Byte
- System.Int16
- System.Int32
- System.Int64

CA1030:在适用处使用事件

2021/11/16 •

■ ID	CA1030
■	设计
▬▬▬	非中断

原因

方法名称以下列项之一开头：

- AddOn
- RemoveOn
- Fire
- Raise

默认情况下，此规则仅查看外部可见的方法，但这是可配置的。

规则说明

该规则检测名称通常用于事件的方法。事件遵循"观察者"或"发布-订阅"设计模式；当必须将一个对象的状态更改传达给其他对象时，它们适用。如果为响应明确定义的状态更改而调用一个方法，则应由事件处理程序调用该方法。调用该方法的对象应引发事件而不是直接调用该方法。

用户界面应用程序中发现了一些常见事件示例，其中用户操作（如单击按钮）会导致执行一段代码。.NET 事件模型并不局限于用户界面。它应在必须将状态更改传达给一个或多个对象的任何位置使用。

如何解决冲突

如果在对象状态发生变化时调用该方法，请考虑更改设计以使用 .NET 事件模型。

何时禁止显示警告

如果该方法不能与 .NET 事件模型一起使用，则禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规

则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

CA1031:不要捕捉一般异常类型

2021/11/16 •

[	"["
RuleId	CA1031
类别	设计
修复是中断修复还是非中断修复	非中断

原因

在 `catch` 语句中捕捉到了 System.Exception 或 System.SystemException 等一般异常，或者已使用一般 catch 子句(如 `catch()`)。

默认情况下，此规则仅标记要捕捉的一般异常类型，但这是可配置的。

规则说明

不应捕捉一般异常。

如何解决冲突

若要解决此规则中的冲突，请捕捉更具体的异常，或者在执行 `catch` 块中的最后一条语句时重新引发一般异常。

何时禁止显示警告

不禁止显示此规则发出的警告。捕获一般异常类型可隐藏库用户的运行时问题，并且可能会使调试变得更加困难。

> **NOTE**
> 从 .NET Framework 4 开始，公共语言运行时 (CLR) 不再提供操作系统和托管代码中发生的损坏状态异常(例如，Windows 中的访问冲突)，然后由托管代码来处理。如果要在 .NET Framework 4 或更高版本中编译某个应用程序，并保留对损坏状态异常的处理，则可将 HandleProcessCorruptedStateExceptionsAttribute 特性应用于负责处理损坏状态异常的方法。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 不允许的异常类型名称

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

不允许的异常类型名称

可配置不允许捕捉哪些异常类型。例如，若要指定规则应使用 `NullReferenceException` 标记 `catch` 处理程序，请将以下键值对添加到项目的 editorconfig 文件中：

```
dotnet_code_quality.CA1031.disallowed_symbol_names = NullReferenceException
```

选项值中允许的类型名称格式(用 `|` 分隔):

- 仅类型名称(包括具有相应名称的所有符号,不考虑包含的类型或命名空间)
- 完全限定的名称,使用符号的文档 ID 格式,前缀为 `T:`。

示例:

`dotnet_code_quality.CA1031.disallowed_symbol_names = ExceptionType`	匹配编译中名为“ExceptionType”的所有符号
`dotnet_code_quality.CA1031.disallowed_symbol_names = ExceptionType1|ExceptionType2`	匹配编译中名为“ExceptionType1”或“ExceptionType2”的所有符号
`dotnet_code_quality.CA1031.disallowed_symbol_names = T:NS.ExceptionType`	将名为“ExceptionType”的特定类型与给定的完全限定名称进行匹配。
`dotnet_code_quality.CA1031.disallowed_symbol_names = T:NS1.ExceptionType1|T:NS1.ExceptionType2`	将名为“ExceptionType1”和“ExceptionType2”的类型与各自的完全限定名称进行匹配

你可以仅为此规则、为所有规则或为此类别(设计)中的所有规则配置这些选项。有关详细信息,请参阅代码质量规则配置选项。

示例

以下示例显示与此规则冲突的类型,以及能正确实现 `catch` 块的类型。

```
Imports System
Imports System.IO

Namespace ca1031

    ' Creates two violations of the rule.
    Public Class GenericExceptionsCaught

        Dim inStream As FileStream
        Dim outStream As FileStream

        Sub New(inFile As String, outFile As String)

            Try
                inStream = File.Open(inFile, FileMode.Open)
            Catch ex As SystemException
                Console.WriteLine("Unable to open {0}.", inFile)
            End Try

            Try
                outStream = File.Open(outFile, FileMode.Open)
            Catch
                Console.WriteLine("Unable to open {0}.", outFile)
            End Try

        End Sub

    End Class

    Public Class GenericExceptionsCaughtFixed

        Dim inStream As FileStream
        Dim outStream As FileStream

        Sub New(inFile As String, outFile As String)

            Try
                inStream = File.Open(inFile, FileMode.Open)

                ' Fix the first violation by catching a specific exception.
            Catch ex As FileNotFoundException
                Console.WriteLine("Unable to open {0}.", inFile)
                ' For functionally equivalent code, also catch the
                ' remaining exceptions that may be thrown by File.Open
            End Try

            Try
                outStream = File.Open(outFile, FileMode.Open)

                ' Fix the second violation by re-throwing the generic
                ' exception at the end of the catch block.
            Catch
                Console.WriteLine("Unable to open {0}.", outFile)
                Throw
            End Try

        End Sub

    End Class

End Namespace
```

```
// Creates two violations of the rule.
public class GenericExceptionsCaught
{
    FileStream inStream;
    FileStream outStream;

    public GenericExceptionsCaught(string inFile, string outFile)
    {
        try
        {
            inStream = File.Open(inFile, FileMode.Open);
        }
        catch (SystemException)
        {
            Console.WriteLine("Unable to open {0}.", inFile);
        }

        try
        {
            outStream = File.Open(outFile, FileMode.Open);
        }
        catch
        {
            Console.WriteLine("Unable to open {0}.", outFile);
        }
    }
}

public class GenericExceptionsCaughtFixed
{
    FileStream inStream;
    FileStream outStream;

    public GenericExceptionsCaughtFixed(string inFile, string outFile)
    {
        try
        {
            inStream = File.Open(inFile, FileMode.Open);
        }

        // Fix the first violation by catching a specific exception.
        catch (FileNotFoundException)
        {
            Console.WriteLine("Unable to open {0}.", inFile);
        };

        // For functionally equivalent code, also catch
        // remaining exceptions that may be thrown by File.Open

        try
        {
            outStream = File.Open(outFile, FileMode.Open);
        }

        // Fix the second violation by rethrowing the generic
        // exception at the end of the catch block.
        catch
        {
            Console.WriteLine("Unable to open {0}.", outFile);
            throw;
        }
    }
}
```

相关规则

CA2200:再次引发以保留堆栈详细信息

CA1032:实现标准异常构造函数

2021/11/16 •

	“”
RuleId	CA1032
类别	设计
修复是中断修复还是非中断修复	非中断

原因

类型扩展了 System.Exception，但未声明所有必需的构造函数。

规则说明

异常类型必须实现以下三个公共构造函数：

- 公共 NewException()
- 公共 NewException(string)
- 公共 NewException(string, Exception)

如果不能提供完整的构造函数集，要正确处理异常将变得比较困难。例如，具有签名 `NewException(string, Exception)` 的构造函数用于创建由其他异常引起的异常。如果没有此构造函数，你无法创建和引发包含内部（嵌套）异常的自定义异常实例，在这种情况下，托管代码应执行此操作。

有关详细信息，请参阅 CA2229：实现序列化构造函数。

如何解决冲突

若要修复此规则的冲突，请将缺少的构造函数添加到异常，并确保它们具有正确的可访问性。

何时禁止显示警告

当冲突是由于对公共构造函数使用不同的访问级别而引起时，可以安全地禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

示例

以下示例包含与此规则冲突的异常类型和正确实现的异常类型。

```
// Violates rule ImplementStandardExceptionConstructors.
public class BadException : Exception
{
    public BadException()
    {
        // Add any type-specific logic, and supply the default message.
    }
}

[Serializable()]
public class GoodException : Exception
{
    public GoodException()
    {
        // Add any type-specific logic, and supply the default message.
    }

    public GoodException(string message) : base(message)
    {
        // Add any type-specific logic.
    }

    public GoodException(string message, Exception innerException) :
       base(message, innerException)
    {
        // Add any type-specific logic for inner exceptions.
    }
}
```

另请参阅

CA2229:实现序列化构造函数

CA1033:接口方法应可由子类型调用

2021/11/16 •

[	"["
RuleId	CA1033
类别	设计
修复是中断修复还是非中断修复	非中断

原因

未密封的外部可见类型提供了显式实现公共接口的方法，但没有提供具有相同名称的其他外部可见方法。

规则说明

考虑到显式实现公共接口方法的基类型。派生自该基类型的类型只能通过引用强制转换到接口的当前实例(C# 中的 `this`)来访问继承接口方法。如果派生类型重新实现(显式)继承接口方法，则无法再访问基实现。通过当前实例引用进行的调用将调用派生实现;这将导致递归和最终的堆栈溢出。

如果提供了外部可见的 `Close()` 或 `System.IDisposable.Dispose(Boolean)` 方法，则此规则不会报告 System.IDisposable.Dispose 的显式实现冲突。

如何解决冲突

若要解决此规则的冲突，请实现新的方法，该方法公开相同的功能，并对派生类型可见或更改为非显示实现。如果可接受中断性变更，还可以选择将类型设为密封类型。

何时禁止显示警告

如果提供了与显式实现的方法具有相同功能但名称不同的外部可见方法，则可以安全地禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

示例

下面的示例显示了一个与此规则冲突的 `ViolatingBase` 类型，以及一个显示冲突修补程序的 `FixedBase` 类型。

```
public interface ITest
{
    void SomeMethod();
}

public class ViolatingBase : ITest
{
    void ITest.SomeMethod()
    {
        // ...
    }
}

public class FixedBase : ITest
{
    void ITest.SomeMethod()
    {
        SomeMethod();
    }

    protected void SomeMethod()
    {
        // ...
    }
}

sealed public class Derived : FixedBase, ITest
{
    public void SomeMethod()
    {
        // The following would cause recursion and a stack overflow.
        // ((ITest)this).SomeMethod();

        // The following is unavailable if derived from ViolatingBase.
        base.SomeMethod();
    }
}
```

另请参阅

- 接口

CA1034:嵌套类型不应是可见的

2021/11/16 •

	“”
RuleId	CA1034
类别	设计
修复是中断修复还是非中断修复	重大

原因

外部可见类型包含外部可见类型声明。嵌套列举、受保护类型和生成器模式不受此规则的限制。

规则说明

嵌套类型是在另一个类型的范围中声明的类型。嵌套类型用于封装包含类型的私有实现详细信息。如果用于此用途，则嵌套类型不应是外部可见的。

不要使用外部可见嵌套类型进行逻辑分组或避免名称冲突；请改为使用命名空间。

嵌套类型包括成员可访问性的概念，对此一些程序员并不清楚了解。

在高级自定义场景中，受保护的类型可用于子类和嵌套类型。

如何解决冲突

如果不打算让嵌套类型在外部可见，请更改该类型的可访问性。否则，请从其父级中删除嵌套类型。如果嵌套的目的是对嵌套类型进行分类，请改为使用命名空间来创建层次结构。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例演示了与规则冲突的类型。

```
public class ParentType
{
    public class NestedType
    {
        public NestedType()
        {
        }
    }

    public ParentType()
    {
        NestedType nt = new NestedType();
    }
}
```

```
Imports System

Namespace ca1034

    Class ParentType

        Public Class NestedType
            Sub New()
            End Sub
        End Class

        Sub New()
        End Sub

    End Class

End Namespace
```

CA1036:重写可比较类型中的方法

2021/11/16 •

RuleId	CA1036
类别	设计
修复是中断修复还是非中断修复	非中断

原因

类型实现 System.IComparable 接口，并且不重写 System.Object.Equals，也不重载表示相等、不等、小于或大于的语言特定运算符。如果类型仅继承接口的实现，则规则不会报告冲突。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

定义自定义排序顺序实现 IComparable 接口的类型。CompareTo 方法返回整数值，该值指示类型的两个实例的正确排序顺序。此规则标识设置排序顺序的类型。设置排序顺序意味着相等、不相等、小于和大于的常规含义不再适用。提供 IComparable 的实现时，通常还必须重写 Equals，以便返回与 CompareTo 一致的值。如果重写 Equals，并使用支持运算符重载的语言进行编码，则还应提供与 Equals 一致的运算符。

如何解决冲突

若要解决此规则的冲突，请重写 Equals。如果编程语言支持运算符重载，请提供以下运算符：

- op_Equality
- op_Inequality
- op_LessThan
- op_GreaterThan

在 C# 中，用来代表这些运算符的令牌如下所示：

```
==
!=
<
>
```

何时禁止显示警告

如果冲突是由缺少运算符引起的，而编程语言也不支持运算符重载，则禁止显示规则 CA1036 中的警告是安全的，这与 Visual Basic 情况一样。如果确定在应用程序上下文中实现运算符没有意义，那么当它在 op_Equality 以外的相等运算符上触发时，也可在该规则中禁止显示警告。但是，如果重写 Object.Equals，则应始终重写 op_Equality 和 == 运算符。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

以下代码包含正确实现 IComparable 的类型。代码注释标识满足与 Equals 和 IComparable 接口相关的各种规则的方法。

```
// Valid ratings are between A and C.
// A is the highest rating; it is greater than any other valid rating.
// C is the lowest rating; it is less than any other valid rating.

public class RatingInformation : IComparable, IComparable<RatingInformation>
{
    public string Rating
    {
        get;
        private set;
    }

    public RatingInformation(string rating)
    {
        if (rating == null)
        {
            throw new ArgumentNullException("rating");
        }

        string v = rating.ToUpper(CultureInfo.InvariantCulture);
        if (v.Length != 1 || string.Compare(v, "C", StringComparison.Ordinal) > 0 || string.Compare(v, "A",
StringComparison.Ordinal) < 0)
        {
            throw new ArgumentException("Invalid rating value was specified.", "rating");
        }

        Rating = v;
    }

    public int CompareTo(object obj)
    {
        if (obj == null)
        {
            return 1;
        }

        RatingInformation other = obj as RatingInformation; // avoid double casting
        if (other == null)
        {
```

```
            throw new ArgumentException("A RatingInformation object is required for comparison.", "obj");
        }

        return CompareTo(other);
    }

    public int CompareTo(RatingInformation other)
    {
        if (other is null)
        {
            return 1;
        }

        // Ratings compare opposite to normal string order,
        // so reverse the value returned by String.CompareTo.
        return -string.Compare(this.Rating, other.Rating, StringComparison.OrdinalIgnoreCase);
    }

    public static int Compare(RatingInformation left, RatingInformation right)
    {
        if (object.ReferenceEquals(left, right))
        {
            return 0;
        }
        if (left is null)
        {
            return -1;
        }
        return left.CompareTo(right);
    }

    // Omitting Equals violates rule: OverrideMethodsOnComparableTypes.
    public override bool Equals(object obj)
    {
        RatingInformation other = obj as RatingInformation; //avoid double casting
        if (other is null)
        {
            return false;
        }
        return this.CompareTo(other) == 0;
    }

    // Omitting getHashCode violates rule: OverrideGetHashCodeOnOverridingEquals.
    public override int GetHashCode()
    {
        char[] c = this.Rating.ToCharArray();
        return (int)c[0];
    }

    // Omitting any of the following operator overloads
    // violates rule: OverrideMethodsOnComparableTypes.
    public static bool operator ==(RatingInformation left, RatingInformation right)
    {
        if (left is null)
        {
            return right is null;
        }
        return left.Equals(right);
    }
    public static bool operator !=(RatingInformation left, RatingInformation right)
    {
        return !(left == right);
    }
    public static bool operator <(RatingInformation left, RatingInformation right)
    {
        return (Compare(left, right) < 0);
    }
    public static bool operator >(RatingInformation left, RatingInformation right)
    {
```

```
        return (Compare(left, right) > 0);
    }
}
```

以下应用程序代码测试前面显示的 IComparable 实现的行为。

```
public class Test
{
    public static void Main1036(string[] args)
    {
        if (args.Length < 2)
        {
            Console.WriteLine("usage - TestRatings  string 1 string2");
            return;
        }
        RatingInformation r1 = new RatingInformation(args[0]);
        RatingInformation r2 = new RatingInformation(args[1]);
        string answer;

        if (r1.CompareTo(r2) > 0)
            answer = "greater than";
        else if (r1.CompareTo(r2) < 0)
            answer = "less than";
        else
            answer = "equal to";

        Console.WriteLine("{0} is {1} {2}", r1.Rating, answer, r2.Rating);
    }
}
```

另请参阅

- System.IComparable
- System.Object.Equals
- 相等运算符

CA1040:避免使用空接口

2021/11/16 •

[	"["
RuleId	CA1040
类别	设计
修复是中断修复还是非中断修复	重大

原因

接口不声明任何成员，或实现两个或两个以上其他接口。

默认情况下，此规则仅查看外部可见的接口，但这是可配置的。

规则说明

接口定义提供某个行为或使用协定的成员。接口所描述的功能可以被任何类型采用，而不管该类型出现在继承层次结构中的哪个位置。类型通过实现接口的成员来实现接口。空接口不定义任何成员。因此，它不定义可实现的协定。

如果设计包含期望实现类型的空接口，则可能会将接口用作标记或标识一组类型的方式。如果在运行时执行此标识，则实现此目的的正确方法是使用自定义特性。使用或不使用该特性，或使用该特性的属性，以标识目标类型。如果必须在编译时执行标识，则可以使用空接口。

如何解决冲突

删除接口或向其添加成员。如果要使用空接口来标记一组类型，请将接口替换为自定义特性。

何时禁止显示警告

当接口用于在编译时标识一组类型时，可以安全地禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面

运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示了空接口。

```
// Violates rule
public interface IBadInterface
{
}
```

```
' Violates rule
Public Interface IBadInterface
End Interface
```

CA1041:提供 ObsoleteAttribute 消息

2021/11/16 •

[	"["
RuleId	CA1041
类别	设计
修复是中断修复还是非中断修复	非中断

原因

类型和成员使用了未指定其 System.ObsoleteAttribute.Message 属性的 System.ObsoleteAttribute 特性进行标记。

默认情况下，此规则仅查看外部可见的类型和成员，但这是可配置的。

规则说明

ObsoleteAttribute 用于标记已弃用的库类型和成员。库使用者应避免使用任何标记为已过时的类型或成员。这是因为它可能不受支持，最终将从库的更高版本中删除。编译使用 ObsoleteAttribute 进行标记的类型和成员时，将显示此特性的 Message 属性。这将为用户提供有关已过时的类型或成员的信息。此信息通常包括库设计人员还将支持已过时类型或成员的时长以及要使用的首选替换项。

如何解决冲突

若要修复此规则的冲突，请将 `message` 参数添加到 ObsoleteAttribute 构造函数。

何时禁止显示警告

不要禁止显示此规则的警告，因为 Message 属性提供了有关已过时类型或成员的关键信息。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

以下示例显示了具有正确声明的 ObsoleteAttribute 的已过时成员。

```
[ObsoleteAttribute("This property is obsolete and will be removed in a " +
"future version. Use the FullName property instead.", false)]
public string Name
{
    get => "Name";
}
```

```
Imports System

Namespace ca1041

    Public Class ObsoleteAttributeOnMember

        <ObsoleteAttribute("This property is obsolete and will " &
             "be removed in a future version. Use the FirstName " &
             "and LastName properties instead.", False)>
        ReadOnly Property Name As String
            Get
                Return "Name"
            End Get
        End Property

        ReadOnly Property FirstName As String
            Get
                Return "FirstName"
            End Get
        End Property

        ReadOnly Property LastName As String
            Get
                Return "LastName"
            End Get
        End Property

    End Class

End Namespace
```

另请参阅

- System.ObsoleteAttribute

CA1043:将整型或字符串参数用于索引器

2021/11/16 •

■ ID	CA1043
■	设计
■■■■■	重大

原因

类型包含索引器，该索引器使用的索引类型不是 System.Int32、System.Int64、System.Object 或 System.String。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

索引器(即索引属性)应将整数或字符串类型用于索引。这些类型通常用于为数据结构编制索引，并可提高库的可用性。应仅限于在设计时无法指定特定整数或字符串类型的情况下使用 Object 类型。如果设计需要其他类型的索引，请重新考虑该类型是否表示逻辑数据存储。如果它不表示逻辑数据存储，请使用方法。

如何解决冲突

若要解决此规则的冲突，请将索引更改为整数或字符串类型，或者使用方法代替索引器。

何时禁止显示警告

仅在仔细考虑了对非标准索引器的需求之后，才能禁止显示此规则的警告。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

以下示例显示了使用 Int32 索引的索引器。

```
string[] Month = new string[] { "Jan", "Feb", "..." };

public string this[int index]
{
    get => Month[index];
}
```

```
Private month() As String = {"Jan", "Feb", "..."}

Default ReadOnly Property Item(index As Integer) As String
    Get
        Return month(index)
    End Get
End Property
```

相关规则

- CA1024:在适用处使用属性

CA1044:属性不应是只写的

2021/11/16 •

	[
■ ID	CA1044
■	设计
▬▬▬	重大

原因

属性具有 set 访问器，但不具有 get 访问器。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

Get 访问器提供对属性的读取访问权限，而 set 访问器提供写入访问权限。虽然可以接受且经常需要使用只读属性，但设计准则禁止使用只写属性。这是因为允许用户设置值但又禁止该用户查看这个值不能提供任何安全性。而且，如果没有读访问，将无法查看共享对象的状态，使其用处受到限制。

如何解决冲突

若要解决此规则的冲突，请将 get 访问器添加到属性。或者，如果需要只写属性的行为，请考虑将该属性转换为方法。

何时禁止显示警告

建议不要禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

在下面的示例中，`BadClassWithWriteOnlyProperty` 是一个具有只写属性的类型。`GoodClassWithReadWriteProperty` 包含更正后的代码。

```
Imports System

Namespace ca1044

    Public Class BadClassWithWriteOnlyProperty

        Dim someName As String

        ' Violates rule PropertiesShouldNotBeWriteOnly.
        WriteOnly Property Name As String
            Set
                someName = Value
            End Set
        End Property

    End Class

    Public Class GoodClassWithReadWriteProperty

        Property Name As String

    End Class

End Namespace
```

```
public class BadClassWithWriteOnlyProperty
{
    string _someName;

    // Violates rule PropertiesShouldNotBeWriteOnly.
    public string Name
    {
        set
        {
            _someName = value;
        }
    }
}

public class GoodClassWithReadWriteProperty
{
    public string Name { get; set; }
}
```

CA1045:不要通过引用来传递类型

2021/11/16 •

	[
■ ID	CA1045
■	设计
▬▬	重大

原因

公共类型中的公共或受保护方法有一个 `ref` 参数，该参数采用基元类型、引用类型或不属于内置类型的值类型。

规则说明

按引用(使用 `out` 或 `ref`)传递类型要求具有使用指针的经验，了解值类型和引用类型的不同之处，以及能处理具有多个返回值的方法。另外，`out` 和 `ref` 参数之间的区别并未得到广泛了解。

如果引用类型"按引用"传递，则该方法会使用参数来返回对象的不同实例。(按引用传递引用类型也称为使用双指针、指向指针的指针或双间接。)使用"按值"传递这一默认调用约定，采用引用类型的参数已经收到指向对象的指针。指针(而不是它指向的对象)按值传递。按值传递表示方法不能更改指针以使其指向引用类型的新实例，但是它可以更改它所指向的对象的内容。对于大多数应用程序，这就足够了，并生成了所需的行为。

如果方法必须返回不同的实例，请使用该方法的返回值来实现此目的。有关对字符串执行操作并返回字符串的新实例的各种方法，请参阅 System.String 类。通过使用此模型，调用方可决定是否保留原始对象。

尽管返回值很常见且被大量使用，但正确应用 `out` 和 `ref` 参数需要中间设计和编码技能。为一般用户进行设计的库架构师不应指望用户能熟练运用 `out` 或 `ref` 参数。

> **NOTE**
> 如果使用的参数是大型结构，则在按值传递时，复制这些结构所需的其他资源可能会对性能产生影响。在这些情况下，可考虑使用 `ref` 或 `out` 参数。

如何解决冲突

要修复由值类型引起的此规则的冲突，需使方法返回对象作为其返回值。如果该方法必须返回多个值，请重新设计它以返回保存值的对象的单个实例。

要修复由引用类型引起的此规则的冲突，需确保所需的行为是否为返回引用的新实例。如果是，则该方法应使用其返回值来执行此操作。

何时禁止显示警告

可禁止显示此规则发出的警告；但这种设计可能会引发可用性问题。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例 1

以下库显示生成对用户反馈响应的类的两个实现。第一个实现（`BadRefAndOut`）强制库用户管理三个返回值。第二个实现（`RedesignedRefAndOut`）通过返回容器类（`ReplyData`）的实例来简化用户体验，该容器类将数据作为单个单元进行管理。

```
public enum Actions
{
    Unknown,
    Discard,
    ForwardToManagement,
    ForwardToDeveloper
}

public enum TypeOfFeedback
{
    Complaint,
    Praise,
    Suggestion,
    Incomprehensible
}

public class BadRefAndOut
{
    // Violates rule: DoNotPassTypesByReference.

    public static bool ReplyInformation(TypeOfFeedback input,
       out string reply, ref Actions action)
    {
        bool returnReply = false;
        string replyText = "Your feedback has been forwarded " +
                           "to the product manager.";

        reply = String.Empty;
        switch (input)
        {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise:
                action = Actions.ForwardToManagement;
                reply = "Thank you. " + replyText;
                returnReply = true;
```

```
                break;
            case TypeOfFeedback.Suggestion:
                action = Actions.ForwardToDeveloper;
                reply = replyText;
                returnReply = true;
                break;
            case TypeOfFeedback.Incomprehensible:
            default:
                action = Actions.Discard;
                returnReply = false;
                break;
        }
        return returnReply;
    }
}

// Redesigned version does not use out or ref parameters;
// instead, it returns this container type.

public class ReplyData
{
    string reply;
    Actions action;
    bool returnReply;

    // Constructors.
    public ReplyData()
    {
        this.reply = String.Empty;
        this.action = Actions.Discard;
        this.returnReply = false;
    }

    public ReplyData(Actions action, string reply, bool returnReply)
    {
        this.reply = reply;
        this.action = action;
        this.returnReply = returnReply;
    }

    // Properties.
    public string Reply { get { return reply; } }
    public Actions Action { get { return action; } }

    public override string ToString()
    {
        return String.Format("Reply: {0} Action: {1} return? {2}",
           reply, action.ToString(), returnReply.ToString());
    }
}

public class RedesignedRefAndOut
{
    public static ReplyData ReplyInformation(TypeOfFeedback input)
    {
        ReplyData answer;
        string replyText = "Your feedback has been forwarded " +
           "to the product manager.";

        switch (input)
        {
            case TypeOfFeedback.Complaint:
            case TypeOfFeedback.Praise:
                answer = new ReplyData(
                   Actions.ForwardToManagement,
                   "Thank you. " + replyText,
                   true);
                break;
            case TypeOfFeedback.Suggestion:
```

```
                answer = new ReplyData(
                   Actions.ForwardToDeveloper,
                   replyText,
                   true);
                break;
            case TypeOfFeedback.Incomprehensible:
            default:
                answer = new ReplyData();
                break;
        }
        return answer;
    }
}
```

示例 2

以下应用程序说明了用户的体验。对重新设计的库的调用（`UseTheSimplifiedClass` 方法）更简单，并且该方法返回的信息非常易于管理。这两个方法的输出是相同的。

```
public class UseComplexMethod
{
    static void UseTheComplicatedClass()
    {
        // Using the version with the ref and out parameters.
        // You do not have to initialize an out parameter.

        string[] reply = new string[5];

        // You must initialize a ref parameter.
        Actions[] action = {Actions.Unknown,Actions.Unknown,
                            Actions.Unknown,Actions.Unknown,
                            Actions.Unknown,Actions.Unknown};
        bool[] disposition = new bool[5];
        int i = 0;

        foreach (TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
        {
            // The call to the library.
            disposition[i] = BadRefAndOut.ReplyInformation(
               t, out reply[i], ref action[i]);
            Console.WriteLine("Reply: {0} Action: {1}  return? {2} ",
               reply[i], action[i], disposition[i]);
            i++;
        }
    }

    static void UseTheSimplifiedClass()
    {
        ReplyData[] answer = new ReplyData[5];
        int i = 0;
        foreach (TypeOfFeedback t in Enum.GetValues(typeof(TypeOfFeedback)))
        {
            // The call to the library.
            answer[i] = RedesignedRefAndOut.ReplyInformation(t);
            Console.WriteLine(answer[i++]);
        }
    }

    public static void Main1045()
    {
        UseTheComplicatedClass();

        // Print a blank line in output.
        Console.WriteLine("");

        UseTheSimplifiedClass();
    }
}
```

示例 3

下面的示例库说明了如何使用引用类型的 `ref` 参数，并演示了实现此功能的更好方法。

```
public class ReferenceTypesAndParameters
{
    // The following syntax will not work. You cannot make a
    // reference type that is passed by value point to a new
    // instance. This needs the ref keyword.

    public static void BadPassTheObject(string argument)
    {
        argument = argument + " ABCDE";
    }

    // The following syntax will work, but is considered bad design.
    // It reassigns the argument to point to a new instance of string.
    // Violates rule DoNotPassTypesByReference.

    public static void PassTheReference(ref string argument)
    {
        argument = argument + " ABCDE";
    }

    // The following syntax will work and is a better design.
    // It returns the altered argument as a new instance of string.

    public static string BetterThanPassTheReference(string argument)
    {
        return argument + " ABCDE";
    }
}
```

示例 4

下面的应用程序调用库中的每个方法来演示行为。

```
public class Test
{
    public static void Main1045()
    {
        string s1 = "12345";
        string s2 = "12345";
        string s3 = "12345";

        Console.WriteLine("Changing pointer - passed by value:");
        Console.WriteLine(s1);
        ReferenceTypesAndParameters.BadPassTheObject(s1);
        Console.WriteLine(s1);

        Console.WriteLine("Changing pointer - passed by reference:");
        Console.WriteLine(s2);
        ReferenceTypesAndParameters.PassTheReference(ref s2);
        Console.WriteLine(s2);

        Console.WriteLine("Passing by return value:");
        s3 = ReferenceTypesAndParameters.BetterThanPassTheReference(s3);
        Console.WriteLine(s3);
    }
}
```

该示例产生下面的输出：

```
Changing pointer - passed by value:
12345
12345
Changing pointer - passed by reference:
12345
12345 ABCDE
Passing by return value:
12345 ABCDE
```

相关规则

CA1021:避免使用 out 参数

CA1046:不要对引用类型重载相等运算符

2021/11/16 •

	[
■ ID	CA1046
■	设计
▬▬▬	重大

原因

公共引用类型或嵌套公共引用类型重载相等运算符。

规则说明

对于引用类型，相等运算符的默认实现几乎始终是正确的。默认情况下，仅当两个引用指向同一对象时，它们才相等。

如何解决冲突

若要解决此规则的冲突，请删除相等运算符的实现。

何时禁止显示警告

当引用类型的行为与内置值类型相同时，可禁止显示此规则的警告。如果对该类型的实例执行加法或减法有意义，则实现相等运算符并禁止显示此冲突可能正确。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项(设计)。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例 1

下面的示例演示了在比较两个引用时的默认行为。

```
public class MyReferenceType
{
    private int a, b;
    public MyReferenceType(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    public override string ToString()
    {
        return String.Format("({0},{1})", a, b);
    }
}
```

示例 2

下面的应用程序比较了一些引用。

```
public class ReferenceTypeEquality
{
    public static void Main1046()
    {
        MyReferenceType a = new MyReferenceType(2, 2);
        MyReferenceType b = new MyReferenceType(2, 2);
        MyReferenceType c = a;

        Console.WriteLine("a = new {0} and b = new {1} are equal? {2}", a, b, a.Equals(b) ? "Yes" : "No");
        Console.WriteLine("c and a are equal? {0}", c.Equals(a) ? "Yes" : "No");
        Console.WriteLine("b and a are == ? {0}", b == a ? "Yes" : "No");
        Console.WriteLine("c and a are == ? {0}", c == a ? "Yes" : "No");
    }
}
```

该示例产生下面的输出：

```
a = new (2,2) and b = new (2,2) are equal? No
c and a are equal? Yes
b and a are == ? No
c and a are == ? Yes
```

请参阅

- System.Object.Equals
- 相等运算符

CA1047:不要在密封类型中声明受保护的成员

2021/11/16 •

	[
■ ID	CA1047
■	设计
▬▬▬	非中断

原因

公共类型是 `sealed`（在 Visual basic 中为 `NotInheritable`），并声明了一个受保护的成员或受保护的嵌套类型。此规则不报告 Finalize 方法的冲突，这些方法必须遵循此模式。

规则说明

类型声明受保护的成员，使继承类型可以访问或重写该成员。按照定义，不能从密封类型继承，这表示不能调用密封类型上的受保护方法。

对于此错误，C# 编译器会发出警告。

如何解决冲突

若要解决此规则的冲突，请将成员的访问级别更改为专用，或使该类型可继承。

何时禁止显示警告

不禁止显示此规则发出的警告。使类型保持当前状态可能会导致维护问题，而且不会带来任何好处。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示了与此规则发生冲突的类型。

```
public sealed class SealedClass
{
    protected void ProtectedMethod(){}
}
```

```
Public NotInheritable Class BadSealedType
    Protected  Sub MyMethod
    End Sub
End Class
```

CA1050:在命名空间中声明类型

2021/11/16 •

■ ID	CA1050
■	设计
▬▬	重大

原因

在命名的命名称空间范围之外定义公共类型或受保护类型。

规则说明

应在命名空间内声明类型以避免名称冲突，并作为一种在对象层次结构中组织相关类型的方式。任何命名的命名称空间之外的类型均位于无法在代码中引用的全局命名空间中。

如何解决冲突

若要修复与此规则的冲突，请将类型置于命名空间中。

何时禁止显示警告

虽然根本不必禁止显示此规则中的警告，但当程序集绝不会与其他组件一起使用时，可以禁止显示。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

示例 1

以下示例显示在命名空间外错误声明类型的库，以及在名称空间中声明同一名称的类型。

```
// Violates rule: DeclareTypesInNamespaces.
using System;

public class Test
{
    public override string ToString()
    {
        return "Test does not live in a namespace!";
    }
}

namespace ca1050
{
    public class Test
    {
        public override string ToString()
        {
            return "Test lives in a namespace!";
        }
    }
}
```

```
' Violates rule: DeclareTypesInNamespaces.
Public Class Test

    Public Overrides Function ToString() As String
        Return "Test does not live in a namespace!"
    End Function

End Class

Namespace ca1050

    Public Class Test

        Public Overrides Function ToString() As String
            Return "Test lives in a namespace!"
        End Function

    End Class

End Namespace
```

示例 2

以下应用程序使用之前定义的库。当命名空间未限定名称 `Test` 时，将创建命名空间之外声明的类型。若要访问在命名空间内声明的 `Test` 类型，需要命名空间名称。

```
public class MainHolder
{
    public static void Main1050()
    {
        Test t1 = new Test();
        Console.WriteLine(t1.ToString());

        ca1050.Test t2 = new ca1050.Test();
        Console.WriteLine(t2.ToString());
    }
}
```

```
Public Class MainHolder

    Public Shared Sub Main1050()
        Dim t1 As New Test()
        Console.WriteLine(t1.ToString())

        Dim t2 As New ca1050.Test()
        Console.WriteLine(t2.ToString())
    End Sub

End Class
```

CA1051:不要声明可见实例字段

2021/11/16 •

	▮
■ ID	CA1051
■	设计
▬▬	重大

原因

类型包含非私有实例字段。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

字段的主要用途应是作为实现的详细信息。字段应为 `private` 或 `internal`，并应通过使用属性公开这些字段。在访问某个字段时，可轻松访问属性，而属性访问器中的代码可在扩展类型功能时更改，而不会引入重大更改。

仅返回私有或内部字段的值的属性，经过优化后，可在与访问字段相同的情况上执行；使用外部可见字段而不是属性时，所带来的性能提升最小。"外部可见"是指 `public`、`protected` 和 `protected internal`（在 Visual Basic中为 `Public`、`Protected` 和 `Protected Friend`）可访问性级别。

此外，链接要求无法保护公共字段。（链接要求不适用于 .NET Core 应用。）

如何解决冲突

要解决此规则的冲突，请将字段设置为 `private` 或 `internal`，并使用外部可见的属性将其公开。

何时禁止显示警告

仅当确定使用者需要直接访问字段时，才禁止显示此警告。对于大多数应用程序，公开的字段不会提供性能或优于属性的可维护权益。

在以下情况下，使用者可能需要字段访问权限：

- ASP.NET Web Forms 中的内容控件。
- 目标平台使用 `ref`（例如 WPF 和 UWP 的模型-视图-视图模型 (MVVM) 框架）来修改字段。

包含或排除 API

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 排除结构

你可以仅为此规则、为所有规则或为此类别（设计）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

排除结构

可从分析中排除 `struct`（在 Visual Basic 中为 `Structure`）字段。

```
dotnet_code_quality.ca1051.exclude_structs = true
```

示例

下面的示例显示了与此规则发生冲突的类型（`BadPublicInstanceFields`）。`GoodPublicInstanceFields` 显示更正后的代码。

```
public class BadPublicInstanceFields
{
    // Violates rule DoNotDeclareVisibleInstanceFields.
    public int instanceData = 32;
}

public class GoodPublicInstanceFields
{
    private int instanceData = 32;

    public int InstanceData
    {
        get { return instanceData; }
        set { instanceData = value; }
    }
}
```

另请参阅

- 链接需求

CA1052：静态容器类型应是 Static 或 NotInheritable

2021/11/16 •

■ ID	CA1052
■	设计
■■■■■	重大

原因

非抽象类型只包含静态成员（可能的默认构造函数除外），而且没有使用 static 或 Shared 修饰符进行声明。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

规则 CA1052 假定仅包含不设计为继承的静态成员的类型，因为该类型不提供任何可在派生类型中重写的功能。未计划继承的类型应该用 C# 中的 `static` 修饰符进行标记，以便禁止其作为基类型使用。此外，应删除其默认构造函数。在 Visual Basic 中，类应转换为模块。

对于抽象类或具有基类的类，不会触发此规则。但是，对于支持空接口的类，则会触发此规则。

> **NOTE**
> 在该规则的最新分析器实现中，还包含规则 CA1053 的功能。

如何解决冲突

若要解决此规则的冲突，请将类型标记为 `static`，并删除默认构造函数 (C#)，或将其转换为模块 (Visual Basic)。

何时禁止显示警告

在以下情况下，可以禁止显示冲突：

- 类型设计为继承。缺少 `static` 修饰符，表明该类型可用作基类型。
- 此类型不能用作类型参数。静态类型不能用作类型参数。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

冲突示例

以下示例显示与此规则冲突的类型：

```
public class StaticMembers
{
    public static int SomeProperty { get; set; }
    public static void SomeMethod() { }
}
```

```
Imports System

Namespace ca1052

    Public Class StaticMembers

        Shared Property SomeProperty As Integer

        Private Sub New()
        End Sub

        Shared Sub SomeMethod()
        End Sub

    End Class

End Namespace
```

使用静态修改器来解决

以下示例演示如何在 C# 中使用 `static` 修饰符来标记类型，以解决此规则的冲突：

```
public static class StaticMembers
{
    public static int SomeProperty { get; set; }
    public static void SomeMethod() { }
}
```

CA1053：静态容器类型不应具有默认构造函数

2021/11/16 •

	▮
■ ID	CA1053
■	设计
▬▬▬	重大

NOTE

规则 CA1053 仅适用于旧版 Visual Studio 代码分析。在 .NET 代码质量分析器中，该规则已合并到规则 CA1052：静态容器类型应为 Static 或 NotInheritable 中。

原因

公共或嵌套公共类型只声明了静态成员，但具有默认构造函数。

规则说明

由于调用静态成员不需要类型的实例，因此没必要使用默认构造函数。另外，由于类型不具有非静态成员，因此创建实例不提供对任何类型成员的访问。

如何解决冲突

若要解决此规则的冲突，请删除默认构造函数。

何时禁止显示警告

不禁止显示此规则发出的警告。如果存在默认构造函数，则表明该类型不是静态类型。

CA1054:URI 参数不应为字符串

2021/11/16 •

	‹
■ ID	CA1054
■	设计
▬▬▬	重大

原因

类型声明一个方法，该方法具有名称中包含“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”的字符串参数，且类型未声明采用 System.Uri 参数的相应重载。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

此规则根据 Camel 大小写约定将参数名称拆分为标记，并检查每个标记是否等于“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”。如果存在匹配项，此规则假定该参数表示统一资源标识符 (URI)。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。如果某方法采用 URI 的字符串表示形式，则应提供采用 Uri 类的实例的相应重载，该类以安全的方式提供这些服务。

如何解决冲突

若要解决此规则的冲突，请将参数更改为 Uri 类型；这是一项中断性变更。或者，提供采用 Uri 参数的方法的重载；这是一项非中断性变更。

何时禁止显示警告

如果该参数不表示 URL，则可以安全地禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例显示了一个与此规则冲突的 `ErrorProne` 类型，以及一个符合此规则的 `SaferWay` 类型。

```
public class ErrorProne
{
    // Violates rule UriPropertiesShouldNotBeStrings.
    public string SomeUri { get; set; }

    // Violates rule UriParametersShouldNotBeStrings.
    public void AddToHistory(string uriString) { }

    // Violates rule UriReturnValuesShouldNotBeStrings.
    public string GetRefererUri(string httpHeader)
    {
        return "http://www.adventure-works.com";
    }
}

public class SaferWay
{
    // To retrieve a string, call SomeUri.ToString().
    // To set using a string, call SomeUri = new Uri(string).
    public Uri SomeUri { get; set; }

    public void AddToHistory(string uriString)
    {
        // Check for UriFormatException.
        AddToHistory(new Uri(uriString));
    }

    public void AddToHistory(Uri uriType) { }

    public Uri GetRefererUri(string httpHeader)
    {
        return new Uri("http://www.adventure-works.com");
    }
}
```

```
Imports System

Namespace ca1054

    Public Class ErrorProne

        ' Violates rule UriPropertiesShouldNotBeStrings.
        Property SomeUri As String

        ' Violates rule UriParametersShouldNotBeStrings.
        Sub AddToHistory(uriString As String)
        End Sub

        ' Violates rule UriReturnValuesShouldNotBeStrings.
        Function GetRefererUri(httpHeader As String) As String
            Return "http://www.adventure-works.com"
        End Function

    End Class

    Public Class SaferWay

        ' To retrieve a string, call SomeUri.ToString().
        ' To set using a string, call SomeUri = New Uri(string).
        Property SomeUri As Uri

        Sub AddToHistory(uriString As String)
            ' Check for UriFormatException.
            AddToHistory(New Uri(uriString))
        End Sub

        Sub AddToHistory(uriString As Uri)
        End Sub

        Function GetRefererUri(httpHeader As String) As Uri
            Return New Uri("http://www.adventure-works.com")
        End Function

    End Class

End Namespace
```

相关规则

- CA1056:URI 属性不应是字符串
- CA1055:URI 返回值不应是字符串
- CA2234:传递 System.Uri 对象，而不传递字符串

CA1055:URI 返回值不应是字符串

2021/11/16 •

	[
■ ID	CA1055
■	设计
▬▬▬	重大

原因

方法名称包含“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”，且方法返回一个字符串。

默认情况下，此规则仅查看外部可见的方法，但这是可配置的。

规则说明

此规则根据 Pascal 大小写约定将方法名称拆分为标记，并检查每个标记是否等于“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”。如果存在匹配项，则规则假定该方法返回统一资源标识符 (URI)。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。

如何解决冲突

若要解决与此规则的冲突，请将返回类型更改为 Uri。

何时禁止显示警告

如果返回值不表示 URI，则可以安全地禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例显示了一个与此规则冲突的 `ErrorProne` 类型，以及一个符合此规则的 `SaferWay` 类型。

```
public class ErrorProne
{
    // Violates rule UriPropertiesShouldNotBeStrings.
    public string SomeUri { get; set; }

    // Violates rule UriParametersShouldNotBeStrings.
    public void AddToHistory(string uriString) { }

    // Violates rule UriReturnValuesShouldNotBeStrings.
    public string GetReferrerUri(string httpHeader)
    {
        return "http://www.adventure-works.com";
    }
}

public class SaferWay
{
    // To retrieve a string, call SomeUri.ToString().
    // To set using a string, call SomeUri = new Uri(string).
    public Uri SomeUri { get; set; }

    public void AddToHistory(string uriString)
    {
        // Check for UriFormatException.
        AddToHistory(new Uri(uriString));
    }

    public void AddToHistory(Uri uriType) { }

    public Uri GetReferrerUri(string httpHeader)
    {
        return new Uri("http://www.adventure-works.com");
    }
}
```

```
Imports System

Namespace ca1055

    Public Class ErrorProne

        ' Violates rule UriPropertiesShouldNotBeStrings.
        Property SomeUri As String

        ' Violates rule UriParametersShouldNotBeStrings.
        Sub AddToHistory(uriString As String)
        End Sub

        ' Violates rule UriReturnValuesShouldNotBeStrings.
        Function GetRefererUri(httpHeader As String) As String
            Return "http://www.adventure-works.com"
        End Function

    End Class

    Public Class SaferWay

        ' To retrieve a string, call SomeUri.ToString().
        ' To set using a string, call SomeUri = New Uri(string).
        Property SomeUri As Uri

        Sub AddToHistory(uriString As String)
            ' Check for UriFormatException.
            AddToHistory(New Uri(uriString))
        End Sub

        Sub AddToHistory(uriString As Uri)
        End Sub

        Function GetRefererUri(httpHeader As String) As Uri
            Return New Uri("http://www.adventure-works.com")
        End Function

    End Class

End Namespace
```

相关规则

- CA1056:URI 属性不应是字符串
- CA1054:URI 参数不应为字符串
- CA2234:传递 System.Uri 对象，而不传递字符串

CA1056:URI 属性不应是字符串

2021/11/16 •

	▮
■ ID	CA1056
■	设计
▬▬	重大

原因

类型声明名称包含“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”的字符串属性。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

此规则根据 Pascal 大小写约定将属性名称拆分为标记，并检查每个标记是否等于“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”。如果存在匹配项，此规则假定该属性表示统一资源标识符 (URI)。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。System.Uri 类以一种安全的方式提供这些服务。

如何解决冲突

若要解决此规则的冲突，请将该属性更改为 Uri 类型。

何时禁止显示警告

如果该属性不表示 URL，则可以安全地禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例显示了一个与此规则冲突的 `ErrorProne` 类型，以及一个符合此规则的 `SaferWay` 类型。

```
public class ErrorProne
{
    // Violates rule UriPropertiesShouldNotBeStrings.
    public string SomeUri { get; set; }

    // Violates rule UriParametersShouldNotBeStrings.
    public void AddToHistory(string uriString) { }

    // Violates rule UriReturnValuesShouldNotBeStrings.
    public string GetRefererUri(string httpHeader)
    {
        return "http://www.adventure-works.com";
    }
}

public class SaferWay
{
    // To retrieve a string, call SomeUri.ToString().
    // To set using a string, call SomeUri = new Uri(string).
    public Uri SomeUri { get; set; }

    public void AddToHistory(string uriString)
    {
        // Check for UriFormatException.
        AddToHistory(new Uri(uriString));
    }

    public void AddToHistory(Uri uriType) { }

    public Uri GetRefererUri(string httpHeader)
    {
        return new Uri("http://www.adventure-works.com");
    }
}
```

```
Imports System

Namespace ca1056

    Public Class ErrorProne

        ' Violates rule UriPropertiesShouldNotBeStrings.
        Property SomeUri As String

        ' Violates rule UriParametersShouldNotBeStrings.
        Sub AddToHistory(uriString As String)
        End Sub

        ' Violates rule UriReturnValuesShouldNotBeStrings.
        Function GetRefererUri(httpHeader As String) As String
            Return "http://www.adventure-works.com"
        End Function

    End Class

    Public Class SaferWay

        ' To retrieve a string, call SomeUri.ToString().
        ' To set using a string, call SomeUri = New Uri(string).
        Property SomeUri As Uri

        Sub AddToHistory(uriString As String)
            ' Check for UriFormatException.
            AddToHistory(New Uri(uriString))
        End Sub

        Sub AddToHistory(uriString As Uri)
        End Sub

        Function GetRefererUri(httpHeader As String) As Uri
            Return New Uri("http://www.adventure-works.com")
        End Function

    End Class

End Namespace
```

相关规则

- CA1054:URI 参数不应为字符串
- CA1055:URI 返回值不应是字符串
- CA2234:传递 System.Uri 对象，而不传递字符串

CA1058:类型不应扩展某些基类型

2021/11/16 •

■ ID	CA1058
■	设计
■■■■■	重大

原因

类型扩展了以下基类型之一：

- System.ApplicationException
- System.Xml.XmlDocument
- System.Collections.CollectionBase
- System.Collections.DictionaryBase
- System.Collections.Queue
- System.Collections.ReadOnlyCollectionBase
- System.Collections.SortedList
- System.Collections.Stack

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

异常应派生自 System.Exception 或其在 System 命名空间中的子类之一。

如果要创建基础对象模型或数据源的 XML 视图，请勿创建 XmlDocument 的子类。

非泛型集合

尽可能使用和/或扩展泛型集合。除非之前已发布过代码，否则请勿在代码中扩展非泛型集合。

错误用法示例

```
public class MyCollection : CollectionBase
{
}

public class MyReadOnlyCollection : ReadOnlyCollectionBase
{
}
```

正确用法示例

```
public class MyCollection : Collection<T>
{
}

public class MyReadOnlyCollection : ReadOnlyCollection<T>
{
}
```

如何解决冲突

若要解决此规则的冲突，请从其他基类型或泛型集合派生该类型。

何时禁止显示警告

对于有关 ApplicationException 的冲突，请勿禁止显示此规则的警告。对于有关 XmlDocument 的冲突，可以安全地禁止显示此规则的警告。如果之前发布过代码，则可以安全地禁止显示有关非泛型集合的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

CA1060：将 P/Invoke 移动到 NativeMethods 类

2021/11/16 •

■ ID	CA1060
■	设计
■■■■■	重大

原因

方法使用平台调用服务访问非托管代码，不是 NativeMethods 类之一的成员。

规则说明

平台调用方法（如使用 System.Runtime.InteropServices.DllImportAttribute 属性标记的方法）或在 Visual Basic 中使用 `Declare` 关键字定义的方法可访问非托管代码。这些方法应是处于以下一个类中：

- NativeMethods - 此类不会对非托管代码权限取消堆栈审核。（System.Security.SuppressUnmanagedCodeSecurityAttribute 不得应用于此类。）此类用于可在任何位置使用的方法，因为会执行堆栈审核。
- SafeNativeMethods - 此类会对非托管代码权限取消堆栈审核。（System.Security.SuppressUnmanagedCodeSecurityAttribute 应用于此类。）此类用于可供任何人安全调用的方法。这些方法的调用方不需要执行完整安全评审以确保使用是安全的，因为这些方法对于任何调用方都无害。
- UnsafeNativeMethods - 此类会对非托管代码权限取消堆栈审核。（System.Security.SuppressUnmanagedCodeSecurityAttribute 应用于此类。）此类用于有潜在危险的方法。这些方法的任何调用方都必须执行完整安全检查，以确保使用是安全的，因为不会执行任何堆栈审核。

这些类声明为 `internal`（在 Visual Basic 中为 `Friend`），并声明一个私有构造函数来阻止创建新实例。这些类中的方法应是 `static` 和 `internal`（在在 Visual Basic 中是 `Shared` 和 `Friend`）。

如何解决冲突

若要解决与此规则的冲突，请将方法移动到合适的 NativeMethods 类中。对于大多数应用程序，将 P/Invoke 移动到名为 NativeMethods 的新类便足够了。

但是，如果要开发在其他应用程序中使用的库，应考虑定义两个名为 SafeNativeMethods 和 UnsafeNativeMethods 的其他类。这些类与 NativeMethods 类相似；但是，它们使用名为 SuppressUnmanagedCodeSecurityAttribute 的特殊属性进行标记 。应用此属性时，运行时不会执行完整堆栈审核来确保所有调用方都具有 UnmanagedCode 权限。运行时通常会在启动时检查是否具有此权限。因此可极大地提高对这些非托管方法的调用的性能，还使具备有限权限的代码可以调用这些方法。

不过，应非常小心地使用此属性。如果未正确实现，则可能会产生严重的安全隐患。

有关如何实现这些方法的信息，请参阅 NativeMethods 示例、SafeNativeMethods 示例和 UnsafeNativeMethods 示例 。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例声明了违反此规则的方法。若要更正该冲突，应将 RemoveDirectory P/Invoke 移动到设计为仅保存 P/invoke 的适当类。

```
' Violates rule: MovePInvokesToNativeMethodsClass.
Friend Class UnmanagedApi
    Friend Declare Function RemoveDirectory Lib "kernel32" (
   ByVal Name As String) As Boolean
End Class
```

```
// Violates rule: MovePInvokesToNativeMethodsClass.
internal class UnmanagedApi
{
    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
    internal static extern bool RemoveDirectory(string name);
}
```

NativeMethods 示例

由于 NativeMethods 类不应使用 SuppressUnmanagedCodeSecurityAttribute 进行标记，因此，置于其中的 P/invoke 需要 UnmanagedCode 权限 。由于大多数应用程序从本地计算机运行并随完全信任一起运行，因此这通常不会成为问题。但是，如果要开发可重用的库，则应考虑定义 SafeNativeMethods 或 UnsafeNativeMethods 类 。

下面的示例演示了一个 Interaction.Beep 方法，它可包装来自 user32.dll 的 MessageBeep 函数 。MessageBeep P/Invoke 会置于 NativeMethods 类中 。

```
Public NotInheritable Class Interaction

    Private Sub New()
    End Sub

    ' Callers require Unmanaged permission
    Public Shared Sub Beep()
        ' No need to demand a permission as callers of Interaction.Beep
        ' will require UnmanagedCode permission
        If Not NativeMethods.MessageBeep(-1) Then
            Throw New Win32Exception()
        End If

    End Sub

End Class

Friend NotInheritable Class NativeMethods

    Private Sub New()
    End Sub

    <DllImport("user32.dll", CharSet:=CharSet.Auto)>
    Friend Shared Function MessageBeep(ByVal uType As Integer) As <MarshalAs(UnmanagedType.Bool)> Boolean
    End Function

End Class
```

```
public static class Interaction
{
    // Callers require Unmanaged permission
    public static void Beep()
    {
        // No need to demand a permission as callers of Interaction.Beep
        // will require UnmanagedCode permission
        if (!NativeMethods.MessageBeep(-1))
            throw new Win32Exception();
    }
}

internal static class NativeMethods
{
    [DllImport("user32.dll", CharSet = CharSet.Auto)]
    [return: MarshalAs(UnmanagedType.Bool)]
    internal static extern bool MessageBeep(int uType);
}
```

SafeNativeMethods 示例

可以安全地向任何应用程序公开并且没有任何副作用的 P/Invoke 方法应置于名为 SafeNativeMethods 的类中。不必要求获得权限，也不必过多注意从其处调用权限。

下面的示例演示了一个 Environment.TickCount 属性，它可包装来自 kernel32.dll 的 GetTickCount 函数 。

```
Public NotInheritable Class Environment

    Private Sub New()
    End Sub

    ' Callers do not require Unmanaged permission
    Public Shared ReadOnly Property TickCount() As Integer
        Get
            ' No need to demand a permission in place of
            ' UnmanagedCode as GetTickCount is considered
            ' a safe method
            Return SafeNativeMethods.GetTickCount()
        End Get
    End Property

End Class

<SuppressUnmanagedCodeSecurityAttribute()>
Friend NotInheritable Class SafeNativeMethods

    Private Sub New()
    End Sub

    <DllImport("kernel32.dll", CharSet:=CharSet.Auto, ExactSpelling:=True)>
    Friend Shared Function GetTickCount() As Integer
    End Function

End Class
```

```
public static class Environment
{
    // Callers do not require UnmanagedCode permission
    public static int TickCount
    {
        get
        {
            // No need to demand a permission in place of
            // UnmanagedCode as GetTickCount is considered
            // a safe method
            return SafeNativeMethods.GetTickCount();
        }
    }
}

[SuppressUnmanagedCodeSecurityAttribute]
internal static class SafeNativeMethods
{
    [DllImport("kernel32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]
    internal static extern int GetTickCount();
}
```

UnsafeNativeMethods 示例

不能安全调用并且可能导致副作用的 P/Invoke 方法应置于名为 UnsafeNativeMethods 的类中。应严格检查这些方法，以确保不会无意中向用户公开它们。此外，这些方法在使用时，还应具有所需的其他权限，而不是 UnmanagedCode。

下面的示例演示了一个 Cursor.Hide 方法，它可包装来自 user32.dll 的 ShowCursor 函数 。

```
Public NotInheritable Class Cursor

    Private Sub New()
    End Sub

    ' Callers do not require Unmanaged permission, however,
    ' they do require UIPermission.AllWindows
    Public Shared Sub Hide()
        ' Need to demand an appropriate permission
        ' in  place of UnmanagedCode permission as
        ' ShowCursor is not considered a safe method
        Dim permission As New UIPermission(UIPermissionWindow.AllWindows)
        permission.Demand()
        UnsafeNativeMethods.ShowCursor(False)

    End Sub

End Class

<SuppressUnmanagedCodeSecurityAttribute()>
Friend NotInheritable Class UnsafeNativeMethods

    Private Sub New()
    End Sub

    <DllImport("user32.dll", CharSet:=CharSet.Auto, ExactSpelling:=True)>
    Friend Shared Function ShowCursor(<MarshalAs(UnmanagedType.Bool)> ByVal bShow As Boolean) As Integer
    End Function

End Class
```

```
public static class Cursor
{
    // Callers do not require UnmanagedCode permission, however,
    // they do require UIPermissionWindow.AllWindows.
    public static void Hide()
    {
        // Need to demand an appropriate permission
        // in place of UnmanagedCode permission as
        // ShowCursor is not considered a safe method.
        new UIPermission(UIPermissionWindow.AllWindows).Demand();
        UnsafeNativeMethods.ShowCursor(false);
    }
}

[SuppressUnmanagedCodeSecurityAttribute]
internal static class UnsafeNativeMethods
{
    [DllImport("user32.dll", CharSet = CharSet.Auto, ExactSpelling = true)]
    internal static extern int ShowCursor([MarshalAs(UnmanagedType.Bool)] bool bShow);
}
```

另请参阅

- 设计规则

CA1061:不要隐藏基类方法

2021/11/16 •

	[
■ ID	CA1061
■	设计
▬▬	重大

原因

派生类型声明的方法与其基方法之一具有相同的名称和相同数量的参数;一个或多个参数是基方法中相应参数的基类型;所有剩余参数的类型都与基方法中相应参数的类型相同。

规则说明

如果派生方法的参数签名只是在类型方面有所不同，而且与基方法的参数签名中的对应类型相比，这些类型的派生方式更弱，则基类型中的方法由派生类型中的同名方法隐藏。

如何解决冲突

若要解决此规则的冲突，请删除或重命名该方法，或者更改参数签名，使该方法不会隐藏基方法。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

以下示例显示与此规则冲突的方法。

```
class BaseType
{
    internal void MethodOne(string inputOne, object inputTwo)
    {
        Console.WriteLine("Base: {0}, {1}", inputOne, inputTwo);
    }

    internal void MethodTwo(string inputOne, string inputTwo)
    {
        Console.WriteLine("Base: {0}, {1}", inputOne, inputTwo);
    }
}

class DerivedType : BaseType
{
    internal void MethodOne(string inputOne, string inputTwo)
    {
        Console.WriteLine("Derived: {0}, {1}", inputOne, inputTwo);
    }

    // This method violates the rule.
    internal void MethodTwo(string inputOne, object inputTwo)
    {
        Console.WriteLine("Derived: {0}, {1}", inputOne, inputTwo);
    }
}

class Test
{
    static void Main1061()
    {
        DerivedType derived = new DerivedType();

        // Calls DerivedType.MethodOne.
        derived.MethodOne("string1", "string2");

        // Calls BaseType.MethodOne.
        derived.MethodOne("string1", (object)"string2");

        // Both of these call DerivedType.MethodTwo.
        derived.MethodTwo("string1", "string2");
        derived.MethodTwo("string1", (object)"string2");
    }
}
```

CA1062:验证公共方法的参数

2021/11/16 •

	[
■ ID	CA1062
■	设计
▬▬▬	非中断

原因

外部可见方法取消引用其中一个引用参数，而不验证该参数是否 `null`（Visual Basic 中 `Nothing`）。

可以将此规则配置为从分析中排除某些类型和参数。还可以指示 null 检查验证方法。

规则说明

对于传递给外部可见方法的所有引用参数，都应检查其是否为 `null`。如果需要，则在参数为 `null` 时引发 ArgumentNullException。

如果某个方法由于被声明为公共或受保护而可以从未知程序集进行调用，则应验证该方法的所有参数。如果该方法设计为仅由已知程序集调用，请将方法标记为 `internal` 并将 InternalsVisibleToAttribute 特性应用于包含该方法的程序集。

如何解决冲突

若要修复与此规则的冲突，请验证每个引用参数是否为 `null`。

何时禁止显示警告

如果确定取消引用的参数已由函数中的其他方法调用进行验证，则可以禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型
- 排除扩展方法“this”参数
- Null 检查验证方法

可以仅为此规则、为所有规则或为此类别（设计）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅符号名称(包括具有相应名称的所有符号,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀,例如表示方法的 `M:`、表示类型的 `T:`,以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数,`.cctor` 表示静态构造函数。

示例:

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如,若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行,请将以下键值对添加到项目中的 .editorconfig 文件:

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅类型名称(包括具有相应名称的所有类型,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式,前缀为 `T:`(可选)。

示例:

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

排除扩展方法“this”参数

默认情况下，此规则分析并标记扩展方法的 `this` 参数。可以通过将以下键值对添加到项目中的 editorconfig 文件，排除扩展方法的 `this` 参数的分析：

```
dotnet_code_quality.CA1062.exclude_extension_method_this_parameter = true
```

Null 检查验证方法

如果代码在引用的库或项目中调用了特殊的 null 检查验证方法，则此规则可能导致误报。可以通过指定 null 检查验证方法的名称或签名来避免这种误报。此分析假定在调用后传递给这些方法的参数为非 null。例如，若要将名为 `Validate` 的所有方法标记为 null 检查验证方法，请将以下键值对添加到项目中的 editorconfig 文件：

```
dotnet_code_quality.CA1062.null_check_validation_methods = Validate
```

选项值中允许的方法名称格式（用 `|` 分隔）：

- 仅方法名称（包括具有相应名称的所有方法，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `M:`（可选）。

示例：

[[[	[[
`dotnet_code_quality.CA1062.null_check_validation_methods = Validate`	匹配编译中所有名为 `Validate` 的方法
`dotnet_code_quality.CA1062.null_check_validation_methods = Validate1|Validate2`	匹配编译中所有名为 `Validate1` 或 `Validate2` 的方法
`dotnet_code_quality.CA1062.null_check_validation_methods = NS.MyType.Validate(ParamType)`	将特定方法 `Validate` 与给定的完全限定签名相匹配
`dotnet_code_quality.CA1062.null_check_validation_methods = NS1.MyType1.Validate1(ParamType)|NS2.MyType2.Validate2(ParamType)`	将特定方法 `Validate1` 和 `Validate2` 与相应的完全限定签名相匹配

示例 1

下面的示例演示了违反规则的方法和符合规则的方法。

```
using System;

namespace DesignLibrary
{
    public class Test
    {
        // This method violates the rule.
        public void DoNotValidate(string input)
        {
            if (input.Length != 0)
            {
                Console.WriteLine(input);
            }
        }

        // This method satisfies the rule.
        public void Validate(string input)
        {
            if (input == null)
            {
                throw new ArgumentNullException(nameof(input));
            }
            if (input.Length != 0)
            {
                Console.WriteLine(input);
            }
        }
    }
}
```

```
Imports System

Namespace DesignLibrary

    Public Class Test

        ' This method violates the rule.
        Sub DoNotValidate(ByVal input As String)

            If input.Length <> 0 Then
                Console.WriteLine(input)
            End If

        End Sub

        ' This method satisfies the rule.
        Sub Validate(ByVal input As String)

            If input Is Nothing Then
                Throw New ArgumentNullException(NameOf(input))
            End If

            If input.Length <> 0 Then
                Console.WriteLine(input)
            End If

        End Sub

    End Class

End Namespace
```

示例 2

填充为引用对象的字段或属性的复制构造函数也可能与 CA1062 规则发生冲突。发生冲突的原因是，传递到复制构造函数的所复制对象可能为 `null`（在 Visual Basic 中为 `Nothing`）。若要解决冲突，请使用 `static`（在 Visual Basic 中为 `Shared`）方法来检查复制的对象是否不为 null。

在下面的 `Person` 类示例中，传递给 `Person` 复制构造函数的 `other` 对象可能为 `null`。

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Copy constructor CA1062 fires because other is dereferenced
    // without being checked for null
    public Person(Person other)
        : this(other.Name, other.Age)
    {
    }
}
```

示例 3

在下面经修订的 `Person` 示例中，系统首先会在 `PassThroughNonNull` 方法中检查传递给复制构造函数的 `other` 对象是否为 null。

```
public class Person
{
    public string Name { get; private set; }
    public int Age { get; private set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // Copy constructor
    public Person(Person other)
        : this(PassThroughNonNull(other).Name, other.Age)
    {
    }

    // Null check method
    private static Person PassThroughNonNull(Person person)
    {
        if (person == null)
            throw new ArgumentNullException(nameof(person));
        return person;
    }
}
```

CA1063:正确实现 IDisposable

2021/11/16 •

	[
■ ID	CA1063
■	设计
▬▬▬	非中断

原因

System.IDisposable 接口无法正确实现。可能的原因包括：

- 在类中重新实现 IDisposable。
- 再次重写 `Finalize`。
- 重写 `Dispose()`。
- `Dispose()` 方法是非公用、已密封或命名为“Dispose”。
- `Dispose(bool)` 未受保护、虚拟或未密封。
- 在未密封类型中，`Dispose()` 必须调用 `Dispose(true)`。
- 对于未密封的类型，`Finalize` 实现不调用或不同时调用 `Dispose(bool)` 或基类终结器。

违反其中任何一个模式都会触发警告 CA1063。

声明和实现 IDisposable 接口的每个未密封类型都必须提供自己的 `protected virtual void Dispose(bool)` 方法。`Dispose()` 应该调用 `Dispose(true)`，而终结器应该调用 `Dispose(false)`。如果创建声明和实现 IDisposable 接口的未密封类型，则必须对 `Dispose(bool)` 进行定义和调用。有关详细信息，请参阅清理非托管资源(.NET 指南)以及 Dispose 模式。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

所有的 IDisposable 类型都应正确实现 Dispose 模式。

如何解决冲突

检查代码，并确定以下哪种解决方法能解决此冲突：

- 从类型实现的接口列表中移除 IDisposable，并重写 Dispose 基类实现。
- 从类型中移除终结器，重写 Dispose(bool disposing)，并在“disposing”为 false 的代码路径中加入终结逻辑。
- 重写 Dispose(bool disposing)，并在“disposing”为 true 的代码路径中加入释放逻辑。
- 确保将 Dispose() 声明为公用且已密封。

- 将 dispose 方法重命名为“Dispose”，并确保将其声明为公用且已密封。
- 确保 Dispose(bool) 声明为受保护、虚拟和未密封。
- 修改 Dispose()，使其调用 Dispose(true)，并在当前对象实例（在 Visual Basic 中为 `this` 或 `Me`）上调用 SuppressFinalize，然后返回。
- 修改终结器，使其调用 Dispose(false)，然后返回。
- 如果创建声明和实现 IDisposable 接口的未密封类型，请确保 IDisposable 的实现遵循本节前面所介绍的模式。

何时禁止显示警告

不禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

伪代码示例

以下伪代码提供了有关如何在使用托管资源和本机资源的类中实现 Dispose(bool) 的常规示例。

```
public class Resource : IDisposable
{
    private bool isDisposed;
    private IntPtr nativeResource = Marshal.AllocHGlobal(100);
    private AnotherResource managedResource = new AnotherResource();

    // Dispose() calls Dispose(true)
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // The bulk of the clean-up code is implemented in Dispose(bool)
    protected virtual void Dispose(bool disposing)
    {
        if (isDisposed) return;

        if (disposing)
        {
            // free managed resources
            managedResource.Dispose();
        }

        // free native resources if there are any.
        if (nativeResource != IntPtr.Zero)
        {
            Marshal.FreeHGlobal(nativeResource);
            nativeResource = IntPtr.Zero;
        }

        isDisposed = true;
    }

    // NOTE: Leave out the finalizer altogether if this class doesn't
    // own unmanaged resources, but leave the other methods
    // exactly as they are.
    ~Resource()
    {
        // Finalizer calls Dispose(false)
        Dispose(false);
    }
}
```

另请参阅

- Dispose 模式（框架设计准则）
- 清理非托管资源（.NET 指南）

CA1064:异常应该是公共的

2021/11/16 •

	[
■ ID	CA1064
■	设计
▬▬▬	非中断

原因

非公共异常直接派生自 Exception、SystemException 或 ApplicationException。

规则说明

内部异常仅在其自己的内部范围内可见。当异常超出内部范围后，只能使用基异常来捕获该异常。如果内部异常继承自 Exception、SystemException 或 ApplicationException，则外部代码将没有足够的信息来了解如何处理该异常。

但是，如果代码有一个公共异常，稍后会用作内部异常的基异常，则有理由认为后续代码将能够对该基异常进行智能化操作。该公共异常将会比 Exception、SystemException 或 ApplicationException 提供更多的信息。

如何解决冲突

使异常成为公共异常，或从不是 Exception、SystemException 或 ApplicationException 的公共异常派生内部异常。

何时禁止显示警告

如果确定在所有情况下私有异常都将在其自己的内部范围内被捕获，则禁止显示此规则的消息。

示例

此规则在第一个示例方法 FirstCustomException 上触发，因为 exception 类直接派生自 Exception，并且是内部类。此规则不会在 SecondCustomException 类上触发，尽管该类也直接派生自 Exception，但该类声明为公共类。第三个类也不会触发该规则，因为它并非直接派生自 System.Exception、System.SystemException 或 System.ApplicationException。

```
// Violates this rule
[Serializable]
internal class FirstCustomException : Exception
{
    internal FirstCustomException()
    {
    }

    internal FirstCustomException(string message)
        : base(message)
    {
    }
```

```
    internal FirstCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    protected FirstCustomException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}

// Does not violate this rule because
// SecondCustomException is public
[Serializable]
public class SecondCustomException : Exception
{
    public SecondCustomException()
    {
    }

    public SecondCustomException(string message)
        : base(message)
    {

    }

    public SecondCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }

    protected SecondCustomException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}

// Does not violate this rule because
// ThirdCustomException it does not derive directly from
// Exception, SystemException, or ApplicationException
[Serializable]
internal class ThirdCustomException : SecondCustomException
{
    internal ThirdCustomException()
    {
    }

    internal ThirdCustomException(string message)
        : base(message)
    {
    }

    internal ThirdCustomException(string message, Exception innerException)
        : base(message, innerException)
    {
    }


    protected ThirdCustomException(SerializationInfo info, StreamingContext context)
        : base(info, context)
    {
    }
}
```

CA1065:不要在意外的位置引发异常

2021/11/16 •

	[
■ ID	CA1065
■	设计
▬▬▬	非中断

原因

不应引发异常的方法引发了异常。

规则说明

不应引发异常的方法可分成以下几类:

- 属性 Get 方法
- 事件访问器方法
- Equals 方法
- GetHashCode 方法
- ToString 方法
- 静态构造函数
- 终结器
- Dispose 方法
- 相等运算符
- 隐式强制转换运算符

以下各节讨论了这些方法类型。

属性 Get 方法

属性基本上都是智能字段。因此，其行为应尽可能类似于字段。字段不会引发异常，属性也不应引发异常。如果有一个引发异常的属性，可考虑将其设为方法。

属性 Get 方法可引发以下异常:

- System.InvalidOperationException 和所有派生项(包括 System.ObjectDisposedException)
- System.NotSupportedException 和所有派生项
- System.ArgumentException(仅从带有索引的 Get)
- KeyNotFoundException(仅从带有索引的 Get)

事件访问器方法

事件访问器应是不会引发异常的简单操作。尝试添加或删除事件处理程序时，事件不应引发异常。

事件访问器可引发以下异常：

- System.InvalidOperationException 和所有派生项（包括 System.ObjectDisposedException）
- System.NotSupportedException 和所有派生项
- ArgumentException 和派生项

Equals 方法

以下 Equals 方法不应引发异常：

- System.Object.Equals
- Equals

Equals 方法应返回 `true` 或 `false` 而不是引发异常。例如，如果 Equals 传递两个不匹配的类型，则应只返回 `false` 而不是引发 ArgumentException。

GetHashCode 方法

以下 GetHashCode 方法通常不应引发异常：

- GetHashCode
- GetHashCode

GetHashCode 应始终返回值。否则，可能会丢失哈希表中的项。

采用参数的 GetHashCode 版本可能会引发 ArgumentException。但是，Object.GetHashCode 应始终不会引发异常。

ToString 方法

调试器使用 System.Object.ToString 来帮助以字符串格式显示有关对象的信息。因此，ToString 不应更改对象的状态，也不应引发异常。

静态构造函数

从静态构造函数引发异常将导致该类型在当前应用程序域中不可用。从静态构造函数引发异常应具备充分的理由（如安全问题）。

终结器

从终结器引发异常将导致 CLR 快速失败，从而中断过程。因此，应始终避免在终结器中引发异常。

Dispose 方法

System.IDisposable.Dispose 方法不应引发异常。Dispose 通常作为 `finally` 子句中清理逻辑的一部分调用。因此，从 Dispose 显式引发异常将强制用户在 `finally` 子句内添加异常处理。

Dispose (false) 代码路径应始终不会引发异常，因为 Dispose 几乎都是从终结器调用的。

相等运算符 (==, !=)

与 Equals 方法一样，相等运算符应返回 `true` 或 `false`，而不应引发异常。

隐式强制转换运算符

由于用户通常不知道已调用了隐式强制转换运算符，因此对它引发的异常会感到意外。因此，隐式强制转换运算符不应引发异常。

如何解决冲突

对于属性 Getter，可更改逻辑，使其不再需要引发异常，或将属性更改为方法。

对于前面列出的所有其他方法类型，可更改逻辑，使其不再必须引发异常。

何时禁止显示警告

如果冲突是由异常声明而不是引发的异常造成的，则可禁止显示此规则发出的警告。

相关规则

- CA2219:在异常子句中不引发异常

另请参阅

- 设计规则

CA1066：重写 Equals 时实现 IEquatable

2021/11/16 •

■ ID	CA1066
■	设计
■■■■■	非中断

原因

值类型(结构)重写 Equals 方法，但不实现 IEquatable<T>。

规则说明

值类型重写 Equals 方法指示它可支持对类型的两个实例进行比较以确定二者的值是否相等。请考虑实现 IEquatable<T> 接口以支持强类型相等性测试。这可确保执行相等性检查的调用方调用强类型 System.IEquatable<T>.Equals 方法，避免对参数进行装箱，从而提高性能。有关详细信息，请参阅此文。

System.IEquatable<T>.Equals 实现应返回与 Equals 一致的结果。

如何解决冲突

若要解决冲突，请实现 IEquatable<T> 并更新 Equals 重写，以调用此实现的方法。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
public struct S
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public override int GetHashCode()
        => _value.GetHashCode();

    public override bool Equals(object other)
        => other is S otherS && _value == otherS._value;
}
```

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public override int GetHashCode()
        => _value.GetHashCode();

    public override bool Equals(object other)
        => other is S otherS && Equals(otherS);

    public bool Equals(S other)
        => _value == other._value;
}
```

何时禁止显示警告

如果实现接口的设计和性能优势并不重要，则可忽略此规则的冲突警告。

相关规则

- CA1067：实现 IEquatable 时重写 Equals

另请参阅

- 设计规则

CA1067：实现 IEquatable 时重写 Equals

2021/11/16 •

	[
■ ID	CA1067
■	设计
▬▬▬	非中断

原因

类型实现 IEquatable<T>，但不实现 Equals。

规则说明

实现 IEquatable<T> 接口的类型指示它可支持对类型的两个实例进行比较以确定二者是否相等。还应重写 Equals 和 GetHashCode() 方法的基类实现，以便其行为与 System.IEquatable<T>.Equals 实现的行为一致。请参阅此处了解详细信息。

Equals 实现应返回与 System.IEquatable<T>.Equals 实现一致的结果。

如何解决冲突

若要解决冲突，请调用 Equals 实现来重写 System.IEquatable<T>.Equals 并实现它。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public bool Equals(S other)
        => _value == other._value;
}
```

```
using System;

public struct S : IEquatable<S>
{
    private readonly int _value;
    public S(int f)
    {
        _value = f;
    }

    public bool Equals(S other)
        => _value == other._value;

    public override bool Equals(object obj)
        => obj is S objS && Equals(objS);

    public override int GetHashCode()
        => _value.GetHashCode();
}
```

何时禁止显示警告

请勿禁止显示此规则的冲突警告。

相关规则

- CA1066：重写 Equals 时实现 IEquatable

另请参阅

- 设计规则

CA1068:CancellationToken 参数必须最后出现

2021/11/16 •

■ ID	CA1068
■	设计
▬▬▬	重大

原因

此方法具有 CancellationToken 参数，该参数不是最后一个参数。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

执行长时间运行操作或异步操作并可取消的方法，通常采用取消令牌参数。每个取消令牌都有一个 CancellationTokenSource，以创建令牌并将其用于可取消的计算。通常的做法是使用一长的方法调用链，将取消令牌从调用方传递到被调用方。因此，参与可取消计算的大量方法最终都具有取消令牌参数。但是，取消令牌本身通常与大多数这些方法的核心功能无关。将此类参数作为列表中的最后一个参数是一种很好的 API 设计实践。

特殊情况

在以下特殊情况下，不会触发规则 CA1068：

- 方法具有一个或多个可选参数(在 Visual Basic 中可选)，这些参数位于非可选取消令牌参数之后。编译器要求，在定义完所有非可选参数之后定义所有可选参数。
- 方法具有一个或多个 ref 或 out 参数(在 Visual Basic 中为 ByRef)，这些参数位于取消令牌参数之后。通常将 `ref` 或 `out` 参数放在列表的末尾，因为它们通常指示方法的输出值。

如何解决冲突

更改方法签名，以将取消令牌参数移到列表末尾。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
// Violates CA1068
public void LongRunningOperation(CancellationToken token, string usefulParameter)
{
    ...
}
```

```
// Does not violate CA1068
public void LongRunningOperation(string usefulParameter, CancellationToken token)
{
    ...
}
```

何时禁止显示警告

如果该方法是一个外部可见的公共 API，该 API 已是已发货库的一部分，则可以安全地禁止显示此规则的警告，以避免库使用者的中断性变更。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(设计)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中

的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

相关规则

- CA1021:避免使用 out 参数

另请参阅

- 适用于 CancellationToken 的推荐模式

CA1069:枚举不得具有重复值

2021/11/16 •

	▮
■ ID	CA1069
■	设计
▬▬	重大

原因

枚举具有多个成员，这些成员显式分配有相同常数值。

规则说明

每个枚举成员都应具有唯一的常数值，或者为其显式分配枚举中的前一个成员以指示共享值的明确意图。例如：

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 2,   // CA1069: This is not fine. Either assign a different constant value or 'Field2' to
indicate explicit intent of sharing value.
}
```

此规则有助于捕获在以下场景中引入的功能性 bug：

- 意外键入错误：用户意外地为多个成员键入了相同的常数值。
- 复制粘贴错误：用户复制了一个现有成员定义，然后重命名了该成员，但忘记更改值。
- 合并多个分支中的解决方案：在不同分支中添加了具有不同名称但有相同值的新成员。

如何解决冲突

若要解决冲突，请分配新的唯一常数值，或分配枚举中的前一个成员以指示共享同一值的明确意图。例如，以下代码片段显示了与此规则的冲突，以及解决冲突的几种方法：


```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 2,   // CA1069: This is not fine. Either assign a different constant value or 'Field2' to
indicate explicit intent of sharing value.
}
```

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = 3,   // This is now fine
}
```

```
enum E
{
   Field1 = 1,
   AnotherNameForField1 = Field1,   // This is fine
   Field2 = 2,
   Field3 = Field2,   // This is also fine
}
```

何时禁止显示警告

请勿禁止显示此规则的冲突。

另请参阅

- 设计规则

CA1070:不要将事件字段声明为“虚拟”

2021/11/16 •

	[
■ ID	CA1070
■	设计
▬▬▬	重大

原因

将类似字段的事件声明为了虚拟事件。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

遵循这些 .NET 设计指南，在派生类中引发基类事件。不要在基类中声明虚拟事件。派生类中的重写事件具有未定义的行为。C# 编译器不会正确处理此事件，并且无法预知派生事件的订阅者是否实际上会订阅基类事件。

```
using System;
public class C
{
    // CA1070: Event 'ThresholdReached' should not be declared virtual.
    public virtual event EventHandler ThresholdReached;
}
```

如何解决冲突

遵循这些 .NET 设计指南，并避免出现类似字段的虚拟事件。

何时禁止显示警告

如果该方法是一个外部可见的公共 API，该 API 已是已发布库的一部分，则可以安全地禁止显示此规则的警告，以避免库使用者遇到中断性变更。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

另请参阅

- 设计规则

文档规则

2021/11/16 •

文档规则支持通过正确为外部可见的 API 使用 XML 文档注释来编写记录详尽的库。

在本节中

"	"
CA1200:不要使用带前缀的 cref 标记	XML 文档标记中的 cref 属性是指“代码引用”。它指定标记的内部文本是一个代码元素，例如类型、方法或属性。避免使用带有前缀的 `cref` 标记，因为它会阻止编译器验证引用。它还会阻止 Visual Studio 集成开发环境 (IDE) 在重构过程中查找和更新这些符号引用。

CA1200:不要使用带前缀的 cref 标记

2021/11/16 •

	[
■ ID	CA1200
■	文档
▬▬▬	非中断

原因

XML 文档注释中的 cref 标记使用了前缀。

规则说明

XML 文档标记中的 cref 属性是指“代码引用”。它指定标记的内部文本是一个代码元素，例如类型、方法或属性。避免使用带有前缀的 `cref` 标记，因为它会阻止编译器验证引用。它还会阻止 Visual Studio 集成开发环境 (IDE) 在重构过程中查找和更新这些符号引用。建议使用不带前缀的完整语法以引用 cref 标记中的符号名称。

如何解决冲突

若要解决此规则的冲突，请从 `cref` 标记中删除前缀。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
// Violates CA1200
/// <summary>
/// Type <see cref="T:C" /> contains method <see cref="F:C.F" />
/// </summary>
class C
{
    public void F() { }
}
```

```
// Does not violate CA1200
/// <summary>
/// Type <see cref="C" /> contains method <see cref="F" />
/// </summary>
class C
{
    public void F() { }
}
```

何时禁止显示警告

如果由于编译器无法找到引用类型，代码引用必须使用前缀，则可以安全地禁止显示此警告。例如，如果代码引用在完整框架中引用特殊属性，但文件根据可移植框架进行编译，则可以禁止显示此警告。

另请参阅

- 使用 XML 注释来记录代码

全球化规则

2021/11/16 •

全球化规则支持世界通用库和应用程序。

在本节中

"	"
CA1303:请不要将文本作为本地化参数传递	某外部可见的方法将一个字符串字面量作为参数传递给 .NET 构造函数或方法，该字符串应该是可本地化的字符串。
CA1304:指定 CultureInfo	某方法或构造函数调用的成员有一个接受 System.Globalization.CultureInfo 参数的重载，但该方法或构造函数没有调用接受 CultureInfo 参数的重载。如果未提供 CultureInfo 或 System.IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生您想要的效果。
CA1305:指定 IFormatProvider	某方法或构造函数调用的一个或多个成员有接受 System.IFormatProvider 参数的重载，但该方法或构造函数没有调用接受 IFormatProvider 参数的重载。如果未提供 System.Globalization.CultureInfo 或 IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生您想要的效果。
CA1307:为了清晰起见，请指定 StringComparison	字符串比较运算使用不设置 StringComparison 参数的方法重载。
CA1308:将字符串规范化为大写	字符串应正常化为大写字母。少量字符转换为小写字母后不能再转换回来。
CA1309:使用按顺序的 StringComparison	非语义的字符串比较运算不会将 StringComparison 参数设置为 Ordinal 或 OrdinalIgnoreCase。因此，通过将参数显式设置为 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase，通常可以提高代码的速度、正确性和可靠性。
CA1310：为了确保正确，请指定 StringComparison	字符串比较操作使用未设置 StringComparison 参数的方法重载，并默认使用区域性特定的字符串比较。
CA2101：指定对 P/Invoke 字符串参数进行封送处理	某平台调用成员允许部分信任的调用方，具有一个字符串参数，并且不显式封送该字符串。这可能导致潜在的安全漏洞。

CA1303:请不要将文本作为本地化参数传递

2021/11/16 •

■ ID	CA1303
■	全球化
■■■■■	非中断

原因

某方法将一个字符串字面量作为参数传递给 .NET 构造函数或方法，该字符串应该是可本地化的字符串。

将文本字符串作为值传递给参数或属性，并且存在以下一种或多种情况时，就会引发此警告：

- 参数的 LocalizableAttribute 特性或属性设置为 `true`。
- 文本字符串将传递给 Console.Write 或 Console.WriteLine 方法重载的 `string value` 或 `string format` 参数。
- 规则 CA1303 配置为使用命名启发式，并且参数或属性名称包含短语 `Text`、`Message` 或 `Caption`。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

嵌入在源代码中的字符串字面量难以本地化。

如何解决冲突

若要解决此规则的冲突，请将字符串字面量替换为通过 ResourceManager 类的实例检索到的字符串。

对于不需要本地化字符串的方法，可通过以下方式消除不必要的 CA1303 警告：

- 如果启用了命名启发式选项，请重命名参数或属性。
- 删除参数上的 LocalizableAttribute 特性或属性，或将其设置为 `false`（`[Localizable(false)]`）。

何时禁止显示警告

如果以下任一表述适用，可禁止显示此规则的警告：

- 不会对代码库进行本地化。
- 该字符串不会向最终用户或使用代码库的开发人员公开。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

- 使用命名启发式

你可以仅为此规则、为所有规则或为此类别(全球化)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

使用命名启发式

可配置包含 `Text`、`Message` 或 `Caption` 的参数或属性名称是否会触发此规则。

```
dotnet_code_quality.CA1303.use_naming_heuristic = true
```

示例

下面的示例演示了在其两个参数之一超出范围时写入控制台的方法。对于 `hour` 参数检查，文本字符串将传递到 `Console.WriteLine`，这与此规则冲突。对于 `minute` 参数检查，通过 ResourceManager 检索的字符串将传递到 `Console.WriteLine`，这符合规则。

```
<Assembly: System.Resources.NeutralResourcesLanguageAttribute("en-US")>
Namespace GlobalizationLibrary

    Public Class DoNotPassLiterals

        Dim stringManager As System.Resources.ResourceManager

        Sub New()
            stringManager = New System.Resources.ResourceManager(
                "en-US", System.Reflection.Assembly.GetExecutingAssembly())
        End Sub

        Sub TimeMethod(hour As Integer, minute As Integer)

            If (hour < 0 Or hour > 23) Then
                'CA1303 fires because a literal string
                'is passed as the 'value' parameter.
                Console.WriteLine("The valid range is 0 - 23.")
            End If

            If (minute < 0 Or minute > 59) Then
                Console.WriteLine(
                    stringManager.GetString("minuteOutOfRangeMessage",
                        System.Globalization.CultureInfo.CurrentUICulture))
            End If

        End Sub

    End Class

End Namespace
```

```
public class DoNotPassLiterals
{
    ResourceManager stringManager;
    public DoNotPassLiterals()
    {
        stringManager = new ResourceManager("en-US", Assembly.GetExecutingAssembly());
    }

    public void TimeMethod(int hour, int minute)
    {
        if (hour < 0 || hour > 23)
        {
            // CA1303 fires because a literal string
            // is passed as the 'value' parameter.
            Console.WriteLine("The valid range is 0 - 23.");
        }

        if (minute < 0 || minute > 59)
        {
            Console.WriteLine(stringManager.GetString(
            "minuteOutOfRangeMessage", CultureInfo.CurrentUICulture));
        }
    }
}
```

另请参阅

- .NET 应用中的资源
- 行为更改的社区请求

CA1304:指定 CultureInfo

2021/11/16 •

■ ID	CA1304
■	全球化
▬▬▬	非中断

原因

某方法或构造函数调用的成员有接受 System.Globalization.CultureInfo 参数的重载，但该方法或构造函数没有调用接受 CultureInfo 参数的重载。此规则将忽略对以下方法的调用：

- Activator.CreateInstance
- ResourceManager.GetObject
- ResourceManager.GetString

还可以配置此规则将排除的多个符号。

规则说明

如果未提供 CultureInfo 或 System.IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生你想要的效果。此外，.NET 成员根据可能不适合你的代码的假设选择默认区域性和格式。为了确保代码满足相应的情况，你应该根据以下准则提供区域性特定的信息：

- 如果将向用户显示值，请使用当前区域性。请参阅 CultureInfo.CurrentCulture。
- 如果通过软件存储和访问值(即保留到文件或数据库)，请使用固定区域性。请参阅 CultureInfo.InvariantCulture。
- 如果不知道该值的目的地，请让数据使用者或提供商指定区域性。

即使重载成员的默认行为适合你的需求，但最好还是显式调用区域性特定的重载，以便你的代码自我记录，且更容易维护。

> **NOTE**
> CultureInfo.CurrentUICulture 仅用于通过使用 System.Resources.ResourceManager 类的实例检索本地化的资源。

如何解决冲突

若要解决此规则的冲突，请使用接受 CultureInfo 参数的重载。

何时禁止显示警告

当确定默认区域性是正确的选择，并且代码可维护性不是重要的开发优先级时，可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(全球化)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

演示如何解决冲突的示例

在下面的示例中，`BadMethod` 将导致此规则出现两个冲突。`GoodMethod` 通过将固定区域性传递给 String.Compare 来更正第一个冲突，通过将当前区域性传递给 String.ToLower 来更正第二个冲突，因为向用户显示了 `string3`。

```
public class CultureInfoTest
{
    public void BadMethod(String string1, String string2, String string3)
    {
        if (string.Compare(string1, string2, false) == 0)
        {
            Console.WriteLine(string3.ToLower());
        }
    }

    public void GoodMethod(String string1, String string2, String string3)
    {
        if (string.Compare(string1, string2, false,
                           CultureInfo.InvariantCulture) == 0)
        {
            Console.WriteLine(string3.ToLower(CultureInfo.CurrentCulture));
        }
    }
}
```

演示格式化输出的示例

下面的示例演示了当前区域性对 DateTime 类型选择的默认 IFormatProvider 的影响。

```
public class IFormatProviderTest
{
    public static void Main1304()
    {
        string dt = "6/4/1900 12:15:12";

        // The default behavior of DateTime.Parse is to use
        // the current culture.

        // Violates rule: SpecifyIFormatProvider.
        DateTime myDateTime = DateTime.Parse(dt);
        Console.WriteLine(myDateTime);

        // Change the current culture to the French culture,
        // and parsing the same string yields a different value.

        Thread.CurrentThread.CurrentCulture = new CultureInfo("Fr-fr", true);
        myDateTime = DateTime.Parse(dt);

        Console.WriteLine(myDateTime);
    }
}
```

该**示例**产**生下面的**输**出**：

```
6/4/1900 12:15:12 PM
06/04/1900 12:15:12
```

相关规则

- CA1305:指定 IFormatProvider

另请参阅

- 使用 `CultureInfo` 类

CA1305:指定 IFormatProvider

2021/11/16 •

■ ID	CA1305
■	全球化
▬▬	非中断

原因

某方法或构造函数调用的一个或多个成员有接受 System.IFormatProvider 参数的重载，但该方法或构造函数没有调用接受 IFormatProvider 参数的重载。

此规则会忽略对记录为忽略 IFormatProvider 参数的 .NET 方法的调用。此规则还会忽略以下方法：

- Activator.CreateInstance
- ResourceManager.GetObject
- ResourceManager.GetString
- Boolean.ToString
- Char.ToString
- Guid.ToString

规则说明

如果未提供 System.Globalization.CultureInfo 或 IFormatProvider 对象，则重载成员提供的默认值可能不会在所有区域设置中产生你想要的效果。此外，.NET 成员根据假设（可能不适合你的代码）选择默认区域性和格式。为了确保代码满足相应的情况，你应该根据以下准则提供区域性特定的信息：

- 如果将向用户显示值，则使用当前区域性。请参阅 CultureInfo.CurrentCulture。
- 若果通过软件存储和访问该值（保留到文件或数据库），则使用固定区域性。请参阅 CultureInfo.InvariantCulture。
- 如果不知道该值的目的地，请让数据使用者或提供商指定区域性。

即使重载成员的默认行为适合你的需求，但最好还是显式调用区域性特定的重载，以便你的代码自我记录，且更容易维护。

如何解决冲突

若要解决此规则的冲突，请使用接受 IFormatProvider 参数的重载。或者，使用 C# 内插字符串，并将其传递给 FormattableString.Invariant 方法。

何时禁止显示警告

当确定默认格式是正确的选择，并且代码可维护性不是重要的开发优先级时，可禁止显示此规则的警告。

示例

在以下代码中，`example1` 字符串与规则 CA1305 冲突。`example2` 字符串通过将 CultureInfo.CurrentCulture（实现 IFormatProvider）传递到 String.Format(IFormatProvider, String, Object) 来满足规则 CA1305。`example3` 字符串通过将内插字符串传递到 Invariant 来满足规则 CA1305。

```
string name = "Georgette";

// Violates CA1305
string example1 = String.Format("Hello {0}", name);

// Satisfies CA1305
string example2 = String.Format(CultureInfo.CurrentCulture, "Hello {0}", name);

// Satisfies CA1305
string example3 = FormattableString.Invariant($"Hello {name}");
```

相关规则

- CA1304:指定 CultureInfo

另请参阅

- 使用 `CultureInfo` 类

CA1307:为了清晰起见，请指定 StringComparison

2021/11/16 •

■ ID	CA1307
■	全球化
▬▬▬	非中断

原因

字符串比较运算使用不设置 StringComparison 参数的方法重载。

规则说明

许多字符串比较操作都提供重载，其接受 StringComparison 枚举值作为参数。

每当存在接受 StringComparison 参数的重载时，应使用此重载，而不使用不接受此参数的重载。通过显式设置此参数，你的代码通常更清晰、更易于维护。有关详细信息，请参阅显式指定字符串比较。

> **NOTE**
> 此规则不考虑比较方法使用的默认 StringComparison 值。因此，对于默认使用 `Ordinal` 字符串比较的方法和打算使用此默认比较模式的用户来说可能产生干扰。如果你只想查看默认使用区域性特定的字符串比较的已知字符串方法的冲突，请转而使用 CA1310：指定 StringComparison 以获得正确性。

如何解决冲突

若要解决此规则的冲突，请将字符串比较方法更改为接受 StringComparison 枚举作为参数的重载。例如，将 `str1.IndexOf(ch1)` 更改为 `str1.IndexOf(ch1, StringComparison.Ordinal)`。

何时禁止显示警告

当不需要明确意图时，可禁止显示此规则的警告。例如，测试代码或不可本地化的代码可能不需要它。

请参阅

- 有关使用 .NET 中字符串的最佳做法
- 全球化规则
- CA1310：为了确保正确，请指定 StringComparison
- CA1309:使用按顺序的 StringComparison

CA1308:将字符串规范化为大写

2021/11/16 •

■ ID	CA1308
■	全球化
▬▬	非中断

原因

操作将字符串规范化为小写。

规则说明

字符串应正常化为大写字母。少量字符转换为小写后不能再转换回来。若要转换回来，需将字符从一个区域设置转换为另一个表示字符数据的区域设置，然后从转换后的字符中准确检索原始字符。

如何解决冲突

更改将字符串转换为小写的操作，以便将字符串转换为大写。例如，将 `String.ToLower(CultureInfo.InvariantCulture)` 更改为 `String.ToUpper(CultureInfo.InvariantCulture)`。

何时禁止显示警告

当不基于规范化的结果做出安全决策时（例如，在 UI 中显示结果时），可禁止显示此规则的警告。

另请参阅

- 有关比较字符串的最佳做法
- 全球化规则

CA1309:使用按顺序的 StringComparison

2021/11/16 •

	▮
■ ID	CA1309
■	全球化
▬▬	非中断

原因

非语义的字符串比较运算不会将 StringComparison 参数设置为 Ordinal 或 OrdinalIgnoreCase 。

规则说明

许多字符串操作(最重要的是 System.String.Compare 和 System.String.Equals 方法)现在提供重载，后者接受 System.StringComparison 枚举值作为参数。

指定 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 时，字符串比较是非语义的 。也就是说，在制定比较决策时，将忽略特定于自然语言的特征。忽略自然语言特征意味着决策基于简单的字节比较，而不是按区域性参数化的大小写或相等表。因此，通过将参数显式设置为 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase，通常可以提高代码的速度、正确性和可靠性 。

如何解决冲突

若要解决此规则的冲突，请将字符串比较方法更改为接受 System.StringComparison 枚举为参数的重载，并指定 Ordinal 或 OrdinalIgnoreCase 。例如，将 `String.Compare(str1, str2)` 更改为 `String.Compare(str1, str2, StringComparison.Ordinal)` 。

何时禁止显示警告

当库或应用程序仅用于有限的本地受众时，或应使用当前区域性的语义时，可忽略此规则的警告。

另请参阅

- 全球化规则
- CA1307:指定 StringComparison

CA1310：为了确保正确，请指定 StringComparison

2021/11/16 •

■ ID	CA1310
■	全球化
▬▬▬	非中断

原因

字符串比较操作使用未设置 StringComparison 参数的方法重载，并默认使用区域性特定的字符串比较。因此，它的行为会因当前用户的区域设置而异。

规则说明

默认使用区域性特定字符串比较的字符串比较方法可能具有与用户意图不匹配的潜在意外运行时行为。建议使用具有 StringComparison 参数的重载，以确保意图的正确性和清晰性。

此规则将标记默认使用区域性特定 StringComparison 值的字符串比较方法。有关详细信息，请参阅使用当前区域性的字符串比较。

> **NOTE**
> 如果想查看所有字符串比较方法的冲突(无论该方法使用什么样的默认字符串比较)，请转而使用 CA1307：指定 StringComparison 以确保清晰性。

如何解决冲突

若要解决此规则的冲突，请将字符串比较方法更改为接受 StringComparison 枚举作为参数的重载。例如，将 `String.Compare(str1, str2)` 更改为 `String.Compare(str1, str2, StringComparison.Ordinal)`。

何时禁止显示警告

当不打算本地化库或应用程序时，可禁止显示此规则的警告。

请参阅

- 有关使用 .NET 中字符串的最佳做法
- 全球化规则
- CA1307:为了清晰起见，请指定 StringComparison
- CA1309:使用按顺序的 StringComparison

CA2101：指定对 P/Invoke 字符串参数进行封送处理

2021/11/16 •

■ ID	CA2101
■	全球化
■■■■■	非中断

原因

某平台调用成员允许部分受信任的调用方，具有一个字符串参数，并且不显式封送该字符串。

规则说明

从 Unicode 转换为 ANSI 时，可能并非所有 Unicode 字符都可在特定 ANSI 代码页中表示。最佳映射尝试用某个字符替换不能表示的字符来解决这个问题。使用此功能可能会导致潜在的安全漏洞，因为你无法控制所选的字符。例如，恶意代码可能会有意创建一个 Unicode 字符串，其中包含未在特定代码页中找到的字符，这些字符将转换为文件系统特殊字符，如“..” 或“/”。另请注意，在将字符串转换为 ANSI 之前，通常会对特殊字符进行安全性检查。

最佳映射是非托管转换(WChar 到 MByte)的默认值。除非显式禁用最佳映射，否则代码可能会因此问题而包含可利用的安全漏洞。

Caution

不应将代码访问安全性 (CAS) 视为安全边界。

如何解决冲突

若要解决与此规则的冲突，请显式封送字符串数据类型。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例显示一个与此规则冲突的方法，并演示如何解决冲突。

```
class NativeMethods
{
    // Violates rule: SpecifyMarshalingForPInvokeStringArguments.
    [DllImport("advapi32.dll", CharSet = CharSet.Auto)]
    internal static extern int RegCreateKey(IntPtr key, String subKey, out IntPtr result);

    // Satisfies rule: SpecifyMarshalingForPInvokeStringArguments.
    [DllImport("advapi32.dll", CharSet = CharSet.Unicode)]
    internal static extern int RegCreateKey2(IntPtr key, String subKey, out IntPtr result);
}
```

可移植性和互操作性规则

2021/11/16 •

可移植性规则支持跨不同平台的可移植性。互操作性规则支持与 COM 客户端交互。

本节内容

"	"
CA1401：P/Invokes 应为不可见	公共类型中的公共或受保护方法具有 System.Runtime.InteropServices.DllImportAttribute 属性（在 Visual Basic 中由 Declare 关键字实现）。这些方法不能公开。
CA1416：验证平台兼容性	在组件上使用依赖于平台的 API 会使代码无法用于所有平台。
CA1417：请勿对 P/Invokes 的字符串参数使用 `OutAttribute`	如果该字符串为暂存的字符串，则通过包含 `OutAttribute` 的值传递的字符串参数可能使运行时变得不稳定。
CA1418：使用有效的平台字符串	平台兼容性分析器需要有效的平台名称和版本。

CA1401：P/Invokes 应该是不可见的

2021/11/16 •

■ ID	CA1401
■	互操作性
■■■■	重大

原因

公共类型中的公共或受保护方法具有 System.Runtime.InteropServices.DllImportAttribute 特性（在 Visual Basic 中也由 `Declare` 关键字实现）。

规则说明

标记有 DllImportAttribute 特性的方法（或在 Visual Basic 中使用 `Declare` 关键字定义的方法）使用平台调用服务来访问非托管代码。这些方法不能公开。通过使这些方法保持专用或内部，可以阻止调用方访问他们不能调用的非托管 API，从而确保库不会破坏安全性。

如何解决冲突

若要解决此规则的冲突，请更改该方法的访问级别。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例声明了违反此规则的方法。

```
// Violates rule: PInvokesShouldNotBeVisible.
public class NativeMethods
{
    [DllImport("kernel32.dll", CharSet = CharSet.Unicode)]
    public static extern bool RemoveDirectory(string name);
}
```

```
Imports System

Namespace ca1401

    ' Violates rule: PInvokesShouldNotBeVisible.
    Public Class NativeMethods
        Public Declare Function RemoveDirectory Lib "kernel32" (
        ByVal Name As String) As Boolean
    End Class

End Namespace
```

CA1416：验证平台兼容性

2021/11/16 •

■ ID	CA1416
■	互操作性
▬▬	非中断

原因

如果特定于平台的 API 用于其他平台的上下文中，或者平台未经过验证(平台中立)，规则将报告违规。如果你使用项目的目标平台不支持的 API，规则也会报告违规。

默认情况下，此规则只对面向 .NET 5 或更高版本的项目启用。但是，可以为面向其他框架的项目启用该分析器。

规则说明

.NET 5.0 增加了新属性(SupportedOSPlatformAttribute 和 UnsupportedOSPlatformAttribute)，用于为特定于平台的 API 添加注释。可以使用或不使用作为平台名称一部分的版本号对两个属性进行实例化。它们还可以在不同平台应用多次。

- 未加注释的 API 被视为可用于所有操作系统 (OS) 平台。
- 标记为 `[SupportedOSPlatform("platformName")]` 的 API 被视为只能移植到指定 OS 平台。如果该平台是另一个平台的一部分，则该属性意味着该平台也受支持。
- 标记为 `[UnsupportedOSPlatform("platformName")]` 的 API 被视为在指定 OS 平台上不受支持。如果该平台是另一个平台的一部分，则该属性意味着该平台也不受支持。

可以在单个 API 上合并 `[SupportedOSPlatform]` 和 `[UnsupportedOSPlatform]` 属性。在这种情况下，下列规则适用：

- 允许列表。如果每个 OS 平台的最低版本是 `[SupportedOSPlatform]` 属性，则 API 会被视为仅在列出的平台上受支持，但在所有其他平台上不受支持。此列表的 `[UnsupportedOSPlatform]` 属性可能包含相同的平台，但只有更高版本，这表示 API 已从该版本中删除。
- 拒绝列表。如果每个 OS 平台的最低版本是 `[UnsupportedOSPlatform]` 属性，则 API 会被视为仅在列出的平台上不受支持，但在所有其他平台上受支持。此列表的 `[SupportedOSPlatform]` 属性可能包含相同的平台，但只有更高版本，这表示从该版本开始支持 API。
- 不一致的列表。如果某些平台的最低版本是 `[SupportedOSPlatform]`，但其他平台的是 `[UnsupportedOSPlatform]`，则此组合被视为不一致。API 上的某些注释将被忽略。将来，我们可能会引入一个在不一致时生成警告的分析器。

如果从其他平台的上下文中访问使用这些属性批注的 API，则可以看到 CA1416 冲突。

TFM 目标平台

分析器不会检查来自 MSBuild 属性的目标框架名字对象 (TFM) 目标平台 `<TargetFramework>` 或 `<TargetFrameworks>`。如果 TFM 具有目标平台，MSBuild 会在 AssemblyInfo.cs 文件中注入带有目标平台名称的 `SupportedOSPlatform` 属性，该文件由分析器使用。例如，如果 TFM 是 `net5.0-windows10.0.19041`，MSBuild 会将

`[assembly: System.Runtime.Versioning.SupportedOSPlatform("windows10.0.19041")]` 属性注入 AssemblyInfo.cs 文件，整个程序集被视为仅限 Windows。因此，调用版本 7.0 或以下的仅限 Windows 的 API 不会导致项目中出现警告。

> **NOTE**
> 如果禁用项目的 AssemblyInfo.cs 文件生成（即 `<GenerateAssemblyInfo>` 属性设置为 `false`），则无法由 MSBuild 添加所需的程序集级别 `SupportedOSPlatform` 属性。在这种情况下，即使面向该平台，也会出现特定于平台的 API 使用情况的警告。若要解决警告，请启用 AssemblyInfo.cs 文件生成或在项目中手动添加属性。

冲突

- 从其他平台可访问的代码中访问仅在指定平台（`[SupportedOSPlatform("platformName")]`）上受支持的 API 时，将看到以下违规行为："platformName"上支持"API"。

```
// An API supported only on Linux.
[SupportedOSPlatform("linux")]
public void LinuxOnlyApi() { }

// API is supported on Windows, iOS from version 14.0, and MacCatalyst from version 14.0.
[SupportedOSPlatform("windows")]
[SupportedOSPlatform("ios14.0")] // MacCatalyst is a superset of iOS, therefore it's also supported.
public void SupportedOnWindowsIos14AndMacCatalyst14() { }

public void Caller()
{
    LinuxOnlyApi(); // This call site is reachable on all platforms. 'LinuxOnlyApi()' is only
supported on: 'linux'

    SupportedOnWindowsIos14AndMacCatalyst14(); // This call site is reachable on all platforms.
'SupportedOnWindowsIos14AndMacCatalyst14()'
                                               // is only supported on: 'windows', 'ios' 14.0 and
later, 'MacCatalyst' 14.0 and later.
}
```

> **NOTE**
> 只有当项目未面向支持的平台（`net5.0-differentPlatform`）时，才会发生违规行为。这同样适用于多目标项目。如果项目面向指定的平台（`net5.0-platformName`）并且为项目启用了 AssemblyInfo.cs 文件生成，则不会发生冲突。

- 从面向不受支持的平台上下文中访问具有 `[UnsupportedOSPlatform("platformName")]` 属性的 API 时，将发生以下违规行为："platformName"上不支持"API"。

```
// An API not supported on Android but supported on all other platforms.
[UnsupportedOSPlatform("android")]
public void DoesNotWorkOnAndroid() { }

// An API was unsupported on Windows until version 10.0.18362.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows10.0.18362")]
public void StartedWindowsSupportFromCertainVersion() { }

public void Caller()
{
    DoesNotWorkOnAndroid(); // This call site is reachable on all platforms.'DoesNotWorkOnAndroid()'
is unsupported on: 'android'

    StartedWindowsSupportFromCertainVersion(); // This call site is reachable on all platforms.
'StartedWindowsSupportFromCertainVersion()' is unsupported on: 'windows' 10.0.18362 and before
}
```

NOTE

如果你要生成的应用不以不受支持的平台为目标，则不会发生任何违规行为。只有在以下情况下，才会发生违规行为：

- 项目面向已被归为不受支持类别的平台。
- `platformName` 包含在默认的 MSBuild `<SupportedPlatform>` 项组中。
- `platformName` 通过手动方式包含在 MSBuild `<SupportedPlatform>` 项目组中。

```
<ItemGroup>
    <SupportedPlatform Include="platformName" />
</ItemGroup>
```

如何解决冲突

若要解决违规问题，建议确保只在相应的平台上运行时调用特定于平台的 API。为此，请在生成时使用 `#if` 和多目标排除代码，或者在运行时有条件地调用代码。分析器会识别 OperatingSystem 类和 System.Runtime.InteropServices.RuntimeInformation.IsOSPlatform 中的平台保护。

- 通过使用标准平台临界方法或带有 `SupportedOSPlatformGuardAttribute` 或 `UnsupportedOSPlatformGuardAttribute` 注释的自定义临界 API 围绕调用站点，阻止违反。

```
// An API supported only on Linux.
[SupportedOSPlatform("linux")]
public void LinuxOnlyApi() { }

// API is supported on Windows, iOS from version 14.0, and MacCatalyst from version 14.0.
[SupportedOSPlatform("windows")]
[SupportedOSPlatform("ios14.0")] // MacCatalyst is a superset of iOS, therefore it's also supported.
public void SupportedOnWindowsIos14AndMacCatalyst14() { }

public void Caller()
{
    LinuxOnlyApi(); // This call site is reachable on all platforms. 'LinuxOnlyApi()' is only
supported on: 'linux'.

    SupportedOnWindowsIos14AndMacCatalyst14(); // This call site is reachable on all platforms.
'SupportedOnWindowsIos14AndMacCatalyst14()'
                                               // is only supported on: 'windows', 'ios' 14.0 and
later, 'MacCatalyst' 14.0 and later.
}
```

```
[SupportedOSPlatformGuard("windows")]  // The platform guard attributes used
[SupportedOSPlatformGuard("ios14.0")]
private readonly bool _isWindowOrIOS14 = OperatingSystem.IsWindows() ||
OperatingSystem.IsIOSVersionAtLeast(14);

// The warnings are avoided using platform guard methods.
public void Caller()
{
    if (OperatingSystem.IsLinux()) // standard guard examples
    {
        LinuxOnlyApi(); // no diagnostic
    }

    if (OperatingSystem.IsIOSVersionAtLeast(14))
    {
        SupportedOnWindowsAndIos14(); // no diagnostic
    }

    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        SupportedOnWindowsAndIos14(); // no diagnostic
    }

    if (_isWindowOrMacOS14) // custom guard example
    {
        SupportedOnWindowsAndIos14(); // no diagnostic
    }
}

// An API not supported on Android but supported on all other platforms.
[UnsupportedOSPlatform("android")]
public void DoesNotWorkOnAndroid() { }

// An API was unsupported on Windows until version 10.0.18362.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows10.0.18362")]
public void StartedWindowsSupportFromCertainVersion();

public void Caller()
{
    DoesNotWorkOnAndroid(); // This call site is reachable on all platforms.'DoesNotWorkOnAndroid()'
is unsupported on: 'android'

    StartedWindowsSupportFromCertainVersion(); // This call site is reachable on all platforms.
'StartedWindowsSupportFromCertainVersion()' is unsupported on: 'windows' 10.0.18362 and before.
}

[UnsupportedOSPlatformGuard("android")] // The platform guard attribute
bool IsNotAndroid => !OperatingSystem.IsAndroid();

public void Caller()
{
    if (!OperatingSystem.IsAndroid()) // using standard guard methods
    {
        DoesNotWorkOnAndroid(); // no diagnostic
    }

    // Use the && and || logical operators to guard combined attributes.
    if (!OperatingSystem.IsWindows() || OperatingSystem.IsWindowsVersionAtLeast(10, 0, 18362))
    {
        StartedWindowsSupportFromCertainVersion(); // no diagnostic
    }

    if (IsNotAndroid) // custom guard example
    {
        DoesNotWorkOnAndroid(); // no diagnostic
    }
```

```
}
```

- 分析器还会使用 System.Diagnostics.Debug.Assert 来防止不受支持的平台访问代码。使用 `Debug.Assert` 时，可以根据需要从发行版本中去除此检查。

```
// An API supported only on Linux.
[SupportedOSPlatform("linux")]
public void LinuxOnlyApi() { }

public void Caller()
{
    Debug.Assert(OperatingSystem.IsLinux());

    LinuxOnlyApi(); // No diagnostic
}
```

- 可以选择将自己的 API 标记为特定于平台，从而有效地将要求转发给调用方。可以将平台属性应用于以下任何 API：
  - 类型
  - 成员（方法、字段、属性和事件）
  - 程序集

```
[SupportedOSPlatform("windows")]
[SupportedOSPlatform("ios14.0")]
public void SupportedOnWindowsAndIos14() { }

[SupportedOSPlatform("ios15.0")] // call site version should be equal to or higher than the API
version
public void Caller()
{
    SupportedOnWindowsAndIos14(); // No diagnostics
}

[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows10.0.18362")]
public void StartedWindowsSupportFromCertainVersion();

[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows10.0.18362")]
public void Caller()
{
    StartedWindowsSupportFromCertainVersion(); // No diagnostics
}
```

- 在应用程序集或类型级别的属性时，程序集或类型中的所有成员都被视为特定于平台。

```
[assembly:SupportedOSPlatform("windows")]
public namespace ns
{
    public class Sample
    {
        public void SupportedOnWindows() { }

        public void Caller()
        {
            SupportedOnWindows(); // No diagnostic as call site and calling method both windows only
        }
    }
}
```

何时禁止显示警告

在没有适当平台上下文或不受保护的情况下，不建议引用特定于平台的 API。不过，你可以通过常规方式（`<NoWarn>`、.editorconfig 文件或 `#pragma`）抑制这些诊断：

```
[SupportedOSPlatform("linux")]
public void LinuxOnlyApi() { }

public void Caller()
{
#pragma warning disable CA1416
    LinuxOnlyApi();
#pragma warning restore CA1416
}
```

配置代码以进行分析

默认情况下，只有当项目面向 .NET 5 或更高版本并且其 AnalysisLevel 为 5 或更高值时，分析器才会启用。若要为低于 `net5.0` 的目标框架启用它，可以向项目中的 .editorconfig 文件添加以下键值对：

```
dotnet_code_quality.enable_platform_analyzer_on_pre_net5_target=true
```

请参阅

- 平台兼容性分析器（概念上）
- 批注特定于平台的 API 并检测其用法
- 在特定平台上将 API 批注为不受支持
- .NET 5 中的目标框架名称
- 互操作性规则

CA1417：不要对 P/Invokes 的字符串参数使用 OutAttribute

2021/11/16 •

■ ID	CA1417
■	互操作性
■■■■■	非中断

原因

P/Invoke 字符串参数通过值传递并标记有 OutAttribute。

规则说明

.NET 运行时自动执行字符串集中。如果将标记有 OutAttribute 的集中字符串按值传递给 P/Invoke，则运行时可能会不稳定。

如何解决冲突

如果需要将修改后的字符串数据封送回调用方，请改为按引用传递字符串。否则，无需进行任何其他更改即可删除 OutAttribute。

```
 // Violation
[DllImport("MyLibrary")]
private static extern void Foo([Out] string s);

// Fixed: passed by reference
[DllImport("MyLibrary")]
private static extern void Foo(out string s);

// Fixed: marshaling data back to caller is not required
[DllImport("MyLibrary")]
private static extern void Foo(string s);
```

何时禁止显示警告

不可禁止显示此规则的警告。

另请参阅

- 互操作性规则
- 本机互操作性最佳做法

CA1418：验证平台兼容性

2021/11/16 •

■ ID	CA1418
■	互操作性
▬▬	非中断

原因

平台兼容性分析器需要有效的平台名称和版本。如果提供给 OSPlatformAttribute 构造函数的平台字符串包含未知平台名称，或者可选版本部分无效，则会报告违规。

规则说明

派生自 OSPlatformAttribute 的平台兼容性属性在操作系统 (OS) 平台名称中使用字符串字面量和可选版本部分。该字符串应包含已知平台名称，不包含版本部分或包含有效版本部分。

已知平台名称列表由两部分构成：

- 名为 `OperatingSystem.Is<PlatformName>[VersionAtLleast]()` 的 OperatingSystem 保护方法的 `PlatformName` 部分。例如，保护方法 OperatingSystem.IsWindows() 将 `Windows` 添加到已知平台名称列表。
- 由 `SupportedPlatform` 项构成的项目的 MSBuild 项组，其中包括默认的 MSBuild SupportedPlatforms 列表。这是已知平台的项目特定信息。它允许类库作者将更多平台添加到已知平台列表。例如：

```
<ItemGroup>
  <SupportedPlatform Include="PlatformName" />
</ItemGroup>
```

如果平台字符串包含版本部分，则它应是具以下格式的有效 Version：`major.minor[.build[.revision]]`。

冲突

- `Solaris` 是未知平台名称，因为它未包含在默认的 MSBuild SupportedPlatforms 列表中，并且 OperatingSystem 类中没有名为 `OperatingSystem.IsSolaris()` 的保护方法。

```
[SupportedOSPlatform("Solaris")] // Warns: The platform 'Solaris' is not a known platform name.
public void SolarisApi() { }
```

- `Android` 是已知平台，因为 OperatingSystem 类型中有 OperatingSystem.IsAndroid() 保护方法。但版本部分不是有效版本。它应至少有两个以点分隔的整数。

```
[UnsupportedOSPlatform("Android10")] // Warns: Version '10' is not valid for platform 'Android'. Use
a version with 2-4 parts for this platform.
public void DoesNotWorkOnAndroid() { }
```

- `Linux` 是已知平台，因为它包含在默认的 MSBuild SupportedPlatforms 列表中，并且有名为 OperatingSystem.IsLinux() 的保护方法。但对于 `Linux` 平台，没有 `System.OperatingSystem.IsLinuxVersionAtLeast(int,int)` 等版本化保护方法，因此 Linux 不支持版本部分。

```
[SupportedOSPlatform("Linux4.8")] // Warns: Version '4.8' is not valid for platform 'Linux'. Do not
use versions for this platform.
public void LinuxApi() { }
```

如何解决冲突

- 将平台更改为已知平台名称。
- 如果平台名称正确，并且你希望使其成为已知平台，请将其添加到项目文件中的 MSBuild SupportedPlatforms 列表：

```
  <ItemGroup>
    <SupportedPlatform Include="Solaris" />
  </ItemGroup>
```

```
[SupportedOSPlatform("Solaris")] // No warning
public void SolarisApi() { }
```

- 修复无效版本。例如，对于 `Android`，`10` 不是有效版本，而 `10.0` 是有效版本。

```
// Before
[UnsupportedOSPlatform("Android10")] // Warns: Version '10' is not valid for platform 'Android'. Use
a version with 2-4 parts for this platform.
public void DoesNotWorkOnAndroid() { }

// After
[UnsupportedOSPlatform("Android10.0")] // No warning.
public void DoesNotWorkOnAndroid() { }
```

- 如果平台不支持某一版本，请删除对应版本部分。

```
// Before
[SupportedOSPlatform("Linux4.8")] // Warns: Version '4.8' is not valid for platform 'Linux'. Do not
use versions for this platform.
public void LinuxApi() { }

// After
[SupportedOSPlatform("Linux")] // No warning.
public void LinuxApi() { }
```

何时禁止显示警告

不建议使用未知平台名称或无效版本。不过，可通过常规方式（`<NoWarn>`、.editorconfig 文件或 `#pragma`）阻止这些诊断，例如：

```
#pragma warning disable CA1418
[SupportedOSPlatform("platform")]
#pragma warning restore CA1418
public void PlatformSpecificApi() { }
```

另请参阅

- CA1416 平台兼容性分析器
- 平台兼容性分析器(概念上)
- 互操作性规则

可维护性规则

2021/11/16 •

可维护性规则支持库和应用程序维护。

在本节中

"	"
CA1501:避免过度继承	类型在继承层次结构中的深度超过四级。深度嵌套的类型层次结构可能很难遵循、理解和维护。
CA1502:避免过度复杂	此规则通过方法来测量线性独立的路径的数量，该数量是由条件分支的数量和复杂度决定的。
CA1505:避免使用无法维护的代码	类型或方法具有较低的可维护性索引值。如果可维护性指数较低，则表示类型或方法可能难以维护，最好重新进行设计。
CA1506:避免过度类耦合度	此规则通过计算类型或方法包含的唯一类型引用的个数来衡量类耦合。
CA1507：使用 nameof 代替字符串	字符串字面量用作参数，可在其中使用 `nameof` 表达式。
CA1508：避免死条件代码	方法具有在运行时计算结果始终为 `true` 或 `false` 的条件代码。这会导致条件的 `false` 分支中出现死代码。
CA1509：代码度量配置文件中的条目无效	代码度量规则（如 CA1501、CA1502、CA1505 和 CA1506）提供了具有无效条目的名为 `CodeMetricsConfig.txt` 的配置文件。

另请参阅

- 测量托管代码的复杂性和可维护性

CA1501:避免过度继承

2021/11/16 •

■ ID	CA1501
■	可维护性
■■■■■	重大

原因

类型在继承层次结构中的深度超过四级。

默认情况下，该规则仅排除 `System` 命名空间中的类型，但这是可配置的。

规则说明

深度嵌套的类型层次结构可能很难遵循、理解和维护。此规则将分析限制在同一模块内的层次结构中。

如何解决冲突

要解决此规则的冲突，请从继承层次结构中不太深入的基类型中派生类型，或者删除部分中间基类型。

何时禁止显示警告

可禁止显示此规则发出的警告。但代码可能较难维护。请注意，根据基类型的可见性，解决此规则的冲突可能会造成中断性变更。例如，删除公共基类型是一项中断性变更。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 继承排除的类型或命名空间名称

你可以仅为此规则、为所有规则或为此类别（可维护性）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

继承排除的类型或命名空间名称

可以将规则配置为从继承层次结构树中排除某些类型或命名空间。默认情况下，将排除 `System.*` 命名空间中的所有类型。无论你设置什么值，都将添加此默认值。

■■■	■■
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = MyType`	匹配所有名为 `MyType` 的类型或其命名空间包含 `MyType` 的类型（以及 `System` 命名空间中的所有类型）

[[[	[[
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = MyType1|MyType2`	匹配所有名为 `MyType1` 或 `MyType2` 的类型，或其命名空间包含 `MyType1` 或 `MyType2` 的类型（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = T:NS.MyType`	匹配 `NS` 命名空间中的特定类型 `MyType`（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = T:NS1.MyType1|T:NS2.MyType2`	匹配具有相应的完全限定名称的特定类型 `MyType1` 和 `MyType2`（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = N:NS`	匹配 `NS` 命名空间中的所有类型（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = My*`	匹配其名称以 `My` 开头的所有类型，或其命名空间部分以 `My` 开头的所有类型（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = T:NS.My*`	匹配 `NS` 命名空间中名称以 `My` 开头的所有类型（以及 `System` 命名空间中的所有类型）
`dotnet_code_quality.CA1501.additional_inheritance_excluded_symbol_names = N:My*`	匹配命名空间以 `My` 开头的所有类型（以及 `System` 命名空间中的所有类型）

示例

下面的示例演示了与此规则冲突的类型：

```
class BaseClass {}
class FirstDerivedClass : BaseClass {}
class SecondDerivedClass : FirstDerivedClass {}
class ThirdDerivedClass : SecondDerivedClass {}
class FourthDerivedClass : ThirdDerivedClass {}

// This class violates the rule.
class FifthDerivedClass : FourthDerivedClass {}
```

```
Imports System

Namespace ca1501

    Class BaseClass
    End Class

    Class FirstDerivedClass
        Inherits BaseClass
    End Class

    Class SecondDerivedClass
        Inherits FirstDerivedClass
    End Class

    Class ThirdDerivedClass
        Inherits SecondDerivedClass
    End Class

    Class FourthDerivedClass
        Inherits ThirdDerivedClass
    End Class

    ' This class violates the rule.
    Class FifthDerivedClass
        Inherits FourthDerivedClass
    End Class

End Namespace
```

CA1502:避免过度复杂

2021/11/16 •

■ ID	CA1502
■	可维护性
▬▬	非中断

原因

某方法具有过度的圈复杂度。

规则说明

圈复杂度衡量经过该方法的线性独立路径的数量，由条件分支的数量和复杂度决定。圈复杂度较低通常表示方法易于理解、测试和维护。圈复杂度通过方法的控制流图计算得出，公式如下：

圈复杂度 = 边缘数 - 节点数 + 1

节点表示逻辑分支点，边缘表示节点之间的线 。

当圈复杂度大于 25 时，规则会报告冲突。

可以在衡量托管代码的复杂性中了解有关代码度量的详细信息。

如何解决冲突

若要解决此规则的冲突，请重构方法以降低其圈复杂度。

何时禁止显示警告

如果无法轻松降低复杂度，并且该方法易于理解、测试和维护，则可禁止显示此规则的警告。具体而言，包含较大 `switch`（在 Visual Basic 中为 `Select`）语句的方法就是可排除的候选方法。在开发周期后期破坏代码库稳定性或在先前发布的代码中引入意外的运行时行为更改的风险可能超过重构代码的可维护性优势。

如何计算圈复杂度

圈复杂度的计算方法是，将以下各项加 1：

- 分支数（例如 `if`、`while` 和 `do`）
- `switch` 中的 `case` 语句数

示例

下面的示例演示具有不同圈复杂度的方法。

圈复杂度为 1

```
public void Method()
{
    Console.WriteLine("Hello World!");
}
```

```
Public Sub Method()
    Console.WriteLine("Hello World!")
End Sub
```

圈复杂度为 2

```
void Method(bool condition)
{
    if (condition)
    {
        Console.WriteLine("Hello World!");
    }
}
```

```
Public Sub Method(ByVal condition As Boolean)
    If (condition) Then
        Console.WriteLine("Hello World!")
    End If
End Sub
```

圈复杂度为 3

```
public void Method(bool condition1, bool condition2)
{
    if (condition1 || condition2)
    {
        Console.WriteLine("Hello World!");
    }
}
```

```
Public Sub Method(ByVal condition1 As Boolean, ByVal condition2 As Boolean)
    If (condition1 OrElse condition2) Then
        Console.WriteLine("Hello World!")
    End If
End Sub
```

圈复杂度为 8

```
public void Method(DayOfWeek day)
{
    switch (day)
    {
        case DayOfWeek.Monday:
            Console.WriteLine("Today is Monday!");
            break;
        case DayOfWeek.Tuesday:
            Console.WriteLine("Today is Tuesday!");
            break;
        case DayOfWeek.Wednesday:
            Console.WriteLine("Today is Wednesday!");
            break;
        case DayOfWeek.Thursday:
            Console.WriteLine("Today is Thursday!");
            break;
        case DayOfWeek.Friday:
            Console.WriteLine("Today is Friday!");
            break;
        case DayOfWeek.Saturday:
            Console.WriteLine("Today is Saturday!");
            break;
        case DayOfWeek.Sunday:
            Console.WriteLine("Today is Sunday!");
            break;
    }
}
```

```
Public Sub Method(ByVal day As DayOfWeek)
    Select Case day
        Case DayOfWeek.Monday
            Console.WriteLine("Today is Monday!")
        Case DayOfWeek.Tuesday
            Console.WriteLine("Today is Tuesday!")
        Case DayOfWeek.Wednesday
            Console.WriteLine("Today is Wednesday!")
        Case DayOfWeek.Thursday
            Console.WriteLine("Today is Thursday!")
        Case DayOfWeek.Friday
            Console.WriteLine("Today is Friday!")
        Case DayOfWeek.Saturday
            Console.WriteLine("Today is Saturday!")
        Case DayOfWeek.Sunday
            Console.WriteLine("Today is Sunday!")
    End Select
End Sub
```

相关规则

CA1501:避免过度继承

另请参阅

- 测量托管代码的复杂性和可维护性

CA1505:避免使用无法维护的代码

2021/11/16 •

■ ID	CA1505
■	可维护性
▬▬	非中断

原因

类型或方法具有较低的可维护性索引值。

规则说明

可维护性指数通过使用以下度量进行计算:代码行、程序量和圈复杂度。程序量是基于代码中的运算符和操作数数量来理解类型或方法的难度的度量值。圈复杂度是类型或方法的结构化复杂度的度量值。可以在衡量托管代码的复杂性和可维护性中了解有关代码度量的详细信息。

如果可维护性指数较低，则表示类型或方法可能难以维护，最好重新进行设计。

如何解决冲突

若要解决此冲突，请重新设计类型或方法，并尝试将其拆分为更小、更有针对性的类型或方法。

何时禁止显示警告

如果类型或方法无法拆分或被视为可维护(尽管其非常大)，可禁止显示此警告。

另请参阅

- 可维护性规则
- 测量托管代码的复杂性和可维护性

CA1506:避免过度类耦合度

2021/11/16 •

■ ID	CA1506
■	可维护性
▬▬	重大

原因

类型或方法与许多其他类型耦合在一起。编译器生成的类型不包括在此指标中。

规则说明

此规则通过计算类型或方法包含的唯一类型引用的个数来衡量类耦合。默认耦合度阈值对于类型为 95，对于方法为 40。

很难维护类耦合度较高的类型和方法。最好拥有低耦合度和高内聚的类型和方法。

如何解决冲突

若要解决此冲突，请尝试重新设计类型或方法，以减少其耦合的类型的数量。

何时禁止显示警告

尽管类型或方法对其他类型有很多依赖项，当它被视为可维护时，请排除此警告。

另请参阅

- 可维护性规则
- 测量托管代码的复杂性和可维护性

CA1507：使用 `nameof` 代替字符串

2021/11/16 •

	[
■ ID	CA1507
■	可维护性
▬▬▬	非中断

原因

与包含方法的参数名或包含类型的属性名匹配的 `string` 文本或常数用作方法的参数。

规则说明

规则 CA1507 将 `string` 文本的使用标记为方法或构造函数的参数，其中 nameof（在 Visual Basic 中为 `NameOf`）表达式将添加可维护性。如果满足以下所有条件，则会触发规则：

- 参数是一个 `string` 文本或常数。
- 参数对应于方法的 `string` 类型化参数或正在调用的构造函数（即调用站点不涉及转换）。
- 可以是：
  - 参数的声明名称为 `paramName`，并且 `string` 文本的常数值与在其中调用方法或构造函数的方法、lambda 或本地函数的参数名称相匹配。
  - 参数的声明名称为 `propertyName`，并且 `string` 文本的常数值与在其中调用方法或构造函数的类型的属性名称相匹配。

在将来可以重命名参数，但不会错误地重命名 `string` 文本的情况下，规则 CA1507 可以提高代码的可维护性。使用 `nameof` 的情况下，当通过重构操作重命名该参数时，将重命名符号。此外，编译器将捕获参数名称中的任何拼写错误。

如何解决冲突

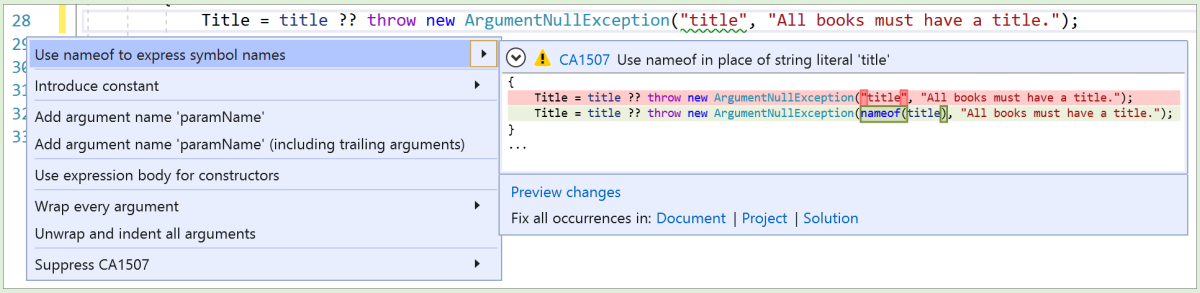
若要解决冲突，请将 `string` 文本替换为 nameof（在 Visual Basic 中为 `NameOf`）表达式。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
public Book(string title)
{
    // Violates rule CA1507
    Title = title ?? throw new ArgumentNullException("title", "All books must have a title.");
}
```

```
public Book(string title)
{
    // Resolves rule CA1507 violation
    Title = title ?? throw new ArgumentNullException(nameof(title), "All books must have a title.");
}
```

TIP

Visual Studio 中为此规则提供了代码修复。若要使用它，请将光标置于 `string` 文本上，然后按“Ctrl+.”（句点）。从显示的选项列表中选择“使用 nameof 表达符号名称”。



何时禁止显示警告

如果不考虑代码的可维护性，可安全地禁止显示此规则的冲突。

相关规则

- CA2208:正确实例化参数异常

另请参阅

- 可维护性规则

CA1508：避免死条件代码

2021/11/16 •

	[
■ ID	CA1508
■	可维护性
修复是中断修复还是非中断修复	非中断

原因

方法具有在运行时计算结果始终为 `true` 或 `false` 的条件代码。这会导致条件的 `false` 分支中出现死代码。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

方法可以具有条件代码，如 if 语句、二进制表达式（`==`、`!=`、`<`、`>`）、null 检查等。例如，请考虑以下代码：

```
public void M(int i, int j)
{
    if (i != 0)
    {
        return;
    }

    if (j != 0)
    {
        return;
    }

    // Below condition will always evaluate to 'false' as 'i' and 'j' are both '0' here.
    if (i != j)
    {
        // Code in this 'if' branch is dead code.
        // It can either be removed or refactored.
        ...
    }
}
```

C# 和 VB 编译器会分析与编译时常量值相关且计算结果始终为 `true` 或 `false` 的条件检查。此分析器会对非常量变量执行数据流分析，以确定与非常量值相关的冗余条件检查。在前面的代码中，对于到达 `i != j` 检查的所有代码路径，分析器确定 `i` 和 `j` 均为 `0`。因此，在运行时，此检查的计算结果将始终为 `false`。if 语句内的代码是死代码，可以删除或重构。同样，分析器还会跟踪变量是否为 null，并报告冗余 null 检查。

> **NOTE**
> 此分析器会对非常量值执行成本高昂的数据流分析。这可能会增加某些代码库的总体编译时间。

何时禁止显示警告

如果不在乎代码的可维护性，可安全地禁止显示此规则的冲突。还可以禁止显示标识为误报的冲突。存在可从多个线程执行的并发代码时，这些情况都可能发生。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

可以仅为此规则、为所有规则或为此类别（可维护性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

已	已
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

另请参阅

- 可维护性规则

CA1509：代码度量配置文件中的条目无效

2021/11/16 •

	[
■ ID	CA1509
■	可维护性
修复是中断修复还是非中断修复	非中断

原因

代码度量规则(如 CA1501、CA1502、CA1505 和 CA1506)提供了具有无效条目的名为 `CodeMetricsConfig.txt` 的配置文件。

规则说明

通过代码度量分析规则的 .NET 代码质量分析器实现，最终用户可以提供名为 `CodeMetricsConfig.txt` 的附加文件。此文件包含配置用于分析的代码度量阈值的条目。以下规则可在此文件中配置：

- CA1501:避免过度继承
- CA1502:避免过度复杂
- CA1505:避免使用无法维护的代码
- CA1506:避免过度类耦合度

此配置文件需要每个条目采用以下格式：

```
'RuleId'(Optional 'SymbolKind'): 'Threshold'
```

- “RuleId”的有效值为 `CA1501`、`CA1502`、`CA1505` 和 `CA1506`。
- 可选的“SymbolKind”的有效值为 `Assembly`、`Namespace`、`Type`、`Method`、`Field`、`Event` 和 `Property`。
- “阈值”的有效值为非负整数。
- 以“#”开头的行被视为注释行

例如，以下是有效的配置文件：

```
# Comment text

CA1501: 1

CA1502(Type): 4
CA1502(Method): 2
```

此配置文件中的无效条目使用 `CA1509` 诊断进行标记。

如何解决冲突

若要解决此规则的冲突，请确保 `CodeMetricsConfig.txt` 中的无效条目采用所需的格式。

何时禁止显示警告

请勿禁止显示此规则的冲突警告。

相关规则

- CA1501:避免过度继承
- CA1502:避免过度复杂
- CA1505:避免使用无法维护的代码
- CA1506:避免过度类耦合度

另请参阅

- 可维护性规则
- 测量托管代码的复杂性和可维护性

命名规则

2021/11/16 •

命名规则支持遵从 .NET 设计准则的命名约定。

本节内容

"	"
CA1700:不要命名“Reserved”枚举值	此规则假定当前不使用名称中包含“reserved”的枚举成员，而是将其作为一个占位符，以在将来的版本中重命名或移除它。重命名或移除成员是一项重大更改。
CA1707:标识符不应包含下划线	按照约定，标识符名称不包含下划线 (_) 字符。该规则将检查命名空间、类型、成员和参数。
CA1708:标识符应以大小写之外的差别进行区分	不能仅通过大小写区分命名空间、类型、成员和参数的标识符，因为针对公共语言运行时的语言不需要区分大小写。
CA1710:标识符应具有正确的后缀	按照约定，扩展某些基类型或实现某些接口的类型的名称，或者由这些类型派生的类型的名称应具有与相应基类型或接口关联的后缀。
CA1711:标识符应采用正确的后缀	按照约定，只有扩展某些基类型或实现某些接口的类型的名称或者从这些类型派生的类型的名称，应该以特定的保留后缀结尾。其他类型名称不应使用这些保留的后缀。
CA1712:不要将类型名用作枚举值的前缀	枚举成员的名称不使用类型名称作为前缀，因为类型信息将由开发工具提供。
CA1713:事件不应具有 before 或 after 前缀	事件的名称以“Before”或“After”开头。若要命名按特定顺序引发的相关事件，请使用现在时或过去时指示一系列操作中的相对位置。
CA1714:Flags 枚举应采用复数形式的名称	公共枚举具有 System.FlagsAttribute 特性并且其名称不是以“s”结尾。用 FlagsAttribute 标记的类型具有复数形式的名称，因为该特性指明可以指定多个值。
CA1715:标识符应具有正确的前缀	外部可见的接口的名称不以大写的“I”开头。外部可见的类型或方法上的泛型类型参数的名称不以大写的“T”开头。
CA1716:标识符不应与关键字冲突	某个命名空间名称或类型名称与编程语言中的保留关键字相同。命名空间和类型的标识符不应与针对公共语言运行时的语言所定义的关键字冲突。
CA1717:只有 FlagsAttribute 枚举应采用复数形式的名称	命名约定规定，复数形式的枚举名称表示可以同时指定多个枚举值。
CA1720:标识符不应包含类型名称	外部可见成员中的某个参数的名称包含一个数据类型名称，或者外部可见成员的名称包含一个语言特定的数据类型名称。

“	“
CA1721:属性名不应与 get 方法冲突	公共或受保护成员的名称以“Get”开头，且其余部分与公共或受保护属性的名称匹配。“Get”方法和属性的名称应能够明确区分其功能上的差异。
CA1724:类型名不应与命名空间冲突	类型名不应与 .NET 命名空间的名称匹配。与该规则冲突将使库的可用性下降。
CA1725:参数名应与基方法中的声明保持一致	以一致的方式命名重写层次结构中的参数可以提高方法重写的可用性。如果派生方法中的参数名与基声明中的名称不同，可能会导致无法区分出该方法是基方法的重写还是该方法的新重载。

CA1700：不要用“Reserved”命名枚举值

2021/11/16 •

	[
■ ID	CA1700
■	命名
▬▬▬	重大

原因

枚举成员的名称包含单词“reserved”。

规则说明

此规则假定当前不使用名称中包含“reserved”的枚举成员，而是将其作为一个占位符，以在将来的版本中重命名或移除它。重命名或移除成员是一项重大更改。不应期望用户仅因为成员名称包含“reserved”而忽略该成员，也不能指望用户阅读或遵守文档。此外，由于预留成员显示在对象浏览器和智能集成开发环境中，因此他们可能会造成混淆，以致于不知道实际使用哪些成员。

在将来的版本中，向枚举添加新成员，而不是使用预留成员。在大多数情况下，只要添加新成员不会导致原始成员的值发生变化，添加新成员就不是一项中断性变更。

在少数情况下，添加成员是一项中断性变更，即使原始成员保留其原始值也是如此。主要问题是，如果不中断对包含整个成员列表的返回值使用 `switch`（在 Visual Basic 中为 `Select`）语句并且在默认情况下引发异常的调用方，则不能从现有代码路径返回新成员。第二个问题是客户端代码可能无法处理反射方法（如 System.Enum.IsDefined）的行为更改。因此，如果新成员必须从现有方法返回，或者由于反射的使用不当而发生已知的应用程序不兼容，则唯一的非中断性解决方案是：

1. 添加包含原始成员和新成员的新枚举。
2. 使用 System.ObsoleteAttribute 属性标记原始枚举。

   对于公开原始枚举的任何外部可见类型或成员，请执行相同的过程。

如何解决冲突

若要解决此规则的冲突，请删除或重命名该成员。

何时禁止显示警告

对于当前使用的成员或以前发布的库，可以安全地禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规

则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

CA2217:不要使用 FlagsAttribute 标记枚举

CA1712:不要将类型名用作枚举值的前缀

CA1028:枚举存储应为 Int32

CA1008:枚举应具有零值

CA1027:用 FlagsAttribute 标记枚举

CA1707:标识符不应包含下划线

2021/11/16 •

	[
■ ID	CA1707
■	命名
修复是中断修复还是非中断修复	中断 - 在程序集上引发时 非中断 - 在类型参数上引发时

原因

标识符的名称包含下划线 (_) 字符。

规则说明

按照约定，标识符名称不包含下划线 (_) 字符。该规则将检查命名空间、类型、成员和参数。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库的学习曲线，让客户更加相信该库是由拥有托管代码开发专业知识的人员开发的。

如何解决冲突

删除名称中的所有下划线字符。

何时禁止显示警告

请勿禁止显示有关生产代码的警告。但对于测试代码，可安全地禁止显示此警告。可通过将警告的严重级别设置为“无”来禁止显示此规则的警告。

对于 Microsoft 代码中当前使用下划线且不能修改的已知方法，应禁止显示此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1708:标识符应以大小写之外的差别进行区分

CA1708:标识符应以大小写之外的差别进行区分

2021/11/16 •

	[
■ ID	CA1708
■	命名
■■■■■	重大

原因

两种类型、成员、参数或完全限定的命名空间的名称转换为小写时是相同的。

默认情况下，此规则仅查看外部可见的类型、成员和命名空间，但这是可配置的。

规则说明

不能仅通过大小写区分命名空间、类型、成员和参数的标识符，因为针对公共语言运行时的语言不需要区分大小写。例如，Visual Basic 是一种广泛使用的不区分大小写的语言。

此规则仅对公共可见成员触发。

如何解决冲突

选择与其他标识符比较时（不区分大小写）具有唯一性的名称。

何时禁止显示警告

不禁止显示此规则发出的警告。库可能无法用于 .NET 中的所有可用语言。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

冲突示例

下面的示例演示与此规则发生冲突的情况。

```
public class Class1
{
    protected string someProperty;

    public string SomeProperty
    {
        get { return someProperty; }
    }
}
```

CA1710:标识符应具有正确的后缀

2021/11/16 •

	ı
■ ID	CA1710
■	命名
▬▬▬	重大

原因

标识符的后缀不正确。

默认情况下，此规则仅查看外部可见的标识符，但这是可配置的。

规则说明

按照约定，扩展某些基类型或实现某些接口的类型的名称，或者由这些类型派生的类型的名称应具有与相应基类型或接口关联的后缀。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库所需的学习曲线，让客户更加有信心，相信该库是由拥有开发托管代码专业知识的人员开发的。

下表列出了具有关联后缀的基类型和接口。

ıtt/tt	SUFFIX
System.Attribute	属性
System.EventArgs	EventArgs
System.Exception	例外
System.Collections.ICollection	集合
System.Collections.IDictionary	字典
System.Collections.IEnumerable	集合
System.Collections.Generic.IReadOnlyDictionary<TKey,TValue>	字典
System.Collections.Queue	集合或队列
System.Collections.Stack	集合或堆栈
System.Collections.Generic.ICollection<T>	集合

类型/基类	SUFFIX
System.Collections.Generic.IDictionary<TKey,TValue>	字典
System.Data.DataSet	数据集
System.Data.DataTable	集合或 DataTable
System.IO.Stream	Stream
System.Security.IPermission	权限
System.Security.Policy.IMembershipCondition	条件
事件处理程序委托。	EventHandler

实现 ICollection 的类型是一种通用的数据结构类型(如字典、堆栈或队列),允许在名称中包含有关该类型预期用途的有用信息。

实现 ICollection 的类型是特定项的集合,其名称以单词 Collection 结尾。例如,Queue 对象的集合的名称会是 QueueCollection。Collection 后缀表示通过使用 `foreach`(Visual Basic中的 `For Each`)语句,可枚举该集合中的成员。

实现 IDictionary 或 IReadOnlyDictionary<TKey,TValue> 的类型的名称以单词 Dictionary 结尾,即使该类型还实现了 IEnumerable 或 ICollection 也是如此。Collection 和 Dictionary 后缀命名约定使用户能够区分以下两个枚举模式。

带有 Collection 后缀的类型遵循以下枚举模式。

```
foreach(SomeType x in SomeCollection) { }
```

带有 Dictionary 后缀的类型遵循以下枚举模式。

```
foreach(SomeType x in SomeDictionary.Values) { }
```

DataSet 对象由 DataTable 对象(由 System.Data.DataColumn 和 System.Data.DataRow 等对象的集合组成)的集合组成。这些集合通过 System.Data.InternalDataCollectionBase 基类实现 ICollection。

如何解决冲突

重命名该类型,使其带有正确的字词后缀。

何时禁止显示警告

如果类型是可扩展的或将保留任意一组不同的项的通用数据结构类型,则可禁止显示使用 Collection 后缀的警告。在这种情况下,可在名称中包含有关实现、性能或数据结构的其他特征的有用信息(例如 BinaryTree)。如果类型表示特定类型的集合(例如 StringCollection),请不要禁止显示此规则发出的警告,因为其后缀指示可使用 `foreach` 语句枚举该类型。

对于其他后缀,请勿禁止显示此规则发出的警告。通过后缀能够从类型名称中看出预期用途。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 排除间接基类型
- 其他所需的后缀

可以仅为此规则、为所有规则或为此类别(命名)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

排除间接基类型

可以配置是否从规则中排除间接基类型。默认情况下，此选项设置为 true，这会将分析限制为对当前的基类型执行。

```
dotnet_code_quality.CA1710.exclude_indirect_base_types = false
```

其他所需的后缀

可通过将以下键值对添加到项目中的 .editorconfig 文件，提供其他所需的后缀或重写某些硬编码的后缀的行为：

```
dotnet_code_quality.CA1710.additional_required_suffixes = [type]->[suffix]
```

用 `|` 这一字符来分隔多个值。可用以下任意一种格式指定类型：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

``	``
`dotnet_code_quality.CA1710.additional_required_suffixes = MyClass->Class`	从 MyClass 继承的所有类型都需要具有 Class 后缀。
`dotnet_code_quality.CA1710.additional_required_suffixes = MyClass->Class|MyNamespace.IPath->Path`	从 MyClass 继承的所有类型都需要具有 Class 后缀，实现 MyNamespace.IPath 的所有类型都必须具有 Path 后缀。
`dotnet_code_quality.CA1710.additional_required_suffixes = T:System.Data.IDataReader->{}`	重写内置后缀。在这种情况下，实现 IDataReader 的所有类型都不再需要以 Collection 结尾。

相关规则

CA1711:标识符应采用正确的后缀

请参阅

- 特性
- 处理和引发事件

CA1711:标识符应采用正确的后缀

2021/11/16 •

	▯
■ ID	CA1711
■	命名
▬▬	重大

原因

标识符的后缀不正确。

默认情况下，此规则仅查看外部可见的标识符，但这是可配置的。

规则说明

按照约定，只有扩展某些基类型或实现某些接口的类型的名称或者从这些类型派生的类型的名称，应以特定的保留后缀结尾。其他类型名称不应使用这些保留的后缀。

下表列出了保留的后缀以及与它们关联的基类型和接口。

SUFFIX	▯▯/▯▯
属性	System.Attribute
集合	System.Collections.ICollection System.Collections.IEnumerable System.Collections.Queue System.Collections.Stack System.Collections.Generic.ICollection<T> System.Data.DataSet System.Data.DataTable
字典	System.Collections.IDictionary System.Collections.Generic.IDictionary<TKey,TValue>
EventArgs	System.EventArgs
EventHandler	事件处理程序委托
例外	System.Exception

SUFFIX	或/和
权限	System.Security.IPermission
队列	System.Collections.Queue
堆栈	System.Collections.Stack
Stream	System.IO.Stream

此外，不应使用以下后缀：

- `Delegate`
- `Enum`
- `Impl`（请改用 `Core`）
- `Ex` 或类似的后缀，用于与同一类型的早期版本区分开来
- 枚举类型的 `Flag` 或 `Flags`

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库的学习曲线，让客户更加相信该库是由拥有托管代码开发专业知识的人员开发的。有关详细信息，请参阅命名准则：类、结构和接口。

如何解决冲突

从类型名称中删除后缀。

何时禁止显示警告

除非后缀在应用程序域中具有明确的含义，否则不要禁止显示来自此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 允许后缀

可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

允许后缀

你可以配置允许的后缀列表，每个后缀用竖线字符分隔（"|"）。例如，若要指定规则不得针对 Flag 或 Flags 前缀运行，请将以下键值对添加到项目的 .editorconfig 文件中：

```
dotnet_code_quality.ca1711.allowed_suffixes = Flag|Flags
```

相关规则

- CA1710:标识符应具有正确的后缀

请参阅

- 特性
- 处理和引发事件
- 命名准则:类、结构和接口

CA1712:不要将类型名用作枚举值的前缀

2021/11/16 •

	[
■ ID	CA1712
■	命名
▬▬▬	重大

原因

枚举包含名称以枚举的类型名称开头的成员。

规则说明

枚举成员的名称不使用类型名称作为前缀，因为类型信息将由开发工具提供。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了学习新软件库所需的时间，让客户更加相信该库是由拥有开发托管代码专业知识的人员所开发。

如何解决冲突

若要解决此规则的冲突，请从枚举成员中删除类型名称前缀。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例演示了一个错误命名的枚举，后跟更正后的版本。

```
public enum DigitalImageMode
{
    DigitalImageModeBitmap = 0,
    DigitalImageModeGrayscale = 1,
    DigitalImageModeIndexed = 2,
    DigitalImageModeRGB = 3
}

public enum DigitalImageMode2
{
    Bitmap = 0,
    Grayscale = 1,
    Indexed = 2,
    RGB = 3
}
```

```
Imports System

Namespace ca1712

    Enum DigitalImageMode

        DigitalImageModeBitmap = 0
        DigitalImageModeGrayscale = 1
        DigitalImageModeIndexed = 2
        DigitalImageModeRGB = 3

    End Enum

    Enum DigitalImageMode2

        Bitmap = 0
        Grayscale = 1
        Indexed = 2
        RGB = 3

    End Enum

End Namespace
```

相关规则

- CA1711:标识符应采用正确的后缀
- CA1027:用 FlagsAttribute 标记枚举
- CA2217:不要使用 FlagsAttribute 标记枚举

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 枚举值前缀触发器

你可以仅为此规则、为所有规则或为此类别(命名)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

枚举值前缀触发器

你可以配置触发规则所需的枚举值数量。例如，若要指定在一个或多个枚举值以枚举类型名称开头时触发规则，请将以下键值对添加到项目中的 editorconfig 文件：

```
dotnet_code_quality.CA1712.enum_values_prefix_trigger = AnyEnumValue
```

示例：

'''	''
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = AnyEnumValue`	如果任意枚举值以枚举类型名称开头，则会触发此规则。
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = AllEnumValues`	如果所有枚举值均以枚举类型名称开头，则会触发此规则。
`dotnet_code_quality.CA1712.enum_values_prefix_trigger = Heuristic`	使用默认启发式(即至少 75% 的枚举值以枚举类型名称开头)触发规则。

另请参阅

- System.Enum

CA1713:事件不应具有 before 或 after 前缀

2021/11/16 •

■ ID	CA1713
■	命名
▬▬	重大

原因

事件的名称以“Before”或“After”开头。

规则说明

事件名称应描述引发该事件的操作。若要命名按特定顺序引发的相关事件，请使用现在时或过去时指示一系列操作中的相对位置。例如，在对关闭资源时引发的一对事件进行命名时，可将其命名为“Closing”和“Closed”，而不是“BeforeClose”和“AfterClose”。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库的学习曲线，让客户更加相信该库是由拥有托管代码开发专业知识的人员开发的。

如何解决冲突

从事件名称中删除前缀，并考虑更改该名称，使用谓词的现在时或过去时。

何时禁止显示警告

不禁止显示此规则发出的警告。

CA1714:Flags 枚举应采用复数形式的名称

2021/11/16 •

	[
■ ID	CA1714
■	命名
▬▬▬	重大

原因

枚举具有 System.FlagsAttribute，并且其名称不是以“s”结尾。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

用 FlagsAttribute 标记的类型具有复数形式的名称，因为该特性指明可以指定多个值。例如，定义一周中各天的枚举可能适用于指定多天的应用程序。此枚举应该具有 FlagsAttribute，并且可称为“Days”。类似的枚举如果只允许指定一天，则不具有该属性，可以称为“Day”。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库的学习曲线，让客户更加相信该库是由拥有托管代码开发专业知识的人员开发的。

如何解决冲突

将枚举的名称设为复数，如果不应同时指定多个枚举值，请删除 FlagsAttribute 属性。

何时禁止显示警告

如果名称是复数形式，但不以“s”结尾，则可以安全地禁止显示冲突。例如，如果前面描述的多天枚举名为“DaysOfTheWeek”，虽然这不是它的本意，但也违反了规则的逻辑。应该禁止显示此类冲突。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1027:用 FlagsAttribute 标记枚举
- CA2217:不要使用 FlagsAttribute 标记枚举

另请参阅

- System.FlagsAttribute
- 枚举设计

CA1715:标识符应具有正确的前缀

2021/11/16 •

	[
■ ID	CA1715
■	命名
▬▬▬	中断 - 在接口上引发时。 非中断 - 在泛型类型参数上引发时。

原因

接口的名称未以大写的“I”开头。

-或-

类型或方法上的泛型类型参数的名称未以大写的“T”开头。

默认情况下，此规则仅查看外部可见的接口、类型和方法，但这是可配置的。

规则说明

按照约定，某些编程元素的名称以特定前缀开头。

接口名称应以大写的“I”开头，后跟另一个大写字母。此规则报告与接口名称（如“MyInterface”和“IsolatedInterface”）相关的冲突。

泛型类型参数名称应以大写的“T”开头，可选择后跟另一个大写字母。此规则报告与泛型类型参数名称（如“V”和“Type”）相关的冲突。

命名约定为面向公共语言运行时的库提供通用外观。这缩短了新软件库的学习曲线，让客户更加相信该库是由拥有托管代码开发专业知识的人员开发的。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 单字符类型参数

可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

单字符类型参数

可以配置是否从该规则中排除单字符类型参数。例如，若要指定此规则不得分析单字符类型参数，请将以下某一键值对添加到项目的 .editorconfig 文件中：

```
# Package version 2.9.0 and later
dotnet_code_quality.CA1715.exclude_single_letter_type_parameters = true

# Package version 2.6.3 and earlier
dotnet_code_quality.CA2007.allow_single_letter_type_parameters = true
```

> **NOTE**
> 对于名为 `T` 的类型参数（例如 `Collection<T>`），不会触发此规则。

如何解决冲突

重命名标识符，使其具有正确的前缀。

何时禁止显示警告

不禁止显示此规则发出的警告。

接口命名示例

以下代码片段显示了一个命名错误的接口：

```
' Violates this rule
Public Interface Book

    ReadOnly Property Title() As String

    Sub Read()

End Interface
```

```
// Violation.
public interface Book
{
    string Title
    {
        get;
    }

    void Read();
}
```

下面的代码片段通过使用“I”作为接口的前缀来解决之前的冲突：

```
// Fixes the violation by prefixing the interface with 'I'.
public interface IBook
{
    string Title
    {
        get;
    }

    void Read();
}
```

```
' Fixes the violation by prefixing the interface with 'I'
Public Interface IBook

    ReadOnly Property Title() As String

    Sub Read()

End Interface
```

类型参数命名示例

以下代码片段显示了命名错误的泛型类型参数：

```
' Violates this rule
Public Class Collection(Of Item)

End Class
```

```
// Violation.
public class Collection<Item>
{
}
```

下面的代码片段通过使用“T”作为泛型类型参数的前缀来解决之前的冲突：

```
// Fixes the violation by prefixing the generic type parameter with 'T'.
public class Collection<TItem>
{
}
```

```
' Fixes the violation by prefixing the generic type parameter with 'T'
Public Class Collection(Of TItem)

End Class
```

CA1716:标识符不应与关键字冲突

2021/11/16 •

■ ID	CA1716
■	命名
■■■■■	重大

原因

命名空间、类型、虚拟或接口成员的名称与编程语言中的保留关键字一致。

默认情况下，此规则仅查看外部可见的命名空间、类型和成员，但你可以配置可见性和符号类型。

规则说明

命名空间、类型以及虚拟和接口成员的标识符不应与面向公共语言运行时的语言所定义的关键字一致。根据所用的语言和关键字，编译器错误和歧义会使库难以使用。

此规则检查以下语言中的关键字：

- Visual Basic
- C#
- C++/CLI

不区分大小写的比较用于 Visual Basic 关键字，区分大小写的比较用于其他语言。

如何解决冲突

选择未显示在关键字列表中的名称。

何时禁止显示警告

如果确信标识符不会使 API 用户混淆，并且库可用于 .NET 中的所有可用语言，则可以禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 分析的符号类型

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

分析的符号类型

可配置此规则将分析的符号类型。允许的值为:

- `Namespace`
- `NamedType`
- `Method`
- `Property`
- `Event`
- `Parameter`

用逗号 `,` 分隔多个值。默认值包括前面列表中的所有符号类型。

```
dotnet_code_quality.CA1716.analyzed_symbol_kinds = Namespace, NamedType, Method, Property, Event
```

CA1717:只有 FlagsAttribute 枚举应采用复数形式的名称

2021/11/16 •

■ ID	CA1717
■	命名
■■■■■	重大

原因

枚举的名称以复数形式结尾，并且枚举未标记 System.FlagsAttribute 特性。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

命名约定规定，复数形式的枚举名称表示可以同时指定多个枚举值。FlagsAttribute 告诉编译器，应将枚举视为对枚举启用位运算的位字段。

如果一次只能指定一个枚举值，则枚举的名称应为单数形式。例如，定义星期的枚举可能适用于可指定多天的应用程序。此枚举应该具有 FlagsAttribute，并且可称为“Days”。类似的枚举如果只允许指定一天，则不具有该属性，可以称为“Day”。

命名约定为面向公共语言运行时的库提供常见外观。这缩短了学习新软件库所需的时间，让客户更加相信该库是由拥有开发托管代码专业知识的人员所开发。

如何解决冲突

将枚举名称设置为单数形式或添加 FlagsAttribute。

何时禁止显示警告

如果名称以单数形式结尾，可以禁止显示规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1714:Flags 枚举应采用复数形式的名称
- CA1027:用 FlagsAttribute 标记枚举
- CA2217:不要使用 FlagsAttribute 标记枚举

另请参阅

- System.FlagsAttribute
- 枚举设计

CA1720:标识符不应包含类型名称

2021/11/16 •

■ ID	CA1720
■	命名
▬▬▬	重大

原因

成员中的参数名称包含数据类型名称。

-或-

成员的名称包含语言特定的数据类型名称。

默认情况下，此规则仅查看外部可见的成员，但这可配置。

规则说明

参数和成员的名称更好地用于传达其含义而不是描述其类型，类型描述通常由开发工具提供。对于成员的名称，如果必须使用数据类型名称，请使用与语言无关的名称，而不要使用语言特定的名称。例如，请使用与语言无关的数据类型名称 `Int32`，而不要使用 C# 类型名称 `int`。

参数或成员名称中的每个离散标记都会对照以下语言特定的数据类型名称进行检查（不区分大小写）：

- Bool
- WChar
- Int8
- UInt8
- Short
- UShort
- int
- UInt
- Integer
- UInteger
- Long
- ULong
- 无符号
- 有符号
- Float
- Float32
- Float64

此外，参数的名称还会对照以下与语言无关的数据类型名称进行检查（不区分大小写）：

- 对象
- 布尔
- Char
- 字符串
- SByte
- Byte
- UByte
- Int16
- UInt16
- Int32
- UInt32
- Int64
- UInt64
- IntPtr
- Ptr
- 指针
- UInptr
- UPtr
- UPointer
- Single
- Double
- 小数
- GUID

如何解决冲突

如果针对**参数触**发：

将参数名称中的数据类型标识符替换为一个可更好地描述其含义的词或更通用的词，如“value”。

如果针对**成**员**触**发：

将成员名称中的语言特定数据类型标识符替换为一个可更好地描述其含义的词、与语言无关的等效词或更通用的词，如“value”。

何时禁止显示警告

如果偶尔使用基于类型的参数和成员名称，则可禁止显示警告。但对于新开发，没有任何已知情况应该禁止显示此规则的警告。对于以前发布的库，可能有必要禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1708:标识符应以大小写之外的差别进行区分
- CA1707:标识符不应包含下划线

CA1721:属性名不应与 get 方法冲突

2021/11/16 •

■ ID	CA1721
■	命名
■■■■■	重大

原因

成员的名称以“Get”开头，且其余部分与属性的名称匹配。例如，包含名为“GetColor”的方法和名为“Color”的属性的类型将导致规则冲突。如果使用 ObsoleteAttribute 对属性或方法进行标记，则不会触发此规则。

默认情况下，此规则仅查看外部可见的成员和属性，但这是可配置的。

规则说明

“Get”方法和属性的名称应能够明确区分其功能上的差异。

命名约定为面向公共语言运行时的库提供通用外观。此一致性缩短了学习新软件库所需的时间，让客户更加相信该库是由拥有开发托管代码专业知识的人员所开发。

如何解决冲突

更改名称，使其与前缀为“Get”的方法名称不匹配。

何时禁止显示警告

不禁止显示此规则发出的警告。

> **NOTE**
> 如果“Get”方法是由实现 IExtenderProvider 接口所引起，则可排除此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例包含与此规则冲突的方法和属性。

```
public class Test
{
    public DateTime Date
    {
        get { return DateTime.Today; }
    }

    // Violates rule: PropertyNamesShouldNotMatchGetMethods.
    public string GetDate()
    {
        return this.Date.ToString();
    }
}
```

```
Imports System

Namespace ca1721

    Public Class Test

        Public ReadOnly Property [Date]() As DateTime
            Get
                Return DateTime.Today
            End Get
        End Property

        ' Violates rule: PropertyNamesShouldNotMatchGetMethods.
        Public Function GetDate() As String
            Return Me.Date.ToString()
        End Function

    End Class

End Namespace
```

相关规则

- CA1024:在适用处使用属性

CA1724：类型名不应与命名空间冲突

2021/11/16 •

■ ID	CA1724
■	命名
▬▬▬	重大

原因

类型名与具有一个或多个外部可见类型的被引用命名空间名称冲突。名称比较不区分大小写。

规则说明

用户创建的类型名不应与具有外部可见类型的被引用命名空间的名称冲突。与该规则冲突将使库的可用性下降。

如何解决冲突

重命名该类型，使其与具有外部可见类型的被引用命名空间的名称不冲突。

何时禁止显示警告

对于新开发，没有任何已知情况必须禁止显示此规则的警告。在禁止显示该警告之前，请仔细考虑库的用户可能会因冲突名称感到困惑。对于发布库，可能必须禁止显示此规则发出的警告。

示例

```
namespace MyNamespace
{
    // This class violates the rule
    public class System
    {
    }
}
```

CA1725:参数名应与基方法中的声明保持一致

2021/11/16 •

	[
■ ID	CA1725
■	命名
▬▬▬	重大

原因

某方法替代中的参数名与该方法的基声明中的参数名或该方法的接口声明中的参数名不一致。

默认情况下，此规则仅查看外部可见的方法，但这是可配置的。

规则说明

以一致的方式命名重写层次结构中的参数可以提高方法重写的可用性。如果派生方法中的参数名与基声明中的名称不同，可能会导致无法区分出该方法是基方法的重写还是该方法的新重载。

如何解决冲突

若要解决此规则的冲突，请重命名参数以与基声明保持一致。此修复是 COM 可见方法的一项中断性变更。

何时禁止显示警告

请勿禁止显示此规则的警告，但之前已发布库中的 COM 可见方法除外。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（命名）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

性能规则

2021/11/16 •

性能规则支持高性能库和应用程序。

在本节中

"	"
CA1802:在合适的位置使用文本	某个字段被声明为 static 和 read-only(在 Visual Basic 中为 Shared 和 ReadOnly)，并使用可在编译时计算的值初始化。因为赋给目标字段的值可在编译时计算，因此请将声明更改为 const(在 Visual Basic 中为 Const)字段，以便在编译时而非运行时计算值。
CA1805：避免进行不必要的初始化	在运行构造函数之前，.NET 运行时将引用类型的所有字段初始化为其默认值。在大多数情况下，将字段显式初始化为其默认值是多余的，这会增加维护成本，并可能会降低性能(例如随着程序集大小的增加)。
CA1806:不要忽略方法结果	创建一个新对象，但从不使用该对象；或者调用会创建并返回一个新字符串的方法，但从不使用这个新字符串；或者组件对象模型 (COM) 或 P/Invoke 方法返回一个从不使用的 HRESULT 或错误代码。
CA1810:以内联方式初始化引用类型的静态字段	当一个类型声明显式静态构造函数时，实时 (JIT) 编译器会向该类型的每个静态方法和实例构造函数中添加一项检查，以确保之前已调用该静态构造函数。静态构造函数检查会降低性能。
CA1812:避免未实例化的内部类	程序集级别类型的实例不是由程序集中的代码创建的。
CA1813:避免使用非密封特性	.NET 提供用于检索自定义属性的方法。默认情况下，这些方法搜索特性继承层次结构。通过密封特性，将无需搜索继承层次结构，且能够提高性能。
CA1814:与多维数组相比，首选使用交错数组	交错数组是元素为数组的数组。构成元素的数组可采用不同的大小，使某些数据集浪费的空间减少。
CA1815:重写值类型上的 Equals 和相等运算符	对于值类型，Equals 的继承的实现使用反射库，并比较所有字段的内容。反射需要消耗大量计算资源，可能没有必要比较每一个字段是否相等。如果希望用户对实例进行比较或排序，或者希望用户将实例用作哈希表键，则值类型应实现 Equals。
CA1819:属性不应返回数组	即使属性是只读的，该属性返回的数组也不受写入保护。若要使数组不会被更改，属性必须返回数组的副本。通常，用户不能理解调用这种属性的负面性能影响。
CA1820:使用字符串长度测试是否有空字符串	使用 String.Length 属性或 String.IsNullOrEmpty 方法比较字符串要比使用 Equals 的速度快得多。
CA1821:移除空终结器	应尽可能避免终结器，因为跟踪对象生存期会产生额外的性能系统开销。空的终结器只会徒增开销，没有一点好处。

CA1822:将成员标记为 static	可以将不访问实例数据或不调用实例方法的成员标记为 static(在 Visual Basic 中为 Shared)。在将这些方法标记为 static 之后，编译器将向这些成员发出非虚拟调用站点。这会使性能敏感的代码的性能得到显著提高。
CA1823:避免未使用的私有字段	检测到程序集内有似乎未访问过的私有字段。
CA1824:用 NeutralResourcesLanguageAttribute 标记程序集	NeutralResourcesLanguage 属性通知资源管理器用于显示程序集的非特定区域性资源的语言。这将改进所加载的第一个资源的查找性能，并缩小工作集。
CA1825:避免数组分配长度为零	初始化长度为零的数组将导致不必要的内存分配。相反，请通过调用 Array.Empty 来使用静态分配的空数组实例。内存分配在此方法的所有调用之间共享。
CA1826:使用属性，而不是 Linq Enumerable 方法	对支持等效且更有效的属性的类型使用了 Enumerable LINQ 方法。
CA1827:如果可以使用 Any，请勿使用 Count/LongCount	在使用 Any 方法会更有效的情况下使用了 Count 或 LongCount 方法。
CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync	在使用 AnyAsync 方法会更有效的情况下使用了 CountAsync 或 LongCountAsync 方法。
CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法	对支持等效且更有效的 `Length` 或 `Count` 属性的类型使用了 Count LINQ 方法。
CA1830:在 StringBuilder 上优先使用强类型“追加和插入”方法重载	Append 和 Insert 为除 System.String 之外的多种类型提供重载。如果可能，首选强类型重载，而非 ToString () 和基于字符串的重载。
CA1831:在合适的情况下，为字符串使用 AsSpan 而不是基于范围的索引器	对字符串使用范围索引器并向 ReadOnlySpan<char> 类型隐式赋值时，将使用方法 Substring 而非 Slice，这会生成字符串请求部分的副本。
CA1832:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 ReadOnlySpan 或 ReadOnlyMemory 部分	对字符串使用范围索引器并向 ReadOnlySpan<T> 或 ReadOnlyMemory<T> 类型隐式赋值时，将使用方法 GetSubArray 而非 Slice，这会生成数组请求部分的副本。
CA1833:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 Span 或 Memory 部分	对字符串使用范围索引器并向 Span<T> 或 Memory<T> 类型隐式赋值时，将使用方法 GetSubArray 而非 Slice，这会生成数组请求部分的副本。
CA1834:对单字符字符串使用 StringBuilder.Append(char)	StringBuilder 具有将 `char` 用作其参数的 `Append` 重载。优先选择调用 `char` 重载以提高性能。
CA1835：对于“ReadAsync”和“WriteAsync”，首选基于“Memory”的重载	“Stream”有一个将“Memory<byte>”用作第一个参数的“ReadAsync”重载和一个将“ReadOnlyMemory<Byte>”用作第一个参数的“WriteAsync”重载。优先选择调用基于内存的重载，它们更有效。

"	"
CA1836：如可用，首选 `IsEmpty` 而不是 `Count`	首选比 `Count`、`Length`、Count<TSource> (IEnumerable<TSource>) 或 LongCount<TSource> (IEnumerable<TSource>) 更有效的 `IsEmpty` 属性，以确定对象是否包含任何项目。
CA1837：使用 `Environment.ProcessId` 而不是 `Process.GetCurrentProcess().Id`	`Environment.ProcessId` 比 `Process.GetCurrentProcess().Id` 更简单、更快速。
CA1838：避免对 P/Invokes 使用 `StringBuilder` 参数	`StringBuilder` 的封送处理总是会创建一个本机缓冲区副本，这会导致一个封送处理操作出现多次分配。
CA1841：首选字典包含方法	对 `Keys` 或 `Values` 集合调用 `Contains` 通常比对字典本身调用 `ContainsKey` 或 `ContainsValue` 开销更高。
CA1844：对“流”进行子分类时，提供异步方法的基于内存的重写	若要提高性能，请在对“流”进行子分类时重写基于内存的异步方法。然后，在基于内存的方法中实现基于数组的方法。
CA1845：使用基于跨度的“string.Concat”	使用 `AsSpan` 和 `string.Concat` 比使用 `Substring` 和串联运算符更高效。
CA1846：首选 `AsSpan`，次选 `Substring`	`AsSpan` 比 `Substring` 更高效。`Substring` 执行 O(n) 字符串复制，而 `AsSpan` 不会执行此操作且具有固定成本。`AsSpan` 也不执行任何堆分配。
CA1847：对单个字符查找使用 char 文本	搜索单个字符时使用 `string.Contains(char)` 而不是 `string.Contains(string)`。
CA1849：当在异步方法中时，调用异步方法	搜索单个字符时使用 `string.Contains(char)` 而不是 `string.Contains(string)`。
CA1850：首选静态 `HashData` 方法，而非 `ComputeHash`	相比创建并管理 `HashAlgorithm` 实例来调用 `ComputeHash`，使用静态 `HashData` 方法更高效。

CA1802:在合适的位置使用文本

2021/11/16 •

	[
■ ID	CA1802
■	"性能"
▬▬▬	非中断

原因

某个字段被声明为 `static` 和 `readonly`（在 Visual Basic 中为 `Shared` 和 `ReadOnly`），并使用可在编译时计算的值初始化。

默认情况下，此规则仅查看外部可见的静态只读字段，但这是可配置的。

规则说明

当调用声明类型的静态构造函数时，将在运行时计算 `static readonly` 字段的值。如果 `static readonly` 字段在声明时被初始化并且静态构造函数不是显式声明的，编译器将发出一个静态构造函数来初始化该字段。

`const` 字段的值是在编译时计算的，并存储在元数据中，这与 `static readonly` 字段相比，运行时性能提高了。

因为赋给目标字段的值可在编译时计算，所以，请将声明更改为 `const` 字段，以便在编译时（而非运行时）计算该值。

如何解决冲突

若要解决此规则的冲突，请将 `static` 和 `readonly` 修饰符替换为 `const` 修饰符。

> **NOTE**
> 不建议对所有方案使用 const 修饰符。

何时禁止显示警告

如果性能无关紧要，则可安全地禁止显示此规则发出的警告，或禁用此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面
- 必需的修饰符

可以仅为此规则、为所有规则或为此类别（性能）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

必需的修饰符

可以将此规则配置为重写必需的字段修饰符。默认情况下，`static` 和 `readonly` 都是所分析字段的必需修饰符。可以将其重写为以逗号分隔的包含下表中一个或多个修饰符值的列表：

“”	“”
`none`	无修饰符要求。
`static` 或 `Shared`	必须声明为“static”（在 Visual Basic 中为“Shared”）。
`const`	必须声明为“const”。
`readonly`	必须声明为“readonly”。

例如，若要指定规则应针对静态或实例字段运行，请将以下键值对添加到项目的 .editorconfig 文件中：

```
dotnet_code_quality.CA1802.required_modifiers = none
```

示例

下面的示例显示了一个与此规则冲突的 `UseReadOnly` 类型，以及一个符合此规则的 `UseConstant` 类型。

```
Imports System

Namespace ca1802

    ' This class violates the rule.
    Public Class UseReadOnly

        Shared ReadOnly x As Integer = 3
        Shared ReadOnly y As Double = x + 2.1
        Shared ReadOnly s As String = "readonly"

    End Class

    ' This class satisfies the rule.
    Public Class UseConstant

        Const x As Integer = 3
        Const y As Double = x + 2.1
        Const s As String = "const"

    End Class

End Namespace
```

```
// This class violates the rule.
public class UseReadOnly
{
    static readonly int x = 3;
    static readonly double y = x + 2.1;
    static readonly string s = "readonly";
}

// This class satisfies the rule.
public class UseConstant
{
    const int x = 3;
    const double y = x + 2.1;
    const string s = "const";
}
```

CA1805：避免进行不必要的初始化。

2021/11/16 •

	▮
■ ID	CA1805
■	"性能"
▬▬▬	非中断

原因

类的字段已显式初始化为该字段类型的默认值。

规则说明

在运行构造函数之前，.NET 运行时将引用类型的所有字段初始化为其默认值。在大多数情况下，在构造函数中将字段显式初始化为其默认值是多余的，这会增加维护成本，并可能会降低性能（例如，随着程序集大小的增加），可以删除显式初始化。

如何解决冲突

在大多数情况下，正确的解决方法是删除不必要的初始化。

```
class C
{
    // Violation
    int _value1 = 0;

    // Fixed
    int _value1;
}
```

在某些情况下，由于字段会永久保留其默认值，因此删除初始化可能会导致发出后续的 CS0649 警告。在这种情况下，更好的解决方法是完全删除该字段，或将其替换为属性：

```
class C
{
    // Violation
    private static readonly int s_value = 0;

    // Fixed
    private static int Value => 0;
}
```

何时禁止显示警告

禁止显示警告始终是安全的，因为警告只是突出显示了可能不必要的代码以及可以避免的工作。

另请参阅

- 性能规则

CA1806:不要忽略方法结果

2021/11/16 •

■ ID	CA1806
■	使用情况
▬▬▬	非中断

原因

出现此警告有几个可能的原因：

- 创建了一个新的对象，但从未使用过它。
- 调用了一个创建并返回新字符串的方法，但从未使用过这个新字符串。
- 从未使用过的 COM 或 P/Invoke 方法，它返回 `HRESULT` 或错误代码。
- 从未使用过的语言集成查询 (LINQ) 方法，该方法返回结果。

规则说明

不必要的对象创建和未使用对象的关联垃圾回收会降低性能。

字符串是不可变的，并且 String.ToUpper 等方法返回字符串的新实例，而不是在调用方法中修改字符串的实例。

忽略 `HRESULT` 或错误代码可能导致在错误情况下或资源不足的情况下发生异常行为。

已知 LINQ 方法不具有副作用，因此不应忽略其结果。

如何解决冲突

如果方法 A 创建从未使用的 B 对象的新实例，请将该实例作为参数传递给另一个方法，或将该实例分配给一个变量。如果不需要创建对象，则将其删除。

-或-

如果方法 A 调用方法 B，但不使用方法 B 返回的新字符串实例，请将此实例作为参数传递给另一个方法，或将此实例分配给一个变量。如果不需要该调用，可以将其删除。

-或-

如果方法 A 调用方法 B，但不使用 `HRESULT` 或方法返回的错误代码，请在条件语句中使用该结果、将该结果分配给一个变量，或将它作为参数传递给另一个方法。

-或-

如果 LINQ 方法 A 调用方法 B，但不使用结果，请在条件语句中使用该结果、将该结果分配给一个变量，或将它作为参数传递给另一个方法。

何时禁止显示警告

请勿禁止显示此规则发出的警告，除非创建对象的行为可用于实现某些目的。

示例 1

下面的示例演示一个类，该类忽略调用 String.Trim 的结果。

```
public class Book
{
    private readonly string _Title;

    public Book(string title)
    {
        if (title != null)
        {
            // Violates this rule
            title.Trim();
        }

        _Title = title;
    }

    public string Title
    {
        get { return _Title; }
    }
}
```

```
Public Class Book
    Public Sub New(ByVal title As String)

        If title IsNot Nothing Then
            ' Violates this rule
            title.Trim()
        End If

        Me.Title = title

    End Sub

    Public ReadOnly Property Title() As String

End Class
```

示例 2

下面的示例通过将 String.Trim 的结果分配回在其上调用的变量来修复之前的冲突。

```
public class Book
{
    private readonly string _Title;

    public Book(string title)
    {
        if (title != null)
        {
            title = title.Trim();
        }

        _Title = title;
    }

    public string Title
    {
        get { return _Title; }
    }
}
```

```
Public Class Book
    Public Sub New(ByVal title As String)

        If title IsNot Nothing Then
            title = title.Trim()
        End If

        Me.Title = title

    End Sub

    Public ReadOnly Property Title() As String

End Class
```

示例 3

下面的示例演示了一个方法，该方法不使用它创建的对象。

> **NOTE**
> Visual Basic 中无法重现此冲突。

```
public class Book
{
    public Book()
    {
    }

    public static Book CreateBook()
    {
        // Violates this rule
        new Book();
        return new Book();
    }
}
```

示例 4

下面的示例通过删除不必要的对象创建来修复之前的冲突。

```
public class Book
{
    public Book()
    {
    }

    public static Book CreateBook()
    {
        return new Book();
    }
}
```

CA1810:以内联方式初始化引用类型的静态字段

2021/11/16 •

	▮
■ ID	CA1810
■	"性能"
▬▬	非中断

原因

引用类型声明显式静态构造函数。

规则说明

当一个类型声明显式静态构造函数时，实时 (JIT) 编译器会向该类型的每个静态方法和实例构造函数中添加一项检查，以确保之前已调用该静态构造函数。访问任何静态成员或创建该类型的实例时，将触发静态初始化。但是，如果声明一个类型的变量，但不使用它，则不会触发静态初始化；这在初始化会更改全局状态的情况下非常重要。

当所有静态数据都以内联方式初始化并且未声明显式静态构造函数时，Microsoft 中间语言 (MSIL) 编译器会将 `beforefieldinit` 标志和隐式静态构造函数（该构造函数初始化静态数据）添加到 MSIL 类型定义。JIT 编译器遇到 `beforefieldinit` 标志时，大多数情况下不会添加静态构造函数检查。静态初始化可以保证在访问任何静态字段之前的某个时间发生，但不能在调用静态方法或实例构造函数之前发生。请注意，在声明类型的变量后，可能会随时发生静态初始化。

静态构造函数检查会降低性能。通常，静态构造函数仅用于初始化静态字段，在这种情况下，必须确保仅在首次访问静态字段之前发生静态初始化。`beforefieldinit` 行为适用于这些类型和大多数其他类型。仅当静态初始化影响全局状态并且满足以下任一条件时，它才是不适当的：

- 影响全局状态的成本非常昂贵，如果不使用该类型，则不需要这样做。
- 可以在不访问该类型的任何静态字段的情况下访问全局状态效果。

如何解决冲突

要修复与该规则的冲突，请在声明它时初始化所有静态数据并移除静态构造函数。

何时禁止显示警告

如果不考虑性能，或者，如果静态初始化导致的全局状态更改成本非常昂贵，或者必须保证在调用该类型的静态方法或创建该类型的实例之前进行静态初始化，则可以安全地禁止显示此规则发出的警告。

示例

下面的示例演示了类型 `StaticConstructor`（该类型违反了规则）以及类型 `NoStaticConstructor`（该类型使用内联初始化替换静态构造函数来满足规则）。

```
public class StaticConstructor
{
    static int someInteger;
    static string resourceString;

    static StaticConstructor()
    {
        someInteger = 3;
        ResourceManager stringManager =
           new ResourceManager("strings", Assembly.GetExecutingAssembly());
        resourceString = stringManager.GetString("string");
    }
}

public class NoStaticConstructor
{
    static int someInteger = 3;
    static string resourceString = InitializeResourceString();

    static string InitializeResourceString()
    {
        ResourceManager stringManager =
           new ResourceManager("strings", Assembly.GetExecutingAssembly());
        return stringManager.GetString("string");
    }
}
```

```
Imports System
Imports System.Resources

Namespace ca1810

    Public Class StaticConstructor

        Shared someInteger As Integer
        Shared resourceString As String

        Shared Sub New()

            someInteger = 3
            Dim stringManager As New ResourceManager("strings",
            System.Reflection.Assembly.GetExecutingAssembly())
            resourceString = stringManager.GetString("string")

        End Sub

    End Class


    Public Class NoStaticConstructor

        Shared someInteger As Integer = 3
        Shared resourceString As String = InitializeResourceString()

        Private Shared Function InitializeResourceString()

            Dim stringManager As New ResourceManager("strings",
            System.Reflection.Assembly.GetExecutingAssembly())
            Return stringManager.GetString("string")

        End Function

    End Class

End Namespace
```

请注意在 `NoStaticConstructor` 类的 MSIL 定义上添加的 `beforefieldinit` 标志。

```
.class public auto ansi StaticConstructor
extends [mscorlib]System.Object
{
} // end of class StaticConstructor

.class public auto ansi beforefieldinit NoStaticConstructor
extends [mscorlib]System.Object
{
} // end of class NoStaticConstructor
```

相关规则

- CA2207:以内联方式初始化值类型的静态字段

CA1812:避免未实例化的内部类

2021/11/16 •

	▮
■ ID	CA1812
■	"性能"
▬▬▬	非中断

原因

永远不会实例化内部(程序集级别)类型。

规则说明

此规则尝试查找对该类型其中一个构造函数的调用，并在找不到调用时报告冲突。

此规则不会检查以下类型：

- 值类型
- 抽象类型
- 枚举
- 委托
- 编译器发出的数组类型
- 无法实例化且仅定义 `static`(在 Visual Basic 中为 `Shared`)方法的类型。

如果将 System.Runtime.CompilerServices.InternalsVisibleToAttribute 应用于正在分析的程序集，那么此规则不会标记标记为 `internal`(在 Visual Basic 中为 `Friend`)的类型，因为友元程序集可能会使用字段。

如何解决冲突

若要解决此规则的冲突，请删除类型或添加使用该类型的代码。如果类型仅包含 `static` 方法，请将以下内容其中之一添加到类型，以阻止编译器生成默认的公共实例构造函数：

- 适用于 C# 类型的 `static` 修饰符面向 .NET Framework 2.0 或更高版本。
- 面向 .NET Framework 版本 1.0 和 1.1 的类型的专用构造函数。

何时禁止显示警告

禁止显示此规则的警告是安全的。建议在以下情况时取消显示此警告：

- 类通过后期绑定反射方法(如 System.Activator.CreateInstance)创建。
- 类由运行时或 ASP.NET 自动创建。自动创建的类的示例包括实现 System.Configuration.IConfigurationSectionHandler 或 System.Web.IHttpHandler 的类。

- 类作为具有 `new` 约束的类型参数进行传递。以下示例将由规则 CA1812 进行标记：

```
internal class MyClass
{
    public void DoSomething()
    {
    }
}
public class MyGeneric<T> where T : new()
{
    public T Create()
    {
        return new T();
    }
}

MyGeneric<MyClass> mc = new MyGeneric<MyClass>();
mc.Create();
```

相关规则

- CA1801:检查未使用的参数

CA1813:避免使用非密封特性

2021/11/16 •

	▮
■ ID	CA1813
■	"性能"
▬▬	重大

原因

继承自 System.Attribute 的公共类型不是抽象类型，也不会密封(Visual Basic 中的 `NotInheritable`)。

规则说明

.NET 提供用于检索自定义特性的方法。默认情况下，这些方法搜索特性继承层次结构。例如，System.Attribute.GetCustomAttribute 搜索指定的特性类型或扩展指定特性类型的所有特性类型。密封特性后，无需通过继承层次结构进行搜索，且能够提高性能。

如何解决冲突

若要解决此规则的冲突，请密封特性类型或使其成为抽象类型。

何时禁止显示警告

可安全地禁止显示此规则的警告。仅当你正在定义特性层次结构，并且不能密封特性或使其成为抽象特性时才禁止显示。

示例

下面的示例显示了一个符合此规则的自定义特性。

```
// Satisfies rule: AvoidUnsealedAttributes.
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public sealed class DeveloperAttribute : Attribute
{
    private string nameValue;
    public DeveloperAttribute(string name)
    {
        nameValue = name;
    }

    public string Name
    {
        get
        {
            return nameValue;
        }
    }
}
```

```
Imports System

Namespace ca1813

    ' Satisfies rule: AvoidUnsealedAttributes.
    <AttributeUsage(AttributeTargets.Class Or AttributeTargets.Struct)>
    Public NotInheritable Class DeveloperAttribute
        Inherits Attribute

        Public Sub New(name As String)
            Me.Name = name
        End Sub


        Public ReadOnly Property Name() As String
    End Class

End Namespace
```

相关规则

- CA1019:定义特性参数的访问器
- CA1018:用 AttributeUsageAttribute 标记特性

请参阅

- 特性

CA1814:与多维数组相比，首选使用交错数组

2021/11/16 •

	[
■ ID	CA1814
■	"性能"
▬▬▬	重大

原因

将成员声明为多维数组，这可能会浪费某些数据集的空间。

规则说明

在多维数组中，每个维度中的每个元素都具有与该维度中的其他元素相同的固定大小。在交错数组（即多个数组构成的数组）中，每个内部数组的大小可以不同。通过仅使用给定数组所需的空间，就不会浪费空间。此 CA1814 规则建议切换到交错数组以节省内存。

如何解决冲突

若要解决此规则的冲突，请将多维数组更改为交错数组。

何时禁止显示警告

如果多维数组不会浪费空间，则可以禁止显示此规则的警告。

示例

以下示例显示了交错数组和多维数组的声明。

```
Imports System

Public Class ArrayHolder
    Private jaggedArray As Integer()() =  {New Integer() {1, 2, 3, 4}, _
                                           New Integer() {5, 6, 7}, _
                                           New Integer() {8}, _
                                           New Integer() {9}}

    Private multiDimArray As Integer(,) =  {{1, 2, 3, 4}, _
                                            {5, 6, 7, 0}, _
                                            {8, 0, 0, 0}, _
                                            {9, 0, 0, 0}}
End Class
```

```
public class ArrayHolder
{
    int[][] jaggedArray = { new int[] {1,2,3,4},
                            new int[] {5,6,7},
                            new int[] {8},
                            new int[] {9}
                          };

    int[,] multiDimArray = {{1,2,3,4},
                             {5,6,7,0},
                             {8,0,0,0},
                             {9,0,0,0}
                            };
}
```

CA1815:重写值类型上的 Equals 和相等运算符

2021/11/16 •

■ ID	CA1815
■	"性能"
▬▬▬	非中断

原因

值类型未重写 System.Object.Equals 或未实现相等运算符 (==)。此规则不检查枚举。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

对于非 blittable 值类型，Equals 的继承实现使用 System.Reflection 库来比较所有字段的内容。反射需要消耗大量计算资源，可能没有必要比较每一个字段是否相等。如果希望用户对实例进行比较或排序，或者希望用户将它们用作哈希表键，则值类型应实现 Equals。如果编程语言支持运算符重载，则还应提供相等和不等运算符的实现。

如何解决冲突

若要解决此规则的冲突，请提供 Equals 的实现。如果可以，请实现相等运算符。

何时禁止显示警告

如果不会将值类型的实例进行相互比较，可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别(性能)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

以下代码显示了违反此规则的结构(值类型)：

```
// Violates this rule
public struct Point
{
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }

    public int Y { get; }
}
```

以下代码通过重写 System.ValueType.Equals 并实现相等运算符（`==` 和 `!=`）来解决前面的冲突：

```
public struct Point : IEquatable<Point>
{
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }

    public int Y { get; }

    public override int GetHashCode()
    {
        return X ^ Y;
    }

    public override bool Equals(object obj)
    {
        if (!(obj is Point))
            return false;

        return Equals((Point)obj);
    }

    public bool Equals(Point other)
    {
        if (X != other.X)
            return false;

        return Y == other.Y;
    }

    public static bool operator ==(Point point1, Point point2)
    {
        return point1.Equals(point2);
    }

    public static bool operator !=(Point point1, Point point2)
    {
        return !point1.Equals(point2);
    }
}
```

相关规则

- CA2231:重写 ValueType.Equals 时应重载相等运算符

- CA2226:运算符应有对称重载

另请参阅

- System.Object.Equals

CA1819:属性不应返回数组

2021/11/16 •

	[
■ ID	CA1819
■	“性能”
▬▬▬	重大

原因

属性返回数组。

默认情况下，此规则仅查看外部可见的属性和类型，但这是可配置的。

规则说明

即使属性是只读的，该属性返回的数组也不受写入保护。若要使数组不会被更改，属性必须返回数组的副本。通常，用户不能理解调用这种属性的负面性能影响。具体来说，他们可能将索引属性作为属性使用。

如何解决冲突

要解决此规则的冲突，请将属性设置为方法或更改属性以返回集合。

何时禁止显示警告

可禁止显示从 Attribute 类派生的特性中由属性引发的警告。特性可以包含返回数组的属性，但不能包含返回集合的属性。

如果属性是数据传输对象 (DTO) 类的一部分，则可以禁止显示警告.

否则，请勿禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别（性能）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例冲突

下面的示例显示了与此规则发生冲突的属性：

```
public class Book
{
    private string[] _Pages;

    public Book(string[] pages)
    {
        _Pages = pages;
    }

    public string[] Pages
    {
        get { return _Pages; }
    }
}
```

```
Public Class Book
    Public Sub New(ByVal pages As String())
        Me.Pages = pages
    End Sub

    Public ReadOnly Property Pages() As String()
End Class
```

若要解决此规则的冲突，请将属性设置为方法或更改属性以返回集合而不是数组。

将属性更改为方法

以下示例通过将属性更改为方法来解决冲突：

```
Public Class Book

    Private _Pages As String()

    Public Sub New(ByVal pages As String())
        _Pages = pages
    End Sub

    Public Function GetPages() As String()
        ' Need to return a clone of the array so that consumers
        ' of this library cannot change its contents
        Return DirectCast(_Pages.Clone(), String())
    End Function

End Class
```

```
public class Book
{
    private string[] _Pages;

    public Book(string[] pages)
    {
        _Pages = pages;
    }

    public string[] GetPages()
    {
        // Need to return a clone of the array so that consumers
        // of this library cannot change its contents
        return (string[])_Pages.Clone();
    }
}
```

更改属性以返回集合

以下示例通过更改属性以返回 System.Collections.ObjectModel.ReadOnlyCollection<T> 来解决冲突：

```
public class Book
{
    private ReadOnlyCollection<string> _Pages;
    public Book(string[] pages)
    {
        _Pages = new ReadOnlyCollection<string>(pages);
    }

    public ReadOnlyCollection<string> Pages
    {
        get { return _Pages; }
    }
}
```

```
Public Class Book
    Public Sub New(ByVal pages As String())
        Me.Pages = New ReadOnlyCollection(Of String)(pages)
    End Sub

    Public ReadOnly Property Pages() As ReadOnlyCollection(Of String)

End Class
```

允许用户修改属性

你可能希望允许类的使用者修改属性。以下示例显示与此规则冲突的读/写属性：

```
public class Book
{
    private string[] _Pages;

    public Book(string[] pages)
    {
        _Pages = pages;
    }

    public string[] Pages
    {
        get { return _Pages; }
        set { _Pages = value; }
    }
}
```

```
Public Class Book
    Public Sub New(ByVal pages As String())
        Me.Pages = pages
    End Sub

    Public Property Pages() As String()

End Class
```

以下示例通过更改属性以返回 System.Collections.ObjectModel.Collection<T> 来解决冲突：

```
Public Class Book
    Public Sub New(ByVal pages As String())
        Me.Pages = New Collection(Of String)(pages)
    End Sub

    Public ReadOnly Property Pages() As Collection(Of String)
End Class
```

```
public class Book
{
    private Collection<string> _Pages;

    public Book(string[] pages)
    {
        _Pages = new Collection<string>(pages);
    }

    public Collection<string> Pages
    {
        get { return _Pages; }
    }
}
```

相关规则

- CA1024:在适用处使用属性

CA1820:使用字符串长度测试是否有空字符串

2021/11/16 •

	[
■ ID	CA1820
■	"性能"
▬▬▬	非中断

原因

使用了 Object.Equals 将字符串与空字符串进行比较。

规则说明

使用 String.Length 属性或 String.IsNullOrEmpty 方法比较字符串比使用 Equals 更快。这是因为 Equals 执行的 MSIL 指令比 IsNullOrEmpty 或执行以用于检索 Length 属性值并将其与零进行比较的指令数要多得多。

对于 NULL 字符串，Equals 和 `<string>.Length == 0` 的行为不同。如果尝试获取 NULL 字符串的 Length 属性值，则公共语言运行时将引发 System.NullReferenceException。如果在 NULL 字符串和空字符串之间执行比较，则公共语言运行时不会引发异常，并将返回 `false`。测试 NULL 不会对这两种方法的相对性能产生显著影响。面向 .NET Framework 2.0 或更高版本时，请使用 IsNullOrEmpty 方法。否则，请尽可能使用 Length == 0 比较。

如何解决冲突

若要解决此规则的冲突，请更改比较以使用 IsNullOrEmpty 方法。

何时禁止显示警告

如果性能不是问题，可禁止显示此规则的警告。

示例

下面的示例演示了用于查找空字符串的不同技术。

```
public class StringTester
{
    string s1 = "test";

    public void EqualsTest()
    {
        // Violates rule: TestForEmptyStringsUsingStringLength.
        if (s1 == "")
        {
            Console.WriteLine("s1 equals empty string.");
        }
    }

    // Use for .NET Framework 1.0 and 1.1.
    public void LengthTest()
    {
        // Satisfies rule: TestForEmptyStringsUsingStringLength.
        if (s1 != null && s1.Length == 0)
        {
            Console.WriteLine("s1.Length == 0.");
        }
    }

    // Use for .NET Framework 2.0.
    public void NullOrEmptyTest()
    {
        // Satisfies rule: TestForEmptyStringsUsingStringLength.
        if (!String.IsNullOrEmpty(s1))
        {
            Console.WriteLine("s1 != null and s1.Length != 0.");
        }
    }
}
```

CA1821:移除空终结器

2021/11/16 •

	[
■ ID	CA1821
■	"性能"
▬▬▬	非中断

原因

类型实现了一个空的终结器，只调用基类型终结器或只调用条件性发出的方法。

规则说明

应尽可能避免终结器，因为跟踪对象生存期会产生额外的性能系统开销。垃圾回收器在收集对象之前运行终结器。这意味着收集对象至少需要两个集合。空的终结器只会徒增开销，没有一点好处。

如何解决冲突

移除空的终结器。如果调试需要终结器，请将整个终结器置于 `#if DEBUG / #endif` 指令中。

何时禁止显示警告

不禁止显示此规则发出的消息。

示例

下面的示例演示了应移除的空终结器、应置于 `#if DEBUG / #endif` 指令中的终结器以及正确使用 `#if DEBUG / #endif` 指令的终结器。

```
    public class Class1
    {
        // Violation occurs because the finalizer is empty.
        ~Class1()
        {
        }
    }

    public class Class2
    {
        // Violation occurs because Debug.Fail is a conditional method.
        // The finalizer will contain code only if the DEBUG directive
        // symbol is present at compile time. When the DEBUG
        // directive is not present, the finalizer will still exist, but
        // it will be empty.
        ~Class2()
        {
            Debug.Fail("Finalizer called!");
        }
    }

    public class Class3
    {
#if DEBUG
        // Violation will not occur because the finalizer will exist and
        // contain code when the DEBUG directive is present. When the
        // DEBUG directive is not present, the finalizer will not exist,
        // and therefore not be empty.
        ~Class3()
        {
            Debug.Fail("Finalizer called!");
        }
#endif
    }
```

CA1822:将成员标记为 static

2021/11/16 •

	▮
■ ID	CA1822
■	"性能"
▬▬▬	非中断性 - 无论进行了何种更改，如果成员在程序集外部不可见，则为非中断修复。 非中断 - 如果只使用 `this` 关键字将成员更改为实例成员，则为非中断修复。 非中断 - 如果将成员从实例成员更改为静态成员，并且该成员在程序集外部可见，则为中断修复。

原因

不访问实例数据的成员未标记为静态(在 Visual Basic 中为共享)。

规则说明

可以将不访问实例数据或不调用实例方法的成员标记为静态(在 Visual Basic 中为共享)。在将这些方法标记为 static 之后，编译器将向这些成员发出非虚拟调用站点。发出非虚拟调用网站将禁止在运行时检查每个调用，以确保当前对象指针为非 NULL。这会使性能敏感的代码的性能得到显著提高。在某些情况下，访问当前对象实例失败表示存在正确性问题。

如何解决冲突

将成员标记为静态(在 Visual Basic 中为共享)，或在方法主体中使用"this"/"Me"(如果适用)。

何时禁止显示警告

对于以前发布的代码，可禁止显示此规则的警告，因为修复是一项中断性变更。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别(性能)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1812:避免未实例化的内部类

CA1823:避免未使用的私有字段

2021/11/16 •

■ ID	CA1823
■	“性能”
▬▬	非中断

原因

当代码中存在专用字段但任何代码路径均未使用该字段时，会报告此规则。

规则说明

检测到程序集内有似乎未访问过的私有字段。

如何解决冲突

若要解决此规则的冲突，请删除该字段或添加使用该字段的代码。

何时禁止显示警告

可禁止显示此规则的警告。

相关规则

- CA1812:避免未实例化的内部类
- CA1801:检查未使用的参数

CA1824:用 NeutralResourcesLanguageAttribute 标记程序集

2021/11/16 •

■ ID	CA1824
■	"性能"
■■■■■	非中断

原因

程序集包含基于 ResX 的资源，但没有向其应用 System.Resources.NeutralResourcesLanguageAttribute。

规则说明

NeutralResourcesLanguageAttribute 属性通知应用默认区域性的资源控制器。如果默认区域性的资源嵌入在应用的主程序集中，并且 ResourceManager 必须检索与默认区域性属于与同一区域性的资源，则 ResourceManager 会自动使用位于主程序集内的资源，而不是搜索附属程序集。这样可绕过常用程序集探测，提高所加载的第一个资源的查找性能，并可缩小工作集。

> **TIP**
> 有关 ResourceManager 用于探测资源文件的过程，请参阅打包和部署资源。

解决冲突

若要解决此规则的冲突，请将属性添加到程序集，并指定非特定区域性的资源的语言。

指定资源的非特定语言

1. 在"解决方案资源管理器"中，右键单击项目，然后选择"属性"。
2. 选择"包"选项卡。

> **NOTE**
> 如果你的项目是一个 .NET Framework 项目，请选择"应用程序"选项卡，然后选择"程序集信息"。

3. 从"非特定语言"或程序集非特定语言"下拉列表中选择语言。
4. 选择"确定"。

何时禁止显示警告

允许禁止显示此规则发出的警告。但是，启动性能可能会降低。若要禁止显示此警告，请向 .globalconfig 或 .editorconfig 文件添加 `dotnet_diagnostic.CA1824.severity = none`。

请参阅

- NeutralResourcesLanguageAttribute
- .NET 应用中的资源

CA1825:避免数组分配长度为零

2021/11/16 •

	[
■ ID	CA1825
■	"性能"
▬▬▬	非中断

原因

分配了一个不包含任何元素的空 Array。

规则说明

初始化长度为零的数组将导致不必要的内存分配。请改为通过调用 Array.Empty 方法来使用静态分配的空数组实例。内存分配在此方法的所有调用之间共享。

如何解决冲突

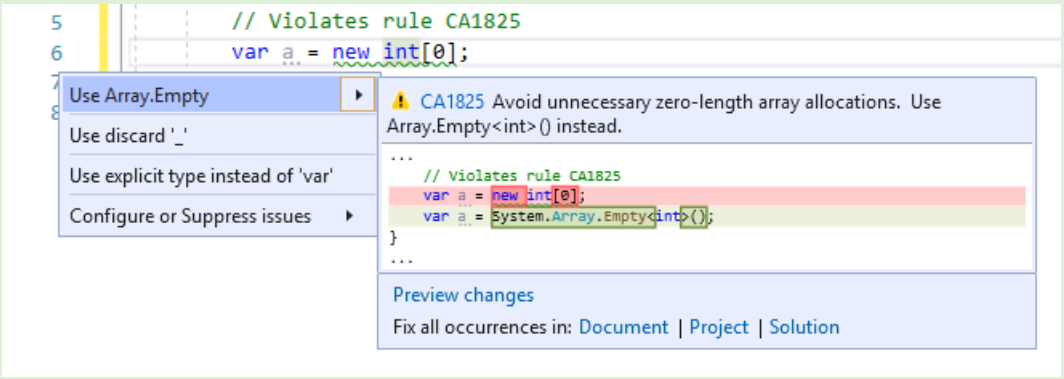
若要解决冲突，请将长度为零的数组分配替换为对 Array.Empty 的调用。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
class C
{
    public void M1()
    {
        // Violates rule CA1825.
        var a = new int[0];
    }
}
```

```
class C
{
    public void M1()
    {
        // Resolves rule CA1825 violation.
        var a = System.Array.Empty<int>();
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用该修补程序，请将光标置于数组分配上，然后按 Ctrl+.（句点）。从显示的选项列表中选择“使用 Array.Empty”。



何时禁止显示警告

如果不在乎额外的内存分配，可禁止显示此规则的冲突。

相关规则

- CA1814:与多维数组相比，首选使用交错数组

另请参阅

- 性能规则

CA1826:使用属性，而不是 Linq Enumerable 方法

2021/11/16 •

	▮
■ ID	CA1826
■	"性能"
▬▬	非中断

原因

对支持等效且更高效的属性的类型使用了 Enumerable LINQ 方法。

规则说明

此规则在具有等效但更高效的属性的类型集合上标记 Enumerable LINQ 方法调用，以提取相同的数据。

此规则分析以下集合类型：

- 实现 IReadOnlyList<T> 但不实现 IList<T> 的类型

此规则标记针对这些集合类型对以下方法进行的调用：

- System.Linq.Enumerable.Count 方法
- System.Linq.Enumerable.First 方法
- System.Linq.Enumerable.FirstOrDefault 方法
- System.Linq.Enumerable.Last 方法
- System.Linq.Enumerable.LastOrDefault 方法

经过分析的集合类型和/或方法可能会在将来扩展，以涵盖更多的情况。

如何解决冲突

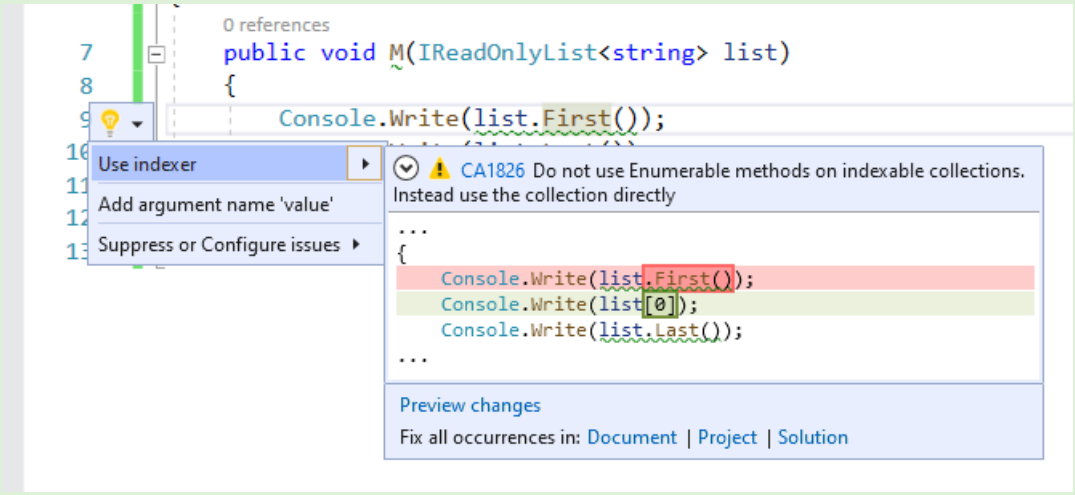
若要解决冲突，请将 Enumerable 方法调用替换为属性访问。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System;
using System.Collections.Generic;
using System.Linq;

class C
{
    public void M(IReadOnlyList<string> list)
    {
        Console.Write(list.First());
        Console.Write(list.Last());
        Console.Write(list.Count());
    }
}
```

```
using System;
using System.Collections.Generic;

class C
{
    public void M(IReadOnlyList<string> list)
    {
        Console.Write(list[0]);
        Console.Write(list[list.Count - 1]);
        Console.Write(list.Count);
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“使用索引器”。



何时禁止显示警告

如果你不关心特定 Enumerable 方法调用造成的性能影响，可禁止显示此规则的冲突警告。

相关规则

- CA1827:如果可以使用 Any，请勿使用 Count/LongCount
- CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync
- CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法

另请参阅

- 性能规则

CA1827:如果可以使用 Any，请勿使用 Count/LongCount

2021/11/16 •

	▮
■ ID	CA1827
■	"性能"
▬▬▬	非中断

原因

在使用 Any 方法会更有效的情况下使用了 Count 或 LongCount 方法。

规则说明

此规则将标记 Count 和 LongCount LINQ 方法调用，用于检查集合是否至少有一个元素。这些方法调用需要枚举整个集合来计算计数。使用 Any 方法进行相同的检查速度更快，因为它可以避免枚举集合。

如何解决冲突

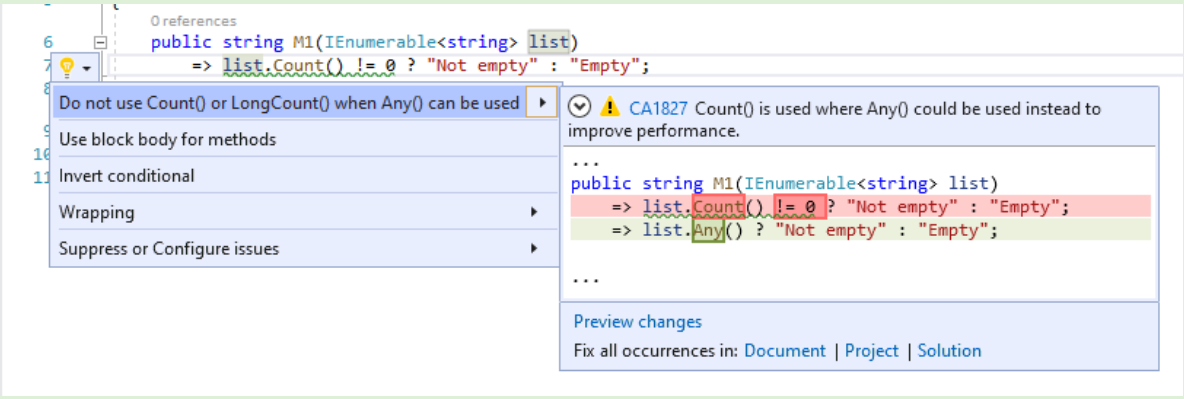
若要解决冲突，请将 Count 或 LongCount 方法调用替换为 Any 方法。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public string M1(IEnumerable<string> list)
        => list.Count() != 0 ? "Not empty" : "Empty";

    public string M2(IEnumerable<string> list)
        => list.LongCount() > 0 ? "Not empty" : "Empty";
}
```

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public string M1(IEnumerable<string> list)
        => list.Any() ? "Not empty" : "Empty";

    public string M2(IEnumerable<string> list)
        => list.Any() ? "Not empty" : "Empty";
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于数组冲突上，然后按 Ctrl+。（句点）。从提供的选项列表中，选择“如果可以使用 Any()，请勿使用 Count() 或 LongCount()”。



何时禁止显示警告

如果不关心不必要的集合枚举计算计数对性能产生的影响，则可禁止显示此规则的冲突警告。

相关规则

- CA1826:使用属性，而不是 Linq Enumerable 方法
- CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync
- CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法

另请参阅

- 性能规则

CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync

2021/11/16 •

	[
■ ID	CA1828
■	"性能"
▬▬▬	非中断

原因

在使用 AnyAsync 方法会更有效的情况下使用了 CountAsync 或 LongCountAsync 方法。

规则说明

此规则将标记 CountAsync 和 LongCountAsync LINQ 方法调用，用于检查集合是否至少有一个元素。这些方法调用需要枚举整个集合来计算计数。使用 AnyAsync 方法进行相同的检查速度更快，因为它可以避免枚举集合。

如何解决冲突

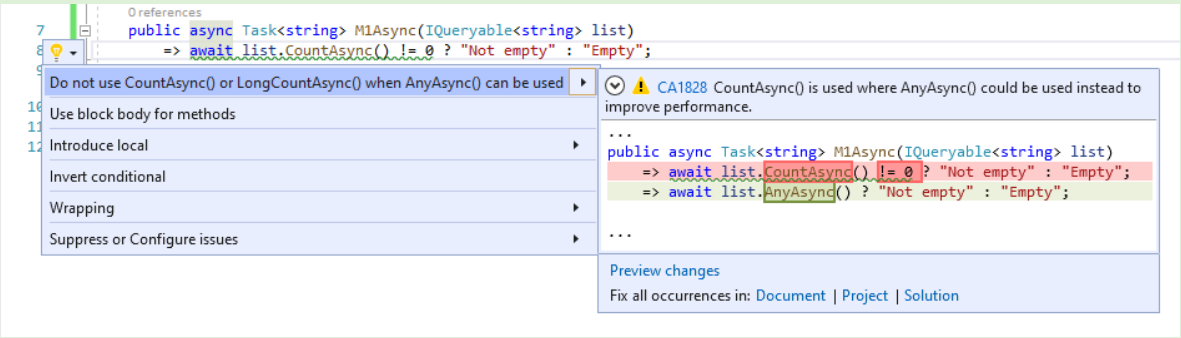
若要解决冲突，请将 CountAsync 或 LongCountAsync 方法调用替换为 AnyAsync 方法。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System.Linq;
using System.Threading.Tasks;
using static Microsoft.EntityFrameworkCore.EntityFrameworkQueryableExtensions;

class C
{
    public async Task<string> M1Async(IQueryable<string> list)
        => await list.CountAsync() != 0 ? "Not empty" : "Empty";

    public async Task<string> M2Async(IQueryable<string> list)
        => await list.LongCountAsync() > 0 ? "Not empty" : "Empty";
}
```

```
using System.Linq;
using System.Threading.Tasks;
using static Microsoft.EntityFrameworkCore.EntityFrameworkQueryableExtensions;

class C
{
    public async Task<string> M1Async(IQueryable<string> list)
        => await list.AnyAsync() ? "Not empty" : "Empty";

    public async Task<string> M2Async(IQueryable<string> list)
        => await list.AnyAsync() ? "Not empty" : "Empty";
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中，选择“如果可以使用 AnyAsync()，请勿使用 CountAsync() 或 LongCountAsync()”。



何时禁止显示警告

如果不关心不必要的集合枚举计算计数对性能产生的影响，则可禁止显示此规则的冲突警告。

相关规则

- CA1826:使用属性，而不是 Linq Enumerable 方法
- CA1827:如果可以使用 Any，请勿使用 Count/LongCount
- CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法

另请参阅

- 性能规则

CA1829:使用 Length/Count 属性，而不是 Enumerable.Count 方法

2021/11/16 •

	ı
■ ID	CA1829
■	"性能"
▬▬▬	非中断

原因

对支持等效且更高效的 `Length` 或 `Count` 属性的类型使用了 Count LINQ 方法。

规则说明

此规则在具有等效但更高效的 `Length` 或 `Count` 属性以提取相同数据的类型的集合上标记 Count LINQ 方法调用。 `Length` 或 `Count` 属性不枚举集合，因此更高效。

此规则标记具有 `Length` 属性的以下集合类型上的 Count 调用：

- System.Array
- System.Collections.Immutable.ImmutableArray<T>

此规则标记具有 `Count` 属性的以下集合类型上的 Count 调用：

- System.Collections.ICollection
- System.Collections.Generic.ICollection<T>
- System.Collections.Generic.IReadOnlyCollection<T>

分析后的集合类型可能会在将来扩展，以涵盖更多的情况。

如何解决冲突

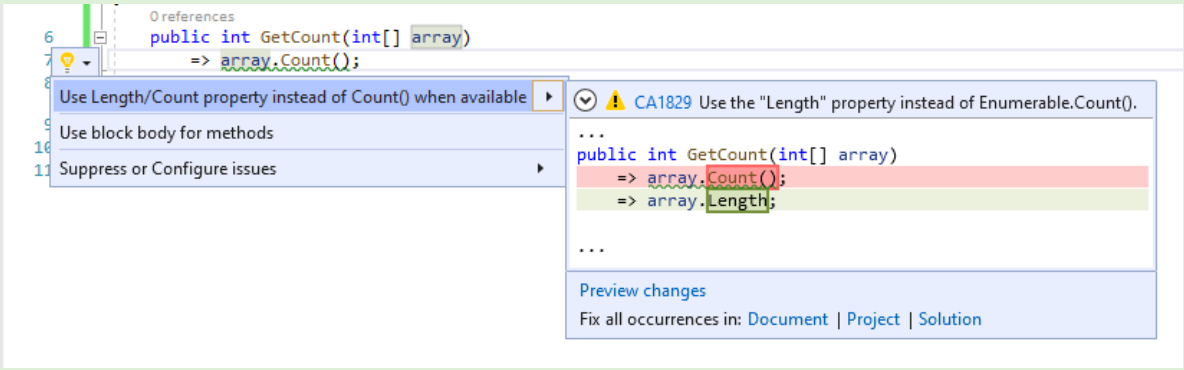
若要解决冲突，请将 Count 方法调用替换为使用 `Length` 或 `Count` 属性访问。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System.Collections.Generic;
using System.Linq;

class C
{
    public int GetCount(int[] array)
        => array.Count();

    public int GetCount(ICollection<int> collection)
        => collection.Count();
}
```

```
using System.Collections.Generic;

class C
{
    public int GetCount(int[] array)
        => array.Length;

    public int GetCount(ICollection<int> collection)
        => collection.Count;
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“在可用时使用 Length/Count 属性，而不是 Count()”。



何时禁止显示警告

如果不关心不必要的集合枚举计算计数对性能产生的影响，则可禁止显示此规则的冲突警告。

相关规则

- CA1826:使用属性，而不是 Linq Enumerable 方法
- CA1827:如果可以使用 Any，请勿使用 Count/LongCount
- CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync

另请参阅

- 性能规则

CA1830：在 StringBuilder 上优先使用强类型 Append 和 Insert 方法重载。

2021/11/16 •

■ ID	CA1830
■	"性能"
■■■■■	非中断

原因

调用 StringBuilder `Append` 或 `Insert` 方法时，使用的是对 `Append` 或 `Insert` 方法有专用重载的类型调用 `ToString` 生成的参数。

规则说明

Append 和 Insert 为除 String 之外的多种类型提供重载。在可能的情况下，请首先使用强类型重载，而不是使用 ToString () 和基于字符串的重载。

如何解决冲突

从调用中删除不必要的 `ToString()`。

```
using System.Text;

class C
{
    int _value;

    // Violation
    public void Log(StringBuilder destination)
    {
        destination.Append("Value: ").Append(_value.ToString()).AppendLine();
    }

    // Fixed
    public void Log(StringBuilder destination)
    {
        destination.Append("Value: ").Append(_value).AppendLine();
    }
}
```

何时禁止显示警告

如果不关心不必要的字符串分配对性能的影响，可禁止显示此规则的冲突警告。

另请参阅

- 性能规则

CA1831:在合适的情况下，为字符串使用 AsSpan 而不是基于范围的索引器

2021/11/16 •

	ı
■ ID	CA1831
■	"性能"
■■■■■	非中断

原因

对字符串使用了范围索引器，并将值隐式分配给了 `ReadOnlySpan<char>`。

规则说明

对字符串使用范围索引器并将其分配给范围类型时，将触发此规则。Span<T> 上的范围索引器是非复制的 Slice 操作，但对于字符串中的范围索引器，将使用方法 Substring 而不是 Slice。这会生成字符串所请求部分的副本。此副本在隐式用作 ReadOnlySpan<T> 或 ReadOnlyMemory<T> 值时常常是不必要的。如果不需要副本，请使用 AsSpan 方法来避免不必要的副本。如果需要副本，请先将其分配给本地变量，或者添加显式强制转换。仅在对范围索引器操作的结果使用隐式强制转换时，分析器才会报告。

检测

隐式转换：

```
ReadOnlySpan<char> slice = str[a..b];
```

不检测

显式转换：

```
ReadOnlySpan<char> slice = (ReadOnlySpan<char>)str[a..b];
```

如何解决冲突

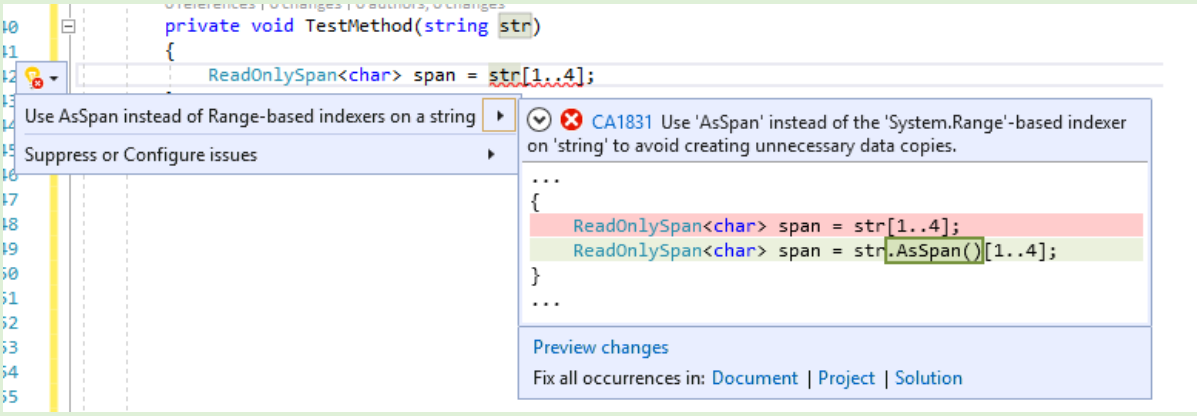
若要解决此规则的冲突，请对字符串使用 AsSpan 而不是基于 Range 的索引器，以避免创建不必要的数据副本。

```
public void TestMethod(string str)
{
    // The violation occurs
    ReadOnlySpan<char> slice = str[1..3];
    ...
}
```

```
public void TestMethod(string str)
{
    // The violation fixed with AsSpan extension method
    ReadOnlySpan<char> slice = str.AsSpan()[1..3];
    ...
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于数组冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“对字符串使用 AsSpan 而不是基于范围的索引器”。



何时禁止显示警告

如果打算创建副本，可禁止显示此规则的冲突。除了代码分析警告的常用方法外，还可以添加显式强制转换以避免显示此警告。

```
public void TestMethod(string str)
{
    // The violation occurs.
    ReadOnlySpan<char> slice = str[1..3];
    ...
}
```

```
public void TestMethod(string str)
{
    // The violation avoided with explicit casting.
    ReadOnlySpan<char> slice = (ReadOnlySpan<char>)str[1..3];
    ...
}
```

相关规则

- CA1832:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 ReadOnlySpan 或 ReadOnlyMemory 部分
- CA1833:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 Span 或 Memory 部分

另请参阅

- 性能规则

CA1832:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 ReadOnlySpan 或 ReadOnlyMemory 部分

2021/11/16 •

	ı
■ ID	CA1832
■	"性能"
▬▬▬	非中断

原因

对数组使用范围索引器并向 ReadOnlySpan<T> 或 ReadOnlyMemory<T> 隐式赋值。

规则说明

对数组使用范围索引器并分配给内存或范围类型：Span<T> 上的范围索引器是非复制的 Slice 操作，但对于数组上的范围索引器，将使用方法 GetSubArray 而不是 Slice，这会生成数组所请求部分的副本。此副本在隐式用作 ReadOnlySpan<T> 或 ReadOnlyMemory<T> 值时常常是不必要的。如果不需要副本，请使用 AsSpan 或 AsMemory 方法来避免不必要的副本。如果需要副本，请先将其分配给本地变量，或者添加显式强制转换。仅在对范围索引器操作的结果使用隐式强制转换时，分析器才会报告。

检测

隐式转换：

- `ReadOnlySpan<SomeT> slice = arr[a..b];`
- `ReadOnlyMemory<SomeT> slice = arr[a..b];`

不检测

显式转换：

- `ReadOnlySpan<SomeT> slice = (ReadOnlySpan<SomeT>)arr[a..b];`
- `ReadOnlyMemory<SomeT> slice = (ReadOnlyMemory<SomeT>)arr[a..b];`

如何解决冲突

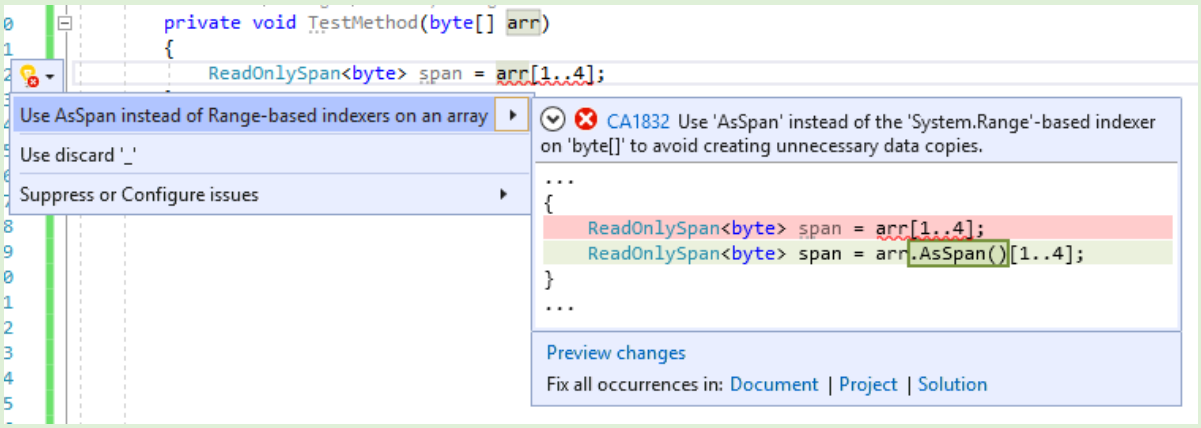
若要解决此规则的冲突，请执行以下操作：使用 AsSpan 或 AsMemory 扩展方法以避免创建不必要的数据副本。

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violation occurs for both statements below
        ReadOnlySpan<byte> tmp1 = arr[0..2];
        ReadOnlyMemory<byte> tmp3 = arr[5..8];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violations fixed with AsSpan or AsMemory accordingly
        ReadOnlySpan<byte> tmp1 = arr.AsSpan()[0..2];
        ReadOnlyMemory<byte> tmp3 = arr.AsMemory()[5..8];
        ...
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修复。若要使用它，请将光标置于数组冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“在数组上使用 AsSpan 而不是基于范围的索引器”。



何时禁止显示警告

如果需要创建副本，则可禁止显示此规则的冲突。若要禁止显示此警告，只需添加显式强制转换即可。

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation occurs
        ReadOnlySpan<byte> tmp1 = arr[0..2];
        ReadOnlyMemory<byte> tmp3 = arr[5..8];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation fixed with explicit casting
        ReadOnlySpan<byte> tmp1 = (ReadOnlySpan<byte>)arr[0..2];
        ReadOnlyMemory<byte> tmp3 = (ReadOnlyMemory<byte>)arr[5..8];
        ...
    }
}
```

相关规则

- CA1831:在合适的情况下，为字符串使用 AsSpan 而不是基于范围的索引器
- CA1833:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 Span 或 Memory 部分

另请参阅

- 性能规则

CA1833:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 Span 或 Memory 部分

2021/11/16 •

	ı
■ ID	CA1833
■	"性能"
▬▬▬	非中断

原因

对数组使用范围索引器并向 Span<T> 或 Memory<T> 隐式赋值。

规则说明

对数组使用范围索引器并分配给内存或范围类型：Span<T> 上的范围索引器是非复制的 Slice 操作，但对于数组中的范围索引器，将使用方法 GetSubArray 而不是 Slice，这会生成所请求数组部分的副本。此副本在隐式用作 Span<T> 或 Memory<T> 值时常常是不必要的。如果不需要副本，请使用 AsSpan 或 AsMemory 方法来避免不必要的副本。如果需要副本，请先将其分配给本地变量，或者添加显式强制转换。仅在对范围索引器操作的结果使用隐式强制转换时，分析器才会报告。

检测

隐式转换：

- `Span<SomeT> slice = arr[a..b];`
- `Memory<SomeT> slice = arr[a..b];`

不检测

显式转换：

- `Span<SomeT> slice = (Span<SomeT>)arr[a..b];`
- `Memory<SomeT> slice = (Memory<SomeT>)arr[a..b];`

如何解决冲突

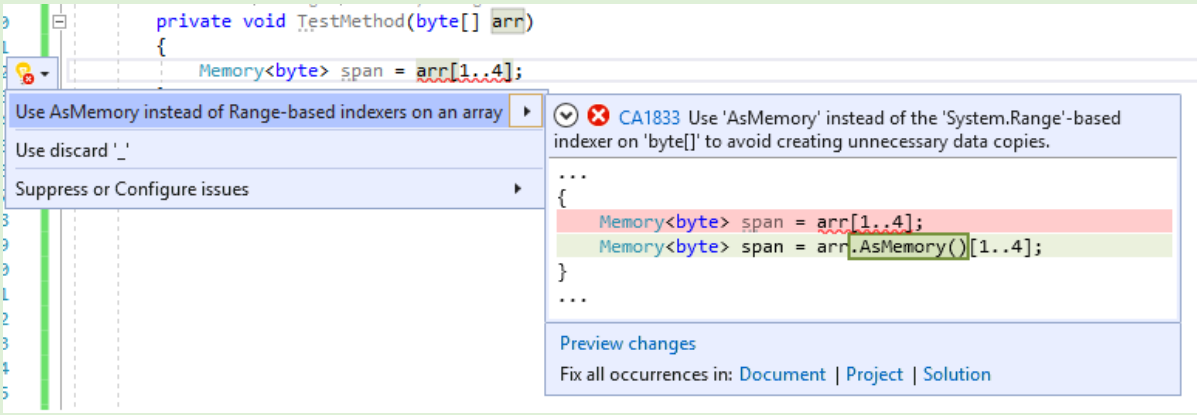
若要解决此规则的冲突：请使用 AsSpan 或 AsMemory 扩展方法，以免创建不必要的数据副本。

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violation occurs for both statements below
        Span<byte> tmp2 = arr[0..5];
        Memory<byte> tmp4 = arr[5..10];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte[] arr)
    {
        // The violations fixed with AsSpan or AsMemory accordingly
        Span<byte> tmp2 = arr.AsSpan()[0..5];
        Memory<byte> tmp4 = arr.AsMemory()[5..10];
        ...
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于数组冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“在数组上使用 AsMemory 而不是基于范围的索引器”。



何时禁止显示警告

如果需要创建副本，则可禁止显示此规则的冲突。若要禁止显示此警告，只需添加显式强制转换即可。

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation occurs
        Span<byte> tmp1 = arr[0..5];
        Memory<byte> tmp2 = arr[5..10];
        ...
    }
}
```

```
class C
{
    public void TestMethod(byte arr[])
    {
        // The violation fixed with explicit casting
        Span<byte> tmp1 = (Span<byte>)arr[0..5];
        Memory<byte> tmp2 = (Memory<byte>)arr[5..10];
        ...
    }
}
```

相关规则

- CA1831:在合适的情况下，为字符串使用 AsSpan 而不是基于范围的索引器
- CA1832:使用 AsSpan 或 AsMemory 而不是基于范围的索引器来获取数组的 ReadOnlySpan 或 ReadOnlyMemory 部分

另请参阅

- 性能规则

CA1834:对单字符字符串使用 StringBuilder.Append(char)

2021/11/16 •

■ ID	CA1834
■	"性能"
■■■■■	非中断

原因

将单位长度字符串传递给 Append 方法时，将触发此规则。

规则说明

使用单位长度字符串调用 `StringBuilder.Append` 时，请考虑使用 `const char` 而不是单位长度 `const string` 来提高性能。

如何解决冲突

可以手动解决冲突，在某些情况下，也使用快速操作来修复 Visual Studio 中的代码。示例：

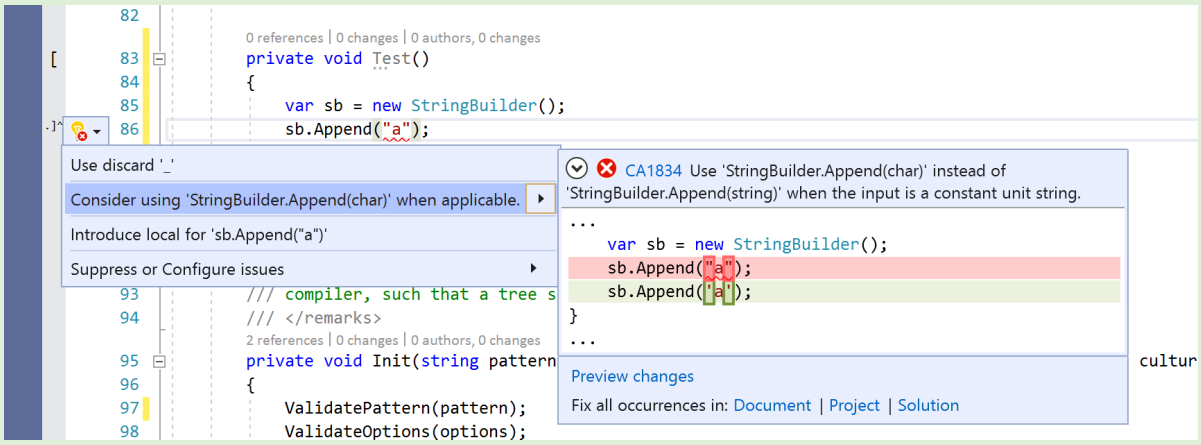
示例 1

使用单位长度的字符串文本的 `StringBuilder.Append` 调用：

```
using System;
using System.Text;

namespace TestNamespace
{
    class TestClass
    {
        private void TestMethod()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append("a");
        }
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中，选择“请考虑使用“StringBuilder.Append(char)”（若适用）。”



Visual Studio 应用的修补程序：

```
using System;
using System.Text;

namespace TestNamespace
{
    class TestClass
    {
        private void TestMethod()
        {
            StringBuilder sb = new StringBuilder();
            sb.Append('a');
        }
    }
}
```

在某些情况下，例如使用单位长度 `const string` 类字段时，Visual Studio 不建议使用代码修补程序（但分析器仍会触发）。这些实例需要手动修复。

示例 2

使用单位长度的 `const string` 类字段的 `StringBuilder.Append` 调用：

```
using System;
using System.Text;

namespace TestNamespace
{
    public class Program
    {
        public const string unitString = "a";

        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(unitString);
        }
    }
}
```

仔细分析后，此处的 `unitString` 可以更改为 `char`，而不会导致任何生成错误。

```
using System;
using System.Text;

namespace TestNamespace
{
    public class Program
    {
        public const char unitString = 'a';

        static void Main(string[] args)
        {
            StringBuilder sb = new StringBuilder();
            sb.Append(unitString);
        }
    }
}
```

何时禁止显示警告

如果你不关心在使用 `StringBuilder` 时提高性能，可以禁止显示此规则的冲突。

请参阅

- 性能规则

CA1835：在基于流的类中，首选 ReadAsync/WriteAsync 方法的基于内存的重载

2021/11/16 •

▬	PreferStreamAsyncMemoryOverloads
■ ID	CA1835
■	“性能”
▬▬▬	非中断

原因

此规则查找 `ReadAsync` 和 `WriteAsync` 的基于字节数组的方法重载的等待调用，并建议改为使用基于内存的方法重载，因为它们的效率更高。

规则说明

基于内存的方法重载具有比基于字节数组的重载更有效的内存使用。

此规则适用于从 Stream 继承的任何类的 `ReadAsync` 和 `WriteAsync` 调用。

仅当方法前面带有 `await` 关键字时，此规则才有效。

ReadAsync(Byte[], Int32, Int32, CancellationToken)	ReadAsync(Memory<Byte>, CancellationToken)
ReadAsync(Byte[], Int32, Int32)	`CancellationToken` 设置为 `default`（在 C# 中）或 `Nothing`（在 Visual Basic 中）的 ReadAsync(Memory<Byte>, CancellationToken)。
WriteAsync(Byte[], Int32, Int32, CancellationToken)	WriteAsync(ReadOnlyMemory<Byte>, CancellationToken)
WriteAsync(Byte[], Int32, Int32)	`CancellationToken` 设置为 `default`（在 C# 中）或 `Nothing`（在 Visual Basic 中）的 WriteAsync(ReadOnlyMemory<Byte>, CancellationToken)。

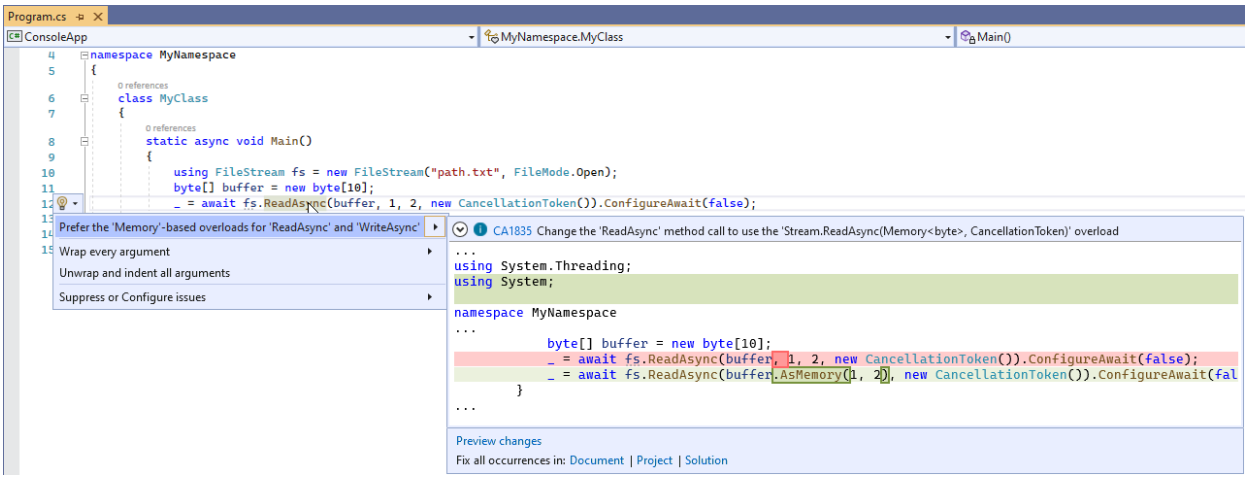
IMPORTANT

确保将 `offset` 和 `count` 整数参数传递到创建的 `Memory` 或 `ReadOnlyMemory` 实例。

> **NOTE**
>
> 规则 CA1835 适用于所有提供基于内存的重载的 .NET 版本：
>
> - .NET Standard 2.1 及更高版本。
> - .NET Core 2.1 及更高版本。

如何解决冲突

可以手动修复，也可以选择让 Visual Studio 执行修复，方法是将鼠标悬停在方法调用旁显示的灯泡图标上，然后选择建议的更改。示例：



此规则可以检测 `ReadAsync` 和 `WriteAsync` 方法的多种冲突。下面是此规则可检测到的情况示例：

示例 1

`ReadAsync` 的调用，未使用和使用 `CancellationToken` 参数：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            byte[] buffer = new byte[s.Length];
            await s.ReadAsync(buffer, 0, buffer.Length);
            await s.ReadAsync(buffer, 0, buffer.Length, ct);
        }
    }
}
```

解决方法：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            byte[] buffer = new byte[s.Length];
            await s.ReadAsync(buffer.AsMemory(0, buffer.Length));
            await s.ReadAsync(buffer.AsMemory(0, buffer.Length), ct);
        }
    }
}
```

示例 2

`WriteAsync` 的调用，未使用和使用 `CancellationToken` 参数：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod(CancellationToken ct)
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer, 0, buffer.Length);
            await s.WriteAsync(buffer, 0, buffer.Length, ct);
        }
    }
}
```

解决方法：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer.AsMemory(0, buffer.Length));
            await s.WriteAsync(buffer.AsMemory(0, buffer.Length), ct);
        }
    }
}
```

示例 3

使用 `ConfigureAwait` 的调用：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer1 = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer1, 0, buffer1.Length).ConfigureAwait(false);

            byte[] buffer2 = new byte[s.Length];
            await s.ReadAsync(buffer2, 0, buffer2.Length).ConfigureAwait(true);
        }
    }
}
```

解决方法：

```
using System;
using System.IO;
using System.Threading;

class MyClass
{
    public async void MyMethod()
    {
        using (FileStream s = File.Open("path.txt", FileMode.Open))
        {
            byte[] buffer1 = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
            await s.WriteAsync(buffer1.AsMemory(0, buffer1.Length)).ConfigureAwait(false);

            byte[] buffer2 = new byte[s.Length];
            await s.ReadAsync(buffer2.AsMemory(0, buffer.Length)).ConfigureAwait(true);
        }
    }
}
```

无冲突

下面是不会触发此规则的一些调用示例。

返回值保存在 `Task` 变量中，而不是在等待：

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class MyClass
{
    public void MyMethod()
    {
        byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            Task t = s.WriteAsync(buffer, 0, buffer.Length);
        }
    }
}
```

返回值由包装方法返回，而不是在等待：

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class MyClass
{
    public Task MyMethod(FileStream s, byte[] buffer)
    {
        return s.WriteAsync(buffer, 0, buffer.Length);
    }
}
```

返回值用于调用 `ContinueWith`，这是在等待的方法：

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

class MyClass
{
    public void MyMethod()
    {
        byte[] buffer = { 0xBA, 0x5E, 0xBA, 0x11, 0xF0, 0x07, 0xBA, 0x11 };
        using (FileStream s = new FileStream("path.txt", FileMode.Create))
        {
            await s.WriteAsync(buffer, 0, buffer.Length).ContinueWith(c => { /* ... */ });
        }
    }
}
```

何时禁止显示警告

如果不考虑在基于流的类中读取或写入缓冲区时提高性能，则可以放心地抑制此规则的冲突。

另请参阅

- 性能规则

CA1836:可用时最好使用 IsEmpty (而不是 Count)

2021/11/16 •

	▮
■ ID	CA1836
■	"性能"
▬▬▬	非中断

原因

使用了 `Count` 或 `Length` 属性或 Count<TSource>(IEnumerable<TSource>) 扩展方法，通过将值与 `0` 或 `1` 进行比较来确定对象是否包含任何项，以及对象是否具有更有效的 `IsEmpty` 属性可以代替使用。

规则说明

当将 `Count` 和 `Length` 属性或 Count<TSource>(IEnumerable<TSource>) 和 LongCount<TSource>(IEnumerable<TSource>) LINQ 方法用于确定对象是否包含任何项以及对象是否具有更有效的 `IsEmpty` 属性时，此规则将标记对它们的调用。

此规则的分析最初与类似规则 CA1827、CA1828 和 CA1829 重叠，这些规则的分析器与 CA1836 的分析器合并在一起，以在发生重叠时报告最佳诊断。

如何解决冲突

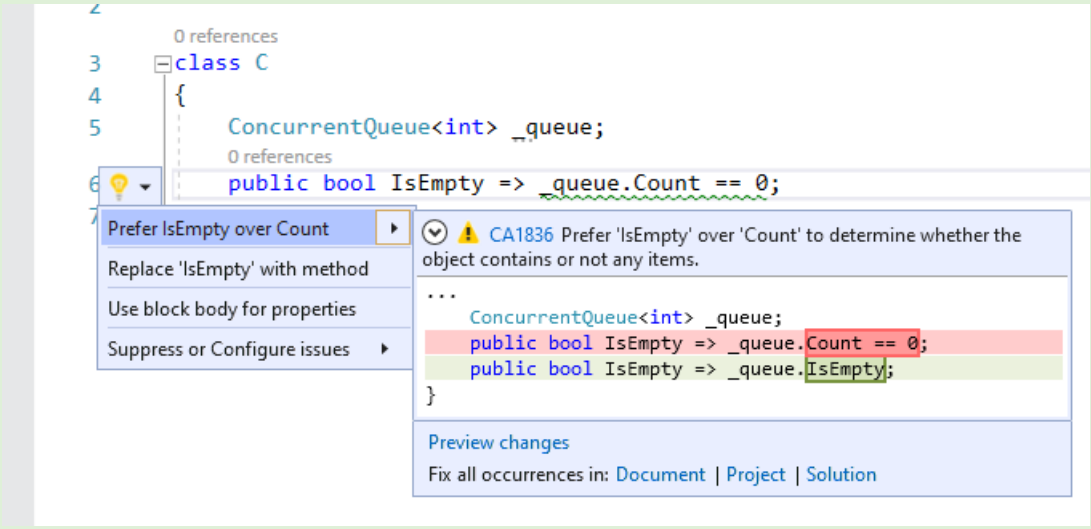
若要解决冲突，在使用 `IsEmpty` 属性访问来确定对象是否为空的操作中，当使用 Count<TSource>(IEnumerable<TSource>) 或 LongCount<TSource>(IEnumerable<TSource>) 方法调用或 `Length` 或 `Count` 属性访问时，请将其替换。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System.Collections.Concurrent;

class C
{
    ConcurrentQueue<int> _queue;
    public bool IsEmpty => _queue.Count == 0;
}
```

```
using System.Collections.Concurrent;

class C
{
    ConcurrentQueue<int> _queue;
    public bool IsEmpty => _queue.IsEmpty;
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“最好使用‘IsEmpty’而不是‘Count’”来确定对象是否包含任何项。



何时禁止显示警告

如果不关心不必要的项枚举是否会对计数计算的性能产生影响，可禁止显示此规则的冲突警告。

相关规则

- CA1827:如果可以使用 Any，请勿使用 Count/LongCount
- CA1828:如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync
- CA1829：如果可以使用 AnyAsync，请勿使用 CountAsync/LongCountAsync

另请参阅

- 性能规则

CA1837：使用 Environment.ProcessId 而不是 Process.GetCurrentProcess().Id

2021/11/16 •

■ ID	CA1837
■	“性能”
■■■■■	非中断

原因

此规则会查找对 `System.Diagnostics.Process.GetCurrentProcess().Id` 的调用，并建议改用 `System.Environment.ProcessId`，因为这样更高效。

规则说明

`System.Diagnostics.Process.GetCurrentProcess().Id` 成本较高：

- 它分配 Process 实例，通常只为了获取 `Id`。
- 需要处置 Process 实例，这会影响性能。
- 很容易忘记调用 Process 实例上的 Dispose()。
- 如果除了 `Id` 之外没有其他内容使用 `Process` 实例，那么随着引用的类型图增加，链接大小也会不必要的增长。
- 发现或查找此 API 有点困难。

`System.Environment.ProcessId` 可避免上述所有情况。

> **NOTE**
> 从 .NET 5.0 开始可以使用规则 CA1837。

如何解决冲突

可以手动解决冲突，或者在某些情况下，使用快速操作来修复 Visual Studio 中的代码。

以下两个代码片段显示了规则冲突及其解决方法：

```
using System.Diagnostics;

class MyClass
{
    void MyMethod()
    {
        int pid = Process.GetCurrentProcess().Id;
    }
}
```

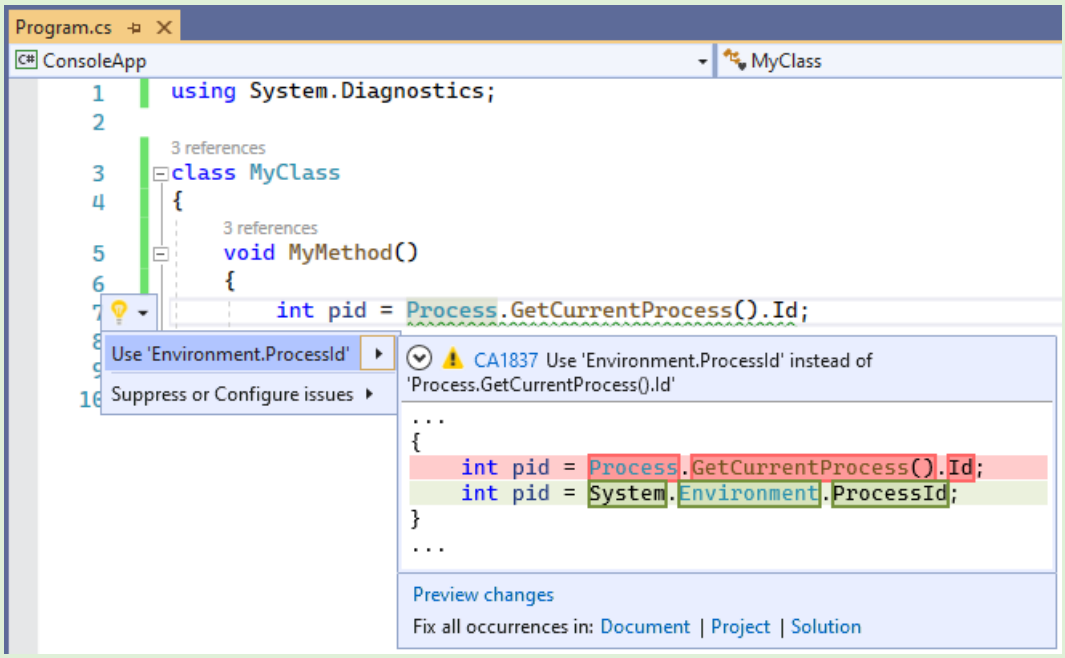
```
Imports System.Diagnostics

Class MyClass
    Private Sub MyMethod()
        Dim pid As Integer = Process.GetCurrentProcess().Id
    End Function
End Class
```

```
using System.Diagnostics;

class MyClass
{
    void MyMethod()
    {
        int pid = System.Environment.ProcessId;
    }
}
```

```
Imports System.Diagnostics

Class MyClass
    Private Sub MyMethod()
        Dim pid As Integer = System.Environment.ProcessId
    End Function
End Class
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“使用‘Environment.ProcessId’而不是‘Process.GetCurrentProcess().Id’”。



何时禁止显示警告

如果你不关心不必要的分配和最终处置 Process 实例造成的性能影响，可禁止显示此规则的冲突警告。

另请参阅

- 性能规则

CA1838：不要对 P/Invoke 使用 `StringBuilder` 参数

2021/11/16 •

■ ID	CA1838
■	“性能”
■■■■■	非中断

原因

P/Invoke 具有一个 StringBuilder 参数。

规则说明

`StringBuilder` 的封送处理总是会创建一个本机缓冲区副本，这导致一个 P/Invoke 调用出现多次分配。若要将 `StringBuilder` 作为 P/Invoke 参数进行封送，运行时将：

- 分配本机缓冲区。
- 如果是 `In` 参数，请将 `StringBuilder` 的内容复制到本机缓冲区。
- 如果是 `Out` 参数，请将本机缓冲区复制到新分配的托管数组中。

默认情况下，`StringBuilder` 为 `In` 和 `Out`。

此规则在默认情况下为禁用状态，因为它可能需要根据具体情况分析冲突是否值得关注，以及是否可能需要进行重大重构来解决冲突。用户可通过配置其严重性来显式启用此规则。

如何解决冲突

通常情况下，解决冲突涉及到重新处理 P/Invoke 及其调用方以使用缓冲区而不是 `StringBuilder`。具体情况取决于 P/Invoke 的用例。

下面是使用 `StringBuilder` 作为要由本机函数填充的输出缓冲区的常见方案示例：

```
// Violation
[DllImport("MyLibrary", CharSet = CharSet.Unicode)]
private static extern void Foo(StringBuilder sb, ref int length);

public void Bar()
{
    int BufferSize = ...
    StringBuilder sb = new StringBuilder(BufferSize);
    int len = sb.Capacity;
    Foo(sb, ref len);
    string result = sb.ToString();
}
```

对于缓冲区较小且可接受 `unsafe` 代码的用例，可以使用 stackalloc 在堆栈上分配缓冲区：

```
[DllImport("MyLibrary", CharSet = CharSet.Unicode)]
private static extern unsafe void Foo(char* buffer, ref int length);

public void Bar()
{
    int BufferSize = ...
    unsafe
    {
        char* buffer = stackalloc char[BufferSize];
        int len = BufferSize;
        Foo(buffer, ref len);
        string result = new string(buffer);
    }
}
```

对于大型缓冲区，可以分配新数组作为缓冲区：

```
[DllImport("MyLibrary", CharSet = CharSet.Unicode)]
private static extern void Foo([Out] char[] buffer, ref int length);

public void Bar()
{
    int BufferSize = ...
    char[] buffer = new char[BufferSize];
    int len = buffer.Length;
    Foo(buffer, ref len);
    string result = new string(buffer);
}
```

如果频繁调用 P/Invoke 以获取大型缓冲区，可使用 ArrayPool<T> 避免随之而来的重复分配和内存压力：

```
[DllImport("MyLibrary", CharSet = CharSet.Unicode)]
private static extern unsafe void Foo([Out] char[] buffer, ref int length);

public void Bar()
{
    int BufferSize = ...
    char[] buffer = ArrayPool<char>.Shared.Rent(BufferSize);
    try
    {
        int len = buffer.Length;
        Foo(buffer, ref len);
        string result = new string(buffer);
    }
    finally
    {
        ArrayPool<char>.Shared.Return(buffer);
    }
}
```

如果缓冲区大小在运行时之前是未知的，则可能需要根据大小以不同的方式创建缓冲区，以避免使用 `stackalloc` 分配大型缓冲区。

前面的示例使用 2 个字节宽的字符（`CharSet.Unicode`）。如果本机函数使用单字节字符（`CharSet.Ansi`），可使用 `byte` 缓冲区而不是 `char` 缓冲区。例如：

```
[DllImport("MyLibrary", CharSet = CharSet.Ansi)]
private static extern unsafe void Foo(byte* buffer, ref int length);

public void Bar()
{
    int BufferSize = ...
    unsafe
    {
        byte* buffer = stackalloc byte[BufferSize];
        int len = BufferSize;
        Foo(buffer, ref len);
        string result = Marshal.PtrToStringAnsi((IntPtr)buffer);
    }
}
```

如果参数还用作输入，则需要使用显示添加了任何 NULL 终止符的字符串数据来填充缓冲区。

何时禁止显示警告

如果你不关心封送 `StringBuilder` 造成的性能影响，可禁止显示此规则的冲突警告。

另请参阅

- 性能规则
- 本机互操作性最佳做法
- ArrayPool<T>
- stackalloc
- 字符集

CA1841：首选字典包含方法

2021/11/16 •

	[
■ ID	CA1841
■	“性能”
▬▬▬	非中断

原因

此规则可找到在 IDictionary<TKey,TValue> 的 `Keys` 或 `Values` 集合上对 `Contains` 方法的调用，这些调用可替换为在字典本身对 `ContainsKey` 或 `ContainsValue` 方法的调用。

规则说明

对 `Keys` 或 `Values` 集合调用 `Contains` 通常比对字典本身调用 `ContainsKey` 或 `ContainsValue` 开销更高：

- 许多字典实现会延迟对键值集合的实例化，这意味着访问 `Keys` 或 `Values` 集合可能导致额外的分配。
- 如果键/值集合使用显式接口实现来隐藏 ICollection<T> 上的方法，可能最终会对 IEnumerable<T> 上调用扩展方法。这可能会降低性能，尤其是在访问键集合时。大多数字典实现都能为键提供快速的 O(1) 包含检查，而 IEnumerable<T> 上的 `Contains` 扩展方法通常执行较慢的 O(n) 包含检查。

如何解决冲突

若要解决冲突，请将对 `dictionary.Keys.Contains` 或 `dictionary.Values.Contains` 的调用分别替换为对 `dictionary.ContainsKey` 或 `dictionary.ContainsValue` 的调用。

下面的代码片段显示了冲突示例及其解决方法。

```
using System.Collections.Generic;
// Importing this namespace brings extension methods for IEnumerable<T> into scope.
using System.Linq;

class Example
{
    void Method()
    {
        var dictionary = new Dictionary<string, int>();

        //  Violation
        dictionary.Keys.Contains("hello world");

        //  Fixed
        dictionary.ContainsKey("hello world");

        //  Violation
        dictionary.Values.Contains(17);

        //  Fixed
        dictionary.ContainsValue(17);
    }
}
```

```
Imports System.Collection.Generic
' Importing this namespace brings extension methods for IEnumerable(Of T) into scope.
' Note that in Visual Basic, this namespace is often imported automatically throughout the project.
Imports System.Linq

Class Example
    Private Sub Method()
        Dim dictionary = New Dictionary(Of String, Of Integer)

        ' Violation
        dictionary.Keys.Contains("hello world")

        ' Fixed
        dictionary.ContainsKey("hello world")

        ' Violation
        dictionary.Values.Contains(17)

        ' Fixed
        dictionary.ContainsValue(17)
    End Sub
End Class
```

何时禁止显示警告

如果有问题的代码不会对性能造成重大影响，可安全地禁止显示此规则发出的警告。

另请参阅

- 性能规则

CA1844：对“流”进行子分类时，提供异步方法的基于内存的重写

2021/11/16 •

	ı
■ ID	CA1844
■	“性能”
▬▬▬	非中断

原因

派生自 Stream 的类型会重写 ReadAsync(Byte[], Int32, Int32, CancellationToken)，但不会重写 ReadAsync(Memory<Byte>, CancellationToken)。或，派生自 Stream 的类型会重写 WriteAsync(Byte[], Int32, Int32, CancellationToken)，但不会重写 WriteAsync(ReadOnlyMemory<Byte>, CancellationToken)。

规则说明

添加了基于内存的 `ReadAsync` 和 `WriteAsync` 方法来提高性能，这些方法的实现方式有多种：

- 它们分别返回 `ValueTask` 和 `ValueTask<int>`，而不是 `Task` 和 `Task<int>`。
- 它们允许传入任意类型的缓冲区，而无需对数组执行额外的复制。

为了实现这些性能优势，派生自 Stream 的类型必须提供自己的基于内存的实现。否则，将强制默认实现将内存复制到数组中，以便调用基于数组的实现，从而降低性能。当调用方传入不受数组支持的 Memory<T> 或 ReadOnlyMemory<T> 实例时，性能会受到影响。

如何解决冲突

修复冲突的最简单方法是将基于数组的实现重写为基于内存的实现，然后根据基于内存的方法实现基于数组的方法。

何时禁止显示警告

如果以下任一情况适用，则禁止显示此规则的警告是安全的：

- 不需要考虑性能损失的问题。
- 如你所知，你的 `Stream` 子类将始终仅使用基于数组的方法。
- 你的 `Stream` 子类具有不支持基于内存的缓冲区的依赖项。

另请参阅

- 性能规则

CA1845：使用基于跨度的“string.Concat”

2021/11/16 •

	[
■ ID	CA1845
■	“性能”
▬▬▬	非中断

原因

此规则查找包含 Substring 调用的字符串串联表达式，同时建议将 Substring 替换为 AsSpan 并使用基于跨度的 String.Concat 重载。

规则说明

调用 `Substring` 会生成提取的子字符串的副本。通过使用 `AsSpan` 代替 `Substring`，并调用接受跨度的 `string.Concat` 重载，可以消除不必要的字符串分配。

如何解决冲突

若要解决冲突，请执行以下操作：

1. 将字符串串联替换为对 `string.Concat` 的调用，以及
2. 将对 `Substring` 的调用替换为对 `AsSpan` 的调用。

下面的代码片段显示了冲突示例及其解决方法。

```
using System;

class Example
{
    public void Method()
    {
        string text = "fwobz the fwutzle";

        //  Violation: allocations by Substring are wasteful.
        string s1 = text.Substring(10) + "---" + text.Substring(0, 5);

        //  Fixed: using AsSpan avoids allocations of temporary strings.
        string s2 = string.Concat(text.AsSpan(10), "---", text.AsSpan(0, 5));
    }
}
```

何时禁止显示警告

不要禁止显示此规则的警告。当提取的子字符串仅传递给具有基于跨度的等效项的方法时，没有理由在 `AsSpan` 上使用 `Substring`。

另请参阅

- 性能规则

CA1846：首选 `AsSpan` 次选 `Substring`

2021/11/16 •

	▮
■ ID	CA1846
■	“性能”
▬▬	非中断

原因

对其中一个 String.Substring 重载的调用结果被传递给具有接受 `ReadOnlySpan<Char>` 的可用重载的方法。

规则说明

`Substring` 在堆上分配一个新的 `string` 对象并执行提取文本的完整副本。字符串操作是多个程序的性能瓶颈。在热路径上分配很多生存期较短的小型字符串可以产生足够的集合压力来影响性能。当子字符串变大时，`Substring` 创建的 O(n) 副本将变得举足轻重。为了解决这些性能问题，创建了 Span<T> 和 ReadOnlySpan<T> 类型。

许多接受字符串的 API 也具有接受 `ReadOnlySpan<System.Char>` 参数的重载。当此类重载可用时，可以通过调用 `AsSpan` 而不是 `Substring` 来提升性能。

如何解决冲突

若要修复与此规则的冲突，请将对 `string.Substring` 的调用替换为对其中一个 MemoryExtensions.AsSpan 扩展方法的调用。

```
using System;

public void MyMethod(string iniFileLine)
{
    // Violation
    int.TryParse(iniFileLine.Substring(7), out int x);
    int.TryParse(iniFileLine.Substring(2, 5), out int y);

    // Fix
    int.TryParse(iniFileLine.AsSpan(7), out int x);
    int.TryParse(iniFileLine.AsSpan(2, 5), out int y);
}
```

```
Imports System

Public Sub MyMethod(iniFileLine As String)
    Dim x As Integer
    Dim y As Integer

    ' Violation
    Integer.TryParse(iniFileLine.Substring(7), x)
    Integer.TryParse(iniFileLine.Substring(2, 5), y)

    ' Fix
    Integer.TryParse(iniFileLine.AsSpan(7), x)
    Integer.TryParse(iniFileLine.AsSpan(2, 5), y)
End Sub
```

何时禁止显示警告

如果不考虑性能，可禁止显示此规则的警告。

另请参阅

- 性能警告

CA1847：对单个字符使用 string.Contains(char) 而不是 string.Contains(string)

2021/11/16 •

	ı
■ ID	CA1847
■	"性能"
▬▬▬	非中断

原因

当 `string.Contains(char)` 可用时使用 `string.Contains(string)`。

规则说明

在搜索单个字符时，使用 `string.Contains(char)` 可获得比使用 `string.Contains(string)` 时更好的性能。

如何解决冲突

通常，只需使用 char 文本而无需使用字符串文本即可解决规则问题。

```
public bool ContainsLetterI()
{
    var testString = "I am a test string.";
    return testString.Contains("I");
}
```

```
Public Function ContainsLetterI() As Boolean
    Dim testString As String = "I am a test string."
    Return testString.Contains("I")
End Function
```

可将此代码更改为使用 char 文本。

```
public bool ContainsLetterI()
{
    var testString = "I am a test string.";
    return testString.Contains('I');
}
```

```
Public Function ContainsLetterI() As Boolean
    Dim testString As String = "I am a test string."
    Return testString.Contains("I"c)
End Function
```

何时禁止显示警告

如果并不在意所讨论的搜索调用对性能的影响，可禁止显示此规则的冲突警告。

另请参阅

- 性能规则

CA1849：当在异步方法中时，调用异步方法

2021/11/16 •

	[
■ ID	CA1849
■	"性能"
▬▬▬	非中断

原因

从任务返回方法调用时，存在 Async 后缀等效项的所有方法都会生成此警告。此外，调用 `Task.Wait()`、`Task<T>.Result` 或 `Task.GetAwaiter().GetResult()` 将生成此警告。

规则说明

在已属于异步的方法中，对其他方法的调用应指向其存在的异步版本。

如何解决冲突

冲突：

```
Task DoAsync()
{
    file.Read(buffer, 0, 10);
}
```

修复：

等待方法的异步版本：

```
async Task DoAsync()
{
    await file.ReadAsync(buffer, 0, 10);
}
```

何时禁止显示警告

在同步和异步代码有两个单独的代码路径的情况下，使用 if 条件抑制来自此规则的警告很安全。此外，如果要检查任务是否已解决，则使用同步方法和属性很安全。

请参阅

- 性能规则

CA1850：首选静态 `HashData` 方法，而非 `ComputeHash`

2021/11/16 •

■ ID	CA1850
■	"性能"
▬▬▬	非中断

原因

创建派生自 HashAlgorithm 的实例类型，用于调用其 `ComputeHash` 方法，并且该类型具有静态 `HashData` 方法。

规则说明

.NET 5 中引入了以下类型的静态 `HashData` 方法：

- MD5
- SHA1
- SHA256
- SHA384
- SHA512

在只需对某些数据进行哈希处理的情况下，这些方法有助于简化代码。

使用这些静态 `HashData` 方法比创建和管理 `HashAlgorithm` 实例来调用 `ComputeHash` 更高效。

如何解决冲突

通常，通过更改代码以调用 `HashData` 并删除对 `HashAlgorithm` 实例的使用，即可修复规则。

```
public bool CheckHash(byte[] buffer)
{
  using (var sha256 = SHA256.Create())
  {
    byte[] digest = sha256.ComputeHash(buffer);
    return DoesHashExist(digest);
  }
}
```

```
Public Function CheckHash(buffer As Byte()) As Boolean
  Using sha256 As SHA256 = SHA256.Create()
    Dim digest As Byte() = sha256.ComputeHash(buffer)
    Return DoesHashExist(digest)
  End Using
End Function
```

可以将前面的代码更改为直接调用静态 HashData(Byte[]) 方法。

```
public bool CheckHash(byte[] buffer)
{
    byte[] digest = SHA256.HashData(buffer);
    return DoesHashExist(digest);
}
```

```
Public Function CheckHash(buffer As Byte()) As Boolean
  Dim digest As Byte() = SHA256.HashData(buffer)
  Return DoesHashExist(digest)
End Function
```

何时禁止显示警告

可禁止显示此规则的警告。

另请参阅

- 性能规则

单文件规则

2021/11/16 •

.NET 的单文件规则。

> **NOTE**
> 在 .NET 6 之前，此类别名为 `Publish`。

在本节中

IL3000 当发布为单个文件时，避免访问程序集文件路径	当发布为单个文件时，避免访问程序集文件路径
IL3001 当发布为单个文件时，避免访问程序集文件路径	当发布为单个文件时，避免访问程序集文件路径
IL3002 当发布为单个文件时，避免调用使用“RequiresAssemblyFilesAttribute”批注的成员	当发布为单个文件时，避免调用使用“RequiresAssemblyFilesAttribute”批注的成员

IL3000：当发布为单个文件时，避免访问程序集文件路径

2021/11/16 •

■ ID	IL3000
■	SingleFile
■■■■■	非中断

原因

发布为单文件(例如将项目中的 PublishSingleFile 属性设置为 true)时，调用嵌入在单文件捆绑包内的程序集的 `Assembly.Location` 属性将始终返回空字符串。

如何解决冲突

如果应用只需要用于单文件捆绑包的包含目录，请考虑改用 `AppContext.BaseDirectory` 属性。否则，请考虑完全删除调用。

何时禁止显示警告

如果要访问的程序集肯定不在单文件捆绑包中，则可关闭此警告。如果从文件路径动态加载程序集，则可能会出现这种情况。

IL3001：当发布为单个文件时，避免访问程序集文件路径

2021/11/16 •

■ ID	IL3001
■	SingleFile
■■■■■	非中断

原因

发布为单文件(例如，通过将项目中的 PublishSingleFile 属性设置为 true)时，为单文件捆绑包内嵌入的程序集调用 `Assembly.GetFile(s)` 方法将始终引发异常。

如何解决冲突

若要将文件嵌入单文件捆绑包中的程序集，请考虑使用嵌入的资源和 `Assembly.GetManifestResourceStream` 方法。

何时禁止显示警告

如果要访问的程序集肯定不在单文件捆绑包中，则可关闭此警告。如果从文件路径动态加载程序集，则可能会出现这种情况。

IL3002：当发布为单个文件时，避免调用使用“RequiresAssemblyFilesAttribute”注释的成员。

2021/11/16 •

■ ID	IL3002
■	SingleFile
■■■■■	非中断

原因

将应用发布为单个文件(例如将项目中的 `PublishSingleFile` 属性设置为 `true`)时，调用使用 `RequiresAssemblyFilesAttribute` 属性注释的成员可能会不安全。这些调用可能不安全，因为使用此属性注释的成员要求程序集文件位于磁盘上，而嵌入单文件应用的程序集已加载到内存中。

如何解决冲突

使用“RequiresAssemblyFilesAttribute”属性注释的成员有一条消息，用于向发布为单个文件的用户提供有用的信息。请考虑根据属性的消息调整现有代码，或者删除有冲突的调用。

何时禁止显示警告

如果已根据“RequiresAssemblyFilesAttribute”属性消息中概述的建议调整现有代码，则可禁止显示警告。

可靠性规则

2021/11/16 •

支持库和应用程序可靠性(例如正确使用内存和线程)的可靠性规则。可靠性规则包括:

"	"
CA2000:丢失范围之前释放对象	由于可能发生异常事件,导致对象的终结器无法运行,因此,应显式释放对象,以避免对该对象的所有引用超出范围。
CA2002:不要锁定具有弱标识的对象	当可以跨应用程序域边界直接进行访问对象时,则认为该对象具有弱标识。对于尝试获取对具有弱标识的对象的锁的线程,该线程可能会被其他应用程序域中持有对同一对象的锁的另一线程所阻止。
CA2007:不直接等待任务	异步方法会直接等待 Task。
CA2008:不要在未传递 TaskScheduler 的情况下创建任务	任务创建或延续操作使用未指定 TaskScheduler 参数的方法重载。
CA2009:请勿对 ImmutableCollection 值调用 ToImmutableCollection	没有必要在 System.Collections.Immutable 命名空间的不可变集合上调用 `ToImmutable` 方法。
CA2011:请勿在其资源库中分配属性	属性在自身的 set 访问器中被意外赋值。
CA2012:正确使用 ValueTask	从成员调用中返回的 ValueTasks 旨在直接等待。多次尝试使用 ValueTask 或在已知完成之前直接访问其结果可能会导致异常或损坏。忽略此类 ValueTask 可能指示出现功能 Bug,还可能降低性能。
CA2013:请勿将 ReferenceEquals 与值类型结合使用	使用 System.Object.ReferenceEquals 比较值时,如果 objA 和 objB 是值类型,则在将其传递给 ReferenceEquals 方法之前将它们装箱。这意味着,即使 objA 和 objB 都表示值类型的同一个实例,ReferenceEquals 方法也会返回 false。
CA2014:请勿在循环中使用 stackalloc。	仅在当前方法调用结束时,Stackalloc 分配的堆栈空间才会释放。在循环中使用此方法可能导致无限堆栈增长,最终出现堆栈溢出的情况。
CA2015:请勿为派生自 MemoryManager<T> 的类型定义终结器	将终结器添加到派生自 MemoryManager<T> 的类型可能使内存在仍被 Span<T> 使用时得到释放。
CA2016:将 CancellationToken 参数转发到采用一个该参数的方法	将 `CancellationToken` 参数转发给方法来确保操作取消通知得到正确传播,或者在 `CancellationToken.None` 中显式传递,以指示有意不传播令牌。
CA2018:`Buffer.BlockCopy` 的 `count` 参数应指定要复制的字节数	使用 `Buffer.BlockCopy` 时,`count` 参数指定要复制的字节数。应仅对元素大小正好为一个字节的数组将 `Array.Length` 用于 `count` 参数。`byte`、`sbyte` 和 `bool` 数组具有大小为一个字节的元素。

CA2000:丢失范围之前释放对象

2021/11/16 •

■ ID	CA2000
■	可靠性
▬▬	非中断

原因

创建了 IDisposable 类型的本地对象，但该对象不会被释放，除非对对象的所有引用都超出范围。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

如果在对某个可释放对象的所有引用超出范围之前未显式释放该对象，则当垃圾回收器运行该对象的终结器时，将在某个不确定的时间释放该对象。由于可能发生异常事件，导致对象的终结器无法运行，因此应显式释放对象。

特殊情况

即使未释放对象，也不会由于以下类型的本地对象触发规则 CA2000：

- System.IO.Stream
- System.IO.StringReader
- System.IO.TextReader
- System.IO.TextWriter
- System.Resources.IResourceReader

将其中一个类型的对象传递给构造函数，然后将其分配给一个字段，表示释放所有权转移到新构造的类型。也就是说，新构造的类型现在负责释放对象。如果代码将其中一个类型的对象传递给构造函数，则即使在对对象的所有引用超出范围之前未释放该对象，也不会发生规则 CA2000 冲突。

如何解决冲突

要解决此规则的冲突，需在对对象的所有引用超出范围之前，在对象上调用 Dispose。

可使用 `using` 语句(Visual Basic 中的 `Using`)来包装实现 IDisposable 的对象。以这种方式包装的对象将自动在 `using` 块的末尾释放。但是，以下情况不应或不能使用 `using` 语句进行处理：

- 要返回可释放对象，该对象必须在 `using` 块外的 `try/finally` 块中构造。
- 请勿在 `using` 语句的构造函数中初始化可释放对象的成员。
- 如果构造函数仅由一个异常处理程序保护并嵌套在 `using` 语句的获取部分，则外部构造函数中的失败会导致始终不会关闭嵌套构造函数所创建的对象。在下面的示例中，StreamReader 构造函数中的失败会导致始终不会关闭 FileStream 对象。在这种情况下，CA2000 标记为违反规则。

```
using (StreamReader sr = new StreamReader(new FileStream("C:\myfile.txt", FileMode.Create)))
{ ... }
```

- 动态对象应使用影子对象实现 IDisposable 对象的释放模式。

何时禁止显示警告

在以下情况可禁止显示此规则发出的警告：

- 在对象上调用了一个调用 `Dispose` 的方法，例如 Close
- 引发警告的方法返回包装对象的 IDisposable 对象
- 分配方法没有释放所有权；也就是说，释放对象的责任将转移到在方法中创建并返回给调用方的另一个对象或包装器

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

可以仅为此规则、为所有规则或为此类别（可靠性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

""	""
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

相关规则

- CA2213:应释放可释放的字段

示例 1

如果要实现返回可释放对象的方法，可在没有 catch 块的情况下使用 try/finally 块来确保释放对象。通过使用 try/finally 块，允许在故障点引发异常，并确保对象已释放。

在 OpenPort1 方法中，调用打开 ISerializable 对象 SerialPort 或对 SomeMethod 的调用可能会失败。此实现会引发 CA2000 警告。

在 OpenPort2 方法中，将声明两个 SerialPort 对象并将其设置为 null：

- `tempPort`，用于测试方法操作是否成功。
- `port`，用于返回方法的值。

构造 `tempPort` 并在 `try` 块中打开，以及在相同的 `try` 块中执行任何其他所需的工作。在 `try` 块的末尾，打开的端口分配给将返回的 `port` 对象，`tempPort` 对象设置为 `null`。

`finally` 块检查 `tempPort` 的值。如果不为 null，则表示方法中的操作失败，`tempPort` 关闭以确保释放所有资源。如果方法的操作成功，则返回的端口对象将包含打开的 SerialPort 对象，如果操作失败，则值为 null。

```
public SerialPort OpenPort1(string portName)
{
   SerialPort port = new SerialPort(portName);
   port.Open();  //CA2000 fires because this might throw
   SomeMethod(); //Other method operations can fail
   return port;
}

public SerialPort OpenPort2(string portName)
{
   SerialPort tempPort = null;
   SerialPort port = null;
   try
   {
      tempPort = new SerialPort(portName);
      tempPort.Open();
      SomeMethod();
      //Add any other methods above this line
      port = tempPort;
      tempPort = null;

   }
   finally
   {
      if (tempPort != null)
      {
         tempPort.Close();
      }
   }
   return port;
}
```

```
Public Function OpenPort1(ByVal PortName As String) As SerialPort

   Dim port As New SerialPort(PortName)
   port.Open()    'CA2000 fires because this might throw
   SomeMethod()   'Other method operations can fail
   Return port

End Function

Public Function OpenPort2(ByVal PortName As String) As SerialPort

   Dim tempPort As SerialPort = Nothing
   Dim port As SerialPort = Nothing

   Try
      tempPort = New SerialPort(PortName)
      tempPort.Open()
      SomeMethod()
      'Add any other methods above this line
      port = tempPort
      tempPort = Nothing

   Finally
      If Not tempPort Is Nothing Then
         tempPort.Close()
      End If

   End Try

   Return port

End Function
```

示例 2

默认情况下，Visual Basic 编译器会检查溢出情况下的所有算术运算符。因此，任何 Visual Basic 算术运算都可能引发 OverflowException。这可能会导致 CA2000 等规则出现意外冲突。例如，以下 CreateReader1 函数会产生 CA2000 冲突，因为 Visual Basic 编译器正在为加法发出的溢出检查指令可能会引发导致 StreamReader 无法释放的异常。

要解决此问题，可在项目中禁用 Visual Basic 编译器发出溢出检查，也可修改代码，如以下 CreateReader2 函数所示。

若要禁用发出溢出检查，请在解决方案资源管理器中右键单击项目名称，然后单击“属性”。依次单击“编译”和“高级编译选项”，然后检查“不做整数溢出检查”。

```
Imports System.IO

Class CA2000
    Public Function CreateReader1(ByVal x As Integer) As StreamReader
        Dim local As New StreamReader("C:\Temp.txt")
        x += 1
        Return local
    End Function


    Public Function CreateReader2(ByVal x As Integer) As StreamReader
        Dim local As StreamReader = Nothing
        Dim localTemp As StreamReader = Nothing
        Try
            localTemp = New StreamReader("C:\Temp.txt")
            x += 1
            local = localTemp
            localTemp = Nothing
        Finally
            If (Not (localTemp Is Nothing)) Then
                localTemp.Dispose()
            End If
        End Try
        Return local
    End Function
End Class
```

另请参阅

- IDisposable
- 释放模式

CA2002:不要锁定具有弱标识的对象

2021/11/16 •

■ ID	CA2002
■	可靠性
▬▬	非中断

原因

线程尝试在具有弱标识的对象上获取锁。

规则说明

当可以跨应用程序域边界直接进行访问对象时，则认为该对象具有弱标识。对于尝试获取对具有弱标识的对象的锁的线程，该线程可能会被其他应用程序域中持有对同一对象的锁的另一线程所阻止。

以下类型具有弱标识，并由规则标记：

- String
- 值类型的数组，包括整数类型、浮点类型和 Boolean。
- MarshalByRefObject
- ExecutionEngineException
- OutOfMemoryException
- StackOverflowException
- MemberInfo
- ParameterInfo
- Thread
- this 或 Me 对象

如何解决冲突

若要解决与此规则的冲突，请使用“描述”部分中未包含的类型的对象。

何时禁止显示警告

如果锁定的对象为 `this` 或 `Me`，且 self 对象类型的可见性为专用或内部，并且不能使用任何公共引用访问该实例，可禁止显示该警告。

否则，请勿禁止显示此规则的警告。

相关规则

CA2213:应释放可释放的字段

示例

以下示例显示了一些与规则冲突的对象锁。

```
Imports System
Imports System.IO
Imports System.Reflection
Imports System.Threading

Namespace ca2002

    Class WeakIdentities

        Sub SyncLockOnWeakId1()

            SyncLock GetType(WeakIdentities)
            End SyncLock

        End Sub

        Sub SyncLockOnWeakId2()

            Dim stream As New MemoryStream()
            SyncLock stream
            End SyncLock

        End Sub

        Sub SyncLockOnWeakId3()

            SyncLock "string"
            End SyncLock

        End Sub

        Sub SyncLockOnWeakId4()

            Dim member As MemberInfo =
            Me.GetType().GetMember("SyncLockOnWeakId1")(0)
            SyncLock member
            End SyncLock

        End Sub

        Sub SyncLockOnWeakId5()

            Dim outOfMemory As New OutOfMemoryException()
            SyncLock outOfMemory
            End SyncLock

        End Sub

    End Class

End Namespace
```

```
class WeakIdentities
{
    void LockOnWeakId1()
    {
        lock (typeof(WeakIdentities)) { }
    }

    void LockOnWeakId2()
    {
        MemoryStream stream = new MemoryStream();
        lock (stream) { }
    }

    void LockOnWeakId3()
    {
        lock ("string") { }
    }

    void LockOnWeakId4()
    {
        MemberInfo member = this.GetType().GetMember("LockOnWeakId1")[0];
        lock (member) { }
    }
    void LockOnWeakId5()
    {
        OutOfMemoryException outOfMemory = new OutOfMemoryException();
        lock (outOfMemory) { }
    }
}
```

另请参阅

- Monitor
- AppDomain
- lock 语句 (C#)
- SyncLock 语句 (Visual Basic)

CA2007：不直接等待任务

2021/11/16 •

	[
■ ID	CA2007
■	可靠性
▬▬▬	非中断

原因

异步方法会直接等待 Task。

规则说明

异步方法直接等待 Task 时，延续任务通常会出现在创建任务的同一线程中，具体取决于异步上下文。此行为可能会降低性能，并且可能会导致 UI 线程发生死锁。请考虑调用 Task.ConfigureAwait(Boolean) 以表示延续任务意图。

如何解决冲突

若要解决冲突，请在等待的 Task 上调用 ConfigureAwait。可以为 `continueOnCapturedContext` 参数传递 `true` 或 `false`。

- 对任务调用 `ConfigureAwait(true)` 与未显式调用 ConfigureAwait 的行为相同。通过显式调用此方法，可让读取者知道你是有意要对原始同步上下文执行延续任务。
- 对任务调用 `ConfigureAwait(false)` 可将延续任务安排到线程池，从而避免 UI 线程上出现死锁。对于与应用无关的库，传递 `false` 是一个好的选择。

示例

下面的代码片段会生成此警告：

```
public async Task Execute()
{
    Task task = null;
    await task;
}
```

若要解决此冲突，请在等待的 Task 上调用 ConfigureAwait：

```
public async Task Execute()
{
    Task task = null;
    await task.ConfigureAwait(false);
}
```

何时禁止显示警告

此警告适用于库，在库中，可能会在任意环境中执行代码，而代码不应对环境或方法的调用方如何调用或等待作出假设。一般来说，对于代表应用程序代码（而不是库代码）的项目，可完全禁止显示此警告；事实上，在应用程序代码上运行该分析器（例如 WinForms 或 WPF 项目中的按钮单击事件处理程序）很可能会导致执行错误的操作。

如果应将延续任务安排回原始上下文，或者还没有此类上下文，都可禁止显示此警告。例如，在 WinForms 或 WPF 应用程序中的按钮单击事件处理程序中编写代码时，通常情况下，等待的延续任务应在 UI 线程上运行，因而需要将延续任务安排回原始上下文的默认行为。另举一例，在 ASP.NET Core 应用程序中编写代码时，默认情况下没有 SynchronizationContext 或 TaskScheduler，原因是 `ConfigureAwait` 不会实际更改任何行为。

抑制警告

可以通过多种方式来禁止显示代码分析警告，包括禁用项目的规则、使用预处理器指令为特定代码行禁用该规则或应用 SuppressMessageAttribute 特性。有关详细信息，请参阅如何禁止显示代码分析警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除 async void 方法
- 输出类型

可以仅为此规则、为所有规则或为此类别（可靠性）中的所有规则配置所有这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除 async void 方法

可配置是否要排除不从此规则返回值的异步方法。要排除这些类型的方法，需将以下键值对添加到项目中的 .editorconfig 文件：

```
# Package version 2.9.0 and later
dotnet_code_quality.CA2007.exclude_async_void_methods = true

# Package version 2.6.3 and earlier
dotnet_code_quality.CA2007.skip_async_void_methods = true
```

输出类型

还可以配置此规则要应用的输出程序集种类。例如，如果仅将此规则应用于生成控制台应用程序或动态链接库的代码（即不是 UI 应用），需将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CA2007.output_kind = ConsoleApplication, DynamicallyLinkedLibrary
```

另请参阅

- ConfigureAwait 常见问题解答
- 是否应使用 ConfigureAwait (false) 来等待任务？
- CA2008：不要在未传递 TaskScheduler 的情况下创建任务
- 可靠性规则

CA2008：不要在未传递 TaskScheduler 的情况下创建任务

2021/11/16 •

■ ID	CA2008
■	可靠性
■■■■■	非中断

原因

任务创建或延续操作使用未指定 TaskScheduler 参数的方法重载。

规则说明

以下 .NET 任务创建和延续方法具有允许指定或省略 TaskScheduler 实例的重载：

- System.Threading.Tasks.TaskFactory.StartNew 方法
- System.Threading.Tasks.Task.ContinueWith 方法

始终指定显式 TaskScheduler 参数以避免默认 Current 值，其行为由调用方定义并且在运行时可能会变化。Current 返回与该线程上当前运行的任何 Task 相关联的计划程序。如果没有此类任务，则返回 Default，它表示线程池。在某些情况下，使用 Current 可能会导致死锁或 UI 响应问题，因为原本打算在线程池上创建任务，但却等待返回到 UI 线程。

有关详细信息和详细示例，请参阅 .NET Framework 4.5 中的新 TaskCreationOptions 和 TaskContinuationOptions。

> **NOTE**
> VSTHRD105 - 避免使用假定 TaskScheduler.Current 是在 Microsoft.VisualStudio.Threading.Analyzers 包中实现的类似规则的方法重载。

如何解决冲突

若要解决冲突，请调用接受 TaskScheduler 并显式传入 Default 或 Current 以使意图明确的方法重载。

何时禁止显示警告

此警告主要针对库，在库中，代码可能会在任意环境中执行，并且代码不应该对环境或方法的调用方如何调用或等待作出假设。对于代表应用程序代码（而不是库代码）的项目，可禁止显示此警告。

另请参阅

- .NET Framework 4.5 中的新 TaskCreationOptions 和 TaskContinuationOptions
- VSTHRD105 - 避免使用假定 TaskScheduler.Current 的方法重载

- CA2007：不直接等待任务
- 可靠性规则

CA2009：请勿对 ImmutableCollection 值调用 ToImmutableCollection

2021/11/16 •

	[
■ ID	CA2009
■	可靠性
▬▬▬	非中断

原因

对 System.Collections.Immutable 命名空间中的不可变集合不必要地调用了 `ToImmutable` 方法。

规则说明

System.Collections.Immutable 命名空间包含用于定义不可变集合的类型。此规则分析以下不可变集合类型：

- System.Collections.Immutable.ImmutableArray<T>
- System.Collections.Immutable.ImmutableList<T>
- System.Collections.Immutable.ImmutableHashSet<T>
- System.Collections.Immutable.ImmutableSortedSet<T>
- System.Collections.Immutable.ImmutableDictionary<TKey,TValue>
- System.Collections.Immutable.ImmutableSortedDictionary<TKey,TValue>

这些类型定义了从现有 IEnumerable<T> 集合创建新的不可变集合的扩展方法。

- ImmutableArray<T> 定义 ToImmutableArray。
- ImmutableList<T> 定义 ToImmutableList。
- ImmutableHashSet<T> 定义 ToImmutableHashSet。
- ImmutableSortedSet<T> 定义 ToImmutableSortedSet。
- ImmutableDictionary<TKey,TValue> 定义 ToImmutableDictionary。
- ImmutableSortedDictionary<TKey,TValue> 定义 ToImmutableSortedDictionary。

这些扩展方法旨在将可变集合转换为不可变集合。但是，调用方可能会意外地将不可变集合作为输入传递给这些方法。这可能表示存在性能和/或功能问题。

- 性能问题：对不可变集合执行了不必要的装箱、取消装箱和/或运行时类型检查。
- 可能的功能问题：调用方假定要在可变集合上操作，而其实际拥有的是一个不可变集合。

如何解决冲突

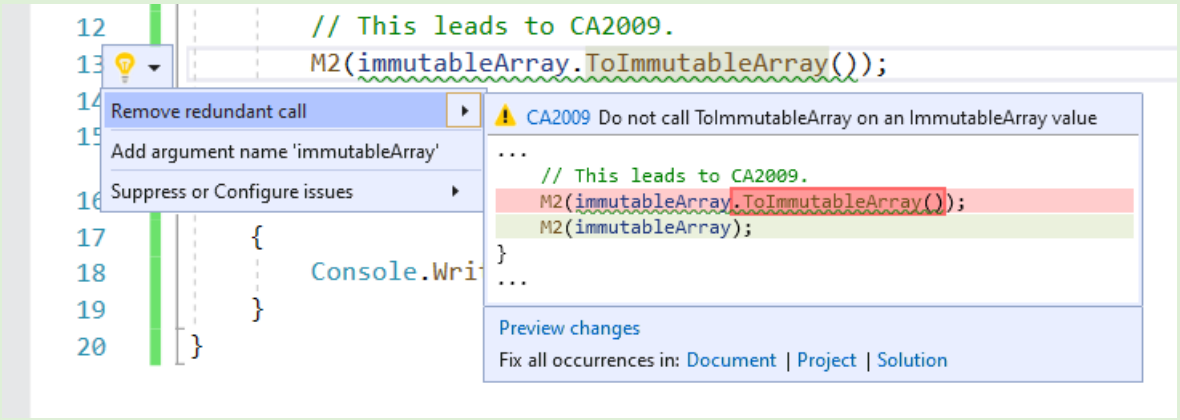
若要解决冲突，请删除对不可变集合的冗余 `ToImmutable` 调用。例如，以下两个代码片段显示了规则冲突及其解决方法：

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;

public class C
{
    public void M(IEnumerable<int> collection, ImmutableArray<int> immutableArray)
    {
        // This is fine.
        M2(collection.ToImmutableArray());

        // This leads to CA2009.
        M2(immutableArray.ToImmutableArray());
    }

    private void M2(ImmutableArray<int> immutableArray)
    {
        Console.WriteLine(immutableArray.Length);
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Collections.Immutable;

public class C
{
    public void M(IEnumerable<int> collection, ImmutableArray<int> immutableArray)
    {
        // This is fine.
        M2(collection.ToImmutableArray());

        // This is now fine.
        M2(immutableArray);
    }

    private void M2(ImmutableArray<int> immutableArray)
    {
        Console.WriteLine(immutableArray.Length);
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修补程序。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“删除冗余调用”。



何时禁止显示警告

除非你不关心不必要的不可变集合分配造成的性能影响，否则不要忽略此规则的冲突警告。

另请参阅

- 可靠性规则
- 性能规则

CA2011:请勿在其资源库中分配属性

2021/11/16 •

■ ID	CA2011
■	可靠性
▬▬▬	非中断

原因

属性在其自身的 set 访问器中被意外赋值。

规则说明

在属性的 set 访问器中将属性赋值给其自身会导致对 set 访问器的无限递归调用链。这将在运行时产生 StackOverflowException。当属性和用于存储属性值的支持字段具有相似的名称时，这种错误很常见。值意外地赋值给属性本身，而不是赋值给支持字段。

如何解决冲突

要解决冲突，请将对属性的违规赋值替换为对支持字段的赋值，或切换为使用自动属性。例如，以下代码片段显示了对此规则的违反以及进行解决的几种方法：

```
public class C
{
    // Backing field for property 'P'
    private int p;

    public int P
    {
        get
        {
            return p;
        }
        set
        {
            // CA2011: Accidentally assigned to property, instead of the backing field.
            P = value;
        }
    }
}
```

```
public class C
{
    // Backing field for property 'P'
    private int _p;

    public int P
    {
        get
        {
            return _p;
        }
        set
        {
            // Option 1: Assign to backing field and rename the backing field for clarity.
            _p = value;
        }
    }
}
```

```
public class C
{
    // Option 2: Use auto-property.
    public int P { get; set; }
}
```

何时禁止显示警告

如果确定对 set 访问器的递归调用有条件地受到保护以防止无限递归，则可以禁止显示此规则引发的冲突。

相关规则

- CA2245:请勿将属性分配给其自身

另请参阅

- 可靠性规则

CA2012:正确使用 ValueTask

2021/11/16 •

■ ID	CA2012
■	可靠性
▬▬▬	非中断

原因

从成员调用中返回的 ValueTask 实例的使用方式可能导致异常、损坏或性能不佳。

规则说明

从成员调用中返回的 ValueTask 实例旨在直接等待。多次尝试使用 ValueTask 或在已知完成之前直接访问其结果可能会导致异常或损坏。忽略此类 ValueTask 可能指示出现功能 Bug，还可能降低性能。

如何解决冲突

通常情况下，应直接等待 ValueTask，而不是将其丢弃或存储到其他位置（如局部变量或字段）。

何时禁止显示警告

对于从任意成员调用返回的 ValueTask，调用方需要假设 ValueTask 必须使用一次（例如等待）并且仅使用一次。但是，如果开发人员还控制被调用的成员并对其实现情况有全面了解，则开发人员可能知道可禁止显示警告，例如，如果返回 ValueTask 始终包装 Task 对象。

另请参阅

- 可靠性规则

CA2013:请勿将 ReferenceEquals 与值类型结合使用

2021/11/16 •

	[
■ ID	CA2013
■	可靠性
▬▬▬	非中断

原因

使用 System.Object.ReferenceEquals 方法来测试一个或多个值类型是否相等。

规则说明

使用 ReferenceEquals 比较值时，如果 objA 和 objB 是值类型，则会先对其进行装箱然后才会将其传递给 ReferenceEquals 方法。这意味着，即使 objA 和 objB 都表示同一个值类型实例，ReferenceEquals 方法也会返回 false，如下面的示例所示。

如何解决冲突

若要解决此冲突，请将其替换为更合适的相等性检查，如 `==`。

```
int int1 = 1, int2 = 1;

// Violation occurs, returns false.
Console.WriteLine(Object.ReferenceEquals(int1, int2));  // false

// Use appropriate equality operator or method instead
Console.WriteLine(int1 == int2);                        // true
Console.WriteLine(Object.Equals(int1, int2));           // true
```

何时禁止显示警告

不可忽略此规则的警告，我们建议使用更合适的相等运算符，如 `==`。

相关规则

- CA2231:重写 ValueType.Equals 时应重载相等运算符

另请参阅

- 可靠性规则

CA2014:请勿在循环中使用 stackalloc

2021/11/16 •

■ ID	CA2014
■	可靠性
▬▬	非中断

原因

在循环中使用 C# stackalloc 表达式。

规则说明

C# `stackalloc` 表达式从当前堆栈帧分配内存，并且在当前方法调用返回之前，不能释放内存。如果在循环中使用 `stackalloc`，则可能会由于耗尽堆栈内存而导致堆栈溢出。

如何解决冲突

将 `stackalloc` 表达式移动到方法中的所有循环之外。

何时禁止显示警告

如果包含的循环仅被调用有限的次数，使得在所有 `stackalloc` 操作中分配的总内存量相对较小时，可能可以禁止显示此规则的冲突警告。

另请参阅

- 可靠性规则

CA2015：请勿为派生自 MemoryManager<> 的类型定义终结器

2021/11/16 •

■ ID	CA2015
■	可靠性
■■■■■	非中断

原因

为派生自 MemoryManager<T> 的类型定义终结器

规则说明

如果发生将终结器添加到派生自 MemoryManager<T> 的类型的情况，可能表示存在 bug，因为这表明在 Span<T> 中分发的本机资源正在被清除，同时 Span<T> 可能仍在使用该资源。

> **NOTE**
> MemoryManager<T> 类适用于高级方案。大多数开发人员不需要使用它。

如何解决冲突

若要解决此冲突，请删除终结器定义。

```
class DerivedClass <T> : MemoryManager<T>
{
    public override bool Dispose(bool disposing)
    {
        if (disposing)
        {
            _handle.Dispose();
        }
    }

    ...

    // Violation occurs, remove the finalizer to fix the warning.
    ~DerivedClass() => Dispose(false);
}
```

何时禁止显示警告

如果目的是创建用于调试或验证的终结器，则可以禁止显示与此规则的冲突。

相关规则

- CA1821:移除空终结器

另请参阅

- 可靠性规则

CA2016：将 CancellationToken 参数转发到采用一个该参数的方法

2021/11/16 •

	[
▬	ForwardCancellationTokenToInvocations
■ ID	CA2016
■	可靠性
▬▬	非中断

原因

此规则查找可以接受 CancellationToken 参数但不传递任何参数的方法调用，并建议将父方法的 `CancellationToken` 转发给它们。

规则说明

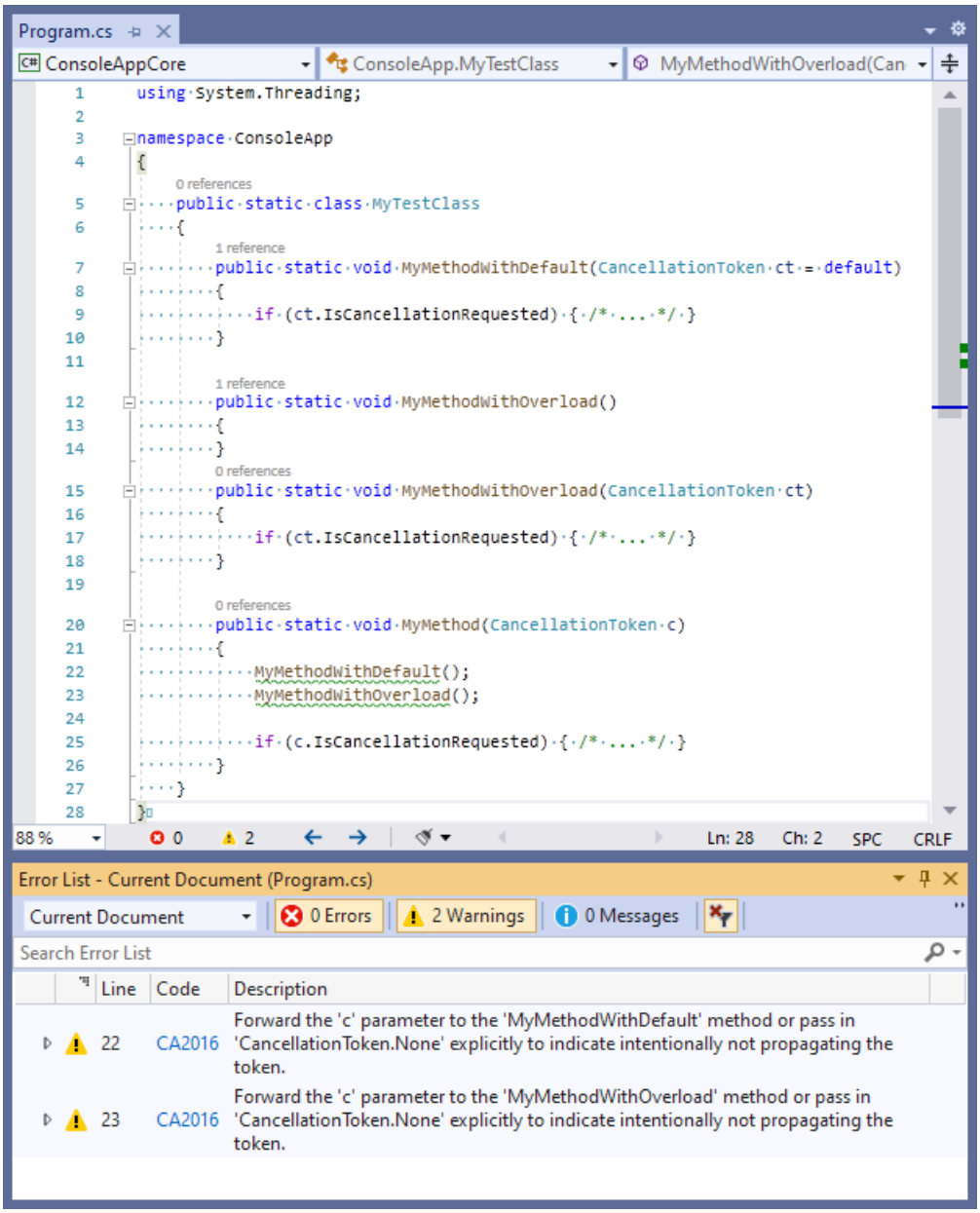
此规则分析将 `CancellationToken` 作为其最后一个参数的方法定义，然后分析其主体中调用的所有方法。如果任何方法调用可以接受 `CancellationToken` 作为最后一个参数，或者具有将 `CancellationToken` 作为最后一个参数的重载，此规则将建议改用该选项，以确保将取消通知传播到可以侦听它的所有操作。

> **NOTE**
> 在 `CancellationToken` 类型可用的所有 .NET 版本中，规则 CA2016 都可用。请参阅 CancellationToken“适用于”部分

如何解决冲突

可以手动修复，也可以选择让 Visual Studio 执行修复，方法是将鼠标悬停在方法调用旁显示的灯泡图标上，然后选择建议的更改。

下面的示例演示了两个建议的更改：



如果不关心是否将已取消的操作通知转发给下层方法调用，则可禁止显示此规则的冲突。也可以在 C# 中显式传递 `default`（在 Visual Basic 中为 `Nothing`）或 None，以禁止显示规则冲突。

此规则可以检测各种冲突。下面的示例演示了此规则可检测的情况：

示例 1

此规则建议将 `c` 参数从 `MyMethod` 转发到 `MyMethodWithDefault` 调用，因为该方法定义了一个可选的令牌参数：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

解决方法:

转发 `c` **参数**:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(c);
        }
```

如果不关心是否要将取消通知转发给**下**层调**用**，**可以**:

显**式**传递 `default`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(default);
        }
```

或显**式**传递 `CancellationToken.None`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault(CancellationToken.None);
        }
```

示例 2

此规则**建**议**将** `c` **参数从** `MyMethod` 转发**到** `MyMethodWithDefault` 调**用**，**因**为该**方法具有接受** `CancellationToken` **参数的重**载:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload()
        {
        }

        public static void MyMethodWithOverload(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

解决方法:

转发 `c` **参数**:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(c);
        }
```

如果不关心是否要将取消通知转发给下层调用，可以:

显**式**传递 `default`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(default);
        }
```

或显**式**传递 `CancellationToken.None`:

```
        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload(CancellationToken.None);
        }
```

不冲突的示例

父方法中的 `CancellationToken` **参数不在最后位置**:

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c, int lastParameter)
        {
            MyMethodWithDefault();
        }
    }
}
```

默认方法中的 `CancellationToken` 参数不在最后位置：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default, int lastParameter = 0)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

重载方法中的 `CancellationToken` 参数不在最后位置：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload(int lastParameter)
        {
        }
        public static void MyMethodWithOverload(CancellationToken ct, int lastParameter)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

父方法定义了多个 `CancellationToken` 参数：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken ct = default)
        {
        }

        public static void MyMethod(CancellationToken c1, CancellationToken c2)
        {
            MyMethodWithDefault();
        }
    }
}
```

具有默认值的方法定义了多个 `CancellationToken` 参数：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithDefault(CancellationToken c1 = default, CancellationToken c2 =
default)
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithDefault();
        }
    }
}
```

方法重载定义了多个 `CancellationToken` 参数：

```
using System.Threading;

namespace ConsoleApp
{
    public static class MyTestClass
    {
        public static void MyMethodWithOverload(CancellationToken c1, CancellationToken c2)
        {
        }

        public static void MyMethodWithOverload()
        {
        }

        public static void MyMethod(CancellationToken c)
        {
            MyMethodWithOverload();
        }
    }
}
```

CA2018：`Buffer.BlockCopy` 的 `count` 参数应指定要复制的字节数

2021/11/16 •

■ ID	CA2018
■	可靠性
■■■■■	非中断

原因

当对元素大小大于一个字节的数组上 `Buffer.BlockCopy` 的 `count` 参数使用 `Array.Length` 时，将触发此规则。

规则说明

使用 `Buffer.BlockCopy` 时，`count` 参数指定要复制的字节数。应仅对元素大小正好为一个字节的数组将 `Array.Length` 用于 `count` 参数。`byte`、`sbyte` 和 `bool` 数组具有大小为一个字节的元素。

如何解决冲突

指定要为 `count` 参数复制的字节数。

示例

冲突：

```
using System;
class Program
{
    static void Main()
    {
        int[] src = new int[] {1, 2, 3, 4};
        int[] dst = new int[] {0, 0, 0, 0};

        Buffer.BlockCopy(src, 0, dst, 0, src.Length);
    }
}
```

修复：

如果数组元素的大小大于一个字节，则可以通过将数组的长度乘以元素的大小来获得字节数。

```
using System;
class Program
{
    static void Main()
    {
        int[] src = new int[] {1, 2, 3, 4};
        int[] dst = new int[] {0, 0, 0, 0};

        Buffer.BlockCopy(src, 0, dst, 0, src.Length * sizeof(int));
    }
}
```

何时禁止显示警告

禁止显示基于此规则的警告通常是不安全的。

另请参阅

- 可靠性规则

安全规则

2021/11/16 •

安全规则可实现更安全的库和应用程序。这些规则有助于防止程序中出现安全漏洞。如果禁用其中任何规则，你应该在代码中清除标记原因，并通知开发项目的指定安全负责人。

本节内容

"	"
CA2100:检查 SQL 查询是否存在安全漏洞	一个方法使用按该方法的字符串参数生成的字符串设置 System.Data.IDbCommand.CommandText 属性。此规则假定字符串参数中包含用户输入。基于用户输入生成的 SQL 命令字符串易于受到 SQL 注入式攻击。
CA2109:检查可见的事件处理程序	检测到公共事件处理方法或受保护事件处理方法。除非绝对必要，否则不应公开事件处理方法。
CA2119:密封满足私有接口的方法	可继承的公共类型为 internal（在 Visual Basic 中为 Friend）接口提供可重写的方法实现。若要修复与此规则的冲突，请禁止方法在程序集外重写。
CA2153：避免处理损坏状态异常	损坏状态异常 (CSE) 指示进程中存在内存损坏。如果攻击者可以将攻击放置到损坏的内存区域，则捕获它们（而非允许进程崩溃）可能导致安全漏洞。
CA2300：请勿使用不安全的反序列化程序 BinaryFormatte	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2301：在未先设置 BinaryFormatter.Binder 的情况下，请不要调用 BinaryFormatter.Deserialize	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2302：在调用 BinaryFormatter.Deserialize 之前，确保设置 BinaryFormatter.Binder	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2305：请勿使用不安全的反序列化程序 LosFormatter	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2310：请勿使用不安全的反序列化程序 NetDataContractSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2311：在未先设置 NetDataContractSerializer.Binder 的情况下，请不要反序列化	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2312：确保在反序列化之前设置 NetDataContractSerializer.Binder	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。

CA2315：请勿使用不安全的反序列化程序 ObjectStateFormatter	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2321：请勿使用 SimpleTypeResolver 对 JavaScriptSerializer 进行反序列化	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2322：确保在反序列化之前没有使用 SimpleTypeResolver 初始化 JavaScriptSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2326：请勿使用 None 以外的 TypeNameHandling 值	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2327：不要使用不安全的 JsonSerializerSettings	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2328：确保 JsonSerializerSettings 是安全的	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2329：不要使用不安全的配置反序列化 JsonSerializer	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2330：在反序列化时确保 JsonSerializer 具有安全配置	反序列化不受信任的数据时，会对不安全的反序列化程序造成风险。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。
CA2350:确保 DataTable.ReadXml() 的输入受信任	对包含不受信任的输入的 DataTable 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。
CA2351:确保 DataSet.ReadXml() 的输入受信任	对包含不受信任的输入的 DataSet 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。
CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击	带有 SerializableAttribute 标记的类或结构包含 DataSet 或 DataTable 字段或属性，但不具有 GeneratedCodeAttribute。
CA2353:可序列化类型中的不安全 DataSet 或 DataTable	使用 XML 序列化特性或数据协定特性进行了标记的类或结构包含 DataSet 或 DataTable 字段或属性。
CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击	当使用序列化的 System.Runtime.Serialization.IFormatter 进行反序列化时，且强制转换的类型的对象图可能包含 DataSet 或 DataTable 时。
CA2355:反序列化对象图中的不安全 DataSet 或 DataTable	当强制转换的或指定的类型的对象图可能包含 DataSet 或 DataTable 类时，进行反序列化。

CA2356：Web 反序列化的对象图中不安全的 DataSet 或 DataTable	带有 System.Web.Services.WebMethodAttribute 或 System.ServiceModel.OperationContractAttribute 的方法具有可能引用 DataSet 或 DataTable 的参数。
CA2361：请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用	对包含不受信任的输入的 DataSet 执行反序列化时，攻击者可能通过创建恶意输入实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。
CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击	当反序列化具有 BinaryFormatter 的不受信任的输入且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可能创建执行远程代码执行攻击的恶意有效负载。
CA3001：查看 SQL 注入漏洞的代码	使用不受信任的输入和 SQL 命令时，请注意防范 SQL 注入攻击。SQL 注入攻击可以执行恶意的 SQL 命令，从而降低应用程序的安全性和完整性。
CA3002：查看 XSS 漏洞的代码	在处理来自 Web 请求的不受信任的输入时，请注意防范跨站脚本 (XSS) 攻击。XSS 攻击会将不受信任的输入注入原始 HTML 输出，使攻击者可以执行恶意脚本或恶意修改网页中的内容。
CA3003:查看文件路径注入漏洞的代码	在处理来自 Web 请求的不受信任的输入时，请谨慎使用用户控制的输入指定文件路径。
CA3004：查看信息泄露漏洞的代码	泄漏异常信息可让攻击者深入了解应用程序的内部机制，从而帮助攻击者找到其他漏洞并利用这些漏洞。
CA3006：查看进程命令注入漏洞的代码	处理不受信任的输入时，请注意防范命令注入攻击。命令注入攻击可在基础操作系统上执行恶意命令，从而降低服务器的安全和完整性。
CA3007：查看公开重定向漏洞的代码	处理不受信任的输入时，请注意防范开放重定向漏洞。攻击者可以利用开放重定向漏洞，使用你的网站提供合法 URL 的外观，但将毫不知情的访客重定向到钓鱼网页或其他恶意网页。
CA3008：查看 XPath 注入漏洞的代码	处理不受信任的输入时，请注意防范 XPath 注入攻击。使用不受信任的输入构造 XPath 查询可能会允许攻击者恶意控制查询，使其返回一个意外的结果，并可能泄漏查询的 XML 的内容。
CA3009：查看 XML 注入漏洞的代码	处理不受信任的输入时，请注意防范 XML 注入攻击。
CA3010：查看 XAML 注入漏洞的代码	处理不受信任的输入时，请注意防范 XAML 注入攻击。XAML 是一种直接表示对象实例化和执行的标记语言。这意味着 XAML 中创建的元素可以与系统资源（例如，网络访问和文件系统 IO）交互。
CA3011：查看 DLL 注入漏洞的代码	处理不受信任的输入时，请谨慎加载不受信任的代码。如果你的 Web 应用加载不受信任的代码，攻击者可能能够将恶意 DLL 注入到你的进程中，并执行恶意代码。
CA3012：查看正则表达式注入漏洞的代码	处理不受信任的输入时，请注意防范正则表达式注入攻击。攻击者可以使用正则表达式注入恶意修改正则表达式，让正则表达式匹配非预期结果，或者让正则表达式占用过多 CPU，从而形成拒绝服务攻击。

CA3061：请勿按 URL 添加架构	请勿使用不安全的“添加”方法重载，因为这可能会导致危险的外部引用。
CA3075:不安全的 DTD 处理	如果使用不安全的 DTDProcessing 实例或引用外部实体源，分析器可能会接受不受信任的输入并将敏感信息泄露给攻击者。
CA3076:不安全的 XSLT 脚本执行	如果在 .NET 应用程序中不安全地执行可扩展样式表语言转换 (XSLT)，处理器可能会解析不受信任的 URI 引用，这种引用会把敏感信息泄露给攻击者，从而导致拒绝服务和跨站点攻击。
CA3077:API 设计、XML 文档和 XML 文本读取器中的不安全处理	当设计派生自 XMLDocument 和 XMLTextReader 的 API 时，请注意 DtdProcessing。当引用或解析外部实体源或设置 XML 中的不安全值时，使用不安全的 DTDProcessing 实例可能会导致信息泄露。
CA3147:使用 ValidateAntiForgeryToken 标记谓词处理程序	设计 ASP.NET MVC 控制器时，请注意防范跨网站请求伪造攻击。跨网站请求伪造攻击可将来自经过身份验证的用户的恶意请求发送到 ASP.NET MVC 控制器。
CA5350:请勿使用弱加密算法	出于多种原因，现今使用弱加密算法和哈希函数，但不应使用它们来保证保密性或它们所保护的数据的完整性。当此规则在代码中找到 TripleDES、SHA1、或 RIPEMD160 算法时，此规则将触发。
CA5351：请勿使用已损坏的加密算法	损坏的加密算法不安全，强烈建议不要使用。当此规则在代码中找到 MD5 哈希算法，或者 DES 或 RC2 加密算法时，此规则将触发。
CA5358：请勿使用不安全的密码模式	请勿使用不安全的密码模式
CA5359:请勿禁用证书验证	证书有助于对服务器的身份进行验证。客户端应验证服务器证书，确保将请求发送到目标服务器。如果 ServerCertificateValidationCallback 始终返回 `true`，那么任何证书都将通过验证。
CA5360:在反序列化中不要调用危险的方法	不安全的反序列化是一种漏洞。当使用不受信任的数据来损害应用程序的逻辑，造成拒绝服务 (DoS) 攻击，或甚至在反序列化时任意执行代码，就会出现该漏洞。应用程序对受其控制的不受信任的数据进行反序列化时，恶意用户很可能会滥用这些反序列化功能。具体来说，就是在反序列化过程中调用危险方法。如果攻击者成功执行不安全的反序列化攻击，就能实施更多攻击，如 DoS 攻击、绕过身份验证和执行远程代码。
CA5361：不禁用较强加密的 SChannel 使用	将 `Switch.System.Net.DontEnableSchUseStrongCrypto` 设置为 `true` 会减弱传出的传输层安全性连接中使用的加密性。较弱的加密性会泄露应用程序与服务器之间通信的机密性，使攻击者更易于窃听敏感数据。
CA5362:反序列化对象图中存在潜在引用循环	反序列化不受信任的数据时，处理反序列化对象图的任何代码都需要在处理引用循环时不进入无限循环。这包括反序列化回叫中的一部分代码和在反序列化完成后处理对象图的代码。否则攻击者可能会利用带有包含引用循环的恶意数据执行拒绝服务攻击。

CA5363：请勿禁用请求验证	请求验证是 ASP.NET 中的一项功能，可检查 HTTP 请求并确定这些请求是否包含可能导致跨站点脚本编写等注入攻击的潜在危险内容。
CA5364：不使用已弃用的安全协议	传输层安全性 (TLS) 通常使用超文本传输协议安全 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。
CA5365:请勿禁用 HTTP 头检查	通过 HTTP 标头检查，可对在响应头中找到的回车符和换行符 (\r 和 \n) 进行编码。此编码有助于避免注入攻击，这些注入攻击会攻击对标头包含的不受信数据进行回显的应用程序。
CA5366:将 XmlReader 用于数据集读取 XML	使用 DataSet 读取包含不受信数据的 XML，可能会加载危险的外部引用，应使用具有安全解析程序或禁用了 DTD 处理的 XmlReader 来限制这种行为。
CA5367:请勿序列化具有 Pointer 字段的类型	此规则检查是否存在带有指针字段或属性的可序列化类。无法进行序列化的成员可能是指针，例如使用 NonSerializedAttribute 进行标记的静态成员或字段。
CA5368:针对派生自 Page 的类设置 ViewStateUserKey	设置 ViewStateUserKey 属性有助于防止对应用程序的攻击，方法是允许你为各个用户的视图状态变量分配标识符，这样攻击者就无法使用变量生成攻击。否则会出现“跨网站请求伪造”漏洞。
CA5369：将 XmlReader 用于反序列化	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用，应使用具有安全解析程序或禁用了 DTD 和 XML 内联架构处理的 XmlReader 来限制这种行为。
CA5370：将 XmlReader 用于验证读取器	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用。此危险的加载行为可使用具有安全解析程序或者禁用了 DTD 和 XML 内联架构处理的 XmlReader 来进行限制。
CA5371：将 XmlReader 用于架构读取	处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用。请使用具有安全解析程序或者禁用了 DTD 和 XML 内联架构处理的 XmlReader 对其进行限制。
CA5372：将 XmlReader 用于 XPathDocument	处理来自不受信任的数据的 XML 时可能会加载危险的外部引用，可使用具有安全解析程序或禁用了 DTD 处理的 XmlReader 对其进行限制。
CA5373：请勿使用已过时的密钥派生功能	此规则会检测对弱密钥派生方法 System.Security.Cryptography.PasswordDeriveBytes 和 `Rfc2898DeriveBytes.CryptDeriveKey` 的调用。System.Security.Cryptography.PasswordDeriveBytes 使用了弱算法 PBKDF1。
CA5374:请勿使用 XslTransform	此规则检查 System.Xml.Xsl.XslTransform 是否在代码中进行了实例化。System.Xml.Xsl.XslTransform 现已过时且不应使用。

CA5375:请勿使用帐户共享访问签名	帐户 SAS 可以委派对 blob 容器、表、队列和文件共享执行读取、写入和删除操作的访问权限，而这是服务 SAS 所不允许的。但是它不支持容器级别的策略，并且其灵活性和对授予的权限的控制力更低。一旦恶意用户获取它后，存储帐户的信息很容易泄露。
CA5376:使用 SharedAccessProtocol HttpsOnly	SAS 是无法在 HTTP 上以纯文本形式传输的敏感数据。
CA5377:使用容器级别访问策略	容器级别的访问策略可以随时修改或撤销。它具有更高的灵活性，对授予的权限的控制力更强。
CA5378：不禁用 ServicePointManagerSecurityProtocols	将 `DisableUsingServicePointManagerSecurityProtocols` 设置为 `true` 会将 Windows Communication Framework (WCF) 的传输层安全性 (TLS) 连接限制为使用 TLS 1.0。该版本的 TLS 将被弃用。
CA5379：确保密钥派生功能算法足够强	Rfc2898DeriveBytes 类默认使用 SHA1 算法。应指定在 SHA256 或更高版本的构造函数的某些重载中使用哈希算法。请注意，HashAlgorithm 属性只具有 `get` 访问器，而没有 `overriden` 修饰符。
CA5380：请勿将证书添加到根存储中	此规则会对将证书添加到“受信任的根证书颁发机构”证书存储的代码进行检测。默认情况下，“受信任的根证书颁发机构”证书存储配置有一组符合 Microsoft 根证书计划要求的公共 CA。
CA5381：请确保证书未添加到根存储中	此规则会对可能将证书添加到“受信任的根证书颁发机构”证书存储的代码进行检测。默认情况下，“受信任的根证书颁发机构”证书存储配置有一组符合 Microsoft 根证书计划要求的公共证书颁发机构 (CA)。
CA5382:在 ASP.NET Core 中使用安全 Cookie	HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用传输层安全性 (TLS) 进行传输。
CA5383:确保在 ASP.NET Core 中使用安全 Cookie	HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用传输层安全性 (TLS) 进行传输。
CA5384:不使用数字签名算法(DSA)	DSA 是一种弱非对称加密算法。
CA5385:设置具有足够密钥大小的 Rivest–Shamir–Adleman (RSA)算法	小于 2048 位的 RSA 密钥更容易受到暴力攻击。
CA5386：避免对 SecurityProtocolType 值进行硬编码	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。TLS 1.2 和 TLS 1.3 将来可能也会弃用。要确保应用程序的安全性，请避免对协议版本进行硬编码，并且至少以 .NET Framework v4.7.1 为目标。
CA5387:请勿使用迭代计数不足的弱密钥派生功能	此规则检查加密密钥是否由迭代计数小于 100,000 的 Rfc2898DeriveBytes 生成。迭代计数较高有助于缓解尝试猜测已生成的加密密钥的字典攻击。
CA5388:使用弱密钥派生功能时，请确保迭代计数足够大	此规则检查加密密钥是否由迭代计数可能小于 100,000 的 Rfc2898DeriveBytes 生成。迭代计数较高有助于缓解尝试猜测已生成的加密密钥的字典攻击。

CA5389:请勿将存档项的路径添加到目标文件系统路径中	文件路径可以是相对的，并且可能导致文件系统访问预期文件系统目标路径以外的内容，从而导致攻击者通过“布局和等待”技术恶意更改配置和执行远程代码。
CA5390:请勿编码加密密钥	要成功使用对称算法，密钥必须只有发送方和接收方知道。如果密钥是硬编码的，就容易被发现。即使使用编译的二进制文件，恶意用户也容易将其提取出来。私钥泄露后，密码文本可直接被解密并且不再受保护。
CA5391:在 ASP.NET Core MVC 控制器中使用防伪造令牌	处理 `POST`、`PUT`、`PATCH` 或 `DELETE` 请求而不验证防伪造令牌可能易受到跨网站请求伪造攻击。跨网站请求伪造攻击可将经过身份验证的用户的恶意请求发送到 ASP.NET Core MVC 控制器。
CA5392:对 P/Invoke 使用 DefaultDllImportSearchPaths 属性	默认情况下，使用 DllImportAttribute 的 P/Invoke 函数会探测大量目录，包括要加载的库的当前工作目录。这对于某些应用程序来说是一个安全隐患，会导致 DLL 劫持。
CA5393:请勿使用不安全的 DllImportSearchPath 值	默认的 DLL 搜索目录和程序集目录中可能存在恶意 DLL。或者根据应用程序运行的位置，应用程序的目录中可能存在恶意 DLL。
CA5394:请勿使用不安全的随机性	如果使用加密较弱的伪随机数生成器，攻击者可以预测将要生成的安全敏感值。
CA5395:缺少操作方法的 HttpVerb 属性	创建、编辑或以其它方式修改数据等所有操作方法都需要使用防伪特性来保护，以避免受跨网站请求伪造攻击的影响。执行 GET 操作应是没有副作用且不会修改持久数据的安全操作。
CA5396:将 HttpCookie 的 HttpOnly 设置为 true	请确保将安全敏感的 HTTP Cookie 标记为 HttpOnly，这是一个深度防御措施。这表明 Web 浏览器应禁止脚本访问 Cookie。注入恶意脚本是常见的窃取 Cookie 的方式。
CA5397：不使用已弃用的 SslProtocols 值	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。
CA5398：避免硬编码的 SslProtocols 值	传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。将来可能也会弃用 TLS 1.2 和 TLS 1.3。要确保应用程序的安全性，请避免对协议版本进行硬编码。
CA5399:绝对禁用 HttpClient 证书吊销列表检查	撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。
CA5400:确保未禁用 HttpClient 证书吊销列表检查	撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。
CA5401:不要将 CreateEncryptor 与非默认 IV 结合使用	对称加密应始终使用非可重复的初始化向量，以防止字典攻击。

𝐂	𝐂
CA5402:将 CreateEncryptor 与默认 IV 结合使用	对称加密应始终使用非可重复的初始化向量，以防止字典攻击。
CA5403：请勿硬编码证书	X509Certificate 或 X509Certificate2 构造函数的 `data` 或 `rawData` 参数是硬编码的。
CA5404：不要禁用令牌验证检查	用于控制令牌验证的 TokenValidationParameters 属性不应设置为 `false`。
CA5405：不要始终跳过委托中的令牌验证	分配给 AudienceValidator 或 LifetimeValidator 的回调始终返回 `true`。

CA2100:检查 SQL 查询是否存在安全漏洞

2021/11/16 •

■ ID	CA2100
■	安全性
修复是中断修复还是非中断修复	非中断

原因

一种方法使用按该方法的字符串参数生成的字符串设置 System.Data.IDbCommand.CommandText 属性。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

此规则假定无法在编译时确定值的任何字符串都可能包含用户输入。基于用户输入生成的 SQL 命令字符串易于受到 SQL 注入式攻击。在 SQL 注入攻击中，恶意用户会提供改变查询设计的输入，企图破坏基础数据库或对该数据库进行未经授权的访问。典型方法包括注入一个单引号或撇号(这是 SQL 文本字符串分隔符)、两个短划线(表示 SQL 注释)和一个分号(指示后跟一个新命令)。如果用户输入必须是查询的一部分，请按照以下方法之一(按有效性排列)来降低遭受攻击的风险。

- 使用存储过程。
- 使用参数化命令字符串。
- 在生成命令字符串之前，先验证用户输入的类型和内容。

下面的 .NET 类型实现 CommandText 属性，或提供使用字符串参数设置属性的构造函数。

- System.Data.Odbc.OdbcCommand 和 System.Data.Odbc.OdbcDataAdapter
- System.Data.OleDb.OleDbCommand 和 System.Data.OleDb.OleDbDataAdapter
- System.Data.OracleClient.OracleCommand 和 System.Data.OracleClient.OracleDataAdapter
- System.Data.SqlClient.SqlCommand 和 System.Data.SqlClient.SqlDataAdapter

在某些情况下，此规则可能不会在编译时确定字符串的值，即使你可以这样做。在这些情况下，当使用这些字符串作为 SQL 命令时，此规则将产生误报。以下是这种字符串的一个示例。

```
int x = 10;
string query = "SELECT TOP " + x.ToString() + " FROM Table";
```

当隐式使用 `ToString()` 时，会出现相同的情况。

```
int x = 10;
string query = String.Format("SELECT TOP {0} FROM Table", x);
```

如何解决冲突

若要解决此规则的冲突，请使用参数化查询。

何时禁止显示警告

如果命令文本不包含任何用户输入，可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

示例

下面的示例演示了违反规则的 `UnsafeQuery` 方法以及符合规则的 `SaferQuery` 方法(使用参数化命令字符串)。

```
Imports System
Imports System.Data
Imports System.Data.SqlClient

Namespace ca2100

    Public Class SqlQueries

        Function UnsafeQuery(connection As String,
         name As String, password As String) As Object

            Dim someConnection As New SqlConnection(connection)
            Dim someCommand As New SqlCommand()
            someCommand.Connection = someConnection

            someCommand.CommandText = "SELECT AccountNumber FROM Users " &
            "WHERE Username='" & name & "' AND Password='" & password & "'"

            someConnection.Open()
            Dim accountNumber As Object = someCommand.ExecuteScalar()
            someConnection.Close()
            Return accountNumber

        End Function

        Function SaferQuery(connection As String,
         name As String, password As String) As Object

            Dim someConnection As New SqlConnection(connection)
            Dim someCommand As New SqlCommand()
            someCommand.Connection = someConnection

            someCommand.Parameters.Add(
            "@username", SqlDbType.NChar).Value = name
            someCommand.Parameters.Add(
            "@password", SqlDbType.NChar).Value = password
            someCommand.CommandText = "SELECT AccountNumber FROM Users " &
            "WHERE Username=@username AND Password=@password"

            someConnection.Open()
            Dim accountNumber As Object = someCommand.ExecuteScalar()
            someConnection.Close()
            Return accountNumber

        End Function

    End Class

    Class MaliciousCode

        Shared Sub Main2100(args As String())

            Dim queries As New SqlQueries()
            queries.UnsafeQuery(args(0), "' OR 1=1 --", "[PLACEHOLDER]")
            ' Resultant query (which is always true):
            ' SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

            queries.SaferQuery(args(0), "' OR 1=1 --", "[PLACEHOLDER]")
            ' Resultant query (notice the additional single quote character):
            ' SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --'
            '                                    AND Password='[PLACEHOLDER]'
        End Sub

    End Class

End Namespace
```

```
public class SqlQueries
{
    public object UnsafeQuery(
       string connection, string name, string password)
    {
        SqlConnection someConnection = new SqlConnection(connection);
        SqlCommand someCommand = new SqlCommand();
        someCommand.Connection = someConnection;

        someCommand.CommandText = "SELECT AccountNumber FROM Users " +
           "WHERE Username='" + name +
           "' AND Password='" + password + "'";

        someConnection.Open();
        object accountNumber = someCommand.ExecuteScalar();
        someConnection.Close();
        return accountNumber;
    }

    public object SaferQuery(
       string connection, string name, string password)
    {
        SqlConnection someConnection = new SqlConnection(connection);
        SqlCommand someCommand = new SqlCommand();
        someCommand.Connection = someConnection;

        someCommand.Parameters.Add(
           "@username", SqlDbType.NChar).Value = name;
        someCommand.Parameters.Add(
           "@password", SqlDbType.NChar).Value = password;
        someCommand.CommandText = "SELECT AccountNumber FROM Users " +
           "WHERE Username=@username AND Password=@password";

        someConnection.Open();
        object accountNumber = someCommand.ExecuteScalar();
        someConnection.Close();
        return accountNumber;
    }
}

class MaliciousCode
{
    static void Main2100(string[] args)
    {
        SqlQueries queries = new SqlQueries();
        queries.UnsafeQuery(args[0], "' OR 1=1 --", "[PLACEHOLDER]");
        // Resultant query (which is always true):
        // SELECT AccountNumber FROM Users WHERE Username='' OR 1=1

        queries.SaferQuery(args[0], "' OR 1=1 --", "[PLACEHOLDER]");
        // Resultant query (notice the additional single quote character):
        // SELECT AccountNumber FROM Users WHERE Username=''' OR 1=1 --'
        //                                   AND Password='[PLACEHOLDER]'
    }
}
```

另请参阅

- 安全性概述

CA2109:检查可见的事件处理程序

2021/11/16 •

	‹
■ ID	CA2109
■	安全性
修复是中断修复还是非中断修复	重大

原因

检测到公共事件处理方法或受保护事件处理方法。

规则说明

外部可见的事件处理方法显示了一个安全问题，需要进行检查。

除非绝对必要，否则不要公开事件处理方法。只要处理程序和事件签名匹配，就可以将调用公开方法的事件处理程序（委托类型）添加到任何事件中。事件可能由任何代码引发，并且经常由高度可信的系统代码引发，以响应用户操作（例如单击某个按钮）。向事件处理方法添加安全检查不会阻止代码注册调用方法的事件处理程序。

需求无法可靠地保护由事件处理程序调用的方法。安全需求通过检查调用堆栈上的调用方，帮助防止代码受到不可信任的调用方利用。事件处理程序的方法运行时，将事件处理程序添加到事件的代码不一定会出现在调用堆栈上。因此，在调用事件处理程序方法时，调用堆栈可能仅具有高度受信任的调用方。这会使事件处理程序方法提出的需求成功。此外，调用方法时，可能会断言所需的权限。由于这些原因，只有在检查事件处理方法后才能评估不解决此规则冲突的风险。检查代码时，请考虑以下问题：

- 你的事件处理程序是否执行任何危险或可利用的操作，如断言权限或禁止非托管代码权限？
- 由于代码可随时仅通过堆栈上高度受信任的调用方运行，因此与代码之间有何安全威胁？

如何解决冲突

若要解决此规则的冲突，请检查方法并评估以下各项：

- 是否可以将事件处理方法设为非公开？
- 是否可以将所有危险功能移出事件处理程序？
- 如果提出了安全需求，是否可以通过其他方式实现？

何时禁止显示警告

仅在仔细检查安全性以确保你的代码不会构成安全威胁之后，才能禁止显示此规则的警告。

示例

下面的代码演示了一种可能被恶意代码滥用的事件处理方法。

```
public class HandleEvents
{
    // Due to the access level and signature, a malicious caller could
    // add this method to system-triggered events where all code in the call
    // stack has the demanded permission.

    // Also, the demand might be canceled by an asserted permission.

    [SecurityPermissionAttribute(SecurityAction.Demand, UnmanagedCode = true)]

    // Violates rule: ReviewVisibleEventHandlers.
    public static void SomeActionHappened(Object sender, EventArgs e)
    {
        Console.WriteLine("Do something dangerous from unmanaged code.");
    }
}
```

另请参阅

- System.Security.CodeAccessPermission.Demand
- System.EventArgs

CA2119:密封满足私有接口的方法

2021/11/16 •

	[
■ ID	CA2119
■	安全性
修复是中断修复还是非中断修复	重大

原因

可继承的公共类型为 `internal`（在 Visual Basic 中为 `Friend`）接口提供可重写的方法实现。

规则说明

接口方法具有公共可访问性，实现类型不能对其进行更改。internal 接口创建一个协定，该协定不应在定义接口的程序集的外部实现。使用 `virtual`（在 Visual Basic 中为 `Overridable`）修饰符实现 internal 接口方法的公共类型允许该方法由程序集外部的派生类型重写。如果定义程序集中的第二种类型调用该方法并需要仅限内部的协定，当在外部程序集中执行重写方法时，行为可能会受到影响。这会造成安全漏洞。

如何解决冲突

若要解决此规则的冲突，请通过以下其中一种方式阻止方法在程序集外部重写。

- 使声明类型为 `sealed`（在 Visual Basic 中为 `NotInheritable`）。
- 将声明类型的可访问性更改为 `internal`（在 Visual Basic 中为 `Friend`）。
- 删除声明类型中的所有公共构造函数。
- 在不使用 `virtual` 修饰符的情况下实现方法。
- 显式实现方法。

何时禁止显示警告

如果仔细检查后，不存在在程序集外部重写此方法时可能会被利用的安全问题，则可禁止显示此规则的警告。

示例 1

下面的示例演示了与此规则发生冲突的类型 `BaseImplementation`。

```
// Internal by default.
interface IValidate
{
    bool UserIsValidated();
}

public class BaseImplementation : IValidate
{
    public virtual bool UserIsValidated()
    {
        return false;
    }
}

public class UseBaseImplementation
{
    public void SecurityDecision(BaseImplementation someImplementation)
    {
        if (someImplementation.UserIsValidated() == true)
        {
            Console.WriteLine("Account number & balance.");
        }
        else
        {
            Console.WriteLine("Please login.");
        }
    }
}
```

```
Interface IValidate
    Function UserIsValidated() As Boolean
End Interface

Public Class BaseImplementation
    Implements IValidate

    Overridable Function UserIsValidated() As Boolean _
     Implements IValidate.UserIsValidated
        Return False
    End Function

End Class

Public Class UseBaseImplementation

    Sub SecurityDecision(someImplementation As BaseImplementation)

        If (someImplementation.UserIsValidated() = True) Then
            Console.WriteLine("Account number & balance.")
        Else
            Console.WriteLine("Please login.")
        End If

    End Sub

End Class
```

示例 2

下面的示例利用上一个示例的虚拟方法实现。

```
public class BaseImplementation
{
    public virtual bool UserIsValidated()
    {
        return false;
    }
}

public class UseBaseImplementation
{
    public void SecurityDecision(BaseImplementation someImplementation)
    {
        if (someImplementation.UserIsValidated() == true)
        {
            Console.WriteLine("Account number & balance.");
        }
        else
        {
            Console.WriteLine("Please login.");
        }
    }
}
```

```
Public Class BaseImplementation

    Overridable Function UserIsValidated() As Boolean
        Return False
    End Function

End Class

Public Class UseBaseImplementation

    Sub SecurityDecision(someImplementation As BaseImplementation)

        If (someImplementation.UserIsValidated() = True) Then
            Console.WriteLine("Account number & balance.")
        Else
            Console.WriteLine("Please login.")
        End If

    End Sub

End Class
```

另请参阅

- 接口 (C#)
- 接口 (Visual Basic)

CA2153:避免处理损坏状态异常

2021/11/16 •

■ ID	CA2153
■	安全性
修复是中断修复还是非中断修复	非中断

原因

损坏状态异常 (CSE) 指示进程中存在内存损坏。如果攻击者可以将攻击放置到损坏的内存区域，则捕获它们(而非允许进程崩溃)可能导致安全漏洞。

规则说明

CSE 指示进程状态已损坏且未被系统捕获。在损坏状态的情况下，仅当你使用 System.Runtime.ExceptionServices.HandleProcessCorruptedStateExceptionsAttribute 特性标记方法时，常规处理程序才会捕获异常。默认情况下，公共语言运行时 (CLR) 不会为 CSE 调用 catch 处理程序。

最安全的选项是允许进程发生故障而不捕获这些类型的异常。甚至日志记录代码都可以使攻击者利用内存破坏 bug。

当使用捕获所有异常的常规处理程序(例如，没有异常参数的 `catch (System.Exception e)` 或 `catch`)捕获 CSE 时，将触发此警告。

如何解决冲突

若要解决此警告，请执行以下其中一项操作：

- 请删除 HandleProcessCorruptedStateExceptionsAttribute 属性。这将恢复为默认运行时行为，其中 CSE 不会传递到 catch 处理程序。
- 删除常规 catch 处理程序，而不是捕获特定异常类型的处理程序。这可能包括假定处理程序代码可以安全处理它们的 CSE(罕见)。
- 重新引发 catch 处理程序中的 CSE，该处理程序会将异常传递给调用方，并应导致结束正在运行的进程。

何时禁止显示警告

不禁止显示此规则发出的警告。

伪代码示例

冲突

下面伪代码说明此规则检测到的模式。

```
[HandleProcessCorruptedStateExceptions]
// Method that handles CSE exceptions.
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Handle exception.
    }
}
```

解决方案 1 - 删除特性

删除 HandleProcessCorruptedStateExceptionsAttribute 特性可确保方法将不会处理损坏状态异常。

```
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Handle exception.
    }
}
```

解决方案 2 - 捕获特定异常

删除常规的 catch 处理程序并只捕获特定异常类型。

```
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (IOException e)
    {
        // Handle IOException.
    }
    catch (UnauthorizedAccessException e)
    {
        // Handle UnauthorizedAccessException.
    }
}
```

解决方案 3 - 重新引发

重新引发异常。

```
[HandleProcessCorruptedStateExceptions]
void TestMethod1()
{
    try
    {
        FileStream fileStream = new FileStream("name", FileMode.Create);
    }
    catch (Exception e)
    {
        // Rethrow the exception.
        throw;
    }
}
```

CA2300：请勿使用不安全的反序列化程序 BinaryFormatte

2021/11/16 •

	[
■ ID	CA2300
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则查找 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法调用或引用。如果只希望在 Binder 属性设置为限制类型时进行反序列化，请禁用此规则并改为启用规则 CA2301 和 CA2302。限制可以反序列化的类型可帮助缓解已知的远程代码执行攻击，但反序列化仍容易遭受拒绝服务攻击。

`BinaryFormatter` 不安全，无法确保安全。有关详细信息，请参阅 BinaryFormatter 安全指南。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`BinaryFormatter` 不安全，无法确保安全。

伪代码示例

冲突

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

相关规则

CA2301：在未先设置 BinaryFormatter.Binder 的情况下，请不要调用 BinaryFormatter.Deserialize

CA2302：在调用 BinaryFormatter.Deserialize 之前，确保设置 BinaryFormatter.Binder

CA2301：在未先设置 BinaryFormatter.Binder 的情况下，请不要调用 BinaryFormatter.Deserialize

2021/11/16 •

	ı
■ ID	CA2301
■	安全性
修复是中断修复还是非中断修复	非中断

原因

在未设置 Binder 属性的情况下调用或引用了 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

> **WARNING**
> 使用 SerializationBinder 限制类型无法阻止所有攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

当 BinaryFormatter 未设置其 Binder 时，此规则查找 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法调用或引用。无论 Binder 属性如何，如果要使用 BinaryFormatter 禁止任何反序列化，请禁用此规则和 CA2302，并启用规则 CA2300。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`BinaryFormatter` 不安全，无法确保安全。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[[[	[[
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

解决方案

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

相关规则

CA2300：请勿使用不安全的反序列化程序 BinaryFormatte

CA2302：在调用 BinaryFormatter.Deserialize 之前，确保设置 BinaryFormatter.Binder

CA2302：在调用 BinaryFormatter.Deserialize 之前，确保设置 BinaryFormatter.Binder

2021/11/16 •

■ ID	CA2302
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法，但 Binder 属性可能为 NULL。

此规则类似于 CA2301，但分析无法确定 Binder 是否一定为 NULL。

默认情况下，此规则会分析整个代码库，但这是可配置的。

> **WARNING**
> 使用 SerializationBinder 限制类型无法阻止所有攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

当 Binder 可能为 NULL 时，此规则查找 System.Runtime.Serialization.Formatters.Binary.BinaryFormatter 反序列化方法调用或引用。无论 Binder 属性如何，如果要使用 BinaryFormatter 禁止任何反序列化，请禁用此规则和 CA2301，并启用规则 CA2300。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`BinaryFormatter` 不安全，无法确保安全。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突 1

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord =
        new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = Binders.BookRecord;
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord)formatter.Deserialize(ms);    // CA2302 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = Binders.BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)    ' CA2302 violation
        End Using
    End Function
End Class
```

解决方案 1

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord =
        new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();

        // Ensure that Binder is always non-null before deserializing
        formatter.Binder = Binders.BookRecord ?? throw new Exception("Expected non-null binder");

        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord)formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()

        ' Ensure that Binder is always non-null before deserializing
        formatter.Binder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

冲突 2

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BinaryFormatter Formatter { get; set; }

    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) this.Formatter.Deserialize(ms);    // CA2302 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Formatter As BinaryFormatter

    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(Me.Formatter.Deserialize(ms), BookRecord)    ' CA2302 violation
        End Using
    End Function
End Class
```

解决方案 2

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[Serializable]
public class BookRecord
{
    public string Title { get; set; }
    public AisleLocation Location { get; set; }
}

[Serializable]
public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) formatter.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization
Imports System.Runtime.Serialization.Formatters.Binary

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<Serializable()>
Public Class BookRecord
    Public Property Title As String
    Public Property Location As AisleLocation
End Class

<Serializable()>
Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim formatter As BinaryFormatter = New BinaryFormatter()
        formatter.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(formatter.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

相关规则

- CA2300：请勿使用不安全的反序列化程序 BinaryFormatte
- CA2301：在未先设置 BinaryFormatter.Binder 的情况下，请不要调用 BinaryFormatter.Deserialize

CA2305：请勿使用不安全的反序列化程序 LosFormatter

2021/11/16 •

■ ID	CA2305
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Web.UI.LosFormatter 反序列化方法。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 System.Web.UI.LosFormatter 反序列化方法调用或引用。

`LosFormatter` 不安全，无法确保安全。有关详细信息，请参阅 BinaryFormatter 安全指南。

如何解决冲突

- 改用安全的序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

`LosFormatter` 不安全，无法确保安全。

伪代码示例

冲突

```
using System.IO;
using System.Web.UI;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        LosFormatter formatter = new LosFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Web.UI

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As LosFormatter = New LosFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

CA2310：请勿使用不安全的反序列化程序 NetDataContractSerializer

2021/11/16 •

	ι
■ ID	CA2310
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则查找 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法调用或引用。如果只希望在 Binder 属性设置为对类型进行限制时进行反序列化，请禁用此规则并改为启用规则 CA2311 和 CA2312。限制可以反序列化的类型可帮助缓解已知的远程代码执行攻击，但反序列化仍容易遭受拒绝服务攻击。

`NetDataContractSerializer` 不安全，无法确保安全。有关详细信息，请参阅 BinaryFormatter 安全指南。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`NetDataContractSerializer` 不安全，无法确保安全。

伪代码示例

冲突

```
using System.IO;
using System.Runtime.Serialization;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        return serializer.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        Return serializer.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

相关规则

CA2311：在未先设置 NetDataContractSerializer.Binder 的情况下，请不要反序列化

CA2312：确保在反序列化之前设置 NetDataContractSerializer.Binder

CA2311：在未先设置 NetDataContractSerializer.Binder 的情况下，请不要反序列化

2021/11/16 •

■ ID	CA2311
■	安全性
修复是中断修复还是非中断修复	非中断

原因

在未设置 Binder 属性的情况下调用或引用了 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

> **WARNING**
> 使用 SerializationBinder 限制类型无法阻止所有攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

当 NetDataContractSerializer 未设置其 Binder 时，此规则会查找 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法调用或引用。无论 Binder 属性如何，如果要使用 NetDataContractSerializer 禁止任何反序列化，请禁用此规则和 CA2312，并启用规则 CA2310。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`NetDataContractSerializer` 不安全，无法确保安全。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

匚匚匚	匚匚
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.IO;
using System.Runtime.Serialization;

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);    // CA2311 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)    ' CA2311 violation
        End Using
    End Function
End Class
```

解决方案

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

相关规则

CA2310：请勿使用不安全的反序列化程序 NetDataContractSerializer

CA2312：确保在反序列化之前设置 NetDataContractSerializer.Binder

CA2312：确保在反序列化之前设置 NetDataContractSerializer.Binder

2021/11/16 •

■ ID	CA2312
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法，但 Binder 属性可能为 NULL。

此规则类似于 CA2311，但无法通过分析确定 Binder 是否肯定为 NULL。

默认情况下，此规则会分析整个代码库，但这是可配置的。

> **WARNING**
> 使用 SerializationBinder 限制类型无法阻止所有攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会在 Binder 可能为 NULL 时查找 System.Runtime.Serialization.NetDataContractSerializer 反序列化方法调用或引用。无论 Binder 属性如何，如果要使用 NetDataContractSerializer 禁止任何反序列化，请禁用此规则和 CA2311，并启用规则 CA2310。

`NetDataContractSerializer` 不安全，无法确保安全。有关详细信息，请参阅 BinaryFormatter 安全指南。

如何解决冲突

- 改为使用安全序列化程序，并且不允许攻击者指定要反序列化的任意类型。有关详细信息，请参阅首选替代方案。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 此选项使代码容易遭受拒绝服务攻击，以及将来可能会发生的远程代码执行攻击。有关详细信息，请参阅 BinaryFormatter 安全指南。限制反序列化的类型。实现自定义 System.Runtime.Serialization.SerializationBinder。在反序列化之前，请在所有代码路径中将 `Binder` 属性设置为自定义 SerializationBinder 的实例。在替代的 BindToType 方法中，如果类型不是预期类型，将引发异常以停止反序列化。

何时禁止显示警告

`NetDataContractSerializer` 不安全，无法确保安全。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`[[[`	`[[`
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突 1

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = Binders.BookRecord;
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class


Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = Binders.BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)   ' CA2312 violation
        End Using
    End Function
End Class
```

解决方案 1

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class Binders
{
    public static SerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();

        // Ensure that Binder is always non-null before deserializing
        serializer.Binder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As SerializationBinder = New BookRecordSerializationBinder()
End Class


Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()

        ' Ensure that Binder is always non-null before deserializing
        serializer.Binder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

冲突 2

```
using System;
using System.IO;
using System.Runtime.Serialization;

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public string Author { get; set; }

    [DataMember]
    public int PageCount { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public NetDataContractSerializer Serializer { get; set; }

    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) this.Serializer.Deserialize(ms);    // CA2312 violation
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Author As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Serializer As NetDataContractSerializer

    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(Me.Serializer.Deserialize(ms), BookRecord)    ' CA2312 violation
        End Using
    End Function
End Class
```

解决方案 2

```
using System;
using System.IO;
using System.Runtime.Serialization;

public class BookRecordSerializationBinder : SerializationBinder
{
    public override Type BindToType(string assemblyName, string typeName)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')");

        if (typeName == "BookRecord" || typeName == "AisleLocation")
        {
            return null;
        }
        else
        {
            throw new ArgumentException("Unexpected type", nameof(typeName));
        }
    }
}

[DataContract]
public class BookRecord
{
    [DataMember]
    public string Title { get; set; }

    [DataMember]
    public string Author { get; set; }

    [DataMember]
    public int PageCount { get; set; }

    [DataMember]
    public AisleLocation Location { get; set; }
}

[DataContract]
public class AisleLocation
{
    [DataMember]
    public char Aisle { get; set; }

    [DataMember]
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(byte[] bytes)
    {
        NetDataContractSerializer serializer = new NetDataContractSerializer();
        serializer.Binder = new BookRecordSerializationBinder();
        using (MemoryStream ms = new MemoryStream(bytes))
        {
            return (BookRecord) serializer.Deserialize(ms);
        }
    }
}
```

```
Imports System
Imports System.IO
Imports System.Runtime.Serialization

Public Class BookRecordSerializationBinder
    Inherits SerializationBinder

    Public Overrides Function BindToType(assemblyName As String, typeName As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        'Console.WriteLine($"BindToType('{assemblyName}', '{typeName}')")

        If typeName = "BinaryFormatterVB.BookRecord" Or typeName = "BinaryFormatterVB.AisleLocation" Then
            Return Nothing
        Else
            Throw New ArgumentException("Unexpected type", NameOf(typeName))
        End If
    End Function
End Class

<DataContract()>
Public Class BookRecord
    <DataMember()>
    Public Property Title As String

    <DataMember()>
    Public Property Author As String

    <DataMember()>
    Public Property Location As AisleLocation
End Class

<DataContract()>
Public Class AisleLocation
    <DataMember()>
    Public Property Aisle As Char

    <DataMember()>
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(bytes As Byte()) As BookRecord
        Dim serializer As NetDataContractSerializer = New NetDataContractSerializer()
        serializer.Binder = New BookRecordSerializationBinder()
        Using ms As MemoryStream = New MemoryStream(bytes)
            Return CType(serializer.Deserialize(ms), BookRecord)
        End Using
    End Function
End Class
```

相关规则

- CA2310：请勿使用不安全的反序列化程序 NetDataContractSerializer
- CA2311：在未先设置 NetDataContractSerializer.Binder 的情况下，请不要反序列化

CA2315：请勿使用不安全的反序列化程序 ObjectStateFormatter

2021/11/16 •

	ι
■ ID	CA2315
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Web.UI.ObjectStateFormatter 反序列化方法。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 System.Web.UI.ObjectStateFormatter 反序列化方法调用或引用。

如何解决冲突

- 如果可能，请改用安全的序列化程序，并且不允许攻击者指定要反序列化的任意类型。一些更安全的序列化程序包括：
  - System.Runtime.Serialization.DataContractSerializer
  - System.Runtime.Serialization.Json.DataContractJsonSerializer
  - System.Web.Script.Serialization.JavaScriptSerializer - 请勿使用 System.Web.Script.Serialization.SimpleTypeResolver。如果必须使用类型解析程序，请将反序列化的类型限制为预期列表。
  - System.Xml.Serialization.XmlSerializer
  - Newtonsoft Json.NET - 使用 TypeNameHandling.None。如果必须为 TypeNameHandling 使用其他值，请将反序列化的类型限制为具有自定义 ISerializationBinder 的预期列表。
  - 协议缓冲区
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.IO;
using System.Web.UI;

public class ExampleClass
{
    public object MyDeserialize(byte[] bytes)
    {
        ObjectStateFormatter formatter = new ObjectStateFormatter();
        return formatter.Deserialize(new MemoryStream(bytes));
    }
}
```

```
Imports System.IO
Imports System.Web.UI

Public Class ExampleClass
    Public Function MyDeserialize(bytes As Byte()) As Object
        Dim formatter As ObjectStateFormatter = New ObjectStateFormatter()
        Return formatter.Deserialize(New MemoryStream(bytes))
    End Function
End Class
```

CA2321：请勿使用 SimpleTypeResolver 对 JavaScriptSerializer 进行反序列化

2021/11/16 •

■ ID	CA2321
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 System.Web.Script.Serialization.SimpleTypeResolver 初始化后，调用或引用了 System.Web.Script.Serialization.JavaScriptSerializer 反序列化方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则在使用 System.Web.Script.Serialization.SimpleTypeResolver 初始化JavaScriptSerializer 后查找 System.Web.Script.Serialization.JavaScriptSerializer 反序列化方法调用或引用。

如何解决冲突

- 不要使用 System.Web.Script.Serialization.SimpleTypeResolver 初始化 JavaScriptTypeResolver。
- 如果代码需要读取使用 SimpleTypeResolver 序列化的数据，可实现自定义 JavaScriptTypeResolver 将反序列化的类型限制为预期列表。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

[[[	[[
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突 1

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer(new SimpleTypeResolver());
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer(New SimpleTypeResolver())
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

解决方案 1

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer();
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer()
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

冲突 2

```
using System.Web.Script.Serialization;

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new SimpleTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New SimpleTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

解决方案 2

```
using System;
using System.Web.Script.Serialization;

public class BookRecordTypeResolver : JavaScriptTypeResolver
{
    // For compatibility with data serialized with a JavaScriptSerializer initialized with
SimpleTypeResolver.
    private static readonly SimpleTypeResolver Simple = new SimpleTypeResolver();

    public override Type ResolveType(string id)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"ResolveType('{id}')");

        if (id == typeof(BookRecord).AssemblyQualifiedName || id ==
typeof(AisleLocation).AssemblyQualifiedName)
        {
            return Simple.ResolveType(id);
        }
        else
        {
            throw new ArgumentException("Unexpected type ID", nameof(id));
        }
    }

    public override string ResolveTypeId(Type type)
    {
        return Simple.ResolveTypeId(type);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new BookRecordTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System
Imports System.Web.Script.Serialization

Public Class BookRecordTypeResolver
    Inherits JavaScriptTypeResolver

    ' For compatibility with data serialized with a JavaScriptSerializer initialized with
SimpleTypeResolver.
    Private Dim Simple As SimpleTypeResolver = New SimpleTypeResolver()

    Public Overrides Function ResolveType(id As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        ''Console.WriteLine($"ResolveType('{id}')")

        If id = GetType(BookRecord).AssemblyQualifiedName Or id =
GetType(AisleLocation).AssemblyQualifiedName Then
            Return Simple.ResolveType(id)
        Else
            Throw New ArgumentException("Unexpected type", NameOf(id))
        End If
    End Function

    Public Overrides Function ResolveTypeId(type As Type) As String
        Return Simple.ResolveTypeId(type)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New BookRecordTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

相关规则

CA2322：确保在反序列化之前没有使用 SimpleTypeResolver 初始化 JavaScriptSerializer

CA2322：确保在反序列化之前没有使用 SimpleTypeResolver 初始化 JavaScriptSerializer

2021/11/16 •

■ ID	CA2322
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 System.Web.Script.Serialization.JavaScriptSerializer 反序列化方法，但可能使用了 System.Web.Script.Serialization.SimpleTypeResolver 对 JavaScriptSerializer 进行初始化。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

当可能使用了 System.Web.Script.Serialization.SimpleTypeResolver 对 JavaScriptSerializer 进行初始化时，此规则将查找 System.Web.Script.Serialization.JavaScriptSerializer 反序列化方法调用或引用。

如何解决冲突

- 确保不使用 System.Web.Script.Serialization.SimpleTypeResolver 初始化对象 JavaScriptTypeResolver。
- 如果代码需要读取使用 SimpleTypeResolver 序列化的数据，可实现自定义 JavaScriptTypeResolver 将反序列化的类型限制为预期列表。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突 1

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public JavaScriptSerializer Serializer { get; set; }

    public T Deserialize<T>(string str)
    {
        return this.Serializer.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Property Serializer As JavaScriptSerializer

    Public Function Deserialize(Of T)(str As String) As T
        Return Me.Serializer.Deserialize(Of T)(str)
    End Function
End Class
```

解决方案 1

```
using System.Web.Script.Serialization;

public class ExampleClass
{
    public T Deserialize<T>(string str)
    {
        JavaScriptSerializer s = new JavaScriptSerializer();
        return s.Deserialize<T>(str);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class ExampleClass
    Public Function Deserialize(Of T)(str As String) As T
        Dim s As JavaScriptSerializer = New JavaScriptSerializer()
        Return s.Deserialize(Of T)(str)
    End Function
End Class
```

冲突 2

```
using System.Web.Script.Serialization;

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public JavaScriptSerializer Serializer { get; set; }

    public BookRecord DeserializeBookRecord(string s)
    {
        return this.Serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System.Web.Script.Serialization

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Property Serializer As JavaScriptSerializer

    Public Function DeserializeBookRecord(str As String) As BookRecord
        Return Me.Serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

解决方案 2

```
using System;
using System.Web.Script.Serialization;

public class BookRecordTypeResolver : JavaScriptTypeResolver
{
    // For compatibility with data serialized with a JavaScriptSerializer initialized with
SimpleTypeResolver.
    private static readonly SimpleTypeResolver Simple = new SimpleTypeResolver();

    public override Type ResolveType(string id)
    {
        // One way to discover expected types is through testing deserialization
        // of **valid** data and logging the types used.

        ////Console.WriteLine($"ResolveType('{id}')");

        if (id == typeof(BookRecord).AssemblyQualifiedName || id ==
typeof(AisleLocation).AssemblyQualifiedName)
        {
            return Simple.ResolveType(id);
        }
        else
        {
            throw new ArgumentException("Unexpected type ID", nameof(id));
        }
    }

    public override string ResolveTypeId(Type type)
    {
        return Simple.ResolveTypeId(type);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public string Author { get; set; }
    public int PageCount { get; set; }
    public AisleLocation Location { get; set; }
}

public class AisleLocation
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JavaScriptSerializer serializer = new JavaScriptSerializer(new BookRecordTypeResolver());
        return serializer.Deserialize<BookRecord>(s);
    }
}
```

```
Imports System
Imports System.Web.Script.Serialization

Public Class BookRecordTypeResolver
    Inherits JavaScriptTypeResolver

    ' For compatibility with data serialized with a JavaScriptSerializer initialized with
SimpleTypeResolver.
    Private Dim Simple As SimpleTypeResolver = New SimpleTypeResolver()

    Public Overrides Function ResolveType(id As String) As Type
        ' One way to discover expected types is through testing deserialization
        ' of **valid** data and logging the types used.

        ''Console.WriteLine($"ResolveType('{id}')")

        If id = GetType(BookRecord).AssemblyQualifiedName Or id =
GetType(AisleLocation).AssemblyQualifiedName Then
            Return Simple.ResolveType(id)
        Else
            Throw New ArgumentException("Unexpected type", NameOf(id))
        End If
    End Function

    Public Overrides Function ResolveTypeId(type As Type) As String
        Return Simple.ResolveTypeId(type)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Author As String
    Public Property Location As AisleLocation
End Class

Public Class AisleLocation
    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(str As String) As BookRecord
        Dim serializer As JavaScriptSerializer = New JavaScriptSerializer(New BookRecordTypeResolver())
        Return serializer.Deserialize(Of BookRecord)(str)
    End Function
End Class
```

相关规则

CA2321：请勿使用 SimpleTypeResolver 对 JavaScriptSerializer 进行反序列化

CA2326：请勿使用 None 以外的 TypeNameHandling 值

2021/11/16 •

	[
■ ID	CA2326
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果满足以下任一条件，则会触发此规则：

- 引用了 `None` 以外的 Newtonsoft.Json.TypeNameHandling 枚举值。
- 将表示非零值的整数值赋给 TypeNameHandling 变量。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 `None` 以外的 Newtonsoft.Json.TypeNameHandling 值。如果仅希望在指定 Newtonsoft.Json.Serialization.ISerializationBinder 来限制反序列化类型时进行反序列化，请禁用此规则并启用规则 CA2327、CA2328、CA2329 和 CA2330。

如何解决冲突

- 如果可能，请使用 TypeNameHandling 的 `None` 值。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 限制反序列化的类型。实现自定义 Newtonsoft.Json.Serialization.ISerializationBinder。在对 Json.NET 执行反序列化前，请确保在 Newtonsoft.Json.JsonSerializerSettings.SerializationBinder 属性中指定自定义 ISerializationBinder。在已重写的 Newtonsoft.Json.Serialization.ISerializationBinder.BindToType 方法中，如果类型是非预期类型，会返回 `null` 或引发异常以停止反序列化。
  - 如果限制反序列化的类型，则可能需要禁用此规则并启用规则 CA2327、CA2328、CA2329 和 CA2330。规则 CA2327、CA2328、CA2329 和 CA2330 有助于确保在使用 `None` 以外的 TypeNameHandling值时使用 ISerializationBinder。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using Newtonsoft.Json;

public class ExampleClass
{
    public JsonSerializerSettings Settings { get; }

    public ExampleClass()
    {
        Settings = new JsonSerializerSettings();
        Settings.TypeNameHandling = TypeNameHandling.All;    // CA2326 violation.
    }
}
```

```
Imports Newtonsoft.Json

Public Class ExampleClass
    Public ReadOnly Property Settings() As JsonSerializerSettings

    Public Sub New()
        Settings = New JsonSerializerSettings()
        Settings.TypeNameHandling = TypeNameHandling.All    ' CA2326 violation.
    End Sub
End Class
```

解决方案

```
using Newtonsoft.Json;

public class ExampleClass
{
    public JsonSerializerSettings Settings { get; }

    public ExampleClass()
    {
        Settings = new JsonSerializerSettings();

        // The default value of Settings.TypeNameHandling is TypeNameHandling.None.
    }
}
```

```
Imports Newtonsoft.Json

Public Class ExampleClass
    Public ReadOnly Property Settings() As JsonSerializerSettings

    Public Sub New()
        Settings = New JsonSerializerSettings()

        ' The default value of Settings.TypeNameHandling is TypeNameHandling.None.
    End Sub
End Class
```

相关规则

CA2327：不要使用不安全的 JsonSerializerSettings

CA2328：确保 JsonSerializerSettings 是安全的

CA2329：不要使用不安全的配置反序列化 JsonSerializer

CA2330：在反序列化时确保 JsonSerializer 具有安全配置

CA2327：不要使用不安全的 JsonSerializerSettings

2021/11/16 •

	▯
■ ID	CA2327
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果 Newtonsoft.Json.JsonSerializerSettings 实例的以下两个条件均为 true，则会触发此规则：

- TypeNameHandling 属性是除 `None` 以外的值。
- SerializationBinder 属性为 NULL。

必须通过以下其中一种方法来使用 JsonSerializerSettings 实例：

- 初始化为类字段或属性。
- 由方法返回。
- 传递给 JsonSerializer.Create 或 JsonSerializer.CreateDefault。
- 传递给具有 JsonSerializerSettings 参数的 JsonConvert 方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 Newtonsoft.Json.JsonSerializerSettings 实例，这些实例配置为对输入中指定的类型执行反序列化，但未配置为限制带有 Newtonsoft.Json.Serialization.ISerializationBinder 的反序列化类型。如果要禁止对完全在输入中指定的类型进行反序列化，请禁用规则 CA2327、CA2328、CA2329 和 CA2330，并启用规则 CA2326。

如何解决冲突

- 如果可能，请使用 TypeNameHandling 的 `None` 值。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 限制反序列化的类型。实现自定义 Newtonsoft.Json.Serialization.ISerializationBinder。在对 Json.NET 执行反序列化前，请确保在 Newtonsoft.Json.JsonSerializerSettings.SerializationBinder 属性中指定自定义 ISerializationBinder。在已重写的 Newtonsoft.Json.Serialization.ISerializationBinder.BindToType 方法中，如果类型是非预期类型，会返回 `null` 或引发异常以停止反序列化。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

选项	说明
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using Newtonsoft.Json;

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);    // CA2327 violation
    }
}
```

```
Imports Newtonsoft.Json

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)    ' CA2327 violation
    End Function
End Class
```

解决方案

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        settings.SerializationBinder = new BookRecordSerializationBinder();
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        settings.SerializationBinder = New BookRecordSerializationBinder()
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)
    End Function
End Class
```

相关规则

CA2326：请勿使用 None 以外的 TypeNameHandling 值

CA2328：确保 JsonSerializerSettings 是安全的

CA2329：不要使用不安全的配置反序列化 JsonSerializer

CA2330：在反序列化时确保 JsonSerializer 具有安全配置

CA2328：确保 JsonSerializerSettings 是安全的

2021/11/16 •

■ ID	CA2328
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果 Newtonsoft.Json.JsonSerializerSettings 实例的以下两个条件均可为 true，则会触发此规则：

- TypeNameHandling 属性是 `None` 以外的值。
- SerializationBinder 属性为 NULL。

必须通过以下其中一种方法来使用 JsonSerializerSettings 实例：

- 初始化为类字段或属性。
- 由方法返回。
- 传递给 JsonSerializer.Create 或 JsonSerializer.CreateDefault。
- 传递给具有 JsonSerializerSettings 参数的 JsonConvert 方法。

此规则类似于 CA2327，但在这种情况下，分析过程无法明确确定设置是否不安全。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 Newtonsoft.Json.JsonSerializerSettings 实例，这些实例可能被配置为对输入中指定的类型执行反序列化，且可能未配置为限制带有 Newtonsoft.Json.Serialization.ISerializationBinder 的反序列化类型。如果要禁止对完全在输入中指定的类型进行反序列化，请禁用规则 CA2327、CA2328、CA2329 和 CA2330，并启用规则 CA2326。

如何解决冲突

- 如果可能，请使用 TypeNameHandling 的 `None` 值。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 限制反序列化的类型。实现自定义 Newtonsoft.Json.Serialization.ISerializationBinder。在对 Json.NET 执行反序列化前，请确保在 Newtonsoft.Json.JsonSerializerSettings.SerializationBinder 属性中指定自定义 ISerializationBinder。在已重写的 Newtonsoft.Json.Serialization.ISerializationBinder.BindToType 方法中，如果类型是非预期类型，会返回 `null` 或引发异常以停止反序列化。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 你采取了如何修复冲突的某项预防措施。
- 你知道，当 TypeNameHandling 属性是 `None` 以外的值时，将始终设置 Newtonsoft.Json.JsonSerializerSettings.SerializationBinder 属性。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅类型名称(包括具有相应名称的所有类型,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式,前缀为 `T:`(可选)。

示例:

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;
        settings.SerializationBinder = Binders.BookRecord;
        return JsonConvert.DeserializeObject<BookRecord>(s, settings);    // CA2328 -- settings might be
null
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto
        settings.SerializationBinder = Binders.BookRecord
        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)   ' CA2328 -- settings might be
Nothing
    End Function
End Class
```

解决方案

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(string s)
    {
        JsonSerializerSettings settings = new JsonSerializerSettings();
        settings.TypeNameHandling = TypeNameHandling.Auto;

        // Ensure that SerializationBinder is non-null before deserializing
        settings.SerializationBinder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        return JsonConvert.DeserializeObject<BookRecord>(s, settings);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(s As String) As BookRecord
        Dim settings As JsonSerializerSettings = New JsonSerializerSettings()
        settings.TypeNameHandling = TypeNameHandling.Auto

        ' Ensure that SerializationBinder is non-null before deserializing
        settings.SerializationBinder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Return JsonConvert.DeserializeObject(Of BookRecord)(s, settings)
    End Function
End Class
```

相关规则

CA2326：请勿使用 None 以外的 TypeNameHandling 值

CA2327：不要使用不安全的 JsonSerializerSettings

CA2329：不要使用不安全的配置反序列化 JsonSerializer

CA2330：在反序列化时确保 JsonSerializer 具有安全配置

CA2329：不要使用不安全的配置反序列化 JsonSerializer

2021/11/16 •

■ ID	CA2329
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果传递到反序列化方法或初始化为字段或属性的 Newtonsoft.Json.JsonSerializer 实例满足以下两个条件，则会触发此规则：

- TypeNameHandling 属性是 `None` 以外的值。
- SerializationBinder 属性为 NULL。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 Newtonsoft.Json.JsonSerializer 实例，这些实例配置为对输入中指定的类型执行反序列化，但未配置为限制带有 Newtonsoft.Json.Serialization.ISerializationBinder 的反序列化类型。如果要禁止对完全在输入中指定的类型进行反序列化，请禁用规则 CA2327、CA2328、CA2329 和 CA2330，并启用规则 CA2326。

如何解决冲突

- 如果可能，请使用 TypeNameHandling 的 `None` 值。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 限制反序列化的类型。实现自定义 Newtonsoft.Json.Serialization.ISerializationBinder。在对 Json.NET 执行反序列化前，请确保在 Newtonsoft.Json.JsonSerializer.SerializationBinder 属性中指定自定义 ISerializationBinder。在已重写的 Newtonsoft.Json.Serialization.ISerializationBinder.BindToType 方法中，如果类型是非预期类型，会返回 `null` 或引发异常以停止反序列化。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[[[	[[
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using Newtonsoft.Json;

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        return jsonSerializer.Deserialize<BookRecord>(reader);    // CA2329 violation
    }
}
```

```
Imports Newtonsoft.Json

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        Return JsonSerializer.Deserialize(Of BookRecord)(reader)    ' CA2329 violation
    End Function
End Class
```

解决方案

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        jsonSerializer.SerializationBinder = new BookRecordSerializationBinder();
        return jsonSerializer.Deserialize<BookRecord>(reader);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        jsonSerializer.SerializationBinder = New BookRecordSerializationBinder()
        Return jsonSerializer.Deserialize(Of BookRecord)(reader)
    End Function
End Class
```

相关规则

CA2326：请勿使用 None 以外的 TypeNameHandling 值

CA2327：不要使用不安全的 JsonSerializerSettings

CA2328：确保 JsonSerializerSettings 是安全的

CA2330：在反序列化时确保 JsonSerializer 具有安全配置

CA2330：在反序列化时确保 JsonSerializer 具有安全配置

2021/11/16 •

■ ID	CA2330
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果传递到反序列化方法或初始化为字段或属性的 Newtonsoft.Json.JsonSerializer 实例满足以下两个条件，则会触发此规则：

- TypeNameHandling 属性是 `None` 以外的值。
- SerializationBinder 属性为 NULL。

此规则类似于 CA2329，但无法通过分析明确确定序列化程序的配置是否不安全。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

反序列化不受信任的数据时，不安全的反序列化程序易受攻击。攻击者可能会修改序列化数据，使其包含非预期类型，进而注入具有不良副作用的对象。例如，针对不安全反序列化程序的攻击可以在基础操作系统上执行命令，通过网络进行通信，或删除文件。

此规则会查找 Newtonsoft.Json.JsonSerializer 实例，这些实例可能被配置为对输入中指定的类型执行反序列化，且可能未配置为限制带有 Newtonsoft.Json.Serialization.ISerializationBinder 的反序列化类型。如果要禁止对完全在输入中指定的类型进行反序列化，请禁用规则 CA2327、CA2328、CA2329 和 CA2330，并启用规则 CA2326。

如何解决冲突

- 如果可能，请使用 TypeNameHandling 的 `None` 值。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并针对密钥轮换进行设计。
- 限制反序列化的类型。实现自定义 Newtonsoft.Json.Serialization.ISerializationBinder。在对 Json.NET 执行反序列化前，请确保在 Newtonsoft.Json.JsonSerializer.SerializationBinder 属性中指定自定义 ISerializationBinder。在已重写的 Newtonsoft.Json.Serialization.ISerializationBinder.BindToType 方法中，如果类型是非预期类型，会返回 `null` 或引发异常以停止反序列化。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 你已采取如何解决冲突的某项预防措施。
- 你知道，当 TypeNameHandling 属性是 `None` 以外的值时，将始终设置 Newtonsoft.Json.JsonSerializer.SerializationBinder 属性。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;
        jsonSerializer.SerializationBinder = Binders.BookRecord;
        return jsonSerializer.Deserialize<BookRecord>(reader);    // CA2330 -- SerializationBinder might be
null
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto
        jsonSerializer.SerializationBinder = Binders.BookRecord
        Return jsonSerializer.Deserialize(Of BookRecord)(reader)    ' CA2330 -- SerializationBinder might be
null
    End Function
End Class
```

解决方案

```
```

```
using System;
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;

public class BookRecordSerializationBinder : ISerializationBinder
{
    // To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    private static readonly DefaultSerializationBinder Binder = new DefaultSerializationBinder();

    public void BindToName(Type serializedType, out string assemblyName, out string typeName)
    {
        Binder.BindToName(serializedType, out assemblyName, out typeName);
    }

    public Type BindToType(string assemblyName, string typeName)
    {
        // If the type isn't expected, then stop deserialization.
        if (typeName != "BookRecord" && typeName != "AisleLocation" && typeName != "WarehouseLocation")
        {
            return null;
        }

        return Binder.BindToType(assemblyName, typeName);
    }
}

public class BookRecord
{
    public string Title { get; set; }
    public object Location { get; set; }
}

public abstract class Location
{
    public string StoreId { get; set; }
}

public class AisleLocation : Location
{
    public char Aisle { get; set; }
    public byte Shelf { get; set; }
}

public class WarehouseLocation : Location
{
    public string Bay { get; set; }
    public byte Shelf { get; set; }
}

public static class Binders
{
    public static ISerializationBinder BookRecord = new BookRecordSerializationBinder();
}

public class ExampleClass
{
    public BookRecord DeserializeBookRecord(JsonReader reader)
    {
        JsonSerializer jsonSerializer = new JsonSerializer();
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto;

        // Ensure that SerializationBinder is assigned non-null before deserializing
        jsonSerializer.SerializationBinder = Binders.BookRecord ?? throw new Exception("Expected non-null");

        return jsonSerializer.Deserialize<BookRecord>(reader);
    }
}
```

```
Imports System
Imports Newtonsoft.Json
Imports Newtonsoft.Json.Serialization

Public Class BookRecordSerializationBinder
    Implements ISerializationBinder

    ' To maintain backwards compatibility with serialized data before using an ISerializationBinder.
    Private Shared ReadOnly Property Binder As New DefaultSerializationBinder()

    Public Sub BindToName(serializedType As Type, ByRef assemblyName As String, ByRef typeName As String)
Implements ISerializationBinder.BindToName
        Binder.BindToName(serializedType, assemblyName, typeName)
    End Sub

    Public Function BindToType(assemblyName As String, typeName As String) As Type Implements
ISerializationBinder.BindToType
        ' If the type isn't expected, then stop deserialization.
        If typeName <> "BookRecord" AndAlso typeName <> "AisleLocation" AndAlso typeName <>
"WarehouseLocation" Then
            Return Nothing
        End If

        Return Binder.BindToType(assemblyName, typeName)
    End Function
End Class

Public Class BookRecord
    Public Property Title As String
    Public Property Location As Location
End Class

Public MustInherit Class Location
    Public Property StoreId As String
End Class

Public Class AisleLocation
    Inherits Location

    Public Property Aisle As Char
    Public Property Shelf As Byte
End Class

Public Class WarehouseLocation
    Inherits Location

    Public Property Bay As String
    Public Property Shelf As Byte
End Class

Public Class Binders
    Public Shared Property BookRecord As ISerializationBinder = New BookRecordSerializationBinder()
End Class

Public Class ExampleClass
    Public Function DeserializeBookRecord(reader As JsonReader) As BookRecord
        Dim jsonSerializer As JsonSerializer = New JsonSerializer()
        jsonSerializer.TypeNameHandling = TypeNameHandling.Auto

        ' Ensure SerializationBinder is non-null before deserializing
        jsonSerializer.SerializationBinder = If(Binders.BookRecord, New Exception("Expected non-null"))

        Return jsonSerializer.Deserialize(Of BookRecord)(reader)
    End Function
End Class
```

相关规则

CA2326：请勿使用 None 以外的 TypeNameHandling 值

CA2327：不要使用不安全的 JsonSerializerSettings

CA2328：确保 JsonSerializerSettings 是安全的

CA2329：不要使用不安全的配置反序列化 JsonSerializer

CA2350:确保 DataTable.ReadXml() 的输入受信任

2021/11/16 •

■ ID	CA2350
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 DataTable.ReadXml 方法。

规则说明

反序列化具有不受信任输入的 DataTable 时，攻击者可创建恶意输入来实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架而不是 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;

public class ExampleClass
{
    public DataTable MyDeserialize(string untrustedXml)
    {
        DataTable dt = new DataTable();
        dt.ReadXml(untrustedXml);
    }
}
```

相关规则

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2361:请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2351:确保 DataSet.ReadXml() 的输入受信任

2021/11/16 •

■ ID	CA2351
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 DataSet.ReadXml 方法，并且该方法不在自动生成的代码内。

此规则对自动生成的代码进行分类：

- 位于名为 `ReadXmlSerializable` 的方法中。
- `ReadXmlSerializable` 方法具有 System.Diagnostics.DebuggerNonUserCodeAttribute。
- `ReadXmlSerializable` 方法位于具有 System.ComponentModel.DesignerCategoryAttribute 的类型内。

CA2361 是类似的规则，适用于 DataSet.ReadXml 出现在自动生成的代码中的情况。

规则说明

反序列化具有不受信任输入的 DataSet 时，攻击者可创建恶意输入来实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架而不是 DataSet。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;

public class ExampleClass
{
    public DataSet MyDeserialize(string untrustedXml)
    {
        DataSet dt = new DataSet();
        dt.ReadXml(untrustedXml);
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2361:请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

2021/11/16 •

■ ID	CA2352
■	安全性
修复是中断修复还是非中断修复	非中断

原因

标记有 SerializableAttribute 的类或结构包含 DataSet 或 DataTable 字段或属性，但不具有 DesignerCategoryAttribute。

CA2362 是一个类似的规则，适用于有 DesignerCategoryAttribute 时。

规则说明

当反序列化具有 BinaryFormatter 的不受信任输入且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可能创建执行远程代码执行攻击的恶意有效负载。

此规则查找反序列化时不安全的类型。如果代码没有反序列化找到的类型，则没有反序列化漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 此规则找到的类型永远不会被直接或间接反序列化。
- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 你采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public DataSet MyDataSet { get; set; }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2361:请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

2021/11/16 •

	‹
■ ID	CA2353
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 XML 序列化特性或数据协定特性进行了标记的类或结构包含 DataSet 或 DataTable 字段或属性。

XML 序列化特性包括：

- XmlAnyAttributeAttribute
- XmlAnyElementAttribute
- XmlArrayAttribute
- XmlArrayItemAttribute
- XmlChoiceIdentifierAttribute
- XmlElementAttribute
- XmlEnumAttribute
- XmlIgnoreAttribute
- XmlIncludeAttribute
- XmlRootAttribute
- XmlTextAttribute
- XmlTypeAttribute

数据协定序列化特性包括：

- DataContractAttribute
- DataMemberAttribute
- IgnoreDataMemberAttribute
- KnownTypeAttribute

规则说明

反序列化具有不受信任的输入，并且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可创建恶意有效负载来执行拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

此规则会查找反序列化时不安全的类型。如果代码没有反序列化找到的类型，则没有反序列化漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 此规则找到的类型永远不会被直接或间接反序列化。
- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 你采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;
using System.Runtime.Serialization;

[XmlRoot]
public class MyClass
{
    public DataSet MyDataSet { get; set; }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356：Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2361：请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

2021/11/16 •

■ ID	CA2354
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用序列化的 System.Runtime.Serialization.IFormatter 进行反序列化时，强制转换的类型的对象图可能包含 DataSet 或 DataTable。

此规则使用不同的方法来实现类似的规则 CA2352：可序列化类型中不安全的 DataSet 或 DataTable 可能容易受到远程代码执行攻击。

规则说明

反序列化具有 BinaryFormatter 的不受信任的输入，并且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可创建恶意有效负载来执行远程代码执行攻击。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public MyOtherClass OtherClass { get; set; }
}

[Serializable]
public class MyOtherClass
{
    private DataSet myDataSet;
}

public class ExampleClass
{
    public MyClass Deserialize(Stream stream)
    {
        BinaryFormatter bf = new BinaryFormatter();
        return (MyClass) bf.Deserialize(stream);
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2361:请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

2021/11/16 •

■ ID	CA2355
■	安全性
修复是中断修复还是非中断修复	非中断

原因

当强制转换的或指定的类型的对象图可能包含 DataSet 或 DataTable 类时，进行反序列化。

此规则使用不同的方法来实现类似的规则 CA2353：可序列化类型中不安全的 DataSet 或 DataTable。

当以下情况发生时，将评估强制转换或指定的类型：

- 初始化 DataContractSerializer 对象
- 初始化 DataContractJsonSerializer 对象
- 初始化 XmlSerializer 对象
- 调用 JavaScriptSerializer.Deserialize
- 调用 JavaScriptSerializer.DeserializeObject
- 调用 XmlSerializer.FromTypes
- 调用 Newtonsoft Json.NET JsonSerializer.Deserialize
- 调用 Newtonsoft Json.NET JsonConvert.DeserializeObject

规则说明

当反序列化具有 BinaryFormatter 的不受信任的输入且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可创建恶意有效负载来执行拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;
using System.IO;
using System.Runtime.Serialization;

[Serializable]
public class MyClass
{
    public MyOtherClass OtherClass { get; set; }
}

[Serializable]
public class MyOtherClass
{
    private DataSet myDataSet;
}

public class ExampleClass
{
    public MyClass Deserialize(Stream stream)
    {
        BinaryFormatter bf = new BinaryFormatter();
        return (MyClass) bf.Deserialize(stream);
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2356:Web 反序列化对象图中不安全的 DataSet 或 DataTable

CA2361:请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable 类型

2021/11/16 •

	ι
■ ID	CA2356
■	安全性
修复是中断修复还是非中断修复	非中断

原因

带有 System.Web.Services.WebMethodAttribute 或 System.ServiceModel.OperationContractAttribute 的方法具有可能引用 DataSet 或 DataTable 的参数。

此规则使用不同的方法来实现类似的规则 CA2355：Web 反序列化对象图中的不安全 DataSet 或 DataTable 类型，并将发现不同的警告。

规则说明

反序列化具有不受信任的输入，并且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可创建恶意有效负载来执行拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System;
using System.Data;
using System.Web.Services;

[WebService(Namespace = "http://contoso.example.com/")]
public class MyService : WebService
{
    [WebMethod]
    public string MyWebMethod(DataTable dataTable)
    {
        return null;
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2361：确保 DataSet.ReadXml() 的输入受信任

CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2361：请确保包含 DataSet.ReadXml() 的自动生成的类没有与不受信任的数据一起使用

2021/11/16 •

■ ID	CA2361
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用或引用了 DataSet.ReadXml 方法，且该方法位于自动生成的代码内。

此规则对自动生成的代码进行分类：

- 位于名为 `ReadXmlSerializable` 的方法中。
- `ReadXmlSerializable` 方法具有 System.Diagnostics.DebuggerNonUserCodeAttribute。
- `ReadXmlSerializable` 方法位于具有 System.ComponentModel.DesignerCategoryAttribute 的类型内。

CA2351 是类似的规则，适用于 DataSet.ReadXml 出现在非自动生成的代码中时。

规则说明

反序列化具有不受信任输入的 DataSet 时，攻击者可创建恶意输入来实施拒绝服务攻击。有可能存在未知的远程代码执行漏洞。

此规则类似于 CA2351，但适用于 GUI 应用程序内数据的内存中表示形式的自动生成的代码。通常，这些自动生成的类不会从不受信任的输入中进行反序列化。应用程序的使用可能会有差异。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架而不是 DataSet。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入受到信任。考虑到应用程序的信任边界和数据流可能会随时间发生变化。
- 已采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
namespace ExampleNamespace
{
    /// <summary>
    ///Represents a strongly typed in-memory cache of data.
    ///</summary>
    [global::System.Serializable()]
    [global::System.ComponentModel.DesignerCategoryAttribute("code")]
    [global::System.ComponentModel.ToolboxItem(true)]
    [global::System.Xml.Serialization.XmlSchemaProviderAttribute("GetTypedDataSetSchema")]
    [global::System.Xml.Serialization.XmlRootAttribute("Package")]
    [global::System.ComponentModel.Design.HelpKeywordAttribute("vs.data.DataSet")]
    public partial class Something : global::System.Data.DataSet {

        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.CodeDom.Compiler.GeneratedCodeAttribute("System.Data.Design.TypedDataSetGenerator",
"4.0.0.0")]
        protected override void ReadXmlSerializable(global::System.Xml.XmlReader reader) {
            if ((this.DetermineSchemaSerializationMode(reader) ==
global::System.Data.SchemaSerializationMode.IncludeSchema)) {
                this.Reset();
                global::System.Data.DataSet ds = new global::System.Data.DataSet();
                ds.ReadXml(reader);
                if ((ds.Tables["Something"] != null)) {
                    base.Tables.Add(new SomethingTable(ds.Tables["Something"]));
                }
                this.DataSetName = ds.DataSetName;
                this.Prefix = ds.Prefix;
                this.Namespace = ds.Namespace;
                this.Locale = ds.Locale;
                this.CaseSensitive = ds.CaseSensitive;
                this.EnforceConstraints = ds.EnforceConstraints;
                this.Merge(ds, false, global::System.Data.MissingSchemaAction.Add);
                this.InitVars();
            }
            else {
                this.ReadXml(reader);
                this.InitVars();
            }
        }
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356：Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA2362：自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

2021/11/16 •

■ ID	CA2362
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 SerializableAttribute 标记的类或结构包含 DataSet 或 DataTable 字段或属性，但不具有 DesignerCategoryAttribute。

CA2352 是一个类似的规则，适用于没有 DesignerCategoryAttribute 的情况。

规则说明

当反序列化具有 BinaryFormatter 的不受信任输入且反序列化的对象图包含 DataSet 或 DataTable 时，攻击者可能创建执行远程代码执行攻击的恶意有效负载。

此规则类似于 CA2352，但适用于 GUI 应用程序内数据的内存中表示形式的自动生成的代码。通常，这些自动生成的类不会从不受信任的输入中进行反序列化。应用程序的使用可能会有差异。

此规则查找反序列化时不安全的类型。如果代码没有反序列化找到的类型，则没有反序列化漏洞。

有关详细信息，请参阅 DataSet 和 DataTable 安全指南。

如何解决冲突

- 如果可能，请使用实体框架，而不是 DataSet 和 DataTable。
- 使序列化的数据免被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 此规则找到的类型永远不会被直接或间接反序列化。
- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 你采取了如何修复冲突的某项预防措施。

伪代码示例

冲突

```
using System.Data;
using System.Xml.Serialization;

namespace ExampleNamespace
{
    [global::System.CodeDom.Compiler.GeneratedCode(""System.Data.Design.TypedDataSetGenerator"",
""2.0.0.0"")]
    [global::System.Serializable()]
    [global::System.ComponentModel.DesignerCategoryAttribute(""code"")]
    [global::System.ComponentModel.ToolboxItem(true)]
    [global::System.Xml.Serialization.XmlSchemaProviderAttribute(""GetTypedDataSetSchema"")]
    [global::System.Xml.Serialization.XmlRootAttribute(""Package"")]
    [global::System.ComponentModel.Design.HelpKeywordAttribute(""vs.data.DataSet"")]
    public class ExampleClass : global::System.Data.DataSet {
        private DataTable table;
    }
}
```

相关规则

CA2350:确保 DataTable.ReadXml() 的输入受信任

CA2351:确保 DataSet.ReadXml() 的输入受信任

CA2352:可序列化类型中的不安全 DataSet 或 DataTable 容易受到远程代码执行攻击

CA2353:可序列化类型中的不安全 DataSet 或 DataTable

CA2354:反序列化对象图中的不安全 DataSet 或 DataTable 可能容易受到远程代码执行攻击

CA2355:反序列化对象图中的不安全 DataSet 或 DataTable

CA2356:Web 反序列化对象图中的不安全 DataSet 或 DataTable

CA2362:自动生成的可序列化类型中不安全的数据集或数据表易受远程代码执行攻击

CA3001：查看 SQL 注入漏洞的代码

2021/11/16 •

■ ID	CA3001
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入进入 SQL 命令文本。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

使用不受信任的输入和 SQL 命令时，请注意防范 SQL 注入攻击。SQL 注入攻击可以执行恶意的 SQL 命令，从而降低应用程序的安全性和完整性。典型的技术包括使用单引号或撇号分隔文本字符串，在注释中使用两个短划线，以及在语句末尾使用分号。有关详细信息，请参阅 SQL Injection。

此规则试图查找 HTTP 请求中要进入 SQL 命令文本的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个执行 SQL 命令的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

通过将不受信任的输入包含在参数中，使用参数化的 SQL 命令或存储过程。

何时禁止显示警告

如果你确定输入始终针对已知安全的一组字符进行验证，则禁止显示此规则的警告是安全的。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "SELECT ProductId FROM Products WHERE ProductName = '" + name + "'",
                    CommandType = CommandType.Text,
                };

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText = "SELECT ProductId FROM
Products WHERE ProductName = '" + name + "'",
                                                                    .CommandType = CommandType.Text}
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

参数化解决方案

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "SELECT ProductId FROM Products WHERE ProductName = @productName",
                    CommandType = CommandType.Text,
                };

                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name;

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText = "SELECT ProductId FROM
Products WHERE ProductName = @productName",
                                                                    .CommandType = CommandType.Text}
                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

存储过程解决方案

```
using System;
using System.Data;
using System.Data.SqlClient;

namespace TestNamespace
{
    public partial class WebForm : System.Web.UI.Page
    {
        public static string ConnectionString { get; set; }

        protected void Page_Load(object sender, EventArgs e)
        {
            string name = Request.Form["product_name"];
            using (SqlConnection connection = new SqlConnection(ConnectionString))
            {
                SqlCommand sqlCommand = new SqlCommand()
                {
                    CommandText = "sp_GetProductIdFromName",
                    CommandType = CommandType.StoredProcedure,
                };

                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name;

                SqlDataReader reader = sqlCommand.ExecuteReader();
            }
        }
    }
}
```

```
Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports System.Linq

Namespace VulnerableWebApp
    Partial Public Class WebForm
        Inherits System.Web.UI.Page

        Public Property ConnectionString As String

        Protected Sub Page_Load(sender As Object, e As EventArgs)
            Dim name As String = Me.Request.Form("product_name")
            Using connection As SqlConnection = New SqlConnection(ConnectionString)
                Dim sqlCommand As SqlCommand = New SqlCommand With {.CommandText =
"sp_GetProductIdFromName",
                                                                    .CommandType =
CommandType.StoredProcedure}
                sqlCommand.Parameters.Add("@productName", SqlDbType.NVarChar, 128).Value = name
                Dim reader As SqlDataReader = sqlCommand.ExecuteReader()
            End Using
        End Sub
    End Class
End Namespace
```

CA3002：查看 XSS 漏洞的代码

2021/11/16 •

■ ID	CA3002
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问原始 HTML 输出。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

在处理来自 Web 请求的不受信任的输入时，请注意防范跨站脚本 (XSS) 攻击。XSS 攻击会将不受信任的输入注入原始 HTML 输出，使攻击者可以执行恶意脚本或恶意修改网页中的内容。一个典型的技术是将包含恶意代码的 `<script>` 元素放入输入中。有关详细信息，请参阅 OWASP 的 XSS。

此规则试图查找 HTTP 请求中要访问原始 HTML 输出的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会输出原始 HTML 的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

- 不要输出原始 HTML，而是使用方法或属性先对输入执行 HTML 编码。
- 先对不受信任的数据执行 HTML 编码，然后再输出原始 HTML。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 你确定输入已针对不包含 HTML 的一组已知安全的字符经过验证。
- 你确定已通过此规则检测不到的方式对数据执行 HTML 编码。

> **NOTE**
> 对于为输入执行 HTML 编码的某些方法或属性，此规则可能会报告误报。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        Response.Write("<HTML>" + input + "</HTML>");
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Me.Request.Form("in")
        Me.Response.Write("<HTML>" + input + "</HTML>")
    End Sub
End Class
```

解决方案

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];

        // Example usage of System.Web.HttpServerUtility.HtmlEncode().
        Response.Write("<HTML>" + Server.HtmlEncode(input) + "</HTML>");
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Me.Request.Form("in")

        ' Example usage of System.Web.HttpServerUtility.HtmlEncode().
        Me.Response.Write("<HTML>" + Me.Server.HtmlEncode(input) + "</HTML>")
    End Sub
End Class
```

CA3003:查看文件路径注入漏洞的代码

2021/11/16 •

	‖
■ ID	CA3003
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问文件操作的路径。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

在处理来自 Web 请求的不受信任的输入时，请谨慎使用用户控制的输入指定文件路径。攻击者可能能够读取非预期文件，从而导致敏感数据出现信息泄漏。或者，攻击者可能能够写入非预期文件，从而导致在未经授权的情况下修改敏感数据，或者降低服务器的安全性。常见的攻击者技术是使用路径遍历访问预期目录之外的文件。

此规则试图查找 HTTP 请求中要访问文件操作中路径的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会写入某个文件的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

- 尽可能将基于用户输入的文件路径限制在显式已知安全列表的范围内。例如，如果应用程序只需访问“red.txt”、“green.txt”或“blue.txt”，则只允许这些值。
- 检查是否存在不受信任的文件名，并验证名称格式是否正确。
- 指定路径时使用完整路径名称。
- 避免潜在的危险构造，如路径环境变量。
- 如果用户提交短名称，则只接受长文件名并验证长名称。
- 将最终用户输入限制在有效字符范围内。
- 拒绝超出 MAX_PATH 长度的名称。
- 按字面处理文件名，不执行解释。
- 确定文件名是否表示文件或设备。

何时禁止显示警告

如果你已按照上一部分中所述验证输入，则可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

``	``
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.IO;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string userInput = Request.Params["UserInput"];
        // Assume the following directory structure:
        //   wwwroot\currentWebDirectory\user1.txt
        //   wwwroot\currentWebDirectory\user2.txt
        //   wwwroot\secret\allsecrets.txt
        // There is nothing wrong if the user inputs:
        //   user1.txt
        // However, if the user input is:
        //   ..\secret\allsecrets.txt
        // Then an attacker can now see all the secrets.

        // Avoid this:
        using (File.Open(userInput, FileMode.Open))
        {
            // Read a file with the name supplied by user
            // Input through request's query string and display
            // The content to the webpage.
        }
    }
}
```

```
Imports System
Imports System.IO

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim userInput As String = Me.Request.Params("UserInput")
        ' Assume the following directory structure:
        '   wwwroot\currentWebDirectory\user1.txt
        '   wwwroot\currentWebDirectory\user2.txt
        '   wwwroot\secret\allsecrets.txt
        ' There is nothing wrong if the user inputs:
        '   user1.txt
        ' However, if the user input is:
        '   ..\secret\allsecrets.txt
        ' Then an attacker can now see all the secrets.

        ' Avoid this:
        Using File.Open(userInput, FileMode.Open)
            ' Read a file with the name supplied by user
            ' Input through request's query string and display
            ' The content to the webpage.
        End Using
    End Sub
End Class
```

CA3004：查看信息泄露漏洞的代码

2021/11/16 •

■ ID	CA3004
■	安全性
修复是中断修复还是非中断修复	非中断

原因

异常消息、堆栈跟踪或字符串表示形式访问 Web 输出。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

泄漏异常信息可让攻击者深入了解应用程序的内部机制，从而帮助攻击者找到其他漏洞并利用这些漏洞。

此规则试图查找输出到 HTTP 响应的异常消息、堆栈跟踪或字符串表示形式。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集捕获一个异常，然后将其传递给会输出该异常的另一个程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

不要将异常信息输出到 HTTP 响应。相反，提供一个一般的错误信息。有关详细信息，请参阅 OWASP 的"以不当方式处理错误"页面。

何时禁止显示警告

如果你确定 Web 输出在应用程序的信任边界内并且从未在外部公开，则可以禁止显示此警告。这种情况很罕见。请注意，应用程序的信任边界和数据流可能会随时间发生变化。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs eventArgs)
    {
        try
        {
            object o = null;
            o.ToString();
        }
        catch (Exception e)
        {
            this.Response.Write(e.ToString());
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Try
            Dim o As Object = Nothing
            o.ToString()
        Catch e As Exception
            Me.Response.Write(e.ToString())
        End Try
    End Sub
End Class
```

解决方案

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs eventArgs)
    {
        try
        {
            object o = null;
            o.ToString();
        }
        catch (Exception e)
        {
            this.Response.Write("An error occurred. Please try again later.");
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Try
            Dim o As Object = Nothing
            o.ToString()
        Catch e As Exception
            Me.Response.Write("An error occurred. Please try again later.")
        End Try
    End Sub
End Class
```

CA3005：查看 LDAP 注入漏洞的代码

2021/11/16 •

■ ID	CA3005
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问 LDAP 语句。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

使用不受信任的输入时，请注意防范轻型目录访问协议 (LDAP) 注入攻击。攻击者可能会对信息目录运行恶意 LDAP 语句。使用用户输入构造动态 LDAP 语句来访问目录服务的应用程序尤其容易受到攻击。

此规则试图查找 HTTP 请求中要访问 LDAP 语句的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个执行 LDAP 语句的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

对于用户控制的 LDAP 语句部分，请考虑：

- 仅允许使用包含非特殊字符的安全列表。
- 不允许使用特殊字符
- 对特殊字符执行转义。

有关更多指导，请参阅 OWASP 的 LDAP 注入防护速查表。

何时禁止显示警告

如果你确定输入已经过验证或已经过转义变得安全，就可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.DirectoryServices;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string userName = Request.Params["user"];
        string filter = "(uid=" + userName + ")";  // searching for the user entry

        // In this example, if we send the * character in the user parameter which will
        // result in the filter variable in the code to be initialized with (uid=*).
        // The resulting LDAP statement will make the server return any object that
        // contains a uid attribute.
        DirectorySearcher searcher = new DirectorySearcher(filter);
        SearchResultCollection results = searcher.FindAll();

        // Iterate through each SearchResult in the SearchResultCollection.
        foreach (SearchResult searchResult in results)
        {
            // ...
        }
    }
}
```

```
Imports System
Imports System.DirectoryServices

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(send As Object, e As EventArgs)
        Dim userName As String = Me.Request.Params(""user"")
        Dim filter As String = ""(uid="" + userName + "")""    ' searching for the user entry

        ' In this example, if we send the * character in the user parameter which will
        ' result in the filter variable in the code to be initialized with (uid=*).
        ' The resulting LDAP statement will make the server return any object that
        ' contains a uid attribute.
        Dim searcher As DirectorySearcher = new DirectorySearcher(filter)
        Dim results As SearchResultCollection = searcher.FindAll()

        ' Iterate through each SearchResult in the SearchResultCollection.
        For Each searchResult As SearchResult in results
            ' ...
        Next searchResult
    End Sub
End Class
```

CA3006：查看进程命令注入漏洞的代码

2021/11/16 •

■ ID	CA3006
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问进程命令。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范命令注入攻击。命令注入攻击可在基础操作系统上执行恶意命令，从而降低服务器的安全和完整性。

此规则试图查找 HTTP 请求中要访问进程命令的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会启动进程的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

- 尽可能避免启动基于用户输入的进程。
- 根据已知安全的一组字符和长度验证输入。

何时禁止显示警告

如果你确定输入已经过验证或已经过转义变得安全，则禁止显示此警告是安全的。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Diagnostics;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        Process p = Process.Start(input);
    }
}
```

```
Imports System
Imports System.Diagnostics

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs as EventArgs)
        Dim input As String = Me.Request.Form("in")
        Dim p As Process = Process.Start(input)
    End Sub
End Class
```

CA3007：查看公开重定向漏洞的代码

2021/11/16 •

	▮
■ ID	CA3007
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问 HTTP 响应重定向。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范开放重定向漏洞。攻击者可以利用开放重定向漏洞，使用你的网站提供合法 URL 的外观，但将毫不知情的访客重定向到钓鱼网页或其他恶意网页。

此规则试图查找 HTTP 请求中要访问 HTTP 重定向 URL 的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个提供 HTTP 重定向响应的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

修复开放重定向漏洞的方法包括：

- 不允许用户启动重定向。
- 不允许用户在重定向方案中指定 URL 的任何部分。
- 将重定向限制在预定义的 URL"允许列表"范围之内。
- 验证重定向 URL。
- 在适当的情况下，考虑在用户从你的网站进行重定向时使用免责声明页面。

何时禁止显示警告

如果你确定已经验证了输入，并将其限制在预期 URL 范围内，则可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[[[	[[
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["url"];
        this.Response.Redirect(input);
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Dim input As String = Me.Request.Form("url")
        Me.Response.Redirect(input)
    End Sub
End Class
```

解决方案

```
using System;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        if (String.IsNullOrWhiteSpace(input))
        {
            this.Response.Redirect("https://example.org/login.html");
        }
    }
}
```

```
Imports System

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, eventArgs As EventArgs)
        Dim input As String = Me.Request.Form("in")
        If String.IsNullOrWhiteSpace(input) Then
            Me.Response.Redirect("https://example.org/login.html")
        End If
    End Sub
End Class
```

CA3008：查看 XPath 注入漏洞的代码

2021/11/16 •

■ ID	CA3008
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问 XPath 查询。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范 XPath 注入攻击。使用不受信任的输入构造 XPath 查询可能会允许攻击者恶意控制查询，使其返回一个意外的结果，并可能泄漏查询的 XML 的内容。

此规则试图查找 HTTP 请求中要访问 XPath 表达式的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个执行 XPath 查询的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

修复 XPath 注入漏洞的部分方法包括：

- 不要通过用户输入构造 XPath 查询。
- 验证输入是否只包含一组安全字符。
- 对引号进行转义。

何时禁止显示警告

如果你确定输入已经过验证并且是安全的，则可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

[[[	[[
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Xml.XPath;

public partial class WebForm : System.Web.UI.Page
{
    public XPathNavigator AuthorizedOperations { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        string operation = Request.Form["operation"];

        // If an attacker uses this for input:
        //     ' or 'a' = 'a
        // Then the XPath query will be:
        //     authorizedOperation[@username = 'anonymous' and @operationName = '' or 'a' = 'a']
        // and it will return any authorizedOperation node.
        XPathNavigator node = AuthorizedOperations.SelectSingleNode(
            "//authorizedOperation[@username = 'anonymous' and @operationName = '" + operation + "']");
    }
}
```

```
Imports System
Imports System.Xml.XPath

Partial Public Class WebForm
    Inherits System.Web.UI.Page

    Public Property AuthorizedOperations As XPathNavigator

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim operation As String = Me.Request.Form("operation")

        ' If an attacker uses this for input:
        '     ' or 'a' = 'a
        ' Then the XPath query will be:
        '      authorizedOperation[@username = 'anonymous' and @operationName = '' or 'a' = 'a']
        ' and it will return any authorizedOperation node.
        Dim node As XPathNavigator = AuthorizedOperations.SelectSingleNode( _
            "//authorizedOperation[@username = 'anonymous' and @operationName = '" + operation + "']")
    End Sub
End Class
```

CA3009：查看 XML 注入漏洞的代码

2021/11/16 •

■ ID	CA3009
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问原始 XML 输出。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范 XML 注入攻击。攻击者可以使用 XML 注入向 XML 文档中插入特殊字符，使文档变成无效的 XML。或者，攻击者可能会恶意插入其所选的 XML 节点。

此规则试图查找 HTTP 请求中要访问原始 XML 写入的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会编写原始 XML 的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

不要编写原始 XML。改为使用方法或属性对输入执行 XML 编码。

或者，在编写原始 XML 之前，先对输入执行 XML 编码。

或者，使用擦除器执行基元类型转换和 XML 编码，以验证用户输入。

何时禁止显示警告

不要禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho[illegible]e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

[illegible]	[illegible]
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

选项	值
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Xml;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        XmlDocument d = new XmlDocument();
        XmlElement root = d.CreateElement("root");
        d.AppendChild(root);

        XmlElement allowedUser = d.CreateElement("allowedUser");
        root.AppendChild(allowedUser);

        allowedUser.InnerXml = "alice";

        // If an attacker uses this for input:
        //     some text<allowedUser>oscar</allowedUser>
        // Then the XML document will be:
        //     <root>some text<allowedUser>oscar</allowedUser></root>
        root.InnerXml = input;
    }
}
```

```
Imports System
Imports System.Xml

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim d As XmlDocument = New XmlDocument()
        Dim root As XmlElement = d.CreateElement("root")
        d.AppendChild(root)

        Dim allowedUser As XmlElement = d.CreateElement("allowedUser")
        root.AppendChild(allowedUser)

        allowedUser.InnerXml = "alice"

        ' If an attacker uses this for input:
        '     some text<allowedUser>oscar</allowedUser>
        ' Then the XML document will be:
        '     <root>some text<allowedUser>oscar</allowedUser></root>
        root.InnerXml = input
    End Sub
End Class
```

解决方案

```
using System;
using System.Xml;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        XmlDocument d = new XmlDocument();
        XmlElement root = d.CreateElement("root");
        d.AppendChild(root);

        XmlElement allowedUser = d.CreateElement("allowedUser");
        root.AppendChild(allowedUser);

        allowedUser.InnerText = "alice";

        // If an attacker uses this for input:
        //     some text<allowedUser>oscar</allowedUser>
        // Then the XML document will be:
        //     <root>&lt;allowedUser&gt;oscar&lt;/allowedUser&gt;some text<allowedUser>alice</allowedUser>
</root>
        root.InnerText = input;
    }
}
```

```
Imports System
Imports System.Xml

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim d As XmlDocument = New XmlDocument()
        Dim root As XmlElement = d.CreateElement("root")
        d.AppendChild(root)

        Dim allowedUser As XmlElement = d.CreateElement("allowedUser")
        root.AppendChild(allowedUser)

        allowedUser.InnerText = "alice"

        ' If an attacker uses this for input:
        '     some text<allowedUser>oscar</allowedUser>
        ' Then the XML document will be:
        '     <root>&lt;allowedUser&gt;oscar&lt;/allowedUser&gt;some text<allowedUser>alice</allowedUser>
</root>
        root.InnerText = input
    End Sub
End Class
```

CA3010：查看 XAML 注入漏洞的代码

2021/11/16 •

■ ID	CA3010
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问 System.Windows.Markup.XamlReader 加载方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范 XAML 注入攻击。XAML 是一种直接表示对象实例化和执行的标记语言。这意味着 XAML 中创建的元素可以与系统资源（例如，网络访问和文件系统 IO）交互。如果攻击者可以控制 System.Windows.Markup.XamlReader 加载方法调用的输入，则攻击者可以执行代码。

此规则试图查找 HTTP 请求中访问 System.Windows.Markup.XamlReader 加载方法的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会加载 XAML 的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

不要加载不受信任的 XAML。

何时禁止显示警告

不要禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

值	说明
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.IO;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        byte[] bytes = Convert.FromBase64String(input);
        MemoryStream ms = new MemoryStream(bytes);
        System.Windows.Markup.XamlReader.Load(ms);
    }
}
```

```
Imports System
Imports System.IO

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim bytes As Byte() = Convert.FromBase64String(input)
        Dim ms As MemoryStream = New MemoryStream(bytes)
        System.Windows.Markup.XamlReader.Load(ms)
    End Sub
End Class
```

CA3011：查看 DLL 注入漏洞的代码

2021/11/16 •

■ ID	CA3011
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问用于加载程序集的方法。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请谨慎加载不受信任的代码。如果你的 Web 应用加载不受信任的代码，攻击者可能能够将恶意 DLL 注入到你的进程中，并执行恶意代码。

此规则试图查找要访问加载程序集的方法的 HTTP 请求输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会加载程序集的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

不要加载用户输入中不受信任的 DLL。

何时禁止显示警告

不要禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。

标题	标题
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Reflection;

public partial class WebForm : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        string input = Request.Form["in"];
        byte[] rawAssembly = Convert.FromBase64String(input);
        Assembly.Load(rawAssembly);
    }
}
```

```
Imports System
Imports System.Reflection

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim input As String = Request.Form("in")
        Dim rawAssembly As Byte() = Convert.FromBase64String(input)
        Assembly.Load(rawAssembly)
    End Sub
End Class
```

CA3012：查看正则表达式注入漏洞的代码

2021/11/16 •

	[
■ ID	CA3012
■	安全性
修复是中断修复还是非中断修复	非中断

原因

可能有不受信任的 HTTP 请求输入访问正则表达式。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

处理不受信任的输入时，请注意防范正则表达式注入攻击。攻击者可以使用正则表达式注入恶意修改正则表达式，让正则表达式匹配非预期结果，或者让正则表达式占用过多 CPU，从而形成拒绝服务攻击。

此规则试图查找 HTTP 请求中要访问正则表达式的输入。

> **NOTE**
> 此规则无法跨程序集跟踪数据。例如，如果一个程序集读取 HTTP 请求输入，然后将其传递给另一个会生成正则表达式的程序集，则此规则不会产生警告。

> **NOTE**
> 对于此规则跨方法调用分析数据流的深入程度存在限制，此限制是可配置的。若要了解如何在 EditorConfig 文件中配置此限制，请参阅分析器配置。

如何解决冲突

针对正则表达式注入的缓解措施包括：

- 使用正则表达式时，始终使用匹配超时。
- 避免使用基于用户输入的正则表达式。
- 通过调用 System.Text.RegularExpressions.Regex.Escape 或其他方法，对用户输入中的特殊字符执行转义。
- 仅允许用户输入中存在非特殊字符。

何时禁止显示警告

如果你确定已在使用匹配超时并且用户输入不含特殊字符，则可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Text.RegularExpressions;

public partial class WebForm : System.Web.UI.Page
{
    public string SearchableText { get; set; }

    protected void Page_Load(object sender, EventArgs e)
    {
        string findTerm = Request.Form["findTerm"];
        Match m = Regex.Match(SearchableText, "^term=" + findTerm);
    }
}
```

```
Imports System
Imports System.Text.RegularExpressions

Public Partial Class WebForm
    Inherits System.Web.UI.Page

    Public Property SearchableText As String

    Protected Sub Page_Load(sender As Object, e As EventArgs)
        Dim findTerm As String = Request.Form("findTerm")
        Dim m As Match = Regex.Match(SearchableText, "^term=" + findTerm)
    End Sub
End Class
```

CA3061：请勿按 URL 添加架构

2021/11/16 •

	[
■ ID	CA3061
■	安全性
修复是中断修复还是非中断修复	非中断

原因

`XmlSchemaCollection.Add(String, String)` 的重载使用 `XmlUrlResolver` 以 URI 的形式指定外部 XML 架构。如果 URI 字符串受污染，则可能会导致解析恶意 XML 架构，这使攻击者能包含 XML 炸弹和恶意的外部实体。这可能导致恶意攻击者实施拒绝服务、信息泄露或服务器端请求伪造攻击。

规则说明

请勿使用 `Add` 方法的不安全重载，因为这可能会导致危险的外部引用。

如何解决冲突

- 请勿使用 `XmlSchemaCollection.Add(String, String)`。

何时禁止显示警告

如果确信 XML 不会解析危险的外部引用，则可禁止显示此规则的警告。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。第二个参数的类型为 `string`。

```
using System;
using System.Xml.Schema;
...
XmlSchemaCollection xsc = new XmlSchemaCollection();
xsc.Add("urn: bookstore - schema", "books.xsd");
```

解决方案

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Schema;
...
XmlSchemaCollection xsc = new XmlSchemaCollection();
xsc.Add("urn: bookstore - schema", new XmlTextReader(new FileStream(""xmlFilename"", FileMode.Open)));
```

CA3075:不安全的 DTD 处理

2021/11/16 •

	[
■ ID	CA3075
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果使用不安全的 DtdProcessing 实例或引用外部实体源，分析器可能会接受不受信任的输入并将敏感信息泄露给攻击者。

规则说明

XML 分析器可以通过两种方式确定文档有效性，*文档类型定义 (DTD)* 是其中一种(根据 万维网联合会 (W3C) 可扩展标记语言 (XML) 1.0的定义)。此规则会查找接受不受信任数据的属性和实例，以提醒开发人员潜在的信息泄漏威胁或拒绝服务 (DoS) 攻击。在以下情况下触发此规则：

- 在 XmlReader 实例上启用了 DtdProcessing，它使用 XmlUrlResolver解析外部 XML 实体。
- 设置了 XML 中的 InnerXml 属性。
- DtdProcessing 属性设置为“分析”。
- 使用 XmlResolver 而不是 XmlSecureResolver 处理不受信任的输入。
- 使用不安全的 XmlReaderSettings 实例或根本不使用任何实例调用 XmlReader.Create 方法。
- 使用不安全的默认设置或值创建 XmlReader。

在这些情况下，结果均相同：来自文件系统或来自处理 XML 的计算机的网络共享的文件都将面临攻击，或 DTD 处理可用作 DoS 向量。

如何解决冲突

- 正确捕获和处理所有 XmlTextReader 异常以避免路径信息泄露。
- 使用 XmlSecureResolver 来限制 XmlTextReader 可以访问的资源。
- 通过将 XmlReader 属性设置为 XmlResolver null，**不允许** 打开任何外部资源。
- 确保从可信任的源赋值 DataViewManager.DataViewSettingCollectionString 属性。

.NET Framework 3.5 及更早版本

- 如果正在处理不可信的源，请通过将 ProhibitDtd 属性设置为“true”来禁用 DTD 处理。
- XmlTextReader 类具有完全信任继承要求。

.NET Framework 4 及更高版本

- 如果正在处理不可信的源，请通过将 XmlReaderSettings.DtdProcessing 属性设置为“禁止”或“忽略”来避免

启用 DtdProcessing 。

- 确保在所有 InnerXml 用例中 load () 方法均采用 XmlReader 实例。

> **NOTE**
> 此规则可能会针对某些有效 XmlSecureResolver 实例进行误报。

何时禁止显示警告

除非确信已知道输入是来自受信任的源，否则请勿禁止显示此警告的规则。

伪代码示例

冲突 1

```
using System.IO;
using System.Xml.Schema;

class TestClass
{
    public XmlSchema Test
    {
        get
        {
            var src = "";
            TextReader tr = new StreamReader(src);
            XmlSchema schema = XmlSchema.Read(tr, null); // warn
            return schema;
        }
    }
}
```

解决方案 1

```
using System.IO;
using System.Xml;
using System.Xml.Schema;

class TestClass
{
    public XmlSchema Test
    {
        get
        {
            var src = "";
            TextReader tr = new StreamReader(src);
            XmlReader reader = XmlReader.Create(tr, new XmlReaderSettings() { XmlResolver = null });
            XmlSchema schema = XmlSchema.Read(reader , null);
            return schema;
        }
    }
}
```

冲突 2

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public XmlReaderSettings settings = new XmlReaderSettings();
        public void TestMethod(string path)
        {
            var reader = XmlReader.Create(path, settings);  // warn
        }
    }
}
```

解决方案 2

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public XmlReaderSettings settings = new XmlReaderSettings()
        {
            DtdProcessing = DtdProcessing.Prohibit
        };

        public void TestMethod(string path)
        {
            var reader = XmlReader.Create(path, settings);
        }
    }
}
```

冲突 3

```
using System.Xml;

namespace TestNamespace
{
    public class DoNotUseSetInnerXml
    {
        public void TestMethod(string xml)
        {
            XmlDocument doc = new XmlDocument() { XmlResolver = null };
            doc.InnerXml = xml; // warn
        }
    }
}
```

```
using System.Xml;

namespace TestNamespace
{
    public class DoNotUseLoadXml
    {
        public void TestMethod(string xml)
        {
            XmlDocument doc = new XmlDocument(){ XmlResolver = null };
            doc.LoadXml(xml); // warn
        }
    }
}
```

解决方法 3

```
using System.Xml;

public static void TestMethod(string xml)
{
    XmlDocument doc = new XmlDocument() { XmlResolver = null };
    System.IO.StringReader sreader = new System.IO.StringReader(xml);
    XmlReader reader = XmlReader.Create(sreader, new XmlReaderSettings() { XmlResolver = null });
    doc.Load(reader);
}
```

冲突 4

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace TestNamespace
{
    public class UseXmlReaderForDeserialize
    {
        public void TestMethod(Stream stream)
        {
            XmlSerializer serializer = new XmlSerializer(typeof(UseXmlReaderForDeserialize));
            serializer.Deserialize(stream); // warn
        }
    }
}
```

解决方法 4

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace TestNamespace
{
    public class UseXmlReaderForDeserialize
    {
        public void TestMethod(Stream stream)
        {
            XmlSerializer serializer = new XmlSerializer(typeof(UseXmlReaderForDeserialize));
            XmlReader reader = XmlReader.Create(stream, new XmlReaderSettings() { XmlResolver = null });
            serializer.Deserialize(reader );
        }
    }
}
```

冲突 5

```
using System.Xml;
using System.Xml.XPath;

namespace TestNamespace
{
    public class UseXmlReaderForXPathDocument
    {
        public void TestMethod(string path)
        {
            XPathDocument doc = new XPathDocument(path); // warn
        }
    }
}
```

解决方案 5

```
using System.Xml;
using System.Xml.XPath;

namespace TestNamespace
{
    public class UseXmlReaderForXPathDocument
    {
        public void TestMethod(string path)
        {
            XmlReader reader = XmlReader.Create(path, new XmlReaderSettings() { XmlResolver = null });
            XPathDocument doc = new XPathDocument(reader);
        }
    }
}
```

冲突 6

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        public XmlDocument doc = new XmlDocument() { XmlResolver = new XmlUrlResolver() };
    }
}
```

解决方案 6

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        public XmlDocument doc = new XmlDocument() { XmlResolver = null }; // or set to a XmlSecureResolver
instance
    }
}
```

冲突 7

```
using System.Xml;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod()
        {
            var reader = XmlTextReader.Create(""doc.xml""); //warn
        }
    }
}
```

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public void TestMethod(string path)
        {
            try {
                XmlTextReader reader = new XmlTextReader(path); // warn
            }
            catch { throw ; }
            finally {}
        }
    }
}
```

解决方案 7

```
using System.Xml;

namespace TestNamespace
{
    public class TestClass
    {
        public void TestMethod(string path)
        {
            XmlReaderSettings settings = new XmlReaderSettings() { XmlResolver = null };
            XmlReader reader = XmlReader.Create(path, settings);
        }
    }
}
```

NOTE

尽管建议使用 XmlReader.Create 来创建 XmlReader 实例，但是其行为与 XmlTextReader 有所不同。Create 中的 XmlReader 在 XML 值中将 `\r\n` 规范化为 `\n`，而 XmlTextReader 保留 `\r\n` 序列。

CA3076:不安全的 XSLT 脚本执行

2021/11/16 •

■ ID	CA3076
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果在 .NET 应用程序中不安全地执行可扩展样式表语言转换 (XSLT)，处理器可能会解析不受信任的 URI 引用，这种引用会把敏感信息泄露给攻击者，从而导致拒绝服务和跨站点攻击。有关详细信息，请参阅 XSLT 安全注意事项(.NET 指南)。

规则说明

XSLT 是万维网联合会 (W3C) 标准，用于转换 XML 数据。XSLT 通常用于编写样式表，以将 XML 数据转换为其他格式，如 HTML、定长文本、以逗号分隔的文本或其他 XML 格式。尽管默认情况下禁止，你仍可以选择为项目启用该功能。

为了确保不暴露攻击面，每当 XslCompiledTransform.Load 接收 XsltSettings 和 XmlResolver 的不安全组合实例时，此规则都会触发，从而允许恶意脚本处理。

如何解决冲突

- 将不安全的 XsltSettings 参数替换为 XsltSettings.Default 或替换为具有禁用文档函数和脚本执行的实例。
- 将 XmlResolver 参数替换为 null 或 XmlSecureResolver 实例。

何时禁止显示警告

除非确信已知道输入是来自受信任的源，否则请勿禁止显示此警告的规则。

伪代码示例

使用 XsltSettings.TrustedXslt 的冲突

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
         void TestMethod()
        {
             XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
             var settings = XsltSettings.TrustedXslt;
             var resolver = new XmlUrlResolver();
             xslCompiledTransform.Load("testStylesheet", settings, resolver); // warn
        }
    }
}
```

使用 XsltSettings.Default 的解决方案

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        void TestMethod()
        {
            XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
            var settings = XsltSettings.Default;
            var resolver = new XmlUrlResolver();
            xslCompiledTransform.Load("testStylesheet", settings, resolver);
        }
    }
}
```

冲突—未禁用文档函数和脚本执行

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod(XsltSettings settings)
        {
            try
            {
                XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
                var resolver = new XmlUrlResolver();
                xslCompiledTransform.Load("testStylesheet", settings, resolver); // warn
            }
            catch { throw; }
            finally { }
        }
    }
}
```

解决方案—禁用文档函数和脚本执行

```
using System.Xml;
using System.Xml.Xsl;

namespace TestNamespace
{
    class TestClass
    {
        private static void TestMethod(XsltSettings settings)
        {
            try
            {
                XslCompiledTransform xslCompiledTransform = new XslCompiledTransform();
                settings.EnableDocumentFunction = false;
                settings.EnableScript = false;
                var resolver = new XmlUrlResolver();
                xslCompiledTransform.Load("testStylesheet", settings, resolver);
            }
            catch { throw; }
            finally { }
        }
    }
}
```

另请参阅

- XSLT 安全注意事项(.NET 指南)

CA3077:API 设计、XML 文档和 XML 文本读取器中的不安全处理

2021/11/16 •

■ ID	CA3077
■	安全性
修复是中断修复还是非中断修复	非中断

原因

当设计派生自 XMLDocument 和 XMLTextReader 的 API 时，请注意 DtdProcessing。当引用或解析外部实体源或设置 XML 中的不安全值时，使用不安全的 DTDProcessing 实例可能会导致信息泄露。

规则说明

XML 分析器可以通过两种方式确定文档有效性，*文档类型定义 (DTD)* 是其中一种(根据 万维网联合会 (W3C) 可扩展标记语言 (XML) 1.0的定义)。此规则查找接受不受信任数据的某些属性和实例以提醒开发人员有关的潜在 Information Disclosure 威胁，该威胁可能会导致 拒绝服务 (DoS) 攻击。在以下情况下触发此规则：

- XmlDocument 或 XmlTextReader 类使用默认解析程序值进行 DTD 处理。
- 没有为 XmlDocument 或 XmlTextReader 派生类定义的构造函数或没有用于 XmlResolver的安全值。

如何解决冲突

- 正确捕获和处理所有 XmlTextReader 异常以避免路径信息泄露。
- 使用 XmlSecureResolver 而不是 XmlResolver 来限制 XmlTextReader 可访问的资源。

何时禁止显示警告

除非确信已知道输入是来自受信任的源，否则请勿禁止显示此警告的规则。

伪代码示例

冲突

```
using System;
using System.Xml;

namespace TestNamespace
{
    class TestClass : XmlDocument
    {
        public TestClass () {} // warn
    }

    class TestClass2 : XmlTextReader
    {
        public TestClass2() // warn
        {
        }
    }
}
```

解决方案

```
using System;
using System.Xml;

namespace TestNamespace
{
    class TestClass : XmlDocument
    {
        public TestClass ()
        {
            XmlResolver = null;
        }
    }

    class TestClass2 : XmlTextReader
    {
        public TestClass2()
        {
               XmlResolver = null;
        }
    }
}
```

CA3147:使用 ValidateAntiForgeryToken 标记谓词处理程序

2021/11/16 •

■ ID	CA3147
■	安全性
修复是中断修复还是非中断修复	非中断

原因

ASP.NET MVC 控制器操作方法未使用 ValidateAntiForgeryTokenAttribute 或指定 HTTP 谓词(如 HttpGetAttribute 或 AcceptVerbsAttribute)的属性来进行标记。

规则说明

设计 ASP.NET MVC 控制器时，请注意防范跨网站请求伪造攻击。跨网站请求伪造攻击可将来自经过身份验证的用户的恶意请求发送到 ASP.NET MVC 控制器。有关详细信息，请参阅 ASP.NET MVC 和网页中的 XSRF/CSRF 防护。

此规则检查 ASP.NET MVC 控制器的操作方法：

- 具有 ValidateAntiforgeryTokenAttribute，并指定允许的 HTTP 谓词，但不包括 HTTP GET。
- 将 HTTP GET 指定为允许的谓词。

如何解决冲突

- 用于处理 HTTP GET 请求且没有潜在有害副作用的 ASP.NET MVC 控制器操作，请向该方法添加 HttpGetAttribute。

  如果有 ASP.NET MVC 控制器操作，该操作可处理 HTTP GET 请求，并可能产生诸如修改敏感数据之类的潜在有害副作用，则应用程序很容易受到跨站点请求伪造攻击。你需要重新设计应用程序，以便仅 HTTP POST、PUT 或 DELETE 请求能执行敏感操作。

- 对于处理 HTTP POST、PUT 或 DELETE 请求的ASP.NET MVC 控制器操作，请添加 ValidateAntiForgeryTokenAttribute 以及指定允许的 HTTP 谓词属性(AcceptVerbsAttribute、HttpPostAttribute、HttpPutAttribute 或 HttpDeleteAttribute)。此外，还需从 MVC 视图或 Razor 网页调用 HtmlHelper.AntiForgeryToken() 方法。有关示例，请参阅检查编辑方法和编辑视图。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- ASP.NET MVC 控制器操作不会产生有害的副作用。
- 该应用程序以另一种方式验证防伪令牌。

ValidateAntiForgeryToken 特性示例

冲突：

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        public ActionResult TransferMoney(string toAccount, string amount)
        {
            // You don't want an attacker to specify to who and how much money to transfer.

            return null;
        }
    }
}
```

解决方案：

```
using System;
using System.Xml;

namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        [HttpPost]
        [ValidateAntiForgeryToken]
        public ActionResult TransferMoney(string toAccount, string amount)
        {
            return null;
        }
    }
}
```

HttpGet 特性示例

冲突：

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        public ActionResult Help(int topicId)
        {
            // This Help method is an example of a read-only operation with no harmful side effects.
            return null;
        }
    }
}
```

解决方案：

```
namespace TestNamespace
{
    using System.Web.Mvc;

    public class TestController : Controller
    {
        [HttpGet]
        public ActionResult Help(int topicId)
        {
            return null;
        }
    }
}
```

CA5350:请勿使用弱加密算法

2021/11/16 •

	[
■ ID	CA5350
■	Microsoft.Cryptography
▬▬▬	非中断

NOTE

此警告上次更新于 2015 年 11 月。

原因

TripleDES 等加密算法和 SHA1 及 RIPEMD160 等哈希算法被视为弱加密算法。

这些加密算法不能与更现代的对应算法提供同样多的安全保证。与更现代的哈希算法相比，加密哈希算法 SHA1 和 RIPEMD160 提供的冲突抗性较低。与更现代的加密算法相比，加密算法 TripleDES 提供的安全位数更少。

规则说明

现今出于多种原因而使用弱加密算法和哈希函数，但不应将其用于保证其所保护数据的保密性。

该规则在代码中发现 3DES、SHA1 或 RIPEMD160 算法时将触发并向用户发送警告。

如何解决冲突

使用更强大的加密选项：

- 对于 TripleDES 加密，请使用 Aes 加密。
- 对于 SHA1 或 RIPEMD160 哈希函数，请从 SHA-2 系列（例如 SHA512、SHA384、SHA256）中选择使用。

何时禁止显示警告

当数据所需的保护级别不需要安全保证时，请禁止显示此规则的警告。

伪代码示例

到本文撰写时为止，下面的伪代码示例说明了此规则检测到的模式。

SHA-1 哈希冲突

```
using System.Security.Cryptography;
...
var hashAlg = SHA1.Create();
```

解决方案：

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

RIPEMD160 哈希冲突

```
using System.Security.Cryptography;
...
var hashAlg = RIPEMD160Managed.Create();
```

解决方案:

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

TripleDES 加密冲突

```
using System.Security.Cryptography;
...
using (TripleDES encAlg = TripleDES.Create())
{
  ...
}
```

解决方案:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

CA5351 不使用损坏的加密算法

2021/11/16 •

■ ID	CA5351
■	安全性
修复是中断修复还是非中断修复	非中断

NOTE
此警告上次更新于 2015 年 11 月。

原因

哈希函数(如 MD5)和加密算法(如 DES 和 RC2)可能会带来重大风险，并可能通过普通的攻击技术(如暴力攻击和哈希冲突)导致暴露敏感信息。

下面的加密算法列表受到已知加密攻击。加密哈希算法 MD5 受到哈希冲突攻击。根据使用情况，哈希冲突可能会使依赖哈希函数这一唯一加密输出的系统受到假冒、篡改或其他类型的攻击。加密算法 DES 和 RC2 受到加密攻击，可能会导致加密数据的意外泄露。

规则说明

损坏的加密算法不安全，不鼓励继续使用。尽管具体的漏洞因使用环境的不同而异，但 MD5 哈希算法仍易遭到已知的冲突攻击。用于确保数据完整性的哈希算法(例如文件签名或数字证书)特别容易受到攻击。在这种情况下，攻击者可能会生成两个独立的数据块，以便在不更改哈希值或使相关数字签名无效的情况下，将良性数据替换为恶意数据。

对于加密算法：

- DES 加密使用的密钥强度较低，可能在一天内被暴力破解。
- RC2 加密容易遭受与密钥相关的攻击，攻击者可以通过这些攻击找出所有密钥值之间的数学关系。

当在源代码中找到上述任何加密函数时，将触发此规则并向用户发出警告。

如何解决冲突

使用更强大的加密选项：

- 对于 MD5，请使用 SHA-2 系列中的哈希(例如 SHA512、SHA384、SHA256)。
- 对于 DES 和 RC2，请使用 Aes 加密。

何时禁止显示警告

除非经过加密专家审查，否则不要禁止显示此规则的警告。

伪代码示例

下面的伪代码示例演示此规则检测到的模式和可能的替代模式。

MD5 哈希冲突

```
using System.Security.Cryptography;
...
var hashAlg = MD5.Create();
```

解决方案:

```
using System.Security.Cryptography;
...
var hashAlg = SHA256.Create();
```

RC2 加密冲突

```
using System.Security.Cryptography;
...
RC2 encAlg = RC2.Create();
```

解决方案:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

DES 加密冲突

```
using System.Security.Cryptography;
...
DES encAlg = DES.Create();
```

解决方案:

```
using System.Security.Cryptography;
...
using (AesManaged encAlg = new AesManaged())
{
  ...
}
```

CA5358：请勿使用不安全的密码模式

2021/11/16 •

■ ID	CA5358
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用下列未批准的不安全加密模式之一：

- System.Security.Cryptography.CipherMode.ECB
- System.Security.Cryptography.CipherMode.OFB
- System.Security.Cryptography.CipherMode.CFB

规则说明

这些模式容易受到攻击，并可能导致敏感信息泄露。例如，使用 `ECB` 对纯文本块进行加密始终都会生成相同的密码文本，因此它可以轻松判断两个加密消息是否相同。使用批准的模式可以避免这些不必要的风险。

如何解决冲突

- 仅使用批准的模式（System.Security.Cryptography.CipherMode.CBC、System.Security.Cryptography.CipherMode.CTS）。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 加密专家已经查看并批准了密码模式的使用。
- 所引用的 CipherMode 未用于加密操作。

伪代码示例

向 Mode 属性分配 ECB

```
using System.Security.Cryptography;

class ExampleClass {
    private static void ExampleMethod () {
        RijndaelManaged rijn = new RijndaelManaged
        {
            Mode = CipherMode.ECB
        };
    }
}
```

使用值 ECB

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    private static void ExampleMethod()
    {
        Console.WriteLine(CipherMode.ECB);
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass {
    private static void ExampleMethod () {
        RijndaelManaged rijn = new RijndaelManaged
        {
            Mode = CipherMode.CBC
        };
    }
}
```

CA5359:请勿禁用证书验证

2021/11/16 •

	[
■ ID	CA5359
■	安全性
修复是中断修复还是非中断修复	非中断

原因

赋值给 ServicePointManager.ServerCertificateValidationCallback 的回叫始终返回 `true`。

规则说明

证书有助于验证服务器的身份。客户端应验证服务器证书，确保将请求发送到目标服务器。如果 ServicePointManager.ServerCertificateValidationCallback 始终返回 `true`，则默认情况下，任何证书都将通过所有传出 HTTPS 请求的验证。

如何解决冲突

- 考虑在需要自定义证书验证的特定传出 HTTPS 请求上重写证书验证逻辑，而不是重写全局 ServicePointManager.ServerCertificateValidationCallback。
- 仅将自定义验证逻辑应用于特定主机名和证书，否则请检查 SslPolicyErrors 枚举值是否为 `None`。

何时禁止显示警告

如果将多个委托附加到 ServerCertificateValidationCallback，则仅考虑最后一个委托的值，因此对于其他委托，可禁止显示警告。但是，你可能需要完全删除未使用的委托。

伪代码示例

冲突

```
using System.Net;

class ExampleClass
{
    public void ExampleMethod()
    {
        ServicePointManager.ServerCertificateValidationCallback += (sender, cert, chain, error) => { return
true; };
    }
}
```

解决方案

```
using System.Net;
using System.Net.Security;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod()
    {
        ServicePointManager.ServerCertificateValidationCallback += SelfSignedForLocalhost;
    }

    private static bool SelfSignedForLocalhost(object sender, X509Certificate certificate, X509Chain chain,
SslPolicyErrors sslPolicyErrors)
    {
        if (sslPolicyErrors == SslPolicyErrors.None)
        {
            return true;
        }

        // For HTTPS requests to this specific host, we expect this specific certificate.
        // In practice, you'd want this to be configurable and allow for multiple certificates per host, to
enable
        // seamless certificate rotations.
        return sender is HttpWebRequest httpWebRequest
                && httpWebRequest.RequestUri.Host == "localhost"
                && certificate is X509Certificate2 x509Certificate2
                && x509Certificate2.Thumbprint == "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
                && sslPolicyErrors == SslPolicyErrors.RemoteCertificateChainErrors;
    }
}
```

CA5360:在反序列化中不要调用危险的方法

2021/11/16 •

	‹
■ ID	CA5360
■	安全性
修复是中断修复还是非中断修复	非中断

原因

在反序列化中调用以下其中一个危险方法：

- System.IO.Directory.Delete
- System.IO.DirectoryInfo.Delete
- System.IO.File.AppendAllLines
- System.IO.File.AppendAllText
- System.IO.File.AppendText
- System.IO.File.Copy
- System.IO.File.Delete
- System.IO.File.WriteAllBytes
- System.IO.File.WriteAllLines
- System.IO.File.WriteAllText
- System.IO.FileInfo.Delete
- System.IO.Log.LogStore.Delete
- System.Reflection.Assembly.GetLoadedModules
- System.Reflection.Assembly.Load
- System.Reflection.Assembly.LoadFrom
- System.Reflection.Assembly.LoadFile
- System.Reflection.Assembly.LoadModule
- System.Reflection.Assembly.LoadWithPartialName
- System.Reflection.Assembly.ReflectionOnlyLoad
- System.Reflection.Assembly.ReflectionOnlyLoadFrom
- System.Reflection.Assembly.UnsafeLoadFrom

所有符合以下其中一项要求的方法都可以是反序列化的回调：

- 标记有 System.Runtime.Serialization.OnDeserializingAttribute。
- 标记有 System.Runtime.Serialization.OnDeserializedAttribute。
- 实现 System.Runtime.Serialization.IDeserializationCallback.OnDeserialization。
- 实现 System.IDisposable.Dispose。
- 为析构函数。

规则说明

不安全的反序列化是一种漏洞。当使用不受信任的数据来损害应用程序的逻辑，造成拒绝服务 (DoS) 攻击，或甚至在反序列化时任意执行代码，就会出现该漏洞。应用程序对受其控制的不受信任数据进行反序列化时，恶意用户很可能会滥用这些反序列化功能。具体来说，就是在反序列化过程中调用危险方法。如果攻击者成功执行不安全的反序列化攻击，就能实施更多攻击，如 DoS 攻击、绕过身份验证和执行远程代码。

如何解决冲突

从自动运行的反序列化回调中删除这些危险方法。仅在验证输入后调用危险方法。

何时禁止显示警告

在以下情况下，可禁止显示此规则的警告：

- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 序列化的数据不会被篡改。序列化后，对序列化的数据进行加密签名。在反序列化之前，验证加密签名。保护加密密钥不被泄露，并设计密钥轮换。
- 验证数据对应用程序安全。

伪代码示例

冲突

```
using System;
using System.IO;
using System.Runtime.Serialization;

[Serializable()]
public class ExampleClass : IDeserializationCallback
{
    private string member;

    void IDeserializationCallback.OnDeserialization(Object sender)
    {
        var sourceFileName = "malicious file";
        var destFileName = "sensitive file";
        File.Copy(sourceFileName, destFileName);
    }
}
```

解决方案

```
using System;
using System.IO;
using System.Runtime.Serialization;

[Serializable()]
public class ExampleClass : IDeserializationCallback
{
    private string member;

    void IDeserializationCallback.OnDeserialization(Object sender)
    {
        var sourceFileName = "malicious file";
        var destFileName = "sensitive file";
        // Remove the potential dangerous operation.
        // File.Copy(sourceFileName, destFileName);
    }
}
```

CA5361：不禁用 SChannel 使用强加密

2021/11/16 •

■ ID	CA5361
■	安全性
修复是中断修复还是非中断修复	非中断

原因

AppContext.SetSwitch 方法调用将 `Switch.System.Net.DontEnableSchUseStrongCrypto` 设置为 `true`。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

将 `Switch.System.Net.DontEnableSchUseStrongCrypto` 设置为 `true`，从而降低传出传输层安全性连接中使用的加密性。较弱的加密性会泄露应用程序与服务器之间通信的机密性，使攻击者更易于窃听敏感数据。有关详细信息，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法。

如何解决冲突

- 如果应用程序面向 .NET Framework v4.6 或更高版本，则可以删除 AppContext.SetSwitch 方法调用，或将开关的值设置为 `false`。
- 如果应用程序面向 .NET Framework 早于 v4.6 的版本，并在 .NET Framework v4.6 或更高版本上运行，则将开关的值设置为 `false`。
- 否则，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法来缓解问题。

何时禁止显示警告

如需连接到无法升级为使用安全 TLS 配置的旧服务，可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅符号名称(包括具有相应名称的所有符号,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀,例如表示方法的 `M:`、表示类型的 `T:`,以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数,`.cctor` 表示静态构造函数。

示例:

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如,若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行,请将以下键值对添加到项目中的 .editorconfig 文件:

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅类型名称(包括具有相应名称的所有类型,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式,前缀为 `T:`(可选)。

示例:

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5361 violation
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", true);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        ' CA5361 violation
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", true)
    End Sub
End Class
```

解决方案

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", false);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        AppContext.SetSwitch("Switch.System.Net.DontEnableSchUseStrongCrypto", false)
    End Sub
End Class
```

CA5362:反序列化对象图中存在潜在引用循环

2021/11/16 •

■ ID	CA5362
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 System.SerializableAttribute 标记的类具有字段或属性，可直接或间接引用包含的对象，从而允许潜在的引用循环。

规则说明

反序列化不受信任的数据时，处理反序列化对象图的任何代码都需要在处理引用循环时不进入无限循环。这包括反序列化回叫中的一部分代码和在反序列化完成后处理对象图的代码。否则攻击者可能会使用包含引用循环的恶意数据执行拒绝服务攻击。

此规则并不一定意味着存在漏洞，它只是在反序列化的对象图中标记潜在的引用循环。

如何解决冲突

请勿序列化类并删除 SerializableAttribute。或者重新设计应用程序，以便可以从可序列化类中删除自引用成员。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- 已知输入为受信任输入。考虑应用程序的信任边界和数据流可能会随时间发生变化。
- 处理反序列化数据的所有代码都会检测并处理引用循环，而不会进入无限循环或使用过多的资源。

伪代码示例

潜在的引用循环冲突

```
using System;

[Serializable()]
class ExampleClass
{
    public ExampleClass ExampleProperty {get; set;}

    public int NormalProperty {get; set;}
}

class AnotherClass
{
    // The argument passed by could be `JsonConvert.DeserializeObject<ExampleClass>(untrustedData)`.
    public void AnotherMethod(ExampleClass ec)
    {
        while(ec != null)
        {
            Console.WriteLine(ec.ToString());
            ec = ec.ExampleProperty;
        }
    }
}
```

解决方案

```
using System;

[Serializable()]
class ExampleClass
{
    [NonSerialized]
    public ExampleClass ExampleProperty {get; set;}

    public int NormalProperty {get; set;}
}

class AnotherClass
{
    // The argument passed by could be `JsonConvert.DeserializeObject<ExampleClass>(untrustedData)`.
    public void AnotherMethod(ExampleClass ec)
    {
        while(ec != null)
        {
            Console.WriteLine(ec.ToString());
            ec = ec.ExampleProperty;
        }
    }
}
```

CA5363：请勿禁用请求验证

2021/11/16 •

	▮
■ ID	CA5363
■	安全性
修复是中断修复还是非中断修复	非中断

原因

对于类或方法，属性 `ValidateInput` 设置为 `false`。

规则说明

请求验证是 ASP.NET 中的一项功能，可检查 HTTP 请求并确定这些请求是否包含可能导致跨站点脚本编写等注入攻击的潜在危险内容。

如何解决冲突

将 `ValidateInput` 属性设置为 `true` 或将其完全删除。或者，使用 `AllowHTMLAttribute` 允许在输入的特定部分中使用 HTML。

何时禁止显示警告

如果传入 HTTP 请求中的所有有效负载来自受信任的实体，并且在传输之前或期间无法被对手篡改，则可以禁止显示此冲突。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。这样会禁用输入验证。

```
using System.Web.Mvc;

class TestControllerClass
{
    [ValidateInput(false)]
    public void TestActionMethod()
    {
    }
}
```

解决方案

```
using System.Web.Mvc;

class TestControllerClass
{
    [ValidateInput(true)]
    public void TestActionMethod()
    {
    }
}
```

CA5364：不使用已弃用的安全协议

2021/11/16 •

■ ID	CA5364
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果满足以下任一条件，则会触发此规则：

- 引用了已弃用的 System.Net.SecurityProtocolType 值。
- 表示已弃用的值分配到 SecurityProtocolType 变量的整数值。

已弃用的值包括：

- Ssl3
- Tls
- Tls10
- Tls11

规则说明

传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。有关标识和删除已弃用协议版本的指导，请参阅解决 TLS 1.0 问题（第 2 版）。

如何解决冲突

请勿使用已弃用的 TLS 协议版本。

何时禁止显示警告

如果出现以下情况，可禁止显示此警告：

- 未使用对已弃用协议版本的引用来启用已弃用的版本。
- 连接到无法升级为使用安全 TLS 配置的旧服务。

伪代码示例

枚举名称冲突

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5364 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls11 | SecurityProtocolType.Tls12;
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5364 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls11 Or SecurityProtocolType.Tls12
    End Sub
End Class
```

整数值冲突

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5364 violation
        ServicePointManager.SecurityProtocol = (SecurityProtocolType) 768;    // TLS 1.1
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5364 violation
        ServicePointManager.SecurityProtocol = CType(768, SecurityProtocolType)   ' TLS 1.1
    End Sub
End Class
```

解决方案

```
using System;
using System.Net;

public class TestClass
{
    public void TestMethod()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    End Sub
End Class
```

相关规则

CA5386：避免对 SecurityProtocolType 值进行硬编码

CA5397：不使用已弃用的 SslProtocols 值

CA5398：避免硬编码的 SslProtocols 值

CA5365:请勿禁用 HTTP 头检查

2021/11/16 •

	[
■ ID	CA5365
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将 EnableHeaderChecking 设置为 `false`。

规则说明

通过 HTTP 标头检查，可对在响应标头中找到的回车符和换行符（`\r` 和 `\n`）进行编码。此编码有助于避免注入攻击，这些注入攻击会攻击对响应标头中包含的不受信数据进行回显的应用程序。

如何解决冲突

将 EnableHeaderChecking 设置为 `true`。或者，删除对 `false` 的赋值，因为默认值为 `true`。

何时禁止显示警告

HTTP 标头延续依赖于跨越多行的标头，并且在其中需要换行。如果需要使用标头延续，则需要将 EnableHeaderChecking 属性设置为 `false`。检查标头会影响性能。如果你确定已经执行了正确的检查，可通过关闭此功能来提高应用程序的性能。在禁用此功能之前，请确保已在这一方面采用正确的预防措施。

伪代码示例

```
using System;
using System.Web.Configuration;

class TestClass
{
    public void TestMethod()
    {
        HttpRuntimeSection httpRuntimeSection = new HttpRuntimeSection();
        httpRuntimeSection.EnableHeaderChecking = false;
    }
}
```

解决方案

```
using System;
using System.Web.Configuration;

class TestClass
{
    public void TestMethod()
    {
        HttpRuntimeSection httpRuntimeSection = new HttpRuntimeSection();
        httpRuntimeSection.EnableHeaderChecking = true;
    }
}
```

CA5366:将 XmlReader 用于数据集读取 XML

2021/11/16 •

	[
■ ID	CA5366
■	安全性
修复是中断修复还是非中断修复	非中断

原因

文档类型定义 (DTD) 定义了 XML 文档的结构以及合法元素和特性。从外部资源引用 DTD 可能导致潜在的拒绝服务 (DoS) 攻击。大多读取器不能禁用 DTD 处理并限制外部引用加载，但 System.Xml.XmlReader 除外。使用其他这些读取器通过以下某种方法加载 XML 会触发此规则：

- ReadXml
- ReadXmlSchema
- ReadXmlSerializable

规则说明

使用 System.Data.DataSet 读取包含不受信任数据的 XML 可能会加载危险的外部引用，这应该使用带有安全解析程序或禁用 DTD 处理的 XmlReader 对其进行限制。

如何解决冲突

使用 XmlReader 或其派生类读取 XML。

何时禁止显示警告

处理受信任数据源时，禁止显示此规则的警告。

伪代码示例

冲突

```
using System.Data;
using System.IO;

public class ExampleClass
{
    public void ExampleMethod()
    {
        new DataSet().ReadXml(new FileStream("xmlFilename", FileMode.Open));
    }
}
```

解决方案

```
using System.Data;
using System.IO;
using System.Xml;

public class ExampleClass
{
    public void ExampleMethod()
    {
        new DataSet().ReadXml(new XmlTextReader(new FileStream("xmlFilename", FileMode.Open)));
    }
}
```

CA5367：请勿序列化具有 Pointer 字段的类型

2021/11/16 •

	[
■ ID	CA5367
■	安全性
修复是中断修复还是非中断修复	非中断

原因

Pointer 不是类型安全的，这意味着你无法保证它们所指向的内存的正确性。因此，序列化具有 Pointer 字段的类型会带来安全风险，它可能允许攻击者控制指针。

规则说明

此规则会检查是否存在具有 Pointer 字段或属性的可序列化类。无法进行序列化的成员可能是指针，例如使用 System.NonSerializedAttribute 进行标记的静态成员或字段。

如何解决冲突

不要对可序列化类中的成员使用指针类型，也不要对作为指针的成员进行序列化。

何时禁止显示警告

不要冒险在可序列化类型中使用指针。

伪代码示例

冲突

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private int* pointer;
}
```

解决方案 1

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private int i;
}
```

解决方案 2

```
using System;

[Serializable()]
unsafe class TestClassA
{
    private static int* pointer;
}
```

CA5368:针对派生自 Page 的类设置 ViewStateUserKey

2021/11/16 •

	ι
■ ID	CA5368
■	安全性
修复是中断修复还是非中断修复	非中断

原因

未在 Page.OnInit 或 `Page_Init` 方法中分配 Page.ViewStateUserKey 属性。

规则说明

设计 ASP.NET Web 窗体时，请注意防范跨网站请求伪造 (CSRF) 攻击。CSRF 攻击可将来自经过身份验证的用户的恶意请求发送到 ASP.NET Web 窗体。

在 ASP.NET Web 窗体中防御 CSRF 攻击的一种方法是将页面的 ViewStateUserKey 设置为不可预测且对会话唯一的字符串。有关详细信息，请参阅利用 ASP.NET 内置功能来防范 Web 攻击。

如何解决冲突

将每个会话的 ViewStateUserKey 属性设置为不可预测的唯一字符串。例如，如果使用 ASP.NET 会话状态，则 HttpSessionState.SessionID 将起作用。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- ASP.NET Web 窗体页不执行敏感操作。
- 跨网站请求伪造攻击以此规则无法检测的方式得以减轻。例如，如果页面继承自包含 CSRF 防御的母版页。

伪代码示例

冲突

```
using System;
using System.Web.UI;

class ExampleClass : Page
{
    protected override void OnInit (EventArgs e)
    {
    }
}
```

解决方案

```
using System;
using System.Web.UI;

class ExampleClass : Page
{
    protected override void OnInit (EventArgs e)
    {
        // Assuming that your page makes use of ASP.NET session state and the SessionID is stable.
        ViewStateUserKey = Session.SessionID;
    }
}
```

CA5369：将 XmlReader 用于反序列化

2021/11/16 •

	‖
■ ID	CA5369
■	安全性
修复是中断修复还是非中断修复	非中断

原因

反序列化具有未使用 `XmlReader` 对象实例化的 XmlSerializer.Deserialize 的不受信任 XML 输入可能会导致拒绝服务、信息泄露和服务器端请求伪造攻击。不受信任的 DTD 和 XML 架构处理使这些攻击成为可能，这使攻击者能在 XML 中包含 XML 炸弹和恶意的外部实体。只有使用 `XmlReader` 才可以禁用 DTD。在 .NET Framework 4.0 及更高版本中，作为 `XmlReader` 的内联 XML 架构处理会默认将 `ProhibitDtd` 和 `ProcessInlineSchema` 属性设置为 `false`。其他选项（例如 `Stream`、`TextReader` 和 `XmlSerializationReader`）不能禁用 DTD 处理。

规则说明

处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用，应使用具有安全解析程序或禁用了 DTD 和 XML 内联架构处理的 `XmlReader` 来限制这种行为。此规则检测使用 XmlSerializer.Deserialize 方法的代码，而不将 `XmlReader` 作为构造函数参数。

如何解决冲突

请勿使用 Deserialize(XmlReader)、Deserialize(XmlReader, String)、Deserialize(XmlReader, XmlDeserializationEvents) 或 Deserialize(XmlReader, String, XmlDeserializationEvents) 之外的 XmlSerializer.Deserialize 重载。

何时禁止显示警告

如果已解析的 XML 来自受信任的源，并且因此无法对其进行篡改，则可禁止显示此警告。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。XmlSerializer.Deserialize 的第一个参数的类型不是 `XmlReader` 或其派生类。

```
using System.IO;
using System.Xml.Serialization;
...
new XmlSerializer(typeof(TestClass).Deserialize(new FileStream("filename", FileMode.Open));
```

解决方案

```
using System.IO;
using System.Xml;
using System.Xml.Serialization;
...
new XmlSerializer(typeof(TestClass)).Deserialize(XmlReader.Create (new FileStream("filename",
FileMode.Open)));
```

CA5370：将 XmlReader 用于验证读取器

2021/11/16 •

■ ID	CA5370
■	安全性
修复是中断修复还是非中断修复	非中断

原因

验证具有未使用 `XmlReader` 对象实例化的 `XmlValidatingReader` 类的不受信任 XML 输入可能会导致拒绝服务、信息泄露和服务器端请求伪造。不受信任的 DTD 和 XML 架构处理使这些攻击成为可能，这使攻击者能在 XML 中包含 XML 炸弹和恶意的外部实体。只有使用 `XmlReader` 才可以禁用 DTD。从 .NET Framework 版本 4.0 开始，作为 `XmlReader` 的内联 XML 架构处理会默认将 `ProhibitDtd` 和 `ProcessInlineSchema` 属性设置为 `false`。

规则说明

处理不受信任的 DTD 和 XML 架构时可能会加载危险的外部引用。可以通过将 `XmlReader` 与安全解析程序结合使用，或者禁用 DTD 和 XML 内联架构处理来限制这种危险加载行为。此规则检测使用 `XmlValidatingReader` 类但不接受 `XmlReader` 作为构造函数参数的代码。

如何解决冲突

- 将 `XmlValidatingReader(XmlReader)` 和 `ProhibitDtd` 一起使用，并将 `ProcessInlineSchema` 属性设置为 `false`。
- 从 .NET Framework 2.0 开始，`XmlValidatingReader` 被视为已过时。可以使用 XmlReader.Create 实例化验证读取器。

何时禁止显示警告

如果始终将 `XmlValidatingReader` 用于验证来自受信任源的 XML，并且因此无法对其进行篡改，则可禁止显示此警告。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。`XmlValidatingReader.XmlValidatingReader()` 的第一个参数的类型不是 `XmlReader`。

```
using System;
using System.IO;
using System.Xml;
...
public void TestMethod(Stream xmlFragment, XmlNodeType fragType, XmlParserContext context)
{
    var obj = new XmlValidatingReader(xmlFragment, fragType, context);
}
```

解决方案

```
using System;
using System.Xml;
...
public void TestMethod(XmlReader xmlReader)
{
    var obj = new XmlValidatingReader(xmlReader);
}
```

CA5371：将 XmlReader 用于架构读取

2021/11/16 •

	[
■ ID	CA5371
■	安全性
修复是中断修复还是非中断修复	非中断

原因

通过未使用 `XmlReader` 对象进行实例化的 XmlSchema.Read 来处理不受信任的 XML 输入可能会导致拒绝服务、信息泄露和服务器端请求伪造攻击。不受信任的 DTD 和 XML 架构处理使这些攻击成为可能，这使攻击者能在 XML 中包含 XML 炸弹和恶意的外部实体。只有使用 `XmlReader` 才可以禁用 DTD。从 .NET Framework 版本 4.0 开始，作为 `XmlReader` 的内联 XML 架构处理会默认将 `ProhibitDtd` 和 `ProcessInlineSchema` 属性设置为 false。其他选项（例如 `Stream`、`TextReader` 和 `XmlSerializationReader`）不能禁用 DTD 处理。

规则说明

处理不受信任的 DTD 和 XML 架构时，可能会加载危险的外部引用。请使用具有安全解析程序或者禁用了 DTD 和 XML 内联架构处理的 `XmlReader` 对其进行限制。此规则检测使用 XmlSchema.Read 方法但不使用 `XmlReader` 作为参数的代码。

如何解决冲突

使用 `XmlSchema.Read(XmlReader, *)` 重载。

何时禁止显示警告

如果始终将 XmlSchema.Read 方法用于处理来自受信任源的 XML，并且因此无法对其进行篡改，则可以禁止显示此警告。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。XmlSchema.Read 的第一个参数的类型不是 `XmlReader`。

```
using System.IO;
using System.Xml.Schema;
...
public void TestMethod(Stream stream, ValidationEventHandler validationEventHandler)
{
    XmlSchema.Read(stream, validationEventHandler);
}
```

解决方案

```
using System.IO;
using System.Xml.Schema;
...
public void TestMethod(XmlReader reader, ValidationEventHandler validationEventHandler)
{
    XmlSchema.Read(reader, validationEventHandler);
}
```

CA5372：将 XmlReader 用于 XPathDocument

2021/11/16 •

	[
■ ID	CA5372
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用未通过 `XmlReader` 对象实例化的 `XPathDocument` 类可能会导致拒绝服务、信息泄露和服务器端请求伪造攻击。不受信任的 DTD 和 XML 架构处理使这些攻击成为可能，这使攻击者能在 XML 中包含 XML 炸弹和恶意的外部实体。只有使用 `XmlReader` 才可以禁用 DTD。从 .NET Framework 版本 4.0 开始，作为 `XmlReader` 的内联 XML 架构处理会默认将 `ProhibitDtd` 和 `ProcessInlineSchema` 属性设置为 false。其他选项（例如 `Stream`、`TextReader` 和 `XmlSerializationReader`）不能禁用 DTD 处理。

规则说明

处理来自不受信任的数据的 XML 时可能会加载危险的外部引用，可使用具有安全解析程序或禁用了 DTD 处理的 `XmlReader` 对其进行限制。此规则检测使用 `XPathDocument` 类但不接受 `XmlReader` 作为构造函数参数的代码。

如何解决冲突

使用 `XPathDocument(XmlReader, *)` 构造函数。

何时禁止显示警告

如果 `XPathDocument` 对象用于处理来自受信任源的 XML 文件并且因此无法对其进行篡改，则可禁止显示此警告。

伪代码示例

冲突

下面的伪代码示例演示了此规则可检测的情况。`XPathDocument` 的第一个参数的类型不是 `XmlReader`。

```
using System.IO;
using System.Xml.XPath;
...
var obj = new XPathDocument(stream);
```

解决方案

```
using System.Xml;
using System.Xml.XPath;
...
public void TestMethod(XmlReader reader)
{
var obj = new XPathDocument(reader);
}
```

CA5373：请勿使用已过时的密钥派生功能

2021/11/16 •

	▮
■ ID	CA5373
■	安全性
修复是中断修复还是非中断修复	非中断

原因

加密弱密钥派生方法 System.Security.Cryptography.PasswordDeriveBytes 和/或 Rfc2898DeriveBytes.CryptDeriveKey 用于生成密钥。

规则说明

此规则会检测弱密钥派生方法 System.Security.Cryptography.PasswordDeriveBytes 和 Rfc2898DeriveBytes.CryptDeriveKey 的调用。System.Security.Cryptography.PasswordDeriveBytes 使用了弱算法 PBKDF1。Rfc2898DeriveBytes.CryptDeriveKey 不使用 `Rfc2898DeriveBytes` 对象中的迭代计数和加盐，这会使其变弱。

如何解决冲突

基于密码的密钥派生应将 PBKDF2 算法与 SHA-2 哈希结合使用。Rfc2898DeriveBytes.GetBytes 可用于实现此目的。

何时禁止显示警告

如果仔细查看并接受与使用 PBKDF1 相关的风险，则可禁止显示此警告。

伪代码示例

冲突

到本文撰写时为止，下面的伪代码示例说明了此规则检测到的模式。

```
using System;
using System.Security.Cryptography;
class TestClass
{
    public void TestMethod(Rfc2898DeriveBytes rfc2898DeriveBytes, string algname, string alghashname, int
keySize, byte[] rgbIV)
    {
        rfc2898DeriveBytes.CryptDeriveKey(algname, alghashname, keySize, rgbIV);
    }
}
```

解决方案

```
using System;
using System.Security.Cryptography;
class TestClass
{
    public void TestMethod(Rfc2898DeriveBytes rfc2898DeriveBytes)
    {
        rfc2898DeriveBytes.GetBytes(1);
    }
}
```

CA5374：请勿使用 XslTransform

2021/11/16 •

	[
■ ID	CA5374
■	安全性
修复是中断修复还是非中断修复	非中断

原因

实例化 System.Xml.Xsl.XslTransform，它不会限制潜在的危险外部引用或阻止脚本。

规则说明

对不受信任的输入进行操作时，XslTransform 易受攻击。攻击可能会执行任意代码。

如何解决冲突

将 XslTransform 替换为 System.Xml.Xsl.XslCompiledTransform。有关更多指南，请参阅 [/dotnet/standard/data/xml/migrating-from-the-xsltransform-class]。

何时禁止显示警告

XslTransform 对象、XSLT 样式表和 XML 源数据都来自受信任的源。

伪代码示例

冲突

目前，下面的伪代码示例演示了此规则可检测的情况。

```
using System;
using System.Xml;
using System.Xml.XPath;
using System.Xml.Xsl;

namespace TestForXslTransform
{
    class Program
    {
        static void Main(string[] args)
        {
            // Create a new XslTransform object.
            XslTransform xslt = new XslTransform();

            // Load the stylesheet.
            xslt.Load("https://server/favorite.xsl");

            // Create a new XPathDocument and load the XML data to be transformed.
            XPathDocument mydata = new XPathDocument("inputdata.xml");

            // Create an XmlTextWriter which outputs to the console.
            XmlWriter writer = new XmlTextWriter(Console.Out);

            // Transform the data and send the output to the console.
            xslt.Transform(mydata, null, writer, null);
        }
    }
}
```

解决方案

```
using System.Xml;
using System.Xml.Xsl;

namespace TestForXslTransform
{
    class Program
    {
        static void Main(string[] args)
        {
            // Default XsltSettings constructor disables the XSLT document() function
            // and embedded script blocks.
            XsltSettings settings = new XsltSettings();

            // Execute the transform.
            XslCompiledTransform xslt = new XslCompiledTransform();
            xslt.Load("https://server/favorite.xsl", settings, new XmlUrlResolver());
            xslt.Transform("inputdata.xml", "outputdata.html");
        }
    }
}
```

CA5375:请勿使用帐户共享访问签名

2021/11/16 •

■ ID	CA5375
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 `Microsoft.WindowsAzure.Storage` 命名空间下的 `GetSharedAccessSignature` 方法生成帐户共享访问签名(SAS)。

规则说明

帐户 SAS 可以委派对 blob 容器、表、队列和文件共享执行读取、写入和删除操作的访问权限，这些是服务 SAS 不能实现的。但是，它不支持容器级别策略，并且其灵活性和对所授予权限的控制度不高。如果可能，请使用服务 SAS 实现精细访问控制。有关详细信息，请参阅使用共享访问签名委托访问权限。

如何解决冲突

使用服务 SAS 而不是帐户 SAS 实现精细访问控制和容器级别访问策略。

何时禁止显示警告

如果确定所有资源的权限都尽可能受到限制，则可禁止显示此规则的警告。

伪代码示例

冲突

目前，下面的伪代码示例演示了此规则可检测的情况。

```
using System;
using Microsoft.WindowsAzure.Storage;

class ExampleClass
{
    public void ExampleMethod(SharedAccessAccountPolicy policy)
    {
        CloudStorageAccount cloudStorageAccount = new CloudStorageAccount();
        cloudStorageAccount.GetSharedAccessSignature(policy);
    }
}
```

解决方案

请使用服务 SAS，而不是帐户 SAS。

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(StorageCredentials storageCredentials, SharedAccessFilePolicy policy,
SharedAccessFileHeaders headers, string groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(storageCredentials);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOnly;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

相关规则

CA5376:使用 SharedAccessProtocol HttpsOnly

CA5377:使用容器级别访问策略

CA5376:使用 SharedAccessProtocol HttpsOnly

2021/11/16 •

■ ID	CA5376
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 `Microsoft.WindowsAzure.Storage` 命名空间下的 `GetSharedAccessSignature` 方法生成共享访问签名 (SAS)，并将 `protocols` 指定为 `HttpsOrHttp`。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

SAS 是敏感数据，不能以纯文本形式在 HTTP 上传输。

如何解决冲突

生成 SAS 时使用 `HttpsOnly`。

何时禁止显示警告

请勿禁止显示此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的

`M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。

- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

目前，下面的伪代码示例演示了此规则可检测的情况。

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(SharedAccessFilePolicy policy, SharedAccessFileHeaders headers, string
groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(null);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOrHttp;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

解决方案

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.File;

class ExampleClass
{
    public void ExampleMethod(SharedAccessFilePolicy policy, SharedAccessFileHeaders headers, string
groupPolicyIdentifier, IPAddressOrRange ipAddressOrRange)
    {
        CloudFile cloudFile = new CloudFile(null);
        SharedAccessProtocol protocols = SharedAccessProtocol.HttpsOnly;
        cloudFile.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

相关规则

CA5375:请勿使用帐户共享访问签名

CA5377:使用容器级别访问策略

CA5377:使用容器级别访问策略

2021/11/16 •

■ ID	CA5377
■	安全性
修复是中断修复还是非中断修复	非中断

原因

生成服务共享访问签名 (SAS) 时，不会设置容器级别策略。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

可随时修改或撤销容器级别访问策略。它具有更高的灵活性，对授予的权限的控制力更强。有关详细信息，请参阅定义存储的访问策略。

如何解决冲突

在生成服务 SAS 时指定有效的组策略标识符。

何时禁止显示警告

如果确定所有资源的权限都尽可能受到限制，可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的

`M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。

- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

目前，下面的伪代码示例演示了此规则可检测的情况。

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;

class ExampleClass
{
    public void ExampleMethod(SharedAccessBlobPolicy policy, SharedAccessBlobHeaders headers,
Nullable<SharedAccessProtocol> protocols, IPAddressOrRange ipAddressOrRange)
    {
        var cloudAppendBlob = new CloudAppendBlob(null);
        string groupPolicyIdentifier = null;
        cloudAppendBlob.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

解决方案

```
using System;
using Microsoft.WindowsAzure.Storage;
using Microsoft.WindowsAzure.Storage.Blob;

class ExampleClass
{
    public void ExampleMethod(SharedAccessBlobPolicy policy, SharedAccessBlobHeaders headers,
Nullable<SharedAccessProtocol> protocols, IPAddressOrRange ipAddressOrRange)
    {
        CloudAppendBlob cloudAppendBlob = new CloudAppendBlob(null);
        string groupPolicyIdentifier = "123";
        cloudAppendBlob.GetSharedAccessSignature(policy, headers, groupPolicyIdentifier, protocols,
ipAddressOrRange);
    }
}
```

相关规则

CA5375:请勿使用帐户共享访问签名

CA5376:使用 SharedAccessProtocol HttpsOnly

CA5378：不禁用 ServicePointManagerSecurityProtocols

2021/11/16 •

	ı
■ ID	CA5378
■	安全性
修复是中断修复还是非中断修复	非中断

原因

AppContext.SetSwitch 方法调用将 `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols` 设置为 `true`。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

将 `Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols` 设置为 `true` 会将 Windows Communication Framework (WCF) 的传输层安全性 (TLS) 连接限制为使用 TLS 1.0。该版本的 TLS 将被弃用。有关详细信息，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法。

如何解决冲突

- 如果应用程序面向 .NET Framework v4.7 或更高版本，则可以删除 AppContext.SetSwitch 方法调用，或将开关的值设置为 `false`。
- 如果应用程序面向 .NET Framework v4.6.2 或更早版本，并在 .NET Framework v4.7 或更高版本上运行，则将开关的值设置为 `false`。
- 否则，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法来缓解问题。

何时禁止显示警告

如需连接到无法升级为使用安全 TLS 配置的旧服务，可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运

行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5378 violation
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
true);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        ' CA5378 violation
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
true)
    End Sub
End Class
```

解决方案

```
using System;

public class ExampleClass
{
    public void ExampleMethod()
    {
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
false);
    }
}
```

```
Imports System

Public Class ExampleClass
    Public Sub ExampleMethod()
        AppContext.SetSwitch("Switch.System.ServiceModel.DisableUsingServicePointManagerSecurityProtocols",
false)
    End Sub
End Class
```

CA5379：确保密钥派生功能算法足够强

2021/11/16 •

■ ID	CA5379
■	安全性
修复是中断修复还是非中断修复	非中断

原因

在实例化 System.Security.Cryptography.Rfc2898DeriveBytes 时使用以下算法之一：

- System.Security.Cryptography.MD5
- System.Security.Cryptography.SHA1
- 规则无法在编译时确定的算法

规则说明

Rfc2898DeriveBytes 类默认使用 SHA1 算法。在实例化 Rfc2898DeriveBytes 对象时，应指定 SHA256 或更高级别的哈希算法。注意，Rfc2898DeriveBytes.HashAlgorithm 属性只具有 `get` 访问器。

如何解决冲突

由于 MD5 或 SHA1 容易出现冲突，因此请对 Rfc2898DeriveBytes 类使用 SHA256 或更高级别。

利用旧版 .NET Framework 或 .NET Core，可能无法指定密钥派生功能哈希算法。在这种情况下，需要升级 .NET 的目标框架版本，以使用更强的算法。

何时禁止显示警告

不建议禁止显示此规则，除非为了应用程序兼容性原因。

伪代码示例

在构造函数冲突中指定哈希算法

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations, HashAlgorithmName.MD5);
    }
}
```

在派生类的构造函数冲突中指定哈希算法

```
using System.Security.Cryptography;

class DerivedClass : Rfc2898DeriveBytes
{
    public DerivedClass (byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm) :
base(password, salt, iterations, hashAlgorithm)
    {
    }
}

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var derivedClass = new DerivedClass(password, salt, iterations, HashAlgorithmName.MD5);
    }
}
```

在派生类冲突中设置哈希算法属性

```
using System.Security.Cryptography;

class DerivedClass : Rfc2898DeriveBytes
{
    public DerivedClass (byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm) :
base(password, salt, iterations, hashAlgorithm)
    {
    }

    public HashAlgorithmName HashAlgorithm { get; set;}
}

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var derivedClass = new DerivedClass(password, salt, iterations, HashAlgorithmName.MD5);
        derivedClass.HashAlgorithm = HashAlgorithmName.SHA256;
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] password, byte[] salt, int iterations, HashAlgorithmName hashAlgorithm)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations,
HashAlgorithmName.SHA256);
    }
}
```

CA5380：请勿将证书添加到根存储中

2021/11/16 •

■ ID	CA5380
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将证书添加到操作系统的受信任的根证书会增加将不受信任的证书颁发机构合法化的风险。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

此规则会对将证书添加到"受信任的根证书颁发机构"证书存储的代码进行检测。默认情况下，"受信任的根证书颁发机构"证书存储配置有一组符合 Microsoft 根证书计划要求的公共 CA。由于所有受信任的根证书颁发机构 (CA) 都可为任意域颁发证书，因此攻击者可能会选择你自行安装的某个安全性较弱或可强迫的 CA 进行攻击；一个易受攻击、恶意或可强迫的 CA 会降低整个系统的安全性。

如何解决冲突

请勿将证书安装到"受信任的根证书颁发机构"证书存储中。

何时禁止显示警告

不建议禁止显示此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

下面的伪代码示例演示了此规则检测到的模式。

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.Root;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

解决方案

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.My;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

CA5381：请确保证书未添加到根存储中

2021/11/16 •

	‖
■ ID	CA5381
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将证书添加到操作系统的受信任的根证书会增加将不受信任的证书颁发机构合法化的风险。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

此规则会对可能将证书添加到"受信任的根证书颁发机构"证书存储的代码进行检测。默认情况下，"受信任的根证书颁发机构"证书存储配置有一组符合 Microsoft 根证书计划要求的公共证书颁发机构 (CA)。由于所有受信任的根 CA 都可为任意域颁发证书，因此攻击者可能会选择你自行安装的某个安全性较弱或可强迫的 CA 进行攻击；一个易受攻击、恶意或可强迫的 CA 会降低整个系统的安全性。

如何解决冲突

请勿将证书安装到"受信任的根证书颁发机构"证书存储中。

何时禁止显示警告

不建议禁止显示此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

下面的伪代码示例演示了此规则检测到的模式。

```
using System;
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.Root;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            storeName = StoreName.My;
        }

        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

解决方案

```
using System.Security.Cryptography.X509Certificates;

class TestClass
{
    public void TestMethod()
    {
        var storeName = StoreName.My;
        var x509Store = new X509Store(storeName);
        x509Store.Add(new X509Certificate2());
    }
}
```

CA5382:在 ASP.NET Core 中使用安全 Cookie

2021/11/16 •

	[
■ ID	CA5382
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用 Microsoft.AspNetCore.Http.IResponseCookies.Append 时，Microsoft.AspNetCore.Http.CookieOptions.Secure 属性设置为 `false`。现在，此规则仅查看 Microsoft.AspNetCore.Http.Internal.ResponseCookies 类，这是 IResponseCookies 的其中一个实现。

此规则类似于 CA5383，但分析可以确定 Secure 属性是否一定为 `false` 或未设置。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用传输层安全性 (TLS) 进行传输。

如何解决冲突

将 Secure 属性设置为 `true`。

何时禁止显示警告

- Cookie 默认配置为安全时，例如在 `Startup.Configure` 中使用 Microsoft.AspNetCore.CookiePolicy.CookiePolicyMiddleware：

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseCookiePolicy(
            new CookiePolicyOptions
            {
                Secure = CookieSecurePolicy.Always
            });
    }
}
```

- 确定 Cookie 中没有敏感数据时。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

示例

下面的代码片段演示了此规则可检测的情况。

冲突：

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = false;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

解决方案：

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = true;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

CA5383:确保在 ASP.NET Core 中使用安全 Cookie

2021/11/16 •

	ı
■ ID	CA5383
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用 Microsoft.AspNetCore.Http.IResponseCookies.Append 时，可以将 Microsoft.AspNetCore.Http.CookieOptions.Secure 属性设置为 `false`。现在，此规则仅查看 Microsoft.AspNetCore.Http.Internal.ResponseCookies 类，这是 IResponseCookies 的其中一个实现。

此规则类似于 CA5382，但分析无法确定 Secure 属性是否一定为 `false` 或未设置。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

HTTPS 上可用的应用程序必须使用安全 Cookie，这会向浏览器指示，Cookie 只能使用传输层安全性 (TLS) 进行传输。

如何解决冲突

在所有情况下，将 Secure 属性设置为 `true`。

何时禁止显示警告

- Cookie 默认配置为安全时，例如在 `Startup.Configure` 中使用 Microsoft.AspNetCore.CookiePolicy.CookiePolicyMiddleware：

```
public class Startup
{
    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        app.UseCookiePolicy(
            new CookiePolicyOptions
            {
                Secure = CookieSecurePolicy.Always
            });
    }
}
```

- 确定 Cookie 中没有敏感数据时。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

示例

下面的代码片段演示了此规则可检测的情况。

冲突:

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = false;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            cookieOptions.Secure = true;
        }

        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

解决方案:

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Internal;

class ExampleClass
{
    public void ExampleMethod(string key, string value)
    {
        var cookieOptions = new CookieOptions();
        cookieOptions.Secure = true;
        var responseCookies = new ResponseCookies(null, null);
        responseCookies.Append(key, value, cookieOptions);
    }
}
```

CA5384:不使用数字签名算法(DSA)

2021/11/16 •

■ ID	CA5384
■	安全性
修复是中断修复还是非中断修复	非中断

原因

通过以下某个方式使用 DSA:

- 返回或实例化 System.Security.Cryptography.DSA 的派生类
- 使用 System.Security.Cryptography.AsymmetricAlgorithm.Create 或 System.Security.Cryptography.CryptoConfig.CreateFromName 来创建 DSA 对象。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

DSA 是一种弱非对称加密算法。

如何解决冲突

请改为使用密钥大小至少为 2048 的 RSA、ECDH 或 ECDsa 算法。

何时禁止显示警告

不建议禁止显示此规则的警告，除非与旧版应用程序和数据兼容。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件:

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

示例

下面的代码片段演示了此规则可检测的情况。

冲突：

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        DSACng dsaCng = new DSACng();
    }
}
```

解决方案:

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AsymmetricAlgorithm asymmetricAlgorithm = AsymmetricAlgorithm.Create("ECDsa");
    }
}
```

CA5385:设置具有足够密钥大小的 Rivest–Shamir–Adleman (RSA)算法

2021/11/16 •

■ ID	CA5385
■	安全性
修复是中断修复还是非中断修复	非中断

原因

通过以下方法使用密钥大小小于 2048 的非对称加密算法 RSA：

- 实例化 System.Security.Cryptography.RSA 的所有后代类，并将 `KeySize` 参数指定为小于 2048。
- 返回类型是 System.Security.Cryptography.RSA 的后代的任何对象。
- 使用不带参数的 System.Security.Cryptography.AsymmetricAlgorithm.Create，这会创建默认密钥大小为 1024 的 RSA。
- 使用 System.Security.Cryptography.AsymmetricAlgorithm.Create 并将 `algName` 参数指定为默认密钥大小为 1024 的 `RSA`。
- 使用 System.Security.Cryptography.CryptoConfig.CreateFromName 并将 `name` 参数指定为默认密钥大小为 1024 的 `RSA`。
- 使用 System.Security.Cryptography.CryptoConfig.CreateFromName 并将 `name` 参数指定为 `RSA`，然后通过 `args` 将密钥大小显式指定为小于 2048。

规则说明

小于 2048 位的 RSA 密钥更容易受到暴力攻击。

如何解决冲突

请改为使用密钥大小至少为 2048 的 RSA、ECDH 或 ECDsa 算法。

何时禁止显示警告

不建议禁止显示此规则，除非是为了与旧版应用程序和数据兼容。

示例

下面的代码片段演示了此规则可检测的情况。

冲突：

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        RSACng rsaCng = new RSACng(1024);
    }
}
```

解决方案:

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        RSACng rsaCng = new RSACng(2048);
    }
}
```

CA5386：避免对 SecurityProtocolType 值进行硬编码

2021/11/16 •

■ ID	CA5386
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果满足以下任一条件，则会触发此规则：

- 引用了安全但硬编码的 System.Net.SecurityProtocolType 值。
- 对 SecurityProtocolType 变量赋予了代表安全协议版本的整数值。

安全值为：

- Tls12
- Tls13

规则说明

传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。TLS 1.2 和 TLS 1.3 将来可能也会弃用。要确保应用程序的安全性，请避免对协议版本进行硬编码，并且至少以 .NET Framework v4.7.1 为目标。有关详细信息，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法。

如何解决冲突

不要对 TLS 协议版本进行硬编码。

何时禁止显示警告

如果你的应用程序以 .NET Framework v4.6.2 或更早版本为目标，并且可能在具有不安全默认值的计算机上运行，则可禁止显示此警告。

伪代码示例

枚举名称冲突

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5386 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5386 violation
        ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12
    End Sub
End Class
```

整数值冲突

```
using System;
using System.Net;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5386 violation
        ServicePointManager.SecurityProtocol = (SecurityProtocolType) 3072;    // TLS 1.2
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5386 violation
        ServicePointManager.SecurityProtocol = CType(3072, SecurityProtocolType)   ' TLS 1.2
    End Sub
End Class
```

解决方案

```
using System;
using System.Net;

public class TestClass
{
    public void TestMethod()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    }
}
```

```
Imports System
Imports System.Net

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
    End Sub
End Class
```

相关规则

CA5364：不使用已弃用的安全协议

CA5397：不使用已弃用的 SslProtocols 值

CA5398：避免硬编码的 SslProtocols 值

CA5387:请勿使用迭代计数不足的弱密钥派生功能

2021/11/16 •

	‹
■ ID	CA5387
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用具有默认迭代计数的 System.Security.Cryptography.Rfc2898DeriveBytes.GetBytes 或指定小于 100,000 的迭代计数。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

此规则检查加密密钥是否由迭代计数小于 100,000 的 Rfc2898DeriveBytes 生成。较高的迭代计数有助于缓解尝试猜测已生成的加密密钥的字典攻击。

此规则类似于 CA5388，但分析确定迭代计数小于 100,000。

如何解决冲突

在调用 GetBytes 之前，将迭代计数设置为大于或等于 100,000。

何时禁止显示警告

如果需要使用较小的迭代计数以与现有数据兼容。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

“	“
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

默认迭代计数冲突

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

在构造函数冲突中指定迭代计数

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, 100);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

按属性分配冲突指定迭代计数

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100000;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

CA5388:使用弱密钥派生功能时，请确保迭代计数足够大

2021/11/16 •

■ ID	CA5388
■	安全性
修复是中断修复还是非中断修复	非中断

原因

通过 System.Security.Cryptography.Rfc2898DeriveBytes.GetBytes 派生加密密钥时，迭代计数可能小于 100,000。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

此规则检查加密密钥是否由迭代计数小于 100,000 的 Rfc2898DeriveBytes 生成。较高的迭代计数有助于缓解尝试猜测已生成的加密密钥的字典攻击。

此规则类似于 CA5387，但分析无法确定迭代计数是否小于 100,000。

如何解决冲突

在显式调用 GetBytes 之前，将迭代计数设置为大于或等于 100k。

何时禁止显示警告

在以下情况下，可禁止显示此规则的警告：

- 需要使用较小的迭代计数以与现有数据兼容。
- 确定迭代计数已设置为大于 100,000。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅符号名称(包括具有相应名称的所有符号,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀,例如表示方法的 `M:`、表示类型的 `T:`,以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数,`.cctor` 表示静态构造函数。

示例:

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如,若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行,请将以下键值对添加到项目中的 .editorconfig 文件:

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅类型名称(包括具有相应名称的所有类型,不考虑包含的类型或命名空间)。
- 完全限定的名称,使用符号的文档 ID 格式,前缀为 `T:`(可选)。

示例:

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

冲突

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var iterations = 100;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            iterations = 100000;
        }

        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt, iterations);
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(string password, byte[] salt, int cb)
    {
        var rfc2898DeriveBytes = new Rfc2898DeriveBytes(password, salt);
        rfc2898DeriveBytes.IterationCount = 100000;
        rfc2898DeriveBytes.GetBytes(cb);
    }
}
```

CA5389：请勿将存档项的路径添加到目标文件系统路径中

2021/11/16 •

■ ID	CA5389
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将非净化的源文件路径用作以下参数之一中的目标文件路径：

- 方法 ZipFileExtensions.ExtractToFile 的参数 `destinationFileName`
- 方法 File.Open 的参数 `path`
- 方法 File.OpenWrite 的参数 `path`
- 方法 File.Create 的参数 `path`
- FileStream 的构造函数的参数 `path`
- FileInfo 的构造函数的参数 `fileName`

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

文件路径可以是相对的，并且可能导致文件系统访问预期文件系统目标路径以外的内容，从而导致攻击者通过“布局和等待”技术恶意更改配置和执行远程代码。

如何解决冲突

不要使用源文件路径来构造目标文件路径，或确保提取路径上的最后一个字符是目录分隔符。

何时禁止显示警告

如果源路径始终来自受信任的源，则可以禁止显示此警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

‘‘‘	‘‘

示例

下面的代码片段演示了此规则可检测的情况。

冲突：

```
using System.IO.Compression;

class TestClass
{
    public void TestMethod(ZipArchiveEntry zipArchiveEntry)
    {
        zipArchiveEntry.ExtractToFile(zipArchiveEntry.FullName);
    }
}
```

解决方案：

```
using System;
using System.IO;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string zipPath = @".\result.zip";

        Console.WriteLine("Provide path where to extract the zip file:");
        string extractPath = Console.ReadLine();

        // Normalizes the path.
        extractPath = Path.GetFullPath(extractPath);

        // Ensures that the last character on the extraction path
        // is the directory separator char.
        // Without this, a malicious zip file could try to traverse outside of the expected
        // extraction path.
        if (!extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(), StringComparison.Ordinal))
            extractPath += Path.DirectorySeparatorChar;

        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                {
                    // Gets the full path to ensure that relative segments are removed.
                    string destinationPath = Path.GetFullPath(Path.Combine(extractPath, entry.FullName));

                    // Ordinal match is safest, case-sensitive volumes can be mounted within volumes that
                    // are case-insensitive.
                    if (destinationPath.StartsWith(extractPath, StringComparison.Ordinal))
                        entry.ExtractToFile(destinationPath);
                }
            }
        }
    }
}
```

CA5390:请勿编码加密密钥

2021/11/16 •

	[
■ ID	CA5390
■	安全性
修复是中断修复还是非中断修复	非中断

原因

System.Security.Cryptography.AesCcm 或 System.Security.Cryptography.AesGcm 构造函数的 `key` 参数、System.Security.Cryptography.SymmetricAlgorithm.Key 属性，或者 System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor 或 System.Security.Cryptography.SymmetricAlgorithm.CreateDecryptor 方法的 `rgbKey` 参数由以下其中一项硬编码：

- 字节数组。
- System.Convert.FromBase64String.
- System.Text.Encoding.GetBytes 的所有重载。

默认情况下，此规则会分析整个代码库，但这是可配置的。

规则说明

要成功使用对称算法，密钥必须只有发送方和接收方知道。如果密钥是硬编码的，就容易被发现。即使使用编译的二进制文件，恶意用户也容易将其提取出来。私钥泄露后，密码文本可直接被解密并且不再受保护。

如何解决冲突

- 考虑重新设计应用程序，以使用安全的密钥管理系统，如 Azure Key Vault。
- 将凭据和密钥保留在安全位置，与源代码分开。

何时禁止显示警告

不禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称（包括具有相应名称的所有符号，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅类型名称（包括具有相应名称的所有类型，不考虑包含的类型或命名空间）。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`（可选）。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

‘‘‘	‘‘

伪代码示例

硬编码字节数组冲突

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] rgbKey = new byte[] {1, 2, 3};
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(rgbKey, someOtherBytesForIV);
    }
}
```

硬编码 Convert.FromBase64String 冲突

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] key = Convert.FromBase64String("AAAAAaazaoensuth");
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

硬编码 Encoding.GetBytes 冲突

```
using System.Text;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] someOtherBytesForIV)
    {
        byte[] key = Encoding.ASCII.GetBytes("AAAAAaazaoensuth");
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

解决方案

```
using System.Text;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(char[] chars, byte[] someOtherBytesForIV)
    {
        byte[] key = Encoding.ASCII.GetBytes(chars);
        SymmetricAlgorithm rijn = SymmetricAlgorithm.Create();
        rijn.CreateEncryptor(key, someOtherBytesForIV);
    }
}
```

CA5391:在 ASP.NET Core MVC 控制器中使用防伪造令牌

2021/11/16 •

■ ID	CA5391
■	安全性
修复是中断修复还是非中断修复	非中断

原因

导致修改操作的操作没有防伪造令牌特性。或者，使用全局防伪造令牌筛选器，而不调用预期的防伪造令牌函数。

规则说明

处理 `POST`、`PUT`、`PATCH` 或 `DELETE` 请求而不验证防伪造令牌可能易受到跨网站请求伪造攻击。跨网站请求伪造攻击可将经过身份验证的用户的恶意请求发送到 ASP.NET Core MVC 控制器。

如何解决冲突

- 使用有效的防伪造令牌特性标记修改操作：
  - Microsoft.AspNetCore.Mvc.ValidateAntiForgeryTokenAttribute.
  - 名称类似 `%Validate%Anti_orgery%Attribute` 的特性。
- 使用 Microsoft.AspNetCore.Mvc.Filters.FilterCollection.Add 将有效的伪造令牌特性添加到全局筛选器中。
- 添加任何自定义的或 MVC 提供的防伪造筛选器类，该类针对任何实现 Microsoft.AspNetCore.Antiforgery.IAntiforgery 接口的类调用 `Validate`。

何时抑制警告

如果采用除了使用防伪造令牌特性之外的其他解决方案来缓解 CSRF 漏洞，则可安全地抑制该规则。有关详细信息，请参阅在 ASP.NET Core 中预防跨网站请求伪造 (XSRF/CSRF) 攻击。

配置代码以进行分析

可配置该规则是否仅适用于代码库中 Microsoft.AspNetCore.Mvc.Controller 的派生类。例如，若要指定规则不应针对 ControllerBase 的派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CA5391.exclude_aspnet_core_mvc_controllerbase = true
```

伪代码示例

无防伪造令牌特性冲突

```
using Microsoft.AspNetCore.Mvc;

class ExampleController : Controller
{
    [HttpDelete]
    public IActionResult ExampleAction (string actionName)
    {
        return null;
    }

    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction (string actionName)
    {
        return null;
    }
}
```

无有效的全局防伪造筛选器

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction (string actionName)
    {
        return null;
    }

    [HttpDelete]
    public IActionResult ExampleAction (string actionName)
    {
        return null;
    }
}

class FilterClass : IAsyncAuthorizationFilter
{
    public Task OnAuthorizationAsync (AuthorizationFilterContext context)
    {
        return null;
    }
}

class BlahClass
{
    public static void BlahMethod ()
    {
        FilterCollection filterCollection = new FilterCollection ();
        filterCollection.Add(typeof(FilterClass));
    }
}
```

使用防伪造令牌特性解决方案进行了标记

```
using Microsoft.AspNetCore.Mvc;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult ExampleAction ()
    {
        return null;
    }

    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction ()
    {
        return null;
    }
}
```

使用有效的全局防伪造筛选器

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Antiforgery;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;

class ExampleController : Controller
{
    [ValidateAntiForgeryToken]
    [HttpDelete]
    public IActionResult AnotherAction()
    {
        return null;
    }

    [HttpDelete]
    public IActionResult ExampleAction()
    {
        return null;
    }
}

class FilterClass : IAsyncAuthorizationFilter
{
    private readonly IAntiforgery antiforgery;

    public FilterClass(IAntiforgery antiforgery)
    {
        this.antiforgery = antiforgery;
    }

    public Task OnAuthorizationAsync(AuthorizationFilterContext context)
    {
        return antiforgery.ValidateRequestAsync(context.HttpContext);
    }
}

class BlahClass
{
    public static void BlahMethod()
    {
        FilterCollection filterCollection = new FilterCollection();
        filterCollection.Add(typeof(FilterClass));
    }
}
```

CA5392:对 P/Invoke 使用 DefaultDllImportSearchPaths 属性

2021/11/16 •

■ ID	CA5392
■	安全性
修复是中断修复还是非中断修复	非中断

原因

没有为平台调用 (P/Invoke) 函数指定 DefaultDllImportSearchPathsAttribute。

规则说明

默认情况下，使用 DllImportAttribute 的 P/Invoke 函数会探测大量目录，包括要加载的库的当前工作目录。这对于某些应用程序来说是一个安全隐患，会导致 DLL 劫持。

例如，如果将与导入的 DLL 同名的恶意 DLL 置于当前工作目录下，由于默认情况下将首先搜索该目录，因此随后就会加载该恶意 DLL。

有关详细信息，请参阅安全加载库。

如何解决冲突

使用 DefaultDllImportSearchPathsAttribute 显式指定用于程序集或方法的 DLL 搜索路径。

何时禁止显示警告

在以下情况下，可禁止显示此规则的警告：

- 确定已加载的程序集是所需的程序集。例如，你的应用程序在受信任的服务器上运行，并且你完全信任这些文件。
- 导入的程序集是常用的系统程序集（如 user32.dll），并且搜索路径策略遵循已知的 DLL 机制。

伪代码示例

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

解决方案

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.UserDirectories)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

相关规则

CA5393:请勿使用不安全的 DllImportSearchPath 值

CA5393:请勿使用不安全的 DllImportSearchPath 值

2021/11/16 •

	‹
■ ID	CA5393
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 <xref:System.Runtime.InteropServices.DllImportSearchPath?displayProperty=fullName 的其中一个不安全值：

- `AssemblyDirectory`
- `UseDllDirectoryForDependencies`
- `ApplicationDirectory`

规则说明

默认的 DLL 搜索目录和程序集目录中可能存在恶意 DLL。或者根据应用程序运行的位置，应用程序的目录中可能存在恶意 DLL。

有关详细信息，请参阅安全加载库。

如何解决冲突

改为使用 DllImportSearchPath 的安全值来指定显式搜索路径：

- `LegacyBehavior`
- `SafeDirectories`
- `System32`
- `UserDirectories`

何时禁止显示警告

在以下情况下，可禁止显示此规则的警告：

- 确定已加载的程序集是所需的程序集。
- 导入的程序集是常用的系统程序集（如 user32.dll），并且搜索路径策略遵循已知的 DLL 机制。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 不安全的 DllImportSearchPath 位

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置此选项。有关详细信息，请参阅代码质量

规则配置选项。

不安全的 DllImportSearchPath 位

你可以配置 DllImportSearchPath 的哪个值对于分析是不安全的。例如，若要指定代码不应使用 `AssemblyDirectory`、`UseDllDirectoryForDependencies` 或 `ApplicationDirectory`，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CA5393.unsafe_DllImportSearchPath_bits = 770
```

应指定枚举值的按位组合的整数值。

伪代码示例

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.AssemblyDirectory)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

解决方案

```
using System;
using System.Runtime.InteropServices;

class ExampleClass
{
    [DllImport("The3rdAssembly.dll")]
    [DefaultDllImportSearchPaths(DllImportSearchPath.UserDirectories)]
    public static extern int MessageBox(IntPtr hWnd, String text, String caption, uint type);

    public void ExampleMethod()
    {
        MessageBox(new IntPtr(0), "Hello World!", "Hello Dialog", 0);
    }
}
```

相关规则

CA5392:对 P/Invoke 使用 DefaultDllImportSearchPaths 属性

CA5394:请勿使用不安全的随机性

2021/11/16 •

	‹
■ ID	CA5394
■	安全性
修复是中断修复还是非中断修复	非中断

原因

调用了其中一个 System.Random 方法。

规则说明

如果使用加密较弱的伪随机数生成器，攻击者可以预测将要生成的安全敏感值。

如何解决冲突

如果需要不可预测的值以确保安全，请使用 System.Security.Cryptography.RandomNumberGenerator 或 System.Security.Cryptography.RNGCryptoServiceProvider 等加密较强的随机数生成器。

何时禁止显示警告

如果你确定弱伪随机数不是以安全敏感的方式使用，则可禁止显示此规则的警告。

伪代码示例

冲突

```
using System;

class ExampleClass
{
    public void ExampleMethod(Random random)
    {
        var sensitiveVariable = random.Next();
    }
}
```

解决方案

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(int toExclusive)
    {
        var sensitiveVariable = RandomNumberGenerator.GetInt32(toExclusive);
    }
}
```

CA5395:缺少操作方法的 HttpVerb 属性

2021/11/16 •

■ ID	CA5395
■	安全性
修复是中断修复还是非中断修复	非中断

原因

未为操作方法显式指定 HTTP 请求的类型。

规则说明

创建、编辑或以其它方式修改数据等所有操作方法都需要使用防伪特性来保护，以避免受跨网站请求伪造攻击的影响。执行 GET 操作应是没有副作用且不会修改持久数据的安全操作。

如何解决冲突

使用 `HttpVerb` 属性标记操作方法。

何时禁止显示警告

在以下情况下，可安全地禁止显示此规则的警告：

- 确定操作方法中未发生修改操作。或者，它根本就不是操作方法。
- 采用除使用防伪造令牌属性之外的其他解决方案来缓解 CSRF 漏洞。有关详细信息，请参阅在 ASP.NET Core 中预防跨网站请求伪造 (XSRF/CSRF) 攻击。

伪代码示例

冲突

```
using Microsoft.AspNetCore.Mvc;

[ValidateAntiForgeryToken]
class BlahController : Controller
{
}

class ExampleController : Controller
{
    public IActionResult ExampleAction()
    {
        return null;
    }
}
```

解决方案

```
using Microsoft.AspNetCore.Mvc;

[ValidateAntiForgeryToken]
class BlahController : Controller
{
}

class ExampleController : Controller
{
    [HttpGet]
    public IActionResult ExampleAction()
    {
        return null;
    }
}
```

CA5396:将 HttpCookie 的 HttpOnly 设置为 true

2021/11/16 •

	[
■ ID	CA5396
■	安全性
修复是中断修复还是非中断修复	非中断

原因

System.Web.HttpCookie.HttpOnly 设置为 `false`。此属性的默认值为 `false`。

规则说明

请确保将安全敏感的 HTTP Cookie 标记为 HttpOnly，这是一个深度防御措施。这表明 Web 浏览器应禁止脚本访问 Cookie。注入恶意脚本是窃取 Cookie 的常见方式。

如何解决冲突

将 System.Web.HttpCookie.HttpOnly 设置为 `true`。

何时禁止显示警告

- 如果设置了全局值 HttpOnly，如以下示例中所示：

```
<system.web>
    ...
    <httpCookies httpOnlyCookies="true" requireSSL="true" />
</system.web>
```

- 如果确定 Cookie 中没有敏感数据。

示例

冲突：

```
using System.Web;

class ExampleClass
{
    public void ExampleMethod()
    {
        HttpCookie httpCookie = new HttpCookie("cookieName");
        httpCookie.HttpOnly = false;
    }
}
```

解决方案：

```
using System.Web;

class ExampleClass
{
    public void ExampleMethod()
    {
        HttpCookie httpCookie = new HttpCookie("cookieName");
        httpCookie.HttpOnly = true;
    }
}
```

CA5397：不使用已弃用的 SslProtocols 值

2021/11/16 •

■ ID	CA5397
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果满足以下任一条件，则会触发此规则：

- 引用了已弃用的 System.Security.Authentication.SslProtocols 值。
- 表示已弃用值的整数值要么已分配给 SslProtocols 变量，要么用作 SslProtocols 返回值或用作 SslProtocols 参数。

已弃用的值包括：

- Ssl2
- Ssl3
- Tls
- Tls10
- Tls11

规则说明

传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。早期版本的 TLS 协议不如 TLS 1.2 和 TLS 1.3 安全，且更容易出现新的漏洞。避免使用旧版本的协议，以便最大程度降低风险。有关标识和删除已弃用协议版本的指导，请参阅解决 TLS 1.0 问题（第 2 版）。

如何解决冲突

请勿使用已弃用的 TLS 协议版本。

何时禁止显示警告

对于以下情况，可禁止显示此警告：

- 未使用对已弃用协议版本的引用来启用已弃用的版本。
- 连接到无法升级为使用安全 TLS 配置的旧服务。

伪代码示例

枚举名称冲突

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5397 violation for using Tls11
        SslProtocols protocols = SslProtocols.Tls11 | SslProtocols.Tls12;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5397 violation for using Tls11
        Dim sslProtocols As SslProtocols = SslProtocols.Tls11 Or SslProtocols.Tls12
    End Sub
End Class
```

整数值冲突

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5397 violation
        SslProtocols sslProtocols = (SslProtocols) 768;    // TLS 1.1
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' CA5397 violation
        Dim sslProtocols As SslProtocols = CType(768, SslProtocols)   ' TLS 1.1
    End Sub
End Class
```

解决方案

```
using System;
using System.Security.Authentication;

public class TestClass
{
    public void Method()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        SslProtocols sslProtocols = SslProtocols.None;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        Dim sslProtocols As SslProtocols = SslProtocols.None
    End Sub
End Class
```

相关规则

CA5364：不使用已弃用的安全协议

CA5386：避免对 SecurityProtocolType 值进行硬编码

CA5398：避免硬编码的 SslProtocols 值

CA5398：避免硬编码的 SslProtocols 值

2021/11/16 •

■ ID	CA5398
■	安全性
修复是中断修复还是非中断修复	非中断

原因

如果满足以下任一条件，则会触发此规则：

- 引用了安全但硬编码的 System.Security.Authentication.SslProtocols 值。
- 表示安全协议版本的整数值已赋值给 SslProtocols 变量，用作 SslProtocols 返回值，或用作 SslProtocols 参数。

安全值为：

- Tls12
- Tls13

规则说明

传输层安全性 (TLS) 通常使用安全超文本传输协议 (HTTPS) 保障计算机之间的通信安全。协议版本 TLS 1.0 和 TLS 1.1 已弃用，目前使用 TLS 1.2 和 TLS 1.3。将来可能也会弃用 TLS 1.2 和 TLS 1.3。要确保应用程序的安全性，请避免对协议版本进行硬编码。有关详细信息，请参阅 .NET Framework 中的传输层安全性 (TLS) 最佳做法。

如何解决冲突

不要对 TLS 协议版本进行硬编码。

何时禁止显示警告

需要连接到无法升级为使用将来的 TLS 协议版本的旧服务。

伪代码示例

枚举名称冲突

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public void ExampleMethod()
    {
        // CA5398 violation
        SslProtocols sslProtocols = SslProtocols.Tls12;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Function ExampleMethod() As SslProtocols
        ' CA5398 violation
        Return SslProtocols.Tls12
    End Function
End Class
```

整数值冲突

```
using System;
using System.Security.Authentication;

public class ExampleClass
{
    public SslProtocols ExampleMethod()
    {
        // CA5398 violation
        return (SslProtocols) 3072;    // TLS 1.2
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Function ExampleMethod() As SslProtocols
        ' CA5398 violation
        Return CType(3072, SslProtocols)   ' TLS 1.2
    End Function
End Class
```

解决方案

```
using System;
using System.Security.Authentication;

public class TestClass
{
    public void Method()
    {
        // Let the operating system decide what TLS protocol version to use.
        // See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        SslProtocols sslProtocols = SslProtocols.None;
    }
}
```

```
Imports System
Imports System.Security.Authentication

Public Class TestClass
    Public Sub ExampleMethod()
        ' Let the operating system decide what TLS protocol version to use.
        ' See https://docs.microsoft.com/dotnet/framework/network-programming/tls
        Dim sslProtocols As SslProtocols = SslProtocols.None
    End Sub
End Class
```

相关规则

CA5364：不使用已弃用的安全协议

CA5386：避免对 SecurityProtocolType 值进行硬编码

CA5397：不使用已弃用的 SslProtocols 值

CA5399：启用 HttpClient 证书吊销列表检查

2021/11/16 •

■ ID	CA5399
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 System.Net.Http.HttpClient 的同时提供 `CheckCertificateRevocationList` 属性未设置为 `true` 的平台特定处理程序（System.Net.Http.WinHttpHandler 或 System.Net.Http.HttpClientHandler），将允许 HttpClient 将撤销的证书视为有效而接受。

此规则类似于 CA5400，但分析可以确定 `CheckCertificateRevocationList` 属性是否一定为 `false` 或未设置。

规则说明

撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。

如何解决冲突

将 System.Net.Http.HttpClientHandler.CheckCertificateRevocationList 属性显式设置为 `true`。如果 CheckCertificateRevocationList 属性不可用，则需要升级目标框架。

何时禁止显示警告

请勿禁止显示此规则。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别（安全性）中的所有规则配置这些选项。有关详细信息，请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号，如类型和方法。例如，若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式（用 `|` 分隔）：

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = false;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

解决方案

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = true;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

CA5400:确保未禁用 HttpClient 证书吊销列表检查

2021/11/16 •

	[
■ ID	CA5400
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 System.Net.Http.HttpClient 的同时提供 `CheckCertificateRevocationList` 属性可能设置为 `false` 的平台特定处理程序(System.Net.Http.WinHttpHandler 或 System.Net.Http.HttpClientHandler),将允许被吊销的证书被 HttpClient 视为有效而接受。

此规则类似于 CA5399,但分析无法确定 `CheckCertificateRevocationList` 属性是否一定为 `false` 或未设置。

规则说明

撤销的证书不再受信任。攻击者可能使用它来传递某些恶意数据或窃取 HTTPS 通信中的敏感数据。

如何解决冲突

将 System.Net.Http.HttpClientHandler.CheckCertificateRevocationList 属性显式设置为 `true`。如果 CheckCertificateRevocationList 属性不可用,则需要升级目标框架。

何时禁止显示警告

如果确定 `CheckCertificateRevocationList` 属性设置正确,则可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 排除特定符号
- 排除特定类型及其派生类型

你可以仅为此规则、为所有规则或为此类别(安全性)中的所有规则配置此选项。有关详细信息,请参阅代码质量规则配置选项。

排除特定符号

可以从分析中排除特定符号,如类型和方法。例如,若要指定规则不应针对名为 `MyType` 的类型中的任何代码运行,请将以下键值对添加到项目中的 .editorconfig 文件:

```
dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔):

- 仅符号名称(包括具有相应名称的所有符号，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式。每个符号名称都需要带有一个符号类型前缀，例如表示方法的 `M:`、表示类型的 `T:`，以及表示命名空间的 `N:`。
- `.ctor` 表示构造函数，`.cctor` 表示静态构造函数。

示例：

`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType`	匹配名为 `MyType` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有符号。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS.MyType.MyMethod(ParamType)`	匹配带有指定的完全限定签名的特定方法 `MyMethod`。
`dotnet_code_quality.CAXXXX.excluded_symbol_names = M:NS1.MyType1.MyMethod1(ParamType)|M:NS2.MyType2.MyMetho...e)`	匹配带有各自的完全限定签名的特定方法 `MyMethod1` 和 `MyMethod2`。

排除特定类型及其派生类型

可以从分析中排除特定类型及其派生类型。例如，若要指定规则不应针对名为 `MyType` 的类型及其派生类型中的任何代码运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType
```

选项值中允许的符号名称格式(用 `|` 分隔)：

- 仅类型名称(包括具有相应名称的所有类型，不考虑包含的类型或命名空间)。
- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `T:`(可选)。

示例：

`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType`	匹配名为 `MyType` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = MyType1|MyType2`	匹配名为 `MyType1` 或 `MyType2` 的所有类型及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS.MyType`	匹配带有给定的完全限定名称的特定类型 `MyType` 及其所有派生类型。
`dotnet_code_quality.CAXXXX.excluded_type_names_with_derived_types = M:NS1.MyType1|M:NS2.MyType2`	匹配带有各自的完全限定名称的特定类型 `MyType1` 和 `MyType2` 及其所有派生类型。

伪代码示例

```
using System;
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod(bool checkCertificateRevocationList)
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = checkCertificateRevocationList;
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            winHttpHandler.CheckCertificateRevocationList = true;
        }

        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

解决方案

```
using System.Net.Http;

class ExampleClass
{
    void ExampleMethod()
    {
        WinHttpHandler winHttpHandler = new WinHttpHandler();
        winHttpHandler.CheckCertificateRevocationList = true;
        HttpClient httpClient = new HttpClient(winHttpHandler);
    }
}
```

CA5401:不要将 CreateEncryptor 与非默认 IV 结合使用

2021/11/16 •

■ ID	CA5401
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将 System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor 与非默认 `rgbIV` 结合使用。

规则说明

对称加密应始终使用不可重复的初始化向量，以防止字典攻击。

此规则类似于 CA5402，但分析确定初始化向量一定为默认值。

如何解决冲突

使用默认 `rgbIV` 值，也就是使用不带任何参数的 System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor 的重载。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- `rgbIV` 参数由 System.Security.Cryptography.SymmetricAlgorithm.GenerateIV 生成。
- 确定 `rgbIV` 确实是随机且不可重复的参数。

伪代码示例

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] rgbIV)
    {
        AesCng aesCng  = new AesCng();
        aesCng.IV = rgbIV;
        aesCng.CreateEncryptor();
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AesCng aesCng  = new AesCng();
        aesCng.CreateEncryptor();
    }
}
```

CA5402:将 CreateEncryptor 与默认 IV 结合使用

2021/11/16 •

■ ID	CA5402
■	安全性
修复是中断修复还是非中断修复	非中断

原因

使用 System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor 时，`rgbIV` 可能为非默认值。

规则说明

对称加密应始终使用不可重复的初始化向量，以防止字典攻击。

此规则类似于 CA5401，但分析无法确定初始化向量是否一定为默认值。

如何解决冲突

显式使用默认 `rgbIV` 值，即，使用不带任何参数的 System.Security.Cryptography.SymmetricAlgorithm.CreateEncryptor 的重载。

何时禁止显示警告

在以下情况下，禁止显示此规则的警告是安全的：

- `rgbIV` 参数由 System.Security.Cryptography.SymmetricAlgorithm.GenerateIV 生成。
- 确定 `rgbIV` 参数确实是随机且不可重复的参数。
- 确定使用了初始化向量。

伪代码示例

```
using System;
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod(byte[] rgbIV)
    {
        AesCng aesCng  = new AesCng();
        Random r = new Random();

        if (r.Next(6) == 4)
        {
            aesCng.IV = rgbIV;
        }

        aesCng.CreateEncryptor();
    }
}
```

解决方案

```
using System.Security.Cryptography;

class ExampleClass
{
    public void ExampleMethod()
    {
        AesCng aesCng  = new AesCng();
        aesCng.CreateEncryptor();
    }
}
```

CA5403：请勿硬编码证书

2021/11/16 •

	[
■ ID	CA5403
■	安全性
修复是中断修复还是非中断修复	非中断

原因

X509Certificate 或 X509Certificate2 构造函数的 `data` 或 `rawData` 参数由以下内容之一进行硬编码：

- 字节数组。
- 字符数组。
- System.Convert.FromBase64String(String).
- System.Text.Encoding.GetBytes 的所有重载。

规则说明

硬编码证书的私钥很轻松就可以发现。即使使用已编译的二进制文件，恶意用户也很容易提取硬编码证书的私钥。私钥泄露后，攻击者便可以模拟该证书，受该证书保护的任何资源或操作都将提供给攻击者。

如何解决冲突

- 考虑重新设计应用程序，以使用安全的密钥管理系统，如 Azure Key Vault。
- 将凭据和证书保留在安全位置，与源代码分开。

何时禁止显示警告

如果硬编码数据不包含证书的私钥，可禁止显示此规则的警告。例如，数据来自 `.cer` 文件。硬编码的公共证书信息在证书过期或被撤销时仍可能为证书轮询带来难题。

伪代码示例

由字节数组硬编码

```
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = new byte[] {1, 2, 3};
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

由字符数组硬编码

```
using System.IO;
using System.Security.Cryptography.X509Certificates;
using System.Text;

class ExampleClass
{
    public void ExampleMethod(byte[] bytes, string path)
    {
        char[] chars = new char[] { '1', '2', '3' };
        Encoding.ASCII.GetBytes(chars, 0, 3, bytes, 0);
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

由 FromBase64String 硬编码

```
using System;
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = Convert.FromBase64String("AAAAAaazaoensuth");
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

由 GetBytes 硬编码

```
using System;
using System.IO;
using System.Security.Cryptography.X509Certificates;
using System.Text;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        byte[] bytes = Encoding.ASCII.GetBytes("AAAAAaazaoensuth");
        File.WriteAllBytes(path, bytes);
        new X509Certificate2(path);
    }
}
```

解决方案

```
using System.IO;
using System.Security.Cryptography.X509Certificates;

class ExampleClass
{
    public void ExampleMethod(string path)
    {
        new X509Certificate2("Certificate.cer");
    }
}
```

CA5404：不要禁用令牌验证检查

2021/11/16 •

	[
■ ID	CA5404
■	安全性
修复是中断修复还是非中断修复	非中断

原因

将 TokenValidationParameters 属性 `RequireExpirationTime`、`ValidateAudience`、`ValidateIssuer` 或 `ValidateLifetime` 设置为 `false`。

规则说明

令牌验证检查确保在验证令牌时分析并验证所有方面。禁用验证会允许不受信任的令牌通过验证，从而可能导致安全漏洞。

有关令牌验证最佳做法的更多详细信息，可参阅库的 Wiki。

如何解决冲突

将 TokenValidationParameters 属性 `RequireExpirationTime`、`ValidateAudience`、`ValidateIssuer` 或 `ValidateLifetime` 设置为 `true`。或者，删除对 `false` 的赋值，因为默认值为 `true`。

何时禁止显示警告

在大多数情况下，此验证对于确保使用方应用的安全性至关重要。但是，在某些情况下不需要此验证，尤其是在非标准令牌类型中。在禁用此验证之前，请务必充分考虑安全后果。有关利弊的信息，请参阅令牌验证库的 Wiki。

伪代码示例

```
using System;
using Microsoft.IdentityModel.Tokens;

class TestClass
{
    public void TestMethod()
    {
        TokenValidationParameters parameters = new TokenValidationParameters();
        parameters.RequireExpirationTime = false;
        parameters.ValidateAudience = false;
        parameters.ValidateIssuer = false;
        parameters.ValidateLifetime = false;
    }
}
```

解决方案

```
using System;
using Microsoft.IdentityModel.Tokens;

class TestClass
{
    public void TestMethod()
    {
        TokenValidationParameters parameters = new TokenValidationParameters();
        parameters.RequireExpirationTime = true;
        parameters.ValidateAudience = true;
        parameters.ValidateIssuer = true;
        parameters.ValidateLifetime = true;
    }
}
```

CA5405：不要始终跳过委托中的令牌验证

2021/11/16 •

	[
■ ID	CA5405
■	安全性
修复是中断修复还是非中断修复	非中断

原因

分配给 AudienceValidator 或 LifetimeValidator 的回调始终返回 `true`。

规则说明

通过设置关键的 `TokenValidationParameter` 验证委托以始终返回 `true`，会禁用重要的身份验证保护机制。禁用保护机制可能导致错误地验证来自任何颁发者的令牌或已过期的令牌。

有关令牌验证最佳做法的详细信息，请参阅库的 Wiki。

如何解决冲突

- 改进委托逻辑，以避免所有代码路径都返回 `true`（实际上会禁用该验证类型）。
- 当你想要使验证失败，让其他案例通过验证（返回 `true`）时，在失败案例中会引发 `SecurityTokenInvalidAudienceException` 或 `SecurityTokenInvalidLifetimeException`。

何时禁止显示警告

在某些特定的情况下，如果你利用委托进行额外的日志记录，而该委托用于不需要特定验证类型的令牌类型，则禁止显示此警告可能会有作用。在禁用此验证之前，请务必充分考虑安全后果。有关利弊的信息，请参阅令牌验证库的 Wiki。

伪代码示例

冲突

```
using System;
using Microsoft.IdentityModel.Tokens;

class TestClass
{
    public void TestMethod()
    {
        TokenValidationParameters parameters = new TokenValidationParameters();
        parameters.AudienceValidator = (audiences, token, tvp) => { return true; };
    }
}
```

解决方案

```
using System;
using Microsoft.IdentityModel.Tokens;

class TestClass
{
    public void TestMethod()
    {
        TokenValidationParameters parameters = new TokenValidationParameters();
        parameters.AudienceValidator = (audiences, token, tvp) =>
        {
            // Implement your own custom audience validation
            if (PerformCustomAudienceValidation(audiences, token))
                return true;
            else
                return false;
        };
    }
}
```

用法规则

2021/11/16 •

使用规则支持 .NET 的正确用法。

本节内容

"	"
CA1801:检查未使用的参数	方法签名包含一个没有在方法体中使用的参数。
CA1816:正确调用 GC.SuppressFinalize	表示 Dispose 实现的方法不调用 `GC.SuppressFinalize`；或表示不是 `Dispose` 实现的方法调用 `GC.SuppressFinalize`；或方法调用 `GC.SuppressFinalize` 并传递除 `this`（Visual Basic 中的 `Me`）以外的其他内容。
CA2200:再次引发以保留堆栈详细信息	再次引发某个异常，在 throw 语句中显式指定了该异常。如果通过在 throw 语句中指定异常来重新引发该异常，则引发该异常的原始方法与当前方法之间的方法调用的列表将丢失。
CA2201:不要引发保留的异常类型	这使得很难检测和调试原始错误。
CA2207:以内联方式初始化值类型的静态字段	某值类型声明了显式静态构造函数。要修复与该规则的冲突，请在声明它时初始化所有静态数据并移除静态构造函数。
CA2208:正确实例化参数异常	调用了异常类型 ArgumentException 或其派生类型的默认（无参数）构造函数，或者向异常类型 ArgumentException 或其派生类型的参数化构造函数传递了错误的字符串参数。
CA2211:非常量字段不应是可见的	不是常数也不是只读的静态字段不是线程安全的。必须谨慎控制对这类字段的访问，并需要使用高级编程技术来实现对类对象的同步访问。
CA2213:应释放可释放的字段	实现 System.IDisposable 的类型声明具有同样实现 `IDisposable` 的类型的字段。字段的 `Dispose` 方法不由声明类型的 `Dispose` 方法调用。
CA2214:不要在构造函数中调用可重写的方法	构造函数调用虚拟方法时，可能尚未执行调用该方法的实例的构造函数。
CA2215:Dispose 方法应调用基类释放	如果某个类型继承自一个可释放类型，则该类型必须从其自身的 `Dispose` 方法内调用基类型的 `Dispose` 方法。
CA2216:可释放类型应声明终结器	实现 System.IDisposable 并包含建议使用非托管资源的字段的类型未实现 `Object.Finalize` 所描述的终结器。
CA2217:不要使用 FlagsAttribute 标记枚举	外部可见的枚举通过 `FlagsAttribute` 进行标记，并且它包含的一个或多个值不是 2 的幂或枚举上定义的其他值的组合。

CA2218:重写 Equals 时重写 GetHashCode	公共类型重写 System.Object.Equals，但不重写 System.Object.GetHashCode。
CA2219:在异常子句中不引发异常	如果在 finally 或 fault 子句中引发异常，新异常将隐藏活动异常。当在 filter 子句中引发异常时，运行时会在不提示的情况下捕捉异常。这使得很难检测和调试原始错误。
CA2224:重载相等运算符时重写 Equals 方法	公共类型会实现相等运算符，但不重写 System.Object.Equals。
CA2225:运算符重载具有命名的备用项	检测到运算符重载，但未找到预期的指定备用方法。命名的备用成员提供对与运算符相同的功能的访问，并且可供以不支持重载运算符的语言进行编程的开发人员使用。
CA2226:运算符应有对称重载	类型实现相等运算符或不相等运算符，但不实现相反运算符。
CA2227:集合属性应为只读	使用可写的集合属性，用户可以将该集合替换为不同的集合。只读属性禁止替换该集合，但仍允许设置单个成员。
CA2229:实现序列化构造函数	要修复与该规则的冲突，请实现序列化构造函数。对于密封类，请使构造函数成为私有；否则，请使构造函数成为受保护。
CA2231:重写 ValueType.Equals 时应重载相等运算符	值类型重写 `Object.Equals`，但不实现相等运算符。
CA2234:传递 System.Uri 对象，而不传递字符串	调用了带有一个字符串参数的方法，该参数的名称中包含“uri”、“URI”、“urn”、“URN”、“url”或“URL”。此方法的声明类型包含具有 System.Uri 参数的对应方法重载。
CA2235:标记所有不可序列化的字段	在可以序列化的类型中声明了类型不可序列化的实例字段。
CA2237:用 SerializableAttribute 标记 ISerializable 类型	若要被公共语言运行时识别为可序列化，类型必须用 SerializableAttribute 特性标记，即使该类型通过实现 `ISerializable` 接口使用了自定义的序列化例程也是如此。
CA2241:为格式化方法提供正确的参数	传递到 String.Format 的 format 实参不包含对应于每个对象实参的格式项，反之亦然。
CA2242:正确测试 NaN	此表达式针对 `Single.Nan` 或 `Double.Nan` 测试值。使用 `Single.IsNan(Single)` 或 `Double.IsNan(Double)` 测试值。
CA2243:特性字符串文本应正确分析	特性的字符串文本形参不能正确解析为 URL、GUID 或版本。
CA2244:不要复制已索引的元素初始值设定项	对象初始值设定项有多个具有相同常量索引的索引元素初始值设定项。除最后一个初始值设定项之外，其余都是冗余的。
CA2245:请勿将属性分配给其自身	属性意外赋值给了其自身。
CA2246:请勿在同一语句中分配符号及其成员	不建议在同一语句中分配符号及其成员（即字段或属性）。目前尚不清楚成员访问是打算在赋值之前使用符号的旧值还是打算使用此语句中赋值的新值。

"	"
CA2247:传递给 TaskCompletionSource 构造函数的参数应为 TaskCreationOptions 枚举，而不是 TaskContinuationOptions 枚举	TaskCompletionSource 既有采用控制基础任务的 TaskCreationOptions 的构造函数，也有采用任务中存储的对象状态的构造函数。如果意外传递 TaskContinuationOptions 而不是 TaskCreationOptions，则将导致调用将选项视为状态。
CA2248：向 Enum.HasFlag 提供正确的 enum 实参	作为实参传递给 `HasFlag` 方法调用的枚举类型不同于调用枚举类型。
CA2249：请考虑使用 String.Contains 而不是 String.IndexOf	对 `string.IndexOf` 的调用(其结果用于检查是否存在子字符串)可以用 `string.Contains` 替换。
CA2250：使用 `ThrowIfCancellationRequested`	`ThrowIfCancellationRequested` 自动检查令牌是否已取消，如果已取消，则引发 `OperationCanceledException` 。
CA2251：使用 `String.Equals` 代替 `String.Compare`	与其将 `String.Compare` 的结果与零进行比较，不如使用 `String.Equals` ，这样更清晰且速度可能更快。
CA2252：选择预览功能	使用预览 API 之前选择预览功能。

CA1801:检查未使用的参数

2021/11/16 •

	[
■ ID	CA1801
■	使用情况
▬▬▬	非中断性 - 无论进行了何种更改，如果成员在程序集外部不可见，则为非中断性修复。 非中断性 - 如果将成员更改为在其主体中使用参数，则为非中断性修复。 中断性 - 如果删除参数，并且该参数在程序集外可见，则为中断性修复。

原因

方法签名包含一个没有在方法主体中使用的参数。

此规则不检查以下几种方法：

- 委托引用的方法。
- 用作事件处理程序的方法。
- 序列化构造函数（请参阅指南）。
- 序列化 GetObjectData 方法。
- 使用 `abstract`（在 Visual Basic 中为 `MustOverride`）修饰符声明的方法。
- 使用 `virtual`（在 Visual Basic 中为 `Overridable`）修饰符声明的方法。
- 使用 `override`（在 Visual Basic 中为 `Overrides`）修饰符声明的方法。
- 使用 `extern`（在 Visual Basic 中为 `Declare` 语句）修饰符声明的方法。

此规则不会标记以丢弃符号命名的参数，例如 `_``_1` 和 `_2`。这可以降低签名要求所需参数的警告干扰，例如，用作委托的方法、具有特殊属性的参数或其值在运行时由框架隐式访问但未在代码中引用的参数。

> **NOTE**
> 此规则已弃用，取而代之的是 IDE0060。有关如何在生成时强制执行 IDE0060 分析器的信息，请参阅代码样式分析。

规则说明

查看未在方法体中使用的非虚拟方法中的参数，确保在访问失败时不存在错误。未使用的参数会产生维护和性能费用。

有时，与此规则的冲突可能会导致方法中的实现 bug。例如，应在方法主体中使用该参数的情况。如果参数因为后向兼容性而必须存在，请禁止显示此规则发出的警告。

如何解决冲突

若要修复与此规则的冲突，请删除未使用的参数(中断性变更)，或在方法主体中使用该参数(非中断性变更)。

何时禁止显示警告

在以下情况下，禁止显示此规则发出的警告是安全的：

- 在以前发布的代码中，修复是一项中断性变更。
- 对于 Microsoft.VisualStudio.TestTools.UnitTesting.Assert 的自定义扩展方法中的 `this` 参数。Assert 类中的函数是静态的，因此无需访问方法主体中的 `this` 参数。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别(性能)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

默认情况下，CA1801 规则适用于所有 API 图面(公共、内部和私有)。

示例

下面的示例展示了两个方法。其中一个方法违反了规则，而另一个方法符合规则。

```
// This method violates the rule.
public static string GetSomething(int first, int second)
{
    return first.ToString(CultureInfo.InvariantCulture);
}

// This method satisfies the rule.
public static string GetSomethingElse(int first)
{
    return first.ToString(CultureInfo.InvariantCulture);
}
```

相关规则

- CA1812:避免未实例化的内部类
- CA2229:实现序列化构造函数

CA1816:正确调用 GC.SuppressFinalize

2021/11/16 •

	▮
■ ID	CA1816
■	使用情况
▬▬▬	非中断

原因

此规则的冲突可能由以下原因引起：

- 一种方法，是 IDisposable.Dispose 的实现，但不调用 GC.SuppressFinalize。
- 一种方法，不是 IDisposable.Dispose 的实现，但调用 GC.SuppressFinalize。
- 一种方法，调用 GC.SuppressFinalize 并传递 this (C#) 或 Me (Visual Basic) 以外的内容。

规则说明

使用 IDisposable.Dispose 方法，用户可在对象用于垃圾回收之前随时释放资源。如果调用 IDisposable.Dispose 方法，则会释放对象的资源。这使终止变得没有必要。IDisposable.Dispose 应该调用 GC.SuppressFinalize，这样垃圾回收器就不会调用对象的终结器。

若要阻止带有终结器的派生类型重新实现 IDisposable 并对其进行调用，则没有终结器的未密封类型仍然应该调用 GC.SuppressFinalize。

如何解决冲突

若有解决此规则的冲突：

- 如果方法是 Dispose 的实现，则添加对 GC.SuppressFinalize 的调用。
- 如果方法不是 Dispose 的实现，请删除对 GC.SuppressFinalize 的调用或将其移动到类型的 Dispose 实现。
- 将所有调用更改为 GC.SuppressFinalize，以传递 this (C#) 或 Me (Visual Basic)。

何时禁止显示警告

只有在故意使用 GC.SuppressFinalize 来控制其他对象的生存期时，才禁止显示此规则发出的警告。如果 Dispose 的实现没有调用 GC.SuppressFinalize，则不要禁止显示此规则发出的警告。在此情况下，未能禁止显示终结会降低性能，且没有任何好处。

与 CA1816 冲突的示例

这代码演示了一种方法，该方法调用 GC.SuppressFinalize，但不传递 this (C#) 或 Me (Visual Basic)。结果，此代码与规则 CA1816 冲突。

```
Public Class DatabaseConnector
    Implements IDisposable

    Private _Connection As New SqlConnection

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        ' Violates rules
        GC.SuppressFinalize(True)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If _Connection IsNot Nothing Then
                _Connection.Dispose()
                _Connection = Nothing
            End If
        End If
    End Sub

End Class
```

```
public class DatabaseConnector : IDisposable
{
    private SqlConnection _Connection = new SqlConnection();

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(true);  // Violates rule
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_Connection != null)
            {
                _Connection.Dispose();
                _Connection = null;
            }
        }
    }
}
```

满足 CA1816 的示例

本示例演示了一种方法，该方法通过传递 this (C#) 或 Me (Visual Basic) 来正确调用 GC.SuppressFinalize。

```
Public Class DatabaseConnector
    Implements IDisposable

    Private _Connection As New SqlConnection

    Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If _Connection IsNot Nothing Then
                _Connection.Dispose()
                _Connection = Nothing
            End If
        End If
    End Sub

End Class
```

```
public class DatabaseConnector : IDisposable
{
    private SqlConnection _Connection = new SqlConnection();

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            if (_Connection != null)
            {
                _Connection.Dispose();
                _Connection = null;
            }
        }
    }
}
```

相关规则

- CA2215:Dispose 方法应调用基类释放
- CA2216:可释放类型应声明终结器

另请参阅

- 释放模式

CA2200:再次引发以保留堆栈详细信息

2021/11/16 •

	[
■ ID	CA2200
■	使用情况
▬▬▬	非中断

原因

再次引发某个异常，在 `throw` 语句中显式指定该异常。

规则说明

引发异常后，它所包含的部分信息为堆栈跟踪。堆栈跟踪是一个方法调用层次结构列表，它以引发异常的方法开头，以捕获异常的方法结尾。如果通过在 `throw` 语句中指定异常来再次引发该异常，则将在当前方法处重启堆栈跟踪，并且引发该异常的原始方法与当前方法之间的方法调用列表将丢失。若要保留原始堆栈跟踪信息和异常信息，请在未指定异常的情况下使用 `throw` 语句。

如何解决冲突

若要解决此规则的冲突，请在不显式指定异常的情况下再次引发异常。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例演示了违反规则的 `CatchAndRethrowExplicitly` 方法和符合规则的 `CatchAndRethrowImplicitly` 方法。

```
class TestsRethrow
{
    static void Main2200()
    {
        TestsRethrow testRethrow = new TestsRethrow();
        testRethrow.CatchException();
    }

    void CatchException()
    {
        try
        {
            CatchAndRethrowExplicitly();
        }
        catch (ArithmeticException e)
        {
            Console.WriteLine("Explicitly specified:{0}{1}",
               Environment.NewLine, e.StackTrace);
        }

        try
        {
            CatchAndRethrowImplicitly();
        }
        catch (ArithmeticException e)
        {
            Console.WriteLine("{0}Implicitly specified:{0}{1}",
               Environment.NewLine, e.StackTrace);
        }
    }

    void CatchAndRethrowExplicitly()
    {
        try
        {
            ThrowException();
        }
        catch (ArithmeticException e)
        {
            // Violates the rule.
            throw e;
        }
    }

    void CatchAndRethrowImplicitly()
    {
        try
        {
            ThrowException();
        }
        catch (ArithmeticException)
        {
            // Satisfies the rule.
            throw;
        }
    }

    void ThrowException()
    {
        throw new ArithmeticException("illegal expression");
    }
}
```

```
Imports System

Namespace ca2200

    Class TestsRethrow

        Shared Sub Main2200()
            Dim testRethrow As New TestsRethrow()
            testRethrow.CatchException()
        End Sub

        Sub CatchException()

            Try
                CatchAndRethrowExplicitly()
            Catch e As ArithmeticException
                Console.WriteLine("Explicitly specified:{0}{1}",
               Environment.NewLine, e.StackTrace)
            End Try

            Try
                CatchAndRethrowImplicitly()
            Catch e As ArithmeticException
                Console.WriteLine("{0}Implicitly specified:{0}{1}",
               Environment.NewLine, e.StackTrace)
            End Try

        End Sub

        Sub CatchAndRethrowExplicitly()

            Try
                ThrowException()
            Catch e As ArithmeticException

                ' Violates the rule.
                Throw e
            End Try

        End Sub

        Sub CatchAndRethrowImplicitly()

            Try
                ThrowException()
            Catch e As ArithmeticException

                ' Satisfies the rule.
                Throw
            End Try

        End Sub

        Sub ThrowException()
            Throw New ArithmeticException("illegal expression")
        End Sub

    End Class

End Namespace
```

CA2201:不要引发保留的异常类型

2021/11/16 •

■ ID	CA2201
■	使用情况
■■■■■	重大

原因

方法引发的异常类型太过笼统，或已由运行时保留。

规则说明

以下异常类型太过笼统，无法为用户提供足够的信息：

- System.Exception
- System.ApplicationException
- System.SystemException

以下异常类型已保留，仅应由公共语言运行时引发：

- System.AccessViolationException
- System.ExecutionEngineException
- System.IndexOutOfRangeException
- System.NullReferenceException
- System.OutOfMemoryException
- System.Runtime.InteropServices.COMException
- System.Runtime.InteropServices.ExternalException
- System.Runtime.InteropServices.SEHException
- System.StackOverflowException

请勿引发一般异常

如果在库或框架中引发一般异常类型，如 Exception 或 SystemException，将强制使用者捕捉所有异常，包括不知该如何处理的未知异常。

相反，要么引发框架中已存在的派生程度更高的类型，要么创建自己从 Exception 派生的类型。

引发特定异常

下表显示参数以及验证参数时引发的异常，包括属性的 set 访问器中的值参数：

情况	异常
`null` 参考	System.ArgumentNullException
超出允许的值（如集合或列表的索引）范围	System.ArgumentOutOfRangeException
`enum` 值无效	System.ComponentModel.InvalidEnumArgumentException
包含不满足方法参数规范的格式（如适用于 `ToString(String)` 格式字符串）	System.FormatException
否则无效	System.ArgumentException

当操作对对象的当前状态无效时，引发 System.InvalidOperationException

当对已释放的对象执行操作时，引发 System.ObjectDisposedException

当不支持某个操作（例如，在因读取而打开的 Stream 中重写 **Stream.Write**）时，引发 System.NotSupportedException

当转换将导致溢出（例如，显式强制转换运算符重载）时，引发 System.OverflowException

对于所有其他情况，请考虑创建自己的从 Exception 派生的类型，并将其引发。

如何解决冲突

要解决此规则的冲突，请将引发的异常类型更改为特定类型，此特定类型不属于保留的类型。

何时禁止显示警告

不禁止显示此规则发出的警告。

相关规则

- CA1031:不要捕捉一般异常类型

CA2207:以内联方式初始化值类型的静态字段

2021/11/16 •

■ ID	CA2207
■	使用情况
■■■■■	非中断

原因

某值类型声明了显式静态构造函数。

规则说明

声明值类型时，它将接受默认的初始化，其中所有值类型字段均设置为零，而所有引用类型字段均设置为 `null`（在 Visual Basic 中为 `Nothing`）。只有在调用该类型的实例构造函数或静态成员之前，才能保证运行显式静态构造函数。因此，如果创建该类型时未调用实例构造函数，则无法保证运行静态构造函数。

如果所有静态数据都以内联方式初始化，并且未声明任何显式静态构造函数，则 C# 和 Visual Basic 编译器会将 `beforefieldinit` 标志添加到 MSIL 类定义。编译器还会添加包含静态初始化代码的专用静态构造函数。可确保在访问该类型的任何静态字段之前运行此专用静态构造函数。

如何解决冲突

若要解决与此规则的冲突，请在声明所有静态数据时对其进行初始化，并删除静态构造函数。

何时禁止显示警告

不禁止显示此规则发出的警告。

相关规则

CA1810:以内联方式初始化引用类型的静态字段

CA2208:正确实例化参数异常

2021/11/16 •

	▮
■ ID	CA2208
■	使用情况
▬▬	非中断

原因

如果方法具有参数，并且引发了异常类型 ArgumentException 或其派生类型，则它应正确调用接受 `paramName` 参数的构造函数。可能的原因包括以下情况：

- 调用了异常类型 ArgumentException 或其派生类型的默认(无参数)构造函数，该异常类型具有接受 `paramName` 参数的构造函数。
- 向异常类型 ArgumentException 或其派生类型的参数化构造函数传递了错误的字符串参数。
- 为 ArgumentException 异常类型或其派生类型的构造函数的 `message` 参数传递了其中一个参数名称。

规则说明

请勿调用默认构造函数，而是调用允许提供更有意义的异常消息的构造函数重载之一。异常消息应面向开发人员，并清楚地说明错误情况以及如何纠正或避免异常。

ArgumentException 及其派生类型的一个和两个字符串构造函数的签名与 `message` 和 `paramName` 参数位置不一致。确保使用正确的字符串参数调用这些构造函数。签名如下：

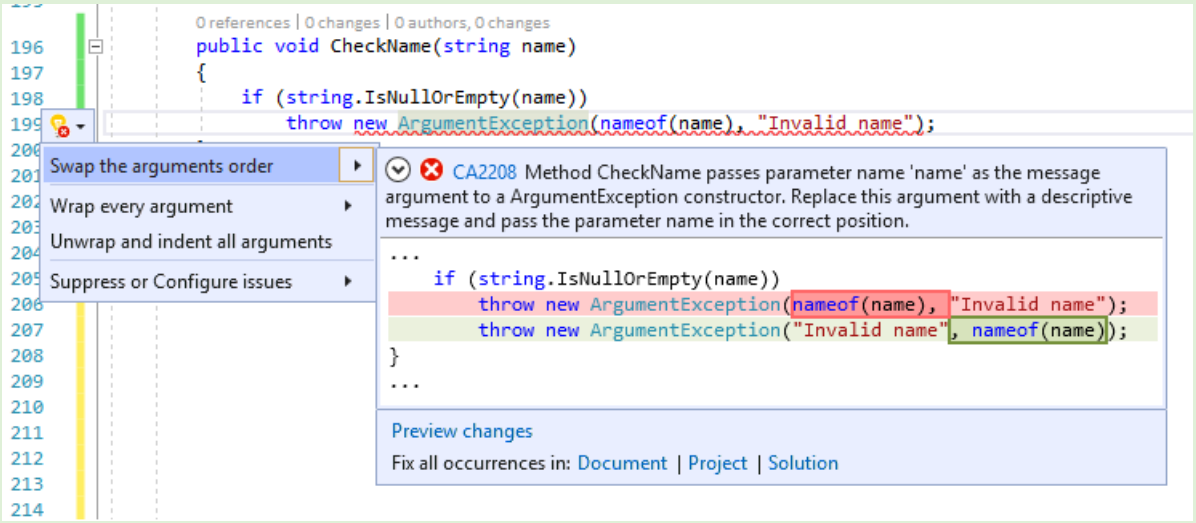
- `ArgumentException(string message)`
- `ArgumentException(string message, string paramName)`
- `ArgumentNullException(string paramName)`
- `ArgumentNullException(string paramName, string message)`
- `ArgumentOutOfRangeException(string paramName)`
- `ArgumentOutOfRangeException(string paramName, string message)`
- `DuplicateWaitObjectException(string parameterName)`
- `DuplicateWaitObjectException(string parameterName, string message)`

如何解决冲突

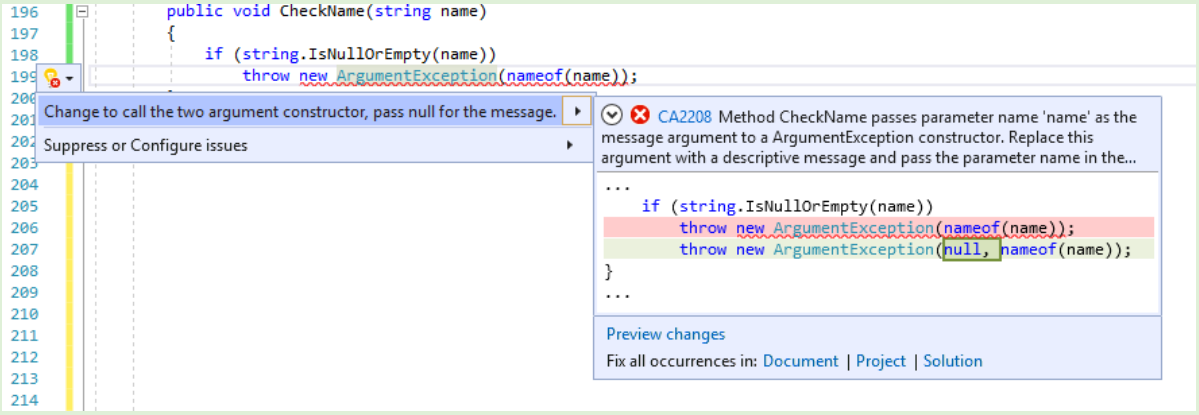
若要解决此规则的冲突，请调用使用消息和/或参数名称的构造函数，并确保参数对于要调用的 ArgumentException 类型是正确的。

TIP

Visual Studio 中提供了针对参数名称位置错误的代码修补程序。若要使用它，请将光标置于警告行上，然后按 Ctrl+.（句点）。从显示的选项列表中选择“交换参数顺序”。



如果将参数名而不是消息传递给 ArgumentException(String) 方法，修补程序将改为提供切换到双参数构造函数的选项。



何时禁止显示警告

仅当使用正确的字符串参数调用参数化构造函数时，才可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

你可以仅为此规则、为所有规则或为此类别中的所有规则配置此选项（设计）。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

默认情况下，CA2208 规则适用于所有 API 图面（公共、内部和私有）。

示例

下面的代码演示了一个错误地实例化 ArgumentNullException 实例的构造函数。

```
public class Book
{
    public Book(string title)
    {
        Title = title ??
            throw new ArgumentNullException("All books must have a title.", nameof(title));
    }

    public string Title { get; }
}
```

```
Public Class Book

    Private ReadOnly _Title As String

    Public Sub New(ByVal title As String)
        ' Violates this rule (constructor arguments are switched)
        If (title Is Nothing) Then
            Throw New ArgumentNullException("title cannot be a null reference (Nothing in Visual Basic)",
"title")
        End If
        _Title = title
    End Sub

    Public ReadOnly Property Title()
        Get
            Return _Title
        End Get
    End Property

End Class
```

下面的代码通过切换构造函数参数来解决前面的冲突。

```
public class Book
{
    public Book(string title)
    {
        Title = title ??
            throw new ArgumentNullException(nameof(title), "All books must have a title.");
    }

    public string Title { get; }
}
```

```
Public Class Book

    Private ReadOnly _Title As String

    Public Sub New(ByVal title As String)
        If (title Is Nothing) Then
            Throw New ArgumentNullException("title", "title cannot be a null reference (Nothing in Visual
Basic)")
        End If

        _Title = title
    End Sub

    Public ReadOnly Property Title()
        Get
            Return _Title
        End Get
    End Property

End Class
```

相关规则

- CA1507：使用 nameof 代替字符串

CA2211:非常量字段不应是可见的

2021/11/16 •

■ ID	CA2211
■	使用情况
▬▬	重大

原因

公共或受保护的静态字段既不是常量，也不是只读字段。

规则说明

不是常数也不是只读字段的静态字段不是线程安全的。必须谨慎控制对这类字段的访问，并需要使用高级编程技术来实现对类对象的同步访问。由于这些都是很难学习和掌握的技能，并且测试这种对象有其自身的难度，因此静态字段最好用于存储不会更改的数据。此规则适用于库；应用程序不应公开任何字段。

如何解决冲突

若要解决与此规则的冲突，请将静态字段设置为常量或只读。如果无法实现，请重新设计类型，使用替代机制（如线程安全属性）来管理对基础字段的线程安全访问。请注意，锁争用和死锁之类的问题可能会影响库的性能和行为。

何时禁止显示警告

如果你正在开发应用程序，则可以安全地禁止显示此规则的警告，从而完全控制对包含静态字段的类型的访问。库设计器不应禁止显示此规则的警告；使用非常量静态字段会让开发人员难以正确使用库。

示例

下面的示例演示了与此规则发生冲突的类型。

```
Imports System

Namespace ca2211

    Public Class SomeStaticFields
        ' Violates rule: AvoidNonConstantStatic;
        ' the field is public and not a literal.
        Public Shared publicField As DateTime = DateTime.Now

        ' Satisfies rule: AvoidNonConstantStatic.
        Public Shared ReadOnly literalField As DateTime = DateTime.Now

        ' Satisfies rule: NonConstantFieldsShouldNotBeVisible;
        ' the field is private.
        Private Shared privateField As DateTime = DateTime.Now
    End Class
End Namespace
```

```
public class SomeStaticFields
{
    // Violates rule: AvoidNonConstantStatic;
    // the field is public and not a literal.
    static public DateTime publicField = DateTime.Now;

    // Satisfies rule: AvoidNonConstantStatic.
    public static readonly DateTime literalField = DateTime.Now;

    // Satisfies rule: NonConstantFieldsShouldNotBeVisible;
    // the field is private.
    static DateTime privateField = DateTime.Now;
}
```

CA2213:应释放可释放的字段

2021/11/16 •

	[
■ ID	CA2213
■	使用情况
▬▬▬	非中断

原因

实现 System.IDisposable 的类型声明具有同样实现 IDisposable 的类型的字段。字段的 Dispose 方法不由声明类型的 Dispose 方法调用。

规则说明

类型负责释放其所有非托管资源。规则 CA2213 检查可释放类型(即实现 IDisposable 的类型) `T` 是否声明字段 `F` (可释放类型 `FT` 的实例)。对于分配了具有包含类型 `T` 的方法或初始化表达式中的本地创建对象的每个字段 `F`,该规则都会尝试查找对 `FT.Dispose` 的调用。该规则搜索 `T.Dispose` 调用的方法和下一级方法(即,由 `T.Dispose` 调用的方法调用的方法)。

> **NOTE**
> 除特殊情况外,规则 CA2213 仅针对分配了包含类型的方法和初始化表达式中的本地创建可释放对象的字段触发。如果对象在类型 `T` 之外创建或分配,则该规则不会触发。对于包含类型不负责释放对象的情况,这可以减少干扰。

特殊情况

即使分配的对象未在本地创建,规则 CA2213 也会针对以下类型的字段触发:

- System.IO.Stream
- System.IO.TextReader
- System.IO.TextWriter
- System.Resources.IResourceReader

将其中一种类型的对象传递给构造函数,然后将其分配到某个字段,可指示释放所有权转移到新构造的类型。也就是说,新构造的类型现在负责释放对象。如果未释放对象,则会发生 CA2213 冲突。

如何解决冲突

若要解决此规则的冲突,请在具有实现 IDisposable 的类型的字段上调用 Dispose。

何时禁止显示警告

在以下情况下,禁止显示此规则的警告是安全的:

- 标记的类型不负责释放字段占用的资源(即,该类型不具有释放所有权)

- 对 Dispose 的调用发生在比规则检查的调用级别更深的级别
- 包含类型不拥有字段的释放所有权。

示例

以下代码片段显示实现 IDisposable 的类型 `TypeA`。

```
public class TypeA : IDisposable
{
    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            // Dispose managed resources
        }

        // Free native resources
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    // Disposable types implement a finalizer.
    ~TypeA()
    {
        Dispose(false);
    }
}
```

以下代码片段显示类型 `TypeB`，它因为将字段 `aFieldOfADisposableType` 声明为可释放类型 (`TypeA`) 且没有在该字段上调用 Dispose 而与规则 CA2213 冲突。

```
public class TypeB : IDisposable
{
    // Assume this type has some unmanaged resources.
    TypeA aFieldOfADisposableType = new TypeA();
    private bool disposed = false;

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            // Dispose of resources held by this instance.

            // Violates rule: DisposableFieldsShouldBeDisposed.
            // Should call aFieldOfADisposableType.Dispose();

            disposed = true;
            // Suppress finalization of this disposed instance.
            if (disposing)
            {
                GC.SuppressFinalize(this);
            }
        }
    }

    public void Dispose()
    {
        if (!disposed)
        {
            // Dispose of resources held by this instance.
            Dispose(true);
        }
    }

    // Disposable types implement a finalizer.
    ~TypeB()
    {
        Dispose(false);
    }
}
```

若要解决冲突，请在可释放字段上调用 `Dispose()`：

```
protected virtual void Dispose(bool disposing)
{
   if (!disposed)
   {
      // Dispose of resources held by this instance.
      aFieldOfADisposableType.Dispose();

      disposed = true;

      // Suppress finalization of this disposed instance.
      if (disposing)
      {
          GC.SuppressFinalize(this);
      }
   }
}
```

另请参阅

- System.IDisposable
- 释放模式

CA2214:不要在构造函数中调用可重写的方法

2021/11/16 •

	[
■ ID	CA2214
■	使用情况
▬▬▬	非中断

原因

非密封类型的构造函数调用其类中定义的虚拟方法。

规则说明

当调用虚拟方法时，直到运行时才会选择执行该方法的实际类型。构造函数调用虚拟方法时，可能尚未执行调用该方法的实例的构造函数。如果已重写的虚拟方法依赖于构造函数中的初始化和其他配置，则可能导致错误或意外行为。

如何解决冲突

若要解决此规则的冲突，请不要从某个类型的构造函数中调用该类型的虚拟方法。

何时禁止显示警告

不禁止显示此规则发出的警告。应重新设计构造函数以消除对虚拟方法的调用。

示例

下面的示例演示了与此规则发生冲突的后果。测试应用程序会创建一个 `DerivedType` 实例，从而执行其基类 (`BadlyConstructedType`) 构造函数。 `BadlyConstructedType` 的构造函数错误调用了虚拟方法 `DoSomething` 。如输出所示，先执行 `DerivedType` 的构造函数，再执行 `DerivedType.DoSomething()` 。

```
public class BadlyConstructedType
{
    protected string initialized = "No";

    public BadlyConstructedType()
    {
        Console.WriteLine("Calling base ctor.");
        // Violates rule: DoNotCallOverridableMethodsInConstructors.
        DoSomething();
    }
    // This will be overridden in the derived type.
    public virtual void DoSomething()
    {
        Console.WriteLine("Base DoSomething");
    }
}

public class DerivedType : BadlyConstructedType
{
    public DerivedType()
    {
        Console.WriteLine("Calling derived ctor.");
        initialized = "Yes";
    }
    public override void DoSomething()
    {
        Console.WriteLine("Derived DoSomething is called - initialized ? {0}", initialized);
    }
}

public class TestBadlyConstructedType
{
    public static void Main2214()
    {
        DerivedType derivedInstance = new DerivedType();
    }
}
```

```
Imports System

Namespace ca2214

    Public Class BadlyConstructedType
        Protected initialized As String = "No"


        Public Sub New()
            Console.WriteLine("Calling base ctor.")
            ' Violates rule: DoNotCallOverridableMethodsInConstructors.
            DoSomething()
        End Sub 'New

        ' This will be overridden in the derived type.
        Public Overridable Sub DoSomething()
            Console.WriteLine("Base DoSomething")
        End Sub 'DoSomething
    End Class 'BadlyConstructedType


    Public Class DerivedType
        Inherits BadlyConstructedType

        Public Sub New()
            Console.WriteLine("Calling derived ctor.")
            initialized = "Yes"
        End Sub 'New

        Public Overrides Sub DoSomething()
            Console.WriteLine("Derived DoSomething is called - initialized ? {0}", initialized)
        End Sub 'DoSomething
    End Class 'DerivedType


    Public Class TestBadlyConstructedType

        Public Shared Sub Main2214()
            Dim derivedInstance As New DerivedType()
        End Sub 'Main
    End Class
End Namespace
```

该**示例**产**生下面的**输**出**:

```
Calling base ctor.
Derived DoSomething is called - initialized ? No
Calling derived ctor.
```

CA2215:Dispose 方法应调用基类释放

2021/11/16 •

■ ID	CA2215
■	使用情况
▬▬▬	非中断

原因

实现 System.IDisposable 的类型继承自同时实现 IDisposable 的类型。继承类型的 Dispose 方法不调用父类型的 Dispose 方法。

规则说明

如果某个类型继承自一个可释放类型，则该类型必须从其自身的 Dispose 方法内调用基类型的 Dispose 方法。 调用基类型 `Dispose` 方法可确保释放由基类型创建的所有资源。

如何解决冲突

若要解决此规则的冲突，请在 `base` 方法中调用 `base`.Dispose 。

何时禁止显示警告

如果对 `base`.Dispose 的调用发生在比规则检查更深的调用级别上，则可以安全地禁止此规则发出的警告。

示例

下面的示例展示了两个类型，一个是实现 IDisposable 的 `TypeA`，还有一个是继承自类型 `TypeA` 并正确调用其 Dispose 方法的 `TypeB` 。

```
Namespace ca2215

    Public Class TypeA
        Implements IDisposable

        Protected Overridable Overloads Sub Dispose(disposing As Boolean)
            If disposing Then
                ' dispose managed resources
            End If

            ' free native resources
        End Sub

        Public Overloads Sub Dispose() Implements IDisposable.Dispose
            Dispose(True)
            GC.SuppressFinalize(Me)
        End Sub

        ' Disposable types implement a finalizer.
        Protected Overrides Sub Finalize()
            Dispose(False)
            MyBase.Finalize()
        End Sub

    End Class

    Public Class TypeB
        Inherits TypeA

        Protected Overrides Sub Dispose(disposing As Boolean)
            If Not disposing Then
                MyBase.Dispose(False)
            End If
        End Sub

    End Class

End Namespace
```

```
using System;

namespace ca2215
{
    public class TypeA : IDisposable
    {
        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                // Dispose managed resources
            }

            // Free native resources
        }

        public void Dispose()
        {
            Dispose(true);
            GC.SuppressFinalize(this);
        }

        // Disposable types implement a finalizer.
        ~TypeA()
        {
            Dispose(false);
        }
    }

    public class TypeB : TypeA
    {
        protected override void Dispose(bool disposing)
        {
            if (!disposing)
            {
                base.Dispose(false);
            }
        }
    }
}
```

另请参阅

- System.IDisposable
- 释放模式

CA2216:可释放类型应声明终结器

2021/11/16 •

■ ID	CA2216
■	使用情况
▬▬▬	非中断

原因

实现 System.IDisposable 且包含指示使用非托管资源的字段的类型不实现 System.Object.Finalize 所描述的终结器。

规则说明

如果可释放类型包含以下类型的字段，则会发生与此规则的冲突：

- System.IntPtr
- System.UIntPtr
- System.Runtime.InteropServices.HandleRef

如何解决冲突

若要解决与此规则的冲突，请实现调用 Dispose 方法的终结器。

何时禁止显示警告

如果该类型不实现 IDisposable 以释放非托管资源，则可以安全地禁止显示此规则的警告。

示例

下面的示例演示了与此规则冲突的类型。

```
public class DisposeMissingFinalize : IDisposable
{
    private bool disposed = false;
    private IntPtr unmanagedResource;

    [DllImport("native.dll")]
    private static extern IntPtr AllocateUnmanagedResource();

    [DllImport("native.dll")]
    private static extern void FreeUnmanagedResource(IntPtr p);

    DisposeMissingFinalize()
    {
        unmanagedResource = AllocateUnmanagedResource();
    }

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            // Dispose of resources held by this instance.
            FreeUnmanagedResource(unmanagedResource);
            disposed = true;

            // Suppress finalization of this disposed instance.
            if (disposing)
            {
                GC.SuppressFinalize(this);
            }
        }
    }

    public void Dispose()
    {
        Dispose(true);
    }

    // Disposable types with unmanaged resources implement a finalizer.
    // Uncomment the following code to satisfy rule:
    //  DisposableTypesShouldDeclareFinalizer
    // ~TypeA()
    // {
    //     Dispose(false);
    // }
}
```

相关规则

CA1816:正确调用 GC.SuppressFinalize

另请参阅

- System.IDisposable
- System.IntPtr
- System.Runtime.InteropServices.HandleRef
- System.UIntPtr
- System.Object.Finalize
- 释放模式

CA2217:不要使用 FlagsAttribute 标记枚举

2021/11/16 •

■ ID	CA2217
■	使用情况
▬▬▬	非中断

原因

使用 FlagsAttribute 标记了枚举，并且它包含的一个或多个值不是 2 的幂或不是为该枚举定义的其他值的组合。

默认情况下，此规则仅查看外部可见的枚举，但这是可配置的。

规则说明

仅当枚举中定义的每个值都是 2 的幂或定义的值的组合时，枚举才应该具有 FlagsAttribute。

如何解决冲突

若要解决与此规则的冲突，请从枚举中删除 FlagsAttribute。

何时禁止显示警告

不禁止显示此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别(使用情况)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

以下代码演示了包含值 3 的枚举 `Color`。3 不是 2 的幂，也不是任何定义的值的组合。不应使用 FlagsAttribute 标记 `Color` 枚举。

```
// Violates this rule
[FlagsAttribute]
public enum Color
{
    None = 0,
    Red = 1,
    Orange = 3,
    Yellow = 4
}
```

```
Imports System

Namespace Samples

    ' Violates this rule
    <FlagsAttribute()> _
    Public Enum Color

        None = 0
        Red = 1
        Orange = 3
        Yellow = 4

    End Enum
End Namespace
```

以下代码演示了枚举 `Days`，该枚举满足使用 FlagsAttribute 进行标记的要求：

```
[FlagsAttribute]
public enum Days
{
    None = 0,
    Monday = 1,
    Tuesday = 2,
    Wednesday = 4,
    Thursday = 8,
    Friday = 16,
    All = Monday | Tuesday | Wednesday | Thursday | Friday
}
```

```
Imports System
Namespace Samples

    <FlagsAttribute()> _
    Public Enum Days

        None = 0
        Monday = 1
        Tuesday = 2
        Wednesday = 4
        Thursday = 8
        Friday = 16
        All = Monday Or Tuesday Or Wednesday Or Thursday Or Friday

    End Enum
End Namespace
```

相关规则

CA1027:用 FlagsAttribute 标记枚举

另请参阅

- System.FlagsAttribute

CA2218:重写 Equals 时重写 GetHashCode

2021/11/16 •

[	"["
RuleId	CA2218
类别	使用情况
重大更改	非中断

原因

公共类型重写 System.Object.Equals，但不重写 System.Object.GetHashCode。

规则说明

GetHashCode 基于当前实例返回一个适合哈希算法和哈希表之类的数据结构的值。两个相同类型且相等的对象必须返回相同的哈希代码，以确保以下类型的实例正常工作：

- System.Collections.Hashtable
- System.Collections.SortedList
- System.Collections.Generic.Dictionary<TKey,TValue>
- System.Collections.Generic.SortedDictionary<TKey,TValue>
- System.Collections.Generic.SortedList<TKey,TValue>
- System.Collections.Specialized.HybridDictionary
- System.Collections.Specialized.ListDictionary
- System.Collections.Specialized.OrderedDictionary
- 实现 System.Collections.Generic.IEqualityComparer<T> 的类型

> **NOTE**
> 此规则仅适用于 Visual Basic 代码。C# 编译器生成单独的警告 CS0659。

如何解决冲突

若要解决与此规则的冲突，请提供 GetHashCode 的实现。对于同一类型的两个对象，请确保实现返回相同的值（如果 Equals 的实现为这两个对象返回 `true` 值）。

何时禁止显示警告

不禁止显示此规则发出的警告。

类示例

下面的示例显示了一个与此规则发生冲突的类（引用类型）。

```
' This class violates the rule.
Public Class Point

    Public Property X As Integer
    Public Property Y As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.X = x
        Me.Y = y
    End Sub

    Public Overrides Function Equals(obj As Object) As Boolean

        If obj = Nothing Then
            Return False
        End If

        If [GetType]() <> obj.GetType() Then
            Return False
        End If

        Dim pt As Point = CType(obj, Point)

        Return X = pt.X AndAlso Y = pt.Y

    End Function

End Class
```

下面的示例通过重写 GetHashCode() 来解决冲突。

```
' This class satisfies the rule.
Public Class Point

    Public Property X As Integer
    Public Property Y As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.X = x
        Me.Y = y
    End Sub

    Public Overrides Function GetHashCode() As Integer
        Return X Or Y
    End Function

    Public Overrides Function Equals(obj As Object) As Boolean

        If obj = Nothing Then
            Return False
        End If

        If [GetType]() <> obj.GetType() Then
            Return False
        End If

        Dim pt As Point = CType(obj, Point)

        Return Equals(pt)

    End Function

    Public Overloads Function Equals(pt As Point) As Boolean
        Return X = pt.X AndAlso Y = pt.Y
    End Function

    Public Shared Operator =(pt1 As Point, pt2 As Point) As Boolean
        Return pt1.Equals(pt2)
    End Operator

    Public Shared Operator <>(pt1 As Point, pt2 As Point) As Boolean
        Return Not pt1.Equals(pt2)
    End Operator

End Class
```

相关规则

- CA1046:不要对引用类型重载相等运算符
- CA2224:重载相等运算符时重写 Equals 方法
- CA2225:运算符重载具有命名的备用项
- CA2226:运算符应有对称重载
- CA2231:重写 ValueType.Equals 时应重载相等运算符

另请参阅

- CS0659
- System.Object.Equals
- System.Object.GetHashCode
- System.Collections.Hashtable
- 相等运算符

CA2219:在异常子句中不引发异常

2021/11/16 •

	▮
■ ID	CA2219
■	使用情况
▬▬	非中断，中断

原因

`finally`、filter 或 fault 子句中引发了异常。

规则说明

当异常子句中引发异常时，会大大增加调试的难度。

如果在 `finally` 或 fault 子句中引发异常，新异常将隐藏活动异常（如果存在）。这使得很难检测和调试原始错误。

当 filter 子句中引发异常时，运行时会默默捕获异常，并导致 filter 评估为 false。无法区分评估为 false 的 filter 和从 filter 中引发的异常。这使得很难检测和调试 filter 逻辑中的错误。

如何解决冲突

若要解决此规则的冲突，请不要从 `finally`、filter 或 fault 子句中显式引发异常。

何时禁止显示警告

请勿禁止显示此规则的警告。无论何种情况，在异常子句中引发的异常都对执行代码无益。

相关规则

CA1065:不要在意外的位置引发异常

另请参阅

- 设计规则

CA2224：重载相等运算符时重写 Equals 方法

2021/11/16 •

[	"["
RuleId	CA2224
类别	使用情况
重大更改	非中断

原因

公共类型会实现相等运算符，但不重写 System.Object.Equals。

规则说明

相等运算符旨在以语法上方便的方式访问 Equals 方法的功能。如果实现相等运算符，其逻辑与 Equals 的逻辑必须相同。

> **NOTE**
> 此规则仅适用于 Visual Basic 代码。C# 编译器生成单独的警告 CS0660。

如何解决冲突

若要解决与此规则的冲突，应删除相等运算符的实现，或重写 Equals 并使两个方法返回相同的值。如果相等运算符未引入不一致的行为，则可以提供可调用基类中 Equals 方法的 Equals 的实现来解决冲突。

何时禁止显示警告

如果相等运算符返回的值与继承的 Equals 实现的值相同，则可以安全地禁止显示此规则发出的警告。本文中的示例包括一个可安全地禁止显示此规则发出的警告的类型。

示例

下面的示例显示了一个与此规则冲突的类（引用类型）。

```
' This class violates the rule.
Public Class Point

    Public Property X As Integer
    Public Property Y As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.X = x
        Me.Y = y
    End Sub

    Public Overrides Function GetHashCode() As Integer
        Return HashCode.Combine(X, Y)
    End Function

    Public Shared Operator =(pt1 As Point, pt2 As Point) As Boolean
        If pt1 Is Nothing OrElse pt2 Is Nothing Then
            Return False
        End If

        If pt1.GetType() <> pt2.GetType() Then
            Return False
        End If

        Return pt1.X = pt2.X AndAlso pt1.Y = pt2.Y
    End Operator

    Public Shared Operator <>(pt1 As Point, pt2 As Point) As Boolean
        Return Not pt1 = pt2
    End Operator

End Class
```

下面的示例通过重写 System.Object.Equals 来解决冲突。

```
' This class satisfies the rule.
Public Class Point

    Public Property X As Integer
    Public Property Y As Integer

    Public Sub New(x As Integer, y As Integer)
        Me.X = x
        Me.Y = y
    End Sub

    Public Overrides Function GetHashCode() As Integer
        Return HashCode.Combine(X, Y)
    End Function

    Public Overrides Function Equals(obj As Object) As Boolean

        If obj = Nothing Then
            Return False
        End If

        If [GetType]() <> obj.GetType() Then
            Return False
        End If

        Dim pt As Point = CType(obj, Point)

        Return X = pt.X AndAlso Y = pt.Y

    End Function

    Public Shared Operator =(pt1 As Point, pt2 As Point) As Boolean
        ' Object.Equals calls Point.Equals(Object).
        Return Object.Equals(pt1, pt2)
    End Operator

    Public Shared Operator <>(pt1 As Point, pt2 As Point) As Boolean
        ' Object.Equals calls Point.Equals(Object).
        Return Not Object.Equals(pt1, pt2)
    End Operator

End Class
```

相关规则

- CA1046:不要对引用类型重载相等运算符
- CA2218:重写 Equals 时重写 GetHashCode
- CA2225:运算符重载具有命名的备用项
- CA2226:运算符应有对称重载
- CA2231:重写 ValueType.Equals 时应重载相等运算符

另请参阅

- CS0660

CA2225:运算符重载具有命名的备用项

2021/11/16 •

	▮
■ ID	CA2225
■	使用情况
▬▬▬	非中断

原因

检测到运算符重载，但未找到预期的命名备用方法。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

运算符重载允许使用符号来表示类型的计算。例如，重载加号 `+` 以进行加法的类型通常会具有一个名为 `Add` 的备用成员。命名的备用成员提供了对与运算符相同的功能的访问。它针对相关开发人员提供，这些开发人员使用不支持重载运算符的语言进行编程。

此规则检查以下内容：

- 通过检查名为 `To<typename>` 和 `From<typename>` 的方法在类型中隐式和显式强制转换运算符。
- 下表中列出了这些运算符：

C#	VISUAL BASIC	C++	▮▮▮▮▮▮
+（二元）	+	+（二元）	添加
+=	+=	+=	添加
&	且	&	BitwiseAnd
&=	And=	&=	BitwiseAnd
|	或	|	BitwiseOr
|=	Or=	|=	BitwiseOr
--	不可用	--	递减
/	/	/	除
/=	/=	/=	除
==	=	==	等于

C#	VISUAL BASIC	C++	[illegible]
^	Xor	^	Xor
^=	Xor=	^=	Xor
>	>	>	CompareTo 或 Compare
>=	>=	>=	CompareTo 或 Compare
++	不可用	++	增量
!=	<>	!=	等于
<<	<<	<<	LeftShift
<<=	<<=	<<=	LeftShift
<	<	<	CompareTo 或 Compare
<=	<=	<=	CompareTo 或 Compare
&&	不可用	&&	LogicalAnd
||	不可用	||	LogicalOr
!	不可用	!	LogicalNot
%	Mod	%	Mod 或 Remainder
%=	不可用	%=	Mod
*（二元）	*	*	乘
*=	不可用	*=	乘
~	Not	~	OnesComplement
>>	>>	>>	RightShift
=	不可用	>>=	RightShift
-（二元）	-（二元）	-（二元）	减
-=	不可用	-=	减
true	IsTrue	不可用	IsTrue（属性）
- (unary)	不可用	-	Negate
+（一元）	不可用	+	Plus

C#	VISUAL BASIC	C++	tttttt
false	IsFalse	False	IsTrue(属性)

*N/A 表示运算符不能重载为所选的语言。

> **NOTE**
> 在 C# 中，重载二元运算符时，也会隐式重载相应的赋值运算符(若有)。

如何解决冲突

若要解决此规则的冲突，请为运算符实现备用方法。使用建议的备用名称为其命名。

何时禁止显示警告

如果要实现共享库，请勿禁止显示此规则的警告。应用程序可以忽略此规则发出的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别(使用情况)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例定义了与此规则冲突的结构。若要更正此示例，请向该结构添加公共 `Add(int x, int y)` 方法。

```
public struct Point
{
    private int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override string ToString()
    {
        return String.Format("({0},{1})", x, y);
    }

    // Violates rule: OperatorOverloadsHaveNamedAlternates.
    public static Point operator +(Point a, Point b)
    {
        return new Point(a.x + b.x, a.y + b.y);
    }

    public int X { get { return x; } }
    public int Y { get { return x; } }
}
```

相关规则

- CA1046:不要对引用类型重载相等运算符
- CA2226:运算符应有对称重载
- CA2231:重写 ValueType.Equals 时应重载相等运算符

CA2226:运算符应有对称重载

2021/11/16 •

■ ID	CA2226
■	使用情况
■■■■■	非中断

原因

某个类型实现了相等运算符或不等运算符，却未实现相反运算符。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

不存在相等或不等运算符适用于某个类型的实例却未定义对应的相反运算符的情况。类型通常通过返回相等运算符的求反值来实现不等运算符。

对于与此规则的冲突，C# 编译器会发出错误。

如何解决冲突

若要解决与此规则的冲突，请同时实现相等和不等运算符，或删除存在的运算符。

何时禁止显示警告

不禁止显示此规则发出的警告。如果禁止了，类型将无法以与 .NET 一致的方式工作。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别（使用情况）中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

相关规则

- CA1046:不要对引用类型重载相等运算符

- CA2225:运算符重载具有命名的备用项
- CA2231:重写 ValueType.Equals 时应重载相等运算符

CA2227:集合属性应为只读

2021/11/16 •

■ ID	CA2227
■	使用情况
■■■■■	重大

原因

外部可见的可写属性属于实现 System.Collections.ICollection 的类型。此规则将忽略数组、索引器(名称为"Item"的属性)、不可变集合、只读集合和权限集。

规则说明

使用可写的集合属性，用户可以将该集合替换为完全不同的集合。只读或 init-only 属性禁止替换该集合，但仍允许设置单个成员。如果目标是替换集合，则首选的设计模式是包含一个从集合中移除所有元素的方法，以及一个重新填充集合的方法。有关此情况的示例，请参阅 System.Collections.ArrayList 类的 Clear 和 AddRange 方法。

二进制和 XML 序列化都支持作为集合的只读属性。为了实现可序列化，System.Xml.Serialization.XmlSerializer 类对实现 ICollection 和 System.Collections.IEnumerable 的类型具有特定的要求。

如何解决冲突

若要解决此规则的冲突，请将属性设置为只读或 init-only。如果设计需要，请添加方法以清除集合并重新填充。

何时禁止显示警告

如果属性是数据传输对象 (DTO) 类的一部分，则可禁止显示警告。

否则，请不要禁止显示此规则的警告。

示例

下面的示例演示一个具有可写集合属性的类型，并演示如何直接替换该集合。此外，它还演示了使用 `Clear` 和 `AddRange` 方法替换只读集合属性的首选方式。

```
public class WritableCollection
{
    public ArrayList SomeStrings
    {
        get;

        // This set accessor violates rule CA2227.
        // To fix the code, remove this set accessor or change it to init.
        set;
    }

    public WritableCollection()
    {
        SomeStrings = new ArrayList(new string[] { "one", "two", "three" });
    }
}

class ReplaceWritableCollection
{
    static void Main2227()
    {
        ArrayList newCollection = new ArrayList(new string[] { "a", "new", "collection" });

        WritableCollection collection = new WritableCollection();

        // This line of code demonstrates how the entire collection
        // can be replaced by a property that's not read only.
        collection.SomeStrings = newCollection;

        // If the intent is to replace an entire collection,
        // implement and/or use the Clear() and AddRange() methods instead.
        collection.SomeStrings.Clear();
        collection.SomeStrings.AddRange(newCollection);
    }
}
```

```
Public Class WritableCollection

    ' This property violates rule CA2227.
    ' To fix the code, add the ReadOnly modifier to the property:
    ' ReadOnly Property SomeStrings As ArrayList
    Property SomeStrings As ArrayList

    Sub New()
        SomeStrings = New ArrayList(New String() {"one", "two", "three"})
    End Sub

End Class

Class ViolatingVersusPreferred

    Shared Sub Main2227()
        Dim newCollection As New ArrayList(New String() {"a", "new", "collection"})

        Dim collection As New WritableCollection()

        ' This line of code demonstrates how the entire collection
        ' can be replaced by a property that's not read only.
        collection.SomeStrings = newCollection

        ' If the intent is to replace an entire collection,
        ' implement and/or use the Clear() and AddRange() methods instead.
        collection.SomeStrings.Clear()
        collection.SomeStrings.AddRange(newCollection)
    End Sub

End Class
```

相关规则

- CA1819:属性不应返回数组

另请参阅

- C# 中的属性

CA2229:实现序列化构造函数

2021/11/16 •

	[
■ ID	CA2229
■	使用情况
▬▬▬	非中断

原因

该类型实现 System.Runtime.Serialization.ISerializable 接口，它不是委托或接口，并且满足以下条件之一：

- 该类型不具有采用 SerializationInfo 对象和 StreamingContext 对象（序列化构造函数的签名）的构造函数。
- 该类型是非密封的，其序列化构造函数的访问修饰符不受保护（系列）。
- 该类型是密封的，其序列化构造函数的访问修饰符不是专用的。

规则说明

此规则适用于支持自定义序列化的类型。如果类型可实现 ISerializable 接口，则它支持自定义序列化。需要序列化构造函数来对已使用 ISerializable.GetObjectData 方法序列化的对象进行反序列化或重新创建这些对象。

如何解决冲突

要修复与该规则的冲突，请实现序列化构造函数。对于密封类，请使构造函数成为私有；否则，请使构造函数成为受保护。

何时禁止显示警告

请勿禁止显示与此规则的冲突。该类型将不可反序列化，并且在许多情况下将不起作用。

示例

以下示例显示了满足此规则的类型。

```
[Serializable]
public class SerializationConstructorsRequired : ISerializable
{
    private int n1;

    // This is a regular constructor.
    public SerializationConstructorsRequired()
    {
        n1 = -1;
    }
    // This is the serialization constructor.
    // Satisfies rule: ImplementSerializationConstructors.

    protected SerializationConstructorsRequired(
       SerializationInfo info,
       StreamingContext context)
    {
        n1 = (int)info.GetValue(nameof(n1), typeof(int));
    }

    // The following method serializes the instance.
    [SecurityPermission(SecurityAction.LinkDemand,
        Flags = SecurityPermissionFlag.SerializationFormatter)]
    void ISerializable.GetObjectData(SerializationInfo info,
       StreamingContext context)
    {
        info.AddValue(nameof(n1), n1);
    }
}
```

相关规则

CA2237:用 SerializableAttribute 标记 ISerializable 类型

另请参阅

- System.Runtime.Serialization.ISerializable
- System.Runtime.Serialization.SerializationInfo
- System.Runtime.Serialization.StreamingContext

CA2231:重写 ValueType.Equals 时应重载相等运算符

2021/11/16 •

	[illegible]
■ ID	CA2231
■	使用情况
■■■■■	非中断

原因

值类型重写 System.Object.Equals，但不实现相等运算符。

默认情况下，此规则仅查看外部可见的类型，但这是可配置的。

规则说明

在大多数编程语言中，对值类型不存在默认的相等运算符 (==) 的实现。如果编程语言支持运算符重载，则应考虑实现相等运算符。其行为应与 Equals 的行为相同。

不能在相等运算符的重载实现中使用默认的相等运算符。这样做将导致堆栈溢出。若要实现相等运算符，请在实现中使用 Object.Equals 方法。例如：

```
If (Object.ReferenceEquals(left, Nothing)) Then
    Return Object.ReferenceEquals(right, Nothing)
Else
    Return left.Equals(right)
End If
```

```
if (Object.ReferenceEquals(left, null))
    return Object.ReferenceEquals(right, null);
return left.Equals(right);
```

如何解决冲突

若要解决与此规则的冲突，请实现相等运算符。

何时禁止显示警告

可以安全地禁止显示此规则发出的警告；但是，我们建议尽可能提供相等运算符。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别(使用情况)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例定义了违反此规则的类型：

```
public struct PointWithoutHash
{
    private int x, y;

    public PointWithoutHash(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public override string ToString()
    {
        return String.Format("({0},{1})", x, y);
    }

    public int X { get { return x; } }

    public int Y { get { return x; } }

    // Violates rule: OverrideGetHashCodeOnOverridingEquals.
    // Violates rule: OverrideOperatorEqualsOnOverridingValueTypeEquals.
    public override bool Equals(object obj)
    {
        if (obj.GetType() != typeof(PointWithoutHash))
            return false;

        PointWithoutHash p = (PointWithoutHash)obj;
        return ((this.x == p.x) && (this.y == p.y));
    }
}
```

相关规则

- CA1046:不要对引用类型重载相等运算符
- CA2225:运算符重载具有命名的备用项
- CA2226:运算符应有对称重载

另请参阅

- System.Object.Equals

CA2234:传递 System.Uri 对象，而不传递字符串

2021/11/16 •

■ ID	CA2234
■	使用情况
▬▬▬	非中断

原因

已调用具有字符串参数的方法，该字符串参数的名称包含 “uri”、“Uri”、“urn”、“url”或“Url”，而且该方法的声明类型包含一个具有 System.Uri 参数的相应方法重载。

默认情况下，此规则仅查看外部可见的方法和类型，但这是可配置的。

规则说明

参数名称根据 camel 大小写约定拆分为标记，然后此规则会检查每个标记是否为“uri”、“Uri”、“urn”、“Urn”、“url”或“Url”。如果存在匹配项，则假定参数表示统一资源标识符 (URI)。URI 的字符串表示形式容易导致分析和编码错误，并且可造成安全漏洞。Uri 类以一种安全的方式提供这些服务。如果在两个重载之间进行选择，而这两个重载只是在 URI 的表示形式方面存在差异，用户应选择采用 Uri 参数的重载。

如何解决冲突

若要解决此规则的冲突，请调用采用 Uri 参数的重载。

何时禁止显示警告

如果参数不表示 URI，可禁止显示此规则的警告。

配置代码以进行分析

使用下面的选项来配置代码库的哪些部分要运行此规则。

- 包含特定的 API 图面

可以仅为此规则、为所有规则或为此类别(使用情况)中的所有规则配置此选项。有关详细信息，请参阅代码质量规则配置选项。

包含特定的 API 图面

你可以根据代码库的可访问性，配置要针对其运行此规则的部分。例如，若要指定规则应仅针对非公共 API 图面运行，请将以下键值对添加到项目中的 .editorconfig 文件：

```
dotnet_code_quality.CAXXXX.api_surface = private, internal
```

示例

下面的示例演示了与规则冲突的 `ErrorProne` 方法，以及正确调用 Uri 重载的 `SaferWay` 方法：

```
Imports System

Namespace ca2234

    Class History

        Friend Sub AddToHistory(uriString As String)
        End Sub

        Friend Sub AddToHistory(uriType As Uri)
        End Sub

    End Class

    Public Class Browser

        Dim uriHistory As New History()

        Sub ErrorProne()
            uriHistory.AddToHistory("http://www.adventure-works.com")
        End Sub

        Sub SaferWay()
            Try
                Dim newUri As New Uri("http://www.adventure-works.com")
                uriHistory.AddToHistory(newUri)
            Catch uriException As UriFormatException
            End Try
        End Sub

    End Class

End Namespace
```

```
class History
{
    internal void AddToHistory(string uriString) { }
    internal void AddToHistory(Uri uriType) { }
}

public class Browser
{
    History uriHistory = new History();

    public void ErrorProne()
    {
        uriHistory.AddToHistory("http://www.adventure-works.com");
    }

    public void SaferWay()
    {
        try
        {
            Uri newUri = new Uri("http://www.adventure-works.com");
            uriHistory.AddToHistory(newUri);
        }
        catch (UriFormatException) { }
    }
}
```

相关规则

- CA1056:URI 属性不应是字符串
- CA1054:URI 参数不应为字符串
- CA1055:URI 返回值不应是字符串

CA2235:标记所有不可序列化的字段

2021/11/16 •

	[
■ ID	CA2235
■	使用情况
▬▬▬	非中断

原因

在可以序列化的类型中声明了类型不可序列化的实例字段。

规则说明

可序列化的类型是标记有 System.SerializableAttribute 特性的类型。序列化类型时，如果该类型包含不可序列化且不可实现 System.Runtime.Serialization.ISerializable 接口的类型的实例字段，则将引发 System.Runtime.Serialization.SerializationException 异常。

> **TIP**
> 对于实现 ISerializable 的类型的实例字段，不会触发 CA2235，因为它们提供了自己的序列化逻辑。

如何解决冲突

若要解决与此规则的冲突，请将 System.NonSerializedAttribute 特性应用于不可序列化的字段。

何时禁止显示警告

仅当声明了允许对字段实例进行序列化和反序列化的 System.Runtime.Serialization.ISerializationSurrogate 类型时，才可禁止显示此规则的警告。

示例

下面的示例演示了两种类型：一种违反了规则，另一种满足了规则。

```
public class Mouse
{
    int buttons;
    string scanTypeValue;

    public int NumberOfButtons
    {
        get { return buttons; }
    }

    public string ScanType
    {
        get { return scanTypeValue; }
    }

    public Mouse(int numberOfButtons, string scanType)
    {
        buttons = numberOfButtons;
        scanTypeValue = scanType;
    }
}

[Serializable]
public class InputDevices1
{
    // Violates MarkAllNonSerializableFields.
    Mouse opticalMouse;

    public InputDevices1()
    {
        opticalMouse = new Mouse(5, "optical");
    }
}

[Serializable]
public class InputDevices2
{
    // Satisfies MarkAllNonSerializableFields.
    [NonSerialized]
    Mouse opticalMouse;

    public InputDevices2()
    {
        opticalMouse = new Mouse(5, "optical");
    }
}
```

```
Imports System
Imports System.Runtime.Serialization

Namespace ca2235

    Public Class Mouse

        ReadOnly Property NumberOfButtons As Integer

        ReadOnly Property ScanType As String

        Sub New(numberOfButtons As Integer, scanType As String)
            Me.NumberOfButtons = numberOfButtons
            Me.ScanType = scanType
        End Sub

    End Class

    <SerializableAttribute>
    Public Class InputDevices1

        ' Violates MarkAllNonSerializableFields.
        Dim opticalMouse As Mouse

        Sub New()
            opticalMouse = New Mouse(5, "optical")
        End Sub

    End Class

    <SerializableAttribute>
    Public Class InputDevices2

        ' Satisfies MarkAllNonSerializableFields.
        <NonSerializedAttribute>
        Dim opticalMouse As Mouse

        Sub New()
            opticalMouse = New Mouse(5, "optical")
        End Sub

    End Class

End Namespace
```

注解

规则 CA2235 不会分析实现 ISerializable 接口的类型(除非它们标记有 SerializableAttribute 特性)。这是因为规则 CA2237 已建议使用 SerializableAttribute 特性来标记实现 ISerializable 接口的类型。

相关规则

- CA2229:实现序列化构造函数
- CA2237:用 SerializableAttribute 标记 ISerializable 类型

CA2237:用 SerializableAttribute 标记 ISerializable 类型

2021/11/16 •

■ ID	CA2237
■	使用情况
■■■■■	非中断

原因

外部可见类型实现 System.Runtime.Serialization.ISerializable 接口，且该类型没有使用 System.SerializableAttribute 属性进行标记。此规则会忽略基类型无法序列化的派生类型。

规则说明

若要被公共语言运行时识别为可序列化，类型必须用 SerializableAttribute 属性进行标记，即使该类型通过实现 ISerializable 接口使用自定义序列化例程也是如此。

如何解决冲突

若要解决此规则的冲突，请将 SerializableAttribute 属性应用于该类型。

何时禁止显示警告

请勿禁止显示此规则的异常类警告，因为它们必须是可序列化类才能在应用程序域中正常工作。

示例

下面的示例演示了与此规则冲突的类型。取消对 SerializableAttribute 属性行的评论以满足规则。

```
Imports System
Imports System.Runtime.Serialization
Imports System.Security.Permissions

Namespace ca2237

    ' <SerializableAttribute> _
    Public Class BaseType
        Implements ISerializable

        Dim baseValue As Integer

        Sub New()
            baseValue = 3
        End Sub

        Protected Sub New(
         info As SerializationInfo, context As StreamingContext)

            baseValue = info.GetInt32("baseValue")

        End Sub

        <SecurityPermissionAttribute(SecurityAction.Demand,
          SerializationFormatter:=True)>
        Overridable Sub GetObjectData(
         info As SerializationInfo, context As StreamingContext) _
         Implements ISerializable.GetObjectData

            info.AddValue("baseValue", baseValue)

        End Sub

    End Class

End Namespace
```

```
// [SerializableAttribute]
public class BaseType : ISerializable
{
    int baseValue;

    public BaseType()
    {
        baseValue = 3;
    }

    protected BaseType(
       SerializationInfo info, StreamingContext context)
    {
        baseValue = info.GetInt32("baseValue");
    }

    [SecurityPermissionAttribute(SecurityAction.Demand,
        SerializationFormatter = true)]
    public virtual void GetObjectData(
       SerializationInfo info, StreamingContext context)
    {
        info.AddValue("baseValue", baseValue);
    }
}
```

相关规则

- CA2229:实现序列化构造函数
- CA2235:标记所有不可序列化的字段

CA2241:为格式化方法提供正确的参数

2021/11/16 •

■ ID	CA2241
■	使用情况
▬▬▬	非中断

原因

传递到 WriteLine、Write 或 System.String.Format 等方法的 `format` 字符串参数不包含与每个对象参数相对应的格式项，反之亦然。

默认情况下，此规则仅分析对之前提到的三种方法的调用，但这是可配置的。

规则说明

WriteLine、Write 和 Format 等方法的参数由格式字符串后跟若干个 System.Object 实例构成。格式字符串由文本和嵌入的格式项组成，其形式为 {index[,alignment][:formatString]}。"index"是一个从零开始的整数，用于指示要格式化的对象。如果对象在格式字符串中没有相应的索引，则忽略该对象。如果"index"指定的对象不存在，则会在运行时引发 System.FormatException。

如何解决冲突

若要解决与此规则的冲突，请为每个对象参数提供一个格式项，并为每个格式项提供一个对象参数。

何时禁止显示警告

不禁止显示此规则发出的警告。

配置代码以进行分析

使用以下选项来配置针对其运行此规则的其他方法。

- 其他字符串格式设置方法
- 自动确定其他字符串格式设置方法

其他字符串格式设置方法

可以配置应由此规则分析的其他字符串格式设置方法的名称。例如，若要将名为 `MyFormat` 的所有方法指定为字符串格式设置方法，可将以下键值对添加到项目的 .editorconfig 文件中：

```
dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat
```

选项值中允许的方法名称格式（用 `|` 分隔）：

- 仅方法名称（包括具有相应名称的所有方法，不考虑包含类型或命名空间）

- 完全限定的名称，使用符号的文档 ID 格式，前缀为 `M:`（可选）。

示例：

`dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat`	匹配编译中所有名为 `MyFormat` 的方法
`dotnet_code_quality.CA2241.additional_string_formatting_methods = MyFormat1|MyFormat2`	匹配编译中所有名为 `MyFormat1` 或 `MyFormat2` 的方法
`dotnet_code_quality.CA2241.additional_string_formatting_methods = NS.MyType.MyFormat(ParamType)`	将特定方法 `MyFormat` 与给定的完全限定签名相匹配
`dotnet_code_quality.CA2241.additional_string_formatting_methods = NS1.MyType1.MyFormat1(ParamType)|NS2.MyType2.MyFormat2(ParamType)`	将特定方法 `MyFormat1` 和 `MyFormat2` 与相应的完全限定签名相匹配

自动确定其他字符串格式设置方法

可以配置分析器来自动尝试确定字符串格式设置方法，而不是指定其他字符串格式设置方法的显式列表。默认情况下禁用此选项。如果启用该选项，则任何具有 `string format` 参数后跟 `params object[]` 参数的方法都会被视作字符串格式设置方法：

```
dotnet_code_quality.CA2241.try_determine_additional_string_formatting_methods_automatically = true
```

示例

下面的示例显示了两个规则冲突。

```
Imports System

Namespace ca2241

    Class CallsStringFormat

        Sub CallFormat()

            Dim file As String = "file name"
            Dim errors As Integer = 13

            ' Violates the rule.
            Console.WriteLine(String.Format("{0}", file, errors))

            Console.WriteLine(String.Format("{0}: {1}", file, errors))

            ' Violates the rule and generates a FormatException at runtime.
            Console.WriteLine(String.Format("{0}: {1}, {2}", file, errors))

        End Sub

    End Class

End Namespace
```

```
class CallsStringFormat
{
    void CallFormat()
    {
        string file = "file name";
        int errors = 13;

        // Violates the rule.
        Console.WriteLine(string.Format("{0}", file, errors));

        Console.WriteLine(string.Format("{0}: {1}", file, errors));

        // Violates the rule and generates a FormatException at runtime.
        Console.WriteLine(string.Format("{0}: {1}, {2}", file, errors));
    }
}
```

CA2242:正确测试 NaN

2021/11/16 •

■ ID	CA2242
■	使用情况
▬▬▬	非中断

原因

针对 System.Single.NaN 或 System.Double.NaN 测试值的表达式。

规则说明

System.Double.NaN，它表示非数值的值，算术运算未定义时会得到这样的值。测试一个值和 System.Double.NaN 是否相等的任何表达式都始终返回 `false`。测试一个值和 System.Double.NaN 是否不相等（C# 中的 `!=`）的任何表达式都始终返回 `true`。

如何解决冲突

若要解决与此规则的冲突，并准确确定某个值是否表示 System.Double.NaN，请使用 System.Single.IsNaN 或 System.Double.IsNaN 来测试该值。

何时禁止显示警告

不禁止显示此规则发出的警告。

示例

下面的示例显示了错误地针对 System.Double.NaN 测试值的两个表达式，以及正确使用 System.Double.IsNaN 来测试值的一个表达式。

```
Imports System

Namespace ca2242

    Class NaNTests

        Shared zero As Double

        Shared Sub Main2242()
            Console.WriteLine(0 / zero = Double.NaN)
            Console.WriteLine(0 / zero <> Double.NaN)
            Console.WriteLine(Double.IsNaN(0 / zero))
        End Sub

    End Class

End Namespace
```

```
class NaNTests
{
    static double zero;

    static void Main()
    {
        Console.WriteLine(0 / zero == double.NaN);
        Console.WriteLine(0 / zero != double.NaN);
        Console.WriteLine(double.IsNaN(0 / zero));
    }
}
```

CA2243:特性字符串文本应正确分析

2021/11/16 •

	[
■ ID	CA2243
■	使用情况
▬▬▬	非中断

原因

特性的字符串文本参数不能正确解析为 URL、GUID 或版本。

规则说明

由于特性从 System.Attribute 派生而来，并在编译时使用，因此只能将常数值传递给其构造函数。必须表示 URL、GUID 和版本的特性参数不能类型化为 System.Uri、System.Guid 和 System.Version，因为这些类型不能表示为常量。而必须将其表示为字符串。

由于参数已类型化为字符串，因此可能会在编译时传递格式错误的参数。

此规则使用命名启发式方法查找表示统一资源标识符 (URI)、全局唯一标识符 (GUID) 或版本的参数，并验证所传递的值是否正确。

如何解决冲突

将参数字符串更改为格式正确的 URL、GUID 或版本。

何时禁止显示警告

如果参数不表示 URL、GUID 或版本，可禁止显示此规则的警告。

示例

下面的示例演示了与此规则冲突的 AssemblyFileVersionAttribute 的代码。

```
[AttributeUsage(AttributeTargets.Assembly, Inherited = false)]
[ComVisible(true)]
public sealed class AssemblyFileVersionAttribute : Attribute
{
    public AssemblyFileVersionAttribute(string version) { }

    public string Version { get; set; }
}

// Since the parameter is typed as a string, it is possible
// to pass an invalid version number at compile time. The rule
// would be violated by the following code: [assembly : AssemblyFileVersion("xxxxx")]
```

规则由以下参数触发：

- 包含“version”且无法解析为 System.Version 的参数。
- 包含“guid”且无法解析为 System.Guid 的参数。
- 包含“uri”、“urn”或“url”且无法解析为 System.Uri 的参数。

另请参阅

- CA1054:URI 参数不应为字符串

CA2244:不要复制已索引的元素初始值设定项

2021/11/16 •

	▮
■ ID	CA2244
■	使用情况
▬▬▬	非中断

原因

对象初始值设定项有多个具有相同常量索引的索引元素初始值设定项。除最后一个初始值设定项之外，其余都是冗余的。

规则说明

使用对象初始值设定项，你可以在创建对象时向对象的任何可访问字段或属性分配值，而无需调用后跟赋值语句行的构造函数。

对象初始值设定项中的索引元素初始值设定项须初始化具有唯一性的元素。重复的索引将覆盖上一个元素初始值设定项。

如何解决冲突

若要解决冲突，请删除所有后续元素初始值设定项覆盖的所有冗余索引元素初始值设定项。例如，以下代码片段显示了规则冲突和几种可能的修复方法：

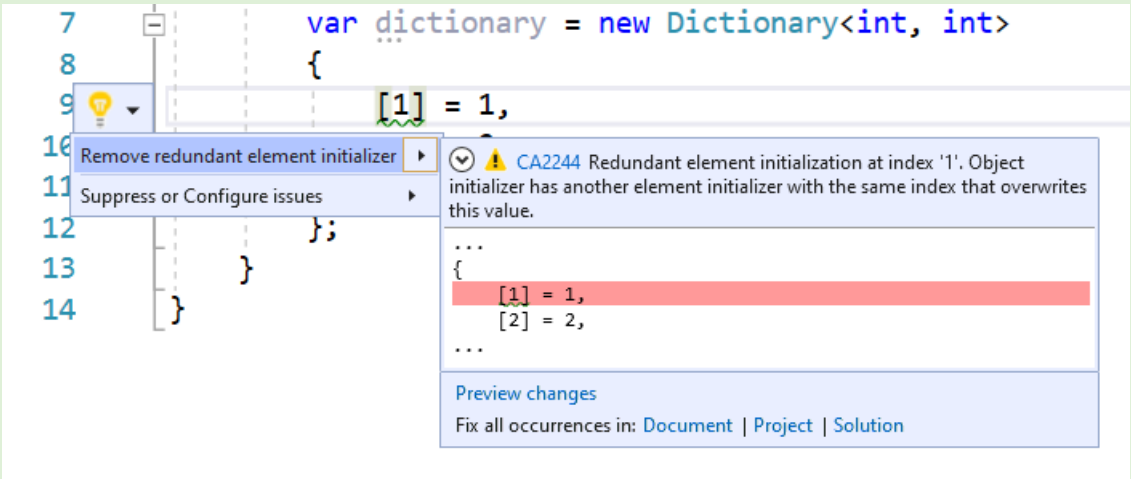
```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [1] = 1, // CA2244
            [2] = 2,
            [1] = 3
        };
    }
}
```

```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [2] = 2,
            [1] = 3
        };
    }
}
```

```
using System.Collections.Generic;

class C
{
    public void M()
    {
        var dictionary = new Dictionary<int, int>
        {
            [1] = 1,
            [2] = 2
        };
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修复。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“删除冗余的元素初始值设定项”。



何时禁止显示警告

请勿禁止显示与此规则的冲突。

另请参阅

- 用法规则

CA2245:请勿将属性分配给其自身

2021/11/16 •

	[
■ ID	CA2245
■	使用情况
▬▬▬	非中断

原因

属性意外赋值给了其自身。

规则说明

将字段符号、本地符号或参数符号赋值给其自身时，C# 编译器将生成警告 CS1717：对同一变量进行赋值；是否希望对其他变量赋值？。如果本地符号、参数符号或字段符号的名称类似于作用域中的另一个符号，这种错误很常见。不要在赋值的左右两侧使用不同的符号，而是在两侧使用相同的符号。这会导致对值自身进行冗余赋值，通常指示功能性 bug。

对于几乎所有真实情况，将属性赋值给自身也是一个类似的功能性 bug。但是，在某些极端情况下，提取属性值可能会产生副作用，并且该属性的新值与原始值不同。如果是这样，则将属性赋值给自身不是冗余的，并且不能删除。这会阻止编译器针对将属性赋值给自身生成 `CS1717` 警告，并且不会因这些情况产生中断性变更。

规则 `CA2245` 旨在弥补这一缺陷。它报告了将属性赋值给自身的冲突，以帮助解决这些功能性 bug。对于需要将属性赋值给自身的一小部分极端情况，可以在具有适当理由注释的源中禁止显示 `CA2245` 冲突警告。

如何解决冲突

若要解决冲突，请在赋值的左右两侧使用不同的符号。例如，以下代码片段显示了规则冲突及其解决方法：

```
public class C
{
    private int p = 0;
    public int P { get; private set; }

    public void M(int p)
    {
        // CS1717: Accidentally assigned the parameter 'p' to itself.
        p = p;

        // CA2245: Accidentally assigned the property 'P' to itself.
        P = P;
    }
}
```

```
public class C
{
    private int p = 0;
    public int P { get; private set; }

    public void M(int p)
    {
        // No violation, now the parameter is assigned to the field.
        this.p = p;

        // No violation, now the parameter is assigned to the property.
        P = p;
    }
}
```

何时禁止显示警告

如果提取属性值可能会产生副作用并且该属性的新值与原始值不同，则可禁止显示此规则的冲突警告。如果是这样，则将属性赋值给自身不是冗余的。应针对可禁止显示警告添加理由注释，以将此行为记录为预期行为。

相关规则

- CS1717：对同一变量进行赋值；是否希望对其他变量赋值？
- CA2011:请勿在其资源库中分配属性
- CA2246:请勿在同一语句中分配符号及其成员

另请参阅

- 用法规则

CA2246:请勿在同一语句中分配符号及其成员

2021/11/16 •

	[
■ ID	CA2246
■	使用情况
▬▬▬	非中断

原因

在同一语句中对符号及其成员进行了赋值。例如：

```
// 'a' and 'a.Field' are assigned in the same statement
a.Field = a = b;
```

规则说明

建议不要请勿在同一语句中对符号及其成员(字段或属性)进行赋值。目前尚不清楚成员访问是打算在赋值之前使用符号的旧值还是打算使用此语句中赋值的新值。为了清楚起见，必须将多赋值语句拆分为两个或更多个简单的赋值语句。

如何解决冲突

若要解决冲突，请将多赋值语句拆分为两个或更多简单的赋值语句。例如，以下代码片段显示了与此规则的冲突，以及根据用户意图进行解决的几种方法：

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // It is not clear if the user intent in the below statement is to assign to 'Instance1.Field' or
'Instance2.Field'.
        // CA2246: Symbol 'a' and its member 'Field' are both assigned in the same statement. You are at
risk of assigning the member of an unintended object.
        a.Field = a = b;
    }
}
```

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // 'Instance1.Field' is intended to be assigned.
        var instance1 = a;
        a = b;
        instance1.Field = a;
    }
}
```

```
public class C
{
    public C Field;
}

public class Test
{
    public void M(C a, C b)
    {
        // Let us assume 'a' points to 'Instance1' and 'b' points to 'Instance2' at the start of the method.
        // 'Instance2.Field' is intended to be assigned.
        a = b;
        b.Field = a; // or 'a.Field = a;'
    }
}
```

何时禁止显示警告

请勿禁止显示此规则的冲突警告。

相关规则

- CA2245:请勿将属性分配给其自身

另请参阅

- 用法规则

CA2247:传递给 TaskCompletionSource 构造函数的参数应为 TaskCreationOptions 枚举，而不是 TaskContinuationOptions 枚举

2021/11/16 •

	[
■ ID	CA2247
■	使用情况
▬▬▬	非中断

原因

使用 System.Threading.Tasks.TaskContinuationOptions 枚举值而不是 System.Threading.Tasks.TaskCreationOptions 枚举值构造 `System.Threading.Tasks.TaskCompletionSource`。使用 System.Object.ReferenceEquals 方法来测试一个或多个值类型是否相等。

规则说明

TaskCompletionSource 类型具有一个接受 System.Threading.Tasks.TaskCreationOptions 枚举值的构造函数和另一个接受 Object 的构造函数。意外传递 System.Threading.Tasks.TaskContinuationOptions 枚举值而不是 System.Threading.Tasks.TaskCreationOptions 枚举值会导致调用基于 Object 的构造函数：它将编译和运行，但它不具有预期的行为。

如何解决冲突

若要解决冲突，请将 System.Threading.Tasks.TaskContinuationOptions 枚举值替换为相应的 System.Threading.Tasks.TaskCreationOptions 枚举值。

```
// Violation
var tcs = new TaskCompletionSource<int>(TaskContinuationOptions.RunContinuationsAsynchronously);

// Fixed
var tcs = new TaskCompletionSource<int>(TaskCreationOptions.RunContinuationsAsynchronously);
```

何时禁止显示警告

此规则的冲突几乎总是突出调用代码中的 Bug，这样一来，代码的行为将与开发人员的预期不符，而 TaskCompletionSource 实际上忽略了指定的选项。唯一可禁止显示警告的情况是，开发人员实际上打算将装箱的 System.Threading.Tasks.TaskContinuationOptions 作为对象状态参数传递给 TaskCompletionSource。

另请参阅

- 用法规则

CA2248:向 Enum.HasFlag 提供正确的 enum 参数

2021/11/16 •

	[
■ ID	CA2248
■	使用情况
▬▬▬	非中断

原因

将枚举类型作为参数传递给 `HasFlag` 方法调用本质上与调用枚举类型不同。

规则说明

`Enum.HasFlag` 方法要求 `enum` 参数的 `enum` 类型与调用该方法的实例的相应类型相同。如果它们是不同的 `enum` 类型，则在运行时将引发未经处理的异常。

如何解决冲突

若要解决冲突，请在参数和调用方上使用相同的枚举类型：

```
public class C
{
    [Flags]
    public enum MyEnum { A, B, }

    [Flags]
    public enum OtherEnum { A, }

    public void Method(MyEnum m)
    {
        m.HasFlag(OtherEnum.A); // Enum types are different, this call will cause an `ArgumentException` to
be thrown at run time

        m.HasFlag(MyEnum.A); // Valid call
    }
}
```

何时禁止显示警告

请勿禁止显示此规则的冲突。

CA2249：请考虑使用 String.Contains 而不是 String.IndexOf

2021/11/16 •

■ ID	CA2249
■	使用情况
■■■■■	非中断

原因

此规则查找对 IndexOf 的调用，其中结果用于检查是否存在 substring，并建议使用 Contains 来提高可读性。

规则说明

当使用 IndexOf 来检查结果是否等于 `-1` 或大于等于 `0` 时，可安全地将该调用替换为 Contains，而不会对性能产生影响。

根据所使用的 IndexOf 重载，建议的解决方法可能是添加 `comparisonType` 参数：

`String.IndexOf(char)`	`String.Contains(char)`
`String.IndexOf(string)`	`String.Contains(string, StringComparison.CurrentCulture)`
`String.IndexOf(char, StringComparison.Ordinal)`	`String.Contains(char)`
`String.IndexOf(string, StringComparison.Ordinal)`	`String.Contains(string)`
`String.IndexOf(char, NON StringComparison.Ordinal)` *	`String.Contains(char, NON StringComparison.Ordinal)` *
`String.IndexOf(string, NON StringComparison.Ordinal)` *	`String.Contains(string, NON StringComparison.Ordinal)` *

* `StringComparison.Ordinal` 之外的任何 `StringComparison` 枚举值：

- CurrentCulture
- CurrentCultureIgnoreCase
- InvariantCulture
- InvariantCultureIgnoreCase
- OrdinalIgnoreCase

如何解决冲突

可以手动解决冲突，在某些情况下，也使用快速操作来修复 Visual Studio 中的代码。

示例

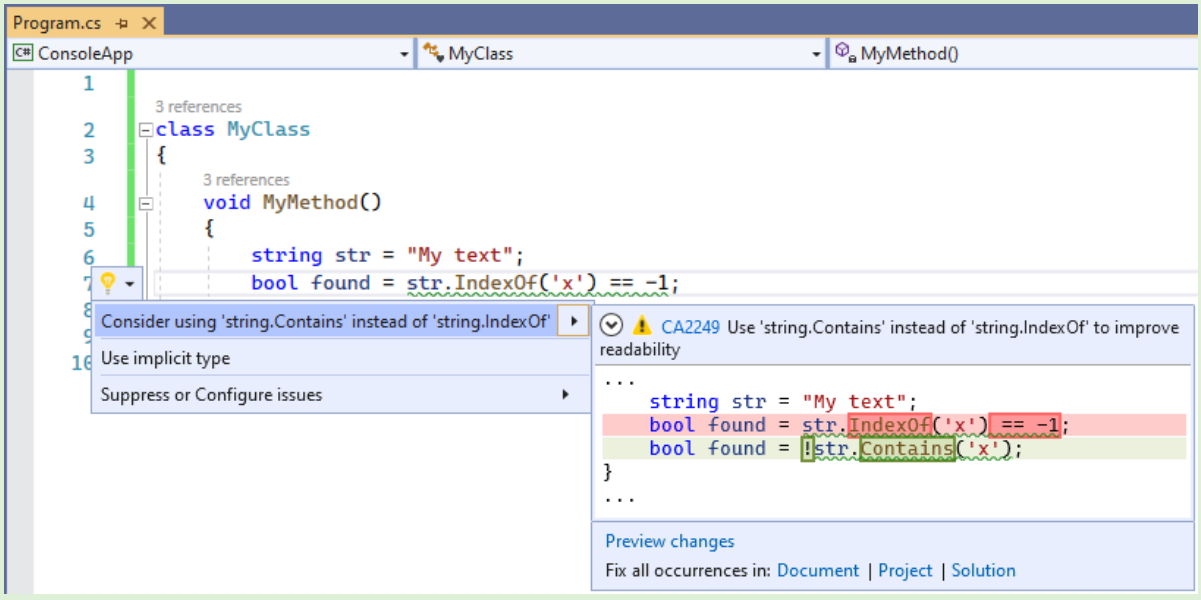
以下两个代码片段显示了 C# 中所有可能的规则冲突及其解决方法：

```
using System;

class MyClass
{
    void MyMethod()
    {
        string str = "My text";
        bool found;

        // No comparisonType in char overload, so no comparisonType added in resulting fix
        found = str.IndexOf('x') == -1;
        found = str.IndexOf('x') >= 0;

        // No comparisonType in string overload, adds StringComparison.CurrentCulture to resulting fix
        found = str.IndexOf("text") == -1;
        found = str.IndexOf("text") >= 0;

        // comparisonType equal to StringComparison.Ordinal, removes the argument
        found = str.IndexOf('x', StringComparison.Ordinal) == -1;
        found = str.IndexOf('x', StringComparison.Ordinal) >= 0;

        found = str.IndexOf("text", StringComparison.Ordinal) == -1;
        found = str.IndexOf("text", StringComparison.Ordinal) >= 0;

        // comparisonType different than StringComparison.Ordinal, preserves the argument
        found = str.IndexOf('x', StringComparison.OrdinalIgnoreCase) == -1;
        found = str.IndexOf('x', StringComparison.CurrentCulture) >= 0;

        found = str.IndexOf("text", StringComparison.InvariantCultureIgnoreCase) == -1;
        found = str.IndexOf("text", StringComparison.InvariantCulture) >= 0;

        // Suggestion message provided, but no automatic fix offered, must be fixed manually
        int index = str.IndexOf("text");
        if (index == -1)
        {
            Console.WriteLine("'text' Not found.");
        }
    }
}
```

```
using System;

class MyClass
{
    void MyMethod()
    {
        string str = "My text";
        bool found;

        // No comparisonType in char overload, so no comparisonType added in resulting fix
        found = !str.Contains('x');
        found = str.Contains('x');

        // No comparisonType in string overload, adds StringComparison.CurrentCulture to resulting fix
        found = !string.Contains("text", StringComparison.CurrentCulture);
        found = string.Contains("text", StringComparison.CurrentCulture);

        // comparisonType equal to StringComparison.Ordinal, removes the argument
        found = !string.Contains('x');
        found = string.Contains('x');

        found = !string.Contains("text");
        found = string.Contains("text");

        // comparisonType different than StringComparison.Ordinal, preserves the argument
        ;found = !string.Contains('x', StringComparison.OrdinalIgnoreCase)
        found = string.Contains('x', StringComparison.CurrentCulture);

        found = !string.Contains("text", StringComparison.InvariantCultureIgnoreCase);
        found = string.Contains("text", StringComparison.InvariantCulture);

        // This case had to be manually fixed
        if (!str.Contains("text"))
        {
            Console.WriteLine("'text' Not found.");
        }
    }
}
```

TIP

Visual Studio 中为此规则提供了代码修复。若要使用它，请将光标置于冲突上，然后按 Ctrl+。（句点）。从显示的选项列表中选择“考虑使用‘String.Contains’而不是‘String.IndexOf’”。



何时禁止显示警告

如果无需考虑改进代码可读性，则可禁止显示此规则的冲突。

另请参阅

- 用法规则

CA2250：使用 `ThrowIfCancellationRequested`

2021/11/16 •

■ ID	CA2250
■	使用情况
▬▬▬	非中断

原因

此规则标记在引发 OperationCanceledException 之前检查 IsCancellationRequested 的条件语句。

规则说明

可以通过调用 CancellationToken.ThrowIfCancellationRequested() 来完成相同的任务。

如何解决冲突

若要解决冲突，请将条件语句替换为对 ThrowIfCancellationRequested() 的调用。

```
using System;
using System.Threading;

public void MySlowMethod(CancellationToken token)
{
    // Violation
    if (token.IsCancellationRequested)
        throw new OperationCanceledException();

    // Fix
    token.ThrowIfCancellationRequested();

    // Violation
    if (token.IsCancellationRequested)
        throw new OperationCanceledException();
    else
        DoSomethingElse();

    // Fix
    token.ThrowIfCancellationRequested();
    DoSomethingElse();
}
```

```
Imports System
Imports System.Threading

Public Sub MySlowMethod(token As CancellationToken)

    ' Violation
    If token.IsCancellationRequested Then
        Throw New OperationCanceledException()
    End If

    ' Fix
    token.ThrowIfCancellationRequested()

    ' Violation
    If token.IsCancellationRequested Then
        Throw New OperationCanceledException()
    Else
        DoSomethingElse()
    End If

    ' Fix
    token.ThrowIfCancellationRequested()
    DoSomethingElse()
End Sub
```

何时禁止显示警告

可以安全地禁止显示此规则的警告。

另请参阅

- 用法警告

CA2251：使用 `String.Equals` 代替 `String.Compare`

2021/11/16 •

■ ID	CA2251
■	使用情况
■■■■■	非中断

原因

对 String.Compare 的调用结果与零进行比较。

规则说明

String.Compare 旨在生成可用于排序的总顺序比较。如果只关心字符串是否相等，使用 String.Equals 的等效重载会更清晰且可能更快。

如何解决冲突

若要解决此规则的冲突，请将比较 String.Compare 结果的表达式替换为对 String.Equals 的调用。

何时禁止显示警告

可以安全地禁止显示此规则的警告。

另请参阅

- 性能警告

CA2252：选择预览功能后再使用它们

2021/11/16 •

	[
■ ID	CA2252
■	Microsoft.Usage
▬▬▬	非中断

原因

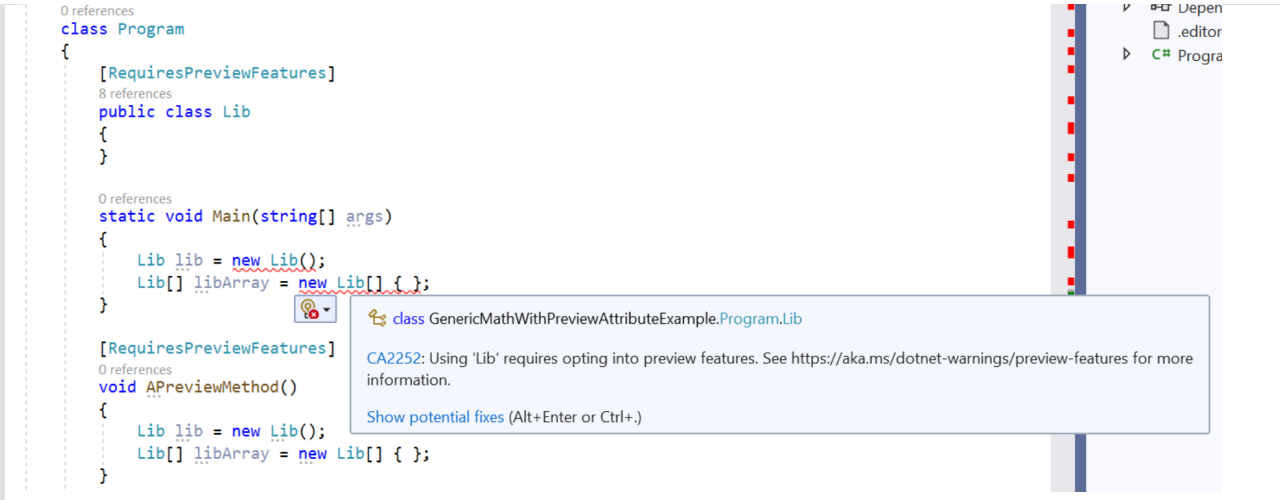
客户端在其程序集中使用预览 API 或类型，而无需在本地或在模块或程序集级别明确选择。

规则说明

当使用通过 RequiresPreviewFeaturesAttribute 属性修饰的 API 或程序集时，此规则会检查调用站点是否已选择预览功能。如果以下情况之一适用，调用站点已选择预览功能：

- 它在 `RequiresPreviewFeaturesAttribute` 注释的范围内。
- 它是已选择预览功能的程序集或模块的一部分。

下图显示了 CA2252 诊断的示例。



此处，`Lib` 是在 `Main` 方法中构造的预览类型。`Main` 本身没有被注释为预览方法，因此在 `Main` 内的两个构造函数调用上产生诊断。

如何解决冲突

有两种方法可以解决违规问题：

- 通过使用 `RequiresPreviewFeaturesAttribute` 注释其父级，将调用站点置于注释范围内。在前面的示例中，`APreviewMethod` 使用 `RequiresPreviewFeatures` 属性进行注释，因此分析器会忽略 `APreviewMethod` 中的预览类型用法。因此，`APreviewMethod` 的调用方必须执行类似的练习。

- 还可以在程序集或模块级别选择预览功能。这向分析器表明需要程序集中的预览类型用法，因此，此规则不会生成任何错误。这是使用预览依赖项的首选方式。若要在整个程序集中启用预览功能，请在 `.csproj` 文件中设置 EnablePreviewFeatures 属性：

```
<PropertyGroup>
  <EnablePreviewFeatures>true</EnablePreviewFeatures>
</PropertyGroup>
```

何时禁止显示错误

仅建议在需要明确禁用 API 诊断的高级用例中抑制此规则中的错误。在这种情况下，必须愿意承担适当标记预览 API 的责任。例如，考虑现有类型实现新预览接口的情况。由于不能将整个类型标记为预览（为了向后兼容性），因此可以在本地禁用围绕类型定义的诊断。此外，需要将预览接口实现标记为预览。现在，可以像以前一样使用现有类型，但对新接口方法的调用将获得诊断信息。System.Private.CoreLib.csproj 使用此技术公开数字类型（例如 `Int32`、`Double` 和 `Decimal`）的通用数学功能。下图显示了如何在本地禁用 CA2252 分析器。

```
public readonly struct Int32 : IComparable, IConvertible, ISpanFormattable, IComparable<int>, IEquatable<int>
#if FEATURE_GENERIC_MATH
#pragma warning disable SA1001, CA2252 // SA1001: Comma positioning; CA2252: Preview Features
    , IBinaryInteger<int>,
      IMinMaxValue<int>,
      ISignedNumber<int>
#pragma warning restore SA1001, CA2252
#endif // FEATURE_GENERIC_MATH
```

```
//
// IBinaryInteger
//

[RequiresPreviewFeatures]
static int IBinaryInteger<int>.LeadingZeroCount(int value)
    => BitOperations.LeadingZeroCount((uint)value);

[RequiresPreviewFeatures]
static int IBinaryInteger<int>.PopCount(int value)
    => BitOperations.PopCount((uint)value);

[RequiresPreviewFeatures]
static int IBinaryInteger<int>.RotateLeft(int value, int rotateAmount)
    => (int)BitOperations.RotateLeft((uint)value, rotateAmount);

[RequiresPreviewFeatures]
static int IBinaryInteger<int>.RotateRight(int value, int rotateAmount)
    => (int)BitOperations.RotateRight((uint)value, rotateAmount);

[RequiresPreviewFeatures]
static int IBinaryInteger<int>.TrailingZeroCount(int value)
    => BitOperations.TrailingZeroCount(value);
```

另请参阅

- EnablePreviewFeatures 和 GenerateRequiresPreviewFeaturesAttribute
- 预览功能设计文档

代码样式规则

2021/11/16 •

.NET 代码样式分析提供目标在于在代码库中保持一致的代码样式的规则。这些规则的规则 ID 具有“IDE”前缀。大多数规则都有关联的选项用于自定义首选样式。这些规则分为以下几个子类别：

- 语言规则

  与 C# 或 Visual Basic 语言相关的规则。例如，可以指定定义变量时有关使用 `var` 的规则，或有关是否首选 expression-bodied 成员的规则。

- 不必要的代码规则

  与不必要的代码有关的规则，指示可能的可读性、可维护性、性能或功能问题。例如，方法或未使用的私有字段、属性或方法中无法访问的代码是不必要的代码。

- 格式设置规则

  与代码的布局和结构有关的规则，其作用是使代码更易于阅读。例如，可以指定有关 Allman 大括号或控制块中是否首选空格的规则。

- 命名规则

  与代码元素命名相关的规则。例如，可以指定 `async` 方法名必须具有“Async”后缀。

- 杂项规则

  不属于其他类别的规则。

索引

下表按 ID 和选项列出了所有代码样式规则（如果有）。

ID		
IDE0001	简化名称	
IDE0002	简化成员访问	
IDE0003	删除 `this` 或 `Me` 限定	dotnet_style_qualification_for_field dotnet_style_qualification_for_property dotnet_style_qualification_for_method dotnet_style_qualification_for_event
IDE0004	删除不必要的 cast	
IDE0005	删除不必要的 import	
IDE0007	用 `var` 替代显式类型	csharp_style_var_for_built_in_types csharp_style_var_when_type_is_apparent csharp_style_var_elsewhere

ID		
IDE0008	用显式类型代替 `var`	csharp_style_var_for_built_in_types csharp_style_var_when_type_is_apparent csharp_style_var_elsewhere
IDE0009	添加 `this` 或 `Me` 限定	dotnet_style_qualification_for_field dotnet_style_qualification_for_property dotnet_style_qualification_for_method dotnet_style_qualification_for_event
IDE0010	将缺失的事例添加到 switch 语句	
IDE0011	添加大括号	csharp_prefer_braces
IDE0016	使用 throw 表达式	csharp_style_throw_expression
IDE0017	使用对象初始值设定项	dotnet_style_object_initializer
IDE0018	内联变量声明	csharp_style_inlined_variable_declaration
IDE0019	使用模式匹配来避免 `as` 后跟 `null` 检查	csharp_style_pattern_matching_over_as_with_null_check
IDE0020	使用模式匹配来避免后跟强制转换的 `is` 检查(带变量)	csharp_style_pattern_matching_over_is_with_cast_check
IDE0021	使用构造函数的表达式主体	csharp_style_expression_bodied_constructors
IDE0022	使用方法的表达式主体	csharp_style_expression_bodied_methods
IDE0023	使用转换运算符的表达式主体	csharp_style_expression_bodied_operators
IDE0024	使用运算符的表达式主体	csharp_style_expression_bodied_operators
IDE0025	使用属性的表达式主体	csharp_style_expression_bodied_properties
IDE0026	使用索引器的表达式主体	csharp_style_expression_bodied_indexers
IDE0027	为访问器使用表达式主体	csharp_style_expression_bodied_accessors
IDE0028	使用集合初始值设定项	dotnet_style_collection_initializer
IDE0029	使用 coalesce 表达式(不可为 null 的类型)	dotnet_style_coalesce_expression

ID		
IDE0030	使用 coalesce 表达式(可以为 null 的类型)	dotnet_style_coalesce_expression
IDE0031	使用 Null 传播	dotnet_style_null_propagation
IDE0032	使用自动属性	dotnet_style_prefer_auto_properties
IDE0033	使用显式提供的元组名称	dotnet_style_explicit_tuple_names
IDE0034	简化 `default` 表达式	csharp_prefer_simple_default_expression
IDE0035	删除无法访问的代码	
IDE0036	对修饰符进行排序	csharp_preferred_modifier_order visual_basic_preferred_modifier_order
IDE0037	使用推断的成员名称	dotnet_style_prefer_inferred_tuple_names dotnet_style_prefer_inferred_anonymous_type_member_names
IDE0038	使用模式匹配来避免后跟强制转换的 `is` 检查(不带变量)	csharp_style_pattern_matching_over_is_with_cast_check
IDE0039	使用本地函数而不是 Lambda	csharp_style_pattern_local_over_anonymous_function
IDE0040	添加可访问性修饰符	dotnet_style_require_accessibility_modifiers
IDE0041	使用 is null 检查	dotnet_style_prefer_is_null_check_over_reference_equality_method
IDE0042	析构变量声明	csharp_style_deconstructed_variable_declaration
IDE0044	添加 readonly 修饰符	dotnet_style_readonly_field
IDE0045	使用条件表达式进行赋值	dotnet_style_prefer_conditional_expression_over_assignment
IDE0046	使用 return 的条件表达式	dotnet_style_prefer_conditional_expression_over_return
IDE0047	删除不必要的括号	dotnet_style_parentheses_in_arithmetic_binary_operators dotnet_style_parentheses_in_relational_binary_operators dotnet_style_parentheses_in_other_binary_operators dotnet_style_parentheses_in_other_operators

ID		
IDE0048	为清楚起见，请添加括号	dotnet_style_parentheses_in_arithmetic_binary_operators dotnet_style_parentheses_in_relational_binary_operators dotnet_style_parentheses_in_other_binary_operators dotnet_style_parentheses_in_other_operators
IDE0049	使用语言关键字，而非类型引用的框架类型名称	dotnet_style_predefined_type_for_locals_parameters_members dotnet_style_predefined_type_for_member_access
IDE0050	将匿名类型转换为元组	
IDE0051	删除未使用的私有成员	
IDE0052	删除未读取的私有成员	
IDE0053	使用 Lambda 的表达式主体	csharp_style_expression_bodied_lambdas
IDE0054	使用复合分配	dotnet_style_prefer_compound_assignment
IDE0055	修正格式	
IDE0056	使用索引运算符	csharp_style_prefer_index_operator
IDE0057	使用范围运算符	csharp_style_prefer_range_operator
IDE0058	删除未使用的表达式值	csharp_style_unused_value_expression_statement_preference visual_basic_style_unused_value_expression_statement_preference
IDE0059	删除不必要的赋值	csharp_style_unused_value_assignment_preference visual_basic_style_unused_value_assignment_preference
IDE0060	删除未使用的参数	dotnet_code_quality_unused_parameters
IDE0061	使用局部函数的表达式主体	csharp_style_expression_bodied_local_functions
IDE0062	将本地函数设置为静态	csharp_prefer_static_local_function
IDE0063	使用简单的 `using` 语句	csharp_prefer_simple_using_statement
IDE0064	将结构字段设置为可写	

ID		
IDE0065	`using` 指令放置	csharp_using_directive_placement
IDE0066	使用 switch 表达式	csharp_style_prefer_switch_expression
IDE0070	使用 System.HashCode.Combine	
IDE0071	简化内插	dotnet_style_prefer_simplified_interpolation
IDE0072	将缺失的事例添加到 switch 表达式	
IDE0073	使用文件头	file_header_template
IDE0074	使用联合复合赋值	dotnet_style_prefer_compound_assignment
IDE0075	简化条件表达式	dotnet_style_prefer_simplified_boolean_expressions
IDE0076	删除无效的全局 `SuppressMessageAttribute`	
IDE0077	避免在全局 `SuppressMessageAttribute` 中使用旧格式目标	
IDE0078	使用模式匹配	csharp_style_prefer_pattern_matching
IDE0079	删除不必要的抑制	dotnet_remove_unnecessary_suppression_exclusions
IDE0080	删除不必要的抑制运算符	
IDE0081	删除了 `ByVal`	
IDE0082	将 `typeof` 转换为 `nameof`	
IDE0083	使用模式匹配（`not` 运算符）	csharp_style_prefer_not_pattern
IDE0084	使用模式匹配（`IsNot` 运算符）	visual_basic_style_prefer_isnot_expression
IDE0090	简化 `new` 表达式	csharp_style_implicit_object_creation_when_type_is_apparent
IDE0100	删除不必要的相等运算符	
IDE0110	删除不必要的弃元	
IDE0140	简化对象的创建	visual_basic_style_prefer_simplified_object_creation

ID		
IDE1005	使用条件委托调用	csharp_style_conditional_delegate_call
IDE1006	命名样式	

图例

下表显示了为参考文档中每个规则提供的信息类型。

ID	规则的唯一标识符。规则 ID 用于在代码文件中配置规则严重性和取消显示警告。
	规则的标题。
	规则的类别。
Subcategory	规则的子类别，例如语言规则、格式设置规则或命名规则。
	适用的 .NET（C# 或 Visual Basic），以及最低语言版本（如果适用）。
	首次引入规则时 .NET SDK 或 Visual Studio 的版本。

对于规则的每个选项，都提供了以下信息。

	规则的代码样式选项名称（如果有）。用于自定义样式的选项在 EditorConfig 文件中指定。
	规则选项的代码样式选项值（如果有）。
	规则选项的默认代码样式选项值（如果有）。
	对应于选项的代码样式的示例。

请参阅

- 在生成时强制执行代码样式
- Visual Studio 中的快速操作
- 在 Visual Studio 中创建可移植的自定义编辑器选项

语言规则

2021/11/16 •

代码样式语言规则会影响 .NET 编程语言的各种构造（如修饰符和括号）的使用方式。规则分为以下几类：

- .NET 样式规则：适用于 C# 和 Visual Basic 的规则。这些规则的 EditorConfig 选项名称以 `dotnet_style_` 前缀开头。
- C# 样式规则：仅适用于 C# 语言的规则。这些规则的 EditorConfig 选项名称以 `csharp_style_` 前缀开头。
- Visual Basic 样式规则：仅适用于 Visual Bsic 语言的规则。这些规则的 EditorConfig 选项名称以 `visual_basic_style_` 前缀开头。

选项格式

可以在 EditorConfig 文件中指定语言规则的选项，格式如下：

`option_name = value`（Visual Studio 2019 版本 16.9 预览版 2 及更高版本）

或

`option_name = value:severity`

- 值

  对于每个语言规则，可指定一个定义是否或何时以此样式为首选项的值。许多规则都接受 `true` 值（以此样式为首选项）或 `false` 值（不以此样式为首选项）。其他规则接受 `when_on_single_line` 或 `never` 等值。

- 严重性（在 Visual Studio 2019 版本 16.9 预览版 2 及更高版本中可选）

  此规则的第二部分指定规则的严重性级别。以这种方式指定时，只有 Visual Studio 之类的开发 IDE 会遵循严重性设置。在生成过程中不会遵循这一点。

  若要在生成时强制实施代码样式规则，请改为使用分析器的基于规则 ID 的严重性配置语法来设置严重性。语法采用形式 `dotnet_diagnostic.<rule ID>.severity = <severity>`（例如，`dotnet_diagnostic.IDE0040.severity = silent`）。有关详细信息，请参阅严重性级别。

> **TIP**
> 自 Visual Studio 2019 版本 16.3 起，在样式冲突发生之后，可以在快速操作灯泡菜单中配置代码样式规则。有关详细信息，请参阅在 Visual Studio 中自动配置代码样式。

.NET 样式规则

本节中的样式规则均适用于 C# 和 Visual Basic。

- "this."和"Me."限定符
  - dotnet_style_qualification_for_field
  - dotnet_style_qualification_for_property
  - dotnet_style_qualification_for_method
  - dotnet_style_qualification_for_event
- 语言关键字，而非类型引用的框架类型名称

- dotnet_style_predefined_type_for_locals_parameters_members
- dotnet_style_predefined_type_for_member_access

- 修饰符首选项
  - dotnet_style_require_accessibility_modifiers
  - csharp_preferred_modifier_order
  - visual_basic_preferred_modifier_order
  - dotnet_style_readonly_field
- 括号首选项
  - dotnet_style_parentheses_in_arithmetic_binary_operators
  - dotnet_style_parentheses_in_relational_binary_operators
  - dotnet_style_parentheses_in_other_binary_operators
  - dotnet_style_parentheses_in_other_operators
- 表达式级首选项
  - dotnet_style_object_initializer
  - dotnet_style_collection_initializer
  - dotnet_style_explicit_tuple_names
  - dotnet_style_prefer_inferred_tuple_names
  - dotnet_style_prefer_inferred_anonymous_type_member_names
  - dotnet_style_prefer_auto_properties
  - dotnet_style_prefer_conditional_expression_over_assignment
  - dotnet_style_prefer_conditional_expression_over_return
  - dotnet_style_prefer_compound_assignment
  - dotnet_style_prefer_simplified_interpolation
  - dotnet_style_prefer_simplified_boolean_expressions
  - 向 switch 语句添加缺失的事例 - 此规则没有代码样式选项。
  - 将匿名类型转换为元组 - 此规则没有代码样式选项。
  - 使用“System.HashCode.Combine”- 此规则没有代码样式选项。
  - 将“typeof”转换为“nameof”- 此规则没有代码样式选项。
- Null 检查首选项
  - dotnet_style_coalesce_expression
  - dotnet_style_null_propagation
  - dotnet_style_prefer_is_null_check_over_reference_equality_method
- 文件头首选项
  - file_header_template

C# 样式规则

本节中的样式规则仅适用于 C# 语言。

- “var”首选项
  - csharp_style_var_for_built_in_types
  - csharp_style_var_when_type_is_apparent
  - csharp_style_var_elsewhere
- Expression-Bodied 成员
  - csharp_style_expression_bodied_methods
  - csharp_style_expression_bodied_constructors
  - csharp_style_expression_bodied_operators
  - csharp_style_expression_bodied_properties

    - csharp_style_expression_bodied_indexers
    - csharp_style_expression_bodied_accessors
    - csharp_style_expression_bodied_lambdas
    - csharp_style_expression_bodied_local_functions
- 模式匹配首选项
    - csharp_style_pattern_matching_over_is_with_cast_check
    - csharp_style_pattern_matching_over_as_with_null_check
    - csharp_style_prefer_switch_expression
    - csharp_style_prefer_pattern_matching
    - csharp_style_prefer_not_pattern
- 表达式级首选项
    - csharp_style_inlined_variable_declaration
    - csharp_prefer_simple_default_expression
    - csharp_style_pattern_local_over_anonymous_function
    - csharp_style_deconstructed_variable_declaration
    - csharp_style_prefer_index_operator
    - csharp_style_prefer_range_operator
    - csharp_style_implicit_object_creation_when_type_is_apparent
    - 向 switch 表达式添加缺失的事例 - 此规则没有代码样式选项。
- “NULL”检查首选项
    - csharp_style_throw_expression
    - csharp_style_conditional_delegate_call
- 代码块首选项
    - csharp_prefer_braces
    - csharp_prefer_simple_using_statement
- “using”指令首选项
    - csharp_using_directive_placement
- 修饰符首选项
    - csharp_prefer_static_local_function
    - 使结构字段可写 - 此规则没有代码样式选项。

Visual Basic 样式规则

本节中的样式规则仅适用于 Visual Basic 语言。

- 模式匹配首选项
    - visual_basic_style_prefer_isnot_expression

请参阅

- 不必要的代码规则
- 格式设置规则
- 命名规则
- .NET 代码样式规则参考

this 和 Me 首选项（IDE0003 和 IDE0009）

2021/11/16 •

■ ID	IDE0003 和 IDE0009
■	IDE0003：删除 `this` 或 `Me` 限定 IDE0009：添加 `this` 或 `Me` 限定
■	Style
Subcategory	语言规则
▬	C# 和 Visual Basic

概述

这些样式规则可应用于字段、属性、方法或事件。选项值为“true”表示代码符号以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。选项值为“false”表示码位元素不以 `this.` 或 `Me.` 开头为首选项。

dotnet_style_qualification_for_field

▬	dotnet_style_qualification_for_field
■	`true` - 字段以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项 `false` - 字段不以 `this.` 或 `Me.` 开头为首选项
▬	`false`

示例

```
// dotnet_style_qualification_for_field = true
this.capacity = 0;

// dotnet_style_qualification_for_field = false
capacity = 0;
```

```
' dotnet_style_qualification_for_field = true
Me.capacity = 0

' dotnet_style_qualification_for_field = false
capacity = 0
```

dotnet_style_qualification_for_property

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_property
[illegible]	`true` - 属性以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项 `false` - 属性不以 `this.` 或 `Me.` 开头为首选项
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_property = true
this.ID = 0;

// dotnet_style_qualification_for_property = false
ID = 0;
```

```
' dotnet_style_qualification_for_property = true
Me.ID = 0

' dotnet_style_qualification_for_property = false
ID = 0
```

dotnet_style_qualification_for_method

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_method
[illegible]	`true` - 方法以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。 `false` - 方法不以 `this.` 或 `Me.` 开头为首选项。
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_method = true
this.Display();

// dotnet_style_qualification_for_method = false
Display();
```

```
' dotnet_style_qualification_for_method = true
Me.Display()

' dotnet_style_qualification_for_method = false
Display()
```

dotnet_style_qualification_for_event

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_event
[illegible]	`true` - 事件以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。 `false` - 事件不以 `this.` 或 `Me.` 开头为首选项。
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_event = true
this.Elapsed += Handler;

// dotnet_style_qualification_for_event = false
Elapsed += Handler;
```

```
' dotnet_style_qualification_for_event = true
AddHandler Me.Elapsed, AddressOf Handler

' dotnet_style_qualification_for_event = false
AddHandler Elapsed, AddressOf Handler
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

this 和 Me 首选项（IDE0003 和 IDE0009）

2021/11/16 •

■ ID	IDE0003 和 IDE0009
■	IDE0003：删除 `this` 或 `Me` 限定 IDE0009：添加 `this` 或 `Me` 限定
■	Style
Subcategory	语言规则
▬	C# 和 Visual Basic

概述

这些样式规则可应用于字段、属性、方法或事件。选项值为“true”表示代码符号以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。选项值为“false”表示码位元素不以 `this.` 或 `Me.` 开头为首选项。

dotnet_style_qualification_for_field

▬	dotnet_style_qualification_for_field
■	`true` - 字段以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项 `false` - 字段不以 `this.` 或 `Me.` 开头为首选项
▬	`false`

示例

```
// dotnet_style_qualification_for_field = true
this.capacity = 0;

// dotnet_style_qualification_for_field = false
capacity = 0;
```

```
' dotnet_style_qualification_for_field = true
Me.capacity = 0

' dotnet_style_qualification_for_field = false
capacity = 0
```

dotnet_style_qualification_for_property

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_property
[illegible]	`true` - 属性以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项 `false` - 属性不以 `this.` 或 `Me.` 开头为首选项
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_property = true
this.ID = 0;

// dotnet_style_qualification_for_property = false
ID = 0;
```

```
' dotnet_style_qualification_for_property = true
Me.ID = 0

' dotnet_style_qualification_for_property = false
ID = 0
```

dotnet_style_qualification_for_method

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_method
[illegible]	`true` - 方法以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。 `false` - 方法不以 `this.` 或 `Me.` 开头为首选项。
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_method = true
this.Display();

// dotnet_style_qualification_for_method = false
Display();
```

```
' dotnet_style_qualification_for_method = true
Me.Display()

' dotnet_style_qualification_for_method = false
Display()
```

dotnet_style_qualification_for_event

[illegible]	[illegible]
[illegible]	dotnet_style_qualification_for_event
[illegible]	`true` - 事件以 `this.` (C#) 或 `Me.` (Visual Basic) 开头为首选项。 `false` - 事件不以 `this.` 或 `Me.` 开头为首选项。
[illegible]	`false`

示例

```
// dotnet_style_qualification_for_event = true
this.Elapsed += Handler;

// dotnet_style_qualification_for_event = false
Elapsed += Handler;
```

```
' dotnet_style_qualification_for_event = true
AddHandler Me.Elapsed, AddressOf Handler

' dotnet_style_qualification_for_event = false
AddHandler Elapsed, AddressOf Handler
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

使用语言关键字，而非类型引用的框架类型名称 (IDE0049)

2021/11/16 •

■ ID	IDE0049
■	使用语言关键字，而非类型引用的框架类型名称
■	Style
Subcategory	语言规则
■■	C# 和 Visual Basic

概述

此样式规则可应用到局部变量、方法参数和类成员，也可作为单独针对类型成员访问表达式的规则。值为“true”表示，如果类型中有用于表示类型的关键字，首选语言关键字（例如 `int` 或 `Integer`），而不是类型名称（例如 `Int32`）。值为“false”代表类型名称为首选项，而非语言关键字。

dotnet_style_predefined_type_for_locals_parameters_members

■■	dotnet_style_predefined_type_for_locals_parameters_members
■	`true` - 对于类型（其中具有用于表示该类型的关键字），本地变量、方法参数和类成员的语言关键字为首选项，而非类型名称 `false` - 本地变量、方法参数和类成员的类型名称为首选项，而非语言关键字
■■	`true`

示例

```
// dotnet_style_predefined_type_for_locals_parameters_members = true
private int _member;

// dotnet_style_predefined_type_for_locals_parameters_members = false
private Int32 _member;
```

```
' dotnet_style_predefined_type_for_locals_parameters_members = true
Private _member As Integer

' dotnet_style_predefined_type_for_locals_parameters_members = false
Private _member As Int32
```

dotnet_style_predefined_type_for_member_access

[illegible]	[illegible]
[illegible]	dotnet_style_predefined_type_for_member_access
[illegible]	`true` - 对于类型(其中具有用于表示该类型的关键字)，成员访问表达式的语言关键字为首选项，而非类型名称 `false` - 成员访问表达式的类型名称为首选项，而非语言关键字
[illegible]	`true`

示例

```
// dotnet_style_predefined_type_for_member_access = true
var local = int.MaxValue;

// dotnet_style_predefined_type_for_member_access = false
var local = Int32.MaxValue;
```

```
' dotnet_style_predefined_type_for_member_access = true
Dim local = Integer.MaxValue

' dotnet_style_predefined_type_for_member_access = false
Dim local = Int32.MaxValue
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

修饰符首选项

2021/11/16 •

.NET 修饰符首选项

本节中的样式规则涉及以下修饰符首选项首选项，这些首选项在 C# 和 Visual Basic 中很常见：

- 对修饰符进行排序 (IDE0036)
- 添加可访问性修饰符 (IDE0040)
- 添加 readonly 修饰符 (IDE0044)

C# 修饰符首选项

本节中的样式规则涉及以下特定于 C# 的修饰符首选项：

- 将本地函数设置为静态 (IDE0062)
- 将结构字段设置为可写 (IDE0064)

另请参阅

- 代码样式规则参考
- 代码样式语言规则

对修饰符进行排序 (IDE0036)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0036
■	对修饰符进行排序
■	Style
Subcategory	语言规则(修饰符首选项)
▬	C# 和 Visual Basic
▬	Visual Studio 2017 版本 15.5

概述

本部分中的样式规则与指定所需修饰符排序顺序有关。

- 如果对一系列修饰符设置该规则，则首选指定的排序。
- 如果文件中省略了此规则，则不优先使用修饰符顺序。

csharp_preferred_modifier_order

PROPERTY	[illegible]
▬	csharp_preferred_modifier_order
▬	C#
▬	一个或多个 C# 修饰符，如 `public`、`private` 和 `protected`
▬	`public, private, protected, internal, static, extern, new, virtual, abstract, sealed, override, readonly, unsafe, volatile, async:silent`

示例

```
// csharp_preferred_modifier_order =
public,private,protected,internal,static,extern,new,virtual,abstract,sealed,override,readonly,unsafe,volatil
e,async
class MyClass
{
    private static readonly int _daysInYear = 365;
}
```

visual_basic_preferred_modifier_order

PROPERTY	[
▬	visual_basic_preferred_modifier_order
▬▬	Visual Basic
▬	一个或多个 Visual Basic 修饰符，如 `Partial`、`Private` 和 `Public`
▬▬	`Partial, Default, Private, Protected, Public, Friend, NotOverridable, Overridable, MustOverride, Overloads, Overrides, MustInherit, NotInheritable, Static, Shared, Shadows, ReadOnly, WriteOnly, Dim, Const, WithEvents, Widening, Narrowing, Custom, Async:silent`

示例

```
' visual_basic_preferred_modifier_order =
Partial,Default,Private,Protected,Public,Friend,NotOverridable,Overridable,MustOverride,Overloads,Overrides,
MustInherit,NotInheritable,Static,Shared,Shadows,ReadOnly,WriteOnly,Dim,Const,WithEvents,Widening,Narrowing,
Custom,Async
Public Class MyClass
    Private Shared ReadOnly daysInYear As Int = 365
End Class
```

另请参阅

- 修饰符首选项
- 代码样式语言规则
- 代码样式规则参考

添加可访问性修饰符 (IDE0040)

2021/11/16 •

■ ID	IDE0040
■	添加可访问性修饰符
■	Style
Subcategory	语言规则(修饰符首选项)
▬	C# 和 Visual Basic
▬	Visual Studio 2017 版本 15.5

概述

此样式规则涉及在声明中要求可访问性修饰符。选项值指定所需可访问性修饰符的首选项。

dotnet_style_require_accessibility_modifiers

▬	dotnet_style_require_accessibility_modifiers
■	`always` - 优先指定可访问性修饰符。 `for_non_interface_members` - 优先声明可访问性修饰符，公共接口成员除外。 `never` - 不优先指定可访问性修饰符。 `omit_if_default` - 优先指定可访问性修饰符(除非它们是默认修饰符)。
▬	`for_non_interface_members`

示例

```
// dotnet_style_require_accessibility_modifiers = always
// dotnet_style_require_accessibility_modifiers = for_non_interface_members
class MyClass
{
    private const string thisFieldIsConst = "constant";
}

// dotnet_style_require_accessibility_modifiers = never
class MyClass
{
    const string thisFieldIsConst = "constant";
}
```

另请参阅

- 修饰符首选项
- 代码样式语言规则
- 代码样式规则参考

添加 readonly 修饰符 (IDE0044)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0044
■	添加 readonly 修饰符
■	Style
Subcategory	语言规则(修饰符首选项)
▬	C# 和 Visual Basic
▬	Visual Studio 2017 15.7 版

概述

此样式规则涉及为已初始化(以内联方式或在构造函数内部)但从未重新赋值的字段指定 readonly 修饰符。

dotnet_style_readonly_field

PROPERTY	[illegible]
▬	dotnet_style_readonly_field
■	`true` - 如果字段只是内联分配或者在构造函数中，偏向将它们标记为 `readonly` (C#) 或 `ReadOnly` (Visual Basic) 的字段 `false` - 就字段是否应标记为 `readonly` (C#) 或 `ReadOnly` (Visual Basic) 无偏向
▬	`true`

示例

```
// dotnet_style_readonly_field = true
class MyClass
{
    private readonly int _daysInYear = 365;
}
```

```
' dotnet_style_readonly_field = true
Public Class MyClass
    Private ReadOnly daysInYear As Int = 365
End Class
```

另请参阅

- 修饰符首选项
- 代码样式语言规则
- 代码样式规则参考

将本地函数设置为静态 (IDE0062)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0062
■	将本地函数设置为静态
■	Style
Subcategory	语言规则(修饰符首选项)
▬	C# 8.0+

概述

此样式规则涉及优先将本地函数 设置为 `static`(如可能)。

csharp_prefer_static_local_function

[illegible]	[illegible]
▬	csharp_prefer_static_local_function
■	`true` - 首选将局部函数标记为 `static` `false` - 不推荐将局部函数标记为 `static`
▬	`true:suggestion`

示例

```
// csharp_prefer_static_local_function = true
void M()
{
    Hello();
    static void Hello()
    {
        Console.WriteLine("Hello");
    }
}

// csharp_prefer_static_local_function = false
void M()
{
    Hello();
    void Hello()
    {
        Console.WriteLine("Hello");
    }
}
```

另请参阅

- 修饰符首选项
- 代码样式语言规则
- 代码样式规则参考

将结构字段设置为可写 (IDE0064)

2021/11/16 •

■ ID	IDE0064
■	将结构字段设置为可写
■	CodeQuality
Subcategory	语言规则(修饰符首选项)
■	C#

概述

此规则检测包含一个或多个 `readonly` 字段的结构，还包含在构造函数之外对 `this` 的赋值。此规则建议将 `readonly` 字段转换为非 readonly，即可写。将此类结构字段标记为 readonly 可能会导致意外的行为，因为在构造函数之外对 `this` 赋值时，分配给该字段的值可能会发生更改。

示例

```
// Code with violations
struct MyStruct
{
    public readonly int Value;

    public MyStruct(int value)
    {
        Value = value;
    }

    public void Test()
    {
        this = new MyStruct(5);
    }
}

// Fixed code
struct MyStruct
{
    public int Value;

    public MyStruct(int value)
    {
        Value = value;
    }

    public void Test()
    {
        this = new MyStruct(5);
    }
}
```

另请参阅

- 修饰符首选项
- 代码样式语言规则
- 代码样式规则参考

括号首选项（IDE0047 和 IDE0048）

2021/11/16 •

■ ID	IDE0047 和 IDE0048
■	IDE0047：删除不必要的括号 IDE0048：添加括号确保清晰
■	Style
Subcategory	语言规则
■	C# 和 Visual Basic
■	Visual Studio 2017 版本 15.8

概述

本部分中的样式规则涉及括号首选项，包括算术、关系和其他二元运算符的括号使用情况。

dotnet_style_parentheses_in_arithmetic_binary_operators

■	dotnet_style_parentheses_in_arithmetic_binary_operators
■	`always_for_clarity` - 优先使用括号来声明算术运算符（`*`、`/`、`%`、`+`、`-`、`<<`、`>>`、`&`、`^`、`|`）优先级 `never_if_unnecessary` - 算术运算符（`*`、`/`、`%`、`+`、`-`、`<<`、`>>`、`&`、`^`、`|`）的优先级显而易见时，最好不要使用括号
■	`always_for_clarity`

示例

```
// dotnet_style_parentheses_in_arithmetic_binary_operators = always_for_clarity
var v = a + (b * c);

// dotnet_style_parentheses_in_arithmetic_binary_operators = never_if_unnecessary
var v = a + b * c;
```

```
' dotnet_style_parentheses_in_arithmetic_binary_operators = always_for_clarity
Dim v = a + (b * c)

' dotnet_style_parentheses_in_arithmetic_binary_operators = never_if_unnecessary
Dim v = a + b * c
```

dotnet_style_parentheses_in_relational_binary_operators

𝚝𝚝	𝚝
▬	dotnet_style_parentheses_in_relational_binary_operators
▬	`always_for_clarity` - 优先使用括号来声明关系运算符(`>`、`<`、`<=`、`>=`、`is`、`as`、`==`、`!=`)优先级 `never_if_unnecessary` - 关系运算符(`>`、`<`、`<=`、`>=`、`is`、`as`、`==`、`!=`)的优先级显而易见时，最好不要使用括号
▬	`always_for_clarity`

示例

```
// dotnet_style_parentheses_in_relational_binary_operators = always_for_clarity
var v = (a < b) == (c > d);

// dotnet_style_parentheses_in_relational_binary_operators = never_if_unnecessary
var v = a < b == c > d;
```

```
' dotnet_style_parentheses_in_relational_binary_operators = always_for_clarity
Dim v = (a < b) = (c > d)

' dotnet_style_parentheses_in_relational_binary_operators = never_if_unnecessary
Dim v = a < b = c > d
```

dotnet_style_parentheses_in_other_binary_operators

𝚝𝚝	𝚝
▬	dotnet_style_parentheses_in_other_binary_operators
▬	`always_for_clarity` - 优先使用括号来声明其他二元运算符(`&&`、`||`、`??`)优先级 `never_if_unnecessary` - 其他二元运算符(`&&`、`||`、`??`)的优先级显而易见时，最好不要使用括号
▬	`always_for_clarity`

示例

```
// dotnet_style_parentheses_in_other_binary_operators = always_for_clarity
var v = a || (b && c);

// dotnet_style_parentheses_in_other_binary_operators = never_if_unnecessary
var v = a || b && c;
```

```
' dotnet_style_parentheses_in_other_binary_operators = always_for_clarity
Dim v = a OrElse (b AndAlso c)

' dotnet_style_parentheses_in_other_binary_operators = never_if_unnecessary
Dim v = a OrElse b AndAlso c
```

dotnet_style_parentheses_in_other_operators

[illegible]	[illegible]
[illegible]	dotnet_style_parentheses_in_other_operators
[illegible]	`always_for_clarity` - 优先使用括号来声明运算符先级 `never_if_unnecessary` - 运算符的优先级显而易见时，最好不要使用括号
[illegible]	`never_if_unnecessary`

示例

```
// dotnet_style_parentheses_in_other_operators = always_for_clarity
var v = (a.b).Length;

// dotnet_style_parentheses_in_other_operators = never_if_unnecessary
var v = a.b.Length;
```

```
' dotnet_style_parentheses_in_other_operators = always_for_clarity
Dim v = (a.b).Length

' dotnet_style_parentheses_in_other_operators = never_if_unnecessary
Dim v = a.b.Length
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

表达式级首选项

2021/11/16 •

.NET 表达式级首选项

本节中的样式规则涉及以下表达式级首选项，这些首选项在 C# 和 Visual Basic 中很常见：

- 将缺失的事例添加到 switch 语句 (IDE0010)
- 使用对象初始值设定项 (IDE0017)
- 使用集合初始值设定项 (IDE0028)
- 使用自动属性 (IDE0032)
- 使用显式提供的元组名称 (IDE0033)
- 使用推断的成员名称 (IDE0037)
- 使用 assignment 的条件表达式 (IDE0045)
- 使用 return 的条件表达式 (IDE0046)
- 将匿名类型转换为元组 (IDE0050)
- 使用复合赋值（IDE0054 和 IDE0074）
- 使用 System.HashCode.Combine (IDE0070)
- 简化内插 (IDE0071)
- 简化条件表达式 (IDE0075)
- 将 typeof 转换为 nameof (IDE0082)

C# 表达式级首选项

本节中的样式规则涉及以下特定于 C# 的表达式级首选项：

- 内联变量声明 (IDE0018)
- 简化 default 表达式 (IDE0034)
- 使用本地函数而不是 Lambda (IDE0039)
- 析构变量声明 (IDE0042)
- 使用索引运算符 (IDE0056)
- 使用范围运算符 (IDE0057)
- 将缺失的事例添加到 switch 表达式 (IDE0072)
- 简化 new 表达式 (IDE0090)

另请参阅

- 代码样式规则参考
- 代码样式语言规则

将缺失的事例添加到 switch 语句 (IDE0010)

2021/11/16 •

■ ID	IDE0010
■	将缺失的事例添加到 switch 语句
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic

概述

此规则涉及为 `switch` 语句指定所有缺失的 switch 事例。在以下情况下，`switch` 语句被视为不完整，有缺失事例：

- 缺失一个或多个枚举成员的事例的 enum `switch` 语句。
- 缺少 `default` 事例的 `switch` 语句。

此规则没有关联的代码样式选项。

示例

```
enum E
{
    A,
    B
}

class C
{
    // Code with violations
    int M(E e)
    {
        // IDE0010: Add missing cases
        switch (e)
        {
            case E.A:
                return 0;
        }

        return -1;
    }

    // Fixed code
    int M(E e)
    {
        switch (e)
        {
            case E.A:
                return 0;
            case E.B:
                return 1;
            default:
                return -1;
        }
    }
}
```

另请参阅

- Switch 语句
- 将缺失的事例添加到 switch 表达式 (IDE0072)
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用对象初始值设定项 (IDE0017)

2021/11/16 •

■ ID	IDE0017
■	使用对象初始值设定项
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 和 Visual Basic

概述

此样式规则涉及使用对象初始值设定项进行对象初始化。选项值指定是否需要初始值设定项。

dotnet_style_object_initializer

■	dotnet_style_object_initializer
■	`true` - 在可能情况下，更倾向使用对象初始值设定项来初始化对象 `false` - 更倾向不使用对象初始值设定项来初始化对象
■	`true`

示例

```
// dotnet_style_object_initializer = true
var c = new Customer() { Age = 21 };

// dotnet_style_object_initializer = false
var c = new Customer();
c.Age = 21;
```

```
' dotnet_style_object_initializer = true
Dim c = New Customer() With {.Age = 21}

' dotnet_style_object_initializer = false
Dim c = New Customer()
c.Age = 21
```

另请参阅

- 使用集合初始值设定项
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

内联变量声明 (IDE0018)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0018
■	内联变量声明
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 7.0+

概述

此样式规则与 `out` 变量是否声明为内联有关。从 C# 7 开始，可以在方法调用的实际参数列表中声明 out 变量，而不是在单独的变量声明中。

csharp_style_inlined_variable_declaration

[illegible]	[illegible]
▬	csharp_style_inlined_variable_declaration
■	`true` - `out` 变量在方法调用的参数列表中声明为内联为首选项(如可能) `false` - 在方法调用之前声明 `out` 变量为首选项
▬	`true`

示例

```
// csharp_style_inlined_variable_declaration = true
if (int.TryParse(value, out int i) {...}

// csharp_style_inlined_variable_declaration = false
int i;
if (int.TryParse(value, out i) {...}
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用集合初始值设定项 (IDE0028)

2021/11/16 •

■ ID	IDE0028
■	使用集合初始值设定项
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 和 Visual Basic

概述

此样式规则涉及使用集合初始化的集合初始值设定项。选项值指定是否需要初始值设定项。

dotnet_style_collection_initializer

■	dotnet_style_collection_initializer
■	`true` - 在可能情况下，更倾向使用集合初始值设定项来初始化集合 `false` - 不使用集合初始值设定项来初始化集合为首选项
■	`true`

示例

```
// dotnet_style_collection_initializer = true
var list = new List<int> { 1, 2, 3 };

// dotnet_style_collection_initializer = false
var list = new List<int>();
list.Add(1);
list.Add(2);
list.Add(3);
```

```
' dotnet_style_collection_initializer = true
Dim list = New List(Of Integer) From {1, 2, 3}

' dotnet_style_collection_initializer = false
Dim list = New List(Of Integer)
list.Add(1)
list.Add(2)
list.Add(3)
```

另请参阅

- 使用对象初始值设定项
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用自动属性 (IDE0032)

2021/11/16 •

■ ID	IDE0032
■	使用自动属性
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 和 Visual Basic
■	Visual Studio 2017 15.7 版

概述

此样式规则涉及使用自动属性，而不是专用支持字段的属性。

dotnet_style_prefer_auto_properties

■	dotnet_style_prefer_auto_properties
■	`true` - 首选自动属性，而不是专用支持字段的属性 `false` - 首选专用支持字段的属性，而不是自动属性
■	`true`

示例

```
// dotnet_style_prefer_auto_properties = true
private int Age { get; }

// dotnet_style_prefer_auto_properties = false
private int age;

public int Age
{
    get
    {
        return age;
    }
}
```

```
' dotnet_style_prefer_auto_properties = true
Public ReadOnly Property Age As Integer

' dotnet_style_prefer_auto_properties = false
Private _age As Integer

Public ReadOnly Property Age As Integer
    Get
        return _age
    End Get
End Property
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用显式提供的元组名称 (IDE0033)

2021/11/16 •

■ ID	IDE0033
■	使用显式提供的元组名称
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 7.0+ 和 Visual Basic 15+

概述

此样式规则涉及使用显式元组名称，而不是隐式“ItemX”属性。

dotnet_style_explicit_tuple_names

■	dotnet_style_explicit_tuple_names
■	`true` - 比起 ItemX 属性更倾向元祖名称 `false` - 比起元组名称更倾向 ItemX 属性
■	`true`

示例

```
// dotnet_style_explicit_tuple_names = true
(string name, int age) customer = GetCustomer();
var name = customer.name;

// dotnet_style_explicit_tuple_names = false
(string name, int age) customer = GetCustomer();
var name = customer.Item1;
```

```
 ' dotnet_style_explicit_tuple_names = true
Dim customer As (name As String, age As Integer) = GetCustomer()
Dim name = customer.name

' dotnet_style_explicit_tuple_names = false
Dim customer As (name As String, age As Integer) = GetCustomer()
Dim name = customer.Item1
```

另请参阅

- 使用对象初始值设定项
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

简化 default 表达式 (IDE0034)

2021/11/16 •

■ ID	IDE0034
■	简化 `default` 表达式
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 7.1+

概述

此样式规则涉及在编译器可推断表达式类型的情况下，使用默认值表达式的 default 文本。

csharp_prefer_simple_default_expression

▬	csharp_prefer_simple_default_expression
■	`true` - 首选 `default` 次选 `default(T)` `false` - 首选 `default(T)` 次选 `default`
▬	`true`

示例

```
// csharp_prefer_simple_default_expression = true
void DoWork(CancellationToken cancellationToken = default) { ... }

// csharp_prefer_simple_default_expression = false
void DoWork(CancellationToken cancellationToken = default(CancellationToken)) { ... }
```

另请参阅

- default 文本
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用推断的成员名称 (IDE0037)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0037
■	使用推断的成员名称
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 7.1+ 和 Visual Basic 15+
▬	Visual Studio 2017 版本 15.6

概述

此样式规则涉及以下使用推断名称的代码样式:

- 使用推断的元组元素名称 (`dotnet_style_prefer_inferred_tuple_names`) 以及
- 使用推断的匿名类型成员名称 (`dotnet_style_prefer_inferred_anonymous_type_member_names`)

dotnet_style_prefer_inferred_tuple_names

[illegible]	[illegible]
▬	dotnet_style_prefer_inferred_tuple_names
■	`true` - 首选推断元组元素名称 `false` - 首选显式元组元素名称
▬	`true`

示例

```
// dotnet_style_prefer_inferred_tuple_names = true
var tuple = (age, name);

// dotnet_style_prefer_inferred_tuple_names = false
var tuple = (age: age, name: name);
```

```
' dotnet_style_prefer_inferred_tuple_names = true
Dim tuple = (name, age)

' dotnet_style_prefer_inferred_tuple_names = false
Dim tuple = (name:=name, age:=age)
```

dotnet_style_prefer_inferred_anonymous_type_member_names

[illegible]	[illegible]
[illegible]	dotnet_style_prefer_inferred_anonymous_type_member_names
[illegible]	`true` - 首选推断匿名类型成员名称 `false` - 首选显式匿名类型成员名称
[illegible]	`true`

示例

```
// dotnet_style_prefer_inferred_anonymous_type_member_names = true
var anon = new { age, name };

// dotnet_style_prefer_inferred_anonymous_type_member_names = false
var anon = new { age = age, name = name };
```

```
' dotnet_style_prefer_inferred_anonymous_type_member_names = true
Dim anon = New With {name, age}

' dotnet_style_prefer_inferred_anonymous_type_member_names = false
Dim anon = New With {.name = name, .age = age}
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用本地函数而不是 Lambda (IDE0039)

2021/11/16 •

■ ID	IDE0039
■	使用本地函数而不是 Lambda
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 7.0+

概述

此样式规则涉及使用本地函数，而不是 Lambda(匿名函数)。

csharp_style_pattern_local_over_anonymous_function

■	csharp_style_pattern_local_over_anonymous_function
■	`true` - 首选本地函数，而非匿名函数 `false` - 首选匿名函数，而不是本地函数
■	`true`

示例

```
// csharp_style_pattern_local_over_anonymous_function = true
int fibonacci(int n)
{
    return n <= 1 ? 1 : fibonacci(n-1) + fibonacci(n-2);
}

// csharp_style_pattern_local_over_anonymous_function = false
Func<int, int> fibonacci = null;
fibonacci = (int n) =>
{
    return n <= 1 ? 1 : fibonacci(n - 1) + fibonacci(n - 2);
};
```

另请参阅

- 表达式级首选项
- 代码样式语言规则

- 代码样式规则参考

析构变量声明 (IDE0042)

2021/11/16 •

■ ID	IDE0042
■	析构变量声明
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 7.0+

概述

此样式规则涉及在变量声明中使用析构(如可能)。

csharp_style_deconstructed_variable_declaration

■	csharp_style_deconstructed_variable_declaration
■	`true` - 首选析构变量声明 `false` - 不首选变量声明中的析构
■	`true`

示例

```
// csharp_style_deconstructed_variable_declaration = true
var (name, age) = GetPersonTuple();
Console.WriteLine($"{name} {age}");

(int x, int y) = GetPointTuple();
Console.WriteLine($"{x} {y}");

// csharp_style_deconstructed_variable_declaration = false
var person = GetPersonTuple();
Console.WriteLine($"{person.name} {person.age}");

(int x, int y) point = GetPointTuple();
Console.WriteLine($"{point.x} {point.y}");
```

另请参阅

- 表达式级首选项
- 代码样式语言规则

- 代码样式规则参考

使用条件表达式进行赋值 (IDE0045)

2021/11/16 •

■ ID	IDE0045
■	使用条件表达式进行赋值
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic
▬	Visual Studio 2017 版本 15.8

概述

此样式规则涉及使用三元条件表达式而不是 if-else 语句来进行需要条件逻辑的赋值。

dotnet_style_prefer_conditional_expression_over_assignment

▬	dotnet_style_prefer_conditional_expression_over_assignment
■	`true` - 与 if-else 语句相比，首选三元条件进行赋值 `false` - 与三元条件相比，首选 if-else 语句进行赋值
▬	`true`

示例

```
// dotnet_style_prefer_conditional_expression_over_assignment = true
string s = expr ? "hello" : "world";

// dotnet_style_prefer_conditional_expression_over_assignment = false
string s;
if (expr)
{
    s = "hello";
}
else
{
    s = "world";
}
```

```
' dotnet_style_prefer_conditional_expression_over_assignment = true
Dim s As String = If(expr, "hello", "world")

' dotnet_style_prefer_conditional_expression_over_assignment = false
Dim s As String
If expr Then
    s = "hello"
Else
    s = "world"
End If
```

另请参阅

- 使用条件表达式进行返回
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用 return 的条件表达式 (IDE0046)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0046
■	使用 return 的条件表达式
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic
▬	Visual Studio 2017 版本 15.8

概述

此样式规则涉及对需要条件逻辑的 return 语句使用三元条件表达式，而不是使用 if-else 语句。

dotnet_style_prefer_conditional_expression_over_return

[illegible]	[illegible]
▬	dotnet_style_prefer_conditional_expression_over_return
■	`true` - 与 if-else 语句相比，return 语句首选三元条件 `false` - 与三元条件相比，return 语句首选 if-else 语句
▬	`true`

示例

```
// dotnet_style_prefer_conditional_expression_over_return = true
return expr ? "hello" : "world"

// dotnet_style_prefer_conditional_expression_over_return = false
if (expr)
{
    return "hello";
}
else
{
    return "world";
}
```

```
' dotnet_style_prefer_conditional_expression_over_return = true
Return If(expr, "hello", "world")

' dotnet_style_prefer_conditional_expression_over_return = false
If expr Then
    Return "hello"
Else
    Return "world"
End If
```

另请参阅

- 使用 assignment 的条件表达式
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

将匿名类型转换为元组 (IDE0050)

2021/11/16 •

■ ID	IDE0050
■	将匿名类型转换为元组
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic

概述

当匿名类型有两个或多个字段时，此规则建议使用元组，而不是匿名类型。此规则没有关联的代码样式选项。

示例

```
// Code with violations
var t1 = new { a = 1, b = 2 };

// Fixed code
var t1 = (a: 1, b: 2);
```

```
' Code with violations
Dim t1 = New With { .a = 1, .b = 2 }

' Fixed code
Dim t1 = (a:=1, b:=2)
```

另请参阅

- 元组
- 匿名类型
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用复合赋值（IDE0054 和 IDE0074）

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0054 和 IDE0074
■	IDE0054：使用复合赋值 IDE0074：使用联合复合赋值
■	Style
Subcategory	语言规则（表达式级首选项）
■■	C# 和 Visual Basic

概述

此样式规则涉及使用复合赋值。选项值指定是否需要它们。联合复合赋值报告 `IDE0074`，其他复合赋值报告 `IDE0054`。

dotnet_style_prefer_compound_assignment

[illegible]	[illegible]
■■	dotnet_style_prefer_compound_assignment
■■	`true` - 首选复合赋值表达式 `false` - 不推荐使用复合赋值表达式
■■	`true`

示例

```
// dotnet_style_prefer_compound_assignment = true
x += 1;

// dotnet_style_prefer_compound_assignment = false
x = x + 1;
```

```
' dotnet_style_prefer_compound_assignment = true
x += 1

' dotnet_style_prefer_compound_assignment = false
x = x + 1
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用索引运算符 (IDE0056)

2021/11/16 •

■ ID	IDE0056
■	使用索引运算符
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 8.0+

概述

此样式规则与是否使用 C# 8.0 及更高版本中提供的 index-from-end 运算符有关。

csharp_style_prefer_index_operator

▬	csharp_style_prefer_index_operator
■	`true` - 在从集合末尾计算索引时，首选使用 `^` 操作符 `false` - 在从集合末尾计算索引时，不推荐使用 `^` 操作符
▬	`true`

示例

```
// csharp_style_prefer_index_operator = true
string[] names = { "Archimedes", "Pythagoras", "Euclid" };
var index = names[^1];

// csharp_style_prefer_index_operator = false
string[] names = { "Archimedes", "Pythagoras", "Euclid" };
var index = names[names.Length - 1];
```

另请参阅

- index-from-end 运算符
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用范围运算符 (IDE0057)

2021/11/16 •

■	■
■ ID	IDE0057
■	使用范围运算符
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 8.0+

概述

此样式规则与是否使用 C# 8.0 及更高版本中提供的范围运算符有关。

csharp_style_prefer_range_operator

■	csharp_style_prefer_range_operator
■	`true` - 在提取集合的“切片”时，首选使用范围操作符 `..` `false` - 在提取集合的“切片”时，不推荐使用范围操作符 `..`
■	`true`

示例

```
// csharp_style_prefer_range_operator = true
string sentence = "the quick brown fox";
var sub = sentence[0..^4];

// csharp_style_prefer_range_operator = false
string sentence = "the quick brown fox";
var sub = sentence.Substring(0, sentence.Length - 4);
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用 System.HashCode.Combine (IDE0070)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0070
■	使用 `System.HashCode.Combine`
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic

概述

此规则建议使用 System.HashCode.Combine 方法来计算哈希代码，而不是使用自定义哈希代码计算逻辑。此规则没有关联的代码样式选项。

示例

```
class B
{
    public override int GetHashCode() => 0;
}

class C : B
{
    int j;

    // Code with violations
    public override int GetHashCode()
    {
        // IDE0070: GetHashCode can be simplified.
        var hashCode = 339610899;
        hashCode = hashCode * -1521134295 + base.GetHashCode();
        hashCode = hashCode * -1521134295 + j.GetHashCode();
        return hashCode;
    }

    // Fixed code
    public override int GetHashCode()
    {
        return System.HashCode.Combine(base.GetHashCode(), j);
    }
}
```

另请参阅

- 表达式级首选项
- 代码样式语言规则

- 代码样式规则参考

简化内插 (IDE0071)

2021/11/16 •

■ ID	IDE0071
■	简化内插
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 6.0+ 和 Visual Basic 14+

概述

此样式规则涉及简化内插字符串以提高代码可读性。如果删除了显式方法调用，当编译器隐式调用相同的方法时，它建议删除某些显式方法调用(例如 `ToString()`)。

dotnet_style_prefer_simplified_interpolation

■	dotnet_style_prefer_simplified_interpolation
■	`true` - 首选简化的内插字符串 `false` - 请勿首选简化的内插字符串
■	`true`

示例

```
// dotnet_style_prefer_simplified_interpolation = true
var str = $"prefix {someValue} suffix";

// dotnet_style_prefer_simplified_interpolation = false
var str = $"prefix {someValue.ToString()} suffix";
```

```
 ' dotnet_style_prefer_simplified_interpolation = true
Dim str = $"prefix {someValue} suffix"

' dotnet_style_prefer_simplified_interpolation = false
Dim str = $"prefix {someValue.ToString()} suffix"
```

另请参阅

- 内插字符串

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

将缺失的事例添加到 switch 表达式 (IDE0072)

2021/11/16 •

■ ID	IDE0072
■	将缺失的事例添加到 switch 表达式
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 8.0+

概述

此规则涉及为 switch 表达式指定所有缺失的 switch 事例。在以下情况下，switch 表达式被视为不完整，有缺失事例：

- 缺失一个或多个枚举成员的事例的 enum switch 表达式。
- 缺失 fallthrough 事例 `_` 的 switch 表达式。

此规则没有关联的代码样式选项。

示例

```
enum E
{
    A,
    B
}

class C
{
    // Code with violations
    int M(E e)
    {
        // IDE0010: Add missing cases
        return e switch
        {
            E.A => 0,
            _ => -1,
        };
    }

    // Fixed code
    int M(E e)
    {
        return e switch
        {
            E.A => 0,
            E.B => 1,
            _ => -1,
        };
    }
}
```

另请参阅

- Switch 表达式
- 将缺失的事例添加到 switch 语句 (IDE0010)
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

使用复合赋值（IDE0054 和 IDE0074）

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0054 和 IDE0074
■	IDE0054：使用复合赋值 IDE0074：使用联合复合赋值
■	Style
Subcategory	语言规则（表达式级首选项）
■■	C# 和 Visual Basic

概述

此样式规则涉及使用复合赋值。选项值指定是否需要它们。联合复合赋值报告 `IDE0074`，其他复合赋值报告 `IDE0054`。

dotnet_style_prefer_compound_assignment

[illegible]	[illegible]
■■	dotnet_style_prefer_compound_assignment
■■	`true` - 首选复合赋值表达式 `false` - 不推荐使用复合赋值表达式
■■	`true`

示例

```
// dotnet_style_prefer_compound_assignment = true
x += 1;

// dotnet_style_prefer_compound_assignment = false
x = x + 1;
```

```
' dotnet_style_prefer_compound_assignment = true
x += 1

' dotnet_style_prefer_compound_assignment = false
x = x + 1
```

另请参阅

- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

简化条件表达式 (IDE0075)

2021/11/16 •

■ ID	IDE0075
■	简化条件表达式
■	Style
Subcategory	语言规则(表达式级首选项)
■	C# 和 Visual Basic

概述

此样式规则涉及简化返回常数值 `true` 或 `false` 的条件表达式，而不是保留具有显式 `true` 或 `false` 返回值的条件表达式。

dotnet_style_prefer_simplified_boolean_expressions

■	dotnet_style_prefer_simplified_boolean_expressions
■	`true` - 首选简化的条件表达式 `false` - 请勿首选简化的条件表达式
■	`true`

示例

```
// dotnet_style_prefer_simplified_boolean_expressions = true
var result1 = M1() && M2();
var result2 = M1() || M2();

// dotnet_style_prefer_simplified_boolean_expressions = false
var result1 = M1() && M2() ? true : false;
var result2 = M1() ? true : M2();
```

```
 ' dotnet_style_prefer_simplified_boolean_expressions = true
Dim result1 = M1() AndAlso M2()
Dim result2 = M1() OrElse M2()

' dotnet_style_prefer_simplified_boolean_expressions = false
Dim result1 = If (M1() AndAlso M2(), True, False)
Dim result2 = If (M1(), True, M2())
```

另请参阅

- 条件运算符
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

将 `typeof` 转换为 `nameof` (IDE0082)

2021/11/16 •

■ ID	IDE0082
■	将 `typeof` 转换为 `nameof`
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 和 Visual Basic

概述

此样式规则建议使用 nameof 运算符而不是 typeof 运算符，然后使用 Name 成员访问。它只有当名称在这两种情况下都相同时才会触发。此规则没有关联的代码样式选项。

示例

```
// Code with violations
var n1 = typeof(T).Name;
var n2 = typeof(int).Name;

// Fixed code
var n1 = nameof(T);
var n2 = nameof(Int32);
```

```
' Code with violations
Dim n1 = GetType(T).Name
Dim n2 = GetType(Integer).Name

' Fixed code
Dim n1 = NameOf(T)
Dim n2 = NameOf(Int32)
```

另请参阅

- nameof 运算符
- typeof 运算符
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

简化 `new` 表达式 (IDE0090)

2021/11/16 •

■ ID	IDE0090
■	简化 `new` 表达式
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C# 9.0+

概述

此样式规则涉及在创建的类型显而易见时使用 C# 9.0 目标类型的 new 表达式。

csharp_style_implicit_object_creation_when_type_is_apparent

▬	csharp_style_implicit_object_creation_when_type_is_apparent
▬	`true` - 当创建的类型显而易见时，首选目标类型的 new 表达式 `false` - 请勿首选目标类型的 `new` 表达式
▬	`true`

示例

```
// csharp_style_implicit_object_creation_when_type_is_apparent = true
C c = new();
C c2 = new() { Field = 0 };

// csharp_style_implicit_object_creation_when_type_is_apparent = false
C c = new C();
C c2 = new C() { Field = 0 };
```

另请参阅

- 目标类型的新表达式
- 表达式级首选项
- 代码样式语言规则
- 代码样式规则参考

Null 检查首选项

2021/11/16 •

.NET null 检查首选项

本节中的样式规则涉及以下 null 检查首选项，这些首选项在 C# 和 Visual Basic 中很常见：

- 使用联合表达式（IDE0029 和 IDE0030）
- 使用 Null 传播 (IDE0031)
- 使用 is null 检查 (IDE0041)

C# null 检查首选项

本节中的样式规则涉及以下特定于 C# 的 null 检查首选项：

- 使用 throw 表达式 (IDE0016)
- 使用条件委托调用 (IDE1005)

另请参阅

- 代码样式规则参考
- 代码样式语言规则

使用 throw 表达式 (IDE0016)

2021/11/16 •

■ ID	IDE0016
■	使用 throw 表达式
■	Style
Subcategory	语言规则(NULL 检查首选项)
▬	C# 7.0+

概述

此样式规则涉及使用 throw 表达式而非 `throw` 语句。

csharp_style_throw_expression

▬	csharp_style_throw_expression
▬	`true` - 更倾向使用 `throw` 表达式，而不是 `throw` 语句 `false` - 更倾向使用 `throw` 语句，而不是 `throw` 表达式
▬	`true`

示例

```
// csharp_style_throw_expression = true
this.s = s ?? throw new ArgumentNullException(nameof(s));

// csharp_style_throw_expression = false
if (s == null) { throw new ArgumentNullException(nameof(s)); }
this.s = s;
```

另请参阅

- throw 表达式
- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

使用 coalesce 表达式（IDE0029 和 IDE0030）

2021/11/16 •

■ ID	IDE0029 和 IDE0030
■	IDE0029：使用 coalesce 表达式（不可为 null 的类型） IDE0030：使用 coalesce 表达式（可以为 null 的类型）
■	Style
Subcategory	语言规则（NULL 检查首选项）
■	C# 和 Visual Basic

概述

这些样式规则涉及使用 null 合并表达式，并与使用带有 null 检查的三元条件表达式进行比较。例如，建议使用 `x ?? y` 而不是 `x != null ? x : y`。在可为 null 和不可为 null 的表达式的上下文中，将使用不同的规则 ID：

- IDE0029：在涉及不可为 null 的表达式时使用。例如，当 `x` 和 `y` 是不可为 null 的引用类型时，建议使用 `x ?? y` 而不是 `x != null ? x : y`。
- IDE0030：在涉及可为 null 的表达式时使用。例如，当 `x` 和 `y` 是可为 null 的值类型或可为 null 的引用类型时，建议使用 `x ?? y` 而不是 `x != null ? x : y`

dotnet_style_coalesce_expression

■	dotnet_style_coalesce_expression
■	`true` - 比起三元运算符检查更倾向 null 合并表达式 `false` - 比起 null 合并表达式更倾向三元运算符检查
■	`true`

示例

```
// dotnet_style_coalesce_expression = true
var v = x ?? y;

// dotnet_style_coalesce_expression = false
var v = x != null ? x : y; // or
var v = x == null ? y : x;
```

```
' dotnet_style_coalesce_expression = true
Dim v = If(x, y)

' dotnet_style_coalesce_expression = false
Dim v = If(x Is Nothing, y, x) ' or
Dim v = If(x IsNot Nothing, x, y)
```

另请参阅

- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

使用 coalesce 表达式（IDE0029 和 IDE0030）

2021/11/16 •

■ ID	IDE0029 和 IDE0030
■	IDE0029：使用 coalesce 表达式（不可为 null 的类型） IDE0030：使用 coalesce 表达式（可以为 null 的类型）
■	Style
Subcategory	语言规则（NULL 检查首选项）
■	C# 和 Visual Basic

概述

这些样式规则涉及使用 null 合并表达式，并与使用带有 null 检查的三元条件表达式进行比较。例如，建议使用 `x ?? y` 而不是 `x != null ? x : y`。在可为 null 和不可为 null 的表达式的上下文中，将使用不同的规则 ID：

- IDE0029：在涉及不可为 null 的表达式时使用。例如，当 `x` 和 `y` 是不可为 null 的引用类型时，建议使用 `x ?? y` 而不是 `x != null ? x : y`。
- IDE0030：在涉及可为 null 的表达式时使用。例如，当 `x` 和 `y` 是可为 null 的值类型或可为 null 的引用类型时，建议使用 `x ?? y` 而不是 `x != null ? x : y`

dotnet_style_coalesce_expression

■	dotnet_style_coalesce_expression
■	`true` - 比起三元运算符检查更倾向 null 合并表达式 `false` - 比起 null 合并表达式更倾向三元运算符检查
■	`true`

示例

```
// dotnet_style_coalesce_expression = true
var v = x ?? y;

// dotnet_style_coalesce_expression = false
var v = x != null ? x : y; // or
var v = x == null ? y : x;
```

```
' dotnet_style_coalesce_expression = true
Dim v = If(x, y)

' dotnet_style_coalesce_expression = false
Dim v = If(x Is Nothing, y, x) ' or
Dim v = If(x IsNot Nothing, x, y)
```

另请参阅

- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

使用 Null 传播 (IDE0031)

2021/11/16 •

■ ID	IDE0031
■	使用 Null 传播
■	Style
Subcategory	语言规则(NULL 检查首选项)
■	C# 6.0+ 和 Visual Basic 14+

概述

此样式规则涉及使用 NULL 条件运算符，而不是带有 NULL 检查的三元条件表达式。

dotnet_style_null_propagation

■	dotnet_style_null_propagation
■	`true` - 如可能，更倾向使用 null 条件运算符 `false` - 如可能，更倾向使用三元 null 检查
■	`true`

示例

```
// dotnet_style_null_propagation = true
var v = o?.ToString();

// dotnet_style_null_propagation = false
var v = o == null ? null : o.ToString(); // or
var v = o != null ? o.String() : null;
```

```
' dotnet_style_null_propagation = true
Dim v = o?.ToString()

' dotnet_style_null_propagation = false
Dim v = If(o Is Nothing, Nothing, o.ToString()) ' or
Dim v = If(o IsNot Nothing, o.ToString(), Nothing)
```

另请参阅

- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

使用 is null 检查 (IDE0041)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0041
■	使用 is null 检查
■	Style
Subcategory	语言规则(NULL 检查首选项)
▬	C# 6.0+ 和 Visual Basic 14+
▬	Visual Studio 2017 15.7 版

概述

此样式规则与是否使用模式匹配的 null 检查和引用相等方法 `object.ReferenceEquals` 相关。

dotnet_style_prefer_is_null_check_over_reference_equality_method

[illegible]	[illegible]
▬	dotnet_style_prefer_is_null_check_over_reference_equality_method
▬	`true` - null 检查优于引用相等性方法 `false` - 引用相等性方法优于 null 检查
▬	`true`

示例

```
// dotnet_style_prefer_is_null_check_over_reference_equality_method = true
if (value is null)
    return;

// dotnet_style_prefer_is_null_check_over_reference_equality_method = false
if (object.ReferenceEquals(value, null))
    return;
```

```
' dotnet_style_prefer_is_null_check_over_reference_equality_method = true
If value Is Nothing
    Return
End If

' dotnet_style_prefer_is_null_check_over_reference_equality_method = false
If Object.ReferenceEquals(value, Nothing)
    Return
End If
```

另请参阅

- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

使用条件委托调用 (IDE1005)

2021/11/16 •

■ ID	IDE1005
■	使用条件委托调用
■	Style
Subcategory	语言规则(NULL 检查首选项)
▬	C# 6.0+

概述

此样式规则涉及在调用 Lambda 表达式时使用 Null 条件运算符 (`?.`), 而非执行 NULL 检查。

csharp_style_conditional_delegate_call

▬	csharp_style_conditional_delegate_call
■	`true` - 在调用 Lambda 表达式时首选使用条件合并运算符 (`?.`), 而非执行 NULL 检查 `false` - 在调用 Lambda 表达式之前执行 NULL 检查为首选项, 而非使用条件合并运算符 (`?.`)
▬	`true`

示例

```
// csharp_style_conditional_delegate_call = true
func?.Invoke(args);

// csharp_style_conditional_delegate_call = false
if (func != null) { func(args); }
```

另请参阅

- Null 条件运算符
- Null 检查首选项
- 代码样式语言规则
- 代码样式规则参考

“var”首选项（IDE0007 和 IDE0008）

2021/11/16 •

■ ID	IDE0007 和 IDE0008
■	IDE0007：使用“var”而不是显式类型 IDE0008：使用显式类型而不是“var”
■	Style
Subcategory	语言规则
▬	C#

概述

本节中的样式规则与变量声明中的 var 关键字和显式类型的使用有关。此规则可单独应用于内置类型、类型明显的位置和其他位置。

csharp_style_var_for_built_in_types

▬	csharp_style_var_for_built_in_types
■	`true` - 使用 `var` 声明 `int` 等内置系统类型的变量为首选项 `false` - 使用显示类型声明 `int` 等内置系统类型的变量为首选项，而非使用 `var`
▬	`true`

示例

```
// csharp_style_var_for_built_in_types = true
var x = 5;

// csharp_style_var_for_built_in_types = false
int x = 5;
```

csharp_style_var_when_type_is_apparent

▬	csharp_style_var_when_type_is_apparent

[illegible]	[illegible]

[illegible]	`true` - 声明表达式右侧已提到该类型时更倾向使用 `var`
	
	`false` - 声明表达式右侧已提到该类型时，使用显式类型为首选项，而非 `var`

[illegible]	`true`

示例

```
// csharp_style_var_when_type_is_apparent = true
var obj = new Customer();

// csharp_style_var_when_type_is_apparent = false
Customer obj = new Customer();
```

csharp_style_var_elsewhere

[illegible]	[illegible]
[illegible]	csharp_style_var_elsewhere
[illegible]	`true` - 在任何情况下，`var` 为首选项，而非显式类型，除非由另一个代码样式规则替代 `false` - 在任何情况下，显式类型为首选项，而非 `var`，除非由另一个代码样式规则替代
[illegible]	`true`

示例

```
// csharp_style_var_elsewhere = true
var f = this.Init();

// csharp_style_var_elsewhere = false
bool f = this.Init();
```

另请参阅

- var 关键字
- 代码样式语言规则
- 代码样式规则参考

“var”首选项（IDE0007 和 IDE0008）

2021/11/16 •

■ ID	IDE0007 和 IDE0008
■	IDE0007：使用“var”而不是显式类型 IDE0008：使用显式类型而不是“var”
■	Style
Subcategory	语言规则
▬	C#

概述

本节中的样式规则与变量声明中的 var 关键字和显式类型的使用有关。此规则可单独应用于内置类型、类型明显的位置和其他位置。

csharp_style_var_for_built_in_types

▬	csharp_style_var_for_built_in_types
■	`true` - 使用 `var` 声明 `int` 等内置系统类型的变量为首选项 `false` - 使用显示类型声明 `int` 等内置系统类型的变量为首选项，而非使用 `var`
▬	`true`

示例

```
// csharp_style_var_for_built_in_types = true
var x = 5;

// csharp_style_var_for_built_in_types = false
int x = 5;
```

csharp_style_var_when_type_is_apparent

▬	csharp_style_var_when_type_is_apparent

[illegible]	[illegible]

[illegible]	`true` - 声明表达式右侧已提到该类型时更倾向使用 `var`
	
	`false` - 声明表达式右侧已提到该类型时，使用显式类型为首选项，而非 `var`

[illegible]	`true`

示例

```
// csharp_style_var_when_type_is_apparent = true
var obj = new Customer();

// csharp_style_var_when_type_is_apparent = false
Customer obj = new Customer();
```

csharp_style_var_elsewhere

[illegible]	[illegible]
[illegible]	csharp_style_var_elsewhere
[illegible]	`true` - 在任何情况下，`var` 为首选项，而非显式类型，除非由另一个代码样式规则替代 `false` - 在任何情况下，显式类型为首选项，而非 `var`，除非由另一个代码样式规则替代
[illegible]	`true`

示例

```
// csharp_style_var_elsewhere = true
var f = this.Init();

// csharp_style_var_elsewhere = false
bool f = this.Init();
```

另请参阅

- var 关键字
- 代码样式语言规则
- 代码样式规则参考

Expression-bodied 成员

2021/11/16 •

概述

本节中的样式规则与在逻辑由单个表达式组成的情况下，expression-bodied 成员的使用有关。

- 使用构造函数的表达式主体 (IDE0021)
- 使用方法的表达式主体 (IDE0022)
- 使用运算符的表达式主体（IDE0023 和 IDE0024）
- 使用属性的表达式主体 (IDE0025)
- 使用索引器的表达式主体 (IDE0026)
- 使用访问器的表达式主体 (IDE0027)
- 使用 Lambda 表达式的表达式主体 (IDE0053)
- 使用本地函数的表达式主体 (IDE0061)

请参阅

- 代码样式规则参考
- 代码样式语言规则

使用构造函数的表达式主体 (IDE0021)

2021/11/16 •

■ ID	IDE0021
■	使用构造函数的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
▬	C# 7.0+

概述

此样式规则涉及到使用构造函数的表达式主体和块主体。

csharp_style_expression_bodied_constructors

▬	csharp_style_expression_bodied_constructors
▬	`true` - 首选构造函数的表达式主体 `when_on_single_line` - 当其将为单行时，首选构造函数的表达式主体 `false` - 倾向于使用构造函数的块主体
▬	`false`

示例

```
// csharp_style_expression_bodied_constructors = true
public Customer(int age) => Age = age;

// csharp_style_expression_bodied_constructors = false
public Customer(int age) { Age = age; }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

使用方法的表达式主体 (IDE0022)

2021/11/16 •

■ ID	IDE0022
■	使用方法的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
■	C# 6.0+

概述

此样式规则涉及使用方法的表达式主体，而不是块主体。

csharp_style_expression_bodied_methods

■	csharp_style_expression_bodied_methods
■	`true` - 首选使用方法的表达式主体 `when_on_single_line` - 当其将为单行时，首先方法的表达式主体 `false` - 优先选择方法的块主体
■	`false`

示例

```
// csharp_style_expression_bodied_methods = true
public int GetAge() => this.Age;

// csharp_style_expression_bodied_methods = false
public int GetAge() { return this.Age; }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

使用运算符的表达式主体（IDE0023 和 IDE0024）

2021/11/16 •

■ ID	IDE0023 和 IDE0024
■	IDE0023：使用转换运算符的表达式主体 IDE0024：使用运算符的表达式主体
■	Style
Subcategory	语言规则（表达式主体成员）
■	C# 7.0+

概述

此样式规则涉及使用运算符的表达式主体，而不是块主体。

csharp_style_expression_bodied_operators

■	csharp_style_expression_bodied_operators
■	`true` - 首选运算符的表达式主体 `when_on_single_line` - 当其将为单行时，首选运算符的表达式主体 `false` - 倾向于使用运算符的块主体
■	`false`

示例

```
// csharp_style_expression_bodied_operators = true
public static ComplexNumber operator + (ComplexNumber c1, ComplexNumber c2)
    => new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

// csharp_style_expression_bodied_operators = false
public static ComplexNumber operator + (ComplexNumber c1, ComplexNumber c2)
{ return new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary); }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则

- 代码样式规则参考

使用运算符的表达式主体（IDE0023 和 IDE0024）

2021/11/16 •

■ ID	IDE0023 和 IDE0024
■	IDE0023：使用转换运算符的表达式主体 IDE0024：使用运算符的表达式主体
■	Style
Subcategory	语言规则（表达式主体成员）
■	C# 7.0+

概述

此样式规则涉及使用运算符的表达式主体，而不是块主体。

csharp_style_expression_bodied_operators

■	csharp_style_expression_bodied_operators
■	`true` - 首选运算符的表达式主体 `when_on_single_line` - 当其将为单行时，首选运算符的表达式主体 `false` - 倾向于使用运算符的块主体
■	`false`

示例

```
// csharp_style_expression_bodied_operators = true
public static ComplexNumber operator + (ComplexNumber c1, ComplexNumber c2)
    => new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary);

// csharp_style_expression_bodied_operators = false
public static ComplexNumber operator + (ComplexNumber c1, ComplexNumber c2)
{ return new ComplexNumber(c1.Real + c2.Real, c1.Imaginary + c2.Imaginary); }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则

- 代码样式规则参考

使用属性的表达式主体 (IDE0025)

2021/11/16 •

■ ID	IDE0025
■	使用属性的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
▬	C# 7.0+

概述

此样式规则涉及使用属性的表达式主体，而不是块主体。

csharp_style_expression_bodied_properties

▬	csharp_style_expression_bodied_properties
▬	`true` - 首选属性的表达式主体 `when_on_single_line` - 当其将为单行时，首选属性的表达式主体 `false` - 倾向于使用属性的块主体
▬	`true`

示例

```
// csharp_style_expression_bodied_properties = true
public int Age => _age;

// csharp_style_expression_bodied_properties = false
public int Age { get { return _age; }}
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

使用索引器的表达式主体 (IDE0026)

2021/11/16 •

■ ID	IDE0026
■	使用索引器的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
■	C# 7.0+

概述

此样式规则涉及使用索引器的表达式主体，而不是块主体。

csharp_style_expression_bodied_indexers

■	csharp_style_expression_bodied_indexers
■	`true` - 首选索引的表达式主体 `when_on_single_line` - 当其将为单行时，首选索引的表达式主体 `false` - 倾向于使用索引器的块主体
■	`true`

示例

```
// csharp_style_expression_bodied_indexers = true
public T this[int i] => _values[i];

// csharp_style_expression_bodied_indexers = false
public T this[int i] { get { return _values[i]; } }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

为访问器使用表达式主体 (IDE0027)

2021/11/16 •

■ ID	IDE0027
■	为访问器使用表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
■	C# 7.0+

概述

此样式规则涉及到为访问器使用表达式主体，并与使用块主体相比较。

csharp_style_expression_bodied_accessors

■	csharp_style_expression_bodied_accessors
■	`true` - 首选取值函数的表达式主体 `when_on_single_line` - 当其将为单行时，首选取值函数的表达式主体 `false` - 倾向于使用访问器的块主体
■	`true`

示例

```
// csharp_style_expression_bodied_accessors = true
public int Age { get => _age; set => _age = value; }

// csharp_style_expression_bodied_accessors = false
public int Age { get { return _age; } set { _age = value; } }
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

使用 Lambda 的表达式主体 (IDE0053)

2021/11/16 •

■ ID	IDE0053
■	使用 Lambda 的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
■	C# 7.0+

概述

此样式规则涉及使用 Lambda 的表达式主体，而不是块主体。

csharp_style_expression_bodied_lambdas

■	csharp_style_expression_bodied_lambdas
■	`true` - 首选 Lambdas 的表达式主体 `when_on_single_line` - 当其将为单行时，首选 lambdas 的表达式主体 `false` - 首选 lambdas 的块主体
■	`true`

示例

```
// csharp_style_expression_bodied_lambdas = true
Func<int, int> square = x => x * x;

// csharp_style_expression_bodied_lambdas = false
Func<int, int> square = x => { return x * x; };
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

使用局部函数的表达式主体 (IDE0061)

2021/11/16 •

■ ID	IDE0061
■	使用局部函数的表达式主体
■	Style
Subcategory	语言规则(表达式主体成员)
■	C# 7.0+

概述

此样式规则涉及使用局部函数的表达式主体和块主体。本地函数是一种嵌套在另一成员中的类型的私有方法。

csharp_style_expression_bodied_local_functions

■	csharp_style_expression_bodied_local_functions
■	`true` - 首选本地函数的表达式主体 `when_on_single_line` - 当其将为单行时，首选本地函数的表达式主体 `false` - 首选本地函数的块主体
■	`false`

示例

```
// csharp_style_expression_bodied_local_functions = true
void M()
{
    Hello();
    void Hello() => Console.WriteLine("Hello");
}

// csharp_style_expression_bodied_local_functions = false
void M()
{
    Hello();
    void Hello()
    {
        Console.WriteLine("Hello");
    }
}
```

另请参阅

- Expression-Bodied 成员
- 适用于表达式主体成员的代码样式规则
- 代码样式语言规则
- 代码样式规则参考

模式匹配首选项

2021/11/16 •

C# 首选项

本节中的样式规则与 C# 中模式匹配的使用有关。

- 使用模式匹配来避免 `as` 后跟 `null` 检查 (IDE0019
- 使用模式匹配来避免 `is` 检查后跟强制转换(IDE0020 和 IDE0038)
- 使用 switch 表达式 (IDE0066)
- 使用模式匹配 (IDE0078)
- 使用模式匹配(`not` 运算符)(IDE0083)

Visual Basic 首选项

本部分中的样式规则涉及使用 Visual Basic 中的模式匹配。

- 使用模式匹配(`IsNot` 运算符)(IDE0084)

另请参阅

- 代码样式规则参考
- 代码样式语言规则

使用模式匹配来避免 as 后跟 null 检查 (IDE0019)

2021/11/16 •

[	[
■ ID	IDE0019
■	使用模式匹配来避免 `as` 后跟 `null` 检查
■	Style
Subcategory	语言规则(模式匹配首选项)
▬	C# 7.0+

概述

此样式规则涉及使用 C# 模式匹配而不是后跟 `null` 检查的 `as` 表达式。

csharp_style_pattern_matching_over_as_with_null_check

[	[
▬	csharp_style_pattern_matching_over_as_with_null_check
■	`true` - 倾向于使用模式匹配，而不是带 null 检查的 `as` 表达式，来确定内容是否为某个特定类型 `false` - 倾向于使用带 null 检查的 `as` 表达式，而不是模式匹配，来确定内容是否为某个特定类型
▬	`true`

示例

```
// csharp_style_pattern_matching_over_as_with_null_check = true
if (o is string s) {...}

// csharp_style_pattern_matching_over_as_with_null_check = false
var s = o as string;
if (s != null) {...}
```

另请参阅

- C# 中的模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用模式匹配来避免后跟强制转换的“is”检查 (IDE0020)

2021/11/16 •

■ ID	IDE0020 和 IDE0038
■	IDE0020：使用模式匹配来避免后跟强制转换的 `is` 检查(带变量) IDE0038：使用模式匹配来避免后跟强制转换的 `is` 检查(不带变量)
■	Style
Subcategory	语言规则(模式匹配首选项)
■	C# 7.0+

概述

此样式规则涉及使用 C# 模式匹配而不是后跟强制转换的 `is` 检查。例如，建议使用 `o is int i` 而不是 `if (o is int) { ... (int)o ...` 。选项值确定是首选模式匹配还是首选后跟强制转换的 `is` 检查。根据强制转换表达式是否保存在原始代码内单独的局部变量中，使用不同的规则 ID：

- IDE0020：强制转换表达式保存到局部变量中。例如，原始代码为 `if (o is int) { var i = (int)o; }`，它将 `(int)o` 的结果保存在局部变量中。
- IDE0038：强制转换表达式未保存到局部变量中。例如，原始代码为 `if (o is int) { if ((int)o == 1) { ... } }`，它不会将 `(int)o` 的结果保存在局部变量中。

csharp_style_pattern_matching_over_is_with_cast_check

■	csharp_style_pattern_matching_over_is_with_cast_check
■	`true` - 倾向于使用模式匹配，而不是带类型强制转换的 `is` 表达式 `false` - 倾向于使用带类型强制转换的 `is` 表达式，而不是模式匹配
■	`true`

示例

```
// csharp_style_pattern_matching_over_is_with_cast_check = true
if (o is int i) {...}

// csharp_style_pattern_matching_over_is_with_cast_check = false
if (o is int) {var i = (int)o; ... }
```

另请参阅

- C# 中的模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用模式匹配来避免后跟强制转换的“is”检查 (IDE0020)

2021/11/16 •

■ ID	IDE0020 和 IDE0038
■	IDE0020：使用模式匹配来避免后跟强制转换的 `is` 检查(带变量) IDE0038：使用模式匹配来避免后跟强制转换的 `is` 检查(不带变量)
■	Style
Subcategory	语言规则(模式匹配首选项)
■	C# 7.0+

概述

此样式规则涉及使用 C# 模式匹配而不是后跟强制转换的 `is` 检查。例如，建议使用 `o is int i` 而不是 `if (o is int) { ... (int)o ...` 。选项值确定是首选模式匹配还是首选后跟强制转换的 `is` 检查。根据强制转换表达式是否保存在原始代码内单独的局部变量中，使用不同的规则 ID：

- IDE0020：强制转换表达式保存到局部变量中。例如，原始代码为 `if (o is int) { var i = (int)o; }`，它将 `(int)o` 的结果保存在局部变量中。
- IDE0038：强制转换表达式未保存到局部变量中。例如，原始代码为 `if (o is int) { if ((int)o == 1) { ... } }`，它不会将 `(int)o` 的结果保存在局部变量中。

csharp_style_pattern_matching_over_is_with_cast_check

■	csharp_style_pattern_matching_over_is_with_cast_check
■	`true` - 倾向于使用模式匹配，而不是带类型强制转换的 `is` 表达式 `false` - 倾向于使用带类型强制转换的 `is` 表达式，而不是模式匹配
■	`true`

示例

```
// csharp_style_pattern_matching_over_is_with_cast_check = true
if (o is int i) {...}

// csharp_style_pattern_matching_over_is_with_cast_check = false
if (o is int) {var i = (int)o; ... }
```

另请参阅

- C# 中的模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用 switch 表达式 (IDE0066)

2021/11/16 •

■ ID	IDE0066
■	使用 switch 表达式
■	Style
Subcategory	语言规则(模式匹配首选项)
■	C# 8.0+
■	Visual Studio 2019 版本 16.2

概述

此样式规则涉及使用 switch 表达式而不是 switch 语句。

csharp_style_prefer_switch_expression

■	csharp_style_prefer_switch_expression
■	`true` - 首选使用 `switch` 表达式(使用 C# 8.0 引入) `false` - 首选使用 `switch` 语句
■	`true`

示例

```
// csharp_style_prefer_switch_expression = true
return x switch
{
    1 => 1 * 1,
    2 => 2 * 2,
    _ => 0,
};

// csharp_style_prefer_switch_expression = false
switch (x)
{
    case 1:
        return 1 * 1;
    case 2:
        return 2 * 2;
    default:
        return 0;
}
```

另请参阅

- switch 表达式
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用模式匹配 (IDE0078)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0078
■	使用模式匹配
■	Style
Subcategory	语言规则(模式匹配首选项)
▬	C# 9.0+

概述

此样式规则涉及到使用 C# 9.0 模式匹配构造(如可用)。

csharp_style_prefer_pattern_matching

[illegible]	[illegible]
▬	csharp_style_prefer_pattern_matching
▬	`true` - 如果可能，首选模式匹配构造(已引入 C# 9.0) `false` - 请勿首选模式匹配构造。
▬	`true`

示例

```
// csharp_style_prefer_pattern_matching = true
var x = i is default(int) or > (default(int));
var y = o is not C c;

// csharp_style_prefer_pattern_matching = false
var x = i == default || i > default(int);
var y = !(o is C c);
```

另请参阅

- C# 9.0 模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用模式匹配（`not` 运算符）(IDE0083)

2021/11/16 •

■	■
■ ID	IDE0083
■	使用模式匹配（`not` 运算符）
■	Style
Subcategory	语言规则（模式匹配首选项）
■	C# 9.0+

概述

在可能的情况下，此样式规则与是否使用 C# 9.0"not"模式有关。

csharp_style_prefer_not_pattern

■	■
■	csharp_style_prefer_not_pattern
■	`true` - 在可能的情况下，首选使用"not"模式（随 C# 9.0 一起引入） `false` - 请勿首选使用 `not` 模式。
■	`true`

NOTE

当此选项设置为 `false` 时，分析器不会标记 `not` 模式的使用。但是，生成的任何代码都不会使用 `not` 模式。当此选项设置为 `true` 时，会标记不使用 `not` 模式的代码，并且生成的任何代码都将使用 `not` 模式（如果适用）。

示例

以下示例演示了当此选项设置为 `true` 或 `false` 时，代码生成功能将如何生成代码。

```
// csharp_style_prefer_not_pattern = true
var y = o is not C c;

// csharp_style_prefer_not_pattern = false
var y = !(o is C c);
```

另请参阅

- IDE0078：使用模式匹配
- IDE0084：使用模式匹配（Visual Basic IsNot 运算符）
- C# 9.0 模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

使用模式匹配（`IsNot` 运算符）(IDE0084)

2021/11/16 •

■ ID	IDE0084
■	使用模式匹配（`IsNot` 运算符）
■	Style
Subcategory	语言规则（模式匹配首选项）
▬	Visual Basic 14.0+

概述

此样式规则涉及使用 Visual Basic 14.0 `IsNot` 模式（如可能）。

visual_basic_style_prefer_isnot_expression

▬	visual_basic_style_prefer_isnot_expression
▬	`true` - 在可能情况下，首选使用 `IsNot` 模式（随 Visual Basic 14.0 一起引入） `false` - 请勿首先使用 `IsNot` 模式。
▬	`true`

示例

```
' visual_basic_style_prefer_isnot_expression = true
Dim y = o IsNot C

' visual_basic_style_prefer_isnot_expression = false
Dim y = Not o Is C
```

另请参阅

- IDE0078：使用模式匹配
- IDE0083：使用模式匹配（C# not 运算符）
- C# 9.0 模式匹配
- 模式匹配首选项
- 代码样式语言规则
- 代码样式规则参考

代码块首选项

2021/11/16 •

概述

本节中的样式规则涉及以下代码块首选项：

- 添加大括号 (IDE0011)
- 使用简单的 using 语句 (IDE0063)

另请参阅

- 代码样式规则参考
- 代码样式语言规则

添加大括号 (IDE0011)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0011
■	添加大括号
■	Style
Subcategory	语言规则(代码块首选项)

概述

此样式规则与是否使用大括号 `{ }` 将代码块括起来有关。

csharp_prefer_braces

[illegible]	[illegible]
▬	csharp_prefer_braces
■	`true` - 使用大括号为首选项，即使只有一个代码行，也是如此 `false` - 如可能，不使用大括号为首选项 `when_multiline` - 对多行首选大括号
▬	`true`

示例

```
// csharp_prefer_braces = true
if (test) { this.Display(); }

// csharp_prefer_braces = false
if (test) this.Display();
```

另请参阅

- 代码块首选项
- 代码样式语言规则
- 代码样式规则参考

使用简单的 using 语句 (IDE0063)

2021/11/16 •

■ ID	IDE0063
■	使用简单的 `using` 语句
■	Style
Subcategory	语言规则(代码块首选项)
▬	C# 8.0+

概述

此样式规则涉及使用 `using` 语句(无大括号)。此替代语法是在 C# 8.0 中引入的。

csharp_prefer_simple_using_statement

▬	csharp_prefer_simple_using_statement
■	`true` - 首选使用简单 `using` 语句 `false` - 不推荐使用简单 `using` 语句
▬	`true`

示例

```
// csharp_prefer_simple_using_statement = true
using var a = b;

// csharp_prefer_simple_using_statement = false
using (var a = b) { }
```

另请参阅

- using 语句
- 代码块首选项
- 代码样式语言规则
- 代码样式规则参考

“using”指令放置 (IDE0065)

2021/11/16 •

■ ID	IDE0065
■	`using` 指令放置
■	Style
Subcategory	语言规则(表达式级首选项)
▬	C#

概述

此样式规则与将 `using` 指令置于命名空间外部或内部的偏好相关。

csharp_using_directive_placement

PROPERTY	
▬	csharp_using_directive_placement
■	`outside_namespace` - 首选将 `using` 指令放在名称空间之外 `inside_namespace` - 首选将 `using` 指令放在名称空间中
▬	`outside_namespace`

示例

```
// csharp_using_directive_placement = outside_namespace
using System;

namespace Conventions
{
    ...
}

// csharp_using_directive_placement = inside_namespace
namespace Conventions
{
    using System;
    ...
}
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

需要文件标头 (IDE0073)

2021/11/16 •

■ ID	IDE0073
■	需要文件标头
■	Style
Subcategory	语言规则(文件标头首选项)
■	C# 和 Visual Basic

概述

此样式规则涉及在源代码文件顶部提供文件标头。用 file_header_template 选项指定所需的标头。

- 当 `file_header_template` 选项值为非空字符串时，需要指定的文件标头。
- 当 `file_header_template` 选项值为 `unset` 或空字符串时，不需要文件标头。

file_header_template

■	file_header_template
■	非空字符串，可选择性地包含 `{fileName}` 占位符 - 首选字符串作为所需文件标头。 `unset` 或空字符串 - 不需要文件标头。
■	`unset`

示例

```
// file_header_template = Copyright (c) SomeCorp. All rights reserved.\nLicensed under the xyz license.

// Copyright (c) SomeCorp. All rights reserved.
// Licensed under the xyz license.
namespace N1
{
    class C1 { }
}

// file_header_template = unset
//      OR
// file_header_template =
namespace N2
{
    class C2 { }
}
```

```
' file_header_template = Copyright (c) SomeCorp. All rights reserved.\nLicensed under the xyz license.

' Copyright (c) SomeCorp. All rights reserved.
' Licensed under the xyz license.
Namespace N1
    Class C1
    End Class
End Namespace

' file_header_template = unset
'      OR
' file_header_template =
Namespace N2
    Class C2
    End Class
End Namespace
```

另请参阅

- 代码样式语言规则
- 代码样式规则参考

不必要的代码规则

2021/11/16 •

不必要的代码规则标识代码基础映像中不必要且可以重构或删除的不同部分。出现不必要的代码表示存在以下一个或多个问题：

- 可读性：代码不必降低可读性。例如，IDE0001 标记不必要的类型名限定。
- 可维护性：不必维护重构后不再使用的代码。例如，IDE0051 标记未使用的专用字段、属性、事件和方法。
- 性能：不必要的计算不会产生副作用，但会导致不必要的性能开销。例如，在计算值的表达式不产生副作用的情况下，IDE0059 会标记未使用的值赋值。
- 功能性：代码中的功能性问题会导致所需的代码变得多余。例如，在方法意外忽略输入参数的情况下，IDE0060 会标记未使用的参数。

本节中的规则包括以下不必要的代码规则：

- 简化名称 (IDE0001)
- 简化成员访问 (IDE0002)
- 删除不必要的强制转换 (IDE0004)
- 删除不必要的导入 (IDE0005)
- 删除无法访问的代码 (IDE0035)
- 删除未使用的私有成员 (IDE0051)
- 删除未读的私有成员 (IDE0052)
- 删除未使用的表达式值 (IDE0058)
- 删除不必要的赋值 (IDE0059)
- 删除未使用的参数 (IDE0060)
- 删除不必要的抑制 (IDE0079)
- 删除不必要的抑制运算符 (IDE0080) - 仅限 C#。
- 删除 ByVal (IDE0081) - 仅限 Visual Basic。
- 删除不必要的相等运算符 (IDE0100)
- 删除不必要的弃元 (IDE0110) - 仅限 C#。
- 简化对象创建 (IDE0140) - 仅限 Visual Basic。

其中某些规则包含配置规则行为的选项：

- csharp_style_unused_value_expression_statement_preference (IDE0058)
- visual_basic_style_unused_value_expression_statement_preference (IDE0058)
- csharp_style_unused_value_assignment_preference (IDE0059)
- visual_basic_style_unused_value_assignment_preference (IDE0059)
- dotnet_code_quality_unused_parameters (IDE0060)
- dotnet_remove_unnecessary_suppression_exclusions (IDE0079)
- visual_basic_style_prefer_simplified_object_creation (IDE0140)

请参阅

- 代码样式语言规则
- 代码样式规则参考

简化名称 (IDE0001)

2021/11/16 •

■ ID	IDE0001
■	简化名称
■	Style
Subcategory	不必要的代码规则
▬	C# 和 Visual Basic

概述

此规则涉及在声明和可执行代码中使用简化的类型名称(如可用)。可删除不必要的名称限定,以简化代码、提高可读性。此规则没有关联的代码样式选项。

示例

```
using System.IO;
class C
{
    // IDE0001: 'System.IO.FileInfo' can be simplified to 'FileInfo'
    System.IO.FileInfo file;

    // Fixed code
    FileInfo file;
}
```

```
Imports System.IO
Class C
    ' IDE0001: 'System.IO.FileInfo' can be simplified to 'FileInfo'
    Private file As System.IO.FileInfo

    ' Fixed code
    Private file As FileInfo
End Class
```

另请参阅

- 简化成员访问 (IDE0002)
- 不必要的代码规则
- 代码样式规则参考

简化成员访问 (IDE0002)

2021/11/16 •

■ ID	IDE0002
■	简化成员访问
■	Style
Subcategory	不必要的代码规则
▬	C# 和 Visual Basic

概述

此规则涉及在声明和可执行代码中使用简化的类型成员访问(如可能)。可以删除不必要的限定，以简化代码并提高可读性。此规则没有关联的代码样式选项。

示例

```
static void M1() { }
static void M2()
{
    // IDE0002: 'C.M1' can be simplified to 'M1'
    C.M1();

    // Fixed code
    M1();
}
```

```
Shared Sub M1()
End Sub

Shared Sub M2()
    ' IDE0002: 'C.M1' can be simplified to 'M1'
    C.M1()

    ' Fixed code
    M1()
End Sub
```

另请参阅

- 简化名称 (IDE0001)
- 不必要的代码规则
- 代码样式规则参考

删除不必要的强制转换 (IDE0004)

2021/11/16 •

■ ID	IDE0004
■	删除不必要的 cast
■	Style
Subcategory	不必要的代码规则
▬	C# 和 Visual Basic

概述

此规则标记不必要的类型强制转换。如果在有或没有强制转换表达式的情况下代码语义均相同，则不需要强制转换表达式。此规则没有关联的代码样式选项。

示例

```
// Code with violations
int v = (int)0;

// Fixed code
int v = 0;
```

```
' Code with violations
Dim v As Integer = CType(0, Integer)

' Fixed code
Dim v As Integer = 0
```

另请参阅

- 类型强制转换和转换
- 不必要的代码规则
- 代码样式规则参考

删除不必要的 import (IDE0005)

2021/11/16 •

■ ID	IDE0005
■	删除不必要的 import
■	Style
Subcategory	不必要的代码规则
■	C# 和 Visual Basic

概述

此规则标记以下不必要的构造：

- C# 不必要的 using 指令。
- C# 不必要的 using static 指令。
- Visual Basic 不必要的 Import 语句。

无需更改代码的语义即可删除这些不必要的构造。此规则没有关联的代码样式选项。

> **NOTE**
> 若要在生成时启用此规则，需要为项目启用 XML 文档注释。如需更多详细信息，请参阅此问题。

示例

```
// Code with violations
using System;
using System.IO;    // IDE0005: Using directive is unnecessary
class C
{
    public static void M()
    {
        Console.WriteLine("Hello");
    }
}

// Fixed code
using System;
class C
{
    public static void M()
    {
        Console.WriteLine("Hello");
    }
}
```

```
' Code with violations
Imports System.IO   ' IDE0005: Imports statement is unnecessary
Class C
    Public Shared Sub M()
        Console.WriteLine("Hello")
    End Sub
End Class

' Fixed code
Class C
    Public Shared Sub M()
        Console.WriteLine("Hello")
    End Sub
End Class
```

另请参阅

- C# using 指令
- C# using static 指令
- Visual Basic Import 语句
- 不必要的代码规则
- 代码样式规则参考

删除无法访问的代码 (IDE0035)

2021/11/16 •

■ ID	IDE0035
■	删除无法访问的代码
■	Style
Subcategory	不必要的代码规则
▬	C# 和 Visual Basic

概述

此规则标记方法和属性内无法访问的可执行代码，这些代码永远无法访问，因此可将其删除。此规则没有关联的代码样式选项。

示例

```
// Code with violations
void M()
{
    throw new System.Exception();

    // IDE0035: Remove unreachable code
    int v = 0;
}

// Fixed code
void M()
{
    throw new System.Exception();
}
```

另请参阅

- 删除未使用的私有成员 (IDE0051)
- 删除未读的私有成员 (IDE0052)
- 不必要的代码规则
- 代码样式规则参考

删除未使用的私有成员 (IDE0051)

2021/11/16 •

■ ID	IDE0051
■	删除未使用的私有成员
■	CodeQuality
Subcategory	不必要的代码规则
■	C# 和 Visual Basic

概述

此规则标记没有读取或写入引用的未使用的私有方法、字段、属性和事件。此规则没有关联的代码样式选项。

示例

```
// Code with violations
class C
{
    // IDE0051: Remove unused private members
    private readonly int _fieldPrivate;
    private int PropertyPrivate => 1;
    private int GetNumPrivate() => 1;

    // No IDE0051
    internal readonly int FieldInternal;
    private readonly int _fieldPrivateUsed;
    public int PropertyPublic => _fieldPrivateUsed;
    private int GetNumPrivateUsed() => 1;
    internal int GetNumInternal() => GetNumPrivateUsed();
    public int GetNumPublic() => GetNumPrivateUsed();
}

// Fixed code
class C
{
    // No IDE0051
    internal readonly int FieldInternal;
    private readonly int _fieldPrivateUsed;
    public int PropertyPublic => _fieldPrivateUsed;
    private int GetNumPrivateUsed() => 1;
    internal int GetNumInternal() => GetNumPrivateUsed();
    public int GetNumPublic() => GetNumPrivateUsed();
}
```

另请参阅

- 删除未读的私有成员 (IDE0052)

- 删除无法访问的代码 (IDE0035)
- 不必要的代码规则
- 代码样式规则参考

删除未读取的私有成员 (IDE0052)

2021/11/16 •

■ ID	IDE0052
■	删除未读取的私有成员
■	CodeQuality
Subcategory	不必要的代码规则
■	C# 和 Visual Basic

概述

此规则标记具有一个或多个写入引用但没有读取引用的私有字段和属性。这表示可以重构或删除代码的某些部分，以解决可维护性、性能或功能问题。此规则没有关联的代码样式选项。

示例

```
// Code with violations
class C
{
    // IDE0052: Remove unread private members
    private readonly int _field1;
    private int _field2;
    private int Property { get; set; }

    public C()
    {
        _field1 = 0;
    }

    public void SetMethod()
    {
        _field2 = 0;
        Property = 0;
    }
}

// Fixed code
class C
{
    public C()
    {
    }

    public void SetMethod()
    {
    }
}
```

另请参阅

- 删除未使用的私有成员 (IDE0051)
- 删除无法访问的代码 (IDE0035)
- 不必要的代码规则
- 代码样式规则参考

删除不必要的表达式值 (IDE0058)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0058
■	删除不必要的表达式值
■	Style
Subcategory	不必要的代码规则
■■	C# 和 Visual Basic

概述

此规则标记未使用的表达式值。例如：

```
void M()
{
  Compute(); // IDE0058: computed value is never used.
}

int Compute();
```

用户可执行下列操作之一来解决此冲突：

- 如果表达式无副作用，则可删除整个语句。这样可避免不必要的计算，从而提高性能。
- 如果表达式有副作用，请将赋值项左侧替换为弃元或从未使用过的局部变量。这样可提高代码明确度，明确丢弃未使用的值的意图。此规则的选项涉及到使用弃元和未使用的局部变量。

  ```
  _ = Compute();
  ```

csharp_style_unused_value_expression_statement_preference

PROPERTY	[illegible]
■■	csharp_style_unused_value_expression_statement_preference
■■	C#
■	`discard_variable` - 首选将未使用的表达式分配给 discard `unused_local_variable` - 优先执行：将未使用的表达式分配给从未使用过的局部变量
■■	`discard_variable`

示例

```
// Original code:
System.Convert.ToInt32("35");

// After code fix for IDE0058:

// csharp_style_unused_value_expression_statement_preference = discard_variable
_ = System.Convert.ToInt32("35");

// csharp_style_unused_value_expression_statement_preference = unused_local_variable
var unused = Convert.ToInt32("35");
```

visual_basic_style_unused_value_expression_statement_preference

PROPERTY	[
▬	visual_basic_style_unused_value_expression_statement_preference
▬	Visual Basic
▬	`unused_local_variable` - 优先执行：将未使用的表达式分配给从未使用过的局部变量
▬	`unused_local_variable`

示例

```
' visual_basic_style_unused_value_expression_statement_preference = unused_local_variable
Dim unused = Computation()
```

另请参阅

- 删除不必要的赋值 (IDE0059)
- 不必要的代码规则
- 代码样式规则参考

删除不必要的赋值 (IDE0059)

2021/11/16 •

[	[
■ ID	IDE0059
■	删除不必要的赋值
■	Style
Subcategory	不必要的代码规则
■■	C# 和 Visual Basic

概述

此规则标记不必要的赋值。例如：

```
int v = Compute(); // IDE0059: value written to 'v' is never read, so assignment to 'v' is unnecessary.
v = Compute2();
```

用户可执行下列操作之一来解决此冲突：

- 如果赋值项右侧的表达式无任何副作用，则删除表达式或整个赋值语句。这样可避免不必要的计算，从而提高性能。

  ```
  int v = Compute2();
  ```

- 如果赋值项右侧的表达式有副作用，请将赋值项左侧替换为从未使用过的弃元或局部变量。这样可提高代码明确度，明确丢弃未使用的值的意图。此规则的选项涉及使用弃元和未使用的局部变量。

  ```
  _ = Compute();
  int v = Compute2();
  ```

csharp_style_unused_value_assignment_preference

PROPERTY	[
■■	csharp_style_unused_value_assignment_preference
■■	C#

PROPERTY	▮
▬	`discard_variable` - 在分配未使用的值时，首选使用 discard `unused_local_variable` - 在分配未使用的值时，首选使用本地变量
▬	`discard_variable`

示例

```
// csharp_style_unused_value_assignment_preference = discard_variable
int GetCount(Dictionary<string, int> wordCount, string searchWord)
{
    _ = wordCount.TryGetValue(searchWord, out var count);
    return count;
}

// csharp_style_unused_value_assignment_preference = unused_local_variable
int GetCount(Dictionary<string, int> wordCount, string searchWord)
{
    var unused = wordCount.TryGetValue(searchWord, out var count);
    return count;
}
```

visual_basic_style_unused_value_assignment_preference

PROPERTY	▮
▬	visual_basic_style_unused_value_assignment_preference
▬	Visual Basic
▬	`unused_local_variable` - 在分配未使用的值时，首选使用本地变量
▬	`unused_local_variable`

示例

```
' visual_basic_style_unused_value_assignment_preference = unused_local_variable
Dim unused = Computation()
```

另请参阅

- 删除未使用的表达式值 (IDE0058)
- 不必要的代码规则
- 代码样式规则参考

删除未使用的参数 (IDE0060)

2021/11/16 •

■ ID	IDE0060
■	删除未使用的参数
■	Style
Subcategory	不必要的代码规则
■	C# 和 Visual Basic

概述

此规则用于标记未使用的参数。选项值确定是否应仅为非公共方法或公共和非公共方法标记未使用的参数。

此规则不会标记以丢弃符号 `_` 命名的参数。此外，此规则将忽略以丢弃符号后跟整数命名的参数，例如 `_1`。此行为可降低签名要求所需参数的警告干扰，例如，用作委托的方法、具有特殊属性的参数或其值在运行时由框架隐式访问但未在代码中引用的参数。

dotnet_code_quality_unused_parameters

■	dotnet_code_quality_unused_parameters
■	`all` - 标记具有包含未使用的参数的任何可访问性的方法 `non_public` - 只标记包含未使用的参数的非公共方法
■	`all`

示例

```
// dotnet_code_quality_unused_parameters = all
public int GetNum1(int unusedParam) { return 1; }
internal int GetNum2(int unusedParam) { return 1; }
private int GetNum3(int unusedParam) { return 1; }

// dotnet_code_quality_unused_parameters = non_public
internal int GetNum4(int unusedParam) { return 1; }
private int GetNum5(int unusedParam) { return 1; }
```

```
' dotnet_code_quality_unused_parameters = all
Public Function GetNum1(unused As Integer)
    Return 1
End Function

Friend Function GetNum2(unused As Integer)
    Return 1
End Function

Private Function GetNum3(unused As Integer)
    Return 1
End Function

' dotnet_code_quality_unused_parameters = non_public
Friend Function GetNum4(arg1 As Integer)
    Return 1
End Function

Private Function GetNum5(arg1 As Integer)
    Return 1
End Function
```

请参阅

- 不必要的代码规则
- 代码样式规则参考

删除不必要的抑制 (IDE0079)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0079
■	删除不必要的抑制
■	CodeQuality
Subcategory	不必要的代码规则
▬	C# 和 Visual Basic

概述

此规则标记源中不必要的 pragma 和 SuppressMessageAttribute 特性抑制。源抑制旨在抑制特定部分源代码与编译器和分析器规则的冲突，但不会在代码的其他部分禁用规则。添加抑制通常是为了抑制误报或用户不打算修复的不重要冲突。抑制可能会频繁过时，原因包括：修复规则以防止这些误报，或重构用户代码以呈现冗余抑制。此规则有助于识别此类可删除的冗余抑制。

示例

```
using System.Diagnostics.CodeAnalysis;

class C1
{
    // Necessary pragma suppression
#pragma warning disable IDE0051 // IDE0051: Remove unused member
    private int UnusedMethod() => 0;
#pragma warning restore IDE0051

    // IDE0079: Unnecessary pragma suppression
#pragma warning disable IDE0051 // IDE0051: Remove unused member
    private int UsedMethod() => 0;
#pragma warning restore IDE0051

    public int PublicMethod() => UsedMethod();
}

class C2
{
    // Necessary SuppressMessage attribute suppression
    [SuppressMessage("CodeQuality", "IDE0051:Remove unused private members", Justification = "<Pending>")]
    private int _unusedField;

    // IDE0079: Unnecessary SuppressMessage attribute suppression
    [SuppressMessage("CodeQuality", "IDE0051:Remove unused private members", Justification = "<Pending>")]
    private int _usedField;

    public int PublicMethod2() => _usedField;
}
```

dotnet_remove_unnecessary_suppression_exclusions

	dotnet_remove_unnecessary_suppression_exclusions
	`,` 分隔了必须从分析中排除其抑制的规则 ID 或规则类别（前缀为 `category:`）的列表 `all` - 禁用规则 `none` -对所有规则 ID 和规则类别启用规则
	`none`

示例

```
using System.Diagnostics.CodeAnalysis;

class C1
{
    // 'dotnet_remove_unnecessary_suppression_exclusions = IDE0051'

    // Unnecessary pragma suppression, but not flagged by IDE0079
#pragma warning disable IDE0051 // IDE0051: Remove unused member
    private int UsedMethod() => 0;
#pragma warning restore IDE0051

    public int PublicMethod() => UsedMethod();
}
```

另请参阅

- pragma
- SuppressMessageAttribute
- 不必要的代码规则
- 代码样式规则参考

删除不必要的抑制运算符 (IDE0080)

2021/11/16 •

■ ID	IDE0080
■	删除不必要的抑制运算符
■	Style
Subcategory	不必要的代码规则
▬	C#

概述

此规则标记不必要的抑制或 null 包容运算符(当该运算符在它不起作用的上下文中使用时)。使用抑制运算符(如 `x!`)来声明引用类型的 `x` 表达式不为 null。但是,在另一个运算符(如 `o !is string` 中的 is 运算符)的上下文中使用时,它不起作用,就可以删除。

此规则没有关联的代码样式选项。

示例

```
// Code with violations
if (o !is string) { }

// Potential fixes:
// 1.
if (o is not string) { }

// 2.
if (!(o is string)) { }

// 3.
if (o is string) { }
```

另请参阅

- 抑制或 null 包容运算符
- 不必要的代码规则
- 代码样式规则参考

删除 `ByVal` (IDE0081)

2021/11/16 •

■ ID	IDE0081
■	删除了 `ByVal`
■	Style
Subcategory	不必要的代码规则
▬	Visual Basic

概述

此规则标记 Visual Basic 的参数声明中不必要的 ByVal 关键字。默认情况下，Visual Basic 中的参数为 `ByVal`，因此无需在方法签名中进行显式指定。它通常会产生干扰代码，经常导致非默认的 ByRef 关键字被忽略。此规则没有关联的代码样式选项。

示例

```
' Code with violations
Sub M(ByVal p1 As Integer, ByRef p2 As Integer)
End Sub

' Fixed code
Sub M(p1 As Integer, ByRef p2 As Integer)
End Sub
```

另请参阅

- ByVal
- RemoveHandler
- 不必要的代码规则
- 代码样式规则参考

删除不必要的相等运算符 (IDE0100)

2021/11/16 •

■ ID	IDE0100
■	删除不必要的相等运算符
■	Style
Subcategory	不必要的代码规则
■	C# 和 Visual Basic

概述

将非常量布尔表达式与常量 `true` 或 `false` 进行比较时，此样式规则标记不必要的相等运算符。此规则没有关联的代码样式选项。

示例

```
// Code with violations
if (x == true) { }
if (M() != false) { }

// Fixed code
if (x) { }
if (M()) { }
```

```
' Code with violations
If x = True Then
End If

If M() <> False Then
End If

' Fixed code
If x Then
End If

If M() Then
End If
```

请参阅

- 不必要的代码规则
- 代码样式规则参考

删除不必要的弃元 (IDE0110)

2021/11/16 •

■ ID	IDE0110
■	删除不必要的弃元
■	Style
Subcategory	不必要的代码规则
▬	C#

概述

此规则可标记不必要的弃元模式(在无法生效的上下文中使用此弃元模式时)。

此规则没有关联的代码样式选项。

示例

```
// Code with violations
switch (o)
{
    case int _:
        Console.WriteLine("Value was an int");
        break;
    case string _:
        Console.WriteLine("Value was a string");
        break;
}

// Fixed code
switch (o)
{
    case int:
        Console.WriteLine("Value was an int");
        break;
    case string:
        Console.WriteLine("Value was a string");
        break;
}
```

请参阅

- 不必要的代码规则
- 代码样式规则参考

简化对象的创建 (IDE0140)

2021/11/16 •

[illegible]	[illegible]
■ ID	IDE0140
■	简化对象的创建
■	Style
Subcategory	不必要的代码规则
▬	Visual Basic

概述

此样式规则可标记代码中不必要的类型重复。

visual_basic_style_prefer_simplified_object_creation

[illegible]	[illegible]
▬	visual_basic_style_prefer_simplified_object_creation
▬	`true` - 首选简化的对象创建形式 `false` - 禁用此规则。
▬	`true`

示例

```
' Code with violations
Dim x As Student = New Student()

' Fixed code
Dim x As New Student()
```

请参阅

- 不必要的代码规则
- 代码样式规则参考

杂项规则

2021/11/16 •

本部分包含不适合任何其他类别的代码样式规则。其他规则包括：

- 删除无效的全局“SuppressMessageAttribute”(IDE0076)
- 避免在全局“SuppressMessageAttribute”中使用旧格式目标 (IDE0077)

另请参阅

- 代码样式规则参考

删除无效的全局 SuppressMessageAttribute (IDE0076)

2021/11/16 •

■ ID	IDE0076
■	删除无效的全局 `SuppressMessageAttribute`
■	CodeQuality
Subcategory	杂项规则
■	C# 和 Visual Basic

概述

此规则标记具有无效 `Scope` 或 `Target` 的全局 SuppressMessageAttributes。应删除此属性或对其进行修补，以引用有效的作用域和目标符号。此规则没有关联的代码样式选项。

示例

```
// IDE0076: Invalid target '~F:N.C.F2' - no matching field named 'F2'
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "member", Target
= "~F:N.C.F2")]
// IDE0076: Invalid scope 'property'
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "property",
Target = "~P:N.C.P")]

// Fixed code
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "member", Target
= "~F:N.C.F")]
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "member", Target
= "~P:N.C.P")]

namespace N
{
    class C
    {
        public int F;
        public int P { get; }
    }
}
```

另请参阅

- 全局 SuppressMessageAttribute
- 避免在全局 SuppressMessageAttribute 中使用旧格式目标 (IDE0077)
- 代码样式规则参考

避免在全局“SuppressMessageAttribute”中使用旧格式目标 (IDE0077)

2021/11/16 •

■ ID	IDE0077
■	避免在全局 `SuppressMessageAttribute` 中使用旧格式目标
■	CodeQuality
Subcategory	杂项规则
■	C# 和 Visual Basic

概述

该规则标记使用旧版 FxCop 目标字符串格式指定 `Target` 的全局 SuppressMessageAttributes。已知使用旧格式 `Target` 会出现性能问题，应避免使用。有关详细信息，请参阅此 GitHub 问题。

`Target` 的建议格式是文档 ID 格式。有关文档 ID 的信息，请参阅文档 ID 格式。

> **TIP**
> Visual Studio 2019 提供了一个代码修补程序，可将属性的 `Target` 自动更改为建议的格式。

此规则没有关联的代码样式选项。

示例

```
// IDE0077: Legacy format target 'N.C.#F'
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "member", Target
= "N.C.#F")]

// Fixed code
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage("Category", "Id: Title", Scope = "member", Target
= "~F:N.C.F")]

namespace N
{
    class C
    {
        public int F;
    }
}
```

另请参阅

- Global SuppressMessageAttribute

- 旧格式属性“Target”的性能问题
- 删除无效的全局“SuppressMessageAttribute”(IDE0076)
- 文档 ID 格式

格式设置规则

2021/11/16 •

格式设置规则会影响 .NET 编程语言构造的缩进、空格和换行的排列方式。规则分为以下几类：

- .NET 格式设置规则：适用于 C# 和 Visual Basic 的规则。这些规则的 EditorConfig 选项名称以 `dotnet_` 前缀开头。
- C# 格式设置规则：仅适用于 C# 语言的规则。这些规则的 EditorConfig 选项名称以 `csharp_` 前缀开头。

规则 ID："IDE0055"（修复格式设置）

所有格式设置选项都具有规则 ID `IDE0055` 和标题 `Fix formatting`。使用以下配置行在 EditorConfig 文件中设置格式设置冲突的严重性。

```
dotnet_diagnostic.IDE0055.severity = <severity value>
```

严重性值必须是 `warning` 或 `error` 才能在生成时强制执行。要了解所有可能的严重性值，请参阅严重性级别。

选项格式

可以在 EditorConfig 文件中指定格式设置规则的选项，格式如下：

```
rule_name = value
```

对于许多规则，可为 `value` 指定 `true`（以此样式为首选项）或 `false`（不以此样式为首选项）。对于其他规则，可指定值（如 `flush_left` 或 `before_and_after`）来说明在什么时间以及在什么位置应用此规则。不需要指定严重性。

.NET 格式设置规则

本节中的格式设置规则适用于 C# 和 Visual Basic。

- 组织 using
  - dotnet_sort_system_directives_first
  - dotnet_separate_import_directive_groups
- 命名空间选项
  - dotnet_style_namespace_match_folder

组织 using 指令

这些格式设置规则与 `using` 指令和 `Imports` 语句的排序和显示有关。

.editorconfig 文件示例：

```
# .NET formatting rules
[*.{cs,vb}]
dotnet_sort_system_directives_first = true
dotnet_separate_import_directive_groups = true
```

dotnet_sort_system_directives_first

PROPERTY	[illegible]
[illegible]	dotnet_sort_system_directives_first
[illegible]	C# 和 Visual Basic
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 按字母顺序对 `System.*` `using` 指令排序，并将它们放在其他 using 指令前面。 `false` - 不要将 `System.*` `using` 指令放在其他 `using` 指令前面。

代码示例：

```
// dotnet_sort_system_directives_first = true
using System.Collections.Generic;
using System.Threading.Tasks;
using Octokit;

// dotnet_sort_system_directives_first = false
using System.Collections.Generic;
using Octokit;
using System.Threading.Tasks;
```

dotnet_separate_import_directive_groups

PROPERTY	[illegible]
[illegible]	dotnet_separate_import_directive_groups
[illegible]	C# 和 Visual Basic
[illegible]	Visual Studio 2017 版本 15.5
[illegible]	`true` - 在 `using` 指令组之间放置一个空白行。 `false` - 不在 `using` 指令组之间放置空白行。

代码示例：

```
// dotnet_separate_import_directive_groups = true
using System.Collections.Generic;
using System.Threading.Tasks;

using Octokit;

// dotnet_separate_import_directive_groups = false
using System.Collections.Generic;
using System.Threading.Tasks;
using Octokit;
```

Dotnet 命名空间选项

这些格式设置规则与 C# 和 Visual Basic 中命名空间的声明有关。

.editorconfig 文件示例：

```
# .NET namespace rules
[*.{cs,vb}]
dotnet_style_namespace_match_folder = true
```

dotnet_style_namespace_match_folder

[illegible]	[illegible]
[illegible]	dotnet_style_namespace_match_folder
[illegible]	C# 和 Visual Basic
[illegible]	Visual Studio 2019 版本 16.10
[illegible]	`true` - 将命名空间与文件夹结构匹配 `false` - 不报告与文件夹结构不匹配的命名空间

代码示例：

```
// dotnet_style_namespace_match_folder = true
// file path: Example/Convention/C.cs
using System;

namespace Example.Convention
{
    class C
    {
    }
}

// dotnet_style_namespace_match_folder = false
// file path: Example/Convention/C.cs
using System;

namespace Example
{
    class C
    {
    }
}
```

C# 格式设置规则

本节中的格式设置规则仅适用于 C# 代码。

- 换行符选项
  - csharp_new_line_before_open_brace
  - csharp_new_line_before_else
  - csharp_new_line_before_catch
  - csharp_new_line_before_finally
  - csharp_new_line_before_members_in_object_initializers
  - csharp_new_line_before_members_in_anonymous_types
  - csharp_new_line_between_query_expression_clauses
- 缩进选项
  - csharp_indent_case_contents

    - csharp_indent_switch_labels
    - csharp_indent_labels
    - csharp_indent_block_contents
    - csharp_indent_braces
    - csharp_indent_case_contents_when_block
- 间距选项
    - csharp_space_after_cast
    - csharp_space_after_keywords_in_control_flow_statements
    - csharp_space_between_parentheses
    - csharp_space_before_colon_in_inheritance_clause
    - csharp_space_after_colon_in_inheritance_clause
    - csharp_space_around_binary_operators
    - csharp_space_between_method_declaration_parameter_list_parentheses
    - csharp_space_between_method_declaration_empty_parameter_list_parentheses
    - csharp_space_between_method_declaration_name_and_open_parenthesis
    - csharp_space_between_method_call_parameter_list_parentheses
    - csharp_space_between_method_call_empty_parameter_list_parentheses
    - csharp_space_between_method_call_name_and_opening_parenthesis
    - csharp_space_after_comma
    - csharp_space_before_comma
    - csharp_space_after_dot
    - csharp_space_before_dot
    - csharp_space_after_semicolon_in_for_statement
    - csharp_space_before_semicolon_in_for_statement
    - csharp_space_around_declaration_statements
    - csharp_space_before_open_square_brackets
    - csharp_space_between_empty_square_brackets
    - csharp_space_between_square_brackets
- 换行选项
    - csharp_preserve_single_line_statements
    - csharp_preserve_single_line_blocks
- sing 指令选项
    - csharp_using_directive_placement
- 命名空间选项
    - csharp_style_namespace_declarations

换行选项

这些格式设置规则与是否使用新行设置代码的格式有关。

.editorconfig 文件示例：

```
# CSharp formatting rules:
[*.cs]
csharp_new_line_before_open_brace = methods, properties, control_blocks, types
csharp_new_line_before_else = true
csharp_new_line_before_catch = true
csharp_new_line_before_finally = true
csharp_new_line_before_members_in_object_initializers = true
csharp_new_line_before_members_in_anonymous_types = true
csharp_new_line_between_query_expression_clauses = true
```

csharp_new_line_before_open_brace

此规则与左大括号 `{` 应放在前面代码的同一行还是新行上有关。对于此规则，指定"全部"、"无"或一个或多个代码位元素，如方法或属性，从而定义此规则的应用时间 。若要指定多个代码元素，请使用逗号 (,) 分隔。

PROPERTY	[
▬	csharp_new_line_before_open_brace
▬	C#
▬	Visual Studio 2017 版本 15.3
▬	`all` - 对于所有表达式，需要将大括号置于新行("Allman"样式)。 `none` - 对于所有表达式，需要将大括号置于同一行("K&R")。 `accessors`、`anonymous_methods`、`anonymous_types`、`control_blocks`、`events`、`indexers`、`lambdas`、`local_functions`、`methods`、`object_collection_array_initializers`、`properties`、`types` - 对于指定的代码元素，需要将大括号置于新行("Allman"样式)。

代码示例：

```
// csharp_new_line_before_open_brace = all
void MyMethod()
{
    if (...)
    {
        ...
    }
}

// csharp_new_line_before_open_brace = none
void MyMethod() {
    if (...) {
        ...
    }
}
```

csharp_new_line_before_else

PROPERTY	[
▬	csharp_new_line_before_else

PROPERTY	[illegible]
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 将 `else` 语句置于新行。 `false` - 将 `else` 语句置于同一行。

代码示例：

```
// csharp_new_line_before_else = true
if (...) {
    ...
}
else {
    ...
}

// csharp_new_line_before_else = false
if (...) {
    ...
} else {
    ...
}
```

csharp_new_line_before_catch

PROPERTY	[illegible]
[illegible]	csharp_new_line_before_catch
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 将 `catch` 语句置于新行。 `false` - 将 `catch` 语句置于同一行。

代码示例：

```
// csharp_new_line_before_catch = true
try {
    ...
}
catch (Exception e) {
    ...
}

// csharp_new_line_before_catch = false
try {
    ...
} catch (Exception e) {
    ...
}
```

csharp_new_line_before_finally

PROPERTY	[illegible]
[illegible]	csharp_new_line_before_finally
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 需要将 `finally` 语句置于右大括号后的新行。 `false` - 需要将 `finally` 语句置于右大括号所在的同一行。

代码示例:

```
// csharp_new_line_before_finally = true
try {
    ...
}
catch (Exception e) {
    ...
}
finally {
    ...
}

// csharp_new_line_before_finally = false
try {
    ...
} catch (Exception e) {
    ...
} finally {
    ...
}
```

csharp_new_line_before_members_in_object_initializers

PROPERTY	[illegible]
[illegible]	csharp_new_line_before_members_in_object_initializers
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 需要将对象初始值设定项的成员置于单独的行 `false` - 需要将对象初始值设定项的成员置于同一行

代码示例:

```
// csharp_new_line_before_members_in_object_initializers = true
var z = new B()
{
    A = 3,
    B = 4
}

// csharp_new_line_before_members_in_object_initializers = false
var z = new B()
{
    A = 3, B = 4
}
```

csharp_new_line_before_members_in_anonymous_types

PROPERTY	[illegible]
[illegible]	csharp_new_line_before_members_in_anonymous_types
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 需要将匿名类型的成员置于单独的行 `false` - 需要将匿名类型的成员置于同一行

代码示例:

```
// csharp_new_line_before_members_in_anonymous_types = true
var z = new
{
    A = 3,
    B = 4
}

// csharp_new_line_before_members_in_anonymous_types = false
var z = new
{
    A = 3, B = 4
}
```

csharp_new_line_between_query_expression_clauses

PROPERTY	[illegible]
[illegible]	csharp_new_line_between_query_expression_clauses
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 要求将查询表达式子句的元素置于单独的行 `false` - 要求将查询表达式子句的元素置于同一行

代码示例:

```
// csharp_new_line_between_query_expression_clauses = true
var q = from a in e
        from b in e
        select a * b;

// csharp_new_line_between_query_expression_clauses = false
var q = from a in e from b in e
        select a * b;
```

缩进选项

这些格式设置规则与是否使用缩进设置代码的格式有关。

.editorconfig 文件示例：

```
# CSharp formatting rules:
[*.cs]
csharp_indent_case_contents = true
csharp_indent_switch_labels = true
csharp_indent_labels = flush_left
csharp_indent_block_contents = true
csharp_indent_braces = false
csharp_indent_case_contents_when_block = true
```

csharp_indent_case_contents

PROPERTY	[illegible]
[illegible]	csharp_indent_case_contents
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 缩进 `switch` case 内容 `false` - 不缩进 `switch` case 内容

代码示例：

```
// csharp_indent_case_contents = true
switch(c) {
    case Color.Red:
        Console.WriteLine("The color is red");
        break;
    case Color.Blue:
        Console.WriteLine("The color is blue");
        break;
    default:
        Console.WriteLine("The color is unknown.");
        break;
}

// csharp_indent_case_contents = false
switch(c) {
    case Color.Red:
    Console.WriteLine("The color is red");
    break;
    case Color.Blue:
    Console.WriteLine("The color is blue");
    break;
    default:
    Console.WriteLine("The color is unknown.");
    break;
}
```

csharp_indent_switch_labels

PROPERTY	[illegible]
[illegible]	csharp_indent_switch_labels
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 缩进 `switch` 标签 `false` - 不缩进 `switch` 标签

代码示例:

```
// csharp_indent_switch_labels = true
switch(c) {
    case Color.Red:
        Console.WriteLine("The color is red");
        break;
    case Color.Blue:
        Console.WriteLine("The color is blue");
        break;
    default:
        Console.WriteLine("The color is unknown.");
        break;
}

// csharp_indent_switch_labels = false
switch(c) {
case Color.Red:
    Console.WriteLine("The color is red");
    break;
case Color.Blue:
    Console.WriteLine("The color is blue");
    break;
default:
    Console.WriteLine("The color is unknown.");
    break;
}
```

csharp_indent_labels

PROPERTY	[illegible]
[illegible]	csharp_indent_labels
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`flush_left` - 标签置于最左侧的列 `one_less_than_current` - 将标签置于比当前上下文少一个缩进的位置 `no_change` - 将标签置于与当前上下文相同的缩进位置

代码示例:

```
// csharp_indent_labels= flush_left
class C
{
    private string MyMethod(...)
    {
        if (...) {
            goto error;
        }
error:
        throw new Exception(...);
    }
}

// csharp_indent_labels = one_less_than_current
class C
{
    private string MyMethod(...)
    {
        if (...) {
            goto error;
        }
    error:
        throw new Exception(...);
    }
}

// csharp_indent_labels= no_change
class C
{
    private string MyMethod(...)
    {
        if (...) {
            goto error;
        }
        error:
        throw new Exception(...);
    }
}
```

csharp_indent_block_contents

PROPERTY	[illegible]
[illegible]	csharp_indent_block_contents
[illegible]	C#
[illegible]	`true` - `false` -

代码示例:

```
// csharp_indent_block_contents = true
static void Hello()
{
    Console.WriteLine("Hello");
}

// csharp_indent_block_contents = false
static void Hello()
{
Console.WriteLine("Hello");
}
```

csharp_indent_braces

PROPERTY	[illegible]
[illegible]	csharp_indent_braces
[illegible]	C#
[illegible]	`true` - `false` -

代码示例:

```
// csharp_indent_braces = true
static void Hello()
    {
    Console.WriteLine("Hello");
    }

// csharp_indent_braces = false
static void Hello()
{
    Console.WriteLine("Hello");
}
```

csharp_indent_case_contents_when_block

PROPERTY	[illegible]
[illegible]	csharp_indent_case_contents_when_block
[illegible]	C#
[illegible]	`true` - `false` -

代码示例:

```
// csharp_indent_case_contents_when_block = true
case 0:
    {
        Console.WriteLine("Hello");
        break;
    }

// csharp_indent_case_contents_when_block = false
case 0:
{
    Console.WriteLine("Hello");
    break;
}
```

间距选项

这些格式设置规则与是否使用空格字符设置代码的格式有关。

.editorconfig 文件示例：

```
# CSharp formatting rules:
[*.cs]
csharp_space_after_cast = true
csharp_space_after_keywords_in_control_flow_statements = true
csharp_space_between_parentheses = control_flow_statements, type_casts
csharp_space_before_colon_in_inheritance_clause = true
csharp_space_after_colon_in_inheritance_clause = true
csharp_space_around_binary_operators = before_and_after
csharp_space_between_method_declaration_parameter_list_parentheses = true
csharp_space_between_method_declaration_empty_parameter_list_parentheses = false
csharp_space_between_method_declaration_name_and_open_parenthesis = false
csharp_space_between_method_call_parameter_list_parentheses = true
csharp_space_between_method_call_empty_parameter_list_parentheses = false
csharp_space_between_method_call_name_and_opening_parenthesis = false
csharp_space_after_comma = true
csharp_space_before_comma = false
csharp_space_after_dot = false
csharp_space_before_dot = false
csharp_space_after_semicolon_in_for_statement = true
csharp_space_before_semicolon_in_for_statement = false
csharp_space_around_declaration_statements = false
csharp_space_before_open_square_brackets = false
csharp_space_between_empty_square_brackets = false
csharp_space_between_square_brackets = false
```

csharp_space_after_cast

PROPERTY	[illegible]
[illegible]	csharp_space_after_cast
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 在强制转换和值之间放置空格字符 `false` - 删除转换和值之间的空格

代码示例：

```
// csharp_space_after_cast = true
int y = (int) x;

// csharp_space_after_cast = false
int y = (int)x;
```

csharp_space_after_keywords_in_control_flow_statements

PROPERTY	[illegible]
[illegible]	csharp_space_after_keywords_in_control_flow_statements
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 在控制流语句(如 `for` 循环)中的关键字后放置空格字符 `false` - 删除控制流语句(如 `for` 循环)中的关键字后的空格

代码示例:

```
// csharp_space_after_keywords_in_control_flow_statements = true
for (int i;i<x;i++) { ... }

// csharp_space_after_keywords_in_control_flow_statements = false
for(int i;i<x;i++) { ... }
```

csharp_space_between_parentheses

PROPERTY	[illegible]
[illegible]	csharp_space_between_parentheses
[illegible]	C#
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`control_flow_statements` - 在控制流语句的括号之间放置空格 `expressions` - 在表达式的括号之间放置空格 `type_casts` - 在类型转换中的括号之间放置空格

如果省略此规则或使用 `control_flow_statements`、`expressions` 或 `type_casts` 以外的值，则不会应用该设置。

代码示例:

```
// csharp_space_between_parentheses = control_flow_statements
for ( int i = 0; i < 10; i++ ) { }

// csharp_space_between_parentheses = expressions
var z = ( x * y ) - ( ( y - x ) * 3 );

// csharp_space_between_parentheses = type_casts
int y = ( int )x;
```

csharp_space_before_colon_in_inheritance_clause

PROPERTY	[illegible]
[illegible]	csharp_space_before_colon_in_inheritance_clause
[illegible]	C#
[illegible]	Visual Studio 2017 15.7 版
[illegible]	`true` - 在类型声明中的基或接口冒号前放置空格字符 `false` - 删除类型声明中基或接口冒号前的空格

代码示例:

```
// csharp_space_before_colon_in_inheritance_clause = true
interface I
{

}

class C : I
{

}

// csharp_space_before_colon_in_inheritance_clause = false
interface I
{

}

class C: I
{

}
```

csharp_space_after_colon_in_inheritance_clause

PROPERTY	[illegible]
[illegible]	csharp_space_after_colon_in_inheritance_clause
[illegible]	C#
[illegible]	Visual Studio 2017 15.7 版

PROPERTY	[
■	`true` - 在类型声明中的基或接口冒号后放置空格字符 `false` - 删除类型声明中基或接口冒号后的空格

代码示例:

```
// csharp_space_after_colon_in_inheritance_clause = true
interface I
{

}

class C : I
{

}

// csharp_space_after_colon_in_inheritance_clause = false
interface I
{

}

class C :I
{

}
```

csharp_space_around_binary_operators

PROPERTY	[
■	csharp_space_around_binary_operators
■	C#
■	Visual Studio 2017 15.7 版
■	`before_and_after` - 在二元运算符前后插入空格 `none` - 删除二元运算符前后的空格 `ignore` - 忽略二元运算符前后的空格

如果省略此规则或使用 `before_and_after`、`none` 或 `ignore` 以外的值，则不会应用该设置。

代码示例:

```
// csharp_space_around_binary_operators = before_and_after
return x * (x - y);

// csharp_space_around_binary_operators = none
return x*(x-y);

// csharp_space_around_binary_operators = ignore
return x  *  (x-y);
```

csharp_space_between_method_declaration_parameter_list_parentheses

PROPERTY	[
▬	csharp_space_between_method_declaration_parameter_list_parentheses
▬	C#
▬	Visual Studio 2017 版本 15.3
▬	`true` - 在方法声明参数列表的左括号之后和右括号之前放置空格字符 `false` - 删除方法声明参数列表的左括号之后和右括号之前的空格字符

代码示例:

```
// csharp_space_between_method_declaration_parameter_list_parentheses = true
void Bark( int x ) { ... }

// csharp_space_between_method_declaration_parameter_list_parentheses = false
void Bark(int x) { ... }
```

csharp_space_between_method_declaration_empty_parameter_list_parentheses

PROPERTY	[
▬	csharp_space_between_method_declaration_empty_parameter_list_parentheses
▬	C#
▬	Visual Studio 2017 15.7 版
▬	`true` - 在方法声明的空参数列表括号内插入空格 `false` - 删除方法声明的空参数列表括号内的空格

代码示例:

```
// csharp_space_between_method_declaration_empty_parameter_list_parentheses = true
void Goo( )
{
    Goo(1);
}

void Goo(int x)
{
    Goo();
}

// csharp_space_between_method_declaration_empty_parameter_list_parentheses = false
void Goo()
{
    Goo(1);
}

void Goo(int x)
{
    Goo();
}
```

csharp_space_between_method_declaration_name_and_open_parenthesis

PROPERTY	[redacted]
[redacted]	csharp_space_between_method_declaration_name_and_open_parenthesis
[redacted]	C#
[redacted]	`true` - 在方法声明中方法名称和左括号之间放置空格字符 `false` - 删除方法声明中方法名称和左括号之间的空格字符

代码示例:

```
// csharp_space_between_method_declaration_name_and_open_parenthesis = true
void M () { }

// csharp_space_between_method_declaration_name_and_open_parenthesis = false
void M() { }
```

csharp_space_between_method_call_parameter_list_parentheses

PROPERTY	[redacted]
[redacted]	csharp_space_between_method_call_parameter_list_parentheses
[redacted]	C#
[redacted]	Visual Studio 2017 版本 15.3
[redacted]	`true` - 在方法调用的左括号之后和右括号之前放置空格字符 `false` - 删除方法调用的左括号之后和右括号之前的空格字符

代码示例：

```
// csharp_space_between_method_call_parameter_list_parentheses = true
MyMethod( argument );

// csharp_space_between_method_call_parameter_list_parentheses = false
MyMethod(argument);
```

csharp_space_between_method_call_empty_parameter_list_parentheses

PROPERTY	[redacted]
[redacted]	csharp_space_between_method_call_empty_parameter_list_parentheses
[redacted]	C#
[redacted]	Visual Studio 2017 15.7 版
[redacted]	`true` - 在空参数列表的括号中插入空格 `false` - 删除空参数列表括号内的空格

代码示例：

```
// csharp_space_between_method_call_empty_parameter_list_parentheses = true
void Goo()
{
    Goo(1);
}

void Goo(int x)
{
    Goo( );
}

// csharp_space_between_method_call_empty_parameter_list_parentheses = false
void Goo()
{
    Goo(1);
}

void Goo(int x)
{
    Goo();
}
```

csharp_space_between_method_call_name_and_opening_parenthesis

PROPERTY	[redacted]
[redacted]	csharp_space_between_method_call_name_and_opening_parenthesis
[redacted]	C#
[redacted]	Visual Studio 2017 15.7 版

PROPERTY	[
■	`true` - 在方法调用名称和左括号之间插入空格 `false` - 删除方法调用名称和左括号之间的空格

代码示例:

```
// csharp_space_between_method_call_name_and_opening_parenthesis = true
void Goo()
{
    Goo (1);
}

void Goo(int x)
{
    Goo ();
}

// csharp_space_between_method_call_name_and_opening_parenthesis = false
void Goo()
{
    Goo(1);
}

void Goo(int x)
{
    Goo();
}
```

csharp_space_after_comma

PROPERTY	[
■	csharp_space_after_comma
■	C#
■	`true` - 在逗号后面插入空格 `false` - 删除逗号后面的空格

代码示例:

```
// csharp_space_after_comma = true
int[] x = new int[] { 1, 2, 3, 4, 5 };

// csharp_space_after_comma = false
int[] x = new int[] { 1,2,3,4,5 }
```

csharp_space_before_comma

PROPERTY	[
■	csharp_space_before_comma
■	C#

PROPERTY	[illegible]
[illegible]	`true` - 在逗号前插入空格 `false` - 删除逗号前的空格

代码示例:

```
// csharp_space_before_comma = true
int[] x = new int[] { 1 , 2 , 3 , 4 , 5 };

// csharp_space_before_comma = false
int[] x = new int[] { 1, 2, 3, 4, 5 };
```

csharp_space_after_dot

PROPERTY	[illegible]
[illegible]	csharp_space_after_dot
[illegible]	C#
[illegible]	`true` - 在点号后面插入空格 `false` - 删除点号后面的空格

代码示例:

```
// csharp_space_after_dot = true
this. Goo();

// csharp_space_after_dot = false
this.Goo();
```

csharp_space_before_dot

PROPERTY	[illegible]
[illegible]	csharp_space_before_dot
[illegible]	C#
[illegible]	`true` - 在点前插入空格 `false` - 删除点前的空格

代码示例:

```
// csharp_space_before_dot = true
this .Goo();

// csharp_space_before_dot = false
this.Goo();
```

csharp_space_after_semicolon_in_for_statement

PROPERTY	[
▬	csharp_space_after_semicolon_in_for_statement
▬▬	C#
▬	`true` - 在 `for` 语句中的每个分号后面插入空格 `false` - 删除 `for` 语句中每个分号后的空格

代码示例:

```
// csharp_space_after_semicolon_in_for_statement = true
for (int i = 0; i < x.Length; i++)

// csharp_space_after_semicolon_in_for_statement = false
for (int i = 0;i < x.Length;i++)
```

csharp_space_before_semicolon_in_for_statement

PROPERTY	[
▬	csharp_space_before_semicolon_in_for_statement
▬▬	C#
▬	`true` - 在 `for` 语句中的每个分号前插入空格 `false` - 删除 `for` 语句中每个分号前的空格

代码示例:

```
// csharp_space_before_semicolon_in_for_statement = true
for (int i = 0 ; i < x.Length ; i++)

// csharp_space_before_semicolon_in_for_statement = false
for (int i = 0; i < x.Length; i++)
```

csharp_space_around_declaration_statements

PROPERTY	[
▬	csharp_space_around_declaration_statements
▬▬	C#
▬	`ignore` - 不删除声明语句中多余的空格字符 `false` - 删除声明语句中多余的空格字符

代码示例:

```
// csharp_space_around_declaration_statements = ignore
int    x    =    0   ;

// csharp_space_around_declaration_statements = false
int x = 0;
```

csharp_space_before_open_square_brackets

PROPERTY	[
▬	csharp_space_before_open_square_brackets
▬	C#
▬	`true` - 在左方括号 `[` 前插入空格 `false` - 删除左方括号 `[` 前的空格

代码示例:

```
// csharp_space_before_open_square_brackets = true
int [] numbers = new int [] { 1, 2, 3, 4, 5 };

// csharp_space_before_open_square_brackets = false
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
```

csharp_space_between_empty_square_brackets

PROPERTY	[
▬	csharp_space_between_empty_square_brackets
▬	C#
▬	`true` - 在空方括号 `[ ]` 之间插入空格 `false` - 删除空方括号 `[]` 之间的空格

代码示例:

```
// csharp_space_between_empty_square_brackets = true
int[ ] numbers = new int[ ] { 1, 2, 3, 4, 5 };

// csharp_space_between_empty_square_brackets = false
int[] numbers = new int[] { 1, 2, 3, 4, 5 };
```

csharp_space_between_square_brackets

PROPERTY	[
▬	csharp_space_between_square_brackets
▬	C#

PROPERTY	[
■	`true` - 在非空方括号 `[ 0 ]` 中插入空格字符 `false` - 删除非空方括号 `[0]` 中的空格字符

代码示例：

```
// csharp_space_between_square_brackets = true
int index = numbers[ 0 ];

// csharp_space_between_square_brackets = false
int index = numbers[0];
```

换行选项

这些格式设置规则与语句和代码块中单一行以及单独的行的使用有关。

.editorconfig 文件示例：

```
# CSharp formatting rules:
[*.cs]
csharp_preserve_single_line_statements = true
csharp_preserve_single_line_blocks = true
```

csharp_preserve_single_line_statements

PROPERTY	[
■	csharp_preserve_single_line_statements
■	C#
■	Visual Studio 2017 版本 15.3
■	`true` - 将语句和成员声明保留在同一行上 `false` - 将语句和成员声明保留在不同行上

代码示例：

```
//csharp_preserve_single_line_statements = true
int i = 0; string name = "John";

//csharp_preserve_single_line_statements = false
int i = 0;
string name = "John";
```

csharp_preserve_single_line_blocks

PROPERTY	[
■	csharp_preserve_single_line_blocks
■	C#

PROPERTY	[illegible]
[illegible]	Visual Studio 2017 版本 15.3
[illegible]	`true` - 将代码块保留在单个行上 `false` - 将代码块保留在单独的行上

代码示例：

```
//csharp_preserve_single_line_blocks = true
public int Foo { get; set; }

//csharp_preserve_single_line_blocks = false
public int MyProperty
{
    get; set;
}
```

using 指令选项

此格式设置规则涉及到使用放置在命名空间内和外的 using 指令。

.editorconfig 文件示例：

```
# 'using' directive preferences
[*.cs]
csharp_using_directive_placement = outside_namespace
csharp_using_directive_placement = inside_namespace
```

csharp_using_directive_placement

PROPERTY	[illegible]
[illegible]	csharp_using_directive_placement
[illegible]	C#
[illegible]	Visual Studio 2019 版本 16.1
[illegible]	`outside_namespace` - 将 using 指令保留在命名空间之外 `inside_namespace` - 将 using 指令保留在命名空间之内

代码示例：

```
// csharp_using_directive_placement = outside_namespace
using System;

namespace Conventions
{

}

// csharp_using_directive_placement = inside_namespace
namespace Conventions
{
    using System;
}
```

命名空间选项

这些格式设置规则与命名空间声明的样式和命名约束有关。

.editorconfig 文件示例：

```
# CSharp formatting rules:
[*.cs]
csharp_style_namespace_declarations = file_scoped
```

csharp_style_namespace_declarations

[illegible]	[illegible]
[illegible]	csharp_style_namespace_declarations
[illegible]	C#
[illegible]	Visual Studio 2019 版本 16.10
[illegible]	`block_scoped` - 命名空间声明应使用块范围进行声明。 `file_scoped` - 命名空间声明应该是文件范围内的。有关详细信息，请参阅文件范围内的命名空间规范。

```
// csharp_style_namespace_declarations = block_scoped
using System;

namespace Convention
{
    class C
    {
    }
}

// csharp_style_namespace_declarations = file_scoped
using System;

namespace Convention;
class C
{
}
```

项目项

`dotnet_style_namespace_match_folder` 要求分析器有权访问项目属性才能正常运行。

对于面向 .NET Core 3.1 或更低版本的项目，必须将以下项手动添加到项目文件中。（自动为 .NET 5 及更高版本添加这些项。）

```
<ItemGroup>
  <CompilerVisibleProperty Include="RootNamespace" />
  <CompilerVisibleProperty Include="ProjectDir" />
</ItemGroup>
```

请参阅

- 语言规则
- 命名规则
- .NET 代码样式规则参考

代码样式命名规则

2021/11/16 •

在 `.editorconfig` 文件中，可以定义命名规则，用于指定并强制执行为 .NET 编程语言代码元素—如类、属性和方法—命名的方式。例如，可以指定公共成员必须采用大写形式，或者私有字段必须以 `_` 开头。

命名规则有三个组件：

- 规则适用的符号组，例如，公共成员或私有字段。
- 要与规则关联的命名样式，例如，名称必须采用大写形式或以下划线开头。
- 用于强制执行约定的严重性。

首先，必须指定符号组和命名样式，并为它们分别指定一个标题。然后指定命名规则，以将所有指定的内容链接在一起。

一般语法

若要定义命名规则、符号组或命名样式，请使用以下语法设置一个或多个属性：

```
<kind>.<title>.<propertyName> = <propertyValue>
```

每个属性只能设置一次，但某些设置允许多个值（以逗号分隔）。

属性的顺序并不重要。

<kind>

<kind> 指定所定义的元素种类—命名规则、符号组或命名样式—并且必须是以下种类之一：

[illegible]	[illegible] <KIND> [illegible]	[illegible]
命名规则	`dotnet_naming_rule`	`dotnet_naming_rule.types_should_be_pascal_case.se = suggestion`
符号组	`dotnet_naming_symbols`	`dotnet_naming_symbols.interface.applicable_kinds = interface`
命名样式	`dotnet_naming_style`	`dotnet_naming_style.pascal_case.capitalization = pascal_case`

每种定义类型—命名规则、符号组或命名样式—都有各自支持的属性，如以下各节所述。

<title>

<title> 是所选的描述性名称，用于将多个属性设置关联到一个定义中。例如，以下属性生成两个符号组定义：`interface` 和 `types`，并为每一个定义都设置了两个属性。

```
dotnet_naming_symbols.interface.applicable_kinds = interface
dotnet_naming_symbols.interface.applicable_accessibilities = public, internal, private, protected,
protected_internal, private_protected

dotnet_naming_symbols.types.applicable_kinds = class, struct, interface, enum, delegate
dotnet_naming_symbols.types.applicable_accessibilities = public, internal, private, protected,
protected_internal, private_protected
```

命名规则属性

要使规则生效，必须具有所有命名规则属性。

`symbols`	符号组的标题；命名规则将应用于此组中的符号
`style`	应与此规则关联的命名样式的标题
`severity`	设置用于强制执行命名规则的严重性。将关联的值设置为任一可用的严重性级别.[1]

注意：

1. 只有 Visual Studio 之类的开发 IDE 会遵循命名规则中的严重性规范。C# 或 VB 编译器无法解读此设置，因此在生成期间不会遵循它。若要在生成时强制执行命名样式规则，应改为通过使用代码规则严重性配置来设置严重性。有关详细信息，请参阅此 GitHub 问题。

符号组属性

你可以为符号组设置以下属性，以限制组中包含的符号。若要为单个属性指定多个值，请用逗号分隔这些值。

`applicable_kinds`	组中符号的种类 [1]	`*`（使用此值可指定所有符号）
		`namespace`
		`class`
		`struct`
		`interface`
		`enum`
		`property`
		`method`
		`field`
		`event`
		`delegate`
		`parameter`
		`type_parameter`
		`local`
		`local_function`
			是

`applicable_accessibilities`	组中符号的可访问性级别	`*`（使用此值可指定所有可访问性级别）
		`public`
		`internal` 或 `friend`
		`private`
		`protected`
		`protected_internal` 或 `protected_friend`
		`private_protected`
		`local`（用于方法内定义的符号）
			是

`required_modifiers`	仅将符号与所有指定的修饰符 [2] 进行匹配	`abstract` 或 `must_inherit`
		`async`
		`const`
		`readonly`
		`static` 或 `shared` [3]
			否

注意：

1. 目前 `applicable_kinds` 不支持元组成员。
2. 符号组与 `required_modifiers` 属性中的所有修饰符匹配。如果你忽略此属性，则无需与任何特定修饰符进行匹配。这意味着符号的修饰符不会影响是否应用此规则。

3. 如果组的 `required_modifiers` 属性包含 `static` 或 `shared`，那么组还将包括 `const` 符号，因为它们是隐式 `static` / `Shared`。不过，如果你不希望将 `static` 命名规则应用于 `const` 符号，可以使用 `const` 符号组创建新的命名规则。
4. `class` 包括 C# 记录。

命名样式属性

命名样式定义要通过规则强制执行的约定。例如：

- 采用 `PascalCase` 大写形式
- 以 `m_` 开头
- 以 `_g` 结尾
- 用 `__` 分隔单词

可以为命名样式设置以下属性：

`capitalization`	符号内的单词的大写样式	`pascal_case`
		`camel_case`
		`first_word_upper`
		`all_upper`
		`all_lower`
			是[1]

`required_prefix`	必须以这些字符开头		否
`required_suffix`	必须以这些字符结尾		否
`word_separator`	符号内的单词必须用此字符分隔		否

注意：

1. 必须在命名样式中指定大写样式，否则会忽略命名样式。

规则顺序

EditorConfig 文件中定义命名规则的顺序并不重要。命名规则根据规则本身的定义自动排序。EditorConfig 语言服务扩展可以分析 EditorConfig 文件，如果文件中的规则顺序与编译器在运行时使用的规则不同，该扩展还会进行报告。

> **NOTE**
> 如果你使用的是 Visual Studio 2019 版本16.2 之前的 Visual Studio 版本，EditorConfig 文件中的命名规则应按照从特定性最强到特定性最弱的顺序排序。遇到的第一个可应用规则是唯一应用的规则。但是，如果有多个具有相同名称的规则属性，则最近找到的具有该名称的属性具有优先权。有关详细信息，请参阅文件层次结构和优先级。

默认命名样式

如果不指定任何自定义命名规则，系统将使用下列默认样式：

- 对于具有任意辅助功能的类、结构、枚举、属性、方法以及事件，默认的命名样式为帕斯卡拼写法。
- 对于具有任意辅助功能的接口，默认的命名样式为帕斯卡拼写法并必须附加 I 前缀。

代码规则 ID: `IDE1006 (Naming rule violation)`

所有命名选项都具有规则 ID `IDE1006` 和标题 `Naming rule violation`。可以使用以下语法在 EditorConfig 文件中以全局方式配置命名违规行为的严重性：

```
dotnet_diagnostic.IDE1006.severity = <severity value>
```

严重性值必须是 `warning` 或 `error` 才能在生成时强制执行。要了解所有可能的严重性值，请参阅严重性级别。

示例

以下 .editorconfig 文件包含命名约定，该约定指定公共属性、方法、字段、事件和委托必须采用大写形式 。请注意，此命名约定指定了多种应用规则的符号，以逗号分隔。

```
[*.{cs,vb}]

# Defining the 'public_symbols' symbol group
dotnet_naming_symbols.public_symbols.applicable_kinds           = property,method,field,event,delegate
dotnet_naming_symbols.public_symbols.applicable_accessibilities = public
dotnet_naming_symbols.public_symbols.required_modifiers         = readonly

# Defining the `first_word_upper_case_style` naming style
dotnet_naming_style.first_word_upper_case_style.capitalization = first_word_upper

# Defining the `public_members_must_be_capitalized` naming rule, by setting the symbol group to the 'public
symbols' symbol group,
dotnet_naming_rule.public_members_must_be_capitalized.symbols   = public_symbols
# setting the naming style to the `first_word_upper_case_style` naming style,
dotnet_naming_rule.public_members_must_be_capitalized.style    = first_word_upper_case_style
# and setting the severity.
dotnet_naming_rule.public_members_must_be_capitalized.severity = suggestion
```

请参阅

- 语言规则
- 格式设置规则
- Roslyn 命名规则
- .NET 代码样式规则参考

平台兼容性分析器

2021/11/16 •

你可能听说过"一个 .NET"的格言：一个统一的平台，用于生成任何类型的应用程序。.NET 5.0 SDK 包括 ASP.NET Core、Entity Framework Core、WinForms、WPF、Xamarin 和 ML.NET，并且随着时间的推移，将添加对更多平台的支持。.NET 5.0 旨在提供一种无需推出不同 .NET 风格，但不会尝试完全抽象出基础操作系统 (OS) 的体验。你将继续能够调用特定于平台的 API，例如 P/Invoke、WinRT 或适用于 iOS 和 Android 的 Xamarin 绑定。

但在组件上使用特定于平台的 API 意味着代码在所有平台上都不再有效。我们需要一种在设计时进行检测的方法，使开发人员在无意中使用特定于平台的 API 时获得诊断。为了实现此目标，.NET 5.0 引入了平台兼容性分析器和补充 API，帮助开发人员根据需要识别和使用特定于平台的 API。

新的 API 包括：

- SupportedOSPlatformAttribute 用于将 API 批注为特定于平台，UnsupportedOSPlatformAttribute 用于将 API 批注为在特定 OS 上不受支持。这些属性可以选择包括版本号，并且已应用于核心 .NET 库中的某些特定于平台的 API。
- System.OperatingSystem 类中的 `Is<Platform>()` 和 `Is<Platform>VersionAtLeast(int major, int minor = 0, int build = 0, int revision = 0)` 静态方法，用于安全地调用特定于平台的 API。例如，OperatingSystem.IsWindows() 可用于保护对特定于 Windows 的 API 的调用，OperatingSystem.IsWindowsVersionAtLeast() 可用于保护受版本控制的特定于 Windows 的 API 调用。请参阅这些示例，了解如何使用这些方法保护特定于平台的 API 引用。

> **TIP**
> 平台兼容性分析器升级并替换 .NET API 分析器的发现跨平台问题。

先决条件

平台兼容性分析器是 Roslyn 代码质量分析器之一。从 .NET 5.0 开始，这些分析器包含在 .NET SDK 中。默认情况下，仅为面向 `net5.0` 或更高版本的项目启用平台兼容性分析器。但是，可以为面向其他框架的项目启用该分析器。

分析器如何确定平台依赖关系

- 无归属的 API 被视为适用于所有 OS 平台。
- 标记为 `[SupportedOSPlatform("platform")]` 的 API 被视为仅可移植到指定 OS `platform`。
  - 可以多次应用该属性，以指示多个平台支持 (`[SupportedOSPlatform("windows"), SupportedOSPlatform("Android6.0")]`)。
  - 如果在没有正确的平台上下文的情况下引用特定于平台的 API，则分析器将生成警告：
    - 如果项目不面向受支持的平台（例如，特定于 Windows 的 API 调用，且项目面向 `<TargetFramework>net5.0-ios14.0</TargetFramework>`），则将生成警告。
    - 如果项目是多定向的 (`<TargetFramework>net5.0</TargetFramework>`)，则将生成警告。
    - 如果在面向任何指定平台的项目中引用特定于平台的 API（例如，对于特定于 Windows 的 API 调用，且项目面向 `<TargetFramework>net5.0-windows</TargetFramework>`），则不会生成警告。
    - 如果特定于平台的 API 调用受到相应的平台检查方法（如 `if(OperatingSystem.IsWindows())`）的保护，则不会生成警告。
    - 如果在相同的特定于平台的上下文（使用 `[SupportedOSPlatform("platform")]` 属性化的调用站

点）中引用特定于平台的 API，则不会生成警告。

- 标记为 `[UnsupportedOSPlatform("platform")]` 的 API 被视为仅在指定的 OS `platform` 上不受支持，但在所有其他平台上受支持。
  - 可以使用不同平台（例如 `[UnsupportedOSPlatform("iOS"), UnsupportedOSPlatform("Android6.0")]`）多次应用属性。
  - 仅当 `platform` 对调用站点有效时，分析器才会生成警告：
    - 如果项目面向被属性化为不受支持的平台（例如，如果 API 使用 `[UnsupportedOSPlatform("windows")]` 进行了属性化，且调用站点面向 `<TargetFramework>net5.0-windows</TargetFramework>`），则将生成警告。
    - 如果项目是多定向的，且 `platform` 包含在默认 MSBuild `<SupportedPlatform>` 项组中，或者 `platform` 手动包含在 `MSBuild` <SupportedPlatform> 项组中，则将生成警告：

      ```
      <ItemGroup>
          <SupportedPlatform Include="platform" />
      </ItemGroup>
      ```

    - 如果要生成的应用不面向不受支持的平台或是多定向的，并且平台未包含在默认 MSBuild `<SupportedPlatform>` 项组中，则不会生成警告。

- 可以使用或不使用作为平台名称一部分的版本号对两个属性进行实例化。
  - 版本号的格式为 `major.minor[.build[.revision]]`；`major.minor` 是必需的，而 `build` 和 `revision` 部分是可选的。例如，“Windows7.0”指示 Windows 版本 7.0，但“Windows”被解释为 Windows 0.0。

有关详细信息，请参阅属性的工作方式及其导致的诊断的示例。

组合属性的高级方案

- 如果存在 `[SupportedOSPlatform]` 和 `[UnsupportedOSPlatform]` 属性的组合，则所有属性都按 OS 平台标识符分组：
  - 仅受支持的列表。如果每个 OS 平台的最低版本是 `[SupportedOSPlatform]` 属性，则 API 会被视为仅在列出的平台上受支持，但在所有其他平台上不受支持。每个平台的可选 `[UnsupportedOSPlatform]` 属性只能具有较高版本的最低支持版本，这表示从指定的版本开始删除 API。

    ```
    // The API only supported on Windows 8.0 and later, not supported for all other.
    // The API is removed/unsupported from version 10.0.19041.0.
    [SupportedOSPlatform("windows8.0")]
    [UnsupportedOSPlatform("windows10.0.19041.0")]
    public void ApiSupportedFromWindows80SupportFromCertainVersion();
    ```

  - 仅不受支持的列表。如果每个 OS 平台的最低版本是 `[UnsupportedOSPlatform]` 属性，则 API 会被视为仅在列出的平台上不受支持，但在所有其他平台上受支持。此列表可能具有包含相同平台但版本较高的 `[SupportedOSPlatform]` 属性，这表示从该版本开始支持 API。

    ```
    // The API was unsupported on Windows until version 10.0.19041.0.
    // The API is considered supported everywhere else without constraints.
    [UnsupportedOSPlatform("windows")]
    [SupportedOSPlatform("windows10.0.19041.0")]
    public void ApiSupportedFromWindows8UnsupportFromWindows10();
    ```

  - 不一致的列表。如果某些平台的最低版本为 `[SupportedOSPlatform]`，而其他平台的最低版本为 `[UnsupportedOSPlatform]`，则会被视为不一致，不受分析器支持。

- 如果 `[SupportedOSPlatform]` 和 `[UnsupportedOSPlatform]` 属性的最低版本相同，则分析器会将平台视为"仅受支持的列表"的一部分。

- 平台属性可应用于类型、成员（方法、字段、属性和事件）以及具有不同平台名称或版本的程序集。
  - 在顶级 `target` 应用的属性会影响其所有成员和类型。
  - 仅当遵守规则"子批注可以缩小平台支持范围，但无法将其扩大"时才会应用子级属性。
    - 当父级具有仅受支持的列表时，子成员属性无法添加新的平台支持，因为这会扩大父级支持。只能将新平台支持添加到父级本身。但对于具有更高版本的同一平台，子级可以有 `Supported` 属性，因为这会缩小支持。另外，子级可以有同一平台的 `Unsupported` 属性，因为这也会缩小父级支持。
    - 当父级有仅限不支持的列表时，子成员属性可以添加对新平台的支持，因为这会缩小父级支持。但它不能具有与父级所在平台相同的 `Supported` 属性，因为这会扩大父级支持。只能将对同一平台的支持添加到应用了原始 `Unsupported` 属性的父级。
  - 如果对具有相同 `platform` 名称的 API 应用 `[SupportedOSPlatform("platformVersion")]` 一次以上，则分析器仅考虑最低版本的 API。
  - 如果对具有相同 `platform` 名称的 API 应用 `[UnsupportedOSPlatform("platformVersion")]` 两次以上，则分析器仅考虑最早版本的两个 API。

> **NOTE**
> 最初受支持但在更高版本中不受支持（删除）的 API 并不希望在更高版本中重新受支持。

属性的工作方式及其导致的诊断的示例

```
// An API supported only on Windows.
[SupportedOSPlatform("Windows")]
public void WindowsOnlyApi() { }

// an API supported on Windows and Linux.
[SupportedOSPlatform("Windows")]
[SupportedOSPlatform("Linux")]
public void SupportedOnWindowsAndLinuxOnly() { }

// an API only supported on Windows 8.0 and later, not supported for all other.
// an API is removed/unsupported from version 10.0.19041.0.
[SupportedOSPlatform("windows8.0")]
[UnsupportedOSPlatform("windows10.0.19041.0")]
public void ApiSupportedFromWindows8UnsupportFromWindows10() { }

// an Assembly supported on Windows, the API added from version 10.0.19041.0.
[assembly: SupportedOSPlatform("Windows")]
[SupportedOSPlatform("windows10.0.19041.0")]
public void AssemblySupportedOnWindowsApiSupportedFromWindows10() { }

public void Caller()
{
    WindowsOnlyApi(); // warns: 'WindowsOnlyApi' is supported on 'windows'

    // warns: 'SupportedOnWindowsAndLinuxOnly' is supported on 'Windows'
    // warns: 'SupportedOnWindowsAndLinuxOnly' is supported on 'Linux'
    SupportedOnWindowsAndLinuxOnly();

    // warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is supported on 'windows' 8.0 and later
    // warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is unsupported on 'windows' 10.0.19041.0 and
later
    ApiSupportedFromWindows8UnsupportFromWindows10();

    // warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is supported on 'windows' 10.0.19041.0 and
later
    // for same platform analyzer only warn for the latest version.
```

```
    AssemblySupportedOnWindowsApiSupportedFromWindows10();
}

// an API not supported on android but supported on all other.
[UnsupportedOSPlatform("android")]
public void DoesNotWorkOnAndroid() { }

// an API was unsupported on Windows until version 8.0.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows8.0")]
public void StartedWindowsSupportFromVersion8() { }

// an API was unsupported on Windows until version8.0.
// Then the API is removed (unsupported) from version 10.0.19041.0.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows8.0")]
[UnsupportedOSPlatform("windows10.0.19041.0")]
public void StartedWindowsSupportFrom8UnsupportedFrom10() { }

public void Caller2()
{
    DoesNotWorkOnAndroid(); // warns 'DoesNotWorkOnAndroid' is unsupported on 'android'

    // warns:'StartedWindowsSupportFromVersion8' is unsupported on 'windows'
    // warns:'StartedWindowsSupportFromVersion8' is supported on 'windows' 8.0 and later
    StartedWindowsSupportFromVersion8();

    // warns:'StartedWindowsSupportFrom8UnsupportedFrom10' is unsupported on 'windows'
    // warns:'StartedWindowsSupportFrom8UnsupportedFrom10' is supported on 'windows' 8.0 and later
    // even there were 3 diagnostics found analyzer warn only for the first 2.
    StartedWindowsSupportFrom8UnsupportedFrom10();
}
```

处理报告的警告

处理这些诊断的建议方法是确保在相应的平台上运行时仅调用特定于平台的 API。下面是可用于解决警告的选项；选择最适合你的情况的选项：

- 保护调用。可以通过在运行时有条件地调用代码来实现此目的。使用平台检查方法之一检查是否正在所需的 `Platform` 上运行，例如 `OperatingSystem.Is<Platform>()` 或 `OperatingSystem.Is<Platform>VersionAtLeast(int major, int minor = 0, int build = 0, int revision = 0)`。
- 将调用站点标记为特定于平台。还可以选择将自己的 API 标记为特定于平台，从而有效地将要求转发给调用方。将包含的方法或类型或具有相同属性的整个程序集标记为引用的依赖平台的调用。示例。
- 通过平台检查来断言调用站点。如果不希望在运行时增加额外的 `if` 语句，请使用 Debug.Assert(Boolean)。示例。
- 删除代码。通常不是你想要的，因为这意味着当 Windows 用户使用代码时将失真。对于存在跨平台替代方法的情况，更好的做法可能是在特定于平台的 API 上使用此方法。
- 禁止显示警告。通过 EditorConfig 条目或 `#pragma warning disable ca1416` 即可禁止显示警告。但是，当使用特定于平台的 API 时，如非绝对必要，请勿使用此选项。

使用保护方法保护特定于平台的 API

保护方法的平台名称应与依赖平台的调用 API 平台名称匹配。如果调用 API 的平台字符串包括版本：

- 对于 `[SupportedOSPlatform("platformVersion")]` 属性，保护方法平台 `version` 应大于或等于调用平台的 `Version`。

- 对于 `[UnsupportedOSPlatform("platformVersion")]` 属性，保护方法平台 `version` 应小于或等于调用平台的 `Version`。

```
public void CallingSupportedOnlyApis() // Allow list calls
{
    if (OperatingSystem.IsWindows())
    {
        WindowsOnlyApi(); // will not warn
    }

    if (OperatingSystem.IsLinux())
    {
        SupportedOnWindowsAndLinuxOnly(); // will not warn, within one of the supported context
    }

    // Can use &&, || logical operators to guard combined attributes
    if (OperatingSystem.IsWindowsVersionAtLeast(8) &&
!OperatingSystem.IsWindowsVersionAtLeast(10.0.19041)))
    {
        ApiSupportedFromWindows8UnsupportFromWindows10();
    }

    if (OperatingSystem.IsWindowsVersionAtLeast(10, 0, 1903))
    {
        AssemblySupportedOnWindowsApiSupportedFromWindows10(); // Only need to check latest supported
version
    }
}

public void CallingUnsupportedApis()
{
    if (!OperatingSystem.IsAndroid())
    {
        DoesNotWorkOnAndroid(); // will not warn
    }

    if (!OperatingSystem.IsWindows() || OperatingSystem.IsWindowsVersionAtLeast(8))
    {
        StartedWindowsSupportFromVersion8(); // will not warn
    }

    if (!OperatingSystem.IsWindows() || // supported all other platforms
       (OperatingSystem.IsWindowsVersionAtLeast(8) && !OperatingSystem.IsWindowsVersionAtLeast(10, 0,
1903)))
    {
        StartedWindowsSupportFrom8UnsupportedFrom10(); // will not warn
    }
}
```

- 如果需要保护面向新 OperatingSystem API 不可用的 `netstandard` 或 `netcoreapp` 的代码，则可以使用 RuntimeInformation.IsOSPlatform API 并由分析器遵守。但不如 OperatingSystem 中添加的新 API 那样优化。如果平台在 OSPlatform 结构中不受支持，则可以调用 OSPlatform.Create(String) 并传入平台名称（分析器也会遵循此名称）。

```
public void CallingSupportedOnlyApis()
{
    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        SupportedOnWindowsAndLinuxOnly(); // will not warn
    }

    if (RuntimeInformation.IsOSPlatform(OSPlatform.Create("browser")))
    {
        ApiOnlySupportedOnBrowser(); // call of browser specific API
    }
}
```

将调用站点标记为特定于平台

平台名称应与依赖平台的调用 API 匹配。如果平台字符串包括版本：

- 对于 `[SupportedOSPlatform("platformVersion")]` 属性，调用站点平台 `version` 应大于或等于调用平台的 `Version`
- 对于 `[UnsupportedOSPlatform("platformVersion")]` 属性，调用站点平台 `version` 应小于或等于调用平台的 `Version`

```
// an API supported only on Windows.
[SupportedOSPlatform("windows")]
public void WindowsOnlyApi() { }

// an API supported on Windows and Linux.
[SupportedOSPlatform("Windows")]
[SupportedOSPlatform("Linux")]
public void SupportedOnWindowsAndLinuxOnly() { }

// an API only supported on Windows 8.0 and later, not supported for all other.
// an API is removed/unsupported from version 10.0.19041.0.
[SupportedOSPlatform("windows8.0")]
[UnsupportedOSPlatform("windows10.0.19041.0")]
public void ApiSupportedFromWindows8UnsupportFromWindows10() { }

// an Assembly supported on Windows, the API added from version 10.0.19041.0.
[assembly: SupportedOSPlatform("Windows")]
[SupportedOSPlatform("windows10.0.19041.0")]
public void AssemblySupportedOnWindowsApiSupportedFromWindows10() { }

[SupportedOSPlatform("windows8.0")] // call site attributed Windows 8.0 or above.
public void Caller()
{
    WindowsOnlyApi(); // will not warn as call site is for Windows.

    // will not warn as call site is for Windows.
    SupportedOnWindowsAndLinuxOnly();

    // will not warn for the API's [SupportedOSPlatform("windows8.0")] attribute.
    // Warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is unsupported on 'windows'
10.0.19041.0 and later
    ApiSupportedFromWindows8UnsupportFromWindows10();

    // warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is supported on 'windows' 10.0.19041.0
and later
    // as the call site version is lower than calling version.
    AssemblySupportedOnWindowsApiSupportedFromWindows10();
}

[SupportedOSPlatform("windows11.0")] // call site attributed with windows 11.0 or above.
public void Caller2()
{
```

```
    // Warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is unsupported on 'windows'
10.0.19041.0 and later
    ApiSupportedFromWindows8UnsupportFromWindows10();

    // will not warn as call site version higher than calling API.
    AssemblySupportedOnWindowsApiSupportedFromWindows10();
}

[SupportedOSPlatform("windows8.0")]
[UnsupportedOSPlatform("windows10.0.19041.0")] // call site supports Windows from version 8.0 to
10.0.19041.0.
public void Caller3()
{
    // will not warn as caller has exact same attributes.
    ApiSupportedFromWindows8UnsupportFromWindows10();

    // Warns: 'ApiSupportedFromWindows8UnsupportFromWindows10' is supported on 'windows' 10.0.19041.0
and later
    // As call site stopped support from that version.
    AssemblySupportedOnWindowsApiSupportedFromWindows10();
}

// an API not supported on Android but supported on all other.
[UnsupportedOSPlatform("android")]
public void DoesNotWorkOnAndroid() { }

// an API was unsupported on Windows until version 8.0.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows8.0")]
public void StartedWindowsSupportFromVersion8() { }

// an API was unsupported on Windows until version8.0.
// Then the API is removed (unsupported) from version 10.0.19041.0.
// The API is considered supported everywhere else without constraints.
[UnsupportedOSPlatform("windows")]
[SupportedOSPlatform("windows8.0")]
[UnsupportedOSPlatform("windows10.0.19041.0")]
public void StartedWindowsSupportFrom8UnsupportedFrom10() { }

[UnsupportedOSPlatform("windows")] // Caller no support Windows for any version.
public void Caller4()
{
    DoesNotWorkOnAndroid(); // warns 'DoesNotWorkOnAndroid' is unsupported on 'Android'.

    // will not warns as the call site not support Windows at all, but supports all other.
    StartedWindowsSupportFromVersion8();

    // same, will not warns as the call site not support Windows at all, but supports all other.
    StartedWindowsSupportFrom8UnsupportedFrom10();
}

[UnsupportedOSPlatform("windows")]
[UnsupportedOSPlatform("android")] // Caller not support Windows and Android for any version.
public void Caller4()
{
    DoesNotWorkOnAndroid(); // will not warn as call site not supports Android.

    // will not warns as the call site not support Windows at all, but supports all other.
    StartedWindowsSupportFromVersion8();

    // same, will not warns as the call site not support Windows at all, but supports all other.
    StartedWindowsSupportFrom8UnsupportedFrom10();
}
```

通过平台检查来断言调用站点

平台保护示例中使用的所有条件检查也可以用作 Debug.Assert(Boolean) 的条件。

```
// An API supported only on Linux.
[SupportedOSPlatform("linux")]
public void LinuxOnlyApi() { }

public void Caller()
{
    Debug.Assert(OperatingSystem.IsLinux());

    LinuxOnlyApi(); // will not warn
}
```

另请参阅

- .NET 5 中的目标框架名称
- 批注特定于平台的 API 并检测其用法
- 在特定平台上将 API 批注为不受支持
- CA1416 平台兼容性分析器
- .NET API 分析器

.NET 可移植性分析器

2021/11/16 •

想让库支持多平台吗？ 想要了解使 .NET Framework 应用程序在 .NET Core 上运行需要花费多大的精力？ .NET 可移植性分析器是一种工具，可分析程序集并为应用程序或库提供有关缺失的 .NET API 的详细报告，以便在指定的目标 .NET 平台上实现可移植性。可移植性分析器作为 Visual Studio Extension 提供，用于分析每个项目的一个程序集；也可以作为 ApiPort 控制台应用提供，用于按指定文件或目录分析程序集。

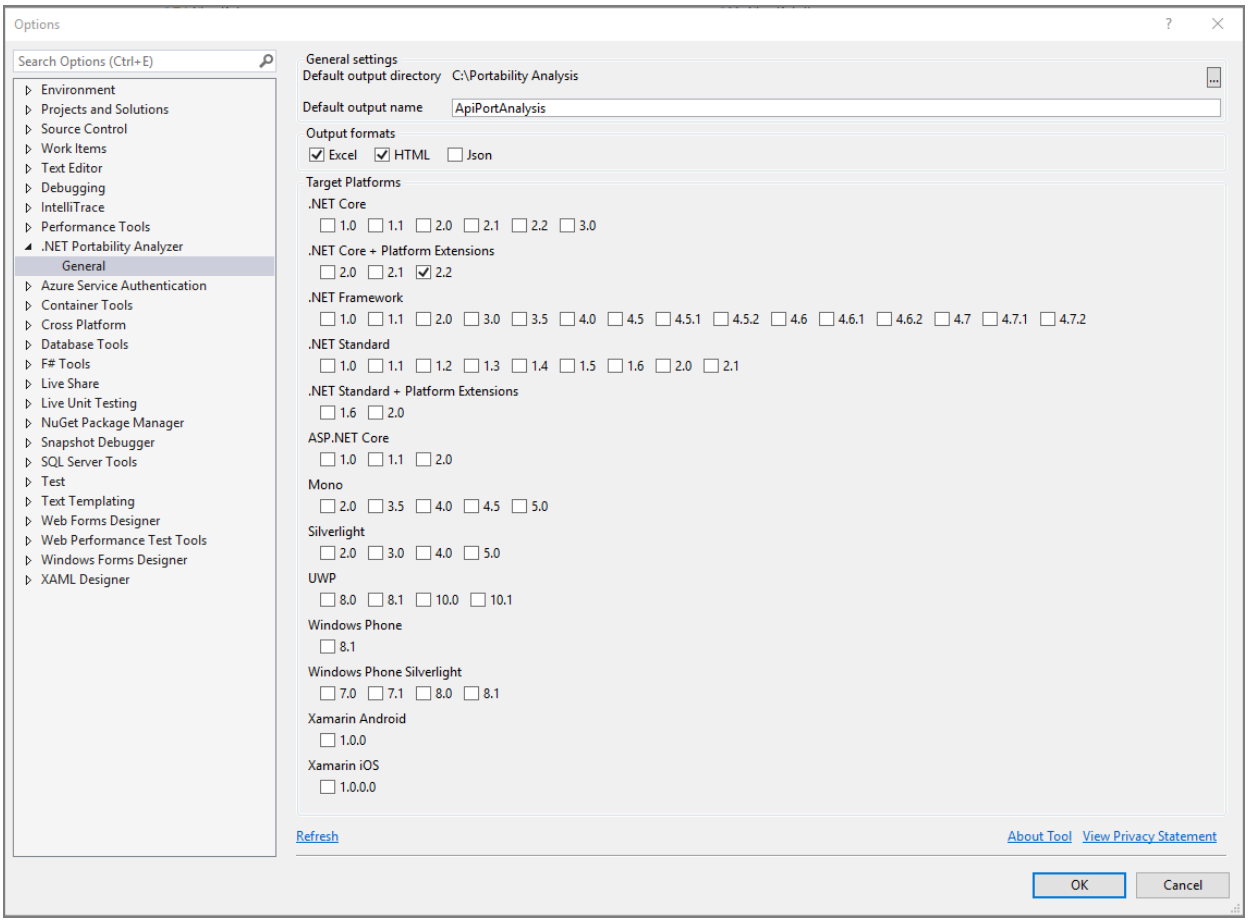
将项目转换为面向 .NET Core 等新平台后，可以使用基于 Roslyn 的 API 分析器工具来识别引发 PlatformNotSupportedException 异常以及其他兼容性问题的 API。

常用对象

- .NET Core：采用模块化设计，支持并行安装，面向跨平台方案。可并行安装意味着无需破坏其他应用即可采用新的 .NET Core 版本。如果目标是将应用移植到 .NET Core 以支持多个平台，则建议使用此对象。
- .NET Standard：包括所有 .NET 实现上提供的 .NET Standard API。如果目标是使自己的库能够在所有 .NET 支持的平台上运行，则建议使用此对象。
- ASP.NET Core：在 .NET Core 基础上构建的现代 Web 框架。如果目标是将 Web 应用移植到 .NET Core 以支持多个平台，则建议使用此对象。
- .NET Core + 平台扩展：除 Windows 兼容包之外，还包括 .NET Core API，后者提供了许多可用的 .NET Framework 技术。这是推荐的对象，用于将 Windows 上的应用从 .NET Framework 移植到 .NET Core。
- .NET Standard + 平台扩展：除 Windows 兼容包之外，还包括 .NET Standard API，后者提供了许多可用的 .NET Framework 技术。这是推荐的对象，用于将 Windows 上的库从 .NET Framework 移植到 .NET Core。

如何使用 .NET 可移植性分析器

若要开始在 Visual Studio 中使用 .NET 可移植性分析器，必须先从 Visual Studio Marketplace 下载扩展并进行安装。它适用于 Visual Studio 2017 及更高版本。可以通过 Visual Studio 中的“分析” > “可移植性分析器设置”对其进行配置，并选择目标平台，即选择 .NET 平台/版本，用于评估与当前程序集构建的平台/版本相比的可移植性差距 。



还可以使用 ApiPort 控制台应用程序，可从 ApiPort 存储库进行下载。可以使用 `listTargets` 命令选项以显示可用的目标列表，然后通过指定 `-t` 或 `--target` 命令选项来选择目标平台。

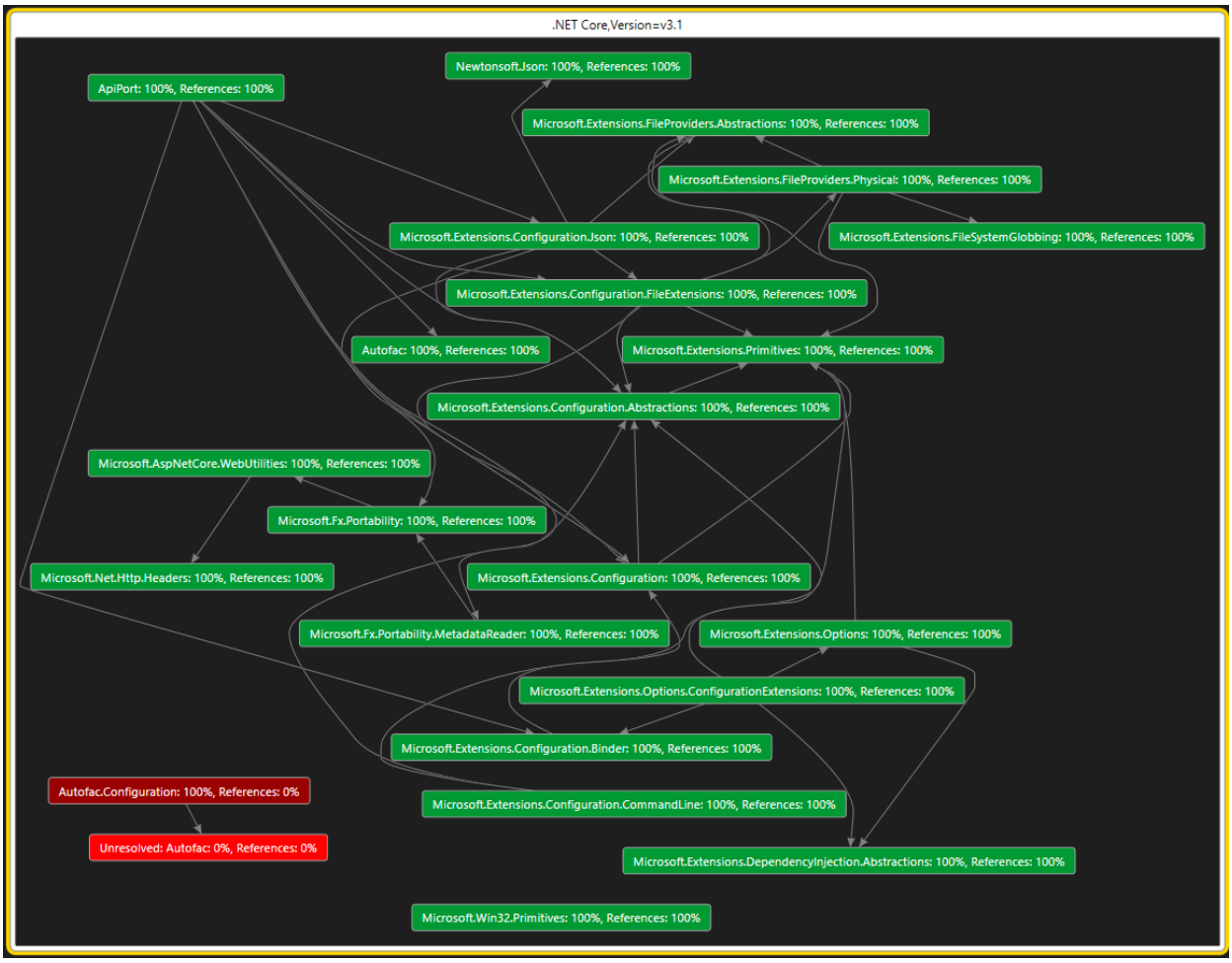
解决方案范围视图

分析包含多个项目的解决方案的一个很有用的步骤是，可视化依赖项以了解程序集中各个子集的依赖关系。一般的建议是，从依赖项关系图中的叶节点开始，以自下而上的方式应用分析结果。

要检索此项，可运行以下命令：

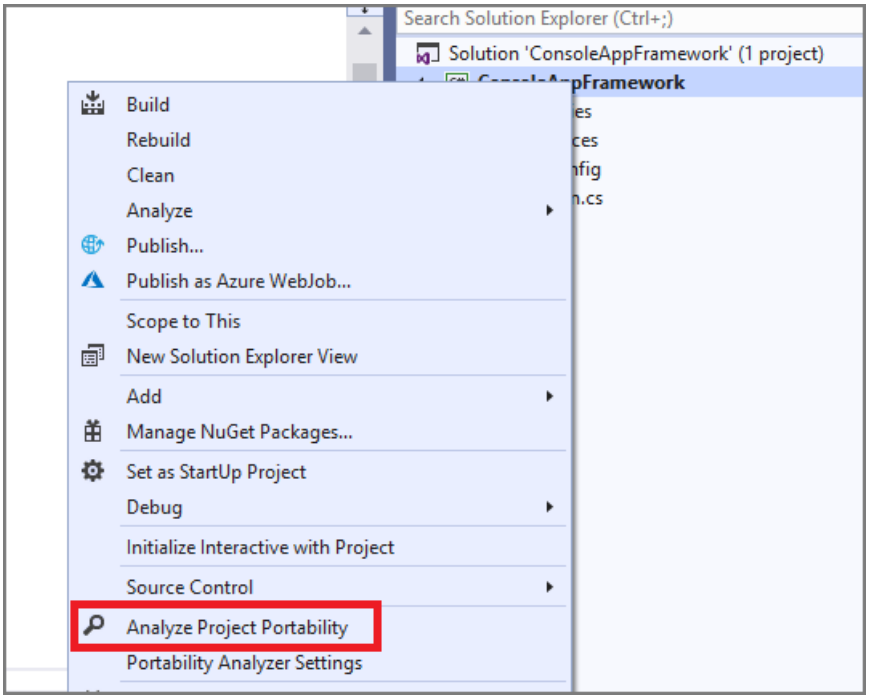
```
ApiPort.exe analyze -r DGML -f [directory or file]
```

在 Visual Studio 中打开时，此结果如下所示：



分析可移植性

若要在 Visual Studio 中分析整个项目，请在“解决方案资源管理器”中右键单击该项目，然后选择“分析程序集可移植性”。也可以转到“分析”菜单，选择“分析程序集可移植性”。在该位置选择项目的可执行文件或 DLL。



还可以使用 ApiPort 控制台应用。

- 键入以下命令即可分析当前目录：`ApiPort.exe analyze -f .`
- 若要分析特定的 .dll 文件列表，请键入以下命令：

```
ApiPort.exe analyze -f first.dll -f second.dll -f third.dll
```

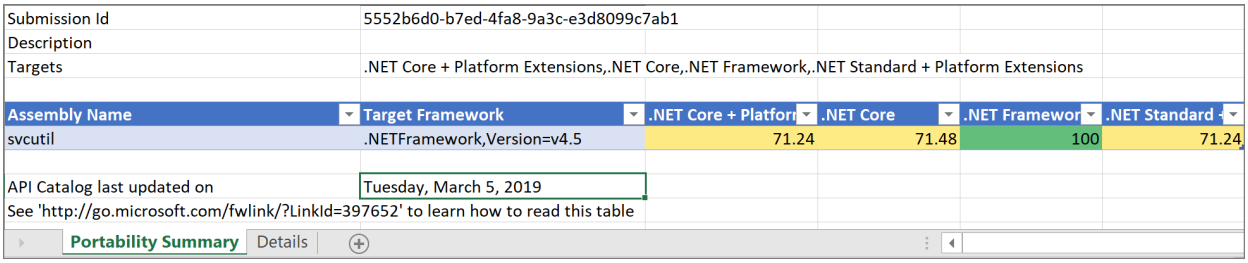
- 运行 `ApiPort.exe -?` 以获取更多帮助

建议包含自己拥有的且要移植的所有相关 exe 和 dll 文件，并且排除应用所依赖的，但你既不拥有又无法移植的文件。这将为你提供最相关的可移植性报表。

查看和解释可移植性结果

报表中仅显示目标平台不支持的 API。在 Visual Studio 中运行分析后，你将看到弹出的 .NET 可移植性报表文件链接。如果使用的是 ApiPort 控制台应用，.NET 可移植性报表将以指定的格式保存为文件。默认位于当前目录中的 Excel 文件 (.xlsx) 中。

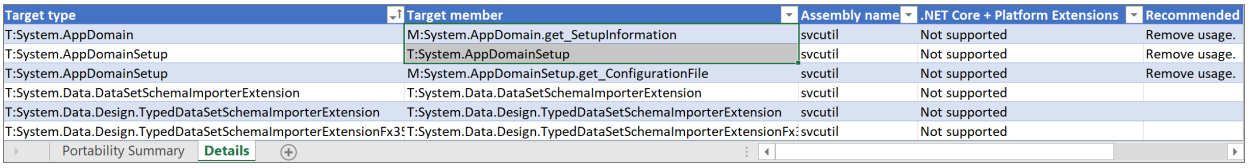
可移植性摘要



Submission Id	5552b6d0-b7ed-4fa8-9a3c-e3d8099c7ab1				
Description					
Targets	.NET Core + Platform Extensions,.NET Core,.NET Framework,.NET Standard + Platform Extensions				
Assembly Name	**Target Framework**	**.NET Core + Platfor**	**.NET Core**	**.NET Framewor**	**.NET Standard**
svcutil	.NETFramework,Version=v4.5	71.24	71.48	100	71.24
API Catalog last updated on	Tuesday, March 5, 2019				
See 'http://go.microsoft.com/fwlink/?LinkId=397652' to learn how to read this table					
Portability Summary Details					

报表的"可移植性摘要"部分显示运行中包含的每个程序集的可移植性百分比。在上述示例中，`svcutil` 应用中使用的 71.24% 的 .NET Framework API 在 .NET Core + Platform Extensions 中可用。如果针对多个程序集运行 .NET 可移植性分析器工具，则每个程序集在"可移植性摘要"报表中都应有一行。

详细信息

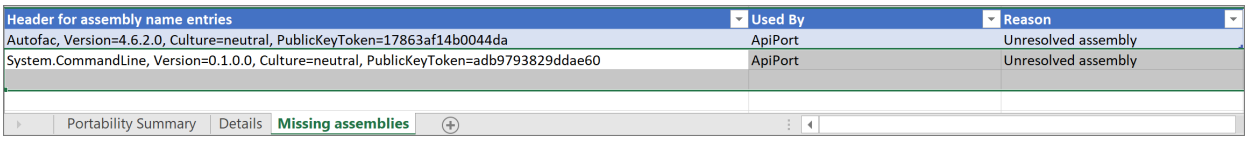


Target type	Target member	Assembly name	.NET Core + Platform Extensions	Recommended
T:System.AppDomain	M:System.AppDomain.get_SetupInformation	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	T:System.AppDomainSetup	svcutil	Not supported	Remove usage.
T:System.AppDomainSetup	M:System.AppDomainSetup.get_ConfigurationFile	svcutil	Not supported	Remove usage.
T:System.Data.DataSetSchemaImporterExtension	T:System.Data.DataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtension	T:System.Data.Design.TypedDataSetSchemaImporterExtension	svcutil	Not supported	
T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx3!	T:System.Data.Design.TypedDataSetSchemaImporterExtensionFx:	svcutil	Not supported	
Portability Summary Details				

报表的"详细信息"部分列出了任意选定目标平台缺少的 API。

- 目标类型：该类型具有目标平台缺少的 API
- 目标成员：目标平台缺少的方法
- 程序集名称：缺少的 API 所在的 .NET Framework 程序集。
- 每个选定的目标平台都是一列，例如".NET Core"："不支持"值表示此目标平台不支持 API。
- 建议的更改：要进行更改的推荐 API 或技术。对于许多 API，此字段当前为空或已过时。由于 API 数量众多，在维护 API 最新状态方面，我们面临着巨大的挑战。我们致力于提供备用解决方案，以便为客户提供有用的信息。

缺少程序集



Header for assembly name entries	Used By	Reason
Autofac, Version=4.6.2.0, Culture=neutral, PublicKeyToken=17863af14b0044da	ApiPort	Unresolved assembly
System.CommandLine, Version=0.1.0.0, Culture=neutral, PublicKeyToken=adb9793829ddae60	ApiPort	Unresolved assembly
Portability Summary Details Missing assemblies		

可以在报表中找到"缺少程序集"部分。此部分包含由你的经过分析的程序集引用的程序集列表(此列表未经过分析)。如果它是你自己拥有的程序集，请将其包含在 API 可移植性分析器运行过程中，以便你可以获得详细的 API 级别可移植性报表。如果它是第三方库，请检查是否存在支持目标平台的更新版本，并考虑转到较新的版本。最终，此列表应该包含你的应用依赖的所有第三方程序集(其中具有支持目标平台的版本)。

有关 .NET 可移植性分析器的详细信息，请访问 GitHub 文档和简要了解 .NET 可移植性分析器第 9 频道视频。

包验证概述

2021/11/16 •

跨平台兼容性已成为 .NET 库作者的主流要求。但是，如果没有针对这些包的验证工具，它们通常就不能正常工作。这对于新兴平台来说尤其成问题，因为这些平台的使用率不够高，难以引起库作者的特别关注。

在引入包验证之前，.NET SDK 工具几乎不提供针对格式标准的多目标包的验证。例如，同时以 .NET 6 和 .NET Standard 2.0 为目标的包需要确保针对 .NET Standard 2.0 二进制文件编译的代码可以针对 .NET 6 二进制文件运行。

如果使用该更改的源继续编译而无需更改，就可以认为该更改是安全且兼容的。但是，如果未重新编译使用者，这些更改仍然会在运行时导致问题。例如，向方法添加可选参数或更改常量的值可能会导致此类兼容性问题。

借助包验证工具，库开发人员可以验证他们的包是否一致且格式是否标准。它提供以下检查：

- 验证各个版本之间是否存在中断性变更。
- 针对所有不同的特定于运行时的实现，验证包是否具有一组相同的公共 API。
- 帮助开发人员捕获任何适用性漏洞。

启用包验证

通过将 `EnablePackageValidation` 属性 设置为 `true`，可以在 .NET 项目中启用包验证。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFrameworks>netstandard2.0;net6.0</TargetFrameworks>
    <EnablePackageValidation>true</EnablePackageValidation>
  </PropertyGroup>

</Project>
```

`EnablePackageValidation` 在 `pack` 任务之后运行一系列检查。有一些额外的检查可通过设置其他 MSBuild 属性来运行。

验证程序类型

作为 `pack` 任务的一部分，可以通过三种不同的验证程序来验证包：

- 基线版本验证程序根据之前发布的稳定版包来验证库项目。
- 兼容的运行时验证程序验证特定于运行时的实现程序集彼此是否兼容以及是否与编译时程序集兼容。
- 兼容的框架验证程序验证针对一个框架编译的代码是否可以针对多目标包中的所有其他框架运行。

禁止显示兼容性错误

若要抑制有意更改的兼容性错误，请将 CompatibilitySuppressions.xml 文件添加到项目。如果从命令行生成项目，则可以通过传递 `/p:GenerateCompatibilitySuppressionFile=true` 或通过将以下属性添加到项目来自动生成此文件：`<GenerateCompatibilitySuppressionFile>true</GenerateCompatibilitySuppressionFile>`

抑制文件如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<Suppressions xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Suppression>
    <DiagnosticId>CP0002</DiagnosticId>
    <Target>M:A.B.DoStringManipulation(System.String)</Target>
    <Left>lib/netstandard2.0/A.dll</Left>
    <Right>lib/net6.0/A.dll</Right>
    <isBaseline>false</isBaseline>
  </Suppression>
</Suppressions>
```

- `DiagnosticID` 指定要抑制的错误的 ID。
- `Target` 指定代码中抑制诊断 ID 的位置
- `Left` 指定 APICompat 比较的左侧操作数。
- `Right` 指定 APICompat 比较的右侧操作数。
- `isBaseline`：若要将抑制应用于基线验证，请设置为 `true`；否则设置为 `false`。

根据基线包版本进行验证

2021/11/16 •

包验证可帮助你根据之前发布的稳定版包来验证库项目。若要启用包验证，请将 `PackageValidationBaselineVersion` 或 `PackageValidationBaselinePath` 属性添加到项目文件。

包验证可检测针对任何已发布的目标框架的任何中断性变更。它还会检测是否已删除任何目标框架支持。

例如，考虑以下情况。你正在处理 AdventureWorks.Client NuGet 包，并且想要确保不会意外进行中断性变更。你将项目配置为指示包验证工具针对早期版本的包运行 API 兼容性检查。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <PackageVersion>2.0.0</PackageVersion>
    <EnablePackageValidation>true</EnablePackageValidation>
    <PackageValidationBaselineVersion>1.0.0</PackageValidationBaselineVersion>
  </PropertyGroup>

</Project>
```

几周后，你的任务是为库添加对连接超时的支持。`Connect` 方法目前如下所示：

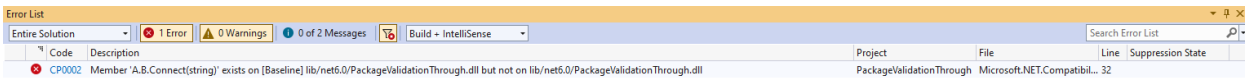
```
public static HttpClient Connect(string url)
{
    // ...
}
```

由于连接超时是一个高级配置设置，因此你认为可以添加一个可选参数：

```
public static HttpClient Connect(string url, TimeSpan timeout = default)
{
    // ...
}
```

但是，当你尝试打包时，它将引发错误。

```
D:\demo>dotnet pack
Microsoft (R) Build Engine version 17.0.0-preview-21460-01+8f208e609 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  You are using a preview version of .NET. See: https://aka.ms/dotnet-core-preview
  PackageValidationThrough -> D:\demo\bin\Debug\net6.0\PackageValidationThrough.dll
  Successfully created package 'D:\demo\bin\Debug\PackageValidationThrough.2.0.0.nupkg'.
C:\Program Files\dotnet\sdk\6.0.100-
rc.1.21463.6\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.Compatibility.Common.targets(32,5): error CP0002:
Member 'A.B.Connect(string)' exists on [Baseline] lib/net6.0/PackageValidationThrough.dll but not on
lib/net6.0/PackageValidationThrough.dll [D:\demo\PackageValidationThrough.csproj]
```



你发现虽然这不是源中断性变更，但它是二进制中断性变更。通过添加新的重载（而不是向现有方法添加参数）即可解决此问题：

```
public static HttpClient Connect(string url)
{
    return Connect(url, Timeout.InfiniteTimeSpan);
}

public static HttpClient Connect(string url, TimeSpan timeout)
{
    // ...
}
```

现在当你打包项目时，它就会成功。

验证兼容的框架

2021/11/16 •

包含兼容框架的包需要确保针对某个框架编译的代码可以针对另一个框架运行。兼容框架对的示例包括：

- .NET Standard 2.0 和 .NET 6
- .NET 5 和 .NET 6

在这两种情况下，使用者均可针对 .NET Standard 2.0 或 NET 5 构建框架并在 .NET 6 上运行。如果二进制文件在这些框架上不兼容，使用者最终可能会遇到编译时或运行时错误。

包验证将在打包时捕获这些错误。示例场景如下：

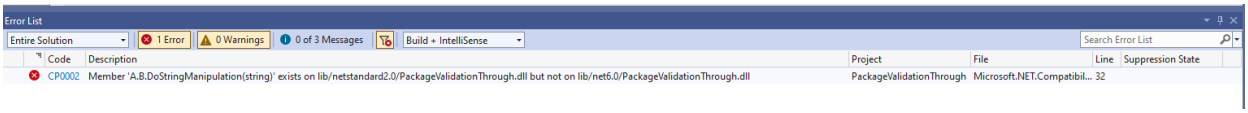
假设你正在编写一个操作字符串的游戏。需要同时支持 .NET Framework 和 .NET (.NET Core) 使用者。最初，项目面向 .NET Standard 2.0，但现在你想利用 .NET 6 中的 `Span<T>` 以避免不必要的字符串分配。为此，需要同时以 .NET Standard 2.0 和 .NET 6 为目标。

你已经编写了以下代码：

```
#if NET6_0_OR_GREATER
    public void DoStringManipulation(ReadOnlySpan<char> input)
    {
        // use spans to do string operations.
    }
#else
    public void DoStringManipulation(string input)
    {
        // Do some string operations.
    }
#endif
```

然后，你尝试(使用 `dotnet pack` 或 Visual Studio)打包项目，但失败了并显示以下错误：

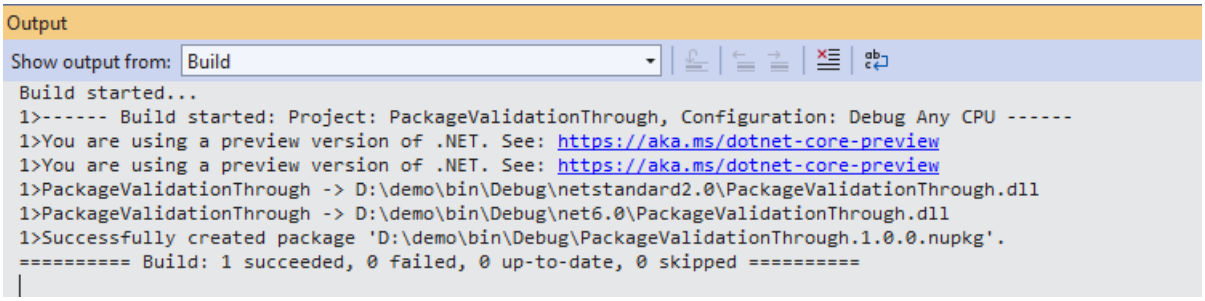
```
D:\demo>dotnet pack
Microsoft (R) Build Engine version 17.0.0-preview-21460-01+8f208e609 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  You are using a preview version of .NET. See: https://aka.ms/dotnet-core-preview
  You are using a preview version of .NET. See: https://aka.ms/dotnet-core-preview
  PackageValidationThrough -> D:\demo\bin\Debug\netstandard2.0\PackageValidationThrough.dll
  PackageValidationThrough -> D:\demo\bin\Debug\net6.0\PackageValidationThrough.dll
  Successfully created package 'D:\demo\bin\Debug\PackageValidationThrough.1.0.0.nupkg'.
C:\Program Files\dotnet\sdk\6.0.100-
rc.1.21463.6\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.Compatibility.Common.targets(32,5): error CP0002:
Member 'A.B.DoStringManipulation(string)' exists on lib/netstandard2.0/PackageValidationThrough.dll but not
on lib/net6.0/PackageValidationThrough.dll [D:\demo\PackageValidationThrough.csproj]
```



你意识到，与其为 .NET 6 排除 `DoStringManipulation(string)`，不如为 .NET 6 提供一个额外的 `DoStringManipulation(ReadOnlySpan<char>)` 方法：

```
#if NET6_0_OR_GREATER
    public void DoStringManipulation(ReadOnlySpan<char> input)
    {
        // use spans to do string operations.
    }
#endif
    public void DoStringManipulation(string input)
    {
        // Do some string operations.
    }
```

你尝试再次打包该项目，然后就成功了。



通过在项目文件中设置 `EnableStrictModeForCompatibleFrameworksInPackage` 属性为此验证程序启用“严格模式”。启用严格模式后，将更改一些规则，并在存在差异时执行一些其他规则。如果希望所比较的双方在领域和标识方面完全相同，这十分有用。

针对不同的运行时验证包

2021/11/16 •

可为 NuGet 包中的不同运行时选择不同的实现程序集。在这种情况下，需要确保这些程序集彼此兼容并与编译时程序集兼容。

例如，考虑以下情况。你正在开发一个库，该库涉及分别对 Unix 和 Windows API 的一些互操作调用。你已经编写了以下代码：

```
#if Unix
    public static void Open(string path, bool securityDescriptor)
    {
        // call unix specific stuff
    }
#else
    public static void Open(string path)
    {
        // call windows specific stuff
    }
#endif
```

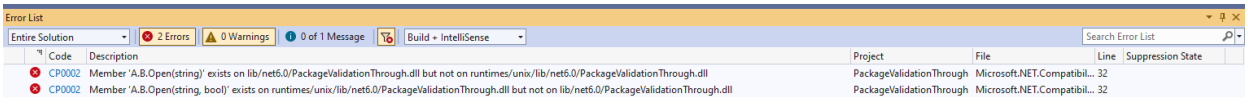
生成的包结构如下所示。

```
lib/net6.0/A.dll
runtimes/unix/lib/net6.0/A.dll
```

无论基础操作系统是哪种，都请在编译时使用 `lib\net6.0\A.dll`。对于非 Unix 系统，也将在运行时使用 `lib\net6.0\A.dll`。但是，对于 Unix 系统，将在运行时使用 `runtimes\unix\lib\net6.0\A.dll`。

当你尝试打包此项目时，会遇到以下错误：

```
D:\demo>dotnet pack
Microsoft (R) Build Engine version 17.0.0-preview-21460-01+8f208e609 for .NET
Copyright (C) Microsoft Corporation. All rights reserved.

  Determining projects to restore...
  All projects are up-to-date for restore.
  You are using a preview version of .NET. See: https://aka.ms/dotnet-core-preview
  PackageValidationThrough -> D:\demo\bin\Debug\net6.0\PackageValidationThrough.dll
  Successfully created package 'D:\demo\bin\Debug\PackageValidationThrough.1.0.0.nupkg'.
C:\Program Files\dotnet\sdk\6.0.100-
rc.1.21463.6\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.Compatibility.Common.targets(32,5): error CP0002:
Member 'A.B.Open(string)' exists on lib/net6.0/PackageValidationThrough.dll but not on
runtimes/unix/lib/net6.0/PackageValidationThrough.dll [D:\demo\PackageValidationThrough.csproj]
C:\Program Files\dotnet\sdk\6.0.100-
rc.1.21463.6\Sdks\Microsoft.NET.Sdk\targets\Microsoft.NET.Compatibility.Common.targets(32,5): error CP0002:
Member 'A.B.Open(string, bool)' exists on runtimes/unix/lib/net6.0/PackageValidationThrough.dll but not on
lib/net6.0/PackageValidationThrough.dll [D:\demo\PackageValidationThrough.csproj]
```



你发现了错误并将 `A.B.Open(string)` 添加到 Unix 运行时。

```
#if Unix
    public static void Open(string path, bool securityDescriptor)
    {
        // call unix specific stuff
    }

    public static void Open(string path)
    {
        throw new PlatformNotSupportedException();
    }
#else
    public static void Open(string path)
    {
        // call windows specific stuff
    }

    public static void Open(string path, bool securityDescriptor)
    {
        throw new PlatformNotSupportedException();
    }
#endif
```

你尝试再次打包该项目，然后就成功了。

通过在项目文件中设置 `EnableStrictModeForCompatibleTfms` 属性为此验证程序启用"严格模式"。启用严格模式后，将更改一些规则，并在存在差异时执行一些其他规则。如果希望所比较的双方在领域和标识方面完全相同，这十分有用。

包验证返回的错误代码

2021/11/16 •

本文仅供参考，其中列出了由包验证生成的所有错误代码。

错误代码列表

ID		
PKV0001	缺少兼容框架的编译时资产。	将适当的目标框架添加到项目中。
PKV0002	缺少兼容框架和运行时的运行时资产。	将相应运行时的适当资产添加到包中。
PKV0003	缺少兼容框架的运行时独立资产。	将适当的运行时独立目标框架添加到项目中。
PKV0004	缺少编译时资产的兼容运行时资产。	将适当的运行时资产添加到包中。
PKV0005	缺少编译时资产的兼容运行时资产和受支持的运行时标识符。	将适当的运行时资产添加到包中。
PKV0006	最新版本中删除了目标框架。	将适当的目标框架添加到项目中。
PKV0007	最新版本中删除了目标框架和运行时标识符对。	将适当的目标框架和 RID 添加到项目中。
CP0001	所比较的程序集中缺少该程序集外部可见的所需类型、枚举、记录或结构。	将缺少的类型添加到缺少该类型的程序集中。
CP0002	所比较的程序集中缺少在该程序集外部可见的所需成员。	将缺少的成员添加到缺少该成员的程序集中。
CP0003	程序集标识的某些部分（名称、公钥令牌、区域性、可重定目标属性或版本）对于比较的双方都不匹配。	更新程序集标识，以便比较的双方都匹配。
CP0004	创建程序集映射时，比较的其中一方找不到匹配的程序集。	确保将缺少的程序集添加到包中。
CP0005	在与非密封类型进行比较的右侧添加了抽象成员。	删除抽象成员或不要将其注释为抽象。
CP0006	将成员添加到了没有默认实现的接口。	如果目标框架和语言版本支持默认实现，请添加一个实现，或者只需从接口中删除该成员。
CP0007	类层次结构上的基类型已从相比较的其中一方中删除。	重新添加基类型（如果需要，可以在层次结构中引入新的基类型）。
CP0008	基接口已从被比较的一方的接口层次结构中删除。	将接口重新添加到层次结构。

ID		
CP0009	一方的非密封类型在另一方被注释为密封。	从类型中删除密封注释。
CP1001	在搜索目录中找不到匹配的程序集。（只有在直接使用 API 兼容性时不适用于包验证。）	当使用 `AssemblySymbolLoader` 加载匹配程序集时，提供搜索目录。
CP1002	在当前目标框架的解析目录中加载要比较的程序集时，找不到引用程序集。	使用以下 MSBuild 项添加可在其中找到该程序集的目录路径： `<PackageValidationReferencePath Include="<path>" TargetFramework="<tfm>" />` 。
CP1003	没有为包验证正在为其运行 API 兼容性的目标框架名字对象提供任何搜索目录。	使用以下 MSBuild 项提供搜索目录，以查找该目标框架的引用： `<PackageValidationReferencePath Include="<path>" TargetFramework="<tfm>" />` 。

公共语言运行时 (CLR) 概述

2021/11/16 •

.NET 提供了一个称为公共语言运行时的运行时环境，它运行代码并提供使开发过程更轻松的服务。

公共语言运行时的功能通过编译器和工具公开，你可以编写利用此托管执行环境的代码。使用面向运行时的语言编译器开发的代码称为托管代码。托管代码具有许多优点，例如：跨语言集成、跨语言异常处理、增强的安全性、版本控制和部署支持、简化的组件交互模型、调试和分析服务等。

> **NOTE**
> 编译器和工具可以产生公共语言运行时可以使用的输出，因为类型系统、元数据格式和该运行时环境（虚拟执行系统）都由公共标准（ECMA 公共语言基础结构规范）定义。有关详细信息，请参阅 ECMA C# 和公共语言基础结构规范。

若要使公共语言运行时能够向托管代码提供服务，语言编译器必须生成一些元数据来描述代码中的类型、成员和引用。元数据与代码一起存储；每个可加载的公共语言运行时可迁移执行 (PE) 文件都包含元数据。公共语言运行时使用元数据来完成以下任务：查找和加载类，在内存中安排实例，解析方法调用，生成本机代码，强制安全性，以及设置运行时上下文边界。

公共语言运行时自动处理对象布局并管理对象引用，当不再使用对象时释放它们。按这种方式实现生存期管理的对象称为托管数据。垃圾回收消除了内存泄漏以及其他一些常见的编程错误。如果你编写的代码是托管代码，则可以在 .NET 应用程序中使用托管数据、非托管数据或者同时使用这两种数据。由于语言编译器会提供自己的类型（如基元类型），因此你可能并不总是知道（或需要知道）这些数据是否是托管的。

有了公共语言运行时，就可以很容易地设计出对象能够跨语言交互的组件和应用程序。也就是说，用不同语言编写的对象可以互相通信，并且它们的行为可以紧密集成。例如，可以定义一个类，然后使用不同的语言从原始类派生出另一个类或调用原始类的方法。还可以将一个类的实例传递到用不同的语言编写的另一个类的方法。这种跨语言集成之所以成为可能，是因为基于公共语言运行时的语言编译器和工具使用由公共语言运行时定义的常规类型系统，而且它们遵循公共语言运行时关于定义新类型以及创建、使用、保持和绑定到类型的规则。

所有托管组件都带有生成它们所基于的组件和资源的信息，这些信息构成了元数据的一部分。公共语言运行时使用这些信息确保组件或应用程序具有它需要的所有内容的指定版本，这样就使代码不太可能由于某些未满足的依赖项而发生中断。注册信息和状态数据不再保存在注册表中（因为在注册表中建立和维护这些信息很困难）。取而代之的是，有关你定义的类型（及其依赖项）的信息作为元数据与代码存储在一起，这样大大降低了组件复制和移除任务的复杂性。

语言编译器和工具公开公共语言运行时的功能的方式对于开发人员来说不仅很有用，而且很直观。这意味着，公共语言运行时的某些功能可能在一个环境中比在另一个环境中更突出。你对公共语言运行时的体验取决于所使用的语言编译器或工具。例如，如果你是一位 Visual Basic 开发人员，你可能会注意到：有了公共语言运行时，Visual Basic 语言的面向对象的功能比以前多了。运行时提供如下优点：

- 性能得到了改进。
- 能够轻松使用用其他语言开发的组件。
- 类库提供的可扩展类型。
- 语言功能，如面向对象的编程的继承、接口和重载。
- 允许创建多线程的可缩放应用程序的显式自由线程处理支持。
- 结构化异常处理支持。
- 自定义特性支持。

- 垃圾回收。
- 使用委托取代函数指针，从而增强了类型安全和安全性。有关委派的详细信息，请参阅通用类型系统。

CLR 版本

.NET Core 和 .NET 5+ 版本具有一个产品版本，即没有单独的 CLR 版本。有关 .NET Core 版本的列表，请参阅下载 .NET Core。

但是，.NET Framework 版本号并非一定要与其包含的 CLR 的版本号相对应。有关 .NET Framework 版本及其相应 CLR 版本的列表，请参阅 .NET Framework 版本及依赖项。

相关主题

TITLE	
托管执行过程	描述使用公共语言运行时所需要的步骤。
自动内存管理	描述垃圾回收器如何分配和释放内存。
.NET Framework 概述	描述关键的 .NET Framework 概念，例如通用类型系统、跨语言互操作性、托管执行、应用程序域和程序集。
常规类型系统	描述在运行时中如何声明、使用和管理类型以支持跨语言集成。

托管执行过程

2021/11/16 •

托管的执行过程包括以下步骤，在本主题后面将对此进行详细讨论：

1. 选择编译器。

   若要获取公共语言运行时提供的好处，必须使用一个或多个面向运行时的语言编译器。

2. 将代码编译为 MSIL。

   编译将你的源代码转换为 Microsoft 中间语言 (MSIL) 并生成必需的元数据。

3. 将 MSIL 编译为本机代码。

   在执行时，实时 (JIT) 编译器将 MSIL 转换为本机代码。在此编译期间，代码必须通过检查 MSIL 和元数据的验证过程以查明是否可以将代码确定为类型安全。

4. 运行代码。

   公共语言运行时提供启用要发生的执行的基础结构以及执行期间可使用的服务。

选择编译器

若要获取公共语言运行时 (CLR) 提供的好处，必须使用一个或多个面向运行时的语言编译器，如 Visual Basic、C#、Visual C++、F# 或众多第三方编译器之一，如 Eiffel、Perl 或 COBOL 编译器。

因为它是一个多语言的执行环境，运行时支持各种各样的数据类型和语言功能。你使用的语言编译器确定哪些运行时功能是可用的，并且可以使用这些功能设计你的代码。编译器（而非运行时）设定代码必须使用的语法。如果组件必须完全可供以其他语言编写的组件使用，则组件的导出类型必须仅公开公共语言规范 (CLS) 中包含的语言功能。可以使用 CLSCompliantAttribute 属性来确保你的代码符合 CLS。有关详细信息，请参阅语言独立性和与语言无关的组件。

返回页首

编译为 MSIL

编译为托管代码时，编译器将源代码转换为 Microsoft 中间语言 (MSIL)，这是一组独立于 CPU 且可以有效地转换为本机代码的说明。MSIL 包括有关加载、存储、初始化和调用对象方法的说明，以及有关算术和逻辑运算、控制流、直接内存访问、异常处理和其他操作的说明。代码可以运行之前，必须将 MSIL 转换为特定于 CPU 的代码，通常通过 实时 (JIT) 编译器实现。由于公共语言运行时为其支持的每个计算机基础结构提供一个或多个 JIT 编译器，同一组的 MSIL 可以在任何受支持的基础结构上进行 JIT 编译和运行。

当编译器生成 MSIL 时，它还生成元数据。元数据描述代码中的类型，包括每种类型的定义、每种类型的成员的签名、代码引用的成员以及运行时在执行时间使用的其他数据。MSIL 和元数据包含在一个可移植的可执行 (PE) 文件中，该文件基于且扩展已发布的 Microsoft PE 和历来用于可执行内容的通用对象文件格式 (COFF)。容纳 MSIL 或本机代码以及元数据的这种文件格式使操作系统能够识别公共语言运行时映像。文件中元数据的存在以及 MSIL 使代码能够描述自身，这意味着将不需要类型库或接口定义语言 (IDL)。运行时在执行期间会根据需要从文件中查找并提取元数据。

返回页首

将 MSIL 编译为本机代码

运行 Microsoft 中间语言 (MSIL) 前，必须根据公共语言运行时将其编译为目标计算机基础结构的本机代码。.NET 提供两种方法来执行此转换：

- .NET 实时 (JIT) 编译器。
- Ngen.exe（本机映像生成器）。

由 JIT 编译器编译

在加载和执行程序集的内容时，JIT 编译在应用程序运行时按需将 MSIL 转换为本机代码。由于公共语言运行时为每个受支持的 CPU 基础结构提供 JIT 编译器，开发人员可以构建一组 MSIL 程序集，这些程序集可以进行 JIT 编译并可在具有不同计算机基础结构的不同计算机上运行。但是，如果你的托管代码调用特定于平台的本机 API 或特定于平台的类库，它将仅在该操作系统上运行。

JIT 编译将执行期间可能永远不会调用的某些代码的可能性考虑在内。它根据需要在执行期间转换 MSIL，而不是使用时间和内存来将 PE 文件中所有 MSIL 转换为本机代码，并在内存中存储生成的本机代码，以便该进程上下文中的后续调用可以对其进行访问。加载类型并将其初始化时，加载程序创建并将存根附加到类型中的每个方法。第一次调用某个方法时，存根将控件传递给 JIT 编译器，后者将该方法的 MSIL 转换为本机代码，并将存根修改为直接指向生成的本机代码。因此，对 JIT 编译的方法的后续调用会直接转到本机代码。

使用 NGen.exe 的安装时代码生成

由于在调用该程序集中定义的各个方法时，JIT 编译器将程序集的 MSIL 转换为本机代码，因此它在运行时中对性能产生负面影响。在大多数情况下，这种性能降低的程度是可以接受的。更为重要的是，由 JIT 编译器生成的代码会绑定到触发编译的进程上。它无法在多个进程之间进行共享。若要允许生成的代码跨应用程序的多个调用或跨共享一组程序集的多个进程进行共享，则公共语言运行时支持预编译模式。这种预编译模式使用 Ngen.exe（本机映像生成器）将 MSIL 程序集转换为本机代码，非常类似 JIT 编译器执行的操作。但是，Ngen.exe 的操作在三个方面不同于 JIT 编译器的操作：

- 它在运行应用程序之前而非运行该应用程序时，将 MSIL 转换为本机代码。
- 它一次编译整个程序集，而不是一次编译一种方法。
- 它将本机映像缓存中生成的代码作为磁盘上的文件保存。

代码验证

作为其编译为本机代码的一部分，MSIL 代码必须通过验证过程，除非管理员已经设定允许代码忽略验证的安全策略。验证过程检查 MSIL 和元数据，以找出代码是否为类型安全，这意味着它仅访问其有权访问的内存位置。类型安全有助于将对象相互隔离，并帮助保护它们免受无意或恶意损坏。它还保障可以可靠地对代码强制执行安全限制。

运行时基于以下语句对于可验证类型安全代码为 true 这一事实：

- 对类型的引用严格符合所引用的类型。
- 在对象上只调用正确定义的操作。
- 标识与声称的要求一致。

在验证过程中，检查 MSIL 代码以试图确认代码可以访问内存位置，并仅通过正确定义的类型调用方法。例如，代码不允许以允许内存位置溢出的方式访问对象的字段。此外，验证检查代码以确定是否已正确生成 MSIL，因为错误的 MSIL 可能会导致违反类型安全规则。验证过程传递一组定义完善的类型安全代码，并且仅传递类型安全的代码。但是，由于验证过程的一些限制，并且按照设计，某些语言不生成可验证的类型安全代码，某些类型安全代码可能无法通过验证。如果安全策略要求提供类型安全代码，而该代码不能通过验证，则在运行该代码时将引发异常。

返回页首

运行代码

公共语言运行时提供启用要发生的托管执行的基础结构以及执行期间可使用的服务。方法可以运行之前，必须编译为特定于处理器的代码。当第一次调用，然后运行时，为其生成 MSIL 的每种方法都是 JIT 编译的。下次运行该方法时，将运行现有的 JIT 编译的本机代码。重复 JIT 编译，然后运行代码的过程，直到执行完毕。

在执行期间，托管代码接收服务，如垃圾收集、安全性、与非托管代码的互操作性、跨语言调试支持以及增强的部署和版本控制支持。

在 Microsoft Windows Vista 中，操作系统加载程序通过检查 COFF 标头中的一个位检查托管模块。所设置的位表示托管模块。如果加载程序检测到托管模块，它将加载 mscoree.dll，`_CorValidateImage` 并且 `_CorImageUnloading` 在加载和卸载托管模块映像时通知加载程序。`_CorValidateImage` 执行以下操作：

1. 确保代码是有效的托管代码。

2. 将映像中的入口点更改为运行时中的入口点。

在 64 位 Windows 上，`_CorValidateImage` 通过将其从 PE32 转换为 PE32+ 格式修改内存中的映像。

返回页首

另请参阅

- 概述
- 语言独立性和与语言无关的组件
- 元数据和自描述组件
- Ilasm.exe（IL 汇编程序）
- 安全性
- 与非托管代码交互操作
- 部署
- .NET 中的程序集
- 应用程序域

.NET 中的程序集

2021/11/16 •

程序集构成了 .NET 应用程序的部署、版本控制、重用、激活范围和安全权限的基本单元。程序集是为协同工作而生成的类型和资源的集合，这些类型和资源构成了一个逻辑功能单元。程序集采用可执行文件 (.exe) 或动态链接库文件 (.dll) 的形式，是 .NET 应用程序的构建基块 。它们向公共语言运行时提供了注意类型实现代码所需的信息。

在 .NET 和 .NET Framework 中，可从一个或多个源代码文件生成程序集。在 .NET Framework 中，程序集可以包含一个或多个模块。因此，大型项目可以采用以下规划：由多个开发者单独开发各源代码文件或模块，最后整合所有这些内容以创建一个程序集。若要详细了解模块，请参阅操作说明：生成多文件程序集。

程序集具有以下属性：

- 程序集以 .exe 或 .dll 文件的形式实现 。
- 对于面向 .NET Framework 的库，可通过将程序集放入全局程序集缓存 (GAC)，在应用程序之间共享程序集。必须先对程序集进行强命名，然后才能将它们包含到 GAC 中。有关详细信息，请参阅具有强名称的程序集。
- 只有在需要使用时才会将程序集加载到内存中。如果未使用程序集，则不加载。也就是说，使用程序集，可以在大型项目中高效管理资源。
- 可以使用反射，以编程方式获取程序集的相关信息。有关详细信息，请参阅反射 (C#) 或反射 (Visual Basic)。
- 可加载一个程序集，使用 .NET 和 .NET Framework 中的 MetadataLoadContext 类来检查该程序集。MetadataLoadContext 取代了 Assembly.ReflectionOnlyLoad 方法。

公共语言运行时中的程序集

程序集向公共语言运行时提供了注意类型实现代码所需的信息。对于运行时，类型不存在于程序集上下文之外。

程序集定义以下信息：

- 公共语言运行时执行的代码。请注意，每个程序集只能有一个入口点：`DllMain`、`WinMain` 或 `Main`。
- 安全边界。程序集就是在其中请求和授予权限的单元。有关程序集中安全边界的详细信息，请参阅程序集安全注意事项。
- 类型边界。每一类型的标识均包括该类型所驻留的程序集的名称。在一个程序集的范围中加载的称为 `MyType` 的类型不同于在另一个程序集范围中加载的称为 `MyType` 的类型。
- 引用范围边界。程序集清单包含用于解析类型和满足资源请求的元数据。该清单指定要在程序集外公开的类型和资源，并枚举它所依赖的其他程序集。除非可迁移可执行 (PE) 文件中的 Microsoft 中间语言 (MSIL) 代码具有相关的程序集清单，否则不执行此代码。
- 版本边界。程序集是公共语言运行时中无版本冲突的最小单元。同一程序集中的所有类型和资源均会被版本化为一个单元。程序集清单描述你为任何依赖项程序集所指定的版本依赖性。有关版本控制的详细信息，请参阅程序集版本控制。
- 部署单元。当一个应用程序启动时，只有该应用程序最初调用的程序集必须存在。其他程序集（例如，包含本地化资源或实用工具类的程序集）可以按需检索。这样，应用在第一次下载时就会比较精简。有关部署程序集的详细信息，请参阅部署应用程序。

- 并行执行单元。有关运行多个版本的程序集的详细信息，请参阅程序集和并行执行。

创建程序集

程序集可以为静态或动态。静态程序集存储在磁盘上的可迁移可执行 (PE) 文件中。静态程序集可以包括接口、类和资源（如位图、JPEG 文件和其他资源文件）。你还可以创建动态程序集，动态程序集直接从内存运行并且在执行前不保存到磁盘上。你可以在执行动态程序集后将它们保存在磁盘上。

有几种创建程序集的方法。你可以使用可创建 .dll 或 .exe 文件的开发工具，例如 Visual Studio 。可以使用 Windows SDK 中的工具创建具有从其他开发环境中创建的模块的程序集。还可以使用公共语言运行时 API（例如 System.Reflection.Emit）来创建动态程序集。

可以采用以下方法编译程序集：在 Visual Studio 中生成程序集、使用 .NET Core 命令行接口工具生成程序集，或使用命令行编译器生成 .NET Framework 程序集。要详细了解如何使用 .NET CLI 生成程序集，请参阅 .NET CLI 概述。

> **NOTE**
> 若要在 Visual Studio 中生成程序集，请在“生成”菜单上选择“生成” 。

程序集清单

每个程序集都有一个程序集清单文件。与目录类似，程序集清单包含以下内容：

- 程序集的标识（名称和版本）。
- 文件表，描述构成程序集的其他所有文件（例如，.exe 或 .dll 文件所依赖的你创建的其他程序集、位图文件或自述文件）。
- 程序集引用列表，即所有外部依赖项的列表，如 .dll 或其他文件 。程序集既可以引用全局对象，也可以引用私有对象。全局对象可用于所有其他应用程序。在 .NET Core 中，全局对象与特定的 .NET Core 运行时结合使用。在 .NET Framework 中，全局对象位于全局程序集缓存 (GAC) 中。System.IO.dll 是 GAC 中程序集的一个示例。私有对象必须位于级别不高于应用安装目录的目录中。

由于程序集包含内容、版本控制和依赖项的相关信息，因此使用它们的应用程序不依赖 Windows 系统上的注册表等外部源也能正常运行。程序集减少了 .dll 冲突，让应用程序变得更可靠、更易于部署。在许多情况下，只需将 .NET 应用程序的文件复制到目标计算机，即可进行安装。有关详细信息，请参阅程序集清单。

添加对程序集的引用

必须添加对应用程序中的程序集的引用，才能使用该程序集。引用程序集后，应用程序可以使用其名称空间的所有可访问类型、属性、方法和其他成员，就好像它们的代码是源文件的一部分一样。

> **NOTE**
> .NET 类库中的大多数程序集都是自动引用的。如果系统程序集不是自动引用的，则通过以下方式之一添加引用：
>
> - 对于 .NET 和 .NET Core，添加对包含该程序集的 NuGet 包的引用。请使用 Visual Studio 中的 NuGet 包管理器，或者将程序集的 <PackageReference> 元素添加到 .csproj 或 .vbproj 项目 。
> - 对于 .NET Framework，可通过在 Visual Studio 中使用“添加引用”对话框，或者通过使用 C# 或 Visual Basic 编译器的 `-reference` 命令行选项，添加对该程序集的引用。

在 C# 中，可以在单个应用程序中使用同一程序集的两个版本。有关详细信息，请参阅外部别名。

相关的内容

TITLE	𝐂
程序集内容	组成程序集的元素。
程序集清单	程序集清单中的数据，以及它如何存储在程序集中。
全局程序集缓存	GAC 如何存储和使用程序集。
具有强名称的程序集	具有强名称的程序集的特征。
程序集安全注意事项	安全性如何作用于程序集。
程序集版本控制	.NET Framework 版本控制策略的概述。
程序集位置	在何处可以找到程序集。
程序集和并行执行	同时使用多个版本的运行时或程序集。
发出动态方法和程序集	如何创建动态程序集。
运行时如何定位程序集	.NET Framework 如何在运行时解析程序集引用。

参考

System.Reflection.Assembly

请参阅

- .NET 程序集文件格式
- 友元程序集
- 引用程序集
- 如何：加载和卸载程序集
- 如何：在 .NET Core 中使用和调试程序集可卸载性
- 如何：确定文件是否为程序集
- 如何：使用 MetadataLoadContext 检查程序集内容

元数据和自描述组件

2021/11/16 •

在过去，以一种语言编写的软件组件（.exe 或 .dll）不能方便地使用以另一种语言编写的软件组件。在这个问题的解决上，COM 向前迈进了一步。.NET 允许编译器向所有的模块和程序集发出附加的说明性信息，从而使组件互用更加简单。这种叫做“元数据”的信息有助于组件无缝交互。

元数据是一种二进制信息，用以对存储在公共语言运行时可迁移可执行文件 (PE) 文件或存储在内存中的程序进行描述。将您的代码编译为 PE 文件时，便会将元数据插入到该文件的一部分中，而将代码转换为 Microsoft 中间语言 (MSIL) 并将其插入到该文件的另一部分中。在模块或程序集中定义和引用的每个类型和成员都将在元数据中进行说明。当执行代码时，运行时将元数据加载到内存中，并引用它来发现有关代码的类、成员、继承等信息。

元数据以非特定语言的方式描述在代码中定义的每一类型和成员。元数据存储以下信息：

- 程序集的说明。
  - 标识（名称、版本、区域性、公钥）。
  - 导出的类型。
  - 该程序集所依赖的其他程序集。
  - 运行所需的安全权限。
- 类型的说明。
  - 名称、可见性、基类和实现的接口。
  - 成员（方法、字段、属性、事件、嵌套的类型）。
- 特性。
  - 修饰类型和成员的其他说明性元素。

元数据的优点

对于一种更简单的编程模型来说，元数据是关键，该模型不再需要接口定义语言 (IDL) 文件、头文件或任何外部组件引用方法。元数据使 .NET 语言能够自动以非特定语言的方式对其自身进行描述，而这是开发人员和用户都无法看见的。另外，通过使用特性，可以对元数据进行扩展。元数据具有以下主要优点：

- 自描述文件。

  公共语言运行时模块和程序集是自描述的。模块的元数据包含与另一个模块进行交互所需的全部信息。元数据自动提供 COM 中 IDL 的功能，因此可以将一个文件同时用于定义和实现。运行时模块和程序集甚至不需要向操作系统注册。结果，运行时使用的说明始终反映编译文件中的实际代码，从而提高应用程序的可靠性。

- 语言互用性和更简单的基于组件的设计。

  元数据提供所有必需的有关已编译代码的信息，以供你从用不同语言编写的 PE 文件中继承类。你可以创建用任何托管语言（任何面向公共语言运行时的语言）编写的任何类的实例，而不用担心显式封送处理或使用自定义的互用代码。

- 特性。

  .NET 允许在编译文件中声明特定种类的元数据（称为特性）。特性在 .NET 中处处可见，用于更精确地控制

程序在运行时的行为。另外，还可以通过用户定义的自定义特性，向 .NET 文件发出自己的自定义元数据。有关更多信息，请参阅特性。

元数据和 PE 文件结构

元数据存储在 .NET 可迁移可执行文件 (PE) 文件的一个部分中，而 Microsoft 中间语言 (MSIL) 则存储在 PE 文件的另一部分中。文件的元数据部分包含一系列的表和堆数据结构。MSIL 部分包含 MSIL 和引用 PE 文件元数据部分的元数据标记。例如，使用 MSIL 反汇编程序 (Ildasm.exe) 等工具查看代码的 MSIL 时，可能会遇到元数据标记。

元数据表和堆

每个元数据表都保留有关程序元素的信息。例如，一个元数据表说明代码中的类，另一个元数据表说明字段等。如果你的代码中有 10 个类，类表将有 10 行，每行一类。元数据表引用其他的表和堆。例如，类的元数据表引用方法表。

元数据还以四种堆结构存储信息：字符串、Blob、用户字符串和 GUID。所有用于对类型和成员进行命名的字符串都存储在字符串堆中。例如，方法表不直接存储特定方法的名称，而是指向存储在字符串堆中的方法的名称。

元数据标记

元数据标记在 PE 文件的 MSIL 部分中唯一确定每个元数据表的每一行。元数据标记在概念上和指针相似，永久驻留在 MSIL 中，引用特定的元数据表。

元数据标记是一个四个字节的数字。最高位字节表示特定标记（方法、类型等）引用的元数据表。剩下的三个字节指定与所说明的编程元素对应的元数据表中的行。如果您用 C# 定义一个方法并将其编译到 PE 文件，下面的元数据标记可能存在于 PE 文件的 MSIL 部分：

```
0x06000004
```

最高位字节 (`0x06`) 表示这是一个 MethodDef 标记。低位的三个字节 (`000004`) 指示公共语言运行时在 MethodDef 表的第四行查找对该方法定义进行描述的信息。

PE 文件中的元数据

当为公共语言运行时编译程序时，该程序转换为由三部分组成的 PE 文件。下表说明了每部分的内容。

PE 区	PE 区区区
PE 头	PE 文件主要部分的索引和入口点的地址。 运行时使用该信息确定该文件为 PE 文件并确定当将程序加载到内存时执行从何处开始。
MSIL 指令	组成代码的 Microsoft 中间语言指令 (MSIL)。许多 MSIL 指令带有元数据标记。
元数据	元数据表和堆。运行时使用该部分记录您的代码中每个类型和成员的信息。本部分还包括自定义特性和安全性信息。

元数据在运行时的作用

要更好地理解元数据和它在公共语言运行时中的作用，构造一个简单的程序并说明元数据如何影响它的运行时情况可能很有帮助。下面的代码示例显示名为 `MyApp` 的类中的两种方法。`Main` 方法是程序入口点，而 `Add` 方法只返回两个整数参数的和。

```
Public Class MyApp
   Public Shared Sub Main()
      Dim ValueOne As Integer = 10
      Dim ValueTwo As Integer = 20
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo))
   End Sub

   Public Shared Function Add(One As Integer, Two As Integer) As Integer
      Return (One + Two)
   End Function
End Class
```

```
using System;
public class MyApp
{
   public static int Main()
   {
      int ValueOne = 10;
      int ValueTwo = 20;
      Console.WriteLine("The Value is: {0}", Add(ValueOne, ValueTwo));
      return 0;
   }
   public static int Add(int One, int Two)
   {
      return (One + Two);
   }
}
```

当代码运行时，运行时将模块加载到内存并向元数据咨询该类的信息。加载后，运行时对方法的 Microsoft 中间语言 (MSIL) 流执行广泛的分析，将其转换为快速本机指令。运行时根据需要使用实时 (JIT) 编译器将 MSIL 指令转换为本机代码，每次转换一个方法。

下面的示例显示了从以前代码的 `Main` 功能生成的部分 MSIL。可使用 MSIL 反汇编程序 (Ildasm.exe) 从任何 .NET 应用程序中查看 MSIL 和元数据。

```
.entrypoint
.maxstack  3
.locals ([0] int32 ValueOne,
         [1] int32 ValueTwo,
         [2] int32 V_2,
         [3] int32 V_3)
IL_0000:  ldc.i4.s   10
IL_0002:  stloc.0
IL_0003:  ldc.i4.s   20
IL_0005:  stloc.1
IL_0006:  ldstr      "The Value is: {0}"
IL_000b:  ldloc.0
IL_000c:  ldloc.1
IL_000d:  call int32 ConsoleApplication.MyApp::Add(int32,int32) /* 06000003 */
```

JIT 编译器读取整个方法的 MSIL，对其进行彻底地分析，然后为该方法生成有效的本机指令。在 `IL_000d` 遇到 `Add` 方法 (`/*` `06000003 */`) 的元数据标记，运行时使用该标记参考 MethodDef 表的第三行 。

下表显示了说明 **方法的元数据标记所引用的 MethodDef** `Add` 表的一部分。虽然程序集中存在其他元数据表并具有它们自己唯一的值，但这里只讨论该表。

r	rrrrrr (RVA)	IMPLFLAGS	FLAGS	"rr" (rrrrrrr)	SIGNATURE(rr BLOB r)
1	0x00002050	IL Managed	Public ReuseSlot SpecialName RTSpecialName .ctor	.ctor(构造函数)	
2	0x00002058	IL Managed	Public Static ReuseSlot	Main	String
3	0x0000208c	IL Managed	Public Static ReuseSlot	添加	int, int, int

该表的每一列都包含有关代码的重要信息。RVA 列允许运行时计算定义此方法的 MSIL 的起始内存地址。ImplFlags 和 Flags 列包含说明该方法的位屏蔽(例如，该方法是公共的还是私有的)。Name 列对来自字符串堆的方法的名称进行了索引。Signature 列对在 Blob 堆中的方法签名的定义进行了索引。

运行时在第三行的 RVA 列计算所需的偏移量地址并将该地址返回到 JIT 编译器，然后，JIT 编译器进入新地址。JIT 编译器继续在新地址处理 MSIL，直到它遇到另一个元数据标记，之后，重复该过程。

使用元数据，运行时可以访问加载代码并将其处理为本机指令所需的所有信息。以这种方式，元数据使自描述文件、常规类型系统和跨语言继承成为可能。

相关主题

TITLE	rr
特性	描述如何应用特性、编写自定义特性及检索存储在特性中的信息。

.NET 中的依赖项加载

2021/11/16 •

每个 .NET 应用程序都有依赖项。即使是简单的 `hello world` 应用程序也在 .NET 类库的各个部分中有依赖项。

了解 .NET 默认程序集加载逻辑有助于排查典型的部署问题。

在某些应用程序中，在运行时动态确定依赖项。在这些情况下，了解托管程序集和非托管依赖项的加载方式至关重要。

AssemblyLoadContext

AssemblyLoadContext API 是 .NET 加载设计的核心。了解 AssemblyLoadContext 一文提供了有关该设计的概念性概述。

加载详细信息

以下几篇文章简要介绍了加载算法的详细信息：

- 托管程序集加载算法
- 附属程序集加载算法
- 非托管（本机）库加载算法
- 默认探测

使用插件创建应用

创建包含插件的 .NET Core 应用程序教程介绍了如何创建自定义 AssemblyLoadContext。它使用 AssemblyDependencyResolver 来解析插件的依赖项。该教程正确地将插件依赖项与主机应用程序隔离开来。

程序集卸载功能

如何在 .NET Core 中使用和调试可卸载性一文是逐步骤教程。其中显示了如何加载、执行和卸载 .NET Core 应用程序。该文章还提供了调试提示。

收集详细的程序集加载信息

收集详细的程序集加载信息一文介绍了如何收集运行时中托管程序集加载的详细信息。它使用 dotnet-trace 工具在正在运行的进程的跟踪中捕获程序集加载程序事件。

了解 System.Runtime.Loader.AssemblyLoadContext

2021/11/16 •

AssemblyLoadContext 类是在 .NET Core 中引入的，在 .NET Framework 中不可用。本文使用概念性信息来补充 AssemblyLoadContext API 文档。

本文与实现动态加载的开发人员有关，尤其是动态加载框架的开发人员。

什么是 AssemblyLoadContext？

每个 .NET 5+ 和 .NET Core 应用程序均隐式使用 AssemblyLoadContext。它是运行时的提供程序，用于定位和加载依赖项。只要加载了依赖项，就会调用 AssemblyLoadContext 实例来定位该依赖项。

- 它提供定位、加载和缓存托管程序集和其他依赖项的服务。
- 为了支持动态代码加载和卸载，它创建了一个独立上下文，用于在其自己的 AssemblyLoadContext 实例中加载代码及其依赖项。

何时需要多个 AssemblyLoadContext 实例？

单个 AssemblyLoadContext 实例限制为每个简单程序集名称 AssemblyName.Name 只加载 Assembly 的一个版本。

当动态加载代码模块时，此限制可能会成为一个问题。每个模块都是独立编译的，并且可能依赖于不同版本的 Assembly。当不同的模块依赖于常用库的不同版本时，通常会出现此问题。

为了支持动态加载代码，AssemblyLoadContext API 提供在同一个应用程序中加载 Assembly 冲突版本的功能。每个 AssemblyLoadContext 实例提供一个唯一字典，该字典将每个 AssemblyName.Name 映射到特定的 Assembly 实例。

它还提供了一种方便的机制，将与代码模块相关的依赖项分组，以便以后进行卸载。

AssemblyLoadContext.Default 实例有什么特别之处？

启动时，运行时将自动填充 AssemblyLoadContext.Default 实例。它使用默认探测来定位和查找所有静态依赖项。

它解决了最常见的依赖项加载方案。

AssemblyLoadContext 如何支持动态依赖项？

AssemblyLoadContext 具有可替代的各种事件和虚函数。

AssemblyLoadContext.Default 实例仅支持替代事件。

托管程序集加载算法、附属程序集加载算法和非托管（本机）库加载算法文章引用了所有可用事件和虚拟函数。这些文章显示加载算法中的每个事件和函数的相对位置。本文不会重现该信息。

本部分介绍相关事件和函数的一般原则。

- **可重复**。针对特定依赖项的查询必须始终产生相同的响应。必须返回同一个已加载的依赖项实例。此要求是保持缓存一致性的基础。特别是对于托管程序集，我们要创建 Assembly 缓存。缓存键是一个简单的程序集名称 AssemblyName.Name。
- **通常不引发**。当找不到请求的依赖项时，这些函数应返回 `null` 而不是引发。引发将提前结束搜索，并将异

常传播到调用方。应将引发限制为针对意外错误，如程序集损坏或内存不足等情况。

- **避免递归**。请注意，这些函数和处理程序实现了用于定位依赖项的加载规则。实现不应调用触发递归的 API。代码通常应调用 AssemblyLoadContext 加载函数，这些函数需要特定路径或内存引用参数。
- **加载到正确的 AssemblyLoadContext**。选择加载依赖项的位置是应用程序特定的。选择是通过这些事件和函数实现的。当代码调用 AssemblyLoadContext 时，按路径加载函数在你要加载代码的实例上调用它们。有时返回 `null`，并让 AssemblyLoadContext.Default 处理加载可能是最简单的选项。
- **注意线程争用**。加载可由多个线程触发。AssemblyLoadContext 通过以原子方式将程序集添加到其缓存来处理线程争用。将丢弃争用失败方的实例。在实现逻辑中，不要添加未正确处理多个线程的额外逻辑。

如何隔离动态依赖项？

每个 AssemblyLoadContext 实例都表示 Assembly 实例和 Type 定义的唯一范围。

这些依赖项之间没有任何二进制隔离。它们仅通过不按名称查找彼此来进行隔离。

在每个 AssemblyLoadContext 中：

- AssemblyName.Name 可以引用不同的 Assembly 实例。
- Type.GetType 可能会为同一类型 `name` 返回不同类型的实例。

如何共享依赖项？

可以在 AssemblyLoadContext 实例之间轻松共享依赖项。常规模型用于一个 AssemblyLoadContext 来加载依赖项。另一个通过使用对已加载程序集的引用来共享依赖项。

此共享是运行时程序集所必需的。这些程序集只能加载到 AssemblyLoadContext.Default。`ASP.NET`、`WPF` 或 `WinForms` 等框架也是如此。

建议将共享依赖项加载到 AssemblyLoadContext.Default。此共享是常见的设计模式。

共享是通过自定义 AssemblyLoadContext 实例编码实现的。AssemblyLoadContext 具有可替代的各种事件和虚函数。当这些函数中的任何函数返回对在另一个 AssemblyLoadContext 实例中加载的 Assembly 实例的引用时，将共享 Assembly 实例。标准加载算法会延迟 AssemblyLoadContext.Default 加载，以简化通用共享模式。有关详细信息，请参阅托管程序集加载算法。

复杂情况

类型转换问题

当两个 AssemblyLoadContext 实例包含具有相同 `name` 的类型定义时，它们不是同一类型。当且仅当它们来自同一个 Assembly 实例时，它们的类型相同。

使事情复杂化的是，这些不匹配类型的异常消息可能会令人困惑。在异常消息中，按简单类型名称来引用这些类型。在这种情况下，常见异常消息的格式如下所示：

> 无法将类型为“IsolatedType”的对象转换为类型“IsolatedType”。

调试类型转换问题

如果给定一对不匹配的类型，还必须要了解：

- 每种类型的 Type.Assembly。
- 每种类型的 AssemblyLoadContext，这可以通过 AssemblyLoadContext.GetLoadContext(Assembly) 函数获得。

如果给定两个对象 `a` 和 `b`，在调试器中评估以下内容将非常有用：

```
// In debugger look at each assembly's instance, Location, and FullName
a.GetType().Assembly
b.GetType().Assembly
// In debugger look at each AssemblyLoadContext's instance and name
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(a.GetType().Assembly)
System.Runtime.Loader.AssemblyLoadContext.GetLoadContext(b.GetType().Assembly)
```

解决类型转换问题

可以通过两种设计模式来解决这些类型转换问题。

1. 使用常见的共享类型。此共享类型可以是基元运行时类型，也可以涉及在共享程序集中创建新的共享类型。共享类型通常是在应用程序的程序集中定义的接口。有关详细信息，请参阅如何共享依赖项。

2. 使用封送处理技术从一种类型转换为另一种类型。

默认探测

2021/11/16 •

AssemblyLoadContext.Default 实例负责定位程序集的依赖项。本文介绍 AssemblyLoadContext.Default 实例的探测逻辑。

主机配置的探测属性

运行时启动时，运行时主机提供一组命名的探测属性，这些属性可配置 AssemblyLoadContext.Default 探测路径。

每个探测属性均可选。如果存在，则每个属性都是一个字符串值，其中包含绝对路径的分隔列表。在 Windows 上，分隔符为";"，在所有其他平台上，分隔符为":"。

`TRUSTED_PLATFORM_ASSEMBLIES`	平台和应用程序程序集文件路径的列表。
`PLATFORM_RESOURCE_ROOTS`	用于搜索附属资源程序集的目录路径的列表。
`NATIVE_DLL_SEARCH_DIRECTORIES`	用于搜索非托管(本机)库的目录路径的列表。
`APP_PATHS`	用于搜索托管程序集的目录路径的列表。

如何填充属性？

填充属性有两个主要方案，具体取决于 <myapp>.deps.json 文件是否存在。

- 当 *.deps.json 文件存在时，将对其进行分析以填充探测属性 。
- 如果 *.deps.json 文件不存在，则假定应用程序的目录以包含所有依赖项。目录的内容用于填充探测属性。

此外，也会对任何引用框架的 *.deps.json 文件进行类似的分析。

最后，可使用环境变量 `ADDITIONAL_DEPS` 添加其他依赖项。`dotnet.exe` 还包含一个 `--additional-deps` 可选参数，用于在应用程序启动时设置此值。

默认不会填充 `APP_PATHS` 属性，大多数应用程序都省略了该属性。

可以通过 `System.AppContext.GetData("APP_CONTEXT_DEPS_FILES")` 访问应用程序使用的所有 *.deps.json 文件的列表。

如何查看托管代码中的探测属性？

通过使用上表中的属性名称调用 AppContext.GetData(String) 函数，可查看每个属性。

如何调试探测属性的构造？

启用某些环境变量后，.NET Core 运行时主机将输出有用的跟踪消息：

`COREHOST_TRACE=1`	启用跟踪。
`COREHOST_TRACEFILE=<path>`	跟踪文件路径而不是默认 `stderr`。

[illegible]	[illegible]
`COREHOST_TRACE_VERBOSITY`	将详细程度设置为从 1（最低）到 4（最高）。

托管程序集默认探测

当探测以定位托管程序集时，AssemblyLoadContext.Default 会按以下顺序查找：

- 与 `TRUSTED_PLATFORM_ASSEMBLIES` 中的 AssemblyName.Name 匹配的文件（在删除文件扩展名之后）。
- 包含公共文件扩展名的 `APP_PATHS` 中的程序集文件。

附属（资源）程序集探测

若要查找特定区域性的附属程序集，请构造一组文件路径。

对于 `PLATFORM_RESOURCE_ROOTS` 和 `APP_PATHS` 中的每个路径，附加 CultureInfo.Name 字符串、目录分隔符、AssemblyName.Name 字符串和扩展名“.dll”。

如果存在任何匹配的文件，请尝试加载并返回该文件。

非托管（本机）库探测

当探测以查找非托管库时，将搜索 `NATIVE_DLL_SEARCH_DIRECTORIES` 以查找匹配库。

托管程序集加载算法

2021/11/16 •

托管程序集与涉及不同阶段的算法一起定位并加载。

除附属程序集和 `WinRT` 程序集之外的所有托管程序集都使用相同算法。

何时加载托管程序集？

触发托管程序集加载的最常见机制是静态程序集引用。每当代码使用在另一个程序集中定义的类型时，编译器都会插入这些引用。根据运行时的需要加载这些程序集（`load-by-name`）。未指定加载静态程序集引用的确切时间。它可能因运行时版本而异，并且受内联等优化影响。

直接使用特定的 API 也将触发加载：

API	从	`ACTIVE` ASSEMBLYLOADCONTEXT
AssemblyLoadContext.LoadFromAssemblyName	`Load-by-name`	this 实例。
AssemblyLoadContext.LoadFromAssemblyPath AssemblyLoadContext.LoadFromNativeImagePath	从路径加载。	this 实例。
AssemblyLoadContext.LoadFromStream	从对象加载。	this 实例。
Assembly.LoadFile	在新的 AssemblyLoadContext 实例中从路径加载	新的 AssemblyLoadContext 实例。
Assembly.LoadFrom	在 AssemblyLoadContext.Default 实例中从路径加载。 向 AssemblyLoadContext.Default 添加 Resolving 处理程序。处理程序将从其目录加载程序集的依赖项。	AssemblyLoadContext.Default 实例。
Assembly.Load(AssemblyName) Assembly.Load(String) Assembly.LoadWithPartialName	`Load-by-name`。	从调用方推断。 首选 AssemblyLoadContext 方法。
Assembly.Load(Byte[]) Assembly.Load(Byte[], Byte[])	从新 AssemblyLoadContext 实例的对象中加载。	新的 AssemblyLoadContext 实例。
Type.GetType(String) Type.GetType(String, Boolean) Type.GetType(String, Boolean, Boolean)	`Load-by-name`。	从调用方推断。 首选使用 `assemblyResolver` 参数的 Type.GetType 方法。

API	使	`ACTIVE` ASSEMBLYLOADCONTEXT

Assembly.GetType	如果类型 `name` 描述程序集限定的泛型类型，则触发 `Load-by-name`。	从调用方推断。 使用程序集限定的类型名称时，首选 Type.GetType。
Activator.CreateInstance(String, String) Activator.CreateInstance(String, String, Object[]) Activator.CreateInstance(String, String, Boolean, BindingFlags, Binder, Object[], CultureInfo, Object[])	`Load-by-name`.	从调用方推断。 首选采用 Type 参数的 Activator.CreateInstance 方法。

算法

以下算法描述运行时如何加载托管程序集。

1. 确定 `active` AssemblyLoadContext。
   - 对于静态程序集引用，`active` AssemblyLoadContext 是已加载引用程序集的实例。
   - 首选 API 使 `active` AssemblyLoadContext 显式。
   - 其他 API 推断 `active` AssemblyLoadContext。对于这些 API，将使用 AssemblyLoadContext.CurrentContextualReflectionContext 属性。如果其值为 `null`，则使用推断的 AssemblyLoadContext 实例。
   - 请见上表。
2. 对于 `Load-by-name` 方法，`active` AssemblyLoadContext 会加载程序集。通过以下方式按优先级排序：
   - 检查其 `cache-by-name`。
   - 调用 AssemblyLoadContext.Load 函数。
   - 检查 AssemblyLoadContext.Default 实例的缓存并运行托管程序集默认探测逻辑。
     - 如果程序集是新加载的，则会向 AssemblyLoadContext.Default 实例的 `cache-by-name` 添加一个引用。
   - 引发 AssemblyLoadContext 活动的 AssemblyLoadContext.Resolving 事件。
   - 引发 AppDomain.AssemblyResolve 事件。
3. 对于其他类型的加载，`active` AssemblyLoadContext 加载程序集。通过以下方式按优先级排序：
   - 检查其 `cache-by-name`。
   - 从指定的路径或原始程序集对象加载。
     - 如果程序集是新加载的，则会向 `active` AssemblyLoadContext 实例的 `cache-by-name` 添加一个引用。
4. 在任一情况下，如果新加载了一个程序集，则：

- 引发 AppDomain.AssemblyLoad 事件。

附属程序集加载算法

2021/11/16 •

使用附属程序集来存储为语言和区域性自定义的本地化资源。

附属程序集使用不同于常规托管程序集的加载算法。

何时加载附属程序集？

加载本地化资源时加载附属程序集。

加载本地化资源的基本 API 是 System.Resources.ResourceManager 类。最后，ResourceManager 类将为每个 CultureInfo.Name 调用 GetSatelliteAssembly 方法。

较高级别的 API 可能会提取低级别 API。

算法

.NET Core 资源回退进程包含以下步骤：

1. 确定 `active` AssemblyLoadContext 实例。在所有情况下，`active` 实例都是执行程序集的 AssemblyLoadContext。
2. `active` 实例尝试通过以下方式按优先级排序来加载请求的区域性的附属程序集：
   - 检查其缓存。
   - 检查当前正在执行程序集的目录，查找与请求的 CultureInfo.Name（例如 `es-MX`）匹配的子目录。

     > **NOTE**
     > 3.0 版之前的 .NET Core 中未实现此功能。

     > **NOTE**
     > 在 Linux 和 macOS 上，子目录区分大小写，并且必须是以下两种情况之一：
     > - 完全匹配大小写。
     > - 为小写。

   - 如果 `active` 是 AssemblyLoadContext.Default 实例，则通过运行默认附属（资源）程序集探测逻辑。
   - 调用 AssemblyLoadContext.Load 函数。
   - 引发 AssemblyLoadContext.Resolving 事件。
   - 引发 AppDomain.AssemblyResolve 事件。
3. 如果加载附属程序集：
   - 引发 AppDomain.AssemblyLoad 事件。
   - 将搜索程序集以查找请求的资源。如果运行时在程序集中找到该资源，则使用它。如果找不到该资源，将继续搜索。

> **NOTE**
> 要在附属程序集中查找资源，运行时将搜索 ResourceManager 为当前 CultureInfo.Name 请求的资源文件。在资源文件中，它搜索请求的资源名称。如果找不到上述任何一个，则资源将被视为未找到。

4. 接下来，运行时通过许多潜在级别搜索父区域性程序集，每次均重复步骤 2 和 3。

   每个区域性只有一个父级，由 CultureInfo.Parent 属性定义。

   当区域性的 Parent 属性为 CultureInfo.InvariantCulture 时，父区域性搜索将停止。

   对于 InvariantCulture，我们不会返回到步骤 2 和 3，而是继续执行步骤 5。

5. 如果仍未找到资源，则使用默认（回退）区域性的资源。

   通常，默认区域性的资源包含在主应用程序集中。不过，可以为 NeutralResourcesLanguageAttribute.Location 属性指定 UltimateResourceFallbackLocation.Satellite。此值指示资源的最终回退位置是附属程序集，而不是主程序集。

   > **NOTE**
   > 默认区域性为最终回退。因此，建议在默认资源文件中始终包含一组详尽的资源。这有助于防止引发异常。通过拥有详尽资源集，可为所有资源提供回退，并确保始终向用户呈现至少一种资源，即使该资源不是特定于区域性的。

6. 最后，

   - 如果运行时找不到默认（回退）区域性的资源文件，将引发 MissingManifestResourceException 或 MissingSatelliteAssemblyException 异常。
   - 如果找到资源文件但是请求的资源不存在，则请求返回 `null`。

非托管（本机）库加载算法

2021/11/16 •

非托管库与涉及不同阶段的算法一起定位并加载。

以下算法描述如何通过 `PInvoke` 加载本机库。

`PInvoke` 加载库算法

`PInvoke` 在尝试加载非托管程序集时使用以下算法：

1. 确定 `active` AssemblyLoadContext。对于非托管加载库，`active` AssemblyLoadContext 是具有定义 `PInvoke` 的程序集的算法。
2. 对于 `active` AssemblyLoadContext，尝试通过以下方式按优先级排序来查找程序集：
   - 检查其缓存。
   - 调用由 NativeLibrary.SetDllImportResolver(Assembly, DllImportResolver) 函数设置的当前 System.Runtime.InteropServices.DllImportResolver 委托。
   - 对 `active` AssemblyLoadContext 调用 AssemblyLoadContext.LoadUnmanagedDll 函数。
   - 检查 AppDomain 实例的缓存并运行非托管（本机）库探测逻辑。
   - 引发 `active` AssemblyLoadContext 的 AssemblyLoadContext.ResolvingUnmanagedDll 事件。

收集详细的程序集加载信息

2021/11/16 •

自 .NET 5 起，运行时可以通过 `EventPipe` 发出事件，其中包含关于托管程序集加载的详细信息，以帮助诊断程序集加载问题。这些事件是由 `Microsoft-Windows-DotNETRuntime` 提供程序在 `AssemblyLoader` 关键字 (`0x4`) 下发出的。

先决条件

- .NET 5 SDK 或更高版本
- `dotnet-trace` 工具

> **NOTE**
> `dotnet-trace` 功能的范围不仅限于收集详细的程序集加载信息。有关 `dotnet-trace` 用法的详细信息，请参阅 `dotnet-trace` 。

收集包含程序集加载事件的跟踪

可以使用 `dotnet-trace` 跟踪现有进程或启动子进程并从启动中跟踪它。

跟踪现有进程

若要在运行时中启用程序集加载事件并收集它们的跟踪，请使用 `dotnet-trace` 和以下命令：

```
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id <pid>
```

此命令将收集指定 `<pid>` 的跟踪，从而在 `Microsoft-Windows-DotNETRuntime` 提供程序中启用 `AssemblyLoader` 事件。结果是 `.nettrace` 文件。

使用 dotnet-trace 启动子进程并从启动中跟踪它

有时，从进程启动中收集进程的跟踪可能很有用。对于运行 .NET 5 或更高版本的应用，可以使用 `dotnet-trace` 来执行此操作。

以下命令以 `arg1` 和 `arg2` 作为其命令行参数启动 hello.exe，并从其运行时启动收集跟踪：

```
dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 -- hello.exe arg1 arg2
```

可以通过按 `Enter` 或 `Ctrl + C` 来停止收集跟踪。这也会关闭 hello.exe。

> **NOTE**
> - 通过 `dotnet-trace` 启动 hello.exe 会重定向其输入和输出；默认情况下，你将无法在控制台上与其交互。使用 `--show-child-io` 开关与其 `stdin` 和 `stdout` 进行交互。
> - 通过 `Ctrl+C` 或 `SIGTERM` 退出工具将正常地结束该工具和子进程。
> - 如果子进程在工具之前退出，工具也将退出，应可安全查看跟踪。

查看跟踪

可以使用 PerfView中的"事件"视图在 Windows 上查看所收集的跟踪文件。所有程序集加载事件都将以 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 为前缀。

示例(在 Windows 上)

本例使用程序集加载扩展点示例。应用程序尝试加载程序集 `MyLibrary`(应用程序未引用的程序集),因此需要在程序集加载扩展点中进行处理才能成功加载。

收集跟踪

1. 导航到包含下载的示例的目录。通过以下方式生成应用程序:

   ```
   dotnet build
   ```

2. 使用指明应用程序应暂停的参数来启动应用程序,并等待按键。恢复后,它将尝试在默认 `AssemblyLoadContext` 中加载程序集,不需要进行成功加载所需的处理。导航到输出目录并运行:

   ```
   AssemblyLoading.exe /d default
   ```

3. 找到应用程序的进程 ID。

   ```
   dotnet-trace ps
   ```

   输出将列出可用进程。例如:。

   ```
   35832 AssemblyLoading C:\src\AssemblyLoading\bin\Debug\net5.0\AssemblyLoading.exe
   ```

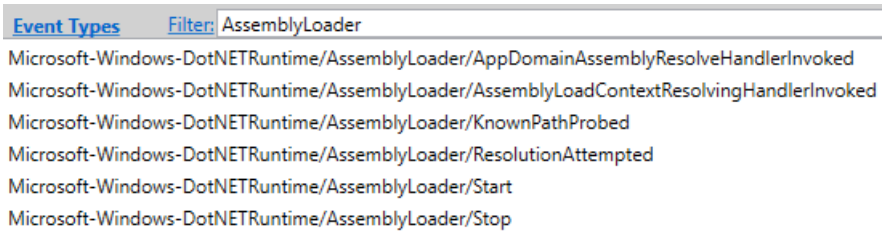
4. 将 `dotnet-trace` 附加到正在运行的应用程序。

   ```
   dotnet-trace collect --providers Microsoft-Windows-DotNETRuntime:4 --process-id 35832
   ```

5. 在运行应用程序的窗口中,按任意键让程序继续运行。一旦应用程序退出,跟踪将自动停止。

查看跟踪

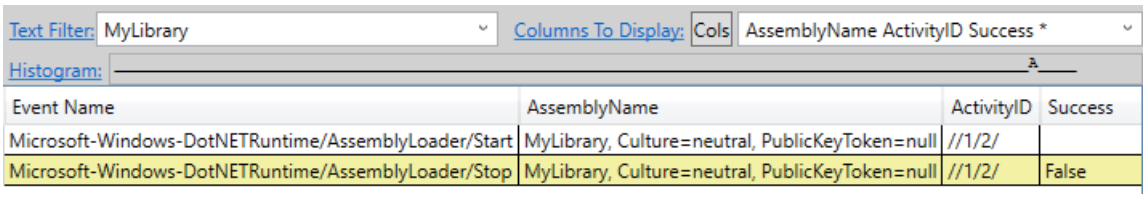
在 PerfView 中打开所收集的跟踪,然后打开"事件"视图。将"事件"列表筛选为 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 事件。



将显示在跟踪启动后应用程序中发生的所有程序集加载。若要检查此示例中关注的程序集 (`MyLibrary`) 的加载操作,可以执行更多筛选。

程序集加载

使用左侧的"事件"列表,将视图筛选为 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 下的 `Start` 和 `Stop` 事件。将列 `AssemblyName`、`ActivityID` 和 `Success` 添加到视图中。筛选为包含 `MyLibrary` 的事件。

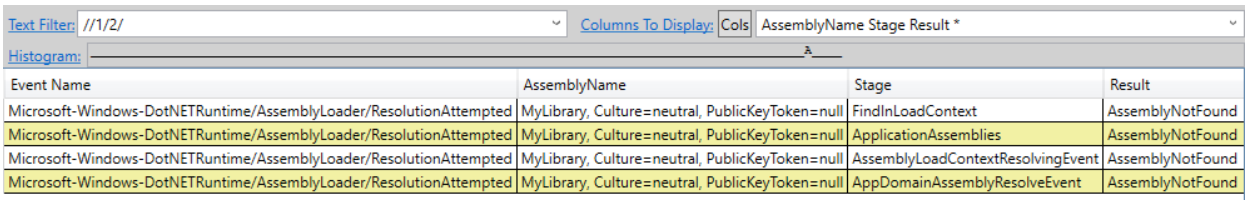


[illegible]	ASSEMBLYNAME	ACTIVITYID	SUCCESS
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	False

你应该会在 `Stop` 事件上看到一个带有 `Success=False` 的 `Start` / `Stop` 对，表明加载操作失败。请注意，这两个事件有相同的活动 ID。活动 ID 可用于将其他所有程序集加载程序事件筛选为仅与此加载操作相对应的事件。

加载尝试明细

有关加载操作的更详细明细，请使用左侧的“事件”列表，将视图筛选为 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 下的 `ResolutionAttempted` 事件。将列 `AssemblyName`、`Stage` 和 `Result` 添加到视图中。筛选为包含 `Start` / `Stop` 对中的活动 ID 的事件。

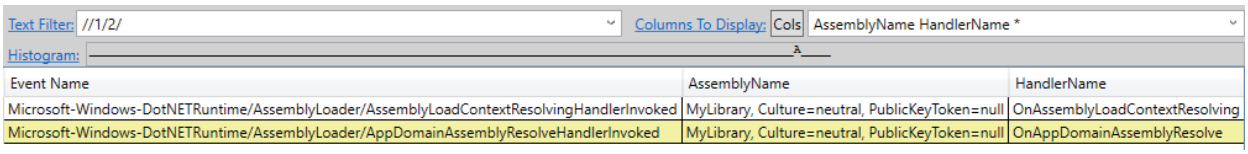


[illegible]	ASSEMBLYNAME	[illegible]	[illegible]
`AssemblyLoader/ResolutionAtt`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAtt`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAtt`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextResolving`	`AssemblyNotFound`
`AssemblyLoader/ResolutionAtt`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AppDomainAssemblyResolveEver`	`AssemblyNotFound`

上面的事件表明，程序集加载程序试图通过以下方式解析程序集：在当前加载上下文中查找、运行托管应用程序程序集的默认探测逻辑、调用 AssemblyLoadContext.Resolving 事件的处理程序，以及调用 AppDomain.AssemblyResolve 的处理程序。对于所有这些步骤，都没有找到程序集。

扩展点

若要查看调用了哪些扩展点，请使用左侧的“事件”列表，将视图筛选为 `Microsoft-Windows-DotNETRuntime/AssemblyLoader` 下的 `AssemblyLoadContextResolvingHandlerInvoked` 和 `AppDomainAssemblyResolveHandlerInvoked`。将列 `AssemblyName` 和 `HandlerName` 添加到视图中。筛选为包含 `Start` / `Stop` 对中的活动 ID 的事件。



IIII	ASSEMBLYNAME	HANDLERNAME
`AssemblyLoader/AssemblyLoadContextRes`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAssemblyLoadContextResolving`
`AssemblyLoader/AppDomainAssemblyResol`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`OnAppDomainAssemblyResolve`

上面的事件表明，为 AssemblyLoadContext.Resolving 事件调用了名为 `OnAssemblyLoadContextResolving` 的处理程序，为 AppDomain.AssemblyResolve 事件调用了名为 `OnAppDomainAssemblyResolve` 的处理程序。

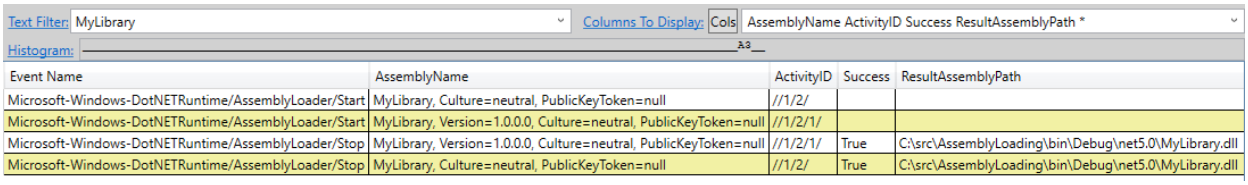
收集另一个跟踪

使用参数运行应用程序，以便 AssemblyLoadContext.Resolving 事件的处理程序将加载 `MyLibrary` 程序集。

```
AssemblyLoading /d default alc-resolving
```

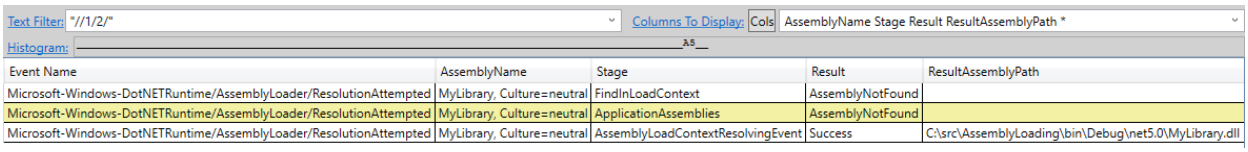
收集并使用上面的步骤打开另一个 `.nettrace` 文件。

再次筛选为 `MyLibrary` 的 `Start` 和 `Stop` 事件。你应该会看到 `Start` / `Stop` 对，以及另一个 `Start` / `Stop` 介于二者之间。内部加载操作表示 AssemblyLoadContext.Resolving 的处理程序在调用 AssemblyLoadContext.LoadFromAssemblyPath 时触发的加载。这一次，你应该会在 `Stop` 事件上看到 `Success=True`，表明加载操作成功。`ResultAssemblyPath` 字段显示生成的程序集的路径。



IIII	ASSEMBLYNAME	ACTIVITYID	SUCCESS	RESULTASSEMBLYPATH
`AssemblyLoader/Start`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/		
`AssemblyLoader/Start`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/		
`AssemblyLoader/Stop`	`MyLibrary, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`	//1/2/1/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll
`AssemblyLoader/Stop`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	//1/2/	True	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

然后，我们可以查看带有外部负载的活动 ID 的 `ResolutionAttempted` 事件，以确定成功解析程序集的步骤。这一次，事件会显示 `AssemblyLoadContextResolvingEvent` 阶段已成功完成。`ResultAssemblyPath` 字段显示生成的程序集的路径。



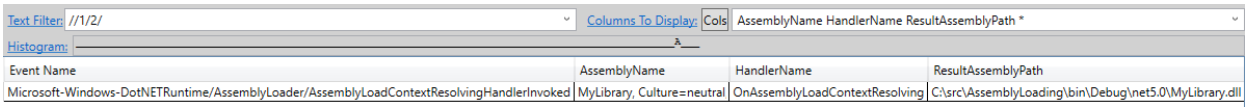
Text Filter: "//1/2/" Columns To Display: Cols AssemblyName Stage Result ResultAssemblyPath *

Histogram:

Event Name	AssemblyName	Stage	Result	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	FindInLoadContext	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	ApplicationAssemblies	AssemblyNotFound	
Microsoft-Windows-DotNETRuntime/AssemblyLoader/ResolutionAttempted	MyLibrary, Culture=neutral	AssemblyLoadContextResolvingEvent	Success	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

[illegible]	ASSEMBLYNAME	[illegible]	[illegible]	RESULTASSEMBLYPATH
`AssemblyLoader/Resolut`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`FindInLoadContext`	`AssemblyNotFound`	
`AssemblyLoader/Resolut`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`ApplicationAssemblies`	`AssemblyNotFound`	
`AssemblyLoader/Resolut`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`AssemblyLoadContextRes`	`Success` `nt`	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

查看 `AssemblyLoadContextResolvingHandlerInvoked` 事件将看到调用了名为 `OnAssemblyLoadContextResolving` 的处理程序。 `ResultAssemblyPath` 字段显示由处理程序返回的程序集的路径。



Text Filter: //1/2/ Columns To Display: Cols AssemblyName HandlerName ResultAssemblyPath *

Histogram:

Event Name	AssemblyName	HandlerName	ResultAssemblyPath
Microsoft-Windows-DotNETRuntime/AssemblyLoader/AssemblyLoadContextResolvingHandlerInvoked	MyLibrary, Culture=neutral	OnAssemblyLoadContextResolving	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

[illegible]	ASSEMBLYNAME	HANDLERNAME	RESULTASSEMBLYPATH
`AssemblyLoader/AssemblyLoadC`	`MyLibrary, Culture=neutral, PublicKeyToken=null`	`voke` `OnAssemblyLoadContextResolving`	C:\src\AssemblyLoading\bin\Debug\net5.0\MyLibrary.dll

请注意，不再存在带有 `AppDomainAssemblyResolveEvent` 阶段的 `ResolutionAttempted` 事件或任何 `AppDomainAssemblyResolveHandlerInvoked` 事件，因为程序集在到达引发 AppDomain.AssemblyResolve 事件的加载算法步骤之前已成功加载。

另请参阅

- 程序集加载程序事件
- dotnet-trace
- PerfView

使用插件创建 .NET Core 应用程序

2021/11/16 •

本教程展示了如何创建自定义的 AssemblyLoadContext 来加载插件。AssemblyDependencyResolver 用于解析插件的依赖项。该教程正确地将插件依赖项与主机应用程序隔离开来。将了解如何执行以下操作：

- 构建支持插件的项目。
- 创建自定义 AssemblyLoadContext 加载每个插件。
- 使用 System.Runtime.Loader.AssemblyDependencyResolver 类型允许插件具有依赖项。
- 只需复制生成项目就可以轻松部署的作者插件。

系统必备

- 安装 .NET 5 SDK 或更高版本。

> **NOTE**
> 示例代码针对 .NET 5，但它使用的所有功能都已在 .NET Core 3.0 中推出，并且在此后所有 .NET 版本中都可用。

创建应用程序

第一步是创建应用程序：

1. 创建新文件夹，并在该文件夹中运行以下命令：

```
dotnet new console -o AppWithPlugin
```

2. 为了更容易生成项目，请在同一文件夹中创建一个 Visual Studio 解决方案文件。运行以下命令：

```
dotnet new sln
```

3. 运行以下命令，向解决方案添加应用项目：

```
dotnet sln add AppWithPlugin/AppWithPlugin.csproj
```

现在，我们可以填写应用程序的主干。使用下面的代码替换 AppWithPlugin/Program.cs 文件中的代码：

```
using PluginBase;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Reflection;

namespace AppWithPlugin
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                if (args.Length == 1 && args[0] == "/d")
                {
                    Console.WriteLine("Waiting for any key...");
                    Console.ReadLine();
                }

                // Load commands from plugins.

                if (args.Length == 0)
                {
                    Console.WriteLine("Commands: ");
                    // Output the loaded commands.
                }
                else
                {
                    foreach (string commandName in args)
                    {
                        Console.WriteLine($"-- {commandName} --");

                        // Execute the command with the name passed as an argument.

                        Console.WriteLine();
                    }
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex);
            }
        }
    }
}
```

创建插件接口

使用插件生成应用的下一步是定义插件需要实现的接口。我们建议创建类库，其中包含计划用于在应用和插件之间通信的任何类型。此部分允许将插件接口作为包发布，而无需发布完整的应用程序。

在项目的根文件夹中，运行 `dotnet new classlib -o PluginBase` 。并运行 `dotnet sln add PluginBase/PluginBase.csproj` 向解决方案文件添加项目。删除 `PluginBase/Class1.cs` 文件，并使用以下接口定义在名为 `ICommand.cs` 的 `PluginBase` 文件夹中创建新的文件：

```
namespace PluginBase
{
    public interface ICommand
    {
        string Name { get; }
        string Description { get; }

        int Execute();
    }
}
```

此 `ICommand` 接口是所有插件将实现的接口。

由于已定义 `ICommand` 接口，所以应用程序项目可以填写更多内容。使用根文件夹中的 `dotnet add AppWithPlugin/AppWithPlugin.csproj reference PluginBase/PluginBase.csproj` 命令将引用从 `AppWithPlugin` 项目添加到 `PluginBase` 项目。

使用以下代码片段替换 `// Load commands from plugins` 注释，使其能够从给定文件路径加载插件：

```
string[] pluginPaths = new string[]
{
    // Paths to plugins to load.
};

IEnumerable<ICommand> commands = pluginPaths.SelectMany(pluginPath =>
{
    Assembly pluginAssembly = LoadPlugin(pluginPath);
    return CreateCommands(pluginAssembly);
}).ToList();
```

然后用以下代码片段替换 `// Output the loaded commands` 注释：

```
foreach (ICommand command in commands)
{
    Console.WriteLine($"{command.Name}\t - {command.Description}");
}
```

使用以下代码片段替换 `// Execute the command with the name passed as an argument` 注释：

```
ICommand command = commands.FirstOrDefault(c => c.Name == commandName);
if (command == null)
{
    Console.WriteLine("No such command is known.");
    return;
}

command.Execute();
```

最后，将静态方法添加到名为 `LoadPlugin` 和 `CreateCommands` 的 `Program` 类，如下所示：

```
static Assembly LoadPlugin(string relativePath)
{
    throw new NotImplementedException();
}

static IEnumerable<ICommand> CreateCommands(Assembly assembly)
{
    int count = 0;

    foreach (Type type in assembly.GetTypes())
    {
        if (typeof(ICommand).IsAssignableFrom(type))
        {
            ICommand result = Activator.CreateInstance(type) as ICommand;
            if (result != null)
            {
                count++;
                yield return result;
            }
        }
    }

    if (count == 0)
    {
        string availableTypes = string.Join(",", assembly.GetTypes().Select(t => t.FullName));
        throw new ApplicationException(
            $"Can't find any type which implements ICommand in {assembly} from {assembly.Location}.\n" +
            $"Available types: {availableTypes}");
    }
}
```

加载插件

现在，应用程序可以正确加载和实例化来自已加载的插件程序集的命令，但仍然无法加载插件程序集。使用以下内容在 AppWithPlugin 文件夹中创建名为 PluginLoadContext.cs 的文件：

```
using System;
using System.Reflection;
using System.Runtime.Loader;

namespace AppWithPlugin
{
    class PluginLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public PluginLoadContext(string pluginPath)
        {
            _resolver = new AssemblyDependencyResolver(pluginPath);
        }

        protected override Assembly Load(AssemblyName assemblyName)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(assemblyName);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }

        protected override IntPtr LoadUnmanagedDll(string unmanagedDllName)
        {
            string libraryPath = _resolver.ResolveUnmanagedDllToPath(unmanagedDllName);
            if (libraryPath != null)
            {
                return LoadUnmanagedDllFromPath(libraryPath);
            }

            return IntPtr.Zero;
        }
    }
}
```

`PluginLoadContext` 类型派生自 AssemblyLoadContext。`AssemblyLoadContext` 类型是运行时中的特殊类型，该类型允许开发人员将已加载的程序集隔离到不同的组中，以确保程序集版本不冲突。此外，自定义 `AssemblyLoadContext` 可以选择不同路径来加载程序集格式并重写默认行为。`PluginLoadContext` 使用 .NET Core 3.0 中引入的 `AssemblyDependencyResolver` 类型的实例将程序集名称解析为路径。`AssemblyDependencyResolver` 对象是使用 .NET 类库的路径构造的。它根据类库的 .deps.json 文件（其路径传递给 `AssemblyDependencyResolver` 构造函数）将程序集和本机库解析为它们的相对路径。自定义 `AssemblyLoadContext` 使插件能够拥有自己的依赖项，`AssemblyDependencyResolver` 使正确加载依赖项变得容易。

由于 `AppWithPlugin` 项目具有 `PluginLoadContext` 类型，所以请使用以下正文更新 `Program.LoadPlugin` 方法：

```
static Assembly LoadPlugin(string relativePath)
{
    // Navigate up to the solution root
    string root = Path.GetFullPath(Path.Combine(
        Path.GetDirectoryName(
            Path.GetDirectoryName(
                Path.GetDirectoryName(
                    Path.GetDirectoryName(
                        Path.GetDirectoryName(typeof(Program).Assembly.Location)))))));

    string pluginLocation = Path.GetFullPath(Path.Combine(root, relativePath.Replace('\\',
Path.DirectorySeparatorChar)));
    Console.WriteLine($"Loading commands from: {pluginLocation}");
    PluginLoadContext loadContext = new PluginLoadContext(pluginLocation);
    return loadContext.LoadFromAssemblyName(new
AssemblyName(Path.GetFileNameWithoutExtension(pluginLocation)));
}
```

通过为每个插件使用不同的 `PluginLoadContext` 实例，插件可以具有不同的甚至冲突的依赖项，而不会出现问题。

不具有依赖项的简单插件

返回到根文件夹，执行以下步骤：

1. 运行以下命令，新建一个名为 `HelloPlugin` 的类库项目：

   ```
   dotnet new classlib -o HelloPlugin
   ```

2. 运行以下命令，将项目添加到 `AppWithPlugin` 解决方案中：

   ```
   dotnet sln add HelloPlugin/HelloPlugin.csproj
   ```

3. 使用以下内容将 HelloPlugin/Class1.cs 文件替换为名为 HelloCommand.cs 的文件：

```
using PluginBase;
using System;

namespace HelloPlugin
{
    public class HelloCommand : ICommand
    {
        public string Name { get => "hello"; }
        public string Description { get => "Displays hello message."; }

        public int Execute()
        {
            Console.WriteLine("Hello !!!");
            return 0;
        }
    }
}
```

现在，打开 HelloPlugin.csproj 文件 。它应类似于以下内容：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5</TargetFramework>
  </PropertyGroup>

</Project>
```

在 `<PropertyGroup>` 标记之间添加以下元素：

```
  <EnableDynamicLoading>true</EnableDynamicLoading>
```

`<EnableDynamicLoading>true</EnableDynamicLoading>` 准备项目，使其可用作插件。此外，这会将其所有依赖项复制到项目的输出中。有关更多详细信息，请参阅 `EnableDynamicLoading`。

在 `<Project>` 标记之间添加以下元素：

```
<ItemGroup>
    <ProjectReference Include="..\PluginBase\PluginBase.csproj">
        <Private>false</Private>
        <ExcludeAssets>runtime</ExcludeAssets>
    </ProjectReference>
</ItemGroup>
```

`<Private>false</Private>` 元素很重要。它告知 MSBuild 不要将 PluginBase.dll 复制到 HelloPlugin 的输出目录。如果 PluginBase.dll 程序集出现在输出目录中，`PluginLoadContext` 将在那里查找到该程序集并在加载 HelloPlugin.dll 程序集时加载它。此时，`HelloPlugin.HelloCommand` 类型将从 `HelloPlugin` 项目的输出目录中的 PluginBase.dll 实现 `ICommand` 接口，而不是加载到默认加载上下文中的 `ICommand` 接口。因为运行时将这两种类型视为不同程序集的不同类型，所以 `AppWithPlugin.Program.CreateCommands` 方法找不到命令。因此，对包含插件接口的程序集的引用需要 `<Private>false</Private>` 元数据。

同样，如果 `PluginBase` 引用其他包，则 `<ExcludeAssets>runtime</ExcludeAssets>` 元素也很重要。此设置与 `<Private>false</Private>` 的效果相同，但适用于 `PluginBase` 项目或它的某个依赖项可能包括的包引用。

因为 `HelloPlugin` 项目已完成，所以应该更新 `AppWithPlugin` 项目，以确认可以找到 `HelloPlugin` 插件的位置。在 `// Paths to plugins to load` 注释后，添加 `@"HelloPlugin\bin\Debug\netcoreapp3.0\HelloPlugin.dll"`（根据所使用的 .NET Core 版本，此路径可能有所不同）作为 `pluginPaths` 数组的元素。

具有库依赖项的插件

几乎所有插件都比简单的“Hello World”更复杂，而且许多插件都具有其他库上的依赖项。示例中的 `JsonPlugin` 和 `OldJsonPlugin` 项目显示了具有 `Newtonsoft.Json` 上的 NuGet 包依赖项的两个插件示例。因此，所有插件项目都应将 `<EnableDynamicLoading>true</EnableDynamicLoading>` 添加到项目属性，以便它们将其所有依赖项复制到 `dotnet build` 的输出中。使用 `dotnet publish` 发布类库也会将其所有依赖项复制到发布输出。

示例中的其他示例

可以在 dotnet/samples 存储库中找到本教程的完整源代码。完成的示例包括 `AssemblyDependencyResolver` 行为的一些其他示例。例如，`AssemblyDependencyResolver` 对象还可以解析本机库和 NuGet 包中所包含的已本地化的附属程序集。示例存储库中的 `UVPlugin` 和 `FrenchPlugin` 演示了这些方案。

从 NuGet 包引用插件接口

假设存在应用 A，它具有 NuGet 包（名为 `A.PluginBase`）中定义的插件接口。如何在插件项目中正确引用包？ 对于项目引用，使用项目文件的 `ProjectReference` 元素上的 `<Private>false</Private>` 元数据会阻止将 dll 复制到输出。

若要正确引用 `A.PluginBase` 包，应将项目文件中的 `<PackageReference>` 元素更改为以下内容：

```
<PackageReference Include="A.PluginBase" Version="1.0.0">
    <ExcludeAssets>runtime</ExcludeAssets>
</PackageReference>
```

此操作会阻止将 `A.PluginBase` 程序集复制到插件的输出目录，并确保插件将使用 A 版本的 `A.PluginBase`。

插件目标框架建议

因为插件依赖项加载使用 .deps.json 文件，所以存在一个与插件的目标框架相关的问题 。具体来说，插件应该以运行时为目标，比如 .NET 5，而不是某一版本的 .NET Standard。.deps.json 文件基于项目所针对的框架生成，而且由于许多与 .NET Standard 兼容的包提供了用于针对 .NET Standard 进行生成的引用程序集和用于特定运行时的实现程序集，因此 .deps.json 可能无法正确查看实现程序集，或者它可能会获取 .NET Standard 版本的程序集，而不是期望的 .NET Core 版本的程序集。

插件框架引用

插件当前无法向该过程引入新的框架。例如，无法将使用 `Microsoft.AspNetCore.App` 框架的插件加载到只使用根 `Microsoft.NETCore.App` 框架的应用程序中。主机应用程序必须声明对插件所需的全部框架的引用。

如何在 .NET Core 中使用和调试程序集可卸载性

2021/11/16 •

从 .NET Core 3.0 开始，支持加载和随后卸载一组程序集功能。在 .NET Framework 中，自定义应用域用于此目的，但 .NET Core 仅支持单个默认应用域。

.NET Core 3.0 及更高版本支持通过 AssemblyLoadContext 进行卸载。可将一组程序集加载到可回收的 `AssemblyLoadContext` 中，在其中执行方法或仅使用反射检查它们，最后卸载 `AssemblyLoadContext`。这会卸载加载到 `AssemblyLoadContext` 中的程序集。

使用 `AssemblyLoadContext` 和使用 AppDomain 进行卸载之间存在一个值得注意的差异。对于 Appdomain，卸载为强制执行。卸载时，会中止目标 AppDomain 中运行的所有线程，会销毁目标 AppDomain 中创建的托管 COM 对象，等等。对于 `AssemblyLoadContext`，卸载是"协作式的"。调用 AssemblyLoadContext.Unload 方法只是为了启动卸载。以下目标达成后，卸载完成：

- 没有线程将程序集中的方法加载到其调用堆栈上的 `AssemblyLoadContext` 中。
- 程序集中的任何类型都不会加载到 `AssemblyLoadContext`，这些类型的实例本身由以下引用：
  - `AssemblyLoadContext` 外部的引用，弱引用（WeakReference 或 WeakReference<T>）除外。
  - `AssemblyLoadContext` 内部和外部的强垃圾回收器 (GC) 句柄（GCHandleType.Normal 或 GCHandleType.Pinned）。

使用可回收的 AssemblyLoadContext

本部分包含详细的分步教程，其中演示了一种简单方法，可以将 .NET Core 应用程序加载到可回收的 `AssemblyLoadContext` 中，执行其入口点，然后将其卸载。可以在 https://github.com/dotnet/samples/tree/master/core/tutorials/Unloading 中找到完整示例。

创建可回收的 AssemblyLoadContext

需要从 AssemblyLoadContext 派生类，并替代其 AssemblyLoadContext.Load 方法。该方法解析对所有程序集的引用，这些程序集是加载到该 `AssemblyLoadContext` 中的程序集的依赖项。

以下代码是最简单的自定义 `AssemblyLoadContext` 示例：

```
class TestAssemblyLoadContext : AssemblyLoadContext
{
    public TestAssemblyLoadContext() : base(isCollectible: true)
    {
    }

    protected override Assembly Load(AssemblyName name)
    {
        return null;
    }
}
```

如你所见，`Load` 方法返回 `null`。这意味着所有依赖项程序集都会加载到默认上下文中，而新上下文仅包含显式加载到其中的程序集。

若要在 `AssemblyLoadContext` 中加载部分或全部依赖项，可以在 `Load` 方法中使用 `AssemblyDependencyResolver`。`AssemblyDependencyResolver` 将程序集名称解析为绝对程序集文件路径。解析程序使用加载到上下文中的主程序集的目录中的 .deps.json 文件和程序集文件 。

```
using System.Reflection;
using System.Runtime.Loader;

namespace complex
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        private AssemblyDependencyResolver _resolver;

        public TestAssemblyLoadContext(string mainAssemblyToLoadPath) : base(isCollectible: true)
        {
            _resolver = new AssemblyDependencyResolver(mainAssemblyToLoadPath);
        }

        protected override Assembly Load(AssemblyName name)
        {
            string assemblyPath = _resolver.ResolveAssemblyToPath(name);
            if (assemblyPath != null)
            {
                return LoadFromAssemblyPath(assemblyPath);
            }

            return null;
        }
    }
}
```

使用自定义且可回收的 AssemblyLoadContext

本部分假定使用的是 `TestAssemblyLoadContext` 的较简单版本。

可以创建自定义 `AssemblyLoadContext` 实例并将程序集加载到其中，如下所示：

```
var alc = new TestAssemblyLoadContext();
Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
```

对于已加载的程序集引用的每个程序集，将调用 `TestAssemblyLoadContext.Load` 方法，这样 `TestAssemblyLoadContext` 可以决定从何处获取程序集。在示例中，它返回 `null` 以指示它应从运行时默认加载程序集的位置加载到默认上下文中。

现在程序集已加载，可从中执行方法。运行 `Main` 方法：

```
var args = new object[1] {new string[] {"Hello"}};
int result = (int) a.EntryPoint.Invoke(null, args);
```

在 `Main` 方法返回后，可通过在自定义 `AssemblyLoadContext` 上调用 `Unload` 方法或删除对 `AssemblyLoadContext` 的引用来启动卸载：

```
alc.Unload();
```

这足以卸载测试程序集。将所有上述内容放入单独的非可内联方法中，以确保 `TestAssemblyLoadContext`、`Assembly` 和 `MethodInfo`（`Assembly.EntryPoint`）无法通过堆栈槽引用（实际或 JIT 引入的本地变量）保持活动状态。这可以使 `TestAssemblyLoadContext` 保持活动状态并阻止其卸载。

此外，返回对 `AssemblyLoadContext` 的弱引用，以便之后可以使用它来检测卸载是否已完成。

```
[MethodImpl(MethodImplOptions.NoInlining)]
static int ExecuteAndUnload(string assemblyPath, out WeakReference alcWeakRef)
{
    var alc = new TestAssemblyLoadContext();
    Assembly a = alc.LoadFromAssemblyPath(assemblyPath);

    alcWeakRef = new WeakReference(alc, trackResurrection: true);

    var args = new object[1] {new string[] {"Hello"}};
    int result = (int) a.EntryPoint.Invoke(null, args);

    alc.Unload();

    return result;
}
```

现在，可运行此函数以加载、执行和卸载程序集。

```
WeakReference testAlcWeakRef;
int result = ExecuteAndUnload("absolute/path/to/your/assembly", out testAlcWeakRef);
```

但是，卸载不会立即完成。如上所述，它依赖于垃圾回收器来收集测试程序集中的所有对象。在许多情况下，没有必要等待卸载完成。但是，某些情况下，了解卸载已经完成非常有用。例如，你可能希望删除从磁盘加载到自定义 `AssemblyLoadContext` 中的程序集文件。在这种情况下，可以使用以下代码片段。它会触发垃圾回收并等待循环中挂起的终结器，直到自定义 `AssemblyLoadContext` 的弱引用被设置为 `null`，表示已收集目标对象。在大多数情况下，只需要通过循环传递一次。但对于更复杂的情况，由 `AssemblyLoadContext` 中运行的代码所创建的对象具有终结器，则可能需要更多传递。

```
for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
}
```

卸载事件

某些情况下，加载到自定义 `AssemblyLoadContext` 中的代码可能需要在启动卸载时执行一些清理。例如，可能需要停止线程或清理某些强 GC 句柄。在这种情况下可以使用 `Unloading` 事件。执行必要清理的处理程序可与此事件挂钩。

解决可卸载性问题

由于卸载的协作性质，很容易忘记可能使内容在可回收的 `AssemblyLoadContext` 中保持活动状态并阻止卸载的引用。以下是可以包含引用实体（其中一些不明显）的摘要：

- 保存于可回收的 `AssemblyLoadContext` 外部、存储在堆栈槽或处理器寄存器（由用户代码显式创建或由实时 (JIT) 编译器隐式创建的方法本地变量）中的常规引用、静态变量或强（固定）GC 句柄以及以可传递的方式指向：
  - 加载到可回收的 `AssemblyLoadContext` 中的程序集。
  - 此类程序集中的类型。
  - 此类程序集中的类型的实例。
- 从加载到可回收的 `AssemblyLoadContext` 程序集中运行代码的线程。
- 在可回收的 `AssemblyLoadContext` 中创建的自定义非可回收 `AssemblyLoadContext` 类型的实例。
- 将回调设置为自定义 `AssemblyLoadContext` 中的方法的挂起 RegisteredWaitHandle 实例。

TIP

在以下情况下可能会出现存储在堆栈槽或处理器寄存器中、可阻止卸载 `AssemblyLoadContext` 的对象引用：

- 当直接将函数调用结果传递给另一个函数时，即使没有用户创建的本地变量。
- 当 JIT 编译器保留对可在方法中的某个点使用的对象的引用时。

调试卸载问题

调试卸载问题可能比较繁琐。你可能会遇到这样的情况：你不知道哪些内容可以使 `AssemblyLoadContext` 保持活动状态，但卸载会失败。帮助解决此问题的最佳武器是带有 SOS 插件的 WinDbg（Unix 上的 LLDB）。需要查找哪些内容使属于特定 `AssemblyLoadContext` 的 `LoaderAllocator` 保持活动状态。SOS 插件可让你查看 GC 堆对象、其层次结构和根。

若要将插件加载到调试器中，请在调试器命令行中输入以下命令：

在 WinDbg（似乎 WinDbg 在进入 .NET Core 应用程序时会自动执行此操作）中：

```
.loadby sos coreclr
```

在 LLDB 中：

```
plugin load /path/to/libsosplugin.so
```

调试卸载时出现问题的示例程序。源代码包含在以下内容中。在 WinDbg 下运行它时，程序将在尝试检查卸载是否成功后立即进入调试器。然后即可开始查找原因。

TIP

如果使用 Unix 上的 LLDB 进行调试，则以下示例中的 SOS 命令的前面没有 `!`。

```
!dumpheap -type LoaderAllocator
```

此命令转储类型名称包含 GC 堆中的 `LoaderAllocator` 的所有对象。下面是一个示例：

```
         Address               MT     Size
000002b78000ce40 00007ffadc93a288       48
000002b78000ceb0 00007ffadc93a218       24

Statistics:
              MT    Count    TotalSize Class Name
00007ffadc93a218        1           24 System.Reflection.LoaderAllocatorScout
00007ffadc93a288        1           48 System.Reflection.LoaderAllocator
Total 2 objects
```

在下面的"Statistics:"部分中，检查属于 `System.Reflection.LoaderAllocator` 的 `MT`（`MethodTable`），这是我们关注的对象。然后在开头的列表中，查找 `MT` 匹配该条目的条目，并获取对象本身的地址。在示例中，其为"000002b78000ce40"。

现在我们知道了 `LoaderAllocator` 对象的地址，可使用另一个命令来查找其 GC 根：

```
!gcroot -all 0x000002b78000ce40
```

此命令转储通向 `LoaderAllocator` 实例的对象引用链。该列表以根（使 `LoaderAllocator` 保持活动状态的实体）开头，因此为问题的核心。根可以是堆栈槽、处理器寄存器、GC 句柄或静态变量。

以下是 `gcroot` 命令输出的示例：

```
Thread 4ac:
    000000cf9499dd20 00007ffa7d0236bc example.Program.Main(System.String[])
[E:\unloadability\example\Program.cs @ 70]
        rbp-20: 000000cf9499dd90
            ->  000002b78000d328 System.Reflection.RuntimeMethodInfo
            ->  000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
            ->  000002b78000d1d0 System.RuntimeType
            ->  000002b78000ce40 System.Reflection.LoaderAllocator

HandleTable:
    000002b7f8a81198 (strong handle)
    -> 000002b78000d948 test.Test
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

    000002b7f8a815f8 (pinned handle)
    -> 000002b790001038 System.Object[]
    -> 000002b78000d390 example.TestInfo
    -> 000002b78000d328 System.Reflection.RuntimeMethodInfo
    -> 000002b78000d1f8 System.RuntimeType+RuntimeTypeCache
    -> 000002b78000d1d0 System.RuntimeType
    -> 000002b78000ce40 System.Reflection.LoaderAllocator

Found 3 roots.
```

下一步是确定根所在的位置以便对其进行修复。最简单的情况是根为堆栈槽或处理器寄存器。在这种情况下，`gcroot` 会显示其框架包含根的函数的名称以及执行该函数的线程。困难的情况是根为静态变量或 GC 句柄。

在上面的示例中，第一个根为 `System.Reflection.RuntimeMethodInfo` 类型的本地变量，存储在函数 `example.Program.Main(System.String[])` 的框架中，地址为 `rbp-20`（`rbp` 意为处理器寄存器 `rbp`，-20 意为该寄存器的十六进制偏移量）。

第二个根为普通（强）`GCHandle`，其包含对 `test.Test` 类实例的引用。

第三个根为固定的 `GCHandle`。这实际上是一个静态变量，但遗憾的是，无法进行澄清。引用类型的静态变量存储在内部运行时结构中的托管对象数组中。

另一种可阻止卸载 `AssemblyLoadContext` 的情况为，线程具有来源于加载到其堆栈上的 `AssemblyLoadContext` 程序集的方法框架。可通过转储所有线程的托管调用堆栈进行检查：

```
~*e !clrstack
```

此命令表示“将 `!clrstack` 命令应用于所有线程”。以下是此示例中该命令的输出。遗憾的是，Unix 上的 LLDB 无法将命令应用于所有线程，因此，必须手动切换线程并重复 `clrstack` 命令。忽略调试器显示“无法审核托管堆栈”的所有线程。

```
OS Thread Id: 0x6ba8 (0)
        Child SP               IP Call Site
0000001fc697d5c8 00007ffb50d9de12 [HelperMethodFrame: 0000001fc697d5c8]
System.Diagnostics.Debugger.BreakInternal()
0000001fc697d6d0 00007ffa864765fa System.Diagnostics.Debugger.Break()
0000001fc697d700 00007ffa864736bc example.Program.Main(System.String[]) [E:\unloadability\example\Program.cs
@ 70]
0000001fc697d998 00007ffae5fdc1e3 [GCFrame: 0000001fc697d998]
0000001fc697df28 00007ffae5fdc1e3 [GCFrame: 0000001fc697df28]
OS Thread Id: 0x2ae4 (1)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x61a4 (2)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x7fdc (3)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5390 (4)
Unable to walk the managed stack. The current thread is likely not a
managed thread. You can run !threads to get a list of managed threads in
the process
Failed to start stack walk: 80070057
OS Thread Id: 0x5ec8 (5)
        Child SP               IP Call Site
0000001fc70ff6e0 00007ffb5437f6e4 [DebuggerU2MCatchHandlerFrame: 0000001fc70ff6e0]
OS Thread Id: 0x4624 (6)
        Child SP               IP Call Site
GetFrameContext failed: 1
0000000000000000 0000000000000000
OS Thread Id: 0x60bc (7)
        Child SP               IP Call Site
0000001fc727f158 00007ffb5437fce4 [HelperMethodFrame: 0000001fc727f158]
System.Threading.Thread.SleepInternal(Int32)
0000001fc727f260 00007ffb37ea7c2b System.Threading.Thread.Sleep(Int32)
0000001fc727f290 00007ffa865005b3 test.Program.ThreadProc() [E:\unloadability\test\Program.cs @ 17]
0000001fc727f2c0 00007ffb37ea6a5b System.Threading.Thread.ThreadMain_ThreadStart()
0000001fc727f2f0 00007ffadbc4cbe3
System.Threading.ExecutionContext.RunInternal(System.Threading.ExecutionContext,
System.Threading.ContextCallback, System.Object)
0000001fc727f568 00007ffae5fdc1e3 [GCFrame: 0000001fc727f568]
0000001fc727f7f0 00007ffae5fdc1e3 [DebuggerU2MCatchHandlerFrame: 0000001fc727f7f0]
```

如你所见，最后一个线程具有 `test.Program.ThreadProc()`。这是从加载到 `AssemblyLoadContext` 的程序集中的函数，因此它使 `AssemblyLoadContext` 保持活动状态。

具有可卸载性问题的示例源

上面的调试示例中使用了以下代码。

主要测试程序

```
using System;
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.Loader;

namespace example
```

```
{
    class TestAssemblyLoadContext : AssemblyLoadContext
    {
        public TestAssemblyLoadContext() : base(true)
        {
        }
        protected override Assembly Load(AssemblyName name)
        {
            return null;
        }
    }

    class TestInfo
    {
        public TestInfo(MethodInfo mi)
        {
            entryPoint = mi;
        }
        MethodInfo entryPoint;
    }

    class Program
    {
        static TestInfo entryPoint;

        [MethodImpl(MethodImplOptions.NoInlining)]
        static int ExecuteAndUnload(string assemblyPath, out WeakReference testAlcWeakRef, out MethodInfo
testEntryPoint)
        {
            var alc = new TestAssemblyLoadContext();
            testAlcWeakRef = new WeakReference(alc);

            Assembly a = alc.LoadFromAssemblyPath(assemblyPath);
            if (a == null)
            {
                testEntryPoint = null;
                Console.WriteLine("Loading the test assembly failed");
                return -1;
            }

            var args = new object[1] {new string[] {"Hello"}};

            // Issue preventing unloading #1 - we keep MethodInfo of a method for an assembly loaded into
the TestAssemblyLoadContext in a static variable
            entryPoint = new TestInfo(a.EntryPoint);
            testEntryPoint = a.EntryPoint;

            int result = (int)a.EntryPoint.Invoke(null, args);
            alc.Unload();

            return result;
        }

        static void Main(string[] args)
        {
            WeakReference testAlcWeakRef;
            // Issue preventing unloading #2 - we keep MethodInfo of a method for an assembly loaded into
the TestAssemblyLoadContext in a local variable
            MethodInfo testEntryPoint;
            int result = ExecuteAndUnload(@"absolute/path/to/test.dll", out testAlcWeakRef, out
testEntryPoint);

            for (int i = 0; testAlcWeakRef.IsAlive && (i < 10); i++)
            {
                GC.Collect();
                GC.WaitForPendingFinalizers();
            }

            System.Diagnostics.Debugger.Break();
```

```
            System.Diagnostics.Debugger.Break();

            Console.WriteLine($"Test completed, result={result}, entryPoint: {testEntryPoint} unload
success: {!testAlcWeakRef.IsAlive}");
        }
    }
}
```

程序已加载到 TestAssemblyLoadContext 中

以下代码表示传递给主测试程序中 `ExecuteAndUnload` 方法的 test.dll。

```
using System;
using System.Runtime.InteropServices;
using System.Threading;

namespace test
{
    class Test
    {
        string message = "Hello";
    }

    class Program
    {
        public static void ThreadProc()
        {
            // Issue preventing unloading #4 - a thread running method inside of the TestAssemblyLoadContext
at the unload time
            Thread.Sleep(Timeout.Infinite);
        }

        static GCHandle handle;
        static int Main(string[] args)
        {
            // Issue preventing unloading #3 - normal GC handle
            handle = GCHandle.Alloc(new Test());
            Thread t = new Thread(new ThreadStart(ThreadProc));
            t.IsBackground = true;
            t.Start();
            Console.WriteLine($"Hello from the test: args[0] = {args[0]}");

            return 1;
        }
    }
}
```

.NET 的版本控制方式概述

2021/11/16 •

.NET 运行时和 .NET SDK 添加新功能的频率不同。通常来说，SDK 的更新频率比运行时高。本文介绍运行时和 SDK 版本号。

.NET 每年 11 月都会发布一个新的主版本。偶数版本（如 .NET 6 或 .NET 8）长期受支持 (LTS)。奇数版本在下一个主版本发布之前受支持。.NET 的最新版本是 .NET 5。

版本控制详细信息

.NET 运行时具有用于版本控制的 major.minor.patch 方法，且该方法遵循语义版本控制。

但是，.NET SDK 不遵循语义版本控制。.NET SDK 发布速度更快，其版本必须显示相应的运行时和 SDK 自己的次要版本及补丁版本。

.NET SDK 版本号的前两个位置与随之一起发布的 .NET 运行时版本一致。每个版本的 SDK 都可以为此版本或任何更低版本的运行时创建应用程序。

SDK 版本号的第三个位置同时传达次要编号和修补程序编号。次要版本乘以 100。最后两位数代表修补程序号。次要版本 1，修补程序版本 2 将表示为 102。例如，下面是运行时和 SDK 版本号可能出现的序列：

[illegible]	.NET [illegible]	.NET SDK (*)
初始版本	5.0.0	5.0.100
SDK 修补程序	5.0.0	5.0.101
运行时和 SDK 修补程序	5.0.1	5.0.102
SDK 功能更改	5.0.1	5.0.200

注意：

- 如果在运行时功能更新之前，SDK 有 10 个功能更新，则版本号将滚动到 1000 系列。版本 5.0.900 之后将是版本 5.0.1000。应该不会出现这种情况。
- 不会出现为发布功能的 99 修补程序版本。如果某版本接近此数字，则会强制发布功能。

可在 dotnet/设计 存储库中查看初始建议的更多详细信息。

语义化版本控制

.NET 运行时大致遵循语义版本控制 (SemVer)，采用 `MAJOR.MINOR.PATCH` 版本控制，通过版本号的各部分来描述更改程度和类型。

```
MAJOR.MINOR.PATCH[-PRERELEASE-BUILDNUMBER]
```

可选的 `PRERELEASE` 和 `BUILDNUMBER` 部分永远不会成为受支持版本的一部分，并且将仅存在于夜间版本、来自源目标的本地版本，以及不受支持的预览版本中。

了解运行时版本号更改

- `MAJOR` 每年递增一次，可能包含：
  - 产品或新产品方向发生的重大更改。
  - API 引入的中断性变更。接受中断性变更存在较大障碍。
  - 采用了现有依赖项的较新 `MAJOR` 版本。

  主版本每年发布一次，偶数版本是长期支持的 (LTS) 版本。使用此版本控制方案的第一个 LTS 版本是 .NET 6，该版本将于 2021 年 11 月发布。最新的非 LTS 版本是 .NET 5。
- `MINOR` 在下列情况时递增：
  - 添加了公共 API 外围应用。
  - 添加了新行为。
  - 采用了现有依赖项的较新 `MINOR` 版本。
  - 引入了新依赖项。
- `PATCH` 在下列情况时递增：
  - 进行了 Bug 修复。
  - 添加了对较新平台的支持。
  - 采用了现有依赖项的较新 `PATCH` 版本。
  - 任何其他不符合上述情况的更改。

存在多处更改时，单个更改影响的最高级别元素会递增，并将随后的元素重置为零。例如，当 `MAJOR` 递增时，`MINOR.PATCH` 将重置为零。当 `MINOR` 递增时，`PATCH` 将重置为零，而 `MAJOR` 保持不变。

文件名中的版本号

为 .NET 下载的文件带有版本，例如 `dotnet-sdk-5.0.301-win10-x64.exe`。

预览版

预览版向版本号追加了 `-preview.[number].[build]`。例如 `6.0.0-preview.5.21302.13`。

服务版本

在版本发布后，版本分支通常停止生成日常版本，而开始生成服务版本。服务版本向版本追加了 `-servicing-[number]`。例如 `5.0.1-servicing-006924`。

请参阅

- 目标框架
- .NET 分发打包
- .NET 支持生命周期简报
- .NET 的 Docker 映像

选择要使用的 .NET 版本

2021/11/16 •

本文介绍了 .NET 工具、SDK 和运行时用来选择版本的策略。这些策略可通过使用指定版本，使正在运行的应用程序之间达到平衡，同时实现开发人员和最终用户计算机的轻松升级。通过这些策略可实现：

- 简单高效的 .NET 部署，包括安全性和可靠性更新。
- 使用独立于目标运行时的最新工具和命令。

需要选择版本的情况如下：

- 运行 SDK 命令时，SDK 使用最新安装的版本。
- 生成程序集时，目标框架名字对象定义生成时 API。
- 运行 .NET 应用程序时，依赖于目标框架的应用会前滚。
- 发布独立应用程序时，独立部署包括所选的运行时。

本文档其余部分将介绍这四种方案。

SDK 使用最新安装的版本

SDK 命令包括 `dotnet new` 和 `dotnet run`。.NET CLI 必须为每个 `dotnet` 命令选择一个 SDK 版本。即使在以下情况下，它也会默认使用计算机上安装的最新 SDK：

- 项目以旧版 .NET 运行时为目标。
- .NET SDK 的最新版本是预览版。

你可以利用最新的 SDK 功能和改进，同时以较旧的 .NET 运行时版本为目标。可以使用相同的 SDK 工具面向不同运行时版本的 .NET。

在少数情况下，可能需要使用版本较旧的 SDK。在 global.json 文件中指定该版本。"使用最新"策略表示仅使用 global.json 指定低于最新安装版本的 .NET SDK 版本。

可将 global.json 放置在文件层次结构中的任何位置。CLI 从项目目录中向上搜索其找到的第一个 global.json。由用户控制对哪些项目应用给定的 global.json（按其在文件系统中的位置）。.NET CLI 从当前工作目录路径向上导航，以迭代方式搜索 global.json 文件。找到的第一个 global.json 文件指定要使用的版本。如果已安装该 SDK 版本，则使用该版本。如果找不到 global.json 中指定的 SDK，则 .NET CLI 将使用匹配规则来选择兼容的 SDK，如果找不到，则会失败。

下面的示例演示 global.json 语法：

```
{
  "sdk": {
    "version": "5.0.0"
  }
}
```

选择 SDK 版本的过程如下：

1. `dotnet` 从当前工作目录向下导航路径，以迭代方式搜索 global.json 文件。
2. `dotnet` 使用所找到的第一个 global.json 中指定的 SDK。
3. 如果未找到 global.json，`dotnet` 使用最新安装的 SDK。

有关 SDK 版本选择的信息，请参阅 global.json 概述一文中的匹配规则和 rollForward 部分。

目标框架名字对象用于定义生成时 API

针对在"目标框架名字对象"(TFM) 中定义的 API 构建项目。在项目文件中指定目标框架。按如下示例所示，设置项目文件中的 `TargetFramework` 元素：

```
<TargetFramework>net5.0</TargetFramework>
```

可能会针对多个 TFM 构建项目。对库设置多个目标框架更为常见，但也可对应用程序执行此操作。指定 `TargetFrameworks` 属性（`TargetFramework` 的复数形式）。目标框架以分号分隔，如下例所示：

```
<TargetFrameworks>net5.0;netcoreapp3.1;net47</TargetFrameworks>
```

给定的 SDK 支持固定的一组框架，其中的上限框架为 SDK 附带的运行时的目标框架。例如，.NET 5 SDK 包含 NET 5 运行时，后者是 `net5.0` 目标框架的实现。.NET 5 SDK 支持 `netcoreapp2.0`、`netcoreapp2.1` 和 `netcoreapp3.0` 等，但不支持 `net6.0`（或更高版本）。需要安装 .NET 6 SDK 来执行 `net6.0` 的生成。

.NET Standard

.NET Standard 是一种面向某个 API 图面的方法，该图面由 .NET 的不同实现所共享。从版本 .NET 5（其本身就是一个 API 标准）开始，就几乎与 .NET Standard 无关了，但一种情况除外：当需要同时面向 .NET 和 .NET Framework 时，.NET Standard 会有用。.NET 5 实现所有 .NET Standard 版本。

有关详细信息，请参阅 .NET 5 和 .NET Standard。

依赖于框架的应用会前滚

在使用 `dotnet run` 从源运行应用程序时，在使用 `dotnet myapp.dll` 从框架相关部署运行应用程序时，或使用 `myapp.exe` 从框架相关可执行文件运行应用程序时，`dotnet` 可执行文件是应用程序的主机。

该主机选择计算机上安装的最新修补程序版本。例如，如果在项目文件中指定 `net5.0`，并且 `5.0.2` 是安装的最新 .NET 运行时，则使用 `5.0.2` 运行时。

如果未找到可接受的 `5.0.*` 版本，则使用新的 `5.*` 版本。例如，如果指定了 `net5.0` 并且仅安装了 `5.1.0`，则应用程序在运行时使用 `5.1.0` 运行时。此行为称为"次要版本前滚"。此外，不会考虑较低版本。如果未安装可接受的运行时，应用程序将不会运行。

下面几个使用示例展示了在面向版本 5.0 的情况下的此行为：

- ✔ 指定了 5.0。5.0.3 是安装的最高修补程序版本。使用了 5.0.3。
- ✘ 指定了 5.0。未安装 5.0.* 版本。3.1.1 是安装的最高运行时版本。会显示一条错误消息。
- ✔ 指定了 5.0。未安装 5.0.* 版本。5.1.0 是安装的最高运行时版本。使用了 5.1.0。
- ✘ 指定 3.0。未安装 3.x 版本。5.0.0 是安装的最高运行时版本。会显示一条错误消息。

次要版本回滚会产生一个可能影响最终用户的副作用。请参考以下方案：

1. 应用程序指定需要版本 5.0。
2. 运行时，未安装版本 5.0.*，安装的是 5.1.0。将使用版本 5.1.0。
3. 稍后，用户重新安装 5.0.3 和运行应用程序，而现将使用版本 5.0.3。

5.0.3 和 5.1.0 可能具有不同行为，序列化二进制数据等方案中尤其如此。

控制前滚行为

可通过四种不同的方式配置应用程序的前滚行为：

1. 通过设置 `<RollForward>` 属性来进行项目级设置：

```
<PropertyGroup>
  <RollForward>LatestMinor</RollForward>
</PropertyGroup>
```

2. `*.runtimeconfig.json` 文件。

   编译应用程序时，将生成此文件。如果项目中设置了 `<RollForward>` 属性，`*.runtimeconfig.json` 文件中会将重新生成为设置 `rollForward`。用户可以编辑此文件以更改应用程序的行为。

```
{
  "runtimeOptions": {
    "tfm": "net5.0",
    "rollForward": "LatestMinor",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "5.0.0"
    }
  }
}
```

3. `dotnet` 命令的 `--roll-forward <value>` 属性。

   运行应用程序时，可以通过命令行控制前滚行为：

```
dotnet run --roll-forward LatestMinor
dotnet myapp.dll --roll-forward LatestMinor
myapp.exe --roll-forward LatestMinor
```

4. `DOTNET_ROLL_FORWARD` 环境变量。

优先级

运行应用时，前滚行为按以下顺序设置，编号较高的项优先于编号较低的项：

1. 首先计算 `*.runtimeconfig.json` 配置文件。
2. 接下来，考虑 `DOTNET_ROLL_FORWARD` 环境变量，替代前面的检查。
3. 最后，传递给正在运行的应用程序的任何 `--roll-forward` 参数将替代其他所有内容。

值

无论采用什么前滚设置，请使用下列值之一设置该行为：

“[”	[[
`Minor`	如果未指定，则为默认值。 如果缺少所请求的次要版本，则前滚到最低的较高次要版本。 如果存在所请求的次要版本，则使用 `LatestPatch` 策略。
`Major`	如果缺少所请求的主要版本，则前滚到下一个可用的更高主要版本和最低的次要版本。如果存在所请求的主要版本，则使用 `Minor` 策略。
`LatestPatch`	前滚到最高补丁版本。此值会禁用次要版本前滚。
`LatestMinor`	即使存在所请求的次要版本，仍前滚到最高次要版本。

"["	[[
`LatestMajor`	即使存在所请求的主要版本，仍前滚到最高主要版本和最高次要版本。
`Disable`	不要前滚，仅绑定到指定的版本。建议不要将此策略用于一般用途，因为它会禁用前滚到最新补丁的功能。该值仅建议用于测试。

独立部署包括所选的运行时

可以将应用程序作为独立分发进行发布。这种方法将 .NET 运行时和库捆绑到应用程序。独立部署不具有对运行时环境的依赖关系。在发布时（而不是运行时）选择运行时版本。

当发布选择了给定运行时系列的最新修补程序版本时所发生的"还原"事件。例如，如果 .NET 5.0.3 是 .NET 5 运行时系列中的最新修补程序版本，`dotnet publish` 将选择该版本。目标框架（包括最新安装的安全修补程序）与应用程序捆绑打包。

如果为应用程序指定的最小版本不满足要求，会出现错误。`dotnet publish` 绑定到最新的运行时修补程序版本（在给定的主要及次要版本系列内）。`dotnet publish` 不支持 `dotnet run` 的前滚语义。若要详细了解补丁和独立式部署，请参阅关于在部署 .NET 应用程序时选择运行时补丁的文章。

独立部署可能需要特定修补程序版本。可以重写项目文件中的最低运行时修补程序版本（重写为更高或更低版本），如下例所示：

```
<PropertyGroup>
  <RuntimeFrameworkVersion>5.0.7</RuntimeFrameworkVersion>
</PropertyGroup>
```

`RuntimeFrameworkVersion` 元素重写默认版本策略。对于独立部署，`RuntimeFrameworkVersion` 指定确切的运行时框架版本。对于依赖于框架的应用程序，`RuntimeFrameworkVersion` 指定所需的最低运行时框架版本。

请参阅

- 下载并安装 .NET。
- 如何删除 .NET 运行时和 SDK。

.NET 运行时配置设置

2021/11/16 •

.NET 5+（包括 .NET Core 版本）支持使用配置文件和环境变量在运行时配置 .NET 应用程序的行为。如果出现以下情况，则运行时配置是一个不错的选择：

- 你不拥有或控制应用程序的源代码，因此无法以编程方式对其进行配置。
- 应用程序的多个实例在单个系统上同时运行，并且你想要将每个实例配置为获得最佳性能。

> **NOTE**
> 本文档正在编写中。如果你注意到此处提供的信息不完整或不准确，可以创建一个问题告知我们，或提交拉取请求以解决问题。要了解如何提交 dotnet/docs 存储库的拉取请求，请参阅参与者指南。

.NET 提供以下机制，它们用于配置运行时应用程序行为：

- runtimeconfig.json 文件
- MSBuild 属性
- 环境变量

> **TIP**
> 如果使用环境变量配置运行时选项，会将设置应用于所有 .NET 应用。如果在 runtimeconfig.json 或项目文件中配置运行时选择，则仅将设置应用于此应用程序。

某些配置值还可以通过调用 AppContext.SetSwitch 方法以编程方式进行设置。

文档此部分的文章按类别组织，例如调试和垃圾回收。如果适用，将显示 runtimeconfig.json 文件、MSBuild 属性、环境变量的配置选项；对于 .NET Framework 项目，还会显示 app.config 文件的配置选项以便交叉引用。

runtimeconfig.json

构建项目时，将在输出目录中生成 *[appname].runtimeconfig.json* 文件。如果项目文件所在的文件夹中存在 *runtimeconfig.template.json* 文件，它包含的任何配置选项都将插入到 *[appname].runtimeconfig.json* 文件中。如果自行构建应用，请将所有配置选项放在 *runtimeconfig.template.json* 文件中。如果只是运行应用，请将其直接插入 [appname].runtimeconfig.template.json 文件中。

> **NOTE**
> - 后续生成中将覆盖 [appname].runtimeconfig.template.json 文件。
> - 如果应用的 `OutputType` 不是 `Exe`，但你想将配置选项从 runtimeconfig.template.json 复制到 [应用名称].runtimeconfig.json，则必须在项目文件中将 `GenerateRuntimeConfigurationFiles` 显式设置为 `true` 。对于需要 runtimeconfig.json 文件的应用，此属性默认设置为 `true` 。

在 runtimeconfig.json 文件的 configProperties 部分指定运行时配置选项。此部分包含窗体：

```
"configProperties": {
  "config-property-name1": "config-value1",
  "config-property-name2": "config-value2"
}
```

示例 [appname].runtimeconfig.template.json 文件

如果要将这些选项放在输出 JSON 文件中，请将它们嵌套在 `runtimeOptions` 属性下。

```
{
  "runtimeOptions": {
    "tfm": "netcoreapp3.1",
    "framework": {
      "name": "Microsoft.NETCore.App",
      "version": "3.1.0"
    },
    "configProperties": {
      "System.GC.Concurrent": false,
      "System.Threading.ThreadPool.MinThreads": 4,
      "System.Threading.ThreadPool.MaxThreads": 25
    }
  }
}
```

示例 runtimeconfig.template.json 文件

如果要将这些选项放在模板 JSON 文件中，请省略 `runtimeOptions` 属性。

```
{
  "configProperties": {
    "System.GC.Concurrent": false,
    "System.Threading.ThreadPool.MinThreads": "4",
    "System.Threading.ThreadPool.MaxThreads": "25"
  }
}
```

MSBuild 属性

可使用 SDK 样式 .NET Core 项目的 .csproj 或 .vbproj 文件中的 MSBuild 属性设置某些运行时配置选项。 MSBuild 属性优先于在 *runtimeconfig.template.json* 文件中设置的选项。

下面是一个示例 SDK 样式项目文件，其中包含用于配置运行时行为的 MSBuild 属性：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
    <ThreadPoolMaxThreads>25</ThreadPoolMaxThreads>
  </PropertyGroup>

</Project>
```

用于配置运行时行为的 MSBuild 属性记录在每个区域各自的文章中，例如垃圾回收。它们还在 SDK 样式项目的 MSBuild 属性参考的运行时配置部分中列出。

环境变量

环境变量可用于提供一些运行时配置信息。如果使用环境变量配置运行时选项，会将设置应用于所有 .NET Core 应用。指定为环境变量的配置旋钮通常带有 DOTNET_ 前缀。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

可以使用 Windows 控制面板、命令行或通过在 Windows 和 Unix 系统上调用 Environment.SetEnvironmentVariable(String, String) 方法以编程方式定义环境变量。

下面的示例演示如何在命令行中设置环境变量：

```
# Windows
set DOTNET_GCRetainVM=1

# Powershell
$env:DOTNET_GCRetainVM="1"

# Unix
export DOTNET_GCRetainVM=1
```

另请参阅

- .NET environment variables](../tools/dotnet-environment-variables.md)

用于编译的运行时配置选项

2021/11/16 •

本文详细介绍可用于配置 .NET 编译的设置。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

分层编译

- 配置实时 (JIT) 编译器是否使用分层编译。分层编译将方法转换到两个层级：
  - 第一层可以更快速地生成代码（快速 JIT）或加载预编译的代码 (ReadyToRun)。
  - 第二层在后台生成优化的代码（“优化 JIT”）。
- 在 .NET Core 3.0 及更高版本中，默认情况下已启用分层编译。
- 在 .NET Core 2.1 和 2.2 中，默认情况下已禁用分层编译。
- 有关详细信息，请参阅分层编译指南。

runtimeconfig.json	`System.Runtime.TieredCompilation`	`true` - 启用 `false` - 禁用
MSBuild 属性	`TieredCompilation`	`true` - 启用 `false` - 禁用
	`COMPlus_TieredCompilation` 或 `DOTNET_TieredCompilation`	`1` - 启用 `0` - 禁用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation": false
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilation>false</TieredCompilation>
  </PropertyGroup>

</Project>
```

快速 JIT

- 配置 JIT 编译器是否使用快速 JIT。对于不包含循环且不可使用预编译代码的方法，快速 JIT 可以更快完成编译，但不会进行优化。
- 启用快速 JIT 会缩短启动时间，但可能会生成性能下降的代码。例如，代码可能会使用更多堆栈空间、分配更多内存并以更慢的速度运行。
- 如果禁用了快速 JIT 但启用了分层编译，则只有预编译的代码参与分层编译。如果未使用 ReadyToRun 预编译方法，则 JIT 行为与禁用分层编译时相同。
- 在 NET Core 3.0 及更高版本中，默认启用快速 JIT。
- 在 NET Core 2.1 和 2.2 中，默认禁用快速 JIT。

runtimeconfig.json	`System.Runtime.TieredCompilation.Quic`	`true` - 启用 `false` - 禁用
MSBuild 属性	`TieredCompilationQuickJit`	`true` - 启用 `false` - 禁用
	`COMPlus_TC_QuickJit` 或 `DOTNET_TC_QuickJit`	`1` - 启用 `0` - 禁用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJit": false
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJit>false</TieredCompilationQuickJit>
  </PropertyGroup>

</Project>
```

适用于循环的快速 JIT

- 配置 JIT 编译器是否对包含循环的方法使用快速 JIT。

- 启用适用于循环的快速 JIT 可以提高启动性能。不过，在优化程度较低的代码中，长时间运行的循环可能会停滞较长时间。
- 如果禁用快速 JIT，则此设置不起作用。
- 如果省略此设置，则不会对包含循环的方法使用“快速 JIT”。它等效于将值设置为 `false`。

runtimeconfig.json	`System.Runtime.TieredCompilation.QuickJitForLoops`	`false` - 禁用 `true` - 启用
MSBuild 属性	`TieredCompilationQuickJitForLoops`	`false` - 禁用 `true` - 启用
[redacted]	`COMPlus_TC_QuickJitForLoops` 或 `DOTNET_TC_QuickJitForLoops`	`0` - 禁用 `1` - 启用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Runtime.TieredCompilation.QuickJitForLoops": false
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TieredCompilationQuickJitForLoops>true</TieredCompilationQuickJitForLoops>
  </PropertyGroup>

</Project>
```

ReadyToRun

- 配置 .NET Core 运行时是否要为具有可用 ReadyToRun 数据的映像使用预编译代码。如果禁用此选项，会强制运行时对框架代码进行 JIT 编译。
- 有关详细信息，请参阅准备好运行。
- 如果省略此设置，则 .NET 将使用 ReadyToRun 数据（如果可用）。它等效于将值设置为 `1`。

[redacted]	`COMPlus_ReadyToRun` 或 `DOTNET_ReadyToRun`	`1` - 启用 `0` - 禁用

按配置优化

此设置在 .NET 6 及更高版本中 (PGO) 启用动态或分层按配置优化。

	[illegible]	[illegible]
[illegible]	`DOTNET_TieredPGO`	`1` - 启用 `0` - 禁用

用于调试和分析的运行时配置选项

2021/11/16 •

本文详细介绍可用于配置 .NET 调试和分析的设置。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

启用诊断

- 配置是启用还是禁用调试器、探查器和 EventPipe 诊断。
- 如果省略此设置，则会启用诊断。它等效于将值设置为 `1`。

runtimeconfig.json	不可用	不可用
■	`COMPlus_EnableDiagnostics` 或 `DOTNET_EnableDiagnostics`	`1` - 启用 `0` - 禁用

启用分析

- 配置是否为当前正在运行的进程启用分析。
- 如果省略此设置，则会禁用分析。它等效于将值设置为 `0`。

runtimeconfig.json	不可用	不可用
■	`CORECLR_ENABLE_PROFILING`	`0` - 禁用 `1` - 启用

探查器 GUID

- 指定要加载到当前正在运行的进程中的探查器 GUID。

runtimeconfig.json	不可用	不可用
■	`CORECLR_PROFILER`	string-guid

探查器位置

- 指定要加载到当前正在运行的进程（或 32 位/64 位进程）的探查器 DLL 路径。

- 如果设置了多个变量，则优先使用指定位数的变量。它们指定要加载的探查器的位数。
- 有关详细信息，请参阅 Finding the profiler library（查找探查器库）。

■	`CORECLR_PROFILER_PATH`	string-path
■	`CORECLR_PROFILER_PATH_32`	string-path
■	`CORECLR_PROFILER_PATH_64`	string-path

写入 Perf 映射

- 允许或禁止在 Linux 系统上写入 /tmp/perf-$pid.map。
- 如果省略此设置，则会禁止写入 Perf 映射。它等效于将值设置为 `0`。

runtimeconfig.json	不可用	不可用
■	`COMPlus_PerfMapEnabled` 或 `DOTNET_PerfMapEnabled`	`0` - 禁用 `1` - 启用

性能日志标记

- 允许或禁止在性能日志中将指定信号作为标记予以接受和忽略。
- 如果省略此设置，则不会忽略指定的信号。它等效于将值设置为 `0`。

runtimeconfig.json	不可用	不可用
■	`COMPlus_PerfMapIgnoreSignal` 或 `DOTNET_PerfMapIgnoreSignal`	`0` - 禁用 `1` - 启用

> **NOTE**
> 如果省略 DOTNET_PerfMapEnabled 或将其设置为 `0`（即禁用），则将忽略此设置。

用于垃圾回收的运行时配置选项

2021/11/16 •

此页包含有关可在运行时更改的垃圾回收器 (GC) 设置的信息。如果你要尝试让正在运行的应用达到最佳性能，请考虑使用这些设置。然而，在特定情况下，默认值为大多数应用程序提供最佳性能。

设置在此页上被排入组中。每个组内的设置通常彼此结合使用以实现特定的结果。

> **NOTE**
>
> - 这些设置也可以在应用运行时由应用动态地进行更改，因此，你设置的任何运行时设置都可能会被覆盖。
> - 某些设置（如延迟级别）通常仅在设计时通过 API 进行设置。此页面省略了此类设置。
> - 对于数值，请对 runtimeconfig.json 文件中的设置使用十进制表示法，而对环境变量设置使用十六进制表示法。对于十六进制值，可以使用或不使用“0x”前缀来指定它们。
> - 如果使用环境变量，则 .NET 6 将标准化前缀 `DOTNET_`，而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则仍应该使用 `COMPlus_` 前缀，例如 `COMPlus_gcServer`。

垃圾回收的风格

垃圾回收的两种主要风格是工作站 GC 和服务器 GC。有关两者之间的差异的详细信息，请参阅工作站和服务器垃圾回收。

垃圾回收的次要风格是后台垃圾回收和非并发垃圾回收。

使用以下设置，选择垃圾回收的风格：

- 工作站与服务器 GC
- 后台垃圾回收

工作站与服务器

- 配置应用程序是使用工作站垃圾回收还是服务器垃圾回收。
- 默认：工作站垃圾回收。它等效于将值设置为 `false`。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.Server`	`false` - 工作站 `true` - 服务器	.NET Core 1.0
MSBuild 属性	`ServerGarbageCollection`	`false` - 工作站 `true` - 服务器	.NET Core 1.0
[illegible]	`COMPlus_gcServer`	`0` - 工作站 `1` - 服务器	.NET Core 1.0
[illegible]	`DOTNET_gcServer`	`0` - 工作站 `1` - 服务器	.NET 6
.NET Framework [illegible] app.config	GCServer	`false` - 工作站 `true` - 服务器	

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Server": true
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ServerGarbageCollection>true</ServerGarbageCollection>
  </PropertyGroup>

</Project>
```

后台垃圾回收

- 配置是否启用后台(并发)垃圾回收。
- 默认:使用后台垃圾回收。它等效于将值设置为 `true`。
- 有关详细信息，请参阅后台垃圾回收。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.Concurrent`	`true` - 后台 GC `false` - 非并发 GC	.NET Core 1.0
MSBuild 属性	`ConcurrentGarbageCollection`	`true` - 后台 GC `false` - 非并发 GC	.NET Core 1.0
[illegible]	`COMPlus_gcConcurrent`	`1` - 后台 GC `0` - 非并发 GC	.NET Core 1.0
[illegible]	`DOTNET_gcConcurrent`	`1` - 后台 GC `0` - 非并发 GC	.NET 6
.NET Framework [illegible] app.config	gcConcurrent	`true` - 后台 GC `false` - 非并发 GC	

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.Concurrent": false
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
  </PropertyGroup>

</Project>
```

管理资源使用情况

使用以下设置来管理垃圾回收器的内存和处理器使用情况：

- 关联
- 关联掩码
- 关联范围
- CPU 组
- 堆计数
- 堆限制
- 堆限制百分比
- 高内存百分比
- 每对象堆限制
- 每对象堆限制百分比
- 保留 VM

有关其中某些设置的详细信息，请参阅 Middle ground between workstation and server GC（服务器和工作站 GC 之间的中间地带）博客条目。

堆计数

- 限制通过垃圾回收器创建的堆数。
- 仅适用于服务器垃圾回收。
- 如果启用了默认的 GC 处理器关联，堆计数设置会将 `n` 个 GC 堆/线程关联到前 `n` 个处理器。（使用关联掩码或关联范围设置可精确指定要关联的处理器。）
- 如果禁用了 GC 处理器关联，则此设置会限制 GC 堆的数量。
- 有关详细信息，请参阅 GCHeapCount 备注。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapCount`	十进制值	.NET Core 3.0
[illegible]	`COMPlus_GCHeapCount`	十六进制值	.NET Core 3.0
[illegible]	`DOTNET_GCHeapCount`	十六进制值	.NET 6
.NET Framework [illegible] app.config	GCHeapCount	十进制值	.NET Framework 4.6.2

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapCount": 16
      }
   }
}
```

> **TIP**
> 如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆数限制为 16，则该值对于 JSON 文件为 16，对于环境变量则为 0x10 或 10。

关联掩码

- 指定垃圾回收器线程应使用的确切处理器数。
- 如果禁用了 GC 处理器关联，则忽略此设置。
- 仅适用于服务器垃圾回收。
- 该值是一个位掩码，用于定义可用于该进程的处理器。例如，十进制值 1023 或十六进制值 0x3FF 或 3FF（如果使用环境变量）在二进制记数法中为 0011 1111 1111。这指定将使用前 10 个处理器。若要指定接下来使用的 10 个处理器（即处理器 10-19），请指定一个十进制值 1047552（或十六进制值 0xFFC00 或 FFC00），它等效于二进制值 1111 1111 1100 0000 0000。

runtimeconfig.json	`System.GC.HeapAffinitizeMask`	十进制值	.NET Core 3.0
■■	`COMPlus_GCHeapAffinitizeMask`	十六进制值	.NET Core 3.0
■■	`DOTNET_GCHeapAffinitizeMask`	十六进制值	.NET 6
.NET Framework ▮ app.config	GCHeapAffinitizeMask	十进制值	.NET Framework 4.6.2

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapAffinitizeMask": 1023
      }
   }
}
```

关联范围

- 指定用于垃圾回收器线程的处理器列表。
- 此设置与 System.GC.HeapAffinitizeMask 类似，只是它允许你指定超过 64 个的处理器。
- 对于 Windows 操作系统，请为处理器编号或范围加上相应的 CPU 组作为前缀，例如“0:1-10,0:12,1:50-52,1:70”。
- 如果禁用了 GC 处理器关联，则忽略此设置。
- 仅适用于服务器垃圾回收。
- 有关详细信息，请参阅 Maoni Stephens 的博客文章 Making CPU configuration better for GC on machines with > 64 CPUs（在 CPU 大于 64 个的计算机上，为 GC 提供更好的 CPU 配置）。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.GCHeapAffinitizeRanges`	以逗号分隔的处理器编号列表或处理器编号范围。 Unix 示例：“1-10,12,50-52,70” Windows 示例：“0:1-10,0:12,1:50-52,1:70”	.NET Core 3.0
[illegible]	`COMPlus_GCHeapAffinitizeRanges`	以逗号分隔的处理器编号列表或处理器编号范围。 Unix 示例：“1-10,12,50-52,70” Windows 示例：“0:1-10,0:12,1:50-52,1:70”	.NET Core 3.0
[illegible]	`DOTNET_GCHeapAffinitizeRanges`	以逗号分隔的处理器编号列表或处理器编号范围。 Unix 示例：“1-10,12,50-52,70” Windows 示例：“0:1-10,0:12,1:50-52,1:70”	.NET 6

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.GCHeapAffinitizeRanges": "0:1-10,0:12,1:50-52,1:70"
      }
   }
}
```

CPU 组

- 配置垃圾回收器是否使用 CPU 组。

  当 64 位 Windows 计算机具有多个 CPU 组（即，有超过 64 个处理器）时，通过启用此元素，可跨所有 CPU 组扩展垃圾回收。垃圾回收器使用所有核心来创建和平衡堆。

- 仅适用于 64 位 Windows 操作系统上的服务器垃圾回收。
- 默认：垃圾回收不会跨 CPU 组扩展。它等效于将值设置为 `0`。
- 有关详细信息，请参阅 Maoni Stephens 的博客文章 Making CPU configuration better for GC on machines with > 64 CPUs（在 CPU 大于 64 个的计算机上，为 GC 提供更好的 CPU 配置）。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.CpuGroup`	`0` - 禁用 `1` - 启用	.NET 5
[illegible]	`COMPlus_GCCpuGroup`	`0` - 禁用 `1` - 启用	.NET Core 1.0
[illegible]	`DOTNET_GCCpuGroup`	`0` - 禁用 `1` - 启用	.NET 6

	[illegible]	[illegible]	[illegible]
.NET Framework [illegible] app.config	GCCpuGroup	`false` - 禁用 `true` - 启用	

> **NOTE**
> 若要配置公共语言运行时 (CLR)，使其也在所有 CPU 组之间分配线程池中的线程，请启用 Thread_UseAllCpuGroups 元素选项。对于 .NET Core 应用，可以通过将 `DOTNET_Thread_UseAllCpuGroups` 环境变量的值设置为 `1` 以启用此选项。

关联

- 指定是否将垃圾回收线程与处理器关联。若要关联一个 GC 线程，则意味着它只能在其特定的 CPU 上运行。为每个 GC 线程创建一个堆。
- 仅适用于服务器垃圾回收。
- 默认：将垃圾回收线程与处理器关联。它等效于将值设置为 `false`。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.NoAffinitize`	`false` - 关联 `true` - 不关联	.NET Core 3.0
[illegible]	`COMPlus_GCNoAffinitize`	`0` - 关联 `1` - 不关联	.NET Core 3.0
[illegible]	`DOTNET_GCNoAffinitize`	`0` - 关联 `1` - 不关联	.NET 6
.NET Framework [illegible] app.config	GCNoAffinitize	`false` - 关联 `true` - 不关联	.NET Framework 4.6.2

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.NoAffinitize": true
      }
   }
}
```

堆限制

- 指定 GC 堆和 GC 簿记的最大提交大小（以字节为单位）。
- 此设置仅适用于 64 位计算机。
- 如果已配置每对象堆限制，则忽略此设置。
- 默认值（仅在某些情况下适用）是 20 MB 或容器内存限制的 75%（以较大者为准）。此默认值在以下情况下适用：
  - 进程正在具有指定内存限制的容器中运行。
  - HeapHardLimitPercent 未设置。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimit`	十进制值	.NET Core 3.0
[illegible]	`COMPlus_GCHeapHardLimit`	十六进制值	.NET Core 3.0
[illegible]	`DOTNET_GCHeapHardLimit`	十六进制值	.NET 6

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimit": 209715200
      }
   }
}
```

TIP

如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆硬限制指定为 200 个兆字节 (MiB)，则该值对于 JSON 文件为 209715200，对于环境变量则为 0xC800000 或 C800000。

堆限制百分比

- 指定允许的 GC 堆使用量占总物理内存的百分比。
- 如果还设置了 System.GC.heaphdlimit，则忽略此设置。
- 此设置仅适用于 64 位计算机。
- 如果进程正在具有指定内存限制的容器中运行，则百分比的计算结果将为该内存限制的百分比。
- 如果已配置每对象堆限制，则忽略此设置。
- 默认值（仅在某些情况下适用）是 20 MB 或容器内存限制的 75%（以较大者为准）。此默认值在以下情况下适用：
  - 进程正在具有指定内存限制的容器中运行。
  - System.GC.HeapHardLimit 未设置。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitPercent`	十进制值	.NET Core 3.0
[illegible]	`COMPlus_GCHeapHardLimitPercent`	十六进制值	.NET Core 3.0
[illegible]	`DOTNET_GCHeapHardLimitPercent`	十六进制值	.NET 6

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.HeapHardLimitPercent": 30
      }
   }
}
```

> **TIP**
> 如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆使用率限制为 30%，则该值对于 JSON 文件为 30，对于环境变量则为 0x1E 或 1E。

每对象堆限制

可以根据每个对象堆指定 GC 的允许堆使用量。不同的堆包括大型对象堆 (LOH)、小型对象堆 (SOH) 和固定对象堆 (POH)。

- 如果为任何 `DOTNET_GCHeapHardLimitSOH`、`DOTNET_GCHeapHardLimitLOH` 或 `DOTNET_GCHeapHardLimitPOH` 设置指定值，则还必须为 `DOTNET_GCHeapHardLimitSOH` 和 `DOTNET_GCHeapHardLimitLOH` 指定值。否则，运行时将无法初始化。
- `DOTNET_GCHeapHardLimitPOH` 的默认值为 0。`DOTNET_GCHeapHardLimitSOH` 和 `DOTNET_GCHeapHardLimitLOH` 没有默认值。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitSOH`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitSOH`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitSOH`	十六进制值	.NET 6

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitLOH`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitLOH`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitLOH`	十六进制值	.NET 6

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitPOH`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitPOH`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitPOH`	十六进制值	.NET 6

TIP

如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆硬限制指定为 200 个兆字节 (MiB)，则该值对于 JSON 文件为 209715200，对于环境变量则为 0xC800000 或 C800000。

每对象堆限制百分比

可以根据每个对象堆指定 GC 的允许堆使用量。不同的堆包括大型对象堆 (LOH)、小型对象堆 (SOH) 和固定对象堆 (POH)。

- 如果为任何 `DOTNET_GCHeapHardLimitSOHPercent`、`DOTNET_GCHeapHardLimitLOHPercent` 或 `DOTNET_GCHeapHardLimitPOHPercent` 设置指定值，则还必须为 `DOTNET_GCHeapHardLimitSOHPercent` 和 `DOTNET_GCHeapHardLimitLOHPercent` 指定值。否则，运行时将无法初始化。
- 如果指定了 `DOTNET_GCHeapHardLimitSOH`、`DOTNET_GCHeapHardLimitLOH` 和 `DOTNET_GCHeapHardLimitPOH`，则忽略这些设置。
- 值为 1 表示 GC 使用该对象堆的总物理内存的 1%。
- 每个值都必须大于 0 并小于 100。此外，3 个百分比值的总和必须小于 100。否则，运行时将无法初始化。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitSOHPercent`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitSOHPercent`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitSOHPercent`	十六进制值	.NET 6

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitLOHPercent`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitLOHPercent`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitLOHPercent`	十六进制值	.NET 6

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HeapHardLimitPOHPercent`	十进制值	.NET 5
[illegible]	`COMPlus_GCHeapHardLimitPOHPercent`	十六进制值	.NET 5
[illegible]	`DOTNET_GCHeapHardLimitPOHPercent`	十六进制值	.NET 6

TIP

如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将堆使用率限制为 30%，则该值对于 JSON 文件为 30，对于环境变量则为 0x1E 或 1E。

高内存百分比

内存负载由正在使用的物理内存的百分比表示。默认情况下，当物理内存负载达到 90%时，垃圾回收对于执行完整的压缩垃圾回收变得更加积极，以避免分页。当内存负载低于 90% 时，GC 优先使用后台回收进行完整的垃

圾回收，这种方法的暂停时间较短，但不会使堆的总大小减少太多。在具有大量内存(80 GB 或更多)的计算机上，默认负载阈值介于 90% 到 97% 之间。

可以通过 `DOTNET_GCHighMemPercent` 环境变量或 `System.GC.HighMemoryPercent` JSON 配置设置来调整高内存负载阈值。如果要控制堆大小，请考虑调整阈值。例如，对于具有 64 GB 内存的计算机上的主要进程，当有 10% 的可用内存时，GC 开始响应是合理的。但是对于较小的进程(例如，仅消耗 1GB 内存的进程)，GC 可以在可用内存少于 10% 的情况下轻松地运行。对于这些较小的进程，请考虑将阈值设置得更高。另一方面，如果你希望较大的进程具有较小的堆大小(即使有足够的物理内存可用)，要使 GC 更快做出反应以缩小堆大小，则降低此阈值是一种有效的方法。

> **NOTE**
> 对于在容器中运行的进程，GC 将根据容器限制考虑物理内存。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.HighMemoryPercent`	十进制值	.NET 5
[illegible]	`COMPlus_GCHighMemPercent`	十六进制值	.NET Core 3.0 .NET Framework 4.7.2
[illegible]	`DOTNET_GCHighMemPercent`	十六进制值	.NET 6

> **TIP**
> 如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，要将高内存阈值设置为 75%，JSON 文件的值将为 75，而环境变量的值为 0x4B 或 4B。

保留 VM

- 配置是将应删除的段置于备用列表上供将来使用，还是将其释放回操作系统 (OS)。
- 默认：将段释放回操作系统。它等效于将值设置为 `false`。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.RetainVM`	`false` - 释放到 OS `true` - 置于备用列表上	.NET Core 1.0
MSBuild 属性	`RetainVMGarbageCollection`	`false` - 释放到 OS `true` - 置于备用列表上	.NET Core 1.0
[illegible]	`COMPlus_GCRetainVM`	`0` - 释放到 OS `1` - 置于备用列表上	.NET Core 1.0
[illegible]	`DOTNET_GCRetainVM`	`0` - 释放到 OS `1` - 置于备用列表上	.NET 6

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.RetainVM": true
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <RetainVMGarbageCollection>true</RetainVMGarbageCollection>
  </PropertyGroup>

</Project>
```

大型页面

- 指定设置堆硬限制时是否应使用大型页面。
- 默认：设置堆硬限制时不要使用大页面。它等效于将值设置为 `0`。
- 这是一项实验性设置。

runtimeconfig.json	不可用	空值	不可用
■■	`COMPlus_GCLargePages`	`0` - 禁用 `1` - 启用	.NET Core 3.0
■■	`DOTNET_GCLargePages`	`0` - 禁用 `1` - 启用	.NET 6

允许大型对象

- 在 64 位平台上，为总大小大于 2 千兆字节 (GB) 的数组配置垃圾回收器支持。
- 默认：垃圾回收支持大于 2 GB 的数组。它等效于将值设置为 `1`。
- 在未来的 .NET 版本中，此选项可能会过时。

runtimeconfig.json	不可用	空值	不可用
■■	`COMPlus_gcAllowVeryLargeObje`	`1` - 启用 `0` - 禁用	.NET Core 1.0
■■	`DOTNET_gcAllowVeryLargeObjec`	`1` - 启用 `0` - 禁用	.NET 6
.NET Framework ■ app.config	gcAllowVeryLargeObjects	`1` - 启用 `0` - 禁用	.NET Framework 4.5

大型对象堆阈值

- 指定导致对象进入大型对象堆 (LOH) 的阈值大小（以字节为单位）。
- 默认阈值为 85,000 字节。
- 指定的值必须大于默认阈值。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.GC.LOHThreshold`	十进制值	.NET Core 1.0
[illegible]	`COMPlus_GCLOHThreshold`	十六进制值	.NET Core 1.0
[illegible]	`DOTNET_GCLOHThreshold`	十六进制值	.NET 6
.NET Framework [illegible] app.config	GCLOHThreshold	十进制值	.NET Framework 4.8

示例：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.GC.LOHThreshold": 120000
      }
   }
}
```

> **TIP**
> 如果要在 runtimeconfig.template.json 中设置该选项，请指定一个十进制值。如果要将选项设置为一个环境变量，请指定一个十六进制值。例如，若要将阈值大小设置为 120,000 个字节，则该值对于 JSON 文件为 120,000，对于环境变量则为 0x1D4C0 或 1D4C0。

独立 GC

- 指定库的路径，该库包含运行时打算加载的垃圾回收器。
- 有关详细信息，请参阅 Standalone GC loader design（独立 GC 加载程序设计）。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	不可用	空值	不可用
[illegible]	`COMPlus_GCName`	string_path	.NET Core 2.0
[illegible]	`DOTNET_GCName`	string_path	.NET 6

用于全球化的运行时配置选项

2021/11/16 •

固定模式

- 确定 .NET Core 应用是否以全球化固定模式运行而无权访问特定区域性的数据和行为。
- 如果省略此设置，应用可运行并可访问区域性数据。它等效于将值设置为 `false`。
- 有关详细信息，请参阅 .NET Core 全球化固定模式。

runtimeconfig.json	`System.Globalization.Invariant`	`false` - 可访问区域性数据 `true` - 以固定模式运行
MSBuild 属性	`InvariantGlobalization`	`false` - 可访问区域性数据 `true` - 以固定模式运行
	`DOTNET_SYSTEM_GLOBALIZATION_INVARIANT`	`0` - 可访问区域性数据 `1` - 以固定模式运行

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Globalization.Invariant": true
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <InvariantGlobalization>true</InvariantGlobalization>
  </PropertyGroup>

</Project>
```

纪元年份范围

- 确定是否放宽对支持多个纪元的日历的范围检查，或者超出纪元日期范围的日期是否引发 ArgumentOutOfRangeException。
- 如果省略此设置，则会放宽范围检查。它等效于将值设置为 `false`。
- 有关详细信息，请参阅日历、纪元和日期范围：放宽的范围检查。

runtimeconfig.json	`Switch.System.Globalization.EnforceJa`	`false` - 放宽的范围检查 `true` - 超出范围导致异常
▬	不可用	不可用

日语日期分析

- 确定是否成功分析包含“1”或“Gannen”作为年份的字符串，或是否仅支持“1”。
- 如果省略此设置，则成功分析包含“1”或“Gannen”作为年份的字符串。它等效于将值设置为 `false`。
- 有关详细信息，请参阅用多个纪元表示日历中的日期。

runtimeconfig.json	`Switch.System.Globalization.EnforceLe`	`false` - 支持“Gannen”或“1” `true` - 仅支持“1”
▬	不可用	不可用

日语年份格式

- 确定是将日本历时代的第一年的格式设置为“Gannen”还是设置为一个数字。
- 如果省略此设置，则第一年的格式设置为“Gannen”。它等效于将值设置为 `false`。
- 有关详细信息，请参阅用多个纪元表示日历中的日期。

runtimeconfig.json	`Switch.System.Globalization.FormatJap`	`false` - 将格式设置为“Gannen” `true` - 将格式设置为数字
▬	不可用	不可用

NLS

- 确定 .NET 是否使用适用于 Windows 应用的 Unicode (ICU) 全球化 API 的区域语言支持 (NLS) 或国际组件。默认情况下，.NET 5 和更高版本在 Windows 10 2019 年 5 月更新和更高版本上使用 ICU 全球化 API。
- 如果省略此设置，则默认情况下，.NET 使用 ICU 全球化 API。它等效于将值设置为 `false`。
- 有关详细信息，请参阅全球化 API 在 Windows 上使用 ICU 库。

runtimeconfig.json	`System.Globalization.UseNls`	`false` - 使用 ICU 全球化 API `true` - 使用 NLS 全球化 API	.NET 5
▬	`DOTNET_SYSTEM_GLOBALIZATION_`	`false` - 使用 ICU 全球化 API `true` - 使用 NLS 全球化 API	.NET 5

预定义的区域性

- 配置应用在启用全球化固定模式时，是否可以创建除固定区域性之外的区域性。
- 如果省略此设置，.NET 会限制全球化固定模式下的区域性创建。它等效于将值设置为 `true`。
- 有关详细信息，请参阅全球化固定模式下的区域性创建和大小写映射。

runtimeconfig.json	`System.Globalization.Predefi` `true` `ltures Only`	在全球化固定模式下，不允许创建除固定区域性以外的任何区域性。 `false` - 允许创建任意区域性。	.NET 6
MSBuild 属性	`PredefinedCulturesOnly`	`true` - 在全球化固定模式下，不允许创建除固定区域性以外的任何区域性。 `false` - 允许创建任意区域性。	.NET 6
[redacted]	`DOTNET_SYSTEM_GLOBALIZATION_` `true` `INVARIANT_MODE`	在全球化固定模式下，不允许创建除固定区域性以外的任何区域性。 `false` - 允许创建任意区域性。	.NET 6

用于网络的运行时配置选项

2021/11/16 •

HTTP/2 协议

- 配置是否启用对 HTTP/2 协议的支持。
- 已在 .NET Core 3.0 中引入。
- 仅限 .NET Core 3.0：如果省略此设置，则会禁用对 HTTP/2 协议的支持。它等效于将值设置为 `false`。
- .NET Core 3.1 和 .NET 5 +：如果省略此设置，则会启用对 HTTP/2 协议的支持。它等效于将值设置为 `true`。

runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.Ht`	`false` or 禁用 `true` - 启用
■	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHANI`	`0` 禁用2SUPPORT `1` - 启用

HttpClient 中的 SPN 创建（.NET 6 及更高版本）

- 当 `Host` 标头缺失并且目标未在默认端口上运行时，会影响用于 Kerberos 和 NTLM 身份验证的服务主体名称 (SPN) 的生成。
- .NET Core 2.x 和 3.x 不在 SPN 中包括端口。
- .NET Core 5.x 在 SPN 中包括端口
- .NET 6 及更高版本不包括端口，但行为是可配置的。

runtimeconfig.json	`System.Net.Http.UsePortInSpn`	`true` - SPN 中包括端口号，例如 `HTTP/host:port` `false` - SPN 中不包括端口，例如 `HTTP/host`
■	`DOTNET_SYSTEM_NET_HTTP_USEPORTINSPN`	`1` - SPN 中包括端口号，例如 `HTTP/host:port` `0` - SPN 中不包括端口，例如 `HTTP/host`

UseSocketsHttpHandler（仅限.NET Core 2.1-3.1）

- 配置 System.Net.Http.HttpClientHandler 是使用 System.Net.Http.SocketsHttpHandler 还是使用旧的 HTTP 协议堆栈（在 Windows 上为 WinHttpHandler，在 Linux 上为基于 libcurl 实现的内部类 `CurlHandler`。）

> **NOTE**
> 可使用高级别网络 API，而不是直接实例化 HttpClientHandler 类。此设置还会影响高级别网络 API 使用的 HTTP 协议堆栈类型，包括 HttpClient 和 HttpClientFactory。

- 如果省略此设置，HttpClientHandler 将使用 SocketsHttpHandler。它等效于将值设置为 `true`。

runtimeconfig.json	`System.Net.Http.UseSocketsHttpHandler`	`true` - 允许使用 SocketsHttpHandler `false` - 允许使用 Windows 上的 WinHttpHandler 或 Linux 上的 libcurl
[redacted]	`DOTNET_SYSTEM_NET_HTTP_USESOCKETSHTTPH`	`1` - 允许使用 SocketsHttpHandler `0` - 允许使用 Windows 上的 WinHttpHandler 或 Linux 上的 libcurl

> **NOTE**
> 从 .NET 5 开始，`System.Net.Http.UseSocketsHttpHandler` 设置不再可用。

Latin1 标头（仅限 .NET Core 3.1）

- 此开关允许放宽 HTTP 标头验证，从而使 SocketsHttpHandler 可以在标头中发送 ISO-8859-1 (Latin-1) 编码的字符。
- 如果省略此设置，则尝试发送非 ASCII 字符将导致 HttpRequestException。它等效于将值设置为 `false`。

runtimeconfig.json	`System.Net.Http.SocketsHttpHandler.Al`	`false` - 禁用 `true` - 启用
[redacted]	`DOTNET_SYSTEM_NET_HTTP_SOCKETSHTTPHAN`	`0` - 禁用 `1` - 启用

> **NOTE**
> 此选项仅在 3.1.9 版本之后 NET Core 3.1 中提供，先前版本或更高版本中则不提供。在 .NET 5 中，我们建议使用 RequestHeaderEncodingSelector。

用于线程的运行时配置选项

2021/11/16 •

本文详细介绍了可用于在 .NET 中配置线程的设置。

> **NOTE**
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

CPU 组

- 配置是否在各 CPU 组之间自动分布线程。
- 如果省略此设置，则不会跨 CPU 组分布线程。它等效于将值设置为 `0`。

runtimeconfig.json	不可用	不可用
	`COMPlus_Thread_UseAllCpuGroups` 或 `DOTNET_Thread_UseAllCpuGroups`	`0` - 禁用 `1` - 启用

最小线程数

- 指定工作线程池的最小线程数。
- 对应于 ThreadPool.SetMinThreads 方法。

runtimeconfig.json	`System.Threading.ThreadPool.MinThreads`	一个表示最小线程数的整数
MSBuild 属性	`ThreadPoolMinThreads`	一个表示最小线程数的整数
	不可用	不可用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MinThreads": 4
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMinThreads>4</ThreadPoolMinThreads>
  </PropertyGroup>

</Project>
```

最大线程数

- 指定工作线程池的最大线程数。
- 对应于 ThreadPool.SetMaxThreads 方法。

runtimeconfig.json	`System.Threading.ThreadPool.MaxThreads`	一个表示最大线程数的整数
MSBuild 属性	`ThreadPoolMaxThreads`	一个表示最大线程数的整数
	不可用	不可用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.MaxThreads": 20
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <ThreadPoolMaxThreads>20</ThreadPoolMaxThreads>
  </PropertyGroup>

</Project>
```

用来响应阻止工作项的线程注入

在某些情况下，线程池会检测阻止其线程的工作项。为了进行补偿，会注入更多线程。在 .NET 6+ 中，可以使用以下运行时配置设置来配置线程注入，以响应阻止工作项。目前，这些设置仅对等待其他任务完成的工作项（例如，在典型的 sync-over-async 情况下）有效。

RUNTIMECONFIG.JSON		
`System.Threading.ThreadPool.Blocking.ThreadsPerCpuCount`	达到基于 `MinThreads` 的线程计数后，此值（在乘以处理器计数后）指定可能创建的其他线程数（不含延迟）。	.NET 6

RUNTIMECONFIG.JSON [illegible]	[illegible]	[illegible]
`System.Threading.ThreadPool.Blocking.[illegible]PerDelayStep_ProcCountFactor`	达到基于 `ThreadsToAddWithoutDelay` 的线程计数后，此值（在乘以处理器计数后）指定在创建每个新线程之前其他 `DelayStepMs` 将添加到延迟的线程数。	.NET 6
`System.Threading.ThreadPool.Blocking.[illegible]epMs`	达到基于 `ThreadsToAddWithoutDelay` 的线程计数后，此值指定要为每个 `ThreadsPerDelayStep` 线程添加的额外延迟数，这会在创建每个新线程之前应用。	.NET 6
`System.Threading.ThreadPool.Blocking.[illegible]sMs`	达到基于 `ThreadsToAddWithoutDelay` 的线程计数后，此值指定在创建每个新线程之前要使用的最大延迟。	.NET 6

配置设置如何生效

- 达到基于 `MinThreads` 的线程计数后，最多可创建 `ThreadsToAddWithoutDelay` 个其他线程（不含延迟）。
- 之后，在创建每个其他线程之前，会引发延迟，从 `DelayStepMs` 开始。
- 对于添加了延迟的每个 `ThreadsPerDelayStep` 线程，会将其他 `DelayStepMs` 添加到延迟。
- 延迟不能超过 `MaxDelayMs`。
- 仅在创建线程之前引发延迟。如果线程已可用，则会在不延迟的情况下将其释放，以便对阻止工作项进行补偿。
- 还会使用物理内存使用情况和限制，超出阈值时，系统会切换到较慢的线程注入。

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.ThreadPool.Blocking.ThreadsToAddWithoutDelay_ProcCountFactor": 5
      }
   }
}
```

托管线程中的 `AutoreleasePool`

此选项用于配置在受支持的 macOS 平台上运行时，每个托管线程是否接收隐式 NSAutoreleasePool。

	[illegible]	[illegible]	[illegible]
runtimeconfig.json	`System.Threading.Thread.Enab[illegible]`	`true` 或 `false`	.NET 6
MSBuild 属性	`AutoreleasePoolSupport`	`true` 或 `false`	.NET 6
[illegible]	不可用	空值	不可用

示例

runtimeconfig.json 文件：

```
{
   "runtimeOptions": {
      "configProperties": {
         "System.Threading.Thread.EnableAutoreleasePool": true
      }
   }
}
```

项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <AutoreleasePoolSupport>true</AutoreleasePoolSupport>
  </PropertyGroup>

</Project>
```

.NET 应用程序发布概述

2021/11/16 •

可在两种模式下发布使用 .NET 创建的应用程序，模式会影响用户运行应用的方式。

将应用作为独立应用，生成的应用程序将包含 .NET 运行时和库，以及该应用程序及其依赖项。应用程序的用户可以在未安装 .NET 运行时的计算机上运行该应用程序。

如果将应用发布为依赖于框架的应用，生成的应用程序将仅包含该应用程序本身及其依赖项。应用程序的用户必须单独安装 .NET 运行时。

默认情况下，这两种发布模式都会生成特定于平台的可执行文件。不使用可执行文件也可以创建依赖于框架的应用程序，这些应用程序是跨平台的。

生成可执行文件时，可以使用运行时标识符 (RID) 指定目标平台。有关 RID 的详细信息，请参阅 .NET RID 目录。

下表概述了每个 SDK 版本用于将应用发布为依赖于框架的应用或独立应用的命令：

	SDK 2.1	SDK 3.1	SDK 5.0	
适用于当前平台的依赖于框架的可执行文件。		✔	✔	`dotnet publish`
适用于特定平台的依赖于框架的可执行文件。		✔	✔	`dotnet publish -r <RID> --self-contained false`
依赖于框架的跨平台二进制文件。	✔	✔	✔	`dotnet publish`
独立可执行文件。	✔	✔	✔	`dotnet publish -r <RID>`

有关详细信息，请参阅 .NET dotnet publish 命令。

生成可执行文件

可执行文件不是跨平台的。它们特定于操作系统和 CPU 体系结构。发布应用并创建可执行文件时，可以将应用发布为独立应用或依赖于框架的应用。将应用发布为独立应用，会在应用中包含 .NET 运行时，该应用的用户无需在运行应用前安装 .NET。如果将应用发布为依赖于框架的应用，则该应用不包含 .NET 运行时和库，而仅包含该应用和第三方依赖项。

以下命令可生成可执行文件：

	SDK 2.1	SDK 3.1	SDK 5.0	
适用于当前平台的依赖于框架的可执行文件。		✔	✔	`dotnet publish`

	SDK 2.1	SDK 3.1	SDK 5.0	
适用于特定平台的依赖于框架的可执行文件。		✔	✔	`dotnet publish -r <RID> --self-contained false`
独立可执行文件。	✔	✔	✔	`dotnet publish -r <RID>`

生成跨平台二进制文件

以 dll 文件的形式将应用发布为依赖于框架的应用时，将创建跨平台二进制文件。dll 文件将与项目同名。例如，如果有名为 word_reader 的应用，则会创建名为 word_reader.dll 的文件。以这种方式发布的应用可通过 `dotnet <filename.dll>` 命令运行，并且可在任意平台上运行。

只要安装了目标 .NET 运行时，就可以在任何操作系统上运行跨平台二进制文件。如果未安装目标 .NET 运行时，如果将应用配置为前滚，则它可以使用较新的运行时运行。有关详细信息，请参阅依赖于框架的应用前滚。

以下命令可生成跨平台二进制文件：

	SDK 2.1	SDK 3.X	SDK 5.0	
依赖于框架的跨平台二进制文件。	✔	✔	✔	`dotnet publish`

发布依赖于框架的应用

如果将应用发布为依赖于框架的应用，则该应用是跨平台的，且不包含 .NET 运行时。应用的用户需要安装 .NET 运行时。

如果将应用发布为依赖于框架的应用，会以 dll 文件的形式生成一个跨平台二进制文件，还会生成面向当前平台的特定于平台的可执行文件。dll 是跨平台的，而可执行文件不是。例如，如果发布名为 word_reader 的应用且面向 Windows，则将创建 word_reader.exe 和 word_reader.dll。面向 Linux 或 macOS 时，将创建 word_reader 可执行文件和 word_reader.dll。有关 RID 的详细信息，请参阅 .NET RID 目录。

> **IMPORTANT**
> 当你发布依赖于框架的应用时，.NET SDK 2.1 不会生成特定于平台的可执行文件。

可以通过 `dotnet <filename.dll>` 命令运行应用的跨平台二进制文件，并且它可以在任何平台上运行。如果应用使用具有特定于平台的实现的 NuGet 包，则所有平台的依赖项都将连同应用一起复制到发布文件夹。

可以通过将 `-r <RID> --self-contained false` 参数传递到 `dotnet publish` 命令，为特定平台创建可执行文件。省略 `-r` 参数时，将为当前平台创建可执行文件。具有特定于目标平台的依赖项的任何 NuGet 包都将复制到发布文件夹。如果不需要特定于平台的可执行文件，则可以在项目文件中指定 `<UseAppHost>False</UseAppHost>`。有关详细信息，请参阅适用于 .NET SDK 项目的 MSBuild 参考。

优点

- 小型部署
  仅分发应用及其依赖项。.NET 运行时和库由用户安装，所有应用共享运行时。
- 跨平台
  应用和任何基于 .NET 的库都可在不同的操作系统上运行。无需为应用定义目标平台。有关 .NET 文件格式的详细信息，请参阅 .NET 程序集文件格式。

- 使用最新修补运行时
  应用会使用目标系统上安装的最新运行时(在 .NET 的目标大小系列中)。这意味着应用将自动使用 .NET 运行时的最新修补版本。可以重写这一默认行为。有关详细信息，请参阅依赖于框架的应用前滚。

缺点

- 要求预先安装运行时
  仅当主机系统上已安装应用设为目标的 .NET 版本时，应用才能运行。可以为应用配置前滚行为，要求使用特定版本的 .NET 或允许使用较新版本的 .NET。有关详细信息，请参阅依赖于框架的应用前滚。

- .NET 可更改
  可以在运行应用的计算机上更新 .NET 运行时和库。在极少数情况下，如果使用 .NET 库(大多数应用都会使用)，这可能会更改应用的行为。可以配置应用如何使用较新版本的 .NET。有关详细信息，请参阅依赖于框架的应用前滚。

以下缺陷仅限于.NET Core 2.1 SDK。

- 使用 `dotnet` 命令启动应用
  用户必须运行 `dotnet <filename.dll>` 命令来启动你的应用。如果将应用发布为依赖于框架的应用，则 .NET Core 2.1 SDK 不会生成特定于平台的可执行文件。

示例

发布依赖于框架的跨平台应用。与 *dll* 文件一起创建面向当前平台的可执行文件。

```
dotnet publish
```

发布依赖于框架的跨平台应用。与 *dll* 文件一起创建 Linux 64 位可执行文件。此命令对于 .NET Core SDK 2.1 无效。

```
dotnet publish -r linux-x64 --self-contained false
```

发布独立应用

将应用发布为独立应用，将生成特定于平台的可执行文件。输出发布文件夹包含应用的所有组件，包括 .NET 库和目标运行时。应用独立于其他 .NET 应用，且不使用本地安装的共享运行时。应用的用户无需下载和安装 .NET。

针对指定的目标平台生成可执行二进制文件。例如，如果你有一个名为 word_reader 的应用，并发布适用于 Windows 的独立可执行文件，则将创建 word_reader.exe 文件。针对 Linux 或 macOS 发布时，将创建 word_reader 文件。用 `dotnet publish` 命令的 `-r <RID>` 参数指定目标平台和体系结构。有关 RID 的详细信息，请参阅 .NET RID 目录。

如果应用具有特定于平台的依赖项(例如包含特定于平台的依赖项的 NuGet 包)，这些依赖项将与应用一起复制到发布文件夹。

优点

- 控制 .NET 版本
  你可以控制随应用部署的 .NET 版本。

- 特定于平台的定向
  由于你必须为每个平台发布应用，因此你知道应用可运行于哪些平台。如果 .NET 引入了新平台，则用户无法在该新平台上运行你的应用，直到你发布面向该平台的版本。在用户在新平台上运行你的应用之前，你可以测试应用以排除兼容性问题。

缺点

- **大型部署**
  由于你的应用包含 .NET 运行时和所有应用依赖项，因此下载大小和所需硬盘空间比依赖于框架的应用的版本要大。

  > **TIP**
  > 可以通过使用 .NET 全球化固定模式在 Linux 系统上减少大约 28 MB 的部署大小。这会强制应用像处理固定区域性一样处理所有区域性。

  > **TIP**
  > IL 剪裁可以进一步减小部署的大小。

- **较难更新 .NET 版本**
  只能通过发布新版本的应用来升级(与应用一起分发的).NET 运行时。但是，.NET 将在运行你的应用的计算机中针对框架库按需更新关键安全修补程序。由你负责此安全修补程序方案的端到端验证。

示例

发布独立应用。创建 macOS 64 位可执行文件。

```
dotnet publish -r osx-x64
```

发布独立应用。创建 Windows 64 位可执行文件。

```
dotnet publish -r win-x64
```

使用 ReadyToRun 映像发布

使用 ReadyToRun 映像发布可以缩短应用程序的启动时间，但代价是增加应用程序的大小。若要使用 ReadyToRun 映像发布，请参阅 ReadyToRun 来了解更多详情。

优点

- 缩短了启动时间
  应用程序运行 JIT 所花费的时间将缩短。

缺点

- 增加了大小
  应用程序在磁盘上的大小将增加。

示例

使用 ReadyToRun 映像发布独立式应用。创建 macOS 64 位可执行文件。

```
dotnet publish -c Release -r osx-x64 -p:PublishReadyToRun=true
```

使用 ReadyToRun 映像发布独立式应用。创建 Windows 64 位可执行文件。

```
dotnet publish -c Release -r win-x64 -p:PublishReadyToRun=true
```

请参阅

- 使用 .NET CLI 部署 .NET 应用。
- 使用 Visual Studio 部署 .NET 应用。
- .NET 运行时标识符 (RID) 目录。
- 选择要使用的 .NET 版本。

使用 Visual Studio 部署 .NET Core 应用

2021/11/16 •

可将 .NET Core 应用程序部署为依赖框架的部署或独立部署，前者包含应用程序二进制文件，但依赖目标系统上存在的 .NET Core，而后者同时包含应用程序和 .NET Core 二进制文件。有关 .NET Core 应用程序部署的概述，请参阅 .NET Core 应用程序部署。

以下各节演示如何使用 Microsoft Visual Studio 创建下列各类部署：

- 依赖框架的部署
- 包含第三方依赖项的依赖框架的部署
- 独立部署
- 包含第三方依赖项的独立部署

有关使用 Visual Studio 开发 .NET Core 应用程序的信息，请参阅 .NET Core 依赖项和要求。

依赖框架的部署

如果不使用第三方依赖项，部属依赖框架的部署只包括生成、测试和发布应用。一个用 C# 编写的简单示例可说明此过程。

1. 创建项目。

   选择“文件” > “新建” > “项目”。在“新建项目”对话框中，在“已安装”项目类型窗格中展开你的语言的(C# 或 Visual Basic)项目类别，选择“.NET Core”，然后在中心窗格中选择控制台应用 (.NET Core) 模板 。在“名称”文本框中输入项目名称，如“FDD”。选择“确定”按钮。

2. 添加应用程序的源代码。

   在编辑器中打开 Program.cs 文件或 Program.vb 文件，然后使用下列代码替换自动生成的代码 。它会提示用户输入文本，并显示用户输入的个别词。它使用正则表达式 `\w+` 来将输入文本中的词分开。

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. 创建应用的调试版本。

   选择“生成” > “生成解决方案”。也可通过选择“调试” > “开始调试”来编译和运行应用程序的调试版本。

4. 部署应用。

   调试并测试程序后，创建要与应用一起部署的文件。若要从 Visual Studio 发布，请执行以下操作：

   a. 将工具栏上的解决方案配置从“调试”更改为“发布”，生成应用的发布（而非调试）版本。

   b. 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“发布”。

   c. 在“发布”选项卡上，选择“发布”。Visual Studio 将包含应用程序的文件写入本地文件系统。

   d. “发布”选项卡现在显示单个配置文件 FolderProfile。该配置文件的配置设置显示在选项卡的“摘要”部分。

   生成的文件位于 Windows 上名为 `Publish` 的目录（对于 Unix 系统，是名为 `publish` 的目录）中的项目 \bin\release\netcoreapp2.1 子目录的子目录中。

与应用程序的文件一起，发布过程将发出包含应用调试信息的程序数据库 (.pdb) 文件。该文件主要用于调试异常。可以选择不使用应用程序文件打包该文件。但是，如果要调试应用的发布版本，则应保存该文件。

可以采用任何喜欢的方式部署完整的应用程序文件集。例如，可以使用简单的 `copy` 命令将其打包为 Zip 文件，或者使用选择的安装包进行部署。安装成功后，用户可通过使用 `dotnet` 命令或提供应用程序文件名（如 `dotnet fdd.dll`）来执行应用程序。

除应用程序二进制文件外，安装程序还应捆绑共享框架安装程序，或在安装应用程序的过程中将其作为先决条件进行检查。安装共享框架需要管理员/根访问权限，因为它属于计算机范围。

包含第三方依赖项的依赖框架的部署

要使用一个或多个第三方依赖项来部署依赖框架的部署，需要任何依赖项都可供项目使用。在生成应用之前，还需执行以下额外步骤：

1. 使用 NuGet 包管理器向项目添加对 NuGet 包的引用；如果系统上还没有此包，请先安装它。要打开包管理器，请选择“工具” > “NuGet 包管理器” > “管理解决方案的 NuGet 包”。

2. 确认系统上安装了第三方依赖项（例如，`Newtonsoft.Json`）；如果没有，请安装它们。“已安装”选项卡列出了系统中已安装的 NuGet 包。如果此处未列出 `Newtonsoft.Json`，请选择“浏览”选项卡，然后在搜索框中输入“NewtonsoftJson”。选择 `Newtonsoft.Json`，在右侧窗格中选择项目，然后选择“安装”。

3. 如果系统中已安装 `Newtonsoft.Json`，请在“管理解决方案包”选项卡的右侧窗格中选择项目，将其添加到项目。

如果依赖框架的部署具有第三方依赖项，则其可移植性只与第三方依赖项相同。例如，如果某个第三方库只支持 macOS，该应用将无法移植到 Windows 系统。当第三方依赖项本身取决于本机代码时，也可能发生此情况。Kestrel 服务器就是一个很好的示例，它需要 libuv 的本机依赖项。当为具有此类第三方依赖项的应用程序创建 FDD 时，已发布的输出会针对每个本机依赖项支持（存在于 NuGet 包中）的运行时标识符 (RID) 包含一个文件夹。

不包含第三方依赖项的独立部署

部署没有第三方依赖项的独立部署包括创建项目、修改 csproj 文件、生成、测试以及发布应用。一个用 C# 编写的简单示例可说明此过程。首先，对项目进行创建、编码及测试，就像对依赖框架的部署的操作一样：

1. 创建项目。

   选择“文件” > “新建” > “项目”。在“新建项目”对话框中，在“已安装”项目类型窗格中展开你的语言的（C# 或 Visual Basic）项目类别，选择“.NET Core”，然后在中心窗格中选择控制台应用 (.NET Core) 模板 。在“名称”文本框中输入项目名称如“SCD”，然后选择“确定”按钮。

2. 添加应用程序的源代码。

   在编辑器中打开 Program.cs 或 Program.vb 文件，然后使用下列代码替换自动生成的代码 。它会提示用户输入文本，并显示用户输入的个别词。它使用正则表达式 `\w+` 来将输入文本中的词分开。

```
using System;
using System.Text.RegularExpressions;

namespace Applications.ConsoleApps
{
    public class ConsoleParser
    {
        public static void Main()
        {
            Console.WriteLine("Enter any text, followed by <Enter>:\n");
            String s = Console.ReadLine();
            ShowWords(s);
            Console.Write("\nPress any key to continue... ");
            Console.ReadKey();
        }

        private static void ShowWords(String s)
        {
            String pattern = @"\w+";
            var matches = Regex.Matches(s, pattern);
            if (matches.Count == 0)
            {
                Console.WriteLine("\nNo words were identified in your input.");
            }
            else
            {
                Console.WriteLine($"\nThere are {matches.Count} words in your string:");
                for (int ctr = 0; ctr < matches.Count; ctr++)
                {
                    Console.WriteLine($"   #{ctr,2}: '{matches[ctr].Value}' at position
{matches[ctr].Index}");
                }
            }
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Namespace Applications.ConsoleApps
    Public Module ConsoleParser
        Public Sub Main()
            Console.WriteLine("Enter any text, followed by <Enter>:")
            Console.WriteLine()
            Dim s = Console.ReadLine()
            ShowWords(s)
            Console.Write($"{vbCrLf}Press any key to continue... ")
            Console.ReadKey()
        End Sub

        Private Sub ShowWords(s As String)
            Dim pattern = "\w+"
            Dim matches = Regex.Matches(s, pattern)
            Console.WriteLine()
            If matches.Count = 0 Then
                Console.WriteLine("No words were identified in your input.")
            Else
                Console.WriteLine($"There are {matches.Count} words in your string:")
                For ctr = 0 To matches.Count - 1
                    Console.WriteLine($"   #{ctr,2}: '{matches(ctr).Value}' at position
{matches(ctr).Index}")
                Next
            End If
            Console.WriteLine()
        End Sub
    End Module
End Namespace
```

3. 确定是否要使用全球化固定模式。

   特别是如果应用面向 Linux，则可以通过利用全球化固定模式来减小部署的总规模。全球化固定模式适用于不具有全局意识且可以使用固定区域性的格式约定、大小写约定以及字符串比较和排序顺序的应用程序。

   要启用固定模式，右键单击“解决方案资源管理器”中的项目（不是解决方案），然后选择“编辑 SCD.csproj”或“编辑 SCD.vbproj”。然后将以下突出显示的行添加到文件中：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Globalization.Invariant" Value="true" />
  </ItemGroup>

</Project>
```

4. 创建应用程序的调试版本。

   选择“生成”>“生成解决方案”。也可通过选择“调试”>“开始调试”来编译和运行应用程序的调试版本。通过此调试步骤，可以识别应用程序在主机平台上运行时出现的问题。仍然必须在每个目标平台上对其进行测试。

   如果已启用全球化固定模式，请特别确保测试缺少对文化敏感的数据是否适合应用程序。

完成调试后，可以发布独立部署：

- Visual Studio 15.6 及更早版本
- Visual Studio 15.7 及更高版本

调试并测试程序后，为应用的每个目标平台创建要与应用一起部署的文件。

若要从 Visual Studio 发布应用，请执行以下操作：

1. 定义应用的目标平台。

   a. 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“编辑 SCD.csproj”。

   b. 在 csproj 文件（该文件用于定义应用的目标平台）的 `<PropertyGroup>` 部分中创建 `<RuntimeIdentifiers>` 标记，然后指定每个目标平台的运行时标识符 (RID)。还需要添加分号来分隔 RID。请查看运行时标识符目录，获取运行时标识符列表。

   例如，以下示例表明应用在 64 位 Windows 10 操作系统和 64 位 OS X 10.11 版本的操作系统上运行。

   ```
   <PropertyGroup>
      <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
   </PropertyGroup>
   ```

   `<RuntimeIdentifiers>` 元素可能会进入 csproj 文件的任何 `<PropertyGroup>` 中。本节后面部分将显示完整的示例 csproj 文件。

2. 发布你的应用。

   调试并测试程序后，为应用的每个目标平台创建要与应用一起部署的文件。

   若要从 Visual Studio 发布应用，请执行以下操作：

   a. 将工具栏上的解决方案配置从“调试”更改为“发布”，生成应用的发布（而非调试）版本。

   b. 在“解决方案资源管理器”中右键单击项目（而非解决方案），然后选择“发布”。

   c. 在“发布”选项卡上，选择“发布”。Visual Studio 将包含应用程序的文件写入本地文件系统。

   d. “发布”选项卡现在显示单个配置文件 FolderProfile。该配置文件的配置设置显示在选项卡的“摘要”部分。目标运行时用于标识已发布的运行时，目标位置用于标识独立部署文件的写入位置。

   e. 默认情况下，Visual Studio 将所有已发布文件写入单个目录。为了方便起见，最好为每个目标运行时创建单个配置文件，并将已发布文件置于特定于平台的目录中。这包括为每个目标平台创建单独的发布配置文件。现在，请执行下列操作，为每个平台重新生成应用程序：

      a. 在“发布”对话框中选择“创建新配置文件”。

      b. 在“选取发布目标”对话框中，将“选择文件夹”位置更改为 bin\Release\PublishOutput\win10-x64。选择“确定”。

      c. 在配置文件列表中选择新配置文件 (FolderProfile1)，并确保“目标运行时”为 `win10-x64`。如果不是，请选择“设置”。在“配置文件设置”对话框中，将“目标运行时”更改为 `win10-x64`，然后选择“保存”。否则，选择“取消”。

      d. 选择“发布”，发布 64 位 Windows 10 平台的应用。

      e. 请再次按照上述步骤创建 `osx.10.11-x64` 平台的配置文件。“目标位置”为 bin\Release\PublishOutput\osx.10.11-x64，“目标运行时”为 `osx.10.11-x64`。Visual Studio 分配给此配置文件的名称是 FolderProfile2。

   每个目标位置中都包含启动应用所需的完整文件集（既包含应用文件，又包含所有 .NET Core 文件）。

与应用程序的文件一起，发布过程将发出包含应用调试信息的程序数据库 (.pdb) 文件。该文件主要用于调试异

常。可以选择不使用应用程序文件打包该文件。但是，如果要调试应用的发布版本，则应保存该文件。

可按照任何喜欢的方式部署已发布的文件。例如，可以使用简单的 `copy` 命令将其打包为 Zip 文件，或者使用选择的安装包进行部署。

下面是此项目完整的 csproj 文件。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
</Project>
```

包含第三方依赖项的独立部署

部署包含一个或多个第三方依赖项的独立部署包括添加依赖项。在生成应用之前，还需执行以下额外步骤：

1. 使用 NuGet 包管理器向项目添加对 NuGet 包的引用；如果系统上还没有此包，请先安装它。要打开包管理器，请选择“工具” > “NuGet 包管理器” > “管理解决方案的 NuGet 包”。
2. 确认系统上安装了第三方依赖项（例如，`Newtonsoft.Json`）；如果没有，请安装它们。“已安装”选项卡列出了系统中已安装的 NuGet 包。如果此处未列出 `Newtonsoft.Json`，请选择“浏览”选项卡，然后在搜索框中输入“NewtonsoftJson”。选择 `Newtonsoft.Json`，在右侧窗格中选择项目，然后选择“安装”。
3. 如果系统中已安装 `Newtonsoft.Json`，请在“管理解决方案包”选项卡的右侧窗格中选择项目，将其添加到项目。

下面是此项目的完整 csproj 文件：

- Visual Studio 15.6 及更早版本
- Visual Studio 15.7 及更高版本

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
    <RuntimeIdentifiers>win10-x64;osx.10.11-x64</RuntimeIdentifiers>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.2" />
  </ItemGroup>
</Project>
```

部署应用程序时，应用中使用的任何第三方依赖项也包含在应用程序文件中。运行应用的系统上不需要第三方库。

可以只将具有一个第三方库的独立部署部署到该库支持的平台。这与依赖框架的部署中具有本机依赖项和第三方依赖项相似，其中的本机依赖项不会存在于目标平台上，除非之前在该平台上安装了该依赖项。

请参阅

- .NET Core 应用程序部署
- .NET Core 运行时标识符 (RID) 目录

使用 .NET CLI 发布 .NET 应用

2021/11/16 •

本文展示了如何使用命令行来发布 .NET 应用程序。.NET 提供了三种发布应用程序的方法。依赖于框架的部署会生成一个跨平台 .dll 文件，此文件使用本地安装的 .NET 运行时。依赖于框架的可执行文件会生成一个特定于平台的可执行文件，此文件使用本地安装的 .NET 运行时。独立式可执行文件会生成一个特定于平台的可执行文件，并包含 .NET 运行时的本地副本。

有关这些发布模式的概述，请参阅 .NET 应用程序部署。

正在查找有关 CLI 的快速帮助？ 下表列出了一些关于如何发布应用的示例。可以使用 `-f <TFM>` 参数或通过编辑项目文件来指定目标框架。有关详细信息，请参阅发布基本知识。

[illegible]	SDK [illegible]	[illegible]
依赖框架的部署	2.1	`dotnet publish -c Release`
	3.1	`dotnet publish -c Release -p:UseAppHost=false`
	5.0	`dotnet publish -c Release -p:UseAppHost=false`
依赖于框架的可执行文件	3.1	`dotnet publish -c Release -r <RID> --self-contained false` `dotnet publish -c Release`
	5.0	`dotnet publish -c Release -r <RID> --self-contained false` `dotnet publish -c Release`
独立部署	2.1	`dotnet publish -c Release -r <RID> --self-contained true`
	3.1	`dotnet publish -c Release -r <RID> --self-contained true`
	5.0	`dotnet publish -c Release -r <RID> --self-contained true`

* 当使用 SDK 版本 3.1 或更高版本时，依赖于框架的可执行文件是运行基本 `dotnet publish` 命令时的默认发布模式。

> **NOTE**
>
> `-c Release` 参数不是必需的。它作为提醒提供来发布应用的发行内部版本。

发布基本知识

发布应用时，项目文件的 `<TargetFramework>` 设置指定默认目标框架。可以将目标框架更改为任意有效目标框架名字对象 (TFM)。例如，如果项目使用 `<TargetFramework>netcoreapp2.1</TargetFramework>`，则会创建以 .NET Core 2.1 为目标的二进制文件。此设置中指定的 TFM 是 `dotnet publish` 命令使用的默认目标。

若要以多个框架为目标，则可以将 `<TargetFrameworks>` 设置设为多个 TFM 值（以分号分隔）。当你生成应用时，会为每个目标框架生成一个内部版本。但是，当你发布应用时，必须使用 `dotnet publish -f <TFM>` 命令指定目标框架。

除非另有设置，否则 `dotnet publish` 命令的输出目录为 `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/`。除非使用 `-c` 参数进行更改，否则默认的 BUILD-CONFIGURATION 模式为 Debug。例如，`dotnet publish -c Release -f netcoreapp2.1` 发布到 `./bin/Release/netcoreapp2.1/publish/`。

如果你使用 .NET Core SDK 3.1 或更高版本，则默认发布模式为依赖于框架的可执行文件。

如果你使用 .NET Core SDK 2.1，则默认发布模式为依赖于框架的部署。

本机依赖项

如果应用具有本机依赖项，则只能在相同操作系统上运行。例如，如果应用使用本机 Windows API，则不能在 macOS 或 Linux 上运行。需要提供特定于平台的代码并为每个平台编译可执行文件。

另外，如果引用的库具有本机依赖项，则应用可能无法在每个平台上运行。但是，引用的 NuGet 包可能包含特定于平台的版本，以便处理所需的本机依赖项。

使用本机依赖项分发应用时，可能需要使用 `dotnet publish -r <RID>` 开关来指定想要发布的目标平台。有关运行时标识符的详细信息，请参阅运行时标识符 (RID) 目录。

有关特定于平台的二进制文件的详细信息，请参阅依赖于框架的可执行文件和独立部署部分。

示例应用

可使用以下应用来探索发布命令。通过在终端中运行以下命令来创建应用：

```
mkdir apptest1
cd apptest1
dotnet new console
dotnet add package Figgle
```

控制台模板生成的 `Program.cs` 或 `Program.vb` 文件需要进行以下更改：

```
using System;

namespace apptest1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"));
        }
    }
}
```

```
Module Program
    Sub Main(args As String())
        Console.WriteLine(Figgle.FiggleFonts.Standard.Render("Hello, World!"))
    End Sub
End Module
```

运行应用 (`dotnet run`) 时，将显示以下输出：

```
 _   _      _ _        __        __         _     _ _
| | | | ___| | | ___    \ \      / /__  _ __| | __| | |
| |_| |/ _ \ | |/ _ \    \ \ /\ / / _ \| '__| |/ _` | |
|  _  |  __/ | | (_) |    \ V  V / (_) | |  | | (_| |_|
|_| |_|\___|_|_|\___( )    \_/\_/ \___/|_|  |_|\__,_(_)
                    |/
```

依赖框架的部署

对于 .NET Core 2.1 SDK CLI，依赖于框架的部署 (FDD) 是基本 `dotnet publish` 命令的默认模式。在更高版本的 SDK 中，依赖于框架的可执行文件是默认模式。

将应用作为 FDD 发布时，会在 `./bin/<BUILD-CONFIGURATION>/<TFM>/publish/` 文件夹中创建 `<PROJECT-NAME>.dll` 文件。若要运行应用，请导航到输出文件夹并使用 `dotnet <PROJECT-NAME>.dll` 命令。

你的应用被配置为以特定版本的 .NET 为目标。目标 .NET 运行时必须在运行应用的任何虚拟机上。例如，如果你的应用以 .NET Core 3.1 为目标，则任何运行你的应用的虚拟机都必须安装 .NET Core 3.1 运行时。如发布基础知识部分中所述，可以编辑项目文件为更改默认目标框架或面向多个框架。

发布 FDD 会创建一个应用，该应用会自动前滚到运行该应用的系统上可用的最新 .NET 安全补丁。若要详细了解编译时的版本绑定，请参阅选择要使用的 .NET 版本。

☐☐☐	SDK ☐☐	☐☐
依赖框架的部署	2.1	`dotnet publish -c Release`
	3.1	`dotnet publish -c Release -p:UseAppHost=false`
	5.0	`dotnet publish -c Release -p:UseAppHost=false`

依赖于框架的可执行文件

对于.NET 5（以及 .NET Core 3.1）SDK CLI，依赖于框架的可执行文件 (FDE) 是基本 `dotnet publish` 命令的默认模式。只要想要面向当前操作系统，就不需要指定任何其他参数。

在此模式下，将创建特定于平台的可执行主机来托管跨平台应用。此模式类似于 FDD，因为 FDD 需要 `dotnet` 命令形式的主机。每个平台的主机可执行文件名各不相同，其文件名类似于 `<PROJECT-FILE>.exe`。可以直接运行此可执行文件，而不是调用 `dotnet <PROJECT-FILE>.dll`，这仍然是运行应用的可接受方式。

你的应用被配置为以特定版本的 .NET 为目标。目标 .NET 运行时必须在运行应用的任何虚拟机上。例如，如果你的应用以 .NET Core 3.1 为目标，则任何运行你的应用的虚拟机都必须安装 .NET Core 3.1 运行时。如发布基础知识部分中所述，可以编辑项目文件为更改默认目标框架或面向多个框架。

发布 FDE 会创建一个应用，此应用会自动前滚到运行此应用的系统上可用的最新 .NET 安全补丁。若要详细了解编译时的版本绑定，请参阅选择要使用的 .NET 版本。

对于 .NET 2.1，必须结合使用以下开关和 `dotnet publish` 命令来发布 FDE：

- `-r <RID>` 此开关使用标识符 (RID) 来指定目标平台。有关运行时标识符的详细信息，请参阅运行时标识符 (RID) 目录。
- `--self-contained false` 此开关会禁用 `-r` 开关的默认行为，即禁止创建独立部署 (SCD)。此开关会创建一个 FDE。

[illegible]	SDK [illegible]	[illegible]
依赖于框架的可执行文件	3.1	`dotnet publish -c Release -r <RID> --self-contained false` `dotnet publish -c Release`
	5.0	`dotnet publish -c Release -r <RID> --self-contained false` `dotnet publish -c Release`

每次使用 `-r` 开关时，输出文件路径都将更改为：`./bin/<BUILD-CONFIGURATION>/<TFM>/<RID>/publish/`

如果使用示例应用，请运行 `dotnet publish -f net5.0 -r win10-x64 --self-contained false`。此命令将创建以下可执行文件：`./bin/Debug/net5.0/win10-x64/publish/apptest1.exe`

> **NOTE**
> 可以通过启用全局固定模式来降低部署的总大小。此模式适用于不具有全局意识且可以使用固定区域性的格式约定、大小写约定以及字符串比较和排序顺序的应用程序。若要详细了解全球化固定模式以及如何启用它，请参阅 .NET 全球化固定模式。

独立部署

当你发布独立式部署 (SCD) 时，.NET SDK 会创建特定于平台的可执行文件。如果发布 SCD，则会包括运行应用所需的所有 .NET 文件，但不包括 .NET 的本机依赖项。这些依赖项必须在应用运行前存在于系统中。

如果发布 SCD，则会创建一个不前滚到最新可用 .NET 安全补丁的应用。若要详细了解编译时的版本绑定，请参阅选择要使用的 .NET 版本。

必须通过 `dotnet publish` 命令使用以下开关来发布 SCD：

- `-r <RID>` 此开关使用标识符 (RID) 来指定目标平台。有关运行时标识符的详细信息，请参阅运行时标识符 (RID) 目录。
- `--self-contained true` 此开关告知 .NET SDK 创建可执行文件作为 SCD。

[illegible]	SDK [illegible]	[illegible]
独立部署	2.1	`dotnet publish -c Release -r <RID> --self-contained true`
	3.1	`dotnet publish -c Release -r <RID> --self-contained true`
	5.0	`dotnet publish -c Release -r <RID> --self-contained true`

> **NOTE**
> 可以通过启用全局固定模式来降低部署的总大小。此模式适用于不具有全局意识且可以使用固定区域性的格式约定、大小写约定以及字符串比较和排序顺序的应用程序。有关全局固定模式及其启用方式的详细信息，请参阅 .NET Core 全局固定模式。

另请参阅

- .NET 应用程序部署概述
- .NET 运行时标识符 (RID) 目录

如何使用 .NET CLI 创建 NuGet 包

2021/11/16 •

> **NOTE**
> 下例是关于使用 Unix 的命令行。此处显示的 `dotnet pack` 命令工作方式与在 Windows 上相同。

对于 .NET Standard 和 .NET Core，所有库都应以 NuGet 包方式发布。实际上，这是所有 .NET 标准库的发布和使用方式。可以使用 `dotnet pack` 命令轻松实现此操作。

假设你刚编写了一个很棒的新库，并想通过 NuGet 发布。你就可以使用跨平台工具创建一个 NuGet 包，完全照做就行！ 下例假定使用一个名为“SuperAwesomeLibrary”的库，该库以 `netstandard1.0` 为目标。

如果存在可传递的依赖项，也就是说，如果一个项目依赖于另一个包，则在创建 NuGet 包前，确保使用 `dotnet restore` 命令还原整个解决方案的包。否则将导致 `dotnet pack` 命令不能正常运行。

无需运行 `dotnet restore`，因为它由所有需要还原的命令隐式运行，如 `dotnet new`、`dotnet build`、`dotnet run`、`dotnet test`、`dotnet publish` 和 `dotnet pack`。若要禁用隐式还原，请使用 `--no-restore` 选项。

在执行显式还原有意义的某些情况下，例如 Azure DevOps Services 中的持续集成生成中，或在需要显式控制还原发生时间的生成系统中，`dotnet restore` 命令仍然有用。

有关如何使用 NuGet 源的信息，请参阅 `dotnet restore` 文档。

确认包已还原后，可以导航到库所在的目录：

```
cd src/SuperAwesomeLibrary
```

然后，只需在命令行中输入一个命令：

```
dotnet pack
```

/bin/Debug 文件夹现在如下所示：

```
$ ls bin/Debug
netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

这将生成能够进行调试的包。如果想要生成二进制文件版本的 NuGet 包，只需添加 `--configuration`（或 `-c`）开关并使用 `release` 作为参数。

```
dotnet pack --configuration release
```

/bin 文件夹现在将包含一个 release 文件夹，后者包含的 NuGet 包为进制文件版本：

```
$ ls bin/release
netstandard1.0/
SuperAwesomeLibrary.1.0.0.nupkg
SuperAwesomeLibrary.1.0.0.symbols.nupkg
```

因此，现在可以使用此必需的文件发布 NuGet 包了！

不要混淆 `dotnet pack` 和 `dotnet publish`

务必注意，不是任何时候都涉及 `dotnet publish` 命令。`dotnet publish` 命令用于在同一个包中部署具有所有依赖项的应用程序 - 而不是用于生成通过 NuGet 发布和使用的 NuGet 包。

另请参阅

- 快速入门：创建和发布包

自包含部署运行时前滚

2021/11/16 •

.NET Core 自包含应用程序部署包括 .NET Core 库和 .NET Core 运行时。从 .NET Core 2.1 SDK（版本 2.1.300）开始，自包含应用程序部署在计算机上发布最高版本的修补程序运行时。默认情况下，自包含部署的 `dotnet publish` 选择作为发布计算机上 SDK 一部分而安装的最新版本。这让部署的应用程序在 `publish` 期间能与安全修补程序（以及其他修补程序）配合运行。若要获取新的修补程序，需要重新发布应用程序。自包含应用程序是通过在 `dotnet publish` 命令上指定 `-r <RID>` 创建的，或是通过在项目文件 (csproj / vbproj) 或命令行中指定运行时标识符 (RID) 创建的。

修补程序版本前滚概述

`restore`、`build` 和 `publish` 是可以单独运行的 `dotnet` 命令。运行时选择属于 `restore` 操作，而不是 `publish` 或 `build` 操作。如果调用 `publish`，则会选择最新的修补程序版本。如果调用带有 `--no-restore` 参数的 `publish`，则可能不会获取所需的修补程序版本，因为没有先使用新的自包含应用程序发布策略执行 `restore`。在这种情况下，会出现生成错误，并显示以下类似文本：

“已使用 Microsoft.NETCore.App 版本 2.0.0 还原该项目，但是按照当前设置，将改为使用版本 2.0.6。若要解决此问题，请确保使用相同的设置执行还原和后续操作，例如生成或发布。通常，如果在生成或发布过程中设置了 RuntimeIdentifier 属性，但在还原过程中未设置此属性，则会出现此问题。”

> **NOTE**
> `restore` 和 `build` 可以作为另一个命令（例如 `publish`）的组成部分隐式运行。当作为另一个命令的组成部分隐式运行时，它们会带有附加的上下文，以便生成正确的项目。如果使用某个运行时（例如 `dotnet publish -r linux-x64`）执行 `publish`，隐式的 `restore` 会还原 linux-x64 运行时的包。如果显示调用 `restore`，则默认情况下它不会还原运行时包，因为没有上下文。

如何避免在 publish 过程中 restore

你可能在进行 `publish` 操作时不需要运行 `restore`。在创建自包含应用程序时，若要避免在 `publish` 过程中进行 `restore`，请执行以下操作：

- 将 `RuntimeIdentifiers` 属性设为一个分号分隔的列表，其中包含所有要发布的 RID。
- 将 `TargetLatestRuntimePatch` 属性设置为 `true`。

使用 dotnet publish 选项的 no-restore 参数

如果要使用同样的项目文件创建自包含应用程序和依赖框架的应用程序，并且想通过 `dotnet publish` 使用 `--no-restore` 参数，请选择以下各项之一：

1. 首选依赖框架的行为。如果是依赖框架的应用程序，则此选项为默认行为。如果是自包含应用程序，并且能使用未带修补程序的 2.1.0 本地运行时，请在项目文件中将 `TargetLatestRuntimePatch` 设为 `false`。
2. 首选自包含行为。如果是自包含应用程序，则此选项为默认行为。如果是依赖框架的应用程序，且需要安装最新版本的修补程序，请在项目文件中将 `TargetLatestRuntimePatch` 设为 `true`。
3. 通过在项目文件中将 `RuntimeFrameworkVersion` 设为特定的修补程序版本，可对运行时框架版本进行显式控制。

单文件部署和可执行文件

2021/11/16 •

通过将所有依赖应用程序的文件捆绑到一个二进制文件中，为应用程序开发人员提供一个具有吸引力的选项，那就是将应用程序作为单个文件进行部署和分发。此部署模型从 .NET Core 3.0 开始提供，在 .NET 5 中进行了增强。之前在 .NET Core 3.0 中，当用户运行单文件应用时，.NET Core 主机会先将所有文件提取到一个目录，然后再运行该应用程序。.NET 5 改进了这一体验，它可直接运行代码，无需从应用中提取文件。

单文件部署可用于依赖框架的部署模型和独立应用程序。独立应用程序中单个文件的大小很大，因为它包含运行时和框架库。单文件部署选项可与 ReadyToRun 和 Trim 发布选项结合使用。

单个文件部署与 Windows 7 不兼容。

与 .NET 3.x 的输出差异

在 .NET Core 3.x 中，作为一个文件发布将只生成一个文件，其中包含应用本身、依赖项和发布期间文件夹中的所有其他文件。当应用启动时，系统将单文件应用提取到一个文件夹，并从该文件夹中运行。从 .NET 5 开始，仅将托管的 DLL 与应用捆绑到一个可执行文件中。当应用启动时，托管的 DLL 将被提取并加载到内存中，从而避免提取到文件夹中。在 Windows 上，这意味着托管二进制文件将嵌入单文件捆绑包中，但核心运行时本身的本机二进制文件是单独的文件。若要嵌入这些文件以进行提取并恰好获取一个输出文件（与在 .NET Core 3.x 中类似），请将属性 `IncludeNativeLibrariesForSelfExtract` 设置为 `true`。有关提取的详细信息，请参阅添加本机库。

API 不兼容

某些 API 与单文件部署不兼容，如果应用程序使用这些 API，可能需要进行修改。如果使用第三方框架或包，则它们也可能使用了这样的 API 并需要修改。出现问题的最常见原因是依赖于应用程序附带的文件或 DLL 的文件路径。

下表提供了用于单文件的相关运行时库 API 详细信息。

API	行为
`Assembly.CodeBase`	引发 PlatformNotSupportedException。
`Assembly.EscapedCodeBase`	引发 PlatformNotSupportedException。
`Assembly.GetFile`	引发 IOException。
`Assembly.GetFiles`	引发 IOException。
`Assembly.Location`	返回空字符串。
`AssemblyName.CodeBase`	返回 `null`。
`AssemblyName.EscapedCodeBase`	返回 `null`。
`Module.FullyQualifiedName`	返回值为 `<Unknown>` 的字符串，或引发异常。
`Module.Name`	返回值为 `<Unknown>` 的字符串。

我们提供了关于修复常见问题的一些建议：

- 若要访问可执行文件旁边的文件，请使用 AppContext.BaseDirectory。
- 若要查找可执行文件的文件名，请使用 Environment.GetCommandLineArgs() 的第一个元素；从 .NET 6 开始，请使用 ProcessPath 中的文件名。
- 若要避免完全发送松散文件，请考虑使用嵌入的资源。

附加调试器

在 Linux 上，可以附加到自包含单文件进程或调试故障转储的唯一调试器是有 LLDB 的 SOS。

在 Windows 和 Mac 上，可以使用 Visual Studio 和 VS Code 调试故障转储。要附加到运行自包含单文件可执行文件，需要额外的文件：*mscordbi.{dll,so}*

如果没有此文件，Visual Studio 可能会生成错误“无法附加到进程。未安装调试组件。” 而 VS Code 可能生成错误“未能附加到进程:未知错误:0x80131c3c。”

若要修复这些错误，需要将 mscordbi 复制到可执行文件旁边。mscordbi 默认 `publish` 到具有应用程序运行时 ID 的子目录中。例如，如果使用适用于 Windows 的 `dotnet` CLI 并使用 `-r win-x64` 参数发布自包含单文件可执行文件，则可执行文件将置于 bin/Debug/net5.0/win-x64/publish 中。mscordbi.dll 的副本将存在于 bin/Debug/net5.0/win-x64 中。

添加本机库

默认情况下，单文件不捆绑本机库。在 Linux 上，我们将运行时预链接到捆绑包中，并且仅将应用程序本机库部署到单文件应用所在的目录中。在 Windows 上，我们仅预链接托管代码，而且运行时库和应用程序本机库都部署到单文件应用所在的目录中。目的是确保提供良好的调试体验，这要求将本机文件从单文件中排除。

从 .NET 6 开始，运行时预链接到所有平台上的捆绑包中。

可选择设置标志 `IncludeNativeLibrariesForSelfExtract`，从而在单文件捆绑包中包含本机库，但在运行单文件应用程序时，这些文件将被提取到客户端计算机上的目录中。

指定 `IncludeAllContentForSelfExtract` 将在运行可执行文件之前提取所有文件（甚至是托管程序集）。这将保留原始 .NET Core 单文件部署行为。

> **NOTE**
>
> 如果进行提取，则会在应用启动前将文件提取到磁盘：
>
> - 如果环境变量 `DOTNET_BUNDLE_EXTRACT_BASE_DIR` 设为路径，则会将文件提取到该路径下的目录。
> - 此外，如果在 Linux 或 MacOS 上运行，则会将文件提取到 `$HOME/.net` 下的目录。
> - 如果在 Windows 上运行，则会将文件提取到 `%TEMP%/.net` 下的目录。
>
> 若要防止篡改，这些目录不可由具有不同权限的用户或服务写入（在大多数 Linux 和 MacOS 系统上，不是 /tmp 或 /var/tmp）。

NOTE

在某些 Linux 环境下(例如在 systemd 下)，由于未定义 `$HOME`，无法进行默认提取。在这种情况下，建议显式设置 `$DOTNET_BUNDLE_EXTRACT_BASE_DIR`。

对于 systemd，还可将服务单元文件中的 DOTNET_BUNDLE_EXTRACT_BASE_DIR 定义为 `%h/.net`，systemd 可为运行该服务的帐户将其正确地扩展为 `$HOME/.net`。

```
[Service]
Environment="DOTNET_BUNDLE_EXTRACT_BASE_DIR=%h/.net"
```

其他注意事项

单文件应用程序旁边将具有所有相关的 PDB 文件，并且默认情况下不会绑定。如果要在生成的项目的程序集内包含 PDB，请将 `DebugType` 设置为 `embedded`，如下方所详述。

托管的 C++ 应组件不太适合进行单文件部署，建议使用 C# 或其他非托管 C++ 语言编写应用程序来实现单文件兼容。

将文件排除在嵌入之外

通过设置以下元数据，可明确指定不在单文件中嵌入某些文件：

```
<ExcludeFromSingleFile>true</ExcludeFromSingleFile>
```

例如，若要将某些文件放置在发布目录中，但不将它们捆绑到单文件中：

```
<ItemGroup>
  <Content Update="Plugin.dll">
    <CopyToPublishDirectory>PreserveNewest</CopyToPublishDirectory>
    <ExcludeFromSingleFile>true</ExcludeFromSingleFile>
  </Content>
</ItemGroup>
```

在绑定内包括 PDB 文件

可以使用下面的设置将程序集的 PDB 文件嵌入到程序集本身 (`.dll`)。由于符号是程序集的一部分，因此，它们也会是单文件应用程序的一部分：

```
<DebugType>embedded</DebugType>
```

例如，将以下属性添加到程序集的项目文件，以将 PDB 文件嵌入到该程序集：

```
<PropertyGroup>
  <DebugType>embedded</DebugType>
</PropertyGroup>
```

在单文件应用中压缩程序集

从 .NET 6 开始，在嵌入式程序集上启用压缩后，可以创建单文件应用。将 `EnableCompressionInSingleFile` 属性设置为 `true` 即可实现此目的。生成的单个文件将包含所有已压缩的嵌入式程序集，这可以显著减小可执行文件的大小。压缩会导致性能下降。在应用程序启动时，必须将程序集解压缩到内存中，这需要花费一些时间。在

使用压缩之前，建议衡量启用压缩造成的大小影响和启动开销影响，因为这种影响在不同应用程序之间有很大的差异。

发布单文件应用 - 示例项目文件

下面是指定单文件发布的示例项目文件：

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <PublishSingleFile>true</PublishSingleFile>
    <SelfContained>true</SelfContained>
    <RuntimeIdentifier>win-x64</RuntimeIdentifier>
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>

</Project>
```

这些属性具有下列函数：

- `PublishSingleFile` - 启用单文件发布。此外，还会在 `dotnet build` 期间启用单一文件警告。
- `SelfContained` - 确定应用是独立的还是依赖于框架的。
- `RuntimeIdentifier` - 指定目标 OS 和 CPU 类型。默认情况下，还会设置 `<SelfContained>true</SelfContained>`。
- `PublishReadyToRun` - 启用预先 (AOT) 编译。

注意：

- 单文件应用始终特定于 OS 和体系结构。需要为每个配置发布，例如 Linux x64、Linux ARM64、Windows x64 等。
- 单个文件中包含运行时配置文件，例如 *.runtimeconfig.json 和 *.deps.json 。如果需要其他配置文件，可将其放在单个文件旁边。

发布单文件应用 - CLI

使用 dotnet publish 命令发布单文件应用程序。

1. 将 `<PublishSingleFile>true</PublishSingleFile>` 添加到项目文件。

   这将针对独立发布生成一个单一文件应用。它还会在生成期间显示单一文件兼容性警告。

   ```
   <PropertyGroup>
       <PublishSingleFile>true</PublishSingleFile>
   </PropertyGroup>
   ```

2. 使用 `dotnet publish -r <RID>` 针对特定运行时标识符发布应用

   以下示例将 Windows 应用作为独立的单一文件应用程序发布。

   ```
   dotnet publish -r win-x64
   ```

   以下示例将 Linux 应用作为依赖框架的单一文件应用程序发布。

   ```
   dotnet publish -r linux-x64 --self-contained false
   ```

应在项目文件中设置 `<PublishSingleFile>` 以在生成期间启用单一文件分析，但也可以将这些选项作为

`dotnet publish` 参数传递：

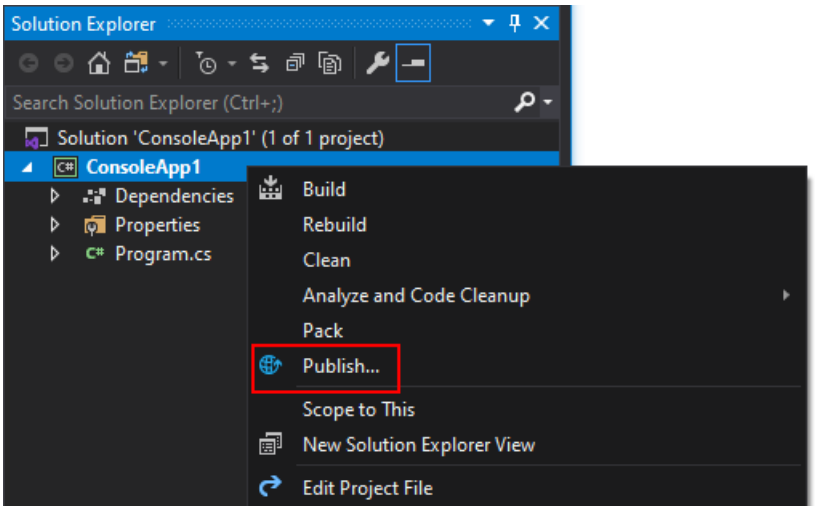
```
dotnet publish -r linux-x64 -p:PublishSingleFile=true --self-contained false
```

有关详细信息，请参阅使用 .NET CLI 发布 .NET Core 应用。

发布单文件应用 - Visual Studio

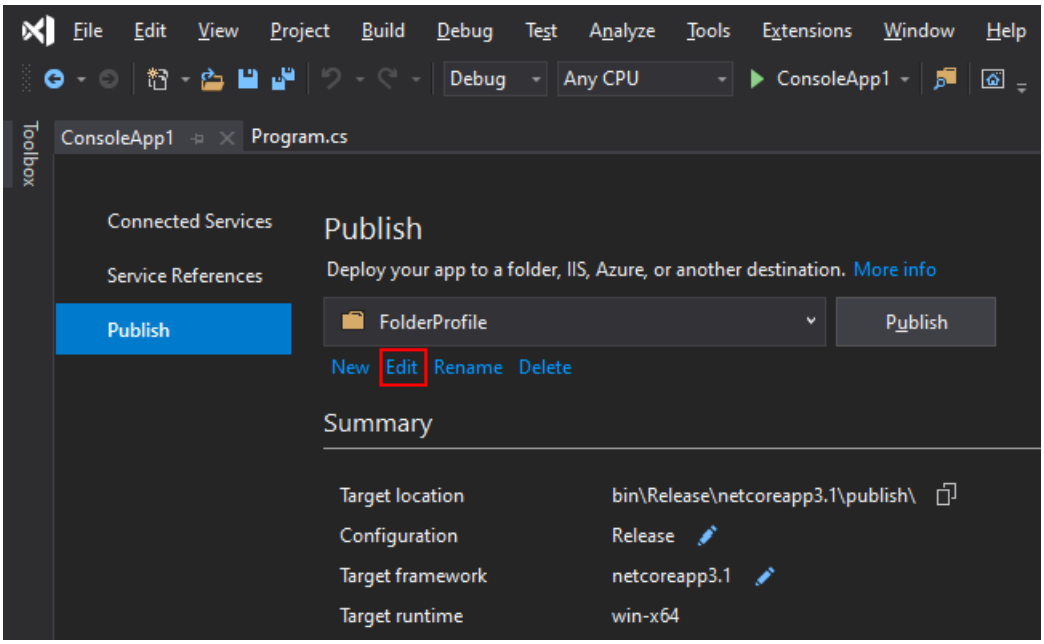
Visual Studio 创建可重用的发布配置文件，用于控制应用程序的发布方式。

1. 将 `<PublishSingleFile>true</PublishSingleFile>` 添加到项目文件。
2. 在“解决方案资源管理器”窗格中，右键单击要发布的项目 。选择“发布”。



如果还没有发布配置文件，请按照说明创建一个并选择“文件夹”目标类型 。

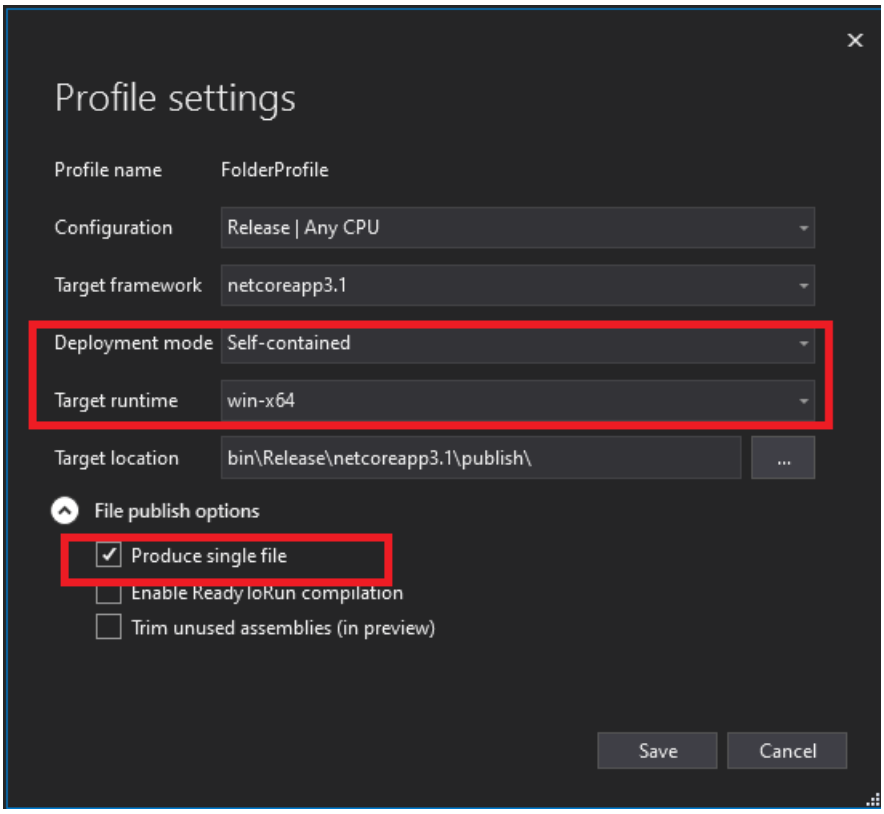
3. 选择“编辑” 。



4. 在“配置文件设置”对话框中，设置以下选项：

- 将“部署模式”设置为“独立式”或“依赖于框架”。
- 将“目标运行时”设置为要发布到的平台 。（必须是除“可移植”以外的设置。）
- 选择“生成单个文件”。

选择“保存”保存设置并返回到“发布”对话框 。



5. 选择“发布”，将应用作为单个文件发布。

有关详细信息，请参阅使用 Visual Studio 发布 .NET Core 应用。

发布单文件应用 - Visual Studio for Mac

Visual Studio for Mac 不提供将应用作为单个文件发布的选项。你需要按照发布单文件应用 - CLI 部分中的说明手动发布。有关详细信息，请参阅使用 .NET CLI 发布 .NET 应用。

请参阅

- .NET Core 应用程序部署。
- 使用 .NET CLI 发布 .NET 应用。
- 使用 Visual Studio 发布 .NET Core 应用。
- `dotnet publish` 命令。

ReadyToRun 编译

2021/11/16 •

可以通过将应用程序集编译为 ReadyToRun (R2R) 格式来改进 .NET Core 应用程序的启动时间和延迟。R2R 是一种预先 (AOT) 编译形式。

R2R 二进制文件通过减少应用程序加载时实时 (JIT) 编译器需要执行的工作量来改进启动性能。二进制文件包含与 JIT 将生成的内容类似的本机代码。但是，R2R 二进制文件更大，因为它们包含中间语言 (IL) 代码（某些情况下仍需要此代码）和相同代码的本机版本。仅当发布面向特定运行时环境 (RID)（如 Linux x64 或 Windows x64）的应用时 R2R 才可用。

若要将项目编译为 ReadyToRun，必须在将 PublishReadyToRun 属性设置为 `true` 时发布应用程序。

可以通过两种方式将应用发布为 ReadyToRun：

1. 向 dotnet publish 命令直接指定 PublishReadyToRun 标志。有关详细信息，请参阅 dotnet publish。

```
dotnet publish -c Release -r win-x64 -p:PublishReadyToRun=true
```

2. 在项目中指定属性。

   - 向项目中添加 `<PublishReadyToRun>` 设置。

```
<PropertyGroup>
  <PublishReadyToRun>true</PublishReadyToRun>
</PropertyGroup>
```

   - 在不使用任何特殊参数的情况下发布应用程序。

```
dotnet publish -c Release -r win-x64
```

使用 ReadyToRun 功能的影响

预先编译会对应用程序性能产生复杂的性能影响，这种影响可能很难预测。通常情况下，程序集的大小将增长到两到三倍。这种文件物理大小的增加可能会降低从磁盘加载程序集的性能，并增加进程的工作集。但相对地，在运行时编译的方法数通常会大幅减少。因此，启用 ReadyToRun，包含大量代码的大多数应用程序都会获得很大的性能增益。由于 .NET 运行时库已使用 ReadyToRun 进行预编译，因此启用 ReadyToRun 时，具有少量代码的应用程序很可能不会获得显著改进。

这里讨论的启动改进不仅适用于应用程序启动，还适用于在应用程序中首次使用任何代码。例如，ReadyToRun 可用于降低在 ASP.NET 应用程序中首次使用 Web API 时的响应延迟。

通过分层编译进行交互

预先生成的代码优化程度不如 JIT 生成的代码。为了解决此问题，分层编译将使用 JIT 生成的方法替换常用的 ReadyToRun 方法。

如何选择预编译的一组程序集？

SDK 将预编译随应用程序一起分发的程序集。对于独立应用程序，这组程序集将包括框架。C++/CLI 二进制文件不适合 ReadyToRun 编译。

若要从 ReadyToRun 处理中排除特定程序集，请使用 `<PublishReadyToRunExclude>` 列表。

```
<ItemGroup>
  <PublishReadyToRunExclude Include="Contoso.Example.dll" />
</ItemGroup>
```

如何选择要预编译的一组方法？

编译器会尝试预编译尽可能多的方法。但由于各种原因，不应预期使用 ReadyToRun 功能会阻止 JIT 执行。这些原因包括(但不限于)：

- 使用在单独的程序集中定义的泛型类型。
- 与本机代码互操作。
- 使用编译器无法证明可在目标计算机上安全使用的硬件内部函数。
- 某些异常 IL 模式。
- 通过反射或 LINQ 创建动态方法。

与探查器结合使用的符号生成

使用 ReadyToRun 编译应用程序时，探查器可能需要使用符号来检查生成的 ReadyToRun 文件。若要启用符号生成，请指定 `<PublishReadyToRunEmitSymbols>` 属性。

```
<PropertyGroup>
  <PublishReadyToRunEmitSymbols>true</PublishReadyToRunEmitSymbols>
</PropertyGroup>
```

这些符号将放置在发布目录中。对于 Windows，其文件扩展名为 .ni.pdb；对于 Linux，其文件扩展名为 .r2rmap。这些文件通常不会重新分发给最终客户，而是一般存储在符号服务器中。通常，这些符号适用于调试与应用程序启动相关的性能问题，因为分层编译会将 ReadyToRun 生成的代码替换为动态生成的代码。但是，如果尝试分析禁用分层编译的应用程序，这些符号会很有用。

复合 ReadyToRun

常规 ReadyToRun 编译会生成可单独处理和操作的二进制文件。从 .NET 6 开始，添加了对复合 ReadyToRun 编译的支持。复合 ReadyToRun 会编译一组必须同时分发的程序集。这样做的好处是编译器能够更好地进行优化，并减少无法通过 ReadyToRun 进程编译的方法集。但缺点是，编译速度显著降低，应用程序的整体文件大小显著增加。由于这些缺点，因此仅建议将复合 ReadyToRun 用于禁用分层编译的应用程序，或者用于在 Linux 上运行且通过自包含部署寻求最佳启动时间的应用程序。若要启用复合 ReadyToRun 编译，请指定 `<PublishReadyToRunComposite>` 属性。

```
<PropertyGroup>
  <PublishReadyToRunComposite>true</PublishReadyToRunComposite>
</PropertyGroup>
```

> **NOTE**
> 在 .NET 6 中，仅独立部署支持复合 ReadyToRun。

跨平台/体系结构限制

对于某些 SDK 平台，ReadyToRun 编译器可以进行针对其他目标平台的交叉编译。

下表描述了在面向 .NET 6 及更高版本时支持的编译目标。

SDK ㄖㄨ	ㄖㄨㄧㄧㄧㄧㄧ
Windows X64	Windows（X86、X64、ARM64）、Linux（X64、ARM32、ARM64）、macOS（X64、ARM64）
Windows X86	Windows (X86)、Linux (ARM32)
Linux X64	Linux（X64、ARM32、ARM64）、macOS（X64、ARM64）
Linux ARM32	Linux ARM32
Linux ARM64	Linux（X64、ARM32、ARM64）、macOS（X64、ARM64）
macOS X64	Linux（X64、ARM32、ARM64）、macOS（X64、ARM64）
macOS ARM64	Linux（X64、ARM32、ARM64）、macOS（X64、ARM64）

下表描述了在面向 .NET 5 及更低版本时支持的编译目标。

SDK ㄖㄨ	ㄖㄨㄧㄧㄧㄧㄧ
Windows X64	Windows X86、Windows X64、Windows ARM64
Windows X86	Windows X86、Windows ARM32
Linux X64	Linux X86、Linux X64、Linux ARM32、Linux ARM64
Linux ARM32	Linux ARM32
Linux ARM64	Linux ARM64
macOS X64	macOS X64

剪裁独立部署和可执行文件

2021/11/16 •

自 .NET 问世以来，依赖框架的部署模型是最成功的部署模型。在这种情况下，应用程序开发人员仅将应用程序和第三方程序集捆绑在一起，期望 .NET 运行时和运行时库在客户端计算机中可用。此部署模型仍是最新 .NET 版本中的主导模型，但在某些情况下，依赖框架的模型并不是最佳模型。替代方法是发布自包含的应用程序，其中 .NET 运行时和运行时库与应用程序和第三方程序集捆绑在一起。

剪裁自包含部署模型是自包含部署模型的专用版本，该模型已优化以减小部署大小。对于一些客户端场景（如 Blazor 应用程序），最大程度地减小部署大小是很重要的要求。根据应用程序的复杂性，只引用 framework 程序集的子集，运行该应用程序时需要每个程序集中代码的子集。不需要这些未使用的库部分，可从打包的应用程序中进行剪裁。

但是，由于无法可靠地分析各种有问题的代码模式（主要集中在反射使用），应用程序的生成时间分析可能会导致运行时失败。为了缓解这些问题，只要剪裁器无法完全分析代码模式，就会生成警告。有关剪裁警告的含义以及警告解决方法的信息，请参阅剪裁警告简介。

> **NOTE**
> 仅 .NET 6 及更高版本支持剪裁。

导致剪裁问题的组件

导致生成时间分析难题的任何代码都不适合剪裁。在应用程序使用时出现问题的一些常见编码模式源自不受限制的反射用法，以及在生成时不可见的外部依赖项。不受限制的反射的示例是旧的序列化程序（如 XML 序列化），并且不可见外部依赖项的示例是内置 COM。有关已知的不兼容，请参阅已知剪裁不兼容。若要解决应用程序中的剪裁警告，请参阅剪裁警告简介；若要使库与剪裁兼容，请参阅准备 .NET 库以进行剪裁。

启用剪裁

1. 将 `<PublishTrimmed>true</PublishTrimmed>` 添加到项目文件。

   这会在进行自包含发布时生成一个经过裁剪的应用。它还会关闭不兼容剪裁的功能，并在生成期间显示剪裁兼容性警告。

   ```
   <PropertyGroup>
       <PublishTrimmed>true</PublishTrimmed>
   </PropertyGroup>
   ```

2. 然后使用 dotnet publish 命令或 Visual Studio 发布应用。

使用 CLI 发布

以下示例将 Windows 应用发布为经过剪裁的独立应用程序。

```
dotnet publish -r win-x64
```

仅独立应用支持剪裁。

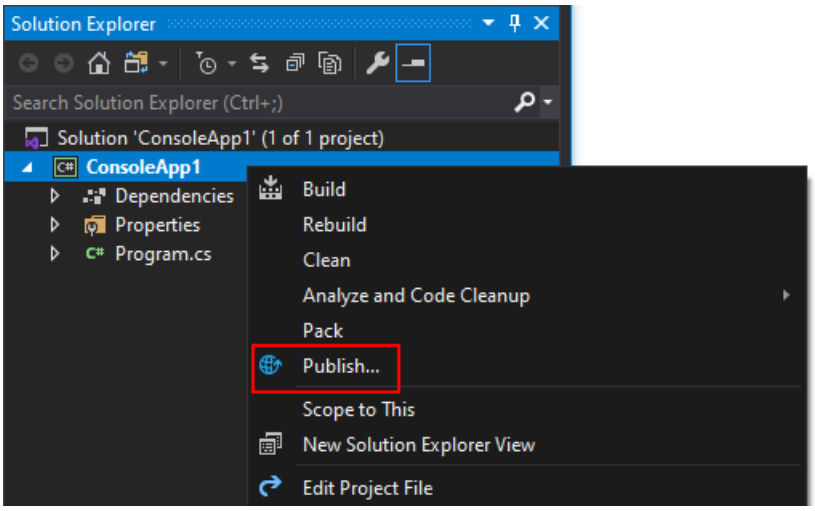
应在项目文件中设置 `<PublishTrimmed>`，以便在 `dotnet build` 期间禁用不兼容剪裁的功能，但也可以将这些选项作为 `dotnet publish` 参数传递：

```
dotnet publish -r win-x64 -p:PublishTrimmed=true
```

有关详细信息，请参阅使用 .NET CLI 发布 .NET 应用。

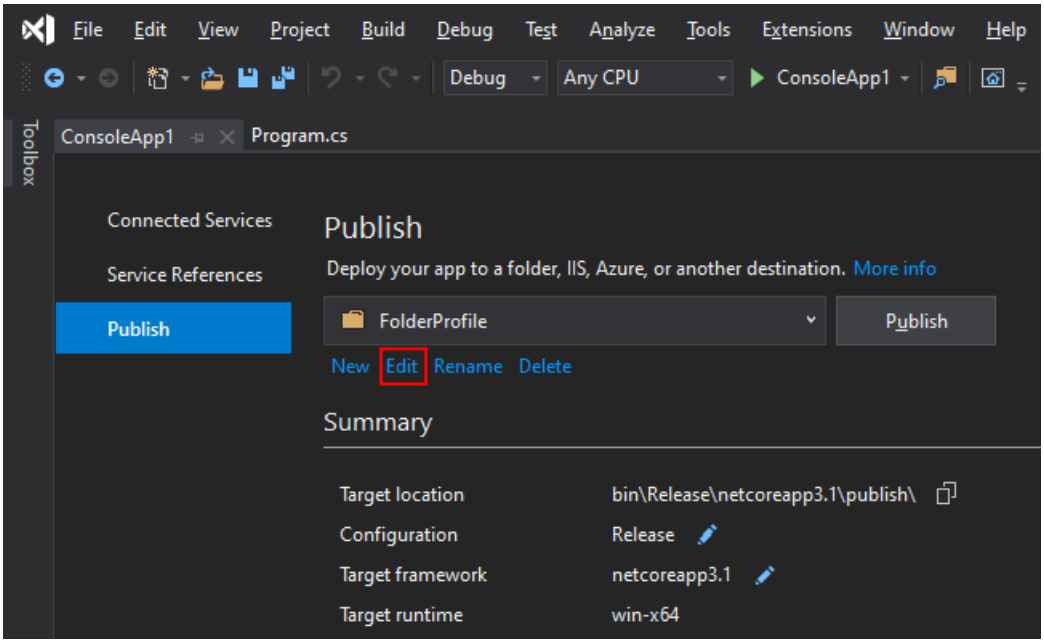
使用 Visual Studio 发布

1. 在“解决方案资源管理器”窗格中，右键单击要发布的项目 。选择“发布...” 。



   如果还没有发布配置文件，请按照说明创建一个并选择“文件夹”目标类型 。

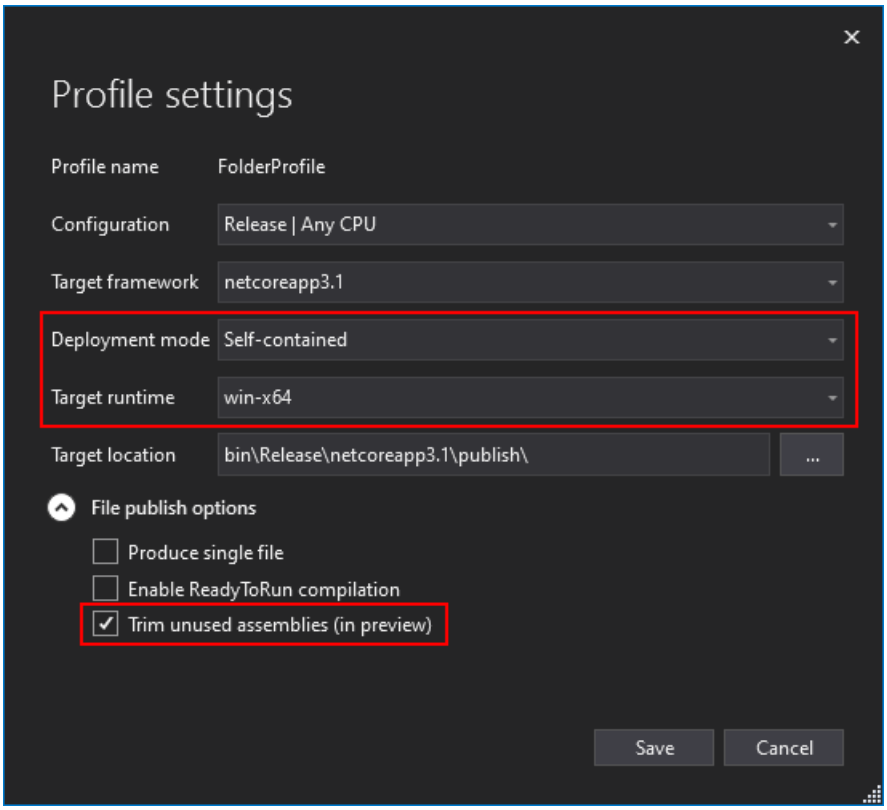
2. 选择“编辑” 。



3. 在“配置文件设置”对话框中，设置以下选项：

   - 将“部署模式”设置为“自包含” 。
   - 将“目标运行时”设置为要发布到的平台 。
   - 选择“剪裁未使用的程序集(预览版)” 。

   选择“保存”保存设置并返回到“发布”对话框 。



4. 选择“发布”以发布剪裁过的应用 。

有关详细信息，请参阅使用 Visual Studio 发布 .NET Core 应用。

使用 Visual Studio for Mac 发布

Visual Studio for Mac 不提供发布应用的选项。你需要按照使用 CLI 发布部分中的说明手动发布。有关详细信息，请参阅使用 .NET CLI 发布 .NET 应用。

请参阅

- .NET Core 应用程序部署。
- 使用 .NET CLI 发布 .NET 应用。
- 使用 Visual Studio 发布 .NET Core 应用。
- dotnet publish 命令。

剪裁警告简介

2021/11/16 •

从概念上讲，剪裁非常简单：发布应用程序时，.NET SDK 会分析整个应用程序并删除所有未使用的代码。然而，可能很难确定什么是未使用的，或者更准确地说是使用了什么。

为防止剪裁应用程序时行为发生变化，.NET SDK 通过"剪裁警告"提供剪裁兼容性的静态分析。当剪裁器发现可能与剪裁不兼容的代码时，剪裁器会产生剪裁警告。与剪裁不兼容的代码可能会在剪裁后的应用程序中产生行为变更，甚至崩溃。理想情况下，所有使用剪裁的应用程序都不应有剪裁警告。如果有任何剪裁警告，则应在剪裁后彻底测试应用，以确保没有行为变更。

本文章将帮助开发人员了解为什么某些模式会产生剪裁警告，以及如何解决这些警告。

剪裁警告的示例

对于大多数 C# 代码，很容易确定使用了哪些代码以及哪些代码未使用 — 剪裁器可以遍历方法调用、字段和属性引用等，并确定访问了哪些代码。遗憾的是，某些功能（如反射）存在重大问题。考虑下列代码：

```
string s = Console.ReadLine();
Type type = Type.GetType(s);
foreach (var m in type.GetMethods())
{
    Console.WriteLine(m.Name);
}
```

在此示例中，GetType() 动态请求名称未知的类型，然后打印其所有方法的名称。由于在发布时无法知道将使用什么类型名称，因此剪裁器无法知道要在输出中保留哪种类型。可能这段代码在剪裁之前可以工作（只要输入是目标框架中已知存在的内容），但在剪裁后可能会产生空引用异常（由于 `Type.GetType` 返回 null）。

在这种情况下，裁边器会在调用 `Type.GetType` 时发出警告，表明它无法确定应用程序将使用哪种类型。

响应剪裁警告

剪裁警告旨在为剪裁带来可预测性。你可能会看到两个大类别的警告：

1. `RequiresUnreferencedCode`
2. `DynamicallyAccessedMembers`

RequiresUnreferencedCode

RequiresUnreferencedCodeAttribute 简单而广泛：它是一个属性，表示成员已被注释为与剪裁不兼容，这意味着它可能使用反射或其他一些机制来访问可能被剪裁掉的代码。当代码从根本上不兼容剪裁时，或者剪裁依赖项太复杂而无法向剪裁器解释时，将使用此属性。对于使用 C# `dynamic` 关键字、从 LoadFrom(String) 访问类型或其他运行时代码生成技术的方法来说，这通常是正确的。例如：

```
[RequiresUnreferencedCode("Use 'MethodFriendlyToTrimming' instead")]
void MethodWithAssemblyLoad() { ... }

void TestMethod()
{
    // IL2026: Using method 'MethodWithAssemblyLoad' which has 'RequiresUnreferencedCodeAttribute'
    // can break functionality when trimming application code. Use 'MethodFriendlyToTrimming' instead.
    MethodWithAssemblyLoad();
}
```

`RequiresUnreferencedCode` 的解决方法并不多。最好的解决方法是在剪裁时完全避免调用该方法并使用其他与剪裁兼容的方法。如果你正在编写一个库并且是否调用该方法不在你的控制范围内，还可以将 `RequiresUnreferencedCode` 添加到你自己的方法中。这会将你的方法注释为不兼容剪裁。添加 `RequiresUnreferencedCode` 将压制给定方法中的所有剪裁警告，但每当其他人调用它时都会产生警告。出于此原因，库作者将警告"向上冒泡"到公共 API 是最有用的。

如果可以通过某种方式确定调用是安全的，并且不会删除所有需要的代码，还可以使用 UnconditionalSuppressMessageAttribute 取消警告。例如：

```
[RequiresUnreferencedCode("Use 'MethodFriendlyToTrimming' instead")]
void MethodWithAssemblyLoad() { ... }

[UnconditionalSuppressMessage("AssemblyLoadTrimming", "IL2026:RequiresUnreferencedCode",
    Justification = "Everything referenced in the loaded assembly is manually preserved, so it's safe")]
void TestMethod()
{
    MethodWithAssemblyLoad(); // Warning suppressed
}
```

`UnconditionalSuppressMessage` 类似于 `SuppressMessage`，但它可以由 `publish` 和其他生成后工具查看。`SuppressMessage` 和 `#pragma` 指令仅存在于源中，因此不能用于从剪裁器压制警告。抑制剪裁警告时要非常小心：调用现在可能与剪裁兼容，但是当更改可能会更改的代码时，你可能会忘记查看所有抑制。

DynamicallyAccessedMembers

DynamicallyAccessedMembersAttribute 通常是关于反射的。与 `RequiresUnreferencedCode` 不同的是，剪裁器有时可以理解反射，因为它是正确注释的。让我们再看一下原始示例：

```
string s = Console.ReadLine();
Type type = Type.GetType(s);
foreach (var m in type.GetMethods())
{
    Console.WriteLine(m.Name);
}
```

在上面的示例中，真正的问题是 `Console.ReadLine()`。由于可以读取任何类型，因此剪裁器无法知道你是否需要 `System.Tuple` 或 `System.Guid` 或任何其他类型的方法。另一方面，如果代码如下所示，

```
Type type = typeof(System.Tuple);
foreach (var m in type.GetMethods())
{
    Console.WriteLine(m.Name);
}
```

这将很好。在这里，剪裁器可以看到被引用的确切类型：`System.Tuple`。现在，它可以使用流分析来确定它是否需要将所有公共方法保留在 `System.Tuple` 上。那么，`DynamicallyAccessMembers` 是从哪里进来的呢？ 当反射跨多个方法拆分时。

```
void Method1()
{
    Method2(typeof(System.Tuple));
}
void Method2(Type type)
{
    var methods = type.GetMethods();
    ...
}
```

如果你编译上述内容，现在你会看到一个警告：

> 剪裁分析警告 IL2070：net6.Program.Method2(Type)："this"参数在调用"System.Type.GetMethods()"时不满足"DynamicallyAccessedMemberTypes.PublicMethods"。方法"net6.Program.Method2(Type)"的参数"type"没有匹配的注释。源值必须至少声明与在其分配到的目标位置上声明的要求相同的要求。

为了性能和稳定性，不会在方法之间执行流分析，因此需要一个注释来在方法之间传递信息，从反射调用 (`GetMethods`) 到 `Type` 的源 (`typeof`)。在上面的示例中，剪裁器警告指出 `GetMethods` 需要对类型进行 `PublicMethods` 注释，但 `type` 变量没有相同的要求。换句话说，我们需要将要求从 `GetMethods` 传递给调用方：

```
void Method1()
{
    Method2(typeof(System.Tuple));
}
void Method2(
    [DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicMethods)] Type type)
{
    var methods = type.GetMethods();
  ...
}
```

现在警告消失了，因为剪裁器确切地知道要保留哪些成员，以及哪些类型要保留它们。通常，这是处理 `DynamicallyAccessedMembers` 警告的最佳方法：添加注释以便剪裁器知道要保留什么。

与 `RequiresUnreferencedCode` 警告一样，添加 `RequiresUnreferencedCode` 或 `UnconditionalSuppressMessage` 属性也会抑制警告，但这些选项都不会使代码与剪裁兼容，而添加 `DynamicallyAccessedMembers` 则可以。

已知裁剪不兼容

2021/11/16 •

有一些模式已知与剪裁不兼容。随着工具的改进或库修改为与剪裁兼容，其中的某些模式可能会变为与剪裁兼容。

内置 COM 封送

替代项：COM 包装器

自 .NET Framework 1.0 开始，.NET 中已内置了自动 COM 封送功能。该功能使用运行时代码分析在本机 COM 对象和托管 .NET 对象之间自动转换。遗憾的是，剪裁分析无法始终预测需要保留哪些 .NET 代码以进行自动 COM 封送。但是，如果改为使用 COM 包装器，剪裁分析便可以保证正确保留所有使用的代码。

WPF

Windows Presentation Foundation (WPF) 框架大量利用反射，而某些功能则很大程度上依赖于运行时代码检查。在 .NET 6 中，剪裁分析无法保留 WPF 应用程序的所有必需代码。遗憾的是，在剪裁后几乎没有 WPF 应用可运行，因此 .NET 6 SDK 中已禁用对 WPF 的剪裁支持。

Windows 窗体

Windows 窗体框架很少使用反射，但非常依赖于内置 COM 封送。在 .NET 6 中，尚未将其转换为使用 ComWrappers。遗憾的是，几乎没有 Windows 窗体应用可在不使用内置 COM 封送的情况下运行，因此 .NET 6 SDK 中已禁用对 Windows 窗体应用的剪裁支持。

基于反射的序列化程序

替代项：无反射的序列化程序，如源生成的 System.Text.Json。

反射的许多用法都可与剪裁兼容，如剪裁警告简介中所述。但是，序列化程序使用反射的方式往往非常复杂。其中的许多用法无法在生成时进行分析。遗憾的是，最佳选择通常是重写系统以改用源生成。

动态程序集加载和执行

对于支持插件或扩展（通常通过 LoadFrom(String) 等 API）的系统，剪裁和动态程序集加载是一个常见问题。剪裁依赖于在生成时查看所有程序集，以便知道使用了哪些代码，而不会剪裁这些代码。大多数插件系统会动态加载第三方代码，此时剪裁器便无法确定所需的代码。

剪裁选项

2021/11/16 •

以下 MSBuild 属性和项会影响剪裁的独立部署行为。一些选项提及 `ILLink`，这是实现剪裁的基础工具的名称。有关基础工具的详细信息，请参阅剪裁器文档。

.NET Core 3.0 中引入了 `PublishTrimmed` 剪裁功能。其他选项仅在 .NET 5 及更高版本中可用。

启用剪裁

- `<PublishTrimmed>true</PublishTrimmed>`

  在发布期间启用剪裁。这还会关闭不兼容剪裁的功能，并在生成期间启用剪裁分析。

将此设置放入项目文件中，以确保不只是在 `dotnet publish` 期间，在 `dotnet build` 期间也会应用该设置。

此设置剪裁已配置用于剪裁的所有程序集。在 .NET 6 的 `Microsoft.NET.Sdk` 中，将剪裁具有 `[AssemblyMetadata("IsTrimmable", "True")]` 的所有程序集，框架程序集就是这种情况。在 .NET 5 中，netcoreapp 运行时包中的框架程序集配置为通过 `<IsTrimmable>` MSBuild 元数据进行剪裁。其他 SDK 可定义不同的默认值。

从 .NET 6 开始，此设置还启用兼容剪裁的 Roslyn 分析器，并禁用不兼容剪裁的功能。

剪裁粒度

以下粒度设置控制如何丢弃未充分利用的 IL。可将其设置为影响所有剪裁器输入程序集的属性，或者设置为可替代该属性设置的单个程序集上的元数据。

- `<TrimMode>link</TrimMode>`

  启用成员级剪裁，这会从类型中删除未使用的成员。这是 .NET 6+ 中的默认设置。

- `<TrimMode>copyused</TrimMode>`

  启用程序集级剪裁；如果使用程序集的任何部分（以静态理解的方式），则将保留整个程序集。

具有 `<IsTrimmable>true</IsTrimmable>` 元数据但没有显式 `TrimMode` 的程序集将使用全局 `TrimMode`。`Microsoft.NET.Sdk` 的默认 `TrimMode` 在 .NET 6+ 中为 `link`，在先前版本中为 `copyused`。

剪裁其他程序集

在 .NET 6+ 中，`PublishTrimmed` 使用以下程序集级别的属性剪裁程序集：

```
[AssemblyMetadata("IsTrimmable", "True")]
```

框架库具有此属性。在 .NET 6+ 中，还可选择按名称指定程序集（不带 `.dll` 扩展名），在不使用此属性的情况下对库进行剪裁。

```
<ItemGroup>
  <TrimmableAssembly Include="MyAssembly" />
</ItemGroup>
```

这等效于为 `ManagedAssemblyToLink` 中的程序集设置 MSBuild 元数据 `<IsTrimmable>true</IsTrimmable>`（见下文）。

剪裁后的程序集

发布剪裁后的应用时，SDK 将计算名为 `ManagedAssemblyToLink` 的 `ItemGroup`，这表示要进行剪裁处理的文件集。`ManagedAssemblyToLink` 可能具有控制每个程序集剪裁行为的元数据。若要设置此元数据，请在内置的 `PrepareForILLink` 目标运行之前创建一个目标。下面的示例演示如何启用 `MyAssembly` 的剪裁。

```
<Target Name="ConfigureTrimming"
        BeforeTargets="PrepareForILLink">
  <ItemGroup>
    <ManagedAssemblyToLink Condition="'%(Filename)' == 'MyAssembly'">
      <IsTrimmable>true</IsTrimmable>
    </ManagedAssemblyToLink>
  </ItemGroup>
</Target>
```

还可为具有 `[AssemblyMetadata("IsTrimmable", "True"])` 的程序集设置 `<IsTrimmable>false</IsTrimmable>`，用它来替代库创建者指定的剪裁行为。

请勿在 `ManagedAssemblyToLink` 中添加或移除项，因为 SDK 会在发布过程中计算此集，并期望它没有发生更改。支持的元数据为：

- `<IsTrimmable>true</IsTrimmable>`

  控制是否剪裁给定的程序集。

- `<TrimMode>copyused</TrimMode>` 或 `<TrimMode>link</TrimMode>`

  控制此程序集的剪裁粒度。这优先于全局 `TrimMode`。在程序集上设置 `TrimMode` 意味着 `<IsTrimmable>true</IsTrimmable>`。

- `<TrimmerSingleWarn>True</TrimmerSingleWarn>` 或 `<TrimmerSingleWarn>False</TrimmerSingleWarn>`

  控制是否显示此程序集的单个警告。

根程序集

所有没有 `<IsTrimmable>true</IsTrimmable>` 的程序集都被视为分析的根，这意味着将保留它们及其所有静态理解的依赖项。其他程序集的名称可能是根（无 `.dll` 扩展名）：

```
<ItemGroup>
  <TrimmerRootAssembly Include="MyAssembly" />
</ItemGroup>
```

根描述符

指定分析的根的另一种方法是使用剪裁器描述符格式的 XML 文件。这使你可以查找特定的成员而非整个程序集。

```
<ItemGroup>
  <TrimmerRootDescriptor Include="MyRoots.xml" />
</ItemGroup>
```

例如，`MyRoots.xml` 可能会查找由应用程序动态访问的特定方法：

```
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyAssembly.MyClass">
      <method name="DynamicallyAccessedMethod" />
    </type>
  </assembly>
</linker>
```

分析警告

剪裁将删除不可静态访问的 IL。使用反射或其他模式来创建动态依赖项的应用可能会因剪裁而中断。警告此类模式：

- `<SuppressTrimAnalysisWarnings>false</SuppressTrimAnalysisWarnings>`

  启用剪裁分析警告。

这将包括有关整个应用的警告，其中包括你自己的代码、库代码以及框架代码。

Roslyn 分析器

在 .NET 6+ 中设置 `PublishTrimmed` 还将启用 Roslyn 分析器，该分析器会显示一组受限的分析警告。还可独立于 `PublishTrimmed` 启用或禁用该分析器。

- `<EnableTrimAnalyzer>true</EnableTrimAnalyzer>`

  为一部分剪裁分析警告启用 Roslyn 分析器。

警告版本

剪裁分析使用的 `AnalysisLevel` 属性可跨 SDK 控制分析警告版本。还有一个可独立控制剪裁分析警告版本的属性（类似于编译器 `WarningLevel`）：

- `<ILLinkWarningLevel>5</ILLinkWarningLevel>`

  仅发出给定级别或更低级别的警告。这可以是 `9999`，以包含所有警告版本。

取消警告

可以使用工具链遵循的常用 MSBuild 属性来取消单个警告代码，包括 `NoWarn`、`WarningsAsErrors`、`WarningsNotAsErrors` 和 `TreatWarningsAsErrors`。还有另一个选项，可单独控制 ILLink 警告即错误的行为：

- `<ILLinkTreatWarningsAsErrors>false</ILLinkTreatWarningsAsErrors>`

  请勿将 ILLink 警告视为错误。在全局将编译器警告视为错误时，这对于避免将剪裁分析警告转换为错误可能很有用。

显示详细警告

在 .NET 6+ 中，对于来自 `PackageReference` 的每个程序集，剪裁分析最多为其生成一个警告，以指示程序集的内部机制与剪裁功能不兼容。你还可显示所有程序集的各个警告：

- `<TrimmerSingleWarn>false</TrimmerSingleWarn>`

  显示所有详细警告，而不是将其折叠为每个程序集的单个警告。

默认显示项目程序集和 `ProjectReference` 的详细警告。也可将 `<TrimmerSingleWarn>` 设置为单个程序集上的元数据，从而控制仅针对该程序集的警告行为。

删除符号

通常会对符号进行剪裁，以匹配剪裁的程序集。还可以删除所有符号：

- `<TrimmerRemoveSymbols>true</TrimmerRemoveSymbols>`

  删除剪裁后的应用程序中的符号，包括嵌入的 PDB 和单独的 PDB 文件。这同时适用于应用程序代码以及符号附带的任何依赖项。

SDK 还可使用属性 `DebuggerSupport` 来禁用调试器支持。禁用调试器支持时，剪裁将自动删除符号（`TrimmerRemoveSymbols` 将默认为 true）。

剪裁框架库功能

框架库的一些功能区域随附有剪裁器指令，这些剪裁器指令可以删除已禁用功能的代码。

- `<AutoreleasePoolSupport>false</AutoreleasePoolSupport>`（默认值）

  删除在支持的平台上创建自动发布池的代码。请参阅用于托管线程的 AutoreleasePool。这是 .NET SDK 的默认值。

- `<DebuggerSupport>false</DebuggerSupport>`

  删除代码可提供更佳的调试体验。此操作也会删除符号。

- `<EnableUnsafeBinaryFormatterSerialization>false</EnableUnsafeBinaryFormatterSerialization>`

  删除 BinaryFormatter 序列化支持。有关详细信息，请参阅 BinaryFormatter 序列化方法已过时。

- `<EnableUnsafeUTF7Encoding>false</EnableUnsafeUTF7Encoding>`

  删除不安全的 UTF-7 编码代码。有关详细信息，请参阅 UTF-7 代码路径已过时。

- `<EventSourceSupport>false</EventSourceSupport>`

  删除与 EventSource 相关的代码或逻辑。

- `<HttpActivityPropagationSupport>false</HttpActivityPropagationSupport>`

  删除与 System.Net.Http 的诊断支持相关的代码。

- `<InvariantGlobalization>true</InvariantGlobalization>`

  删除全球化特定的代码和数据。有关详细信息，请参阅固定模式。

- `<MetadataUpdaterSupport>false</MetadataUpdaterSupport>`

  删除与热重载相关的特定元数据更新逻辑。

- `<UseNativeHttpHandler>true</UseNativeHttpHandler>`

  将 HttpMessageHandler 的默认平台实现用于 Android/iOS，并删除托管实现。

- `<UseSystemResourceKeys>true</UseSystemResourceKeys>`

  删除 `System.*` 程序集的异常消息。当 `System.*` 程序集中引发异常时，该消息将是简化的资源 ID，而不是完整的消息。

这些属性将导致剪裁相关代码，同时还将通过 runtimeconfig 文件禁用功能。有关这些属性（包括相应的 runtimeconfig 选项）的详细信息，请参阅功能切换。某些 SDK 可能具有这些属性的默认值。

剪裁时禁用的框架功能

以下功能不兼容剪裁，因为它们需要不以静态方式引用的代码。这些设置在剪裁应用中默认处于禁用状态。

> **WARNING**
> 启用这些功能的风险由你自担。这些功能可能会中断经过裁剪的应用，且不会执行任何操作来保留动态引用的代码。

- `<BuiltInComInteropSupport>`

  内置的 COM 支持已禁用。

- `<CustomResourceTypesSupport>`

  不支持使用自定义资源类型。将裁剪向自定义资源类型应用反射的 ResourceManager 代码路径。

- `<EnableCppCLIHostActivation>`

  C++/CLI 主机激活已禁用。

- `<EnableUnsafeBinaryFormatterInDesigntimeLicenseContextSerialization>`

  禁止 DesigntimeLicenseContextSerializer 使用 `BinaryFormatter` 序列化。

- `<StartupHookSupport>`

  不支持使用 `DOTNET_STARTUP_HOOKS` 运行 `Main` 之前的代码。有关详细信息，请参阅主机启动挂钩。

准备 .NET 库以进行剪裁

2021/11/16 •

可使用 .NET SDK 通过剪裁从应用及其依赖项中删除未使用的代码，来减小自包含应用的大小。并非所有代码都与剪裁功能兼容，因此 .NET 6 提供剪裁分析警告来检测可能会使已剪裁的应用发生中断的模式。本文介绍如何在这些警告的帮助下准备库进行剪裁，还包含用于解决一些常见问题的建议。

应用中的剪裁警告

在 .NET 6+ 中，当发布应用时，针对未静态理解为与剪裁功能兼容的模式(包括代码和依赖项中的模式)，`PublishTrimmed` 项目文件元素会为其生成剪裁分析警告。

你将会遇到源自你自己的代码和 `ProjectReference` 依赖项的详细警告。还可能会看到有关 `PackageReference` 库的警告，它与 `warning IL2104: Assembly 'SomeAssembly' produced trim warnings` 类似。此警告意味着库中包含的模式不能保证在已剪裁应用的上下文中工作，并可能导致应用出现问题。请考虑联系创建者，查看是否可对库注释来进行剪裁。

若要消除源自应用代码的警告，请查看消除剪裁警告。如果你希望让自己的 `ProjectReference` 库易于剪裁，请按照相关说明启用库剪裁警告。

如果应用仅使用库中与剪裁功能兼容的那部分，请考虑启用剪裁功能(如果尚未对其进行剪裁)。启用剪裁后，如果应用使用库中有问题的部分，则 .NET 只会生成警告。(你还可显示有关该库的详细警告，查看哪些部分存在问题。)

启用库剪裁警告

这些说明演示如何启用和消除静态分析警告，以便准备好库进行剪裁。如果你正在创作库，想要主动让库变得可以剪裁，或者有应用创建者联系你，称其遇到了来自你的库的剪裁警告，那么请按照以下步骤操作。

> **TIP**
> 确保使用 .NET 6 SDK 或更高版本执行这些步骤。它们将无法在以前的版本中正常工作。

设置 IsTrimmable

在库项目中设置 `<IsTrimmable>true</IsTrimmable>`(在 .NET 6+ 中)。这会将程序集标记为“可剪裁”。可剪裁意味着将库用于经过剪裁的应用程序时，程序集可以在最终输出中剪裁掉其未使用的成员。

设置 `<IsTrimmable>true</IsTrimmable>` 可使 Roslyn 分析器具有剪裁兼容性。Roslyn 分析器对于在 IDE 中提供快速反馈很有用，但目前不完整。它未涵盖所有剪裁分析警告，但它理解的那组模式会随着时间的推移进行改进，从而使涵盖范围更加全面。Roslyn 分析器还无法分析你所依赖的引用程序集的实现。请务必按照本文其余部分中列出的步骤操作，以确保库完全兼容剪裁。

或者在库项目中设置 `<EnableTrimAnalyzer>true</EnableTrimAnalyzer>`(.NET 6+ 中)。此操作不会对输出产生任何影响，但会在生成过程中通过 Roslyn 分析器启用剪裁分析。

显示所有警告

若要显示有关库的所有分析警告(包括有关依赖项的警告)，请创建一个单独的应用项目(例如下面引用你的库的项目)，并使用 `PublishTrimmed` 将其发布。

要获得有关库的完整警告，必须执行“创建应用项目”这个额外的步骤，因为在 `dotnet build` 期间依赖项的实现通常不可用，而且引用程序集未包含足够的信息来确定它们是否与剪裁功能兼容。发布自包含应用可确保在库

的依赖项可用的情况下对库进行分析，如果你的库使用可能会让已剪裁的应用发生中断的依赖项中的任何代码，便会收到警报。

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net6.0</TargetFramework>
    <!-- Use a RID of your choice. -->
    <RuntimeIdentifier>linux-x64</RuntimeIdentifier>
    <PublishTrimmed>true</PublishTrimmed>
    <!-- Prevent warnings from unused code in dependencies -->
    <TrimmerDefaultAction>link</TrimmerDefaultAction>
  </PropertyGroup>

  <ItemGroup>
    <ProjectReference Include="path/to/MyLibrary.csproj" />
    <!-- Analyze the whole library, even if attributed with "IsTrimmable" -->
    <TrimmerRootAssembly Include="MyLibrary" />
  </ItemGroup>

</Project>
```

```
dotnet publish -c Release
```

- `TrimmerRootAssembly` 可确保对库中的每个部分进行分析。如果库有 `[AssemblyMetadata("IsTrimmable", "True")]`，则必须要有此项，否则剪裁功能会删除未使用的库且不会对其进行分析。
- `<TrimmerDefaultAction>link</TrimmerDefaultAction>` 可确保仅分析依赖项中已使用的部分。如果没有此选项，则会看到因依赖项有任何部分（包括库未使用的部分）未设置 `[AssemblyMetadata("IsTrimmable", "True")]` 而导致的警告。

你还可以让多个库遵循相同的模式，将它们作为 `ProjectReference` 和 `TrimmerRootAssembly` 项全部添加到同一个项目中，从而一次查看多个库的剪裁分析警告。但请注意，如果有任何根库在依赖项中使用不易于剪裁的 API，则会发出有关依赖项的警告。若要查看只与特定库有关的警告，应只引用该库。

> **NOTE**
> 分析结果取决于依赖项的实现细节。如果更新到依赖项的新版本，则在新版本添加了无法理解的反射模式时，可能会引入分析警告，即使未更改任何 API 也是如此。换句话说，将库与 `PublishTrimmed` 一起使用时，向库引入剪裁分析警告是一项中断性变更。

消除剪裁警告

以上步骤将生成有关代码的警告，在剪裁的应用中使用这些代码时，可能会导致问题。下面列举了你可能会遇到的最常见的警告类型，并提供相关的修复建议。

RequiresUnreferencedCode

```
using System.Diagnostics.CodeAnalysis;

public class MyLibrary
{
    public static void Method()
    {
        // warning IL2026 : MyLibrary.Method: Using method 'MyLibrary.DynamicBehavior' which has
        // 'RequiresUnreferencedCodeAttribute' can break functionality
        // when trimming application code.
        DynamicBehavior();
    }

    [RequiresUnreferencedCode("DynamicBehavior is incompatible with trimming.")]
    static void DynamicBehavior()
    {
    }
}
```

这意味着，库使用 RequiresUnreferencedCodeAttribute 调用被显式注释为"与剪裁功能不兼容"的方法。若要消除此警告，请考虑 `Method` 是否需要调用 `DynamicBehavior` 才能执行其作业。如果需要，也请使用 `RequiresUnreferencedCode` 注释调用方 `Method`；这样会弹出警告，使 `Method` 的调用方可以收到警告：

```
// Warn for calls to Method, but not for Method's call to DynamicBehavior.
[RequiresUnreferencedCode("Calls DynamicBehavior.")]
public static void Method()
{
    DynamicBehavior(); // OK. Doesn't warn now.
}
```

公共 API 中弹出该属性后（这样可以只为公共方法生成这些警告，如果真会发生的话），就算完成了。现在，调用库的应用将在调用这些公共 API 时收到警告，但不再生成类似 `IL2104: Assembly 'MyLibrary' produced trim warnings` 的警告。

DynamicallyAccessedMembers

```
using System.Diagnostics.CodeAnalysis;

public class MyLibrary
{
    static void UseMethods(Type type)
    {
        // warning IL2070: MyLibrary.UseMethods(Type): 'this' argument does not satisfy
        // 'DynamicallyAccessedMemberTypes.PublicMethods' in call to 'System.Type.GetMethods()'.
        // The parameter 't' of method 'MyLibrary.UseMethods(Type)' does not have matching annotations.
        foreach (var method in type.GetMethods())
        {
            // ...
        }
    }
}
```

这里，`UseMethods` 调用具有 DynamicallyAccessedMembersAttribute 要求的反射方法。该要求说明类型的公共方法可用。在这种情况下，可向 `UseMethods` 参数施加相同的要求来满足此要求。

```
static void UseMethods(
    // State the requirement in the UseMethods parameter.
    [DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicMethods)]
    Type type)
{
    // ...
}
```

此时，如果传入的值不满足 `PublicMethods` 要求，那么对 `UseMethods` 的任何调用都会生成警告。与使用 `RequiresUnreferencedCode` 一样，在公共 API 中弹出这类警告后，就算完成了。

下面是另一个示例，其中未知的 `Type` 流入注释的方法参数中，这次是通过字段：

```
static Type type;

static void UseMethodsHelper()
{
    // warning IL2077: MyLibrary.UseMethodsHelper(Type): 'type' argument does not satisfy
    // 'DynamicallyAccessedMemberTypes.PublicMethods' in call to 'MyLibrary.UseMethods(Type)'.
    // The field 'System.Type MyLibrary::type' does not have matching annotations.
    UseMethods(type);
}
```

同样，这里的问题在于，字段 `type` 被传入具有这些要求的参数。将 `DynamicallyAccessedMembers` 添加到字段中即可解决此问题。如果代码将不兼容的值赋给字段，将发出相关警告。有时，此过程将一直持续到对公共 API 添加注释为止，在其他时候，当具体类型流入具有这些要求的位置时，这个过程就会结束。例如：

```
[DynamicallyAccessedMembers(DynamicallyAccessedMembers.PublicMethods)]
static Type type;

static void InitializeTypeField()
{
    MyLibrary.type = typeof(System.Tuple);
}
```

在这种情况下，剪裁分析只会保留 `System.Tuple` 的公共方法，不会生成进一步的警告。

建议

通常，如果可能，应尽量避免反射。使用反射时，请限制其范围，以便只能从库的一小部分位置访问它。

- 应避免在静态构造函数等位置使用无法理解的模式，这将导致警告传播到类的所有成员。
- 应避免对虚拟方法或接口方法进行注释，这将要求所有重写都具有匹配的注释。
- 在某些情况下，你将能够通过代码机械地传播警告，不会遇到问题。有时，这会导致许多公共 API 使用 `RequiresUnreferencedCode` 注释，如果剪裁分析无法静态理解库的行为，则需要执行此操作。
- 在其他情况下，你可能会发现代码的使用模式不能用 `DynamicallyAccessedMembers` 属性来表示，即使它只使用反射来处理已知静态的类型也是如此。在这些情况下，可能需要重新组织某些代码，使其遵循可分析的模式。
- 有时，API 的现有设计使其几乎无法与剪裁功能兼容，你可能需要寻找其他方法来完成其工作。常见示例是基于反射的序列化程序。在这些情况下，请考虑采用其他技术（如源生成器）来生成更易于静态分析的代码。

消除有关不可分析的模式的警告

最好使用 `RequiresUnreferencedCode` 和 `DynamicallyAccessedMembers`（如可能）来表达代码意图，以消除警告。但在某些情况下，你可能想要为使用无法通过这些属性表示的模式的库或无需重构现有代码的库启用剪裁功能。本部分介绍了消除剪裁分析警告的一些高级方法。

> **WARNING**
> 如果未正确使用这些方法，则可能会导致代码中断。

UnconditionalSuppressMessage

如果代码意图无法用注释表示，但你知道该警告在运行时并不表示真正的问题，可使用 UnconditionalSuppressMessageAttribute 禁止显示警告。这与 `SuppressMessageAttribute` 非常类似，但它保留在 IL 中且在剪裁分析过程中被采用。

> **WARNING**
> 如果禁止显示警告，你应负责根据已知检查结果正确的不变量来保证代码的剪裁功能兼容性。请注意这些注释，因为如果它们不正确，或者代码的不变量发生更改，则最终可能会隐藏真正的问题。

例如：

```
class TypeCollection
{
    Type[] types;

    // Ensure that only types with ctors are stored in the array
    [DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicParameterlessConstructor)]
    public Type this[int i]
    {
        // warning IL2063: TypeCollection.Item.get: Value returned from method 'TypeCollection.Item.get'
        // can not be statically determined and may not meet 'DynamicallyAccessedMembersAttribute'
requirements.
        get => types[i];
        set => types[i] = value;
    }
}

class TypeCreator
{
    TypeCollection types;

    public void CreateType(int i)
    {
        types[i] = typeof(TypeWithConstructor);
        Activator.CreateInstance(types[i]); // No warning!
    }
}

class TypeWithConstructor
{
}
```

此处已对索引器属性进行注释，使返回的 `Type` 满足 `CreateInstance` 的要求。这已经确保保留了 `TypeWithConstructor` 构造函数，而且对 `CreateInstance` 的调用不会发出警告。此外，索引器资源库注释也确保了存储在 `Type[]` 中的任何类型都有一个构造函数。但分析功能无法看到这一点，并且仍会生成 Getter 警告，因为它不知道返回的类型已保留其构造函数。

如果确定满足要求，可将 `UnconditionalSuppressMessage` 添加到 Getter 来禁止显示此警告：

```
[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicParameterlessConstructor)]
public Type this[int i]
{
    [UnconditionalSuppressMessage("ReflectionAnalysis", "IL2063",
        Justification = "The list only contains types stored through the annotated setter.")]
    get => types[i];
    set => types[i] = value;
}
```

DynamicDependency

此属性可用于指示某个成员动态依赖于其他成员。这会导致每次保留具有该属性的成员时都会保留被引用的成员，但警告不会自行消除。与向剪裁分析功能告知代码反射行为的其他属性不同，`DynamicDependency` 仅保留附加成员。可将此属性与 `UnconditionalSuppressMessageAttribute` 一起使用，以修复某些分析警告。

> **WARNING**
> 仅当其他方法都不可行时，才使用 `DynamicDependencyAttribute` 作为最终手段。最好使用 `RequiresUnreferencedCodeAttribute` 或 `DynamicallyAccessedMembersAttribute` 来表达代码的反射行为。

```
[DynamicDependency("Helper", "MyType", "MyAssembly")]
static void RunHelper()
{
    var helper = Assembly.Load("MyAssembly").GetType("MyType").GetMethod("Helper");
    helper.Invoke(null, null);
}
```

如果没有 `DynamicDependency`，剪裁可能会从 `MyAssembly` 中删除 `Helper`，或者完全删除 `MyAssembly`（如果没有在其他位置引用它），这会生成警告，指出在运行时可能发生了故障。该属性确保 `Helper` 会保留。

该属性指定要通过 `string` 或 `DynamicallyAccessedMemberTypes` 保留的成员。类型和程序集要么隐含在属性上下文中，要么在属性中显式指定（按照分别表示类型和程序集名称的 `Type` 或 `string`）。

类型和成员字符串使用 C# 文档注释 ID 字符串格式的变体，不带成员前缀。成员字符串不应包含声明类型的名称，可以省略参数以保留指定名称的所有成员。下面是一些格式示例：

```
[DynamicDependency("Method()")]
[DynamicDependency("Method(System,Boolean,System.String)")]
[DynamicDependency("MethodOnDifferentType()", typeof(ContainingType))]
[DynamicDependency("MemberName")]
[DynamicDependency("MemberOnUnreferencedAssembly", "ContainingType", "UnreferencedAssembly")]
[DynamicDependency("MemberName", "Namespace.ContainingType.NestedType", "Assembly")]
// generics
[DynamicDependency("GenericMethodName``1")]
[DynamicDependency("GenericMethod``2(``0,``1)")]
[DynamicDependency("MethodWithGenericParameterTypes(System.Collections.Generic.List{System.String})")]
[DynamicDependency("MethodOnGenericType(`0)", "GenericType`1", "UnreferencedAssembly")]
[DynamicDependency("MethodOnGenericType(`0)", typeof(GenericType<>))]
```

当方法包含即使借助 `DynamicallyAccessedMembersAttribute` 也无法分析的反射模式时，适合使用此属性。

IL2001：描述符文件尝试在没有字段的类型上保留字段

2021/11/16 •

原因

XML 描述符文件正在尝试在没有字段的类型上保留字段。

规则说明

描述符文件用于指示 IL 剪裁器始终保留程序集中的某些成员，不管剪裁器是否可以找到对这些成员的引用。但是，尝试保留找不到的成员将触发警告。

示例

```
<linker>
  <assembly fullname="test">
    <type fullname="TestType" preserve="fields" />
  </assembly>
</linker>
```

```
// IL2001: Type 'TestType' has no fields to preserve
class TestType
{
    void OnlyMethod() {}
}
```

IL2002：描述符文件尝试在没有方法的类型上保留方法

2021/11/16 •

原因

XML 描述符文件正在尝试在没有方法的类型上保留方法。

规则说明

描述符文件用于指示 IL 剪裁器始终保留程序集中的某些成员，不管剪裁器是否可以找到对这些成员的引用。但是，尝试保留找不到的成员将触发警告。

示例

```
<linker>
  <assembly fullname="test">
    <type fullname="TestType" preserve="methods" />
  </assembly>
</linker>
```

```
// IL2002: Type 'TestType' has no methods to preserve
struct TestType
{
    public int Number;
}
```

IL2003：无法解析“PreserveDependency”属性中指定的依赖项程序集

2021/11/16 •

原因

无法解析 `PreserveDependencyAttribute` 中指定的程序集。

规则说明

剪裁器会将缓存与它所看到的程序集保留在一起。如果在此缓存中找不到 `PreserveDependencyAttribute` 中指定的程序集，则剪裁器无法查找要保留的成员。

示例

```
// IL2003: Could not resolve dependency assembly 'NonExistentAssembly' specified in a 'PreserveDependency'
attribute
[PreserveDependency("MyMethod", "MyType", "NonExistentAssembly")]
void TestMethod()
{
}
```

IL2004：无法解析“PreserveDependency”属性中指定的依赖项类型

2021/11/16 •

原因

无法解析 `PreserveDependencyAttribute` 中指定的类型。

规则说明

剪裁器会将缓存与它所看到的程序集保留在一起。如果在此缓存中的程序集内找不到 `PreserveDependencyAttribute` 中指定的类型，则剪裁器无法查找要保留的成员。

示例

```
// IL2004: Could not resolve dependency type 'NonExistentType' specified in a 'PreserveDependency' attribute
[PreserveDependency("MyMethod", "NonExistentType", "MyAssembly")]
void TestMethod()
{
}
```

IL2005：无法解析“PreserveDependency”特性中指定的依赖项成员

2021/11/16 •

原因

无法解析 `PreserveDependencyAttribute` 中指定的类型的成员。

示例

```
// IL2005: Could not resolve dependency member 'NonExistentMethod' declared on type 'MyType' specified in a
'PreserveDependency' attribute
[PreserveDependency("NonExistentMethod", "MyType", "MyAssembly")]
void TestMethod()
{
}
```

IL2007：无法解析描述符文件中指定的程序集

2021/11/16 •

原因

无法解析描述符文件中指定的某个程序集。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在剪裁器看到的任何程序集中都找不到描述符文件中按完整名称指定的程序集。

示例

```
<!-- IL2007: Could not resolve assembly 'NonExistentAssembly' -->
<linker>
  <assembly fullname="NonExistentAssembly" />
</linker>
```

IL2008：无法解析描述符文件中指定的类型

2021/11/16 •

原因

无法解析描述符文件中指定的某个类型。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在与传递给 `type` 元素父级的 `fullname` 参数匹配的程序集中，找不到描述符文件中指定的某个类型。

示例

```
<!-- IL2008: Could not resolve type 'NonExistentType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="NonExistentType" />
  </assembly>
</linker>
```

IL2009：无法解析描述符文件中指定的方法

2021/11/16 •

原因

无法解析描述符文件中的类型上指定的某个方法。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在与传递给 `method` 元素父级的 `fullname` 参数匹配的类型中，找不到描述符文件中指定的某个方法。

示例

```
<!-- IL2009: Could not find method 'NonExistentMethod' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="NonExistentMethod" />
    </type>
  </assembly>
</linker>
```

IL2010：方法替换的值无效

2021/11/16 •

原因

替换文件中用来替换方法主体的值不表示内置类型的值或与方法的返回类型不匹配。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 `throw` 语句或 return constant 语句。

传递给 `method` 元素的 `value` 参数的值无法由剪裁器转换为与指定方法的返回类型匹配的类型。

示例

```
<!-- IL2010: Invalid value for 'MyType.MyMethodReturningInt()' stub -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethodReturningInt" body="stub" value="NonNumber" />
    </type>
  </assembly>
</linker>
```

IL2011：主体修改操作未知

2021/11/16 •

原因

传递给替换文件中 `method` 元素的 `body` 参数的操作值无效。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 `throw` 语句或 return constant 语句。

传递给 `method` 元素的 `body` 参数的值无效。此参数支持的选项只有 `remove` 和 `stub`。

示例

```
<!-- IL2011: Unknown body modification 'nonaction' for 'MyType.MyMethod()' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod" body="nonaction" value="NonNumber" />
    </type>
  </assembly>
</linker>
```

IL2012：在替换文件中的类型上找不到字段

2021/11/16 •

原因

在替换文件中找不到指定用于替换的字段。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

在与传递给 `field` 元素父级的 `fullname` 参数匹配的类型中，找不到替换文件中指定的某个字段。

示例

```
<!-- IL2012: Could not find field 'NonExistentField' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <field name="NonExistentField" />
    </type>
  </assembly>
</linker>
```

IL2013：替换字段必须为静态或常量

2021/11/16 •

原因

替换文件中指定用于替换的字段为非静态或常量。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

剪裁器不能替换非静态或常量字段。

示例

```
<!-- IL2013: Substituted field 'MyType.InstanceField' needs to be static field -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <field name="InstanceField" value="5" />
    </type>
  </assembly>
</linker>
```

IL2014：字段替换的值缺失

2021/11/16 •

原因

在替换文件中为替换指定了一个字段，但没有给出要替换的值。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

替换文件中指定的 `field` 元素未指定必需的 `value` 参数。

示例

```
<!-- IL2014: Missing 'value' attribute for field 'MyType.MyField' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <field name="MyField" />
    </type>
  </assembly>
</linker>
```

IL2015：字段替换的值无效

2021/11/16 •

原因

替换文件中用来替换字段值的值不表示内置类型的值或与字段的类型不匹配。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

传递给 `field` 元素的 `value` 参数的值无法由剪裁器转换为与指定字段的返回类型匹配的类型。

示例

```
<!-- IL2015: Invalid value 'NonNumber' for 'MyType.IntField' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <field name="IntField" value="NonNumber" />
    </type>
  </assembly>
</linker>
```

IL2016：找不到有关类型的事件

2021/11/16 •

原因

找不到有关类型的事件。

规则说明

在与传递给 `event` 元素父级的 `fullname` 参数匹配的类型中，找不到剪裁器 XML 文件中指定的某个事件。

示例

```
<!-- IL2016: Could not find event 'NonExistentEvent' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <event name="NonExistentEvent" />
    </type>
  </assembly>
</linker>
```

IL2017：在类型上找不到属性

2021/11/16 •

原因

在类型上找不到属性。

规则说明

在与传递给 `property` 元素父级的 `fullname` 参数匹配的类型中，找不到剪裁器 XML 文件中指定的某个属性。

示例

```
<!-- IL2017: Could not find property 'NonExistentProperty' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <property name="NonExistentProperty" />
    </type>
  </assembly>
</linker>
```

IL2018：在描述符文件中的类型上找不到属性的 get 访问器

2021/11/16 •

原因

找不到描述符文件中指定的 `get` 访问器。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在与传递给 `property` 元素的 `signature` 参数匹配的属性中，找不到描述符文件中指定的 `get` 访问器。

示例

```
<!-- IL2018: Could not find the get accessor of property 'SetOnlyProperty' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <property signature="System.Boolean SetOnlyProperty" accessors="get" />
    </type>
  </assembly>
</linker>
```

IL2019：在描述符文件中的类型上找不到属性的 set 访问器

2021/11/16 •

原因

找不到描述符文件中指定的 `set` 访问器。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在与传递给 `property` 元素的 `signature` 参数匹配的属性中，找不到描述符文件中指定的 `set` 访问器。

示例

```
<!-- IL2019: Could not find the set accessor of property 'GetOnlyProperty' on type 'MyType' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <property signature="System.Boolean GetOnlyProperty" accessors="set" />
    </type>
  </assembly>
</linker>
```

IL2022：找不到自定义属性注释文件中指定的自定义属性的匹配构造函数

2021/11/16 •

原因

找不到自定义属性注释文件中指定的自定义属性的构造函数。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性；将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

传递给 `attribute` 元素的 `argument` 子元素的值无法由剪裁器转换为与属性的构造函数参数类型匹配的类型。

示例

```
<!-- IL2022: Could not find matching constructor for custom attribute 'attribute-type' arguments -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <attribute fullname="AttributeWithNoParametersAttribute">
        <argument>ExtraArgumentValue</argument>
      </attribute>
    </type>
  </assembly>
</linker>
```

IL2023：在自定义属性注释文件中为某个方法指定了多个 `return` 子元素

2021/11/16 •

原因

为某个方法指定了多个 `return` 元素。将属性放置在方法的返回参数上时，只能有一个 `return` 元素。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性，将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

为某个 `method` 元素指定了多个 `return` 元素。剪裁器只允许对给定方法的返回类型使用一个属性注释。

示例

```
<!-- IL2023: There is more than one 'return' child element specified for method 'method' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod">
        <return>
          <attribute fullname="FirstAttribute"/>
        </return>
        <return>
          <attribute fullname="SecondAttribute"/>
        </return>
      </method>
    </type>
  </assembly>
</linker>
```

IL2024：在自定义属性注释文件中为同一方法参数指定了多个值

2021/11/16 •

原因

为某个方法参数指定了多个值元素。只能为每个方法参数指定一个值。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性，将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

给定 `method` 中有多个具有相同 `name` 值的 `parameter` 元素。`parameter` 上的所有属性都应放入单个元素中。

示例

```
<!-- IL2024: More than one value specified for parameter 'parameter' of method 'method' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod">
        <parameter name="methodParameter">
          <attribute fullname="FirstAttribute"/>
        </parameter>
        <parameter name="methodParameter">
          <attribute fullname="SecondAttribute"/>
        </parameter>
      </method>
    </type>
  </assembly>
</linker>
```

IL2025：重复保留描述符文件中的成员

2021/11/16 •

原因

类型上的成员标记为在描述符文件中保留多次。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

此文件中的成员应仅出现一次。

示例

```
<!-- IL2025: Duplicate preserve of 'method' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod"/>
      <method name="MyMethod"/>
    </type>
  </assembly>
</linker>
```

IL2026：具有 RequiresUnreferencedCode 属性的成员在剪裁时可能会中断

2021/11/16 •

原因

调用(或通过反射访问)使用 RequiresUnreferencedCodeAttribute 进行批注的成员。

例如：

```
[RequiresUnreferencedCode("Use 'MethodFriendlyToTrimming' instead", Url="http://help/unreferencedcode")]
void MethodWithUnreferencedCodeUsage()
{
}

void TestMethod()
{
    // IL2026: Using method 'MethodWithUnreferencedCodeUsage' which has 'RequiresUnreferencedCodeAttribute'
    // can break functionality when trimming application code. Use 'MethodFriendlyToTrimming' instead.
http://help/unreferencedcode
    MethodWithUnreferencedCodeUsage();
}
```

规则说明

RequiresUnreferencedCodeAttribute 表示成员引用了可被剪裁器删除的代码。

常见示例包括：

- Load(String) 被标记为 `RequiresUnreferencedCode`，因为被加载的程序集可以访问已被剪裁掉的成员。剪裁器从框架中删除除应用程序直接使用的成员之外的所有成员，因此在运行时加载新程序集很可能会尝试访问丢失的成员。
- XmlSerializer 被标记为 `RequiresUnreferencedCode`，因为 `XmlSerializer` 使用复杂反射来扫描输入类型。剪裁器无法跟踪反射，因此输入类型以可传递的方式使用的成员可能会被剪裁掉。

IL2027：已知剪裁器特性在单个成员上使用了多次

2021/11/16 •

原因

剪裁器在单个成员上找到了支持剪裁器的同一特性的多个实例。

IL2028：已知剪裁器属性不具有所需的参数数目

2021/11/16 •

原因

剪裁器找到了已知属性的实例，该实例缺少所需的构造函数参数或超过接受的参数。如果自定义程序集定义的自定义属性的全名与已知的剪裁器属性冲突，则可能会发生这种情况，因为剪裁器通过匹配命名空间和类型名称来识别自定义属性。

IL2029：自定义属性注释文件中的属性元素没有必需的参数 `fullname` 或为空

2021/11/16 •

原因

自定义属性注释文件中的属性元素没有必需的参数 `fullname`，或者其值为空字符串。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性，将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

所有 `attribute` 元素都必须具有必需的 `fullname` 参数，并且其值不能为空字符串。

示例

```
<!-- IL2029: 'attribute' element does not contain required attribute 'fullname' or it's empty -->
<linker>
  <assembly fullname="MyAssembly">
    <attribute/>
  </assembly>
</linker>
```

IL2030：无法解析自定义属性注释文件中指定的程序集

2021/11/16 •

原因

无法从自定义属性注释文件中某个属性元素的 `assembly` 参数解析程序集。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性，将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

`attribute` 元素中 `assembly` 参数的值与剪裁器看到的任何程序集都不匹配。

示例

```
<!-- IL2030: Could not resolve assembly 'NonExistentAssembly' for attribute 'MyAttribute' -->
<linker>
  <assembly fullname="MyAssembly">
    <attribute fullname="MyAttribute" assembly="NonExistentAssembly"/>
  </assembly>
</linker>
```

IL2031：无法解析自定义属性注释文件中指定的自定义属性

2021/11/16 •

原因

无法从自定义属性注释文件的属性元素的 `fullname` 参数中指定的类型名称解析自定义属性。

规则说明

自定义属性注释文件用于指示剪裁器像指定的项具有给定属性那样运行。属性注释只能用于添加对剪裁器行为有影响的属性，将忽略所有其他属性。通过属性注释添加的属性仅影响剪裁器行为，并且永远不会添加到输出程序集。

在与传递给 `attribute` 元素父级的 `fullname` 参数匹配的程序集中，找不到自定义属性注释文件中指定的某个属性。

示例

```
<!-- IL2031: Attribute type 'NonExistentTypeAttribute' could not be found -->
<linker>
  <assembly fullname="MyAssembly">
    <attribute fullname="NonExistentTypeAttribute"/>
  </assembly>
</linker>
```

IL2032：将无法识别的值传递给了“System.Activator.CreateInstance”方法的参数“parameter”

2021/11/16 •

原因

无法静态分析传递给 CreateInstance 方法的程序集或类型名称的值。剪裁器无法保证目标类型的可用性。

示例

```
void TestMethod(string assemblyName, string typeName)
{
    // IL2032 Trim analysis: Unrecognized value passed to the parameter 'typeName' of method
'System.Activator.CreateInstance(string, string)'. It's not possible to guarantee the availability of the
target type.
    Activator.CreateInstance("MyAssembly", typeName);

    // IL2032 Trim analysis: Unrecognized value passed to the parameter 'assemblyName' of method
'System.Activator.CreateInstance(string, string)'. It's not possible to guarantee the availability of the
target type.
    Activator.CreateInstance(assemblyName, "MyType");
}
```

IL2033：“PreserveDependencyAttribute”已弃用

2021/11/16 •

原因

`PreserveDependencyAttribute` 是剪裁器使用的内部属性，不受支持。请改用 DynamicDependencyAttribute。

示例

```
// IL2033: 'PreserveDependencyAttribute' is deprecated. Use 'DynamicDependencyAttribute' instead.
[PreserveDependency("OtherMethod")]
public void TestMethod()
{
}
```

IL2034：无法分析“DynamicDependencyAttribute”

2021/11/16 •

原因

应用程序包含对 DynamicDependencyAttribute 的无效使用。确保使用的是正式支持的构造函数之一。

IL2035：无法解析“DynamicDependencyAttribute”中的程序集

2021/11/16 •

原因

无法解析传递给 DynamicDependencyAttribute **的** `assemblyName` **参数的**值。

示例

```
// IL2035: Unresolved assembly 'NonExistentAssembly' in 'DynamicDependencyAttribute'
[DynamicDependency("Method", "Type", "NonExistentAssembly")]
public void TestMethod()
{
}
```

IL2036：无法解析“DynamicDependencyAttribute”中的类型

2021/11/16 •

原因

无法解析传递给 DynamicDependencyAttribute **的** `typeName` **参数的**值。

示例

```
// IL2036: Unresolved type 'NonExistentType' in 'DynamicDependencyAttribute'
[DynamicDependency("Method", "NonExistentType", "MyAssembly")]
public void TestMethod()
{
}
```

IL2037：无法解析“DynamicDependencyAttribute”中的成员

2021/11/16 •

原因

传递给 DynamicDependencyAttribute 的成员签名参数的值无法解析为任何成员。确保传递的值引用现有成员并使用正确的 ID 字符串格式。

示例

```
// IL2037: Unresolved type 'NonExistentType' in 'DynamicDependencyAttribute'
[DynamicDependency("Method", "NonExistentType", "MyAssembly")]
public void TestMethod()
{
}
```

IL2038：替换文件中的 resource 元素上缺少 `name` 参数

2021/11/16 •

原因

替换文件中的 `resource` 元素未指定必需的 `name` 参数。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

替换文件中的所有 `resource` 元素必须具有必需的 `name` 参数，该参数指定要删除的资源。

示例

```
<!-- IL2038: Missing 'name' attribute for resource. -->
<linker>
  <assembly fullname="MyAssembly">
    <resource />
  </assembly>
</linker>
```

IL2039：替换文件中 resource 元素上的 `action` 值无效

2021/11/16 •

原因

传递给替换文件中 `resource` 元素的 `action` 参数的值无效。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

传递给 `resource` 元素的 `action` 参数的值无效。此参数支持的唯一一个值为 `remove`。

示例

```
<!-- IL2039: Invalid value 'NonExistentAction' for attribute 'action' for resource 'MyResource'. -->
<linker>
  <assembly fullname="MyAssembly">
    <resource name="MyResource" action="NonExistentAction"/>
  </assembly>
</linker>
```

IL2040：找不到替换文件中指定的嵌入资源

2021/11/16 •

原因

在指定的程序集中找不到名称与 `name` 参数中使用的值匹配的嵌入资源。

规则说明

替换文件用于指示剪裁器将特定的方法主体替换为 throw 语句或 return constant 语句。

在指定的程序集中找不到替换文件中的资源名称。要删除的资源的名称必须与程序集中的嵌入资源的名称匹配。

示例

```
<!-- IL2040: Could not find embedded resource 'NonExistentResource' to remove in assembly 'MyAssembly'. -->
<linker>
  <assembly fullname="MyAssembly">
    <resource name="NonExistentResource" action="remove"/>
  </assembly>
</linker>
```

IL2041：方法上不允许“DynamicallyAccessedMembersAttribute”

2021/11/16 •

原因

DynamicallyAccessedMembersAttribute 已直接放置在方法上。只允许对 Type 上的实例方法采用这种做法。通常应将此属性放置在方法的返回值上或其中一个参数上。

示例

```
// IL2041: The 'DynamicallyAccessedMembersAttribute' is not allowed on methods. It is allowed on method
return value or method parameters though.
[DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]

[return: DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]
public Type GetInterestingType()
{
}
```

IL2042：找不到用于在属性上传播“DynamicallyAccessedMembersAttribute”注释的唯一支持字段

2021/11/16 •

原因

剪裁器无法确定使用 DynamicallyAccessedMembersAttribute 进行注释的属性的支持字段。

示例

```
// IL2042: Could not find a unique backing field for property 'MyProperty' to propagate
'DynamicallyAccessedMembersAttribute'
[DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]
public Type MyProperty
{
  get { return GetTheValue(); }
  set { }
}

// To fix this annotate the accessors manually:
public Type MyProperty
{
  [return: DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]
  get { return GetTheValue(); }

  [param: DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]
  set { }
}
```

IL2043：属性上的“DynamicallyAccessedMembersAttribute”与其访问器方法上的相同特性有冲突

2021/11/16 •

原因

将 DynamicallyAccessedMembersAttribute 从已批注的属性传播到其访问器方法时，剪裁器发现访问器已有此类特性。只会使用现有特性。

示例

```
// IL2043: 'DynamicallyAccessedMembersAttribute' on property 'MyProperty' conflicts with the same attribute
on its accessor 'get_MyProperty'.
[DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicMethods)]
public Type MyProperty
{
  [return: DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicFields)]
  get { return GetTheValue(); }
}
```

IL2044：在描述符文件中指定的命名空间中找不到任何类型

2021/11/16 •

原因

描述符文件指定了一个命名空间，该命名空间中没有任何类型。

规则说明

描述符文件用于指示剪裁器始终保留程序集中的某些项，不管剪裁器是否可以找到对这些项的任何引用。

在与传递给 `namespace` 元素父级的 `fullname` 参数匹配的程序集中，找不到描述符文件中指定的某个命名空间。

示例

```
<!-- IL2044: Could not find any type in namespace 'NonExistentNamespace' -->
<linker>
  <assembly fullname="MyAssembly">
    <namespace fullname="NonExistentNamespace" />
  </assembly>
</linker>
```

IL2045：在代码中引用了自定义属性，但已指示剪裁器删除其所有实例

2021/11/16 •

原因

已指示剪裁器删除自定义属性的所有实例，但在分析过程中保留其类型。这可能会导致正在使用该自定义属性类型的代码中断。

示例

```
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyAttribute">
      <attribute internal="RemoveAttributeInstances"/>
    </type>
  </assembly>
</linker>
```

```
// This attribute instance will be removed
[MyAttribute]
class MyType
{
}

public void TestMethod()
{
  // IL2045 for 'MyAttribute' reference
  typeof(MyType).GetCustomAttributes(typeof(MyAttribute), false);
}
```

IL2046：所有接口实现和方法重写都必须包含与接口匹配的注释或重写虚拟方法“RequiresUnreferencedCodeAttribute”注释

2021/11/16 •

原因

接口及其实现或虚拟方法及其重写之间的 RequiresUnreferencedCodeAttribute 注释不匹配。

示例

基成员具有属性，但派生成员没有属性。

```
public class Base
{
  [RequiresUnreferencedCode("Message")]
  public virtual void TestMethod() {}
}

public class Derived : Base
{
  // IL2046: Base member 'Base.TestMethod' with 'RequiresUnreferencedCodeAttribute' has a derived member
'Derived.TestMethod()' without 'RequiresUnreferencedCodeAttribute'. For all interfaces and overrides the
implementation attribute must match the definition attribute.
  public override void TestMethod() {}
}
```

派生成员具有属性，但重写的基成员没有属性。

```
public class Base
{
  public virtual void TestMethod() {}
}

public class Derived : Base
{
  // IL2046: Member 'Derived.TestMethod()' with 'RequiresUnreferencedCodeAttribute' overrides base member
'Base.TestMethod()' without 'RequiresUnreferencedCodeAttribute'. For all interfaces and overrides the
implementation attribute must match the definition attribute.
  [RequireUnreferencedCode("Message")]
  public override void TestMethod() {}
}
```

接口成员具有属性，但其实现没有属性。

```
interface IRUC
{
  [RequiresUnreferencedCode("Message")]
  void TestMethod();
}

class Implementation : IRUC
{
  // IL2046: Interface member 'IRUC.TestMethod()' with 'RequiresUnreferencedCodeAttribute' has an
implementation member 'Implementation.TestMethod()' without 'RequiresUnreferencedCodeAttribute'. For all
interfaces and overrides the implementation attribute must match the definition attribute.
  public void TestMethod () { }
}
```

实现成员具有属性，但它实现的接口没有属性。

```
interface IRUC
{
  void TestMethod();
}

class Implementation : IRUC
{
  [RequiresUnreferencedCode("Message")]
  // IL2046: Member 'Implementation.TestMethod()' with 'RequiresUnreferencedCodeAttribute' implements
interface member 'IRUC.TestMethod()' without 'RequiresUnreferencedCodeAttribute'. For all interfaces and
overrides the implementation attribute must match the definition attribute.
  public void TestMethod () { }
}
```

IL2048：正在成员上使用内部剪裁器属性“RemoveAttributeInstances”

2021/11/16 •

原因

正在成员上使用内部剪裁器属性 `RemoveAttributeInstances`，但只能在类型上使用该属性。

示例

```
<!-- IL2048: Internal attribute 'RemoveAttributeInstances' can only be used on a type, but is being used on
'MyMethod' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod">
        <attribute internal="RemoveAttributeInstances" />
      </method>
    </type>
  </assembly>
</linker>
```

IL2049：无法识别的内部特性

2021/11/16 •

原因

剪裁器不支持自定义特性批注文件中指定的内部特性名称。

示例

```
<!-- IL2049: Unrecognized internal attribute 'InvalidInternalAttributeName' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <method name="MyMethod">
        <attribute internal="InvalidInternalAttributeName" />
      </method>
    </type>
  </assembly>
</linker>
```

IL2050：无法保证 COM 互操作的正确性

2021/11/16 •

原因

剪裁器找到了一个 p/invoke 方法，该方法使用 COM 封送声明了一个参数。剪裁后无法保证 COM 互操作的正确性。

示例

```
// IL2050: M1(): P/invoke method 'M2(C)' declares a parameter with COM marshalling. Correctness of COM
interop cannot be guaranteed after trimming. Interfaces and interface members might be removed.
static void M1 ()
{
  M2 (null);
}

[DllImport ("Foo")]
static extern void M2 (C autoLayout);

[StructLayout (LayoutKind.Auto)]
public class C
{
}
```

IL2051：属性元素在自定义属性注释文件中没有必需的参数 `name`

2021/11/16 •

原因

自定义属性注释文件中的属性元素未指定必需的参数 `name`。

示例

```
<!-- IL2051: Property element does not contain attribute 'name' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <attribute fullname="MyAttribute">
        <property>UnspecifiedPropertyName</property>
      </attribute>
    </type>
  </assembly>
</linker>
```

IL2052：找不到自定义属性注释文件中指定的属性

2021/11/16 •

原因

在自定义属性注释文件中，找不到与 `property` 元素中指定的 `name` 参数的值匹配的属性。

示例

```
<!-- IL2052: Property 'NonExistentPropertyName' could not be found -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <attribute fullname="MyAttribute">
        <property name="NonExistentPropertyName">SomeValue</property>
      </attribute>
    </type>
  </assembly>
</linker>
```

IL2053：自定义属性注释文件中的属性元素使用的值无效

2021/11/16 •

原因

自定义属性注释文件中的 `property` 元素中使用的值与特性的属性类型不匹配。

示例

```
<!-- IL2053: Invalid value 'StringValue' for property 'IntProperty' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <attribute fullname="MyAttribute">
        <property name="IntProperty">StringValue</property>
      </attribute>
    </type>
  </assembly>
</linker>
```

IL2054：自定义属性注释文件中的参数值无效

2021/11/16 •

原因

自定义属性注释文件中的 `argument` 元素中使用的值与属性的构造函数参数的类型不匹配。

示例

```
<!-- IL2054: Invalid argument value 'NonExistentEnumValue' for parameter of type 'MyEnumType' of attribute
'AttributeWithEnumParameterAttribute' -->
<linker>
  <assembly fullname="MyAssembly">
    <type fullname="MyType">
      <attribute fullname="AttributeWithEnumParameterAttribute">
        <argument>NonExistentEnumValue</argument>
      </attribute>
    </type>
  </assembly>
</linker>
```

IL2055：剪裁器无法静态分析对“System.Type.MakeGenericType”的调用

2021/11/16 •

原因

剪裁器无法静态分析对 Type.MakeGenericType(Type[]) 的调用。

规则说明

这可以是无法静态确定调用 MakeGenericType(Type[]) 的类型，或者无法静态确定用于泛型参数的类型参数。如果开放式泛型类型的任何泛型参数上都有 DynamicallyAccessedMembersAttribute 注释，则剪裁器当前无法验证调用方法是否满足要求。

示例

```
class Lazy<[DynamicallyAccessedMembers(DynamicallyAccessedMemberType.PublicParameterlessConstructor)] T>
{
    // ...
}

void TestMethod(Type unknownType)
{
    // IL2055 Trim analysis: Call to `System.Type.MakeGenericType(Type[])` can not be statically analyzed.
It's not possible to guarantee the availability of requirements of the generic type.
    typeof(Lazy<>).MakeGenericType(new Type[] { typeof(TestType) });

    // IL2055 Trim analysis: Call to `System.Type.MakeGenericType(Type[])` can not be statically analyzed.
It's not possible to guarantee the availability of requirements of the generic type.
    unknownType.MakeGenericType(new Type[] { typeof(TestType) });
}
```

IL2056：属性上的“System.Diagnostics.CodeAnalysis.DynamicallyAccessedMembersAttr释与其支持字段上的相同属性有冲突

2021/11/16 •

原因

使用 DynamicallyAccessedMembersAttribute 进行注释的属性在其支持字段上也有该属性。

规则说明

将 DynamicallyAccessedMembersAttribute 从某个属性传播到其支持字段时，剪裁器发现它的支持字段已注释。只会使用现有属性。

剪裁器只会将注释传播到编译器生成的支持字段，因此，仅当使用 CompilerGeneratedAttribute 显式注释了支持字段时，才会显示此警告。

示例

```
[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicConstructors)]
[CompilerGenerated]
Type backingField;

[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicConstructors)]
Type PropertyWithAnnotatedBackingField
{
  get { return backingField; }
  set { backingField = value; }
}
```

IL2057：将无法识别的值传递给了“System.Type.GetType(String)”的 `typeName` 参数

2021/11/16 •

原因

将一个无法识别的值传递给了 Type.GetType(String) 的 `typeName` 参数。

规则说明

如果类型名称(已传递给 GetType(String) 的 `typeName` 参数)是静态已知的，则剪裁器可以确保保留该名称，应用程序代码在剪裁后将继续工作。如果类型未知并且剪裁器无法找到在其他任何位置使用的类型，则剪裁器可能最终会从应用程序中删除该类型，因此可能会中断应用程序。

示例

```
void TestMethod()
{
    string typeName = ReadName();

    // IL2057 Trim analysis: Unrecognized value passed to the parameter 'typeName' of method
'System.Type.GetType(String typeName)'
    Type.GetType(typeName);
}
```

IL2058：无法静态分析传递给“Assembly.CreateInstance”的参数

2021/11/16 •

原因

在已分析的代码中找到了对 CreateInstance 的调用。

规则说明

剪裁器不会分析程序集实例，因此不知道调用了哪个程序集 CreateInstance。

示例

```
void TestMethod()
{
    // IL2058 Trim analysis: Parameters passed to method 'Assembly.CreateInstance(string)' cannot be
analyzed. Consider using methods 'System.Type.GetType' and `System.Activator.CreateInstance` instead.
    AssemblyLoadContext.Default.Assemblies.First(a => a.Name == "MyAssembly").CreateInstance("MyType");

    // This can be replaced by
    Activator.CreateInstance(Type.GetType("MyType, MyAssembly"));
}
```

如何解决

剪裁器支持 Type.GetType(String)。可将结果传递给 CreateInstance 以创建类型实例。

IL2059：将无法识别的值传递给了“System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstru `type` 参数

2021/11/16 •

原因

将一个无法识别的值传递给了 RuntimeHelpers.RunClassConstructor(RuntimeTypeHandle) 的 `type` 参数。

规则说明

如果传递给 RunClassConstructor(RuntimeTypeHandle) 的类型不是静态已知的，则剪裁器无法保证目标静态构造函数的可用性。

示例

```
void TestMethod(Type type)
{
    // IL2059 Trim analysis: Unrecognized value passed to the parameter 'type' of method
'System.Runtime.CompilerServices.RuntimeHelpers.RunClassConstructor(RuntimeTypeHandle type)'.
    // It's not possible to guarantee the availability of the target static constructor.
    RuntimeHelpers.RunClassConstructor(type.TypeHandle);
}
```

IL2060：剪裁器无法静态分析对“System.Reflection.MethodInfo.MakeGenericMethod”的调用

2021/11/16 •

原因

剪裁器无法静态分析对 MethodInfo.MakeGenericMethod(Type[]) 的调用。

规则说明

这可以是无法静态确定调用 MakeGenericMethod(Type[]) 的方法，或者无法静态确定用于泛型参数的类型参数。 如果开放式泛型方法的任何泛型参数上都有 DynamicallyAccessedMembersAttribute 注释，则剪裁器当前无法验证调用方法是否满足要求。

示例

```
class Test
{
  public static void
TestGenericMethod<[DynamicallyAccessedMembers(DynamicallyAccessedMemberTypes.PublicProperties)] T>()
  {
  }

  void TestMethod(Type unknownType)
  {
    // IL2060 Trim analysis: Call to 'System.Reflection.MethodInfo.MakeGenericMethod' can not be statically
analyzed. It's not possible to guarantee the availability of requirements of the generic method
    typeof(Test).GetMethod("TestGenericMethod").MakeGenericMethod(new Type[] { typeof(TestType) });

    // IL2060 Trim analysis: Call to 'System.Reflection.MethodInfo.MakeGenericMethod' can not be statically
analyzed. It's not possible to guarantee the availability of requirements of the generic method
    unknownMethod.MakeGenericMethod(new Type[] { typeof(TestType) });
  }
}
```

运行时包存储区

2021/11/16 •

自 .NET Core 2.0 起，可以根据目标环境中已知的一组包来打包和部署应用程序。优点是部署速度更快、磁盘空间使用更少，并可以在某些情况下提升启动性能。

此功能实现为运行时包存储区，这是包在磁盘上的存储目录（通常情况下，在 macOS/Linux 上是 /usr/local/share/dotnet/store，在 Windows 上是 C:/Program Files/dotnet/store）。此目录下有各个体系结构和目标框架的子目录。文件布局类似于磁盘上的 NuGet 资产布局：

```
\dotnet
    \store
        \x64
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
        \x86
            \netcoreapp2.0
                \microsoft.applicationinsights
                \microsoft.aspnetcore
                ...
```

目标清单文件列出了运行时包存储区中的包。开发者可以在发布应用程序时以此清单为目标。目标清单通常是由目标生产环境的所有者提供。

准备运行时环境

运行时环境管理员可以生成运行时包存储区和相应的目标清单，从而优化应用程序，以加快部署速度并释放磁盘空间。

第一步是创建包存储区清单，用于列出运行时包存储区中的包。此文件格式与项目文件格式 (csproj) 兼容。

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="NUGET_PACKAGE" Version="VERSION" />
    <!-- Include additional packages here -->
  </ItemGroup>
</Project>
```

示例

下面的示例包存储区清单 (packages.csproj) 用于将 `Newtonsoft.Json` 和 `Moq` 添加到运行时包存储区：

```
<Project Sdk="Microsoft.NET.Sdk">
  <ItemGroup>
    <PackageReference Include="Newtonsoft.Json" Version="10.0.3" />
    <PackageReference Include="Moq" Version="4.7.63" />
  </ItemGroup>
</Project>
```

通过执行 `dotnet store` 并指定包存储区清单、运行时和框架，预配运行时包存储区：

```
dotnet store --manifest <PATH_TO_MANIFEST_FILE> --runtime <RUNTIME_IDENTIFIER> --framework <FRAMEWORK>
```

示例

```
dotnet store --manifest packages.csproj --runtime win10-x64 --framework netcoreapp2.0 --framework-version
2.0.0
```

可以将多个目标包存储区清单路径传递到一个 `dotnet store` 命令，具体方法是在此命令中重复指定选项和路径。

默认情况下，命令输出是在用户配置文件的 .dotnet/store 子目录下生成包存储区。可以使用 `--output <OUTPUT_DIRECTORY>` 选项指定其他位置。存储区的根目录包含目标清单 artifact.xml 文件。此文件可供下载。如果应用程序创建者在发布时以这个存储区为目标，可以使用此文件。

示例

运行上一示例后生成以下 artifact.xml 文件。请注意，由于 `Castle.Core` 是 `Moq` 的依赖项，因此 artifacts.xml 清单文件会自动包含并显示它。

```
<StoreArtifacts>
  <Package Id="Newtonsoft.Json" Version="10.0.3" />
  <Package Id="Castle.Core" Version="4.1.0" />
  <Package Id="Moq" Version="4.7.63" />
</StoreArtifacts>
```

根据目标清单发布应用程序

如果磁盘上有目标清单文件，请在使用 `dotnet publish` 命令发布应用程序时指定此文件的路径：

```
dotnet publish --manifest <PATH_TO_MANIFEST_FILE>
```

示例

```
dotnet publish --manifest manifest.xml
```

将生成的已发布应用程序部署到包含目标清单中所述包的环境内。如果不这样做，应用程序便无法启动。

发布应用程序时，通过重复指定选项和路径（例如，`--manifest manifest1.xml --manifest manifest2.xml`），可以指定多个目标清单。这样，应用程序会进行剪裁，依据为提供给命令的目标清单文件中指定的包并集。

如果使用部署中的清单依赖项（程序集位于 bin 文件夹中）部署应用程序，运行时包存储区不会 在主机上用于相应程序集。将使用 bin 文件夹程序集，无论它是否位于主机上的运行时包存储区中。

清单中指明的依赖项版本必须与运行时包存储区中的依赖项版本一致。如果目标清单与运行时包存储区中的依赖项版本不一致，并且应用程序的部署中没有包的相应版本，那么应用程序将无法启动。例外情况包括，为运行时包存储区程序集调用的目标清单名称，这有助于排查不一致问题。

如果在发布时 部署发生剪裁，只有指明的特定版本清单包，才不会出现在已发布的输出中。主机上必须有指明的包版本，应用程序才能启动。

在项目文件中指定目标清单

除了使用 `dotnet publish` 命令指定目标清单，还可以在项目文件中将目标清单指定为 <TargetManifestFiles> 标

记下的路径分号分隔列表。

```
<PropertyGroup>
  <TargetManifestFiles>manifest1.xml;manifest2.xml</TargetManifestFiles>
</PropertyGroup>
```

仅在应用程序的目标环境已知的情况下(如 .NET Core 项目)，才在项目文件中指定目标清单。开放源代码项目的情况有所不同。开放源代码项目的用户通常将项目部署到不同的生产环境。这些生产环境通常都预安装了各组不同的包。在这样的环境中，不能对目标清单作出假设，所以应使用 `dotnet publish` 的 `--manifest` 选项。

ASP.NET Core 隐式存储区(仅限 .NET Core 2.0)

ASP.NET Core 隐式存储仅适用于 ASP.NET Core 2.0。我们强烈建议应用程序使用 ASP.NET Core 2.1 及更高版本，这些版本不使用隐式存储 。ASP.NET Core 2.1 及更高版本使用共享框架。

对于 .NET Core 2.0，当 ASP.NET Core 应用部署为依赖框架的部署应用时，该应用会隐式使用运行时包存储区功能。`Microsoft.NET.Sdk.Web` 中的目标包括引用目标系统上的隐式包存储区的清单。另外，如果依赖框架的应用依赖 `Microsoft.AspNetCore.All` 包，则会生成仅包含应用及其资产的已发布应用，而不是 `Microsoft.AspNetCore.All` 元包中列出的包。假定这些包都位于目标系统上。

安装 .NET SDK 后，便会在主机上安装运行时包存储区。其他安装程序可能会提供运行时包存储区，包括 .NET SDK 的 Zip/tarball 安装、`apt-get`、Red Hat Yum、.NET Core Windows Server Hosting 捆绑包和手动运行时包存储区安装。

部署依赖于框架的部署应用时，请确保目标环境中已安装 .NET SDK。如果应用部署环境中未安装 ASP.NET Core，可以在项目文件中指定将 <PublishWithAspNetCoreTargetManifest> 设置为 `false`，从而选择退出隐式存储区，如以下示例所示：

```
<PropertyGroup>
  <PublishWithAspNetCoreTargetManifest>false</PublishWithAspNetCoreTargetManifest>
</PropertyGroup>
```

> **NOTE**
> 对于独立部署应用，假定目标系统不一定包含所需的清单包。因此，对于自包含应用，不能将 <PublishWithAspNetCoreTargetManifest> 设置为 `true`。

请参阅

- dotnet-publish
- dotnet-store

.NET RID 目录

2021/11/16 •

RID 是运行时标识符的缩写。RID 值用于标识应用程序运行所在的目标平台。.NET 包使用它们来表示 NuGet 包中特定于平台的资产。以下值是 RID 的示例：`linux-x64`、`ubuntu.14.04-x64`、`win7-x64` 或 `osx.10.12-x64`。对于具有本机依赖项的包，RID 将指定在其中可以还原包的平台。

可以在项目文件的 `<RuntimeIdentifier>` 元素中设置一个 RID。可以将多个 RID 定义为项目文件的 `<RuntimeIdentifiers>` 元素中的列表（以分号分隔）。也可使用以下 .NET CLI 命令 通过 `--runtime` 选项使用它们：

- dotnet build
- dotnet clean
- dotnet pack
- dotnet publish
- dotnet restore
- dotnet run
- dotnet store

表示具体操作系统的 RID 通常遵循以下模式：`[os].[version]-[architecture]-[additional qualifiers]`，其中：

- `[os]` 是操作系统/平台系统名字对象。例如 `ubuntu`。
- `[version]` 是操作系统版本，使用的格式是以点 (`.`) 分隔的版本号。例如 `15.10`。
  - 版本不应为营销版本，因为它们通常代表该操作系统的多个离散版本，且具有不同的平台 API 外围应用。
- `[architecture]` 是处理器体系结构。例如：`x86`、`x64`、`arm` 或 `arm64`。
- `[additional qualifiers]` 进一步区分了不同的平台。例如：`aot`。

RID 图表

RID 图表或运行时回退图表是互相兼容的 RID 列表。Microsoft.NETCore.Platforms 包中定义了 RID。可以在 `dotnet/runtime` 存储库的 runtime.json 文件中查看支持的 RID 列表和 RID 图表。在此文件中，可以看到除基 RID 以外的所有 RID 均包含 `"#import"` 语句。这些语句指示的是兼容的 RID。

NuGet 还原包时，它将尝试找到指定运行时的完全匹配项。如果未找到完全匹配项，NuGet 将返回此图表，直至它根据 RID 图表找到最相近的兼容系统。

以下示例是 `osx.10.12-x64` RID 的实际条目：

```
"osx.10.12-x64": {
    "#import": [ "osx.10.12", "osx.10.11-x64" ]
}
```

上述 RID 指定 `osx.10.12-x64` 导入 `osx.10.11-x64`。因此，当 NuGet 还原包时，它将尝试找到包中的 `osx.10.12-x64` 的完全匹配项。例如，如果 NuGet 无法找到特定的运行时，可以还原指定 `osx.10.11-x64` 运行时的包。

以下示例演示了 runtime.json 文件中定义的另一个略大的 RID 图表：

```
win7-x64    win7-x86
   |   \   /    |
   |   win7     |
   |     |      |
win-x64  |  win-x86
      \  |  /
        win
         |
        any
```

所有 RID 最终都会映射回根 `any` RID。

使用 RID 时，必须牢记以下几个注意事项：

- 请勿尝试分析 RID 来检索组件部分。
- 请勿以编程方式生成 RID。
- 使用已为平台定义的 RID。
- RID 必须具有特定性，因此请勿通过实际的 RID 值假定任何情况。

使用 RID

若要使用 RID，必须知道有哪些 RID。新值将定期添加到该平台。若要获取最新的完整版，请参阅 `dotnet/runtime` 存储库中的 runtime.json 文件。

未绑定到特定版本或 OS 发行版的 RID 是优先选择，尤其是在处理多个 Linux 发行版时，因为大多数发行版 RID 映射到非特定于发行版的 RID。

以下列表显示了一小部分用于每个 OS 的最常见 RID。

Windows RID

仅列出了公共值。若要获取最新的完整版，请参阅 `dotnet/runtime` 存储库中的 runtime.json 文件。

- 非特定于版本的 Windows
  - `win-x64`
  - `win-x86`
  - `win-arm`
  - `win-arm64`
- Windows 7 / Windows Server 2008 R2
  - `win7-x64`
  - `win7-x86`
- Windows 8.1 / Windows Server 2012 R2
  - `win81-x64`
  - `win81-x86`
  - `win81-arm`
- Windows 10 / Windows Server 2016
  - `win10-x64`
  - `win10-x86`
  - `win10-arm`
  - `win10-arm64`

有关详细信息，请参阅 .NET 依赖项和要求。

Linux RID

仅列出了公共值。若要获取最新的完整版，请参阅 `dotnet/runtime` 存储库中的 runtime.json 文件。运行以下未列出的发行版的设备可能适用于其中一个非特定于发行版的 RID。例如，可以将运行未列出的 Linux 发行版的 Raspberry Pi 设备定向为使用 `linux-arm`。

- 非特定于分发版的 Linux
  - `linux-x64`（大多数桌面发行版，如 CentOS、Debian、Fedora、Ubuntu 及派生版本）
  - `linux-musl-x64`（使用 musl 的轻量级发行版，如 Alpine Linux）
  - `linux-arm`（在 ARM 上运行的 Linux 发行版本，如 Raspberry Pi Model 2 及更高版本上的 Raspbian）
  - `linux-arm64`（在 64 位 ARM 上运行的 Linux 发行版本，如 Raspberry Pi Model 3 及更高版本上的 Ubuntu 服务器 64 位）
- Red Hat Enterprise Linux
  - `rhel-x64`（被 `linux-x64` 取代，适用于 RHEL 6 以上版本）
  - `rhel.6-x64`
- Tizen
  - `tizen`
  - `tizen.4.0.0`
  - `tizen.5.0.0`

有关详细信息，请参阅 .NET 依赖项和要求。

macOS RID

macOS RID 使用较早的“OSX”品牌。仅列出了公共值。若要获取最新的完整版，请参阅 `dotnet/runtime` 存储库中的 runtime.json 文件。

- 非特定于版本的 macOS
  - `osx-x64`（最低 OS 版本为 macOS 10.12 Sierra）
- macOS 10.10 Yosemite
  - `osx.10.10-x64`
- macOS 10.11 El Capitan
  - `osx.10.11-x64`
- macOS 10.12 Sierra
  - `osx.10.12-x64`
- macOS 10.13 High Sierra
  - `osx.10.13-x64`
- macOS 10.14 Mojave
  - `osx.10.14-x64`
- macOS 10.15 Catalina
  - `osx.10.15-x64`
- macOS 11.0 Big Sur
  - `osx.11.0-x64`
  - `osx.11.0-arm64`

有关详细信息，请参阅 .NET 依赖项和要求。

请参阅

- 运行时 ID

资源清单文件的命名方式

2021/11/16 •

当 MSBuild 编译 .NET Core 项目时，文件扩展名为 .resx 的 XML 资源文件将转换为二进制 .resources 文件 。该二进制文件会嵌入编译器的输出中，并且可以由 ResourceManager 读取。本文介绍 MSBuild 如何为每个 .resources 文件选择名称。

> **TIP**
> 如果将资源项显式添加到项目文件中，并且 .NET Core 的默认 include glob 也包含该项，则会出现生成错误。要以资源文件的形式手动包含 `EmbeddedResource` 项，请将 `EnableDefaultEmbeddedResourceItems` 属性设置为 false。

默认名称

在 .NET Core 3.0 及更高版本中，同时满足以下两个条件时，将使用资源清单的默认名称：

- 资源文件未作为具有 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据的 `EmbeddedResource` 项显式包含在项目文件中。
- 项目文件中未将 `EmbeddedResourceUseDependentUponConvention` 属性设置为 `false`。默认情况下，此属性设置为 `true`。有关详细信息，请参阅 EmbeddedResourceUseDependentUponConvention。

如果资源文件与具有同一根文件名的源文件（.cs 或 .vb）并置，清单文件名便使用源文件中定义的第一种类型的全名 。例如，如果 `MyNamespace.Form1` 是 Form1.cs 中定义的第一种类型，并且 Form1.cs 与 Form1.resx 并置，则该资源文件的生成清单名称是 MyNamespace.Form1.resources 。

LogicalName 元数据

如果资源文件作为具有 `LogicalName` 元数据的 `EmbeddedResource` 项显式包含在项目文件中，则将 `LogicalName` 值用作清单名称。`LogicalName` 优先于其他任何元数据或设置。

例如，以下项目文件片段中定义的资源文件的清单名称是 SomeName.resources。

```
<EmbeddedResource Include="X.resx" LogicalName="SomeName.resources" />
```

- 或 -

```
<EmbeddedResource Include="X.fr-FR.resx" LogicalName="SomeName.resources" />
```

NOTE

- 如果未指定 `LogicalName`，文件名中带有两个点 (`.`) 的 `EmbeddedResource` 不起作用，这意味着 `GetManifestResourceNames` 不会返回该文件。

  以下示例工作正常：

  ```
  <EmbeddedResource Include="X.resx" />
  ```

  以下示例无法工作：

  ```
  <EmbeddedResource Include="X.fr-FR.resx" />
  ```

ManifestResourceName 元数据

如果资源文件作为具有 `ManifestResourceName` 元数据（并且 `LogicalName` 不存在）的 `EmbeddedResource` 项显式包含在项目文件中，则 `ManifestResourceName` 值与文件扩展名 .resources 一起用作清单文件名。

例如，以下项目文件片段中定义的资源文件的清单名称是 SomeName.resources。

```
<EmbeddedResource Include="X.resx" ManifestResourceName="SomeName" />
```

以下项目文件片段中定义的资源文件的清单名称是 SomeName.fr-FR.resources。

```
<EmbeddedResource Include="X.fr-FR.resx" ManifestResourceName="SomeName.fr-FR" />
```

DependentUpon 元数据

如果资源文件作为具有 `DependentUpon` 元数据（并且 `LogicalName` 和 `ManifestResourceName` 不存在）的 `EmbeddedResource` 项显式包含在项目文件中，则将 `DependentUpon` 定义的源文件中的信息用于资源清单文件名。具体来说，清单名称中会使用源文件中定义的第一种类型的名称，如下所示：*Namespace.Classname[.Culture].resources*。

例如，在以下项目文件片段中定义的资源文件的清单名称是 Namespace.Classname.resources（其中 `Namespace.Classname` 是 MyTypes.cs 中定义的第一个类）。

```
<EmbeddedResource Include="X.resx" DependentUpon="MyTypes.cs">
```

在以下项目文件片段中定义的资源文件的清单名称是 Namespace.Classname.fr-FR.resources（其中 `Namespace.Classname` 是 MyTypes.cs 中定义的第一个类）。

```
<EmbeddedResource Include="X.fr-FR.resx" DependentUpon="MyTypes.cs">
```

EmbeddedResourceUseDependentUponConvention 属性

如果在项目文件中将 `EmbeddedResourceUseDependentUponConvention` 设置为 `false`，则每个资源清单文件名都基于项目的根命名空间以及从项目根目录到 .resx 文件的相对路径。更具体地说，生成的资源清单文件名是 *RootNamespace.RelativePathWithDotsForSlashes.[Culture.]resources*。这也是用于在低于 3.0 的 .NET Core 版本中生成清单名称的逻辑。

NOTE

- 如果未定义 `RootNamespace`，则默认为项目名称。
- 如果为项目文件中的 `EmbeddedResource` 项指定了 `LogicalName`、`ManifestResourceName` 或 `DependentUpon` 元数据，则该命名规则不适用于该资源文件。

另请参阅

- 清单资源的命名方式
- .NET SDK 项目的 MSBuild 属性
- MSBuild 中断性变更

.NET 和 Docker 简介

2021/11/16 •

.NET Core 可以在 Docker 容器中轻松运行。容器提供一种轻量级方法，用于将应用程序与主机系统的其他组件分隔，以便仅共享内核，并使用提供给应用程序的资源。如果你不熟悉 Docker，强烈建议通读 Docker 的概述文档。

若要详细了解如何安装 Docker，请参阅 Docker 桌面：社区版的下载页面。

Docker 基础知识

首先来熟悉几个概念。Docker 客户端具有可用于管理映像和容器的 CLI。如前文所述，应花时间通读 Docker 概述文档。

映像

映像是构成容器基础的文件系统更改的有序集合。映像不具有状态，并且是只读的。大多情况下，一个映像以另一个映像为基础，但具有一些自定义设置。例如，为应用程序创建新映像时，需使其基于已包含 .NET Core 运行时的现有映像。

由于容器是从映像创建的，因此，映像具有一组在容器启动时运行的运行参数（例如启动可执行文件）。

容器

容器是映像的可运行实例。生成映像时，部署应用程序和依赖项。然后可以实例化多个容器，且每个容器相互独立。每个容器实例具有其自己的文件系统、内存和网络接口。

注册表

容器注册表是映像存储库的集合。映像可以基于注册表映像。可以直接从注册表中的映像创建容器。Docker 容器、映像和注册表之间的关系是构建和生成容器化应用程序或微服务时的一个重要概念。此方法大大缩短了开发和部署之间的时间。

Docker 具有一个托管在 Docker 中心的公共注册表，可供用户使用。.NET Core 相关映像均在 Docker 中心列出。

Microsoft 容器注册表 (MCR) 是 Microsoft 提供的容器映像的官方来源。MCR 构建在 Azure CDN 之上，可提供用于全局复制的映像。但是，MCR 没有面向公众的网站，了解有关 Microsoft 提供的容器映像的主要方法是通过 Microsoft Docker 中心页面。

Dockerfile

Dockerfile 是定义可创建映像的一组指令的文件。Dockerfile 中的每个指令创建映像中的一个层。大多数情况下，在重新生成映像时，只会重新生成已发生更改的层。可以将 Dockerfile 分发给其他人，便于他们采用你创建映像的方式重新创建一个新映像。尽管可以分发有关如何创建映像的指令，但分发映像的主要方式是将其发布到注册表。

.NET Core 映像

官方 .NET Core Docker 映像发布到 Microsoft 容器注册表 (MCR)，用户可以在 Microsoft.NET Core Docker 中心存储库中找到这些映像。每个存储库包含 .NET（SDK 或运行时）和可以使用的操作系统的不同组合的映像。

Microsoft 提供适合特定场景的映像。例如，ASP.NET Core 存储库提供针对在生产环境中运行 ASP.NET Core 应用生成的映像。

Azure 服务

各种 Azure 服务都支持容器。为应用程序创建 Docker 映像并将其部署到以下服务之一：

- Azure Kubernetes 服务 (AKS)
  使用 kubernetes 缩放和编排 Linux 容器。
- Azure 应用服务
  在 PaaS 环境中使用 Linux 容器部署 Web 应用或 API。
- Azure 容器实例
  将容器托管在云中，而不使用任何高级管理服务。
- Azure Batch
  使用容器运行重复的计算作业。
- Azure Service Fabric
  使用 Windows Server 容器直接迁移 .NET 应用程序并将其现代化为微服务。
- Azure 容器注册表
  在所有类型的 Azure 部署中存储和管理容器映像。

后续步骤

- 了解如何容器化 .NET Core 应用。
- 了解如何容器化 ASP.NET Core 应用程序。
- 试学“ASP.NET Core 微服务”教程。
- 了解 Visual Studio 中的容器工具

教程：使 .NET Core 应用程序容器化

2021/11/16 •

在本教程中，你将了解如何使用 Docker 容器化 .NET Core 应用。容器具有很多特性和优点，如具有不可变的基础结构、提供可移植的体系结构和实现可伸缩性。此映像可用于为本地开发环境、私有云或公有云创建容器。

在本教程中，你将了解：

- 创建并发布简单的 .NET Core 应用
- 创建并配置用于 .NET Core 的 Dockerfile
- 生成 Docker 映像
- 创建并运行 Docker 容器

你将了解用于 .NET Core 应用的 Docker 容器生成和部署任务。Docker 平台 使用 Docker 引擎 快速生成应用，并将应用打包为 Docker 映像 。这些映像采用 Dockerfile 格式编写，以供在分层容器中部署和运行。

> **NOTE**
> 本教程不适用于 ASP.NET Core 应用 。如果使用的是 ASP.NET Core，请参阅教程了解如何容器化 ASP.NET Core 应用程序。

先决条件

安装以下必备组件：

- .NET Core 5.0 SDK
  如果已安装 .NET Core，请使用 `dotnet --info` 命令来确定使用的是哪个 SDK。
- Docker 社区版
- Dockerfile 和 .NET Core 示例应用的临时工作文件夹 。在本教程中，docker-working 用作工作文件夹的名称。

创建 .Net Core 应用

需要有可供 Docker 容器运行的 .NET Core 应用。打开终端、创建工作文件夹（如果尚没有），然后进入该文件夹。在工作文件夹中，运行下面的命令，在名为“app”的子目录中新建一个项目 ：

```
dotnet new console -o App -n NetCore.Docker
```

文件夹树将如下所示：

```
docker-working
    └─App
        ├─NetCore.Docker.csproj
        ├─Program.cs
        └─obj
            ├─NetCore.Docker.csproj.nuget.dgspec.json
            ├─NetCore.Docker.csproj.nuget.g.props
            ├─NetCore.Docker.csproj.nuget.g.targets
            ├─project.assets.json
            └─project.nuget.cache
```

`dotnet new` 命令会新建一个名为“应用”的文件夹，并生成一个“Hello World”控制台应用程序 。从终端会话更改目录并导航到“App”文件夹 。使用 `dotnet run` 命令启动应用。应用程序将运行，并在命令下方打印

`Hello World!`：

```
dotnet run
Hello World!
```

默认模板创建应用，此应用先打印输出到终端，然后立即终止。本教程将使用无限循环的应用。在文本编辑器中，打开“Program.cs” 文件。

> **TIP**
>
> 如果使用 Visual Studio Code，则从上一个终端会话中键入以下命令：
>
> ```
> code .
> ```
>
> 这会在 Visual Studio Code 中打开包含该项目的“App”文件夹 。

Program.cs 应如下面的 C# 代码所示 ：

```
using System;

namespace NetCore.Docker
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

将此文件替换为以下每秒计数一次的代码：

```
using System;
using System.Threading.Tasks;

namespace NetCore.Docker
{
    class Program
    {
        static async Task Main(string[] args)
        {
            var counter = 0;
            var max = args.Length != 0 ? Convert.ToInt32(args[0]) : -1;
            while (max == -1 || counter < max)
            {
                Console.WriteLine($"Counter: {++counter}");
                await Task.Delay(1000);
            }
        }
    }
}
```

保存此文件，并使用 `dotnet run` 再次测试程序。请注意，此应用无限期运行。使用取消命令 Ctrl+C 可以停止运行。下面是一个示例输出：

```
dotnet run
Counter: 1
Counter: 2
Counter: 3
Counter: 4
^C
```

如果你在命令行中向应用传递一个数字，它就只会计数到这个数字，然后退出。试一试用 `dotnet run -- 5` 计数到 5。

> **IMPORTANT**
> `--` 之后的参数都不传递到 `dotnet run` 命令，而是传递到你的应用程序。

发布 .Net Core 应用

在将 .NET Core 应用添加到 Docker 映像之前，必须先发布该应用。最好让容器运行应用的已发布版本。若要发布应用，请运行以下命令：

```
dotnet publish -c Release
```

此命令将应用编译到"发布"文件夹中 。从工作文件夹到"发布"文件夹的路径应为 `.\App\bin\Release\net5.0\publish\`

- Windows
- Linux

在"应用"文件夹中获取"发布"文件夹的目录列表，以验证 NetCore.Docker.dll 文件是否已创建 。

```
dir .\bin\Release\net5.0\publish\

    Directory: C:\Users\dapine\App\bin\Release\net5.0\publish

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        4/27/2020   8:27 AM            434 NetCore.Docker.deps.json
-a----        4/27/2020   8:27 AM           6144 NetCore.Docker.dll
-a----        4/27/2020   8:27 AM         171520 NetCore.Docker.exe
-a----        4/27/2020   8:27 AM            860 NetCore.Docker.pdb
-a----        4/27/2020   8:27 AM            154 NetCore.Docker.runtimeconfig.json
```

创建 Dockerfile

`docker build` 命令使用 Dockerfile 文件来创建容器映像。此文件是名为"Dockerfile" 的文本文件，它没有扩展名。

在包含 .csproj 的目录中创建名为"Dockerfile"的文件，并在文本编辑器中将其打开 。本教程将使用 ASP.NET Core 运行时映像（包含 .NET Core 运行时映像），并与 .NET Core 控制台应用程序相对应。

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0
```

> **NOTE**
> 尽管可能已使用 `mcr.microsoft.com/dotnet/runtime:5.0` 映像，但此处有意使用 ASP.NET Core 运行时映像。

`FROM` 关键字需要完全限定的 Docker 容器映像名称。Microsoft 容器注册表（MCR，mcr.microsoft.com）是 Docker Hub 的联合，可托管可公开访问的容器。`dotnet` 段是容器存储库，其中 `aspnet` 段是容器映像名称。该映像使用 `5.0` 进行标记，它用于版本控制。因此，`mcr.microsoft.com/dotnet/aspnet:5.0` 是 .NET Core 5.0 运行时。请确保拉取的运行时版本与 SDK 面向的运行时一致。例如，在上一节中创建的应用使用了 .NET Core 5.0 SDK，在 Dockerfile 中引用的基础映像被标记为 5.0。

保存 Dockerfile 文件 。工作文件夹的目录结果应如下所示。为节省本文的空间，省略了一些更深级别的文件和文件夹：

```
docker-working
    └─App
        ├─Dockerfile
        ├─NetCore.Docker.csproj
        ├─Program.cs
        ├─bin
        │   └─Release
        │       └─net5.0
        │           └─publish
        │               ├─NetCore.Docker.deps.json
        │               ├─NetCore.Docker.exe
        │               ├─NetCore.Docker.dll
        │               ├─NetCore.Docker.pdb
        │               └─NetCore.Docker.runtimeconfig.json
        └─obj
            └─...
```

在终端中运行以下命令：

```
docker build -t counter-image -f Dockerfile .
```

Docker 会处理 Dockerfile 中的每一行。`docker build` 命令中的 `.` 指示 Docker 在当前文件夹中查找 Dockerfile 。此命令生成映像，并创建指向相应映像的本地存储库“counter-image” 。在此命令完成后，运行 `docker images` 以列出已安装的映像：

```
docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
counter-image                       latest              e6780479db63        4 days ago          190MB
mcr.microsoft.com/dotnet/aspnet     5.0                 e6780479db63        4 days ago          190MB
```

请注意，两个映像共用相同的“IMAGE ID” 值。两个映像使用的 ID 值相同是因为，Dockerfile 中的唯一命令是在现有映像的基础之上生成新映像。接下来，在 Dockerfile 中添加三个命令 。每个命令（或指令）都创建一个新的映像层 — 最后一个命令表示生成的 counter-image 存储库入口点。

```
COPY bin/Release/net5.0/publish/ App/
WORKDIR /App
ENTRYPOINT ["dotnet", "NetCore.Docker.dll"]
```

`COPY` 命令指示 Docker 将计算机上的指定文件夹复制到容器中的文件夹。在此示例中，“publish”文件夹被复制到容器中的“App”文件夹 。

`WORKDIR` 命令将容器内的当前目录更改为“App” 。

下一个命令 `ENTRYPOINT` 指示 Docker 将容器配置为可执行文件运行。在容器启动时，`ENTRYPOINT` 命令运行。当此命令结束时，容器也会自动停止。

> **TIP**
>
> 为了提高安全性，可以选择退出诊断管道。选择退出后，容器将以只读方式运行。为此，请将 `DOTNET_EnableDiagnostics` 环境变量指定为 `0`（就在 `ENTRYPOINT` 步骤之前）：
>
> ```
> ENV DOTNET_EnableDiagnostics=0
> ```

> **NOTE**
>
> .NET 6 为用于配置 .NET 运行时行为的环境变量标准化前缀 `DOTNET_` 而不是 `COMPlus_`。但是，`COMPlus_` 前缀仍将继续正常工作。如果使用的是早期版本的 .NET 运行时，则环境变量仍应该使用 `COMPlus_` 前缀。

在终端中，运行 `docker build -t counter-image -f Dockerfile .`；在此命令完成后，运行 `docker images`。

```
docker build -t counter-image -f Dockerfile .
Sending build context to Docker daemon  1.117MB
Step 1/4 : FROM mcr.microsoft.com/dotnet/aspnet:5.0
 ---> e6780479db63
Step 2/4 : COPY bin/Release/net5.0/publish/ App/
 ---> d1732740eed2
Step 3/4 : WORKDIR /App
 ---> Running in b1701a42f3ff
Removing intermediate container b1701a42f3ff
 ---> 919aab5b95e3
Step 4/4 : ENTRYPOINT ["dotnet", "NetCore.Docker.dll"]
 ---> Running in c12aebd26ced
Removing intermediate container c12aebd26ced
 ---> cd11c3df9b19
Successfully built cd11c3df9b19
Successfully tagged counter-image:latest

docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
counter-image                       latest              cd11c3df9b19        41 seconds ago      190MB
mcr.microsoft.com/dotnet/aspnet     5.0                 e6780479db63        4 days ago          190MB
```

Dockerfile 中的每个命令生成了一个层，并创建了“IMAGE ID”。最终“IMAGE ID”是“cd11c3df9b19”（你的 ID 会有所不同），接下来在此映像的基础之上创建容器。

创建容器

至此，已有包含应用的映像，可以创建容器了。可以通过两种方式来创建容器。首先，新建已停止的容器。

```
docker create --name core-counter counter-image
0f281cb3af994fba5d962cc7d482828484ea14ead6bfe386a35e5088c0058851
```

上面的 `docker create` 命令在 counter-image 映像的基础之上创建容器。此命令的输出显示已创建容器的“CONTAINER ID”（你的 ID 会有所不同）。若要查看所有 容器的列表，请使用 `docker ps -a` 命令：

```
docker ps -a
CONTAINER ID    IMAGE           COMMAND                  CREATED          STATUS     PORTS    NAMES
0f281cb3af99    counter-image   "dotnet NetCore.Dock…"   40 seconds ago   Created             core-
counter
```

管理容器

容器是使用特定名称 `core-counter` 创建的，此名称用于管理容器。下面的示例使用 `docker start` 命令来启动容器，然后使用 `docker ps` 命令仅显示正在运行的容器：

```
docker start core-counter
core-counter

docker ps
CONTAINER ID    IMAGE           COMMAND                  CREATED          STATUS          PORTS    NAMES
2f6424a7ddce    counter-image   "dotnet NetCore.Dock…"   2 minutes ago    Up 11 seconds            core-
counter
```

同样，`docker stop` 命令会停止容器。下面的示例使用 `docker stop` 命令来停止容器，然后使用 `docker ps` 命令来显示未在运行的容器：

```
docker stop core-counter
core-counter

docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

连接到容器

在容器运行后，可以连接到它来查看输出。使用 `docker start` 和 `docker attach` 命令，启动容器并查看输出流。在此示例中，Ctrl+C 击键用于从正在运行的容器中分离出来。除非另行指定，否则此击键将结束容器中的进程，这会停止容器。`--sig-proxy=false` 参数可确保 Ctrl+C 不会停止容器中的进程。

从容器中分离出来后重新连接，以验证它是否仍在运行和计数。

```
docker start core-counter
core-counter

docker attach --sig-proxy=false core-counter
Counter: 7
Counter: 8
Counter: 9
^C

docker attach --sig-proxy=false core-counter
Counter: 17
Counter: 18
Counter: 19
^C
```

删除容器

就本文而言，你不希望存在不执行任何操作的容器。删除前面创建的容器。如果容器正在运行，停止容器。

```
docker stop core-counter
```

下面的示例列出了所有容器。然后，它使用 `docker rm` 命令来删除容器，并再次检查是否有任何正在运行的容器。

```
docker ps -a
CONTAINER ID    IMAGE            COMMAND                  CREATED          STATUS
PORTS    NAMES
2f6424a7ddce    counter-image    "dotnet NetCore.Dock…"   7 minutes ago    Exited (143) 20 seconds ago
core-counter

docker rm core-counter
core-counter

docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

单次运行

Docker 提供了 `docker run` 命令，用于将容器作为单一命令进行创建和运行。使用此命令，无需依次运行 `docker create` 和 `docker start`。另外，还可以将此命令设置为，在容器停止时自动删除容器。例如，使用 `docker run -it --rm` 可以执行两项操作，先自动使用当前终端连接到容器，再在容器完成时删除容器：

```
docker run -it --rm counter-image
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5
^C
```

容器还会将参数传递给 .NET Core 应用的执行。指示 .NET Core 应用仅计数为 3 个传入 3 个。

```
docker run -it --rm counter-image 3
Counter: 1
Counter: 2
Counter: 3
```

使用 `docker run -it`，`Ctrl+C` 命令会停止在容器中运行的进程，进而停止容器。由于提供了 `--rm` 参数，因此在进程停止时自动删除容器。验证它是否不存在：

```
docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
```

更改 ENTRYPOINT

使用 `docker run` 命令，还可以修改 Dockerfile 中的 `ENTRYPOINT` 命令，并运行其他内容，但只能针对相应容器。例如，使用以下命令来运行 `bash` 或 `cmd.exe`。根据需要，编辑此命令。

- Windows
- Linux

在本例中，`ENTRYPOINT` 更改为 `cmd.exe`。通过按下 `Ctrl+C` 来结束进程并停止容器。

```
docker run -it --rm --entrypoint "cmd.exe" counter-image

Microsoft Windows [Version 10.0.17763.379]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\>dir
 Volume in drive C has no label.
 Volume Serial Number is 3005-1E84

 Directory of C:\

04/09/2019  08:46 AM    <DIR>          app
03/07/2019  10:25 AM             5,510 License.txt
04/02/2019  01:35 PM    <DIR>          Program Files
04/09/2019  01:06 PM    <DIR>          Users
04/02/2019  01:35 PM    <DIR>          Windows
               1 File(s)          5,510 bytes
               4 Dir(s)  21,246,517,248 bytes free

C:\>^C
```

重要命令

Docker 包含许多不同的命令，可用于创建、管理以及与容器和映像进行交互。下面这些 Docker 命令对于管理容器来说至关重要：

- docker build
- docker run
- docker ps
- docker stop
- docker rm
- docker rmi
- docker image

清理资源

在本教程中，你创建了容器和映像。如果需要，请删除这些资源。以下命令可用于

1. 列出所有容器

   ```
   docker ps -a
   ```

2. 按名称停止正在运行的容器。

   ```
   docker stop counter-image
   ```

3. 删除容器

   ```
   docker rm counter-image
   ```

接下来，删除计算机上不再需要的任何映像。依次删除 Dockerfile 创建的映像，以及 Dockerfile 所依据的 .NET Core 映像。可以使用 IMAGE ID 或 REPOSITORY:TAG 格式字符串。

```
docker rmi counter-image:latest
docker rmi mcr.microsoft.com/dotnet/aspnet:5.0
```

使用 `docker images` 命令来列出已安装的映像。

> **TIP**
>
> 映像文件可能很大。通常情况下，需要删除在测试和开发应用期间创建的临时容器。如果计划在相应运行时的基础之上生成其他映像，通常会将基础映像与运行时一同安装。

后续步骤

- 了解如何容器化 ASP.NET Core 应用程序。
- 试学“ASP.NET Core 微服务”教程。
- 查看支持容器的 Azure 服务。
- 了解 Dockerfile 命令。
- 了解用于 Visual Studio 的容器工具

常规类型系统和公共语言规范

2021/11/16 •

再次介绍 .NET 领域内大量使用的两个术语，对于理解 .NET 实现如何能够允许多语言开发及其工作原理，这两个术语至关重要。

常规类型系统

首先，请记住 .NET 实现对语言不可知。这不仅意味着程序员可以使用可编译为 IL 的任意语言编写代码。还意味着程序员需要能够与使用 .NET 实现上可用的其他语言编写的代码进行交互。

为透明地执行此操作，必须使用某种通用方式描所有受支持类型。这正是通用类型系统 (CTS) 的职责。其功能如下：

- 建立用于跨语言执行的框架。
- 提供面向对象的模型，支持在 .NET 实现上实现各种语言。
- 定义处理类型时所有语言都必须遵守的一组规则。
- 提供包含应用程序开发中使用的基本基元数据类型（如 `Boolean`、`Byte`、`Char` 等）的库。

CTS 将定义应支持的两种主要类型：引用和值类型。其名称表明了其定义。

引用类型对象由对对象实际值的引用表示；此处的引用与 C/C++ 中的指针类似。它仅引用对象值所在的内存位置。这对这些类型的用法有重大影响。例如，如果将引用类型分配给个变量，然后将该变量传递到方法，则对该对象的任意更改都将反映到主对象上；不涉及复制操作。

值类型则相反，其中对象由其值表示。如果将值类型分配给变量，实质上就是复制对象的值。

CTS 将定义多个类型类别，每个类别均有其特定的语义和用法：

- 类
- 结构
- 枚举
- 界面
- 委托

CTS 还将定义类型的所有其他属性，例如访问修饰符、有效的类型成员以及继承和重载的工作原理等。遗憾的是，有关上述任意一点的深入介绍已超出此类介绍性文章的介绍范围，但可以参阅底部的更多资源部分，获取包含这些主题的详细内容的链接。

公共语言规范

为实现完全互操作性情景，代码中创建的所有对象都必须依赖于使用它的语言（即其调用方）的某些共性。由于存在多种不同语言，因此 .NET 在公共语言规范 (CLS) 中指定了这些共性。CLS 定义了许多常见应用程序所需的一组功能。对于在 .NET 上实现的语言，它还就语言需要支持的内容提供了一组脚本。

CLS 是 CTS 的子集。这意味着，CTS 中的所有规则也适用于 CLS，除非 CLS 规则更严格。如果仅使用 CLS 中的规则生成组件（即在其 API 中仅公开 CLS 功能），则将该组件视为 **符合 CLS**。例如，`<framework-librares>` 完全符合 CLS，因为它们需要对 .NET 支持的所有语言有效。

可参阅下方更多资源部分中的文档，大致了解 CLS 中的所有功能。

更多资源

- 常规类型系统
- 公共语言规范

通用类型系统

2021/11/16 •

通用类型系统定义了如何在公共语言运行时中声明、使用和管理类型，同时也是运行时跨语言集成支持的一个重要组成部分。常规类型系统执行以下功能：

- 建立一个支持跨语言集成、类型安全和高性能代码执行的框架。
- 提供一个支持完整实现多种编程语言的面向对象的模型。
- 定义各语言必须遵守的规则，有助于确保用不同语言编写的对象能够交互作用。
- 提供包含应用程序开发中使用的基元数据类型（如Boolean、Byte、Char、Int32 和 UInt64）的库。

.NET 中的类型

.NET 中的所有类型不是值类型就是引用类型。

值类型是使用对象实际值来表示对象的数据类型。如果向一个变量分配值类型的实例，则该变量将被赋以该值的全新副本。

引用类型是使用对对象实际值的引用（类似于指针）来表示对象的数据类型。如果为某个变量分配一个引用类型，则该变量将引用（或指向）原始值。不创建任何副本。

.NET 中的通用类型系统支持以下五种类别的类型：

- 类
- 结构
- 枚举
- 接口
- 委托

类

类是可以直接从另一个类派生以及从 System.Object 隐式派生的引用类型。类定义对象（它是该类的实例）可以执行的操作（方法、事件或属性）和该对象包含的数据（字段）。尽管类通常同时包含定义和实现（与接口不同，例如，接口只包含定义而不包含实现），但它也可以有一个或多个没有实现的成员。

下表介绍了类可以具有的一些特征。支持运行时的每种语言都提供了一种方法，来指示类或类成员具有其中的一种或多种特征。但是，针对 .NET 的各个编程语言不能使所有这些特征都可用。

sealed	指定不能从此类型派生出另一个类。
实现	指出该类通过提供接口成员的实现，使用一个或多个接口。
abstract	指出无法实例化类。若要使用它，必须由其派生出另一个类。
继承	指出可以在指定了基类的任何地方使用类的实例。从基类继承的派生类可以使用基类提供的任何公共成员的实现，派生类也可以用自己的实现重写这些公共成员的实现。

“	“
exported 或 not exported	指出某个类在定义它的程序集之外是否可见。此特征仅适用于顶级类，不适用于嵌套类。

> **NOTE**
> 类也可以嵌套在父类或结构中。嵌套类也有成员特征。有关详细信息，请参阅嵌套类型。

没有实现的类成员是抽象成员。有一个或更多抽象成员的类其本身也是抽象的；不可以创建它的新实例。以运行时为目标的某些语言允许将类标记为抽象类，即使其成员都不是抽象成员也是如此。当要封装一组派生类可在适当时候继承或重写的基本功能时，可以使用抽象类。非抽象的类称为具体类。

类可以实现任意数目的接口，但是它除了 System.Object（所有类都可以隐式从它继承）之外，只能从一个基类继承。所有的类都必须至少有一个构造函数，该函数初始化此类的新实例。如果没有显式定义构造函数，大多数编译器将自动提供一个无参数构造函数。

结构

结构是隐式从 System.ValueType 派生的值类型，后者则是从 System.Object 派生的。对于表示内存要求较小的值以及将值作为按值参数传递给具有强类型参数的方法，结构很有用。在 .NET 中，所有基元数据类型（Boolean、Byte、Char、DateTime、Decimal、Double、Int16、Int32、Int64、SByte、Single、UInt16、UInt32 和 UInt64）都定义为结构。

像类一样，结构同时定义数据（结构的字段）和可以对该数据执行的操作（结构的方法）。这意味着可以对结构调用方法，包括在 System.Object 和 System.ValueType 类上定义的虚方法以及在值类型自身上定义的任意方法。换句话说，结构可以具有字段、属性和事件以及静态和非静态方法。您可以创建结构的实例，将它们作为参数传递，将它们存储为局部变量，或将它们存储在另一值类型或引用类型的字段中。结构也可以实现接口。

值类型与类在一些方面也是不同的。首先，尽管它们都从 System.ValueType 隐式继承，但是它们不能从任何类型直接继承。同样，所有值类型都是密封的，这意味着不能从它们派生出其他类型。它们也不需要构造函数。

对于每一种值类型，公共语言运行时都提供一种相应的已装箱类型，它是与值类型有着相同状态和行为的类。将值类型的实例传递到接受 System.Object 类型的参数的方法时，会将它装箱。当控件从接受值类型作为传引用参数的方法调用返回时，会将它拆箱（即它从类实例重新转换回值类型的实例）。当需要已装箱的类型时，某些语言要求使用特殊的语法；而另外一些语言会在需要时自动使用已装箱的类型。在定义值类型时，需要同时定义已装箱和未装箱的类型。

枚举

枚举是一种值类型，该值类型直接从 System.Enum 继承并为基础基元类型的值提供替代名称。枚举类型具有一个名称、一个必须为某个内置带符号或不带符号的整数类型的基础类型（如 Byte、Int32 或 UInt64）以及一组字段。字段是静态文本字段，其中的每一个字段都表示常数。同一个值可以分配给多个字段。出现这种情况时，必须将其中某个值标记为主要枚举值，以便进行反射和字符串转换。

可以将基础类型的值分配给枚举，反之亦然（运行时不要求强制转换）。可以创建枚举的实例，并调用 System.Enum 的方法以及在枚举的基础类型上定义的任何方法。但是，某些语言可能不允许在要求基础类型的实例时将枚举作为参数传递（反之亦然）。

对于枚举还有以下附加限制：

- 它们不能定义自己的方法。
- 它们不能实现接口。
- 它们不能定义属性或事件。
- 枚举不能是泛型，除非它嵌套在泛型类型中，才能是泛型。也就是说，枚举不能有自己的类型参数。

> **NOTE**
>
> 用 Visual Basic、C# 和 C++ 创建的嵌套类型(包括枚举)包含所有封闭泛型类型的类型参数，因此即使这些嵌套类型没有自己的类型参数，它们也是泛型的。有关更多信息，请参见 Type.MakeGenericType 参考主题中的"嵌套类型"。

FlagsAttribute 特性表示一种特殊的枚举，称为位域。运行时本身不区分传统枚举与位域，但您的语言可能会区分二者。当区分二者的时候，可以对位域(而不是枚举)使用位操作符以产生未命名的值。枚举一般用于列出唯一的元素，如一周的各天、国家/地区名称，等等。位域一般用于列出可能联合发生的质量或数量，比如 `Red And Big And Fast`。

下面的示例说明如何使用位域和传统枚举。

```
using System;
using System.Collections.Generic;

// A traditional enumeration of some root vegetables.
public enum SomeRootVegetables
{
    HorseRadish,
    Radish,
    Turnip
}

// A bit field or flag enumeration of harvesting seasons.
[Flags]
public enum Seasons
{
    None = 0,
    Summer = 1,
    Autumn = 2,
    Winter = 4,
    Spring = 8,
    All = Summer | Autumn | Winter | Spring
}

public class Example
{
   public static void Main()
   {
       // Hash table of when vegetables are available.
       Dictionary<SomeRootVegetables, Seasons> AvailableIn = new Dictionary<SomeRootVegetables, Seasons>();

       AvailableIn[SomeRootVegetables.HorseRadish] = Seasons.All;
       AvailableIn[SomeRootVegetables.Radish] = Seasons.Spring;
       AvailableIn[SomeRootVegetables.Turnip] = Seasons.Spring |
            Seasons.Autumn;

       // Array of the seasons, using the enumeration.
       Seasons[] theSeasons = new Seasons[] { Seasons.Summer, Seasons.Autumn,
            Seasons.Winter, Seasons.Spring };

       // Print information of what vegetables are available each season.
       foreach (Seasons season in theSeasons)
       {
          Console.Write(String.Format(
              "The following root vegetables are harvested in {0}:\n",
              season.ToString("G")));
          foreach (KeyValuePair<SomeRootVegetables, Seasons> item in AvailableIn)
          {
             // A bitwise comparison.
             if (((Seasons)item.Value & season) > 0)
                 Console.Write(String.Format("  {0:G}\n",
                      (SomeRootVegetables)item.Key));
          }
       }
```

```
        }
    }
}
// The example displays the following output:
//    The following root vegetables are harvested in Summer:
//      HorseRadish
//    The following root vegetables are harvested in Autumn:
//      Turnip
//      HorseRadish
//    The following root vegetables are harvested in Winter:
//      HorseRadish
//    The following root vegetables are harvested in Spring:
//      Turnip
//      Radish
//      HorseRadish
```

```
Imports System.Collections.Generic

' A traditional enumeration of some root vegetables.
Public Enum SomeRootVegetables
    HorseRadish
    Radish
    Turnip
End Enum

' A bit field or flag enumeration of harvesting seasons.
<Flags()> Public Enum Seasons
    None = 0
    Summer = 1
    Autumn = 2
    Winter = 4
    Spring = 8
    All = Summer Or Autumn Or Winter Or Spring
End Enum

' Entry point.
Public Class Example
    Public Shared Sub Main()
        ' Hash table of when vegetables are available.
        Dim AvailableIn As New Dictionary(Of SomeRootVegetables, Seasons)()

        AvailableIn(SomeRootVegetables.HorseRadish) = Seasons.All
        AvailableIn(SomeRootVegetables.Radish) = Seasons.Spring
        AvailableIn(SomeRootVegetables.Turnip) = Seasons.Spring Or _
                                                 Seasons.Autumn

        ' Array of the seasons, using the enumeration.
        Dim theSeasons() As Seasons = {Seasons.Summer, Seasons.Autumn, _
                                       Seasons.Winter, Seasons.Spring}

        ' Print information of what vegetables are available each season.
        For Each season As Seasons In theSeasons
            Console.WriteLine(String.Format( _
                 "The following root vegetables are harvested in {0}:", _
                 season.ToString("G")))
            For Each item As KeyValuePair(Of SomeRootVegetables, Seasons) In AvailableIn
                ' A bitwise comparison.
                If (CType(item.Value, Seasons) And season) > 0 Then
                    Console.WriteLine("  " + _
                          CType(item.Key, SomeRootVegetables).ToString("G"))
                End If
            Next
        Next
    End Sub
End Class
' The example displays the following output:
'    The following root vegetables are harvested in Summer:
'      HorseRadish
'    The following root vegetables are harvested in Autumn:
'      Turnip
'      HorseRadish
'    The following root vegetables are harvested in Winter:
'      HorseRadish
'    The following root vegetables are harvested in Spring:
'      Turnip
'      Radish
'      HorseRadish
```

接口

接口定义用于指定"可以执行"关系或"具有"关系的协定。接口通常用于实现某种功能，如比较和排序（IComparable 和 IComparable<T> 接口）、测试相等性（IEquatable<T> 接口）或枚举集合中的项（IEnumerable 和

IEnumerable<T> 接口)。接口可具有属性、方法和事件,所有这些都是抽象成员;也就是说,虽然接口定义这些成员及其签名,但每个接口成员的功能由实现该接口的类型定义。这意味着实现接口的任何类或结构都必须为该接口中声明的抽象成员提供定义。接口也可以要求任何实现类或结构实现一个或多个其他接口。

对接口有以下限制:

- 接口可以用任何可访问性来声明,但接口成员必须全都具有公共可访问性。
- 接口不能定义构造函数。
- 接口不能定义字段。
- 接口只能定义实例成员。它们不能定义静态成员。

每种语言都必须提供映射规则,将实现映射到需要该成员的接口,因为多个接口可以使用同一个签名声明成员,而这些成员可以分别具有单独的实现。

委托

委托是用途类似于 C++ 中的函数指针的引用类型。它们用于 .NET 中的事件处理程序和回调函数。与函数指针不同,委托是安全、可验证和类型安全的。委托类型可以表示任何具有兼容签名的实例方法或静态方法。

如果委托参数的类型的限制性强于方法参数的类型,则该委托的参数与该方法的相应参数兼容,因为这可保证传递给委托的参数可以安全地传递给方法。

同样,如果方法的返回类型的限制性强于委托的返回类型,则该委托的返回类型与该方法的返回类型兼容,因为这可保证方法的返回值可以安全地强制转换为委托的返回类型。

例如,具有类型为 IEnumerable 的参数和 Object 返回类型的委托可以表示具有类型为 Object 的参数和类型为 IEnumerable 的返回值的方法。有关更多信息及代码示例,请参见 Delegate.CreateDelegate(Type, Object, MethodInfo)。

委托一般绑定到它表示的方法。除了绑定到方法之外,委托还可以绑定到对象。该对象表示方法的第一个参数,每次调用委托时将该对象传递给方法。如果方法是实例方法,则将绑定的对象作为隐式 `this` 参数(在 Visual Basic 中为 `Me`)传递;如果方法为静态方法,则将该对象作为方法的第一个形参传递,并且委托签名必须匹配其余参数。有关更多信息及代码示例,请参见 System.Delegate。

所有委托从 System.MulticastDelegate(继承自 System.Delegate)继承。C#、Visual Basic 和 C++ 语言不允许从这些类型继承,而是提供了用于声明委托的关键字。

由于委托从 MulticastDelegate 继承,因此委托具有一个调用列表,其中列出了委托表示的方法,在调用委托时将执行该列表中的方法。列表中的所有方法接收调用委托时提供的自变量。

> **NOTE**
> 没有为在调用列表中包含多个方法的委托(即使委托具有返回类型)定义返回值。

在许多情况下(例如使用回调方法),一个委托只表示一个方法,而您需要做的就是创建委托并调用它。

对于表示多个方法的委托,.NET 提供了 Delegate 和 MulticastDelegate 委托类的方法,以支持如下操作:将方法添加到委托的调用列表(Delegate.Combine 方法)、移除方法(Delegate.Remove 方法)、获取调用列表(Delegate.GetInvocationList 方法)。

> **NOTE**
> 不需要将这些方法用于 C#、C++ 和 Visual Basic 中的事件处理程序委托,因为这些语言为添加和移除事件处理程序提供了语法。

类型定义

类型定义包括以下内容：

- 对该类型定义的任何特性。
- 类型的可访问性（可见性）。
- 类型的名称。
- 类型的基本类型。
- 该类型实现的任何接口。
- 每个类型的成员的定义。

特性

特性提供附加的用户定义元数据。它们通常用于在程序集中存储有关类型的附加信息，或在设计时或运行时环境中用于修改类型成员的行为。

特性本身是从 System.Attribute 继承的类。每种支持使用特性的语言都有自己的语法，用于将特性应用到某个语言元素。特性可应用到几乎任意语言元素；特性可以应用到的特定元素由应用到该特性类的 AttributeUsageAttribute 定义。

类型可访问性

所有类型都有一个修饰符，控制从其他类型对它们的可访问性。下表说明了运行时所支持的类型可访问性。

tttt	tt
public	所有程序集都可以访问此类型。
程序集 (assembly)	只能在其程序集内访问此类型。

嵌套类型的可访问性依赖于它的可访问域，该域是由已声明的成员可访问性和直接包含类型的可访问域这二者共同确定的。但是，嵌套类型的可访问域不能超出包含类型的可访问域。

在程序 `M` 的类型 `T` 中，声明的嵌套成员 `P` 的可访问域是按以下方法定义的（注意，`M` 本身也可能是一个类型）：

- 如果 `M` 的已声明可访问性为 `public`，则 `M` 的可访问域就是 `T` 的可访问域。
- 如果 `M` 的已声明可访问性为 `protected internal`，则 `M` 的可访问域就是 `T` 的可访问域与 `P` 的程序文本和从 `T` 派生的（在 `P` 之外声明的）任何类型的程序文本之间的交集。
- 如果 `M` 的已声明可访问性为 `protected`，则 `M` 的可访问域就是 `T` 的可访问域与 `T` 的程序文本和从 `T` 派生的任何类型的程序文本之间的交集。
- 如果 `M` 的已声明可访问性为 `internal`，则 `M` 的可访问域就是 `T` 的可访问域与 `P` 的程序文本之间的交集。
- 如果 `M` 的已声明可访问性为 `private`，则 `M` 的可访问域就是 `T` 的程序文本。

类型名称

常规类型系统对名称只有两种限制：

- 所有的名称都按 Unicode（16 位）字符串进行编码。
- 名称不允许有嵌入的（16 位）0x0000 值。

但是，大多数语言对类型名称都有附加限制。所有比较都是逐字节进行的，因此会区分大小写并与区域设置无

关。

尽管类型可能引用来自其他模块和程序集的类型，但类型在一个 .NET 模块内必须是完全定义的。（但是，根据编译器支持的情况，它可以分成多个源代码文件。）类型名称只需要在命名空间内唯一。要完全标识一个类型，其类型名称必须由包含此类型实现的命名空间加以限定。

基本类型和接口

一个类型可以从另一个类型继承值和行为。常规类型系统不允许类型从多个基本类型进行继承。

一个类型可以实现任何数量的接口。要实现接口，类型必须实现该接口的所有虚拟成员。虚方法可以由派生的类型来实现，既可静态调用，也可动态调用。

类型成员

运行时允许您定义指定类型行为和状态的类型成员。类型成员包括以下内容：

- 字段
- 属性
- 方法
- 构造函数
- 事件
- 嵌套类型

字段

字段描述并包含类型状态的一部分。字段可以是运行时支持的任何类型。字段通常是 `private` 或 `protected`，因此只能在类内部或从派生类访问。如果可以从类型外部修改字段值，通常使用属性集访问器。公开的字段通常是只读的，可以为以下两种类型：

- 常量，在设计时赋值。尽管没有使用 `static`（在 Visual Basic 中为 `Shared`）关键字定义，但它们都是类的静态成员。
- 只读变量，可以在类构造函数中赋值。

下面的示例演示只读字段的这两种用法。

```
using System;

public class Constants
{
   public const double Pi = 3.1416;
   public readonly DateTime BirthDate;

   public Constants(DateTime birthDate)
   {
      this.BirthDate = birthDate;
   }
}

public class Example
{
   public static void Main()
   {
      Constants con = new Constants(new DateTime(1974, 8, 18));
      Console.Write(Constants.Pi + "\n");
      Console.Write(con.BirthDate.ToString("d") + "\n");
   }
}
// The example displays the following output if run on a system whose current
// culture is en-US:
//    3.1416
//    8/18/1974
```

```
Public Class Constants
    Public Const Pi As Double = 3.1416
    Public ReadOnly BirthDate As Date

    Public Sub New(birthDate As Date)
        Me.BirthDate = birthDate
    End Sub
End Class

Public Module Example
    Public Sub Main()
        Dim con As New Constants(#8/18/1974#)
        Console.WriteLine(Constants.Pi.ToString())
        Console.WriteLine(con.BirthDate.ToString("d"))
    End Sub
End Module
' The example displays the following output if run on a system whose current
' culture is en-US:
'    3.1416
'    8/18/1974
```

属性

属性命名类型的值或状态，并定义获得或设置属性值的方法。属性可以是基元类型、基元类型的集合、用户定义的类型或用户定义类型的集合。属性通常用于使类型的公共接口独立于类型的实际表示形式。这使属性能够反映不直接在类中存储的值（例如，当属性返回计算值时）或能够在将值赋给私有字段前进行验证。下面的示例演示后一种模式。

```
using System;

public class Person
{
   private int m_Age;

   public int Age
   {
      get { return m_Age; }
      set {
         if (value < 0 || value > 125)
         {
            throw new ArgumentOutOfRangeException("The value of the Age property must be between 0 and
125.");
         }
         else
         {
            m_Age = value;
         }
      }
   }
}
```

```
Public Class Person
    Private m_Age As Integer

    Public Property Age As Integer
        Get
            Return m_Age
        End Get
        Set
            If value < 0 Or value > 125 Then
                Throw New ArgumentOutOfRangeException("The value of the Age property must be between 0 and
125.")
            Else
                m_Age = value
            End If
        End Set
    End Property
End Class
```

除了包含属性本身之外，包含可读属性的类型的 Microsoft 中间语言 (MSIL) 还包含 `get_` propertyname 方法，包含可写属性的类型的 MSIL 还包含 `set_` propertyname 方法。

方法

方法描述可用于类型的操作。方法的签名指定其所有参数和返回值可使用的类型。

尽管大多数方法都定义方法调用所需的参数的确切数目，但某些方法支持可变数目的参数。这些方法的最后声明的参数用 ParamArrayAttribute 特性标记。语言编译器通常会提供一个关键字（例如，在 C# 中为 `params`，在 Visual Basic 中为 `ParamArray`），使得不必显式使用 ParamArrayAttribute。

构造函数

构造函数是一种特殊类型的方法，可创建类或结构的新实例。像任何其他方法一样，构造函数可以包含参数，但是它不返回值（即它返回 `void`）。

如果类的源代码没有显式定义构造函数，则编译器包含一个无参数构造函数。但是，如果某个类的源代码只定义参数化的构造函数，则 Visual Basic 和 C# 编译器将不会生成无参数构造函数。

如果一个结构的源代码定义多个构造函数，则这些构造函数必须是参数化的；结构不能定义无参数构造函数，并且编译器不会为结构或其他值类型生成无参数构造函数。所有值类型都具有隐式无参数构造函数。此构造函数由公共语言运行时实现，并且将该结构的所有字段都初始化为其默认值。

事件

事件定义可以响应的事情，并定义订阅、取消订阅及引发事件的方法。事件通常用于通知其他类型的状态改变。有关详细信息，请参阅事件。

嵌套类型

嵌套类型是作为某其他类型的成员的类型。嵌套类型应与其包含类型紧密关联，并且不得用作通用类型。在声明类型使用和创建嵌套类型实例时，嵌套类型很有用，但不在公共成员中公开嵌套类型的使用。

嵌套类型可能会使有些开发人员感到困惑，因此除非有必要的理由，否则嵌套类型不应是公开可见的。在设计完善的库中，开发人员几乎不需要使用嵌套类型实例化对象或声明变量。

类型成员的特征

通用类型系统允许类型成员具有多种特征，但并不要求语言能支持所有这些特征。下表介绍了这些成员特征。

abstract	方法、属性和事件	类型不提供方法的实现。继承或实现抽象方法的类型必须提供方法的实现。只有当派生的类型本身是抽象类型的时候，情况例外。所有的抽象方法都是虚的。
private、family、assembly、family 和 assembly、family 或 assembly，或者 public	全部	定义成员的可访问性： private 只能在与成员相同的类型或在嵌套类型中访问。 family 在与成员相同的类型中和从它继承的派生类型访问。 程序集 只能在定义该类型的程序集中访问。 family 和 assembly 只能从同时具备族和程序集访问权的类型进行访问。 family 或 assembly 只能从具备族和程序集访问权的类型进行访问。 public 可从任何类型访问。
final	方法、属性和事件	虚方法不能在派生类型中被重写。
initialize-only	字段	该值只能被初始化，不能在初始化之后写入。
实例	字段、方法、属性和事件	如果成员未标记为 `static`（C# 和 C++）、`Shared` (Visual Basic)、`virtual`（C# 和 C++）或 `Overridable` (Visual Basic)，则它是一个实例成员（没有实例关键字）。内存中这些成员的副本数将会像使用它们的对象数一样多。

[illegible]	[illegible]	[illegible]
文本	字段	分配给该字段的值是一个内置值类型的固定值(在编译时已知)。文本字段有时指的是常数。
newslot 或 override	全部	定义成员如何与具有相同签名的继承成员进行交互: newslot 隐藏具有相同签名的继承成员。 override 替换继承的虚方法的定义。 默认为 newslot。
静态	字段、方法、属性和事件	成员属于定义它的类型,而不属于该类型的特定实例;即使不创建类型的实例,成员也会存在,并且它由该类型的所有实例共享。
virtual	方法、属性和事件	此方法可以由派生类型实现,并且既可静态调用,也可动态调用。如果使用动态调用,在运行时执行调用的实例类型(而不是编译时已知的类型)将确定调用方法的哪一种实现。若要静态调用虚方法,可能必须将变量强制转换为使用所需方法版本的类型。

重载

每个类型成员都有一个唯一的签名。方法签名由方法名称和一个参数列表(方法的参数的顺序和类型)组成。可以在一种类型内定义具有相同名称的多种方法,只要这些方法的签名不同。当定义两种或多种具有相同名称的方法时,就称作重载。例如,在 System.Char 中,重载了 IsDigit 方法。一个方法采用 Char。另一个方法采用 String 和 Int32。

> **NOTE**
> 返回类型不被视为方法签名的一部分。这意味着如果方法只是返回类型不同,就不能重载。

继承、重写和隐藏成员

派生类型继承其基类型的所有成员;也就是说,会在派生类型上定义这些成员,并供派生类型使用。继承成员的行为和质量可以通过以下两种方式来修改:

- 派生类型可通过使用相同的签名定义一个新成员,从而隐藏继承的成员。将先前的公共成员变成私有成员,或者为标记为 `final` 的继承方法定义新行为时,可以采取这种方法。
- 派生类型可以重写继承的虚方法。重写方法提供了对方法的一种新定义,将根据运行时的值的类型,而不是编译时已知的变量类型来调用方法。只有在虚拟方法未标记为 `final` 且新方法至少可以像虚拟方法一样进行访问的情况下,方法才能重写虚拟方法。

请参阅

- .NET API 浏览器
- 公共语言运行时
- .NET 中的类型转换

语言独立性和与语言无关的组件

2021/11/16 •

.NET 是独立的语言。这意味着，开发人员可使用面向 .NET 实现的多种语言之一(例如 C#、F# 和 Visual Basic)进行开发。可访问针对 .NET 实现开发的类库的类型和成员，而不必了解它们的初始编写语言，也不必遵循任何原始语言的约定。如果你是组件开发人员，无论组件采用哪种语言，均可由任何 .NET 应用访问。

> **NOTE**
> 本文的第一部分讨论如何创建独立于语言的组件(即以任何语言编写的应用均可以使用的组件)。还可以从用多种语言编写的源代码创建单个组件或应用;请参阅本文第二部分的跨语言互操作性。

若要与使用任何语言编写的其他对象完全交互，对象必须只向调用方公开那些所有语言共有的功能。此组通用功能由公共语言规范 (CLS) 定义，后者是一组适用于生成的程序集的规则。公共语言规范在 ECMA-335 标准:公共语言基础结构的第 I 部分的第 7 条至第 11 条中进行了定义。

如果你的组件符合公共语言规范，则保证其符合 CLS 并可通过支持 CLS 的任何编程语言编写的程序集中的代码对其进行访问。可以通过将 CLSCompliantAttribute 特性应用于源代码来确定自己的组件在编译时是否符合公共语言规范。有关详细信息，请参阅 CLSCompliantAttribute 特性。

CLS 遵从性规则

本节讨论用于创建符合 CLS 的组件的规则。有关规则的完整列表，请参阅 ECMA-335 标准:公共语言基础结构的第 I 部分的第 7 条至第 11 条中进行了定义。

> **NOTE**
> 公共语言规范讨论 CLS 遵从性的每个规则，因为它应用于使用者(以编程方式访问符合 CLS 的组件的开发人员)、框架(使用语言编译器创建符合 CLS 的库的开发人员)和扩展人员(创建可创建符合 CLS 的组件的语言编译器或代码分析器等工具的开发人员)。本文重点介绍适用于框架的规则。但请注意，一些适用于扩展程序的规则也适用于使用 Reflection.Emit 创建的程序集。

若要设计与语言无关的组件，只需将 CLS 符合性规则应用于组件的公共接口即可。您的私有实现不必符合规范。

> **IMPORTANT**
> CLS 遵从性的规则仅适用于组件的公共接口，而非其私有实现。

例如，Byte 之外的无符号整数不符合 CLS。由于以下示例中的 `Person` 类公开了 `Age` 类型的 UInt16 属性，因此以下代码显示编译器警告。

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private UInt16 personAge = 0;

   public UInt16 Age
   { get { return personAge; } }
}
// The attempt to compile the example displays the following compiler warning:
//    Public1.cs(10,18): warning CS3003: Type of 'Person.Age' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
    Private personAge As UInt16

    Public ReadOnly Property Age As UInt16
        Get
            Return personAge
        End Get
    End Property
End Class
' The attempt to compile the example displays the following compiler warning:
'    Public1.vb(9) : warning BC40027: Return type of function 'Age' is not CLS-compliant.
'
'       Public ReadOnly Property Age As UInt16
'                                ~~~
```

你可以通过将 `Age` 属性的类型从 UInt16 更改为 Int16，使 `Person` 类符合 CLS，即一个符合 CLS 的 16 位带符号整数。你无需更改私有 `personAge` 字段的类型。

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private Int16 personAge = 0;

   public Int16 Age
   { get { return personAge; } }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
    Private personAge As UInt16

    Public ReadOnly Property Age As Int16
        Get
            Return CType(personAge, Int16)
        End Get
    End Property
End Class
```

库的公共接口包括：

- 公共类的定义。

- 公共类中公共成员的定义，以及派生类可以访问的成员（即受保护的成员）的定义。
- 公共类中公共方法的参数和返回类型，以及派生类可以访问的方法的参数和返回类型。

下表中将列出 CLS 遵从性规则。规则的文本摘自 ECMA-335 标准：公共语言基础结构（版权所有 2012，Ecma International）。有关这些规则的详细信息，请参阅以下各节。

可访问性	成员可访问性	重写继承的方法时，可访问性应不会更改（重写一个继承自其他具有可访问性 `family-or-assembly` 的程序集的方法除外）。在此情况下，重写应具有可访问性 `family`。	10
可访问性	成员可访问性	类型和成员的可见性和可访问性应是这样的：只要任何成员是可见的且可访问的，则该成员签名中的类型应是可见且可访问的。例如，在其程序集外部可见的公共方法不应具有其类型仅在程序集内可见的参数。只要任何成员是可见且可访问的，则构成该成员签名中使用的实例化泛型类型的类型应是可见且可访问的。例如，一个在其程序集外部可见的成员签名中出现的实例化泛型类型不应具有其类型仅在程序集内可见的泛型实参。	12
数组	数组	数组应具有符合 CLS 的类型的元素，且数组的所有维度的下限应为零。各重载间只需区别：项是数组还是数组的元素类型。当重载基于两个或更多数组类型时，元素类型应为命名类型。	16
特性	特性	特性应是 System.Attribute 类型或从该类型继承的类型。	41
特性	特性	CLS 只允许自定义特性编码的子集。这些编码中仅应显示的类型（请参阅第 IV 部分）：System.Type、System.String、System.Char、System.Boolean、System.Byte、System.Int16、System.Int32、System.Int64、System.Single、System.Double 以及基于符合 CLS 的基整数类型的任何枚举类型。	34

特性	特性	CLS 不允许公开可见的所需修饰符（`modreq`，请参阅第 II 部分），但允许其不了解的可选修饰符（`modopt`，请参阅第 II 部分）。	35
构造函数	构造函数	在访问继承的实例数据之前，对象构造函数应调用其基类的某个实例构造函数。（这不适用于不需要构造函数的值类型。）	21
构造函数	构造函数	对象构造函数只能在创建对象的过程中调用，并且不应将对象初始化两次。	22
枚举	枚举	枚举的基础类型应是内置的 CLS 整数类型，字段的名称应为“value__”，并且应将该字段标记为 `RTSpecialName`。	7
枚举	枚举	有两种不同的枚举，二者由是否存在 System.FlagsAttribute（请参阅第 IV 部分库）自定义特性指示。一个表示命名的整数值；另一个表示命名的位标记（可合并以生成一个未命名的值）。`enum` 的值不仅限于指定的值。	8
枚举	枚举	枚举的文本静态字段需具有枚举本身的类型。	9
事件	事件	实现事件的方法将在元数据中标记为 `SpecialName`。	29
事件	事件	事件及其访问器的可访问性应相同。	30
事件	事件	事件的 `add` 和 `remove` 方法应同时存在或不存在。	31
事件	事件	事件的 `add` 和 `remove` 方法均应采用一个参数，该参数的类型定义了事件的类型，且应派生自 System.Delegate。	32
事件	事件	事件应遵循特定的命名模式。CLS 规则 29 中引用的 SpecialName 特性将在适当名称比较中被忽略，且应遵循标识符规则。	33

异常	异常	引发的对象应为 System.Exception 类型或从该类型继承的类型。但是，阻止其他类型的异常的传播无需符合 CLS 的方法。	40
常规	CLS 遵从性规则	CLS 规则仅适用于类型中在定义程序集之外可访问或可显示的部分。	1
常规	CLS 遵从性规则	不应将不符合 CLS 的类型的成员标记为符合 CLS。	2
泛型	泛型类型和成员	嵌套类型拥有的泛型形参的数目至少应与封闭类型的一样多。嵌套类型中的泛型参数在位置上与其封闭类型中的泛型参数对应。	42
泛型	泛型类型和成员	根据上面定义的规则，泛型类型的名称将对非嵌套类型上声明的或新引入到类型（如果嵌套）中的类型参数的数量进行编码。	43
泛型	泛型类型和成员	泛型类型应该重新声明足够多的约束，以确保对基类型或接口的任何约束能够满足泛型类型约束的需要。	44
泛型	泛型类型和成员	用作对泛型参数的约束的类型本身应符合 CLS。	45
泛型	泛型类型和成员	实例化泛型类型中的成员（包括嵌套类型）的可见性和可访问性应被认为限制在特定的实例化中，而不是限制在整个泛型类型声明中。作出以下假定：CLS 规则 12 的可见性和可访问性仍适用。	46
泛型	泛型类型和成员	对于每个抽象或虚拟泛型方法，应该有默认的具体（非抽象）实现	47
接口	接口	符合 CLS 的接口不需要不符合 CLS 的方法的定义即可实现这些方法。	18
接口	接口	符合 CLS 的接口不应定义静态方法，也不应定义字段。	19
成员	类型成员概述	全局静态字段和方法不符合 CLS。	36

成员	--	使用字段初始化元数据指定文本静态值。符合 CLS 的文本必须在类型与文本(或基本类型，如果文本为 `enum`)完全相同的字段初始化元数据中指定值。	13
成员	类型成员概述	vararg 约束不属于 CLS，并且 CLS 支持的唯一调用约定是标准托管调用约定。	15
命名约定	命名约定	程序集应遵守用于管理允许启用且包含在标识符中的字符集的 Unicode 标准 3.0 的技术报告 15 的附件 7(可通过 Unicode 范式在线获得)。标识符应是由 Unicode 范式 C 定义的规范格式。对于 CLS，如果两个标识符的小写映射(由不区分区域设置的 Unicode、一对一小写映射指定)相同，则它们也相同。也就是说，对于要在 CLS 下视为不同的两个标识符，它们应以大小写之外的差别进行区分。但是，若要重写继承的定义，CLI 需要对使用的原始声明进行准确编码。	4
重载	命名约定	在符合 CLS 的范围中引入的所有名称都应是明显独立的类型，除非名称完全相同且通过重载解析。也就是说，CTS 允许单个类型对方法和字段使用相同的名称，但 CLS 不允许。	5
重载	命名约定	即使 CTS 允许区分不同的签名，但字段和嵌套类型只能由标识符比较区分。(标识符比较后)具有相同名称的方法、属性和事件应在除返回类型不同之外还具有其他差异，CLS 规则 39 中指定的差异除外	6
重载	重载	只可重载属性和方法。	37
重载	重载	属性和方法仅可基于其参数的数目和类型进行重载，名为 `op_Implicit` 和 `op_Explicit` 的转换运算符除外，这两种转换运算符也可基于其返回类型进行重载。	38

[1]	[2]	[3]	[4]
重载	--	如果类型中声明的两种或更多符合 CLS 的方法都具有相同的名称，并且对于特定的类型实例化集而言，它们具有相同的参数和返回类型，则所有这些方法在这些类型实例化时都应在语义上保持等效。	48
属性	属性	实现属性的 getter 和 setter 方法的方法应在元数据中标记为 `SpecialName`。	24
属性	属性	属性的访问器必须同为静态、同为虚拟或同为实例。	26
属性	属性	属性的类型应该是 getter 的返回类型和 setter 的最后一个自变量的类型。属性参数的类型应是对应于 getter 的参数的类型和 setter 的除最后一个参数之外所有参数的类型。所有这些类型都应该符合 CLS，并且不应是托管指针（也就是说，不应该被引用传递）。	27
属性	属性	属性应遵循特定的命名模式。CLS 规则 24 中引用的 `SpecialName` 特性将在适当名称比较中被忽略，且应遵循标识符规则。属性应具有 getter 和/或 setter 方法。	28
类型转换	类型转换	如果提供 op_Implicit 或 op_Explicit，则必须提供实现强制转换的替代方法。	39
类型	类型和类型成员签名	装箱的值类型不符合 CLS。	3
类型	类型和类型成员签名	显示在签名中的所有类型应符合 CLS。构成实例化泛型类型的所有类型应符合 CLS。	11
类型	类型和类型成员签名	类型化的引用是不符合 CLS 的。	14
类型	类型和类型成员签名	非托管的指针类型不符合 CLS。	17
类型	类型和类型成员签名	符合 CLS 的类、值类型和接口不应要求实现不符合 CLS 的成员	20

类型	类型和类型成员签名	System.Object 符合 CLS。任何其他符合 CLS 的类应从符合 CLS 的类继承。	23

索引到各个子部分：

- 类型和类型成员签名
- 命名约定
- 类型转换
- 数组
- 接口
- 枚举
- 类型成员概述
- 成员可访问性
- 泛型类型和成员
- 构造函数
- 属性
- 事件
- 重载
- 异常
- 特性

类型和类型成员签名

System.Object 类型符合 CLS，是 .NET 类型系统中所有对象类型的基础类型。.NET 中的继承要么是隐式的（例如，String 类从 `Object` 类隐式继承），要么是显式的（例如，CultureNotFoundException 类从 ArgumentException 类显式继承，后者又从 Exception 类显式继承。对于要符合 CLS 的派生类型，其基本类型也必须符合 CLS。

下面的示例显示基本类型不符合 CLS 的派生类型。它定义使用无符号 32 位整数作为计数器的基本 `Counter` 类。由于类通过对无符号整数进行换行来提供计数器功能，因此类标记为不符合 CLS。因此，派生类 `NonZeroCounter` 也不符合 CLS。

```
using System;

[assembly: CLSCompliant(true)]

[CLSCompliant(false)]
public class Counter
{
   UInt32 ctr;

   public Counter()
   {
      ctr = 0;
   }

   protected Counter(UInt32 ctr)
   {
      this.ctr = ctr;
   }

   public override string ToString()
   {
      return String.Format("{0}). ", ctr);
   }

   public UInt32 Value
   {
      get { return ctr; }
   }

   public void Increment()
   {
      ctr += (uint) 1;
   }
}

public class NonZeroCounter : Counter
{
   public NonZeroCounter(int startIndex) : this((uint) startIndex)
   {
   }

   private NonZeroCounter(UInt32 startIndex) : base(startIndex)
   {
   }
}
// Compilation produces a compiler warning like the following:
//    Type3.cs(37,14): warning CS3009: 'NonZeroCounter': base type 'Counter' is not
//            CLS-compliant
//    Type3.cs(7,14): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> _
Public Class Counter
    Dim ctr As UInt32

    Public Sub New
        ctr = 0
    End Sub

    Protected Sub New(ctr As UInt32)
        ctr = ctr
    End Sub

    Public Overrides Function ToString() As String
        Return String.Format("{0}). ", ctr)
    End Function

    Public ReadOnly Property Value As UInt32
        Get
            Return ctr
        End Get
    End Property

    Public Sub Increment()
        ctr += CType(1, UInt32)
    End Sub
End Class

Public Class NonZeroCounter : Inherits Counter
    Public Sub New(startIndex As Integer)
        MyClass.New(CType(startIndex, UInt32))
    End Sub

    Private Sub New(startIndex As UInt32)
        MyBase.New(CType(startIndex, UInt32))
    End Sub
End Class
' Compilation produces a compiler warning like the following:
'    Type3.vb(34) : warning BC40026: 'NonZeroCounter' is not CLS-compliant
'    because it derives from 'Counter', which is not CLS-compliant.
'
'    Public Class NonZeroCounter : Inherits Counter
'                 ~~~~~~~~~~~~~~
```

成员签名中出现的所有类型（包括方法的返回类型或属性类型）必须符合 CLS。此外，对于泛型类型：

- 构成实例化泛型类型的所有类型必须符合 CLS。
- 所有用作针对泛型参数的约束的类型必须符合 CLS。

.NET 通用类型系统包括许多直接受公共语言运行时支持且专以程序集的元数据编码的内置类型。在这些内部类型中，下表中所列的类型都符合 CLS。

CLS	
Byte	8 位无符号整数
Int16	16 位带符号整数
Int32	32 位带符号整数

[CLS]	[
Int64	64 位带符号整数
Half	半精度浮点值
单精度	单精度浮点值
双精度	双精度浮点值
布尔值	true 或 false 值类型
Char	UTF 16 编码单元
小数	非浮点十进制数字
IntPtr	平台定义的大小的指针或句柄
字符串	零个、一个或多个 Char 对象的集合

下表中所列的内部类型不符合 CLS。

[[[[[	[	[CLS]]]]]
SByte	8 位带符号整数数据类型	Int16
UInt16	16 位无符号整数	Int32
UInt32	32 位无符号整数	Int64
UInt64	64 位无符号整数	Int64（可能溢出）、BigInteger 或 Double
UIntPtr	未签名的指针或句柄	IntPtr

.NET 类库或任何其他类库可能包含不符合 CLS 的其他类型，例如：

- 装箱的值类型。下面的 C# 示例创建一个具有名为 `int*` 的 `Value` 类型的公共属性的类。由于 `int*` 是一个装箱的值类型，因此编译器将其标记为不符合 CLS。

```
using System;

[assembly:CLSCompliant(true)]

public unsafe class TestClass
{
   private int* val;

   public TestClass(int number)
   {
      val = (int*) number;
   }

   public int* Value {
      get { return val; }
   }
}
// The compiler generates the following output when compiling this example:
//        warning CS3003: Type of 'TestClass.Value' is not CLS-compliant
```

- 类型化的引用是一个特殊构造，它包含一个对对象的引用和一个对类型的引用。类型化的引用由 TypedReference 类显示在 .NET 中。

如果类型不符合 CLS，则需对其应用 CLSCompliantAttribute 值为 `isCompliant` 的 `false` 特性。有关更多信息，请参阅 CLSCompliantAttribute 特性部分。

下面的示例说明了方法签名和泛型类型实例化中 CLS 遵从性的问题。它定义一个具有类型为 `InvoiceItem` 的属性和类型为 UInt32 的属性的 `Nullable<UInt32>` 类，以及一个具有类型为 UInt32 和 `Nullable<UInt32>` 的参数的构造函数。您尝试编译此示例时会收到四个编译器警告。

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<uint> qty;

   public InvoiceItem(uint sku, Nullable<uint> quantity)
   {
      itemId = sku;
      qty = quantity;
   }

   public Nullable<uint> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public uint InvoiceId
   {
      get { return invId; }
      set { invId = value; }
   }
}
// The attempt to compile the example displays the following output:
//    Type1.cs(13,23): warning CS3001: Argument type 'uint' is not CLS-compliant
//    Type1.cs(13,33): warning CS3001: Argument type 'uint?' is not CLS-compliant
//    Type1.cs(19,26): warning CS3003: Type of 'InvoiceItem.Quantity' is not CLS-compliant
//    Type1.cs(25,16): warning CS3003: Type of 'InvoiceItem.InvoiceId' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

    Private invId As UInteger = 0
    Private itemId As UInteger = 0
    Private qty AS Nullable(Of UInteger)

    Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
        itemId = sku
        qty = quantity
    End Sub

    Public Property Quantity As Nullable(Of UInteger)
        Get
            Return qty
        End Get
        Set
            qty = value
        End Set
    End Property

    Public Property InvoiceId As UInteger
        Get
            Return invId
        End Get
        Set
            invId = value
        End Set
    End Property
End Class
' The attempt to compile the example displays output similar to the following:
'    Type1.vb(13) : warning BC40028: Type of parameter 'sku' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                      ~~~
'    Type1.vb(13) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Sub New(sku As UInteger, quantity As Nullable(Of UInteger))
'                                                              ~~~~~~~~
'    Type1.vb(18) : warning BC40041: Type 'UInteger' is not CLS-compliant.
'
'       Public Property Quantity As Nullable(Of UInteger)
'                                               ~~~~~~~~
'    Type1.vb(27) : warning BC40027: Return type of function 'InvoiceId' is not CLS-compliant.
'
'       Public Property InvoiceId As UInteger
'                       ~~~~~~~~~~
```

若要消除编译器警告，请将 `InvoiceItem` 公共接口中不符合 CLS 的类型替换为符合 CLS 的类型：

```
using System;

[assembly: CLSCompliant(true)]

public class InvoiceItem
{
   private uint invId = 0;
   private uint itemId = 0;
   private Nullable<int> qty;

   public InvoiceItem(int sku, Nullable<int> quantity)
   {
      if (sku <= 0)
         throw new ArgumentOutOfRangeException("The item number is zero or negative.");
      itemId = (uint) sku;

      qty = quantity;
   }

   public Nullable<int> Quantity
   {
      get { return qty; }
      set { qty = value; }
   }

   public int InvoiceId
   {
      get { return (int) invId; }
      set {
         if (value <= 0)
            throw new ArgumentOutOfRangeException("The invoice number is zero or negative.");
         invId = (uint) value; }
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class InvoiceItem

    Private invId As UInteger = 0
    Private itemId As UInteger = 0
    Private qty AS Nullable(Of Integer)

    Public Sub New(sku As Integer, quantity As Nullable(Of Integer))
        If sku <= 0 Then
            Throw New ArgumentOutOfRangeException("The item number is zero or negative.")
        End If
        itemId = CUInt(sku)
        qty = quantity
    End Sub

    Public Property Quantity As Nullable(Of Integer)
        Get
            Return qty
        End Get
        Set
            qty = value
        End Set
    End Property

    Public Property InvoiceId As Integer
        Get
            Return CInt(invId)
        End Get
        Set
            invId = CUInt(value)
        End Set
    End Property
End Class
```

除了列出的特定类型之外，某些类型的类别也不符合 CLS。其中包括非托管指针类型和函数指针类型。下面的示例将生成一个编译器警告，因为它使用指向整数的指针来创建整数数组。

```
using System;

[assembly: CLSCompliant(true)]

public class ArrayHelper
{
   unsafe public static Array CreateInstance(Type type, int* ptr, int items)
   {
      Array arr = Array.CreateInstance(type, items);
      int* addr = ptr;
      for (int ctr = 0; ctr < items; ctr++) {
          int value = *addr;
          arr.SetValue(value, ctr);
          addr++;
      }
      return arr;
   }
}
// The attempt to compile this example displays the following output:
//    UnmanagedPtr1.cs(8,57): warning CS3001: Argument type 'int*' is not CLS-compliant
```

对于符合 CLS 的抽象类(即在 C# 中标记为 `abstract` 的累或在 Visual Basic 中标记为 `MustInherit` 的类)，该类的所有成员也必须符合 CLS。

命名约定

由于一些编程语言不区分大小写，因此标识符（例如，命名空间、类型和成员的名称）必须不仅限于大小写不同。如果两个标识符的小写映射是相同的，则这两个标识符被视为等效。下面的 C# 示例定义两个公共类：`Person` 和 `person`。由于它们只是大小写不同，因此 C# 编译器会将其标记为不符合 CLS。

```
using System;

[assembly: CLSCompliant(true)]

public class Person : person
{
}

public class person
{
}
// Compilation produces a compiler warning like the following:
//    Naming1.cs(11,14): warning CS3005: Identifier 'person' differing
//                       only in case is not CLS-compliant
//    Naming1.cs(6,14): (Location of symbol related to previous warning)
```

编程语言标识符（例如，命名空间、类型和成员的名称）必须遵照 Unicode 标准。这表示：

- 标识符的第一个字符可以是任何 Unicode 大写字母、小写字母、标题大小写字母、修饰符字母、其他字母或字母数字。有关 Unicode 字符类别的信息，请参阅 System.Globalization.UnicodeCategory 枚举。
- 后继字符可以是任何类别的第一个字符，还可以包含无间隔标记、间距组合标记、十进制数字、连接器标点符号以及格式设置代码。

在比较标识符之前，应筛选格式设置代码，并将标识符转换为 Unicode 范式 C，因为单个字符可由多个 UTF 16 编码的代码单位表示。在 Unicode 范式 C 中生成相同代码单位的字符序列不符合 CLS。下列示例定义了一个名为 `Å` 的属性（包含字符 ANGSTROM SIGN (U+212B)）和另一个名为 `Å` 的属性（包含字符 LATIN CAPITAL LETTER A WITH RING ABOVE (U+00C5)）。C# 和 Visual Basic 编译器都将源代码标记为不符合 CLS。

```
public class Size
{
   private double a1;
   private double a2;

   public double Å
   {
       get { return a1; }
       set { a1 = value; }
   }

   public double Å
   {
       get { return a2; }
       set { a2 = value; }
   }
}
// Compilation produces a compiler warning like the following:
//    Naming2a.cs(16,18): warning CS3005: Identifier 'Size.Å' differing only in case is not
//              CLS-compliant
//    Naming2a.cs(10,18): (Location of symbol related to previous warning)
//    Naming2a.cs(18,8): warning CS3005: Identifier 'Size.Å.get' differing only in case is not
//              CLS-compliant
//    Naming2a.cs(12,8): (Location of symbol related to previous warning)
//    Naming2a.cs(19,8): warning CS3005: Identifier 'Size.Å.set' differing only in case is not
//              CLS-compliant
//    Naming2a.cs(13,8): (Location of symbol related to previous warning)
```

```
<Assembly: CLSCompliant(True)>
Public Class Size
    Private a1 As Double
    Private a2 As Double

    Public Property Å As Double
        Get
            Return a1
        End Get
        Set
            a1 = value
        End Set
    End Property

    Public Property Å As Double
        Get
            Return a2
        End Get
        Set
            a2 = value
        End Set
    End Property
End Class
' Compilation produces a compiler warning like the following:
'    Naming1.vb(9) : error BC30269: 'Public Property Å As Double' has multiple definitions
'     with identical signatures.
'
'       Public Property Å As Double
'                       ~
```

特定范围内的成员名称(如程序集中的命名空间、命名空间中的类型或某类型中的成员)必须是唯一的，通过重载解析的名称除外。此要求比常规类型系统的要求更加严格，后者允许一个范围内的多个成员在作为不同类型的成员(例如，一个是方法，一个是字段时)时，可以具有相同的名称。具体而言，对于类型成员：

- 字段和嵌套类型只能通过名称区分。
- 具有相同名称的方法、属性和事件必须具有除返回类型不同之外的其他不同之处。

下面的示例说明了成员名称在其范围内必须是唯一的要求。它定义了一个名为 `Converter` 的类，该类包括四个名为 `Conversion` 的成员。其中三个是方法，一个是属性。包含 Int64 参数的方法具有唯一名称，但是，具有 Int32 参数的两个方法不是，因为返回值不被视为成员签名的一部分。由于属性不能具有与重载方法相同的名称，因此 `Conversion` 属性也违反了此要求。

```
using System;

[assembly: CLSCompliant(true)]

public class Converter
{
   public double Conversion(int number)
   {
      return (double) number;
   }

   public float Conversion(int number)
   {
      return (float) number;
   }

   public double Conversion(long number)
   {
      return (double) number;
   }

   public bool Conversion
   {
      get { return true; }
   }
}
// Compilation produces a compiler error like the following:
//    Naming3.cs(13,17): error CS0111: Type 'Converter' already defines a member called
//              'Conversion' with the same parameter types
//    Naming3.cs(8,18): (Location of symbol related to previous error)
//    Naming3.cs(23,16): error CS0102: The type 'Converter' already contains a definition for
//              'Conversion'
//    Naming3.cs(8,18): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Converter
    Public Function Conversion(number As Integer) As Double
        Return CDbl(number)
    End Function

    Public Function Conversion(number As Integer) As Single
        Return CSng(number)
    End Function

    Public Function Conversion(number As Long) As Double
        Return CDbl(number)
    End Function

    Public ReadOnly Property Conversion As Boolean
        Get
            Return True
        End Get
    End Property
End Class
' Compilation produces a compiler error like the following:
'    Naming3.vb(8) : error BC30301: 'Public Function Conversion(number As Integer) As Double'
'                    and 'Public Function Conversion(number As Integer) As Single' cannot
'                    overload each other because they differ only by return types.
'
'       Public Function Conversion(number As Integer) As Double
'                       ~~~~~~~~~~
'    Naming3.vb(20) : error BC30260: 'Conversion' is already declared as 'Public Function
'                     Conversion(number As Integer) As Single' in this class.
'
'       Public ReadOnly Property Conversion As Boolean
'                                ~~~~~~~~~~
```

各种语言都包含唯一关键字，因此面向公共语言运行时的语言还必须提供一些机制，以便引用与关键字相符的标识符（例如，类型名称）。例如，`case` 是 C# 和 Visual Basic 两者中的关键字。但是，下面的 Visual Basic 示例可以通过使用左大括号和右大括号将名为 `case` 的类从 `case` 关键字中消除。否则，该示例将生成错误消息“关键字无法用作标识符”，并且编译将失败。

```
Public Class [case]
    Private _id As Guid
    Private name As String

    Public Sub New(name As String)
        _id = Guid.NewGuid()
        Me.name = name
    End Sub

    Public ReadOnly Property ClientName As String
        Get
            Return name
        End Get
    End Property
End Class
```

下面的 C# 示例可以通过使用 `case` 符号从语言关键字中消除标识符来实例化 `@` 类。如果没有该符号，C# 编译器会显示两条错误消息：“应为类型”和“无效表达式术语‘case’”。

```
using System;

public class Example
{
   public static void Main()
   {
      @case c = new @case("John");
      Console.WriteLine(c.ClientName);
   }
}
```

类型转换

公共语言规范定义了两个转换运算符：

- `op_Implicit` 用于扩大转换，不会丢失数据或精度。例如，Decimal 结构包含一个已重载的 `op_Implicit` 运算符，以将整数类型值和 Char 值转换为 Decimal 值。
- `op_Explicit` 用于收缩可能导致丢失量级（将一个值转换为一个范围更小的值）或精度的转换。例如，Decimal 结构包含一个已重载的 `op_Explicit` 运算符，以将 Double 和 Single 值转换为 Decimal，并将 Decimal 值转换为整数值：Double、Single 和 Char。

但是，并非所有语言都支持运算符重载或定义自定义运算符。如果选择实现这些转换运算符，您还应提供另一种执行转换的方法。我们建议提供 `From`*Xxx* 和 `To`*Xxx* 方法。

下面的示例定义符合 CLS 的隐式和显式转换。它创建表示无符号的双精度浮点数的 `UDouble` 类。它提供从 `UDouble` 到 Double 的隐式转换和从 `UDouble` 到 Single、从 Double 到 `UDouble` 以及从 Single 到 `UDouble` 的显式转换。它还将 `ToDouble` 方法定义为隐式转换运算符的替代方法，并将 `ToSingle`、`FromDouble` 和 `FromSingle` 方法定义为显式转换运算符的替代方法。

```
using System;

public struct UDouble
{
   private double number;

   public UDouble(double value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public UDouble(float value)
   {
      if (value < 0)
         throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

      number = value;
   }

   public static readonly UDouble MinValue = (UDouble) 0.0;
   public static readonly UDouble MaxValue = (UDouble) Double.MaxValue;

   public static explicit operator Double(UDouble value)
   {
      return value.number;
   }

   public static implicit operator Single(UDouble value)
   {
      if (value.number > (double) Single.MaxValue)
         throw new InvalidCastException("A UDouble value is out of range of the Single type.");
```

```
            throw new InvalidCastException("A UDouble value is out of range of the Single type.");

         return (float) value.number;
      }

      public static explicit operator UDouble(double value)
      {
         if (value < 0)
            throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

         return new UDouble(value);
      }

      public static implicit operator UDouble(float value)
      {
         if (value < 0)
            throw new InvalidCastException("A negative value cannot be converted to a UDouble.");

         return new UDouble(value);
      }

      public static Double ToDouble(UDouble value)
      {
         return (Double) value;
      }

      public static float ToSingle(UDouble value)
      {
         return (float) value;
      }

      public static UDouble FromDouble(double value)
      {
         return new UDouble(value);
      }

      public static UDouble FromSingle(float value)
      {
         return new UDouble(value);
      }
   }
```

```
Public Structure UDouble
    Private number As Double

    Public Sub New(value As Double)
        If value < 0 Then
            Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
        End If
        number = value
    End Sub

    Public Sub New(value As Single)
        If value < 0 Then
            Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
        End If
        number = value
    End Sub

    Public Shared ReadOnly MinValue As UDouble = CType(0.0, UDouble)
    Public Shared ReadOnly MaxValue As UDouble = Double.MaxValue

    Public Shared Widening Operator CType(value As UDouble) As Double
        Return value.number
    End Operator

    Public Shared Narrowing Operator CType(value As UDouble) As Single
        If value.number > CDbl(Single.MaxValue) Then
            Throw New InvalidCastException("A UDouble value is out of range of the Single type.")
        End If
        Return CSng(value.number)
    End Operator

    Public Shared Narrowing Operator CType(value As Double) As UDouble
        If value < 0 Then
            Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
        End If
        Return New UDouble(value)
    End Operator

    Public Shared Narrowing Operator CType(value As Single) As UDouble
        If value < 0 Then
            Throw New InvalidCastException("A negative value cannot be converted to a UDouble.")
        End If
        Return New UDouble(value)
    End Operator

    Public Shared Function ToDouble(value As UDouble) As Double
        Return CType(value, Double)
    End Function

    Public Shared Function ToSingle(value As UDouble) As Single
        Return CType(value, Single)
    End Function

    Public Shared Function FromDouble(value As Double) As UDouble
        Return New UDouble(value)
    End Function

    Public Shared Function FromSingle(value As Single) As UDouble
        Return New UDouble(value)
    End Function
End Structure
```

数组

符合 CLS 的数组符合以下规则：

- 数组的所有维度必须具有零下限。下面的示例创建一个不符合 CLS 的数组，其下限为 1。请注意，无论是

否存在 CLSCompliantAttribute 特性，编译器都不检测由 `Numbers.GetTenPrimes` 方法返回的数组是否符合 CLS。

```
[assembly: CLSCompliant(true)]

public class Numbers
{
   public static Array GetTenPrimes()
   {
      Array arr = Array.CreateInstance(typeof(Int32), new int[] {10}, new int[] {1});
      arr.SetValue(1, 1);
      arr.SetValue(2, 2);
      arr.SetValue(3, 3);
      arr.SetValue(5, 4);
      arr.SetValue(7, 5);
      arr.SetValue(11, 6);
      arr.SetValue(13, 7);
      arr.SetValue(17, 8);
      arr.SetValue(19, 9);
      arr.SetValue(23, 10);

      return arr;
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
    Public Shared Function GetTenPrimes() As Array
        Dim arr As Array = Array.CreateInstance(GetType(Int32), {10}, {1})
        arr.SetValue(1, 1)
        arr.SetValue(2, 2)
        arr.SetValue(3, 3)
        arr.SetValue(5, 4)
        arr.SetValue(7, 5)
        arr.SetValue(11, 6)
        arr.SetValue(13, 7)
        arr.SetValue(17, 8)
        arr.SetValue(19, 9)
        arr.SetValue(23, 10)

        Return arr
    End Function
End Class
```

- 所有数组元素必须包括符合 CLS 的类型。下面的示例定义返回不符合 CLS 的数组的两种方法。第一个返回 UInt32 值的数组。第二个返回包括 Object 和 Int32 值的 UInt32 数组。虽然编译器因第一个数组是 UInt32 类型而将其标识为不合规，但无法识别第二个包含不符合 CLS 元素的数组。

```
using System;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static UInt32[] GetTenPrimes()
   {
      uint[] arr = { 1u, 2u, 3u, 5u, 7u, 11u, 13u, 17u, 19u };
      return arr;
   }

   public static Object[] GetFivePrimes()
   {
      Object[] arr = { 1, 2, 3, 5u, 7u };
      return arr;
   }
}
// Compilation produces a compiler warning like the following:
//    Array2.cs(8,27): warning CS3002: Return type of 'Numbers.GetTenPrimes()' is not
//            CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class Numbers
    Public Shared Function GetTenPrimes() As UInt32()
        Return {1ui, 2ui, 3ui, 5ui, 7ui, 11ui, 13ui, 17ui, 19ui}
    End Function

    Public Shared Function GetFivePrimes() As Object()
        Dim arr() As Object = {1, 2, 3, 5ui, 7ui}
        Return arr
    End Function
End Class
' Compilation produces a compiler warning like the following:
'    warning BC40027: Return type of function 'GetTenPrimes' is not CLS-compliant.
'
'       Public Shared Function GetTenPrimes() As UInt32()
'                              ~~~~~~~~~~~~
```

- 具有数组参数的方法的重载决策基于它们是否为数组及它们的元素类型。因此，以下对重载的 `GetSquares` 方法的定义符合 CLS。

```
using System;
using System.Numerics;

[assembly: CLSCompliant(true)]

public class Numbers
{
   public static byte[] GetSquares(byte[] numbers)
   {
      byte[] numbersOut = new byte[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++) {
         int square = ((int) numbers[ctr]) * ((int) numbers[ctr]);
         if (square <= Byte.MaxValue)
            numbersOut[ctr] = (byte) square;
         // If there's an overflow, assign MaxValue to the corresponding
         // element.
         else
            numbersOut[ctr] = Byte.MaxValue;
      }
      return numbersOut;
   }

   public static BigInteger[] GetSquares(BigInteger[] numbers)
   {
      BigInteger[] numbersOut = new BigInteger[numbers.Length];
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         numbersOut[ctr] = numbers[ctr] * numbers[ctr];

      return numbersOut;
   }
}
```

```
Imports System.Numerics

<Assembly: CLSCompliant(True)>

Public Module Numbers
    Public Function GetSquares(numbers As Byte()) As Byte()
        Dim numbersOut(numbers.Length - 1) As Byte
        For ctr As Integer = 0 To numbers.Length - 1
            Dim square As Integer = (CInt(numbers(ctr)) * CInt(numbers(ctr)))
            If square <= Byte.MaxValue Then
                numbersOut(ctr) = CByte(square)
                ' If there's an overflow, assign MaxValue to the corresponding
                ' element.
            Else
                numbersOut(ctr) = Byte.MaxValue
            End If
        Next
        Return numbersOut
    End Function

    Public Function GetSquares(numbers As BigInteger()) As BigInteger()
        Dim numbersOut(numbers.Length - 1) As BigInteger
        For ctr As Integer = 0 To numbers.Length - 1
            numbersOut(ctr) = numbers(ctr) * numbers(ctr)
        Next
        Return numbersOut
    End Function
End Module
```

接口

符合 CLS 的接口可以定义属性、事件和虚拟方法(没有实现的方法)。符合 CLS 的接口不能有:

- 静态方法或静态字段。如果您在接口中定义静态成员，那么 C# 和 Visual Basic 编译器将生成编译器错误。
- 字段。如果您在接口中定义字段，则 C# 和 Visual Basic 编译器将生成编译器错误。
- 不符合 CLS 的方法。例如，下面的接口定义包括方法 `INumber.GetUnsigned`，该方法标记为不符合 CLS。此示例生成编译器警告。

```
using System;

[assembly:CLSCompliant(true)]

public interface INumber
{
   int Length();
   [CLSCompliant(false)] ulong GetUnsigned();
}
// Attempting to compile the example displays output like the following:
//    Interface2.cs(8,32): warning CS3010: 'INumber.GetUnsigned()': CLS-compliant interfaces
//            must have only CLS-compliant members
```

```
<Assembly: CLSCompliant(True)>

Public Interface INumber
    Function Length As Integer

    <CLSCompliant(False)> Function GetUnsigned As ULong
End Interface
' Attempting to compile the example displays output like the following:
'    Interface2.vb(9) : warning BC40033: Non CLS-compliant 'function' is not allowed in a
'    CLS-compliant interface.
'
'       <CLSCompliant(False)> Function GetUnsigned As ULong
'                                      ~~~~~~~~~~~
```

  由于存在此规则，因此符合 CLS 的类型不需要实现不符合 CLS 的成员。如果一个符合 CLS 的框架公开实现不符合 CLS 接口的类，则其还应提供所有不符合 CLS 的成员的具体实现。

符合 CLS 的语言编译器还必须允许类提供在多个接口中具有相同名称和签名的成员的单独实现。C# 和 Visual Basic 都支持显式接口实现以提供同名方法的不同实现。Visual Basic 还支持 `Implements` 关键字，可让您显式指定特定成员要实现的接口和成员。下面的示例通过定义一个将 `Temperature` 和 `ICelsius` 接口作为显式接口实现的 `IFahrenheit` 类来说明此方案。

```
using System;

[assembly: CLSCompliant(true)]

public interface IFahrenheit
{
   decimal GetTemperature();
}

public interface ICelsius
{
   decimal GetTemperature();
}

public class Temperature : ICelsius, IFahrenheit
{
   private decimal _value;

   public Temperature(decimal value)
   {
      // We assume that this is the Celsius value.
      _value = value;
   }

   decimal IFahrenheit.GetTemperature()
   {
      return _value * 9 / 5 + 32;
   }

   decimal ICelsius.GetTemperature()
   {
      return _value;
   }
}
public class Example
{
   public static void Main()
   {
      Temperature temp = new Temperature(100.0m);
      ICelsius cTemp = temp;
      IFahrenheit fTemp = temp;
      Console.WriteLine("Temperature in Celsius: {0} degrees",
                        cTemp.GetTemperature());
      Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                        fTemp.GetTemperature());
   }
}
// The example displays the following output:
//       Temperature in Celsius: 100.0 degrees
//       Temperature in Fahrenheit: 212.0 degrees
```

```
<Assembly: CLSCompliant(True)>

Public Interface IFahrenheit
    Function GetTemperature() As Decimal
End Interface

Public Interface ICelsius
    Function GetTemperature() As Decimal
End Interface

Public Class Temperature : Implements ICelsius, IFahrenheit
    Private _value As Decimal

    Public Sub New(value As Decimal)
        ' We assume that this is the Celsius value.
        _value = value
    End Sub

    Public Function GetFahrenheit() As Decimal _
           Implements IFahrenheit.GetTemperature
        Return _value * 9 / 5 + 32
    End Function

    Public Function GetCelsius() As Decimal _
           Implements ICelsius.GetTemperature
        Return _value
    End Function
End Class

Module Example
    Public Sub Main()
        Dim temp As New Temperature(100.0d)
        Console.WriteLine("Temperature in Celsius: {0} degrees",
                          temp.GetCelsius())
        Console.WriteLine("Temperature in Fahrenheit: {0} degrees",
                          temp.GetFahrenheit())
    End Sub
End Module
' The example displays the following output:
'       Temperature in Celsius: 100.0 degrees
'       Temperature in Fahrenheit: 212.0 degrees
```

枚举

符合 CLS 的枚举必须遵循下列规则：

- 枚举的基础类型必须是符合 CLS 的内部整数（Byte、Int16、Int32 或 Int64）。例如，下面的代码尝试定义基础类型为 UInt32 的枚举并生成编译器警告。

```
using System;

[assembly: CLSCompliant(true)]

public enum Size : uint {
   Unspecified = 0,
   XSmall = 1,
   Small = 2,
   Medium = 3,
   Large = 4,
   XLarge = 5
};

public class Clothing
{
   public string Name;
   public string Type;
   public string Size;
}
// The attempt to compile the example displays a compiler warning like the following:
//    Enum3.cs(6,13): warning CS3009: 'Size': base type 'uint' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Enum Size As UInt32
    Unspecified = 0
    XSmall = 1
    Small = 2
    Medium = 3
    Large = 4
    XLarge = 5
End Enum

Public Class Clothing
    Public Name As String
    Public Type As String
    Public Size As Size
End Class
' The attempt to compile the example displays a compiler warning like the following:
'    Enum3.vb(6) : warning BC40032: Underlying type 'UInt32' of Enum is not CLS-compliant.
'
'    Public Enum Size As UInt32
'                ~~~~
```

- 枚举类型必须具有名为 `value__` 且标有 FieldAttributes.RTSpecialName 特性的单个实例字段。这使得您可以隐式引用字段值。
- 枚举包括文本静态字段，该字段的类型与枚举本身的类型匹配。例如，如果您用 `State` 和 `State.On` 的值定义 `State.Off` 枚举，则 `State.On` 和 `State.Off` 都是文本静态字段，其类型为 `State`。
- 有两种枚举：
  - 一种表示一组互斥的命名整数值的枚举。这种类型的枚举由缺少 System.FlagsAttribute 自定义特性表示。
  - 一种表示可结合用来生成未命名值的一组位标志的枚举。这种类型的枚举由存在 System.FlagsAttribute 自定义特性表示。

  有关详细信息，请参阅 Enum 结构的文档。
- 枚举的值不限于其指定值的范围。换言之，枚举中的值的范围是其基础值的范围。您可以使用 Enum.IsDefined 方法来确定指定的值是否为枚举成员。

类型成员概述

公共语言规范要求将所有字段和方法作为特定类的成员进行访问。因此，全局静态字段和方法（即，除类型外定义的静态字段或方法）不符合 CLS。如果您尝试在您的源代码中包括全局字段或方法，则 C# 和 Visual Basic 编译器都会生成编译器错误。

公共语言规范仅支持标准托管调用约定。它不支持非托管调用约定和带使用 `varargs` 关键字标记的变量参数列表的方法。对于符合标准托管调用约定的变量自变量列表，请使用 ParamArrayAttribute 特性或单个语言的实现，如 C# 中的 `params` 关键字和 Visual Basic 中的 `ParamArray` 关键字。

成员可访问性

重写继承成员不能更改该成员的可访问性。例如，无法在派生类中通过私有方法重写基类中的公共方法。有一个例外：由其他程序集中的类型重写的程序集中的 `protected internal`（在 C# 中）或 `Protected Friend`（在 Visual Basic 中）成员。在该示例中，重写的可访问性是 `Protected`。

下面的示例说明了当 CLSCompliantAttribute 特性设置为 `true`，并且 `Human`（它是派生自 `Animal` 的类）尝试将 `Species` 属性的可访问性从公开更改为私有时生成的错误。如果该示例的可访问性更改为公共，则其编译成功。

```
using System;

[assembly: CLSCompliant(true)]

public class Animal
{
   private string _species;

   public Animal(string species)
   {
      _species = species;
   }

   public virtual string Species
   {
      get { return _species; }
   }

   public override string ToString()
   {
      return _species;
   }
}

public class Human : Animal
{
   private string _name;

   public Human(string name) : base("Homo Sapiens")
   {
      _name = name;
   }

   public string Name
   {
      get { return _name; }
   }

   private override string Species
   {
      get { return base.Species; }
   }

   public override string ToString()
   {
      return _name;
   }
}

public class Example
{
   public static void Main()
   {
      Human p = new Human("John");
      Console.WriteLine(p.Species);
      Console.WriteLine(p.ToString());
   }
}
// The example displays the following output:
//    error CS0621: 'Human.Species': virtual or abstract members cannot be private
```

```
<Assembly: CLSCompliant(True)>

Public Class Animal
    Private _species As String

    Public Sub New(species As String)
        _species = species
    End Sub

    Public Overridable ReadOnly Property Species As String
        Get
            Return _species
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return _species
    End Function
End Class

Public Class Human : Inherits Animal
    Private _name As String

    Public Sub New(name As String)
        MyBase.New("Homo Sapiens")
        _name = name
    End Sub

    Public ReadOnly Property Name As String
        Get
            Return _name
        End Get
    End Property

    Private Overrides ReadOnly Property Species As String
        Get
            Return MyBase.Species
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return _name
    End Function
End Class

Public Module Example
    Public Sub Main()
        Dim p As New Human("John")
        Console.WriteLine(p.Species)
        Console.WriteLine(p.ToString())
    End Sub
End Module
' The example displays the following output:
'     'Private Overrides ReadOnly Property Species As String' cannot override
'     'Public Overridable ReadOnly Property Species As String' because
'      they have different access levels.
'
'         Private Overrides ReadOnly Property Species As String
```

如果某个成员是可访问的，则该成员签名中的类型必须是可访问的。例如，这意味着公共成员不能包含类型是私有的、受保护的或内部的参数。下面的示例说明了当 `StringWrapper` 类构造函数公开一个用于确定如何包装字符串值的内部 `StringOperationType` 枚举值时出现的编译器错误。

```
using System;
using System.Text;

public class StringWrapper
{
   string internalString;
   StringBuilder internalSB = null;
   bool useSB = false;

   public StringWrapper(StringOperationType type)
   {
      if (type == StringOperationType.Normal) {
         useSB = false;
      }
      else {
         useSB = true;
         internalSB = new StringBuilder();
      }
   }

   // The remaining source code...
}

internal enum StringOperationType { Normal, Dynamic }
// The attempt to compile the example displays the following output:
//    error CS0051: Inconsistent accessibility: parameter type
//            'StringOperationType' is less accessible than method
//            'StringWrapper.StringWrapper(StringOperationType)'
```

```
Imports System.Text

<Assembly: CLSCompliant(True)>

Public Class StringWrapper

    Dim internalString As String
    Dim internalSB As StringBuilder = Nothing
    Dim useSB As Boolean = False

    Public Sub New(type As StringOperationType)
        If type = StringOperationType.Normal Then
            useSB = False
        Else
            internalSB = New StringBuilder()
            useSB = True
        End If
    End Sub

    ' The remaining source code...
End Class

Friend Enum StringOperationType As Integer
    Normal = 0
    Dynamic = 1
End Enum
' The attempt to compile the example displays the following output:
'    error BC30909: 'type' cannot expose type 'StringOperationType'
'     outside the project through class 'StringWrapper'.
'
'       Public Sub New(type As StringOperationType)
'                                ~~~~~~~~~~~~~~~~~~~
```

泛型类型和成员

嵌套类型拥有的泛型参数数目总是至少与封闭类型的一样多。它们按位置对应于封闭类型中的泛型参数。泛型

类型还可以包括新的泛型参数。

一个包含类型及其嵌套类型的泛型类型参数之间的关系可能由各种语言的语法隐藏。在下面的示例中，泛型类型 `Outer<T>` 包含两个嵌套的类：`Inner1A` 和 `Inner1B<U>`。对于每个类从 `ToString` 继承的 Object.ToString() 方法的调用，表示每个嵌套的类包括其包含的类的类型参数。

```
using System;

[assembly:CLSCompliant(true)]

public class Outer<T>
{
   T value;

   public Outer(T value)
   {
      this.value = value;
   }

   public class Inner1A : Outer<T>
   {
      public Inner1A(T value) : base(value)
      {  }
   }

   public class Inner1B<U> : Outer<T>
   {
      U value2;

      public Inner1B(T value1, U value2) : base(value1)
      {
         this.value2 = value2;
      }
   }
}

public class Example
{
   public static void Main()
   {
      var inst1 = new Outer<String>("This");
      Console.WriteLine(inst1);

      var inst2 = new Outer<String>.Inner1A("Another");
      Console.WriteLine(inst2);

      var inst3 = new Outer<String>.Inner1B<int>("That", 2);
      Console.WriteLine(inst3);
   }
}
// The example displays the following output:
//       Outer`1[System.String]
//       Outer`1+Inner1A[System.String]
//       Outer`1+Inner1B`1[System.String,System.Int32]
```

```
<Assembly: CLSCompliant(True)>

Public Class Outer(Of T)
    Dim value As T

    Public Sub New(value As T)
        Me.value = value
    End Sub

    Public Class Inner1A : Inherits Outer(Of T)
        Public Sub New(value As T)
            MyBase.New(value)
        End Sub
    End Class

    Public Class Inner1B(Of U) : Inherits Outer(Of T)
        Dim value2 As U

        Public Sub New(value1 As T, value2 As U)
            MyBase.New(value1)
            Me.value2 = value2
        End Sub
    End Class
End Class

Public Module Example
    Public Sub Main()
        Dim inst1 As New Outer(Of String)("This")
        Console.WriteLine(inst1)

        Dim inst2 As New Outer(Of String).Inner1A("Another")
        Console.WriteLine(inst2)

        Dim inst3 As New Outer(Of String).Inner1B(Of Integer)("That", 2)
        Console.WriteLine(inst3)
    End Sub
End Module
' The example displays the following output:
'       Outer`1[System.String]
'       Outer`1+Inner1A[System.String]
'       Outer`1+Inner1B`1[System.String,System.Int32]
```

泛型类型名称采用 name`n 格式进行编码，其中 name 是类型名称，` 是字符文本，而 n 是针对类型声明的参数数目，或对于嵌套泛型类型为最近引入的类型参数的数目 。此泛型类型名称的编码主要对使用反射来访问库中符合 CLS 的泛型类型的开发人员很有用。

如果将约束应用于泛型类型，则任何用作约束的类型也必须符合 CLS。下面的示例定义一个名为 `BaseClass` 的不符合 CLS 的类和一个其类型参数必须派生自 `BaseCollection` 的名为 `BaseClass` 的泛型类。但由于 `BaseClass` 不符合 CLS，因此编译器会发出警告。

```
using System;

[assembly:CLSCompliant(true)]

[CLSCompliant(false)] public class BaseClass
{}

public class BaseCollection<T> where T : BaseClass
{}
// Attempting to compile the example displays the following output:
//    warning CS3024: Constraint type 'BaseClass' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

<CLSCompliant(False)> Public Class BaseClass
End Class


Public Class BaseCollection(Of T As BaseClass)
End Class
' Attempting to compile the example displays the following output:
'    warning BC40040: Generic parameter constraint type 'BaseClass' is not
'    CLS-compliant.
'
'    Public Class BaseCollection(Of T As BaseClass)
'                                       ~~~~~~~~
```

如果泛型类型派生自泛型基本类型，则其必须重新声明所有约束，以确保也满足对基本类型的约束。下面的示例定义可表示任何数值类型的 `Number<T>`。它还定义表示浮点值的 `FloatingPoint<T>` 类。但是，源代码无法编译，因为它未将 `Number<T>` 上（T 必须是值类型）的约束应用于 `FloatingPoint<T>`。

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T>
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
// The attempt to comple the example displays the following output:
//       error CS0453: The type 'T' must be a non-nullable value type in
//               order to use it as parameter 'T' in the generic type or method 'Number<T>'
```

```
<Assembly: CLSCompliant(True)>

Public Class Number(Of T As Structure)
    ' Use Double as the underlying type, since its range is a superset of
    ' the ranges of all numeric types except BigInteger.
    Protected number As Double

    Public Sub New(value As T)
        Try
            Me.number = Convert.ToDouble(value)
        Catch e As OverflowException
            Throw New ArgumentException("value is too large.", e)
        Catch e As InvalidCastException
            Throw New ArgumentException("The value parameter is not numeric.", e)
        End Try
    End Sub

    Public Function Add(value As T) As T
        Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
    End Function

    Public Function Subtract(value As T) As T
        Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
    End Function
End Class

Public Class FloatingPoint(Of T) : Inherits Number(Of T)
    Public Sub New(number As T)
        MyBase.New(number)
        If TypeOf number Is Single Or
                 TypeOf number Is Double Or
                 TypeOf number Is Decimal Then
            Me.number = Convert.ToDouble(number)
        Else
            throw new ArgumentException("The number parameter is not a floating-point number.")
        End If
    End Sub
End Class
' The attempt to comple the example displays the following output:
'    error BC32105: Type argument 'T' does not satisfy the 'Structure'
'    constraint for type parameter 'T'.
'
'    Public Class FloatingPoint(Of T) : Inherits Number(Of T)
'                                                    ~
```

如果将此约束添加到 `FloatingPoint<T>` 类**中**，则该**示例成功**编译。

```
using System;

[assembly:CLSCompliant(true)]

public class Number<T> where T : struct
{
   // use Double as the underlying type, since its range is a superset of
   // the ranges of all numeric types except BigInteger.
   protected double number;

   public Number(T value)
   {
      try {
         this.number = Convert.ToDouble(value);
      }
      catch (OverflowException e) {
         throw new ArgumentException("value is too large.", e);
      }
      catch (InvalidCastException e) {
         throw new ArgumentException("The value parameter is not numeric.", e);
      }
   }

   public T Add(T value)
   {
      return (T) Convert.ChangeType(number + Convert.ToDouble(value), typeof(T));
   }

   public T Subtract(T value)
   {
      return (T) Convert.ChangeType(number - Convert.ToDouble(value), typeof(T));
   }
}

public class FloatingPoint<T> : Number<T> where T : struct
{
   public FloatingPoint(T number) : base(number)
   {
      if (typeof(float) == number.GetType() ||
          typeof(double) == number.GetType() ||
          typeof(decimal) == number.GetType())
         this.number = Convert.ToDouble(number);
      else
         throw new ArgumentException("The number parameter is not a floating-point number.");
   }
}
```

```
<Assembly: CLSCompliant(True)>

Public Class Number(Of T As Structure)
    ' Use Double as the underlying type, since its range is a superset of
    ' the ranges of all numeric types except BigInteger.
    Protected number As Double

    Public Sub New(value As T)
        Try
            Me.number = Convert.ToDouble(value)
        Catch e As OverflowException
            Throw New ArgumentException("value is too large.", e)
        Catch e As InvalidCastException
            Throw New ArgumentException("The value parameter is not numeric.", e)
        End Try
    End Sub

    Public Function Add(value As T) As T
        Return CType(Convert.ChangeType(number + Convert.ToDouble(value), GetType(T)), T)
    End Function

    Public Function Subtract(value As T) As T
        Return CType(Convert.ChangeType(number - Convert.ToDouble(value), GetType(T)), T)
    End Function
End Class

Public Class FloatingPoint(Of T As Structure) : Inherits Number(Of T)
    Public Sub New(number As T)
        MyBase.New(number)
        If TypeOf number Is Single Or
                 TypeOf number Is Double Or
                 TypeOf number Is Decimal Then
            Me.number = Convert.ToDouble(number)
        Else
            throw new ArgumentException("The number parameter is not a floating-point number.")
        End If
    End Sub
End Class
```

公共语言规范对嵌套类型和受保护成员规定了一个保守的按实例化模型。开放式泛型类型不能公开具有包含嵌套的、受保护的泛型类型的特定实例化的签名的字段或成员。扩大泛型基类或接口的特定实例化的非泛型类型不能公开带签名的字段或成员，此类字段或成员包含嵌套的、受保护的泛型类型的不同实例化。

下面的示例定义一个泛型类型 `C1<T>`（或 Visual Basic 中的 `C1(Of T)`）和一个受保护的类 `C1<T>.N`（或 Visual Basic 中的 `C1(Of T).N`）。`C1<T>` 有两个方法：`M1` 和 `M2`。但是，`M1` 并不符合 CLS，因为它试图从 C1<T>（或 `C1(Of T)`）返回一个 `C1<int>.N`（或 `C1(Of Integer).N`）对象。另一个名为 `C2` 的类派生自 `C1<long>`（或 `C1(Of Long)`）。它有两个方法：`M3` 和 `M4`。`M3` 不符合 CLS，因为它尝试从 `C1<int>.N` 的子类中返回 `C1(Of Integer).N`（或 `C1<long>`）对象。语言编译器可能具有更高限制。在此示例中，Visual Basic 在尝试编译 `M4` 时显示错误。

```
using System;

[assembly:CLSCompliant(true)]

public class C1<T>
{
   protected class N { }

   protected void M1(C1<int>.N n) { } // Not CLS-compliant - C1<int>.N not
                                     // accessible from within C1<T> in all
                                     // languages
   protected void M2(C1<T>.N n) { }   // CLS-compliant – C1<T>.N accessible
                                     // inside C1<T>
}

public class C2 : C1<long>
{
   protected void M3(C1<int>.N n) { }  // Not CLS-compliant – C1<int>.N is not
                                       // accessible in C2 (extends C1<long>)

   protected void M4(C1<long>.N n) { } // CLS-compliant, C1<long>.N is
                                       // accessible in C2 (extends C1<long>)
}
// Attempting to compile the example displays output like the following:
//       Generics4.cs(9,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
//       Generics4.cs(18,22): warning CS3001: Argument type 'C1<int>.N' is not CLS-compliant
```

```
<Assembly: CLSCompliant(True)>

Public Class C1(Of T)
    Protected Class N
    End Class

    Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
        ' accessible from within C1(Of T) in all
    End Sub                                   ' languages


    Protected Sub M2(n As C1(Of T).N)     ' CLS-compliant – C1(Of T).N accessible
    End Sub                               ' inside C1(Of T)
End Class

Public Class C2 : Inherits C1(Of Long)
    Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant – C1(Of Integer).N is not
    End Sub                                   ' accessible in C2 (extends C1(Of Long))

    Protected Sub M4(n As C1(Of Long).N)
    End Sub
End Class
' Attempting to compile the example displays output like the following:
'    error BC30508: 'n' cannot expose type 'C1(Of Integer).N' in namespace
'    '<Default>' through class 'C1'.
'
'       Protected Sub M1(n As C1(Of Integer).N)   ' Not CLS-compliant - C1<int>.N not
'                             ~~~~~~~~~~~~~~~~~
'    error BC30389: 'C1(Of T).N' is not accessible in this context because
'    it is 'Protected'.
'
'       Protected Sub M3(n As C1(Of Integer).N)   ' Not CLS-compliant - C1(Of Integer).N is not
'
'                             ~~~~~~~~~~~~~~~~~
'
'    error BC30389: 'C1(Of T).N' is not accessible in this context because it is 'Protected'.
'
'       Protected Sub M4(n As C1(Of Long).N)
'                             ~~~~~~~~~~~~~
```

构造函数

符合 CLS 的类和结构中的构造函数必须遵循下列规则：

- 派生类的构造函数必须先调用其基类的实例构造函数，然后才能访问继承的实例数据。存在此要求是因为基类构造函数并不由它们的派生类继承。此规则不适用于不支持直接继承的结构。

  通常，编译器独立地强制实施 CLS 遵从性的此规则，如下面的示例所示。它创建派生自 `Doctor` 类的 `Person` 类，但 `Doctor` 类无法调用 `Person` 类构造函数来初始化继承的实例字段。

```
using System;

[assembly: CLSCompliant(true)]

public class Person
{
   private string fName, lName, _id;

   public Person(string firstName, string lastName, string id)
   {
      if (String.IsNullOrEmpty(firstName + lastName))
         throw new ArgumentNullException("Either a first name or a last name must be provided.");

      fName = firstName;
      lName = lastName;
      _id = id;
   }

   public string FirstName
   {
      get { return fName; }
   }

   public string LastName
   {
      get { return lName; }
   }

   public string Id
   {
      get { return _id; }
   }

   public override string ToString()
   {
      return String.Format("{0}{1}{2}", fName,
                           String.IsNullOrEmpty(fName) ?  "" : " ",
                           lName);
   }
}

public class Doctor : Person
{
   public Doctor(string firstName, string lastName, string id)
   {
   }

   public override string ToString()
   {
      return "Dr. " + base.ToString();
   }
}
// Attempting to compile the example displays output like the following:
//    ctor1.cs(45,11): error CS1729: 'Person' does not contain a constructor that takes 0
//            arguments
//    ctor1.cs(10,11): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

Public Class Person
    Private fName, lName, _id As String

    Public Sub New(firstName As String, lastName As String, id As String)
        If String.IsNullOrEmpty(firstName + lastName) Then
            Throw New ArgumentNullException("Either a first name or a last name must be provided.")
        End If

        fName = firstName
        lName = lastName
        _id = id
    End Sub

    Public ReadOnly Property FirstName As String
        Get
            Return fName
        End Get
    End Property

    Public ReadOnly Property LastName As String
        Get
            Return lName
        End Get
    End Property

    Public ReadOnly Property Id As String
        Get
            Return _id
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return String.Format("{0}{1}{2}", fName,
                             If(String.IsNullOrEmpty(fName), "", " "),
                             lName)
    End Function
End Class

Public Class Doctor : Inherits Person
    Public Sub New(firstName As String, lastName As String, id As String)
    End Sub

    Public Overrides Function ToString() As String
        Return "Dr. " + MyBase.ToString()
    End Function
End Class
' Attempting to compile the example displays output like the following:
'    Ctor1.vb(46) : error BC30148: First statement of this 'Sub New' must be a call
'    to 'MyBase.New' or 'MyClass.New' because base class 'Person' of 'Doctor' does
'    not have an accessible 'Sub New' that can be called with no arguments.
'
'       Public Sub New()
'                  ~~~
```

- 只有在创建对象时才能调用对象构造函数。此外，不能将一个对象初始化两次。例如，这意味着 Object.MemberwiseClone 和反序列化方法（如 BinaryFormatter.Deserialize）不得调用构造函数。

属性

符合 CLS 的类型的属性必须遵循下列规则：

- 属性必须具有 setter 和/或 getter。在程序集中，这些作为特殊方法实现，这意味着它们将显示为单独的方法（getter 命名为 `get_`*propertyname*，setter 命名为 `set_`*propertyname*），且在程序集元数据中标记为 `SpecialName`。C# 和 Visual Basic 编译器会自动执行此规则，而无需应用 CLSCompliantAttribute 特性。

- 属性的类型是属性 getter 的返回类型和 setter 的最后一个自变量。这些类型必须符合 CLS，并且不能通过引用将参数分配到属性中(即它们不能为托管指针)。
- 如果属性包含 getter 和 setter 两者，则它们必须都是虚拟的、静态的或实例。C# 和 Visual Basic 编译器通过它们的属性定义语法自动执行此规则。

事件

事件由其名称和类型定义。事件类型是用于指示事件的委托。例如，AppDomain.AssemblyResolve 事件的类型为 ResolveEventHandler。除事件本身外，带有基于事件名称的名称的三种方法提供事件的实现并在程序集的元数据中标记为 `SpecialName`：

- 用于添加事件处理程序的名为 `add_`*EventName* 的方法。例如，AppDomain.AssemblyResolve 事件的事件订阅方法名为 `add_AssemblyResolve`。
- 用于移除事件处理程序的名为 `remove_`*EventName* 的方法。例如，AppDomain.AssemblyResolve 事件的移除方法名为 `remove_AssemblyResolve`。
- 用于指示事件已发生的名为 `raise_`*EventName* 的方法。

> **NOTE**
> 大多数关于事件的公共语言规范的规则都通过语言编译器实施，且对组件开发人员是透明的。

用于添加、移除和引发事件的方法必须拥有相同的可访问性。它们还必须都为静态、实例或虚拟的。用于添加和移除事件的方法具有一个类型为事件委托类型的参数。添加和移除方法必须同时存在或同时不存在。

如果两个读取之间的温度更改等于或超过阈值，则下面的示例将定义一个名为 `Temperature` 的类，该类会引发 `TemperatureChanged` 事件且符合 CLS。`Temperature` 类显式定义 `raise_TemperatureChanged` 方法，以便其可以有选择地执行事件处理程序。

```
using System;
using System.Collections;
using System.Collections.Generic;

[assembly: CLSCompliant(true)]

public class TemperatureChangedEventArgs : EventArgs
{
   private Decimal originalTemp;
   private Decimal newTemp;
   private DateTimeOffset when;

   public TemperatureChangedEventArgs(Decimal original, Decimal @new, DateTimeOffset time)
   {
      originalTemp = original;
      newTemp = @new;
      when = time;
   }

   public Decimal OldTemperature
   {
      get { return originalTemp; }
   }

   public Decimal CurrentTemperature
   {
      get { return newTemp; }
   }

   public DateTimeOffset Time
   {
```

```
      get { return when; }
   }
}

public delegate void TemperatureChanged(Object sender, TemperatureChangedEventArgs e);

public class Temperature
{
   private struct TemperatureInfo
   {
      public Decimal Temperature;
      public DateTimeOffset Recorded;
   }

   public event TemperatureChanged TemperatureChanged;

   private Decimal previous;
   private Decimal current;
   private Decimal tolerance;
   private List<TemperatureInfo> tis = new List<TemperatureInfo>();

   public Temperature(Decimal temperature, Decimal tolerance)
   {
      current = temperature;
      TemperatureInfo ti = new TemperatureInfo();
      ti.Temperature = temperature;
      tis.Add(ti);
      ti.Recorded = DateTimeOffset.UtcNow;
      this.tolerance = tolerance;
   }

   public Decimal CurrentTemperature
   {
      get { return current; }
      set {
         TemperatureInfo ti = new TemperatureInfo();
         ti.Temperature = value;
         ti.Recorded = DateTimeOffset.UtcNow;
         previous = current;
         current = value;
         if (Math.Abs(current - previous) >= tolerance)
            raise_TemperatureChanged(new TemperatureChangedEventArgs(previous, current, ti.Recorded));
      }
   }

   public void raise_TemperatureChanged(TemperatureChangedEventArgs eventArgs)
   {
      if (TemperatureChanged == null)
         return;

      foreach (TemperatureChanged d in TemperatureChanged.GetInvocationList()) {
         if (d.Method.Name.Contains("Duplicate"))
            Console.WriteLine("Duplicate event handler; event handler not executed.");
         else
            d.Invoke(this, eventArgs);
      }
   }
}

public class Example
{
   public Temperature temp;

   public static void Main()
   {
      Example ex = new Example();
   }

   public Example()
```

```
   {
      temp = new Temperature(65, 3);
      temp.TemperatureChanged += this.TemperatureNotification;
      RecordTemperatures();
      Example ex = new Example(temp);
      ex.RecordTemperatures();
   }

   public Example(Temperature t)
   {
      temp = t;
      RecordTemperatures();
   }

   public void RecordTemperatures()
   {
      temp.TemperatureChanged += this.DuplicateTemperatureNotification;
      temp.CurrentTemperature = 66;
      temp.CurrentTemperature = 63;
   }

   internal void TemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }

   public void DuplicateTemperatureNotification(Object sender, TemperatureChangedEventArgs e)
   {
      Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature);
   }
}
```

```
Imports System.Collections
Imports System.Collections.Generic

<Assembly: CLSCompliant(True)>

Public Class TemperatureChangedEventArgs : Inherits EventArgs
    Private originalTemp As Decimal
    Private newTemp As Decimal
    Private [when] As DateTimeOffset

    Public Sub New(original As Decimal, [new] As Decimal, [time] As DateTimeOffset)
        originalTemp = original
        newTemp = [new]
        [when] = [time]
    End Sub

    Public ReadOnly Property OldTemperature As Decimal
        Get
            Return originalTemp
        End Get
    End Property

    Public ReadOnly Property CurrentTemperature As Decimal
        Get
            Return newTemp
        End Get
    End Property

    Public ReadOnly Property [Time] As DateTimeOffset
        Get
            Return [when]
        End Get
    End Property
```

```
End Class

Public Delegate Sub TemperatureChanged(sender As Object, e As TemperatureChangedEventArgs)

Public Class Temperature
    Private Structure TemperatureInfo
        Dim Temperature As Decimal
        Dim Recorded As DateTimeOffset
    End Structure

    Public Event TemperatureChanged As TemperatureChanged

    Private previous As Decimal
    Private current As Decimal
    Private tolerance As Decimal
    Private tis As New List(Of TemperatureInfo)

    Public Sub New(temperature As Decimal, tolerance As Decimal)
        current = temperature
        Dim ti As New TemperatureInfo()
        ti.Temperature = temperature
        ti.Recorded = DateTimeOffset.UtcNow
        tis.Add(ti)
        Me.tolerance = tolerance
    End Sub

    Public Property CurrentTemperature As Decimal
        Get
            Return current
        End Get
        Set
            Dim ti As New TemperatureInfo
            ti.Temperature = value
            ti.Recorded = DateTimeOffset.UtcNow
            previous = current
            current = value
            If Math.Abs(current - previous) >= tolerance Then
                raise_TemperatureChanged(New TemperatureChangedEventArgs(previous, current, ti.Recorded))
            End If
        End Set
    End Property

    Public Sub raise_TemperatureChanged(eventArgs As TemperatureChangedEventArgs)
        If TemperatureChangedEvent Is Nothing Then Exit Sub

        Dim ListenerList() As System.Delegate = TemperatureChangedEvent.GetInvocationList()
        For Each d As TemperatureChanged In TemperatureChangedEvent.GetInvocationList()
            If d.Method.Name.Contains("Duplicate") Then
                Console.WriteLine("Duplicate event handler; event handler not executed.")
            Else
                d.Invoke(Me, eventArgs)
            End If
        Next
    End Sub
End Class

Public Class Example
    Public WithEvents temp As Temperature

    Public Shared Sub Main()
        Dim ex As New Example()
    End Sub

    Public Sub New()
        temp = New Temperature(65, 3)
        RecordTemperatures()
        Dim ex As New Example(temp)
        ex.RecordTemperatures()
    End Sub
```

```
    Public Sub New(t As Temperature)
        temp = t
        RecordTemperatures()
    End Sub

    Public Sub RecordTemperatures()
        temp.CurrentTemperature = 66
        temp.CurrentTemperature = 63

    End Sub

    Friend Shared Sub TemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
        Console.WriteLine("Notification 1: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub

    Friend Shared Sub DuplicateTemperatureNotification(sender As Object, e As TemperatureChangedEventArgs) _
           Handles temp.TemperatureChanged
        Console.WriteLine("Notification 2: The temperature changed from {0} to {1}", e.OldTemperature,
e.CurrentTemperature)
    End Sub
End Class
```

Overloads

公共语言规范对重载成员有下列要求：

- 成员可以根据参数数量和任何参数的类型进行重载。在重载间进行区分时，不考虑应用于方法或其参数的调用约定、返回类型、自定义修饰符，也不考虑是按照值还是引用传递参数。有关示例，请参阅命名约定部分中名称在范围内必须是唯一的代码需求。
- 只可重载属性和方法。无法重载字段和事件。
- 泛型方法可以基于其泛型参数的数目进行重载。

> **NOTE**
>
> `op_Explicit` 和 `op_Implicit` 运算符是返回值不被视为重载决策的方法签名的一部分的规则的例外情况。可以基于这两个运算符的参数和返回值对其进行重载。

异常

异常对象必须从 System.Exception 派生或从派生自 System.Exception 的另一种类型派生。下面的示例说明了当名为 `ErrorClass` 的自定义类用于异常处理时产生的编译器错误。

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
// Compilation produces a compiler error like the following:
//    Exceptions1.cs(26,16): error CS0155: The type caught or thrown must be derived from
//            System.Exception
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass
    Dim msg As String

    Public Sub New(errorMessage As String)
        msg = errorMessage
    End Sub

    Public ReadOnly Property Message As String
        Get
            Return msg
        End Get
    End Property
End Class

Public Module StringUtilities
    <Extension()> Public Function SplitString(value As String, index As Integer) As String()
        If index < 0 Or index > value.Length Then
            Dim BadIndex As New ErrorClass("The index is not within the string.")
            Throw BadIndex
        End If
        Dim retVal() As String = {value.Substring(0, index - 1),
                                  value.Substring(index)}
        Return retVal
    End Function
End Module
' Compilation produces a compiler error like the following:
'    Exceptions1.vb(27) : error BC30665: 'Throw' operand must derive from 'System.Exception'.
'
'             Throw BadIndex
'             ~~~~~~~~~~~~~~
```

若要更正此错误，`ErrorClass` 类必须继承自 System.Exception。此外，必须重写 `Message` 属性。下面的示例更正这些错误以定义符合 CLS 的 `ErrorClass` 类。

```
using System;

[assembly: CLSCompliant(true)]

public class ErrorClass : Exception
{
   string msg;

   public ErrorClass(string errorMessage)
   {
      msg = errorMessage;
   }

   public override string Message
   {
      get { return msg; }
   }
}

public static class StringUtilities
{
   public static string[] SplitString(this string value, int index)
   {
      if (index < 0 | index > value.Length) {
         ErrorClass badIndex = new ErrorClass("The index is not within the string.");
         throw badIndex;
      }
      string[] retVal = { value.Substring(0, index - 1),
                          value.Substring(index) };
      return retVal;
   }
}
```

```
Imports System.Runtime.CompilerServices

<Assembly: CLSCompliant(True)>

Public Class ErrorClass : Inherits Exception
    Dim msg As String

    Public Sub New(errorMessage As String)
        msg = errorMessage
    End Sub

    Public Overrides ReadOnly Property Message As String
        Get
            Return msg
        End Get
    End Property
End Class

Public Module StringUtilities
    <Extension()> Public Function SplitString(value As String, index As Integer) As String()
        If index < 0 Or index > value.Length Then
            Dim BadIndex As New ErrorClass("The index is not within the string.")
            Throw BadIndex
        End If
        Dim retVal() As String = {value.Substring(0, index - 1),
                                  value.Substring(index)}
        Return retVal
    End Function
End Module
```

特性

在 .NET 程序集中，自定义特性提供了一个可扩展机制，用于存储自定义特性和检索有关编程对象（如程序集、类型、成员和方法参数）的元数据。自定义特性必须从 System.Attribute 派生或从派生自 `System.Attribute` 的类型派生。

下面的示例与此规则冲突。它定义了不是从 `NumericAttribute` 派生的 System.Attribute 类。编译器错误仅当应用不符合 CLS 的特性时会出现，而在定义类时不会出现。

```
using System;

[assembly: CLSCompliant(true)]

[AttributeUsageAttribute(AttributeTargets.Class | AttributeTargets.Struct)]
public class NumericAttribute
{
   private bool _isNumeric;

   public NumericAttribute(bool isNumeric)
   {
      _isNumeric = isNumeric;
   }

   public bool IsNumeric
   {
      get { return _isNumeric; }
   }
}

[Numeric(true)] public struct UDouble
{
   double Value;
}
// Compilation produces a compiler error like the following:
//    Attribute1.cs(22,2): error CS0616: 'NumericAttribute' is not an attribute class
//    Attribute1.cs(7,14): (Location of symbol related to previous error)
```

```
<Assembly: CLSCompliant(True)>

<AttributeUsageAttribute(AttributeTargets.Class Or AttributeTargets.Struct)> _
Public Class NumericAttribute
    Private _isNumeric As Boolean

    Public Sub New(isNumeric As Boolean)
        _isNumeric = isNumeric
    End Sub

    Public ReadOnly Property IsNumeric As Boolean
        Get
            Return _isNumeric
        End Get
    End Property
End Class

<Numeric(True)> Public Structure UDouble
    Dim Value As Double
End Structure
' Compilation produces a compiler error like the following:
'    error BC31504: 'NumericAttribute' cannot be used as an attribute because it
'    does not inherit from 'System.Attribute'.
'
'    <Numeric(True)> Public Structure UDouble
'     ~~~~~~~~~~~~~
```

构造函数或符合 CLS 的特性的属性只能公开以下类型：

- Boolean
- Byte
- Char
- Double
- Int16
- Int32
- Int64
- Single
- String
- Type
- 基础类型为 Byte、Int16、Int32 或 Int64 的任意枚举类型。

下面的示例定义了从 Attribute 派生的 `DescriptionAttribute` 类。类构造函数具有 `Descriptor` 类型的参数，因此，该类不符合 CLS。C# 编译器发出警告，但编译成功，而 Visual Basic 编译器既不发出警告也不报告错误。

```
using System;

[assembly:CLSCompliantAttribute(true)]

public enum DescriptorType { type, member };

public class Descriptor
{
   public DescriptorType Type;
   public String Description;
}

[AttributeUsage(AttributeTargets.All)]
public class DescriptionAttribute : Attribute
{
   private Descriptor desc;

   public DescriptionAttribute(Descriptor d)
   {
      desc = d;
   }

   public Descriptor Descriptor
   { get { return desc; } }
}
// Attempting to compile the example displays output like the following:
//       warning CS3015: 'DescriptionAttribute' has no accessible
//               constructors which use only CLS-compliant types
```

```
<Assembly: CLSCompliantAttribute(True)>

Public Enum DescriptorType As Integer
    Type = 0
    Member = 1
End Enum

Public Class Descriptor
    Public Type As DescriptorType
    Public Description As String
End Class

<AttributeUsage(AttributeTargets.All)> _
Public Class DescriptionAttribute : Inherits Attribute
    Private desc As Descriptor

    Public Sub New(d As Descriptor)
        desc = d
    End Sub

    Public ReadOnly Property Descriptor As Descriptor
        Get
            Return desc
        End Get
    End Property
End Class
```

CLSCompliantAttribute 特性

CLSCompliantAttribute 特性用于指示程序元素是否使用公共语言规范进行编译。
CLSCompliantAttribute(Boolean) 构造函数包含一个必需的参数 *isCompliant*，此参数指示该程序元素是否符合 CLS。

编译时，编译器检测到假定为符合 CLS 的不合规元素，并发出警告。编译器不会对显式声明为不符合标准的类型或成员发出警告。

组件开发人员可以通过两种方法使用 `CLSCompliantAttribute` 特性：

- 定义由组件公开的公共接口的符合 CLS 的部件以及不符合 CLS 的部件。如果特性用于将特定程序元素标记为符合 CLS，则使用该特性可保证能通过面向 .NET 的所有语言和工具访问这些元素。
- 确保组件库的公共接口仅公开符合 CLS 的程序元素。如果元素不符合 CLS，则编译器通常会发出一个警告。

> **WARNING**
> 在某些情况下，语言编译器执行符合 CLS 的规则，而不管是否使用 `CLSCompliantAttribute` 特性。例如，在接口中定义静态成员会违反 CLS 规则。就这一点而言，如果在接口中定义 `static`（在 C# 中）或 `Shared`（在 Visual Basic 中）成员，C# 和 Visual Basic 编译器都会显示错误消息，且无法编译应用。

CLSCompliantAttribute 特性标记为具有 AttributeUsageAttribute 值的 AttributeTargets.All 特性。利用此值，您可以将 CLSCompliantAttribute 特性应用于任何程序元素，包括程序集、模块、类型（类、结构、枚举、接口和委托）、类型成员（构造函数、方法、属性、字段和事件）、参数、泛型参数和返回值。但实际上，您只应将该特性应用于程序集、类型和类型成员。否则，编译器在库的公共接口中遇到不符合标准的参数、泛型参数或返回值时，将忽略此特性并继续生成编译器警告。

CLSCompliantAttribute 特性的值由包含的程序元素继承。例如，如果程序集标记为符合 CLS，则其类型也符合 CLS。如果类型标记为符合 CLS，则其嵌套的类型和成员也符合 CLS。

您可以通过将 CLSCompliantAttribute 特性应用到包含的编程元素来显式重写继承的遵从性。例如，可以使用具有 CLSCompliantAttribute 值为 `isCompliant` 的 `false` 特性来定义符合标准的程序集中的不符合标准的类型，还可以使用 `isCompliant` 值为 `true` 的特性来定义不符合标准的程序集中的符合标准的类型。您还可以在符合标准的类型中定义不符合标准的成员。但是，不符合标准的类型无法拥有符合标准的成员，因此您无法使用 `isCompliant` 值为 `true` 的特性从一个不符合标准的类型重写继承。

在开发组件时，应始终使用 CLSCompliantAttribute 特性来指示您的程序集、其类型及其成员是否符合 CLS。

创建符合 CLS 的组件：

1. 使用 CLSCompliantAttribute 将程序集标记为符合 CLS。

2. 将程序集中不符合 CLS 的所有公开的类型标记为不符合标准。

3. 将符合 CLS 的类型中的所有公开的成员标记为不符合标准。

4. 为不符合 CLS 的成员提供符合 CLS 的替代项。

如果已成功标记所有不符合标准的类型和成员，您的编译器不会发出任何不符合警告。但是，您应指出哪些成员不符合 CLS 并在产品文档中列出其不符合 CLS 的替代项。

下面的示例使用 CLSCompliantAttribute 特性定义符合 CLS 的程序集和类型 `CharacterUtilities`，该类型具有两个不符合 CLS 的成员。由于这两个成员标记有 `CLSCompliant(false)` 特性，因此编译器不生成任何警告。该类还为两种方法提供符合 CLS 的替代项。通常，我们只向 `ToUTF16` 方法添加两个重载，以便提供符合 CLS 的替代项。但是，由于无法基于返回值重载这些方法，因此符合 CLS 的方法的名称不同于不符合标准的方法的名称。

```
using System;
using System.Text;

[assembly:CLSCompliant(true)]

public class CharacterUtilities
{
   [CLSCompliant(false)] public static ushort ToUTF16(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return Convert.ToUInt16(s[0]);
   }

   [CLSCompliant(false)] public static ushort ToUTF16(Char ch)
   {
      return Convert.ToUInt16(ch);
   }

   // CLS-compliant alternative for ToUTF16(String).
   public static int ToUTF16CodeUnit(String s)
   {
      s = s.Normalize(NormalizationForm.FormC);
      return (int) Convert.ToUInt16(s[0]);
   }

   // CLS-compliant alternative for ToUTF16(Char).
   public static int ToUTF16CodeUnit(Char ch)
   {
      return Convert.ToInt32(ch);
   }

   public bool HasMultipleRepresentations(String s)
   {
      String s1 = s.Normalize(NormalizationForm.FormC);
      return s.Equals(s1);
   }

   public int GetUnicodeCodePoint(Char ch)
   {
      if (Char.IsSurrogate(ch))
         throw new ArgumentException("ch cannot be a high or low surrogate.");

      return Char.ConvertToUtf32(ch.ToString(), 0);
   }

   public int GetUnicodeCodePoint(Char[] chars)
   {
      if (chars.Length > 2)
         throw new ArgumentException("The array has too many characters.");

      if (chars.Length == 2) {
         if (! Char.IsSurrogatePair(chars[0], chars[1]))
            throw new ArgumentException("The array must contain a low and a high surrogate.");
         else
            return Char.ConvertToUtf32(chars[0], chars[1]);
      }
      else {
         return Char.ConvertToUtf32(chars.ToString(), 0);
      }
   }
}
```

```
Imports System.Text

<Assembly: CLSCompliant(True)>

Public Class CharacterUtilities
    <CLSCompliant(False)> Public Shared Function ToUTF16(s As String) As UShort
        s = s.Normalize(NormalizationForm.FormC)
        Return Convert.ToUInt16(s(0))
    End Function

    <CLSCompliant(False)> Public Shared Function ToUTF16(ch As Char) As UShort
        Return Convert.ToUInt16(ch)
    End Function

    ' CLS-compliant alternative for ToUTF16(String).
    Public Shared Function ToUTF16CodeUnit(s As String) As Integer
        s = s.Normalize(NormalizationForm.FormC)
        Return CInt(Convert.ToInt16(s(0)))
    End Function

    ' CLS-compliant alternative for ToUTF16(Char).
    Public Shared Function ToUTF16CodeUnit(ch As Char) As Integer
        Return Convert.ToInt32(ch)
    End Function

    Public Function HasMultipleRepresentations(s As String) As Boolean
        Dim s1 As String = s.Normalize(NormalizationForm.FormC)
        Return s.Equals(s1)
    End Function

    Public Function GetUnicodeCodePoint(ch As Char) As Integer
        If Char.IsSurrogate(ch) Then
            Throw New ArgumentException("ch cannot be a high or low surrogate.")
        End If
        Return Char.ConvertToUtf32(ch.ToString(), 0)
    End Function

    Public Function GetUnicodeCodePoint(chars() As Char) As Integer
        If chars.Length > 2 Then
            Throw New ArgumentException("The array has too many characters.")
        End If
        If chars.Length = 2 Then
            If Not Char.IsSurrogatePair(chars(0), chars(1)) Then
                Throw New ArgumentException("The array must contain a low and a high surrogate.")
            Else
                Return Char.ConvertToUtf32(chars(0), chars(1))
            End If
        Else
            Return Char.ConvertToUtf32(chars.ToString(), 0)
        End If
    End Function
End Class
```

如果您开发的是应用程序而不是库(即，如果不公开可由其他应用程序开发人员使用的类型或成员)，则只有在您的语言不支持程序元素时，您的应用程序使用的程序元素的 CLS 遵从性才会引起关注。在这种情况下，当您尝试使用不符合 CLS 的元素时，您的语言编译器将生成错误。

跨语言互操作性

语言独立性可能有几种含义。其中一种含义涉及到从使用一种语言编写的应用取用以另一种语言编写的类型。本文介绍第二个含义，涉及将用多种语言编写的代码组合到一个 .NET 程序集。

以下示例通过创建一个名为 Utilities.dll 的包含两个类(`NumericLib` 和 `StringLib`)的类库演示了跨语言互操作性。`NumericLib` 类用 C# 编写类，`StringLib` 类用 Visual Basic 编写。以下是 `StringUtil.vb` 的源代码，该源代

码在其 `StringLib` 类中包含单个成员 `ToTitleCase`。

```
Imports System.Collections.Generic
Imports System.Runtime.CompilerServices

Public Module StringLib
    Private exclusions As List(Of String)

    Sub New()
        Dim words() As String = {"a", "an", "and", "of", "the"}
        exclusions = New List(Of String)
        exclusions.AddRange(words)
    End Sub

    <Extension()> _
    Public Function ToTitleCase(title As String) As String
        Dim words() As String = title.Split()
        Dim result As String = String.Empty

        For ctr As Integer = 0 To words.Length - 1
            Dim word As String = words(ctr)
            If ctr = 0 OrElse Not exclusions.Contains(word.ToLower()) Then
                result += word.Substring(0, 1).ToUpper() + _
                          word.Substring(1).ToLower()
            Else
                result += word.ToLower()
            End If
            If ctr <= words.Length - 1 Then
                result += " "
            End If
        Next
        Return result
    End Function
End Module
```

以下是 NumberUtil.cs 的源代码，定义具有 `NumericLib` 和 `IsEven` 两个成员的 `NearZero` 类。

```
using System;

public static class NumericLib
{
   public static bool IsEven(this IConvertible number)
   {
      if (number is Byte ||
          number is SByte ||
          number is Int16 ||
          number is UInt16 ||
          number is Int32 ||
          number is UInt32 ||
          number is Int64)
         return Convert.ToInt64(number) % 2 == 0;
      else if (number is UInt64)
         return ((ulong) number) % 2 == 0;
      else
         throw new NotSupportedException("IsEven called for a non-integer value.");
   }

   public static bool NearZero(double number)
   {
      return Math.Abs(number) < .00001;
   }
}
```

若要将两个类打包到单个程序集中，必须将它们编译到模块中。要将 Visual Basic 源代码文件编译到模块，请使

用此命令：

```
vbc /t:module StringUtil.vb
```

有关 Visual Basic 编译器的命令行语法的详细信息，请参阅从命令行生成。

要将 C# 源代码文件编译到模块，请使用此命令：

```
csc /t:module NumberUtil.cs
```

然后，可以使用链接器选项将两个模块编译到一个程序集：

```
link numberutil.netmodule stringutil.netmodule /out:UtilityLib.dll /dll
```

然后以下实例调用 `NumericLib.NearZero` 和 `StringLib.ToTitleCase` 方法。Visual Basic 代码和 C# 代码都能够访问这两个类中的方法。

```
using System;

public class Example
{
   public static void Main()
   {
      Double dbl = 0.0 - Double.Epsilon;
      Console.WriteLine(NumericLib.NearZero(dbl));

      string s = "war and peace";
      Console.WriteLine(s.ToTitleCase());
   }
}
// The example displays the following output:
//       True
//       War and Peace
```

```
Module Example
    Public Sub Main()
        Dim dbl As Double = 0.0 - Double.Epsilon
        Console.WriteLine(NumericLib.NearZero(dbl))

        Dim s As String = "war and peace"
        Console.WriteLine(s.ToTitleCase())
    End Sub
End Module
' The example displays the following output:
'       True
'       War and Peace
```

要编译 Visual Basic 代码，请使用此命令：

```
vbc example.vb /r:UtilityLib.dll
```

若要使用 C# 进行编译，请将编译器的名称从 `vbc` 更改为 `csc`，并将文件扩展名从 .vb 更改为 .cs：

```
csc example.cs /r:UtilityLib.dll
```

.NET 中的类型转换

2021/11/16 •

每个值都有与之关联的类型，此类型定义分配给该值的空间大小、它可以具有的可能值的范围以及它可以提供的成员等特性。许多值可以表示为多种类型。例如，值 4 可以表示为整数或浮点值。类型转换可以创建一个等同于旧类型值的新类型值，但却不必保留原始对象的恒等值（或精确值）。

.NET 自动支持以下转换：

- 从派生类转换为基类。例如，这意味着可将任何类或结构的实例转换为 Object 实例。此转换不需要强制转换或转换运算符。
- 从基类转换回原始的派生类。在 C# 中，此转换需要强制转换运算符。在 Visual Basic 中，如果 `Option Strict` 处于开启状态，则它需要 `CType` 运算符。
- 从实现接口的类型转换为表示该接口的接口对象。此转换不需要强制转换或转换运算符。
- 从接口对象转换回实现该接口的原始类型。在 C# 中，此转换需要强制转换运算符。在 Visual Basic 中，如果 `Option Strict` 处于开启状态，则它需要 `CType` 运算符。

除这些自动转换外，.NET 还提供支持自定义类型转换的多种功能。其中包括：

- `Implicit` 运算符，该运算符定义类型之间可用的扩大转换。有关详细信息，请参阅使用隐式运算符的隐式转换部分。
- `Explicit` 运算符，该运算符定义类型之间可用的收缩转换。有关详细信息，请参阅使用显式运算符的显式转换部分。
- IConvertible 接口，该接口定义到 .NET 每个基数据类型的转换。有关更多信息，请参阅 IConvertible 接口部分。
- Convert 类，该类提供了一组方法来实现 IConvertible 接口中的方法。有关更多信息，请参阅 Convert 类部分。
- TypeConverter 类，该类是一个基类，可以扩展该类以支持指定的类型到任何其他类型的转换。有关更多信息，请参阅 TypeConverter 类部分。

使用隐式运算符的隐式转换

扩大转换涉及从现有类型的值创建一个新值，该现有类型比目标类型具有限制性更强的范围或限制性更强的成员列表。扩大转换不会导致数据丢失（但可能导致精度损失）。由于不会丢失数据，因此编译器可以隐式或透明地处理转换，无需使用显式转换方法或强制转换运算符。

> **NOTE**
> 虽然执行隐式转换的代码可以调用转换方法或使用强制转换运算符，但支持隐式转换的编译器不需要调用转换方法或使用强制转换运算符。

例如，Decimal 类型支持从 Byte、Char、Int16、Int32、Int64、SByte、UInt16、UInt32 和 UInt64 值进行的隐式转换。下面的示例通过为 Decimal 变量赋值演示了其中的一些隐式转换。

```
byte byteValue = 16;
short shortValue = -1024;
int intValue = -1034000;
long longValue = 1152921504606846976;
ulong ulongValue = UInt64.MaxValue;

decimal decimalValue;

decimalValue = byteValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  byteValue.GetType().Name, decimalValue);

decimalValue = shortValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  shortValue.GetType().Name, decimalValue);

decimalValue = intValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  intValue.GetType().Name, decimalValue);

decimalValue = longValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue);

decimalValue = ulongValue;
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue);
// The example displays the following output:
//    After assigning a Byte value, the Decimal value is 16.
//    After assigning a Int16 value, the Decimal value is -1024.
//    After assigning a Int32 value, the Decimal value is -1034000.
//    After assigning a Int64 value, the Decimal value is 1152921504606846976.
//    After assigning a Int64 value, the Decimal value is 18446744073709551615.
```

```
Dim byteValue As Byte = 16
Dim shortValue As Short = -1024
Dim intValue As Integer = -1034000
Dim longValue As Long = CLng(1024 ^ 6)
Dim ulongValue As ULong = ULong.MaxValue

Dim decimalValue As Decimal

decimalValue = byteValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  byteValue.GetType().Name, decimalValue)

decimalValue = shortValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  shortValue.GetType().Name, decimalValue)

decimalValue = intValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  intValue.GetType().Name, decimalValue)

decimalValue = longValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue)

decimalValue = ulongValue
Console.WriteLine("After assigning a {0} value, the Decimal value is {1}.",
                  longValue.GetType().Name, decimalValue)
' The example displays the following output:
'    After assigning a Byte value, the Decimal value is 16.
'    After assigning a Int16 value, the Decimal value is -1024.
'    After assigning a Int32 value, the Decimal value is -1034000.
'    After assigning a Int64 value, the Decimal value is 1152921504606846976.
'    After assigning a Int64 value, the Decimal value is 18446744073709551615.
```

如果特定语言编译器支持自定义运算符，则您还可以在自己的自定义类型中定义隐式转换。下面的示例提供了一个名为 `ByteWithSign` 的有符号字节数据类型的分部实现，该分部实现使用符号数值表示法。它支持 Byte 和 SByte 值到 `ByteWithSign` 值的隐式转换。

```
public struct ByteWithSign
{
   private SByte signValue;
   private Byte value;

   public static implicit operator ByteWithSign(SByte value)
   {
      ByteWithSign newValue;
      newValue.signValue = (SByte) Math.Sign(value);
      newValue.value = (byte) Math.Abs(value);
      return newValue;
   }

   public static implicit operator ByteWithSign(Byte value)
   {
      ByteWithSign  newValue;
      newValue.signValue = 1;
      newValue.value = value;
      return newValue;
   }

   public override string ToString()
   {
      return (signValue * value).ToString();
   }
}
```

```
Public Structure ByteWithSign
    Private signValue As SByte
    Private value As Byte

    Public Overloads Shared Widening Operator CType(value As SByte) As ByteWithSign
        Dim newValue As ByteWithSign
        newValue.signValue = CSByte(Math.Sign(value))
        newValue.value = CByte(Math.Abs(value))
        Return newValue
    End Operator

    Public Overloads Shared Widening Operator CType(value As Byte) As ByteWithSign
        Dim NewValue As ByteWithSign
        newValue.signValue = 1
        newValue.value = value
        Return newValue
    End Operator

    Public Overrides Function ToString() As String
        Return (signValue * value).ToString()
    End Function
End Structure
```

然后，客户端代码可以声明一个 `ByteWithSign` 变量，并为该变量赋予 Byte 和 SByte 值，而无需执行任何显式转换或使用任何强制转换运算符，如下面的示例所示。

```
SByte sbyteValue = -120;
ByteWithSign value = sbyteValue;
Console.WriteLine(value);
value = Byte.MaxValue;
Console.WriteLine(value);
// The example displays the following output:
//       -120
//       255
```

```
Dim sbyteValue As SByte = -120
Dim value As ByteWithSign = sbyteValue
Console.WriteLine(value.ToString())
value = Byte.MaxValue
Console.WriteLine(value.ToString())
' The example displays the following output:
'       -120
'       255
```

使用显式运算符的显式转换

收缩转换涉及从现有类型的值创建一个新值，该现有类型比目标类型具有更大的范围和更大的成员列表。由于收缩转换可以导致数据丢失，因此编译器通常需要通过调用转换方法或使用强制转换运算符来进行显式转换。也就是说，必须在开发人员代码中显式处理收缩转换。

> **NOTE**
>
> 收缩转换之所以需要使用转换方法或强制转换运算符，主要是为提醒开发人员可能会丢失数据或引发 OverflowException，以便可以在代码中对其进行处理。但是，有些编译器可以放宽此要求。例如，在 Visual Basic 中，如果 `Option Strict` 关闭（其默认设置），则 Visual Basic 编译器会尝试隐式执行收缩转换。

例如，UInt32、Int64 和 UInt64 数据类型均具有超过 Int32 数据类型的范围，如下表所示。

类型	与 Int32 类型的比较
Int64	Int64.MaxValue 大于 Int32.MaxValue；Int64.MinValue 小于 Int32.MinValue（即比后者具有更大的负范围）。
UInt32	UInt32.MaxValue 大于 Int32.MaxValue。
UInt64	UInt64.MaxValue 大于 Int32.MaxValue。

为了处理这种收缩转换，.NET 允许类型定义 `Explicit` 运算符。然后，各种语言编译器可以使用自己的语法实现此运算符，也可以调用 Convert 类的成员来执行此转换。（有关 Convert 类的详细信息，请参阅本主题后面的 Convert 类。）下面的示例演示如何使用语言功能来处理这些可能超出范围的整数值到 Int32 值的显式转换。

```
long number1 = int.MaxValue + 20L;
uint number2 = int.MaxValue - 1000;
ulong number3 = int.MaxValue;

int intNumber;

try {
   intNumber = checked((int) number1);
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number1.GetType().Name, intNumber);
}
catch (OverflowException) {
   if (number1 > int.MaxValue)
      Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                        number1, int.MaxValue);
   else
      Console.WriteLine("Conversion failed: {0} is less than {1}.",
                        number1, int.MinValue);
}

try {
   intNumber = checked((int) number2);
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number2.GetType().Name, intNumber);
}
catch (OverflowException) {
   Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                     number2, int.MaxValue);
}

try {
   intNumber = checked((int) number3);
   Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                     number3.GetType().Name, intNumber);
}
catch (OverflowException) {
   Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                     number1, int.MaxValue);
}

// The example displays the following output:
//    Conversion failed: 2147483667 exceeds 2147483647.
//    After assigning a UInt32 value, the Integer value is 2147482647.
//    After assigning a UInt64 value, the Integer value is 2147483647.
```

```
Dim number1 As Long = Integer.MaxValue + 20L
Dim number2 As UInteger = Integer.MaxValue - 1000
Dim number3 As ULong = Integer.MaxValue

Dim intNumber As Integer

Try
    intNumber = CInt(number1)
    Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                      number1.GetType().Name, intNumber)
Catch e As OverflowException
    If number1 > Integer.MaxValue Then
        Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                        number1, Integer.MaxValue)
    Else
        Console.WriteLine("Conversion failed: {0} is less than {1}.\n",
                                        number1, Integer.MinValue)
    End If
End Try

Try
    intNumber = CInt(number2)
    Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                      number2.GetType().Name, intNumber)
Catch e As OverflowException
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                    number2, Integer.MaxValue)
End Try

Try
    intNumber = CInt(number3)
    Console.WriteLine("After assigning a {0} value, the Integer value is {1}.",
                      number3.GetType().Name, intNumber)
Catch e As OverflowException
    Console.WriteLine("Conversion failed: {0} exceeds {1}.",
                                    number1, Integer.MaxValue)
End Try
' The example displays the following output:
'    Conversion failed: 2147483667 exceeds 2147483647.
'    After assigning a UInt32 value, the Integer value is 2147482647.
'    After assigning a UInt64 value, the Integer value is 2147483647.
```

显式转换在不同的语言中可能会产生不同的结果，并且这些结果可能因对应的 Convert 方法所返回的值而异。例如，如果将 Double 值 12.63251 转换为 Int32，则 Visual Basic `CInt` 方法和 .NET Convert.ToInt32(Double) 方法会对 Double 进行舍入以返回值 13，而 C# `(int)` 运算符会截断 Double 以返回值 12。类似地，C# `(int)` 运算符不支持从布尔值到整数的转换，而 Visual Basic `CInt` 方法会将值 `true` 转换为 -1。另一方面，Convert.ToInt32(Boolean) 方法将值 `true` 转换为 1。

大多数编译器允许以有检查或无检查的方式执行显式转换。当执行有检查转换时，如果被转换类型的值超出了目标类型的范围，则会引发 OverflowException。在相同条件下执行无检查转换时，转换可能不会引发异常，但无法确定确切的行为，并且可能产生不正确的值。

NOTE

在 C# 中，可将 `checked` 关键字与强制转换运算符一起使用来执行有检查转换，也可通过指定 `/checked+` 编译器选项来执行有检查转换。反过来，可将 `unchecked` 关键字与强制转换运算符一起使用来执行无检查转换，或者通过指定 `/checked-` 编译器选项来执行无检查转换。默认情况下，显式转换将为无检查转换。在 Visual Basic 中，通过清除项目的“高级编译器设置”对话框中的“不做整数溢出检查”复选框或指定 `/removeintchecks-` 编译器选项，可以执行有检查转换。反之，通过选中项目的“高级编译器设置”对话框中的“不做整数溢出检查”复选框，或者指定 `/removeintchecks+` 编译器选项，可以执行无检查转换。默认情况下，显式转换将为有检查转换。

下面的 C# 示例使用 `checked` 和 `unchecked` 关键字阐释了将 Byte 范围外的值转换为 Byte 时的行为差异。有检查转换会引发异常，但无检查转换会向 Byte.MaxValue 变量赋予 Byte。

```
int largeValue = Int32.MaxValue;
byte newValue;

try {
   newValue = unchecked((byte) largeValue);
   Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                     largeValue.GetType().Name, largeValue,
                     newValue.GetType().Name, newValue);
}
catch (OverflowException) {
   Console.WriteLine("{0} is outside the range of the Byte data type.",
                     largeValue);
}

try {
   newValue = checked((byte) largeValue);
   Console.WriteLine("Converted the {0} value {1} to the {2} value {3}.",
                     largeValue.GetType().Name, largeValue,
                     newValue.GetType().Name, newValue);
}
catch (OverflowException) {
   Console.WriteLine("{0} is outside the range of the Byte data type.",
                     largeValue);
}
// The example displays the following output:
//    Converted the Int32 value 2147483647 to the Byte value 255.
//    2147483647 is outside the range of the Byte data type.
```

如果特定语言编译器支持自定义重载运算符，您还可以在自己的自定义类型中定义显式转换。下面的示例提供了一个名为 `ByteWithSign` 的有符号字节数据类型的分部实现，该分部实现使用符号数值表示法。它支持 Int32 和 UInt32 值到 `ByteWithSign` 值的显式转换。

```
public struct ByteWithSign
{
   private SByte signValue;
   private Byte value;

   private const byte MaxValue = byte.MaxValue;
   private const int MinValue = -1 * byte.MaxValue;

   public static explicit operator ByteWithSign(int value)
   {
      // Check for overflow.
      if (value > ByteWithSign.MaxValue || value < ByteWithSign.MinValue)
         throw new OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data type.",
                                                   value));

      ByteWithSign newValue;
      newValue.signValue = (SByte) Math.Sign(value);
      newValue.value = (byte) Math.Abs(value);
      return newValue;
   }

   public static explicit operator ByteWithSign(uint value)
   {
      if (value > ByteWithSign.MaxValue)
         throw new OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data type.",
                                                   value));

      ByteWithSign newValue;
      newValue.signValue = 1;
      newValue.value = (byte) value;
      return newValue;
   }

   public override string ToString()
   {
      return (signValue * value).ToString();
   }
}
```

```
Public Structure ByteWithSign
    Private signValue As SByte
    Private value As Byte

    Private Const MaxValue As Byte = Byte.MaxValue
    Private Const MinValue As Integer = -1 * Byte.MaxValue

    Public Overloads Shared Narrowing Operator CType(value As Integer) As ByteWithSign
        ' Check for overflow.
        If value > ByteWithSign.MaxValue Or value < ByteWithSign.MinValue Then
            Throw New OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data
type.", value))
        End If

        Dim newValue As ByteWithSign

        newValue.signValue = CSByte(Math.Sign(value))
        newValue.value = CByte(Math.Abs(value))
        Return newValue
    End Operator

    Public Overloads Shared Narrowing Operator CType(value As UInteger) As ByteWithSign
        If value > ByteWithSign.MaxValue Then
            Throw New OverflowException(String.Format("'{0}' is out of range of the ByteWithSign data
type.", value))
        End If

        Dim NewValue As ByteWithSign

        newValue.signValue = 1
        newValue.value = CByte(value)
        Return newValue
    End Operator

    Public Overrides Function ToString() As String
        Return (signValue * value).ToString()
    End Function
End Structure
```

然后，客户端代码可以声明一个 `ByteWithSign` 变量，并为该变量赋予 Int32 和 UInt32 值（如果赋值中包括一个强制转换运算符或转换方法），如下面的示例所示。

```
ByteWithSign value;

try {
   int intValue = -120;
   value = (ByteWithSign) intValue;
   Console.WriteLine(value);
}
catch (OverflowException e) {
   Console.WriteLine(e.Message);
}

try {
   uint uintValue = 1024;
   value = (ByteWithSign) uintValue;
   Console.WriteLine(value);
}
catch (OverflowException e) {
    Console.WriteLine(e.Message);
}
// The example displays the following output:
//       -120
//       '1024' is out of range of the ByteWithSign data type.
```

```
Dim value As ByteWithSign

Try
    Dim intValue As Integer = -120
    value = CType(intValue, ByteWithSign)
    Console.WriteLine(value)
Catch e As OverflowException
    Console.WriteLine(e.Message)
End Try

Try
    Dim uintValue As UInteger = 1024
    value = CType(uintValue, ByteWithSign)
    Console.WriteLine(value)
Catch e As OverflowException
    Console.WriteLine(e.Message)
End Try
' The example displays the following output:
'       -120
'       '1024' is out of range of the ByteWithSign data type.
```

IConvertible 接口

为了支持任意类型到公共语言运行时基类型的转换，.NET 提供了 IConvertible 接口。需要使用实现类型以提供以下方法：

- 一个返回实现类型的 TypeCode 的方法。
- 用于将实现类型转换为公共语言运行时的每一种基类型（Boolean、Byte、DateTime、Decimal 和 Double 等）的各种方法。
- 一个用于将实现类型的实例转换为另一个指定类型的通用转换方法。不支持的转换应引发 InvalidCastException。

公共语言运行时的每一种基类型（即 Boolean、Byte、Char、DateTime、Decimal、Double、Int16、Int32、Int64、SByte、Single、String、UInt16、UInt32 和 UInt64）以及 DBNull 和 Enum 类型都可以实现 IConvertible 接口。但是，这些是显式接口实现；因此只能通过 IConvertible 接口变量来调用转换方法，如下面的示例所示。此示例将一个 Int32 值转换为其等效的 Char 值。

```
int codePoint = 1067;
IConvertible iConv = codePoint;
char ch = iConv.ToChar(null);
Console.WriteLine("Converted {0} to {1}.", codePoint, ch);
```

```
Dim codePoint As Integer = 1067
Dim iConv As IConvertible = codePoint
Dim ch As Char = iConv.ToChar(Nothing)
Console.WriteLine("Converted {0} to {1}.", codePoint, ch)
```

对转换方法的接口（而不是实现类型）调用转换方法的要求使显式接口实现成为一种代价相对较大的操作。因此，在公共语言运行时基类型之间进行转换时，建议您调用 Convert 类的适当成员。有关详细信息，请参阅下一部分 Convert 类。

> **NOTE**
>
> 除了 .NET 提供的 IConvertible 接口和 Convert 类，各种语言还可能会提供其他方法来执行转换。例如，C# 使用强制转换运算符；Visual Basic 使用编译器实现的转换函数，例如 `CType`、`CInt` 和 `DirectCast`。

大多数情况下，IConvertible 接口设计为支持 .NET 中基类型之间的转换。但是，通过自定义类型也可以实现该接口，以便支持该类型到其他自定义类型的转换。有关详细信息，请参阅本主题后面的使用 ChangeType 方法的自定义转换部分。

Convert 类

虽然可以调用每个基类型的 IConvertible 接口实现来执行类型转换，但从一种基类型转换为另一种基类型时，建议您调用 System.Convert 类的方法，这种方式与语言无关。此外，Convert.ChangeType(Object, Type, IFormatProvider) 方法还可用于从一个指定的自定义类型转换为另一种类型。

基类型之间的转换

Convert 类提供了一种与语言无关的方式来执行基类型之间的转换，并且该类可用于面向公共语言运行时的所有语言。它为扩大转换和收缩转换提供了一组完整的方法，并且会对不支持的转换（例如 InvalidCastException 值到整数值的转换）引发 DateTime。收缩转换是在已检查的上下文中执行的，如果转换失败，将引发 OverflowException。

> **IMPORTANT**
> 由于 Convert 类包含用于转换为每个基类型和从每个基类型进行转换的方法，因此不再需要调用每个基类型的 IConvertible 显式接口实现。

下面的示例演示如何使用 System.Convert 类来执行 .NET 基类型之间的多种扩大转换和收缩转换。

```
// Convert an Int32 value to a Decimal (a widening conversion).
int integralValue = 12534;
decimal decimalValue = Convert.ToDecimal(integralValue);
Console.WriteLine("Converted the {0} value {1} to " +
                                  "the {2} value {3:N2}.",
                                  integralValue.GetType().Name,
                                  integralValue,
                                  decimalValue.GetType().Name,
                                  decimalValue);
// Convert a Byte value to an Int32 value (a widening conversion).
byte byteValue = Byte.MaxValue;
int integralValue2 = Convert.ToInt32(byteValue);
Console.WriteLine("Converted the {0} value {1} to " +
                                  "the {2} value {3:G}.",
                                  byteValue.GetType().Name,
                                  byteValue,
                                  integralValue2.GetType().Name,
                                  integralValue2);

// Convert a Double value to an Int32 value (a narrowing conversion).
double doubleValue = 16.32513e12;
try {
   long longValue = Convert.ToInt64(doubleValue);
   Console.WriteLine("Converted the {0} value {1:E} to " +
                                     "the {2} value {3:N0}.",
                                     doubleValue.GetType().Name,
                                     doubleValue,
                                     longValue.GetType().Name,
                                     longValue);
}
catch (OverflowException) {
   Console.WriteLine("Unable to convert the {0:E} value {1}.",
                                     doubleValue.GetType().Name, doubleValue);
}

// Convert a signed byte to a byte (a narrowing conversion).
sbyte sbyteValue = -16;
try {
   byte byteValue2 = Convert.ToByte(sbyteValue);
   Console.WriteLine("Converted the {0} value {1} to " +
                                     "the {2} value {3:G}.",
                                     sbyteValue.GetType().Name,
                                     sbyteValue,
                                     byteValue2.GetType().Name,
                                     byteValue2);
}
catch (OverflowException) {
   Console.WriteLine("Unable to convert the {0} value {1}.",
                                     sbyteValue.GetType().Name, sbyteValue);
}
// The example displays the following output:
//       Converted the Int32 value 12534 to the Decimal value 12,534.00.
//       Converted the Byte value 255 to the Int32 value 255.
//       Converted the Double value 1.632513E+013 to the Int64 value 16,325,130,000,000.
//       Unable to convert the SByte value -16.
```

```
' Convert an Int32 value to a Decimal (a widening conversion).
Dim integralValue As Integer = 12534
Dim decimalValue As Decimal = Convert.ToDecimal(integralValue)
Console.WriteLine("Converted the {0} value {1} to the {2} value {3:N2}.",
                  integralValue.GetType().Name,
                  integralValue,
                  decimalValue.GetType().Name,
                  decimalValue)

' Convert a Byte value to an Int32 value (a widening conversion).
Dim byteValue As Byte = Byte.MaxValue
Dim integralValue2 As Integer = Convert.ToInt32(byteValue)
Console.WriteLine("Converted the {0} value {1} to " +
                                 "the {2} value {3:G}.",
                                 byteValue.GetType().Name,
                                 byteValue,
                                 integralValue2.GetType().Name,
                                 integralValue2)

' Convert a Double value to an Int32 value (a narrowing conversion).
Dim doubleValue As Double = 16.32513e12
Try
    Dim longValue As Long = Convert.ToInt64(doubleValue)
    Console.WriteLine("Converted the {0} value {1:E} to " +
                                     "the {2} value {3:N0}.",
                                     doubleValue.GetType().Name,
                                     doubleValue,
                                     longValue.GetType().Name,
                                     longValue)
Catch e As OverflowException
    Console.WriteLine("Unable to convert the {0:E} value {1}.",
                                     doubleValue.GetType().Name, doubleValue)
End Try

' Convert a signed byte to a byte (a narrowing conversion).
Dim sbyteValue As SByte = -16
Try
    Dim byteValue2 As Byte = Convert.ToByte(sbyteValue)
    Console.WriteLine("Converted the {0} value {1} to " +
                                     "the {2} value {3:G}.",
                                     sbyteValue.GetType().Name,
                                     sbyteValue,
                                     byteValue2.GetType().Name,
                                     byteValue2)
Catch e As OverflowException
    Console.WriteLine("Unable to convert the {0} value {1}.",
                                     sbyteValue.GetType().Name, sbyteValue)
End Try
' The example displays the following output:
'       Converted the Int32 value 12534 to the Decimal value 12,534.00.
'       Converted the Byte value 255 to the Int32 value 255.
'       Converted the Double value 1.632513E+013 to the Int64 value 16,325,130,000,000.
'       Unable to convert the SByte value -16.
```

在某些情况下，尤其是当转换为浮点值和从浮点值进行转换时，转换可能会丢失精度，即使不引发 OverflowException 时也是如此。下面的示例演示了这种精度丢失。在第一种情况下，Decimal 值在转换为 Double 后精度降低（有效位减少）。在第二种情况下，Double 值从 42.72 四舍五入为 43 以完成转换。

```
double doubleValue;

// Convert a Double to a Decimal.
decimal decimalValue = 13956810.967028881234514711211m;
doubleValue = Convert.ToDouble(decimalValue);
Console.WriteLine("{0} converted to {1}.", decimalValue, doubleValue);

doubleValue = 42.72;
try {
   int integerValue = Convert.ToInt32(doubleValue);
   Console.WriteLine("{0} converted to {1}.",
                                    doubleValue, integerValue);
}
catch (OverflowException) {
   Console.WriteLine("Unable to convert {0} to an integer.",
                                    doubleValue);
}
// The example displays the following output:
//       13956810.96702888123451471211 converted to 13956810.9670289.
//       42.72 converted to 43.
```

```
Dim doubleValue As Double

' Convert a Double to a Decimal.
Dim decimalValue As Decimal = 13956810.96702888123451471211d
doubleValue = Convert.ToDouble(decimalValue)
Console.WriteLine("{0} converted to {1}.", decimalValue, doubleValue)

doubleValue = 42.72
Try
    Dim integerValue As Integer = Convert.ToInt32(doubleValue)
    Console.WriteLine("{0} converted to {1}.",
                                     doubleValue, integerValue)
Catch e As OverflowException
    Console.WriteLine("Unable to convert {0} to an integer.",
                                     doubleValue)
End Try
' The example displays the following output:
'       13956810.96702888123451471211 converted to 13956810.9670289.
'       42.72 converted to 43.
```

有关列出 Convert 类支持的扩大转换和收缩转换的表，请参阅类型转换表。

使用 ChangeType 方法的自定义转换

除了支持到每个基类型的转换外，Convert 类还可用于将一个自定义类型转换为一个或多个预定义类型。此转换是通过 Convert.ChangeType(Object, Type, IFormatProvider) 方法执行的，而此方法包装了对 IConvertible.ToType 参数的 `value` 方法的调用。这意味着 `value` 参数所表示的对象必须提供 IConvertible 接口的实现。

> **NOTE**
>
> 由于 Convert.ChangeType(Object, Type) 和 Convert.ChangeType(Object, Type, IFormatProvider) 方法使用 Type 对象来指定 `value` 将转换为的目标类型，因此它们可用于执行对象（其类型在编译时是未知的）的动态转换。但请注意，IConvertible 的 `value` 实现必须仍支持此转换。

下面的示例演示 IConvertible 接口的一个可能实现，该实现允许将 `TemperatureCelsius` 对象转换为 `TemperatureFahrenheit` 对象，反之亦然。此示例定义一个基类 `Temperature`，该基类实现 IConvertible 接口并重写 Object.ToString 方法。派生的 `TemperatureCelsius` 和 `TemperatureFahrenheit` 类分别重写该基类的 `ToType` 和 `ToString` 方法。

```
using System;

public abstract class Temperature : IConvertible
{
   protected decimal temp;

   public Temperature(decimal temperature)
   {
      this.temp = temperature;
   }

   public decimal Value
   {
      get { return this.temp; }
      set { this.temp = value; }
   }

   public override string ToString()
   {
      return temp.ToString(null as IFormatProvider) + "º";
   }

   // IConvertible implementations.
   public TypeCode GetTypeCode() {
      return TypeCode.Object;
   }

   public bool ToBoolean(IFormatProvider provider) {
      throw new InvalidCastException(String.Format("Temperature-to-Boolean conversion is not supported."));
   }

   public byte ToByte(IFormatProvider provider) {
      if (temp < Byte.MinValue || temp > Byte.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the Byte data type.", temp));
      else
         return (byte) temp;
   }

   public char ToChar(IFormatProvider provider) {
      throw new InvalidCastException("Temperature-to-Char conversion is not supported.");
   }

   public DateTime ToDateTime(IFormatProvider provider) {
      throw new InvalidCastException("Temperature-to-DateTime conversion is not supported.");
   }

   public decimal ToDecimal(IFormatProvider provider) {
      return temp;
   }

   public double ToDouble(IFormatProvider provider) {
      return (double) temp;
   }

   public short ToInt16(IFormatProvider provider) {
      if (temp < Int16.MinValue || temp > Int16.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the Int16 data type.", temp));
      else
         return (short) Math.Round(temp);
   }

   public int ToInt32(IFormatProvider provider) {
      if (temp < Int32.MinValue || temp > Int32.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the Int32 data type.", temp));
      else
         return (int) Math.Round(temp);
   }

   public long ToInt64(IFormatProvider provider) {
```

```
   if (temp < Int64.MinValue || temp > Int64.MaxValue)
      throw new OverflowException(String.Format("{0} is out of range of the Int64 data type.", temp));
   else
      return (long) Math.Round(temp);
}

public sbyte ToSByte(IFormatProvider provider) {
   if (temp < SByte.MinValue || temp > SByte.MaxValue)
      throw new OverflowException(String.Format("{0} is out of range of the SByte data type.", temp));
   else
      return (sbyte) temp;
}

public float ToSingle(IFormatProvider provider) {
   return (float) temp;
}

public virtual string ToString(IFormatProvider provider) {
   return temp.ToString(provider) + "°";
}

// If conversionType is implemented by another IConvertible method, call it.
public virtual object ToType(Type conversionType, IFormatProvider provider) {
   switch (Type.GetTypeCode(conversionType))
   {
      case TypeCode.Boolean:
         return this.ToBoolean(provider);
      case TypeCode.Byte:
         return this.ToByte(provider);
      case TypeCode.Char:
         return this.ToChar(provider);
      case TypeCode.DateTime:
         return this.ToDateTime(provider);
      case TypeCode.Decimal:
         return this.ToDecimal(provider);
      case TypeCode.Double:
         return this.ToDouble(provider);
      case TypeCode.Empty:
         throw new NullReferenceException("The target type is null.");
      case TypeCode.Int16:
         return this.ToInt16(provider);
      case TypeCode.Int32:
         return this.ToInt32(provider);
      case TypeCode.Int64:
         return this.ToInt64(provider);
      case TypeCode.Object:
         // Leave conversion of non-base types to derived classes.
         throw new InvalidCastException(String.Format("Cannot convert from Temperature to {0}.",
                                        conversionType.Name));
      case TypeCode.SByte:
         return this.ToSByte(provider);
      case TypeCode.Single:
         return this.ToSingle(provider);
      case TypeCode.String:
         IConvertible iconv = this;
         return iconv.ToString(provider);
      case TypeCode.UInt16:
         return this.ToUInt16(provider);
      case TypeCode.UInt32:
         return this.ToUInt32(provider);
      case TypeCode.UInt64:
         return this.ToUInt64(provider);
      default:
         throw new InvalidCastException("Conversion not supported.");
   }
}

public ushort ToUInt16(IFormatProvider provider) {
   if (temp < UInt16.MinValue || temp > UInt16.MaxValue)
```

```
         throw new OverflowException(String.Format("{0} is out of range of the UInt16 data type.", temp));
      else
         return (ushort) Math.Round(temp);
   }

   public uint ToUInt32(IFormatProvider provider) {
      if (temp < UInt32.MinValue || temp > UInt32.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the UInt32 data type.", temp));
      else
         return (uint) Math.Round(temp);
   }

   public ulong ToUInt64(IFormatProvider provider) {
      if (temp < UInt64.MinValue || temp > UInt64.MaxValue)
         throw new OverflowException(String.Format("{0} is out of range of the UInt64 data type.", temp));
      else
         return (ulong) Math.Round(temp);
   }
}

public class TemperatureCelsius : Temperature, IConvertible
{
   public TemperatureCelsius(decimal value) : base(value)
   {
   }

   // Override ToString methods.
   public override string ToString()
   {
      return this.ToString(null);
   }

   public override string ToString(IFormatProvider provider)
   {
      return temp.ToString(provider) + "°C";
   }

   // If conversionType is a implemented by another IConvertible method, call it.
   public override object ToType(Type conversionType, IFormatProvider provider) {
      // For non-objects, call base method.
      if (Type.GetTypeCode(conversionType) != TypeCode.Object) {
         return base.ToType(conversionType, provider);
      }
      else
      {
         if (conversionType.Equals(typeof(TemperatureCelsius)))
            return this;
         else if (conversionType.Equals(typeof(TemperatureFahrenheit)))
            return new TemperatureFahrenheit((decimal) this.temp * 9 / 5 + 32);
         else
            throw new InvalidCastException(String.Format("Cannot convert from Temperature to {0}.",
                                       conversionType.Name));
      }
   }
}

public class TemperatureFahrenheit : Temperature, IConvertible
{
   public TemperatureFahrenheit(decimal value) : base(value)
   {
   }

   // Override ToString methods.
   public override string ToString()
   {
      return this.ToString(null);
   }

   public override string ToString(IFormatProvider provider)
```

```
   public override String ToString(IFormatProvider provider)
   {
      return temp.ToString(provider) + "°F";
   }

   public override object ToType(Type conversionType, IFormatProvider provider)
   {
      // For non-objects, call base methood.
      if (Type.GetTypeCode(conversionType) != TypeCode.Object) {
         return base.ToType(conversionType, provider);
      }
      else
      {
         // Handle conversion between derived classes.
         if (conversionType.Equals(typeof(TemperatureFahrenheit)))
            return this;
         else if (conversionType.Equals(typeof(TemperatureCelsius)))
            return new TemperatureCelsius((decimal) (this.temp - 32) * 5 / 9);
         // Unspecified object type: throw an InvalidCastException.
         else
            throw new InvalidCastException(String.Format("Cannot convert from Temperature to {0}.",
                                       conversionType.Name));
      }
   }
}
```

```
Public MustInherit Class Temperature
    Implements IConvertible

    Protected temp As Decimal

    Public Sub New(temperature As Decimal)
        Me.temp = temperature
    End Sub

    Public Property Value As Decimal
        Get
            Return Me.temp
        End Get
        Set
            Me.temp = Value
        End Set
    End Property

    Public Overrides Function ToString() As String
        Return temp.ToString() & "º"
    End Function

    ' IConvertible implementations.
    Public Function GetTypeCode() As TypeCode Implements IConvertible.GetTypeCode
        Return TypeCode.Object
    End Function

    Public Function ToBoolean(provider As IFormatProvider) As Boolean Implements IConvertible.ToBoolean
        Throw New InvalidCastException(String.Format("Temperature-to-Boolean conversion is not supported."))
    End Function

    Public Function ToByte(provider As IFormatProvider) As Byte Implements IConvertible.ToByte
        If temp < Byte.MinValue Or temp > Byte.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the Byte data type.", temp))
        Else
            Return CByte(temp)
        End If
    End Function

    Public Function ToChar(provider As IFormatProvider) As Char Implements IConvertible.ToChar
        Throw New InvalidCastException("Temperature-to-Char conversion is not supported.")
    End Function
```

```
    Public Function ToDateTime(provider As IFormatProvider) As DateTime Implements IConvertible.ToDateTime
        Throw New InvalidCastException("Temperature-to-DateTime conversion is not supported.")
    End Function

    Public Function ToDecimal(provider As IFormatProvider) As Decimal Implements IConvertible.ToDecimal
        Return temp
    End Function

    Public Function ToDouble(provider As IFormatProvider) As Double Implements IConvertible.ToDouble
        Return CDbl(temp)
    End Function

    Public Function ToInt16(provider As IFormatProvider) As Int16 Implements IConvertible.ToInt16
        If temp < Int16.MinValue Or temp > Int16.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the Int16 data type.", temp))
        End If
        Return CShort(Math.Round(temp))
    End Function

    Public Function ToInt32(provider As IFormatProvider) As Int32 Implements IConvertible.ToInt32
        If temp < Int32.MinValue Or temp > Int32.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the Int32 data type.", temp))
        End If
        Return CInt(Math.Round(temp))
    End Function

    Public Function ToInt64(provider As IFormatProvider) As Int64 Implements IConvertible.ToInt64
        If temp < Int64.MinValue Or temp > Int64.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the Int64 data type.", temp))
        End If
        Return CLng(Math.Round(temp))
    End Function

    Public Function ToSByte(provider As IFormatProvider) As SByte Implements IConvertible.ToSByte
        If temp < SByte.MinValue Or temp > SByte.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the SByte data type.", temp))
        Else
            Return CSByte(temp)
        End If
    End Function

    Public Function ToSingle(provider As IFormatProvider) As Single Implements IConvertible.ToSingle
        Return CSng(temp)
    End Function

    Public Overridable Overloads Function ToString(provider As IFormatProvider) As String Implements
IConvertible.ToString
        Return temp.ToString(provider) & " °C"
    End Function

    ' If conversionType is a implemented by another IConvertible method, call it.
    Public Overridable Function ToType(conversionType As Type, provider As IFormatProvider) As Object
Implements IConvertible.ToType
        Select Case Type.GetTypeCode(conversionType)
            Case TypeCode.Boolean
                Return Me.ToBoolean(provider)
            Case TypeCode.Byte
                Return Me.ToByte(provider)
            Case TypeCode.Char
                Return Me.ToChar(provider)
            Case TypeCode.DateTime
                Return Me.ToDateTime(provider)
            Case TypeCode.Decimal
                Return Me.ToDecimal(provider)
            Case TypeCode.Double
                Return Me.ToDouble(provider)
            Case TypeCode.Empty
                Throw New NullReferenceException("The target type is null.")
```

```
                Throw New NullReferenceException("The target type is Null.")
            Case TypeCode.Int16
                Return Me.ToInt16(provider)
            Case TypeCode.Int32
                Return Me.ToInt32(provider)
            Case TypeCode.Int64
                Return Me.ToInt64(provider)
            Case TypeCode.Object
                ' Leave conversion of non-base types to derived classes.
                Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                                conversionType.Name))
            Case TypeCode.SByte
                Return Me.ToSByte(provider)
            Case TypeCode.Single
                Return Me.ToSingle(provider)
            Case TypeCode.String
                Return Me.ToString(provider)
            Case TypeCode.UInt16
                Return Me.ToUInt16(provider)
            Case TypeCode.UInt32
                Return Me.ToUInt32(provider)
            Case TypeCode.UInt64
                Return Me.ToUInt64(provider)
            Case Else
                Throw New InvalidCastException("Conversion not supported.")
        End Select
    End Function

    Public Function ToUInt16(provider As IFormatProvider) As UInt16 Implements IConvertible.ToUInt16
        If temp < UInt16.MinValue Or temp > UInt16.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the UInt16 data type.", temp))
        End If
        Return CUShort(Math.Round(temp))
    End Function

    Public Function ToUInt32(provider As IFormatProvider) As UInt32 Implements IConvertible.ToUInt32
        If temp < UInt32.MinValue Or temp > UInt32.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the UInt32 data type.", temp))
        End If
        Return CUInt(Math.Round(temp))
    End Function

    Public Function ToUInt64(provider As IFormatProvider) As UInt64 Implements IConvertible.ToUInt64
        If temp < UInt64.MinValue Or temp > UInt64.MaxValue Then
            Throw New OverflowException(String.Format("{0} is out of range of the UInt64 data type.", temp))
        End If
        Return CULng(Math.Round(temp))
    End Function
End Class

Public Class TemperatureCelsius : Inherits Temperature : Implements IConvertible
    Public Sub New(value As Decimal)
        MyBase.New(value)
    End Sub

    ' Override ToString methods.
    Public Overrides Function ToString() As String
        Return Me.ToString(Nothing)
    End Function

    Public Overrides Function ToString(provider As IFormatProvider) As String
        Return temp.ToString(provider) + "°C"
    End Function

    ' If conversionType is a implemented by another IConvertible method, call it.
    Public Overrides Function ToType(conversionType As Type, provider As IFormatProvider) As Object
        ' For non-objects, call base method.
        If Type.GetTypeCode(conversionType) <> TypeCode.Object Then
            Return MyBase.ToType(conversionType, provider)
        Else
```

```
        Else
            If conversionType.Equals(GetType(TemperatureCelsius)) Then
                Return Me
            ElseIf conversionType.Equals(GetType(TemperatureFahrenheit))
                Return New TemperatureFahrenheit(CDec(Me.temp * 9 / 5 + 32))
                ' Unspecified object type: throw an InvalidCastException.
            Else
                Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                                conversionType.Name))
            End If
        End If
    End Function
End Class

Public Class TemperatureFahrenheit : Inherits Temperature : Implements IConvertible
    Public Sub New(value As Decimal)
        MyBase.New(value)
    End Sub

    ' Override ToString methods.
    Public Overrides Function ToString() As String
        Return Me.ToString(Nothing)
    End Function

    Public Overrides Function ToString(provider As IFormatProvider) As String
        Return temp.ToString(provider) + "°F"
    End Function

    Public Overrides Function ToType(conversionType As Type, provider As IFormatProvider) As Object
        ' For non-objects, call base method.
        If Type.GetTypeCode(conversionType) <> TypeCode.Object Then
            Return MyBase.ToType(conversionType, provider)
        Else
            ' Handle conversion between derived classes.
            If conversionType.Equals(GetType(TemperatureFahrenheit)) Then
                Return Me
            ElseIf conversionType.Equals(GetType(TemperatureCelsius))
                Return New TemperatureCelsius(CDec((MyBase.temp - 32) * 5 / 9))
                ' Unspecified object type: throw an InvalidCastException.
            Else
                Throw New InvalidCastException(String.Format("Cannot convert from Temperature to {0}.", _
                                                conversionType.Name))
            End If
        End If
    End Function
End Class
```

下面的示例演示对这些 IConvertible 实现的多个调用，以实现 `TemperatureCelsius` 对象和 `TemperatureFahrenheit` 对象之间的相互转换。

```
TemperatureCelsius tempC1 = new TemperatureCelsius(0);
TemperatureFahrenheit tempF1 = (TemperatureFahrenheit) Convert.ChangeType(tempC1,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempC1, tempF1);
TemperatureCelsius tempC2 = (TemperatureCelsius) Convert.ChangeType(tempC1, typeof(TemperatureCelsius),
null);
Console.WriteLine("{0} equals {1}.", tempC1, tempC2);
TemperatureFahrenheit tempF2 = new TemperatureFahrenheit(212);
TemperatureCelsius tempC3 = (TemperatureCelsius) Convert.ChangeType(tempF2, typeof(TemperatureCelsius),
null);
Console.WriteLine("{0} equals {1}.", tempF2, tempC3);
TemperatureFahrenheit tempF3 = (TemperatureFahrenheit) Convert.ChangeType(tempF2,
typeof(TemperatureFahrenheit), null);
Console.WriteLine("{0} equals {1}.", tempF2, tempF3);
// The example displays the following output:
//       0°C equals 32°F.
//       0°C equals 0°C.
//       212°F equals 100°C.
//       212°F equals 212°F.
```

```
Dim tempC1 As New TemperatureCelsius(0)
Dim tempF1 As TemperatureFahrenheit = CType(Convert.ChangeType(tempC1, GetType(TemperatureFahrenheit),
Nothing), TemperatureFahrenheit)
Console.WriteLine("{0} equals {1}.", tempC1, tempF1)
Dim tempC2 As TemperatureCelsius = CType(Convert.ChangeType(tempC1, GetType(TemperatureCelsius), Nothing),
TemperatureCelsius)
Console.WriteLine("{0} equals {1}.", tempC1, tempC2)
Dim tempF2 As New TemperatureFahrenheit(212)
Dim tempC3 As TEmperatureCelsius = CType(Convert.ChangeType(tempF2, GEtType(TemperatureCelsius), Nothing),
TemperatureCelsius)
Console.WriteLine("{0} equals {1}.", tempF2, tempC3)
Dim tempF3 As TemperatureFahrenheit = CType(Convert.ChangeType(tempF2, GetType(TemperatureFahrenheit),
Nothing), TemperatureFahrenheit)
Console.WriteLine("{0} equals {1}.", tempF2, tempF3)
' The example displays the following output:
'       0°C equals 32°F.
'       0°C equals 0°C.
'       212°F equals 100°C.
'       212°F equals 212°F.
```

TypeConverter 类

.NET 还允许你通过下面的方法为自定义类型定义类型转换器：扩展 System.ComponentModel.TypeConverter 类，然后通过 System.ComponentModel.TypeConverterAttribute 特性将类型转换器与该类型关联。下表列出了此方法与为自定义类型实现 IConvertible 接口之间的差异。

> **NOTE**
> 只能为已定义了类型转换器的自定义类型提供设计时支持。

TYPECONVERTER	ICONVERTIBLE
通过从 TypeConverter 派生单独的类来为自定义类型实现。此派生类通过应用 TypeConverterAttribute 特性与自定义类型关联。	由自定义类型实现，以执行转换。该类型的用户必须对该类型调用 IConvertible 转换方法。
在设计时和运行时都可以使用。	只能在运行时使用。

TYPECONVERTER	ICONVERTIBLE
使用反射；因此，比 IConvertible 所启用的转换慢。	不使用反射。
允许自定义类型和其他数据类型间的双向类型转换。例如，为 TypeConverter 定义的 `MyType` 允许从 `MyType` 转换为 String 以及从 String 转换为 `MyType`。	允许从自定义类型转换为其他数据类型，但不允许从其他数据类型转换为自定义类型。

有关使用类型转换器执行转换的更多信息，请参见 System.ComponentModel.TypeConverter。

请参阅

- System.Convert
- IConvertible
- 类型转换表

.NET 中的类型转换表

2021/11/16 •

当一种类型的值转换为大小相等或更大的另一类型时，将发生扩大转换。当一种类型的值转换为较小的另一种类型时，将发生收缩转换。本主题中的表格解释了这两种转换类型的行为。

扩大转换

下表列出了执行不会导致信息丢失的扩大转换。

Byte	UInt16、Int16、UInt32、Int32、UInt64、Int64、Single、Double、Decimal
SByte	Int16、Int32、Int64、Single、Double、Decimal
Int16	Int32, Int64, Single, Double, Decimal
UInt16	UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Char	UInt16, UInt32, Int32, UInt64, Int64, Single, Double, Decimal
Int32	Int64, Double, Decimal
UInt32	Int64、UInt64、Double、Decimal
Int64	Decimal
UInt64	Decimal
Single	Double

一些目标为 Single 或 Double 的扩大转换可能会导致精度丢失。下面的表格描述了有时会导致信息丢失的扩大转换。

Int32	Single
UInt32	Single
Int64	Single, Double
UInt64	Single, Double
Decimal	Single, Double

收缩转换

目标为 Single 或 Double 的收缩转换可能会导致信息丢失。如果目标类型无法正确表达源类型的大小，则结果类型将设置为常数 `PositiveInfinity` 或 `NegativeInfinity`。`PositiveInfinity` 是正数除以 0 的结果，也在 Single 或 Double 的值大于 `MaxValue` 字段的值时返回。`NegativeInfinity` 是负数除以 0 的结果，也在 Single 或 Double 的值小于 `MinValue` 字段的值时返回。从 Double 转换到 Single 可能会导致 `PositiveInfinity` 或 `NegativeInfinity`。

收缩转换还可能导致其他数据类型的信息丢失。不过，如果要转换的类型值不在目标类型的 `MaxValue` 和 `MinValue` 字段指定的范围内，就会抛出 OverflowException，并且运行时会检查转换，以确保目标类型的值不超出它的 `MaxValue` 或 `MinValue`。始终以这种方式检查使用 System.Convert 类执行的转换。

下表列出了使用 System.Convert 抛出 OverflowException 的转换，或要转换类型的值不在生成类型的定义范围内的任何已检查转换。

Byte	SByte
SByte	Byte、UInt16、UInt32、UInt64
Int16	Byte, SByte, UInt16
UInt16	Byte, SByte, Int16
Int32	Byte, SByte, Int16, UInt16,UInt32
UInt32	Byte, SByte, Int16, UInt16, Int32
Int64	Byte, SByte, Int16, UInt16, Int32,UInt32,UInt64
UInt64	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64
Decimal	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Single	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64
Double	Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64

另请参阅

- System.Convert
- .NET 中的类型转换

在匿名类型和元组类型之间进行选择

2021/11/16 •

选择适当的类型需要考虑其可用性和性能，还要与其他类型相比较并进行权衡。匿名类型自 C# 3.0 即可使用，而泛型 System.Tuple<T1,T2> 类型是随 .NET Framework 4.0 引入的。自那时起，通过语言级别支持引入了多个新选项，例如 System.ValueTuple<T1,T2>（它名副其实，提供具有匿名类型灵活性的值类型）。在本文中，你将了解何时应选择一种类型而不是另一种。

可用性和功能

匿名类型是在 C# 3.0 中随语言集成查询 (LINQ) 表达式引入的。使用 LINQ，开发人员通常会将查询结果投影到匿名类型中，这些类型保存着来自其所处理对象的几个选择属性。请考虑以下示例，它实例化 DateTime 对象的一个数组，然后循环访问它们，并将其投影到具有两个属性的匿名类型。

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var anonymous in
             dates.Select(
                 date => new { Formatted = $"{date:MMM dd, yyyy hh:mm zzz}", date.Ticks }))
{
    Console.WriteLine($"Ticks: {anonymous.Ticks}, formatted: {anonymous.Formatted}");
}
```

使用 `new` 运算符来实例化匿名类型，从声明推断属性名称和类型。如果同一程序集中的两个或多个匿名对象初始值设定项指定了属性序列，这些属性采用相同顺序且具有相同的名称和类型，则编译器将对象视为相同类型的实例。它们共享同一编译器生成的类型信息。

以上 C# 代码片段投影具有两个属性的匿名类型，非常类似于以下编译器生成的 C# 类：

```
internal sealed class f__AnonymousType0
{
    public string Formatted { get; }
    public long Ticks { get; }

    public f__AnonymousType0(string formatted, long ticks)
    {
        Formatted = formatted;
        Ticks = ticks;
    }
}
```

有关详细信息，请参阅匿名类型。投影到 LINQ 查询时，元组具有相同的功能，你可以选择添加属性到元组。这些元组会贯穿查询，就像匿名类型一样。现在请考虑以下使用 `System.Tuple<string, long>` 的示例。

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var tuple in
            dates.Select(
                date => new Tuple<string, long>($"{date:MMM dd, yyyy hh:mm zzz}", date.Ticks)))
{
    Console.WriteLine($"Ticks: {tuple.Item2}, formatted: {tuple.Item1}");
}
```

在 System.Tuple<T1,T2> 中，实例公开编号项属性，如 `Item1` 和 `Item2`。由于属性名称只提供序号，因此这些属性名称让人更难理解属性值的意图。此外，`System.Tuple` 类型是引用 `class` 类型。而 System.ValueTuple<T1,T2> 是值 `struct` 类型。以下 C# 代码片段使用 `ValueTuple<string, long>` 投影。在此过程中，它使用文字语法进行分配。

```
var dates = new[]
{
    DateTime.UtcNow.AddHours(-1),
    DateTime.UtcNow,
    DateTime.UtcNow.AddHours(1),
};

foreach (var (formatted, ticks) in
            dates.Select(
                date => (Formatted: $"{date:MMM dd, yyyy at hh:mm zzz}", date.Ticks)))
{
    Console.WriteLine($"Ticks: {ticks}, formatted: {formatted}");
}
```

有关元组的详细信息，请参阅元组类型（C# 参考）或元组 (Visual Basic)。

以上示例在功能上是等效的；但其可用性和基础实现存在细微差异。

权衡

你可能希望始终使用 ValueTuple（而不是 Tuple）和匿名类型，但应进行一些权衡考虑。ValueTuple 类型是可变的，而 Tuple 是只读的。匿名类型可用于表达式树，而元组不行。下表概述了一些关键区别。

主要区别

“[illegible]”	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
匿名类型	`internal`	`class`	✔	✘	✔
Tuple	`public`	`class`	✘	✘	✔
ValueTuple	`public`	`struct`	✔	✔	✘

序列化

选择类型时的一个重要考量是，是否需要对其进行序列化。序列化是将对象状态转换为可保持或传输的形式的过程。有关详细信息，请参阅序列化。当序列化非常重要时，创建 `class` 或 `struct` 优于使用匿名类型或元组类型。

性能

这些类型各自的性能取决于场景。主要影响涉及分配和复制之间的权衡。在大多数情况下，影响很小。如果可能出现重大影响，应采取措施来通知决策者。

结束语

开发人员在元组和匿名类型之间进行选择时，需要考虑几个因素。一般来说，如果不使用表达式树，并且你熟悉元组语法，请选择 ValueTuple，因为它们提供可灵活命名属性的值类型。如果使用表达式树并且想要命名属性，请选择匿名类型。否则，请使用 Tuple。

请参阅

- 匿名类型
- 表达式树
- 元组类型（C# 参考）
- 元组 (Visual Basic)
- 类型设计准则

框架库

2021/11/16 •

.NET 提供广泛的标准类库集，称为基类库（核心集）或框架类库（完整集）。这些库为许多常见和特定于与应用的类型、算法和实用程序功能提供实现。商用库和社区库都构建在框架类库的顶层，可让用户针对各种计算任务轻松使用现成的库。

每个 .NET 实现随附了这些库的子集。任何 .NET 实现预期都要使用基类库 (BCL) API，原因有两种：开发人员需要这些 API，流行的库需要这些 API 才能运行。位于 BCL 上层的应用特定的库（例如 ASP.NET）并不能在所有 .NET 实现中使用。

基类库

BCL 提供最基本的类型和实用工具功能，是其他所有 .NET 类库的基础。BCL 旨在提供常规实现，而不偏向任何工作负载。性能是一个重要的考虑因素，因为应用可能会优先使用特定的策略，例如，优先考虑低延迟而不是高吞吐量，或者优先考虑低内存而不是低 CPU 使用率。BCL 在总体上可以保证高性能，同时会根据不同的性能考虑因素采取折衷方案。对于大多数应用而言，这种方案相当成功。

基元和其他基本类型

.NET 包含一组在所有程序中使用（使用程度或大或小）的基本类型。这些类型包含数据，例如数字、字符串、字节和任意对象。C# 语言包括这些类型的关键字。下面列出了这些类型的一组示例，以及匹配的 C# 关键字。

[illegible]	C# [illegible]	[illegible]
System.Object	对象	CLR 类型系统中的最基本基类。它位于类型层次结构的根级别。
System.Int16	short	16 位带符号整数类型。也存在无符号 UInt16。
System.Int32	int	32 位带符号整数类型。也存在无符号 UInt32。
System.Single	float	32 位浮点类型。
System.Decimal	decimal	128 位十进制类型。
System.Byte	byte	表示内存字节的无符号 8 位整数。
System.Boolean	bool	表示 `true` 或 `false` 的布尔类型。
System.Char	char	表示 Unicode 字符的 16 位数字类型。
System.String	string	表示一系列字符。与 `char[]` 不同，但会针对 `string` 中的每个 `char` 启用索引。

数据结构

.NET 包含一组数据结构，这些结构是许多 .NET 应用的工作主力。它们主要是集合，不过也包括其他类型。

- Array- 表示可通过索引访问的强类型对象的数组。具有与构造相符的固定大小。
- List<T> - 表示可通过索引访问的对象的强类型列表。可根据需要自动调整大小。
- Dictionary<TKey,TValue> -表示根据键编制索引的值的集合。可以通过键访问值。可根据需要自动调整大小。
- Uri - 提供统一资源标识符 (URI) 的对象表示形式和对 URI 各部分的轻松访问。
- DateTime - 表示时间上的一刻，通常以日期和当天的时间表示。

实用工具 API

.NET 包含一组可为许多重要任务提供功能的实用工具 API。

- HttpClient - 用于发送 HTTP 请求以及从 URI 所标识资源接收 HTTP 响应的 API。
- XDocument - 用于配合 LINQ 加载和查询 XML 文档的 API。
- StreamReader - 用于读取文件的 API。
- StreamWriter - 用于写入文件的 API。

应用模型 API

有许多应用模型可与 .NET 配合使用，例如：

- ASP.NET - 用于构建网站和服务的 Web 框架。受 Windows、Linux 和 macOS 的支持（取决于 ASP.NET 版本）。
- Xamarin - 用于通过 .NET 和 C# 构建 Android 和 iOS 应用的应用平台。在 Windows 和 macOS 上受支持。
- Windows 桌面 - 包括 Windows Presentation Foundation (WPF) 和 Windows 窗体。

.NET 类库概述

2021/11/16 •

.NET API 包括可加快和优化开发过程并提供对系统功能的访问的类、接口、委托和值类型。为了便于语言之间进行交互操作，大多数 .NET 类型都符合 CLS，因而可在编译器符合公共语言规范 (CLS) 的任何编程语言中使用。

.NET 类型是生成 .NET 应用程序、组件和控件的基础。.NET 包括的类型可执行下列功能：

- 表示基础数据类型和异常。
- 封装数据结构。
- 执行 I/O。
- 访问关于加载类型的信息。
- 调用 .NET 安全检查。
- 提供数据访问、多客户端 GUI 和服务器控制的客户端 GUI。

.NET 提供了一组丰富的接口以及抽象类和具体（非抽象）类。可以按原样使用这些具体的类，或者在多数情况下从这些类派生你自己的类。若要使用接口的功能，既可以创建实现接口的类，也可以从某个实现接口的 .NET 类中派生类。

命名约定

.NET 类型使用点语法命名方案，该方案隐含了层次结构的意思。此技术将相关类型分为不同的命名空间组，以便可以更容易地搜索和引用它们。全名的第一部分（最右边的点之前的内容）是命名空间名。全名的最后一部分是类型名。例如，`System.Collections.Generic.List<T>` 表示 `List<T>` 类型，属于 `System.Collections.Generic` 命名空间。可使用 System.Collections.Generic 中的类型使用泛型集合。

此命名方案使扩展 .NET 的库开发人员可以轻松创建分层类型组，并用一致的、带有提示性的方式对其进行命名。它还允许用全名（即命名空间和类型名称）明确地标识类型，这样可以防止类型名称发生冲突。库开发人员在创建其命名空间的名称时应使用以下约定：

CompanyName.TechnologyName

例如，`Microsoft.Word` 命名空间就符合此原则。

利用命名模式将相关类型分组为命名空间是生成和记录类库的一种有用的方式。但是，此命名方案对可见性、成员访问、继承、安全性或绑定无效。一个命名空间可以被划分在多个程序集中，而单个程序集可以包含来自多个命名空间的类型。程序集为公共语言运行时中的版本控制、部署、安全性、加载和可见性提供外形结构。

有关命名空间和类型名的更多信息，请参阅通用类型系统。

系统命名空间

System 命名空间是 .NET 中基本类型的根命名空间。此命名空间包括表示由所有应用程序使用的基本数据类型的类，例如 Object（继承层次结构的根）、Byte、Char、Array、Int32 和 String。在这些类型中，有许多与编程语言所使用的基元数据类型相对应。当使用 .NET 类型编写代码时，可以在应使用 .NET 基础数据类型时使用编程语言的相应关键字。

下表列出了 .NET 提供的基类型，并对每种类型进行了简单描述，同时指出了 Visual Basic、C#、C++ 和 F# 中的相应类型。

类	类	类	Visual Basic 类型	C# 类型	C++/CLI 类型	F# 类型
整数	Byte	8 位无符号整数。	`Byte`	`byte`	`unsigned char`	`byte`
	SByte	8 位有符号整数。 不符合 CLS。	`SByte`	`sbyte`	`char` 或 `signed char`	`sbyte`
	Int16	16 位带符号整数。	`Short`	`short`	`short`	`int16`
	Int32	32 位带符号整数。	`Integer`	`int`	`int` 或 `long`	`int`
	Int64	64 位带符号整数。	`Long`	`long`	`__int64`	`int64`
	UInt16	16 位无符号整数。 不符合 CLS。	`UShort`	`ushort`	`unsigned short`	`uint16`
	UInt32	32 位无符号整数。 不符合 CLS。	`UInteger`	`uint`	`unsigned int` 或 `unsigned long`	`uint32`
	UInt64	64 位无符号整数。 不符合 CLS。	`ULong`	`ulong`	`unsigned __int64`	`uint64`
浮点	Half	半精度(16 位)浮点数。				
	Single	单精度(32 位)浮点数字。	`Single`	`float`	`float`	`float32` 或 `single`
	Double	双精度(64 位)浮点数字。	`Double`	`double`	`double`	`float` 或 `double`
逻辑运算	Boolean	布尔值(真或假)。	`Boolean`	`bool`	`bool`	`bool`
其他	Char	Unicode(16 位)字符。	`Char`	`char`	`wchar_t`	`char`
	Decimal	十进制(128 位)值。	`Decimal`	`decimal`	`Decimal`	`decimal`

类	类	类	VISUAL BASIC 类型	C# 类型	C++/CLI 类型	F# 类型
	IntPtr	大小取决于基础平台(32 位平台上为 32 位值，64 位平台上为 64 位值)的有符号整数。		`nint`		`unativeint`
	UIntPtr	大小取决于基础平台的无符号整数(32 位平台上为 32 位值，64 位平台上为 64 位值)。 不符合 CLS。		`nuint`		`unativeint`
	Object	对象层次结构的根。	`Object`	`object`	`Object^`	`obj`
	String	Unicode 字符的不变的定长串。	`String`	`string`	`String^`	`string`

除了基本数据类型外，System 命名空间还包含 100 多个类，范围从处理异常的类到处理核心运行时概念的类，如应用程序域和垃圾回收器。System 命名空间还包含许多二级命名空间。

有关命名空间的详细信息，请使用 .NET API 浏览器来浏览 .NET 类库。API 参考文档提供了有关每个名称空间及其类型和每个成员的文档。

请参阅

- 常规类型系统
- .NET API 浏览器
- 概述

.NET 中的泛型

2021/11/16 •

借助泛型，你可以根据要处理的精确数据类型定制方法、类、结构或接口。例如，不使用允许键和值为任意类型的 Hashtable 类，而使用 Dictionary<TKey,TValue> 泛型类并指定允许的密钥和值类型。泛型的优点包括：代码的可重用性增加，类型安全性提高。

定义和使用泛型

泛型是为所存储或使用的一个或多个类型具有占位符（类型形参）的类、结构、接口和方法。泛型集合类可以将类型形参用作其存储的对象类型的占位符；类型形参呈现为其字段的类型和其方法的参数类型。泛型方法可将其类型形参用作其返回值的类型或用作其形参之一的类型。以下代码举例说明了一个简单的泛型类定义。

```
generic<typename T>
public ref class Generics
{
public:
    T Field;
};
```

```
public class Generic<T>
{
    public T Field;
}
```

```
Public Class Generic(Of T)
    Public Field As T

End Class
```

创建泛型类的实例时，指定用于替代类型形参的实际类型。在类型形参出现的每一处位置用选定的类型进行替代，这会建立一个被称为构造泛型类的新泛型类。你将得到根据你选择的类型而定制的类型安全类，如以下代码所示。

```
static void Main()
{
    Generics<String^>^ g = gcnew Generics<String^>();
    g->Field = "A string";
    //...
    Console::WriteLine("Generics.Field           = \"{0}\"", g->Field);
    Console::WriteLine("Generics.Field.GetType() = {0}", g->Field->GetType()->FullName);
}
```

```
public static void Main()
{
    Generic<string> g = new Generic<string>();
    g.Field = "A string";
    //...
    Console.WriteLine("Generic.Field           = \"{0}\"", g.Field);
    Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName);
}
```

```
Public Shared Sub Main()
    Dim g As New Generic(Of String)
    g.Field = "A string"
    '...
    Console.WriteLine("Generic.Field           = ""{0}""", g.Field)
    Console.WriteLine("Generic.Field.GetType() = {0}", g.Field.GetType().FullName)
End Sub
```

泛型术语

介绍 .NET 中的泛型需要用到以下术语：

- *泛型类型定义* 是用作模板的类、结构或接口声明，带有可包含或使用的类型的占位符。例如，System.Collections.Generic.Dictionary<TKey,TValue> 类可以包含两种类型：密钥和值。由于泛型类型定义只是一个模板，所以你无法创建作为泛型类型定义的类、结构或接口的实例。
- *泛型类型参数*（或 *类型参数*）是泛型类型或方法定义中的占位符。System.Collections.Generic.Dictionary<TKey,TValue> 泛型类型具有两个类型形参 `TKey` 和 `TValue`，它们分别代表密钥和值的类型。
- *构造泛型类型*（或 *构造类型*）是为泛型类型定义的泛型类型形参指定类型的结果。
- *泛型类型实参* 是被泛型类型形参所替代的任何类型。
- 常见术语 *泛型类型* 包括构造类型和泛型类型定义。
- 借助泛型类型参数的 *协变* 和 *逆变*，可以使用类型自变量的派生程度比目标构造类型更高（协变）或更低（逆变）的构造泛型类型。协变和逆变统称为“变体”。有关详细信息，请参阅协变和逆变。
- *约束* 是对泛型类型参数的限制。例如，你可能会将一个类型形参限制为实现 System.Collections.Generic.IComparer<T> 泛型接口的类型，以确保可对该类型的实例进行排序。此外，你还可以将类型形参限制为具有特定基类、具有无参数构造函数或作为引用类型或值类型的类型。泛型类型的用户不能替换不满足约束条件的类型实参。
- *泛型方法定义* 是具有两个形参列表的方法：泛型类型形参列表和形参列表。类型形参可作为返回类型或形参类型出现，如以下代码所示。

```
generic<typename T>
T Generic(T arg)
{
    T temp = arg;
    //...
    return temp;
}
```

```
T Generic<T>(T arg)
{
    T temp = arg;
    //...
    return temp;
}
```

```
Function Generic(Of T)(ByVal arg As T) As T
    Dim temp As T = arg
    '...
    Return temp
End Function
```

泛型方法可出现在泛型或非泛型类型中。值得注意的是，方法不会仅因为它属于泛型类型或甚至因为它有类型为封闭类型泛型参数的形参而成为泛型方法。只有当方法有属于自己的类型形参列表时才是泛型方法。在以下代码中，只有方法 `G` 是泛型方法。

```
ref class A
{
    generic<typename T>
    T G(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
};
generic<typename T>
ref class Generic
{
    T M(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
};
```

```
class A
{
    T G<T>(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
class Generic<T>
{
    T M(T arg)
    {
        T temp = arg;
        //...
        return temp;
    }
}
```

```
Class A
    Function G(Of T)(ByVal arg As T) As T
        Dim temp As T = arg
        '...
        Return temp
    End Function
End Class
Class Generic(Of T)
    Function M(ByVal arg As T) As T
        Dim temp As T = arg
        '...
        Return temp
    End Function
End Class
```

泛型的利与弊

使用泛型集合和委托有很多好处：

- 类型安全。泛型将类型安全的负担从你那里转移到编译器。没有必要编写代码来测试正确的数据类型，因为它会在编译时强制执行。降低了强制类型转换的必要性和运行时错误的可能性。
- 代码更少且可以更轻松地重用代码。无需从基类型继承，无需重写成员。例如，可立即使用 LinkedList<T> 。例如，你可以使用下列变量声明来创建字符串的链接列表：

```
LinkedList<String^>^ llist = gcnew LinkedList<String^>();
```

```
LinkedList<string> llist = new LinkedList<string>();
```

```
Dim llist As New LinkedList(Of String)()
```

- 性能更好。泛型集合类型通常能更好地存储和操作值类型，因为无需对值类型进行装箱。
- 泛型委托可以在无需创建多个委托类的情况下进行类型安全的回调。例如，Predicate<T> 泛型委托允许你创建一种为特定类型实现你自己的搜索标准的方法并将你的方法与 Array 类型比如 Find、FindLast和 FindAll方法一起使用。
- 泛型简化动态生成的代码。使用具有动态生成的代码的泛型时，无需生成类型。这会增加方案数量，在这些方案中你可以使用轻量动态方法而非生成整个程序集。有关详细信息，请参阅如何：定义和执行动态方法 和 DynamicMethod。

以下是泛型的一些局限：

- 泛型类型可从多数基类中派生，如 MarshalByRefObject（约束可用于要求泛型类型形参派生自诸如 MarshalByRefObject的基类）。不过，.NET 不支持上下文绑定的泛型类型。泛型类型可派生自 ContextBoundObject，但尝试创建该类型实例会导致 TypeLoadException。
- 枚举不能具有泛型类型形参。枚举偶尔可为泛型（例如，因为它嵌套在被定义使用 Visual Basic、C# 或 C++ 的泛型类型中）。有关详细信息，请参阅 “常规类型系统”中的“枚举”。
- 轻量动态方法不能是泛型。
- 在 Visual Basic、C# 和 C++ 中，包含在泛型类型中的嵌套类型不能被实例化，除非已将类型分配给所有封闭类型的类型形参。另一种说法是：在反射中，定义使用这些语言的嵌套类型包括其所有封闭类型的类型形参。这使封闭类型的类型形参可在嵌套类型的成员定义中使用。有关详细信息，请参阅 MakeGenericType中的“嵌套类型”。

  > **NOTE**
  > 通过在动态程序集中触发代码或通过使用 Ilasm.exe (IL 汇编程序) 定义的嵌套类型不需要包括其封闭类型的类型参数；然而，如果不包括，类型参数就不会在嵌套类的范围内。

  有关详细信息，请参阅 MakeGenericType中的“嵌套类型”。

类库和语言支持

.NET 在以下命名空间中提供了大量泛型集合类：

- System.Collections.Generic 命名空间包含 .NET 提供的大部分泛型集合类型（如 List<T> 和 Dictionary<TKey,TValue> 泛型类）。
- System.Collections.ObjectModel 命名空间包含向类用户公开对象模型的其他泛型集合类型（如

ReadOnlyCollection<T> 泛型类)。

System 命名空间提供实现排序和等同性比较的泛型接口，还提供事件处理程序、转换和搜索谓词的泛型委托类型。

已将对泛型的支持添加到：System.Reflection 命名空间(以检查泛型类型和泛型方法)、System.Reflection.Emit(以发出包含泛型类型和方法的动态程序集)和 System.CodeDom (以生成包括泛型的源图)。

公共语言运行时提供了新的操作码和前缀来支持 Microsoft 中间语言 (MSIL) 中的泛型类型，包括 Stelem、Ldelem、Unbox_Any、Constrained和 Readonly。

Visual C++、C# 和 Visual Basic 都对定义和使用泛型提供完全支持。有关语言支持的详细信息，请参阅 Visual Basic 中的泛型类型、泛型简介和 Visual C++ 中的泛型概述。

嵌套类型和泛型

嵌套在泛型类型中的类型可取决于封闭泛型类型的类型参数。公共语言运行时将嵌套类型看作泛型，即使它们不具有自己的泛型类型形参。创建嵌套类型的实例时，必须指定所有封闭泛型类型的类型实参。

相关主题

TITLE	II
.NET 中的泛型集合	介绍了 .NET 中的泛型集合类和其他泛型类型。
用于控制数组和列表的泛型委托	描述用于转换、搜索谓词以及要对数组或集合中的元素执行的操作的泛型委托。
泛型接口	描述跨泛型类型系列提供通用功能的泛型接口。
协变和逆变	描述泛型类型实参中的协变和逆变。
常用的集合类型	总结了 .NET 中集合类型(包括泛型类型)的特征和使用方案。
何时使用泛型集合	描述用于确定何时使用泛型集合类型的一般规则。
如何:使用反射发出定义泛型类型	解释如何生成包括泛型类型和方法的动态程序集。
Generic Types in Visual Basic	为 Visual Basic 用户描述泛型功能，包括有关使用和定义泛型类型的帮助主题。
泛型介绍	为 C# 用户概述定义和使用泛型类型。
Visual C++ 中的泛型概述	为 C++ 用户描述泛型功能，包括泛型和模板之间的差异。

参考

System.Collections.Generic

System.Collections.ObjectModel

System.Reflection.Emit.OpCodes

泛型类型概述

2021/11/16 •

在 .NET 中，开发人员随时会使用泛型，有时隐式使用，有时显式使用。在 .NET 中使用 LINQ 时，你是否曾经注意到，使用的正是 IEnumerable<T>？ 或者，你是否曾经看到过有关使用实体框架来与数据库通信的“泛型存储库”在线示例，其中的大多数方法返回 `IQueryable<T>`？ 你可能很想知道，这些示例中的 T 是什么意思，为什么要使用它？

泛型在 .NET Framework 2.0 中首次引入，它本质上是一个“代码模板”，可让开发者定义类型安全数据结构，无需处理实际数据类型。例如，List<T> 是一个可以声明的泛型集合，可与 `List<int>`、`List<string>` 或 `List<Person>` 等任何类型结合使用。

为方便理解泛型的作用，让我们看看添加泛型之前和之后的一个特定类：ArrayList。在 .NET Framework 1.0 中，`ArrayList` 元素属于 Object 类型。添加到集合的任何元素都会以静默方式转换为 `Object`。从列表读取元素时，会发生相同的情况。此过程称为装箱和取消装箱，它会影响性能。但除了性能之外，在编译时无法确定列表中的数据的类型，这会形成一些脆弱的代码。泛型解决了此问题，它可以定义每个列表实例将要包含的数据类型。例如，只能将整数添加到 `List<int>`，只能将人员添加到 `List<Person>`。

泛型还可以在运行时使用。运行时知道你要使用的数据结构类型，并可以更高效地将数据结构存储在内存中。

下面的示例是一个小程序，演示了在运行时如何有效地了解数据结构类型：

```
  using System;
  using System.Collections;
  using System.Collections.Generic;
  using System.Diagnostics;

  namespace GenericsExample {
    class Program {
      static void Main(string[] args) {
        //generic list
        List<int> ListGeneric = new List<int> { 5, 9, 1, 4 };
        //non-generic list
        ArrayList ListNonGeneric = new ArrayList { 5, 9, 1, 4 };
        // timer for generic list sort
        Stopwatch s = Stopwatch.StartNew();
        ListGeneric.Sort();
        s.Stop();
        Console.WriteLine($"Generic Sort: {ListGeneric}  \n Time taken: {s.Elapsed.TotalMilliseconds}ms");

        //timer for non-generic list sort
        Stopwatch s2 = Stopwatch.StartNew();
        ListNonGeneric.Sort();
        s2.Stop();
        Console.WriteLine($"Non-Generic Sort: {ListNonGeneric}  \n Time taken:
{s2.Elapsed.TotalMilliseconds}ms");
        Console.ReadLine();
      }
    }
  }
```

此程序将生成类似下面的输出：

```
Generic Sort: System.Collections.Generic.List`1[System.Int32]
 Time taken: 0.0034ms
Non-Generic Sort: System.Collections.ArrayList
 Time taken: 0.2592ms
```

在此处可以看到的第一个优点是，泛型列表的排序比非泛型列表要快得多。还可以看到，泛型列表的类型是不同的 ([System.Int32])，而非泛型列表的类型已通用化。由于运行时知道泛型 `List<int>` 的类型是 Int32，因此可以将列表元素存储在内存中的基础整数数组内；而非泛型 `ArrayList` 必须将每个列表元素强制转换为对象。如本示例中所示，多余的强制转换会占用时间，降低列表排序的速度。

运行时知道泛型类型的另一个优点是可以改善调试体验。在 C# 中调试泛型时，可以知道数据结构中每个元素的类型。如果不使用泛型，则无从知道每个元素的类型是什么。

请参阅

- C# 编程指南 - 泛型

.NET 中的泛型集合

2021/11/16 •

.NET 类库提供了许多 System.Collections.Generic 和 System.Collections.ObjectModel 命名空间中的泛型集合类。若要详细了解这些类，请参阅常用集合类型。

System.Collections.Generic

许多泛型集合类型均为非泛型类型的直接模拟。Dictionary<TKey,TValue> 是 Hashtable 的泛型版本；它使用枚举的泛型结构 KeyValuePair<TKey,TValue> 而不是 DictionaryEntry。

List<T> 是 ArrayList 的泛型版本。存在响应非泛型版本的泛型 Queue<T> 和 Stack<T> 类。

存在 SortedList<TKey,TValue> 的泛型和非泛型版本。这两个版本均为字典和列表的混合。SortedDictionary<TKey,TValue> 泛型类是一个纯字典，并且没有任何非泛型对应项。

LinkedList<T> 泛型类是真正的链接列表。它没有任何非泛型对应项。

System.Collections.ObjectModel

Collection<T> 泛型类提供用于派生自己的泛型集合类型的基类。ReadOnlyCollection<T> 类提供了任何从实现 IList<T> 泛型接口的类型生成只读集合的简便方法。KeyedCollection<TKey,TItem> 泛型类提供了存储包含其自己的键的对象的方法。

其他泛型类型

Nullable<T> 泛型结构允许使用值类型，如同它们可分配 `null`。这在处理数据库查询时很有用，其中字段包含可能丢失的值类型。泛型类型参数可为任意值类型。

> **NOTE**
> 在 C# 和 Visual Basic 中，无需显式使用 Nullable<T>，因为语言具有可以为 null 类型的语法。请参阅可为 null 的值类型 (C# 引用)和可为 null 的值类型 (Visual Basic)。

ArraySegment<T> 泛型结构提供了分隔任何类型的从零开始的一维数组内的一系列元素的方法。泛型类型参数是数组中元素的类型。

如果你的事件采用 .NET 所使用的事件处理模式，则 EventHandler<TEventArgs> 泛型委托不需要声明委托类型来处理事件。例如，假设已创建从 EventArgs 派生的 `MyEventArgs` 类，以包含事件的数据。则可以声明此事件，如下所示：

```
public:
    event EventHandler<MyEventArgs^>^ MyEvent;
```

```
public event EventHandler<MyEventArgs> MyEvent;
```

```
Public Event MyEvent As EventHandler(Of MyEventArgs)
```

请参阅

- System.Collections.Generic
- System.Collections.ObjectModel
- 泛型
- 用于控制数组和列表的泛型委托
- 泛型接口

用于操作数组和列表的泛型委托

2021/11/16 •

此主题概述了用于转换的泛型委托、搜索谓词以及对数组或集合中的元素所采取的操作。

用于操作数组和列表的泛型委托

Action<T> 泛型委托表示对指定类型的元素执行某些操作的方法。你可以创建一种对元素执行所需操作的方法，创建 Action<T> 委托的实例来表示该方法，然后将该数组和委托传递给 Array.ForEach 静态泛型方法。数组的每个元素都可以调用该方法。

List<T> 泛型类还提供了 ForEach 方法，该方法使用 Action<T> 委托。此方法不属泛型方法。

> **NOTE**
> 这使泛型类型和方法变得有趣。Array.ForEach 方法必须是静态(在 Visual Basic 中为 `Shared`)和泛型的，因为 Array 不是泛型类型；你可以对 Array.ForEach 指定一种类型以继续运行的唯一原因是该方法有自己的类型参数列表。与之相反，非泛型 List<T>.ForEach 方法属于泛型类 List<T>，因此它仅使用其类的类型参数。该类为强类型，因此此方法可以作为实例方法。

Predicate<T> 泛型委托表示的是用于确定特定元素是否满足你定义的标准的方法。你可以将其用于 Array 的以下静态泛型方法来搜索一个或一组元素：Exists、Find、FindAll、FindIndex、FindLast、FindLastIndex 和TrueForAll。

Predicate<T> 也适用于 List<T> 泛型类相应的非泛型实例方法。

Comparison<T> 泛型委托让你能为不具有本机排序顺序的数组或列表元素提供顺序排序或重写本机排序顺序。创建执行比较的方法，创建一个 Comparison<T> 委托的实例，以表示你的方法，然后将该数组和委托传递给 Array.Sort<T>(T[], Comparison<T>) 静态泛型方法。List<T> 泛型类提供相应的实例方法重载，List<T>.Sort(Comparison<T>)。

Converter<TInput,TOutput>泛型委托让你能定义两个类型之间的转换，将一个类型的数组转换到另一个类型的数组，或者将一个类型的列表转换到另一个类型的列表。创建将现有列表元素转换为新类型的方法，创建一个委托实例来表示该方法，并使用 Array.ConvertAll 泛型静态方法，从原始数组生成新类型数组；或使用 List<T>.ConvertAll<TOutput>(Converter<T,TOutput>) 泛型实例方法，从原始列表生成新类型列表。

链接委托

使用这些委托的许多方法返回数组或列表，然后传递到另一种方法。例如，如果你想要选择某些数组元素，将这些元素转换为新类型，并将其保存在新的数组中，则可以将 FindAll 泛型方法返回的数组传递到 ConvertAll 泛型方法。如果新的元素类型缺少自然排序顺序，你可以将 ConvertAll 泛型方法返回的数组传递到 Sort<T>(T[], Comparison<T>) 泛型方法。

另请参阅

- System.Collections.Generic
- System.Collections.ObjectModel
- 泛型
- .NET 中的泛型集合
- 泛型接口
- 协变和逆变

泛型接口

2021/11/16 •

本文概述跨泛型类型系列提供通用功能的泛型接口。

泛型接口提供与非泛型接口对应的类型安全接口，用于实现排序比较、相等比较以及泛型集合类型所共享的功能。

> **NOTE**
> 多个泛型接口的类型参数标记为协变或逆变，这为分配和使用实现这些接口的类型提供了更好的灵活性。请参阅 协变和逆变。

相等比较和排序比较

在 System 命名空间中，System.IComparable<T> 和 System.IEquatable<T> 泛型接口与它们对应的非泛型接口一样，各自定义了用于排序比较和相等比较的方法。类型通过实现这些接口来提供执行这些比较的能力。

在 System.Collections.Generic 命名空间中，IComparer<T> 和 IEqualityComparer<T> 泛型接口为没有实现 System.IComparable<T> 或 System.IEquatable<T> 泛型接口的类型提供一种定义排序比较和相等比较的方式，并为实现了上述泛型接口的类型提供了重新定义这些关系的方式。这些接口由许多泛型集合类的方法和构造函数使用。例如，可以将泛型 IComparer<T> 对象传递至 SortedDictionary<TKey,TValue> 类的构造函数，以便为没有实现泛型 System.IComparable<T> 的类型指定排列顺序。存在 Array.Sort 泛型静态方法与通过泛型 IComparer<T> 实现对数组和列表进行排序的 List<T>.Sort 实例方法的重载。

Comparer<T> 和 EqualityComparer<T> 泛型类为 IComparer<T> 和 IEqualityComparer<T> 泛型接口的实现提供基类，同时还通过其各自的 Comparer<T>.Default 和 EqualityComparer<T>.Default 属性提供默认排序比较和相等比较。

集合功能

ICollection<T> 泛型接口是泛型集合类型的基本接口。它提供添加、删除、复制和枚举元素的基本功能。ICollection<T> 继承自泛型 IEnumerable<T> 和非泛型 IEnumerable。

IList<T> 泛型接口使用索引检索的方法扩展 ICollection<T> 泛型接口。

IDictionary<TKey,TValue> 泛型接口使用键控检索的方法扩展 ICollection<T> 泛型接口。.NET 基类库中的泛型字典类型还实现非泛型 IDictionary 接口。

IEnumerable<T> 泛型接口提供泛型枚举器结构。泛型枚举器实现的 IEnumerator<T> 泛型接口继承自非泛型 IEnumerator 接口；MoveNext 和 Reset 成员（不依赖于类型参数 `T`）仅出现在非泛型接口中。这意味着非泛型接口的任何使用者还可以使用泛型接口。

请参阅

- System.Collections.Generic
- System.Collections.ObjectModel
- 泛型
- .NET 中的泛型集合
- 用于控制数组和列表的泛型委托
- 协变和逆变

泛型中的协变和逆变

2021/11/16 •

协变和逆变都是术语，前者指能够使用比原始指定的派生类型的派生程度更大（更具体的）的类型，后者指能够使用比原始指定的派生类型的派生程度更小（不太具体的）的类型 。泛型类型参数支持协变和逆变，可在分配和使用泛型类型方面提供更大的灵活性。

在引用类型系统时，协变、逆变和不变性具有如下定义。这些示例假定一个名为 `Base` 的基类和一个名为 `Derived` 的派生类。

- `Covariance`

  使你能够使用比原始指定的类型派生程度更大的类型。

  你可以将 `IEnumerable<Derived>` 的实例分配给 `IEnumerable<Base>` 类型的变量。

- `Contravariance`

  使你能够使用比原始指定的类型更泛型（派生程度更小）的类型。

  你可以将 `Action<Base>` 的实例分配给 `Action<Derived>` 类型的变量。

- `Invariance`

  表示只能使用最初指定的类型。固定泛型类型参数既不是协变，也不是逆变。

  你无法将 `List<Base>` 的实例分配给 `List<Derived>` 类型的变量，反之亦然。

利用协变类型参数，你可以执行非常类似于普通的多态性的分配，如以下代码中所示。

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

```
Dim d As IEnumerable(Of Derived) = New List(Of Derived)
Dim b As IEnumerable(Of Base) = d
```

List<T> 类实现 IEnumerable<T> 接口，因此 `List<Derived>`（在 Visual Basic 中为 `List(Of Derived)`）实现 `IEnumerable<Derived>`。协变类型参数将执行其余的工作。

相反，逆变看起来却不够直观。下面的示例创建类型 `Action<Base>`（在 Visual Basic 中为 `Action(Of Base)`）的委托，然后将此委托分配给类型 `Action<Derived>` 的变量。

```
Action<Base> b = (target) => { Console.WriteLine(target.GetType().Name); };
Action<Derived> d = b;
d(new Derived());
```

```
Dim b As Action(Of Base) = Sub(target As Base)
                               Console.WriteLine(target.GetType().Name)
                           End Sub
Dim d As Action(Of Derived) = b
d(New Derived())
```

此示例看起来是倒退了，但它是可编译和运行的类型安全代码。由于 Lambda 表达式与其自身所分配到的委托

相匹配，因此它会定义一个方法，此方法采用一个 `Base` 类型的参数且没有返回值。可以将结果委托分配给类型类型 `Action<Derived>` 的变量，因为 `T` 委托的类型参数 Action<T> 是逆变类型参数。由于 `T` 指定了一个参数类型，因此该代码是类型安全代码。在调用类型 `Action<Base>` 的委托（就像它是类型 `Action<Derived>` 的委托一样）时，其参数必须属于类型 `Derived` 。始终可以将此实参安全地传递给基础方法，因为该方法的形参属于类型 `Base` 。

通常，协变类型参数可用作委托的返回类型，而逆变类型参数可用作参数类型。对于接口，协变类型参数可用作接口的方法的返回类型，而逆变类型参数可用作接口的方法的参数类型。

协变和逆变统称为“变体”。未标记为协变或逆变的泛型类型参数称为“固定参数”。有关公共语言运行时中变体的事项的简短摘要：

- Variant 类型参数仅限于泛型接口和泛型委托类型。
- 泛型接口或泛型委托类型可以同时具有协变和逆变类型参数。
- 变体仅适用于引用类型；如果为 Variant 类型参数指定值类型，则该类型参数对于生成的构造类型是不变的。
- 变体不适用于委托组合。也就是说，在给定类型 `Action<Derived>` 和 `Action<Base>`（在 Visual Basic 中为 `Action(Of Derived)` 和 `Action(Of Base)` ）的两个委托的情况下，无法将第二个委托与第一个委托结合起来，尽管结果将是类型安全的。变体允许将第二个委托分配给类型 `Action<Derived>` 的变量，但只能在这两个委托的类型完全匹配的情况下对它们进行组合。
- 从 C# 9 开始，支持协变返回类型。重写方法可以声明比它重写的方法派生程度更高的返回类型，而重写的只读属性可以声明派生程度更高的类型。

具有协变类型参数的泛型接口

某些泛型接口具有协变类型参数；例如：IEnumerable<T>、IEnumerator<T>、IQueryable<T> 和 IGrouping<TKey,TElement>。由于这些接口的所有类型参数都是协变类型参数，因此这些类型参数只用于成员的返回类型。

下面的示例阐释了协变类型参数。此示例定义了两个类型：`Base` 具有一个名为 `PrintBases` 的静态方法，该方法采用 `IEnumerable<Base>`（在 Visual Basic 中为 `IEnumerable(Of Base)` ）并输出元素。`Derived` 继承自 `Base` 。此示例创建一个空 `List<Derived>`（在 Visual Basic 中为 `List(Of Derived)` ），并且说明可以将该类型传递给 `PrintBases` 且在不进行强制转换的情况下将该类型分配给类型 `IEnumerable<Base>` 的变量。List<T> 实现 IEnumerable<T>，它具有一个协变类型参数。协变类型参数是可使用 `IEnumerable<Derived>` 的实例而非 `IEnumerable<Base>` 的原因。

```
using System;
using System.Collections.Generic;

class Base
{
    public static void PrintBases(IEnumerable<Base> bases)
    {
        foreach(Base b in bases)
        {
            Console.WriteLine(b);
        }
    }
}

class Derived : Base
{
    public static void Main()
    {
        List<Derived> dlist = new List<Derived>();

        Derived.PrintBases(dlist);
        IEnumerable<Base> bIEnum = dlist;
    }
}
```

```
Imports System.Collections.Generic

Class Base
    Public Shared Sub PrintBases(ByVal bases As IEnumerable(Of Base))
        For Each b As Base In bases
            Console.WriteLine(b)
        Next
    End Sub
End Class

Class Derived
    Inherits Base

    Shared Sub Main()
        Dim dlist As New List(Of Derived)()

        Derived.PrintBases(dlist)
        Dim bIEnum As IEnumerable(Of Base) = dlist
    End Sub
End Class
```

具有逆变类型参数的泛型接口

某些泛型接口具有逆变类型参数；例如：IComparer<T>、IComparable<T> 和 IEqualityComparer<T>。由于这些接口只具有逆变类型参数，因此这些类型参数只用作接口成员中的参数类型。

下面的示例阐释了逆变类型参数。该示例定义具有 `Area` 属性的抽象（在 Visual Basic 中为 `MustInherit`）`Shape` 类。该示例还定义一个实现 `IComparer<Shape>`（在 Visual Basic 中为 `IComparer(Of Shape)`）的 `ShapeAreaComparer` 类。IComparer<T>.Compare 方法的实现基于 `Area` 属性的值，所以 `ShapeAreaComparer` 可用于按区域对 `Shape` 对象排序。

`Circle` 类继承 `Shape` 并替代 `Area`。该示例使用一个采用 `IComparer<Circle>`（在 Visual Basic 中为 `IComparer(Of Circle)`）的构造函数创建 `Circle` 对象的 SortedSet<T>。但是，该示例不传递 `IComparer<Circle>`，而是传递一个用于实现 `IComparer<Shape>` 的 `ShapeAreaComparer` 对象。当代码调用派生程度较高的类型（`Circle`）的比较器时，该示例可以传递派生程度较低的类型（`Shape`）的比较器，因为 IComparer<T> 泛型接口的类型参数是逆变参数。

向 `Circle` 中添加新 `SortedSet<Circle>` 对象时，每次将新元素与现有元素进行比较时，都会调用 `IComparer<Shape>.Compare` 对象的 `IComparer(Of Shape).Compare` 方法(在 Visual Basic 中为 `ShapeAreaComparer` 方法)。该方法的参数类型 (`Shape`) 比所传递的类型 (`Circle`) 的派生程度低，所以调用是类型安全的。逆变使 `ShapeAreaComparer` 可以对派生自 `Shape` 的任意单个类型的集合以及混合类型的集合排序。

```
using System;
using System.Collections.Generic;

abstract class Shape
{
    public virtual double Area { get { return 0; }}
}

class Circle : Shape
{
    private double r;
    public Circle(double radius) { r = radius; }
    public double Radius { get { return r; }}
    public override double Area { get { return Math.PI * r * r; }}
}

class ShapeAreaComparer : System.Collections.Generic.IComparer<Shape>
{
    int IComparer<Shape>.Compare(Shape a, Shape b)
    {
        if (a == null) return b == null ? 0 : -1;
        return b == null ? 1 : a.Area.CompareTo(b.Area);
    }
}

class Program
{
    static void Main()
    {
        // You can pass ShapeAreaComparer, which implements IComparer<Shape>,
        // even though the constructor for SortedSet<Circle> expects
        // IComparer<Circle>, because type parameter T of IComparer<T> is
        // contravariant.
        SortedSet<Circle> circlesByArea =
            new SortedSet<Circle>(new ShapeAreaComparer())
                { new Circle(7.2), new Circle(100), null, new Circle(.01) };

        foreach (Circle c in circlesByArea)
        {
            Console.WriteLine(c == null ? "null" : "Circle with area " + c.Area);
        }
    }
}

/* This code example produces the following output:

null
Circle with area 0.000314159265358979
Circle with area 162.860163162095
Circle with area 31415.9265358979
 */
```

```
Imports System.Collections.Generic

MustInherit Class Shape
    Public MustOverride ReadOnly Property Area As Double
End Class

Class Circle
    Inherits Shape

    Private r As Double
    Public Sub New(ByVal radius As Double)
        r = radius
    End Sub
    Public ReadOnly Property Radius As Double
        Get
            Return r
        End Get
    End Property
    Public Overrides ReadOnly Property Area As Double
        Get
            Return Math.Pi * r * r
        End Get
    End Property
End Class

Class ShapeAreaComparer
    Implements System.Collections.Generic.IComparer(Of Shape)

    Private Function AreaComparer(ByVal a As Shape, ByVal b As Shape) As Integer _
            Implements System.Collections.Generic.IComparer(Of Shape).Compare
        If a Is Nothing Then Return If(b Is Nothing, 0, -1)
        Return If(b Is Nothing, 1, a.Area.CompareTo(b.Area))
    End Function
End Class

Class Program
    Shared Sub Main()
        ' You can pass ShapeAreaComparer, which implements IComparer(Of Shape),
        ' even though the constructor for SortedSet(Of Circle) expects
        ' IComparer(Of Circle), because type parameter T of IComparer(Of T)
        ' is contravariant.
        Dim circlesByArea As New SortedSet(Of Circle)(New ShapeAreaComparer()) _
            From {New Circle(7.2), New Circle(100), Nothing, New Circle(.01)}

        For Each c As Circle In circlesByArea
            Console.WriteLine(If(c Is Nothing, "Nothing", "Circle with area " & c.Area))
        Next
    End Sub
End Class

' This code example produces the following output:
'
'Nothing
'Circle with area 0.000314159265358979
'Circle with area 162.860163162095
'Circle with area 31415.9265358979
```

具有 Variant 类型参数的泛型委托

`Func` 泛型委托(如 Func<T,TResult>)具有协变返回类型和逆变参数类型。`Action` 泛型委托(如 Action<T1,T2>)具有逆变参数类型。这意味着，可以将委托指派给具有派生程度较高的参数类型和(对于 `Func` 泛型委托)派生程度较低的返回类型的变量。

NOTE

`Func` 泛型委托的最后一个泛型类型参数指定委托签名中返回值的类型。该参数是协变的(`out` 关键字)，而其他泛型类型参数是逆变的(`in` 关键字)。

下面的代码阐释这一点。第一段代码定义了一个名为 `Base` 的类、一个名为 `Derived` 的类(此类继承 `Base`)和另一个具有名为 `MyMethod` 的 `static` 方法(在 Visual Basic 中为 `Shared`)的类。该方法接受 `Base` 的实例，并返回 `Derived` 的实例。(如果参数是 `Derived` 的实例，则 `MyMethod` 将返回该实例；如果参数是 `Base` 的实例，则 `MyMethod` 将返回 `Derived` 的新实例。)在 `Main()` 中，该示例创建一个表示 `Func<Base, Derived>` 的 `Func(Of Base, Derived)`(在 Visual Basic 中为 `MyMethod`)的实例，并将此实例存储在变量 `f1` 中。

```
public class Base {}
public class Derived : Base {}

public class Program
{
    public static Derived MyMethod(Base b)
    {
        return b as Derived ?? new Derived();
    }

    static void Main()
    {
        Func<Base, Derived> f1 = MyMethod;
```

```
Public Class Base
End Class
Public Class Derived
    Inherits Base
End Class

Public Class Program
    Public Shared Function MyMethod(ByVal b As Base) As Derived
        Return If(TypeOf b Is Derived, b, New Derived())
    End Function

    Shared Sub Main()
        Dim f1 As Func(Of Base, Derived) = AddressOf MyMethod
```

第二段代码说明可以将委托分配给类型 `Func<Base, Base>`(在 Visual Basic 中为 `Func(Of Base, Base)`)的变量，因为返回类型是协变的。

```
// Covariant return type.
Func<Base, Base> f2 = f1;
Base b2 = f2(new Base());
```

```
' Covariant return type.
Dim f2 As Func(Of Base, Base) = f1
Dim b2 As Base = f2(New Base())
```

第三段代码说明可以将委托分配给类型 `Func<Derived, Derived>`(在 Visual Basic 中为 `Func(Of Derived, Derived)`)的变量，因为参数类型是逆变的。

```
// Contravariant parameter type.
Func<Derived, Derived> f3 = f1;
Derived d3 = f3(new Derived());
```

```
' Contravariant parameter type.
Dim f3 As Func(Of Derived, Derived) = f1
Dim d3 As Derived = f3(New Derived())
```

最后一段代码说明可以将委托分配给类型 `Func<Derived, Base>`（在 Visual Basic 中为 `Func(Of Derived, Base)`）的变量，从而将逆变参数类型和协变返回类型的作用结合起来。

```
// Covariant return type and contravariant parameter type.
Func<Derived, Base> f4 = f1;
Base b4 = f4(new Derived());
```

```
' Covariant return type and contravariant parameter type.
Dim f4 As Func(Of Derived, Base) = f1
Dim b4 As Base = f4(New Derived())
```

非泛型委托中的变体

在上面的代码中，`MyMethod` 的签名与所构造的泛型委托 `Func<Base, Derived>`（在 Visual Basic 中为 `Func(Of Base, Derived)`）的签名完全匹配。此示例说明，只要所有委托类型都是从泛型委托类型 Func<T,TResult> 构造的，就可以将此泛型委托存储在具有派生程度更大的参数类型和派生程度更小的返回类型的变量或方法参数中。

这一点非常重要。泛型委托的类型参数中的协方差和逆变的效果类似于普通委托绑定中的协方差和逆变的效果（请参阅委托中的差异 (C#) 和委托中的差异 (Visual Basic)）。但是，委托绑定中的变化适用于所有委托类型，而不仅仅适用于具有 Variant 类型参数的泛型委托类型。此外，通过委托绑定中的变化，可以将方法绑定到具有限制较多的参数类型和限制较少的返回类型的任何委托，而对于泛型委托的指派，只有在委托类型是基于同一个泛型类型定义构造的时才可以进行。

下面的示例演示委托绑定中的变化和泛型类型参数中的变化的组合效果。该示例定义了一个类型层次结构，其中包含三个按派生程度从低到高排列的类型，即 `Type1` 的派生程度最低，`Type3` 的派生程度最高。普通委托绑定中的变化用于将参数类型为 `Type1` 、返回类型为 `Type3` 的方法绑定到参数类型为 `Type2` 、返回类型为 `Type2` 的泛型委托。然后，使用泛型类型参数的协变和逆变，将得到的泛型委托指派给另一个变量，此变量的泛型委托类型的参数类型为 `Type3` ，返回类型为 `Type1` 。第二个指派要求变量类型和委托类型是基于同一个泛型类型定义（在本例中为 Func<T,TResult>）构造的。

```
using System;

public class Type1 {}
public class Type2 : Type1 {}
public class Type3 : Type2 {}

public class Program
{
    public static Type3 MyMethod(Type1 t)
    {
        return t as Type3 ?? new Type3();
    }

    static void Main()
    {
        Func<Type2, Type2> f1 = MyMethod;

        // Covariant return type and contravariant parameter type.
        Func<Type3, Type1> f2 = f1;
        Type1 t1 = f2(new Type3());
    }
}
```

```
Public Class Type1
End Class
Public Class Type2
    Inherits Type1
End Class
Public Class Type3
    Inherits Type2
End Class

Public Class Program
    Public Shared Function MyMethod(ByVal t As Type1) As Type3
        Return If(TypeOf t Is Type3, t, New Type3())
    End Function

    Shared Sub Main()
        Dim f1 As Func(Of Type2, Type2) = AddressOf MyMethod

        ' Covariant return type and contravariant parameter type.
        Dim f2 As Func(Of Type3, Type1) = f1
        Dim t1 As Type1 = f2(New Type3())
    End Sub
End Class
```

定义变体泛型接口和委托

Visual Basic 和 C# 提供了一些关键字，利用这些关键字，可以将接口和委托的泛型类型参数标记为协变或逆变。

协变类型参数用 `out` 关键字(在 Visual Basic 中为 `Out` 关键字)标记。可以将协变类型参数用作属于接口的方法的返回值，或用作委托的返回类型。但不能将协变类型参数用作接口方法的泛型类型约束。

> **NOTE**
> 如果接口的方法具有泛型委托类型的参数，则接口类型的协变类型参数可用于指定委托类型的逆变类型参数。

逆变类型参数用 `in` 关键字(在 Visual Basic 中为 `In` 关键字)标记。可以将逆变类型参数用作属于接口的方法的参数类型，或用作委托的参数类型。也可以将逆变类型参数用作接口方法的泛型类型约束。

只有接口类型和委托类型才能具有 Variant 类型参数。接口或委托类型可以同时具有协变和逆变类型参数。

Visual Basic 和 C# 不允许违反协变和逆变类型参数的使用规则，也不允许将协变和逆变批注添加到接口和委托类型之外的类型参数中。

有关信息和示例代码，请参阅泛型接口中的差异 (C#) 和泛型接口中的差异 (Visual Basic)。

类型列表

下面的接口和委托类型具有协变和/或逆变类型参数。

[illegible]	[illegible]	[illegible]
Action<T> 至 Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>		是
Comparison<T>		是
Converter<TInput,TOutput>	是	是
Func<TResult>	是	
Func<T,TResult> 至 Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>	是	是
IComparable<T>		是
Predicate<T>		是
IComparer<T>		是
IEnumerable<T>	是	
IEnumerator<T>	是	
IEqualityComparer<T>		是
IGrouping<TKey,TElement>	是	
IOrderedEnumerable<TElement>	是	
IOrderedQueryable<T>	是	
IQueryable<T>	是	

请参阅

- 协变和逆变 (C#)
- 协变和逆变 (Visual Basic)
- 委托中的变体 (C#)
- 委托中的变体 (Visual Basic)

集合和数据结构

2021/11/16 •

类似的数据在作为集合而存储和操作时通常可以得到更高效地处理。可以使用 System.Array 类或 System.Collections、System.Collections.Generic、System.Collections.Concurrent 和 System.Collections.Immutable 命名空间中的类来添加、删除和修改集合中的任一单个元素或某个范围的元素。

有两种主要的集合类型：泛型集合和非泛型集合。泛型集合在编译时是类型安全的。因此，泛型集合通常能提供更好的性能。构造泛型集合时，它们接受类型形参；并在向该集合添加项或从该集合删除项时无需在 Object 类型间来回转换。此外，Windows 应用商店应用支持大多数泛型集合。非泛型集合将项存储为 Object，需要强制转换，并且大多数不支持 Windows 应用商店应用开发。但是，你可能会看到在较旧的代码中有非泛型集合。

自 .NET Framework 4 起，System.Collections.Concurrent 命名空间中的集合可提供高效的线程安全操作，以便从多个线程访问集合项。System.Collections.Immutable 命名空间（NuGet 包）中的不可变集合类本质上就是线程安全的，因为操作在原始集合的副本上进行且不能修改原始集合。

常用集合功能

所有集合都提供用于在集合中添加、删除或查找项的方法。此外，所有直接或间接实现 ICollection 接口或 ICollection<T> 接口的集合均共享这些功能：

- **可枚举集合**

  .NET 集合实现 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T>，以启用要循环访问的集合。可将枚举器看作集合中可指向任何元素的可移动指针。foreach, in 语句和 For Each...Next 语句使用 GetEnumerator 方法公开的枚举器并隐藏操作枚举器的复杂性。此外，任何实现 System.Collections.Generic.IEnumerable<T> 的集合均被认为是可查询类型，并可使用 LINQ 对其进行查询。LINQ 查询提供数据访问的一个通用模式。它们通常比标准 `foreach` 循环更简洁、更具可读性，并提供筛选、排序和分组功能。LINQ 查询还可提高性能。有关详细信息，请参阅 LINQ to Objects (C#)、LINQ to Objects (Visual Basic)、并行 LINQ (PLINQ)、LINQ 查询 (C#) 简介和基本查询操作 (Visual Basic)。

- **可将集合内容复制到数组**

  可使用 CopyTo 方法将所有集合复制到数组中；但新数组中的元素顺序是以枚举器返回元素的顺序为依据。得到的数组始终是一维的，下限为零。

此外，许多集合类包含下列功能：

- **容量和计数属性**

  集合的容量是它可包含的元素数。集合的计数是它实际所含的元素数。某些集合隐藏容量、计数或将这两者都隐藏。

  达到当前容量时，大多数集合会自动扩展容量。重新分配内存并将元素从旧集合复制到新集合。这减少了要求使用集合的代码；但集合的性能可能会受到不利影响。例如，对 List<T> 来说，如果 Count 比 Capacity 少，那么添加项就是一项 O(1) 操作。如需增加容量以容纳新元素，则添加项成为 O(`n`) 操作，其中 `n` 是 Count。避免因多次重新分配而导致的性能较差的最佳方式是：将初始容量设置为集合的估计大小。

  BitArray 是一种特殊情况；它的容量与其长度相同，而其长度与其计数相同。

- **下限一致**

  集合的下限是其第一个元素的索引。System.Collections 命名空间中的所有索引集合的下限均为零，这表

示它们从 0 开始建立索引。Array 默认下限为零，但使用 Array.CreateInstance 创建 Array 类的实例时可定义其他下限。

- **同步以从多个线程进行访问**（仅 System.Collections 类）。

  System.Collections 命名空间中的非泛型集合类型通过同步提供一些线程安全性；通常通过 SyncRoot 和 IsSynchronized 成员公开。这些集合不是默认为线程安全的。如需对集合进行可扩展、高效的多线程访问，请使用 System.Collections.Concurrent 命名空间中的一个类或考虑使用不可变集合。有关详细信息，请参阅线程安全集合。

选择集合

一般情况下，应使用泛型集合。下表介绍了一些常用的集合方案和可用于这些方案的集合类。如果你是使用泛型集合的新手，此表将帮助你选择最适合你的任务的泛型集合。

[illegible]	[illegible]	[illegible]	[illegible]
将项存储为键/值对以通过键进行快速查找	Dictionary<TKey,TValue>	Hashtable （根据键的哈希代码组织的键/值对的集合。）	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
按索引访问项	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
使用项先进先出 (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
使用数据后进先出 (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
按顺序访问项	LinkedList<T>	无建议	无建议
删除集合中的项或向集合添加项时接收通知。（实现 INotifyPropertyChanged 和 INotifyCollectionChanged）	ObservableCollection<T>	无建议	无建议
已排序的集合	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue> ImmutableSortedSet<T>
数学函数的一个集	HashSet<T> SortedSet<T>	无建议	ImmutableHashSet<T> ImmutableSortedSet<T>

集合的算法复杂性

选择集合类时，应该考虑可能牺牲的性能。使用下表来参考各种可变集合类型在算法复杂性方面与其对应不可变对应项的比较情况。不可变的集合类型通常性能较低，但却提供了不可变性，这通常是一种非常有效的优点。

[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
`Stack<T>.Push`	O(1)	O(`n`)	`ImmutableStack<T>.Push`	O(1)
`Queue<T>.Enqueue`	O(1)	O(`n`)	`ImmutableQueue<T>.Enqueue`	O(1)
`List<T>.Add`	O(1)	O(`n`)	`ImmutableList<T>.Add`	O(log `n`)
`List<T>.Item[Int32]`	O(1)	O(1)	`ImmutableList<T>.Item[Int32]`	O(log `n`)
`List<T>.Enumerator`	O(`n`)	O(`n`)	`ImmutableList<T>.Enumerator`	O(`n`)
`HashSet<T>.Add` , lookup	O(1)	O(`n`)	`ImmutableHashSet<T>.Add`	O(log `n`)
`SortedSet<T>.Add`	O(log `n`)	O(`n`)	`ImmutableSortedSet<T>.Add`	O(log `n`)
`Dictionary<T>.Add`	O(1)	O(`n`)	`ImmutableDictionary<T>.Add`	O(log `n`)
`Dictionary<T>` lookup	O(1)	O(1) - 或者从严格意义上说，O(`n`)	`ImmutableDictionary<T>` lookup	O(log `n`)
`SortedDictionary<T>.Add`	O(log `n`)	O(`n` log `n`)	`ImmutableSortedDictionary<T>.Add`	O(log `n`)

可以使用 `for` 循环或 `foreach` 循环来有效地枚举 `List<T>` 。但是，由于其索引器的 O(log `n`) 时间， `ImmutableList<T>` 在 `for` 循环内的效果较差。使用 `foreach` 循环枚举 `ImmutableList<T>` 很有效，因为 `ImmutableList<T>` 使用二进制树来存储其数据，而不是像 `List<T>` 那样使用简单数组。数组可以非常快速地编入索引，但必须向下遍历二进制树，直到找到具有所需索引的节点。

此外， `SortedSet<T>` 与 `ImmutableSortedSet<T>` 的复杂性相同。这是因为它们都使用了二进制树。当然，显著的差异在于 `ImmutableSortedSet<T>` 使用不可变的二进制树。由于 `ImmutableSortedSet<T>` 还提供了一个允许变化的 System.Collections.Immutable.ImmutableSortedSet<T>.Builder 类，因此可以同时实现不可变性和保障性能。

相关主题

TITLE	[illegible]
选择集合类	描述不同的集合并帮助你为你的方案选择一个集合。
常用的集合类型	描述诸如 System.Array、System.Collections.Generic.List<T> 和 System.Collections.Generic.Dictionary<TKey,TValue> 等常用泛型和非泛型集合类型。
何时使用泛型集合	讨论泛型集合类型的使用。
集合内的比较和排序	讨论在集合中使用等同性比较和排序比较。
已排序的集合类型	描述已排序集合的性能和特征
哈希表和字典集合类型	描述泛型和非泛型基于哈希的字典类型的功能。

TITLE	说明
线程安全集合	介绍支持从多个线程进行安全有效的并发访问的集合类型，例如 System.Collections.Concurrent.BlockingCollection<T> 和 System.Collections.Concurrent.ConcurrentBag<T>。
System.Collections.Immutable	介绍不可变集合并提供各集合类型的链接。

参考

System.Array System.Collections System.Collections.Concurrent System.Collections.Generic System.Collections.Specialized System.Linq System.Collections.Immutable

选择集合类

2021/11/16 •

请务必仔细选择你的集合类。使用错误的类型可能会限制集合的使用。

> **IMPORTANT**
> 请避免使用 System.Collections 命名空间中的类型。推荐使用泛型版本和并发版本的集合，因为它们的类型安全性很高，并且还经过了其他改进。

请考虑下列问题：

- 是否需要顺序列表（其中通常在检索元素值后就将该元素丢弃）？
  - 在需要的情况下，如果需要先进先出 (FIFO) 行为，请考虑使用 Queue 类或 Queue<T> 泛型类。如果需要后进先出 (LIFO) 行为，请考虑使用 Stack 类或 Stack<T> 泛型类。若要从多个线程进行安全访问，请使用并发版本（ConcurrentQueue<T> 和 ConcurrentStack<T>）。如果要获得不可变性，请考虑不可变版本 ImmutableQueue<T> 和 ImmutableStack<T>。
  - 如果不需要，请考虑使用其他集合。
- 是否需要以特定顺序（如先进先出、后进先出或随机）访问元素？
  - Queue 类以及 Queue<T>、ConcurrentQueue<T> 和 ImmutableQueue<T> 泛型类都提供 FIFO 访问权限。有关详细信息，请参阅何时使用线程安全集合。
  - Stack 类以及 Stack<T>、ConcurrentStack<T> 和 ImmutableStack<T> 泛型类都提供 LIFO 访问权限。有关详细信息，请参阅何时使用线程安全集合。
  - LinkedList<T> 泛型类允许从开头到末尾或从末尾到开头的顺序访问。
- 是否需要按索引访问每个元素？
  - ArrayList 和 StringCollection 类以及 List<T> 泛型类按从零开始的元素索引提供对其元素的访问。如果要获得不可变性，请考虑不可变泛型版本 ImmutableArray<T> 和 ImmutableList<T>。
  - Hashtable、SortedList、ListDictionary 和 StringDictionary 类以及 Dictionary<TKey,TValue> 和 SortedDictionary<TKey,TValue> 泛型类按元素的键提供对其元素的访问。此外，还有几个相应类型的不可变版本：ImmutableHashSet<T>、ImmutableDictionary<TKey,TValue>、ImmutableSortedSet<T> 和 ImmutableSortedDictionary<TKey,TValue>。
  - NameObjectCollectionBase 和 NameValueCollection 类以及 KeyedCollection<TKey,TItem> 和 SortedList<TKey,TValue> 泛型类按从零开始的元素索引或元素的键提供对其元素的访问。
- 是否每个元素都包含一个值、一个键和一个值的组合或一个键和多个值的组合？
  - 一个值：使用任何基于 IList 接口或 IList<T> 泛型接口的集合。要获得不可变选项，请考虑 IImmutableList<T> 泛型接口。
  - 一个键和一个值：使用任何基于 IDictionary 接口或 IDictionary<TKey,TValue> 泛型接口的集合。要获得不可变选项，请考虑 IImmutableSet<T> 或 IImmutableDictionary<TKey,TValue> 泛型接口。
  - 带有嵌入键的值：使用 KeyedCollection<TKey,TItem> 泛型类。

    - 一个键和多个值：使用 NameValueCollection 类。

- 是否需要以与输入方式不同的方式对元素进行排序？

    - Hashtable 类按其哈希代码对其元素进行排序。

    - SortedList 类以及 SortedList<TKey,TValue> 和 SortedDictionary<TKey,TValue> 泛型类按键对元素进行排序。排序顺序的依据为，实现 SortedList 类的 IComparer 接口和实现 SortedList<TKey,TValue> 和 SortedDictionary<TKey,TValue> 泛型类的 IComparer<T> 泛型接口。在这两种泛型类型中，虽然 SortedDictionary<TKey,TValue> 的性能优于 SortedList<TKey,TValue>，但 SortedList<TKey,TValue> 占用的内存更少。

    - ArrayList 提供了一种 Sort 方法，此方法采用 IComparer 实现作为参数。其泛型对应项（List<T> 泛型类）提供一种 Sort 方法，此方法采用 IComparer<T> 泛型接口的实现作为参数。

- 是否需要快速搜索和信息检索？

    - 对于小集合（10 项或更少），ListDictionary 速度比 Hashtable 快。Dictionary<TKey,TValue> 泛型类提供比 SortedDictionary<TKey,TValue> 泛型类更快的查找。多线程的实现为 ConcurrentDictionary<TKey,TValue>。ConcurrentBag<T> 为无序数据提供快速的多线程插入。有关这两种多线程类型的详细信息，请参阅何时使用线程安全集合。

- 是否需要只接受字符串的集合？

    - StringCollection（基于 IList）和 StringDictionary（基于 IDictionary）位于 System.Collections.Specialized 命名空间。

    - 此外，通过指定其泛型类参数的 String 类，可以使用 System.Collections.Generic 命名空间中的任何泛型集合类作为强类型字符串集合。例如，可以将变量声明为采用 List<String> 或 Dictionary<String,String> 类型。

LINQ to Objects 与 PLINQ

LINQ to Objects 让开发人员能够使用 LINQ 查询访问内存中对象，条件是该对象类型实现 IEnumerable 或 IEnumerable<T>。LINQ 查询提供了一种通用的数据访问模式，与标准 `foreach` 循环相比，它通常更加简洁，可读性更高，并且可提供筛选、排序和分组功能。有关详细信息，请参阅 LINQ to Objects (C#) 和 LINQ to Objects (Visual Basic)。

PLINQ 提供 LINQ to Objects 的并行实现，在许多情况下，可通过更有效地利用多核计算机提供更快的查询执行。有关详细信息，请参阅并行 LINQ (PLINQ)。

请参阅

- System.Collections
- System.Collections.Specialized
- System.Collections.Generic
- 线程安全集合

常用的集合类型

2021/11/16 •

集合类型是数据集合(如哈希表、队列、堆栈、包、字典和列表)的常见变体。

集合基于 ICollection 接口、IList 接口、IDictionary 接口或它们对应的泛型集合。IList 接口和 IDictionary 接口都派生自 ICollection 接口:因此,所有集合都直接或间接基于 ICollection 接口。在基于 IList 接口(如 Array、ArrayList 或 List<T>)或直接基于 ICollection 接口(如 Queue、ConcurrentQueue<T>、Stack、ConcurrentStack<T> 或 LinkedList<T>)的集合里,每个元素都只有一个值。在基于 IDictionary 接口(比如 Hashtable 和 SortedList 类,Dictionary<TKey,TValue> 和SortedList<TKey,TValue> 泛型类)或 ConcurrentDictionary<TKey,TValue> 类的集合中,每个元素都有一个键和一个值。KeyedCollection<TKey,TItem> 类是唯一的,因为它是值中嵌键的值的列表,因此,它的行为类似列表和字典。

泛型集合都是强类型的最佳解决方案。但,如果你的语言不支持泛型,那么 System.Collections 命名空间包含基集合,如 CollectionBase、ReadOnlyCollectionBase 和 DictionaryBase,这些集合都是可扩展以创建强类型集合类的抽象基类。需要高效的多线程集合访问时,请使用 System.Collections.Concurrent 命名空间中的泛型集合。

集合会因元素的存储方式、排序方式、执行搜索的方式以及比较方式的不同而不同。Queue 类和 Queue<T> 泛型类提供先进先出列表,而 Stack 类和 Stack<T> 泛型类提供后进先出列表。SortedList 类和 SortedList<TKey,TValue> 泛型类提供 Hashtable 类和 Dictionary<TKey,TValue> 泛型类的已排序版本。Hashtable 或 Dictionary<TKey,TValue> 的元素只能通过元素的键访问,但 SortedList 或 KeyedCollection<TKey,TItem> 的元素能通过元素的键或索引访问。所有集合中的索引都从零开始,Array 除外,它允许不从零开始的数组。

LINQ to Objects 功能让你可以通过使用 LINQ 查询来访问内存中的对象,条件是该对象类型实现 IEnumerable 或 IEnumerable<T>。LINQ 查询提供了一种通用的数据访问模式;与标准 `foreach` 循环相比,它通常更加简洁;可读性更高,并且可提供筛选、排序和分组功能。LINQ 查询还可提高性能。有关详细信息,请参阅 "LINQ to Objects (C#)"、"LINQ to Objects (Visual Basic)" 和 "并行 LINQ (PLINQ)"。

相关主题

TITLE	ıı
集合和数据结构	讨论 .NET 中的各种集合类型,包括堆栈、队列、列表、数组和字典。
哈希表和字典集合类型	描述泛型和非泛型的基于哈希的字典类型的功能。
已排序的集合类型	描述为列表和集提供排序功能的类。
泛型	描述泛型功能,包括 .NET 提供的泛型集合、委托和接口。为 C#、Visual Basic 和 Visual C++ 提供功能文档链接和支持技术(如反射)链接。

参考

System.Collections

System.Collections.Generic

System.Collections.ICollection

System.Collections.Generic.ICollection<T>

System.Collections.IList

System.Collections.Generic.IList<T>

System.Collections.IDictionary

System.Collections.Generic.IDictionary<TKey,TValue>

何时使用泛型集合

2021/11/16 •

使用泛型集合可获得类型安全的自动化优点而无需从基集合类型派生和实现特定类型的成员。当集合元素为值类型时，泛型集合类型也通常优于对应的非泛型集合类型（比从非泛型基集合类型派生的类型好），因为使用泛型时不必对元素进行装箱。

对于面向 .NET Standard 1.0 或更高版本的程序，请在多个线程可能会同时向集合添加项或从集合中删除项时使用 System.Collections.Concurrent 命名空间中的泛型集合。此外，当需要不可变性时，请考虑 System.Collections.Immutable 命名空间中的泛型集合类。

以下泛型类型对应于现有集合类型：

- List<T> 泛型类对应于 ArrayList。
- Dictionary<TKey,TValue> 和 ConcurrentDictionary<TKey,TValue> 泛型类对应 Hashtable。
- Collection<T> 泛型类对应于 CollectionBase。Collection<T> 可以用作基类，但是与 CollectionBase 不同，它不抽象，这大大降低了其使用难度。
- ReadOnlyCollection<T> 泛型类对应于 ReadOnlyCollectionBase。ReadOnlyCollection<T> 不是抽象的并且拥有可以轻松地公开现有的 List<T> 为只读集合的构造函数。
- Queue<T>、ConcurrentQueue<T>、ImmutableQueue<T>、ImmutableArray<T>、SortedList<TKey,TValue> 和 ImmutableSortedSet<T> 泛型类对应有着相应的相同名称的非泛型类。

其他类型

几种泛型集合类型没有对应的非泛型集合类型。它们包括以下类型：

- LinkedList<T> 是一个通用的链接列表，该列表提供 O(1) 插入和删除操作。
- SortedDictionary<TKey,TValue> 是一个有 O(log `n`) 插入和检索操作的已排序字典，这使它有效代替了 SortedList<TKey,TValue>。
- KeyedCollection<TKey,TItem> 是列表和字典的结合，它提供了一种方法来存储包含自己的键的对象。
- BlockingCollection<T> 通过限制和阻止功能实现集合类。
- ConcurrentBag<T> 能快速插入和移除未排序元素。

不可变生成器

如果需要在应用中使用不可变性功能，System.Collections.Immutable 命名空间提供了可使用的泛型集合类型。所有不可变的集合类型都提供 `Builder` 类，此类在执行多个突变时可以优化性能。`Builder` 类在不可变状态下批处理操作。所有突变都完成后，调用 `ToImmutable` 方法来“冻结”所有节点并创建一个不可变的泛型集合，例如 ImmutableList<T>。

可以通过调用非泛型 `CreateBuilder()` 方法来创建 `Builder` 对象。通过 `Builder` 实例，可以调用 `ToImmutable()`。同样，通过 `Immutable*` 集合中，可以调用 `ToBuilder()` 从泛型不可变集合创建生成器实例。以下是各种 `Builder` 类型。

- ImmutableArray<T>.Builder
- ImmutableDictionary<TKey,TValue>.Builder

- ImmutableHashSet<T>.Builder
- ImmutableList<T>.Builder
- ImmutableSortedDictionary<TKey,TValue>.Builder
- ImmutableSortedSet<T>.Builder

LINQ to Objects

你可以通过 LINQ to Objects 功能使用 LINQ 查询来访问内存中的对象，但条件是该对象类型要实现 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T> 接口。LINQ 查询提供了一种通用的数据访问模式；与标准 `foreach` 循环相比，它通常更加简洁；可读性更高，并且可提供筛选、排序和分组功能。LINQ 查询还可提高性能。有关详细信息，请参阅 "LINQ to Objects (C#)"、"LINQ to Objects (Visual Basic)" 和 "并行 LINQ (PLINQ)"。

其他功能

一些泛型类型具有非泛型集合类型中找不到的功能。比如与非泛型 List<T> 类相对的 ArrayList 类有大量接受泛型委托的方法，例如允许你指定搜索列表的方法的 Predicate<T> 委托、代表对列表中每个元素发挥作用的 Action<T> 委托和在类型间定义对话的 Converter<TInput,TOutput> 委托。

List<T> 类使你可以指定你自己的用于排序和搜索列表的 IComparer<T> 泛型接口实现。SortedDictionary<TKey,TValue> 和 SortedList<TKey,TValue> 类也有这个功能。另外，这些类使你可以在创建集合时指定比较器。同样地，Dictionary<TKey,TValue> 和 KeyedCollection<TKey,TItem> 类让你指定自己的相等比较器。

请参阅

- 集合和数据结构
- 常用的集合类型
- 泛型

集合内的比较和排序

2021/11/16 •

System.Collections 类在管理集合所涉及的几乎所有进程中执行比较，无论是搜索待删除的元素或返回键值对的值。

集合通常使用相等比较器和/或排序比较器。有两个构造用于比较。

检查是否相等

诸如 `Contains`、IndexOf、LastIndexOf和 `Remove` 的方法将相等比较器用于集合元素。如果集合是泛型的，则按照以下原则比较项是否相等：

- 如果类型 T 实现 IEquatable<T> 泛型接口，则相等比较器是该接口的 Equals 方法。
- 如果类型 T 未实现 IEquatable<T>，则使用 Object.Equals 。

此外，字典集合的某些构造函数重载接受 IEqualityComparer<T> 实现，用于比较键是否相等。有关示例，请参见 Dictionary<TKey,TValue> 构造函数。

确定排序顺序

`BinarySearch` 和 `Sort` 等方法将排序比较器用于集合元素。可在集合的元素间进行比较，或在元素或指定值之间进行比较。对于比较对象，有 `default comparer` 和 `explicit comparer` 的概念。

默认比较器依赖至少一个正在被比较的对象来实现 **IComparable** 接口。在用作列表集合的值或字典集合的键的所有类上实现 **IComparable** 不失为一个好办法。对泛型集合而言，等同性比较是根据以下内容确定的：

- 如果类型 T 实现 System.IComparable<T> 泛型接口，则默认比较器是该接口的 IComparable<T>.CompareTo(T) 方法。
- 如果类型 T 实现非泛型 System.IComparable 接口，则默认比较器是该接口的 IComparable.CompareTo(Object) 方法。
- 如果类型 T 未实现任何接口，则没有默认比较器，必须显式提供一个比较器或比较委托。

为了提供显式比较，某些方法接受 **IComparer** 实现作为参数。例如，List<T>.Sort 方法接受 System.Collections.Generic.IComparer<T> 实现。

系统当前的区域性设置可影响集合中的比较和排序。默认情况下，**Collections** 类中的比较和排序是区分区域性的。若要忽略区域性设置并因此获得一致的比较和排序结果，请使用具有接受 InvariantCulture 的成员重载的 CultureInfo。有关详细信息，请参阅 “在集合中执行不区分区域性的字符串操作” 和 “在数组中执行不区分区域性的字符串操作”。

等同性和排序示例

以下代码展示了 IEquatable<T> 和 IComparable<T> 在简单的业务对象上的实现。此外，如果对象被存储在列表中并已排序，那么你会发现调用 Sort() 方法会导致 `Part` 类型使用默认比较器，并通过使用匿名方法实现 Sort(Comparison<T>) 方法。

```
using System;
using System.Collections.Generic;
```

```
// Simple business object. A PartId is used to identify the
// type of part but the part name can change.
public class Part : IEquatable<Part>, IComparable<Part>
{
    public string PartName { get; set; }

    public int PartId { get; set; }

    public override string ToString() =>
        $"ID: {PartId}   Name: {PartName}";

    public override bool Equals(object obj) =>
        (obj is Part part)
                ? Equals(part)
                : false;

    public int SortByNameAscending(string name1, string name2) =>
        name1?.CompareTo(name2) ?? 1;

    // Default comparer for Part type.
    // A null value means that this object is greater.
    public int CompareTo(Part comparePart) =>
        comparePart == null ? 1 : PartId.CompareTo(comparePart.PartId);

    public override int GetHashCode() => PartId;

    public bool Equals(Part other) =>
        other is null ? false : PartId.Equals(other.PartId);

    // Should also override == and != operators.
}

public class Example
{
    public static void Main()
    {
        // Create a list of parts.
        var parts = new List<Part>
        {
            // Add parts to the list.
            new Part { PartName = "regular seat", PartId = 1434 },
            new Part { PartName = "crank arm", PartId = 1234 },
            new Part { PartName = "shift lever", PartId = 1634 },
            // Name intentionally left null.
            new Part { PartId = 1334 },
            new Part { PartName = "banana seat", PartId = 1444 },
            new Part { PartName = "cassette", PartId = 1534 }
        };

        // Write out the parts in the list. This will call the overridden
        // ToString method in the Part class.
        Console.WriteLine("\nBefore sort:");
        parts.ForEach(Console.WriteLine);

        // Call Sort on the list. This will use the
        // default comparer, which is the Compare method
        // implemented on Part.
        parts.Sort();

        Console.WriteLine("\nAfter sort by part number:");
        parts.ForEach(Console.WriteLine);

        // This shows calling the Sort(Comparison<T> comparison) overload using
        // a lambda expression as the Comparison<T> delegate.
        // This method treats null as the lesser of two values.
        parts.Sort((Part x, Part y) =>
            x.PartName == null && y.PartName == null
                ? 0
```

```
                : x.PartName == null
                    ? -1
                    : y.PartName == null
                        ? 1
                        : x.PartName.CompareTo(y.PartName));

        Console.WriteLine("\nAfter sort by name:");
        parts.ForEach(Console.WriteLine);

        /*

            Before sort:
        ID: 1434   Name: regular seat
        ID: 1234   Name: crank arm
        ID: 1634   Name: shift lever
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette

        After sort by part number:
        ID: 1234   Name: crank arm
        ID: 1334   Name:
        ID: 1434   Name: regular seat
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1634   Name: shift lever

        After sort by name:
        ID: 1334   Name:
        ID: 1444   Name: banana seat
        ID: 1534   Name: cassette
        ID: 1234   Name: crank arm
        ID: 1434   Name: regular seat
        ID: 1634   Name: shift lever

         */
    }
}
```

```
Imports System.Collections.Generic

' Simple business object. A PartId is used to identify the type of part
' but the part name can change.
Public Class Part
    Implements IEquatable(Of Part)
    Implements IComparable(Of Part)
    Public Property PartName() As String
        Get
            Return m_PartName
        End Get
        Set(value As String)
            m_PartName = Value
        End Set
    End Property
    Private m_PartName As String

    Public Property PartId() As Integer
        Get
            Return m_PartId
        End Get
        Set(value As Integer)
            m_PartId = Value
        End Set
    End Property
    Private m_PartId As Integer

    Public Overrides Function ToString() As String
```

```
        Return "ID: " & PartId & "   Name: " & PartName
    End Function

    Public Overrides Function Equals(obj As Object) As Boolean
        If obj Is Nothing Then
            Return False
        End If
        Dim objAsPart As Part = TryCast(obj, Part)
        If objAsPart Is Nothing Then
            Return False
        Else
            Return Equals(objAsPart)
        End If
    End Function

    Public Function SortByNameAscending(name1 As String, name2 As String) As Integer

        Return name1.CompareTo(name2)
    End Function

    ' Default comparer for Part.
    Public Function CompareTo(comparePart As Part) As Integer _
            Implements IComparable(Of ListSortVB.Part).CompareTo
        ' A null value means that this object is greater.
        If comparePart Is Nothing Then
            Return 1
        Else

            Return Me.PartId.CompareTo(comparePart.PartId)
        End If
    End Function
    Public Overrides Function GetHashCode() As Integer
        Return PartId
    End Function
    Public Overloads Function Equals(other As Part) As Boolean Implements IEquatable(Of
ListSortVB.Part).Equals
        If other Is Nothing Then
            Return False
        End If
        Return (Me.PartId.Equals(other.PartId))
    End Function
    ' Should also override == and != operators.

End Class
Public Class Example
    Public Shared Sub Main()
        ' Create a list of parts.
        Dim parts As New List(Of Part)()

        ' Add parts to the list.
        parts.Add(New Part() With { _
             .PartName = "regular seat", _
             .PartId = 1434 _
        })
        parts.Add(New Part() With { _
             .PartName = "crank arm", _
             .PartId = 1234 _
        })
        parts.Add(New Part() With { _
             .PartName = "shift lever", _
             .PartId = 1634 _
        })


        ' Name intentionally left null.
        parts.Add(New Part() With { _
             .PartId = 1334 _
        })
        parts.Add(New Part() With {
```

```
parts.Add(New Part() With { _
     .PartName = "banana seat", _
     .PartId = 1444 _
})
parts.Add(New Part() With { _
     .PartName = "cassette", _
     .PartId = 1534 _
})


' Write out the parts in the list. This will call the overridden
' ToString method in the Part class.
Console.WriteLine(vbLf & "Before sort:")
For Each aPart As Part In parts
    Console.WriteLine(aPart)
Next


' Call Sort on the list. This will use the
' default comparer, which is the Compare method
' implemented on Part.
parts.Sort()


Console.WriteLine(vbLf & "After sort by part number:")
For Each aPart As Part In parts
    Console.WriteLine(aPart)
Next

' This shows calling the Sort(Comparison(T) overload using
' an anonymous delegate method.
' This method treats null as the lesser of two values.
parts.Sort(Function(x As Part, y As Part)
               If x.PartName Is Nothing AndAlso y.PartName Is Nothing Then
                   Return 0
               ElseIf x.PartName Is Nothing Then
                   Return -1
               ElseIf y.PartName Is Nothing Then
                   Return 1
               Else
                   Return x.PartName.CompareTo(y.PartName)
               End If
           End Function)


Console.WriteLine(vbLf & "After sort by name:")
For Each aPart As Part In parts
    Console.WriteLine(aPart)
Next

'
'
'            Before sort:
'            ID: 1434   Name: regular seat
'            ID: 1234   Name: crank arm
'            ID: 1634   Name: shift lever
'            ID: 1334   Name:
'            ID: 1444   Name: banana seat
'            ID: 1534   Name: cassette
'
'            After sort by part number:
'            ID: 1234   Name: crank arm
'            ID: 1334   Name:
'            ID: 1434   Name: regular seat
'            ID: 1444   Name: banana seat
'            ID: 1534   Name: cassette
'            ID: 1634   Name: shift lever
'
'            After sort by name:
'            ID: 1334   Name:
```

```
                    ID: 1534   Name:
        '           ID: 1444   Name: banana seat
        '           ID: 1534   Name: cassette
        '           ID: 1234   Name: crank arm
        '           ID: 1434   Name: regular seat
        '           ID: 1634   Name: shift lever

    End Sub
End Class
```

请参阅

- IComparer
- IEquatable<T>
- IComparer<T>
- IComparable
- IComparable<T>

已排序的集合类型

2021/11/16 •

System.Collections.SortedList 类、System.Collections.Generic.SortedList<TKey,TValue> 泛型类和 System.Collections.Generic.SortedDictionary<TKey,TValue> 泛型类的相似之处在于均实现 IDictionary 接口，不同之处在于它们让元素一直按键的排序顺序排列，并且不具备哈希表的 O(1) 插入和检索特性。这三个类具有若干共性：

- 这三个类全都实现 System.Collections.IDictionary 接口。两个泛型类还实现 System.Collections.Generic.IDictionary<TKey,TValue> 泛型接口。
- 每个元素都是用于枚举的键/值对。

> **NOTE**
> 枚举时，非泛型 SortedList 类返回 DictionaryEntry 对象，尽管两个泛型类型返回 KeyValuePair<TKey,TValue> 对象。

- 元素按 System.Collections.IComparer 实现代码（对于非泛型 SortedList）或 System.Collections.Generic.IComparer<T> 实现代码（对于两个泛型类）进行排序。
- 每个类提供了返回仅包含键或仅包含值的集合的属性。

下表列出了两个已排序列表类与 SortedDictionary<TKey,TValue> 类之间的一些区别。

SortedList 和 SortedList<TKey,TValue>	SortedDictionary<TKey,TValue>
返回键和值的属性是有索引的，从而允许高效的索引检索。	无索引的检索。
检索属于 O(log `n`) 操作。	检索属于 O(log `n`) 操作。
插入和删除通常属于 O(`n`) 操作；不过，对于已按排序顺序排列的数据，插入属于 O(log `n`) 操作，这样每个元素都可以添加到列表的末尾。（这假设不需要调整大小。）	插入和删除属于 O(log `n`) 操作。
比 SortedDictionary<TKey,TValue> 使用更少的内存。	比 SortedList 非泛型类和 SortedList<TKey,TValue> 泛型类使用更多内存。

对于必须可通过多个线程并发访问的已排序列表或字典，可以向派生自 ConcurrentDictionary<TKey,TValue> 的类添加排序逻辑。考虑不可变性时，以下相应的不可变类型遵循类似的排序语义：ImmutableSortedSet<T> 和 ImmutableSortedDictionary<TKey,TValue>。

> **NOTE**
> 对于包含自己的键的值（例如，包含雇员 ID 编号的雇员记录），可以通过派生自 KeyedCollection<TKey,TItem> 泛型类，创建具有列表和字典的某些特性的键控集合。

从 .NET Framework 4 开始，SortedSet<T> 类提供在执行插入、删除和搜索操作之后让数据一直按排序顺序排列的自平衡树。此类和 HashSet<T> 类实现 ISet<T> 接口。

请参阅

- System.Collections.IDictionary
- System.Collections.Generic.IDictionary<TKey,TValue>
- ConcurrentDictionary<TKey,TValue>
- 常用的集合类型

哈希表和字典集合类型

2021/11/16 •

System.Collections.Hashtable 类以及 System.Collections.Generic.Dictionary<TKey,TValue> 和 System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue> 泛型类实现 System.Collections.IDictionary 接口。Dictionary<TKey,TValue> 泛型类还实现 IDictionary<TKey,TValue> 泛型接口。因此，这些集合中的每个元素都是一个键值对。

Hashtable 由包含集合元素的存储桶组成。存储桶是 Hashtable 中元素的虚拟子组，与在大多数集合中进行搜索和检索相比，其搜索和检索更加容易和快速。每个存储桶都与一个哈希代码相关联，该哈希代码通过哈希函数生成并基于元素的键。

泛型 HashSet<T> 类是用于包含唯一元素的无序集合。

哈希函数是一种算法，返回基于键的数值哈希代码。该键是所存储对象的某个属性的值。哈希函数必须始终返回同一个键的同一哈希代码。哈希函数有可能为两个不同的键生成相同的哈希代码，但从哈希表中检索元素时，为每个唯一的键生成唯一哈希代码的哈希函数具有更好的性能。

在 Hashtable 中用作元素的每个对象必须能够通过使用 GetHashCode 方法的实现为自身生成哈希代码。但是，还可以为 Hashtable 中的所有元素指定哈希函数，方法是使用接受 IHashCodeProvider 实现作为其参数之一的 Hashtable 构造函数。

当将对象添加到 Hashtable时，其存储在与哈希代码相关联的存储桶中，此哈希代码匹配该对象的哈希代码。当在 Hashtable 中对一个值进行搜索时，则为该值生成哈希代码，并搜索与该哈希代码相关联的存储桶。

例如，用于字符串的哈希函数可能采用字符串中每个字符的 ASCII 代码，并将它们加总以生成哈希代码。字符串“picnic”的哈希代码可能与字符串“basket”的哈希代码不同；因此，字符串“picnic”和“basket”可能在不同的存储桶中。与此相反，“stressed”和“desserts”可能具有相同的哈希代码，并且位于同一个存储桶中。

Dictionary<TKey,TValue> 和 ConcurrentDictionary<TKey,TValue> 类具有与 Hashtable 类相同的功能。特定类型（不包括 Object）的 Dictionary<TKey,TValue> 与 Hashtable 相比可为值类型提供更好的性能。这是因为 Hashtable 的元素属于 Object 类型；因此，装箱和取消装箱通常发生在存储或检索值类型时。可能有多个线程同时访问该集合时，应使用 ConcurrentDictionary<TKey,TValue> 类。

另请参阅

- Hashtable
- IDictionary
- IHashCodeProvider
- Dictionary<TKey,TValue>
- System.Collections.Generic.IDictionary<TKey,TValue>
- System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>
- 常用的集合类型

线程安全集合

2021/11/16 •

.NET Framework 4 引入了 System.Collections.Concurrent 命名空间，其中包含多个线程安全且可缩放的集合类。多个线程可以安全高效地从这些集合添加或删除项，而无需在用户代码中进行其他同步。编写新代码时，只要将多个线程同时写入到集合时，就使用并发集合类。如果仅从共享集合进行读取，则可使用 System.Collections.Generic 命名空间中的类。建议不要使用 1.0 集合类，除非需要定位 .NET Framework 1.1 或更低版本运行时。

.NET Framework 1.0 和 2.0 集合中的线程同步

.NET Framework 1.0 中引入的集合位于 System.Collections 命名空间中。这些集合（包括常用的 ArrayList 和 Hashtable）通过 `Synchronized` 属性（此属性围绕集合返回线程安全的包装器）提供一些线程安全性。该包装器通过对每个添加或删除操作锁定整个集合进行工作。因此，每个尝试访问集合的线程必须等待，直到轮到它获取锁定。这不可缩放，并且可能导致大型集合的性能显著下降。此外，这一设计并不能完全防止争用情况的出现。有关详细信息，请参阅泛型集合中的同步。

.NET Framework 2.0 中引入的集合类位于 System.Collections.Generic 命名空间中。它们包括 List<T>、Dictionary<TKey,TValue> 等。与 .NET Framework 1.0 类相比，这些类提升了类型安全性和性能。不过，.NET Framework 2.0 集合类不提供任何线程同步；多线程同时添加或删除项时，用户代码必须提供所有同步。

建议使用 .NET Framework 4 中的并发集合类，因为它们不仅能够提供 .NET Framework 2.0 集合类的类型安全性，而且能够比 .NET Framework 1.0 集合更高效完整地提供线程安全性。

细粒度锁定和无锁机制

某些并发集合类型使用轻量同步机制，如 SpinLock、SpinWait、SemaphoreSlim 和 CountdownEvent，这些机制是 .NET Framework 4 中的新增功能。这些同步类型通常在将线程真正置于等待状态之前，会在短时间内使用 *忙旋转*。预计等待时间非常短时，旋转比等待所消耗的计算资源少得多，因为后者涉及资源消耗量大的内核转换。对于使用旋转的集合类，这种效率意味着多个线程能够以非常快的速率添加和删除项。有关旋转与锁定的详细信息，请参阅 SpinLock 和 SpinWait。

ConcurrentQueue<T> 和 ConcurrentStack<T> 类完全不使用锁定。相反，它们依赖于 Interlocked 操作来实现线程安全性。

> **NOTE**
> 由于并发集合类支持 ICollection，因此该类可提供针对 IsSynchronized 和 SyncRoot 属性的实现，即使这些属性不相关。`IsSynchronized` 始终返回 `false`，而 `SyncRoot` 始终为 `null`（在 Visual Basic 中是 `Nothing`）。

下表列出了 System.Collections.Concurrent 命名空间中的集合类型。

"	"
BlockingCollection<T>	为实现 IProducerConsumerCollection<T> 的所有类型提供限制和阻止功能。有关详细信息，请参阅 BlockingCollection 概述。
ConcurrentDictionary<TKey,TValue>	键值对字典的线程安全实现。

ConcurrentQueue<T>	FIFO（先进先出）队列的线程安全实现。
ConcurrentStack<T>	LIFO（后进先出）堆栈的线程安全实现。
ConcurrentBag<T>	无序元素集合的线程安全实现。
IProducerConsumerCollection<T>	类型必须实现以在 `BlockingCollection` 中使用的接口。

相关主题

TITLE	
BlockingCollection 概述	描述 BlockingCollection<T> 类型提供的功能。
如何：在 ConcurrentDictionary 中添加和移除项	描述如何从 ConcurrentDictionary<TKey,TValue> 添加和删除元素
如何：在 BlockingCollection 中逐个添加和取出项	描述如何在不使用只读枚举器的情况下，从阻止的集合添加和检索项。
如何：向集合添加限制和阻塞功能	描述如何将任一集合类用作 IProducerConsumerCollection<T> 集合的基础存储机制。
如何：使用 ForEach 移除 BlockingCollection 中的项	介绍了如何使用 `foreach`（在 Visual Basic 中是 `For Each`）在锁定集合中删除所有项。
如何：在管道中使用阻塞集合的数组	描述如何同时使用多个阻塞集合来实现一个管道。
如何：使用 ConcurrentBag 创建目标池	演示如何使用并发包在可重用对象（而不是继续创建新对象）的情况下改进性能。

参考

System.Collections.Concurrent

委托和 lambda

2021/11/16 •

委托定义指定特定方法签名的类型。可将满足此签名的方法(静态或实例)分配给该类型的变量，然后(使用适当参数)直接调用该方法，或将其作为参数本身传递给另一方法再进行调用。以下示例演示了委托的用法。

```
using System;
using System.Linq;

public class Program
{
    public delegate string Reverse(string s);

    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Reverse rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

- `public delegate string Reverse(string s);` 行创建特定签名的委托类型，在本例中即接受字符串参数并返回字符串参数的方法。
- `static string ReverseString(string s)` 方法与定义的委托类型具有完全相同的签名，用于实现委托。
- `Reverse rev = ReverseString;` 行显示可将方法分配给相应委托类型的变量。
- `Console.WriteLine(rev("a string"));` 行演示如何使用委托类型的变量来调用委托。

为简化开发过程，.NET 包含一组委托类型，程序员可重用这些类型而无需创建新类型。这些类型是 `Func<>`、`Action<>` 和 `Predicate<>`，可以使用它们而无需定义新的委托类型。这三种类型之间有一些区别，与它们的预期使用方式有关：

- `Action<>` 用于需要使用委托参数执行操作的情况。它所封装的方法不返回值。
- `Func<>` 通常用于现有转换的情况，也就是说需要将委托参数转换为其他结果时。投影是一个很好的示例。它所封装的方法返回指定值。
- `Predicate<>` 用于需要确定参数是否满足委托条件的情况。它也可以编写为 `Func<T, bool>`，这意味着方法返回布尔值。

现在可使用 `Func<>` 委托而非自定义类型重新编写上述示例。程序将照旧继续运行。

```
using System;
using System.Linq;

public class Program
{
    static string ReverseString(string s)
    {
        return new string(s.Reverse().ToArray());
    }

    static void Main(string[] args)
    {
        Func<string, string> rev = ReverseString;

        Console.WriteLine(rev("a string"));
    }
}
```

对于此简单示例而言，在 `Main` 方法之外定义方法似乎有些多余。.NET Framework 2.0 引入了匿名委托的概念，使你可以创建“内联”委托，而无需指定任何其他类型或方法。

在下面的示例中，匿名委托将列表筛选为只包含偶数，然后将它们打印到控制台。

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(
          delegate (int no)
          {
              return (no % 2 == 0);
          }
        );

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

如你所见，该委托的正文只是一组表达式，其他所有委托也是如此。但它并非单独定义，而是在调用 List<T>.FindAll 方法时临时引入。

但是，即使使用此方法，仍有许多可以丢弃的代码。此时就需要使用 *lambda 表达式*。lambda 表达式（或简称“lambda”）在 C# 3.0 中作为语言集成查询 (LINQ) 的核心构建基块被引入。这种表达式只是使用委托的更方便的语法。它们将声明签名和方法正文，但在分配到委托之前没有自己的正式标识。与委托不同，可将其作为事件注册的右侧内容或在各种 LINQ 子句和方法中直接分配。

由于 lambda 表达式只是指定委托的另一种方式，因此应可重新编写上述示例，令其使用 lambda 表达式而不是匿名委托。

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main(string[] args)
    {
        List<int> list = new List<int>();

        for (int i = 1; i <= 100; i++)
        {
            list.Add(i);
        }

        List<int> result = list.FindAll(i => i % 2 == 0);

        foreach (var item in result)
        {
            Console.WriteLine(item);
        }
    }
}
```

在前面的示例中，所使用的 Lambda 表达式为 `i => i % 2 == 0`。同样，这只是使用委托的便捷语法。内部原理与匿名委托相似。

再次强调，lambda 只是委托，这意味着可将其顺利用作事件处理程序，如以下代码片段所示。

```
public MainWindow()
{
    InitializeComponent();

    Loaded += (o, e) =>
    {
        this.Title = "Loaded";
    };
}
```

此上下文中的 `+=` 运算符用于订阅事件。有关详细信息，请参阅如何订阅和取消订阅事件。

其他阅读材料和资源

- 委托
- Lambda 表达式

处理和引发事件

2021/11/16 •

.NET 中的事件基于委托模型。委托模型遵循观察者设计模式，使订阅者能够向提供方注册并接收相关通知。事件发送方推送事件发生的通知，事件接收器接收该通知并定义对它的响应。本文介绍委托模型的主要组件、如何在应用程序中使用事件以及如何在你的代码中实现事件。

事件

事件是由对象发送的用于发出操作信号的消息。该操作可能是由用户交互引起，例如单击按钮；也可能是由某些其他程序的逻辑导致，例如更改属性值。引发事件的对象称为“事件发送方”。事件发送方不知道哪个对象或方法将接收（处理）它引发的事件。事件通常是事件发送方的成员，例如 Click 事件是 Button 类的成员，PropertyChanged 事件是实现 INotifyPropertyChanged 接口的类的成员。

若要定义一个事件，可以在事件类签名中使用 `event`（在 C# 中）或 `Event`（在 Visual Basic 中）关键字，并指定事件的委托类型。委托在下一节中介绍。

通常，为了引发事件，您可以在 C# 中添加一个标记为 `protected` 和 `virtual` 或在 Visual Basic 中标记为 `Protected` 和 `Overridable` 的方法。将此方法命名为 `On` *EventName*；例如，`OnDataReceived`。此方法应接受一个指定事件数据对象（EventArgs 类型或派生类型）的参数。您提供此方法以允许派生类重写引发事件的逻辑。派生类应始终调用基类的 `On` *EventName* 方法，以确保注册的委托接收事件。

下面的示例显示如何声明名为 `ThresholdReached` 事件。该事件与 EventHandler 委托相关联并且由 `OnThresholdReached` 方法引发。

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the class
}
```

```
Public Class Counter
    Public Event ThresholdReached As EventHandler

    Protected Overridable Sub OnThresholdReached(e As EventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    ' provide remaining implementation for the class
End Class
```

委托

委托是一种保存对方法的引用的类型。委托是通过显示所引用方法的返回类型和参数的签名来声明的，并可以仅保存与其签名匹配的方法的引用。因此，委托等同于类型安全函数指针或回叫。委托声明足以定义委托类。

委托在 .NET 中有许多用途。在事件上下文中，委托是事件源和处理事件的代码之间的媒介（或类似指针的机制）。如上一节的例子所示，可以通过在事件声明中包括委托类型来将委托和事件相关联。有关委托的详细信息，请参阅 Delegate 类。

.NET 提供了 EventHandler 和 EventHandler<TEventArgs> 委托来支持大部分事件场景。使用 EventHandler 委托处理不包含事件数据的所有事件。使用 EventHandler<TEventArgs> 委托处理包含事件相关数据的事件。这些委托没有返回类型值，并且接受两个参数（事件源的对象和事件数据的对象）。

委托是多播，这意味着它们可以保存对多个事件处理方法的引用。有关详细信息，请参阅 Delegate 参考页。委托提供了事件处理中的灵活性和精确控制。委托人通过维护事件的已注册事件处理程序列表来充当引发事件的类的事件调度程序。

在 EventHandler 和 EventHandler<TEventArgs> 委托不可用的场景下，您可以定义一个委托。要求你定义委托的场景非常少见的，例如，当你必须处理无法识别泛型的代码时。在声明中使用 `delegate`（在 C# 中）和 `Delegate`（在 Visual Basic 中）关键字标记委托。下面的示例说明如何声明 `ThresholdReachedEventHandler` 委托。

```
public delegate void ThresholdReachedEventHandler(object sender, ThresholdReachedEventArgs e);
```

```
Public Delegate Sub ThresholdReachedEventHandler(sender As Object, e As ThresholdReachedEventArgs)
```

事件数据

与事件相关的数据可以通过事件数据类提供。.NET 提供了许多事件数据类，用户可以在自己的应用程序中使用它们。例如，SerialDataReceivedEventArgs 类是 SerialPort.DataReceived 事件的事件数据类。.NET 遵循所有事件数据类以 `EventArgs` 结尾的命名模式。您通过查看事件的委托来确定哪个事件数据类与事件相关联。例如，SerialDataReceivedEventHandler 委托包含 SerialDataReceivedEventArgs 类作为它的一个参数。

EventArgs 类是所有事件数据类的基类型。当一个事件没有任何与其相关联的数据时，您也会用到 EventArgs 类。当您创建一个事件仅用来通知其他类出问题了，不需要传递任何数据时，请包括 EventArgs 类作为委托中的第二个参数。当没有数据提供时，您可以传递 EventArgs.Empty 值。EventHandler 委托包括 EventArgs 类作为一个参数。

当您想创建一个自定义的事件数据类时，请创建一个派生自 EventArgs 的类，然后提供所需的所有成员，来传递与该事件相关的数据。通常，应使用与 .NET 相同的命名模式，并且事件数据类名称应以 `EventArgs` 结尾。

下面的示例演示了一个名为 `ThresholdReachedEventArgs` 的事件数据类。它包含特定于引发事件的属性。

```
public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

```
Public Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class
```

事件处理程序

若要响应事件，您需要在事件接收器中定义一个事件处理程序方法。此方法必须与您正处理的事件的委托签名

匹配。在事件处理程序中，当事件引发时会执行所需操作，例如在用户单击按钮之后收集用户输入。若当事件发生时收到通知，您的事件处理程序方法必须订阅该事件。

下面的示例演示与 `c_ThresholdReached` 委托的签名匹配的 EventHandler 事件处理程序方法。该方法订阅 `ThresholdReached` 事件。

```
class Program
{
    static void Main()
    {
        var c = new Counter();
        c.ThresholdReached += c_ThresholdReached;

        // provide remaining implementation for the class
    }

    static void c_ThresholdReached(object sender, EventArgs e)
    {
        Console.WriteLine("The threshold was reached.");
    }
}
```

```
Module Module1

    Sub Main()
        Dim c As New Counter()
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        ' provide remaining implementation for the class
    End Sub

    Sub c_ThresholdReached(sender As Object, e As EventArgs)
        Console.WriteLine("The threshold was reached.")
    End Sub
End Module
```

静态和动态事件处理程序

借助 .NET，订阅者可以进行静态或动态注册以获得事件通知。对于其事件由静态事件处理程序进行处理的类，静态事件处理程序对其整个生命周期有效。通常为响应某些条件程序逻辑，会在程序执行期间显式激活和停用动态事件处理程序。例如，如果仅在特定条件下需要事件通知，或如果应用程序提供多个事件处理程序且由运行时条件定义要使用的适当事件处理程序，则可以使用动态事件处理程序。上一节中的示例演示如何动态添加事件处理程序。有关详细信息，请查看 Visual Basic 中的事件和 C# 中的事件。

引发多个事件

如果您的类引发多个事件，编译器会为每一个事件委托实例生成一个字段。如果事件数量很大，则可能无法接受按一个委托计算一个字段的存储成本。对于这些情况，.NET 提供一个事件属性，可以将其与选择的另一数据结构一起用于存储事件委托。

事件属性由事件声明和事件访问器组成。事件访问器是您定义的方法，用来从存储数据结构添加和移除事件委托实例。请注意，事件属性要比事件字段慢，这是因为必须先检索每个事件委托，然后才能调用它。需在内存和速度之间进行权衡。如果类定义许多不常引发的事件，那么需要实现事件属性。有关详细信息，请参阅如何：使用事件属性处理多个事件。

相关文章

TITLE
如何：抛出和使用事件	包含引发和使用事件的示例。
如何：使用事件属性处理多个事件	演示如何使用事件属性处理多个事件。
观察程序设计模式	描述允许订阅者向提供方注册和接收通知的设计模式。

请参阅

- EventHandler
- EventHandler<TEventArgs>
- EventArgs
- Delegate
- 事件 (Visual Basic)
- 事件（C# 编程指南）
- 事件和路由事件概述（UWP 应用）
- Windows 应用商店 8.x 应用中的事件

如何：引发事件和使用事件

2021/11/16 •

本主题中的示例演示如何处理事件。它们包含 EventHandler、EventHandler<TEventArgs> 委托和自定义委托的示例，用于说明包含数据和不包含数据的事件。

这些示例使用事件一文中介绍的概念。

示例

第一个示例演示如何引发和使用一个没有数据的事件。它包含一个名为 `Counter` 类，该类具有一个名为 `ThresholdReached` 的事件。当计数器值等于或者超过阈值时，将引发此事件。EventHandler 委托与此事件关联，因为没有提供任何事件数据。

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, EventArgs e)
        {
            Console.WriteLine("The threshold was reached.");
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReached?.Invoke(this, EventArgs.Empty);
            }
        }

        public event EventHandler ThresholdReached;
    }
}
```

```
Module Module1

    Sub Main()
        Dim c As Counter = New Counter(New Random().Next(10))
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        Console.WriteLine("press 'a' key to increase total")
        While Console.ReadKey(True).KeyChar = "a"
            Console.WriteLine("adding one")
            c.Add(1)
        End While
    End Sub

    Sub c_ThresholdReached(sender As Object, e As EventArgs)
        Console.WriteLine("The threshold was reached.")
        Environment.Exit(0)
    End Sub
End Module

Class Counter
    Private threshold As Integer
    Private total As Integer

    Public Sub New(passedThreshold As Integer)
        threshold = passedThreshold
    End Sub

    Public Sub Add(x As Integer)
        total = total + x
        If (total >= threshold) Then
            ThresholdReached?.Invoke(this. EventArgs.Empty)
        End If
    End Sub

    Public Event ThresholdReached As EventHandler
End Class
```

示例

下一个示例演示如何引发和使用提供数据的事件。EventHandler<TEventArgs> 委托与此事件关联，示例还提供了一个自定义事件数据对象的实例。

```
using namespace System;

public ref class ThresholdReachedEventArgs : public EventArgs
{
   public:
      property int Threshold;
      property DateTime TimeReached;
};

public ref class Counter
{
   private:
      int threshold;
      int total;

   public:
      Counter() {};

      Counter(int passedThreshold)
      {
         threshold = passedThreshold;
      }

      void Add(int x)
      {
          total += x;
          if (total >= threshold) {
             ThresholdReachedEventArgs^ args = gcnew ThresholdReachedEventArgs();
             args->Threshold = threshold;
             args->TimeReached = DateTime::Now;
             OnThresholdReached(args);
          }
      }

      event EventHandler<ThresholdReachedEventArgs^>^ ThresholdReached;

   protected:
      virtual void OnThresholdReached(ThresholdReachedEventArgs^ e)
      {
         ThresholdReached(this, e);
      }
};

public ref class SampleHandler
{
   public:
      static void c_ThresholdReached(Object^ sender, ThresholdReachedEventArgs^ e)
      {
         Console::WriteLine("The threshold of {0} was reached at {1}.",
                            e->Threshold,  e->TimeReached);
         Environment::Exit(0);
      }
};

void main()
{
   Counter^ c = gcnew Counter((gcnew Random())->Next(10));
   c->ThresholdReached += gcnew EventHandler<ThresholdReachedEventArgs^>(SampleHandler::c_ThresholdReached);

   Console::WriteLine("press 'a' key to increase total");
   while (Console::ReadKey(true).KeyChar == 'a') {
      Console::WriteLine("adding one");
      c->Add(1);
   }
}
```

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold,  e.TimeReached);
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
                args.Threshold = threshold;
                args.TimeReached = DateTime.Now;
                OnThresholdReached(args);
            }
        }

        protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
        {
            EventHandler<ThresholdReachedEventArgs> handler = ThresholdReached;
            if (handler != null)
            {
                handler(this, e);
            }
        }

        public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;
    }

    public class ThresholdReachedEventArgs : EventArgs
    {
        public int Threshold { get; set; }
        public DateTime TimeReached { get; set; }
    }
}
```

```
Module Module1

    Sub Main()
        Dim c As Counter = New Counter(New Random().Next(10))
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        Console.WriteLine("press 'a' key to increase total")
        While Console.ReadKey(True).KeyChar = "a"
            Console.WriteLine("adding one")
            c.Add(1)
        End While
    End Sub

    Sub c_ThresholdReached(sender As Object, e As ThresholdReachedEventArgs)
        Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold, e.TimeReached)
        Environment.Exit(0)
    End Sub
End Module

Class Counter
    Private threshold As Integer
    Private total As Integer

    Public Sub New(passedThreshold As Integer)
        threshold = passedThreshold
    End Sub

    Public Sub Add(x As Integer)
        total = total + x
        If (total >= threshold) Then
            Dim args As ThresholdReachedEventArgs = New ThresholdReachedEventArgs()
            args.Threshold = threshold
            args.TimeReached = DateTime.Now
            OnThresholdReached(args)
        End If
    End Sub

    Protected Overridable Sub OnThresholdReached(e As ThresholdReachedEventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    Public Event ThresholdReached As EventHandler(Of ThresholdReachedEventArgs)
End Class

Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class
```

示例

下一个示例演示如何声明事件的委托。该委托名为 `ThresholdReachedEventHandler`。这只是一个示例。通常不需要为事件声名委托，因为可以使用 EventHandler 或者 EventHandler<TEventArgs> 委托。只有在极少数情况下才应声明委托，例如，在向无法使用泛型的旧代码提供类时，就需要如此。

```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(Object sender, ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold, e.TimeReached);
            Environment.Exit(0);
        }
    }

    class Counter
    {
        private int threshold;
        private int total;

        public Counter(int passedThreshold)
        {
            threshold = passedThreshold;
        }

        public void Add(int x)
        {
            total += x;
            if (total >= threshold)
            {
                ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
                args.Threshold = threshold;
                args.TimeReached = DateTime.Now;
                OnThresholdReached(args);
            }
        }

        protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
        {
            ThresholdReachedEventHandler handler = ThresholdReached;
            if (handler != null)
            {
                handler(this, e);
            }
        }

        public event ThresholdReachedEventHandler ThresholdReached;
    }

    public class ThresholdReachedEventArgs : EventArgs
    {
        public int Threshold { get; set; }
        public DateTime TimeReached { get; set; }
    }

    public delegate void ThresholdReachedEventHandler(Object sender, ThresholdReachedEventArgs e);
}
```

```
Module Module1

    Sub Main()
        Dim c As Counter = New Counter(New Random().Next(10))
        AddHandler c.ThresholdReached, AddressOf c_ThresholdReached

        Console.WriteLine("press 'a' key to increase total")
        While Console.ReadKey(True).KeyChar = "a"
            Console.WriteLine("adding one")
            c.Add(1)
        End While
    End Sub

    Sub c_ThresholdReached(sender As Object, e As ThresholdReachedEventArgs)
        Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold, e.TimeReached)
        Environment.Exit(0)
    End Sub
End Module

Class Counter
    Private threshold As Integer
    Private total As Integer

    Public Sub New(passedThreshold As Integer)
        threshold = passedThreshold
    End Sub

    Public Sub Add(x As Integer)
        total = total + x
        If (total >= threshold) Then
            Dim args As ThresholdReachedEventArgs = New ThresholdReachedEventArgs()
            args.Threshold = threshold
            args.TimeReached = DateTime.Now
            OnThresholdReached(args)
        End If
    End Sub

    Protected Overridable Sub OnThresholdReached(e As ThresholdReachedEventArgs)
        RaiseEvent ThresholdReached(Me, e)
    End Sub

    Public Event ThresholdReached As ThresholdReachedEventHandler
End Class

Public Class ThresholdReachedEventArgs
    Inherits EventArgs

    Public Property Threshold As Integer
    Public Property TimeReached As DateTime
End Class

Public Delegate Sub ThresholdReachedEventHandler(sender As Object, e As ThresholdReachedEventArgs)
```

请参阅

- 事件

如何：使用事件属性处理多个事件

2021/11/16 •

若要使用事件属性，请在引发事件的类中定义事件属性，然后在处理事件的类中设置事件属性的委托。若要在类中实现多个事件属性，此类必须在内部存储和维护为每个事件定义的委托。典型的方法是实现通过事件键索引的委托集合。

若要存储每个事件的委托，则可以使用 EventHandlerList 类，或实现自己的集合。集合类必须基于事件键提供用于设置、访问和检索事件处理程序委托的方法。例如，可以使用 Hashtable 类，或从 DictionaryBase 类中派生一个自定义的类。委托集合的实现细节不需要在你的类的外部公开。

类中的每个事件属性都定义了一个 add 访问器方法和一个 remove 访问器方法。事件属性的 add 访问器将输入委托实例添加到委托集合。事件属性的 remove 访问器将输入委托实例从委托集合中移除。事件属性访问器使用的预定义的事件属性键来将实例添加到委托集合或从中将实例移除。

使用事件属性处理多个事件

1. 定义引发事件的类中的委托集合。
2. 定义每个事件的键。
3. 定义引发事件的类中的事件属性。
4. 使用委托集合实现添加和移除事件属性的 add 和 remove 访问器方法。
5. 使用公共事件属性来添加和移除处理事件的类中的事件处理程序委托。

示例

下面的 C# 示例实现事件属性 `MouseDown` 和 `MouseUp`，其中使用 EventHandlerList 来存储每个事件的委托。事件属性构造的关键字是粗体类型。

```
// The class SampleControl defines two event properties, MouseUp and MouseDown.
ref class SampleControl : Component
{
    // :
    // Define other control methods and properties.
    // :

    // Define the delegate collection.
protected:
    EventHandlerList^ listEventDelegates;

private:
    // Define a unique key for each event.
    static Object^ mouseDownEventKey = gcnew Object();
    static Object^ mouseUpEventKey = gcnew Object();

    // Define the MouseDown event property.
public:
    SampleControl()
    {
        listEventDelegates = gcnew EventHandlerList();
    }

    event MouseEventHandler^ MouseDown
    {
        // Add the input delegate to the collection.
        void add(MouseEventHandler^ value)
        {
            listEventDelegates->AddHandler(mouseDownEventKey, value);
        }
        // Remove the input delegate from the collection.
        void remove(MouseEventHandler^ value)
        {
            listEventDelegates->RemoveHandler(mouseDownEventKey, value);
        }
        // Raise the event with the delegate specified by mouseDownEventKey
        void raise(Object^ sender, MouseEventArgs^ e)
        {
            MouseEventHandler^ mouseEventDelegate =
                (MouseEventHandler^)listEventDelegates[mouseDownEventKey];
            mouseEventDelegate(sender, e);
        }
    }

    // Define the MouseUp event property.
    event MouseEventHandler^ MouseUp
    {
        // Add the input delegate to the collection.
        void add(MouseEventHandler^ value)
        {
            listEventDelegates->AddHandler(mouseUpEventKey, value);
        }
        // Remove the input delegate from the collection.
        void remove(MouseEventHandler^ value)
        {
            listEventDelegates->RemoveHandler(mouseUpEventKey, value);
        }
        // Raise the event with the delegate specified by mouseUpEventKey
        void raise(Object^ sender, MouseEventArgs^ e)
        {
            MouseEventHandler^ mouseEventDelegate =
                (MouseEventHandler^)listEventDelegates[mouseUpEventKey];
            mouseEventDelegate(sender, e);
        }
    }
};
```

```
// The class SampleControl defines two event properties, MouseUp and MouseDown.
class SampleControl : Component
{
    // :
    // Define other control methods and properties.
    // :

    // Define the delegate collection.
    protected EventHandlerList listEventDelegates = new EventHandlerList();

    // Define a unique key for each event.
    static readonly object mouseDownEventKey = new object();
    static readonly object mouseUpEventKey = new object();

    // Define the MouseDown event property.
    public event MouseEventHandler MouseDown
    {
        // Add the input delegate to the collection.
        add
        {
            listEventDelegates.AddHandler(mouseDownEventKey, value);
        }
        // Remove the input delegate from the collection.
        remove
        {
            listEventDelegates.RemoveHandler(mouseDownEventKey, value);
        }
    }

    // Raise the event with the delegate specified by mouseDownEventKey
    private void OnMouseDown(MouseEventArgs e)
    {
        MouseEventHandler mouseEventDelegate =
            (MouseEventHandler)listEventDelegates[mouseDownEventKey];
        mouseEventDelegate(this, e);
    }

    // Define the MouseUp event property.
    public event MouseEventHandler MouseUp
    {
        // Add the input delegate to the collection.
        add
        {
            listEventDelegates.AddHandler(mouseUpEventKey, value);
        }
        // Remove the input delegate from the collection.
        remove
        {
            listEventDelegates.RemoveHandler(mouseUpEventKey, value);
        }
    }

    // Raise the event with the delegate specified by mouseUpEventKey
    private void OnMouseUp(MouseEventArgs e)
    {
        MouseEventHandler mouseEventDelegate =
            (MouseEventHandler)listEventDelegates[mouseUpEventKey];
        mouseEventDelegate(this, e);
    }
}
```

```
' The class SampleControl defines two event properties, MouseUp and MouseDown.
Class SampleControl
    Inherits Component
    ' :
    ' Define other control methods and properties.
    ' :

    ' Define the delegate collection.
    Protected listEventDelegates As New EventHandlerList()

    ' Define a unique key for each event.
    Shared ReadOnly mouseDownEventKey As New Object()
    Shared ReadOnly mouseUpEventKey As New Object()

    ' Define the MouseDown event property.
    Public Custom Event MouseDown As MouseEventHandler
        ' Add the input delegate to the collection.
        AddHandler(Value As MouseEventHandler)
            listEventDelegates.AddHandler(mouseDownEventKey, Value)
        End AddHandler
        ' Remove the input delegate from the collection.
        RemoveHandler(Value As MouseEventHandler)
            listEventDelegates.RemoveHandler(mouseDownEventKey, Value)
        End RemoveHandler
        ' Raise the event with the delegate specified by mouseDownEventKey
        RaiseEvent(sender As Object, e As MouseEventArgs)
            Dim mouseEventDelegate As MouseEventHandler = _
                listEventDelegates(mouseDownEventKey)
            mouseEventDelegate(sender, e)
        End RaiseEvent
    End Event

    ' Define the MouseUp event property.
    Public Custom Event MouseUp As MouseEventHandler
        ' Add the input delegate to the collection.
        AddHandler(Value As MouseEventHandler)
            listEventDelegates.AddHandler(mouseUpEventKey, Value)
        End AddHandler
        ' Remove the input delegate from the collection.
        RemoveHandler(Value As MouseEventHandler)
            listEventDelegates.RemoveHandler(mouseUpEventKey, Value)
        End RemoveHandler
        ' Raise the event with the delegate specified by mouseDownUpKey
        RaiseEvent(sender As Object, e As MouseEventArgs)
            Dim mouseEventDelegate As MouseEventHandler = _
                listEventDelegates(mouseUpEventKey)
            mouseEventDelegate(sender, e)
        End RaiseEvent
    End Event
End Class
```

请参阅

- System.ComponentModel.EventHandlerList
- 事件
- Control.Events
- 如何:声明自定义事件以节省内存

观察者设计模式

2021/11/16 •

观察者设计模式使订阅者能够从提供程序注册并接收通知。它适用于需要基于推送的通知的任何方案。此模式定义提供程序(亦称为"使用者"或"可观察对象"),以及零个、一个或多个观察者。观察者注册提供程序,并且每当预定义的条件、事件或状态发生更改时,该提供程序会通过调用其方法之一来自动通知所有观察者。在此方法调用中,该提供程序还可向观察者提供当前状态信息。在 .NET 中,通过实现泛型 System.IObservable<T> 和 System.IObserver<T> 接口来应用观察者设计模式。泛型类型参数表示提供通知信息的类型。

应用模式

观察者设计模式适用于分布式基于推送的通知,因为它支持两个不同的组件或应用程序层(如数据源(业务逻辑)层和用户界面(显示)层)之间完全分离。每当提供程序使用回调向其客户端提供当前信息时,均可以实现此模式。

实现此模式要求提供以下内容:

- 提供程序或主题,即将通知发送给观察者的对象。提供程序是实现 IObservable<T> 接口的类或结构。提供程序必须实现单个方法 IObservable<T>.Subscribe,该方法由希望从提供程序接收通知的观察者调用。
- 观察者,即从提供程序接收通知的对象。观察者是实现 IObserver<T> 接口的类或结构。观察者必须实现以下三个方法,这三个方法均由提供程序调用:
  - IObserver<T>.OnNext,它向观察者提供新信息或当前信息。
  - IObserver<T>.OnError,它通知观察者已发生错误。
  - IObserver<T>.OnCompleted,它指示提供程序已完成发送通知。
- 允许提供程序跟踪观察者的一种机制。通常情况下,提供程序使用容器对象(如 System.Collections.Generic.List<T> 对象)来保存对已订阅通知的 IObserver<T> 实现的引用。将存储容器用于此目的使提供程序能够处理零到无限数量的观察者。未定义观察者接收通知的顺序;提供程序可以随意使用任何方法来确定顺序。
- IDisposable 实现,它使提供程序在能够通知完成时删除观察者。观察者从 Subscribe 方法接收对 IDisposable 实现的引用,因此它们还可以调用 IDisposable.Dispose 方法,以便在提供程序已完成发送通知之前取消订阅。
- 包含提供程序发送到其观察者的数据的对象。此对象的类型对应 IObservable<T> 和 IObserver<T> 接口的泛型类型参数。尽管此对象可与 IObservable<T> 实现相同,但通常情况下,它是一个单独的类型。

> **NOTE**
> 除实现观察者设计模式外,你还可能对浏览使用 IObservable<T> 和 IObserver<T> 接口构建的库感兴趣。例如,Reactive Extensions for .NET (Rx) 包含一组支持异步编程的扩展方法和 LINQ 标准序列运算符。

实现模式

下面的示例使用观察者设计模式来实现机场行李认领信息系统。`BaggageInfo` 类提供有关到达航班以及可领取每次航班行李的行李传送带的信息。如以下示例所示。

```
using System;
using System.Collections.Generic;

public class BaggageInfo
{
   private int flightNo;
   private string origin;
   private int location;

   internal BaggageInfo(int flight, string from, int carousel)
   {
      this.flightNo = flight;
      this.origin = from;
      this.location = carousel;
   }

   public int FlightNumber {
      get { return this.flightNo; }
   }

   public string From {
      get { return this.origin; }
   }

   public int Carousel {
      get { return this.location; }
   }
}
```

```
Public Class BaggageInfo
    Private flightNo As Integer
    Private origin As String
    Private location As Integer

    Friend Sub New(ByVal flight As Integer, ByVal from As String, ByVal carousel As Integer)
        Me.flightNo = flight
        Me.origin = from
        Me.location = carousel
    End Sub

    Public ReadOnly Property FlightNumber As Integer
        Get
            Return Me.flightNo
        End Get
    End Property

    Public ReadOnly Property From As String
        Get
            Return Me.origin
        End Get
    End Property

    Public ReadOnly Property Carousel As Integer
        Get
            Return Me.location
        End Get
    End Property
End Class
```

`BaggageHandler` 类负责接收有关到达航班和行李认领传送带的信息。在内部，它维护两个集合：

- `observers` - 将接收更新的信息的客户端集合。
- `flights` - 航班及其指定行李传送带的集合。

这两个集合都由在 `BaggageHandler` 类构造函数中实例化的泛型 List<T> 对象表示。下面的示例演示了 `BaggageHandler` 类的源代码。

```
public class BaggageHandler : IObservable<BaggageInfo>
{
   private List<IObserver<BaggageInfo>> observers;
   private List<BaggageInfo> flights;

   public BaggageHandler()
   {
      observers = new List<IObserver<BaggageInfo>>();
      flights = new List<BaggageInfo>();
   }

   public IDisposable Subscribe(IObserver<BaggageInfo> observer)
   {
      // Check whether observer is already registered. If not, add it
      if (! observers.Contains(observer)) {
         observers.Add(observer);
         // Provide observer with existing data.
         foreach (var item in flights)
            observer.OnNext(item);
      }
      return new Unsubscriber<BaggageInfo>(observers, observer);
   }

   // Called to indicate all baggage is now unloaded.
   public void BaggageStatus(int flightNo)
   {
      BaggageStatus(flightNo, String.Empty, 0);
   }

   public void BaggageStatus(int flightNo, string from, int carousel)
   {
      var info = new BaggageInfo(flightNo, from, carousel);

      // Carousel is assigned, so add new info object to list.
      if (carousel > 0 && ! flights.Contains(info)) {
         flights.Add(info);
         foreach (var observer in observers)
            observer.OnNext(info);
      }
      else if (carousel == 0) {
         // Baggage claim for flight is done
         var flightsToRemove = new List<BaggageInfo>();
         foreach (var flight in flights) {
            if (info.FlightNumber == flight.FlightNumber) {
               flightsToRemove.Add(flight);
               foreach (var observer in observers)
                  observer.OnNext(info);
            }
         }
         foreach (var flightToRemove in flightsToRemove)
            flights.Remove(flightToRemove);

         flightsToRemove.Clear();
      }
   }

   public void LastBaggageClaimed()
   {
      foreach (var observer in observers)
         observer.OnCompleted();

      observers.Clear();
   }
}
```

```
Public Class BaggageHandler : Implements IObservable(Of BaggageInfo)

    Private observers As List(Of IObserver(Of BaggageInfo))
    Private flights As List(Of BaggageInfo)

    Public Sub New()
        observers = New List(Of IObserver(Of BaggageInfo))
        flights = New List(Of BaggageInfo)
    End Sub

    Public Function Subscribe(ByVal observer As IObserver(Of BaggageInfo)) As IDisposable _
                   Implements IObservable(Of BaggageInfo).Subscribe
        ' Check whether observer is already registered. If not, add it
        If Not observers.Contains(observer) Then
            observers.Add(observer)
            ' Provide observer with existing data.
            For Each item In flights
                observer.OnNext(item)
            Next
        End If
        Return New Unsubscriber(Of BaggageInfo)(observers, observer)
    End Function

    ' Called to indicate all baggage is now unloaded.
    Public Sub BaggageStatus(ByVal flightNo As Integer)
        BaggageStatus(flightNo, String.Empty, 0)
    End Sub

    Public Sub BaggageStatus(ByVal flightNo As Integer, ByVal from As String, ByVal carousel As Integer)
        Dim info As New BaggageInfo(flightNo, from, carousel)

        ' Carousel is assigned, so add new info object to list.
        If carousel > 0 And Not flights.Contains(info) Then
            flights.Add(info)
            For Each observer In observers
                observer.OnNext(info)
            Next
        ElseIf carousel = 0 Then
            ' Baggage claim for flight is done
            Dim flightsToRemove As New List(Of BaggageInfo)
            For Each flight In flights
                If info.FlightNumber = flight.FlightNumber Then
                    flightsToRemove.Add(flight)
                    For Each observer In observers
                        observer.OnNext(info)
                    Next
                End If
            Next
            For Each flightToRemove In flightsToRemove
                flights.Remove(flightToRemove)
            Next
            flightsToRemove.Clear()
        End If
    End Sub

    Public Sub LastBaggageClaimed()
        For Each observer In observers
            observer.OnCompleted()
        Next
        observers.Clear()
    End Sub
End Class
```

想要接收更新的信息的客户端调用 `BaggageHandler.Subscribe` 方法。如果客户端以前未订阅通知，则将对客户端的 IObserver<T> 实现的引用添加到 `observers` 集合中。

可调用重载的 `BaggageHandler.BaggageStatus` 方法来指示是正在卸载航班行李还是不再卸载航班行李。在第一种情况下，该方法传递航班号、航班起飞机场和正在卸载行李的传送带。在第二种情况下，该方法仅传递航班号。对于正在卸载的行李，该方法检查传递到方法的 `BaggageInfo` 信息是否存在于 `flights` 集合中。如果不存在，该方法将添加信息，并调用每个观察者的 `OnNext` 方法。对于不再卸载其行李的航班，该方法检查此航班的相关信息是否存储在 `flights` 集合中。如果是，则该方法调用每个观察者的 `OnNext` 方法，并从 `flights` 集合中删除 `BaggageInfo` 对象。

当一天中的最后一个航班已着陆并且已处理了其行李时，调用 `BaggageHandler.LastBaggageClaimed` 方法。此方法调用每个观察者的 `OnCompleted` 方法来指示所有通知均已完成，然后再清除 `observers` 集合。

提供程序的 Subscribe 方法将返回 IDisposable 实现，该实现可使观察者在调用 OnCompleted 方法之前停止接收通知。下面的示例演示了 `Unsubscriber(Of BaggageInfo)` 类的源代码。此类在 `BaggageHandler.Subscribe` 方法中实例化时，它将传递对 `observers` 集合的引用以及对添加到集合中的观察者的引用。这些引用被分配给局部变量。调用对象的 `Dispose` 方法时，它会检查观察者是否仍存在于 `observers` 集合中，如果存在，则删除观察者。

```
internal class Unsubscriber<BaggageInfo> : IDisposable
{
   private List<IObserver<BaggageInfo>> _observers;
   private IObserver<BaggageInfo> _observer;

   internal Unsubscriber(List<IObserver<BaggageInfo>> observers, IObserver<BaggageInfo> observer)
   {
      this._observers = observers;
      this._observer = observer;
   }

   public void Dispose()
   {
      if (_observers.Contains(_observer))
         _observers.Remove(_observer);
   }
}
```

```
Friend Class Unsubscriber(Of BaggageInfo) : Implements IDisposable
    Private _observers As List(Of IObserver(Of BaggageInfo))
    Private _observer As IObserver(Of BaggageInfo)

    Friend Sub New(ByVal observers As List(Of IObserver(Of BaggageInfo)), ByVal observer As IObserver(Of
BaggageInfo))
        Me._observers = observers
        Me._observer = observer
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        If _observers.Contains(_observer) Then
            _observers.Remove(_observer)
        End If
    End Sub
End Class
```

下面的示例提供了名为 `ArrivalsMonitor` 的 IObserver<T> 实现，它是一个显示行李认领信息的基类。根据始发城市的名称，按字母顺序显示信息。将 `ArrivalsMonitor` 的方法标记为 `overridable`（在 Visual Basic 中）或 `virtual`（在 C# 中），因此它们都可由派生类重写。

```
using System;
using System.Collections.Generic;

public class ArrivalsMonitor : IObserver<BaggageInfo>
{
```

```
{
   private string name;
   private List<string> flightInfos = new List<string>();
   private IDisposable cancellation;
   private string fmt = "{0,-20} {1,5}  {2, 3}";

   public ArrivalsMonitor(string name)
   {
      if (String.IsNullOrEmpty(name))
         throw new ArgumentNullException("The observer must be assigned a name.");

      this.name = name;
   }

   public virtual void Subscribe(BaggageHandler provider)
   {
      cancellation = provider.Subscribe(this);
   }

   public virtual void Unsubscribe()
   {
      cancellation.Dispose();
      flightInfos.Clear();
   }

   public virtual void OnCompleted()
   {
      flightInfos.Clear();
   }

   // No implementation needed: Method is not called by the BaggageHandler class.
   public virtual void OnError(Exception e)
   {
      // No implementation.
   }

   // Update information.
   public virtual void OnNext(BaggageInfo info)
   {
      bool updated = false;

      // Flight has unloaded its baggage; remove from the monitor.
      if (info.Carousel == 0) {
         var flightsToRemove = new List<string>();
         string flightNo = String.Format("{0,5}", info.FlightNumber);

         foreach (var flightInfo in flightInfos) {
            if (flightInfo.Substring(21, 5).Equals(flightNo)) {
               flightsToRemove.Add(flightInfo);
               updated = true;
            }
         }
         foreach (var flightToRemove in flightsToRemove)
            flightInfos.Remove(flightToRemove);

         flightsToRemove.Clear();
      }
      else {
         // Add flight if it does not exist in the collection.
         string flightInfo = String.Format(fmt, info.From, info.FlightNumber, info.Carousel);
         if (! flightInfos.Contains(flightInfo)) {
            flightInfos.Add(flightInfo);
            updated = true;
         }
      }
      if (updated) {
         flightInfos.Sort();
         Console.WriteLine("Arrivals information from {0}", this.name);
         foreach (var flightInfo in flightInfos)
            Console.WriteLine(flightInfo);
```

```
            Console.WriteLine(flightInfo);

         Console.WriteLine();
      }
   }
}
```

```
Public Class ArrivalsMonitor : Implements IObserver(Of BaggageInfo)
    Private name As String
    Private flightInfos As New List(Of String)
    Private cancellation As IDisposable
    Private fmt As String = "{0,-20} {1,5}  {2, 3}"

    Public Sub New(ByVal name As String)
        If String.IsNullOrEmpty(name) Then Throw New ArgumentNullException("The observer must be assigned a
name.")

        Me.name = name
    End Sub

    Public Overridable Sub Subscribe(ByVal provider As BaggageHandler)
        cancellation = provider.Subscribe(Me)
    End Sub

    Public Overridable Sub Unsubscribe()
        cancellation.Dispose()
        flightInfos.Clear()
    End Sub

    Public Overridable Sub OnCompleted() Implements System.IObserver(Of BaggageInfo).OnCompleted
        flightInfos.Clear()
    End Sub

    ' No implementation needed: Method is not called by the BaggageHandler class.
    Public Overridable Sub OnError(ByVal e As System.Exception) Implements System.IObserver(Of
BaggageInfo).OnError
        ' No implementation.
    End Sub

    ' Update information.
    Public Overridable Sub OnNext(ByVal info As BaggageInfo) Implements System.IObserver(Of
BaggageInfo).OnNext
        Dim updated As Boolean = False

        ' Flight has unloaded its baggage; remove from the monitor.
        If info.Carousel = 0 Then
            Dim flightsToRemove As New List(Of String)
            Dim flightNo As String = String.Format("{0,5}", info.FlightNumber)
            For Each flightInfo In flightInfos
                If flightInfo.Substring(21, 5).Equals(flightNo) Then
                    flightsToRemove.Add(flightInfo)
                    updated = True
                End If
            Next
            For Each flightToRemove In flightsToRemove
                flightInfos.Remove(flightToRemove)
            Next
            flightsToRemove.Clear()
        Else
            ' Add flight if it does not exist in the collection.
            Dim flightInfo As String = String.Format(fmt, info.From, info.FlightNumber, info.Carousel)
            If Not flightInfos.Contains(flightInfo) Then
                flightInfos.Add(flightInfo)
                updated = True
            End If
        End If
        If updated Then
            flightInfos.Sort()
```

```
            flightInfos.Sort()
            Console.WriteLine("Arrivals information from {0}", Me.name)
            For Each flightInfo In flightInfos
                Console.WriteLine(flightInfo)
            Next
            Console.WriteLine()
        End If
    End Sub
End Class
```

`ArrivalsMonitor` 类包括 `Subscribe` 和 `Unsubscribe` 方法。`Subscribe` 方法使类可将由对 Subscribe 的调用返回的 IDisposable 实现保存到私有变量中。`Unsubscribe` 方法使类可以通过调用提供程序的 Dispose 实现来取消订阅通知。`ArrivalsMonitor` 也提供 OnNext、OnError 和 OnCompleted 方法的实现。仅 OnNext 实现包含大量的代码。该方法处理私有的、已排序的泛型 List<T> 对象，该对象维护有关抵港航班的始发机场以及可提取行李的传送带的信息。如果 `BaggageHandler` 类报告新的航班抵达，则 OnNext 方法实现将该航班的相关信息添加到列表中。如果 `BaggageHandler` 类报告已卸载该航班的行李，则 OnNext 方法从列表中移除该航班。每当进行了更改，就会对列表进行排序并向控制台显示。

下面的示例包含对 `BaggageHandler` 类以及对 `ArrivalsMonitor` 类的两个实例进行实例化的应用程序入口点，并使用 `BaggageHandler.BaggageStatus` 方法来添加和删除有关抵港航班的信息。在每种情况下，观察者均接收更新，并且正确显示行李认领信息。

```
using System;
using System.Collections.Generic;

public class Example
{
   public static void Main()
   {
      BaggageHandler provider = new BaggageHandler();
      ArrivalsMonitor observer1 = new ArrivalsMonitor("BaggageClaimMonitor1");
      ArrivalsMonitor observer2 = new ArrivalsMonitor("SecurityExit");

      provider.BaggageStatus(712, "Detroit", 3);
      observer1.Subscribe(provider);
      provider.BaggageStatus(712, "Kalamazoo", 3);
      provider.BaggageStatus(400, "New York-Kennedy", 1);
      provider.BaggageStatus(712, "Detroit", 3);
      observer2.Subscribe(provider);
      provider.BaggageStatus(511, "San Francisco", 2);
      provider.BaggageStatus(712);
      observer2.Unsubscribe();
      provider.BaggageStatus(400);
      provider.LastBaggageClaimed();
   }
}
// The example displays the following output:
//      Arrivals information from BaggageClaimMonitor1
//      Detroit               712    3
//
//      Arrivals information from BaggageClaimMonitor1
//      Detroit               712    3
//      Kalamazoo             712    3
//
//      Arrivals information from BaggageClaimMonitor1
//      Detroit               712    3
//      Kalamazoo             712    3
//      New York-Kennedy      400    1
//
//      Arrivals information from SecurityExit
//      Detroit               712    3
//
//      Arrivals information from SecurityExit
//      Detroit               712    3
//      Kalamazoo             712    3
```

```
//      Kalamazoo               712    3
//
//      Arrivals information from SecurityExit
//      Detroit                 712    3
//      Kalamazoo               712    3
//      New York-Kennedy        400    1
//
//      Arrivals information from BaggageClaimMonitor1
//      Detroit                 712    3
//      Kalamazoo               712    3
//      New York-Kennedy        400    1
//      San Francisco           511    2
//
//      Arrivals information from SecurityExit
//      Detroit                 712    3
//      Kalamazoo               712    3
//      New York-Kennedy        400    1
//      San Francisco           511    2
//
//      Arrivals information from BaggageClaimMonitor1
//      New York-Kennedy        400    1
//      San Francisco           511    2
//
//      Arrivals information from SecurityExit
//      New York-Kennedy        400    1
//      San Francisco           511    2
//
//      Arrivals information from BaggageClaimMonitor1
//      San Francisco           511    2
```

```
Module Example
    Public Sub Main()
        Dim provider As New BaggageHandler()
        Dim observer1 As New ArrivalsMonitor("BaggageClaimMonitor1")
        Dim observer2 As New ArrivalsMonitor("SecurityExit")

        provider.BaggageStatus(712, "Detroit", 3)
        observer1.Subscribe(provider)
        provider.BaggageStatus(712, "Kalamazoo", 3)
        provider.BaggageStatus(400, "New York-Kennedy", 1)
        provider.BaggageStatus(712, "Detroit", 3)
        observer2.Subscribe(provider)
        provider.BaggageStatus(511, "San Francisco", 2)
        provider.BaggageStatus(712)
        observer2.Unsubscribe()
        provider.BaggageStatus(400)
        provider.LastBaggageClaimed()
    End Sub
End Module
' The example displays the following output:
'      Arrivals information from BaggageClaimMonitor1
'      Detroit               712    3
'
'      Arrivals information from BaggageClaimMonitor1
'      Detroit               712    3
'      Kalamazoo             712    3
'
'      Arrivals information from BaggageClaimMonitor1
'      Detroit               712    3
'      Kalamazoo             712    3
'      New York-Kennedy      400    1
'
'      Arrivals information from SecurityExit
'      Detroit               712    3
'
'      Arrivals information from SecurityExit
'      Detroit               712    3
'      Kalamazoo             712    3
'
'      Arrivals information from SecurityExit
'      Detroit               712    3
'      Kalamazoo             712    3
'      New York-Kennedy      400    1
'
'      Arrivals information from BaggageClaimMonitor1
'      Detroit               712    3
'      Kalamazoo             712    3
'      New York-Kennedy      400    1
'      San Francisco         511    2
'
'      Arrivals information from SecurityExit
'      Detroit               712    3
'      Kalamazoo             712    3
'      New York-Kennedy      400    1
'      San Francisco         511    2
'
'      Arrivals information from BaggageClaimMonitor1
'      New York-Kennedy      400    1
'      San Francisco         511    2
'
'      Arrivals information from SecurityExit
'      New York-Kennedy      400    1
'      San Francisco         511    2
'
'      Arrivals information from BaggageClaimMonitor1
'      San Francisco         511    2
```

相关主题

TITLE	tt
监视程序设计模式最佳做法	描述开发实现观察者设计模式的应用程序时要采用的最佳做法。
如何：实现提供程序	为温度监控应用程序提供一个提供程序的分步实现。
如何：实现监视程序	为温度监控应用程序提供一个观察者的分步实现。

观察程序设计模式最佳做法

2021/11/16 •

在 .NET 中，将观察者设计模式作为一组接口实现。System.IObservable<T> 接口表示数据提供程序，也负责提供允许观察者取消订阅通知的 IDisposable 实现。System.IObserver<T> 接口表示观察者。本主题描述使用这些接口实现观察者设计模式时开发人员应遵循的最佳做法。

线程

提供程序通常通过向由一些集合对象表示的订阅者列表添加特定观察者来实现 IObservable<T>.Subscribe 方法，并通过从订阅者列表中删除特定观察者来实现 IDisposable.Dispose 方法。观察者可在任何时候调用这些方法。此外，由于提供程序/观察者协定未指定由谁负责在 IObserver<T>.OnCompleted 回调方法后取消订阅，因此提供程序和观察者都可能尝试从列表中删除相同成员。由于这种可能性，Subscribe 和 Dispose 方法都应该是线程安全的。这通常需要使用并发回收或锁。非线程安全的实现应显式注明它们非线程安全。

任何其他保证均须在提供程序/观察者协定之上指定。实施者应清楚地调出何时施加其他需求，从而避免用户对观察者协定产生混淆。

处理异常

由于数据提供程序和观察者之间的松耦合，观察者设计模式中的异常旨在提供信息。这会影响提供程序和观察者处理观察者设计模式中的异常的方式。

提供者 -- 调用 OnError 方法

OnError 方法提供旨在给观察者提供信息性消息，正如 IObserver<T>.OnNext 方法一样。然而，OnNext 方法旨在为观察者提供当前数据或已更新数据，而 OnError 方法旨在表明该提供程序不能提供有效的数据。

在处理异常和调用 OnError 方法时，提供程序应遵循以下最佳做法并：

- 如果提供程序有任何具体需求，则它必须处理自己的异常。
- 提供程序不应期望或要求观察者以任何特定方式处理异常。
- 提供程序应在处理削弱其提供更新的能力的异常时调用 OnError 方法。可将有关这种异常的信息传递给观察者。在其他情况下，则无需通知观察器有异常。

只要提供程序调用了 OnError 或 IObserver<T>.OnCompleted 方法，就不应有进一步的通知，并且提供程序可以取消订阅其观察者。然而，观察者也可以在任何时候取消订阅他们自己，包括在他们收到 OnError 或IObserver<T>.OnCompleted 通知前后。观察者设计模式并未规定负责取消订阅的提供程序还是观察者；因此，可能这两者都试图取消订阅。通常情况下，当观察者取消订阅时，会从订阅者集合中把他们删除。在单线程应用程序中，IDisposable.Dispose 实现应确保对象引用有效，并在尝试删除对象前确保该对象是订阅者集合的成员。在多线程应用程序中，应使用诸如 System.Collections.Concurrent.BlockingCollection<T> 对象的线程安全集合对象。

观察者 -- 实现 OnError 方法

当观察者收到来自提供程序的错误通知时，应将异常视为信息，且不需采取任何特殊操作。

在回应从提供程序调用的 OnError 方法时，观察者应遵循以下最佳做法：

- 观察者不应从其接口实现引发异常，如 OnNext 或 OnError。但如果观察者确实引发了异常，应预计到这些异常不会得到处理。
- 若要保留调用堆栈，希望引发传递到 Exception 方法的 OnError 对象的观察者应在引发之前包装该异常。

为此，应使用标准异常对象。

其他最佳做法

尝试在 IObservable<T>.Subscribe 方法中取消注册可能会导致空引用。因此，我们建议你避免这种做法。

虽然可将观察者附加到多个提供程序到，建议的模式是将 IObserver<T> 实例附加到唯一的 IObservable<T> 实例。

另请参阅

- 观察程序设计模式
- 如何：实现监视程序
- 如何：实现提供程序

如何：实现提供程序

2021/11/16 •

观察程序设计模式要求区分提供程序(监视数据并发送通知)和一个或多个观察程序(通过提供程序接收通知(回调))。本主题介绍了如何创建提供程序。相关主题如何:实现观察程序介绍了如何创建观察程序。

创建提供程序的具体步骤

1. 定义提供程序负责发送给观察程序的数据。尽管提供程序和发送给观察程序的数据可能是一种类型，但它们通常由不同类型表示。例如，在温度监视应用中，`Temperature` 结构定义提供程序(由下一步中定义的 `TemperatureMonitor` 类表示)监视和观察程序订阅的数据。

```
using System;

public struct Temperature
{
   private decimal temp;
   private DateTime tempDate;

   public Temperature(decimal temperature, DateTime dateAndTime)
   {
      this.temp = temperature;
      this.tempDate = dateAndTime;
   }

   public decimal Degrees
   { get { return this.temp; } }

   public DateTime Date
   { get { return this.tempDate; } }
}
```

```
Public Structure Temperature
    Private temp As Decimal
    Private tempDate As DateTime

    Public Sub New(ByVal temperature As Decimal, ByVal dateAndTime As DateTime)
        Me.temp = temperature
        Me.tempDate = dateAndTime
    End Sub

    Public ReadOnly Property Degrees As Decimal
        Get
            Return Me.temp
        End Get
    End Property

    Public ReadOnly Property [Date] As DateTime
        Get
            Return tempDate
        End Get
    End Property
End Structure
```

2. 定义数据提供程序，即实现 System.IObservable<T> 接口的类型。提供程序的泛型类型参数是提供程序发送给观察程序的类型。下面的示例定义了 `TemperatureMonitor` 类，即使用 `Temperature` 泛型类型参数的构造 System.IObservable<T> 实现。

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
```

```
Imports System.Collections.Generic


Public Class TemperatureMonitor : Implements IObservable(Of Temperature)
```

3. 确定提供程序如何存储对观察程序的引用，以便视情况通知每个观察程序。最常见的情况是，集合对象（如泛型 List<T> 对象）用于实现此目的。下面的示例定义了在 `TemperatureMonitor` 类构造函数中实例化的专用 List<T> 对象。

```
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
   List<IObserver<Temperature>> observers;

   public TemperatureMonitor()
   {
      observers = new List<IObserver<Temperature>>();
   }
```

```
Imports System.Collections.Generic


Public Class TemperatureMonitor : Implements IObservable(Of Temperature)
    Dim observers As List(Of IObserver(Of Temperature))

    Public Sub New()
        observers = New List(Of IObserver(Of Temperature))
    End Sub
```

4. 定义提供程序可以返回给订阅者的 IDisposable 实现，以便订阅者能够随时停止接收通知。下面的示例定义了嵌套类 `Unsubscriber`，在类实例化时向其中传递对订阅者集合和订阅者的引用。使用此代码，订阅者可以调用对象的 IDisposable.Dispose 实现，将自身从订阅者集合中删除。

```
private class Unsubscriber : IDisposable
{
   private List<IObserver<Temperature>> _observers;
   private IObserver<Temperature> _observer;

   public Unsubscriber(List<IObserver<Temperature>> observers, IObserver<Temperature> observer)
   {
      this._observers = observers;
      this._observer = observer;
   }

   public void Dispose()
   {
      if (! (_observer == null)) _observers.Remove(_observer);
   }
}
```

```
Private Class Unsubscriber : Implements IDisposable
    Private _observers As List(Of IObserver(Of Temperature))
    Private _observer As IObserver(Of Temperature)

    Public Sub New(ByVal observers As List(Of IObserver(Of Temperature)), ByVal observer As
IObserver(Of Temperature))
        Me._observers = observers
        Me._observer = observer
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        If _observer IsNot Nothing Then _observers.Remove(_observer)
    End Sub
End Class
```

5. 实现 IObservable<T>.Subscribe 方法。向此方法传递对 System.IObserver<T> 接口的引用，此方法应存储在第 3 步中为实现相应目的而设计的对象中。然后，此方法应返回在第 4 步中开发的 IDisposable 实现。下面的示例展示了 `TemperatureMonitor` 类中 Subscribe 方法的实现。

```
public IDisposable Subscribe(IObserver<Temperature> observer)
{
   if (! observers.Contains(observer))
      observers.Add(observer);

   return new Unsubscriber(observers, observer);
}
```

```
Public Function Subscribe(ByVal observer As System.IObserver(Of Temperature)) As System.IDisposable
Implements System.IObservable(Of Temperature).Subscribe
    If Not observers.Contains(observer) Then
        observers.Add(observer)
    End If
    Return New Unsubscriber(observers, observer)
End Function
```

6. 通过调用 IObserver<T>.OnNext、IObserver<T>.OnError 和 IObserver<T>.OnCompleted 实现，视情况通知观察程序。在某些情况下，如果出错，提供程序可能不会调用 OnError 方法。例如，下面的 `GetTemperature` 方法模拟监视器，每 5 秒读取一次温度数据，如果温度自从上一次读数以来出现至少 0.1 度的变化，就会通知观察程序。如果设备不报告温度(即如果值为 null)，提供程序便会通知观察程序传输已完成。请注意，除了调用每个观察程序的 OnCompleted 方法外，`GetTemperature` 方法还可以清除 List<T> 集合。在这种情况下，提供程序不调用观察程序的 OnError 方法。

```
public void GetTemperature()
{
   // Create an array of sample data to mimic a temperature device.
   Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m, 15.2m, 15.25m, 15.2m,
                                15.4m, 15.45m, null };
   // Store the previous temperature, so notification is only sent after at least .1 change.
   Nullable<Decimal> previous = null;
   bool start = true;

   foreach (var temp in temps) {
      System.Threading.Thread.Sleep(2500);
      if (temp.HasValue) {
         if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m )) {
            Temperature tempData = new Temperature(temp.Value, DateTime.Now);
            foreach (var observer in observers)
               observer.OnNext(tempData);
            previous = temp;
            if (start) start = false;
         }
      }
      else {
         foreach (var observer in observers.ToArray())
            if (observer != null) observer.OnCompleted();

         observers.Clear();
         break;
      }
   }
}
```

```
Public Sub GetTemperature()
    ' Create an array of sample data to mimic a temperature device.
    Dim temps() As Nullable(Of Decimal) = {14.6D, 14.65D, 14.7D, 14.9D, 14.9D, 15.2D, 15.25D, 15.2D,
                                           15.4D, 15.45D, Nothing}
    ' Store the previous temperature, so notification is only sent after at least .1 change.
    Dim previous As Nullable(Of Decimal)
    Dim start As Boolean = True

    For Each temp In temps
        System.Threading.Thread.Sleep(2500)

        If temp.HasValue Then
            If start OrElse Math.Abs(temp.Value - previous.Value) >= 0.1 Then
                Dim tempData As New Temperature(temp.Value, Date.Now)
                For Each observer In observers
                    observer.OnNext(tempData)
                Next
                previous = temp
                If start Then start = False
            End If
        Else
            For Each observer In observers.ToArray()
                If observer IsNot Nothing Then observer.OnCompleted()
            Next
            observers.Clear()
            Exit For
        End If
    Next
End Sub
```

示例

下面的示例包含完整的源代码，用于定义温度监视应用的 IObservable<T> 实现。它包括 `Temperature` 结构(即

发送给观察程序的数据）和 `TemperatureMonitor` 类（即 IObservable<T> 实现）。

```
using System.Threading;
using System;
using System.Collections.Generic;

public class TemperatureMonitor : IObservable<Temperature>
{
   List<IObserver<Temperature>> observers;

   public TemperatureMonitor()
   {
      observers = new List<IObserver<Temperature>>();
   }

   private class Unsubscriber : IDisposable
   {
      private List<IObserver<Temperature>> _observers;
      private IObserver<Temperature> _observer;

      public Unsubscriber(List<IObserver<Temperature>> observers, IObserver<Temperature> observer)
      {
         this._observers = observers;
         this._observer = observer;
      }

      public void Dispose()
      {
         if (! (_observer == null)) _observers.Remove(_observer);
      }
   }

   public IDisposable Subscribe(IObserver<Temperature> observer)
   {
      if (! observers.Contains(observer))
         observers.Add(observer);

      return new Unsubscriber(observers, observer);
   }

   public void GetTemperature()
   {
      // Create an array of sample data to mimic a temperature device.
      Nullable<Decimal>[] temps = {14.6m, 14.65m, 14.7m, 14.9m, 14.9m, 15.2m, 15.25m, 15.2m,
                                   15.4m, 15.45m, null };
      // Store the previous temperature, so notification is only sent after at least .1 change.
      Nullable<Decimal> previous = null;
      bool start = true;

      foreach (var temp in temps) {
         System.Threading.Thread.Sleep(2500);
         if (temp.HasValue) {
            if (start || (Math.Abs(temp.Value - previous.Value) >= 0.1m )) {
               Temperature tempData = new Temperature(temp.Value, DateTime.Now);
               foreach (var observer in observers)
                  observer.OnNext(tempData);
               previous = temp;
               if (start) start = false;
            }
         }
         else {
            foreach (var observer in observers.ToArray())
               if (observer != null) observer.OnCompleted();

            observers.Clear();
            break;
         }
      }
```

```
    }
}
```

```
Imports System.Threading
Imports System.Collections.Generic


Public Class TemperatureMonitor : Implements IObservable(Of Temperature)
    Dim observers As List(Of IObserver(Of Temperature))

    Public Sub New()
        observers = New List(Of IObserver(Of Temperature))
    End Sub

    Private Class Unsubscriber : Implements IDisposable
        Private _observers As List(Of IObserver(Of Temperature))
        Private _observer As IObserver(Of Temperature)

        Public Sub New(ByVal observers As List(Of IObserver(Of Temperature)), ByVal observer As IObserver(Of
Temperature))
            Me._observers = observers
            Me._observer = observer
        End Sub

        Public Sub Dispose() Implements IDisposable.Dispose
            If _observer IsNot Nothing Then _observers.Remove(_observer)
        End Sub
    End Class

    Public Function Subscribe(ByVal observer As System.IObserver(Of Temperature)) As System.IDisposable
Implements System.IObservable(Of Temperature).Subscribe
        If Not observers.Contains(observer) Then
            observers.Add(observer)
        End If
        Return New Unsubscriber(observers, observer)
    End Function

    Public Sub GetTemperature()
        ' Create an array of sample data to mimic a temperature device.
        Dim temps() As Nullable(Of Decimal) = {14.6D, 14.65D, 14.7D, 14.9D, 14.9D, 15.2D, 15.25D, 15.2D,
                                               15.4D, 15.45D, Nothing}
        ' Store the previous temperature, so notification is only sent after at least .1 change.
        Dim previous As Nullable(Of Decimal)
        Dim start As Boolean = True

        For Each temp In temps
            System.Threading.Thread.Sleep(2500)

            If temp.HasValue Then
                If start OrElse Math.Abs(temp.Value - previous.Value) >= 0.1 Then
                    Dim tempData As New Temperature(temp.Value, Date.Now)
                    For Each observer In observers
                        observer.OnNext(tempData)
                    Next
                    previous = temp
                    If start Then start = False
                End If
            Else
                For Each observer In observers.ToArray()
                    If observer IsNot Nothing Then observer.OnCompleted()
                Next
                observers.Clear()
                Exit For
            End If
        Next
    End Sub
End Class
```

另请参阅

- IObservable<T>
- 观察程序设计模式
- 如何：实现监视程序
- 监视程序设计模式最佳做法

如何：实现监视程序

2021/11/16 •

观察程序设计模式要求区分观察程序(注册获取通知)和提供程序(监视数据并将通知发送到一个或多个观察程序)。本主题介绍了如何创建观察程序。相关主题如何：实现提供程序介绍了如何创建提供程序。

创建观察程序的具体步骤

1. 定义观察程序，即实现 System.IObserver<T> 接口的类型。例如，下面的代码定义了 `TemperatureReporter` 类型，即使用 `Temperature` 泛型类型参数的构造 System.IObserver<T> 实现。

```
public class TemperatureReporter : IObserver<Temperature>
```

```
Public Class TemperatureReporter : Implements IObserver(Of Temperature)
```

2. 如果观察程序可以在提供程序调用 IObserver<T>.OnCompleted 实现前停止接收通知，请定义专用变量，用于保留提供程序 IObservable<T>.Subscribe 方法返回的 IDisposable 实现。还应定义订阅方法，用于调用提供程序的 Subscribe 方法，并存储返回的 IDisposable 对象。例如，下面的代码定义 `unsubscriber` 专用变量，并定义 `Subscribe` 方法，用于调用提供程序的 Subscribe 方法，并将返回的对象分配给 `unsubscriber` 变量。

```
public class TemperatureReporter : IObserver<Temperature>
{
   private IDisposable unsubscriber;
   private bool first = true;
   private Temperature last;

   public virtual void Subscribe(IObservable<Temperature> provider)
   {
      unsubscriber = provider.Subscribe(this);
   }
```

```
Public Class TemperatureReporter : Implements IObserver(Of Temperature)

    Private unsubscriber As IDisposable
    Private first As Boolean = True
    Private last As Temperature

    Public Overridable Sub Subscribe(ByVal provider As IObservable(Of Temperature))
        unsubscriber = provider.Subscribe(Me)
    End Sub
```

3. 定义一个方法，以便观察程序可以在提供程序调用 IObserver<T>.OnCompleted 实现前停止接收通知(如果需要此功能的话)。下面的示例定义了 `Unsubscribe` 方法。

```
public virtual void Unsubscribe()
{
   unsubscriber.Dispose();
}
```

```
Public Overridable Sub Unsubscribe()
    unsubscriber.Dispose()
End Sub
```

4. 实现 IObserver<T> 接口定义的三个方法：IObserver<T>.OnNext、IObserver<T>.OnError 和 IObserver<T>.OnCompleted。根据提供程序和应用需求，OnError 和 OnCompleted 方法可以是存根实现。请注意，OnError 方法不得将传入的 Exception 对象处理为异常，OnCompleted 方法可以随意调用提供程序的 IDisposable.Dispose 实现。下面的示例展示了 `TemperatureReporter` 类的 IObserver<T> 实现。

```
public virtual void OnCompleted()
{
   Console.WriteLine("Additional temperature data will not be transmitted.");
}

public virtual void OnError(Exception error)
{
   // Do nothing.
}

public virtual void OnNext(Temperature value)
{
   Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees, value.Date);
   if (first)
   {
      last = value;
      first = false;
   }
   else
   {
      Console.WriteLine("   Change: {0}° in {1:g}", value.Degrees - last.Degrees,
                                                    value.Date.ToUniversalTime() -
last.Date.ToUniversalTime());
   }
}
```

```
Public Overridable Sub OnCompleted() Implements System.IObserver(Of Temperature).OnCompleted
    Console.WriteLine("Additional temperature data will not be transmitted.")
End Sub

Public Overridable Sub OnError(ByVal [error] As System.Exception) Implements System.IObserver(Of
Temperature).OnError
    ' Do nothing.
End Sub

Public Overridable Sub OnNext(ByVal value As Temperature) Implements System.IObserver(Of
Temperature).OnNext
    Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees, value.Date)
    If first Then
        last = value
        first = False
    Else
        Console.WriteLine("   Change: {0}° in {1:g}", value.Degrees - last.Degrees,
                                                      value.Date.ToUniversalTime -
last.Date.ToUniversalTime)
    End If
End Sub
```

示例

下面的示例包含 `TemperatureReporter` 类的完整源代码，提供了温度监视应用的 IObserver<T> 实现。

```
public class TemperatureReporter : IObserver<Temperature>
{
   private IDisposable unsubscriber;
   private bool first = true;
   private Temperature last;

   public virtual void Subscribe(IObservable<Temperature> provider)
   {
      unsubscriber = provider.Subscribe(this);
   }

   public virtual void Unsubscribe()
   {
      unsubscriber.Dispose();
   }

   public virtual void OnCompleted()
   {
      Console.WriteLine("Additional temperature data will not be transmitted.");
   }

   public virtual void OnError(Exception error)
   {
      // Do nothing.
   }

   public virtual void OnNext(Temperature value)
   {
      Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees, value.Date);
      if (first)
      {
         last = value;
         first = false;
      }
      else
      {
         Console.WriteLine("   Change: {0}° in {1:g}", value.Degrees - last.Degrees,
                                                       value.Date.ToUniversalTime() -
last.Date.ToUniversalTime());
      }
   }
}
```

```
Public Class TemperatureReporter : Implements IObserver(Of Temperature)

    Private unsubscriber As IDisposable
    Private first As Boolean = True
    Private last As Temperature

    Public Overridable Sub Subscribe(ByVal provider As IObservable(Of Temperature))
        unsubscriber = provider.Subscribe(Me)
    End Sub

    Public Overridable Sub Unsubscribe()
        unsubscriber.Dispose()
    End Sub

    Public Overridable Sub OnCompleted() Implements System.IObserver(Of Temperature).OnCompleted
        Console.WriteLine("Additional temperature data will not be transmitted.")
    End Sub

    Public Overridable Sub OnError(ByVal [error] As System.Exception) Implements System.IObserver(Of
Temperature).OnError
        ' Do nothing.
    End Sub

    Public Overridable Sub OnNext(ByVal value As Temperature) Implements System.IObserver(Of
Temperature).OnNext
        Console.WriteLine("The temperature is {0}°C at {1:g}", value.Degrees, value.Date)
        If first Then
            last = value
            first = False
        Else
            Console.WriteLine("   Change: {0}° in {1:g}", value.Degrees - last.Degrees,
                                                          value.Date.ToUniversalTime -
last.Date.ToUniversalTime)
        End If
    End Sub
End Class
```

请参阅

- IObserver<T>
- 观察程序设计模式
- 如何:实现提供程序
- 监视程序设计模式最佳做法

在 .NET 中处理和引发异常

2021/11/16 •

应用程序必须能够以一致的方式处理执行期间发生的错误。.NET 提供一种以统一方式向应用程序报错的模型：.NET 操作通过引发异常来指示故障。

异常

异常是执行程序遇到的所有错误条件或意外行为。异常可能由你的代码或调用的代码（如共享库）中的错误、不可用的操作系统资源、运行时遇到的意外情况（如无法验证的代码）等引发。应用程序可从这些情况中的一些中恢复，但无法从其他情况中恢复。尽管可以从大多数应用程序异常中恢复，但不能从大多数运行时异常中恢复。

在 .NET中，异常是从 System.Exception 类继承的对象。异常引发自发生问题的代码区域。异常在堆栈中向上传递，直到应用程序对其进行处理或者程序终止。

异常与传统的错误处理方法

传统上，语言的错误处理模型依赖语言检测错误和针对错误查找其处理程序的独特方式，或者依赖操作系统提供的错误处理机制。.NET 实现异常处理的方式有以下优点：

- 引发和处理异常的方式与 .NET 编程语言的相同。
- 处理异常不需要任何特定的语言语法，但允许每种语言定义自己的语法。
- 可跨进程，甚至跨计算机边界引发异常。
- 可向应用程序添加异常处理代码以提高程序的可靠性。

异常相较于其他错误通知方法（如返回代码）具有多种优势。故障不会被忽略掉，因为如果引发了异常且未得到解决，运行时会终止应用程序。因为代码未能检查出是否存在故障返回代码，所以无效值不会继续在系统中传播。

常见异常

下表列出了一些常见的异常，以及会引发这些异常的原因的示例。

tttt	tt	tt
Exception	所有异常的基类。	无（使用此异常的派生类）。
IndexOutOfRangeException	仅当错误地对数组进行索引时，才由运行时引发。	在数组的有效范围外对数组进行索引：`arr[arr.Length+1]`
NullReferenceException	仅当引用 null 对象时，才由运行时引发。	`object o = null;` `o.ToString();`
InvalidOperationException	当处于无效状态时，由方法引发。	从基础集合删除项后调用 `Enumerator.MoveNext()`。
ArgumentException	所有自变量异常的基类。	无（使用此异常的派生类）。

[illegible]	[illegible]	[illegible]
ArgumentNullException	由不允许参数为 null 的方法引发。	`String s = null;` `"Calculate".IndexOf(s);`
ArgumentOutOfRangeException	由验证自变量是否位于给定范围内的方法引发。	`String s = "string";` `s.Substring(s.Length+1);`

请参阅

- 异常类和属性
- 如何：使用 Try-Catch 块捕获异常
- 如何：在 Catch 块中使用特定异常
- 如何：显式抛出异常
- 如何：创建用户定义异常
- 使用用户筛选的异常处理程序
- 如何：使用 Finally 块
- 处理 COM 互操作异常
- 与异常有关的最佳做法
- 每个开发人员都需要了解的有关运行时异常方面的内容

异常类和属性

2021/11/16 •

Exception 类是异常从中继承的基类。例如，InvalidCastException 类层次结构如下所示：

Object
  Exception
    SystemException
      InvalidCastException

Exception 类具有以下属性，有助于更轻松地理解异常。

☐☐☐☐	☐☐
Data	IDictionary 包含键/值对中的任意数据。
HelpLink	可容纳指向帮助文件的 URL（或 URN），帮助文件中提供了大量信息说明了异常的原因。
InnerException	在处理异常时此属性可用于创建和保留一系列异常。可将其用于创建新异常，其中包含之前捕获到的异常。可通过 InnerException 属性中的第二个异常捕获原始异常，允许处理第二个异常的代码检查其他信息。例如，假设有一个方法可接收格式不正确的参数。该代码尝试读取参数，但会引发异常。该方法会捕获异常并引发 FormatException。若要提高调用方确定引发异常的原因的能力，有时最好用一个方法捕获帮助器例程引发的异常，然后引发一个更能说明已发生错误的异常。可创建一个新的且更有意义的异常，其中可将内部异常引用设为原始异常。然后可向调用方引发此更有意义的异常。请注意，使用此功能，可创建一系列链接的异常，以最先引发的异常结尾。
Message	提供有关异常原因的详细信息。
Source	获取或设置导致错误的应用程序或对象的名称。
StackTrace	包含可用于确定错误位置的堆栈跟踪。如果有可用的调试信息，则堆栈跟踪包含源文件名和程序行号。

继承自 Exception 的大多数类无法实现其他成员或提供其他功能；它们只从 Exception 进行继承。因此，可在异常类层次结构、异常名称和异常所含的信息中找到异常的重要信息。

建议仅抛出和捕获派生自 Exception 的对象，但可以将派生自 Object 类的任何对象作为异常抛出。请注意，并非所有语言都支持引发和捕获不是从 Exception 派生的对象。

另请参阅

- 异常

如何使用 try/catch 块捕获异常

2021/11/16 •

将可能引发异常的任何代码语句放置在 `try` 块中，将用于处理异常的语句放置在 `try` 块下的一个或多个 `catch` 块中。每个 `catch` 块包括异常类型，并且可以包含处理该异常类型所需的其他语句。

在以下示例中，StreamReader 将打开一个名为 data.txt 的文件，并从文件中检索行。因为代码可能会引发任何三个异常，因此将其放置于 `try` 块中。三个 `catch` 块捕获异常并通过将结果向控制台显示来处理它们。

```
using System;
using System.IO;

public class ProcessFile
{
    public static void Main()
    {
        try
        {
            using (StreamReader sr = File.OpenText("data.txt"))
            {
                Console.WriteLine($"The first line of this file is {sr.ReadLine()}");
            }
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine($"The file was not found: '{e}'");
        }
        catch (DirectoryNotFoundException e)
        {
            Console.WriteLine($"The directory was not found: '{e}'");
        }
        catch (IOException e)
        {
            Console.WriteLine($"The file could not be opened: '{e}'");
        }
    }
}
```

```
Imports System.IO

Public Class ProcessFile
    Public Shared Sub Main()
        Try
            Using sr As StreamReader = File.OpenText("data.txt")
                Console.WriteLine($"The first line of this file is {sr.ReadLine()}")
            End Using
        Catch e As FileNotFoundException
            Console.WriteLine($"The file was not found: '{e}'")
        Catch e As DirectoryNotFoundException
            Console.WriteLine($"The directory was not found: '{e}'")
        Catch e As IOException
            Console.WriteLine($"The file could not be opened: '{e}'")
        End Try
    End Sub
End Class
```

公共语言运行时 (CLR) 会捕获未由 `catch` 块处理的异常。如果异常由 CLR 捕获，则可能出现以下结果之一，具体取决于 CLR 配置：

- 出现“调试”对话框。
- 该程序停止执行，出现含有异常信息的对话框。
- 错误输出到标准错误输出流。

> **NOTE**
>
> 大多数代码可能会引发异常，并且一些异常（如 OutOfMemoryException）可能随时由 CLR 本身引发。虽然应用程序无需处理这些异常，但在编写供他人使用的库时，应注意到这种可能性。有关何时在 `try` 块中设置代码的建议，请参阅异常的最佳做法。

请参阅

- 异常
- 处理 .NET 中的 I/O 错误

如何在 catch 块中使用特定异常

2021/11/16 •

一般来说，编程最佳做法是，捕获特定类型的异常，而不是使用基本 `catch` 语句。

异常发生时，会在堆栈中向上传递，每个 catch 块都有机会处理异常。Catch 语句的顺序非常重要。在一般异常 catch 块或编译器发出错误前，将 catch 块指向特定异常。通过匹配异常类型与 catch 块中指定的异常名称，确定合适的 catch 块。如果没有特定的 catch 块，则将由一般 catch 块捕获异常（如果异常存在）。

下方代码示例使用 `try` / `catch` 块来捕获 InvalidCastException。该示例创建了一个名为 `Employee` 的类，其具有单一属性，雇员级别为 (`Emlevel`)。`PromoteEmployee` 方法接收对象并递增雇员级别。DateTime 实例传递给 `PromoteEmployee` 方法时，InvalidCastException 发生。

```
using namespace System;

public ref class Employee
{
public:
    Employee()
    {
        emlevel = 0;
    }

    //Create employee level property.
    property int Emlevel
    {
        int get()
        {
            return emlevel;
        }
        void set(int value)
        {
            emlevel = value;
        }
    }

private:
    int emlevel;
};

public ref class Ex13
{
public:
    static void PromoteEmployee(Object^ emp)
    {
        //Cast object to Employee.
        Employee^ e = (Employee^) emp;
        // Increment employee level.
        e->Emlevel++;
    }

    static void Main()
    {
        try
        {
            Object^ o = gcnew Employee();
            DateTime^ newyears = gcnew DateTime(2001, 1, 1);
            //Promote the new employee.
            PromoteEmployee(o);
            //Promote DateTime; results in InvalidCastException as newyears is not an employee instance.
            PromoteEmployee(newyears);
        }
        catch (InvalidCastException^ e)
        {
            Console::WriteLine("Error passing data to PromoteEmployee method. " + e->Message);
        }
    }
};

int main()
{
    Ex13::Main();
}
```

```
using System;

public class Employee
{
    //Create employee level property.
    public int Emlevel
    {
        get
        {
            return(emlevel);
        }
        set
        {
            emlevel = value;
        }
    }

    private int emlevel = 0;
}

public class Ex13
{
    public static void PromoteEmployee(Object emp)
    {
        // Cast object to Employee.
        var e = (Employee) emp;
        // Increment employee level.
        e.Emlevel = e.Emlevel + 1;
    }

    public static void Main()
    {
        try
        {
            Object o = new Employee();
            DateTime newYears = new DateTime(2001, 1, 1);
            // Promote the new employee.
            PromoteEmployee(o);
            // Promote DateTime; results in InvalidCastException as newYears is not an employee instance.
            PromoteEmployee(newYears);
        }
        catch (InvalidCastException e)
        {
            Console.WriteLine("Error passing data to PromoteEmployee method. " + e.Message);
        }
    }
}
```

```
Public Class Employee
    'Create employee level property.
    Public Property Emlevel As Integer
        Get
            Return emlevelValue
        End Get
        Set
            emlevelValue = Value
        End Set
    End Property

    Private emlevelValue As Integer = 0
End Class

Public Class Ex13
    Public Shared Sub PromoteEmployee(emp As Object)
        ' Cast object to Employee.
        Dim e As Employee = CType(emp, Employee)
        ' Increment employee level.
        e.Emlevel = e.Emlevel + 1
    End Sub

    Public Shared Sub Main()
        Try
            Dim o As Object = New Employee()
            Dim newYears As New DateTime(2001, 1, 1)
            ' Promote the new employee.
            PromoteEmployee(o)
            ' Promote DateTime; results in InvalidCastException as newYears is not an employee instance.
            PromoteEmployee(newYears)
        Catch e As InvalidCastException
            Console.WriteLine("Error passing data to PromoteEmployee method. " + e.Message)
        End Try
    End Sub
End Class
```

另请参阅

- 异常

如何显式引发异常

2021/11/16 •

可使用 C# `throw` 或 Visual Basic `Throw` 语句显式引发异常。可使用 `throw` 语句再次引发捕获的异常。向重新引发的异常添加信息以在调试时提供详细信息，这是很好的编码做法。

下方代码示例使用 `try` / `catch` 块来捕获可能的 FileNotFoundException。以下 `try` 块为可捕获 FileNotFoundException 并在未找到数据文件时将消息写入控制台的 `catch` 块。下一语句为 `throw` 语句，可引发新的 FileNotFoundException 并向异常添加文本信息。

```
using System;
using System.IO;

public class ProcessFile
{
    public static void Main()
    {
        FileStream fs;
        try
        {
            // Opens a text tile.
            fs = new FileStream(@"C:\temp\data.txt", FileMode.Open);
            var sr = new StreamReader(fs);

            // A value is read from the file and output to the console.
            string line = sr.ReadLine();
            Console.WriteLine(line);
        }
        catch(FileNotFoundException e)
        {
            Console.WriteLine($"[Data File Missing] {e}");
            throw new FileNotFoundException(@"[data.txt not in c:\temp directory]", e);
        }
        finally
        {
            if (fs != null)
                fs.Close();
        }
    }
}
```

```
Option Strict On

Imports System.IO

Public Class ProcessFile

    Public Shared Sub Main()
        Dim fs As FileStream
        Try
            ' Opens a text file.
            fs = New FileStream("c:\temp\data.txt", FileMode.Open)
            Dim sr As New StreamReader(fs)

            ' A value is read from the file and output to the console.
            Dim line As String = sr.ReadLine()
            Console.WriteLine(line)
        Catch e As FileNotFoundException
            Console.WriteLine($"[Data File Missing] {e}")
            Throw New FileNotFoundException("[data.txt not in c:\temp directory]", e)
        Finally
            If fs IsNot Nothing Then fs.Close()
        End Try
    End Sub
End Class
```

请参阅

- 异常

如何创建用户定义的异常

2021/11/16 •

.NET 可提供由基类 Exception 最终派生的异常类层次结构。然而，如果预定义的异常都不符合需求，可通过从 Exception 类派生来创建自己的异常类。

创建自己的异常时，用户定义的异常类的名称需要以“Exception”一词结尾，并实现三个常见的构造函数，如以下示例所示。该示例定义名为 `EmployeeListNotFoundException` 的新异常类。该类从 Exception 派生，且包含三个构造函数。

```
using namespace System;

public ref class EmployeeListNotFoundException : Exception
{
public:
    EmployeeListNotFoundException()
    {
    }

    EmployeeListNotFoundException(String^ message)
        : Exception(message)
    {
    }

    EmployeeListNotFoundException(String^ message, Exception^ inner)
        : Exception(message, inner)
    {
    }
};
```

```
using System;

public class EmployeeListNotFoundException : Exception
{
    public EmployeeListNotFoundException()
    {
    }

    public EmployeeListNotFoundException(string message)
        : base(message)
    {
    }

    public EmployeeListNotFoundException(string message, Exception inner)
        : base(message, inner)
    {
    }
}
```

```
Public Class EmployeeListNotFoundException
    Inherits Exception

    Public Sub New()
    End Sub

    Public Sub New(message As String)
        MyBase.New(message)
    End Sub

    Public Sub New(message As String, inner As Exception)
        MyBase.New(message, inner)
    End Sub
End Class
```

NOTE

使用远程处理时，必须确保所有用户定义的异常的元数据在服务器（被调用方）可用，在客户端（代理对象或调用方）也可用。有关详细信息，请参阅异常的最佳做法。

请参阅

- 异常

如何使用本地化的异常消息创建用户定义的异常

2021/11/16 •

在本文中，你将了解如何通过使用附属程序集的本地化异常消息创建从 Exception 基类继承的用户定义异常。

创建自定义异常

.NET 包含许多你可以使用的不同异常。但是，在某些情况下，如果它们都无法满足你的需要，则可以创建自己的自定义异常。

假设要创建一个 `StudentNotFoundException`，其中包含 `StudentName` 属性。若要创建自定义异常，请执行以下步骤：

1. 创建一个从 Exception 继承的可序列化类。类名称应以"Exception"结尾：

```
[Serializable]
public class StudentNotFoundException : Exception { }
```

```
<Serializable>
Public Class StudentNotFoundException
    Inherits Exception
End Class
```

2. 添加默认构造函数：

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }
}
```

```
<Serializable>
Public Class StudentNotFoundException
    Inherits Exception

    Public Sub New()
    End Sub

    Public Sub New(message As String)
        MyBase.New(message)
    End Sub

    Public Sub New(message As String, inner As Exception)
        MyBase.New(message, inner)
    End Sub
End Class
```

3. 定义任何其他属性和构造函数：

```
[Serializable]
public class StudentNotFoundException : Exception
{
    public string StudentName { get; }

    public StudentNotFoundException() { }

    public StudentNotFoundException(string message)
        : base(message) { }

    public StudentNotFoundException(string message, Exception inner)
        : base(message, inner) { }

    public StudentNotFoundException(string message, string studentName)
        : this(message)
    {
        StudentName = studentName;
    }
}
```

```
<Serializable>
Public Class StudentNotFoundException
    Inherits Exception

    Public ReadOnly Property StudentName As String

    Public Sub New()
    End Sub

    Public Sub New(message As String)
        MyBase.New(message)
    End Sub

    Public Sub New(message As String, inner As Exception)
        MyBase.New(message, inner)
    End Sub

    Public Sub New(message As String, studentName As String)
        Me.New(message)
        StudentName = studentName
    End Sub
End Class
```

创建本地化异常消息

你已创建一个自定义异常，可以使用如下所示的代码在任何位置将其抛出：

```
throw new StudentNotFoundException("The student cannot be found.", "John");
```

```
Throw New StudentNotFoundException("The student cannot be found.", "John")
```

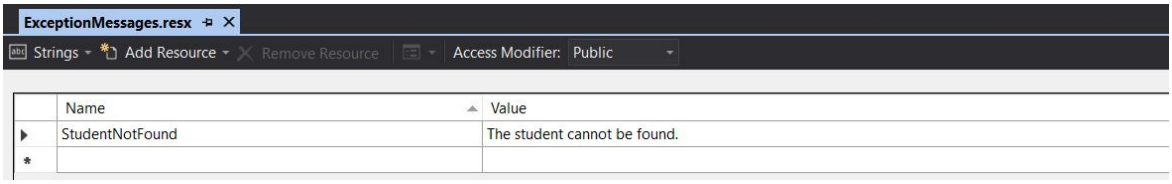
上一行的问题是，`"The student cannot be found."` 只是一个常量字符串。在本地化应用程序中，你需要根据用户区域性使用不同的消息。附属程序集是执行此操作的好方法。附属程序集是一个 .dll，其中包含特定语言的资源。当你在运行时请求特定资源时，CLR 将根据用户区域性查找该资源。如果找不到该区域性对应的附属程序集，则使用默认区域性的资源。

创建本地化异常消息：

1. 创建一个名为“Resources”的新文件夹来保存资源文件。

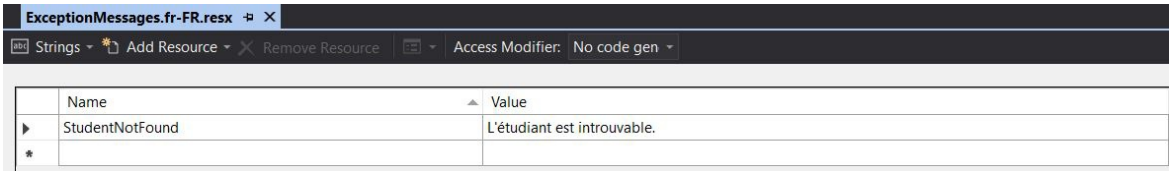
2. 向其中添加新的资源文件。若要在 Visual Studio 中执行此操作，请在“解决方案资源管理器”中右键单击该文件夹，然后选择“添加” > “新项” > “资源文件”。将该文件命名为“ExceptionMessages.resx”。这是默认的资源文件。

3. 为异常消息添加名称/值对，如下图所示：



4. 为法语添加新的资源文件。将其命名为“ExceptionMessages.fr-FR.resx”。

5. 再次为异常消息添加名称/值对，但使用法语值：



6. 生成项目后，生成的输出文件夹应包含 fr-FR 文件夹，其中具有 .dll 文件，它是附属程序集。

7. 使用如下所示代码抛出异常：

```
var resourceManager = new ResourceManager("FULLY_QUALIFIED_NAME_OF_RESOURCE_FILE",
Assembly.GetExecutingAssembly());
throw new StudentNotFoundException(resourceManager.GetString("StudentNotFound"), "John");
```

```
Dim resourceManager As New ResourceManager("FULLY_QUALIFIED_NAME_OF_RESOURCE_FILE",
Assembly.GetExecutingAssembly())
Throw New StudentNotFoundException(resourceManager.GetString("StudentNotFound"), "John")
```

> **NOTE**
> 如果项目名称为 `TestProject`，并且资源文件 ExceptionMessages.resx 位于项目的 Resources 文件夹中，则资源文件的完全限定名称为 `TestProject.Resources.ExceptionMessages`。

请参阅

- 如何创建用户定义的异常
- 创建桌面应用程序的附属程序集
- base（C# 参考）
- this（C# 参考）

如何使用 finally 块

2021/11/16 •

发生异常时，将停止执行且会向相应异常处理程序赋予控制权。这意味着会绕过预计要执行的代码行。即使引发了异常，也需要完成某些资源清理，例如关闭文件。若要执行此操作，可以使用 `finally` 块。无论是否引发异常，始终会执行 `finally` 块。

下方代码示例使用 `try` / `catch` 块来捕获 ArgumentOutOfRangeException。 `Main` 方法会创建两个数组，并尝试将一个数组复制到另一个。该操作将生成 ArgumentOutOfRangeException 并且会向控制台写入错误。 `finally` 块执行时不考虑复制操作的结果。

```
using namespace System;

ref class ArgumentOutOfRangeExample
{
public:
    static void Main()
    {
        array<int>^ array1 = {0, 0};
        array<int>^ array2 = {0, 0};

        try
        {
            Array::Copy(array1, array2, -1);
        }
        catch (ArgumentOutOfRangeException^ e)
        {
            Console::WriteLine("Error: {0}", e);
            throw;
        }
        finally
        {
            Console::WriteLine("This statement is always executed.");
        }
    }
};

int main()
{
    ArgumentOutOfRangeExample::Main();
}
```

```
using System;

class ArgumentOutOfRangeExample
{
    public static void Main()
    {
        int[] array1 = {0, 0};
        int[] array2 = {0, 0};

        try
        {
            Array.Copy(array1, array2, -1);
        }
        catch (ArgumentOutOfRangeException e)
        {
            Console.WriteLine("Error: {0}", e);
            throw;
        }
        finally
        {
            Console.WriteLine("This statement is always executed.");
        }
    }
}
```

```
Class ArgumentOutOfRangeExample
    Public Shared Sub Main()
        Dim array1() As Integer = {0, 0}
        Dim array2() As Integer = {0, 0}

        Try
            Array.Copy(array1, array2, -1)
        Catch e As ArgumentOutOfRangeException
            Console.WriteLine("Error: {0}", e)
            Throw
        Finally
            Console.WriteLine("This statement is always executed.")
        End Try
    End Sub
End Class
```

另请参阅

- 异常

使用用户筛选的异常处理程序

2021/11/16 •

目前，Visual Basic 支持用户筛选的异常。用户筛选的异常处理程序基于定义的异常需求来捕获和处理异常。这些处理程序使用含有关键字 When 的 Catch 语句。

特定异常对象对应于多个错误时，此方法非常有用。在这种情况下，该对象通常包含与错误关联的特定错误代码的属性。可以在表达式中使用此错误代码属性，以便只选择要在 Catch 子句中处理的特定错误。

下面的 Visual Basic 示例阐释 Catch/When 语句。

```
Try
    'Try statements.
    Catch When Err = VBErr_ClassLoadException
    'Catch statements.
End Try
```

用户筛选的子句的表达式不受任何限制。如果在执行用户筛选的表达式期间发生异常，则将放弃该异常，并视筛选表达式的值为 false。在这种情况下，公共语言运行时继续搜索当前异常的处理程序。

结合特定异常和用户筛选的子句

Catch 语句可以同时包含特定异常和用户筛选的子句。运行时首先测试特定异常。如果特定异常成功，运行时会执行用户筛选。普通筛选可包含对类筛选器中声明的变量的引用。请注意，两个筛选子句的顺序不能颠倒。

下面的 Visual Basic 示例介绍 Catch 语句中的特定异常 `ClassLoadException` 以及使用 When 关键字的用户筛选的子句。

```
Try
    'Try statements.
    Catch cle As ClassLoadException When cle.IsRecoverable()
    'Catch statements.
End Try
```

另请参阅

- 异常

处理 COM 互操作异常

2021/11/16 •

托管和非托管代码可协同工作来处理异常。如果方法在托管代码中引发异常，公共语言运行时可将 HRESULT 传递至 COM 对象。如果方法因返回失败 HRESULT 而在非托管代码中失败，运行时会引发可由托管代码捕获的异常。

运行时自动将 HRESULT 从 COM 互操作映射到更具体的异常。例如，E_ACCESSDENIED 成为 UnauthorizedAccessException、E_OUTOFMEMORY 成为 OutOfMemoryException，依次类推。

如果 HRESULT 为自定义结果或运行时不知道它，运行时会将泛型 COMException 传递到客户端。COMException 的 ErrorCode 属性包含 HRESULT 值。

处理 IErrorInfo

当错误从 COM 传递至托管代码时，运行时会将错误信息填充至异常对象。支持 IErrorInfo 并返回 HRESULT 的 COM 对象将向托管代码异常提供此信息。例如，运行时将“说明”从 COM 错误映射至异常的 Message 属性。如果 HRESULT 未提供任何其他错误信息，运行时将对很多异常的属性填充默认值。

如果方法在非托管代码中失败，则异常可以传递至托管代码段中。主题 HRESULTS 和异常中的表展示了如何将 HRESULTS 映射到运行时异常对象。

另请参阅

- 异常

异常的最佳做法

2021/11/16 •

设计良好的应用处理异常和错误以防止应用崩溃。本部分介绍了处理和创建异常的最佳做法。

使用 try/catch/finally 块从错误中恢复或释放资源

对可能生成异常的代码使用 `try` / `catch` 块，代码就可以从该异常中恢复。在 `catch` 块中，始终按从派生程度最高到派生程度最低的顺序对异常排序。所有异常都派生自 Exception。位于处理基本异常类的 catch 子句之后的 catch 子句不处理派生程度较高的异常。当代码无法从异常中恢复时，请勿捕获该异常。如有可能，请启用调用堆栈中更上层的方法来进行恢复。

使用 `using` 语句或 `finally` 块清除分配的资源。当引发了异常时，优先使用 `using` 语句自动清除资源。使用 `finally` 块清除未实现 IDisposable 的资源。即使引发了异常，通常也会执行 `finally` 子句中的代码。

在不引发异常的前提下，处理常见情况

对于易于发生但可能会触发异常的情况，请考虑使用能避免引发异常的方法进行处理。例如，如果尝试关闭已关闭的连接，则会获得 `InvalidOperationException`。尝试关闭前，可通过使用 `if` 语句检查连接状态，避免该情况。

```
if (conn->State != ConnectionState::Closed)
{
    conn->Close();
}
```

```
if (conn.State != ConnectionState.Closed)
{
    conn.Close();
}
```

```
If conn.State <> ConnectionState.Closed Then
    conn.Close()
End IF
```

如果关闭前未检查连接状态，则可能捕获 `InvalidOperationException` 异常。

```
try
{
    conn->Close();
}
catch (InvalidOperationException^ ex)
{
    Console::WriteLine(ex->GetType()->FullName);
    Console::WriteLine(ex->Message);
}
```

```
try
{
    conn.Close();
}
catch (InvalidOperationException ex)
{
    Console.WriteLine(ex.GetType().FullName);
    Console.WriteLine(ex.Message);
}
```

```
Try
    conn.Close()
Catch ex As InvalidOperationException
    Console.WriteLine(ex.GetType().FullName)
    Console.WriteLine(ex.Message)
End Try
```

选择的方法取决于希望时间发生的频率。

- 如果此事件未经常发生(也就是说，如果此事件确实为异常并指示错误(如意外的文件尾))，则使用异常处理。如果使用异常处理，将在正常条件下执行较少代码。
- 如果事件例行发生，且被视为正常性执行的一部分，请检查代码中是否存在错误情况。检查常见错误情况时，为了避免异常，执行较少的代码。

设计类，以避免异常

类可提供一些方法或属性来确保避免生成会引发异常的调用。例如，FileStream 类提供可帮助确实是否已到达文件末尾的方法。它可用于避免在读取超过文件末尾时引发的异常。下方示例显示如何读取文件末尾而不会引发异常。

```
class FileRead
{
public:
    void ReadAll(FileStream^ fileToRead)
    {
        // This if statement is optional
        // as it is very unlikely that
        // the stream would ever be null.
        if (fileToRead == nullptr)
        {
            throw gcnew System::ArgumentNullException();
        }

        int b;

        // Set the stream position to the beginning of the file.
        fileToRead->Seek(0, SeekOrigin::Begin);

        // Read each byte to the end of the file.
        for (int i = 0; i < fileToRead->Length; i++)
        {
            b = fileToRead->ReadByte();
            Console::Write(b.ToString());
            // Or do something else with the byte.
        }
    }
};
```

```
class FileRead
{
    public void ReadAll(FileStream fileToRead)
    {
        // This if statement is optional
        // as it is very unlikely that
        // the stream would ever be null.
        if (fileToRead == null)
        {
            throw new ArgumentNullException();
        }

        int b;

        // Set the stream position to the beginning of the file.
        fileToRead.Seek(0, SeekOrigin.Begin);

        // Read each byte to the end of the file.
        for (int i = 0; i < fileToRead.Length; i++)
        {
            b = fileToRead.ReadByte();
            Console.Write(b.ToString());
            // Or do something else with the byte.
        }
    }
}
```

```
Class FileRead
    Public Sub ReadAll(fileToRead As FileStream)
        ' This if statement is optional
        ' as it is very unlikely that
        ' the stream would ever be null.
        If fileToRead Is Nothing Then
            Throw New System.ArgumentNullException()
        End If

        Dim b As Integer

        ' Set the stream position to the beginning of the file.
        fileToRead.Seek(0, SeekOrigin.Begin)

        ' Read each byte to the end of the file.
        For i As Integer = 0 To fileToRead.Length - 1
            b = fileToRead.ReadByte()
            Console.Write(b.ToString())
            ' Or do something else with the byte.
        Next i
    End Sub
End Class
```

避免异常的另一方法是，对极为常见的错误案例返回 NULL（或默认值），而不是引发异常。极其常见的错误案例可被视为常规控制流。通过在这些情况下返回 NULL（或默认值），可最大程度地减小对应用的性能产生的影响。

对于值类型，是否使用 `Nullable<T>` 或默认值作为错误指示符是特定应用需要考虑的内容。通过使用 `Nullable<Guid>`，`default` 变为 `null` 而非 `Guid.Empty`。有时，添加 `Nullable<T>` 可更加明确值何时存在或不存在。在其他时候，添加 `Nullable<T>` 可以创建额外的案例以查看不必要的内容，并且仅用于创建潜在的错误源。

引发异常而不是返回错误代码

异常可确保故障不被忽略，因为调用代码不会检查返回代码。

使用预定义的 .NET 异常类型

仅当预定义的异常类不适用时，引入新异常类。例如：

- 如果根据对象的当前状态，属性集或方法调用不适当，则会引发 InvalidOperationException 异常。
- 如果传送了无效的参数，则引发 ArgumentException 异常或从 ArgumentException 派生的一个预定义类。

异常类名称的结尾为 `Exception`

需要自定义异常时，对其正确命名并从 Exception 类进行派生。例如：

```
public ref class MyFileNotFoundException : public Exception
{
};
```

```
public class MyFileNotFoundException : Exception
{
}
```

```
Public Class MyFileNotFoundException
    Inherits Exception
End Class
```

在自定义异常类中包括三种构造函数

创建自己的异常类时，请至少使用三种公共构造函数：无参数构造函数、采用字符串消息的构造函数以及采用字符串消息和内部异常的构造函数。

- Exception()（使用默认值）。
- Exception(String)，它接受字符串消息。
- Exception(String, Exception)，它接受字符串消息和内部异常。

有关示例，请参见 如何：创建用户定义的异常。

确保代码远程执行时异常数据可用

创建用户定义的异常时，请确保异常的元数据对远程执行的代码可用。

例如，在支持应用域的 .NET 实现中，异常可能会跨应用域抛出。假设应用域 A 创建应用域 B，后者执行引发异常的代码。应用域 A 若想正确捕获和处理异常，它必须能够找到包含应用域 B 所引发的异常的程序集。如果应用域 B 在其应用程序基下（但未在应用域 A 的应用程序基下）引发了一个包含在程序集内的异常，那么应用域 A 将无法找到异常，且公共语言运行时将引发 FileNotFoundException 异常。为避免此情况，可以两种方式部署包含异常信息的程序集：

- 将程序集放在两个应用域共享的公共应用程序基中。

  - 或 -

- 如果两个应用域不共享一个公共应用程序基，则用强名称为包含异常信息的程序集签名并将其部署到全局程序集缓存中。

使用语法正确的错误消息

编写清晰的句子，包括结束标点。分配给 Exception.Message 属性的字符串中的每个句子应以句点结尾。例如，“日志表已溢出”。是一个正确的消息字符串。

在每个异常中都包含一个本地化字符串消息

用户看到的错误消息派生自引发的异常的 Exception.Message 属性，而不是派生自异常类的名称。通常将值赋给 Exception.Message 属性，方法是将消息字符串传递到异常构造函数的 `message` 参数。

对于本地化应用程序，应为应用程序可能引发的每个异常提供本地化消息字符串。资源文件用于提供本地化错误消息。有关本地化应用程序和检索本地化字符串的信息，请参阅以下文章：

- 如何：使用本地化的异常消息创建用户定义的异常
- 桌面应用中的资源
- System.Resources.ResourceManager

在自定义异常中，按需提供其他属性

仅当存在附加信息有用的编程方案时，才在异常中提供附加属性（不包括自定义消息字符串）。例如，FileNotFoundException 提供 FileName 属性。

放置引发语句，使得堆栈跟踪有所帮助

堆栈跟踪从引发异常的语句开始，到捕获异常的 `catch` 语句结束。

使用异常生成器方法

类从其实现中的不同位置引发同一异常是常见的情况。为避免过多的代码，应使用帮助器方法创建异常并将其返回。例如：

```
ref class FileReader
{
private:
    String^ fileName;

public:
    FileReader(String^ path)
    {
        fileName = path;
    }

    array<Byte>^ Read(int bytes)
    {
        array<Byte>^ results = FileUtils::ReadFromFile(fileName, bytes);
        if (results == nullptr)
        {
            throw NewFileIOException();
        }
        return results;
    }

    FileReaderException^ NewFileIOException()
    {
        String^ description = "My NewFileIOException Description";

        return gcnew FileReaderException(description);
    }
};
```

```
class FileReader
{
    private string fileName;

    public FileReader(string path)
    {
        fileName = path;
    }

    public byte[] Read(int bytes)
    {
        byte[] results = FileUtils.ReadFromFile(fileName, bytes);
        if (results == null)
        {
            throw NewFileIOException();
        }
        return results;
    }

    FileReaderException NewFileIOException()
    {
        string description = "My NewFileIOException Description";

        return new FileReaderException(description);
    }
}
```

```
Class FileReader
    Private fileName As String


    Public Sub New(path As String)
        fileName = path
    End Sub

    Public Function Read(bytes As Integer) As Byte()
        Dim results() As Byte = FileUtils.ReadFromFile(fileName, bytes)
        If results Is Nothing
            Throw NewFileIOException()
        End If
        Return results
    End Function

    Function NewFileIOException() As FileReaderException
        Dim description As String = "My NewFileIOException Description"

        Return New FileReaderException(description)
    End Function
End Class
```

在某些情况下，更适合使用异常的构造函数生成异常。例如，ArgumentException 等全局异常类。

因发生异常而未完成方法时还原状态

当异常从方法引发时，调用方应能够假定没有副作用。例如，如果你的代码可以通过从一个帐户取钱并存入另一个帐户来转移资金，而在存款时引发了异常，你不希望取款仍然有效。

```
public void TransferFunds(Account from, Account to, decimal amount)
{
    from.Withdrawal(amount);
    // If the deposit fails, the withdrawal shouldn't remain in effect.
    to.Deposit(amount);
}
```

```
Public Sub TransferFunds(from As Account, [to] As Account, amount As Decimal)
    from.Withdrawal(amount)
    ' If the deposit fails, the withdrawal shouldn't remain in effect.
    [to].Deposit(amount)
End Sub
```

上面的方法不会直接引发任何异常，但必须以防御方式进行编写，以便在存款操作失败时撤销取款。

解决这一情况的一种方法是，捕获由存款交易引发的异常，然后回滚取款。

```
private static void TransferFunds(Account from, Account to, decimal amount)
{
    string withdrawalTrxID = from.Withdrawal(amount);
    try
    {
        to.Deposit(amount);
    }
    catch
    {
        from.RollbackTransaction(withdrawalTrxID);
        throw;
    }
}
```

```
Private Shared Sub TransferFunds(from As Account, [to] As Account, amount As Decimal)
    Dim withdrawalTrxID As String = from.Withdrawal(amount)
    Try
        [to].Deposit(amount)
    Catch
        from.RollbackTransaction(withdrawalTrxID)
        Throw
    End Try
End Sub
```

此示例介绍如何使用 `throw` 重新引发原始异常，让调用方更轻松地发现问题的真正原因，而无需检查 InnerException 属性。另一种方法是，引发一个新的异常并将原始异常包括在其中作为内部异常：

```
catch (Exception ex)
{
    from.RollbackTransaction(withdrawalTrxID);
    throw new TransferFundsException("Withdrawal failed.", innerException: ex)
    {
        From = from,
        To = to,
        Amount = amount
    };
}
```

```
Catch ex As Exception
    from.RollbackTransaction(withdrawalTrxID)
    Throw New TransferFundsException("Withdrawal failed.", innerException:=ex) With
    {
        .From = from,
        .[To] = [to],
        .Amount = amount
    }
End Try
```

请参阅

- 异常

.NET 中的数字

2021/11/16 •

.NET 提供了一系列数值整数和浮点基元，还提供：

- System.Numerics.BigInteger，它表示没有理论上的上限或下限的整型类型。
- System.Numerics.Complex，它表示复数。
- System.Numerics 命名空间中一组启用了 SIMD 的类型。

整数类型

.NET 支持带符号和无符号的 8 位、16 位、32 位和 64 位整数类型，如下表所示。

带符号整数类型

[illegible]	[illegible]	[illegible]	[illegible]
System.Int16	2	-32,768	32,767
System.Int32	4	-2,147,483,648	2,147,483,647
System.Int64	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
System.SByte	1	-128	127
System.IntPtr（32 位进程中）	4	-2,147,483,647	2,147,483,647
System.IntPtr（64 位进程中）	8	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

无符号整数类型

[illegible]	[illegible]	[illegible]	[illegible]
System.Byte	1	0	255
System.UInt16	2	0	65,535
System.UInt32	4	0	4,294,967,295
System.UInt64	8	0	18,446,744,073,709,551,615
System.UIntPtr（32 位进程中）	4	0	4,294,967,295
System.UIntPtr（64 位进程中）	8	0	18,446,744,073,709,551,615

每个整数类型都支持一组标准算术运算符。System.Math 类为更广泛的数学函数集提供方法。

还可以使用 System.BitConverter 类对整数值中的单个位进行运算。

> **NOTE**
> 无符号整数类型不符合 CLS。有关详细信息，请参阅语言独立性和与语言无关的组件。

BigInteger

System.Numerics.BigInteger 结构是不可变类型，表示其值没有理论上限或下限的任意大型整数。BigInteger 类型的方法几乎与其他整数类型的方法一致。

浮点类型

.NET 包含以下基元浮点类型：

[illegible]	[illegible]	[illegible]	[illegible]
System.Half	2	±65504	已在 .NET 5 中引入
System.Single	4	$\pm 3.4 \times 10^{38}$	
System.Double	8	$\pm 1.7 \times 10^{308}$	
System.Decimal	16	$\pm 7.9228 \times 10^{28}$	

Half、Single 和 Double 类型都支持表示非数字和无穷大的特殊值。例如，Double 类型提供以下值：Double.NaN、Double.NegativeInfinity 和 Double.PositiveInfinity。可以使用 Double.IsNaN、Double.IsInfinity、Double.IsPositiveInfinity 和 Double.IsNegativeInfinity 方法来测试这些特殊值。

每个浮点类型都支持一组标准的算术运算符。System.Math 类为更广泛的数学函数集提供方法。.NET Core 2.0 及更高版本包含 System.MathF 类，该类提供接受 Single 类型的参数的方法。

还可以使用 System.BitConverter 类对 Double、Single 和 Half 值中的单个位进行运算。System.Decimal 结构具有自己处理十进制值单个位的方法（Decimal.GetBits 和 Decimal(Int32[])）以及一套执行其他数学运算的方法。

Double、Single 和 Half 类型旨在用于本质上不精确的值（例如两颗行星之间的距离）和无需高度精确和舍入误差小的应用程序。在需要较高准确度和尽量减小舍入误差的情况下，使用 System.Decimal 类型。

> **NOTE**
> Decimal 类型不会消除对舍入的要求。相反，它最大限度地减少了因舍入而导致的错误。

Complex

System.Numerics.Complex 结构表示复数，即带实数部分和虚数部分的数字。此类型支持一套标准的算术、比较、相等、显式和隐式转换运算符，以及数学、代数和三角方法。

启用了 SIMD 的类型

System.Numerics 命名空间包含一组启用了 .NET SIMD 的类型。SIMD (Single Instruction Multiple Data) 操作可以在硬件级别并行化。这可以增加向量化计算的吞吐量，这在数学、科学和图形应用中很常见。

启用了 .NET SIMD 的类型如下：

- Vector2、Vector3 和 Vector4 类型，用于表示具有 2、3 和 4 Single 值的向量。
- 两个矩阵类型：Matrix3x2（表示 3x2 矩阵）和 Matrix4x4（表示 4x4 矩阵）。
- Plane 类型，表示三维空间中的一个平面。
- Quaternion 类型，表示一个用于对三维物理旋转进行编码的向量。
- Vector<T> 类型，表示指定数字类型的向量，并提供受益于 SIMD 支持的一组广泛的运算符。Vector<T> 实例的计数是固定的，但其值 Vector<T>.Count 取决于执行代码的计算机的 CPU。

  > **NOTE**
  > Vector<T> 类型随 .NET Core 和 .NET 5+ 一起提供，但 .NET Framework 中不提供。如果你使用的是 .NET Framework，请安装 System.Numerics.Vectors NuGet 包来访问此类型。

启用了 SIMD 的类型以这样一种方式实现：即它们可以与未启用 SIMD 的硬件或 JIT 编译器一起使用。要利用 SIMD 指令，你的 64 位应用必须由使用 RyuJIT 编译器的运行时运行，该编译器包含在 .NET Core 和 .NET Framework 4.6 及更高版本中。它针对 64 位处理器增加了 SIMD 支持。

有关详细信息，请参阅使用 SIMD 加速数值类型。

另请参阅

- 标准数字格式字符串
- 浮点数值类型 (C#)

日期、时间和时区

2021/11/16 •

除基本 DateTime 结构外，.NET 还提供以下支持处理时区的类型：

- TimeZone

  使用此类处理系统的本地时区和协调世界时 (UTC) 区域。类的功能在很大程度上被 类 TimeZone TimeZoneInfo 取代。

- TimeZoneInfo

  此类可用于处理系统上预定义的任何时区、创建新时区，以及将日期和时间从一个时区轻松转换成另一个时区。对于新的开发，使用 TimeZoneInfo 类而不使用 TimeZone 类。

- DateTimeOffset

  使用此结构处理已知 UTC 偏移量（或差值）的日期和时间。DateTimeOffset 结构结合了日期和时间值与该时间的 UTC 偏移量。由于其与 UTC 间的关系，单独的日期和时间值可以明确地识别单个时间点。这使 DateTimeOffset 值比 DateTime 值更容易从一台计算机移植到另一台计算机。

文档的此部分提供处理时区和创建时区识别应用程序所需的信息，时区识别程序可将日期和时间从一个时区转换到另一时区。

本节内容

时区概述讨论了创建时区识别应用程序时涉及的术语、概念以及问题。

在 DateTime、DateTimeOffset、TimeSpan 和 TimeZoneInfo 之间选择 讨论处理日期和时间数据时何时使用 、和 DateTime DateTimeOffset TimeZoneInfo 类型。

查找本地系统上定义的时区介绍如何枚举在本地系统上找到的时区。

如何：枚举计算机上存在的时区举例说明枚举在计算机注册表中定义的时区，以及允许用户从列表选择一个预定义时区。

如何：访问预定义的 UTC 和本地时区对象介绍如何访问协调世界时和本地时区。

如何：实例化 TimeZoneInfo 对象介绍如何实例化本地系统注册表中的 TimeZoneInfo 对象。

实例化 DateTimeOffset 对象讨论实例化 DateTimeOffset 对象的方式，以及可以将 DateTime 值转化为 DateTimeOffset 值的方法。

如何：创建不含调整规则的时区介绍如何创建不支持夏令时转换规则的自定义时区。

如何：创建含调整规则的时区介绍如何创建支持一个或多个夏令时转换规则的自定义时区。

保存和还原时区介绍 TimeZoneInfo 提供的时区数据序列化和反序列化支持，并通过一些应用场景介绍了如何使用这些功能。

如何：将时区保存到嵌入的资源中介绍如何创建自定义时区，并将其信息保存到资源文件中。

如何：从嵌入的资源还原时区介绍如何实例化已保存到嵌入的资源文件中的自定义时区。

使用日期和时间执行算术运算讨论加上，减去和比较 DateTime 与 DateTimeOffset 值时会出现的问题。

如何：在日期和时间算法中使用时区讨论如何执行反映时区调整规则的日期和时间算术。

在 DateTime 与 DateTimeOffset 之间进行转换介绍如何在 DateTime 和 DateTimeOffset 值间进行转换。

在各时区之间转换时间介绍如何将时间从一个时区转换到另一个时区。

如何：解决不明确时间介绍如何通过将不明确时间映射到时区标准时间解决该时间。

如何：让用户解决不明确时间介绍如何让用户确定不明确本地时间与协调世界时之间的映射。

参考

System.TimeZoneInfo

利用特性扩展元数据

2021/11/16 •

公共语言运行时使你能够添加类似于关键字的描述性声明（称为特性），以便批注编程元素（如类型、字段、方法和属性）。编译运行时的代码时，它将被转换为 Microsoft 中间语言 (MSIL)，并和编译器生成的元数据一起放置在可移植可执行 (PE) 文件内。特性使你能够将额外的描述性信息放到可使用运行时反射服务提取的元数据中。当你声明派生自 System.Attribute 的特殊类的实例时，编译器会创建特性。

.NET 出于多种原因且为解决许多问题而使用特性。特性描述如何将数据序列化、指定用于强制安全性的特征并限制通过实时 (JIT) 编译器进行优化，从而使代码易于调试。特性还可记录文件的名称或代码的作者，或控制窗体开发过程中控件和成员的可见性。

相关文章

TITLE	
应用特性	描述如何将特性应用于代码的元素。
编写自定义特性	描述如何设计自定义特性类。
检索存储在特性中的信息	描述如何检索加载到执行上下文中的代码的自定义特性。
元数据和自描述组件	提供元数据的概述，并说明它是如何在 .NET 可移植可执行 (PE) 文件中实现的。
如何：将程序集加载到仅反射上下文中	说明如何检索仅反射上下文中的自定义特性信息。

参考

- System.Attribute

应用属性

2021/11/16 •

使用以下过程将特性应用于代码的元素。

1. 定义新特性或使用现有的 .NET 特性。
2. 通过将特性置于紧邻元素之前，将该特性应用于代码元素。

   每种语言都有其自己的特性语法。在 C++ 和 C# 中，特性是用方括号括起来的，并通过空格与元素分开（可能包括换行符）。在 Visual Basic 中，特性是用尖括号括起来的，且必须位于同一逻辑线上；如果需要换行符，可使用行继续字符。
3. 指定特性的位置参数和命名参数。

   位置参数是必需的，且必须位于所有命名参数之前；它们对应于特性的其中一个构造函数的参数。命名参数是可选的且对应于特性的读/写属性。在 C++ 和 C# 中，为每个可选参数指定 `name=value`，其中 `name` 是属性名。在 Visual Basic 中，指定 `name:=value`。

编译代码时，特性将被发到元数据中，并且通过运行时反射服务可用于公共语言运行时和任何自定义工具或应用程序。

按照惯例，所有特性名称都以“Attribute”结尾。但是，面向运行时的几种语言（如 Visual Basic 和 C#）无需指定特性的全名。例如，若要初始化 System.ObsoleteAttribute，只需将它引用为 Obsolete 即可。

将特性应用于方法

以下代码示例显示如何使用 System.ObsoleteAttribute（其将代码标记为已过时）。将字符串 `"Will be removed in next version"` 传递给特性。当特性描述的代码被调用时，此特性会导致产生编译器警告，显示传递的字符串。

```
public ref class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
public:
    [Obsolete("Will be removed in next version.")]
    static int Add(int a, int b)
    {
        return (a + b);
    }
};

ref class Test
{
public:
    static void Main()
    {
        // This generates a compile-time warning.
        int i = Example::Add(2, 2);
    }
};

int main()
{
    Test::Main();
}
```

```
public class Example
{
    // Specify attributes between square brackets in C#.
    // This attribute is applied only to the Add method.
    [Obsolete("Will be removed in next version.")]
    public static int Add(int a, int b)
    {
        return (a + b);
    }
}

class Test
{
    public static void Main()
    {
        // This generates a compile-time warning.
        int i = Example.Add(2, 2);
    }
}
```

```
Public Class Example
    ' Specify attributes between square brackets in C#.
    ' This attribute is applied only to the Add method.
    <Obsolete("Will be removed in next version.")>
    Public Shared Function Add(a As Integer, b As Integer) As Integer
        Return a + b
    End Function
End Class

Class Test
    Public Shared Sub Main()
        ' This generates a compile-time warning.
        Dim i As Integer = Example.Add(2, 2)
    End Sub
End Class
```

在程序集级别应用特性

如果要在程序集级别应用属性，请使用 assembly `Assembly`（Visual Basic 中用 `assembly`）关键字。下列代码显示在程序集级别应用的 AssemblyTitleAttribute。

```
using namespace System::Reflection;
[assembly:AssemblyTitle("My Assembly")];
```

```
using System.Reflection;
[assembly:AssemblyTitle("My Assembly")]
```

```
Imports System.Reflection
<Assembly: AssemblyTitle("My Assembly")>
```

应用此特性时，字符串 `"My Assembly"` 将被放置在文件元数据部分的程序集清单中。可通过后列方法查看特性：使用 MSIL 反汇编程序 (Ildasm.exe)，或创建一个自定义程序来检索特性。

请参阅

- 特性
- 检索存储在特性中的信息
- 概念
- 特性 (C#)
- 特性概述 (Visual Basic)

编写自定义特性

2021/11/16 •

要设计你自己的自定义特性，无需掌握许多新的概念。如果你熟悉面向对象的编程，并且知道如何设计类，那么你已经具备大部分所需知识。自定义特性本质上是直接或间接派生自 System.Attribute的传统类。与传统类一样，自定义特性包含用于存储和检索数据的方法。

正确设计自定义特性的主要步骤如下：

- 应用 AttributeUsageAttribute
- 声明特性类
- 声明构造函数
- 声明属性

本节描述上述各个步骤，并以 自定义特性的示例结束本节的描述。

应用 AttributeUsageAttribute

自定义特性声明以 System.AttributeUsageAttribute 开头，定义了特性类的一些主要特征。例如，你可以指定其他类是否可以继承你的特性，或者指定此特性可以应用到哪些元素。下面的代码片段演示了 AttributeUsageAttribute 的使用方式。

```
[AttributeUsage(AttributeTargets::All, Inherited = false, AllowMultiple = true)]
```

```
[AttributeUsage(AttributeTargets.All, Inherited = false, AllowMultiple = true)]
```

```
<AttributeUsage(AttributeTargets.All, Inherited:=False, AllowMultiple:=True)>
Public Class SomeClass
    Inherits Attribute
    '...
End Class
```

AttributeUsageAttribute 包含下列三个成员，它们对创建自定义属性非常重要：AttributeTargets、Inherited 和 AllowMultiple。

AttributeTargets 成员

上述示例中指定了 AttributeTargets.All，它表示此特性可应用于所有程序元素。或者，你可指定 AttributeTargets.Class 和 AttributeTargets.Method，前者表示你的特性仅可适用于一个类，后者表示你的特性仅可应用于一种方法。所有程序元素都可以通过这种方式使用自定义特性来标记，以对其进行描述。

你还可传递多个 AttributeTargets 值。下面的代码段指定了自定义特性可应用于任何类或方法。

```
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Method)]
```

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
```

```
<AttributeUsage(AttributeTargets.Class Or AttributeTargets.Method)>
Public Class SomeOtherClass
    Inherits Attribute
    '...
End Class
```

Inherited 属性

AttributeUsageAttribute.Inherited 属性指明要对其应用属性的类的派生类能否继承此属性。此属性使用 `true`（默认值）或 `false` 标志。在下例中，`MyAttribute` 的 Inherited 值为默认值 `true`，而 `YourAttribute` 的 Inherited 值为 `false`。

```
// This defaults to Inherited = true.
public ref class MyAttribute : Attribute
{
    //...
};

[AttributeUsage(AttributeTargets::Method, Inherited = false)]
public ref class YourAttribute : Attribute
{
    //...
};
```

```
// This defaults to Inherited = true.
public class MyAttribute : Attribute
{
    //...
}

[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class YourAttribute : Attribute
{
    //...
}
```

```
' This defaults to Inherited = true.
Public Class MyAttribute
    Inherits Attribute
    '...
End Class

<AttributeUsage(AttributeTargets.Method, Inherited:=False)>
Public Class YourAttribute
    Inherits Attribute
    '...
End Class
```

然后，这两个特性应用于基类 `MyClass` 中的方法。

```
public ref class MyClass
{
public:
    [MyAttribute]
    [YourAttribute]
    virtual void MyMethod()
    {
        //...
    }
};
```

```
public class MyClass
{
    [MyAttribute]
    [YourAttribute]
    public virtual void MyMethod()
    {
        //...
    }
}
```

```
Public Class MeClass
    <MyAttribute>
    <YourAttribute>
    Public Overridable Sub MyMethod()
        '...
    End Sub
End Class
```

最后，从基类 `YourClass` **中继承类** `MyClass` **。方法** `MyMethod` **显示** `MyAttribute`**，但不显示** `YourAttribute` **。**

```
public ref class YourClass : MyClass
{
public:
    // MyMethod will have MyAttribute but not YourAttribute.
    virtual void MyMethod() override
    {
        //...
    }

};
```

```
public class YourClass : MyClass
{
    // MyMethod will have MyAttribute but not YourAttribute.
    public override void MyMethod()
    {
        //...
    }
}
```

```
Public Class YourClass
    Inherits MeClass
    ' MyMethod will have MyAttribute but not YourAttribute.
    Public Overrides Sub MyMethod()
        '...
    End Sub

End Class
```

AllowMultiple 属性

AttributeUsageAttribute.AllowMultiple 属性指明元素能否包含属性的多个实例。如果设置为 `true`，则允许多个实例；如果设置为 `false`（默认值），则只允许一个实例。

在下例中，`MyAttribute` 的 AllowMultiple 值为默认值 `false`，而 `YourAttribute` 的值为 `true`。

```
//This defaults to AllowMultiple = false.
public ref class MyAttribute : Attribute
{
};

[AttributeUsage(AttributeTargets::Method, AllowMultiple = true)]
public ref class YourAttribute : Attribute
{
};
```

```
//This defaults to AllowMultiple = false.
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class YourAttribute : Attribute
{
}
```

```
' This defaults to AllowMultiple = false.
Public Class MyAttribute
    Inherits Attribute
End Class

<AttributeUsage(AttributeTargets.Method, AllowMultiple:=true)>
Public Class YourAttribute
    Inherits Attribute
End Class
```

当应用这些特性的多个实例时，`MyAttribute` 会生成编译器错误。下面的代码示例显示 `YourAttribute` 的有效用法和 `MyAttribute` 的无效用法。

```
public ref class MyClass
{
public:
    // This produces an error.
    // Duplicates are not allowed.
    [MyAttribute]
    [MyAttribute]
    void MyMethod()
    {
        //...
    }

    // This is valid.
    [YourAttribute]
    [YourAttribute]
    void YourMethod()
    {
        //...
    }
};
```

```
public class MyClass
{
    // This produces an error.
    // Duplicates are not allowed.
    [MyAttribute]
    [MyAttribute]
    public void MyMethod()
    {
        //...
    }

    // This is valid.
    [YourAttribute]
    [YourAttribute]
    public void YourMethod()
    {
        //...
    }
}
```

```
Public Class MyClass
    ' This produces an error.
    ' Duplicates are not allowed.
    <MyAttribute>
    <MyAttribute>
    Public Sub MyMethod()
        '...
    End Sub

    ' This is valid.
    <YourAttribute>
    <YourAttribute>
    Public Sub YourMethod()
        '...
    End Sub
End Class
```

如果 AllowMultiple 属性和 Inherited 属性都设置为 `true`，则从另一个类继承的类可继承一个特性，并且具有同一子类中应用的同一特性的另一个实例。如果 AllowMultiple 设置为 `false`，则父类中的所有特性的值将被子类中同一特性的新实例覆盖。

声明特性类

应用 AttributeUsageAttribute以后，可以开始定义特性的细节。特性类的声明类似于传统类的声明，如以下代码所示。

```
[AttributeUsage(AttributeTargets::Method)]
public ref class MyAttribute : Attribute
{
    // . . .
};
```

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
    // . . .
}
```

```
<AttributeUsage(AttributeTargets.Method)>
Public Class MyAttribute
    Inherits Attribute
    ' . . .
End Class
```

此特性定义说明了以下几点：

- 特性类必须声明为公共类。
- 按照约定，特性类的名称以单词 **Attribute** 结束。尽管没有要求，但仍建议执行此约定以保证可读性。应用特性时，可以选择是否包含单词 Attribute。
- 所有特性类必须直接或间接从 System.Attribute 继承。
- 在 Microsoft Visual Basic 中，所有自定义特性类必须具有 System.AttributeUsageAttribute 特性。

声明构造函数

与传统类的初始化方式相同，使用构造函数初始化特性。下面的代码段阐明了典型的特性构造函数。此公共构造函数采用一个参数，并设置一个等于其值的成员变量。

```
MyAttribute(bool myvalue)
{
    this->myvalue = myvalue;
}
```

```
public MyAttribute(bool myvalue)
{
    this.myvalue = myvalue;
}
```

```
Public Sub New(myvalue As Boolean)
    Me.myvalue = myvalue
End Sub
```

可以重载此构造函数以适应值的各种组合。如果你还为自定义特性类定义了 属性，则在初始化该特性时可以使用命名参数和定位参数的组合。通常情况下，将所有必选的参数定义为定位参数，将所有可选的参数定义为命名

参数。在这种情况下，没有必选参数则无法初始化特性。其他所有参数都是可选参数。请注意，在 Visual Basic 中，特性类的构造函数不应使用 ParamArray 参数。

下面的代码示例显示如何使用可选和必选参数应用使用上例中的构造函数的特性。该示例假定特性有一个必选的布尔值和一个可选的字符串属性。

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public ref class SomeClass
{
    //...
};
// One required (positional) parameter is applied.
[MyAttribute(false)]
public ref class SomeOtherClass
{
    //...
};
```

```
// One required (positional) and one optional (named) parameter are applied.
[MyAttribute(false, OptionalParameter = "optional data")]
public class SomeClass
{
    //...
}
// One required (positional) parameter is applied.
[MyAttribute(false)]
public class SomeOtherClass
{
    //...
}
```

```
' One required (positional) and one optional (named) parameter are applied.
<MyAttribute(false, OptionalParameter:="optional data")>
Public Class SomeClass
    '...
End Class

' One required (positional) parameter is applied.
<MyAttribute(false)>
Public Class SomeOtherClass
    '...
End Class
```

声明属性

如果你想要定义一个命名参数，或者提供一种简单的方法来返回由特性存储的值，请声明 属性。应将特性的属性声明为公共实体，此公告实体包含将返回的数据类型的描述。定义将保存属性值的变量，并将此变量与 get 和 set 方法相关联。下面的代码示例说明如何在特性中实现一个简单属性。

```
property bool MyProperty
{
    bool get() {return this->myvalue;}
    void set(bool value) {this->myvalue = value;}
}
```

```
public bool MyProperty
{
    get {return this.myvalue;}
    set {this.myvalue = value;}
}
```

```
Public Property MyProperty As Boolean
    Get
        Return Me.myvalue
    End Get
    Set
        Me.myvalue = Value
    End Set
End Property
```

自定义特性的示例

本节内容结合了前面的信息，显示如何设计一个简单特性来记录有关代码段的作者的信息。本示例中的特性存储了编程人员的姓名和级别，以及是否已检查此代码的信息。它使用三个私有变量来存储要保存的实际值。每个变量用获取和设置这些值的公共属性表示。最后，使用两个必选参数定义构造函数。

```
[AttributeUsage(AttributeTargets::All)]
public ref class DeveloperAttribute : Attribute
{
    // Private fields.
private:
    String^ name;
    String^ level;
    bool reviewed;

public:
    // This constructor defines two required parameters: name and level.

    DeveloperAttribute(String^ name, String^ level)
    {
        this->name = name;
        this->level = level;
        this->reviewed = false;
    }

    // Define Name property.
    // This is a read-only attribute.

    virtual property String^ Name
    {
        String^ get() {return name;}
    }

    // Define Level property.
    // This is a read-only attribute.

    virtual property String^ Level
    {
        String^ get() {return level;}
    }

    // Define Reviewed property.
    // This is a read/write attribute.

    virtual property bool Reviewed
    {
        bool get() {return reviewed;}
        void set(bool value) {reviewed = value;}
    }
};
```

```
[AttributeUsage(AttributeTargets.All)]
public class DeveloperAttribute : Attribute
{
    // Private fields.
    private string name;
    private string level;
    private bool reviewed;

    // This constructor defines two required parameters: name and level.

    public DeveloperAttribute(string name, string level)
    {
        this.name = name;
        this.level = level;
        this.reviewed = false;
    }

    // Define Name property.
    // This is a read-only attribute.

    public virtual string Name
    {
        get {return name;}
    }

    // Define Level property.
    // This is a read-only attribute.

    public virtual string Level
    {
        get {return level;}
    }

    // Define Reviewed property.
    // This is a read/write attribute.

    public virtual bool Reviewed
    {
        get {return reviewed;}
        set {reviewed = value;}
    }
}
```

```
<AttributeUsage(AttributeTargets.All)>
Public Class DeveloperAttribute
    Inherits Attribute
    ' Private fields.
    Private myname As String
    Private mylevel As String
    Private myreviewed As Boolean

    ' This constructor defines two required parameters: name and level.

    Public Sub New(name As String, level As String)
        Me.myname = name
        Me.mylevel = level
        Me.myreviewed = False
    End Sub

    ' Define Name property.
    ' This is a read-only attribute.

    Public Overridable ReadOnly Property Name() As String
        Get
            Return myname
        End Get
    End Property

    ' Define Level property.
    ' This is a read-only attribute.

    Public Overridable ReadOnly Property Level() As String
        Get
            Return mylevel
        End Get
    End Property

    ' Define Reviewed property.
    ' This is a read/write attribute.

    Public Overridable Property Reviewed() As Boolean
        Get
            Return myreviewed
        End Get
        Set
            myreviewed = value
        End Set
    End Property
End Class
```

可以采用以下任一种方法，使用全称 `DeveloperAttribute` 或缩写名称 `Developer` 应用此特性。

```
[Developer("Joan Smith", "1")]

-or-

[Developer("Joan Smith", "1", Reviewed = true)]
```


```
[Developer("Joan Smith", "1")]

-or-

[Developer("Joan Smith", "1", Reviewed = true)]
```

```
<Developer("Joan Smith", "1")>

-or-

<Developer("Joan Smith", "1", Reviewed := true)>
```

第一个示例显示只应用了必选命名参数的特性，第二个示例显示同时应用了必选参数和可选参数的特性。

请参阅

- System.Attribute
- System.AttributeUsageAttribute
- 特性

检索存储在特性中的信息

2021/11/16 •

检索自定义属性的过程非常简单。首先，声明要检索的属性实例。然后，使用 Attribute.GetCustomAttribute 方法，用要检索的属性的值初始化新属性。在初始化新属性后，只需使用它的属性即可获取值。

> **IMPORTANT**
> 本主题介绍了如何为执行上下文中加载的代码检索属性。若要为仅反射上下文中加载的代码检索属性，必须使用 CustomAttributeData 类，如以下所述：如何：将程序集加载到仅反射上下文中。

此部分介绍了如何通过以下方式检索属性：

- 检索一个属性实例
- 检索应用于同一范围的多个属性实例
- 检索应用于不同范围的多个属性实例

检索一个属性实例

在下面的示例中，`DeveloperAttribute`（如上一部分所述）在类一级适用于 `MainApp` 类。`GetAttribute` 方法使用 GetCustomAttribute 在类一级检索 `DeveloperAttribute` 中存储的值，再在控制台中显示它们。

```
using namespace System;
using namespace System::Reflection;
using namespace CustomCodeAttributes;

[Developer("Joan Smith", "42", Reviewed = true)]
ref class MainApp
{
public:
    static void Main()
    {
        // Call function to get and display the attribute.
        GetAttribute(MainApp::typeid);
    }

    static void GetAttribute(Type^ t)
    {
        // Get instance of the attribute.
        DeveloperAttribute^ MyAttribute =
            (DeveloperAttribute^) Attribute::GetCustomAttribute(t, DeveloperAttribute::typeid);

        if (MyAttribute == nullptr)
        {
            Console::WriteLine("The attribute was not found.");
        }
        else
        {
            // Get the Name value.
            Console::WriteLine("The Name Attribute is: {0}." , MyAttribute->Name);
            // Get the Level value.
            Console::WriteLine("The Level Attribute is: {0}." , MyAttribute->Level);
            // Get the Reviewed value.
            Console::WriteLine("The Reviewed Attribute is: {0}." , MyAttribute->Reviewed);
        }
    }
};
```

```
using System;
using System.Reflection;
using CustomCodeAttributes;

[Developer("Joan Smith", "42", Reviewed = true)]
class MainApp
{
    public static void Main()
    {
        // Call function to get and display the attribute.
        GetAttribute(typeof(MainApp));
    }

    public static void GetAttribute(Type t)
    {
        // Get instance of the attribute.
        DeveloperAttribute MyAttribute =
            (DeveloperAttribute) Attribute.GetCustomAttribute(t, typeof (DeveloperAttribute));

        if (MyAttribute == null)
        {
            Console.WriteLine("The attribute was not found.");
        }
        else
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." , MyAttribute.Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." , MyAttribute.Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}." , MyAttribute.Reviewed);
        }
    }
}
```

```
Imports System.Reflection
Imports CustomCodeAttributes

<Developer("Joan Smith", "42", Reviewed:=True)>
Class MainApp
    Public Shared Sub Main()
        ' Call function to get and display the attribute.
        GetAttribute(GetType(MainApp))
    End Sub

    Public Shared Sub GetAttribute(t As Type)
        ' Get instance of the attribute.
        Dim MyAttribute As DeveloperAttribute =
            CType(Attribute.GetCustomAttribute(t, GetType(DeveloperAttribute)), DeveloperAttribute)

        If MyAttribute Is Nothing Then
            Console.WriteLine("The attribute was not found.")
        Else
            ' Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}.", MyAttribute.Name)
            ' Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}.", MyAttribute.Level)
            ' Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}.", MyAttribute.Reviewed)
        End If
    End Sub
End Class
```

此程序在执行时显示以下文本。

```
The Name Attribute is: Joan Smith.
The Level Attribute is: 42.
The Reviewed Attribute is: True.
```

如果找不到属性，GetCustomAttribute 方法会将 `MyAttribute` 初始化为 NULL 值。此示例在 `MyAttribute` 中查找此类实例，并在找不到属性时通知用户。如果在类范围中找不到 `DeveloperAttribute`，控制台中显示以下消息。

```
The attribute was not found.
```

此示例假定属性定义位于当前的命名空间中。请注意，如果属性定义不在当前的命名空间中，请导入属性定义所在的命名空间。

检索应用于同一范围的多个属性实例

在上一示例中，要检查的类和要查找的特定属性都传递给 GetCustomAttribute。此代码非常适用于只有一个属性实例在类一级应用的情况。不过，如果在相同的类一级应用多个属性实例，GetCustomAttribute 方法不会检索所有信息。如果同一属性的多个实例应用于相同范围，可以使用 Attribute.GetCustomAttributes 将所有属性实例添加到数组中。例如，如果在相同的类一级应用两个 `DeveloperAttribute` 实例，可以将 `GetAttribute` 方法修改为显示在这两个属性中找到的信息。请注意，若要在同一级别应用多个属性，必须在 AttributeUsageAttribute 中定义属性，并将 AllowMultiple 属性设为 true。

下面的代码示例展示了如何使用 GetCustomAttributes 方法来创建数组，以引用任何给定类中的所有 `DeveloperAttribute` 实例。然后，所有属性的值都显示在控制台中。

```
public:
    static void GetAttribute(Type^ t)
    {
        array<DeveloperAttribute^>^ MyAttributes =
            (array<DeveloperAttribute^>^) Attribute::GetCustomAttributes(t, DeveloperAttribute::typeid);

        if (MyAttributes->Length == 0)
        {
            Console::WriteLine("The attribute was not found.");
        }
        else
        {
            for (int i = 0 ; i < MyAttributes->Length; i++)
            {
                // Get the Name value.
                Console::WriteLine("The Name Attribute is: {0}." , MyAttributes[i]->Name);
                // Get the Level value.
                Console::WriteLine("The Level Attribute is: {0}." , MyAttributes[i]->Level);
                // Get the Reviewed value.
                Console::WriteLine("The Reviewed Attribute is: {0}.", MyAttributes[i]->Reviewed);
            }
        }
    }
```

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute[] MyAttributes =
        (DeveloperAttribute[]) Attribute.GetCustomAttributes(t, typeof (DeveloperAttribute));

    if (MyAttributes.Length == 0)
    {
        Console.WriteLine("The attribute was not found.");
    }
    else
    {
        for (int i = 0 ; i < MyAttributes.Length ; i++)
        {
            // Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}." , MyAttributes[i].Name);
            // Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}." , MyAttributes[i].Level);
            // Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}.", MyAttributes[i].Reviewed);
        }
    }
}
```

```
Public Shared Sub GetAttribute(t As Type)
    Dim MyAttributes() As DeveloperAttribute =
        CType(Attribute.GetCustomAttributes(t, GetType(DeveloperAttribute)), DeveloperAttribute())

    If MyAttributes.Length = 0 Then
        Console.WriteLine("The attribute was not found.")
    Else
        For i As Integer = 0 To MyAttributes.Length - 1
            ' Get the Name value.
            Console.WriteLine("The Name Attribute is: {0}.", MyAttributes(i).Name)
            ' Get the Level value.
            Console.WriteLine("The Level Attribute is: {0}.", MyAttributes(i).Level)
            ' Get the Reviewed value.
            Console.WriteLine("The Reviewed Attribute is: {0}.", MyAttributes(i).Reviewed)
        Next i
    End If
End Sub
```

如果找不到任何属性，此代码会向用户发出警报。如果找到，就会显示两个 `DeveloperAttribute` 实例中包含的信息。

检索应用于不同范围的多个属性实例

GetCustomAttributes 和 GetCustomAttribute 方法不会搜索整个类，也不会返回相应类中的所有属性实例。而是一次只搜索一个指定方法或成员。如果类中的相同属性应用于每个成员，且要检索应用于这些成员的所有属性的值，必须将各个方法或成员单独提供给 GetCustomAttributes 和 GetCustomAttribute。

下面的代码示例将类用作参数，并在类一级以及相应类的所有方法一级搜索 `DeveloperAttribute`（如前面所定义）。

```
public:
    static void GetAttribute(Type^ t)
    {
        DeveloperAttribute^ att;

        // Get the class-level attributes.

        // Put the instance of the attribute on the class level in the att object.
        att = (DeveloperAttribute^) Attribute::GetCustomAttribute (t, DeveloperAttribute::typeid);

        if (att == nullptr)
        {
            Console::WriteLine("No attribute in class {0}.\n", t->ToString());
        }
        else
        {
            Console::WriteLine("The Name Attribute on the class level is: {0}.", att->Name);
            Console::WriteLine("The Level Attribute on the class level is: {0}.", att->Level);
            Console::WriteLine("The Reviewed Attribute on the class level is: {0}.\n", att->Reviewed);
        }

        // Get the method-level attributes.

        // Get all methods in this class, and put them
        // in an array of System.Reflection.MemberInfo objects.
        array<MemberInfo^>^ MyMemberInfo = t->GetMethods();

        // Loop through all methods in this class that are in the
        // MyMemberInfo array.
        for (int i = 0; i < MyMemberInfo->Length; i++)
        {
            att = (DeveloperAttribute^) Attribute::GetCustomAttribute(MyMemberInfo[i],
DeveloperAttribute::typeid);
            if (att == nullptr)
            {
                Console::WriteLine("No attribute in member function {0}.\n" , MyMemberInfo[i]->ToString());
            }
            else
            {
                Console::WriteLine("The Name Attribute for the {0} member is: {1}.",
                    MyMemberInfo[i]->ToString(), att->Name);
                Console::WriteLine("The Level Attribute for the {0} member is: {1}.",
                    MyMemberInfo[i]->ToString(), att->Level);
                Console::WriteLine("The Reviewed Attribute for the {0} member is: {1}.\n",
                    MyMemberInfo[i]->ToString(), att->Reviewed);
            }
        }
    }
```

```
public static void GetAttribute(Type t)
{
    DeveloperAttribute att;

    // Get the class-level attributes.

    // Put the instance of the attribute on the class level in the att object.
    att = (DeveloperAttribute) Attribute.GetCustomAttribute (t, typeof (DeveloperAttribute));

    if (att == null)
    {
        Console.WriteLine("No attribute in class {0}.\n", t.ToString());
    }
    else
    {
        Console.WriteLine("The Name Attribute on the class level is: {0}.", att.Name);
        Console.WriteLine("The Level Attribute on the class level is: {0}.", att.Level);
        Console.WriteLine("The Reviewed Attribute on the class level is: {0}.\n", att.Reviewed);
    }

    // Get the method-level attributes.

    // Get all methods in this class, and put them
    // in an array of System.Reflection.MemberInfo objects.
    MemberInfo[] MyMemberInfo = t.GetMethods();

    // Loop through all methods in this class that are in the
    // MyMemberInfo array.
    for (int i = 0; i < MyMemberInfo.Length; i++)
    {
        att = (DeveloperAttribute) Attribute.GetCustomAttribute(MyMemberInfo[i], typeof
(DeveloperAttribute));
        if (att == null)
        {
            Console.WriteLine("No attribute in member function {0}.\n" , MyMemberInfo[i].ToString());
        }
        else
        {
            Console.WriteLine("The Name Attribute for the {0} member is: {1}.",
                MyMemberInfo[i].ToString(), att.Name);
            Console.WriteLine("The Level Attribute for the {0} member is: {1}.",
                MyMemberInfo[i].ToString(), att.Level);
            Console.WriteLine("The Reviewed Attribute for the {0} member is: {1}.\n",
                MyMemberInfo[i].ToString(), att.Reviewed);
        }
    }
}
```

```
Public Shared Sub GetAttribute(t As Type)
    Dim att As DeveloperAttribute

    ' Get the class-level attributes.

    ' Put the instance of the attribute on the class level in the att object.
    att = CType(Attribute.GetCustomAttribute(t, GetType(DeveloperAttribute)), DeveloperAttribute)

    If att Is Nothing
        Console.WriteLine("No attribute in class {0}.\n", t.ToString())
    Else
        Console.WriteLine("The Name Attribute on the class level is: {0}.", att.Name)
        Console.WriteLine("The Level Attribute on the class level is: {0}.", att.Level)
        Console.WriteLine("The Reviewed Attribute on the class level is: {0}.\n", att.Reviewed)
    End If

    ' Get the method-level attributes.

    ' Get all methods in this class, and put them
    ' in an array of System.Reflection.MemberInfo objects.
    Dim MyMemberInfo() As MemberInfo = t.GetMethods()

    ' Loop through all methods in this class that are in the
    ' MyMemberInfo array.
    For i As Integer = 0 To MyMemberInfo.Length - 1
        att = CType(Attribute.GetCustomAttribute(MyMemberInfo(i), _
            GetType(DeveloperAttribute)), DeveloperAttribute)
        If att Is Nothing Then
            Console.WriteLine("No attribute in member function {0}.\n", MyMemberInfo(i).ToString())
        Else
            Console.WriteLine("The Name Attribute for the {0} member is: {1}.",
                MyMemberInfo(i).ToString(), att.Name)
            Console.WriteLine("The Level Attribute for the {0} member is: {1}.",
                MyMemberInfo(i).ToString(), att.Level)
            Console.WriteLine("The Reviewed Attribute for the {0} member is: {1}.\n",
                MyMemberInfo(i).ToString(), att.Reviewed)
        End If
    Next
End Sub
```

如果在方法或类一级找不到 `DeveloperAttribute` 实例，`GetAttribute` 方法会通知用户找不到属性，并显示不包含属性的方法名称或类名称。如果找到属性，控制台中会显示 `Name`、`Level` 和 `Reviewed` 字段。

可以使用 Type 类的成员，在传递的类中获取各个方法和成员。此示例先查询 Type 对象，以获取类一级的属性信息。接下来，它使用 Type.GetMethods 将所有方法实例都放入 System.Reflection.MemberInfo 对象数组，以检索方法一级的属性信息。还可以使用 Type.GetProperties 方法检查属性一级的属性，或使用 Type.GetConstructors方法检查构造函数一级的属性。

请参阅

- System.Type
- Attribute.GetCustomAttribute
- Attribute.GetCustomAttributes
- 特性

.NET 中的格式类型

2021/11/16 •

格式设置是指将类、结构或枚举值的实例转换为其字符串表示形式的过程，通常使得最终的字符串可以显示给用户，或者进行反序列化以还原为原始数据类型。此转换可能面临一系列挑战：

- 在内部存储值的方式不一定反映用户想要查看它们的方式。例如，电话号码可以存储为 8009999999 格式，但此格式并非是用户友好的格式。该电话号码应显示为 800-999-9999。有关以这种方式设置数字格式的示例，请参见 自定义格式字符串 一节。
- 有时对象到其字符串表示形式的转换不是直观的。例如，不清楚 Temperature 对象或 Person 对象的字符串表示形式应如何显示。有关以各种方式设置 Temperature 对象格式的示例，请参见 标准格式字符串 一节。
- 值通常需要区分区域性的格式。例如，在使用数字表示货币值的应用程序中，数字字符串应包括当前区域性的货币符号、组分隔符（在大多数区域性中，组分隔符为千位分隔符）和小数点符号。有关示例，请参阅使用格式提供程序进行区分区域性的格式设置部分。
- 应用程序可能需要以不同方式显示相同的值。例如，应用程序可能通过显示名称的字符串表示形式来表示一个枚举成员，或通过显示基础值来表示该枚举成员。有关以不同方式设置 DayOfWeek 枚举成员格式的示例，请参见 标准格式字符串 一节。

> **NOTE**
> 格式设置将类型的值转换为字符串表示形式。分析是格式设置的反向操作。分析操作根据数据类型的字符串表示形式创建该数据类型的实例。有关将字符串转换成其他数据类型的信息，请参见 “分析字符串”。

.NET 提供了丰富的格式设置支持，使得开发人员可以满足这些要求。

.NET 中的格式设置

格式设置的基本机制是 Object.ToString 方法的默认实现。有关信息，请参阅本主题稍后将介绍的使用 ToString 方法的默认格式设置部分。不过，.NET 提供了几种方法来修改和扩展其默认格式设置支持。其中包括：

- 重写 Object.ToString 方法以定义对象值的自定义字符串表示形式。有关详细信息，请参见本主题后面的重写 ToString 方法部分。
- 定义格式说明符，格式说明符允许对象值的字符串表示形式采用多种形式。例如，以下语句中的“X”格式说明符将整数转换为十六进制值的字符串表示形式。

```
int integerValue = 60312;
Console.WriteLine(integerValue.ToString("X"));   // Displays EB98.
```

```
Dim integerValue As Integer = 60312
Console.WriteLine(integerValue.ToString("X"))   ' Displays EB98.
```

  有关格式说明符的更多信息，请参见 ToString 方法和格式字符串 部分。

- 使用格式提供程序以利用特定区域性的格式设置约定。例如，以下语句通过使用 en-US 区域性的格式设置约定来显示货币值。

```
double cost = 1632.54;
Console.WriteLine(cost.ToString("C",
                  new System.Globalization.CultureInfo("en-US")));
// The example displays the following output:
//       $1,632.54
```

```
Dim cost As Double = 1632.54
Console.WriteLine(cost.ToString("C", New System.Globalization.CultureInfo("en-US")))
' The example displays the following output:
'       $1,632.54
```

有关使用格式提供程序进行格式设置的详细信息，请参阅格式提供程序部分。

- 实现 IFormattable 接口可以支持使用 Convert 类的字符串转换以及复合格式设置。有关更多信息，请参见 IFormattable 接口 部分。
- 使用复合格式设置来嵌入较大字符串中值的字符串表示形式。有关更多信息，请参见 复合格式设置 部分。
- 实现 ICustomFormatter 和 IFormatProvider 可以提供完全自定义的格式设置解决方案。有关更多信息，请参见 使用 ICustomFormatter 进行自定义格式设置 部分。

以下各部分分别使用这些方法来将对象转换为其字符串表示形式。

使用 ToString 方法的默认格式设置

每个从 System.Object 派生的类型都自动继承无参数的 `ToString` 方法，该方法在默认情况下返回类型的名称。下面的示例演示默认 `ToString` 方法。它定义一个名为 `Automobile` 、不具有实现的类。当对该类进行实例化并调用其 `ToString` 方法时，它显示其类型名称。请注意，此示例中未显式调用 `ToString` 方法。Console.WriteLine(Object) 方法隐式调用作为参数传递给它的对象的 `ToString` 方法。

```
using System;

public class Automobile
{
   // No implementation. All members are inherited from Object.
}

public class Example
{
   public static void Main()
   {
      Automobile firstAuto = new Automobile();
      Console.WriteLine(firstAuto);
   }
}
// The example displays the following output:
//       Automobile
```

```
Public Class Automobile
    ' No implementation. All members are inherited from Object.
End Class

Module Example
    Public Sub Main()
        Dim firstAuto As New Automobile()
        Console.WriteLine(firstAuto)
    End Sub
End Module
' The example displays the following output:
'       Automobile
```

WARNING

自 Windows 8.1 起，Windows 运行时包括具有单个方法 IStringable.ToString 的 IStringable 接口，用于提供默认格式支持。但是，我们建议托管类型不实现 `IStringable` 接口。有关详细信息，请参阅 Object.ToString 参考页上的“Windows 运行时和 `IStringable` 接口”部分。

由于除接口以外的所有类型都派生自 Object，因此会向自定义类或结构自动提供此功能。但是，默认的 `ToString` 方法提供的功能限于：尽管它标识类型，但无法提供有关类型实例的任何信息。若要提供可提供该对象相关信息的对象的字符串表示形式，必须重写 `ToString` 方法。

NOTE

结构继承自 ValueType，而后者又派生自 Object。虽然 ValueType 会重写 Object.ToString，但是其实现是相同的。

重写 ToString 方法

显示类型的名称这一用法往往有限，它不允许类型使用者区分实例。但是，你可以重写 `ToString` 方法，以提供更有用的对象值表示形式。下面的示例定义 `Temperature` 对象并重写其 `ToString` 方法，以便以摄氏度显示温度。

```
using System;

public class Temperature
{
   private decimal temp;

   public Temperature(decimal temperature)
   {
      this.temp = temperature;
   }

   public override string ToString()
   {
      return this.temp.ToString("N1") + "°C";
   }
}

public class Example
{
   public static void Main()
   {
      Temperature currentTemperature = new Temperature(23.6m);
      Console.WriteLine("The current temperature is " +
                        currentTemperature.ToString());
   }
}
// The example displays the following output:
//       The current temperature is 23.6°C.
```

```
Public Class Temperature
    Private temp As Decimal

    Public Sub New(temperature As Decimal)
        Me.temp = temperature
    End Sub

    Public Overrides Function ToString() As String
        Return Me.temp.ToString("N1") + "°C"
    End Function
End Class

Module Example
    Public Sub Main()
        Dim currentTemperature As New Temperature(23.6d)
        Console.WriteLine("The current temperature is " +
                          currentTemperature.ToString())
    End Sub
End Module
' The example displays the following output:
'       The current temperature is 23.6°C.
```

在 .NET 中，每个基元值类型的 `ToString` 方法已重写为显示对象值（而不是对象名称）。下表显示每种基元类型的重写。请注意，大多数重写方法调用 `ToString` 方法的另一个重载并向其传递用于定义其类型的一般格式的“G”格式说明符和表示当前区域性的 IFormatProvider 对象。

	TOSTRING
Boolean	返回 Boolean.TrueString 或 Boolean.FalseString。

类	TOSTRING 类
Byte	调用 `Byte.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Byte 值的格式。
Char	以字符串形式返回字符。
DateTime	调用 `DateTime.ToString("G", DatetimeFormatInfo.CurrentInfo)` 可以为当前区域性设置日期和时间值的格式。
Decimal	调用 `Decimal.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Decimal 值的格式。
Double	调用 `Double.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Double 值的格式。
Int16	调用 `Int16.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Int16 值的格式。
Int32	调用 `Int32.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Int32 值的格式。
Int64	调用 `Int64.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Int64 值的格式。
SByte	调用 `SByte.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 SByte 值的格式。
Single	调用 `Single.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 Single 值的格式。
UInt16	调用 `UInt16.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 UInt16 值的格式。
UInt32	调用 `UInt32.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 UInt32 值的格式。
UInt64	调用 `UInt64.ToString("G", NumberFormatInfo.CurrentInfo)` 可以为当前区域性设置 UInt64 值的格式。

ToString 方法和格式字符串

对象具有单一字符串表示形式时，可以依赖于默认 `ToString` 方法或重写 `ToString` 。但是，对象的值通常具有

多种表示形式。例如，温度可以用华氏度、摄氏度或开氏度来表示。同样，整数值 10 可以表示为多种形式，包括 10、10.0、1.0e01 或 $10.00。

为了允许单个值具有多种字符串表示形式，.NET 使用格式字符串。格式字符串是包含一个或多个预定义格式说明符的字符串，这些格式说明符是单一字符或字符组，用于定义 `ToString` 方法应如何设置其输出格式。然后将格式字符串作为参数传递给对象的 `ToString` 方法，并确定应如何显示该对象值的字符串表示形式。

.NET 中的所有数字类型、日期和时间类型以及枚举类型都支持一组预定义的格式说明符。还可以使用格式字符串定义你应用程序所定义的数据类型的多种字符串表示形式。

标准格式字符串

标准格式字符串包含单个格式说明符，该格式说明符是一个字母字符，用于定义应用该格式说明符的对象的字符串表示形式，此外，它还包含一个可选的精度说明符，该精度说明符影响在结果字符串中显示的位数。如果省略或不支持精度说明符，则标准格式说明符等效于标准格式字符串。

.NET 为所有数字类型、所有日期和时间类型以及所有枚举类型定义一组标准格式说明符。例如，这些类别中的每一类别都支持“G”标准格式说明符，该标准格式说明符定义该类型的值的一般字符串表示形式。

枚举类型的标准格式字符串直接控制值的字符串表示形式。传递给枚举值的 `ToString` 方法的格式字符串决定是使用其字符串名称（“G”和“F”格式说明符）、基础整数值（“D”格式说明符）还是十六进制值（“X”格式说明符）来显示值。下面的示例演示如何使用标准格式字符串来设置 DayOfWeek 枚举值的格式。

```
DayOfWeek thisDay = DayOfWeek.Monday;
string[] formatStrings = {"G", "F", "D", "X"};

foreach (string formatString in formatStrings)
   Console.WriteLine(thisDay.ToString(formatString));
// The example displays the following output:
//       Monday
//       Monday
//       1
//       00000001
```

```
Dim thisDay As DayOfWeek = DayOfWeek.Monday
Dim formatStrings() As String = {"G", "F", "D", "X"}

For Each formatString As String In formatStrings
    Console.WriteLine(thisDay.ToString(formatString))
Next
' The example displays the following output:
'       Monday
'       Monday
'       1
'       00000001
```

有关枚举格式字符串的信息，请参见 Enumeration Format Strings。

数字类型的标准格式字符串通常定义一个结果字符串，该结果字符串的确切显示由一个或多个属性值控制。例如，“C”格式说明符会将数字的格式设置为货币值。调用 `ToString` 方法并使用“C”格式说明符作为唯一参数时，来自当前区域性的 NumberFormatInfo 对象的以下属性值用于定义数字值的字符串表示形式：

- CurrencySymbol 属性，指定当前区域性的货币符号。
- CurrencyNegativePattern 或 CurrencyPositivePattern 属性，其返回的整数决定：
  - 货币符号的位置。
  - 负值由前导负号、尾随负号还是括号来表示。
  - 在数字值和货币符号之间是否有空格。

- CurrencyDecimalDigits 属性，定义结果字符串中的小数位数。
- CurrencyDecimalSeparator 属性，定义结果字符串中的小数分隔符符号。
- CurrencyGroupSeparator 属性，定义组分隔符符号。
- CurrencyGroupSizes 属性，定义小数点左边每个组的数字位数。
- NegativeSign 属性，确定在未使用括号表示负值时结果字符串中使用的负号。

此外，数字格式字符串可以包含一个精度说明符。该说明符的含义取决于与其一起使用的格式字符串，但是，它通常指示应在结果字符串中显示的总位数或小数位数。例如，下面的示例使用“X4”标准数字字符串和精度说明符来创建具有四个十六进制位的字符串值。

```
byte[] byteValues = { 12, 163, 255 };
foreach (byte byteValue in byteValues)
   Console.WriteLine(byteValue.ToString("X4"));
// The example displays the following output:
//       000C
//       00A3
//       00FF
```

```
Dim byteValues() As Byte = {12, 163, 255}
For Each byteValue As Byte In byteValues
    Console.WriteLine(byteValue.ToString("X4"))
Next
' The example displays the following output:
'       000C
'       00A3
'       00FF
```

有关标准数字格式字符串的更多信息，请参见 Standard Numeric Format Strings。

日期和时间值的标准格式字符串是由特定 DateTimeFormatInfo 属性存储的自定义格式字符串的别名。例如，如果使用“D”格式说明符调用日期和时间值的 `ToString` 方法，则使用当前区域性的 DateTimeFormatInfo.LongDatePattern 属性中存储的自定义格式字符串来显示日期和时间。（若要详细了解自定义格式字符串，请参阅下一部分。）下面的示例阐释了此关系。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime date1 = new DateTime(2009, 6, 30);
      Console.WriteLine("D Format Specifier:     {0:D}", date1);
      string longPattern = CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern;
      Console.WriteLine("'{0}' custom format string:     {1}",
                        longPattern, date1.ToString(longPattern));
   }
}
// The example displays the following output when run on a system whose
// current culture is en-US:
//    D Format Specifier:     Tuesday, June 30, 2009
//    'dddd, MMMM dd, yyyy' custom format string:     Tuesday, June 30, 2009
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim date1 As Date = #6/30/2009#
        Console.WriteLine("D Format Specifier:     {0:D}", date1)
        Dim longPattern As String = CultureInfo.CurrentCulture.DateTimeFormat.LongDatePattern
        Console.WriteLine("'{0}' custom format string:     {1}", _
                          longPattern, date1.ToString(longPattern))
    End Sub
End Module
' The example displays the following output when run on a system whose
' current culture is en-US:
'    D Format Specifier:     Tuesday, June 30, 2009
'    'dddd, MMMM dd, yyyy' custom format string:     Tuesday, June 30, 2009
```

有关标准日期和时间格式字符串的更多信息，请参见“标准日期和时间格式字符串”。

还可以使用标准格式字符串来定义应用程序所定义的对象的字符串表示形式，它由对象的 `ToString(String)` 方法生成。可以定义对象支持的特定标准格式说明符，还可以决定这些格式说明符是否区分大小写。`ToString(String)` 方法的实现应支持下列各项：

- 一个“G”格式说明符，表示对象的常用或通用格式。对象的 `ToString` 方法的无参数重载应调用其 `ToString(String)` 重载，并向其传递“G”标准格式字符串。
- 支持等于空引用（在 Visual Basic 中为 `Nothing` ）的格式说明符。应视等于空引用的格式说明符与“G”格式说明符等效。

例如，`Temperature` 类可以用摄氏度在内部存储温度，并使用格式限定符以摄氏度、华氏度和开氏度表示 `Temperature` 对象的值。下面的示例进行了这方面的演示。

```
using System;

public class Temperature
{
   private decimal m_Temp;

   public Temperature(decimal temperature)
   {
      this.m_Temp = temperature;
   }

   public decimal Celsius
   {
      get { return this.m_Temp; }
   }

   public decimal Kelvin
   {
      get { return this.m_Temp + 273.15m; }
   }

   public decimal Fahrenheit
   {
      get { return Math.Round(((decimal) (this.m_Temp * 9 / 5 + 32)), 2); }
   }

   public override string ToString()
   {
      return this.ToString("C");
   }

   public string ToString(string format)
   {
```

```
    {
       // Handle null or empty string.
       if (String.IsNullOrEmpty(format)) format = "C";
       // Remove spaces and convert to uppercase.
       format = format.Trim().ToUpperInvariant();

       // Convert temperature to Fahrenheit and return string.
       switch (format)
       {
          // Convert temperature to Fahrenheit and return string.
          case "F":
             return this.Fahrenheit.ToString("N2") + " °F";
          // Convert temperature to Kelvin and return string.
          case "K":
             return this.Kelvin.ToString("N2") + " K";
          // return temperature in Celsius.
          case "G":
          case "C":
             return this.Celsius.ToString("N2") + " °C";
          default:
             throw new FormatException(String.Format("The '{0}' format string is not supported.", format));
       }
    }
}

public class Example
{
   public static void Main()
   {
      Temperature temp1 = new Temperature(0m);
      Console.WriteLine(temp1.ToString());
      Console.WriteLine(temp1.ToString("G"));
      Console.WriteLine(temp1.ToString("C"));
      Console.WriteLine(temp1.ToString("F"));
      Console.WriteLine(temp1.ToString("K"));

      Temperature temp2 = new Temperature(-40m);
      Console.WriteLine(temp2.ToString());
      Console.WriteLine(temp2.ToString("G"));
      Console.WriteLine(temp2.ToString("C"));
      Console.WriteLine(temp2.ToString("F"));
      Console.WriteLine(temp2.ToString("K"));

      Temperature temp3 = new Temperature(16m);
      Console.WriteLine(temp3.ToString());
      Console.WriteLine(temp3.ToString("G"));
      Console.WriteLine(temp3.ToString("C"));
      Console.WriteLine(temp3.ToString("F"));
      Console.WriteLine(temp3.ToString("K"));

      Console.WriteLine(String.Format("The temperature is now {0:F}.", temp3));
   }
}
// The example displays the following output:
//       0.00 °C
//       0.00 °C
//       0.00 °C
//       32.00 °F
//       273.15 K
//       -40.00 °C
//       -40.00 °C
//       -40.00 °C
//       -40.00 °F
//       233.15 K
//       16.00 °C
//       16.00 °C
//       16.00 °C
//       60.80 °F
//       289.15 K
//       The temperature is now 16.00 °C.
```

```
//       The temperature is now 10.00 °C.
```

```
Public Class Temperature
    Private m_Temp As Decimal

    Public Sub New(temperature As Decimal)
        Me.m_Temp = temperature
    End Sub

    Public ReadOnly Property Celsius() As Decimal
        Get
            Return Me.m_Temp
        End Get
    End Property

    Public ReadOnly Property Kelvin() As Decimal
        Get
            Return Me.m_Temp + 273.15d
        End Get
    End Property

    Public ReadOnly Property Fahrenheit() As Decimal
        Get
            Return Math.Round(CDec(Me.m_Temp * 9 / 5 + 32), 2)
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return Me.ToString("C")
    End Function

    Public Overloads Function ToString(format As String) As String
        ' Handle null or empty string.
        If String.IsNullOrEmpty(format) Then format = "C"
        ' Remove spaces and convert to uppercase.
        format = format.Trim().ToUpperInvariant()

        Select Case format
            Case "F"
                ' Convert temperature to Fahrenheit and return string.
                Return Me.Fahrenheit.ToString("N2") & " °F"
            Case "K"
                ' Convert temperature to Kelvin and return string.
                Return Me.Kelvin.ToString("N2") & " K"
            Case "C", "G"
                ' Return temperature in Celsius.
                Return Me.Celsius.ToString("N2") & " °C"
            Case Else
                Throw New FormatException(String.Format("The '{0}' format string is not supported.",
format))
        End Select
    End Function
End Class

Public Module Example
    Public Sub Main()
        Dim temp1 As New Temperature(0d)
        Console.WriteLine(temp1.ToString())
        Console.WriteLine(temp1.ToString("G"))
        Console.WriteLine(temp1.ToString("C"))
        Console.WriteLine(temp1.ToString("F"))
        Console.WriteLine(temp1.ToString("K"))

        Dim temp2 As New Temperature(-40d)
        Console.WriteLine(temp2.ToString())
        Console.WriteLine(temp2.ToString("G"))
        Console.WriteLine(temp2.ToString("C"))
        Console.WriteLine(temp2.ToString("F"))
```

```
        Console.WriteLine(temp2.ToString("K"))

        Dim temp3 As New Temperature(16d)
        Console.WriteLine(temp3.ToString())
        Console.WriteLine(temp3.ToString("G"))
        Console.WriteLine(temp3.ToString("C"))
        Console.WriteLine(temp3.ToString("F"))
        Console.WriteLine(temp3.ToString("K"))

        Console.WriteLine(String.Format("The temperature is now {0:F}.", temp3))
    End Sub
End Module
' The example displays the following output:
'       0.00 °C
'       0.00 °C
'       0.00 °C
'       32.00 °F
'       273.15 K
'       -40.00 °C
'       -40.00 °C
'       -40.00 °C
'       -40.00 °F
'       233.15 K
'       16.00 °C
'       16.00 °C
'       16.00 °C
'       60.80 °F
'       289.15 K
'       The temperature is now 16.00 °C.
```

自定义格式字符串

除了标准格式字符串之外，.NET 还为数字值以及日期和时间值定义了自定义格式字符串。自定义格式字符串由定义值的字符串表示形式的一个或多个自定义格式说明符组成。例如，对于 en-US 区域性，自定义日期和时间格式字符串“yyyy/mm/dd hh:mm:ss.ffff t zzz”将日期转换为“2008/11/15 07:45:00.0000 P -08:00”形式的字符串表示形式。同样，自定义格式字符串“0000”将整数值 12 转换为“0012”。有关自定义格式字符串的完整列表，请参见 “自定义日期和时间格式字符串” 和 “自定义数字格式字符串”。

如果格式字符串仅包含一个自定义格式说明符，则此格式说明符前面应带有百分比 (%) 符号，以免与标准格式说明符混淆。下面的示例使用“M”自定义格式说明符来显示特定日期的一位数或两位数的月份。

```
DateTime date1 = new DateTime(2009, 9, 8);
Console.WriteLine(date1.ToString("%M"));       // Displays 9
```

```
Dim date1 As Date = #09/08/2009#
Console.WriteLine(date1.ToString("%M"))      ' Displays 9
```

日期和时间值的许多标准格式字符串均是由 DateTimeFormatInfo 对象的属性所定义的自定义格式字符串的别名。自定义格式字符串还为设置数字值或日期和时间值的应用程序定义格式提供了很大的灵活性。你可以通过将多个自定义格式说明符组合成一个自定义格式字符串来为数字值以及日期和时间值定义你自己的自定义结果字符串。下面的示例定义一个自定义格式字符串，该字符串在月份名称、日期和年份后的括号中显示星期几。

```
string customFormat = "MMMM dd, yyyy (dddd)";
DateTime date1 = new DateTime(2009, 8, 28);
Console.WriteLine(date1.ToString(customFormat));
// The example displays the following output if run on a system
// whose language is English:
//       August 28, 2009 (Friday)
```

```
Dim customFormat As String = "MMMM dd, yyyy (dddd)"
Dim date1 As Date = #8/28/2009#
Console.WriteLine(date1.ToString(customFormat))
' The example displays the following output if run on a system
' whose language is English:
'       August 28, 2009 (Friday)
```

以下示例定义了自定义格式字符串，其中 Int64 值显示为标准的美国七位数电话号码及其区号。

```
using System;

public class Example
{
   public static void Main()
   {
      long number = 8009999999;
      string fmt = "000-000-0000";
      Console.WriteLine(number.ToString(fmt));
   }
}
// The example displays the following output:
//        800-999-9999
```

```
Module Example
    Public Sub Main()
        Dim number As Long = 8009999999
        Dim fmt As String = "000-000-0000"
        Console.WriteLine(number.ToString(fmt))
    End Sub
End Module
' The example displays the following output:

' The example displays the following output:
'       800-999-9999
```

尽管标准格式字符串一般可以满足应用程序定义的类型的大多数格式设置需求，但你还可以定义自定义格式说明符来设置类型的格式。

格式字符串和 .NET 类型

所有数字类型（即 Byte、Decimal、Double、Int16、Int32、Int64、SByte、Single、UInt16、UInt32、UInt64 和 BigInteger）以及 DateTime、DateTimeOffset、TimeSpan、Guid 和所有枚举类型都支持使用格式字符串设置格式。有关各类型支持的特定格式字符串的信息，请参见下列主题：

TITLE	tt
Standard Numeric Format Strings	描述用于创建数字值的常用字符串表示形式的标准格式字符串。
Custom Numeric Format Strings	描述用于创建数字值的应用程序特定格式的自定义格式字符串。
标准日期和时间格式字符串	介绍了创建 DateTime 和 DateTimeOffset 值的常用字符串表示形式的标准格式字符串。
自定义日期和时间格式字符串	介绍了创建 DateTime 和 DateTimeOffset 值的应用专用格式的自定义格式字符串。

TITLE	此
标准 TimeSpan 格式字符串	描述用于创建时间间隔的常用字符串表示形式的标准格式字符串。
自定义 TimeSpan 格式字符串	描述用于创建时间间隔的应用程序特定格式的自定义格式字符串。
Enumeration Format Strings	描述用于创建枚举值的字符串表示形式的标准格式字符串。
Guid.ToString(String)	描述 Guid 值的标准格式字符串。

使用格式提供程序进行区分区域性的格式设置

尽管格式说明符允许你自定义对象的格式设置，但是生成有意义的对象字符串表示形式通常需要附加格式设置信息。例如，通过使用“C”标准格式字符串或自定义格式字符串（如“$ #,#.00”）来将数字格式设置为货币值至少需要提供有关正确的货币符号、组分隔符和小数点分隔符的信息，以便包括在带有格式的字符串中。在 .NET 中，此附加格式设置信息通过 IFormatProvider 接口提供，具体是以参数形式提供给数字类型以及日期和时间类型的 `ToString` 方法的一个或多个重载。IFormatProvider 实现在 .NET 中用于支持区域性专用格式设置。下面的示例演示在使用三个代表不同区域的 IFormatProvider 对象设置某个对象的格式时，该对象的字符串表示形式将如何变化。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      decimal value = 1603.42m;
      Console.WriteLine(value.ToString("C3", new CultureInfo("en-US")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("fr-FR")));
      Console.WriteLine(value.ToString("C3", new CultureInfo("de-DE")));
   }
}
// The example displays the following output:
//       $1,603.420
//       1 603,420 €
//       1.603,420 €
```

```
Imports System.Globalization

Public Module Example
    Public Sub Main()
        Dim value As Decimal = 1603.42d
        Console.WriteLine(value.ToString("C3", New CultureInfo("en-US")))
        Console.WriteLine(value.ToString("C3", New CultureInfo("fr-FR")))
        Console.WriteLine(value.ToString("C3", New CultureInfo("de-DE")))
    End Sub
End Module
' The example displays the following output:
'       $1,603.420
'       1 603,420 €
'       1.603,420 €
```

IFormatProvider 接口包含一个 GetFormat(Type)方法，该方法只有一个参数，该参数指定提供格式设置信息的对象类型。如果该方法可以提供该类型的对象，则返回它。否则，它返回空引用（在 Visual Basic 中为 `Nothing` ）。

IFormatProvider.GetFormat 为回调方法。调用包含 `ToString` 参数的 IFormatProvider 方法重载时，它调用该 GetFormat 对象的 IFormatProvider 方法。GetFormat 方法负责将提供所需格式设置信息（就像 `formatType` 参数指定的一样）的对象返回给 `ToString` 方法。

一些格式设置或字符串转换方法包含 IFormatProvider类型的参数，但是很多情况下在调用该方法时将忽略该参数的值。下表列出了使用 Type 对象的参数和类型的一些格式设置方法，该对象传递给 IFormatProvider.GetFormat 方法。

数字类型的 `ToString` 方法	System.Globalization.NumberFormatInfo
日期和时间类型的 `ToString` 方法	System.Globalization.DateTimeFormatInfo
String.Format	System.ICustomFormatter
StringBuilder.AppendFormat	System.ICustomFormatter

NOTE

将重载数字类型以及日期和时间类型的 `ToString` 方法，并且只有某些重载包含 IFormatProvider 参数。如果方法没有 IFormatProvider类型的参数，则改为传递 CultureInfo.CurrentCulture 属性所返回的对象。例如，对默认 Int32.ToString() 方法的调用最终将导致诸如以下的方法调用：
`Int32.ToString("G", System.Globalization.CultureInfo.CurrentCulture)` 。

.NET 提供了三个实现 IFormatProvider 的类：

- DateTimeFormatInfo类，提供特定区域性的日期和时间值的格式设置信息。其 IFormatProvider.GetFormat 实现返回它自身的实例。
- NumberFormatInfo类，提供特定区域性的数字格式设置信息。其 IFormatProvider.GetFormat 实现返回它自身的实例。
- CultureInfo。其 IFormatProvider.GetFormat 实现可以返回一个 NumberFormatInfo 对象（可提供数字格式设置信息）或一个 DateTimeFormatInfo 对象（可提供日期和时间值的格式设置信息）。

你还可以实现自己的格式提供程序来替换上述任意一个类。但是，如果你的实现的 GetFormat 方法必须向 `ToString` 方法提供格式设置信息，则它必须返回上表中列出的相应类型的对象。

数值的区分区域性的格式设置

默认情况下，数值的格式设置是区分区域性的。如果在调用格式设置方法时不指定区域性，则将使用当前线程区域性的格式设置约定。下面的示例演示了这一点，其中对当前线程区域性进行了四次更改，随后调用了 Decimal.ToString(String) 方法。每次更改后，结果字符串均反映当前区域性的格式设置约定。这是因为 `ToString` 和 `ToString(String)` 方法会包装对每个数值类型的 `ToString(String, IFormatProvider)` 方法的调用。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      Decimal value = 1043.17m;

      foreach (var cultureName in cultureNames) {
         // Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name);
         Console.WriteLine(value.ToString("C2"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       $1,043.17
//
//       The current culture is fr-FR
//       1 043,17 €
//
//       The current culture is es-MX
//       $1,043.17
//
//       The current culture is de-DE
//       1.043,17 €
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-US", "fr-FR", "es-MX", "de-DE"}
        Dim value As Decimal = 1043.17d

        For Each cultureName In cultureNames
            ' Change the current thread culture.
            Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName)
            Console.WriteLine("The current culture is {0}",
                              Thread.CurrentThread.CurrentCulture.Name)
            Console.WriteLine(value.ToString("C2"))
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       The current culture is en-US
'       $1,043.17
'
'       The current culture is fr-FR
'       1 043,17 €
'
'       The current culture is es-MX
'       $1,043.17
'
'       The current culture is de-DE
'       1.043,17 €
```

你还可以设置特定区域性数值的格式，方法是调用具有 `ToString` 参数的 `provider` 重载，并将其作为以下对象之一的参数进行传递：

- 一个 CultureInfo 对象，此对象代表要使用其格式设置约定的区域性。它的 CultureInfo.GetFormat 方法会返回 CultureInfo.NumberFormat 属性的值，即提供数字区域性特定格式设置信息的 NumberFormatInfo 对象。
- 一个 NumberFormatInfo 对象，此对象用于定义要使用的区域性特定格式设置约定。它的 GetFormat 方法会返回它自身的一个实例。

下面的示例使用了表示英语(美国)和英语(英国)区域性以及法语和俄罗斯语非特定区域性的 NumberFormatInfo 对象，来设置浮点数字的格式。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      Double value = 1043.62957;
      string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

      foreach (var name in cultureNames) {
         NumberFormatInfo nfi = CultureInfo.CreateSpecificCulture(name).NumberFormat;
         Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3", nfi));
      }
   }
}
// The example displays the following output:
//       en-US: 1,043.630
//       en-GB: 1,043.630
//       ru:    1 043,630
//       fr:    1 043,630
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim value As Double = 1043.62957
        Dim cultureNames() As String = {"en-US", "en-GB", "ru", "fr"}

        For Each name In cultureNames
            Dim nfi As NumberFormatInfo = CultureInfo.CreateSpecificCulture(name).NumberFormat
            Console.WriteLine("{0,-6} {1}", name + ":", value.ToString("N3", nfi))
        Next
    End Sub
End Module
' The example displays the following output:
'       en-US: 1,043.630
'       en-GB: 1,043.630
'       ru:    1 043,630
'       fr:    1 043,630
```

日期和时间值的区分区域性的格式设置

默认情况下，日期和时间值的格式设置是区分区域性的。如果在调用格式设置方法时不指定区域性，则将使用当前线程区域性的格式设置约定。下面的示例演示了这一点，其中对当前线程区域性进行了四次更改，随后调用了 DateTime.ToString(String) 方法。每次更改后，结果字符串均反映当前区域性的格式设置约定。这是因为 DateTime.ToString()、DateTime.ToString(String)、DateTimeOffset.ToString()和 DateTimeOffset.ToString(String) 方法会包装对 DateTime.ToString(String, IFormatProvider) 及 DateTimeOffset.ToString(String, IFormatProvider)

方法的调用。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "es-MX", "de-DE" };
      DateTime dateToFormat = new DateTime(2012, 5, 28, 11, 30, 0);

      foreach (var cultureName in cultureNames) {
         // Change the current thread culture.
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);
         Console.WriteLine("The current culture is {0}",
                           Thread.CurrentThread.CurrentCulture.Name);
         Console.WriteLine(dateToFormat.ToString("F"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The current culture is en-US
//       Monday, May 28, 2012 11:30:00 AM
//
//       The current culture is fr-FR
//       lundi 28 mai 2012 11:30:00
//
//       The current culture is es-MX
//       lunes, 28 de mayo de 2012 11:30:00 a.m.
//
//       The current culture is de-DE
//       Montag, 28. Mai 2012 11:30:00
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-US", "fr-FR", "es-MX", "de-DE"}
        Dim dateToFormat As Date = #5/28/2012 11:30AM#

        For Each cultureName In cultureNames
            ' Change the current thread culture.
            Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName)
            Console.WriteLine("The current culture is {0}",
                              Thread.CurrentThread.CurrentCulture.Name)
            Console.WriteLine(dateToFormat.ToString("F"))
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       The current culture is en-US
'       Monday, May 28, 2012 11:30:00 AM
'
'       The current culture is fr-FR
'       lundi 28 mai 2012 11:30:00
'
'       The current culture is es-MX
'       lunes, 28 de mayo de 2012 11:30:00 a.m.
'
'       The current culture is de-DE
'       Montag, 28. Mai 2012 11:30:00
```

你还可以设置特定区域性日期和时间值的格式，方法是调用具有 DateTime.ToString 参数的 DateTimeOffset.ToString 或 `provider` 重载，并将其作为以下对象之一的参数进行传递：

- 一个 CultureInfo 对象，此对象代表要使用其格式设置约定的区域性。它的 CultureInfo.GetFormat 方法会返回 CultureInfo.DateTimeFormat 属性的值，即提供日期和时间值区域性特定格式设置信息的 DateTimeFormatInfo 对象。
- 一个 DateTimeFormatInfo 对象，此对象用于定义要使用的区域性特定格式设置约定。它的 GetFormat 方法会返回它自身的一个实例。

下面的示例使用了表示英语(美国)和英语(英国)区域性以及法语和俄罗斯语非特定区域性的 DateTimeFormatInfo 对象，来设置日期的格式。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dat1 = new DateTime(2012, 5, 28, 11, 30, 0);
      string[] cultureNames = { "en-US", "en-GB", "ru", "fr" };

      foreach (var name in cultureNames) {
         DateTimeFormatInfo dtfi = CultureInfo.CreateSpecificCulture(name).DateTimeFormat;
         Console.WriteLine("{0}: {1}", name, dat1.ToString(dtfi));
      }
   }
}
// The example displays the following output:
//       en-US: 5/28/2012 11:30:00 AM
//       en-GB: 28/05/2012 11:30:00
//       ru: 28.05.2012 11:30:00
//       fr: 28/05/2012 11:30:00
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim dat1 As Date = #5/28/2012 11:30AM#
        Dim cultureNames() As String = {"en-US", "en-GB", "ru", "fr"}

        For Each name In cultureNames
            Dim dtfi As DateTimeFormatInfo = CultureInfo.CreateSpecificCulture(name).DateTimeFormat
            Console.WriteLine("{0}: {1}", name, dat1.ToString(dtfi))
        Next
    End Sub
End Module
' The example displays the following output:
'       en-US: 5/28/2012 11:30:00 AM
'       en-GB: 28/05/2012 11:30:00
'       ru: 28.05.2012 11:30:00
'       fr: 28/05/2012 11:30:00
```

IFormattable 接口

通常，使用格式字符串和一个 `ToString` 参数来重载 IFormatProvider 方法的类型还实现 IFormattable 接口。此接口具有一个成员 IFormattable.ToString(String, IFormatProvider)，该成员同时将格式字符串和格式提供程序作为参数。

对应用程序定义的类实现 IFormattable 接口具有两大优势：

- 支持使用 Convert 类进行字符串转换。对 Convert.ToString(Object) 和 Convert.ToString(Object, IFormatProvider) 方法的调用会自动调用 IFormattable 实现。
- 支持复合格式设置。如果使用包含格式字符串的格式项设置自定义类型的格式，则公共语言运行时自动调用 IFormattable 实现，并向其传递该格式字符串。有关采用 String.Format 或 Console.WriteLine等方法进行复合格式设置的更多信息，请参见 复合格式设置 部分。

下面的示例定义一个实现 `Temperature` 接口的 IFormattable 类。它支持“C”或“G”格式说明符（用于以摄氏度显示温度）、“F”格式说明符（用于以华氏度显示温度）和“K”格式说明符（用于以开氏度显示温度）。

```
using System;
using System.Globalization;

public class Temperature : IFormattable
{
   private decimal m_Temp;

   public Temperature(decimal temperature)
   {
      this.m_Temp = temperature;
   }

   public decimal Celsius
   {
      get { return this.m_Temp; }
   }

   public decimal Kelvin
   {
      get { return this.m_Temp + 273.15m; }
   }

   public decimal Fahrenheit
   {
      get { return Math.Round((decimal) this.m_Temp * 9 / 5 + 32, 2); }
   }

   public override string ToString()
   {
      return this.ToString("G", null);
   }

   public string ToString(string format)
   {
      return this.ToString(format, null);
   }

   public string ToString(string format, IFormatProvider provider)
   {
      // Handle null or empty arguments.
      if (String.IsNullOrEmpty(format))
         format = "G";
      // Remove any white space and covert to uppercase.
      format = format.Trim().ToUpperInvariant();

      if (provider == null)
         provider = NumberFormatInfo.CurrentInfo;

      switch (format)
      {
         // Convert temperature to Fahrenheit and return string.
         case "F":
            return this.Fahrenheit.ToString("N2", provider) + "°F";
         // Convert temperature to Kelvin and return string.
         case "K":
            return this.Kelvin.ToString("N2", provider) + "K";
         // Return temperature in Celsius.
         case "C":
         case "G":
            return this.Celsius.ToString("N2", provider) + "°C";
         default:
            throw new FormatException(String.Format("The '{0}' format string is not supported.", format));
      }
   }
}
```

```
Imports System.Globalization

Public Class Temperature : Implements IFormattable
    Private m_Temp As Decimal

    Public Sub New(temperature As Decimal)
        Me.m_Temp = temperature
    End Sub

    Public ReadOnly Property Celsius() As Decimal
        Get
            Return Me.m_Temp
        End Get
    End Property

    Public ReadOnly Property Kelvin() As Decimal
        Get
            Return Me.m_Temp + 273.15d
        End Get
    End Property

    Public ReadOnly Property Fahrenheit() As Decimal
        Get
            Return Math.Round(CDec(Me.m_Temp * 9 / 5 + 32), 2)
        End Get
    End Property

    Public Overrides Function ToString() As String
        Return Me.ToString("G", Nothing)
    End Function

    Public Overloads Function ToString(format As String) As String
        Return Me.ToString(format, Nothing)
    End Function

    Public Overloads Function ToString(format As String, provider As IFormatProvider) As String _
       Implements IFormattable.ToString

        ' Handle null or empty arguments.
        If String.IsNullOrEmpty(format) Then format = "G"
        ' Remove any white space and convert to uppercase.
        format = format.Trim().ToUpperInvariant()

        If provider Is Nothing Then provider = NumberFormatInfo.CurrentInfo

        Select Case format
         ' Convert temperature to Fahrenheit and return string.
            Case "F"
                Return Me.Fahrenheit.ToString("N2", provider) & "°F"
         ' Convert temperature to Kelvin and return string.
            Case "K"
                Return Me.Kelvin.ToString("N2", provider) & "K"
         ' Return temperature in Celsius.
            Case "C", "G"
                Return Me.Celsius.ToString("N2", provider) & "°C"
            Case Else
                Throw New FormatException(String.Format("The '{0}' format string is not supported.",
format))
        End Select
    End Function
End Class
```

下面的示例实例化一个 `Temperature` 对象。然后，它调用 ToString 方法，并使用多个复合格式字符串获取 `Temperature` 对象的不同字符串表示形式。其中每一个方法调用都依次调用 IFormattable 类的 `Temperature` 实现。

```
public class Example
{
   public static void Main()
   {
      CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US");
      Temperature temp = new Temperature(22m);
      Console.WriteLine(Convert.ToString(temp, new CultureInfo("ja-JP")));
      Console.WriteLine("Temperature: {0:K}", temp);
      Console.WriteLine("Temperature: {0:F}", temp);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"), "Temperature: {0:F}", temp));
   }
}
// The example displays the following output:
//       22.00°C
//       Temperature: 295.15K
//       Temperature: 71.60°F
//       Temperature: 71,60°F
```

```
Public Module Example
    Public Sub Main()
        Dim temp As New Temperature(22d)
        CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("en-US")
        Console.WriteLine(Convert.ToString(temp1, New CultureInfo("ja-JP")))
        Console.WriteLine("Temperature: {0:K}", temp)
        Console.WriteLine("Temperature: {0:F}", temp)
        Console.WriteLine(String.Format(New CultureInfo("fr-FR"), "Temperature: {0:F}", temp))
    End Sub
End Module
' The example displays the following output:
'       22.00°C
'       Temperature: 295.15K
'       Temperature: 71.60°F
'       Temperature: 71,60°F
```

复合格式设置

一些方法(如 String.Format 和 StringBuilder.AppendFormat)支持复合格式设置。复合格式字符串是一种模板，该模板返回合并了零个、一个或多个对象的字符串表示形式的单一字符串。每个对象均由复合格式字符串中的索引格式项表示。格式项的索引对应于格式项在方法的参数列表中所表示的对象位置。索引是从零开始的。例如，在以下对 String.Format 方法的调用中，第一个格式项 `{0:D}` 被 `thatDate` 的字符串表示形式；第二个格式项 `{1}` 被 `item1` 的字符串表示形式替换；第三个格式项 `{2:C2}` 被 `item1.Value` 的字符串表示形式替换。

```
result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.",
                       thatDate, item1, item1.Value);
Console.WriteLine(result);
// The example displays output like the following if run on a system
// whose current culture is en-US:
//       On 5/1/2009, the inventory of WidgetA was worth $107.44.
```

```
result = String.Format("On {0:d}, the inventory of {1} was worth {2:C2}.", _
                       thatDate, item1, item1.Value)
Console.WriteLine(result)
' The example displays output like the following if run on a system
' whose current culture is en-US:
'       On 5/1/2009, the inventory of WidgetA was worth $107.44.
```

除了将格式项替换为其相应对象的字符串表示形式之外，格式项还可让你控制：

- 将对象表示为字符串的特定方法(如果对象实现 IFormattable 接口并支持格式字符串)。为此，可在格式

项的索引后加上 `:`（冒号），后跟一个有效的格式字符串。前面的示例执行此操作的方式是：格式化带有“d”（短日期模式）格式字符串（例如，`{0:d}`）的日期值，并格式化带有“C2”格式字符串（例如，`{2:C2}`）的数值，将数量表示为具有两位小数位数的货币值。

- 包含对象的字符串表示形式的字段的宽度以及该字段中字符串表现形式的对齐方式。为此，可在格式项的索引后加上 `,`（逗号），后跟字段宽度。如果字段宽度为正值，则字段中的字符串为右对齐，如果字段宽度是负值，则为左对齐。在下面的示例中，在由 20 个字符组成的字段中的日期值左对齐，而在由 11 个字符组成的字段中，带有一位小数的十进制值右对齐。

```
DateTime startDate = new DateTime(2015, 8, 28, 6, 0, 0);
decimal[] temps = { 73.452m, 68.98m, 72.6m, 69.24563m,
                    74.1m, 72.156m, 72.228m };
Console.WriteLine("{0,-20} {1,11}\n", "Date", "Temperature");
for (int ctr = 0; ctr < temps.Length; ctr++)
   Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr), temps[ctr]);

// The example displays the following output:
//       Date                 Temperature
//
//       8/28/2015 6:00 AM           73.5
//       8/29/2015 6:00 AM           69.0
//       8/30/2015 6:00 AM           72.6
//       8/31/2015 6:00 AM           69.2
//       9/1/2015 6:00 AM            74.1
//       9/2/2015 6:00 AM            72.2
//       9/3/2015 6:00 AM            72.2
```

```
Dim startDate As New Date(2015, 8, 28, 6, 0, 0)
Dim temps() As Decimal = {73.452, 68.98, 72.6, 69.24563,
                          74.1, 72.156, 72.228}
Console.WriteLine("{0,-20} {1,11}", "Date", "Temperature")
Console.WriteLine()
For ctr As Integer = 0 To temps.Length - 1
    Console.WriteLine("{0,-20:g} {1,11:N1}", startDate.AddDays(ctr), temps(ctr))
Next
' The example displays the following output:
'       Date                 Temperature
'
'       8/28/2015 6:00 AM           73.5
'       8/29/2015 6:00 AM           69.0
'       8/30/2015 6:00 AM           72.6
'       8/31/2015 6:00 AM           69.2
'       9/1/2015 6:00 AM            74.1
'       9/2/2015 6:00 AM            72.2
'       9/3/2015 6:00 AM            72.2
```

请注意，如果对齐字符串组件和格式字符串组件均存在，则前者位于后者之前（例如，`{0,-20:g}`）。

有关复合格式的更多信息，请参见“复合格式设置”。

使用 ICustomFormatter 进行自定义格式设置

两种复合格式设置方法（String.Format(IFormatProvider, String, Object[]) 和 StringBuilder.AppendFormat(IFormatProvider, String, Object[])）包括一个支持自定义格式设置的格式提供程序。当调用其中一种格式设置方法时，该方法会将表示 Type 接口的 ICustomFormatter 对象传递到格式提供程序的 GetFormat 方法。GetFormat 方法然后负责返回提供自定义格式设置功能的 ICustomFormatter 实现。

ICustomFormatter 接口具有一个方法 Format(String, Object, IFormatProvider)，复合格式设置方法为复合格式字符串中的每一格式项自动调用一次该方法。Format(String, Object, IFormatProvider) 方法具有三个参数：一个格式字符串（表示格式项中的 `formatString` 参数）、一个要设置格式的对象和一个提供格式设置服务的

IFormatProvider 对象。通常，实现 ICustomFormatter 的类还会实现 IFormatProvider，因此上述最后一个参数是对自定义格式设置类自身的引用。该方法返回要设置格式的对象的带格式自定义字符串表示形式。如果该方法无法设置对象的格式，则应返回空引用（在 Visual Basic 中为 `Nothing`）。

下面的示例提供一个名为 ICustomFormatter 的 `ByteByByteFormatter` 实现，该实现将整数值显示为两位的十六进制值后跟一个空格的序列。

```
public class ByteByByteFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string format, object arg,
                        IFormatProvider formatProvider)
   {
      if (! formatProvider.Equals(this)) return null;

      // Handle only hexadecimal format string.
      if (! format.StartsWith("X")) return null;

      byte[] bytes;
      string output = null;

      // Handle only integral types.
      if (arg is Byte)
         bytes = BitConverter.GetBytes((Byte) arg);
      else if (arg is Int16)
         bytes = BitConverter.GetBytes((Int16) arg);
      else if (arg is Int32)
         bytes = BitConverter.GetBytes((Int32) arg);
      else if (arg is Int64)
         bytes = BitConverter.GetBytes((Int64) arg);
      else if (arg is SByte)
         bytes = BitConverter.GetBytes((SByte) arg);
      else if (arg is UInt16)
         bytes = BitConverter.GetBytes((UInt16) arg);
      else if (arg is UInt32)
         bytes = BitConverter.GetBytes((UInt32) arg);
      else if (arg is UInt64)
         bytes = BitConverter.GetBytes((UInt64) arg);
      else
         return null;

      for (int ctr = bytes.Length - 1; ctr >= 0; ctr--)
         output += String.Format("{0:X2} ", bytes[ctr]);

      return output.Trim();
   }
}
```

```
Public Class ByteByByteFormatter : Implements IFormatProvider, ICustomFormatter
    Public Function GetFormat(formatType As Type) As Object _
                    Implements IFormatProvider.GetFormat
        If formatType Is GetType(ICustomFormatter) Then
            Return Me
        Else
            Return Nothing
        End If
    End Function

    Public Function Format(fmt As String, arg As Object,
                           formatProvider As IFormatProvider) As String _
                           Implements ICustomFormatter.Format

        If Not formatProvider.Equals(Me) Then Return Nothing

        ' Handle only hexadecimal format string.
        If Not fmt.StartsWith("X") Then
            Return Nothing
        End If

        ' Handle only integral types.
        If Not typeof arg Is Byte AndAlso
           Not typeof arg Is Int16 AndAlso
           Not typeof arg Is Int32 AndAlso
           Not typeof arg Is Int64 AndAlso
           Not typeof arg Is SByte AndAlso
           Not typeof arg Is UInt16 AndAlso
           Not typeof arg Is UInt32 AndAlso
           Not typeof arg Is UInt64 Then _
              Return Nothing

        Dim bytes() As Byte = BitConverter.GetBytes(arg)
        Dim output As String = Nothing

        For ctr As Integer = bytes.Length - 1 To 0 Step -1
            output += String.Format("{0:X2} ", bytes(ctr))
        Next

        Return output.Trim()
    End Function
End Class
```

下面的示例使用 `ByteByByteFormatter` 类设置整数值的格式。请注意，此示例中未显式调用 ICustomFormatter.Format 方法调用中多次调用了 String.Format(IFormatProvider, String, Object[]) 方法，并在第三个方法调用中使用了默认 NumberFormatInfo 提供程序，这是因为 `ByteByByteFormatter.Format` 方法无法识别"N0"格式字符串并返回空引用（在 Visual Basic 中为 `Nothing` ）的格式说明符。

```
public class Example
{
   public static void Main()
   {
      long value = 3210662321;
      byte value1 = 214;
      byte value2 = 19;

      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X}", value));
      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0:X} And {1:X} = {2:X} ({2:000})",
                                      value1, value2, value1 & value2));
      Console.WriteLine(String.Format(new ByteByByteFormatter(), "{0,10:N0}", value));
   }
}
// The example displays the following output:
//       00 00 00 00 BF 5E D1 B1
//       00 D6 And 00 13 = 00 12 (018)
//       3,210,662,321
```

```
Public Module Example
    Public Sub Main()
        Dim value As Long = 3210662321
        Dim value1 As Byte = 214
        Dim value2 As Byte = 19

        Console.WriteLine((String.Format(New ByteByByteFormatter(), "{0:X}", value)))
        Console.WriteLine((String.Format(New ByteByByteFormatter(), "{0:X} And {1:X} = {2:X} ({2:000})",
                                         value1, value2, value1 And value2)))
        Console.WriteLine(String.Format(New ByteByByteFormatter(), "{0,10:N0}", value))
    End Sub
End Module
' The example displays the following output:
'       00 00 00 00 BF 5E D1 B1
'       00 D6 And 00 13 = 00 12 (018)
'       3,210,662,321
```

相关主题

TITLE
Standard Numeric Format Strings	描述用于创建数字值的常用字符串表示形式的标准格式字符串。
Custom Numeric Format Strings	描述用于创建数字值的应用程序特定格式的自定义格式字符串。
标准日期和时间格式字符串	描述用于创建 DateTime 值的常用字符串表示形式的标准格式字符串。
自定义日期和时间格式字符串	描述用于创建 DateTime 值的应用程序特定格式的自定义格式字符串。
标准 TimeSpan 格式字符串	描述用于创建时间间隔的常用字符串表示形式的标准格式字符串。
自定义 TimeSpan 格式字符串	描述用于创建时间间隔的应用程序特定格式的自定义格式字符串。

TITLE	𝐂
Enumeration Format Strings	描述用于创建枚举值的字符串表示形式的标准格式字符串。
复合格式设置	描述如何将一个或多个设置了格式的值嵌入字符串。然后该字符串可以显示在控制台上或被写至流。
分析字符串	描述如何将对象初始化为这些对象的字符串表示形式所描述的值。分析是格式化的反向操作。

参考

- System.IFormattable
- System.IFormatProvider
- System.ICustomFormatter

标准数字格式字符串

2021/11/16 •

标准数字格式字符串用于格式化通用数值类型。标准数字格式字符串采用 `Axx` 的形式，其中：

- `A` 是称为“格式说明符”的单个字母字符。任何包含一个以上字母字符（包括空白）的数字格式字符串都被解释为自定义数字格式字符串。有关更多信息，请参见自定义数字格式字符串。
- `xx` 是称为“精度说明符”的可选整数。精度说明符的范围从 0 到 99，并且影响结果中的位数。请注意，精度说明符控制数字的字符串表示形式中的数字个数。它不舍入该数字。若要执行舍入运算，请使用 Math.Ceiling、Math.Floor 或 Math.Round 方法。

  当精度说明符控制结果字符串中的小数位数时，结果字符串会反映一个数字，该数字四舍五入到最接近无限精确结果的可表示结果。如果有两个同样接近的可表示结果：

  - 在 .NET Framework 和.NET Core（.NET Core 2.0 及以下）上，运行时选择最低有效数字更高的结果（即使用 MidpointRounding.AwayFromZero）。
  - 在 .NET Core 2.1 及更高版本上，运行时选择最低有效数字为偶数的结果（即使用 MidpointRounding.ToEven）。

  > **NOTE**
  > 精度说明符确定结果字符串中的位数。若要使用前导或尾随空格填充结果字符串，请使用 复合格式设置功能，并在格式项中定义 *对齐组件*。

下列支持标准数字格式字符串：

- 所有数字类型的一些 `ToString` 方法重载。例如，可以向 Int32.ToString(String) 和 Int32.ToString(String, IFormatProvider) 方法提供数字格式字符串。
- .NET 复合格式功能，由 Console 和 StreamWriter 类的一些 `Write` 和 `WriteLine` 方法、String.Format 方法以及 StringBuilder.AppendFormat 方法使用。复合格式功能允许你将多个数据项的字符串表示形式包含在单个字符串中，以指定字段宽度，并在字段中对齐数字。有关更多信息，请参见复合格式设置。
- C# 和 Visual Basic 中的内插的字符串，与复合格式字符串相比，语法更简化。

> **TIP**
> 你可以下载格式设置实用工具，它属于一种 .NET Core Windows 窗体应用程序，通过该应用程序，你可将格式字符串应用于数值或日期和时间值并显示结果字符串。源代码适用于 C# 和 Visual Basic。

下表介绍标准数字格式说明符并显示每个格式说明符产生的示例输出。有关使用标准数字格式字符串的其他信息，请参见注释一节；有关使用方法的完整演示，请参见示例一节。

¤¤¤¤¤	“¤¤”	¤¤	¤¤

[illegible]	"[illegible]"	[illegible]	[illegible]
"C"或"c"	货币	结果:货币值。 受以下类型支持:所有数值类型。 精度说明符:十进制小数位数。 默认值精度说明符:由 NumberFormatInfo.CurrencyDecimalDigits 定义。 更多信息:货币("C")格式说明符。	123.456 ("C", en-US) -> \$123.46 123.456 ("C", fr-FR) -> 123,46 € 123.456 ("C", ja-JP) -> ¥123 -123.456 ("C3", en-US) -> (\$123.456) -123.456 ("C3", fr-FR) -> -123,456 € -123.456 ("C3", ja-JP) -> -¥123.456
"D"或"d"	十进制	结果:整型数字, 负号可选。 受以下类型支持:仅限整型类型。 精度说明符:数字位数下限。 默认值精度说明符:所需数字位数下限。 更多信息:十进制("D")格式说明符。	1234 ("D") -> 1234 -1234 ("D6") -> -001234
"E"或"e"	指数(科学型)	结果:指数表示法。 受以下类型支持:所有数值类型。 精度说明符:十进制小数位数。 默认值精度说明符:6. 更多信息:指数("E")格式说明符。	1052.0329112756 ("E", en-US) -> 1.052033E+003 1052.0329112756 ("e", fr-FR) -> 1,052033e+003 -1052.0329112756 ("e2", en-US) -> -1.05e+003 -1052.0329112756 ("E2", fr-FR) -> -1,05E+003

[illegible]	"[illegible]"	[illegible]	[illegible]
"F"或"f"	定点	结果:整数和十进制小数，负号可选。 受以下类型支持:所有数值类型。 精度说明符:十进制小数位数。 默认值精度说明符:由 NumberFormatInfo.NumberDecimalDigits 定义。 更多信息:定点("F")格式说明符。	1234.567 ("F", en-US) -> 1234.57 1234.567 ("F", de-DE) -> 1234,57 1234 ("F1", en-US) -> 1234.0 1234 ("F1", de-DE) -> 1234,0 -1234.56 ("F4", en-US) -> -1234.5600 -1234.56 ("F4", de-DE) -> -1234,5600
"G"或"g"	常规	结果:更紧凑的定点表示法或科学记数法。 受以下类型支持:所有数值类型。 精度说明符:有效位数。 默认值精度说明符:具体取决于数值类型。 更多信息:常规("G")格式说明符。	-123.456 ("G", en-US) -> -123.456 -123.456 ("G", sv-SE) -> -123,456 123.4546 ("G4", en-US) -> 123.5 123.4546 ("G4", sv-SE) -> 123,5 -1.234567890e-25 ("G", en-US) -> -1.23456789E-25 -1.234567890e-25 ("G", sv-SE) -> -1,23456789E-25
"N"或"n"	数字	结果:整数和十进制小数、组分隔符和十进制小数分隔符，负号可选。 受以下类型支持:所有数值类型。 精度说明符:所需的小数位数。 默认值精度说明符:由 NumberFormatInfo.NumberDecimalDigits 定义。 更多信息:数字("N")格式说明符。	1234.567 ("N", en-US) -> 1,234.57 1234.567 ("N", ru-RU) -> 1 234,57 1234 ("N1", en-US) -> 1,234.0 1234 ("N1", ru-RU) -> 1 234,0 -1234.56 ("N3", en-US) -> -1,234.560 -1234.56 ("N3", ru-RU) -> -1 234,560

[illegible]	"[illegible]"	[illegible]	[illegible]
"P"或"p"	百分比	结果:数字乘以 100 并以百分比符号显示。 受以下类型支持:所有数值类型。 精度说明符:所需的小数位数。 默认值精度说明符:由 NumberFormatInfo.PercentDecimalDigits 定义。 更多信息:百分比("P")格式说明符。	1 ("P", en-US) -> 100.00 % 1 ("P", fr-FR) -> 100,00 % -0.39678 ("P1", en-US) -> -39.7 % -0.39678 ("P1", fr-FR) -> -39,7 %
"R"或"r"	往返过程	结果:可以往返至相同数字的字符串。 受以下类型支持:Single、Double 和 BigInteger。 注意:建议只用于 BigInteger 类型。对于 Double 类型,请使用 "G17";对于 Single 类型,请使用 "G9"。 精度说明符:已忽略。 更多信息:往返过程("R")格式说明符。	123456789.12345678 ("R") -> 123456789.12345678 -1234567890.12345678 ("R") -> -1234567890.1234567
"X"或"x"	十六进制	结果:十六进制字符串。 受以下类型支持:仅限整型类型。 精度说明符:结果字符串中的位数。 更多信息:十六进制("X")格式说明符。	255 ("X") -> FF -1 ("x") -> ff 255 ("x4") -> 00ff -1 ("X4") -> 00FF
任何其他单个字符	未知说明符	结果:在运行时引发 FormatException。	

使用标准数字格式字符串

> **NOTE**
> 本文中的 C# 示例运行在 Try.NET 内联代码运行程序和演练环境中。选择"运行"按钮以在交互窗口中运行示例。执行代码后,可通过再次选择"运行"来修改它并运行已修改的代码。已修改的代码要么在交互窗口中运行,要么编译失败时,交互窗口将显示所有 C# 编译器错误消息。

可采用以下两种方式之一使用标准数字格式字符串定义数值的格式:

- 可以传递给拥有 `ToString` 参数的 `format` 方法的重载。下面的示例将数值的格式设置为当前区域性(在

此示例中，为 en-US 区域性）中的货币字符串。

```
Decimal value = static_cast<Decimal>(123.456);
Console::WriteLine(value.ToString("C2"));
// Displays $123.46
```

```
decimal value = 123.456m;
Console.WriteLine(value.ToString("C2"));
// Displays $123.46
```

```
Dim value As Decimal = 123.456d
Console.WriteLine(value.ToString("C2"))
' Displays $123.46
```

- 它可作为与 String.Format、Console.WriteLine 和 StringBuilder.AppendFormat 等方法一起使用的格式项中的 `formatString` 参数提供。有关更多信息，请参见复合格式设置。下面的示例使用格式项在字符串中插入货币值。

```
Decimal value = static_cast<Decimal>(123.456);
Console::WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

```
decimal value = 123.456m;
Console.WriteLine("Your account balance is {0:C2}.", value);
// Displays "Your account balance is $123.46."
```

```
Dim value As Decimal = 123.456d
Console.WriteLine("Your account balance is {0:C2}.", value)
' Displays "Your account balance is $123.46."
```

可以视需要选择提供 `alignment` 参数，以指定数字字段宽度以及值是右对齐还是左对齐。下面的示例在 28 位字符的字段中左对齐货币值，在 14 位字符的字段中右对齐货币值。

```
array<Decimal>^ amounts = { static_cast<Decimal>(16305.32),
                            static_cast<Decimal>(18794.16) };
Console::WriteLine("   Beginning Balance           Ending Balance");
Console::WriteLine("   {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
// Displays:
//        Beginning Balance           Ending Balance
//        $16,305.32                      $18,794.16
```

```
decimal[] amounts = { 16305.32m, 18794.16m };
Console.WriteLine("   Beginning Balance           Ending Balance");
Console.WriteLine("   {0,-28:C2}{1,14:C2}", amounts[0], amounts[1]);
// Displays:
//        Beginning Balance           Ending Balance
//        $16,305.32                      $18,794.16
```

```
Dim amounts() As Decimal = {16305.32d, 18794.16d}
Console.WriteLine("   Beginning Balance           Ending Balance")
Console.WriteLine("   {0,-28:C2}{1,14:C2}", amounts(0), amounts(1))
' Displays:
'        Beginning Balance           Ending Balance
'        $16,305.32                      $18,794.16
```

- 可以提供它作为内插字符串的内插表达式项中的 `formatString` 参数。有关详细信息，请参阅 C# 参考中的字符串内插主题，或 Visual Basic 参考中的内插字符串主题。

以下各节提供有关每个标准数字格式字符串的详细信息。

货币("C")格式说明符

"C"(或货币)格式说明符将数字转换为表示货币金额的字符串。精度说明符指示结果字符串中的所需小数位数。如果省略精度说明符，则默认精度由 NumberFormatInfo.CurrencyDecimalDigits 属性定义。

如果要设置格式的值的小数位数多于指定或默认数量，将在结果字符串中舍入小数值。如果指定的小数位数右侧的值大于或等于 5，则结果字符串中的最后一位数将向远离零的一侧舍入。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 属性，这些属性控制返回字符串的格式。

NUMBERFORMATINFO	
CurrencyPositivePattern	定义正值的货币符号的位置。
CurrencyNegativePattern	定义负值的货币符号的位置，并指定负号由括号表示还是由 NegativeSign 属性表示。
NegativeSign	定义在 CurrencyNegativePattern 指明不使用括号时使用的负号。
CurrencySymbol	定义货币符号。
CurrencyDecimalDigits	定义货币值中的默认小数位数。可使用精度说明符重写此值。
CurrencyDecimalSeparator	定义分隔整数位和小数位的字符串。
CurrencyGroupSeparator	定义分隔整数的组的字符串。
CurrencyGroupSizes	指定在组中显示的整数位数。

下面的示例使用货币格式说明符设置 Double 值的格式：

```
double value = 12345.6789;
Console::WriteLine(value.ToString("C", CultureInfo::CurrentCulture));

Console::WriteLine(value.ToString("C3", CultureInfo::CurrentCulture));

Console::WriteLine(value.ToString("C3",
                  CultureInfo::CreateSpecificCulture("da-DK")));
// The example displays the following output on a system whose
// current culture is English (United States):
//       $12,345.68
//       $12,345.679
//       kr 12.345,679
```

```
double value = 12345.6789;
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture));

Console.WriteLine(value.ToString("C3",
                  CultureInfo.CreateSpecificCulture("da-DK")));
// The example displays the following output on a system whose
// current culture is English (United States):
//       $12,345.68
//       $12,345.679
//       12.345,679 kr
```

```
Dim value As Double = 12345.6789
Console.WriteLine(value.ToString("C", CultureInfo.CurrentCulture))

Console.WriteLine(value.ToString("C3", CultureInfo.CurrentCulture))

Console.WriteLine(value.ToString("C3", _
                  CultureInfo.CreateSpecificCulture("da-DK")))
' The example displays the following output on a system whose
' current culture is English (United States):
'       $12,345.68
'       $12,345.679
'       kr 12.345,679
```

返回表首

十进制("D")格式说明符

"D"(或十进制)格式说明符将数字转换为十进制数字 (0-9) 的字符串，如果数字为负，则前面加负号。只有整型才支持此格式。

精度说明符指示结果字符串中所需的最少数字个数。如果需要的话，则用零填充该数字的左侧，以产生精度说明符给定的数字个数。如果未指定精度说明符，则默认值为表示不带前导零的整数所需的最小值。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。如下表所示，一个属性会影响结果字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义指示数字为负值的字符串。

下面的示例使用十进制格式说明符设置 Int32 值的格式。

```
int value;

value = 12345;
Console::WriteLine(value.ToString("D"));
// Displays 12345
Console::WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console::WriteLine(value.ToString("D"));
// Displays -12345
Console::WriteLine(value.ToString("D8"));
// Displays -00012345
```

```
int value;

value = 12345;
Console.WriteLine(value.ToString("D"));
// Displays 12345
Console.WriteLine(value.ToString("D8"));
// Displays 00012345

value = -12345;
Console.WriteLine(value.ToString("D"));
// Displays -12345
Console.WriteLine(value.ToString("D8"));
// Displays -00012345
```

```
Dim value As Integer

value = 12345
Console.WriteLine(value.ToString("D"))
' Displays 12345
Console.WriteLine(value.ToString("D8"))
' Displays 00012345

value = -12345
Console.WriteLine(value.ToString("D"))
' Displays -12345
Console.WriteLine(value.ToString("D8"))
' Displays -00012345
```

返回表首

指数("E")格式说明符

指数("E")格式说明符将数字转换为"-d.ddd...E+ddd"或"-d.ddd...e+ddd"形式的字符串，其中每个"d"表示一个数字 (0-9)。如果该数字为负，则该字符串以减号开头。小数点前总是恰好有一个数字。

精度说明符指示小数点后所需的位数。如果省略精度说明符，则使用默认值，即小数点后六位数字。

格式说明符的大小写指示为指数加前缀"E"还是"e"。指数总是由正号或负号以及最少三位数字组成。如果需要，用零填充指数以满足最少三位数字的要求。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 属性，这些属性控制返回字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义表示数字对于系数和指数都为负值的字符串。
NumberDecimalSeparator	定义在系数中将整数位与小数位分隔的字符串。
PositiveSign	定义指示指数为正值的字符串。

下面的示例使用指数格式说明符设置 Double 值的格式：

```
double value = 12345.6789;
Console::WriteLine(value.ToString("E", CultureInfo::InvariantCulture));
// Displays 1.234568E+004

Console::WriteLine(value.ToString("E10", CultureInfo::InvariantCulture));
// Displays 1.2345678900E+004

Console::WriteLine(value.ToString("e4", CultureInfo::InvariantCulture));
// Displays 1.2346e+004

Console::WriteLine(value.ToString("E",
                 CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

```
double value = 12345.6789;
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture));
// Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture));
// Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture));
// Displays 1.2346e+004

Console.WriteLine(value.ToString("E",
                 CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 1,234568E+004
```

```
Dim value As Double = 12345.6789
Console.WriteLine(value.ToString("E", CultureInfo.InvariantCulture))
' Displays 1.234568E+004

Console.WriteLine(value.ToString("E10", CultureInfo.InvariantCulture))
' Displays 1.2345678900E+004

Console.WriteLine(value.ToString("e4", CultureInfo.InvariantCulture))
' Displays 1.2346e+004

Console.WriteLine(value.ToString("E", _
                 CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 1,234568E+004
```

返回表首

定点（“F”）格式说明符

定点（“F”）格式说明符将数字转换为“-ddd.ddd…”形式的字符串，其中每个“d”表示一个数字 (0-9)。如果该数字为负，则该字符串以减号开头。

精度说明符指示所需的小数位数。如果省略精度说明符，则当前 NumberFormatInfo.NumberDecimalDigits 属性提供数值精度。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 对象的属性，这些属性控制结果字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义指示数字为负值的字符串。
NumberDecimalSeparator	定义将整数位与小数位分隔的字符串。
NumberDecimalDigits	定义默认小数位数。可使用精度说明符重写此值。

下面的示例使用定点格式说明符设置 Double 和 Int32 值的格式：

```
int integerNumber;
integerNumber = 17843;
Console::WriteLine(integerNumber.ToString("F",
                   CultureInfo::InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console::WriteLine(integerNumber.ToString("F3",
                   CultureInfo::InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console::WriteLine(doubleNumber.ToString("F", CultureInfo::InvariantCulture));
// Displays 18934.19

Console::WriteLine(doubleNumber.ToString("F0", CultureInfo::InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console::WriteLine(doubleNumber.ToString("F1", CultureInfo::InvariantCulture));
// Displays -1898300.2

Console::WriteLine(doubleNumber.ToString("F3",
                   CultureInfo::CreateSpecificCulture("es-ES")));
// Displays -1898300,199
```

```
int integerNumber;
integerNumber = 17843;
Console.WriteLine(integerNumber.ToString("F",
                  CultureInfo.InvariantCulture));
// Displays 17843.00

integerNumber = -29541;
Console.WriteLine(integerNumber.ToString("F3",
                  CultureInfo.InvariantCulture));
// Displays -29541.000

double doubleNumber;
doubleNumber = 18934.1879;
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture));
// Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0", CultureInfo.InvariantCulture));
// Displays 18934

doubleNumber = -1898300.1987;
Console.WriteLine(doubleNumber.ToString("F1", CultureInfo.InvariantCulture));
// Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays -1898300,199
```

```
Dim integerNumber As Integer
integerNumber = 17843
Console.WriteLine(integerNumber.ToString("F", CultureInfo.InvariantCulture))
' Displays 17843.00

integerNumber = -29541
Console.WriteLine(integerNumber.ToString("F3", CultureInfo.InvariantCulture))
' Displays -29541.000

Dim doubleNumber As Double
doubleNumber = 18934.1879
Console.WriteLine(doubleNumber.ToString("F", CultureInfo.InvariantCulture))
' Displays 18934.19

Console.WriteLine(doubleNumber.ToString("F0", CultureInfo.InvariantCulture))
' Displays 18934

doubleNumber = -1898300.1987
Console.WriteLine(doubleNumber.ToString("F1", CultureInfo.InvariantCulture))
' Displays -1898300.2

Console.WriteLine(doubleNumber.ToString("F3", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays -1898300,199
```

返回表首

常规("G")格式说明符

根据数字类型以及是否存在精度说明符，常规("G")格式说明符将数字转换为更紧凑的定点表示法或科学记数法。精度说明符定义可以出现在结果字符串中的最大有效位数。如果精度说明符被省略或为零，则数字的类型决定默认精度，如下表所示。

[illegible]	[illegible]
Byte 或 SByte	3 位
Int16 或 UInt16	5 位
Int32 或 UInt32	10 位
Int64	19 位
UInt64	20 位
BigInteger	无限制（与“R”相同）
Single	7 位
Double	15 位
Decimal	29 位

如果用科学记数法表示数字时指数大于 -5 而且小于精度说明符，则使用定点表示法；否则使用科学记数法。结果包含小数点（如果需要），并且忽略小数点后面的尾部零。如果精度说明符存在，并且结果的有效位数超过指定精度，则通过舍入移除多余的尾部数字。

但是，如果数字是 Decimal 并且省略精度说明符，将总是使用定点表示法并保留尾部零。

使用科学记数法时，如果格式说明符是“G”，则结果的指数带前缀“E”；如果格式说明符是“g”，则结果的指数带前缀“e”。指数最少包含两个数字。这与由指数格式说明符生成的科学记数法的格式不同，后者在指数中最少包括三个数字。

请注意，如果与 Double 值结合使用，"G17" 格式说明符可确保原始 Double 值成功往返。这是因为 Double 是符合 IEEE 754-2008 要求的双精度 (`binary64`) 浮点数，最高可提供 17 位有效数字的精度。建议使用此说明符，而不是 "R" 格式说明符，因为在某些情况下，"R" 无法成功往返双精度浮点值。下面的示例阐释了这样一种情况。

```
double original = 0.84551240822557006;
var rSpecifier = original.ToString("R");
var g17Specifier = original.ToString("G17");

var rValue = Double.Parse(rSpecifier);
var g17Value = Double.Parse(g17Specifier);

Console.WriteLine($"{original:G17} = {rSpecifier} (R): {original.Equals(rValue)}");
Console.WriteLine($"{original:G17} = {g17Specifier} (G17): {original.Equals(g17Value)}");
// The example displays the following output:
//     0.84551240822557006 = 0.84551240822557: False
//     0.84551240822557006 = 0.84551240822557006: True
```

```
Module Example
    Public Sub Main()
        Dim original As Double = 0.84551240822557006
        Dim rSpecifier = original.ToString("R")
        Dim g17Specifier = original.ToString("G17")

        Dim rValue = Double.Parse(rSpecifier)
        Dim g17Value = Double.Parse(g17Specifier)

        Console.WriteLine($"{original:G17} = {rSpecifier} (R): {original.Equals(rValue)}")
        Console.WriteLine($"{original:G17} = {g17Specifier} (G17): {original.Equals(g17Value)}")
    End Sub
End Module
' The example displays the following output:
'     0.84551240822557006 = 0.84551240822557 (R): False
'     0.84551240822557006 = 0.84551240822557006 (G17): True
```

如果与 Single 值结合使用，"G9" 格式说明符可确保原始 Single 值成功往返。这是因为 Single 是符合 IEEE 754-2008 要求的但精度 (`binary32`) 浮点数，最高可提供 9 位有效数字的精度。出于性能原因，我们建议使用它而不是"R"格式说明符。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 属性，这些属性控制结果字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义指示数字为负值的字符串。
NumberDecimalSeparator	定义将整数位与小数位分隔的字符串。
PositiveSign	定义指示指数为正值的字符串。

下面的示例使用常规格式说明符设置各种浮点值的格式：

```
double number;

number = 12345.6789;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays  12345.6789
Console::WriteLine(number.ToString("G",
                  CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console::WriteLine(number.ToString("G7", CultureInfo::InvariantCulture));
// Displays 12345.68

number = .0000023;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays 2.3E-06
Console::WriteLine(number.ToString("G",
                  CultureInfo::CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console::WriteLine(number.ToString("G", CultureInfo::InvariantCulture));
// Displays 0.0023

number = 1234;
Console::WriteLine(number.ToString("G2", CultureInfo::InvariantCulture));
// Displays 1.2E+03

number = Math::PI;
Console::WriteLine(number.ToString("G5", CultureInfo::InvariantCulture));
// Displays 3.1416
```

```
double number;

number = 12345.6789;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays  12345.6789
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture));
// Displays 12345.68

number = .0000023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 2.3E-06
Console.WriteLine(number.ToString("G",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 2,3E-06

number = .0023;
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture));
// Displays 0.0023

number = 1234;
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture));
// Displays 1.2E+03

number = Math.PI;
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture));
// Displays 3.1416
```

```
Dim number As Double

number = 12345.6789
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays  12345.6789
Console.WriteLine(number.ToString("G", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 12345,6789

Console.WriteLine(number.ToString("G7", CultureInfo.InvariantCulture))
' Displays 12345.68

number = .0000023
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays 2.3E-06
Console.WriteLine(number.ToString("G", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 2,3E-06

number = .0023
Console.WriteLine(number.ToString("G", CultureInfo.InvariantCulture))
' Displays 0.0023

number = 1234
Console.WriteLine(number.ToString("G2", CultureInfo.InvariantCulture))
' Displays 1.2E+03

number = Math.Pi
Console.WriteLine(number.ToString("G5", CultureInfo.InvariantCulture))
' Displays 3.1416
```

返回表首

数字（“N”）格式说明符

数字（"N"）格式说明符将数字转换为"-d,ddd,ddd.ddd…"形式的字符串，其中"-"表示负数符号（如果需要），"d"表示数字 (0-9)，","表示组分隔符，"."表示小数点符号。精度说明符指示小数点后所需的位数。如果省略精度限定符，则小数位数由当前的 NumberFormatInfo.NumberDecimalDigits 属性来定义。

结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 属性，这些属性控制结果字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义指示数字为负值的字符串。
NumberNegativePattern	定义负值的格式，并指定负号由括号表示还是由 NegativeSign 属性表示。
NumberGroupSizes	指定在组分隔符之间显示的整数位数。
NumberGroupSeparator	定义分隔整数的组的字符串。
NumberDecimalSeparator	定义分隔整数位和小数位的字符串。
NumberDecimalDigits	定义默认小数位数。可使用精度说明符重写此值。

下面的示例使用数字格式说明符设置各种浮点值的格式：

```
double dblValue = -12445.6789;
Console::WriteLine(dblValue.ToString("N", CultureInfo::InvariantCulture));
// Displays -12,445.68
Console::WriteLine(dblValue.ToString("N1",
                  CultureInfo::CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console::WriteLine(intValue.ToString("N1", CultureInfo::InvariantCulture));
// Displays 123,456,789.0
```

```
double dblValue = -12445.6789;
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture));
// Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1",
                  CultureInfo.CreateSpecificCulture("sv-SE")));
// Displays -12 445,7

int intValue = 123456789;
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture));
// Displays 123,456,789.0
```

```
Dim dblValue As Double = -12445.6789
Console.WriteLine(dblValue.ToString("N", CultureInfo.InvariantCulture))
' Displays -12,445.68
Console.WriteLine(dblValue.ToString("N1", _
                  CultureInfo.CreateSpecificCulture("sv-SE")))
' Displays -12 445,7

Dim intValue As Integer = 123456789
Console.WriteLine(intValue.ToString("N1", CultureInfo.InvariantCulture))
' Displays 123,456,789.0
```

返回表首

百分比("P")格式说明符

百分比("P")格式说明符将数字乘以 100 并将其转换为表示百分比的字符串。精度说明符指示所需的小数位数。如果省略精度说明符，则使用当前 PercentDecimalDigits 属性提供的默认数值精度。

下表列出了 NumberFormatInfo 属性，这些属性控制返回字符串的格式。

NUMBERFORMATINFO 属性	说明
PercentPositivePattern	定义正值的百分比符号的位置。
PercentNegativePattern	定义负值的百分比符号和负号的位置。
NegativeSign	定义指示数字为负值的字符串。
PercentSymbol	定义百分比符号。
PercentDecimalDigits	定义百分比值中的默认小数位数。可使用精度说明符重写此值。
PercentDecimalSeparator	定义分隔整数位和小数位的字符串。

NUMBERFORMATINFO 属性	说明
PercentGroupSeparator	定义分隔整数的组的字符串。
PercentGroupSizes	指定在组中显示的整数位数。

下面的示例使用百分比格式说明符设置浮点值的格式：

```
double number = .2468013;
Console::WriteLine(number.ToString("P", CultureInfo::InvariantCulture));
// Displays 24.68 %
Console::WriteLine(number.ToString("P",
                  CultureInfo::CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console::WriteLine(number.ToString("P1", CultureInfo::InvariantCulture));
// Displays 24.7 %
```

```
double number = .2468013;
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture));
// Displays 24.68 %
Console.WriteLine(number.ToString("P",
                  CultureInfo.CreateSpecificCulture("hr-HR")));
// Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture));
// Displays 24.7 %
```

```
Dim number As Double = .2468013
Console.WriteLine(number.ToString("P", CultureInfo.InvariantCulture))
' Displays 24.68 %
Console.WriteLine(number.ToString("P", _
                  CultureInfo.CreateSpecificCulture("hr-HR")))
' Displays 24,68%
Console.WriteLine(number.ToString("P1", CultureInfo.InvariantCulture))
' Displays 24.7 %
```

返回表首

往返过程（“R”）格式说明符

往返 ("R") 格式说明符试图确保将转换为字符串的数值分析回相同的数值。只有 Single、Double 和 BigInteger 类型支持此格式。

对于 Double 值，某些情况下的“R”格式说明符未能成功往返原始值。对于两个 Double 和 Single 值，它还提供相对较差的性能。建议改用 "G17" 格式说明符以成功往返 Double 值，并改用 "G9" 格式说明符以成功往返 Single 值。

如果使用此说明符设置 BigInteger 值的格式，其字符串表示形式将包含 BigInteger 值中的所有有效位。

尽管可以包括精度说明符，但会忽略它。使用此说明符时，往返过程优先于精度。结果字符串受当前 NumberFormatInfo 对象的格式信息的影响。下表列出了 NumberFormatInfo 属性，这些属性控制结果字符串的格式。

NUMBERFORMATINFO 属性	说明
NegativeSign	定义指示数字为负值的字符串。

NUMBERFORMATINFO 属性	说明
NumberDecimalSeparator	定义将整数位与小数位分隔的字符串。
PositiveSign	定义指示指数为正值的字符串。

下面的示例使用往返格式说明符设置 BigInteger 值的格式。

```
#using <System.Numerics.dll>

using namespace System;
using namespace System::Numerics;

void main()
{
   BigInteger value = BigInteger::Pow(Int64::MaxValue, 2);
   Console::WriteLine(value.ToString("R"));
}
// The example displays the following output:
//      85070591730234615847396907784232501249
```

```
using System;
using System.Numerics;

public class Example
{
   public static void Main()
   {
      var value = BigInteger.Pow(Int64.MaxValue, 2);
      Console.WriteLine(value.ToString("R"));
   }
}
// The example displays the following output:
//      85070591730234615847396907784232501249
```

```
Imports System.Numerics

Module Example
    Public Sub Main()
        Dim value = BigInteger.Pow(Int64.MaxValue, 2)
        Console.WriteLine(value.ToString("R"))
    End Sub
End Module
' The example displays the following output:
'      85070591730234615847396907784232501249
```

IMPORTANT

在某些情况下，如果使用“R”标准数字格式字符串格式化的 Double 值使用 `/platform:x64` 或 `/platform:anycpu` 开关编译并在 64 位系统上运行，则该值将无法成功往返。有关详细信息，请参阅以下段落。

若要解决使用“R”标准数字格式字符串格式化的 Double 值在使用 `/platform:x64` 或 `/platform:anycpu` 交换机进行编译并在 64 位系统上运行时所出现的往返不成功的问题，可以使用“G17”标准数字格式字符格式化 Double 值。以下示例将“R”格式字符串与无法成功往返的 Double 值配合使用，并使用“G17”格式字符串以成功往返原始值：

```
Console.WriteLine("Attempting to round-trip a Double with 'R':");
double initialValue = 0.6822871999174;
string valueString = initialValue.ToString("R",
                                           CultureInfo.InvariantCulture);
double roundTripped = double.Parse(valueString,
                                   CultureInfo.InvariantCulture);
Console.WriteLine("{0:R} = {1:R}: {2}\n",
                  initialValue, roundTripped, initialValue.Equals(roundTripped));

Console.WriteLine("Attempting to round-trip a Double with 'G17':");
string valueString17 = initialValue.ToString("G17",
                                             CultureInfo.InvariantCulture);
double roundTripped17 = double.Parse(valueString17,
                                     CultureInfo.InvariantCulture);
Console.WriteLine("{0:R} = {1:R}: {2}\n",
                  initialValue, roundTripped17, initialValue.Equals(roundTripped17));
// If compiled to an application that targets anycpu or x64 and run on an x64 system,
// the example displays the following output:
//       Attempting to round-trip a Double with 'R':
//       0.6822871999174 = 0.68228719991740006: False
//
//       Attempting to round-trip a Double with 'G17':
//       0.6822871999174 = 0.6822871999174: True
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Console.WriteLine("Attempting to round-trip a Double with 'R':")
        Dim initialValue As Double = 0.6822871999174
        Dim valueString As String = initialValue.ToString("R",
                                                  CultureInfo.InvariantCulture)
        Dim roundTripped As Double = Double.Parse(valueString,
                                                  CultureInfo.InvariantCulture)
        Console.WriteLine("{0:R} = {1:R}: {2}",
                          initialValue, roundTripped, initialValue.Equals(roundTripped))
        Console.WriteLine()

        Console.WriteLine("Attempting to round-trip a Double with 'G17':")
        Dim valueString17 As String = initialValue.ToString("G17",
                                                    CultureInfo.InvariantCulture)
        Dim roundTripped17 As Double = double.Parse(valueString17,
                                                CultureInfo.InvariantCulture)
        Console.WriteLine("{0:R} = {1:R}: {2}",
                          initialValue, roundTripped17, initialValue.Equals(roundTripped17))
    End Sub
End Module
' If compiled to an application that targets anycpu or x64 and run on an x64 system,
' the example displays the following output:
'       Attempting to round-trip a Double with 'R':
'       0.6822871999174 = 0.68228719991740006: False
'
'       Attempting to round-trip a Double with 'G17':
'       0.6822871999174 = 0.6822871999174: True
```

返回表首

十六进制(“X”)格式说明符

十六进制(“X”)格式说明符将数字转换为十六进制数的字符串。格式说明符的大小写指示对大于 9 的十六进制数使用大写字符还是小写字符。例如，使用“X”产生“ABCDEF”，使用“x”产生“abcdef”。只有整型才支持此格式。

精度说明符指示结果字符串中所需的最少数字个数。如果需要的话，则用零填充该数字的左侧，以产生精度说明

符给定的数字个数。

结果字符串不受当前 NumberFormatInfo 对象的格式信息的影响。

下面的示例使用十六进制数法格式说明符设置 Int32 值的格式。

```
int value;

value = 0x2045e;
Console::WriteLine(value.ToString("x"));
// Displays 2045e
Console::WriteLine(value.ToString("X"));
// Displays 2045E
Console::WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console::WriteLine(value.ToString("X"));
// Displays 75BCD15
Console::WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

```
int value;

value = 0x2045e;
Console.WriteLine(value.ToString("x"));
// Displays 2045e
Console.WriteLine(value.ToString("X"));
// Displays 2045E
Console.WriteLine(value.ToString("X8"));
// Displays 0002045E

value = 123456789;
Console.WriteLine(value.ToString("X"));
// Displays 75BCD15
Console.WriteLine(value.ToString("X2"));
// Displays 75BCD15
```

```
Dim value As Integer

value = &h2045e
Console.WriteLine(value.ToString("x"))
' Displays 2045e
Console.WriteLine(value.ToString("X"))
' Displays 2045E
Console.WriteLine(value.ToString("X8"))
' Displays 0002045E

value = 123456789
Console.WriteLine(value.ToString("X"))
' Displays 75BCD15
Console.WriteLine(value.ToString("X2"))
' Displays 75BCD15
```

返回表首

说明

控制面板设置

控制面板中“**区域和语言选项**”项中的设置会影响由格式化操作产生的结果字符串。这些设置用于初始化与当

前线程区域性关联的 NumberFormatInfo 对象，当前线程区域性提供用于控制格式设置的值。使用不同设置的计算机将生成不同的结果字符串。

此外，如果使用 CultureInfo(String) 构造函数实例化表示当前系统区域性的新 CultureInfo 对象，通过控制面板中的"区域和语言选项"项创建的任何自定义都会应用于新 CultureInfo 对象。可以使用 CultureInfo(String, Boolean) 构造函数来创建不会反映系统的自定义项的 CultureInfo 对象。

NumberFormatInfo 属性

格式设置受当前 NumberFormatInfo 对象的属性影响，它由当前线程区域性隐式提供或由调用格式设置的方法的 IFormatProvider 参数显式提供。为该参数指定 NumberFormatInfo 或 CultureInfo 对象。

> **NOTE**
> 有关自定义用于格式化数值的模式或字符串的信息，请参见 NumberFormatInfo 类主题。

整型和浮点型数值类型

对标准数字格式说明符的一些说明涉及到整型或浮点型数值类型。整型数值类型包括 Byte、SByte、Int16、Int32、Int64、UInt16、UInt32、UInt64 和 BigInteger。浮点型数值类型有 Decimal、Single 和 Double。

浮点型无穷大和 NaN

无论格式字符串原来是什么值，只要 Single 或 Double 浮点类型的值为正无穷大、负无穷大或非数值 (NaN)，格式字符串就分别是当前适用的 PositiveInfinitySymbol 对象指定的 NegativeInfinitySymbol、NaNSymbol 或 NumberFormatInfo 属性的值。

示例

> **NOTE**
> 本文中的一些 C# 示例在 Try.NET 内联代码运行程序和演练环境中运行。选择"运行"按钮以在交互窗口中运行示例 。执行代码后，可通过再次选择"运行"来修改它并运行已修改的代码 。已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

下面的示例使用 en-US 区域性和所有标准数字格式说明符设置一个整型数值和一个浮点型数值的格式。此示例使用两个特定的数值类型（Double 和 Int32），但对于任何一个其他数值基类型（Byte、SByte、Int16、Int32、Int64、UInt16、UInt32、UInt64、BigInteger、Decimal 和 Single）都将产生类似的结果。

```
// Display string representations of numbers for en-us culture
CultureInfo ci = new CultureInfo("en-us");

// Output floating point values
double floating = 10761.937554;
Console.WriteLine("C: {0}",
        floating.ToString("C", ci));           // Displays "C: $10,761.94"
Console.WriteLine("E: {0}",
        floating.ToString("E03", ci));         // Displays "E: 1.076E+004"
Console.WriteLine("F: {0}",
        floating.ToString("F04", ci));         // Displays "F: 10761.9376"
Console.WriteLine("G: {0}",
        floating.ToString("G", ci));           // Displays "G: 10761.937554"
Console.WriteLine("N: {0}",
        floating.ToString("N03", ci));         // Displays "N: 10,761.938"
Console.WriteLine("P: {0}",
        (floating/10000).ToString("P02", ci)); // Displays "P: 107.62 %"
Console.WriteLine("R: {0}",
        floating.ToString("R", ci));           // Displays "R: 10761.937554"
Console.WriteLine();

// Output integral values
int integral = 8395;
Console.WriteLine("C: {0}",
        integral.ToString("C", ci));           // Displays "C: $8,395.00"
Console.WriteLine("D: {0}",
        integral.ToString("D6", ci));          // Displays "D: 008395"
Console.WriteLine("E: {0}",
        integral.ToString("E03", ci));         // Displays "E: 8.395E+003"
Console.WriteLine("F: {0}",
        integral.ToString("F01", ci));         // Displays "F: 8395.0"
Console.WriteLine("G: {0}",
        integral.ToString("G", ci));           // Displays "G: 8395"
Console.WriteLine("N: {0}",
        integral.ToString("N01", ci));         // Displays "N: 8,395.0"
Console.WriteLine("P: {0}",
        (integral/10000.0).ToString("P02", ci)); // Displays "P: 83.95 %"
Console.WriteLine("X: 0x{0}",
        integral.ToString("X", ci));           // Displays "X: 0x20CB"
Console.WriteLine();
```

```
Option Strict On

Imports System.Globalization
Imports System.Threading

Module NumericFormats
    Public Sub Main()
        ' Display string representations of numbers for en-us culture
        Dim ci As New CultureInfo("en-us")

        ' Output floating point values
        Dim floating As Double = 10761.937554
        Console.WriteLine("C: {0}", _
                floating.ToString("C", ci))           ' Displays "C: $10,761.94"
        Console.WriteLine("E: {0}", _
                floating.ToString("E03", ci))         ' Displays "E: 1.076E+004"
        Console.WriteLine("F: {0}", _
                floating.ToString("F04", ci))         ' Displays "F: 10761.9376"
        Console.WriteLine("G: {0}", _
                floating.ToString("G", ci))           ' Displays "G: 10761.937554"
        Console.WriteLine("N: {0}", _
                floating.ToString("N03", ci))         ' Displays "N: 10,761.938"
        Console.WriteLine("P: {0}", _
                (floating / 10000).ToString("P02", ci)) ' Displays "P: 107.62 %"
        Console.WriteLine("R: {0}", _
                floating.ToString("R", ci))           ' Displays "R: 10761.937554"
        Console.WriteLine()

        ' Output integral values
        Dim integral As Integer = 8395
        Console.WriteLine("C: {0}", _
                integral.ToString("C", ci))           ' Displays "C: $8,395.00"
        Console.WriteLine("D: {0}", _
                integral.ToString("D6"))              ' Displays "D: 008395"
        Console.WriteLine("E: {0}", _
                integral.ToString("E03", ci))         ' Displays "E: 8.395E+003"
        Console.WriteLine("F: {0}", _
                integral.ToString("F01", ci))         ' Displays "F: 8395.0"
        Console.WriteLine("G: {0}", _
                integral.ToString("G", ci))           ' Displays "G: 8395"
        Console.WriteLine("N: {0}", _
                integral.ToString("N01", ci))         ' Displays "N: 8,395.0"
        Console.WriteLine("P: {0}", _
                (integral / 10000).ToString("P02", ci)) ' Displays "P: 83.95 %"
        Console.WriteLine("X: 0x{0}", _
                integral.ToString("X", ci))           ' Displays "X: 0x20CB"
        Console.WriteLine()
    End Sub
End Module
```

请参阅

- NumberFormatInfo
- 自定义数字格式字符串
- 格式设置类型
- 如何：用前导零填充数字
- 复合格式设置
- 示例：.NET Core WinForms 格式设置实用工具 (C#)
- 示例：.NET Core WinForms 格式设置实用工具 (Visual Basic)

自定义数字格式字符串

2021/11/16 •

你可以创建自定义数字格式字符串，这种字符串由一个或多个自定义数字说明符组成，用于定义设置数值数据格式的方式。自定义数字格式字符串是任何不属于 标准数字格式字符串的格式字符串。

所有数字类型的 `ToString` 方法的某些重载支持自定义数字格式字符串。例如，可将数字格式字符串提供给 ToString(String) 类型的 ToString(String, IFormatProvider) 方法和 Int32 方法。.NET 复合格式功能也支持自定义数字格式字符串，以供 Console 和 StreamWriter 类的一些 `Write` 和 `WriteLine` 方法、String.Format 方法以及 StringBuilder.AppendFormat 方法使用。字符串内插功能还支持自定义数字格式字符串。

> **TIP**
> 你可以下载格式设置实用工具，它属于一种 .NET Core Windows 窗体应用程序，通过该应用程序，你可将格式字符串应用于数值或日期和时间值并显示结果字符串。源代码适用于 C# 和 Visual Basic。

下表描述自定义数字格式说明符并显示由每个格式说明符产生的示例输出。有关使用自定义数字格式字符串的其他信息，请参见 说明 一节，有关使用方法的完整演示，请参见 示例 一节。

tttttt	“tt”	tt	tt
“0”	零占位符	用对应的数字（如果存在）替换零；否则，将在结果字符串中显示零。 更多信息：“0”自定义说明符。	1234.5678 ("00000") -> 01235 0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46
"#"	数字占位符	用对应的数字（如果存在）替换“#”符号；否则，不会在结果字符串中显示任何数字。 请注意，如果输入字符串中的相应数字是无意义的 0，则在结果字符串中不会出现任何数字。例如，0003 ("####") -> 3。 更多信息：“#”自定义说明符。	1234.5678 ("#####") -> 1235 0.45678 ("#.##", en-US) -> .46 0.45678 ("#.##", fr-FR) -> ,46
"."	小数点	确定小数点分隔符在结果字符串中的位置。 更多信息：“.”自定义说明符。	0.45678 ("0.00", en-US) -> 0.46 0.45678 ("0.00", fr-FR) -> 0,46

格式	名称	说明	示例
","	组分隔符和数字比例换算	用作组分隔符和数字比例换算说明符。作为组分隔符时，它在各个组之间插入本地化的组分隔符字符。作为数字比例换算说明符，对于每个指定的逗号，它将数字除以 1000。 更多信息："," 自定义说明符。	组分隔符说明符： 2147483647 ("##,#", en-US) -> 2,147,483,647 2147483647 ("##,#", es-ES) -> 2.147.483.647 比例换算说明符： 2147483647 ("#,#,,", en-US) -> 2,147 2147483647 ("#,#,,", es-ES) -> 2.147
"%"	百分比占位符	将数字乘以 100，并在结果字符串中插入本地化的百分比符号。 更多信息："%" 自定义说明符。	0.3697 ("%#0.00", en-US) -> %36.97 0.3697 ("%#0.00", el-GR) -> %36,97 0.3697 ("##.0 %", en-US) -> 37.0 % 0.3697 ("##.0 %", el-GR) -> 37,0 %
"‰"	千分比占位符	将数字乘以 1000，并在结果字符串中插入本地化的千分比符号。 更多信息："‰" 自定义说明符。	0.03697 ("#0.00‰", en-US) -> 36.97‰ 0.03697 ("#0.00‰", ru-RU) -> 36,97‰
"E0" "E+0" "E-0" "e0" "e+0" "e-0"	指数表示法	如果后跟至少一个 0（零），则使用指数表示法设置结果格式。"E" 或 "e" 指示指数符号在结果字符串中是大写还是小写。跟在 "E" 或 "e" 字符后面的零的数目确定指数中的最小位数。加号 (+) 指示符号字符总是置于指数前面。减号 (-) 指示符号字符仅置于负指数前面。 更多信息："E" 和 "e" 自定义说明符。	987654 ("#0.0e0") -> 98.8e4 1503.92311 ("0.0##e+00") -> 1.504e+03 1.8901385E-16 ("0.0e+00") -> 1.9e-16
"\"	转义符	使下一个字符被解释为文本而不是自定义格式说明符。 更多信息："\" 转义字符。	987654 ("\###00\#") -> #987654#
'string' "*string*"	文本字符串分隔符	指示应复制到未更改的结果字符串的封闭字符。 更多信息：字符文本。	68 ("# 'degrees'") -> 68 degrees 68 ("# ' degrees'") -> 68 degrees

[ILLEGIBLE]	"[ILLEGIBLE]"	[ILLEGIBLE]	[ILLEGIBLE]
;	部分分隔符	通过分隔格式字符串定义正数、负数和零各部分。 更多信息："；"部分分隔符。	12.345 ("#0.0#;(#0.0#);-\0-") -> 12.35 0 ("#0.0#;(#0.0#);-\0-") -> -0- -12.345 ("#0.0#;(#0.0#);-\0-") -> (12.35) 12.345 ("#0.0#;(#0.0#)") -> 12.35 0 ("#0.0#;(#0.0#)") -> 0.0 -12.345 ("#0.0#;(#0.0#)") -> (12.35)
其他	所有其他字符	字符将复制到未更改的结果字符串。 更多信息：字符文本。	68 ("# °") -> 68 °

以下各节提供有关每个自定义数字格式说明符的详细信息。

> **NOTE**
> 本文中的一些 C# 示例在 Try.NET 内联代码运行程序和演练环境中运行。选择"运行"按钮以在交互窗口中运行示例 。执行代码后，可通过再次选择"运行"来修改它并运行已修改的代码 。已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。

"0"自定义说明符

"0"自定义格式说明符用作零占位符符号。如果要设置格式的值在格式字符串中出现零的位置有一个数字，则将此数字复制到结果字符串中；否则，在结果字符串中显示零。小数点前最左边的零的位置和小数点后最右边的零的位置确定总在结果字符串中出现的数字范围。

"00"说明符使得值被舍入到小数点前最近的数字，其中零位总被舍去。例如，用"00"格式化 34.5 将得到值 35。

下面的示例显示几个使用包含零占位符的自定义格式字符串设置格式的值。

```
double value;

value = 123;
Console::WriteLine(value.ToString("00000"));
Console::WriteLine(String::Format("{0:00000}", value));
// Displays 00123

value = 1.2;
Console::WriteLine(value.ToString("0.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                  "{0:0.00}", value));
// Displays 1.20

Console::WriteLine(value.ToString("00.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:00.00}", value));
// Displays 01.20

CultureInfo^ daDK = CultureInfo::CreateSpecificCulture("da-DK");
Console::WriteLine(value.ToString("00.00", daDK));
Console::WriteLine(String::Format(daDK, "{0:00.00}", value));
// Displays 01,20

value = .56;
Console::WriteLine(value.ToString("0.0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0.0}", value));
// Displays 0.6

value = 1234567890;
Console::WriteLine(value.ToString("0,0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0,0}", value));
// Displays 1,234,567,890

CultureInfo^ elGR = CultureInfo::CreateSpecificCulture("el-GR");
Console::WriteLine(value.ToString("0,0", elGR));
Console::WriteLine(String::Format(elGR, "{0:0,0}", value));
// Displays 1.234.567.890

value = 1234567890.123456;
Console::WriteLine(value.ToString("0,0.0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0,0.0}", value));
// Displays 1,234,567,890.1

value = 1234.567890;
Console::WriteLine(value.ToString("0,0.00", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0,0.00}", value));
// Displays 1,234.57
```

```
 double value;

 value = 123;
 Console.WriteLine(value.ToString("00000"));
 Console.WriteLine(String.Format("{0:00000}", value));
 // Displays 00123

 value = 1.2;
 Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.00}", value));
 // Displays 1.20

 Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                 "{0:00.00}", value));
 // Displays 01.20

 CultureInfo daDK = CultureInfo.CreateSpecificCulture("da-DK");
 Console.WriteLine(value.ToString("00.00", daDK));
 Console.WriteLine(String.Format(daDK, "{0:00.00}", value));
 // Displays 01,20

 value = .56;
 Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                 "{0:0.0}", value));
 // Displays 0.6

 value = 1234567890;
 Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                 "{0:0,0}", value));
 // Displays 1,234,567,890

 CultureInfo elGR = CultureInfo.CreateSpecificCulture("el-GR");
 Console.WriteLine(value.ToString("0,0", elGR));
Console.WriteLine(String.Format(elGR, "{0:0,0}", value));
 // Displays 1.234.567.890

 value = 1234567890.123456;
 Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                 "{0:0,0.0}", value));
 // Displays 1,234,567,890.1

 value = 1234.567890;
 Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture));
 Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                 "{0:0,0.00}", value));
 // Displays 1,234.57
```

```
Dim value As Double

value = 123
Console.WriteLine(value.ToString("00000"))
Console.WriteLine(String.Format("{0:00000}", value))
' Displays 00123

value = 1.2
Console.Writeline(value.ToString("0.00", CultureInfo.InvariantCulture))
Console.Writeline(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.00}", value))
' Displays 1.20
Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value))
' Displays 01.20
Dim daDK As CultureInfo = CultureInfo.CreateSpecificCulture("da-DK")
Console.WriteLine(value.ToString("00.00", daDK))
Console.WriteLine(String.Format(daDK, "{0:00.00}", value))
' Displays 01,20

value = .56
Console.WriteLine(value.ToString("0.0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.0}", value))
' Displays 0.6

value = 1234567890
Console.WriteLine(value.ToString("0,0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0}", value))
' Displays 1,234,567,890
Dim elGR As CultureInfo = CultureInfo.CreateSpecificCulture("el-GR")
Console.WriteLine(value.ToString("0,0", elGR))
Console.WriteLine(String.Format(elGR, "{0:0,0}", value))
' Displays 1.234.567.890

value = 1234567890.123456
Console.WriteLine(value.ToString("0,0.0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.0}", value))
' Displays 1,234,567,890.1

value = 1234.567890
Console.WriteLine(value.ToString("0,0.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0,0.00}", value))
' Displays 1,234.57
```

返回表首

“#”自定义说明符

“#”自定义格式说明符用作数字占位符符号。如果设置了格式的值在格式字符串中显示"#"符号的位置有一个数字，则此数字被复制到结果字符串中。否则，结果字符串中的此位置不存储任何值。

请注意，如果零不是有效数字，此说明符永不显示零，即使零是字符串中的唯一数字也是如此。仅当零是所显示的数字中的有效数字时，才会显示零。

“##”格式字符串使得值被舍入到小数点前最近的数字，其中零总被舍去。例如，用“##”格式化 34.5 将得到值 35。

下面的示例显示几个使用包含数字占位符的自定义格式字符串设置格式的值。

```
 double value;

 value = 1.2;
 Console::WriteLine(value.ToString("#.##", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                   "{0:#.##}", value));
 // Displays 1.2

 value = 123;
 Console::WriteLine(value.ToString("#####"));
 Console::WriteLine(String::Format("{0:#####}", value));
 // Displays 123

 value = 123456;
 Console::WriteLine(value.ToString("[##-##-##]"));
 Console::WriteLine(String::Format("{0:[##-##-##]}", value));
// Displays [12-34-56]

 value = 1234567890;
 Console::WriteLine(value.ToString("#"));
 Console::WriteLine(String::Format("{0:#}", value));
 // Displays 1234567890

 Console::WriteLine(value.ToString("(###) ###-####"));
 Console::WriteLine(String::Format("{0:(###) ###-####}", value));
 // Displays (123) 456-7890
```

```
double value;

value = 1.2;
Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#.##}", value));
// Displays 1.2

value = 123;
Console.WriteLine(value.ToString("#####"));
Console.WriteLine(String.Format("{0:#####}", value));
// Displays 123

value = 123456;
Console.WriteLine(value.ToString("[##-##-##]"));
Console.WriteLine(String.Format("{0:[##-##-##]}", value));
 // Displays [12-34-56]

value = 1234567890;
Console.WriteLine(value.ToString("#"));
Console.WriteLine(String.Format("{0:#}", value));
// Displays 1234567890

Console.WriteLine(value.ToString("(###) ###-####"));
Console.WriteLine(String.Format("{0:(###) ###-####}", value));
// Displays (123) 456-7890
```

```
Dim value As Double

value = 1.2
Console.WriteLine(value.ToString("#.##", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#.##}", value))
' Displays 1.2

value = 123
Console.WriteLine(value.ToString("#####"))
Console.WriteLine(String.Format("{0:#####}", value))
' Displays 123

value = 123456
Console.WriteLine(value.ToString("[##-##-##]"))
Console.WriteLine(String.Format("{0:[##-##-##]}", value))
' Displays [12-34-56]

value = 1234567890
Console.WriteLine(value.ToString("#"))
Console.WriteLine(String.Format("{0:#}", value))
' Displays 1234567890

Console.WriteLine(value.ToString("(###) ###-####"))
Console.WriteLine(String.Format("{0:(###) ###-####}", value))
' Displays (123) 456-7890
```

若要返回空缺数字或前导零替换为空格的结果字符串，请使用 复合格式功能 并指定字段宽度，如以下示例所示。

```
using namespace System;

void main()
{
    Double value = .324;
    Console::WriteLine("The value is: '{0,5:#.###}'", value);
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'
```

```
using System;

public class Example
{
   public static void Main()
   {
      Double value = .324;
      Console.WriteLine("The value is: '{0,5:#.###}'", value);
   }
}
// The example displays the following output if the current culture
// is en-US:
//      The value is: ' .324'
```

```
Module Example
    Public Sub Main()
        Dim value As Double = .324
        Console.WriteLine("The value is: '{0,5:#.###}'", value)
    End Sub
End Module
' The example displays the following output if the current culture
' is en-US:
'      The value is: ' .324'
```

返回表首

"."自定义说明符

"."自定义格式说明符在结果字符串中插入本地化的小数分隔符。格式字符串中的第一个小数点确定设置了格式的值中的小数分隔符的位置;任何其他小数点会被忽略。

在结果字符串中用作小数分隔符的字符并非总是小数点;它由控制格式设置的 NumberDecimalSeparator 对象的 NumberFormatInfo 属性确定。

下面的示例使用"."格式说明符定义几个结果字符串中的小数点的位置。

```
 double value;

 value = 1.2;
 Console::WriteLine(value.ToString("0.00", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                   "{0:0.00}", value));
 // Displays 1.20

 Console::WriteLine(value.ToString("00.00", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                   "{0:00.00}", value));
 // Displays 01.20

 Console::WriteLine(value.ToString("00.00",
                    CultureInfo::CreateSpecificCulture("da-DK")));
 Console::WriteLine(String::Format(CultureInfo::CreateSpecificCulture("da-DK"),
                    "{0:00.00}", value));
 // Displays 01,20

 value = .086;
 Console::WriteLine(value.ToString("#0.##%", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                   "{0:#0.##%}", value));
 // Displays 8.6%

 value = 86000;
 Console::WriteLine(value.ToString("0.###E+0", CultureInfo::InvariantCulture));
 Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                    "{0:0.###E+0}", value));
// Displays 8.6E+4
```

```
double value;

value = 1.2;
Console.WriteLine(value.ToString("0.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.00}", value));
// Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value));
// Displays 01.20

Console.WriteLine(value.ToString("00.00",
                  CultureInfo.CreateSpecificCulture("da-DK")));
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                  "{0:00.00}", value));
// Displays 01,20

value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%

value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.###E+0}", value));
 // Displays 8.6E+4
```

```
Dim value As Double

value = 1.2
Console.Writeline(value.ToString("0.00", CultureInfo.InvariantCulture))
Console.Writeline(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.00}", value))
' Displays 1.20

Console.WriteLine(value.ToString("00.00", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:00.00}", value))
' Displays 01.20

Console.WriteLine(value.ToString("00.00", _
                  CultureInfo.CreateSpecificCulture("da-DK")))
Console.WriteLine(String.Format(CultureInfo.CreateSpecificCulture("da-DK"),
                  "{0:00.00}", value))
' Displays 01,20

value = .086
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value))
' Displays 8.6%

value = 86000
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                  "{0:0.###E+0}", value))
' Displays 8.6E+4
```

返回表首

“,”自定义说明符

“,”字符用作组分隔符和数字比例换算说明符。

- 组分隔符：如果在两个设置数字的整数位格式的数字占位符（0 或 #）之间指定一个或多个逗号，则在输出的整数部分中的每个数字组之间插入一个组分隔符字符。

  当前 NumberGroupSeparator 对象的 NumberGroupSizes 和 NumberFormatInfo 属性将确定用作数字组分隔符的字符以及每个数字组的大小。例如，如果使用字符串“#,#”和固定区域性对数字 1000 进行格式化，则输出为“1,000”。

- 数字比例换算符：如果在紧邻显式或隐式小数点的左侧指定一个或多个逗号，则对于每个逗号，将要设置格式的数字除以 1000。例如，如果使用字符串“0,,”对数字 100000000 进行格式化，则输出为“100”。

可以在同一格式字符串中使用组分隔符和数字比例换算说明符。例如，如果使用字符串“#,0,,”和固定区域性对数字 1000000000 进行格式化，则输出为“1,000”。

下面的示例演示如何使用逗号作为组分隔符。

```
double value = 1234567890;
Console::WriteLine(value.ToString("#,#", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:#,#}", value));
// Displays 1,234,567,890

Console::WriteLine(value.ToString("#,##0,,", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:#,##0,,}", value));
// Displays 1,235
```

```
double value = 1234567890;
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,#}", value));
// Displays 1,234,567,890

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value));
// Displays 1,235
```

```
Dim value As Double = 1234567890
Console.WriteLine(value.ToString("#,#", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,#}", value))
' Displays 1,234,567,890

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value))
' Displays 1,235
```

下面的示例演示如何使用逗号作为数字比例换算说明符。

```
  double value = 1234567890;
  Console::WriteLine(value.ToString("#,,", CultureInfo::InvariantCulture));
  Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,,}", value));
  // Displays 1235

  Console::WriteLine(value.ToString("#,,,", CultureInfo::InvariantCulture));
  Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,,,}", value));
// Displays 1

  Console::WriteLine(value.ToString("#,##0,,", CultureInfo::InvariantCulture));
  Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                    "{0:#,##0,,}", value));
// Displays 1,235
```

```
double value = 1234567890;
Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,,}", value));
// Displays 1235

Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,,,}", value));
// Displays 1

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value));
// Displays 1,235
```

```
Dim value As Double = 1234567890
Console.WriteLine(value.ToString("#,,", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture, "{0:#,,}", value))
' Displays 1235

Console.WriteLine(value.ToString("#,,,", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,,,}", value))
' Displays 1

Console.WriteLine(value.ToString("#,##0,,", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#,##0,,}", value))
' Displays 1,235
```

返回表首

“%”自定义说明符

格式字符串中的百分号 (%) 将使数字在设置格式之前乘以 100。本地化的百分比符号插入到数字在格式字符串中出现 % 的位置。使用的百分比字符由当前 PercentSymbol 对象的 NumberFormatInfo 属性定义。

下面的示例定义几个包含“%”自定义说明符的自定义格式字符串。

```
double value = .086;
Console::WriteLine(value.ToString("#0.##%", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:#0.##%}", value));
// Displays 8.6%
```

```
double value = .086;
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value));
// Displays 8.6%
```

```
Dim value As Double = .086
Console.WriteLine(value.ToString("#0.##%", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:#0.##%}", value))
' Displays 8.6%
```

返回表首

“‰”自定义说明符

格式字符串中的千分比字符(‰ 或 \u2030)将使数字在设置格式之前乘以 1000。在返回的字符串中，相应的千分比符号插在格式字符串中出现 ‰ 符号的位置。所用的千分比字符由提供特定于区域性的格式设置信息的对象的 NumberFormatInfo.PerMilleSymbol 属性定义。

下面的示例定义一个包含“‰”自定义说明符的自定义格式字符串。

```
double value = .00354;
String^ perMilleFmt = "#0.## " + '\u2030';
Console::WriteLine(value.ToString(perMilleFmt, CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:" + perMilleFmt + "}", value));
// Displays 3.54‰
```

```
double value = .00354;
string perMilleFmt = "#0.## " + '\u2030';
Console.WriteLine(value.ToString(perMilleFmt, CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:" + perMilleFmt + "}", value));
// Displays 3.54‰
```

```
Dim value As Double = .00354
Dim perMilleFmt As String = "#0.## " & ChrW(&h2030)
Console.WriteLine(value.ToString(perMilleFmt, CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:" + perMilleFmt + "}", value))
' Displays 3.54 ‰
```

返回表首

“E”和“e”自定义说明符

如果“E”、“E+”、“E-”、“e”、“e+”或“e-”中的任何一个字符串出现在格式字符串中，而且后面紧跟至少一个零，则数

字用科学记数法来设置格式，在数字和指数之间插入“E”或“e”。跟在科学记数法指示符后面的零的数目确定指数输出的最小位数。“E+”和“e+”格式指示加号或减号应总是置于指数前面。“E”、“E-”、“e”或“e-”格式指示符号字符应仅置于负指数前面。

下面的示例使用科学记数法说明符设置几个数值的格式。

```
double value = 86000;
Console::WriteLine(value.ToString("0.###E+0", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0.###E+0}", value));
// Displays 8.6E+4

Console::WriteLine(value.ToString("0.###E+000", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0.###E+000}", value));
// Displays 8.6E+004

Console::WriteLine(value.ToString("0.###E-000", CultureInfo::InvariantCulture));
Console::WriteLine(String::Format(CultureInfo::InvariantCulture,
                                  "{0:0.###E-000}", value));
// Displays 8.6E004
```

```
double value = 86000;
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+0}", value));
// Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+000}", value));
// Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000", CultureInfo.InvariantCulture));
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E-000}", value));
// Displays 8.6E004
```

```
Dim value As Double = 86000
Console.WriteLine(value.ToString("0.###E+0", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+0}", value))
' Displays 8.6E+4

Console.WriteLine(value.ToString("0.###E+000", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E+000}", value))
' Displays 8.6E+004

Console.WriteLine(value.ToString("0.###E-000", CultureInfo.InvariantCulture))
Console.WriteLine(String.Format(CultureInfo.InvariantCulture,
                                "{0:0.###E-000}", value))
' Displays 8.6E004
```

返回表首

“\”转义字符

格式字符串中的“#”、“0”、“.”、“,”、“%”和“‰”符号被解释为格式说明符而不是文本字符。大写和小写“E”以及 + 和 - 符号也可能被解释为格式说明符，具体取决于它们在自定义格式字符串中的位置。

若要防止某个字符被解释为格式说明符，你可以在该字符前面加上反斜杠(即转义字符)。转义字符表示以下字符为应包含在未更改的结果字符串中的字符文本。

若在要结果字符串中包括反斜杠，必须使用另一个反斜杠 (`\\`) 对其转义。

> **NOTE**
>
> 一些编译器(如 C++ 和 C# 编译器)也可能会将单个反斜杠字符解释为转义字符。若要确保在设置格式时正确解释字符串，在 C# 中，可以在字符串之前使用原义字符串文本字符(@ 字符)，或者在 C# 和 C++ 中，在每个反斜杠之前另外添加一个反斜杠字符。下面的 C# 示例阐释了这两种方法。

下面的示例使用转义字符，以防格式设置操作将"#"、"0"和"\"字符解释为转义字符或格式说明符。C# 示例使用附加的反斜杠以确保将原反斜杠解释为文本字符。

```
int value = 123;
Console::WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#"));
Console::WriteLine(String::Format("{0:\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#}",
                                  value));
// Displays ### 123 dollars and 00 cents ###

Console::WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \0\0 cents \\#\\#\\#"));
Console::WriteLine(String::Format("{0:\\#\\#\\# ##0 dollars and \0\0 cents \\#\\#\\#}",
                                  value));
// Displays ### 123 dollars and 00 cents ###

Console::WriteLine(value.ToString("\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\"));
Console::WriteLine(String::Format("{0:\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\}",
                                  value));
// Displays \\\ 123 dollars and 00 cents \\\

Console::WriteLine(value.ToString("\\\\\\ ##0 dollars and \0\0 cents \\\\\\"));
Console::WriteLine(String::Format("{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                  value));
// Displays \\\ 123 dollars and 00 cents \\\
```

```
int value = 123;
Console.WriteLine(value.ToString("\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#"));
Console.WriteLine(String.Format("{0:\\#\\#\\# ##0 dollars and \\0\\0 cents \\#\\#\\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString(@"\#\#\# ##0 dollars and \0\0 cents \#\#\#"));
Console.WriteLine(String.Format(@"{0:\#\#\# ##0 dollars and \0\0 cents \#\#\#}",
                                value));
// Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\"));
Console.WriteLine(String.Format("{0:\\\\\\\\\\\\ ##0 dollars and \\0\\0 cents \\\\\\\\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\

Console.WriteLine(value.ToString(@"\\\\\\ ##0 dollars and \0\0 cents \\\\\\"));
Console.WriteLine(String.Format(@"{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value));
// Displays \\\ 123 dollars and 00 cents \\\
```

```
Dim value As Integer = 123
Console.WriteLine(value.ToString("\#\#\# ##0 dollars and \0\0 cents \#\#\#"))
Console.WriteLine(String.Format("{0:\#\#\# ##0 dollars and \0\0 cents \#\#\#}",
                                value))
' Displays ### 123 dollars and 00 cents ###

Console.WriteLine(value.ToString("\\\\\\ ##0 dollars and \0\0 cents \\\\\\"))
Console.WriteLine(String.Format("{0:\\\\\\ ##0 dollars and \0\0 cents \\\\\\}",
                                value))
' Displays \\\ 123 dollars and 00 cents \\\
```

返回表首

";"部分分隔符

分号 (;) 是条件格式说明符，它可以对数字应用不同的格式设置，具体取决于值为正、为负还是为零。为产生这种行为，自定义格式字符串可以包含最多三个用分号分隔的部分。下表描述了这些部分。

tttt	tt
一个部分	格式字符串应用于所有值。
两个部分	第一部分应用于正值和零，第二部分应用于负值。 如果要设置格式的数字为负，但根据第二部分中的格式舍入后为零，则最终的零根据第一部分进行格式设置。
三个部分	第一部分应用于正值，第二部分应用于负值，第三部分应用于零。 第二部分可以留空(分号间没有任何内容)，在这种情况下，第一部分应用于所有非零值。 如果要设置格式的数字为非零值，但根据第一部分或第二部分中的格式舍入后为零，则最终的零根据第三部分进行格式设置。

格式化最终值时，部分分隔符忽略所有先前存在的与数字关联的格式设置。例如，使用部分分隔符时，显示的负值永远不带负号。如果你希望格式化后的最终值带有负号，则应明确包含负号，让它作为自定义格式说明符的组成部分。

下面的示例使用";"格式说明符来分别设置正数、负数和零的格式。

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

String^ fmt2 = "##;(##)";
String^ fmt3 = "##;(##);**Zero**";

Console::WriteLine(posValue.ToString(fmt2));
Console::WriteLine(String::Format("{0:" + fmt2 + "}", posValue));
// Displays 1234

Console::WriteLine(negValue.ToString(fmt2));
Console::WriteLine(String::Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console::WriteLine(zeroValue.ToString(fmt3));
Console::WriteLine(String::Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

```
double posValue = 1234;
double negValue = -1234;
double zeroValue = 0;

string fmt2 = "##;(##)";
string fmt3 = "##;(##);**Zero**";

Console.WriteLine(posValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue));
// Displays 1234

Console.WriteLine(negValue.ToString(fmt2));
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue));
// Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3));
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue));
// Displays **Zero**
```

```
Dim posValue As Double = 1234
Dim negValue As Double = -1234
Dim zeroValue As Double = 0

Dim fmt2 As String = "##;(##)"
Dim fmt3 As String = "##;(##);**Zero**"

Console.WriteLine(posValue.ToString(fmt2))
Console.WriteLine(String.Format("{0:" + fmt2 + "}", posValue))
' Displays 1234

Console.WriteLine(negValue.ToString(fmt2))
Console.WriteLine(String.Format("{0:" + fmt2 + "}", negValue))
' Displays (1234)

Console.WriteLine(zeroValue.ToString(fmt3))
Console.WriteLine(String.Format("{0:" + fmt3 + "}", zeroValue))
' Displays **Zero**
```

返回表首

字符文本

出现在自定义数值格式字符串中的格式说明符始终解释为格式字符而不是文本字符。这包括以下字符：

- 0
- #
- %
- ‰
- '
- \
- .
- ,
- E 或 e，取决于它在格式字符串中的位置。

所有其他字符始终解释为字符文本，在格式设置操作中，将按原样包含在结果字符串中。在分析操作中，这些字符必须与输入字符串中的字符完全匹配；比较时区分大小写。

以下示例演示了文本字符单位（这里是"千"）的一种常见用法：

```
double n = 123.8;
Console.WriteLine($"{n:#,##0.0K}");
// The example displays the following output:
//       123.8K
```

```
Dim n As Double = 123.8
Console.WriteLine($"{n:#,##0.0K}")
' The example displays the following output:
'       123.8K
```

可通过两种方法来指示要将字符解释为文本字符而不是格式字符，以便这些字符可以包含在结果字符串中，或者在输入字符串中成功完成分析：

- 通过对格式字符进行转义处理。有关详细信息，请参阅"\"转义字符。
- 通过将整个文本字符串括在单引号中。

以下示例使用这两种方法将保留字符包含在自定义数值格式字符串中。

```
double n = 9.3;
Console.WriteLine($@"{n:##.0\%}");
Console.WriteLine($@"{n:\'##\'}");
Console.WriteLine($@"{n:\\##\\}");
Console.WriteLine();
Console.WriteLine($"{n:##.0'%'}");
Console.WriteLine($@"{n:'\'##'\'}");
// The example displays the following output:
//       9.3%
//       '9'
//       \9\
//
//       9.3%
//       \9\
```

```
Dim n As Double = 9.3
Console.WriteLine($"{n:##.0\%}")
Console.WriteLine($"{n:\'##\'}")
Console.WriteLine($"{n:\\##\\}")
Console.WriteLine()
Console.WriteLine($"{n:##.0'%'}")
Console.WriteLine($"{n:'\'##'\'}")
' The example displays the following output:
'       9.3%
'       '9'
'       \9\
'
'       9.3%
'       \9\
```

说明

浮点型无穷大和 NaN

无论格式字符串原来是什么值，只要 Single 或 Double 浮点类型的值为正无穷大、负无穷大或非数值 (NaN)，格式字符串就分别是当前适用的 PositiveInfinitySymbol对象指定的 NegativeInfinitySymbol、NaNSymbol 或 NumberFormatInfo 属性的值。

控制面板设置

控制面板中“**区域和语言选项**”项中的设置会影响由格式化操作产生的结果字符串。这些设置用于初始化与当前线程区域性关联的 NumberFormatInfo 对象，并且当前线程区域性将提供用于控制格式设置的值。使用不同设置的计算机将生成不同的结果字符串。

此外，如果使用 CultureInfo(String) 构造函数实例化一个新的 CultureInfo 对象以表示与当前的系统区域性相同的区域性，则通过控制面板中的“**区域和语言选项**”建立的任何自定义都将应用到新的 CultureInfo 对象。可以使用 CultureInfo(String, Boolean) 构造函数来创建不会反映系统的自定义项的 CultureInfo 对象。

舍入和定点格式字符串

对于固定点格式字符串(即不包含科学记数法格式字符的格式字符串)，数字被舍入为与小数点右边的数字占位符数目相同的小数位数。如果格式字符串不包含小数点，数字被舍入为最接近的整数。如果数字位数多于小数点左边数字占位符的个数，多余的数字被复制到结果字符串中紧挨着第一个数字占位符的前面。

返回表首

示例

下面的示例演示两个自定义数字格式字符串。在这两个示例中，数字占位符 (`#`) 显示数值数据，且所有其他字符被复制到结果字符串。

```
double number1 = 1234567890;
String^ value1 = number1.ToString("(###) ###-####");
Console::WriteLine(value1);

int number2 = 42;
String^ value2 = number2.ToString("My Number = #");
Console::WriteLine(value2);
// The example displays the following output:
//       (123) 456-7890
//       My Number = 42
```

```
double number1 = 1234567890;
string value1 = number1.ToString("(###) ###-####");
Console.WriteLine(value1);

int number2 = 42;
string value2 = number2.ToString("My Number = #");
Console.WriteLine(value2);
// The example displays the following output:
//       (123) 456-7890
//       My Number = 42
```

```
Dim number1 As Double = 1234567890
Dim value1 As String = number1.ToString("(###) ###-####")
Console.WriteLine(value1)

Dim number2 As Integer = 42
Dim value2 As String = number2.ToString("My Number = #")
Console.WriteLine(value2)
' The example displays the following output:
'       (123) 456-7890
'       My Number = 42
```

返回表首

请参阅

- System.Globalization.NumberFormatInfo
- 格式设置类型
- 标准数字格式字符串
- 如何：用前导零填充数字
- 示例：.NET Core WinForms 格式设置实用工具 (C#)
- 示例：.NET Core WinForms 格式设置实用工具 (Visual Basic)

标准日期和时间格式字符串

2021/11/16 •

标准日期和时间格式字符串使用单个字符作为格式说明符来定义 DateTime 或 DateTimeOffset 值的文本表示形式。任何包含一个以上字符(包括空白)的日期和时间格式字符串都会作为自定义日期和时间格式字符串进行解释。可通过两种方式使用标准或自定义格式字符串:

- 定义由格式设置操作生成的字符串。
- 定义可通过分析操作转换为 DateTime 或 DateTimeOffset 值的日期和时间值的文本表示形式。

> **TIP**
> 你可以下载格式设置实用工具,它属于一种 .NET Windows 窗体应用程序,通过该应用程序,你可将格式字符串应用于数值或日期和时间值并显示结果字符串。源代码适用于 C# 和 Visual Basic。

> **NOTE**
> 本文中的一些 C# 示例在 Try.NET 内联代码运行程序和演练环境中运行。选择"运行"按钮以在交互窗口中运行示例 。执行代码后,可通过再次选择"运行"来修改它并运行已修改的代码 。已修改的代码要么在交互窗口中运行,要么编译失败时,交互窗口将显示所有 C# 编译器错误消息。
>
> Try.NET 内联代码运行程序和演练环境的本地时区是协调世界时 (UTC)。这可能会影响用于说明 DateTime、DateTimeOffset 和 TimeZoneInfo 类型及其成员的示例的行为和输出。

格式说明符表

下表描述了标准日期和时间格式说明符。除非另行说明,否则,特定的标准日期和时间格式说明符将产生相同的字符串表示形式,这与它是与 DateTime 值还是 DateTimeOffset 值一起使用无关。有关使用标准日期和时间格式字符串的更多信息,请参阅控制面板设置和DateTimeFormatInfo 属性。

tttttt	tt	tt
"d"	短日期模式。 有关详细信息,请参阅短日期("d")格式说明符。	2009-06-15T13:45:30 -> 6/15/2009 (en-US) 2009-06-15T13:45:30 -> 15/06/2009 (fr-FR) 2009-06-15T13:45:30 -> 2009/06/15 (ja-JP)
"D"	长日期模式。 有关详细信息,请参阅长日期("D")格式说明符。	2009-06-15T13:45:30 -> Monday, June 15, 2009 (en-US) 2009-06-15T13:45:30 -> 15 июня 2009 г. (ru-RU) 2009-06-15T13:45:30 -> Montag, 15. Juni 2009 (de-DE)

[illegible]	[illegible]	[illegible]
“f”	完整日期/时间模式（短时间）。 更多信息：完整日期短时间（“f”）格式说明符。	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45 μμ (el-GR)
“F”	完整日期/时间模式（长时间）。 更多信息：完整日期长时间（“F”）格式说明符。	2009-06-15T13:45:30 -> Monday, June 15, 2009 1:45:30 PM (zh-CN) 2009-06-15T13:45:30 -> den 15 juni 2009 13:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 1:45:30 μμ (el-GR)
“g”	常规日期/时间模式（短时间）。 更多信息：常规日期短时间（“g”）格式说明符。	2009-06-15T13:45:30 -> 6/15/2009 1:45 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45 (zh-CN)
“G”	常规日期/时间模式（长时间）。 更多信息：常规日期长时间（“G”）格式说明符。	2009-06-15T13:45:30 -> 6/15/2009 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 15/06/2009 13:45:30 (es-ES) 2009-06-15T13:45:30 -> 2009/6/15 13:45:30 (zh-CN)
“M”、“m”	月/日模式。 更多信息：月（“M”、“m”）格式说明符。	2009-06-15T13:45:30 -> June 15 (en-US) 2009-06-15T13:45:30 -> 15. juni (da-DK) 2009-06-15T13:45:30 -> 15 Juni (id-ID)

ffffff	ff	ff
“O”、“o”	往返日期/时间模式。 更多信息：往返（“O”、“o”）格式说明符。	DateTime 值： 2009-06-15T13:45:30 (DateTimeKind.Local) --> 2009-06-15T13:45:30.0000000-07:00 2009-06-15T13:45:30 (DateTimeKind.Utc) --> 2009-06-15T13:45:30.0000000Z 2009-06-15T13:45:30 (DateTimeKind.Unspecified) --> 2009-06-15T13:45:30.0000000 DateTimeOffset 值： 2009-06-15T13:45:30-07:00 --> 2009-06-15T13:45:30.0000000-07:00
“R”、“r”	RFC1123 模式。 更多信息：RFC1123（“R”、“r”）格式说明符。	2009-06-15T13:45:30 -> Mon, 15 Jun 2009 20:45:30 GMT
“s”	可排序日期/时间模式。 更多信息：可排序（“s”）格式说明符。	2009-06-15T13:45:30 (DateTimeKind.Local) -> 2009-06-15T13:45:30 2009-06-15T13:45:30 (DateTimeKind.Utc) -> 2009-06-15T13:45:30
“t”	短时间模式。 更多信息：短时间（“t”）格式说明符。	2009-06-15T13:45:30 -> 1:45 PM (en-US) 2009-06-15T13:45:30 -> 13:45 (hr-HR) 2009-06-15T13:45:30 -> 01:45 م (ar-EG)
“T”	长时间模式。 更多信息：长时间（“T”）格式说明符。	2009-06-15T13:45:30 -> 1:45:30 PM (en-US) 2009-06-15T13:45:30 -> 13:45:30 (hr-HR) 2009-06-15T13:45:30 -> 01:45:30 م (ar-EG)
“u”	通用可排序日期/时间模式。 更多信息：通用可排序（“u”）格式说明符。	带有 DateTime 值：2009-06-15T13:45:30 -> 2009-06-15 13:45:30Z 带有 DateTimeOffset 值：2009-06-15T13:45:30 -> 2009-06-15 20:45:30Z

“”“”	“”	“”
“U”	通用完整日期/时间模式。 更多信息：通用完整（“U”）格式说明符。	2009-06-15T13:45:30 -> Monday, June 15, 2009 8:45:30 PM (en-US) 2009-06-15T13:45:30 -> den 15 juni 2009 20:45:30 (sv-SE) 2009-06-15T13:45:30 -> Δευτέρα, 15 Ιουνίου 2009 8:45:30 μμ (el-GR)
“Y”、“y”	年月模式。 更多信息：年月（“Y”、“y”）格式说明符。	2009-06-15T13:45:30 -> June 2009 (en-US) 2009-06-15T13:45:30 -> juni 2009 (da-DK) 2009-06-15T13:45:30 -> Juni 2009 (id-ID)
任何其他单个字符	未知说明符。	引发运行时 FormatException。

标准格式字符串的工作原理

在格式设置操作中，标准格式字符串只是自定义格式字符串的别名。使用别名引用自定义格式字符串的优点是：尽管别名保持固定不变，自定义格式字符串自身也可以变化。这很重要，因为日期和时间值的字符串表示形式通常会因区域性而异。例如，“d”标准格式字符串指示应使用短日期模式显示日期和时间值。对于固定区域性，此模式为“MM/dd/yyyy”。对于 fr-FR 区域性，此模式为“dd/MM/yyyy”。对于 ja-JP 区域性，此模式为“yyyy/MM/dd”。

如果格式设置操作中的标准格式字符串映射到某个特定区域性的自定义格式字符串，则应用程序可定义该特定区域性，并通过以下方式之一使用其自定义格式字符串：

- 可使用默认的（或当前的）区域性。下面的示例使用当前区域性的短日期格式显示日期。在此情况下，当前区域性为 en-US。

```
// Display using current (en-us) culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
Console.WriteLine(thisDate.ToString("d"));           // Displays 3/15/2008
```

```
' Display using current (en-us) culture's short date format
Dim thisDate As Date = #03/15/2008#
Console.WriteLine(thisDate.ToString("d"))     ' Displays 3/15/2008
```

- 可以传递一个表示区域性的 CultureInfo 对象，该区域性的格式设置将用于具有 IFormatProvider 参数的方法。下面的示例使用 pt-BR 区域性的短日期格式显示日期。

```
// Display using pt-BR culture's short date format
DateTime thisDate = new DateTime(2008, 3, 15);
CultureInfo culture = new CultureInfo("pt-BR");
Console.WriteLine(thisDate.ToString("d", culture));  // Displays 15/3/2008
```

```
' Display using pt-BR culture's short date format
Dim thisDate As Date = #03/15/2008#
Dim culture As New CultureInfo("pt-BR")
Console.WriteLine(thisDate.ToString("d", culture))   ' Displays 15/3/2008
```

- 可以传递一个 DateTimeFormatInfo 对象，该对象向具有 IFormatProvider 参数的方法提供格式设置信息。下面的示例使用 hr-HR 区域性的 DateTimeFormatInfo 对象中的短日期格式显示日期。

```
// Display using date format information from hr-HR culture
DateTime thisDate = new DateTime(2008, 3, 15);
DateTimeFormatInfo fmt = (new CultureInfo("hr-HR")).DateTimeFormat;
Console.WriteLine(thisDate.ToString("d", fmt));      // Displays 15.3.2008
```

```
' Display using date format information from hr-HR culture
Dim thisDate As Date = #03/15/2008#
Dim fmt As DateTimeFormatInfo = (New CultureInfo("hr-HR")).DateTimeFormat
Console.WriteLine(thisDate.ToString("d", fmt))   ' Displays 15.3.2008
```

NOTE

有关自定义用于格式化日期和时间值的模式或字符串的信息，请参见 NumberFormatInfo 类主题。

某些情况下，标准格式字符串用作固定不变的较长自定义格式字符串的简便缩写。有四个标准格式字符串属于这一类别："O"（或"o"）、"R"（或"r"）、"s"和"u"。这些字符串对应于由固定区域性定义的自定义格式字符串。通过这些字符串得到的日期和时间值的字符串表示形式在各个区域性中都应是相同的。下表提供了有关这四个标准日期和时间格式字符串的信息。

[illegible]	[illegible] DATETIMEFORMATINFO.INVARIANTINFO [illegible]	[illegible]
"O"或"o"	None	yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzz
"R"或"r"	RFC1123Pattern	ddd, dd MMM yyyy HH':'mm':'ss 'GMT'
"s"	SortableDateTimePattern	yyyy'-'MM'-'dd'T'HH':'mm':'ss
"u"	UniversalSortableDateTimePattern	yyyy'-'MM'-'dd HH':'mm':'ss'Z'

通过 DateTime.ParseExact 或 DateTimeOffset.ParseExact 方法，还可以在分析操作中使用标准格式字符串，这些方法需要输入字符串才能完全符合确保分析操作成功的特定模式。许多标准格式字符串都映射到多个自定义格式字符串，因此，可采用各种格式表示日期和时间值并且分析操作仍然会成功。通过调用 DateTimeFormatInfo.GetAllDateTimePatterns(Char) 方法，你可以确定与标准格式字符串对应的自定义格式字符串。下面的示例显示了映射到"d"（短日期模式）标准格式字符串的自定义格式字符串。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      Console.WriteLine("'d' standard format string:");
      foreach (var customString in DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns('d'))
          Console.WriteLine("   {0}", customString);
   }
}
// The example displays the following output:
//       'd' standard format string:
//          M/d/yyyy
//          M/d/yy
//          MM/dd/yy
//          MM/dd/yyyy
//          yy/MM/dd
//          yyyy-MM-dd
//          dd-MMM-yy
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Console.WriteLine("'d' standard format string:")
        For Each customString In DateTimeFormatInfo.CurrentInfo.GetAllDateTimePatterns("d"c)
            Console.WriteLine("   {0}", customString)
        Next
    End Sub
End Module
' The example displays the following output:
'    'd' standard format string:
'       M/d/yyyy
'       M/d/yy
'       MM/dd/yy
'       MM/dd/yyyy
'       yy/MM/dd
'       yyyy-MM-dd
'       dd-MMM-yy
```

以下几节描述了 DateTime 和 DateTimeOffset 值的标准格式说明符。

日期格式

此组包括以下格式：

- 短日期（“d”）格式说明符
- 长日期（“D”）格式说明符

短日期（“d”）格式说明符

“d”标准格式说明符表示通过特定区域性的 DateTimeFormatInfo.ShortDatePattern 属性定义的自定义日期和时间格式字符串。例如，由固定区域性的 ShortDatePattern 属性返回的自定义格式字符串为“MM/dd/yyyy”。

下表列出用于控制返回字符串格式的 DateTimeFormatInfo 对象属性。

PROPERTY	说
ShortDatePattern	定义结果字符串的总体格式。

PROPERTY	说明
DateSeparator	定义用于分隔日期中年、月、日部分的字符串。

下面的示例使用“d”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008,4, 10);
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 4/10/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("en-NZ")));
// Displays 10/04/2008
Console.WriteLine(date1.ToString("d",
                  CultureInfo.CreateSpecificCulture("de-DE")));
// Displays 10.04.2008
```

```
Dim date1 As Date = #4/10/2008#
Console.WriteLine(date1.ToString("d", DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays 4/10/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("en-NZ")))
' Displays 10/04/2008
Console.WriteLine(date1.ToString("d", _
                  CultureInfo.CreateSpecificCulture("de-DE")))
' Displays 10.04.2008
```

返回表首

长日期（“D”）格式说明符

“D”标准格式说明符表示由当前的 DateTimeFormatInfo.LongDatePattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为“dddd, dd MMMM yyyy”。

下表列出了用于控制返回字符串格式的 DateTimeFormatInfo 对象的属性。

PROPERTY	说明
LongDatePattern	定义结果字符串的总体格式。
DayNames	定义可在结果字符串中出现的本地化日名称。
MonthNames	定义可在结果字符串中出现的本地化月份名称。

下面的示例使用“D”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10);
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("pt-BR")));
// Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays jueves, 10 de abril de 2008
```

```
Dim date1 As Date = #4/10/2008#
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("pt-BR")))
' Displays quinta-feira, 10 de abril de 2008
Console.WriteLine(date1.ToString("D", _
                  CultureInfo.CreateSpecificCulture("es-MX")))
' Displays jueves, 10 de abril de 2008
```

返回表首

日期和时间格式

此组包括以下格式：

- 完整日期短时间（“f”）格式说明符
- 完整日期长时间（“F”）格式说明符
- 常规日期短时间（“g”）格式说明符
- 常规日期长时间（“G”）格式说明符
- 往返（“O”、“o”）格式说明符
- RFC1123（“R”、“r”）格式说明符
- 可排序（“s”）格式说明符
- 通用可排序（“u”）格式说明符
- 通用完整（“U”）格式说明符

完整日期短时间（“f”）格式说明符

“f”标准格式说明符表示长日期（“D”）和短时间（“t”）模式的组合，由空格分隔。

结果字符串受特定 DateTimeFormatInfo 对象的格式信息的影响。下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 DateTimeFormatInfo.LongDatePattern 和 DateTimeFormatInfo.ShortTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	“
LongDatePattern	定义结果字符串中日期部分的格式。
ShortTimePattern	定义结果字符串中时间部分的格式。
DayNames	定义可在结果字符串中出现的本地化日名称。
MonthNames	定义可在结果字符串中出现的本地化月份名称。

PROPERTY	说明
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用“f”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("f",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("f", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 6:30 AM
Console.WriteLine(date1.ToString("f", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays jeudi 10 avril 2008 06:30
```

返回表首

完整日期长时间（“F”）格式说明符

“F”标准格式说明符表示由当前的 DateTimeFormatInfo.FullDateTimePattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为“dddd, dd MMMM yyyy HH:mm:ss”。

下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 FullDateTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说明
FullDateTimePattern	定义结果字符串的总体格式。
DayNames	定义可在结果字符串中出现的本地化日名称。
MonthNames	定义可在结果字符串中出现的本地化月份名称。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用“F”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays jeudi 10 avril 2008 06:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("F", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 6:30:00 AM
Console.WriteLine(date1.ToString("F", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays jeudi 10 avril 2008 06:30:00
```

返回表首

常规日期短时间（“g”）格式说明符

“g”标准格式说明符表示短日期（“d”）和短时间（“t”）模式的组合，由空格分隔。

结果字符串受特定 DateTimeFormatInfo 对象的格式信息的影响。下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 DateTimeFormatInfo.ShortDatePattern 和 DateTimeFormatInfo.ShortTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说
ShortDatePattern	定义结果字符串中日期部分的格式。
ShortTimePattern	定义结果字符串中时间部分的格式。
DateSeparator	定义用于分隔日期中年、月、日部分的字符串。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用“g”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("g",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g",
                  CultureInfo.CreateSpecificCulture("fr-BE")));
// Displays 10/04/2008 6:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("g", _
                  DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008 06:30
Console.WriteLine(date1.ToString("g", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 4/10/2008 6:30 AM
Console.WriteLine(date1.ToString("g", _
                  CultureInfo.CreateSpecificCulture("fr-BE")))
' Displays 10/04/2008 6:30
```

返回表首

常规日期长时间("G")格式说明符

"G"标准格式说明符表示短日期("d")和长时间("T")模式的组合，由空格分隔。

结果字符串受特定 DateTimeFormatInfo 对象的格式信息的影响。下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 DateTimeFormatInfo.ShortDatePattern 和 DateTimeFormatInfo.LongTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说
ShortDatePattern	定义结果字符串中日期部分的格式。
LongTimePattern	定义结果字符串中时间部分的格式。
DateSeparator	定义用于分隔日期中年、月、日部分的字符串。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用"G"格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("G",
                  DateTimeFormatInfo.InvariantInfo));
// Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G",
                  CultureInfo.CreateSpecificCulture("nl-BE")));
// Displays 10/04/2008 6:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("G", _
                  DateTimeFormatInfo.InvariantInfo))
' Displays 04/10/2008 06:30:00
Console.WriteLine(date1.ToString("G", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 4/10/2008 6:30:00 AM
Console.WriteLine(date1.ToString("G", _
                  CultureInfo.CreateSpecificCulture("nl-BE")))
' Displays 10/04/2008 6:30:00
```

返回表首

往返（“O”、“o”）格式说明符

“O”或“o”标准格式说明符表示使用保留时区信息的模式的自定义日期和时间格式字符串，并发出符合 ISO8601 的结果字符串。对于 DateTime 值，此格式说明符设计用于在文本中将日期和时间值与 DateTime.Kind 属性一起保留。如果将 DateTime.Parse(String, IFormatProvider, DateTimeStyles) 参数设置为 DateTime.ParseExact，则可通过使用 `styles` 或 DateTimeStyles.RoundtripKind 方法对设置了格式的字符串进行反向分析。

对于 DateTime 值，“O”或“o”标准格式说明符对应于“yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffK”自定义格式字符串，对于 DateTimeOffset 值，“O”或“o”标准格式说明符则对应于“yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffzzz”自定义格式字符串。在此字符串中，分隔各个字符（例如连字符、冒号和字母“T”）的单引号标记对指示各个字符是不能更改的文本。撇号不会出现在输出字符串中。

"O" 或 "o" 标准格式说明符（和 "yyyy'-'MM'-'dd'T'HH':'mm':'ss'.'fffffffK" 自定义格式字符串）利用 ISO 8601 表示时区信息的三种方式，从而暂留 DateTime 值的 Kind 属性：

- DateTimeKind.Local 日期和时间值的时区组件是相对于 UTC 的偏移量（例如，+01:00，-07:00）。所有 DateTimeOffset 值也以这种格式表示。
- DateTimeKind.Utc 日期和时间值的时区组件使用“Z”（它代表零偏移量）以表示 UTC。
- DateTimeKind.Unspecified 日期和时间值没有时区信息。

由于“O”或“o”标准格式说明符遵循国际标准，使用说明符的格式设置或分析操作始终使用固定区域性和公历。

如果字符串采用了这些格式中的某个格式，则可以通过使用“O”或“o”格式说明符分析传递到 `Parse` 和 `TryParse` 的 `ParseExact`、`TryParseExact`、DateTime 和 DateTimeOffset 方法的这些字符串。对于 DateTime 对象，你调用的分析重载还应当包含带有 `styles` 值的 DateTimeStyles.RoundtripKind 参数。请注意，如果你使用对应于“O”或“o”格式说明符的自定义格式字符串调用分析方法，则你不会获得与“O”或“o”相同的结果。这是因为使用自定义格式字符串的分析方法不能分析缺少时区组件的日期和时间值的字符串表示形式，或使用“Z”指示 UTC。

下面的示例使用“o”格式说明符在美国太平洋时区中的系统上显示一系列 DateTime 值和 DateTimeOffset 值。

```
using System;

public class Example
{
   public static void Main()
   {
       DateTime dat = new DateTime(2009, 6, 15, 13, 45, 30,
                                   DateTimeKind.Unspecified);
       Console.WriteLine("{0} ({1}) --> {0:O}", dat, dat.Kind);

       DateTime uDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Utc);
       Console.WriteLine("{0} ({1}) --> {0:O}", uDat, uDat.Kind);

       DateTime lDat = new DateTime(2009, 6, 15, 13, 45, 30,
                                    DateTimeKind.Local);
       Console.WriteLine("{0} ({1}) --> {0:O}\n", lDat, lDat.Kind);

       DateTimeOffset dto = new DateTimeOffset(lDat);
       Console.WriteLine("{0} --> {0:O}", dto);
   }
}
// The example displays the following output:
//    6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
//    6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
//    6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
//
//    6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

```
Module Example
    Public Sub Main()
        Dim dat As New Date(2009, 6, 15, 13, 45, 30,
                            DateTimeKind.Unspecified)
        Console.WriteLine("{0} ({1}) --> {0:O}", dat, dat.Kind)

        Dim uDat As New Date(2009, 6, 15, 13, 45, 30, DateTimeKind.Utc)
        Console.WriteLine("{0} ({1}) --> {0:O}", uDat, uDat.Kind)

        Dim lDat As New Date(2009, 6, 15, 13, 45, 30, DateTimeKind.Local)
        Console.WriteLine("{0} ({1}) --> {0:O}", lDat, lDat.Kind)
        Console.WriteLine()

        Dim dto As New DateTimeOffset(lDat)
        Console.WriteLine("{0} --> {0:O}", dto)
    End Sub
End Module
' The example displays the following output:
'    6/15/2009 1:45:30 PM (Unspecified) --> 2009-06-15T13:45:30.0000000
'    6/15/2009 1:45:30 PM (Utc) --> 2009-06-15T13:45:30.0000000Z
'    6/15/2009 1:45:30 PM (Local) --> 2009-06-15T13:45:30.0000000-07:00
'
'    6/15/2009 1:45:30 PM -07:00 --> 2009-06-15T13:45:30.0000000-07:00
```

下面的示例使用“O”格式说明符创建格式字符串，然后通过调用日期和时间 `Parse` 方法还原原始日期和时间值。

```
// Round-trip DateTime values.
DateTime originalDate, newDate;
string dateString;
// Round-trip a local time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 10, 6, 30, 0), DateTimeKind.Local);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);
// Round-trip a UTC time.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 12, 9, 30, 0), DateTimeKind.Utc);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);
// Round-trip time in an unspecified time zone.
originalDate = DateTime.SpecifyKind(new DateTime(2008, 4, 13, 12, 30, 0), DateTimeKind.Unspecified);
dateString = originalDate.ToString("o");
newDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind,
                  newDate, newDate.Kind);

// Round-trip a DateTimeOffset value.
DateTimeOffset originalDTO = new DateTimeOffset(2008, 4, 12, 9, 30, 0, new TimeSpan(-8, 0, 0));
dateString = originalDTO.ToString("o");
DateTimeOffset newDTO = DateTimeOffset.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Round-tripped {0} to {1}.", originalDTO, newDTO);
// The example displays the following output:
//    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM Local.
//    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
//    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00 PM Unspecified.
//    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM -08:00.
```

```
' Round-trip DateTime values.
Dim originalDate, newDate As Date
Dim dateString As String
' Round-trip a local time.
originalDate = Date.SpecifyKind(#4/10/2008 6:30AM#, DateTimeKind.Local)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)
' Round-trip a UTC time.
originalDate = Date.SpecifyKind(#4/12/2008 9:30AM#, DateTimeKind.Utc)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)
' Round-trip time in an unspecified time zone.
originalDate = Date.SpecifyKind(#4/13/2008 12:30PM#, DateTimeKind.Unspecified)
dateString = originalDate.ToString("o")
newDate = Date.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} {1} to {2} {3}.", originalDate, originalDate.Kind, _
                  newDate, newDate.Kind)

' Round-trip a DateTimeOffset value.
Dim originalDTO As New DateTimeOffset(#4/12/2008 9:30AM#, New TimeSpan(-8, 0, 0))
dateString = originalDTO.ToString("o")
Dim newDTO As DateTimeOffset = DateTimeOffset.Parse(dateString, Nothing, DateTimeStyles.RoundtripKind)
Console.WriteLine("Round-tripped {0} to {1}.", originalDTO, newDTO)
' The example displays the following output:
'    Round-tripped 4/10/2008 6:30:00 AM Local to 4/10/2008 6:30:00 AM Local.
'    Round-tripped 4/12/2008 9:30:00 AM Utc to 4/12/2008 9:30:00 AM Utc.
'    Round-tripped 4/13/2008 12:30:00 PM Unspecified to 4/13/2008 12:30:00 PM Unspecified.
'    Round-tripped 4/12/2008 9:30:00 AM -08:00 to 4/12/2008 9:30:00 AM -08:00.
```

返回表首

RFC1123（“R”、“r”）格式说明符

“R”或“r”标准格式说明符表示由 DateTimeFormatInfo.RFC1123Pattern 属性定义的自定义日期和时间格式字符串。该模式反映已定义的标准，并且属性是只读的。因此，无论所使用的区域性或所提供的格式提供程序是什么，它总是相同的。定义格式字符串为“ddd, dd MMM yyyy HH':'mm':'ss 'GMT'”。当使用此标准格式说明符时，格式设置或分析操作始终使用固定区域性。

结果字符串受由 DateTimeFormatInfo 属性（该属性表示固定区域性）返回的 DateTimeFormatInfo.InvariantInfo 对象的下列属性的影响。

PROPERTY	说明
RFC1123Pattern	定义结果字符串的格式。
AbbreviatedDayNames	定义可在结果字符串中出现的缩写的日期名称。
AbbreviatedMonthNames	定义可在结果字符串中出现的缩写的月份名称。

尽管 RFC 1123 标准将时间表示为协调世界时 (UTC)，格式设置操作也不会修改正在格式化的 DateTime 对象的值。因此，执行格式设置操作之前，必须通过调用 DateTime 方法将 DateTime.ToUniversalTime 值转换为 UTC。相反，DateTimeOffset 值自动执行此转换；即执行格式设置操作之前无需调用 DateTimeOffset.ToUniversalTime 方法。

下面的示例使用“r”格式说明符在美国太平洋时区中的系统上显示 DateTime 和 DateTimeOffset 值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
DateTimeOffset dateOffset = new DateTimeOffset(date1,
                            TimeZoneInfo.Local.GetUtcOffset(date1));
Console.WriteLine(date1.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime().ToString("r"));
// Displays Thu, 10 Apr 2008 13:30:00 GMT
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Dim dateOffset As New DateTimeOffset(date1, TimeZoneInfo.Local.GetUtcOffset(date1))
Console.WriteLine(date1.ToUniversalTime.ToString("r"))
' Displays Thu, 10 Apr 2008 13:30:00 GMT
Console.WriteLine(dateOffset.ToUniversalTime.ToString("r"))
' Displays Thu, 10 Apr 2008 13:30:00 GMT
```

返回表首

可排序（“s”）格式说明符

“s”标准格式说明符表示由 DateTimeFormatInfo.SortableDateTimePattern 属性定义的自定义日期和时间格式字符串。该模式反映已定义的标准 (ISO 8601)，并且属性是只读的。因此，无论所使用的区域性或所提供的格式提供程序是什么，它总是相同的。自定义格式字符串为“yyyy'-'MM'-'dd'T'HH':'mm':'ss”。

使用“s”格式说明符的目的是使生成的结果字符串基于日期和时间值一致按升序或降序顺序进行排序。因此，尽管“s”标准格式说明符采用一致格式表示日期和时间值，但是格式化操作不会修改正在格式化以反映其 DateTime.Kind 属性或 DateTimeOffset.Offset 值的日期和时间对象的值。例如，通过格式化日期和时间值 2014-11-15T18:32:17+00:00 和 2014-11-15T18:32:17+08:00 生成的结果字符串完全相同。

当使用此标准格式说明符时，格式设置或分析操作始终使用固定区域性。

下面的示例使用"s"格式说明符在美国太平洋时区中的系统上显示 DateTime 和 DateTimeOffset 值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("s"));
// Displays 2008-04-10T06:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("s"))
' Displays 2008-04-10T06:30:00
```

返回表首

通用可排序("u")格式说明符

"u"标准格式说明符表示由 DateTimeFormatInfo.UniversalSortableDateTimePattern 属性定义的自定义日期和时间格式字符串。该模式反映已定义的标准，并且属性是只读的。因此，无论所使用的区域性或所提供的格式提供程序是什么，它总是相同的。自定义格式字符串为"yyyy'-'MM'-'dd HH':'mm':'ss'Z'"。当使用此标准格式说明符时，格式设置或分析操作始终使用固定区域性。

尽管结果字符串应将时间表达为协调世界时 (UTC)，但在格式设置操作过程中不转换原始 DateTime 值。因此，在对 DateTime 值进行格式设置之前，必须通过调用 DateTime.ToUniversalTime 方法将该值转换为 UTC。相反，DateTimeOffset 值自动执行此转换；即执行格式设置操作之前无需调用 DateTimeOffset.ToUniversalTime 方法。

下面的示例使用"u"格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToUniversalTime().ToString("u"));
// Displays 2008-04-10 13:30:00Z
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToUniversalTime.ToString("u"))
' Displays 2008-04-10 13:30:00Z
```

返回表首

通用完整("U")格式说明符

"U"标准格式说明符表示由特定区域性的 DateTimeFormatInfo.FullDateTimePattern 属性定义的自定义日期和时间格式字符串。此模式与"F"模式相同。但是，在对 DateTime 值进行格式设置之前，该值自动转换为 UTC。

下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 FullDateTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说明
FullDateTimePattern	定义结果字符串的总体格式。
DayNames	定义可在结果字符串中出现的本地化日名称。
MonthNames	定义可在结果字符串中出现的本地化月份名称。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。

PROPERTY	说明
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

DateTimeOffset 类型不支持"U"格式说明符。如果使用"U"格式说明符来设置 FormatException 值的格式，则将引发 DateTimeOffset。

下面的示例使用"U"格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U",
                  CultureInfo.CreateSpecificCulture("sv-FI")));
// Displays den 10 april 2008 13:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("U", CultureInfo.CreateSpecificCulture("en-US")))
' Displays Thursday, April 10, 2008 1:30:00 PM
Console.WriteLine(date1.ToString("U", CultureInfo.CreateSpecificCulture("sv-FI")))
' Displays den 10 april 2008 13:30:00
```

返回表首

时间格式

此组包括以下格式：

- 短时间("t")格式说明符
- 长时间("T")格式说明符

短时间("t")格式说明符

"t"标准格式说明符表示由当前的 DateTimeFormatInfo.ShortTimePattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为"HH:mm"。

结果字符串受特定 DateTimeFormatInfo 对象的格式信息的影响。下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 DateTimeFormatInfo.ShortTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说明
ShortTimePattern	定义结果字符串中时间部分的格式。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。

PROPERTY	说明
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用“t”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30 AM
Console.WriteLine(date1.ToString("t",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("t", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 6:30 AM
Console.WriteLine(date1.ToString("t", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays 6:30
```

返回表首

长时间（“T”）格式说明符

“T”标准格式说明符表示由特定区域性的 DateTimeFormatInfo.LongTimePattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为“HH:mm:ss”。

下表列出了 DateTimeFormatInfo 对象属性，这些属性可控制返回字符串的格式。由某些区域性的 DateTimeFormatInfo.LongTimePattern 属性返回的自定义格式说明符可能未利用所有属性。

PROPERTY	说明
LongTimePattern	定义结果字符串中时间部分的格式。
TimeSeparator	定义分隔时间中小时、分钟和秒钟几个组成部分的字符串。
AMDesignator	定义以 12 小时时钟制表示午夜至正午之前这段时间的字符串。
PMDesignator	定义以 12 小时时钟制表示正午至午夜之前这段时间的字符串。

下面的示例使用“T”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T",
                  CultureInfo.CreateSpecificCulture("es-ES")));
// Displays 6:30:00
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("T", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays 6:30:00 AM
Console.WriteLine(date1.ToString("T", _
                  CultureInfo.CreateSpecificCulture("es-ES")))
' Displays 6:30:00
```

返回表首

部分日期格式

此组包括以下格式：

- 月（“M”、“m”）格式说明符
- 年月（“Y”、“y”）格式说明符

月（“M”、“m”）格式说明符

“M”或“m”标准格式说明符表示由当前的 DateTimeFormatInfo.MonthDayPattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为“MMMM dd”。

下表列出用于控制返回字符串格式的 DateTimeFormatInfo 对象属性。

PROPERTY	说
MonthDayPattern	定义结果字符串的总体格式。
MonthNames	定义可在结果字符串中出现的本地化月份名称。

下面的示例使用“m”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("en-us")));
// Displays April 10
Console.WriteLine(date1.ToString("m",
                  CultureInfo.CreateSpecificCulture("ms-MY")));
// Displays 10 April
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("m", _
                  CultureInfo.CreateSpecificCulture("en-us")))
' Displays April 10
Console.WriteLine(date1.ToString("m", _
                  CultureInfo.CreateSpecificCulture("ms-MY")))
' Displays 10 April
```

返回表首

年月（“Y”、“y”）格式说明符

“Y”或“y”标准格式说明符表示由指定区域性的 DateTimeFormatInfo.YearMonthPattern 属性定义的自定义日期和时间格式字符串。例如，用于固定区域性的自定义格式字符串为“yyyy MMMM”。

下表列出用于控制返回字符串格式的 DateTimeFormatInfo 对象属性。

PROPERTY	此
YearMonthPattern	定义结果字符串的总体格式。
MonthNames	定义可在结果字符串中出现的本地化月份名称。

下面的示例使用“y”格式说明符来显示日期和时间值。

```
DateTime date1 = new DateTime(2008, 4, 10, 6, 30, 0);
Console.WriteLine(date1.ToString("Y",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays April, 2008
Console.WriteLine(date1.ToString("y",
                  CultureInfo.CreateSpecificCulture("af-ZA")));
// Displays April 2008
```

```
Dim date1 As Date = #4/10/2008 6:30AM#
Console.WriteLine(date1.ToString("Y", CultureInfo.CreateSpecificCulture("en-US")))
' Displays April, 2008
Console.WriteLine(date1.ToString("y", CultureInfo.CreateSpecificCulture("af-ZA")))
' Displays April 2008
```

返回表首

控制面板设置

Windows 控制面板中“区域和语言选项”项的设置会影响由格式化操作产生的结果字符串。这些设置用于初始化与当前线程区域性关联的 DateTimeFormatInfo 对象，当前线程区域性提供用于控制格式设置的值。使用不同设置的计算机将生成不同的结果字符串。

此外，如果使用 CultureInfo(String) 构造函数实例化一个新的 CultureInfo 对象以表示与当前的系统区域性相同的区域性，则通过控制面板中的“**区域和语言选项**”建立的任何自定义都将应用到新的 CultureInfo 对象。可以使用 CultureInfo(String, Boolean) 构造函数来创建不会反映系统的自定义项的 CultureInfo 对象。

DateTimeFormatInfo 属性

格式化受当前的 DateTimeFormatInfo 对象的属性影响，其由当前线程区域性隐式提供或由调用格式化的方法的 IFormatProvider 参数显式提供。对于 IFormatProvider 参数，应用程序应指定一个表示区域性的 CultureInfo 对象或表示特定区域性的日期和时间格式设置约定的 DateTimeFormatInfo 对象。许多标准日期和时间格式说明符是由当前的 DateTimeFormatInfo 对象的属性定义的格式设置模式的别名。应用程序通过更改相应 DateTimeFormatInfo 属性的相应日期和时间格式模式，可以更改由某些标准日期和时间格式说明符产生的结果。

请参阅

- System.DateTime
- System.DateTimeOffset
- 格式设置类型
- 自定义日期和时间格式字符串
- 示例：.NET Core WinForms 格式设置实用工具 (C#)
- 示例：.NET Core WinForms 格式设置实用工具 (Visual Basic)

自定义日期和时间格式字符串

2021/11/16 •

日期和时间格式字符串定义由格式设置操作生成的 DateTime 或 DateTimeOffset 值的文本表示形式。它还可定义分析操作中需要的日期和时间值的表示形式，以便成功将字符串转换为日期和时间。自定义格式字符串由一个或多个自定义日期和时间格式说明符组成。任何不是标准日期和时间格式字符串的字符串都会解释为自定义日期和时间格式字符串。

> **TIP**
> 你可以下载格式设置实用工具，它属于一种 .NET Core Windows 窗体应用程序，通过该应用程序，你可将格式字符串应用于数值或日期和时间值并显示结果字符串。源代码适用于 C# 和 Visual Basic。

自定义日期和时间格式字符串可以与 DateTime 和 DateTimeOffset 值一起使用。

> **NOTE**
> 本文中的一些 C# 示例在 Try.NET 内联代码运行程序和演练环境中运行。选择“运行”按钮以在交互窗口中运行示例 。执行代码后，可通过再次选择“运行”来修改它并运行已修改的代码 。已修改的代码要么在交互窗口中运行，要么编译失败时，交互窗口将显示所有 C# 编译器错误消息。
>
> Try.NET 内联代码运行程序和演练环境的本地时区是协调世界时 (UTC)。这可能会影响用于说明 DateTime、DateTimeOffset 和 TimeZoneInfo 类型及其成员的示例的行为和输出。

在格式设置操作中，可将自定义日期和时间格式字符串与日期和时间实例的 `ToString` 方法或支持复合格式设置的方法结合使用。下面的示例演示了这两种用法。

```
DateTime thisDate1 = new DateTime(2011, 6, 10);
Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".");

DateTimeOffset thisDate2 = new DateTimeOffset(2011, 6, 10, 15, 24, 16,
                                              TimeSpan.Zero);
Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                  thisDate2);
// The example displays the following output:
//    Today is June 10, 2011.
//    The current date and time: 06/10/11 15:24:16 +00:00
```

```
Dim thisDate1 As Date = #6/10/2011#
Console.WriteLine("Today is " + thisDate1.ToString("MMMM dd, yyyy") + ".")

Dim thisDate2 As New DateTimeOffset(2011, 6, 10, 15, 24, 16, TimeSpan.Zero)
Console.WriteLine("The current date and time: {0:MM/dd/yy H:mm:ss zzz}",
                  thisDate2)
' The example displays the following output:
'    Today is June 10, 2011.
'    The current date and time: 06/10/11 15:24:16 +00:00
```

在分析操作中，自定义日期和时间格式字符串可用于 DateTime.ParseExact、DateTime.TryParseExact、DateTimeOffset.ParseExact 和 DateTimeOffset.TryParseExact 方法。这些方法需要一个完全符合使分析操作成功所需的特定模式的输入字符串。下面的示例演示对 DateTimeOffset.ParseExact(String, String, IFormatProvider) 方法的调用，以分析必须包括日、月和两位数年份的

日期。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] dateValues = { "30-12-2011", "12-30-2011",
                              "30-12-11", "12-30-11" };
      string pattern = "MM-dd-yy";
      DateTime parsedDate;

      foreach (var dateValue in dateValues) {
         if (DateTime.TryParseExact(dateValue, pattern, null,
                                    DateTimeStyles.None, out parsedDate))
            Console.WriteLine("Converted '{0}' to {1:d}.",
                              dateValue, parsedDate);
         else
            Console.WriteLine("Unable to convert '{0}' to a date and time.",
                              dateValue);
      }
   }
}
// The example displays the following output:
//    Unable to convert '30-12-2011' to a date and time.
//    Unable to convert '12-30-2011' to a date and time.
//    Unable to convert '30-12-11' to a date and time.
//    Converted '12-30-11' to 12/30/2011.
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim dateValues() As String = {"30-12-2011", "12-30-2011",
                                      "30-12-11", "12-30-11"}
        Dim pattern As String = "MM-dd-yy"
        Dim parsedDate As Date

        For Each dateValue As String In dateValues
            If DateTime.TryParseExact(dateValue, pattern, Nothing,
                                      DateTimeStyles.None, parsedDate) Then
                Console.WriteLine("Converted '{0}' to {1:d}.",
                                  dateValue, parsedDate)
            Else
                Console.WriteLine("Unable to convert '{0}' to a date and time.",
                                  dateValue)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'    Unable to convert '30-12-2011' to a date and time.
'    Unable to convert '12-30-2011' to a date and time.
'    Unable to convert '30-12-11' to a date and time.
'    Converted '12-30-11' to 12/30/2011.
```

下表描述自定义日期和时间格式说明符并显示由每个格式说明符生成的结果字符串。默认情况下，结果字符串反映 zh-cn 区域性的格式设置约定。如果特定格式说明符生成本地化结果字符串，则该示例还注明结果字符串适用的区域性。有关使用自定义日期和时间格式字符串的详细信息，请参阅备注部分。

ttttt	tt	tt
"d"	一个月中的某一天(1 到 31)。 更多信息:"d"自定义格式说明符。	2009-06-01T13:45:30 -> 1 2009-06-15T13:45:30 -> 15
"dd"	一个月中的某一天(01 到 31)。 更多信息:"dd"自定义格式说明符。	2009-06-01T13:45:30 -> 01 2009-06-15T13:45:30 -> 15
"ddd"	一周中某天的缩写名称。 更多信息:"ddd"自定义格式说明符。	2009-06-15T13:45:30 -> Mon (en-US) 2009-06-15T13:45:30 -> Пн (ru-RU) 2009-06-15T13:45:30 -> lun. (fr-FR)
"dddd"	一周中某天的完整名称。 更多信息:"dddd"自定义格式说明符。	2009-06-15T13:45:30 -> Monday (en-US) 2009-06-15T13:45:30 -> понедельник (ru-RU) 2009-06-15T13:45:30 -> lundi (fr-FR)
"f"	日期和时间值的十分之几秒。 更多信息:"f"自定义格式说明符。	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.05 -> 0
"ff"	日期和时间值的百分之几秒。 更多信息:"ff"自定义格式说明符。	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 -> 00
"fff"	日期和时间值的千分之几秒。 更多信息:"fff"自定义格式说明符。	6/15/2009 13:45:30.617 -> 617 6/15/2009 13:45:30.0005 -> 000
"ffff"	日期和时间值的万分之几秒。 更多信息:"ffff"自定义格式说明符。	2009-06-15T13:45:30.6175000 -> 6175 2009-06-15T13:45:30.0000500 -> 0000
"fffff"	日期和时间值的十万分之几秒。 更多信息:"fffff"自定义格式说明符。	2009-06-15T13:45:30.6175400 -> 61754 6/15/2009 13:45:30.000005 -> 00000
"ffffff"	日期和时间值的百万分之几秒。 更多信息:"ffffff"自定义格式说明符。	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 -> 000000
"fffffff"	日期和时间值的千万分之几秒。 更多信息:"fffffff"自定义格式说明符。	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 0001150

tttttt	tt	tt
“F”	如果非零，则为日期和时间值的十分之几秒。 更多信息：“F”自定义格式说明符。	2009-06-15T13:45:30.6170000 -> 6 2009-06-15T13:45:30.0500000 ->（无输出）
“FF”	如果非零，则为日期和时间值的百分之几秒。 更多信息：“FF”自定义格式说明符。	2009-06-15T13:45:30.6170000 -> 61 2009-06-15T13:45:30.0050000 ->（无输出）
“FFF”	如果非零，则为日期和时间值的千分之几秒。 更多信息：“FFF”自定义格式说明符。	2009-06-15T13:45:30.6170000 -> 617 2009-06-15T13:45:30.0005000 ->（无输出）
“FFFF”	如果非零，则为日期和时间值的万分之几秒。 更多信息：“FFFF”自定义格式说明符。	2009-06-15T13:45:30.5275000 -> 5275 2009-06-15T13:45:30.0000500 ->（无输出）
“FFFFF”	如果非零，则为日期和时间值的十万分之几秒。 更多信息：“FFFFF”自定义格式说明符。	2009-06-15T13:45:30.6175400 -> 61754 2009-06-15T13:45:30.0000050 ->（无输出）
“FFFFFF”	如果非零，则为日期和时间值的百万分之几秒。 更多信息：“FFFFFF”自定义格式说明符。	2009-06-15T13:45:30.6175420 -> 617542 2009-06-15T13:45:30.0000005 ->（无输出）
“FFFFFFF”	如果非零，则为日期和时间值的千万分之几秒。 更多信息：“FFFFFFF”自定义格式说明符。	2009-06-15T13:45:30.6175425 -> 6175425 2009-06-15T13:45:30.0001150 -> 000115
“g”、“gg”	时期或纪元。 更多信息：“g”或“gg”自定义格式说明符。	2009-06-15T13:45:30.6170000 -> A.D.
“h”	采用 12 小时制的小时（从 1 到 12）。 更多信息：“h”自定义格式说明符。	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 1
“hh”	采用 12 小时制的小时（从 01 到 12）。 更多信息：“hh”自定义格式说明符。	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 01
“H”	采用 24 小时制的小时（从 0 到 23）。 更多信息：“H”自定义格式说明符。	2009-06-15T01:45:30 -> 1 2009-06-15T13:45:30 -> 13

“”“”	“”	“”
“HH”	采用 24 小时制的小时(从 00 到 23)。 更多信息:“HH”自定义格式说明符。	2009-06-15T01:45:30 -> 01 2009-06-15T13:45:30 -> 13
“K”	时区信息。 更多信息:“K”自定义格式说明符。	带 DateTime 值: 2009-06-15T13:45:30, Kind Unspecified -> 2009-06-15T13:45:30, Kind Utc -> Z 2009-06-15T13:45:30, Kind Local -> -07:00(取决于本地计算机的设置) 带 DateTimeOffset 值: 2009-06-15T01:45:30-07:00 --> -07:00 2009-06-15T08:45:30+00:00 --> +00:00
“m”	分钟(0 到 59)。 更多信息:“m”自定义格式说明符。	2009-06-15T01:09:30 -> 9 2009-06-15T13:29:30 -> 29
“mm”	分钟(00 到 59)。 更多信息:“mm”自定义格式说明符。	2009-06-15T01:09:30 -> 09 2009-06-15T01:45:30 -> 45
“M”	月份(1 到 12)。 更多信息:“M”自定义格式说明符。	2009-06-15T13:45:30 -> 6
“MM”	月份(1 到 12)。 更多信息:“MM”自定义格式说明符。	2009-06-15T13:45:30 -> 06
“MMM”	月份的缩写名称。 更多信息:“MMM”自定义格式说明符。	2009-06-15T13:45:30 -> Jun (en-US) 2009-06-15T13:45:30 -> juin (fr-FR) 2009-06-15T13:45:30 -> Jun (zu-ZA)
“MMMM”	月份的完整名称。 更多信息:“MMMM”自定义格式说明符。	2009-06-15T13:45:30 -> June (en-US) 2009-06-15T13:45:30 -> juni (da-DK) 2009-06-15T13:45:30 -> uJuni (zu-ZA)
“s”	秒(0 到 59)。 更多信息:“s”自定义格式说明符。	2009-06-15T13:45:09 -> 9

[illegible]	[illegible]	[illegible]
“ss”	秒（00 到 59）。 更多信息：“ss”自定义格式说明符。	2009-06-15T13:45:09 -> 09
“t”	AM/PM 指示符的第一个字符。 更多信息：“t”自定义格式说明符。	2009-06-15T13:45:30 -> P (en-US) 2009-06-15T13:45:30 -> 午 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
“tt”	AM/PM 指示符。 更多信息：“tt”自定义格式说明符。	2009-06-15T13:45:30 -> PM (en-US) 2009-06-15T13:45:30 -> 午後 (ja-JP) 2009-06-15T13:45:30 -> (fr-FR)
“y”	年份（0 到 99）。 更多信息：“y”自定义格式说明符。	0001-01-01T00:00:00 -> 1 0900-01-01T00:00:00 -> 0 1900-01-01T00:00:00 -> 0 2009-06-15T13:45:30 -> 9 2019-06-15T13:45:30 -> 19
“yy”	年份（00 到 99）。 更多信息：“yy”自定义格式说明符。	0001-01-01T00:00:00 -> 01 0900-01-01T00:00:00 -> 00 1900-01-01T00:00:00 -> 00 2019-06-15T13:45:30 -> 19
“yyy”	年份（最少三位数字）。 更多信息：“yyy”自定义格式说明符。	0001-01-01T00:00:00 -> 001 0900-01-01T00:00:00 -> 900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
“yyyy”	由四位数字表示的年份。 更多信息：“yyyy”自定义格式说明符。	0001-01-01T00:00:00 -> 0001 0900-01-01T00:00:00 -> 0900 1900-01-01T00:00:00 -> 1900 2009-06-15T13:45:30 -> 2009
“yyyyy”	由五位数字表示的年份。 更多信息：“yyyyy”自定义格式说明符。	0001-01-01T00:00:00 -> 00001 2009-06-15T13:45:30 -> 02009
“z”	相对于 UTC 的小时偏移量，无前导零。 更多信息：“z”自定义格式说明符。	2009-06-15T13:45:30-07:00 -> -7

tttttt	tt	tt
“zz”	相对于 UTC 的小时偏移量，带有表示一位数值的前导零。 更多信息：“zz”自定义格式说明符。	2009-06-15T13:45:30-07:00 -> -07
“zzz”	相对于 UTC 的小时和分钟偏移量。 更多信息：“zzz”自定义格式说明符。	2009-06-15T13:45:30-07:00 -> -07:00
":"	时间分隔符。 更多信息：“:”自定义格式说明符。	2009-06-15T13:45:30 -> : (en-US) 2009-06-15T13:45:30 -> . (it-IT) 2009-06-15T13:45:30 -> : (ja-JP)
"/"	日期分隔符。 详细信息：“/”自定义格式说明符。	2009-06-15T13:45:30 -> / (en-US) 2009-06-15T13:45:30 -> - (ar-DZ) 2009-06-15T13:45:30 -> . (tr-TR)
"string" 'string'	文本字符串分隔符。 更多信息：字符文本。	2009-06-15T13:45:30 ("arr:" h:m t) -> arr:1:45 P 2009-06-15T13:45:30 ('arr:' h:m t) -> arr:1:45 P
%	将下面的字符定义为自定义格式说明符。 有关详细信息，请参阅使用单个自定义格式说明符。	2009-06-15T13:45:30 (%h) -> 1
\	转义字符。 更多信息：字符文本和使用转义字符。	2009-06-15T13:45:30 (h \h) -> 1 h
任何其他字符	字符将复制到未更改的结果字符串。 更多信息：字符文本。	2009-06-15T01:45:30 (arr hh:mm t) -> arr 01:45 A

以下各节提供有关每个自定义日期和时间格式说明符的附加信息。除非另行说明，否则，每个说明符将生成相同的字符串表示形式，这与它是与 DateTime 值一起使用还是与 DateTimeOffset 值一起使用无关。

日期“d”格式说明符

“d”自定义格式说明符

“d”自定义格式说明符将一个月中的某一天表示为从 1 到 31 的数字。一位数的日期设置为不带前导零的格式。

如果使用“d”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“d”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在几个格式字符串中包含“d”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("d, M",
                  CultureInfo.InvariantCulture));
// Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays 29 August
Console.WriteLine(date1.ToString("d MMMM",
                  CultureInfo.CreateSpecificCulture("es-MX")));
// Displays 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("d, M", _
                  CultureInfo.InvariantCulture))
' Displays 29, 8

Console.WriteLine(date1.ToString("d MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays 29 August
Console.WriteLine(date1.ToString("d MMMM", _
                  CultureInfo.CreateSpecificCulture("es-MX")))
' Displays 29 agosto
```

返回表首

“dd”自定义格式说明符

“dd”自定义格式字符串将一个月中的某一天表示为从 01 到 31 的数字。一位数的日期设置为带有前导零的格式。

下面的示例在一个自定义格式字符串中包含“dd”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

```
Dim date1 As Date = #1/2/2008 6:30:15AM#

Console.WriteLine(date1.ToString("dd, MM", _
                  CultureInfo.InvariantCulture))
' 02, 01
```

返回表首

“ddd”自定义格式说明符

“ddd”自定义格式说明符表示一周中某天的缩写名称。一周中某天的本地化缩写名称通过当前或指定区域性的 DateTimeFormatInfo.AbbreviatedDayNames 属性进行检索。

下面的示例在一个自定义格式字符串中包含“ddd”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays ven. 29 août
```

返回表首

“dddd”自定义格式说明符

“dddd”自定义格式说明符（以及任意数量的附加“d”说明符）表示一周中某天的完整名称。一周中某天的本地化名称通过当前或指定区域性的 DateTimeFormatInfo.DayNames 属性进行检索。

下面的示例在一个自定义格式字符串中包含“dddd”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("it-IT")))
' Displays venerdì 29 agosto
```

返回表首

小写秒“f”分数说明符

“f”自定义格式说明符

“f”自定义格式说明符表示秒部分的最高有效位；也就是说，它表示日期和时间值的十分之几秒。

如果使用“f”格式说明符而没有其他格式说明符，则将该说明符解释为“f”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

当使用作为格式字符串的一部分提供给 ParseExact、TryParseExact、ParseExact 或 TryParseExact 方法的“f”格式说明符时，所用的“f”格式说明符的数目指示为成功分析字符串而必须呈现的秒部分的最高有效位的数目。

下面的示例在一个自定义格式字符串中包含“f”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“ff”自定义格式说明符

“ff”自定义格式说明符表示秒部分的前两个有效位;也就是说，它表示日期和时间值的百分之几秒。

下面的示例在一个自定义格式字符串中包含“ff”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“fff”自定义格式说明符

“fff”自定义格式说明符表示秒部分的前三个有效位；也就是说，它表示日期和时间值的毫秒。

下面的示例在一个自定义格式字符串中包含“fff”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“ffff”自定义格式说明符

“ffff”自定义格式说明符表示秒部分的前四个有效位；也就是说，它表示日期和时间值的万分之几秒。

尽管可以显示时间值的秒部分的万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟

的分辨率。在 Windows NT 3.5 版和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“fffff”自定义格式说明符

“fffff”自定义格式说明符表示秒部分的前五个有效位；也就是说，它表示日期和时间值的十万分之几秒。

尽管可以显示时间值的秒部分的十万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“ffffff”自定义格式说明符

“ffffff”自定义格式说明符表示秒部分的前六个有效位；也就是说，它表示日期和时间值的百万分之几秒。

尽管可以显示时间值的秒部分的百万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“fffffff”自定义格式说明符

“fffffff”自定义格式说明符表示秒部分的前七个有效位；也就是说，它表示日期和时间值的千万分之几秒。

尽管可以显示时间值的秒部分的千万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

大写秒“F”分数说明符

“F”自定义格式说明符

“F”自定义格式说明符表示秒部分的最高有效位；也就是说，它表示日期和时间值的十分之几秒。如果该数字为零，则不显示任何内容。

如果使用“F”格式说明符而没有其他格式说明符，则将该说明符解释为“F”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

用于 ParseExact、TryParseExact、ParseExact 或 TryParseExact 方法的“F”格式说明符的数目指示为成功分析字符串而可以呈现的秒部分的最高有效位的最大数目。

下面的示例在一个自定义格式字符串中包含“F”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“FF”自定义格式说明符

“FF”自定义格式说明符表示秒部分的前两个有效位；也就是说，它表示日期和时间值的百分之几秒。但不显示尾随零或两个零位。

下面的示例在一个自定义格式字符串中包含“FF”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“FFF”自定义格式说明符

“FFF”自定义格式说明符表示秒部分的前三个有效位；也就是说，它表示日期和时间值的毫秒。但不显示尾随零或三个零位。

下面的示例在一个自定义格式字符串中包含“FFF”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15, 18);
CultureInfo ci = CultureInfo.InvariantCulture;

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci));
// Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci));
// Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci));
// Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci));
// Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci));
// Displays 07:27:15.018
```

```
Dim date1 As New Date(2008, 8, 29, 19, 27, 15, 018)
Dim ci As CultureInfo = CultureInfo.InvariantCulture

Console.WriteLine(date1.ToString("hh:mm:ss.f", ci))
' Displays 07:27:15.0
Console.WriteLine(date1.ToString("hh:mm:ss.F", ci))
' Displays 07:27:15
Console.WriteLine(date1.ToString("hh:mm:ss.ff", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.FF", ci))
' Displays 07:27:15.01
Console.WriteLine(date1.ToString("hh:mm:ss.fff", ci))
' Displays 07:27:15.018
Console.WriteLine(date1.ToString("hh:mm:ss.FFF", ci))
' Displays 07:27:15.018
```

返回表首

“FFFF”自定义格式说明符

“FFFF”自定义格式说明符表示秒部分的前四个有效位；也就是说，它表示日期和时间值的万分之几秒。但不显示尾随零或四个零位。

尽管可以显示时间值的秒部分的万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“FFFFF”自定义格式说明符

“FFFFF”自定义格式说明符表示秒部分的前五个有效位；也就是说，它表示日期和时间值的十万分之几秒。但不显示尾随零或五个零位。

尽管可以显示时间值的秒部分的十万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“FFFFFF”自定义格式说明符

“FFFFFF”自定义格式说明符表示秒部分的前六个有效位；也就是说，它表示日期和时间值的百万分之几秒。但不显示尾随零或六个零位。

尽管可以显示时间值的秒部分的百万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

“FFFFFFF”自定义格式说明符

“FFFFFFF”自定义格式说明符表示秒部分的前七个有效位；也就是说，它表示日期和时间值的千万分之几秒。但不显示尾随零或七个零位。

尽管可以显示时间值的秒部分的千万分之几秒，但是该值可能并没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

返回表首

纪元“g”格式说明符

“g”或“gg”自定义格式说明符

“g”或“gg”自定义格式说明符（以及任意数量的附加“g”说明符）表示时期或纪元（例如 A.D）。如果要设置格式的日期没有关联的时期或纪元字符串，则格式设置操作将忽略此说明符。

如果使用“g”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“g”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“g”自定义格式说明符。

```
DateTime date1 = new DateTime(70, 08, 04);

Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.InvariantCulture));
// Displays 08/04/0070 A.D.
Console.WriteLine(date1.ToString("MM/dd/yyyy g",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays 08/04/0070 ap. J.-C.
```

```
Dim date1 As Date = #08/04/0070#

Console.WriteLine(date1.ToString("MM/dd/yyyy g", _
                  CultureInfo.InvariantCulture))
' Displays 08/04/0070 A.D.
Console.WriteLine(date1.ToString("MM/dd/yyyy g", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays 08/04/0070 ap. J.-C.
```

返回表首

小写小时“h”格式说明符

“h”自定义格式说明符

“h”自定义格式说明符将小时表示为从 1 至 12 的数字，即采用 12 小时制表示小时，自午夜或中午开始对整小时计数。午夜后经过的某特定小时数与中午过后的相同小时数无法加以区分。小时数不进行舍入，一位数字的小时数设置为不带前导零的格式。例如，给定上午或下午时间为 5:43，则此自定义格式说明符显示“5”。

如果使用“h”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“h”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 μ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 μ
```

返回表首

“hh”自定义格式说明符

“hh”自定义格式说明符（以及任意数量的附加“h”说明符）将小时表示为从 01 至 12 的数字，即采用 12 小时制表

示小时，自午夜或中午开始对整小时计数。午夜后经过的某特定小时数与中午过后的相同小时数无法加以区分。小时数不进行舍入，一位数字的小时数设置为带前导零的格式。例如，给定上午或下午时间为 5:43，则此格式说明符显示“05”。

下面的示例在一个自定义格式字符串中包含“hh”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

返回表首

大写小时“H”格式说明符

“H”自定义格式说明符

“H”自定义格式说明符将小时表示为从 0 至 23 的数字，即通过从零开始的 24 小时制表示小时，自午夜或中午开始对整小时计数。一位数字的小时数设置为不带前导零的格式。

如果使用“H”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“H”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("H:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 6:09:01
```

```
Dim date1 As Date = #6:09:01AM#
Console.WriteLine(date1.ToString("H:mm:ss", _
                  CultureInfo.InvariantCulture))
' Displays 6:09:01
```

返回表首

“HH”自定义格式说明符

“HH”自定义格式说明符（以及任意数量的附加“H”说明符）将小时表示为从 00 至 23 的数字，即通过从零开始的 24 小时制表示小时，自午夜或中午开始对整小时计数。一位数字的小时数设置为带前导零的格式。

下面的示例在一个自定义格式字符串中包含“HH”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 1, 1, 6, 9, 1);
Console.WriteLine(date1.ToString("HH:mm:ss",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01
```

```
Dim date1 As Date = #6:09:01AM#
Console.WriteLine(date1.ToString("HH:mm:ss", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01
```

返回表首

时区“K”格式说明符

“K”自定义格式说明符

“K”自定义格式说明符表示日期和时间值的时区信息。当此格式说明符与 DateTime 值一起使用时，结果字符串由 DateTime.Kind 属性的值进行定义：

- 对于本地时区（DateTime.Kind 的 DateTimeKind.Local 属性值），此说明符等效于“zzz”说明符，并产生一个包含相对于协调世界时 (UTC) 的本地偏移量的结果字符串，例如“-07:00”。
- 对于 UTC 时间（DateTime.Kind 的 DateTimeKind.Utc 属性值），结果字符串包含用于表示 UTC 日期的“Z”字符。
- 对于未指定时区的时间（其 DateTime.Kind 属性等于 DateTimeKind.Unspecified 的时间），结果等效于 String.Empty。

对于 DateTimeOffset 值，"K" 格式说明符相当于 "zzz" 格式说明符，生成的结果字符串包含 DateTimeOffset 值与 UTC 的偏移。

如果使用“K”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例显示对美国太平洋时区中的系统上的各种 DateTime 和 DateTimeOffset 值使用“K”自定义格式说明符。

```
Console.WriteLine(DateTime.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTime.UtcNow.ToString("%K"));
// Displays Z
Console.WriteLine("'{0}'",
                  DateTime.SpecifyKind(DateTime.Now,
                       DateTimeKind.Unspecified).ToString("%K"));
// Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"));
// Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"));
// Displays +00:00
Console.WriteLine(new DateTimeOffset(2008, 5, 1, 6, 30, 0,
                      new TimeSpan(5, 0, 0)).ToString("%K"));
// Displays +05:00
```

```
Console.WriteLine(Date.Now.ToString("%K"))
' Displays -07:00
Console.WriteLine(Date.UtcNow.ToString("%K"))
' Displays Z
Console.WriteLine("'{0}'", _
                  Date.SpecifyKind(Date.Now, _
                                   DateTimeKind.Unspecified). _
                  ToString("%K"))
' Displays ''
Console.WriteLine(DateTimeOffset.Now.ToString("%K"))
' Displays -07:00
Console.WriteLine(DateTimeOffset.UtcNow.ToString("%K"))
' Displays +00:00
Console.WriteLine(New DateTimeOffset(2008, 5, 1, 6, 30, 0, _
                                     New TimeSpan(5, 0, 0)). _
                  ToString("%K"))
' Displays +05:00
```

返回表首

分钟“m”格式说明符

“m”自定义格式说明符

“m”自定义格式说明符将分钟表示为从 0 到 59 的数字。分钟表示自上一小时以来经过的整分钟数。一位数字的分钟数设置为不带前导零的格式。

如果使用“m”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“m”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“m”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 μ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 μ
```

返回表首

“mm”自定义格式说明符

“mm”自定义格式说明符（以及任意数量的附加“m”说明符）将分钟表示为从 00 到 59 的数字。分钟表示自上一小时以来经过的整分钟数。一位数字的分钟数设置为带前导零的格式。

下面的示例在一个自定义格式字符串中包含“mm”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

返回表首

月份“M”格式说明符

“M”自定义格式说明符

“M”自定义格式说明符将月份表示为从 1 到 12 的数字(对于有 13 个月的日历，将月份表示为从 1 到 13 的数字)。一位数字的月份数设置为不带前导零的格式。

如果使用“M”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“M”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“M”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 18);
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("nl-NL")));
// Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMM",
                  CultureInfo.CreateSpecificCulture("lv-LV")));
// Displays (8) Aug, augusts
```

```
Dim date1 As Date = #8/18/2008#
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays (8) Aug, August
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("nl-NL")))
' Displays (8) aug, augustus
Console.WriteLine(date1.ToString("(M) MMM, MMMM", _
                  CultureInfo.CreateSpecificCulture("lv-LV")))
' Displays (8) Aug, augusts
```

返回表首

“MM”自定义格式说明符

“MM”自定义格式说明符将月份表示为从 01 到 12 的数字(对于有 13 个月的日历，将月份表示为从 1 到 13 的数字)。一位数字的月份数设置为带有前导零的格式。

下面的示例在一个自定义格式字符串中包含“MM”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 1, 2, 6, 30, 15);

Console.WriteLine(date1.ToString("dd, MM",
                  CultureInfo.InvariantCulture));
// 02, 01
```

```
Dim date1 As Date = #1/2/2008 6:30:15AM#

Console.WriteLine(date1.ToString("dd, MM", _
                  CultureInfo.InvariantCulture))
' 02, 01
```

返回表首

“MMM”自定义格式说明符

“MMM”自定义格式说明符表示月份的缩写名称。月份的本地化缩写名称通过当前或指定区域性的 DateTimeFormatInfo.AbbreviatedMonthNames 属性进行检索。

下面的示例在一个自定义格式字符串中包含“MMM”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM",
                  CultureInfo.CreateSpecificCulture("fr-FR")));
// Displays ven. 29 août
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Fri 29 Aug
Console.WriteLine(date1.ToString("ddd d MMM", _
                  CultureInfo.CreateSpecificCulture("fr-FR")))
' Displays ven. 29 août
```

返回表首

“MMMM”自定义格式说明符

“MMMM”自定义格式说明符表示月份的完整名称。月份的本地化名称通过当前或指定区域性的 DateTimeFormatInfo.MonthNames 属性进行检索。

下面的示例在一个自定义格式字符串中包含“MMMM”自定义格式说明符。

```
DateTime date1 = new DateTime(2008, 8, 29, 19, 27, 15);

Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("en-US")));
// Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM",
                  CultureInfo.CreateSpecificCulture("it-IT")));
// Displays venerdì 29 agosto
```

```
Dim date1 As Date = #08/29/2008 7:27:15PM#

Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("en-US")))
' Displays Friday 29 August
Console.WriteLine(date1.ToString("dddd dd MMMM", _
                  CultureInfo.CreateSpecificCulture("it-IT")))
' Displays venerdì 29 agosto
```

返回表首

秒“s”格式说明符

“s”自定义格式说明符

“s”自定义格式说明符将秒表示为从 0 到 59 的数字。结果表示自上一分钟以来经过的整秒数。一位数字的秒数设置为不带前导零的格式。

如果使用“s”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“s”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“s”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 μ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 μ
```

返回表首

“ss”自定义格式说明符

“ss”自定义格式说明符（以及任意数量的附加“s”说明符）将秒表示为从 00 到 59 的数字。结果表示自上一分钟以来经过的整秒数。一位数字的秒数设置为带前导零的格式。

下面的示例在一个自定义格式字符串中包含“ss”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

返回表首

正午“t”格式说明符

“t”自定义格式说明符

“t”自定义格式说明符表示 AM/PM 指示符的第一个字符。相应的本地化指示符通过当前或特定区域性的 DateTimeFormatInfo.AMDesignator 或 DateTimeFormatInfo.PMDesignator 属性进行检索。AM 指示符用于自 0:00:00（午夜）到 11:59:59.999 的所有时间。PM 指示符用于自 12:00:00（中午）到 23:59:59.999 的所有时间。

如果使用“t”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“t”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“t”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1 μ
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.InvariantCulture));
// Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t",
                  CultureInfo.CreateSpecificCulture("el-GR")));
// Displays 6:9:1.5 μ
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1 μ
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.InvariantCulture))
' Displays 6:9:1.5 P
Console.WriteLine(date1.ToString("h:m:s.F t", _
                  CultureInfo.CreateSpecificCulture("el-GR")))
' Displays 6:9:1.5 μ
```

返回表首

“tt”自定义格式说明符

“tt”自定义格式说明符（以及任意数量的附加“t”说明符）表示整个 AM/PM 指示符。相应的本地化指示符通过当前或特定区域性的 DateTimeFormatInfo.AMDesignator 或 DateTimeFormatInfo.PMDesignator 属性进行检索。AM 指示符用于自 0:00:00（午夜）到 11:59:59.999 的所有时间。PM 指示符用于自 12:00:00（中午）到 23:59:59.999 的所有时间。

对于需要维护 AM 与 PM 之间的差异的语言，应确保使用“tt”说明符。以日语为例，其 AM 和 PM 指示符的差异点为第二个字符，而非第一个字符。

下面的示例在一个自定义格式字符串中包含“tt”自定义格式说明符。

```
DateTime date1;
date1 = new DateTime(2008, 1, 1, 18, 9, 1);
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01 du.
date1 = new DateTime(2008, 1, 1, 18, 9, 1, 500);
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.InvariantCulture));
// Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt",
                  CultureInfo.CreateSpecificCulture("hu-HU")));
// Displays 06:09:01.50 du.
```

```
Dim date1 As Date
date1 = #6:09:01PM#
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01 PM
Console.WriteLine(date1.ToString("hh:mm:ss tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01 du.
date1 = New Date(2008, 1, 1, 18, 9, 1, 500)
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.InvariantCulture))
' Displays 06:09:01.50 PM
Console.WriteLine(date1.ToString("hh:mm:ss.ff tt", _
                  CultureInfo.CreateSpecificCulture("hu-HU")))
' Displays 06:09:01.50 du.
```

返回表首

年份“y”格式说明符

“y”自定义格式说明符

“y”自定义格式说明符将年份表示为一位或两位数字。如果年份多于两位数，则结果中仅显示两位低位数。如果两位数字的年份的第一个数字以零开始（例如，2008），则该数字设置为不带前导零的格式。

如果使用“y”格式说明符而没有其他自定义格式说明符，则将该说明符解释为“y”标准日期和时间格式说明符。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“y”自定义格式说明符。

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

返回表首

“yy”自定义格式说明符

“yy”自定义格式说明符将年份表示为一位或两位数字。如果年份多于两位数，则结果中仅显示两位低位数。如果两位数年份的有效数字少于两个，则用前导零填充该数字以产生两位数。

在分析操作中，使用“yy”自定义格式说明符分析的两位数年份是基于格式提供程序的当前日历的 Calendar.TwoDigitYearMax 属性被释义的。下面的示例通过使用默认 zh-cn 区域性（在这种情况下为当前区域性）的公历来分析具有二位数年份的日期的字符串表示形式。然后，它将更改当前区域的 CultureInfo 对象以便使用其 GregorianCalendar 属性已更改的 TwoDigitYearMax 对象。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string fmt = "dd-MMM-yy";
      string value = "24-Jan-49";

      Calendar cal = (Calendar) CultureInfo.CurrentCulture.Calendar.Clone();
      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);

      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
      Console.WriteLine();

      cal.TwoDigitYearMax = 2099;
      CultureInfo culture = (CultureInfo) CultureInfo.CurrentCulture.Clone();
      culture.DateTimeFormat.Calendar = cal;
      Thread.CurrentThread.CurrentCulture = culture;

      Console.WriteLine("Two Digit Year Range: {0} - {1}",
                        cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax);
      Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, null));
   }
}
// The example displays the following output:
//       Two Digit Year Range: 1930 - 2029
//       1/24/1949
//
//       Two Digit Year Range: 2000 - 2099
//       1/24/2049
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim fmt As String = "dd-MMM-yy"
        Dim value As String = "24-Jan-49"

        Dim cal As Calendar = CType(CultureInfo.CurrentCulture.Calendar.Clone(), Calendar)
        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                          cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax)

        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, Nothing))
        Console.WriteLine()

        cal.TwoDigitYearMax = 2099
        Dim culture As CultureInfo = CType(CultureInfo.CurrentCulture.Clone(), CultureInfo)
        culture.DateTimeFormat.Calendar = cal
        Thread.CurrentThread.CurrentCulture = culture

        Console.WriteLine("Two Digit Year Range: {0} - {1}",
                          cal.TwoDigitYearMax - 99, cal.TwoDigitYearMax)
        Console.WriteLine("{0:d}", DateTime.ParseExact(value, fmt, Nothing))
    End Sub
End Module
' The example displays the following output:
'       Two Digit Year Range: 1930 - 2029
'       1/24/1949
'
'       Two Digit Year Range: 2000 - 2099
'       1/24/2049
```

下面的示例在一个自定义格式字符串中包含“yy”自定义格式说明符。

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

返回表首

“yyy”自定义格式说明符

“yyy”自定义格式说明符至少使用三位数字表示年份。如果年份的有效数字多于三个，则将它们包括在结果字符串中。如果年份少于三位数，则用前导零填充该数字以产生三位数。

> **NOTE**
>
> 对于年份可以为五位数的泰国佛历，此格式说明符将显示全部有效数字。

下面的示例在一个自定义格式字符串中包含“yyy”自定义格式说明符。

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

返回表首

“yyyy”自定义格式说明符

“yyyy”自定义格式说明符至少使用四位数字表示年份。如果年份的有效数字多于四个，则将它们包括在结果字符串中。如果年份少于四位数，则用前导零填充该数字使其达到四位数。

> **NOTE**
>
> 对于年份可以为五位数的泰国佛历，此格式说明符将最少显示四位数字。

下面的示例在一个自定义格式字符串中包含“yyyy”自定义格式说明符。

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

返回表首

“yyyyy”自定义格式说明符

“yyyyy”自定义格式说明符（以及任意数量的附加“y”说明符）最少将年份表示为五位数字。如果年份的有效数字多于五个，则将它们包括在结果字符串中。如果年份少于五位数，则用前导零填充该数字以产生五位数。

如果存在额外的“y”说明符，则用所需个数的前导零填充该数字以产生“y”说明符的数目。

下面的示例在一个自定义格式字符串中包含“yyyyy”自定义格式说明符。

```
DateTime date1 = new DateTime(1, 12, 1);
DateTime date2 = new DateTime(2010, 1, 1);
Console.WriteLine(date1.ToString("%y"));
// Displays 1
Console.WriteLine(date1.ToString("yy"));
// Displays 01
Console.WriteLine(date1.ToString("yyy"));
// Displays 001
Console.WriteLine(date1.ToString("yyyy"));
// Displays 0001
Console.WriteLine(date1.ToString("yyyyy"));
// Displays 00001
Console.WriteLine(date2.ToString("%y"));
// Displays 10
Console.WriteLine(date2.ToString("yy"));
// Displays 10
Console.WriteLine(date2.ToString("yyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyy"));
// Displays 2010
Console.WriteLine(date2.ToString("yyyyy"));
// Displays 02010
```

```
Dim date1 As Date = #12/1/0001#
Dim date2 As Date = #1/1/2010#
Console.WriteLine(date1.ToString("%y"))
' Displays 1
Console.WriteLine(date1.ToString("yy"))
' Displays 01
Console.WriteLine(date1.ToString("yyy"))
' Displays 001
Console.WriteLine(date1.ToString("yyyy"))
' Displays 0001
Console.WriteLine(date1.ToString("yyyyy"))
' Displays 00001
Console.WriteLine(date2.ToString("%y"))
' Displays 10
Console.WriteLine(date2.ToString("yy"))
' Displays 10
Console.WriteLine(date2.ToString("yyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyy"))
' Displays 2010
Console.WriteLine(date2.ToString("yyyyy"))
' Displays 02010
```

返回表首

偏移量“z”格式说明符

“z”自定义格式说明符

与 DateTime 值一起使用时，“z”自定义格式说明符表示本地操作系统的时区相对于协调世界时 (UTC) 的有符号偏移量（以小时为单位）。它不反映一个实例的 DateTime.Kind 属性的值。出于此原因，不建议将“z”格式说明符与 DateTime 值一起使用。

与 DateTimeOffset 值一起使用时，此格式说明符表示 DateTimeOffset 值相对于 UTC 的偏移量（以小时为单位）。

偏移量始终显示为带有前导符号。加号 (+) 指示早于 UTC 的小时数，减号 (-) 指示晚于 UTC 的小时数。一位数字的偏移量设置为不带前导零的格式。

如果使用“z”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

下面的示例在一个自定义格式字符串中包含“z”自定义格式说明符。

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

返回表首

"zz"自定义格式说明符

与 DateTime 值一起使用时，"zz"自定义格式说明符表示本地操作系统的时区相对于 UTC 的有符号偏移量（以小时为单位）。它不反映一个实例的 DateTime.Kind 属性的值。出于此原因，不建议将"zz"格式说明符与 DateTime 值一起使用。

与 DateTimeOffset 值一起使用时，此格式说明符表示 DateTimeOffset 值相对于 UTC 的偏移量（以小时为单位）。

偏移量始终显示为带有前导符号。加号 (+) 指示早于 UTC 的小时数，减号 (-) 指示晚于 UTC 的小时数。一位数字的偏移量设置为带前导零的格式。

下面的示例在一个自定义格式字符串中包含"zz"自定义格式说明符。

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

返回表首

"zzz"自定义格式说明符

与 DateTime 值一起使用时，"zzz"自定义格式说明符表示本地操作系统的时区相对于 UTC 的有符号偏移量（以小时和分钟为单位）。它不反映一个实例的 DateTime.Kind 属性的值。出于此原因，不建议将"zzz"格式说明符与 DateTime 值一起使用。

与 DateTimeOffset 值一起使用时，此格式说明符表示 DateTimeOffset 值相对于 UTC 的偏移量（以小时和分钟为单位）。

偏移量始终显示为带有前导符号。加号 (+) 指示早于 UTC 的小时数，减号 (-) 指示晚于 UTC 的小时数。一位数字的偏移量设置为带前导零的格式。

下面的示例在一个自定义格式字符串中包含“zzz”自定义格式说明符。

```
DateTime date1 = DateTime.UtcNow;
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date1));
// Displays -7, -07, -07:00

DateTimeOffset date2 = new DateTimeOffset(2008, 8, 1, 0, 0, 0,
                                          new TimeSpan(6, 0, 0));
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}",
                  date2));
// Displays +6, +06, +06:00
```

```
Dim date1 As Date = Date.UtcNow
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date1))
' Displays -7, -07, -07:00

Dim date2 As New DateTimeOffset(2008, 8, 1, 0, 0, 0, _
                                New Timespan(6, 0, 0))
Console.WriteLine(String.Format("{0:%z}, {0:zz}, {0:zzz}", _
                  date2))
' Displays +6, +06, +06:00
```

返回表首

日期和时间分隔符说明符

“:”自定义格式说明符

“:”自定义格式说明符表示时间分隔符，它用于区分小时、分钟和秒。相应的本地化时间分隔符通过当前或指定区域性的 DateTimeFormatInfo.TimeSeparator 属性进行检索。

> **NOTE**
> 若要更改特定日期和时间字符串的时间分隔符，请指定文本字符串分隔符内的分隔字符。例如，在自定义格式字符串 `hh'_'dd'_'ss` 产生的结果字符串中，始终将“_”(下划线)用作时间分隔符。若要更改区域所有日期的时间分隔符，可更改当前区域的 DateTimeFormatInfo.TimeSeparator 属性，或者实例化 DateTimeFormatInfo 对象，将字符分配到其 TimeSeparator 属性并调用包含 IFormatProvider 形参的格式设置方法的重载。

如果使用“:”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

返回表首

“/”自定义格式说明符

“/”自定义格式说明符表示日期分隔符，它用于区分年、月和日。相应的本地化日期分隔符检索自当前或指定区域性的 DateTimeFormatInfo.DateSeparator 属性。

> **NOTE**
> 若要更改特定日期和时间字符串的日期分隔符，请指定文本字符串分隔符内的分隔字符。例如，在自定义格式字符串 `mm'/'dd'/'yyyy` 产生的结果字符串中，始终将“/”用作日期分隔符。若要更改区域所有日期的日期分隔符，可更改当前区域的 DateTimeFormatInfo.DateSeparator 属性，或者实例化 DateTimeFormatInfo 对象，将字符分配到其 DateSeparator 属性并调用包含 IFormatProvider 形参的格式设置方法的重载。

如果使用“/”格式说明符而没有其他自定义格式说明符，则将该说明符解释为标准日期和时间格式说明符，并引

发 FormatException。有关使用单个格式说明符的更多信息，请参阅本文后面的使用单个自定义格式说明符。

返回表首

字符文本

自定义日期和时间格式字符串中的以下字符是保留的字符，始终解释为格式字符，但 `"`、`'`、`/` 和 `\` 解释为特殊字符。

`F`	`H`	`K`	`M`	`d`
`f`	`g`	`h`	`m`	`s`
`t`	`y`	`z`	`%`	`:`
`/`	`"`	`'`	`\`	

所有其他字符始终解释为字符文本，在格式设置操作中，将按原样包含在结果字符串中。在分析操作中，这些字符必须与输入字符串中的字符完全匹配；比较时区分大小写。

下面的示例在格式字符串中包含用于表示时区的文本字符“PST”（太平洋标准时间）和“PDT”（太平洋夏令时）。请注意，该字符串将包含在结果字符串中，并且包含本地时区字符串的字符串也可成功完成分析。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String[] formats = { "dd MMM yyyy hh:mm tt PST",
                           "dd MMM yyyy hh:mm tt PDT" };
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(formats[1]));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm PST";
      DateTime newDate;
      if (DateTime.TryParseExact(value, formats, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim formats() As String = {"dd MMM yyyy hh:mm tt PST",
                                   "dd MMM yyyy hh:mm tt PDT"}
        Dim dat As New Date(2016, 8, 18, 16, 50, 0)
        ' Display the result string.
        Console.WriteLine(dat.ToString(formats(1)))

        ' Parse a string.
        Dim value As String = "25 Dec 2016 12:00 pm PST"
        Dim newDate As Date
        If Date.TryParseExact(value, formats, Nothing,
                              DateTimeStyles.None, newDate) Then
            Console.WriteLine(newDate)
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If
    End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM PDT
'       12/25/2016 12:00:00 PM
```

可通过两种方法来指示要将字符解释为文本字符而不是保留字符，以便这些字符可以包含在结果字符串中，或者在输入字符串中成功完成分析：

- 通过转义每个保留字符。有关详细信息，请参阅使用转义字符。

下面的示例在格式字符串中包含用于表示时区的文本字符“pst”（太平洋标准时间）。由于“s”和“t”都是自定义格式字符串，因此这两个字符必须经过转义才能解释为字符文本。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String format = "dd MMM yyyy hh:mm tt p\\s\\t";
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(format));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm pst";
      DateTime newDate;
      if (DateTime.TryParseExact(value, format, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim fmt As String = "dd MMM yyyy hh:mm tt p\s\t"
        Dim dat As New Date(2016, 8, 18, 16, 50, 0)
        ' Display the result string.
        Console.WriteLine(dat.ToString(fmt))

        ' Parse a string.
        Dim value As String = "25 Dec 2016 12:00 pm pst"
        Dim newDate As Date
        If Date.TryParseExact(value, fmt, Nothing,
                              DateTimeStyles.None, newDate) Then
            Console.WriteLine(newDate)
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If
    End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM pst
'       12/25/2016 12:00:00 PM
```

- 通过将整个文本字符串括在双引号或单引号中。下面的示例与前一个示例类似，不过，pst 括在引号中，指示应将整个分隔字符串解释为字符文本。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      String format = "dd MMM yyyy hh:mm tt \"pst\"";
      var dat = new DateTime(2016, 8, 18, 16, 50, 0);
      // Display the result string.
      Console.WriteLine(dat.ToString(format));

      // Parse a string.
      String value = "25 Dec 2016 12:00 pm pst";
      DateTime newDate;
      if (DateTime.TryParseExact(value, format, null,
                                 DateTimeStyles.None, out newDate))
         Console.WriteLine(newDate);
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       18 Aug 2016 04:50 PM PDT
//       12/25/2016 12:00:00 PM
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim fmt As String = "dd MMM yyyy hh:mm tt ""pst"""
        Dim dat As New Date(2016, 8, 18, 16, 50, 0)
        ' Display the result string.
        Console.WriteLine(dat.ToString(fmt))

        ' Parse a string.
        Dim value As String = "25 Dec 2016 12:00 pm pst"
        Dim newDate As Date
        If Date.TryParseExact(value, fmt, Nothing,
                              DateTimeStyles.None, newDate) Then
            Console.WriteLine(newDate)
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If
    End Sub
End Module
' The example displays the following output:
'       18 Aug 2016 04:50 PM pst
'       12/25/2016 12:00:00 PM
```

说明

使用单个自定义格式说明符

自定义日期和时间格式字符串由两个或更多字符组成。日期和时间格式设置方法将任何单字符字符串解释标准日期和时间格式字符串。如果它们无法将该字符识别为有效格式说明符，则会引发 FormatException。例如，仅包含说明符“h”的格式字符串解释为标准日期和时间格式字符串。但是，在此特定情况下将引发异常，原因是不存在“h”标准日期和时间格式说明符。

若要将任何自定义日期和时间格式说明符作为格式字符串中的唯一说明符使用（即若要使用“d”、“f”、“F”、“g”、“h”、“H”、“K”、“m”、“M”、“s”、“t”、“y”、“z”、“:”或“/”自定义格式说明符自身），请在该说明符之前或之后添加一个空格，或在单个自定义日期和时间说明符之前包括一个百分号（“%”）格式说明符。

例如，“`%h`”将被解释为一个自定义日期和时间格式字符串，用于显示当前日期和时间值表示的小时。也可以使用“ h”或“h ”格式字符串，尽管它在结果字符串及小时中包含一个空格。下面的示例阐释了这三个格式字符串。

```
DateTime dat1 = new DateTime(2009, 6, 15, 13, 45, 0);

Console.WriteLine("'{0:%h}'", dat1);
Console.WriteLine("'{0: h}'", dat1);
Console.WriteLine("'{0:h }'", dat1);
// The example displays the following output:
//       '1'
//       ' 1'
//       '1 '
```

```
Dim dat1 As Date = #6/15/2009 1:45PM#

Console.WriteLine("'{0:%h}'", dat1)
Console.WriteLine("'{0: h}'", dat1)
Console.WriteLine("'{0:h }'", dat1)
' The example displays the following output:
'       '1'
'       ' 1'
'       '1 '
```

使用转义字符

格式字符串中的“d”、“f”、“F”、“g”、“h”、“H”、“K”、“m”、“M”、“s”、“t”、“y”、“z”、“:”或“/”字符被解释为自定义格式说明符而不是文本字符。若要防止某个字符被解释为格式说明符，可以在该字符前面加上反斜杠 (\)，即转义字符。转义字符表示以下字符为应包含在未更改的结果字符串中的字符文本。

若在要结果字符串中包括反斜杠，必须使用另一个反斜杠 (`\\`) 对其转义。

> **NOTE**
> 一些编译器(如 C++ 和 C# 编译器)也可能会将单个反斜杠字符解释为转义字符。若要确保在设置格式时正确解释字符串，在 C# 中，可以在字符串之前使用原义字符串文本字符(@ 字符)，或者在 C# 和 C++ 中，在每个反斜杠之前另外添加一个反斜杠字符。下面的 C# 示例阐释了这两种方法。

下面的示例使用转义字符，以防止格式设置操作将“h”和“m”字符解释为格式说明符。

```
DateTime date = new DateTime(2009, 06, 15, 13, 45, 30, 90);
string fmt1 = "h \\h m \\m";
string fmt2 = @"h \h m \m";

Console.WriteLine("{0} ({1}) -> {2}", date, fmt1, date.ToString(fmt1));
Console.WriteLine("{0} ({1}) -> {2}", date, fmt2, date.ToString(fmt2));
// The example displays the following output:
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
//       6/15/2009 1:45:30 PM (h \h m \m) -> 1 h 45 m
```

```
Dim date1 As Date = #6/15/2009 13:45#
Dim fmt As String = "h \h m \m"

Console.WriteLine("{0} ({1}) -> {2}", date1, fmt, date1.ToString(fmt))
' The example displays the following output:
'       6/15/2009 1:45:00 PM (h \h m \m) -> 1 h 45 m
```

控制面板设置

对于包含许多自定义日期和时间格式说明符的格式设置操作，控制面板中的“区域和语言选项”设置会影响其产生的结果字符串。这些设置用于初始化与当前线程区域性关联的 DateTimeFormatInfo 对象，当前线程区域性提供用于控制格式设置的值。使用不同设置的计算机将生成不同的结果字符串。

此外，如果使用 CultureInfo(String) 构造函数实例化一个新的 CultureInfo 对象以表示与当前的系统区域性相同的区域性，则通过控制面板中的 **“区域和语言选项”** 建立的任何自定义都将应用到新的 CultureInfo 对象。可以使用 CultureInfo(String, Boolean) 构造函数来创建不会反映系统的自定义项的 CultureInfo 对象。

DateTimeFormatInfo 属性

格式化受当前的 DateTimeFormatInfo 对象的属性影响，其由当前线程区域性隐式提供或由调用格式化的方法的 IFormatProvider 参数显式提供。对于 IFormatProvider 参数，应当指定一个表示区域性的 CultureInfo 对象或指定一个 DateTimeFormatInfo 对象。

由许多自定义日期和时间格式说明符产生的结果字符串还取决于当前的 DateTimeFormatInfo 对象的属性。应用程序通过更改相应的 DateTimeFormatInfo 属性，可以改变由某些自定义日期和时间格式说明符产生的结果。例如，“ddd”格式说明符将在 AbbreviatedDayNames 字符串数组中找到的缩写的星期名称添加到结果字符串。类似地，"MMMM"格式说明符将在 MonthNames 字符串数组中找到的月的完整名称添加到结果字符串。

请参阅

- System.DateTime
- System.IFormatProvider

- 格式设置类型
- 标准日期和时间格式字符串
- 示例：.NET Core WinForms 格式设置实用工具 (C#)
- 示例：.NET Core WinForms 格式设置实用工具 (Visual Basic)

标准 TimeSpan 格式字符串

2021/11/16 •

标准 TimeSpan 格式字符串使用一个格式说明符，定义格式设置操作生成的 TimeSpan 值的文本表示形式。任何包含一个以上字符（包括空格）的格式字符串都被解释为自定义 TimeSpan 格式字符串。有关详细信息，请参阅自定义 TimeSpan 格式字符串。

通过调用 TimeSpan 方法的重载以及通过支持复合格式设置的方法（如 TimeSpan.ToString）产生 String.Format 值的字符串表示形式。有关更多信息，请参见格式设置类型和复合格式设置。以下示例演示了标准格式字符串在格式设置操作中的用法。

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);
      string output = "Time of Travel: " + duration.ToString("c");
      Console.WriteLine(output);

      Console.WriteLine("Time of Travel: {0:c}", duration);
   }
}
// The example displays the following output:
//       Time of Travel: 1.12:24:02
//       Time of Travel: 1.12:24:02
```

```
Module Example
    Public Sub Main()
        Dim duration As New TimeSpan(1, 12, 23, 62)
        Dim output As String = "Time of Travel: " + duration.ToString("c")
        Console.WriteLine(output)

        Console.WriteLine("Time of Travel: {0:c}", duration)
    End Sub
End Module
' The example displays the following output:
'       Time of Travel: 1.12:24:02
'       Time of Travel: 1.12:24:02
```

标准 TimeSpan 格式字符串也被 TimeSpan.ParseExact 和 TimeSpan.TryParseExact 方法用于定义分析操作所需的输入字符串的格式。（分析将值的字符串表示形式转换成该值。）以下示例演示了标准格式字符串在分析操作中的用法。

```
using System;

public class Example
{
   public static void Main()
   {
      string value = "1.03:14:56.1667";
      TimeSpan interval;
      try {
         interval = TimeSpan.ParseExact(value, "c", null);
         Console.WriteLine("Converted '{0}' to {1}", value, interval);
      }
      catch (FormatException) {
         Console.WriteLine("{0}: Bad Format", value);
      }
      catch (OverflowException) {
         Console.WriteLine("{0}: Out of Range", value);
      }

      if (TimeSpan.TryParseExact(value, "c", null, out interval))
         Console.WriteLine("Converted '{0}' to {1}", value, interval);
      else
         Console.WriteLine("Unable to convert {0} to a time interval.",
                           value);
   }
}
// The example displays the following output:
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
//       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
```

```
Module Example
    Public Sub Main()
        Dim value As String = "1.03:14:56.1667"
        Dim interval As TimeSpan
        Try
            interval = TimeSpan.ParseExact(value, "c", Nothing)
            Console.WriteLine("Converted '{0}' to {1}", value, interval)
        Catch e As FormatException
            Console.WriteLine("{0}: Bad Format", value)
        Catch e As OverflowException
            Console.WriteLine("{0}: Out of Range", value)
        End Try

        If TimeSpan.TryParseExact(value, "c", Nothing, interval) Then
            Console.WriteLine("Converted '{0}' to {1}", value, interval)
        Else
            Console.WriteLine("Unable to convert {0} to a time interval.",
                              value)
        End If
    End Sub
End Module
' The example displays the following output:
'       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
'       Converted '1.03:14:56.1667' to 1.03:14:56.1667000
```

下表列出了标准时间间隔格式说明符。

[illegible]	"[illegible]"	[illegible]	[illegible]
"c"	常量(固定)格式	此说明符不区分区域性。它的形式是 `[-][d'.']hh':'mm':'ss['.'fffffff]`。 ("t"格式与"T"格式字符串产生的结果相同。) 更多信息:常量("c")格式说明符。	`TimeSpan.Zero` -> 00:00:00 `New TimeSpan(0, 0, 30, 0)` -> 00:30:00 `New TimeSpan(3, 17, 25, 30, 500)` -> 3.17:25:30.5000000
"g"	常规短格式	该说明符仅输出需要的内容。它区分区域性并采用 `[-][d':']h':'mm':'ss[.FFFFFFF]` 形式。 更多信息:常规短("g")格式说明符。	`New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50.5 (en-US) `New TimeSpan(1, 3, 16, 50, 500)` -> 1:3:16:50,5 (fr-FR) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50.599 (en-US) `New TimeSpan(1, 3, 16, 50, 599)` -> 1:3:16:50,599 (fr-FR)
"G"	常规长格式	此说明符始终输出天数和七个小数位。它区分区域性并采用 `[-]d':'hh':'mm':'ss.fffffff` 形式。 更多信息:常规长("G")格式说明符。	`New TimeSpan(18, 30, 0)` -> 0:18:30:00.0000000 (en-US) `New TimeSpan(18, 30, 0)` -> 0:18:30:00,0000000 (fr-FR)

常量("c")格式说明符。

"c"格式说明符返回的 TimeSpan 值的字符串表示形式具有以下形式:

[-][*d*.]*hh*:*mm*:*ss*[.*fffffff*]

方括号 ([and]) 中的元素是可选的。句点 (.) 和冒号 (:) 是文字符号。下表介绍了剩余的元素。

[illegible]	[illegible]
-	可选负号,指示负时间间隔。
d	不带前导零的可选天数。
hh	小时数,范围为"00"到"23"。
mm	分钟数,范围为"00"到"59"。
ss	秒数,范围为"0"到"59"。

“	“
fffffff	秒的可选小数部分。其值的范围为“0000001”(一刻度或一秒的一千万分之一)到“9999999”(一秒的一千万分之九百九十九万九千九百九或一秒少一刻度)。

与“g”和“G”格式说明符不同，“c”格式说明符不区分区域性。它产生了 TimeSpan 值的字符串表示形式，该值不变且在 .NET Framework 4 之前的版本中通用。“c”是默认的 TimeSpan 格式字符串；TimeSpan.ToString() 方法使用“c”格式字符串设置时间间隔值的格式。

> **NOTE**
>
> TimeSpan 也支持“t”和“T”标准格式字符串，其行为与“c”标准格式字符串相同。

以下示例对两个 TimeSpan 对象进行了实例化，使用它们来执行算术运算并显示结果。在每种情况下，它都通过“c”格式说明符使用复合格式设置来显示 TimeSpan 值。

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:c} - {1:c} = {2:c}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2);

      interval1 = new TimeSpan(0, 0, 1, 14, 365);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       07:45:16 - 18:12:38 = -10:27:22
//       07:45:16 + 18:12:38 = 1.01:57:54
//       00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

```
Module Example
    Public Sub Main()
        Dim interval1, interval2 As TimeSpan
        interval1 = New TimeSpan(7, 45, 16)
        interval2 = New TimeSpan(18, 12, 38)

        Console.WriteLine("{0:c} - {1:c} = {2:c}", interval1,
                          interval2, interval1 - interval2)
        Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                          interval2, interval1 + interval2)

        interval1 = New TimeSpan(0, 0, 1, 14, 365)
        interval2 = TimeSpan.FromTicks(2143756)
        Console.WriteLine("{0:c} + {1:c} = {2:c}", interval1,
                          interval2, interval1 + interval2)
    End Sub
End Module
' The example displays the following output:
'       07:45:16 - 18:12:38 = -10:27:22
'       07:45:16 + 18:12:38 = 1.01:57:54
'       00:01:14.3650000 + 00:00:00.2143756 = 00:01:14.5793756
```

常规短("g")格式说明符

"g"TimeSpan 格式说明符通过只包含所需元素来返回简洁形式的 TimeSpan 值的字符串表示形式。它具有以下形式：

[-][*d*.]*h*:*mm*:*ss*[.*FFFFFFF*]

方括号 ([and]) 中的元素是可选的。冒号 (:) 是一种文字符号。下表介绍了剩余的元素。

"	"
-	可选负号，指示负时间间隔。
d	不带前导零的可选天数。
h	范围为"0"到"23"的小时数，无前导零。
mm	分钟数，范围为"00"到"59"。
ss	秒数，范围为"00"到"59"。
.	秒的小数部分的分隔符。它相当于指定区域中无需用户重写的 NumberDecimalSeparator 属性。
FFFFFFF	秒的小数部分。显示尽可能少的数位。

和"G"格式一样，对"g"格式说明符进行了本地化。其秒的小数部分的分隔符基于当前区域或指定区域的 NumberDecimalSeparator 属性。

以下示例对两个 TimeSpan 对象进行了实例化，使用它们来执行算术运算并显示结果。在每种情况下，它都通过"g"格式说明符使用复合格式设置来显示 TimeSpan 值。此外，它通过使用当前系统区域(在此情况下为"英语 - 美国"或"en-US")和"法语 - 法国 (fr-FR)"区域的格式设置约定来设置 TimeSpan 值的格式。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:g} - {1:g} = {2:g}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:g} + {1:g} = {2:g}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       7:45:16 - 18:12:38 = -10:27:22
//       7:45:16 + 18:12:38 = 1:1:57:54
//       0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim interval1, interval2 As TimeSpan
        interval1 = New TimeSpan(7, 45, 16)
        interval2 = New TimeSpan(18, 12, 38)

        Console.WriteLine("{0:g} - {1:g} = {2:g}", interval1,
                          interval2, interval1 - interval2)
        Console.WriteLine(String.Format(New CultureInfo("fr-FR"),
                          "{0:g} + {1:g} = {2:g}", interval1,
                          interval2, interval1 + interval2))

        interval1 = New TimeSpan(0, 0, 1, 14, 36)
        interval2 = TimeSpan.FromTicks(2143756)
        Console.WriteLine("{0:g} + {1:g} = {2:g}", interval1,
                          interval2, interval1 + interval2)
    End Sub
End Module
' The example displays the following output:
'       7:45:16 - 18:12:38 = -10:27:22
'       7:45:16 + 18:12:38 = 1:1:57:54
'       0:01:14.036 + 0:00:00.2143756 = 0:01:14.2503756
```

常规长("G")格式说明符

("G")TimeSpan 格式说明符用始终包含日期和秒的小数部分的长格式返回 TimeSpan 值的字符串表示形式。
"G"标准格式说明符生成的字符串具有以下形式:

[-]*d.hh:mm:ss.fffffff*

方括号 ([and]) 中的元素是可选的。冒号 (:) 是一种文字符号。下表介绍了剩余的元素。

“	“
-	可选负号，指示负时间间隔。
d	不带前导零的天数。
hh	小时数，范围为“00”到“23”。
mm	分钟数，范围为“00”到“59”。
ss	秒数，范围为“00”到“59”。
.	秒的小数部分的分隔符。它相当于指定区域中无需用户重写的 NumberDecimalSeparator 属性。
fffffff	秒的小数部分。

和“G”格式一样，对“g”格式说明符进行了本地化。其秒的小数部分的分隔符基于当前区域或指定区域的 NumberDecimalSeparator 属性。

以下示例对两个 TimeSpan 对象进行了实例化，使用它们来执行算术运算并显示结果。在每种情况下，它都通过“G”格式说明符使用复合格式设置来显示 TimeSpan 值。此外，它通过使用当前系统区域（在此情况下为“英语 - 美国”或“en-US”）和“法语 - 法国 (fr-FR)”区域的格式设置约定来设置 TimeSpan 值的格式。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      TimeSpan interval1, interval2;
      interval1 = new TimeSpan(7, 45, 16);
      interval2 = new TimeSpan(18, 12, 38);

      Console.WriteLine("{0:G} - {1:G} = {2:G}", interval1,
                        interval2, interval1 - interval2);
      Console.WriteLine(String.Format(new CultureInfo("fr-FR"),
                        "{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2));

      interval1 = new TimeSpan(0, 0, 1, 14, 36);
      interval2 = TimeSpan.FromTicks(2143756);
      Console.WriteLine("{0:G} + {1:G} = {2:G}", interval1,
                        interval2, interval1 + interval2);
   }
}
// The example displays the following output:
//       0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
//       0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
//       0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim interval1, interval2 As TimeSpan
        interval1 = New TimeSpan(7, 45, 16)
        interval2 = New TimeSpan(18, 12, 38)

        Console.WriteLine("{0:G} - {1:G} = {2:G}", interval1,
                          interval2, interval1 - interval2)
        Console.WriteLine(String.Format(New CultureInfo("fr-FR"),
                          "{0:G} + {1:G} = {2:G}", interval1,
                          interval2, interval1 + interval2))

        interval1 = New TimeSpan(0, 0, 1, 14, 36)
        interval2 = TimeSpan.FromTicks(2143756)
        Console.WriteLine("{0:G} + {1:G} = {2:G}", interval1,
                          interval2, interval1 + interval2)
    End Sub
End Module
' The example displays the following output:
'       0:07:45:16.0000000 - 0:18:12:38.0000000 = -0:10:27:22.0000000
'       0:07:45:16,0000000 + 0:18:12:38,0000000 = 1:01:57:54,0000000
'       0:00:01:14.0360000 + 0:00:00:00.2143756 = 0:00:01:14.2503756
```

请参阅

- 格式设置类型
- 自定义 TimeSpan 格式字符串
- 分析字符串

自定义 TimeSpan 格式字符串

2021/11/16 •

TimeSpan 格式字符串定义格式设置操作生成的 TimeSpan 值的字符串表示形式。自定义格式字符串包含一个或多个自定义 TimeSpan 格式说明符，以及任意数量的文本字符。任何不是标准 TimeSpan 格式字符串的字符串都会解释为自定义 TimeSpan 格式字符串。

> **IMPORTANT**
>
> 自定义 TimeSpan 格式说明符不包含占位符分隔符符号，如分隔天与小时、小时与分钟或秒与秒若干分之一的符号。相反，这些符号必须以字符串形式包含在自定义格式字符串中。例如，`"dd\.hh\:mm"` 将句点 (.) 定义为天与小时之间的分隔符，将冒号 (:) 定义为小时与分钟之间的分隔符。
>
> 自定义 TimeSpan 格式说明符还不包括正负符号，无法区分正负时间间隔。若要包含正负符号，必须使用条件逻辑构造格式字符串。其他字符部分包括一个示例。

TimeSpan 值的字符串表示形式通过调用 TimeSpan.ToString 方法重载由支持复合格式的方法（如 String.Format）生成。有关更多信息，请参见格式设置类型和复合格式设置。下面的示例展示了自定义格式字符串在格式设置操作中的用法。

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan duration = new TimeSpan(1, 12, 23, 62);

      string output = null;
      output = "Time of Travel: " + duration.ToString("%d") + " days";
      Console.WriteLine(output);
      output = "Time of Travel: " + duration.ToString(@"dd\.hh\:mm\:ss");
      Console.WriteLine(output);

      Console.WriteLine("Time of Travel: {0:%d} day(s)", duration);
      Console.WriteLine("Time of Travel: {0:dd\\.hh\\:mm\\:ss} days", duration);
   }
}
// The example displays the following output:
//       Time of Travel: 1 days
//       Time of Travel: 01.12:24:02
//       Time of Travel: 1 day(s)
//       Time of Travel: 01.12:24:02 days
```

```
Module Example
    Public Sub Main()
        Dim duration As New TimeSpan(1, 12, 23, 62)

        Dim output As String = Nothing
        output = "Time of Travel: " + duration.ToString("%d") + " days"
        Console.WriteLine(output)
        output = "Time of Travel: " + duration.ToString("dd\.hh\:mm\:ss")
        Console.WriteLine(output)

        Console.WriteLine("Time of Travel: {0:%d} day(s)", duration)
        Console.WriteLine("Time of Travel: {0:dd\.hh\:mm\:ss} days", duration)
    End Sub
End Module
' The example displays the following output:
'       Time of Travel: 1 days
'       Time of Travel: 01.12:24:02
'       Time of Travel: 1 day(s)
'       Time of Travel: 01.12:24:02 days
```

自定义 TimeSpan 格式字符串也被 TimeSpan.ParseExact 和 TimeSpan.TryParseExact 方法用于定义分析操作所需的输入字符串格式。（分析将值的字符串表示形式转换成该值。）以下示例演示了标准格式字符串在分析操作中的用法。

```
using System;

public class Example
{
   public static void Main()
   {
      string value = null;
      TimeSpan interval;

      value = "6";
      if (TimeSpan.TryParseExact(value, "%d", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);

      value = "16:32.05";
      if (TimeSpan.TryParseExact(value, @"mm\:ss\.ff", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);

      value= "12.035";
      if (TimeSpan.TryParseExact(value, "ss\\.fff", null, out interval))
         Console.WriteLine("{0} --> {1}", value, interval.ToString("c"));
      else
         Console.WriteLine("Unable to parse '{0}'", value);
   }
}
// The example displays the following output:
//       6 --> 6.00:00:00
//       16:32.05 --> 00:16:32.0500000
//       12.035 --> 00:00:12.0350000
```

```
Module Example
    Public Sub Main()
        Dim value As String = Nothing
        Dim interval As TimeSpan

        value = "6"
        If TimeSpan.TryParseExact(value, "%d", Nothing, interval) Then
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If

        value = "16:32.05"
        If TimeSpan.TryParseExact(value, "mm\:ss\.ff", Nothing, interval) Then
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If

        value = "12.035"
        If TimeSpan.TryParseExact(value, "ss\.fff", Nothing, interval) Then
            Console.WriteLine("{0} --> {1}", value, interval.ToString("c"))
        Else
            Console.WriteLine("Unable to parse '{0}'", value)
        End If
    End Sub
End Module
' The example displays the following output:
'       6 --> 6.00:00:00
'       16:32.05 --> 00:16:32.0500000
'       12.035 --> 00:00:12.0350000
```

下表列出了自定义日期和时间格式说明符。

[illegible]	[illegible]	[illegible]
“d”, “%d”	时间间隔中的整天数。 更多信息：“d”自定义格式说明符。	`new TimeSpan(6, 14, 32, 17, 685):` `%d` --> "6" `d\.hh\:mm` --> "6.14:32"
"dd"-"dddddddd"	时间间隔中的整天数，根据需要使用前导零填充。 更多信息：“dd”-“dddddddd”自定义格式说明符。	`new TimeSpan(6, 14, 32, 17, 685):` `ddd` --> "006" `dd\.hh\:mm` --> "06.14:32"
“h”, “%h”	时间间隔中不计为天数一部分的整小时数。一位数小时数没有前导零。 更多信息：“h”自定义格式说明符。	`new TimeSpan(6, 14, 32, 17, 685):` `%h` --> "14" `hh\:mm` --> "14:32"

[illegible]	[illegible]	[illegible]

“hh”	时间间隔中不计为天数一部分的整小时数。一位数小时具有前导零。
	
	更多信息：“hh”自定义格式说明符。
		`new TimeSpan(6, 14, 32, 17, 685):`
		
		`hh` --> "14"
		
		`new TimeSpan(6, 8, 32, 17, 685):`
		
		`hh` --> 08

“m”, “%m”	时间间隔中不包含在小时或天数中的整分钟数。一位数分钟数没有前导零。 更多信息：“m”自定义格式说明符。	`new TimeSpan(6, 14, 8, 17, 685):` `%m` --> "8" `h\:m` --> "14:8"

“mm”	时间间隔中不包含在小时或天数中的整分钟数。一位数分钟具有前导零。
	
	更多信息：“mm”自定义格式说明符。
		`new TimeSpan(6, 14, 8, 17, 685):`
		
		`mm` --> "08"
		
		`new TimeSpan(6, 8, 5, 17, 685):`
		
		`d\.hh\:mm\:ss` --> 6.08:05:17

“s”, “%s”	时间间隔中不包含在小时、天数或分钟中的整秒数。一位数秒数没有前导零。 更多信息：“s”自定义格式说明符。	`TimeSpan.FromSeconds(12.965)`： `%s` --> 12 `s\.fff` --> 12.965
“ss”	时间间隔中不包含在小时、天数或分钟中的整秒数。一位数秒具有前导零。 更多信息：“ss”自定义格式说明符。	`TimeSpan.FromSeconds(6.965)`： `ss` --> 06 `ss\.fff` --> 06.965
“f”, “%f”	时间间隔中的十分之几秒。 更多信息：“f”自定义格式说明符。	`TimeSpan.FromSeconds(6.895)`： `f` --> 8 `ss\.f` --> 06.8
“ff”	时间间隔中的百分之几秒。 更多信息：“ff”自定义格式说明符。	`TimeSpan.FromSeconds(6.895)`： `ff` --> 89 `ss\.ff` --> 06.89
“fff”	时间间隔中的毫秒。 更多信息：“fff”自定义格式说明符。	`TimeSpan.FromSeconds(6.895)`： `fff` --> 895 `ss\.fff` --> 06.895

fffff	ff	ff
“ffff”	时间间隔中的万分之几秒。 更多信息：“ffff”自定义格式说明符。	`TimeSpan.Parse("0:0:6.8954321")`： `ffff` --> 8954 `ss\.ffff` --> 06.8954
“fffff”	时间间隔中的十万分之几秒。 更多信息：“fffff”自定义格式说明符。	`TimeSpan.Parse("0:0:6.8954321")`： `fffff` --> 89543 `ss\.fffff` --> 06.89543
“ffffff”	时间间隔中的百万分之几秒。 更多信息：“ffffff”自定义格式说明符。	`TimeSpan.Parse("0:0:6.8954321")`： `ffffff` --> 895432 `ss\.ffffff` --> 06.895432
“fffffff”	时间间隔中的千万分之几秒（或小数时钟周期）。 更多信息：“fffffff”自定义格式说明符。	`TimeSpan.Parse("0:0:6.8954321")`： `fffffff` --> 8954321 `ss\.fffffff` --> 06.8954321
“F”，“%F”	时间间隔中的十分之几秒。如果该数字为零，则不显示任何内容。 更多信息：“F”自定义格式说明符。	`TimeSpan.Parse("00:00:06.32")`： `%F`：3 `TimeSpan.Parse("0:0:3.091")`： `ss\.F`：03.
“FF”	时间间隔中的百分之几秒。不包含任何小数尾随零或两个零位。 更多信息：“FF”自定义格式说明符。	`TimeSpan.Parse("00:00:06.329")`： `FF`：32 `TimeSpan.Parse("0:0:3.101")`： `ss\.FF`：03.1
“FFF”	时间间隔中的毫秒。不包含任何小数尾随零。 更多信息：	`TimeSpan.Parse("00:00:06.3291")`： `FFF`：329 `TimeSpan.Parse("0:0:3.1009")`： `ss\.FFF`：03.1

fffff	ff	ff
“FFFF”	时间间隔中的万分之几秒。不包含任何小数尾随零。 更多信息：“FFFF”自定义格式说明符。	`TimeSpan.Parse("00:00:06.32917")`： `FFFFF`：3291 `TimeSpan.Parse("0:0:3.10009")`： `ss\.FFFF`：03.1
“FFFFF”	时间间隔中的十万分之几秒。不包含任何小数尾随零。 更多信息：“FFFFF”自定义格式说明符。	`TimeSpan.Parse("00:00:06.329179")`： `FFFFF`：32917 `TimeSpan.Parse("0:0:3.100009")`： `ss\.FFFFF`：03.1
“FFFFFF”	时间间隔中的百万分之几秒。不显示任何小数尾随零。 更多信息：“FFFFFF”自定义格式说明符。	`TimeSpan.Parse("00:00:06.3291791")`： `FFFFFF`：329179 `TimeSpan.Parse("0:0:3.1000009")`： `ss\.FFFFFF`：03.1
“FFFFFFF”	时间间隔中的千万分之几秒。不显示任何小数尾随零或七个零。 更多信息：“FFFFFFF”自定义格式说明符。	`TimeSpan.Parse("00:00:06.3291791")`： `FFFFFF`：3291791 `TimeSpan.Parse("0:0:3.1900000")`： `ss\.FFFFFF`：03.19
'string'	文本字符串分隔符。 更多信息：其他字符。	`new TimeSpan(14, 32, 17):` `hh':'mm':'ss` --> "14:32:17"
\	转义字符。 更多信息：其他字符。	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"
任何其他字符	任何其他未转义字符会解释为自定义格式说明符。 详细信息：其他字符。	`new TimeSpan(14, 32, 17):` `hh\:mm\:ss` --> "14:32:17"

“d”自定义格式说明符

"d" 自定义格式说明符输出 TimeSpan.Days 属性的值，表示时间间隔中的整天数。它在 TimeSpan 值中输出整天数（即使值有多位，也不例外）。如果 TimeSpan.Days 属性的值为零，说明符输出“0”。

如果“d”自定义格式说明符单独使用，则指定“%d”，以便它不会错误地解释为标准格式字符串。下面的示例进行

了这方面的演示。

```
TimeSpan ts1 = new TimeSpan(16, 4, 3, 17, 250);
Console.WriteLine(ts1.ToString("%d"));
// Displays 16
```

```
Dim ts As New TimeSpan(16, 4, 3, 17, 250)
Console.WriteLine(ts.ToString("%d"))
' Displays 16
```

下面的示例演示如何使用“d”自定义格式说明符。

```
TimeSpan ts2 = new TimeSpan(4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts3 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts3.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.04:03:17
//       3.04:03:17
```

```
Dim ts2 As New TimeSpan(4, 3, 17)
Console.WriteLine(ts2.ToString("d\.hh\:mm\:ss"))

Dim ts3 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts3.ToString("d\.hh\:mm\:ss"))
' The example displays the following output:
'       0.04:03:17
'       3.04:03:17
```

返回表首

“dd”-“ddddddd”自定义格式说明符

"dd"、"ddd"、"dddd"、"ddddd"、"dddddd"、"ddddddd" 和 "dddddddd" 自定义格式说明符输出 TimeSpan.Days 属性的值，表示时间间隔中的整天数。

输出字符串包含格式说明符中的“d”字符数指定的最小位数，会根据需要使用前导零填充。如果天数中的位数超过格式说明符中的“d”字符数，则完整天数会在结果字符串中输出。

下面的示例使用这些格式说明符，显示两个 TimeSpan 值的字符串表示形式。第一个时间间隔的天数部分的值为 0；第二个的天数部分的值为 365。

```
TimeSpan ts1 = new TimeSpan(0, 23, 17, 47);
TimeSpan ts2 = new TimeSpan(365, 21, 19, 45);

for (int ctr = 2; ctr <= 8; ctr++)
{
   string fmt = new String('d', ctr) + @"\.hh\:mm\:ss";
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts1);
   Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts2);
   Console.WriteLine();
}
// The example displays the following output:
//       dd\.hh\:mm\:ss --> 00.23:17:47
//       dd\.hh\:mm\:ss --> 365.21:19:45
//
//       ddd\.hh\:mm\:ss --> 000.23:17:47
//       ddd\.hh\:mm\:ss --> 365.21:19:45
//
//       dddd\.hh\:mm\:ss --> 0000.23:17:47
//       dddd\.hh\:mm\:ss --> 0365.21:19:45
//
//       ddddd\.hh\:mm\:ss --> 00000.23:17:47
//       ddddd\.hh\:mm\:ss --> 00365.21:19:45
//
//       dddddd\.hh\:mm\:ss --> 000000.23:17:47
//       dddddd\.hh\:mm\:ss --> 000365.21:19:45
//
//       ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
//       ddddddd\.hh\:mm\:ss --> 0000365.21:19:45
//
//       dddddddd\.hh\:mm\:ss --> 00000000.23:17:47
//       dddddddd\.hh\:mm\:ss --> 00000365.21:19:45
```

```
Dim ts1 As New TimeSpan(0, 23, 17, 47)
Dim ts2 As New TimeSpan(365, 21, 19, 45)

For ctr As Integer = 2 To 8
    Dim fmt As String = New String("d"c, ctr) + "\.hh\:mm\:ss"
    Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts1)
    Console.WriteLine("{0} --> {1:" + fmt + "}", fmt, ts2)
    Console.WriteLine()
Next
' The example displays the following output:
'       dd\.hh\:mm\:ss --> 00.23:17:47
'       dd\.hh\:mm\:ss --> 365.21:19:45
'
'       ddd\.hh\:mm\:ss --> 000.23:17:47
'       ddd\.hh\:mm\:ss --> 365.21:19:45
'
'       dddd\.hh\:mm\:ss --> 0000.23:17:47
'       dddd\.hh\:mm\:ss --> 0365.21:19:45
'
'       ddddd\.hh\:mm\:ss --> 00000.23:17:47
'       ddddd\.hh\:mm\:ss --> 00365.21:19:45
'
'       dddddd\.hh\:mm\:ss --> 000000.23:17:47
'       dddddd\.hh\:mm\:ss --> 000365.21:19:45
'
'       ddddddd\.hh\:mm\:ss --> 0000000.23:17:47
'       ddddddd\.hh\:mm\:ss --> 0000365.21:19:45
'
'       dddddddd\.hh\:mm\:ss --> 00000000.23:17:47
'       dddddddd\.hh\:mm\:ss --> 00000365.21:19:45
```

返回表首

“h”自定义格式说明符

“h”自定义格式说明符输出 TimeSpan.Hours 属性的值，表示时间间隔中不计入天数的整小时数。如果 TimeSpan.Hours 属性的值介于 0 和 9 之间，它返回一位数字符串值；如果 TimeSpan.Hours 属性的值介于 10 和 23 之间，它返回两位数字符串值。

如果“h”自定义格式说明符单独使用，则指定“%h”，以便它不会错误地解释为标准格式字符串。下面的示例进行了这方面的演示。

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);
// The example displays the following output:
//       3 hours 42 minutes
```

```
Dim ts As New TimeSpan(3, 42, 0)
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts)
' The example displays the following output:
'       3 hours 42 minutes
```

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“%h”自定义格式说明符将数字字符串解释为小时数。下面的示例进行了这方面的演示。

```
string value = "8";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%h", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       08:00:00
```

```
Dim value As String = "8"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%h", Nothing, interval) Then
    Console.WriteLine(interval.ToString("c"))
Else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value)
End If
' The example displays the following output:
'       08:00:00
```

下面的示例演示如何使用“h”自定义格式说明符。

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.h\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.h\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.4:03:17
```

```
Dim ts1 As New TimeSpan(14, 3, 17)
Console.WriteLine(ts1.ToString("d\.h\:mm\:ss"))

Dim ts2 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts2.ToString("d\.h\:mm\:ss"))
' The example displays the following output:
'       0.14:03:17
'       3.4:03:17
```

返回表首

“hh”自定义格式说明符

“hh”自定义格式说明符输出 TimeSpan.Hours 属性的值，表示时间间隔中不计入天数的整小时数。对于 0 到 9 的值，输出字符串包含一个前导零。

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“hh”自定义格式说明符将数字字符串解释为小时数。下面的示例进行了这方面的演示。

```
string value = "08";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "hh", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       08:00:00
```

```
Dim value As String = "08"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "hh", Nothing, interval) Then
    Console.WriteLine(interval.ToString("c"))
Else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value)
End If
' The example displays the following output:
'       08:00:00
```

下面的示例演示如何使用“hh”自定义格式说明符。

```
TimeSpan ts1 = new TimeSpan(14, 3, 17);
Console.WriteLine(ts1.ToString(@"d\.hh\:mm\:ss"));

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine(ts2.ToString(@"d\.hh\:mm\:ss"));
// The example displays the following output:
//       0.14:03:17
//       3.04:03:17
```

```
Dim ts1 As New TimeSpan(14, 3, 17)
Console.WriteLine(ts1.ToString("d\.hh\:mm\:ss"))

Dim ts2 As New TimeSpan(3, 4, 3, 17)
Console.WriteLine(ts2.ToString("d\.hh\:mm\:ss"))
' The example displays the following output:
'       0.14:03:17
'       3.04:03:17
```

返回表首

“m”自定义格式说明符

“m”自定义格式说明符输出 TimeSpan.Minutes 属性的值，表示时间间隔中不计入天数的整分钟数。如果 TimeSpan.Minutes 属性的值介于 0 和 9 之间，它返回一位数字符串值；如果 TimeSpan.Minutes 属性的值介于 10 和 59 之间，它返回两位数字符串值。

如果“m”自定义格式说明符单独使用，则指定“%m”，以便它不会错误地解释为标准格式字符串。下面的示例进行了这方面的演示。

```
TimeSpan ts = new TimeSpan(3, 42, 0);
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts);
// The example displays the following output:
//       3 hours 42 minutes
```

```
Dim ts As New TimeSpan(3, 42, 0)
Console.WriteLine("{0:%h} hours {0:%m} minutes", ts)
' The example displays the following output:
'       3 hours 42 minutes
```

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“%m”自定义格式说明符将数字字符串解释为分钟数。下面的示例进行了这方面的演示。

```
string value = "3";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%m", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:03:00
```

```
Dim value As String = "3"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%m", Nothing, interval) Then
    Console.WriteLine(interval.ToString("c"))
Else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value)
End If
' The example displays the following output:
'       00:03:00
```

下面的示例演示如何使用“m”自定义格式说明符。

```
TimeSpan ts1 = new TimeSpan(0, 6, 32);
Console.WriteLine("{0:m\\:ss} minutes", ts1);

TimeSpan ts2 = new TimeSpan(3, 4, 3, 17);
Console.WriteLine("Elapsed time: {0:m\\:ss}", ts2);
// The example displays the following output:
//       6:32 minutes
//       Elapsed time: 18:44
```

```
Dim ts1 As New TimeSpan(0, 6, 32)
Console.WriteLine("{0:m\:ss} minutes", ts1)

Dim ts2 As New TimeSpan(0, 18, 44)
Console.WriteLine("Elapsed time: {0:m\:ss}", ts2)
' The example displays the following output:
'       6:32 minutes
'       Elapsed time: 18:44
```

返回表首

“mm”自定义格式说明符

“mm”自定义格式说明符输出 TimeSpan.Minutes 属性的值，表示时间间隔中不计入小时数或天数的整分钟数。对于 0 到 9 的值，输出字符串包含一个前导零。

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“mm”自定义格式说明符将数字字符串解释为分钟数。下面的示例进行了这方面的演示。

```
string value = "07";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "mm", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:07:00
```

```
Dim value As String = "05"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "mm", Nothing, interval) Then
    Console.WriteLine(interval.ToString("c"))
Else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value)
End If
' The example displays the following output:
'       00:05:00
```

下面的示例演示如何使用“mm”自定义格式说明符。

```
TimeSpan departTime = new TimeSpan(11, 12, 00);
TimeSpan arriveTime = new TimeSpan(16, 28, 00);
Console.WriteLine("Travel time: {0:hh\\:mm}",
                  arriveTime - departTime);
// The example displays the following output:
//       Travel time: 05:16
```

```
Dim departTime As New TimeSpan(11, 12, 00)
Dim arriveTime As New TimeSpan(16, 28, 00)
Console.WriteLine("Travel time: {0:hh\:mm}",
                  arriveTime - departTime)
' The example displays the following output:
'       Travel time: 05:16
```

返回表首

“s”自定义格式说明符

“s”自定义格式说明符输出 TimeSpan.Seconds 属性的值，表示时间间隔中不计入小时数、天数或分钟数的整秒数。如果 TimeSpan.Seconds 属性的值介于 0 和 9 之间，它返回一位数字符串值；如果 TimeSpan.Seconds 属性的值介于 10 和 59 之间，它返回两位数字符串值。

如果“s”自定义格式说明符单独使用，则指定“%s”，以便它不会错误地解释为标准格式字符串。下面的示例进行了这方面的演示。

```
TimeSpan ts = TimeSpan.FromSeconds(12.465);
Console.WriteLine(ts.ToString("%s"));
// The example displays the following output:
//       12
```

```
Dim ts As TimeSpan = TimeSpan.FromSeconds(12.465)
Console.WriteLine(ts.ToString("%s"))
' The example displays the following output:
'       12
```

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“%s”自定义格式说明符将数字字符串解释为秒数。下面的示例进行了这方面的演示。

```
string value = "9";
TimeSpan interval;
if (TimeSpan.TryParseExact(value, "%s", null, out interval))
   Console.WriteLine(interval.ToString("c"));
else
   Console.WriteLine("Unable to convert '{0}' to a time interval",
                     value);
// The example displays the following output:
//       00:00:09
```

```
Dim value As String = "9"
Dim interval As TimeSpan
If TimeSpan.TryParseExact(value, "%s", Nothing, interval) Then
    Console.WriteLine(interval.ToString("c"))
Else
    Console.WriteLine("Unable to convert '{0}' to a time interval",
                      value)
End If
' The example displays the following output:
'       00:00:09
```

下面的示例演示如何使用“s”自定义格式说明符。

```
TimeSpan startTime = new TimeSpan(0, 12, 30, 15, 0);
TimeSpan endTime = new TimeSpan(0, 12, 30, 21, 3);
Console.WriteLine(@"Elapsed Time: {0:s\:fff} seconds",
                  endTime - startTime);
// The example displays the following output:
//       Elapsed Time: 6:003 seconds
```

```
Dim startTime As New TimeSpan(0, 12, 30, 15, 0)
Dim endTime As New TimeSpan(0, 12, 30, 21, 3)
Console.WriteLine("Elapsed Time: {0:s\:fff} seconds",
                  endTime - startTime)
' The example displays the following output:
'       Elapsed Time: 6:003 seconds
```

返回表首

“ss”自定义格式说明符

“ss”自定义格式说明符输出 TimeSpan.Seconds 属性的值，表示时间间隔中不计入小时数、天数或分钟数的整秒数。对于 0 到 9 的值，输出字符串包含一个前导零。

通常在分析操作中，只包含单个数字的输入字符串会解释为天数。可以改为使用“ss”自定义格式说明符将数字字符串解释为秒数。下面的示例进行了这方面的演示。

```
string[] values = { "49", "9", "06" };
TimeSpan interval;
foreach (string value in values)
{
   if (TimeSpan.TryParseExact(value, "ss", null, out interval))
      Console.WriteLine(interval.ToString("c"));
   else
      Console.WriteLine("Unable to convert '{0}' to a time interval",
                        value);
}
// The example displays the following output:
//       00:00:49
//       Unable to convert '9' to a time interval
//       00:00:06
```

```
Dim values() As String = {"49", "9", "06"}
Dim interval As TimeSpan
For Each value As String In values
    If TimeSpan.TryParseExact(value, "ss", Nothing, interval) Then
        Console.WriteLine(interval.ToString("c"))
    Else
        Console.WriteLine("Unable to convert '{0}' to a time interval",
                          value)
    End If
Next
' The example displays the following output:
'       00:00:49
'       Unable to convert '9' to a time interval
'       00:00:06
```

下面的示例演示如何使用“ss”自定义格式说明符。

```
TimeSpan interval1 = TimeSpan.FromSeconds(12.60);
Console.WriteLine(interval1.ToString(@"ss\.fff"));

TimeSpan interval2 = TimeSpan.FromSeconds(6.485);
Console.WriteLine(interval2.ToString(@"ss\.fff"));
// The example displays the following output:
//       12.600
//       06.485
```

```
Dim interval1 As TimeSpan = TimeSpan.FromSeconds(12.60)
Console.WriteLine(interval1.ToString("ss\.fff"))
Dim interval2 As TimeSpan = TimeSpan.FromSeconds(6.485)
Console.WriteLine(interval2.ToString("ss\.fff"))
' The example displays the following output:
'       12.600
'       06.485
```

返回表首

“f”自定义格式说明符

“f”自定义格式说明符输出时间间隔中的十分之几秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有一个小数位。

如果“f”自定义格式说明符单独使用，则指定“%f”，以便它不会错误地解释为标准格式字符串。

下面的示例使用 "f" 自定义格式说明符在 TimeSpan 值中显示秒的十分之几。“f”首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

返回表首

“ff”自定义格式说明符

“ff”自定义格式说明符输出时间间隔中的百分之几秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有两个小数位。

下面的示例使用 "ff" 自定义格式说明符在 TimeSpan 值中显示秒的百分之几。“ff”首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'             %f: 8
'             ff: 87
'            fff: 876
'           ffff: 8765
'          fffff: 87654
'         ffffff: 876543
'        fffffff: 8765432
'
'            s\.f: 29.8
'           s\.ff: 29.87
'          s\.fff: 29.876
'         s\.ffff: 29.8765
'        s\.fffff: 29.87654
'       s\.ffffff: 29.876543
'      s\.fffffff: 29.8765432
```

返回表首

“fff”自定义格式说明符

“fff”自定义格式说明符（包含三个“f”字符）输出时间间隔中的毫秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有三个小数位。

下面的示例使用 "fff" 自定义格式说明符在 TimeSpan 值中显示毫秒。“fff”首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

返回表首

“ffff”自定义格式说明符

“ffff”自定义格式说明符（包含四个“f”字符）输出时间间隔中的万分之几秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有四个小数位。

下面的示例使用 "ffff" 自定义格式说明符在 TimeSpan 值中显示秒的万分之几。“ffff”首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'             %f: 8
'             ff: 87
'            fff: 876
'           ffff: 8765
'          fffff: 87654
'         ffffff: 876543
'        fffffff: 8765432
'
'            s\.f: 29.8
'           s\.ff: 29.87
'          s\.fff: 29.876
'         s\.ffff: 29.8765
'        s\.fffff: 29.87654
'       s\.ffffff: 29.876543
'      s\.fffffff: 29.8765432
```

返回表首

“fffff”自定义格式说明符

“fffff”自定义格式说明符（包含五个“f”字符）输出时间间隔中的十万分之几秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有五个小数位。

下面的示例使用 "fffff" 自定义格式说明符在 TimeSpan 值中显示秒的十万分之几。“fffff”首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'            ff: 87
'           fff: 876
'          ffff: 8765
'         fffff: 87654
'        ffffff: 876543
'       fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

返回表首

“ffffff”自定义格式说明符

“ffffff”自定义格式说明符（包含六个“f”字符）输出时间间隔中的百万分之几秒。在格式设置操作中，会截断其余任何小数位数。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有六个小数位。

下面的示例使用 "ffffff" 自定义格式说明符在 TimeSpan 值中显示秒的百万分之几。它首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'             %f: 8
'             ff: 87
'            fff: 876
'           ffff: 8765
'          fffff: 87654
'         ffffff: 876543
'        fffffff: 8765432
'
'            s\.f: 29.8
'           s\.ff: 29.87
'          s\.fff: 29.876
'         s\.ffff: 29.8765
'        s\.fffff: 29.87654
'       s\.ffffff: 29.876543
'      s\.fffffff: 29.8765432
```

返回表首

“fffffff”自定义格式说明符

“fffffff”自定义格式说明符（包含六个“f”字符）输出时间间隔中的千万分之几秒（或时钟周期的小数）。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，输入字符串必须只有七个小数位。

下面的示例使用 "fffffff" 自定义格式说明符在 TimeSpan 值中显示秒的千万分之一。它首先用作唯一的格式说明符，然后在自定义格式字符串中与“s”说明符结合使用。

```
TimeSpan ts = new TimeSpan(1003498765432);
string fmt;
Console.WriteLine(ts.ToString("c"));
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   if (fmt.Length == 1) fmt = "%" + fmt;
   Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts);
}
Console.WriteLine();

for (int ctr = 1; ctr <= 7; ctr++) {
   fmt = new String('f', ctr);
   Console.WriteLine("{0,10}: {1:s\\." + fmt + "}", "s\\." + fmt, ts);
}
// The example displays the following output:
//               %f: 8
//               ff: 87
//              fff: 876
//             ffff: 8765
//            fffff: 87654
//           ffffff: 876543
//          fffffff: 8765432
//
//              s\.f: 29.8
//             s\.ff: 29.87
//            s\.fff: 29.876
//           s\.ffff: 29.8765
//          s\.fffff: 29.87654
//         s\.ffffff: 29.876543
//        s\.fffffff: 29.8765432
```

```
Dim ts As New TimeSpan(1003498765432)
Dim fmt As String
Console.WriteLine(ts.ToString("c"))
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    If fmt.Length = 1 Then fmt = "%" + fmt
    Console.WriteLine("{0,10}: {1:" + fmt + "}", fmt, ts)
Next
Console.WriteLine()

For ctr = 1 To 7
    fmt = New String("f"c, ctr)
    Console.WriteLine("{0,10}: {1:s\." + fmt + "}", "s\." + fmt, ts)
Next
' The example displays the following output:
'            %f: 8
'             ff: 87
'            fff: 876
'           ffff: 8765
'          fffff: 87654
'         ffffff: 876543
'        fffffff: 8765432
'
'           s\.f: 29.8
'          s\.ff: 29.87
'         s\.fff: 29.876
'        s\.ffff: 29.8765
'       s\.fffff: 29.87654
'      s\.ffffff: 29.876543
'     s\.fffffff: 29.8765432
```

返回表首

“F”自定义格式说明符

“F”自定义格式说明符输出时间间隔中的十分之几秒。在格式设置操作中，会截断其余任何小数位数。如果时间间隔的十分之几秒的值为零，则它不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几。

如果“F”自定义格式说明符单独使用，则指定“%F”，以便它不会错误地解释为标准格式字符串。

下面的示例使用 "F" 自定义格式说明符在 TimeSpan 值中显示秒的十分之几。它还在分析操作中使用此自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.669");
Console.WriteLine("{0} ('%F') --> {0:%F}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.091");
Console.WriteLine("{0} ('ss\\.F') --> {0:ss\\.F}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.12" };
string fmt = @"h\:m\:ss\.F";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6690000 ('%F') --> 6
//       00:00:03.0910000 ('ss\.F') --> 03.
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.F') --> 00:00:03
//       0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.1000000
//       Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.669")
Console.WriteLine("{0} ('%F') --> {0:%F}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.091")
Console.WriteLine("{0} ('ss\.F') --> {0:ss\.F}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.1", "0:0:03.12"}
Dim fmt As String = "h\:m\:ss\.F"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6690000 ('%F') --> 6
'       00:00:03.0910000 ('ss\.F') --> 03.
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.F') --> 00:00:03
'       0:0:03.1 ('h\:m\:ss\.F') --> 00:00:03.1000000
'       Cannot parse 0:0:03.12 with 'h\:m\:ss\.F'.
```

返回表首

“FF”自定义格式说明符

“FF”自定义格式说明符输出时间间隔中的百分之几秒。在格式设置操作中，会截断其余任何小数位数。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几和百分之几。

下面的示例使用 "FF" 自定义格式说明符在 TimeSpan 值中显示秒的百分之几。它还在分析操作中使用此自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697");
Console.WriteLine("{0} ('FF') --> {0:FF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.809");
Console.WriteLine("{0} ('ss\\.FF') --> {0:ss\\.FF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.1", "0:0:03.127" };
string fmt = @"h\:m\:ss\.FF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6970000 ('FF') --> 69
//       00:00:03.8090000 ('ss\.FF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
//       0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
//       Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.697")
Console.WriteLine("{0} ('FF') --> {0:FF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.809")
Console.WriteLine("{0} ('ss\.FF') --> {0:ss\.FF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.1", "0:0:03.127"}
Dim fmt As String = "h\:m\:ss\.FF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6970000 ('FF') --> 69
'       00:00:03.8090000 ('ss\.FF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FF') --> 00:00:03
'       0:0:03.1 ('h\:m\:ss\.FF') --> 00:00:03.1000000
'       Cannot parse 0:0:03.127 with 'h\:m\:ss\.FF'.
```

返回表首

“FFF”自定义格式说明符

“FFF”自定义格式说明符（包含三个“F”字符）输出时间间隔中的毫秒。在格式设置操作中，会截断其余任何小数位数。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几、百分之几和千分之几。

下面的示例使用 "FFF" 自定义格式说明符在 TimeSpan 值中显示秒的千分之几。它还在分析操作中使用此自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974");
Console.WriteLine("{0} ('FFF') --> {0:FFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8009");
Console.WriteLine("{0} ('ss\\.FFF') --> {0:ss\\.FFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279" };
string fmt = @"h\:m\:ss\.FFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974000 ('FFF') --> 697
//       00:00:03.8009000 ('ss\.FFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974")
Console.WriteLine("{0} ('FFF') --> {0:FFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.8009")
Console.WriteLine("{0} ('ss\.FFF') --> {0:ss\.FFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.12", "0:0:03.1279"}
Dim fmt As String = "h\:m\:ss\.FFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974000 ('FFF') --> 697
'       00:00:03.8009000 ('ss\.FFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.1279 with 'h\:m\:ss\.FFF'.
```

返回表首

“FFFF”自定义格式说明符

“FFFF”自定义格式说明符（包含四个“F”字符）输出时间间隔中的万分之几秒。在格式设置操作中，会截断其余任何小数位数。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几、百分之几、千分之几和万分之几。

下面的示例使用 "FFFF" 自定义格式说明符在 TimeSpan 值中显示秒的万分之几。它还在分析操作中使用“FFFF”自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.69749");
Console.WriteLine("{0} ('FFFF') --> {0:FFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.80009");
Console.WriteLine("{0} ('ss\\.FFFF') --> {0:ss\\.FFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.12795" };
string fmt = @"h\:m\:ss\.FFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974900 ('FFFF') --> 6974
//       00:00:03.8000900 ('ss\.FFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.69749")
Console.WriteLine("{0} ('FFFF') --> {0:FFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.80009")
Console.WriteLine("{0} ('ss\.FFFF') --> {0:ss\.FFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.12", "0:0:03.12795"}
Dim fmt As String = "h\:m\:ss\.FFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974900 ('FFFF') --> 6974
'       00:00:03.8000900 ('ss\.FFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.12795 with 'h\:m\:ss\.FFFF'.
```

返回表首

“FFFFF”自定义格式说明符

“FFFFF”自定义格式说明符（包含五个“F”字符）输出时间间隔中的十万分之几秒。在格式设置操作中，会截断其余任何小数位数。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几、百分之几、千分之几、万分之几和十万分之几。

下面的示例使用 "FFFFF" 自定义格式说明符在 TimeSpan 值中显示秒的十万分之几。它还在分析操作中使用“FFFFF”自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.697497");
Console.WriteLine("{0} ('FFFFF') --> {0:FFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.800009");
Console.WriteLine("{0} ('ss\\.FFFFF') --> {0:ss\\.FFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.127956" };
string fmt = @"h\:m\:ss\.FFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974970 ('FFFFF') --> 69749
//       00:00:03.8000090 ('ss\.FFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.697497")
Console.WriteLine("{0} ('FFFFF') --> {0:FFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.800009")
Console.WriteLine("{0} ('ss\.FFFFF') --> {0:ss\.FFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.12", "0:0:03.127956"}
Dim fmt As String = "h\:m\:ss\.FFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974970 ('FFFFF') --> 69749
'       00:00:03.8000090 ('ss\.FFFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.127956 with 'h\:m\:ss\.FFFF'.
```

返回表首

“FFFFFF”自定义格式说明符

“FFFFFF”自定义格式说明符（包含六个“F”字符）输出时间间隔中的百万分之几秒。在格式设置操作中，会截断其余任何小数位数。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否显示秒的十分之几、百分之几、千分之几、万分之几、十万分之几和百万分之几。

下面的示例使用 "FFFFFF" 自定义格式说明符在 TimeSpan 值中显示秒的百万分之几。它还在分析操作中使用此自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine("{0} ('FFFFFF') --> {0:FFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.8000009");
Console.WriteLine("{0} ('ss\\.FFFFFF') --> {0:ss\\.FFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//       Formatting:
//       00:00:03.6974974 ('FFFFFF') --> 697497
//       00:00:03.8000009 ('ss\.FFFFFF') --> 03.8
//
//       Parsing:
//       0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
//       0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
//       Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974974")
Console.WriteLine("{0} ('FFFFFF') --> {0:FFFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.8000009")
Console.WriteLine("{0} ('ss\.FFFFFF') --> {0:ss\.FFFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.12", "0:0:03.1279569"}
Dim fmt As String = "h\:m\:ss\.FFFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'       Formatting:
'       00:00:03.6974974 ('FFFFFF') --> 697497
'       00:00:03.8000009 ('ss\.FFFFFF') --> 03.8
'
'       Parsing:
'       0:0:03. ('h\:m\:ss\.FFFFFF') --> 00:00:03
'       0:0:03.12 ('h\:m\:ss\.FFFFFF') --> 00:00:03.1200000
'       Cannot parse 0:0:03.1279569 with 'h\:m\:ss\.FFFFFF'.
```

返回表首

“FFFFFFF”自定义格式说明符

“FFFFFFF”自定义格式说明符（包含六个“F”字符）输出时间间隔中的千万分之几秒（或时钟周期的小数）。如果存在任何尾随小数零，则它们不包含在结果字符串中。在调用 TimeSpan.ParseExact 或 TimeSpan.TryParseExact 方法的分析操作中，可以视需要选择是否在输入字符串中显示秒的千万分之一。

下面的示例使用 "FFFFFFF" 自定义格式说明符在 TimeSpan 值中显示秒的千万分之一。它还在分析操作中使用此自定义格式说明符。

```
Console.WriteLine("Formatting:");
TimeSpan ts1 = TimeSpan.Parse("0:0:3.6974974");
Console.WriteLine("{0} ('FFFFFFF') --> {0:FFFFFFF}", ts1);

TimeSpan ts2 = TimeSpan.Parse("0:0:3.9500000");
Console.WriteLine("{0} ('ss\\.FFFFFFF') --> {0:ss\\.FFFFFFF}", ts2);
Console.WriteLine();

Console.WriteLine("Parsing:");
string[] inputs = { "0:0:03.", "0:0:03.12", "0:0:03.1279569" };
string fmt = @"h\:m\:ss\.FFFFFFF";
TimeSpan ts3;

foreach (string input in inputs) {
   if (TimeSpan.TryParseExact(input, fmt, null, out ts3))
      Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3);
   else
      Console.WriteLine("Cannot parse {0} with '{1}'.",
                        input, fmt);
}
// The example displays the following output:
//    Formatting:
//    00:00:03.6974974 ('FFFFFFF') --> 6974974
//    00:00:03.9500000 ('ss\.FFFFFFF') --> 03.95
//
//    Parsing:
//    0:0:03. ('h\:m\:ss\.FFFFFFF') --> 00:00:03
//    0:0:03.12 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1200000
//    0:0:03.1279569 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1279569
```

```
Console.WriteLine("Formatting:")
Dim ts1 As TimeSpan = TimeSpan.Parse("0:0:3.6974974")
Console.WriteLine("{0} ('FFFFFFF') --> {0:FFFFFFF}", ts1)

Dim ts2 As TimeSpan = TimeSpan.Parse("0:0:3.9500000")
Console.WriteLine("{0} ('ss\.FFFFFFF') --> {0:ss\.FFFFFFF}", ts2)
Console.WriteLine()

Console.WriteLine("Parsing:")
Dim inputs() As String = {"0:0:03.", "0:0:03.12", "0:0:03.1279569"}
Dim fmt As String = "h\:m\:ss\.FFFFFFF"
Dim ts3 As TimeSpan

For Each input As String In inputs
    If TimeSpan.TryParseExact(input, fmt, Nothing, ts3)
        Console.WriteLine("{0} ('{1}') --> {2}", input, fmt, ts3)
    Else
        Console.WriteLine("Cannot parse {0} with '{1}'.",
                          input, fmt)
    End If
Next
' The example displays the following output:
'    Formatting:
'    00:00:03.6974974 ('FFFFFFF') --> 6974974
'    00:00:03.9500000 ('ss\.FFFFFFF') --> 03.95
'
'    Parsing:
'    0:0:03. ('h\:m\:ss\.FFFFFFF') --> 00:00:03
'    0:0:03.12 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1200000
'    0:0:03.1279569 ('h\:m\:ss\.FFFFFFF') --> 00:00:03.1279569
```

返回表首

其他字符

格式字符串中的任何其他未转义字符(包括空白字符)都会解释为自定义格式说明符。大多数情况下，若有其他任何未转义字符，便会导致 FormatException 抛出。

可通过两种方法在格式字符串中包含文本字符：

- 将它括在单引号(文本字符串分隔符)中。
- 在它前面加上一个反斜杠("\")，即可将它解释为转义字符。这意味着在 C# 中，格式字符串必须是 @-quoted，或者文本字符的前面必须加上额外的反斜杠。

  在某些情况下，可能需要使用条件逻辑才能在格式字符串中包括转义文本。下面的示例使用条件逻辑包括表示负时间间隔的正负符号。

```
using System;

public class Example
{
   public static void Main()
   {
      TimeSpan result = new DateTime(2010, 01, 01) - DateTime.Now;
      String fmt = (result < TimeSpan.Zero ?  "\\-" : "") + "dd\\.hh\\:mm";

      Console.WriteLine(result.ToString(fmt));
      Console.WriteLine("Interval: {0:" + fmt + "}", result);
   }
}
// The example displays output like the following:
//       -1291.10:54
//       Interval: -1291.10:54
```

```
Module Example
    Public Sub Main()
        Dim result As TimeSpan = New DateTime(2010, 01, 01) - Date.Now
        Dim fmt As String = If(result < TimeSpan.Zero, "\-", "") + "dd\.hh\:mm"

        Console.WriteLine(result.ToString(fmt))
        Console.WriteLine("Interval: {0:" + fmt + "}", result)
    End Sub
End Module
' The example displays output like the following:
'       -1291.10:54
'       Interval: -1291.10:54
```

.NET 没有为时间间隔中的分隔符定义语法。这意味着在格式字符串中，天与小时、小时与分钟、分钟与秒以及秒与秒的小数之间的分隔符必须全都被视为字符文本。

下面的示例使用转义字符和单引号来定义在输出字符串中包含"minutes"一词的自定义格式字符串。

```
TimeSpan interval = new TimeSpan(0, 32, 45);
// Escape literal characters in a format string.
string fmt = @"mm\:ss\ \m\i\n\u\t\e\s";
Console.WriteLine(interval.ToString(fmt));
// Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'";
Console.WriteLine(interval.ToString(fmt));
// The example displays the following output:
//       32:45 minutes
//       32:45 minutes
```

```
Dim interval As New TimeSpan(0, 32, 45)
' Escape literal characters in a format string.
Dim fmt As String = "mm\:ss\ \m\i\n\u\t\e\s"
Console.WriteLine(interval.ToString(fmt))
' Delimit literal characters in a format string with the ' symbol.
fmt = "mm':'ss' minutes'"
Console.WriteLine(interval.ToString(fmt))
' The example displays the following output:
'       32:45 minutes
'       32:45 minutes
```

返回表首

请参阅

- 格式设置类型
- 标准 TimeSpan 格式字符串

枚举格式字符串

2021/11/16 •

可以使用 Enum.ToString 方法，新建表示枚举成员的数字值、十六进制值或字符串值的字符串对象。此方法采用枚举格式设置字符串之一来指定要返回的值。

以下各部分列出了枚举格式设置字符串和它们返回的值。这些格式说明符不区分大小写。

G 或 g

如有可能，将枚举项显示为字符串值，否则显示当前实例的整数值。如果枚举使用 Flags 属性集进行定义，则每个有效项的字符串值会连接在一起（以逗号分隔）。如果未设置 Flags 属性，则将无效值显示为数字项。下面的示例演示 G 格式说明符。

```
Console.WriteLine(ConsoleColor.Red.ToString("G"));         // Displays Red
FileAttributes attributes = FileAttributes.Hidden |
                            FileAttributes.Archive;
Console.WriteLine(attributes.ToString("G"));   // Displays Hidden, Archive
```

```
Console.WriteLine(ConsoleColor.Red.ToString("G"))           ' Displays Red
Dim attributes As FileAttributes = FileAttributes.Hidden Or _
                                   FileAttributes.Archive
Console.WriteLine(attributes.ToString("G"))     ' Displays Hidden, Archive
```

F 或 f

如有可能，将枚举项显示为字符串值。如果值可以完全显示为枚举中项的总和（即使未提供 Flags 属性），则每个有效项的字符串值会连接在一起（以逗号分隔）。如果值不能由枚举项完全确定，则值会格式化为整数值。下面的示例演示 F 格式说明符。

```
Console.WriteLine(ConsoleColor.Blue.ToString("F"));       // Displays Blue
FileAttributes attributes = FileAttributes.Hidden |
                            FileAttributes.Archive;
Console.WriteLine(attributes.ToString("F"));   // Displays Hidden, Archive
```

```
Console.WriteLine(ConsoleColor.Blue.ToString("F"))         ' Displays Blue
Dim attributes As FileAttributes = FileAttributes.Hidden Or _
                                   FileAttributes.Archive
Console.WriteLine(attributes.ToString("F"))     ' Displays Hidden, Archive
```

D 或 d

以尽可能短的表示形式将枚举项显示为整数值。下面的示例演示 D 格式说明符。

```
Console.WriteLine(ConsoleColor.Cyan.ToString("D"));         // Displays 11
FileAttributes attributes = FileAttributes.Hidden |
                            FileAttributes.Archive;
Console.WriteLine(attributes.ToString("D"));                // Displays 34
```

```
Console.WriteLine(ConsoleColor.Cyan.ToString("D"))          ' Displays 11
Dim attributes As FileAttributes = FileAttributes.Hidden Or _
                                   FileAttributes.Archive
Console.WriteLine(attributes.ToString("D"))                 ' Displays 34
```

X 或 x

将枚举项显示为十六进制值。根据需要以前导零表示此值，以确保在枚举类型的基础数值类型中，结果字符串的每个字节都有两个字符。下面的示例演示 X 格式说明符。在示例中，这两者的基础类型 ConsoleColor 和 FileAttributes 为 Int32，或 32 位（或 4 字节）整数，它将生成 8 个字符的结果字符串。

```
Console.WriteLine(ConsoleColor.Cyan.ToString("X"));   // Displays 0000000B
FileAttributes attributes = FileAttributes.Hidden |
                            FileAttributes.Archive;
Console.WriteLine(attributes.ToString("X"));          // Displays 00000022
```

```
Console.WriteLine(ConsoleColor.Cyan.ToString("X"))     ' Displays 0000000B
Dim attributes As FileAttributes = FileAttributes.Hidden Or _
                                   FileAttributes.Archive
Console.WriteLine(attributes.ToString("X"))            ' Displays 00000022
```

示例

下面的示例定义一个名为 `Colors` 的枚举，它由三个项组成：`Red`、`Blue` 和 `Green`。

```
public enum Color {Red = 1, Blue = 2, Green = 3}
```

```
Public Enum Color
    Red = 1
    Blue = 2
    Green = 3
End Enum
```

定义该枚举之后，可以按以下方式声明实例。

```
Color myColor = Color.Green;
```

```
Dim myColor As Color = Color.Green
```

`Color.ToString(System.String)` 方法随后可以用于以不同方式显示枚举值（具体取决于传递给它的格式说明符）。

```
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("G"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("F"));
Console.WriteLine("The value of myColor is {0}.",
                  myColor.ToString("D"));
Console.WriteLine("The value of myColor is 0x{0}.",
                  myColor.ToString("X"));
// The example displays the following output to the console:
//       The value of myColor is Green.
//       The value of myColor is Green.
//       The value of myColor is 3.
//       The value of myColor is 0x00000003.
```

```
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("G"))
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("F"))
Console.WriteLine("The value of myColor is {0}.", _
                  myColor.ToString("D"))
Console.WriteLine("The value of myColor is 0x{0}.", _
                  myColor.ToString("X"))
' The example displays the following output to the console:
'       The value of myColor is Green.
'       The value of myColor is Green.
'       The value of myColor is 3.
'       The value of myColor is 0x00000003.
```

请参阅

- 格式设置类型

复合格式设置

2021/11/16 •

.NET 复合格式设置功能使用对象列表和复合格式字符串作为输入。复合格式字符串由固定文本和索引占位符混和组成，其中索引占位符称为格式项，对应于列表中的对象。格式设置操作产生的结果字符串由原始固定文本和列表中对象的字符串表示形式混和组成。

> **IMPORTANT**
>
> 相较使用复合格式字符串，如果正在使用的语言和语言版本支持，则可使用 *内插字符串*。内插字符串是包含内插表达式的字符串。每个内插表达式都使用表达式的值进行解析，并在分配字符串时包含在结果字符串中。有关详细信息，请参阅字符串内插（C# 参考）和内插字符串（Visual Basic 参考）。

复合格式设置功能受诸如以下方法的支持：

- String.Format，它返回格式化的结果字符串。
- StringBuilder.AppendFormat，它将格式化的结果字符串追加到 StringBuilder 对象。
- Console.WriteLine 方法的某些重载，它将格式化的结果字符串显示到控制台上。
- TextWriter.WriteLine 方法的某些重载，它将格式化的结果字符串写入流或文件中。派生自 TextWriter 的类（如 StreamWriter 和 HtmlTextWriter）也共享此功能。
- Debug.WriteLine(String, Object[])，它将格式化消息输出到跟踪侦听器。
- Trace.TraceError(String, Object[])、Trace.TraceInformation(String, Object[]) 和 Trace.TraceWarning(String, Object[]) 方法，它们将格式化消息输出到跟踪侦听器。
- TraceSource.TraceInformation(String, Object[]) 方法，它将信息性方法写入跟踪侦听器中。

复合格式字符串

复合格式字符串和对象列表将用作支持复合格式设置功能的方法的参数。复合格式字符串由零个或多个固定文本段与一个或多个格式项混和组成。固定文本是所选择的任何字符串，并且每个格式项对应于列表中的一个对象或装箱的结构。复合格式设置功能返回新的结果字符串，其中每个格式项都被列表中相应对象的字符串表示形式取代。

可考虑使用以下 Format 代码段。

```
string name = "Fred";
String.Format("Name = {0}, hours = {1:hh}", name, DateTime.Now);
```

```
Dim name As String = "Fred"
String.Format("Name = {0}, hours = {1:hh}", name, DateTime.Now)
```

固定文本为“`Name =`”和“`, hours =`”。格式项为“`{0}`”和“`{1:hh}`”，前者的索引为 0，对应于对象 `name`，后者的索引为 1，对应于对象 `DateTime.Now`。

格式项语法

每个格式项都采用下面的形式并包含以下组件：

`{` *index*[`,` *alignment*][`:` *formatString*] `}`

必须使用成对的大括号（“{”和“}”）。

索引组件

必需的 *索引* 组件（也叫参数说明符）是一个从 0 开始的数字，可标识对象列表中对应的项。也就是说，参数说明符为 0 的格式项列表中的第一个对象，参数说明符为 1 的格式项列表中的第二个对象，依次类推。下面的示例包括四个参数说明符，编号为 0 到 3，用于表示小于 10 的质数：

```
string primes;
primes = String.Format("Prime numbers less than 10: {0}, {1}, {2}, {3}",
                       2, 3, 5, 7 );
Console.WriteLine(primes);
// The example displays the following output:
//      Prime numbers less than 10: 2, 3, 5, 7
```

```
Dim primes As String
primes = String.Format("Prime numbers less than 10: {0}, {1}, {2}, {3}",
                       2, 3, 5, 7)
Console.WriteLine(primes)
' The example displays the following output:
'      Prime numbers less than 10: 2, 3, 5, 7
```

通过指定相同的参数说明符，多个格式项可以引用对象列表中的同一个元素。例如，可以将同一个数值设置为十六进制、科学记数法和数字格式，方法是通过指定复合格式字符串（例如：“0x{0:X} {0:E} {0:N}”），如以下示例所示。

```
string multiple = String.Format("0x{0:X} {0:E} {0:N}",
                                Int64.MaxValue);
Console.WriteLine(multiple);
// The example displays the following output:
//      0x7FFFFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

```
Dim multiple As String = String.Format("0x{0:X} {0:E} {0:N}",
                                       Int64.MaxValue)
Console.WriteLine(multiple)
' The example displays the following output:
'      0x7FFFFFFFFFFFFFFF 9.223372E+018 9,223,372,036,854,775,807.00
```

每个格式项都可以引用列表中的任一对象。例如，如果有三个对象，可以指定“{1} {0} {2}”等复合格式字符串，以设置第二个、第一个和第三个对象的格式。格式项未引用的对象会被忽略。如果参数说明符指定了超出对象列表范围的项，将引发运行时 FormatException。

对齐组件

可选的 *对齐* 组件是一个带符号的整数，指示首选的设置了格式的字段宽度。如果 *alignment* 值小于设置了格式的字符串的长度，*alignment* 将被忽略，并使用设置了格式的字符串的长度作为字段宽度。如果 *alignment* 为正数，字段中设置了格式的数据为右对齐；如果 *alignment* 为负数，字段中的设置了格式的数据为左对齐。如果需要填充，则使用空白。如果指定 *alignment*，则需要使用逗号。

下面的示例定义两个数组，一个包含雇员的姓名，另一个则包含雇员在两周内的工作小时数。复合格式字符串使 20 字符字段中的姓名左对齐，使 5 字符字段中的工作小时数右对齐。请注意“N1”标准格式字符串还用于设置带有小数位的小时数格式。

```
using System;

public class Example
{
   public static void Main()
   {
      string[] names = { "Adam", "Bridgette", "Carla", "Daniel",
                         "Ebenezer", "Francine", "George" };
      decimal[] hours = { 40, 6.667m, 40.39m, 82, 40.333m, 80,
                                16.75m };

      Console.WriteLine("{0,-20} {1,5}\n", "Name", "Hours");
      for (int ctr = 0; ctr < names.Length; ctr++)
         Console.WriteLine("{0,-20} {1,5:N1}", names[ctr], hours[ctr]);
   }
}
// The example displays the following output:
//       Name                 Hours
//
//       Adam                  40.0
//       Bridgette              6.7
//       Carla                 40.4
//       Daniel                82.0
//       Ebenezer              40.3
//       Francine              80.0
//       George                16.8
```

```
Module Example
    Public Sub Main()
        Dim names() As String = {"Adam", "Bridgette", "Carla", "Daniel",
                                 "Ebenezer", "Francine", "George"}
        Dim hours() As Decimal = {40, 6.667d, 40.39d, 82, 40.333d, 80,
                                  16.75d}

        Console.WriteLine("{0,-20} {1,5}", "Name", "Hours")
        Console.WriteLine()
        For ctr As Integer = 0 To names.Length - 1
            Console.WriteLine("{0,-20} {1,5:N1}", names(ctr), hours(ctr))
        Next
    End Sub
End Module
' The example displays the following output:
'       Name                 Hours
'
'       Adam                  40.0
'       Bridgette              6.7
'       Carla                 40.4
'       Daniel                82.0
'       Ebenezer              40.3
'       Francine              80.0
'       George                16.8
```

格式字符串组件

可选的 *格式字符串* 组件是适合正在设置格式的对象类型的格式字符串。如果相应对象是数值，指定标准或自定义数字格式字符串；如果相应对象是 DateTime 对象，指定标准或自定义日期和时间格式字符串；或者，如果相应对象是枚举值，指定枚举格式字符串。如果不指定 *formatString*，则对数字、日期和时间或者枚举类型使用常规（“G”）格式说明符。如果指定 *formatString*，则需要使用冒号。

下表列出了 .NET 类库中支持预定义的格式字符串集的类型或类型的类别，并提供指向列出了支持的格式字符串的主题的链接。请注意，字符串格式化是一个可扩展的机制，可使用该机制定义所有现有类型的新的格式字符串，并定义受应用程序定义的类型支持的格式字符串集。有关详细信息，请参阅 IFormattable 和 ICustomFormatter 接口主题。

tttttttt	tt
日期和时间类型(DateTime, DateTimeOffset)	标准日期和时间格式字符串 自定义日期和时间格式字符串
枚举类型(所有派生自 System.Enum 的类型)	枚举格式字符串
数值类型 (BigInteger、Byte、Decimal、Double、Int16、Int32、Int64、SByte、Single、UInt16、UInt32、UInt64)	标准数字格式字符串 自定义数字格式字符串
Guid	Guid.ToString(String)
TimeSpan	标准 TimeSpan 格式字符串 自定义 TimeSpan 格式字符串

转义大括号

左大括号和右大括号被解释为格式项的开始和结束。因此，必须使用转义序列显示文本左大括号或右大括号。在固定文本中指定两个左大括号 ("{{") 以显示一个左大括号 ("{")，或指定两个右大括号 ("}}") 以显示一个右大括号 ("}")。按照在格式项中遇到大括号的顺序依次解释它们。不支持解释嵌套的大括号。

解释转义大括号的方式会导致意外的结果。例如，假设格式项为“{{{0:D}}}”，旨在显示左大括号、采用十进制数格式的数值和右大括号。但是，实际是按照以下方式解释该格式项：

1. 前两个左大括号 ("{{") 被转义，生成一个左大括号。
2. 之后的三个字符 ("{0:") 被解释为格式项的开始。
3. 下一个字符 ("D") 将被解释为 Decimal 标准数值格式说明符，但后面的两个转义大括号 ("}}") 生成单个大括号。由于得到的字符串 ("D}") 不是标准数值格式说明符号，所以得到的字符串会被解释为用于显示字符串“D}”的自定义格式字符串。
4. 最后一个大括号 ("}") 被解释为格式项的结束。
5. 显示的最终结果是字符串“{D}”。不会显示本来要设置格式的数值。

在编写代码时，避免错误解释转义大括号和格式项的一种方法是单独设置大括号和格式项的格式。也就是说，在第一个格式设置操作中显示文本左大括号，在下一操作中显示格式项的结果，然后在最后一个操作中显示文本右大括号。下面的示例阐释了这种方法。

```
int value = 6324;
string output = string.Format("{0}{1:D}{2}",
                              "{", value, "}");
Console.WriteLine(output);
// The example displays the following output:
//       {6324}
```

```
Dim value As Integer = 6324
Dim output As String = String.Format("{0}{1:D}{2}", _
                                     "{", value, "}")
Console.WriteLine(output)
' The example displays the following output:
'       {6324}
```

处理顺序

如果对复合格式设置方法的调用包括其值不为 `null` 的IFormatProvider参数，则运行时会调用其 IFormatProvider.GetFormat 方法来请求 ICustomFormatter 实现。如果此方法能够返回 ICustomFormatter 实现，那么它将在复合格式方法调用期间缓存。

如下所示，将参数列表中与格式项对应的每个值转换为字符串：

1. 如果要设置格式的值为 `null`，则将返回空字符串 String.Empty。
2. 如果 ICustomFormatter 实现可用，则运行时将调用其 Format 方法。它向方法传递格式项的 formatString 值（若有）或 `null`（若无）以及 IFormatProvider 实现。如果对 ICustomFormatter.Format 方法的调用返回 `null`，则继续执行下一步骤，将返回 ICustomFormatter.Format 调用的结果。
3. 如果该值实现 IFormattable 接口，则调用此接口的 ToString(String, IFormatProvider) 方法。如果格式项中存在 formatString 值，则向方法传递该值；如果不存在该值，则传递 `null`。按如下方式确定 IFormatProvider 自变量：
   - 对于数值，如果调用带非 null IFormatProvider 自变量的复合格式设置方法，则运行时从其 NumberFormatInfo 方法请求 IFormatProvider.GetFormat 对象。在以下情况下，使用当前线程区域性的 NumberFormatInfo 对象：无法提供该值、参数值为 `null` 或复合格式设置方法没有 IFormatProvider 参数。
   - 对于日期和时间值，如果调用带非 null IFormatProvider 自变量的复合格式设置方法，则运行时从其 DateTimeFormatInfo 方法请求 IFormatProvider.GetFormat 对象。在以下情况下，使用当前线程区域性的 DateTimeFormatInfo 对象：无法提供该值、参数值为 `null` 或复合格式设置方法没有 IFormatProvider 参数。
   - 对于其他类型的对象，如果调用带 IFormatProvider 参数的复合格式设置方法，它的值会直接传递到 IFormattable.ToString 实现。否则，`null` 传递到 IFormattable.ToString 实现。
4. 调用类型的无参数的 `ToString` 方法（该方法将重写 Object.ToString() 或继承其基类的行为）。在这种情况下，如果格式项中存在 formatString 组件指定的格式字符串，则将忽略该字符串。

前面的步骤执行完毕之后应用对齐。

代码示例

下面的示例显示使用复合格式设置创建的一个字符串和使用对象的 `ToString` 方法创建的另一个字符串。两种格式设置类型产生相同的结果。

```
string FormatString1 = String.Format("{0:dddd MMMM}", DateTime.Now);
string FormatString2 = DateTime.Now.ToString("dddd MMMM");
```

```
Dim FormatString1 As String = String.Format("{0:dddd MMMM}", DateTime.Now)
Dim FormatString2 As String = DateTime.Now.ToString("dddd MMMM")
```

假定当前日期是五月的星期四，那么在美国英语区域性中上述示例中的两个字符串的值都是 `Thursday May` 英语区域性。

Console.WriteLine 提供与 String.Format 相同的功能。这两种方法的唯一差异是 String.Format 将其结果作为字符串返回，而 Console.WriteLine 将结果写入与 Console 对象关联的输出流中。下面的示例使用 Console.WriteLine 方法将 `MyInt` 的值的格式设置为货币值。

```
int MyInt = 100;
Console.WriteLine("{0:C}", MyInt);
// The example displays the following output
// if en-US is the current culture:
//        $100.00
```

```
Dim MyInt As Integer = 100
Console.WriteLine("{0:C}", MyInt)
' The example displays the following output
' if en-US is the current culture:
'        $100.00
```

下面的示例说明为多个对象设置格式，包括用两种不同的方式为一个对象设置格式。

```
string myName = "Fred";
Console.WriteLine(String.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}",
      myName, DateTime.Now));
// Depending on the current time, the example displays output like the following:
//    Name = Fred, hours = 11, minutes = 30
```

```
Dim myName As String = "Fred"
Console.WriteLine(String.Format("Name = {0}, hours = {1:hh}, minutes = {1:mm}", _
                  myName, DateTime.Now))
' Depending on the current time, the example displays output like the following:
'    Name = Fred, hours = 11, minutes = 30
```

下面的示例演示了对齐在格式设置中的使用方式。设置了格式的参数放置在竖线字符 (|) 之间以突出显示得到的对齐。

```
string myFName = "Fred";
string myLName = "Opals";
int myInt = 100;
string FormatFName = String.Format("First Name = |{0,10}|", myFName);
string FormatLName = String.Format("Last Name = |{0,10}|", myLName);
string FormatPrice = String.Format("Price = |{0,10:C}|", myInt);
Console.WriteLine(FormatFName);
Console.WriteLine(FormatLName);
Console.WriteLine(FormatPrice);
Console.WriteLine();

FormatFName = String.Format("First Name = |{0,-10}|", myFName);
FormatLName = String.Format("Last Name = |{0,-10}|", myLName);
FormatPrice = String.Format("Price = |{0,-10:C}|", myInt);
Console.WriteLine(FormatFName);
Console.WriteLine(FormatLName);
Console.WriteLine(FormatPrice);
// The example displays the following output on a system whose current
// culture is en-US:
//          First Name = |      Fred|
//          Last Name = |     Opals|
//          Price = |   $100.00|
//
//          First Name = |Fred      |
//          Last Name = |Opals     |
//          Price = |$100.00   |
```

```
Dim myFName As String = "Fred"
Dim myLName As String = "Opals"

Dim myInt As Integer = 100
Dim FormatFName As String = String.Format("First Name = |{0,10}|", myFName)
Dim FormatLName As String = String.Format("Last Name = |{0,10}|", myLName)
Dim FormatPrice As String = String.Format("Price = |{0,10:C}|", myInt)
Console.WriteLine(FormatFName)
Console.WriteLine(FormatLName)
Console.WriteLine(FormatPrice)
Console.WriteLine()

FormatFName = String.Format("First Name = |{0,-10}|", myFName)
FormatLName = String.Format("Last Name = |{0,-10}|", myLName)
FormatPrice = String.Format("Price = |{0,-10:C}|", myInt)
Console.WriteLine(FormatFName)
Console.WriteLine(FormatLName)
Console.WriteLine(FormatPrice)
' The example displays the following output on a system whose current
' culture is en-US:
'          First Name = |      Fred|
'          Last Name = |     Opals|
'          Price = |   $100.00|
'
'          First Name = |Fred      |
'          Last Name = |Opals     |
'          Price = |$100.00   |
```

请参阅

- WriteLine
- String.Format
- 字符串内插 (C#)
- 字符串内插 (Visual Basic)
- 格式设置类型
- 标准数字格式字符串
- 自定义数字格式字符串
- 标准日期和时间格式字符串
- 自定义日期和时间格式字符串
- 标准 TimeSpan 格式字符串
- 自定义 TimeSpan 格式字符串
- 枚举格式字符串

如何：用前导零填充数字

2021/11/16 •

通过结合使用"D"标准数字格式字符串和精度说明符，将前导零添加到整数。你可以通过使用自定义数字格式字符串，将前导零添加到整数和浮点数。本文介绍如何通过这两种方法用前导零填充数字。

使用前导零将整数填充到特定的长度

1. 确定整数值要显示的最小位数。在此数字中包括任何前导位。
2. 确定是要将整数显示为十进制值还是十六进制值。
   - 若要将整数显示为十进制值，则调用其 `ToString(String)` 方法，并传递字符串"Dn"作为 `format` 参数的值，其中 n 表示字符串的最小长度 。
   - 若要将整数显示为十六进制值，则调用其 `ToString(String)` 方法，并传递字符串"Xn"作为 format 参数的值，其中 n 表示字符串的最小长度。

此外，你也可以在 C# 和 Visual Basic 的内插字符串中使用格式字符串，或者可以调用一个 String.Format 或 Console.WriteLine 等使用复合格式设置的方法。

以下示例使用前导零设置若干整数值的格式，以便格式化数字的总长度至少为八个字符。

```
byte byteValue = 254;
short shortValue = 10342;
int intValue = 1023983;
long lngValue = 6985321;
ulong ulngValue = UInt64.MaxValue;

// Display integer values by calling the ToString method.
Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"), byteValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"), shortValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"), intValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"), lngValue.ToString("X8"));
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"), ulngValue.ToString("X8"));
Console.WriteLine();

// Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue);
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue);
// The example displays the following output:
//                     00000254               000000FE
//                     00010342               00002866
//                     01023983               000F9FEF
//                     06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//
//                     00000254               000000FE
//                     00010342               00002866
//                     01023983               000F9FEF
//                     06985321               006A9669
//         18446744073709551615       FFFFFFFFFFFFFFFF
//         18446744073709551615       FFFFFFFFFFFFFFFF
```

```
Dim byteValue As Byte = 254
Dim shortValue As Short = 10342
Dim intValue As Integer = 1023983
Dim lngValue As Long = 6985321
Dim ulngValue As ULong = UInt64.MaxValue

' Display integer values by calling the ToString method.
Console.WriteLine("{0,22} {1,22}", byteValue.ToString("D8"), byteValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", shortValue.ToString("D8"), shortValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", intValue.ToString("D8"), intValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", lngValue.ToString("D8"), lngValue.ToString("X8"))
Console.WriteLine("{0,22} {1,22}", ulngValue.ToString("D8"), ulngValue.ToString("X8"))
Console.WriteLine()

' Display the same integer values by using composite formatting.
Console.WriteLine("{0,22:D8} {0,22:X8}", byteValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", shortValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", intValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", lngValue)
Console.WriteLine("{0,22:D8} {0,22:X8}", ulngValue)
' The example displays the following output:
'                     00000254               000000FE
'                     00010342               00002866
'                     01023983               000F9FEF
'                     06985321               006A9669
'         18446744073709551615       FFFFFFFFFFFFFFFF
'
'                     00000254               000000FE
'                     00010342               00002866
'                     01023983               000F9FEF
'                     06985321               006A9669
'         18446744073709551615       FFFFFFFFFFFFFFFF
```

使用特定数目的前导零填充整数

1. 确定希望整数值显示多少个前导零。
2. 确定是要将整数显示为十进制值还是十六进制值。
   - 将其格式设置为十进制值需要使用“D”标准格式说明符。
   - 将其格式设置为十六进制值需要使用“X”标准格式说明符。
3. 通过调用整数值的 `ToString("D").Length` 或 `ToString("X").Length` 方法，确定未填充的数字字符串的长度。
4. 将你要在格式化字符串中包括的前导零的数目添加到未填充的数字字符串的长度上。添加前导零的个数将定义填充字符串的总长度。
5. 调用整数值的 `ToString(String)` 方法，并且对于十进制字符串传递字符串“Dn”，对于十六进制字符串传递“Xn”；其中，n 表示填充的字符串的总长度。也可以在支持复合格式设置的方法中使用“Dn”或“Xn”格式字符串。

下面的示例使用五个前导零来填充整数值。

```
int value = 160934;
int decimalLength = value.ToString("D").Length + 5;
int hexLength = value.ToString("X").Length + 5;
Console.WriteLine(value.ToString("D" + decimalLength.ToString()));
Console.WriteLine(value.ToString("X" + hexLength.ToString()));
// The example displays the following output:
//       00000160934
//       00000274A6
```

```
Dim value As Integer = 160934
Dim decimalLength As Integer = value.ToString("D").Length + 5
Dim hexLength As Integer = value.ToString("X").Length + 5
Console.WriteLine(value.ToString("D" + decimalLength.ToString()))
Console.WriteLine(value.ToString("X" + hexLength.ToString()))
' The example displays the following output:
'       00000160934
'       00000274A6
```

使用前导零将数值填充到特定的长度

1. 确定数字的字符串表示形式要在小数点的左侧保留的位数。在此总位数中包括任何前导零。
2. 定义一个使用零占位符“0”来表示零的最小数目的自定义数字格式字符串。
3. 调用数字的 `ToString(String)` 方法并向其传递自定义格式字符串。此外，你也可以将自定义格式字符串与字符串内插或与支持复合格式设置的方法一起使用。

下面的示例使用前导零设置几个数值的格式。因此，格式化后的数字总长度是小数点左侧至少八位数字。

```
string fmt = "00000000.##";
int intValue = 1053240;
decimal decValue = 103932.52m;
float sngValue = 1549230.10873992f;
double dblValue = 9034521202.93217412;

// Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt));
Console.WriteLine(decValue.ToString(fmt));
Console.WriteLine(sngValue.ToString(fmt));
Console.WriteLine(dblValue.ToString(fmt));
Console.WriteLine();

// Display the numbers using composite formatting.
string formatString = " {0,15:" + fmt + "}";
Console.WriteLine(formatString, intValue);
Console.WriteLine(formatString, decValue);
Console.WriteLine(formatString, sngValue);
Console.WriteLine(formatString, dblValue);
// The example displays the following output:
//       01053240
//       00103932.52
//       01549230
//       9034521202.93
//
//               01053240
//            00103932.52
//               01549230
//          9034521202.93
```

```
Dim fmt As String = "00000000.##"
Dim intValue As Integer = 1053240
Dim decValue As Decimal = 103932.52d
Dim sngValue As Single = 1549230.10873992
Dim dblValue As Double = 9034521202.93217412

' Display the numbers using the ToString method.
Console.WriteLine(intValue.ToString(fmt))
Console.WriteLine(decValue.ToString(fmt))
Console.WriteLine(sngValue.ToString(fmt))
Console.WriteLine(dblValue.ToString(fmt))
Console.WriteLine()

' Display the numbers using composite formatting.
Dim formatString As String = " {0,15:" + fmt + "}"
Console.WriteLine(formatString, intValue)
Console.WriteLine(formatString, decValue)
Console.WriteLine(formatString, sngValue)
Console.WriteLine(formatString, dblValue)
' The example displays the following output:
'       01053240
'       00103932.52
'       01549230
'       9034521202.93
'
'               01053240
'            00103932.52
'               01549230
'          9034521202.93
```

使用特定数目的前导零填充数值

1. 确定希望数值具有多少个前导零。
2. 确定未填充数字字符串中小数点左侧的位数：
   a. 确定数字的字符串表示形式是否包括小数点符号。
   b. 如果它包括小数点符号，则确定小数点左侧的字符数。

      - 或 -

      如果它不包括小数点符号，则确定字符串的长度。
3. 创建使用以下格式的自定义格式字符串：
   - 零占位符“0”表示在字符串中出现的每个前导零。
   - 零占位符或数字占位符“#”表示默认字符串中的每个数字。
4. 将自定义格式字符串作为对数字的 `ToString(String)` 方法或支持复合格式设置的方法的参数提供。

下面的示例使用五个前导零来填充两个 Double 值。

```
double[] dblValues = { 9034521202.93217412, 9034521202 };
foreach (double dblValue in dblValues)
{
   string decSeparator = System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator;
   string fmt, formatString;

   if (dblValue.ToString().Contains(decSeparator))
   {
      int digits = dblValue.ToString().IndexOf(decSeparator);
      fmt = new String('0', 5) + new String('#', digits) + ".##";
   }
   else
   {
      fmt = new String('0', dblValue.ToString().Length);
   }
   formatString = "{0,20:" + fmt + "}";

   Console.WriteLine(dblValue.ToString(fmt));
   Console.WriteLine(formatString, dblValue);
}
// The example displays the following output:
//       000009034521202.93
//         000009034521202.93
//       9034521202
//                 9034521202
```

```
Dim dblValues() As Double = {9034521202.93217412, 9034521202}
For Each dblValue As Double In dblValues
    Dim decSeparator As String = System.Globalization.NumberFormatInfo.CurrentInfo.NumberDecimalSeparator
    Dim fmt, formatString As String

    If dblValue.ToString.Contains(decSeparator) Then
        Dim digits As Integer = dblValue.ToString().IndexOf(decSeparator)
        fmt = New String("0"c, 5) + New String("#"c, digits) + ".##"
    Else
        fmt = New String("0"c, dblValue.ToString.Length)
    End If
    formatString = "{0,20:" + fmt + "}"

    Console.WriteLine(dblValue.ToString(fmt))
    Console.WriteLine(formatString, dblValue)
Next
' The example displays the following output:
'       000009034521202.93
'         000009034521202.93
'       9034521202
'                 9034521202
```

请参阅

- 自定义数字格式字符串
- 标准数字格式字符串
- 复合格式设置

如何：从特定日期中提取星期几

2021/11/16 •

利用 .NET，可以很容易地确定某个特定日期是星期几，以及显示某个特定日期的本地化星期几名称。指示与特定日期相对应的星期几的枚举值可以从 DayOfWeek 或 DayOfWeek 属性中获取。与此不同的是，检索星期几名称是一项格式化操作，可通过调用格式化方法来执行，例如日期和时间值的 `ToString` 方法或 String.Format 方法。本主题演示如何执行这些格式化操作。

提取指示星期几的数字

1. 如果要使用日期的字符串表示形式，请使用静态 DateTime 或 DateTimeOffset 方法将其转换为 DateTime.Parse 或 DateTimeOffset.Parse 值。
2. 使用 DateTime.DayOfWeek 或 DateTimeOffset.DayOfWeek 属性检索指示星期几的 DayOfWeek 值。
3. 如有必要，可将 DayOfWeek 值强制转换（在 C# 中）或转换（在 Visual Basic 中）为整数。

下面的示例将显示一个整数，用于表示特定日期的星期几。

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine((int) dateValue.DayOfWeek);
   }
}
// The example displays the following output:
//       3
```

```
Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#
        Console.WriteLine(dateValue.DayOfWeek)
    End Sub
End Module
' The example displays the following output:
'    3
```

提取缩写的工作日名称

1. 如果要使用日期的字符串表示形式，请使用静态 DateTime 或 DateTimeOffset 方法将其转换为 DateTime.Parse 或 DateTimeOffset.Parse 值。
2. 你可以提取当前区域性或特定区域性的缩写的星期几名称：
   a. 若要提取当前区域性的缩写的星期几名称，请调用日期和时间值的 DateTime.ToString(String) 或 DateTimeOffset.ToString(String) 实例方法，并以 `format` 参数的形式传递字符串“ddd”。下面的示例演示 ToString(String) 方法的调用。

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd"));
   }
}
// The example displays the following output:
 //       Wed
```

```
Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#
        Console.WriteLine(dateValue.ToString("ddd"))
    End Sub
End Module
' The example displays the following output:
'       Wed
```

b. 若要提取特定区域性的缩写的星期几名称，请调用日期和时间值的 DateTime.ToString(String, IFormatProvider) 或 DateTimeOffset.ToString(String, IFormatProvider) 实例方法。同时以 `format` 参数形式传递字符串"ddd"。以 CultureInfo 参数的形式传递表示要检索其星期几名称的区域性的 DateTimeFormatInfo 或 `provider` 对象。下面的代码阐释如何使用表示 fr-FR 区域性的 ToString(String, IFormatProvider) 对象调用 CultureInfo 方法。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("ddd",
                        new CultureInfo("fr-FR")));
   }
}
// The example displays the following output:
//       mer.
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#
        Console.WriteLine(dateValue.ToString("ddd",
                          New CultureInfo("fr-FR")))
    End Sub
End Module
' The example displays the following output:
'       mer.
```

提取完整的工作日名称

1. 如果要使用日期的字符串表示形式，请使用静态 DateTime 或 DateTimeOffset 方法将其转换为

DateTime.Parse 或 DateTimeOffset.Parse 值。

2. 你可以提取当前区域性或特定区域性的完整的星期几名称：

   a. 若要提取当前区域性的缩写的星期几名称，请调用日期和时间值的 DateTime.ToString(String) 或 DateTimeOffset.ToString(String) 实例方法，并以 `format` 参数的形式传递字符串“dddd”。下面的示例演示 ToString(String) 方法的调用。

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd"));
   }
}
// The example displays the following output:
//       Wednesday
```

```
Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#
        Console.WriteLine(dateValue.ToString("dddd"))
    End Sub
End Module
' The example displays the following output:
'       Wednesday
```

   b. 若要提取特定区域性的星期几名称，请调用日期和时间值的 DateTime.ToString(String, IFormatProvider) 或 DateTimeOffset.ToString(String, IFormatProvider) 实例方法。同时以 `format` 参数形式传递字符串“dddd”。以 CultureInfo 参数的形式传递表示要检索其星期几名称的区域性的 DateTimeFormatInfo 或 `provider` 对象。下面的代码阐释如何使用表示 es-ES 区域性的 ToString(String, IFormatProvider) 对象调用 CultureInfo 方法。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      DateTime dateValue = new DateTime(2008, 6, 11);
      Console.WriteLine(dateValue.ToString("dddd",
                        new CultureInfo("es-ES")));
   }
}
// The example displays the following output:
//       miércoles.
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#
        Console.WriteLine(dateValue.ToString("dddd", _
                          New CultureInfo("es-ES")))
    End Sub
End Module
' The example displays the following output:
'       miércoles.
```

示例

该示例演示如何调用 DateTime.DayOfWeek 和 DateTimeOffset.DayOfWeek 属性以及 DateTime.ToString 和 DateTimeOffset.ToString 方法，以检索特定日期中表示星期几的数字、缩写的星期几名称和完整的星期几名称。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string dateString = "6/11/2007";
      DateTime dateValue;
      DateTimeOffset dateOffsetValue;

      try
      {
         DateTimeFormatInfo dateTimeFormats;
         // Convert date representation to a date value
         dateValue = DateTime.Parse(dateString, CultureInfo.InvariantCulture);
         dateOffsetValue = new DateTimeOffset(dateValue,
                                    TimeZoneInfo.Local.GetUtcOffset(dateValue));

         // Convert date representation to a number indicating the day of week
         Console.WriteLine((int) dateValue.DayOfWeek);
         Console.WriteLine((int) dateOffsetValue.DayOfWeek);

         // Display abbreviated weekday name using current culture
         Console.WriteLine(dateValue.ToString("ddd"));
         Console.WriteLine(dateOffsetValue.ToString("ddd"));

         // Display full weekday name using current culture
         Console.WriteLine(dateValue.ToString("dddd"));
         Console.WriteLine(dateOffsetValue.ToString("dddd"));

         // Display abbreviated weekday name for de-DE culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("de-DE")));
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                    new CultureInfo("de-DE")));

         // Display abbreviated weekday name with de-DE DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("de-DE").DateTimeFormat;
         Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats));
         Console.WriteLine(dateOffsetValue.ToString("ddd", dateTimeFormats));

         // Display full weekday name for fr-FR culture
         Console.WriteLine(dateValue.ToString("ddd", new CultureInfo("fr-FR")));
         Console.WriteLine(dateOffsetValue.ToString("ddd",
                                                  new CultureInfo("fr-FR")));

         // Display abbreviated weekday name with fr-FR DateTimeFormatInfo object
         dateTimeFormats = new CultureInfo("fr-FR").DateTimeFormat;
```

```
            dateTimeFormats = new CultureInfo("fr-FR").DateTimeFormat;
            Console.WriteLine(dateValue.ToString("dddd", dateTimeFormats));
            Console.WriteLine(dateOffsetValue.ToString("dddd", dateTimeFormats));
         }
         catch (FormatException)
         {
            Console.WriteLine("Unable to convert {0} to a date.", dateString);
         }
      }
   }
// The example displays the following output:
//       1
//       1
//       Mon
//       Mon
//       Monday
//       Monday
//       Mo
//       Mo
//       Mo
//       Mo
//       lun.
//       lun.
//       lundi
//       lundi
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim dateString As String = "6/11/2007"
        Dim dateValue As Date
        Dim dateOffsetValue As DateTimeOffset

        Try
            Dim dateTimeFormats As DateTimeFormatInfo
            ' Convert date representation to a date value
            dateValue = Date.Parse(dateString, CultureInfo.InvariantCulture)
            dateOffsetValue = New DateTimeOffset(dateValue, _
                                      TimeZoneInfo.Local.GetUtcOffset(dateValue))
            ' Convert date representation to a number indicating the day of week
            Console.WriteLine(dateValue.DayOfWeek)
            Console.WriteLine(dateOffsetValue.DayOfWeek)

            ' Display abbreviated weekday name using current culture
            Console.WriteLine(dateValue.ToString("ddd"))
            Console.WriteLine(dateOffsetValue.ToString("ddd"))

            ' Display full weekday name using current culture
            Console.WriteLine(dateValue.ToString("dddd"))
            Console.WriteLine(dateOffsetValue.ToString("dddd"))

            ' Display abbreviated weekday name for de-DE culture
            Console.WriteLine(dateValue.ToString("ddd", New CultureInfo("de-DE")))
            Console.WriteLine(dateOffsetValue.ToString("ddd", _
                                                       New CultureInfo("de-DE")))

            ' Display abbreviated weekday name with de-DE DateTimeFormatInfo object
            dateTimeFormats = New CultureInfo("de-DE").DateTimeFormat
            Console.WriteLine(dateValue.ToString("ddd", dateTimeFormats))
            Console.WriteLine(dateOffsetValue.ToString("ddd", dateTimeFormats))

            ' Display full weekday name for fr-FR culture
            Console.WriteLine(dateValue.ToString("ddd", New CultureInfo("fr-FR")))
            Console.WriteLine(dateOffsetValue.ToString("ddd", _
                                                       New CultureInfo("fr-FR")))

            ' Display abbreviated weekday name with fr-FR DateTimeFormatInfo object
            dateTimeFormats = New CultureInfo("fr-FR").DateTimeFormat
            Console.WriteLine(dateValue.ToString("dddd", dateTimeFormats))
            Console.WriteLine(dateOffsetValue.ToString("dddd", dateTimeFormats))
        Catch e As FormatException
            Console.WriteLine("Unable to convert {0} to a date.", dateString)
        End Try
    End Sub
End Module
' The example displays the following output to the console:
'       1
'       1
'       Mon
'       Mon
'       Monday
'       Monday
'       Mo
'       Mo
'       Mo
'       Mo
'       lun.
'       lun.
'       lundi
'       lundi
```

个别语言可能提供与 .NET 所提供的功能相同或互为补充的功能。例如，Visual Basic 包括这样的两个函数：

- `Weekday`，它返回指示特定日期中表示星期几的数字。此函数将一周中第一天的序数值视为一，而 DateTime.DayOfWeek 属性却将其视为零。
- `WeekdayName`，它返回当前区域性中与特定星期几相对应的周的名称。

下面的示例演示了 Visual Basic `Weekday` 和 `WeekdayName` 函数的用法。

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim dateValue As Date = #6/11/2008#

        ' Get weekday number using Visual Basic Weekday function
        Console.WriteLine(Weekday(dateValue))                ' Displays 4
        ' Compare with .NET DateTime.DayOfWeek property
        Console.WriteLine(dateValue.DayOfWeek)               ' Displays 3

        ' Get weekday name using Weekday and WeekdayName functions
        Console.WriteLine(WeekdayName(Weekday(dateValue)))   ' Displays Wednesday

        ' Change culture to de-DE
        Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
        Thread.CurrentThread.CurrentCulture = New CultureInfo("de-DE")
        ' Get weekday name using Weekday and WeekdayName functions
        Console.WriteLine(WeekdayName(Weekday(dateValue)))   ' Displays Donnerstag

        ' Restore original culture
        Thread.CurrentThread.CurrentCulture = originalCulture
    End Sub
End Module
```

也可以使用 DateTime.DayOfWeek 属性返回的值检索特定日期的星期几名称。此过程只需对 ToString 属性返回的值调用 DayOfWeek 方法。但是，此技术并不生成当前区域性的本地化星期几名称，如下面的示例所示。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Change current culture to fr-FR
      CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

      DateTime dateValue = new DateTime(2008, 6, 11);
      // Display the DayOfWeek string representation
      Console.WriteLine(dateValue.DayOfWeek.ToString());
      // Restore original current culture
      Thread.CurrentThread.CurrentCulture = originalCulture;
   }
}
// The example displays the following output:
//       Wednesday
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        ' Change current culture to fr-FR
        Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
        Thread.CurrentThread.CurrentCulture = New CultureInfo("fr-FR")

        Dim dateValue As Date = #6/11/2008#
        ' Display the DayOfWeek string representation
        Console.WriteLine(dateValue.DayOfWeek.ToString())
        ' Restore original current culture
        Thread.CurrentThread.CurrentCulture = originalCulture
    End Sub
End Module
' The example displays the following output:
'       Wednesday
```

请参阅

- 标准日期和时间格式字符串
- 自定义日期和时间格式字符串

如何：定义和使用自定义数值格式提供程序

2021/11/16 •

.NET 使你可以全面控制数值的字符串表示形式。它支持用于自定义数值格式的以下功能：

- 标准数字格式字符串，提供一组预定义格式以用于将数字转换为其字符串表示形式。可以将它们与包含 `format` 参数的任何数字格式设置方法（如 Decimal.ToString(String)）结合使用。有关详细信息，请参阅标准数字格式字符串。
- 自定义数字格式字符串，提供一组可以进行组合以定义自定义数字格式说明符的符号。它们还可以与包含 `format` 参数的任何数字格式设置方法（如 Decimal.ToString(String)）结合使用。有关详细信息，请参阅自定义数字格式字符串。
- 自定义 CultureInfo 或 NumberFormatInfo 对象，定义用于显示数值的字符串表示形式的符号和格式模式。可以将它们与包含 `provider` 参数的任何数字格式设置方法（如 ToString）结合使用。`provider` 参数通常用于指定区域性专用格式设置。

在某些情况下（例如当应用程序必须显示格式化帐号、标识号或邮政编码），这三种方法都不合适。借助 .NET，你还可以定义既不是 CultureInfo 也不是 NumberFormatInfo 对象的格式设置对象，用于确定如何设置数值的格式。本主题提供用于实现这类对象的分步说明，并提供对电话号码设置格式的示例。

定义自定义格式提供程序

1. 定义实现 IFormatProvider 和 ICustomFormatter 接口的类。
2. 实现 IFormatProvider.GetFormat 方法。GetFormat 是格式设置方法（如 String.Format(IFormatProvider, String, Object[]) 方法）调用的回调方法，用于检索实际负责执行自定义格式设置的对象。GetFormat 的典型实现执行以下操作：
   a. 确定以方法参数形式传递的 Type 对象是否表示 ICustomFormatter 接口。
   b. 如果此参数确实表示 ICustomFormatter 接口，GetFormat 会返回对象，用于实现负责执行自定义格式设置的 ICustomFormatter 接口。通常，自定义格式设置对象返回其自身。
   c. 如果参数不表示 ICustomFormatter 接口，GetFormat 返回的是 `null`。
3. 实现 Format 方法。此方法由 String.Format(IFormatProvider, String, Object[]) 方法调用，负责返回数字的字符串表示形式。实现方法通常涉及以下步骤：
   a. （可选）通过检查 `provider` 参数，确保此方法旨在以合法方式提供格式设置服务。对于实现 IFormatProvider 和 ICustomFormatter 的格式设置对象，这涉及测试 `provider` 参数是否与当前格式设置对象相等。
   b. 确定格式设置对象是否应支持自定义格式说明符。（例如，格式说明符“N”可能指示应以 NANP 格式输出美国电话号码，而“I”可能指示以 ITU-T 建议 E.123 格式进行输出。）如果使用格式说明符，则方法应处理特定格式说明符。它会在 `format` 参数中传递给方法。如果没有说明符，`format` 参数的值是 String.Empty。
   c. 检索作为 `arg` 参数传递给方法的数值。执行将它转换为其字符串表示形式所需的任何操作。
   d. 返回 `arg` 参数的字符串表示形式。

使用自定义数字格式设置对象

1. 创建自定义格式设置类的新实例。

2. 调用 String.Format(IFormatProvider, String, Object[]) 格式设置方法，同时向它传递自定义格式设置对象、格式设置说明符（或 String.Empty，如果未使用说明符的话），以及要设置格式的数值。

示例

下面的示例定义了一个名为 `TelephoneFormatter` 的自定义数值格式提供程序，该提供程序将代表美国电话号码的数字转化为它的 NANP 或 E.123 格式。该方法处理两个格式说明符“N”（输出 NANP 格式）和“I”（输出国际 E.123 格式）。

```
using System;
using System.Globalization;

public class TelephoneFormatter : IFormatProvider, ICustomFormatter
{
   public object GetFormat(Type formatType)
   {
      if (formatType == typeof(ICustomFormatter))
         return this;
      else
         return null;
   }

   public string Format(string format, object arg, IFormatProvider formatProvider)
   {
      // Check whether this is an appropriate callback
      if (! this.Equals(formatProvider))
         return null;

      // Set default format specifier
      if (string.IsNullOrEmpty(format))
         format = "N";

      string numericString = arg.ToString();

      if (format == "N")
      {
         if (numericString.Length <= 4)
            return numericString;
         else if (numericString.Length == 7)
            return numericString.Substring(0, 3) + "-" + numericString.Substring(3, 4);
         else if (numericString.Length == 10)
               return "(" + numericString.Substring(0, 3) + ") " +
                      numericString.Substring(3, 3) + "-" + numericString.Substring(6);
         else
            throw new FormatException(
                      string.Format("'{0}' cannot be used to format {1}.",
                                    format, arg.ToString()));
      }
      else if (format == "I")
      {
         if (numericString.Length < 10)
            throw new FormatException(string.Format("{0} does not have 10 digits.", arg.ToString()));
         else
            numericString = "+1 " + numericString.Substring(0, 3) + " " + numericString.Substring(3, 3) + "
" + numericString.Substring(6);
      }
      else
      {
         throw new FormatException(string.Format("The {0} format specifier is invalid.", format));
      }
      return numericString;
   }
}
```

```
public class TestTelephoneFormatter
{
   public static void Main()
   {
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 0));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 911));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 8490216));
      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 4257884748));

      Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:I}", 4257884748));
   }
}
```

```
Public Class TelephoneFormatter : Implements IFormatProvider, ICustomFormatter
    Public Function GetFormat(formatType As Type) As Object _
                    Implements IFormatProvider.GetFormat
        If formatType Is GetType(ICustomFormatter) Then
            Return Me
        Else
            Return Nothing
        End If
    End Function

    Public Function Format(fmt As String, arg As Object, _
                           formatProvider As IFormatProvider) As String _
                    Implements ICustomFormatter.Format
        ' Check whether this is an appropriate callback
        If Not Me.Equals(formatProvider) Then Return Nothing

        ' Set default format specifier
        If String.IsNullOrEmpty(fmt) Then fmt = "N"

        Dim numericString As String = arg.ToString

        If fmt = "N" Then
            Select Case numericString.Length
                Case <= 4
                    Return numericString
                Case 7
                    Return Left(numericString, 3) & "-" & Mid(numericString, 4)
                Case 10
                    Return "(" & Left(numericString, 3) & ") " & _
                           Mid(numericString, 4, 3) & "-" & Mid(numericString, 7)
                Case Else
                    Throw New FormatException( _
                              String.Format("'{0}' cannot be used to format {1}.", _
                                            fmt, arg.ToString()))
            End Select
        ElseIf fmt = "I" Then
            If numericString.Length < 10 Then
                Throw New FormatException(String.Format("{0} does not have 10 digits.", arg.ToString()))
            Else
                numericString = "+1 " & Left(numericString, 3) & " " & Mid(numericString, 4, 3) & " " &
Mid(numericString, 7)
            End If
        Else
            Throw New FormatException(String.Format("The {0} format specifier is invalid.", fmt))
        End If
        Return numericString
    End Function
End Class

Public Module TestTelephoneFormatter
    Public Sub Main
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 0))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 911))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 8490216))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 4257884748))

        Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 0))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 911))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 8490216))
        Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 4257884748))

        Console.WriteLine(String.Format(New TelephoneFormatter, "{0:I}", 4257884748))
    End Sub
End Module
```

自定义数字格式提供程序只能与 String.Format(IFormatProvider, String, Object[]) 方法配合使用。包含

IFormatProvider 类型参数的数字格式设置方法(如 `ToString`)的其他重载，都会向 IFormatProvider.GetFormat 实现传递表示 NumberFormatInfo 类型的 Type 对象。此方法应返回 NumberFormatInfo 对象。如果未返回，将会忽略自定义数字格式提供程序，而改用当前区域性的 NumberFormatInfo 对象。在此示例中，`TelephoneFormatter.GetFormat` 方法检查方法参数，并在它表示除 ICustomFormatter 之外的类型时返回 `null`，从而处理它可能会被不恰当地传递给数字格式设置方法的情况。

如果自定义数字格式提供程序支持一组格式说明符，请确保提供在 String.Format(IFormatProvider, String, Object[]) 方法调用中使用的格式项没有格式说明符时的默认行为。在示例中，"N"是默认格式说明符。这使数字可以通过提供显式格式说明符来转换为格式化电话号码。下面的示例演示了此类方法调用。

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0:N}", 4257884748));
```

```
Console.WriteLine(String.Format(New TelephoneFormatter, "{0:N}", 4257884748))
```

但是它还允许在不存在格式说明符时进行转换。下面的示例演示了此类方法调用。

```
Console.WriteLine(String.Format(new TelephoneFormatter(), "{0}", 4257884748));
```

```
Console.WriteLine(String.Format(New TelephoneFormatter, "{0}", 4257884748))
```

如果未定义默认格式说明符，ICustomFormatter.Format 方法实现应包含如下代码，以便 .NET 能够提供代码不支持的格式设置。

```
if (arg is IFormattable)
   s = ((IFormattable)arg).ToString(format, formatProvider);
else if (arg != null)
   s = arg.ToString();
```

```
If TypeOf (arg) Is IFormattable Then
    s = DirectCast(arg, IFormattable).ToString(fmt, formatProvider)
ElseIf arg IsNot Nothing Then
    s = arg.ToString()
End If
```

在此示例中，实现 ICustomFormatter.Format 的方法旨在用作 String.Format(IFormatProvider, String, Object[]) 方法的回调方法。因此，它会检查 `formatProvider` 参数，以确定它是否包含对当前 `TelephoneFormatter` 对象的引用。但是，也可以直接从代码调用该方法。在这种情况下，可以使用 `formatProvider` 参数，提供用于提供区域性专用格式设置信息的 CultureInfo 或 NumberFormatInfo 对象。

如何：往返行程日期和时间值

2021/11/16 •

在许多应用程序中，日期和时间值旨在明确标识单个时间点。本文演示了如何保存和还原 DateTime 值、DateTimeOffset 值以及包含时区信息的日期和时间值，以便还原后的值与保存的值标识的时间相同。

往返 DateTime 值

1. 通过调用包含 "o" 格式说明符的 DateTime.ToString(String) 方法，将 DateTime 值转换为字符串表示形式。
2. 将 DateTime 值的字符串表示形式保存到文件中，或跨进程、应用域或计算机边界传递它。
3. 检索表示 DateTime 值的字符串。
4. 调用 DateTime.Parse(String, IFormatProvider, DateTimeStyles) 方法，并以 `styles` 参数值的形式传递 DateTimeStyles.RoundtripKind。

下面的示例展示了如何往返 DateTime 值。

```
const string fileName = @".\DateFile.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTime dateToSave = DateTime.SpecifyKind(new DateTime(2008, 6, 12, 18, 45, 15),
                                           DateTimeKind.Local);
string dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} ({1}) to {2}.",
                  dateToSave.ToString(),
                  dateToSave.Kind.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTime restoredDate;

StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();
inFile.Close();
restoredDate = DateTime.Parse(dateString, null, DateTimeStyles.RoundtripKind);
Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(),
                                              fileName,
                                              restoredDate.Kind.ToString());
// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-05:00.
//    Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
//    Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.
```

```
Const fileName As String = ".\DateFile.txt"

Dim outFile As New StreamWriter(fileName)

' Save DateTime value.
Dim dateToSave As Date = DateTime.SpecifyKind(#06/12/2008 6:45:15 PM#, _
                                              DateTimeKind.Local)
Dim dateString As String = dateToSave.ToString("o")
Console.WriteLine("Converted {0} ({1}) to {2}.", dateToSave.ToString(), _
                  dateToSave.Kind.ToString(), dateString)
outFile.WriteLine(dateString)
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName)
outFile.Close()

' Restore DateTime value.
Dim restoredDate As Date

Dim inFile As New StreamReader(fileName)
dateString = inFile.ReadLine()
inFile.Close()
restoredDate = DateTime.Parse(dateString, Nothing, DateTimeStyles.RoundTripKind)
Console.WriteLine("Read {0} ({2}) from {1}.", restoredDate.ToString(), _
                  fileName, restoredDAte.Kind.ToString())
' The example displays the following output:
'    Converted 6/12/2008 6:45:15 PM (Local) to 2008-06-12T18:45:15.0000000-05:00.
'    Wrote 2008-06-12T18:45:15.0000000-05:00 to .\DateFile.txt.
'    Read 6/12/2008 6:45:15 PM (Local) from .\DateFile.txt.
```

往返 DateTime 值时，此方法成功暂留所有本地时间和世界时间。例如，如果本地 DateTime 值保存在采用美国太平洋标准时区的系统会，并在位于美国中部标准时区的系统上还原，则还原的日期和时间会比原始时间晚两个小时，这反映了两个时区之间的时差。但是，此方法对于未指定时间不一定准确。所有 Kind 属性为 Unspecified 的 DateTime 值都会被视为本地时间。如果不是本地时间，则 DateTime 不会成功标识正确的时间点。针对此限制的解决方法是将日期和时间值与其时区紧密耦合，以便进行保存和还原操作。

往返 DateTimeOffset 值

1. 通过调用包含 "o" 格式说明符的 DateTimeOffset.ToString(String) 方法，将 DateTimeOffset 值转换为字符串表示形式。
2. 将 DateTimeOffset 值的字符串表示形式保存到文件中，或跨进程、应用域或计算机边界传递它。
3. 检索表示 DateTimeOffset 值的字符串。
4. 调用 DateTimeOffset.Parse(String, IFormatProvider, DateTimeStyles) 方法，并以 `styles` 参数值的形式传递 DateTimeStyles.RoundtripKind。

下面的示例展示了如何往返 DateTimeOffset 值。

```
const string fileName = @".\DateOff.txt";

StreamWriter outFile = new StreamWriter(fileName);

// Save DateTime value.
DateTimeOffset dateToSave = new DateTimeOffset(2008, 6, 12, 18, 45, 15,
                                               new TimeSpan(7, 0, 0));
string dateString = dateToSave.ToString("o");
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(),
                  dateString);
outFile.WriteLine(dateString);
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName);
outFile.Close();

// Restore DateTime value.
DateTimeOffset restoredDateOff;

StreamReader inFile = new StreamReader(fileName);
dateString = inFile.ReadLine();
inFile.Close();
restoredDateOff = DateTimeOffset.Parse(dateString, null,
                                       DateTimeStyles.RoundtripKind);
Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(),
                  fileName);
// The example displays the following output:
//    Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-12T18:45:15.0000000+07:00.
//    Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
//    Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

```
Const fileName As String = ".\DateOff.txt"

Dim outFile As New StreamWriter(fileName)

' Save DateTime value.
Dim dateToSave As New DateTimeOffset(2008, 6, 12, 18, 45, 15, _
                                     New TimeSpan(7, 0, 0))
Dim dateString As String = dateToSave.ToString("o")
Console.WriteLine("Converted {0} to {1}.", dateToSave.ToString(), dateString)
outFile.WriteLine(dateString)
Console.WriteLine("Wrote {0} to {1}.", dateString, fileName)
outFile.Close()

' Restore DateTime value.
Dim restoredDateOff As DateTimeOffset

Dim inFile As New StreamReader(fileName)
dateString = inFile.ReadLine()
inFile.Close()
restoredDateOff = DateTimeOffset.Parse(dateString, Nothing, DateTimeStyles.RoundTripKind)
Console.WriteLine("Read {0} from {1}.", restoredDateOff.ToString(), fileName)
' The example displays the following output:
'    Converted 6/12/2008 6:45:15 PM +07:00 to 2008-06-12T18:45:15.0000000+07:00.
'    Wrote 2008-06-12T18:45:15.0000000+07:00 to .\DateOff.txt.
'    Read 6/12/2008 6:45:15 PM +07:00 from .\DateOff.txt.
```

此方法始终将 DateTimeOffset 值明确标识为单个时间点。然后，可以调用 DateTimeOffset.ToUniversalTime 方法，将此值转换为协调世界时 (UTC)；也可以调用 DateTimeOffset.ToOffset 或 TimeZoneInfo.ConvertTime(DateTimeOffset, TimeZoneInfo) 方法，将此值转换为特定时区时间。此方法的主要限制在于，如果对表示特定时区时间的 DateTimeOffset 值执行日期和时间算术，生成的结果对于相应时区来说可能并不准确。这是因为 DateTimeOffset 值在实例化时就与时区解除关联。因此，执行日期和时间计算时，无法再应用该时区的调整规则。可以通过定义包含日期和时间值以及其随附时区的自定义类型，来解决此问题。

往返包含时区信息的日期和时间值的具体步骤

1. 定义包含两个字段的类或结构。第一个字段是 DateTime 或 DateTimeOffset 对象，第二个字段是 TimeZoneInfo 对象。下面的示例展示了这种类型的简单版本。

```
[Serializable] public class DateInTimeZone
{
   private TimeZoneInfo tz;
   private DateTimeOffset thisDate;

   public DateInTimeZone() {}

   public DateInTimeZone(DateTimeOffset date, TimeZoneInfo timeZone)
   {
      if (timeZone == null)
         throw new ArgumentNullException("The time zone cannot be null.");

      this.thisDate = date;
      this.tz = timeZone;
   }

   public DateTimeOffset DateAndTime
   {
      get {
         return this.thisDate;
      }
      set {
         if (value.Offset != this.tz.GetUtcOffset(value))
            this.thisDate = TimeZoneInfo.ConvertTime(value, tz);
         else
            this.thisDate = value;
      }
   }

   public TimeZoneInfo TimeZone
   {
      get {
         return this.tz;
      }
   }
}
```

```
<Serializable> Public Class DateInTimeZone
    Private tz As TimeZoneInfo
    Private thisDate As DateTimeOffset

    Public Sub New()
    End Sub

    Public Sub New(date1 As DateTimeOffset, timeZone As TimeZoneInfo)
        If timeZone Is Nothing Then
            Throw New ArgumentNullException("The time zone cannot be null.")
        End If
        Me.thisDate = date1
        Me.tz = timeZone
    End Sub

    Public Property DateAndTime As DateTimeOffset
        Get
            Return Me.thisDate
        End Get
        Set
            If Value.Offset <> Me.tz.GetUtcOffset(Value) Then
                Me.thisDate = TimeZoneInfo.ConvertTime(Value, tz)
            Else
                Me.thisDate = Value
            End If
        End Set
    End Property

    Public ReadOnly Property TimeZone As TimeZoneInfo
        Get
            Return tz
        End Get
    End Property
End Class
```

2. 使用 SerializableAttribute 属性标记类。

3. 使用 BinaryFormatter.Serialize 方法串行化对象。

4. 使用 Deserialize 方法还原对象。

5. 将反串行化的对象强制转换(在 C# 中)或转换(在 Visual Basic 中)为相应类型的对象。

下面的示例展示了如何往返同时存储时区以及日期和时间信息的对象。

```
const string fileName = @".\DateWithTz.dat";

DateTime tempDate = new DateTime(2008, 9, 3, 19, 0, 0);
TimeZoneInfo tempTz = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time");
DateInTimeZone dateWithTz = new DateInTimeZone(new DateTimeOffset(tempDate,
                                tempTz.GetUtcOffset(tempDate)),
                                tempTz);

// Store DateInTimeZone value to a file
FileStream outFile = new FileStream(fileName, FileMode.Create);
try
{
   BinaryFormatter formatter = new BinaryFormatter();
   formatter.Serialize(outFile, dateWithTz);
   Console.WriteLine("Saving {0} {1} to {2}", dateWithTz.DateAndTime,
                     dateWithTz.TimeZone.IsDaylightSavingTime(dateWithTz.DateAndTime) ?
                     dateWithTz.TimeZone.DaylightName : dateWithTz.TimeZone.DaylightName,
                     fileName);
}
catch (SerializationException)
{
   Console.WriteLine("Unable to serialize time data to {0}.", fileName);
}
finally
{
   outFile.Close();
}

// Retrieve DateInTimeZone value
if (File.Exists(fileName))
{
   FileStream inFile = new FileStream(fileName, FileMode.Open);
   DateInTimeZone dateWithTz2 = new DateInTimeZone();
   try
   {
      BinaryFormatter formatter = new BinaryFormatter();
      dateWithTz2 = formatter.Deserialize(inFile) as DateInTimeZone;
      Console.WriteLine("Restored {0} {1} from {2}", dateWithTz2.DateAndTime,
                        dateWithTz2.TimeZone.IsDaylightSavingTime(dateWithTz2.DateAndTime) ?
                        dateWithTz2.TimeZone.DaylightName : dateWithTz2.TimeZone.DaylightName,
                        fileName);
   }
   catch (SerializationException)
   {
      Console.WriteLine("Unable to retrieve date and time information from {0}",
                        fileName);
   }
   finally
   {
      inFile.Close();
   }
}
// This example displays the following output to the console:
//    Saving 9/3/2008 7:00:00 PM -05:00 Central Daylight Time to .\DateWithTz.dat
//    Restored 9/3/2008 7:00:00 PM -05:00 Central Daylight Time from .\DateWithTz.dat
```

```
Const fileName As String = ".\DateWithTz.dat"

Dim tempDate As Date = #9/3/2008 7:00:00 PM#
Dim tempTz As TimeZoneInfo = TimeZoneInfo.FindSystemTimeZoneById("Central Standard Time")
Dim dateWithTz As New DateInTimeZone(New DateTimeOffset(tempDate, _
                                     tempTz.GetUtcOffset(tempDate)), _
                                  tempTz)

' Store DateInTimeZone value to a file
Dim outFile As New FileStream(fileName, FileMode.Create)
Try
    Dim formatter As New BinaryFormatter()
    formatter.Serialize(outFile, dateWithTz)
    Console.WriteLine("Saving {0} {1} to {2}", dateWithTz.DateAndTime, _
            IIf(dateWithTz.TimeZone.IsDaylightSavingTime(dateWithTz.DateAndTime), _
                dateWithTz.TimeZone.DaylightName, dateWithTz.TimeZone.DaylightName), _
            fileName)
Catch e As SerializationException
    Console.WriteLine("Unable to serialize time data to {0}.", fileName)
Finally
    outFile.Close()
End Try

' Retrieve DateInTimeZone value
If File.Exists(fileName) Then
    Dim inFile As New FileStream(fileName, FileMode.Open)
    Dim dateWithTz2 As New DateInTimeZone()
    Try
        Dim formatter As New BinaryFormatter()
        dateWithTz2 = DirectCast(formatter.Deserialize(inFile), DateInTimeZone)
        Console.WriteLine("Restored {0} {1} from {2}", dateWithTz2.DateAndTime, _
                          IIf(dateWithTz2.TimeZone.IsDaylightSavingTime(dateWithTz2.DateAndTime), _
                          dateWithTz2.TimeZone.DaylightName, dateWithTz2.TimeZone.DaylightName), _
                          fileName)
    Catch e As SerializationException
        Console.WriteLine("Unable to retrieve date and time information from {0}", _
                          fileName)
    Finally
        inFile.Close
    End Try
End If
' This example displays the following output to the console:
'    Saving 9/3/2008 7:00:00 PM -05:00 Central Daylight Time to .\DateWithTz.dat
'    Restored 9/3/2008 7:00:00 PM -05:00 Central Daylight Time from .\DateWithTz.dat
```

此方法应始终在保存和还原时间前后明确反映正确的时间点，前提是组合的日期和时间及时区对象的实现不允许日期值与时区值不同步。

编译代码

这些示例需要：

- 使用 C# `using` 指令或 Visual Basic `Imports` 语句导入下列命名空间：
  - System（仅限 C#）
  - System.Globalization
  - System.IO
  - System.Runtime.Serialization
  - System.Runtime.Serialization.Formatters.Binary
- 每个代码示例（`DateInTimeZone` 除外）都添加到类或 Visual Basic 模块中，且被包装到方法中，并通过

`Main` 进行调用。

请参阅

- 在 DateTime、DateTimeOffset、TimeSpan 和 TimeZoneInfo 之间进行选择
- 标准日期和时间格式字符串

如何：显示日期和时间值中的毫秒

2021/11/16 •

默认日期和时间格式设置方法（如 DateTime.ToString()）包含时间值的小时、分钟和秒部分，但不包含毫秒部分。本主题说明如何在格式化日期和时间字符串中包含日期和时间的毫秒部分。

显示的 DateTime 值的毫秒部分

1. 如果要使用日期的字符串表示形式，请使用静态 DateTime 或 DateTimeOffset 方法将其转换为 DateTime.Parse(String) 或 DateTimeOffset.Parse(String) 值。
2. 若要提取时间值的毫秒部分的字符串表示形式，请调用日期和时间值的 DateTime.ToString(String) 或 ToString 方法，并将 `fff` 或 `FFF` 自定义格式模式单独传递，或与其他自定义格式说明符一起作为 `format` 参数传递。

示例

此示例展示了如何向控制台传递 DateTime 和 DateTimeOffset 值的毫秒部分（单独传递以及包含在更长的日期和时间字符串中传递）。

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class MillisecondDisplay
{
   public static void Main()
   {
      string dateString = "7/16/2008 8:32:45.126 AM";

      try
      {
         DateTime dateValue = DateTime.Parse(dateString);
         DateTimeOffset dateOffsetValue = DateTimeOffset.Parse(dateString);

         // Display Millisecond component alone.
         Console.WriteLine("Millisecond component only: {0}",
                           dateValue.ToString("fff"));
         Console.WriteLine("Millisecond component only: {0}",
                           dateOffsetValue.ToString("fff"));

         // Display Millisecond component with full date and time.
         Console.WriteLine("Date and Time with Milliseconds: {0}",
                           dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));
         Console.WriteLine("Date and Time with Milliseconds: {0}",
                           dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"));

         // Append millisecond pattern to current culture's full date time pattern
         string fullPattern = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern;
         fullPattern = Regex.Replace(fullPattern, "(:ss|:s)", "$1.fff");

         // Display Millisecond component with modified full date and time pattern.
         Console.WriteLine("Modified full date time pattern: {0}",
                           dateValue.ToString(fullPattern));
         Console.WriteLine("Modified full date time pattern: {0}",
                           dateOffsetValue.ToString(fullPattern));
      }
      catch (FormatException)
      {
         Console.WriteLine("Unable to convert {0} to a date.", dateString);
      }
   }
}
// The example displays the following output if the current culture is en-US:
//    Millisecond component only: 126
//    Millisecond component only: 126
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
//    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
```

```
Imports System.Globalization
Imports System.Text.REgularExpressions

Module MillisecondDisplay
    Public Sub Main()

        Dim dateString As String = "7/16/2008 8:32:45.126 AM"

        Try
            Dim dateValue As Date = Date.Parse(dateString)
            Dim dateOffsetValue As DateTimeOffset = DateTimeOffset.Parse(dateString)

            ' Display Millisecond component alone.
            Console.WriteLine("Millisecond component only: {0}", _
                              dateValue.ToString("fff"))
            Console.WriteLine("Millisecond component only: {0}", _
                              dateOffsetValue.ToString("fff"))

            ' Display Millisecond component with full date and time.
            Console.WriteLine("Date and Time with Milliseconds: {0}", _
                              dateValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"))
            Console.WriteLine("Date and Time with Milliseconds: {0}", _
                              dateOffsetValue.ToString("MM/dd/yyyy hh:mm:ss.fff tt"))

            ' Append millisecond pattern to current culture's full date time pattern
            Dim fullPattern As String = DateTimeFormatInfo.CurrentInfo.FullDateTimePattern
            fullPattern = Regex.Replace(fullPattern, "(:ss|:s)", "$1.fff")

            ' Display Millisecond component with modified full date and time pattern.
            Console.WriteLine("Modified full date time pattern: {0}", _
                              dateValue.ToString(fullPattern))
            Console.WriteLine("Modified full date time pattern: {0}", _
                              dateOffsetValue.ToString(fullPattern))
        Catch e As FormatException
            Console.WriteLine("Unable to convert {0} to a date.", dateString)
        End Try
    End Sub
End Module
' The example displays the following output if the current culture is en-US:
'    Millisecond component only: 126
'    Millisecond component only: 126
'    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
'    Date and Time with Milliseconds: 07/16/2008 08:32:45.126 AM
'    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
'    Modified full date time pattern: Wednesday, July 16, 2008 8:32:45.126 AM
```

`fff` 格式模式包含毫秒值中的任何尾随零。`FFF` 格式模式禁止显示它们。下面的示例阐释了差异。

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine(dateValue.ToString("fff"));
Console.WriteLine(dateValue.ToString("FFF"));
// The example displays the following output to the console:
//    180
//    18
```

```
Dim dateValue As New Date(2008, 7, 16, 8, 32, 45, 180)
Console.WriteLIne(dateValue.ToString("fff"))
Console.WriteLine(dateValue.ToString("FFF"))
' The example displays the following output to the console:
'    180
'    18
```

与定义包含日期和时间的毫秒部分的完整自定义格式说明符有关的一个问题在于，它定义可能与应用程序当前

区域性中的时间元素排列不对应的硬编码格式。更好的替换方法是，检索当前区域性的 DateTimeFormatInfo 对象定义的日期和时间显示模式之一，并将它修改为包含毫秒部分。该示例也阐释了这种方法。它会从 DateTimeFormatInfo.FullDateTimePattern 属性检索当前区域性的完整日期和时间模式，再在秒模式后面插入自定义模式 `.ffff`。请注意，该示例使用正则表达式在单个方法调用中执行此操作。

还可以使用自定义格式说明符来显示毫秒之外的秒的小数部分。例如，`f` 或 `F` 自定义格式说明符显示十分之几秒，`ff` 或 `FF` 自定义格式说明符显示百分之几秒，`ffff` 或 `FFFF` 自定义格式说明符显示万分之几秒。毫秒的小数部分在返回的字符串中会进行截断而不是舍入。下面的示例中使用了这些格式说明符。

```
DateTime dateValue = new DateTime(2008, 7, 16, 8, 32, 45, 180);
Console.WriteLine("{0} seconds", dateValue.ToString("s.f"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ff"));
Console.WriteLine("{0} seconds", dateValue.ToString("s.ffff"));
// The example displays the following output to the console:
//    45.1 seconds
//    45.18 seconds
//    45.1800 seconds
```

```
Dim dateValue As New DateTime(2008, 7, 16, 8, 32, 45, 180)
Console.WriteLine("{0} seconds", dateValue.ToString("s.f"))
Console.WriteLine("{0} seconds", dateValue.ToString("s.ff"))
Console.WriteLine("{0} seconds", dateValue.ToString("s.ffff"))
' The example displays the following output to the console:
'    45.1 seconds
'    45.18 seconds
'    45.1800 seconds
```

NOTE

可以显示秒的非常小的小数单位，如万分之几秒或十万分之几秒。但是，这些值可能没有意义。日期和时间值的精度取决于系统时钟的分辨率。在 Windows NT 3.5 和更高版本以及 Windows Vista 操作系统上，时钟的分辨率大约为 10-15 毫秒。

请参阅

- DateTimeFormatInfo
- 自定义日期和时间格式字符串

如何：用非公历日历显示日期

2021/11/16 •

DateTime 和 DateTimeOffset 类型使用公历作为默认日历。这意味着，调用日期和时间值的 `ToString` 方法会用公历日历显示该日期和时间的字符串表示形式，即使该日期和时间是使用其他日历创建的。下面的示例对此进行了展示，虽然使用两种不同的方式创建采用波斯历的日期和时间值，但在调用 ToString 方法时仍采用公历显示这些日期和时间值。此示例对于用特定日历显示日期，反映了两种常用但不正确的方法。

```
PersianCalendar persianCal = new PersianCalendar();

DateTime persianDate = persianCal.ToDateTime(1387, 3, 18, 12, 0, 0, 0);
Console.WriteLine(persianDate.ToString());

persianDate = new DateTime(1387, 3, 18, persianCal);
Console.WriteLine(persianDate.ToString());
// The example displays the following output to the console:
//       6/7/2008 12:00:00 PM
//       6/7/2008 12:00:00 AM
```

```
Dim persianCal As New PersianCalendar()

Dim persianDate As Date = persianCal.ToDateTime(1387, 3, 18, _
                                                12, 0, 0, 0)
Console.WriteLine(persianDate.ToString())

persianDate = New DateTime(1387, 3, 18, persianCal)
Console.WriteLine(persianDate.ToString())
' The example displays the following output to the console:
'       6/7/2008 12:00:00 PM
'       6/7/2008 12:00:00 AM
```

两种不同的方法可以用于以特定日历显示日期。第一种方法要求日历是特定区域性的默认日历。第二种方法可以与任何日历一起使用。

针对区域性默认日历显示日期

1. 实例化从 Calendar 类派生的日历对象，表示要使用的日历。
2. 实例化 CultureInfo 对象，表示用于显示日期的区域性格式设置。
3. 调用 Array.Exists 方法，以确定日历对象是否为 CultureInfo.OptionalCalendars 属性返回的数组成员。这表明日历可用作 CultureInfo 对象的默认日历。如果它不是数组的成员，请按照"用任何日历显示日期"部分中的说明执行。
4. 将日历对象分配到 CultureInfo.DateTimeFormat 属性返回的 DateTimeFormatInfo 对象的 Calendar 属性。

   > **NOTE**
   > CultureInfo 类还包含 Calendar 属性。不过，它是只读常量，并不会为了反映分配到 DateTimeFormatInfo.Calendar 属性的新默认日历而发生变化。

5. 调用 ToString 或 ToString 方法，并将它传递到在上一步中修改其默认日历的 CultureInfo 对象。

用任何日历显示日期

1. 实例化从 Calendar 类派生的日历对象，表示要使用的日历。

2. 确定应在日期和时间值的字符串表示形式中出现的日期和时间元素。

3. 对于要显示的每个日期和时间元素，调用日历对象的 `Get...` 方法。有以下方法可用：

   - GetYear：采用适当日历显示年份。
   - GetMonth：采用适当日历显示月份。
   - GetDayOfMonth：采用适当日历显示月份中的多少号。
   - GetHour：采用适当日历显示一天中的小时段。
   - GetMinute：采用适当日历显示小时中的分钟数。
   - GetSecond：采用适当日历显示分钟中的秒数。
   - GetMilliseconds：采用适当日历显示分钟中的毫秒数。

示例

该示例使用两个不同日历显示日期。它在将回历定义为 ar-JO 区域性的默认日历之后显示日期，并使用波斯日历（fa-IR 区域性不支持将它作为可选日历）显示日期。

```
using System;
using System.Globalization;

public class CalendarDates
{
   public static void Main()
   {
      HijriCalendar hijriCal = new HijriCalendar();
      CalendarUtility hijriUtil = new CalendarUtility(hijriCal);
      DateTime dateValue1 = new DateTime(1429, 6, 29, hijriCal);
      DateTimeOffset dateValue2 = new DateTimeOffset(dateValue1,
                                  TimeZoneInfo.Local.GetUtcOffset(dateValue1));
      CultureInfo jc = CultureInfo.CreateSpecificCulture("ar-JO");

      // Display the date using the Gregorian calendar.
      Console.WriteLine("Using the system default culture: {0}",
                        dateValue1.ToString("d"));
      // Display the date using the ar-JO culture's original default calendar.
      Console.WriteLine("Using the ar-JO culture's original default calendar: {0}",
                        dateValue1.ToString("d", jc));
      // Display the date using the Hijri calendar.
      Console.WriteLine("Using the ar-JO culture with Hijri as the default calendar:");
      // Display a Date value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc));
      // Display a DateTimeOffset value.
      Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc));

      Console.WriteLine();

      PersianCalendar persianCal = new PersianCalendar();
      CalendarUtility persianUtil = new CalendarUtility(persianCal);
      CultureInfo ic = CultureInfo.CreateSpecificCulture("fa-IR");

      // Display the date using the ir-FA culture's default calendar.
      Console.WriteLine("Using the ir-FA culture's default calendar: {0}",
                        dateValue1.ToString("d", ic));
      // Display a Date value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic));
      // Display a DateTimeOffset value.
      Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic));
   }
```

```
}

public class CalendarUtility
{
   private Calendar thisCalendar;
   private CultureInfo targetCulture;

   public CalendarUtility(Calendar cal)
   {
      this.thisCalendar = cal;
   }

   private bool CalendarExists(CultureInfo culture)
   {
      this.targetCulture = culture;
      return Array.Exists(this.targetCulture.OptionalCalendars,
                          this.HasSameName);
   }

   private bool HasSameName(Calendar cal)
   {
      if (cal.ToString() == thisCalendar.ToString())
         return true;
      else
         return false;
   }

   public string DisplayDate(DateTime dateToDisplay, CultureInfo culture)
   {
      DateTimeOffset displayOffsetDate = dateToDisplay;
      return DisplayDate(displayOffsetDate, culture);
   }

   public string DisplayDate(DateTimeOffset dateToDisplay,
                             CultureInfo culture)
   {
      string specifier = "yyyy/MM/dd";

      if (this.CalendarExists(culture))
      {
         Console.WriteLine("Displaying date in supported {0} calendar...",
                           this.thisCalendar.GetType().Name);
         culture.DateTimeFormat.Calendar = this.thisCalendar;
         return dateToDisplay.ToString(specifier, culture);
      }
      else
      {
         Console.WriteLine("Displaying date in unsupported {0} calendar...",
                           thisCalendar.GetType().Name);

         string separator = targetCulture.DateTimeFormat.DateSeparator;

         return thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") +
                separator +
                thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00") +
                separator +
                thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00");
      }
   }
}
// The example displays the following output to the console:
//       Using the system default culture: 7/3/2008
//       Using the ar-JO culture's original default calendar: 03/07/2008
//       Using the ar-JO culture with Hijri as the default calendar:
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//       Displaying date in supported HijriCalendar calendar...
//       1429/06/29
//
```

```
//       Using the ir-FA culture's default calendar: 7/3/2008
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
//       Displaying date in unsupported PersianCalendar calendar...
//       1387/04/13
```

```
Imports System.Globalization

Public Class CalendarDates
    Public Shared Sub Main()
        Dim hijriCal As New HijriCalendar()
        Dim hijriUtil As New CalendarUtility(hijriCal)
        Dim dateValue1 As Date = New Date(1429, 6, 29, hijriCal)
        Dim dateValue2 As DateTimeOffset = New DateTimeOffset(dateValue1, _
                                           TimeZoneInfo.Local.GetUtcOffset(dateValue1))
        Dim jc As CultureInfo = CultureInfo.CreateSpecificCulture("ar-JO")

        ' Display the date using the Gregorian calendar.
        Console.WriteLine("Using the system default culture: {0}", _
                          dateValue1.ToString("d"))
        ' Display the date using the ar-JO culture's original default calendar.
        Console.WriteLine("Using the ar-JO culture's original default calendar: {0}", _
                          dateValue1.ToString("d", jc))
        ' Display the date using the Hijri calendar.
        Console.WriteLine("Using the ar-JO culture with Hijri as the default calendar:")
        ' Display a Date value.
        Console.WriteLine(hijriUtil.DisplayDate(dateValue1, jc))
        ' Display a DateTimeOffset value.
        Console.WriteLine(hijriUtil.DisplayDate(dateValue2, jc))

        Console.WriteLine()

        Dim persianCal As New PersianCalendar()
        Dim persianUtil As New CalendarUtility(persianCal)
        Dim ic As CultureInfo = CultureInfo.CreateSpecificCulture("fa-IR")

        ' Display the date using the ir-FA culture's default calendar.
        Console.WriteLine("Using the ir-FA culture's default calendar: {0}", _
                          dateValue1.ToString("d", ic))
        ' Display a Date value.
        Console.WriteLine(persianUtil.DisplayDate(dateValue1, ic))
        ' Display a DateTimeOffset value.
        Console.WriteLine(persianUtil.DisplayDate(dateValue2, ic))
    End Sub
End Class

Public Class CalendarUtility
    Private thisCalendar As Calendar
    Private targetCulture As CultureInfo

    Public Sub New(cal As Calendar)
        Me.thisCalendar = cal
    End Sub

    Private Function CalendarExists(culture As CultureInfo) As Boolean
        Me.targetCulture = culture
        Return Array.Exists(Me.targetCulture.OptionalCalendars, _
                            AddressOf Me.HasSameName)
    End Function

    Private Function HasSameName(cal As Calendar) As Boolean
        If cal.ToString() = thisCalendar.ToString() Then
            Return True
        Else
            Return False
        End If
    End Function
```

```
    Public Function DisplayDate(dateToDisplay As Date, _
                                culture As CultureInfo) As String
        Dim displayOffsetDate As DateTimeOffset = dateToDisplay
        Return DisplayDate(displayOffsetDate, culture)
    End Function

    Public Function DisplayDate(dateToDisplay As DateTimeOffset, _
                                culture As CultureInfo) As String
        Dim specifier As String = "yyyy/MM/dd"

        If Me.CalendarExists(culture) Then
            Console.WriteLine("Displaying date in supported {0} calendar...", _
                              thisCalendar.GetType().Name)
            culture.DateTimeFormat.Calendar = Me.thisCalendar
            Return dateToDisplay.ToString(specifier, culture)
        Else
            Console.WriteLine("Displaying date in unsupported {0} calendar...", _
                              thisCalendar.GetType().Name)

            Dim separator As String = targetCulture.DateTimeFormat.DateSeparator

            Return thisCalendar.GetYear(dateToDisplay.DateTime).ToString("0000") & separator & _
                   thisCalendar.GetMonth(dateToDisplay.DateTime).ToString("00") & separator & _
                   thisCalendar.GetDayOfMonth(dateToDisplay.DateTime).ToString("00")
        End If
    End Function
End Class
' The example displays the following output to the console:
'       Using the system default culture: 7/3/2008
'       Using the ar-JO culture's original default calendar: 03/07/2008
'       Using the ar-JO culture with Hijri as the default calendar:
'       Displaying date in supported HijriCalendar calendar...
'       1429/06/29
'       Displaying date in supported HijriCalendar calendar...
'       1429/06/29
'
'       Using the ir-FA culture's default calendar: 7/3/2008
'       Displaying date in unsupported PersianCalendar calendar...
'       1387/04/13
'       Displaying date in unsupported PersianCalendar calendar...
'       1387/04/13
```

每个 CultureInfo 对象都可以支持一个或多个日历(用 OptionalCalendars 属性表示)。日历之一被指定为区域性的默认日历，通过只读属性 CultureInfo.Calendar 进行返回。可以将另一个可选日历指定为默认日历，只需将表示日历的 Calendar 对象分配给 CultureInfo.DateTimeFormat 属性返回的 DateTimeFormatInfo.Calendar 属性即可。不过，某些日历(如 PersianCalendar 类表示的波斯历)不是任何区域性的可选日历。

该示例定义了一个可重用的日历实用工具类 `CalendarUtility`，用于处理有关使用特定日历生成日期的字符串表示形式的许多详细信息。`CalendarUtility` 类包含以下成员：

- 参数化构造函数，其中一个参数是要用来表示日期的 Calendar 对象。这会分配给类的私有字段。
- `CalendarExists`：返回布尔值的专用方法，用于指明以参数形式传递到方法的 CultureInfo 对象是否支持 `CalendarUtility` 对象表示的日历。此方法包装对 Array.Exists 方法的调用，并向其传递 CultureInfo.OptionalCalendars 数组。
- `HasSameName`：专用方法，分配给以参数形式传递到 Array.Exists 方法的 Predicate<T> 委托。数组的每个成员都会传递给该方法，直到该方法返回 `true`。该方法确定可选日历的名称是否与 `CalendarUtility` 对象表示的日历相同。
- `DisplayDate`：重载的公共方法，向它传递下面两个参数：要以 `CalendarUtility` 对象表示的日历表示的 DateTime 或 DateTimeOffset 值，以及要使用其格式设置规则的区域性。它在返回日期的字符串表示形式时的行为取决于要使用其格式设置规则的区域性是否支持目标日历。

无论在此示例中使用哪种日历创建 DateTime 或 DateTimeOffset 值，相应值通常都表示为公历日期。这是因为 DateTime 和 DateTimeOffset 类型不暂留任何日历信息。它们在内部表示自 0001 年 1 月 1 日午夜以来所经历的时钟周期数。该数字的解释取决于日历。对于大多数区域性，默认日历是公历。

.NET 中的字符编码

2021/11/16 •

本文介绍 .NET 使用的 character 编码系统。具体说明如何将 String、Char、Rune和 StringInfo 类型用于 Unicode、UTF-16 和 UTF-8。

本文中使用的术语“character”从读者的角度通常是指单个显示元素 。常见的示例是字母“a”、“@”和表情符号 □ □。有时，一个 character 实际上由多个独立的显示元素组成，具体可以参考介绍字形群集的部分。

string 和 char 类型

string 类的实例表示一些文本。`string` 在逻辑上是一个 16 位值的序列，其中每个值都是 char 结构的实例。string.Length 属性返回 `string` 实例中 `char` 实例的数目。

下面的示例函数以十六进制表示法打印 `string` 中所有 `char` 实例的值：

```
void PrintChars(string s)
{
    Console.WriteLine($"\"{s}\".Length = {s.Length}");
    for (int i = 0; i < s.Length; i++)
    {
        Console.WriteLine($"s[{i}] = '{s[i]}' ('\\u{(int)s[i]:x4}')");
    }
    Console.WriteLine();
}
```

将 string "Hello" 传递给此函数，将获得以下输出：

```
PrintChars("Hello");
```

```
"Hello".Length = 5
s[0] = 'H' ('\u0048')
s[1] = 'e' ('\u0065')
s[2] = 'l' ('\u006c')
s[3] = 'l' ('\u006c')
s[4] = 'o' ('\u006f')
```

每个 character 由一个 `char` 值表示。这种模式适用于世界上大多数语言。例如，下面是两个中文 character 的输出，听起来像“nǐ hǎo”，它们表示“Hello”：

```
PrintChars("你好");
```

```
"你好".Length = 2
s[0] = '你' ('\u4f60')
s[1] = '好' ('\u597d')
```

但是，对于某些语言以及某些符号和表情符号，需要两个 `char` 实例来表示一个 character。例如，比较奥塞治文中表示 Osage 的单词中的 character 和 `char` 实例：

```
PrintChars("𐐷𐑍𐐮𐐨𐑌 𐐏𐐲");
```

```
"𐐷𐑍𐐮𐐨𐑌 𐐏𐐲".Length = 17
s[0] = '?' ('\ud801')
s[1] = '?' ('\udccf')
s[2] = '?' ('\ud801')
s[3] = '?' ('\udcd8')
s[4] = '?' ('\ud801')
s[5] = '?' ('\udcfb')
s[6] = '?' ('\ud801')
s[7] = '?' ('\udcd8')
s[8] = '?' ('\ud801')
s[9] = '?' ('\udcfb')
s[10] = '?' ('\ud801')
s[11] = '?' ('\udcdf')
s[12] = ' ' ('\u0020')
s[13] = '?' ('\ud801')
s[14] = '?' ('\udcbb')
s[15] = '?' ('\ud801')
s[16] = '?' ('\udcdf')
```

在前面的示例中，除空格以外的每个 character 都由两个 `char` 实例表示。

单个 Unicode 表情符号也由两个 `char` 表示，如以下示例中所示的 ox 表情符号：

```
"🐂".Length = 2
s[0] = '?' ('\ud83d')
s[1] = '?' ('\udc02')
```

这些示例表明，`string.Length` 的值表示 `char` 实例的数量，不一定表示显示的 character 数。一个 `char` 实例本身不一定表示一个 character。

映射到单个 character 的 `char` 对称为“代理项对”。若要了解它们的工作原理，需要了解 Unicode 和 UTF-16 编码。

Unicode 码位

Unicode 是一种国际编码标准，可用于各种平台以及各种语言和脚本。

Unicode 标准定义了超过 110 万个码位。码位是一个整数值，范围从 0 到 `U+10FFFF`（十进制 1,114,111）。一些码位被分配给字母、符号或表情符号。其他码位分配给控制文本或 character 显示方式的操作，例如换行。很多码位尚未经分配。

下面是码位分配的一些示例，其中包含指向它们所在的 Unicode chart 的链接：

▯▯▯	HEX	▯▯	▯▯
10	`U+000A`	不可用	换行
97	`U+0061`	a	拉丁文小写字母 a
562	`U+0232`	Ȳ	带长音符的拉丁文大写字母 Y
68,675	`U+10C43`	𐱃	古突厥文字母鄂尔浑文 AT

[illegible]	HEX	[illegible]	[illegible]
127,801	`U+1F339`	🌹	玫瑰花表情符号

通常使用语法 `U+xxxx` 来表示码位，其中 `xxxx` 是十六进制编码的整数值。

整个码位范围包含两个子范围：

- `U+0000..U+FFFF` 范围内的基本多语言平面 (BMP)。这个 16 位范围提供 65,536 个码位，足以涵盖世界上大多数编写系统。
- `U+10000..U+10FFFF` 范围内的补充码位。这个 21 位范围提供了超过一百万个额外的码位，可用于不太知名的语言和其他用途，例如表情符号。

下图说明了 BMP 与补充码位之间的关系。

UTF-16 代码单位

16 位 Unicode 转换格式 (UTF-16) 是一种 character 编码系统，它使用 16 位代码单位来表示 Unicode 码位。 .NET 使用 UTF-16 对 `string` 中的文本进行编码。 `char` 实例表示一个 16 位代码单位。

单个 16 位代码单位可以表示基本多语言平面的 16 位范围内的任何码位。但对于补充范围内的码位，需要两个 `char` 实例。

代理项对

通过称为“代理项码位”的特殊范围（从 `U+D800` 到 `U+DFFF`，十进制 55,296 到 57,343，含限值），可以将两个 16 位值转换为一个 21 位值。

下图说明了 BMP 与代理项码位之间的关系。

如果高代理项码位 (`U+D800..U+DBFF`) 后紧跟低代理项码位 (`U+DC00..U+DFFF`)，则通过使用以下公式，此代理项对将解释为补充码位：

```
code point = 0x10000 +
  ((high surrogate code point - 0xD800) * 0x0400) +
  (low surrogate code point - 0xDC00)
```

下面是使用十进制表示法的相同公式：

```
code point = 65,536 +
  ((high surrogate code point - 55,296) * 1,024) +
  (low surrogate code point - 56,320)
```

高代理项码位的数字值不高于低代理项码位的数字值。高代理项码位之所以称为“高”，是因为它用于计算完整 21 位码位范围的高阶 11 位。低代理项码位用于计算低阶 10 位。

例如，与代理项对对应的实际码位 `0xD83C` 和 `0xDF39` 按如下方式计算：

```
actual = 0x10000 + ((0xD83C - 0xD800) * 0x0400) + (0xDF39 - 0xDC00)
       = 0x10000 + (          0x003C  * 0x0400) +           0x0339
       = 0x10000 +                      0xF000  +           0x0339
       = 0x1F339
```

下面是使用十进制表示法的相同计算：

```
actual =  65,536 + ((55,356 - 55,296) * 1,024) + (57,145 - 56320)
       =  65,536 + (              60  * 1,024) +            825
       =  65,536 +                    61,440  +            825
       = 127,801
```

前一个示例展示的 `"\ud83c\udf39"` 是前面提到的 `U+1F339 ROSE ('🌹')` 码位的 UTF-16 编码。

Unicode 标量值

术语"Unicode 标量值"是指除代理项码位之外的所有码位。换句话说，标量值是分配有 character 或将来可以为其分配 character 的任何码位。此处的"字符"是指可以分配给码位的任何内容，其中包括控制文本或 character 显示方式的操作。

下图演示了标量值码位。

作为标量值的 Rune 类型

从 .NET Core 3.0 开始，System.Text.Rune 类型表示 Unicode 标量值。`Rune` 在 .NET Core 2.x 或 .NET Framework 4.x 中不可用。

`Rune` 构造函数验证生成的实例是否为有效的 Unicode 标量值，如果无效，则引发异常。下面的示例展示成功实例化 `Rune` 实例的代码，因为输入代表有效的标量值：

```
Rune a = new Rune('a');
Rune b = new Rune(0x0061);
Rune c = new Rune('\u0061');
Rune d = new Rune(0x10421);
Rune e = new Rune('\ud801', '\udc21');
```

下面的示例引发异常，因为码位在代理项范围内，但不是代理项对的一部分：

```
Rune f = new Rune('\ud801');
```

下面的示例引发异常，因为码位超出了补充范围：

```
Rune g = new Rune(0x12345678);
```

Rune 用法示例：更改字母大小写

如果 `char` 来自代理项对，则采用 `char` 并假设正在使用作为标量值的码位的 API 将无法正常工作。例如，来看看以下方法，此方法对字符串 (string 中的每个 char 调用 Char.ToUpperInvariant：

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string ConvertToUpperBadExample(string input)
{
    StringBuilder builder = new StringBuilder(input.Length);
    for (int i = 0; i < input.Length; i++) /* or 'foreach' */
    {
        builder.Append(char.ToUpperInvariant(input[i]));
    }
    return builder.ToString();
}
```

如果 `input` 字符串 (string) 包含小写德塞莱特文字母 `er` (𐐽)，则此代码不会将其转换为大写形式 (𐐕)。此代码对代理项码位 `U+D801` 和 `U+DC49` 分别调用 `char.ToUpperInvariant`。但 `U+D801` 本身没有足够的信息将其标识为小写字母，因此 `char.ToUpperInvariant` 将其保持不变。它以相同的方式处理 `U+DC49`。结果是 `input` 字符串 (string) 中的小写“𐐽”不会转换为大写的“𐐕”。

以下两个选项可用于将字符串 (string) 正确地转换为大写形式：

- 对 input 字符串 (string) 调用 String.ToUpperInvariant，而不是循环访问（一个 `char` 接着一个 `char`）。`string.ToUpperInvariant` 方法可以访问每个代理项对的两个部分，因此它可以正确地处理所有 Unicode 码位。
- 循环访问 Unicode 标量值作为 `Rune` 实例，而不是 `char` 实例，如以下示例中所示。由于 `Rune` 实例是有效的 Unicode 标量值，因此可将其传递给应对标量值执行操作的 API。例如，如以下示例中所示，调用 Rune.ToUpperInvariant 可以得到正确的结果：

  ```
  static string ConvertToUpper(string input)
  {
      StringBuilder builder = new StringBuilder(input.Length);
      foreach (Rune rune in input.EnumerateRunes())
      {
          builder.Append(Rune.ToUpperInvariant(rune));
      }
      return builder.ToString();
  }
  ```

其他 Rune API

`Rune` 类型公开了许多 `char` API 的类似 API。例如，以下方法在 `char` 类型上镜像静态 API：

- Rune.IsLetter
- Rune.IsWhiteSpace
- Rune.IsLetterOrDigit
- Rune.GetUnicodeCategory

若要从 `Rune` 实例获取原始标量值，请使用 Rune.Value 属性。

若要将 `Rune` 实例转换回一连串 `char`，请使用 Rune.ToString 或 Rune.EncodeToUtf16 方法。

由于任何 Unicode 标量值都可以由单个 `char` 或代理项对表示，因此任何 `Rune` 实例最多可由 2 个 `char` 实例表示。使用 Rune.Utf16SequenceLength 查看表示 `Rune` 实例所需的 `char` 实例数目。

有关 .NET `Rune` 类型的详细信息，请参阅 `Rune` API 参考。

字形群集

看起来像一个 character 的内容可能由多个码位组合而成，因此，相比“character”，“字形群集”术语的表述通常更

贴合。在 .NET 中使用“文本元素”术语表示相同的内容。

比如，`string` 实例“a”、“á”、“á”和“🤦🏼‍♀️”。如果你的操作系统按照 Unicode 标准指定的方式来处理，则这些 `string` 实例中的每个实例都将显示为单个文本元素或字形群集。但最后两个实例用多个标量值码位表示。

- 字符串 (string)“a”由一个标量值表示，并包含一个 `char` 实例。
  - `U+0061 LATIN SMALL LETTER A`
- 字符串 (string)“á”由一个标量值表示，并包含一个 `char` 实例。
  - `U+00E1 LATIN SMALL LETTER A WITH ACUTE`
- 字符串 (string)“á”看起来与“á”相同，但由两个标量值表示，并包含两个 `char` 实例。
  - `U+0061 LATIN SMALL LETTER A`
  - `U+0301 COMBINING ACUTE ACCENT`
- 最后，字符串 (string)“🤦🏼‍♀️”由四个标量值表示，并包含七个 `char` 实例。
  - `U+1F469 WOMAN`（补充范围，需要一个代理项对）
  - `U+1F3FD EMOJI MODIFIER FITZPATRICK TYPE-4`（补充范围，需要一个代理项对）
  - `U+200D ZERO WIDTH JOINER`
  - `U+1F692 FIRE ENGINE`（补充范围，需要一个代理项对）

在前面的一些示例中（例如组合的重音修饰符或肤色修饰符），码位不会在屏幕上显示为独立元素。相反，它用于修改之前出现的文本元素的外观。这些示例表明，可能需要采用多个标量值来构成我们认为的单个“character”或“字形群集”。

若要枚举 `string` 的字形群集，请使用 StringInfo 类，如下面的示例中所示。如果你熟悉 Swift，那么 .NET `StringInfo` 类型在概念上类似于 Swift `character` 类型。

示例：char、Rune 和文本元素实例计数

在 .NET API 中，字形群集称为“文本元素”。下面的方法演示 `string` 中 `char`、`Rune` 和文本元素实例之间的差异：

```
static void PrintTextElementCount(string s)
{
    Console.WriteLine(s);
    Console.WriteLine($"Number of chars: {s.Length}");
    Console.WriteLine($"Number of runes: {s.EnumerateRunes().Count()}");

    TextElementEnumerator enumerator = StringInfo.GetTextElementEnumerator(s);

    int textElementCount = 0;
    while (enumerator.MoveNext())
    {
        textElementCount++;
    }

    Console.WriteLine($"Number of text elements: {textElementCount}");
```

```
PrintTextElementCount("á");
// Number of chars: 1
// Number of runes: 1
// Number of text elements: 1

PrintTextElementCount("ǎ");
// Number of chars: 2
// Number of runes: 2
// Number of text elements: 1

PrintTextElementCount("□□□");
// Number of chars: 7
// Number of runes: 4
// Number of text elements: 1
```

如果在 .NET Framework 或 .NET Core 3.1 或更早版本中运行此代码，表情符号的文本元素计数将显示 `4`。这是由于 `StringInfo` 类中的 bug 所致（已在 .NET 5 中修复）。

示例：拆分 string 实例

拆分 `string` 实例时，请避免拆分代理项对和字形群集。下面的示例展示不正确的代码，此代码的目的是在 string 中每隔 10 个 character 插入一个换行符：

```
// THE FOLLOWING METHOD SHOWS INCORRECT CODE.
// DO NOT DO THIS IN A PRODUCTION APPLICATION.
static string InsertNewlinesEveryTencharsBadExample(string input)
{
    StringBuilder builder = new StringBuilder();

    // First, append chunks in multiples of 10 chars
    // followed by a newline.
    int i = 0;
    for (; i < input.Length - 10; i += 10)
    {
        builder.Append(input, i, 10);
        builder.AppendLine(); // newline
    }

    // Then append any leftover data followed by
    // a final newline.
    builder.Append(input, i, input.Length - i);
    builder.AppendLine(); // newline

    return builder.ToString();
}
```

由于此代码枚举 `char` 实例，因此会拆分一个跨越 10 个 `char` 边界的代理项对，并在它们之间插入一个换行符。这种插入会导致数据损坏，因为代理项码位只有在作为代理项对时才具有意义。

如果枚举 `Rune` 实例（标量值）而不是 `char` 实例，仍可能损坏数据。一组 `Rune` 实例可能构成跨越 10 个 `char` 边界的字形群集。如果一组字形群集被拆分开，就不再有效。

更好的方法是通过对字形群集或文本元素进行计数来中断 string，如以下示例中所示：

```
static string InsertNewlinesEveryTenTextElements(string input)
{
    StringBuilder builder = new StringBuilder();

    // Append chunks in multiples of 10 chars

    TextElementEnumerator enumerator = StringInfo.GetTextElementEnumerator(input);

    int textElementCount = 1;
    while (enumerator.MoveNext())
    {
        builder.Append(enumerator.Current);
        if (textElementCount % 10 == 0 && textElementCount > 0)
        {
            builder.AppendLine(); // newline
        }
        textElementCount++;
    }

    // Add a final newline.
    builder.AppendLine(); // newline
    return builder.ToString();

}
```

但是，如前所述，在 .NET（而非 .NET 5）的实现中，`StringInfo` 类可能会错误地处理某些字形群集。

UTF-8 和 UTF-32

前面几节着重于介绍 UTF-16，因为 .NET 要使用它对 `string` 实例进行编码。Unicode 还有其他编码系统：即 UTF-8 和 UTF-32。这些编码分别使用 8 位代码单位和 32 位代码单位。

与 UTF-16 类似，UTF-8 需要使用多个代码单位表示某些 Unicode 标量值。UTF-32 可以表示单个 32 位代码单位中的任何标量值。

下面的示例展示如何分别使用这三个 Unicode 编码系统表示同一个 Unicode 码位：

```
Scalar: U+0061 LATIN SMALL LETTER A ('a')
UTF-8 : [ 61 ]           (1x  8-bit code unit  = 8 bits total)
UTF-16: [ 0061 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000061 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+0429 CYRILLIC CAPITAL LETTER SHCHA ('Щ')
UTF-8 : [ D0 A9 ]        (2x  8-bit code units = 16 bits total)
UTF-16: [ 0429 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 00000429 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+A992 JAVANESE LETTER GA ('ꦒ')
UTF-8 : [ EA A6 92 ]     (3x  8-bit code units = 24 bits total)
UTF-16: [ A992 ]         (1x 16-bit code unit  = 16 bits total)
UTF-32: [ 0000A992 ]     (1x 32-bit code unit  = 32 bits total)

Scalar: U+104CC OSAGE CAPITAL LETTER TSHA ('𐓌')
UTF-8 : [ F0 90 93 8C ]  (4x  8-bit code units = 32 bits total)
UTF-16: [ D801 DCCC ]    (2x 16-bit code units = 32 bits total)
UTF-32: [ 000104CC ]     (1x 32-bit code unit  = 32 bits total)
```

如前文所述，代理项对中的单个 UTF-16 代码单位本身是没有意义的。同样，如果单个 UTF-8 代码单位处于由两个、三个或四个用于计算标量值的一连串代码单位中，那么它自身也毫无意义。

字节排序方式

在 .NET 中，string 的 UTF-16 代码单位以 16 位整数（`char` 实例）的序列形式存储在连续内存中。各个代码单位

的位数根据当前体系结构的字节顺序布局。

在 little-endian 体系结构中，由 UTF-16 码位 `[ D801 DCCC ]` 组成的 string 会在内存中以 `[ 0x01, 0xD8, 0xCC, 0xDC ]` 字节进行布局。在 big-endian 体系结构中，同一 string 将在内存中以 `[ 0xD8, 0x01, 0xDC, 0xCC ]` 字节进行布局。

相互通信的计算机系统必须就跨网络数据的表示形式达成共识。大多数网络协议在传输文本时都使用 UTF-8 标准，部分原因是为了避免 big-endian 计算机与 little-endian 计算机通信可能导致的问题。不管字节顺序如何，由 UTF-8 码位 `[ F0 90 93 8C ]` 组成的 string 将始终表示为字节 `[ 0xF0, 0x90, 0x93, 0x8C ]`。

若要使用 UTF-8 传输文本，.NET 应用程序通常使用类似于以下示例的代码：

```
string stringToWrite = GetString();
byte[] stringAsUtf8Bytes = Encoding.UTF8.GetBytes(stringToWrite);
await outputStream.WriteAsync(stringAsUtf8Bytes, 0, stringAsUtf8Bytes.Length);
```

在前面的示例中，方法 Encoding.UTF8.GetBytes 将 UTF-16 `string` 解码回一系列 Unicode 标量值，然后将这些标量值重新编码为 UTF-8，并将生成的序列放入 `byte` 数组。Encoding.UTF8.GetString 方法执行相反的转换，将 UTF-8 `byte` 数组转换为 UTF-16 `string`。

> **WARNING**
> 由于 UTF-8 在 Internet 上很普遍，因此从网络读取原始字节并将数据视为 UTF-8 会很有吸引力。但是，应验证它的格式是否正确。恶意客户端可能向你的服务提交格式错误的 UTF-8。如果你按正确的格式处理数据，可能会导致应用程序出错或存在安全漏洞。要验证 UTF-8 数据，可以使用类似于 `Encoding.UTF8.GetString` 的方法，此方法在将传入数据转换为 `string` 时将执行验证。

格式正确的编码

格式正确的 Unicode 编码是一串 (string) 代码单位，可以将它毫无歧义地正确解码为一系列 Unicode 标量值。格式正确的数据可以在 UTF-8、UTF-16 和 UTF-32 之间自由地来回转码。

编码序列格式是否正确的问题与计算机体系结构的字节顺序无关。在 big-endian 和 little-endian 计算机上，格式错误的 UTF-8 序列都具有相同的错误方式。

以下是一些格式错误的编码示例：

- 在 UTF-8 中，序列 `[ 6C C2 61 ]` 格式错误，因为 `C2` 后面不能跟有 `61`。
- 在 UTF-16 中，序列 `[ DC00 DD00 ]`（或者，在 C# 中为字符串 (string) `"\udc00\udd00"`）格式错误，因为低代理项 `DC00` 后面不能跟有另一个低代理项 `DD00`。
- 在 UTF-32 中，序列 `[ 0011ABCD ]` 格式错误，因为 `0011ABCD` 不在 Unicode 标量值的范围内。

在 .NET 中，`string` 实例几乎总是包含格式正确的 UTF-16 数据，但也不能百分之百地保证。以下示例展示有效的 C# 代码在 `string` 实例中创建格式不正确的 UTF-16 数据。

- 格式错误的文字：

```
const string s = "\ud800";
```

- 拆分代理项对的 substring：

```
string x = "\ud83e\udd70"; // "🥰"
string y = x.Substring(1, 1); // "\udd70" standalone low surrogate
```

像 `Encoding.UTF8.GetString` 这样的 API 永远不会返回格式错误的 `string` 实例。`Encoding.GetString` 和 `Encoding.GetBytes` 方法检测输入中格式错误的序列，并在生成输出时执行 character 替换。例如，如果 `Encoding.ASCII.GetString(byte[])` 在输入中发现非 ASCII 字节（超出 U+0000..U+007F 的范围），它会在返回的 `string` 实例中插入一个“?”。`Encoding.UTF8.GetString(byte[])` 在返回的 `string` 实例中将格式错误的 UTF-8 序列替换为 `U+FFFD REPLACEMENT CHARACTER ('')`。有关详细信息，请参阅 5.22 和 3.9 小节中的 Unicode 标准。

在出现格式错误的序列时，也可以将内置 `Encoding` 类配置为引发异常，而不是执行 character 替换。在可能无法接受 character 替换的安全敏感的应用程序中，通常可以使用这种方法。

```
byte[] utf8Bytes = ReadFromNetwork();
UTF8Encoding encoding = new UTF8Encoding(encoderShouldEmitUTF8Identifier: false, throwOnInvalidBytes: true);
string asString = encoding.GetString(utf8Bytes); // will throw if 'utf8Bytes' is ill-formed
```

要了解如何使用内置 `Encoding` 类，请参阅如何在 .NET 中使用 character 编码类。

请参阅

- String
- Char
- Rune
- 全球化和本地化

如何在 .NET 中使用字符编码类

2021/11/16 •

本文介绍如何使用 .NET 提供的类通过各种编码方案对文本进行编码和解码。这些说明假定你已阅读 .NET 中的字符编码简介。

编码器和解码器

.NET 提供了编码类，这些类使用各种编码系统对文本进行编码和解码。例如，UTF8Encoding 类描述用于编码到 UTF-8 以及从 UTF-8 解码的规则。.NET 对 `string` 实例使用 UTF-16 编码（由 UnicodeEncoding 类表示）。编码器和解码器还适用于其他编码方案。

编码和解码还可以包括验证。例如，UnicodeEncoding 类检查代理项范围内的所有 `char` 实例，确保它们是有效的代理项对。回退策略确定编码器处理无效字符的方式或解码器处理无效字节的方式。

> **WARNING**
> .NET 编码类可用于存储和转换字符数据。它们不应用于存储字符串形式的二进制数据。根据所使用的编码，用编码类将二进制数据转换为字符串格式可引起意外的行为，并生成不准确或损坏的数据。若要将二进制数据转换为字符串形式，请使用 Convert.ToBase64String 方法。

.NET 中的所有字符编码类都继承自 System.Text.Encoding 类，这是定义所有字符编码通用功能的抽象类。若要访问在 .NET 中实现的单个编码对象，请执行以下操作：

- 使用 Encoding 类的静态属性，这些属性返回表示 .NET 标准字符编码（ASCII、UTF-7、UTF-8、UTF-16 和 UTF-32）的对象。例如，Encoding.Unicode 属性返回 UnicodeEncoding 对象。每个对象都使用替换回退处理不能进行编码的字符串和不能进行解码的字节。有关详细信息，请参阅替换回退。
- 调用编码的类构造函数。以这种方式可以将 ASCII、utf-7、utf-8、utf-16 和 utf-32 编码对象实例化。默认情况下，每个对象都使用替换回退处理不能进行编码的字符串和不能进行解码的字节，但你可指定应引发异常。有关详细信息，请参阅替换回退和异常回退。
- 调用 Encoding(Int32) 构造函数并向其传递一个表示编码的整数。标准编码对象使用替换回退，代码页编码和双字节字符集 (DBCS) 编码对象使用最佳回退处理不能进行编码的字符串和不能进行解码的字节。有关详细信息，请参阅最佳回退。
- 调用 Encoding.GetEncoding 方法，此方法返回 .NET 中的任何标准编码、代码页编码或 DBCS 编码。可通过重载同时指定编码器和解码器的回退对象。

可以检索所有 .NET 编码的相关信息，具体操作是调用 Encoding.GetEncodings 方法。.NET 支持下表中列出的字符编码系统。

ASCII	通过使用较低的七位字节将有限范围的字符进行编码。由于此编码仅支持从 U+0000 到 U+007F 的字符值，因此在大多数情况下不足以支持国际化的应用程序。

UTF-7	将字符表示为 7 位 ASCII 字符的序列。非 ASCII Unicode 字符由 ASCII 字符的转义序列表示。UTF-7 支持电子邮件和新闻组协等协议。但是，utf-7 不是特别安全或可靠。在某些情况下，更改一位可以彻底更改对整个 utf-7 字符串的解释。在其他情况下，不同的 utf-7 字符串可以对相同的文本进行编码。对于包含非 ASCII 字符的序列，utf-7 需要比 utf-8 更多的空间，且编码/解码更慢。因此，应尽可能使用 utf-8，而不是 utf-7。
UTF-8	将每个 Unicode 码位表示为一至四个字节的序列。Utf-8 支持 8 位数据大小，适用于许多现有的操作系统。对于字符的 ASCII 范围，utf-8 等同于 ASCII 编码，并适用于范围更广的字符集。但是，对于中日韩 (CJK) 脚本而言，针对每个字符 utf-8 可能需要三个字节，并且可能导致比 utf-16 更大的数据大小。有时 ASCII 数据（如 HTML 标记）的量证明了 CJK 范围大小增加的合理性。
UTF-16	将每个 Unicode 码位表示为一至两个 16 位整数的序列。尽管 Unicode 补充字符（U+10000 和更高版本）需要两个 utf-16 代理项码位，但最常见的 Unicode 字符仅需一个 utf-16 码位。little-endian 和 big endian 字节顺序均受支持。公共语言运行时使用 Utf-16 编码表示 Char 和 String 值，Windows 操作系统使用它表示 `WCHAR` 值。
UTF-32	将每个 Unicode 码位表示为一个 32 位整数。little-endian 和 big endian 字节顺序均受支持。当应用程序想要避免操作系统上的 utf-16 编码的代理项码位行为时，则使用 utf-32 编码，编码空间对操作系统十分重要。显示器上呈现的单个标志符号仍可使用多个 UTF-32 字符进行编码。
ANSI/ISO 编码	为各种代码页提供支持。在 Windows 操作系统上，代码页用于支持特定语言或语言组。有关列出 .NET 支持代码页的表，请参阅 Encoding 类。通过调用 Encoding.GetEncoding(Int32) 方法可检索特定代码页的编码对象。一个代码页包含 256 个码位，并且是从零开始。在大多数代码页中，码位 0 到 127 表示 ASCII 字符集，而码位 128 到 255 在代码页之间存在显著差异。例如，代码页 1252 为拉丁语书写系统（包括英语、德语和法语）提供字符。代码页 1252 中最后 128 个码位包含重音字符。代码页 1253 提供希腊语书写系统中所需的字符代码。代码页 1253 中最后 128 个码位包含希腊语字符。因此，基于 ANSI 代码页的应用程序不能将希腊语和德语存储在同一个文本流中，除非它包含一个指示所引用的代码页的标识符。
双字节字符集 (DBCS) 编码	支持包含超过 256 个字符的语言，例如中文、日语和朝鲜语。在 DBCS 中，一对码位（双字节）表示一个字符。Encoding.IsSingleByte 属性返回 DBCS 编码的 `false` 。通过调用 Encoding.GetEncoding(Int32) 方法可以为特定的 DBCS 检索编码对象。当应用程序处理 DBCS 数据时，DBCS 字符的第一个字节（前导字节）与紧随其后的结尾字节一起处理。因为根据代码页，一对双字节码位可以代表不同的字符，此模式仍不支持在同一数据流中进行两种语言（如日语和中文）的组合。

这些编码使你能够使用 Unicode 字符以及旧版应用程序中最常用的编码。此外，通过定义从 Encoding 派生的类并重写其成员，可创建自定义编码。

.NET Core 编码支持

默认情况下，.NET Core 不提供除代码页 28591 以外的其他任何代码页编码和 Unicode 编码，例如 UTF-8 和 UTF-16。不过，可以向应用中添加定目标到 .NET 的标准 Windows 应用中的代码页编码。有关详细信息，请参见CodePagesEncodingProvider主题。

选择编码类

如果有机会选择应用程序要使用的编码，则应使用 Unicode 编码，最好是 UTF8Encoding 或 UnicodeEncoding。（.NET 还支持第三种 Unicode 编码，即 UTF32Encoding。）

如果打算使用 ASCII 编码 (ASCIIEncoding)，请选择 UTF8Encoding 。这两个编码对于 ASCII 字符集而言是相同的，但 UTF8Encoding 具有以下优点：

- 它可以表示每个 Unicode 字符，而 ASCIIEncoding 仅支持 U+0000 到 U+007F 之间的 Unicode 字符值。
- 它提供错误检测，具有更高的安全性。
- 速度已达到最优，应快于任何其他编码。即使对于全是 ASCII 的内容，使用 UTF8Encoding 执行操作的速度要快于 ASCIIEncoding。

应考虑仅针对旧版应用程序使用 ASCIIEncoding 。但是，即使对于旧版应用程序，由于以下原因，UTF8Encoding 可能是更好的选择（假定采用默认设置）：

- 如果应用程序具有的内容不完全是 ASCII 并使用 ASCIIEncoding将其编码，则每个非 ASCII 字符将编码为一个问号 (?)。如果应用程序随后对此数据进行解码，则信息丢失。
- 如果应用程序具有的内容不完全是 ASCII 并使用 UTF8Encoding将其编码，则结果看起来将无法识别（如果解释为 ASCII）。但是，如果应用程序随后使用 utf-8 解码器将此数据进行解码，则数据成功执行一次往返过程。

在 web 应用程序中，发送到响应 web 请求的客户端中的字符应反映客户端上所使用的编码。在大多数情况下，应将 HttpResponse.ContentEncoding 属性设置为 HttpRequest.ContentEncoding 属性返回的值，从而以用户期望的编码显示文本。

使用编码对象

编码器将一个字符串（最常见的为 Unicode 字符）转换为其数字（字节）等效项。例如，你可能会使用 ASCII 编码器将 Unicode 字符转换为 ASCII，以便可在控制台中显示。若要执行此转换，调用 Encoding.GetBytes 方法。如果想要在执行编码前确定需要多少个字节来存储编码字符，可调用 GetByteCount 方法。

下面的示例使用单个字节数组，从而在两个单独的操作中对字符串进行编码。它为下一组 ASCII 编码字节保持指示字节数组中的起始位置的索引。调用 ASCIIEncoding.GetByteCount(String) 方法，以确保字节数组足够大，从而可容纳已编码的字符串。然后调用 ASCIIEncoding.GetBytes(String, Int32, Int32, Byte[], Int32) 方法以在字符串中对字符进行编码。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings= { "This is the first sentence. ",
                          "This is the second sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;

      // Create array of adequate size.
      byte[] bytes = new byte[49];
      // Create index for current position of array.
      int index = 0;

      Console.WriteLine("Strings to encode:");
      foreach (var stringValue in strings) {
         Console.WriteLine("   {0}", stringValue);

         int count = asciiEncoding.GetByteCount(stringValue);
         if (count + index >=  bytes.Length)
            Array.Resize(ref bytes, bytes.Length + 50);

         int written = asciiEncoding.GetBytes(stringValue, 0,
                                              stringValue.Length,
                                              bytes, index);

         index = index + written;
      }
      Console.WriteLine("\nEncoded bytes:");
      Console.WriteLine("{0}", ShowByteValues(bytes, index));
      Console.WriteLine();

      // Decode Unicode byte array to a string.
      string newString = asciiEncoding.GetString(bytes, 0, index);
      Console.WriteLine("Decoded: {0}", newString);
   }

   private static string ShowByteValues(byte[] bytes, int last )
   {
      string returnString = "   ";
      for (int ctr = 0; ctr <= last - 1; ctr++) {
         if (ctr % 20 == 0)
            returnString += "\n   ";
         returnString += String.Format("{0:X2} ", bytes[ctr]);
      }
      return returnString;
   }
}
// The example displays the following output:
//       Strings to encode:
//          This is the first sentence.
//          This is the second sentence.
//
//       Encoded bytes:
//
//          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
//          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
//
//       Decoded: This is the first sentence. This is the second sentence.
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim strings() As String = {"This is the first sentence. ",
                                   "This is the second sentence. "}
        Dim asciiEncoding As Encoding = Encoding.ASCII

        ' Create array of adequate size.
        Dim bytes(50) As Byte
        ' Create index for current position of array.
        Dim index As Integer = 0

        Console.WriteLine("Strings to encode:")
        For Each stringValue In strings
            Console.WriteLine("   {0}", stringValue)

            Dim count As Integer = asciiEncoding.GetByteCount(stringValue)
            If count + index >= bytes.Length Then
                Array.Resize(bytes, bytes.Length + 50)
            End If
            Dim written As Integer = asciiEncoding.GetBytes(stringValue, 0,
                                                            stringValue.Length,
                                                            bytes, index)

            index = index + written
        Next
        Console.WriteLine()
        Console.WriteLine("Encoded bytes:")
        Console.WriteLine("{0}", ShowByteValues(bytes, index))
        Console.WriteLine()

        ' Decode Unicode byte array to a string.
        Dim newString As String = asciiEncoding.GetString(bytes, 0, index)
        Console.WriteLine("Decoded: {0}", newString)
    End Sub

    Private Function ShowByteValues(bytes As Byte(), last As Integer) As String
        Dim returnString As String = "   "
        For ctr As Integer = 0 To last - 1
            If ctr Mod 20 = 0 Then returnString += vbCrLf + "   "
            returnString += String.Format("{0:X2} ", bytes(ctr))
        Next
        Return returnString
    End Function
End Module
' The example displays the following output:
'       Strings to encode:
'          This is the first sentence.
'          This is the second sentence.
'
'       Encoded bytes:
'
'          54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
'          6E 74 65 6E 63 65 2E 20 54 68 69 73 20 69 73 20 74 68 65 20
'          73 65 63 6F 6E 64 20 73 65 6E 74 65 6E 63 65 2E 20
'
'       Decoded: This is the first sentence. This is the second sentence.
```

解码器将反映特定字符编码的字节数组转换为字符数组或字符串中的一组字符。若要将字节数组解码为字符数组，则调用 Encoding.GetChars 方法。若要将字节数组解码为字符串，则调用 GetString 方法。如果想在执行解码前确定需要多少个字符来存储已解码的字节，则可调用 GetCharCount 方法。

下面的示例对三个字符串进行编码，然后将它们解码为单个字符数组。它为下一组解码的字符保持指示字符数组中的起始位置的索引。调用 GetCharCount 方法，以确保字符数组足够大，从而可容纳所有已解码的字符。然

后调用 ASCIIEncoding.GetChars(Byte[], Int32, Int32, Char[], Int32) 方法对字节数组进行解码。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      string[] strings = { "This is the first sentence. ",
                           "This is the second sentence. ",
                           "This is the third sentence. " };
      Encoding asciiEncoding = Encoding.ASCII;
      // Array to hold encoded bytes.
      byte[] bytes;
      // Array to hold decoded characters.
      char[] chars = new char[50];
      // Create index for current position of character array.
      int index = 0;

      foreach (var stringValue in strings) {
         Console.WriteLine("String to Encode: {0}", stringValue);
         // Encode the string to a byte array.
         bytes = asciiEncoding.GetBytes(stringValue);
         // Display the encoded bytes.
         Console.Write("Encoded bytes: ");
         for (int ctr = 0; ctr < bytes.Length; ctr++)
            Console.Write(" {0}{1:X2}",
                          ctr % 20 == 0 ? Environment.NewLine : "",
                          bytes[ctr]);
         Console.WriteLine();

         // Decode the bytes to a single character array.
         int count = asciiEncoding.GetCharCount(bytes);
         if (count + index >=  chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         int written = asciiEncoding.GetChars(bytes, 0,
                                              bytes.Length,
                                              chars, index);
         index = index + written;
         Console.WriteLine();
      }

      // Instantiate a single string containing the characters.
      string decodedString = new string(chars, 0, index - 1);
      Console.WriteLine("Decoded string: ");
      Console.WriteLine(decodedString);
   }
}
// The example displays the following output:
//    String to Encode: This is the first sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the second sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
//    65 6E 74 65 6E 63 65 2E 20
//
//    String to Encode: This is the third sentence.
//    Encoded bytes:
//    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
//    6E 74 65 6E 63 65 2E 20
//
//    Decoded string:
//    This is the first sentence. This is the second sentence. This is the third sentence.
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim strings() As String = {"This is the first sentence. ",
                                   "This is the second sentence. ",
                                   "This is the third sentence. "}
        Dim asciiEncoding As Encoding = Encoding.ASCII
        ' Array to hold encoded bytes.
        Dim bytes() As Byte
        ' Array to hold decoded characters.
        Dim chars(50) As Char
        ' Create index for current position of character array.
        Dim index As Integer

        For Each stringValue In strings
            Console.WriteLine("String to Encode: {0}", stringValue)
            ' Encode the string to a byte array.
            bytes = asciiEncoding.GetBytes(stringValue)
            ' Display the encoded bytes.
            Console.Write("Encoded bytes: ")
            For ctr As Integer = 0 To bytes.Length - 1
                Console.Write(" {0}{1:X2}", If(ctr Mod 20 = 0, vbCrLf, ""),
                                            bytes(ctr))
            Next
            Console.WriteLine()

            ' Decode the bytes to a single character array.
            Dim count As Integer = asciiEncoding.GetCharCount(bytes)
            If count + index >= chars.Length Then
                Array.Resize(chars, chars.Length + 50)
            End If
            Dim written As Integer = asciiEncoding.GetChars(bytes, 0,
                                                            bytes.Length,
                                                            chars, index)
            index = index + written
            Console.WriteLine()
        Next

        ' Instantiate a single string containing the characters.
        Dim decodedString As New String(chars, 0, index - 1)
        Console.WriteLine("Decoded string: ")
        Console.WriteLine(decodedString)
    End Sub
End Module
' The example displays the following output:
'    String to Encode: This is the first sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 66 69 72 73 74 20 73 65
'    6E 74 65 6E 63 65 2E 20
'
'    String to Encode: This is the second sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 73 65 63 6F 6E 64 20 73
'    65 6E 74 65 6E 63 65 2E 20
'
'    String to Encode: This is the third sentence.
'    Encoded bytes:
'    54 68 69 73 20 69 73 20 74 68 65 20 74 68 69 72 64 20 73 65
'    6E 74 65 6E 63 65 2E 20
'
'    Decoded string:
'    This is the first sentence. This is the second sentence. This is the third sentence.
```

从 Encoding 派生的类的编码和解码方法旨在用于一组完整的数据；也就是说，在单个方法调用中提供要进行编码或解码的所有数据。但是，在某些情况下，数据在流中可用，并且要编码或解码的数据可能仅可从单独的读取

操作中获取。这要求编码或解码操作记住其之前调用中保存的任何状态。从 Encoder 和 Decoder 派生的类的方法能够处理跨多个方法调用的编码和解码操作。

特定编码的 Encoder 对象可从此编码的 Encoding.GetEncoder 属性获取。特定编码的 Decoder 对象可从此编码的 Encoding.GetDecoder 属性获取。对于解码操作，请注意，从 Decoder 派生的类包括 Decoder.GetChars 方法，但它们不具有对应于 Encoding.GetString的方法。

下面的示例演示使用 Encoding.GetString 和 Decoder.GetChars 方法解码 Unicode 字节数组的差异。该示例将包含某些 Unicode 字符的字符串编码为文件，然后使用两种解码方法一次解码十个字节。由于代理项对发生在第十个和第十一个字节，因此它在单独的方法调用中进行解码。如输出所示，Encoding.GetString 方法不能正确地对字节进行解码，而是将它们替换为 U + FFFD（替换字符）。另一方面，Decoder.GetChars 方法能够成功地对字节数组进行解码以获取原始字符串。

```
using System;
using System.IO;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Use default replacement fallback for invalid encoding.
      UnicodeEncoding enc = new UnicodeEncoding(true, false, false);

      // Define a string with various Unicode characters.
      string str1 = "AB YZ 19 \uD800\udc05 \u00e4";
      str1 += "Unicode characters. \u00a9 \u010C s \u0062\u0308";
      Console.WriteLine("Created original string...\n");

      // Convert string to byte array.
      byte[] bytes = enc.GetBytes(str1);

      FileStream fs = File.Create(@".\characters.bin");
      BinaryWriter bw = new BinaryWriter(fs);
      bw.Write(bytes);
      bw.Close();

      // Read bytes from file.
      FileStream fsIn = File.OpenRead(@".\characters.bin");
      BinaryReader br = new BinaryReader(fsIn);

      const int count = 10;            // Number of bytes to read at a time.
      byte[] bytesRead = new byte[10]; // Buffer (byte array).
      int read;                        // Number of bytes actually read.
      string str2 = String.Empty;      // Decoded string.

      // Try using Encoding object for all operations.
      do {
         read = br.Read(bytesRead, 0, count);
         str2 += enc.GetString(bytesRead, 0, read);
      } while (read == count);
      br.Close();
      Console.WriteLine("Decoded string using UnicodeEncoding.GetString()...");
      CompareForEquality(str1, str2);
      Console.WriteLine();

      // Use Decoder for all operations.
      fsIn = File.OpenRead(@".\characters.bin");
      br = new BinaryReader(fsIn);
      Decoder decoder = enc.GetDecoder();
      char[] chars = new char[50];
      int index = 0;                   // Next character to write in array.
      int written = 0;                 // Number of chars written to array.
      do {
         read = br.Read(bytesRead, 0, count);
         if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >= chars.Length)
```

```
         if (index + decoder.GetCharCount(bytesRead, 0, read) - 1 >= chars.Length)
            Array.Resize(ref chars, chars.Length + 50);

         written = decoder.GetChars(bytesRead, 0, read, chars, index);
         index += written;
      } while (read == count);
      br.Close();
      // Instantiate a string with the decoded characters.
      string str3 = new String(chars, 0, index);
      Console.WriteLine("Decoded string using UnicodeEncoding.Decoder.GetString()...");
      CompareForEquality(str1, str3);
   }

   private static void CompareForEquality(string original, string decoded)
   {
      bool result = original.Equals(decoded);
      Console.WriteLine("original = decoded: {0}",
                        original.Equals(decoded, StringComparison.Ordinal));
      if (! result) {
         Console.WriteLine("Code points in original string:");
         foreach (var ch in original)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();

         Console.WriteLine("Code points in decoded string:");
         foreach (var ch in decoded)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    Created original string...
//
//    Decoded string using UnicodeEncoding.GetString()...
//    original = decoded: False
//    Code points in original string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055 006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//    Code points in decoded string:
//    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055 006E 0069 0063 006F
//    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
//    0020 0073 0020 0062 0308
//
//    Decoded string using UnicodeEncoding.Decoder.GetString()...
//    original = decoded: True
```

```
Imports System.IO
Imports System.Text

Module Example
    Public Sub Main()
        ' Use default replacement fallback for invalid encoding.
        Dim enc As New UnicodeEncoding(True, False, False)

        ' Define a string with various Unicode characters.
        Dim str1 As String = String.Format("AB YZ 19 {0}{1} {2}",
                                           ChrW(&hD800), ChrW(&hDC05), ChrW(&h00e4))
        str1 += String.Format("Unicode characters. {0} {1} s {2}{3}",
                              ChrW(&h00a9), ChrW(&h010C), ChrW(&h0062), ChrW(&h0308))
        Console.WriteLine("Created original string...")
        Console.WriteLine()

        ' Convert string to byte array.
        Dim bytes() As Byte = enc.GetBytes(str1)

        Dim fs As FileStream = File.Create(".\characters.bin")
```

```
        Dim fs As FileStream = File.Create(".\characters.bin")
        Dim bw As New BinaryWriter(fs)
        bw.Write(bytes)
        bw.Close()

        ' Read bytes from file.
        Dim fsIn As FileStream = File.OpenRead(".\characters.bin")
        Dim br As New BinaryReader(fsIn)

        Const count As Integer = 10      ' Number of bytes to read at a time.
        Dim bytesRead(9) As Byte         ' Buffer (byte array).
        Dim read As Integer              ' Number of bytes actually read.
        Dim str2 As String = ""          ' Decoded string.

        ' Try using Encoding object for all operations.
        Do
            read = br.Read(bytesRead, 0, count)
            str2 += enc.GetString(bytesRead, 0, read)
        Loop While read = count
        br.Close()
        Console.WriteLine("Decoded string using UnicodeEncoding.GetString()...")
        CompareForEquality(str1, str2)
        Console.WriteLine()

        ' Use Decoder for all operations.
        fsIn = File.OpenRead(".\characters.bin")
        br = New BinaryReader(fsIn)
        Dim decoder As Decoder = enc.GetDecoder()
        Dim chars(50) As Char
        Dim index As Integer = 0         ' Next character to write in array.
        Dim written As Integer = 0       ' Number of chars written to array.
        Do
            read = br.Read(bytesRead, 0, count)
            If index + decoder.GetCharCount(bytesRead, 0, read) - 1 >= chars.Length Then
                Array.Resize(chars, chars.Length + 50)
            End If
            written = decoder.GetChars(bytesRead, 0, read, chars, index)
            index += written
        Loop While read = count
        br.Close()
        ' Instantiate a string with the decoded characters.
        Dim str3 As New String(chars, 0, index)
        Console.WriteLine("Decoded string using UnicodeEncoding.Decoder.GetString()...")
        CompareForEquality(str1, str3)
    End Sub

    Private Sub CompareForEquality(original As String, decoded As String)
        Dim result As Boolean = original.Equals(decoded)
        Console.WriteLine("original = decoded: {0}",
                          original.Equals(decoded, StringComparison.Ordinal))
        If Not result Then
            Console.WriteLine("Code points in original string:")
            For Each ch In original
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()

            Console.WriteLine("Code points in decoded string:")
            For Each ch In decoded
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
        End If
    End Sub
End Module
' The example displays the following output:
'    Created original string...
'
'    Decoded string using UnicodeEncoding.GetString()...
'    original = decoded: False
```

```
'    original = decoded: False
'    Code points in original string:
'    0041 0042 0020 0059 005A 0020 0031 0039 0020 D800 DC05 0020 00E4 0055 006E 0069 0063 006F
'    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
'    0020 0073 0020 0062 0308
'    Code points in decoded string:
'    0041 0042 0020 0059 005A 0020 0031 0039 0020 FFFD FFFD 0020 00E4 0055 006E 0069 0063 006F
'    0064 0065 0020 0063 0068 0061 0072 0061 0063 0074 0065 0072 0073 002E 0020 00A9 0020 010C
'    0020 0073 0020 0062 0308
'
'    Decoded string using UnicodeEncoding.Decoder.GetString()...
'    original = decoded: True
```

选择回退策略

当某个方法尝试对字符进行编码或解码，但不存在映射时，它必须实现回退策略，以确定应如何处理失败的映射。有三种类型的回退策略：

- Best-Fit Fallback
- Replacement Fallback
- Exception Fallback

> **IMPORTANT**
>
> 当某一 Unicode 字符不能映射到特定代码页编码时，在编码操作中将发生最常见的问题。当无效的字节序列无法转换为有效的 Unicode 字符，在解码操作中将发生最常见的问题。出于这些原因，应该了解特定的编码对象使用哪种回退策略。只要有可能，应指定实例化对象时编码对象使用的回退策略。

Best-Fit Fallback

当一个字符在目标编码中不具有准确匹配时，编码器可以尝试将其映射到类似的字符。（最佳回退主要是编码问题而非解码问题。很少有代码页包含无法成功映射到 Unicode 的字符。）最佳回退是代码页的默认设置，以及 Encoding.GetEncoding(Int32) 和 Encoding.GetEncoding(String) 重载检索的双字节字符集编码。

> **NOTE**
>
> 从理论上讲，.NET 中的 Unicode 编码类（UTF8Encoding、UnicodeEncoding 和 UTF32Encoding）支持所有字符集中的每个字符，因此可用于消除最佳匹配回退问题。

最佳匹配策略因代码页而异。例如，对于某些代码页，全角拉丁字符映射到更常见的半角拉丁字符。对于其他代码页，不进行此映射。即使是在积极的最佳策略下，也不能完全适合某些编码中的某些字符。例如，中文象形文字不具有到代码页 1252 的合理映射。在这种情况下，使用替换字符串。默认情况下，此字符串只是一个问号 (U+003F)。

> **NOTE**
>
> 最佳匹配策略不会得到详细记录。不过，Unicode 联盟网站上记录了多个代码页。若要了解如何解释映射文件，请查看相应文件夹中的 readme.txt 文件。

下面的示例使用代码页 1252（适合西欧语言 Windows 代码页）演示最佳映射及其缺点。Encoding.GetEncoding(Int32) 方法用于检索代码页 1252 的编码对象。默认情况下，它使用其不支持的 Unicode 字符的最佳映射。该示例将包含三个非 ASCII 字符的字符串实例化，这三个字符分别为带圆圈拉丁文大写字母 S (U+24C8)、上标五 (U+2075) 和无穷大 (U+221E) 且由空格分隔。如示例输出所示，当对字符串进行编码时，三

个原始的非空格字符替换为问号 (U+003F)、数字五 (U+0035) 和数字八 (U+0038)。数字八是对不受支持的无穷大字符的不良替换，问号指示没有映射可用于原始字符。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Get an encoding for code page 1252 (Western Europe character set).
      Encoding cp1252 = Encoding.GetEncoding(1252);

      // Define and display a string.
      string str = "\u24c8 \u2075 \u221e";
      Console.WriteLine("Original string: " + str);
      Console.Write("Code points in string: ");
      foreach (var ch in str)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode a Unicode string.
      Byte[] bytes = cp1252.GetBytes(str);
      Console.Write("Encoded bytes: ");
      foreach (byte byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the string.
      string str2 = cp1252.GetString(bytes);
      Console.WriteLine("String round-tripped: {0}", str.Equals(str2));
      if (! str.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));
      }
   }
}
// The example displays the following output:
//       Original string: Ⓢ ⁵ ∞
//       Code points in string: 24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 35 20 38
//
//       String round-tripped: False
//       ? 5 8
//       003F 0020 0035 0020 0038
```

```
Imports System.Text

Module Example
    Public Sub Main()
        ' Get an encoding for code page 1252 (Western Europe character set).
        Dim cp1252 As Encoding = Encoding.GetEncoding(1252)

        ' Define and display a string.
        Dim str As String = String.Format("{0} {1} {2}", ChrW(&h24c8), ChrW(&H2075), ChrW(&h221E))
        Console.WriteLine("Original string: " + str)
        Console.Write("Code points in string: ")
        For Each ch In str
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Encode a Unicode string.
        Dim bytes() As Byte = cp1252.GetBytes(str)
        Console.Write("Encoded bytes: ")
        For Each byt In bytes
            Console.Write("{0:X2} ", byt)
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Decode the string.
        Dim str2 As String = cp1252.GetString(bytes)
        Console.WriteLine("String round-tripped: {0}", str.Equals(str2))
        If Not str.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
        End If
    End Sub
End Module
' The example displays the following output:
'       Original string: Ⓢ ⁵ ∞
'       Code points in string: 24C8 0020 2075 0020 221E
'
'       Encoded bytes: 3F 20 35 20 38
'
'       String round-tripped: False
'       ? 5 8
'       003F 0020 0035 0020 0038
```

最佳映射是 Encoding 对象的默认行为，该对象将 Unicode 数据编码为代码页数据，并且存在依赖此行为的旧版应用程序。但是，为了安全起见，大多数新应用程序应避免最佳行为。例如，应用程序不应通过最佳编码放置域名。

> **NOTE**
> 还可实现编码的自定义最佳回退映射。有关详细信息，请参阅 Implementing a Custom Fallback Strategy 一节。

如果最佳回退是编码对象的默认设置，当通过调用 Encoding 或 Encoding.GetEncoding(Int32, EncoderFallback, DecoderFallback) 重载检索 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 对象时，可选择另一个回退策略。以下一节的内容包括一个示例，该示例用星号 (*) 替换每个不可映射到代码页 1252 的每个字符。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                 new EncoderReplacementFallback("*"),
                                 new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim cp1252r As Encoding = Encoding.GetEncoding(1252,
                                          New EncoderReplacementFallback("*"),
                                          New DecoderReplacementFallback("*"))

        Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
        Console.WriteLine(str1)
        For Each ch In str1
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
        Next
        Console.WriteLine()

        Dim bytes() As Byte = cp1252r.GetBytes(str1)
        Dim str2 As String = cp1252r.GetString(bytes)
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
        If Not str1.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
        End If
    End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'       Round-trip: False
'       * * *
'       002A 0020 002A 0020 002A
```

Replacement Fallback

当字符在目标方案中没有准确匹配，但是也没有其可映射到的相应字符，则应用程序可指定替换字符或字符串。这是 Unicode 编码器的默认行为，此默认行为替换其无法用 REPLACEMENT_CHARACTER (U+FFFD) 进行解码的任何双字节序列。它也是 ASCIIEncoding 类的默认行为，此默认行为替换其无法用问号进行编码或解码的每个字符。下面的示例演示上一示例中 Unicode 字符串的字符替换。如输出所示，不能解码为 ASCII 字节值的每个字符都替换为 0x3F，这是问号的 ASCII 代码。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.ASCII;

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);
      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Encoded bytes: 3F 20 3F 20 3F
//
//       Round-trip: False
//       ? ? ?
//       003F 0020 003F 0020 003F
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim enc As Encoding = Encoding.Ascii

        Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
        Console.WriteLine(str1)
        For Each ch In str1
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Encode the original string using the ASCII encoder.
        Dim bytes() As Byte = enc.GetBytes(str1)
        Console.Write("Encoded bytes: ")
        For Each byt In bytes
            Console.Write("{0:X2} ", byt)
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Decode the ASCII bytes.
        Dim str2 As String = enc.GetString(bytes)
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
        If Not str1.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
        End If
    End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'
'       Encoded bytes: 3F 20 3F 20 3F
'
'       Round-trip: False
'       ? ? ?
'       003F 0020 003F 0020 003F
```

.NET 包括 EncoderReplacementFallback 和 DecoderReplacementFallback 类，用于替换字符没有在编码或解码操作中完全映射时的替换字符串。默认情况下，此替换字符串是一个问号，但可以调用类构造函数重载以选择不同的字符串。通常，替换字符串是单个字符，但这不是一项要求。下面的示例通过将以星号 (*) 作为替换字符串的 EncoderReplacementFallback 对象实例化，更改代码页 1252 编码器的行为。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding cp1252r = Encoding.GetEncoding(1252,
                                 new EncoderReplacementFallback("*"),
                                 new DecoderReplacementFallback("*"));

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine();

      byte[] bytes = cp1252r.GetBytes(str1);
      string str2 = cp1252r.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//       Round-trip: False
//       * * *
//       002A 0020 002A 0020 002A
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim cp1252r As Encoding = Encoding.GetEncoding(1252,
                                          New EncoderReplacementFallback("*"),
                                          New DecoderReplacementFallback("*"))

        Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
        Console.WriteLine(str1)
        For Each ch In str1
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
        Next
        Console.WriteLine()

        Dim bytes() As Byte = cp1252r.GetBytes(str1)
        Dim str2 As String = cp1252r.GetString(bytes)
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
        If Not str1.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
        End If
    End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'       Round-trip: False
'       * * *
'       002A 0020 002A 0020 002A
```

NOTE

还可以实现编码的替换类。有关详细信息，请参阅 Implementing a Custom Fallback Strategy 一节。

除了问号 (U+003F) 外，Unicode 替换字符 (U+FFFD) 通常用作替换字符串，特别是当解码无法成功转换为 Unicode 字符的字节序列时。但是，你可以自由选择任何替换字符串，并且它可以包含多个字符。

Exception Fallback

如果编码器不能对一组字符进行编码，它就不提供最佳回退或替换字符串，而是可能引发 EncoderFallbackException ，如果解码器不能对字节数组进行解码，它就可能引发 DecoderFallbackException 。若要在编码和解码操作中引发异常，请分别提供一个 EncoderExceptionFallback 对象和一个 DecoderExceptionFallback 对象到 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 方法。下面的示例用 ASCIIEncoding 类演示异常回退。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii",
                                          new EncoderExceptionFallback(),
                                          new DecoderExceptionFallback());

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
```

```
      foreach (var ch in str1)
         Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

      Console.WriteLine("\n");

      // Encode the original string using the ASCII encoder.
      byte[] bytes = {};
      try {
         bytes = enc.GetBytes(str1);
         Console.Write("Encoded bytes: ");
         foreach (var byt in bytes)
            Console.Write("{0:X2} ", byt);

         Console.WriteLine();
      }
      catch (EncoderFallbackException e) {
         Console.Write("Exception: ");
         if (e.IsUnknownSurrogate())
            Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} 0x{1:X3} at index {2}.",
                              Convert.ToUInt16(e.CharUnknownHigh),
                              Convert.ToUInt16(e.CharUnknownLow),
                              e.Index);
         else
            Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                              Convert.ToUInt16(e.CharUnknown),
                              e.Index);
         return;
      }
      Console.WriteLine();

      // Decode the ASCII bytes.
      try {
         string str2 = enc.GetString(bytes);
         Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
         if (! str1.Equals(str2)) {
            Console.WriteLine(str2);
            foreach (var ch in str2)
               Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

            Console.WriteLine();
         }
      }
      catch (DecoderFallbackException e) {
         Console.Write("Unable to decode byte(s) ");
         foreach (byte unknown in e.BytesUnknown)
            Console.Write("0x{0:X2} ");

         Console.WriteLine("at index {0}", e.Index);
      }
   }
}
// The example displays the following output:
//       Ⓢ ⁵ ∞
//       24C8 0020 2075 0020 221E
//
//       Exception: Unable to encode 0x24C8 at index 0.
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim enc As Encoding = Encoding.GetEncoding("us-ascii",
                                                   New EncoderExceptionFallback(),
                                                   New DecoderExceptionFallback())

        Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&h24C8), ChrW(&h2075), ChrW(&h221E))
        Console.WriteLine(str1)
        For Each ch In str1
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Encode the original string using the ASCII encoder.
        Dim bytes() As Byte = {}
        Try
            bytes = enc.GetBytes(str1)
            Console.Write("Encoded bytes: ")
            For Each byt In bytes
                Console.Write("{0:X2} ", byt)
            Next
            Console.WriteLine()
        Catch e As EncoderFallbackException
            Console.Write("Exception: ")
            If e.IsUnknownSurrogate() Then
                Console.WriteLine("Unable to encode surrogate pair 0x{0:X4} 0x{1:X3} at index {2}.",
                                  Convert.ToUInt16(e.CharUnknownHigh),
                                  Convert.ToUInt16(e.CharUnknownLow),
                                  e.Index)
            Else
                Console.WriteLine("Unable to encode 0x{0:X4} at index {1}.",
                                  Convert.ToUInt16(e.CharUnknown),
                                  e.Index)
            End If
            Exit Sub
        End Try
        Console.WriteLine()

        ' Decode the ASCII bytes.
        Try
            Dim str2 As String = enc.GetString(bytes)
            Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
            If Not str1.Equals(str2) Then
                Console.WriteLine(str2)
                For Each ch In str2
                    Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
                Next
                Console.WriteLine()
            End If
        Catch e As DecoderFallbackException
            Console.Write("Unable to decode byte(s) ")
            For Each unknown As Byte In e.BytesUnknown
                Console.Write("0x{0:X2} ")
            Next
            Console.WriteLine("at index {0}", e.Index)
        End Try
    End Sub
End Module
' The example displays the following output:
'       Ⓢ ⁵ ∞
'       24C8 0020 2075 0020 221E
'
'       Exception: Unable to encode 0x24C8 at index 0.
```

> **NOTE**
> 还可以实现编码操作的自定义异常处理程序。有关详细信息，请参阅 Implementing a Custom Fallback Strategy 一节。

EncoderFallbackException 和 DecoderFallbackException 对象提供以下有关导致异常的条件的信息：

- EncoderFallbackException 对象包括 IsUnknownSurrogate 方法，该方法指示不能对其进行编码的一个字符或多个字符是代表未知的代理项对（在这种情况下，该方法返回 `true`）还是未知的单个字符（在这种情况下，该方法将返回 `false`）。代理项对中的字符可从 EncoderFallbackException.CharUnknownHigh 和 EncoderFallbackException.CharUnknownLow 属性获取。未知的单个字符可从 EncoderFallbackException.CharUnknown 属性获取。EncoderFallbackException.Index 属性指示字符串中第一个无法进行编码的字符的位置。

- DecoderFallbackException 对象包括 BytesUnknown 属性，该属性返回一个无法解码的字节数组。DecoderFallbackException.Index 属性指示未知字节的起始位置。

尽管 EncoderFallbackException 和 DecoderFallbackException 对象提供足够的有关异常的诊断信息，但它们不提供对编码或解码缓冲区的访问权限。因此，它们不允许在编码或解码方法内替换或更正无效数据。

Implementing a Custom Fallback Strategy

除了由代码页在内部实现的最佳映射，.NET 包括用于实现回退策略的以下类：

- 使用 EncoderReplacementFallback 和 EncoderReplacementFallbackBuffer 来替换编码操作中的字符。
- 使用 DecoderReplacementFallback 和 DecoderReplacementFallbackBuffer 来替换解码操作中的字符。
- 当字符不能进行编码时，使用 EncoderExceptionFallback 和 EncoderExceptionFallbackBuffer 来引发 EncoderFallbackException 。
- 当字符不能进行解码时，使用 DecoderExceptionFallback 和 DecoderExceptionFallbackBuffer 引发 DecoderFallbackException 。

此外，通过执行以下步骤，可以实现使用最佳回退、替换回退或异常回退的自定义解决方案：

1. 从 EncoderFallback 派生一个类用于编码操作，并从 DecoderFallback 派生一个类用于解码操作。
2. 从 EncoderFallbackBuffer 派生一个类用于编码操作，并从 DecoderFallbackBuffer 派生一个类用于解码操作。
3. 有关异常回退，如果预定义的 EncoderFallbackException 和 DecoderFallbackException 类不能满足你的需求，则从异常对象中派生一个类，如 Exception 或 ArgumentException。

从 EncoderFallback 或 DecoderFallback 中派生

若要实现自定义的回退解决方案，必须创建一个继承自 EncoderFallback 的类用于编码操作，以及一个继承自 DecoderFallback 的类用于解码操作。这些类的实例传递给 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 方法，并作为编码类和回退实现之间的媒介。

当为编码器或解码器创建自定义回退解决方案时，必须实现以下成员：

- EncoderFallback.MaxCharCount 或 DecoderFallback.MaxCharCount 属性，返回最佳、替换或异常回退可能返回以替换单个字符的最大数量的字符数。对于自定义异常回退，其值为 0。
- EncoderFallback.CreateFallbackBuffer 或 DecoderFallback.CreateFallbackBuffer 方法，该方法返回自定义的 EncoderFallbackBuffer 或 DecoderFallbackBuffer 实现。当遇到不能成功进行编码的第一个字节时或当遇到不能成功进行解码的第一个字节时，则解码器调用该方法。

从 EncoderFallbackBuffer 或 DecoderFallbackBuffer 中派生

若要实现自定义的回退解决方案，还必须创建一个继承自 EncoderFallbackBuffer 的类用于编码操作，以及一个继承自 DecoderFallbackBuffer 的类用于解码操作。CreateFallbackBuffer 和 EncoderFallback 类的 DecoderFallback 方法返回这些类的实例。当遇到不能对其进行编码第一个字符时，编码器调用 EncoderFallback.CreateFallbackBuffer 方法，当遇到不能对其进行解码的一个或多个字节时，解码器调用 DecoderFallback.CreateFallbackBuffer 方法。EncoderFallbackBuffer 和 DecoderFallbackBuffer 类提供回退实现。每个实例表示一个包含回退字符的缓冲区，该回退字符将替换不能进行编码的字符或不能进行解码的字节序列。

当为编码器或解码器创建自定义回退解决方案时，必须实现以下成员：

- EncoderFallbackBuffer.Fallback 或 DecoderFallbackBuffer.Fallback 方法。编码器调用EncoderFallbackBuffer.Fallback 来为回退缓冲区提供其不能进行编码的字符的相关信息。因为要进行编码的字符可能是代理项对，此方法被重载。向重载传递了字符串中将进行编码的字符及其索引。向第二个重载传递了字符串中高代理项和低代理项及其索引。解码器调用 DecoderFallbackBuffer.Fallback 方法来为回退缓冲区提供有关无法解码的字节的信息。向此方法传递了其无法解码的字节数组以及第一个字节的索引。如果回退缓冲区可以提供一个或多个最佳或替换字符，则回退方法应返回 `true` ，否则应返回 `false` 。对于异常回退，回退方法应引发异常。
- EncoderFallbackBuffer.GetNextChar 或 DecoderFallbackBuffer.GetNextChar 方法，编码器或解码器重复调用该方法以从回退缓冲区获取下一个字符。返回所有回退字符后，该方法应返回 U+0000。
- EncoderFallbackBuffer.Remaining 或 DecoderFallbackBuffer.Remaining 属性，返回回退缓冲区中的剩余字符数。
- EncoderFallbackBuffer.MovePrevious 或 DecoderFallbackBuffer.MovePrevious 方法，将回退缓冲区中的当前位置移到前一个字符。
- EncoderFallbackBuffer.Reset 或 DecoderFallbackBuffer.Reset 方法，将回退缓冲区重新初始化。

如果回退实现是最佳回退或替换回退，则从 EncoderFallbackBuffer 和 DecoderFallbackBuffer 派生的类还保持两个私有实例字段：缓冲区中字符的精确数目；缓冲区中下一个要返回的字符的索引。

EncoderFallback 示例

前面的一个示例使用替换回退替换与带星号 (*) 的 ASCII 字符不对应的 Unicode 字符。下面的示例改为使用自定义的最佳回退实现提供更好的非 ASCII 字符映射。

下面的代码定义一个名为 `CustomMapper` 、派生自 EncoderFallback 的类，用以处理非 ASCII 字符的最佳映射。其 `CreateFallbackBuffer` 方法返回 `CustomMapperFallbackBuffer` 对象，该对象提供 EncoderFallbackBuffer 实现。 `CustomMapper` 类使用 Dictionary<TKey,TValue> 对象存储不受支持的 Unicode 字符（键值）和其对应的 8 位字符（以 64 位整数存储在两个连续字节中）的映射。若要使此映射可用于回退缓冲区，将 `CustomMapper` 实例作为参数传递给 `CustomMapperFallbackBuffer` 类构造函数。因为最长的映射是 Unicode 字符 U+221E 的字符串“INF”，所以 `MaxCharCount` 属性返回 3。

```
public class CustomMapper : EncoderFallback
{
   public string DefaultString;
   internal Dictionary<ushort, ulong> mapping;

   public CustomMapper() : this("*")
   {
   }

   public CustomMapper(string defaultString)
   {
      this.DefaultString = defaultString;

      // Create table of mappings
      mapping = new Dictionary<ushort, ulong>();
      mapping.Add(0x24C8, 0x53);
      mapping.Add(0x2075, 0x35);
      mapping.Add(0x221E, 0x49004E0046);
   }

   public override EncoderFallbackBuffer CreateFallbackBuffer()
   {
      return new CustomMapperFallbackBuffer(this);
   }

   public override int MaxCharCount
   {
      get { return 3; }
   }
}
```

```
Public Class CustomMapper : Inherits EncoderFallback
    Public DefaultString As String
    Friend mapping As Dictionary(Of UShort, ULong)

    Public Sub New()
        Me.New("?")
    End Sub

    Public Sub New(ByVal defaultString As String)
        Me.DefaultString = defaultString

        ' Create table of mappings
        mapping = New Dictionary(Of UShort, ULong)
        mapping.Add(&H24C8, &H53)
        mapping.Add(&H2075, &H35)
        mapping.Add(&H221E, &H49004E0046)
    End Sub

    Public Overrides Function CreateFallbackBuffer() As System.Text.EncoderFallbackBuffer
        Return New CustomMapperFallbackBuffer(Me)
    End Function

    Public Overrides ReadOnly Property MaxCharCount As Integer
        Get
            Return 3
        End Get
    End Property
End Class
```

下面的代码定义 `CustomMapperFallbackBuffer` 类，该类派生自 EncoderFallbackBuffer。包含最佳映射并在 `CustomMapper` 实例中定义的字典可从其类构造函数获取。如果映射字典中定义了 ASCII 编码器无法对其进行编码的 Unicode 字符，则其 `Fallback` 方法返回 `true` ；否则返回 `false` 。对于每个回退，私有 `count` 变量指示仍

需返回的字符数目，私有 `index` 变量指示字符串缓冲区 `charsToReturn` 中下一个要返回的字符的位置。

```
public class CustomMapperFallbackBuffer : EncoderFallbackBuffer
{
   int count = -1;                  // Number of characters to return
   int index = -1;                  // Index of character to return
   CustomMapper fb;
   string charsToReturn;

   public CustomMapperFallbackBuffer(CustomMapper fallback)
   {
      this.fb = fallback;
   }

   public override bool Fallback(char charUnknownHigh, char charUnknownLow, int index)
   {
      // Do not try to map surrogates to ASCII.
      return false;
   }

   public override bool Fallback(char charUnknown, int index)
   {
      // Return false if there are already characters to map.
      if (count >= 1) return false;

      // Determine number of characters to return.
      charsToReturn = String.Empty;

      ushort key = Convert.ToUInt16(charUnknown);
      if (fb.mapping.ContainsKey(key)) {
         byte[] bytes = BitConverter.GetBytes(fb.mapping[key]);
         int ctr = 0;
         foreach (var byt in bytes) {
            if (byt > 0) {
               ctr++;
               charsToReturn += (char) byt;
            }
         }
         count = ctr;
      }
      else {
         // Return default.
         charsToReturn = fb.DefaultString;
         count = 1;
      }
      this.index = charsToReturn.Length - 1;

      return true;
   }

   public override char GetNextChar()
   {
      // We'll return a character if possible, so subtract from the count of chars to return.
      count--;
      // If count is less than zero, we've returned all characters.
      if (count < 0)
         return '\u0000';

      this.index--;
      return charsToReturn[this.index + 1];
   }

   public override bool MovePrevious()
   {
      // Original: if count >= -1 and pos >= 0
      if (count >= -1) {
         count++;
         return true;
```

```
      }
      else {
         return false;
      }
   }

   public override int Remaining
   {
      get { return count < 0 ? 0 : count; }
   }

   public override void Reset()
   {
      count = -1;
      index = -1;
   }
}
```

```
Public Class CustomMapperFallbackBuffer : Inherits EncoderFallbackBuffer

    Dim count As Integer = -1        ' Number of characters to return
    Dim index As Integer = -1        ' Index of character to return
    Dim fb As CustomMapper
    Dim charsToReturn As String

    Public Sub New(ByVal fallback As CustomMapper)
        MyBase.New()
        Me.fb = fallback
    End Sub

    Public Overloads Overrides Function Fallback(ByVal charUnknownHigh As Char, ByVal charUnknownLow As
Char, ByVal index As Integer) As Boolean
        ' Do not try to map surrogates to ASCII.
        Return False
    End Function

    Public Overloads Overrides Function Fallback(ByVal charUnknown As Char, ByVal index As Integer) As
Boolean
        ' Return false if there are already characters to map.
        If count >= 1 Then Return False

        ' Determine number of characters to return.
        charsToReturn = String.Empty

        Dim key As UShort = Convert.ToUInt16(charUnknown)
        If fb.mapping.ContainsKey(key) Then
            Dim bytes() As Byte = BitConverter.GetBytes(fb.mapping.Item(key))
            Dim ctr As Integer
            For Each byt In bytes
                If byt > 0 Then
                    ctr += 1
                    charsToReturn += Chr(byt)
                End If
            Next
            count = ctr
        Else
            ' Return default.
            charsToReturn = fb.DefaultString
            count = 1
        End If
        Me.index = charsToReturn.Length - 1

        Return True
    End Function

    Public Overrides Function GetNextChar() As Char
        ' We'll return a character if possible, so subtract from the count of chars to return.
```

```
        count -= 1
        ' If count is less than zero, we've returned all characters.
        If count < 0 Then Return ChrW(0)

        Me.index -= 1
        Return charsToReturn(Me.index + 1)
    End Function

    Public Overrides Function MovePrevious() As Boolean
        ' Original: if count >= -1 and pos >= 0
        If count >= -1 Then
            count += 1
            Return True
        Else
            Return False
        End If
    End Function

    Public Overrides ReadOnly Property Remaining As Integer
        Get
            Return If(count < 0, 0, count)
        End Get
    End Property

    Public Overrides Sub Reset()
        count = -1
        index = -1
    End Sub
End Class
```

然后下面的代码实例化 `CustomMapper` 对象并将它的一个实例传递给 Encoding.GetEncoding(String, EncoderFallback, DecoderFallback) 方法。输出指示最佳回退实现成功处理原始字符串中的三个非 ASCII 字符。

```
using System;
using System.Collections.Generic;
using System.Text;

class Program
{
   static void Main()
   {
      Encoding enc = Encoding.GetEncoding("us-ascii", new CustomMapper(), new DecoderExceptionFallback());

      string str1 = "\u24C8 \u2075 \u221E";
      Console.WriteLine(str1);
      for (int ctr = 0; ctr <= str1.Length - 1; ctr++) {
         Console.Write("{0} ", Convert.ToUInt16(str1[ctr]).ToString("X4"));
         if (ctr == str1.Length - 1)
            Console.WriteLine();
      }
      Console.WriteLine();

      // Encode the original string using the ASCII encoder.
      byte[] bytes = enc.GetBytes(str1);
      Console.Write("Encoded bytes: ");
      foreach (var byt in bytes)
         Console.Write("{0:X2} ", byt);

      Console.WriteLine("\n");

      // Decode the ASCII bytes.
      string str2 = enc.GetString(bytes);
      Console.WriteLine("Round-trip: {0}", str1.Equals(str2));
      if (! str1.Equals(str2)) {
         Console.WriteLine(str2);
         foreach (var ch in str2)
            Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"));

         Console.WriteLine();
      }
   }
}
```

```
Imports System.Text
Imports System.Collections.Generic

Module Module1

    Sub Main()
        Dim enc As Encoding = Encoding.GetEncoding("us-ascii", New CustomMapper(), New
DecoderExceptionFallback())

        Dim str1 As String = String.Format("{0} {1} {2}", ChrW(&H24C8), ChrW(&H2075), ChrW(&H221E))
        Console.WriteLine(str1)
        For ctr As Integer = 0 To str1.Length - 1
            Console.Write("{0} ", Convert.ToUInt16(str1(ctr)).ToString("X4"))
            If ctr = str1.Length - 1 Then Console.WriteLine()
        Next
        Console.WriteLine()

        ' Encode the original string using the ASCII encoder.
        Dim bytes() As Byte = enc.GetBytes(str1)
        Console.Write("Encoded bytes: ")
        For Each byt In bytes
            Console.Write("{0:X2} ", byt)
        Next
        Console.WriteLine()
        Console.WriteLine()

        ' Decode the ASCII bytes.
        Dim str2 As String = enc.GetString(bytes)
        Console.WriteLine("Round-trip: {0}", str1.Equals(str2))
        If Not str1.Equals(str2) Then
            Console.WriteLine(str2)
            For Each ch In str2
                Console.Write("{0} ", Convert.ToUInt16(ch).ToString("X4"))
            Next
            Console.WriteLine()
        End If
    End Sub
End Module
```

请参阅

- .NET 中的字符编码简介
- Encoder
- Decoder
- DecoderFallback
- Encoding
- EncoderFallback
- 全球化和本地化

有关比较 .NET 中字符串的最佳做法

2021/11/16 •

.NET 为开发本地化和全球化应用程序提供广泛支持，在执行排序和显示字符串等常见操作时，轻松应用当前区域性或特定区域性的约定。但排序或比较字符串并不总是区分区域性的操作。例如，对于应用程序内部使用的字符串，通常应该跨所有区域性以相同的方式对其进行处理。如果将 XML 标记、HTML 标记、用户名、文件路径和系统对象名称等与区域性无关的字符串数据解释为区分区域性，则应用程序代码会遭遇细微的错误、不佳的性能，在某些情况下，还会遭遇安全性问题。

本文介绍 .NET 中的字符串排序、比较和大小写方法，针对如何选择适当的字符串处理方法提出建议，并提供有关字符串处理方法的其他信息。

对字符串用法的建议

使用 .NET 进行开发时，请遵循以下简要建议比较字符串：

- 使用为字符串操作显式指定字符串比较规则的重载。通常情况下，这涉及调用具有 StringComparison类型的参数的方法重载。
- 使用 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 进行比较，并以此作为匹配区域性不明确的字符串的安全默认设置。
- 将比较与 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 配合使用，以获得更好的性能。
- 向用户显示输出时，使用基于 StringComparison.CurrentCulture 的字符串操作。
- 当进行与语言（例如，符号）无关的比较时，使用非语言的 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 值，而不使用基于 CultureInfo.InvariantCulture 的字符串操作。
- 在规范化要比较的字符串时，使用 String.ToUpperInvariant 方法而非 String.ToLowerInvariant 方法。
- 使用 String.Equals 方法的重载来测试两个字符串是否相等。
- 使用 String.Compare 和 String.CompareTo 方法可对字符串进行排序，而不是检查字符串是否相等。
- 在用户界面，使用区分区域性的格式显示非字符串数据，如数字和日期。使用格式以固定区域性使非字符串数据显示为字符串形式。

比较字符串时，请避免采用以下做法：

- 不要使用未显式或隐式为字符串操作指定字符串比较规则的重载。
- 在大多数情况下，不要使用基于 StringComparison.InvariantCulture 的字符串操作。其中的一个少数例外情况是，保存在语言上有意义但区域性不明确的数据。
- 不要使用 String.Compare 或 CompareTo 方法的重载和用于确定两个字符串是否相等的返回值为 0 的测试。

显式指定字符串比较

重载 .NET 中大部分字符串操作方法。通常，一个或多个重载会接受默认设置，然而其他重载则不接受默认设置，而是定义比较或操作字符串的精确方式。大多数不依赖于默认设置的方法都包括 StringComparison类型的参数，该参数是按区域性和大小写为字符串比较显式指定规则的枚举。下表描述 StringComparison 枚举成员。

STRINGCOMPARISON 枚	枚
CurrentCulture	使用当前区域性执行区分大小写的比较。
CurrentCultureIgnoreCase	使用当前区域性执行不区分大小写的比较。

STRINGCOMPARISON 枚举	说明
InvariantCulture	使用固定区域性执行区分大小写的比较。
InvariantCultureIgnoreCase	使用固定区域性执行不区分大小写的比较。
Ordinal	执行序号比较。
OrdinalIgnoreCase	执行不区分大小写的序号比较。

例如，IndexOf 方法（它返回 String 对象中与某字符或字符串匹配的子字符串的索引）具有九种重载：

- 默认情况下，IndexOf(Char), IndexOf(Char, Int32)和 IndexOf(Char, Int32, Int32)对字符串中的字符执行序号（区分大小写但不区分区域性的）搜索。
- 默认情况下，IndexOf(String), IndexOf(String, Int32)和 IndexOf(String, Int32, Int32)对字符串中的子字符串执行区分大小写且区分区域性的搜索。
- IndexOf(String, StringComparison)、IndexOf(String, Int32, StringComparison)和 IndexOf(String, Int32, Int32, StringComparison)，其中包括 StringComparison 类型的参数，该类型允许指定比较形式。

我们建议选择不使用默认值的重载，原因如下：

- 具有默认参数的一些重载（在字符串实例中搜索 Char 的重载）执行序号比较，而其他重载（在字符串实例中搜索字符串的重载）执行的是区分区域性的比较。要记住哪种方法使用哪个默认值并非易事，并很容易混淆重载。
- 依赖于方法调用默认值的代码的意图并不清楚。在下面依赖于默认值的示例中，很难了解开发人员对两个字符串的实际意图是执行序号比较还是语言比较，或者 `protocol` 和“http”之间存在的大小写差异是否会导致相等性测试返回 `false` 类型的参数的方法重载。

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http")) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

```
Dim protocol As String = GetProtocol(url)
If String.Equals(protocol, "http") Then
    ' ...Code to handle HTTP protocol.
Else
    Throw New InvalidOperationException()
End If
```

一般情况下，我们建议调用不依赖于默认设置的方法，因为这会明确代码的意图。这进而使代码更具可读性且更易于调试和维护。下面的示例解决了前面示例中提出的问题。使用序号比较并且忽略大小写差异。

```
string protocol = GetProtocol(url);
if (String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase)) {
   // ...Code to handle HTTP protocol.
}
else {
   throw new InvalidOperationException();
}
```

```
Dim protocol As String = GetProtocol(url)
If String.Equals(protocol, "http", StringComparison.OrdinalIgnoreCase) Then
    ' ...Code to handle HTTP protocol.
Else
    Throw New InvalidOperationException()
End If
```

字符串比较的详细信息

字符串比较是许多字符串相关操作的核心，特别是排序和相等性测试操作。字符串以确定的顺序进行排序：如果在排序的字符串列表中，“my”出现在“string”之前，则“my”必定小于或等于“string”。此外，比较可隐式确定相等性。对于认为是相等的字符串，比较操作将返回零。对此很好的解释是两个字符串都不小于对方。涉及到字符串的最有意义的操作包括这些步骤中的一个或两个步骤：与另一个字符串进行比较和执行明确的排序操作。

> **NOTE**
>
> 可以下载排序权重表，这是一组文本文件，其中包含有关 Windows 操作系统排序和比较操作中所使用的字符权重的信息，也可以下载默认 Unicode 排序元素表，这是适用于 Linux 和 macOS 的最新版排序权重表。Linux 和 macOS 上的特定排序权重表版本取决于系统上安装的 International Components for Unicode 库的版本。有关 ICU 版本及它们所实现的 Unicode 版本的信息，请参阅下载 ICU。

但是，评估两个字符串的相等性或排序顺序不会生成一个正确的结果；其结果取决于用于比较这两个字符串的条件。特别是，序号或基于当前区域性或固定区域性（基于英语语言的区域设置不明确的区域性）的大小写和排序约定的字符串比较可能会产生不同的结果。

此外，使用不同 .NET 版本或在不同操作系统或不同的操作系统版本上使用 .NET 进行字符串比较时，返回的结果可能不同。有关详细信息，请参阅字符串和 Unicode 标准。

使用当前区域性的字符串比较

一个条件涉及在比较字符串时使用当前区域性的约定。基于当前区域性的比较使用线程的当前区域性或区域设置。如果用户未设置该区域性，则默认为“控制面板”中“区域选项” 窗口中的设置。当数据与语言相关并反映区分区域性的用户交互时，应始终使用基于当前区域性的比较。

但是，当区域性发生更改时，.NET 中的比较和大小写行为也发生更改。如果执行应用程序的计算机与用于开发该应用程序的计算机具有不同的区域性，或者执行线程改变它的区域性，则会发生这种情况。此行为是有意而为之的，但许多开发人员不易察觉此行为。下面的示例说明了美国英语（“en-US”）与瑞典语（“sv-SE”）区域性在排序顺序中的差异。请注意，单词“ångström”、“Windows”和“Visual Studio”将出现在已排序的字符串数组的不同位置。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] values= { "able", "ångström", "apple", "Æble",
                         "Windows", "Visual Studio" };
      Array.Sort(values);
      DisplayArray(values);

      // Change culture to Swedish (Sweden).
      string originalCulture = CultureInfo.CurrentCulture.Name;
      Thread.CurrentThread.CurrentCulture = new CultureInfo("sv-SE");
      Array.Sort(values);
      DisplayArray(values);

      // Restore the original culture.
      Thread.CurrentThread.CurrentCulture = new CultureInfo(originalCulture);
    }

    private static void DisplayArray(string[] values)
    {
      Console.WriteLine("Sorting using the {0} culture:",
                        CultureInfo.CurrentCulture.Name);
      foreach (string value in values)
         Console.WriteLine("   {0}", value);

      Console.WriteLine();
    }
}
// The example displays the following output:
//       Sorting using the en-US culture:
//          able
//          Æble
//          ångström
//          apple
//          Visual Studio
//          Windows
//
//       Sorting using the sv-SE culture:
//          able
//          Æble
//          apple
//          Windows
//          Visual Studio
//          ångström
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim values() As String = {"able", "ångström", "apple", _
                                  "Æble", "Windows", "Visual Studio"}
        Array.Sort(values)
        DisplayArray(values)

        ' Change culture to Swedish (Sweden).
        Dim originalCulture As String = CultureInfo.CurrentCulture.Name
        Thread.CurrentThread.CurrentCulture = New CultureInfo("sv-SE")
        Array.Sort(values)
        DisplayArray(values)

        ' Restore the original culture.
        Thread.CurrentThread.CurrentCulture = New CultureInfo(originalCulture)
    End Sub

    Private Sub DisplayArray(values() As String)
        Console.WRiteLine("Sorting using the {0} culture:", _
                          CultureInfo.CurrentCulture.Name)
        For Each value As String In values
            Console.WriteLine("   {0}", value)
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'       Sorting using the en-US culture:
'          able
'          Æble
'          ångström
'          apple
'          Visual Studio
'          Windows
'
'       Sorting using the sv-SE culture:
'          able
'          Æble
'          apple
'          Windows
'          Visual Studio
'          ångström
```

使用当前区域性的不区分大小写比较和区分区域性的比较是相同的，只不过前者忽略由线程的当前区域性指示的大小写。这种情况也可表明它的排序顺序。

以下方法默认利用使用当前区域性语义的比较：

- 不包括String.Compare 参数的 StringComparison 重载。
- String.CompareTo 重载。
- 默认 String.StartsWith(String) 方法和具有 String.StartsWith(String, Boolean, CultureInfo) null `null` CultureInfo 重载。
- 默认 String.EndsWith(String) 方法和需要使用 `null` CultureInfo 参数的 String.EndsWith(String, Boolean, CultureInfo) 方法。
- 接受String.IndexOf 作为搜索参数且不包含 String 参数的 StringComparison 重载。
- 接受String.LastIndexOf 作为搜索参数且不包含 String 参数的 StringComparison 重载。

总之，我们建议调用具有 StringComparison 参数的重载，以便明确方法调用的意图。

当从语言角度解释非语言的字符串数据，或利用其他区域性的约定解释某个特定区域性中的字符串时，则会发生

或大或小的错误。土耳其语 I 问题便是一个规范示例。

对于几乎所有拉丁字母来讲(包括美国英语)，字符“i”(\u0069) 是字符“I”(\u0049) 的小写形式。此大小写规则快速成为在此类区域性中编程的人员的默认设置。但是，土耳其语(“tr-TR”)字母表中包含一个“带有点的 I”的字符“İ”(\u0130)，该字符是“i”的大写形式。土耳其语还包括一个小写“不带点的 i”字符，即为“ı”(\u0131)，该字符的大写形式为“I”。阿塞拜疆语(“az”)区域也会出现这种情况。

因此，关于将“i”变为大写或将“I”变为小写的假设并非在所有区域性中都是有效的。如果为字符串比较例程使用默认重载，则它们可能会因区域性不同而异。如果对非语言的数据进行比较，使用默认重载会产生不良后果，如以下对字符串“file”和“FILE”执行不区分大小写的比较尝试所示。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string fileUrl = "file";
      Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                       fileUrl.StartsWith("FILE", true, null));
      Console.WriteLine();

      Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");
      Console.WriteLine("Culture = {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      Console.WriteLine("(file == FILE) = {0}",
                        fileUrl.StartsWith("FILE", true, null));
   }
}
// The example displays the following output:
//       Culture = English (United States)
//       (file == FILE) = True
//
//       Culture = Turkish (Turkey)
//       (file == FILE) = False
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim fileUrl = "file"
        Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
        Console.WriteLine("Culture = {0}", _
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        Console.WriteLine("(file == FILE) = {0}", _
                         fileUrl.StartsWith("FILE", True, Nothing))
        Console.WriteLine()

        Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")
        Console.WriteLine("Culture = {0}", _
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        Console.WriteLine("(file == FILE) = {0}", _
                          fileUrl.StartsWith("FILE", True, Nothing))
    End Sub
End Module
' The example displays the following output:
'       Culture = English (United States)
'       (file == FILE) = True
'
'       Culture = Turkish (Turkey)
'       (file == FILE) = False
```

如果无意中在安全敏感设置中使用了区域性，则此比较会导致发生重大问题，如以下示例所示。如果当前区域性为美国英语，则 `IsFileURI("file:")` 等方法调用将返回 `true`；但如果当前区域性为土耳其语，则将返回 `false`。因此，在土耳其语系统中，有人可能会避开阻止访问以“FILE:”开头的不区分大小写的安全措施。

```
public static bool IsFileURI(String path)
{
   return path.StartsWith("FILE:", true, null);
}
```

```
Public Shared Function IsFileURI(path As String) As Boolean
    Return path.StartsWith("FILE:", True, Nothing)
End Function
```

在这种情况下，由于“file:”会被解释为非语言的、不区分区域性的标识符，因此，应按照下面的示例所示编写代码：

```
public static bool IsFileURI(string path)
{
   return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase);
}
```

```
Public Shared Function IsFileURI(path As String) As Boolean
    Return path.StartsWith("FILE:", StringComparison.OrdinalIgnoreCase)
End Function
```

序号字符串操作

在方法调用中指定 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 值表示非语言比较，这种比较忽略了自然语言的特性。利用 StringComparison 值调用的方法将字符串操作决策建立在简单的字节比较的基础之上，而不是按区域性参数化的大小写或相等表。在大多数情况下，这种方法最符合字符串的预期解释，并使代码更快更可靠。

序号比较就是字符串比较，在这种比较中，将比较每个字符串中的每个字节且不进行语言解释；例如，“windows”不匹配“Windows”。实质上，这是对 C 运行时 `strcmp` 函数的调用。当上下文指示应完全匹配字符串或要求保守匹配策略时，请使用这种比较。此外，序号比较是最快的比较操作，因为它在确定结果时不应用任何语言规则。

.NET 中的字符串可以包括嵌入的空字符。序号比较与区分区域性的比较（包括使用固定区域性的比较）之间最明显的区别之一是对字符串中嵌入的空字符的处理方式。当使用 String.Compare 和 String.Equals 方法执行区分区域性的比较（包括使用固定区域性的比较）时，将忽略这些字符。因此，在区分区域性的比较中，包含嵌入的空字符的字符串可视为等于不包含空字符的字符串。

> **IMPORTANT**
> 尽管字符串比较方法忽略嵌入的空字符，但是 String.Contains、String.EndsWith、String.IndexOf、String.LastIndexOf和 String.StartsWith 等字符串搜索方法并不会忽略这些字符。

下面的示例对字符串“Aa”与在“A”和“a”之间嵌入了多个空字符的相似字符串进行区分区域性的比较，并显示如何将这两个字符串视为相等的字符串：

```
using System;

public class Example
{
   public static void Main()
   {
      string str1 = "Aa";
      string str2 = "A" + new String('\u0000', 3) + "a";
      Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                        str1, ShowBytes(str1), str2, ShowBytes(str2));
      Console.WriteLine("   With String.Compare:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Compare(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Compare(str1, str2, StringComparison.InvariantCulture));

      Console.WriteLine("   With String.Equals:");
      Console.WriteLine("      Current Culture: {0}",
                        String.Equals(str1, str2, StringComparison.CurrentCulture));
      Console.WriteLine("      Invariant Culture: {0}",
                        String.Equals(str1, str2, StringComparison.InvariantCulture));
   }

   private static string ShowBytes(string str)
   {
      string hexString = String.Empty;
      for (int ctr = 0; ctr < str.Length; ctr++)
      {
         string result = String.Empty;
         result = Convert.ToInt32(str[ctr]).ToString("X4");
         result = " " + result.Substring(0,2) + " " + result.Substring(2, 2);
         hexString += result;
      }
      return hexString.Trim();
   }
}
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Current Culture: 0
//          Invariant Culture: 0
//       With String.Equals:
//          Current Culture: True
//          Invariant Culture: True
```

```
Module Example
    Public Sub Main()
        Dim str1 As String = "Aa"
        Dim str2 As String = "A" + New String(Convert.ToChar(0), 3) + "a"
        Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):", _
                          str1, ShowBytes(str1), str2, ShowBytes(str2))
        Console.WriteLine("   With String.Compare:")
        Console.WriteLine("      Current Culture: {0}", _
                          String.Compare(str1, str2, StringComparison.CurrentCulture))
        Console.WriteLine("      Invariant Culture: {0}", _
                          String.Compare(str1, str2, StringComparison.InvariantCulture))

        Console.WriteLine("   With String.Equals:")
        Console.WriteLine("      Current Culture: {0}", _
                          String.Equals(str1, str2, StringComparison.CurrentCulture))
        Console.WriteLine("      Invariant Culture: {0}", _
                          String.Equals(str1, str2, StringComparison.InvariantCulture))
    End Sub

    Private Function ShowBytes(str As String) As String
        Dim hexString As String = String.Empty
        For ctr As Integer = 0 To str.Length - 1
            Dim result As String = String.Empty
            result = Convert.ToInt32(str.Chars(ctr)).ToString("X4")
            result = " " + result.Substring(0, 2) + " " + result.Substring(2, 2)
            hexString += result
        Next
        Return hexString.Trim()
    End Function
End Module
```

但是，当使用序号比较时，这两个字符串不会视为相等，如下面的示例所示：

```
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):",
                  str1, ShowBytes(str1), str2, ShowBytes(str2));
Console.WriteLine("   With String.Compare:");
Console.WriteLine("      Ordinal: {0}",
                  String.Compare(str1, str2, StringComparison.Ordinal));

Console.WriteLine("   With String.Equals:");
Console.WriteLine("      Ordinal: {0}",
                  String.Equals(str1, str2, StringComparison.Ordinal));
// The example displays the following output:
//    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 00 61):
//       With String.Compare:
//          Ordinal: 97
//       With String.Equals:
//          Ordinal: False
```

```
Console.WriteLine("Comparing '{0}' ({1}) and '{2}' ({3}):", _
                  str1, ShowBytes(str1), str2, ShowBytes(str2))
Console.WriteLine("   With String.Compare:")
Console.WriteLine("      Ordinal: {0}", _
                  String.Compare(str1, str2, StringComparison.Ordinal))

Console.WriteLine("   With String.Equals:")
Console.WriteLine("      Ordinal: {0}", _
                  String.Equals(str1, str2, StringComparison.Ordinal))
' The example displays the following output:
'    Comparing 'Aa' (00 41 00 61) and 'A   a' (00 41 00 00 00 00 00 00 61):
'       With String.Compare:
'          Ordinal: 97
'       With String.Equals:
'          Ordinal: False
```

不区分大小写的序号比较是第二种最保守的方法。这些比较会忽略大多数的大小写；例如，“windows”会匹配“Windows”。在处理 ASCII 字符时，此策略等同于 StringComparison.Ordinal，只不过它会忽略常用的 ASCII 大小写。因此，[A, Z] (\u0041-\u005A) 中的任何字符都会匹配 [a,z] (\u0061-\007A) 中的相应字符。超出 ASCII 范围的大小写使用固定区域性的表。因此，下面的比较：

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase);
```

```
String.Compare(strA, strB, StringComparison.OrdinalIgnoreCase)
```

等效于（但会更快）这种比较：

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal);
```

```
String.Compare(strA.ToUpperInvariant(), strB.ToUpperInvariant(),
               StringComparison.Ordinal)
```

这些比较仍非常快。

StringComparison.Ordinal 和 StringComparison.OrdinalIgnoreCase 均直接使用二进制值并最适合匹配。当不确定比较设置时，请使用这两个值中的其中一个。不过，由于它们执行逐字节比较，因此不会按照语言排序顺序（如英语词典）进行排序，而是按照二进制排序顺序。如果向用户显示结果，则在大多数上下文中结果都看上去不正常。

序号语义是不包括 String.Equals 参数（包括相等运算符）的 StringComparison 重载的默认项。总之，我们建议调用具有 StringComparison 参数的重载。

使用固定区域性的字符串操作

具有固定区域性的比较使用由静态 CompareInfo 属性返回的 CultureInfo.InvariantCulture 属性。此行为在所有系统中都相同；它会将其范围外的任何字符转换为其认为等效的固定字符。此策略对于在各个区域性中维护一组字符串行为很有用，但经常产生意外的结果。

具有固定区域性的不区分大小写的比较也使用由静态 CompareInfo 属性返回的静态 CultureInfo.InvariantCulture 属性以获取比较信息。所转换字符中的任何大小写差异都将被忽略。

使用 StringComparison.InvariantCulture 和 StringComparison.Ordinal 的比较对 ASCII 字符串产生相同的作用。但是，StringComparison.InvariantCulture 会做出可能不适用于解释为一组字节的字符串的语言性决策。还可以使用 `CultureInfo.InvariantCulture.CompareInfo` 对象使 Compare 方法将一组特定的字符解释为等效字符。例

如，下面的等效字符在固定区域性中是有效的：

InvariantCulture: a +°= å

如果 A 字符的小写拉丁字母“a”(\u0061) 旁边有上方组合圆圈字符“+ "°”(\u030a)，A 字符就会被解释为，上方带有圆圈的小写拉丁字母“å”(\u00e5)。如下面的示例所示，此行为不同于序号比较。

```
string separated = "\u0061\u030a";
string combined = "\u00e5";

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.InvariantCulture) == 0);

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}",
                  separated, combined,
                  String.Compare(separated, combined,
                                 StringComparison.Ordinal) == 0);
// The example displays the following output:
//    Equal sort weight of a° and å using InvariantCulture: True
//    Equal sort weight of a° and å using Ordinal: False
```

```
Dim separated As String = ChrW(&h61) + ChrW(&h30a)
Dim combined As String = ChrW(&he5)

Console.WriteLine("Equal sort weight of {0} and {1} using InvariantCulture: {2}", _
                  separated, combined, _
                  String.Compare(separated, combined, _
                                 StringComparison.InvariantCulture) = 0)

Console.WriteLine("Equal sort weight of {0} and {1} using Ordinal: {2}", _
                  separated, combined, _
                  String.Compare(separated, combined, _
                                 StringComparison.Ordinal) = 0)
' The example displays the following output:
'    Equal sort weight of a° and å using InvariantCulture: True
'    Equal sort weight of a° and å using Ordinal: False
```

当解释其中出现如“å”组合的文件名称、cookie 或其他内容时，序号比较仍会提供最透明和最合适的行为。

总的来说，固定区域性具有极少的对比较有用的属性。它会以与语言相关的方式执行比较，使其无法保证完整的符号等效性，但它并不是任何区域性中显示的选择。使用 StringComparison.InvariantCulture 进行比较的其中一个原因是为多个区域性相同的显示保留已排序的数据。例如，如果应用程序附带包含用于显示的已排序标识符列表的大型数据文件，则添加到此列表将需要使用固定条件样式排序插入。

为方法调用选择 StringComparison 成员

下表概述了从语义字符串上下文到 StringComparison 枚举成员的映射：

		SYSTEM.STRINGCOMPARISON VALUE
区分大小写的内部标识符。 区分大小写的标准标识符（例如 XML 和 HTTP）。 区分大小写的安全相关设置。	字节完全匹配的非语言标识符。	Ordinal

使用	不使用	使用 System.StringComparison 值
不区分大小写的内部标识符。 不区分大小写的标准标识符(例如 XML 和 HTTP)。 文件路径。 注册表项和值。 环境变量。 资源标识符(例如，句柄名称)。 不区分大小写的安全相关设置。	无关大小写的非语言标识符;尤其是存储在大多数 Windows 系统服务中的数据。	OrdinalIgnoreCase
某些保留的、与语言相关的数据。 需要固定排序顺序的语言数据的显示。	仍与语言相关的区域性不明确数据。	InvariantCulture - 或 - InvariantCultureIgnoreCase
向用户显示的数据。 大多数用户输入。	需要本地语言自定义的数据。	CurrentCulture - 或 - CurrentCultureIgnoreCase

.NET 中的常见字符串比较方法

以下各节介绍最常用于执行字符串比较的方法。

String.Compare

默认解释：StringComparison.CurrentCulture。

作为字符串解释最核心的操作，应根据当前区域性检查这些方法调用的所有实例来确定是否应该从区域性(符号)解释或分离字符串。通常情况下，采用后者，并且应改用 StringComparison.Ordinal 比较。

System.Globalization.CompareInfo 属性返回的 CultureInfo.CompareInfo 类也包括利用 Compare 标记枚举的方式提供大量匹配选项(序号、忽略空白、忽略假名类型等)的 CompareOptions 方法。

String.CompareTo

默认解释：StringComparison.CurrentCulture。

此方法当前不提供指定 StringComparison 类型的重载。通常可以将此方法转换为建议的 String.Compare(String, String, StringComparison) 形式。

实现 IComparable 和 IComparable<T> 接口的类型实现此方法。由于它不提供 StringComparison 参数选项，因此实现类型经常使用户在其构造函数中指定 StringComparer。下面的示例定义 `FileName` 类，其类构造函数包括 StringComparer 参数。然后此 StringComparer 对象将用于 `FileName.CompareTo` 方法。

```
using System;

public class FileName : IComparable
{
   string fname;
   StringComparer comparer;

   public FileName(string name, StringComparer comparer)
   {
      if (String.IsNullOrEmpty(name))
         throw new ArgumentNullException("name");

      this.fname = name;

      if (comparer != null)
         this.comparer = comparer;
      else
         this.comparer = StringComparer.OrdinalIgnoreCase;
   }

   public string Name
   {
      get { return fname; }
   }

   public int CompareTo(object obj)
   {
      if (obj == null) return 1;

      if (! (obj is FileName))
         return comparer.Compare(this.fname, obj.ToString());
      else
         return comparer.Compare(this.fname, ((FileName) obj).Name);
   }
}
```

```
Public Class FileName : Implements IComparable
    Dim fname As String
    Dim comparer As StringComparer

    Public Sub New(name As String, comparer As StringComparer)
        If String.IsNullOrEmpty(name) Then
            Throw New ArgumentNullException("name")
        End If

        Me.fname = name

        If comparer IsNot Nothing Then
            Me.comparer = comparer
        Else
            Me.comparer = StringComparer.OrdinalIgnoreCase
        End If
    End Sub

    Public ReadOnly Property Name As String
        Get
            Return fname
        End Get
    End Property

    Public Function CompareTo(obj As Object) As Integer _
           Implements IComparable.CompareTo
        If obj Is Nothing Then Return 1

        If Not TypeOf obj Is FileName Then
            obj = obj.ToString()
        Else
            obj = CType(obj, FileName).Name
        End If
        Return comparer.Compare(Me.fname, obj)
    End Function
End Class
```

String.Equals

默认解释：StringComparison.Ordinal。

String 类可通过调用静态或实例 Equals 方法重载或使用静态相等运算符，测试是否相等。默认情况下，重载和运算符使用序号比较。但是，我们仍然建议调用显式指定 StringComparison 类型的重载，即使想要执行序号比较；这将更轻松地搜索特定字符串解释的代码。

String.ToUpper 和 String.ToLower

默认解释：StringComparison.CurrentCulture。

应谨慎使用这些方法，因为将字符串强制为大写或小写经常用作在不考虑大小写的情况下比较字符串的较小规范化。如果是这样，请考虑使用不区分大小写的比较。

还可以使用 String.ToUpperInvariant 和 String.ToLowerInvariant 方法。ToUpperInvariant 是规范化大小写的标准方式。使用 StringComparison.OrdinalIgnoreCase 进行的比较在行为上是两个调用的组合：对两个字符串参数调用 ToUpperInvariant，并使用 StringComparison.Ordinal执行比较。

通过向方法传递表示区域性的 CultureInfo 对象，重载也已可用于转换该特性区域性中的大写和小写字母。

Char.ToUpper 和 Char.ToLower

默认解释：StringComparison.CurrentCulture。

这些方法的工作原理类似于上一节中所述的 String.ToUpper 和 String.ToLower 方法。

String.StartsWith 和 String.EndsWith

默认解释：StringComparison.CurrentCulture。

默认情况下，这两种方法执行区分区域性的比较。

String.IndexOf 和 String.LastIndexOf

默认解释：StringComparison.CurrentCulture。

这些方法的默认重载如何执行比较方面缺乏一致性。包含 String.IndexOf 参数的所有 String.LastIndexOf 和 Char 方法都执行序号比较，但是包含 String.IndexOf 参数的默认 String.LastIndexOf 和 String 方法都执行区分区域性的比较。

如果调用 String.IndexOf(String) 或 String.LastIndexOf(String) 方法并向其传递一个字符串以在当前实例中查找，那么我们建议调用显式指定 StringComparison 类型的重载。包括 Char 参数的重载不允许指定 StringComparison 类型。

间接执行字符串比较的方法

将字符串比较作为核心操作的一些非字符串方法使用 StringComparer 类型。StringComparer 类型包含六个返回 StringComparer 实例的静态属性，这些实例的 StringComparer.Compare 方法可执行以下类型的字符串比较：

- 使用当前区域性的区分区域性的字符串比较。此 StringComparer 对象由 StringComparer.CurrentCulture 属性返回。
- 使用当前区域性的不区分区域性的比较。此 StringComparer 对象由 StringComparer.CurrentCultureIgnoreCase 属性返回。
- 使用固定区域性的单词比较规则的不区分区域性的比较。此 StringComparer 对象由 StringComparer.InvariantCulture 属性返回。
- 使用固定区域性的单词比较规则的不区分大小写和不区分区域性的比较。此 StringComparer 对象由 StringComparer.InvariantCultureIgnoreCase 属性返回。
- 序号比较。此 StringComparer 对象由 StringComparer.Ordinal 属性返回。
- 不区分大小写的序号比较。此 StringComparer 对象由 StringComparer.OrdinalIgnoreCase 属性返回。

Array.Sort 和 Array.BinarySearch

默认解释：StringComparison.CurrentCulture。

当在集合中存储任何数据，或将持久数据从文件或数据库中读取到集合中时，切换当前区域性可能会使集合中的固定条件无效。Array.BinarySearch 方法假定已对数组中要搜索的元素排序。若要对数组中的任何字符串元素进行排序，Array.Sort 方法会调用 String.Compare 方法以对各个元素进行排序。如果对数组进行排序和搜索其内容的时间范围内区域性发生变化，那么使用区分区域性的比较器会很危险。例如在下面的代码中，是在由 `Thread.CurrentThread.CurrentCulture` 属性。如果在调用 `StoreNames` 和 `DoesNameExist` 之间更改了区域性（尤其是数组内容保存在两个方法调用之间的某个位置），那么二进制搜索可能会失败。

```
// Incorrect.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names); // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name) >= 0);  // Line B.
}
```

```
' Incorrect.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
    Dim index As Integer = 0
    ReDim storedNames(names.Length - 1)

    For Each name As String In names
        Me.storedNames(index) = name
        index += 1
    Next

    Array.Sort(names)          ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
    Return Array.BinarySearch(Me.storedNames, name) >= 0      ' Line B.
End Function
```

建议的变体将显示在下面使用相同序号(不区分区域性)比较方法进行排序并搜索数组的示例中。在这两个示例中，更改代码会反映在标记 `Line A` 和 `Line B` 的代码行中。

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.Ordinal);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.Ordinal) >= 0);  // Line B.
}
```

```
' Correct.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
    Dim index As Integer = 0
    ReDim storedNames(names.Length - 1)

    For Each name As String In names
        Me.storedNames(index) = name
        index += 1
    Next

    Array.Sort(names, StringComparer.Ordinal)           ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
    Return Array.BinarySearch(Me.storedNames, name, StringComparer.Ordinal) >= 0      ' Line B.
End Function
```

如果此数据永久保留并跨区域性移动，并且使用排序来向用户显示此数据，则可以考虑使用 StringComparison.InvariantCulture，其语言操作可获得更好的用户输出且不受区域性更改的影响。下面的示例修改了前面两个示例，使用固定区域性对数组进行排序和搜索。

```
// Correct.
string []storedNames;

public void StoreNames(string [] names)
{
   int index = 0;
   storedNames = new string[names.Length];

   foreach (string name in names)
   {
      this.storedNames[index++] = name;
   }

   Array.Sort(names, StringComparer.InvariantCulture);  // Line A.
}

public bool DoesNameExist(string name)
{
   return (Array.BinarySearch(this.storedNames, name, StringComparer.InvariantCulture) >= 0);  // Line B.
}
```

```
' Correct.
Dim storedNames() As String

Public Sub StoreNames(names() As String)
    Dim index As Integer = 0
    ReDim storedNames(names.Length - 1)

    For Each name As String In names
        Me.storedNames(index) = name
        index += 1
    Next

    Array.Sort(names, StringComparer.InvariantCulture)           ' Line A.
End Sub

Public Function DoesNameExist(name As String) As Boolean
    Return Array.BinarySearch(Me.storedNames, name, StringComparer.InvariantCulture) >= 0      ' Line B.
End Function
```

集合示例：哈希表构造函数

哈希字符串提供了第二个运算示例，该运算受比较字符串的方式影响。

下面的示例实例化 Hashtable 对象，方法是向其传递由 StringComparer 属性返回的 StringComparer.OrdinalIgnoreCase 对象。由于派生自 StringComparer 的类 StringComparer 实现 IEqualityComparer 接口，其 GetHashCode 方法用于计算哈希表中的字符串的哈希代码。

```
const int initialTableCapacity = 100;
Hashtable h;

public void PopulateFileTable(string directory)
{
   h = new Hashtable(initialTableCapacity,
                     StringComparer.OrdinalIgnoreCase);

   foreach (string file in Directory.GetFiles(directory))
         h.Add(file, File.GetCreationTime(file));
}

public void PrintCreationTime(string targetFile)
{
   Object dt = h[targetFile];
   if (dt != null)
   {
      Console.WriteLine("File {0} was created at time {1}.",
         targetFile,
         (DateTime) dt);
   }
   else
   {
      Console.WriteLine("File {0} does not exist.", targetFile);
   }
}
```

```
Const initialTableCapacity As Integer = 100
Dim h As Hashtable

Public Sub PopulateFileTable(dir As String)
    h = New Hashtable(initialTableCapacity, _
                      StringComparer.OrdinalIgnoreCase)

    For Each filename As String In Directory.GetFiles(dir)
        h.Add(filename, File.GetCreationTime(filename))
    Next
End Sub

Public Sub PrintCreationTime(targetFile As String)
    Dim dt As Object = h(targetFile)
    If dt IsNot Nothing Then
        Console.WriteLine("File {0} was created at {1}.", _
           targetFile, _
           CDate(dt))
    Else
        Console.WriteLine("File {0} does not exist.", targetFile)
    End If
End Sub
```

请参阅

- .NET 应用中的全球化

显示和保存格式化数据的最佳做法

2021/11/16 •

本文讨论如何处理数据格式（如数字数据以及日期和时间数据）以用于显示和存储。

当你使用 .NET 进行开发时，在用户界面中，使用区分区域性的格式显示非字符串数据，如数字和日期。使用格式以固定区域性使非字符串数据显示为字符串形式。不要使用区分区域性格式以字符串形式来保存数值或日期和时间数据。

显示格式化数据

当给用户显示非字符串数据（如数字、日期和时间）时，使用用户的区域性设置来格式化他们。默认情况下，以下所有内容都在格式设置操作中使用当前线程区域性：

- C# 和 Visual Basic 编译器支持的内插字符串。
- 字符串串联操作，它使用 C# 或 Visual Basic 串联运算符或直接调用 String.Concat 方法。
- String.Format 方法。
- 数值类型的 `ToString` 方法以及日期和时间类型。

若要显式指定应使用指定区域性约定或固定区域性设置字符串的格式，可以执行以下操作：

- 当使用 String.Format 和 `ToString` 方法时，调用具有 `provider` 参数（如 String.Format(IFormatProvider, String, Object[]) 或 DateTime.ToString(IFormatProvider)）的重载，并将 CultureInfo.CurrentCulture 属性（表示所需区域性的 CultureInfo 实例）或 CultureInfo.InvariantCulture 属性传递给它。

- 对于字符串串联，不允许编译器执行任何隐式转换。可通过调用具有 `provider` 参数的 `ToString` 重载来执行显式转换。例如，在将 Double 值转换为以下代码中的字符串时，编译器隐式使用当前区域性：

  ```
  string concat1 = "The amount is " + 126.03 + ".";
  Console.WriteLine(concat1);
  ```

  ```
  Dim concat1 As String = "The amount is " & 126.03 & "."
  Console.WriteLine(concat1)
  ```

  可以通过调用 Double.ToString(IFormatProvider) 方法显式指定在转换中使用格式约定的区域性，如下面的代码所示：

  ```
  string concat2 = "The amount is " + 126.03.ToString(CultureInfo.InvariantCulture) + ".";
  Console.WriteLine(concat2);
  ```

  ```
  Dim concat2 As String = "The amount is " & 126.03.ToString(CultureInfo.InvariantCulture) & "."
  Console.WriteLine(concat2)
  ```

- 对于字符串内插，不是将内插字符串分配给 String 实例，而是将其分配给 FormattableString。然后，可以调用其 FormattableString.ToString() 方法生成反映当前区域性约定的结果字符串，也可以调用 FormattableString.ToString(IFormatProvider) 方法生成反映指定区域性约定的结果字符串。还可以将可格式化字符串传递给静态 FormattableString.Invariant 方法，以生成反映固定区域性约定的结果字符串。下面的示例阐释了这种方法。（该示例的输出反映了当前的 zh-CN 区域性。）

```
using System;
using System.Globalization;

class Program
{
    static void Main()
    {
        Decimal value = 126.03m;
        FormattableString amount = $"The amount is {value:C}";
        Console.WriteLine(amount.ToString());
        Console.WriteLine(amount.ToString(new CultureInfo("fr-FR")));
        Console.WriteLine(FormattableString.Invariant(amount));
    }
}
// The example displays the following output:
//    The amount is $126.03
//    The amount is 126,03 €
//    The amount is ¤126.03
```

```
Imports System.Globalization

Module Program
    Sub Main()
        Dim value As Decimal = 126.03
        Dim amount As FormattableString = $"The amount is {value:C}"
        Console.WriteLine(amount.ToString())
        Console.WriteLine(amount.ToString(new CultureInfo("fr-FR")))
        Console.WriteLine(FormattableString.Invariant(amount))
    End Sub
End Module
' The example displays the following output:
'    The amount is $126.03
'    The amount is 126,03 €
'    The amount is ¤126.03
```

保存格式化数据

您可以保留非字符串数据作为二进制数据或作为格式化数据。如果您选择将其保存为格式化数据，您应调用包括 `provider` 参数的格式设置方法重载，并向其传递 CultureInfo.InvariantCulture 属性。固定区域性为独立于区域性和计算机的格式化数据提供一致的格式。相反，使用区域性而非固定区域性进行格式化的持久性数据具有许多限制：

- 如果在具有不同区域性的系统上检索数据，或者如果当前系统用户更改当前区域性或者尝试检索数据时，该数据可能不可用。
- 特定计算机上的区域性属性可能与标准值不同。任何时候，用户都可以自定义区分区域性的显示设置。因此，在系统保存的格式化数据在用户自定义区域性设置之后可能无法读取。格式化数据在计算机之间移植可能会受到更多的限制。
- 管理数值或日期时间格式的国际、区域或国家标准会随着时间发生更改，这些更改会合并到 Windows 操作系统更新中。在格式设置约定更改时，将无法读取使用以前的约定格式化的数据。

下面的示例演示了使用区分区域性格式设置进行持久化数据导致的有限可移植性。该示例将日期和时间数组值保存到文件中。这些数据通过使用英语(美国)区域性约定进行格式化。在应用程序将当前线程区域性更改为法语(瑞士)后，它尝试使用当前区域性的格式设置约定来读取保存的值。尝试读取两个数据条目时引发 FormatException 异常，现在日期数组包含相当于 MinValue的两个错误元素。

```
using System;
using System.Globalization;
using System.IO;
using System.Text;
```

```
using System.Threading;

public class Example
{
   private static string filename = @".\dates.dat";

   public static void Main()
   {
      DateTime[] dates = { new DateTime(1758, 5, 6, 21, 26, 0),
                           new DateTime(1818, 5, 5, 7, 19, 0),
                           new DateTime(1870, 4, 22, 23, 54, 0),
                           new DateTime(1890, 9, 8, 6, 47, 0),
                           new DateTime(1905, 2, 18, 15, 12, 0) };
      // Write the data to a file using the current culture.
      WriteData(dates);
      // Change the current culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-CH");
      // Read the data using the current culture.
      DateTime[] newDates = ReadData();
      foreach (var newDate in newDates)
         Console.WriteLine(newDate.ToString("g"));
   }

   private static void WriteData(DateTime[] dates)
   {
      StreamWriter sw = new StreamWriter(filename, false, Encoding.UTF8);
      for (int ctr = 0; ctr < dates.Length; ctr++) {
         sw.Write("{0}", dates[ctr].ToString("g", CultureInfo.CurrentCulture));
         if (ctr < dates.Length - 1) sw.Write("|");
      }
      sw.Close();
   }

   private static DateTime[] ReadData()
   {
      bool exceptionOccurred = false;

      // Read file contents as a single string, then split it.
      StreamReader sr = new StreamReader(filename, Encoding.UTF8);
      string output = sr.ReadToEnd();
      sr.Close();

      string[] values = output.Split( new char[] { '|' } );
      DateTime[] newDates = new DateTime[values.Length];
      for (int ctr = 0; ctr < values.Length; ctr++) {
         try {
            newDates[ctr] = DateTime.Parse(values[ctr], CultureInfo.CurrentCulture);
         }
         catch (FormatException) {
            Console.WriteLine("Failed to parse {0}", values[ctr]);
            exceptionOccurred = true;
         }
      }
      if (exceptionOccurred) Console.WriteLine();
      return newDates;
   }
}
// The example displays the following output:
//       Failed to parse 4/22/1870 11:54 PM
//       Failed to parse 2/18/1905 3:12 PM
//
//       05.06.1758 21:26
//       05.05.1818 07:19
//       01.01.0001 00:00
//       09.08.1890 06:47
//       01.01.0001 00:00
//       01.01.0001 00:00
```

```
Imports System.Globalization
Imports System.IO
Imports System.Text
Imports System.Threading

Module Example
    Private filename As String = ".\dates.dat"

    Public Sub Main()
        Dim dates() As Date = {#5/6/1758 9:26PM#, #5/5/1818 7:19AM#, _
                               #4/22/1870 11:54PM#, #9/8/1890 6:47AM#, _
                               #2/18/1905 3:12PM#}
        ' Write the data to a file using the current culture.
        WriteData(dates)
        ' Change the current culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-CH")
        ' Read the data using the current culture.
        Dim newDates() As Date = ReadData()
        For Each newDate In newDates
            Console.WriteLine(newDate.ToString("g"))
        Next
    End Sub

    Private Sub WriteData(dates() As Date)
        Dim sw As New StreamWriter(filename, False, Encoding.Utf8)
        For ctr As Integer = 0 To dates.Length - 1
            sw.Write("{0}", dates(ctr).ToString("g", CultureInfo.CurrentCulture))
            If ctr < dates.Length - 1 Then sw.Write("|")
        Next
        sw.Close()
    End Sub

    Private Function ReadData() As Date()
        Dim exceptionOccurred As Boolean = False

        ' Read file contents as a single string, then split it.
        Dim sr As New StreamReader(filename, Encoding.Utf8)
        Dim output As String = sr.ReadToEnd()
        sr.Close()

        Dim values() As String = output.Split({"|"c})
        Dim newDates(values.Length - 1) As Date
        For ctr As Integer = 0 To values.Length - 1
            Try
                newDates(ctr) = DateTime.Parse(values(ctr), CultureInfo.CurrentCulture)
            Catch e As FormatException
                Console.WriteLine("Failed to parse {0}", values(ctr))
                exceptionOccurred = True
            End Try
        Next
        If exceptionOccurred Then Console.WriteLine()
        Return newDates
    End Function
End Module
' The example displays the following output:
'       Failed to parse 4/22/1870 11:54 PM
'       Failed to parse 2/18/1905 3:12 PM
'
'       05.06.1758 21:26
'       05.05.1818 07:19
'       01.01.0001 00:00
'       09.08.1890 06:47
'       01.01.0001 00:00
'       01.01.0001 00:00
'
```

然而，如果你在对 DateTime.ToString(String, IFormatProvider) 和 DateTime.Parse(String, IFormatProvider) 的调用中将 CultureInfo.CurrentCulture 属性替换为 CultureInfo.InvariantCulture，则会成功还原持久的日期和时间数据，如以下输出所示：

```
06.05.1758 21:26
05.05.1818 07:19
22.04.1870 23:54
08.09.1890 06:47
18.02.1905 15:12
```

在 .NET 5 及更高版本中比较字符串时的行为更改

2021/11/16 •

.NET 5.0 引入了一项运行时行为更改，其中，全球化 API 目前在所有支持的平台上默认使用 ICU。这明显有别于较早的 .NET Core 和 .NET Framework 版本，在 Windows 上运行这些版本时，它们利用操作系统的区域语言支持 (NLS) 功能。有关这些更改的详细信息，包括还原该行为更改的兼容性开关，请参阅 .NET 全球化和 ICU。

更改原因

引入此更改是为了统一所有支持的操作系统上的 .NET 的全球化行为。它还能让应用程序捆绑自己的全球化库，而不是依赖于操作系统的内置库。有关详细信息，请参阅中断性变更。

行为差异

如果在使用 `string.IndexOf(string)` 这样的函数时不调用使用 StringComparison 参数的重载，则可能在计划执行序号搜索时无意中依赖于特定于区域性的行为。由于 NLS 和 ICU 在其语言比较器中实现的逻辑有所不同，因此 `string.IndexOf(string)` 等方法的结果可能会返回意外的值。

即使在全球化功能并非总是处于活动状态的地方，也会出现这类问题。例如，根据当前运行时，下面的代码可能会生成不同的答案。

```
string s = "Hello\r\nworld!";
int idx = s.IndexOf("\n");
Console.WriteLine(idx);

// The snippet prints:
//
// '6' when running on .NET Framework (Windows)
// '6' when running on .NET Core 2.x - 3.x (Windows)
// '-1' when running on .NET 5 (Windows)
// '-1' when running on .NET Core 2.x - 3.x or .NET 5 (non-Windows)
// '6' when running on .NET Core 2.x or .NET 5 (in invariant mode)
```

防范意外行为

本节提供了处理 .NET 5.0 中的意外行为更改的两种方法。

启用代码分析器

代码分析器可以检测可能存在错误的调用站点。为了帮助防范任何意外行为，建议在项目中启用 .NET Compiler Platform (Roslyn) 分析器。该分析器有助于标记在计划使用序号比较器时可能无意中使用语言比较器的代码。以下规则应有助于标记这些问题：

- CA1307:为了清晰起见，请指定 StringComparison
- CA1309:使用按顺序的 StringComparison
- CA1310：为了确保正确，请指定 StringComparison

默认不启用这些特定规则。若要启用它们并将任何冲突显示为生成错误，请在项目文件中设置以下属性：

```
<PropertyGroup>
  <AnalysisMode>AllEnabledByDefault</AnalysisMode>
  <WarningsAsErrors>$(WarningsAsErrors);CA1307;CA1309;CA1310</WarningsAsErrors>
</PropertyGroup>
```

以下代码片段显示生成相关代码分析器警告或错误的代码示例。

```
//
// Potentially incorrect code - answer might vary based on locale.
//
string s = GetString();
// Produces analyzer warning CA1310 for string; CA1307 matches on char ','
int idx = s.IndexOf(",");
Console.WriteLine(idx);

//
// Corrected code - matches the literal substring ",".
//
string s = GetString();
int idx = s.IndexOf(",", StringComparison.Ordinal);
Console.WriteLine(idx);

//
// Corrected code (alternative) - searches for the literal ',' character.
//
string s = GetString();
int idx = s.IndexOf(',');
Console.WriteLine(idx);
```

同样，在实例化已排序的字符串集合或对现有基于字符串的集合进行排序时，请指定显式比较器。

```
//
// Potentially incorrect code - behavior might vary based on locale.
//
SortedSet<string> mySet = new SortedSet<string>();
List<string> list = GetListOfStrings();
list.Sort();

//
// Corrected code - uses ordinal sorting; doesn't vary by locale.
//
SortedSet<string> mySet = new SortedSet<string>(StringComparer.Ordinal);
List<string> list = GetListOfStrings();
list.Sort(StringComparer.Ordinal);
```

还原到 NLS 行为

在 Windows 上运行 .NET 5 应用程序时，若要将其还原到早前的 NLS 行为，请按照 .NET 全球化和 ICU 中的步骤操作。必须在应用程序级别设置此应用程序范围的兼容性开关。单个库不能选择加入或选择退出此行为。

> **TIP**
> 强烈建议启用 CA1307、CA1309 和 CA1310 代码分析规则，以帮助改进代码卫生和发现任何现有的潜在 bug。有关详细信息，请参阅启用代码分析器。

受影响的 API

大多数 .NET 应用程序不会遇到因 .NET 5.0 中的更改而导致的任何意外行为。但是，由于受影响的 API 数量以及这些 API 对更广泛的 .NET 生态系统的基础性，你应知道 .NET 5.0 可能会引入不需要的行为或公开应用程序中已

存在的潜在 bug。

受影响的 API 包括：

- System.String.Compare
- System.String.EndsWith
- System.String.IndexOf
- System.String.StartsWith
- System.String.ToLower
- System.String.ToLowerInvariant
- System.String.ToUpper
- System.String.ToUpperInvariant
- System.Globalization.TextInfo（大多数成员）
- System.Globalization.CompareInfo（大多数成员）
- System.Array.Sort（对字符串数组进行排序时）
- System.Collections.Generic.List<T>.Sort()（当列表元素为字符串时）
- System.Collections.Generic.SortedDictionary<TKey,TValue>（当键为字符串时）
- System.Collections.Generic.SortedList<TKey,TValue>（当键为字符串时）
- System.Collections.Generic.SortedSet<T>（当集包含字符串时）

> **NOTE**
> 这不是受影响的 API 的详尽列表。

默认情况下，上述所有 API 都使用语言字符串搜索与比较，它们使用线程的当前区域性。序号和语言搜索与比较中指明了语言和序号搜索与比较之间的区别。

由于 ICU 实现语言字符串比较的方式与 NLS 不同，从早期版本的 .NET Core 或 .NET Framework 升级到 .NET 5.0 的基于 Windows 的应用程序，在调用受影响的 API 之一时可能会注意到，这些 API 开始表现出不同的行为。

异常

- 如果 API 接受显式 `StringComparison` 或 `CultureInfo` 参数，则该参数将覆盖 API 的默认行为。
- 第一个参数类型为 `char` 的 `System.String` 成员（例如 String.IndexOf(Char)）使用序号搜索，除非调用方传递指定 `CurrentCulture[IgnoreCase]` 或 `InvariantCulture[IgnoreCase]` 的显式 `StringComparison` 参数。

若要详细分析每个 String API 的默认行为，请参阅默认搜索和比较类型部分。

序号和语言搜索与比较

序号（也称为"非语言"）搜索与比较将字符串拆分为其单独的 `char` 元素，并执行逐字符搜索或比较。例如，字符串 `"dog"` 和 `"dog"` 在 `Ordinal` 比较器下的比较结果为"相等"，因为这两个字符串由完全相同的字符序列组成。但是，`"dog"` 和 `"Dog"` 在 `Ordinal` 比较器下的比较结果为"不相等"，因为它们由不完全相同的字符序列组成。也就是说，大写 `'D'` 的码位 `U+0044` 出现在小写 `'d'` 的码位 `U+0064` 之前，因此 `"dog"` 排在 `"Dog"` 之前。

`OrdinalIgnoreCase` 比较器仍执行逐字符操作，但它在执行该操作时消除了大小写差异。在 `OrdinalIgnoreCase` 比较器下，字符对 `'d'` 和 `'D'` 的比较结果为"相等"，字符对 `'á'` 和 `'Á'` 亦是如此。但非重音字符 `'a'` 与重音字符 `'á'` 的比较结果为"不相等"。

下表提供了此操作的一些示例：

字符串 1	字符串 2	`ORDINAL` 结果	`ORDINALIGNORECASE` 结果
`"dog"`	`"dog"`	equal	equal
`"dog"`	`"Dog"`	不等于	equal
`"resume"`	`"Resume"`	不等于	equal
`"resume"`	`"résumé"`	不等于	不等于

Unicode 还允许字符串具有多个不同的内存中表示形式。例如，带锐音符的 e (é) 可以用两种方式表示：

- 单个文本 `'é'` 字符（亦可写为 `'\u00E9'`）。
- 文本无重音 `'e'` 字符，后跟一个组合用重音修饰符字符 `'\u0301'`。

这意味着下面的四个字符串的显示结果都为 `"résumé"`，即使它们的组成部分有所不同。该字符串使用文本 `'é'` 字符或文本无重音 `'e'` 字符，以及组合用重音修饰符 `'\u0301'`。

- `"r\u00E9sum\u00E9"`
- `"r\u00E9sume\u0301"`
- `"re\u0301sum\u00E9"`
- `"re\u0301sume\u0301"`

在序号比较器下，这些字符串相互比较后的结果都不是“相等”。这是因为它们都包含不同的基础字符序列，即使在屏幕上显示时，它们看起来都是相同的。

执行 `string.IndexOf(..., StringComparison.Ordinal)` 操作时，运行时查找完全匹配的子字符串。结果如下所示：

```
Console.WriteLine("resume".IndexOf("e", StringComparison.Ordinal)); // prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("e", StringComparison.Ordinal)); // prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf("e", StringComparison.Ordinal)); // prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf("e", StringComparison.Ordinal)); // prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf("e", StringComparison.Ordinal)); // prints '1'
Console.WriteLine("resume".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '-1'
Console.WriteLine("r\u00E9sume\u0301".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '5'
Console.WriteLine("re\u0301sum\u00E9".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
Console.WriteLine("re\u0301sume\u0301".IndexOf("E", StringComparison.OrdinalIgnoreCase)); // prints '1'
```

序号搜索与比较例程从来不受当前线程的区域性设置的影响。

语言搜索与比较例程将字符串拆分为排序元素，并对这些元素执行搜索或比较。字符串的字符和其构成的排序元素之间不一定存在 1:1 映射。例如，长度为 2 的字符串可能只包含单个排序元素。使用识别语言的方式比较两个字符串时，比较器会检查两个字符串的排序元素是否具有相同的语义含义，即使这两个字符串的文本字符并不相同。

再次考虑字符串 `"résumé"` 及其四种不同的表示形式。下表展示了拆分为其排序元素的每种表示形式。

STRING	排序元素
`"r\u00E9sum\u00E9"`	`"r" + "\u00E9" + "s" + "u" + "m" + "\u00E9"`
`"r\u00E9sume\u0301"`	`"r" + "\u00E9" + "s" + "u" + "m" + "e\u0301"`
`"re\u0301sum\u00E9"`	`"r" + "e\u0301" + "s" + "u" + "m" + "\u00E9"`

STRING	ıccccc
`"re\u0301sume\u0301"`	`"r" + "e\u00E9" + "s" + "u" + "m" + "e\u0301"`

排序元素松散地对应于读取器视为单个字符或字符群集的字符串。它在概念上类似于字形群集，但包含更大的字符串。

在语言比较器下，不需要完全匹配。排序元素根据其语义含义进行比较。例如，语言比较器对子字符串 `"\u00E9"` 和 `"e\u0301"` 的比较结果为"相等"，因为它们在语义上都是指"带锐音符修饰符的小写字母 e"。这允许 `IndexOf` 方法匹配更大字符串中的子字符串 `"e\u0301"`，该更大字符串包含语义上等效的子字符串 `"\u00E9"`，如下面的代码示例中所示。

```
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("e")); // prints '-1' (not found)
Console.WriteLine("r\u00E9sum\u00E9".IndexOf("e\u00E9")); // prints '1'
Console.WriteLine("\u00E9".IndexOf("e\u00E9")); // prints '0'
```

因此，如果使用语言比较，则两个不同长度的字符串的比较结果可能为"相等"。调用方应注意，在这种情况下处理字符串长度时，不要使用特殊情况逻辑。

识别区域性搜索与比较例程是语言搜索与比较例程的一种特殊形式。在识别区域性比较器下，排序元素的概念扩展为包含特定于指定区域性的信息。

例如，在匈牙利字母中，如果两个字符 <dz> 连续出现，则它们被认为是与 <d> 或 <z> 不同的独特字母。这意味着，如果在字符串中出现 <dz>，则匈牙利语识别区域性比较器会将其视为单个排序元素。

STRING	ıccccc	ıı
`"endz"`	`"e" + "n" + "d" + "z"`	（使用标准语言比较器）
`"endz"`	`"e" + "n" + "dz"`	（使用匈牙利语识别区域性比较器）

使用匈牙利语识别区域性比较器时，字符串 `"endz"` 不以子字符串 `"z"` 结尾，因为 <\dz> 和 <\z> 被视为具有不同语义含义的排序元素。

```
// Set thread culture to Hungarian
CultureInfo.CurrentCulture = CultureInfo.GetCultureInfo("hu-HU");
Console.WriteLine("endz".EndsWith("z")); // Prints 'False'

// Set thread culture to invariant culture
CultureInfo.CurrentCulture = CultureInfo.InvariantCulture;
Console.WriteLine("endz".EndsWith("z")); // Prints 'True'
```

NOTE

- 行为：语言和识别区域性比较器可以不时地进行行为调整。ICU 和较旧的 Windows NLS 功能都将更新，以考虑世界语言的变化。有关详细信息，请参阅博客文章区域设置（区域性）数据改动。序号比较器的行为将永远不会发生更改，因为它执行严格的按位搜索和比较。但是，OrdinalIgnoreCase 比较器的行为可能会随着 Unicode 的增加而改变，以包含更多的字符集，并纠正现有大小写数据中的遗漏。
- 使用情况：比较器 `StringComparison.InvariantCulture` 和 `StringComparison.InvariantCultureIgnoreCase` 是不能识别区域性的语言比较器。也就是说，这些比较器理解一些概念，例如，重音字符 é 具有多种可能的基础表示形式，且所有这些表示形式都应视为"相等"。但不识别区域性的语言比较器不包含对 <dz> 区别于 <d> 或 <z> 的特殊处理，如上所示。它们也不能处理像德语 Eszett (ß) 这样的特殊字符。

.NET 还提供固定全球化模式。此选择加入模式禁用处理语言搜索与比较例程的代码路径。在此模式下，无论调

用方提供何种 `CultureInfo` 或 `StringComparison` 参数，所有操作都使用序号或 OrdinalIgnoreCase 行为。有关详细信息，请参阅全球化运行时配置选项和 .NET Core 全球化固定模式。

有关详细信息，请参阅比较 .NET 中的字符串的最佳做法。

安全隐患

如果你的应用使用受影响的 API 进行筛选，我们建议启用 CA1307 和 CA1309 代码分析规则，以帮助查找可能无意中使用了语言搜索而不是序号搜索的位置。下面这样的代码模式可能易受安全漏洞的攻击。

```
//
// THIS SAMPLE CODE IS INCORRECT.
// DO NOT USE IT IN PRODUCTION.
//
public bool ContainsHtmlSensitiveCharacters(string input)
{
    if (input.IndexOf("<") >= 0) { return true; }
    if (input.IndexOf("&") >= 0) { return true; }
    return false;
}
```

由于 `string.IndexOf(string)` 方法默认使用语言搜索，因此字符串可能包含文本 `'<'` 或 `'&'` 字符，且 `string.IndexOf(string)` 例程可能返回 `-1`，表示找不到搜索子字符串。代码分析规则 CA1307 和 CA1309 标记此类调用站点，并警告开发人员存在潜在的问题。

默认搜索和比较类型

下表列出了各种字符串和类似于字符串的 API 的默认搜索和比较类型。如果调用方提供显式 `CultureInfo` 或 `StringComparison` 参数，则该参数将优先于任何默认值。

API	[illegible]	[illegible]
`string.Compare`	CurrentCulture	
`string.CompareTo`	CurrentCulture	
`string.Contains`	Ordinal	
`string.EndsWith`	Ordinal	（当第一个参数为 `char` 时）
`string.EndsWith`	CurrentCulture	（当第一个参数为 `string` 时）
`string.Equals`	Ordinal	
`string.GetHashCode`	Ordinal	
`string.IndexOf`	Ordinal	（当第一个参数为 `char` 时）
`string.IndexOf`	CurrentCulture	（当第一个参数为 `string` 时）
`string.IndexOfAny`	Ordinal	
`string.LastIndexOf`	Ordinal	（当第一个参数为 `char` 时）

API	[illegible]	[illegible]
`string.LastIndexOf`	CurrentCulture	（当第一个参数为 `string` 时）
`string.LastIndexOfAny`	Ordinal	
`string.Replace`	Ordinal	
`string.Split`	Ordinal	
`string.StartsWith`	Ordinal	（当第一个参数为 `char` 时）
`string.StartsWith`	CurrentCulture	（当第一个参数为 `string` 时）
`string.ToLower`	CurrentCulture	
`string.ToLowerInvariant`	InvariantCulture	
`string.ToUpper`	CurrentCulture	
`string.ToUpperInvariant`	InvariantCulture	
`string.Trim`	Ordinal	
`string.TrimEnd`	Ordinal	
`string.TrimStart`	Ordinal	
`string == string`	Ordinal	
`string != string`	Ordinal	

与 `string` API 不同，默认情况下，所有 `MemoryExtensions` API 都执行序号搜索与比较，但以下情况例外。

API	[illegible]	[illegible]
`MemoryExtensions.ToLower`	CurrentCulture	（当传递 null `CultureInfo` 参数时）
`MemoryExtensions.ToLowerInvariant`	InvariantCulture	
`MemoryExtensions.ToUpper`	CurrentCulture	（当传递 null `CultureInfo` 参数时）
`MemoryExtensions.ToUpperInvariant`	InvariantCulture	

结果是，在将代码从使用 `string` 转换为使用 `ReadOnlySpan<char>` 时，可能会无意中引入行为更改。相关示例如下。

```
string str = GetString();
if (str.StartsWith("Hello")) { /* do something */ } // this is a CULTURE-AWARE (linguistic) comparison

ReadOnlySpan<char> span = s.AsSpan();
if (span.StartsWith("Hello")) { /* do something */ } // this is an ORDINAL (non-linguistic) comparison
```

解决此问题的建议方法是将显式 `StringComparison` 参数传递给这些 API。代码分析规则 CA1307 和 CA1309 可帮助解决此问题。

```
string str = GetString();
if (str.StartsWith("Hello", StringComparison.Ordinal)) { /* do something */ } // ordinal comparison

ReadOnlySpan<char> span = s.AsSpan();
if (span.StartsWith("Hello", StringComparison.Ordinal)) { /* do something */ } // ordinal comparison
```

请参阅

- 全球化中断性变更
- 有关比较 .NET 中字符串的最佳做法
- 如何比较 C# 中的字符串
- .NET 全球化和 ICU
- 序号与识别区域性字符串操作
- .NET 源代码分析概述

.NET 中的基本字符串操作

2021/11/16 •

应用程序经常通过构造基于用户输入的消息来响应用户。例如，网站用包含用户名的专用问候语来响应新登录的用户的情况并不少见。

使用 System.String 和 System.Text.StringBuilder 类中的多个方法，可以动态构造要在用户界面中显示的自定义字符串。借助这些方法还可执行许多基本字符串操作，例如，从字节数组创建新字符串，比较字符串的值和修改现有字符串。

相关章节

.NET 中的类型转换
介绍了如何从一种类型转换为另一种类型。

格式设置类型
介绍了如何使用格式说明符设置字符串格式。

新建 .NET 中的字符串

2021/11/16 •

.NET 允许通过简单赋值创建字符串，并且还重载一个类构造函数，以支持使用一些不同参数来创建字符串。.NET 还在 System.String 类中提供了多个方法，可通过合并多个字符串、字符串数组或对象来新建字符串对象。

通过赋值创建字符串

新建 String 对象的最简单方法是，将字符串文本分配给 String 对象。

使用类构造函数创建字符串

可以重载 String 类构造函数，通过字符数组创建字符串。还可以通过将特定字符重复指定次数来创建新字符串。

返回字符串的方法

下表列出了返回新字符串对象的几个有用方法。

tttt	tt
String.Format	从一组输入对象生成格式化的字符串。
String.Concat	从两个或更多个字符串生成字符串。
String.Join	通过合并字符串数组生成新字符串。
String.Insert	通过将一个字符串插入现有字符串的指定索引处生成新字符串。
String.CopyTo	将一个字符串中的指定字符复制到一个字符数组中的指定位置。

格式

可以使用 String.Format 方法，创建格式化字符串，并连接表示多个对象的字符串。此方法自动将传递给它的任何对象转换为字符串。例如，如果应用必须向用户显示 Int32 值和 DateTime 值，可以使用 Format 方法，轻松构造表示这些值的字符串。有关此方法使用的格式化约定的信息，请参阅有关复合格式化的部分。

下面的示例使用 Format 方法，创建使用整数变量的字符串。

```
int numberOfFleas = 12;
string miscInfo = String.Format("Your dog has {0} fleas. " +
                                "It is time to get a flea collar. " +
                                "The current universal date is: {1:u}.",
                                numberOfFleas, DateTime.Now);
Console.WriteLine(miscInfo);
// The example displays the following output:
//       Your dog has 12 fleas. It is time to get a flea collar.
//       The current universal date is: 2008-03-28 13:31:40Z.
```

```
Dim numberOfFleas As Integer = 12
Dim miscInfo As String = String.Format("Your dog has {0} fleas. " & _
                                       "It is time to get a flea collar. " & _
                                       "The current universal date is: {1:u}.", _
                                       numberOfFleas, Date.Now)
Console.WriteLine(miscInfo)
' The example displays the following output:
'       Your dog has 12 fleas. It is time to get a flea collar.
'       The current universal date is: 2008-03-28 13:31:40Z.
```

在此示例中，DateTime.Now 按照与当前线程关联的区域性指定的方式，显示当前日期和时间。

Concat

使用 String.Concat 方法，可以通过两个或多个现有对象轻松新建字符串对象。它提供了一种与语言无关的方法来连接字符串。此方法接受派生自 System.Object 的任何类。下面的示例使用两个现有字符串对象和一个分隔符创建一个字符串。

```
string helloString1 = "Hello";
string helloString2 = "World!";
Console.WriteLine(String.Concat(helloString1, ' ', helloString2));
// The example displays the following output:
//      Hello World!
```

```
Dim helloString1 As String = "Hello"
Dim helloString2 As String = "World!"
Console.WriteLine(String.Concat(helloString1, " "c, helloString2))
' The example displays the following output:
'      Hello World!
```

联接

String.Join 方法通过字符串数组和分隔符字符串新建字符串。如果想要将多个字符串连接在一起，构成一个可能由逗号分隔的列表，则此方法非常有用。

下面的示例使用空格来连接字符串数组。

```
string[] words = {"Hello", "and", "welcome", "to", "my" , "world!"};
Console.WriteLine(String.Join(" ", words));
// The example displays the following output:
//      Hello and welcome to my world!
```

```
Dim words() As String = {"Hello", "and", "welcome", "to", "my", "world!"}
Console.WriteLine(String.Join(" ", words))
' The example displays the following output:
'      Hello and welcome to my world!
```

Insert

String.Insert 方法通过将一个字符串插入另一个字符串中的指定位置来新建字符串。此方法使用从零开始的索引。下面的示例将一个字符串插入 `MyString` 的第五个索引位置，并用此值创建新字符串。

```
string sentence = "Once a time.";
 Console.WriteLine(sentence.Insert(4, " upon"));
 // The example displays the following output:
 //      Once upon a time.
```

```
Dim sentence As String = "Once a time."
Console.WriteLine(sentence.Insert(4, " upon"))
' The example displays the following output:
'      Once upon a time.
```

CopyTo

String.CopyTo 方法将字符串的某些部分复制到字符数组中。可以同时指定字符串的开始索引和要复制的字符数。此方法采用源索引、字符数组、目标索引和要复制的字符数。所有索引都从零开始。

下面的示例使用 CopyTo 方法，将字符串对象中的"Hello"字词字符复制到字符数组的第一个索引位置。

```
string greeting = "Hello World!";
char[] charArray = {'W','h','e','r','e'};
Console.WriteLine("The original character array: {0}", new string(charArray));
greeting.CopyTo(0, charArray,0 ,5);
Console.WriteLine("The new character array: {0}", new string(charArray));
// The example displays the following output:
//       The original character array: Where
//       The new character array: Hello
```

```
Dim greeting As String = "Hello World!"
Dim charArray() As Char = {"W"c, "h"c, "e"c, "r"c, "e"c}
Console.WriteLine("The original character array: {0}", New String(charArray))
greeting.CopyTo(0, charArray, 0, 5)
Console.WriteLine("The new character array: {0}", New String(charArray))
' The example displays the following output:
'       The original character array: Where
'       The new character array: Hello
```

请参阅

- 基本字符串操作
- 复合格式设置

修整和删除 .NET 中的字符串字符

2021/11/16 •

如果将句子分析成单个单词，最后得到的结果可能是任意一端带有空白（也称为空格）的单词。在这种情形下，可以使用 System.String 类中的其中一种剪裁方法，从字符串中的指定位置移除任何数量的空格或其他字符。下表描述了可用的剪裁方法。

“””	“”
String.Trim	从字符串的开头和结尾移除空白或者指定的字符
String.TrimEnd	从字符串的结尾移除在字符数组中指定的字符。
String.TrimStart	从字符串的开头移除在字符数组中指定的字符。
String.Remove	从字符串中的指定索引位置移除指定数量的字符。

Trim

通过使用 String.Trim 方法，可以方便地从字符串两端移除空白，如下面的示例所示。

```
String^ MyString = " Big   ";
Console::WriteLine("Hello{0}World!", MyString);
String^ TrimString = MyString->Trim();
Console::WriteLine("Hello{0}World!", TrimString);
// The example displays the following output:
//       Hello Big   World!
//       HelloBigWorld!
```

```
string MyString = " Big   ";
Console.WriteLine("Hello{0}World!", MyString);
string TrimString = MyString.Trim();
Console.WriteLine("Hello{0}World!", TrimString);
//       The example displays the following output:
//             Hello Big   World!
//             HelloBigWorld!
```

```
Dim MyString As String = " Big   "
Console.WriteLine("Hello{0}World!", MyString)
Dim TrimString As String = MyString.Trim()
Console.WriteLine("Hello{0}World!", TrimString)
' The example displays the following output:
'       Hello Big   World!
'       HelloBigWorld!
```

还可以从字符串的开始和结尾移除指定的字符。下面的示例将移除空白字符、句点和星号。

```
using System;

public class Example
{
   public static void Main()
   {
      String header = "* A Short String. *";
      Console.WriteLine(header);
      Console.WriteLine(header.Trim( new Char[] { ' ', '*', '.' } ));
   }
}
// The example displays the following output:
//       * A Short String. *
//       A Short String
```

```
Module Example
    Public Sub Main()
        Dim header As String = "* A Short String. *"
        Console.WriteLine(header)
        Console.WriteLine(header.Trim({" "c, "*"c, "."c}))
    End Sub
End Module
' The example displays the following output:
'       * A Short String. *
'       A Short String
```

TrimEnd

String.TrimEnd 方法从字符串的结尾移除字符，同时创建新的字符串对象。通过向此方法传递一个字符数组来指定要移除的字符。字符数组中的元素顺序并不影响剪裁操作。找到未在数组中指定的字符时，剪裁停止。

下面的示例使用 TrimEnd 方法，删除字符串的最后几个字母。在此示例中，`'r'` 字符和 `'W'` 字符的位置交换，说明数组中字符的顺序并不重要。请注意，此代码移除 `MyString` 的最后一个单词和第一个单词的一部分。

```
String^ MyString = "Hello World!";
array<Char>^ MyChar = {'r','o','W','l','d','!',' '};
String^ NewString = MyString->TrimEnd(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello World!"
Dim MyChar() As Char = {"r", "o", "W", "l", "d", "!", " "}
Dim NewString As String = MyString.TrimEnd(MyChar)
Console.WriteLine(NewString)
```

此代码向控制台显示 `He`。

下面的示例使用 TrimEnd 方法移除字符串的最后一个单词。在此代码中，单词 `Hello` 后跟一个逗号，而由于在要剪裁的字符数组中未指定逗号，因此剪裁在逗号处结束。

```
String^ MyString = "Hello, World!";
array<Char>^ MyChar = {'r','o','W','l','d','!',' '};
String^ NewString = MyString->TrimEnd(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello, World!";
char[] MyChar = {'r','o','W','l','d','!',' '};
string NewString = MyString.TrimEnd(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello, World!"
Dim MyChar() As Char = {"r", "o", "W", "l", "d", "!", " "}
Dim NewString As String = MyString.TrimEnd(MyChar)
Console.WriteLine(NewString)
```

此代码向控制台显示 `Hello,`。

TrimStart

String.TrimStart 方法类似于 String.TrimEnd 方法，不同之处在于它通过从现有字符串对象的开头移除字符来创建新的字符串。通过向 TrimStart 方法传递一个字符数组来指定要移除的字符。与 TrimEnd 方法一样，字符数组中元素的顺序并不影响剪裁操作。找到未在数组中指定的字符时，剪裁停止。

下面的示例移除字符串的第一个单词。在此示例中，`'l'` 字符和 `'H'` 字符的位置交换，说明数组中字符的顺序并不重要。

```
String^ MyString = "Hello World!";
array<Char>^ MyChar = {'e', 'H','l','o',' ' };
String^ NewString = MyString->TrimStart(MyChar);
Console::WriteLine(NewString);
```

```
string MyString = "Hello World!";
char[] MyChar = {'e', 'H','l','o',' ' };
string NewString = MyString.TrimStart(MyChar);
Console.WriteLine(NewString);
```

```
Dim MyString As String = "Hello World!"
Dim MyChar() As Char = {"e", "H", "l", "o", " "}
Dim NewString As String = MyString.TrimStart(MyChar)
Console.WriteLine(NewString)
```

此代码向控制台显示 `World!`。

删除

String.Remove 方法从现有字符串的指定位置开始移除指定数量的字符。此方法采用从零开始的索引。

下面的示例在字符串从零开始的索引中第五个位置开始移除十个字符。

```
String^ MyString = "Hello Beautiful World!";
Console::WriteLine(MyString->Remove(5,10));
// The example displays the following output:
//         Hello World!
```

```
string MyString = "Hello Beautiful World!";
Console.WriteLine(MyString.Remove(5,10));
// The example displays the following output:
//         Hello World!
```

```
Dim MyString As String = "Hello Beautiful World!"
Console.WriteLine(MyString.Remove(5, 10))
' The example displays the following output:
'         Hello World!
```

替换

通过调用 String.Replace(String, String) 方法并指定空字符串 (String.Empty) 作为替代，也可以从字符串中移除指定字符或子字符串。下面的示例从字符串中移除所有逗号。

```
using System;

public class Example
{
   public static void Main()
   {
      String phrase = "a cold, dark night";
      Console.WriteLine("Before: {0}", phrase);
      phrase = phrase.Replace(",", "");
      Console.WriteLine("After: {0}", phrase);
   }
}
// The example displays the following output:
//       Before: a cold, dark night
//       After: a cold dark night
```

```
Module Example
    Public Sub Main()
        Dim phrase As String = "a cold, dark night"
        Console.WriteLine("Before: {0}", phrase)
        phrase = phrase.Replace(",", "")
        Console.WriteLine("After: {0}", phrase)
    End Sub
End Module
' The example displays the following output:
'       Before: a cold, dark night
'       After: a cold dark night
```

请参阅

- 基本字符串操作

填充 .NET 中的字符串

2021/11/16 •

使用下面的 String 方法之一，在原始字符串中填充前导或尾随字符，以达到指定总长度，从而新建字符串。填充字符可以是空格或指定字符。生成的字符串可能显示为右对齐或左对齐。如果原始字符串的长度已经等于或大于所需总长度，则填充方法返回未经更改的原始字符串；有关详细信息，请参阅 String.PadLeft 和 String.PadRight 方法的两个重载的"返回"部分。

String.PadLeft	使用前导字符将字符串填充到指定总长度。
String.PadRight	使用尾随字符将字符串填充到指定总长度。

PadLeft

String.PadLeft 方法将足够多的前导填充字符连接到原始字符串，以达到指定总长度，从而新建字符串。String.PadLeft(Int32) 方法使用空格作为填充字符，String.PadLeft(Int32, Char) 方法支持指定自己的填充字符。

下面的代码示例使用 PadLeft 方法新建长度为 20 个字符的字符串。该示例向控制台显示"`--------Hello World!`"。

```
String^ MyString = "Hello World!";
Console::WriteLine(MyString->PadLeft(20, '-'));
```

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadLeft(20, '-'));
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.PadLeft(20, "-"c))
```

PadRight

String.PadRight 方法将足够多的尾随填充字符连接到原始字符串，以达到指定总长度，从而新建字符串。String.PadRight(Int32) 方法使用空格作为填充字符，String.PadRight(Int32, Char) 方法支持指定自己的填充字符。

下面的代码示例使用 PadRight 方法新建长度为 20 个字符的字符串。该示例向控制台显示"`Hello World!--------`"。

```
String^ MyString = "Hello World!";
Console::WriteLine(MyString->PadRight(20, '-'));
```

```
string MyString = "Hello World!";
Console.WriteLine(MyString.PadRight(20, '-'));
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.PadRight(20, "-"c))
```

请参阅

- 基本字符串操作

比较 .NET 中的字符串

2021/11/16 •

.NET 提供几种方法来比较字符串的值。下表列出和描述值比较方法。

String.Compare	比较两个字符串的值。返回一个整数值。
String.CompareOrdinal	比较两个字符串的值而不考虑本地区域性。返回一个整数值。
String.CompareTo	比较当前字符串对象和另一个字符串。返回一个整数值。
String.StartsWith	确定字符串是否以传递字的符串开头。返回一个布尔值。
String.EndsWith	确定字符串是否以传递的字符串结尾。返回一个布尔值。
String.Contains	确定一个字符或字符串是否出现在另一个字符串中。返回一个布尔值。
String.Equals	确定两个字符串是否相同。返回一个布尔值。
String.IndexOf	返回字符或字符串的索引位置，从正在检查的字符串的开头开始。返回一个整数值。
String.LastIndexOf	返回字符或字符串的索引位置，从正在检查的字符串的结尾开始。返回一个整数值。

“比较”方法

静态 String.Compare 方法可以全面比较两个字符串。此方法区分区域性。你可以使用此函数比较两个字符串或两个字符串的子字符串。此外，还提供考虑或忽略大小写和区域性差别的重载。下表显示此方法可能返回三个整数值。

负整数	在排序顺序中，第一个字符串在第二个字符串之前。 - 或 - 第一个字符串是 `null`。
0	第一个字符串和第二个字符串相等。 - 或 - 两个字符串都是 `null`。

正整数 - 或 - 1	在排序顺序中，第一个字符串在第二个字符串之后。 - 或 - 第二个字符串是 `null`。

> **IMPORTANT**
> String.Compare 方法主要用于对字符串进行排序。不应使用 String.Compare 方法来测试相等性（即，显式查找返回值 0 而不考虑一个字符串是否小于或大于另一个）。相反，若要确定两个字符串是否相等，请使用 String.Equals(String, String, StringComparison) 方法。

下面的示例使用 String.Compare 方法来确定两个字符串的相对值。

```
String^ string1 = "Hello World!";
Console::WriteLine(String::Compare(string1, "Hello World?"));
```

```
string string1 = "Hello World!";
Console.WriteLine(String.Compare(string1, "Hello World?"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(String.Compare(string1, "Hello World?"))
```

此示例向控制台显示 `-1`。

前面的示例默认区分区域性。若要执行不区分区域性的字符串比较，请使用 String.Compare 方法的重载，这样就可以通过提供 *区域性* 参数来指定要使用的区域性。有关展示了如何使用 String.Compare 方法执行非区域性敏感型比较的示例，请参阅**非区域性敏感型字符串比较**。

CompareOrdinal 方法

String.CompareOrdinal 方法比较两个字符串对象而不考虑本地区域性。此方法的返回值与上表中 `Compare` 方法返回的值相同。

> **IMPORTANT**
> String.CompareOrdinal 方法主要用于对字符串进行排序。不应使用 String.CompareOrdinal 方法来测试相等性（即，显式查找返回值 0 而不考虑一个字符串是否小于或大于另一个）。相反，若要确定两个字符串是否相等，请使用 String.Equals(String, String, StringComparison) 方法。

下面的示例使用 `CompareOrdinal` 方法来比较两个字符串的值。

```
String^ string1 = "Hello World!";
Console::WriteLine(String::CompareOrdinal(string1, "hello world!"));
```

```
string string1 = "Hello World!";
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(String.CompareOrdinal(string1, "hello world!"))
```

此示例向控制台显示 `-32` 。

CompareTo 方法

String.CompareTo 方法比较当前字符串对象封装到另一个字符串或对象的字符串。此方法的返回值与上表中 String.Compare 方法返回的值相同。

> **IMPORTANT**
> String.CompareTo 方法主要用于对字符串进行排序。不应使用 String.CompareTo 方法来测试相等性(即，显式查找返回值 0 而不考虑一个字符串是否小于或大于另一个)。相反，若要确定两个字符串是否相等，请使用 String.Equals(String, String, StringComparison) 方法。

下面的示例使用 String.CompareTo 方法来比较 `string1` 对象和 `string2` 对象。

```
String^ string1 = "Hello World";
String^ string2 = "Hello World!";
int MyInt = string1->CompareTo(string2);
Console::WriteLine( MyInt );
```

```
string string1 = "Hello World";
string string2 = "Hello World!";
int MyInt = string1.CompareTo(string2);
Console.WriteLine( MyInt );
```

```
Dim string1 As String = "Hello World"
Dim string2 As String = "Hello World!"
Dim MyInt As Integer = string1.CompareTo(string2)
Console.WriteLine(MyInt)
```

此示例向控制台显示 `-1` 。

String.CompareTo 方法的所有重载均默认执行区分区域性且区分大小写的比较。此方法不提供任何允许执行不区分区域性的比较的重载。为了代码的清楚起见，建议改为使用 `String.Compare` 方法，指定 CultureInfo.CurrentCulture 执行区分区域性的操作或指定 CultureInfo.InvariantCulture 执行不区分区域性的操作。有关演示如何使用 `String.Compare` 方法来执行区分和不区分区域性的比较的示例，请参阅 执行不区分区域性的字符串比较。

Equals 方法

String.Equals 方法能够轻松确定两个字符串是否相等。这个区分大小写的方法返回 `true` 或 `false` 布尔值。它可以在现有类中使用，如下一个示例所示。下面的示例使用 `Equals` 方法来确定一个字符串对象是否包含短语“Hello World”。

```
String^ string1 = "Hello World";
Console::WriteLine(string1->Equals("Hello World"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.Equals("Hello World"));
```

```
Dim string1 As String = "Hello World"
Console.WriteLine(string1.Equals("Hello World"))
```

此示例向控制台显示 `True` 。

此方法还可作为静态方法使用。以下示例使用静态方法比较两个字符串对象。

```
String^ string1 = "Hello World";
String^ string2 = "Hello World";
Console::WriteLine(String::Equals(string1, string2));
```

```
string string1 = "Hello World";
string string2 = "Hello World";
Console.WriteLine(String.Equals(string1, string2));
```

```
Dim string1 As String = "Hello World"
Dim string2 As String = "Hello World"
Console.WriteLine(String.Equals(string1, string2))
```

此示例向控制台显示 `True` 。

StartsWith 和 EndsWith 方法

可以使用 String.StartsWith 方法来确定一个字符串对象是否与另一个字符串以相同字符开头。如果当前字符串对象以传递的字符串开头，这个区分大小写的方法将返回 `true` ，否则返回 `false` 。以下示例使用此方法来确定一个字符串对象是否以"Hello"开头。

```
String^ string1 = "Hello World";
Console::WriteLine(string1->StartsWith("Hello"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.StartsWith("Hello"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.StartsWith("Hello"))
```

此示例向控制台显示 `True` 。

String.EndsWith 方法比较传递的字符串和当前字符串对象末尾的字符。它也返回一个布尔值。下面的示例使用 `EndsWith` 方法检查字符串的末尾。

```
String^ string1 = "Hello World";
Console::WriteLine(string1->EndsWith("Hello"));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.EndsWith("Hello"));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.EndsWith("Hello"))
```

此示例向控制台显示 `False` 。

IndexOf 和 LastIndexOf 方法

可以使用 String.IndexOf 方法来确定特定字符在字符串中的第一个匹配项的位置。这个区分大小写的方法使用从零开始的索引从字符串的开头开始计数，并返回所传递字符的位置。如果无法找到该字符，则返回值 –1。

下面的示例使用 `IndexOf` 方法搜索字符"`l`"在字符串中的第一个匹配项。

```
String^ string1 = "Hello World";
Console::WriteLine(string1->IndexOf('l'));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.IndexOf('l'));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.IndexOf("l"))
```

此示例向控制台显示 `2` 。

String.LastIndexOf 方法类似于 `String.IndexOf` 方法，但它返回特定字符在字符串中的最后一个匹配项的位置。它不区分大小写，并且使用从零开始的索引。

下面的示例使用 `LastIndexOf` 方法搜索字符"`l`"在字符串中的最后一个匹配项。

```
String^ string1 = "Hello World";
Console::WriteLine(string1->LastIndexOf('l'));
```

```
string string1 = "Hello World";
Console.WriteLine(string1.LastIndexOf('l'));
```

```
Dim string1 As String = "Hello World!"
Console.WriteLine(string1.LastIndexOf("l"))
```

此示例向控制台显示 `9` 。

与 String.Remove 方法结合使用时，这两种方法都很有用。可以使用 `IndexOf` 或 `LastIndexOf` 方法来检索字符的位置，然后将该位置提供给 `Remove` 方法，以删除字符或以该字符开头的单词。

请参阅

- 有关使用 .NET 中字符串的最佳做法
- 基本字符串操作
- 执行不区分区域性的字符串操作

- 排序权重表 - 适用于 Windows 上的 .NET Framework 和 .NET Core 1.0-3.1
- 默认 Unicode 排序元素表 - 适用于所有平台上的 .NET 5，以及 Linux 和 macOS 上的 .NET Core

更改 .NET 中的大小写

2021/11/16 •

如果你编写可接受用户输入的应用程序，则永远无法确定用户以大写还是小写来输入数据。通常，你希望字符串统一采用大写或小写，尤其是在用户界面显示时。下表介绍 3 种更改大小写的方法：前两个方法提供可接受区域性的重载。

""	""
String.ToUpper	将字符串中的所有字符均转换为大写。
String.ToLower	将字符串中的所有字符均转换为小写。
TextInfo.ToTitleCase	将字符串转换为首字母大写。

WARNING

请注意，为了对 String.ToUpper 和 String.ToLower 方法进行比较或测试它们是否相等，这两种方法不应用于转换字符串。有关详细信息，请参阅比较混合大小写的字符串部分。

比较混合大小写的字符串

若要比较混合大小写的字符串以确定它们的顺序，请调用具有 String.CompareTo 方法中具有 `comparisonType` 参数的其中一个重载，并向 `comparisonType` 自变量提供 StringComparison.CurrentCultureIgnoreCase、StringComparison.InvariantCultureIgnoreCase 或 StringComparison.OrdinalIgnoreCase 的值。对于使用特定区域性（而非当前区域性）的比较，请调用 String.CompareTo 方法中具有 `culture` 和 `options` 参数的重载，并提供 CompareOptions.IgnoreCase 的值作为 `options` 参数。

若要比较混合大小写的字符串以确定它们是否相等，请调用 String.Equals 方法中具有 `comparisonType` 参数的其中一个重载，并向 `comparisonType` 自变量提供 StringComparison.CurrentCultureIgnoreCase、StringComparison.InvariantCultureIgnoreCase 或 StringComparison.OrdinalIgnoreCase 的值。

有关详细信息，请参阅有关使用字符串的最佳实践。

ToUpper

String.ToUpper 方法将字符串中的所有字符均更改为大写。下面的示例将字符串"Hello World!"转换为 从混合大小写转换为大写。

```
string properString = "Hello World!";
Console.WriteLine(properString.ToUpper());
// This example displays the following output:
//       HELLO WORLD!
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.ToUpper())
' This example displays the following output:
'       HELLO WORLD!
```

默认情况下，上述示例区分区域性；它应用当前区域性的大小写约定。若要执行非区域性敏感型大小写更改或应用特定区域性的大小写约定，请使用 String.ToUpper(CultureInfo) 方法重载，并向 culture 参数提供 CultureInfo.InvariantCulture 值或表示指定区域性的 System.Globalization.CultureInfo 对象。有关展示了如何使用 ToUpper 方法执行非区域性敏感型大小写更改的示例，请参阅执行非区域性敏感型大小写更改。

ToLower

String.ToLower 方法与上述方法类似，但改为将字符串中的所有字符均转换为小写。下面的示例将字符串"Hello World!"转换为 小写。

```
string properString = "Hello World!";
Console.WriteLine(properString.ToLower());
// This example displays the following output:
//       hello world!
```

```
Dim MyString As String = "Hello World!"
Console.WriteLine(MyString.ToLower())
' This example displays the following output:
'       hello world!
```

默认情况下，上述示例区分区域性；它应用当前区域性的大小写约定。若要执行非区域性敏感型大小写更改或应用特定区域性的大小写约定，请使用 String.ToLower(CultureInfo) 方法重载，并向 culture 参数提供 CultureInfo.InvariantCulture 值或表示指定区域性的 System.Globalization.CultureInfo 对象。有关展示了如何使用 ToLower(CultureInfo) 方法执行非区域性敏感型大小写更改的示例，请参阅执行非区域性敏感型大小写更改。

ToTitleCase

TextInfo.ToTitleCase 将每个单词的第一个字符转换为大写并将其余字符转换为小写。但是，全部大写的单词被假定为缩写词且不会转换。

TextInfo.ToTitleCase 方法区分区域性；即是，它使用特定区域性的大小写约定。为了调用方法，首先要从特定区域性的 CultureInfo.TextInfo 属性中检索表示特定区域性的大小写约定的 TextInfo 对象。

下面的示例将数组中的每个字符串传递至 TextInfo.ToTitleCase 方法。字符串包含适当的标题字符串以及首字母缩写词。通过使用英语(美国)区域性的大小写约定将字符串转换为首字母大写。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "a tale of two cities", "gROWL to the rescue",
                          "inside the US government", "sports and MLB baseball",
                          "The Return of Sherlock Holmes", "UNICEF and children"};

      TextInfo ti = CultureInfo.CurrentCulture.TextInfo;
      foreach (var value in values)
         Console.WriteLine("{0} --> {1}", value, ti.ToTitleCase(value));
   }
}
// The example displays the following output:
//    a tale of two cities --> A Tale Of Two Cities
//    gROWL to the rescue --> Growl To The Rescue
//    inside the US government --> Inside The US Government
//    sports and MLB baseball --> Sports And MLB Baseball
//    The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
//    UNICEF and children --> UNICEF And Children
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim values() As String = {"a tale of two cities", "gROWL to the rescue",
                                  "inside the US government", "sports and MLB baseball",
                                  "The Return of Sherlock Holmes", "UNICEF and children"}

        Dim ti As TextInfo = CultureInfo.CurrentCulture.TextInfo
        For Each value In values
            Console.WriteLine("{0} --> {1}", value, ti.ToTitleCase(value))
        Next
    End Sub
End Module
' The example displays the following output:
'    a tale of two cities --> A Tale Of Two Cities
'    gROWL to the rescue --> Growl To The Rescue
'    inside the US government --> Inside The US Government
'    sports and MLB baseball --> Sports And MLB Baseball
'    The Return of Sherlock Holmes --> The Return Of Sherlock Holmes
'    UNICEF and children --> UNICEF And Children
```

请注意，TextInfo.ToTitleCase 方法虽然区分区域性，但不提供语言方面的正确大小写规则。例如，在上述示例中，方法将“a tale of two cities”转换为“A Tale Of Two Cities”。但是，对于 en-US 区域性，语言方面的正确首字母大小写应为“A Tale of Two Cities”。

请参阅

- 基本字符串操作
- 执行不区分区域性的字符串操作

从字符串中提取子字符串

2021/11/16 •

本文介绍了一些用于提取字符串各个部分的不同技术。

- 当所需的子字符串由一个已知的分隔符(或多个分隔符)分隔时，请使用 Split 方法。
- 如果字符串符合某种固定模式，则可使用正则表达式。
- 如果不需要提取字符串中的所有子字符串，请结合使用 IndexOf 和 Substring 方法。

String.Split 方法

String.Split 可提供少量重载，来根据指定的一个或多个分隔符将字符串分解为一组子字符串。可以选择限制最终结果中子字符串的总数、剪裁子字符串中的空白字符或排除空子字符串。

下面的示例显示了三种不同的 `String.Split()` 重载。第一个示例调用 Split(Char[]) 重载，而不传递任何分隔符。如果未指定任何分隔符，`String.Split()` 将使用默认分隔符(空白字符)来拆分字符串。

```
string s = "You win some. You lose some.";

string[] subs = s.Split();

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some.
// Substring: You
// Substring: lose
// Substring: some.
```

```
Dim s As String = "You win some. You lose some."
Dim subs As String() = s.Split()

For Each substring As String In subs
    Console.WriteLine("Substring: {0}", substring)
Next

' This example produces the following output:
'
' Substring: You
' Substring: win
' Substring: some.
' Substring: You
' Substring: lose
' Substring: some.
```

正如你所看到的那样，两个子字符串之间包含句点字符 (`.`)。如果要排除句点字符，可以将句点字符添加为额外的分隔符。下面的示例演示了如何执行此操作。

```
string s = "You win some. You lose some.";

string[] subs = s.Split(' ', '.');

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some
// Substring:
// Substring: You
// Substring: lose
// Substring: some
// Substring:
```

```
Dim s As String = "You win some. You lose some."
Dim subs As String() = s.Split(" "c, "."c)

For Each substring As String In subs
    Console.WriteLine("Substring: {0}", substring)
Next

' This example produces the following output:
'
' Substring: You
' Substring: win
' Substring: some
' Substring:
' Substring: You
' Substring: lose
' Substring: some
' Substring:
```

子字符串之间的句点消息，但现在包含了两个额外的空子字符串。这些空子字符串表示单词与紧跟单词之后的句点之间的子字符串。若要从生成的数组中删除空字符串，可以调用 Split(Char[], StringSplitOptions) 重载，并为 `options` 参数指定 StringSplitOptions.RemoveEmptyEntries。

```
string s = "You win some. You lose some.";
char[] separators = new char[] { ' ', '.' };

string[] subs = s.Split(separators, StringSplitOptions.RemoveEmptyEntries);

foreach (string sub in subs)
{
    Console.WriteLine($"Substring: {sub}");
}

// This example produces the following output:
//
// Substring: You
// Substring: win
// Substring: some
// Substring: You
// Substring: lose
// Substring: some
```

```
Dim s As String = "You win some. You lose some."
Dim separators As Char() = New Char() {" "c, "."c}
Dim subs As String() = s.Split(separators, StringSplitOptions.RemoveEmptyEntries)

For Each substring As String In subs
    Console.WriteLine("Substring: {0}", substring)
Next

' This example produces the following output:
'
' Substring: You
' Substring: win
' Substring: some
' Substring: You
' Substring: lose
' Substring: some
```

正则表达式

如果字符串符合某种固定模式，则可以使用正则表达式提取并处理其元素。例如，如果字符串采用“数字 操作数 数字” 格式，则可以使用正则表达式提取并处理字符串的元素。下面是一个示例：

```
String[] expressions = { "16 + 21", "31 * 3", "28 / 3",
                         "42 - 18", "12 * 7",
                         "2, 4, 6, 8" };
String pattern = @"(\d+)\s+([-+*/])\s+(\d+)";

foreach (string expression in expressions)
{
    foreach (System.Text.RegularExpressions.Match m in
    System.Text.RegularExpressions.Regex.Matches(expression, pattern))
    {
        int value1 = Int32.Parse(m.Groups[1].Value);
        int value2 = Int32.Parse(m.Groups[3].Value);
        switch (m.Groups[2].Value)
        {
            case "+":
                Console.WriteLine("{0} = {1}", m.Value, value1 + value2);
                break;
            case "-":
                Console.WriteLine("{0} = {1}", m.Value, value1 - value2);
                break;
            case "*":
                Console.WriteLine("{0} = {1}", m.Value, value1 * value2);
                break;
            case "/":
                Console.WriteLine("{0} = {1:N2}", m.Value, value1 / value2);
                break;
        }
    }
}

// The example displays the following output:
//       16 + 21 = 37
//       31 * 3 = 93
//       28 / 3 = 9.33
//       42 - 18 = 24
//       12 * 7 = 84
```

```
Dim expressions() As String = {"16 + 21", "31 * 3", "28 / 3",
                               "42 - 18", "12 * 7",
                               "2, 4, 6, 8"}

Dim pattern As String = "(\d+)\s+([-+*/])\s+(\d+)"
For Each expression In expressions
    For Each m As Match In Regex.Matches(expression, pattern)
        Dim value1 As Integer = Int32.Parse(m.Groups(1).Value)
        Dim value2 As Integer = Int32.Parse(m.Groups(3).Value)
        Select Case m.Groups(2).Value
            Case "+"
                Console.WriteLine("{0} = {1}", m.Value, value1 + value2)
            Case "-"
                Console.WriteLine("{0} = {1}", m.Value, value1 - value2)
            Case "*"
                Console.WriteLine("{0} = {1}", m.Value, value1 * value2)
            Case "/"
                Console.WriteLine("{0} = {1:N2}", m.Value, value1 / value2)
        End Select
    Next
Next

' The example displays the following output:
'       16 + 21 = 37
'       31 * 3 = 93
'       28 / 3 = 9.33
'       42 - 18 = 24
'       12 * 7 = 84
```

正则表达式模式 `(\d+)\s+([-+*/])\s+(\d+)` 的定义如下：

“	“
`(\d+)`	匹配一个或多个十进制数字。这是第一个捕获组。
`\s+`	匹配一个或多个空白字符。
`([-+*/])`	匹配算术运算符(+、-、* 或 /)。这是第二个捕获组。
`\s+`	匹配一个或多个空白字符。
`(\d+)`	匹配一个或多个十进制数字。这是第三个捕获组。

你也可以使用正则表达式根据某种模式而非固定字符集提取字符串中的子字符串。在下面的任意一种情况下，常用此方案：

- 一个或多个分隔符不总是用作 String 实例中的分隔符。
- 分隔符的顺序和数量多变或未知。

例如，不能使用 Split 方法拆分以下字符串，因为 `\n`(换行)符的数量是可变的，并且它们不总是用作分隔符。

```
[This is captured\ntext.]\n\n[\n[This is more captured text.]\n]
\n[Some more captured text:\n   Option1\n   Option2][Terse text.]
```

正则表达式可以轻松拆分此字符串，如下面的示例所示。

```
String input = "[This is captured\ntext.]\n\n[\n" +
               "[This is more captured text.]\n]\n" +
               "[Some more captured text:\n   Option1" +
               "\n   Option2][Terse text.]";
String pattern = @"\[([^\[\]]+)\]";
int ctr = 0;

foreach (System.Text.RegularExpressions.Match m in
   System.Text.RegularExpressions.Regex.Matches(input, pattern))
{
    Console.WriteLine("{0}: {1}", ++ctr, m.Groups[1].Value);
}

// The example displays the following output:
//       1: This is captured
//       text.
//       2: This is more captured text.
//       3: Some more captured text:
//          Option1
//          Option2
//       4: Terse text.
```

```
Dim input As String = String.Format("[This is captured{0}text.]" +
                                    "{0}{0}[{0}[This is more " +
                                    "captured text.]{0}{0}" +
                                    "[Some more captured text:" +
                                    "{0}   Option1" +
                                    "{0}   Option2][Terse text.]",
                                    vbCrLf)
Dim pattern As String = "\[([^\[\]]+)\]"
Dim ctr As Integer = 0
For Each m As Match In Regex.Matches(input, pattern)
    ctr += 1
    Console.WriteLine("{0}: {1}", ctr, m.Groups(1).Value)
Next

' The example displays the following output:
'       1: This is captured
'       text.
'       2: This is more captured text.
'       3: Some more captured text:
'          Option1
'          Option2
'       4: Terse text.
```

正则表达式模式 `\[([^\[\]]+)\]` 的定义如下：

"	"
`\[`	匹配左方括号。
`([^\[\]]+)`	一次或多次与非左或右方括号的字符匹配 这是第一个捕获组。
`\]`	匹配右方括号。

Regex.Split 方法几乎与 String.Split 相同，不同之处在于，它根据正则表达式模式而非固定字符集拆分字符串。例如，下面的示例使用 Regex.Split 方法拆分字符串，该字符串包含由连字符和其他字符的各种组合分隔的子字符串。

```
String input = "abacus -- alabaster - * - atrium -+- " +
               "any -*- actual - + - armoire - - alarm";
String pattern = @"\s-\s?[+*]?\s?-\s";
String[] elements = System.Text.RegularExpressions.Regex.Split(input, pattern);

foreach (string element in elements)
    Console.WriteLine(element);

// The example displays the following output:
//       abacus
//       alabaster
//       atrium
//       any
//       actual
//       armoire
//       alarm
```

```
Dim input As String = "abacus -- alabaster - * - atrium -+- " +
                      "any -*- actual - + - armoir - - alarm"
Dim pattern As String = "\s-\s?[+*]?\s?-\s"
Dim elements() As String = Regex.Split(input, pattern)
For Each element In elements
    Console.WriteLine(element)
Next

' The example displays the following output:
'       abacus
'       alabaster
'       atrium
'       any
'       actual
'       armoir
'       alarm
```

正则表达式模式 `\s-\s?[+*]?\s?-\s` 的定义如下：

“	“
`\s-`	匹配后跟一个连字符的空白字符。
`\s?`	匹配零个或一个空白字符。
`[+*]?`	与 + 或 * 字符的零个或一个匹配项匹配。
`\s?`	匹配零个或一个空白字符。
`-\s`	匹配后跟一个空白字符的连字符。

String.IndexOf 和 String.Substring 方法

如果并不需要字符串中的所有子字符串，则可能更想要使用一种字符串比较方法来返回匹配开始之处的索引。然后，可以调用 Substring 方法来提取所需的子字符串。字符串比较方法包括：

- IndexOf，它返回字符串实例中的某个字符或字符串的第一个匹配项的从零开始的索引。
- IndexOfAny，它返回在当前字符串实例中字符数组中任何字符的第一个匹配项的从零开始的索引。
- LastIndexOf，它返回字符串实例中的某个字符或字符串的最后一个匹配项的从零开始的索引。

- LastIndexOfAny，它返回在当前字符串实例中字符数组中任何字符的最后一个匹配项的从零开始的索引。

以下示例使用 IndexOf 方法查找字符串中的句点。然后，它使用 Substring 方法返回完整句子。

```
String s = "This is the first sentence in a string. " +
               "More sentences will follow. For example, " +
               "this is the third sentence. This is the " +
               "fourth. And this is the fifth and final " +
               "sentence.";
var sentences = new List<String>();
int start = 0;
int position;

// Extract sentences from the string.
do
{
    position = s.IndexOf('.', start);
    if (position >= 0)
    {
        sentences.Add(s.Substring(start, position - start + 1).Trim());
        start = position + 1;
    }
} while (position > 0);

// Display the sentences.
foreach (var sentence in sentences)
    Console.WriteLine(sentence);

// The example displays the following output:
//       This is the first sentence in a string.
//       More sentences will follow.
//       For example, this is the third sentence.
//       This is the fourth.
//       And this is the fifth and final sentence.
```

```
    Dim input As String = "This is the first sentence in a string. " +
                          "More sentences will follow. For example, " +
                          "this is the third sentence. This is the " +
                          "fourth. And this is the fifth and final " +
                          "sentence."
    Dim sentences As New List(Of String)
    Dim start As Integer = 0
    Dim position As Integer

    ' Extract sentences from the string.
    Do
        position = input.IndexOf("."c, start)
        If position >= 0 Then
            sentences.Add(input.Substring(start, position - start + 1).Trim())
            start = position + 1
        End If
    Loop While position > 0

    ' Display the sentences.
    For Each sentence In sentences
        Console.WriteLine(sentence)
    Next
End Sub

' The example displays the following output:
'       This is the first sentence in a string.
'       More sentences will follow.
'       For example, this is the third sentence.
'       This is the fourth.
'       And this is the fifth and final sentence.
```

请参阅

- .NET 中的基本字符串操作
- .NET 正则表达式
- 如何使用 C# 中的 String.Split 分析字符串

使用 .NET 中的 StringBuilder 类

2021/11/16 •

String 对象不可变。每次使用 System.String 类中的方法之一，都要在内存中新建字符串对象，这就需要为新对象分配新空间。在需要重复修改字符串的情况下，与新建 String 对象关联的开销可能会非常大。若要修改字符串（而不新建对象），可以使用 System.Text.StringBuilder 类。例如，如果在循环中将许多字符串连接在一起，使用 StringBuilder 类可以提升性能。

导入 System.Text 命名空间

StringBuilder 类位于 System.Text 命名空间中。为了避免必须在代码中提供完全限定的类型名称，可以导入 System.Text 命名空间：

```
using namespace System;
using namespace System::Text;
```

```
using System;
using System.Text;
```

```
Imports System.Text
```

实例化 StringBuilder 对象

通过使用重载的构造函数方法之一初始化变量，可以新建 StringBuilder 类的实例，如下面的示例所示。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
```

设置容量和长度

虽然 StringBuilder 是动态对象，支持扩展它封装的字符串中的字符数，但可以指定值，作为对象可保留的字符数上限。此值称为“对象容量”，不得将它与当前 StringBuilder 保留的字符串长度相混淆。例如，可以使用长度为 5 的字符串“Hello”新建 StringBuilder 类的实例，同时可以指定此对象的最大容量为 25。修改 StringBuilder 时，除非达到容量，否则对象不会为自己重新分配空间。当达到容量时，将自动分配新的空间且容量翻倍。可以使用重载的构造函数之一，指定 StringBuilder 类的容量。下面的示例指定可以将 `myStringBuilder` 对象增加到最多 25 个空间。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!", 25);
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!", 25);
```

```
Dim myStringBuilder As New StringBuilder("Hello World!", 25)
```

另外，还可以使用读/写 Capacity 属性，设置对象的长度上限。下面的示例使用 Capacity 属性来定义对象的最大长度。

```
myStringBuilder->Capacity = 25;
```

```
myStringBuilder.Capacity = 25;
```

```
myStringBuilder.Capacity = 25
```

EnsureCapacity 方法可用于检查当前 StringBuilder 的容量。如果容量大于传递的值，则不进行任何更改；但是，如果容量小于传递的值，则会更改当前的容量以使其与传递的值匹配。

也可以查看或设置 Length 属性。如果将 Length 属性设置为大于 Capacity 属性的值，则自动将 Capacity 属性更改为与 Length 属性相同的值。如果将 Length 属性设置为小于当前 StringBuilder 对象内的字符串长度的值，则会缩短该字符串。

修改 StringBuilder 字符串

下表列出了可用于修改 StringBuilder 内容的方法。

“”	“”
StringBuilder.Append	将信息追加到当前 StringBuilder 的末尾。
StringBuilder.AppendFormat	用带格式文本替换字符串中传递的格式说明符。
StringBuilder.Insert	将字符串或对象插入到当前 StringBuilder 的指定索引中。
StringBuilder.Remove	从当前 StringBuilder 中删除指定数量的字符。
StringBuilder.Replace	将当前 StringBuilder 中出现的所有指定字符或字符串替换为其他的指定字符或字符串。

追加

Append 方法可用于将对象的文本或字符串表示形式添加到当前 StringBuilder 表示的字符串末尾。下面的示例将 StringBuilder 对象初始化为“Hello World”，然后将一些文本追加到该对象的末尾。将根据需要自动分配空间。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Append(" What a beautiful day.");
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World! What a beautiful day.
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Append(" What a beautiful day.");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World! What a beautiful day.
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Append(" What a beautiful day.")
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello World! What a beautiful day.
```

AppendFormat

StringBuilder.AppendFormat 方法将文本添加到 StringBuilder 对象末尾。它调用要设置格式的一个或多个对象的 IFormattable 实现，支持复合格式功能(有关详细信息，请参阅复合格式)。因此，它接受数字、日期和时间以及枚举值的标准格式字符串、数字以及日期和时间值的自定义格式字符串，以及为自定义类型定义的格式字符串。(有关格式化的信息，请参阅格式设置类型。)此方法可用于自定义变量格式，并将这些值追加到 StringBuilder。下面的示例使用 AppendFormat 方法，将格式为货币值的整数值添加到 StringBuilder 对象末尾。

```
int MyInt = 25;
StringBuilder^ myStringBuilder = gcnew StringBuilder("Your total is ");
myStringBuilder->AppendFormat("{0:C} ", MyInt);
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Your total is $25.00
```

```
int MyInt = 25;
StringBuilder myStringBuilder = new StringBuilder("Your total is ");
myStringBuilder.AppendFormat("{0:C} ", MyInt);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Your total is $25.00
```

```
Dim MyInt As Integer = 25
Dim myStringBuilder As New StringBuilder("Your total is ")
myStringBuilder.AppendFormat("{0:C} ", MyInt)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'     Your total is $25.00
```

Insert

Insert 方法将字符串或对象添加到当前 StringBuilder 对象中的指定位置。下面的示例使用此方法，将字词插入 StringBuilder 对象的第六个位置。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Insert(6,"Beautiful ");
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello Beautiful World!
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Insert(6,"Beautiful ");
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello Beautiful World!
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Insert(6, "Beautiful ")
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'      Hello Beautiful World!
```

删除

可以使用 Remove方法，从当前 StringBuilder 对象中指定索引（从零开始编制）处开始删除指定数量的字符。下面的示例使用 Remove 方法缩短 StringBuilder 对象。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Remove(5,7);
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Remove(5,7);
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Remove(5, 7)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello
```

替换

Replace 方法可用于将 StringBuilder 对象内的字符替换为另一个指定的字符。下面的示例使用 Replace 方法，在 StringBuilder 对象中搜索感叹号字符 (!) 的所有实例，并将它们替换为问号字符 (?)。

```
StringBuilder^ myStringBuilder = gcnew StringBuilder("Hello World!");
myStringBuilder->Replace('!', '?');
Console::WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World?
```

```
StringBuilder myStringBuilder = new StringBuilder("Hello World!");
myStringBuilder.Replace('!', '?');
Console.WriteLine(myStringBuilder);
// The example displays the following output:
//       Hello World?
```

```
Dim myStringBuilder As New StringBuilder("Hello World!")
myStringBuilder.Replace("!"c, "?"c)
Console.WriteLine(myStringBuilder)
' The example displays the following output:
'       Hello World?
```

将 StringBuilder 对象转换为字符串

必须先将 StringBuilder 对象转换为 String 对象，然后才能将 StringBuilder 对象表示的字符串传递给包含 String 参数的方法，或在用户界面中显示它。可通过调用 StringBuilder.ToString 方法来执行此转换。下面的示例先调用许多 StringBuilder 方法，再调用 StringBuilder.ToString() 方法来显示字符串。

```
using System;
using System.Text;

public class Example
{
   public static void Main()
   {
      StringBuilder sb = new StringBuilder();
      bool flag = true;
      string[] spellings = { "recieve", "receeve", "receive" };
      sb.AppendFormat("Which of the following spellings is {0}:", flag);
      sb.AppendLine();
      for (int ctr = 0; ctr <= spellings.GetUpperBound(0); ctr++) {
         sb.AppendFormat("   {0}. {1}", ctr, spellings[ctr]);
         sb.AppendLine();
      }
      sb.AppendLine();
      Console.WriteLine(sb.ToString());
   }
}
// The example displays the following output:
//       Which of the following spellings is True:
//          0. recieve
//          1. receeve
//          2. receive
```

```
Imports System.Text

Module Example
    Public Sub Main()
        Dim sb As New StringBuilder()
        Dim flag As Boolean = True
        Dim spellings() As String = {"recieve", "receeve", "receive"}
        sb.AppendFormat("Which of the following spellings is {0}:", flag)
        sb.AppendLine()
        For ctr As Integer = 0 To spellings.GetUpperBound(0)
            sb.AppendFormat("   {0}. {1}", ctr, spellings(ctr))
            sb.AppendLine()
        Next
        sb.AppendLine()
        Console.WriteLine(sb.ToString())
    End Sub
End Module
' The example displays the following output:
'       Which of the following spellings is True:
'          0. recieve
'          1. receeve
'          2. receive
```

请参阅

- System.Text.StringBuilder
- 基本字符串操作
- 格式设置类型

如何：执行 .NET 中的基本字符串控制

2021/11/16 •

下面的示例使用基本字符串操作主题中介绍的一些方法，构造模拟现实应用执行字符串控制的类。`MailToData` 类将个人的姓名和地址存储在单独的属性中，并提供一种将 `City`、`State` 和 `Zip` 字段合并成向用户显示的单个字符串的方式。此外，该类允许用户以单个字符串的形式输入城市、省/市/自治区和邮政编码信息。应用程序将自动分析单个字符串，并将正确的信息输入到相应的属性中。

为简单起见，此示例使用带命令行接口的控制台应用程序。

示例

```
using System;

class MainClass
{
   static void Main()
   {
      MailToData MyData = new MailToData();

      Console.Write("Enter Your Name: ");
      MyData.Name = Console.ReadLine();
      Console.Write("Enter Your Address: ");
      MyData.Address = Console.ReadLine();
      Console.Write("Enter Your City, State, and ZIP Code separated by spaces: ");
      MyData.CityStateZip = Console.ReadLine();
      Console.WriteLine();

      if (MyData.Validated) {
         Console.WriteLine("Name: {0}", MyData.Name);
         Console.WriteLine("Address: {0}", MyData.Address);
         Console.WriteLine("City: {0}", MyData.City);
         Console.WriteLine("State: {0}", MyData.State);
         Console.WriteLine("Zip: {0}", MyData.Zip);

         Console.WriteLine("\nThe following address will be used:");
         Console.WriteLine(MyData.Address);
         Console.WriteLine(MyData.CityStateZip);
      }
   }
}

public class MailToData
{
   string name = "";
   string address = "";
   string citystatezip = "";
   string city = "";
   string state = "";
   string zip = "";
   bool parseSucceeded = false;

   public string Name
   {
      get{return name;}
      set{name = value;}
   }

   public string Address
   {
```

```
      get{return address;}
      set{address = value;}
   }

   public string CityStateZip
   {
      get {
         return String.Format("{0}, {1} {2}", city, state, zip);
      }
      set {
         citystatezip = value.Trim();
         ParseCityStateZip();
      }
   }

   public string City
   {
      get{return city;}
      set{city = value;}
   }

   public string State
   {
      get{return state;}
      set{state = value;}
   }

   public string Zip
   {
      get{return zip;}
      set{zip = value;}
   }

   public bool Validated
   {
      get { return parseSucceeded; }
   }

   private void ParseCityStateZip()
   {
      string msg = "";
      const string msgEnd = "\nYou must enter spaces between city, state, and zip code.\n";

      // Throw a FormatException if the user did not enter the necessary spaces
      // between elements.
      try
      {
         // City may consist of multiple words, so we'll have to parse the
         // string from right to left starting with the zip code.
         int zipIndex = citystatezip.LastIndexOf(" ");
         if (zipIndex == -1) {
            msg = "\nCannot identify a zip code." + msgEnd;
            throw new FormatException(msg);
         }
         zip = citystatezip.Substring(zipIndex + 1);

         int stateIndex = citystatezip.LastIndexOf(" ", zipIndex - 1);
         if (stateIndex == -1) {
            msg = "\nCannot identify a state." + msgEnd;
            throw new FormatException(msg);
         }
         state = citystatezip.Substring(stateIndex + 1, zipIndex - stateIndex - 1);
         state = state.ToUpper();

         city = citystatezip.Substring(0, stateIndex);
         if (city.Length == 0) {
            msg = "\nCannot identify a city." + msgEnd;
            throw new FormatException(msg);
         }
```

```
         parseSucceeded = true;
      }
      catch (FormatException ex)
      {
         Console.WriteLine(ex.Message);
      }
   }

   private string ReturnCityStateZip()
    {
        // Make state uppercase.
        state = state.ToUpper();

        // Put the value of city, state, and zip together in the proper manner.
        string MyCityStateZip = String.Concat(city, ", ", state, " ", zip);

        return MyCityStateZip;
    }
}
```

```
Class MainClass
    Public Shared Sub Main()
        Dim MyData As New MailToData()

        Console.Write("Enter Your Name: ")
        MyData.Name = Console.ReadLine()
        Console.Write("Enter Your Address: ")
        MyData.Address = Console.ReadLine()
        Console.Write("Enter Your City, State, and ZIP Code separated by spaces: ")
        MyData.CityStateZip = Console.ReadLine()
        Console.WriteLine()

        If MyData.Validated Then
            Console.WriteLine("Name: {0}", MyData.Name)
            Console.WriteLine("Address: {0}", MyData.Address)
            Console.WriteLine("City: {0}", MyData.City)
            Console.WriteLine("State: {0}", MyData.State)
            Console.WriteLine("ZIP Code: {0}", MyData.Zip)

            Console.WriteLine("The following address will be used:")
            Console.WriteLine(MyData.Address)
            Console.WriteLine(MyData.CityStateZip)
        End If
    End Sub
End Class

Public Class MailToData
    Private strName As String = ""
    Private strAddress As String = ""
    Private strCityStateZip As String = ""
    Private strCity As String = ""
    Private strState As String = ""
    Private strZip As String = ""
    Private parseSucceeded As Boolean = False

    Public Property Name() As String
        Get
            Return strName
        End Get
        Set
            strName = value
        End Set
    End Property

    Public Property Address() As String
        Get
            Return strAddress
```

```
        End Get
        Set
            strAddress = value
        End Set
    End Property

    Public Property CityStateZip() As String
        Get
            Return String.Format("{0}, {1} {2}", strCity, strState, strZip)
        End Get
        Set
            strCityStateZip = value.Trim()
            ParseCityStateZip()
        End Set
    End Property

    Public Property City() As String
        Get
            Return strCity
        End Get
        Set
            strCity = value
        End Set
    End Property

    Public Property State() As String
        Get
            Return strState
        End Get
        Set
            strState = value
        End Set
    End Property

    Public Property Zip() As String
        Get
            Return strZip
        End Get
        Set
            strZip = value
        End Set
    End Property

    Public ReadOnly Property Validated As Boolean
        Get
            Return parseSucceeded
        End Get
    End Property

    Private Sub ParseCityStateZip()
        Dim msg As String = Nothing
        Const msgEnd As String = vbCrLf +
                                 "You must enter spaces between city, state, and zip code." +
                                 vbCrLf

        ' Throw a FormatException if the user did not enter the necessary spaces
        ' between elements.
        Try
            ' City may consist of multiple words, so we'll have to parse the
            ' string from right to left starting with the zip code.
            Dim zipIndex As Integer = strCityStateZip.LastIndexOf(" ")
            If zipIndex = -1 Then
                msg = vbCrLf + "Cannot identify a zip code." + msgEnd
                Throw New FormatException(msg)
            End If
            strZip = strCityStateZip.Substring(zipIndex + 1)

            Dim stateIndex As Integer = strCityStateZip.LastIndexOf(" ", zipIndex - 1)
            If stateIndex = -1 Then
```

```
                msg = vbCrLf + "Cannot identify a state." + msgEnd
                Throw New FormatException(msg)
            End If
            strState = strCityStateZip.Substring(stateIndex + 1, zipIndex - stateIndex - 1)
            strState = strState.ToUpper()

            strCity = strCityStateZip.Substring(0, stateIndex)
            If strCity.Length = 0 Then
                msg = vbCrLf + "Cannot identify a city." + msgEnd
                Throw New FormatException(msg)
            End If
            parseSucceeded = True
        Catch ex As FormatException
            Console.WriteLine(ex.Message)
        End Try
    End Sub
End Class
```

执行上面的代码时，系统会要求用户输入其姓名和地址。应用程序将这些信息放入相应的属性并将信息返回向用户显示，同时创建一个显示城市、省/市/自治区和邮政编码信息的字符串。

请参阅

- 基本字符串操作

分析 .NET 中的字符串

2021/11/16 •

分析操作将表示某种 .NET 基类型的字符串转换为该基类型。例如，分析操作用于将字符串转换为浮点数字或日期和时间值。最常用于执行分析操作的方法是 `Parse` 方法。因为分析是格式设置（涉及将基类型转换为其字符串表示形式）的反向操作，所以有许多相同规则和约定适用。就像格式设置使用对象来实现 IFormatProvider 接口以提供区域性敏感型格式设置信息一样，分析也使用对象来实现 IFormatProvider 接口，以确定如何解释字符串表示形式。有关详细信息，请参阅格式类型。

本节内容

分析数值字符串
介绍了如何将字符串转换为 .NET 数字类型。

分析日期和时间字符串
介绍了如何将字符串转换为 .NET DateTime 类型。

分析其他字符串
介绍了如何将字符串转换为 Char、Boolean 和 Enum 类型。

相关章节

格式设置类型
介绍了基本格式设置概念，如格式说明符和格式提供程序。

.NET 中的类型转换
介绍了如何转换类型。

分析 .NET 中的数字字符串

2021/11/16 •

所有数字类型都具有两个静态分析方法（`Parse` 和 `TryParse`），可以使用它们将数字的字符串表示形式转换为数字类型。这两个方法使你可以分析使用标准数字格式字符串和自定义数字格式字符串中所述的格式字符串生成的字符串。默认情况下，`Parse` 和 `TryParse` 方法可以成功地将仅包含整数十进制数字的字符串转化为整数值。它们可以将包含整数和小数十进制数字、组分隔符和十进制分隔符的字符串转换为浮点值。`Parse` 方法在操作失败时引发异常，而 `TryParse` 方法返回 `false`。

分析和格式提供程序

通常，数值的字符串表示因区域性而异。数值字符串的元素都会因区域性而异，如货币符号、组（或千位）分隔符和十进制分隔符。分析方法可隐式或显式使用可以识别这些特定于区域性的变体的格式提供程序。如果在 `Parse` 或 `TryParse` 方法调用中未指定任何格式提供程序，使用的是与当前线程区域性关联的格式提供程序（NumberFormatInfo.CurrentInfo 属性返回的 NumberFormatInfo 对象）。

格式提供程序由 IFormatProvider 实现表示。此接口包含一个成员，即 GetFormat 方法；它需要使用一个参数，即表示要设置格式的类型的 Type 对象。此方法返回提供格式设置信息的对象。.NET 支持以下两个 IFormatProvider 实现，用于分析数字字符串：

- CultureInfo 对象，它的 CultureInfo.GetFormat 方法返回 NumberFormatInfo 对象，提供区域性专用格式设置信息。
- NumberFormatInfo 对象，它的 NumberFormatInfo.GetFormat 方法返回自己本身。

下面的示例尝试将数组中的每个字符串转换为 Double 值。它首先尝试使用反映“英语(美国)”区域性约定的格式提供程序来分析字符串。如果此操作抛出 FormatException，就会尝试使用反映“法语(法国)”区域性约定的格式提供程序分析字符串。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] values = { "1,304.16", "$1,456.78", "1,094", "152",
                          "123,45 €", "1 304,16", "Ae9f" };
      double number;
      CultureInfo culture = null;

      foreach (string value in values) {
         try {
            culture = CultureInfo.CreateSpecificCulture("en-US");
            number = Double.Parse(value, culture);
            Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number);
         }
         catch (FormatException) {
            Console.WriteLine("{0}: Unable to parse '{1}'.",
                              culture.Name, value);
            culture = CultureInfo.CreateSpecificCulture("fr-FR");
            try {
               number = Double.Parse(value, culture);
               Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number);
            }
            catch (FormatException) {
               Console.WriteLine("{0}: Unable to parse '{1}'.",
                                 culture.Name, value);
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//    en-US: 1,304.16 --> 1304.16
//
//    en-US: Unable to parse '$1,456.78'.
//    fr-FR: Unable to parse '$1,456.78'.
//
//    en-US: 1,094 --> 1094
//
//    en-US: 152 --> 152
//
//    en-US: Unable to parse '123,45 €'.
//    fr-FR: Unable to parse '123,45 €'.
//
//    en-US: Unable to parse '1 304,16'.
//    fr-FR: 1 304,16 --> 1304.16
//
//    en-US: Unable to parse 'Ae9f'.
//    fr-FR: Unable to parse 'Ae9f'.
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim values() As String = {"1,304.16", "$1,456.78", "1,094", "152",
                                  "123,45 €", "1 304,16", "Ae9f"}
        Dim number As Double
        Dim culture As CultureInfo = Nothing

        For Each value As String In values
            Try
                culture = CultureInfo.CreateSpecificCulture("en-US")
                number = Double.Parse(value, culture)
                Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number)
            Catch e As FormatException
                Console.WriteLine("{0}: Unable to parse '{1}'.",
                                  culture.Name, value)
                culture = CultureInfo.CreateSpecificCulture("fr-FR")
                Try
                    number = Double.Parse(value, culture)
                    Console.WriteLine("{0}: {1} --> {2}", culture.Name, value, number)
                Catch ex As FormatException
                    Console.WriteLine("{0}: Unable to parse '{1}'.",
                                      culture.Name, value)
                End Try
            End Try
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'    en-US: 1,304.16 --> 1304.16
'
'    en-US: Unable to parse '$1,456.78'.
'    fr-FR: Unable to parse '$1,456.78'.
'
'    en-US: 1,094 --> 1094
'
'    en-US: 152 --> 152
'
'    en-US: Unable to parse '123,45 €'.
'    fr-FR: Unable to parse '123,45 €'.
'
'    en-US: Unable to parse '1 304,16'.
'    fr-FR: 1 304,16 --> 1304.16
'
'    en-US: Unable to parse 'Ae9f'.
'    fr-FR: Unable to parse 'Ae9f'.
```

分析和 NumberStyles 值

分析操作可以处理的样式元素（如空格、组分隔符和十进制分隔符）由 NumberStyles 枚举值定义。默认情况下，表示整数值的字符串是使用 NumberStyles.Integer 值进行分析，此值仅允许数字、前导和尾随空格以及前导符号。表示浮点值的字符串是通过结合使用 NumberStyles.Float 和 NumberStyles.AllowThousands 值进行分析；此复合样式允许数字以及前导和尾随空格、前导符号、十进制分隔符、组分隔符和指数。通过调用包含 NumberStyles 类型参数的 `Parse` 或 `TryParse` 方法重载，并设置一个或多个 NumberStyles 标志，可以控制字符串中能够包含的样式元素，以便分析操作成功。

例如，包含组分隔符的字符串使用 Int32 方法不能转换为 Int32.Parse(String) 值。但是，如果使用 NumberStyles.AllowThousands 标记，可成功转换，如下面的示例所示。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string value = "1,304";
      int number;
      IFormatProvider provider = CultureInfo.CreateSpecificCulture("en-US");
      if (Int32.TryParse(value, out number))
         Console.WriteLine("{0} --> {1}", value, number);
      else
         Console.WriteLine("Unable to convert '{0}'", value);

      if (Int32.TryParse(value, NumberStyles.Integer | NumberStyles.AllowThousands,
                         provider, out number))
         Console.WriteLine("{0} --> {1}", value, number);
      else
         Console.WriteLine("Unable to convert '{0}'", value);
   }
}
// The example displays the following output:
//       Unable to convert '1,304'
//       1,304 --> 1304
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim value As String = "1,304"
        Dim number As Integer
        Dim provider As IFormatProvider = CultureInfo.CreateSpecificCulture("en-US")
        If Int32.TryParse(value, number) Then
            Console.WriteLine("{0} --> {1}", value, number)
        Else
            Console.WriteLine("Unable to convert '{0}'", value)
        End If

        If Int32.TryParse(value, NumberStyles.Integer Or NumberStyles.AllowThousands,
                          provider, number) Then
            Console.WriteLine("{0} --> {1}", value, number)
        Else
            Console.WriteLine("Unable to convert '{0}'", value)
        End If
    End Sub
End Module
' The example displays the following output:
'       Unable to convert '1,304'
'       1,304 --> 1304
```

WARNING

分析操作始终使用特定区域性的格式设置约定。如果未通过传递 CultureInfo 或 NumberFormatInfo 对象来指定区域性，使用的是与当前线程关联的区域性。

下表列出了 NumberStyles 枚举成员及其对分析操作的影响。

NUMBERSTYLES	
NumberStyles.None	仅允许数字。

NUMBERSTYLES 枚举	说明
NumberStyles.AllowDecimalPoint	允许十进制分隔符和小数数字。对于整数值，仅允许将零作为小数数字。有效的十进制分隔符是由 NumberFormatInfo.NumberDecimalSeparator 或 NumberFormatInfo.CurrencyDecimalSeparator 属性确定。
NumberStyles.AllowExponent	“e”或“E”字符可以用于指示指数记数法。有关详细信息，请参阅 NumberStyles。
NumberStyles.AllowLeadingWhite	允许前导空格。
NumberStyles.AllowTrailingWhite	允许尾随空格。
NumberStyles.AllowLeadingSign	正号或负号可以位于数字前面。
NumberStyles.AllowTrailingSign	正号或负号可以位于数字后面。
NumberStyles.AllowParentheses	括号可以用于指示负值。
NumberStyles.AllowThousands	允许组分隔符。组分隔符是由 NumberFormatInfo.NumberGroupSeparator 或 NumberFormatInfo.CurrencyGroupSeparator 属性确定。
NumberStyles.AllowCurrencySymbol	允许货币符号。货币符号是由 NumberFormatInfo.CurrencySymbol 属性定义。
NumberStyles.AllowHexSpecifier	要分析的字符串解释为十六进制数。它可以包含十六进制数 0-9、A-F 和 a-f。此标志只能用于分析整数值。

此外，NumberStyles 枚举提供以下包括多个 NumberStyles 标志的复合样式。

复合 NUMBERSTYLES 值	说明
NumberStyles.Integer	包括 NumberStyles.AllowLeadingWhite、NumberStyles.AllowTrailingWhite 和 NumberStyles.AllowLeadingSign 样式。这是用于分析整数值的默认样式。
NumberStyles.Number	包括 NumberStyles.AllowLeadingWhite、NumberStyles.AllowTrailingWhite、NumberStyles.AllowLeadingSign、NumberStyles.AllowTrailingSign、NumberStyles.AllowDecimalPoint 和 NumberStyles.AllowThousands 样式。
NumberStyles.Float	包括 NumberStyles.AllowLeadingWhite、NumberStyles.AllowTrailingWhite、NumberStyles.AllowLeadingSign、NumberStyles.AllowDecimalPoint 和 NumberStyles.AllowExponent 样式。
NumberStyles.Currency	包括除 NumberStyles.AllowExponent 和 NumberStyles.AllowHexSpecifier 之外的所有样式。
NumberStyles.Any	包括除 NumberStyles.AllowHexSpecifier 之外的所有样式。

NUMBERSTYLES	
NumberStyles.HexNumber	包括 NumberStyles.AllowLeadingWhite、NumberStyles.AllowTrailingWhite 和 NumberStyles.AllowHexSpecifier 样式。

分析和 Unicode 数字

Unicode 标准为各种书写系统中的数字定义码位。例如，从 U+0030 到 U+0039 的码位表示从 0 到 9 的基本拉丁文数字，从 U+09E6 到 U+09EF 的码位表示从 0 到 9 的孟加拉文数字，而从 U+FF10 到 U+FF19 的码位表示从 0 到 9 的全角数字。但是，分析方法唯一可识别的数字是具有 U+0030 到 U+0039 的码位的基本拉丁文数字 0-9。如果向数字分析方法传递包含其他任何数字的字符串，此方法会抛出 FormatException。

下面的示例使用 Int32.Parse 方法，分析包含不同书写系统中数字的字符串。正如示例输出所示，虽然尝试分析基本拉丁文数字获得成功，但分析全角、阿拉伯 - 印度文以及孟加拉文数字的尝试将失败。

```
using System;

public class Example
{
   public static void Main()
   {
      string value;
      // Define a string of basic Latin digits 1-5.
      value = "\u0031\u0032\u0033\u0034\u0035";
      ParseDigits(value);

      // Define a string of Fullwidth digits 1-5.
      value = "\uFF11\uFF12\uFF13\uFF14\uFF15";
      ParseDigits(value);

      // Define a string of Arabic-Indic digits 1-5.
      value = "\u0661\u0662\u0663\u0664\u0665";
      ParseDigits(value);

      // Define a string of Bangla digits 1-5.
      value = "\u09e7\u09e8\u09e9\u09ea\u09eb";
      ParseDigits(value);
   }

   static void ParseDigits(string value)
   {
      try {
         int number = Int32.Parse(value);
         Console.WriteLine("'{0}' --> {1}", value, number);
      }
      catch (FormatException) {
         Console.WriteLine("Unable to parse '{0}'.", value);
      }
   }
}
// The example displays the following output:
//       '12345' --> 12345
//       Unable to parse '12345'.
//       Unable to parse '١٢٣٤٥'.
//       Unable to parse '■■■■■'.
```

```
Module Example
    Public Sub Main()
        Dim value As String
        ' Define a string of basic Latin digits 1-5.
        value = ChrW(&h31) + ChrW(&h32) + ChrW(&h33) + ChrW(&h34) + ChrW(&h35)
        ParseDigits(value)

        ' Define a string of Fullwidth digits 1-5.
        value = ChrW(&hff11) + ChrW(&hff12) + ChrW(&hff13) + ChrW(&hff14) + ChrW(&hff15)
        ParseDigits(value)

        ' Define a string of Arabic-Indic digits 1-5.
        value = ChrW(&h661) + ChrW(&h662) + ChrW(&h663) + ChrW(&h664) + ChrW(&h665)
        ParseDigits(value)

        ' Define a string of Bangla digits 1-5.
        value = ChrW(&h09e7) + ChrW(&h09e8) + ChrW(&h09e9) + ChrW(&h09ea) + ChrW(&h09eb)
        ParseDigits(value)
    End Sub

    Sub ParseDigits(value As String)
        Try
            Dim number As Integer = Int32.Parse(value)
            Console.WriteLine("'{0}' --> {1}", value, number)
        Catch e As FormatException
            Console.WriteLine("Unable to parse '{0}'.", value)
        End Try
    End Sub
End Module
' The example displays the following output:
'       '12345' --> 12345
'       Unable to parse '12345'.
'       Unable to parse '١٢٣٤٥'.
'       Unable to parse '■■■■■'.
```

请参阅

- NumberStyles
- 分析字符串
- 格式设置类型

分析 .NET 中的日期和时间字符串

2021/11/16 •

分析字符串以将其转换为 DateTime 对象需要你指定有关如何以文本格式表示日期和时间的信息。不同的区域性所使用的日、月和年的顺序也不尽相同。某些时间表示方法使用 24 小时制，其他时间表示方法则会指定“AM”和“PM”。某些应用程序仅需要日期。其他应用程序仅需要时间。还有其他应用程序需要同时指定日期和时间。通过将字符串转换为 DateTime 对象的方法，你将能够提供有关你预期的格式和你的应用程序需要的日期和时间元素的详细信息。将文本正确转换为 DateTime 需要执行三个子任务：

1. 必须指定表示日期和时间的文本的预期格式。
2. 可以指定日期时间格式的区域性。
3. 可以指定如何以日期和时间格式设置文本表示方法中缺少的组成部分。

Parse 和 TryParse 方法可转换日期和时间的多个常见表示方法。ParseExact 和 TryParseExact 方法可转换符合日期和时间格式字符串指定的模式的字符串表示形式。（如需详细信息，请参阅有关标准日期和时间格式字符串和自定义日期和时间格式字符串的文章。）

当前的 DateTimeFormatInfo 对象提供对如何将文本解释为日期和时间的更好的控制。DateTimeFormatInfo 的属性描述了日期和时间分隔符，以及月、日、年代的名称，还有“AM”和“PM”标志的格式。当前线程区域性提供了表示当前区域性的 DateTimeFormatInfo。如果你希望使用特定区域性或自定义设置，请指定分析方法的 IFormatProvider 参数。对于 IFormatProvider 参数，应指定表示区域性的 CultureInfo 对象，或指定 DateTimeFormatInfo 对象。

表示日期或时间的文本可能缺少某些信息。例如，大多数人都会假定“3 月 12 日”这个日期表示当前年份。同样，“2018 年 3 月”表示年份为 2018，月份为 3 月。表示时间的文本通常仅包括小时、分钟和 AM/PM 标志。分析方法通过使用合理的默认值处理此类缺少的信息：

- 当仅存在时间时，日期部分将使用当前日期。
- 当仅存在日期时，时间部分将是午夜。
- 如果日期中未指定年份，则使用当前年份。
- 如果未指定一个月中的第几天，则使用一个月中的第一天。

如果字符串中存在日期，则它必须包括月份、某日或某年。如果存在时间，则它必须包括小时和分钟或 AM/PM 标志。

你可以指定 NoCurrentDateDefault 常量，以覆盖这些默认值。使用该常量时，任何缺少的年、月或天属性将设置为值 `1`。使用 Parse 的最后一个示例对此行为进行了演示。

除了日期和时间组成部分，日期和时间的字符串表示形式还可以包含指示时间与协调世界时 (UTC) 相差多少的偏移量。例如，字符串“2/14/2007 5:32:00 -7:00”定义比 UTC 早七个小时的时间。如果在时间的字符串表示形式中省略了偏移，分析方法返回 DateTime 对象，它的 Kind 属性设置为 DateTimeKind.Unspecified。如果指定了偏移，分析方法返回 DateTime 对象，它的 Kind 属性设置为 DateTimeKind.Local，且值调整为采用计算机的本地时区。可以通过结合使用 DateTimeStyles 值和分析方法来修改此行为。

格式提供程序还用于解释不明确的数字日期。不清楚字符串“02/03/04”所表示的日期的哪些组成部分是月、日和年。组成部分根据格式提供程序中相似日期格式的顺序进行解释。

Parse

下面的示例说明了如何使用 DateTime.Parse 方法将 `string` 转换为 DateTime。此示例使用与当前线程关联的区域性。如果与当前区域性关联的 CultureInfo 无法分析输入字符串，则会抛出 FormatException。

TIP

本文中的所有 C# 示例均在你的浏览器中运行。按“运行”按钮查看输出。你还可以对其进行编辑以自行实验。

NOTE

这些示例可在适用于 C# 和 Visual Basic 的 GitHub 文档存储库中获取。

```
string dateInput = "Jan 1, 2009";
var parsedDate = DateTime.Parse(dateInput);
Console.WriteLine(parsedDate);
// Displays the following output on a system whose culture is en-US:
//       1/1/2009 00:00:00
```

```
Dim MyString As String = "Jan 1, 2009"
Dim MyDateTime As DateTime = DateTime.Parse(MyString)
Console.WriteLine(MyDateTime)
' Displays the following output on a system whose culture is en-US:
'       1/1/2009 00:00:00
```

你也可以显式定义分析字符串时将使用其格式设置约定的区域性。指定 CultureInfo.DateTimeFormat 属性返回的一个标准 DateTimeFormatInfo 对象。下面的示例使用格式提供程序将德语字符串分析为 DateTime。它创建了一个表示 `de-DE` 区域性的 CultureInfo。`CultureInfo` 对象可以确保成功分析此特定的字符串。这会排除处于 CurrentThread 的 CurrentCulture 中的任何设置。

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo);
Console.WriteLine(dateTime);
// The example displays the following output:
//       6/12/2008 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo)
Console.WriteLine(MyDateTime)
' The example displays the following output:
'       6/12/2008 00:00:00
```

不过，虽然可以使用 Parse 方法重载指定自定义格式提供程序，但此方法不支持分析非标准格式。若要分析非标准格式的日期和时间，请改用 ParseExact 方法。

下面的示例使用 DateTimeStyles 枚举，指定不得将当前日期和时间信息添加到未指定字段的 DateTime。

```
var cultureInfo = new CultureInfo("de-DE");
string dateString = "12 Juni 2008";
var dateTime = DateTime.Parse(dateString, cultureInfo,
                              DateTimeStyles.NoCurrentDateDefault);
Console.WriteLine(dateTime);
// The example displays the following output if the current culture is en-US:
//      6/12/2008 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("de-DE")
Dim MyString As String = "12 Juni 2008"
Dim MyDateTime As DateTime = DateTime.Parse(MyString, MyCultureInfo,
                         DateTimeStyles.NoCurrentDateDefault)
Console.WriteLine(MyDateTime)
' The example displays the following output if the current culture is en-US:
'       6/12/2008 00:00:00
```

ParseExact

DateTime.ParseExact 方法将符合其中一个指定字符串模式的字符串转换为 DateTime 对象。将未采用其中一种指定格式的字符串传递给此方法时，会引发 FormatException。可以指定一种标准日期和时间格式说明符或自定义格式说明符的组合。使用自定义格式说明符可以构造自定义识别字符串。有关说明符的说明，请参见有关标准日期和时间格式字符串和自定义日期和时间格式字符串的主题。

在下面的示例中，向 DateTime.ParseExact 方法传递了一个要分析的字符串对象，后跟一个格式说明符，再后跟一个 CultureInfo 对象。此 ParseExact 方法只能分析在 `en-US` 区域性中遵循长日期模式的字符串。

```
var cultureInfo = new CultureInfo("en-US");
string[] dateStrings = { " Friday, April 10, 2009", "Friday, April 10, 2009" };
foreach (string dateString in dateStrings)
{
    try
    {
        var dateTime = DateTime.ParseExact(dateString, "D", cultureInfo);
        Console.WriteLine(dateTime);
    }
    catch (FormatException)
    {
        Console.WriteLine("Unable to parse '{0}'", dateString);
    }
}
// The example displays the following output:
//       Unable to parse ' Friday, April 10, 2009'
//       4/10/2009 00:00:00
```

```
Dim MyCultureInfo As New CultureInfo("en-US")
Dim MyString() As String = {" Friday, April 10, 2009", "Friday, April 10, 2009"}
For Each dateString As String In MyString
    Try
        Dim MyDateTime As DateTime = DateTime.ParseExact(dateString, "D",
                                                         MyCultureInfo)
        Console.WriteLine(MyDateTime)
    Catch e As FormatException
        Console.WriteLine("Unable to parse '{0}'", dateString)
    End Try
Next
' The example displays the following output:
'       Unable to parse ' Friday, April 10, 2009'
'       4/10/2009 00:00:00
```

Parse 和 ParseExact 方法的每个重载还包含 IFormatProvider 参数，用于提供有关字符串格式设置的区域性专用信息。此 IFormatProvider 对象为 CultureInfo 对象，表示标准区域性或 CultureInfo.DateTimeFormat 属性返回的 DateTimeFormatInfo 对象。ParseExact 还使用定义一个或多个自定义日期和时间格式的其他字符串或字符串数组参数。

请参阅

- 分析字符串
- 格式设置类型
- .NET 中的类型转换
- 标准日期和时间格式
- 自定义日期和时间格式字符串

分析 .NET 中的其他字符串

2021/11/16 •

除了数字和 DateTime 字符串之外，还可以将表示类型 Char、Boolean 和 Enum 的字符串分析为数据类型。

Char

与 Char 数据类型关联的静态分析方法 可用于将包含单个字符的字符串转换为其 Unicode 值。下面的代码示例将字符串分析为 Unicode 字符。

```
String^ MyString1 = "A";
char MyChar = Char::Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

```
string MyString1 = "A";
char MyChar = Char.Parse(MyString1);
// MyChar now contains a Unicode "A" character.
```

```
Dim MyString1 As String = "A"
Dim MyChar As Char = Char.Parse(MyString1)
' MyChar now contains a Unicode "A" character.
```

Boolean

Boolean 数据类型包含 Parse 方法，可用于将表示 Boolean 值的字符串转换为实际 Boolean 类型。此方法不区分大小写，可以成功分析包含“True”或“False”的字符串。与 Boolean 类型关联的 Parse 方法还可以分析两端是空格的字符串。如果传递的是其他任何字符串，FormatException 就会抛出。

下面的代码示例使用 Parse 方法，将字符串转换为 Boolean 值。

```
String^ MyString2 = "True";
bool MyBool = bool::Parse(MyString2);
// MyBool now contains a True Boolean value.
```

```
string MyString2 = "True";
bool MyBool = bool.Parse(MyString2);
// MyBool now contains a True Boolean value.
```

```
Dim MyString2 As String = "True"
Dim MyBool As Boolean = Boolean.Parse(MyString2)
' MyBool now contains a True Boolean value.
```

枚举

可以使用静态 Parse 方法将枚举类型初始化为字符串的值。此方法接受要分析的枚举类型、要分析的字符串和可选 Boolean 标志（指明分析是否区分大小写）。所分析的字符串可以包含用逗号分隔的多个值，这些值前面或后面可以是一个或多个空白（也称为空格）。当字符串包含多个值时，返回的对象的值是所有指定值通过按位 OR

运算组合的值。

下面的示例使用 Parse 方法，将字符串表示形式转换为枚举值。DayOfWeek 枚举从字符串初始化为 Thursday 。

```
String^ MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum::Parse(DayOfWeek::typeid, MyString3);
Console::WriteLine(MyDays);
// The result is Thursday.
```

```
string MyString3 = "Thursday";
DayOfWeek MyDays = (DayOfWeek)Enum.Parse(typeof(DayOfWeek), MyString3);
Console.WriteLine(MyDays);
// The result is Thursday.
```

```
Dim MyString3 As String = "Thursday"
Dim MyDays As DayOfWeek = CType([Enum].Parse(GetType(DayOfWeek), MyString3), DayOfWeek)
Console.WriteLine("{0:G}", MyDays)
' The result is Thursday.
```

另请参阅

- Parsing Strings
- 格式设置类型
- .NET 中的类型转换

.NET 正则表达式

2021/11/16 •

正则表达式提供了功能强大、灵活而又高效的方法来处理文本。正则表达式丰富的泛模式匹配表示法使你可以快速分析大量文本，以便：

- 查找特定字符模式。
- 验证文本以确保它匹配预定义模式（如电子邮件地址）。
- 提取、编辑、替换或删除文本子字符串。
- 将提取的字符串添加到集合中，以便生成报告。

对于处理字符串或分析大文本块的许多应用程序而言，正则表达式是不可缺少的工具。

正则表达式的工作方式

使用正则表达式处理文本的中心构件是正则表达式引擎（由 .NET 中的 System.Text.RegularExpressions.Regex 对象表示）。使用正则表达式处理文本至少要求向该正则表达式引擎提供以下两方面的信息：

- 要在文本中标识的正则表达式模式。

  在 .NET 中，正则表达式模式用特殊的语法或语言定义，该语法或语言与 Perl 5 正则表达式兼容，并添加了一些其他功能，例如从右到左匹配。有关更多信息，请参见正则表达式语言 - 快速参考。

- 要为正则表达式模式分析的文本。

Regex 类的方法使你可以执行以下操作：

- 通过调用 Regex.IsMatch 方法确定输入文本中是否具有正则表达式模式。有关使用 IsMatch 方法验证文本的示例，请参阅如何：确认字符串是有效的电子邮件格式。

- 通过调用 Regex.Match 或 Regex.Matches 方法检索匹配正则表达式模式的一个或所有文本匹配项。第一个方法返回提供有关匹配文本的信息的 System.Text.RegularExpressions.Match 对象。第二个方法返回 MatchCollection 对象，该对象对于在分析的文本中找到的每个匹配项包含一个 System.Text.RegularExpressions.Match 对象。

- 通过调用 Regex.Replace 方法替换匹配正则表达式模式的文本。有关使用 Replace 方法更改日期格式和移除字符串中的无效字符的示例，请参阅如何：从字符串中剥离无效字符以及示例：更改日期格式。

有关正则表达式对象模型的概述，请参见正则表达式对象模型。

若要详细了解正则表达式语言，请参阅正则表达式语言 - 快速参考，或下载和打印下面的小册子之一：

- 快速参考（Word (.docx) 格式）
- 快速参考（PDF (.pdf) 格式）

正则表达式示例

String 类包括许多字符串搜索和替换方法，当你要在较大字符串中定位文本字符串时，可以使用这些方法。当你希望在较大字符串中定位若干子字符串之一时，或者当你希望在字符串中标识模式时，正则表达式最有用，如以下示例所示。

WARNING

如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

TIP

System.Web.RegularExpressions 命名空间包含大量正则表达式对象，这些对象实现预定义的正则表达式模式，用于分析 HTML、XML 和 ASP.NET 文档中的字符串。例如，TagRegex 类标识字符串中的开始标记，CommentRegex 类标识字符串中的 ASP.NET 注释。

示例 1：替换子字符串

假设一个邮件列表包含一些姓名，这些姓名有时包括称谓（Mr.、Mrs.、Miss 或 Ms.）以及姓氏和名字。如果你从列表中生成信封标签时不希望包括称谓，则可以使用正则表达式移除称谓，如以下示例所示。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(Mr\\.? |Mrs\\.? |Miss |Ms\\.? )";
      string[] names = { "Mr. Henry Hunt", "Ms. Sara Samuels",
                         "Abraham Adams", "Ms. Nicole Norris" };
      foreach (string name in names)
         Console.WriteLine(Regex.Replace(name, pattern, String.Empty));
   }
}
// The example displays the following output:
//    Henry Hunt
//    Sara Samuels
//    Abraham Adams
//    Nicole Norris
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(Mr\.? |Mrs\.? |Miss |Ms\.? )"
        Dim names() As String = {"Mr. Henry Hunt", "Ms. Sara Samuels", _
                                 "Abraham Adams", "Ms. Nicole Norris"}
        For Each name As String In names
            Console.WriteLine(Regex.Replace(name, pattern, String.Empty))
        Next
    End Sub
End Module
' The example displays the following output:
'    Henry Hunt
'    Sara Samuels
'    Abraham Adams
'    Nicole Norris
```

正则表达式模式 `(Mr\.? |Mrs\.? |Miss |Ms\.? )` 可匹配任何“Mr”、“Mr.”、“Mrs”、“Mrs.”、“Miss”、“Ms”或“Ms.”。对 Regex.Replace 方法的调用会将匹配的字符串替换为 String.Empty；换句话说，将其从原始字符串中移除。

示例 2：识别重复单词

意外地重复单词是编写者常犯的错误。可以使用正则表达式标识重复的单词，如以下示例所示。

```
using System;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string pattern = @"\b(\w+?)\s\1\b";
      string input = "This this is a nice day. What about this? This tastes good. I saw a a dog.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("{0} (duplicates '{1}') at position {2}",
                           match.Value, match.Groups[1].Value, match.Index);
   }
}
// The example displays the following output:
//       This this (duplicates 'This') at position 0
//       a a (duplicates 'a') at position 66
```

```
Imports System.Text.RegularExpressions

Module modMain
    Public Sub Main()
        Dim pattern As String = "\b(\w+?)\s\1\b"
        Dim input As String = "This this is a nice day. What about this? This tastes good. I saw a a dog."
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine("{0} (duplicates '{1}') at position {2}", _
                              match.Value, match.Groups(1).Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       This this (duplicates 'This') at position 0
'       a a (duplicates 'a') at position 66
```

正则表达式模式 `\b(\w+?)\s\1\b` 的解释如下:

"	"
`\b`	在单词边界处开始。
`(\w+?)`	匹配一个或多个单词字符，但字符要尽可能的少。它们一起构成可称为 `\1` 的组。
`\s`	与空白字符匹配。
`\1`	与等于名为 `\1` 的组的子字符串匹配。
`\b`	与字边界匹配。

通过将正则表达式选项设置为 Regex.Matches，调用 RegexOptions.IgnoreCase 方法。因此，匹配操作不区分大小写，此示例将子字符串“This this”标识为重复。

输入字符串包括子字符串“this? This”。但是，由于插入标点符号，该子字符串不被标识为重复。

示例 3:动态生成区分区域性的正则表达式

下面的示例演示如何将正则表达式的功能与 .NET 的全球化功能所提供的灵活性结合在一起。它使用 NumberFormatInfo 对象确定系统的当前区域性设置中货币值的格式。然后使用该信息动态构造从文本提取货

币值的正则表达式。对于每个匹配，它提取仅包含数字字符串的子组，将其转换为 Decimal 值，然后计算累计值。

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define text to be parsed.
      string input = "Office expenses on 2/13/2008:\n" +
                     "Paper (500 sheets)                    $3.95\n" +
                     "Pencils (box of 10)                   $1.00\n" +
                     "Pens (box of 10)                      $4.49\n" +
                     "Erasers                               $2.19\n" +
                     "Ink jet printer                      $69.95\n\n" +
                     "Total Expenses                      $ 81.58\n";

      // Get current culture's NumberFormatInfo object.
      NumberFormatInfo nfi = CultureInfo.CurrentCulture.NumberFormat;
      // Assign needed property values to variables.
      string currencySymbol = nfi.CurrencySymbol;
      bool symbolPrecedesIfPositive = nfi.CurrencyPositivePattern % 2 == 0;
      string groupSeparator = nfi.CurrencyGroupSeparator;
      string decimalSeparator = nfi.CurrencyDecimalSeparator;

      // Form regular expression pattern.
      string pattern = Regex.Escape( symbolPrecedesIfPositive ? currencySymbol : "") +
                       @"\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*(" +
                       Regex.Escape(decimalSeparator) + "[0-9]+)?)" +
                       (! symbolPrecedesIfPositive ? currencySymbol : "");
      Console.WriteLine( "The regular expression pattern is:");
      Console.WriteLine("   " + pattern);

      // Get text that matches regular expression pattern.
      MatchCollection matches = Regex.Matches(input, pattern,
                                              RegexOptions.IgnorePatternWhitespace);
      Console.WriteLine("Found {0} matches.", matches.Count);

      // Get numeric string, convert it to a value, and add it to List object.
      List<decimal> expenses = new List<Decimal>();

      foreach (Match match in matches)
         expenses.Add(Decimal.Parse(match.Groups[1].Value));

      // Determine whether total is present and if present, whether it is correct.
      decimal total = 0;
      foreach (decimal value in expenses)
         total += value;

      if (total / 2 == expenses[expenses.Count - 1])
         Console.WriteLine("The expenses total {0:C2}.", expenses[expenses.Count - 1]);
      else
         Console.WriteLine("The expenses total {0:C2}.", total);
   }
}
// The example displays the following output:
//       The regular expression pattern is:
//          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
//       Found 6 matches.
//       The expenses total $81.58.
```

```
Imports System.Collections.Generic
Imports System.Globalization
Imports System.Text.RegularExpressions

Public Module Example
    Public Sub Main()
        ' Define text to be parsed.
        Dim input As String = "Office expenses on 2/13/2008:" + vbCrLf + _
                              "Paper (500 sheets)                    $3.95" + vbCrLf + _
                              "Pencils (box of 10)                   $1.00" + vbCrLf + _
                              "Pens (box of 10)                      $4.49" + vbCrLf + _
                              "Erasers                               $2.19" + vbCrLf + _
                              "Ink jet printer                      $69.95" + vbCrLf + vbCrLf + _
                              "Total Expenses                     $ 81.58" + vbCrLf
        ' Get current culture's NumberFormatInfo object.
        Dim nfi As NumberFormatInfo = CultureInfo.CurrentCulture.NumberFormat
        ' Assign needed property values to variables.
        Dim currencySymbol As String = nfi.CurrencySymbol
        Dim symbolPrecedesIfPositive As Boolean = CBool(nfi.CurrencyPositivePattern Mod 2 = 0)
        Dim groupSeparator As String = nfi.CurrencyGroupSeparator
        Dim decimalSeparator As String = nfi.CurrencyDecimalSeparator

        ' Form regular expression pattern.
        Dim pattern As String = Regex.Escape(CStr(IIf(symbolPrecedesIfPositive, currencySymbol, ""))) + _
                                "\s*[-+]?" + "([0-9]{0,3}(" + groupSeparator + "[0-9]{3})*(" + _
                                Regex.Escape(decimalSeparator) + "[0-9]+)?)" + _
                                CStr(IIf(Not symbolPrecedesIfPositive, currencySymbol, ""))
        Console.WriteLine("The regular expression pattern is: ")
        Console.WriteLine("   " + pattern)

        ' Get text that matches regular expression pattern.
        Dim matches As MatchCollection = Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace)
        Console.WriteLine("Found {0} matches. ", matches.Count)

        ' Get numeric string, convert it to a value, and add it to List object.
        Dim expenses As New List(Of Decimal)

        For Each match As Match In matches
            expenses.Add(Decimal.Parse(match.Groups.Item(1).Value))
        Next

        ' Determine whether total is present and if present, whether it is correct.
        Dim total As Decimal
        For Each value As Decimal In expenses
            total += value
        Next

        If total / 2 = expenses(expenses.Count - 1) Then
            Console.WriteLine("The expenses total {0:C2}.", expenses(expenses.Count - 1))
        Else
            Console.WriteLine("The expenses total {0:C2}.", total)
        End If
    End Sub
End Module
' The example displays the following output:
'       The regular expression pattern is:
'          \$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)
'       Found 6 matches.
'       The expenses total $81.58.
```

在当前区域性设置为"英语 - 美国"(en-US) 的计算机上，该示例动态生成正则表达式 `\$\s*[-+]?([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)` 。此正则表达式模式可以按以下方式解释：

“	“
`\$`	在输入字符串中查找美元符号 (`$`) 的一个匹配项。正则表达式模式字符串包含一个反斜杠来指示按字面解释美元符号而非将其作为正则表达式定位点。(单独的 `$` 符号将指示正则表达式引擎应尝试在字符串的末尾开始匹配。)为了确保当前区域性设置的货币符号不被错误解释为正则表达式符号，该示例调用 Regex.Escape 方法使该字符转义。
`\s*`	查找空白字符的零个或多个匹配项。
`[-+]?`	查找正号或负号的零个或一个匹配项。
`([0-9]{0,3}(,[0-9]{3})*(\.[0-9]+)?)`	括起此表达式的外部括号将表达式定义为捕获组或子表达式。如果找到匹配项，则有关匹配字符串的此部分的信息可以从第二个 Group 对象中检索(该对象位于 GroupCollection 属性所返回的 Match.Groups 对象中)。(集合中的第一个元素表示整个匹配。)
`[0-9]{0,3}`	查找十进制数字 0 到 9 的零到三个匹配项。
`(,[0-9]{3})*`	查找后跟三个十进制数字的组分隔符的零个或多个匹配项。
`\.`	查找小数分隔符的一个匹配项。
`[0-9]+`	查找一个或多个十进制数字。
`(\.[0-9]+)?`	查找后跟至少一个十进制数字的小数分隔符的零个或一个匹配项。

如果在输入字符串中找到所有这些子模式，则匹配成功，并将包含有关匹配的信息的 Match 对象添加到 MatchCollection 对象。

相关主题

TITLE	“
正则表达式语言 - 快速参考	提供有关可用来定义正则表达式的字符集、运算符和构造的信息。
正则表达式对象模型	提供演示如何使用正则表达式类的信息和代码示例。
正则表达式行为的详细信息	介绍了 .NET 正则表达式的功能和行为。
在 Visual Studio 中使用正则表达式	

参考

- System.Text.RegularExpressions
- System.Text.RegularExpressions.Regex
- 正则表达式 - 快速参考(以 Word 格式下载)
- 正则表达式 - 快速参考(以 PDF 格式下载)

正则表达式语言 - 快速参考

2021/11/16 •

正则表达式是正则表达式引擎尝试匹配输入文本的一种模式。模式由一个或多个字符文本、运算符或构造组成。有关简要介绍，请参阅 .NET 正则表达式。

此快速参考中的每一节都列出了可用于定义正则表达式的字符、运算符和构造的一种特定类别。

此外，我们还以两种格式提供此信息，供你下载和打印以便参考：

- 以 Word (.docx) 格式下载
- 以 PDF (.pdf) 格式下载

字符转义

正则表达式中的反斜杠字符 (\) 指示其后跟的字符是特殊字符（如下表所示），或应按原义解释该字符。有关详细信息，请参阅字符转义。

`\a`	与报警 (bell) 符 \u0007 匹配。	`\a`	`"Error!" + '\u0007'` 中的 `"\u0007"`
`\b`	在字符类中，与退格键 \u0008 匹配。	`[\b]{3,}`	`"\b\b\b\b"` 中的 `"\b\b\b\b"`
`\t`	与制表符 \u0009 匹配。	`(\w+)\t`	`"item1\titem2\t"` 中的 `"item1\t"` 和 `"item2\t"`
`\r`	与回车符 \u000D 匹配。（`\r` 与换行符 `\n` 不是等效的。）	`\r\n(\w+)`	`"\r\nThese are\ntwo lines."` 中的 `"\r\nThese"`
`\v`	与垂直制表符 \u000B 匹配。	`[\v]{2,}`	`"\v\v\v"` 中的 `"\v\v\v"`
`\f`	与换页符 \u000C 匹配。	`[\f]{2,}`	`"\f\f\f"` 中的 `"\f\f\f"`
`\n`	与换行符 \u000A 匹配。	`\r\n(\w+)`	`"\r\nThese are\ntwo lines."` 中的 `"\r\nThese"`
`\e`	与转义符 \u001B 匹配。	`\e`	`"\x001B"` 中的 `"\x001B"`
`\` *nnn*	使用八进制表示形式指定字符（*nnn* 由二位或三位数字组成）。	`\w\040\w`	`"a bc d"` 中的 `"a b"` 和 `"c d"`
`\x` *nn*	使用十六进制表示形式指定字符（*nn* 恰好由两位数字组成）。	`\w\x20\w`	`"a bc d"` 中的 `"a b"` 和 `"c d"`

`\c` *X* `\c` x	匹配 *X* 或 *x* 指定的 ASCII 控制字符，其中 *X* 或 *x* 是控制字符的字母。	`\cC`	`"\x0003"` 中的 `"\x0003"` (Ctrl-C)
`\u` *nnnn*	使用十六进制表示形式匹配 Unicode 字符（由 *nnnn* 正确表示的四位数）。	`\w\u0020\w`	`"a bc d"` 中的 `"a b"` 和 `"c d"`
`\`	在后面带有不识别为本主题的此表和其他表中的转义符的字符时，与该字符匹配。例如，`\*` 与 `\x2A` 相同，而 `\.` 与 `\x2E` 相同。这样一来，正则表达式引擎可以区分语言元素（如 * 或 ?）和字符文本（由 `\*` 或 `\?` 表示）。	`\d+[\+-x\*]\d+`	`"(2+2) * 3*9"` 中的 `"2+2"` 和 `"3*9"`

字符类

字符类与一组字符中的任何一个字符匹配。字符类包括下表中列出的语言元素。有关更多信息，请参见 字符类。

`[` character_group `]`	匹配 *character_group* 中的任何单个字符。默认情况下，匹配区分大小写。	`[ae]`	`"gray"` 中的 `"a"`
			
			`"lane"` 中的 `"a"` 和 `"e"`

`[^` character_group `]`	求反：与不在 character_group 中的任意单个字符匹配。默认情况下，*character_group* 中的字符区分大小写。	`[^aei]`	`"reign"` 中的 `"r"`、`"g"` 和 `"n"`
`[` first `-` last `]`	字符范围：与从 *first* 到 *last* 的范围中的任意单个字符匹配。	`[A-Z]`	`"AB123"` 中的 `"A"` 和 `"B"`

`.`	通配符：与除 \n 之外的任意单个字符匹配。
	
	若要匹配文本句点字符（. 或 `\u002E`），你必须在该字符前面加上转义符（`\.`）。
		`a.e`	`"nave"` 中的 `"ave"`
			
			`"water"` 中的 `"ate"`

`\p{` *name* `}`	与 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	`\p{Lu}`
		
		`\p{IsCyrillic}`
			`"City Lights"` 中的 `"C"` 和 `"L"`
			
			`"Джем"` 中的 `"Д"` 和 `"Ж"`

`\P{` *name* `}`	与不在 *name* 指定的 Unicode 通用类别或命名块中的任何单个字符匹配。	`\P{Lu}`
		
		`\P{IsCyrillic}`
			`"City"` 中的 `"i"`、`"t"` 和 `"y"`
			
			`"ДЖem"` 中的 `"e"` 和 `"m"`

`\w`	与任何单词字符匹配。	`\w`	`"ID A1.3"` 中的 `"I"`、`"D"`、`"A"`、`"1"` 和 `"3"`
`\W`	与任何非单词字符匹配。	`\W`	`"ID A1.3"` 中的 `" "` 和 `"."`
`\s`	与任何空白字符匹配。	`\w\s`	`"ID A1.3"` 中的 `"D "`
`\S`	与任何非空白字符匹配。	`\s\S`	`"int __ctr"` 中的 `" _"`
`\d`	与任何十进制数字匹配。	`\d`	`"4 = IV"` 中的 `"4"`
`\D`	匹配不是十进制数的任意字符。	`\D`	`"4 = IV"` 中的 `" "`、`"="`、`" "`、`"I"` 和 `"V"`

定位点

定位点或原子零宽度断言会使匹配成功或失败，具体取决于字符串中的当前位置，但它们不会使引擎在字符串中前进或使用字符。下表中列出的元字符是定位点。有关详细信息，请参阅 定位点。

`^`	默认情况下，必须从字符串的开头开始匹配；在多行模式中，必须从该行的开头开始。	`^\d{3}`	`"901-333-"` 中的 `"901"`
`$`	默认情况下，匹配必须出现在字符串的末尾，或在字符串末尾的 `\n` 之前；在多行模式中，必须出现在该行的末尾之前，或在该行末尾的 `\n` 之前。	`-\d{3}$`	`"-901-333"` 中的 `"-333"`
`\A`	匹配必须出现在字符串的开头。	`\A\d{3}`	`"901-333-"` 中的 `"901"`
`\Z`	匹配必须出现在字符串的末尾或出现在字符串末尾的 `\n` 之前。	`-\d{3}\Z`	`"-901-333"` 中的 `"-333"`
`\z`	匹配必须出现在字符串的末尾。	`-\d{3}\z`	`"-901-333"` 中的 `"-333"`
`\G`	匹配必须出现在上一个匹配结束的地方。	`\G\(\d\)`	`"(1)(3)(5)[7](9)"` 中的 `"(1)"`、`"(3)"` 和 `"(5)"`

`\b`	匹配必须出现在 `\w`（字母数字）和 `\W`（非字母数字）字符之间的边界上。	`\b\w+\s\w+\b`	`"them theme them them"` 中的 `"them theme"` 和 `"them them"`
`\B`	匹配不得出现在 `\b` 边界上。	`\Bend\w*\b`	`"end sends endure lender"` 中的 `"ends"` 和 `"ender"`

分组构造

分组构造描述了正则表达式的子表达式，通常用于捕获输入字符串的子字符串。分组构造包括下表中列出的语言元素。有关详细信息，请参阅 分组构造。

`(` subexpression `)`	捕获匹配的子表达式并将其分配到一个从 1 开始的序号中。	`(\w)\1`	`"deep"` 中的 `"ee"`
`(?<` name `>` subexpression `)` 或 `(?'` name `'` subexpression `)`	将匹配的子表达式捕获到一个命名组中。	`(?<double>\w)\k<double>`	`"deep"` 中的 `"ee"`
`(?<` name1 `-` name2 `>` subexpression `)` 或 `(?'` name1 `-` name2 `'` subexpression `)`	定义平衡组定义。有关详细信息，请参阅 分组构造中的"平衡组定义"部分。	`(((?'Open'\()[^\(\)]*)+((?'Close-Open'\))[^\(\)]*)+)*(?(Open)(?!))$`	`"3+2^((1-3)*(3-1))"` 中的 `"((1-3)*(3-1))"`

`(?:` subexpression `)`	定义非捕获组。	`Write(?:Line)?`	`"Console.WriteLine()"` 中的 `"WriteLine"`
			
			`"Console.Write(value)"` 中的 `"Write"`

`(?imnsx-imnsx:` subexpression `)`	应用或禁用 *子表达式* 中指定的选项。有关详细信息，请参阅 正则表达式选项。	`A\d{2}(?i:\w+)\b`	`"A12xl A12XL a12xl"` 中的 `"A12xl"` 和 `"A12XL"`
`(?=` subexpression `)`	零宽度正预测先行断言。	`\w+(?=\.)`	`"He is. The dog ran. The sun is out."` 中的 `"is"`、`"ran"` 和 `"out"`
`(?!` subexpression `)`	零宽度负预测先行断言。	`\b(?!un)\w+\b`	`"unsure sure unity used"` 中的 `"sure"` 和 `"used"`
`(?<=` subexpression `)`	零宽度正回顾后发断言。	`(?<=19)\d{2}\b`	`"1851 1999 1950 1905 2003"` 中的 `"99"`、`"50"` 和 `"05"`

[[[[	[[	[[	[[
`(?<!` subexpression `)`	零宽度负回顾后发断言。	`(?<!19)\d{2}\b`	`"1851 1999 1950 1905 2003"` 中的 `"51"` 和 `"03"`
`(?>` subexpression `)`	原子组。	`[13579](?>A+B+)`	`"1ABB 3ABBC 5AB 5AC"` 中的 `"1ABB"`、`"3ABB"` 和 `"5AB"`

数量词

限定符指定在输入字符串中必须存在上一个元素(可以是字符、组或字符类)的多少个实例才能出现匹配项。限定符包括下表中列出的语言元素。有关更多信息，请参见 数量词。

[[[[	[[	[[	[[
`*`	匹配上一个元素零次或多次。	`\d*\.\d`	`".0"`, `"19.9"`, `"219.9"`
`+`	匹配上一个元素一次或多次。	`"be+"`	`"been"` 中的 `"bee"`、`"bent"` 中的 `"be"`
`?`	匹配上一个元素零次或一次。	`"rai?n"`	`"ran"`, `"rain"`
`{` n `}`	匹配上一个元素恰好 *n* 次。	`",\d{3}"`	`"1,043.6"` 中的 `",043"`、`"9,876,543,210"` 中的 `",876"`、`",543"` 和 `",210"`
`{` n `,}`	匹配上一个元素至少 *n* 次。	`"\d{2,}"`	`"166"`, `"29"`, `"1930"`
`{` n `,` m `}`	匹配上一个元素至少 *n* 次，但不多于 *m* 次。	`"\d{3,5}"`	`"166"`, `"17668"` `"193024"` 中的 `"19302"`
`*?`	匹配上一个元素零次或多次，但次数尽可能少。	`\d*?\.\d`	`".0"`, `"19.9"`, `"219.9"`
`+?`	匹配上一个元素一次或多次，但次数尽可能少。	`"be+?"`	`"been"` 中的 `"be"`、`"bent"` 中的 `"be"`
`??`	匹配上一个元素零次或一次，但次数尽可能少。	`"rai??n"`	`"ran"`, `"rain"`
`{` n `}?`	匹配前面的元素恰好 *n* 次。	`",\d{3}?"`	`"1,043.6"` 中的 `",043"`、`"9,876,543,210"` 中的 `",876"`、`",543"` 和 `",210"`
`{` n `,}?`	匹配上一个元素至少 *n* 次，但次数尽可能少。	`"\d{2,}?"`	`"166"`, `"29"`, `"1930"`

`{` n `,` m `}?`	匹配上一个元素的次数介于 *n* 和 *m* 之间，但次数尽可能少。	`"\d{3,5}?"`	`"166"`, `"17668"` `"193024"` 中的 `"193"` 和 `"024"`

反向引用构造

反向引用允许在同一正则表达式中随后标识以前匹配的子表达式。下表列出了 .NET 正则表达式支持的反向引用构造。有关详细信息，请参阅 反向引用构造。

`\` *number*	后向引用。匹配编号子表达式的值。	`(\w)\1`	`"seek"` 中的 `"ee"`
`\k<` *name* `>`	命名后向引用。匹配命名表达式的值。	`(?<char>\w)\k<char>`	`"seek"` 中的 `"ee"`

替换构造

替换构造用于修改正则表达式以启用 either/or 匹配。这些构造包括下表中列出的语言元素。有关详细信息，请参阅 替换构造。

`|`	匹配以竖线 (`|`) 字符分隔的任何一个元素。	`th(e|is|at)`	`"this is the day."` 中的 `"the"` 和 `"this"`
`(?(` *expression* `)` *yes* `|` *no* `)`	如果由 *expression* 指定的正则表达式模式匹配，则匹配 *yes*；否则，匹配可的 *no* 部分。*expression* 解释为零宽度的断言。	`(?(A)A\d{2}\b|\b\d{3}\b)`	`"A10 C103 910"` 中的 `"A10"` 和 `"910"`
`(?(` *name* `)` *yes* `|` *no* `)`	如果 *name* (已命名或已编号的捕获组) 具有匹项，则匹配 *yes*；否则，匹配可的 *no*。	`(?<quoted>")?(?(quoted).+?"|\S+\s)`	`"Dogs.jpg \"Yiska playing.jpg\""` 中的 `"Dogs.jpg "` 和 `"\"Yiska playing.jpg\""`

替代

替换是替换模式中支持的正则表达式语言元素。有关更多信息，请参见 替代。下表中列出的元字符是原子零宽度断言。

`$` *number*	替换按组 *number* 匹配的子字符串。	`\b(\w+)(\s)(\w+)\b`	`$3$2$1`	`"one two"`	`"two one"`

`${` *name* `}`	替换按命名组 *name* 匹配的子字符串。	`\b(?<word1>\w+)(\s)(?<word2>\w+)\b`	`${word2} ${word1}`	`"one two"`	`"two one"`
`$$`	替换字符"$"。	`\b(\d+)\s?USD`	`$$$1`	`"103 USD"`	`"$103"`
`$&`	替换整个匹配项的一个副本。	`\$?\d*\.?\d+`	`**$&**`	`"$1.30"`	`"**$1.30**"`
`$``	替换匹配前的输入字符串的所有文本。	`B+`	`$``	`"AABBCC"`	`"AAAACC"`
`$'`	替换匹配后的输入字符串的所有文本。	`B+`	`$'`	`"AABBCC"`	`"AACCCC"`
`$+`	替换最后捕获的组。	`B+(C+)`	`$+`	`"AABBCCDD"`	`"AACCDD"`
`$_`	替换整个输入字符串。	`B+`	`$_`	`"AABBCC"`	`"AAAABBCCCC"`

正则表达式选项

可以指定控制正则表达式引擎如何解释正则表达式模式的选项。其中的许多选项可以指定为内联(在正则表达式模式中)或指定为一个或多个 RegexOptions 常量。本快速参考仅列出内联选项。有关内联和 RegexOptions 选项的详细信息，请参阅文章 正则表达式选项。

可通过两种方式指定内联选项：

- 通过使用其他构造 `(?imnsx-imnsx)`，可用选项或选项组前的减号 (-) 关闭这些选项。例如，`(?i-mn)` 启用不区分大小写的匹配 (`i`)，关闭多行模式 (`m`) 并关闭未命名的组捕获 (`n`)。该选项自定义选项的点开始应用于此正则表达式，且持续有效直到模式结束或者到另一构造反转此选项的点。
- 通过使用 分组构造 `(?imnsx-imnsx:` *子表达式* `)`(只定义指定组的选项)。

.NET 正则表达式引擎支持以下内联选项：

`i`	使用不区分大小写的匹配。	`\b(?i)a(?-i)a\w+\b`	`"aardvark AAAuto aaaAuto Adam breakfast"` 中的 `"aardvark"` 和 `"aaaAuto"`
`m`	使用多行模式。`^` 和 `$` 匹配行的开头和结尾，但不匹配字符串的开头和结尾。	有关示例，请参阅 正则表达式选项中的"多行模式"部分。	
`n`	不捕获未命名的组。	有关示例，请参阅 正则表达式选项中的"仅显式捕获"部分。	

`s`	使用单行模式。	有关示例，请参阅 正则表达式选项中的"单行模式"部分。	
`x`	忽略正则表达式模式中的非转义空白。	`\b(?x) \d+ \s \w+`	`"1 aardvark 2 cats IV centurions"` 中的 `"1 aardvark"` 和 `"2 cats"`

其他构造

其他构造可修改某个正则表达式模式或提供有关该模式的信息。下表列出了 .NET 支持的其他构造。有关详细信息，请参阅 其他构造。

`(?imnsx-imnsx)`	在模式中间对诸如不区分大小写这样的选项进行设置或禁用。有关详细信息，请参阅正则表达式选项。	`\bA(?i)b\w+\b` 匹配 `"ABA Able Act"` 中的 `"ABA"` 和 `"Able"`
`(?#` comment `)`	内联注释。该注释在第一个右括号处终止。	`\bA(?#Matches words starting with A)\w+\b`
`#` [至行尾]	X 模式注释。该注释以非转义的 `#` 开头，并继续到行的结尾。	`(?x)\bA\w+\b#Matches words starting with A`

请参阅

- System.Text.RegularExpressions
- System.Text.RegularExpressions.Regex
- 正则表达式
- 正则表达式类
- 正则表达式 - 快速参考（以 Word 格式下载）
- 正则表达式 - 快速参考（以 PDF 格式下载）

正则表达式中的字符转义

2021/11/16 •

正则表达式中的反斜杠 (\) 指示以下值之一：

- 后接字符为特殊字符，如下节表中所示。例如，`\b` 是定位标记，用于指示正则表达式的匹配应从单词边界开始，`\t` 表示制表符，而 `\x020` 表示空格。
- 本应解释为未转义语言构造的字符应按字面意思进行解释。例如，大括号 (`{`) 开始定义限定符，而反斜杠后接大括号 (`\{`) 表示正则表达式引擎应匹配大括号。同样，单个反斜杠标记转义的语言构造的开始，而两个反斜杠 (`\\`) 表示正则表达式引擎应匹配反斜杠。

> **NOTE**
> 字符转义可在正则表达式模式中识别，但无法在替换模式中识别。

.NET 中的字符转义

下表列出了 .NET 中正则表达式支持的字符转义。

字符	说明
除以下字符外的所有字符： 。$ ^ {[(|)* + ?\	“字符或序列” 列中未包含的字符在正则表达式中没有特殊含义；此类字符与自身匹配。 “字符或序列” 列中包括的字符均为特殊的正则表达式语言元素。若要在正则表达式中匹配这些字符，必须将其转义或纳入正字符组。例如，正则表达式 `\$\d+` 或 `[$]\d+` 匹配“$1200”。
`\a`	匹配响铃（警报）字符，`\u0007`。
`\b`	在 `[` character_group `]` 字符类中，匹配退格，`\u0008`。（请参阅字符类。）在字符类之外，`\b` 是匹配字边界的定位点。（请参阅定位标记）
`\t`	匹配制表符，`\u0009`。
`\r`	匹配回车，`\u000D`。请注意，`\r` 不等同于换行符，`\n`。
`\v`	匹配垂直制表符，`\u000B`。
`\f`	匹配换页，`\u000C`。
`\n`	匹配换行，`\u000A`。
`\e`	匹配转义，`\u001B`。

[[[[[	[[
`\` *nnn*	匹配 ASCII 字符，其中 nnn 包含表示八进制字符代码的两位数或三位数。例如，`\040` 表示空格字符。如果此构造仅包含一个数字（如 `\2`）或者它对应捕获组的编号，则将它解释为向后引用。（请参阅向后引用构造。）
`\x` *nn*	匹配 ASCII 字符，其中 nn 是两位数的十六进制字符代码。
`\c` *X*	匹配 ASCII 控制字符，其中 X 是控制字符的字母。例如，`\cC` 为 CTRL-C。
`\u` *nnnn*	匹配的 UTF-16 代码单元，单元值是 nnnn 十六进制。■: .NET 不支持用于指定 Unicode 的 Perl 5 字符转义。Perl 5 字符转义采用以下格式 `\x{` #### `…}`，其中 #### `…` 是一系列十六进制数字。改用 `\u` nnnn 。
`\`	后接字符未识别为转义字符时，将匹配此字符。例如，`\*` 匹配星号 (*) 并等同于 `\x2A`。

示例

以下示例说明了如何使用正则表达式中的字符转义。分析了包含 2009 年世界上最大城市的名称及其人口的字符串。使用制表符（`\t`）或垂直条（| 或 `\u007c`）将每个城市名与其人口数量分开。使用回车符和换行符分隔各个城市及其人口。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string delimited = @"\G(.+)[\t\u007c](.+)\r?\n";
      string input = "Mumbai, India|13,922,125\t\n" +
                           "Shanghai, China\t13,831,900\n" +
                           "Karachi, Pakistan|12,991,000\n" +
                           "Delhi, India\t12,259,230\n" +
                           "Istanbul, Turkey|11,372,613\n";
      Console.WriteLine("Population of the World's Largest Cities, 2009");
      Console.WriteLine();
      Console.WriteLine("{0,-20} {1,10}", "City", "Population");
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, delimited))
         Console.WriteLine("{0,-20} {1,10}", match.Groups[1].Value,
                                             match.Groups[2].Value);
   }
}
// The example displays the following output:
//       Population of the World's Largest Cities, 2009
//
//       City                 Population
//
//       Mumbai, India        13,922,125
//       Shanghai, China      13,831,900
//       Karachi, Pakistan    12,991,000
//       Delhi, India         12,259,230
//       Istanbul, Turkey     11,372,613
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim delimited As String = "\G(.+)[\t\u007c](.+)\r?\n"
        Dim input As String = "Mumbai, India|13,922,125" + vbCrLf + _
                              "Shanghai, China" + vbTab + "13,831,900" + vbCrLf + _
                              "Karachi, Pakistan|12,991,000" + vbCrLf + _
                              "Delhi, India" + vbTab + "12,259,230" + vbCrLf + _
                              "Istanbul, Turkey|11,372,613" + vbCrLf
        Console.WriteLine("Population of the World's Largest Cities, 2009")
        Console.WriteLine()
        Console.WriteLine("{0,-20} {1,10}", "City", "Population")
        Console.WriteLine()
        For Each match As Match In Regex.Matches(input, delimited)
            Console.WriteLine("{0,-20} {1,10}", match.Groups(1).Value, _
                                                match.Groups(2).Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       Population of the World's Largest Cities, 2009
'
'       City                 Population
'
'       Mumbai, India        13,922,125
'       Shanghai, China      13,831,900
'       Karachi, Pakistan    12,991,000
'       Delhi, India         12,259,230
'       Istanbul, Turkey     11,372,613
```

正则表达式 `\G(.+)[\t\u007c](.+)\r?\n` 可以解释为下表中所示内容。

"	"
`\G`	从上次匹配结束处开始匹配。
`(.+)`	一次或多次匹配任何字符。这是第一个捕获组。
`[\t\u007c]`	匹配制表符 (`\t`) 或垂直条 (|)。
`(.+)`	一次或多次匹配任何字符。这是第二个捕获组。
`\r?\n`	匹配零次或一次回车符后接新行的情况。

请参阅

- 正则表达式语言 - 快速参考

正则表达式中的字符类

2021/11/16 •

一个字符类定义一组字符，其中的任一字符均可出现在输入字符串中以便成功匹配。.NET 中的正则表达式语言支持以下字符类：

- 正字符组。输入字符串中的字符必须匹配一组指定的字符中的某个字符。有关详细信息，请参阅正字符组。
- 负字符组。输入字符串中的字符不得匹配一组指定的字符中的某个字符。有关详细信息，请参阅负字符组。
- 任意字符。正则表达式中的 `.`（圆点或句点）字符是匹配除 `\n` 之外的任何字符的通配符字符。有关详细信息，请参阅任意字符。
- 通用 Unicode 类别或命名块。输入字符串中的字符必须为特定 Unicode 类别的成员，或必须位于一系列连续的 Unicode 字符中才能成功匹配。有关详细信息，请参阅 Unicode 类别或 Unicode 块。
- 负通用 Unicode 类别或命名块。输入字符串中的字符不得为特定 Unicode 类别的成员，也不得位于一系列连续的 Unicode 字符中以便成功匹配。有关详细信息，请参阅负 Unicode 类别或 Unicode 块。
- 单词字符。输入字符串中的字符可以属于适合单词中字符的任何 Unicode 类别。有关详细信息，请参阅单词字符。
- 非单词字符。输入字符串中的字符可以属于作为非单词字符的任何 Unicode 类别。有关详细信息，请参阅非单词字符。
- 空白字符。输入字符串中的字符可以是任何 Unicode 分隔符字符以及众多控制字符中的任一字符。有关详细信息，请参阅空白字符。
- 非空白字符。输入字符串中的字符可以是作为非空白字符的任何字符。有关详细信息，请参阅非空白字符。
- 十进制数字。输入字符串中的字符可以是归类为 Unicode 十进制数字的众多字符中的任一字符。有关详细信息，请参阅十进制数字字符。
- 非十进制数字。输入字符串中的字符可以是任何非 Unicode 十进制数字。有关详细信息，请参阅十进制数字字符。

.NET 支持字符类减法表达式，通过该表达式可以定义一组字符作为从一个字符类中排除另一字符类的结果。有关详细信息，请参阅字符类减法。

> **NOTE**
> 按类别匹配字符的字符类（如用于匹配字词字符的 \w，或用于匹配 Unicode 类别的 \p{}）依赖 CharUnicodeInfo 类提供字符类别信息。在 .NET Framework 4.6.2 及更高版本中，字符类别基于 Unicode 标准 8.0.0 版。

正字符组：[ ]

正字符组指定一个字符列表，其中的任何一个字符可出现在输入字符串中以便进行匹配。此字符列表可以单独指定和/或作为范围指定。

用于指定各个字符列表的语法如下所示：

```
[*character_group*]
```

其中，*character_group* 是单个字符的列表，这些字符可出现在输入字符串中以便成功匹配。character_group 可以包含一个或多个文本字符、转义字符或字符类的任意组合。

用于指定字符范围的语法如下：

```
[firstCharacter-lastCharacter]
```

其中，*firstCharacter* 是范围的开始字符，*lastCharacter* 是范围的结束字符。字符范围是通过以下方式定义的一系列连续字符：指定系列中的第一个字符，连字符 (-)，然后指定系列中的最后一个字符。如果两个字符具有相邻的 Unicode 码位，则这两个字符是连续的。firstCharacter 必须是码位较低的字符，而 lastCharacter 必须是码位较高的字符。

> **NOTE**
>
> 由于正字符组可以包含一组字符和一个字符范围，因此连字符 (`-`) 始终被解释为范围分隔符，除非它是该组的第一个或最后一个字符。

下表列出了一些常见的包含正字符类的正则表达式模式。

"	"
`[aeiou]`	匹配所有元音。
`[\p{P}\d]`	匹配所有标点符号和十进制数字字符。
`[\s\p{P}]`	匹配所有空白和标点符号。

下面的示例定义包含字符“a”和“e”的正字符组，以使输入字符串必须包含单词“grey”或“gray”且后跟另一个单词以便进行匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"gr[ae]y\s\S+?[\s\p{P}]";
      string input = "The gray wolf jumped over the grey wall.";
      MatchCollection matches = Regex.Matches(input, pattern);
      foreach (Match match in matches)
         Console.WriteLine($"'{match.Value}'");
   }
}
// The example displays the following output:
//       'gray wolf '
//       'grey wall.'
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "gr[ae]y\s\S+?[\s\p{P}]"
        Dim input As String = "The gray wolf jumped over the grey wall."
        Dim matches As MatchCollection = Regex.Matches(input, pattern)
        For Each match As Match In matches
            Console.WriteLine($"'{match.Value}'")
        Next
    End Sub
End Module
' The example displays the following output:
'       'gray wolf '
'       'grey wall.'
```

按以下方式定义正则表达式 `gr[ae]y\s\S+?[\s|\p{P}]`：

“	“
`gr`	匹配文本字符“gr”。
`[ae]`	匹配“a”或“e”。
`y\s`	匹配后跟空白字符的文本字符“y”。
`\S+?`	匹配一个或多个非空白字符（但尽可能少）。
`[\s\p{P}]`	匹配空白字符或标点符号。

下面的示例匹配以任何大写字母开头的单词。它使用子表达式 `[A-Z]` 表示从 A 到 Z 的大写字母范围。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b[A-Z]\w*\b";
      string input = "A city Albany Zulu maritime Marseilles";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       A
//       Albany
//       Zulu
//       Marseilles
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b[A-Z]\w*\b"
        Dim input As String = "A city Albany Zulu maritime Marseilles"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
```

正则表达式 `\b[A-Z]\w*\b` 的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`[A-Z]`	匹配从 A 到 Z 的所有大写字符。
`\w*`	匹配零个或多个单词字符。
`\b`	与字边界匹配。

负字符组：[^]

负字符组指定一个字符列表，这些字符不得出现在输入字符串中以便进行匹配。此字符列表可以单独指定和/或作为范围指定。

用于指定各个字符列表的语法如下所示：

```
[*^character_group*]
```

其中，*character_group* 是单个字符的列表，这些字符不可出现在输入字符串中以便成功匹配。character_group 可以包含一个或多个文本字符、转义字符或字符类的任意组合。

用于指定字符范围的语法如下：

```
[^*firstCharacter*-*lastCharacter*]
```

其中，*firstCharacter* 是范围的开始字符，*lastCharacter* 是范围的结束字符。字符范围是通过以下方式定义的一系列连续字符：指定系列中的第一个字符，连字符 (-)，然后指定系列中的最后一个字符。如果两个字符具有相邻的 Unicode 码位，则这两个字符是连续的。firstCharacter 必须是码位较低的字符，而 lastCharacter 必须是码位较高的字符。

> **NOTE**
> 由于负字符组可以包含一组字符和一个字符范围，因此连字符 (`-`) 始终被解释为范围分隔符，除非它是该组的第一个或最后一个字符。

可以连接两个或更多字符范围。例如，若要指定从“0”至“9”的十进制数范围、从“a”至“f”的小写字母范围，以及从“A”至“F”的大写字母范围，请使用 `[0-9a-fA-F]` 。

负字符组中的前导符 (`^`) 是强制的，指示字符组为负字符组，而不是正字符组。

> **IMPORTANT**
> 较大正则表达式模式中的负字符组不是零宽度断言。也就是说，在评估负字符组后，正则表达式引擎会在输入字符串中提升一个字符。

下表列出了一些常见的包含负字符组的正则表达式模式。

"	"
`[^aeiou]`	匹配除元音以外的所有字符。
`[^\p{P}\d]`	匹配标点符号和十进制数字字符以外的所有字符。

下面的示例匹配以字符“th”开头且后面不跟“o”的任何单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bth[^o]\w+\b";
      string input = "thought thing though them through thus thorough this";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       thing
//       them
//       through
//       thus
//       this
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\bth[^o]\w+\b"
        Dim input As String = "thought thing though them through thus " + _
                              "thorough this"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       thing
'       them
'       through
'       thus
'       this
```

正则表达式 `\bth[^o]\w+\b` 的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`th`	匹配文本字符“th”。
`[^o]`	与不是“o”的任何字符匹配。
`\w+`	匹配一个或多个单词字符。
`\b`	在字边界结束。

任意字符 `.`

句点字符 (.) 匹配除 `\n`（换行符 \u000A）之外的任何字符，有以下两个限制：

- 如果通过 RegexOptions.Singleline 选项修改正则表达式模式，或者通过 `.` 选项修改包含 `s` 字符类的模式的部分，则 `.` 可匹配任何字符。有关详细信息，请参阅 正则表达式选项。

  下面的示例阐释了默认情况下以及使用 `.` 选项的情况下 RegexOptions.Singleline 字符类的不同的行为。正则表达式 `^.+` 在字符串开头开始并匹配每个字符。默认情况下，匹配在第一行的结尾结束；正则表达式模式匹配回车符、`\r` 或 \u000D，但不匹配 `\n`。由于 RegexOptions.Singleline 选项将整个输入字符串解释为单行，因此它匹配输入字符串中的每个字符，包括 `\n`。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Singleline))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^.+"
        Dim input As String = "This is one line and" + vbCrLf + "this is the second."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(Regex.Escape(match.Value))
        Next
        Console.WriteLine()
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.SingleLine)
            Console.WriteLine(Regex.Escape(match.Value))
        Next
    End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r
'
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

NOTE

由于它匹配除 `\n` 之外的任何字符，因此 `.` 字符类也匹配 `\r`（回车符 \u000D）。

- 正字符组或负字符组中的句点字符将被视为原义句点字符，而非字符类。有关详细信息，请参阅本主题前面部分的正字符组和负字符组。下面的示例通过定义包括句点字符 (`.`) 的正则表达式作为字符类和正字符组的成员来进行这方面的演示。正则表达式 `\b.*[.?!;:](\s|\z)` 在字边界处开始，匹配任何字符直到遇到五个标点符号标记之一（包括句点），然后匹配空白字符或字符串的末尾。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b.*[.?!;:](\s|\z)";
      string input = "this. what: is? go, thing.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       this. what: is? go, thing.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As STring = "\b.*[.?!;:](\s|\z)"
        Dim input As String = "this. what: is? go, thing."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       this. what: is? go, thing.
```

> **NOTE**
> 由于它匹配任何字符，因此当正则表达式模式尝试多次匹配任何字符时，`.` 语言元素通常会与惰性限定符一起使用。有关更多信息，请参见 数量词。

Unicode 类别或 Unicode 块：\p{}

Unicode 标准为每个常规类别分配一个字符。例如，特定字符可以是大写字母（由 `Lu` 类别表示），十进制数字（`Nd` 类别）、数学符号（`Sm` 类别）或段落分隔符（`Zl` 类别）。Unicode 标准中的特定字符集也占据连续码位的特定区域或块。例如，可在 \u0000 和 \u007F 之间找到基本拉丁字符集，并可在 \u0600 和 \u06FF 之间找到阿拉伯语字符集。

正则表达式构造

`\p{` *name* `}`

匹配属于 Unicode 常规类别或命名块的任何字符，其中，name 是类别缩写或命名块的名称。有关类别缩写的列表，请参阅本主题稍后的支持的 Unicode 常规类别部分。有关命名块的列表，请参阅本主题稍后的支持的命名块部分。

下面的示例使用 `\p{` 名称 `}` 构造以匹配 Unicode 常规类别（在该示例中为 `Pd` 或"标点，短划线"类别）和命名块（`IsGreek` 和 `IsBasicLatin` 命名块）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+";
      string input = "Κατα Μαθθαίον - The Gospel of Matthew";

      Console.WriteLine(Regex.IsMatch(input, pattern));        // Displays True.
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+"
        Dim input As String = "Κατα Μαθθαίον - The Gospel of Matthew"

        Console.WriteLine(Regex.IsMatch(input, pattern))         ' Displays True.
    End Sub
End Module
```

正则表达式 `\b(\p{IsGreek}+(\s)?)+\p{Pd}\s(\p{IsBasicLatin}+(\s)?)+` 的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`\p{IsGreek}+`	匹配一个或多个希腊语字符。

“	“
`(\s)?`	匹配零个或一个空白字符。
`(\p{IsGreek}+(\s)?)+`	匹配一个或多个希腊语字符后跟零个或一个空白字符的模式一次或多次。
`\p{Pd}`	匹配“标点，短划线”字符。
`\s`	与空白字符匹配。
`\p{IsBasicLatin}+`	匹配一个或多个基本拉丁字符。
`(\s)?`	匹配零个或一个空白字符。
`(\p{IsBasicLatin}+(\s)?)+`	匹配一个或多个基本拉丁字符后跟零个或一个空白字符的模式一次或多次。

负 Unicode 类别或 Unicode 块：\P{}

Unicode 标准为每个常规类别分配一个字符。例如，特定字符可以是大写字母（由 `Lu` 类别表示），十进制数字（`Nd` 类别）、数学符号（`Sm` 类别）或段落分隔符（`Zl` 类别）。Unicode 标准中的特定字符集也占据连续码位的特定区域或块。例如，可在 \u0000 和 \u007F 之间找到基本拉丁字符集，并可在 \u0600 和 \u06FF 之间找到阿拉伯语字符集。

正则表达式构造

`\P{` *name* `}`

匹配不属于 Unicode 常规类别或命名块的任何字符，其中，name是类别缩写或命名块的名称。有关类别缩写的列表，请参阅本主题稍后的支持的 Unicode 常规类别部分。有关命名块的列表，请参阅本主题稍后的支持的命名块部分。

下面的示例使用 `\P{` name `}` 构造来删除数字字符串中的任何货币符号（在该示例中为 `Sc` 或“符号，货币”类别）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\P{Sc})+";

      string[] values = { "$164,091.78", "£1,073,142.68", "73¢", "€120" };
      foreach (string value in values)
         Console.WriteLine(Regex.Match(value, pattern).Value);
   }
}
// The example displays the following output:
//       164,091.78
//       1,073,142.68
//       73
//       120
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(\P{Sc})+"

        Dim values() As String = {"$164,091.78", "£1,073,142.68", "73¢", "€120"}
        For Each value As String In values
            Console.WriteLine(Regex.Match(value, pattern).Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       164,091.78
'       1,073,142.68
'       73
'       120
```

正则表达式模式 `(\P{Sc})+` 匹配不为货币符号的一个或多个字符；它有效地从结果字符串中抽出任何货币符号。

单词字符：\w

`\w` 与任何单词字符匹配。单词字符是下表中列出的任何 Unicode 类别的成员。

CATEGORY	说明
Ll	字母，小写
Lu	字母，大写
Lt	字母，首字母大写
Lo	字母，其他
Lm	字母，修饰符
Mn	标记，非间距
Nd	数字，十进制数
Pc	标点，连接符。此类别包含 10 个字符，最常用的字符是 LOWLINE 字符 (_)，u+005F。

如果指定了符合 ECMAScript 的行为，则 `\w` 等效于 `[a-zA-Z_0-9]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的“ECMAScript 匹配行为”部分。

> **NOTE**
> 由于它匹配任何单词字符，因此当正则表达式模式尝试多次匹配任何单词字符且后跟特定单词字符时，`\w` 语言元素通常会与惰性限定符一起使用。有关更多信息，请参见 数量词。

下面的示例使用 `\w` 语言元素来匹配单词中的重复字符。该示例定义可按如下方式解释的正则表达式模式 `(\w)\1`。

“	“
(\w)	匹配单词字符。这是第一个捕获组。
\1	匹配第一次捕获的值。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string[] words = { "trellis", "seer", "latter", "summer",
                         "hoarse", "lesser", "aardvark", "stunned" };
      foreach (string word in words)
      {
         Match match = Regex.Match(word, pattern);
         if (match.Success)
            Console.WriteLine("'{0}' found in '{1}' at position {2}.",
                              match.Value, word, match.Index);
         else
            Console.WriteLine("No double characters in '{0}'.", word);
      }
   }
}
// The example displays the following output:
//       'll' found in 'trellis' at position 3.
//       'ee' found in 'seer' at position 1.
//       'tt' found in 'latter' at position 2.
//       'mm' found in 'summer' at position 2.
//       No double characters in 'hoarse'.
//       'ss' found in 'lesser' at position 2.
//       'aa' found in 'aardvark' at position 0.
//       'nn' found in 'stunned' at position 3.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(\w)\1"
        Dim words() As String = {"trellis", "seer", "latter", "summer", _
                                 "hoarse", "lesser", "aardvark", "stunned"}
        For Each word As String In words
            Dim match As Match = Regex.Match(word, pattern)
            If match.Success Then
                Console.WriteLine("'{0}' found in '{1}' at position {2}.", _
                                  match.Value, word, match.Index)
            Else
                Console.WriteLine("No double characters in '{0}'.", word)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       'll' found in 'trellis' at position 3.
'       'ee' found in 'seer' at position 1.
'       'tt' found in 'latter' at position 2.
'       'mm' found in 'summer' at position 2.
'       No double characters in 'hoarse'.
'       'ss' found in 'lesser' at position 2.
'       'aa' found in 'aardvark' at position 0.
'       'nn' found in 'stunned' at position 3.
```

非单词字符:\W

`\W` 匹配任何非单词字符。\W 语言元素等效于以下字符类:

```
[^\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]
```

换言之，它与下表列出的 Unicode 类别中的字符以外的任何字符匹配。

CATEGORY	说明
Ll	字母，小写
Lu	字母，大写
Lt	字母，首字母大写
Lo	字母，其他
Lm	字母，修饰符
Mn	标记，非间距
Nd	数字，十进制数
Pc	标点，连接符。此类别包含 10 个字符，最常用的字符是 LOWLINE 字符 (_)，u+005F。

如果指定了符合 ECMAScript 的行为，则 `\W` 等效于 `[^a-zA-Z_0-9]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的"ECMAScript 匹配行为"部分。

NOTE

由于它匹配任何非单词字符，因此当正则表达式模式尝试多次匹配任何非单词字符且后跟特定非单词字符时，`\W` 语言元素通常会与惰性限定符一起使用。有关更多信息，请参见 数量词。

下面的示例阐释 `\W` 字符类。它定义正则表达式模式 `\b(\w+)(\W){1,2}`，该模式匹配后跟一个或两个非单词字符（例如，空白或标点符号）的单词。正则表达式模式可以解释为下表中所示内容。

"	"
\b	在单词边界处开始匹配。
(\w+)	匹配一个或多个单词字符。这是第一个捕获组。
(\W){1,2}	匹配非单词字符一次或两次。这是第二个捕获组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)(\W){1,2}";
      string input = "The old, grey mare slowly walked across the narrow, green pasture.";
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         Console.Write("   Non-word character(s):");
         CaptureCollection captures = match.Groups[2].Captures;
         for (int ctr = 0; ctr < captures.Count; ctr++)
             Console.Write(@"'{0}' (\u{1}){2}", captures[ctr].Value,
                           Convert.ToUInt16(captures[ctr].Value[0]).ToString("X4"),
                           ctr < captures.Count - 1 ? ", " : "");
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       The
//          Non-word character(s):' ' (\u0020)
//       old,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       grey
//          Non-word character(s):' ' (\u0020)
//       mare
//          Non-word character(s):' ' (\u0020)
//       slowly
//          Non-word character(s):' ' (\u0020)
//       walked
//          Non-word character(s):' ' (\u0020)
//       across
//          Non-word character(s):' ' (\u0020)
//       the
//          Non-word character(s):' ' (\u0020)
//       narrow,
//          Non-word character(s):',' (\u002C), ' ' (\u0020)
//       green
//          Non-word character(s):' ' (\u0020)
//       pasture.
//          Non-word character(s):'.' (\u002E)
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\w+)(\W){1,2}"
        Dim input As String = "The old, grey mare slowly walked across the narrow, green pasture."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
            Console.Write("   Non-word character(s):")
            Dim captures As CaptureCollection = match.Groups(2).Captures
            For ctr As Integer = 0 To captures.Count - 1
                Console.Write("'{0}' (\u{1}){2}", captures(ctr).Value, _
                              Convert.ToUInt16(captures(ctr).Value.Chars(0)).ToString("X4"), _
                              If(ctr < captures.Count - 1, ", ", ""))
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       The
'          Non-word character(s):' ' (\u0020)
'       old,
'          Non-word character(s):',' (\u002C), ' ' (\u0020)
'       grey
'          Non-word character(s):' ' (\u0020)
'       mare
'          Non-word character(s):' ' (\u0020)
'       slowly
'          Non-word character(s):' ' (\u0020)
'       walked
'          Non-word character(s):' ' (\u0020)
'       across
'          Non-word character(s):' ' (\u0020)
'       the
'          Non-word character(s):' ' (\u0020)
'       narrow,
'          Non-word character(s):',' (\u002C), ' ' (\u0020)
'       green
'          Non-word character(s):' ' (\u0020)
'       pasture.
'          Non-word character(s):'.' (\u002E)
```

由于第二个捕获组的 Group 对象仅包含单个捕获的非单词字符，因此该示例将从 CaptureCollection 属性返回的 Group.Captures 对象中检索所有捕获的非单词字符。

空格字符：\s

`\s` 匹配任意空格字符。它等效于下表中列出的转义序列和 Unicode 类别。

CATEGORY	匹
`\f`	窗体换页符，\u000C。
`\n`	换行符，\u000A。
`\r`	回车符，\u000D。
`\t`	制表符，\u0009。

CATEGORY	说明
`\v`	垂直制表符，\u000B。
`\x85`	省略号或 NEXT LINE (NEL) 字符 (…)，\u0085。
`\p{Z}`	匹配任何分隔符。

如果指定了符合 ECMAScript 的行为，则 `\s` 等效于 `[ \f\n\r\t\v]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的“ECMAScript 匹配行为”部分。

下面的示例阐释 `\s` 字符类。它定义正则表达式模式 `\b\w+(e)?s(\s|$)`，该模式匹配以“s”或“es”结尾且后跟一个空白字符或输入字符串末尾的单词。正则表达式模式可以解释为下表中所示内容。

模式	说明
\b	在单词边界处开始匹配。
\w+	匹配一个或多个单词字符。
(e)?	匹配“e”零次或一次。
s	匹配“s”。
(\s|$)	匹配空白字符或输入字符串的末尾。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+(e)?s(\s|$)";
      string input = "matches stores stops leave leaves";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       matches
//       stores
//       stops
//       leaves
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b\w+(e)?s(\s|$)"
        Dim input As String = "matches stores stops leave leaves"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       matches
'       stores
'       stops
'       leaves
```

非空格字符：\S

`\S` 匹配任何非空白字符。它等效于 `[^\f\n\r\t\v\x85\p{Z}]` 正则表达式模式或与等效于 `\s` 的正则表达式模式（与空白字符匹配）相反。有关详细信息，请参阅空白字符：\s。

如果指定了符合 ECMAScript 的行为，则 `\S` 等效于 `[^ \f\n\r\t\v]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的"ECMAScript 匹配行为"部分。

下面的示例阐释 `\S` 语言元素。正则表达式模式 `\b(\S+)\s?` 匹配由空白字符分隔的字符串。匹配项的 GroupCollection 对象中的第二个元素包含匹配的字符串。正则表达式可按下表中的方式解释。

"	"
`\b`	在单词边界处开始匹配。
`(\S+)`	匹配一个或多个非空白字符。这是第一个捕获组。
`\s?`	匹配零个或一个空白字符。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\S+)\s?";
      string input = "This is the first sentence of the first paragraph. " +
                           "This is the second sentence.\n" +
                           "This is the only sentence of the second paragraph.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Groups[1]);
   }
}
// The example displays the following output:
//    This
//    is
//    the
//    first
//    sentence
//    of
//    the
//    first
//    paragraph.
//    This
//    is
//    the
//    second
//    sentence.
//    This
//    is
//    the
//    only
//    sentence
//    of
//    the
//    second
//    paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\S+)\s?"
        Dim input As String = "This is the first sentence of the first paragraph. " + _
                              "This is the second sentence." + vbCrLf + _
                              "This is the only sentence of the second paragraph."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Groups(1))
        Next
    End Sub
End Module
' The example displays the following output:
'    This
'    is
'    the
'    first
'    sentence
'    of
'    the
'    first
'    paragraph.
'    This
'    is
'    the
'    second
'    sentence.
'    This
'    is
'    the
'    only
'    sentence
'    of
'    the
'    second
'    paragraph.
```

十进制数字字符：\d

`\d` 匹配任何十进制数字。它等效于 `\p{Nd}` 正则表达式模式，该模式包含标准的十进制数字 0-9 以及众多其他字符集的十进制数字。

如果指定了符合 ECMAScript 的行为，则 `\d` 等效于 `[0-9]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的“ECMAScript 匹配行为”部分。

下面的示例阐释 `\d` 语言元素。它测试输入字符串是否表示美国和加拿大的有效电话号码。正则表达式模式 `^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$` 的定义如下表所示。

“	“
`^`	从输入字符串的开头部分开始匹配。
`\(?`	匹配零个或一个“(”文本字符。
`\d{3}`	匹配三个十进制数字。
`\)?`	匹配零个或一个“)”文本字符。

“	“
`[\s-]`	匹配连字符或空白字符。
`(\(?\d{3}\)?[\s-])?`	匹配后跟三个十进制数字的可选左括号、可选右括号和空白字符或连字符零次或一次。这是第一个捕获组。
`\d{3}-\d{4}`	匹配后跟连字符和四个以上的十进制数字的三个十进制数字。
`$`	匹配输入字符串的末尾部分。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$";
      string[] inputs = { "111 111-1111", "222-2222", "222 333-444",
                          "(212) 111-1111", "111-AB1-1111",
                          "212-111-1111", "01 999-9999" };

      foreach (string input in inputs)
      {
         if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
         else
            Console.WriteLine(input + ": match failed");
      }
   }
}
// The example displays the following output:
//       111 111-1111: matched
//       222-2222: matched
//       222 333-444: match failed
//       (212) 111-1111: matched
//       111-AB1-1111: match failed
//       212-111-1111: matched
//       01 999-9999: match failed
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^(\(?\d{3}\)?[\s-])?\d{3}-\d{4}$"
        Dim inputs() As String = {"111 111-1111", "222-2222", "222 333-444", _
                                  "(212) 111-1111", "111-AB1-1111", _
                                  "212-111-1111", "01 999-9999"}

        For Each input As String In inputs
            If Regex.IsMatch(input, pattern) Then
                Console.WriteLine(input + ": matched")
            Else
                Console.WriteLine(input + ": match failed")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       111 111-1111: matched
'       222-2222: matched
'       222 333-444: match failed
'       (212) 111-1111: matched
'       111-AB1-1111: match failed
'       212-111-1111: matched
'       01 999-9999: match failed
```

非数字字符：\D

`\D` 匹配任何非数字字符。它等效于 `\P{Nd}` 正则表达式模式。

如果指定了符合 ECMAScript 的行为，则 `\D` 等效于 `[^0-9]`。有关 ECMAScript 正则表达式的信息，请参阅正则表达式选项中的“ECMAScript 匹配行为”部分。

下面的示例阐释了 \D 语言元素。它测试部件号等字符串是否包含适当的十进制和非十进制数字字符的组合。正则表达式模式 `^\D\d{1,5}\D*$` 的定义如下表所示。

“	“
`^`	从输入字符串的开头部分开始匹配。
`\D`	匹配非数字字符。
`\d{1,5}`	匹配一到五个十进制数字。
`\D*`	匹配零个、一个或多个非十进制字符。
`$`	匹配输入字符串的末尾部分。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^\D\d{1,5}\D*$";
      string[] inputs = { "A1039C", "AA0001", "C18A", "Y938518" };

      foreach (string input in inputs)
      {
         if (Regex.IsMatch(input, pattern))
            Console.WriteLine(input + ": matched");
         else
            Console.WriteLine(input + ": match failed");
      }
   }
}
// The example displays the following output:
//       A1039C: matched
//       AA0001: match failed
//       C18A: matched
//       Y938518: match failed
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^\D\d{1,5}\D*$"
        Dim inputs() As String = {"A1039C", "AA0001", "C18A", "Y938518"}

        For Each input As String In inputs
            If Regex.IsMatch(input, pattern) Then
                Console.WriteLine(input + ": matched")
            Else
                Console.WriteLine(input + ": match failed")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
```

支持的 Unicode 常规类别

Unicode 定义下表列出的常规类别。有关详细信息，请参阅 Unicode 字符数据库中的“UCD 文件格式”和“常规类别值”子主题。

CATEGORY	“
`Lu`	字母，大写
`Ll`	字母，小写
`Lt`	字母，首字母大写
`Lm`	字母，修饰符

CATEGORY	匚
`Lo`	字母，其他
`L`	所有字母字符。这包括 `Lu`、`Ll`、`Lt`、`Lm` 和 `Lo` 字符。
`Mn`	标记，非间距
`Mc`	标记，间距组合
`Me`	标记，封闭
`M`	所有音调符号标记。这包括 `Mn`、`Mc` 和 `Me` 类别。
`Nd`	数字，十进制数
`Nl`	数字，字母
`No`	数字，其他
`N`	所有数字。这包括 `Nd`、`Nl` 和 `No` 类别。
`Pc`	标点，连接符
`Pd`	标点，短划线
`Ps`	标点，开始
`Pe`	标点，结束
`Pi`	标点，前引号（根据具体使用情况，作用可能像 Ps 或 Pe）
`Pf`	标点，后引号（根据具体使用情况，作用可能像 Ps 或 Pe）
`Po`	标点，其他
`P`	所有标点字符。这包括 `Pc`、`Pd`、`Ps`、`Pe`、`Pi`、`Pf` 和 `Po` 类别。
`Sm`	符号，数学
`Sc`	符号，货币
`Sk`	符号，修饰符
`So`	符号，其他
`S`	所有符号。这包括 `Sm`、`Sc`、`Sk` 和 `So` 类别。
`Zs`	分隔符，空白

CATEGORY	说明
`Zl`	分隔符，行
`Zp`	分隔符，段落
`Z`	所有分隔符字符。这包括 `Zs`、`Zl` 和 `Zp` 类别。
`Cc`	其他，控制
`Cf`	其他，格式
`Cs`	其他，代理项
`Co`	其他，私用
`Cn`	其他，未赋值（任何字符都不具有此属性）
`C`	所有控制字符。这包括 `Cc`、`Cf`、`Cs`、`Co` 和 `Cn` 类别。

可以通过将任何特定字符传递到 GetUnicodeCategory 方法来确定该字符的 Unicode 类别。下面的示例使用 GetUnicodeCategory 方法来确定包含所选拉丁字符的数组中的每个元素的类别。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      char[] chars = { 'a', 'X', '8', ',', ' ', '\u0009', '!' };

      foreach (char ch in chars)
         Console.WriteLine("'{0}': {1}", Regex.Escape(ch.ToString()),
                           Char.GetUnicodeCategory(ch));
   }
}
// The example displays the following output:
//       'a': LowercaseLetter
//       'X': UppercaseLetter
//       '8': DecimalDigitNumber
//       ',': OtherPunctuation
//       '\ ': SpaceSeparator
//       '\t': Control
//       '!': OtherPunctuation
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim chars() As Char = {"a"c, "X"c, "8"c, ","c, " "c, ChrW(9), "!"c}

        For Each ch As Char In chars
            Console.WriteLine("'{0}': {1}", Regex.Escape(ch.ToString()), _
                              Char.GetUnicodeCategory(ch))
        Next
    End Sub
End Module
' The example displays the following output:
'       'a': LowercaseLetter
'       'X': UppercaseLetter
'       '8': DecimalDigitNumber
'       ',': OtherPunctuation
'       '\ ': SpaceSeparator
'       '\t': Control
'       '!': OtherPunctuation
```

支持的命名块

.NET 提供下表中所列的命名块。该组支持的命名块基于 Unicode 4.0 和 Perl 5.6。有关正则表达式使用命名块，请参阅 Unicode 类别或 Unicode 块：\p{} 部分。

范围	块
0000 - 007F	`IsBasicLatin`
0080 - 00FF	`IsLatin-1Supplement`
0100 - 017F	`IsLatinExtended-A`
0180 - 024F	`IsLatinExtended-B`
0250 - 02AF	`IsIPAExtensions`
02B0 - 02FF	`IsSpacingModifierLetters`
0300 - 036F	`IsCombiningDiacriticalMarks`

0370 - 03FF	`IsGreek`
	或
	`IsGreekandCoptic`

0400 - 04FF	`IsCyrillic`
0500 - 052F	`IsCyrillicSupplement`
0530 - 058F	`IsArmenian`
0590 - 05FF	`IsHebrew`

[[[[	[[[
0600 - 06FF	`IsArabic`
0700 - 074F	`IsSyriac`
0780 - 07BF	`IsThaana`
0900 - 097F	`IsDevanagari`
0980 - 09FF	`IsBengali`
0A00 - 0A7F	`IsGurmukhi`
0A80 - 0AFF	`IsGujarati`
0B00 - 0B7F	`IsOriya`
0B80 - 0BFF	`IsTamil`
0C00 - 0C7F	`IsTelugu`
0C80 - 0CFF	`IsKannada`
0D00 - 0D7F	`IsMalayalam`
0D80 - 0DFF	`IsSinhala`
0E00 - 0E7F	`IsThai`
0E80 - 0EFF	`IsLao`
0F00 - 0FFF	`IsTibetan`
1000 - 109F	`IsMyanmar`
10A0 - 10FF	`IsGeorgian`
1100 - 11FF	`IsHangulJamo`
1200 - 137F	`IsEthiopic`
13A0 - 13FF	`IsCherokee`
1400 - 167F	`IsUnifiedCanadianAboriginalSyllabics`
1680 - 169F	`IsOgham`
16A0 - 16FF	`IsRunic`

[[[[	[[[
1700 - 171F	`IsTagalog`
1720 - 173F	`IsHanunoo`
1740 - 175F	`IsBuhid`
1760 - 177F	`IsTagbanwa`
1780 - 17FF	`IsKhmer`
1800 - 18AF	`IsMongolian`
1900 - 194F	`IsLimbu`
1950 - 197F	`IsTaiLe`
19E0 - 19FF	`IsKhmerSymbols`
1D00 - 1D7F	`IsPhoneticExtensions`
1E00 - 1EFF	`IsLatinExtendedAdditional`
1F00 - 1FFF	`IsGreekExtended`
2000 - 206F	`IsGeneralPunctuation`
2070 - 209F	`IsSuperscriptsandSubscripts`
20A0 - 20CF	`IsCurrencySymbols`

20D0 - 20FF	`IsCombiningDiacriticalMarksforSymbols`
	或
	`IsCombiningMarksforSymbols`

2100 - 214F	`IsLetterlikeSymbols`
2150 - 218F	`IsNumberForms`
2190 - 21FF	`IsArrows`
2200 - 22FF	`IsMathematicalOperators`
2300 - 23FF	`IsMiscellaneousTechnical`
2400 - 243F	`IsControlPictures`
2440 - 245F	`IsOpticalCharacterRecognition`

[[[[	[[[
2460 - 24FF	`IsEnclosedAlphanumerics`
2500 - 257F	`IsBoxDrawing`
2580 - 259F	`IsBlockElements`
25A0 - 25FF	`IsGeometricShapes`
2600 - 26FF	`IsMiscellaneousSymbols`
2700 - 27BF	`IsDingbats`
27C0 - 27EF	`IsMiscellaneousMathematicalSymbols-A`
27F0 - 27FF	`IsSupplementalArrows-A`
2800 - 28FF	`IsBraillePatterns`
2900 - 297F	`IsSupplementalArrows-B`
2980 - 29FF	`IsMiscellaneousMathematicalSymbols-B`
2A00 - 2AFF	`IsSupplementalMathematicalOperators`
2B00 - 2BFF	`IsMiscellaneousSymbolsandArrows`
2E80 - 2EFF	`IsCJKRadicalsSupplement`
2F00 - 2FDF	`IsKangxiRadicals`
2FF0 - 2FFF	`IsIdeographicDescriptionCharacters`
3000 - 303F	`IsCJKSymbolsandPunctuation`
3040 - 309F	`IsHiragana`
30A0 - 30FF	`IsKatakana`
3100 - 312F	`IsBopomofo`
3130 - 318F	`IsHangulCompatibilityJamo`
3190 - 319F	`IsKanbun`
31A0 - 31BF	`IsBopomofoExtended`
31F0 - 31FF	`IsKatakanaPhoneticExtensions`

[[[[	[[[
3200 - 32FF	`IsEnclosedCJKLettersandMonths`
3300 - 33FF	`IsCJKCompatibility`
3400 - 4DBF	`IsCJKUnifiedIdeographsExtensionA`
4DC0 - 4DFF	`IsYijingHexagramSymbols`
4E00 - 9FFF	`IsCJKUnifiedIdeographs`
A000 - A48F	`IsYiSyllables`
A490 - A4CF	`IsYiRadicals`
AC00 - D7AF	`IsHangulSyllables`
D800 - DB7F	`IsHighSurrogates`
DB80 - DBFF	`IsHighPrivateUseSurrogates`
DC00 - DFFF	`IsLowSurrogates`
E000 - F8FF	`IsPrivateUse` 或 `IsPrivateUseArea`
F900 - FAFF	`IsCJKCompatibilityIdeographs`
FB00 - FB4F	`IsAlphabeticPresentationForms`
FB50 - FDFF	`IsArabicPresentationForms-A`
FE00 - FE0F	`IsVariationSelectors`
FE20 - FE2F	`IsCombiningHalfMarks`
FE30 - FE4F	`IsCJKCompatibilityForms`
FE50 - FE6F	`IsSmallFormVariants`
FE70 - FEFF	`IsArabicPresentationForms-B`
FF00 - FFEF	`IsHalfwidthandFullwidthForms`
FFF0 - FFFF	`IsSpecials`

字符类减法：[base_group - [excluded_group]]

一个字符类定义一组字符。字符类减法将产生一组字符，该组字符是从一个字符类中排除另一个字符类中的字符的结果。

字符类减法表达式具有以下形式：

`[` *base_group* `-[` *excluded_group* `]]`

方括号 (`[]`) 和连字符 (`-`) 是强制的。base_group 是正字符组或负字符组。*excluded_group* 部分是另一个正字符组或负字符组，或者是另一个字符类减法表达式（即，可以嵌套字符类减法表达式）。

例如，假设你有一个由从“a”至“z”范围内的字符组成的基本组。若要定义由除字符“m”之外的基本组组成的字符集，请使用 `[a-z-[m]]`。若要定义由除字符集“d”、“j”和“p”之外的基本组组成的字符集，请使用 `[a-z-[djp]]`。若要定义由除从“m”至“p”字符范围之外的基本组组成的字符集，请使用 `[a-z-[m-p]]`。

可考虑使用嵌套字符类减法表达式 `[a-z-[d-w-[m-o]]]`。该表达式由最里面的字符范围向外计算。首先，在从“d”至“w”的字符范围中减去从“m”至“o”的字符范围，这将产生从“d”至“l”和从“p”至“w”的字符集。然后，在从“a”至“z”的字符范围中减去该集合，这将产生字符集 `[abcmnoxyz]`。

可以将任何字符类用于字符类减法。若要定义字符集，且该字符集包括除空白字符 (`\s`)、标点通用类别中的字符 (`\p{P}`)、`IsGreek` 命名块中的字符 (`\p{IsGreek}`) 以及 Unicode NEXT LINE 控制字符 (\x85) 之外的所有从 \u0000 至 \uFFFF 的 Unicode 字符，请使用 `[\u0000-\uFFFF-[\s\p{P}\p{IsGreek}\x85]]`。

为字符类减法表达式选择将会产生有用结果的字符类。避免使用产生空字符集的表达式，这将无法匹配任何内容，同时避免使用等效于初始基本组的表达式。例如，表达式 `[\p{IsBasicLatin}-[\x00-\x7F]]` 从 `IsBasicLatin` 常规类别中减去 `IsBasicLatin` 字符范围内的所有字符，其结果为空集合。类似地，表达式 `[a-z-[0-9]]` 的结果为初始基本组。这是因为，基本组（它是从“a”至“z”的字母组成的字符范围）不包含排除组（它是从“0”至“9”的十进制数组成的字符范围）中的任何字符。

下面的示例定义正则表达式 `^[0-9-[2468]]+$`，该表达式匹配输入字符串中的零和奇数。正则表达式模式可以解释为下表中所示内容。

“	“
^	从输入字符串的开头处开始进行匹配。
`[0-9-[2468]]+`	匹配任意字符（从 0 到 9，除了 2、4、6 和 8 之外）的一个或多个匹配项。换句话说，匹配零或奇数的一个或多个匹配项。
$	在输入字符串末尾结束匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "123", "13579753", "3557798", "335599901" };
      string pattern = @"^[0-9-[2468]]+$";

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
      }
   }
}
// The example displays the following output:
//       13579753
//       335599901
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"123", "13579753", "3557798", "335599901"}
        Dim pattern As String = "^[0-9-[2468]]+$"

        For Each input As String In inputs
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       13579753
'       335599901
```

请参阅

- GetUnicodeCategory
- **正则表达式**语言 - **快速参考**
- **正则表达式**选项

正则表达式中的定位点

2021/11/16 •

定位点(原子零宽度断言)指定字符串中必须出现匹配的位置。在搜索表达式中使用定位点时，正则表达式引擎不在字符串中前进或使用字符，它仅在指定位置查找匹配。例如，`^` 指定必须从行或字符串的开头开始匹配。因此，正则表达式 `^http:` 仅当 "http:" 出现在行开头时才与之匹配。下表列出了 .NET 中正则表达式支持的定位点。

`^`	默认情况下，匹配必须出现在字符串的开头；在多行模式中，必须出现在该行的开头。有关详细信息，请参阅 字符串或行的开头。
`$`	默认情况下，匹配必须出现在字符串的末尾，或在字符串末尾的 `\n` 之前；在多行模式中，必须出现在该行的末尾，或在该行末尾的 `\n` 之前。有关详细信息，请参阅 字符串或行的末尾。
`\A`	匹配必须仅出现在字符串的开头位置(无多行支持)。有关详细信息，请参阅 仅字符串的开头。
`\Z`	匹配必须出现在字符串的末尾，或出现在字符串末尾的 `\n` 之前。有关详细信息，请参阅 字符串的末尾或结束换行之前。
`\z`	匹配必须仅出现在字符串的末尾。有关详细信息，请参阅 仅字符串的末尾。
`\G`	匹配必须从上一个匹配结束的位置开始。有关详细信息，请参阅 连续匹配。
`\b`	匹配必须出现在字边界。有关详细信息，请参阅 字边界。
`\B`	匹配不得出现在字边界上。有关详细信息，请参阅 非字边界。

字符串或行的开头：^

默认情况下，`^` 定位点指定以下模式必须从字符串的第一个字符位置开始。如果结合使用 `^` 与 RegexOptions.Multiline 选项(请参阅正则表达式选项)，匹配必须出现在每行的开头。

以下示例在正则表达式中使用 `^` 定位点，可提取有关某些职业棒球队存在年限的信息。该示例调用 Regex.Matches 方法的两个重载：

- 调用 Matches(String, String) 重载仅找到输入字符串中与正则表达式模式匹配的第一个子字符串。
- 调用 Matches(String, String, RegexOptions) 重载，并将 `options` 参数设置为 RegexOptions.Multiline 可找到所有五个子字符串。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957\n" +
                       "Chicago Cubs, National League, 1903-present\n" +
                       "Detroit Tigers, American League, 1901-present\n" +
                       "New York Giants, National League, 1885-1957\n" +
                       "Washington Senators, American League, 1901-1960\n";
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";
        Match match;

        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf +
                              "Chicago Cubs, National League, 1903-present" + vbCrLf +
                              "Detroit Tigers, American League, 1901-present" + vbCrLf +
                              "New York Giants, National League, 1885-1957" + vbCrLf +
                              "Washington Senators, American League, 1901-1960" + vbCrLf

        Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"
        Dim match As Match

        match = Regex.Match(input, pattern)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            match = match.NextMatch()
        Loop
        Console.WriteLine()

        match = Regex.Match(input, pattern, RegexOptions.Multiline)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            match = match.NextMatch()
        Loop
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
```

正则表达式模式 `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+` 的定义如下表所示。

`^`	从输入字符串的开头开始匹配(如果在调用该方法时选择了 RegexOptions.Multiline 选项，则从行的开头开始匹配)。
`((\w+(\s?)){2,}`	匹配至少两次后跟零个或一个空格的一个或多个单词字符。这是第一个捕获组。此表达式还定义第二个和第三个捕获组：第二个捕获组由捕获的单词组成，第三个捕获组由捕获的空格组成。
`,\s`	匹配后跟一个空白字符的逗号。
`(\w+\s\w+)`	匹配后跟一个空格再后跟一个或多个单词字符的一个或多个单词字符。这是第四个捕获组。

"	"

`,`	匹配逗号。
`\s\d{4}`	匹配后跟四个十进制数字的空格。
`(-(\d{4}|present))?`	匹配连字符后跟四个十进制数字或字符串“present”的零或一个匹配项。这是第六个捕获组。还包括第七个捕获组。
`,?`	匹配逗号的零个或一个匹配项。
`(\s\d{4}(-(\d{4}|present))?,?)+`	匹配以下内容的一个或多个匹配项：空格、四个十进制数字、连字符后跟四个十进制数字或字符串“present”的零个或一个匹配项以及零个或一个逗号。这是第五个捕获组。

字符串或行的末尾：$

`$` 定位点指定前面的模式必须出现在输入字符串的末尾，或出现在输入字符串末尾的 `\n` 之前。

如果结合使用 `$` 与 RegexOptions.Multiline 选项，则匹配也可能出现在行的末尾。请注意，`$` 与 `\n` 匹配但与 `\r\n`（回车符和换行符的组合，或称 CR/LF）不匹配。若要匹配 CR/LF 字符组合，请将 `\r?$` 包括到正则表达式模式中。

以下示例将 `$` 定位点添加到 字符串或行的开头 部分的示例中所使用的正则表达式模式中。配合包括五行文本的原始输入字符串使用时，Regex.Matches(String, String) 方法找不到匹配项，因为第一行的末尾与 `$` 模式不匹配。当原始输入字符串被拆分为字符串数组时，Regex.Matches(String, String) 方法会成功匹配五行中的每一行。如果调用 Regex.Matches(String, String, RegexOptions) 方法时将 `options` 参数设置为 RegexOptions.Multiline，则找不到匹配项，因为正则表达式模式不考虑回车符元素 (\u+000D)。但是，如果通过用将 `$` 替换成 `\r?$` 修改了正则表达式模式，则调用 Regex.Matches(String, String, RegexOptions) 方法并将 `options` 参数设置为 RegexOptions.Multiline 将再次找到五个匹配项。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string cr = Environment.NewLine;
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + cr +
                       "Chicago Cubs, National League, 1903-present" + cr +
                       "Detroit Tigers, American League, 1901-present" + cr +
                       "New York Giants, National League, 1885-1957" + cr +
                       "Washington Senators, American League, 1901-1960" + cr;
        Match match;

        string basePattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";
        string pattern = basePattern + "$";
        Console.WriteLine("Attempting to match the entire input string:");
        match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);
```

```
                Console.WriteLine(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        string[] teams = input.Split(new String[] { cr }, StringSplitOptions.RemoveEmptyEntries);
        Console.WriteLine("Attempting to match each element in a string array:");
        foreach (string team in teams)
        {
            match = Regex.Match(team, pattern);
            if (match.Success)
            {
                Console.Write("The {0} played in the {1} in",
                              match.Groups[1].Value, match.Groups[4].Value);
                foreach (Capture capture in match.Groups[5].Captures)
                    Console.Write(capture.Value);
                Console.WriteLine(".");
            }
        }
        Console.WriteLine();

        Console.WriteLine("Attempting to match each line of an input string with '$':");
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();

        pattern = basePattern + "\r?$";
        Console.WriteLine(@"Attempting to match each line of an input string with '\r?$':");
        match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    Attempting to match the entire input string:
//
//    Attempting to match each element in a string array:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
//
//    Attempting to match each line of an input string with '$':
//
//    Attempting to match each line of an input string with '\r?$':
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
//    The Chicago Cubs played in the National League in 1903-present.
//    The Detroit Tigers played in the American League in 1901-present.
//    The New York Giants played in the National League in 1885-1957.
```

```
//    The New York Giants played in the National League in 1885-1957.
//    The Washington Senators played in the American League in 1901-1960.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf +
                              "Chicago Cubs, National League, 1903-present" + vbCrLf +
                              "Detroit Tigers, American League, 1901-present" + vbCrLf +
                              "New York Giants, National League, 1885-1957" + vbCrLf +
                              "Washington Senators, American League, 1901-1960" + vbCrLf

        Dim basePattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"
        Dim match As Match

        Dim pattern As String = basePattern + "$"
        Console.WriteLine("Attempting to match the entire input string:")
        match = Regex.Match(input, pattern)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            match = match.NextMatch()
        Loop
        Console.WriteLine()

        Dim teams() As String = input.Split(New String() {vbCrLf}, StringSplitOptions.RemoveEmptyEntries)
        Console.WriteLine("Attempting to match each element in a string array:")
        For Each team As String In teams
            match = Regex.Match(team, pattern)
            If match.Success Then
                Console.Write("The {0} played in the {1} in",
                              match.Groups(1).Value, match.Groups(4).Value)
                For Each capture As Capture In match.Groups(5).Captures
                    Console.Write(capture.Value)
                Next
                Console.WriteLine(".")
            End If
        Next
        Console.WriteLine()

        Console.WriteLine("Attempting to match each line of an input string with '$':")
        match = Regex.Match(input, pattern, RegexOptions.Multiline)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            match = match.NextMatch()
        Loop
        Console.WriteLine()

        pattern = basePattern + "\r?$"
        Console.WriteLine("Attempting to match each line of an input string with '\r?$':")
        match = Regex.Match(input, pattern, RegexOptions.Multiline)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
```

```
            match = match.NextMatch()
        Loop
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'    Attempting to match the entire input string:
'
'    Attempting to match each element in a string array:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
'
'    Attempting to match each line of an input string with '$':
'
'    Attempting to match each line of an input string with '\r?$':
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
'    The Chicago Cubs played in the National League in 1903-present.
'    The Detroit Tigers played in the American League in 1901-present.
'    The New York Giants played in the National League in 1885-1957.
'    The Washington Senators played in the American League in 1901-1960.
```

仅字符串的开头:\A

`\A` 定位点指定匹配必须出现在输入字符串的开头。它等同于 `^` 定位点，只不过 `\A` 忽略了 RegexOptions.Multiline 选项。因此，在多行的输入字符串中，它只能匹配第一行的开头。

以下示例与 `^` 和 `$` 定位点的示例类似。它在正则表达式中使用 `\A` 定位点，可提取有关某些职业棒球队存在年限的信息。输入字符串包括五行。调用 Regex.Matches(String, String, RegexOptions) 方法仅找到输入字符串中与正则表达式模式匹配的第一个子字符串。如示例所示， Multiline 选项不起作用。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957\n" +
                       "Chicago Cubs, National League, 1903-present\n" +
                       "Detroit Tigers, American League, 1901-present\n" +
                       "New York Giants, National League, 1885-1957\n" +
                       "Washington Senators, American League, 1901-1960\n";

        string pattern = @"\A((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+";

        Match match = Regex.Match(input, pattern, RegexOptions.Multiline);
        while (match.Success)
        {
            Console.Write("The {0} played in the {1} in",
                          match.Groups[1].Value, match.Groups[4].Value);
            foreach (Capture capture in match.Groups[5].Captures)
                Console.Write(capture.Value);

            Console.WriteLine(".");
            match = match.NextMatch();
        }
        Console.WriteLine();
    }
}
// The example displays the following output:
//    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957" + vbCrLf +
                              "Chicago Cubs, National League, 1903-present" + vbCrLf +
                              "Detroit Tigers, American League, 1901-present" + vbCrLf +
                              "New York Giants, National League, 1885-1957" + vbCrLf +
                              "Washington Senators, American League, 1901-1960" + vbCrLf

        Dim pattern As String = "\A((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+"

        Dim match As Match = Regex.Match(input, pattern, RegexOptions.Multiline)
        Do While match.Success
            Console.Write("The {0} played in the {1} in",
                          match.Groups(1).Value, match.Groups(4).Value)
            For Each capture As Capture In match.Groups(5).Captures
                Console.Write(capture.Value)
            Next
            Console.WriteLine(".")
            match = match.NextMatch()
        Loop
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'    The Brooklyn Dodgers played in the National League in 1911, 1912, 1932-1957.
```

字符串末尾或结束换行之前:\Z

`\Z` 定位点指定匹配项必须出现在输入字符串的末尾，或出现在输入字符串末尾的 `\n` 之前。它等同于 `$` 定位

点，只不过 `\z` 忽略了 RegexOptions.Multiline 选项。因此，在多行字符串中，它只能匹配最后一行的末尾，或 `\n` 前的最后一行。

请注意，`\Z` 与 `\n` 匹配但与 `\r\n`（CR/LF 字符组合）不匹配。若要匹配 CR/LF，请将 `\r?\Z` 包括到正则表达式模式中。

以下示例在正则表达式中使用 `\Z` 定位点，与 字符串或行的开头 部分的示例类似，可提取有关某些职业棒球队存在年限的信息。正则表达式 `\r?\Z` 中的子表达式 `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z` 匹配字符串的末尾，也匹配以 `\n` 或 `\r\n` 结尾的字符串。因此，数组中的每个元素都与正则表达式模式匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                            "Chicago Cubs, National League, 1903-present" + Environment.NewLine,
                            "Detroit Tigers, American League, 1901-present" + Regex.Unescape(@"\n"),
                            "New York Giants, National League, 1885-1957",
                            "Washington Senators, American League, 1901-1960" + Environment.NewLine};
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z";

        foreach (string input in inputs)
        {
            Console.WriteLine(Regex.Escape(input));
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("   Match succeeded.");
            else
                Console.WriteLine("   Match failed.");
        }
    }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match succeeded.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match succeeded.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match succeeded.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                                  "Chicago Cubs, National League, 1903-present" + vbCrLf,
                                  "Detroit Tigers, American League, 1901-present" + vbLf,
                                  "New York Giants, National League, 1885-1957",
                                  "Washington Senators, American League, 1901-1960" + vbCrLf}
        Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\Z"

        For Each input As String In inputs
            Console.WriteLine(Regex.Escape(input))
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then
                Console.WriteLine("   Match succeeded.")
            Else
                Console.WriteLine("   Match failed.")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
'       Match succeeded.
'    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
'       Match succeeded.
'    Detroit\ Tigers,\ American\ League,\ 1901-present\n
'       Match succeeded.
'    New\ York\ Giants,\ National\ League,\ 1885-1957
'       Match succeeded.
'    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
'       Match succeeded.
```

仅字符串末尾：\z

`\z` 定位点指定匹配必须出现在输入字符串末尾。与 `$` 语言元素类似，`\z` 忽略了 RegexOptions.Multiline 选项。与 `\Z` 语言元素不同，`\z` 不匹配字符串末尾的 `\n` 字符。因此，它只能匹配输入字符串的最后一行。

以下示例在正则表达式中使用 `\z` 定位点，与上一部分的示例中使用的定位点在其他方面相同，用于提取有关某些职业棒球队存在年限的信息。此示例尝试使用正则表达式模式 `^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z` 匹配字符串数组中五个元素的每一个。两个字符串以回车符和换行符结尾，一个字符串以换行符结尾，另外两个既不以回车符也不以换行符结尾。如输出所示，只有不包含回车符或换行符的字符串与模式匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string[] inputs = { "Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                            "Chicago Cubs, National League, 1903-present" + Environment.NewLine,
                            "Detroit Tigers, American League, 1901-present\n",
                            "New York Giants, National League, 1885-1957",
                            "Washington Senators, American League, 1901-1960" + Environment.NewLine };
        string pattern = @"^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z";

        foreach (string input in inputs)
        {
            Console.WriteLine(Regex.Escape(input));
            Match match = Regex.Match(input, pattern);
            if (match.Success)
                Console.WriteLine("   Match succeeded.");
            else
                Console.WriteLine("   Match failed.");
        }
    }
}
// The example displays the following output:
//    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
//       Match succeeded.
//    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
//       Match failed.
//    Detroit\ Tigers,\ American\ League,\ 1901-present\n
//       Match failed.
//    New\ York\ Giants,\ National\ League,\ 1885-1957
//       Match succeeded.
//    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
//       Match failed.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"Brooklyn Dodgers, National League, 1911, 1912, 1932-1957",
                                  "Chicago Cubs, National League, 1903-present" + vbCrLf,
                                  "Detroit Tigers, American League, 1901-present" + vbLf,
                                  "New York Giants, National League, 1885-1957",
                                  "Washington Senators, American League, 1901-1960" + vbCrLf}
        Dim pattern As String = "^((\w+(\s?)){2,}),\s(\w+\s\w+),(\s\d{4}(-(\d{4}|present))?,?)+\r?\z"

        For Each input As String In inputs
            Console.WriteLine(Regex.Escape(input))
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then
                Console.WriteLine("   Match succeeded.")
            Else
                Console.WriteLine("   Match failed.")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'    Brooklyn\ Dodgers,\ National\ League,\ 1911,\ 1912,\ 1932-1957
'       Match succeeded.
'    Chicago\ Cubs,\ National\ League,\ 1903-present\r\n
'       Match failed.
'    Detroit\ Tigers,\ American\ League,\ 1901-present\n
'       Match failed.
'    New\ York\ Giants,\ National\ League,\ 1885-1957
'       Match succeeded.
'    Washington\ Senators,\ American\ League,\ 1901-1960\r\n
'       Match failed.
```

连续匹配：\G

`\G` 定位符指定匹配必须出现在上一个匹配结束的点。将此定位点与 Regex.Matches 或 Match.NextMatch 方法配合使用时，它可确保所有匹配项是连续的。

以下示例使用正则表达式从一个以逗号分隔的字符串中提取啮齿类动物的名称。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "capybara,squirrel,chipmunk,porcupine,gopher," +
                       "beaver,groundhog,hamster,guinea pig,gerbil," +
                       "chinchilla,prairie dog,mouse,rat";
        string pattern = @"\G(\w+\s?\w*),?";
        Match match = Regex.Match(input, pattern);
        while (match.Success)
        {
            Console.WriteLine(match.Groups[1].Value);
            match = match.NextMatch();
        }
    }
}
// The example displays the following output:
//       capybara
//       squirrel
//       chipmunk
//       porcupine
//       gopher
//       beaver
//       groundhog
//       hamster
//       guinea pig
//       gerbil
//       chinchilla
//       prairie dog
//       mouse
//       rat
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "capybara,squirrel,chipmunk,porcupine,gopher," +
                              "beaver,groundhog,hamster,guinea pig,gerbil," +
                              "chinchilla,prairie dog,mouse,rat"
        Dim pattern As String = "\G(\w+\s?\w*),?"
        Dim match As Match = Regex.Match(input, pattern)
        Do While match.Success
            Console.WriteLine(match.Groups(1).Value)
            match = match.NextMatch()
        Loop
    End Sub
End Module
' The example displays the following output:
'       capybara
'       squirrel
'       chipmunk
'       porcupine
'       gopher
'       beaver
'       groundhog
'       hamster
'       guinea pig
'       gerbil
'       chinchilla
'       prairie dog
'       mouse
'       rat
```

正则表达式 `\G(\w+\s?\w*),?` 可以解释为下表中所示内容。

"	"
`\G`	从上次匹配结束的位置开始。
`\w+`	匹配一个或多个单词字符。
`\s?`	匹配零个或一个空格。
`\w*`	匹配零个或多个单词字符。
`(\w+\s?\w*)`	匹配后跟零个或一个空格再后跟零个或多个单词字符的一个或多个单词字符。这是第一个捕获组。
`,?`	匹配文本逗号字符的零个或一个匹配项。

字边界:\b

`\b` 定位符指定匹配必须出现单词字符(`\w` 语言元素)和非单词字符(`\W` 语言元素)之间的边界上。单词字符包括字母数字字符和下划线;非单词字符包括不为字母数字字符或下划线的任何字符。(有关详细信息,请参阅字符类。)匹配也可以出现在字符串开头或结尾处的单词边界上。

`\b` 定位点经常用于确保子表达式与整个单词而不仅与单词的开头或结尾匹配。以下示例中的正则表达式 `\bare\w*\b` 阐释了这种用法。它与任何以子字符串"are"开头的单词匹配。该示例的输出也阐释了 `\b` 与输入字符串的开头和结尾均匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "area bare arena mare";
        string pattern = @"\bare\w*\b";
        Console.WriteLine("Words that begin with 'are':");
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("'{0}' found at position {1}",
                              match.Value, match.Index);
    }
}
// The example displays the following output:
//       Words that begin with 'are':
//       'area' found at position 0
//       'arena' found at position 10
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "area bare arena mare"
        Dim pattern As String = "\bare\w*\b"
        Console.WriteLine("Words that begin with 'are':")
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("'{0}' found at position {1}",
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Words that begin with 'are':
'       'area' found at position 0
'       'arena' found at position 10
```

正则表达式模式可以解释为下表中所示内容。

“	“
`\b`	在单词边界处开始匹配。
`are`	匹配子字符串“are”。
`\w*`	匹配零个或多个单词字符。
`\b`	在单词边界处结束匹配。

非字边界:\B

`\B` 定位符指定匹配不得出现在单词边界上。它与 `\b` 定位点截然相反。

以下示例使用 `\B` 定位点定位单词中的子字符串“qu”匹配项。正则表达式模式 `\Bqu\w+` 与以“qu”开头（但“qu”并不位于单词之首）且延续到单词末尾的子字符串匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    public static void Main()
    {
        string input = "equity queen equip acquaint quiet";
        string pattern = @"\Bqu\w+";
        foreach (Match match in Regex.Matches(input, pattern))
            Console.WriteLine("'{0}' found at position {1}",
                              match.Value, match.Index);
    }
}
// The example displays the following output:
//       'quity' found at position 1
//       'quip' found at position 14
//       'quaint' found at position 21
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "equity queen equip acquaint quiet"
        Dim pattern As String = "\Bqu\w+"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("'{0}' found at position {1}",
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       'quity' found at position 1
'       'quip' found at position 14
'       'quaint' found at position 21
```

正则表达式模式可以解释为下表中所示内容。

“	“
`\B`	不在单词边界处开始匹配。
`qu`	匹配子字符串“qu”。
`\w+`	匹配一个或多个单词字符。

请参阅

- 正则表达式语言 - 快速参考
- 正则表达式选项

正则表达式中的分组构造

2021/11/16 •

分组构造描述了正则表达式的子表达式，用于捕获输入字符串的子字符串。你可以使用分组构造来完成下列任务：

- 匹配输入字符串中重复的子表达式。
- 将限定符应用于拥有多个正则表达式语言元素的子表达式。有关限定符的更多信息，请参见 Quantifiers。
- 包括由 Regex.Replace 和 Match.Result 方法返回的字符串的子表达式。
- 从 Match.Groups 属性中检索各个子表达式，并分别从匹配的文本作为一个整体处理它们。

下表列出了 .NET 正则表达式引擎支持的分组构造，并指明它们是捕获构造，还是非捕获构造。

tttt	tttttt
匹配的子表达式	捕获
命名匹配的子表达式	捕获
平衡组定义	捕获
非捕获组	非捕获
组选项	非捕获
零宽度正预测先行断言	非捕获
零宽度负预测先行断言	非捕获
零宽度正回顾后发断言	非捕获
零宽度负回顾后发断言	非捕获
原子组	非捕获

有关组和正则表达式对象模型的信息，请参见 分组构造和正则表达式对象。

匹配的子表达式

以下分组构造捕获匹配的子表达式：

`(` subexpression `)`

其中 *子表达式* 为任何有效正则表达式模式。使用括号的捕获按正则表达式中左括号的顺序从一开始从左到右自动编号。捕获元素编号为零的捕获是由整个正则表达式模式匹配的文本。

NOTE

默认情况下，`(` *子表达式* `)` 语言元素捕获匹配的子表达式。但是，如果正则表达式模式匹配方法的 RegexOptions 参数包含 RegexOptions.ExplicitCapture 标志，或者如果 `n` 选项应用于此子表达式（参见本主题后面的 组选项 ），则不会捕获匹配的子表达式。

可以四种方法访问捕获的组：

- 通过使用正则表达式中的反向引用构造。使用语法 `\` *数字* 在同一正则表达式中引用匹配的子表达式，其中 *数字* 是捕获的表达式的初始数字。
- 通过使用正则表达式中的命名的反向引用构造。使用语法 `\k<` *name* `>` 在同一正则表达式中引用匹配的子表达式，其中 *name* 是捕获组的名称，或使用 `\k<` *数字* `>` 在同一正则表达式中引用匹配的子表达式，其中 *数字* 是捕获组的初始数字。捕获组具有与其原始编号相同的默认名称。有关更多信息，请参见本主题后面的 命名匹配的子表达式 。
- 通过使用 `$` *数字* $ Regex.Replace number Match.Result 替换序列，其中 *数字* 是捕获的表达式的初始数字。
- 以编程的方式，通过使用 GroupCollection 对象的方式，该对象由 Match.Groups 属性返回。集合中位置零上的成员表示正则表达式匹配。每个后续成员表示匹配的子表达式。有关更多信息，请参见 分组构造和正则表达式对象 一节。

以下示例阐释表示文本中重复单词的正则表达式。正则表达式模式的两个捕获组表示重复的单词的两个实例。捕获第二个实例，以报告它在输入字符串的起始位置。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w+)\s(\1)\W";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("Duplicate '{0}' found at positions {1} and {2}.",
                           match.Groups[1].Value, match.Groups[1].Index, match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'that' found at positions 8 and 13.
//       Duplicate 'the' found at positions 22 and 26.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(\w+)\s(\1)\W"
        Dim input As String = "He said that that was the the correct answer."
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine("Duplicate '{0}' found at positions {1} and {2}.", _
                              match.Groups(1).Value, match.Groups(1).Index, match.Groups(2).Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Duplicate 'that' found at positions 8 and 13.
'       Duplicate 'the' found at positions 22 and 26.
```

正则表达式模式为：

```
(\w+)\s(\1)\W
```

下表演示了如何解释正则表达式模式。

"	"
`(\w+)`	匹配一个或多个单词字符。这是第一个捕获组。
`\s`	与空白字符匹配。
`(\1)`	与第一个捕获组捕获中的字符串匹配。这是第二个捕获组。该示例将其指定到捕获组上，以便可从 `Match.Index` 属性返回。
`\W`	匹配包括空格和标点符号的一个非单词字符。这样可以防止正则表达式模式匹配以第一个捕获组的单词开头的单词。

命名匹配的子表达式

以下分组构造捕获匹配的子表达式，并允许你按名称或编号访问它：

```
(?<name>subexpression)
```

或：

```
(?'name'subexpression)
```

其中 *名称* 是有效的组名称，而 *子表达式* 是任何有效的正则表达式模式。*名称* 不得包含任何标点符号字符，并且不能以数字开头。

> **NOTE**
> 如果正则表达式模式匹配方法的 RegexOptions 参数包含 RegexOptions.ExplicitCapture 标志，或者如果 `n` 选项应用于此子表达式（参见本主题后面的 组选项 ），则捕获子表达式的唯一方法就是显式命名捕获组。

可用以下方式访问已命名的捕获组：

- 通过使用正则表达式中的命名的反向引用构造。使用语法 `\k<`*name*`>` 在同一正则表达式中引用匹配的子表达式，其中 *name* 是捕获子表达式的名称。
- 通过使用正则表达式中的反向引用构造。使用语法 `\`*数字* 在同一正则表达式中引用匹配的子表达式，其中 *数字* 是捕获的表达式的初始数字。已命名的匹配子表达式在匹配子表达式后从左到右连续编号。
- 通过使用 `${`*name*`}` $ Regex.Replace number Match.Result 替换序列，其中 *name* 是捕获子表达式的名称。
- 通过在 Regex.Replace 或 Match.Result 方法调用中使用 `$` 数字 替换序列，其中"数字"为捕获的子表达式的序号。
- 以编程的方式，通过使用 GroupCollection 对象的方式，该对象由 Match.Groups 属性返回。集合中位置零上的成员表示正则表达式匹配。每个后续成员表示匹配的子表达式。已命名的捕获组在集合中存储在已编号的捕获组后面。
- 以编程方式，通过将子表达式名称提供至 GroupCollection 对象的索引器（在 C# 中），或者提供至其 Item[] 属性（在 Visual Basic 中）。

简单的正则表达式模式会阐释如何编号(未命名)，并且可以以编程方式或通过正则表达式语言语法引用已命名的组。正则表达式 `((?<One>abc)\d+)?(?<Two>xyz)(.*)` 按编号和名称产生下列捕获组。编号为 0 的第一个捕获组总是指整个模式。

“”	“”	“”
0	0(默认名称)	`((?<One>abc)\d+)?(?<Two>xyz)(.*)`
1	1(默认名称)	`((?<One>abc)\d+)`
2	2(默认名称)	`(.*)`
3	One	`(?<One>abc)`
4	Two	`(?<Two>xyz)`

下面的示例阐释了一个正则表达式，标识出重复的单词和紧随每个重复的单词的单词。正则表达式模式定义了两个命名的子表达式：`duplicateWord`，它表示重复的单词；和 `nextWord`，它表示后面跟随重复单词的单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)";
      string input = "He said that that was the the correct answer.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine("A duplicate '{0}' at position {1} is followed by '{2}'.",
                           match.Groups["duplicateWord"].Value, match.Groups["duplicateWord"].Index,
                           match.Groups["nextWord"].Value);
   }
}
// The example displays the following output:
//       A duplicate 'that' at position 8 is followed by 'was'.
//       A duplicate 'the' at position 22 is followed by 'correct'.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)"
        Dim input As String = "He said that that was the the correct answer."
        Console.WriteLine(Regex.Matches(input, pattern, RegexOptions.IgnoreCase).Count)
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine("A duplicate '{0}' at position {1} is followed by '{2}'.", _
                              match.Groups("duplicateWord").Value, match.Groups("duplicateWord").Index, _
                              match.Groups("nextWord").Value)
        Next
    End Sub
End Module
' The example displays the following output:
'    A duplicate 'that' at position 8 is followed by 'was'.
'    A duplicate 'the' at position 22 is followed by 'correct'.
```

正则表达式模式按如下方式定义：

```
(?<duplicateWord>\w+)\s\k<duplicateWord>\W(?<nextWord>\w+)
```

下表演示了正则表达式的含义。

"	"
`(?<duplicateWord>\w+)`	匹配一个或多个单词字符。命名此捕获组 `duplicateWord`。
`\s`	与空白字符匹配。
`\k<duplicateWord>`	匹配名为 `duplicateWord` 的捕获的组。
`\W`	匹配包括空格和标点符号的一个非单词字符。这样可以防止正则表达式模式匹配以第一个捕获组的单词开头的单词。
`(?<nextWord>\w+)`	匹配一个或多个单词字符。命名此捕获组 `nextWord`。

请注意可在正则表达式中重复组名。例如，可将多个组命名为 `digit`，如下面的示例所示。在名称重复的情况下，Group 对象的值由输入字符串中最后一个成功的捕获确定。此外，如果组名不重复，则使用有关每个捕获的信息填充 CaptureCollection 。

在下面的示例中，正则表达式 `\D+(?<digit>\d+)\D+(?<digit>\d+)?` 中两次出现了名为 `digit` 的组。第一个名为 `digit` 的组捕获一个或多个数字字符。第二个名为 `digit` 的组捕获一个或多个数字字符的零个或一个匹配项。如示例的输出所示，如果第二个捕获组成功匹配文本，则文本的值定义 Group 对象的值。如果第二个捕获组无法匹配输入字符串，则最后一个成功匹配的值定义 Group 对象的值。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      String pattern = @"\D+(?<digit>\d+)\D+(?<digit>\d+)?";
      String[] inputs = { "abc123def456", "abc123def" };
      foreach (var input in inputs) {
         Match m = Regex.Match(input, pattern);
         if (m.Success) {
            Console.WriteLine("Match: {0}", m.Value);
            for (int grpCtr = 1; grpCtr < m.Groups.Count; grpCtr++) {
               Group grp = m.Groups[grpCtr];
               Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value);
               for (int capCtr = 0; capCtr < grp.Captures.Count; capCtr++)
                  Console.WriteLine("   Capture {0}: {1}", capCtr,
                                    grp.Captures[capCtr].Value);
            }
         }
         else {
            Console.WriteLine("The match failed.");
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: abc123def456
//       Group 1: 456
//          Capture 0: 123
//          Capture 1: 456
//
//       Match: abc123def
//       Group 1: 123
//          Capture 0: 123
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\D+(?<digit>\d+)\D+(?<digit>\d+)?"
        Dim inputs() As String = {"abc123def456", "abc123def"}
        For Each input As String In inputs
            Dim m As Match = Regex.Match(input, pattern)
            If m.Success Then
                Console.WriteLine("Match: {0}", m.Value)
                For grpCtr As Integer = 1 to m.Groups.Count - 1
                    Dim grp As Group = m.Groups(grpCtr)
                    Console.WriteLine("Group {0}: {1}", grpCtr, grp.Value)
                    For capCtr As Integer = 0 To grp.Captures.Count - 1
                        Console.WriteLine("   Capture {0}: {1}", capCtr,
                                          grp.Captures(capCtr).Value)
                    Next
                Next
            Else
                Console.WriteLine("The match failed.")
            End If
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       Match: abc123def456
'       Group 1: 456
'          Capture 0: 123
'          Capture 1: 456
'
'       Match: abc123def
'       Group 1: 123
'          Capture 0: 123
```

下表演示了正则表达式的含义。

“	“
`\D+`	匹配一个或多个非十进制数字字符。
`(?<digit>\d+)`	匹配一个或多个十进制数字字符。将匹配分配到 `digit` 命名组。
`\D+`	匹配一个或多个非十进制数字字符。
`(?<digit>\d+)?`	匹配一个或多个十进制数字字符的零个或一个匹配项。将匹配分配到 `digit` 命名组。

平衡组定义

平衡组定义将删除以前定义的组和存储的定义，并在当前组中存储以前定义的组和当前组之间的间隔。此分组构造具有以下格式：

`(?<name1-name2>subexpression)`

或：

`(?'name1-name2' subexpression)`

name1 位置是当前的组(可选)， *name2* 是一个以前定义的组，而 *子表达式* 是任何有效的正则表达式模式。平衡组定义删除 *name2* 的定义并在 *name1* 中保存 *name2* 和 *name1* 之间的间隔。如果未定义 *name2* 组，则匹配将回溯。由于删除 *name2* 的最后一个定义会显示 *name2* 以前的定义，因此该构造允许将 *name2* 组的捕获堆栈用作计数器，用于跟踪嵌套构造(如括号或者左括号和右括号)。

平衡组定义将 *name2* 作为堆栈使用。将每个嵌套构造的开头字符放在组中，并放在其 Group.Captures 集合中。当匹配结束字符时，从组中删除其相应的开始字符，并且 Captures 集合减少 1。所有嵌套构造的开始和结束字符匹配完后，name2 为空。

> **NOTE**
>
> 通过修改下面示例中的正则表达式来使用合适的嵌套构造的开始和结束字符后，你可以用它来处理多数嵌套构造，如数学表达式或包括多个嵌套方法调用的程序代码行。

下面的示例使用平衡组定义匹配输入字符串中的左右尖括号 (<>)。该示例定义两个已命名的组， `Open` 和 `Close`，用作堆栈来跟踪配对的尖括号。将每个已捕获的左尖括号推入到 `Open` 组的捕获集合，而将每个已捕获的右尖括号推入到 `Close` 组的捕获集合。平衡组定义确保每个左尖括号都有一个匹配的右尖角括号。如果没有，则仅会在 `(?(Open)(?!))` 组不为空的情况下计算最终子模式 `Open` 的值(因此，如果所有嵌套构造尚未关闭)。如果计算了最终子模式的值，则匹配将失败，因为 `(?!)` 子模式是始终失败的零宽度负预测先行断言。

```
using System;
using System.Text.RegularExpressions;

class Example
{
   public static void Main()
   {
      string pattern = "^[^<>]*" +
                       "(" +
                       "((?'Open'<)[^<>]*)+" +
                       "((?'Close-Open'>)[^<>]*)+" +
                       ")*" +
                       "(?(Open)(?!))$";
      string input = "<abc><mno<xyz>>";

      Match m = Regex.Match(input, pattern);
      if (m.Success == true)
      {
         Console.WriteLine("Input: \"{0}\" \nMatch: \"{1}\"", input, m);
         int grpCtr = 0;
         foreach (Group grp in m.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", grpCtr, grp.Value);
            grpCtr++;
            int capCtr = 0;
            foreach (Capture cap in grp.Captures)
            {
                Console.WriteLine("      Capture {0}: {1}", capCtr, cap.Value);
                capCtr++;
            }
          }
      }
      else
      {
         Console.WriteLine("Match failed.");
      }
    }
}
// The example displays the following output:
//    Input: "<abc><mno<xyz>>"
//    Match: "<abc><mno<xyz>>"
//       Group 0: <abc><mno<xyz>>
//          Capture 0: <abc><mno<xyz>>
//       Group 1: <mno<xyz>>
//          Capture 0: <abc>
//          Capture 1: <mno<xyz>>
//       Group 2: <xyz
//          Capture 0: <abc
//          Capture 1: <mno
//          Capture 2: <xyz
//       Group 3: >
//          Capture 0: >
//          Capture 1: >
//          Capture 2: >
//       Group 4:
//       Group 5: mno<xyz>
//          Capture 0: abc
//          Capture 1: xyz
//          Capture 2: mno<xyz>
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^[^<>]*" & _
                                "(" + "((?'Open'<)[^<>]*)+" & _
                                "((?'Close-Open'>)[^<>]*)+" + ")*" & _
                                "(?(Open)(?!))$"
        Dim input As String = "<abc><mno<xyz>>"
        Dim rgx AS New Regex(pattern) '
        Dim m As Match = Regex.Match(input, pattern)
        If m.Success Then
            Console.WriteLine("Input: ""{0}"" " & vbCrLf & "Match: ""{1}""", _
                              input, m)
            Dim grpCtr As Integer = 0
            For Each grp As Group In m.Groups
                Console.WriteLine("   Group {0}: {1}", grpCtr, grp.Value)
                grpCtr += 1
                Dim capCtr As Integer = 0
                For Each cap As Capture In grp.Captures
                    Console.WriteLine("      Capture {0}: {1}", capCtr, cap.Value)
                    capCtr += 1
                Next
            Next
        Else
            Console.WriteLine("Match failed.")
        End If
    End Sub
End Module
' The example displays the following output:
'       Input: "<abc><mno<xyz>>"
'       Match: "<abc><mno<xyz>>"
'          Group 0: <abc><mno<xyz>>
'             Capture 0: <abc><mno<xyz>>
'          Group 1: <mno<xyz>>
'             Capture 0: <abc>
'             Capture 1: <mno<xyz>>
'          Group 2: <xyz
'             Capture 0: <abc
'             Capture 1: <mno
'             Capture 2: <xyz
'          Group 3: >
'             Capture 0: >
'             Capture 1: >
'             Capture 2: >
'          Group 4:
'          Group 5: mno<xyz>
'             Capture 0: abc
'             Capture 1: xyz
'             Capture 2: mno<xyz>
```

正则表达式模式为：

```
^[^<>]*(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*(?(Open)(?!))$
```

正则表达式按如下方式解释：

|
 |
 |
| --- | --- |
| `^` | 从字符串的开头部分开始。 |
| `[^<>]*` | 匹配零个或多个不是左侧或右侧角度方括号的字符。 |

`(?'Open'<)`	匹配左尖括号并分配给名为 `Open` 的组。
`[^<>]*`	匹配零个或多个不是左侧或右侧角度方括号的字符。
`((?'Open'<)[^<>]*)+`	匹配跟在非左尖括号或非右尖括号的零个或多个字符后面的一个或多个左尖括号匹配项。这是第二个捕获组。
`(?'Close-Open'>)`	匹配右尖括号，将 `Open` 组和当前组分配给 `Close` 组并删除 `Open` 组的定义。
`[^<>]*`	匹配非左尖括号或非右尖括号的任何字符的零个或多个匹配项。
`((?'Close-Open'>)[^<>]*)+`	匹配跟在零后面或跟在非左尖括号或非右尖括号的多个字符后面的一个或多个右尖括号匹配项。在匹配右尖括号时，将 `Open` 组和当前组分配给 `Close` 组并删除 `Open` 组的定义。这是第三个捕获组。
`(((?'Open'<)[^<>]*)+((?'Close-Open'>)[^<>]*)+)*`	匹配零个或多个下列模式的匹配项：一个或多个左尖括号匹配项，后跟零个或多个非尖括号字符，后跟一个或多个右尖括号的匹配项，后跟零个或多个非尖括号的匹配项。在匹配右尖括号时，删除 `Open` 组的定义，并将 `Open` 组和当前组之间的子字符串分配给 `Close` 组。这是第一个捕获组。
`(?(Open)(?!))`	如果 `Open` 组存在，并可以匹配空字符串，则放弃匹配，但不前移字符串中的正则表达式引擎的位置。这是零宽度负预测先行断言。因为空字符串总是隐式地存在于输入字符串中，所以此匹配始终失败。此匹配的失败表示尖括号不平衡。
`$`	匹配输入字符串的末尾部分。

最终子表达式 `(?(Open)(?!))`，指示是否正确平衡输入字符串中的嵌套构造（例如，是否每个左尖括号由右键括号匹配）。它使用基于有效的捕获组的条件匹配，有关详细信息请参阅 替换构造。如果定义了 `Open` 组，则正则表达式引擎会尝试匹配输入字符串中的子表达式 `(?!)` 。仅当嵌套构造不均衡时，才应该定义 `Open` 组。因此，要在输入字符串中匹配的模式应该是一个始终导致匹配失败的模式。在此情况下，`(?!)` 是始终失败的零宽度负预测先行断言，因为空字符串总是隐式地存在于输入字符串中的下一个位置。

在此示例中，正则表达式引擎评估输入字符串“<abc><mno<xyz>>”，如下表所示。

1	`^`	从输入字符串的开头部分开始匹配。
2	`[^<>]*`	查找左尖括号之前的非尖括号字符；未找到匹配项。
3	`(((?'Open'<)`	匹配“<abc>”中的左尖括号并将它分配给 `Open` 组。
4	`[^<>]*`	与“abc”匹配。

5	`)+`	“<abc”是第二个捕获组的值。 输入字符串中的下一个字符不是左尖括号，因此正则表达式引擎不会循环回到 `(?'Open'<)[^<>]*)` 子模式。
6	`((?'Close-Open'>)`	匹配“<abc>”中的右尖括号，将“abc”（ `Open` 组和右尖括号之间的子字符串）分配给 `Close` 组并删除 `Open` 组的当前值（“<”）。
7	`[^<>]*`	查找右尖括号之后的非尖括号字符；未找到匹配项。
8	`)+`	第三个捕获组的值是“>”。 输入字符串中的下一个字符不是右尖括号，因此正则表达式引擎不会循环回到 `((?'Close-Open'>)[^<>]*)` 子模式。
9	`)*`	第一个捕获组的值是“<abc>”。 输入字符串中的下一个字符是左尖括号，因此正则表达式引擎会循环回到 `(((?'Open'<)` 子模式。
10	`(((?'Open'<)`	匹配 <mno" and assigns it to the `Open` group. Its Group.Captures 集合中的左尖括号现在具有单个值“<”。
11	`[^<>]*`	与“mno”匹配。
12	`)+`	“<mno”是第二个捕获组的值。 输入字符串中的下一个字符是左尖括号，因此正则表达式引擎会循环回到 `(?'Open'<)[^<>]*)` 子模式。
13	`(((?'Open'<)`	匹配“<xyz>”中的左尖括号并将它分配给 `Open` 组。 `Open` 组的 Group.Captures 集合现在包括两个捕获，即“<mno", and the left angle bracket from "<xyz>”中的左尖括号。
14	`[^<>]*`	与“xyz”匹配。
15	`)+`	“<xyz”是第二个捕获组的值。 输入字符串中的下一个字符不是左尖括号，因此正则表达式引擎不会循环回到 `(?'Open'<)[^<>]*)` 子模式。

16	`((?'Close-Open'>)`	匹配“<xyz>”中的右尖括号。“xyz”将 `Open` 组合右尖括号之间的子字符串分配给 `Close` 组，并删除 `Open` 组的当前值。`Open` 组的 <mno") becomes the current value of the `Open` group. The Captures 集合中上一个捕获（左尖括号）的值现在包括一个捕获，即“<xyz>”中的左尖括号。
17	`[^<>]*`	查找非尖括号字符；未找到匹配项。
18	`)+`	第三个捕获组的值是“>”。 输入字符串中的下一个字符是右尖括号，因此正则表达式引擎会循环回到 `((?'Close-Open'>)[^<>]*)` 子模式。
19	`((?'Close-Open'>)`	匹配“xyz>>”中的最后右尖括号，将“mno<xyz>”（`Open` 组和右尖括号之间的子字符串）分配给 `Close` 组并删除 `Open` 组的当前值。`Open` 组现在为空。
20	`[^<>]*`	查找非尖括号字符；未找到匹配项。
21	`)+`	第三个捕获组的值是“>”。 输入字符串中的下一个字符不是右尖括号，因此正则表达式引擎不会循环回到 `((?'Close-Open'>)[^<>]*)` 子模式。
22	`)*`	第一个捕获组的值是“<mno<xyz>>”。 输入字符串中的下一个字符不是左尖括号，因此正则表达式引擎不会循环回到 `(((?'Open'<)` 子模式。
23	`(?(Open)(?!))`	`Open` 组是未定义的，因此没有尝试匹配。
24	`$`	匹配输入字符串的末尾部分。

非捕获组

以下分组构造不会捕获由子表达式匹配的子字符串：

`(?:subexpression)`

其中 *子表达式* 为任何有效正则表达式模式。当一个限定符应用到一个组，但组捕获的子字符串并非所需时，通常会使用非捕获组构造。

> **NOTE**
> 如果正则表达式包含嵌套的分组构造，则外部非捕获组构造不适用于内部嵌套组构造。

下面的示例阐释包括非捕获组的正则表达式。请注意输出不包含任何已捕获的组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?:\b(?:\w+)\W*)+\.";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: {0}", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
         Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
   }
}
// The example displays the following output:
//       Match: This is a short sentence.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?:\b(?:\w+)\W*)+\."
        Dim input As String = "This is a short sentence."
        Dim match As Match = Regex.Match(input, pattern)
        Console.WriteLine("Match: {0}", match.Value)
        For ctr As Integer = 1 To match.Groups.Count - 1
            Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       Match: This is a short sentence.
```

正则表达式 `(?:\b(?:\w+)\W*)+\.` 匹配由句号终止的语句。因为正则表达式重点介绍句子，而不是个别单词，所以分组构造以独占方式用作限定符。正则表达式模式可以解释为下表中所示内容。

"	"
`\b`	在单词边界处开始匹配。
`(?:\w+)`	匹配一个或多个单词字符。不将匹配的文本分配给捕获的组。
`\W*`	匹配零个或多个非单词字符。
`(?:\b(?:\w+)\W*)+`	一次或多次匹配跟在零个或多个非单词字符后面以单词边界开头的一个或多个单词字符的模式。不将匹配的文本分配给捕获的组。
`\.`	匹配句点。

组选项

以下分组构造应用或禁用子表达式中指定的选项：

`(?imnsx-imnsx:` subexpression `)`

其中 *子表达式* 为任何有效正则表达式模式。例如，`(?i-s:)` 将打开不区分大小写并禁用单行模式。有关可以指定的内联选项的更多信息，请参见 正则表达式选项。

> **NOTE**
> 可以指定将应用于整个正则表达式，而不是子表达式的选项，方法是使用 System.Text.RegularExpressions.Regex 类构造函数或静态方法。也可指定在正则表达式特定点后使用的内联选项，方法是使用 `(?imnsx-imnsx)` 语言构造。

组的选项构造并非捕获组。即尽管 *子表达式* 捕获的字符串的任意部分包含在匹配中，但不会包含在捕获的组中也不会用于填充 GroupCollection 对象。

例如，以下示例中的正则表达式 `\b(?ix: d \w+)\s` 使用分组构造中的内联选项，以启用不区分大小写的匹配和在识别所有以字母“d”开头的单词时忽略模式空白。该正则表达式的定义如下表所示。

“	“
`\b`	在单词边界处开始匹配。
`(?ix: d \w+)`	使用不区分大小写的匹配并忽略此模式中的空白，匹配后跟一个或多个单词字符的“d”。
`\s`	与空白字符匹配。

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix: d \w+)\s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

零宽度正预测先行断言

以下分组构造定义零宽度正预测先行断言：

`(?=` subexpression `)`

其中 *子表达式* 为任何正则表达式模式。若要成功匹配，则输入字符串必须匹配 *子表达式* 中的正则表达式模式，

尽管匹配的子字符串未包含在匹配结果中。零宽度正预测先行断言不会回溯。

通常，零宽度正预测先行断言是在正则表达式模式的末尾找到的。它定义了一个子字符串，该子字符串必须出现在匹配字符串的末尾但又不能包含在匹配结果中。还有助于防止过度回溯。可使用零宽度正预测先行断言来确保特定捕获组以与专为该捕获组定义的模式的子集相匹配的文本开始。例如，如果捕获组与连续单词字符相匹配，可以使用零宽度正预测先行断言要求第一个字符是按字母顺序排列的大写字符。

下面的示例使用零宽度正预测先行断言，以匹配输入字符串中谓词“is”前的单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+(?=\sis\b)";
      string[] inputs = { "The dog is a Malamute.",
                          "The island has beautiful birds.",
                          "The pitch missed home plate.",
                          "Sunday is a weekend day." };

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine("'{0}' precedes 'is'.", match.Value);
         else
            Console.WriteLine("'{0}' does not match the pattern.", input);
      }
   }
}
// The example displays the following output:
//    'dog' precedes 'is'.
//    'The island has beautiful birds.' does not match the pattern.
//    'The pitch missed home plate.' does not match the pattern.
//    'Sunday' precedes 'is'.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b\w+(?=\sis\b)"
        Dim inputs() As String = {"The dog is a Malamute.", _
                                  "The island has beautiful birds.", _
                                  "The pitch missed home plate.", _
                                  "Sunday is a weekend day."}

        For Each input As String In inputs
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then
                Console.WriteLine("'{0}' precedes 'is'.", match.Value)
            Else
                Console.WriteLine("'{0}' does not match the pattern.", input)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       'dog' precedes 'is'.
'       'The island has beautiful birds.' does not match the pattern.
'       'The pitch missed home plate.' does not match the pattern.
'       'Sunday' precedes 'is'.
```

正则表达式 `\b\w+(?=\sis\b)` 可以解释为下表中所示内容。

“	“
`\b`	在单词边界处开始匹配。
`\w+`	匹配一个或多个单词字符。
`(?=\sis\b)`	确定单词字符是否后接空白字符和字符串“is”，其在单词边界处结束。如果如此，则匹配成功。

零宽度负预测先行断言

以下分组构造定义零宽度负预测先行断言：

`(?!` subexpression `)`

其中 *子表达式* 为任何正则表达式模式。若要成功匹配，则输入字符串不得匹配 *子表达式* 中的正则表达式模式，尽管匹配的子字符串未包含在匹配结果中。

零宽度负预测先行断言通常用在正则表达式的开头或结尾。正则表达式的开头可以定义当其定义了要被匹配的相似但更常规的模式时，不应被匹配的特定模式。在这种情况下，它通常用于限制回溯。正则表达式的末尾可以定义不能出现在匹配项末尾处的子表达式。

下面的示例定义了正则表达式匹配，其在正则表达式的开头使用零宽度预测先行断言，以匹配未以“un”开头的单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?!un)\w+\b";
      string input = "unite one unethical ethics use untie ultimate";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       one
//       ethics
//       use
//       ultimate
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(?!un)\w+\b"
        Dim input As String = "unite one unethical ethics use untie ultimate"
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       one
'       ethics
'       use
'       ultimate
```

正则表达式 `\b(?!un)\w+\b` 可以解释为下表中所示内容。

“	“
`\b`	在单词边界处开始匹配。
`(?!un)`	确定接下来的两个的字符是否为“un”。如果没有，则可能匹配。
`\w+`	匹配一个或多个单词字符。
`\b`	在单词边界处结束匹配。

下面的示例定义了正则表达式匹配，其在正则表达式的末尾使用零宽度预测先行断言，以匹配未以标点字符结束的单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\w+\b(?!\p{P})";
      string input = "Disconnected, disjointed thoughts in a sentence fragment.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       disjointed
//       thoughts
//       in
//       a
//       sentence
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b\w+\b(?!\p{P})"
        Dim input As String = "Disconnected, disjointed thoughts in a sentence fragment."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       disjointed
'       thoughts
'       in
'       a
'       sentence
```

正则表达式 `\b\w+\b(?!\p{P})` 可以解释为下表中所示内容。

"	"
`\b`	在单词边界处开始匹配。
`\w+`	匹配一个或多个单词字符。
`\b`	在单词边界处结束匹配。
`\p{P})`	如果下个字符不是一个标点符号(如句点或逗号)，则匹配成功。

零宽度正回顾后发断言

以下分组构造定义零宽度正回顾后发断言：

`(?<=` subexpression `)`

其中 *子表达式* 为任何正则表达式模式。若要成功匹配，则 *子表达式* 必须在输入字符串当前位置左侧出现，尽管 `subexpression` 未包含在匹配结果中。零宽度正回顾后发断言不会回溯。

零宽度正预测后发断言通常在正则表达式的开头使用。它们定义的模式是一个匹配的前提条件，但它不是匹配结果的一部分。

例如，下面的示例匹配二十一世纪年份的最后两个数字(也就是说，数字"20"要在匹配的字符串之前)。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "2010 1999 1861 2140 2009";
      string pattern = @"(?<=\b20)\d{2}\b";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       10
//       09
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "2010 1999 1861 2140 2009"
        Dim pattern As String = "(?<=\b20)\d{2}\b"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       10
'       09
```

正则表达式模式 `(?<=\b20)\d{2}\b` 的含义如下表所示。

“	“
`\d{2}`	匹配两个十进制数字。
`(?<=\b20)`	如果两个十进制数字的字边界以小数位数“20”开头，则继续匹配。
`\b`	在单词边界处结束匹配。

零宽度正回顾后发断言还用于在捕获组中的最后一个或多个字符不得为与该捕获组的正则表达式模式相匹配的字符的子集时限制回溯。例如，如果组捕获所有的连续单词字符，可以使用零宽度正回顾后发断言要求最后一个字符时按字母顺序的。

零宽度负回顾后发断言

以下组构造定义零宽度负回顾后发断言：

`(?<!` subexpression `)`

其中 *子表达式* 为任何正则表达式模式。若要成功匹配，则 *子表达式* 不得在输入字符串当前位置的左侧出现。但是，任何不匹配 `subexpression` 的子字符串不包含在匹配结果中。

零宽度负回顾后发断言通常在正则表达式的开头使用。它们定义的模式预先排除在后面的字符串中的匹配项。

它们还用于在捕获组中的最后一个或多个字符不得为与该捕获组的正则表达式模式相匹配的其中一个或多个字符时限制回溯。例如，如果如果组捕获了所有的连续单词字符，可以使用零宽度正回顾后发断言要求最后一个字符不是下划线 (_)。

下面的示例匹配除周末之外的一周的任何一天（也就是星期六和星期日都没有）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] dates = { "Monday February 1, 2010",
                         "Wednesday February 3, 2010",
                         "Saturday February 6, 2010",
                         "Sunday February 7, 2010",
                         "Monday, February 8, 2010" };
      string pattern = @"(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b";

      foreach (string dateValue in dates)
      {
         Match match = Regex.Match(dateValue, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
      }
   }
}
// The example displays the following output:
//       February 1, 2010
//       February 3, 2010
//       February 8, 2010
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim dates() As String = {"Monday February 1, 2010", _
                                 "Wednesday February 3, 2010", _
                                 "Saturday February 6, 2010", _
                                 "Sunday February 7, 2010", _
                                 "Monday, February 8, 2010"}
        Dim pattern As String = "(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b"

        For Each dateValue As String In dates
            Dim match As Match = Regex.Match(dateValue, pattern)
            If match.Success Then
                Console.WriteLine(match.Value)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       February 1, 2010
'       February 3, 2010
'       February 8, 2010
```

正则表达式模式 `(?<!(Saturday|Sunday) )\b\w+ \d{1,2}, \d{4}\b` 的含义如下表所示。

`\b`	在单词边界处开始匹配。

“	“
`\w+`	匹配一个或多个后跟空白字符的单词字符。
`\d{1,2},`	匹配空白字符和逗号后面的一个或两个十进制数字。
`\d{4}\b`	匹配四个十进制数字并在单词边界处结束匹配。
`(?<!(Saturday|Sunday) )`	如果匹配以字符串“星期六”或者“星期日”开头，后跟一个空格，则匹配成功。

原子组

以下分组构造表示原子组（在其他一些正则表达式引擎中称为非回溯子表达式、原子子表达式或一次性子表达式）：

`(?>` subexpression `)`

其中 *子表达式* 为任何正则表达式模式。

通常，如果正则表达式包含一个可选或可替代匹配模式并且备选不成功的话，正则表达式引擎可以在多个方向上分支以将输入的字符串与某种模式进行匹配。如果未找到使用第一个分支的匹配项，则正则表达式引擎可以备份或回溯到使用第一个匹配项的点并尝试使用第二个分支的匹配项。此过程可继续进行，直到尝试所有分支。

仅当嵌套构造不均衡时，才应该定义 `(?>` *子表达式* `)` 语言构造禁用回溯。正则表达式引擎将在输入字符串中匹配尽可能多的字符。在没有任何进一步匹配可用时，它将不回溯以尝试备用模式匹配。（也就是说，该子表达式仅与可由该子表达式单独匹配的字符串匹配；子表达式不会尝试与基于该子表达式的字符串和任何该子表达式之后的子表达式匹配。）

如果你知道回溯不会成功，则建议使用此选项。防止正则表达式引擎执行不需要的搜索可以提高性能。

以下示例展示了原子组如何修改模式匹配的结果。回溯正则表达式成功匹配一系列重复字符，在字边界上其后为相同字符，但非回溯正则表达式不会匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "cccd.", "aaad", "aaaa" };
      string back = @"(\w)\1+.\b";
      string noback = @"(?>(\w)\1+).\b";

      foreach (string input in inputs)
      {
         Match match1 = Regex.Match(input, back);
         Match match2 = Regex.Match(input, noback);
         Console.WriteLine("{0}: ", input);

         Console.Write("   Backtracking : ");
         if (match1.Success)
            Console.WriteLine(match1.Value);
         else
            Console.WriteLine("No match");

         Console.Write("   Nonbacktracking: ");
         if (match2.Success)
            Console.WriteLine(match2.Value);
         else
            Console.WriteLine("No match");
      }
   }
}
// The example displays the following output:
//    cccd.:
//       Backtracking : cccd
//       Nonbacktracking: cccd
//    aaad:
//       Backtracking : aaad
//       Nonbacktracking: aaad
//    aaaa:
//       Backtracking : aaaa
//       Nonbacktracking: No match
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"cccd.", "aaad", "aaaa"}
        Dim back As String = "(\w)\1+.\b"
        Dim noback As String = "(?>(\w)\1+).\b"

        For Each input As String In inputs
            Dim match1 As Match = Regex.Match(input, back)
            Dim match2 As Match = Regex.Match(input, noback)
            Console.WriteLine("{0}: ", input)

            Console.Write("   Backtracking : ")
            If match1.Success Then
                Console.WriteLine(match1.Value)
            Else
                Console.WriteLine("No match")
            End If

            Console.Write("   Nonbacktracking: ")
            If match2.Success Then
                Console.WriteLine(match2.Value)
            Else
                Console.WriteLine("No match")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'    cccd.:
'       Backtracking : cccd
'       Nonbacktracking: cccd
'    aaad:
'       Backtracking : aaad
'       Nonbacktracking: aaad
'    aaaa:
'       Backtracking : aaaa
'       Nonbacktracking: No match
```

非回溯正则表达式 `(?>(\w)\1+).\b` 的定义如下表所示。

“	“
`(\w)`	匹配单个单词字符，并将其分配给第一捕获组。
`\1+`	一次或多次匹配的第一个捕获子字符串的值。
`.`	匹配任意字符。
`\b`	在单词边界处结束匹配。
`(?>(\w)\1+)`	匹配一个重复的单词字符的一个或多个匹配项，但不执行回溯以匹配在单词边界上的最后一个字符。

分组构造和正则表达式对象

由正则表达式捕获组匹配的子字符串由 System.Text.RegularExpressions.Group 对象表示，其从 System.Text.RegularExpressions.GroupCollection 对象检索，其由 Match.Groups 属性返回。填充 GroupCollection 对象，如下所示：

- 集合中的第一个 Group 对象(位于索引零的对象)表示整个匹配。
- 下一组 Group 对象表示未命名(编号)的捕获组。它们以在正则表达式中定义的顺序出现，从左至右。这些组的索引值范围从 1 到集合中未命名捕获组的数目。(特定组索引等效于其带编号的反向引用。有关向后引用的更多信息，请参见 反向引用构造。)
- 最后的 Group 对象组表示命名的捕获组。它们以在正则表达式中定义的顺序出现，从左至右。第一个名为捕获组的索引值是一个大于最后一个未命名的捕获组的索引。如果正则表达式中没有未命名捕获组，则第一个命名的捕获组的索引值为 1。

如果将限定符应用于捕获组，则对应的 Group 对象的 Capture.Value、Capture.Index和 Capture.Length 属性反映捕获组捕获的最后一个子字符串。可以检索一整组子字符串，其是按组捕获的并具有来自 CaptureCollection 对象的限定符，其由 Group.Captures 属性返回。

下面的示例阐释 Group 和 Capture 对象之间的关系。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\b(\w+)\W+)+";
      string input = "This is a short sentence.";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}'", match.Value);
      for (int ctr = 1; ctr < match.Groups.Count; ctr++)
      {
         Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups[ctr].Value);
         int capCtr = 0;
         foreach (Capture capture in match.Groups[ctr].Captures)
         {
            Console.WriteLine("      Capture {0}: '{1}'", capCtr, capture.Value);
            capCtr++;
         }
      }
   }
}
// The example displays the following output:
//       Match: 'This is a short sentence.'
//          Group 1: 'sentence.'
//             Capture 0: 'This '
//             Capture 1: 'is '
//             Capture 2: 'a '
//             Capture 3: 'short '
//             Capture 4: 'sentence.'
//          Group 2: 'sentence'
//             Capture 0: 'This'
//             Capture 1: 'is'
//             Capture 2: 'a'
//             Capture 3: 'short'
//             Capture 4: 'sentence'
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(\b(\w+)\W+)+"
        Dim input As String = "This is a short sentence."
        Dim match As Match = Regex.Match(input, pattern)
        Console.WriteLine("Match: '{0}'", match.Value)
        For ctr As Integer = 1 To match.Groups.Count - 1
            Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups(ctr).Value)
            Dim capCtr As Integer = 0
            For Each capture As Capture In match.Groups(ctr).Captures
                Console.WriteLine("      Capture {0}: '{1}'", capCtr, capture.Value)
                capCtr += 1
            Next
        Next
    End Sub
End Module
' The example displays the following output:
'       Match: 'This is a short sentence.'
'          Group 1: 'sentence.'
'             Capture 0: 'This '
'             Capture 1: 'is '
'             Capture 2: 'a '
'             Capture 3: 'short '
'             Capture 4: 'sentence.'
'          Group 2: 'sentence'
'             Capture 0: 'This'
'             Capture 1: 'is'
'             Capture 2: 'a'
'             Capture 3: 'short'
'             Capture 4: 'sentence'
```

正则表达式模式 `(\b(\w+)\W+)+` 从字符串提取各个单词。其定义如下表所示。

“	“
`\b`	在单词边界处开始匹配。
`(\w+)`	匹配一个或多个单词字符。这些字符一起构成一个单词。这是第二个捕获组。
`\W+`	匹配一个或多个非单词字符。
`(\b(\w+)\W+)`	一次或多次匹配跟在一个或多个非单词字符后面的一个或多个单词字符的模式。这是第一个捕获组。

第二个捕获组匹配句子的每个单词。第一个捕获组匹配每个单词，连同标点符号和该单词后的空白区域。Group 对象的索引是 2，提供了有关由第二个捕获组匹配的文本的信息。可从 CaptureCollection 对象获取捕获组捕获的整组单词，该对象由 Group.Captures 属性返回。

请参阅

- 正则表达式语言 - 快速参考
- 回溯

正则表达式中的限定符

2021/11/16 •

限定符指定输入中必须存在字符、组或字符类的多少实例才能找到匹配项。下表列出了 .NET 支持的限定符。

[illegible]	[illegible]	[illegible]
`*`	`*?`	匹配零次或多次。
`+`	`+?`	匹配一次或多次。
`?`	`??`	匹配零次或一次。
`{n}`	`{n}?`	恰好匹配 n 次。
`{n,}`	`{n,}?`	至少匹配 n 次。
`{n,m}`	`{n,m}?`	匹配 n 到 m 次 。

数量 `n` 和 `m` 是整数常量。通常，限定符是贪婪的；它们使正则表达式引擎匹配尽可能多的特定模式实例。向限定符追加 `?` 字符可使它成为惰性的；会使正则表达式引擎匹配尽可能少的实例。有关贪婪与惰性限定符之间的差异的完整说明，请参见本主题后面的贪婪与惰性限定符部分。

> **IMPORTANT**
> 嵌套限定符（例如正则表达式模式 `(a*)*` 的行为）可以按输入字符串中的字符数的指数函数形式，来增加正则表达式引擎必须执行的比较次数。若要详细了解此行为及其解决方法，请参阅回溯。

正则表达式限定符

以下部分列出了 .NET 正则表达式支持的限定符。

> **NOTE**
> 如果在正则表达式模式中遇到 *、+、?、{ 和 } 字符，正则表达式引擎会将它们解释为量符或量符构造的一部分，除非它们包含在字符类中。若要在字符类外部将这些字符解释文本字符，必须通过在它们前面加反斜杠来对它们进行转义。例如，正则表达式模式中的字符串 `\*` 会被解释为文本星号（"*"）字符。

匹配零次或多次：*

`*` 限定符与前面的元素匹配零次或多次。它相当于 `{0,}` 量符。`*` 是贪婪量符，相当的惰性量符是 `*?`。

下面的示例说明此正则表达式。在输入字符串中的九个数字组中，五个与模式匹配，四个（`95`、`929`、`9219` 和 `9919`）不匹配。

```
string pattern = @"\b91*9*\b";
string input = "99 95 919 929 9119 9219 999 9919 91119";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//       '99' found at position 0.
//       '919' found at position 6.
//       '9119' found at position 14.
//       '999' found at position 24.
//       '91119' found at position 33.
```

```
Dim pattern As String = "\b91*9*\b"
Dim input As String = "99 95 919 929 9119 9219 999 9919 91119"
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '99' found at position 0.
'       '919' found at position 6.
'       '9119' found at position 14.
'       '999' found at position 24.
'       '91119' found at position 33.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`91*`	匹配后跟零个或多个“1”字符的“9”。
`9*`	匹配零个或多个“9”字符。
`\b`	在字边界结束。

匹配一次或多次：+

`+` 量符匹配上一元素一次或多次。它相当于 `{1,}`。`+` 是贪婪量符，相当的惰性量符是 `+?`。

例如，正则表达式 `\ban+\w*?\b` 会尝试匹配以后跟字母 `n` 的一个或多个实例的字母 `a` 开头的完整单词。下面的示例说明此正则表达式。正则表达式会匹配单词 `an`、`annual`、`announcement` 和 `antique`，并且正确地无法匹配 `autumn` 和 `all`。

```
string pattern = @"\ban+\w*?\b";

string input = "Autumn is a great time for an annual announcement to all antique collectors.";
foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//       'an' found at position 27.
//       'annual' found at position 30.
//       'announcement' found at position 37.
//       'antique' found at position 57.
```

```
Dim pattern As String = "\ban+\w*?\b"

Dim input As String = "Autumn is a great time for an annual announcement to all antique collectors."
For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'an' found at position 27.
'       'annual' found at position 30.
'       'announcement' found at position 37.
'       'antique' found at position 57.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`an+`	匹配后跟一个或多个“n”字符的“a”。
`\w*?`	匹配单词字符零次或多次，但次数尽可能少。
`\b`	在字边界结束。

匹配零次或一次：?

`?` 量符匹配上一元素零次或一次。它相当于 `{0,1}`。`?` 是贪婪量符，相当的惰性量符是 `??`。

例如，正则表达式 `\ban?\b` 会尝试匹配以后跟字母 `n` 的零个或一个实例的字母 `a` 开头的完整单词。换句话说，它会尝试匹配单词 `a` 和 `an`。下面的示例说明此正则表达式。

```
string pattern = @"\ban?\b";
string input = "An amiable animal with a large snount and an animated nose.";
foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

// The example displays the following output:
//        'An' found at position 0.
//        'a' found at position 23.
//        'an' found at position 42.
```

```
Dim pattern As String = "\ban?\b"
Dim input As String = "An amiable animal with a large snount and an animated nose."
For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'An' found at position 0.
'       'a' found at position 23.
'       'an' found at position 42.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。

“	“
`an?`	匹配后跟零个或一个“n”字符的“a”。
`\b`	在字边界结束。

恰好匹配 n 次：{n}

`{n}` 限定符与前面的元素恰好匹配 n 次，其中 n 是任何整数 。`{n}` 是贪婪限定符，其惰性等效项是 `{n}?` 。

例如，正则表达式 `\b\d+\,\d{3}\b` 会尝试匹配依次后跟一个或多个十进制数字、三个十进制数字、一个单词边界的单词边界。下面的示例说明此正则表达式。

```
string pattern = @"\b\d+\,\d{3}\b";
string input = "Sales totaled 103,524 million in January, " +
                      "106,971 million in February, but only " +
                      "943 million in March.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '103,524' found at position 14.
//        '106,971' found at position 45.
```

```
Dim pattern As String = "\b\d+\,\d{3}\b"
Dim input As String = "Sales totaled 103,524 million in January, " + _
                      "106,971 million in February, but only " + _
                      "943 million in March."
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '103,524' found at position 14.
'       '106,971' found at position 45.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`\d+`	匹配一个或多个十进制数字。
`\,`	匹配逗号字符。
`\d{3}`	匹配三个十进制数字。
`\b`	在字边界结束。

至少匹配 n 次：{n,}

`{n,}` 限定符与前面的元素至少匹配 n 次，其中 n 是任何整数 。`{n,}` 是贪婪限定符，其惰性等效项是 `{n,}?` 。

例如，正则表达式 `\b\d{2,}\b\D+` 会尝试匹配依次后跟至少两个数字、一个单词边界和一个非数字字符的单词边界。下面的示例说明此正则表达式。正则表达式无法匹配短语 `"7 days"`，因为它只包含一个十进制数字，但可

以成功匹配短语 `"10 weeks and 300 years"` 。

```
string pattern = @"\b\d{2,}\b\D+";
string input = "7 days, 10 weeks, 300 years";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '10 weeks, ' found at position 8.
//        '300 years' found at position 18.
```

```
Dim pattern As String = "\b\d{2,}\b\D+"
Dim input As String = "7 days, 10 weeks, 300 years"
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '10 weeks, ' found at position 8.
'       '300 years' found at position 18.
```

正则表达式模式的定义如下表所示。

"	"
`\b`	在单词边界处开始。
`\d{2,}`	匹配至少两个十进制数字。
`\b`	与字边界匹配。
`\D+`	匹配至少一个非十进制数字。

匹配 n 到 m 次：{n,m}

`{` n `,` m `}` 限定符与前面的元素至少匹配 n 次，但不超过 m 次，其中 n 和 m 是整数 。`{` n `,` m `}` 是贪婪限定符，相当的惰性限定符是 `{` n `,` m `}?` 。

在下面的示例中，正则表达式 `(00\s){2,4}` 尝试与后跟一个空格的两个零数字匹配两到四次。请注意，输入字符串的最后一部分包含此模式五次，而不是最大值四次。但是，只有此子字符串的初始部分（到空格和第五对零）与正则表达式模式匹配。

```
string pattern = @"(00\s){2,4}";
string input = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        '00 00 ' found at position 8.
//        '00 00 00 ' found at position 23.
//        '00 00 00 00 ' found at position 35.
```

```
Dim pattern As String = "(00\s){2,4}"
Dim input As String = "0x00 FF 00 00 18 17 FF 00 00 00 21 00 00 00 00 00"
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       '00 00 ' found at position 8.
'       '00 00 00 ' found at position 23.
'       '00 00 00 00 ' found at position 35.
```

匹配零次或多次(惰性匹配):*?

`*?` 量符匹配上一元素零次或多次，但次数尽可能少。它是贪婪量符 `*` 对应的惰性量符。

在下面的示例中，正则表达式 `\b\w*?oo\w*?\b` 匹配包含字符串 `oo` 的所有单词。

```
 string pattern = @"\b\w*?oo\w*?\b";
 string input = "woof root root rob oof woo woe";
 foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

 //  The example displays the following output:
//        'woof' found at position 0.
//        'root' found at position 5.
//        'root' found at position 10.
//        'oof' found at position 19.
//        'woo' found at position 23.
```

```
Dim pattern As String = "\b\w*?oo\w*?\b"
Dim input As String = "woof root root rob oof woo woe"
For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'woof' found at position 0.
'       'root' found at position 5.
'       'root' found at position 10.
'       'oof' found at position 19.
'       'woo' found at position 23.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`\w*?`	匹配零个或多个单词字符，但字符要尽可能的少。
`oo`	匹配字符串“oo”。
`\w*?`	匹配零个或多个单词字符，但字符要尽可能的少。
`\b`	在单词边界处结束。

匹配一次或多次(惰性匹配):+?

`+?` 量符匹配上一元素一次或多次，但次数尽可能少。它是贪婪量符 `+` 对应的惰性量符。

例如，正则表达式 `\b\w+?\b` 匹配由单词边界分隔的一个或多个字符。下面的示例说明此正则表达式。

```
string pattern = @"\b\w+?\b";
string input = "Aa Bb Cc Dd Ee Ff";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'Aa' found at position 0.
//        'Bb' found at position 3.
//        'Cc' found at position 6.
//        'Dd' found at position 9.
//        'Ee' found at position 12.
//        'Ff' found at position 15.
```

```
Dim pattern As String = "\b\w+?\b"
Dim input As String = "Aa Bb Cc Dd Ee Ff"
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'Aa' found at position 0.
'       'Bb' found at position 3.
'       'Cc' found at position 6.
'       'Dd' found at position 9.
'       'Ee' found at position 12.
'       'Ff' found at position 15.
```

匹配零次或一次（惰性匹配）：??

`??` 量符匹配上一元素零次或一次，但次数尽可能少。它是贪婪量符 `?` 对应的惰性量符。

例如，正则表达式 `^\s*(System.)??Console.Write(Line)??\(??` 尝试匹配字符串“Console.Write”或“Console.WriteLine”。字符串还可以在“Console”前面包含“System.”，并且可以后跟左括号。字符串必须处于行的开头，不过前面可以是空格。下面的示例说明此正则表达式。

```
string pattern = @"^\s*(System.)??Console.Write(Line)??\(??";
string input = "System.Console.WriteLine(\"Hello!\")\n" +
                     "Console.Write(\"Hello!\")\n" +
                     "Console.WriteLine(\"Hello!\")\n" +
                     "Console.ReadLine()\n" +
                     "   Console.WriteLine";
foreach (Match match in Regex.Matches(input, pattern,
                                      RegexOptions.IgnorePatternWhitespace |
                                      RegexOptions.IgnoreCase |
                                      RegexOptions.Multiline))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'System.Console.Write' found at position 0.
//        'Console.Write' found at position 36.
//        'Console.Write' found at position 61.
//        '   Console.Write' found at position 110.
```

```
Dim pattern As String = "^\s*(System.)??Console.Write(Line)??\(??"
Dim input As String = "System.Console.WriteLine(""Hello!"")" + vbCrLf + _
                      "Console.Write(""Hello!"")" + vbCrLf + _
                      "Console.WriteLine(""Hello!"")" + vbCrLf + _
                      "Console.ReadLine()" + vbCrLf + _
                      "   Console.WriteLine"
For Each match As Match In Regex.Matches(input, pattern, _
                                          RegexOptions.IgnorePatternWhitespace Or RegexOptions.IgnoreCase Or
RegexOptions.MultiLine)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'System.Console.Write' found at position 0.
'       'Console.Write' found at position 36.
'       'Console.Write' found at position 61.
'       '   Console.Write' found at position 110.
```

正则表达式模式的定义如下表所示。

“	“
`^`	匹配输入流的开头。
`\s*`	匹配零个或多个空白字符。
`(System.)??`	匹配字符串“System.”的零个或一个匹配项。
`Console.Write`	匹配字符串“Console.Write”。
`(Line)??`	匹配字符串“Line”的零个或一个匹配项。
`\(??`	匹配左括号的零个或一个匹配项。

恰好匹配 n 次（惰性匹配）：{n}?

`{n}?` 限定符与前面的元素恰好匹配 `n` 次，其中 n 是任何整数 。它是贪婪限定符 `{n}` 的惰性对应项。

在下面的示例中，正则表达式 `\b(\w{3,}?\.){2}?\w{3,}?\b` 用于标识网站地址。请注意，它匹配“www.microsoft.com”和“msdn.microsoft.com”，但不匹配“mywebsite”或“mycompany.com”。

```
string pattern = @"\b(\w{3,}?\.){2}?\w{3,}?\b";
string input = "www.microsoft.com msdn.microsoft.com mywebsite mycompany.com";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'www.microsoft.com' found at position 0.
//        'msdn.microsoft.com' found at position 18.
```

```
Dim pattern As String = "\b(\w{3,}?\.){2}?\w{3,}?\b"
Dim input As String = "www.microsoft.com msdn.microsoft.com mywebsite mycompany.com"
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'www.microsoft.com' found at position 0.
'       'msdn.microsoft.com' found at position 18.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`(\w{3,}?\.)`	匹配至少 3 个后跟一个点或句点字符的单词字符，但字符数尽可能少。这是第一个捕获组。
`(\w{3,}?\.){2}?`	匹配第一个组中的模式两次，但次数尽可能少。
`\b`	在单词边界处结束匹配。

至少匹配 n 次（惰性匹配）：{n,}?

`{n,}?` 限定符与前面的元素至少匹配 `n` 次，其中 n 是任何整数，但次数尽可能少 。它是贪婪限定符 `{n,}` 的惰性对应项。

有关说明，请参阅上一部分中的 `{n}?` 限定符示例。该示例中的正则表达式使用 `{n,}` 限定符匹配包含后跟一个句点的至少三个字符的字符串。

匹配 n 到 m 次（惰性匹配）：{n,m}?

`{n,m}?` 限定符匹配上一元素 `n` 次到 `m` 次，其中 n 和 m 是整数，但次数尽可能少 。它是贪婪限定符 `{n,m}` 的惰性对应项 。

在下面的示例中，正则表达式 `\b[A-Z](\w*?\s*?){1,10}[.!?]` 匹配包含一到十个单词的句子。它可匹配输入字符串中的所有句子（除了包含 18 个单词的一个句子）。

```
string pattern = @"\b[A-Z](\w*?\s*?){1,10}[.!?]";
string input = "Hi. I am writing a short note. Its purpose is " +
                     "to test a regular expression that attempts to find " +
                     "sentences with ten or fewer words. Most sentences " +
                     "in this note are short.";
foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);

//  The example displays the following output:
//        'Hi.' found at position 0.
//        'I am writing a short note.' found at position 4.
//        'Most sentences in this note are short.' found at position 132.
```

```
Dim pattern As String = "\b[A-Z](\w*\s?){1,10}?[.!?]"
Dim input As String = "Hi. I am writing a short note. Its purpose is " + _
                     "to test a regular expression that attempts to find " + _
                     "sentences with ten or fewer words. Most sentences " + _
                     "in this note are short."
For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'       'Hi.' found at position 0.
'       'I am writing a short note.' found at position 4.
'       'Most sentences in this note are short.' found at position 132.
```

正则表达式模式的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`[A-Z]`	匹配从 A 到 Z 的大写字符。
`(\w*?\s*?)`	匹配零个或多个后跟一个或多个空白字符的单词字符，但次数应尽可能少。这是第一个捕获组。
`{1,10}`	与前面的模式匹配 1 到 10 次。
`[.!?]`	匹配标点字符“.”、“!”或“?”中的任何一种。

贪婪与惰性限定符

一些限定符具有两个版本：

- 贪婪版本。

  贪婪限定符尝试尽可能多地匹配元素。

- 非贪婪（或惰性）版本。

  非贪婪限定符尝试尽可能少地匹配元素。只需添加 `?`，即可将贪婪量符转换为惰性量符。

请考虑一个简单的正则表达式，它旨在从数字字符串（如信用卡号）中提取最后四位数。使用 `*` 贪婪量符的正则表达式版本是 `\b.*([0-9]{4})\b`。但是，如果字符串包含两个数字，则此正则表达式仅匹配第二个数字的最后四位数，如下面的示例所示。

```
string greedyPattern = @"\b.*([0-9]{4})\b";
string input1 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input1, greedyPattern))
   Console.WriteLine("Account ending in ******{0}.", match.Groups[1].Value);

// The example displays the following output:
//       Account ending in ******1999.
```

```
Dim greedyPattern As String = "\b.*([0-9]{4})\b"
Dim input1 As String = "1112223333 3992991999"
For Each match As Match In Regex.Matches(input1, greedypattern)
    Console.WriteLine("Account ending in ******{0}.", match.Groups(1).Value)
Next
' The example displays the following output:
'       Account ending in ******1999.
```

正则表达式无法匹配第一个数字，因为 `*` 量符尝试在整个字符串中尽可能多地匹配上一元素，所以它会在字符串末尾找到匹配。

这不是所需行为。相反，可以使用 `*?` 惰性量符，从这两个数字提取数字，如下面的示例所示。

```
string lazyPattern = @"\b.*?([0-9]{4})\b";
string input2 = "1112223333 3992991999";
foreach (Match match in Regex.Matches(input2, lazyPattern))
   Console.WriteLine("Account ending in ******{0}.", match.Groups[1].Value);

// The example displays the following output:
//       Account ending in ******3333.
//       Account ending in ******1999.
```

```
Dim lazyPattern As String = "\b.*?([0-9]{4})\b"
Dim input2 As String = "1112223333 3992991999"
For Each match As Match In Regex.Matches(input2, lazypattern)
    Console.WriteLine("Account ending in ******{0}.", match.Groups(1).Value)
Next
' The example displays the following output:
'       Account ending in ******3333.
'       Account ending in ******1999.
```

在大多数情况下，具有贪婪和惰性限定符的正则表达式返回相同匹配项。与匹配任何字符的通配符 (`.`) 元字符一起使用时，它们通常会返回不同的结果。

限定符和空匹配项

如果已找到最小捕获数，限定符 `*`、`+` 和 `{n,m}` 及对应的惰性限定符绝不会在空匹配项后重复 。此规则会在最大可能组捕获数是无限或接近无限时，阻止限定符在空的子表达式匹配项上进入无限循环。

例如，下面的代码展示了使用正则表达式模式 `(a?)*`（匹配零个或一个“a”字符零次或多次）调用 Regex.Match 方法的结果。请注意，一个捕获组捕获所有“a”以及 String.Empty，但没有第二个空匹配，因为第一个空匹配导致量符停止重复运行。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(a?)*";
      string input = "aaabbb";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}' at index {1}",
                        match.Value, match.Index);
      if (match.Groups.Count > 1) {
         GroupCollection groups = match.Groups;
         for (int grpCtr = 1; grpCtr <= groups.Count - 1; grpCtr++) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}",
                              grpCtr,
                              groups[grpCtr].Value,
                              groups[grpCtr].Index);
            int captureCtr = 0;
            foreach (Capture capture in groups[grpCtr].Captures) {
               captureCtr++;
               Console.WriteLine("      Capture {0}: '{1}' at index {2}",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'aaa' at index 0
//          Group 1: '' at index 3
//             Capture 1: 'a' at index 0
//             Capture 2: 'a' at index 1
//             Capture 3: 'a' at index 2
//             Capture 4: '' at index 3
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(a?)*"
        Dim input As String = "aaabbb"
        Dim match As Match = Regex.Match(input, pattern)
        Console.WriteLine("Match: '{0}' at index {1}",
                          match.Value, match.Index)
        If match.Groups.Count > 1 Then
            Dim groups As GroupCollection = match.Groups
            For grpCtr As Integer = 1 To groups.Count - 1
                Console.WriteLine("   Group {0}: '{1}' at index {2}",
                                  grpCtr,
                                  groups(grpCtr).Value,
                                  groups(grpCtr).Index)
                Dim captureCtr As Integer = 0
                For Each capture As Capture In groups(grpCtr).Captures
                    captureCtr += 1
                    Console.WriteLine("      Capture {0}: '{1}' at index {2}",
                                      captureCtr, capture.Value, capture.Index)
                Next
            Next
        End If
    End Sub
End Module
' The example displays the following output:
'       Match: 'aaa' at index 0
'          Group 1: '' at index 3
'             Capture 1: 'a' at index 0
'             Capture 2: 'a' at index 1
'             Capture 3: 'a' at index 2
'             Capture 4: '' at index 3
```

若要查看定义最小和最大捕获数的捕获组与定义固定捕获数的捕获组之间的实际差异，请考虑正则表达式模式 `(a\1|(?(1)\1)){0,2}` 和 `(a\1|(?(1)\1)){2}`。这两个正则表达式包含单个捕获组，其定义如下表所示。

“	“
`(a\1`	匹配“a”以及第一个捕获组的值...
`|(?(1)`	... 或测试是否定义了第一个捕获组。（请注意，`(?(1)` 构造不定义捕获组。）
`\1))`	如果第一个捕获组存在，则匹配其值。如果组不存在，组会匹配 String.Empty。

第一个正则表达式尝试与此模式匹配零到二次；第二个正则表达式尝试恰好匹配两次。由于第一个模式在首次捕获 String.Empty 时达到最小捕获数，因此它绝不会重复尝试匹配 `a\1`；`{0,2}` 量符仅允许在最后一个迭代中有空匹配。相反，第二个正则表达式匹配“a”，因为它会第二次计算 `a\1`；最小迭代数 2 会强制引擎在空匹配项后面重复。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern, input;

      pattern = @"(a\1|(?(1)\1)){0,2}";
      input = "aaabbb";

      Console.WriteLine("Regex pattern: {0}", pattern);
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match: '{0}' at position {1}.",
                        match.Value, match.Index);
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1; groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index);
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
      Console.WriteLine();

      pattern = @"(a\1|(?(1)\1)){2}";
      Console.WriteLine("Regex pattern: {0}", pattern);
      match = Regex.Match(input, pattern);
         Console.WriteLine("Matched '{0}' at position {1}.",
                           match.Value, match.Index);
      if (match.Groups.Count > 1) {
         for (int groupCtr = 1; groupCtr <= match.Groups.Count - 1; groupCtr++)
         {
            Group group = match.Groups[groupCtr];
            Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                              groupCtr, group.Value, group.Index);
            int captureCtr = 0;
            foreach (Capture capture in group.Captures) {
               captureCtr++;
               Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                 captureCtr, capture.Value, capture.Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Regex pattern: (a\1|(?(1)\1)){0,2}
//       Match: '' at position 0.
//          Group: 1: '' at position 0.
//             Capture: 1: '' at position 0.
//
//       Regex pattern: (a\1|(?(1)\1)){2}
//       Matched 'a' at position 0.
//          Group: 1: 'a' at position 0.
//             Capture: 1: '' at position 0.
//             Capture: 2: 'a' at position 0.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern, input As String

        pattern = "(a\1|(?(1)\1)){0,2}"
        input = "aaabbb"

        Console.WriteLine("Regex pattern: {0}", pattern)
        Dim match As Match = Regex.Match(input, pattern)
        Console.WriteLine("Match: '{0}' at position {1}.",
                          match.Value, match.Index)
        If match.Groups.Count > 1 Then
            For groupCtr As Integer = 1 To match.Groups.Count - 1
                Dim group As Group = match.Groups(groupCtr)
                Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                                  groupCtr, group.Value, group.Index)
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    captureCtr += 1
                    Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                      captureCtr, capture.Value, capture.Index)
                Next
            Next
        End If
        Console.WriteLine()

        pattern = "(a\1|(?(1)\1)){2}"
        Console.WriteLine("Regex pattern: {0}", pattern)
        match = Regex.Match(input, pattern)
        Console.WriteLine("Matched '{0}' at position {1}.",
                          match.Value, match.Index)
        If match.Groups.Count > 1 Then
            For groupCtr As Integer = 1 To match.Groups.Count - 1
                Dim group As Group = match.Groups(groupCtr)
                Console.WriteLine("   Group: {0}: '{1}' at position {2}.",
                                  groupCtr, group.Value, group.Index)
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    captureCtr += 1
                    Console.WriteLine("      Capture: {0}: '{1}' at position {2}.",
                                      captureCtr, capture.Value, capture.Index)
                Next
            Next
        End If
    End Sub
End Module
' The example displays the following output:
'       Regex pattern: (a\1|(?(1)\1)){0,2}
'       Match: '' at position 0.
'          Group: 1: '' at position 0.
'             Capture: 1: '' at position 0.
'
'       Regex pattern: (a\1|(?(1)\1)){2}
'       Matched 'a' at position 0.
'          Group: 1: 'a' at position 0.
'             Capture: 1: '' at position 0.
'             Capture: 2: 'a' at position 0.
```

请参阅

- 正则表达式语言 - 快速参考
- 回溯

正则表达式中的反向引用构造

2021/11/16 •

反向引用提供了标识字符串中的重复字符或子字符串的方便途径。例如，如果输入字符串包含某任意子字符串的多个匹配项，可以使用捕获组匹配第一个出现的子字符串，然后使用反向引用匹配后面出现的子字符串。

> **NOTE**
> 单独语法用于引用替换字符串中命名的和带编号的捕获组。有关更多信息，请参见 替代。

.NET 定义引用编号和命名捕获组的单独语言元素。若要详细了解捕获组，请参阅分组构造。

带编号的反向引用

带编号的反向引用使用以下语法：

`\` *number*

其中 *number* 是正则表达式中捕获组的序号位置。例如，`\4` 匹配第四个捕获组的内容。如果正则表达式模式中未定义 number，将会发生分析错误，并且正则表达式引擎会抛出 ArgumentException。例如，正则表达式 `\b(\w+)\s\1` 有效，因为 `(\w+)` 是表达式中的第一个也是唯一一个捕获组。`\b(\w+)\s\2` 无效，该表达式会因为没有捕获组编号 `\2` 而引发自变量异常。此外，如果 number 标识特定序号位置中的捕获组，但该捕获组已被分配了一个不同于其序号位置的数字名称，则正则表达式分析器还会引发 ArgumentException。

请注意八进制转义代码（如 `\16`）和使用相同表示法的 `\` number 反向引用之间的不明确问题。这种多义性可通过如下方式解决：

- 表达式 `\1` 到 `\9` 始终解释为反向应用，而不是八进制代码。
- 如果多位表达式的第一个数字是 8 或 9（如 `\80` 或 `\91`），该表达式将解释为文本。
- 对于编号为 `\10` 或更大值的表达式，如果存在与该编号对应的反向引用，则将该表达式视为反向引用；否则，将这些表达式解释为八进制代码。
- 如果正则表达式包含对未定义的组成员的反向引用，将会发生分析错误，并且正则表达式引擎会抛出 ArgumentException。

如果存在不明确问题，可以使用 `\k<` name `>` 表示法，此表示法非常明确，不会与八进制字符代码混淆。同样，诸如 `\xdd` 的十六进制代码也是明确的，不会与反向引用混淆。

下面的示例查找字符串中双写的单词字符。它定义一个由下列元素组成的正则表达式 `(\w)\1`。

“	“
`(\w)`	匹配单词字符，并将其分配给第一个捕获组。
`\1`	匹配值与第一捕获组相同的下一个字符。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(\w)\1";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(\w)\1"
        Dim input As String = "trellis llama webbing dresser swagger"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found '{0}' at position {1}.", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

命名的反向引用

使用以下语法定义命名的反向引用：

`\k<` *name* `>`

或：

`\k'` *name* `'`

其中，*name* 是正则表达式模式中定义的捕获组的名称。如果正则表达式模式中未定义 name，将会发生分析错误，并且正则表达式引擎会抛出 ArgumentException。

下面的示例查找字符串中双写的单词字符。它定义一个由下列元素组成的正则表达式 `(?<char>\w)\k<char>`。

“	“
`(?<char>\w)`	匹配字词字符，并将结果分配到 `char` 捕获组。
`\k<char>`	匹配下一个与 `char` 捕获组的值相同的字符。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<char>\w)\k<char>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?<char>\w)\k<char>"
        Dim input As String = "trellis llama webbing dresser swagger"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found '{0}' at position {1}.", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

已命名数值的反向引用

在具有 `\k` 的已命名反向引用中，name 也可以是 number 的字符串表示形式 。例如，下面的示例使用正则表达式 `(?<2>\w)\k<2>` 查找字符串中双写的单词字符。在此情况下，该示例定义了显式命名为“2”的捕获组，反向引用相应地命名为“2”。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<2>\w)\k<2>";
      string input = "trellis llama webbing dresser swagger";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found '{0}' at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found 'll' at position 3.
//       Found 'll' at position 8.
//       Found 'bb' at position 16.
//       Found 'ss' at position 25.
//       Found 'gg' at position 33.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?<2>\w)\k<2>"
        Dim input As String = "trellis llama webbing dresser swagger"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found '{0}' at position {1}.", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Found 'll' at position 3.
'       Found 'll' at position 8.
'       Found 'bb' at position 16.
'       Found 'ss' at position 25.
'       Found 'gg' at position 33.
```

如果 name 是 number 的字符串表示形式，且没有捕获组具有该名称，`\k<` name `>` 与反向引用 `\` number 相同，其中 number 是捕获的序号位置。在以下示例中，有名为 `char` 的单个捕获组。反向引用构造将其称为 `\k<1>`。正如示例中的输出所示，由于 `char` 是第一个捕获组，所以对 Regex.IsMatch 的调用成功。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<char>\w)\k<1>"));
      // Displays "True".
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Console.WriteLine(Regex.IsMatch("aa", "(?<char>\w)\k<1>"))
        ' Displays "True".
    End Sub
End Module
```

但是，如果 name 是 number 的字符串表示形式，并且已向该位置中的捕获组明确分配了数字名称，正则表达式分析器无法通过其序号位置识别捕获组 。相反，它会引发 ArgumentException。以下示例中的唯一捕获组名为“2”。由于 `\k` 结构用于定义名为“1”的反向引用，因此正则表达式分析器无法识别第一个捕获组并引发异常。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Console.WriteLine(Regex.IsMatch("aa", @"(?<2>\w)\k<1>"));
      // Throws an ArgumentException.
   }
}
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Console.WriteLine(Regex.IsMatch("aa", "(?<2>\w)\k<1>"))
        ' Throws an ArgumentException.
    End Sub
End Module
```

反向引用匹配什么内容

反向引用引用组的最新定义（从左向右匹配时，最靠近左侧的定义）。当组建立多个捕获时，反向引用会引用最新的捕获。

下面的示例包含正则表达式模式 `(?<1>a)(?<1>\1b)*`，该模式重新定义 \1 命名组。下表描述了正则表达式中的每个模式。

“	“
`(?<1>a)`	匹配字符“a”，并将结果分配到 `1` 捕获组。
`(?<1>\1b)*`	匹配 `1` 组的 0 更大发生次数以及“b”，并将结果分配到 `1` 捕获组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"(?<1>a)(?<1>\1b)*";
      string input = "aababb";
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("Match: " + match.Value);
         foreach (Group group in match.Groups)
            Console.WriteLine("   Group: " + group.Value);
      }
   }
}
// The example displays the following output:
//          Group: aababb
//          Group: abb
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?<1>a)(?<1>\1b)*"
        Dim input As String = "aababb"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Match: " + match.Value)
            For Each group As Group In match.Groups
                Console.WriteLIne("   Group: " + group.Value)
            Next
        Next
    End Sub
End Module
' The example display the following output:
'          Group: aababb
'          Group: abb
```

在比较正则表达式与输入字符串(“aababb”)时，正则表达式引擎执行以下操作：

1. 从该字符串的开头开始，成功将“a”与表达式 `(?<1>a)` 匹配。此时，`1` 组的值为“a”。
2. 继续匹配第二个字符，成功将字符串“ab”与表达式 `\1b` 或“ab”匹配。然后，将结果“ab”分配到 `\1`。
3. 继续匹配第四个字符。表达式 `(?<1>\1b)*` 要匹配零次或多次，因此会成功将字符串“abb”与表达式 `\1b` 匹配。然后，将结果“abb”分配回到 `\1`。

在本示例中，`*` 是循环限定符 -- 它将被重复计算，直到正则表达式引擎不能与它定义的模式匹配为止。循环限定符不会清除组定义。

如果某个组尚未捕获任何子字符串，则对该组的反向引用是不确定的，永远不会匹配。下面展示了正则表达式模式 `\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b` 的定义：

“	“
`\b`	在单词边界处开始匹配。
`(\p{Lu}{2})`	匹配两个大写字母。这是第一个捕获组。

"	"
`(\d{2})?`	匹配两个十进制数的零个或一个匹配项。这是第二个捕获组。
`(\p{Lu}{2})`	匹配两个大写字母。这是第三个捕获组。
`\b`	在单词边界处结束匹配。

输入字符串可以匹配此正则表达式，即使第二个捕获组定义的两个十进制数字都不存在。下面的示例显示了即使匹配成功，也仍会在两个成功的捕获组之间找到空捕获组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b";
      string[] inputs = { "AA22ZZ", "AABB" };
      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
         {
            Console.WriteLine("Match in {0}: {1}", input, match.Value);
            if (match.Groups.Count > 1)
            {
               for (int ctr = 1; ctr <= match.Groups.Count - 1; ctr++)
               {
                  if (match.Groups[ctr].Success)
                     Console.WriteLine("Group {0}: {1}",
                                       ctr, match.Groups[ctr].Value);
                  else
                     Console.WriteLine("Group {0}: <no match>", ctr);
               }
            }
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match in AA22ZZ: AA22ZZ
//       Group 1: AA
//       Group 2: 22
//       Group 3: ZZ
//
//       Match in AABB: AABB
//       Group 1: AA
//       Group 2: <no match>
//       Group 3: BB
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\p{Lu}{2})(\d{2})?(\p{Lu}{2})\b"
        Dim inputs() As String = {"AA22ZZ", "AABB"}
        For Each input As String In inputs
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then
                Console.WriteLine("Match in {0}: {1}", input, match.Value)
                If match.Groups.Count > 1 Then
                    For ctr As Integer = 1 To match.Groups.Count - 1
                        If match.Groups(ctr).Success Then
                            Console.WriteLine("Group {0}: {1}", _
                                              ctr, match.Groups(ctr).Value)
                        Else
                            Console.WriteLine("Group {0}: <no match>", ctr)
                        End If
                    Next
                End If
            End If
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       Match in AA22ZZ: AA22ZZ
'       Group 1: AA
'       Group 2: 22
'       Group 3: ZZ
'
'       Match in AABB: AABB
'       Group 1: AA
'       Group 2: <no match>
'       Group 3: BB
```

另请参阅

- 正则表达式语言 - 快速参考

正则表达式中的替换构造

2021/11/16 •

替换构造可修改正则表达式以启用 either/or 或条件匹配。.NET 支持三种替换构造：

- 利用 | 的模式匹配
- 利用 (?(expression)yes|no) 的条件匹配
- 基于有效的捕获组的条件匹配

利用 | 的模式匹配

可以使用竖线 (`|`) 字符匹配一系列模式中的任何一种模式，其中 `|` 字符用于分隔每个模式。

与正向字符集一样，`|` 字符可用于匹配多个字符中的任意一个字符。以下示例使用正向字符集和 either/or 模式匹配（使用 `|` 字符）查找字符串中单词“gray”或“grey”的匹配项。在该示例中，`|` 字符生成了更为详细的正则表达式。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Regular expression using character class.
      string pattern1 = @"\bgr[ae]y\b";
      // Regular expression using either/or.
      string pattern2 = @"\bgr(a|e)y\b";

      string input = "The gray wolf blended in among the grey rocks.";
      foreach (Match match in Regex.Matches(input, pattern1))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern2))
         Console.WriteLine("'{0}' found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       'gray' found at position 4
//       'grey' found at position 35
//
//       'gray' found at position 4
//       'grey' found at position 35
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        ' Regular expression using character class.
        Dim pattern1 As String = "\bgr[ae]y\b"
        ' Regular expression using either/or.
        Dim pattern2 As String = "\bgr(a|e)y\b"

        Dim input As String = "The gray wolf blended in among the grey rocks."
        For Each match As Match In Regex.Matches(input, pattern1)
            Console.WriteLine("'{0}' found at position {1}", _
                              match.Value, match.Index)
        Next
        Console.WriteLine()
        For Each match As Match In Regex.Matches(input, pattern2)
            Console.WriteLine("'{0}' found at position {1}", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       'gray' found at position 4
'       'grey' found at position 35
'
'       'gray' found at position 4
'       'grey' found at position 35
```

使用 `|` 字符的正则表达式 `\bgr(a|e)y\b` 的解释如下表所示：

“	“
`\b`	在单词边界处开始。
`gr`	匹配字符“gr”。
`(a|e)`	匹配“a”或“e”。
`y\b`	匹配单词边界中的“y”。

还可以使用 `|` 字符执行具有多个字符或子表达式（包含任意组合的字符常量和正则表达式语言元素）的 either/or 匹配。（字符类不提供此功能。）下面的示例使用 `|` 字符提取美国社会安全号码 (SSN)（格式为 ddd-dd-dddd 的 9 位数字），或美国雇主标识号 (EIN)（格式为 dd-ddddddd 的 9 位数字）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
        Console.WriteLine("Matches for {0}:", pattern)
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

正则表达式 `\b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` 可以解释为下表中所示内容：

"	"
`\b`	在单词边界处开始。
`(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})`	匹配以下其中一个内容：连字符连接的两个十进制数字和七个十进制数字；或三个十进制数字后接连字符，后接两个十进制数字，后接另一个连字符，然后再接四个十进制数字。
`\b`	在单词边界处结束匹配。

条件匹配的表达式

此语言元素尝试根据是否可以匹配初始模式来匹配两种模式之一。语法为：

`(?(` *expression* `)` *yes* `|` *no* `)`

其中，*expression* 是要匹配的初始模式，*yes* 是当匹配 *expression* 时要匹配的模式，而 *no* 是未匹配 *expression* 时要匹配的可选模式。正则表达式引擎将 *expression* 视为一个宽度为零的断言；也就是说，正则表达式引擎在计算 *expression* 之后，不再处理输入流的后续数据。因此，该构造是等效于以下语法：

`(?(?=` *expression* `)` *yes* `|` *no* `)`

其中 `(?=` expression `)` 是宽度为零的断言构造。(有关详细信息，请参阅分组构造。)由于正则表达式引擎将 expression 解释为定位点(零宽断言)，因此 expression 必须是零宽断言(有关详细信息，请参阅定位标记)，或者是也包含在 yes 中的子表达式。否则，无法匹配 *yes* 模式。

> **NOTE**
> 如果 expression 是命名捕获组或带编号的捕获组，则备用构造将被解释为捕获测试；有关详细信息，请参阅下一部分基于有效捕获组的条件匹配。换而言之，正则表达式引擎不会尝试匹配捕获的子字符串，而是测试该组是否存在。

下面的示例是利用 | 的 Either/Or 模式匹配一节中的示例变体。它使用条件匹配来确定单词边界之后的前三个字符是否是后接一个连字符的两个数字。如果是，则将尝试匹配美国雇主标识号 (EIN)。如果不是，则将尝试匹配美国社会保障号 (SSN)。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
        Console.WriteLine("Matches for {0}:", pattern)
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Matches for \b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

正则表达式模式 `\b(?(\d{2}-)\d{2}-\d{7}|\d{3}-\d{2}-\d{4})\b` 的释义如下表所示：

"	"
`\b`	在单词边界处开始。
`(?(\d{2}-)`	确定接下来的三个字符是否由两个数字后接一个连字符组成。

“	“
`\d{2}-\d{7}`	如果前面的模式匹配，则匹配后接一个连字符和七个数字的两个数字。
`\d{3}-\d{2}-\d{4}`	如果前面的模式不匹配，则匹配三个十进制数字，后接一个连字符，再接两个十进制数字，再接另一个连字符，再接四个十进制数字。
`\b`	与字边界匹配。

基于有效的捕获组的条件匹配

此语言元素尝试根据是否已经匹配指定的捕获组来匹配两种模式之一。语法为：

`(?(` *name* `)` *yes* `|` *no* `)`

or

`(?(` *number* `)` *yes* `|` *no* `)`

其中， *name* 是捕获组的名称， *number* 是捕获组的编号；*yes* 是当 *name* 或 *number* 具有匹配项时要匹配的表达式；*no* 是当不具有匹配项时要匹配的可选表达式。

如果 *name* 与正则表达式模式中所用捕获组的名称不对应，则替换构造将解释为表达式测试，如上一节中所述。通常，这意味着 *expression* 的计算结果为 `false` 。如果 *number* 与正则表达式模式中所用带编号的捕获组不对应，则正则表达式引擎将引发 ArgumentException。

下面的示例是利用 | 的 Either/Or 模式匹配一节中的示例变体。它使用一个名为 `n2` 的捕获组，其中包含两个数字，后接一个连字符。替换构造测试此捕获组是否在输入字符串中找到匹配项。如果有匹配项，则替换构造会尝试匹配九位数的美国雇主标识号 (EIN)。如果没有匹配项，则将尝试匹配九位数的美国社会保障号 (SSN)。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Matches for \b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
        Console.WriteLine("Matches for {0}:", pattern)
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
    End Sub
End Module
```

正则表达式模式 `\b(?<n2>\d{2}-)?(?(n2)\d{7}|\d{3}-\d{2}-\d{4})\b` 的释义如下表所示：

"	"
`\b`	在单词边界处开始。
`(?<n2>\d{2}-)?`	匹配两个数字后接一个连字符的零或一个匹配项。命名此捕获组 `n2`。
`(?(n2)`	测试输入字符串中是否有 `n2` 的匹配项。
`\d{7}`	如果找到 `n2` 的匹配项，则匹配 7 个十进制数字。
`|\d{3}-\d{2}-\d{4}`	如果未找到 `n2` 的匹配项，则匹配 3 个十进制数字，后接一个连字符，再接 2 个十进制数字，再接另一个连字符，再接 4 个十进制数字。
`\b`	与字边界匹配。

下面示例中显示此示例变体使用编号组而非命名组。正则表达式模式为 `\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b`。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b";
      string input = "01-9999999 020-333333 777-88-9999";
      Console.WriteLine("Matches for {0}:", pattern);
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
   }
}
// The example display the following output:
//       Matches for \b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b:
//          01-9999999 at position 0
//          777-88-9999 at position 22
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b"
        Dim input As String = "01-9999999 020-333333 777-88-9999"
        Console.WriteLine("Matches for {0}:", pattern)
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Matches for \b(\d{2}-)?(?(1)\d{7}|\d{3}-\d{2}-\d{4})\b:
'          01-9999999 at position 0
'          777-88-9999 at position 22
```

请参阅

- 正则表达式语言 - 快速参考

正则表达式中的替代

2021/11/16 •

替换是只能在替换模式中识别的语言元素。它们使用正则表达式模式定义全部或部分用于替换输入字符串中的匹配文本的文本。替换模式可以包含一个或多个替换以及本文字符。提供替换模式以将拥有 Regex.Replace 参数的 `replacement` 方法重载至 Match.Result 方法。该方法将匹配的模式替换为 `replacement` 参数定义的模式。

.NET 定义下表列出的替换元素。

"	"
$ number	包括替换字符串中的由 *number* 标识的捕获组所匹配的最后一个子字符串，其中 *number* 是一个十进制值。有关详细信息，请参阅 替换已编号的组。
${ name }	包括替换字符串中由 `(?<` *name* `>` `)` 指定的命名组所匹配的最后一个子字符串。有关详细信息，请参阅 替换命名组。
$$	包括替换字符串中的单个"$"文本。有关详细信息，请参阅 替换"$"符号。
$&	包括替换字符串中整个匹配项的副本。有关详细信息，请参阅 替换整个匹配项。
$`	包括替换字符串中的匹配项前的输入字符串的所有文本。有关详细信息，请参阅 替换匹配项前的文本。
$'	包括替换字符串中的匹配项后的输入字符串的所有文本。有关详细信息，请参阅 替换匹配项后的文本。
$+	包括在替换字符串中捕获的最后一个组。有关详细信息，请参阅 替换最后捕获的组。
$_	包括替换字符串中的整个输入字符串。有关详细信息，请参阅 替换整个输入字符串。

替换元素和替换模式

替换是替换模式中唯一可识别的特殊构造。与任何字符匹配的其他正则表达式语言元素（包括字符转义和句点 (`.`)）均不受支持。同样，替换语言元素只能在替换模式中识别，并且在正则表达式模式中永远无效。

可以出现在正则表达式模式或替换中的唯一字符是 `$` 字符，尽管它在每个上下文中具有不同的含义。在正则表达式模式中，`$` 是与字符串的末尾匹配的定位点。在替换模式中，`$` 指示替换的开头。

> **NOTE**
> 对于类似于正则表达式中替换模式的功能，使用反向引用。有关反向引用的更多信息，请参见 反向引用构造。

替换已编号的组

`$` *number* 语言元素包括替换字符串中 *number* 捕获组所匹配的最后一个子字符串，其中 *number* 是捕获组的索

引。例如，替换模式 `$1` 指示匹配的子字符串将由捕获的第一个组替换。有关为已编号的捕获组的详细信息，请参见 分组构造。

`$` 后面的所有数字解释为属于 *number* 组。如果这不是你想要的结果，可改为替换命名组。例如，可以使用替换字符串 `${1}1` 而不是 `$11` 来将替换字符串定义为带数字"1"的首个捕获组的值。有关详细信息，请参阅 替换命名组。

没有使用 `(?<`*name*`>)` 语法显式分配的名称的捕获组从左到右进行编号（从 1 开始）。命名组还从左到右进行编号，从比最后一个未命名组的索引大 1 的值开始。例如，在正则表达式 `(\w)(?<digit>\d)` 中，`digit` 命名组的索引为 2。

如果 *number* 未指定在正则表达式模式中定义的有效捕获组，`$`*number* 将被解释为用于替换每个匹配项的文本字符序列。

下面的示例使用 `$`*number* 替换去除十进制值中的货币符号。它移除在货币值的开头或末尾找到的货币符号，并识别两个最常见的小数点分隔符（"."和","）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*";
      string replacement = "$1";
      string input = "$16.32 12.19 £16.29 €18.29  €18,29";
      string result = Regex.Replace(input, pattern, replacement);
      Console.WriteLine(result);
   }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*"
        Dim replacement As String = "$1"
        Dim input As String = "$16.32 12.19 £16.29 €18.29  €18,29"
        Dim result As String = Regex.Replace(input, pattern, replacement)
        Console.WriteLine(result)
    End Sub
End Module
' The example displays the following output:
'       16.32 12.19 16.29 18.29  18,29
```

正则表达式模式 `\p{Sc}*(\s?\d+[.,]?\d*)\p{Sc}*` 的定义如下表所示。

"	"
`\p{Sc}*`	与零个或多个货币符号字符匹配。
`\s?`	匹配零个或一个空白字符。
`\d+`	匹配一个或多个十进制数字。

"	"
`[.,]?`	与零个或一个句点或逗号匹配。
`\d*`	匹配零个或多个十进制数字。
`(\s?\d+[.,]?\d*)`	匹配空白，后跟一个或多个十进制数，再后跟零个或一个句点或逗号，后跟零个或多个十进制数。这是第一个捕获组。因为替换模式为 `$1`，所以调用 Regex.Replace 方法会将整个匹配的子字符串替换为此捕获组。

替换命名组

`${`*name*`}` 语言元素替换 *name* 捕获组匹配的最后一个子字符串，其中 *name* 是 `(?<`*name*`>)` 语言元素所定义的捕获组名称。有关命名的捕获组的详细信息，请参见 分组构造。

如果 *name* 未指定正则表达式模式中定义的有效的命名捕获组但包含数字，则 `${`*name*`}` 被解释为已编号的组。

如果 *name* 既未指定有效的命名捕获组也未指定正则表达式模式中定义的有效的已编号捕获组，则 `${`*name*`}` 被解释为用于替换每个匹配的文本字符序列。

下面的示例使用 `${`*name*`}` 替换去除十进制值中的货币符号。它移除在货币值的开头或末尾找到的货币符号，并识别两个最常见的小数点分隔符（"."和","）。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*";
      string replacement = "${amount}";
      string input = "$16.32 12.19 £16.29 €18.29  €18,29";
      string result = Regex.Replace(input, pattern, replacement);
      Console.WriteLine(result);
   }
}
// The example displays the following output:
//       16.32 12.19 16.29 18.29  18,29
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\p{Sc}*(?<amount>\s?\d+[.,]?\d*)\p{Sc}*"
        Dim replacement As String = "${amount}"
        Dim input As String = "$16.32 12.19 £16.29 €18.29  €18,29"
        Dim result As String = Regex.Replace(input, pattern, replacement)
        Console.WriteLine(result)
    End Sub
End Module
' The example displays the following output:
'       16.32 12.19 16.29 18.29  18,29
```

正则表达式模式 `\p{Sc}*(?<amount>\s?\d[.,]?\d*)\p{Sc}*` 的定义如下表所示。

"	"
`\p{Sc}*`	与零个或多个货币符号字符匹配。
`\s?`	匹配零个或一个空白字符。
`\d+`	匹配一个或多个十进制数字。
`[.,]?`	与零个或一个句点或逗号匹配。
`\d*`	匹配零个或多个十进制数字。
`(?<amount>\s?\d[.,]?\d*)`	匹配空白，后跟一个或多个十进制数，再后跟零个或一个句点或逗号，后跟零个或多个十进制数。这是名为 `amount` 的捕获组。因为替换模式为 `${amount}`，所以调用 Regex.Replace 方法会将整个匹配的子字符串替换为此捕获组。

替换“$”字符

`$$` 替换将在替换的字符串中插入文本“$”字符。

下面的示例使用 NumberFormatInfo 对象确定当前区域性的货币符号及其在货币字符串中的位置。然后，它动态构建一个正则表达式模式和一个替换模式。如果示例在当前区域性为 en-US 的计算机上运行，则它将生成正则表达式模式 `\b(\d+)(\.(\d+))?` 和替换模式 `$$ $1$2`。替换模式将匹配的文本替换为一个后跟第一个和第二个捕获组的货币符号和空格。

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      // Define array of decimal values.
      string[] values= { "16.35", "19.72", "1234", "0.99"};
      // Determine whether currency precedes (True) or follows (False) number.
      bool precedes = NumberFormatInfo.CurrentInfo.CurrencyPositivePattern % 2 == 0;
      // Get decimal separator.
      string cSeparator = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator;
      // Get currency symbol.
      string symbol = NumberFormatInfo.CurrentInfo.CurrencySymbol;
      // If symbol is a "$", add an extra "$".
      if (symbol == "$") symbol = "$$";

      // Define regular expression pattern and replacement string.
      string pattern = @"\b(\d+)(" + cSeparator + @"(\d+))?";
      string replacement = "$1$2";
      replacement = precedes ? symbol + " " + replacement : replacement + " " + symbol;
      foreach (string value in values)
         Console.WriteLine("{0} --> {1}", value, Regex.Replace(value, pattern, replacement));
   }
}
// The example displays the following output:
//       16.35 --> $ 16.35
//       19.72 --> $ 19.72
//       1234 --> $ 1234
//       0.99 --> $ 0.99
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        ' Define array of decimal values.
        Dim values() As String = {"16.35", "19.72", "1234", "0.99"}
        ' Determine whether currency precedes (True) or follows (False) number.
        Dim precedes As Boolean = (NumberFormatInfo.CurrentInfo.CurrencyPositivePattern Mod 2 = 0)
        ' Get decimal separator.
        Dim cSeparator As String = NumberFormatInfo.CurrentInfo.CurrencyDecimalSeparator
        ' Get currency symbol.
        Dim symbol As String = NumberFormatInfo.CurrentInfo.CurrencySymbol
        ' If symbol is a "$", add an extra "$".
        If symbol = "$" Then symbol = "$$"

        ' Define regular expression pattern and replacement string.
        Dim pattern As String = "\b(\d+)(" + cSeparator + "(\d+))?"
        Dim replacement As String = "$1$2"
        replacement = If(precedes, symbol + " " + replacement, replacement + " " + symbol)
        For Each value In values
            Console.WriteLine("{0} --> {1}", value, Regex.Replace(value, pattern, replacement))
        Next
    End Sub
End Module
' The example displays the following output:
'       16.35 --> $ 16.35
'       19.72 --> $ 19.72
'       1234 --> $ 1234
'       0.99 --> $ 0.99
```

正则表达式模式 `\b(\d+)(\.(\d+))?` 的定义如下表所示。

“	“
`\b`	从字边界开始进行匹配。
`(\d+)`	匹配一个或多个十进制数字。这是第一个捕获组。
`\.`	与句点(小数点分隔符)匹配。
`(\d+)`	匹配一个或多个十进制数字。这是第三个捕获组。
`(\.(\d+))?`	与零个或一个后跟一个或多个十进制数字的句点匹配。这是第二个捕获组。

替换整个匹配项

`$&` 替换包括替换字符串中的整个匹配项。通常，它用于将子字符串添加至匹配字符串的开头或末尾。例如，`($&)` 替换模式向每个匹配项的开头和结尾添加括号。如果没有匹配项，则 `$&` 替换将不起作用。

下面的示例使用 `$&` 替换在存储于字符串数组中的书名的开头和结尾添加引号。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^(\w+\s?)+$";
      string[] titles = { "A Tale of Two Cities",
                          "The Hound of the Baskervilles",
                          "The Protestant Ethic and the Spirit of Capitalism",
                          "The Origin of Species" };
      string replacement = "\"$&\"";
      foreach (string title in titles)
         Console.WriteLine(Regex.Replace(title, pattern, replacement));
   }
}
// The example displays the following output:
//       "A Tale of Two Cities"
//       "The Hound of the Baskervilles"
//       "The Protestant Ethic and the Spirit of Capitalism"
//       "The Origin of Species"
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^(\w+\s?)+$"
        Dim titles() As String = {"A Tale of Two Cities", _
                                  "The Hound of the Baskervilles", _
                                  "The Protestant Ethic and the Spirit of Capitalism", _
                                  "The Origin of Species"}
        Dim replacement As String = """$&"""
        For Each title As String In titles
            Console.WriteLine(Regex.Replace(title, pattern, replacement))
        Next
    End Sub
End Module
' The example displays the following output:
'       "A Tale of Two Cities"
'       "The Hound of the Baskervilles"
'       "The Protestant Ethic and the Spirit of Capitalism"
'       "The Origin of Species"
```

正则表达式模式 `^(\w+\s?)+$` 的定义如下表所示。

"	"
`^`	从输入字符串的开头部分开始匹配。
`(\w+\s?)+`	匹配一个或多个单词字符后跟零个或一个空白字符的模式一次或多次。
`$`	匹配输入字符串的末尾部分。

`"$&"` 替换模式将文本引号添加到每个匹配项的开头和结尾。

替换匹配项前的文本

`$`` 替换将匹配的字符串替换为匹配项前面的整个输入字符串。即，它将在删除匹配的文本时重复输入字符串，

直至匹配。匹配文本后面的任何文本在结果字符串中保持不变。如果输入字符串中有多个匹配项，则替换文本将派生自原始输入字符串，而不是派生自文本已由早期匹配项替换的字符串。(说明如示例所示。)如果没有匹配项，则 `$`` 替换将不起作用。

下面的示例使用正则表达式模式 `\d+` 来匹配输入字符串中一个或多个十进制数字的序列。替换字符串 `$`` 将这些数字替换为该匹配项前的文本。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$`";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);

      Console.WriteLine("Input string:  {0}", input);
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddeeaa1bb2cc3dd4ee
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "aa1bb2cc3dd4ee5"
        Dim pattern As String = "\d+"
        Dim substitution As String = "$`"
        Console.WriteLine("Matches:")
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
        Console.WriteLine("Input string:  {0}", input)
        Console.WriteLine("Output string: " + _
                          Regex.Replace(input, pattern, substitution))
    End Sub
End Module
' The example displays the following output:
'    Matches:
'       1 at position 2
'       2 at position 5
'       3 at position 8
'       4 at position 11
'       5 at position 14
'    Input string:  aa1bb2cc3dd4ee5
'    Output string: aaaabbaa1bbccaa1bb2ccddaa1bb2cc3ddeeaa1bb2cc3dd4ee
```

在此示例中，输入字符串 `"aa1bb2cc3dd4ee5"` 包含五个匹配项。下表说明了 `$`` 替换如何使正则表达式引擎替换

输入字符串中的每个匹配项。插入文本在结果列中以粗体显示。

ɪɪ	ɪɪ	ɪɪɪɪɪɪɪɪ	ɪɪɪɪɪ
1	2	aa	aa **aa** bb2cc3dd4ee5
2	5	aa1bb	aaaabb **aa1bb** cc3dd4ee5
3	8	aa1bb2cc	aaaabbaa1bbcc **aa1bb2cc** dd4ee5
4	11	aa1bb2cc3dd	aaaabbaa1bbccaa1bb2ccdd **aa1bb2cc3dd** ee5
5	14	aa1bb2cc3dd4ee	aaaabbaa1bbccaa1bb2ccdd aa1bb2cc3ddee **aa1bb2cc3dd4ee**

替换匹配项后的文本

`$'` 替换将匹配的字符串替换为匹配项后的整个输入字符串。即，它将在删除匹配的文本时重复匹配项后的输入字符串。匹配文本前面的任何文本在结果字符串中保持不变。如果没有匹配项，则 `$'` 替换将不起作用。

下面的示例使用正则表达式模式 `\d+` 来匹配输入字符串中一个或多个十进制数字的序列。替换字符串 `$'` 将这些数字替换为匹配项后的文本。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aa1bb2cc3dd4ee5";
      string pattern = @"\d+";
      string substitution = "$'";
      Console.WriteLine("Matches:");
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("   {0} at position {1}", match.Value, match.Index);
      Console.WriteLine("Input string:  {0}", input);
      Console.WriteLine("Output string: " +
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//    Matches:
//       1 at position 2
//       2 at position 5
//       3 at position 8
//       4 at position 11
//       5 at position 14
//    Input string:  aa1bb2cc3dd4ee5
//    Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "aa1bb2cc3dd4ee5"
        Dim pattern As String = "\d+"
        Dim substitution As String = "$'"
        Console.WriteLine("Matches:")
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("   {0} at position {1}", match.Value, match.Index)
        Next
        Console.WriteLine("Input string:  {0}", input)
        Console.WriteLine("Output string: " + _
                          Regex.Replace(input, pattern, substitution))
    End Sub
End Module
' The example displays the following output:
'    Matches:
'       1 at position 2
'       2 at position 5
'       3 at position 8
'       4 at position 11
'       5 at position 14
'    Input string:  aa1bb2cc3dd4ee5
'    Output string: aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee
```

在此示例中，输入字符串 `"aa1bb2cc3dd4ee5"` 包含五个匹配项。下表说明了 `$'` 替换如何使正则表达式引擎替换输入字符串中的每个匹配项。插入文本在结果列中以粗体显示。

1	2	bb2cc3dd4ee5	aa **bb2cc3dd4ee5** bb2cc3dd4ee5
2	5	cc3dd4ee5	aabb2cc3dd4ee5bb **cc3dd4ee5** cc3dd4ee5
3	8	dd4ee5	aabb2cc3dd4ee5bbcc3dd4ee5cc **dd4ee5** dd4ee5
4	11	ee5	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5dd **ee5** ee5
5	14	String.Empty	aabb2cc3dd4ee5bbcc3dd4ee5ccdd4ee5ddee5ee

替换最后捕获的组

`$+` 替换将匹配的字符串替换为最后捕获的组。如果没有捕获组，或者最后的捕获组的值是 String.Empty，则 `$+` 替换将不起作用。

下面的示例标识字符串中的重复单词，并使用 `$+` 替换将其替换为该单词的单一匹配项。RegexOptions.IgnoreCase 选项用于确保大小写不同但其他内容都相同的单词被认为是重复的。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s\1\b";
      string substitution = "$+";
      string input = "The the dog jumped over the fence fence.";
      Console.WriteLine(Regex.Replace(input, pattern, substitution,
                        RegexOptions.IgnoreCase));
   }
}
// The example displays the following output:
//      The dog jumped over the fence.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\w+)\s\1\b"
        Dim substitution As String = "$+"
        Dim input As String = "The the dog jumped over the fence fence."
        Console.WriteLine(Regex.Replace(input, pattern, substitution, _
                                        RegexOptions.IgnoreCase))
    End Sub
End Module
' The example displays the following output:
'      The dog jumped over the fence.
```

正则表达式模式 `\b(\w+)\s\1\b` 的定义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`(\w+)`	匹配一个或多个单词字符。这是第一个捕获组。
`\s`	与空白字符匹配。
`\1`	与第一个捕获的组匹配。
`\b`	在单词边界处结束匹配。

替换整个输入字符串

`$_` 替换将匹配的字符串替换为整个输入字符。即，它将删除匹配的文本并将其替换为整个字符串(包括匹配的文本)。

下面的示例匹配输入字符串中的一个或多个十进制数字。它使用 `$_` 替换来将其替换为整个输入字符串。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "ABC123DEF456";
      string pattern = @"\d+";
      string substitution = "$_";
      Console.WriteLine("Original string:          {0}", input);
      Console.WriteLine("String with substitution: {0}",
                        Regex.Replace(input, pattern, substitution));
   }
}
// The example displays the following output:
//       Original string:          ABC123DEF456
//       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "ABC123DEF456"
        Dim pattern As String = "\d+"
        Dim substitution As String = "$_"
        Console.WriteLine("Original string:          {0}", input)
        Console.WriteLine("String with substitution: {0}", _
                          Regex.Replace(input, pattern, substitution))
    End Sub
End Module
' The example displays the following output:
'       Original string:          ABC123DEF456
'       String with substitution: ABCABC123DEF456DEFABC123DEF456
```

在此示例中，输入字符串 `"ABC123DEF456"` 包含两个匹配项。下表说明了 `$_` 替换如何使正则表达式引擎替换输入字符串中的每个匹配项。插入文本在结果列中以粗体显示。

匹配	索引	长度	结果
1	3	123	ABC **ABC123DEF456** DEF456
2	5	456	ABCABC123DEF456DEF **ABC123DEF456**

请参阅

- 正则表达式语言 - 快速参考

正则表达式选项

2021/11/16 •

默认情况下，正则表达式模式中带有任意文本字符的输入字符串比较区分大小写，正则表达式模式中的空白将被解释为文本空白字符且正则表达式中的捕获组通过隐式和显式命名。可通过指定正则表达式选项修改默认正则表达式行为的这些和其他数个方面。列于下表的这些选项，可将内联作为正则表达式的一部分包含，或者可将它们作为 System.Text.RegularExpressions.Regex 枚举值提供给 System.Text.RegularExpressions.RegexOptions 类构造函数或静态模式匹配方法。

REGEXOPTIONS		
None	不可用	使用默认行为。有关更多信息，请参见默认选项。
IgnoreCase	`i`	使用不区分大小写的匹配。有关更多信息，请参见不区分大小写的匹配。
Multiline	`m`	使用多线模式，其中 `^` 和 `$` 匹配每行的开头和末尾（不是输入字符串的开头和末尾）。有关更多信息，请参见多行模式。
Singleline	`s`	使用单行模式，其中的句号 (.) 匹配每个字符（而不是除了 `\n` 以外的每个字符）。有关详细信息，请参阅单行模式。
ExplicitCapture	`n`	不捕获未命名的组。唯一有效的捕获是显式命名或编号的 `(?<` name `>` subexpression `)` 形式的组 。有关更多信息，请参见仅显式捕获。
Compiled	不可用	将正则表达式编译为程序集。有关更多信息，请参见已编译的正则表达式。
IgnorePatternWhitespace	`x`	从模式中排除保留的空白并启用数字符号 (`#`) 后的注释。有关更多信息，请参见忽略空白。
RightToLeft	不可用	更改搜索方向。搜索是从右向左而不是从左向右进行。有关更多信息，请参见从右向左模式。
ECMAScript	不可用	为表达式启用符合 ECMAScript 的行为。有关更多信息，请参见 ECMAScript 匹配行为。
CultureInvariant	不可用	忽略语言的区域性差异。有关更多信息，请参见使用固定区域性的比较。

指定选项

可以用下面三种方法之一指定正则表达式的选项：

- 在 `options` 类构造函数或静态(在 Visual Basic 中为 System.Text.RegularExpressions.Regex)模式匹配方法的 `Shared` 参数中，如 Regex(String, RegexOptions) 或 Regex.Match(String, String, RegexOptions)。`options` 参数是 System.Text.RegularExpressions.RegexOptions 枚举值的按位“或”组合。

  当通过使用类构造函数的 `options` 参数，将选项提供给 Regex 实例时，这些选项将分配给 System.Text.RegularExpressions.RegexOptions 属性。然而，System.Text.RegularExpressions.RegexOptions 属性不会在正则表达式模式本身中反映内联选项。

  下面的示例进行了这方面的演示。在标识以字母“d”开头的单词时，它使用 `options` 方法的 Regex.Match(String, String, RegexOptions) 参数来启用不区分大小写匹配和忽略模式空白。

```
string pattern = @"d \w+ \s";
string input = "Dogs are decidedly good pets.";
RegexOptions options = RegexOptions.IgnoreCase | RegexOptions.IgnorePatternWhitespace;

foreach (Match match in Regex.Matches(input, pattern, options))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "d \w+ \s"
Dim input As String = "Dogs are decidedly good pets."
Dim options As RegexOptions = RegexOptions.IgnoreCase Or RegexOptions.IgnorePatternWhitespace

For Each match As Match In Regex.Matches(input, pattern, options)
    Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

- 通过在包含语法 `(?imnsx-imnsx)` 的正则表达式模式中应用内联选项。该选项从选项定义为模式末尾的点应用于该模式，或应用于另一内联选项未定义选项的点。请注意，Regex 实例的 System.Text.RegularExpressions.RegexOptions 属性不会反映这些内联选项。有关详细信息，请参阅其他构造主题。

  下面的示例进行了这方面的演示。在标识以字母“d”开头的单词时，它使用内联选项来启用不区分大小写匹配和忽略模式空白。

```
string pattern = @"(?ix) d \w+ \s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix) d \w+ \s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

- 通过在包含语法 `(?imnsx-imnsx:` subexpression `)` 的正则表达式模式的特定分组构造中，应用内联选项。一组选项前面没有符号用于打开该设置；一组选项前面的减号用于关闭该设置。（无论选项是启用还是禁用，`?` 都是所需的语言构造语法的固定部分。）选项只应用于该组。有关详细信息，请参阅分组构造。

  下面的示例进行了这方面的演示。在标识以字母“d”开头的单词时，它使用分组构造中的内联选项来启用不区分大小写匹配和忽略模式空白。

```
string pattern = @"\b(?ix: d \w+)\s";
string input = "Dogs are decidedly good pets.";

foreach (Match match in Regex.Matches(input, pattern))
   Console.WriteLine("'{0}// found at index {1}.", match.Value, match.Index);
// The example displays the following output:
//    'Dogs // found at index 0.
//    'decidedly // found at index 9.
```

```
Dim pattern As String = "\b(?ix: d \w+)\s"
Dim input As String = "Dogs are decidedly good pets."

For Each match As Match In Regex.Matches(input, pattern)
    Console.WriteLine("'{0}' found at index {1}.", match.Value, match.Index)
Next
' The example displays the following output:
'    'Dogs ' found at index 0.
'    'decidedly ' found at index 9.
```

如果选项指定为内联，一个选项或一组选项前面的减号 (`-`) 用于关闭这些选项。例如，内联构造 `(?ix-ms)` 将打开 RegexOptions.IgnoreCase 和 RegexOptions.IgnorePatternWhitespace 选项而关闭 RegexOptions.Multiline 和 RegexOptions.Singleline 选项。默认情况下，关闭所有正则表达式选项。

> **NOTE**
> 如果构造函数或方法调用的 `options` 形参中指定的正则表达式选项与正则表达式模式中的内联指定的选项冲突，那么将使用该内联选项。

可为下面的五个正则表达式选项同时设置选项形参和内联：

- RegexOptions.IgnoreCase
- RegexOptions.Multiline
- RegexOptions.Singleline
- RegexOptions.ExplicitCapture
- RegexOptions.IgnorePatternWhitespace

可为下面的五个正则表达式选项设置使用 `options` 形参，但不能为其设置内联：

- RegexOptions.None
- RegexOptions.Compiled
- RegexOptions.RightToLeft
- RegexOptions.CultureInvariant
- RegexOptions.ECMAScript

确定选项

可以确定向 Regex 对象提供哪些选项，在通过检索只读 Regex.Options 属性的值将其实例化时。该属性尤其可用于确定为编译的正则表达式定义的选项，该正则表达式由 Regex.CompileToAssembly 方法创建。

要测试除 RegexOptions.None 之外的任何选项的存在，使用 Regex.Options 属性的值和需要的 RegexOptions 值执行 AND 运算。然后测试结果是否等于该 RegexOptions 值。下面的示例测试是否设置了 RegexOptions.IgnoreCase 选项。

```
if ((rgx.Options & RegexOptions.IgnoreCase) == RegexOptions.IgnoreCase)
   Console.WriteLine("Case-insensitive pattern comparison.");
else
   Console.WriteLine("Case-sensitive pattern comparison.");
```

```
If (rgx.Options And RegexOptions.IgnoreCase) = RegexOptions.IgnoreCase Then
    Console.WriteLine("Case-insensitive pattern comparison.")
Else
    Console.WriteLine("Case-sensitive pattern comparison.")
End If
```

要测试 RegexOptions.None，确定 Regex.Options 属性的值是否等于 RegexOptions.None，如以下示例所示。

```
if (rgx.Options == RegexOptions.None)
   Console.WriteLine("No options have been set.");
```

```
If rgx.Options = RegexOptions.None Then
    Console.WriteLine("No options have been set.")
End If
```

下面各部分列出了 .NET 正则表达式支持的选项。

默认选项

RegexOptions.None 选项指示尚未指定任何选项，正则表达式引擎使用其默认行为。这包括：

- 该模式将被解释为一个规范而非 ECMAScript 正则表达式。
- 从左到右在输入字符串中匹配的正则表达式模式。
- 比较区分大小写。
- `^` 和 `$` 语言元素与输入字符串的开头和结尾匹配。
- `.` 语言元素与除 `\n` 之外的每个字符匹配。
- 正则表达式模式中的任意空白均解释为文本空白字符。
- 将模式与输入字符串进行比较时将使用当前区域性的约定。
- 正则表达式模式中的捕获组可以是隐式的，也可以是显式的。

> **NOTE**
>
> RegexOptions.None 选项没有内联等效项。当内联应用正则表达式选项时，默认行为通过关闭特定选项以逐个选项方式存储。例如，`(?i)` 打开不区分大小写的比较，`(?-i)` 还原默认区分大小写的比较。

因为 RegexOptions.None 选项表示正则表达式引擎的默认行为，因此它很少显式地在方法调用中指定。而改为调用构造函数或静态模式匹配的方法，其中不包含 `options` 参数。

不区分大小写的匹配

IgnoreCase 选项或 `i` 内联选项提供了不区分大小写匹配。默认情况下，使用当前区域性的大小写约定。

下面的示例定义与以"the"开头的所有单词匹配的正则表达式模式 `\bthe\w*\b`。因为对 Match 方法的第一次调用使用默认区分大小写的比较，因此输出会指示以字符串"The"开头的句子不匹配。通过将选项设置为 Match，调用 IgnoreCase 方法时对其进行匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bthe\w*\b";
      string input = "The man then told them about that event.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern,
                                            RegexOptions.IgnoreCase))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\bthe\w*\b"
        Dim input As String = "The man then told them about that event."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
        Next
        Console.WriteLine()
        For Each match As Match In Regex.Matches(input, pattern, _
                                                 RegexOptions.IgnoreCase)
            Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Found then at index 8.
'       Found them at index 18.
'
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
```

下面的示例修改了上一示例中的正则表达式模式，以使用内联选项而不是 `options` 参数来提供不区分大小写的

比较。第一个模式定义只应用于字符串"the"中的字母"t"的分组构造中的不区分大小写的选项。因为选项构造在模式的开始处出现，所以第二个模式将不区分大小写的选项应用于整个正则表达式。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?i:t)he\w*\b";
      string input = "The man then told them about that event.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);

      Console.WriteLine();
      pattern = @"(?i)\bthe\w*\b";
      foreach (Match match in Regex.Matches(input, pattern,
                                            RegexOptions.IgnoreCase))
         Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
//
//       Found The at index 0.
//       Found then at index 8.
//       Found them at index 18.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(?i:t)he\w*\b"
        Dim input As String = "The man then told them about that event."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
        Next
        Console.WriteLine()
        pattern = "(?i)\bthe\w*\b"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Found {0} at index {1}.", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
'
'       Found The at index 0.
'       Found then at index 8.
'       Found them at index 18.
```

多行模式

RegexOptions.Multiline 选项或 `m` 内联选项使正则表达式引擎能够处理由多个行组成的输入字符串。它更改了 `^` 和 `$` 语言元素的解释，以使它们分别与行的开头和结尾匹配，而不是与输入字符串的开头和结尾匹配。

默认情况下，`$` 仅与输入字符串的末尾匹配。如果指定了 RegexOptions.Multiline 选项，它将与换行符 (`\n`) 或

输入字符串的末尾匹配。但是，它并不与回车符/换行符的组合匹配。若要成功匹配它们，使用子表达式 `\r?$` 只替代 `$`。

下面的示例提取投手的姓名和分数，并将它们添加到 SortedList<TKey,TValue> 集合中，该集合将按降序顺序对它们进行排序。调用了两次 Matches 方法。在第一个方法调用中，正则表达式是 `^(\w+)\s(\d+)$`，且没有设置任何选项。如输出所示，因为正则表达式引擎与输入模式及输入字符串的开头和结尾均不匹配，因此没有找到匹配。在第二个方法调用中，正则表达式更改为 `^(\w+)\s(\d+)\r?$`，选项设置为 RegexOptions.Multiline。如输出所示，姓名和分数成功匹配，且分数按降序顺序显示。

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SortedList<int, string> scores = new SortedList<int, string>(new DescendingComparer<int>());

      string input = "Joe 164\n" +
                     "Sam 208\n" +
                     "Allison 211\n" +
                     "Gwen 171\n";
      string pattern = @"^(\w+)\s(\d+)$";
      bool matched = false;

      Console.WriteLine("Without Multiline option:");
      foreach (Match match in Regex.Matches(input, pattern))
      {
         scores.Add(Int32.Parse(match.Groups[2].Value), (string) match.Groups[1].Value);
         matched = true;
      }
      if (! matched)
         Console.WriteLine("   No matches.");
      Console.WriteLine();

      // Redefine pattern to handle multiple lines.
      pattern = @"^(\w+)\s(\d+)\r*$";
      Console.WriteLine("With multiline option:");
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
         scores.Add(Int32.Parse(match.Groups[2].Value), (string) match.Groups[1].Value);

      // List scores in descending order.
      foreach (KeyValuePair<int, string> score in scores)
         Console.WriteLine("{0}: {1}", score.Value, score.Key);
   }
}

public class DescendingComparer<T> : IComparer<T>
{
   public int Compare(T x, T y)
   {
      return Comparer<T>.Default.Compare(x, y) * -1;
   }
}
// The example displays the following output:
//   Without Multiline option:
//      No matches.
//
//   With multiline option:
//   Allison: 211
//   Sam: 208
//   Gwen: 171
//   Joe: 164
```

```
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim scores As New SortedList(Of Integer, String)(New DescendingComparer(Of Integer)())

        Dim input As String = "Joe 164" + vbCrLf + _
                              "Sam 208" + vbCrLf + _
                              "Allison 211" + vbCrLf + _
                              "Gwen 171" + vbCrLf
        Dim pattern As String = "^(\w+)\s(\d+)$"
        Dim matched As Boolean = False

        Console.WriteLine("Without Multiline option:")
        For Each match As Match In Regex.Matches(input, pattern)
            scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
            matched = True
        Next
        If Not matched Then Console.WriteLine("   No matches.")
        Console.WriteLine()

        ' Redefine pattern to handle multiple lines.
        pattern = "^(\w+)\s(\d+)\r*$"
        Console.WriteLine("With multiline option:")
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
            scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
        Next
        ' List scores in descending order.
        For Each score As KeyValuePair(Of Integer, String) In scores
            Console.WriteLine("{0}: {1}", score.Value, score.Key)
        Next
    End Sub
End Module

Public Class DescendingComparer(Of T) : Implements IComparer(Of T)
    Public Function Compare(x As T, y As T) As Integer _
           Implements IComparer(Of T).Compare
        Return Comparer(Of T).Default.Compare(x, y) * -1
    End Function
End Class
' The example displays the following output:
'    Without Multiline option:
'       No matches.
'
'    With multiline option:
'    Allison: 211
'    Sam: 208
'    Gwen: 171
'    Joe: 164
```

正则表达式模式 `^(\w+)\s(\d+)\r*$` 的定义如下表所示。

“	“
`^`	从行首开始。
`(\w+)`	匹配一个或多个单词字符。这是第一个捕获组。
`\s`	与空白字符匹配。
`(\d+)`	匹配一个或多个十进制数字。这是第二个捕获组。

𝙸𝙸	𝙸𝙸
`\r?`	与零个或一个回车符匹配。
`$`	在行尾结束。

下面的示例与上一示例等效，不同之处是下面的示例使用内联选项 `(?m)` 来设置多行选项。

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SortedList<int, string> scores = new SortedList<int, string>(new DescendingComparer<int>());

      string input = "Joe 164\n" +
                     "Sam 208\n" +
                     "Allison 211\n" +
                     "Gwen 171\n";
      string pattern = @"(?m)^(\w+)\s(\d+)\r*$";

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
         scores.Add(Convert.ToInt32(match.Groups[2].Value), match.Groups[1].Value);

      // List scores in descending order.
      foreach (KeyValuePair<int, string> score in scores)
         Console.WriteLine("{0}: {1}", score.Value, score.Key);
   }
}

public class DescendingComparer<T> : IComparer<T>
{
   public int Compare(T x, T y)
   {
      return Comparer<T>.Default.Compare(x, y) * -1;
   }
}
// The example displays the following output:
//    Allison: 211
//    Sam: 208
//    Gwen: 171
//    Joe: 164
```

```
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim scores As New SortedList(Of Integer, String)(New DescendingComparer(Of Integer)())

        Dim input As String = "Joe 164" + vbCrLf + _
                              "Sam 208" + vbCrLf + _
                              "Allison 211" + vbCrLf + _
                              "Gwen 171" + vbCrLf
        Dim pattern As String = "(?m)^(\w+)\s(\d+)\r*$"

        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
            scores.Add(CInt(match.Groups(2).Value), match.Groups(1).Value)
        Next
        ' List scores in descending order.
        For Each score As KeyValuePair(Of Integer, String) In scores
            Console.WriteLine("{0}: {1}", score.Value, score.Key)
        Next
    End Sub
End Module

Public Class DescendingComparer(Of T) : Implements IComparer(Of T)
    Public Function Compare(x As T, y As T) As Integer _
           Implements IComparer(Of T).Compare
        Return Comparer(Of T).Default.Compare(x, y) * -1
    End Function
End Class
' The example displays the following output:
'    Allison: 211
'    Sam: 208
'    Gwen: 171
'    Joe: 164
```

单行模式

RegexOptions.Singleline 选项或 `s` 内联选项导致正则表达式引擎将输入字符串视为由单行组成。它通过更改句号 (`.`) 语言元素的行为，使其与每个字符匹配，而不是与除换行符 `\n` 或 \u000A 之外的每个字符匹配来执行此操作。

下面的示例演示了在使用 `.` 选项时如何更改 RegexOptions.Singleline 语言元素的行为。正则表达式 `^.+` 在字符串开头开始并匹配每个字符。默认情况下，匹配在第一行的结尾结束；正则表达式模式匹配回车符、`\r` 或 \u000D，但不匹配 `\n`。由于 RegexOptions.Singleline 选项将整个输入字符串解释为单行，因此它匹配输入字符串中的每个字符，包括 `\n`。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));

      Console.WriteLine();
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Singleline))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r
//
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^.+"
        Dim input As String = "This is one line and" + vbCrLf + "this is the second."
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(Regex.Escape(match.Value))
        Next
        Console.WriteLine()
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.SingleLine)
            Console.WriteLine(Regex.Escape(match.Value))
        Next
    End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r
'
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

下面的示例与上一示例等效，不同之处是下面的示例使用内联选项 `(?s)` 来启用单行模式。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "(?s)^.+";
      string input = "This is one line and" + Environment.NewLine + "this is the second.";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(Regex.Escape(match.Value));
   }
}
// The example displays the following output:
//       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "(?s)^.+"
        Dim input As String = "This is one line and" + vbCrLf + "this is the second."

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(Regex.Escape(match.Value))
        Next
    End Sub
End Module
' The example displays the following output:
'       This\ is\ one\ line\ and\r\nthis\ is\ the\ second\.
```

仅显式捕获

默认情况下，通过在正则表达式模式中使用括号来定义捕获组。通过 `(?<` name `>` subexpression `)` 语言选项为命名组指定名称或编号，而未命名组按索引进行访问。在 GroupCollection 对象中，未命名的组先于已命名的组。

分组构造通常仅用于将限定符应用于多个语言元素，而非应用于捕获的子字符串。例如，如果下面的正则表达式：

`\b\(?((\w+),?\s?)+[\.!?]\)?`

旨在仅从文档提取末尾有句号、感叹点或问号的句子，仅产生的句子（这由 Match 对象表示）有意义。集合中的各单词不是。

随后未使用的捕获组可能很昂贵，因为正则表达式引擎必须填充 GroupCollection 和 CaptureCollection 集合对象。作为替换方法，也可以使用 RegexOptions.ExplicitCapture 选项或 `n` 内联选项，指定显式命名的唯一有效捕获，或由 `(?<` 名称 `>` 子表达式 `)` 构造指定的编号组 。

以下示例显示 `\b\(?((\w+),?\s?)+[\.!?]\)?` 正则表达式模式在 Match 方法被调用且没有 RegexOptions.ExplicitCapture 选项时返回的匹配信息。如第一个方法调用输出所示，正则表达式引擎使用有关已捕获的子字符串的信息完全填充 GroupCollection 和 CaptureCollection 集合对象。因为第二个方法使用设置为 `options` 的 RegexOptions.ExplicitCapture 进行调用，所以它不会捕获有关组的信息。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b\(?((?>\w+),?\s?)+[\.!?]\)?";
      Console.WriteLine("With implicit captures:");
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
```

```
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
      Console.WriteLine();
      Console.WriteLine("With explicit captures only:");
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.ExplicitCapture))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//    With implicit captures:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//       Group 1: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//       Group 2: sentence
//          Capture 0: This
//          Capture 1: is
//          Capture 2: the
//          Capture 3: first
//          Capture 4: sentence
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//       Group 1: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
//          Capture 3: beginning
//          Capture 4: of
//          Capture 5: a
//          Capture 6: literary
//          Capture 7: masterpiece
//       Group 2: masterpiece
//          Capture 0: Is
//          Capture 1: it
//          Capture 2: the
//          Capture 3: beginning
//          Capture 4: of
//          Capture 5: a
//          Capture 6: literary
//          Capture 7: masterpiece
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//       Group 1: not
//          Capture 0: I
//          Capture 1: think
//          Capture 2: not
```

```
//          Capture 2: not
//       Group 2: not
//          Capture 0: I
//          Capture 1: think
//          Capture 2: not
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
//       Group 1: paragraph
//          Capture 0: Instead,
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//       Group 2: paragraph
//          Capture 0: Instead
//          Capture 1: it
//          Capture 2: is
//          Capture 3: a
//          Capture 4: nonsensical
//          Capture 5: paragraph
//
//    With explicit captures only:
//    The match: This is the first sentence.
//       Group 0: This is the first sentence.
//          Capture 0: This is the first sentence.
//    The match: Is it the beginning of a literary masterpiece?
//       Group 0: Is it the beginning of a literary masterpiece?
//          Capture 0: Is it the beginning of a literary masterpiece?
//    The match: I think not.
//       Group 0: I think not.
//          Capture 0: I think not.
//    The match: Instead, it is a nonsensical paragraph.
//       Group 0: Instead, it is a nonsensical paragraph.
//          Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is the first sentence. Is it the beginning " + _
                              "of a literary masterpiece? I think not. Instead, " + _
                              "it is a nonsensical paragraph."
        Dim pattern As String = "\b\(?((?>\w+),?\s?)+[\.!?]\)?"
        Console.WriteLine("With implicit captures:")
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("The match: {0}", match.Value)
            Dim groupCtr As Integer = 0
            For Each group As Group In match.Groups
                Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
                groupCtr += 1
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
                    captureCtr += 1
                Next
            Next
        Next
        Console.WriteLine()
        Console.WriteLine("With explicit captures only:")
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.ExplicitCapture)
            Console.WriteLine("The match: {0}", match.Value)
            Dim groupCtr As Integer = 0
            For Each group As Group In match.Groups
                Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
                groupCtr += 1
                Dim captureCtr As Integer = 0
```

```
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
                    captureCtr += 1
                Next
            Next
        Next
    End Sub
End Module
' The example displays the following output:
'    With implicit captures:
'    The match: This is the first sentence.
'       Group 0: This is the first sentence.
'          Capture 0: This is the first sentence.
'       Group 1: sentence
'          Capture 0: This
'          Capture 1: is
'          Capture 2: the
'          Capture 3: first
'          Capture 4: sentence
'       Group 2: sentence
'          Capture 0: This
'          Capture 1: is
'          Capture 2: the
'          Capture 3: first
'          Capture 4: sentence
'    The match: Is it the beginning of a literary masterpiece?
'       Group 0: Is it the beginning of a literary masterpiece?
'          Capture 0: Is it the beginning of a literary masterpiece?
'       Group 1: masterpiece
'          Capture 0: Is
'          Capture 1: it
'          Capture 2: the
'          Capture 3: beginning
'          Capture 4: of
'          Capture 5: a
'          Capture 6: literary
'          Capture 7: masterpiece
'       Group 2: masterpiece
'          Capture 0: Is
'          Capture 1: it
'          Capture 2: the
'          Capture 3: beginning
'          Capture 4: of
'          Capture 5: a
'          Capture 6: literary
'          Capture 7: masterpiece
'    The match: I think not.
'       Group 0: I think not.
'          Capture 0: I think not.
'       Group 1: not
'          Capture 0: I
'          Capture 1: think
'          Capture 2: not
'       Group 2: not
'          Capture 0: I
'          Capture 1: think
'          Capture 2: not
'    The match: Instead, it is a nonsensical paragraph.
'       Group 0: Instead, it is a nonsensical paragraph.
'          Capture 0: Instead, it is a nonsensical paragraph.
'       Group 1: paragraph
'          Capture 0: Instead,
'          Capture 1: it
'          Capture 2: is
'          Capture 3: a
'          Capture 4: nonsensical
'          Capture 5: paragraph
'       Group 2: paragraph
```

```
'          Capture 0: Instead
'          Capture 1: it
'          Capture 2: is
'          Capture 3: a
'          Capture 4: nonsensical
'          Capture 5: paragraph
'
'    With explicit captures only:
'    The match: This is the first sentence.
'       Group 0: This is the first sentence.
'          Capture 0: This is the first sentence.
'    The match: Is it the beginning of a literary masterpiece?
'       Group 0: Is it the beginning of a literary masterpiece?
'          Capture 0: Is it the beginning of a literary masterpiece?
'    The match: I think not.
'       Group 0: I think not.
'          Capture 0: I think not.
'    The match: Instead, it is a nonsensical paragraph.
'       Group 0: Instead, it is a nonsensical paragraph.
'          Capture 0: Instead, it is a nonsensical paragraph.
```

正则表达式模式 `\b\(?((?>\w+),?\s?)+[\.!?]\)?` 的定义如下表所示。

“	“
`\b`	在单词边界处开始。
`\(?`	匹配左括号（“(”）的零或一个匹配项。
`(?>\w+),?`	匹配一个或多个单词字符，后跟零或一个逗号。当匹配单词字符请不要回溯。
`\s?`	匹配零个或一个空白字符。
`((\w+),?\s?)+`	一次或多次匹配一个或多个单词字符、零或一个逗号以及零或一个空白字符的组合。
`[\.!?]\)?`	与后无右括号或后跟一个右括号（“)”）的三个标点符号匹配。

还可以使用 `(?n)` 内联元素来禁止自动捕获。以下示例修改了上一示例中的正则表达式模式，使用的是内联元素 `(?n)` 而非 RegexOptions.ExplicitCapture 选项。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"(?n)\b\(?((?>\w+),?\s?)+[\.!?]\)?";

      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is the first sentence. Is it the beginning " + _
                              "of a literary masterpiece? I think not. Instead, " + _
                              "it is a nonsensical paragraph."
        Dim pattern As String = "(?n)\b\(?((?>\w+),?\s?)+[\.!?]\)?"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("The match: {0}", match.Value)
            Dim groupCtr As Integer = 0
            For Each group As Group In match.Groups
                Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
                groupCtr += 1
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
                    captureCtr += 1
                Next
            Next
        Next
    End Sub
End Module
' The example displays the following output:
'       The match: This is the first sentence.
'          Group 0: This is the first sentence.
'             Capture 0: This is the first sentence.
'       The match: Is it the beginning of a literary masterpiece?
'          Group 0: Is it the beginning of a literary masterpiece?
'             Capture 0: Is it the beginning of a literary masterpiece?
'       The match: I think not.
'          Group 0: I think not.
'             Capture 0: I think not.
'       The match: Instead, it is a nonsensical paragraph.
'          Group 0: Instead, it is a nonsensical paragraph.
'             Capture 0: Instead, it is a nonsensical paragraph.
```

最后，可以使用内联组元素 `(?n:)` 禁止逐组进行自动捕获。下面的示例修改了之前的模式，以取消外部组 `((?>\w+),?\s?)` 中的非命名捕获。请注意，这也取消了内部组中的非命名捕获。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b\(?(?n:(?>\w+),?\s?)+[\.!?]\)?";

      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine("The match: {0}", match.Value);
         int groupCtr = 0;
         foreach (Group group in match.Groups)
         {
            Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value);
            groupCtr++;
            int captureCtr = 0;
            foreach (Capture capture in group.Captures)
            {
               Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value);
               captureCtr++;
            }
         }
      }
   }
}
// The example displays the following output:
//       The match: This is the first sentence.
//          Group 0: This is the first sentence.
//             Capture 0: This is the first sentence.
//       The match: Is it the beginning of a literary masterpiece?
//          Group 0: Is it the beginning of a literary masterpiece?
//             Capture 0: Is it the beginning of a literary masterpiece?
//       The match: I think not.
//          Group 0: I think not.
//             Capture 0: I think not.
//       The match: Instead, it is a nonsensical paragraph.
//          Group 0: Instead, it is a nonsensical paragraph.
//             Capture 0: Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is the first sentence. Is it the beginning " + _
                              "of a literary masterpiece? I think not. Instead, " + _
                              "it is a nonsensical paragraph."
        Dim pattern As String = "\b\(?(?n:(?>\w+),?\s?)+[\.!?]\)?"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("The match: {0}", match.Value)
            Dim groupCtr As Integer = 0
            For Each group As Group In match.Groups
                Console.WriteLine("   Group {0}: {1}", groupCtr, group.Value)
                groupCtr += 1
                Dim captureCtr As Integer = 0
                For Each capture As Capture In group.Captures
                    Console.WriteLine("      Capture {0}: {1}", captureCtr, capture.Value)
                    captureCtr += 1
                Next
            Next
        Next
    End Sub
End Module
' The example displays the following output:
'       The match: This is the first sentence.
'          Group 0: This is the first sentence.
'             Capture 0: This is the first sentence.
'       The match: Is it the beginning of a literary masterpiece?
'          Group 0: Is it the beginning of a literary masterpiece?
'             Capture 0: Is it the beginning of a literary masterpiece?
'       The match: I think not.
'          Group 0: I think not.
'             Capture 0: I think not.
'       The match: Instead, it is a nonsensical paragraph.
'          Group 0: Instead, it is a nonsensical paragraph.
'             Capture 0: Instead, it is a nonsensical paragraph.
```

已编译的正则表达式

默认情况下，.NET 中的正则表达式会有解释。当实例化 Regex 对象或者调用静态 Regex 方法时，将把正则表达式模式解析为一组自定义操作代码，并且解释器使用这些操作代码来运行正则表达式。这涉及一个权衡：初始化正则表达式引擎的成本通过运行时性能的消耗而最小化。

通过使用 RegexOptions.Compiled 选项可以使用编译的而非解释的正则表达式。在此情况下，当模式传递给正则表达式引擎时，它将分析为一组操作码，然后转换为 Microsoft 中间语言 (MSIL)，该语言可以被直接传递到公共语言运行时。已编译的正则表达式最大限度地提高运行时性能，代价是会影响初始化时间。

> **NOTE**
> 仅可以通过将 RegexOptions.Compiled 值提供给 `options` 类构造函数或静态模式匹配方法的 Regex 参数来编译正则表达式。它不可作为内联选项使用。

在调用静态和实例正则表达式时，可使用编译的正则表达式。在静态正则表达式中，RegexOptions.Compiled 选项将传递到正则表达式模式匹配方法的 `options` 参数。在实例正则表达式中，将它传递到 `options` 类构造函数的 Regex 参数。在这两种情况中它将导致性能增强。

但是，这种性能改进只有在以下情况下才发生：

- 表示特定正则表达式的 Regex 对象可用于多个正则表达式模式匹配方法调用。

- 不允许 Regex 对象超出范围，以便可以重用它。
- 静态正则表达式在对正则表达式模式匹配方法的多个调用中使用。（之所以能够提高性能，是因为静态方法调用中使用的正则表达式由正则表达式引擎缓存。）

> **NOTE**
> RegexOptions.Compiled 选项与 Regex.CompileToAssembly 方法无关，该方法创建一个特殊用途的程序集，其中包含预定义的已编译的正则表达式。

忽略空白

默认情况下，正则表达式模式中的空白非常重要；它会强制正则表达式引擎与输入字符串中的空白字符相匹配。因此，正则表达式"`\b\w+\s`"和"`\b\w+`"是大致等效的正则表达式。此外，正则表达式模式中出现数字符号 (#) 时，它被解释为要进行匹配的原义字符。

RegexOptions.IgnorePatternWhitespace 选项或 `x` 内联选项更改此默认行为，如下所示：

- 正则表达式模式中的非转义的空白将被忽略。作为正则表达式模式的部分，必须避开空白字符（例如 `\s` 或"`\ `"）。
- 数字符号 (#) 被解释为注释的开头，而不是原义字符。正则表达式模式中的所有文本，从 # 字符到字符串的结尾都解释为注释。

但是，在下列情况下，不会忽略正则表达式中的空白字符，即使使用 RegexOptions.IgnorePatternWhitespace 选项也是如此：

- 始终按原义解释字符内的空格。例如，正则表达式模式 `[ .,;:]` 匹配任意单个空白字符、句号、逗号、分号或冒号。
- 加括号的限定符内不允许有空格，如 `{n}`、`{n,}` 和 `{n,m}`。例如，因为它包含一个空白字符，所以正则表达式模式 `\d{1, 3}` 与任何从 1 到 3 位数的数字序列不匹配。
- 引入语言元素的字符序列内不允许有空格。例如：
  - 语言元素 `(?:`subexpression`)` 表示非捕获组，并且该元素的 `(?:` 部分不能有嵌入空格。模式 `(? :`子表达式`)` 在运行时抛出 ArgumentException，因为正则表达式引擎无法分析此模式，且模式 `( ?:`子表达式`)` 与子表达式不匹配。
  - 语言元素 `\p{`name`}` 表示一个 Unicode 类别或命名块，它不能在元素的 `\p{` 部分中包括嵌入空格。如果你包括了空格，则该元素会在运行时引发 ArgumentException 异常。

启用此选项有助于简化通常很难分析和理解的正则表达式。它提高了可读性，并可以记录正则表达式。

下面的示例定义以下正则表达式模式：

```
\b \(? ( (?>\w+) ,?\s? )+ [\.!?] \)? # Matches an entire sentence.
```

此模式与仅显式捕获部分中定义的模式相似，不同之处在于它使用 RegexOptions.IgnorePatternWhitespace 选项忽略模式空格。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"\b \(? ( (?>\w+) ,?\s? )+ [\.!?] \)? # Matches an entire sentence.";

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is the first sentence. Is it the beginning " + _
                              "of a literary masterpiece? I think not. Instead, " + _
                              "it is a nonsensical paragraph."
        Dim pattern As String = "\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence."

        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnorePatternWhitespace)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       This is the first sentence.
'       Is it the beginning of a literary masterpiece?
'       I think not.
'       Instead, it is a nonsensical paragraph.
```

下面的示例使用内联选项 `(?x)` 来忽略模式空白。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is the first sentence. Is it the beginning " +
                     "of a literary masterpiece? I think not. Instead, " +
                     "it is a nonsensical paragraph.";
      string pattern = @"(?x)\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence.";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This is the first sentence.
//       Is it the beginning of a literary masterpiece?
//       I think not.
//       Instead, it is a nonsensical paragraph.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is the first sentence. Is it the beginning " + _
                              "of a literary masterpiece? I think not. Instead, " + _
                              "it is a nonsensical paragraph."
        Dim pattern As String = "(?x)\b \(? ( (?>\w+) ,?\s? )+  [\.!?] \)? # Matches an entire sentence."

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       This is the first sentence.
'       Is it the beginning of a literary masterpiece?
'       I think not.
'       Instead, it is a nonsensical paragraph.
```

从右到左模式

默认情况下，正则表达式引擎从左向右进行搜索。可通过使用 RegexOptions.RightToLeft 选项反转搜索方向。搜索在字符串的最后一个字符位置自动开始。对于包括起始位置参数的模式匹配方法，例如 Regex.Match(String, Int32)，起始位置是最右边字符位置（即搜索开始位置）的索引。

> **NOTE**
> 仅能通过将 RegexOptions.RightToLeft 值提供给 `options` 类构造函数或静态模式匹配方法的 Regex 参数来提供从右到左模式。它不可作为内联选项使用。

RegexOptions.RightToLeft 选项仅更改搜索方向；它不解释正则表达式模式是从右到左。例如，正则表达式 `\bb\w+\s` 匹配以字母“b”开头的单词，且后跟一个空白字符。在下面的示例中，输入字符串由其中包括一个或多个“b”字符的三个单词组成。第一个单词以“b”开头，第二个单词以“b”结尾，第三个单词的中间包括两个“b”字符。如示例输出所示，只有第一个词与正则表达式模式匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\bb\w+\s";
      string input = "builder rob rabble";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.RightToLeft))
         Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index);
   }
}
// The example displays the following output:
//       'builder ' found at position 0.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\bb\w+\s"
        Dim input As String = "builder rob rabble"
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.RightToLeft)
            Console.WriteLine("'{0}' found at position {1}.", match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       'builder ' found at position 0.
```

另请注意，预测先行断言（`(?=` subexpression `)` 语言元素）和回顾后发断言（`(?<=` subexpression `)` 语言元素）不会更改方向。预测先行断言向右搜索；回顾后发断言向左搜索。例如，正则表达式 `(?<=\d{1,2}\s)\w+,?\s\d{4}` 使用回顾后发断言测试月份名称前面的日期。然后该正则表达式匹配月份和年份。有关预测先行和回顾后发断言的信息，请参阅分组构造。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "1 May 1917", "June 16, 2003" };
      string pattern = @"(?<=\d{1,2}\s)\w+,?\s\d{4}";

      foreach (string input in inputs)
      {
         Match match = Regex.Match(input, pattern, RegexOptions.RightToLeft);
         if (match.Success)
            Console.WriteLine("The date occurs in {0}.", match.Value);
         else
            Console.WriteLine("{0} does not match.", input);
      }
   }
}
// The example displays the following output:
//       The date occurs in May 1917.
//       June 16, 2003 does not match.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"1 May 1917", "June 16, 2003"}
        Dim pattern As String = "(?<=\d{1,2}\s)\w+,?\s\d{4}"

        For Each input As String In inputs
            Dim match As Match = Regex.Match(input, pattern, RegexOptions.RightToLeft)
            If match.Success Then
                Console.WriteLine("The date occurs in {0}.", match.Value)
            Else
                Console.WriteLine("{0} does not match.", input)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       The date occurs in May 1917.
'       June 16, 2003 does not match.
```

正则表达式模式的定义如下表所示。

`(?<=\d{1,2}\s)`	匹配项的开头必须有后跟一个空格的一个或两个十进制数字。
`\w+`	匹配一个或多个单词字符。
`,?`	匹配零个或一个逗号字符。
`\s`	与空白字符匹配。
`\d{4}`	匹配四个十进制数字。

ECMAScript 匹配行为

默认情况下，当正则表达式模式与输入文本匹配时，正则表达式引擎会采用规范行为。但是，可以指示正则表达式引擎通过指定 RegexOptions.ECMAScript 选项使用 ECMAScript 匹配行为。

> **NOTE**
> 仅在通过将 RegexOptions.ECMAScript 值提供给 `options` 类构造函数造函数或静态模式匹配方法的 Regex 参数后，符合 ECMAScript 的行为才可用。它不可作为内联选项使用。

RegexOptions.ECMAScript 选项只能与 RegexOptions.IgnoreCase 和 RegexOptions.Multiline 选项结合使用。在正则表达式中使用其他选项会导致 ArgumentOutOfRangeException。

ECMAScript 和规范化正则表达式的行为在三个方面不同：字符类语法、自引用捕获组和八进制与反向引用的解释。

- 字符类语法。因为规范的正则表达式支持 Unicode，却不支持 ECMAScript，ECMAScript 中的字符类具有一个受限更多的语法且某些字符类语言元素具有不同的含义。例如，ECMAScript 不支持语言元素（例如 Unicode 类别或块元素 `\p` 和 `\P`）。同样，使用 ECMAScript 时，与单词字符匹配的 `\w` 元素等效于 `[a-zA-Z_0-9]` 字符类，使用规范化行为时，该元素等效于 `[\p{Ll}\p{Lu}\p{Lt}\p{Lo}\p{Nd}\p{Pc}\p{Lm}]`

。有关更多信息，请参见 字符类。

下面的示例阐释了规范化与 ECMAScript 模式匹配之间的差异。它定义了正则表达式 `\b(\w+\s*)+`，该表达式与后跟空白字符的单词匹配。由两个字符串组成的输入，其中一个字符串使用拉丁字符集，另一个则使用西里尔字符集。如输出所示，对使用 ECMAScript 匹配的 Regex.IsMatch(String, String, RegexOptions) 方法的调用无法与西里尔文的单词匹配，而使用规范化匹配的方法调用与这些单词匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] values = { "целый мир", "the whole world" };
      string pattern = @"\b(\w+\s*)+";
      foreach (var value in values)
      {
         Console.Write("Canonical matching: ");
         if (Regex.IsMatch(value, pattern))
            Console.WriteLine("'{0}' matches the pattern.", value);
         else
            Console.WriteLine("{0} does not match the pattern.", value);

         Console.Write("ECMAScript matching: ");
         if (Regex.IsMatch(value, pattern, RegexOptions.ECMAScript))
            Console.WriteLine("'{0}' matches the pattern.", value);
         else
            Console.WriteLine("{0} does not match the pattern.", value);
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Canonical matching: 'целый мир' matches the pattern.
//       ECMAScript matching: целый мир does not match the pattern.
//
//       Canonical matching: 'the whole world' matches the pattern.
//       ECMAScript matching: 'the whole world' matches the pattern.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim values() As String = {"целый мир", "the whole world"}
        Dim pattern As String = "\b(\w+\s*)+"
        For Each value In values
            Console.Write("Canonical matching: ")
            If Regex.IsMatch(value, pattern)
                Console.WriteLine("'{0}' matches the pattern.", value)
            Else
                Console.WriteLine("{0} does not match the pattern.", value)
            End If

            Console.Write("ECMAScript matching: ")
            If Regex.IsMatch(value, pattern, RegexOptions.ECMAScript)
                Console.WriteLine("'{0}' matches the pattern.", value)
            Else
                Console.WriteLine("{0} does not match the pattern.", value)
            End If
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       Canonical matching: 'целый мир' matches the pattern.
'       ECMAScript matching: целый мир does not match the pattern.
'
'       Canonical matching: 'the whole world' matches the pattern.
'       ECMAScript matching: 'the whole world' matches the pattern.
```

- 自引用捕获组。自身具有后向引用的正则表达式捕获类必须在每次捕获迭代时得到更新。如以下示例所示，此功能将在使用 ECMAScript 时使正则表达式 `((a+)(\1) ?)+` 与输入字符串“aa aaaa aaaaaa”匹配，但在使用规范化匹配时则不会匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   static string pattern;

   public static void Main()
   {
      string input = "aa aaaa aaaaaa ";
      pattern = @"((a+)(\1) ?)+";

      // Match input using canonical matching.
      AnalyzeMatch(Regex.Match(input, pattern));

      // Match input using ECMAScript.
      AnalyzeMatch(Regex.Match(input, pattern, RegexOptions.ECMAScript));
   }

   private static void AnalyzeMatch(Match m)
   {
      if (m.Success)
      {
         Console.WriteLine("'{0}' matches {1} at position {2}.",
                           pattern, m.Value, m.Index);
         int grpCtr = 0;
         foreach (Group grp in m.Groups)
         {
            Console.WriteLine("   {0}: '{1}'", grpCtr, grp.Value);
            grpCtr++;
            int capCtr = 0;
            foreach (Capture cap in grp.Captures)
            {
               Console.WriteLine("      {0}: '{1}'", capCtr, cap.Value);
               capCtr++;
            }
         }
      }
      else
      {
         Console.WriteLine("No match found.");
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//    No match found.
//
//    '((a+)(\1) ?)+' matches aa aaaa aaaaaa  at position 0.
//       0: 'aa aaaa aaaaaa '
//          0: 'aa aaaa aaaaaa '
//       1: 'aaaaaa '
//          0: 'aa '
//          1: 'aaaa '
//          2: 'aaaaaa '
//       2: 'aa'
//          0: 'aa'
//          1: 'aa'
//          2: 'aa'
//       3: 'aaaa '
//          0: ''
//          1: 'aa '
//          2: 'aaaa '
```

```
Imports System.Text.RegularExpressions

Module Example
    Dim pattern As String

    Public Sub Main()
        Dim input As String = "aa aaaa aaaaaa "
        pattern = "((a+)(\1) ?)+"

        ' Match input using canonical matching.
        AnalyzeMatch(Regex.Match(input, pattern))

        ' Match input using ECMAScript.
        AnalyzeMatch(Regex.Match(input, pattern, RegexOptions.ECMAScript))
    End Sub

    Private Sub AnalyzeMatch(m As Match)
        If m.Success
            Console.WriteLine("'{0}' matches {1} at position {2}.", _
                              pattern, m.Value, m.Index)
            Dim grpCtr As Integer = 0
            For Each grp As Group In m.Groups
                Console.WriteLine("   {0}: '{1}'", grpCtr, grp.Value)
                grpCtr += 1
                Dim capCtr As Integer = 0
                For Each cap As Capture In grp.Captures
                    Console.WriteLine("      {0}: '{1}'", capCtr, cap.Value)
                    capCtr += 1
                Next
            Next
        Else
            Console.WriteLine("No match found.")
        End If
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'    No match found.
'
'    '((a+)(\1) ?)+' matches aa aaaa aaaaaa  at position 0.
'       0: 'aa aaaa aaaaaa '
'          0: 'aa aaaa aaaaaa '
'       1: 'aaaaaa '
'          0: 'aa '
'          1: 'aaaa '
'          2: 'aaaaaa '
'       2: 'aa'
'          0: 'aa'
'          1: 'aa'
'          2: 'aa'
'       3: 'aaaa '
'          0: ''
'          1: 'aa '
'          2: 'aaaa '
```

该正则表达式的定义如下表所示。

"	"
(a+)	与字母“a”匹配一次或多次。这是第二个捕获组。
(\1)	与第一个捕获组捕获的子字符串匹配。这是第三个捕获组。

模式	说明
?	匹配零个或一个空白字符。
((a+)(\1) ?)+	与某个模式匹配一次或多次，该模式有一个或多个“a”字符，后跟与第一个捕获组（后无空白字符或后跟一个空白字符）匹配的字符串。这是第一个捕获组。

- 八进制转义和反向引用间的多义性的解析。下表总结了规范化和 ECMAScript 正则表达式在八进制与后向引用解释中的区别。

模式	说明	ECMAScript 模式
`\0` 后跟 0 到 2 个八进制数字	解释为八进制。例如，`\044` 总是解释为八进制值并表示“$”。	行为相同。
`\` 后跟一个从 1 到 9 的数字，后面再没有任何其他十进制数字，	解释为反向引用。例如，`\9` 始终表示后向引用 9，即使第九捕获组不存在。如果捕获组不存在，则正则表达式分析器将引发 ArgumentException。	如果存在单个十进制数字捕获组，则后向引用该数字。否则将该值解释为文本。
`\` 后跟一个从 1 到 9 的数字，后跟其他十进制数字	将数字解释为十进制值。如果存在该捕获组，则将该表达式解释为后向引用。 否则，将前导的八进制数字解释为不超过八进制值 377 的八进制数字；也就是说，仅考虑该值的后八位。将其余数字解释为文本。例如，如果表达式 `\3000` 中存在捕获组 300，则解释为后向引用 300；如果捕获组 300 不存在，则解释为后跟 0 的八进制数字 300。	通过将尽可能多的数字转换为可引用捕获的十进制值解释为反向引用。如果任何数字都不能转换，则解释为使用其值不超过八进制值 377 的前导八进制数字的八进制数字；将其余数字解释为文本。

使用固定区域性的比较

默认情况下，当正则表达式引擎执行不区分大小写的比较时，它使用当前区域性的大小写约定来确定等效的大写和小写字符。

但是，此行为不需要某些类型的比较，尤其是在比较用户输入与系统资源名称时（如密码、文件或 URL）。下面的示例阐释此类方案。该代码旨在阻止对 URL 开头为 FILE:// 的所有资源的访问。正则表达式通过使用正则表达式 `$FILE://` 尝试与字符串的不区分大小写的匹配。但是，在当前系统区域性为 tr-TR（土耳其语-土耳其）时，“I”不是“i”的大写等效项。因此，对 Regex.IsMatch 方法的调用返回 `false`，并允许访问该文件。

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-sensitive matching ({0} culture)...",
                  Thread.CurrentThread.CurrentCulture.Name);
if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
   Console.WriteLine("URLs that access files are not allowed.");
else
   Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-sensitive matching (tr-TR culture)...
//       Access to file://c:/Documents.MyReport.doc is allowed.
```

```
Dim defaultCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")

Dim input As String = "file://c:/Documents.MyReport.doc"
Dim pattern As String = "$FILE://"

Console.WriteLine("Culture-sensitive matching ({0} culture)...", _
                  Thread.CurrentThread.CurrentCulture.Name)
If Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase) Then
    Console.WriteLine("URLs that access files are not allowed.")
Else
    Console.WriteLine("Access to {0} is allowed.", input)
End If

Thread.CurrentThread.CurrentCulture = defaultCulture
' The example displays the following output:
'       Culture-sensitive matching (tr-TR culture)...
'       Access to file://c:/Documents.MyReport.doc is allowed.
```

NOTE

有关区分大小写和使用固定区域性的字符串比较的更多信息，请参见针对使用字符串的最佳做法。

不使用当前区域性的不区分大小写比较，可以指定 RegexOptions.CultureInvariant 选项忽略语言的区域性差异，并使用固定区域性的约定。

NOTE

仅能通过将 RegexOptions.CultureInvariant 值提供给 `options` 类构造函数或静态模式匹配方法的 Regex 参数来提供使用固定区域性的比较。它不可作为内联选项使用。

下面的示例与上一示例相等，不同之处是下面的示例使用包含 Regex.IsMatch(String, String, RegexOptions) 的选项调用静态 RegexOptions.CultureInvariant 方法。即使设置当前区域性到土耳其语（土耳其），正则表达式引擎仍能够成功匹配“FILE”和“file”并能阻止对文件资源的访问。

```
CultureInfo defaultCulture = Thread.CurrentThread.CurrentCulture;
Thread.CurrentThread.CurrentCulture = new CultureInfo("tr-TR");

string input = "file://c:/Documents.MyReport.doc";
string pattern = "FILE://";

Console.WriteLine("Culture-insensitive matching...");
if (Regex.IsMatch(input, pattern,
                  RegexOptions.IgnoreCase | RegexOptions.CultureInvariant))
   Console.WriteLine("URLs that access files are not allowed.");
else
   Console.WriteLine("Access to {0} is allowed.", input);

Thread.CurrentThread.CurrentCulture = defaultCulture;
// The example displays the following output:
//       Culture-insensitive matching...
//       URLs that access files are not allowed.
```

```
Dim defaultCulture As CultureInfo = Thread.CurrentThread.CurrentCulture
Thread.CurrentThread.CurrentCulture = New CultureInfo("tr-TR")

Dim input As String = "file://c:/Documents.MyReport.doc"
Dim pattern As String = "$FILE://"

Console.WriteLine("Culture-insensitive matching...")
If Regex.IsMatch(input, pattern, _
               RegexOptions.IgnoreCase Or RegexOptions.CultureInvariant) Then
    Console.WriteLine("URLs that access files are not allowed.")
Else
    Console.WriteLine("Access to {0} is allowed.", input)
End If
Thread.CurrentThread.CurrentCulture = defaultCulture
' The example displays the following output:
'        Culture-insensitive matching...
'        URLs that access files are not allowed.
```

请参阅

- 正则表达式语言 - 快速参考

正则表达式中的其他构造

2021/11/16 •

.NET 中的正则表达式包括三个其他语言构造。其中一个使你可以在正则表达式模式中间启用或禁用特定匹配选项。其余两个使你可以在正则表达式中包含注释。

内联选项

可以使用语法为正则表达式的一部分设置或禁用特定模式匹配选项

`(?imnsx-imnsx)`

在问号后列出要启用的选项，在负号后列出要禁用的选项。下表对每个选项进行了描述。有关每个选项的更多信息，请参见正则表达式选项。

“	“
`i`	不区分大小写的匹配。
`m`	多行模式。
`n`	仅显式捕获。（圆括号不充当捕获组。）
`s`	单行模式。
`x`	忽略未转义空格，并允许 x 模式注释。

如果 `(?imnsx-imnsx)` 构造定义的正则表达式选项有任何更改，更改在封闭组结束前一直有效。

> **NOTE**
> `(?imnsx-imnsx:` subexpression `)` 分组构造为子表达式提供了完全相同的功能。有关详细信息，请参阅 分组构造。

下面的示例使用 `i`、`n` 和 `x` 选项，启用不区分大小写和显式捕获，并在正则表达式中间忽略正则表达式模式中的空格。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern;
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      pattern = @"\b(D\w+)\s(d\w+)\b";
      // Match pattern using default options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
      }
      Console.WriteLine();

      // Change regular expression pattern to include options.
      pattern = @"\b(D\w+)(?ixn) \s (d\w+) \b";
      // Match new pattern with options.
      foreach (Match match in Regex.Matches(input, pattern))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
            for (int ctr = 1; ctr < match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups[ctr].Value);
      }
   }
}
// The example displays the following output:
//       Drooling dog
//          Group 1: Drooling
//          Group 2: dog
//
//       Drooling dog
//          Group 1: 'Drooling'
//       Dreaded Deep
//          Group 1: 'Dreaded'
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String
        Dim input As String = "double dare double Double a Drooling dog The Dreaded Deep"

        pattern = "\b(D\w+)\s(d\w+)\b"
        ' Match pattern using default options.
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
            If match.Groups.Count > 1 Then
                For ctr As Integer = 1 To match.Groups.Count - 1
                    Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
                Next
            End If
        Next
        Console.WriteLine()

        ' Change regular expression pattern to include options.
        pattern = "\b(D\w+)(?ixn) \s (d\w+) \b"
        ' Match new pattern with options.
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
            If match.Groups.Count > 1 Then
                For ctr As Integer = 1 To match.Groups.Count - 1
                    Console.WriteLine("   Group {0}: '{1}'", ctr, match.Groups(ctr).Value)
                Next
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       Drooling dog
'          Group 1: Drooling
'          Group 2: dog
'
'       Drooling dog
'          Group 1: 'Drooling'
'       Dreaded Deep
'          Group 1: 'Dreaded'
```

该示例定义两个正则表达式。第一个 `\b(D\w+)\s(d\w+)\b` 匹配以一个大写“D”和一个小写“d”开头的两个连续单词。第二个正则表达式 `\b(D\w+)(?ixn) \s (d\w+) \b` 使用内联选项修改此模式，如下表所述。结果的比较会确认 `(?ixn)` 构造的效果。

“	“
`\b`	在单词边界处开始。
`(D\w+)`	匹配后跟一个或多个单词字符的大写“D”。这是第一个捕获组。
`(?ixn)`	从此处起，使比较不区分大小写，仅进行显式捕获，以及忽略正则表达式模式中的空格。
`\s`	与空白字符匹配。
`(d\w+)`	匹配后跟一个或多个单词字符的大写或小写“d”。因为 `n`（显式捕获）选项已启用，所以不会捕获此组。

“	“
`\b`	与字边界匹配。

内联注释

`(?#` comment `)` 构造可用于在正则表达式中添加内联注释 。正则表达式引擎在模式匹配中不使用注释的任何部分，尽管注释仍包含在 Regex.ToString 方法返回的字符串中。该注释在第一个右括号处终止。

下面的示例重复了上一部分的示例中的第一个正则表达式模式。它将两个内联注释添加到该正则表达式，以指示比较是否区分大小写。正则表达式模式 `\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comparison)d\w+)\b` 按以下方式定义。

“	“
`\b`	在单词边界处开始。
`(?# case-sensitive comparison)`	注释。它不影响模式匹配行为。
`(D\w+)`	匹配后跟一个或多个单词字符的大写“D”。这是第一个捕获组。
`\s`	与空白字符匹配。
`(?ixn)`	从此处起，使比较不区分大小写，仅进行显式捕获，以及忽略正则表达式模式中的空格。
`(?#case-insensitive comparison)`	注释。它不影响模式匹配行为。
`(d\w+)`	匹配后跟一个或多个单词字符的大写或小写“d”。这是第二个捕获组。
`\b`	与字边界匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive
comparison)d\w+)\b";
      Regex rgx = new Regex(pattern);
      string input = "double dare double Double a Drooling dog The Dreaded Deep";

      Console.WriteLine("Pattern: " + pattern.ToString());
      // Match pattern using default options.
      foreach (Match match in rgx.Matches(input))
      {
         Console.WriteLine(match.Value);
         if (match.Groups.Count > 1)
         {
            for (int ctr = 1; ctr <match.Groups.Count; ctr++)
               Console.WriteLine("   Group {0}: {1}", ctr, match.Groups[ctr].Value);
         }
      }
   }
}
// The example displays the following output:
//    Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comp
//    arison)d\w+)\b
//    Drooling dog
//       Group 1: Drooling
//    Dreaded Deep
//       Group 1: Dreaded
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive
comparison)d\w+)\b"
        Dim rgx As New Regex(pattern)
        Dim input As String = "double dare double Double a Drooling dog The Dreaded Deep"

        Console.WriteLine("Pattern: " + pattern.ToString())
        ' Match pattern using default options.
        For Each match As Match In rgx.Matches(input)
            Console.WriteLine(match.Value)
            If match.Groups.Count > 1 Then
                For ctr As Integer = 1 To match.Groups.Count - 1
                    Console.WriteLine("   Group {0}: {1}", ctr, match.Groups(ctr).Value)
                Next
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'    Pattern: \b((?# case-sensitive comparison)D\w+)\s(?ixn)((?#case-insensitive comp
'    arison)d\w+)\b
'    Drooling dog
'       Group 1: Drooling
'    Dreaded Deep
'       Group 1: Dreaded
```

行尾注释

数字符号 (`#`) 标记 x 模式注释，即从正则表达式模式末尾的未转义 # 字符开始一直延续到行末。若要使用此构造，必须启用 `x` 选项(通过内联选项)，或在实例化 Regex 对象或调用静态 Regex 方法时向 `option` 参数提供 RegexOptions.IgnorePatternWhitespace 值。

下面的示例说明行尾注释构造。它确定字符串是否为包含至少一个格式项的复合格式字符串。下表描述了正则表达式模式中的构造：

```
\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item.
```

"	"
`\{`	匹配左大括号。
`\d+`	匹配一个或多个十进制数字。
`(,-*\d+)*`	与零个或一个后跟一个可选负号、再后跟一个或多个十进制数字的逗号匹配。
`(\:\w{1,4}?)*`	与零个或一个后跟一到四个(但尽可能少)空白字符的冒号匹配。
`\}`	匹配右大括号。
`(?x)`	启用忽略模式空格选项，以便识别行尾注释。
`# Looks for a composite format item.`	行尾注释。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item.";
      string input = "{0,-3:F}";
      Console.WriteLine("'{0}':", input);
      if (Regex.IsMatch(input, pattern))
         Console.WriteLine("   contains a composite format item.");
      else
         Console.WriteLine("   does not contain a composite format item.");
   }
}
// The example displays the following output:
//       '{0,-3:F}':
//          contains a composite format item.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\{\d+(,-*\d+)*(\:\w{1,4}?)*\}(?x) # Looks for a composite format item."
        Dim input As String = "{0,-3:F}"
        Console.WriteLine("'{0}':", input)
        If Regex.IsMatch(input, pattern) Then
            Console.WriteLine("   contains a composite format item.")
        Else
            Console.WriteLine("   does not contain a composite format item.")
        End If
    End Sub
End Module
' The example displays the following output:
'       '{0,-3:F}':
'          contains a composite format item.
```

请注意，还可以调用 Regex.IsMatch(String, String, RegexOptions) 方法并向它传递 RegexOptions.IgnorePatternWhitespace 枚举值，从而识别注释，而不用在正则表达式中提供 `(?x)` 构造。

请参阅

- 正则表达式语言 - 快速参考

.NET 中的正则表达式最佳做法

2021/11/16 •

.NET 中的正则表达式引擎是一种功能强大而齐全的工具，它基于模式匹配（而不是比较和匹配文本）来处理文本。在大多数情况下，它可以快速、高效地执行模式匹配。但在某些情况下，正则表达式引擎的速度似乎很慢。在极端情况下，它甚至看似停止响应，因为它会用若干个小时甚至若干天处理相对小的输入。

本主题概述开发人员为了确保其正则表达式实现最佳性能可以采纳的一些最佳做法。

> **WARNING**
> 如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

考虑输入源

通常，正则表达式可接受两种类型的输入：受约束的输入或不受约束的输入。受约束的输入是源自已知或可靠的源并遵循预定义格式的文本。不受约束的输入是源自不可靠的源（如 Web 用户）并且可能不遵循预定义或预期格式的文本。

编写的正则表达式模式的目的通常是匹配有效输入。也就是说，开发人员检查他们要匹配的文本，然后编写与其匹配的正则表达式模式。然后，开发人员使用多个有效输入项进行测试，以确定此模式是否需要更正或进一步细化。当模式可匹配所有假定的有效输入时，则将其声明为生产就绪并且可包括在发布的应用程序中。这使得正则表达式模式适合匹配受约束的输入。但它不适合匹配不受约束的输入。

若要匹配不受约束的输入，正则表达式必须能够高效处理以下三种文本：

- 与正则表达式模式匹配的文本。
- 与正则表达式模式不匹配的文本。
- 与正则表达式模式大致匹配的文本。

对于为了处理受约束的输入而编写的正则表达式，最后一种文本类型尤其存在问题。如果该正则表达式还依赖大量回溯，则正则表达式引擎可能会花费大量时间（在有些情况下，需要许多个小时或许多天）来处理看似无害的文本。

> **WARNING**
> 下面的示例使用容易过度回溯并可能拒绝有效电子邮件地址的正则表达式。不应在电子邮件验证例程中使用。如需可验证电子邮件地址的正则表达式，请参阅如何：确认字符串是有效的电子邮件格式。

例如，考虑一种很常用但很有问题的用于验证电子邮件地址别名的正则表达式。编写正则表达式 `^[0-9A-Z]([-.\w]*[0-9A-Z])*$` 的目的是处理被视为有效的电子邮件地址，该地址包含一个字母数字字符，后跟零个或多个可为字母数字、句点或连字符的字符。该正则表达式必须以字母数字字符结束。但正如下面的示例所示，尽管此正则表达式可以轻松处理有效输入，但在处理接近有效的输入时性能非常低效。

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;
```

```
public class Example
{
   public static void Main()
   {
      Stopwatch sw;
      string[] addresses = { "AAAAAAAAAA@contoso.com",
                             "AAAAAAAAAAaaaaaaaaaa!@contoso.com" };
      // The following regular expression should not actually be used to
      // validate an email address.
      string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])*$";
      string input;

      foreach (var address in addresses) {
         string mailBox = address.Substring(0, address.IndexOf("@"));
         int index = 0;
         for (int ctr = mailBox.Length - 1; ctr >= 0; ctr--) {
            index++;

            input = mailBox.Substring(ctr, index);
            sw = Stopwatch.StartNew();
            Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
            sw.Stop();
            if (m.Success)
               Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                 index, m.Value, sw.Elapsed);
            else
               Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                 index, input, sw.Elapsed);
         }
         Console.WriteLine();
      }
   }
}

// The example displays output similar to the following:
//     1. Matched '                        A' in 00:00:00.0007122
//     2. Matched '                       AA' in 00:00:00.0000282
//     3. Matched '                      AAA' in 00:00:00.0000042
//     4. Matched '                     AAAA' in 00:00:00.0000038
//     5. Matched '                    AAAAA' in 00:00:00.0000042
//     6. Matched '                   AAAAAA' in 00:00:00.0000042
//     7. Matched '                  AAAAAAA' in 00:00:00.0000042
//     8. Matched '                 AAAAAAAA' in 00:00:00.0000087
//     9. Matched '                AAAAAAAAA' in 00:00:00.0000045
//    10. Matched '               AAAAAAAAAA' in 00:00:00.0000045
//    11. Matched '              AAAAAAAAAAA' in 00:00:00.0000045
//
//     1. Failed  '                        !' in 00:00:00.0000447
//     2. Failed  '                       a!' in 00:00:00.0000071
//     3. Failed  '                      aa!' in 00:00:00.0000071
//     4. Failed  '                     aaa!' in 00:00:00.0000061
//     5. Failed  '                    aaaa!' in 00:00:00.0000081
//     6. Failed  '                   aaaaa!' in 00:00:00.0000126
//     7. Failed  '                  aaaaaa!' in 00:00:00.0000359
//     8. Failed  '                 aaaaaaa!' in 00:00:00.0000414
//     9. Failed  '                aaaaaaaa!' in 00:00:00.0000758
//    10. Failed  '               aaaaaaaaa!' in 00:00:00.0001462
//    11. Failed  '              aaaaaaaaaa!' in 00:00:00.0002885
//    12. Failed  '             Aaaaaaaaaaa!' in 00:00:00.0005780
//    13. Failed  '            AAaaaaaaaaaa!' in 00:00:00.0011628
//    14. Failed  '           AAAaaaaaaaaaa!' in 00:00:00.0022851
//    15. Failed  '          AAAAaaaaaaaaaa!' in 00:00:00.0045864
//    16. Failed  '         AAAAAaaaaaaaaaa!' in 00:00:00.0093168
//    17. Failed  '        AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
//    18. Failed  '       AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
//    19. Failed  '      AAAAAAAAaaaaaaaaaa!' in 00:00:00.1370108
//    20. Failed  '     AAAAAAAAAaaaaaaaaaa!' in 00:00:00.1553966
//    21. Failed  '    AAAAAAAAAAaaaaaaaaaa!' in 00:00:00.3223372
```

```
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim sw As Stopwatch
        Dim addresses() As String = {"AAAAAAAAAA@contoso.com",
                                     "AAAAAAAAAAaaaaaaaaaa!@contoso.com"}
        ' The following regular expression should not actually be used to
        ' validate an email address.
        Dim pattern As String = "^[0-9A-Z]([-.\w]*[0-9A-Z])*$"
        Dim input As String

        For Each address In addresses
            Dim mailBox As String = address.Substring(0, address.IndexOf("@"))
            Dim index As Integer = 0
            For ctr As Integer = mailBox.Length - 1 To 0 Step -1
                index += 1
                input = mailBox.Substring(ctr, index)
                sw = Stopwatch.StartNew()
                Dim m As Match = Regex.Match(input, pattern, RegexOptions.IgnoreCase)
                sw.Stop()
                if m.Success Then
                    Console.WriteLine("{0,2}. Matched '{1,25}' in {2}",
                                      index, m.Value, sw.Elapsed)
                Else
                    Console.WriteLine("{0,2}. Failed  '{1,25}' in {2}",
                                      index, input, sw.Elapsed)
                End If
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays output similar to the following:
'     1. Matched '                        A' in 00:00:00.0007122
'     2. Matched '                       AA' in 00:00:00.0000282
'     3. Matched '                      AAA' in 00:00:00.0000042
'     4. Matched '                     AAAA' in 00:00:00.0000038
'     5. Matched '                    AAAAA' in 00:00:00.0000042
'     6. Matched '                   AAAAAA' in 00:00:00.0000042
'     7. Matched '                  AAAAAAA' in 00:00:00.0000042
'     8. Matched '                 AAAAAAAA' in 00:00:00.0000087
'     9. Matched '                AAAAAAAAA' in 00:00:00.0000045
'    10. Matched '               AAAAAAAAAA' in 00:00:00.0000045
'    11. Matched '              AAAAAAAAAAA' in 00:00:00.0000045
'
'     1. Failed  '                        !' in 00:00:00.0000447
'     2. Failed  '                       a!' in 00:00:00.0000071
'     3. Failed  '                      aa!' in 00:00:00.0000071
'     4. Failed  '                     aaa!' in 00:00:00.0000061
'     5. Failed  '                    aaaa!' in 00:00:00.0000081
'     6. Failed  '                   aaaaa!' in 00:00:00.0000126
'     7. Failed  '                  aaaaaa!' in 00:00:00.0000359
'     8. Failed  '                 aaaaaaa!' in 00:00:00.0000414
'     9. Failed  '                aaaaaaaa!' in 00:00:00.0000758
'    10. Failed  '               aaaaaaaaa!' in 00:00:00.0001462
'    11. Failed  '              aaaaaaaaaa!' in 00:00:00.0002885
'    12. Failed  '             Aaaaaaaaaaa!' in 00:00:00.0005780
'    13. Failed  '            AAaaaaaaaaaa!' in 00:00:00.0011628
'    14. Failed  '           AAAaaaaaaaaaa!' in 00:00:00.0022851
'    15. Failed  '          AAAAaaaaaaaaaa!' in 00:00:00.0045864
'    16. Failed  '         AAAAAaaaaaaaaaa!' in 00:00:00.0093168
'    17. Failed  '        AAAAAAaaaaaaaaaa!' in 00:00:00.0185993
'    18. Failed  '       AAAAAAAaaaaaaaaaa!' in 00:00:00.0366723
'    19. Failed  '      AAAAAAAAaaaaaaaaaa!' in 00:00:00.1370108
'    20. Failed  '     AAAAAAAAAaaaaaaaaaa!' in 00:00:00.1553966
```

```
'    21. Failed  '    AAAAAAAAAAaaaaaaaaaa!' in 00:00:00.3223372
```

如该示例输出所示，正则表达式引擎处理有效电子邮件别名的时间间隔大致相同，与其长度无关。另一方面，当接近有效的电子邮件地址包含五个以上字符时，字符串中每增加一个字符，处理时间会大约增加一倍。这意味着，处理接近有效的 28 个字符构成的字符串将需要一个小时，处理接近有效的 33 个字符构成的字符串将需要接近一天的时间。

由于开发此正则表达式时只考虑了要匹配的输入的格式，因此未能考虑与模式不匹配的输入。这反过来会使与正则表达式模式近似匹配的不受约束输入的性能显著降低。

若要解决此问题，可执行下列操作：

- 开发模式时，应考虑回溯对正则表达式引擎的性能的影响程度，特别是当正则表达式设计用于处理不受约束的输入时。有关详细信息，请参阅控制回溯部分。
- 使用无效输入、接近有效的输入以及有效输入对正则表达式进行完全测试。若要为特定正则表达式随机生成输入，可以使用 Rex，这是 Microsoft Research 提供的正则表达式探索工具。

适当处理对象实例化

.NET 正则表达式对象模型的核心是 System.Text.RegularExpressions.Regex 类，表示正则表达式引擎。通常，影响正则表达式性能的单个最大因素是 Regex 引擎的使用方式。定义正则表达式需要将正则表达式引擎与正则表达式模式紧密耦合。无论该耦合过程是需要通过向其构造函数传递正则表达式模式来实例化 Regex 还是通过向其传递正则表达式模式和要分析的字符串来调用静态方法，都必然会消耗大量资源。

> **NOTE**
> 若要详细了解使用已解释和已编译正则表达式造成的性能影响，请参阅 BCL 团队博客中的 Optimizing Regular Expression Performance, Part II:Taking Charge of Backtracking（优化正则表达式性能，第 II 部分：控制回溯）。

可将正则表达式引擎与特定正则表达式模式耦合，然后使用该引擎以若干种方式匹配文本：

- 可以调用静态模式匹配方法，如 Regex.Match(String, String)。这不需要实例化正则表达式对象。
- 可以实例化一个 Regex 对象并调用已解释的正则表达式的实例模式匹配方法。这是将正则表达式引擎绑定到正则表达式模式的默认方法。如果实例化 Regex 对象时未使用包括 `options` 标记的 Compiled 自变量，则会生成此方法。
- 可以实例化一个 Regex 对象并调用已编译的正则表达式的实例模式匹配方法。当使用包括 Regex 标记的 `options` 参数实例化 Compiled 对象时，正则表达式对象表示已编译的模式。
- 可以创建一个与特定正则表达式模式紧密耦合的特殊用途的 Regex 对象，编译该对象，并将其保存到独立程序集中。为此，可调用 Regex.CompileToAssembly 方法。

这种调用正则表达式匹配方法的特殊方式会对应用程序产生显著影响。以下各节讨论何时使用静态方法调用、已解释的正则表达式和已编译的正则表达式，以改进应用程序的性能。

> **IMPORTANT**
> 如果方法调用中重复使用同一正则表达式或者应用程序大量使用正则表达式对象，则方法调用的形式（静态、已解释、已编译）会影响性能。

静态正则表达式

建议将静态正则表达式方法用作使用同一正则表达式重复实例化正则表达式对象的替代方法。与正则表达式对象使用的正则表达式模式不同，静态方法调用所使用的模式中的操作代码或已编译的 Microsoft 中间语言 (MSIL) 由正则表达式引擎缓存在内部。

例如，事件处理程序会频繁调用其他方法来验证用户输入。下面的代码中反映了这一点，其中一个 Button 控件的 Click 事件用于调用名为 `IsValidCurrency` 的方法，该方法检查用户是否输入了后跟至少一个十进制数的货币符号。

```
public void OKButton_Click(object sender, EventArgs e)
{
   if (! String.IsNullOrEmpty(sourceCurrency.Text))
      if (RegexLib.IsValidCurrency(sourceCurrency.Text))
         PerformConversion();
      else
         status.Text = "The source currency value is invalid.";
}
```

```
Public Sub OKButton_Click(sender As Object, e As EventArgs) _
           Handles OKButton.Click

    If Not String.IsNullOrEmpty(sourceCurrency.Text) Then
        If RegexLib.IsValidCurrency(sourceCurrency.Text) Then
            PerformConversion()
        Else
            status.Text = "The source currency value is invalid."
        End If
    End If
End Sub
```

下面的示例显示 `IsValidCurrency` 方法的一个非常低效的实现。请注意，每个方法调用使用相同模式重新实例化 Regex 对象。这反过来意味着，每次调用该方法时，都必须重新编译正则表达式模式。

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      Regex currencyRegex = new Regex(pattern);
      return currencyRegex.IsMatch(currencyValue);
   }
}
```

```
Imports System.Text.RegularExpressions

Public Module RegexLib
    Public Function IsValidCurrency(currencyValue As String) As Boolean
        Dim pattern As String = "\p{Sc}+\s*\d+"
        Dim currencyRegex As New Regex(pattern)
        Return currencyRegex.IsMatch(currencyValue)
    End Function
End Module
```

应将此低效代码替换为对静态 Regex.IsMatch(String, String) 方法的调用。这样便不必在你每次要调用模式匹配方法时都实例化 Regex 对象，还允许正则表达式引擎从其缓存中检索正则表达式的已编译版本。

```
using System;
using System.Text.RegularExpressions;

public class RegexLib
{
   public static bool IsValidCurrency(string currencyValue)
   {
      string pattern = @"\p{Sc}+\s*\d+";
      return Regex.IsMatch(currencyValue, pattern);
   }
}
```

```
Imports System.Text.RegularExpressions

Public Module RegexLib
    Public Function IsValidCurrency(currencyValue As String) As Boolean
        Dim pattern As String = "\p{Sc}+\s*\d+"
        Return Regex.IsMatch(currencyValue, pattern)
    End Function
End Module
```

默认情况下，将缓存最后 15 个最近使用的静态正则表达式模式。对于需要大量已缓存的静态正则表达式的应用程序，可通过设置 Regex.CacheSize 属性来调整缓存大小。

此示例中使用的正则表达式 `\p{Sc}+\s*\d+` 可验证输入字符串是否包含一个货币符号和至少一个十进制数。模式的定义如下表所示。

“	“
`\p{Sc}+`	与 Unicode 符号、货币类别中的一个或多个字符匹配。
`\s*`	匹配零个或多个空白字符。
`\d+`	匹配一个或多个十进制数字。

已解释与已编译的正则表达式

将解释未通过 Compiled 选项的规范绑定到正则表达式引擎的正则表达式模式。在实例化正则表达式对象时，正则表达式引擎会将正则表达式转换为一组操作代码。调用实例方法时，操作代码会转换为 MSIL 并由 JIT 编译器执行。同样，当调用一种静态正则表达式方法并且在缓存中找不到该正则表达式时，正则表达式引擎会将该正则表达式转换为一组操作代码并将其存储在缓存中。然后，它将这些操作代码转换为 MSIL，以便于 JIT 编译器执行。已解释的正则表达式会减少启动时间，但会使执行速度变慢。因此，在少数方法调用中使用正则表达式时或调用正则表达式方法的确切数量未知但预期很小时，使用已解释的正则表达式的效果最佳。随着方法调用数量的增加，执行速度变慢对性能的影响会超过减少启动时间带来的性能改进。

将编译通过 Compiled 选项的规范绑定到正则表达式引擎的正则表达式模式。这意味着，当实例化正则表达式对象时或当调用一种静态正则表达式方法并且在缓存中找不到该正则表达式时，正则表达式引擎会将该正则表达式转换为一组中间操作代码，这些代码之后会转换为 MSIL。调用方法时，JIT 编译器将执行该 MSIL。与已解释的正则表达式相比，已编译的正则表达式增加了启动时间，但执行各种模式匹配方法的速度更快。因此，相对于调用的正则表达式方法的数量，因编译正则表达式而产生的性能产生了改进。

简言之，当你使用特定正则表达式调用正则表达式方法相对不频繁时，建议使用已解释的正则表达式。当你使用特定正则表达式调用正则表达式方法相对频繁时，应使用已编译的正则表达式。很难确定已解释的正则表达式执行速度减慢超出启动时间减少带来的性能增益的确切阈值，或已编译的正则表达式启动速度减慢超出执行速度加快带来的性能增益的阈值。这依赖于各种因素，包括正则表达式的复杂程度和它处理的特定数据。若要确定已解释或已编译的正则表达式是否可为特定应用程序方案提供最佳性能，可以使用 Stopwatch 类来比较其执

行时间。

下面的示例比较了已编译和已解释正则表达式在读取 Theodore Dreiser 所著《金融家》中前十句文本和所有句文本时的性能。如示例输出所示，当只对匹配方法的正则表达式进行十次调用时，已解释的正则表达式与已编译的正则表达式相比，可提供更好的性能。但是，当进行大量调用（在此示例中，超过 13,000 次调用）时，已编译的正则表达式可提供更好的性能。

```
using System;
using System.Diagnostics;
using System.IO;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]";
      Stopwatch sw;
      Match match;
      int ctr;

      StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");
      string input = inFile.ReadToEnd();
      inFile.Close();

      // Read first ten sentences with interpreted regex.
      Console.WriteLine("10 Sentences with Interpreted Regex:");
      sw = Stopwatch.StartNew();
      Regex int10 = new Regex(pattern, RegexOptions.Singleline);
      match = int10.Match(input);
      for (ctr = 0; ctr <= 9; ctr++) {
         if (match.Success)
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
         else
            break;
      }
      sw.Stop();
      Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed);

      // Read first ten sentences with compiled regex.
      Console.WriteLine("10 Sentences with Compiled Regex:");
      sw = Stopwatch.StartNew();
      Regex comp10 = new Regex(pattern,
                  RegexOptions.Singleline | RegexOptions.Compiled);
      match = comp10.Match(input);
      for (ctr = 0; ctr <= 9; ctr++) {
         if (match.Success)
            // Do nothing with the match except get the next match.
            match = match.NextMatch();
         else
            break;
      }
      sw.Stop();
      Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed);

      // Read all sentences with interpreted regex.
      Console.WriteLine("All Sentences with Interpreted Regex:");
      sw = Stopwatch.StartNew();
      Regex intAll = new Regex(pattern, RegexOptions.Singleline);
      match = intAll.Match(input);
      int matches = 0;
      while (match.Success) {
         matches++;
         // Do nothing with the match except get the next match.
         match = match.NextMatch();
      }
```

```
      sw.Stop();
      Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed);

      // Read all sentences with compiled regex.
      Console.WriteLine("All Sentences with Compiled Regex:");
      sw = Stopwatch.StartNew();
      Regex compAll = new Regex(pattern,
                      RegexOptions.Singleline | RegexOptions.Compiled);
      match = compAll.Match(input);
      matches = 0;
      while (match.Success) {
         matches++;
         // Do nothing with the match except get the next match.
         match = match.NextMatch();
      }
      sw.Stop();
      Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed);
   }
}
// The example displays the following output:
//       10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0047491
//       10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0141872
//       All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1929928
//       All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7635869
//
//       >compare1
//       10 Sentences with Interpreted Regex:
//          10 matches in 00:00:00.0046914
//       10 Sentences with Compiled Regex:
//          10 matches in 00:00:00.0143727
//       All Sentences with Interpreted Regex:
//          13,443 matches in 00:00:01.1514100
//       All Sentences with Compiled Regex:
//          13,443 matches in 00:00:00.7432921
```

```
Imports System.Diagnostics
Imports System.IO
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]"
        Dim sw As Stopwatch
        Dim match As Match
        Dim ctr As Integer

        Dim inFile As New StreamReader(".\Dreiser_TheFinancier.txt")
        Dim input As String = inFile.ReadToEnd()
        inFile.Close()

        ' Read first ten sentences with interpreted regex.
        Console.WriteLine("10 Sentences with Interpreted Regex:")
        sw = Stopwatch.StartNew()
        Dim int10 As New Regex(pattern, RegexOptions.SingleLine)
        match = int10.Match(input)
        For ctr = 0 To 9
            If match.Success Then
                ' Do nothing with the match except get the next match.
                match = match.NextMatch()
            Else
                Exit For
            End If
        Next
```

```
        sw.Stop()
        Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed)

        ' Read first ten sentences with compiled regex.
        Console.WriteLine("10 Sentences with Compiled Regex:")
        sw = Stopwatch.StartNew()
        Dim comp10 As New Regex(pattern,
                    RegexOptions.SingleLine Or RegexOptions.Compiled)
        match = comp10.Match(input)
        For ctr = 0 To 9
            If match.Success Then
                ' Do nothing with the match except get the next match.
                match = match.NextMatch()
            Else
                Exit For
            End If
        Next
        sw.Stop()
        Console.WriteLine("   {0} matches in {1}", ctr, sw.Elapsed)

        ' Read all sentences with interpreted regex.
        Console.WriteLine("All Sentences with Interpreted Regex:")
        sw = Stopwatch.StartNew()
        Dim intAll As New Regex(pattern, RegexOptions.SingleLine)
        match = intAll.Match(input)
        Dim matches As Integer = 0
        Do While match.Success
            matches += 1
            ' Do nothing with the match except get the next match.
            match = match.NextMatch()
        Loop
        sw.Stop()
        Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed)

        ' Read all sentences with compiled regex.
        Console.WriteLine("All Sentences with Compiled Regex:")
        sw = Stopwatch.StartNew()
        Dim compAll As New Regex(pattern,
                       RegexOptions.SingleLine Or RegexOptions.Compiled)
        match = compAll.Match(input)
        matches = 0
        Do While match.Success
            matches += 1
            ' Do nothing with the match except get the next match.
            match = match.NextMatch()
        Loop
        sw.Stop()
        Console.WriteLine("   {0:N0} matches in {1}", matches, sw.Elapsed)
    End Sub
End Module
' The example displays output like the following:
'       10 Sentences with Interpreted Regex:
'          10 matches in 00:00:00.0047491
'       10 Sentences with Compiled Regex:
'          10 matches in 00:00:00.0141872
'       All Sentences with Interpreted Regex:
'          13,443 matches in 00:00:01.1929928
'       All Sentences with Compiled Regex:
'          13,443 matches in 00:00:00.7635869
'
'       >compare1
'       10 Sentences with Interpreted Regex:
'          10 matches in 00:00:00.0046914
'       10 Sentences with Compiled Regex:
'          10 matches in 00:00:00.0143727
'       All Sentences with Interpreted Regex:
'          13,443 matches in 00:00:01.1514100
'       All Sentences with Compiled Regex:
'          13,443 matches in 00:00:00.7432921
```

该示例中使用的正则表达式模式 `\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]` 的定义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`\w+`	匹配一个或多个单词字符。
`(\r?\n)|,?\s)`	匹配零个或一个回车符后跟一个换行符，或零个或一个逗号后跟一个空白字符。
`(\w+((\r?\n)|,?\s))*`	匹配一个或多个单词字符的零个或多个事例，后跟零个或一个回车符和换行符，或后跟零个或一个逗号、一个空格字符。
`\w+`	匹配一个或多个单词字符。
`[.?:;!]`	匹配句号、问号、冒号、分号或感叹号。

正则表达式：编译为程序集

借助 .NET，还可以创建包含已编译正则表达式的程序集。这样会将正则表达式编译对性能造成的影响从运行时转移到设计时。但是，这还涉及一些其他工作：必须提前定义正则表达式并将其编译为程序集。然后，编译器在编译使用该程序集的正则表达式的源代码时，可以引用此程序集。程序集内的每个已编译正则表达式都由从 Regex 派生的类来表示。

若要将正则表达式编译为程序集，可调用 Regex.CompileToAssembly(RegexCompilationInfo[], AssemblyName) 方法并向其传递表示要编译的正则表达式的 RegexCompilationInfo 对象数组和包含有关要创建的程序集的信息的 AssemblyName 对象。

建议你在以下情况下将正则表达式编译为程序集：

- 如果你是要创建可重用正则表达式库的组件开发人员。
- 如果你预期正则表达式的模式匹配方法要被调用的次数无法确定 -- 从任意位置，次数可能为一次两次到上千上万次。与已编译或已解释的正则表达式不同，编译为单独程序集的正则表达式可提供与方法调用数量无关的一致性能。

如果使用已编译的正则表达式来优化性能，则不应使用反射来创建程序集，加载正则表达式引擎并执行其模式匹配方法。这要求你避免动态生成正则表达式模式，并且要在创建程序集时指定模式匹配选项（如不区分大小写的模式匹配）。它还要求将创建程序集的代码与使用正则表达式的代码分离。

下面的示例演示如何创建包含已编译的正则表达式的程序集。它创建包含一个正则表达式类 `SentencePattern` 的程序集 `RegexLib.dll`，其中包含已解释与已编译的正则表达式部分中使用的句子匹配的正则表达式模式。

```
using System;
using System.Reflection;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      RegexCompilationInfo SentencePattern =
                          new RegexCompilationInfo(@"\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]",
                                                   RegexOptions.Multiline,
                                                   "SentencePattern",
                                                   "Utilities.RegularExpressions",
                                                   true);
      RegexCompilationInfo[] regexes = { SentencePattern };
      AssemblyName assemName = new AssemblyName("RegexLib, Version=1.0.0.1001, Culture=neutral,
PublicKeyToken=null");
      Regex.CompileToAssembly(regexes, assemName);
   }
}
```

```
Imports System.Reflection
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim SentencePattern As New RegexCompilationInfo("\b(\w+((\r?\n)|,?\s))*\w+[.?:;!]",
                                                        RegexOptions.Multiline,
                                                        "SentencePattern",
                                                        "Utilities.RegularExpressions",
                                                        True)
        Dim regexes() As RegexCompilationInfo = {SentencePattern}
        Dim assemName As New AssemblyName("RegexLib, Version=1.0.0.1001, Culture=neutral,
PublicKeyToken=null")
        Regex.CompileToAssembly(regexes, assemName)
    End Sub
End Module
```

在将示例编译为可执行文件并运行时，它会创建一个名为 `RegexLib.dll` 的程序集。正则表达式用名为 `Utilities.RegularExpressions.SentencePattern` 并由 Regex 派生的类来表示。然后，下面的示例使用已编译正则表达式，从 Theodore Dreiser 所著《金融家》文本中提取句子。

```
using System;
using System.IO;
using System.Text.RegularExpressions;
using Utilities.RegularExpressions;

public class Example
{
   public static void Main()
   {
      SentencePattern pattern = new SentencePattern();
      StreamReader inFile = new StreamReader(@".\Dreiser_TheFinancier.txt");
      string input = inFile.ReadToEnd();
      inFile.Close();

      MatchCollection matches = pattern.Matches(input);
      Console.WriteLine("Found {0:N0} sentences.", matches.Count);
   }
}
// The example displays the following output:
//       Found 13,443 sentences.
```

```
Imports System.IO
Imports System.Text.RegularExpressions
Imports Utilities.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As New SentencePattern()
        Dim inFile As New StreamReader(".\Dreiser_TheFinancier.txt")
        Dim input As String = inFile.ReadToEnd()
        inFile.Close()

        Dim matches As MatchCollection = pattern.Matches(input)
        Console.WriteLine("Found {0:N0} sentences.", matches.Count)
    End Sub
End Module
' The example displays the following output:
'       Found 13,443 sentences.
```

控制回溯

通常，正则表达式引擎使用线性进度在输入字符串中移动并将其编译为正则表达式模式。但是，当在正则表达式模式中使用不确定限定符（如 `*`、`+` 和 `?`）时，正则表达式引擎可能会放弃一部分成功的分部匹配，并返回以前保存的状态，以便为整个模式搜索成功匹配。此过程称为回溯。

> **NOTE**
> 若要详细了解回溯，请参阅正则表达式行为的详细信息和回溯。若要详细了解回溯，请参阅 BCL 团队博客中的 Optimizing Regular Expression Performance, Part II:Taking Charge of Backtracking（优化正则表达式性能，第 II 部分：控制回溯）。

支持回溯可为正则表达式提供强大的功能和灵活性。还可将控制正则表达式引擎操作的职责交给正则表达式开发人员来处理。由于开发人员通常不了解此职责，因此其误用回溯或依赖过多回溯通常会显著降低正则表达式的性能。在最糟糕的情况下，输入字符串中每增加一个字符，执行时间会加倍。实际上，如果过多使用回溯，则在输入与正则表达式模式近似匹配时很容易创建无限循环的编程等效形式；正则表达式引擎可能需要几小时甚至几天来处理相对短的输入字符串。

通常，尽管回溯不是匹配所必需的，但应用程序会因使用回溯而对性能产生负面影响。例如，正则表达式 `\b\p{Lu}\w*\b` 将匹配以大写字符开头的所有单词，如下表所示。

“	“
`\b`	在单词边界处开始匹配。
`\p{Lu}`	匹配大写字符。
`\w*`	匹配零个或多个单词字符。
`\b`	在单词边界处结束匹配。

由于单词边界与单词字符不同也不是其子集，因此正则表达式引擎在匹配单词字符时无法跨越单词边界。这意味着，对于此正则表达式而言，回溯对任何匹配的总体成功不会有任何贡献 -- 由于正则表达式引擎被强制为单词字符的每个成功的初步匹配保存其状态，因此它只会降低性能。

如果确定不需要回溯，可使用 `(?>subexpression)` 语言元素（被称为原子组）来禁用它。下面的示例通过使用两个正则表达式来分析输入字符串。第一个正则表达式 `\b\p{Lu}\w*\b` 依赖于回溯。第二个正则表达式 `\b\p{Lu}(?>\w*)\b` 禁用回溯。如示例输出所示，这两个正则表达式产生的结果相同。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This this word Sentence name Capital";
      string pattern = @"\b\p{Lu}\w*\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);

      Console.WriteLine();

      pattern = @"\b\p{Lu}(?>\w*)\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       This
//       Sentence
//       Capital
//
//       This
//       Sentence
//       Capital
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This this word Sentence name Capital"
        Dim pattern As String = "\b\p{Lu}\w*\b"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
        Console.WriteLine()

        pattern = "\b\p{Lu}(?>\w*)\b"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       This
'       Sentence
'       Capital
'
'       This
'       Sentence
'       Capital
```

在许多情况下，在将正则表达式模式与输入文本匹配时，回溯很重要。但是，过度回溯会严重降低性能，并且会产生应用程序已停止响应的感觉。特别需要指出的是，当嵌套限定符并且与外部子表达式匹配的文本为与内部子表达式匹配的文本的子集时，尤其会出现这种情况。

> **WARNING**
>
> 除避免过度回溯之外，还应使用超时功能以确保过度回溯不会严重降低正则表达式性能。有关详细信息，请参阅使用超时值部分。

例如，正则表达式模式 `^[0-9A-Z]([-.\w]*[0-9A-Z])*\$$` 用于匹配至少包括一个字母数字字符的部件号。任何附加字符可以包含字母数字字符、连字符、下划线或句号，但最后一个字符必须为字母数字。美元符号用于终止部件号。在某些情况下，由于限定符嵌套并且子表达式 `[0-9A-Z]` 是子表达式 `[-.\w]*` 的子集，因此此正则表达式模式会表现出极差的性能。

在这些情况下，可通过移除嵌套限定符并将外部子表达式替换为零宽度预测先行和回顾断言来优化正则表达式性能。预测先行和回顾断言是定位点；它们不在输入字符串中移动指针，而是通过预测先行或回顾来检查是否满足指定条件。例如，可将部件号正则表达式重写为 `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$`。此正则表达式模式的定义如下表所示。

“	“
`^`	从输入字符串的开头部分开始匹配。
`[0-9A-Z]`	匹配字母数字字符。部件号至少要包含此字符。
`[-.\w]*`	匹配零个或多个任意单词字符、连字符或句号。
`\$`	匹配美元符号。
`(?<=[0-9A-Z])`	查看作为结束的美元符号，以确保前一个字符是字母数字。
`$`	在输入字符串末尾结束匹配。

下面的示例演示了如何使用此正则表达式来匹配包含可能部件号的数组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$";
      string[] partNos = { "A1C$", "A4", "A4$", "A1603D$", "A1603D#" };

      foreach (var input in partNos) {
         Match match = Regex.Match(input, pattern);
         if (match.Success)
            Console.WriteLine(match.Value);
         else
            Console.WriteLine("Match not found.");
      }
   }
}
// The example displays the following output:
//       A1C$
//       Match not found.
//       A4$
//       A1603D$
//       Match not found.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^[0-9A-Z][-.\w]*(?<=[0-9A-Z])\$$"
        Dim partNos() As String = {"A1C$", "A4", "A4$", "A1603D$",
                                   "A1603D#"}

        For Each input As String In partNos
            Dim match As Match = Regex.Match(input, pattern)
            If match.Success Then
                Console.WriteLine(match.Value)
            Else
                Console.WriteLine("Match not found.")
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       A1C$
'       Match not found.
'       A4$
'       A1603D$
'       Match not found.
```

.NET 中的正则表达式语言包括以下可用于消除嵌套限定符的语言元素。有关详细信息，请参阅 分组构造。

[[[[	[[
`(?=` `subexpression` `)`	零宽度正预测先行。预测先行当前位置，以确定 `subexpression` 是否与输入字符串匹配。
`(?!` `subexpression` `)`	零宽度负预测先行。预测先行当前位置，以确定 `subexpression` 是否不与输入字符串匹配。
`(?<=` `subexpression` `)`	零宽度正回顾。回顾后发当前位置，以确定 `subexpression` 是否与输入字符串匹配。
`(?<!` `subexpression` `)`	零宽度负回顾。回顾后发当前位置，以确定 `subexpression` 是否不与输入字符串匹配。

使用超时值

如果正则表达式处理与正则表达式模式大致匹配的输入，则通常依赖于会严重影响其性能的过度回溯。除认真考虑对回溯的使用以及针对大致匹配输入对正则表达式进行测试之外，还应始终设置一个超时值以确保最大程度地降低过度回溯的影响（如果有）。

正则表达式超时间隔定义了在超时前正则表达式引擎用于查找单个匹配项的时间长度。默认超时间隔为 Regex.InfiniteMatchTimeout，这意味着正则表达式不会超时。可以按如下所示重写此值并定义超时间隔：

- 在实例化一个 Regex 对象（通过调用 Regex(String, RegexOptions, TimeSpan) 构造函数）时，提供一个超时值。
- 调用静态模式匹配方法，如 Regex.Match(String, String, RegexOptions, TimeSpan) 或 Regex.Replace(String, String, String, RegexOptions, TimeSpan)，其中包含 `matchTimeout` 参数。
- 对于通过调用 Regex.CompileToAssembly 方法创建的已编译的正则表达式，可调用带有 TimeSpan 类型的参数的构造函数。

如果定义了超时间隔并且在此间隔结束时未找到匹配项，则正则表达式方法将引发 RegexMatchTimeoutException 异常。在异常处理程序中，可以选择使用一个更长的超时间隔来重试匹配、放弃匹配尝试并假定没有匹配项，或者放弃匹配尝试并记录异常信息以供未来分析。

下面的示例定义了一种 `GetWordData` 方法，此方法实例化了一个正则表达式，使其具有 350 毫秒的超时间隔，用于计算文本文件中的词语数和一个词语中的平均字符数。如果匹配操作超时，则超时间隔将延长 350 毫秒并重新实例化 Regex 对象。如果新的超时间隔超过 1 秒，则此方法将再次向调用方引发异常。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      RegexUtilities util = new RegexUtilities();
      string title = "Doyle - The Hound of the Baskervilles.txt";
      try {
         var info = util.GetWordData(title);
         Console.WriteLine("Words:               {0:N0}", info.Item1);
         Console.WriteLine("Average Word Length: {0:N2} characters", info.Item2);
      }
      catch (IOException e) {
         Console.WriteLine("IOException reading file '{0}'", title);
         Console.WriteLine(e.Message);
      }
      catch (RegexMatchTimeoutException e) {
         Console.WriteLine("The operation timed out after {0:N0} milliseconds",
                           e.MatchTimeout.TotalMilliseconds);
      }
   }
}

public class RegexUtilities
{
   public Tuple<int, double> GetWordData(string filename)
   {
      const int MAX_TIMEOUT = 1000;   // Maximum timeout interval in milliseconds.
      const int INCREMENT = 350;      // Milliseconds increment of timeout.

      List<string> exclusions = new List<string>( new string[] { "a", "an", "the" });
      int[] wordLengths = new int[29];        // Allocate an array of more than ample size.
      string input = null;
      StreamReader sr = null;
      try {
         sr = new StreamReader(filename);
         input = sr.ReadToEnd();
      }
      catch (FileNotFoundException e) {
         string msg = String.Format("Unable to find the file '{0}'", filename);
         throw new IOException(msg, e);
      }
      catch (IOException e) {
         throw new IOException(e.Message, e);
      }
      finally {
         if (sr != null) sr.Close();
      }

      int timeoutInterval = INCREMENT;
      bool init = false;
      Regex rgx = null;
      Match m = null;
      int indexPos = 0;
      do {
```

```
      do {
         try {
            if (! init) {
               rgx = new Regex(@"\b\w+\b", RegexOptions.None,
                               TimeSpan.FromMilliseconds(timeoutInterval));
               m = rgx.Match(input, indexPos);
               init = true;
            }
            else {
               m = m.NextMatch();
            }
            if (m.Success) {
               if ( !exclusions.Contains(m.Value.ToLower()))
                  wordLengths[m.Value.Length]++;

               indexPos += m.Length + 1;
            }
         }
         catch (RegexMatchTimeoutException e) {
            if (e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT) {
               timeoutInterval += INCREMENT;
               init = false;
            }
            else {
               // Rethrow the exception.
               throw;
            }
         }
      } while (m.Success);

      // If regex completed successfully, calculate number of words and average length.
      int nWords = 0;
      long totalLength = 0;

      for (int ctr = wordLengths.GetLowerBound(0); ctr <= wordLengths.GetUpperBound(0); ctr++) {
         nWords += wordLengths[ctr];
         totalLength += ctr * wordLengths[ctr];
      }
      return new Tuple<int, double>(nWords, totalLength/nWords);
   }
}
```

```
Imports System.Collections.Generic
Imports System.IO
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim util As New RegexUtilities()
        Dim title As String = "Doyle - The Hound of the Baskervilles.txt"
        Try
            Dim info = util.GetWordData(title)
            Console.WriteLine("Words:               {0:N0}", info.Item1)
            Console.WriteLine("Average Word Length: {0:N2} characters", info.Item2)
        Catch e As IOException
            Console.WriteLine("IOException reading file '{0}'", title)
            Console.WriteLine(e.Message)
        Catch e As RegexMatchTimeoutException
            Console.WriteLine("The operation timed out after {0:N0} milliseconds",
                              e.MatchTimeout.TotalMilliseconds)
        End Try
    End Sub
End Module

Public Class RegexUtilities
    Public Function GetWordData(filename As String) As Tuple(Of Integer, Double)
        Const MAX_TIMEOUT As Integer = 1000  ' Maximum timeout interval in milliseconds.
        Const INCREMENT As Integer = 250     ' Milliseconds increment of timeout
```

```
        Const INCREMENT As Integer = 350       Milliseconds increment of timeout.

        Dim exclusions As New List(Of String)({"a", "an", "the"})
        Dim wordLengths(30) As Integer        ' Allocate an array of more than ample size.
        Dim input As String = Nothing
        Dim sr As StreamReader = Nothing
        Try
            sr = New StreamReader(filename)
            input = sr.ReadToEnd()
        Catch e As FileNotFoundException
            Dim msg As String = String.Format("Unable to find the file '{0}'", filename)
            Throw New IOException(msg, e)
        Catch e As IOException
            Throw New IOException(e.Message, e)
        Finally
            If sr IsNot Nothing Then sr.Close()
        End Try

        Dim timeoutInterval As Integer = INCREMENT
        Dim init As Boolean = False
        Dim rgx As Regex = Nothing
        Dim m As Match = Nothing
        Dim indexPos As Integer = 0
        Do
            Try
                If Not init Then
                    rgx = New Regex("\b\w+\b", RegexOptions.None,
                                    TimeSpan.FromMilliseconds(timeoutInterval))
                    m = rgx.Match(input, indexPos)
                    init = True
                Else
                    m = m.NextMatch()
                End If
                If m.Success Then
                    If Not exclusions.Contains(m.Value.ToLower()) Then
                        wordLengths(m.Value.Length) += 1
                    End If
                    indexPos += m.Length + 1
                End If
            Catch e As RegexMatchTimeoutException
                If e.MatchTimeout.TotalMilliseconds < MAX_TIMEOUT Then
                    timeoutInterval += INCREMENT
                    init = False
                Else
                    ' Rethrow the exception.
                    Throw
                End If
            End Try
        Loop While m.Success

        ' If regex completed successfully, calculate number of words and average length.
        Dim nWords As Integer
        Dim totalLength As Long

        For ctr As Integer = wordLengths.GetLowerBound(0) To wordLengths.GetUpperBound(0)
            nWords += wordLengths(ctr)
            totalLength += ctr * wordLengths(ctr)
        Next
        Return New Tuple(Of Integer, Double)(nWords, totalLength / nWords)
    End Function
End Class
```

只在必要时捕获

.NET 中的正则表达式支持许多分组构造，这样，便可以将正则表达式模式分组为一个或多个子表达式。.NET 正则表达式语言中最常用的分组构造为 `(` subexpression `)`（用于定义编号捕获组）和 `(?<` name `>` subexpression

`)`(用于定义命名捕获组)。分组构造是创建反向引用和定义要应用限定符的子表达式时所必需的。

但是，使用这些语言元素会产生一定的开销。它们会导致用最近的未命名或已命名捕获来填充 GroupCollection 属性返回的 Match.Groups 对象，如果单个分组构造已捕获输入字符串中的多个子字符串，则还会填充包含多个 CaptureCollection 对象的特定捕获组的 Group.Captures 属性返回的 Capture 对象。

通常，只在正则表达式中使用分组构造，这样可对其应用限定符，而且以后不会使用这些子表达式捕获的组。例如，正则表达式 `\b(\w+[;,]?\s?)+[.?!]` 用于捕获整个句子。下表描述了此正则表达式模式中的语言元素及其对 Match 对象的 Match.Groups 和 Group.Captures 集合的影响。

“	“
`\b`	在单词边界处开始匹配。
`\w+`	匹配一个或多个单词字符。
`[;,]?`	匹配零个或一个逗号或分号。
`\s?`	匹配零个或一个空白字符。
`(\w+[;,]?\s?)+`	匹配以下一个或多个事例：一个或多个单词字符，后跟一个可选逗号或分号，一个可选的空白字符。用于定义第一个捕获组，它是必需的，以便将重复多个单词字符的组合(即单词)后跟可选标点符号，直至正则表达式引擎到达句子末尾。
`[.?!]`	匹配句号、问号或感叹号。

如下面的示例所示，当找到匹配时，GroupCollection 和 CaptureCollection 对象都将用匹配中的捕获内容来填充。在此情况下，存在捕获组 `(\w+[;,]?\s?)`，因此可对其应用 `+` 限定符，从而使得正则表达式模式可与句子中的每个单词匹配。否则，它将匹配句子中的最后一个单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is one sentence. This is another.";
      string pattern = @"\b(\w+[;,]?\s?)+[.?!]";

      foreach (Match match in Regex.Matches(input, pattern)) {
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index);
         int grpCtr = 0;
         foreach (Group grp in match.Groups) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index);
            int capCtr = 0;
            foreach (Capture cap in grp.Captures) {
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index);
               capCtr++;
            }
            grpCtr++;
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//          Group 1: 'sentence' at index 12.
//             Capture 0: 'This ' at 0.
//             Capture 1: 'is ' at 5.
//             Capture 2: 'one ' at 8.
//             Capture 3: 'sentence' at 12.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
//          Group 1: 'another' at index 30.
//             Capture 0: 'This ' at 22.
//             Capture 1: 'is ' at 27.
//             Capture 2: 'another' at 30.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is one sentence. This is another."
        Dim pattern As String = "\b(\w+[;,]?\s?)+[.?!]"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Match: '{0}' at index {1}.",
                              match.Value, match.Index)
            Dim grpCtr As Integer = 0
            For Each grp As Group In match.Groups
                Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                                  grpCtr, grp.Value, grp.Index)
                Dim capCtr As Integer = 0
                For Each cap As Capture In grp.Captures
                    Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                      capCtr, cap.Value, cap.Index)
                    capCtr += 1
                Next
                grpCtr += 1
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       Match: 'This is one sentence.' at index 0.
'          Group 0: 'This is one sentence.' at index 0.
'             Capture 0: 'This is one sentence.' at 0.
'          Group 1: 'sentence' at index 12.
'             Capture 0: 'This ' at 0.
'             Capture 1: 'is ' at 5.
'             Capture 2: 'one ' at 8.
'             Capture 3: 'sentence' at 12.
'
'       Match: 'This is another.' at index 22.
'          Group 0: 'This is another.' at index 22.
'             Capture 0: 'This is another.' at 22.
'          Group 1: 'another' at index 30.
'             Capture 0: 'This ' at 22.
'             Capture 1: 'is ' at 27.
'             Capture 2: 'another' at 30.
```

当你只使用子表达式来对其应用限定符并且你对捕获的文本不感兴趣时，应禁用组捕获。例如，`(?:subexpression)` 语言元素可防止应用此元素的组捕获匹配的子字符串。在下面的示例中，上一示例中的正则表达式模式更改为 `\b(?:\w+[;,]?\s?)+[.?!]`。正如输出所示，它禁止正则表达式引擎填充 GroupCollection 和 CaptureCollection 集合。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is one sentence. This is another.";
      string pattern = @"\b(?:\w+[;,]?\s?)+[.?!]";

      foreach (Match match in Regex.Matches(input, pattern)) {
         Console.WriteLine("Match: '{0}' at index {1}.",
                           match.Value, match.Index);
         int grpCtr = 0;
         foreach (Group grp in match.Groups) {
            Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                              grpCtr, grp.Value, grp.Index);
            int capCtr = 0;
            foreach (Capture cap in grp.Captures) {
               Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                 capCtr, cap.Value, cap.Index);
               capCtr++;
            }
            grpCtr++;
         }
         Console.WriteLine();
      }
   }
}
// The example displays the following output:
//       Match: 'This is one sentence.' at index 0.
//          Group 0: 'This is one sentence.' at index 0.
//             Capture 0: 'This is one sentence.' at 0.
//
//       Match: 'This is another.' at index 22.
//          Group 0: 'This is another.' at index 22.
//             Capture 0: 'This is another.' at 22.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is one sentence. This is another."
        Dim pattern As String = "\b(?:\w+[;,]?\s?)+[.?!]"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Match: '{0}' at index {1}.",
                              match.Value, match.Index)
            Dim grpCtr As Integer = 0
            For Each grp As Group In match.Groups
                Console.WriteLine("   Group {0}: '{1}' at index {2}.",
                                  grpCtr, grp.Value, grp.Index)
                Dim capCtr As Integer = 0
                For Each cap As Capture In grp.Captures
                    Console.WriteLine("      Capture {0}: '{1}' at {2}.",
                                      capCtr, cap.Value, cap.Index)
                    capCtr += 1
                Next
                grpCtr += 1
            Next
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays the following output:
'       Match: 'This is one sentence.' at index 0.
'          Group 0: 'This is one sentence.' at index 0.
'             Capture 0: 'This is one sentence.' at 0.
'
'       Match: 'This is another.' at index 22.
'          Group 0: 'This is another.' at index 22.
'             Capture 0: 'This is another.' at 22.
```

可以通过以下方式之一来禁用捕获：

- 使用 `(?:subexpression)` 语言元素。此元素可防止在它应用的组中捕获匹配的子字符串。它不在任何嵌套的组中禁用子字符串捕获。
- 使用 ExplicitCapture 选项。在正则表达式模式中禁用所有未命名或隐式捕获。使用此选项时，只能捕获与使用 `(?<name>subexpression)` 语言元素定义的命名组匹配的子字符串。可将 ExplicitCapture 标记传递给 `options` 类构造函数的 Regex 参数或 `options` 静态匹配方法的 Regex 参数。
- 在 `n` 语言元素中使用 `(?imnsx)` 选项。此选项将在元素出现的正则表达式模式中的点处禁用所有未命名或隐式捕获。捕获将一直禁用到模式结束或 `(-n)` 选项启用未命名或隐式捕获。有关详细信息，请参阅其他构造。
- 在 `n` 语言元素中使用 `(?imnsx:subexpression)` 选项。此选项可在 `subexpression` 中禁用所有未命名或隐式捕获。同时禁用任何未命名或隐式的嵌套捕获组进行的任何捕获。

相关主题

TITLE	tt
正则表达式行为的详细信息	在 .NET 中检查正则表达式引擎的实现。该主题重点介绍正则表达式的灵活性，并说明开发人员确保正则表达式引擎高效、强健运行的职责。
回溯	说明何为回溯及其对正则表达式性能有何影响，并检查为回溯提供替代项的语言元素。

TITLE	𝐂
正则表达式语言 - 快速参考	介绍 .NET 中的正则表达式语言的元素，并提供每个语言元素的详细文档链接。

正则表达式对象模型

2021/11/16 •

本主题介绍了处理 .NET 正则表达式时使用的对象模型。它包含下列部分：

- 正则表达式引擎
- MatchCollection 和 Match 对象
- 组集合
- 捕获组
- 捕获集合
- 单个捕获

正则表达式引擎

.NET 中的正则表达式引擎由 Regex 类表示。正则表达式引擎负责分析和编译正则表达式，并执行用于将正则表达式模式与输入字符串相匹配的操作。此引擎是 .NET 正则表达式对象模型中的主要组件。

可以通过以下两种方式之一使用正则表达式引擎：

- 通过调用 Regex 类的静态方法。方法参数包含输入字符串和正则表达式模式。正则表达式引擎会缓存静态方法调用中使用的正则表达式，这样一来，重复调用使用同一正则表达式的静态正则表达式方法将提供相对良好的性能。
- 通过实例化 Regex 对象，采用的方式是将一个正则表达式传递给类构造函数。在此情况下，Regex 对象是不可变的（只读），它表示一个与单个正则表达式紧密耦合的正则表达式引擎。由于未对 Regex 实例使用的正则表达式进行缓存，因此不应使用同一正则表达式实例化 Regex 对象多次。

可以调用 Regex 类的方法来执行下列操作：

- 确定字符串是否与正则表达式模式匹配。
- 提取单个匹配项或第一个匹配项。
- 提取所有匹配项。
- 替换匹配的子字符串。
- 将单个字符串拆分成一个字符串数组。

以下各部分对这些操作进行了描述。

匹配正则表达式模式

如果字符串与此模式匹配，则 Regex.IsMatch 方法返回 `true`；如果字符串与此模式不匹配，则该方法返回 `false`。IsMatch 方法通常用于验证字符串输入。例如，下面的代码将确保字符串与有效的美国社会保障号匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] values = { "111-22-3333", "111-2-3333"};
      string pattern = @"^\d{3}-\d{2}-\d{4}$";
      foreach (string value in values) {
         if (Regex.IsMatch(value, pattern))
            Console.WriteLine("{0} is a valid SSN.", value);
         else
            Console.WriteLine("{0}: Invalid", value);
      }
   }
}
// The example displays the following output:
//       111-22-3333 is a valid SSN.
//       111-2-3333: Invalid
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim values() As String = {"111-22-3333", "111-2-3333"}
        Dim pattern As String = "^\d{3}-\d{2}-\d{4}$"
        For Each value As String In values
            If Regex.IsMatch(value, pattern) Then
                Console.WriteLine("{0} is a valid SSN.", value)
            Else
                Console.WriteLine("{0}: Invalid", value)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       111-22-3333 is a valid SSN.
'       111-2-3333: Invalid
```

正则表达式模式 `^\d{3}-\d{2}-\d{4}$` 的含义如下表所示。

“	“
`^`	匹配输入字符串的开头部分。
`\d{3}`	匹配三个十进制数字。
`-`	匹配连字符。
`\d{2}`	匹配两个十进制数字。
`-`	匹配连字符。
`\d{4}`	匹配四个十进制数字。
`$`	匹配输入字符串的末尾部分。

提取单个匹配项或第一个匹配项

Regex.Match 方法返回一个 Match 对象，该对象包含有关与正则表达式模式匹配的第一个子字符串的信息。如果 `Match.Success` 属性返回 `true`，则表示已找到一个匹配项，可以通过调用 Match.NextMatch 方法来检索有关后续匹配项的信息。这些方法调用可以继续进行，直到 `Match.Success` 属性返回 `false`。例如，下面的代码使用 Regex.Match(String, String) 方法查找重复的单词在字符串中的第一个匹配项。然后，此代码调用 Match.NextMatch 方法查找任何其他匹配项。该示例将在每次调用方法后检查 `Match.Success` 属性以确定当前匹配是否成功，并确定是否应接着调用 Match.NextMatch 方法。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is a a farm that that raises dairy cattle."
        Dim pattern As String = "\b(\w+)\W+(\1)\b"
        Dim match As Match = Regex.Match(input, pattern)
        Do While match.Success
            Console.WriteLine("Duplicate '{0}' found at position {1}.", _
                              match.Groups(1).Value, match.Groups(2).Index)
            match = match.NextMatch()
        Loop
    End Sub
End Module
' The example displays the following output:
'       Duplicate 'a' found at position 10.
'       Duplicate 'that' found at position 22.
```

正则表达式模式 `\b(\w+)\W+(\1)\b` 的含义如下表所示。

“	“
`\b`	在单词边界处开始匹配。
`(\w+)`	匹配一个或多个单词字符。这是第一个捕获组。
`\W+`	匹配一个或多个非单词字符。
`(\1)`	与第一个捕获的字符串匹配。这是第二个捕获组。

𝐈𝐈	𝐈𝐈
`\b`	在单词边界处结束匹配。

提取所有匹配项

Regex.Matches 方法返回一个 MatchCollection 对象，该对象包含有关正则表达式引擎在输入字符串中找到的所有匹配项的信息。例如，可重写上一示例以调用 Matches 方法，而不是调用 Match 和 NextMatch 方法。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "This is a a farm that that raises dairy cattle.";
      string pattern = @"\b(\w+)\W+(\1)\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("Duplicate '{0}' found at position {1}.",
                           match.Groups[1].Value, match.Groups[2].Index);
   }
}
// The example displays the following output:
//       Duplicate 'a' found at position 10.
//       Duplicate 'that' found at position 22.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "This is a a farm that that raises dairy cattle."
        Dim pattern As String = "\b(\w+)\W+(\1)\b"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("Duplicate '{0}' found at position {1}.", _
                              match.Groups(1).Value, match.Groups(2).Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       Duplicate 'a' found at position 10.
'       Duplicate 'that' found at position 22.
```

替换匹配的子字符串

Regex.Replace 方法会将与正则表达式模式匹配的每个子字符串替换为指定的字符串或正则表达式模式，并返回进行了替换的整个输入字符串。例如，下面的代码在字符串的十进制数字前添加了美国货币符号。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+\.\d{2}\b";
      string replacement = "$$$&";
      string input = "Total Cost: 103.64";
      Console.WriteLine(Regex.Replace(input, pattern, replacement));
   }
}
// The example displays the following output:
//       Total Cost: $103.64
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b\d+\.\d{2}\b"
        Dim replacement As String = "$$$&"
        Dim input As String = "Total Cost: 103.64"
        Console.WriteLine(Regex.Replace(input, pattern, replacement))
    End Sub
End Module
' The example displays the following output:
'       Total Cost: $103.64
```

正则表达式模式 `\b\d+\.\d{2}\b` 的含义如下表所示。

`\b`	在单词边界处开始匹配。
`\d+`	匹配一个或多个十进制数字。
`\.`	匹配句点。
`\d{2}`	匹配两个十进制数字。
`\b`	在单词边界处结束匹配。

替换模式 `$$$&` 的含义如下表所示。

`$$`	美元符号 ($) 字符。
`$&`	整个匹配的子字符串。

将单个字符串拆分成一个字符串数组

Regex.Split 方法在由正则表达式匹配项定义的位置拆分输入字符串。例如，下面的代码将编号列表中的项置于字符串数组中。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea";
      string pattern = @"\b\d{1,2}\.\s";
      foreach (string item in Regex.Split(input, pattern))
      {
         if (! String.IsNullOrEmpty(item))
            Console.WriteLine(item);
      }
   }
}
// The example displays the following output:
//       Eggs
//       Bread
//       Milk
//       Coffee
//       Tea
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "1. Eggs 2. Bread 3. Milk 4. Coffee 5. Tea"
        Dim pattern As String = "\b\d{1,2}\.\s"
        For Each item As String In Regex.Split(input, pattern)
            If Not String.IsNullOrEmpty(item) Then
                Console.WriteLine(item)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       Eggs
'       Bread
'       Milk
'       Coffee
'       Tea
```

正则表达式模式 `\b\d{1,2}\.\s` 的含义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`\d{1,2}`	匹配一个或两个十进制数字。
`\.`	匹配句点。
`\s`	与空白字符匹配。

MatchCollection 和 Match 对象

Regex 方法返回作为正则表达式对象模型的一部分的两个对象：MatchCollection 对象和 Match 对象。

Match 集合

Regex.Matches 方法返回一个 MatchCollection 对象，该对象包含多个 Match 对象，这些对象表示正则表达式引擎在输入字符串中找到的所有匹配项（其顺序为这些匹配项在输入字符串中的显示顺序）。如果没有匹配项，则此方法将返回一个不包含任何成员的 MatchCollection 对象。利用 MatchCollection.Item[] 属性，你可以按照索引（从零到将 MatchCollection.Count 属性的值减 1 所得的值）访问集合中的各个成员。Item[] 是集合的索引器（在 C# 中）和默认属性（在 Visual Basic 中）。

默认情况下，调用 Regex.Matches 方法会使用延迟计算来填充 MatchCollection 对象。访问需要完全填充的集合的属性（如 MatchCollection.Count 和 MatchCollection.Item[] 属性）可能会降低性能。因此，建议你使用由 IEnumerator 方法返回的 MatchCollection.GetEnumerator 对象访问该集合。各种语言都提供了用于包装集合的 IEnumerator 接口的构造（如 Visual Basic 中的 `For Each` 和 C# 中的 `foreach`）。

下面的示例使用 Regex.Matches(String) 方法将在输入字符串中找到的所有匹配项填充到 MatchCollection 对象中。此示例枚举了该集合，将匹配项复制到字符串数组并将字符位置记录在整数数组中。

```
using System;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
       MatchCollection matches;
       List<string> results = new List<string>();
       List<int> matchposition = new List<int>();

       // Create a new Regex object and define the regular expression.
       Regex r = new Regex("abc");
       // Use the Matches method to find all matches in the input string.
       matches = r.Matches("123abc4abcd");
       // Enumerate the collection to retrieve all matches and positions.
       foreach (Match match in matches)
       {
          // Add the match string to the string array.
           results.Add(match.Value);
           // Record the character position where the match was found.
           matchposition.Add(match.Index);
       }
       // List the results.
       for (int ctr = 0; ctr < results.Count; ctr++)
         Console.WriteLine("'{0}' found at position {1}.",
                           results[ctr], matchposition[ctr]);
   }
}
// The example displays the following output:
//       'abc' found at position 3.
//       'abc' found at position 7.
```

```
Imports System.Collections.Generic
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim matches As MatchCollection
        Dim results As New List(Of String)
        Dim matchposition As New List(Of Integer)

        ' Create a new Regex object and define the regular expression.
        Dim r As New Regex("abc")
        ' Use the Matches method to find all matches in the input string.
        matches = r.Matches("123abc4abcd")
        ' Enumerate the collection to retrieve all matches and positions.
        For Each match As Match In matches
            ' Add the match string to the string array.
            results.Add(match.Value)
            ' Record the character position where the match was found.
            matchposition.Add(match.Index)
        Next
        ' List the results.
        For ctr As Integer = 0 To results.Count - 1
            Console.WriteLine("'{0}' found at position {1}.", _
                              results(ctr), matchposition(ctr))
        Next
    End Sub
End Module
' The example displays the following output:
'       'abc' found at position 3.
'       'abc' found at position 7.
```

Match 类

Match 类表示单个正则表达式匹配项的结果。可以通过两种方式访问 Match 对象：

- 通过从 MatchCollection 方法返回的 Regex.Matches 对象检索这些对象。若要检索各个 Match 对象，请通过使用 `foreach`（在 C# 中）或 `For Each` ... `Next`（在 Visual Basic 中）构造循环访问集合；或者使用 MatchCollection.Item[] 属性以按索引或名称检索特定的 Match 对象。也可以通过按索引（从零到将集合中的对象数减去 1 所得的值）循环访问集合来检索集合中的各个 Match 对象。但是，此方法不使用延迟计算，因为它将访问 MatchCollection.Count 属性。

  下面的示例通过使用 Match 或 MatchCollection... `foreach` 构造循环访问集合，来从 `For Each` 对象中检索各个 `Next` 对象。正则表达式只是与输入字符串中的字符串“abc”匹配。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "abc";
      string input = "abc123abc456abc789";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("{0} found at position {1}.",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       abc found at position 0.
//       abc found at position 6.
//       abc found at position 12.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "abc"
        Dim input As String = "abc123abc456abc789"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("{0} found at position {1}.", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       abc found at position 0.
'       abc found at position 6.
'       abc found at position 12.
```

- 通过调用 Regex.Match 方法，此方法返回一个 Match 对象，该对象表示字符串中的第一个匹配项或字符串的一部分。可以通过检索 `Match.Success` 属性的值确定是否已找到匹配项。若要检索表示后续匹配项的 Match 对象，请重复调用 Match.NextMatch 方法，直到返回的 `Success` 对象的 Match 属性为 `false`。

  下面的示例使用 Regex.Match(String, String) 和 Match.NextMatch 方法来匹配输入字符串中的字符串“abc”。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "abc";
      string input = "abc123abc456abc789";
      Match match = Regex.Match(input, pattern);
      while (match.Success)
      {
         Console.WriteLine("{0} found at position {1}.",
                           match.Value, match.Index);
         match = match.NextMatch();
      }
   }
}
// The example displays the following output:
//       abc found at position 0.
//       abc found at position 6.
//       abc found at position 12.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "abc"
        Dim input As String = "abc123abc456abc789"
        Dim match As Match = Regex.Match(input, pattern)
        Do While match.Success
            Console.WriteLine("{0} found at position {1}.", _
                              match.Value, match.Index)
            match = match.NextMatch()
        Loop
    End Sub
End Module
' The example displays the following output:
'       abc found at position 0.
'       abc found at position 6.
'       abc found at position 12.
```

Match 类的以下两个属性都将返回集合对象：

- Match.Groups 属性返回一个 GroupCollection 对象，该对象包含有关与正则表达式模式中的捕获组匹配的子字符串的信息。
- `Match.Captures` 属性返回一个 CaptureCollection 对象，该对象的使用是有限制的。不会为其 `Success` 属性为 `false` 的 Match 的对象填充集合。否则，它将包含一个 Capture 对象，该对象具有的信息与 Match 对象具有的信息相同。

有关这些对象的更多信息，请参阅本主题后面的组集合和捕获集合部分。

Match 类的另外两个属性提供了有关匹配项的信息。`Match.Value` 属性返回输入字符串中与正则表达式模式匹配的子字符串。`Match.Index` 属性返回输入字符串中匹配的字符串的起始位置（从零开始）。

Match 类还具有两个模式匹配方法：

- Match.NextMatch 方法查找位于由当前的 Match 对象表示的匹配项之后的匹配项，并返回表示该匹配项的 Match 对象。
- Match.Result 方法对匹配的字符串执行指定的替换操作并返回相应结果。

下面的示例使用 Match.Result 方法在每个包含两个小数位的数字前预置一个 $ 符号和一个空格。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b\d+(,\d{3})*\.\d{2}\b";
      string input = "16.32\n194.03\n1,903,672.08";

      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine(match.Result("$$ $&"));
   }
}
// The example displays the following output:
//       $ 16.32
//       $ 194.03
//       $ 1,903,672.08
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b\d+(,\d{3})*\.\d{2}\b"
        Dim input As String = "16.32" + vbCrLf + "194.03" + vbCrLf + "1,903,672.08"

        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine(match.Result("$$ $&"))
        Next
    End Sub
End Module
' The example displays the following output:
'       $ 16.32
'       $ 194.03
'       $ 1,903,672.08
```

正则表达式模式 `\b\d+(,\d{3})*\.\d{2}\b` 的定义如下表所示。

“	“
`\b`	在单词边界处开始匹配。
`\d+`	匹配一个或多个十进制数字。
`(,\d{3})*`	匹配零个或多个以下模式：一个逗号后跟三个十进制数字。
`\.`	匹配小数点字符。
`\d{2}`	匹配两个十进制数字。
`\b`	在单词边界处结束匹配。

替换模式 `$$ $&` 指示匹配的子字符串应由美元符号 ($)（`$$` 模式）、空格和匹配项的值（`$&` 模式）替换。

返回页首

组集合

Match.Groups 属性返回一个 GroupCollection 对象，该对象包含多个 Group 对象，这些对象表示单个匹配项中的捕获的组。集合中的第一个 Group 对象（位于索引 0 处）表示整个匹配项。此对象后面的每个对象均表示一个捕获组的结果。

可以使用 Group 属性检索集合中的各个 GroupCollection.Item[] 对象。可以在集合中按未命名组的序号位置来检索未命名组，也可以按命名组的名称或序号位置来检索命名组。未命名捕获将首先在集合中显示，并将按照未命名捕获在正则表达式模式中出现的顺序从左至右对它们进行索引。在对未命名捕获进行索引后，将按照命名捕获在正则表达式模式中出现的顺序从左至右对它们进行索引。若要确定在特定的正则表达式匹配方法返回的集合中哪些编号的组可用，可以调用实例 Regex.GetGroupNumbers 方法。若要确定集合中哪些命名的组可用，可以调用实例 Regex.GetGroupNames 方法。这两种方法在分析通过任何正则表达式找到的匹配的常规用途例程中都特别有用。

GroupCollection.Item[] 属性是集合的索引器（在 C# 中）和集合对象的默认属性（在 Visual Basic 中）。这表示可以按索引（对于命名组，可以按名称）访问各个 Group 对象，如下所示：

```
Group group = match.Groups[ctr];
```

```
Dim group As Group = match.Groups(ctr)
```

下面的示例定义一个正则表达式，该表达式使用分组构造捕获日期的年、月和日部分。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(\w+)\s(\d{1,2}),\s(\d{4})\b";
      string input = "Born: July 28, 1989";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
         for (int ctr = 0; ctr <  match.Groups.Count; ctr++)
            Console.WriteLine("Group {0}: {1}", ctr, match.Groups[ctr].Value);
    }
}
// The example displays the following output:
//       Group 0: July 28, 1989
//       Group 1: July
//       Group 2: 28
//       Group 3: 1989
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(\w+)\s(\d{1,2}),\s(\d{4})\b"
        Dim input As String = "Born: July 28, 1989"
        Dim match As Match = Regex.Match(input, pattern)
        If match.Success Then
            For ctr As Integer = 0 To match.Groups.Count - 1
                Console.WriteLine("Group {0}: {1}", ctr, match.Groups(ctr).Value)
            Next
        End If
    End Sub
End Module
' The example displays the following output:
'       Group 0: July 28, 1989
'       Group 1: July
'       Group 2: 28
'       Group 3: 1989
```

正则表达式模式 `\b(\w+)\s(\d{1,2}),\s(\d{4})\b` 的定义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`(\w+)`	匹配一个或多个单词字符。这是第一个捕获组。
`\s`	与空白字符匹配。
`(\d{1,2})`	匹配一个或两个十进制数字。这是第二个捕获组。
`,`	匹配逗号。

“	“
`\s`	与空白字符匹配。
`(\d{4})`	匹配四个十进制数字。这是第三个捕获组。
`\b`	在单词边界处结束匹配。

返回页首

捕获的组

Group 类表示来自单个捕获组的结果。表示正则表达式中定义的捕获组的组对象由 Item[] 属性所返回的 GroupCollection 对象的 Match.Groups 属性返回。Item[] 属性是索引器(在 C# 中)和 Group 类的默认属性(在 Visual Basic 中)。也可以使用 `foreach` 或 `For Each` 构造循环访问集合，从而检索各个成员。有关示例，请参见上一部分。

下面的示例使用嵌套的分组构造来将子字符串捕获到组中。正则表达式模式 `(a(b))c` 将匹配字符串“abc”。它会将子字符串“ab”分配给第一个捕获组，并将子字符串“b”分配给第二个捕获组。

```
var matchposition = new List<int>();
var results = new List<string>();
// Define substrings abc, ab, b.
var r = new Regex("(a(b))c");
Match m = r.Match("abdabc");
for (int i = 0; m.Groups[i].Value != ""; i++)
{
    // Add groups to string array.
    results.Add(m.Groups[i].Value);
    // Record character position.
    matchposition.Add(m.Groups[i].Index);
}

// Display the capture groups.
for (int ctr = 0; ctr < results.Count; ctr++)
    Console.WriteLine("{0} at position {1}",
                      results[ctr], matchposition[ctr]);
// The example displays the following output:
//       abc at position 3
//       ab at position 3
//       b at position 4
```

```
Dim matchposition As New List(Of Integer)
Dim results As New List(Of String)
' Define substrings abc, ab, b.
Dim r As New Regex("(a(b))c")
Dim m As Match = r.Match("abdabc")
Dim i As Integer = 0
While Not (m.Groups(i).Value = "")
    ' Add groups to string array.
    results.Add(m.Groups(i).Value)
    ' Record character position.
    matchposition.Add(m.Groups(i).Index)
    i += 1
End While

' Display the capture groups.
For ctr As Integer = 0 to results.Count - 1
    Console.WriteLine("{0} at position {1}", _
                      results(ctr), matchposition(ctr))
Next
' The example displays the following output:
'       abc at position 3
'       ab at position 3
'       b at position 4
```

下面的示例使用命名的分组构造，从包含“DATANAME:VALUE”格式的数据的字符串中捕获子字符串，正则表达式通过冒号 (:) 拆分数据。

```
var r = new Regex(@"^(?<name>\w+):(?<value>\w+)");
Match m = r.Match("Section1:119900");
Console.WriteLine(m.Groups["name"].Value);
Console.WriteLine(m.Groups["value"].Value);
// The example displays the following output:
//       Section1
//       119900
```

```
Dim r As New Regex("^(?<name>\w+):(?<value>\w+)")
Dim m As Match = r.Match("Section1:119900")
Console.WriteLine(m.Groups("name").Value)
Console.WriteLine(m.Groups("value").Value)
' The example displays the following output:
'       Section1
'       119900
```

正则表达式模式 `^(?<name>\w+):(?<value>\w+)` 的定义如下表所示。

“	“
`^`	从输入字符串的开头部分开始匹配。
`(?<name>\w+)`	匹配一个或多个单词字符。此捕获组的名称为 `name`。
`:`	匹配冒号。
`(?<value>\w+)`	匹配一个或多个单词字符。此捕获组的名称为 `value`。

Group 类的属性提供有关捕获组的信息：`Group.Value` 属性包含捕获子字符串，`Group.Index` 属性在输入文本中指示捕获组的起始位置，`Group.Length` 属性包含捕获文本的长度，`Group.Success` 属性指示子字符串是否与捕获组所定义的模式匹配。

通过对组应用量符(有关详细信息，请参阅量符)，可以每捕获组修改一个捕获的关系，具体方式分为以下两种：

- 如果对组应用 `*` 或 `*?` 限定符(将指定零个或多个匹配项)，则捕获组在输入字符串中可能没有匹配项。在没有捕获的文本时，将如下表所示设置 Group 对象的属性。

""	"["
`Success`	`false`
`Value`	String.Empty
`Length`	0

下面的示例进行了这方面的演示。在正则表达式模式 `aaa(bbb)*ccc` 中，可以匹配第一个捕获组(子字符串"bbb")零次或多次。由于输入字符串"aaaccc"与此模式匹配，因此该捕获组没有匹配项。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "aaa(bbb)*ccc";
      string input = "aaaccc";
      Match match = Regex.Match(input, pattern);
      Console.WriteLine("Match value: {0}", match.Value);
      if (match.Groups[1].Success)
         Console.WriteLine("Group 1 value: {0}", match.Groups[1].Value);
      else
         Console.WriteLine("The first capturing group has no match.");
   }
}
// The example displays the following output:
//       Match value: aaaccc
//       The first capturing group has no match.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "aaa(bbb)*ccc"
        Dim input As String = "aaaccc"
        Dim match As Match = Regex.Match(input, pattern)
        Console.WriteLine("Match value: {0}", match.Value)
        If match.Groups(1).Success Then
            Console.WriteLine("Group 1 value: {0}", match.Groups(1).Value)
        Else
            Console.WriteLine("The first capturing group has no match.")
        End If
    End Sub
End Module
' The example displays the following output:
'       Match value: aaaccc
'       The first capturing group has no match.
```

- 限定符可以匹配由捕获组定义的模式的多个匹配项。在此情况下，`Value` 对象的 `Length` 和 Group 属性仅包含有关最后捕获的子字符串的信息。例如，下面的正则表达式匹配以句点结束的单个句子。此表达式使用两个分组构造：第一个分组构造捕获单个单词和空白字符；第二个分组构造捕获单个单词。如示例中的输出所示，虽然正则表达式成功捕获整个句子，但第二个捕获组仅捕获了最后一个单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b((\w+)\s?)+\.";
      string input = "This is a sentence. This is another sentence.";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: " + match.Value);
         Console.WriteLine("Group 2: " + match.Groups[2].Value);
      }
   }
}
// The example displays the following output:
//       Match: This is a sentence.
//       Group 2: sentence
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b((\w+)\s?)+\."
        Dim input As String = "This is a sentence. This is another sentence."
        Dim match As Match = Regex.Match(input, pattern)
        If match.Success Then
            Console.WriteLine("Match: " + match.Value)
            Console.WriteLine("Group 2: " + match.Groups(2).Value)
        End If
    End Sub
End Module
' The example displays the following output:
'       Match: This is a sentence.
'       Group 2: sentence
```

返回页首

捕获集合

Group 对象仅包含有关最后一个捕获的信息。但仍可从 CaptureCollection 属性返回的 Group.Captures 对象中获取由捕获组生成的整个捕获集。集合中的每个成员均为一个表示由该捕获组生成的捕获的 Capture 对象，这些对象按被捕获的顺序排列(因而也就是遵循在输入字符串中按从左至右匹配捕获的字符串的顺序)。可以通过以下两种方式之一来检索集合中的各个 Capture 对象：

- 通过使用构造循环访问集合，如 `foreach` 构造(在 C# 中)或 `For Each` 构造(在 Visual Basic 中)。
- 通过使用 CaptureCollection.Item[] 属性按索引检索特定对象。Item[] 属性是 CaptureCollection 对象的默认属性(在 Visual Basic 中)或索引器(在 C# 中)。

如果未对捕获组应用限定符，则 CaptureCollection 对象将包含一个 Capture 对象，但该对象的作用不大，因为它提供的是有关与其 Group 对象相同的匹配项的信息。如果对一个捕获组应用限定符，则 CaptureCollection 对象将包含该捕获组所生成的所有捕获，并且集合的最后一个成员将表示与 Group 对象相同的捕获。

例如，如果使用正则表达式模式 `((a(b))c)+`(其中，+ 限定符指定一个或多个匹配项)捕获字符串“abcabcabc”中的匹配项，则每个 CaptureCollection 对象的 Group 对象都将包含三个成员。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "((a(b))c)+";
      string input = "abcabcabc";

      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Match: '{0}' at position {1}",
                           match.Value, match.Index);
         GroupCollection groups = match.Groups;
         for (int ctr = 0; ctr < groups.Count; ctr++) {
            Console.WriteLine("   Group {0}: '{1}' at position {2}",
                              ctr, groups[ctr].Value, groups[ctr].Index);
            CaptureCollection captures = groups[ctr].Captures;
            for (int ctr2 = 0; ctr2 < captures.Count; ctr2++) {
               Console.WriteLine("      Capture {0}: '{1}' at position {2}",
                                 ctr2, captures[ctr2].Value, captures[ctr2].Index);
            }
         }
      }
   }
}
// The example displays the following output:
//       Match: 'abcabcabc' at position 0
//          Group 0: 'abcabcabc' at position 0
//             Capture 0: 'abcabcabc' at position 0
//          Group 1: 'abc' at position 6
//             Capture 0: 'abc' at position 0
//             Capture 1: 'abc' at position 3
//             Capture 2: 'abc' at position 6
//          Group 2: 'ab' at position 6
//             Capture 0: 'ab' at position 0
//             Capture 1: 'ab' at position 3
//             Capture 2: 'ab' at position 6
//          Group 3: 'b' at position 7
//             Capture 0: 'b' at position 1
//             Capture 1: 'b' at position 4
//             Capture 2: 'b' at position 7
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "((a(b))c)+"
        Dim input As STring = "abcabcabc"

        Dim match As Match = Regex.Match(input, pattern)
        If match.Success Then
            Console.WriteLine("Match: '{0}' at position {1}", _
                              match.Value, match.Index)
            Dim groups As GroupCollection = match.Groups
            For ctr As Integer = 0 To groups.Count - 1
                Console.WriteLine("   Group {0}: '{1}' at position {2}", _
                                  ctr, groups(ctr).Value, groups(ctr).Index)
                Dim captures As CaptureCollection = groups(ctr).Captures
                For ctr2 As Integer = 0 To captures.Count - 1
                    Console.WriteLine("      Capture {0}: '{1}' at position {2}", _
                                      ctr2, captures(ctr2).Value, captures(ctr2).Index)
                Next
            Next
        End If
    End Sub
End Module
' The example dosplays the following output:
'       Match: 'abcabcabc' at position 0
'          Group 0: 'abcabcabc' at position 0
'             Capture 0: 'abcabcabc' at position 0
'          Group 1: 'abc' at position 6
'             Capture 0: 'abc' at position 0
'             Capture 1: 'abc' at position 3
'             Capture 2: 'abc' at position 6
'          Group 2: 'ab' at position 6
'             Capture 0: 'ab' at position 0
'             Capture 1: 'ab' at position 3
'             Capture 2: 'ab' at position 6
'          Group 3: 'b' at position 7
'             Capture 0: 'b' at position 1
'             Capture 1: 'b' at position 4
'             Capture 2: 'b' at position 7
```

下面的示例使用正则表达式 `(Abc)+` 来在字符串“XYZAbcAbcAbcXYZAbcAb”中查找字符串“Abc”的一个或多个连续匹配项。该示例演示了使用 Group.Captures 属性来返回多组捕获的子字符串。

```
int counter;
Match m;
CaptureCollection cc;
GroupCollection gc;

// Look for groupings of "Abc".
var r = new Regex("(Abc)+");
// Define the string to search.
m = r.Match("XYZAbcAbcAbcXYZAbcAb");
gc = m.Groups;

// Display the number of groups.
Console.WriteLine("Captured groups = " + gc.Count.ToString());

// Loop through each group.
for (int i = 0; i < gc.Count; i++)
{
    cc = gc[i].Captures;
    counter = cc.Count;

    // Display the number of captures in this group.
    Console.WriteLine("Captures count = " + counter.ToString());

    // Loop through each capture in the group.
    for (int ii = 0; ii < counter; ii++)
    {
        // Display the capture and its position.
        Console.WriteLine(cc[ii] + "   Starts at character " +
             cc[ii].Index);
    }
}
// The example displays the following output:
//       Captured groups = 2
//       Captures count = 1
//       AbcAbcAbc   Starts at character 3
//       Captures count = 3
//       Abc   Starts at character 3
//       Abc   Starts at character 6
//       Abc   Starts at character 9
```

```
Dim counter As Integer
Dim m As Match
Dim cc As CaptureCollection
Dim gc As GroupCollection

' Look for groupings of "Abc".
Dim r As New Regex("(Abc)+")
' Define the string to search.
m = r.Match("XYZAbcAbcAbcXYZAbcAb")
gc = m.Groups

' Display the number of groups.
Console.WriteLine("Captured groups = " & gc.Count.ToString())

' Loop through each group.
Dim i, ii As Integer
For i = 0 To gc.Count - 1
    cc = gc(i).Captures
    counter = cc.Count

    ' Display the number of captures in this group.
    Console.WriteLine("Captures count = " & counter.ToString())

    ' Loop through each capture in the group.
    For ii = 0 To counter - 1
        ' Display the capture and its position.
        Console.WriteLine(cc(ii).ToString() _
            & "   Starts at character " & cc(ii).Index.ToString())
    Next ii
Next i
' The example displays the following output:
'       Captured groups = 2
'       Captures count = 1
'       AbcAbcAbc   Starts at character 3
'       Captures count = 3
'       Abc   Starts at character 3
'       Abc   Starts at character 6
'       Abc   Starts at character 9
```

返回页首

单个捕获

Capture 类包含来自单个子表达式捕获的结果。Capture.Value 属性包含匹配的文本，而 Capture.Index 属性指示匹配的子字符串在输入字符串中的起始位置（从零开始）。

下面的示例分析针对选定城市的温度的输入字符串。逗号（","）用于将城市与其温度分隔开，而分号（";"）用于将每个城市的数据分隔开。整个输入字符串表示一个匹配项。在用于分析字符串的正则表达式模式 `((\w+(\s\w+)*),(\d+);)+` 中，城市名称将分配给第二个捕获组，而温度将分配到第四个捕获组。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "Miami,78;Chicago,62;New York,67;San Francisco,59;Seattle,58;";
      string pattern = @"((\w+(\s\w+)*),(\d+);)+";
      Match match = Regex.Match(input, pattern);
      if (match.Success)
      {
         Console.WriteLine("Current temperatures:");
         for (int ctr = 0; ctr < match.Groups[2].Captures.Count; ctr++)
            Console.WriteLine("{0,-20} {1,3}", match.Groups[2].Captures[ctr].Value,
                              match.Groups[4].Captures[ctr].Value);
      }
   }
}
// The example displays the following output:
//       Current temperatures:
//       Miami                78
//       Chicago              62
//       New York             67
//       San Francisco        59
//       Seattle              58
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "Miami,78;Chicago,62;New York,67;San Francisco,59;Seattle,58;"
        Dim pattern As String = "((\w+(\s\w+)*),(\d+);)+"
        Dim match As Match = Regex.Match(input, pattern)
        If match.Success Then
            Console.WriteLine("Current temperatures:")
            For ctr As Integer = 0 To match.Groups(2).Captures.Count - 1
                Console.WriteLine("{0,-20} {1,3}", match.Groups(2).Captures(ctr).Value, _
                                  match.Groups(4).Captures(ctr).Value)
            Next
        End If
    End Sub
End Module
' The example displays the following output:
'       Current temperatures:
'       Miami                78
'       Chicago              62
'       New York             67
'       San Francisco        59
'       Seattle              58
```

该正则表达式的定义如下表所示。

“	“
`\w+`	匹配一个或多个单词字符。
`(\s\w+)*`	匹配零个或多个以下模式：一个空白字符后跟一个或多个单词字符。此模式匹配包含多个单词的城市名称。这是第三个捕获组。

"	"
`(\w+(\s\w+)*)`	匹配以下模式：一个或多个单词字符，后跟零个或多个一个空白字符与一个或多个单词字符的组合。这是第二个捕获组。
`,`	匹配逗号。
`(\d+)`	匹配一个或多个数字。这是第四个捕获组。
`;`	匹配分号。
`((\w+(\s\w+)*),(\d+);)+`	匹配一个或多个以下模式：一个单词后跟任何其他单词，后跟一个逗号、一个或多个数字和一个分号。这是第一个捕获组。

请参阅

- System.Text.RegularExpressions
- .NET 正则表达式
- 正则表达式语言 - 快速参考

正则表达式行为的详细信息

2021/11/16 •

.NET 正则表达式引擎是回溯正则表达式匹配程序，其中包含传统的非确定性有限自动机 (NFA) 引擎（如 Perl、Python、Emacs 和 Tcl 所使用的引擎）。这使它有别于速度更快、但是限制更多的纯正则表达式确定性有限自动机 (DFA) 引擎（如 awk、egrep 或 lex 中的引擎）。这也使它有别于标准化、但速度较慢的 POSIX NFA。以下部分介绍正则表达式引擎的三种类型，并说明为何使用传统 NFA 引擎实现 .NET 中的正则表达式。

NFA 引擎的优势

DFA 引擎执行模式匹配时，其处理顺序由输入字符串驱动。该引擎从输入字符串的开头处开始，按顺序继续进行以确定下一个字符是否与正则表达式模式匹配。它们可以保证匹配可能最长的字符串。因为它们绝不会对相同字符测试两次，所以 DFA 引擎不支持回溯。但是，由于 DFA 引擎只包含有限状态，因此它无法匹配具有反向引用的模式，并且因为它不构造显式扩展，所以无法捕获子表达式。

与 DFA 引擎不同，传统 NFA 引擎执行模式匹配时，其处理顺序由正则表达式模式驱动。处理特定语言元素时，该引擎使用贪婪匹配；也就是说，它尽可能多地匹配输入字符串。但是，它还会在成功匹配子表达式之后保存其状态。如果匹配最终失败，则该引擎可以返回到已保存状态，以便可以尝试其他匹配项。这样的过程称为“回溯”，即放弃成功子表达式匹配，以便正则表达式中后面的语言元素也可以进行匹配。NFA 引擎使用回溯按特定顺序测试正则表达式的所有可能扩展，并接受第一个匹配项。因为传统 NFA 引擎针对成功匹配构造正则表达式的特定扩展，所以它可以捕获子表达式匹配项以及匹配反向引用。但是，由于传统 NFA 会进行回溯，因此如果它通过不同路径到达相同状态，则可能会多次访问该状态。因此，其运行速度在最糟糕的情况下可能会极端缓慢。因为传统 NFA 引擎接受它找到的第一个匹配项，所以它还可能会使其他（可能更长）的匹配项保持未发现状态。

POSIX NFA 引擎类似于传统 NFA 引擎，只不过它们会继续回溯，直到可以保证已找到最长的可能匹配项。因此，POSIX NFA 引擎速度低于传统 NFA 引擎，而且使用 POSIX NFA 引擎时，无法通过更改回溯搜索的顺序，使较短匹配项优先于较长匹配项。

程序员更喜欢传统 NFA 引擎，因为与 DFA 或 POSIX NFA 引擎相比，通过它们可更好地控制字符串匹配。不过在最糟糕的情况下，它们可能会运行缓慢，你可以使用减少歧义和限制回溯的模式，控制它们以线性方式或在多项式时间内找到匹配项。换句话说，虽然 NFA 引擎以牺牲性能为代价来实现强大功能和灵活性，不过在大多数情况下，如果正则表达式编写良好并避免回溯呈指数级降低性能的情况，它们可提供可接受的良好性能。

> **NOTE**
> 若要了解过度回溯导致的性能损失，以及如何生成正则表达式来解决此问题，请参阅回溯。

.NET 引擎功能

为了利用传统 NFA 引擎的优势，.NET 正则表达式引擎包括一组全面的构造，使程序员可以控制回溯引擎。这些构造可以用于更快地找到匹配项或使特定扩展优先于其他扩展。

.NET 正则表达式引擎的其他功能包括以下这些：

- 惰性限定符：`??`、`*?`、`+?`、`{n,m}?`。这些构造会指示回溯引擎首先搜索最小数量的重复项。相反，普通贪婪限定符会尝试首先匹配最大数量的重复项。以下示例演示了两者之间的差异。正则表达式匹配以数字结尾的句子，捕获组旨在提取该数字。正则表达式 `.+(\d+)\.` 包含贪婪限定符 `.+`，这使正则表达式引擎仅捕获数字的最后一位数。相反，正则表达式 `.+?(\d+)\.` 包含惰性限定符 `.+?`，这使正则表达式引擎捕获整个数字。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string greedyPattern = @".+(\d+)\.";
      string lazyPattern = @".+?(\d+)\.";
      string input = "This sentence ends with the number 107325.";
      Match match;

      // Match using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (greedy): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);

      // Match using lazy quantifier .+?.
      match = Regex.Match(input, lazyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (lazy): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", lazyPattern);
   }
}
// The example displays the following output:
//       Number at end of sentence (greedy): 5
//       Number at end of sentence (lazy): 107325
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim greedyPattern As String = ".+(\d+)\."
        Dim lazyPattern As String = ".+?(\d+)\."
        Dim input As String = "This sentence ends with the number 107325."
        Dim match As Match

        ' Match using greedy quantifier .+.
        match = Regex.Match(input, greedyPattern)
        If match.Success Then
            Console.WriteLine("Number at end of sentence (greedy): {0}",
                              match.Groups(1).Value)
        Else
            Console.WriteLine("{0} finds no match.", greedyPattern)
        End If

        ' Match using lazy quantifier .+?.
        match = Regex.Match(input, lazyPattern)
        If match.Success Then
            Console.WriteLine("Number at end of sentence (lazy): {0}",
                              match.Groups(1).Value)
        Else
            Console.WriteLine("{0} finds no match.", lazyPattern)
        End If
    End Sub
End Module
' The example displays the following output:
'       Number at end of sentence (greedy): 5
'       Number at end of sentence (lazy): 107325
```

下表定义了此正则表达式的贪婪和惰性版本：

"	"
`.+`（贪婪限定符）	匹配任何字符的至少一个匹配项。这会导致正则表达式引擎匹配整个字符串，然后根据需要进行回溯以匹配模式的其余部分。
`.+?`（惰性限定符）	匹配任何字符的至少一个匹配项，但匹配尽可能少。
`(\d+)`	匹配至少一个数字字符，并将其分配给第一个捕获组。
`\.`	匹配句点。

若要详细了解惰性量符，请参阅量符。

- 正预测先行断言：`(?=` subexpression `)`。此功能允许回溯引擎在匹配子表达式之后返回到文本中的相同位置。它可用于通过验证从相同位置开始的多个模式来搜索整个文本。它还允许引擎验证匹配项末尾是否存在某个子字符串，而无需在匹配的文本中包含该子字符串。下面的示例使用正预测先行提取句子中后面不是标点符号的单词。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b[A-Z]+\b(?=\P{P})";
      string input = "If so, what comes next?";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       If
//       what
//       comes
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b[A-Z]+\b(?=\P{P})"
        Dim input As String = "If so, what comes next?"
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       If
'       what
'       comes
```

正则表达式 `\b[A-Z]+\b(?=\P{P})` 的定义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`[A-Z]+`	匹配任何字母字符一次或多次。由于 Regex.Matches 方法是使用 RegexOptions.IgnoreCase 选项进行调用，因此比较不区分大小写。
`\b`	在单词边界处结束匹配。
`(?=\P{P})`	预测先行以确定下一个字符是否为标点符号。如果不是，则匹配成功。

若要详细了解正预测先行断言，请参阅分组构造。

- 负预测先行断言：`(?!` subexpression `)`。通过此功能可以仅当子表达式未能匹配时才匹配表达式。这对于修剪搜索十分有用，因为针对应消除的情况提供表达式通常比针对必须包括的情况提供表达式要更简单。例如，难以为不以"non"开头的单词编写表达式。下面的示例使用负预测先行排除它们。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = @"\b(?!non)\w+\b";
      string input = "Nonsense is not always non-functional.";
      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.IgnoreCase))
         Console.WriteLine(match.Value);
   }
}
// The example displays the following output:
//       is
//       not
//       always
//       functional
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "\b(?!non)\w+\b"
        Dim input As String = "Nonsense is not always non-functional."
        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.IgnoreCase)
            Console.WriteLine(match.Value)
        Next
    End Sub
End Module
' The example displays the following output:
'       is
'       not
'       always
'       functional
```

正则表达式模式 `\b(?!non)\w+\b` 的定义如下表所示。

"	"
`\b`	在单词边界处开始匹配。
`(?!non)`	预测先行以确保当前字符串不以"non"开头。如果以"non"开头，则匹配失败。
`(\w+)`	匹配一个或多个单词字符。
`\b`	在单词边界处结束匹配。

若要详细了解负预测先行断言，请参阅分组构造。

- 条件求值：`(?(` expression `)` yes `|` no `)` 和 `(?(` name `)` yes `|` no `)`，其中 expression 是要匹配的子表达式，name 是捕获组的名称，yes 是在 expression 匹配或 name 是有效的非空捕获组时要匹配的字符串，no 是在 expression 不匹配或 name 不是有效的非空捕获组时要匹配的子表达式。此功能允许引擎使用多个备用模式进行搜索（具体取决于上一个子表达式匹配的结果或零宽度断言的结果）。这样可实现功能更强大的反向引用形式，例如，它允许基于上一个子表达式是否匹配来匹配子表达式。下面示例中的正则表达式匹配旨在供公共和内部使用的段落。仅供内部使用的段落以 `<PRIVATE>` 标记开头。正则表达式模式 `^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$` 使用条件评估将旨在供公共使用和内部使用的段落内容分配给不同的捕获组。这些段落随后可以按不同方式进行处理。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "<PRIVATE> This is not for public consumption." + Environment.NewLine +
                     "But this is for public consumption." + Environment.NewLine +
                     "<PRIVATE> Again, this is confidential.\n";
      string pattern = @"^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?$";
      string publicDocument = null, privateDocument = null;

      foreach (Match match in Regex.Matches(input, pattern, RegexOptions.Multiline))
      {
         if (match.Groups[1].Success) {
            privateDocument += match.Groups[1].Value + "\n";
         }
         else {
            publicDocument += match.Groups[3].Value + "\n";
            privateDocument += match.Groups[3].Value + "\n";
         }
      }

      Console.WriteLine("Private Document:");
      Console.WriteLine(privateDocument);
      Console.WriteLine("Public Document:");
      Console.WriteLine(publicDocument);
   }
}
// The example displays the following output:
//    Private Document:
//    This is not for public consumption.
//    But this is for public consumption.
//    Again, this is confidential.
//
//    Public Document:
//    But this is for public consumption.
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "<PRIVATE> This is not for public consumption." + vbCrLf + _
                              "But this is for public consumption." + vbCrLf + _
                              "<PRIVATE> Again, this is confidential." + vbCrLf
        Dim pattern As String = "^(?<Pvt>\<PRIVATE\>\s)?(?(Pvt)((\w+\p{P}?\s)+)|((\w+\p{P}?\s)+))\r?
$"
        Dim publicDocument As String = Nothing
        Dim privateDocument As String = Nothing

        For Each match As Match In Regex.Matches(input, pattern, RegexOptions.Multiline)
            If match.Groups(1).Success Then
                privateDocument += match.Groups(1).Value + vbCrLf
            Else
                publicDocument += match.Groups(3).Value + vbCrLf
                privateDocument += match.Groups(3).Value + vbCrLf
            End If
        Next

        Console.WriteLine("Private Document:")
        Console.WriteLine(privateDocument)
        Console.WriteLine("Public Document:")
        Console.WriteLine(publicDocument)
    End Sub
End Module
' The example displays the following output:
'    Private Document:
'    This is not for public consumption.
'    But this is for public consumption.
'    Again, this is confidential.
'
'    Public Document:
'    But this is for public consumption.
```

正则表达式模式的定义如下表所示。

"	"
`^`	从行的开头开始匹配。
`(?<Pvt>\<PRIVATE\>\s)?`	匹配后跟一个空白字符的字符串 `<PRIVATE>` 的零个或一个匹配项。将匹配项分配给 `Pvt` 捕获组。
`(?(Pvt)((\w+\p{P}?\s)+)`	如果 `Pvt` 捕获组存在，则匹配后跟零个或一个标点分隔符、再后跟一个空白字符的一个或多个单词字符的一个或多个匹配项。将子字符串分配给第一个捕获组。
`|((\w+\p{P}?\s)+))`	如果 `Pvt` 捕获组不存在，则匹配后跟零个或一个标点分隔符、再后跟一个空白字符的一个或多个单词字符的一个或多个匹配项。将子字符串分配给第三个捕获组。
`\r?$`	匹配行尾或字符串末尾。

若要详细了解条件求值，请参阅替换构造。

- 平衡组定义：`(?<` *name1* `-` *name2* `>` *subexpression* `)` 。此功能允许正则表达式引擎跟踪嵌套构造（如圆括号或者左方括号和右方括号）。有关示例，请参阅分组构造。
- 原子组：`(?>` subexpression `)` 。此功能允许回溯引擎保证子表达式仅匹配为该子表达式找到的第一个匹

配项，就如同该表达式独立于其包含表达式运行一样。如果不使用此构造，则来自较大表达式的回溯搜索可能会更改子表达式的行为。例如，正则表达式 `(a+)\w` 除了一系列“a”字符后面的单词字符匹配，还与一个或多个“a”字符相匹配，并且它将该系列“a”字符分配给第一个捕获组。但是，如果输入字符串的最后一个字符也是“a”，则它由 `\w` 语言元素匹配且不包含在捕获的组中。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "aaaaa", "aaaaab" };
      string backtrackingPattern = @"(a+)\w";
      Match match;

      foreach (string input in inputs) {
         Console.WriteLine("Input: {0}", input);
         match = Regex.Match(input, backtrackingPattern);
         Console.WriteLine("   Pattern: {0}", backtrackingPattern);
         if (match.Success) {
            Console.WriteLine("      Match: {0}", match.Value);
            Console.WriteLine("      Group 1: {0}", match.Groups[1].Value);
         }
         else {
            Console.WriteLine("      Match failed.");
         }
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: (a+)\w
//             Match: aaaaa
//             Group 1: aaaa
//       Input: aaaaab
//          Pattern: (a+)\w
//             Match: aaaaab
//             Group 1: aaaaa
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"aaaaa", "aaaaab"}
        Dim backtrackingPattern As String = "(a+)\w"
        Dim match As Match

        For Each input As String In inputs
            Console.WriteLine("Input: {0}", input)
            match = Regex.Match(input, backtrackingPattern)
            Console.WriteLine("   Pattern: {0}", backtrackingPattern)
            If match.Success Then
                Console.WriteLine("      Match: {0}", match.Value)
                Console.WriteLine("      Group 1: {0}", match.Groups(1).Value)
            Else
                Console.WriteLine("      Match failed.")
            End If
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'       Input: aaaaa
'          Pattern: (a+)\w
'             Match: aaaaa
'             Group 1: aaaa
'       Input: aaaaab
'          Pattern: (a+)\w
'             Match: aaaaab
'             Group 1: aaaaa
```

正则表达式 `((?>a+))\w` 会阻止此行为。因为所有连续“a”字符会在不进行回溯的情况下匹配，所以第一个捕获组包含所有连续“a”字符。如果“a”字符后面不是至少一个“a”之外的字符，则匹配会失败。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "aaaaa", "aaaaab" };
      string nonbacktrackingPattern = @"((?>a+))\w";
      Match match;

      foreach (string input in inputs) {
         Console.WriteLine("Input: {0}", input);
         match = Regex.Match(input, nonbacktrackingPattern);
         Console.WriteLine("   Pattern: {0}", nonbacktrackingPattern);
         if (match.Success) {
            Console.WriteLine("      Match: {0}", match.Value);
            Console.WriteLine("      Group 1: {0}", match.Groups[1].Value);
         }
         else {
            Console.WriteLine("      Match failed.");
         }
      }
      Console.WriteLine();
   }
}
// The example displays the following output:
//       Input: aaaaa
//          Pattern: ((?>a+))\w
//             Match failed.
//       Input: aaaaab
//          Pattern: ((?>a+))\w
//             Match: aaaaab
//             Group 1: aaaaa
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"aaaaa", "aaaaab"}
        Dim nonbacktrackingPattern As String = "((?>a+))\w"
        Dim match As Match

        For Each input As String In inputs
            Console.WriteLine("Input: {0}", input)
            match = Regex.Match(input, nonbacktrackingPattern)
            Console.WriteLine("   Pattern: {0}", nonbacktrackingPattern)
            If match.Success Then
                Console.WriteLine("      Match: {0}", match.Value)
                Console.WriteLine("      Group 1: {0}", match.Groups(1).Value)
            Else
                Console.WriteLine("      Match failed.")
            End If
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays the following output:
'       Input: aaaaa
'          Pattern: ((?>a+))\w
'             Match failed.
'       Input: aaaaab
'          Pattern: ((?>a+))\w
'             Match: aaaaab
'             Group 1: aaaaa
```

要详细了解原子组，请参阅分组构造。

- 从右到左匹配：指定方式为向 Regex 类构造函数或静态实例匹配方法提供 RegexOptions.RightToLeft 选项。当从右到左（而不是从左到右）进行搜索时，或是在从模式右侧部分（而不是左侧部分）开始匹配效率更高的情况下，此功能非常有用。如下面的示例所示，使用从右到左匹配可以更改贪婪限定符的行为。该示例对以数字结尾的句子执行两个搜索。使用贪婪限定符 `+` 的从左到右搜索匹配句子中六个数字之一，而从右到左搜索匹配所有六个数字。有关正则表达式模式的介绍，请参见此部分前面说明惰性限定符的示例。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string greedyPattern = @".+(\d+)\.";
      string input = "This sentence ends with the number 107325.";
      Match match;

      // Match from left-to-right using lazy quantifier .+?.
      match = Regex.Match(input, greedyPattern);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (left-to-right): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);

      // Match from right-to-left using greedy quantifier .+.
      match = Regex.Match(input, greedyPattern, RegexOptions.RightToLeft);
      if (match.Success)
         Console.WriteLine("Number at end of sentence (right-to-left): {0}",
                           match.Groups[1].Value);
      else
         Console.WriteLine("{0} finds no match.", greedyPattern);
   }
}
// The example displays the following output:
//       Number at end of sentence (left-to-right): 5
//       Number at end of sentence (right-to-left): 107325
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim greedyPattern As String = ".+(\d+)\."
        Dim input As String = "This sentence ends with the number 107325."
        Dim match As Match

        ' Match from left-to-right using lazy quantifier .+?.
        match = Regex.Match(input, greedyPattern)
        If match.Success Then
            Console.WriteLine("Number at end of sentence (left-to-right): {0}",
                              match.Groups(1).Value)
        Else
            Console.WriteLine("{0} finds no match.", greedyPattern)
        End If

        ' Match from right-to-left using greedy quantifier .+.
        match = Regex.Match(input, greedyPattern, RegexOptions.RightToLeft)
        If match.Success Then
            Console.WriteLine("Number at end of sentence (right-to-left): {0}",
                              match.Groups(1).Value)
        Else
            Console.WriteLine("{0} finds no match.", greedyPattern)
        End If
    End Sub
End Module
' The example displays the following output:
'       Number at end of sentence (left-to-right): 5
'       Number at end of sentence (right-to-left): 107325
```

有关从右到左匹配的更多信息，请参见正则表达式选项。

- 正负回顾后发断言：`(?<=` subexpression `)`（正回顾后发断言）和 `(?<!` subexpression `)`（负回顾后发断言）。此功能非常类似于本主题前面讨论的预测先行。由于正则表达式引擎允许完全的从右到左匹配，因此正则表达式允许无限制回顾。当嵌套子表达式是外部表达式的超集时，正回顾和负回顾还可以用于避免嵌套限定符。具有此类嵌套限定符的正则表达式通常性能不佳。例如，下面的示例验证字符串是否以字母数字字符开头和结尾，以及字符串中的任何其他字符是否为更大子集之一。它形成用于验证电子邮件地址的正则表达式的一部分内容；有关详细信息，请参阅如何：确认字符串是有效的电子邮件格式。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string[] inputs = { "jack.sprat", "dog#", "dog#1", "me.myself",
                          "me.myself!" };
      string pattern = @"^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$";
      foreach (string input in inputs) {
         if (Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase))
            Console.WriteLine("{0}: Valid", input);
         else
            Console.WriteLine("{0}: Invalid", input);
      }
   }
}
// The example displays the following output:
//       jack.sprat: Valid
//       dog#: Invalid
//       dog#1: Valid
//       me.myself: Valid
//       me.myself!: Invalid
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim inputs() As String = {"jack.sprat", "dog#", "dog#1", "me.myself",
                                  "me.myself!"}
        Dim pattern As String = "^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$"
        For Each input As String In inputs
            If Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase) Then
                Console.WriteLine("{0}: Valid", input)
            Else
                Console.WriteLine("{0}: Invalid", input)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       jack.sprat: Valid
'       dog#: Invalid
'       dog#1: Valid
'       me.myself: Valid
'       me.myself!: Invalid
```

正则表达式 `^[A-Z0-9]([-!#$%&'.*+/=?^`{}|~\w])*(?<=[A-Z0-9])$` 的定义如下表所示。

“	“
`^`	从字符串开头开始匹配。
`[A-Z0-9]`	匹配任意数字或字母数字字符。（比较不区分大小写。）
`([-!#$%&'.*+/=?^`{}|~\w])*`	匹配零个或多个任意单词字符或下列任意字符：-、!、#、$、%、&、'、.、*、+、/、=、?、^、`、{ }、| 或 ~。
`(?<=[A-Z0-9])`	回顾上一个字符（必须是数字或字母数字）。（比较不区分大小写。）

"	"
`$`	在字符串的结尾结束匹配。

若要详细了解正负向后行断言，请参阅分组构造。

相关文章

TITLE	"
回溯	提供有关正则表达式回溯如何进行分支以查找替代匹配的信息。
编译和重用	提供有关编译和重复使用正则表达式以提高性能的信息。
线程安全性	提供有关正则表达式线程安全的信息，并说明何时应同步对正则表达式对象进行的访问。
.NET 正则表达式	提供正则表达式的编程语言方面的概述。
正则表达式对象模型	提供演示如何使用正则表达式类的信息和代码示例。
正则表达式语言 - 快速参考	提供有关可用来定义正则表达式的字符集、运算符和构造的信息。

参考

- System.Text.RegularExpressions

正则表达式中的回溯

2021/11/16 •

当正则表达式模式包含可选限定符或备用构造时，会发生回溯，并且正则表达式引擎会返回以前保存的状态，以继续搜索匹配项。回溯是正则表达式的强大功能的中心；它使得表达式强大、灵活，可以匹配非常复杂的模式。同时，这种强大功能需要付出一定代价。通常，回溯是影响正则表达式引擎性能的单个最重要的因素。幸运的是，开发人员可以控制正则表达式引擎的行为及其使用回溯的方式。本主题说明回溯的工作方式以及如何对其进行控制。

> **NOTE**
> 通常情况下，非确定性有限自动机 (NFA) 引擎（如 .NET 正则表达式引擎）会将构造快速高效的正则表达式的职责交给开发人员。

不使用回溯的线性比较

如果正则表达式模式没有可选限定符或替换构造，正则表达式引擎将以线性时间执行。也就是说，在正则表达式引擎将模式中的第一个语言元素与输入字符串中的文本匹配后，它尝试将模式中的下一个语言元素与输入字符串中的下一个字符或字符组匹配。此操作将继续，直至匹配成功或失败。在任何一种情况下，在同一时间，正则表达式引擎都比输入字符串中提前一个字符。

下面的示例进行了这方面的演示。正则表达式 `e{2}\w\b` 查找字母 "e" 后跟任意单词字符再后跟单词边界的两个匹配项。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "needing a reed";
      string pattern = @"e{2}\w\b";
      foreach (Match match in Regex.Matches(input, pattern))
         Console.WriteLine("{0} found at position {1}",
                           match.Value, match.Index);
   }
}
// The example displays the following output:
//       eed found at position 11
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "needing a reed"
        Dim pattern As String = "e{2}\w\b"
        For Each match As Match In Regex.Matches(input, pattern)
            Console.WriteLine("{0} found at position {1}", _
                              match.Value, match.Index)
        Next
    End Sub
End Module
' The example displays the following output:
'       eed found at position 11
```

尽管此正则表达式包括限定符 `{2}`，但它仍以线性方式进行计算。由于 `{2}` 不是可选限定符，因此该正则表达式引擎不回溯；它指定确切数字，而不是前一个子表达式必须匹配的可变次数。因此，正则表达式引擎尝试使正则表达式模式与输入字符串匹配，如下表所示。

tt	tttttttt	tttttttttt	tt
1	e	“needing a reed”（索引 0）	无匹配。
2	e	“eeding a reed”（索引 1）	可能匹配。
3	e{2}	“eding a reed”（索引 2）	可能匹配。
4	\w	“ding a reed”（索引 3）	可能匹配。
5	\b	“ing a reed”（索引 4）	可能的匹配失败。
6	e	“eding a reed”（索引 2）	可能匹配。
7	e{2}	“ding a reed”（索引 3）	可能的匹配失败。
8	e	“ding a reed”（索引 3）	匹配失败。
9	e	“ing a reed”（索引 4）	无匹配。
10	e	“ng a reed”（索引 5）	无匹配。
11	e	“g a reed”（索引 6）	无匹配。
12	e	“a reed”（索引 7）	无匹配。
13	e	“a reed”（索引 8）	无匹配。
14	e	“reed”（索引 9）	无匹配。
15	e	“reed”（索引 10）	无匹配
16	e	“eed”（索引 11）	可能匹配。
17	e{2}	“ed”（索引 12）	可能匹配。

ㄍ	ㄍㄍㄍㄍㄍㄍ	ㄍㄍㄍㄍㄍㄍㄍㄍ	ㄍ
18	\w	“d”（索引 13）	可能匹配。
19	\b	“”(索引 14)	匹配。

如果正则表达式模式中不包括可选限定符或替换构造，则将正则表达式模式与输入字符串匹配所需要的最大比较数大致等于输入字符串中的字符数。在这种情况下，正则表达式引擎通过 19 次比较来标识该 13 个字符的字符串中可能的匹配项。换句话说，如果正则表达式引擎不包含可选限定符或替换构造，则正则表达式引擎将以近线性时间运行。

使用可选限定符或替换构造的回溯

当正则表达式模式包含可选限定符或替换构造时，输入字符串的计算将不再为线性。使用 NFA 引擎的模式匹配由正则表达式中的语言元素驱动，而不是由输入字符串中要匹配的字符驱动。因此，正则表达式引擎将尝试完全匹配可选或可替换的子表达式。当它前进到子表达式中的下一个语言元素并且匹配不成功时，正则表达式引擎可放弃其成功匹配的一部分，并返回以前保存的与将正则表达式作为一个整体与输入字符串匹配有关的状态。返回到以前保存状态以查找匹配的这一过程称为回溯。

例如，考虑正则表达式模式 `.*(es)`，它匹配字符“es”以及它前面的所有字符。如下面的示例所示，如果输入字符串为“Essential services are provided by regular expressions.”，模式将匹配“expressions”之前且包括“es”在内的整个字符串。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "Essential services are provided by regular expressions.";
      string pattern = ".*(es)";
      Match m = Regex.Match(input, pattern, RegexOptions.IgnoreCase);
      if (m.Success) {
         Console.WriteLine("'{0}' found at position {1}",
                           m.Value, m.Index);
         Console.WriteLine("'es' found at position {0}",
                           m.Groups[1].Index);
      }
   }
}
//    'Essential services are provided by regular expres' found at position 0
//    'es' found at position 47
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "Essential services are provided by regular expressions."
        Dim pattern As String = ".*(es)"
        Dim m As Match = Regex.Match(input, pattern, RegexOptions.IgnoreCase)
        If m.Success Then
            Console.WriteLine("'{0}' found at position {1}", _
                              m.Value, m.Index)
            Console.WriteLine("'es' found at position {0}", _
                              m.Groups(1).Index)
        End If
    End Sub
End Module
'    'Essential services are provided by regular expres' found at position 0
'    'es' found at position 47
```

为此，正则表达式引擎按如下所示使用回溯：

- 它将 `.*` （它对应于出现零次、一次或多次任意字符）与整个输入字符串匹配。
- 它尝试在正则表达式模式中匹配“e”。但是，输入字符串没有剩余的可用字符来匹配。
- 它回溯到上一次成功的匹配“Essential services are provided by regular expressions”，并尝试将“e”与句尾的句号匹配。匹配失败。
- 它继续回溯到上一个成功匹配，一次一个字符，直至临时匹配的子字符串为“Essential services are provided by regular expr”。然后，它将模式中的“e”与“expressions”中的第二个“e”进行比较，并找到匹配。
- 它将模式中的“s”与匹配的“e”字符之后的“s”（“expressions”中的第一个“s”）进行比较。匹配成功。

当您使用回溯将正则表达式模式与输入字符串（长度为 55 个字符）匹配时，需要执行 67 次比较操作。通常，如果正则表达式模式包括单个替换构造或单个可选限定符，则匹配模式所需要的比较操作数大于输入字符串中字符数的两倍。

使用嵌套的可选限定符的回溯

如果模式中包括大量替换构造、嵌套的替换构造（或最常见的是嵌套的可选限定符），则匹配正则表达式模式所需要的比较操作数会成指数增加。例如，正则表达式模式 `^(a+)+$` 用于匹配包含一个或多个“a”字符的完整字符串。该示例提供了两个长度相同的输入字符串，但只有第一个字符串与模式匹配。
System.Diagnostics.Stopwatch 类用于确定匹配操作所需的时间。

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string pattern = "^(a+)+$";
      string[] inputs = { "aaaaaa", "aaaaa!" };
      Regex rgx = new Regex(pattern);
      Stopwatch sw;

      foreach (string input in inputs) {
         sw = Stopwatch.StartNew();
         Match match = rgx.Match(input);
         sw.Stop();
         if (match.Success)
            Console.WriteLine("Matched {0} in {1}", match.Value, sw.Elapsed);
         else
            Console.WriteLine("No match found in {0}", sw.Elapsed);
      }
   }
}
```

```
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim pattern As String = "^(a+)+$"
        Dim inputs() As String = {"aaaaaa", "aaaaa!"}
        Dim rgx As New Regex(pattern)
        Dim sw As Stopwatch

        For Each input As String In inputs
            sw = Stopwatch.StartNew()
            Dim match As Match = rgx.Match(input)
            sw.Stop()
            If match.Success Then
                Console.WriteLine("Matched {0} in {1}", match.Value, sw.Elapsed)
            Else
                Console.WriteLine("No match found in {0}", sw.Elapsed)
            End If
        Next
    End Sub
End Module
```

正如示例输出所示，正则表达式引擎查找输入字符串与模式不匹配所需的时间大约为标识匹配字符串所需时间的两倍。这是因为，不成功的匹配始终表示最糟糕的情况。正则表达式引擎必须使用正则表达式来遵循通过数据的所有可能路径，然后才能得出匹配不成功的结论，嵌套的括号会创建通过数据的许多其他路径。正则表达式引擎通过执行以下操作来确定第二个字符串与模式不匹配：

- 它检查到正位于字符串开头，然后将字符串中的前五个字符与模式 `a+` 匹配。然后确定字符串中没有其他成组的“a”字符。最后，它测试是否位于字符串结尾。由于还有一个附加字符保留在字符串中，所以匹配失败。这一失败的匹配需要进行 9 次比较。正则表达式引擎也从其“a”（我们将其称为匹配 1）、“aa”（匹配 2）、“aaa”（匹配 3）和“aaaa”（匹配 4）的匹配中保存状态信息。

- 它返回到以前保存的匹配 4。它确定没有一个附加的“a”字符可分配给其他捕获的组。最后，它测试是否位于字符串结尾。由于还有一个附加字符保留在字符串中，所以匹配失败。该失败的匹配需要进行 4 次比较。到目前为止，总共执行了 13 次比较。

- 它返回到以前保存的匹配 3。它确定有两个附加的"a"字符可分配给其他捕获的组。但是，字符串末尾测试失败。然后，它返回 match3 并尝试在两个附加的捕获组中匹配两个附加的"a"字符。字符串末尾测试仍失败。这些失败的匹配需要进行 12 次比较。到目前为止，总共执行了 25 次比较。

输入字符串与正则表达式的比较将以此方式继续，直到正则表达式引擎已尝试所有可能的匹配组合然后得出无匹配的结论。因为存在嵌套的限定符，所以此比较为 O(2^n) 或指数操作，其中 n 是输入字符串中的字符数 。这意味着在最糟糕的情况下，包含 30 个字符的输入字符串大约需要进行 1,073,741,824 次比较，包含 40 个字符的输入字符串大约需要进行 1,099,511,627,776 次比较。如果使用上述长度甚至更长的字符串，则正则表达式方法在处理与正则表达式模式不匹配的输入时，会需要超长的时间来完成。

控制回溯

通过回溯可以创建强大、灵活的正则表达式。但如上一节所示，回溯在提供这些优点的同时，可能也会使性能差的无法接受。若要防止过度回溯，则应在实例化 Regex 对象或调用静态正则表达式匹配方法时定义超时间隔。下一节中将对此进行讨论。此外，.NET 支持下面三个正则表达式语言元素，它们限制或禁止回溯、支持复杂的正则表达式，且不或几乎不损害性能：原子组、回顾后发断言和先行断言。有关每个语言元素的详细信息，请参见分组构造。

定义超时间隔

从 .NET Framework 4.5 开始，可以设置超时值，该值表示正则表达式引擎在放弃尝试并引发 RegexMatchTimeoutException 异常之前将搜索单个匹配项的最长间隔。你可以通过向实例正则表达式的 TimeSpan 构造函数提供 Regex(String, RegexOptions, TimeSpan) 值来指定超时间隔。此外，每种静态模式匹配方法都具有带 TimeSpan 参数的重载，该参数允许你指定超时值。默认情况下，超时间隔设置为 Regex.InfiniteMatchTimeout 且正则表达式引擎不会超时。

> **IMPORTANT**
> 如果正则表达式依赖回溯，建议你始终设置超时间隔。

RegexMatchTimeoutException 异常指示正则表达式引擎无法在指定的超时间隔内找到匹配项，但不指示引发异常的原因。原因可能是过度回溯，但也可能是超时间隔设置得过小(在引发异常时产生系统负载)。在处理异常时，你可以选择放弃与输入字符串的进一步匹配或增大超时间隔，然后重试匹配操作。

例如，下面的代码调用 Regex(String, RegexOptions, TimeSpan) 构造函数来实例化超时值为 1 秒的 Regex 对象。正则表达式模式 `(a+)+$` (与行尾的一个或多个"a"字符的一个或多个序列匹配)受过度回溯的约束。如果引发 RegexMatchTimeoutException 异常，该示例会将超时值增大到三秒最长间隔。之后，它放弃尝试匹配模式。

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Security;
using System.Text.RegularExpressions;
using System.Threading;

public class Example
{
   const int MaxTimeoutInSeconds = 3;

   public static void Main()
   {
      string pattern = @"(a+)+$";    // DO NOT REUSE THIS PATTERN.
      Regex rgx = new Regex(pattern, RegexOptions.IgnoreCase, TimeSpan.FromSeconds(1));
      Stopwatch sw = null;

      string[] inputs= { "aa", "aaaa>",
                         "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                         "aaaaaaaaaaaaaaaaaaaa>",
                         "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>" };
```

```
      foreach (var inputValue in inputs) {
         Console.WriteLine("Processing {0}", inputValue);
         bool timedOut = false;
         do {
            try {
               sw = Stopwatch.StartNew();
               // Display the result.
               if (rgx.IsMatch(inputValue)) {
                  sw.Stop();
                  Console.WriteLine(@"Valid: '{0}' ({1:ss\.fffffff} seconds)",
                                    inputValue, sw.Elapsed);
               }
               else {
                  sw.Stop();
                  Console.WriteLine(@"'{0}' is not a valid string. ({1:ss\.fffff} seconds)",
                                    inputValue, sw.Elapsed);
               }
            }
            catch (RegexMatchTimeoutException e) {
               sw.Stop();
               // Display the elapsed time until the exception.
               Console.WriteLine(@"Timeout with '{0}' after {1:ss\.fffff}",
                                 inputValue, sw.Elapsed);
               Thread.Sleep(1500);       // Pause for 1.5 seconds.

               // Increase the timeout interval and retry.
               TimeSpan timeout = e.MatchTimeout.Add(TimeSpan.FromSeconds(1));
               if (timeout.TotalSeconds > MaxTimeoutInSeconds) {
                  Console.WriteLine("Maximum timeout interval of {0} seconds exceeded.",
                                    MaxTimeoutInSeconds);
                  timedOut = false;
               }
               else {
                  Console.WriteLine("Changing the timeout interval to {0}",
                                    timeout);
                  rgx = new Regex(pattern, RegexOptions.IgnoreCase, timeout);
                  timedOut = true;
               }
            }
         } while (timedOut);
         Console.WriteLine();
      }
   }
}
// The example displays output like the following :
//    Processing aa
//    Valid: 'aa' (00.0000779 seconds)
//
//    Processing aaaa>
//    'aaaa>' is not a valid string. (00.00005 seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
//    Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043 seconds)
//
//    Processing aaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 01.00469
//    Changing the timeout interval to 00:00:02
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 02.01202
//    Changing the timeout interval to 00:00:03
//    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 03.01043
//    Maximum timeout interval of 3 seconds exceeded.
//
//    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
//    Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>' after 03.01018
//    Maximum timeout interval of 3 seconds exceeded.
```

```
Imports System.ComponentModel
Imports System.Diagnostics
Imports System.Security
Imports System.Text.RegularExpressions
Imports System.Threading

Module Example
    Const MaxTimeoutInSeconds As Integer = 3

    Public Sub Main()
        Dim pattern As String = "(a+)+$"    ' DO NOT REUSE THIS PATTERN.
        Dim rgx As New Regex(pattern, RegexOptions.IgnoreCase, TimeSpan.FromSeconds(1))
        Dim sw As Stopwatch = Nothing

        Dim inputs() As String = {"aa", "aaaa>",
                                  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
                                  "aaaaaaaaaaaaaaaaaaaa>",
                                  "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>"}

        For Each inputValue In inputs
            Console.WriteLine("Processing {0}", inputValue)
            Dim timedOut As Boolean = False
            Do
                Try
                    sw = Stopwatch.StartNew()
                    ' Display the result.
                    If rgx.IsMatch(inputValue) Then
                        sw.Stop()
                        Console.WriteLine("Valid: '{0}' ({1:ss\.fffffff} seconds)",
                                          inputValue, sw.Elapsed)
                    Else
                        sw.Stop()
                        Console.WriteLine("'{0}' is not a valid string. ({1:ss\.fffff} seconds)",
                                          inputValue, sw.Elapsed)
                    End If
                Catch e As RegexMatchTimeoutException
                    sw.Stop()
                    ' Display the elapsed time until the exception.
                    Console.WriteLine("Timeout with '{0}' after {1:ss\.fffff}",
                                      inputValue, sw.Elapsed)
                    Thread.Sleep(1500)       ' Pause for 1.5 seconds.

                    ' Increase the timeout interval and retry.
                    Dim timeout As TimeSpan = e.MatchTimeout.Add(TimeSpan.FromSeconds(1))
                    If timeout.TotalSeconds > MaxTimeoutInSeconds Then
                        Console.WriteLine("Maximum timeout interval of {0} seconds exceeded.",
                                          MaxTimeoutInSeconds)
                        timedOut = False
                    Else
                        Console.WriteLine("Changing the timeout interval to {0}",
                                          timeout)
                        rgx = New Regex(pattern, RegexOptions.IgnoreCase, timeout)
                        timedOut = True
                    End If
                End Try
            Loop While timedOut
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays output like the following:
'    Processing aa
'    Valid: 'aa' (00.0000779 seconds)
'
'    Processing aaaa>
'    'aaaa>' is not a valid string. (00.00005 seconds)
'
'    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
'    Valid: 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa' (00.0000043 seconds)
```

```
'
'    Processing aaaaaaaaaaaaaaaaaaaa>
'    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 01.00469
'    Changing the timeout interval to 00:00:02
'    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 02.01202
'    Changing the timeout interval to 00:00:03
'    Timeout with 'aaaaaaaaaaaaaaaaaaaa>' after 03.01043
'    Maximum timeout interval of 3 seconds exceeded.
'
'    Processing aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>
'    Timeout with 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa>' after 03.01018
'    Maximum timeout interval of 3 seconds exceeded.
```

原子组

`(?>` subexpression `)` 语言元素禁止在子表达式中使用回溯 。成功匹配后，它不会将其匹配项的任何部分提供给后续回溯。例如，在模式 `(?>\w*\d*)1` 中，如果无法匹配 `1`，则即使意味着会允许 `1` 成功匹配，`\d*` 也不会放弃其任何匹配项。原子组可帮助防止与失败匹配关联的性能问题。

下面的示例演示在使用嵌套的限定符时禁止回溯如何改进性能。它测量正则表达式引擎确定输入字符串与两个正则表达式不匹配所需要的时间。第一个正则表达式使用回溯尝试匹配一个字符串，在该字符串中，一个或多个十六进制数出现了一次或多次，然后依次为冒号、一个或多个十六进制数、两个冒号。第二个正则表达式与第一个相同，不同之处是它禁用了回溯。如该示例输出所示，禁用回溯对性能的改进非常显著。

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:";
      bool matched;
      Stopwatch sw;

      Console.WriteLine("With backtracking:");
      string backPattern = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*(::)$";
      sw = Stopwatch.StartNew();
      matched = Regex.IsMatch(input, backPattern);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, backPattern), sw.Elapsed);
      Console.WriteLine();

      Console.WriteLine("Without backtracking:");
      string noBackPattern = "^((?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]{1,4}))*(::)$";
      sw = Stopwatch.StartNew();
      matched = Regex.IsMatch(input, noBackPattern);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, noBackPattern), sw.Elapsed);
   }
}
// The example displays output like the following:
//       With backtracking:
//       Match: False in 00:00:27.4282019
//
//       Without backtracking:
//       Match: False in 00:00:00.0001391
```

```
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "b51:4:1DB:9EE1:5:27d60:f44:D4:cd:E:5:0A5:4a:D24:41Ad:"
        Dim matched As Boolean
        Dim sw As Stopwatch

        Console.WriteLine("With backtracking:")
        Dim backPattern As String = "^(([0-9a-fA-F]{1,4}:)*([0-9a-fA-F]{1,4}))*(::)$"
        sw = Stopwatch.StartNew()
        matched = Regex.IsMatch(input, backPattern)
        sw.Stop()
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, backPattern), sw.Elapsed)
        Console.WriteLine()

        Console.WriteLine("Without backtracking:")
        Dim noBackPattern As String = "^((?>[0-9a-fA-F]{1,4}:)*(?>[0-9a-fA-F]{1,4}))*(::)$"
        sw = Stopwatch.StartNew()
        matched = Regex.IsMatch(input, noBackPattern)
        sw.Stop()
        Console.WriteLine("Match: {0} in {1}", Regex.IsMatch(input, noBackPattern), sw.Elapsed)
    End Sub
End Module
' The example displays the following output:
'       With backtracking:
'       Match: False in 00:00:27.4282019
'
'       Without backtracking:
'       Match: False in 00:00:00.0001391
```

回顾断言

.NET 包括两个语言元素（`(?<=` subexpression `)` 和 `(?<!` subexpression `)`），它们与输入字符串中之前的一个或多个字符匹配。这两个语言元素都是零宽度断言；也就是说，它们通过 *subexpression* 而不是前移或回溯来确定当前字符之前紧挨着的一个或多个字符是否匹配。

`(?<=` subexpression `)` 是正回顾断言；也就是说，当前位置之前的一个或多个字符必须与 subexpression 匹配 。`(?<!` *subexpression* `)` 是负回顾断言；也就是说，当前位置之前的一个或多个字符不得与 *subexpression* 匹配。当 *subexpression* 为前一个子表达式的子集时，正回顾断言和负回顾断言都最为有用。

下面的示例使用两个相当的正则表达式模式，验证电子邮件地址中的用户名。第一个模式由于过多使用回溯，性能极差。第二个模式通过将嵌套的限定符替换为正回顾断言来修改第一个正则表达式。该示例的输出显示 Regex.IsMatch 方法的执行时间。

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      Stopwatch sw;
      string input = "test@contoso.com";
      bool result;

      string pattern = @"^[0-9A-Z]([-.\w]*[0-9A-Z])?@";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed);

      string behindPattern = @"^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, behindPattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("Match with Lookbehind: {0} in {1}", result, sw.Elapsed);
   }
}
// The example displays output similar to the following:
//       Match: True in 00:00:00.0017549
//       Match with Lookbehind: True in 00:00:00.0000659
```

```
Module Example
    Public Sub Main()
        Dim sw As Stopwatch
        Dim input As String = "test@contoso.com"
        Dim result As Boolean

        Dim pattern As String = "^[0-9A-Z]([-.\w]*[0-9A-Z])?@"
        sw = Stopwatch.StartNew()
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase)
        sw.Stop()
        Console.WriteLine("Match: {0} in {1}", result, sw.Elapsed)

        Dim behindPattern As String = "^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@"
        sw = Stopwatch.StartNew()
        result = Regex.IsMatch(input, behindPattern, RegexOptions.IgnoreCase)
        sw.Stop()
        Console.WriteLine("Match with Lookbehind: {0} in {1}", result, sw.Elapsed)
    End Sub
End Module
' The example displays output similar to the following:
'       Match: True in 00:00:00.0017549
'       Match with Lookbehind: True in 00:00:00.0000659
```

第一个正则表达式模式 `^[0-9A-Z]([-.\w]*[0-9A-Z])*@` 的定义如下表所示。

`^`	从字符串开头开始匹配。
`[0-9A-Z]`	匹配字母数字字符。因为 Regex.IsMatch 方法是使用 RegexOptions.IgnoreCase 选项调用的，所以此比较不区分大小写。

“	“
`[-.\w]*`	匹配零个、一个或多个连字符、句号或单词字符。
`[0-9A-Z]`	匹配字母数字字符。
`([-.\w]*[0-9A-Z])*`	匹配以下零个或多个事例：即零个或多个连字符、句号或单词字符后跟一个字母数字字符的组合。这是第一个捕获组。
`@`	匹配 at 符号(“@”)。

第二个正则表达式模式 `^[0-9A-Z][-.\w]*(?<=[0-9A-Z])@` 使用正回顾断言。其定义如下表所示。

“	“
`^`	从字符串开头开始匹配。
`[0-9A-Z]`	匹配字母数字字符。因为 Regex.IsMatch 方法是使用 RegexOptions.IgnoreCase 选项调用的，所以此比较不区分大小写。
`[-.\w]*`	匹配零个或多个连字符、句号或单词字符。
`(?<=[0-9A-Z])`	回顾最后一个匹配的字符，如果该字符是字母数字字符，则继续匹配。请注意，字母数字字符是由句号、连字符和所有单词字符构成的集合的子集。
`@`	匹配 at 符号(“@”)。

预测先行断言

.NET 包括两个语言元素(`(?=` subexpression `)` 和 `(?!` subexpression `)`)，它们与输入字符串中接下来的一个或多个字符匹配。这两个语言元素都是零宽度断言；也就是说，它们通过 *subexpression* 而不是前移或回溯来确定当前字符之后紧挨着的一个或多个字符是否匹配。

`(?=` subexpression `)` 是正预测先行断言；也就是说，当前位置之后的一个或多个字符必须与 subexpression 匹配。`(?!` *subexpression* `)` 是负预测先行断言；也就是说，当前位置之后的一个或多个字符不得与 *subexpression* 匹配。当 *subexpression* 为下一个子表达式的子集时，正预测先行断言和负预测先行断言都最为有用。

下面的示例使用两个可验证完全限定的类型名称的等效正则表达式模式。第一个模式由于过多使用回溯，性能极差。第二个模式通过将嵌套的限定符替换为正预测先行断言来修改第一个正则表达式。该示例的输出显示 Regex.IsMatch 方法的执行时间。

```
using System;
using System.Diagnostics;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string input = "aaaaaaaaaaaaaaaaaaaaa.";
      bool result;
      Stopwatch sw;

      string pattern = @"^(([A-Z]\w*)+\.)*[A-Z]\w*$";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("{0} in {1}", result, sw.Elapsed);

      string aheadPattern = @"^((?=[A-Z])\w+\.)*[A-Z]\w*$";
      sw = Stopwatch.StartNew();
      result = Regex.IsMatch(input, aheadPattern, RegexOptions.IgnoreCase);
      sw.Stop();
      Console.WriteLine("{0} in {1}", result, sw.Elapsed);
   }
}
// The example displays the following output:
//       False in 00:00:03.8003793
//       False in 00:00:00.0000866
```

```
Imports System.Diagnostics
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim input As String = "aaaaaaaaaaaaaaaaaaaaa."
        Dim result As Boolean
        Dim sw As Stopwatch

        Dim pattern As String = "^(([A-Z]\w*)+\.)*[A-Z]\w*$"
        sw = Stopwatch.StartNew()
        result = Regex.IsMatch(input, pattern, RegexOptions.IgnoreCase)
        sw.Stop()
        Console.WriteLine("{0} in {1}", result, sw.Elapsed)

        Dim aheadPattern As String = "^((?=[A-Z])\w+\.)*[A-Z]\w*$"
        sw = Stopwatch.StartNew()
        result = Regex.IsMatch(input, aheadPattern, RegexOptions.IgnoreCase)
        sw.Stop()
        Console.WriteLine("{0} in {1}", result, sw.Elapsed)
    End Sub
End Module
' The example displays the following output:
'       False in 00:00:03.8003793
'       False in 00:00:00.0000866
```

第一个正则表达式模式 `^(([A-Z]\w*)+\.)*[A-Z]\w*$` 的定义如下表所示。

“	“
`^`	从字符串开头开始匹配。

“	“
`([A-Z]\w*)+\.`	对后跟零个或多个单词字符、句点的字母字符 (A-Z) 匹配一次或多次。因为 Regex.IsMatch 方法是使用 RegexOptions.IgnoreCase 选项调用的，所以此比较不区分大小写。
`(([A-Z]\w*)+\.)*`	对前一个模式匹配零次或多次。
`[A-Z]\w*`	匹配后跟零个或多个单词字符的字母字符。
`$`	在输入字符串末尾结束匹配。

第二个正则表达式模式 `^((?=[A-Z])\w+\.)*[A-Z]\w*$` 使用正预测先行断言。其定义如下表所示。

“	“
`^`	从字符串开头开始匹配。
`(?=[A-Z])`	预测先行到第一个字符，如果它是字母 (A-Z)，则继续匹配。因为 Regex.IsMatch 方法是使用 RegexOptions.IgnoreCase 选项调用的，所以此比较不区分大小写。
`\w+\.`	匹配后跟句号的一个或多个单词字符。
`((?=[A-Z])\w+\.)*`	对一个或多个单词字符后跟句号的模式进行一次或多次匹配。初始单词字符必须为字母字符。
`[A-Z]\w*`	匹配后跟零个或多个单词字符的字母字符。
`$`	在输入字符串末尾结束匹配。

请参阅

- .NET 正则表达式
- 正则表达式语言 - 快速参考
- 数量词
- 替换构造
- 分组构造

正则表达式中的编译和重复使用

2021/11/16 •

通过了解正则表达式引擎编译表达式的方式以及正则表达式的缓存方式，可以优化大量使用正则表达式的应用程序的性能。本主题介绍编译和缓存。

已编译的正则表达式

默认情况下，正则表达式引擎将正则表达式编译成内部指令序列（这些指令序列是不同于 Microsoft 中间语言 (MSIL) 的高级代码）。当引擎执行正则表达式时，会解释内部代码。

如果 Regex 对象是通过 RegexOptions.Compiled 选项构造而成，它会将正则表达式编译为显式 MSIL 代码，而不是高级正则表达式内部指令。这样，.NET 的实时 (JIT) 编译器便可以将表达式转换为本机代码以获得更高的性能。构造 Regex 对象的成本可能会更高，但执行其匹配项的开销可能会小得多。

替换方法是，使用预编译正则表达式。可以使用 CompileToAssembly 方法，将所有表达式都编译到可重用的 DLL 中。这样一来，就无需在运行时编译，同时还仍受益于已编译正则表达式的速度优势。

正则表达式缓存

为了提高性能，正则表达式引擎为已编译的正则表达式维护了一个应用程序范围的缓存。该缓存只存储静态方法调用中使用的正则表达式模式。（不缓存提供给实例方法的正则表达式模式。）这样，在每次使用正则表达式时，就无需将正则表达式重新分析成高级字节代码。

缓存正则表达式数上限由 `static`（Visual Basic 中的 `Shared`）Regex.CacheSize 属性的值决定。默认情况下，正则表达式引擎最多可缓存 15 个已编译的正则表达式。如果已编译正则表达式的数目超过缓存大小，则丢弃最早使用的正则表达式并缓存新的正则表达式。

应用程序可通过以下两种方式之一来重用正则表达式：

- 使用 Regex 对象的静态方法定义正则表达式。如果要使用的正则表达式模式已由其他静态方法调用定义，则正则表达式引擎将尝试从缓存中检索该模式。如果它在缓存中不可用，则引擎将编译正则表达式并将其添加到缓存中。
- 重用现有 Regex 对象（只要需要使用正则表达式模式）。

鉴于对象实例化和正则表达式编译产生的开销，因此创建并迅速销毁大量 Regex 对象的进程成本非常高。对于使用大量不同正则表达式的应用，可以调用静态方法 `Regex`，并尽量增加正则表达式缓存大小，从而优化性能。

请参阅

- .NET 正则表达式

正则表达式中的线程安全

2021/11/16 •

Regex 类本身是线程安全且不可变的(只读)。也就是说，可以在任何线程上创建 Regex 对象并在线程间共享；可以从任何线程调用匹配方法并且始终不会更改全局状态。

不过，应对一个线程使用 Regex 返回的结果对象(Match 和 MatchCollection)。尽管其中许多对象在逻辑上是不可变的，但其实现可以延迟某些结果的计算以提高性能，因此，调用方必须序列化对这些对象的访问。

如果需要在多个线程上共享 Regex 结果对象，则通过调用对象的同步方法，可以将这些对象转换成线程安全的实例。除枚举器外，所有正则表达式类都是线程安全的或者可以通过同步方法转换成线程安全对象。

枚举器是唯一例外。应用程序必须序列化对集合枚举器的调用。规则为，如果可以在多个线程上同时枚举一个集合，则应该同步枚举器所遍历集合的根对象上的枚举器方法。

另请参阅

- .NET 正则表达式

正则表达式示例：扫描 HREF

2021/11/16 •

下面的示例搜索输入字符串并显示所有 href="..." 的值和它们在字符串中的位置。

WARNING

如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

Regex 对象

因为可以通过用户代码多次调用 `DumpHRefs` 方法，所以它使用 `static`（Visual Basic 中的 `Shared`）Regex.Match(String, String, RegexOptions) 方法。这样一来，正则表达式引擎不仅可以缓存正则表达式，还杜绝了每次调用方法时实例化新 Regex 对象产生的开销。随后使用 Match 对象循环访问字符串中的所有匹配。

```
private static void DumpHRefs(string inputString)
{
    Match m;
    string HRefPattern = @"href\s*=\s*(?:[""']?(?<1>[^""']*)[""']?|(?<1>\S+))";

    try
    {
        m = Regex.Match(inputString, HRefPattern,
                        RegexOptions.IgnoreCase | RegexOptions.Compiled,
                        TimeSpan.FromSeconds(1));
        while (m.Success)
        {
            Console.WriteLine("Found href " + m.Groups[1] + " at "
               + m.Groups[1].Index);
            m = m.NextMatch();
        }
    }
    catch (RegexMatchTimeoutException)
    {
        Console.WriteLine("The matching operation timed out.");
    }
}
```

```
Private Sub DumpHRefs(inputString As String)
    Dim m As Match
    Dim HRefPattern As String = "href\s*=\s*(?:[""']?(?<1>[^""']*)[""']?|(?<1>\S+))"

    Try
        m = Regex.Match(inputString, HRefPattern, _
                        RegexOptions.IgnoreCase Or RegexOptions.Compiled,
                        TimeSpan.FromSeconds(1))
        Do While m.Success
            Console.WriteLine("Found href {0} at {1}.", _
                              m.Groups(1), m.Groups(1).Index)
            m = m.NextMatch()
        Loop
    Catch e As RegexMatchTimeoutException
        Console.WriteLine("The matching operation timed out.")
    End Try
End Sub
```

下面的示例演示对 `DumpHRefs` 方法的调用。

```
public static void Main()
{
    string inputString = "My favorite web sites include:</P>" +
                         "<A HREF=\"http://msdn2.microsoft.com\">" +
                         "MSDN Home Page</A></P>" +
                         "<A HREF=\"http://www.microsoft.com\">" +
                         "Microsoft Corporation Home Page</A></P>" +
                         "<A HREF=\"http://blogs.msdn.com/bclteam\">" +
                         ".NET Base Class Library blog</A></P>";
    DumpHRefs(inputString);
}
// The example displays the following output:
//       Found href http://msdn2.microsoft.com at 43
//       Found href http://www.microsoft.com at 102
//       Found href http://blogs.msdn.com/bclteam at 176
```

```
Public Sub Main()
    Dim inputString As String = "My favorite web sites include:</P>" & _
                                "<A HREF=""http://msdn2.microsoft.com"">" & _
                                "MSDN Home Page</A></P>" & _
                                "<A HREF=""http://www.microsoft.com"">" & _
                                "Microsoft Corporation Home Page</A></P>" & _
                                "<A HREF=""http://blogs.msdn.com/bclteam"">" & _
                                ".NET Base Class Library blog</A></P>"
    DumpHRefs(inputString)
End Sub
' The example displays the following output:
'       Found href http://msdn2.microsoft.com at 43
'       Found href http://www.microsoft.com at 102
'       Found href http://blogs.msdn.com/bclteam/) at 176
```

正则表达式模式 `href\s*=\s*(?:["']?(?<1>[^"']*)["']?|(?<1>\S+))` 的含义如下表所示。

“	“
`href`	匹配文本字符串“href”。匹配不区分大小写。
`\s*`	匹配零个或多个空白字符。
`=`	匹配等于号。

"	"
`\s*`	匹配零个或多个空白字符。
`(?:\["'\](?<1>\[^"'\]*)["']|(?<1>\S+))`	匹配以下项之一，而不将结果分配到捕获组： • 一个引号或单引号，后跟零个或多个引号或单引号以外的任意字符，然后再后跟一个引号或单引号。名为 `1` 的组包含在此模式。 • 一个或多个非空格字符。名为 `1` 的组包含在此模式。
`(?<1>[^"']*)`	将零个或多个引号或单引号以外的任意字符分配给名为 `1` 的捕获组。
`(?<1>\S+)`	将一个或多个非空白字符分配给名为 `1` 的捕获组。

匹配结果类

搜索结果存储在 Match 类中，此类可访问搜索提取的所有子字符串。它还会记住搜索的字符串和使用的正则表达式，因此可以调用 Match.NextMatch 方法，从上一次搜索结束的位置开始执行另一次搜索。

显式命名的捕获

在传统正则表达式中，捕获圆括号会自动按顺序编号。这会导致两个问题。首先，如果通过插入或删除一组圆括号修改正则表达式，则必须重新编写引用带编号捕获的所有代码才能反映新编号。其次，由于不同的圆括号组通常用于为可接受的匹配项提供两个替代表达式，则可能难以确定这两个表达式中的哪个表达式实际返回了结果。

为了解决这些问题，Regex 类支持语法 `(?<name>…)`，以便将匹配捕获到指定槽中（可以使用字符串或整数命名槽；如果使用整数命名，可以更快地召回）。因此，相同字符串的替代匹配全都可以定向到相同位置。发生冲突时，放入槽中的最后一个匹配项是成功的匹配项。（但是，提供了适用于单个槽的多个匹配项的完整列表。有关详细信息，请参阅 Group.Captures 集合。）

请参阅

- .NET 正则表达式

正则表达式示例：更改日期格式

2021/11/16 •

下面的代码示例使用 Regex.Replace 方法，将格式为 mm /dd /yy 的日期替换为格式为 dd -mm -yy 的日期。

> **WARNING**
> 如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

示例

```
static string MDYToDMY(string input)
{
   try {
      return Regex.Replace(input,
             @"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b",
            "${day}-${month}-${year}", RegexOptions.None,
            TimeSpan.FromMilliseconds(150));
   }
   catch (RegexMatchTimeoutException) {
      return input;
   }
}
```

```
Function MDYToDMY(input As String) As String
    Try
        Return Regex.Replace(input, _
               "\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b", _
               "${day}-${month}-${year}", RegexOptions.None,
               TimeSpan.FromMilliseconds(150))
    Catch e As RegexMatchTimeoutException
        Return input
    End Try
End Function
```

下面的代码演示如何在应用程序中调用 `MDYToDMY` 方法。

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

public class Class1
{
   public static void Main()
   {
      string dateString = DateTime.Today.ToString("d",
                                       DateTimeFormatInfo.InvariantInfo);
      string resultString = MDYToDMY(dateString);
      Console.WriteLine("Converted {0} to {1}.", dateString, resultString);
   }

   static string MDYToDMY(string input)
   {
      try {
         return Regex.Replace(input,
                @"\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b",
               "${day}-${month}-${year}", RegexOptions.None,
               TimeSpan.FromMilliseconds(150));
      }
      catch (RegexMatchTimeoutException) {
         return input;
      }
   }
}
// The example displays the following output to the console if run on 8/21/2007:
//      Converted 08/21/2007 to 21-08-2007.
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Module DateFormatReplacement
    Public Sub Main()
        Dim dateString As String = Date.Today.ToString("d", _
                                           DateTimeFormatInfo.InvariantInfo)
        Dim resultString As String = MDYToDMY(dateString)
        Console.WriteLine("Converted {0} to {1}.", dateString, resultString)
    End Sub

    Function MDYToDMY(input As String) As String
        Try
            Return Regex.Replace(input, _
                   "\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b", _
                   "${day}-${month}-${year}", RegexOptions.None,
                   TimeSpan.FromMilliseconds(150))
        Catch e As RegexMatchTimeoutException
            Return input
        End Try
    End Function
End Module
' The example displays the following output to the console if run on 8/21/2007:
'      Converted 08/21/2007 to 21-08-2007.
```

注释

正则表达式模式 `\b(?<month>\d{1,2})/(?<day>\d{1,2})/(?<year>\d{2,4})\b` 的释义如下表所示。

“	“
`\b`	在单词边界处开始匹配。

“	“
`(?<month>\d{1,2})`	匹配一个或两个十进制数字。这是 `month` 捕获的组。
`/`	匹配斜杠标记。
`(?<day>\d{1,2})`	匹配一个或两个十进制数字。这是 `day` 捕获的组。
`/`	匹配斜杠标记。
`(?<year>\d{2,4})`	匹配两个到四个十进制数。这是 `year` 捕获的组。
`\b`	在单词边界处结束匹配。

模式 `${day}-${month}-${year}` 如下表所示定义替换字符串。

“	“
`$(day)`	添加由 `day` 捕获组捕获的字符串。
`-`	添加连字符。
`$(month)`	添加由 `month` 捕获组捕获的字符串。
`-`	添加连字符。
`$(year)`	添加由 `year` 捕获组捕获的字符串。

另请参阅

- .NET 正则表达式

如何：从 URL 中提取协议和端口号

2021/11/16 •

下面的示例从 URL 中提取协议和端口号。

> **WARNING**
> 如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

示例

此示例使用 Match.Result 方法返回协议，后面依次跟的是冒号和端口号。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
   public static void Main()
   {
      string url = "http://www.contoso.com:8080/letters/readme.html";

      Regex r = new Regex(@"^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
                          RegexOptions.None, TimeSpan.FromMilliseconds(150));
      Match m = r.Match(url);
      if (m.Success)
         Console.WriteLine(m.Result("${proto}${port}"));
   }
}
// The example displays the following output:
//       http:8080
```

```
Imports System.Text.RegularExpressions

Module Example
    Public Sub Main()
        Dim url As String = "http://www.contoso.com:8080/letters/readme.html"
        Dim r As New Regex("^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/",
                           RegexOptions.None, TimeSpan.FromMilliseconds(150))

        Dim m As Match = r.Match(url)
        If m.Success Then
            Console.WriteLine(m.Result("${proto}${port}"))
        End If
    End Sub
End Module
' The example displays the following output:
'       http:8080
```

正则表达式模式 `^(?<proto>\w+)://[^/]+?(?<port>:\d+)?/` 可按下表中的方式解释。

“	“
`^`	从字符串的开头部分开始匹配。
`(?<proto>\w+)`	匹配一个或多个单词字符。将此组命名为 `proto`。
`://`	匹配后跟两个正斜线的冒号。
`[^/]+?`	匹配正斜线以外的任何字符的一次或多次出现(但尽可能少)。
`(?<port>:\d+)?`	匹配后跟一个或多个数字字符的冒号的零次或一次出现。将此组命名为 `port`。
`/`	匹配正斜线。

Match.Result 方法扩展 `${proto}${port}` 替换序列，以连接在正则表达式模式中捕获的两个命名组的值。便捷的替换方法是，显式连接从 Match.Groups 属性返回的集合对象检索到的字符串。

此示例使用有两处替换(`${proto}` 和 `${port}`)的 Match.Result 方法，在输出字符串中添加捕获组。可以改为从匹配的 GroupCollection 对象检索捕获组，如下面的代码所示。

```
Console.WriteLine(m.Groups["proto"].Value + m.Groups["port"].Value);
```

```
Console.WriteLine(m.Groups("proto").Value + m.Groups("port").Value)
```

另请参阅

- .NET 正则表达式

如何：从字符串中剥离无效字符

2021/11/16 •

下面的示例使用静态 Regex.Replace 方法，从字符串中剥离无效字符。

> **WARNING**
> 如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

示例

可以使用此示例中定义的 `CleanInput` 方法来剥离在接受用户输入的文本字段中输入的可能有害的字符。在此情况下，`CleanInput` 会剥离所有非字母数字字符（句点 (.)、at 符号 (@) 和连字符 (-) 除外），并返回剩余字符串。但是，可以修改正则表达式模式，使其剥离不应包含在输入字符串内的所有字符。

```
using System;
using System.Text.RegularExpressions;

public class Example
{
    static string CleanInput(string strIn)
    {
        // Replace invalid characters with empty strings.
        try {
           return Regex.Replace(strIn, @"[^\w\.@-]", "",
                                RegexOptions.None, TimeSpan.FromSeconds(1.5));
        }
        // If we timeout when replacing invalid characters,
        // we should return Empty.
        catch (RegexMatchTimeoutException) {
           return String.Empty;
        }
    }
}
```

```
Imports System.Text.RegularExpressions

Module Example
    Function CleanInput(strIn As String) As String
        ' Replace invalid characters with empty strings.
        Try
            Return Regex.Replace(strIn, "[^\w\.@-]", "")
            ' If we timeout when replacing invalid characters,
            ' we should return String.Empty.
        Catch e As RegexMatchTimeoutException
            Return String.Empty
        End Try
    End Function
End Module
```

正则表达式模式 `[^\w\.@-]` 与非单词字符、句点、@ 符号或连字符的任何字符相匹配。单词字符可以是任何字母、十进制数字或标点连接符（如下划线符号）。与此模式匹配的任何字符被替换为 String.Empty（即替换模式定

义的字符串）。若要允许用户输入中出现其他字符，请将该字符添加到正则表达式模式中的字符类。例如，正则表达式模式 `[^\w\.@-\\%]` 还允许输入字符串中包含百分号和反斜杠。

请参阅

- .NET 正则表达式

如何确认字符串是有效的电子邮件格式

2021/11/16 •

下面的示例使用正则表达式来验证一个字符串是否为有效的电子邮件格式。

此正则表达式相对比实际上可用作电子邮件的表达式简单。使用正则表达式来验证电子邮件，这对于确保电子邮件的结构正确是很有用的，但这不是验证电子邮件是否实际存在的替代方法。

✔ 请使用小型正则表达式来检查电子邮件的有效结构。

✔ 请将测试电子邮件发送到应用用户提供的地址。

✘ 请勿将正则表达式用作验证电子邮件的唯一方法。

如果尝试创建完美的正则表达式来验证电子邮件的结构是否正确，表达式会变得非常复杂，以至于很难进行调试或改进。即使电子邮件的结构正确，正则表达式也无法验证其是否存在。验证电子邮件的最佳方法是将测试电子邮件发送到该地址。

WARNING

如果使用 System.Text.RegularExpressions 处理不受信任的输入，则传递一个超时。恶意用户可能会向 `RegularExpressions` 提供输入，从而导致拒绝服务攻击。使用 `RegularExpressions` 的 ASP.NET Core 框架 API 会传递一个超时。

示例

该示例定义 `IsValidEmail` 方法，如果字符串包含有效的电子邮件地址，则该方法返回 `true`；如果字符串不包含有效的电子邮件地址，但没有采取其他任何操作，该方法返回 `false`。

若要验证电子邮件地址是否有效，方法 `IsValidEmail` 将使用 Regex.Replace(String, String, MatchEvaluator) 正则表达式模式调用 `(@)(.+)$` 方法将域名从电子邮件地址分离。第三个参数是表示了处理和替换匹配文本的方法的 MatchEvaluator 委托。正则表达式模式的解释如下。

"	"
`(@)`	匹配 @ 字符。此部分是第一个捕获组。
`(.+)`	匹配任意字符的一个或多个匹配项。此部分是第二个捕获组。
`$`	在字符串的结尾结束匹配。

使用 @ 字符的域名已传递给 `DomainMapper` 方法，该方法使用 IdnMapping 类将 US-ASCII 字符范围外的 Unicode 字符转换为 Punycode。如果 `invalid` 方法在域名中检测到任何无效字符，该方法还会将 `True` 标志设置为 IdnMapping.GetAscii 。该方法将冠以 @ 符号的 Punycode 域名返回给 `IsValidEmail` 方法。

TIP

建议使用简单的 `(@)(.+)$` 正则表达式模式对域进行规范化，然后返回一个值，指示该域是通过还是失败。但本文中的示例介绍如何进一步使用正则表达式来验证电子邮件。无论验证电子邮件的方式如何，你都应始终将测试电子邮件发送到该地址，以确保其存在。

然后 `IsValidEmail` 方法调用 Regex.IsMatch(String, String) 方法验证该地址是否符合正则表达式模式。

`IsValidEmail` 方法只能确定电子邮件地址的格式是否有效，而不能验证电子邮件是否存在。同样，`IsValidEmail` 方法不会验证顶级域名是否是 IANA 根区域数据库上列出的有效域名，这需执行查找操作。

```
using System;
using System.Globalization;
using System.Text.RegularExpressions;

namespace RegexExamples
{
    class RegexUtilities
    {
        public static bool IsValidEmail(string email)
        {
            if (string.IsNullOrWhiteSpace(email))
                return false;

            try
            {
                // Normalize the domain
                email = Regex.Replace(email, @"(@)(.+)$", DomainMapper,
                                      RegexOptions.None, TimeSpan.FromMilliseconds(200));

                // Examines the domain part of the email and normalizes it.
                string DomainMapper(Match match)
                {
                    // Use IdnMapping class to convert Unicode domain names.
                    var idn = new IdnMapping();

                    // Pull out and process domain name (throws ArgumentException on invalid)
                    string domainName = idn.GetAscii(match.Groups[2].Value);

                    return match.Groups[1].Value + domainName;
                }
            }
            catch (RegexMatchTimeoutException e)
            {
                return false;
            }
            catch (ArgumentException e)
            {
                return false;
            }

            try
            {
                return Regex.IsMatch(email,
                    @"^[^@\s]+@[^@\s]+\.[^@\s]+$",
                    RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250));
            }
            catch (RegexMatchTimeoutException)
            {
                return false;
            }
        }
    }
}
```

```
Imports System.Globalization
Imports System.Text.RegularExpressions

Public Class RegexUtilities
    Public Shared Function IsValidEmail(email As String) As Boolean

        If String.IsNullOrWhiteSpace(email) Then Return False

        ' Use IdnMapping class to convert Unicode domain names.
        Try
            'Examines the domain part of the email and normalizes it.
            Dim DomainMapper =
                Function(match As Match) As String

                    'Use IdnMapping class to convert Unicode domain names.
                    Dim idn = New IdnMapping

                    'Pull out and process domain name (throws ArgumentException on invalid)
                    Dim domainName As String = idn.GetAscii(match.Groups(2).Value)

                    Return match.Groups(1).Value & domainName

                End Function

            'Normalize the domain
            email = Regex.Replace(email, "(@)(.+)$", DomainMapper,
                                  RegexOptions.None, TimeSpan.FromMilliseconds(200))

        Catch e As RegexMatchTimeoutException
            Return False

        Catch e As ArgumentException
            Return False

        End Try

        Try
            Return Regex.IsMatch(email,
                                 "^[^@\s]+@[^@\s]+\.[^@\s]+$",
                                 RegexOptions.IgnoreCase, TimeSpan.FromMilliseconds(250))

        Catch e As RegexMatchTimeoutException
            Return False

        End Try

    End Function
End Class
```

在本例中，正则表达式模式 `^[^@\s]+@[^@\s]+\.[^@\s]+$` 按下表中的方式解释。使用 RegexOptions.IgnoreCase 标志编译正则表达式。

"	"
`^`	从字符串的开头部分开始匹配。
`[^@\s]+`	匹配一次或多次出现的任何字符，@ 字符或空格除外。
`@`	匹配 @ 字符。
`[^@\s]+`	匹配一次或多次出现的任何字符，@ 字符或空格除外。

"	"
`\.`	匹配一个句点字符。
`[^@\s]+`	匹配一次或多次出现的任何字符，@ 字符或空格除外。
`$`	在字符串的结尾结束匹配。

> **IMPORTANT**
>
> 此正则表达式没有涵盖有效电子邮件地址的所有字符。它作为示例提供，你可以按需对其进行扩展。

请参阅

- .NET 正则表达式
- 对电子邮件地址进行验证的频率应为多久一次？

.NET 中的序列化

2021/11/16 •

序列化是将对象状态转换为可保持或传输的形式的过程。序列化的补集是反序列化，后者将流转换为对象。这两个过程一起保证能够存储和传输数据。

.NET 具有以下序列化技术：

- 二进制序列化保持类型保真，这对于多次调用应用程序时保持对象状态非常有用。例如，通过将对象序列化到剪贴板，可在不同的应用程序之间共享对象。您可以将对象序列化到流、磁盘、内存和网络等。远程处理使用序列化，“按值”在计算机或应用程序域之间传递对象。
- XML 和 SOAP 序列化只序列化公共属性和字段，并且不保持类型保真。当您希望提供或使用数据而不限制使用该数据的应用程序时，这一点非常有用。由于 XML 是开放式的标准，因此它对于通过 Web 共享数据来说是一个理想选择。SOAP 同样是开放式的标准，这使它也成为一个理想选择。
- JSON 序列化只序列化公共属性，并且不保持类型保真。JSON 是开放式的标准，对于通过 Web 共享数据来说是一个理想选择。

参考

System.Runtime.Serialization
包含可用于序列化和反序列化对象的类。

System.Xml.Serialization
包含可用于将对象序列化为 XML 格式的文档或流的类。

System.Text.Json
包含可用于将对象序列化为 JSON 格式的文档或流的类。

.NET 中的 JSON 序列化和反序列化（封送和拆收）- 概述

2021/11/16 •

`System.Text.Json` 命名空间提供用于序列化和反序列化 JavaScript 对象表示法 (JSON) 的功能。

库的设计强调对广泛的功能集实现高性能和低内存分配。内置的 UTF-8 支持可优化读写以 UTF-8 编码的 JSON 文本的过程，UTF-8 编码是针对 Web 上的数据和磁盘上的文件的最普遍的编码方式。

库还提供了用于处理内存中文档对象模型 (DOM) 的类。此功能允许对 JSON 文件或字符串中的元素进行随机只读访问。

如何获取库

- 该库是作为 .NET Core 3.0 及更高版本共享框架的一部分内置的。
- 对于早期版本的框架，请安装 System.Text.Json NuGet 包。包支持以下框架：
  - .NET Standard 2.0 及更高版本
  - .NET Framework 4.7.2 及更高版本
  - .NET Core 2.0、2.1 和 2.2

其他资源

- 如何使用库
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何在 .NET 中对 JSON 进行序列化和反序列化（封送和拆收）

2021/11/16 •

本文演示如何使用 System.Text.Json 命名空间向/从 JavaScript 对象表示法 (JSON) 进行序列化和反序列化。如果要从 `Newtonsoft.Json` 移植现有代码，请参阅如何迁移到 `System.Text.Json`。

方向和示例代码直接使用库，而不是通过框架（如 ASP.NET Core）。

大多数序列化示例代码将 JsonSerializerOptions.WriteIndented 设置为 `true`，以 JSON 进行优质打印（包含缩进和空格，以提高可读性）。对于生产用途，通常对于此设置会接受默认值 `false`。

代码示例引用下面的类及其变体：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

命名空间

System.Text.Json 命名空间包含所有入口点和主要类型。System.Text.Json.Serialization 命名空间包含用于高级方案的特性和 API，以及特定于序列化和反序列化的自定义。本文中演示的代码示例要求将 `using` 指令用于其中一个或两个命名空间：

```
using System.Text.Json;
using System.Text.Json.Serialization;
```

IMPORTANT

`System.Text.Json` 中不支持 System.Runtime.Serialization 命名空间中的特性。

如何将 .NET 对象编写为 JSON（序列化）

若要将 JSON 编写为字符串或文件，请调用 JsonSerializer.Serialize 方法。

下面的示例将 JSON 创建为字符串：

```
string jsonString = JsonSerializer.Serialize(weatherForecast);
```

下面的示例使用同步代码创建 JSON 文件：

```
jsonString = JsonSerializer.Serialize(weatherForecast);
File.WriteAllText(fileName, jsonString);
```

下面的示例使用异步代码创建 JSON 文件：

```
using FileStream createStream = File.Create(fileName);
await JsonSerializer.SerializeAsync(createStream, weatherForecast);
```

前面的示例对要序列化的类型使用类型推理。`Serialize()` 的重载采用泛型类型参数：

```
jsonString = JsonSerializer.Serialize<WeatherForecastWithPOCOs>(weatherForecast);
```

序列化示例

下面是一个包含集合类型属性和用户定义类型的示例类：

```
public class WeatherForecastWithPOCOs
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public string SummaryField;
    public IList<DateTimeOffset> DatesAvailable { get; set; }
    public Dictionary<string, HighLowTemps> TemperatureRanges { get; set; }
    public string[] SummaryWords { get; set; }
}

public class HighLowTemps
{
    public int High { get; set; }
    public int Low { get; set; }
}
```

> **TIP**
> "POCO"代表普通旧 CLR 对象。POCO 是一种 .NET 类型，它不通过继承或特性等依赖于任何框架特定类型。

序列化以上类型的实例的 JSON 输出类似于下面的示例。默认情况下，JSON 输出会缩小（将删除空格、缩进和换行符）：

```
{"Date":"2019-08-01T00:00:00-07:00","TemperatureCelsius":25,"Summary":"Hot","DatesAvailable":["2019-08-
01T00:00:00-07:00","2019-08-02T00:00:00-07:00"],"TemperatureRanges":{"Cold":{"High":20,"Low":-10},"Hot":
{"High":60,"Low":20}},"SummaryWords":["Cool","Windy","Humid"]}
```

下面的示例演示相同的 JSON，但进行了格式设置（即，具有空格和缩进的优质打印版本）：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
  },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

序列化为 UTF-8

若要序列化为 UTF-8，请调用 JsonSerializer.SerializeToUtf8Bytes 方法：

```
byte[] jsonUtf8Bytes =JsonSerializer.SerializeToUtf8Bytes(weatherForecast);
```

还有一个采用 Utf8JsonWriter 的 Serialize 重载可用。

序列化为 UTF-8 比使用基于字符串的方法大约快 5-10%。出现这种差别的原因是字节（作为 UTF-8）不需要转换为字符串 (UTF-16)。

序列化行为

- 默认情况下，所有公共属性都会序列化。可以指定要忽略的属性。
- 默认编码器会转义非 ASCII 字符、ASCII 范围内的 HTML 敏感字符以及根据 RFC 8259 JSON 规范必须进行转义的字符。
- 默认情况下，JSON 会缩小。可以对 JSON 进行优质打印。
- 默认情况下，JSON 名称的大小写与 .NET 名称匹配。可以自定义 JSON 名称大小写。
- 默认情况下，检测到循环引用并引发异常。可以保留引用并处理循环引用。
- 默认情况下忽略字段。可以包含字段。

当你在 ASP.NET Core 应用中间接使用 System.Text.Json 时，某些默认行为会有所不同。有关详细信息，请参阅 JsonSerializerOptions 的 Web 默认值。

- 默认情况下，所有公共属性都会序列化。可以指定要忽略的属性。
- 默认编码器会转义非 ASCII 字符、ASCII 范围内的 HTML 敏感字符以及根据 RFC 8259 JSON 规范必须进行转义的字符。
- 默认情况下，JSON 会缩小。可以对 JSON 进行优质打印。
- 默认情况下，JSON 名称的大小写与 .NET 名称匹配。可以自定义 JSON 名称大小写。
- 检测到循环引用并引发异常。
- 将忽略字段。

支持的类型包括：

- 映射到 JavaScript 基元的 .NET 基元，如数值类型、字符串和布尔。
- 用户定义的普通旧 CLR 对象 (POCO)。
- 一维和交错数组 (`T[][]`)。
- 以下命名空间中的集合和字典。
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Immutable
  - System.Collections.Concurrent
  - System.Collections.Specialized
  - System.Collections.ObjectModel

- 映射到 JavaScript 基元的 .NET 基元，如数值类型、字符串和布尔。
- 用户定义的普通旧 CLR 对象 (POCO)。
- 一维和交错数组 (`ArrayName[][]`)。
- `Dictionary<string,TValue>` 其中 `TValue` 是 `object`、`JsonElement` 或 POCO。
- 以下命名空间中的集合。
  - System.Collections
  - System.Collections.Generic
  - System.Collections.Immutable
  - System.Collections.Concurrent
  - System.Collections.Specialized
  - System.Collections.ObjectModel

可以实现自定义转换器以处理其他类型或提供内置转换器不支持的功能。

如何将 JSON 读取为 .NET 对象（反序列化）

若要从字符串或文件进行反序列化，请调用 JsonSerializer.Deserialize 方法。

下面的示例从字符串读取 JSON，并创建前面为序列化示例显示的 `WeatherForecastWithPOCOs` 类的实例：

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithPOCOs>(jsonString);
```

若要使用同步代码从文件进行反序列化，请将文件读入字符串中，如下面的示例中所示：

```
jsonString = File.ReadAllText(fileName);
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString);
```

若要使用异步代码从文件进行反序列化，请调用 DeserializeAsync 方法：

```
using FileStream openStream = File.OpenRead(fileName);
weatherForecast = await JsonSerializer.DeserializeAsync<WeatherForecast>(openStream);
```

从 UTF-8 进行反序列化

若要从 UTF-8 进行反序列化，请调用采用 `ReadOnlySpan<byte>` 或 `Utf8JsonReader` 的 JsonSerializer.Deserialize 重载，如下面的示例中所示。这些示例假设 JSON 处于名为 jsonUtf8Bytes 的字节数组中。

```
var readOnlySpan = new ReadOnlySpan<byte>(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(readOnlySpan);
```

```
var utf8Reader = new Utf8JsonReader(jsonUtf8Bytes);
WeatherForecast deserializedWeatherForecast =
    JsonSerializer.Deserialize<WeatherForecast>(ref utf8Reader);
```

反序列化行为

对 JSON 进行反序列化时，以下行为适用：

- 默认情况下，属性名称匹配区分大小写。可以指定不区分大小写。
- 如果 JSON 包含只读属性的值，则会忽略该值，并且不引发异常。
- 序列化程序会忽略非公共构造函数。
- 支持反序列化为不可变对象或只读属性。请参阅不可变类型和记录。
- 默认情况下，支持将枚举作为数字。可以将枚举名称序列化为字符串。
- 默认情况下忽略字段。可以包含字段。
- 默认情况下，JSON 中的注释或尾随逗号会引发异常。可以允许注释和尾随逗号。
- 默认最大深度为 64。

当你在 ASP.NET Core 应用中间接使用 System.Text.Json 时，某些默认行为会有所不同。有关详细信息，请参阅 JsonSerializerOptions 的 Web 默认值。

- 默认情况下，属性名称匹配区分大小写。可以指定不区分大小写。ASP.NET Core 应用默认指定不区分大小写。
- 如果 JSON 包含只读属性的值，则会忽略该值，并且不引发异常。
- 无参数构造函数（可以是公共的、内部的或专用的）用于反序列化。
- 不支持反序列化为不可变对象或只读属性。
- 默认情况下，支持将枚举作为数字。可以将枚举名称序列化为字符串。
- 不支持字段。
- 默认情况下，JSON 中的注释或尾随逗号会引发异常。可以允许注释和尾随逗号。
- 默认最大深度为 64。

当你在 ASP.NET Core 应用中间接使用 System.Text.Json 时，某些默认行为会有所不同。有关详细信息，请参阅 JsonSerializerOptions 的 Web 默认值。

可以实现自定义转换器以提供内置转换器不支持的功能。

序列化为格式化 JSON

若要对 JSON 输出进行优质打印，请将 JsonSerializerOptions.WriteIndented 设置为 `true`：

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

下面是一个要进行序列化的示例类型，以及进行了优质打印的 JSON 输出：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

如果你通过相同的选项重复使用 `JsonSerializerOptions`，则请勿在每次使用时都创建新的 `JsonSerializerOptions` 实例。对每个调用重复使用同一实例。有关详细信息，请参阅重用 JsonSerializerOptions 实例。

包含字段

在序列化或反序列化时，使用 JsonSerializerOptions.IncludeFields 全局设置或 [JsonInclude] 特性来包含字段，如以下示例中所示：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace Fields
{
    public class Forecast
    {
        public DateTime Date;
        public int TemperatureC;
        public string Summary;
    }

    public class Forecast2
    {
        [JsonInclude]
        public DateTime Date;
        [JsonInclude]
        public int TemperatureC;
        [JsonInclude]
        public string Summary;
    }

    public class Program
    {
        public static void Main()
        {
            var json =
                @"{""Date"":""2020-09-06T11:31:01.923395"",""TemperatureC"":-1,""Summary"":""Cold""} ";
            Console.WriteLine($"Input JSON: {json}");

            var options = new JsonSerializerOptions
            {
                IncludeFields = true,
            };
            var forecast = JsonSerializer.Deserialize<Forecast>(json, options);

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC: {forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");
```

```
            var roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");

            var forecast2 = JsonSerializer.Deserialize<Forecast2>(json);

            Console.WriteLine($"forecast2.Date: {forecast2.Date}");
            Console.WriteLine($"forecast2.TemperatureC: {forecast2.TemperatureC}");
            Console.WriteLine($"forecast2.Summary: {forecast2.Summary}");

            roundTrippedJson = JsonSerializer.Serialize<Forecast2>(forecast2);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast.Date: 9/6/2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
//forecast2.Date: 9/6/2020 11:31:01 AM
//forecast2.TemperatureC: -1
//forecast2.Summary: Cold
//Output JSON: { "Date":"2020-09-06T11:31:01.923395","TemperatureC":-1,"Summary":"Cold"}
```

若要忽略只读字段，请使用 JsonSerializerOptions.IgnoreReadOnlyFields 全局设置。

.NET Core 3.1 的 System.Text.Json 中不支持字段。自定义转换器可提供此功能。

HttpClient 和 HttpContent 扩展方法

序列化和反序列化来自网络的 JSON 有效负载是常见的操作。HttpClient 和 HttpContent 上的扩展方法允许在单个代码行中执行这些操作。这些扩展方法使用 JsonSerializerOptions 的 Web 默认值。

以下示例说明了 HttpClientJsonExtensions.GetFromJsonAsync 和 HttpClientJsonExtensions.PostAsJsonAsync 的用法：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;

namespace HttpClientExtensionMethods
{
    public class User
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Username { get; set; }
        public string Email { get; set; }
    }

    public class Program
    {
        public static async Task Main()
        {
            using HttpClient client = new()
            {
                BaseAddress = new Uri("https://jsonplaceholder.typicode.com")
            };

            // Get the user information.
            User user = await client.GetFromJsonAsync<User>("users/1");
            Console.WriteLine($"Id: {user.Id}");
            Console.WriteLine($"Name: {user.Name}");
            Console.WriteLine($"Username: {user.Username}");
            Console.WriteLine($"Email: {user.Email}");

            // Post a new user.
            HttpResponseMessage response = await client.PostAsJsonAsync("users", user);
            Console.WriteLine(
                $"{(response.IsSuccessStatusCode ? "Success" : "Error")} - {response.StatusCode}");
        }
    }
}

// Produces output like the following example but with different names:
//
//Id: 1
//Name: Tyler King
//Username: Tyler
//Email: Tyler @contoso.com
//Success - Created
```

此外还有 HttpContent 上的 System.Text.Json 的扩展方法。

`HttpClient` 和 `HttpContent` 上的扩展方法在 .NET Core 3.1 的 System.Text.Json 中不可用。

请参阅

- System.Text.Json 概述
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用

- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 实例化 JsonSerializerOptions 实例

2021/11/16 •

本文介绍如何避免在使用 JsonSerializerOptions 时出现性能问题。还介绍了如何使用可用的参数化构造函数。

重用 JsonSerializerOptions 实例

如果你通过相同的选项重复使用 `JsonSerializerOptions`，则请勿在每次使用时都创建新的 `JsonSerializerOptions` 实例。对每个调用重复使用同一实例。本指南适用于你为自定义转换器编写的代码，以及调用 JsonSerializer.Serialize 或 JsonSerializer.Deserialize时。

以下代码演示使用新的选项实例所带来的性能损失。

```
using System;
using System.Diagnostics;
using System.Text.Json;

namespace OptionsPerfDemo
{
    public record Forecast(DateTime Date, int TemperatureC, string Summary);

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new(DateTime.Now, 40, "Hot");
            JsonSerializerOptions options = new() { WriteIndented = true };
            int iterations = 100000;

            var watch = Stopwatch.StartNew();
            for (int i = 0; i < iterations; i++)
            {
                Serialize(forecast, options);
            }
            watch.Stop();
            Console.WriteLine($"Elapsed time using one options instance: {watch.ElapsedMilliseconds}");

            watch = Stopwatch.StartNew();
            for (int i = 0; i < iterations; i++)
            {
                Serialize(forecast);
            }
            watch.Stop();
            Console.WriteLine($"Elapsed time creating new options instances: {watch.ElapsedMilliseconds}");
        }

        private static void Serialize(Forecast forecast, JsonSerializerOptions? options = null)
        {
            _ = JsonSerializer.Serialize<Forecast>(
                forecast,
                options ?? new JsonSerializerOptions() { WriteIndented = true });
        }
    }
}

// Produces output like the following example:
//
//Elapsed time using one options instance: 190
//Elapsed time creating new options instances: 40140
```

前面的代码使用相同的选项实例对一个小型对象进行了 100,000 次序列化。然后，它对相同的对象进行了相同次数的序列化，且每次都创建一个新的选项实例。典型运行时的差值为 190 毫秒，而不是 40,140 毫秒。如果增加迭代的次数，差值甚至越大。

当向序列化程序传递新的选项实例时，在对象图中每种类型的第一次序列化期间，序列化程序将执行一个预热阶段。此预热包括创建序列化所需的元数据的缓存。元数据包括对属性 Getter 的委托、Setter、构造函数参数、指定特性等。此元数据缓存存储在选项实例中。相同的预热过程和缓存适用于反序列化。

`JsonSerializerOptions` 实例中元数据缓存的大小取决于要序列化的类型的数量。如果向序列化程序传递多种类型（例如动态生成的类型），缓存大小将继续增长，最终可导致 `OutOfMemoryException`。

复制 JsonSerializerOptions

存在 JsonSerializerOptions constructor，可让你使用与现有实例相同的选项来创建新的实例，如以下示例中所示：

```
using System;
using System.Text.Json;

namespace CopyOptions
{
    public class Forecast
    {
        public DateTime Date { get; init; }
        public int TemperatureC { get; set; }
        public string Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new()
            {
                WriteIndented = true
            };

            JsonSerializerOptions optionsCopy = new(options);
            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, optionsCopy);

            Console.WriteLine($"Output JSON:\n{forecastJson}");
        }
    }
}

// Produces output like the following example:
//
//Output JSON:
//{
//  "Date": "2020-10-21T15:40:06.8998502-07:00",
//  "TemperatureC": 40,
//  "Summary": "Hot"
//}
```

使用现有实例的 `JsonSerializerOptions` 构造函数在 .NET Core 3.1 中不可用。

JsonSerializerOptions 的 Web 默认值

下面是具有 Web 应用不同默认值的选项：

- PropertyNameCaseInsensitive = `true`
- JsonNamingPolicy = CamelCase
- NumberHandling = AllowReadingFromString

存在 JsonSerializerOptions constructor，可让你通过 ASP.NET Core 对 Web 应用使用的默认选项创建新的实例，如以下示例中所示：

```
using System;
using System.Text.Json;

namespace OptionsDefaults
{
    public class Forecast
    {
        public DateTime Date { get; init; }
        public int TemperatureC { get; set; }
        public string Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new(JsonSerializerDefaults.Web)
            {
                WriteIndented = true
            };

            Console.WriteLine(
                $"PropertyNameCaseInsensitive: {options.PropertyNameCaseInsensitive}");
            Console.WriteLine(
                $"JsonNamingPolicy: {options.PropertyNamingPolicy}");
            Console.WriteLine(
                $"NumberHandling: {options.NumberHandling}");

            string forecastJson = JsonSerializer.Serialize<Forecast>(forecast, options);
            Console.WriteLine($"Output JSON:\n{forecastJson}");

            Forecast forecastDeserialized =
                JsonSerializer.Deserialize<Forecast>(forecastJson, options);

            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC: {forecastDeserialized.TemperatureC}");
            Console.WriteLine($"Summary: {forecastDeserialized.Summary}");
        }
    }
}

// Produces output like the following example:
//
//PropertyNameCaseInsensitive: True
//JsonNamingPolicy: System.Text.Json.JsonCamelCaseNamingPolicy
//NumberHandling: AllowReadingFromString
//Output JSON:
//{
//  "date": "2020-10-21T15:40:06.9040831-07:00",
//  "temperatureC": 40,
//  "summary": "Hot"
//}
//Date: 10 / 21 / 2020 3:40:06 PM
//TemperatureC: 40
//Summary: Hot
```

下面是具有 Web 应用不同默认值的选项：

- PropertyNameCaseInsensitive = `true`

- JsonNamingPolicy = CamelCase

指定一组默认值的 `JsonSerializerOptions` 构造函数在 .NET Core 3.1 中不可用。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 启用不区分大小写的属性名称

2021/11/16 •

本文将介绍如何使用 `System.Text.Json` 命名空间启用不区分大小写的属性名称匹配。

不区分大小写的属性匹配

默认情况下，反序列化会查找 JSON 与目标对象属性之间区分大小写的属性名称匹配。若要更改该行为，请将 JsonSerializerOptions.PropertyNameCaseInsensitive 设置为 `true`：

> **NOTE**
> Web 默认值为不区分大小写。

```
var options = new JsonSerializerOptions
{
    PropertyNameCaseInsensitive = true
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString, options);
```

下面是具有 camel 大小写属性名称的示例 JSON。它可以反序列化为具有帕斯卡拼写法属性名称的以下类型。

```
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
}
```

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器

- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 自定义属性名称和值

2021/11/16 •

默认情况下，属性名称和字典键在 JSON 输出中保持不变，包括大小写。枚举值表示为数字。本文将指导如何进行以下操作：

> **NOTE**
> Web 默认值为 camel 形式大小写。

- 自定义单个属性名称
- 将所有属性名称转换为 camel 大小写
- 实现自定义属性命名策略
- 将字典键转换为 camel 大小写
- 将枚举转换为字符串和 camel 大小写

对于需要对 JSON 属性名称和值进行特殊处理的其他方案，可以实现自定义转换器。

自定义单个属性名称

若要设置单个属性的名称，请使用 [JsonPropertyName] 特性。

下面是要进行序列化的示例类型和生成的 JSON：

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "Wind": 35
}
```

此特性设置的属性名称：

- 同时适用于两个方向（序列化和反序列化）。
- 优先于属性命名策略。

对所有 JSON 属性名称使用 camel 大小写

若要对所有 JSON 属性名称使用 camel 大小写，请将 JsonSerializerOptions.PropertyNamingPolicy 设置为 `JsonNamingPolicy.CamelCase`，如下面的示例中所示：

```
var serializeOptions = new JsonSerializerOptions
{
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

下面是要进行序列化的示例类和 JSON 输出：

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

```
{
  "date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "summary": "Hot",
  "Wind": 35
}
```

camel 大小写属性命名策略：

- 适用于序列化和反序列化。
- 由 `[JsonPropertyName]` 特性替代。这便是示例中的 JSON 属性名称 `Wind` 不是 camel 大小写的原因。

使用自定义 JSON 属性命名策略

若要使用自定义 JSON 属性命名策略，请创建派生自 JsonNamingPolicy 的类，并替代 ConvertName 方法，如下面的示例中所示：

```
using System.Text.Json;

namespace SystemTextJsonSamples
{
    public class UpperCaseNamingPolicy : JsonNamingPolicy
    {
        public override string ConvertName(string name) =>
            name.ToUpper();
    }
}
```

然后，将 JsonSerializerOptions.PropertyNamingPolicy 属性设置为命名策略类的实例：

```
var options = new JsonSerializerOptions
{
    PropertyNamingPolicy = new UpperCaseNamingPolicy(),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

下面是要进行序列化的示例类和 JSON 输出：

```
public class WeatherForecastWithPropertyNameAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonPropertyName("Wind")]
    public int WindSpeed { get; set; }
}
```

```
{
  "DATE": "2019-08-01T00:00:00-07:00",
  "TEMPERATURECELSIUS": 25,
  "SUMMARY": "Hot",
  "Wind": 35
}
```

JSON 属性命名策略：

- 适用于序列化和反序列化。
- 由 `[JsonPropertyName]` 特性替代。这便是示例中的 JSON 属性名称 `Wind` 不是大写的原因。

Camel 大小写字典键

如果要序列化的对象的属性为 `Dictionary<string,TValue>` 类型，则 `string` 键可转换为 camel 大小写。为此，请将 DictionaryKeyPolicy 设置为 `JsonNamingPolicy.CamelCase`，如下面的示例中所示：

```
var options = new JsonSerializerOptions
{
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

使用名为 `TemperatureRanges` 且具有键值对 `"ColdMinTemp", 20` 和 `"HotMinTemp", 40` 的字典序列化对象会产生类似于以下示例的 JSON 输出：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "TemperatureRanges": {
    "coldMinTemp": 20,
    "hotMinTemp": 40
  }
}
```

字典键的 camel 大小写命名策略仅适用于序列化。如果对字典进行反序列化，即使为 `DictionaryKeyPolicy` 指定 `JsonNamingPolicy.CamelCase`，键也会与 JSON 文件匹配。

作为字符串的枚举

默认情况下，枚举会序列化为数字。若要将枚举名称序列化为字符串，请使用 JsonStringEnumConverter。

例如，假设需要序列化以下具有枚举的类：

```
public class WeatherForecastWithEnum
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public Summary Summary { get; set; }
}

public enum Summary
{
    Cold, Cool, Warm, Hot
}
```

如果 Summary 为 `Hot`，则默认情况下序列化的 JSON 具有数值 3：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": 3
}
```

下面的示例代码序列化枚举名称(而不是数值)，并将名称转换为 camel 大小写：

```
options = new JsonSerializerOptions
{
    WriteIndented = true,
    Converters =
    {
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)
    }
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

生成的 JSON 类似于以下示例：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "hot"
}
```

还可以反序列化枚举字符串名称，如以下示例中所示：

```
options = new JsonSerializerOptions
{
    Converters =
    {
        new JsonStringEnumConverter(JsonNamingPolicy.CamelCase)
    }
};
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithEnum>(jsonString, options);
```

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配

- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 忽略属性

2021/11/16 •

将 C# 对象序列化为 JavaScript 对象表示法 (JSON) 时，默认情况下，所有公共属性都会序列化。如果不想让某些属性出现在生成的 JSON 中，则有几个选项可用。本文将介绍如何根据各种条件忽略属性：

- 单个属性
- 所有只读属性
- 所有 null 值属性

忽略单个属性

若要忽略单个属性，请使用 [JsonIgnore] 特性。

下面是要进行序列化的示例类型和 JSON 输出：

```
public class WeatherForecastWithIgnoreAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    [JsonIgnore]
    public string Summary { get; set; }
}
```

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
}
```

可以通过设置 [JsonIgnore] 特性的 `Condition` 属性来指定条件排除。JsonIgnoreCondition 枚举提供下列选项：

- `Always` - 始终忽略属性。如果未指定 `Condition`，则假设此选项。
- `Never` - 无论 `DefaultIgnoreCondition`、`IgnoreReadOnlyProperties` 和 `IgnoreReadOnlyFields` 全局设置如何，始终序列化和反序列化属性。
- `WhenWritingDefault` - 如果属性是引用类型 `null` 可为 null 的值类型 `null` 或值类型 `default`，则在序列化中忽略属性。
- `WhenWritingNull` - 如果属性是引用类型 `null` 或可为 null 的值类型 `null`，则在序列化中忽略属性。

下面的示例说明 [JsonIgnore] 特性的 `Condition` 属性的用法：

```
#nullable enable
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace JsonIgnoreAttributeExample
{
    public class Forecast
    {
        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingDefault)]
        public DateTime Date { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.Never)]
        public int TemperatureC { get; set; }

        [JsonIgnore(Condition = JsonIgnoreCondition.WhenWritingNull)]
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = default,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast,options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{"TemperatureC":0}
```

忽略所有只读属性

如果属性包含公共 getter 而不是公共 setter，则属性为只读。若要在序列化时忽略所有只读属性，请将 JsonSerializerOptions.IgnoreReadOnlyProperties 设置为 `true`，如以下示例中所示：

```
var options = new JsonSerializerOptions
{
    IgnoreReadOnlyProperties = true,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

下面是要进行序列化的示例类型和 JSON 输出：

```
public class WeatherForecastWithROProperty
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public int WindSpeedReadOnly { get; private set; } = 35;
}
```

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
}
```

此选项仅适用于序列化。在反序列化过程中，默认情况下会忽略只读属性。

此选项仅适用于属性。若要在序列化字段时忽略只读字段，请使用 JsonSerializerOptions.IgnoreReadOnlyFields 全局设置。

忽略所有 null 值属性

若要忽略所有 null 值属性，请将 DefaultIgnoreCondition 属性设置为 WhenWritingNull，如以下示例中所示：

```
#nullable enable
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreNullOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingNull
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{"Date":"2020-10-30T10:11:40.2359135-07:00","TemperatureC":0}
```

若要在序列化时忽略所有 null 值属性，请将 IgnoreNullValues 属性设置为 `true`，如以下示例中所示：

```
var options = new JsonSerializerOptions
{
    IgnoreNullValues = true,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

下面是要进行序列化的示例对象和 JSON 输出：

PROPERTY	[
`Date`	`8/1/2019 12:00:00 AM -07:00`
`TemperatureCelsius`	`25`
`Summary`	`null`

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25
}
```

忽略所有默认值属性

若要防止对值类型属性中的默认值进行序列化，请将 DefaultIgnoreCondition 属性设置为 WhenWritingDefault，如以下示例中所示：

```
#nullable enable
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace IgnoreValueDefaultOnSerialize
{
    public class Forecast
    {
        public DateTime Date { get; set; }
        public int TemperatureC { get; set; }
        public string? Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                Summary = null,
                TemperatureC = default
            };

            JsonSerializerOptions options = new()
            {
                DefaultIgnoreCondition = JsonIgnoreCondition.WhenWritingDefault
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine(forecastJson);
        }
    }
}

// Produces output like the following example:
//
//{ "Date":"2020-10-21T15:40:06.8920138-07:00"}
```

WhenWritingDefault 设置还会阻止对 null 值引用类型和可为 null 的值类型属性进行序列化。

在 .NET Core 3.1 的 `System.Text.Json` 中，无内置方式来阻止对值类型为默认值的属性进行序列化。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化

- 启用不区分大小写的匹配
- 自定义属性名称和值
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 支持某种无效的 JSON

2021/11/16 •

本文将介绍如何能够在 JSON 中使用注释、尾随逗号和带引号的数字，以及如何将数字编写为字符串。

允许注释和尾随逗号

默认情况下，JSON 中不允许使用注释和尾随逗号。若要在 JSON 中允许注释，请将 JsonSerializerOptions.ReadCommentHandling 属性设置为 `JsonCommentHandling.Skip`。若要允许尾随逗号，请将 JsonSerializerOptions.AllowTrailingCommas 属性设置为 `true`。下面的示例演示如何允许这两者：

```
var options = new JsonSerializerOptions
{
    ReadCommentHandling = JsonCommentHandling.Skip,
    AllowTrailingCommas = true,
};
var weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString, options);
```

下面是包含注释和尾随逗号的示例 JSON：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25, // Fahrenheit 77
  "Summary": "Hot", /* Zharko */
  // Comments on
  /* separate lines */
}
```

允许或写入带引号的数字

某些序列化程序将数字编码为 JSON 字符串（括在引号中）。

例如：

```
{
    "DegreesCelsius": "23"
}
```

不是：

```
{
    "DegreesCelsius": 23
}
```

若要在整个输入对象图中序列化带引号的数字或接受带引号的数字，请设置 JsonSerializerOptions.NumberHandling，如以下示例中所示：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace QuotedNumbers
{
    public class Forecast
    {
        public DateTime Date { get; init; }
        public int TemperatureC { get; set; }
        public string Summary { get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new()
            {
                Date = DateTime.Now,
                TemperatureC = 40,
                Summary = "Hot"
            };

            JsonSerializerOptions options = new()
            {
                NumberHandling =
                    JsonNumberHandling.AllowReadingFromString |
                    JsonNumberHandling.WriteAsString,
                WriteIndented = true
            };

            string forecastJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON:\n{forecastJson}");

            Forecast forecastDeserialized =
                JsonSerializer.Deserialize<Forecast>(forecastJson, options);

            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC: {forecastDeserialized.TemperatureC}");
            Console.WriteLine($"Summary: {forecastDeserialized.Summary}");
        }
    }
}

// Produces output like the following example:
//
//Output JSON:
//{
//  "Date": "2020-10-23T12:27:06.4017385-07:00",
//  "TemperatureC": "40",
//  "Summary": "Hot"
//}
//Date: 10/23/2020 12:27:06 PM
//TemperatureC: 40
//Summary: Hot
```

在通过 ASP.NET Core 间接使用 `System.Text.Json` 时，反序列化时允许使用带引号的数字，因为 ASP.NET Core 指定 Web 默认选项。

若要允许或写入特定属性、字段或类型的带引号的数字，请使用 [JsonNumberHandling] 特性。

.NET Core 3.1 中的 `System.Text.Json` 不支持序列化或反序列化用引号引起来的数字。有关详细信息，请参阅允许或写入带引号的数字。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 处理溢出 JSON

2021/11/16 •

本文将介绍如何使用 `System.Text.Json` 命名空间处理溢出 JSON。

处理溢出 JSON

反序列化时，可能会在 JSON 中收到不是由目标类型的属性表示的数据。例如，假设目标类型如下：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

要反序列化的 JSON 如下：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "temperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

如果将显示的 JSON 反序列化为显示的类型，则 `DatesAvailable` 和 `SummaryWords` 属性无处可去，会丢失。若要捕获额外数据（如这些属性），请将 [JsonExtensionData] 特性应用于类型 `Dictionary<string,object>` 或 `Dictionary<string,JsonElement>` 的属性：

```
public class WeatherForecastWithExtensionData
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    [JsonExtensionData]
    public Dictionary<string, JsonElement> ExtensionData { get; set; }
}
```

将前面显示的 JSON 反序列化为此示例类型时，额外数据会成为 `ExtensionData` 属性的键值对：

PROPERTY	[	[[
`Date`	`"8/1/2019 12:00:00 AM -07:00"`	

PROPERTY	[	[[
`TemperatureCelsius`	`0`	区分大小写的不匹配(JSON 中的 `temperatureCelsius`),因此未设置属性。
`Summary`	`"Hot"`	
`ExtensionData`	`temperatureCelsius: 25`	因为大小写不匹配,所以此 JSON 属性是额外属性,会成为字典中的键值对。
`DatesAvailable`	`[ "8/1/2019 12:00:00 AM -07:00", "8/2/2019 12:00:00 AM -07:00" ]`	JSON 中的额外属性会成为键值对,将数组作为值对象。
`SummaryWords`	`[ "Cool", "Windy", "Humid" ]`	JSON 中的额外属性会成为键值对,将数组作为值对象。

序列化目标对象时,扩展数据键值对会成为 JSON 属性,就如同它们处于传入 JSON 中一样:

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 0,
  "Summary": "Hot",
  "temperatureCelsius": 25,
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
```

请注意,`ExtensionData` 属性名称不会出现在 JSON 中。此行为使 JSON 可以进行往返,而不会丢失任何不会以其他方式进行反序列化的额外数据。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器

- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 保留引用并处理循环引用

2021/11/16 •

若要保留引用并处理循环引用，请将 ReferenceHandler 设置为 Preserve。此设置会导致以下行为：

- 在序列化时：

  编写复杂类型时，序列化程序还会写入元数据属性（`$id`、`$values` 和 `$ref`）。

- 在反序列化时：

  需要元数据（虽然不是必需的），并且反序列化程序会尝试理解它。

下面的代码演示 `Preserve` 属性的用法。

```
using System;
using System.Collections.Generic;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace PreserveReferences
{
    public class Employee
    {
        public string Name { get; set; }
        public Employee Manager { get; set; }
        public List<Employee> DirectReports { get; set; }
    }

    public class Program
    {
        public static void Main()
        {
            Employee tyler = new()
            {
                Name = "Tyler Stein"
            };

            Employee adrian = new()
            {
                Name = "Adrian King"
            };

            tyler.DirectReports = new List<Employee> { adrian };
            adrian.Manager = tyler;

            JsonSerializerOptions options = new()
            {
                ReferenceHandler = ReferenceHandler.Preserve,
                WriteIndented = true
            };

            string tylerJson = JsonSerializer.Serialize(tyler, options);
            Console.WriteLine($"Tyler serialized:\n{tylerJson}");

            Employee tylerDeserialized =
                JsonSerializer.Deserialize<Employee>(tylerJson, options);

            Console.WriteLine(
```

```
                "Tyler is manager of Tyler's first direct report: ");
            Console.WriteLine(
                tylerDeserialized.DirectReports[0].Manager == tylerDeserialized);
        }
    }
}

// Produces output like the following example:
//
//Tyler serialized:
//{
//  "$id": "1",
//  "Name": "Tyler Stein",
//  "Manager": null,
//  "DirectReports": {
//    "$id": "2",
//    "$values": [
//      {
//        "$id": "3",
//        "Name": "Adrian King",
//        "Manager": {
//          "$ref": "1"
//        },
//        "DirectReports": null
//      }
//    ]
//  }
//}
//Tyler is manager of Tyler's first direct report:
//True
```

此功能不能用于保留值类型或不可变类型。在反序列化时，将在读取整个有效负载后创建不可变类型的实例。因此，如果对同一实例的引用出现在 JSON 有效负载中，则无法对其进行反序列化。

对于值类型、不可变类型和数组，不会序列化任何引用元数据。反序列化时，如果发现 `$ref` 或 `$id`，则会引发异常。但是，值类型忽略 `$id`（对于集合，则为 `$values`），以便可以反序列化使用 Newtonsoft.Json 序列化的有效负载。Newtonsoft.Json 为此类类型序列化元数据。

为了确定对象是否相等，System.Text.Json 在比较两个对象实例时使用引用相等性 (Object.ReferenceEquals(Object, Object)) 而不是值相等性 (Object.Equals(Object) 的 ReferenceEqualityComparer.Instance。

有关如何序列化和反序列化引用的详细信息，请参阅 ReferenceHandler.Preserve。

ReferenceResolver 类定义在序列化和反序列化过程中保留引用的行为。创建派生类以指定自定义行为。有关示例，请参阅 GuidReferenceResolver。

.NET Core 3.1 中的 System.Text.Json 仅支持按值进行进行序列化，并对循环引用引发异常。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 不可变类型和非公共访问器

- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何通过 System.Text.Json 使用不可变类型和非公共访问器

2021/11/16 •

本文将介绍如何通过 `System.Text.Json` 命名空间使用不可变类型（如“记录”）。

不可变类型和记录

`System.Text.Json` 可以使用参数化构造函数，这可以反序列化不可变的类或结构。对于类，如果唯一构造函数是参数化的构造函数，则将使用该构造函数。对于结构或包含多个构造函数的类，通过应用 [JsonConstructor] 特性来指定要使用的构造函数。如果未使用该特性，则始终使用公共无参数构造函数（如果存在）。该特性只能与公共构造函数一起使用。以下示例使用 `[JsonConstructor]` 特性：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace ImmutableTypes
{
    public struct Forecast
    {
        public DateTime Date { get; }
        public int TemperatureC { get; }
        public string Summary { get; }

        [JsonConstructor]
        public Forecast(DateTime date, int temperatureC, string summary) =>
            (Date, TemperatureC, Summary) = (date, temperatureC, summary);
    }

    public class Program
    {
        public static void Main()
        {
            var json = @"{""date"":""2020-09-06T11:31:01.923395-07:00"",""temperatureC"":-
1,""summary"":""Cold""} ";
            Console.WriteLine($"Input JSON: {json}");

            var options = new JsonSerializerOptions(JsonSerializerDefaults.Web);

            var forecast = JsonSerializer.Deserialize<Forecast>(json, options);

            Console.WriteLine($"forecast.Date: {forecast.Date}");
            Console.WriteLine($"forecast.TemperatureC: {forecast.TemperatureC}");
            Console.WriteLine($"forecast.Summary: {forecast.Summary}");

            var roundTrippedJson =
                JsonSerializer.Serialize<Forecast>(forecast, options);

            Console.WriteLine($"Output JSON: {roundTrippedJson}");

        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "date":"2020-09-06T11:31:01.923395-07:00","temperatureC":-1,"summary":"Cold"}
//forecast.Date: 9 / 6 / 2020 11:31:01 AM
//forecast.TemperatureC: -1
//forecast.Summary: Cold
//Output JSON: { "date":"2020-09-06T11:31:01.923395-07:00","temperatureC":-1,"summary":"Cold"}
```

还支持 C# 9 记录，如以下示例中所示：

```
#nullable enable
using System;
using System.Text.Json;

namespace Records
{
    public record Forecast(DateTime Date, int TemperatureC)
    {
        public string? Summary { get; init; }
    };

    public class Program
    {
        public static void Main()
        {
            Forecast forecast = new(DateTime.Now, 40)
            {
                Summary = "Hot!"
            };

            string forecastJson = JsonSerializer.Serialize<Forecast>(forecast);
            Console.WriteLine(forecastJson);
            Forecast? forecastObj = JsonSerializer.Deserialize<Forecast>(forecastJson);
            Console.WriteLine(forecastObj);
        }
    }
}

// Produces output like the following example:
//
//{ "Date":"2020-10-21T15:26:10.5044594-07:00","TemperatureC":40,"Summary":"Hot!"}
//Forecast { Date = 10 / 21 / 2020 3:26:10 PM, TemperatureC = 40, Summary = Hot! }
```

对于因其所有属性资源库都是非公共的而不可变的类型，请参阅以下部分，了解非公共属性访问器。

.NET Core 3.1 不支持 `JsonConstructorAttribute` 和 C# 9 记录。

非公共属性访问器

若要允许使用非公共属性访问器，请使用 [JsonInclude] 特性，如以下示例中所示：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace NonPublicAccessors
{
    public class Forecast
    {
        public DateTime Date { get; init; }

        [JsonInclude]
        public int TemperatureC { get; private set; }

        [JsonInclude]
        public string Summary { private get; set; }
    };

    public class Program
    {
        public static void Main()
        {
            string json = @"{""Date"":""2020-10-23T09:51:03.8702889-
07:00"",""TemperatureC"":40,""Summary"":""Hot""}";
            Console.WriteLine($"Input JSON: {json}");

            Forecast forecastDeserialized = JsonSerializer.Deserialize<Forecast>(json);
            Console.WriteLine($"Date: {forecastDeserialized.Date}");
            Console.WriteLine($"TemperatureC: {forecastDeserialized.TemperatureC}");

            json = JsonSerializer.Serialize<Forecast>(forecastDeserialized);
            Console.WriteLine($"Output JSON: {json}");
        }
    }
}

// Produces output like the following example:
//
//Input JSON: { "Date":"2020-10-23T09:51:03.8702889-07:00","TemperatureC":40,"Summary":"Hot"}
//Date: 10 / 23 / 2020 9:51:03 AM
//TemperatureC: 40
//Output JSON: { "Date":"2020-10-23T09:51:03.8702889-07:00","TemperatureC":40,"Summary":"Hot"}
```

.NET Core 3.1 不支持非公共属性访问器。有关详细信息，请参阅从 Newtonsoft.Json 迁移一文。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序

- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 序列化派生类的属性

2021/11/16 •

本文将介绍如何使用 `System.Text.Json` 命名空间序列化派生类的属性。

序列化派生类的属性

不支持多态类型层次结构的序列化。例如，如果属性定义为接口或抽象类，则即使运行时类型具有其他属性，也只会序列化对接口或抽象类定义的属性。此部分中介绍了此行为的例外情况。

例如，假设有一个 `WeatherForecast` 类和一个派生类 `WeatherForecastDerived`：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

```
public class WeatherForecastDerived : WeatherForecast
{
    public int WindSpeed { get; set; }
}
```

并且假设 `Serialize` 方法的类型参数在编译时为 `WeatherForecast`：

```
var options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<WeatherForecast>(weatherForecast, options);
```

在这种情况下，即使 `weatherForecast` 对象实际上是 `WeatherForecastDerived` 对象，也不会序列化 `WindSpeed` 属性。仅序列化基类属性：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

此行为旨在帮助防止在运行时创建的派生类型中发生意外数据泄露。

若要序列化前面示例中派生类型的属性，请使用以下方法之一：

- 调用 Serialize 的重载，以便在运行时指定类型：

  ```
  options = new JsonSerializerOptions
  {
      WriteIndented = true
  };
  jsonString = JsonSerializer.Serialize(weatherForecast, weatherForecast.GetType(), options);
  ```

- 将要序列化的对象声明为 `object`。

```
options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize<object>(weatherForecast, options);
```

在前面的示例方案中，这两种方法都会使 `WindSpeed` 属性包含在 JSON 输出中：

```
{
  "WindSpeed": 35,
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

> **IMPORTANT**
>
> 这些方法只为要序列化的根对象提供多态序列化，而不为该根对象的属性提供。

如果将较低级别的对象定义为类型 `object`，则可以对它们进行多态序列化。例如，假设 `WeatherForecast` 类具有一个名为 `PreviousForecast` 的属性，该属性可以定义为类型 `WeatherForecast` 或 `object`：

```
public class WeatherForecastWithPrevious
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public WeatherForecast PreviousForecast { get; set; }
}
```

```
public class WeatherForecastWithPreviousAsObject
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public object PreviousForecast { get; set; }
}
```

如果 `PreviousForecast` 属性包含 `WeatherForecastDerived` 的实例：

- 序列化 `WeatherForecastWithPrevious` 的 JSON 输出不包含 `WindSpeed`。
- 序列化 `WeatherForecastWithPreviousAsObject` 的 JSON 输出包含 `WindSpeed`。

若要序列化 `WeatherForecastWithPreviousAsObject`，无需调用 `Serialize<object>` 或 `GetType`，因为根对象不是可能属于派生类型的对象。下面的代码示例不调用 `Serialize<object>` 或 `GetType`：

```
options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecastWithPreviousAsObject, options);
```

前面的代码会正确地序列化 `WeatherForecastWithPreviousAsObject`：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "PreviousForecast": {
    "WindSpeed": 35,
    "Date": "2019-08-01T00:00:00-07:00",
    "TemperatureCelsius": 25,
    "Summary": "Hot"
  }
}
```

将属性定义为 `object` 的相同方法适用于接口。假设具有以下接口和实现，并且要使用包含实现实例的属性序列化类：

```
using System;

namespace SystemTextJsonSamples
{
    public interface IForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string Summary { get; set; }
    }

    public class Forecast : IForecast
    {
        public DateTimeOffset Date { get; set; }
        public int TemperatureCelsius { get; set; }
        public string Summary { get; set; }
        public int WindSpeed { get; set; }
    }

    public class Forecasts
    {
        public IForecast Monday { get; set; }
        public object Tuesday { get; set; }
    }
}
```

序列化 `Forecasts` 的实例时，只有 `Tuesday` 显示 `WindSpeed` 属性，因为 `Tuesday` 定义为 `object`：

```
var forecasts = new Forecasts
{
    Monday = new Forecast
    {
        Date = DateTime.Parse("2020-01-06"),
        TemperatureCelsius = 10,
        Summary = "Cool",
        WindSpeed = 8
    },
    Tuesday = new Forecast
    {
        Date = DateTime.Parse("2020-01-07"),
        TemperatureCelsius = 11,
        Summary = "Rainy",
        WindSpeed = 10
    }
};

options = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(forecasts, options);
```

下面的示例显示前面代码生成的 JSON：

```
{
  "Monday": {
    "Date": "2020-01-06T00:00:00-08:00",
    "TemperatureCelsius": 10,
    "Summary": "Cool"
  },
  "Tuesday": {
    "Date": "2020-01-07T00:00:00-08:00",
    "TemperatureCelsius": 11,
    "Summary": "Rainy",
    "WindSpeed": 10
  }
}
```

有关多态序列化的详细信息，以及有关反序列化的信息，请参阅如何从 Newtonsoft.Json 迁移到 System.Text.Json。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序

- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何从 Newtonsoft.Json 迁移到 System.Text.Json

2021/11/16 •

本文演示如何从 Newtonsoft.Json 迁移到 System.Text.Json。

`System.Text.Json` 命名空间提供用于序列化和反序列化 JavaScript 对象表示法 (JSON) 的功能。`System.Text.Json` 库包含在 .NET Core 3.1 和更高版本的运行时中。对于其他目标框架，请安装 System.Text.Json NuGet 包。包支持以下框架：

- .NET Standard 2.0 及更高版本
- .NET Framework 4.7.2 及更高版本
- .NET Core 2.0、2.1 和 2.2

`System.Text.Json` 主要关注性能、安全性和标准符合性。它在默认行为方面有一些重要差异，不打算具有与 `Newtonsoft.Json` 相同的功能。对于某些方案，`System.Text.Json` 没有内置功能，但有建议解决方法。对于其他方案，解决方法是不切实际的。如果你的应用程序依赖于缺少的功能，请考虑提交问题以了解是否可以添加对你的方案的支持。

本文的大部分内容介绍如何使用 JsonSerializer API，不过也包含有关如何使用 JsonDocument（表示文档对象模型或 DOM）、Utf8JsonReader 和 Utf8JsonWriter 类型的指导。

介绍 Newtonsoft.Json 与 System.Text.Json 之间差异的表格

下表列出 `Newtonsoft.Json` 功能和 `System.Text.Json` 等效功能。这些等效功能分为以下类别：

- 受支持功能支持。从 `System.Text.Json` 获取类似行为可能需要使用特性或全局选项。
- 不受支持，可能有解决方法。解决方法是自定义转换器，它们可能无法提供与 `Newtonsoft.Json` 功能完全相同的功能。对于其中一些功能，提供示例代码作为示例。如果你依赖于这些 `Newtonsoft.Json` 功能，迁移需要修改 .NET 对象模型或进行其他代码更改。
- 不受支持，解决方法不可行或无法提供。如果你依赖于这些 `Newtonsoft.Json` 功能，则无法在不进行重大更改的情况下进行迁移。

NEWTONSOFT.JSON	SYSTEM.TEXT.JSON	
默认情况下不区分大小写的反序列化	✔ PropertyNameCaseInsensitive 全局设置	
Camel 大小写属性名称	✔ PropertyNamingPolicy 全局设置	
最小字符转义	✔ 严格字符转义，可配置	
`NullValueHandling.Ignore` 全局设置	✔ DefaultIgnoreCondition 全局选项	有条件地忽略属性
允许注释	✔ ReadCommentHandling 全局设置	
允许尾随逗号	✔ AllowTrailingCommas 全局设置	
自定义转换器注册	✔ 优先级顺序不同	

NEWTONSOFT.JSON	SYSTEM.TEXT.JSON	
默认情况下无最大深度	✔ 默认最大深度为 64，可配置	
`PreserveReferencesHandling` 全局设置	✔ ReferenceHandling 全局设置	
序列化或反序列化带引号的数字	✔ NumberHandling 全局设置，[JsonNumberHandling] 特性	
反序列化为不可变类和结构	✔ JsonConstructor，C# 9 记录	
支持字段	✔ IncludeFields 全局设置，[JsonInclude] 特性	
`DefaultValueHandling` 全局设置	✔ DefaultIgnoreCondition 全局设置	
`[JsonProperty]` 上的 `NullValueHandling` 设置	✔ JsonIgnore 特性	
`[JsonProperty]` 上的 `DefaultValueHandling` 设置	✔ JsonIgnore 特性	
反序列化具有非字符串键的 `Dictionary`	✔ 受支持	
支持非公共属性资源库和 Getter	✔ JsonInclude 特性	
`[JsonConstructor]` 特性	✔ [JsonConstructor] 特性	
支持范围广泛的类型	⚠ 某些类型需要自定义转换器	
多态序列化	⚠ 不受支持，解决方法，示例	
多态反序列化	⚠ 不受支持，解决方法，示例	
将推断类型反序列化为 `object` 属性	⚠ 不受支持，解决方法，示例	
将 JSON `null` 文本反序列化为不可为 null 的值类型	⚠ 不受支持，解决方法，示例	
`[JsonProperty]` 特性上的 `Required` 设置	⚠ 不受支持，解决方法，示例	
`DefaultContractResolver` 用于忽略属性	⚠ 不受支持，解决方法，示例	
`DateTimeZoneHandling`、`DateFormatString` 设置	⚠ 不受支持，解决方法，示例	
回调	⚠ 不受支持，解决方法，示例	
`JsonConvert.PopulateObject` 方法	⚠ 不受支持，解决方法	

NEWTONSOFT.JSON	SYSTEM.TEXT.JSON	
`ObjectCreationHandling` 全局设置	⚠ 不受支持，解决方法	
在不带 setter 的情况下添加到集合	⚠ 不受支持，解决方法	
`ReferenceLoopHandling` 全局设置	✕ 不受支持	
支持 `System.Runtime.Serialization` 特性	✕ 不受支持	
`MissingMemberHandling` 全局设置	✕ 不受支持	
允许不带引号的属性名称	✕ 不受支持	
字符串值前后允许单引号	✕ 不受支持	
对字符串属性允许非字符串 JSON 值	✕ 不受支持	

NEWTONSOFT.JSON	SYSTEM.TEXT.JSON
默认情况下不区分大小写的反序列化	✔ PropertyNameCaseInsensitive 全局设置
Camel 大小写属性名称	✔ PropertyNamingPolicy 全局设置
最小字符转义	✔ 严格字符转义，可配置
`NullValueHandling.Ignore` 全局设置	✔ IgnoreNullValues 全局选项
允许注释	✔ ReadCommentHandling 全局设置
允许尾随逗号	✔ AllowTrailingCommas 全局设置
自定义转换器注册	✔ 优先级顺序不同
默认情况下无最大深度	✔ 默认最大深度为 64，可配置
支持范围广泛的类型	⚠ 某些类型需要自定义转换器
将字符串反序列化为数字	⚠ 不受支持，解决方法，示例
反序列化具有非字符串键的 `Dictionary`	⚠ 不受支持，解决方法，示例
多态序列化	⚠ 不受支持，解决方法，示例
多态反序列化	⚠ 不受支持，解决方法，示例
将推断类型反序列化为 `object` 属性	⚠ 不受支持，解决方法，示例
将 JSON `null` 文本反序列化为不可为 null 的值类型	⚠ 不受支持，解决方法，示例

NEWTONSOFT.JSON	SYSTEM.TEXT.JSON
反序列化为不可变类和结构	⚠ 不受支持，解决方法，示例
`[JsonConstructor]` 特性	⚠ 不受支持，解决方法，示例
`[JsonProperty]` 特性上的 `Required` 设置	⚠ 不受支持，解决方法，示例
`[JsonProperty]` 特性上的 `NullValueHandling` 设置	⚠ 不受支持，解决方法，示例
`[JsonProperty]` 特性上的 `DefaultValueHandling` 设置	⚠ 不受支持，解决方法，示例
`DefaultValueHandling` 全局设置	⚠ 不受支持，解决方法，示例
`DefaultContractResolver` 用于忽略属性	⚠ 不受支持，解决方法，示例
`DateTimeZoneHandling`、`DateFormatString` 设置	⚠ 不受支持，解决方法，示例
回调	⚠ 不受支持，解决方法，示例
支持公共和非公共字段	⚠ 不受支持，解决方法
支持非公共属性资源库和 Getter	⚠ 不受支持，解决方法
`JsonConvert.PopulateObject` 方法	⚠ 不受支持，解决方法
`ObjectCreationHandling` 全局设置	⚠ 不受支持，解决方法
在不带 setter 的情况下添加到集合	⚠ 不受支持，解决方法
`PreserveReferencesHandling` 全局设置	✕ 不受支持
`ReferenceLoopHandling` 全局设置	✕ 不受支持
支持 `System.Runtime.Serialization` 特性	✕ 不受支持
`MissingMemberHandling` 全局设置	✕ 不受支持
允许不带引号的属性名称	✕ 不受支持
字符串值前后允许单引号	✕ 不受支持
对字符串属性允许非字符串 JSON 值	✕ 不受支持

这不是 `Newtonsoft.Json` 功能的详尽列表。此列表包含在 GitHub 问题或 StackOverflow 文章中请求的许多方案。如果对此处所列且当前没有示例代码的一个方案实现了解决方法，并且如果要共享解决方案，请在本页底部的“反馈”部分选择“此页面”。这会在本文档的 GitHub 存储库中创建一个问题，并将它也列在此页面上的“反馈”部分中。

默认 JsonSerializer 行为相较于 Newtonsoft.Json 的差异

System.Text.Json 在默认情况下十分严格，避免代表调用方进行任何猜测或解释，强调确定性行为。该库是为了

实现性能和安全性而特意这样设计的。`Newtonsoft.Json` 默认情况下十分灵活。设计中的这种根本差异是默认行为中以下许多特定差异的背后原因。

不区分大小写的反序列化

在反序列化过程中，默认情况下 `Newtonsoft.Json` 进行不区分大小写的属性名称匹配。System.Text.Json 默认值区分大小写，这可提供更好的性能，因为它执行精确匹配。有关如何执行不区分大小写的匹配的信息，请参阅不区分大小写的属性匹配。

如果使用 ASP.NET Core 间接使用 `System.Text.Json`，则无需执行任何操作即可获得类似于 `Newtonsoft.Json` 的行为。ASP.NET Core 在使用 `System.Text.Json` 时，会为 camel 大小写属性名称和不区分大小写的匹配指定设置。

默认情况下，ASP.NET Core 还允许反序列化带引号的数字。

最小字符转义

在序列化过程中，`Newtonsoft.Json` 对于让字符通过而不进行转义相对宽松。也就是说，它不会将它们替换为 `\uxxxx`（其中 `xxxx` 是字符的码位）。对字符进行转义时，它会通过在字符前发出 `\` 来实现此目的（例如，`"` 会变为 `\"`）。System.Text.Json 会在默认情况下转义较多字符，以对跨站点脚本 (XSS) 或信息泄露攻击提供深度防御保护，并使用六字符序列执行此操作。`System.Text.Json` 会在默认情况下转义所有非 ASCII 字符，因此如果在 `Newtonsoft.Json` 中使用 `StringEscapeHandling.EscapeNonAscii`，则无需执行任何操作。`System.Text.Json` 在默认情况下还会转义 HTML 敏感字符。有关如何替代默认 `System.Text.Json` 行为的信息，请参阅自定义字符编码。

注释

在反序列化过程中，`Newtonsoft.Json` 在默认情况下会忽略 JSON 中的注释。System.Text.Json 默认值是对注释引发异常，因为 RFC 8259 规范不包含它们。有关如何允许注释的信息，请参阅允许注释和尾随逗号。

尾随逗号

在反序列化过程中，默认情况下 `Newtonsoft.Json` 会忽略尾随逗号。它还会忽略多个尾随逗号（例如 `[{"Color":"Red"},{"Color":"Green"},,]`）。System.Text.Json 默认值是对尾随逗号引发异常，因为 RFC 8259 规范不允许使用它们。有关如何使 `System.Text.Json` 接受它们的信息，请参阅允许注释和尾随逗号。无法允许多个尾随逗号。

转换器注册优先级

自定义转换器的 `Newtonsoft.Json` 注册优先级如下所示：

- 属性上的特性
- 类型上的特性
- 转换器 集合

此顺序意味着 `Converters` 集合中的自定义转换器会由通过在类型级别应用特性而注册的转换器替代。这两个注册都会由属性级别的特性替代。

自定义转换器的 System.Text.Json 注册优先级是不同的：

- 属性上的特性
- Converters 集合
- 类型上的特性

此处的差别在于 `Converters` 集合中的自定义转换器会替代类型级别的特性。此优先级顺序的目的是使运行时更改替代设计时选项。无法更改优先级。

有关自定义转换器注册的详细信息，请参阅注册自定义转换器。

最大深度

`Newtonsoft.Json` 默认情况下没有最大深度限制。对于 System.Text.Json，默认限制为 64，可通过设置 JsonSerializerOptions.MaxDepth 进行配置。

如果使用 ASP.NET Core 时间接使用 `System.Text.Json`，则默认的最大深度限制为 32。默认值与模型绑定的默认值相同，并且在 JsonOptions 类中设置。

JSON 字符串（属性名称和字符串值）

在反序列化过程中，`Newtonsoft.Json` 接受用双引号、单引号括起来或不带引号的属性名称。它接受用双引号或单引号括起来的字符串值。例如，`Newtonsoft.Json` 接受以下 JSON：

```
{
  "name1": "value",
  'name2': "value",
  name3: 'value'
}
```

`System.Text.Json` 仅接受双引号中的属性名称和字符串值，因为 RFC 8259 规范要求使用该格式，这是唯一视为有效 JSON 的格式。

用单引号括起来的值会导致 JsonException，并出现以下消息：

```
''' is an invalid start of a value.
```

字符串属性的非字符串值

`Newtonsoft.Json` 接受非字符串值（如数字或文本 `true` 和 `false`），以便反序列化为类型字符串的属性。下面是 `Newtonsoft.Json` 成功反序列化为以下类的 JSON 示例：

```
{
  "String1": 1,
  "String2": true,
  "String3": false
}
```

```
public class ExampleClass
{
    public string String1 { get; set; }
    public string String2 { get; set; }
    public string String3 { get; set; }
}
```

`System.Text.Json` 不将非字符串值反序列化为字符串属性。字符串字段接收的非字符串值会导致 JsonException，并出现以下消息：

```
The JSON value could not be converted to System.String.
```

使用 JsonSerializer 的方案

下面一部分方案不受内置功能支持，但有解决方法可用。解决方法是自定义转换器，它们可能无法提供与 `Newtonsoft.Json` 功能完全相同的功能。对于其中一些功能，提供示例代码作为示例。如果你依赖于这些 `Newtonsoft.Json` 功能，迁移需要修改 .NET 对象模型或进行其他代码更改。

对于下面的一部分方案，解决方法不可行或无法提供。如果你依赖于这些 `Newtonsoft.Json` 功能，则无法在不进行重大更改的情况下进行迁移。

允许或写入带引号的数字

`Newtonsoft.Json` 可以序列化或反序列化由 JSON 字符串表示的数字(括在引号中)。例如，它可以接受 `{"DegreesCelsius":"23"}` 而不是 `{"DegreesCelsius":23}`。若要在 System.Text.Json 中启用该行为，请将 JsonSerializerOptions.NumberHandling 设置为 WriteAsString 或 AllowReadingFromString，或使用 [JsonNumberHandling] 特性。

如果使用 ASP.NET Core 间接使用 `System.Text.Json`，则无需执行任何操作即可获得类似于 `Newtonsoft.Json` 的行为。ASP.NET Core 在使用 `System.Text.Json` 时指定 Web 默认值，Web 默认值允许带引号的数字。

有关详细信息，请参阅允许或写入带引号的数字。

`Newtonsoft.Json` 可以序列化或反序列化由 JSON 字符串表示的数字(括在引号中)。例如，它可以接受 `{"DegreesCelsius":"23"}` 而不是 `{"DegreesCelsius":23}`。若要在 .NET Core 3.1 的 System.Text.Json 中启用该行为，请实现类似于以下示例的自定义转换器。该转换器处理定义为 `long` 的属性：

- 它将这些属性序列化为 JSON 字符串。
- 它在反序列化期间接受 JSON 数字和括在引号中的数字。

```
using System;
using System.Buffers;
using System.Buffers.Text;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class LongToStringConverter : JsonConverter<long>
    {
        public override long Read(
            ref Utf8JsonReader reader, Type type, JsonSerializerOptions options)
        {
            if (reader.TokenType == JsonTokenType.String)
            {
                ReadOnlySpan<byte> span =
                    reader.HasValueSequence ? reader.ValueSequence.ToArray() : reader.ValueSpan;

                if (Utf8Parser.TryParse(span, out long number, out int bytesConsumed) &&
                    span.Length == bytesConsumed)
                {
                    return number;
                }

                if (long.TryParse(reader.GetString(), out number))
                {
                    return number;
                }
            }

            return reader.GetInt64();
        }

        public override void Write(
            Utf8JsonWriter writer, long longValue, JsonSerializerOptions options) =>
            writer.WriteStringValue(longValue.ToString());
    }
}
```

通过对各个 `long` 属性使用特性或是通过向 集合添加转换器Converters来注册此自定义转换器。

指定要在反序列化时使用的构造函数

使用 `Newtonsoft.Json` `[JsonConstructor]` 特性可以指定在反序列化为 POCO 时要调用的构造函数。

`System.Text.Json` 还具有 [JsonConstructor] 特性。有关详细信息，请参阅不可变类型和记录。

.NET Core 3.1 中的 System.Text.Json 仅支持无参数构造函数。作为一种解决方法，可以在自定义转换器中调用所需的任何构造函数。请参阅反序列化为不可变类和结构的示例。

有条件地忽略属性

`Newtonsoft.Json` 有多种方法可在序列化或反序列化时有条件地忽略属性：

- `DefaultContractResolver` 使你可以基于任意条件选择要包含或忽略的属性。
- `JsonSerializerSettings` 上的 `NullValueHandling` 和 `DefaultValueHandling` 设置使你指定应忽略所有 null 值或默认值属性。
- `[JsonProperty]` 特性上的 `NullValueHandling` 和 `DefaultValueHandling` 设置使你可以指定在设置为 null 或默认值时应忽略的单个属性。

System.Text.Json 提供以下方法，用于在序列化期间忽略属性或字段：

- 属性上的 [JsonIgnore] 特性会导致在序列化过程中从 JSON 中省略属性。
- IgnoreReadOnlyProperties 全局选项使你可以忽略所有只读属性。
- 如果你包含字段，则 JsonSerializerOptions.IgnoreReadOnlyFields 全局选项使你可以忽略所有只读字段。
- 使用 `DefaultIgnoreCondition` 全局选项，你可以忽略具有默认值的所有值类型属性，或忽略具有 null 值的所有引用类型属性。

.NET Core 3.1 中的 System.Text.Json 提供以下方法，用于在序列化期间忽略属性：

- 属性上的 [JsonIgnore] 特性会导致在序列化过程中从 JSON 中省略属性。
- IgnoreNullValues 全局选项使你可以忽略所有 null 值属性。
- IgnoreReadOnlyProperties 全局选项使你可以忽略所有只读属性。

不能通过这些选项：

- 基于运行时计算的任意条件忽略所选属性。
- 忽略具有类型的默认值的所有属性。
- 忽略具有类型的默认值的所选属性。
- 忽略值为 null 的所选属性。
- 基于运行时计算的任意条件忽略所选属性。

对于该功能，可以编写自定义转换器。下面是一个示例 POCO 和一个适用于它的自定义转换器，用于说明此方法：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class WeatherForecastRuntimeIgnoreConverter : JsonConverter<WeatherForecast>
    {
        public override WeatherForecast Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
```

```
            JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            }

            var wf = new WeatherForecast();

            while (reader.Read())
            {
                if (reader.TokenType == JsonTokenType.EndObject)
                {
                    return wf;
                }

                if (reader.TokenType == JsonTokenType.PropertyName)
                {
                    string propertyName = reader.GetString();
                    reader.Read();
                    switch (propertyName)
                    {
                        case "Date":
                            DateTimeOffset date = reader.GetDateTimeOffset();
                            wf.Date = date;
                            break;
                        case "TemperatureCelsius":
                            int temperatureCelsius = reader.GetInt32();
                            wf.TemperatureCelsius = temperatureCelsius;
                            break;
                        case "Summary":
                            string summary = reader.GetString();
                            wf.Summary = string.IsNullOrWhiteSpace(summary) ? "N/A" : summary;
                            break;
                    }
                }
            }

            throw new JsonException();
        }

        public override void Write(Utf8JsonWriter writer, WeatherForecast wf, JsonSerializerOptions options)
        {
            writer.WriteStartObject();

            writer.WriteString("Date", wf.Date);
            writer.WriteNumber("TemperatureCelsius", wf.TemperatureCelsius);
            if (!string.IsNullOrWhiteSpace(wf.Summary) && wf.Summary != "N/A")
            {
                writer.WriteString("Summary", wf.Summary);
            }

            writer.WriteEndObject();
        }
    }
}
```

如果值为 null、空字符串或 "N/A"，则转换器会导致从序列化中省略 `Summary` 属性。

通过对类使用特性或是通过向 Converters 集合添加转换器来注册此自定义转换器。

此方法在以下情况下需要其他逻辑：

- POCO 包含复杂属性。
- 需要处理特性（如 `[JsonIgnore]`）或选项（如自定义编码器）。

公共和非公共字段

`Newtonsoft.Json` 可以序列化和反序列化字段以及属性。

在 System.Text.Json 中，在序列化或反序列化时，使用 JsonSerializerOptions.IncludeFields 全局设置或 [JsonInclude] 特性来包含公共字段。有关示例，请参阅包含字段。

.NET Core 3.1 中的 System.Text.Json 仅适用于公共属性。自定义转换器可提供此功能。

保留对象引用并处理循环

默认情况下，`Newtonsoft.Json` 按值进行序列化。例如，如果对象包含两个属性，而这些属性包含对同一个 `Person` 对象的引用，该 `Person` 对象属性的值会在 JSON 重复。

`Newtonsoft.Json` 在 `JsonSerializerSettings` 上有一个 `PreserveReferencesHandling` 设置，可让你按引用进行序列化：

- 标识符元数据会添加到为第一个 `Person` 对象创建的 JSON。
- 为第二个 `Person` 对象创建的 JSON 包含对该标识符（而不是属性值）的引用。

`Newtonsoft.Json` 还具有一个 `ReferenceLoopHandling` 设置，使你可以忽略循环引用，而不是引发异常。

若要在 System.Text.Json 中保留引用并处理循环引用，请将 JsonSerializerOptions.ReferenceHandler 设置为 Preserve。`ReferenceHandler.Preserve` 设置等效于 `Newtonsoft.Json` 中的 `PreserveReferencesHandling` = `PreserveReferencesHandling.All`。

与 NewtonsoftJson ReferenceResolver 一样，System.Text.Json.Serialization.ReferenceResolver 类定义在序列化和反序列化过程中保留引用的行为。创建派生类以指定自定义行为。有关示例，请参阅 GuidReferenceResolver。

一些相关的 `Newtonsoft.Json` 功能不受支持：

- JsonPropertyAttribute.IsReference
- JsonPropertyAttribute.ReferenceLoopHandling
- JsonSerializerSettings.ReferenceLoopHandling

有关详细信息，请参阅保留引用并处理循环引用。

.NET Core 3.1 中的 System.Text.Json 仅支持按值进行进行序列化，并对循环引用引发异常。

包含非字符串键的字典

`Newtonsoft.Json` 和 `System.Text.Json` 都支持 `Dictionary<TKey, TValue>` 类型的集合。

`Newtonsoft.Json` 支持类型 `Dictionary<TKey, TValue>` 的集合。.NET Core 3.1 的 System.Text.Json 中对字典集合的内置支持仅限于 `Dictionary<string, TValue>`。即，键必须是字符串。

若要在 .NET Core 3.1 中支持将整数或某种其他类型用作键的字典，请创建转换器（类似于如何编写自定义转换器中的示例）。

没有内置支持的类型

System.Text.Json 不为以下类型提供内置支持：

- DataTable 和相关类型
- F# 类型（如可区分联合、记录类型和匿名记录类型）。
- ExpandoObject
- TimeZoneInfo
- BigInteger
- TimeSpan
- DBNull

- Type
- ValueTuple 及其关联泛型类型

对于没有内置支持的类型，可以实现自定义转换器。

多态序列化

`Newtonsoft.Json` 会自动执行多态序列化。有关 System.Text.Json 的有限多态序列化功能的信息，请参阅序列化派生类的属性。

所述的解决方法是定义可能以类型 `object` 的形式包含派生类的属性。如果无法这样，则另一种选择是为整个继承类型层次结构创建带有 `Write` 方法的转换器（类似于如何编写自定义转换器中的示例）。

多态反序列化

`Newtonsoft.Json` 具有 `TypeNameHandling` 设置，它在序列化期间将类型名称元数据添加到 JSON。它在反序列化期间使用元数据执行多态反序列化。System.Text.Json 可以执行有限范围的多态序列化，但不能执行多态反序列化。

若要支持多态反序列化，请创建转换器（类似于如何编写自定义转换器中的示例）。

对象属性的反序列化

当 `Newtonsoft.Json` 反序列化为 Object 时，它会：

- 推断 JSON 有效负载中的基元值的类型（不是 `null`），并以装箱对象的形式返回存储的 `string`、`long`、`double`、`boolean` 或 `DateTime`。基元值是单个 JSON 值，如 JSON 数字、字符串、`true`、`false` 或 `null`。
- 为 JSON 有效负载中的复杂值返回 `JObject` 或 `JArray`。复杂值是括在大括号 (`{}`) 中的 JSON 键值对的集合或括在方括号 (`[]`) 中的值的列表。括在大括号或方括号中的属性和值可以具有附加属性或值。
- 当有效负载具有 `null` JSON 文本时，返回空引用。

System.Text.Json 在每次反序列化为 Object 时，为基元和复数值存储装箱 `JsonElement`，例如：

- `object` 属性。
- `object` 字典值。
- `object` 数组值。
- 根 `object`。

但是，`System.Text.Json` 处理 `null` 的方式与 `Newtonsoft.Json` 相同，会在有效负载中包含 `null` JSON 文本时返回空引用。

若要为 `object` 实现类型推理，请创建转换器（类似于如何编写自定义转换器中的示例）。

将 null 反序列化为不可为 null 的类型

`Newtonsoft.Json` 在以下方案中不会引发异常：

- `NullValueHandling` 设置为 `Ignore`，并且
- 在反序列化过程中，JSON 对于不可为 null 的值类型包含 null 值。

在相同方案中，System.Text.Json 会引发异常。（`System.Text.Json` 中对应的 null 处理设置为 JsonSerializerOptions.IgnoreNullValues = `true`。）

如果你拥有目标类型，在最佳解决方法是使相关属性可为 null（例如，将 `int` 更改为 `int?`）。

另一种解决方法是为类型创建转换器，如以下为 `DateTimeOffset` 类型处理 null 值的示例：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetNullHandlingConverter : JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            reader.TokenType == JsonTokenType.Null
                ? default
                : reader.GetDateTimeOffset();

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(dateTimeValue);
    }
}
```

通过对属性使用特性或是通过向 Converters 集合添加转换器来注册此自定义转换器。

注意：前面的转换器处理 null 值的方式与 `Newtonsoft.Json` 为指定默认值的 POCO 进行处理的方式不同。例如，假设以下代码表示目标对象：

```
public class WeatherForecastWithDefault
{
    public WeatherForecastWithDefault()
    {
        Date = DateTimeOffset.Parse("2001-01-01");
        Summary = "No summary";
    }
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

并且假设使用前面的转换器反序列化以下 JSON：

```
{
  "Date": null,
  "TemperatureCelsius": 25,
  "Summary": null
}
```

反序列化之后，`Date` 属性具有 1/1/0001 (`default(DateTimeOffset)`)，即，在构造函数中设置的值会被覆盖。给定相同 POCO 和 JSON，`Newtonsoft.Json` 反序列化会将 1/1/2001 保留在 `Date` 属性中。

反序列化为不可变类和结构

`Newtonsoft.Json` 可以反序列化为不可变类和结构，因为它可以使用具有参数的构造函数。

在 System.Text.Json 中，使用 [JsonConstructor] 特性来指定参数化构造函数的用法。C# 9 记录也是不可变的，并且支持作为反序列化目标。有关详细信息，请参阅不可变类型和记录。

.NET Core 3.1 中的 System.Text.Json 仅支持公共无参数构造函数。作为一种解决方法，可以在自定义转换器中调用具有参数的构造函数。

下面是具有多个构造函数参数的不可变结构：

```
public readonly struct ImmutablePoint
{
    public ImmutablePoint(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; }
    public int Y { get; }
}
```

下面是序列化和反序列化此结构的转换器：

```
using System;
using System.Diagnostics;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class ImmutablePointConverter : JsonConverter<ImmutablePoint>
    {
        private readonly JsonEncodedText _xName = JsonEncodedText.Encode("X");
        private readonly JsonEncodedText _yName = JsonEncodedText.Encode("Y");

        private readonly JsonConverter<int> _intConverter;

        public ImmutablePointConverter(JsonSerializerOptions options) =>
            _intConverter = options?.GetConverter(typeof(int)) is JsonConverter<int> intConverter
                ? intConverter
                : throw new InvalidOperationException();

        public override ImmutablePoint Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            };

            int? x = default;
            int? y = default;

            // Get the first property.
            reader.Read();
            if (reader.TokenType != JsonTokenType.PropertyName)
            {
                throw new JsonException();
            }

            if (reader.ValueTextEquals(_xName.EncodedUtf8Bytes))
            {
                x = ReadProperty(ref reader, options);
            }
            else if (reader.ValueTextEquals(_yName.EncodedUtf8Bytes))
            {
                y = ReadProperty(ref reader, options);
            }
            else
            {
                throw new JsonException();
```

```
            }

            // Get the second property.
            reader.Read();
            if (reader.TokenType != JsonTokenType.PropertyName)
            {
                throw new JsonException();
            }

            if (x.HasValue && reader.ValueTextEquals(_yName.EncodedUtf8Bytes))
            {
                y = ReadProperty(ref reader, options);
            }
            else if (y.HasValue && reader.ValueTextEquals(_xName.EncodedUtf8Bytes))
            {
                x = ReadProperty(ref reader, options);
            }
            else
            {
                throw new JsonException();
            }

            reader.Read();

            if (reader.TokenType != JsonTokenType.EndObject)
            {
                throw new JsonException();
            }

            return new ImmutablePoint(x.GetValueOrDefault(), y.GetValueOrDefault());
        }

        private int ReadProperty(ref Utf8JsonReader reader, JsonSerializerOptions options)
        {
            Debug.Assert(reader.TokenType == JsonTokenType.PropertyName);

            reader.Read();
            return _intConverter.Read(ref reader, typeof(int), options);
        }

        private void WriteProperty(Utf8JsonWriter writer, JsonEncodedText name, int intValue,
JsonSerializerOptions options)
        {
            writer.WritePropertyName(name);
            _intConverter.Write(writer, intValue, options);
        }

        public override void Write(
            Utf8JsonWriter writer,
            ImmutablePoint point,
            JsonSerializerOptions options)
        {
            writer.WriteStartObject();
            WriteProperty(writer, _xName, point.X, options);
            WriteProperty(writer, _yName, point.Y, options);
            writer.WriteEndObject();
        }
    }
}
```

通过向 Converters 集合添加转换器来注册此自定义转换器。

有关处理开放式泛型属性的类似转换器的示例，请参阅用于键/值对的内置转换器。

必需的属性

在 `Newtonsoft.Json` 中，通过对 `[JsonProperty]` 特性设置 `Required` 来指定属性是必需的。如果在 JSON 中没

有为标记为必需的属性收到值，`Newtonsoft.Json` 会引发异常。

如果没有为目标类型的某个属性收到值，System.Text.Json 不会引发异常。例如，如果具有 `WeatherForecast` 类：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

以下 JSON 可反序列化，不会发生错误：

```
{
    "TemperatureCelsius": 25,
    "Summary": "Hot"
}
```

若要使反序列化在 JSON 中没有 `Date` 属性时失败，请实现自定义转换器。如果反序列化完成之后未设置 `Date` 属性，则以下示例转换器代码会引发异常：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class WeatherForecastRequiredPropertyConverter : JsonConverter<WeatherForecast>
    {
        public override WeatherForecast Read(
            ref Utf8JsonReader reader,
            Type type,
            JsonSerializerOptions options)
        {
            // Don't pass in options when recursively calling Deserialize.
            WeatherForecast forecast = JsonSerializer.Deserialize<WeatherForecast>(ref reader);

            // Check for required fields set by values in JSON
            return forecast.Date == default
                ? throw new JsonException("Required property not received in the JSON")
                : forecast;
        }

        public override void Write(
            Utf8JsonWriter writer,
            WeatherForecast forecast, JsonSerializerOptions options)
        {
            // Don't pass in options when recursively calling Serialize.
            JsonSerializer.Serialize(writer, forecast);
        }
    }
}
```

通过向 JsonSerializerOptions.Converters 集合添加转换器来注册此自定义转换器。

这种以递归方式调用转换器的模式要求使用 JsonSerializerOptions 而不是使用属性注册转换器。如果使用属性注册转换器，则自定义转换器将以递归方式调入其自身。结果是一个以堆栈溢出异常结尾的无限循环。

使用选项对象注册转换器时，请通过在以递归方式调用 Serialize 或 Deserialize 时不传入选项对象来避免无限循环。选项对象包含 Converters 集合。如果将它传递给 `Serialize` 或 `Deserialize`，则自定义转换器会调入其自身，从而产生导致堆栈溢出异常的无限循环。如果默认选项不可行，请使用所需设置创建选项的新实例。此方法

会速度较慢，因为每个新实例都会独立缓存。

有一种替代模式，可在要转换的类上使用 `JsonConverterAttribute` 注册。在此方法中，转换器代码对派生自要转换的类的类调用 `Serialize` 或 `Deserialize`。派生类没有应用 `JsonConverterAttribute`。在此替代的以下示例中：

- `WeatherForecastWithRequiredPropertyConverterAttribute` 是要进行反序列化并应用 `JsonConverterAttribute` 的类。
- `WeatherForecastWithoutRequiredPropertyConverterAttribute` 是不具有转换器属性的派生类。
- 转换器中的代码调用 `WeatherForecastWithoutRequiredPropertyConverterAttribute` 上的 `Serialize` 和 `Deserialize` 以避免无限循环。此方法对于序列化是一种性能开销，因为需要实例化额外的对象和复制属性值。

`WeatherForecast*` 类型如下：

```
[JsonConverter(typeof(WeatherForecastRequiredPropertyConverterForAttributeRegistration))]
public class WeatherForecastWithRequiredPropertyConverterAttribute
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}

public class WeatherForecastWithoutRequiredPropertyConverterAttribute :
    WeatherForecastWithRequiredPropertyConverterAttribute
{
}
```

下面是转换器：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class WeatherForecastRequiredPropertyConverterForAttributeRegistration :
        JsonConverter<WeatherForecastWithRequiredPropertyConverterAttribute>
    {
        public override WeatherForecastWithRequiredPropertyConverterAttribute Read(
            ref Utf8JsonReader reader,
            Type type,
            JsonSerializerOptions options)
        {
            // OK to pass in options when recursively calling Deserialize.
            WeatherForecastWithRequiredPropertyConverterAttribute forecast =
                JsonSerializer.Deserialize<WeatherForecastWithoutRequiredPropertyConverterAttribute>(
                    ref reader,
                    options);

            // Check for required fields set by values in JSON.
            return forecast.Date == default
                ? throw new JsonException("Required property not received in the JSON")
                : forecast;
        }

        public override void Write(
            Utf8JsonWriter writer,
            WeatherForecastWithRequiredPropertyConverterAttribute forecast,
            JsonSerializerOptions options)
        {
            var weatherForecastWithoutConverterAttributeOnClass =
                new WeatherForecastWithoutRequiredPropertyConverterAttribute
                {
                    Date = forecast.Date,
                    TemperatureCelsius = forecast.TemperatureCelsius,
                    Summary = forecast.Summary
                };

            // OK to pass in options when recursively calling Serialize.
            JsonSerializer.Serialize(
                writer,
                weatherForecastWithoutConverterAttributeOnClass,
                options);
        }
    }
}
```

如果需要处理特性(例如 [JsonIgnore])或不同选项(如自定义编码器),必需的属性转换器需要其他逻辑。此外,示例代码不处理在构造函数中为其设置了默认值的属性。而且此方法不区分以下情况:

- JSON 中缺少属性。
- JSON 中存在不可为 null 的类型的属性,但值是该类型的默认值,如 `int` 的值为零。
- JSON 中存在可为 null 的值类型的属性,但值为 null。

指定日期格式

`Newtonsoft.Json` 提供多种方法来控制如何序列化和反序列化 `DateTime` 和 `DateTimeOffset` 类型的属性:

- `DateTimeZoneHandling` 设置可用于将所有 `DateTime` 值序列化为 UTC 日期。
- `DateFormatString` 设置和 `DateTime` 转换器可用于自定义日期字符串的格式。

System.Text.Json 支持 ISO 8601-1:2019,包括 RFC 3339 配置文件。此格式被广泛采用,无歧义,并且精确地进行往返。若要使用任何其他格式,请创建自定义转换器。有关详细信息,请参阅 System.Text.Json 中的 DateTime

和 DateTimeOffset 支持。

回调

`Newtonsoft.Json` 使你可以在序列化或反序列化过程中的多个点执行自定义代码：

- OnDeserializing（开始反序列化对象时）
- OnDeserialized（对象反序列化完成时）
- OnSerializing（开始序列化对象时）
- OnSerialized（对象序列化完成时）

在 System.Text.Json 中，可以通过编写自定义转换器来模拟回调。以下示例演示适用于 POCO 的自定义转换器。该转换器包含在与 `Newtonsoft.Json` 回调相对应的每个点显示消息的代码。

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class WeatherForecastCallbacksConverter : JsonConverter<WeatherForecast>
    {
        public override WeatherForecast Read(
            ref Utf8JsonReader reader,
            Type type,
            JsonSerializerOptions options)
        {
            // Place "before" code here (OnDeserializing),
            // but note that there is no access here to the POCO instance.
            Console.WriteLine("OnDeserializing");

            // Don't pass in options when recursively calling Deserialize.
            WeatherForecast forecast = JsonSerializer.Deserialize<WeatherForecast>(ref reader);

            // Place "after" code here (OnDeserialized)
            Console.WriteLine("OnDeserialized");

            return forecast;
        }

        public override void Write(
            Utf8JsonWriter writer,
            WeatherForecast forecast, JsonSerializerOptions options)
        {
            // Place "before" code here (OnSerializing)
            Console.WriteLine("OnSerializing");

            // Don't pass in options when recursively calling Serialize.
            JsonSerializer.Serialize(writer, forecast);

            // Place "after" code here (OnSerialized)
            Console.WriteLine("OnSerialized");
        }
    }
}
```

通过向 Converters 集合添加转换器来注册此自定义转换器。

如果使用遵循前面示例的自定义转换器：

- `OnDeserializing` 代码无权访问新 POCO 实例。若要在反序列化开始时操作新 POCO 实例，请将该代码放入 POCO 构造函数中。
- 通过在选项对象中注册转换器而在以递归方式调用 `Serialize` 或 `Deserialize` 时不传入选项对象，避免无限

循环。

若要详细了解递归调用 `Serialize` 或 `Deserialize` 的自定义转换器，请参阅本文前面的必需属性部分。

非公共属性资源库和 Getter

`Newtonsoft.Json` 可以通过 `JsonProperty` 特性使用私有和内部属性 setter 和 getter。

System.Text.Json 支持通过 [JsonInclude] 特性使用私有和内部属性资源库和 Getter。有关示例代码，请参阅非公共属性访问器。

.NET Core 3.1 中的 System.Text.Json 仅支持公共资源库。自定义转换器可提供此功能。

填充现有对象

`Newtonsoft.Json` 中的 `JsonConvert.PopulateObject` 方法将 JSON 文档反序列化为类的现有实例，而不是创建新实例。System.Text.Json 始终使用默认公共无参数构造函数创建目标类型的新实例。自定义转换器可以反序列化为现有实例。

重用而不是替换属性

`Newtonsoft.Json` `ObjectCreationHandling` 设置使你可以指定在反序列化过程中应重用属性中的对象，而不是进行替换。System.Text.Json 始终替换属性中的对象。自定义转换器可提供此功能。

在不带 setter 的情况下添加到集合

在反序列化过程中，`Newtonsoft.Json` 会将对象添加到集合，即使属性没有 setter。System.Text.Json 会忽略没有 setter 的属性。自定义转换器可提供此功能。

System.Runtime.Serialization 特性

System.Text.Json 不支持 `System.Runtime.Serialization` 命名空间中的特性，如 `DataMemberAttribute` 和 `IgnoreDataMemberAttribute`。

八进制数字

`Newtonsoft.Json` 将带前导零的数字视为八进制数字。System.Text.Json 不允许存在前导零，因为 RFC 8259 规范不允许。

MissingMemberHandling

`Newtonsoft.Json` 可以配置为在 JSON 包含目标类型中缺少的属性时，在反序列化过程中引发异常。System.Text.Json 会忽略 JSON 中的额外属性，但在使用 [JsonExtensionData] 特性时除外。对于缺少成员功能，没有解决方法。

TraceWriter

`Newtonsoft.Json` 使你可以使用 `TraceWriter` 进行调试，以查看序列化或反序列化所生成的日志。System.Text.Json 不执行日志记录。

与 JToken（如 JObject、JArray）相比的 JsonDocument 和 JsonElement

System.Text.Json.JsonDocument 提供从现有 JSON 有效负载分析和生成只读文档对象模型 (DOM) 的功能。DOM 提供对 JSON 有效负载中的数据的随机访问。可以通过 JsonElement 类型访问构成有效负载的 JSON 元素。`JsonElement` 类型提供用于将 JSON 文本转换为常见 .NET 类型的 API。`JsonDocument` 公开了 RootElement 属性。

JsonDocument 为 IDisposable

`JsonDocument` 将内存中的数据视图生成到共用缓冲区中。因此，与 `Newtonsoft.Json` 中的 `JObject` 或 `JArray` 不同，`JsonDocument` 类型实现 `IDisposable` 并且需要在 using 块中使用。

如果要将生存期所有权和释放责任转移到调用方，则只需从 API 返回 `JsonDocument`。在大多数情况下，这不是必需的。如果调用方需要处理整个 JSON 文档，则返回 RootElement 的 Clone，这是 JsonElement。如果调用方

需要处理 JSON 文档中的特定元素，则返回该 JsonElement 的 Clone。如果在不进行 `Clone` 的情况下直接返回 `RootElement` 或子元素，则在释放拥有返回的 `JsonElement` 的 `JsonDocument` 之后，调用方将无法访问它。

下面是一个要求你进行 `Clone` 的示例：

```
public JsonElement LookAndLoad(JsonElement source)
{
    string json = File.ReadAllText(source.GetProperty("fileName").GetString());

    using (JsonDocument doc = JsonDocument.Parse(json))
    {
        return doc.RootElement.Clone();
    }
}
```

前面的代码需要包含 `fileName` 属性的 `JsonElement`。它会打开 JSON 文件并创建一个 `JsonDocument`。该方法假设调用方要处理整个文档，因此会返回 `RootElement` 的 `Clone`。

如果收到 `JsonElement` 并要返回子元素，则无需返回子元素的 `Clone`。调用方负责使传入的 `JsonElement` 所属的 `JsonDocument` 保持活动状态。例如：

```
public JsonElement ReturnFileName(JsonElement source)
{
   return source.GetProperty("fileName");
}
```

JsonDocument 为只读

System.Text.Json DOM 无法添加、删除或修改 JSON 元素。它这样设计是为了实现性能，并减少用于分析常见 JSON 有效负载大小（即 < 1 MB）的分配。如果你的方案当前使用可修改的 DOM，则以下解决方法之一可能是可行的：

- 若要从头开始生成 `JsonDocument`（即，不将现有 JSON 有效负载传入到 `Parse` 方法），请使用 `Utf8JsonWriter` 编写 JSON 文本，并分析这样做的输出以创建新 `JsonDocument`。
- 若要修改现有 `JsonDocument`，请使用它编写 JSON 文本（在编写时进行更改），并分析这样做的输出以创建新 `JsonDocument`。
- 若要合并现有 JSON 文档（与 `Newtonsoft.Json` 中的 `JObject.Merge` 或 `JContainer.Merge` API 等效），请参阅此 GitHub 问题。

JsonElement 是联合结构

`JsonDocument` 将 `RootElement` 公开为类型 JsonElement 的属性，该类型是包含任何 JSON 元素的联合结构类型。`Newtonsoft.Json` 使用专用分层类型，如 `JObject`、`JArray`、`JToken` 等。`JsonElement` 是可以搜索和枚举的内容，你可以使用 `JsonElement` 将 JSON 元素具体化为 .NET 类型。

如何搜索子元素的 JsonDocument 和 JsonElement

使用 `Newtonsoft.Json` 中的 `JObject` 或 `JArray` 搜索 JSON 令牌的速度往往相对较快，因为它们是在某个字典中查找。相比之下，对 `JsonElement` 进行搜索需要对属性进行线性搜索，因此速度相对较慢（例如在使用 `TryGetProperty` 时）。System.Text.Json 旨在最大程度减少初始分析时间，而不是查找时间。因此，在通过 `JsonDocument` 对象搜索时，请使用以下方法优化性能：

- 使用内置枚举器（EnumerateArray 和 EnumerateObject），而不是执行自己的索引或循环。
- 不要使用 `RootElement` 通过每个属性对整个 `JsonDocument` 执行线性搜索。而是基于 JSON 数据的已知结构对嵌套 JSON 对象进行搜索。例如，如果要在 `Student` 对象中查找 `Grade` 属性，请循环访问 `Student` 对象，并获取每个对象的 `Grade` 值，而不是搜索所有 `Grade` 对象来查找 `JsonElement` 属性。执行后者将导致不必要浏览相同数据。

有关代码示例，请参阅使用 JsonDocument 访问数据。

Utf8JsonReader 与 JsonTextReader 的比较

System.Text.Json.Utf8JsonReader 是面向 UTF-8 编码 JSON 文本的一个高性能、低分配的只进读取器，从 ReadOnlySpan<byte> 或 ReadOnlySequence<byte> 读取信息。`Utf8JsonReader` 是一种低级类型，可用于生成自定义分析器和反序列化程序。

以下各节说明使用 `Utf8JsonReader` 的推荐编程模式。

Utf8JsonReader 是 ref struct

由于 `Utf8JsonReader` 类型是 ref struct，因此它具有某些限制。例如，它无法作为字段存储在 ref struct 之外的类或结构中。若要实现高性能，此类型必须为 `ref struct`，因为它需要缓存输入 ReadOnlySpan<byte>（这本身便是 ref struct）。此外，此类型是可变的，因为它包含状态。因此，它按引用传递而不是按值传递。按值传递会产生结构副本，状态更改会对调用方不可见。这与 `Newtonsoft.Json` 不同，因为 `Newtonsoft.Json` `JsonTextReader` 是一个类。有关如何使用 ref struct 的详细信息，请参阅编写安全有效的 C# 代码。

读取 UTF-8 文本

若要在使用 `Utf8JsonReader` 时实现可能的最佳性能，请读取已编码为 UTF-8 文本（而不是 UTF-16 字符串）的 JSON 有效负载。有关代码示例，请参阅使用 Utf8JsonReader 筛选数据。

使用流或 PipeReader 进行读取

`Utf8JsonReader` 支持从 UTF-8 编码的 ReadOnlySpan<byte> 或 ReadOnlySequence<byte>（这是从 PipeReader 读取的结果）进行读取。

对于同步读取，可以读取 JSON 有效负载，直到流的末尾进入字节数组中，并将该数组传递给读取器。若要从字符串（编码为 UTF-16）进行读取，请调用 UTF8.GetBytes 以首先将字符串转码为 UTF-8 编码的字节数组。然后将该数组传递给 `Utf8JsonReader`。

由于 `Utf8JsonReader` 将输入视为 JSON 文本，因此 UTF-8 字节顺序标记 (BOM) 被视为无效 JSON。调用方需要在将数据传递给读取器之前将该标记筛选出来。

有关代码示例，请参阅使用 Utf8JsonReader。

使用多段 ReadOnlySequence 进行读取

如果 JSON 输入是 ReadOnlySpan<byte>，则在运行读取循环时，可以从读取器上的 `ValueSpan` 属性访问每个 JSON 元素。但是，如果输入是 ReadOnlySequence<byte>（这是从 PipeReader 读取的结果），则某些 JSON 元素可能会跨 `ReadOnlySequence<byte>` 对象的多个段。无法在连续内存块中从 ValueSpan 访问这些元素。而是在每次将多段 `ReadOnlySequence<byte>` 作为输入时，轮询读取器上的 HasValueSequence 属性，以确定如何访问当前 JSON 元素。下面是推荐模式：

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        // ...
        ReadOnlySpan<byte> jsonElement = reader.HasValueSequence ?
            reader.ValueSequence.ToArray() :
            reader.ValueSpan;
        // ...
    }
}
```

使用 ValueTextEquals 进行属性名称查找

不要使用 ValueSpan 通过对属性名称查找调用 SequenceEqual 来执行逐字节比较。改为调用 ValueTextEquals，因为该方法会对在 JSON 中转义的任何字符取消转义。下面的示例演示如何搜索名为“name”的属性：

```
private static readonly byte[] s_nameUtf8 = Encoding.UTF8.GetBytes("name");
```

```
while (reader.Read())
{
    switch (reader.TokenType)
    {
        case JsonTokenType.StartObject:
            total++;
            break;
        case JsonTokenType.PropertyName:
            if (reader.ValueTextEquals(s_nameUtf8))
            {
                count++;
            }
            break;
    }
}
```

将 null 值读取到可为 null 的值类型中

`Newtonsoft.Json` 提供返回 Nullable<T> 的 API，如 `ReadAsBoolean`（它通过返回 `bool?` 来处理 `Null` `TokenType`）。内置 `System.Text.Json` API 仅返回不可为 null 的值类型。例如，Utf8JsonReader.GetBoolean 返回 `bool`。如果它在 JSON 中发现 `Null`，则会引发异常。下面的示例演示两种用于处理 null 的方法，一种方法是返回可为 null 的值类型，另一种方法是返回默认值：

```
public bool? ReadAsNullableBoolean()
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
        return null;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType != JsonTokenType.False)
    {
        throw new JsonException();
    }
    return _reader.GetBoolean();
}
```

```
public bool ReadAsBoolean(bool defaultValue)
{
    _reader.Read();
    if (_reader.TokenType == JsonTokenType.Null)
    {
        return defaultValue;
    }
    if (_reader.TokenType != JsonTokenType.True && _reader.TokenType != JsonTokenType.False)
    {
        throw new JsonException();
    }
    return _reader.GetBoolean();
}
```

多目标

如果需要继续为某些目标框架使用 `Newtonsoft.Json`，则可以使用多目标，并具有两种实现。但是，这并非易事，需要进行一些 `#ifdefs` 和源文件复制。共享尽可能多代码的一种方法是围绕 `Utf8JsonReader` 和 `Newtonsoft.Json` `JsonTextReader` 创建 `ref struct` 包装器。此包装器会统一公共外围应用，同时隔离行为差异。这使你可以隔离主要对类型的构造进行的更改，以及按引用传递新类型。下面是

Microsoft.Extensions.DependencyModel 库遵循的模式：

- UnifiedJsonReader.JsonTextReader.cs
- UnifiedJsonReader.Utf8JsonReader.cs

Utf8JsonWriter 与 JsonTextWriter 的比较

System.Text.Json.Utf8JsonWriter 是一种高性能方式，从常见 .NET 类型（例如，`String`、`Int32` 和 `DateTime`）编写 UTF-8 编码的 JSON 文本。该编写器是一种低级类型，可用于生成自定义序列化程序。

以下各节说明使用 `Utf8JsonWriter` 的推荐编程模式。

使用 UTF-8 文本进行编写

若要在使用 `Utf8JsonWriter` 时实现可能的最佳性能，请编写已编码为 UTF-8 文本（而不是 UTF-16 字符串）的 JSON 有效负载。使用 JsonEncodedText 可缓存已知字符串属性名称和值并预先编码为静态，并将这些内容传递给编写器，而不是使用 UTF-16 字符串文本。这比缓存并使用 UTF-8 字节数组更快。

如果需要进行自定义转义，此方法也适用。`System.Text.Json` 不允许在编写字符串时禁用转义。但是，可以将自己的自定义 JavaScriptEncoder 作为一个选项传入编写器，或创建自己的 `JsonEncodedText` 以使用你的 `JavascriptEncoder` 进行转义，然后编写 `JsonEncodedText` 而不是字符串。有关详细信息，请参阅自定义字符编码。

编写原始值

`Newtonsoft.Json` `WriteRawValue` 方法可编写原始 JSON（其中需要值）。System.Text.Json 没有直接等效项，但下面是确保仅编写有效 JSON 的解决方法：

```
using JsonDocument doc = JsonDocument.Parse(string);
doc.WriteTo(writer);
```

自定义字符转义

`JsonTextWriter` 的 StringEscapeHandling 设置提供用于转移所有非 ASCII 字符或 HTML 字符的选项。默认情况下，`Utf8JsonWriter` 会转义所有非 ASCII 和 HTML 字符。进行此转义是出于深度防御安全原因。若要指定不同的转义策略，请创建 JavaScriptEncoder 并设置 JsonWriterOptions.Encoder。有关详细信息，请参阅自定义字符编码。

自定义 JSON 格式

`JsonTextWriter` 包含以下设置（`Utf8JsonWriter` 对于它们没有等效项）：

- 缩进 - 指定要缩进的字符数。`Utf8JsonWriter` 始终执行 2 字符缩进。
- IndentChar - 指定要用于缩进的字符。`Utf8JsonWriter` 始终使用空格。
- QuoteChar - 指定要用于围绕字符串值的字符。`Utf8JsonWriter` 始终使用双引号。
- QuoteName - 指定是否要使用引号围绕属性名称。`Utf8JsonWriter` 始终使用引号围绕它们。

没有解决方法可让你自定义 `Utf8JsonWriter` 以这些方式生成的 JSON。

编写 null 值

若要使用 `Utf8JsonWriter` 编写 null 值，请调用：

- WriteNull，用于将具有 null 的键值对编写为值。
- WriteNullValue用于将 null 编写为 JSON 数组的元素。

对于字符串属性，如果字符串为 null，则 WriteString 和 WriteStringValue 等效于 `WriteNull` 和 `WriteNullValue`。

编写 Timespan、Uri 或 char 值

`JsonTextWriter` 提供 `WriteValue` 方法以用于 TimeSpan、Uri 和 char 值。`Utf8JsonWriter` 没有等效方法。而是将这些值格式化为字符串（例如，通过调用 `ToString()`）并调用 WriteStringValue。

多目标

如果需要继续为某些目标框架使用 `Newtonsoft.Json`，则可以使用多目标，并具有两种实现。但是，这并非易事，需要进行一些 `#ifdefs` 和源文件复制。共享尽可能多代码的一种方法是围绕 `Utf8JsonWriter` 和 `Newtonsoft JsonTextWriter` 创建包装器。此包装器会统一公共外围应用，同时隔离行为差异。这使你可以隔离主要对类型的构造进行的更改。Microsoft.Extensions.DependencyModel 库遵循：

- UnifiedJsonWriter.JsonTextWriter.cs
- UnifiedJsonWriter.Utf8JsonWriter.cs

其他资源

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何使用 System.Text.Json 自定义字符编码

2021/11/16 •

默认情况下，序列化程序会转义所有非 ASCII 字符。即，会将它们替换为 `\uxxxx`，其中 `xxxx` 为字符的 Unicode 代码。例如，如果以下 JSON 中的 `Summary` 属性设置为西里尔文 `жарко`，则 `WeatherForecast` 对象会进行序列化，如以下示例中所示：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "\u0436\u0430\u0440\u043A\u043E"
}
```

序列化语言字符集

若要序列化一种或多种语言的字符集而不进行转义，请在创建 System.Text.Encodings.Web.JavaScriptEncoder 的实例时指定 Unicode 范围，如以下示例中所示：

```
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

```
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(UnicodeRanges.BasicLatin, UnicodeRanges.Cyrillic),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

此代码不转义西里尔文或希腊语字符。如果 `Summary` 属性设置为西里尔文 `жарко`，则 `WeatherForecast` 对象会进行序列化，如以下示例中所示：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жарко"
}
```

若要序列化所有语言集而不进行转义，请使用 UnicodeRanges.All。

序列化特定字符

一种替代方法是指定要允许的单个字符，而不进行转义。下面的示例仅序列化 `жарко` 的前两个字符：

```
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

```
var encoderSettings = new TextEncoderSettings();
encoderSettings.AllowCharacters('\u0436', '\u0430');
encoderSettings.AllowRange(UnicodeRanges.BasicLatin);
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.Create(encoderSettings),
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

下面是前面代码生成的 JSON 的示例：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "жа\u0440\u043A\u043E"
}
```

序列化所有字符

若要最大程度地减少转义，可以使用 JavaScriptEncoder.UnsafeRelaxedJsonEscaping，如以下示例中所示：

```
using System;
using System.Text.Encodings.Web;
using System.Text.Json;
using System.Text.Unicode;
```

```
options = new JsonSerializerOptions
{
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping,
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, options);
```

Caution

与默认编码器相比，`UnsafeRelaxedJsonEscaping` 编码器在允许字符通过而不进行转义方面更加宽松：

- 它不转义 HTML 敏感字符，如 `<`、`>`、`&` 和 `'`。
- 它不提供任何针对 XSS 或信息泄露攻击（如客户端和服务器在字符集方面不一致所可能导致的攻击）的额外深度防御保护。

仅当知道客户端将生成的有效负载解释为 UTF-8 编码的 JSON 时，才使用不安全编码器。例如，如果服务器在发送响应标头 `Content-Type: application/json; charset=utf-8`，则可以使用它。永远不允许将原始 `UnsafeRelaxedJsonEscaping` 输出发出到 HTML 页面或 `<script>` 元素。

请参阅

- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON

- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 编写自定义序列化程序和反序列化程序
- 编写用于 JSON 序列化的自定义转换器
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

如何在 .NET 中编写用于 JSON 序列化（封送）的自定义转换器

2021/11/16 •

本文介绍如何为 System.Text.Json 命名空间中提供的 JSON 序列化类创建自定义转换器。有关 `System.Text.Json` 简介，请参阅如何在 .NET 中对 JSON 数据进行序列化和反序列化。

转换器是一种将对象或值与 JSON 相互转换的类。`System.Text.Json` 命名空间为映射到 JavaScript 基元的大多数基元类型提供内置转换器。可以编写自定义转换器来实现以下目标：

- 重写内置转换器的默认行为。例如，你可能希望通过 mm/dd/yyyy 格式来表示 `DateTime` 值。默认情况下，支持 ISO 8601-1:2019，包括 RFC 3339 配置文件。有关详细信息，请参阅 System.Text.Json 中的 DateTime 和 DateTimeOffset 支持。
- 支持自定义值类型。例如，`PhoneNumber` 结构。

还可以编写自定义转换器，以使用当前版本中未包含的功能自定义或扩展 `System.Text.Json`。本文后面部分介绍了以下方案：

- 将推断类型反序列化为对象属性。
- 支持多态反序列化。
- 支持堆栈的往返<T>。

- 将推断类型反序列化为对象属性。
- 支持包含非字符串键的字典。
- 支持多态反序列化。
- 支持堆栈的往返<T>。

在为自定义转换器编写的代码中，请注意，使用新的 JsonSerializerOptions 实例会带来重大性能损失。有关详细信息，请参阅重用 JsonSerializerOptions 实例。

自定义转换器模式

用于创建自定义转换器的模式有两种：基本模式和工厂模式。工厂模式适用于处理类型 `Enum` 或开放式泛型的转换器。基本模式适用于非泛型或封闭式泛型类型。例如，适用于以下类型的转换器需要工厂模式：

- Dictionary<TKey,TValue>
- Enum
- List<T>

可以通过基本模式处理的类型的一些示例包括：

- `Dictionary<int, string>`
- `WeekdaysEnum`
- `List<DateTimeOffset>`
- DateTime
- Int32

基本模式创建的类可以处理一种类型。工厂模式创建的类在运行时确定所需的特定类型，并动态创建适当的转换器。

示例基本转换器

下面的示例是一个转换器，可重写现有数据类型的默认序列化。该转换器将 mm/dd/yyyy 格式用于 `DateTimeOffset` 属性。

```
using System;
using System.Globalization;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DateTimeOffsetJsonConverter : JsonConverter<DateTimeOffset>
    {
        public override DateTimeOffset Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
                DateTimeOffset.ParseExact(reader.GetString(),
                    "MM/dd/yyyy", CultureInfo.InvariantCulture);

        public override void Write(
            Utf8JsonWriter writer,
            DateTimeOffset dateTimeValue,
            JsonSerializerOptions options) =>
                writer.WriteStringValue(dateTimeValue.ToString(
                    "MM/dd/yyyy", CultureInfo.InvariantCulture));
    }
}
```

示例工厂模式转换器

下面的代码演示一个处理 `Dictionary<Enum,TValue>` 的自定义转换器。该代码遵循工厂模式，因为第一个泛型类型参数是 `Enum`，第二个参数是开放参数。`CanConvert` 方法仅对具有两个泛型参数的 `Dictionary` 返回 `true`，其中第一个参数是 `Enum` 类型。内部转换器获取现有转换器，以处理在运行时为 `TValue` 提供的任何类型。

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DictionaryTKeyEnumTValueConverter : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
        {
            if (!typeToConvert.IsGenericType)
            {
                return false;
            }

            if (typeToConvert.GetGenericTypeDefinition() != typeof(Dictionary<,>))
            {
                return false;
            }

            return typeToConvert.GetGenericArguments()[0].IsEnum;
        }

        public override JsonConverter CreateConverter(
            Type type,
```

```
        JsonSerializerOptions options)
    {
        Type keyType = type.GetGenericArguments()[0];
        Type valueType = type.GetGenericArguments()[1];

        JsonConverter converter = (JsonConverter)Activator.CreateInstance(
            typeof(DictionaryEnumConverterInner<,>).MakeGenericType(
                new Type[] { keyType, valueType }),
            BindingFlags.Instance | BindingFlags.Public,
            binder: null,
            args: new object[] { options },
            culture: null);

        return converter;
    }

    private class DictionaryEnumConverterInner<TKey, TValue> :
        JsonConverter<Dictionary<TKey, TValue>> where TKey : struct, Enum
    {
        private readonly JsonConverter<TValue> _valueConverter;
        private readonly Type _keyType;
        private readonly Type _valueType;

        public DictionaryEnumConverterInner(JsonSerializerOptions options)
        {
            // For performance, use the existing converter if available.
            _valueConverter = (JsonConverter<TValue>)options
                .GetConverter(typeof(TValue));

            // Cache the key and value types.
            _keyType = typeof(TKey);
            _valueType = typeof(TValue);
        }

        public override Dictionary<TKey, TValue> Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            }

            var dictionary = new Dictionary<TKey, TValue>();

            while (reader.Read())
            {
                if (reader.TokenType == JsonTokenType.EndObject)
                {
                    return dictionary;
                }

                // Get the key.
                if (reader.TokenType != JsonTokenType.PropertyName)
                {
                    throw new JsonException();
                }

                string propertyName = reader.GetString();

                // For performance, parse with ignoreCase:false first.
                if (!Enum.TryParse(propertyName, ignoreCase: false, out TKey key) &&
                    !Enum.TryParse(propertyName, ignoreCase: true, out key))
                {
                    throw new JsonException(
                        $"Unable to convert \"{propertyName}\" to Enum \"{_keyType}\".");
                }
```

```
                    // Get the value.
                    TValue value;
                    if (_valueConverter != null)
                    {
                        reader.Read();
                        value = _valueConverter.Read(ref reader, _valueType, options);
                    }
                    else
                    {
                        value = JsonSerializer.Deserialize<TValue>(ref reader, options);
                    }

                    // Add to dictionary.
                    dictionary.Add(key, value);
                }

                throw new JsonException();
            }

            public override void Write(
                Utf8JsonWriter writer,
                Dictionary<TKey, TValue> dictionary,
                JsonSerializerOptions options)
            {
                writer.WriteStartObject();

                foreach ((TKey key, TValue value) in dictionary)
                {
                    var propertyName = key.ToString();
                    writer.WritePropertyName
                        (options.PropertyNamingPolicy?.ConvertName(propertyName) ?? propertyName);

                    if (_valueConverter != null)
                    {
                        _valueConverter.Write(writer, value, options);
                    }
                    else
                    {
                        JsonSerializer.Serialize(writer, value, options);
                    }
                }

                writer.WriteEndObject();
            }
        }
    }
}
```

前面的代码与本文后面的支持包含非字符串键的字典中演示的代码相同。

遵循基本模式的步骤

以下步骤说明如何遵循基本模式来创建转换器：

- 创建一个派生自 JsonConverter<T> 的类，其中 `T` 是要进行序列化和反序列化的类型。
- 重写 `Read` 方法，以反序列化传入 JSON 并将其转换为类型 `T`。使用传递给方法的 Utf8JsonReader 读取 JSON。
- 重写 `Write` 方法以序列化 `T` 类型的传入对象。使用传递给方法的 Utf8JsonWriter 写入 JSON。
- 仅当需要时才重写 `CanConvert` 方法。当要转换的类型属于类型 `T` 时，默认实现会返回 `true`。因此，仅支持类型 `T` 的转换器不需要重写此方法。有关的确需要重写此方法的转换器的示例，请参阅本文后面的多态反序列化部分。

可以参阅内置转换器源代码作为用于编写自定义转换器的参考实现。

遵循工厂模式的步骤

以下步骤说明如何遵循工厂模式来创建转换器：

- 创建一个从 JsonConverterFactory 派生的类。
- 重写 `CanConvert` 方法，以在要转换的类型是转换器可处理的类型时返回 true。例如，如果转换器适用于 `List<T>`，则它可能仅处理 `List<int>`、`List<string>` 和 `List<DateTime>`。
- 重写 `CreateConverter` 方法，以返回将处理在运行时提供的要转换的类型的转换器类实例。
- 创建 `CreateConverter` 方法实例化的转换器类。

开放式泛型需要工厂模式，因为用于将对象与字符串相互转换的代码对于所有类型并不相同。适用于开放式泛型类型（例如 `List<T>`）的转换器必须在幕后为封闭式泛型类型（例如 `List<DateTime>`）创建转换器。必须编写代码来处理转换器可处理的每种封闭式泛型类型。

`Enum` 类型类似于开放式泛型类型：适用于 `Enum` 的转换器必须在幕后为特定 `Enum`（例如 `WeekdaysEnum`）创建转换器。

错误处理

序列化程序为 JsonException 和 NotSupportedException 异常类型提供特殊处理。

JsonException

如果你引发不带消息的 `JsonException`，则序列化程序会创建一条消息，其中包括导致错误的 JSON 部分的路径。例如，语句 `throw new JsonException()` 会生成如以下示例的错误消息：

```
Unhandled exception. System.Text.Json.JsonException:
The JSON value could not be converted to System.Object.
Path: $.Date | LineNumber: 1 | BytePositionInLine: 37.
```

如果你确实提供了消息（例如 `throw new JsonException("Error occurred")`），则序列化程序仍会设置 Path、LineNumber 和 BytePositionInLine 属性。

NotSupportedException

如果你引发 `NotSupportedException`，则始终会在消息中获取路径信息。如果你提供了消息，则路径信息将追加到该消息中。例如，语句 `throw new NotSupportedException("Error occurred.")` 会生成如以下示例的错误消息：

```
Error occurred. The unsupported member type is located on type
'System.Collections.Generic.Dictionary`2[Samples.SummaryWords,System.Int32]'.
Path: $.TemperatureRanges | LineNumber: 4 | BytePositionInLine: 24
```

何时引发哪种异常类型

当 JSON 有效负载包含对于正在进行反序列化的类型无效的令牌时，引发 `JsonException`。

当你要禁止某些类型时，引发 `NotSupportedException`。对于不支持的类型，序列化程序会自动引发此异常。例如，出于安全原因，不支持 `System.Type`，因此尝试对其进行反序列化会导致 `NotSupportedException`。

可根据需要引发其他异常，但它们不会自动包括 JSON 路径信息。

注册自定义转换器

注册自定义转换器，使 `Serialize` 和 `Deserialize` 方法可使用它。选择以下方法之一：

- 向 JsonSerializerOptions.Converters 集合添加转换器类的实例。
- 将 [JsonConverter] 特性应用于需要自定义转换器的属性。

- 将 [JsonConverter] 特性应用于表示自定义值类型的类或结构。

注册示例 - 转换器集合

下面是一个示例，该示例将 DateTimeOffsetConverter 设为类型 DateTimeOffset 的属性的默认值：

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true,
    Converters =
    {
        new DateTimeOffsetJsonConverter()
    }
};

jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

假设序列化以下类型的实例：

```
public class WeatherForecast
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

下面是演示使用自定义转换器的 JSON 输出的示例：

```
{
  "Date": "08/01/2019",
  "TemperatureCelsius": 25,
  "Summary": "Hot"
}
```

下面的代码使用的方法与使用自定义 `DateTimeOffset` 转换器进行反序列化相同：

```
var deserializeOptions = new JsonSerializerOptions();
deserializeOptions.Converters.Add(new DateTimeOffsetJsonConverter());
weatherForecast = JsonSerializer.Deserialize<WeatherForecast>(jsonString, deserializeOptions);
```

注册示例 - 属性上的 [JsonConverter]

下面的代码为 `Date` 属性选择自定义转换器：

```
public class WeatherForecastWithConverterAttribute
{
    [JsonConverter(typeof(DateTimeOffsetJsonConverter))]
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

用于序列化 `WeatherForecastWithConverterAttribute` 的代码不需要使用 `JsonSerializeOptions.Converters`：

```
var serializeOptions = new JsonSerializerOptions
{
    WriteIndented = true
};
jsonString = JsonSerializer.Serialize(weatherForecast, serializeOptions);
```

用于反序列化的代码也不需要使用 `Converters`：

```
weatherForecast = JsonSerializer.Deserialize<WeatherForecastWithConverterAttribute>(jsonString);
```

注册示例 - 类型上的 [JsonConverter]

下面的代码创建一个结构并向它应用 `[JsonConverter]` 属性：

```
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    [JsonConverter(typeof(TemperatureConverter))]
    public struct Temperature
    {
        public Temperature(int degrees, bool celsius)
        {
            Degrees = degrees;
            IsCelsius = celsius;
        }

        public int Degrees { get; }
        public bool IsCelsius { get; }
        public bool IsFahrenheit => !IsCelsius;

        public override string ToString() =>
            $"{Degrees}{(IsCelsius ? "C" : "F")}";

        public static Temperature Parse(string input)
        {
            int degrees = int.Parse(input.Substring(0, input.Length - 1));
            bool celsius = input.Substring(input.Length - 1) == "C";

            return new Temperature(degrees, celsius);
        }
    }
}
```

下面是适用于上述结构的自定义转换器：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class TemperatureConverter : JsonConverter<Temperature>
    {
        public override Temperature Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
                Temperature.Parse(reader.GetString());

        public override void Write(
            Utf8JsonWriter writer,
            Temperature temperature,
            JsonSerializerOptions options) =>
                writer.WriteStringValue(temperature.ToString());
    }
}
```

结构上的 `[JsonConvert]` 属性将自定义转换器注册为类型 `Temperature` 的属性的默认值。进行序列化或反序列化时，转换器会自动用于以下类型的 `TemperatureCelsius` 属性：

```
public class WeatherForecastWithTemperatureStruct
{
    public DateTimeOffset Date { get; set; }
    public Temperature TemperatureCelsius { get; set; }
    public string Summary { get; set; }
}
```

转换器注册优先级

在序列化或反序列化过程中，按以下顺序（从最高优先级到最低优先级来列出）为每个 JSON 元素选择转换器：

- 应用于属性的 `[JsonConverter]`。
- 向 `Converters` 集合添加的转换器。
- 应用于自定义值类型或 POCO 的 `[JsonConverter]`。

如果在 `Converters` 集合中注册了适用于某种类型的多个自定义转换器，则使用第一个为 `CanConvert` 返回 true 的转换器。

仅当未注册适用自定义转换器时，才会选择内置转换器。

常见方案的转换器示例

以下各部分提供的转换器示例用于解决内置功能不处理的一些常见方案。

- 将推断类型反序列化为对象属性。
- 支持多态反序列化。
- 支持堆栈的往返<T>。

- 将推断类型反序列化为对象属性。
- 支持包含非字符串键的字典。
- 支持多态反序列化。
- 支持堆栈的往返<T>。

将推断类型反序列化为对象属性

反序列化为类型 `object` 的属性时，将创建一个 `JsonElement` 对象。这是因为反序列化程序不知道要创建的 CLR 类型，也不会尝试进行猜测。例如，如果 JSON 属性具有“true”，则反序列化程序不会推断值为 `Boolean`，如果元素具有“01/01/2019”，则反序列化程序不会推断它是 `DateTime`。

类型推理可能不准确。如果反序列化程序将没有小数点的 JSON 数字分析为 `long`，则当值最初序列化为 `ulong` 或 `BigInteger` 时，这可能会导致超出范围问题。如果数字最初序列化为 `decimal`，则将具有小数点的数字分析为 `double` 可能会损失精度。

对于需要类型推理的方案，以下代码演示适用于 `object` 属性的自定义转换器。代码：

- 将 `true` 和 `false` 转换为 `Boolean`
- 将不带小数的数字转换为 `long`
- 将带有小数的数字转换为 `double`
- 将日期转换为 `DateTime`
- 将字符串转换为 `string`
- 将其他所有内容转换为 `JsonElement`

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class ObjectToInferredTypesConverter
        : JsonConverter<object>
    {
        public override object Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) => reader.TokenType switch
            {
                JsonTokenType.True => true,
                JsonTokenType.False => false,
                JsonTokenType.Number when reader.TryGetInt64(out long l) => l,
                JsonTokenType.Number => reader.GetDouble(),
                JsonTokenType.String when reader.TryGetDateTime(out DateTime datetime) => datetime,
                JsonTokenType.String => reader.GetString(),
                _ => JsonDocument.ParseValue(ref reader).RootElement.Clone()
            };

        public override void Write(
            Utf8JsonWriter writer,
            object objectToWrite,
            JsonSerializerOptions options) =>
            throw new InvalidOperationException("Should not get here.");
    }
}
```

下面的代码注册转换器：

```
var deserializeOptions = new JsonSerializerOptions
{
    Converters =
    {
        new ObjectToInferredTypesConverter()
    }
};
```

下面是一种具有 `object` 属性的示例类型：

```
public class WeatherForecastWithObjectProperties
{
    public object Date { get; set; }
    public object TemperatureCelsius { get; set; }
    public object Summary { get; set; }
}
```

以下要反序列化的 JSON 示例包含将作为 `DateTime`、`long` 和 `string` 进行反序列化的值：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
}
```

如果没有自定义转换器，则反序列化会将 `JsonElement` 放入每个属性中。

`System.Text.Json.Serialization` 命名空间中的单元测试文件夹包含处理到 `object` 属性的反序列化的自定义转换器的更多示例。

支持包含非字符串键的字典

对字典集合的内置支持适用于 `Dictionary<string, TValue>`。即，键必须是字符串。若要支持将整数或某种其他类型用作键的字典，需要自定义转换器。

下面的代码演示一个处理 `Dictionary<Enum,TValue>` 的自定义转换器：

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class DictionaryTKeyEnumTValueConverter : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
        {
            if (!typeToConvert.IsGenericType)
            {
                return false;
            }

            if (typeToConvert.GetGenericTypeDefinition() != typeof(Dictionary<,>))
            {
                return false;
            }

            return typeToConvert.GetGenericArguments()[0].IsEnum;
        }

        public override JsonConverter CreateConverter(
            Type type,
            JsonSerializerOptions options)
        {
            Type keyType = type.GetGenericArguments()[0];
            Type valueType = type.GetGenericArguments()[1];

            JsonConverter converter = (JsonConverter)Activator.CreateInstance(
                typeof(DictionaryEnumConverterInner<,>).MakeGenericType(
                    new Type[] { keyType, valueType }),
```

```
                new Type[] { keyType, valueType }),
            BindingFlags.Instance | BindingFlags.Public,
            binder: null,
            args: new object[] { options },
            culture: null);

        return converter;
    }

    private class DictionaryEnumConverterInner<TKey, TValue> :
        JsonConverter<Dictionary<TKey, TValue>> where TKey : struct, Enum
    {
        private readonly JsonConverter<TValue> _valueConverter;
        private readonly Type _keyType;
        private readonly Type _valueType;

        public DictionaryEnumConverterInner(JsonSerializerOptions options)
        {
            // For performance, use the existing converter if available.
            _valueConverter = (JsonConverter<TValue>)options
                .GetConverter(typeof(TValue));

            // Cache the key and value types.
            _keyType = typeof(TKey);
            _valueType = typeof(TValue);
        }

        public override Dictionary<TKey, TValue> Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
                throw new JsonException();
            }

            var dictionary = new Dictionary<TKey, TValue>();

            while (reader.Read())
            {
                if (reader.TokenType == JsonTokenType.EndObject)
                {
                    return dictionary;
                }

                // Get the key.
                if (reader.TokenType != JsonTokenType.PropertyName)
                {
                    throw new JsonException();
                }

                string propertyName = reader.GetString();

                // For performance, parse with ignoreCase:false first.
                if (!Enum.TryParse(propertyName, ignoreCase: false, out TKey key) &&
                    !Enum.TryParse(propertyName, ignoreCase: true, out key))
                {
                    throw new JsonException(
                        $"Unable to convert \"{propertyName}\" to Enum \"{_keyType}\".");
                }

                // Get the value.
                TValue value;
                if (_valueConverter != null)
                {
                    reader.Read();
                    value = _valueConverter.Read(ref reader, _valueType, options);
                }
```

```
                    else
                    {
                        value = JsonSerializer.Deserialize<TValue>(ref reader, options);
                    }

                    // Add to dictionary.
                    dictionary.Add(key, value);
                }

                throw new JsonException();
            }

            public override void Write(
                Utf8JsonWriter writer,
                Dictionary<TKey, TValue> dictionary,
                JsonSerializerOptions options)
            {
                writer.WriteStartObject();

                foreach ((TKey key, TValue value) in dictionary)
                {
                    var propertyName = key.ToString();
                    writer.WritePropertyName
                        (options.PropertyNamingPolicy?.ConvertName(propertyName) ?? propertyName);

                    if (_valueConverter != null)
                    {
                        _valueConverter.Write(writer, value, options);
                    }
                    else
                    {
                        JsonSerializer.Serialize(writer, value, options);
                    }
                }

                writer.WriteEndObject();
            }
        }
    }
}
```

下面的代码注册转换器：

```
var serializeOptions = new JsonSerializerOptions();
serializeOptions.Converters.Add(new DictionaryTKeyEnumTValueConverter());
```

该转换器可以序列化和反序列化使用以下 `Enum` 的以下类的 `TemperatureRanges` 属性：

```
public class WeatherForecastWithEnumDictionary
{
    public DateTimeOffset Date { get; set; }
    public int TemperatureCelsius { get; set; }
    public string Summary { get; set; }
    public Dictionary<SummaryWordsEnum, int> TemperatureRanges { get; set; }
}

public enum SummaryWordsEnum
{
    Cold, Hot
}
```

来自序列化的 JSON 输出类似于以下示例：

```
{
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "TemperatureRanges": {
    "Cold": 20,
    "Hot": 40
  }
}
```

`System.Text.Json.Serialization` 命名空间中的单元测试文件夹包含处理非字符串键字典的自定义转换器的更多示例。

支持多态反序列化

内置功能提供有限范围的多态序列化，但完全不支持反序列化。反序列化需要自定义转换器。

例如，假设有一个 `Person` 抽象基类，其中包含 `Employee` 和 `Customer` 派生类。多态反序列化意味着可以在设计时将 `Person` 指定为反序列化目标，JSON 中的 `Customer` 和 `Employee` 对象会在运行时正确地进行反序列化。在反序列化过程中，必须查找标识 JSON 中所需类型的线索。可用的线索类型因各个方案而异。例如，可以使用鉴别器属性，或者可能必须依赖于特定属性是否存在。`System.Text.Json` 的当前版本不提供属性来指定如何处理多态反序列化方案，因此需要自定义转换器。

下面的代码演示一个基类、两个派生类和适用于它们的一个自定义转换器。该转换器使用鉴别器属性执行多态反序列化。类型鉴别器不在类定义中，而是在序列化过程中创建，在反序列化过程中进行读取。

```
public class Person
{
    public string Name { get; set; }
}

public class Customer : Person
{
    public decimal CreditLimit { get; set; }
}

public class Employee : Person
{
    public string OfficeNumber { get; set; }
}
```

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace SystemTextJsonSamples
{
    public class PersonConverterWithTypeDiscriminator : JsonConverter<Person>
    {
        enum TypeDiscriminator
        {
            Customer = 1,
            Employee = 2
        }

        public override bool CanConvert(Type typeToConvert) =>
            typeof(Person).IsAssignableFrom(typeToConvert);

        public override Person Read(
            ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartObject)
            {
```

```
        {
            throw new JsonException();
        }

        reader.Read();
        if (reader.TokenType != JsonTokenType.PropertyName)
        {
            throw new JsonException();
        }

        string propertyName = reader.GetString();
        if (propertyName != "TypeDiscriminator")
        {
            throw new JsonException();
        }

        reader.Read();
        if (reader.TokenType != JsonTokenType.Number)
        {
            throw new JsonException();
        }

        TypeDiscriminator typeDiscriminator = (TypeDiscriminator)reader.GetInt32();
        Person person = typeDiscriminator switch
        {
            TypeDiscriminator.Customer => new Customer(),
            TypeDiscriminator.Employee => new Employee(),
            _ => throw new JsonException()
        };

        while (reader.Read())
        {
            if (reader.TokenType == JsonTokenType.EndObject)
            {
                return person;
            }

            if (reader.TokenType == JsonTokenType.PropertyName)
            {
                propertyName = reader.GetString();
                reader.Read();
                switch (propertyName)
                {
                    case "CreditLimit":
                        decimal creditLimit = reader.GetDecimal();
                        ((Customer)person).CreditLimit = creditLimit;
                        break;
                    case "OfficeNumber":
                        string officeNumber = reader.GetString();
                        ((Employee)person).OfficeNumber = officeNumber;
                        break;
                    case "Name":
                        string name = reader.GetString();
                        person.Name = name;
                        break;
                }
            }
        }

        throw new JsonException();
    }

    public override void Write(
        Utf8JsonWriter writer, Person person, JsonSerializerOptions options)
    {
        writer.WriteStartObject();

        if (person is Customer customer)
        {
```

```
                writer.WriteNumber("TypeDiscriminator", (int)TypeDiscriminator.Customer);
                writer.WriteNumber("CreditLimit", customer.CreditLimit);
            }
            else if (person is Employee employee)
            {
                writer.WriteNumber("TypeDiscriminator", (int)TypeDiscriminator.Employee);
                writer.WriteString("OfficeNumber", employee.OfficeNumber);
            }

            writer.WriteString("Name", person.Name);

            writer.WriteEndObject();
        }
    }
}
```

下面的代码注册转换器：

```
var serializeOptions = new JsonSerializerOptions();
serializeOptions.Converters.Add(new PersonConverterWithTypeDiscriminator());
```

该转换器可以反序列化通过用于序列化的相同转换器而创建的 JSON，例如：

```
[
  {
    "TypeDiscriminator": 1,
    "CreditLimit": 10000,
    "Name": "John"
  },
  {
    "TypeDiscriminator": 2,
    "OfficeNumber": "555-1234",
    "Name": "Nancy"
  }
]
```

前面示例中的转换器代码会手动读取和写入每个属性。一种替代方法是调用 `Deserialize` 或 `Serialize` 以执行某些工作。有关示例，请参阅此 StackOverflow 文章。

支持堆栈的往返<T>

如果将 JSON 字符串反序列化为 Stack<T> 对象，然后再序列化该对象，则堆栈的内容将按相反的顺序排列。此行为适用于以下类型和接口以及从它们派生的用户定义类型：

- Stack
- Stack<T>
- ConcurrentStack<T>
- ImmutableStack<T>
- IImmutableStack<T>

若要支持在堆栈中保留原始顺序的序列化和反序列化，则需要自定义转换器。

下面的代码演示了一个自定义转换器，用于实现与 `Stack<T>` 对象之间的来回转换：

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Reflection;
using System.Text.Json;
using System.Text.Json.Serialization;
```

```
namespace SystemTextJsonSamples
{
    public class JsonConverterFactoryForStackOfT : JsonConverterFactory
    {
        public override bool CanConvert(Type typeToConvert)
            => typeToConvert.IsGenericType
            && typeToConvert.GetGenericTypeDefinition() == typeof(Stack<>);

        public override JsonConverter CreateConverter(
            Type typeToConvert, JsonSerializerOptions options)
        {
            Debug.Assert(typeToConvert.IsGenericType &&
                typeToConvert.GetGenericTypeDefinition() == typeof(Stack<>));

            Type elementType = typeToConvert.GetGenericArguments()[0];

            JsonConverter converter = (JsonConverter)Activator.CreateInstance(
                typeof(JsonConverterForStackOfT<>)
                    .MakeGenericType(new Type[] { elementType }),
                BindingFlags.Instance | BindingFlags.Public,
                binder: null,
                args: null,
                culture: null)!;

            return converter;
        }
    }

    public class JsonConverterForStackOfT<T> : JsonConverter<Stack<T>>
    {
        public override Stack<T> Read(
            ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
        {
            if (reader.TokenType != JsonTokenType.StartArray)
            {
                throw new JsonException();
            }
            reader.Read();

            var elements = new Stack<T>();

            while (reader.TokenType != JsonTokenType.EndArray)
            {
                elements.Push(JsonSerializer.Deserialize<T>(ref reader, options));

                reader.Read();
            }

            return elements;
        }

        public override void Write(
            Utf8JsonWriter writer, Stack<T> value, JsonSerializerOptions options)
        {
            writer.WriteStartArray();

            var reversed = new Stack<T>(value);

            foreach (T item in reversed)
            {
                JsonSerializer.Serialize(writer, item, options);
            }

            writer.WriteEndArray();
        }
    }
}
```

下面的代码注册转换器：

```
var options = new JsonSerializerOptions
{
    Converters = { new JsonConverterFactoryForStackOfT() },
};
```

处理 NULL 值

默认情况下，序列化程序处理 null 值，如下所示：

- 对于引用类型和 Nullable<T> 类型：
  - 它在序列化时不会将 `null` 传递到自定义转换器。
  - 它在反序列化时不会将 `JsonTokenType.Null` 传递到自定义转换器。
  - 它在反序列化时返回 `null` 实例。
  - 它在序列化时直接使用编写器写入 `null`。
- 对于不可为 null 的值类型：
  - 它在反序列化时将 `JsonTokenType.Null` 传递到自定义转换器。（如果没有可用的自定义转换器，则由该类型的内部转换器引发 `JsonException` 异常。）

此 null 处理行为主要用于，通过跳过对转换器的额外调用来优化性能。此外，它可避免在每个 `Read` 和 `Write` 方法重写开始时强制可以为 null 的类型的转换器检查 `null`。

若要启用自定义转换器来处理引用或值类型的 `null`，请重写 JsonConverter<T>.HandleNull 以返回 `true`，如以下示例中所示：

```
using System;
using System.Text.Json;
using System.Text.Json.Serialization;

namespace CustomConverterHandleNull
{
    public class Point
    {
        public int X { get; set; }
        public int Y { get; set; }

        [JsonConverter(typeof(DescriptionConverter))]
        public string Description { get; set; }
    }

    public class DescriptionConverter : JsonConverter<string>
    {
        public override bool HandleNull => true;

        public override string Read(
            ref Utf8JsonReader reader,
            Type typeToConvert,
            JsonSerializerOptions options) =>
            reader.GetString() ?? "No description provided.";

        public override void Write(
            Utf8JsonWriter writer,
            string value,
            JsonSerializerOptions options) =>
            writer.WriteStringValue(value);
    }

    public class Program
    {
        public static void Main()
        {
            string json = @"{""x"":1,""y"":2,""Description"":null}";

            Point point = JsonSerializer.Deserialize<Point>(json);
            Console.WriteLine($"Description: {point.Description}");
        }
    }
}

// Produces output like the following example:
//
//Description: No description provided.
```

其他自定义转换器示例

从 Newtonsoft.Json 迁移到 System.Text.Json 一文包含自定义转换器的其他示例。

`System.Text.Json.Serialization` 源代码中的单元测试文件夹包含其他自定义转换器示例，例如：

- 在反序列化时将 null 转换为 0 的 Int32 转换器
- 在反序列化时允许同时使用字符串和数字值的 Int32 转换器
- 枚举转换器
- 接受外部数据的 List<T> 转换器
- 处理以逗号分隔的数字列表的 Long[] 转换器

如果需要创建修改现有内置转换器行为的转换器，则可以获取现有转换器的源代码作为自定义的起点。

其他资源

- 内置转换器的源代码
- System.Text.Json 概述
- 如何对 JSON 进行序列化和反序列化
- 对 JsonSerializerOptions 实例进行实例化
- 启用不区分大小写的匹配
- 自定义属性名称和值
- 忽略属性
- 允许无效的 JSON
- 处理溢出 JSON
- 保留引用
- 不可变类型和非公共访问器
- 多态序列化
- 从 Newtonsoft.Json 迁移到 System.Text.Json
- 自定义字符编码
- 编写自定义序列化程序和反序列化程序
- DateTime 和 DateTimeOffset 支持
- System.Text.Json API 参考
- System.Text.Json.Serialization API 参考

二进制序列化

2021/11/16 •

可以将序列化定义为一个将对象状态存储到存储介质的过程。在这个过程中，对象的公共字段和私有字段以及类(包括含有该类的程序集)的名称，将转换成字节流，而字节流接着将写入数据流。随后对该对象进行反序列化时，将创建原始对象的准确克隆。

在面向对象的环境中实现序列化机制时，必须多在易用性与灵活性之间做出权衡。很大程度上，这个过程可以自动完成，但前提是您对该过程拥有足够的控制权。例如，如果简单的二进制序列化不足，或者可能有特定原因决定需要对类中的哪些字段进行序列化，可能就会出现这种情况。以下章节验证了随 .NET Framework 一起提供的可靠序列化机制，并强调了根据需要自定义该过程所能使用的一些重要功能。

> **NOTE**
> 如果使用不同的 .NET 版本序列化和反序列化以 UTF-8 或 UTF-7 编码的对象，则不保留该对象的状态。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

二进制序列化允许修改对象内部的私有成员，从而更改其状态。因此，建议采用在公共 API 表面运行的其他序列化框架，例如 System.Text.Json。

.NET Core

.NET Core 支持类型子集的二进制序列化。可在下面的可序列化类型部分查看受支持的类型列表。列出的类型是完全可以在 .NET Framework 4.5.1 及更高版本之间和 .NET Core 2.0 及更高版本之间进行序列化的。其他 .NET 实现(如 Mono)并未正式得到支持，但应该也能使用。

可序列化类型

"	"
Microsoft.CSharp.RuntimeBinder.RuntimeBinderException	从 .NET Core 2.0.4 开始。
Microsoft.CSharp.RuntimeBinder.RuntimeBinderInternalCompilerException	从 .NET Core 2.0.4 开始。
System.AccessViolationException	从 .NET Core 2.0.4 开始。
System.AggregateException	从 .NET Core 2.0.4 开始。
System.AppDomainUnloadedException	从 .NET Core 2.0.4 开始。
System.ApplicationException	从 .NET Core 2.0.4 开始。
System.ArgumentException	从 .NET Core 2.0.4 开始。
System.ArgumentNullException	从 .NET Core 2.0.4 开始。

从	从
System.ArgumentOutOfRangeException	从 .NET Core 2.0.4 开始。
System.ArithmeticException	从 .NET Core 2.0.4 开始。
System.Array	
System.ArraySegment<T>	
System.ArrayTypeMismatchException	从 .NET Core 2.0.4 开始。
System.Attribute	
System.BadImageFormatException	从 .NET Core 2.0.4 开始。
System.Boolean	
System.Byte	
System.CannotUnloadAppDomainException	从 .NET Core 2.0.4 开始。
System.Char	
System.Collections.ArrayList	
System.Collections.BitArray	
System.Collections.Comparer	
System.Collections.DictionaryEntry	
System.Collections.Generic.Comparer<T>	
System.Collections.Generic.Dictionary<TKey,TValue>	
System.Collections.Generic.EqualityComparer<T>	
System.Collections.Generic.HashSet<T>	
System.Collections.Generic.KeyNotFoundException	从 .NET Core 2.0.4 开始。
System.Collections.Generic.KeyValuePair<TKey,TValue>	
System.Collections.Generic.LinkedList<T>	
System.Collections.Generic.List<T>	
System.Collections.Generic.Queue<T>	
System.Collections.Generic.SortedDictionary<TKey,TValue>	

𝙸𝙸	𝙸𝙸
System.Collections.Generic.SortedList<TKey,TValue>	
System.Collections.Generic.SortedSet<T>	
System.Collections.Generic.Stack<T>	
System.Collections.Hashtable	
System.Collections.ObjectModel.Collection<T>	
System.Collections.ObjectModel.KeyedCollection<TKey,TItem>	
System.Collections.ObjectModel.ObservableCollection<T>	
System.Collections.ObjectModel.ReadOnlyCollection<T>	
System.Collections.ObjectModel.ReadOnlyDictionary<TKey,TValue>	
System.Collections.ObjectModel.ReadOnlyObservableCollection<T>	
System.Collections.Queue	
System.Collections.SortedList	
System.Collections.Specialized.HybridDictionary	
System.Collections.Specialized.ListDictionary	
System.Collections.Specialized.OrderedDictionary	
System.Collections.Specialized.StringCollection	
System.Collections.Specialized.StringDictionary	
System.Collections.Stack	
`System.Collections.Generic.NonRandomizedStringEqualityComparer`	从 .NET Core 2.0.4 开始。
System.ComponentModel.BindingList<T>	
System.ComponentModel.DataAnnotations.ValidationException	从 .NET Core 2.0.4 开始。
System.ComponentModel.Design.CheckoutException	从 .NET Core 2.0.4 开始。
System.ComponentModel.InvalidAsynchronousStateException	从 .NET Core 2.0.4 开始。

从	从
System.ComponentModel.InvalidEnumArgumentException	从 .NET Core 2.0.4 开始。
System.ComponentModel.LicenseException	从 .NET Core 2.0.4 开始。 不支持从 .NET Framework 到 .NET Core 的序列化。
System.ComponentModel.WarningException	从 .NET Core 2.0.4 开始。
System.ComponentModel.Win32Exception	从 .NET Core 2.0.4 开始。
System.Configuration.ConfigurationErrorsException	从 .NET Core 2.0.4 开始。
System.Configuration.ConfigurationException	从 .NET Core 2.0.4 开始。
System.Configuration.Provider.ProviderException	从 .NET Core 2.0.4 开始。
System.Configuration.SettingsPropertyIsReadOnlyException	从 .NET Core 2.0.4 开始。
System.Configuration.SettingsPropertyNotFoundException	从 .NET Core 2.0.4 开始。
System.Configuration.SettingsPropertyWrongTypeException	从 .NET Core 2.0.4 开始。
System.ContextMarshalException	从 .NET Core 2.0.4 开始。
System.DBNull	从 .NET Core 2.0.2 和更高版本开始。
System.Data.Common.DbException	从 .NET Core 2.0.4 开始。
System.Data.ConstraintException	从 .NET Core 2.0.4 开始。
System.Data.DBConcurrencyException	从 .NET Core 2.0.4 开始。
System.Data.DataException	从 .NET Core 2.0.4 开始。
System.Data.DataSet	
System.Data.DataTable	如果将 `RemotingFormat` 设置为 `SerializationFormat.Binary`，则只能与 .NET Core 2.1 和更高版本进行交换。
System.Data.DeletedRowInaccessibleException	从 .NET Core 2.0.4 开始。
System.Data.DuplicateNameException	从 .NET Core 2.0.4 开始。
System.Data.EvaluateException	从 .NET Core 2.0.4 开始。
System.Data.InRowChangingEventException	从 .NET Core 2.0.4 开始。
System.Data.InvalidConstraintException	从 .NET Core 2.0.4 开始。
System.Data.InvalidExpressionException	从 .NET Core 2.0.4 开始。

System.Data.MissingPrimaryKeyException	从 .NET Core 2.0.4 开始。
System.Data.NoNullAllowedException	从 .NET Core 2.0.4 开始。
System.Data.Odbc.OdbcException	从 .NET Core 2.0.4 开始。
System.Data.OperationAbortedException	从 .NET Core 2.0.4 开始。
System.Data.PropertyCollection	
System.Data.ReadOnlyException	从 .NET Core 2.0.4 开始。
System.Data.RowNotInTableException	从 .NET Core 2.0.4 开始。
System.Data.SqlClient.SqlException	从 .NET Core 2.0.4 开始。 不支持从 .NET Framework 到 .NET Core 的序列化
System.Data.SqlTypes.SqlAlreadyFilledException	从 .NET Core 2.0.4 开始。
System.Data.SqlTypes.SqlBoolean	
System.Data.SqlTypes.SqlByte	
System.Data.SqlTypes.SqlDateTime	
System.Data.SqlTypes.SqlDouble	
System.Data.SqlTypes.SqlGuid	
System.Data.SqlTypes.SqlInt16	
System.Data.SqlTypes.SqlInt32	
System.Data.SqlTypes.SqlInt64	
System.Data.SqlTypes.SqlNotFilledException	从 .NET Core 2.0.4 开始。
System.Data.SqlTypes.SqlNullValueException	从 .NET Core 2.0.4 开始。
System.Data.SqlTypes.SqlString	
System.Data.SqlTypes.SqlTruncateException	从 .NET Core 2.0.4 开始。
System.Data.SqlTypes.SqlTypeException	从 .NET Core 2.0.4 开始。
System.Data.StrongTypingException	从 .NET Core 2.0.4 开始。
System.Data.SyntaxErrorException	从 .NET Core 2.0.4 开始。
System.Data.VersionNotFoundException	从 .NET Core 2.0.4 开始。

从	从
System.DataMisalignedException	从 .NET Core 2.0.4 开始。
System.DateTime	
System.DateTimeOffset	
System.Decimal	
`System.Diagnostics.Contracts.ContractException`	从 .NET Core 2.0.4 开始。
System.Diagnostics.Tracing.EventSourceException	从 .NET Core 2.0.4 开始。
System.IO.DirectoryNotFoundException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.MultipleMatchesException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.NoMatchingPrincipalException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.PasswordException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.PrincipalException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.PrincipalExistsException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.PrincipalOperationException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.AccountManagement.PrincipalServerDownException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.ActiveDirectoryObjectExistsException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.ActiveDirectoryObjectNotFoundException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.ActiveDirectoryOperationException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.ActiveDirectoryServerDownException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.ForestTrustCollisionException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.ActiveDirectory.SyncFromAllServersOperationException	从 .NET Core 2.0.4 开始。

从	从
System.DirectoryServices.DirectoryServicesCOMException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.Protocols.BerConversionException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.Protocols.DirectoryException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.Protocols.DirectoryOperationException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.Protocols.LdapException	从 .NET Core 2.0.4 开始。
System.DirectoryServices.Protocols.TlsOperationException	从 .NET Core 2.0.4 开始。
System.DivideByZeroException	从 .NET Core 2.0.4 开始。
System.DllNotFoundException	从 .NET Core 2.0.4 开始。
System.Double	
System.Drawing.Color	
System.Drawing.Point	
System.Drawing.PointF	
System.Drawing.Rectangle	
System.Drawing.RectangleF	
System.Drawing.Size	
System.Drawing.SizeF	
System.DuplicateWaitObjectException	从 .NET Core 2.0.4 开始。
System.EntryPointNotFoundException	从 .NET Core 2.0.4 开始。
System.Enum	
System.EventArgs	从 .NET Core 2.0.6 开始。
System.Exception	
System.ExecutionEngineException	从 .NET Core 2.0.4 开始。
System.FieldAccessException	从 .NET Core 2.0.4 开始。
System.FormatException	从 .NET Core 2.0.4 开始。
System.Globalization.CompareInfo	

从	从
System.Globalization.CultureNotFoundException	从 .NET Core 2.0.4 开始。
System.Globalization.SortVersion	
System.Guid	
`System.IO.Compression.ZLibException`	从 .NET Core 2.0.4 开始。
System.IO.DriveNotFoundException	从 .NET Core 2.0.4 开始。
System.IO.EndOfStreamException	从 .NET Core 2.0.4 开始。
System.IO.FileFormatException	从 .NET Core 2.0.4 开始。
System.IO.FileLoadException	从 .NET Core 2.0.4 开始。
System.IO.FileNotFoundException	从 .NET Core 2.0.4 开始。
System.IO.IOException	从 .NET Core 2.0.4 开始。
System.IO.InternalBufferOverflowException	从 .NET Core 2.0.4 开始。
System.IO.InvalidDataException	从 .NET Core 2.0.4 开始。
System.IO.IsolatedStorage.IsolatedStorageException	从 .NET Core 2.0.4 开始。
System.IO.PathTooLongException	从 .NET Core 2.0.4 开始。
System.IndexOutOfRangeException	从 .NET Core 2.0.4 开始。
System.InsufficientExecutionStackException	从 .NET Core 2.0.4 开始。
System.InsufficientMemoryException	从 .NET Core 2.0.4 开始。
System.Int16	
System.Int32	
System.Int64	
System.IntPtr	
System.InvalidCastException	从 .NET Core 2.0.4 开始。
System.InvalidOperationException	从 .NET Core 2.0.4 开始。
System.InvalidProgramException	从 .NET Core 2.0.4 开始。
System.InvalidTimeZoneException	从 .NET Core 2.0.4 开始。

从	从
System.MemberAccessException	从 .NET Core 2.0.4 开始。
System.MethodAccessException	从 .NET Core 2.0.4 开始。
System.MissingFieldException	从 .NET Core 2.0.4 开始。
System.MissingMemberException	从 .NET Core 2.0.4 开始。
System.MissingMethodException	从 .NET Core 2.0.4 开始。
System.MulticastNotSupportedException	从 .NET Core 2.0.4 开始。
System.Net.Cookie	
System.Net.CookieCollection	
System.Net.CookieContainer	
System.Net.CookieException	从 .NET Core 2.0.4 开始。
System.Net.HttpListenerException	从 .NET Core 2.0.4 开始。
System.Net.Mail.SmtpException	从 .NET Core 2.0.4 开始。
System.Net.Mail.SmtpFailedRecipientException	从 .NET Core 2.0.4 开始。
System.Net.Mail.SmtpFailedRecipientsException	从 .NET Core 2.0.4 开始。
System.Net.NetworkInformation.NetworkInformationException	从 .NET Core 2.0.4 开始。
System.Net.NetworkInformation.PingException	从 .NET Core 2.0.4 开始。
System.Net.ProtocolViolationException	从 .NET Core 2.0.4 开始。
System.Net.Sockets.SocketException	从 .NET Core 2.0.4 开始。
System.Net.WebException	从 .NET Core 2.0.4 开始。
System.Net.WebSockets.WebSocketException	从 .NET Core 2.0.4 开始。
System.NotFiniteNumberException	从 .NET Core 2.0.4 开始。
System.NotImplementedException	从 .NET Core 2.0.4 开始。
System.NotSupportedException	从 .NET Core 2.0.4 开始。
System.NullReferenceException	从 .NET Core 2.0.4 开始。
System.Nullable<T>	

从	从
System.Numerics.BigInteger	
System.Numerics.Complex	
System.Object	
System.ObjectDisposedException	从 .NET Core 2.0.4 开始。
System.OperationCanceledException	从 .NET Core 2.0.4 开始。
System.OutOfMemoryException	从 .NET Core 2.0.4 开始。
System.OverflowException	从 .NET Core 2.0.4 开始。
System.PlatformNotSupportedException	从 .NET Core 2.0.4 开始。
System.RankException	从 .NET Core 2.0.4 开始。
System.Reflection.AmbiguousMatchException	从 .NET Core 2.0.4 开始。
System.Reflection.CustomAttributeFormatException	从 .NET Core 2.0.4 开始。
System.Reflection.InvalidFilterCriteriaException	从 .NET Core 2.0.4 开始。
System.Reflection.ReflectionTypeLoadException	从 .NET Core 2.0.4 开始。 不支持从 .NET Framework 到 .NET Core 的序列化。
System.Reflection.TargetException	从 .NET Core 2.0.4 开始。
System.Reflection.TargetInvocationException	从 .NET Core 2.0.4 开始。
System.Reflection.TargetParameterCountException	从 .NET Core 2.0.4 开始。
System.Resources.MissingManifestResourceException	从 .NET Core 2.0.4 开始。
System.Resources.MissingSatelliteAssemblyException	从 .NET Core 2.0.4 开始。
System.Runtime.CompilerServices.RuntimeWrappedException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.COMException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.ExternalException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.InvalidComObjectException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.InvalidOleVariantTypeException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.MarshalDirectiveException	从 .NET Core 2.0.4 开始。

从	从
System.Runtime.InteropServices.SEHException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.SafeArrayRankMismatchException	从 .NET Core 2.0.4 开始。
System.Runtime.InteropServices.SafeArrayTypeMismatchException	从 .NET Core 2.0.4 开始。
System.Runtime.Serialization.InvalidDataContractException	从 .NET Core 2.0.4 开始。
System.Runtime.Serialization.SerializationException	从 .NET Core 2.0.4 开始。
System.SByte	
System.Security.AccessControl.PrivilegeNotHeldException	从 .NET Core 2.0.4 开始。
System.Security.Authentication.AuthenticationException	从 .NET Core 2.0.4 开始。
System.Security.Authentication.InvalidCredentialException	从 .NET Core 2.0.4 开始。
System.Security.Cryptography.CryptographicException	从 .NET Core 2.0.4 开始。
System.Security.Cryptography.CryptographicUnexpectedOperationException	从 .NET Core 2.0.4 开始。
`System.Security.Cryptography.Xml.CryptoSignedXmlRecursionException`	从 .NET Core 2.0.4 开始。
System.Security.HostProtectionException	从 .NET Core 2.0.4 开始。
System.Security.Policy.PolicyException	从 .NET Core 2.0.4 开始。
System.Security.Principal.IdentityNotMappedException	从 .NET Core 2.0.4 开始。
System.Security.SecurityException	从 .NET Core 2.0.4 开始。 有限的序列化数据。
System.Security.VerificationException	从 .NET Core 2.0.4 开始。
System.Security.XmlSyntaxException	从 .NET Core 2.0.4 开始。
System.ServiceProcess.TimeoutException	从 .NET Core 2.0.4 开始。
System.Single	
System.StackOverflowException	从 .NET Core 2.0.4 开始。
System.String	
System.StringComparer	
System.SystemException	从 .NET Core 2.0.4 开始。

𝚝	𝚝
System.Text.DecoderFallbackException	从 .NET Core 2.0.4 开始。
System.Text.EncoderFallbackException	从 .NET Core 2.0.4 开始。
System.Text.RegularExpressions.RegexMatchTimeoutException	从 .NET Core 2.0.4 开始。
System.Text.StringBuilder	
System.Threading.AbandonedMutexException	从 .NET Core 2.0.4 开始。
System.Threading.BarrierPostPhaseException	从 .NET Core 2.0.4 开始。
System.Threading.LockRecursionException	从 .NET Core 2.0.4 开始。
System.Threading.SemaphoreFullException	从 .NET Core 2.0.4 开始。
System.Threading.SynchronizationLockException	从 .NET Core 2.0.4 开始。
System.Threading.Tasks.TaskCanceledException	从 .NET Core 2.0.4 开始。
System.Threading.Tasks.TaskSchedulerException	从 .NET Core 2.0.4 开始。
System.Threading.ThreadAbortException	从 .NET Core 2.0.4 开始。
System.Threading.ThreadInterruptedException	从 .NET Core 2.0.4 开始。
System.Threading.ThreadStartException	从 .NET Core 2.0.4 开始。
System.Threading.ThreadStateException	从 .NET Core 2.0.4 开始。
System.Threading.WaitHandleCannotBeOpenedException	从 .NET Core 2.0.4 开始。
System.TimeSpan	
System.TimeZoneInfo.AdjustmentRule	
System.TimeZoneInfo	
System.TimeZoneNotFoundException	从 .NET Core 2.0.4 开始。
System.TimeoutException	从 .NET Core 2.0.4 开始。
System.Transactions.TransactionAbortedException	从 .NET Core 2.0.4 开始。
System.Transactions.TransactionException	从 .NET Core 2.0.4 开始。
System.Transactions.TransactionInDoubtException	从 .NET Core 2.0.4 开始。

𝙸	𝙸
System.Transactions.TransactionManagerCommunicationException	从 .NET Core 2.0.4 开始。
System.Transactions.TransactionPromotionException	从 .NET Core 2.0.4 开始。
System.Tuple	
System.TypeAccessException	从 .NET Core 2.0.4 开始。
System.TypeInitializationException	从 .NET Core 2.0.4 开始。
System.TypeLoadException	从 .NET Core 2.0.4 开始。
System.TypeUnloadedException	从 .NET Core 2.0.4 开始。
System.UInt16	
System.UInt32	
System.UInt64	
System.UIntPtr	
System.UnauthorizedAccessException	从 .NET Core 2.0.4 开始。
System.Uri	
System.UriFormatException	从 .NET Core 2.0.4 开始。
System.ValueTuple	在 .NET Framework 4.7 及早期版本中不可序列化。
System.ValueType	
System.Version	
System.WeakReference<T>	
System.WeakReference	
System.Xml.Schema.XmlSchemaException	从 .NET Core 2.0.4 开始。
System.Xml.Schema.XmlSchemaInferenceException	从 .NET Core 2.0.4 开始。
System.Xml.Schema.XmlSchemaValidationException	从 .NET Core 2.0.4 开始。
System.Xml.XPath.XPathException	从 .NET Core 2.0.4 开始。
System.Xml.XmlException	从 .NET Core 2.0.4 开始。
System.Xml.Xsl.XsltCompileException	从 .NET Core 2.0.4 开始。

“	“
System.Xml.Xsl.XsltException	从 .NET Core 2.0.4 开始。

请参阅

- System.Runtime.Serialization
  包含可用于序列化和反序列化对象的类。
- XML 和 SOAP 序列化
  描述随公共语言运行库一起提供的 XML 序列化机制。
- 安全和序列化
  描述写入执行序列化的代码时需要遵循的安全编码原则。
- .NET 远程处理
  描述 .NET Framework 中提供的用于远程通信的多种方法。
- 使用 ASP.NET 创建的 XML Web service 以及 XML Web service 客户端
  描述并解释如何对使用 ASP.NET 创建的 XML Web services 进行编程的文章。

BinaryFormatter 安全指南

2021/11/16 •

本文适用于以下 .NET 实现：

- 所有版本的 .NET Framework
- .NET Core 2.1 - 3.1
- .NET 5.0 及更高版本

背景

> **WARNING**
> BinaryFormatter 类型会带来风险，不建议将其用于数据处理。即使应用程序认为自己正在处理的数据是可信的，也应尽快停止使用 `BinaryFormatter`。 `BinaryFormatter` 不安全，无法确保安全。

本文适用于以下类型：

- SoapFormatter
- NetDataContractSerializer
- LosFormatter
- ObjectStateFormatter

反序列化漏洞是指不安全地处理请求有效负载的威胁类别。成功利用这些漏洞攻击应用的攻击者可导致目标应用内出现拒绝服务 (DoS)、信息泄露或远程代码执行。此风险类别始终是 10 项最严重的 OWASP 风险之一。攻击目标包括使用多种语言（包括 C/C++、Java 和 C#）编写的应用。

在 .NET 中，风险最大的目标是使用 BinaryFormatter 类型来反序列化数据的应用。 `BinaryFormatter` 因为其强大的功能和易用性而广泛用于整个 .NET 生态系统。但是，其强大的功能也让攻击者能够影响目标应用内的控制流。成功的攻击可能导致攻击者能够在目标进程的上下文中运行代码。

更简单的比喻是，假设在有效负载上调用 `BinaryFormatter.Deserialize` 相当于将该有效负载解释为独立的可执行文件并启动它。

BinaryFormatter 安全漏洞

> **WARNING**
> 将 `BinaryFormatter.Deserialize` 方法用于不受信任的输入时，该方法永远都不安全。强烈建议使用者改为考虑使用本文后面概述的替代方法之一。

`BinaryFormatter` 是在反序列化漏洞成为一个众所周知的威胁类别之前实现的。因此，代码不遵循现代最佳做法。 `Deserialize` 方法可用作攻击者对使用中的应用执行 DoS 攻击的载体。这些攻击可能导致应用无响应或进程意外终止。使用 `SerializationBinder` 或任何其他 `BinaryFormatter` 配置开关都无法缓解此类攻击。.NET 认为此行为是设计使然，因此不会发布代码更新来修改此行为。

使用 `BinaryFormatter.Deserialize` 可能容易遭受其他攻击类别的攻击，如信息泄露或远程代码执行。利用自定义 SerializationBinder 等功能可能不足以适当缓解这些风险。存在发现新漏洞的可能性，而 .NET 实际上无法为此发布安全更新。使用者应该评估其各个应用场景，并考虑他们遇到这些风险的可能性。

我们建议 `BinaryFormatter` 使用者对其应用执行单独的风险评估。由使用者完全负责确定是否利用 `BinaryFormatter`。使用者应该对使用 `BinaryFormatter` 的安全性、技术、声誉、法律和监管要求进行风险评估。

首选替代方法

.NET 提供了多个随附的序列化程序，可用于安全处理不受信任的数据：

- XmlSerializer 和 DataContractSerializer，用于将对象图序列化为 XML 或从 XML 序列化对象图。不要将 `DataContractSerializer` 与 NetDataContractSerializer 混淆。
- BinaryReader 和 BinaryWriter，适用于 XML 和 JSON。
- System.Text.Json API，用于将对象图序列化为 JSON。

危险的替代方法

避免使用以下序列化程序：

- SoapFormatter
- LosFormatter
- NetDataContractSerializer
- ObjectStateFormatter

上述序列化程序都执行不受限制的多态反序列化，并且会带来风险，就像 `BinaryFormatter` 一样。

假设数据值得信任的风险

通常，应用开发人员可能会认为他们只是在处理受信任的输入。在一些罕见的情况下，可实现真正的安全输入。但更常见的情况是，有效负载跨越了信任边界，而开发人员却没有意识到这一点。

考虑本地服务器，员工在其中使用其工作站的桌面客户端与服务进行交互。这个场景可能被天真地视为可以接受使用 `BinaryFormatter` 的"安全"设置。但是，这个场景为恶意软件提供了一个载体，使恶意软件能够访问单个员工的计算机，从而能够在整个企业中传播。该恶意软件可以利用企业使用 `BinaryFormatter` 造成的漏洞，从员工的工作站横向移动到后端服务器。然后，它可以泄露公司的敏感数据。此类数据可能包括商业机密或客户数据。

还考虑使用借助 `BinaryFormatter` 来保持保存状态的应用。最初看来这似乎是一个安全的方案，因为在你自己的硬盘驱动器上读写数据威胁较低。但是，通过电子邮件或 Internet 共享文档是很常见的，并且大多数最终用户不会认为打开这些下载的文件属于危险行为。

攻击者可以利用此场景来制造恶意结果。如果应用是一款游戏，则共享保存文件的用户会在不知情的情况下面临风险。开发者自身也可能成为目标。攻击者可能会通过电子邮件向开发者的技术支持人员发送电子邮件，并添加恶意数据文件作为附件，然后要求支持人员打开该文件。这种攻击可以为攻击者提供一个在企业中的据点。

另一种场景是数据文件存储在云存储空间中，并在用户的计算机之间自动同步。能够访问云存储帐户的攻击者可以对数据文件进行病毒攻击。此数据文件将自动同步到用户的计算机。用户下一次打开数据文件时，攻击者的有效负载就会运行。因此，攻击者可以利用云存储帐户泄露来获得完整的代码执行权限。

考虑从桌面安装模型迁移到云优先模型的应用。此场景包括从桌面应用或丰富客户端模型迁移到基于 Web 的模型的应用。任何为桌面应用绘制的威胁模型都不一定适用于基于云的服务。桌面应用的威胁模型可能会消除某个给定的威胁，因为"客户端对攻击自己不感兴趣"。但是，当考虑到远程用户（客户端）攻击云服务本身时，同样的威胁可能会变得有意义。

> **NOTE**
> 通常，序列化的目的是将对象传入或传出应用。威胁建模练习几乎始终将此类数据传输标记为跨越信任边界。

其他资源

- YSoSerial.Net，提供有关攻击者如何使用 `BinaryFormatter` 来攻击应用的研究。
- 反序列化漏洞的一般背景：
  - 10 项最严重的 OWASP 风险 - A8:2017 不安全的反序列化
  - CWE-502：不受信任的数据的反序列化

BinaryFormatter 事件源

2021/11/16 •

从 .NET 5.0 开始，BinaryFormatter 包含内置的 EventSource，你可通过此功能了解进行对象序列化或反序列化的时间。应用可以使用 EventListener 派生的类型侦听这些通知，并进行记录。

此功能不能代替 SerializationBinder 或 ISerializationSurrogate，不能用于修改要序列化或反序列化的数据。此事件系统用于深入了解要序列化或反序列化的类型。它还可用于检测对 `BinaryFormatter` 基础结构的意外调用，如来自第三方库代码的调用。

事件说明

`BinaryFormatter` 事件源的已知名称为 `System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource`。侦听器可以订阅 6 个事件。

SerializationStart 事件（ID 为 `10`）

在 BinaryFormatter.Serialize 已调用并且序列化进程已启动时引发。此事件与 `SerializationEnd` 事件成对出现。对象在其自己的序列化例程中调用 `SerializationStart` 时，`BinaryFormatter.Serialize` 事件可以递归方式调用。

此事件不包含有效负载。

SerializationEnd 事件（ID 为 `11`）

在 `BinaryFormatter.Serialize` 已完成其工作时引发。`SerializationEnd` 每次出现时，都表示最后一个未成对的 `SerializationStart` 事件完成。

此事件不包含有效负载。

SerializingObject 事件（ID 为 `12`）

在 `BinaryFormatter.Serialize` 正在序列化非基元类型时引发。`BinaryFormatter` 基础结构会对某些类型（如 `string` 和 `int`）进行特殊处理，当它遇到这些类型时不会引发此事件。此事件将针对用户定义的类型和 `BinaryFormatter` 本机无法理解的其他类型引发。

此事件可在 `SerializationStart` 和 `SerializationEnd` 事件之间引发零次或多次。

此事件包含有效负载，其中包括一个参数：

- `typeName`（`string`）：要序列化的类型的程序集限定名称（请参阅 Type.AssemblyQualifiedName）。

DeserializationStart 事件（ID 为 `20`）

在 BinaryFormatter.Deserialize 已调用并且反序列化进程已启动时引发。此事件与 `DeserializationEnd` 事件成对出现。对象在其自己的反序列化例程中调用 `DeserializationStart` 时，`BinaryFormatter.Deserialize` 事件可以递归方式调用。

此事件不包含有效负载。

DeserializationEnd 事件（ID 为 `21`）

在 `BinaryFormatter.Deserialize` 已完成其工作时引发。`DeserializationEnd` 每次出现时，都表示最后一个未成对的 `DeserializationStart` 事件完成。

此事件不包含有效负载。

DeserializingObject 事件（ID 为 `22`）

在 `BinaryFormatter.Deserialize` 正在反序列化非基元类型时引发。`BinaryFormatter` 基础结构会对某些类型(如 `string` 和 `int`)进行特殊处理，当它遇到这些类型时不会引发此事件。此事件将针对用户定义的类型和 `BinaryFormatter` 本机无法理解的其他类型引发。

此事件可在 `DeserializationStart` 和 `DeserializationEnd` 事件之间引发零次或多次。

此事件包含有效负载，其中包括一个参数。

- `typeName` (`string`):要反序列化的类型的程序集限定名称(请参阅 Type.AssemblyQualifiedName)。

[高级] 订阅部分通知

仅需要订阅部分通知的侦听器可以选择要启用的关键字。

- `Serialization` = `(EventKeywords)1`:引发 `SerializationStart`、`SerializationEnd` 和 `SerializingObject` 事件。
- `Deserialization` = `(EventKeywords)2`:引发 `DeserializationStart`、`DeserializationEnd` 和 `DeserializingObject` 事件。

如果在 `EventListener` 注册过程中未提供关键字筛选器，则系统将引发所有事件。

有关详细信息，请参阅 System.Diagnostics.Tracing.EventKeywords。

示例代码

下面的代码:

- 创建将写入 `System.Console` 的 `EventListener` 派生的类型，
- 为该侦听器订阅 `BinaryFormatter` 生成的通知，
- 使用 `BinaryFormatter` 序列化和反序列化简单的对象图，并
- 分析已引发的事件。

```
using System;
using System.Diagnostics.Tracing;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace BinaryFormatterEventSample
{
    class Program
    {
        static EventListener _globalListener = null;

        static void Main(string[] args)
        {
            // First, set up the event listener.
            // Note: We assign it to a static field so that it doesn't get GCed.
            // We also provide a callback that subscribes this listener to all
            // events produced by the well-known BinaryFormatter source.

            _globalListener = new ConsoleEventListener();
            _globalListener.EventSourceCreated += (sender, args) =>
            {
                if (args.EventSource?.Name ==
                    "System.Runtime.Serialization.Formatters.Binary.BinaryFormatterEventSource")
                {
                    ((EventListener)sender)
                        .EnableEvents(args.EventSource, EventLevel.LogAlways);
                }
            };

            // Next, create the Person object and serialize it.
```

```
            Person originalPerson = new Person()
            {
                FirstName = "Logan",
                LastName = "Edwards",
                FavoriteBook = new Book()
                {
                    Title = "A Tale of Two Cities",
                    Author = "Charles Dickens",
                    Price = 10.25m
                }
            };

            byte[] serializedPerson = SerializePerson(originalPerson);

            // Finally, deserialize the Person object.

            Person rehydratedPerson = DeserializePerson(serializedPerson);

            Console.WriteLine
                ($"Rehydrated person {rehydratedPerson.FirstName} {rehydratedPerson.LastName}");
            Console.Write
                ($"Favorite book: {rehydratedPerson.FavoriteBook.Title} ");
            Console.Write
                ($"by {rehydratedPerson.FavoriteBook.Author}, ");
            Console.WriteLine
                ($"list price {rehydratedPerson.FavoriteBook.Price}");
        }

        private static byte[] SerializePerson(Person p)
        {
            MemoryStream memStream = new MemoryStream();
            BinaryFormatter formatter = new BinaryFormatter();
#pragma warning disable SYSLIB0011 // BinaryFormatter.Serialize is obsolete
            formatter.Serialize(memStream, p);
#pragma warning restore SYSLIB0011

            return memStream.ToArray();
        }

        private static Person DeserializePerson(byte[] serializedData)
        {
            MemoryStream memStream = new MemoryStream(serializedData);
            BinaryFormatter formatter = new BinaryFormatter();

#pragma warning disable SYSLIB0011 // Danger: BinaryFormatter.Deserialize is insecure for untrusted input
            return (Person)formatter.Deserialize(memStream);
#pragma warning restore SYSLIB0011
        }
    }

    [Serializable]
    public class Person
    {
        public string FirstName;
        public string LastName;
        public Book FavoriteBook;
    }

    [Serializable]
    public class Book
    {
        public string Title;
        public string Author;
        public decimal Price;
    }

    // A sample EventListener that writes data to System.Console.
    public class ConsoleEventListener : EventListener
    {
```

```
    {
        protected override void OnEventWritten(EventWrittenEventArgs eventData)
        {
            base.OnEventWritten(eventData);

            Console.WriteLine($"Event {eventData.EventName} (id={eventData.EventId}) received.");
            if (eventData.PayloadNames != null)
            {
                for (int i = 0; i < eventData.PayloadNames.Count; i++)
                {
                    Console.WriteLine($"{eventData.PayloadNames[i]} = {eventData.Payload[i]}");
                }
            }
        }
    }
}
```

上面的代码生成的输出与下面的示例相似：

```
Event SerializationStart (id=10) received.
Event SerializingObject (id=12) received.
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
Event SerializingObject (id=12) received.
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
Event SerializationStop (id=11) received.
Event DeserializationStart (id=20) received.
Event DeserializingObject (id=22) received.
typeName = BinaryFormatterEventSample.Person, BinaryFormatterEventSample, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
Event DeserializingObject (id=22) received.
typeName = BinaryFormatterEventSample.Book, BinaryFormatterEventSample, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
Event DeserializationStop (id=21) received.
Rehydrated person Logan Edwards
Favorite book: A Tale of Two Cities by Charles Dickens, list price 10.25
```

在此示例中，基于控制台的 `EventListener` 将对序列化启动、`Person` 和 `Book` 的实例序列化以及序列化完成进行记录。同样，反序列化启动后，`Person` 和 `Book` 的实例将进行反序列化，然后反序列化完成。

然后，应用将打印已反序列化的 `Person` 中包含的值，以证明对象确实已正确地进行序列化和反序列化。

请参阅

有关使用 `EventListener` 接收基于 `EventSource` 的通知的详细信息，请参阅 `EventListener` 类。

序列化概念

2021/11/16 •

为什么要使用序列化？ 两个最重要的原因是将对象状态保存到存储媒体，以便可以在以后阶段重新创建精确副本；以及将对象按值从一个应用程序域发送至另一个应用程序域。例如，序列化用于在 ASP.NET 中保存会话状态，并将对象复制到 Windows 窗体的剪贴板中。它还可用于在远程处理中将对象按值从一个应用程序域传递至另一个应用程序域。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

永久性存储

经常有必要将对象的字段值存储至磁盘，以后再检索此数据。尽管不依赖序列化也能很容易地实现这一点，但方法通常麻烦而容易出错，并且需要跟踪对象的层次结构时会逐渐变得更加复杂。假设要编写一个包含数千个对象的大型商务应用程序，并且必须为每个对象编写代码，以便将字段和属性保存至磁盘以及从磁盘进行还原。序列化为实现这一目标提供了方便的机制。

公共语言运行库可管理对象在内存中存储的方式，并通过使用反射提供一种自动序列化机制。当序列化对象时，类的名称、程序集和类实例的所有数据成员被写入存储区。对象经常以成员变量方式将引用存储至其他实例。当序列化类时，序列化引擎跟踪被引用的对象（已序列化），以确保同一对象不会被多次序列化。.NET 提供的序列化体系结构可自动正确地处理对象图和循环引用。对于对象图的唯一要求是，必须将已序列化对象引用的所有对象也标记为 `Serializable`（有关详细信息，请参阅基本序列化）。如果未进行标记，序列化程序尝试序列化未标记的对象时将引发异常。

反序列化已序列化的类时，将重新创建该类，并且将自动还原所有数据成员的值。

按值封送

对象仅在创建它们的应用程序域中有效。除非对象从 `MarshalByRefObject` 派生或被标记为 `Serializable`，否则尝试将对象作为参数传递或者作为结果返回时都将失败。如果将对象标记为 `Serializable`，则会自动序列化该对象，将该对象从一个应用程序域传输至另一个应用程序域，然后反序列化，以便在第二个应用程序域中生成该对象的一个精确副本。此过程通常称为按值封送。

如果对象从 `MarshalByRefObject` 派生，则会将对象引用（而不是对象本身）从一个应用程序域传递至另一个应用程序域。还可将派生自 `MarshalByRefObject` 的对象标记为 `Serializable`。此对象用于远程处理时，负责反序列化的格式化程序已通过代理项选择器（`SurrogateSelector`）预配置，该格式化程序控制序列化过程，并用代理替换从 `MarshalByRefObject` 派生的所有对象。如果本地没有 `SurrogateSelector`，则序列化体系结构遵循序列化过程中的步骤中所述的标准序列化规则。

相关章节

二进制序列化
描述随公共语言运行库一起提供的二进制序列化机制。

XML 和 SOAP 序列化
描述随公共语言运行库一起提供的 XML 和 SOAP 序列化机制。

基本序列化

2021/11/16 •

WARNING

二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

使类可序列化最简单的方法是按以下方式使用 SerializableAttribute 对其进行标记。

```
[Serializable]
public class MyObject {
  public int n1 = 0;
  public int n2 = 0;
  public String str = null;
}
```

下面的代码示例演示如何将此类的实例序列化为文件。

```
MyObject obj = new MyObject();
obj.n1 = 1;
obj.n2 = 24;
obj.str = "Some String";
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Create, FileAccess.Write, FileShare.None);
formatter.Serialize(stream, obj);
stream.Close();
```

此示例使用二进制格式化程序执行序列化。只需创建该流的实例及要使用的格式化程序，然后在该格式化程序上调用 Serialize 方法。要序列化的流和对象将作为参数提供给此调用。尽管未在此示例中明确演示，但类的所有成员变量都将被序列化，即使将变量标记为私有也是如此。在这一方面，二进制序列化与 XmlSerializer 类不同，后者只序列化公共字段。有关从二进制序列化排除成员变量的信息，请参阅选择性的序列化。

将对象还原回以前的状态同样很容易。首先，创建用于读取的流和 Formatter，然后指导格式化程序对该对象进行反序列化。下面的代码示例演示如何完成以上过程。

```
IFormatter formatter = new BinaryFormatter();
Stream stream = new FileStream("MyFile.bin", FileMode.Open, FileAccess.Read, FileShare.Read);
MyObject obj = (MyObject) formatter.Deserialize(stream);
stream.Close();

// Here's the proof.
Console.WriteLine("n1: {0}", obj.n1);
Console.WriteLine("n2: {0}", obj.n2);
Console.WriteLine("str: {0}", obj.str);
```

上面使用的 BinaryFormatter 非常有效，还可生成压缩字节流。使用此格式化程序序列化的所有对象也可使用它进行反序列化，这使得它成为对将在 .NET 上反序列化的对象进行序列化的理想工具。需要特别注意的是，反序列化对象时不调用构造函数。这是出于性能原因而对反序列化进行的约束。然而，这违反了运行库对对象编写器制定的某些常用协定，并且将对象标记为可序列化时开发人员应确保了解其后果。

如果要求可迁移性，请改用 SoapFormatter。只需将以上代码中的 BinaryFormatter 替换为 SoapFormatter，然后同前面一样调用 Serialize 和 Deserialize 。此格式化程序针对以上使用的示例产生下面的输出：

```
<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:a1="http://schemas.microsoft.com/clr/assem/ToFile">

  <SOAP-ENV:Body>
    <a1:MyObject id="ref-1">
      <n1>1</n1>
      <n2>24</n2>
      <str id="ref-3">Some String</str>
    </a1:MyObject>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

请注意，无法继承 Serializable 属性。如果从 `MyObject` 派生新类，新类也必须标记为以上特性，否则将无法序列化。例如，尝试序列化以下类的实例时，将收到 SerializationException 信息，提示 `MyStuff` 类型未标记为可序列化。

```
public class MyStuff : MyObject
{
  public int n3;
}
```

使用 Serializable 属性非常方便，但有以上所述的限制。有关何时应标记类以进行序列化的信息，请参阅序列化准则。若类已编译，则无法对其进行序列化。

请参阅

- 二进制序列化
- XML 和 SOAP 序列化

有选择的序列化

2021/11/16 •

类通常包含不应进行序列化的字段。例如，假定类将线程 ID 存储在成员变量中。如果反序列化该类，而在对该类进行序列化时，存储了该 ID 的线程可能不再运行。因此，序列化该值毫无意义。按如下方法使用 NonSerialized 属性标记成员变量，可防止其被序列化。

```
[Serializable]
public class MyObject
{
  public int n1;
  [NonSerialized] public int n2;
  public String str;
}
```

如有可能，应使可能包含安全敏感数据的对象不可序列化。如果必须序列化该对象，则可将 `NonSerialized` 属性应用于存储敏感数据的特定字段。如果没有将这些字段排除在序列化之外，应该注意字段存储的数据会向有权序列化的所有代码公开。有关编写安全的序列化代码的详细信息，请参阅安全和序列化。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

请参阅

- 二进制序列化
- XML 和 SOAP 序列化
- 安全性和序列化

自定义序列化

2021/11/16 •

自定义序列化是控制类型的序列化和反序列化的过程。通过控制序列化，可以确保序列化兼容性。换而言之，在不中断类型核心功能的情况下，可在类型的不同版本之间序列化和反序列化。例如，在类型的第一个版本中，可能只有两个字段。在类型的下一个版本中，添加了其他几个字段。但是，第二个版本的应用程序必须可对这两种类型进行序列化和反序列化。以下各节说明如何控制序列化。

WARNING

二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

IMPORTANT

在早于 .NET Framework 4.0 的版本中，部分受信任的程序集中自定义用户数据的序列化是使用 `GetObjectData` 完成的。从版本 4.0 开始，该方法将标记有 SecurityCriticalAttribute 特性，该特性阻止在部分受信任的程序集中执行。若要解决此情况，请实现 ISafeSerializationData 接口。

在序列化期间和序列化之后运行自定义方法

序列化期间和序列化之后运行自定义方法的建议方法，是在序列化期间和序列化之后，将下列属性应用于更正数据所用的方法：

- OnDeserializedAttribute
- OnDeserializingAttribute
- OnSerializedAttribute
- OnSerializingAttribute

这些属性允许类型参与序列化和反序列化过程中的任何一个阶段或所有四个阶段。这些特性为类型指定了应该在每个阶段调用的方法。这些方法不会访问序列化流，但是反而允许您在序列化或反序列化前后更改对象。可以在类型继承层次结构中的所有级别上应用这些特性，而每种方法在该层次结构中是按从基类型到派生程度最大的类型的顺序调用的。这种机制使得序列化和反序列化可以负责到派生程度最大的实现，从而可以避免实现 ISerializable 接口时的复杂性和任何导致的问题。此外，这种机制允许格式化程序忽略字段的填充以及从序列化流中检索。有关控制序列化和反序列化的详细信息和示例，请单击以上任一链接。

另外，向现有的可序列化类型中添加新字段时，可将 OptionalFieldAttribute 特性应用于该字段。处理缺少新字段的流时，BinaryFormatter 和 SoapFormatter 可以忽略不存在该字段的情况。

实现 ISerializable 接口

控制序列化的另一种方法是对某个对象实现 ISerializable 接口。但请注意，上一节采用的方法会取代这种方法对序列化进行控制。

除此之外，如果使用 Serializable 属性对某个类进行标记，且该类在类级别或对其构造函数具有声明性或命令性安全，则不应对该类执行默认的序列化。相反，这样的类应该始终实现 ISerializable 接口。

实现 ISerializable 涉及到实现 `GetObjectData` 方法以及反序列化对象时所用的特殊构造函数。下面的示例代码演示如何对上一节中的 ISerializable 类实现 `MyObject`。

```
[Serializable]
public class MyObject : ISerializable
{
    public int n1;
    public int n2;
    public String str;

    public MyObject()
    {
    }

    protected MyObject(SerializationInfo info, StreamingContext context)
    {
      n1 = info.GetInt32("i");
      n2 = info.GetInt32("j");
      str = info.GetString("k");
    }

    [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    public virtual void GetObjectData(SerializationInfo info, StreamingContext context)
    {
        info.AddValue("i", n1);
        info.AddValue("j", n2);
        info.AddValue("k", str);
    }
}
```

```
<Serializable()>  _
Public Class MyObject
    Implements ISerializable
    Public n1 As Integer
    Public n2 As Integer
    Public str As String

    Public Sub New()
    End Sub

    Protected Sub New(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        n1 = info.GetInt32("i")
        n2 = info.GetInt32("j")
        str = info.GetString("k")
    End Sub 'New

    <SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter := True)> _
    Public Overridable Sub GetObjectData(ByVal info As SerializationInfo, ByVal context As StreamingContext)
        info.AddValue("i", n1)
        info.AddValue("j", n2)
        info.AddValue("k", str)
    End Sub
End Class
```

序列化期间调用 GetObjectData 时，由用户负责填充随方法调用一起提供的 SerializationInfo。将待序列化变量以名称和值对的形式添加。任何文本都能用作名称。如果对足够多的数据进行序列化，以在反序列化期间还原对象，您便可随意决定将哪些成员变量添加到 SerializationInfo。如果基对象实现 ISerializable，则派生类应该对该基对象调用 GetObjectData 方法。

请注意，序列化可以允许其他代码查看或修改用其他方式无法访问的对象实例数据。因此，执行序列化的代码需使用指定了 SerializationFormatter 标志的 SecurityPermission。在默认策略下，通过 Internet 下载的代码或 Intranet 代码不会授予该权限；只有本地计算机上的代码才被授予该权限。必须采用下列方式显式保护 GetObjectData 方法：要求使用使用指定了 SerializationFormatter 标志的 SecurityPermission，或者要求具备专门用于帮助保护私有数据的其他权限。

若私有字段存储的是敏感信息，应该要求对 GetObjectData 具备相应权限，以便保护该数据。请记住，如果已向代码授予指定了 SerializationFormatter 标志的 SecurityPermission，则该代码可查看和修改私有字段中存储的数据。被授予此 SecurityPermission 的恶意调用方可以查看相关数据（如隐藏的目录位置或授予的权限），还可以通过这些数据利用计算机中的安全漏洞。有关可以指定的安全权限标志的完整列表，请参阅 SecurityPermissionFlag 枚举。

强调何时将 ISerializable 添加到必须同时实现 **GetObjectData** 和特殊构造函数的某个类，这一点很重要。如果缺少 GetObjectData，编译器会发出警告。但是，鉴于无法强制实现构造函数，如果不存在该构造函数，则不会发出任何警告，但此时如果尝试对某个类进行反序列化，将会引发异常。

若要解决潜在的安全和版本化问题，当前的设计应该优先于 SetObjectData 方法。例如，如果将 `SetObjectData` 方法定义为接口的组成部分，则该方法必须是公共方法。因此，用户必须编写代码，以防多次调用 SetObjectData 方法。否则，在执行某项操作的过程中，对某个对象调用 SetObjectData 方法的恶意应用程序可能会导致潜在的问题。

在反序列化期间，使用为这个目的提供的构造函数将 SerializationInfo 传递到类。反序列化该对象时，将会忽略对该构造函数施加的任何可见性限制。因此，可将该类标记为公共类、受保护的类、内部类或私有类。但是，除非密封了该类，否则最好将构造函数设为受保护的函数；如果密封了该类，应将构造函数标记为私有函数。构造函数还应该执行彻底的输入验证。为避免被恶意代码误用，构造函数应该强制实施安全性检查和权限，而这些安全性检查和权限也是使用其他任何构造函数获取该类的实例所必需的。如果不采纳上述建议，恶意代码会跳过使用公共构造函数在标准实例构造期间应用的所有安全性，预序列化对象，获取对使用指定了 SerializationFormatter 标志的 SecurityPermission 的控制，并反序列化客户端计算机上的对象。

若要还原对象的状态，只需使用序列化期间采用的名称从 SerializationInfo 中检索变量值即可。如果基类实现 ISerializable，则应调用基构造函数，以使基对象可以还原其变量。

从实现 ISerializable 的类派生新类时，如果派生类的变量需要进行序列化，则该派生类必须同时实现构造函数和 GetObjectData 方法。下面的代码示例演示如何使用前面说明的 `MyObject` 类完成此项操作。

```
[Serializable]
public class ObjectTwo : MyObject
{
    public int num;

    public ObjectTwo()
      : base()
    {
    }

    protected ObjectTwo(SerializationInfo si, StreamingContext context)
      : base(si, context)
    {
        num = si.GetInt32("num");
    }

    [SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter = true)]
    public override void GetObjectData(SerializationInfo si, StreamingContext context)
    {
        base.GetObjectData(si,context);
        si.AddValue("num", num);
    }
}
```

```
<Serializable()>  _
Public Class ObjectTwo
    Inherits MyObject
    Public num As Integer

    Public Sub New()

    End Sub

    Protected Sub New(ByVal si As SerializationInfo, _
    ByVal context As StreamingContext)
        MyBase.New(si, context)
        num = si.GetInt32("num")
    End Sub

    <SecurityPermissionAttribute(SecurityAction.Demand, SerializationFormatter := True)> _
    Public Overrides Sub GetObjectData(ByVal si As SerializationInfo, ByVal context As StreamingContext)
        MyBase.GetObjectData(si, context)
        si.AddValue("num", num)
    End Sub
End Class
```

不要忘记在反序列化构造函数中调用基类。如果没有完成此操作，决不会调用基类上的构造函数，也不会在反序列化之后完全构造该对象。

对象是从内到外重新构造的；在反序列化期间调用方法时，可能会产生非预期的副作用，原因是调用的方法引用的可能是执行调用时尚未反序列化的对象引用。如果正在被反序列化的类实现 IDeserializationCallback，则反序列化整个对象图后，将会自动调用 OnDeserialization 方法。此时，便已完全还原引用的所有子对象。哈希表是类的典型示例，在不使用事件侦听器的情况下，很难对哈希表进行反序列化。虽然在反序列化期间易于检索键和值对，但是，将这些对象重新添加到哈希表时，可能会引起问题，原因是不能保证派生自哈希表的类已被反序列化。因此，不建议在这个阶段对哈希表调用方法。

请参阅

- 二进制序列化
- XML 和 SOAP 序列化
- 安全性和序列化

序列化过程中的步骤

2021/11/16 •

对格式化程序调用 Serialize 方法时，将按照以下规则顺序进行对象序列化：

- 进行检查以确定格式化程序是否具有代理项选择器。如果有，则检查代理项选择器是否处理给定类型的对象。如果选择器处理该对象类型，则在代理项选择器上调用 ISerializable.GetObjectData。
- 如果没有代理项选择器，或者代理项选择器不处理该对象类型，则进行检查以确定是否用 Serializable 属性标记了该对象。如果未标记该对象，则会引发 SerializationException。
- 如果已相应地标记了对象，则检查该对象是否实现 ISerializable 接口。如果对象实现接口，则对该对象调用 GetObjectData。
- 如果对象未实现 ISerializable，则使用默认序列化策略，从而序列化所有未标记为 NonSerialized 的字段。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

请参阅

- 二进制序列化
- XML 和 SOAP 序列化

版本容错序列化

2021/11/16 •

在最早版本的 .NET Framework 中，创建可从应用程序的一个版本重用至下一个版本的可序列化类型时会产生问题。如果通过添加额外字段来修改类型，则会出现以下问题：

- 要求应用程序的较旧版本反序列化旧类型的新版本时，会引发异常。
- 应用程序的较新版本反序列化缺少数据的类型的较旧版本时，会引发异常。

版本容错序列化 (VTS) 是一组功能，它使得修改可序列化类型随着时间推移而变得更加容易。VTS 功能尤其是为应用了 SerializableAttribute 特性的类（包括泛型类型）而启用的。VTS 允许向这些类添加新字段，而不破坏与该类型其他版本的兼容性。

当使用 BinaryFormatter 时，将启用 VTS 功能。此外，当使用 SoapFormatter 时，也会启用除外来数据容错以外的其他所有功能。有关将这些类用于序列化的详细信息，请参见二进制序列化。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

功能列表

功能集包括：

- 外来或意外数据容错功能。该功能允许类型的较新版本将数据发送至较旧版本。
- 缺少可选数据容错功能。该功能允许较旧版本将数据发送至较新版本。
- 序列化回调。该功能在缺少数据的情况下启用智能默认值设置。

此外，添加了新的可选字段时还有声明功能。这是 VersionAdded 特性的 OptionalFieldAttribute 属性。

以下部分详细描述了这些功能。

外来或意外数据容错功能

过去在反序列化时，任何外来或意外数据都会导致引发异常。有了 VTS 后，在相同情况下会忽略所有外来或意外数据，而不会导致引发异常。这就使得使用类型的较新版本（即包含更多字段的版本）的应用程序可以将信息发送至需要使用同一类型的较旧版本的应用程序。

在下面的示例中，当较旧版本的应用程序反序列化较新版本时，将忽略 `CountryField` 类 2.0 版的 `Address` 中包含的额外数据。

```
// Version 1 of the Address class.
[Serializable]
public class Address
{
    public string Street;
    public string City;
}
// Version 2.0 of the Address class.
[Serializable]
public class Address
{
    public string Street;
    public string City;
    // The older application ignores this data.
    public string CountryField;
}
```

```
' Version 1 of the Address class.
<Serializable> _
Public Class Address
    Public Street As String
    Public City As String
End Class

' Version 2.0 of the Address class.
<Serializable> _
Public Class Address
    Public Street As String
    Public City As String
    ' The older application ignores this data.
    Public CountryField As String
End Class
```

丢失数据容错功能

通过将 OptionalFieldAttribute 特性应用于字段，可以将字段标记为可选。在反序列化期间，如果缺少可选的数据，序列化引擎会忽略这一缺失且不会引发异常。因此，需要使用类型的较旧版本的应用程序可以将数据发送至需要使用同一类型的较新版本的应用程序。

下面的示例显示的是 `Address` 类的 2.0 版，其 `CountryField` 字段标记为可选。如果较旧版本的应用程序将版本 1 发送至需要使用 2.0 版的较新版本的应用程序，则会忽略数据缺失。

```
[Serializable]
public class Address
{
    public string Street;
    public string City;

    [OptionalField]
    public string CountryField;
}
```

```
<Serializable> _
Public Class Address
    Public Street As String
    Public City As String

    <OptionalField> _
    Public CountryField As String
End Class
```

序列化回调

序列化回调是一种机制，它在序列化/反序列化过程中的四个点提供挂钩。

[illegible]	[illegible]	[illegible]
OnDeserializingAttribute	反序列化之前。*	初始化可选字段的默认值。
OnDeserializedAttribute	反序列化之后。	根据其他字段的内容修改可选字段值。
OnSerializingAttribute	序列化之前。	准备序列化。例如，创建可选数据结构。
OnSerializedAttribute	序列化之后。	记录序列化事件。

* 如果有反序列化构造函数，则在该构造函数之前调用此回调。

使用回调

要使用回调，请将相应的特性应用于接受 StreamingContext 参数的方法。每个类只有一个方法可以用其中每个特性进行标记。例如：

```
[OnDeserializing]
private void SetCountryRegionDefault(StreamingContext sc)
{
    CountryField = "Japan";
}
```

```
<OnDeserializing>
Private Sub SetCountryRegionDefault(sc As StreamingContext)
    CountryField = "Japan"
End Sub
```

这些方法旨在用于版本管理。在反序列化期间，如果可选字段缺少数据，则可能无法正确初始化该字段。若要更正这一情况，可以创建分配正确值的方法，然后将 OnDeserializingAttribute 或 OnDeserializedAttribute 特性应用于该方法。

下面的示例演示类型上下文中的方法。如果应用程序的较旧版本将 `Address` 类的实例发送至该应用程序的较新版本，将会丢失 `CountryField` 字段数据。但是反序列化之后，会将字段设置为默认值“Japan”。

```
[Serializable]
public class Address
{
    public string Street;
    public string City;
    [OptionalField]
    public string CountryField;

    [OnDeserializing]
    private void SetCountryRegionDefault(StreamingContext sc)
    {
        CountryField = "Japan";
    }
}
```

```
<Serializable> _
Public Class Address
    Public Street As String
    Public City As String
    <OptionalField> _
    Public CountryField As String

    <OnDeserializing> _
    Private Sub SetCountryRegionDefault(sc As StreamingContext)
        CountryField = "Japan"
    End Sub
End Class
```

VersionAdded 属性

OptionalFieldAttribute 具有 VersionAdded 属性。该属性指示向给定字段添加了类型的哪个版本。每次修改类型时，版本应该正好增加一（从 2 开始），如下例所示：

```
// Version 1.0
[Serializable]
public class Person
{
    public string FullName;
}

// Version 2.0
[Serializable]
public class Person
{
    public string FullName;

    [OptionalField(VersionAdded = 2)]
    public string NickName;
    [OptionalField(VersionAdded = 2)]
    public DateTime BirthDate;
}

// Version 3.0
[Serializable]
public class Person
{
    public string FullName;

    [OptionalField(VersionAdded=2)]
    public string NickName;
    [OptionalField(VersionAdded=2)]
    public DateTime BirthDate;

    [OptionalField(VersionAdded=3)]
    public int Weight;
}
```

```
' Version 1.0
<Serializable> _
Public Class Person
    Public FullName
End Class

' Version 2.0
<Serializable> _
Public Class Person
    Public FullName As String

    <OptionalField(VersionAdded := 2)> _
    Public NickName As String
    <OptionalField(VersionAdded := 2)> _
    Public BirthDate As DateTime
End Class

' Version 3.0
<Serializable> _
Public Class Person
    Public FullName As String

    <OptionalField(VersionAdded := 2)> _
    Public NickName As String
    <OptionalField(VersionAdded := 2)> _
    Public BirthDate As DateTime

    <OptionalField(VersionAdded := 3)> _
    Public Weight As Integer
End Class
```

SerializationBinder

由于服务器和客户端要求使用不同的类版本，因此，有些用户可能需要控制要序列化和反序列化哪些类。 SerializationBinder 是抽象类，用于控制在序列化和反序列化期间使用的实际类型。若要使用此类，请从 SerializationBinder 派生类，并重写 BindToName 和 BindToType 方法。有关详细信息，请参阅使用 SerializationBinder 控制序列化和反序列化。

最佳实践

要确保版本管理行为正确，修改类型版本时请遵循以下规则：

- 切勿移除已序列化的字段。
- 如果未在以前版本中将 NonSerializedAttribute 特性应用于某个字段，则切勿将该特性应用于该字段。
- 切勿更改已序列化字段的名称或类型。
- 添加新的已序列化字段时，请应用 OptionalFieldAttribute 特性。
- 从字段（在以前版本中不可序列化）中移除 NonSerializedAttribute 特性时，请应用 OptionalFieldAttribute 特性。
- 对于所有可选字段，除非可接受 0 或 null 作为默认值，否则请使用序列化回调设置有意义的默认值。

要确保类型与将来的序列化引擎兼容，请遵循以下准则：

- 始终正确设置 OptionalFieldAttribute 特性上的 VersionAdded 属性。
- 避免版本管理分支。

请参阅

- SerializableAttribute

- BinaryFormatter
- SoapFormatter
- VersionAdded
- OptionalFieldAttribute
- OnDeserializingAttribute
- OnDeserializedAttribute
- OnSerializingAttribute
- OnSerializedAttribute
- StreamingContext
- NonSerializedAttribute
- 二进制序列化

序列化准则

2021/11/16 •

本文列出了在设计要序列化的 API 时要考虑的准则。

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

.NET 提供了针对各种序列化方案进行优化的三种主要序列化技术。下表列出了这些技术以及与这些技术相关的主要 .NET 类型。

数据协定序列化	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	常规持久性 Web 服务 JSON
XML 序列化	XmlSerializer	具有完全控制的 XML 格式
运行时 -序列化（二进制和 SOAP）	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	.NET 远程处理

设计新的类型时，应决定这些类型需要支持哪种技术（如果有的话）。以下准则介绍了如何做出此决定以及如何提供这类支持。这些准则并不表示帮助您选择在实现应用程序或库时应使用哪种序列化技术。这些准则与 API 设计不是直接相关的，因此它们不在本主题的讨论范围内。

准则

- 设计新类型时请务必考虑序列化。

  对于任何类型来说，序列化都是一个重要的设计考虑事项，因为程序可能需要持久保持或传输类型的实例。

选择要支持的合适的序列化技术

任何给定类型均可支持一种或多种序列化技术，或者不支持任何序列化技术。

- 如果可能需要在 Web 服务中持久保持或使用类型的实例，则应考虑支持数据协定序列化。
- 如果需要对序列化类型时生成的 XML 格式具有更多控制，则应考虑改为支持 XML 序列化，或者在支持数

据协定序列化之外还支持 XML 序列化。

这对于某些需要使用数据协定序列化所不支持的 XML 构造(例如，用于生成 XML 特性)的互操作性方案可能是必需的。

- 如果类型实例需要越过 .NET 远程处理边界传递，则应考虑支持运行时序列化。
- 应避免仅仅因为常规持久性原因而支持运行时序列化或 XML 序列化，而应首选数据协定序列化。

数据协定序列化

通过将 DataContractAttribute 应用到类型，并将 DataMemberAttribute 应用到类型的成员(字段和属性)，类型可支持数据协定序列化。

```
[DataContract]
class Person
{

    [DataMember]
    string LastName { get; set; }
    [DataMember]
    string FirstName { get; set; }

    public Person(string firstNameValue, string lastNameValue)
    {
        FirstName = firstNameValue;
        LastName = lastNameValue;
    }
}
```

```
<DataContract()> Public Class Person
    <DataMember()> Public Property LastName As String
    <DataMember()> Public Property FirstName As String

    Public Sub New(ByVal firstNameValue As String, ByVal lastNameValue As String)
        FirstName = firstNameValue
        LastName = lastNameValue
    End Sub

End Class
```

1. 如果可以在部分信任模式下使用类型，则考虑将类型的数据成员标记为公共的。在完全信任模式下，数据协定序列化程序可对非公共类型和成员进行序列化和反序列化，但在部分信任模式下，仅可对公共成员进行序列化和反序列化。

2. 请务必在包含 Data-MemberAttribute 的所有属性上实现 getter 和 setter。对于考虑可进行序列化的类型，数据协定序列化程序要求同时具备 getter 和 setter。如果不会在部分信任模式下使用类型，则这两个属性访问器中的一个或两个都可以是非公共的。

```
[DataContract]
class Person2
{

    string lastName;
    string firstName;

    public Person2(string firstName, string lastName)
    {
        this.lastName = lastName;
        this.firstName = firstName;
    }

    [DataMember]
    public string LastName
    {
        // Implement get and set.
        get { return lastName; }
        private set { lastName = value; }
    }

    [DataMember]
    public string FirstName
    {
        // Implement get and set.
        get { return firstName; }
        private set { firstName = value; }
    }
}
```

```
<DataContract()> Class Person2
    Private lastNameValue As String
    Private firstNameValue As String

    Public Sub New(ByVal firstName As String, ByVal lastName As String)
        Me.lastNameValue = lastName
        Me.firstNameValue = firstName
    End Sub

    <DataMember()> Property LastName As String
        Get
            Return lastNameValue
        End Get

        Set(ByVal value As String)
            lastNameValue = value
        End Set

    End Property

    <DataMember()> Property FirstName As String
        Get
            Return firstNameValue

        End Get
        Set(ByVal value As String)
            firstNameValue = value
        End Set
    End Property

End Class
```

3. 对于反序列化实例的初始化，应考虑使用序列化回调。

   反序列化对象时，不调用任何构造函数。因此，在正常构造期间执行的任何逻辑都需要作为一个序列化回

调实现。

```
[DataContract]
class Person3
{
    [DataMember]
    string lastName;
    [DataMember]
    string firstName;
    string fullName;

    public Person3(string firstName, string lastName)
    {
        // This constructor is not called during deserialization.
        this.lastName = lastName;
        this.firstName = firstName;
        fullName = firstName + " " + lastName;
    }

    public string FullName
    {
        get { return fullName; }
    }

    // This method is called after the object
    // is completely deserialized. Use it instead of the
    // constructror.
    [OnDeserialized]
    void OnDeserialized(StreamingContext context)
    {
        fullName = firstName + " " + lastName;
    }
}
```

```
<DataContract()> _
Class Person3
    <DataMember()> Private lastNameValue As String
    <DataMember()> Private firstNameValue As String
    Dim fullNameValue As String

    Public Sub New(ByVal firstName As String, ByVal lastName As String)
        lastNameValue = lastName
        firstNameValue = firstName
        fullNameValue = firstName & " " & lastName
    End Sub

    Public ReadOnly Property FullName As String
        Get
            Return fullNameValue
        End Get
    End Property

    <OnDeserialized()> Sub OnDeserialized(ByVal context As StreamingContext)
        fullNameValue = firstNameValue & " " & lastNameValue
    End Sub
End Class
```

OnDeserializedAttribute 属性是最常用的回调属性。此系列中的其他属性还包括 OnDeserializingAttribute、OnSerializingAttribute 和 OnSerializedAttribute。这些属性可分别用来标记在反序列化之前、序列化之前以及序列化之后执行的回调。

4. 请考虑使用 KnownTypeAttribute 指示在反序列化复杂对象关系图时应使用的具体类型。

   例如，如果使用一个抽象类表示反序列化数据成员的类型，则序列化程序将需要已知类型信息，以便决定

要实例化并分配给成员的具体类型。如果未使用相关属性指定已知类型，则需要将已知类型显式传递给序列化程序（可通过将已知类型传递给序列化程序构造函数来执行此操作），或者需要在配置文件中指定已知类型。

```
// The KnownTypeAttribute specifies types to be
// used during serialization.
[KnownType(typeof(USAddress))]
[DataContract]
class Person4
{

    [DataMember]
    string fullNameValue;
    [DataMember]
    Address address; // Address is abstract

    public Person4(string fullName, Address address)
    {
        this.fullNameValue = fullName;
        this.address = address;
    }

    public string FullName
    {
        get { return fullNameValue; }
    }
}

[DataContract]
public abstract class Address
{
    public abstract string FullAddress { get; }
}

[DataContract]
public class USAddress : Address
{

    [DataMember]
    public string Street { get; set; }
    [DataMember]
    public string City { get; set; }
    [DataMember]
    public string State { get; set; }
    [DataMember]
    public string ZipCode { get; set; }

    public override string FullAddress
    {
        get
        {
            return Street + "\n" + City + ", " + State + " " + ZipCode;
        }
    }
}
```

```
<KnownType(GetType(USAddress)), _
DataContract()> Class Person4
    <DataMember()> Property fullNameValue As String
    <DataMember()> Property addressValue As USAddress ' Address is abstract

    Public Sub New(ByVal fullName As String, ByVal address As Address)
        fullNameValue = fullName
        addressValue = address
    End Sub

    Public ReadOnly Property FullName() As String
        Get
            Return fullNameValue
        End Get

    End Property
End Class

<DataContract()> Public MustInherit Class Address
    Public MustOverride Function FullAddress() As String
End Class

<DataContract()> Public Class USAddress
    Inherits Address
    <DataMember()> Public Property Street As String
    <DataMember()> Public City As String
    <DataMember()> Public State As String
    <DataMember()> Public ZipCode As String

    Public Overrides Function FullAddress() As String
        Return Street & "\n" & City & ", " & State & " " & ZipCode
    End Function
End Class
```

在无法通过静态方式了解已知类型列表的情况下（当编译 Person 类时），KnownTypeAttribute 还可指向一个在运行时返回已知类型列表的方法 。

5. 创建或更改可序列化的类型时，请务必考虑向后兼容性和向前兼容性。

   请记住，类型的未来版本的序列化流可反序列化到类型的当前版本，反之亦然。您一定要清楚，数据成员（甚至对于私有成员和内部成员）不能更改其名称和类型，甚至不能更改其在类型的未来版本中的顺序，除非特别留意将使用显式参数的协定保存到数据协定属性。在对可序列化的类型进行更改时，测试序列化的兼容性。尝试将新版本反序列化为旧版本，以及将旧版本反序列化为新版本。

6. 考虑实现 IExtensibleDataObject 接口以允许在类型的两个不同版本之间进行往返。

   序列化程序可通过此接口确保在往返期间不丢失任何数据。ExtensionData 属性可存储类型的未来版本中不为当前版本所知的任何数据。在随后序列化当前版本并将其反序列化为未来版本时，可通过 ExtensionData 属性值在序列化流中使用附加数据。

```
// Implement the IExtensibleDataObject interface.
[DataContract]
class Person5 : IExtensibleDataObject
{
    ExtensionDataObject serializationData;
    [DataMember]
    string fullNameValue;

    public Person5(string fullName)
    {
        this.fullNameValue = fullName;
    }

    public string FullName
    {
        get { return fullNameValue; }
    }

    ExtensionDataObject IExtensibleDataObject.ExtensionData
    {
        get
        {
            return serializationData;
        }
        set { serializationData = value; }
    }
}
```

```
<DataContract()> Class Person5
    Implements IExtensibleDataObject
    <DataMember()> Dim fullNameValue As String

    Public Sub New(ByVal fullName As String)
        fullName = fullName
    End Sub

    Public ReadOnly Property FullName
        Get
            Return fullNameValue
        End Get
    End Property
    Private serializationData As ExtensionDataObject
    Public Property ExtensionData As ExtensionDataObject Implements
IExtensibleDataObject.ExtensionData
        Get
            Return serializationData
        End Get
        Set(ByVal value As ExtensionDataObject)
            serializationData = value
        End Set
    End Property
End Class
```

有关详细信息，请参阅向前兼容的数据协定。

XML 序列化

数据协定序列化是 .NET Framework 中的主要(默认)序列化技术，但也存在数据协定序列化不支持的序列化情况。例如，数据协定序列化无法让您完全控制序列化程序生成或使用的 XML 的形状。如果要求此类精细控制，则必须使用 XML 序列化，而且需要设计类型以支持此序列化技术。

1. 应避免专门针对 XML 序列化设计类型，除非您有很充分的理由要控制所生成的 XML 的形状。此序列化技术已由上一节讨论的数据协定序列化所取代。

换句话说，不要将 System.Xml.Serialization 命名空间中的属性应用到新类型，除非你清楚该类型将会用于 XML 序列化。以下示例展示如何使用 System.Xml.Serialization 来控制生成的 XML 的形状。

```
public class Address2
{
    [System.Xml.Serialization.XmlAttribute] // Serialize as XML attribute, instead of an element.
    public string Name { get { return "Poe, Toni"; } set { } }
    [System.Xml.Serialization.XmlElement(ElementName = "StreetLine")] // Explicitly name the element.
    public string Street = "1 Main Street";
}
```

```
Public Class Address2
    ' Supports XML Serialization.
    <System.Xml.Serialization.XmlAttribute()> _
    Public ReadOnly Property Name As String ' Serialize as XML attribute, instead of an element.
        Get
            Return "Poe, Toni"
        End Get
    End Property
    <System.Xml.Serialization.XmlElement(ElementName:="StreetLine")> _
    Public Street As String = "1 Main Street"  ' Explicitly names the element 'StreetLine'.
End Class
```

2. 如果通过应用 XML 序列化属性所提供的对于序列化 XML 的形状的控制还无法满足你的需要，则可考虑实现 IXmlSerializable 接口。利用此接口的两种方法（ReadXml 和 WriteXml），你可以完全控制序列化的 XML 流。还可以通过应用 XmlSchemaProviderAttribute 属性来控制为类型生成的 XML 架构。

运行时序列化

运行时序列化是 .NET 远程处理所使用的一项技术。如果你认为将会使用 .NET 远程处理传输类型，则确保类型支持运行时序列化。

可通过应用 SerializableAttribute 属性提供对运行时序列化的基本支持，更高级的方案涉及实现一个简单的运行时可序列化模式（实现 -ISerializable 并提供一个序列化构造函数）。

1. 如果您的类型将要用于 .NET 远程处理，则应考虑支持运行时序列化。例如，System.AddIn 命名空间使用 .NET 远程处理，因此在 System.AddIn 加载项之间交换的所有类型都需要支持运行时序列化。

```
// Apply SerializableAttribute to support runtime serialization.
[Serializable]
public class Person6
{
    // Code not shown.
}
```

```
<Serializable()> Public Class Person6 ' Support runtime serialization with the SerializableAttribute.

    ' Code not shown.
End Class
```

2. 如果需要完全控制序列化过程，则应考虑实现运行时可序列化模式。例如，如果您需要在对数据进行序列化或反序列化时转换数据。

   此模式非常简单。您需要执行的全部操作就是实现 ISerializable 接口，并提供在反序列化对象时使用的特殊构造函数。

```
// Implement the ISerializable interface for more control.
[Serializable]
public class Person_Runtime_Serializable : ISerializable
{
    string fullName;

    public Person_Runtime_Serializable() { }
    protected Person_Runtime_Serializable(SerializationInfo info, StreamingContext context){
        if (info == null) throw new System.ArgumentNullException("info");
        fullName = (string)info.GetValue("name", typeof(string));
    }
    [SecurityPermission(SecurityAction.LinkDemand,
    Flags = SecurityPermissionFlag.SerializationFormatter)]
    void ISerializable.GetObjectData(SerializationInfo info,
            StreamingContext context) {
        if (info == null) throw new System.ArgumentNullException("info");
        info.AddValue("name", fullName);
    }

    public string FullName
    {
        get { return fullName; }
        set { fullName = value; }
    }
}
```

```
' Implement the ISerializable interface for more control.
<Serializable()> Public Class Person_Runtime_Serializable
    Implements ISerializable

    Private fullNameValue As String

    Public Sub New()
        ' empty constructor.
    End Sub
    Protected Sub New(ByVal info As SerializationInfo, _
                      ByVal context As StreamingContext)
        If info Is Nothing Then
            Throw New System.ArgumentNullException("info")
            FullName = CType(info.GetValue("name", GetType(String)), String)
        End If
    End Sub

    Private Sub GetObjectData(ByVal info As SerializationInfo, _
                              ByVal context As StreamingContext) _
                          Implements ISerializable.GetObjectData
        If info Is Nothing Then
            Throw New System.ArgumentNullException("info")
        End If
        info.AddValue("name", FullName)
    End Sub

    Public Property FullName As String

        Get
            Return fullNameValue
        End Get
        Set(ByVal value As String)
            fullNameValue = value

        End Set
    End Property

End Class
```

3. 请务必保护序列化构造函数，并提供完全按照此处示例中的显示类型化和命名的两个参数。

```
protected Person_Runtime_Serializable(SerializationInfo info, StreamingContext context){
```

```
Protected Sub New(ByVal info As SerializationInfo, _
                  ByVal context As StreamingContext)
```

4. 请务必显式实现 ISerializable 成员。

```
void ISerializable.GetObjectData(SerializationInfo info,
        StreamingContext context) {
```

```
Private Sub GetObjectData(ByVal info As SerializationInfo, _
                          ByVal context As StreamingContext) _
                      Implements ISerializable.GetObjectData
```

5. 请务必向 ISerializable.GetObjectData 实现应用链接要求。这将确保只有完全受信任的核心和运行时序列化程序才有权访问成员。

```
[SecurityPermission(SecurityAction.LinkDemand,
Flags = SecurityPermissionFlag.SerializationFormatter)]
```

```
<Serializable()> Public Class Person_Runtime_Serializable2
    Implements ISerializable
    <SecurityPermission(SecurityAction.LinkDemand,
Flags:=SecurityPermissionFlag.SerializationFormatter)> _
    Private Sub GetObjectData(ByVal info As System.Runtime.Serialization.SerializationInfo, _
                              ByVal context As System.Runtime.Serialization.StreamingContext) _
                          Implements System.Runtime.Serialization.ISerializable.GetObjectData
        ' Code not shown.
    End Sub
End Class
```

请参阅

- 使用数据协定
- 数据协定序列化程序
- 数据协定序列化程序支持的类型
- 二进制序列化
- .NET 远程处理
- XML 和 SOAP 序列化
- 安全性和序列化

如何：对序列化数据进行分块

2021/11/16 •

> **WARNING**
> 二进制序列化可能会十分危险。有关详细信息，请参阅BinaryFormatter security guide。

在 Web 服务消息中发送大型数据集时，会出现下面两个问题：

1. 因序列化引擎执行缓冲而使工作集（内存）很大。
2. 因使用 Base64 编码后产生 33% 的膨胀而过度占用带宽。

若要解决这两个问题，可以通过实现 IXmlSerializable 接口，来控制序列化和反序列化。具体来讲，就是实现 WriteXml 和 ReadXml 方法，对数据进行分块。

实现服务器端数据分块

1. 在服务器计算机上，必须采用 Web 方法禁用 ASP.NET 缓冲，并返回实现 IXmlSerializable 的类型。
2. 实现 IXmlSerializable 的类型可以采用 WriteXml 方法对数据进行分块。

实现客户端处理

1. 通过更改客户端代理上的 Web 方法，可以返回实现 IXmlSerializable 的类型。可以使用 SchemaImporterExtension 自动执行这项操作，但此处不加以说明。
2. 通过实现 ReadXml 方法，可以读取分块数据流，并将字节写入磁盘。此实现还会引发可供图形控件（如进度栏）使用的进度事件。

示例

下面的代码示例演示客户端上用于禁用 ASP.NET 缓冲的 Web 方法。它还演示了如何在客户端实现 IXmlSerializable 接口，该实现可以采用 WriteXml 方法对数据进行分块。

```
[WebMethod]
[System.Web.Services.Protocols.SoapDocumentMethodAttribute
(ParameterStyle= SoapParameterStyle.Bare)]
public SongStream DownloadSong(DownloadAuthorization Authorization, string filePath)
{

// Turn off response buffering.
System.Web.HttpContext.Current.Response.Buffer = false;
// Return a song.
SongStream song = new SongStream(filePath);
return song;
}
```

```
    <WebMethod(),
System.Web.Services.Protocols.SoapDocumentMethodAttribute(ParameterStyle:=SoapParameterStyle.Bare)> _
    Public Function DownloadSong(ByVal Authorization As DownloadAuthorization, ByVal filePath As String) As
SongStream

        ' Turn off response buffering.
        System.Web.HttpContext.Current.Response.Buffer = False
        ' Return a song.
        Dim song As New SongStream(filePath)
        Return song

    End Function
End Class
```

```
[XmlSchemaProvider("MySchema")]
public class SongStream : IXmlSerializable
```

```
{

    private const string ns = "http://demos.Contoso.com/webservices";
    private string filePath;

    public SongStream(){ }

    public SongStream(string filePath)
    {
     this.filePath = filePath;
    }

    // This is the method named by the XmlSchemaProviderAttribute applied to the type.
    public static XmlQualifiedName MySchema(XmlSchemaSet xs)
    {
     // This method is called by the framework to get the schema for this type.
     // We return an existing schema from disk.

     XmlSerializer schemaSerializer = new XmlSerializer(typeof(XmlSchema));
     string xsdPath = null;
     // NOTE: replace the string with your own path.
     xsdPath = System.Web.HttpContext.Current.Server.MapPath("SongStream.xsd");
     XmlSchema s = (XmlSchema)schemaSerializer.Deserialize(
         new XmlTextReader(xsdPath), null);
     xs.XmlResolver = new XmlUrlResolver();
     xs.Add(s);

     return new XmlQualifiedName("songStream", ns);
    }

    void IXmlSerializable.WriteXml(System.Xml.XmlWriter writer)
    {
       // This is the chunking code.
       // ASP.NET buffering must be turned off for this to work.

     int bufferSize = 4096;
     char[] songBytes = new char[bufferSize];
     FileStream inFile = File.Open(this.filePath, FileMode.Open, FileAccess.Read);

     long length = inFile.Length;

     // Write the file name.
     writer.WriteElementString("fileName", ns, Path.GetFileNameWithoutExtension(this.filePath));

     // Write the size.
     writer.WriteElementString("size", ns, length.ToString());

     // Write the song bytes.
     writer.WriteStartElement("song", ns);

     StreamReader sr = new StreamReader(inFile, true);
     int readLen = sr.Read(songBytes, 0, bufferSize);

     while (readLen > 0)
     {
         writer.WriteStartElement("chunk", ns);
         writer.WriteChars(songBytes, 0, readLen);
         writer.WriteEndElement();

         writer.Flush();
         readLen = sr.Read(songBytes, 0, bufferSize);
     }

     writer.WriteEndElement();
     inFile.Close();
    }

    System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
    {
     throw new System.NotImplementedException();
    }

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
     throw new System.NotImplementedException();
    }
}
```

```
<XmlSchemaProvider("MySchema")> _
Public Class SongStream
    Implements IXmlSerializable

    Private Const ns As String = "http://demos.Contoso.com/webservices"
    Private filePath As String

    Public Sub New()

    End Sub

    Public Sub New(ByVal filePath As String)
        Me.filePath = filePath
    End Sub


    ' This is the method named by the XmlSchemaProviderAttribute applied to the type.
    Public Shared Function MySchema(ByVal xs As XmlSchemaSet) As XmlQualifiedName
        ' This method is called by the framework to get the schema for this type.
        ' We return an existing schema from disk.
        Dim schemaSerializer As New XmlSerializer(GetType(XmlSchema))
        Dim xsdPath As String = Nothing
        ' NOTE: replace SongStream.xsd with your own schema file.
        xsdPath = System.Web.HttpContext.Current.Server.MapPath("SongStream.xsd")
        Dim s As XmlSchema = CType(schemaSerializer.Deserialize(New XmlTextReader(xsdPath)), XmlSchema)
        xs.XmlResolver = New XmlUrlResolver()
        xs.Add(s)

        Return New XmlQualifiedName("songStream", ns)

    End Function



    Sub WriteXml(ByVal writer As System.Xml.XmlWriter) Implements IXmlSerializable.WriteXml
        ' This is the chunking code.
        ' ASP.NET buffering must be turned off for this to work.

        Dim bufferSize As Integer = 4096
        Dim songBytes(bufferSize) As Char
        Dim inFile As FileStream = File.Open(Me.filePath, FileMode.Open, FileAccess.Read)

        Dim length As Long = inFile.Length

        ' Write the file name.
        writer.WriteElementString("fileName", ns, Path.GetFileNameWithoutExtension(Me.filePath))

        ' Write the size.
        writer.WriteElementString("size", ns, length.ToString())

        ' Write the song bytes.
        writer.WriteStartElement("song", ns)

        Dim sr As New StreamReader(inFile, True)
        Dim readLen As Integer = sr.Read(songBytes, 0, bufferSize)

        While readLen > 0
            writer.WriteStartElement("chunk", ns)
            writer.WriteChars(songBytes, 0, readLen)
            writer.WriteEndElement()

            writer.Flush()
            readLen = sr.Read(songBytes, 0, bufferSize)
        End While

        writer.WriteEndElement()
        inFile.Close()
    End Sub

    Function GetSchema() As System.Xml.Schema.XmlSchema Implements IXmlSerializable.GetSchema
        Throw New System.NotImplementedException()
    End Function

    Sub ReadXml(ByVal reader As System.Xml.XmlReader) Implements IXmlSerializable.ReadXml
        Throw New System.NotImplementedException()
    End Sub
```

```
End Class
```

```
public class SongFile : IXmlSerializable
{
    public static event ProgressMade OnProgress;

    public SongFile()
    { }

    private const string ns = "http://demos.teched2004.com/webservices";
    public static string MusicPath;
    private string filePath;
    private double size;

    void IXmlSerializable.ReadXml(System.Xml.XmlReader reader)
    {
        reader.ReadStartElement("DownloadSongResult", ns);
        ReadFileName(reader);
        ReadSongSize(reader);
        ReadAndSaveSong(reader);
        reader.ReadEndElement();
    }

    void ReadFileName(XmlReader reader)
    {
        string fileName = reader.ReadElementString("fileName", ns);
        this.filePath =
            Path.Combine(MusicPath, Path.ChangeExtension(fileName, ".mp3"));
    }

    void ReadSongSize(XmlReader reader)
    {
        this.size = Convert.ToDouble(reader.ReadElementString("size", ns));
    }

    void ReadAndSaveSong(XmlReader reader)
    {
        FileStream outFile = File.Open(
            this.filePath, FileMode.Create, FileAccess.Write);

        string songBase64;
        byte[] songBytes;
        reader.ReadStartElement("song", ns);
        double totalRead=0;
        while(true)
        {
            if (reader.IsStartElement("chunk", ns))
            {
                  songBase64 = reader.ReadElementString();
                  totalRead += songBase64.Length;
                  songBytes = Convert.FromBase64String(songBase64);
                  outFile.Write(songBytes, 0, songBytes.Length);
                  outFile.Flush();

                  if (OnProgress != null)
                  {
                      OnProgress(100 * (totalRead / size));
                  }
            }

            else
            {
                  break;
            }
        }

        outFile.Close();
        reader.ReadEndElement();
    }

    [PermissionSet(SecurityAction.Demand, Name="FullTrust")]
    public void Play()
    {
        System.Diagnostics.Process.Start(this.filePath);
    }
```

```
    }

    System.Xml.Schema.XmlSchema IXmlSerializable.GetSchema()
    {
        throw new System.NotImplementedException();
    }

    public void WriteXml(XmlWriter writer)
    {
        throw new System.NotImplementedException();
    }
}
```

```
Public Class SongFile
    Implements IXmlSerializable
    Public Shared Event OnProgress As ProgressMade


    Public Sub New()

    End Sub

    Private Const ns As String = "http://demos.teched2004.com/webservices"
    Public Shared MusicPath As String
    Private filePath As String
    Private size As Double

    Sub ReadXml(ByVal reader As System.Xml.XmlReader) Implements IXmlSerializable.ReadXml
        reader.ReadStartElement("DownloadSongResult", ns)
        ReadFileName(reader)
        ReadSongSize(reader)
        ReadAndSaveSong(reader)
        reader.ReadEndElement()
    End Sub

    Sub ReadFileName(ByVal reader As XmlReader)
        Dim fileName As String = reader.ReadElementString("fileName", ns)
        Me.filePath = Path.Combine(MusicPath, Path.ChangeExtension(fileName, ".mp3"))

    End Sub

    Sub ReadSongSize(ByVal reader As XmlReader)
        Me.size = Convert.ToDouble(reader.ReadElementString("size", ns))

    End Sub

    Sub ReadAndSaveSong(ByVal reader As XmlReader)
        Dim outFile As FileStream = File.Open(Me.filePath, FileMode.Create, FileAccess.Write)

        Dim songBase64 As String
        Dim songBytes() As Byte
        reader.ReadStartElement("song", ns)
        Dim totalRead As Double = 0
        While True
            If reader.IsStartElement("chunk", ns) Then
                songBase64 = reader.ReadElementString()
                totalRead += songBase64.Length
                songBytes = Convert.FromBase64String(songBase64)
                outFile.Write(songBytes, 0, songBytes.Length)
                outFile.Flush()
                RaiseEvent OnProgress((100 * (totalRead / size)))
            Else
                Exit While
            End If
        End While

        outFile.Close()
        reader.ReadEndElement()
    End Sub

    <PermissionSet(SecurityAction.Demand, Name:="FullTrust")> _
    Public Sub Play()
        System.Diagnostics.Process.Start(Me.filePath)
    End Sub


    Function GetSchema() As System.Xml.Schema.XmlSchema Implements IXmlSerializable.GetSchema
        Throw New System.NotImplementedException()
    End Function


    Public Sub WriteXml(ByVal writer As XmlWriter) Implements IXmlSerializable.WriteXml
        Throw New System.NotImplementedException()
    End Sub
End Class
```

编译代码

- 代码使用下面的命名空间：System、System.Runtime.Serialization、System.Web.Services、System.Web.Services.Protocols、System.Xml、System.Xml.Seri和 System.Xml.Schema。

请参阅

- 自定义序列化

如何确定 .NET Standard 对象是否可序列化

2021/11/16 •

.NET Standard 是一种规范，用于定义符合该标准版本的特定 .NET 实现上必须存在的类型和成员。但 .NET Standard 未定义某个类型是否可序列化。.NET Standard 库中定义的类型未使用 SerializableAttribute 属性进行标记。相反，特定 .NET 实现（如 .NET Framework 和 .NET Core）可随意确定特定类型是否可序列化。

如果你开发了面向 .NET Standard 的库，则支持 .NET Standard 的任何 .NET 实现都可以使用你的库。这意味着你无法提前知道特定类型是否可序列化；只能在运行时确定它是否可序列化。

可以通过检索表示对象类型的 Type 对象的 IsSerializable 属性值，在运行时确定对象是否可序列化。以下示例提供了一个实现。它定义一个 `IsSerializable(Object)` 扩展方法，用于指示任何 Object 实例是否可以序列化。

```
namespace Libraries
{
    using System;

    public static class UtilityLibrary
    {
        public static bool IsSerializable(this object obj)
        {
            if (obj == null)
                return false;

            Type t = obj.GetType();
            return t.IsSerializable;
        }
    }
}
```

```
Imports System.Runtime.CompilerServices

Namespace Global.Libraries

    Public Module UtilityLibrary
        <Extension>
        Public Function IsSerializable(obj As Object) As Boolean
            If obj Is Nothing Then Return False

            Dim t As Type = obj.GetType()
            Return t.IsSerializable
        End Function
    End Module
End Namespace
```

随后可以将任何对象传递给该方法，以确定它是否可以在当前 .NET 实现上进行序列化和反序列化，如以下示例所示：

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using Libraries;

namespace test_serialization
{
   class Program
   {
      static void Main()
      {
         var value = ValueTuple.Create("03244562", DateTime.Now, 13452.50m);
         if (value.IsSerializable())
         {
            // Serialize the value tuple.
            var formatter = new BinaryFormatter();
            using (var stream = new FileStream("data.bin", FileMode.Create,
                                    FileAccess.Write, FileShare.None))
            {
               formatter.Serialize(stream, value);
            }
            // Deserialize the value tuple.
            using (var readStream = new FileStream("data.bin", FileMode.Open))
            {
               object restoredValue = formatter.Deserialize(readStream);
               Console.WriteLine($"{restoredValue.GetType().Name}: {restoredValue}");
            }
         }
         else
         {
            Console.WriteLine($"{nameof(value)} is not serializable");
         }
      }
   }
}
// The example displays output like the following:
//    ValueTuple`3: (03244562, 10/18/2017 5:25:22 PM, 13452.50)
```

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
Imports Libraries

Module Program
    Sub Main()
        Dim value = ValueTuple.Create("03244562", DateTime.Now, 13452.50d)
        If value.IsSerializable() Then
            Dim formatter As New BinaryFormatter()
            ' Serialize the value tuple.
            Using stream As New FileStream("data.bin", FileMode.Create,
                                           FileAccess.Write, FileShare.None)
                formatter.Serialize(stream, value)
            End Using
            ' Deserialize the value tuple.
            Using readStream As New FileStream("data.bin", FileMode.Open)
                Dim restoredValue = formatter.Deserialize(readStream)
                Console.WriteLine($"{restoredValue.GetType().Name}: {restoredValue}")
            End Using
        Else
            Console.WriteLine($"{nameof(value)} is not serializable")
        End If
    End Sub
End Module
' The example displays output like the following:
'    ValueTuple`3: (03244562, 10/18/2017 5:25:22 PM, 13452.50)
```

请参阅

- 二进制序列化
- System.SerializableAttribute
- Type.IsSerializable

XML 和 SOAP 序列化

2021/11/16 •

XML 序列化将对象的公共字段和属性以及方法的参数和返回值转换(序列化)为符合特定 XML 架构定义语言 (XSD) 文档的 XML 流。XML 序列化会生成强类型类，同时将公共属性和字段转换为序列格式(在此情况下为 XML)，以便存储或传输。

由于 XML 是开放式的标准，因此可以根据需要由任何应用程序处理 XML 流，而与平台无关。例如，用 ASP.NET 创建的 XML Web services 使用 XmlSerializer 类来创建 XML 流，这些流在整个 Internet 中或在 Intranet 上的 XML Web services 应用程序之间传递数据。相反，反序列化采用这样一个 XML 流并重新构造对象。

XML 序列化还可用于将对象序列化为符合 SOAP 规范的 XML 流。SOAP 是一种基于 XML 的协议，它是专门为使用 XML 来传输过程调用而设计的。

若要序列化或反序列化对象，请使用 XmlSerializer 类。要创建待序列化的类，请使用 XML 架构定义工具。

请参阅

- 二进制序列化
- 使用 ASP.NET 创建的 XML Web service 以及 XML Web Service 客户端

XML 序列化

2021/11/16 •

序列化是将对象转换成易于传输的形式的过程。例如，可以序列化对象，并使用 HTTP 通过 Internet 在客户端和服务器之间进行传输。另一方面，反序列化在流中重新构建对象。

XML 序列化只将对象的公共字段和属性值序列化为 XML 流。XML 序列化不包括类型信息。例如，如果"Library"命名空间中存在"Book"对象，则不能保证将它反序列化为同一类型的对象 。

> **NOTE**
> XML 序列化不能转换方法、索引器、私有字段或只读属性(只读集合除外)。若要序列化对象的所有公共和私有字段和属性，请使用 DataContractSerializer 而不要使用 XML 序列化。

XML 序列化中的中心类是 XmlSerializer 类，此类中最重要的方法是 Serialize 和 Deserialize 方法 。XmlSerializer 创建 C# 文件并将其编译为 .dll 文件，以执行此序列化。XML 序列化程序生成器工具 (Sgen.exe) 旨在预先生成要与应用程序一起部署的这些序列化程序集，并改进启动性能。XmlSerializer 生成的 XML 流符合万维网联合会 (W3C) XML 架构定义语言 (XSD) 1.0 建议。而且，生成的数据类型符合文档"XML 架构第 2 部分:数据类型"。

对象中的数据是用编程语言构造来描述的，如类、字段、属性、基元类型、数组，甚至是 XmlElement 或 XmlAttribute 对象形式的嵌入 XML 。您可以创建自己的用特性批注的类，也可以使用 XML 架构定义工具生成基于现有 XML 架构的类。

如果有 XML 架构，则可以运行 XML 架构定义工具生成一组类，将这组类的类型强制为此架构，并用特性进行批注。当序列化这种类的实例时，生成的 XML 符合 XML 架构。对于这种类，可以采用易于操作的对象模型进行编程，同时确保生成的 XML 符合 XML 架构。这是使用 .NET 中的其他类(如 XmlReader 和 XmlWriter 类)分析和编写 XML 流的另一种方法 。有关详细信息，请参阅 XML 文档和数据。这些类可让您分析任何 XML 流。相反，如果 XML 流应符合已知的 XML 架构，则使用 XmlSerializer。

特性可控制 XmlSerializer 类生成的 XML 流，使你能够设置 XML 流的 XML 命名空间、元素名称、特性名称等。有关这些特性以及它们如何控制 XML 序列化的详细信息，请参阅使用特性控制 XML 序列化。有关那些用于控制生成的 XML 的特性的表格，请参阅控制 XML 序列化的特性。

XmlSerializer 类可以进一步序列化对象并生成编码的 SOAP XML 流。生成的 XML 符合万维网联合会文档"简单对象访问协议 (SOAP) 1.1"(Simple Object Access Protocol (SOAP) 1.1) 的第 5 节。有关此进程的详细信息，请参阅如何:将对象序列化为 SOAP 编码的 XML 流。有关可控制生成的 XML 的特性的表格，请参阅控制编码的 SOAP 序列化的特性。

XmlSerializer 类生成由 XML Web 服务创建并传递给 XML Web 服务的 SOAP 信息。要控制 SOAP 消息，可以将特性应用于在 XML Web services 文件 (.asmx) 中找到的类、返回值、参数和字段。由于 XML Web services 可以使用文本或编码的 SOAP 样式，因此既可以使用"用来控制 XML 序列化的特性"中列出的特性，也可以使用"用来控制编码的 SOAP 序列化的特性"中列出的特性。有关使用特性来控制由 XML Web 服务生成的 XML 的详细信息，请参阅使用 XML Web 服务进行 XML 序列化。有关 SOAP 和 XML Web 服务的详细信息，请参阅自定义 SOAP 消息格式。

XmlSerializer 应用程序的安全注意事项

创建使用 XmlSerializer 的应用程序时，应了解以下各项及其含义:

- XmlSerializer 在由 TEMP 环境变量命名的目录中创建 C# (.cs) 文件并将它们编译为 dll 文件;这些 DLL 文件将进行序列化。

> **NOTE**
>
> 可预先生成这些序列化程序集，并使用 SGen.exe 工具对它们进行签名。这不适用于 Web 服务的服务器。也就是说，这只适用于客户端用法和手动序列化。

代码和 DLL 在创建和编译时易受恶意进程破坏。当使用的计算机运行 Microsoft Windows NT 4.0 或更高版本时，可能有两个或多个用户共享 TEMP 目录。如果两个帐户的安全权限不同，并且权限较高的帐户使用 XmlSerializer 运行应用程序，则共享 TEMP 目录是危险的操作。在这种情况下，一个用户可能因替换已编译的 .cs 或 .dll 文件而违反计算机的安全性。要消除这一问题，应始终确保计算机上的每个帐户具有各自的配置文件。默认情况下，TEMP 环境变量为每个帐户指向不同的目录。

- 如果恶意用户将连续的 XML 数据流发送至 Web 服务器（拒绝服务攻击），则 XmlSerializer 会继续处理数据，直到计算机资源不足为止。

  如果您使用的计算机运行了 Internet 信息服务 (IIS)，并且您的应用程序在 IIS 内运行，则可消除这种攻击。IIS 提供了一种网关，这种网关不处理长度超过设定量（默认值为 4 KB）的流。如果创建的应用程序不使用 IIS 并且使用 XmlSerializer 进行反序列化，则应该实现防止拒绝服务攻击的类似网关。

- XmlSerializer 使用提供给它的任何类型序列化数据并运行任何代码。

  恶意对象造成威胁的方式有两种。它可以运行恶意代码，也可以将恶意代码插入 XmlSerializer 创建的 C# 文件中。在第一种情况下，如果恶意对象试图运行破坏性的过程，代码访问安全有助于防止发生任何损坏。在第二种情况下，从理论上来说，恶意对象有可能以某种方式将代码插入 XmlSerializer 创建的 C# 文件中。尽管这一问题已得到彻底检查并且认为不可能发生这种攻击，但还是应当采取防范措施，一定不要使用未知和不受信任的类型来序列化数据。

- 已序列化的敏感数据可能容易受到攻击。

  XmlSerializer 序列化数据之后，数据可存储为 XML 文件或其他数据存储。如果数据存储区可供其他进程使用或者可在 Intranet 或 Internet 上看见，则数据可能被盗和恶意使用。例如，如果您创建一个应用程序来序列化包含信用卡号码的订单，则数据是高度敏感的。为防止被盗和恶意使用，应始终保护您的数据存储区并采取步骤保持其私密性。

简单类的序列化

下面的代码示例显示具有公共字段的基类。

```
Public Class OrderForm
    Public OrderDate As DateTime
End Class
```

```
public class OrderForm
{
    public DateTime OrderDate;
}
```

此类的实例序列化后可能如下所示。

```
<OrderForm>
    <OrderDate>12/12/01</OrderDate>
</OrderForm>
```

有关序列化的更多示例，请参阅 XML 序列化示例。

可序列化的项

可使用 XmlSerializer 类对以下各项进行序列化：

- 公共类的公共读/写属性和字段。
- 执行 ICollection 或 IEnumerable 的类 。

> **NOTE**
> 仅序列化集合，不序列化公共属性。

- XmlElement 对象。
- XmlNode 对象。
- DataSet 对象。

有关序列化或反序列化对象的更多信息，请参阅如何：序列化对象和如何：反序列化对象。

使用 XML 序列化的优点

将对象序列化为 XML 时，XmlSerializer 类可提供完整而灵活的控制。如果要创建 XML Web services，可以将控制序列化的特性应用于类和成员，以确保 XML 输出符合特定架构。

例如，XmlSerializer 使你能够：

- 指定字段或属性是否应编码为特性或元素。
- 指定要使用的 XML 命名空间。
- 指定字段或属性名称不恰当时的元素或特性名称。

XML 序列化的另一个优点是，只要生成的 XML 流符合给定的架构，就对开发的应用程序没有任何约束。假设有一个架构，它用于描述书籍。它提供有书名、作者、出版商和 ISBN 号元素。您可以开发一个应用程序来以任何想要的方式（例如以书籍订单或书籍库存方式）处理 XML 数据。在任一种情况下，唯一的要求是 XML 流符合指定的 XML 架构定义语言 (XSD) 架构。

XML 序列化注意事项

使用 XmlSerializer 类时，应注意以下事项：

- Sgen.exe 工具特别设计为生成序列化程序集，以获得最佳性能。
- 序列化数据只包含数据本身和类的结构。类型标识和程序集信息不包括在内。
- 只能序列化公共属性和字段。属性必须具有公共访问器（get 和 set 方法）。如果必须序列化非公共数据，请使用 DataContractSerializer 类而不使用 XML 序列化。
- 类必须具有无参数构造函数才能被 XmlSerializer 序列化。
- 方法不能被序列化。
- 如下所述，如果实现 IEnumerable 或 ICollection 的类满足某些要求，XmlSerializer 则可以处理这些类 。

  实现 IEnumerable 的类必须实现采用单个参数的公共 Add 方法 。Add 方法的参数必须与从“IEnumerator.Current”属性返回的类型一致（多态），该属性是从 GetEnumerator 方法返回的 。

  除了实现 IEnumerable 之外，还能实现 ICollection 的类（如 CollectionBase）必须具有采用整型的公共“Item”索引属性（在 C# 中为索引器），而且它必须有一个“integer”类型的公共“Count”属性 。传递给 Add

方法的参数必须与从“Item”属性返回的类型相同，或者为此类型的基之一 。

对于实现 ICollection 的类，可从已编制索引的“Item”属性检索要序列化的值，而不是通过调用 GetEnumerator 进行检索 。此外，除了返回另一个集合类（实现 ICollection 的一个类）的公共字段外，公共字段和属性不会被序列化。有关示例，请参阅 XML 序列化示例。

XSD 数据类型映射

标题为 XML 架构第 2 部分：数据类型的 W3C 文档指定 XML 架构定义语言 (XSD) 架构中允许的简单数据类型。对于其中的许多数据类型（如“int”和“decimal”），.NET 中都有相应的数据类型 。然而，某些 XML 数据类型没有相应的 .NET 数据类型（如“NMTOKEN”数据类型）。在这种情况下，如果使用 XML 架构定义工具（XML 架构定义工具 (Xsd.exe)）从架构中生成类，相应的特性会应用到字符串类型的成员，其“DataType”属性会设置为 XML 数据类型名称。例如，如果架构包含 XML 数据类型为“NMTOKEN”且名为“MyToken”的元素，则生成的类可能包含下例中所示的成员。

```
<XmlElement(DataType:="NMTOKEN")> _
Public MyToken As String
```

```
[XmlElement(DataType = "NMTOKEN")]
public string MyToken;
```

同样，如果要创建的类必须符合特定的 XML 架构 (XSD)，则应该应用相应的特性并且将其“DataType”属性设置为所需的 XML 数据类型名称。

有关类型映射的完整列表，请参阅以下任一特性类的“DataType”属性：

- SoapAttributeAttribute
- SoapElementAttribute
- XmlArrayItemAttribute
- XmlAttributeAttribute
- XmlElementAttribute
- XmlRootAttribute

请参阅

- XmlSerializer
- DataContractSerializer
- FileStream
- XML 和 SOAP 序列化
- 二进制序列化
- 序列化
- XmlSerializer
- XML 序列化示例
- 如何：序列化对象
- 如何：反序列化对象

XML 序列化示例

2021/11/16 •

XML 序列化可以采用从简单到复杂的多种形式。例如，可以序列化只包含公共字段和公共属性的类，如 XML 序列化简介中所示。下面的代码示例讨论各种高级方案，包括如何使用 XML 序列化生成符合特定 XML 架构 (XSD) 文档的 XML 流。

序列化数据集

除了序列化公共类的实例外，还可序列化 DataSet 的实例，如下面的代码示例所示。

```
Private Sub SerializeDataSet(filename As String)
    Dim ser As XmlSerializer = new XmlSerializer(GetType(DataSet))
    ' Creates a DataSet; adds a table, column, and ten rows.
    Dim ds As DataSet = new DataSet("myDataSet")
    Dim t As DataTable = new DataTable("table1")
    Dim c As DataColumn = new DataColumn("thing")
    t.Columns.Add(c)
    ds.Tables.Add(t)
    Dim r As DataRow
    Dim i As Integer
    for i = 0 to 10
        r = t.NewRow()
        r(0) = "Thing " &  i
        t.Rows.Add(r)
    Next
    Dim writer As TextWriter = new StreamWriter(filename)
    ser.Serialize(writer, ds)
    writer.Close()
End Sub
```

```
private void SerializeDataSet(string filename){
    XmlSerializer ser = new XmlSerializer(typeof(DataSet));

    // Creates a DataSet; adds a table, column, and ten rows.
    DataSet ds = new DataSet("myDataSet");
    DataTable t = new DataTable("table1");
    DataColumn c = new DataColumn("thing");
    t.Columns.Add(c);
    ds.Tables.Add(t);
    DataRow r;
    for(int i = 0; i<10;i++){
        r = t.NewRow();
        r[0] = "Thing " + i;
        t.Rows.Add(r);
    }
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, ds);
    writer.Close();
}
```

序列化 XmlElement 和 XmlNode

还可序列化 XmlElement 或 XmlNode 类的实例，如下面的代码示例所示。

```
private Sub SerializeElement(filename As String)
    Dim ser As XmlSerializer = new XmlSerializer(GetType(XmlElement))
    Dim myElement As XmlElement = _
    new XmlDocument().CreateElement("MyElement", "ns")
    myElement.InnerText = "Hello World"
    Dim writer As TextWriter = new StreamWriter(filename)
    ser.Serialize(writer, myElement)
    writer.Close()
End Sub

Private Sub SerializeNode(filename As String)
    Dim ser As XmlSerializer = _
    new XmlSerializer(GetType(XmlNode))
    Dim myNode As XmlNode = new XmlDocument(). _
    CreateNode(XmlNodeType.Element, "MyNode", "ns")
    myNode.InnerText = "Hello Node"
    Dim writer As TextWriter = new StreamWriter(filename)
    ser.Serialize(writer, myNode)
    writer.Close()
End Sub
```

```
private void SerializeElement(string filename){
    XmlSerializer ser = new XmlSerializer(typeof(XmlElement));
    XmlElement myElement=
    new XmlDocument().CreateElement("MyElement", "ns");
    myElement.InnerText = "Hello World";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myElement);
    writer.Close();
}

private void SerializeNode(string filename){
    XmlSerializer ser = new XmlSerializer(typeof(XmlNode));
    XmlNode myNode= new XmlDocument().
    CreateNode(XmlNodeType.Element, "MyNode", "ns");
    myNode.InnerText = "Hello Node";
    TextWriter writer = new StreamWriter(filename);
    ser.Serialize(writer, myNode);
    writer.Close();
}
```

序列化包含返回复杂对象的字段的类

如果属性或字段返回一个复杂对象(如数组或类实例)，则 XmlSerializer 将其转换为嵌套在主 XML 文档内的元素。例如，以下代码示例中的第一个类返回第二个类的实例。

```
Public Class PurchaseOrder
    Public MyAddress As Address
End Class

Public Class Address
    Public FirstName As String
End Class
```

```
public class PurchaseOrder
{
    public Address MyAddress;
}
public class Address
{
    public string FirstName;
}
```

已序列化的 XML 输出可能如下所示。

```
<PurchaseOrder>
    <MyAddress>
        <FirstName>George</FirstName>
    </MyAddress>
</PurchaseOrder>
```

序列化对象数组

还可以序列化返回对象数组的字段，如下面的代码示例所示。

```
Public Class PurchaseOrder
    public ItemsOrders () As Item
End Class

Public Class Item
    Public ItemID As String
    Public ItemPrice As decimal
End Class
```

```
public class PurchaseOrder
{
    public Item [] ItemsOrders;
}

public class Item
{
    public string ItemID;
    public decimal ItemPrice;
}
```

如果两个项已排序，则已序列化的类实例可能如下所示。

```
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <ItemsOrders>
        <Item>
            <ItemID>aaa111</ItemID>
            <ItemPrice>34.22</ItemPrice>
        </Item>
        <Item>
            <ItemID>bbb222</ItemID>
            <ItemPrice>2.89</ItemPrice>
        </Item>
    </ItemsOrders>
</PurchaseOrder>
```

序列化实现 ICollection 接口的类

您可以通过实现 ICollection 接口创建自己的集合类，并使用 XmlSerializer 序列化这些类的实例。请注意，当类实现 ICollection 接口时，只序列化该类包含的集合，而不会序列化添加至该类的任何公共属性或字段。该类必须包含 Add 方法和 Item 属性(C# 索引器)才能被序列化 。

```
Imports System.Collections
Imports System.IO
Imports System.Xml.Serialization

Public Class Test
    Shared Sub Main()
        Dim t As Test= new Test()
        t.SerializeCollection("coll.xml")
    End Sub

    Private Sub SerializeCollection(filename As String)
        Dim Emps As Employees  = new Employees()
        ' Note that only the collection is serialized -- not the
        ' CollectionName or any other public property of the class.
        Emps.CollectionName = "Employees"
        Dim John100 As Employee = new Employee("John", "100xxx")
        Emps.Add(John100)
        Dim x As XmlSerializer = new XmlSerializer(GetType(Employees))
        Dim writer As TextWriter = new StreamWriter(filename)
        x.Serialize(writer, Emps)
        writer.Close()
    End Sub
End Class

Public Class Employees
    Implements ICollection
    Public CollectionName As String
    Private empArray As ArrayList = new ArrayList()

    Public ReadOnly Default Overloads _
    Property Item(index As Integer) As Employee
        get
        return CType (empArray(index), Employee)
        End Get
    End Property

    Public Sub CopyTo(a As Array, index As Integer) _
    Implements ICollection.CopyTo
        empArray.CopyTo(a, index)
    End Sub

    Public ReadOnly Property Count () As integer Implements _
    ICollection.Count
        get
            Count = empArray.Count
        End Get

    End Property

    Public ReadOnly Property SyncRoot ()As Object _
    Implements ICollection.SyncRoot
        get
        return me
        End Get
    End Property

    Public ReadOnly Property IsSynchronized () As Boolean _
    Implements ICollection.IsSynchronized
        get
        return false
        End Get
```

```
        End Set
    End Property

    Public Function GetEnumerator() As IEnumerator _
    Implements IEnumerable.GetEnumerator

        return empArray.GetEnumerator()
    End Function

    Public Function Add(newEmployee As Employee) As Integer
        empArray.Add(newEmployee)
        return empArray.Count
    End Function
End Class

Public Class Employee
    Public EmpName As String
    Public EmpID As String

    Public Sub New ()
    End Sub

    Public Sub New (newName As String , newID As String )
        EmpName = newName
        EmpID = newID
    End Sub
End Class
```

```
using System;
using System.Collections;
using System.IO;
using System.Xml.Serialization;

public class Test {
    static void Main(){
        Test t = new Test();
        t.SerializeCollection("coll.xml");
    }

    private void SerializeCollection(string filename){
        Employees Emps = new Employees();
        // Note that only the collection is serialized -- not the
        // CollectionName or any other public property of the class.
        Emps.CollectionName = "Employees";
        Employee John100 = new Employee("John", "100xxx");
        Emps.Add(John100);
        XmlSerializer x = new XmlSerializer(typeof(Employees));
        TextWriter writer = new StreamWriter(filename);
        x.Serialize(writer, Emps);
    }
}
public class Employees:ICollection {
    public string CollectionName;
    private ArrayList empArray = new ArrayList();

    public Employee this[int index]{
        get{return (Employee) empArray[index];}
    }

    public void CopyTo(Array a, int index){
        empArray.CopyTo(a, index);
    }
    public int Count{
        get{return empArray.Count;}
    }
    public object SyncRoot{
        get{return this;}
    }
    public bool IsSynchronized{
        get{return false;}
    }
    public IEnumerator GetEnumerator(){
        return empArray.GetEnumerator();
    }

    public void Add(Employee newEmployee){
        empArray.Add(newEmployee);
    }
}

public class Employee {
    public string EmpName;
    public string EmpID;
    public Employee(){}
    public Employee(string empName, string empID){
        EmpName = empName;
        EmpID = empID;
    }
}
```

订单**示例**

您可以将下面的示例代码剪切并粘贴到以 .cs 或 .vb 文件扩展名重命名的文本文件中。使用 C# 或 Visual Basic

编译器编译该文件，然后使用生成的可执行文件的名称运行该文件。

此示例使用一个简单方案演示如何创建对象的实例，并使用 Serialize 方法将该对象实例序列化为文件流。将 XML 流保存到文件，然后使用 Deserialize 方法读回该文件，并将其重新构造为原始对象的副本。

在此示例中，对名为 `PurchaseOrder` 的类进行序列化，然后进行反序列化。另一个名为 `Address` 的类也包含在内，因为名为 `ShipTo` 的公共字段必须设置为 `Address`。同样，`OrderedItem` 类也包含在内，因为 `OrderedItem` 对象的数组必须设置为 `OrderedItems` 字段。最后，名为 `Test` 的类包含序列化和反序列化这些类的代码。

`CreatePO` 方法创建 `PurchaseOrder`、`Address` 和 `OrderedItem` 类对象，并设置公共字段值。该方法还构造 XmlSerializer 类的实例，该类用于序列化和反序列化 `PurchaseOrder`。请注意，代码传递的是将序列化为构造函数的类的类型。代码还创建可用于将 XML 流写入 XML 文档的 `FileStream`。

`ReadPo` 方法稍简单一些。它只创建要反序列化的对象并读出它们的值。与 `CreatePo` 方法一样，必须先构造 XmlSerializer，并将要反序列化的类的类型传递给该构造函数。此外，还需要使用 FileStream 读取 XML 文档。要反序列化对象，请调用带有 Deserialize 参数的 FileStream 方法。已反序列化的对象必须强制转换为 `PurchaseOrder` 类型的对象变量。然后代码读取已反序列化的 `PurchaseOrder` 的值。请注意，您还可以读取 PO.xml 文件，创建该文件是为了查看实际的 XML 输出。

```
Imports System.IO
Imports System.Xml
Imports System.Xml.Serialization
Imports Microsoft.VisualBasic

' The XmlRoot attribute allows you to set an alternate name
' (PurchaseOrder) for the XML element and its namespace. By
' default, the XmlSerializer uses the class name. The attribute
' also allows you to set the XML namespace for the element. Lastly,
' the attribute sets the IsNullable property, which specifies whether
' the xsi:null attribute appears if the class instance is set to
' a null reference.
<XmlRoot("PurchaseOrder", _
 Namespace := "http://www.cpandl.com", IsNullable := False)> _
Public Class PurchaseOrder
    Public ShipTo As Address
    Public OrderDate As String
    ' The XmlArrayAttribute changes the XML element name
    ' from the default of "OrderedItems" to "Items".
    <XmlArray("Items")> _
    Public OrderedItems() As OrderedItem
    Public SubTotal As Decimal
    Public ShipCost As Decimal
    Public TotalCost As Decimal
End Class

Public Class Address
    ' The XmlAttribute attribute instructs the XmlSerializer to serialize the
    ' Name field as an XML attribute instead of an XML element (XML element is
    ' the default behavior).
    <XmlAttribute()> _
    Public Name As String
    Public Line1 As String

    ' Setting the IsNullable property to false instructs the
    ' XmlSerializer that the XML attribute will not appear if
    ' the City field is set to a null reference.
    <XmlElement(IsNullable := False)> _
    Public City As String
    Public State As String
    Public Zip As String
End Class

Public Class OrderedItem
    Public ItemName As String
```

```
    Public Description As String
    Public UnitPrice As Decimal
    Public Quantity As Integer
    Public LineTotal As Decimal

    ' Calculate is a custom method that calculates the price per item
    ' and stores the value in a field.
    Public Sub Calculate()
    LineTotal = UnitPrice * Quantity
    End Sub
End Class

Public Class Test
        Public Shared Sub Main()
    ' Read and write purchase orders.
    Dim t As New Test()
    t.CreatePO("po.xml")
    t.ReadPO("po.xml")
    End Sub

    Private Sub CreatePO(filename As String)
        ' Creates an instance of the XmlSerializer class;
        ' specifies the type of object to serialize.
        Dim serializer As New XmlSerializer(GetType(PurchaseOrder))
        Dim writer As New StreamWriter(filename)
        Dim po As New PurchaseOrder()

        ' Creates an address to ship and bill to.
        Dim billAddress As New Address()
        billAddress.Name = "Teresa Atkinson"
        billAddress.Line1 = "1 Main St."
        billAddress.City = "AnyTown"
        billAddress.State = "WA"
        billAddress.Zip = "00000"
        ' Set ShipTo and BillTo to the same addressee.
        po.ShipTo = billAddress
        po.OrderDate = System.DateTime.Now.ToLongDateString()

        ' Creates an OrderedItem.
        Dim i1 As New OrderedItem()
        i1.ItemName = "Widget S"
        i1.Description = "Small widget"
        i1.UnitPrice = CDec(5.23)
        i1.Quantity = 3
        i1.Calculate()

        ' Inserts the item into the array.
        Dim items(0) As OrderedItem
        items(0) = i1
        po.OrderedItems = items
        ' Calculates the total cost.
        Dim subTotal As New Decimal()
        Dim oi As OrderedItem
        For Each oi In  items
            subTotal += oi.LineTotal
        Next oi
        po.SubTotal = subTotal
        po.ShipCost = CDec(12.51)
        po.TotalCost = po.SubTotal + po.ShipCost
        ' Serializes the purchase order, and close the TextWriter.
        serializer.Serialize(writer, po)
        writer.Close()
    End Sub

    Protected Sub ReadPO(filename As String)
        ' Creates an instance of the XmlSerializer class;
        ' specifies the type of object to be deserialized.
        Dim serializer As New XmlSerializer(GetType(PurchaseOrder))
        ' If the XML document has been altered with unknown
```

```
        ' nodes or attributes, handles them with the
        ' UnknownNode and UnknownAttribute events.
        AddHandler serializer.UnknownNode, AddressOf serializer_UnknownNode
        AddHandler serializer.UnknownAttribute, AddressOf _
        serializer_UnknownAttribute

        ' A FileStream is needed to read the XML document.
        Dim fs As New FileStream(filename, FileMode.Open)
        ' Declare an object variable of the type to be deserialized.
        Dim po As PurchaseOrder
        ' Uses the Deserialize method to restore the object's state
        ' with data from the XML document.
        po = CType(serializer.Deserialize(fs), PurchaseOrder)
        ' Reads the order date.
        Console.WriteLine(("OrderDate: " & po.OrderDate))

        ' Reads the shipping address.
        Dim shipTo As Address = po.ShipTo
        ReadAddress(shipTo, "Ship To:")
        ' Reads the list of ordered items.
        Dim items As OrderedItem() = po.OrderedItems
        Console.WriteLine("Items to be shipped:")
        Dim oi As OrderedItem
        For Each oi In items
            Console.WriteLine((ControlChars.Tab & oi.ItemName & _
            ControlChars.Tab & _
                oi.Description & ControlChars.Tab & oi.UnitPrice & _
                ControlChars.Tab & _
                oi.Quantity & ControlChars.Tab & oi.LineTotal))
        Next oi
        ' Reads the subtotal, shipping cost, and total cost.
        Console.WriteLine((ControlChars.Cr & New String _
        (ControlChars.Tab, 5) & _
        " Subtotal" & ControlChars.Tab & po.SubTotal & ControlChars.Cr & _
        New String(ControlChars.Tab, 5) & " Shipping" & ControlChars.Tab & _
        po.ShipCost & ControlChars.Cr &  New String(ControlChars.Tab, 5) & _
        " Total" & New String(ControlChars.Tab, 2) & po.TotalCost))
    End Sub

    Protected Sub ReadAddress(a As Address, label As String)
        ' Reads the fields of the Address.
        Console.WriteLine(label)
        Console.Write((ControlChars.Tab & a.Name & ControlChars.Cr & _
        ControlChars.Tab & a.Line1 & ControlChars.Cr & ControlChars.Tab & _
        a.City & ControlChars.Tab & a.State & ControlChars.Cr & _
        ControlChars.Tab & a.Zip & ControlChars.Cr))
    End Sub

    Protected Sub serializer_UnknownNode(sender As Object, e As _
    XmlNodeEventArgs)
        Console.WriteLine(("Unknown Node:" & e.Name & _
        ControlChars.Tab & e.Text))
    End Sub

    Protected Sub serializer_UnknownAttribute(sender As Object, _
    e As XmlAttributeEventArgs)
        Dim attr As System.Xml.XmlAttribute = e.Attr
        Console.WriteLine(("Unknown attribute " & attr.Name & "='" & _
        attr.Value & "'"))
    End Sub 'serializer_UnknownAttribute
End Class 'Test
```

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

// The XmlRootAttribute allows you to set the name
```

```
// The XmlRoot attribute allows you to set an alternate name
// (PurchaseOrder) for the XML element and its namespace. By
// default, the XmlSerializer uses the class name. The attribute
// also allows you to set the XML namespace for the element. Lastly,
// the attribute sets the IsNullable property, which specifies whether
// the xsi:null attribute appears if the class instance is set to
// a null reference.
[XmlRoot("PurchaseOrder", Namespace="http://www.cpandl.com",
IsNullable = false)]
public class PurchaseOrder
{
    public Address ShipTo;
    public string OrderDate;
    // The XmlArray attribute changes the XML element name
    // from the default of "OrderedItems" to "Items".
    [XmlArray("Items")]
    public OrderedItem[] OrderedItems;
    public decimal SubTotal;
    public decimal ShipCost;
    public decimal TotalCost;
}

public class Address
{
    // The XmlAttribute attribute instructs the XmlSerializer to serialize the
    // Name field as an XML attribute instead of an XML element (XML element is
    // the default behavior).
    [XmlAttribute]
    public string Name;
    public string Line1;

    // Setting the IsNullable property to false instructs the
    // XmlSerializer that the XML attribute will not appear if
    // the City field is set to a null reference.
    [XmlElement(IsNullable = false)]
    public string City;
    public string State;
    public string Zip;
}

public class OrderedItem
{
    public string ItemName;
    public string Description;
    public decimal UnitPrice;
    public int Quantity;
    public decimal LineTotal;

    // Calculate is a custom method that calculates the price per item
    // and stores the value in a field.
    public void Calculate()
    {
        LineTotal = UnitPrice * Quantity;
    }
}

public class Test
{
    public static void Main()
    {
        // Read and write purchase orders.
        Test t = new Test();
        t.CreatePO("po.xml");
        t.ReadPO("po.xml");
    }

    private void CreatePO(string filename)
    {
        // Creates an instance of the XmlSerializer class;
```

```
        // specifies the type of object to serialize.
        XmlSerializer serializer =
        new XmlSerializer(typeof(PurchaseOrder));
        TextWriter writer = new StreamWriter(filename);
        PurchaseOrder po=new PurchaseOrder();

        // Creates an address to ship and bill to.
        Address billAddress = new Address();
        billAddress.Name = "Teresa Atkinson";
        billAddress.Line1 = "1 Main St.";
        billAddress.City = "AnyTown";
        billAddress.State = "WA";
        billAddress.Zip = "00000";
        // Sets ShipTo and BillTo to the same addressee.
        po.ShipTo = billAddress;
        po.OrderDate = System.DateTime.Now.ToLongDateString();

        // Creates an OrderedItem.
        OrderedItem i1 = new OrderedItem();
        i1.ItemName = "Widget S";
        i1.Description = "Small widget";
        i1.UnitPrice = (decimal) 5.23;
        i1.Quantity = 3;
        i1.Calculate();

        // Inserts the item into the array.
        OrderedItem [] items = {i1};
        po.OrderedItems = items;
        // Calculate the total cost.
        decimal subTotal = new decimal();
        foreach(OrderedItem oi in items)
        {
            subTotal += oi.LineTotal;
        }
        po.SubTotal = subTotal;
        po.ShipCost = (decimal) 12.51;
        po.TotalCost = po.SubTotal + po.ShipCost;
        // Serializes the purchase order, and closes the TextWriter.
        serializer.Serialize(writer, po);
        writer.Close();
    }

    protected void ReadPO(string filename)
    {
        // Creates an instance of the XmlSerializer class;
        // specifies the type of object to be deserialized.
        XmlSerializer serializer = new XmlSerializer(typeof(PurchaseOrder));
        // If the XML document has been altered with unknown
        // nodes or attributes, handles them with the
        // UnknownNode and UnknownAttribute events.
        serializer.UnknownNode+= new
        XmlNodeEventHandler(serializer_UnknownNode);
        serializer.UnknownAttribute+= new
        XmlAttributeEventHandler(serializer_UnknownAttribute);

        // A FileStream is needed to read the XML document.
        FileStream fs = new FileStream(filename, FileMode.Open);
        // Declares an object variable of the type to be deserialized.
        PurchaseOrder po;
        // Uses the Deserialize method to restore the object's state
        // with data from the XML document. */
        po = (PurchaseOrder) serializer.Deserialize(fs);
        // Reads the order date.
        Console.WriteLine ("OrderDate: " + po.OrderDate);

        // Reads the shipping address.
        Address shipTo = po.ShipTo;
        ReadAddress(shipTo, "Ship To:");
        // Reads the list of ordered items.
```

```
        OrderedItem [] items = po.OrderedItems;
        Console.WriteLine("Items to be shipped:");
        foreach(OrderedItem oi in items)
        {
            Console.WriteLine("\t"+
            oi.ItemName + "\t" +
            oi.Description + "\t" +
            oi.UnitPrice + "\t" +
            oi.Quantity + "\t" +
            oi.LineTotal);
        }
        // Reads the subtotal, shipping cost, and total cost.
        Console.WriteLine(
        "\n\t\t\t\t\t Subtotal\t" + po.SubTotal +
        "\n\t\t\t\t\t Shipping\t" + po.ShipCost +
        "\n\t\t\t\t\t Total\t\t" + po.TotalCost
        );
    }

    protected void ReadAddress(Address a, string label)
    {
        // Reads the fields of the Address.
        Console.WriteLine(label);
        Console.Write("\t"+
        a.Name +"\n\t" +
        a.Line1 +"\n\t" +
        a.City +"\t" +
        a.State +"\n\t" +
        a.Zip +"\n");
    }

    protected void serializer_UnknownNode
    (object sender, XmlNodeEventArgs e)
    {
        Console.WriteLine("Unknown Node:" +   e.Name + "\t" + e.Text);
    }

    protected void serializer_UnknownAttribute
    (object sender, XmlAttributeEventArgs e)
    {
        System.Xml.XmlAttribute attr = e.Attr;
        Console.WriteLine("Unknown attribute " +
        attr.Name + "='" + attr.Value + "'");
    }
}
```

XML 输出可能如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.cpandl.com">
    <ShipTo Name="Teresa Atkinson">
        <Line1>1 Main St.</Line1>
        <City>AnyTown</City>
        <State>WA</State>
        <Zip>00000</Zip>
    </ShipTo>
    <OrderDate>Wednesday, June 27, 2001</OrderDate>
    <Items>
        <OrderedItem>
            <ItemName>Widget S</ItemName>
            <Description>Small widget</Description>
            <UnitPrice>5.23</UnitPrice>
            <Quantity>3</Quantity>
            <LineTotal>15.69</LineTotal>
        </OrderedItem>
    </Items>
    <SubTotal>15.69</SubTotal>
    <ShipCost>12.51</ShipCost>
    <TotalCost>28.2</TotalCost>
</PurchaseOrder>
```

请参阅

- XML 序列化简介
- 使用属性控制 XML 序列化
- 用来控制 XML 序列化的属性
- XmlSerializer 类
- 如何：序列化对象
- 如何：反序列化对象

XML 架构定义工具和 XML 序列化

2021/11/16 •

XML 架构定义工具(XML 架构定义工具 (Xsd.exe))是作为 Windows® 软件开发工具包 (SDK) 的一部分随 .NET Framework 工具一起安装的。该工具主要有下面两个用途:

- 生成 C# 或 Visual Basic 类文件,而这些文件符合特定的 XML 架构定义语言 (XSD) 架构。该工具将 XML 架构用作参数,并输出一个包含若干类的文件,在使用 XmlSerializer 进行序列化时,这些类符合这种架构。有关如何使用该工具生成符合特定架构的类的信息,请参阅如何:使用 XML 架构定义工具生成类和 XML 架构文档。
- 通过 .dll 或 .exe 文件生成 XML 架构文档。若要查看已经创建或已使用特性进行修改的一组文件的架构,可以将 DLL 或 EXE 作为参数传递到该工具,以便生成 XML 架构。有关如何使用该工具从一组类生成 XML 架构文档的信息,请参阅如何:使用 XML 架构定义工具生成类和 XML 架构文档。

有关使用此工具的详细信息,请参阅 XML 架构定义工具 (Xsd.exe)。

请参阅

- DataSet
- XML 序列化简介
- XML 架构定义工具 (Xsd.exe)
- XmlSerializer
- 如何:序列化对象
- 如何:反序列化对象
- 如何:使用 XML 架构定义工具生成类和 XML 架构文档
- XML 架构绑定支持

使用属性控制 XML 序列化

2021/11/16 •

使用属性可以控制对象的 XML 序列化，还可以利用同一组类创建其他 XML 流。有关创建其他 XML 流的详细信息，请参阅如何：指定 XML 流的替代元素名称。

> **NOTE**
> 如果生成的 XML 必须符合万维网联合会 (W3C) 文档简单对象访问协议 (SOAP) 1.1 第 5 节的内容，则需使用控制编码的 SOAP 序列化的特性中所列的特性。

默认情况下，XML 元素名称由类或成员名称确定。在名为 `Book` 的简单类中，名为 `ISBN` 的字段将生成 XML 元素标记 <ISBN>，如下面的示例所示。

```
Public Class Book
    Public ISBN As String
End Class
' When an instance of the Book class is serialized, it might
' produce this XML:
' <ISBN>1234567890</ISBN>.
```

```
public class Book
{
    public string ISBN;
}
// When an instance of the Book class is serialized, it might
// produce this XML:
// <ISBN>1234567890</ISBN>.
```

若要重新命名元素，可以更改这种默认行为。下面的代码演示属性 (Attribute) 如何通过设置 ElementName 的 XmlElementAttribute 属性 (Property) 实现此目的。

```
Public Class TaxRates
   < XmlElement(ElementName = "TaxRate")> _
    Public ReturnTaxRate As Decimal
End Class
```

```
public class TaxRates {
    [XmlElement(ElementName = "TaxRate")]
    public decimal ReturnTaxRate;
}
```

有关属性的详细信息，请参阅属性。有关控制 XML 序列化的特性的列表，请参阅控制 XML 序列化的特性。

控制数组序列化

XmlArrayAttribute 和 XmlArrayItemAttribute 属性旨在用于控制数组的序列化。使用这些特性可以控制元素名称、命名空间以及 XML 架构 (XSD) 数据类型（在万维网联合会 [www.w3.org] 文档"XML 架构第 2 部分：数据类型"中进行了定义）。此外，还可以指定数组所能包含的类型。

对于序列化数组时生成的封闭 XML 元素，其属性将由 XmlArrayAttribute 确定。例如，默认情况下，序列化下面的数组时，将会生成名为 `Employees` 的 XML 元素。`Employees` 元素将包含在数组类型 `Employee` 之后命名的一系列元素。

```
Public Class Group
    Public Employees() As Employee
End Class
Public Class Employee
    Public Name As String
End Class
```

```
public class Group {
    public Employee[] Employees;
}
public class Employee {
    public string Name;
}
```

序列化实例可能如下所示。

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</Employees>
</Group>
```

通过应用 XmlArrayAttribute，可以按照以下方式更改 XML 元素的名称。

```
Public Class Group
    <XmlArray("TeamMembers")> _
    Public Employees() As Employee
End Class
```

```
public class Group {
    [XmlArray("TeamMembers")]
    public Employee[] Employees;
}
```

生成的 XML 可能如下所示。

```
<Group>
<TeamMembers>
    <Employee>
        <Name>Haley</Name>
    </Employee>
</TeamMembers>
</Group>
```

另一方面，XmlArrayItemAttribute 可以控制如何序列化数组中包含的项。请注意，该特性将应用于返回数组的字段。

```
Public Class Group
    <XmlArrayItem("MemberName")> _
    Public Employee() As Employees
End Class
```

```
public class Group {
    [XmlArrayItem("MemberName")]
    public Employee[] Employees;
}
```

生成的 XML 可能如下所示。

```
<Group>
<Employees>
    <MemberName>Haley</MemberName>
</Employees>
</Group>
```

序列化派生类

XmlArrayItemAttribute 的另一种用法是，允许序列化派生类。例如，可将派生自 `Manager` 的另一个名为 `Employee` 的类添加至上一示例中。如果没有应用 XmlArrayItemAttribute，代码将在运行时失败，原因是无法识别派生类类型。若要解决这个问题，每次为每个可接受类型（基类和派生类）设置 Type 属性时，需要应用该特性两次。

```
Public Class Group
    <XmlArrayItem(Type:=GetType(Employee)), _
    XmlArrayItem(Type:=GetType(Manager))> _
    Public Employees() As Employee
End Class
Public Class Employee
    Public Name As String
End Class
Public Class Manager
Inherits Employee
    Public Level As Integer
End Class
```

```
public class Group {
    [XmlArrayItem(Type = typeof(Employee)),
    XmlArrayItem(Type = typeof(Manager))]
    public Employee[] Employees;
}
public class Employee {
    public string Name;
}
public class Manager:Employee {
    public int Level;
}
```

序列化实例可能如下所示。

```
<Group>
<Employees>
    <Employee>
        <Name>Haley</Name>
    </Employee>
    <Employee xsi:type = "Manager">
        <Name>Ann</Name>
        <Level>3</Level>
    </Employee>
</Employees>
</Group>
```

将数组作为元素序列进行序列化

通过将 XmlElementAttribute 应用于返回数组的字段，还可以将该数组作为 XML 元素的平面序列进行序列化，如下所示。

```
Public Class Group
    <XmlElement> _
    Public Employees() As Employee
End Class
```

```
public class Group {
    [XmlElement]
    public Employee[] Employees;
}
```

序列化实例可能如下所示。

```
<Group>
<Employees>
    <Name>Haley</Name>
</Employees>
<Employees>
    <Name>Noriko</Name>
</Employees>
<Employees>
    <Name>Marco</Name>
</Employees>
</Group>
```

区别两种 XML 流的另一个方法是，使用 XML 架构定义工具，从编译好的代码生成 XML 架构 (XSD) 文档文件。（有关使用该工具的详细信息，请参见XML 架构定义工具和 XML 序列化。）没有将属性应用于字段时，架构会以下列方式描述元素。

```
<xs:element minOccurs="0" maxOccurs ="1" name="Employees" type="ArrayOfEmployee" />
```

将 XmlElementAttribute 应用于字段时，生成的架构会以下列方式描述元素。

```
<xs:element minOccurs="0" maxOccurs="unbounded" name="Employees" type="Employee" />
```

序列化 ArrayList

ArrayList 类可能包含各种不同对象的集合。因此，可以按照使用数组的类似方式使用 ArrayList。您可以创建返

回单个 ArrayList 的字段，而不用创建返回类型化对象的数组的字段。但是，与数组相同的是，必须将 XmlSerializer 包含的对象的类型告知 ArrayList。为此，需要为该字段分配 XmlElementAttribute 的多个实例，如下面的示例所示。

```
Public Class Group
    <XmlElement(Type:=GetType(Employee)), _
    XmlElement(Type:=GetType(Manager))> _
    Public Info As ArrayList
End Class
```

```
public class Group {
    [XmlElement(Type = typeof(Employee)),
    XmlElement(Type = typeof(Manager))]
    public ArrayList Info;
}
```

使用 XmlRootAttribute 和 XmlTypeAttribute 控制类的序列化

能且只能应用于一个类的特性有下面两种：XmlRootAttribute 和 XmlTypeAttribute。这两种特性非常相似。XmlRootAttribute 只能应用于一个类：序列化时，该类表示 XML 文档的开始和结束元素，也就是根元素。另一方面，XmlTypeAttribute 可以应用于任何一个类，包括根类。

例如，在上面的示例中，`Group` 类就是根类，而其所有的公共字段和属性变成 XML 文档中的 XML 元素。因此，只能有一个根类。通过应用 XmlRootAttribute，可以控制 XmlSerializer 所生成的 XML 流。例如，可以更改元素名称和命名空间。

使用 XmlTypeAttribute 可以控制所生成 XML 的架构。需要通过 XML Web services 发布架构时，这项功能很有用。下面的示例将 XmlTypeAttribute 和 XmlRootAttribute 同时应用于同一个类。

```
<XmlRoot("NewGroupName"), _
XmlType("NewTypeName")> _
Public Class Group
    Public Employees() As Employee
End Class
```

```
[XmlRoot("NewGroupName")]
[XmlType("NewTypeName")]
public class Group {
    public Employee[] Employees;
}
```

如果对该类进行编译，并且使用 XML 架构定义工具生成其架构，可能会找到下面描述 `Group` 的 XML。

```
<xs:element name="NewGroupName" type="NewTypeName" />
```

相比之下，如果是对该类的实例进行序列化，则只能在 XML 文档中找到 `NewGroupName`。

```
<NewGroupName>
    . . .
</NewGroupName>
```

使用 XmlIgnoreAttribute 防止序列化

可能会出现公共属性或字段无需进行序列化的情况。例如，字段或属性可能用于包含元数据。在这样的情况下，可将 XmlIgnoreAttribute 应用于该字段或属性，而 XmlSerializer 将跳过它。

请参阅

- 用来控制 XML 序列化的属性
- 用来控制编码的 SOAP 序列化的属性
- XML 序列化简介
- XML 序列化示例
- 如何：指定 XML 流的替代元素名称
- 如何：序列化对象
- 如何：反序列化对象

用来控制 XML 序列化的属性

2021/11/16 •

通过将下表中的特性应用于类和类成员，可以控制 XmlSerializer 序列化或反序列化该类的实例的方式。若要了解这些属性如何控制 XML 序列化，请参阅使用属性控制 XML 序列化。

这些属性还可用于控制 XML Web services 生成的文本样式的 SOAP 消息。有关将这些属性应用于 XML Web service 方法的更多信息，请参阅 使用 XML Web service 进行 XML 序列化。

有关属性的详细信息，请参阅属性。

XmlAnyAttributeAttribute	公共字段、属性、参数或返回 XmlAttribute 对象数组的返回值。	反序列化时，将会使用 XmlAttribute 对象填充数组，而这些对象代表对于架构未知的所有 XML 特性。
XmlAnyElementAttribute	公共字段、属性、参数或返回 XmlElement 对象数组的返回值。	反序列化时，将会使用 XmlElement 对象填充数组，而这些对象代表对于架构未知的所有 XML 元素。
XmlArrayAttribute	公共字段、属性、参数或返回复杂对象的数组的返回值。	数组成员将作为 XML 数组的成员生成。
XmlArrayItemAttribute	公共字段、属性、参数或返回复杂对象的数组的返回值。	可以插入数组的派生类型。通常与 XmlArrayAttribute 一起应用。
XmlAttributeAttribute	公共字段、属性、参数或返回值。	成员将作为 XML 属性进行序列化。
XmlChoiceIdentifierAttribute	公共字段、属性、参数或返回值。	可以使用枚举进一步消除成员的歧义。
XmlElementAttribute	公共字段、属性、参数或返回值。	字段或属性将作为 XML 元素进行序列化。
XmlEnumAttribute	作为枚举标识符的公共字段。	枚举成员的元素名称。
XmlIgnoreAttribute	公共属性和公共字段。	序列化包含类时，应该忽略属性或字段。
XmlIncludeAttribute	公共派生类声明，以及 Web 服务描述语言 (WSDL) 文档的公共方法的返回值。	生成要在序列化时识别的架构时，应该将该类包括在内。
XmlRootAttribute	公共类声明。	控制视为 XML 根元素的属性目标的 XML 序列化。使用该属性可进一步指定命名空间和元素名称。
XmlTextAttribute	公共属性和公共字段。	属性或字段应该作为 XML 文本进行序列化。
XmlTypeAttribute	公共类声明。	XML 类型的名称和命名空间。

除了这些特性（全部位于 System.Xml.Serialization 命名空间中）之外，还可以将 DefaultValueAttribute 特性应用

于字段。如果没有指定值，使用 DefaultValueAttribute 可设置将自动分配给成员的值。

若要控制编码的 SOAP XML 序列化，请参阅控制编码的 SOAP 序列化的特性。

请参阅

- XML 和 SOAP 序列化
- XmlSerializer
- 使用属性控制 XML 序列化
- 如何：指定 XML 流的替代元素名称
- 如何：序列化对象
- 如何：反序列化对象

使用 XML Web services 进行 XML 序列化

2021/11/16 •

XML 序列化是在 XML Web services 体系结构中使用的基础传输机制，由 XmlSerializer 类执行。要控制由 XML Web service 生成的 XML，可以对用于创建 XML Web service (.asmx) 的文件的类、返回值、参数和字段应用 控制 XML 序列化的特性和 控制编码的 SOAP 序列化的特性中列出的特性。有关创建 XML Web service 的更多信息，请参阅使用 ASP.NET 的 XML Web service。

文本样式和编码样式

XML Web service 生成的 XML 可以用两种方式设置格式，一种是文本方式，另一种是编码方式，如自定义 SOAP 消息格式设置中所述。因此有两组控制 XML 序列化的属性。控制 XML 序列化的特性中列出的属性旨在控制文本样式的 XML。控制编码的 SOAP 序列化的特性中列出的特性控制编码样式。通过有选择地应用这些属性，可以调整应用程序，使其返回两种样式或其中一种。而且，这些特性可以根据需要应用于返回值和参数。

使用两种样式的示例

创建 XML Web services 时，可以对方法使用两组特性。在下面的代码示例中，名为 `MyService` 的类包含两种 XML Web services 方法：`MyLiteralMethod` 和 `MyEncodedMethod`。这两种方法执行相同的功能，即返回 `Order` 类的实例。在 `Order` 类中，XmlTypeAttribute 和 SoapTypeAttribute 特性都应用于 `OrderID` 字段，且这两个特性的 `ElementName` 属性设置为不同的值。

要运行此示例，请将代码粘贴到扩展名为 .asmx 的文件中，然后将该文件放入由 Internet 信息服务 (IIS) 管理的虚拟目录中。在 HTML 浏览器（如 Internet Explorer）中键入计算机、虚拟目录和文件的名称。

```
<%@ WebService Language="VB" Class="MyService" %>
Imports System
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Xml.Serialization
Public Class Order
    ' Both types of attributes can be applied. Depending on which type
    ' the method used, either one will affect the call.
    <SoapElement(ElementName:= "EncodedOrderID"), _
    XmlElement(ElementName:= "LiteralOrderID")> _
    public OrderID As String
End Class

Public Class MyService
    <WebMethod, SoapDocumentMethod> _
    public Function MyLiteralMethod() As Order
        Dim myOrder As Order = New Order()
        return myOrder
    End Function
    <WebMethod, SoapRpcMethod> _
    public Function MyEncodedMethod() As Order
        Dim myOrder As Order = New Order()
        return myOrder
    End Function
End Class
```

```
<%@ WebService Language="C#" Class="MyService" %>
using System;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml.Serialization;
public class Order {
    // Both types of attributes can be applied. Depending on which type
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
    public String OrderID;
}
public class MyService {
    [WebMethod][SoapDocumentMethod]
    public Order MyLiteralMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
    [WebMethod][SoapRpcMethod]
    public Order MyEncodedMethod(){
        Order myOrder = new Order();
        return myOrder;
    }
}
```

下面的代码示例调用 `MyLiteralMethod`。会将元素名称更改为“LiteralOrderID”。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <MyLiteralMethodResult>
                <LiteralOrderID>string</LiteralOrderID>
            </MyLiteralMethodResult>
        </MyLiteralMethodResponse>
    </soap:Body>
</soap:Envelope>
```

下面的代码示例调用 `MyEncodedMethod`。元素名称为“EncodedOrderID”。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://tempuri.org/" xmlns:types="http://tempuri.org/encodedTypes"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <tns:MyEncodedMethodResponse>
            <MyEncodedMethodResult href="#id1" />
        </tns:MyEncodedMethodResponse>
        <types:Order id="id1" xsi:type="types:Order">
            <EncodedOrderID xsi:type="xsd:string">string</EncodedOrderID>
        </types:Order>
    </soap:Body>
</soap:Envelope>
```

将特性应用于返回值

您还可以将特性应用于返回值，以控制命名空间和元素名称等。下面的代码示例将 `XmlElementAttribute` 特性应用于 `MyLiteralMethod` 方法的返回值。这样可让您控制命名空间和元素名称。

```
<WebMethod, SoapDocumentMethod> _
public Function MyLiteralMethod() As _
<XmlElement(Namespace:="http://www.cohowinery.com", _
ElementName:= "BookOrder")> _
Order
    Dim myOrder As Order = New Order()
    return myOrder
End Function
```

```
[return: XmlElement(Namespace = "http://www.cohowinery.com",
ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod(){
    Order myOrder = new Order();
    return myOrder;
}
```

代码被调用时，将返回类似如下所示的 XML。

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethodResponse xmlns="http://tempuri.org/">
            <BookOrder xmlns="http://www.cohowinery.com">
                <LiteralOrderID>string</LiteralOrderID>
            </BookOrder>
        </MyLiteralMethodResponse>
    </soap:Body>
</soap:Envelope>
```

应用于参数的特性

您还可以将属性应用于参数，以指定命名空间和元素名称等。下面的代码示例向 `MyLiteralMethodResponse` 方法中添加一个参数，并将 `XmlAttributeAttribute` 特性应用于该参数。为该参数设置了元素名称和命名空间。

```
<WebMethod, SoapDocumentMethod> _
public Function MyLiteralMethod(<XmlElement _
("MyOrderID", Namespace:="http://www.microsoft.com")>ID As String) As _
<XmlElement(Namespace:="http://www.cohowinery.com", _
ElementName:= "BookOrder")> _
Order
    Dim myOrder As Order = New Order()
    myOrder.OrderID = ID
    return myOrder
End Function
```

```
[return: XmlElement(Namespace = "http://www.cohowinery.com",
ElementName = "BookOrder")]
[WebMethod][SoapDocumentMethod]
public Order MyLiteralMethod([XmlElement("MyOrderID",
Namespace="http://www.microsoft.com")] string ID){
    Order myOrder = new Order();
    myOrder.OrderID = ID;
    return myOrder;
}
```

SOAP 请求会类似于：

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
        <MyLiteralMethod xmlns="http://tempuri.org/">
            <MyOrderID xmlns="http://www.microsoft.com">string</MyOrderID>
        </MyLiteralMethod>
    </soap:Body>
</soap:Envelope>
```

将特性应用于类

如果需要控制与类相关联的元素的命名空间，则可以根据需要应用 `XmlTypeAttribute`、`XmlRootAttribute` 和 `SoapTypeAttribute`。下面的代码示例将这三者全部应用于 `Order` 类。

```
<XmlType("BigBookService"), _
SoapType("SoapBookService"), _
XmlRoot("BookOrderForm")> _
Public Class Order
    ' Both types of attributes can be applied. Depending on which
    ' the method used, either one will affect the call.
    <SoapElement(ElementName:= "EncodedOrderID"), _
    XmlElement(ElementName:= "LiteralOrderID")> _
    public OrderID As String
End Class
```

```
[XmlType("BigBooksService", Namespace = "http://www.cpandl.com")]
[SoapType("SoapBookService")]
[XmlRoot("BookOrderForm")]
public class Order {
    // Both types of attributes can be applied. Depending on which
    // the method used, either one will affect the call.
    [SoapElement(ElementName = "EncodedOrderID")]
    [XmlElement(ElementName = "LiteralOrderID")]
    public String OrderID;
}
```

检查服务说明时，可以看见 `XmlTypeAttribute` 和 `SoapTypeAttribute` 的应用结果，如下面的代码示例所示。

```
<s:element name="BookOrderForm" type="s0:BigBookService" />
<s:complexType name="BigBookService">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="1" name="LiteralOrderID" type="s:string" />
  </s:sequence>

  <s:schema targetNamespace="http://tempuri.org/encodedTypes">
    <s:complexType name="SoapBookService">
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="EncodedOrderID" type="s:string" />
      </s:sequence>
    </s:complexType>
  </s:schema>
</s:complexType>
```

`XmlRootAttribute` 的作用也可以在 HTTP GET 和 HTTP POST 结果中看见，如下所示。

```
<?xml version="1.0" encoding="utf-8"?>
<BookOrderForm xmlns="http://tempuri.org/">
    <LiteralOrderID>string</LiteralOrderID>
</BookOrderForm>
```

请参阅

- XML 和 SOAP 序列化
- 用来控制编码的 SOAP 序列化的属性
- 如何：将对象序列化为 SOAP 编码的 XML 流
- 如何：替代编码的 SOAP XML 序列化
- XML 序列化简介
- 如何：序列化对象
- 如何：反序列化对象

用来控制编码的 SOAP 序列化的属性

2021/11/16 •

命名为"简单对象访问协议 (SOAP) 1.1"的万维网联合会 (W3C) 文档包含一个可选章节（第 5 节），该节描述了如何编码 SOAP 参数。要符合该规范的第 5 节，必须使用在 System.Xml.Serialization 命名空间中找到的一组特殊属性。将这些特性适当应用于类和类的成员，然后使用 XmlSerializer 序列化一个或多个类的实例。

下表显示属性、属性的应用范围及其作用。有关使用这些特性控制 XML 序列化的更多信息，请参阅如何：将对象序列化为 SOAP 编码的 XML 流以及如何：替代编码的 SOAP XML 序列化。

有关属性的详细信息，请参阅属性。

SoapAttributeAttribute	公共字段、属性、参数或返回值。	类成员将序列化为 XML 属性。
SoapElementAttribute	公共字段、属性、参数或返回值。	类将序列化为 XML 元素。
SoapEnumAttribute	作为枚举标识符的公共字段。	枚举成员的元素名称。
SoapIgnoreAttribute	公共属性和公共字段。	序列化包含类时，应该忽略属性或字段。
SoapIncludeAttribute	公共的派生类声明，以及 Web 服务描述语言 (WSDL) 文档的公共方法。	生成要在序列化时识别的架构时，应该将该类型包括在内。
SoapTypeAttribute	公共类声明。	类应序列化为 XML 类型。

请参阅

- XML 和 SOAP 序列化
- 如何：将对象序列化为 SOAP 编码的 XML 流
- 如何：替代编码的 SOAP XML 序列化
- 特性
- XmlSerializer
- 如何：序列化对象
- 如何：反序列化对象

如何：序列化对象

2021/11/16 •

要序列化对象，首先应创建要序列化的对象，然后设置其公共属性和字段。为此，必须确定 XML 流的传输格式，即它是作为流还是作为文件进行存储。例如，如果 XML 流必须以永久形式保存，则应创建 FileStream 对象。

> **NOTE**
> 有关 XML 序列化的更多示例，请参见 XML 序列化示例。

序列化对象

1. 创建对象并设置其公共字段和属性。
2. 使用对象的类型构造 XmlSerializer。有关更多信息，请参见 XmlSerializer 类构造函数。
3. 调用 Serialize 方法生成对象的公共属性和字段的 XML 流或文件表示形式。下面的示例将创建一个文件。

```
Dim myObject As MySerializableClass = New MySerializableClass()
' Insert code to set properties and fields of the object.
Dim mySerializer As XmlSerializer = New XmlSerializer(GetType(MySerializableClass))
' To write to a file, create a StreamWriter object.
Dim myWriter As StreamWriter = New StreamWriter("myFileName.xml")
mySerializer.Serialize(myWriter, myObject)
myWriter.Close()
```

```
MySerializableClass myObject = new MySerializableClass();
// Insert code to set properties and fields of the object.
XmlSerializer mySerializer = new
XmlSerializer(typeof(MySerializableClass));
// To write to a file, create a StreamWriter object.
StreamWriter myWriter = new StreamWriter("myFileName.xml");
mySerializer.Serialize(myWriter, myObject);
myWriter.Close();
```

请参阅

- XML 序列化简介
- 如何：反序列化对象

如何使用 XmlSerializer 反序列化对象

2021/11/16 •

当您反序列化对象时，传输格式确定您将创建流还是文件对象。确定了传输格式之后，就可以根据需要调用 Serialize 或 Deserialize 方法。

反序列化对象

1. 使用要反序列化的对象的类型构造 XmlSerializer。

2. 调用 Deserialize 方法以生成该对象的副本。在反序列化时，必须将返回的对象强制转换为原始对象的类型，如以下示例所示，该示例从文件反序列化该对象（尽管也可以从流反序列化该对象）。

```
' Construct an instance of the XmlSerializer with the type
' of object that is being deserialized.
Dim mySerializer As New XmlSerializer(GetType(MySerializableClass))
' To read the file, create a FileStream.
Dim myFileStream As New FileStream("myFileName.xml", FileMode.Open)
' Call the Deserialize method and cast to the object type.
Dim myObject = CType( _
mySerializer.Deserialize(myFileStream), MySerializableClass)
```

```
// Construct an instance of the XmlSerializer with the type
// of object that is being deserialized.
var mySerializer = new XmlSerializer(typeof(MySerializableClass));
// To read the file, create a FileStream.
var myFileStream = new FileStream("myFileName.xml", FileMode.Open);
// Call the Deserialize method and cast to the object type.
var myObject = (MySerializableClass) mySerializer.Deserialize(myFileStream)
```

请参阅

- XML 序列化简介
- 如何：序列化对象

如何：使用 XML 架构定义工具生成类和 XML 架构文档

2021/11/16 •

使用 XML 架构定义工具 (Xsd.exe) 可以生成描述类的 XML 架构，也可以生成 XML 架构定义的类。下面的过程说明如何执行这两种操作。

XML 架构定义工具 (Xsd.exe) 通常可在以下路径中找到：
C:\Program Files (x86)\Microsoft SDKs\Windows\{版本}\bin\NETFX {版本} Tools\

生成符合特定架构的类

1. 打开命令提示。
2. 将 XML 架构作为参数传递给 XML 架构定义工具，该工具将创建与 XML 架构精确匹配的一组类，例如：

   ```
   xsd mySchema.xsd
   ```

   该工具只能处理引用万维网联合会 2001 年 3 月 16 日的 XML 规范的架构。换句话说，XML 架构命名空间必须是“http://www.w3.org/2001/XMLSchema”，如以下示例所示。

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <xs:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace=""
   xmlns:xs="http://www.w3.org/2001/XMLSchema" />
   ```

3. 必要时用方法、属性或字段修改类。有关利用特性修改类的详细信息，请参阅使用特性控制 XML 序列化和控制编码的 SOAP 序列化的特性。

当序列化一个或多个类的实例后，检查生成的 XML 流的架构通常非常有用。例如，您可能发布架构以供其他人使用，或者可能将其与想达成一致的架构进行比较。

从一组类生成 XML 架构文档

1. 将一个或多个类编译成 DLL。
2. 打开命令提示。
3. 将 DLL 作为参数传递给 Xsd.exe，例如：

   ```
   xsd MyFile.dll
   ```

   架构将被写入，以名称“schema0.xsd”开头。

请参阅

- DataSet
- XML 架构定义工具和 XML 序列化
- XML 序列化简介
- XML 架构定义工具 (Xsd.exe)
- XmlSerializer
- 如何：序列化对象

- 如何：反序列化对象

如何：控制派生类的序列化

2021/11/16 •

使用 XmlElementAttribute 属性更改 XML 元素的名称不是自定义对象序列化的唯一方法。您还可以自定义 XML 流，具体方法为从现有类派生以及指示 XmlSerializer 实例如何序列化新类。

例如，假设有一个 `Book` 类，您可以从该类派生，并创建一个具有多个属性的 `ExpandedBook` 类。然而，序列化或反序列化时，必须指导 XmlSerializer 接受派生类型。可以通过创建一个 XmlElementAttribute 实例并将其 Type 属性设置为派生类类型来完成此操作。将 XmlElementAttribute 添加到 XmlAttributes 实例。再将 XmlAttributes 添加到 XmlAttributeOverrides 实例，指定被重写的类型和接受派生类型的成员名。这在下面的示例中显示。

示例

```
Public Class Orders
    public Books() As Book
End Class

Public Class Book
    public ISBN As String
End Class

Public Class ExpandedBook
    Inherits Book
    public NewEdition As Boolean
End Class

Public Class Run
    Shared Sub Main()
        Dim t As Run = New Run()
        t.SerializeObject("Book.xml")
        t.DeserializeObject("Book.xml")
    End Sub

    Public Sub SerializeObject(filename As String)
        ' Each overridden field, property, or type requires
        ' an XmlAttributes instance.
        Dim attrs As XmlAttributes = New XmlAttributes()

        ' Creates an XmlElementAttribute instance to override the
        ' field that returns Book objects. The overridden field
        ' returns Expanded objects instead.
        Dim attr As XmlElementAttribute = _
        New XmlElementAttribute()
        attr.ElementName = "NewBook"
        attr.Type = GetType(ExpandedBook)

        ' Adds the element to the collection of elements.
        attrs.XmlElements.Add(attr)

        ' Creates the XmlAttributeOverrides.
        Dim attrOverrides As XmlAttributeOverrides = _
        New XmlAttributeOverrides()

        ' Adds the type of the class that contains the overridden
        ' member, as well as the XmlAttributes instance to override it
        ' with, to the XmlAttributeOverrides instance.
        attrOverrides.Add(GetType(Orders), "Books", attrs)

        ' Creates the XmlSerializer using the XmlAttributeOverrides.
        Dim s As XmlSerializer  = _
```

```
        New XmlSerializer(GetType(Orders), attrOverrides)

        ' Writing the file requires a TextWriter instance.
        Dim writer As TextWriter = New StreamWriter(filename)

        ' Creates the object to be serialized.
        Dim myOrders As Orders = New Orders()

        ' Creates an object of the derived type.
        Dim b As ExpandedBook = New ExpandedBook()
        b.ISBN= "123456789"
        b.NewEdition = True
        myOrders.Books = New ExpandedBook(){b}

        ' Serializes the object.
        s.Serialize(writer,myOrders)
        writer.Close()
    End Sub

    Public Sub DeserializeObject(filename As String)
        Dim attrOverrides As XmlAttributeOverrides = _
        New XmlAttributeOverrides()
        Dim attrs As XmlAttributes = New XmlAttributes()

        ' Creates an XmlElementAttribute to override the
        ' field that returns Book objects. The overridden field
        ' returns Expanded objects instead.
        Dim attr As XmlElementAttribute = _
        New XmlElementAttribute()
        attr.ElementName = "NewBook"
        attr.Type = GetType(ExpandedBook)

        ' Adds the XmlElementAttribute to the collection of objects.
        attrs.XmlElements.Add(attr)

        attrOverrides.Add(GetType(Orders), "Books", attrs)

        ' Creates the XmlSerializer using the XmlAttributeOverrides.
        Dim s As XmlSerializer = _
        New XmlSerializer(GetType(Orders), attrOverrides)

        Dim fs As FileStream = New FileStream(filename, FileMode.Open)
        Dim myOrders As Orders = CType( s.Deserialize(fs), Orders)
        Console.WriteLine("ExpandedBook:")

        ' The difference between deserializing the overridden
        ' XML document and serializing it is this: To read the derived
        ' object values, you must declare an object of the derived type
        ' and cast the returned object to it.
        Dim expanded As ExpandedBook
        Dim b As Book
        for each b  in myOrders.Books
            expanded = CType(b, ExpandedBook)
            Console.WriteLine(expanded.ISBN)
            Console.WriteLine(expanded.NewEdition)
        Next
    End Sub
End Class
```

```
public class Orders
{
    public Book[] Books;
}

public class Book
{
    public string ISBN;
```

```
}

public class ExpandedBook:Book
{
    public bool NewEdition;
}

public class Run
{
    public void SerializeObject(string filename)
    {
        // Each overridden field, property, or type requires
        // an XmlAttributes instance.
        XmlAttributes attrs = new XmlAttributes();

        // Creates an XmlElementAttribute instance to override the
        // field that returns Book objects. The overridden field
        // returns Expanded objects instead.
        XmlElementAttribute attr = new XmlElementAttribute();
        attr.ElementName = "NewBook";
        attr.Type = typeof(ExpandedBook);

        // Adds the element to the collection of elements.
        attrs.XmlElements.Add(attr);

        // Creates the XmlAttributeOverrides instance.
        XmlAttributeOverrides attrOverrides = new XmlAttributeOverrides();

        // Adds the type of the class that contains the overridden
        // member, as well as the XmlAttributes instance to override it
        // with, to the XmlAttributeOverrides.
        attrOverrides.Add(typeof(Orders), "Books", attrs);

        // Creates the XmlSerializer using the XmlAttributeOverrides.
        XmlSerializer s =
        new XmlSerializer(typeof(Orders), attrOverrides);

        // Writing the file requires a TextWriter instance.
        TextWriter writer = new StreamWriter(filename);

        // Creates the object to be serialized.
        Orders myOrders = new Orders();

        // Creates an object of the derived type.
        ExpandedBook b = new ExpandedBook();
        b.ISBN= "123456789";
        b.NewEdition = true;
        myOrders.Books = new ExpandedBook[]{b};

        // Serializes the object.
        s.Serialize(writer,myOrders);
        writer.Close();
    }

    public void DeserializeObject(string filename)
    {
        XmlAttributeOverrides attrOverrides =
            new XmlAttributeOverrides();
        XmlAttributes attrs = new XmlAttributes();

        // Creates an XmlElementAttribute to override the
        // field that returns Book objects. The overridden field
        // returns Expanded objects instead.
        XmlElementAttribute attr = new XmlElementAttribute();
        attr.ElementName = "NewBook";
        attr.Type = typeof(ExpandedBook);

        // Adds the XmlElementAttribute to the collection of objects.
        attrs.XmlElements.Add(attr);
```

```
        attrOverrides.Add(typeof(Orders), "Books", attrs);

        // Creates the XmlSerializer using the XmlAttributeOverrides.
        XmlSerializer s =
        new XmlSerializer(typeof(Orders), attrOverrides);

        FileStream fs = new FileStream(filename, FileMode.Open);
        Orders myOrders = (Orders) s.Deserialize(fs);
        Console.WriteLine("ExpandedBook:");

        // The difference between deserializing the overridden
        // XML document and serializing it is this: To read the derived
        // object values, you must declare an object of the derived type
        // and cast the returned object to it.
        ExpandedBook expanded;
        foreach(Book b in myOrders.Books)
        {
            expanded = (ExpandedBook)b;
            Console.WriteLine(
            expanded.ISBN + "\n" +
            expanded.NewEdition);
        }
    }
}
```

请参阅

- XmlSerializer
- XmlElementAttribute
- XmlAttributes
- XmlAttributeOverrides
- XML 和 SOAP 序列化
- 如何：序列化对象
- 如何：指定 XML 流的替代元素名称

如何：指定 XML 流的替代元素名称

2021/11/16 •

使用 XmlSerializer，可以用同一组类生成多个 XML 流。由于两个不同的 XML Web services 需要的基本信息相同（略有差异），因此您或许希望用同一组类生成多个 XML 流。例如，假设有两个处理书籍订单的 XML Web services，它们都需要 ISBN 号。一个服务使用标记 <ISBN>，而另一个使用标记 <BookID>。您已经有一个名为 `Book` 的类，其中包含名为 `ISBN` 的字段。当序列化 `Book` 类的实例时，该实例将在默认情况下使用成员名称 (ISBN) 作为标记元素名称。对于第一个 XML Web services，以上行为与预期相同。但如果要将 XML 流发送至第二个 XML Web services，则必须重写序列化，以便使标记的元素名称采用 `BookID`。

用替代元素名称创建 XML 流

1. 创建 XmlElementAttribute 类的一个实例。
2. 将 ElementName 的 XmlElementAttribute 设置为“BookID”。
3. 创建 XmlAttributes 类的一个实例。
4. 向通过 `XmlElementAttribute` 的 XmlElements 属性访问的集合中添加 XmlAttributes 对象。
5. 创建 XmlAttributeOverrides 类的一个实例。
6. 将 `XmlAttributes` 添加至 XmlAttributeOverrides，同时传递要重写的对象类型以及要被重写的成员名称。
7. 用 `XmlSerializer` 创建 `XmlAttributeOverrides` 类的实例。
8. 创建 `Book` 类的实例，并将其序列化或反序列化。

示例

```
Public Function SerializeOverride()
    ' Creates an XmlElementAttribute with the alternate name.
    Dim myElementAttribute As XmlElementAttribute = _
    New XmlElementAttribute()
    myElementAttribute.ElementName = "BookID"
    Dim myAttributes As XmlAttributes = New XmlAttributes()
    myAttributes.XmlElements.Add(myElementAttribute)
    Dim myOverrides As XmlAttributeOverrides = New XmlAttributeOverrides()
    myOverrides.Add(typeof(Book), "ISBN", myAttributes)
    Dim mySerializer As XmlSerializer = _
    New XmlSerializer(GetType(Book), myOverrides)
    Dim b As Book = New Book()
    b.ISBN = "123456789"
    ' Creates a StreamWriter to write the XML stream to.
    Dim writer As StreamWriter = New StreamWriter("Book.xml")
    mySerializer.Serialize(writer, b);
End Class
```

```
public void SerializeOverride()
{
    // Creates an XmlElementAttribute with the alternate name.
    XmlElementAttribute myElementAttribute = new XmlElementAttribute();
    myElementAttribute.ElementName = "BookID";
    XmlAttributes myAttributes = new XmlAttributes();
    myAttributes.XmlElements.Add(myElementAttribute);
    XmlAttributeOverrides myOverrides = new XmlAttributeOverrides();
    myOverrides.Add(typeof(Book), "ISBN", myAttributes);
    XmlSerializer mySerializer =
    new XmlSerializer(typeof(Book), myOverrides)
    Book b = new Book();
    b.ISBN = "123456789"
    // Creates a StreamWriter to write the XML stream to.
    StreamWriter writer = new StreamWriter("Book.xml");
    mySerializer.Serialize(writer, b);
}
```

XML 流可能如下所示。

```
<Book>
    <BookID>123456789</BookID>
</Book>
```

请参阅

- XmlElementAttribute
- XmlAttributes
- XmlAttributeOverrides
- XML 和 SOAP 序列化
- XmlSerializer
- 如何：序列化对象
- 如何：反序列化对象

如何限定 XML 元素和 XML 属性名

2021/11/16 •

XmlSerializerNamespaces 类的实例所包含的 XML 命名空间必须符合万维网联合会 (W3C) 规范，即 XML 中的命名空间。

XML 命名空间提供了一种方法，用来限定 XML 文档中 XML 元素和 XML 特性的名称。限定名由前缀和本地名称组成，两者之间用冒号分隔。前缀仅用作占位符；它将映射到用于指定命名空间的 URI。统一管理的 URI 命名空间和本地名称的组合能够产生保证为全局唯一的名称。

通过创建 `XmlSerializerNamespaces` 的实例，并向对象中添加命名空间对，可以指定在 XML 文档中使用的前缀。

在 XML 文档中创建限定名称

1. 创建 `XmlSerializerNamespaces` 类的一个实例。
2. 将所有的前缀和命名空间对添加至 `XmlSerializerNamespaces`。
3. 对每个由 `System.Xml.Serialization` 序列化为 XML 文档的成员或类应用相应的 XmlSerializer 特性。

   可用的特性包括：XmlAnyElementAttribute、XmlArrayAttribute、XmlArrayItemAttribute、XmlAttributeAttribute、XmlElementAttribute、XmlRootAttribute 和 XmlTypeAttribute。
4. 将每个属性 (Attribute) 的 `Namespace` 属性 (Property) 设置为 `XmlSerializerNamespaces` 的命名空间值之一。
5. 将 `XmlSerializerNamespaces` 传递到 `Serialize` 的 `XmlSerializer` 方法。

示例

下面的示例创建一个 `XmlSerializerNamespaces`，并向对象添加两个前缀和命名空间对。该代码创建了用来序列化 `XmlSerializer` 类实例的 `Books`，并通过 `Serialize` 调用 `XmlSerializerNamespaces` 方法，使 XML 可以包含带前缀的命名空间。

```
Imports System.IO
Imports System.Xml
Imports System.Xml.Serialization

Public Module Program

    Public Sub Main()
        SerializeObject("XmlNamespaces.xml")
    End Sub

    Public Sub SerializeObject(filename As String)
        Dim mySerializer As New XmlSerializer(GetType(Books))
        ' Writing a file requires a TextWriter.
        Dim myWriter As New StreamWriter(filename)

        ' Creates an XmlSerializerNamespaces and adds two
        ' prefix-namespace pairs.
        Dim myNamespaces As New XmlSerializerNamespaces()
        myNamespaces.Add("books", "http://www.cpandl.com")
        myNamespaces.Add("money", "http://www.cohowinery.com")

        ' Creates a Book.
        Dim myBook As New Book()
        myBook.TITLE = "A Book Title"
        Dim myPrice As New Price()
        myPrice.price = CDec(9.95)
        myPrice.currency = "US Dollar"
        myBook.PRICE = myPrice
        Dim myBooks As New Books()
        myBooks.Book = myBook
        mySerializer.Serialize(myWriter, myBooks, myNamespaces)
        myWriter.Close()
    End Sub
End Module

Public Class Books
    <XmlElement([Namespace] := "http://www.cohowinery.com")> _
    Public Book As Book
End Class

<XmlType([Namespace] := "http://www.cpandl.com")> _
Public Class Book
    <XmlElement([Namespace] := "http://www.cpandl.com")> _
    Public TITLE As String
    <XmlElement([Namespace] := "http://www.cohowinery.com")> _
    Public PRICE As Price
End Class

Public Class Price
    <XmlAttribute([Namespace] := "http://www.cpandl.com")> _
    Public currency As String
    <XmlElement([Namespace] := "http://www.cohowinery.com")> _
    Public price As Decimal
End Class
```

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

public class Program
{
    public static void Main()
    {
        SerializeObject("XmlNamespaces.xml");
    }

    public static void SerializeObject(string filename)
    {
        var mySerializer = new XmlSerializer(typeof(Books));
        // Writing a file requires a TextWriter.
        TextWriter myWriter = new StreamWriter(filename);

        // Creates an XmlSerializerNamespaces and adds two
        // prefix-namespace pairs.
        var myNamespaces = new XmlSerializerNamespaces();
        myNamespaces.Add("books", "http://www.cpandl.com");
        myNamespaces.Add("money", "http://www.cohowinery.com");

        // Creates a Book.
        var myBook = new Book();
        myBook.TITLE = "A Book Title";
        var myPrice = new Price();
        myPrice.price = (decimal) 9.95;
        myPrice.currency = "US Dollar";
        myBook.PRICE = myPrice;
        var myBooks = new Books();
        myBooks.Book = myBook;
        mySerializer.Serialize(myWriter, myBooks, myNamespaces);
        myWriter.Close();
    }
}

public class Books
{
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public Book Book;
}

[XmlType(Namespace ="http://www.cpandl.com")]
public class Book
{
    [XmlElement(Namespace = "http://www.cpandl.com")]
    public string TITLE;
    [XmlElement(Namespace ="http://www.cohowinery.com")]
    public Price PRICE;
}

public class Price
{
    [XmlAttribute(Namespace = "http://www.cpandl.com")]
    public string currency;
    [XmlElement(Namespace = "http://www.cohowinery.com")]
    public decimal price;
}
```

请参阅

- XmlSerializer
- XML 架构定义工具和 XML 序列化

- XML 序列化简介
- XmlSerializer 类
- 用来控制 XML 序列化的属性
- 如何：指定 XML 流的替代元素名称
- 如何：序列化对象
- 如何：反序列化对象

如何：将对象序列化为 SOAP 编码的 XML 流

2021/11/16 •

由于 SOAP 消息是使用 XML 生成的，因此 XmlSerializer 类可用于序列化类和生成编码的 SOAP 消息。生成的 XML 符合万维网联合会文档"简单对象访问协议 (SOAP) 1.1"的第 5 节。如果您要创建通过 SOAP 消息进行通信的 XML Web services，则可以将一组特殊的 SOAP 属性应用于类和类的成员来自定义 XML 流。有关属性列表，请参阅控制编码的 SOAP 序列化的特性。

将对象序列化为 SOAP 编码的 XML 流

1. 使用 XML 架构定义工具 (Xsd.exe) 创建类。
2. 应用在 `System.Xml.Serialization` 中找到的一个或多个特殊属性。请参见"用来控制编码的 SOAP 序列化的属性"中的列表。
3. 通过创建新的 `XmlTypeMapping`，然后用已序列化类的类型调用 `SoapReflectionImporter` 方法，来创建 `ImportTypeMapping`。

   以下代码示例调用 `SoapReflectionImporter` 类的 `ImportTypeMapping` 方法来创建 `XmlTypeMapping`。

   ```
   ' Serializes a class named Group as a SOAP message.
   Dim myTypeMapping As XmlTypeMapping =
       New SoapReflectionImporter().ImportTypeMapping(GetType(Group))
   ```

   ```
   // Serializes a class named Group as a SOAP message.
   XmlTypeMapping myTypeMapping =
       new SoapReflectionImporter().ImportTypeMapping(typeof(Group));
   ```

4. 通过将 `XmlSerializer` 传递给 `XmlTypeMapping` 构造函数，来创建 XmlSerializer(XmlTypeMapping) 类的实例。

   ```
   Dim mySerializer As XmlSerializer = New XmlSerializer(myTypeMapping)
   ```

   ```
   XmlSerializer mySerializer = new XmlSerializer(myTypeMapping);
   ```

5. 调用 `Serialize` 或 `Deserialize` 方法。

示例

```
' Serializes a class named Group as a SOAP message.
Dim myTypeMapping As XmlTypeMapping =
    New SoapReflectionImporter().ImportTypeMapping(GetType(Group))
Dim mySerializer As XmlSerializer = New XmlSerializer(myTypeMapping)
```

```
// Serializes a class named Group as a SOAP message.
XmlTypeMapping myTypeMapping =
    new SoapReflectionImporter().ImportTypeMapping(typeof(Group));
XmlSerializer mySerializer = new XmlSerializer(myTypeMapping);
```

请参阅

- XML 和 SOAP 序列化
- 用来控制编码的 SOAP 序列化的属性
- 使用 XML Web services 进行 XML 序列化
- 如何：序列化对象
- 如何：反序列化对象
- 如何：替代编码的 SOAP XML 序列化

如何：替代编码的 SOAP XML 序列化

2021/11/16 •

将对象的 XML 序列化重写为 SOAP 消息的过程类似于重写标准 XML 序列化的过程。有关重写标准 XML 序列化的信息，请参见如何：指定 XML 流的替代元素名称。

将对象的序列化重写为 SOAP 消息

1. 创建 SoapAttributeOverrides 类的一个实例。
2. 为正在序列化的每个类成员创建 `SoapAttributes`。
3. 对正在序列化的成员适当创建影响 XML 序列化的一个或多个特性的实例。有关更多信息，请参见“用来控制编码的 SOAP 序列化的特性”。
4. 将 `SoapAttributes` 的相应属性设置为在步骤 3 中创建的特性。
5. 将 `SoapAttributes` 添加到 `SoapAttributeOverrides`。
6. 使用 `XmlTypeMapping` 创建 `SoapAttributeOverrides`。使用 `SoapReflectionImporter.ImportTypeMapping` 方法。
7. 使用 `XmlSerializer` 创建 `XmlTypeMapping`。
8. 序列化或反序列化对象。

示例

下面的代码示例以两种方式序列化文件：第一种方式，不重写 `XmlSerializer` 类的行为；第二种方式，重写该行为。示例包含带有几个成员的名为 `Group` 的类。已将各个特性（如 `SoapElementAttribute`）应用于类成员。当已使用 `SerializeOriginal` 方法序列化该类时，特性会控制 SOAP 消息的内容。调用 `SerializeOverride` 方法后，`XmlSerializer` 的行为会被重写，方法是创建各个特性并根据需要将 `SoapAttributes` 的属性设置为这些特性。

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Serialization;
using System.Xml.Schema;

public class Group
{
    [SoapAttribute (Namespace = "http://www.cpandl.com")]
    public string GroupName;

    [SoapAttribute(DataType = "base64Binary")]
    public Byte [] GroupNumber;

    [SoapAttribute(DataType = "date", AttributeName = "CreationDate")]
    public DateTime Today;
    [SoapElement(DataType = "nonNegativeInteger", ElementName = "PosInt")]
    public string PositiveInt;
    // This is ignored when serialized unless it is overridden.
    [SoapIgnore]
    public bool IgnoreThis;

    public GroupType Grouptype;

    [SoapInclude(typeof(Car))]
```

```
    [SoapInclude(typeof(Car))]
    public Vehicle myCar(string licNumber)
    {
        Vehicle v;
        if(licNumber == "")
            {
                v = new Car();
            v.licenseNumber = "!!!!!!";
        }
        else
        {
            v = new Car();
            v.licenseNumber = licNumber;
        }
        return v;
    }
}

public abstract class Vehicle
{
    public string licenseNumber;
    public DateTime makeDate;
}

public class Car: Vehicle
{
}

public enum GroupType
{
    // These enums can be overridden.
    small,
    large
}

public class Run
{
    public static void Main()
    {
        Run test = new Run();
        test.SerializeOriginal("SoapOriginal.xml");
        test.SerializeOverride("SoapOverrides.xml");
        test.DeserializeOriginal("SoapOriginal.xml");
        test.DeserializeOverride("SoapOverrides.xml");

    }
    public void SerializeOriginal(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlTypeMapping myMapping =
        (new SoapReflectionImporter().ImportTypeMapping(
        typeof(Group)));
        XmlSerializer mySerializer =
        new XmlSerializer(myMapping);

        // Writing the file requires a TextWriter.
        TextWriter writer = new StreamWriter(filename);

        // Creates an instance of the class that will be serialized.
        Group myGroup = new Group();

        // Sets the object properties.
        myGroup.GroupName = ".NET";

        Byte [] hexByte = new Byte[2]{Convert.ToByte(100),
        Convert.ToByte(50)};
        myGroup.GroupNumber = hexByte;

        DateTime myDate = new DateTime(2002,5,2);
```

```
        myGroup.Today = myDate;

        myGroup.PositiveInt= "10000";
        myGroup.IgnoreThis=true;
        myGroup.Grouptype= GroupType.small;
        Car thisCar =(Car)  myGroup.myCar("1234566");

        // Prints the license number just to prove the car was created.
        Console.WriteLine("License#: " + thisCar.licenseNumber + "\n");

        // Serializes the class and closes the TextWriter.
        mySerializer.Serialize(writer, myGroup);
        writer.Close();
    }

    public void SerializeOverride(string filename)
    {
        // Creates an instance of the XmlSerializer class
        // that overrides the serialization.
        XmlSerializer overRideSerializer = CreateOverrideSerializer();

        // Writing the file requires a TextWriter.
        TextWriter writer = new StreamWriter(filename);

        // Creates an instance of the class that will be serialized.
        Group myGroup = new Group();

        // Sets the object properties.
        myGroup.GroupName = ".NET";

        Byte [] hexByte = new Byte[2]{Convert.ToByte(100),
        Convert.ToByte(50)};
        myGroup.GroupNumber = hexByte;

        DateTime myDate = new DateTime(2002,5,2);
        myGroup.Today = myDate;

        myGroup.PositiveInt= "10000";
        myGroup.IgnoreThis=true;
        myGroup.Grouptype= GroupType.small;
        Car thisCar =(Car)  myGroup.myCar("1234566");

        // Serializes the class and closes the TextWriter.
        overRideSerializer.Serialize(writer, myGroup);
         writer.Close();
    }

    public void DeserializeOriginal(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlTypeMapping myMapping =
        (new SoapReflectionImporter().ImportTypeMapping(
        typeof(Group)));
        XmlSerializer mySerializer =
        new XmlSerializer(myMapping);

        TextReader reader = new StreamReader(filename);

        // Deserializes and casts the object.
        Group myGroup;
        myGroup = (Group) mySerializer.Deserialize(reader);

        Console.WriteLine(myGroup.GroupName);
        Console.WriteLine(myGroup.GroupNumber[0]);
        Console.WriteLine(myGroup.GroupNumber[1]);
        Console.WriteLine(myGroup.Today);
        Console.WriteLine(myGroup.PositiveInt);
        Console.WriteLine(myGroup.IgnoreThis);
        Console.WriteLine();
```

```
    }

    public void DeserializeOverride(string filename)
    {
        // Creates an instance of the XmlSerializer class.
        XmlSerializer overRideSerializer = CreateOverrideSerializer();
        // Reading the file requires a TextReader.
        TextReader reader = new StreamReader(filename);

        // Deserializes and casts the object.
        Group myGroup;
        myGroup = (Group) overRideSerializer.Deserialize(reader);

        Console.WriteLine(myGroup.GroupName);
        Console.WriteLine(myGroup.GroupNumber[0]);
        Console.WriteLine(myGroup.GroupNumber[1]);
        Console.WriteLine(myGroup.Today);
        Console.WriteLine(myGroup.PositiveInt);
        Console.WriteLine(myGroup.IgnoreThis);
    }

    private XmlSerializer CreateOverrideSerializer()
    {
        SoapAttributeOverrides mySoapAttributeOverrides =
        new SoapAttributeOverrides();
        SoapAttributes soapAtts = new SoapAttributes();

        SoapElementAttribute mySoapElement = new SoapElementAttribute();
        mySoapElement.ElementName = "xxxx";
        soapAtts.SoapElement = mySoapElement;
        mySoapAttributeOverrides.Add(typeof(Group), "PositiveInt",
        soapAtts);

        // Overrides the IgnoreThis property.
        SoapIgnoreAttribute myIgnore = new SoapIgnoreAttribute();
        soapAtts = new SoapAttributes();
        soapAtts.SoapIgnore = false;
        mySoapAttributeOverrides.Add(typeof(Group), "IgnoreThis",
        soapAtts);

        // Overrides the GroupType enumeration.
        soapAtts = new SoapAttributes();
        SoapEnumAttribute xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "Over1000";
        soapAtts.SoapEnum = xSoapEnum;

        // Adds the SoapAttributes to the
        // mySoapAttributeOverrides.
        mySoapAttributeOverrides.Add(typeof(GroupType), "large",
        soapAtts);

        // Creates a second enumeration and adds it.
        soapAtts = new SoapAttributes();
        xSoapEnum = new SoapEnumAttribute();
        xSoapEnum.Name = "ZeroTo1000";
        soapAtts.SoapEnum = xSoapEnum;
        mySoapAttributeOverrides.Add(typeof(GroupType), "small",
        soapAtts);

        // Overrides the Group type.
        soapAtts = new SoapAttributes();
        SoapTypeAttribute soapType = new SoapTypeAttribute();
        soapType.TypeName = "Team";
        soapAtts.SoapType = soapType;
        mySoapAttributeOverrides.Add(typeof(Group),soapAtts);

        // Creates an XmlTypeMapping that is used to create an instance
        // of the XmlSerializer class. Then returns the XmlSerializer.
        XmlTypeMapping myMapping = (new SoapReflectionImporter(
```

```
        mySoapAttributeOverrides)).ImportTypeMapping(typeof(Group));

        XmlSerializer ser = new XmlSerializer(myMapping);
        return ser;
    }
}
```

请参阅

- XML 和 SOAP 序列化
- 用来控制编码的 SOAP 序列化的属性
- 使用 XML Web services 进行 XML 序列化
- 如何：序列化对象
- 如何：反序列化对象
- 如何：将对象序列化为 SOAP 编码的 XML 流

<system.xml.serialization> 元素

2021/11/16 •

用于控制 XML 序列化的顶级元素。有关配置文件的详细信息，请参阅配置文件架构。

<configuration>
<system.xml.serialization>

语法

```
<system.xml.serialization>
</system.xml.serialization>
```

特性和元素

下列各节描述了特性、子元素和父元素。

特性

无。

子元素

“	“
<dateTimeSerialization> 元素	确定 DateTime 对象的序列化模式。
<schemaImporterExtensions> 元素	包含将 XSD 类型映射到 .NET 类型时 XmlSchemaImporter 所用的类型。

父元素

“	“
<configuration> 元素	公共语言运行库和 .NET Framework 应用程序所使用的每个配置文件中的根元素。

示例

下面的代码示例演示如何指定 DateTime 对象的序列化模式，以及将 XSD 类型映射到 .NET 类型时 XmlSchemaImporter 所用的其他类型。

```
<system.xml.serialization>
    <xmlSerializer checkDeserializeAdvances="false" />
    <dateTimeSerialization mode = "Local" />
    <schemaImporterExtensions>
        <add
        name = "MobileCapabilities"
        type = "System.Web.Mobile.MobileCapabilities,
        System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
        PublicKeyToken=b03f5f6f11d40a3a" />
    </schemaImporterExtensions>
</system.xml.serialization>
```

请参阅

- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 配置文件架构
- <dateTimeSerialization> 元素
- <schemaImporterExtensions> 元素
- <add> 的 <schemaImporterExtensions> 元素

<dateTimeSerialization> 元素

2021/11/16 •

确定 DateTime 对象的序列化模式。

<configuration>
<dateTimeSerialization>

语法

```
<dateTimeSerialization
    mode = "Roundtrip|Local"
/>
```

特性和元素

下列各节描述了特性、子元素和父元素。

特性

“	“
`mode`	可选。指定序列化模式。设置为 DateTimeSerializationSection.DateTimeSerializationMode 值之一。默认值为 RoundTrip。

子元素

无。

父元素

“	“
system.xml.serialization	用于控制 XML 序列化的顶级元素。

备注

将此属性设置为 Local 时，DateTime 对象始终设置为本地时间格式。即，序列化的数据中总是包含本地时区信息。

将此属性设置为 Roundtrip 时，系统会检查 DateTime 对象以确定这些对象位于本地时区、UTC 时区还是未指定的时区中。随后会序列化 DateTime 对象并保留该信息。这是默认行为，建议为所有不与 Framework 的较早版本通信的新应用程序使用此行为。

请参阅

- DateTime
- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 配置文件架构

- <schemaImporterExtensions> 元素
- <add> 的 <schemaImporterExtensions> 元素
- <system.xml.serialization> 元素

<schemaImporterExtensions> 元素

2021/11/16 •

包含将 XSD 类型映射到 .NET 类型时 XmlSchemaImporter 所用的类型。有关配置文件的详细信息，请参阅配置文件架构。

语法

```
<schemaImporterExtensions>
    <!-- Add types -->
</schemaImporterExtensions>
```

子元素

"	"
<add> 的 <schemaImporterExtensions> 元素	添加 XmlSchemaImporter 用来创建映射的类型。

父元素

"	"
<system.xml.serialization> 元素	用于控制 XML 序列化的顶级元素。

示例

下面的代码示例演示如何添加将 XSD 类型映射到 .NET 类型时 XmlSchemaImporter 所用的类型。

```
<system.xml.serialization>
    <schemaImporterExtensions>
        <add name = "MobileCapabilities" type =
        "System.Web.Mobile.MobileCapabilities,
        System.Web.Mobile, Version - 2.0.0.0, Culture = neutral,
        PublicKeyToken = b03f5f6f11d40a3a" />
    </schemaImporterExtensions>
</system.xml.serialization>
```

请参阅

- XmlSchemaImporter
- DateTimeSerializationSection.DateTimeSerializationMode
- 配置文件架构
- <dateTimeSerialization> 元素
- <add> 的 <schemaImporterExtensions> 元素
- <system.xml.serialization> 元素

<schemaImporterExtensions> 的 <add> 元素

2021/11/16 •

添加将 XSD 类型映射到 .NET 类型时 XmlSchemaImporter 所用的类型。有关配置文件的详细信息，请参阅配置文件架构。

<configuration>
<system.xml.serialization>
<schemaImporterExtensions>
<add>

语法

```
<add name = "typeName" type="fully qualified type [,Version=version number] [,Culture=culture]
[,PublicKeyToken= token]"/>
```

特性和元素

下列各节描述了特性、子元素和父元素。

特性

“	“
name	用于查找实例的简单名称。
type	必需。指定要添加的架构扩展类。type 特性值必须位于一行上，并且包含完全限定的类型名称。当程序集放置在全局程序集缓存 (GAC) 中时，该特性值还必须包括已签名程序集的版本、区域性和公钥标记。

子元素

无。

父元素

“	“
<schemaImporterExtensions>	包含 XmlSchemaImporter 所使用的类型。

示例

下面的代码示例添加 XmlSchemaImporter 可以在映射类型时使用的扩展类型。

```
<configuration>
  <system.xml.serialization>
    <schemaImporterExtensions>
       <add name="contoso" type="System.Web.Mobile.MobileCapabilities,
       System.Web.Mobile, Version=2.0.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a" />
    </schemaImporterExtensions>
  </system.xml.serialization>
</configuration>
```

请参阅

- XmlSchemaImporter
- <system.xml.serialization> 元素
- <schemaImporterExtensions> 元素

<xmlSerializer> 元素

2021/11/16 •

指定是否完成 XmlSerializer 进度的额外检查。

<configuration>
<system.xml.serialization>

语法

```
<xmlSerializer checkDeserializerAdvance = "true|false" />
```

特性和元素

下列各节描述了特性、子元素和父元素。

特性

checkDeserializeAdvances	指定是否已检查 XmlSerializer 的进度。将特性设置为“true”或“false”。默认值为“true”。
useLegacySerializationGeneration	指定 XmlSerializer 是否使用旧的序列化生成，该方法通过将 C# 代码写入到一个文件，然后将其编译为程序集来生成程序集。默认值为 false。

子元素

无。

父元素

<system.xml.serialization> 元素	包含 XmlSerializer 和 XmlSchemaImporter 类的配置设置。

备注

默认情况下，当反序列化不受信任的数据时，XmlSerializer 会额外提供一层防范潜在拒绝服务攻击的安全保护。它通过在反序列化期间尝试检测无限循环来实现以上保护。若检测到此类情况，将引发异常，并出现以下消息：“内部错误: 反序列化无法越过基础流。”

接收到此消息并不一定表示正在发生拒绝服务攻击。在某些极少出现的情况下，无限循环检测机制会产生误报，并对合法的传入消息引发异常。如果发现在你的特定应用程序中，合法消息被这一额外的保护层拒绝，请将 checkDeserializeAdvances 属性设置为“false”。

示例

下面的代码示例将 checkDeserializeAdvances 属性设置为“false”。

```
<configuration>
  <system.xml.serialization>
    <xmlSerializer checkDeserializeAdvances="false" />
  </system.xml.serialization>
</configuration>
```

请参阅

- XmlSerializer
- <system.xml.serialization> 元素
- XML 和 SOAP 序列化

XML 序列化程序生成器工具 (Sgen.exe)

2021/11/16 •

XML 序列化程序生成器创建指定程序集中的类型的 XML 序列化程序集。序列化程序集在序列化或反序列化指定类型的对象时，将改进 XmlSerializer 的启动性能。

语法

从命令行运行该工具。

```
sgen [options]
```

> **TIP**
> 要使 .NET Framework 工具正常发挥作用，必须正确设置 `Path`、`Include` 和 `Lib` 环境变量。可以通过运行 SDKVars.bat（位于 <SDK>\<version>\Bin 目录中）来设置这些环境变量。必须在每个命令 shell 程序中执行 SDKVars.bat。

参数

/a[ssembly]: *filename*	为由 filename 指定的程序集或可执行文件中包含的所有类型生成序列化代码。只能提供一个文件名。如果该参数重复，将使用最后一个文件名。
/c[ompiler]: *options*	指定要传递给 C# 编译器的选项。支持所有传递到编译器的 csc.exe 选项。这可用于指定应该对程序集进行签名，以及用于指定密钥文件。
/d[ebug]	生成一个可用于调试器的映像。
/f[orce]	强制覆盖同名的现有程序集。默认值为 false。
/help 或 **/?**	显示该工具的命令语法和选项。
/k[eep]	取消在生成的源文件和其他临时文件编译到序列化程序集内之后对它们的删除操作。这可用于确定工具是否正在为某个特定类型生成序列化代码。
/n[ologo]	取消显示 Microsoft 启动版权标志。
/o[ut]: *path*	指定要在其中保存生成的程序集的目录。默认: 生成的程序集的名称由输入程序集的名称加上“xmlSerializers.dll”组成。
/p[roxytypes]	仅生成 XML Web services 代理类型的序列化代码。
/r[eference]: *assemblyfiles*	指定由需要 XML 序列化的类型引用的程序集。接受多个程序集文件（由逗号分隔）。

"	"
/s[ilent]	取消显示成功消息。
/t[ype]: *type*	仅生成指定类型的序列化代码。
/v[erbose]	显示详细输出，以进行调试。列出目标程序集中无法使用 XmlSerializer 进行序列化的类型。
/?	显示该工具的命令语法和选项。

备注

不使用 XML 序列化程序生成器时，XmlSerializer 在应用程序每次运行时为每个类型生成序列化代码和一个序列化程序集。若要改进 XML 序列化的启动性能，请预先使用 Sgen.exe 工具生成那些程序集。然后可以使用应用程序部署这些程序集。

XML 序列化程序生成器还可以改进使用 XML Web services 代理与服务器通信的客户端的性能，因为在第一次加载类型时，序列化进程将不会导致性能受损。

这些生成的程序集无法在 Web 服务的服务器端使用。该工具仅能用于 Web 服务客户端和手动序列化方案。

如果包含要序列化的类型的程序集名为 MyType.dll，则关联的序列化程序集的名称将为 MyType.XmlSerializers.dll。

示例

下面的命令创建一个名为 Data.XmlSerializers.dll 的程序集，用于序列化名为 Data.dll 的程序集中包含的所有类型。

```
sgen Data.dll
```

可以从代码中引用需要序列化和反序列化 Data.dll 中的类型的 Data.XmlSerializers.dll 程序集。

请参阅

- 工具
- 命令提示

XML Schema Definition Tool (Xsd.exe)

2021/11/16 •

XML 架构定义 (Xsd.exe) 工具从 XDR、XML 和 XSD 文件或者从运行时程序集中的类生成 XML 架构或公共语言运行时类。

XML 架构定义工具 (Xsd.exe) 通常可在以下路径中找到：
C:\Program Files (x86)\Microsoft SDKs\Windows\{版本}\bin\NETFX {版本} Tools\

语法

从命令行运行该工具。

```
xsd file.xdr [-outputdir:directory][/parameters:file.xml]
xsd file.xml [-outputdir:directory] [/parameters:file.xml]
xsd file.xsd {/classes | /dataset} [/element:element]
             [/enableLinqDataSet] [/language:language]
                          [/namespace:namespace] [-outputdir:directory] [URI:uri]
                          [/parameters:file.xml]
xsd {file.dll | file.exe} [-outputdir:directory] [/type:typename [...]][/parameters:file.xml]
```

TIP
要使 .NET Framework 工具正常发挥作用，必须正确设置 `Path`、`Include` 和 `Lib` 环境变量。可以通过运行 SDKVars.bat（位于 <SDK>\<version>\Bin 目录中）来设置这些环境变量。必须在每个命令 shell 程序中执行 SDKVars.bat。

参数

"	"
file.extension	指定要转换的输入文件。必须将扩展名指定为下列之一：.xdr、.xml、.xsd、.dll 或 .exe。 如果指定一个 XDR 架构文件（.xdr 扩展名），则 Xsd.exe 将 XDR 架构转换为 XSD 架构。输出文件与 XDR 架构同名，但扩展名为 .xsd。 如果指定一个 XML 文件（.xml 扩展名），则 Xsd.exe 从文件中的数据推导出架构并产生一个 XSD 架构。输出文件与 XML 文件同名，但扩展名为 .xsd。 如果指定一个 XML 架构文件（.xsd 扩展名），则 Xsd.exe 将为对应于 XML 架构的运行时对象生成源代码。 如果指定一个运行时程序集文件（.exe 或 .dll 扩展名），则 Xsd.exe 为该程序集中的一个或多个类型生成架构。可以使用 `/type` 选项来指定为其生成架构的类型。输出架构被命名为 schema0.xsd、schema1.xsd，依此类推。仅当给定类型使用 `XMLRoot` 自定义特性指定命名空间时，Xsd.exe 才生成多个架构。

常规选项

/h[elp]	显示该工具的命令语法和选项。
/o[utputdir]:directory	指定输出文件的目录。此参数只能出现一次。默认为当前目录。
/?	显示该工具的命令语法和选项。
/p[arameters]:file.xml	从指定的 .xml 文件读取各种操作模式的选项。缩写形式为 `/p:`。有关详细信息，请参阅备注部分。

XSD 文件选项

必须为 xsd 文件仅指定下列选项中的一个。

/c[lasses]	生成与指定架构相对应的类。若要将 XML 数据读入对象中，请使用 XmlSerializer.Deserialize 方法。
/d[ataset]	生成一个从 DataSet 派生的类，该类与指定的架构相对应。若要将 XML 数据读入派生类中，请使用 DataSet.ReadXml 方法。

还可以为 .xsd 文件指定下列任何选项。

/e[lement]:element	指定架构中要为其生成代码的元素。默认情况下，键入所有元素。可以多次指定该参数。
/enableDataBinding	在所有生成的类型上实现 INotifyPropertyChanged 接口以启用数据绑定。缩写形式为 `/edb`。
/enableLinqDataSet	(缩写形式：`/eld`。)指定可使用 LINQ to DataSet 查询的生成的数据集。此选项在同时指定 /dataset 选项的情况下使用。有关详细信息，请参阅 LINQ to DataSet 概述和查询类型化数据集。有关使用 LINQ 的常规信息，请参阅语言集成查询 (LINQ) - C# 或语言集成查询 (LINQ) - Visual Basic。
/f[ields]	生成字段，而不是生成属性。默认情况下生成属性。
/l[anguage]:language	指定要使用的编程语言。从 `CS` (默认情况下为 C#)、`VB` (Visual Basic)、`JS` (JScript) 或 `VJS` (Visual J#) 中进行选择。也可指定实现 System.CodeDom.Compiler.CodeDomProvider 的类的完全限定名
/n[amespace]:namespace	为生成的类型指定运行时命名空间。默认命名空间为 `Schemas`。
/nologo	取消显示版权标志。
/order	在所有粒子成员上生成显式顺序标识符。

/o[ut]:directoryName	指定用来放置文件的输出目录。默认为当前目录。
/u[ri]:uri	为架构中要为其生成代码的元素指定 URI。该 URI（如果存在）应用于使用 `/element` 选项指定的所有元素。

DLL 和 EXE 文件选项

/t[ype]:typename	指定要为其创建架构的类型的名称。可以指定多个类型参数。如果 typename 不指定一个命名空间，则 Xsd.exe 将程序集中的所有类型与指定类型相匹配。如果 typename 指定一个命名空间，则仅匹配那个类型。如果 typename 以星号字符 (*) 结尾，则此工具匹配所有以 * 前的字符串开头的类型。如果省略 `/type` 选项，则 Xsd.exe 为程序集中的所有类型生成架构。

备注

下表显示了 Xsd.exe 执行的操作。

XDR 到 XSD	使用精简 XML 数据架构文件生成 XML 架构。XDR 为早期基于 XML 的架构格式。
XML 到 XSD	使用 XML 文件生成 XML 架构。
XSD 到 DataSet	使用 XSD 架构文件生成公共语言运行时 DataSet 类。生成的类为规则 XML 数据提供复杂对象模型。
XSD 到类	使用 XSD 架构文件生成运行时类。生成的类可以与 System.Xml.Serialization.XmlSerializer 一起使用，来读写遵循该架构的 XML 代码。
类到 XSD	使用运行时程序集文件中的一个或多个类型生成 XML 架构。生成的架构定义了 XmlSerializer 使用的 XML 格式。

Xsd.exe 只允许操作遵循由万维网联合会 (W3C) 提议的 XML 架构定义 (XSD) 语言的 XML 架构。有关 XML 架构定义提议或 XML 标准的详细信息，请参阅 https://w3.org。

通过 XML 文件设置选项

通过使用 `/parameters` 开关，可指定设置各种选项的单个 XML 文件。可设置的选项取决于您如何使用 XSD.exe 工具。选择包括生成架构、生成代码文件，或生成包含 `DataSet` 功能的代码文件。例如，生成架构时可将 `<assembly>` 元素设置为可执行文件 (.exe) 或类库文件 (.dll) 的名称，但生成代码文件时则不能。下面的 XML 演示如何将 `<generateSchemas>` 元素用于指定的可执行文件：

```
<!-- This is in a file named GenerateSchemas.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
   <assembly>ConsoleApplication1.exe</assembly>
</generateSchemas>
</xsd>
```

如果前面的 XML 包含在名为 GenerateSchemas.xml 的文件中，则通过在命令提示处键入下面的内容并按 Enter 来使用 `/parameters` 开关：

```
 xsd /p:GenerateSchemas.xml
```

另外，如果为程序集中的单个类型生成架构，则可以使用下面的 XML：

```
<!-- This is in a file named GenerateSchemaFromType.xml. -->
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateSchemas>
   <type>IDItems</type>
</generateSchemas>
</xsd>
```

但是若要使用上面的代码，您还必须在命令提示处提供程序集的名称。在命令提示处输入下面的内容（假设 XML 文件名为 GenerateSchemaFromType.xml）：

```
xsd /p:GenerateSchemaFromType.xml ConsoleApplication1.exe
```

必须为 `<generateSchemas>` 元素仅指定以下选项中的一个。

"	"
<assembly>	指定将从中生成架构的程序集。
<type>	指定程序集中找到的要为其生成架构的类型。
<xml>	指定要为其生成架构的 XML 文件。
<xdr>	指定要为其生成架构的 XDR 文件。

若要生成代码文件，请使用 `<generateClasses>` 元素。下面的示例生成一个代码文件。请注意，另外还显示了两个特性，它们允许您为生成的文件设置编程语言和命名空间。

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
<generateClasses language='VB' namespace='Microsoft.Serialization.Examples'/>
</xsd>
<!-- You must supply an .xsd file when typing in the command line.-->
<!-- For example: xsd /p:genClasses mySchema.xsd -->
```

可为 `<generateClasses>` 元素设置以下选项。

"	"
<element>	指定 .xsd 文件中要为其生成代码的元素。

<schemaImporterExtensions>	指定派生自 SchemaImporterExtension 类的类型。
<schema>	指定要为其生成代码的 XML 架构文件。可使用多个 <schema> 元素指定多个 XML 架构文件。

下表显示也可用于 `<generateClasses>` 元素的特性。

语言	指定要使用的编程语言。从 `CS`（默认情况下为 C#）、`VB` (Visual Basic)、`JS` (JScript) 或 `VJS` (Visual J#) 中进行选择。也可指定实现 CodeDomProvider 的类的完全限定名。
namespace	为生成的代码指定命名空间。命名空间必须符合 CLR 标准（例如，没有空格或反斜杠字符）。
选项	以下值之一：`none`、`properties`（生成属性而不是公共字段）、`order` 或 `enableDataBinding`（请参见前面“XSD 文件选项”一节的 `/order` 和 `/enableDataBinding` 开关）。

使用 `DataSet` 元素还可以控制如何生成 `<generateDataSet>` 代码。下面的 XML 指定生成的代码使用 `DataSet` 结构（如 DataTable 类）为指定元素创建 Visual Basic 代码。生成的数据集结构将支持 LINQ 查询。

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/'>
    <generateDataSet language='VB' namespace='Microsoft.Serialization.Examples' enableLinqDataSet='true'>
    </generateDataSet>
</xsd>
```

可为 `<generateDataSet>` 元素设置以下选项。

<schema>	指定要为其生成代码的 XML 架构文件。可使用多个 <schema> 元素指定多个 XML 架构文件。

下表显示可与 `<generateDataSet>` 元素一起使用的特性。

enableLinqDataSet	指定可使用 LINQ to DataSet 查询的生成的数据集。默认值为 False。
语言	指定要使用的编程语言。从 `CS`（默认情况下为 C#）、`VB` (Visual Basic)、`JS` (JScript) 或 `VJS` (Visual J#) 中进行选择。也可指定实现 CodeDomProvider 的类的完全限定名。
namespace	为生成的代码指定命名空间。命名空间必须符合 CLR 标准（例如，没有空格或反斜杠字符）。

有些特性可在顶级 `<xsd>` 元素上设置。这些选项可用于任何子元素（`<generateSchemas>`、`<generateClasses>` 或 `<generateDataSet>`）。下面的 XML 代码在名为“MyOutputDirectory”的输出目录中为名为“IDItems”的元素生成代码。

```
<xsd xmlns='http://microsoft.com/dotnet/tools/xsd/' output='MyOutputDirectory'>
<generateClasses>
    <element>IDItems</element>
</generateClasses>
</xsd>
```

下表显示也可用于 `<xsd>` 元素的特性。

"	"
输出	将放置生成的架构或代码文件的目录的名称。
nologo	取消显示版权标志。设置为 `true` 或 `false`。
帮助	显示该工具的命令语法和选项。设置为 `true` 或 `false`。

示例

下面的命令从 `myFile.xdr` 生成一个 XML 架构并将它保存到当前目录中。

```
xsd myFile.xdr
```

下面的命令从 `myFile.xml` 生成一个 XML 架构并将它保存到指定目录中。

```
xsd myFile.xml /outputdir:myOutputDir
```

下面的命令生成一个与 C# 语言中的指定架构相对应的数据集，并在当前目录中将其保存为 `XSDSchemaFile.cs`。

```
xsd /dataset /language:CS XSDSchemaFile.xsd
```

下面的命令为程序集 `myAssembly.dll` 中的所有类型生成 XML 架构，并在当前目录中将它们保存为 `schema0.xsd`。

```
xsd myAssembly.dll
```

请参阅

- DataSet
- System.Xml.Serialization.XmlSerializer
- 工具
- 命令提示
- LINQ to DataSet 概述
- 查询类型化数据集
- LINQ（语言集成查询）(C#)
- LINQ（语言集成查询）(Visual Basic)

文件和流 I/O

2021/11/16 •

文件和流 I/O(输入/输出)是指在存储媒介中传入或传出数据。在 .NET 中，`System.IO` 命名空间包含允许以异步方式和同步方式对数据流和文件进行读取和写入操作的类型。这些命名空间还包含对文件执行压缩和解压缩的类型，以及通过管道和串行端口启用通信的类型。

文件是一个由字节组成的有序的命名集合，它具有永久存储。在处理文件时，你将处理目录路径、磁盘存储、文件和目录名称。相反，流是一个字节序列，可用于对后备存储进行读取和写入操作，后备存储可以是多个存储媒介之一(例如，磁盘或内存)。正如存在除磁盘之外的多种后备存储一样，也存在除文件流之外的多种流(如网络、内存和管道流)。

文件和目录

你可以使用 System.IO 命名空间中的类型与文件和目录进行交互。例如，你可以获取和设置文件和目录的属性，并基于搜索条件检索文件和目录的集合。

若要深入了解 Windows 系统中的路径命名约定和文件路径的表示方式，包括在 .NET Core 1.1 及更高版本和 .NET Framework 4.6.2 及更高版本中支持的 DOS 设备语法，请参阅 Windows 系统中的文件路径格式。

下面是一些常用的文件和目录类：

- File - 提供用于创建、复制、删除、移动和打开文件的静态方法，并可帮助创建 FileStream 对象。
- FileInfo - 提供用于创建、复制、删除、移动和打开文件的实例方法，并可帮助创建 FileStream 对象。
- Directory - 提供用于创建、移动和枚举目录和子目录的静态方法。
- DirectoryInfo - 提供用于创建、移动和枚举目录和子目录的实例方法。
- Path - 提供用于以跨平台的方式处理目录字符串的方法和属性。

调用文件系统方法时，应始终提供强大的异常处理。有关更多信息，请参阅处理 I/O 错误异常。

除了使用这些类之外，Visual Basic 用户还可以对文件 I/O 使用 Microsoft.VisualBasic.FileIO.FileSystem 类提供的方法和属性。

请参阅如何：复制目录、如何：创建目录列表和如何：枚举目录和文件。

流

抽象基类 Stream 支持读取和写入字节。所有表示流的类都继承自 Stream 类。Stream 类及其派生类提供数据源和存储库的常见视图，使程序员不必了解操作系统和基础设备的具体细节。

流涉及三个基本操作：

- 读取 - 将数据从流传输到数据结构(如字节数组)中。
- 写入 - 将数据从数据源传输到流。
- 查找 - 对流中的当前位置进行查询和修改。

根据基础数据源或存储库，流可能只支持这些功能中的一部分。例如，PipeStream 类不支持查找。流的 CanRead、CanWrite 和 CanSeek 属性指定流支持的操作。

下面是一些常用的流类：

- FileStream - 用于对文件进行读取和写入操作。
- IsolatedStorageFileStream - 用于对独立存储中的文件进行读取或写入操作。
- MemoryStream - 用于作为后备存储对内存进行读取和写入操作。
- BufferedStream - 用于改进读取和写入操作的性能。
- NetworkStream - 用于通过网络套接字进行读取和写入。
- PipeStream - 用于通过匿名和命名管道进行读取和写入。
- CryptoStream - 用于将数据流链接到加密转换。

有关异步使用流的示例，请参阅异步文件 I/O。

读取器和编写器

System.IO 命名空间还提供用于在流中读取和写入已编码字符的类型。通常，流用于字节输入和输出。读取器和编写器类型处理编码字符与字节之间的来回转换，以便流可以完成操作。每个读取器和编写器类都与流关联，可以通过类的 `BaseStream` 属性进行检索。

下面是一些常用的读取器和编写器类：

- BinaryReader 和 BinaryWriter - 用于将基元数据类型作为二进制值进行读取和写入。
- StreamReader 和 StreamWriter - 用于通过使用编码值在字符和字节之间来回转换来读取和写入字符。
- StringReader 和 StringWriter - 用于从字符串读取字符以及将字符写入字符串中。
- TextReader 和 TextWriter - 用作其他读取器和编写器（读取和写入字符和字符串，而不是二进制数据）的抽象基类。

请参阅如何：从文件读取文本、如何：向文件写入文本、如何：从字符串中读取字符和如何：向字符串写入字符。

异步 I/O 操作

读取或写入大量数据会占用大量资源。如果你的应用程序需要保持对用户的响应性，则你应异步执行这些任务。在执行同步 I/O 操作时，UI 线程将受阻，直至完成占用大量资源的操作。在开发 Windows 8.x 应用商店应用时，使用异步 I/O 操作可防止造成应用已停止工作的印象。

异步成员在其名称中包含 `Async`，如 CopyToAsync、FlushAsync、ReadAsync 和 WriteAsync 方法。你可以将这些方法与 `async` 和 `await` 关键字一起使用。

有关详细信息，请参阅异步文件 I/O。

压缩

压缩是指减小文件大小以便存储的过程。解压缩是提取压缩文件的内容以使这些内容采用可用格式的过程。System.IO.Compression 命名空间包含用于对文件和流进行压缩或解压缩的类型。

在对文件和流进行压缩和解压缩时，经常使用以下类：

- ZipArchive - 用于在 zip 存档中创建和检索条目。
- ZipArchiveEntry - 用于表示压缩文件。
- ZipFile - 用于创建、提取和打开压缩包。
- ZipFileExtensions - 用于创建和提供压缩包中的条目。

- DeflateStream - 用于使用 Deflate 算法对流进行压缩和解压缩。
- GZipStream - 用于采用 gzip 数据格式对流进行压缩和解压缩。

请参阅如何：压缩和解压缩文件。

独立存储

独立存储是一种数据存储机制，它在代码与保存的数据之间定义了标准化的关联方式，从而提供隔离性和安全性。存储提供按用户、程序集和（可选）域隔离的虚拟文件系统。当你的应用程序无权访问用户文件时，独立存储特别有用。你可以通过一种由计算机的安全策略控制的方式保存应用程序的设置或文件。

独立存储不可用于 Windows 8.x 应用商店应用；请改用 Windows.Storage 命名空间中的应用程序数据类。有关详细信息，请参阅应用程序数据。

在实现独立存储时，经常使用以下类：

- IsolatedStorage - 提供用于独立存储实现的基类。
- IsolatedStorageFile - 提供包含文件和目录的独立存储区。
- IsolatedStorageFileStream - 公开独立存储中的文件。

请参阅独立存储。

Windows 应用商店应用程序中的 I/O 操作

适用于 Windows 8.x 应用商店应用的 .NET 包含许多用于对流进行读取和写入操作的类型；但是，该集不包含所有的 .NET I/O 类型。

在 Windows 8.x 应用商店应用中使用 I/O 操作时，要注意一些重要差异：

- 专门与文件操作相关的类型（如 File、FileInfo、Directory 和 DirectoryInfo）未包含在适用于 Windows 8.x 应用商店应用的 .NET 中。请改用 Windows 运行时的 Windows.Storage 命名空间中的类型（如 StorageFile 和 StorageFolder）。
- 独立存储不可用；请改用应用程序数据。
- 使用异步方法（如 ReadAsync 和 WriteAsync）可防止 UI 线程受阻。
- 基于路径的压缩类型 ZipFile 和 ZipFileExtensions 不可用。请改用 Windows.Storage.Compression 命名空间中的所有类型。

如果需要，你可以在 .NET Framework 流和 Windows 运行时流之间进行转换。有关详细信息，请参阅如何：在 .NET Framework 流和 Windows 运行时流之间进行转换或 WindowsRuntimeStreamExtensions。

要深入了解 Windows 8.x 应用商店应用中的 I/O 操作，请参阅快速入门：对文件执行读取和写入操作。

I/O 和安全性

在使用 System.IO 命名空间中的类时，你必须遵循操作系统安全性要求（如访问控制列表 (ACL)）来控制对文件和目录的访问。此要求是在所有 FileIOPermission 要求之外的要求。可以用编程方式管理 ACL。有关详细信息，请参阅如何：添加或删除访问控制列表条目。

默认安全策略将阻止 Internet 或 Intranet 应用程序访问用户计算机上的文件。因此，在编写将通过 Internet 或 Intranet 下载的代码时，请不要使用需要物理文件路径的 I/O 类。请改用 .NET 应用程序的独立存储。

仅在构造流时执行安全性检查。因此，请不要打开流并将其传递给受信程度较低的代码或应用程序域。

相关主题

- 通用 I/O 任务
  提供与文件、目录和流关联的 I/O 任务的列表以及指向每个任务的相关内容和示例的链接。
- 异步文件 I/O
  描述异步 I/O 的性能优势和基本操作。
- 独立存储
  描述一种数据存储机制，该机制通过定义标准的代码与保存数据的关联方式来提供隔离和安全性。
- 管道
  描述 .NET 中的匿名和命名管道操作。
- 内存映射文件
  描述内存映射文件，这些文件包含虚拟内存中磁盘上文件的内容。可以使用内存映射文件编辑非常大的文件和创建共享内存以进行进程间通信。

Windows 系统中的文件路径格式

2021/11/16 •

System.IO 命名空间中很多类型的成员都包括 `path` 参数，让你可以指定指向某个文件系统资源的绝对路径或相对路径。此路径随后会传递至 Windows 文件系统 API。本主题讨论可在 Windows 系统上使用的文件路径格式。

传统 DOS 路径

标准的 DOS 路径可由以下三部分组成：

- 卷号或驱动器号，后跟卷分隔符 (`:`)。
- 目录名称。目录分隔符用来分隔嵌套目录层次结构中的子目录。
- 可选的文件名。目录分隔符用来分隔文件路径和文件名。

如果以上三项都存在，则为绝对路径。如未指定卷号或驱动器号，且目录名称的开头是目录分隔符，则路径属于当前驱动器根路径上的相对路径。否则路径相对于当前目录。下表显示了一些可能出现的目录和文件路径。

"	"
`C:\Documents\Newsletters\Summer2018.pdf`	`C:` 驱动器的根目录中的绝对文件路径。
`\Program Files\Custom Utilities\StringFinder.exe`	当前驱动器根路径上的绝对路径。
`2018\January.xlsx`	指向当前目录的子目录中的文件的相对路径。
`..\Publications\TravelBrochure.pdf`	指向当前目录的同级目录中的文件的相对路径。
`C:\Projects\apilibrary\apilibrary.sln`	`C:` 驱动器的根目录中的文件的绝对路径。
`C:Projects\apilibrary\apilibrary.sln`	`C:` 驱动器的当前目录中的相对路径。

> **IMPORTANT**
> 请注意最后两个路径之间的差异。两者都指定了可选的卷说明符（均为 `C:`），但前者以指定的卷的根目录开头，而后者不是。因此，前者表示 `C:` 驱动器的根目录中的绝对路径，而后者表示 `C:` 驱动器的当前目录中的相对路径。应使用前者时使用了后者是涉及 Windows 文件路径的 bug 的常见原因。

可以通过调用 Path.IsPathFullyQualified 方法来确定文件路径是否完全限定（即是说，该路径独立于当前目录，且在当前目录更改时不发生变化）。请注意，如果解析的路径始终指向同样的位置，那么此类路径可以包括相对目录段（`.` 和 `..`），而同时依然是完全限定的。

以下示例演示绝对路径和相对路径之间的差异。假定存在目录 `D:\FY2018\`，且在运行该示例之前还没有通过命令提示符为 `D:\` 设置任何当前目录。

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;

public class Example
```

```
{
   public static void Main(string[] args)
   {
      Console.WriteLine($"Current directory is '{Environment.CurrentDirectory}'");
      Console.WriteLine("Setting current directory to 'C:\\'");

      Directory.SetCurrentDirectory(@"C:\");
      string path = Path.GetFullPath(@"D:\FY2018");
      Console.WriteLine($"'D:\\FY2018' resolves to {path}");
      path = Path.GetFullPath(@"D:FY2018");
      Console.WriteLine($"'D:FY2018' resolves to {path}");

      Console.WriteLine("Setting current directory to 'D:\\Docs'");
      Directory.SetCurrentDirectory(@"D:\Docs");

      path = Path.GetFullPath(@"D:\FY2018");
      Console.WriteLine($"'D:\\FY2018' resolves to {path}");
      path = Path.GetFullPath(@"D:FY2018");

      // This will be "D:\Docs\FY2018" as it happens to match the drive of the current directory
      Console.WriteLine($"'D:FY2018' resolves to {path}");

      Console.WriteLine("Setting current directory to 'C:\\'");
      Directory.SetCurrentDirectory(@"C:\");

      path = Path.GetFullPath(@"D:\FY2018");
      Console.WriteLine($"'D:\\FY2018' resolves to {path}");

      // This will be either "D:\FY2018" or "D:\FY2018\FY2018" in the subprocess. In the sub process,
      // the command prompt set the current directory before launch of our application, which
      // sets a hidden environment variable that is considered.
      path = Path.GetFullPath(@"D:FY2018");
      Console.WriteLine($"'D:FY2018' resolves to {path}");

      if (args.Length < 1)
      {
         Console.WriteLine(@"Launching again, after setting current directory to D:\FY2018");
         Uri currentExe = new Uri(Assembly.GetExecutingAssembly().GetName().CodeBase, UriKind.Absolute);
         string commandLine = $"/C cd D:\\FY2018 & \"{currentExe.LocalPath}\" stop";
         ProcessStartInfo psi = new ProcessStartInfo("cmd", commandLine); ;
         Process.Start(psi).WaitForExit();

         Console.WriteLine("Sub process returned:");
         path = Path.GetFullPath(@"D:\FY2018");
         Console.WriteLine($"'D:\\FY2018' resolves to {path}");
         path = Path.GetFullPath(@"D:FY2018");
         Console.WriteLine($"'D:FY2018' resolves to {path}");
      }
      Console.WriteLine("Press any key to continue... ");
      Console.ReadKey();
   }
}
// The example displays the following output:
//      Current directory is 'C:\Programs\file-paths'
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
//      Setting current directory to 'D:\Docs'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\Docs\FY2018
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
//      Launching again, after setting current directory to D:\FY2018
//      Sub process returned:
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to d:\FY2018
// The subprocess displays the following output:
//      Current directory is 'C:\'
```

```
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\FY2018\FY2018
//      Setting current directory to 'D:\Docs'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\Docs\FY2018
//      Setting current directory to 'C:\'
//      'D:\FY2018' resolves to D:\FY2018
//      'D:FY2018' resolves to D:\FY2018\FY2018
```

```
Imports System.Diagnostics
Imports System.IO
Imports System.Reflection

Public Module Example

    Public Sub Main(args() As String)
        Console.WriteLine($"Current directory is '{Environment.CurrentDirectory}'")
        Console.WriteLine("Setting current directory to 'C:\'")
        Directory.SetCurrentDirectory("C:\")

        Dim filePath As String = Path.GetFullPath("D:\FY2018")
        Console.WriteLine($"'D:\\FY2018' resolves to {filePath}")
        filePath = Path.GetFullPath("D:FY2018")
        Console.WriteLine($"'D:FY2018' resolves to {filePath}")

        Console.WriteLine("Setting current directory to 'D:\\Docs'")
        Directory.SetCurrentDirectory("D:\Docs")

        filePath = Path.GetFullPath("D:\FY2018")
        Console.WriteLine($"'D:\\FY2018' resolves to {filePath}")
        filePath = Path.GetFullPath("D:FY2018")

        ' This will be "D:\Docs\FY2018" as it happens to match the drive of the current directory
        Console.WriteLine($"'D:FY2018' resolves to {filePath}")

        Console.WriteLine("Setting current directory to 'C:\\'")
        Directory.SetCurrentDirectory("C:\")

        filePath = Path.GetFullPath("D:\FY2018")
        Console.WriteLine($"'D:\\FY2018' resolves to {filePath}")

        ' This will be either "D:\FY2018" or "D:\FY2018\FY2018" in the subprocess. In the sub process,
        ' the command prompt set the current directory before launch of our application, which
        ' sets a hidden environment variable that is considered.
        filePath = Path.GetFullPath("D:FY2018")
        Console.WriteLine($"'D:FY2018' resolves to {filePath}")

        If args.Length < 1 Then
            Console.WriteLine("Launching again, after setting current directory to D:\FY2018")
            Dim currentExe As New Uri(Assembly.GetExecutingAssembly().GetName().CodeBase, UriKind.Absolute)
            Dim commandLine As String = $"/C cd D:\FY2018 & ""{currentExe.LocalPath}"" stop"
            Dim psi As New ProcessStartInfo("cmd", commandLine)
            Process.Start(psi).WaitForExit()

            Console.WriteLine("Sub process returned:")
            filePath = Path.GetFullPath("D:\FY2018")
            Console.WriteLine($"'D:\\FY2018' resolves to {filePath}")
            filePath = Path.GetFullPath("D:FY2018")
            Console.WriteLine($"'D:FY2018' resolves to {filePath}")
        End If
        Console.WriteLine("Press any key to continue... ")
        Console.ReadKey()
    End Sub
End Module
' The example displays the following output:
'      Current directory is 'C:\Programs\file-paths'
'      Setting current directory to 'C:\'
```

```
'      Setting current directory to 'C:\'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to d:\FY2018
'      Setting current directory to 'D:\Docs'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to D:\Docs\FY2018
'      Setting current directory to 'C:\'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to d:\FY2018
'      Launching again, after setting current directory to D:\FY2018
'      Sub process returned:
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to d:\FY2018
' The subprocess displays the following output:
'      Current directory is 'C:\'
'      Setting current directory to 'C:\'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to D:\FY2018\FY2018
'      Setting current directory to 'D:\Docs'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to D:\Docs\FY2018
'      Setting current directory to 'C:\'
'      'D:\FY2018' resolves to D:\FY2018
'      'D:FY2018' resolves to D:\FY2018\FY2018
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

UNC 路径

通用命名约定 (UNC) 路径，用于访问网络资源，具有以下格式：

- 一个以 `\\` 开头的服务器名或主机名。服务器名称可以为 NetBIOS 计算机名称或者 IP/FQDN 地址（支持 IPv4 和 IPv6）。
- 共享名，使用 `\` 将其与主机名分隔开。服务器名和共享名共同组成了卷。
- 目录名称。目录分隔符用来分隔嵌套目录层次结构中的子目录。
- 可选的文件名。目录分隔符用来分隔文件路径和文件名。

以下是一些 UNC 路径的示例：

“	“
`\\system07\C$\`	`system07` 上 `C:` 驱动器的根目录。
`\\Server2\Share\Test\Foo.txt`	`\\Server2\Share` 卷的测试目录中的 `Foo.txt` 文件。

UNC 路径必须始终是完全限定的。它们可以包括相对目录段（`.` 和 `..`），但是这些目录段必须是完全限定的路径的一部分。只能通过将 UNC 路径映射至驱动器号来使用相对路径。

DOS 设备路径

Windows 操作系统有一个指向所有资源（包括文件）的统一对象模型。可从控制台窗口访问这些对象路径；并通过旧版 DOS 和 UNC 路径映射到的符号链接的特殊文件，将这些对象路径公开至 Win32 层。此特殊文件夹可通过 DOS 设备路径语法（以下任一）进行访问：

`\\.\C:\Test\Foo.txt` `\\?\C:\Test\Foo.txt`

除了通过驱动器号识别驱动器以外，还可以使用卷 GUID 来识别卷。它采用以下形式：

`\\.\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt`
`\\?\Volume{b75e2c83-0000-0000-0000-602f00000000}\Test\Foo.txt`

> **NOTE**
> 从 NET Core 1.1 和 .NET Framework 4.6.2 开始，运行在 Windows 上的 .NET 实现支持 DOS 设备路径语法。

DOS 设备路径由以下部分组成：

- 设备路径说明符（`\\.\` 或 `\\?\`），它将路径标识为 DOS 设备路径。

  > **NOTE**
  > .NET Core 和 .NET 5+ 的所有版本以及从 4.6.2 开始的 .NET Framework 版本都支持 `\\?\`。

- “实际”设备对象的符号链接（如果是驱动器名称则为 C:，如果是卷 GUID 则为卷{b75e2c83-0000-0000-0000-602f00000000}）。

  设备路径说明符后的第一个 DOS 设备路径段标识了卷或驱动器。（例如，`\\?\C:\` 和 `\\.\BootPartition\`。）

  UNC 有个特定的链接，很自然地名为 `UNC`。例如：

  `\\.\UNC\Server\Share\Test\Foo.txt` `\\?\UNC\Server\Share\Test\Foo.txt`

  对于设备 UNC，服务器/共享部分构成了卷。例如在 `\\?\server1\e:\utilities\\filecomparer\` 中，服务器/共享部分是 `server1\utilities`。使用相对目录段调用 Path.GetFullPath(String, String) 等方法时，这一点非常重要；决不可能越过卷。

DOS 设备路径通过定义进行完全限定。不允许使用相对目录段（`.` 和 `..`）。也不会包含当前目录。

示例：引用同一个文件的方法

以下示例演示了一些方法，以此可在使用 System.IO 命名空间中的 API 时引用文件。该示例实例化 FileInfo 对象，并使用它的 Name 和 Length 属性来显示文件名以及文件长度。

```
using System;
using System.IO;

class Program
{
    static void Main()
    {
        string[] filenames = {
            @"c:\temp\test-file.txt",
            @"\\127.0.0.1\c$\temp\test-file.txt",
            @"\\LOCALHOST\c$\temp\test-file.txt",
            @"\\.\c:\temp\test-file.txt",
            @"\\?\c:\temp\test-file.txt",
            @"\\.\UNC\LOCALHOST\c$\temp\test-file.txt",
            @"\\127.0.0.1\c$\temp\test-file.txt" };

        foreach (var filename in filenames)
        {
            FileInfo fi = new FileInfo(filename);
            Console.WriteLine($"file {fi.Name}: {fi.Length:N0} bytes");
        }
    }
}
// The example displays output like the following:
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
//      file test-file.txt: 22 bytes
```

```
Imports System.IO

Module Program
    Sub Main()
        Dim filenames() As String = {
                "c:\temp\test-file.txt",
                "\\127.0.0.1\c$\temp\test-file.txt",
                "\\LOCALHOST\c$\temp\test-file.txt",
                "\\.\c:\temp\test-file.txt",
                "\\?\c:\temp\test-file.txt",
                "\\.\UNC\LOCALHOST\c$\temp\test-file.txt",
                "\\127.0.0.1\c$\temp\test-file.txt"}

        For Each filename In filenames
            Dim fi As New FileInfo(filename)
            Console.WriteLine($"file {fi.Name}: {fi.Length:N0} bytes")
        Next
    End Sub
End Module
```

路径规范化

几乎所有传递至 Windows API 的路径都经过规范化。规范化过程中，Windows 执行了以下步骤：

- 识别路径。
- 将当前目录应用于部分限定（相对）路径。
- 规范化组件和目录分隔符。
- 评估相对目录组件（当前目录是 `.`，父目录是 `..`）。
- 剪裁特定字符。

这种规范化隐式进行，若想显式进行规范化，可以调用 Path.GetFullPath 方法，这会包装对 GetFullPathName() 函数的调用。还可以使用 P/Invoke 直接调用 Windows GetFullPathName() 函数。

标识路径

路径规范化的第一步就是识别路径类型。路径归为以下几个类别之一：

- 它们是设备路径；就是说，它们的开头是两个分隔符和一个问号或句点（`\\?` 或 `\\.`）。
- 它们是 UNC 路径；就是说，它们的开头是两个分隔符，没有问号或句点。
- 它们是完全限定的 DOS 路径；就是说，它们的开头是驱动器号、卷分隔符和组件分隔符（`C:\`）。
- 它们指定旧设备（`CON`、`LPT1`）。
- 它们相对于当前驱动器的根路径；就是说，它们的开头是单个组件分隔符（`\`）。
- 它们相对于指定驱动器的当前目录；就是说，它们的开头是驱动器号和卷分隔符，而没有组件分隔符（`C:`）。
- 它们相对于当前目录；就是说，它们的开头是上述情况以外的任何内容（`temp\testfile.txt`）。

路径的类型决定是否以某种方式应用当前目录。还决定该路径的“根”是什么。

处理旧设备

如果路径是旧版 DOS 设备（例如 `CON`、`COM1` 或 `LPT1`），则会转换为设备路径（方法是在其前面追加 `\\.\`）并返回。

开头为旧设备名的路径始终被 Path.GetFullPath(String) 方法解释为旧设备。例如，`CON.TXT` 的 DOS 设备路径为 `\\.\CON`，而 `COM1.TXT\file1.txt` 的 DOS 设备路径为 `\\.\COM1`。

应用当前目录

如果路径非完全限定，Windows 会向其应用当前目录。不会向 UNC 和设备路径应用当前目录。带有分隔符的 `C:\` 完整驱动器也不会应用当前目录。

如果路径的开头是单个组件分隔符，则会应用当前目录中的驱动器。例如，如果文件路径是 `\utilities` 且当前目录为 `C:\temp\`，规范化后路径则为 `C:\utilities`。

如果路径开头是驱动器号和卷分隔符，而没有组件分隔符，则应用从命令行界面为指定驱动器设置的最新当前目录。如未设置最新当前目录，则只应用驱动器。例如，如果文件路径为 `D:sources`，当前目录为 `C:\Documents\`，且 D: 盘上的最新当前目录为 `D:\sources\`，则结果为 `D:\sources\sources`。这些“驱动器相对”路径是导致程序和脚本逻辑错误的常见原因。假设以字母和冒号开头的路径不是相对路径，显然是不正确的。

如果路径不是以分隔符开头的，则应用当前驱动器和当前目录。例如，如果路径是 `filecompare` 且当前目录是 `C:\utilities\`，则结果为 `C:\utilities\filecompare\`。

> **IMPORTANT**
> 相对路径在多线程应用程序（也就是大多数应用程序）中很危险，因为当前目录是分进程的设置。任何线程都能在任何时候更改当前目录。从 .NET Core 2.1 开始，可以调用 Path.GetFullPath(String, String) 方法，从想要据此解析绝对路径的相对路径和基础路径（当前目录）获取绝对路径。

规范化分隔符

将所有正斜杠（`/`）转换为标准的 Windows 分隔符，也就是反斜杠（`\`）。如果存在斜杠，前两个斜杠后面的一系列斜杠都将折叠为一个斜杠。

评估相对组件

处理路径时，会评估所有由一个或两个句点（`.` 或 `..`）组成的组件或分段：

- 如果是单句点，则删除当前分段，因为它表示当前目录。
- 如果是双句点，则删除当前分段和父级分段，因为双句点表示父级目录。

仅当父级目录未越过路径的根时，才删除父级目录。路径的根取决于路径的类型。对于 DOS 路径，根是驱动器（`C:\`）；对于UNC，根是服务器/共享（`\\Server\Share`）；对于设备路径，则为设备路径前缀（`\\?\` 或 `\\.\`）。

剪裁字符

随着分隔符的运行和相对段先遭删除，一些其他字符在规范化过程中也删除了：

- 如果某段以单个句点结尾，则删除此句点。（单个或两个句点的段在之前的步骤中已规范化。三个或更多句点的段未规范化，并且实际上是有效的文件/目录名。）
- 如果路径的结尾不是分隔符，则删除所有尾随句点和空格 (U+0020)。如果最后的段只是单个或两个句点，则按上述相对组件规则处理。

  此规则意味着可以创建以空格结尾的目录名称，方法是在空格后添加结尾分隔符。

  > **IMPORTANT**
  > 请勿创建以空格结尾的目录名或文件名。如果以空格结尾，则可能难以或者无法访问目录，并且应用程序在尝试处理这样的目录或文件时通常会操作失败。

跳过规范化过程

一般来说，任何传递到 Windows API 的路径都会（有效地）传递到 GetFullPathName 函数并进行规范化。但是有一种很重要的例外情况：以问号（而不是句点）开头的设备路径。除非路径确切地以 `\\?\` 开头（注意使用的是规范的反斜杠），否则会对它进行规范化。

为什么要跳过规范化过程？ 主要有三方面的原因：

1. 为了访问那些通常无法访问但合法的路径。例如名为 `hidden.` 的文件或目录，这是能访问它的唯一方式。
2. 为了在已规范化的情况下通过跳过规范化过程来提升性能。
3. 为了跳过路径长度的 `MAX_PATH` 检查以允许多于 259 个字符的路径（仅在 .NET Framework 上）。大多数 API 都允许这一点，也有一些例外情况。

> **NOTE**
> .NET Core 和 .NET 5+ 显式处理长路径，且不执行 `MAX_PATH` 检查。`MAX_PATH` 检查只适用于 .NET Framework。

跳过规范化和路径上限检查是两种设备路径语法之间唯一的区别；除此以外它们是完全相同的。请谨慎地选择跳过规范化，因为很容易就会创建出“一般”应用程序难以处理的路径。

如果将开头为 `\\?\` 的路径显式地传递至 GetFullPathName 函数，则依然会对它们进行规范化。

可将超过 `MAX_PATH` 字符数的路径传递至 GetFullPathName，前提是该路径不含 `\\?\`。支持任意长度的路径，只要其字符串大小在 Windows 能处理的范围内。

大小写和 Windows 文件系统

Windows 文件系统有一个让非 Window 用户和开发人员感到困惑的特性，就是路径和目录名称不区分大小写。也就是说，目录名和文件名反映的是创建它们时所使用的字符串的大小写。例如，名为

```
Directory.Create("TeStDiReCtOrY");
```

```
Directory.Create("TeStDiReCtOrY")
```

的方法创建名为 TeStDiReCtOrY 的目录。如果重命名目录或文件以改变大小写，则目录名或文件名反映的是重命名它们时所使用的字符串的大小写。例如，以下代码将文件 test.txt 重命名为 Test.txt：

```
using System.IO;

class Example
{
   static void Main()
   {
      var fi = new FileInfo(@".\test.txt");
      fi.MoveTo(@".\Test.txt");
   }
}
```

```
Imports System.IO

Module Example
    Public Sub Main()
        Dim fi As New FileInfo(".\test.txt")
        fi.MoveTo(".\Test.txt")
    End Sub
End Module
```

但是比较目录名和文件名时不区分大小写。如果搜索名为“test.txt”的文件，.NET 文件系统 API 会在比较时忽略大小写问题。“Test.txt”、“TEST.TXT”、“test.TXT”和其他任何大写和小写的字母组合都会成为“test.txt”的匹配项。

通用 I/O 任务

2021/11/16 •

System.IO 命名空间提供若干个类，通过这些类可以对文件、目录和流执行各种操作（如读取和写入）。有关详细信息，请参阅文件和流 I/O。

通用文件任务

ttttttt...	tttttttttt...
创建文本文件	File.CreateText 方法 FileInfo.CreateText 方法 File.Create 方法 FileInfo.Create 方法
写入到文本文件	如何：将文本写入文件 如何：编写文本文件 (C++/CLI)
从文本文件读取	如何：从文件中读取文本
向文件中追加文本	如何：打开并追加到日志文件 File.AppendText 方法 FileInfo.AppendText 方法
重命名或移动文件	File.Move 方法 FileInfo.MoveTo 方法
删除文件	File.Delete 方法 FileInfo.Delete 方法
复制文件	File.Copy 方法 FileInfo.CopyTo 方法
获取文件大小	FileInfo.Length 属性
获取文件特性	File.GetAttributes 方法
设置文件特性	File.SetAttributes 方法
确定文件是否存在	File.Exists 方法
从二进制文件读取	如何：对新建的数据文件进行读取和写入

tttttttt...	tttttttttttt...
写入二进制文件	如何：对新建的数据文件进行读取和写入
检索文件扩展名	Path.GetExtension 方法
检索文件的完全限定路径	Path.GetFullPath 方法
检索路径中的文件名和扩展名	Path.GetFileName 方法
更改文件扩展名	Path.ChangeExtension 方法

通用目录任务

tttttttt...	tttttttttttt...
访问特定文件夹(如“My Documents”)中的文件	如何：将文本写入文件
创建目录	Directory.CreateDirectory 方法 FileInfo.Directory 属性
创建子目录	DirectoryInfo.CreateSubdirectory 方法
重命名或移动目录	Directory.Move 方法 DirectoryInfo.MoveTo 方法
复制目录	如何：复制目录
删除目录	Directory.Delete 方法 DirectoryInfo.Delete 方法
查看目录中的文件和子目录	如何：枚举目录和文件
查明目录大小	System.IO.Directory 类
确定目录是否存在	Directory.Exists 方法

请参阅

- 文件和流 I/O
- 撰写流
- 异步文件 I/O

如何：复制目录

2021/11/16 •

本文演示如何使用 I/O 类将目录下的内容同步复制到另一个位置。

有关异步文件复制的示例，请参阅异步文件 I/O。

此示例通过将 `DirectoryCopy` 方法的 `copySubDirs` 设置为 `true` 来复制子目录。`DirectoryCopy` 方法通过对每个子目录调用其自身的方法来递归复制它们，直到再也没有子目录可以复制为止。

示例

```
using System;
using System.IO;

class DirectoryCopyExample
{
    static void Main()
    {
        // Copy from the current directory, include subdirectories.
        DirectoryCopy(".", @".\temp", true);
    }

    private static void DirectoryCopy(string sourceDirName, string destDirName, bool copySubDirs)
    {
        // Get the subdirectories for the specified directory.
        DirectoryInfo dir = new DirectoryInfo(sourceDirName);

        if (!dir.Exists)
        {
            throw new DirectoryNotFoundException(
                "Source directory does not exist or could not be found: "
                + sourceDirName);
        }

        DirectoryInfo[] dirs = dir.GetDirectories();

        // If the destination directory doesn't exist, create it.
        Directory.CreateDirectory(destDirName);

        // Get the files in the directory and copy them to the new location.
        FileInfo[] files = dir.GetFiles();
        foreach (FileInfo file in files)
        {
            string tempPath = Path.Combine(destDirName, file.Name);
            file.CopyTo(tempPath, false);
        }

        // If copying subdirectories, copy them and their contents to new location.
        if (copySubDirs)
        {
            foreach (DirectoryInfo subdir in dirs)
            {
                string tempPath = Path.Combine(destDirName, subdir.Name);
                DirectoryCopy(subdir.FullName, tempPath, copySubDirs);
            }
        }
    }
}
```

```
Imports System.IO

Class DirectoryCopyExample

    Shared Sub Main()
        ' Copy from the current directory, include subdirectories.
        DirectoryCopy(".", ".\\temp", True)
    End Sub

    Private Shared Sub DirectoryCopy( _
        ByVal sourceDirName As String, _
        ByVal destDirName As String, _
        ByVal copySubDirs As Boolean)

        ' Get the subdirectories for the specified directory.
        Dim dir As DirectoryInfo = New DirectoryInfo(sourceDirName)

        If Not dir.Exists Then
            Throw New DirectoryNotFoundException( _
                "Source directory does not exist or could not be found: " _
                + sourceDirName)
        End If

        Dim dirs As DirectoryInfo() = dir.GetDirectories()

        ' If the destination directory doesn't exist, create it.
        Directory.CreateDirectory(destDirName)

        ' Get the files in the directory and copy them to the new location.
        Dim files As FileInfo() = dir.GetFiles()
        For Each file In files
            Dim temppath As String = Path.Combine(destDirName, file.Name)
            file.CopyTo(temppath, False)
        Next file

        ' If copying subdirectories, copy them and their contents to new location.
        If copySubDirs Then
            For Each subdir In dirs
                Dim temppath As String = Path.Combine(destDirName, subdir.Name)
                DirectoryCopy(subdir.FullName, temppath, copySubDirs)
            Next subdir
        End If
    End Sub
End Class
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

请参阅

- FileInfo
- DirectoryInfo
- FileStream
- 文件和流 I/O
- 通用 I/O 任务
- 异步文件 I/O

如何：枚举目录和文件

2021/11/16 •

在处理目录和文件的大型集合时，可枚举的集合能够比数组提供更好的性能。要枚举目录和文件，请使用可返回目录和文件名的可枚举集合的方法或其 DirectoryInfo、FileInfo 或 FileSystemInfo 对象。

如果只想搜索并返回目录名称或文件名，请使用 Directory 类的枚举方法。若要搜索并返回目录或文件的其他属性，请使用 DirectoryInfo 和 FileSystemInfo 类。

可以使用这些方法中的可枚举集合作为集合类（如 List<T>）的构造函数的 IEnumerable<T> 参数。

下表总结了返回可枚举的文件和目录集合的方法：

目录名称	Directory.EnumerateDirectories
目录信息 (DirectoryInfo)	DirectoryInfo.EnumerateDirectories
文件名	Directory.EnumerateFiles
文件信息 (FileInfo)	DirectoryInfo.EnumerateFiles
文件系统条目名称	Directory.EnumerateFileSystemEntries
文件系统条目信息 (FileSystemInfo)	DirectoryInfo.EnumerateFileSystemInfos
目录和文件名称	Directory.EnumerateFileSystemEntries

NOTE

虽然可以使用可选的 SearchOption 枚举的 AllDirectories 选项迅速枚举父目录的子目录中的所有文件，但 UnauthorizedAccessException 错误可能会使枚举不完整。可以通过先枚举目录，然后枚举文件来捕获这些异常。

示例：使用目录类

以下示例使用 Directory.EnumerateDirectories(String) 方法获取指定路径中顶级目录名称的列表。

```
using System;
using System.Collections.Generic;
using System.IO;

class Program
{
    private static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            List<string> dirs = new List<string>(Directory.EnumerateDirectories(docPath));

            foreach (var dir in dirs)
            {
                Console.WriteLine($"{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}");
            }
            Console.WriteLine($"{dirs.Count} directories found.");
        }
        catch (UnauthorizedAccessException ex)
        {
            Console.WriteLine(ex.Message);
        }
        catch (PathTooLongException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

```
Imports System.Collections.Generic
Imports System.IO

Module Module1

    Sub Main()
        Try
            Dim dirPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)

            Dim dirs As List(Of String) = New List(Of String)(Directory.EnumerateDirectories(dirPath))

            For Each folder In dirs
                Console.WriteLine($"{dir.Substring(dir.LastIndexOf(Path.DirectorySeparatorChar) + 1)}")
            Next
            Console.WriteLine($"{dirs.Count} directories found.")
        Catch ex As UnauthorizedAccessException
            Console.WriteLine(ex.Message)
        Catch ex As PathTooLongException
            Console.WriteLine(ex.Message)
        End Try

    End Sub
End Module
```

以下示例使用 Directory.EnumerateFiles(String, String, SearchOption) 方法递归枚举目录中的所有文件名以及与特定模式匹配的子目录。然后它读取每个文件的每一行，并显示包含指定字符串的行及其文件名和路径。

```
using System;
using System.IO;
using System.Linq;

class Program
{
    static void Main(string[] args)
    {
        try
        {
            // Set a variable to the My Documents path.
            string docPath =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            var files = from file in Directory.EnumerateFiles(docPath, "*.txt", SearchOption.AllDirectories)
                        from line in File.ReadLines(file)
                        where line.Contains("Microsoft")
                        select new
                        {
                            File = file,
                            Line = line
                        };

            foreach (var f in files)
            {
                Console.WriteLine($"{f.File}\t{f.Line}");
            }
            Console.WriteLine($"{files.Count().ToString()} files found.");
        }
        catch (UnauthorizedAccessException uAEx)
        {
            Console.WriteLine(uAEx.Message);
        }
        catch (PathTooLongException pathEx)
        {
            Console.WriteLine(pathEx.Message);
        }
    }
}
```

```
Imports System.IO
Imports System.Xml.Linq

Module Module1

    Sub Main()
        Try
            Dim docPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
            Dim files = From chkFile In Directory.EnumerateFiles(docPath, "*.txt",
SearchOption.AllDirectories)
                        From line In File.ReadLines(chkFile)
                        Where line.Contains("Microsoft")
                        Select New With {.curFile = chkFile, .curLine = line}

            For Each f In files
                Console.WriteLine($"{f.File}\t{f.Line}")
            Next
            Console.WriteLine($"{files.Count} files found.")
        Catch uAEx As UnauthorizedAccessException
            Console.WriteLine(uAEx.Message)
        Catch pathEx As PathTooLongException
            Console.WriteLine(pathEx.Message)
        End Try
    End Sub
End Module
```

示例：使用 DirectoryInfo 类

下面的示例使用 DirectoryInfo.EnumerateDirectories 方法列出顶级目录的集合，这些顶级目录的 CreationTimeUtc 早于某个 DateTime 值。

```
using System;
using System.IO;

namespace EnumDir
{
    class Program
    {
        static void Main(string[] args)
        {
            // Set a variable to the Documents path.
            string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

            DirectoryInfo dirPrograms = new DirectoryInfo(docPath);
            DateTime StartOf2009 = new DateTime(2009, 01, 01);

            var dirs = from dir in dirPrograms.EnumerateDirectories()
            where dir.CreationTimeUtc > StartOf2009
            select new
            {
                ProgDir = dir,
            };

            foreach (var di in dirs)
            {
                Console.WriteLine($"{di.ProgDir.Name}");
            }
        }
    }
}
// </Snippet1>
```

```
Imports System.IO

Module Module1

    Sub Main()

        Dim dirPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
        Dim dirPrograms As New DirectoryInfo(dirPath)
        Dim StartOf2009 As New DateTime(2009, 1, 1)

        Dim dirs = From dir In dirPrograms.EnumerateDirectories()
                   Where dir.CreationTimeUtc > StartOf2009

        For Each di As DirectoryInfo In dirs
            Console.WriteLine("{0}", di.Name)
        Next

    End Sub

End Module
```

下例使用 DirectoryInfo.EnumerateFiles 方法列出 Length 超过 10MB 的所有文件。此示例先枚举顶级目录以捕获可能的未授权访问异常，再枚举文件。

```
using System;
using System.IO;
```

```
class Program
{
    static void Main(string[] args)
    {
        // Set a variable to the My Documents path.
        string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        DirectoryInfo diTop = new DirectoryInfo(docPath);

        try
        {
            foreach (var fi in diTop.EnumerateFiles())
            {
                try
                {
                    // Display each file over 10 MB;
                    if (fi.Length > 10000000)
                    {
                        Console.WriteLine($"{fi.FullName}\t\t{fi.Length.ToString("NO")}");
                    }
                }
                catch (UnauthorizedAccessException unAuthTop)
                {
                    Console.WriteLine($"{unAuthTop.Message}");
                }
            }

            foreach (var di in diTop.EnumerateDirectories("*"))
            {
                try
                {
                    foreach (var fi in di.EnumerateFiles("*", SearchOption.AllDirectories))
                    {
                        try
                        {
                            // Display each file over 10 MB;
                            if (fi.Length > 10000000)
                            {
                                Console.WriteLine($"{fi.FullName}\t\t{fi.Length.ToString("NO")}");
                            }
                        }
                        catch (UnauthorizedAccessException unAuthFile)
                        {
                            Console.WriteLine($"unAuthFile: {unAuthFile.Message}");
                        }
                    }
                }
                catch (UnauthorizedAccessException unAuthSubDir)
                {
                    Console.WriteLine($"unAuthSubDir: {unAuthSubDir.Message}");
                }
            }
        }
        catch (DirectoryNotFoundException dirNotFound)
        {
            Console.WriteLine($"{dirNotFound.Message}");
        }
        catch (UnauthorizedAccessException unAuthDir)
        {
            Console.WriteLine($"unAuthDir: {unAuthDir.Message}");
        }
        catch (PathTooLongException longPath)
        {
            Console.WriteLine($"{longPath.Message}");
        }
    }
}
```

```
Imports System.IO

Class Program
    Public Shared Sub Main(ByVal args As String())
        Dim dirPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)
        Dim diTop As New DirectoryInfo(dirPath)
        Try
            For Each fi In diTop.EnumerateFiles()
                Try
                    ' Display each file over 10 MB;
                    If fi.Length > 10000000 Then
                        Console.WriteLine("{0}" & vbTab & vbTab & "{1}", fi.FullName,
fi.Length.ToString("N0"))
                    End If
                Catch unAuthTop As UnauthorizedAccessException
                    Console.WriteLine($"{unAuthTop.Message}")
                End Try
            Next

            For Each di In diTop.EnumerateDirectories("*")
                Try
                    For Each fi In di.EnumerateFiles("*", SearchOption.AllDirectories)
                        Try
                            ' // Display each file over 10 MB;
                            If fi.Length > 10000000 Then
                                Console.WriteLine("{0}" & vbTab &
                                vbTab & "{1}", fi.FullName, fi.Length.ToString("N0"))
                            End If
                        Catch unAuthFile As UnauthorizedAccessException
                            Console.WriteLine($"unAuthFile: {unAuthFile.Message}")
                        End Try
                    Next
                Catch unAuthSubDir As UnauthorizedAccessException
                    Console.WriteLine($"unAuthSubDir: {unAuthSubDir.Message}")
                End Try
            Next
        Catch dirNotFound As DirectoryNotFoundException
            Console.WriteLine($"{dirNotFound.Message}")
        Catch unAuthDir As UnauthorizedAccessException
            Console.WriteLine($"unAuthDir: {unAuthDir.Message}")
        Catch longPath As PathTooLongException
            Console.WriteLine($"{longPath.Message}")
        End Try
    End Sub
End Class
```

请参阅

- 文件和流 I/O

如何：对新建的数据文件进行读取和写入

2021/11/16 •

System.IO.BinaryWriter 和 System.IO.BinaryReader 类用于写入和读取字符串以外的数据。下面的示例演示如何创建空文件流，向其写入数据并从中读取数据。

示例将在当前目录中创建名为 Test.data 的数据文件，也就同时创建了相关的 BinaryWriter 和 BinaryReader 对象，并且 BinaryWriter 对象用于向 Test.data 写入整数 0 到 10，这会将文件指针置于文件末尾 。BinaryReader 对象将文件指针设置回原始位置并读取指定的内容。

> **NOTE**
>
> 如果当前目录中已存在 Test.data，则会引发 IOException 异常。使用文件模型选项 FileMode.Create 而不是 FileMode.CreateNew 以始终创建新文件，而不引发异常。

示例

```
using System;
using System.IO;

class MyStream
{
    private const string FILE_NAME = "Test.data";

    public static void Main()
    {
        if (File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} already exists!");
            return;
        }

        using (FileStream fs = new FileStream(FILE_NAME, FileMode.CreateNew))
        {
            using (BinaryWriter w = new BinaryWriter(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    w.Write(i);
                }
            }
        }

        using (FileStream fs = new FileStream(FILE_NAME, FileMode.Open, FileAccess.Read))
        {
            using (BinaryReader r = new BinaryReader(fs))
            {
                for (int i = 0; i < 11; i++)
                {
                    Console.WriteLine(r.ReadInt32());
                }
            }
        }
    }
}


// The example creates a file named "Test.data" and writes the integers 0 through 10 to it in binary format.
// It then writes the contents of Test.data to the console with each integer on a separate line.
```

```
Imports System.IO

Class MyStream
    Private Const FILE_NAME As String = "Test.data"

    Public Shared Sub Main()
        If File.Exists(FILE_NAME) Then
            Console.WriteLine($"{FILE_NAME} already exists!")
            Return
        End If

        Using fs As New FileStream(FILE_NAME, FileMode.CreateNew)
            Using w As New BinaryWriter(fs)
                For i As Integer = 0 To 10
                    w.Write(i)
                Next
            End Using
        End Using

        Using fs As New FileStream(FILE_NAME, FileMode.Open, FileAccess.Read)
            Using r As New BinaryReader(fs)
                For i As Integer = 0 To 10
                    Console.WriteLine(r.ReadInt32())
                Next
            End Using
        End Using
    End Sub
End Class

' The example creates a file named "Test.data" and writes the integers 0 through 10 to it in binary format.
' It then writes the contents of Test.data to the console with each integer on a separate line.
```

请参阅

- BinaryReader
- BinaryWriter
- FileStream
- FileStream.Seek
- SeekOrigin
- 如何:枚举目录和文件
- 如何:打开并追加到日志文件
- 如何:从文件中读取文本
- 如何:将文本写入文件
- 如何:从字符串中读取字符
- 如何:向字符串写入字符
- 文件和流 I/O

如何：打开并追加到日志文件

2021/11/16 •

StreamWriter 和 StreamReader 对流执行字符写入和读取操作。下面的代码示例打开 log.txt 文件以供输入，或创建该文件（如果尚无文件的话），并将日志信息追加到文件末尾。然后，示例将文件内容写入标准输出以供显示。

作为此示例的替换方法，可以将信息存储为一个字符串或字符串数组，并使用 File.WriteAllText 或 File.WriteAllLines 方法实现相同的功能。

> **NOTE**
> Visual Basic 用户可以选择使用 Log 类或 FileSystem 类提供的方法和属性，以创建或写入日志文件。

示例

```
using System;
using System.IO;

class DirAppend
{
    public static void Main()
    {
        using (StreamWriter w = File.AppendText("log.txt"))
        {
            Log("Test1", w);
            Log("Test2", w);
        }

        using (StreamReader r = File.OpenText("log.txt"))
        {
            DumpLog(r);
        }
    }

    public static void Log(string logMessage, TextWriter w)
    {
        w.Write("\r\nLog Entry : ");
        w.WriteLine($"{DateTime.Now.ToLongTimeString()} {DateTime.Now.ToLongDateString()}");
        w.WriteLine("  :");
        w.WriteLine($"  :{logMessage}");
        w.WriteLine ("-------------------------------");
    }

    public static void DumpLog(StreamReader r)
    {
        string line;
        while ((line = r.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }
}
// The example creates a file named "log.txt" and writes the following lines to it,
// or appends them to the existing "log.txt" file:

// Log Entry : <current long time string> <current long date string>
//  :
//  :Test1
// -------------------------------

// Log Entry : <current long time string> <current long date string>
//  :
//  :Test2
// -------------------------------

// It then writes the contents of "log.txt" to the console.
```

```
Imports System.IO

Class DirAppend
    Public Shared Sub Main()
        Using w As StreamWriter = File.AppendText("log.txt")
            Log("Test1", w)
            Log("Test2", w)
        End Using

        Using r As StreamReader = File.OpenText("log.txt")
            DumpLog(r)
        End Using
    End Sub

    Public Shared Sub Log(logMessage As String, w As TextWriter)
        w.Write(vbCrLf + "Log Entry : ")
        w.WriteLine($"{DateTime.Now.ToLongTimeString()} {DateTime.Now.ToLongDateString()}")
        w.WriteLine("  :")
        w.WriteLine($"  :{logMessage}")
        w.WriteLine("-------------------------------")
    End Sub

    Public Shared Sub DumpLog(r As StreamReader)
        Dim line As String
        line = r.ReadLine()
        While Not (line Is Nothing)
            Console.WriteLine(line)
            line = r.ReadLine()
        End While
    End Sub
End Class

' The example creates a file named "log.txt" and writes the following lines to it,
' or appends them to the existing "log.txt" file:

' Log Entry : <current long time string> <current long date string>
'  :
'  :Test1
' -------------------------------

' Log Entry : <current long time string> <current long date string>
'  :
'  :Test2
' -------------------------------

' It then writes the contents of "log.txt" to the console.
```

请参阅

- StreamWriter
- StreamReader
- File.AppendText
- File.OpenText
- StreamReader.ReadLine
- 如何：枚举目录和文件
- 如何：对新建的数据文件进行读取和写入
- 如何：从文件中读取文本
- 如何：将文本写入文件
- 如何：从字符串中读取字符
- 如何：向字符串写入字符

- 文件和流 I/O

如何：将文本写入文件

2021/11/16 •

本主题介绍将文本写入 .NET 应用文件的不同方法。

下面的类和方法通常用于将文本写入文件：

- StreamWriter 包含同步写入文件的方法（Write 或 WriteLine）或者异步写入文件的方法（WriteAsync 和 WriteLineAsync）。
- File 提供了将文本写入文件的静态方法（例如，WriteAllLines 和 WriteAllText），或者向文件中追加文本的静态方法（例如，AppendAllLines、AppendAllText 和 AppendText）。
- Path 适用于包含文件或目录路径信息的字符串。它包含 Combine 方法，在 .NET Core 2.1 及更高版本中包含 Join 和 TryJoin 方法，允许串联字符串以生成文件或目录路径。

> **NOTE**
> 以下示例仅显示所需的最少代码量。实际的应用通常提供更可靠的错误检查和异常处理。

示例：使用 StreamWriter 同步写入文本

以下示例演示如何使用 StreamWriter 类，一次一行同步地将文本写入新文件。因为在 StreamWriter 语句中已声明并实例化 `using` 对象，所以会调用自动刷新并关闭流的 Dispose 方法。

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Create a string array with the lines of text
        string[] lines = { "First line", "Second line", "Third line" };

        // Set a variable to the Documents path.
        string docPath =
          Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the string array to a new file named "WriteLines.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath, "WriteLines.txt")))
        {
            foreach (string line in lines)
                outputFile.WriteLine(line);
        }
    }
}
// The example creates a file named "WriteLines.txt" with the following contents:
// First line
// Second line
// Third line
```

```
Imports System.IO

Class WriteText

    Public Shared Sub Main()

        ' Create a string array with the lines of text
        Dim lines() As String = {"First line", "Second line", "Third line"}

        ' Set a variable to the Documents path.
        Dim docPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)

        ' Write the string array to a new file named "WriteLines.txt".
        Using outputFile As New StreamWriter(Path.Combine(docPath, Convert.ToString("WriteLines.txt")))
            For Each line As String In lines
                outputFile.WriteLine(line)
            Next
        End Using

    End Sub

End Class

' The example creates a file named "WriteLines.txt" with the following contents:
' First line
' Second line
' Third line
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

示例：使用 StreamWriter 同步追加文本

以下示例演示如何使用 StreamWriter 类以同步方式将文本追加到第一个示例中创建的文本文件。

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {

        // Set a variable to the Documents path.
        string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Append text to an existing file named "WriteLines.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath, "WriteLines.txt"), true))
        {
            outputFile.WriteLine("Fourth Line");
        }
    }
}
// The example adds the following line to the contents of "WriteLines.txt":
// Fourth Line
```

```
Imports System.IO

Class AppendText

    Public Shared Sub Main()

        ' Set a variable to the Documents path.
        Dim docPath As String =
            Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)

        ' Append text to an existing file named "WriteLines.txt".
        Using outputFile As New StreamWriter(Path.Combine(docPath, Convert.ToString("WriteLines.txt")),
True)
            outputFile.WriteLine("Fourth Line")
        End Using

    End Sub

End Class

' The example adds the following line to the contents of "WriteLines.txt":
' Fourth Line
```

示例：使用 StreamWriter 异步写入文本

下面的示例演示如何使用 StreamWriter 类异步地将文本写入新文件。要调用 WriteAsync 方法，方法调用必须在 `async` 方法内。C# 示例需要 C# 7.1或更高版本，这会在对程序入口点上增加对 `async` 修饰符的支持。

```
using System;
using System.IO;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        // Set a variable to the Documents path.
        string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the specified text asynchronously to a new file named "WriteTextAsync.txt".
        using (StreamWriter outputFile = new StreamWriter(Path.Combine(docPath, "WriteTextAsync.txt")))
        {
            await outputFile.WriteAsync("This is a sentence.");
        }
    }
}
// The example creates a file named "WriteTextAsync.txt" with the following contents:
// This is a sentence.
```

```
Imports System.IO

Public Module Example
    Public Sub Main()
        WriteTextAsync()
    End Sub

    Async Sub WriteTextAsync()
        ' Set a variable to the Documents path.
        Dim docPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)

        ' Write the text asynchronously to a new file named "WriteTextAsync.txt".
        Using outputFile As New StreamWriter(Path.Combine(docPath, Convert.ToString("WriteTextAsync.txt")))
            Await outputFile.WriteAsync("This is a sentence.")
        End Using
    End Sub
End Module

' The example creates a file named "WriteTextAsync.txt" with the following contents:
' This is a sentence.
```

示例：使用文件类编写和追加文本

下面的示例演示如何使用 File 类将文本写入新文件并将新的文本行追加到同一文件。WriteAllText 和 AppendAllLines 方法会自动打开和关闭文件。如果提供给 WriteAllText 方法的路径已存在，则覆盖该文件。

```
using System;
using System.IO;

class Program
{
    static void Main(string[] args)
    {
        // Create a string with a line of text.
        string text = "First line" + Environment.NewLine;

        // Set a variable to the Documents path.
        string docPath = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments);

        // Write the text to a new file named "WriteFile.txt".
        File.WriteAllText(Path.Combine(docPath, "WriteFile.txt"), text);

        // Create a string array with the additional lines of text
        string[] lines = { "New line 1", "New line 2" };

        // Append new lines of text to the file
        File.AppendAllLines(Path.Combine(docPath, "WriteFile.txt"), lines);
    }
}
// The example creates a file named "WriteFile.txt" with the contents:
// First line
// And then appends the following contents:
// New line 1
// New line 2
```

```
Imports System.IO

Class WriteFile

    Public Shared Sub Main()

        ' Create a string array with the lines of text
        Dim text As String = "First line" & Environment.NewLine

        ' Set a variable to the Documents path.
        Dim docPath As String = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments)

        ' Write the text to a new file named "WriteFile.txt".
        File.WriteAllText(Path.Combine(docPath, Convert.ToString("WriteFile.txt")), text)

        ' Create a string array with the additional lines of text
        Dim lines() As String = {"New line 1", "New line 2"}

        ' Append new lines of text to the file
        File.AppendAllLines(Path.Combine(docPath, Convert.ToString("WriteFile.txt")), lines)

    End Sub

End Class

' The example creates a file named "WriteFile.txt" with the following contents:
' First line
' And then appends the following contents:
' New line 1
' New line 2
```

请参阅

- StreamWriter
- Path
- File.CreateText
- 如何：枚举目录和文件
- 如何：对新建的数据文件进行读取和写入
- 如何：打开并追加到日志文件
- 如何：从文件中读取文本
- 文件和流 I/O

如何：从文件中读取文本

2021/11/16 •

下面的示例演示如何使用适用于桌面应用的 .NET 以异步方式和同步方式从文本文件中读取文本。在这两个示例中，当你创建 StreamReader 类的实例时，你会提供文件的绝对路径或相对路径。

> **NOTE**
> 这些代码示例不适用于通用 Windows (UWP) 应用，因为 Windows 运行时提供了对文件进行读写操作的不同流类型。有关演示如何在 UWP 应用中读取文本的示例，请参阅快速入门：对文件执行读取和写入操作。有关演示如何在 .NET Framework 流和 Windows 运行时流之间进行转换的示例，请参阅如何：在 .NET Framework 流和 Windows 运行时流之间进行转换。

示例：控制台应用中的同步读取

以下示例演示控制台应用中的同步读取操作。此示例使用流读取器打开文本文件，将内容复制到字符串并将字符串输出到控制台。

> **IMPORTANT**
> 该示例假定名为 TestFile.txt 的文件已存在于与应用相同的文件夹中。

```
using System;
using System.IO;

class Program
{
    public static void Main()
    {
        try
        {
            // Open the text file using a stream reader.
            using (var sr = new StreamReader("TestFile.txt"))
            {
                // Read the stream as a string, and write the string to the console.
                Console.WriteLine(sr.ReadToEnd());
            }
        }
        catch (IOException e)
        {
            Console.WriteLine("The file could not be read:");
            Console.WriteLine(e.Message);
        }
    }
}
```

```
Imports System.IO

Module Program
    Public Sub Main()
        Try
            ' Open the file using a stream reader.
            Using sr As New StreamReader("TestFile.txt")
                ' Read the stream as a string and write the string to the console.
                Console.WriteLine(sr.ReadToEnd())
            End Using
        Catch e As IOException
            Console.WriteLine("The file could not be read:")
            Console.WriteLine(e.Message)
        End Try
    End Sub
End Module
```

示例：WPF 应用中的异步读取

以下示例演示 Windows Presentation Foundation (WPF) 应用中的异步读取操作。

> **IMPORTANT**
> 该示例假定名为 TestFile.txt 的文件已存在于与应用相同的文件夹中。

```
using System;
using System.IO;
using System.Windows;

namespace TextFiles
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void MainWindow_Loaded(object sender, RoutedEventArgs e)
        {
            try
            {
                using (var sr = new StreamReader("TestFile.txt"))
                {
                    ResultBlock.Text = await sr.ReadToEndAsync();
                }
            }
            catch (FileNotFoundException ex)
            {
                ResultBlock.Text = ex.Message;
            }
        }
    }
}
```

```
Imports System.IO
Imports System.Windows

''' <summary>
''' Interaction logic for MainWindow.xaml
''' </summary>

Partial Public Class MainWindow
    Inherits Window
    Public Sub New()
        InitializeComponent()
    End Sub

    Private Async Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs)
        Try
            Using sr As New StreamReader("TestFile.txt")
                ResultBlock.Text = Await sr.ReadToEndAsync()
            End Using
        Catch ex As FileNotFoundException
            ResultBlock.Text = ex.Message
        End Try
    End Sub
End Class
```

请参阅

- StreamReader
- File.OpenText
- StreamReader.ReadLine
- 异步文件 I/O
- 如何：创建目录列表
- 快速入门：读取和写入文件
- 如何：在 .NET Framework 流和 Windows 运行时流之间进行转换
- 如何：对新建的数据文件进行读取和写入
- 如何：打开并追加到日志文件
- 如何：将文本写入文件
- 如何：从字符串中读取字符
- 如何：向字符串写入字符
- 文件和流 I/O

如何：从字符串中读取字符

2021/11/16 •

下面的代码示例展示了如何从字符串中异步或同步读取字符。

示例：同步读取字符

此示例从字符串中同步读取 13 个字符，将它们存储到数组中，并显示这些字符。然后，示例将读取字符串中的剩余字符，将它们存储到数组中（从第六个元素开始），并显示数组的内容。

```
using System;
using System.IO;

public class CharsFromStr
{
    public static void Main()
    {
        string str = "Some number of characters";
        char[] b = new char[str.Length];

        using (StringReader sr = new StringReader(str))
        {
            // Read 13 characters from the string into the array.
            sr.Read(b, 0, 13);
            Console.WriteLine(b);

            // Read the rest of the string starting at the current string position.
            // Put in the array starting at the 6th array member.
            sr.Read(b, 5, str.Length - 13);
            Console.WriteLine(b);
        }
    }
}
// The example has the following output:
//
// Some number o
// Some f characters
```

```
Imports System.IO

Public Class CharsFromStr
    Public Shared Sub Main()
        Dim str As String = "Some number of characters"
        Dim b(str.Length - 1) As Char

        Using sr As StringReader = New StringReader(str)
            ' Read 13 characters from the string into the array.
            sr.Read(b, 0, 13)
            Console.WriteLine(b)

            ' Read the rest of the string starting at the current string position.
            ' Put in the array starting at the 6th array member.
            sr.Read(b, 5, str.Length - 13)
            Console.WriteLine(b)
        End Using
    End Sub
End Class
' The example has the following output:
'
' Some number o
' Some f characters
```

示例：异步读取字符

下一个示例是 WPF 应用背后的代码。在窗口加载时，示例从 TextBox 控件异步读取所有字符，并将其存储在数组中。随后，它以异步方式将每个字母或空格字符写入单独的 TextBlock 控件行。

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

```
Imports System.IO
Imports System.Text

''' <summary>
''' Interaction logic for MainWindow.xaml
''' </summary>

Partial Public Class MainWindow
    Inherits Window
    Public Sub New()
        InitializeComponent()
    End Sub
    Private Async Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
        Dim charsRead As Char() = New Char(UserInput.Text.Length) {}
        Using reader As StringReader = New StringReader(UserInput.Text)
            Await reader.ReadAsync(charsRead, 0, UserInput.Text.Length)
        End Using

        Dim reformattedText As StringBuilder = New StringBuilder()
        Using writer As StringWriter = New StringWriter(reformattedText)
            For Each c As Char In charsRead
                If Char.IsLetter(c) Or Char.IsWhiteSpace(c) Then
                    Await writer.WriteLineAsync(Char.ToLower(c))
                End If
            Next
        End Using
        Result.Text = reformattedText.ToString()
    End Sub
End Class
```

请参阅

- StringReader
- StringReader.Read
- 异步文件 I/O
- 如何：创建目录列表
- 如何：对新建的数据文件进行读取和写入
- 如何：打开并追加到日志文件
- 如何：从文件中读取文本
- 如何：将文本写入文件
- 如何：向字符串写入字符
- 文件和流 I/O

如何：向字符串写入字符

2021/11/16 •

下面的代码示例从字符数组以同步或异步方式向字符串写入字符。

示例：在控制台应用中以同步方式编写字符

下面的示例使用 StringWriter 将五个字符同步写入 StringBuilder 对象。

```
using System;
using System.IO;
using System.Text;

public class CharsToStr
{
    public static void Main()
    {
        StringBuilder sb = new StringBuilder("Start with a string and add from ");
        char[] b = { 'c', 'h', 'a', 'r', '.', ' ', 'B', 'u', 't', ' ', 'n', 'o', 't', ' ', 'a', 'l', 'l' };

        using (StringWriter sw = new StringWriter(sb))
        {
            // Write five characters from the array into the StringBuilder.
            sw.Write(b, 0, 5);
            Console.WriteLine(sb);
        }
    }
}
// The example has the following output:
//
// Start with a string and add from char.
```

```
Imports System.IO
Imports System.Text

Public Class CharsToStr
    Public Shared Sub Main()
        Dim sb As New StringBuilder("Start with a string and add from ")
        Dim b() As Char = {"c", "h", "a", "r", ".", " ", "B", "u", "t", " ", "n", "o", "t", " ", "a", "l",
"l"}

        Using sw As StringWriter = New StringWriter(sb)
            ' Write five characters from the array into the StringBuilder.
            sw.Write(b, 0, 5)
            Console.WriteLine(sb)
        End Using
    End Sub
End Class
' The example has the following output:
'
' Start with a string and add from char.
```

示例：在 WPF 应用中异步写入字符

下一个示例是 WPF 应用背后的代码。在窗口加载时，示例从 TextBox 控件异步读取所有字符，并将其存储在数组中。随后，它以异步方式将每个字母或空格字符写入单独的 TextBlock 控件行。

```
using System;
using System.Text;
using System.Windows;
using System.IO;

namespace StringReaderWriter
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Window_Loaded(object sender, RoutedEventArgs e)
        {
            char[] charsRead = new char[UserInput.Text.Length];
            using (StringReader reader = new StringReader(UserInput.Text))
            {
                await reader.ReadAsync(charsRead, 0, UserInput.Text.Length);
            }

            StringBuilder reformattedText = new StringBuilder();
            using (StringWriter writer = new StringWriter(reformattedText))
            {
                foreach (char c in charsRead)
                {
                    if (char.IsLetter(c) || char.IsWhiteSpace(c))
                    {
                        await writer.WriteLineAsync(char.ToLower(c));
                    }
                }
            }
            Result.Text = reformattedText.ToString();
        }
    }
}
```

```
Imports System.IO
Imports System.Text

''' <summary>
''' Interaction logic for MainWindow.xaml
''' </summary>

Partial Public Class MainWindow
    Inherits Window
    Public Sub New()
        InitializeComponent()
    End Sub
    Private Async Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
        Dim charsRead As Char() = New Char(UserInput.Text.Length) {}
        Using reader As StringReader = New StringReader(UserInput.Text)
            Await reader.ReadAsync(charsRead, 0, UserInput.Text.Length)
        End Using

        Dim reformattedText As StringBuilder = New StringBuilder()
        Using writer As StringWriter = New StringWriter(reformattedText)
            For Each c As Char In charsRead
                If Char.IsLetter(c) Or Char.IsWhiteSpace(c) Then
                    Await writer.WriteLineAsync(Char.ToLower(c))
                End If
            Next
        End Using
        Result.Text = reformattedText.ToString()
    End Sub
End Class
```

请参阅

- StringWriter
- StringWriter.Write
- StringBuilder
- 文件和流 I/O
- 异步文件 I/O
- 如何：枚举目录和文件
- 如何：对新建的数据文件进行读取和写入
- 如何：打开并追加到日志文件
- 如何：从文件中读取文本
- 如何：将文本写入文件
- 如何：从字符串中读取字符

如何：添加或删除访问控制列表条目（仅限 .NET Framework）

2021/11/16 •

若要向文件或目录添加或从文件或目录删除访问控制列表 (ACL) 条目，请从文件或目录获取 FileSecurity 或 DirectorySecurity 对象。修改对象，然后将其应用回文件或目录。

添加或从文件中删除 ACL 条目

1. 调用 File.GetAccessControl 方法以获取 FileSecurity 对象，该对象包含文件的当前 ACL 条目。
2. 添加或从步骤 1 返回的 FileSecurity 对象中删除 ACL 条目。
3. 将 FileSecurity 对象传递给 File.SetAccessControl 方法以应用更改。

添加或从目录中删除 ACL 条目

1. 调用 Directory.GetAccessControl 方法以获取 DirectorySecurity 对象，该对象包含目录的当前 ACL 条目。
2. 添加或从步骤 1 返回的 DirectorySecurity 对象中删除 ACL 条目。
3. 将 DirectorySecurity 对象传递给 Directory.SetAccessControl 方法以应用更改。

示例

你必须使用有效的用户或组帐户以运行此示例。此示例使用 File 对象。对 FileInfo、Directory 和 DirectoryInfo 类使用相同的过程。

```
using System;
using System.IO;
using System.Security.AccessControl;

namespace FileSystemExample
{
    class FileExample
    {
        public static void Main()
        {
            try
            {
                string fileName = "test.xml";

                Console.WriteLine("Adding access control entry for "
                    + fileName);

                // Add the access control entry to the file.
                AddFileSecurity(fileName, @"DomainName\AccountName",
                    FileSystemRights.ReadData, AccessControlType.Allow);

                Console.WriteLine("Removing access control entry from "
                    + fileName);

                // Remove the access control entry from the file.
                RemoveFileSecurity(fileName, @"DomainName\AccountName",
                    FileSystemRights.ReadData, AccessControlType.Allow);

                Console.WriteLine("Done.");
```

```
            }
            catch (Exception e)
            {
                Console.WriteLine(e);
            }
        }

        // Adds an ACL entry on the specified file for the specified account.
        public static void AddFileSecurity(string fileName, string account,
            FileSystemRights rights, AccessControlType controlType)
        {

            // Get a FileSecurity object that represents the
            // current security settings.
            FileSecurity fSecurity = File.GetAccessControl(fileName);

            // Add the FileSystemAccessRule to the security settings.
            fSecurity.AddAccessRule(new FileSystemAccessRule(account,
                rights, controlType));

            // Set the new access settings.
            File.SetAccessControl(fileName, fSecurity);
        }

        // Removes an ACL entry on the specified file for the specified account.
        public static void RemoveFileSecurity(string fileName, string account,
            FileSystemRights rights, AccessControlType controlType)
        {

            // Get a FileSecurity object that represents the
            // current security settings.
            FileSecurity fSecurity = File.GetAccessControl(fileName);

            // Remove the FileSystemAccessRule from the security settings.
            fSecurity.RemoveAccessRule(new FileSystemAccessRule(account,
                rights, controlType));

            // Set the new access settings.
            File.SetAccessControl(fileName, fSecurity);
        }
    }
}
```

```
Imports System.IO
Imports System.Security.AccessControl



Module FileExample

    Sub Main()
        Try
            Dim fileName As String = "test.xml"

            Console.WriteLine("Adding access control entry for " & fileName)

            ' Add the access control entry to the file.
            AddFileSecurity(fileName, "DomainName\AccountName", _
                FileSystemRights.ReadData, AccessControlType.Allow)

            Console.WriteLine("Removing access control entry from " & fileName)

            ' Remove the access control entry from the file.
            RemoveFileSecurity(fileName, "DomainName\AccountName", _
                FileSystemRights.ReadData, AccessControlType.Allow)

            Console.WriteLine("Done.")
```

```
        Catch e As Exception
            Console.WriteLine(e)
        End Try

    End Sub


    ' Adds an ACL entry on the specified file for the specified account.
    Sub AddFileSecurity(ByVal fileName As String, ByVal account As String, _
        ByVal rights As FileSystemRights, ByVal controlType As AccessControlType)

        ' Get a FileSecurity object that represents the
        ' current security settings.
        Dim fSecurity As FileSecurity = File.GetAccessControl(fileName)

        ' Add the FileSystemAccessRule to the security settings.
        Dim accessRule As FileSystemAccessRule = _
            New FileSystemAccessRule(account, rights, controlType)

        fSecurity.AddAccessRule(accessRule)

        ' Set the new access settings.
        File.SetAccessControl(fileName, fSecurity)

    End Sub


    ' Removes an ACL entry on the specified file for the specified account.
    Sub RemoveFileSecurity(ByVal fileName As String, ByVal account As String, _
        ByVal rights As FileSystemRights, ByVal controlType As AccessControlType)

        ' Get a FileSecurity object that represents the
        ' current security settings.
        Dim fSecurity As FileSecurity = File.GetAccessControl(fileName)

        ' Remove the FileSystemAccessRule from the security settings.
        fSecurity.RemoveAccessRule(New FileSystemAccessRule(account, _
            rights, controlType))

        ' Set the new access settings.
        File.SetAccessControl(fileName, fSecurity)

    End Sub
End Module
```

如何：压缩和解压缩文件

2021/11/16 •

System.IO.Compression 命名空间包含以下类型来对文件和流进行压缩或解压缩。还可以使用这些类型来读取和修改压缩文件的内容。

- ZipFile
- ZipArchive
- ZipArchiveEntry
- DeflateStream
- GZipStream

下面的示例演示使用压缩文件可以执行的某些操作。这些示例需要将以下 NuGet 包添加到项目中：

- System.IO.Compression
- System.IO.Compression.ZipFile

如果你使用的是 .NET Framework，请将对这两个库的引用添加到项目中：

- `System.IO.Compression`
- `System.IO.Compression.FileSystem`

示例 1：创建和提取 .zip 文件

以下示例演示如何使用 ZipFile 类创建和提取压缩的 .zip 文件。该示例将文件夹的内容压缩为一个新的 .zip 文件，然后将该 .zip 提取到一个新文件夹。

若要运行示例，请在程序文件夹中创建 start 文件夹，然后在其中放入要压缩的文件。

```
using System;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string startPath = @".\start";
        string zipPath = @".\result.zip";
        string extractPath = @".\extract";

        ZipFile.CreateFromDirectory(startPath, zipPath);

        ZipFile.ExtractToDirectory(zipPath, extractPath);
    }
}
```

```
Imports System.IO.Compression

Module Module1

    Sub Main()
        Dim startPath As String = ".\start"
        Dim zipPath As String = ".\result.zip"
        Dim extractPath As String = ".\extract"

        ZipFile.CreateFromDirectory(startPath, zipPath)

        ZipFile.ExtractToDirectory(zipPath, extractPath)
    End Sub

End Module
```

示例 2:提取特定文件扩展名

下一示例循环访问现有 .zip 文件的内容并提取扩展名为 .txt 的文件 。它使用 ZipArchive 类访问 zip，使用 ZipArchiveEntry 类检查各个条目。适用于 ExtractToFile 对象的扩展方法 ZipArchiveEntry 可以在 System.IO.Compression.ZipFileExtensions 类中使用。

若要运行示例，请将名为 result.zip 的 .zip 文件放到程序文件夹中 。出现提示时，提供要提取到的文件夹名称。

> **IMPORTANT**
> 在解压缩文件时，必须查找可以转义出你想要解压缩到的目录的恶意文件路径。这被称为"路径遍历攻击"。下面的示例演示如何检查恶意文件路径，并提供一种安全的解压缩方法。

```
using System;
using System.IO;
using System.IO.Compression;

class Program
{
    static void Main(string[] args)
    {
        string zipPath = @".\result.zip";

        Console.WriteLine("Provide path where to extract the zip file:");
        string extractPath = Console.ReadLine();

        // Normalizes the path.
        extractPath = Path.GetFullPath(extractPath);

        // Ensures that the last character on the extraction path
        // is the directory separator char.
        // Without this, a malicious zip file could try to traverse outside of the expected
        // extraction path.
        if (!extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(), StringComparison.Ordinal))
            extractPath += Path.DirectorySeparatorChar;

        using (ZipArchive archive = ZipFile.OpenRead(zipPath))
        {
            foreach (ZipArchiveEntry entry in archive.Entries)
            {
                if (entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase))
                {
                    // Gets the full path to ensure that relative segments are removed.
                    string destinationPath = Path.GetFullPath(Path.Combine(extractPath, entry.FullName));

                    // Ordinal match is safest, case-sensitive volumes can be mounted within volumes that
                    // are case-insensitive.
                    if (destinationPath.StartsWith(extractPath, StringComparison.Ordinal))
                        entry.ExtractToFile(destinationPath);
                }
            }
        }
    }
}
```

```
Imports System.IO
Imports System.IO.Compression

Module Module1

    Sub Main()
        Dim zipPath As String = ".\result.zip"

        Console.WriteLine("Provide path where to extract the zip file:")
        Dim extractPath As String = Console.ReadLine()

        ' Normalizes the path.
        extractPath = Path.GetFullPath(extractPath)

        ' Ensures that the last character on the extraction path
        ' is the directory separator char.
        ' Without this, a malicious zip file could try to traverse outside of the expected
        ' extraction path.
        If Not extractPath.EndsWith(Path.DirectorySeparatorChar.ToString(), StringComparison.Ordinal) Then
            extractPath += Path.DirectorySeparatorChar
        End If

        Using archive As ZipArchive = ZipFile.OpenRead(zipPath)
            For Each entry As ZipArchiveEntry In archive.Entries
                If entry.FullName.EndsWith(".txt", StringComparison.OrdinalIgnoreCase) Then

                    ' Gets the full path to ensure that relative segments are removed.
                    Dim destinationPath As String = Path.GetFullPath(Path.Combine(extractPath,
entry.FullName))

                    ' Ordinal match is safest, case-sensitive volumes can be mounted within volumes that
                    ' are case-insensitive.
                    If destinationPath.StartsWith(extractPath, StringComparison.Ordinal) Then
                        entry.ExtractToFile(destinationPath)
                    End If

                End If
            Next
        End Using
    End Sub

End Module
```

示例 3：将文件添加到现有 zip

以下示例使用 ZipArchive 类访问现有的 .zip 文件，然后向其添加文件。当将其添加到现有的 zip 时，会对新文件进行压缩。

```
using System;
using System.IO;
using System.IO.Compression;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            using (FileStream zipToOpen = new FileStream(@"c:\users\exampleuser\release.zip",
FileMode.Open))
            {
                using (ZipArchive archive = new ZipArchive(zipToOpen, ZipArchiveMode.Update))
                {
                    ZipArchiveEntry readmeEntry = archive.CreateEntry("Readme.txt");
                    using (StreamWriter writer = new StreamWriter(readmeEntry.Open()))
                    {
                            writer.WriteLine("Information about this package.");
                            writer.WriteLine("========================");
                    }
                }
            }
        }
    }
}
```

```
Imports System.IO
Imports System.IO.Compression

Module Module1

    Sub Main()
        Using zipToOpen As FileStream = New FileStream("c:\users\exampleuser\release.zip", FileMode.Open)
            Using archive As ZipArchive = New ZipArchive(zipToOpen, ZipArchiveMode.Update)
                Dim readmeEntry As ZipArchiveEntry = archive.CreateEntry("Readme.txt")
                Using writer As StreamWriter = New StreamWriter(readmeEntry.Open())
                    writer.WriteLine("Information about this package.")
                    writer.WriteLine("========================")
                End Using
            End Using
        End Using
    End Sub

End Module
```

示例 4：压缩和解压缩 .gz 文件

您也可以使用 GZipStream 和 DeflateStream 类压缩和解压缩数据。它们使用相同的压缩算法。可以使用许多常用工具解压缩写入 .gz 文件的 GZipStream 对象。以下示例展示了如何使用 GZipStream 类压缩和解压缩文件目录：

```
using System;
using System.IO;
using System.IO.Compression;

public class Program
{
    private static string directoryPath = @".\temp";
    public static void Main()
    {
        DirectoryInfo directorySelected = new DirectoryInfo(directoryPath);
        Compress(directorySelected);

        foreach (FileInfo fileToDecompress in directorySelected.GetFiles("*.gz"))
        {
            Decompress(fileToDecompress);
        }
    }

    public static void Compress(DirectoryInfo directorySelected)
    {
        foreach (FileInfo fileToCompress in directorySelected.GetFiles())
        {
            using (FileStream originalFileStream = fileToCompress.OpenRead())
            {
                if ((File.GetAttributes(fileToCompress.FullName) &
                   FileAttributes.Hidden) != FileAttributes.Hidden & fileToCompress.Extension != ".gz")
                {
                    using (FileStream compressedFileStream = File.Create(fileToCompress.FullName + ".gz"))
                    {
                        using (GZipStream compressionStream = new GZipStream(compressedFileStream,
                           CompressionMode.Compress))
                        {
                            originalFileStream.CopyTo(compressionStream);
                        }
                    }
                    FileInfo info = new FileInfo(directoryPath + Path.DirectorySeparatorChar +
fileToCompress.Name + ".gz");
                    Console.WriteLine($"Compressed {fileToCompress.Name} from
{fileToCompress.Length.ToString()} to {info.Length.ToString()} bytes.");
                }
            }
        }
    }

    public static void Decompress(FileInfo fileToDecompress)
    {
        using (FileStream originalFileStream = fileToDecompress.OpenRead())
        {
            string currentFileName = fileToDecompress.FullName;
            string newFileName = currentFileName.Remove(currentFileName.Length -
fileToDecompress.Extension.Length);

            using (FileStream decompressedFileStream = File.Create(newFileName))
            {
                using (GZipStream decompressionStream = new GZipStream(originalFileStream,
CompressionMode.Decompress))
                {
                    decompressionStream.CopyTo(decompressedFileStream);
                    Console.WriteLine($"Decompressed: {fileToDecompress.Name}");
                }
            }
        }
    }
}
```

```
Imports System.IO
Imports System.IO.Compression

Module Module1

    Private directoryPath As String = ".\temp"
    Public Sub Main()
        Dim directorySelected As New DirectoryInfo(directoryPath)
        Compress(directorySelected)

        For Each fileToDecompress As FileInfo In directorySelected.GetFiles("*.gz")
            Decompress(fileToDecompress)
        Next
    End Sub

    Public Sub Compress(directorySelected As DirectoryInfo)
        For Each fileToCompress As FileInfo In directorySelected.GetFiles()
            Using originalFileStream As FileStream = fileToCompress.OpenRead()
                If (File.GetAttributes(fileToCompress.FullName) And FileAttributes.Hidden) <>
FileAttributes.Hidden And fileToCompress.Extension <> ".gz" Then
                    Using compressedFileStream As FileStream = File.Create(fileToCompress.FullName & ".gz")
                        Using compressionStream As New GZipStream(compressedFileStream,
CompressionMode.Compress)

                            originalFileStream.CopyTo(compressionStream)
                        End Using
                    End Using
                    Dim info As New FileInfo(directoryPath & Path.DirectorySeparatorChar &
fileToCompress.Name & ".gz")
                    Console.WriteLine($"Compressed {fileToCompress.Name} from
{fileToCompress.Length.ToString()} to {info.Length.ToString()} bytes.")

                End If
            End Using
        Next
    End Sub


    Private Sub Decompress(ByVal fileToDecompress As FileInfo)
        Using originalFileStream As FileStream = fileToDecompress.OpenRead()
            Dim currentFileName As String = fileToDecompress.FullName
            Dim newFileName = currentFileName.Remove(currentFileName.Length -
fileToDecompress.Extension.Length)

            Using decompressedFileStream As FileStream = File.Create(newFileName)
                Using decompressionStream As GZipStream = New GZipStream(originalFileStream,
CompressionMode.Decompress)
                    decompressionStream.CopyTo(decompressedFileStream)
                    Console.WriteLine($"Decompressed: {fileToDecompress.Name}")
                End Using
            End Using
        End Using
    End Sub
End Module
```

请参阅

- ZipArchive
- ZipFile
- ZipArchiveEntry
- DeflateStream
- GZipStream
- 文件和流 I/O

撰写流

2021/11/16 •

后备存储是磁盘或内存等存储介质。各个后备存储都实现自己的流，作为 Stream 类的实现。

每个流类型对给定的后备存储执行字节读取和写入操作。连接到后备存储的流称为“基流”。基流的构造函数包含将流连接到后备存储所需的参数。例如，FileStream 包含指定路径参数的构造函数，此参数指定了进程如何共享文件。

System.IO 类旨在简化流撰写。可以将基流附加到一个或多个提供所需功能的直通流。可以将读取器或编写器附加到链的末尾，这样便能轻松读取或编写首选类型。

下面的代码示例根据现有 MyFile.txt 创建了 FileStream，以便缓冲 MyFile.txt。请注意，FileStreams 默认缓冲。

> **IMPORTANT**
> 这些示例假定名为 MyFile.txt 的文件已存在于与应用相同的文件夹中。

示例：使用 StreamReader

以下示例将创建从 FileStream 读取字符的 StreamReader，此读取器作为构造函数参数传递给 StreamReader。除非 StreamReader.Peek 找不到更多字符，否则 StreamReader.ReadLine 会一直读取字符。

```
using System;
using System.IO;

public class CompBuf
{
    private const string FILE_NAME = "MyFile.txt";

    public static void Main()
    {
        if (!File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} does not exist!");
            return;
        }
        FileStream fsIn = new FileStream(FILE_NAME, FileMode.Open,
            FileAccess.Read, FileShare.Read);
        // Create an instance of StreamReader that can read
        // characters from the FileStream.
        using (StreamReader sr = new StreamReader(fsIn))
        {
            string input;
            // While not at the end of the file, read lines from the file.
            while (sr.Peek() > -1)
            {
                input = sr.ReadLine();
                Console.WriteLine(input);
            }
        }
    }
}
```

```
Imports System.IO

Public Class CompBuf
    Private Const FILE_NAME As String = "MyFile.txt"

    Public Shared Sub Main()
        If Not File.Exists(FILE_NAME) Then
            Console.WriteLine($"{FILE_NAME} does not exist!")
            Return
        End If
        Dim fsIn As new FileStream(FILE_NAME, FileMode.Open, _
            FileAccess.Read, FileShare.Read)
        ' Create an instance of StreamReader that can read
        ' characters from the FileStream.
        Using sr As New StreamReader(fsIn)
            Dim input As String
            ' While not at the end of the file, read lines from the file.
            While sr.Peek() > -1
                input = sr.ReadLine()
                Console.WriteLine(input)
            End While
        End Using
    End Sub
End Class
```

示例：使用 BinaryReader

以下示例将创建从 FileStream 读取字节的 BinaryReader，此读取器作为构造函数参数传递给 BinaryReader 。除非 PeekChar 找不到更多字节，否则 ReadByte 会一直读取字节。

```
using System;
using System.IO;

public class ReadBuf
{
    private const string FILE_NAME = "MyFile.txt";

    public static void Main()
    {
        if (!File.Exists(FILE_NAME))
        {
            Console.WriteLine($"{FILE_NAME} does not exist.");
            return;
        }
        FileStream f = new FileStream(FILE_NAME, FileMode.Open,
            FileAccess.Read, FileShare.Read);
        // Create an instance of BinaryReader that can
        // read bytes from the FileStream.
        using (BinaryReader br = new BinaryReader(f))
        {
            byte input;
            // While not at the end of the file, read lines from the file.
            while (br.PeekChar() > -1 )
            {
                input = br.ReadByte();
                Console.WriteLine(input);
            }
        }
    }
}
```

```
Imports System.IO

Public Class ReadBuf
    Private Const FILE_NAME As String = "MyFile.txt"

    Public Shared Sub Main()
        If Not File.Exists(FILE_NAME) Then
            Console.WriteLine($"{FILE_NAME} does not exist.")
            Return
        End If
        Dim f As New FileStream(FILE_NAME, FileMode.Open, _
            FileAccess.Read, FileShare.Read)
        ' Create an instance of BinaryReader that can
        ' read bytes from the FileStream.
        Using br As new BinaryReader(f)
            Dim input As Byte
            ' While not at the end of the file, read lines from the file.
            While br.PeekChar() > -1
                input = br.ReadByte()
                Console.WriteLine(input)
            End While
        End Using
    End Sub
End Class
```

请参阅

- StreamReader
- StreamReader.ReadLine
- StreamReader.Peek
- FileStream
- BinaryReader
- BinaryReader.ReadByte
- BinaryReader.PeekChar

如何：在 .NET Framework 和 Windows 运行时流之间进行转换（仅限 Windows）

2021/11/16 •

适用于 UWP 应用的 .NET Framework 是完整的 .NET Framework 的子集。由于 UWP 应用的安全性和其他要求，你无法使用整套 .NET Framework API 来打开和读取文件。有关详细信息，请参阅适用于 UWP 应用的 .NET 概述。但是，你可能需要将 .NET Framework API 用于其他流处理操作。若要操作这些流，可以在 .NET Framework 流类型（如 MemoryStream 或 FileStream）和 Windows 运行时流（如 IInputStream、IOutputStream 或 IRandomAccessStream）之间进行转换。

System.IO.WindowsRuntimeStreamExtensions 类包含简化这些转换的方法。但是，.NET Framework 与 Windows 运行时流之间存在一些基本差异，这将影响使用这些方法所获得的结果，如以下部分中所述：

从 Windows 运行时流转换为 .NET Framework 流

若要从 Windows 运行时流转换为 .NET Framework 流，请使用以下 System.IO.WindowsRuntimeStreamExtensions 方法之一：

- WindowsRuntimeStreamExtensions.AsStream 将 Windows 运行时中的随机访问流转换为 .NET 中适用于 UWP 应用的托管流。
- WindowsRuntimeStreamExtensions.AsStreamForWrite 将 Windows 运行时中的输出流转换为 .NET 中为 UWP 应用的托管流。
- WindowsRuntimeStreamExtensions.AsStreamForRead 将 Windows 运行时中的输入流转换为 .NET 中为 UWP 应用的托管流。

Windows 运行时提供支持只读、只写或读写的流类型。如果将 Windows 运行时流转换为 .NET Framework 流，这些功能将保留。此外，如果你在 Windows 运行时流与 .NET Framework 流之间转换，则将取回原始 Windows 运行时实例。

最佳做法是使用与要转换的 Windows 运行时流的功能匹配的转换方法。但是，由于 IRandomAccessStream 是可读写的（它同时实现了 IOutputStream 和 IInputStream），因此转换方法将保留原始流的功能。例如，使用 WindowsRuntimeStreamExtensions.AsStreamForRead 转换 IRandomAccessStream 不会限制转换的 .NET Framework 流的可读性。它也是可写的。

示例：将 Windows 运行时随机访问流转换为 .NET Framework 流

若要从 Windows 运行时随机访问流转换为 .NET Framework 流，请使用 WindowsRuntimeStreamExtensions.AsStream 方法。

下面的代码示例会提示用户选择文件，使用 Windows 运行时 API 将其打开，然后将其转换为 .NET Framework 流。它可读取流，并将其输出为文本块。在输出结果之前，通常使用 .NET Framework API 操作流。

若要运行此示例，请创建包含一个名为 `TextBlock1` 的文本块和一个名为 `Button1` 的按钮的 UWP XAML 应用。将按钮单击事件与此示例中所示的 `button1_Click` 方法关联。

```
using System;
using System.IO;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Imaging;
using Windows.Storage;
using System.Net.Http;
using Windows.Storage.Pickers;

private async void button1_Click(object sender, RoutedEventArgs e)
{
    // Create a file picker.
    FileOpenPicker picker = new FileOpenPicker();

    // Set properties on the file picker such as start location and the type
    // of files to display.
    picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
    picker.ViewMode = PickerViewMode.List;
    picker.FileTypeFilter.Add(".txt");

    // Show picker enabling user to pick one file.
    StorageFile result = await picker.PickSingleFileAsync();

    if (result != null)
    {
        try
        {
            // Retrieve the stream. This method returns a IRandomAccessStreamWithContentType.
            var stream = await result.OpenReadAsync();

            // Convert the stream to a .NET stream using AsStream, pass to a
            // StreamReader and read the stream.
            using (StreamReader sr = new StreamReader(stream.AsStream()))
            {
                TextBlock1.Text = sr.ReadToEnd();
            }
        }
        catch (Exception ex)
        {
            TextBlock1.Text = "Error occurred reading the file. " + ex.Message;
        }
    }
    else
    {
        TextBlock1.Text = "User did not pick a file";
    }
}
```

```
Imports System.IO
Imports System.Runtime.InteropServices.WindowsRuntime
Imports Windows.UI.Xaml
Imports Windows.UI.Xaml.Controls
Imports Windows.UI.Xaml.Media.Imaging
Imports Windows.Storage
Imports System.Net.Http
Imports Windows.Storage.Pickers

Private Async Sub button1_Click(sender As Object, e As RoutedEventArgs)
    ' Create a file picker.
    Dim picker As New FileOpenPicker()

    ' Set properties on the file picker such as start location and the type of files to display.
    picker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary
    picker.ViewMode = PickerViewMode.List
    picker.FileTypeFilter.Add(".txt")

    ' Show picker that enable user to pick one file.
    Dim result As StorageFile = Await picker.PickSingleFileAsync()

    If result IsNot Nothing Then
        Try
            ' Retrieve the stream. This method returns a IRandomAccessStreamWithContentType.
            Dim stream = Await result.OpenReadAsync()

            ' Convert the stream to a .NET stream using AsStreamForRead, pass to a
            ' StreamReader and read the stream.
            Using sr As New StreamReader(stream.AsStream())
                TextBlock1.Text = sr.ReadToEnd()
            End Using
        Catch ex As Exception
            TextBlock1.Text = "Error occurred reading the file. " + ex.Message
        End Try
    Else
        TextBlock1.Text = "User did not pick a file"
    End If
End Sub
```

从 .NET Framework 转换为 Windows 运行时流

若要从.NET Framework 流转换为 Windows 运行时流，请使用下列任一 System.IO.WindowsRuntimeStreamExtensions 方法：

- WindowsRuntimeStreamExtensions.AsInputStream 将 .NET 中适用于为 UWP 应用的托管流转换为 Windows 运行时中的输入流。
- WindowsRuntimeStreamExtensions.AsOutputStream 将 .NET 中适用于 UWP 应用的托管流转换为 Windows 运行时中的输出流。
- WindowsRuntimeStreamExtensions.AsRandomAccessStream 将适用于 UWP 应用的 .NET 中的托管流转换为 Windows 运行时可用于读取或写入的随机访问流。

在将 .NET Framework 流转换为 Windows 运行时流时，转换后的流的功能将取决于原始流。例如，如果原始流支持读取和写入，并且你调用 WindowsRuntimeStreamExtensions.AsInputStream 来转换流，则返回的类型为 `IRandomAccessStream`。 `IRandomAccessStream` 可实现 `IInputStream` 和 `IOutputStream`，且支持读取和写入。

.NET Framework 流不支持克隆，即使转换后也是如此。如果将 .NET Framework 流转换为 Windows 运行时流，并调用 GetInputStreamAt 或 GetOutputStreamAt(这会直接调用 CloneStream 或 CloneStream)，则会引发异常。

示例：将 .NET Framework 转换为 Windows 运行时随机访问流

要从 .NET Framework 流转换为 Windows 运行时随机访问流，请使用 AsRandomAccessStream 方法，如以下示例所示：

> **IMPORTANT**
> 确保所使用的 .NET Framework 流支持查找或将其复制到支持查找的流。可使用 Stream.CanSeek 属性来确定这一点。

若要运行此示例，请创建一个面向 .NET Framework 4.5.1 的且包含一个名为 `TextBlock2` 的文本块和一个名为 `Button2` 的按钮的 UWP XAML 应用。将按钮单击事件与此示例中所示的 `button2_Click` 方法关联。

```
using System;
using System.IO;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Imaging;
using Windows.Storage;
using System.Net.Http;
using Windows.Storage.Pickers;

private async void button2_Click(object sender, RoutedEventArgs e)
{
    // Create an HttpClient and access an image as a stream.
    var client = new HttpClient();
    Stream stream = await client.GetStreamAsync("https://docs.microsoft.com/en-
us/dotnet/images/hub/featured-1.png");
    // Create a .NET memory stream.
    var memStream = new MemoryStream();
    // Convert the stream to the memory stream, because a memory stream supports seeking.
    await stream.CopyToAsync(memStream);
    // Set the start position.
    memStream.Position = 0;
    // Create a new bitmap image.
    var bitmap = new BitmapImage();
    // Set the bitmap source to the stream, which is converted to a IRandomAccessStream.
    bitmap.SetSource(memStream.AsRandomAccessStream());
    // Set the image control source to the bitmap.
    image1.Source = bitmap;
}
```

```
Imports System.IO
Imports System.Runtime.InteropServices.WindowsRuntime
Imports Windows.UI.Xaml
Imports Windows.UI.Xaml.Controls
Imports Windows.UI.Xaml.Media.Imaging
Imports Windows.Storage
Imports System.Net.Http
Imports Windows.Storage.Pickers

Private Async Sub button2_Click(sender As Object, e As RoutedEventArgs)


    ' Create an HttpClient and access an image as a stream.
    Dim client = New HttpClient()
    Dim stream As Stream = Await client.GetStreamAsync("https://docs.microsoft.com/en-
us/dotnet/images/hub/featured-1.png")
    ' Create a .NET memory stream.
    Dim memStream = New MemoryStream()

    ' Convert the stream to the memory stream, because a memory stream supports seeking.
    Await stream.CopyToAsync(memStream)

    ' Set the start position.
    memStream.Position = 0

    ' Create a new bitmap image.
    Dim bitmap = New BitmapImage()

    ' Set the bitmap source to the stream, which is converted to a IRandomAccessStream.
    bitmap.SetSource(memStream.AsRandomAccessStream())

    ' Set the image control source to the bitmap.
    image1.Source = bitmap
End Sub
```

请参阅

- 快速入门：读取和写入文件 (Windows)
- 适用于 Microsoft Store 应用的 .NET 概述
- 适用于 Windows 应用商店应用的 .NET API

异步文件 I/O

2021/11/16 •

异步操作使您能在不阻塞主线程的情况下执行占用大量资源的 I/O 操作。在 Windows 8.x 应用商店应用或桌面应用中一个耗时的流操作可能阻塞 UI 线程并让应用看起来好像不工作时，这种性能的考虑就显得尤为重要了。

从 .NET Framework 4.5 开始，I/O 类型包括了异步方法，以简化异步操作。异步方法在其名称中包括 `Async`，例如 ReadAsync、WriteAsync、CopyToAsync、FlushAsync、ReadLineAsync和 ReadToEndAsync。这些异步方法基于流类（例如 Stream、FileStream和 MemoryStream）和用来向流中读出或写入数据的类（例如 TextReader 和 TextWriter）实现。

在 .NET Framework 4 和更早的版本中，你必须使用 BeginRead 和 EndRead 等方法来实现异步 I/O 操作。这些方法仍然在当前 .NET 版本中可用，从而支持传统的代码；但是，异步方法能帮助你更轻松地实现异步 I/O 操作。

C# 和 Visual Basic 分别具有两个用于异步编程的关键字：

- `Async` (Visual Basic) 或 `async` (C#) 修饰符，您可以用来标记包含异步操作的方法。
- `Await` (Visual Basic) 或 `await` (C#) 运算符，可以应用到异步方法的结果中。

如下面的示例所示，若要实现异步 I/O 操作，请把这些关键字和异步方法结合使用。有关详细信息，请参阅使用 async 和 await 的异步编程 (C#) 或 使用 Async 和 Await 的异步编程 (Visual Basic)。

下面的示例演示如何使用两个 FileStream 对象把文件从一个目录异步复制到另一个目录。需要注意 Click 控件的 Button 事件处理程序具有 `async` 修饰符标记，因为它调用异步方法。

```
using System;
using System.Threading.Tasks;
using System.Windows;
using System.IO;

namespace WpfApplication
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private async void Button_Click(object sender, RoutedEventArgs e)
        {
            string StartDirectory = @"c:\Users\exampleuser\start";
            string EndDirectory = @"c:\Users\exampleuser\end";

            foreach (string filename in Directory.EnumerateFiles(StartDirectory))
            {
                using (FileStream SourceStream = File.Open(filename, FileMode.Open))
                {
                    using (FileStream DestinationStream = File.Create(EndDirectory +
filename.Substring(filename.LastIndexOf('\\'))))
                    {
                        await SourceStream.CopyToAsync(DestinationStream);
                    }
                }
            }
        }
    }
}
```

```
Imports System.IO

Class MainWindow

    Private Async Sub Button_Click(sender As Object, e As RoutedEventArgs)
        Dim StartDirectory As String = "c:\Users\exampleuser\start"
        Dim EndDirectory As String = "c:\Users\exampleuser\end"

        For Each filename As String In Directory.EnumerateFiles(StartDirectory)
            Using SourceStream As FileStream = File.Open(filename, FileMode.Open)
                Using DestinationStream As FileStream = File.Create(EndDirectory +
filename.Substring(filename.LastIndexOf("\"c)))
                    Await SourceStream.CopyToAsync(DestinationStream)
                End Using

            End Using
        Next
    End Sub

End Class
```

下一个例子类似于前面的例子，但是使用 StreamReader 和 StreamWriter 对象以异步方式读取和写入文本文件的内容。

```
private async void Button_Click(object sender, RoutedEventArgs e)
{
    string UserDirectory = @"c:\Users\exampleuser\";

    using (StreamReader SourceReader = File.OpenText(UserDirectory + "BigFile.txt"))
    {
        using (StreamWriter DestinationWriter = File.CreateText(UserDirectory + "CopiedFile.txt"))
        {
            await CopyFilesAsync(SourceReader, DestinationWriter);
        }
    }
}

public async Task CopyFilesAsync(StreamReader Source, StreamWriter Destination)
{
    char[] buffer = new char[0x1000];
    int numRead;
    while ((numRead = await Source.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        await Destination.WriteAsync(buffer, 0, numRead);
    }
}
```

```
Private Async Sub Button_Click(sender As Object, e As RoutedEventArgs)
    Dim UserDirectory As String = "c:\Users\exampleuser\"

    Using SourceReader As StreamReader = File.OpenText(UserDirectory + "BigFile.txt")
        Using DestinationWriter As StreamWriter = File.CreateText(UserDirectory + "CopiedFile.txt")
            Await CopyFilesAsync(SourceReader, DestinationWriter)
        End Using
    End Using
End Sub

Public Async Function CopyFilesAsync(Source As StreamReader, Destination As StreamWriter) As Task
    Dim buffer(4095) As Char
    Dim numRead As Integer

    numRead = Await Source.ReadAsync(buffer, 0, buffer.Length)
    Do While numRead <> 0
        Await Destination.WriteAsync(buffer, 0, numRead)
        numRead = Await Source.ReadAsync(buffer, 0, buffer.Length)
    Loop

End Function
```

下一个示例演示用于在 Windows 8.x 应用商店应用中以 Stream 的形式打开文件的代码隐藏文件和 XAML 文件，并且通过使用 StreamReader 类的实例来读取其内容。它使用异步方法以流的形式打开文件并读取其内容。

```
using System;
using System.IO;
using System.Text;
using Windows.Storage.Pickers;
using Windows.Storage;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace ExampleApplication
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
        }

        private async void Button_Click_1(object sender, RoutedEventArgs e)
        {
            StringBuilder contents = new StringBuilder();
            string nextLine;
            int lineCounter = 1;

            var openPicker = new FileOpenPicker();
            openPicker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary;
            openPicker.FileTypeFilter.Add(".txt");
            StorageFile selectedFile = await openPicker.PickSingleFileAsync();

            using (StreamReader reader = new StreamReader(await selectedFile.OpenStreamForReadAsync()))
            {
                while ((nextLine = await reader.ReadLineAsync()) != null)
                {
                    contents.AppendFormat("{0}. ", lineCounter);
                    contents.Append(nextLine);
                    contents.AppendLine();
                    lineCounter++;
                    if (lineCounter > 3)
                    {
                        contents.AppendLine("Only first 3 lines shown.");
                        break;
                    }
                }
            }
            DisplayContentsBlock.Text = contents.ToString();
        }
    }
}
```

```
Imports System.Text
Imports System.IO
Imports Windows.Storage.Pickers
Imports Windows.Storage

NotInheritable Public Class BlankPage
    Inherits Page



    Private Async Sub Button_Click_1(sender As Object, e As RoutedEventArgs)
        Dim contents As StringBuilder = New StringBuilder()
        Dim nextLine As String
        Dim lineCounter As Integer = 1

        Dim openPicker = New FileOpenPicker()
        openPicker.SuggestedStartLocation = PickerLocationId.DocumentsLibrary

        openPicker.FileTypeFilter.Add(".txt")
        Dim selectedFile As StorageFile = Await openPicker.PickSingleFileAsync()

        Using reader As StreamReader = New StreamReader(Await selectedFile.OpenStreamForReadAsync())
            nextLine = Await reader.ReadLineAsync()
            While (nextLine <> Nothing)
                contents.AppendFormat("{0}. ", lineCounter)
                contents.Append(nextLine)
                contents.AppendLine()
                lineCounter = lineCounter + 1
                If (lineCounter > 3) Then
                    contents.AppendLine("Only first 3 lines shown.")
                    Exit While
                End If
                nextLine = Await reader.ReadLineAsync()
            End While
        End Using
        DisplayContentsBlock.Text = contents.ToString()
    End Sub
End Class
```

```
<Page
    x:Class="ExampleApplication.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ExampleApplication"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <StackPanel Background="{StaticResource ApplicationPageBackgroundBrush}" VerticalAlignment="Center"
HorizontalAlignment="Center">
        <TextBlock Text="Display lines from a file."></TextBlock>
        <Button Content="Load File" Click="Button_Click_1"></Button>
        <TextBlock Name="DisplayContentsBlock"></TextBlock>
    </StackPanel>
</Page>
```

请参阅

- Stream
- 文件和流 I/O
- 使用 Async 和 Await 的异步编程 (C#)
- 使用 Async 和 Await 的异步编程 (Visual Basic)

处理 .NET 中的 I/O 错误

2021/11/16 •

除了可能在任何方法调用中引发异常(如系统压力过大导致 OutOfMemoryException 或由于编程器错误导致 NullReferenceException),.NET 文件系统方法还可能引发以下异常:

- System.IO.IOException,所有 System.IO 异常类型的基类。当出现操作系统的返回代码不直接映射到任何其他异常类型的错误时将引发此异常。
- System.IO.FileNotFoundException。
- System.IO.DirectoryNotFoundException。
- DriveNotFoundException。
- System.IO.PathTooLongException。
- System.OperationCanceledException。
- System.UnauthorizedAccessException。
- 当 .NET Framework 和 .NET Core 2.0 以及先前版本上的路径字符无效时,将引发 System.ArgumentException。
- 当 .NET Framework 中存在无效冒号时,将引发 System.NotSupportedException。
- 当在受限制的信任中运行的应用程序缺少仅限于 .NET Framework 的所需权限时,将引发 System.Security.SecurityException。(完全信任是 .NET Framework 的默认设置。)

将错误代码映射到异常

由于文件系统为操作系统资源,.NET Core 和 .NET Framework 中的 I/O 方法将包装对基础操作系统的调用。当由操作系统执行的代码出现 I/O 错误时,操作系统将对 .NET I/O 方法返回错误信息。然后,该方法会将错误信息(通常采用错误代码形式)转换为 .NET 异常类型。大多数情况下,可以通过直接将错误代码转换为其相应异常类型来完成此操作;它不基于方法调用的上下文执行任何特殊的错误映射。

例如,在 Windows 操作系统,返回 `ERROR_FILE_NOT_FOUND`(或 0x02)错误代码的方法调用会映射到 FileNotFoundException,`ERROR_PATH_NOT_FOUND` 错误代码(或 0x03)则映射到 DirectoryNotFoundException。

但是,操作系统返回特定错误代码的精确条件通常未记录或记录不当。因此,会出现意外异常。例如,因为使用的是目录而不是文件,可以预料到向 DirectoryInfo 构造函数提供无效目录路径将引发 DirectoryNotFoundException。但是,它也可能引发 FileNotFoundException。

I/O 操作中的异常处理

由于操作系统的这一依赖性,相同异常条件(例如在我们的示例中没有发现错误目录)可能会导致 I/O 方法引发任何一种 I/O 异常。这意味着,在调用 I/O API 时,代码应准备好处理大多数或者所有这些异常,如下表所示:

	.NET CORE/.NET 5+	.NET FRAMEWORK
IOException	是	是
FileNotFoundException	是	是
DirectoryNotFoundException	是	是
DriveNotFoundException	是	是

例外	.NET Core/.NET 5+	.NET Framework
PathTooLongException	是	是
OperationCanceledException	是	是
UnauthorizedAccessException	是	是
ArgumentException	.NET core 2.0 及早期版本	是
NotSupportedException	No	是
SecurityException	否	仅受限的信任

处理 IOException

作为 System.IO 命名空间中异常的基类，当任何错误代码未映射到预定义的异常类型时，也将引发 IOException。这意味着异常可以由任何 I/O 操作引发。

> **IMPORTANT**
> 因为 IOException 是 System.IO 命名空间中其他异常类型的基类，应在处理其他 I/O 相关异常后处理 `catch` 块。

此外，从 .NET Core 2.1 开始，已删除对路径正确性（例如，为了确保路径中不存在无效字符）的验证检查，且运行时会引发从操作系统错误代码（而非从它自己的验证代码）映射的异常。在这种情况下，最有可能引发的异常是 IOException，虽然也可能引发任何其他异常类型。

请注意，在异常处理代码中，应始终最后处理 IOException。否则，因为它是所有其他 IO 异常的基类，将不会评估派生类的 catch 块。

在 IOException 情况下，可以从 IOException.HResult 属性获取更多错误信息。若要将 HResult 值转换为 Win32 错误代码，可以删除 32 位值的前 16 位。下表列出了可能包装在 IOException 中的错误代码。

HRESULT	十六进制	十进制
ERROR_SHARING_VIOLATION	32	缺少文件名称，或文件或目录正在使用中。
ERROR_FILE_EXISTS	80	该文件已存在。
ERROR_INVALID_PARAMETER	87	提供给该方法的参数无效。
ERROR_ALREADY_EXISTS	183	文件或目录已存在。

可以使用 catch 语句中的 `When` 子句来处理这些问题，如以下示例所示。

```
using System;
using System.IO;
using System.Text;

class Program
{
    static void Main()
    {
        var sw = OpenStream(@".\textfile.txt");
        if (sw is null)
            return;
        sw.WriteLine("This is the first line.");
        sw.WriteLine("This is the second line.");
        sw.Close();
    }

   static StreamWriter OpenStream(string path)
   {
        if (path is null) {
            Console.WriteLine("You did not supply a file path.");
            return null;
        }

        try {
            var fs = new FileStream(path, FileMode.CreateNew);
            return new StreamWriter(fs);
        }
        catch (FileNotFoundException) {
            Console.WriteLine("The file or directory cannot be found.");
        }
        catch (DirectoryNotFoundException) {
            Console.WriteLine("The file or directory cannot be found.");
        }
        catch (DriveNotFoundException) {
            Console.WriteLine("The drive specified in 'path' is invalid.");
        }
        catch (PathTooLongException) {
            Console.WriteLine("'path' exceeds the maxium supported path length.");
        }
        catch (UnauthorizedAccessException) {
            Console.WriteLine("You do not have permission to create this file.");
        }
        catch (IOException e) when ((e.HResult & 0x0000FFFF) == 32 ) {
            Console.WriteLine("There is a sharing violation.");
        }
        catch (IOException e) when ((e.HResult & 0x0000FFFF) == 80) {
            Console.WriteLine("The file already exists.");
        }
        catch (IOException e) {
            Console.WriteLine($"An exception occurred:\nError code: " +
                              $"{e.HResult & 0x0000FFFF}\nMessage: {e.Message}");
        }
        return null;
    }
}
```

```
Imports System.IO

Module Program
    Sub Main(args As String())
        Dim sw = OpenStream(".\textfile.txt")
        If sw Is Nothing Then Return

        sw.WriteLine("This is the first line.")
        sw.WriteLine("This is the second line.")
        sw.Close()
    End Sub

    Function OpenStream(path As String) As StreamWriter
        If path Is Nothing Then
            Console.WriteLine("You did not supply a file path.")
            Return Nothing
        End If

        Try
            Dim fs As New FileStream(path, FileMode.CreateNew)
            Return New StreamWriter(fs)
        Catch e As FileNotFoundException
            Console.WriteLine("The file or directory cannot be found.")
        Catch e As DirectoryNotFoundException
            Console.WriteLine("The file or directory cannot be found.")
        Catch e As DriveNotFoundException
            Console.WriteLine("The drive specified in 'path' is invalid.")
        Catch e As PathTooLongException
            Console.WriteLine("'path' exceeds the maxium supported path length.")
        Catch e As UnauthorizedAccessException
            Console.WriteLine("You do not have permission to create this file.")
        Catch e As IOException When (e.HResult And &h0000FFFF) = 32
            Console.WriteLine("There is a sharing violation.")
        Catch e As IOException When (e.HResult And &h0000FFFF) = 80
            Console.WriteLine("The file already exists.")
        Catch e As IOException
            Console.WriteLine($"An exception occurred:{vbCrLf}Error code: " +
                              $"{e.HResult And &h0000FFFF}{vbCrLf}Message: {e.Message}")
        End Try
        Return Nothing
    End Function
End Module
```

请参阅

- 在 .NET 中处理和引发异常
- 异常处理(任务并行库)
- 针对异常的最佳做法
- 如何在 catch 块中使用特定异常

独立存储

2021/11/16 •

对于桌面应用，独立存储是一种数据存储机制，它定义了将代码与保存的数据关联的标准化方式，从而提供隔离性和安全性。同时，标准化也提供了其他好处。管理员可以使用旨在操作独立存储的工具来配置文件存储空间、设置安全策略及删除未使用的数据。通过独立存储，代码不再需要使用唯一的路径来指定文件系统中的安全位置，同时可以保护数据免遭只具有独立存储访问权限的其他应用程序的损坏。不再需要指示应用程序的存储区域位置的硬编码信息。

> **IMPORTANT**
> 独立存储不适用于 Windows 8.x 应用商店应用。请改用 Windows 运行时 API 包含的 `Windows.Storage` 命名空间中的应用程序数据类来存储本地数据和文件。有关详细信息，请参阅 Windows 开发人员中心的 应用程序数据 。

数据隔离舱和存储区

当应用程序在文件中存储数据时，必须仔细选择文件名和存储位置，最大程度地减小其他应用程序知道该存储位置的可能性，从而使数据不易受到损坏。如果没有标准系统来管理这些问题，想临时开发最大程度地减少存储冲突的技术可能并非易事，而且开发出来的技术也不见得可靠。

通过使用独立存储，数据将始终按用户和程序集进行隔离。程序集的源或强名称等凭据确定程序集的身份。通过使用类似的凭据，数据还可以按应用程序域进行隔离。

当使用独立存储时，应用程序将数据保存到与代码标识的某些方面（例如，其发行者或签名）关联的独特数据隔离舱。数据隔离舱是一个抽象的存储位置，而不是具体的存储位置，它由一个或多个独立的存储文件（叫做存储区）组成，这些独立的存储文件包含存储数据的实际目录位置。例如，应用程序可能有一个与其关联的数据隔离舱，文件系统中的某个目录将实现实际保留应用程序数据的存储区。保存在存储区中的数据可以是任意类型的数据，无论是用户首选项信息还是应用程序状态都可以。对于开发人员来说，数据隔离舱的位置是透明的。应用商店通常位于客户端，但是，服务器应用程序可以使用独立存储通过模拟该服务的用户存储信息。独立存储还可以将信息和用户漫游配置文件一起存储在服务器上，这样，漫游用户就可以随时使用该信息。

独立存储的配额

配额是对可使用的独立存储数量的限制。配额包括文件空间的字节及与存储区中目录和其他信息关联的系统开销。独立存储使用权限配额，这些配额是使用 IsolatedStoragePermission 对象设置的存储限制。如果尝试写入的数据超出配额，则会引发 IsolatedStorageException 异常。安全策略确定向代码授予的权限，它可以使用 .NET Framework 配置工具 (Mscorcfg.msc) 来修改。已授予 IsolatedStoragePermission 的代码所使用的存储范围不能超过 UserQuota 属性的限制。但是，由于代码可以通过表示不同的用户标识绕过权限配额，所以权限配额用作指导代码如何工作的指南，而不是对代码行为的硬性限制。

不对漫游存储区强制执行配额。因此，对使用它们的代码要求稍高级别的权限。枚举值 AssemblyIsolationByRoamingUser 和 DomainIsolationByRoamingUser 为漫游用户指定使用独立存储的权限。

安全访问

通过使用独立存储，可以使部分受信任的应用程序以由计算机安全策略控制的方式存储数据。对于用户需慎重运行的下载的组件来说，这尤为有用。在使用标准 I/O 机制访问文件系统时，安全策略很少向这种代码授予权限。但是默认情况下，会对在本地计算机、本地网络或 Internet 中运行的代码授予使用独立存储的权限。

管理员可以根据适当的信任级别限制应用程序或用户可以使用多少独立存储。另外，管理员可以完全移除用户的持久性数据。若要创建或访问独立存储，则必须授予代码相应的 IsolatedStorageFilePermission 权限。

要访问独立存储，代码必须具有所有必要的本机平台操作系统权限。必须满足用来控制哪些用户有权使用文件系统的访问控制列表 (ACL)。除非执行(特定于平台的)模拟，否则 .NET 应用程序已经具有访问独立存储的操作系统权限。在这种情况下，应用程序负责确保被模拟的用户标识具有访问独立存储的适当操作系统权限。对于在 Web 上运行或从 Web 下载的代码而言，这种访问为之提供了一种对与特定用户相关的存储区域进行读写操作的简便方法。

为了控制对独立存储的访问，公共语言运行时使用 IsolatedStorageFilePermission 对象。每个对象都具有指定以下值的属性：

- 允许的用法，这指出了所允许的访问类型。这些值是 IsolatedStorageContainment 枚举的成员。有关这些值的更多信息，请参见下一节中的表。
- 存储配额(如上一节所述)。

当代码第一次尝试打开存储时，运行时要求 IsolatedStorageFilePermission 权限。它根据代码的受信任程度决定是否授予此权限。如果授予此权限，则允许的用法和存储配额值由安全策略和代码对 IsolatedStorageFilePermission的请求决定。安全策略使用 .NET Framework 配置工具 (Mscorcfg.msc) 来进行设置。检查调用堆栈中的所有调用方以确保每个调用方至少具有适当的允许的用法。运行时还检查强加于代码的配额，该代码打开或创建将在其中保存文件的存储区。如果满足这些条件，就授予权限。每次文件写入存储区时，都将再次检查配额。

因为公共语言运行时将根据安全策略授予任何适当的 IsolatedStorageFilePermission，所以请求权限不需要应用程序代码。然而，有很好的理由来请求应用程序需要的特定权限，包括 IsolatedStorageFilePermission。

允许的用法和安全风险

IsolatedStorageFilePermission 指定的允许的用法确定允许代码创建和使用独立存储的程度。下表显示了权限中指定的允许的用法如何与隔离的类型对应，并总结了与每种允许的用法关联的安全风险。

None	不允许使用任何独立存储。	没有安全影响。
DomainIsolationByUser	按用户、域和程序集隔离。每个程序集在域中都有单独的子存储区。使用此权限的存储也由计算机隐式隔离。	此权限级别无法阻止他人未经授权滥用资源，尽管强制的配额对此做法增添了一些难度。这叫做拒绝服务攻击。
DomainIsolationByRoamingUser	与 `DomainIsolationByUser` 相同，但如果启用漫游用户配置文件且不强制配额，则存储将保存到将漫游的位置。	因为必须禁用配额，所以存储资源更易受到拒绝服务攻击。
AssemblyIsolationByUser	按用户和程序集隔离。使用此权限的存储也由计算机隐式隔离。	在此级别强制实施配额以帮助防止拒绝服务攻击。由于另一个域中相同的程序集可以访问该存储区，这就使信息可能在应用程序间泄露。
AssemblyIsolationByRoamingUser	与 `AssemblyIsolationByUser` 相同，但如果启用漫游用户配置文件且不强制配额，则存储将保存到将漫游的位置。	与 `AssemblyIsolationByUser` 中相同，但没有配额，增加了拒绝服务攻击的风险。

[illegible]	[illegible]	[illegible]
AdministerIsolatedStorageByUser	按用户隔离。通常，只有管理或调试工具才使用此级别的权限。	使用该权限访问允许代码查看或删除任何的用户独立存储文件或目录(而不论程序集是否隔离)。存在的风险包括(但不限于)泄露信息和数据丢失。
UnrestrictedIsolatedStorage	按所有用户、域和程序集隔离。通常，只有管理或调试工具才使用此级别的权限。	此权限有可能会整个危害所有用户的所有独立存储区。

与不受信任的数据相关的独立存储组件的安全性

本节适用于以下框架：

- .NET Framework(所有版本)
- .NET Core 2.1+
- .NET 5.0+

.NET Framework 和 .NET Core 提供独立存储作为一种为用户、应用程序或组件保留数据的机制。这是一个旧组件，主要用于现已弃用的代码访问安全性方案。

各种独立存储 API 和工具可用于跨信任边界读取数据。例如，从计算机范围的作用域中读取数据会从计算机上其他可能不太受信任的用户帐户聚合数据。从计算机范围的独立存储作用域读取的组件或应用程序应了解读取此数据的后果。

可从计算机范围作用域读取的安全敏感 API

调用以下任意 API 的组件或应用程序从计算机范围的作用域中读取：

- IsolatedStorageFile.GetEnumerator，传递包含 IsolatedStorageScope.Machine 标志的作用域
- IsolatedStorageFile.GetMachineStoreForApplication
- IsolatedStorageFile.GetMachineStoreForAssembly
- IsolatedStorageFile.GetMachineStoreForDomain
- IsolatedStorageFile.GetStore，传递包含 IsolatedStorageScope.Machine 标志的作用域
- IsolatedStorageFile.Remove，传递包含 `IsolatedStorageScope.Machine` 标志的作用域

如果通过 `/machine` 开关调用，则会影响独立存储工具 `storeadm.exe`，如以下代码所示：

```
storeadm.exe /machine [any-other-switches]
```

Visual Studio 和 .NET Framework SDK 包含独立存储工具。

如果应用程序不涉及调用上述 API，或者工作流不涉及以这种方式调用 `storeadm.exe`，则不会应用此文档。

多用户环境中的影响

如前所述，从一个信任环境写入的数据产生的这些 API 安全影响可从不同的信任环境中读取。独立存储通常使用三个位置中的一个来读取和写入数据：

1. `%LOCALAPPDATA%\IsolatedStorage\`：例如，`User` 范围的 `C:\Users\<username>\AppData\Local\IsolatedStorage\`。
2. `%APPDATA%\IsolatedStorage\`：例如，`User|Roaming` 范围的 `C:\Users\<username>\AppData\Roaming\IsolatedStorage\`。
3. `%PROGRAMDATA%\IsolatedStorage\`：例如，`Machine` 范围的 `C:\ProgramData\IsolatedStorage\`。

每个用户的前两个位置都是独立的。Windows 可确保同一计算机上的不同用户帐户无法访问彼此的用户配置文

件文件夹。使用 `User` 或 `User|Roaming` 存储区的两个不同用户帐户不会看到彼此的数据，并且不会干扰彼此的数据。

第三个位置在计算机上的所有用户帐户之间共享。不同的帐户可以从此位置读取数据以及将数据写入此位置，并且可以查看彼此的数据。

上述路径可能因使用的 Windows 版本而异。

现在，假设有一个多用户系统，其中有两个注册用户 Mallory 和 Bob 。Mallory 能够访问用户配置文件目录 `C:\Users\Mallory\`，并且可以访问共享计算机范围的存储位置 `C:\ProgramData\IsolatedStorage\`。她无法访问 Bob 的用户配置文件目录 `C:\Users\Bob\`。

如果 Mallory 要攻击 Bob，她可以将数据写入到计算机范围的存储位置，然后尝试影响 Bob 从计算机范围的存储中读取数据。当 Bob 运行从该存储区读取的应用时，该应用将对 Mallory 放置在此处的数据进行操作，但从 Bob 的用户帐户的上下文中运行。本文档的其余部分介绍了各种攻击途径和应用可执行的步骤，以最大程度地降低这些攻击的风险。

> **NOTE**
>
> 为了进行这种攻击，Mallory 需要：
>
> - 计算机上的用户帐户。
> - 能够将文件放在文件系统上的已知位置。
> - 了解 Bob 将在某个时间点运行尝试读取此数据的应用。
>
> 这些不是适用于标准单用户桌面环境（例如家庭电脑或单员工企业工作站）的威胁途径。

特权提升

当 Bob 的应用读取 Mallory 的文件并根据该负载的内容自动尝试执行某些操作时，会出现特权提升攻击。假设某个应用从计算机范围的存储读取启动脚本的内容，并将这些内容传递到 `Process.Start`。如果 Mallory 可以在计算机范围的存储中放置恶意脚本，则在 Bob 启动其应用时：

- Bob 的应用会在其用户配置文件的上下文中分析和启动 Mallory 的恶意脚本。
- Mallory 会在本地计算机上获取对 Bob 帐户的访问权限。

拒绝服务

当 Bob 的应用读取 Mallory 的文件后崩溃或正常停止运行时，就会出现拒绝服务攻击。再次假设前面提到的应用，它尝试从计算机范围的存储分析启动脚本。如果 Mallory 可以将格式不正确的文件放在计算机范围的存储中，那么她可以：

- 使 Bob 的应用在启动路径初期引发异常。
- 出于异常原因，阻止应用成功启动。

然后，她拒绝 Bob 在他自己的用户帐户下启动该应用。

信息泄露

当 Mallory 可以诱使 Bob 泄露 Mallory 通常不能访问的文件内容时，会出现信息泄露。假设 Bob 有一个机密文件 C:\Users\Bob\secret.txt，Mallory 想要读取该文件。她知道该文件的路径，但无法读取，因为 Windows 禁止她获取对 Bob 用户配置文件目录的访问权限。

相反，Mallory 会将硬链接放置在计算机范围的存储中。这是一种特殊类型的文件，它本身不包含任何内容，而是指向磁盘上的另一个文件。尝试读取硬链接文件将改为读取链接所指向文件的内容。创建硬链接后，Mallory 仍无法读取文件内容，因为她无权访问链接的目标（`C:\Users\Bob\secret.txt`）。不过，Bob 有权访问此文件。

当 Bob 的应用现在从计算机范围的存储中读取内容时，会无意中读取他的 `secret.txt` 文件的内容，就像文件本身已存在于计算机范围的存储中一样。当 Bob 的应用退出时，如果该应用尝试将文件重新保存到计算机范围的存储中，则最终会在 *C:\ProgramData\IsolatedStorage* 目录中放置该文件的实际副本。由于此目录可由计算机上的任何用户读取，Mallory 现在可以读取该文件的内容。

防范这些攻击的最佳做法

重要提示：如果你的环境具有多个相互不受信任的用户，请不要调用 API `IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.Machine)` 或调用工具 `storeadm.exe /machine /list`。这两种情况都假设它们正在对受信任的数据进行操作。如果攻击者可以在计算机范围的存储中放置恶意的有效负载，则该有效负载会在运行这些命令的用户的上下文中导致特权提升攻击。

如果在多用户环境中操作，请重新考虑使用针对计算机范围的独立存储功能。如果应用必须从计算机范围的位置中读取数据，则最好从仅由管理员帐户写入的位置读取数据。`%PROGRAMFILES%` 目录和 `HKLM` 注册表配置单元是仅由管理员写入并可由所有人读取的位置的示例。因此从这些位置读取的数据为可信数据。

如果应用必须在多用户环境中使用计算机范围，请验证从计算机范围的存储中读取的任何文件内容。如果应用反序列化这些文件中的对象图，请考虑使用更安全的序列化程序，如 `XmlSerializer`，而不是 `BinaryFormatter` 或 `NetDataContractSerializer` 等危险序列化程序。使用深度嵌套的对象图或根据文件内容执行资源分配的对象图。

独立存储位置

有时候，使用操作系统的文件系统来验证对独立存储进行的更改会非常有帮助。你可能还需要了解独立存储文件的位置。该位置随操作系统的不同而不同。下表显示了在几个常见操作系统上创建独立存储的根位置。在此根位置下查找 Microsoft\IsolatedStorage 目录。您必须更改文件夹设置以显示隐藏文件和文件夹，才能查看到文件系统中的独立存储。

tttt	ttttttttt
Windows 2000、Windows XP、Windows Server 2003（从 Windows NT 4.0 升级）	支持漫游的存储区 = <SYSTEMROOT>\Profiles\<user>\Application Data 非漫游存储区 = <SYSTEMROOT>\Profiles\<user>\Local Settings\Application Data
Windows 2000 - 全新安装（和从 Windows 98 及 Windows NT 3.51 升级）	支持漫游的存储区 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Application Data 非漫游存储区 = <SYSTEMDRIVE>\Documents and Settings\<user>\Local Settings\Application Data
Windows XP、Windows Server 2003 - 全新安装（和从 Windows 2000 及 Windows 98 升级）	支持漫游的存储区 = <SYSTEMDRIVE>\Documents and Settings\ <user>\Application Data 非漫游存储区 = <SYSTEMDRIVE>\Documents and Settings\<user>\Local Settings\Application Data

tttt	ttttttttt
Windows 8、Windows 7、Windows Server 2008、Windows Vista	支持漫游的存储区 = <SYSTEMDRIVE>\Users\<user>\AppData\Roaming 非漫游存储区 = <SYSTEMDRIVE>\Users\<user>\AppData\Local

创建、枚举和删除独立存储

.NET 在 System.IO.IsolatedStorage 命名空间中提供了三个类来帮助你执行涉及独立存储的任务：

- IsolatedStorageFile派生自 System.IO.IsolatedStorage.IsolatedStorage，它提供对存储的程序集和应用程序文件的基本管理。IsolatedStorageFile 类的实例表示位于文件系统中的单个存储区。
- IsolatedStorageFileStream 派生自 System.IO.FileStream，它提供对存储中的文件的访问。
- IsolatedStorageScope 是一个枚举，使您可以创建并选择具有适当隔离类型的存储区。

独立存储类使您可以创建、枚举并删除独立存储。通过 IsolatedStorageFile 对象可以使用执行这些任务的方法。某些操作要求你具有 IsolatedStorageFilePermission 权限（表示管理独立存储的权限）；你可能还需要具有访问文件或目录的操作系统权限。

有关演示常见的独立存储任务的一系列示例，请参见 相关主题中列出的帮助主题。

独立存储的情况

在许多情况下，独立存储非常有用，包括这四种场景：

- 下载的控件。不允许从 Internet 下载的托管代码控件通过正常的 I/O 类写入硬盘，但它们可以使用独立存储来持久保存用户设置和应用程序状态。
- 共享组件存储。应用程序间共享的组件可以使用独立存储来提供对数据存储区的有控制的访问。
- 服务器存储。服务器应用程序可以使用独立存储为请求应用程序的大量用户提供单独的存储区。因为独立存储始终按用户进行隔离，所以服务器必须模拟发出请求的用户。在这种情况下，根据主体的标识隔离数据，该标识与应用程序用来区分其用户的标识是同一个标识。
- 漫游。应用程序还可以将独立存储和漫游用户配置文件一起使用。这允许用户的独立存储区和配置文件一起漫游。

不要在以下情况下使用独立存储：

- 用来存储重要机密，例如不加密的密钥或密码，因为独立存储对高度受信任的代码、非托管代码或计算机的受信任用户不设防。
- 用来存储代码。
- 用来存储管理员控制的配置和部署设置。（因为管理员不控制用户首选项，所以用户首选项不被认为是配置设置。）

许多应用程序都使用数据库来存储和隔离数据，在这种情况下，数据库中的一个或多个行可能代表某个特定用户的存储。当用户数较少时、当使用数据库的系统开销非常大时或当不存在数据库功能时，您可以选择使用独立存储而不使用数据库。另外，当应用程序要求比数据库的行所提供的存储更加灵活和复杂的存储时，独立存储也可以提供一个可行的替代方案。

相关文章

TITLE	𝐂
隔离的类型	描述不同类型的隔离。
如何：获取独立存储的存储区	提供使用 IsolatedStorageFile 类获取按用户和程序集隔离的存储区的示例。
如何：枚举独立存储的存储区	演示如何使用 IsolatedStorageFile.GetEnumerator 方法计算用户的所有独立存储的大小。
如何：删除独立存储中的存储区	演示如何使用 IsolatedStorageFile.Remove 方法以两种不同方式删除独立存储区。
如何：预见独立存储中的空间不足条件	说明如何测量独立存储区中剩余的空间。
如何：在独立存储中创建文件和目录	提供一些在独立存储区中创建文件和目录的示例。
如何：在独立存储中查找现有文件和目录	演示如何读取独立存储区中的目录结构和文件。
如何：在独立存储中读取和写入文件	提供一个向独立存储文件写入字符串并将其读取回的示例。
如何：在独立存储中删除文件和目录	演示如何删除独立存储文件和目录。
文件和流 I/O	解释如何执行同步和异步文件和数据流访问。

参考

- System.IO.IsolatedStorage.IsolatedStorage
- System.IO.IsolatedStorage.IsolatedStorageFile
- System.IO.IsolatedStorage.IsolatedStorageFileStream
- System.IO.IsolatedStorage.IsolatedStorageScope

隔离的类型

2021/11/16 •

独立存储始终仅限创建它的用户访问。为了实现这种隔离，公共语言运行时使用操作系统识别的相同用户标识，即与存储打开时的代码运行进程相关联的标识。虽然此标识是已验证用户标识，但模拟可能会导致当前用户的标识发生动态变化。

独立存储访问的限制条件还包括，与应用的域和程序集相关联的标识或仅与程序集相关联的标识。运行时通过以下方式获取这些标识：

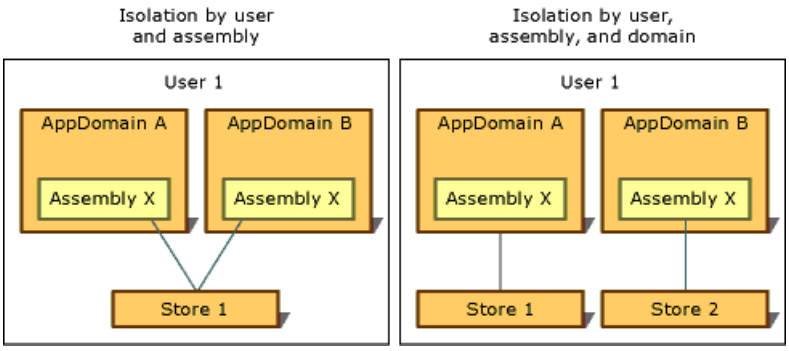
- 域标识表示证明应用的证据，对于 Web 应用，这可能就是完整 URL。对于 shell 托管代码，域标识可能基于应用目录路径。例如，如果通过路径 C:\Office\MyApp.exe 运行可执行文件，域标识为 C:\Office\MyApp.exe。
- 程序集标识是证明程序集的证据。这可能来自加密数字签名，可以是程序集的强名称、程序集的软件发行者或程序集的 URL 标识。如果程序集同时包含强名称和软件发行者标识，使用的是软件发行者标识。如果程序集来自 Internet 且未签名，使用的是 URL 标识。若要详细了解程序集和强名称，请参阅使用程序集编程。
- 漫游存储与有漫游用户策略文件的用户一起移动。文件被写入网络目录，并下载到用户登录的所有计算机中。若要详细了解漫游用户策略文件，请参阅 IsolatedStorageScope.Roaming。

通过将用户、域和程序集标识这些概念相结合，独立存储可以通过下列方式隔离数据，每种方式都有自己的使用方案：

- 按用户和程序集隔离
- 按用户、域和程序集隔离

这两种隔离都可以与漫游用户策略文件结合使用。有关详细信息，请参阅独立存储和漫游部分。

下图展示了存储在不同范围的隔离情况：



除漫游存储外，独立存储始终按计算机隐式隔离，因为它使用指定计算机的本地存储设备。

> **IMPORTANT**
> 独立存储不适用于 Windows 8.x 应用商店应用。请改用 Windows 运行时 API 包含的 `Windows.Storage` 命名空间中的应用程序数据类来存储本地数据和文件。有关详细信息，请参阅 Windows 开发人员中心的 应用程序数据 。

按用户和程序集隔离

如果需要从任何应用的域都可以访问程序集使用的数据存储，按用户和程序集隔离更为合适。在这种情况下，独立存储通常用于存储跨多个应用的数据，而不是与任何特定应用绑定的数据，如用户名或许可证信息。若要访问按用户和程序集隔离的存储，必须信任代码在应用之间传输信息。通常情况下，按用户和程序集隔离可用于 Intranet，但不可用于 Internet。调用静态 IsolatedStorageFile.GetStore 方法并传入用户和程序集 IsolatedStorageScope，即可返回采用这种隔离的存储。

下面的代码示例检索按用户和程序集隔离的存储。可通过 `isoFile` 对象访问此存储。

```
IsolatedStorageFile^ isoFile =
    IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
        IsolatedStorageScope::Assembly, (Type^)nullptr, (Type^)nullptr);
```

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly, null, null);
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
        IsolatedStorageScope.Assembly, Nothing, Nothing)
```

有关使用证据参数的示例，请参阅 GetStore(IsolatedStorageScope, Evidence, Type, Evidence, Type)。

GetUserStoreForAssembly 方法可用作快捷方式，如下面的代码示例所示。此快捷方式不能用于打开漫游存储；在这种情况下，请使用 GetStore。

```
IsolatedStorageFile^ isoFile = IsolatedStorageFile::GetUserStoreForAssembly();
```

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForAssembly();
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetUserStoreForAssembly()
```

按用户、域和程序集隔离

如果应用使用需要专用数据存储的第三方程序集，可以使用独立存储来存储专用数据。按用户、域和程序集隔离可确保，仅当使用程序集的应用在程序集创建存储时正在运行时，且仅当为其创建存储的用户运行应用时，只有给定程序集中的代码才能访问数据。按用户、域和程序集隔离可防止第三方程序集将数据泄漏给其他应用。如果确定要使用独立存储，但不确定要使用哪种类型的隔离，此隔离类型应为默认选择。调用 IsolatedStorageFile 的静态 GetStore 方法并传入用户、域和程序集 IsolatedStorageScope，即可返回采用这种隔离的存储。

下面的代码示例检索按用户、域和程序集隔离的存储。可通过 `isoFile` 对象访问此存储。

```
IsolatedStorageFile^ isoFile =
    IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
        IsolatedStorageScope::Domain |
        IsolatedStorageScope::Assembly, (Type^)nullptr, (Type^)nullptr);
```

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Domain |
        IsolatedStorageScope.Assembly, null, null);
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
        IsolatedStorageScope.Domain Or _
        IsolatedStorageScope.Assembly, Nothing, Nothing)
```

另一个方法可用作快捷方式，如下面的代码示例所示。此快捷方式不能用于打开漫游存储；在这种情况下，请使用 GetStore。

```
IsolatedStorageFile^ isoFile = IsolatedStorageFile::GetUserStoreForDomain();
```

```
IsolatedStorageFile isoFile = IsolatedStorageFile.GetUserStoreForDomain();
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetUserStoreForDomain()
```

独立存储和漫游

漫游用户策略文件是一项 Windows 功能，可便于用户在网络上设置标识，并使用此标识登录任何网络计算机，同时应用所有个性化设置。使用独立存储的程序集可以指定，用户的独立存储应随漫游用户策略文件一起移动。漫游可以与按用户和程序集隔离或按用户、域和程序集隔离结合使用。如果未使用漫游范围，即使使用漫游用户策略文件，存储也不会漫游。

下面的代码示例检索按用户和程序集隔离的漫游存储。可通过 `isoFile` 对象访问此存储。

```
IsolatedStorageFile^ isoFile =
    IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
        IsolatedStorageScope::Assembly |
        IsolatedStorageScope::Roaming, (Type^)nullptr, (Type^)nullptr);
```

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly |
        IsolatedStorageScope.Roaming, null, null);
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
        IsolatedStorageScope.Assembly Or _
        IsolatedStorageScope.Roaming, Nothing, Nothing)
```

可以添加域范围，以创建按用户、域和应用隔离的漫游存储。下面的代码示例展示了此操作。

```
IsolatedStorageFile^ isoFile =
    IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
        IsolatedStorageScope::Assembly | IsolatedStorageScope::Domain |
        IsolatedStorageScope::Roaming, (Type^)nullptr, (Type^)nullptr);
```

```
IsolatedStorageFile isoFile =
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
        IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain |
        IsolatedStorageScope.Roaming, null, null);
```

```
Dim isoFile As IsolatedStorageFile = _
    IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
        IsolatedStorageScope.Assembly Or IsolatedStorageScope.Domain Or _
        IsolatedStorageScope.Roaming, Nothing, Nothing)
```

请参阅

- IsolatedStorageScope
- 独立存储

如何：获取独立存储的存储区

2021/11/16 •

独立存储区公开数据隔离舱中的虚拟文件系统。IsolatedStorageFile 类提供许多与独立存储区交互的方法。为了创建和检索存储区，IsolatedStorageFile 提供了三种静态方法：

- GetUserStoreForAssembly 返回按用户和程序集隔离的存储。
- GetUserStoreForDomain 返回按域和程序集隔离的存储。

  这两种方法检索属于所调用代码的存储区。
- 静态方法 GetStore 返回通过传入一组范围参数指定的独立存储。

下面的示例返回按用户、程序集和域隔离的存储区。

```
IsolatedStorageFile^ isoStore = IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
    IsolatedStorageScope::Assembly | IsolatedStorageScope::Domain, (Type ^)nullptr, (Type ^)nullptr);
```

```
IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
    IsolatedStorageScope.Assembly | IsolatedStorageScope.Domain, null, null);
```

```
Dim isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
    IsolatedStorageScope.Assembly Or IsolatedStorageScope.Domain, Nothing, Nothing)
```

您可以使用 GetStore 方法指定存储区应该和漫游用户配置文件一起漫游。若要详细了解如何对此进行设置，请参阅隔离类型。

从不同程序集获取的独立存储默认就是不同的存储。通过在 GetStore 方法的参数中传递程序集或域证据，可以访问不同程序集或域的存储区。这需要拥有权限，才能按应用域标识访问独立存储。有关详细信息，请参阅 GetStore 方法重载。

GetUserStoreForAssembly、GetUserStoreForDomain 和 GetStore 方法返回 IsolatedStorageFile 对象。若要了解如何确定哪种隔离类型最适合自己的情况，请参阅隔离类型。当您具有了独立存储文件对象时，您便可以使用独立存储方法来读取、写入、创建和删除文件及目录了。

没有一种机制可用来防止代码向没有足够访问权限来自己获取存储区的代码传递 IsolatedStorageFile 对象。仅当获得对 IsolatedStorage 对象的引用时（通常在 GetUserStoreForAssembly、GetUserStoreForDomain 或 GetStore 方法中），才会检查域和程序集标识以及独立存储权限。因此，保护对 IsolatedStorageFile 对象的引用是使用这些引用的代码的责任。

示例

下面的代码提供了一个简单的类示例，它包含按用户和程序集隔离的存储区。通过向 IsolatedStorageScope.Domain 方法传递的自变量添加 GetStore，此代码可更改为检索按用户、域和程序集隔离的存储区。

运行代码后，可以在命令行处键入“ StoreAdm /LIST”，以确认存储是否已创建。这会运行独立存储工具 (Storeadm.exe)，并列出用户当前的所有独立存储。

```
using namespace System;
using namespace System::IO::IsolatedStorage;

public ref class ObtainingAStore
{
public:
    static void Main()
    {
        // Get a new isolated store for this assembly and put it into an
        // isolated store object.

        IsolatedStorageFile^ isoStore = IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type ^)nullptr);
    }
};
```

```
using System;
using System.IO.IsolatedStorage;

public class ObtainingAStore
{
    public static void Main()
    {
        // Get a new isolated store for this assembly and put it into an
        // isolated store object.

        IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
    }
}
```

```
Imports System.IO.IsolatedStorage

Public Class ObtainingAStore
    Public Shared Sub Main()
        ' Get a new isolated store for this assembly and put it into an
        ' isolated store object.

        Dim isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
            IsolatedStorageScope.Assembly, Nothing, Nothing)
    End Sub
End Class
```

另请参阅

- IsolatedStorageFile
- IsolatedStorageScope
- 独立存储
- 隔离的类型
- .NET 中的程序集

如何：枚举独立存储的存储区

2021/11/16 •

您可以使用 IsolatedStorageFile.GetEnumerator 静态方法枚举当前用户的所有独立存储区。该方法采用 IsolatedStorageScope 值并且返回 IsolatedStorageFile 枚举器。若要枚举存储区，您必须具有指定IsolatedStorageFilePermission 值的 AdministerIsolatedStorageByUser 权限。如果调用采用 GetEnumerator 值的 User 方法，它将返回一组为当前用户定义的 IsolatedStorageFile 对象。

示例

下面的代码示例使用 GetEnumerator 方法获取按用户和程序集隔离的存储区，创建一些文件，并检索这些文件。

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;

public class EnumeratingStores
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateFile("TestFileA.Txt");
            isoStore.CreateFile("TestFileB.Txt");
            isoStore.CreateFile("TestFileC.Txt");
            isoStore.CreateFile("TestFileD.Txt");
        }

        IEnumerator allFiles = IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User);
        long totalsize = 0;

        while (allFiles.MoveNext())
        {
            IsolatedStorageFile storeFile = (IsolatedStorageFile)allFiles.Current;
            totalsize += (long)storeFile.UsedSize;
        }

        Console.WriteLine("The total size = " + totalsize);
    }
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage

Module Module1
    Sub Main()
        Using isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
IsolatedStorageScope.Assembly, Nothing, Nothing)
            isoStore.CreateFile("TestFileA.Txt")
            isoStore.CreateFile("TestFileB.Txt")
            isoStore.CreateFile("TestFileC.Txt")
            isoStore.CreateFile("TestFileD.Txt")
        End Using

        Dim allFiles As IEnumerator = IsolatedStorageFile.GetEnumerator(IsolatedStorageScope.User)
        Dim totalsize As Long = 0

        While (allFiles.MoveNext())
            Dim storeFile As IsolatedStorageFile = CType(allFiles.Current, IsolatedStorageFile)
            totalsize += CType(storeFile.UsedSize, Long)
        End While

        Console.WriteLine("The total size = " + totalsize.ToString())

    End Sub
End Module
```

另请参阅

- IsolatedStorageFile
- 独立存储

如何：删除独立存储中的存储区

2021/11/16 •

IsolatedStorageFile 类提供了两个用于删除独立存储文件的方法：

- 实例方法 Remove() 不采用任何参数，并删除调用它的存储区。此操作无需任何权限。可以访问此存储区的任何代码都可以删除该存储区内的任何或所有数据。
- 静态方法 Remove(IsolatedStorageScope) 采用 User 枚举值，并删除当前运行该代码的用户的所有存储区。此操作需要 IsolatedStorageFilePermission 权限来写入 AdministerIsolatedStorageByUser 值。

示例

下面的代码示例演示了如何使用静态的实例方法 Remove 。该类获取两个存储区：一个按用户和程序集隔离；另一个按用户、域和程序集隔离。随后，将通过调用独立存储文件 Remove() 的 `isoStore1` 方法来删除用户、域和程序集存储区。接下来，将通过调用静态方法 Remove(IsolatedStorageScope)来删除该用户的所有剩余存储区。

```
using namespace System;
using namespace System::IO::IsolatedStorage;

public ref class DeletingStores
{
public:
    static void Main()
    {
        // Get a new isolated store for this user, domain, and assembly.
        // Put the store into an IsolatedStorageFile object.

        IsolatedStorageFile^ isoStore1 =  IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Domain | IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type
^)nullptr);
        Console::WriteLine("A store isolated by user, assembly, and domain has been obtained.");

        // Get a new isolated store for user and assembly.
        // Put that store into a different IsolatedStorageFile object.

        IsolatedStorageFile^ isoStore2 = IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type ^)nullptr);
        Console::WriteLine("A store isolated by user and assembly has been obtained.");

        // The Remove method deletes a specific store, in this case the
        // isoStore1 file.

        isoStore1->Remove();
        Console::WriteLine("The user, domain, and assembly isolated store has been deleted.");

        // This static method deletes all the isolated stores for this user.

        IsolatedStorageFile::Remove(IsolatedStorageScope::User);
        Console::WriteLine("All isolated stores for this user have been deleted.");
    } // End of Main.
};

int main()
{
    DeletingStores::Main();
}
```

```
using System;
using System.IO.IsolatedStorage;

public class DeletingStores
{
    public static void Main()
    {
        // Get a new isolated store for this user, domain, and assembly.
        // Put the store into an IsolatedStorageFile object.

        IsolatedStorageFile isoStore1 =  IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null, null);
        Console.WriteLine("A store isolated by user, assembly, and domain has been obtained.");

        // Get a new isolated store for user and assembly.
        // Put that store into a different IsolatedStorageFile object.

        IsolatedStorageFile isoStore2 = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
        Console.WriteLine("A store isolated by user and assembly has been obtained.");

        // The Remove method deletes a specific store, in this case the
        // isoStore1 file.

        isoStore1.Remove();
        Console.WriteLine("The user, domain, and assembly isolated store has been deleted.");

        // This static method deletes all the isolated stores for this user.

        IsolatedStorageFile.Remove(IsolatedStorageScope.User);
        Console.WriteLine("All isolated stores for this user have been deleted.");
    } // End of Main.
}
```

```
Imports System.IO.IsolatedStorage

Public Class DeletingStores
    Public Shared Sub Main()
        ' Get a new isolated store for this user, domain, and assembly.
        ' Put the store into an IsolatedStorageFile object.

        Dim isoStore1 As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
            IsolatedStorageScope.Domain Or IsolatedStorageScope.Assembly, Nothing, Nothing)
        Console.WriteLine("A store isolated by user, assembly, and domain has been obtained.")

        ' Get a new isolated store for user and assembly.
        ' Put that store into a different IsolatedStorageFile object.

        Dim isoStore2 As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
            IsolatedStorageScope.Assembly, Nothing, Nothing)
        Console.WriteLine("A store isolated by user and assembly has been obtained.")

        ' The Remove method deletes a specific store, in this case the
        ' isoStore1 file.

        isoStore1.Remove()
        Console.WriteLine("The user, domain, and assembly isolated store has been deleted.")

        ' This static method deletes all the isolated stores for this user.

        IsolatedStorageFile.Remove(IsolatedStorageScope.User)
        Console.WriteLine("All isolated stores for this user have been deleted.")

    End Sub
End Class
```

另请参阅

- IsolatedStorageFile
- 独立存储

如何：预见独立存储中的空间不足条件

2021/11/16 •

使用独立存储的代码受配额限制，配额指定了独立存储文件和目录所在数据隔离舱的大小上限。该配额由安全策略定义，管理员可以对其进行配置。如果尝试写入数据时超过了所允许的最大大小，将引发 IsolatedStorageException 异常并使操作失败。这有助于防止恶意的拒绝服务攻击，此类攻击可能会导致应用因为数据存储已满而拒绝请求。

为有助于确定给定写入尝试是否有可能由于此原因而失败，IsolatedStorage 类提供了下列三个只读属性：AvailableFreeSpace、UsedSize 和 Quota。您可以使用这些属性来确定写入存储区是否将导致超过存储区所允许的最大大小。记住独立存储可能被同时访问；因此，当计算剩余存储量时，该存储空间可能在您尝试写入存储区时已被使用。但是，您可以使用存储区的最大大小来确定是否将达到可用存储的上限。

Quota 属性依赖程序集中的证据来正常工作。因此，只应在使用 IsolatedStorageFile、GetUserStoreForAssembly 或 GetUserStoreForDomain 方法创建的 GetStore 对象上检索此属性。以其他任何方式创建的 IsolatedStorageFile 对象（例如从 GetEnumerator 方法返回的对象）将无法返回准确的最大大小。

示例

下面的代码示例获取独立存储，创建一些文件，并检索 AvailableFreeSpace 属性。剩余空间以字节为单位进行报告。

```
using namespace System;
using namespace System::IO;
using namespace System::IO::IsolatedStorage;

public ref class CheckingSpace
{
public:
    static void Main()
    {
        // Get an isolated store for this assembly and put it into an
        // IsolatedStoreFile object.
        IsolatedStorageFile^ isoStore =  IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type ^)nullptr);

        // Create a few placeholder files in the isolated store.
        gcnew IsolatedStorageFileStream("InTheRoot.txt", FileMode::Create, isoStore);
        gcnew IsolatedStorageFileStream("Another.txt", FileMode::Create, isoStore);
        gcnew IsolatedStorageFileStream("AThird.txt", FileMode::Create, isoStore);
        gcnew IsolatedStorageFileStream("AFourth.txt", FileMode::Create, isoStore);
        gcnew IsolatedStorageFileStream("AFifth.txt", FileMode::Create, isoStore);

        Console::WriteLine(isoStore->AvailableFreeSpace + " bytes of space remain in this isolated store.");
    } // End of Main.
};

int main()
{
    CheckingSpace::Main();
}
```

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CheckingSpace
{
    public static void Main()
    {
        // Get an isolated store for this assembly and put it into an
        // IsolatedStoreFile object.
        IsolatedStorageFile isoStore =  IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);

        // Create a few placeholder files in the isolated store.
        new IsolatedStorageFileStream("InTheRoot.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("Another.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AThird.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AFourth.txt", FileMode.Create, isoStore);
        new IsolatedStorageFileStream("AFifth.txt", FileMode.Create, isoStore);

        Console.WriteLine(isoStore.AvailableFreeSpace + " bytes of space remain in this isolated store.");
    } // End of Main.
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage

Public Class CheckingSpace
    Public Shared Sub Main()
        ' Get an isolated store for this assembly and put it into an
        ' IsolatedStoreFile object.
        Dim isoStore As IsolatedStorageFile = _
            IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
            IsolatedStorageScope.Assembly, Nothing, Nothing)

        ' Create a few placeholder files in the isolated store.
        Dim aStream As New IsolatedStorageFileStream("InTheRoot.txt", FileMode.Create, isoStore)
        Dim bStream As New IsolatedStorageFileStream("Another.txt", FileMode.Create, isoStore)
        Dim cStream As New IsolatedStorageFileStream("AThird.txt", FileMode.Create, isoStore)
        Dim dStream As New IsolatedStorageFileStream("AFourth.txt", FileMode.Create, isoStore)
        Dim eStream As New IsolatedStorageFileStream("AFifth.txt", FileMode.Create, isoStore)

        Console.WriteLine(isoStore.AvailableFreeSpace + " bytes of space remain in this isolated store.")
    End Sub
End Class
```

另请参阅

- IsolatedStorageFile
- 独立存储
- 如何：获取独立存储的存储区

如何：在独立存储中创建文件和目录

2021/11/16 •

获得独立存储区之后，可以创建用于存储数据的目录和文件。在存储中，文件名和目录名称是相对于虚拟文件系统的根目录进行指定。

若要创建目录，请使用 IsolatedStorageFile.CreateDirectory 实例方法。如果为不存在的目录指定了一个子目录，则会同时创建这两个目录。如果您指定的目录已存在，该方法将返回而不创建目录，并且不会引发异常。但是，如果您指定的目录名称包含无效字符，将引发 IsolatedStorageException 异常。

若要创建文件，请使用 IsolatedStorageFile.CreateFile 方法。

在 Windows 操作系统，独立存储文件和目录名不区分大小写。这样，如果您创建了一个名为 `ThisFile.txt` 的文件，然后又创建了名为 `THISFILE.TXT` 的另一个文件，实际上只创建了一个文件。文件名保留原始大小写只为了方便本文演示。

如果路径包含的目录不存在，则创建独立存储文件会引发 IsolatedStorageException。

示例

下面的代码示例展示了如何在独立存储中创建文件和目录。

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class CreatingFilesDirectories
{
    public static void Main()
    {
        using (IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null, null))
        {
            isoStore.CreateDirectory("TopLevelDirectory");
            isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
            isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
            Console.WriteLine("Created directories.");

            isoStore.CreateFile("InTheRoot.txt");
            Console.WriteLine("Created a new file in the root.");

            isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
            Console.WriteLine("Created a new file in the InsideDirectory.");
        }
    }
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage

Module Module1
    Sub Main()
        Using isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
IsolatedStorageScope.Assembly Or IsolatedStorageScope.Domain, Nothing, Nothing)

            isoStore.CreateDirectory("TopLevelDirectory")
            isoStore.CreateDirectory("TopLevelDirectory/SecondLevel")
            isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory")
            Console.WriteLine("Created directories.")

            isoStore.CreateFile("InTheRoot.txt")
            Console.WriteLine("Created a new file in the root.")

            isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt")
            Console.WriteLine("Created a new file in the InsideDirectory.")
        End Using
    End Sub
End Module
```

请参阅

- IsolatedStorageFile
- IsolatedStorageFileStream
- 独立存储

如何：在独立存储中查找现有文件和目录

2021/11/16 •

为了搜索独立存储中的目录，请使用 IsolatedStorageFile.GetDirectoryNames 方法。此方法接受表示搜索模式的字符串。你可以在搜索模式中使用单字符 (?) 和多字符 (*) 通配符，但是通配符必须出现在名称的最后一部分。例如，`directory1/*ect*` 是有效的搜索字符串，但 `*ect*/directory2` 不是。

若要搜索文件，请使用 IsolatedStorageFile.GetFileNames 方法。对适用于 GetDirectoryNames 的搜索字符串中通配符的限制也适用于 GetFileNames。

这些方法都不是递归的；IsolatedStorageFile 类不提供用于列出存储区中所有目录或文件的任何方法。但是，在下面的代码示例中显示有递归方法。

示例

下面的代码示例展示了如何在独立存储中创建文件和目录。首先，检索一个为用户、域和程序集隔离的存储区，并放入 `isoStore` 变量。CreateDirectory 方法用于设置几个不同的目录，IsolatedStorageFileStream(String, FileMode, IsolatedStorageFile) 构造函数在这些目录中创建了一些文件。然后，代码循环访问 `GetAllDirectories` 方法的结果。该方法使用 GetDirectoryNames 来查找当前目录中的所有目录名。这些名称存储在数组中，然后 `GetAllDirectories` 调用其本身，传入它所找到的每个目录。结果是所有目录名都返回到数组中。接下来，代码调用 `GetAllFiles` 方法。该方法调用 `GetAllDirectories` 来查找所有目录的名称，然后它使用 GetFileNames 方法检查文件的每个目录。结果是在数组中返回，以供显示。

```
using namespace System;
using namespace System::IO;
using namespace System::IO::IsolatedStorage;
using namespace System::Collections;
using namespace System::Collections::Generic;

public class FindingExistingFilesAndDirectories
{
public:
    // Retrieves an array of all directories in the store, and
    // displays the results.
    static void Main()
    {
        // This part of the code sets up a few directories and files in the
        // store.
        IsolatedStorageFile^ isoStore = IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type ^)nullptr);
        isoStore->CreateDirectory("TopLevelDirectory");
        isoStore->CreateDirectory("TopLevelDirectory/SecondLevel");
        isoStore->CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        gcnew IsolatedStorageFileStream("InTheRoot.txt", FileMode::Create, isoStore);
        gcnew IsolatedStorageFileStream("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt",
            FileMode::Create, isoStore);
        // End of setup.

        Console::WriteLine('\r');
        Console::WriteLine("Here is a list of all directories in this isolated store:");

        for each (String^ directory in GetAllDirectories("*", isoStore))
        {
            Console::WriteLine(directory);
        }
        Console::WriteLine('\r');

        // Retrieve all the files in the directory by calling the GetFiles
```

```
        // Retrieve all the files in the directory by calling the GetFiles
        // method.

        Console::WriteLine("Here is a list of all the files in this isolated store:");
        for each (String^ file in GetAllFiles("*", isoStore))
        {
            Console::WriteLine(file);
        }

    } // End of Main.

    // Method to retrieve all directories, recursively, within a store.
    static List<String^>^ GetAllDirectories(String^ pattern, IsolatedStorageFile^ storeFile)
    {
        // Get the root of the search string.
        String^ root = Path::GetDirectoryName(pattern);

        if (root != "")
        {
            root += "/";
        }

        // Retrieve directories.
        array<String^>^ directories = storeFile->GetDirectoryNames(pattern);

        List<String^>^ directoryList = gcnew List<String^>(directories);

        // Retrieve subdirectories of matches.
        for (int i = 0, max = directories->Length; i < max; i++)
        {
             String^ directory = directoryList[i] + "/";
             List<String^>^ more = GetAllDirectories (root + directory + "*", storeFile);

             // For each subdirectory found, add in the base path.
             for (int j = 0; j < more->Count; j++)
             {
                 more[j] = directory + more[j];
             }

            // Insert the subdirectories into the list and
            // update the counter and upper bound.
            directoryList->InsertRange(i + 1, more);
            i += more->Count;
            max += more->Count;
        }

        return directoryList;
    }

    static List<String^>^ GetAllFiles(String^ pattern, IsolatedStorageFile^ storeFile)
    {
        // Get the root and file portions of the search string.
        String^ fileString = Path::GetFileName(pattern);
        array<String^>^ files = storeFile->GetFileNames(pattern);

        List<String^>^ fileList = gcnew List<String^>(files);

        // Loop through the subdirectories, collect matches,
        // and make separators consistent.
        for each (String^ directory in GetAllDirectories( "*", storeFile))
        {
            for each (String^ file in storeFile->GetFileNames(directory + "/" + fileString))
            {
                fileList->Add((directory + "/" + file));
            }
        }

        return fileList;
    } // End of GetFiles.
};
```

```
};

int main()
{
    FindingExistingFilesAndDirectories::Main();
}
```

```
using System;
using System.IO;
using System.IO.IsolatedStorage;
using System.Collections;
using System.Collections.Generic;

public class FindingExistingFilesAndDirectories
{
    // Retrieves an array of all directories in the store, and
    // displays the results.
    public static void Main()
    {
        // This part of the code sets up a few directories and files in the
        // store.
        IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);
        isoStore.CreateDirectory("TopLevelDirectory");
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        isoStore.CreateFile("InTheRoot.txt");
        isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        // End of setup.

        Console.WriteLine('\r');
        Console.WriteLine("Here is a list of all directories in this isolated store:");

        foreach (string directory in GetAllDirectories("*", isoStore))
        {
            Console.WriteLine(directory);
        }
        Console.WriteLine('\r');

        // Retrieve all the files in the directory by calling the GetFiles
        // method.

        Console.WriteLine("Here is a list of all the files in this isolated store:");
        foreach (string file in GetAllFiles("*", isoStore)){
            Console.WriteLine(file);
        }
    } // End of Main.

    // Method to retrieve all directories, recursively, within a store.
    public static List<String> GetAllDirectories(string pattern, IsolatedStorageFile storeFile)
    {
        // Get the root of the search string.
        string root = Path.GetDirectoryName(pattern);

        if (root != "")
        {
            root += "/";
        }

        // Retrieve directories.
        List<String> directoryList = new List<String>(storeFile.GetDirectoryNames(pattern));

        // Retrieve subdirectories of matches.
        for (int i = 0, max = directoryList.Count; i < max; i++)
        {
            string directory = directoryList[i] + "/";
            List<String> more = GetAllDirectories(root + directory + "*", storeFile);
```

```
            // For each subdirectory found, add in the base path.
            for (int j = 0; j < more.Count; j++)
            {
                more[j] = directory + more[j];
            }

            // Insert the subdirectories into the list and
            // update the counter and upper bound.
            directoryList.InsertRange(i + 1, more);
            i += more.Count;
            max += more.Count;
        }

        return directoryList;
    }

    public static List<String> GetAllFiles(string pattern, IsolatedStorageFile storeFile)
    {
        // Get the root and file portions of the search string.
        string fileString = Path.GetFileName(pattern);

        List<String> fileList = new List<String>(storeFile.GetFileNames(pattern));

        // Loop through the subdirectories, collect matches,
        // and make separators consistent.
        foreach (string directory in GetAllDirectories("*", storeFile))
        {
            foreach (string file in storeFile.GetFileNames(directory + "/" + fileString))
            {
                fileList.Add((directory + "/" + file));
            }
        }

        return fileList;
    } // End of GetFiles.
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage
Imports System.Collections
Imports System.Collections.Generic

Public class FindingExistingFilesAndDirectories
    ' These arrayLists hold the directory and file names as they are found.

    Private Shared directoryList As New List(Of String)
    Private Shared fileList As New List(Of String)

    ' Retrieves an array of all directories in the store, and
    ' displays the results.

    Public Shared Sub Main()
        ' This part of the code sets up a few directories and files in the store.
        Dim isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or _
            IsolatedStorageScope.Assembly Or IsolatedStorageScope.Domain, Nothing, Nothing)
        isoStore.CreateDirectory("TopLevelDirectory")
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel")
        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory")
        isoStore.CreateFile("InTheRoot.txt")
        isoStore.CreateFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt")
        ' End of setup.

        Console.WriteLine()
        Console.WriteLine("Here is a list of all directories in this isolated store:")

        GetAllDirectories("*", isoStore)
        For Each directory As String In directoryList
            Console.WriteLine(directory)
        Next

        Console.WriteLine()
        Console.WriteLine("Retrieve all the files in the directory by calling the GetFiles method.")

        GetAllFiles(isoStore)
        For Each file As String In fileList
            Console.WriteLine(file)
        Next
    End Sub

    Public Shared Sub GetAllDirectories(ByVal pattern As String, ByVal storeFile As IsolatedStorageFile)
        ' Retrieve directories.
        Dim directories As String() = storeFile.GetDirectoryNames(pattern)

        For Each directory As String In directories
            ' Add the directory to the final list.
            directoryList.Add((pattern.TrimEnd(CChar("*"))) + directory + "/")
            ' Call the method again using directory.
            GetAllDirectories((pattern.TrimEnd(CChar("*")) + directory + "/*"), storeFile)
        Next
    End Sub

    Public Shared Sub GetAllFiles(ByVal storefile As IsolatedStorageFile)
        ' This adds the root to the directory list.
        directoryList.Add("*")
        For Each directory As String In directoryList
            Dim files As String() = storefile.GetFileNames(directory + "*")
            For Each dirfile As String In files
                fileList.Add(dirfile)
            Next
        Next
    End Sub
End Class
```

另请参阅

- IsolatedStorageFile
- 独立存储

如何：在独立存储中读取和写入文件

2021/11/16 •

若要读取或写入独立存储区的文件，请使用包含流读取器（IsolatedStorageFileStream 对象）或流编写器（StreamReader 对象）的 StreamWriter 对象。

示例

下面的代码示例获取独立存储区，并检查存储区中是否存在 TestStore.txt 文件。如果不存在，则创建该文件并将“Hello Isolated Storage”写入该文件。如果 TestStore.txt 已存在，则示例代码会读取该文件。

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
IsolatedStorageScope.Assembly, null, null);

            if (isoStore.FileExists("TestStore.txt"))
            {
                Console.WriteLine("The file already exists!");
                using (IsolatedStorageFileStream isoStream = new IsolatedStorageFileStream("TestStore.txt",
FileMode.Open, isoStore))
                {
                    using (StreamReader reader = new StreamReader(isoStream))
                    {
                        Console.WriteLine("Reading contents:");
                        Console.WriteLine(reader.ReadToEnd());
                    }
                }
            }
            else
            {
                using (IsolatedStorageFileStream isoStream = new IsolatedStorageFileStream("TestStore.txt",
FileMode.CreateNew, isoStore))
                {
                    using (StreamWriter writer = new StreamWriter(isoStream))
                    {
                        writer.WriteLine("Hello Isolated Storage");
                        Console.WriteLine("You have written to the file.");
                    }
                }
            }
        }
    }
}
```

```
Imports System.IO
Imports System.IO.IsolatedStorage

Module Module1

    Sub Main()
        Dim isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
IsolatedStorageScope.Assembly, Nothing, Nothing)

        If (isoStore.FileExists("TestStore.txt")) Then
            Console.WriteLine("The file already exists!")
            Using isoStream As IsolatedStorageFileStream = New IsolatedStorageFileStream("TestStore.txt",
FileMode.Open, isoStore)
                Using reader As StreamReader = New StreamReader(isoStream)
                    Console.WriteLine("Reading contents:")
                    Console.WriteLine(reader.ReadToEnd())
                End Using
            End Using
        Else
            Using isoStream As IsolatedStorageFileStream = New IsolatedStorageFileStream("TestStore.txt",
FileMode.CreateNew, isoStore)
                Using writer As StreamWriter = New StreamWriter(isoStream)
                    writer.WriteLine("Hello Isolated Storage")
                    Console.WriteLine("You have written to the file.")
                End Using
            End Using
        End If
    End Sub

End Module
```

另请参阅

- IsolatedStorageFile
- IsolatedStorageFileStream
- System.IO.FileMode
- System.IO.FileAccess
- System.IO.StreamReader
- System.IO.StreamWriter
- 文件和流 I/O
- 独立存储

如何：在独立存储中删除文件和目录

2021/11/16 •

可以删除独立存储文件中的目录和文件。在存储区中，文件名和目录名依赖于操作系统，并指定为虚拟文件系统根目录的相对路径。在 Windows 操作系统中，它们不区分大小写。

System.IO.IsolatedStorage.IsolatedStorageFile 类提供了两个用于删除目录和文件的方法：DeleteDirectory 和 DeleteFile。如果尝试删除并不存在的文件和目录，则会引发 IsolatedStorageException 异常。如果在名称中包含通配符，DeleteDirectory 将引发 IsolatedStorageException 异常，并且 DeleteFile 会引发 ArgumentException 异常。

如果目录中包含任何文件或子目录，DeleteDirectory 方法将会失败。可以使用 GetFileNames 和 GetDirectoryNames 方法检索现有文件和目录。若要详细了解如何搜索存储的虚拟文件系统，请参阅如何：在独立存储中查找现有文件和目录。

示例

下面的代码示例创建并删除多个目录和文件。

```
using namespace System;
using namespace System::IO::IsolatedStorage;
using namespace System::IO;

public ref class DeletingFilesDirectories
{
public:
    static void Main()
    {
        // Get a new isolated store for this user domain and assembly.
        // Put the store into an isolatedStorageFile object.

        IsolatedStorageFile^ isoStore =  IsolatedStorageFile::GetStore(IsolatedStorageScope::User |
            IsolatedStorageScope::Domain | IsolatedStorageScope::Assembly, (Type ^)nullptr, (Type
^)nullptr);

        Console::WriteLine("Creating Directories:");

        // This code creates several different directories.

        isoStore->CreateDirectory("TopLevelDirectory");
        Console::WriteLine("TopLevelDirectory");
        isoStore->CreateDirectory("TopLevelDirectory/SecondLevel");
        Console::WriteLine("TopLevelDirectory/SecondLevel");

        // This code creates two new directories, one inside the other.

        isoStore->CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        Console::WriteLine("AnotherTopLevelDirectory/InsideDirectory");
        Console::WriteLine();

        // This code creates a few files and places them in the directories.

        Console::WriteLine("Creating Files:");

        // This file is placed in the root.

        IsolatedStorageFileStream^ isoStream1 = gcnew IsolatedStorageFileStream("InTheRoot.txt",
            FileMode::Create, isoStore);
        Console::WriteLine("InTheRoot.txt");
```

```
        isoStream1->Close();

        // This file is placed in the InsideDirectory.

        IsolatedStorageFileStream^ isoStream2 = gcnew IsolatedStorageFileStream(
            "AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt", FileMode::Create, isoStore);
        Console::WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console::WriteLine();

        isoStream2->Close();

        Console::WriteLine("Deleting File:");

        // This code deletes the HereIAm.txt file.
        isoStore->DeleteFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console::WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console::WriteLine();

        Console::WriteLine("Deleting Directory:");

        // This code deletes the InsideDirectory.

        isoStore->DeleteDirectory("AnotherTopLevelDirectory/InsideDirectory/");
        Console::WriteLine("AnotherTopLevelDirectory/InsideDirectory/");
        Console::WriteLine();

    } // End of main.
};

int main()
{
    DeletingFilesDirectories::Main();
}
```

```
using System;
using System.IO.IsolatedStorage;
using System.IO;

public class DeletingFilesDirectories
{
    public static void Main()
    {
        // Get a new isolated store for this user domain and assembly.
        // Put the store into an isolatedStorageFile object.

        IsolatedStorageFile isoStore =  IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Domain | IsolatedStorageScope.Assembly, null, null);

        Console.WriteLine("Creating Directories:");

        // This code creates several different directories.

        isoStore.CreateDirectory("TopLevelDirectory");
        Console.WriteLine("TopLevelDirectory");
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel");
        Console.WriteLine("TopLevelDirectory/SecondLevel");

        // This code creates two new directories, one inside the other.

        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory");
        Console.WriteLine();

        // This code creates a few files and places them in the directories.

        Console.WriteLine("Creating Files:");

        // This file is placed in the root.

        IsolatedStorageFileStream isoStream1 = new IsolatedStorageFileStream("InTheRoot.txt",
            FileMode.Create, isoStore);
        Console.WriteLine("InTheRoot.txt");

        isoStream1.Close();

        // This file is placed in the InsideDirectory.

        IsolatedStorageFileStream isoStream2 = new IsolatedStorageFileStream(
            "AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt", FileMode.Create, isoStore);
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine();

        isoStream2.Close();

        Console.WriteLine("Deleting File:");

        // This code deletes the HereIAm.txt file.
        isoStore.DeleteFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt");
        Console.WriteLine();

        Console.WriteLine("Deleting Directory:");

        // This code deletes the InsideDirectory.

        isoStore.DeleteDirectory("AnotherTopLevelDirectory/InsideDirectory/");
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/");
        Console.WriteLine();
    } // End of main.
}
```

```
Imports System.IO.IsolatedStorage
Imports System.IO

Public Class DeletingFilesDirectories
    Public Shared Sub Main()
        ' Get a new isolated store for this user domain and assembly.
        ' Put the store into an isolatedStorageFile object.

        Dim isoStore As IsolatedStorageFile = IsolatedStorageFile.GetStore(IsolatedStorageScope.User Or
            IsolatedStorageScope.Domain Or IsolatedStorageScope.Assembly, Nothing, Nothing)

        Console.WriteLine("Creating Directories:")

        ' This code creates several different directories.

        isoStore.CreateDirectory("TopLevelDirectory")
        Console.WriteLine("TopLevelDirectory")
        isoStore.CreateDirectory("TopLevelDirectory/SecondLevel")
        Console.WriteLine("TopLevelDirectory/SecondLevel")

        ' This code creates two new directories, one inside the other.

        isoStore.CreateDirectory("AnotherTopLevelDirectory/InsideDirectory")
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory")
        Console.WriteLine()

        ' This code creates a few files and places them in the directories.

        Console.WriteLine("Creating Files:")

        ' This file is placed in the root.

        Dim isoStream1 As New IsolatedStorageFileStream("InTheRoot.txt", FileMode.Create, isoStore)
        Console.WriteLine("InTheRoot.txt")

        isoStream1.Close()

        ' This file is placed in the InsideDirectory.

        Dim isoStream2 As New IsolatedStorageFileStream(
            "AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt", FileMode.Create, isoStore)
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt")
        Console.WriteLine()

        isoStream2.Close()

        Console.WriteLine("Deleting File:")

        ' This code deletes the HereIAm.txt file.
        isoStore.DeleteFile("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt")
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/HereIAm.txt")
        Console.WriteLine()

        Console.WriteLine("Deleting Directory:")

        ' This code deletes the InsideDirectory.

        isoStore.DeleteDirectory("AnotherTopLevelDirectory/InsideDirectory/")
        Console.WriteLine("AnotherTopLevelDirectory/InsideDirectory/")
        Console.WriteLine()

    End Sub
End Class
```

另请参阅

- System.IO.IsolatedStorage.IsolatedStorageFile
- 独立存储

.NET 中的管道操作

2021/11/16 •

管道为进程间通信提供了平台。管道分为两种类型：

- 匿名管道。

  匿名管道在本地计算机上提供进程间通信。与命名管道相比，虽然匿名管道需要的开销更少，但提供的服务有限。匿名管道是单向的，不能通过网络使用。仅支持一个服务器实例。匿名管道可用于线程间通信，也可用于父进程和子进程之间的通信，因为管道句柄可以轻松传递给所创建的子进程。

  在 .NET 中，可通过使用 AnonymousPipeServerStream 和 AnonymousPipeClientStream 类来实现匿名管道。

  请参阅如何：使用匿名管道进行本地进程间通信。

- 命名管道。

  命名管道在管道服务器和一个或多个管道客户端之间提供进程间通信。命名管道可以是单向的，也可以是双向的。它们支持基于消息的通信，并允许多个客户端使用相同的管道名称同时连接到服务器进程。命名管道还支持模拟，这样连接进程就可以在远程服务器上使用自己的权限。

  在 .NET 中，可通过使用 NamedPipeServerStream 和 NamedPipeClientStream 类来实现命名管道。

  请参阅如何：使用命名管道进行网络进程间通信。

请参阅

- 文件和流 I/O
- 如何：使用匿名管道进行本地进程间通信
- 如何：使用命名管道进行网络进程间通信

如何：使用匿名管道进行本地进程间通信

2021/11/16 •

匿名管道在本地计算机上提供进程间通信。它们提供的功能比命名管道少，但所需要的系统开销也少。使用匿名管道，可以在本地计算机上更轻松地进行进程间通信。不能使用匿名管道进行网络通信。

若要实现匿名管道，请使用 AnonymousPipeServerStream 和 AnonymousPipeClientStream 类。

示例

下面的示例展示了如何使用匿名管道将字符串从父进程发送到子进程。此示例在父进程中创建 AnonymousPipeServerStream 对象，它的 PipeDirection 值为 Out。然后，父进程使用客户端句柄创建 AnonymousPipeClientStream 对象，以创建子进程。子进程的 PipeDirection 值为 In。

接下来，父进程将用户提供的字符串发送给子进程。字符串在子进程的控制台中显示。

下面的示例展示了服务器进程。

```
#using <System.dll>
#using <System.Core.dll>

using namespace System;
using namespace System::IO;
using namespace System::IO::Pipes;
using namespace System::Diagnostics;

ref class PipeServer
{
public:
    static void Main()
    {
        Process^ pipeClient = gcnew Process();

        pipeClient->StartInfo->FileName = "pipeClient.exe";

        AnonymousPipeServerStream^ pipeServer =
            gcnew AnonymousPipeServerStream(PipeDirection::Out,
            HandleInheritability::Inheritable);

        Console::WriteLine("[SERVER] Current TransmissionMode: {0}.",
            pipeServer->TransmissionMode);

        // Pass the client process a handle to the server.
        pipeClient->StartInfo->Arguments =
            pipeServer->GetClientHandleAsString();
        pipeClient->StartInfo->UseShellExecute = false;
        pipeClient->Start();

        pipeServer->DisposeLocalCopyOfClientHandle();

        try
        {
            // Read user input and send that to the client process.
            StreamWriter^ sw = gcnew StreamWriter(pipeServer);

            sw->AutoFlush = true;
            // Send a 'sync message' and wait for client to receive it.
            sw->WriteLine("SYNC");
            pipeServer->WaitForPipeDrain();
            // Send the console input to the client process.
            Console::Write("[SERVER] Enter text: ");
            sw->WriteLine(Console::ReadLine());
            sw->Close();
        }
        // Catch the IOException that is raised if the pipe is broken
        // or disconnected.
        catch (IOException^ e)
        {
            Console::WriteLine("[SERVER] Error: {0}", e->Message);
        }
        pipeServer->Close();
        pipeClient->WaitForExit();
        pipeClient->Close();
        Console::WriteLine("[SERVER] Client quit. Server terminating.");
    }
};

int main()
{
    PipeServer::Main();
}
```

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Diagnostics;

class PipeServer
{
    static void Main()
    {
        Process pipeClient = new Process();

        pipeClient.StartInfo.FileName = "pipeClient.exe";

        using (AnonymousPipeServerStream pipeServer =
            new AnonymousPipeServerStream(PipeDirection.Out,
            HandleInheritability.Inheritable))
        {
            Console.WriteLine("[SERVER] Current TransmissionMode: {0}.",
                pipeServer.TransmissionMode);

            // Pass the client process a handle to the server.
            pipeClient.StartInfo.Arguments =
                pipeServer.GetClientHandleAsString();
            pipeClient.StartInfo.UseShellExecute = false;
            pipeClient.Start();

            pipeServer.DisposeLocalCopyOfClientHandle();

            try
            {
                // Read user input and send that to the client process.
                using (StreamWriter sw = new StreamWriter(pipeServer))
                {
                    sw.AutoFlush = true;
                    // Send a 'sync message' and wait for client to receive it.
                    sw.WriteLine("SYNC");
                    pipeServer.WaitForPipeDrain();
                    // Send the console input to the client process.
                    Console.Write("[SERVER] Enter text: ");
                    sw.WriteLine(Console.ReadLine());
                }
            }
            // Catch the IOException that is raised if the pipe is broken
            // or disconnected.
            catch (IOException e)
            {
                Console.WriteLine("[SERVER] Error: {0}", e.Message);
            }
        }

        pipeClient.WaitForExit();
        pipeClient.Close();
        Console.WriteLine("[SERVER] Client quit. Server terminating.");
    }
}
```

```
Imports System.IO
Imports System.IO.Pipes
Imports System.Diagnostics

Class PipeServer
    Shared Sub Main()
        Dim pipeClient As New Process()

        pipeClient.StartInfo.FileName = "pipeClient.exe"

        Using pipeServer As New AnonymousPipeServerStream(PipeDirection.Out, _
            HandleInheritability.Inheritable)

            Console.WriteLine("[SERVER] Current TransmissionMode: {0}.",
                pipeServer.TransmissionMode)

            ' Pass the client process a handle to the server.
            pipeClient.StartInfo.Arguments = pipeServer.GetClientHandleAsString()
            pipeClient.StartInfo.UseShellExecute = false
            pipeClient.Start()

            pipeServer.DisposeLocalCopyOfClientHandle()

            Try
                ' Read user input and send that to the client process.
                Using sw As New StreamWriter(pipeServer)
                    sw.AutoFlush = true
                    ' Send a 'sync message' and wait for client to receive it.
                    sw.WriteLine("SYNC")
                    pipeServer.WaitForPipeDrain()
                    ' Send the console input to the client process.
                    Console.Write("[SERVER] Enter text: ")
                    sw.WriteLine(Console.ReadLine())
                End Using
            Catch e As IOException
                ' Catch the IOException that is raised if the pipe is broken
                ' or disconnected.
                Console.WriteLine("[SERVER] Error: {0}", e.Message)
            End Try
        End Using

        pipeClient.WaitForExit()
        pipeClient.Close()
        Console.WriteLine("[SERVER] Client quit. Server terminating.")
    End Sub
End Class
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

示例

下面的示例展示了客户端进程。服务器进程启动客户端进程，并为此进程提供客户端句柄。客户端代码生成的可执行文件应命名为 `pipeClient.exe`，并在运行服务器进程前复制到服务器可执行文件所在的相同目录中。

```
#using <System.Core.dll>

using namespace System;
using namespace System::IO;
using namespace System::IO::Pipes;

ref class PipeClient
{
public:
    static void Main(array<String^>^ args)
    {
        if (args->Length > 1)
        {
            PipeStream^ pipeClient = gcnew AnonymousPipeClientStream(PipeDirection::In, args[1]);

            Console::WriteLine("[CLIENT] Current TransmissionMode: {0}.",
                pipeClient->TransmissionMode);

            StreamReader^ sr = gcnew StreamReader(pipeClient);

            // Display the read text to the console
            String^ temp;

            // Wait for 'sync message' from the server.
            do
            {
                Console::WriteLine("[CLIENT] Wait for sync...");
                temp = sr->ReadLine();
            }
            while (!temp->StartsWith("SYNC"));

            // Read the server data and echo to the console.
            while ((temp = sr->ReadLine()) != nullptr)
            {
                Console::WriteLine("[CLIENT] Echo: " + temp);
            }
            sr->Close();
            pipeClient->Close();
        }
        Console::Write("[CLIENT] Press Enter to continue...");
        Console::ReadLine();
    }
};

int main()
{
    array<String^>^ args = Environment::GetCommandLineArgs();
    PipeClient::Main(args);
}
```

```
using System;
using System.IO;
using System.IO.Pipes;

class PipeClient
{
    static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            using (PipeStream pipeClient =
                new AnonymousPipeClientStream(PipeDirection.In, args[0]))
            {
                Console.WriteLine("[CLIENT] Current TransmissionMode: {0}.",
                   pipeClient.TransmissionMode);

                using (StreamReader sr = new StreamReader(pipeClient))
                {
                    // Display the read text to the console
                    string temp;

                    // Wait for 'sync message' from the server.
                    do
                    {
                        Console.WriteLine("[CLIENT] Wait for sync...");
                        temp = sr.ReadLine();
                    }
                    while (!temp.StartsWith("SYNC"));

                    // Read the server data and echo to the console.
                    while ((temp = sr.ReadLine()) != null)
                    {
                        Console.WriteLine("[CLIENT] Echo: " + temp);
                    }
                }
            }
        }
        Console.Write("[CLIENT] Press Enter to continue...");
        Console.ReadLine();
    }
}
```

```
Imports System.IO
Imports System.IO.Pipes

Class PipeClient
    Shared Sub Main(args() as String)
        If args.Length > 0 Then
            Using pipeClient As New AnonymousPipeClientStream(PipeDirection.In, args(0))
                Console.WriteLine("[CLIENT] Current TransmissionMode: {0}.", _
                   pipeClient.TransmissionMode)

                Using sr As New StreamReader(pipeClient)
                    ' Display the read text to the console
                    Dim temp As String

                    ' Wait for 'sync message' from the server.
                    Do
                        Console.WriteLine("[CLIENT] Wait for sync...")
                        temp = sr.ReadLine()
                    Loop While temp.StartsWith("SYNC") = False

                    ' Read the server data and echo to the console.
                    temp = sr.ReadLine()
                    While Not temp = Nothing
                        Console.WriteLine("[CLIENT] Echo: " + temp)
                        temp = sr.ReadLine()
                    End While
                End Using
            End Using
        End If
        Console.Write("[CLIENT] Press Enter to continue...")
        Console.ReadLine()
    End Sub
End Class
```

请参阅

- 管道
- 如何:使用命名管道进行网络进程间通信

如何：使用命名管道进行网络进程间通信

2021/11/16 •

命名管道在管道服务器和一个或多个管道客户端之间提供进程间通信。它们比匿名管道(用于在本地计算机上提供进程间的通信)提供更多的功能。命名管道支持跨网络和多个服务器实例的全双工通信、基于消息的通信以及客户端模拟，这样连接进程便可在远程服务器上使用自己的权限集。

若要实现名称管道，请使用 NamedPipeServerStream 和 NamedPipeClientStream 类。

示例

下面的示例展示了如何使用 NamedPipeServerStream 类创建命名管道。在此示例中，服务器进程创建四个线程。每个线程都可以接受客户端连接。然后，连接的客户端进程向服务器提供文件名。如果客户端拥有足够的权限，服务器进程就会打开文件，并将它的内容发送回客户端。

```
#using <System.Core.dll>

using namespace System;
using namespace System::IO;
using namespace System::IO::Pipes;
using namespace System::Text;
using namespace System::Threading;

// Defines the data protocol for reading and writing strings on our stream
public ref class StreamString
{
private:
    Stream^ ioStream;
    UnicodeEncoding^ streamEncoding;

public:
    StreamString(Stream^ ioStream)
    {
        this->ioStream = ioStream;
        streamEncoding = gcnew UnicodeEncoding();
    }

    String^ ReadString()
    {
        int len;

        len = ioStream->ReadByte() * 256;
        len += ioStream->ReadByte();
        array<Byte>^ inBuffer = gcnew array<Byte>(len);
        ioStream->Read(inBuffer, 0, len);

        return streamEncoding->GetString(inBuffer);
    }

    int WriteString(String^ outString)
    {
        array<Byte>^ outBuffer = streamEncoding->GetBytes(outString);
        int len = outBuffer->Length;
        if (len > UInt16::MaxValue)
        {
            len = (int)UInt16::MaxValue;
        }
        ioStream->WriteByte((Byte)(len / 256));
        ioStream->WriteByte((Byte)(len & 255));
        ioStream->Write(outBuffer, 0, len);
```

```
        ioStream->Flush();

        return outBuffer->Length + 2;
    }
};

// Contains the method executed in the context of the impersonated user
public ref class ReadFileToStream
{
private:
    String^ fn;
    StreamString ^ss;

public:
    ReadFileToStream(StreamString^ str, String^ filename)
    {
        fn = filename;
        ss = str;
    }

    void Start()
    {
        String^ contents = File::ReadAllText(fn);
        ss->WriteString(contents);
    }
};

public ref class PipeServer
{
private:
    static int numThreads = 4;

public:
    static void Main()
    {
        int i;
        array<Thread^>^ servers = gcnew array<Thread^>(numThreads);

        Console::WriteLine("\n*** Named pipe server stream with impersonation example ***\n");
        Console::WriteLine("Waiting for client connect...\n");
        for (i = 0; i < numThreads; i++)
        {
            servers[i] = gcnew Thread(gcnew ThreadStart(&ServerThread));
            servers[i]->Start();
        }
        Thread::Sleep(250);
        while (i > 0)
        {
            for (int j = 0; j < numThreads; j++)
            {
                if (servers[j] != nullptr)
                {
                    if (servers[j]->Join(250))
                    {
                        Console::WriteLine("Server thread[{0}] finished.", servers[j]->ManagedThreadId);
                        servers[j] = nullptr;
                        i--;    // decrement the thread watch count
                    }
                }
            }
        }
        Console::WriteLine("\nServer threads exhausted, exiting.");
    }

private:
    static void ServerThread()
    {
        NamedPipeServerStream^ pipeServer =
            gcnew NamedPipeServerStream("testpipe", PipeDirection::InOut, numThreads);
```

```
            gcnew NamedPipeServerStream("testpipe", PipeDirection.InOut, numThreads);

        int threadId = Thread::CurrentThread->ManagedThreadId;

        // Wait for a client to connect
        pipeServer->WaitForConnection();

        Console::WriteLine("Client connected on thread[{0}].", threadId);
        try
        {
            // Read the request from the client. Once the client has
            // written to the pipe its security token will be available.

            StreamString^ ss = gcnew StreamString(pipeServer);

            // Verify our identity to the connected client using a
            // string that the client anticipates.

            ss->WriteString("I am the one true server!");
            String^ filename = ss->ReadString();

            // Read in the contents of the file while impersonating the client.
            ReadFileToStream^ fileReader = gcnew ReadFileToStream(ss, filename);

            // Display the name of the user we are impersonating.
            Console::WriteLine("Reading file: {0} on thread[{1}] as user: {2}.",
                filename, threadId, pipeServer->GetImpersonationUserName());
            pipeServer->RunAsClient(gcnew PipeStreamImpersonationWorker(fileReader,
&ReadFileToStream::Start));
        }
        // Catch the IOException that is raised if the pipe is broken
        // or disconnected.
        catch (IOException^ e)
        {
            Console::WriteLine("ERROR: {0}", e->Message);
        }
        pipeServer->Close();
    }
};

int main()
{
    PipeServer::Main();
}
```

```
using System;
using System.IO;
using System.IO.Pipes;
using System.Text;
using System.Threading;

public class PipeServer
{
    private static int numThreads = 4;

    public static void Main()
    {
        int i;
        Thread[] servers = new Thread[numThreads];

        Console.WriteLine("\n*** Named pipe server stream with impersonation example ***\n");
        Console.WriteLine("Waiting for client connect...\n");
        for (i = 0; i < numThreads; i++)
        {
            servers[i] = new Thread(ServerThread);
            servers[i].Start();
        }
        Thread.Sleep(250);
```

```
        while (i > 0)
        {
            for (int j = 0; j < numThreads; j++)
            {
                if (servers[j] != null)
                {
                    if (servers[j].Join(250))
                    {
                        Console.WriteLine("Server thread[{0}] finished.", servers[j].ManagedThreadId);
                        servers[j] = null;
                        i--;    // decrement the thread watch count
                    }
                }
            }
        }
        Console.WriteLine("\nServer threads exhausted, exiting.");
    }

    private static void ServerThread(object data)
    {
        NamedPipeServerStream pipeServer =
            new NamedPipeServerStream("testpipe", PipeDirection.InOut, numThreads);

        int threadId = Thread.CurrentThread.ManagedThreadId;

        // Wait for a client to connect
        pipeServer.WaitForConnection();

        Console.WriteLine("Client connected on thread[{0}].", threadId);
        try
        {
            // Read the request from the client. Once the client has
            // written to the pipe its security token will be available.

            StreamString ss = new StreamString(pipeServer);

            // Verify our identity to the connected client using a
            // string that the client anticipates.

            ss.WriteString("I am the one true server!");
            string filename = ss.ReadString();

            // Read in the contents of the file while impersonating the client.
            ReadFileToStream fileReader = new ReadFileToStream(ss, filename);

            // Display the name of the user we are impersonating.
            Console.WriteLine("Reading file: {0} on thread[{1}] as user: {2}.",
                filename, threadId, pipeServer.GetImpersonationUserName());
            pipeServer.RunAsClient(fileReader.Start);
        }
        // Catch the IOException that is raised if the pipe is broken
        // or disconnected.
        catch (IOException e)
        {
            Console.WriteLine("ERROR: {0}", e.Message);
        }
        pipeServer.Close();
    }
}

// Defines the data protocol for reading and writing strings on our stream
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
```

```
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()
    {
        int len = 0;

        len = ioStream.ReadByte() * 256;
        len += ioStream.ReadByte();
        byte[] inBuffer = new byte[len];
        ioStream.Read(inBuffer, 0, len);

        return streamEncoding.GetString(inBuffer);
    }

    public int WriteString(string outString)
    {
        byte[] outBuffer = streamEncoding.GetBytes(outString);
        int len = outBuffer.Length;
        if (len > UInt16.MaxValue)
        {
            len = (int)UInt16.MaxValue;
        }
        ioStream.WriteByte((byte)(len / 256));
        ioStream.WriteByte((byte)(len & 255));
        ioStream.Write(outBuffer, 0, len);
        ioStream.Flush();

        return outBuffer.Length + 2;
    }
}

// Contains the method executed in the context of the impersonated user
public class ReadFileToStream
{
    private string fn;
    private StreamString ss;

    public ReadFileToStream(StreamString str, string filename)
    {
        fn = filename;
        ss = str;
    }

    public void Start()
    {
        string contents = File.ReadAllText(fn);
        ss.WriteString(contents);
    }
}
```

```
Imports System.IO
Imports System.IO.Pipes
Imports System.Text
Imports System.Threading

Public Class PipeServer
    Private Shared numThreads As Integer = 4

    Public Shared Sub Main()
        Dim i As Integer
        Dim servers(numThreads) As Thread

        Console.WriteLine(vbNewLine + "*** Named pipe server stream with impersonation example ***" +
vbNewLine)
        Console.WriteLine("Waiting for client connect..." + vbNewLine)
        For i = 0 To numThreads - 1
```

```
            servers(i) = New Thread(AddressOf ServerThread)
            servers(i).Start()
        Next i
        Thread.Sleep(250)
        While i > 0
            For j As Integer = 0 To numThreads - 1
                If Not (servers(j) Is Nothing) Then
                    if servers(j).Join(250)
                        Console.WriteLine("Server thread[{0}] finished.", servers(j).ManagedThreadId)
                        servers(j) = Nothing
                        i -= 1    ' decrement the thread watch count
                    End If
                End If
            Next j
        End While
        Console.WriteLine(vbNewLine + "Server threads exhausted, exiting.")
    End Sub

    Private Shared Sub ServerThread(data As Object)
        Dim pipeServer As New _
            NamedPipeServerStream("testpipe", PipeDirection.InOut, numThreads)

        Dim threadId As Integer = Thread.CurrentThread.ManagedThreadId

        ' Wait for a client to connect
        pipeServer.WaitForConnection()

        Console.WriteLine("Client connected on thread[{0}].", threadId)
        Try
            ' Read the request from the client. Once the client has
            ' written to the pipe its security token will be available.

            Dim ss As new StreamString(pipeServer)

            ' Verify our identity to the connected client using a
            ' string that the client anticipates.

            ss.WriteString("I am the one true server!")
            Dim filename As String = ss.ReadString()

            ' Read in the contents of the file while impersonating the client.
            Dim fileReader As New ReadFileToStream(ss, filename)

            ' Display the name of the user we are impersonating.
            Console.WriteLine("Reading file: {0} on thread[{1}] as user: {2}.",
                filename, threadId, pipeServer.GetImpersonationUserName())
            pipeServer.RunAsClient(AddressOf fileReader.Start)
            ' Catch the IOException that is raised if the pipe is broken
            ' or disconnected.
        Catch e As IOException
            Console.WriteLine("ERROR: {0}", e.Message)
        End Try
        pipeServer.Close()
    End Sub
End Class

' Defines the data protocol for reading and writing strings on our stream
Public Class StreamString
    Private ioStream As Stream
    Private streamEncoding As UnicodeEncoding

    Public Sub New(ioStream As Stream)
        Me.ioStream = ioStream
        streamEncoding = New UnicodeEncoding(False, False)
    End Sub

    Public Function ReadString() As String
        Dim len As Integer = 0
        len = CType(ioStream.ReadByte(), Integer) * 256
```

```
        len = CType(ioStream.ReadByte(), Integer) * 256
        len += CType(ioStream.ReadByte(), Integer)
        Dim inBuffer As Array = Array.CreateInstance(GetType(Byte), len)
        ioStream.Read(inBuffer, 0, len)

        Return streamEncoding.GetString(inBuffer)
    End Function

    Public Function WriteString(outString As String) As Integer
        Dim outBuffer() As Byte = streamEncoding.GetBytes(outString)
        Dim len As Integer = outBuffer.Length
        If len > UInt16.MaxValue Then
            len = CType(UInt16.MaxValue, Integer)
        End If
        ioStream.WriteByte(CType(len \ 256, Byte))
        ioStream.WriteByte(CType(len And 255, Byte))
        ioStream.Write(outBuffer, 0, outBuffer.Length)
        ioStream.Flush()

        Return outBuffer.Length + 2
    End Function
End Class

' Contains the method executed in the context of the impersonated user
Public Class ReadFileToStream
    Private fn As String
    Private ss As StreamString

    Public Sub New(str As StreamString, filename As String)
        fn = filename
        ss = str
    End Sub

    Public Sub Start()
        Dim contents As String = File.ReadAllText(fn)
        ss.WriteString(contents)
    End Sub
End Class
```

示例

下面的示例展示了使用 NamedPipeClientStream 类的客户端进程。客户端连接到服务器进程，并将文件名发送到服务器。因为此示例使用模拟，所以运行客户端应用的标识必须有权访问文件。然后，服务器将文件内容发送回客户端。接下来，文件内容在控制台中显示。

```
using System;
using System.Diagnostics;
using System.IO;
using System.IO.Pipes;
using System.Security.Principal;
using System.Text;
using System.Threading;

public class PipeClient
{
    private static int numClients = 4;

    public static void Main(string[] args)
    {
        if (args.Length > 0)
        {
            if (args[0] == "spawnclient")
            {
                var pipeClient =
                    new NamedPipeClientStream(".", "testpipe",
                        PipeDirection.InOut, PipeOptions.None,
```

```
                    TokenImpersonationLevel.Impersonation);

            Console.WriteLine("Connecting to server...\n");
            pipeClient.Connect();

            var ss = new StreamString(pipeClient);
            // Validate the server's signature string.
            if (ss.ReadString() == "I am the one true server!")
            {
                // The client security token is sent with the first write.
                // Send the name of the file whose contents are returned
                // by the server.
                ss.WriteString("c:\\textfile.txt");

                // Print the file to the screen.
                Console.Write(ss.ReadString());
            }
            else
            {
                Console.WriteLine("Server could not be verified.");
            }
            pipeClient.Close();
            // Give the client process some time to display results before exiting.
            Thread.Sleep(4000);
        }
    }
    else
    {
        Console.WriteLine("\n*** Named pipe client stream with impersonation example ***\n");
        StartClients();
    }
}

// Helper function to create pipe client processes
private static void StartClients()
{
    string currentProcessName = Environment.CommandLine;

    // Remove extra characters when launched from Visual Studio
    currentProcessName = currentProcessName.Trim('"', ' ');

    currentProcessName = Path.ChangeExtension(currentProcessName, ".exe");
    Process[] plist = new Process[numClients];

    Console.WriteLine("Spawning client processes...\n");

    if (currentProcessName.Contains(Environment.CurrentDirectory))
    {
        currentProcessName = currentProcessName.Replace(Environment.CurrentDirectory, String.Empty);
    }

    // Remove extra characters when launched from Visual Studio
    currentProcessName = currentProcessName.Replace("\\", String.Empty);
    currentProcessName = currentProcessName.Replace("\"", String.Empty);

    int i;
    for (i = 0; i < numClients; i++)
    {
        // Start 'this' program but spawn a named pipe client.
        plist[i] = Process.Start(currentProcessName, "spawnclient");
    }
    while (i > 0)
    {
        for (int j = 0; j < numClients; j++)
        {
            if (plist[j] != null)
            {
                if (plist[j].HasExited)
                {
```

```
                    Console.WriteLine($"Client process[{plist[j].Id}] has exited.");
                    plist[j] = null;
                    i--;    // decrement the process watch count
                }
                else
                {
                    Thread.Sleep(250);
                }
            }
        }
    }
    Console.WriteLine("\nClient processes finished, exiting.");
  }
}

// Defines the data protocol for reading and writing strings on our stream.
public class StreamString
{
    private Stream ioStream;
    private UnicodeEncoding streamEncoding;

    public StreamString(Stream ioStream)
    {
        this.ioStream = ioStream;
        streamEncoding = new UnicodeEncoding();
    }

    public string ReadString()
    {
        int len;
        len = ioStream.ReadByte() * 256;
        len += ioStream.ReadByte();
        var inBuffer = new byte[len];
        ioStream.Read(inBuffer, 0, len);

        return streamEncoding.GetString(inBuffer);
    }

    public int WriteString(string outString)
    {
        byte[] outBuffer = streamEncoding.GetBytes(outString);
        int len = outBuffer.Length;
        if (len > UInt16.MaxValue)
        {
            len = (int)UInt16.MaxValue;
        }
        ioStream.WriteByte((byte)(len / 256));
        ioStream.WriteByte((byte)(len & 255));
        ioStream.Write(outBuffer, 0, len);
        ioStream.Flush();

        return outBuffer.Length + 2;
    }
}
```

```
Imports System.Diagnostics
Imports System.IO
Imports System.IO.Pipes
Imports System.Security.Principal
Imports System.Text
Imports System.Threading

Public Class PipeClient
    Private Shared numClients As Integer = 4

    Public Shared Sub Main(args() As String)
        If args.Length > 0 Then
```

```
            If args(0) = "spawnclient" Then
                Dim pipeClient As New NamedPipeClientStream( _
                    ".", "testpipe", _
                    PipeDirection.InOut, PipeOptions.None, _
                    TokenImpersonationLevel.Impersonation)

                Console.WriteLine("Connecting to server..." + vbNewLine)
                pipeClient.Connect()

                Dim ss As New StreamString(pipeClient)
                ' Validate the server's signature string.
                If ss.ReadString() = "I am the one true server!" Then
                    ' The client security token is sent with the first write.
                    ' Send the name of the file whose contents are returned
                    ' by the server.
                    ss.WriteString("c:\textfile.txt")

                    ' Print the file to the screen.
                    Console.Write(ss.ReadString())
                Else
                    Console.WriteLine("Server could not be verified.")
                End If
                pipeClient.Close()
                ' Give the client process some time to display results before exiting.
                Thread.Sleep(4000)
            End If
        Else
            Console.WriteLine(vbNewLine + "*** Named pipe client stream with impersonation example ***" +
vbNewLine)
            StartClients()
        End If
    End Sub

    ' Helper function to create pipe client processes
    Private Shared Sub StartClients()
        Dim currentProcessName As String = Environment.CommandLine
        Dim plist(numClients - 1) As Process

        Console.WriteLine("Spawning client processes..." + vbNewLine)

        If currentProcessName.Contains(Environment.CurrentDirectory) Then
            currentProcessName = currentProcessName.Replace(Environment.CurrentDirectory, String.Empty)
        End If

        ' Remove extra characters when launched from Visual Studio.
        currentProcessName = currentProcessName.Replace("\", String.Empty)
        currentProcessName = currentProcessName.Replace("""", String.Empty)

        ' Change extension for .NET Core "dotnet run" returns the DLL, not the host exe.
        currentProcessName = Path.ChangeExtension(currentProcessName, ".exe")

        Dim i As Integer
        For i = 0 To numClients - 1
            ' Start 'this' program but spawn a named pipe client.
            plist(i) = Process.Start(currentProcessName, "spawnclient")
        Next
        While i > 0
            For j As Integer = 0 To numClients - 1
                If plist(j) IsNot Nothing Then
                    If plist(j).HasExited Then
                        Console.WriteLine($"Client process[{plist(j).Id}] has exited.")
                        plist(j) = Nothing
                        i -= 1    ' decrement the process watch count
                    Else
                        Thread.Sleep(250)
                    End If
                End If
            Next
        End While
```

```
        Console.WriteLine(vbNewLine + "Client processes finished, exiting.")
    End Sub
End Class

' Defines the data protocol for reading and writing strings on our stream
Public Class StreamString
    Private ioStream As Stream
    Private streamEncoding As UnicodeEncoding

    Public Sub New(ioStream As Stream)
        Me.ioStream = ioStream
        streamEncoding = New UnicodeEncoding(False, False)
    End Sub

    Public Function ReadString() As String
        Dim len As Integer = 0
        len = CType(ioStream.ReadByte(), Integer) * 256
        len += CType(ioStream.ReadByte(), Integer)
        Dim inBuffer As Array = Array.CreateInstance(GetType(Byte), len)
        ioStream.Read(inBuffer, 0, len)

        Return streamEncoding.GetString(inBuffer)
    End Function

    Public Function WriteString(outString As String) As Integer
        Dim outBuffer As Byte() = streamEncoding.GetBytes(outString)
        Dim len As Integer = outBuffer.Length
        If len > UInt16.MaxValue Then
            len = CType(UInt16.MaxValue, Integer)
        End If
        ioStream.WriteByte(CType(len \ 256, Byte))
        ioStream.WriteByte(CType(len And 255, Byte))
        ioStream.Write(outBuffer, 0, outBuffer.Length)
        ioStream.Flush()

        Return outBuffer.Length + 2
    End Function
End Class
```

可靠编程

因为此示例中的客户端进程和服务器进程可以在同一台计算机上运行，所以提供给 NamedPipeClientStream 对象的服务器名称为 `"."`。如果客户端进程和服务器进程在不同的计算机上运行，`"."` 会被替换为运行服务器进程的计算机的网络名称。

请参阅

- TokenImpersonationLevel
- GetImpersonationUserName
- 管道
- 如何：使用匿名管道进行本地进程间通信

.NET 中的 System.IO.Pipelines

2021/11/16 •

System.IO.Pipelines 是一个新库，旨在使在 .NET 中执行高性能 I/O 更加容易。该库的目标为适用于所有 .NET 实现的 .NET Standard。

System.IO.Pipelines 解决什么问题

分析流数据的应用由样板代码组成，后者由许多专门且不寻常的代码流组成。样板代码和特殊情况代码很复杂且难以进行维护。

`System.IO.Pipelines` 已构建为：

- 具有高性能的流数据分析功能。
- 减少代码复杂性。

下面的代码是典型的 TCP 服务器，它从客户机接收行分隔的消息（由 `'\n'` 分隔）：

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    var buffer = new byte[1024];
    await stream.ReadAsync(buffer, 0, buffer.Length);

    // Process a single line from the buffer
    ProcessLine(buffer);
}
```

前面的代码有几个问题：

- 单次调用 `ReadAsync` 可能无法接收整条消息（行尾）。
- 忽略了 `stream.ReadAsync` 的结果。`stream.ReadAsync` 返回读取的数据量。
- 它不能处理在单个 `ReadAsync` 调用中读取多行的情况。
- 它为每次读取分配一个 `byte` 数组。

要解决上述问题，需要进行以下更改：

- 缓冲传入的数据，直到找到新行。
- 分析缓冲区中返回的所有行。
- 该行可能大于 1KB（1024 字节）。此代码需要调整输入缓冲区的大小，直到找到分隔符后，才能在缓冲区内容纳完整行。
    - 如果调整缓冲区的大小，当输入中出现较长的行时，将生成更多缓冲区副本。
    - 压缩用于读取行的缓冲区，以减少空余。
- 请考虑使用缓冲池来避免重复分配内存。
- 下面的代码解决了其中一些问题：

```
async Task ProcessLinesAsync(NetworkStream stream)
{
    byte[] buffer = ArrayPool<byte>.Shared.Rent(1024);
    var bytesBuffered = 0;
    var bytesConsumed = 0;

    while (true)
    {
        // Calculate the amount of bytes remaining in the buffer.
        var bytesRemaining = buffer.Length - bytesBuffered;

        if (bytesRemaining == 0)
        {
            // Double the buffer size and copy the previously buffered data into the new buffer.
            var newBuffer = ArrayPool<byte>.Shared.Rent(buffer.Length * 2);
            Buffer.BlockCopy(buffer, 0, newBuffer, 0, buffer.Length);
            // Return the old buffer to the pool.
            ArrayPool<byte>.Shared.Return(buffer);
            buffer = newBuffer;
            bytesRemaining = buffer.Length - bytesBuffered;
        }

        var bytesRead = await stream.ReadAsync(buffer, bytesBuffered, bytesRemaining);
        if (bytesRead == 0)
        {
            // EOF
            break;
        }

        // Keep track of the amount of buffered bytes.
        bytesBuffered += bytesRead;
        var linePosition = -1;

        do
        {
            // Look for a EOL in the buffered data.
            linePosition = Array.IndexOf(buffer, (byte)'\n', bytesConsumed,
                                         bytesBuffered - bytesConsumed);

            if (linePosition >= 0)
            {
                // Calculate the length of the line based on the offset.
                var lineLength = linePosition - bytesConsumed;

                // Process the line.
                ProcessLine(buffer, bytesConsumed, lineLength);

                // Move the bytesConsumed to skip past the line consumed (including \n).
                bytesConsumed += lineLength + 1;
            }
        }
        while (linePosition >= 0);
    }
}
```

前面的代码很复杂，不能解决所识别的所有问题。高性能网络通常意味着编写非常复杂的代码以使性能最大化。`System.IO.Pipelines` 的设计目的是使编写此类代码更容易。

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

管道

Pipe 类可用于创建 `PipeWriter/PipeReader` 对。写入 `PipeWriter` 的所有数据都可用于 `PipeReader`：

```
var pipe = new Pipe();
PipeReader reader = pipe.Reader;
PipeWriter writer = pipe.Writer;
```

管道基本用法

```
async Task ProcessLinesAsync(Socket socket)
{
    var pipe = new Pipe();
    Task writing = FillPipeAsync(socket, pipe.Writer);
    Task reading = ReadPipeAsync(pipe.Reader);

    await Task.WhenAll(reading, writing);
}

async Task FillPipeAsync(Socket socket, PipeWriter writer)
{
    const int minimumBufferSize = 512;

    while (true)
    {
        // Allocate at least 512 bytes from the PipeWriter.
        Memory<byte> memory = writer.GetMemory(minimumBufferSize);
        try
        {
            int bytesRead = await socket.ReceiveAsync(memory, SocketFlags.None);
            if (bytesRead == 0)
            {
                break;
            }
            // Tell the PipeWriter how much was read from the Socket.
            writer.Advance(bytesRead);
        }
        catch (Exception ex)
        {
            LogError(ex);
            break;
        }

        // Make the data available to the PipeReader.
        FlushResult result = await writer.FlushAsync();

        if (result.IsCompleted)
        {
            break;
        }
    }

     // By completing PipeWriter, tell the PipeReader that there's no more data coming.
    await writer.CompleteAsync();
}

async Task ReadPipeAsync(PipeReader reader)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync();
        ReadOnlySequence<byte> buffer = result.Buffer;

        while (TryReadLine(ref buffer, out ReadOnlySequence<byte> line))
        {
            // Process the line.
            ProcessLine(line);
        }

        // Tell the PipeReader how much of the buffer has been consumed.
```

```
        // Tell the PipeReader how much of the buffer has been consumed.
        reader.AdvanceTo(buffer.Start, buffer.End);

        // Stop reading if there's no more data coming.
        if (result.IsCompleted)
        {
            break;
        }
    }

    // Mark the PipeReader as complete.
    await reader.CompleteAsync();
}

bool TryReadLine(ref ReadOnlySequence<byte> buffer, out ReadOnlySequence<byte> line)
{
    // Look for a EOL in the buffer.
    SequencePosition? position = buffer.PositionOf((byte)'\n');

    if (position == null)
    {
        line = default;
        return false;
    }

    // Skip the line + the \n.
    line = buffer.Slice(0, position.Value);
    buffer = buffer.Slice(buffer.GetPosition(1, position.Value));
    return true;
}
```

有两个循环：

- `FillPipeAsync` 从 `Socket` 读取并写入 `PipeWriter`。
- `ReadPipeAsync` 从 `PipeReader` 读取并分析传入的行。

没有分配显式缓冲区。所有缓冲区管理都委托给 `PipeReader` 和 `PipeWriter` 实现。委派缓冲区管理使使用代码更容易集中关注业务逻辑。

在第一个循环中：

- 调用 PipeWriter.GetMemory(Int32) 从基础编写器获取内存。
- 调用 PipeWriter.Advance(Int32) 以告知 `PipeWriter` 有多少数据已写入缓冲区。
- 调用 PipeWriter.FlushAsync 以使数据可用于 `PipeReader`。

在第二个循环中，`PipeReader` 使用由 `PipeWriter` 写入的缓冲区。缓冲区来自套接字。对 `PipeReader.ReadAsync` 的调用：

- 返回包含两条重要信息的 ReadResult：
  - 以 `ReadOnlySequence<byte>` 形式读取的数据。
  - 布尔值 `IsCompleted`，指示是否已到达数据结尾 (EOF)。

找到行尾 (EOL) 分隔符并分析该行后：

- 该逻辑处理缓冲区以跳过已处理的内容。
- 调用 `PipeReader.AdvanceTo` 以告知 `PipeReader` 已消耗和检查了多少数据。

读取器和编写器循环通过调用 `Complete` 结束。`Complete` 使基础管道释放其分配的内存。

反压和流量控制

理想情况下，读取和分析可协同工作：

- 写入线程使用来自网络的数据并将其放入缓冲区。
- 分析线程负责构造适当的数据结构。

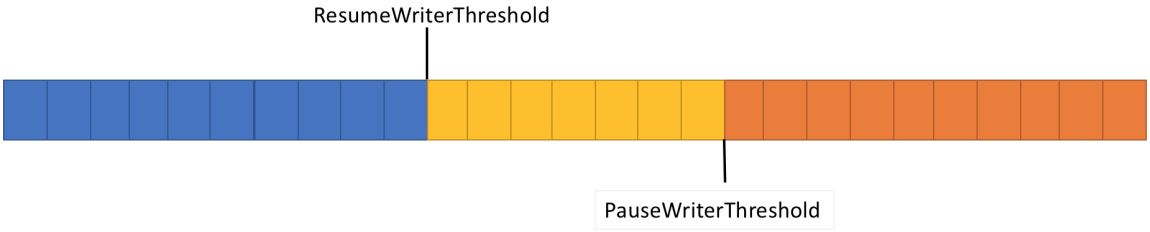
通常，分析所花费的时间比仅从网络复制数据块所用时间更长：

- 读取线程领先于分析线程。
- 读取线程必须减缓或分配更多内存来存储用于分析线程的数据。

为了获得最佳性能，需要在频繁暂停和分配更多内存之间取得平衡。

为解决上述问题，`Pipe` 提供了两个设置来控制数据流：

- PauseWriterThreshold：确定在调用 FlushAsync 暂停之前应缓冲多少数据。
- ResumeWriterThreshold：确定在恢复对 `PipeWriter.FlushAsync` 的调用之前，读取器必须观察多少数据。



PipeWriter.FlushAsync：

- 当 `Pipe` 中的数据量超过 `PauseWriterThreshold` 时，返回不完整的 `ValueTask<FlushResult>`。
- 低于 `ResumeWriterThreshold` 时，返回完整的 `ValueTask<FlushResult>`。

使用两个值可防止快速循环，如果只使用一个值，则可能发生这种循环。

示例

```
// The Pipe will start returning incomplete tasks from FlushAsync until
// the reader examines at least 5 bytes.
var options = new PipeOptions(pauseWriterThreshold: 10, resumeWriterThreshold: 5);
var pipe = new Pipe(options);
```

PipeScheduler

通常在使用 `async` 和 `await` 时，异步代码会在 TaskScheduler 或当前 SynchronizationContext 上恢复。

在执行 I/O 时，对执行 I/O 的位置进行细粒度控制非常重要。此控件允许高效利用 CPU 缓存。高效的缓存对于 Web 服务器等高性能应用至关重要。PipeScheduler 提供对异步回调运行位置的控制。默认情况下：

- 使用当前的 SynchronizationContext。
- 如果没有 `SynchronizationContext`，它将使用线程池运行回调。

```
public static void Main(string[] args)
{
    var writeScheduler = new SingleThreadPipeScheduler();
    var readScheduler = new SingleThreadPipeScheduler();

    // Tell the Pipe what schedulers to use and disable the SynchronizationContext.
    var options = new PipeOptions(readerScheduler: readScheduler,
                                  writerScheduler: writeScheduler,
                                  useSynchronizationContext: false);
    var pipe = new Pipe(options);
}

// This is a sample scheduler that async callbacks on a single dedicated thread.
public class SingleThreadPipeScheduler : PipeScheduler
{
    private readonly BlockingCollection<(Action<object> Action, object State)> _queue =
     new BlockingCollection<(Action<object> Action, object State)>();
    private readonly Thread _thread;

    public SingleThreadPipeScheduler()
    {
        _thread = new Thread(DoWork);
        _thread.Start();
    }

    private void DoWork()
    {
        foreach (var item in _queue.GetConsumingEnumerable())
        {
            item.Action(item.State);
        }
    }

    public override void Schedule(Action<object> action, object state)
    {
        _queue.Add((action, state));
    }
}
```

PipeScheduler.ThreadPool 是 PipeScheduler 实现，用于对线程池的回调进行排队。`PipeScheduler.ThreadPool` 是默认选项，通常也是最佳选项。PipeScheduler.Inline 可能会导致意外后果，如死锁。

管道重置

通常重用 `Pipe` 对象即可重置。若要重置管道，请在 `PipeReader` 和 `PipeWriter` 完成时调用 PipeReader Reset。

PipeReader

PipeReader 代表调用方管理内存。在调用 PipeReader.ReadAsync 之后始终调用 PipeReader.AdvanceTo 。这使 `PipeReader` 知道调用方何时用完内存，以便可以对其进行跟踪。从 `PipeReader.ReadAsync` 返回的 `ReadOnlySequence<byte>` 仅在调用 `PipeReader.AdvanceTo` 之前有效。调用 `PipeReader.AdvanceTo` 后，不能使用 `ReadOnlySequence<byte>` 。

`PipeReader.AdvanceTo` 采用两个 SequencePosition 参数：

- 第一个参数确定消耗的内存量。
- 第二个参数确定观察到的缓冲区数。

将数据标记为“已使用”意味着管道可以将内存返回到底层缓冲池。将数据标记为“已观察”可控制对 `PipeReader.ReadAsync` 的下一个调用的操作。将所有内容都标记为“已观察”意味着下次对 `PipeReader.ReadAsync` 的调用将不会返回，直到有更多数据写入管道。任何其他值都将使对 `PipeReader.ReadAsync` 的下一次调用立即返回并包含已观察到的和未观察到的数据，但不是已被使用的数据 。

读取流数据方案

尝试读取流数据时会出现以下几种典型模式：

- 给定数据流时，分析单条消息。
- 给定数据流时，分析所有可用消息。

以下示例使用 `TryParseMessage` 方法分析来自 `ReadOnlySequence<byte>` 的消息。`TryParseMessage` 分析单条消息并更新输入缓冲区，以从缓冲区中剪裁已分析的消息。`TryParseMessage` 不是 .NET 的一部分，它是在以下部分中使用的用户编写的方法。

```
bool TryParseMessage(ref ReadOnlySequence<byte> buffer, out Message message);
```

读取单条消息

下面的代码从 `PipeReader` 读取一条消息并将其返回给调用方。

```
async ValueTask<Message> ReadSingleMessageAsync(PipeReader reader,
 CancellationToken cancellationToken = default)
{
    while (true)
    {
        ReadResult result = await reader.ReadAsync(cancellationToken);
        ReadOnlySequence<byte> buffer = result.Buffer;

        // In the event that no message is parsed successfully, mark consumed
        // as nothing and examined as the entire buffer.
        SequencePosition consumed = buffer.Start;
        SequencePosition examined = buffer.End;

        try
        {
            if (TryParseMessage(ref buffer, out Message message))
            {
                // A single message was successfully parsed so mark the start of the
                // parsed buffer as consumed. TryParseMessage trims the buffer to
                // point to the data after the message was parsed.
                consumed = buffer.Start;

                // Examined is marked the same as consumed here, so the next call
                // to ReadSingleMessageAsync will process the next message if there's
                // one.
                examined = consumed;

                return message;
            }

            // There's no more data to be processed.
            if (result.IsCompleted)
            {
                if (buffer.Length > 0)
                {
                    // The message is incomplete and there's no more data to process.
                    throw new InvalidDataException("Incomplete message.");
                }

                break;
            }
        }
        finally
        {
            reader.AdvanceTo(consumed, examined);
        }
    }

    return null;
}
```

前面的代码：

- 分析单条消息。
- 更新已使用的 `SequencePosition` 并检查 `SequencePosition` 以指向已剪裁的输入缓冲区的开始。

因为 `TryParseMessage` 从输入缓冲区中删除了已分析的消息，所以更新了两个 `SequencePosition` 参数。通常，分析来自缓冲区的单条消息时，检查的位置应为以下位置之一：

- 消息的结尾。
- 如果未找到消息，则返回接收缓冲区的结尾。

单条消息案例最有可能出现错误。将错误的值传递给“已检查”可能会导致内存不足异常或无限循环 。有关详细信息，请参阅本文中的 PipeReader 常见问题部分。

读取多条消息

以下代码从 `PipeReader` 读取所有消息，并在每条消息上调用 `ProcessMessageAsync`。

```
async Task ProcessMessagesAsync(PipeReader reader, CancellationToken cancellationToken = default)
{
    try
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync(cancellationToken);
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                // Process all messages from the buffer, modifying the input buffer on each
                // iteration.
                while (TryParseMessage(ref buffer, out Message message))
                {
                    await ProcessMessageAsync(message);
                }

                // There's no more data to be processed.
                if (result.IsCompleted)
                {
                    if (buffer.Length > 0)
                    {
                        // The message is incomplete and there's no more data to process.
                        throw new InvalidDataException("Incomplete message.");
                    }
                    break;
                }
            }
            finally
            {
                // Since all messages in the buffer are being processed, you can use the
                // remaining buffer's Start and End position to determine consumed and examined.
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    finally
    {
        await reader.CompleteAsync();
    }
}
```

取消

`PipeReader.ReadAsync`：

- 支持传递 CancellationToken。
- 如果在读取挂起期间取消了 `CancellationToken`，则会引发 OperationCanceledException。
- 支持通过 PipeReader.CancelPendingRead 取消当前读取操作的方法，这样可以避免引发异常。调用 `PipeReader.CancelPendingRead` 将导致对 `PipeReader.ReadAsync` 的当前或下次调用返回 ReadResult，并将 `IsCanceled` 设置为 `true`。这对于以非破坏性和非异常的方式停止现有的读取循环非常有用。

```
private PipeReader reader;

public MyConnection(PipeReader reader)
{
    this.reader = reader;
}

public void Abort()
{
    // Cancel the pending read so the process loop ends without an exception.
    reader.CancelPendingRead();
}

public async Task ProcessMessagesAsync()
{
    try
    {
        while (true)
        {
            ReadResult result = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = result.Buffer;

            try
            {
                if (result.IsCanceled)
                {
                    // The read was canceled. You can quit without reading the existing data.
                    break;
                }

                // Process all messages from the buffer, modifying the input buffer on each
                // iteration.
                while (TryParseMessage(ref buffer, out Message message))
                {
                    await ProcessMessageAsync(message);
                }

                // There's no more data to be processed.
                if (result.IsCompleted)
                {
                    break;
                }
            }
            finally
            {
                // Since all messages in the buffer are being processed, you can use the
                // remaining buffer's Start and End position to determine consumed and examined.
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    finally
    {
        await reader.CompleteAsync();
    }
}
```

PipeReader 常见问题

- 将错误的值传递给 `consumed` 或 `examined` 可能会导致读取已读取的数据。
- 传递 `buffer.End` 作为检查对象可能会导致以下问题：
  - 数据停止
  - 如果数据未使用，可能最终会出现内存不足 (OOM) 异常。例如，当一次处理来自缓冲区的单条消息

时，可能会出现 `PipeReader.AdvanceTo(position, buffer.End)`。

- 将错误的值传递给 `consumed` 或 `examined` 可能会导致无限循环。例如，如果 `buffer.Start` 没有更改，则 `PipeReader.AdvanceTo(buffer.Start)` 将导致在下一个对 `PipeReader.ReadAsync` 的调用在新数据到来之前立即返回。
- 将错误的值传递给 `consumed` 或 `examined` 可能会导致无限缓冲（最终导致 OOM）。
- 在调用 `PipeReader.AdvanceTo` 之后使用 `ReadOnlySequence<byte>` 可能会导致内存损坏（在释放之后使用）。
- 未能调用 `PipeReader.Complete/CompleteAsync` 可能会导致内存泄漏。
- 在处理缓冲区之前检查 ReadResult.IsCompleted 并退出读取逻辑会导致数据丢失。循环退出条件应基于 `ReadResult.Buffer.IsEmpty` 和 `ReadResult.IsCompleted`。如果错误执行此操作，可能会导致无限循环。

有问题的代码

✕ 数据丢失

当 `IsCompleted` 被设置为 `true` 时，`ReadResult` 可能会返回最后一段数据。在退出读循环之前不读取该数据将导致数据丢失。

> **WARNING**
> 不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> dataLossBuffer = result.Buffer;

    if (result.IsCompleted)
    {
        break;
    }

    Process(ref dataLossBuffer, out Message message);

    reader.AdvanceTo(dataLossBuffer.Start, dataLossBuffer.End);
}
```

> **WARNING**
> 不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

✕ 无限循环

如果 `Result.IsCompleted` 是 `true`，则以下逻辑可能会导致无限循环，但缓冲区中永远不会有完整的消息。

> **WARNING**
> 不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;
    if (result.IsCompleted && infiniteLoopBuffer.IsEmpty)
    {
        break;
    }

    Process(ref infiniteLoopBuffer, out Message message);

    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);
}
```

WARNING

不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

下面是另一段具有相同问题的代码。该代码在检查 `ReadResult.IsCompleted` 之前检查非空缓冲区。由于该代码位于 `else if` 中，如果缓冲区中没有完整的消息，它将永远循环。

WARNING

不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> infiniteLoopBuffer = result.Buffer;

    if (!infiniteLoopBuffer.IsEmpty)
    {
        Process(ref infiniteLoopBuffer, out Message message);
    }
    else if (result.IsCompleted)
    {
        break;
    }

    reader.AdvanceTo(infiniteLoopBuffer.Start, infiniteLoopBuffer.End);
}
```

WARNING

不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

✕ 意外挂起

在分析单条消息时，如果无条件调用 `PipeReader.AdvanceTo` 而 `buffer.End` 位于 `examined` 位置，则可能导致挂起。对 `PipeReader.AdvanceTo` 的下次调用将在以下情况下返回：

- 有更多数据写入管道。

- 以及之前未检查过新数据。

> **WARNING**
> 不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> hangBuffer = result.Buffer;

    Process(ref hangBuffer, out Message message);

    if (result.IsCompleted)
    {
        break;
    }

    reader.AdvanceTo(hangBuffer.Start, hangBuffer.End);

    if (message != null)
    {
        return message;
    }
}
```

> **WARNING**
> 不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

✕ 内存不足 (OOM)

在满足以下条件的情况下，以下代码将保持缓冲，直到发生 OutOfMemoryException：

- 没有最大消息大小。
- 从 `PipeReader` 返回的数据不会生成完整的消息。例如，它不会生成完整的消息，因为另一端正在编写一条大消息（例如，一条为 4GB 的消息）。

> **WARNING**
> 不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
Environment.FailFast("This code is terrible, don't use it!");
while (true)
{
    ReadResult result = await reader.ReadAsync(cancellationToken);
    ReadOnlySequence<byte> thisCouldOutOfMemory = result.Buffer;

    Process(ref thisCouldOutOfMemory, out Message message);

    if (result.IsCompleted)
    {
        break;
    }

    reader.AdvanceTo(thisCouldOutOfMemory.Start, thisCouldOutOfMemory.End);

    if (message != null)
    {
        return message;
    }
}
```

WARNING

不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

✕ 内存损坏

当写入读取缓冲区的帮助程序时，应在调用 `Advance` 之前复制任何返回的有效负载。下面的示例将返回 `Pipe` 已丢弃的内存，并可能将其重新用于下一个操作（读/写）。

WARNING

不要使用以下代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。以下示例用于解释 PipeReader 常见问题。

```
public class Message
{
    public ReadOnlySequence<byte> CorruptedPayload { get; set; }
}
```

```
    Environment.FailFast("This code is terrible, don't use it!");
    Message message = null;

    while (true)
    {
        ReadResult result = await reader.ReadAsync(cancellationToken);
        ReadOnlySequence<byte> buffer = result.Buffer;

        ReadHeader(ref buffer, out int length);

        if (length <= buffer.Length)
        {
            message = new Message
            {
                // Slice the payload from the existing buffer
                CorruptedPayload = buffer.Slice(0, length)
            };

            buffer = buffer.Slice(length);
        }

        if (result.IsCompleted)
        {
            break;
        }

        reader.AdvanceTo(buffer.Start, buffer.End);

        if (message != null)
        {
            // This code is broken since reader.AdvanceTo() was called with a position *after* the buffer
            // was captured.
            break;
        }
    }

    return message;
}
```

WARNING

不要使用上述代码。使用此示例将导致数据丢失、挂起和安全问题，并且不应复制。前面的示例用于解释 PipeReader 常见问题。

PipeWriter

PipeWriter 管理用于代表调用方写入的缓冲区。`PipeWriter` 实现 `IBufferWriter<byte>`。`IBufferWriter<byte>` 使得无需额外的缓冲区副本就可以访问缓冲区来执行写入操作。

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken = default)
{
    // Request at least 5 bytes from the PipeWriter.
    Memory<byte> memory = writer.GetMemory(5);

    // Write directly into the buffer.
    int written = Encoding.ASCII.GetBytes("Hello".AsSpan(), memory.Span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);

    await writer.FlushAsync(cancellationToken);
}
```

之前的代码：

- 使用 GetMemory 从 `PipeWriter` 请求至少 5 个字节的缓冲区。
- 将 ASCII 字符串 `"Hello"` 的字节写入返回的 `Memory<byte>`。
- 调用 Advance 以指示写入缓冲区的字节数。
- 刷新 `PipeWriter`，以便将字节发送到基础设备。

以前的写入方法使用 `PipeWriter` 提供的缓冲区。或者，PipeWriter.WriteAsync：

- 将现有缓冲区复制到 `PipeWriter`。
- 根据需要调用 `GetSpan``Advance`，然后调用 FlushAsync。

```
async Task WriteHelloAsync(PipeWriter writer, CancellationToken cancellationToken = default)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");

    // Write helloBytes to the writer, there's no need to call Advance here
    // (Write does that).
    await writer.WriteAsync(helloBytes, cancellationToken);
}
```

取消

FlushAsync 支持传递 CancellationToken。如果令牌在刷新挂起时被取消，则传递 `CancellationToken` 将导致 `OperationCanceledException`。`PipeWriter.FlushAsync` 支持通过 PipeWriter.CancelPendingFlush 取消当前刷新操作而不引发异常的方法。调用 `PipeWriter.CancelPendingFlush` 将导致对 `PipeWriter.FlushAsync` 或 `PipeWriter.WriteAsync` 的当前或下次调用返回 FlushResult，并将 `IsCanceled` 设置为 `true`。这对于以非破坏性和非异常的方式停止暂停刷新非常有用。

PipeWriter 常见问题

- GetSpan 和 GetMemory 返回至少具有请求内存量的缓冲区。请勿假设确切的缓冲区大小 。
- 无法保证连续的调用将返回相同的缓冲区或相同大小的缓冲区。
- 在调用 Advance 之后，必须请求一个新的缓冲区来继续写入更多数据。不能写入先前获得的缓冲区。
- 如果未完成对 `FlushAsync` 的调用，则调用 `GetMemory` 或 `GetSpan` 将不安全。
- 如果未刷新数据，则调用 `Complete` 或 `CompleteAsync` 可能导致内存损坏。

IDuplexPipe

IDuplexPipe 是支持读写的类型的协定。例如，网络连接将由 `IDuplexPipe` 表示。

与包含 `PipeReader` 和 `PipeWriter` 的 `Pipe` 不同，`IDuplexPipe` 表示全双工连接的一侧。这意味着写入 `PipeWriter` 的内容不会从 `PipeReader` 中读取。

流

在读取或写入流数据时，通常使用反序列化程序读取数据，并使用序列化程序写入数据。大多数读取和写入流 API 都有一个 `Stream` 参数。为了更轻松地与这些现有 API 集成，`PipeReader` 和 `PipeWriter` 公开了一个 AsStream 方法。AsStream 返回围绕 `PipeReader` 或 `PipeWriter` 的 `Stream` 实现。

流示例

可使用给定了 Stream 对象和可选的相应创建选项的静态 `Create` 方法创建 `PipeReader` 和 `PipeWriter` 实例。

StreamPipeReaderOptions 允许使用以下参数控制 `PipeReader` 实例的创建：

- StreamPipeReaderOptions.BufferSize 是从池中租用内存时使用的最小缓冲区大小（以字节为单位），默认值为 `4096`。
- StreamPipeReaderOptions.LeaveOpen 标志确定在 `PipeReader` 完成之后基础流是否保持打开状态，默认值为 `false`。
- StreamPipeReaderOptions.MinimumReadSize 表示分配新缓冲区之前缓冲区中剩余字节的阈值，默认值为 `1024`。
- StreamPipeReaderOptions.Pool 是分配内存时使用的 `MemoryPool<byte>`，默认值为 `null`。

StreamPipeWriterOptions 允许使用以下参数控制 `PipeWriter` 实例的创建：

- StreamPipeWriterOptions.LeaveOpen 标志确定在 `PipeWriter` 完成之后基础流是否保持打开状态，默认值为 `false`。
- StreamPipeWriterOptions.MinimumBufferSize 表示从 Pool 租用内存时要使用的最小缓冲区大小，默认值为 `4096`。
- StreamPipeWriterOptions.Pool 是分配内存时使用的 `MemoryPool<byte>`，默认值为 `null`。

> **IMPORTANT**
> 使用 `Create` 方法创建 `PipeReader` 和 `PipeWriter` 实例时，需要考虑 `Stream` 对象的生存期。如果在读取器或编写器使用该方法完成操作后，你需要访问流，则需要在创建选项上将 `LeaveOpen` 标志设置为 `true`。否则，流将关闭。

以下代码演示了使用 `Create` 方法从流中创建 `PipeReader` 和 `PipeWriter` 实例。

```
using System;
using System.Buffers;
using System.IO;
using System.IO.Pipelines;
using System.Text;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        using var stream = File.OpenRead("lorem-ipsum.txt");

        var reader = PipeReader.Create(stream);
        var writer = PipeWriter.Create(
            Console.OpenStandardOutput(),
            new StreamPipeWriterOptions(leaveOpen: true));

        WriteUserCancellationPrompt();

        var processMessagesTask = ProcessMessagesAsync(reader, writer);
        var userCanceled = false;
        var cancelProcessingTask = Task.Run(() =>
        {
            while (char.ToUpperInvariant(Console.ReadKey().KeyChar) != 'C')
```

```
        while (char.ToUpperInvariant(Console.ReadKey().KeyChar) != 'C')
        {
            WriteUserCancellationPrompt();
        }

        userCanceled = true;

        // No exceptions thrown
        reader.CancelPendingRead();
        writer.CancelPendingFlush();
    });

    await Task.WhenAny(cancelProcessingTask, processMessagesTask);

    Console.WriteLine(
        $"\n\nProcessing {(userCanceled ? "cancelled" : "completed")}.\n");
}

static void WriteUserCancellationPrompt() =>
    Console.WriteLine("Press 'C' to cancel processing...\n");

static async Task ProcessMessagesAsync(
    PipeReader reader,
    PipeWriter writer)
{
    try
    {
        while (true)
        {
            ReadResult readResult = await reader.ReadAsync();
            ReadOnlySequence<byte> buffer = readResult.Buffer;

            try
            {
                if (readResult.IsCanceled)
                {
                    break;
                }

                if (TryParseMessage(ref buffer, out string message))
                {
                    FlushResult flushResult =
                        await WriteMessagesAsync(writer, message);

                    if (flushResult.IsCanceled || flushResult.IsCompleted)
                    {
                        break;
                    }
                }

                if (readResult.IsCompleted)
                {
                    if (!buffer.IsEmpty)
                    {
                        throw new InvalidDataException("Incomplete message.");
                    }
                    break;
                }
            }
            finally
            {
                reader.AdvanceTo(buffer.Start, buffer.End);
            }
        }
    }
    catch (Exception ex)
    {
        Console.Error.WriteLine(ex);
    }
```

```
        finally
        {
            await reader.CompleteAsync();
            await writer.CompleteAsync();
        }
    }

    static bool TryParseMessage(
        ref ReadOnlySequence<byte> buffer,
        out string message) =>
        (message = Encoding.ASCII.GetString(buffer)) != null;

    static ValueTask<FlushResult> WriteMessagesAsync(
        PipeWriter writer,
        string message) =>
        writer.WriteAsync(Encoding.ASCII.GetBytes(message));
}
```

应用程序使用 StreamReader 将 lorem-ipsum.txt 文件作为流进行读取。FileStream 传递给 PipeReader.Create，后者实例化 `PipeReader` 对象。然后，控制台应用程序使用 Console.OpenStandardOutput() 将其标准输出流传递到 PipeWriter.Create。示例支持取消。

使用 .NET 中的缓冲区

2021/11/16 •

本文概述了有助于读取跨多个缓冲区运行的数据的类型。它们主要用于支持 PipeReader 对象。

IBufferWriter<T>

System.Buffers.IBufferWriter<T> 是同步缓冲写入的协定。在最低级别上，接口：

- 是基本的，不难使用。
- 允许访问 Memory<T> 或 Span<T>。可以写入 `Memory<T>` 或 `Span<T>`，而且你可以确定写入了多少个 `T` 项。

```
void WriteHello(IBufferWriter<byte> writer)
{
    // Request at least 5 bytes.
    Span<byte> span = writer.GetSpan(5);
    ReadOnlySpan<char> helloSpan = "Hello".AsSpan();
    int written = Encoding.ASCII.GetBytes(helloSpan, span);

    // Tell the writer how many bytes were written.
    writer.Advance(written);
}
```

前面的方法：

- 使用 `GetSpan(5)` 从 `IBufferWriter<byte>` 请求至少 5 个字节的缓冲区。
- 将 ASCII 字符串“Hello”的字节写入返回的 `Span<byte>`。
- 调用 IBufferWriter<T> 以指示写入缓冲区的字节数。

此写入方法使用 `IBufferWriter<T>` 提供的 `Memory<T>` / `Span<T>` 缓冲区。或者，可以使用 Write 扩展方法将现有缓冲区复制到 `IBufferWriter<T>`。`Write` 根据需要调用 `GetSpan` / `Advance`，因此在写入后无需调用 `Advance`：

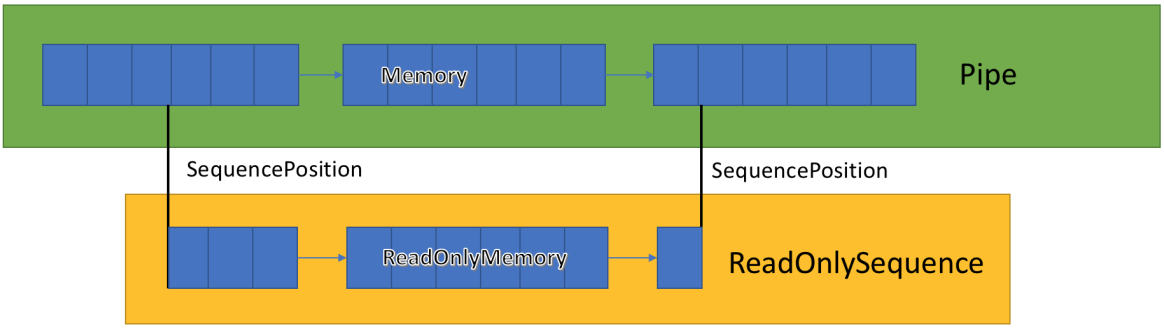
```
void WriteHello(IBufferWriter<byte> writer)
{
    byte[] helloBytes = Encoding.ASCII.GetBytes("Hello");

    // Write helloBytes to the writer. There's no need to call Advance here
    // since Write calls Advance.
    writer.Write(helloBytes);
}
```

ArrayBufferWriter<T> 是 `IBufferWriter<T>` 的实现，其后备存储是单个连续数组。

IBufferWriter 常见问题

- `GetSpan` 和 `GetMemory` 返回至少具有请求内存量的缓冲区。请勿假设确切的缓冲区大小。
- 无法保证连续的调用将返回相同的缓冲区或相同大小的缓冲区。
- 在调用 `Advance` 之后，必须请求一个新的缓冲区来继续写入更多数据。调用 `Advance` 后无法写入先前获取的缓冲区。

ReadOnlySequence<T>



ReadOnlySequence<T> 是一个可以表示 `T` 的连续或非连续序列的结构。它通过以下方法进行构造：

1. 一个 `T[]`
2. 一个 `ReadOnlyMemory<T>`
3. 一对链接列表节点 ReadOnlySequenceSegment<T> 和索引，用于表示序列的开始位置和结束位置。

第三种表示形式最值得关注，因为它对 `ReadOnlySequence<T>` 上的各种操作有性能影响：

`T[]` / `ReadOnlyMemory<T>`	`Length`	`O(1)`
`T[]` / `ReadOnlyMemory<T>`	`GetPosition(long)`	`O(1)`
`T[]` / `ReadOnlyMemory<T>`	`Slice(int, int)`	`O(1)`
`T[]` / `ReadOnlyMemory<T>`	`Slice(SequencePostion, SequencePostion)`	`O(1)`
`ReadOnlySequenceSegment<T>`	`Length`	`O(1)`
`ReadOnlySequenceSegment<T>`	`GetPosition(long)`	`O(number of segments)`
`ReadOnlySequenceSegment<T>`	`Slice(int, int)`	`O(number of segments)`
`ReadOnlySequenceSegment<T>`	`Slice(SequencePostion, SequencePostion)`	`O(1)`

由于这种混合表示形式，`ReadOnlySequence<T>` 将索引作为 `SequencePosition`（而不是整数）公开。`SequencePosition`：

- 是一个不透明的值，该值将索引表示为其起源的 `ReadOnlySequence<T>`。
- 由两个部分组成：整数和对象。这两个值的含义与 `ReadOnlySequence<T>` 的实现相关联。

访问数据

`ReadOnlySequence<T>` 将数据作为 `ReadOnlyMemory<T>` 的枚举公开。可以使用基本 foreach 来枚举每个段：

```
long FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    long position = 0;

    foreach (ReadOnlyMemory<byte> segment in buffer)
    {
        ReadOnlySpan<byte> span = segment.Span;
        var index = span.IndexOf(data);
        if (index != -1)
        {
            return position + index;
        }

        position += span.Length;
    }

    return -1;
}
```

前面的方法搜索特定字节的每个段。如果需要跟踪每个段的 `SequencePosition`，则 ReadOnlySequence<T>.TryGet 更为合适。下一个示例更改前面的代码，以返回 `SequencePosition` 而不是整数。返回 `SequencePosition` 的好处是使调用方能够避免第二次扫描以获取特定索引处的数据。

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    SequencePosition position = buffer.Start;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;
        var index = span.IndexOf(data);
        if (index != -1)
        {
            return buffer.GetPosition(position, index);
        }
    }
    return null;
}
```

`SequencePosition` 和 `TryGet` 的组合类似于枚举器。将在每次迭代开始时修改位置字段，使其为 `ReadOnlySequence<T>` 中的每个段的开头。

前面的方法作为 `ReadOnlySequence<T>` 的扩展方法存在。PositionOf 可用于简化前面的代码：

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data) => buffer.PositionOf(data);
```

处理 ReadOnlySequence<T>

处理 `ReadOnlySequence<T>` 可能比较困难，因为数据可能会拆分到序列中的多个段。为了获得最佳性能，请将代码拆分为两个路径：

- 处理单段情况的快速路径。
- 处理拆分到各个段中的数据的慢速路径。

可以使用多种方法来处理多段序列中的数据：

- 使用 `SequenceReader<T>`。
- 逐段分析数据，同时跟踪已分析的段内的 `SequencePosition` 和索引。这样可以避免不必要的分配，但可能效率会很低，尤其是对于小型缓冲区。
- 将 `ReadOnlySequence<T>` 复制到连续数组，并将其视为单个缓冲区：

- 如果 `ReadOnlySequence<T>` 的大小较小，则可以使用 stackalloc 运算符将数据复制到堆栈分配的缓冲区。
- 使用 ArrayPool<T>.Shared 将 `ReadOnlySequence<T>` 复制到共用的数组。
- 使用 `ReadOnlySequence<T>.ToArray()`。不建议在热路径中使用这种方法，因为它会在堆上分配新的 `T[]`。

以下示例演示了处理 `ReadOnlySequence<byte>` 的一些常见情况：

处理二进制数据

以下示例从 `ReadOnlySequence<byte>` 的开头分析 4 字节的大端整数长度。

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)
{
    // If there's not enough space, the length can't be obtained.
    if (buffer.Length < 4)
    {
        length = 0;
        return false;
    }

    // Grab the first 4 bytes of the buffer.
    var lengthSlice = buffer.Slice(buffer.Start, 4);
    if (lengthSlice.IsSingleSegment)
    {
        // Fast path since it's a single segment.
        length = BinaryPrimitives.ReadInt32BigEndian(lengthSlice.First.Span);
    }
    else
    {
        // There are 4 bytes split across multiple segments. Since it's so small, it
        // can be copied to a stack allocated buffer. This avoids a heap allocation.
        Span<byte> stackBuffer = stackalloc byte[4];
        lengthSlice.CopyTo(stackBuffer);
        length = BinaryPrimitives.ReadInt32BigEndian(stackBuffer);
    }

    // Move the buffer 4 bytes ahead.
    buffer = buffer.Slice(lengthSlice.End);

    return true;
}
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

处理文本数据

如下示例中：

- 查找 `ReadOnlySequence<byte>` 中的第一个换行符 (`\r\n`)，并通过 out 'line' 参数返回该符号。
- 剪裁该行，从输入缓冲区中排除 `\r\n`。

```
static bool TryParseLine(ref ReadOnlySequence<byte> buffer, out ReadOnlySequence<byte> line)
{
    SequencePosition position = buffer.Start;
    SequencePosition previous = position;
    var index = -1;
    line = default;

    while (buffer.TryGet(ref position, out ReadOnlyMemory<byte> segment))
    {
        ReadOnlySpan<byte> span = segment.Span;

        // Look for \r in the current segment.
        index = span.IndexOf((byte)'\r');

        if (index != -1)
        {
            // Check next segment for \n.
            if (index + 1 >= span.Length)
            {
                var next = position;
                if (!buffer.TryGet(ref next, out ReadOnlyMemory<byte> nextSegment))
                {
                    // You're at the end of the sequence.
                    return false;
                }
                else if (nextSegment.Span[0] == (byte)'\n')
                {
                    //  A match was found.
                    break;
                }
            }
            // Check the current segment of \n.
            else if (span[index + 1] == (byte)'\n')
            {
                // It was found.
                break;
            }
        }

        previous = position;
    }

    if (index != -1)
    {
        // Get the position just before the \r\n.
        var delimeter = buffer.GetPosition(index, previous);

        // Slice the line (excluding \r\n).
        line = buffer.Slice(buffer.Start, delimeter);

        // Slice the buffer to get the remaining data after the line.
        buffer = buffer.Slice(buffer.GetPosition(2, delimeter));
        return true;
    }

    return false;
}
```

空段

将空段存储在 `ReadOnlySequence<T>` 中是有效的。显式枚举段时，可能会出现空段：

```
static void EmptySegments()
{
    // This logic creates a ReadOnlySequence<byte> with 4 segments,
    // two of which are empty.
    var first = new BufferSegment(new byte[0]);
    var last = first.Append(new byte[] { 97 })
                    .Append(new byte[0]).Append(new byte[] { 98 });

    // Construct the ReadOnlySequence<byte> from the linked list segments.
    var data = new ReadOnlySequence<byte>(first, 0, last, 1);

    // Slice using numbers.
    var sequence1 = data.Slice(0, 2);

    // Slice using SequencePosition pointing at the empty segment.
    var sequence2 = data.Slice(data.Start, 2);

    Console.WriteLine($"sequence1.Length={sequence1.Length}"); // sequence1.Length=2
    Console.WriteLine($"sequence2.Length={sequence2.Length}"); // sequence2.Length=2

    // sequence1.FirstSpan.Length=1
    Console.WriteLine($"sequence1.FirstSpan.Length={sequence1.FirstSpan.Length}");

    // Slicing using SequencePosition will Slice the ReadOnlySequence<byte> directly
    // on the empty segment!
    // sequence2.FirstSpan.Length=0
    Console.WriteLine($"sequence2.FirstSpan.Length={sequence2.FirstSpan.Length}");

    // The following code prints 0, 1, 0, 1.
    SequencePosition position = data.Start;
    while (data.TryGet(ref position, out ReadOnlyMemory<byte> memory))
    {
        Console.WriteLine(memory.Length);
    }
}

class BufferSegment : ReadOnlySequenceSegment<byte>
{
    public BufferSegment(Memory<byte> memory)
    {
        Memory = memory;
    }

    public BufferSegment Append(Memory<byte> memory)
    {
        var segment = new BufferSegment(memory)
        {
            RunningIndex = RunningIndex + Memory.Length
        };
        Next = segment;
        return segment;
    }
}
```

前面的代码将创建一个包含空段的 `ReadOnlySequence<byte>`，并显示这些空段对各种 API 的影响：

- 包含指向空段的 `SequencePosition` 的 `ReadOnlySequence<T>.Slice` 会保留该段。
- 包含 int 的 `ReadOnlySequence<T>.Slice` 会跳过空段。
- 枚举 `ReadOnlySequence<T>` 会枚举空段。

ReadOnlySequence<T> 和 SequencePosition 的潜在问题

处理 `ReadOnlySequence<T>` / `SequencePosition` 与常规 `ReadOnlySpan<T>` / `ReadOnlyMemory<T>` / `T[]` / `int` 时，有几个异常的结果：

- `SequencePosition` 是特定 `ReadOnlySequence<T>` 的位置标记，而不是绝对位置。由于它是相对于特定 `ReadOnlySequence<T>` 的，因此如果在其起源的 `ReadOnlySequence<T>` 之外使用，则没有意义。
- 不能对没有 `ReadOnlySequence<T>` 的 `SequencePosition` 执行算术运算。这意味着，执行 `position++` 等基本操作将以 `ReadOnlySequence<T>.GetPosition(position, 1)` 的形式写入。
- `GetPosition(long)` 不支持负索引。这意味着，如果没有遍历所有段，就无法获取倒数第二个字符。
- 无法比较两个 `SequencePosition`，这使得难以：
  - 了解一个位置是否大于或小于另一个位置。
  - 编写一些分析算法。
- `ReadOnlySequence<T>` 大于对象引用，并且应尽可能通过 in 或 ref 进行传递。通过 `in` 或 `ref` 传递 `ReadOnlySequence<T>` 可减少结构的复制。
- 空段：
  - 在 `ReadOnlySequence<T>` 中有效。
  - 可能会在使用 `ReadOnlySequence<T>.TryGet` 方法进行循环访问时出现。
  - 可能会在结合使用 `ReadOnlySequence<T>.Slice()` 方法与 `SequencePosition` 对象来对序列进行切片时出现。

SequenceReader<T>

SequenceReader<T>：

- 是 .NET Core 3.0 中引入的一种新类型，用于简化 `ReadOnlySequence<T>` 的处理。
- 统一了单段 `ReadOnlySequence<T>` 和多段 `ReadOnlySequence<T>` 之间的差异。
- 提供用于读取二进制数据和文本数据（`byte` 和 `char`，不一定拆分到各个段）的帮助程序。

提供用于处理二进制数据和带分隔符的数据的内置方法。以下部分演示了与 `SequenceReader<T>` 相同的方法：

访问数据

`SequenceReader<T>` 具有用于直接枚举 `ReadOnlySequence<T>` 内的数据的方法。以下代码是一次处理 `ReadOnlySequence<byte>` 和 `byte` 的示例：

```
while (reader.TryRead(out byte b))
{
    Process(b);
}
```

`CurrentSpan` 公开了当前段的 `Span`，这类似于在方法中手动完成的操作。

使用位置

以下代码是使用 `SequenceReader<T>` 实现 `FindIndexOf` 的示例：

```
SequencePosition? FindIndexOf(in ReadOnlySequence<byte> buffer, byte data)
{
    var reader = new SequenceReader<byte>(buffer);

    while (!reader.End)
    {
        // Search for the byte in the current span.
        var index = reader.CurrentSpan.IndexOf(data);
        if (index != -1)
        {
            // It was found, so advance to the position.
            reader.Advance(index);

            return reader.Position;
        }
        // Skip the current segment since there's nothing in it.
        reader.Advance(reader.CurrentSpan.Length);
    }

    return null;
}
```

处理二进制数据

以下示例从 `ReadOnlySequence<byte>` 的开头分析 4 字节的大端整数长度。

```
bool TryParseHeaderLength(ref ReadOnlySequence<byte> buffer, out int length)
{
    var reader = new SequenceReader<byte>(buffer);
    return reader.TryReadBigEndian(out length);
}
```

处理文本数据

```
static ReadOnlySpan<byte> NewLine => new byte[] { (byte)'\r', (byte)'\n' };

static bool TryParseLine(ref ReadOnlySequence<byte> buffer,
                         out ReadOnlySequence<byte> line)
{
    var reader = new SequenceReader<byte>(buffer);

    if (reader.TryReadTo(out line, NewLine))
    {
        buffer = buffer.Slice(reader.Position);

        return true;
    }

    line = default;
    return false;
}
```

SequenceReader<T> 常见问题

- 由于 `SequenceReader<T>` 是可变结构，因此应始终通过引用进行传递。
- `SequenceReader<T>` 是引用结构，因此只能在同步方法中使用，不能存储在字段中。有关详细信息，请参阅编写安全有效的 C# 代码。
- `SequenceReader<T>` 已进行了优化，可用作只进读取器。`Rewind` 适用于无法利用其他 `Read`、`Peek` 和 `IsNext` API 来解决的小型备份。

内存映射文件

2021/11/16 •

内存映射文件包含虚拟内存中文件的内容。借助文件和内存空间之间的这种映射，应用（包括多个进程）可以直接对内存执行读取和写入操作，从而修改文件。可以使用托管代码访问内存映射文件，就像本机 Windows 函数访问内存映射文件（如管理内存映射文件中所述）一样。

内存映射文件分为两种类型：

- 持久化内存映射文件

  持久化文件是与磁盘上的源文件相关联的内存映射文件。当最后一个进程处理完文件时，数据保存到磁盘上的源文件中。此类内存映射文件适用于处理非常大的源文件。

- 非持久化内存映射文件

  非持久化文件是不与磁盘上的文件相关联的内存映射文件。当最后一个进程处理完文件时，数据会丢失，且文件被垃圾回收器回收。此类文件适合创建共享内存，以进行进程内通信 (IPC)。

进程、视图和管理内存

可以跨多个进程共享内存映射文件。进程可以映射到相同的内存映射文件，只需使用文件创建进程分配的通用名称即可。

必须创建整个或部分内存映射文件的视图，才能使用内存映射文件。还可以为内存映射文件的同一部分创建多个视图，从而创建并发内存。若要让两个视图一直处于并发状态，必须通过同一个内存映射文件创建它们。

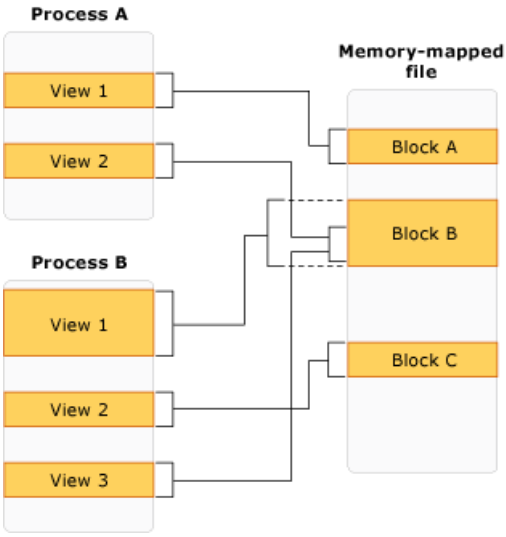
如果文件大于可用于内存映射的应用程序逻辑内存空间（在 32 位计算机中为 2 GB），可能也有必要使用多个视图。

视图分为以下两种类型：流访问视图和随机访问视图。使用流访问视图，可以顺序访问文件；建议对非持久化文件和 IPC 使用这种类型。随机访问视图是处理持久化文件的首选类型。

由于内存映射文件是通过操作系统的内存管理程序进行访问，因此文件会被自动分区到很多页面，并根据需要进行访问。无需自行处理内存管理。

下图展示了多个进程如何同时对同一个内存映射文件有多个重叠视图。

下图显示了内存映射文件的多个重叠视图：



使用内存映射文件编程

下表列出了与使用内存映射文件对象及其成员相关的指南。

㎝	㎜㎜㎜㎜㎜
从磁盘上的文件获取表示持久化内存映射文件的 MemoryMappedFile 对象。	MemoryMappedFile.CreateFromFile 方法。
获取表示非持久化内存映射文件的 MemoryMappedFile 对象（未与磁盘上的文件关联）。	MemoryMappedFile.CreateNew 方法。 - 或 - MemoryMappedFile.CreateOrOpen 方法。
获取现有内存映射文件（持久化或非持久化）的 MemoryMappedFile 对象。	MemoryMappedFile.OpenExisting 方法。
获取内存映射文件的顺序访问视图的 UnmanagedMemoryStream 对象。	MemoryMappedFile.CreateViewStream 方法。
获取内存映射文件的随机访问视图的 UnmanagedMemoryAccessor 对象。	MemoryMappedFile.CreateViewAccessor 方法。
获取要与非托管代码结合使用的 SafeMemoryMappedViewHandle 对象。	MemoryMappedFile.SafeMemoryMappedFileHandle 属性。 - 或 - MemoryMappedViewAccessor.SafeMemoryMappedViewHandle 属性。 - 或 - MemoryMappedViewStream.SafeMemoryMappedViewHandle 属性。
将内存分配一直延迟到视图创建完成（仅限非持久化文件）。 （若要确定当前系统页面大小，请使用 Environment.SystemPageSize 属性。）	值为 MemoryMappedFileOptions.DelayAllocatePages 的 CreateNew 方法。 - 或 - 将 MemoryMappedFileOptions 枚举用作参数的 CreateOrOpen 方法。

安全性

可以在创建内存映射文件时应用访问权限，具体操作是运行以下需要将 MemoryMappedFileAccess 枚举用作参数的方法：

- MemoryMappedFile.CreateFromFile
- MemoryMappedFile.CreateNew
- MemoryMappedFile.CreateOrOpen

若要指定打开现有内存映射文件所需的访问权限，可以运行需要将 MemoryMappedFileRights 用作参数的 OpenExisting 方法。

另外，还可以添加包含预定义访问规则的 MemoryMappedFileSecurity 对象。

若要将新的或更改后的访问规则应用于内存映射文件，请使用 SetAccessControl 方法。若要从现有文件检索访

问或审核规则，请使用 GetAccessControl 方法。

示例

持久化内存映射文件

CreateFromFile 方法通过磁盘上的现有文件创建内存映射文件。

下面的示例为极大文件的一部分创建内存映射视图，并控制其中一部分。

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        long offset = 0x10000000; // 256 megabytes
        long length = 0x20000000; // 512 megabytes

        // Create the memory-mapped file.
        using (var mmf = MemoryMappedFile.CreateFromFile(@"c:\ExtremelyLargeImage.data",
FileMode.Open,"ImgA"))
        {
            // Create a random access view, from the 256th megabyte (the offset)
            // to the 768th megabyte (the offset plus length).
            using (var accessor = mmf.CreateViewAccessor(offset, length))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < length; i += colorSize)
                {
                    accessor.Read(i, out color);
                    color.Brighten(10);
                    accessor.Write(i, ref color);
                }
            }
        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brighter.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}
```

```
Imports System.IO
Imports System.IO.MemoryMappedFiles
Imports System.Runtime.InteropServices

Class Program

    Sub Main()
        Dim offset As Long = &H10000000 ' 256 megabytes
        Dim length As Long = &H20000000 ' 512 megabytes

        ' Create the memory-mapped file.
        Using mmf = MemoryMappedFile.CreateFromFile("c:\ExtremelyLargeImage.data", FileMode.Open, "ImgA")
            ' Create a random access view, from the 256th megabyte (the offset)
            ' to the 768th megabyte (the offset plus length).
            Using accessor = mmf.CreateViewAccessor(offset, length)
                Dim colorSize As Integer = Marshal.SizeOf(GetType(MyColor))
                Dim color As MyColor
                Dim i As Long = 0

                ' Make changes to the view.
                Do While (i < length)
                    accessor.Read(i, color)
                    color.Brighten(10)
                    accessor.Write(i, color)
                    i += colorSize
                Loop
            End Using
        End Using
    End Sub
End Class

Public Structure MyColor
    Public Red As Short
    Public Green As Short
    Public Blue As Short
    Public Alpha As Short

    ' Make the view brighter.
    Public Sub Brighten(ByVal value As Short)
        Red = CType(Math.Min(Short.MaxValue, (CType(Red, Integer) + value)), Short)
        Green = CType(Math.Min(Short.MaxValue, (CType(Green, Integer) + value)), Short)
        Blue = CType(Math.Min(Short.MaxValue, (CType(Blue, Integer) + value)), Short)
        Alpha = CType(Math.Min(Short.MaxValue, (CType(Alpha, Integer) + value)), Short)
    End Sub
End Structure
```

若要查看翻译为非英语语言的代码注释，请在 此 GitHub 讨论问题中告诉我们。

下面的示例为另一个进程打开相同的内存映射文件。

```
using System;
using System.IO.MemoryMappedFiles;
using System.Runtime.InteropServices;

class Program
{
    static void Main(string[] args)
    {
        // Assumes another process has created the memory-mapped file.
        using (var mmf = MemoryMappedFile.OpenExisting("ImgA"))
        {
            using (var accessor = mmf.CreateViewAccessor(4000000, 2000000))
            {
                int colorSize = Marshal.SizeOf(typeof(MyColor));
                MyColor color;

                // Make changes to the view.
                for (long i = 0; i < 1500000; i += colorSize)
                {
                    accessor.Read(i, out color);
                    color.Brighten(20);
                    accessor.Write(i, ref color);
                }
            }
        }
    }
}

public struct MyColor
{
    public short Red;
    public short Green;
    public short Blue;
    public short Alpha;

    // Make the view brigher.
    public void Brighten(short value)
    {
        Red = (short)Math.Min(short.MaxValue, (int)Red + value);
        Green = (short)Math.Min(short.MaxValue, (int)Green + value);
        Blue = (short)Math.Min(short.MaxValue, (int)Blue + value);
        Alpha = (short)Math.Min(short.MaxValue, (int)Alpha + value);
    }
}
```

```
Imports System.IO.MemoryMappedFiles
Imports System.Runtime.InteropServices

Class Program
    Public Shared Sub Main(ByVal args As String())
        ' Assumes another process has created the memory-mapped file.
        Using mmf = MemoryMappedFile.OpenExisting("ImgA")
            Using accessor = mmf.CreateViewAccessor(4000000, 2000000)
                Dim colorSize As Integer = Marshal.SizeOf(GetType(MyColor))
                Dim color As MyColor

                ' Make changes to the view.
                Dim i As Long = 0
                While i < 1500000
                    accessor.Read(i, color)
                    color.Brighten(30)
                    accessor.Write(i, color)
                    i += colorSize
                End While
            End Using
        End Using
    End Sub
End Class

Public Structure MyColor
    Public Red As Short
    Public Green As Short
    Public Blue As Short
    Public Alpha As Short

    ' Make the view brigher.
    Public Sub Brighten(ByVal value As Short)
        Red = CShort(Math.Min(Short.MaxValue, CInt(Red) + value))
        Green = CShort(Math.Min(Short.MaxValue, CInt(Green) + value))
        Blue = CShort(Math.Min(Short.MaxValue, CInt(Blue) + value))
        Alpha = CShort(Math.Min(Short.MaxValue, CInt(Alpha) + value))
    End Sub
End Structure
```

非持久化内存映射文件

CreateNew 和 CreateOrOpen 方法创建未映射到磁盘上现有文件的内存映射文件。

下面的示例包含三个独立进程(控制台应用)，以将布尔值写入内存映射文件。各操作按下面的顺序发生：

1. `Process A` 创建内存映射文件，并向其中写入值。
2. `Process B` 打开内存映射文件，并向其中写入值。
3. `Process C` 打开内存映射文件，并向其中写入值。
4. `Process A` 读取并显示内存映射文件中的值。
5. 在 `Process A` 处理完内存映射文件后，此文件立即被垃圾回收器回收。

若要运行此示例，请按照以下步骤操作：

1. 编译应用并打开三个命令提示符窗口。
2. 在第一个命令提示符窗口中，运行 `Process A`。
3. 在第二个命令提示符窗口中，运行 `Process B`。
4. 返回到 `Process A`，再按 Enter。

5. 在第三个命令提示符窗口中，运行 `Process C`。

6. 返回到 `Process A`，再按 Enter。

`Process A` 的输出如下所示：

```
Start Process B and press ENTER to continue.
Start Process C and press ENTER to continue.
Process A says: True
Process B says: False
Process C says: True
```

Process A

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process A:
    static void Main(string[] args)
    {
        using (MemoryMappedFile mmf = MemoryMappedFile.CreateNew("testmap", 10000))
        {
            bool mutexCreated;
            Mutex mutex = new Mutex(true, "testmapmutex", out mutexCreated);
            using (MemoryMappedViewStream stream = mmf.CreateViewStream())
            {
                BinaryWriter writer = new BinaryWriter(stream);
                writer.Write(1);
            }
            mutex.ReleaseMutex();

            Console.WriteLine("Start Process B and press ENTER to continue.");
            Console.ReadLine();

            Console.WriteLine("Start Process C and press ENTER to continue.");
            Console.ReadLine();

            mutex.WaitOne();
            using (MemoryMappedViewStream stream = mmf.CreateViewStream())
            {
                BinaryReader reader = new BinaryReader(stream);
                Console.WriteLine("Process A says: {0}", reader.ReadBoolean());
                Console.WriteLine("Process B says: {0}", reader.ReadBoolean());
                Console.WriteLine("Process C says: {0}", reader.ReadBoolean());
            }
            mutex.ReleaseMutex();
        }
    }
}
```

```
Imports System.IO
Imports System.IO.MemoryMappedFiles
Imports System.Threading

Module Module1

    ' Process A:
    Sub Main()
        Using mmf As MemoryMappedFile = MemoryMappedFile.CreateNew("testmap", 10000)
            Dim mutexCreated As Boolean
            Dim mTex As Mutex = New Mutex(True, "testmapmutex", mutexCreated)
            Using Stream As MemoryMappedViewStream = mmf.CreateViewStream()
                Dim writer As BinaryWriter = New BinaryWriter(Stream)
                writer.Write(1)
            End Using
            mTex.ReleaseMutex()
            Console.WriteLine("Start Process B and press ENTER to continue.")
            Console.ReadLine()

            Console.WriteLine("Start Process C and press ENTER to continue.")
            Console.ReadLine()

            mTex.WaitOne()
            Using Stream As MemoryMappedViewStream = mmf.CreateViewStream()
                Dim reader As BinaryReader = New BinaryReader(Stream)
                Console.WriteLine("Process A says: {0}", reader.ReadBoolean())
                Console.WriteLine("Process B says: {0}", reader.ReadBoolean())
                Console.WriteLine("Process C says: {0}", reader.ReadBoolean())
            End Using
            mTex.ReleaseMutex()

        End Using

    End Sub

End Module
```

Process B

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process B:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf = MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
                mutex.WaitOne();

                using (MemoryMappedViewStream stream = mmf.CreateViewStream(1, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(0);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run Process A first.");
        }
    }
}
```

```
Imports System.IO
Imports System.IO.MemoryMappedFiles
Imports System.Threading

Module Module1
    ' Process B:
    Sub Main()
        Try
            Using mmf As MemoryMappedFile = MemoryMappedFile.OpenExisting("testmap")
                Dim mTex As Mutex = Mutex.OpenExisting("testmapmutex")
                mTex.WaitOne()
                Using Stream As MemoryMappedViewStream = mmf.CreateViewStream(1, 0)
                    Dim writer As BinaryWriter = New BinaryWriter(Stream)
                    writer.Write(0)
                End Using
                mTex.ReleaseMutex()
            End Using
        Catch noFile As FileNotFoundException
            Console.WriteLine("Memory-mapped file does not exist. Run Process A first." & vbCrLf &
noFile.Message)
        End Try

    End Sub

End Module
```

Process C

```
using System;
using System.IO;
using System.IO.MemoryMappedFiles;
using System.Threading;

class Program
{
    // Process C:
    static void Main(string[] args)
    {
        try
        {
            using (MemoryMappedFile mmf = MemoryMappedFile.OpenExisting("testmap"))
            {

                Mutex mutex = Mutex.OpenExisting("testmapmutex");
                mutex.WaitOne();

                using (MemoryMappedViewStream stream = mmf.CreateViewStream(2, 0))
                {
                    BinaryWriter writer = new BinaryWriter(stream);
                    writer.Write(1);
                }
                mutex.ReleaseMutex();
            }
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("Memory-mapped file does not exist. Run Process A first, then B.");
        }
    }
}
```

```
Imports System.IO
Imports System.IO.MemoryMappedFiles
Imports System.Threading

Module Module1
    ' Process C:
    Sub Main()
        Try
            Using mmf As MemoryMappedFile = MemoryMappedFile.OpenExisting("testmap")
                Dim mTex As Mutex = Mutex.OpenExisting("testmapmutex")
                mTex.WaitOne()
                Using Stream As MemoryMappedViewStream = mmf.CreateViewStream(2, 0)
                    Dim writer As BinaryWriter = New BinaryWriter(Stream)
                    writer.Write(1)
                End Using
                mTex.ReleaseMutex()
            End Using
        Catch noFile As FileNotFoundException
            Console.WriteLine("Memory-mapped file does not exist. Run Process A first, then B." & vbCrLf &
noFile.Message)
        End Try

    End Sub

End Module
```

请参阅

- 文件和流 I/O

.NET 中的控制台应用

2021/11/16 •

.NET 应用程序可以使用 System.Console 类在控制台中读取和写入字符。读取自控制台的数据是从标准输入流读取的，而写入到控制台的数据将写入标准输出流，并且写入控制台的错误数据将写入标准错误输出流。应用程序启动时，这些数据流会自动与控制台关联，并分别表示为 In、Out 和 Error 属性。

Console.In 属性的值是 System.IO.TextReader 对象，而 Console.Out 和 Console.Error 属性的值都是 System.IO.TextWriter 对象。你可以将这些属性与不表示控制台的流关联，以便可以将该流指向不同的输入位置或输出位置。例如，你可以通过将 Console.Out 属性设置为 System.IO.StreamWriter 将输出重定向到一个文件，这将通过 Console.SetOut 方法封装 System.IO.FileStream。Console.In 和 Console.Out 属性不需要引用相同流。

> **NOTE**
> 有关生成控制台应用程序（包括 C#、Visual Basic 和 C++ 中的示例）的详细信息，请参阅 Console 类文档。

如果不存在控制台（比如在 Windows 窗体应用程序中），写入标准输出流的输出将不可见，因为没有可以将信息写入的控制台。将信息写入不可访问的控制台不会引发异常。（例如在 Visual Studio 的项目属性页中，你始终可以将应用程序类型更改为“控制台应用程序”）。

System.Console 类具有从控制台读取单独的字符或整行的方法。其他方法转换数据和格式字符串，然后将设置了格式的字符串写入控制台。有关设置字符串格式的详细信息，请参阅格式设置类型。

> **TIP**
> 控制台应用程序缺少在默认情况下启动的消息泵。因此，控制台调用 Microsoft Win32 计时器时可能会失败。

另请参阅

- System.Console
- 格式设置类型

.NET 中的依赖关系注入

2021/11/16 •

.NET 支持依赖关系注入 (DI) 软件设计模式，这是一种在类及其依赖项之间实现控制反转 (IoC) 的技术。.NET 中的依赖关系注入是"一等公民"，提供配置、日志记录和选项模式。

依赖项是指另一个对象所依赖的对象。使用其他类所依赖的 `Write` 方法检查以下 `MessageWriter` 类：

```
public class MessageWriter
{
    public void Write(string message)
    {
        Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
    }
}
```

类可以创建 `MessageWriter` 类的实例，以便利用其 `Write` 方法。在以下示例中，`MessageWriter` 类是 `Worker` 类的依赖项：

```
public class Worker : BackgroundService
{
    private readonly MessageWriter _messageWriter = new MessageWriter();

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

该类创建并直接依赖于 `MessageWriter` 类。硬编码的依赖项（如前面的示例）会产生问题，应避免使用，原因如下：

- 要用不同的实现替换 `MessageWriter`，必须修改 `Worker` 类。
- 如果 `MessageWriter` 具有依赖项，则必须由 `Worker` 类对其进行配置。在具有多个依赖于 `MessageWriter` 的类的大型项目中，配置代码将分散在整个应用中。
- 这种实现很难进行单元测试。应用需使用模拟或存根 `MessageWriter` 类，而该类不能使用此方法。

依赖关系注入通过以下方式解决了这些问题：

- 使用接口或基类将依赖关系实现抽象化。
- 在服务容器中注册依赖关系。.NET 提供了一个内置的服务容器 IServiceProvider。服务通常在应用启动时注册，并追加到 IServiceCollection。添加所有服务后，可以使用 BuildServiceProvider 创建服务容器。
- 将服务注入到使用它的类的构造函数中。框架负责创建依赖关系的实例，并在不再需要时将其释放。

例如，`IMessageWriter` 接口定义 `Write` 方法：

```
namespace DependencyInjection.Example
{
    public interface IMessageWriter
    {
        void Write(string message);
    }
}
```

此接口由具体类型 `MessageWriter` 实现：

```
using System;

namespace DependencyInjection.Example
{
    public class MessageWriter : IMessageWriter
    {
        public void Write(string message)
        {
            Console.WriteLine($"MessageWriter.Write(message: \"{message}\")");
        }
    }
}
```

示例代码使用具体类型 `MessageWriter` 注册 `IMessageWriter` 服务。AddScoped 方法使用范围内生存期（单个请求的生存期）注册服务。本文后面将介绍服务生存期。

```
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace DependencyInjection.Example
{
    class Program
    {
        static Task Main(string[] args) =>
            CreateHostBuilder(args).Build().RunAsync();

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((_, services) =>
                    services.AddHostedService<Worker>()
                            .AddScoped<IMessageWriter, MessageWriter>());
    }
}
```

在示例应用中，请求 `IMessageWriter` 服务并用于调用 `Write` 方法：

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;

namespace DependencyInjection.Example
{
    public class Worker : BackgroundService
    {
        private readonly IMessageWriter _messageWriter;

        public Worker(IMessageWriter messageWriter) =>
            _messageWriter = messageWriter;

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                _messageWriter.Write($"Worker running at: {DateTimeOffset.Now}");
                await Task.Delay(1000, stoppingToken);
            }
        }
    }
}
```

通过使用 DI 模式，辅助角色服务：

- 不使用具体类型 `MessageWriter`，只使用实现它的 `IMessageWriter` 接口。这样可以轻松地更改控制器使用的实现，而无需修改控制器。
- 不创建 `MessageWriter` 的实例，这由 DI 容器创建。

可以通过使用内置日志记录 API 来改善 `IMessageWriter` 接口的实现：

```
using Microsoft.Extensions.Logging;

namespace DependencyInjection.Example
{
    public class LoggingMessageWriter : IMessageWriter
    {
        private readonly ILogger<LoggingMessageWriter> _logger;

        public LoggingMessageWriter(ILogger<LoggingMessageWriter> logger) =>
            _logger = logger;

        public void Write(string message) =>
            _logger.LogInformation(message);
    }
}
```

更新的 `ConfigureServices` 方法注册新的 `IMessageWriter` 实现：

```
static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureServices((_, services) =>
            services.AddHostedService<Worker>()
                    .AddScoped<IMessageWriter, LoggingMessageWriter>());
```

`LoggingMessageWriter` 依赖于 ILogger<TCategoryName>，并在构造函数中对其进行请求。`ILogger<TCategoryName>` 是框架提供的服务。

以链式方式使用依赖关系注入并不罕见。每个请求的依赖关系相应地请求其自己的依赖关系。容器解析图中的

依赖关系并返回完全解析的服务。必须被解析的依赖关系的集合通常被称为"依赖关系树"、"依赖关系图"或"对象图"。

容器通过利用(泛型)开放类型解析 `ILogger<TCategoryName>`，而无需注册每个(泛型)构造类型。

在依赖项注入术语中，服务：

- 通常是向其他对象提供服务的对象，如 `IMessageWriter` 服务。
- 与 Web 服务无关，尽管服务可能使用 Web 服务。

框架提供可靠的日志记录系统。编写上述示例中的 `IMessageWriter` 实现来演示基本的 DI，而不是来实现日志记录。大多数应用都不需要编写记录器。下面的代码展示了如何使用默认日志记录，只需要将 `Worker` 在 `ConfigureServices` 中注册为托管服务 AddHostedService：

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger) =>
        _logger = logger;

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}", DateTimeOffset.Now);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

使用前面的代码时，无需更新 `ConfigureServices`，因为框架提供日志记录。

多个构造函数发现规则

当某个类型定义多个构造函数时，服务提供程序具有用于确定要使用哪个构造函数的逻辑。选择最多参数的构造函数，其中的类型是可 DI 解析的类型。请考虑以下 C# 示例服务：

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(FooService fooService, BarService barService)
    {
        // omitted for brevity
    }
}
```

在前面的代码中，假定已添加日志记录，并且可以从服务提供程序解析，但 `FooService` 和 `BarService` 类型不可解析。使用 `ILogger<ExampleService>` 参数的构造函数用于解析 `ExampleService` 实例。即使有定义多个参数的构造函数，`FooService` 和 `BarService` 类型也不能进行 DI 解析。

如果发现构造函数时存在歧义，将引发异常。请考虑以下 C# 示例服务：

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(ILogger<ExampleService> logger)
    {
        // omitted for brevity
    }

    public ExampleService(IOptions<ExampleService> options)
    {
        // omitted for brevity
    }
}
```

WARNING

具有不明确的可 DI 解析的类型参数的 `ExampleService` 代码将引发异常。不要执行此操作，它旨在显示“不明确的可 DI 解析类型”的含义。

在前面的示例中，有三个构造函数。第一个构造函数是无参数的，不需要服务提供商提供的服务。假设日志记录和选项都已添加到 DI 容器，并且是可 DI 解析的服务。当 DI 容器尝试解析 `ExampleService` 类型时，将引发异常，因为这两个构造函数不明确。在此示例中，可以通过定义一个接受可 DI 解析类型的构造函数来避免多义性：

```
public class ExampleService
{
    public ExampleService()
    {
    }

    public ExampleService(
        ILogger<ExampleService> logger,
        IOptions<ExampleService> options)
    {
        // omitted for brevity
    }
}
```

使用扩展方法注册服务组

Microsoft 扩展使用一种约定来注册一组相关服务。约定使用单个 `Add{GROUP_NAME}` 扩展方法来注册该框架功能所需的所有服务。例如，AddOptions 扩展方法会注册使用选项所需的所有服务。

框架提供的服务

`ConfigureServices` 方法注册应用使用的服务，包括平台功能。最初，提供给 `ConfigureServices` 的 `IServiceCollection` 具有框架定义的服务（具体取决于主机配置方式）。对于基于 .NET 模板的应用，该框架会注册数百个服务。

下表列出了框架注册的这些服务的一小部分：

tttt	ttt
Microsoft.Extensions.DependencyInjection.IServiceScopeFactory	单例

[illegible]	[illegible]
IHostApplicationLifetime	单例
Microsoft.Extensions.Logging.ILogger<TCategoryName>	单例
Microsoft.Extensions.Logging.ILoggerFactory	单例
Microsoft.Extensions.ObjectPool.ObjectPoolProvider	单例
Microsoft.Extensions.Options.IConfigureOptions<TOptions>	暂时
Microsoft.Extensions.Options.IOptions<TOptions>	单例
System.Diagnostics.DiagnosticListener	单例
System.Diagnostics.DiagnosticSource	单例

服务生存期

可以使用以下任一生存期注册服务：

- 暂时
- 作用域
- 单例

下列各部分描述了上述每个生存期。为每个注册的服务选择适当的生存期。

暂时

暂时生存期服务是每次从服务容器进行请求时创建的。这种生存期适合轻量级、无状态的服务。向 AddTransient 注册暂时性服务。

在处理请求的应用中，在请求结束时会释放暂时服务。

范围内

对于 Web 应用，指定了作用域的生存期指明了每个客户端请求（连接）创建一次服务。向 AddScoped 注册范围内服务。

在处理请求的应用中，在请求结束时会释放有作用域的服务。

使用 Entity Framework Core 时，默认情况下 AddDbContext 扩展方法使用范围内生存期来注册 `DbContext` 类型。

> **NOTE**
> 不要从单一实例解析限定范围的服务，并小心不要间接地这样做，例如通过暂时性服务。当处理后续请求时，它可能会导致服务处于不正确的状态。可以：
>
> - 从范围内或暂时性服务解析单一实例服务。
> - 从其他范围内或暂时性服务解析范围内服务。

默认情况下在开发环境中，从具有较长生存期的其他服务解析服务将引发异常。有关详细信息，请参阅作用域验证。

单例

创建单例生命周期服务的情况如下：

- 在首次请求它们时进行创建；或者
- 在向容器直接提供实现实例时由开发人员进行创建。很少用到此方法。

来自依赖关系注入容器的服务实现的每一个后续请求都使用同一个实例。如果应用需要单一实例行为，则允许服务容器管理服务的生存期。不要实现单一实例设计模式，或提供代码来释放单一实例。服务永远不应由解析容器服务的代码释放。如果类型或工厂注册为单一实例，则容器自动释放单一实例。

向 AddSingleton 注册单一实例服务。单一实例服务必须是线程安全的，并且通常在无状态服务中使用。

在处理请求的应用中，当应用关闭并释放 ServiceProvider 时，会释放单一实例服务。由于应用关闭之前不释放内存，因此请考虑单一实例服务的内存使用。

服务注册方法

框架提供了适用于特定场景的服务注册扩展方法：

ㄍㄧ	ㄍㄧ (OBJECT) ㄍㄧ	ㄍㄧ ㄍㄧ	ㄍㄍㄍㄍ

`Add{LIFETIME} <{SERVICE}, {IMPLEMENTATION}>()`
 	
示例：
 	
`services.AddSingleton<IMyDep, MyDep>();`
	是	是	否

`Add{LIFETIME} <{SERVICE}>(sp => new {IMPLEMENTATION})`
 	
示例：
 	
`services.AddSingleton<IMyDep> (sp => new MyDep());`
`services.AddSingleton<IMyDep> (sp => new MyDep(99));`
	是	是	是

`Add{LIFETIME} <{IMPLEMENTATION}>()`
 	
示例：
 	
`services.AddSingleton<MyDep> ();`
	是	否	否

`AddSingleton<{SERVICE}> (new {IMPLEMENTATION})`
 	
示例：
 	
`services.AddSingleton<IMyDep> (new MyDep());`
`services.AddSingleton<IMyDep> (new MyDep(99));`
	否	是	是

☐☐	☐☐ ☐☐ (OBJECT) ☐☐	☐☐ ☐☐	☐☐☐☐

`AddSingleton(new {IMPLEMENTATION})`
 	
示例：
 	
`services.AddSingleton(new MyDep());`
`services.AddSingleton(new MyDep(99));`
	否	否	是

要详细了解释放类型，请参阅服务释放部分。

仅使用实现类型注册服务等效于使用相同的实现和服务类型注册该服务。因此，我们不能使用捕获显式服务类型的方法来注册服务的多个实现。这些方法可以注册服务的多个实例，但它们都具有相同的实现类型 。

上述任何服务注册方法都可用于注册同一服务类型的多个服务实例。下面的示例以 `IMessageWriter` 作为服务类型调用 `AddSingleton` 两次。第二次对 `AddSingleton` 的调用在解析为 `IMessageWriter` 时替代上一次调用，在通过 `IEnumerable<IMessageWriter>` 解析多个服务时添加到上一次调用。通过 `IEnumerable<{SERVICE}>` 解析服务时，服务按其注册顺序显示。

```
using System.Threading.Tasks;
using ConsoleDI.IEnumerableExample;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace ConsoleDI.Example
{
    class Program
    {
        static Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            _ = host.Services.GetService<ExampleService>();

            return host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((_, services) =>
                    services.AddSingleton<IMessageWriter, ConsoleMessageWriter>()
                            .AddSingleton<IMessageWriter, LoggingMessageWriter>()
                            .AddSingleton<ExampleService>());
    }
}
```

前面的示例源代码注册了 `IMessageWriter` 的两个实现。

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;

namespace ConsoleDI.IEnumerableExample
{
    public class ExampleService
    {
        public ExampleService(
            IMessageWriter messageWriter,
            IEnumerable<IMessageWriter> messageWriters)
        {
            Trace.Assert(messageWriter is LoggingMessageWriter);

            var dependencyArray = messageWriters.ToArray();
            Trace.Assert(dependencyArray[0] is ConsoleMessageWriter);
            Trace.Assert(dependencyArray[1] is LoggingMessageWriter);
        }
    }
}
```

`ExampleService` 定义两个构造函数参数：一个是 `IMessageWriter`，另一个是 `IEnumerable<IMessageWriter>`。第一个 `IMessageWriter` 是已注册的最后一个实现，而 `IEnumerable<IMessageWriter>` 表示所有已注册的实现。

框架还提供 `TryAdd{LIFETIME}` 扩展方法，只有当尚未注册某个实现时，才注册该服务。

在下面的示例中，对 `AddSingleton` 的调用会将 `ConsoleMessageWriter` 注册为 `IMessageWriter` 的实现。对 `TryAddSingleton` 的调用没有任何作用，因为 `IMessageWriter` 已有一个已注册的实现：

```
services.AddSingleton<IMessageWriter, ConsoleMessageWriter>();
services.TryAddSingleton<IMessageWriter, LoggingMessageWriter>();
```

`TryAddSingleton` 不起作用，因为已添加它并且“try”将失败。`ExampleService` 将断言以下内容：

```
public class ExampleService
{
    public ExampleService(
        IMessageWriter messageWriter,
        IEnumerable<IMessageWriter> messageWriters)
    {
        Trace.Assert(messageWriter is ConsoleMessageWriter);
        Trace.Assert(messageWriters.Single() is ConsoleMessageWriter);
    }
}
```

有关详细信息，请参阅：

- TryAdd
- TryAddTransient
- TryAddScoped
- TryAddSingleton

TryAddEnumerable(ServiceDescriptor) 方法仅会在没有同一类型实现的情况下才注册该服务。多个服务通过 `IEnumerable<{SERVICE}>` 解析。注册服务时，如果还没有添加相同类型的实例，就添加一个实例。库作者使用 `TryAddEnumerable` 来避免在容器中注册一个实现的多个副本。

在下面的示例中，对 `TryAddEnumerable` 的第一次调用会将 `MessageWriter` 注册为 `IMessageWriter1` 的实现。第二次调用向 `IMessageWriter2` 注册 `MessageWriter`。第三次调用没有任何作用，因为 `IMessageWriter1` 已有一个

`MessageWriter` 的已注册的实现：

```
public interface IMessageWriter1 { }
public interface IMessageWriter2 { }

public class MessageWriter : IMessageWriter1, IMessageWriter2
{
}

services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1, MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter2, MessageWriter>());
services.TryAddEnumerable(ServiceDescriptor.Singleton<IMessageWriter1, MessageWriter>());
```

服务注册通常与顺序无关，除了注册同一类型的多个实现时。

`IServiceCollection` 是 ServiceDescriptor 对象的集合。以下示例演示如何通过创建和添加 `ServiceDescriptor` 来注册服务：

```
string secretKey = Configuration["SecretKey"];
var descriptor = new ServiceDescriptor(
    typeof(IMessageWriter),
    _ => new DefaultMessageWriter(secretKey),
    ServiceLifetime.Transient);

services.Add(descriptor);
```

内置 `Add{LIFETIME}` 方法使用同一种方式。相关示例请参阅 AddScoped 源代码。

构造函数注入行为

服务可使用以下方式来解析：

- IServiceProvider
- ActivatorUtilities:
  - 创建未在容器中注册的对象。
  - 用于某些框架功能。

构造函数可以接受非依赖关系注入提供的参数，但参数必须分配默认值。

当服务由 `IServiceProvider` 或 `ActivatorUtilities` 解析时，*构造函数注入* 需要 public 构造函数。

当服务由 `ActivatorUtilities` 解析时，构造函数注入要求只存在一个适用的构造函数。支持构造函数重载，但其参数可以全部通过依赖注入来实现的重载只能存在一个。

作用域验证

如果应用在 `Development` 环境中运行，并调用 CreateDefaultBuilder 以生成主机，默认服务提供程序会执行检查，以确认以下内容：

- 没有从根服务提供程序解析到范围内服务。
- 未将范围内服务注入单一实例。

调用 BuildServiceProvider 时创建根服务提供程序。在启动提供程序和应用时，根服务提供程序的生存期对应于应用的生存期，并在关闭应用时释放。

有作用域的服务由创建它们的容器释放。如果范围内服务创建于根容器，则该服务的生存期实际上提升至单一实例，因为根容器只会在应用关闭时将其释放。验证服务作用域，将在调用 `BuildServiceProvider` 时收集这类情况。

范围场景

IServiceScopeFactory 始终注册为单一实例，但 IServiceProvider 可能因包含类的生存期而异。例如，如果从某个范围解析服务，而这些服务中的任意一种采用 IServiceProvider，该服务将是区分范围的实例。

若要在 IHostedService 的实现（例如 BackgroundService）中实现范围服务，请不要通过构造函数注入来注入服务依赖项。请改为注入 IServiceScopeFactory，创建范围，然后从该范围解析依赖项以使用适当的服务生存期。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace WorkerScope.Example
{
    public class Worker : BackgroundService
    {
        private readonly ILogger<Worker> _logger;
        private readonly IServiceScopeFactory _serviceScopeFactory;

        public Worker(ILogger<Worker> logger, IServiceScopeFactory serviceScopeFactory) =>
            (_logger, _serviceScopeFactory) = (logger, serviceScopeFactory);

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                using (IServiceScope scope = _serviceScopeFactory.CreateScope())
                {
                    try
                    {
                        _logger.LogInformation(
                            "Starting scoped work, provider hash: {hash}.",
                            scope.ServiceProvider.GetHashCode());

                        var store = scope.ServiceProvider.GetRequiredService<IObjectStore>();
                        var next = await store.GetNextAsync();
                        _logger.LogInformation("{next}", next);

                        var processor = scope.ServiceProvider.GetRequiredService<IObjectProcessor>();
                        await processor.ProcessAsync(next);
                        _logger.LogInformation("Processing {name}.", next.Name);

                        var relay = scope.ServiceProvider.GetRequiredService<IObjectRelay>();
                        await relay.RelayAsync(next);
                        _logger.LogInformation("Processed results have been relayed.");

                        var marked = await store.MarkAsync(next);
                        _logger.LogInformation("Marked as processed: {next}", marked);
                    }
                    finally
                    {
                        _logger.LogInformation(
                            "Finished scoped work, provider hash: {hash}.{nl}",
                            scope.ServiceProvider.GetHashCode(), Environment.NewLine);
                    }
                }
            }
        }
    }
}
```

在上述代码中，当应用运行时，后台服务：

- 依赖于 IServiceScopeFactory。
- 创建 IServiceScope 用于解析其他服务。
- 解析区分范围内的服务以供使用。
- 处理要处理的对象，然后对其执行中继操作，最后将其标记为已处理。

在示例源代码中，可以看到 IHostedService 的实现如何从区分范围的服务生存期中获益。

另请参阅

- 在 .NET 中使用依赖关系注入
- 依赖关系注入指南
- ASP.NET Core 中的依赖关系注入
- 用于 DI 应用开发的 NDC 会议模式
- 显式依赖关系原则
- 控制反转容器和依赖关系注入模式 (Martin Fowler)
- 应在 github.com/dotnet/extensions 存储库中创建 DI bug

教程：在 .NET 中使用依赖注入

2021/11/16 •

本教程介绍如何在 .NET 中使用依赖注入 (DI)。使用 Microsoft 扩展时，DI 是“一等公民”，其中服务是在 IServiceCollection 中添加和配置的。IHost 接口会公开 IServiceProvider 实例，它充当所有已注册的服务的容器。

本教程介绍如何执行下列操作：

- 创建一个使用依赖注入的 .NET 控制台应用
- 生成和配置通用主机
- 编写多个接口及相应的实现
- 为 DI 使用服务生存期和范围设定

先决条件

- .NET Core 3.1 SDK 或更高版本。
- 熟悉如何创建新的 .NET 应用程序以及如何安装 NuGet 包。

创建新的控制台应用程序

通过 dotnet new 命令或 IDE 的“新建项目”向导，新建一个名为 ConsoleDI 的 .NET 控制台应用程序 **Example** 。 将 NuGet 包 Microsoft.Extensions.Hosting 添加到项目。

添加接口

将以下接口添加到项目根目录：

IOperation.cs

```
namespace ConsoleDI.Example
{
    public interface IOperation
    {
        string OperationId { get; }
    }
}
```

`IOperation` 接口会定义一个 `OperationId` 属性。

IOperation.cs *Transient*

```
namespace ConsoleDI.Example
{
    public interface ITransientOperation : IOperation
    {
    }
}
```

IOperation.cs *Scoped*

```
namespace ConsoleDI.Example
{
    public interface IScopedOperation : IOperation
    {
    }
}
```

IOperation.cs *Singleton*

```
namespace ConsoleDI.Example
{
    public interface ISingletonOperation : IOperation
    {
    }
}
```

`IOperation` 的所有子接口会会命名其预期服务生存期。例如，“Transient”或“Singleton”。

添加默认实现

添加以下默认实现来进行各种操作：

DefaultOperation.cs

```
using static System.Guid;

namespace ConsoleDI.Example
{
    public class DefaultOperation :
        ITransientOperation,
        IScopedOperation,
        ISingletonOperation
    {
        public string OperationId { get; } = NewGuid().ToString()[^4..];
    }
}
```

`DefaultOperation` 会实现所有已命名的标记接口，并将 `OperationId` 属性初始化为新的全局唯一标识符 (GUID) 的最后 4 个字符。

添加需要 DI 的服务

添加以下操作记录器对象，它作为服务添加到控制台应用：

OperationLogger.cs

```
using System;

namespace ConsoleDI.Example
{
    public class OperationLogger
    {
        private readonly ITransientOperation _transientOperation;
        private readonly IScopedOperation _scopedOperation;
        private readonly ISingletonOperation _singletonOperation;

        public OperationLogger(
            ITransientOperation transientOperation,
            IScopedOperation scopedOperation,
            ISingletonOperation singletonOperation) =>
            (_transientOperation, _scopedOperation, _singletonOperation) =
                (transientOperation, scopedOperation, singletonOperation);

        public void LogOperations(string scope)
        {
            LogOperation(_transientOperation, scope, "Always different");
            LogOperation(_scopedOperation, scope, "Changes only with scope");
            LogOperation(_singletonOperation, scope, "Always the same");
        }


        private static void LogOperation<T>(T operation, string scope, string message)
            where T : IOperation =>
            Console.WriteLine(
                $"{scope}: {typeof(T).Name,-19} [ {operation.OperationId}...{message,-23} ]");
    }
}
```

`OperationLogger` 会定义一个构造函数，该函数需要上述每一个标记接口（即 `ITransientOperation`、`IScopedOperation` 和 `ISingletonOperation`）。对象会公开一个方法，使用者可通过该方法使用给定的 `scope` 参数记录操作。被调用时，`LogOperations` 方法会使用范围字符串和消息记录每个操作的唯一标识符。

为 DI 注册服务

使用以下代码更新 Program.cs：

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace ConsoleDI.Example
{
    class Program
    {
        static Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            ExemplifyScoping(host.Services, "Scope 1");
            ExemplifyScoping(host.Services, "Scope 2");

            return host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((_, services) =>
                    services.AddTransient<ITransientOperation, DefaultOperation>()
                            .AddScoped<IScopedOperation, DefaultOperation>()
                            .AddSingleton<ISingletonOperation, DefaultOperation>()
                            .AddTransient<OperationLogger>());

        static void ExemplifyScoping(IServiceProvider services, string scope)
        {
            using IServiceScope serviceScope = services.CreateScope();
            IServiceProvider provider = serviceScope.ServiceProvider;

            OperationLogger logger = provider.GetRequiredService<OperationLogger>();
            logger.LogOperations($"{scope}-Call 1 .GetRequiredService<OperationLogger>()");

            Console.WriteLine("...");

            logger = provider.GetRequiredService<OperationLogger>();
            logger.LogOperations($"{scope}-Call 2 .GetRequiredService<OperationLogger>()");

            Console.WriteLine();
        }
    }
}
```

每个 `services.Add{SERVICE_NAME}` 扩展方法添加(并可能配置)服务。我们建议应用遵循此约定。将扩展方法置于 Microsoft.Extensions.DependencyInjection 命名空间中以封装服务注册的组。还包括用于 DI 扩展方法的命名空间部分 `Microsoft.Extensions.DependencyInjection`:

- 允许在不添加其他 `using` 块的情况下在 IntelliSense 中显示它们。
- 在通常会调用这些扩展方法的 `Program` 或 `Startup` 类中，避免出现过多的 `using` 语句。

应用会执行以下操作:

- 使用默认活页夹设置创建一个 IHostBuilder 实例。
- 配置服务并对其添加相应的服务生存期。
- 调用 Build() 并分配 IHost 的实例。
- 调用 `ExemplifyScoping`，传入 IHost.Services。

结束语

应用会显示如下例所示的输出:

```
Scope 1-Call 1 .GetRequiredService<OperationLogger>(): ITransientOperation [ 80f4...Always different
]
Scope 1-Call 1 .GetRequiredService<OperationLogger>(): IScopedOperation    [ c878...Changes only with scope
]
Scope 1-Call 1 .GetRequiredService<OperationLogger>(): ISingletonOperation [ 1586...Always the same
]
...
Scope 1-Call 2 .GetRequiredService<OperationLogger>(): ITransientOperation [ f3c0...Always different
]
Scope 1-Call 2 .GetRequiredService<OperationLogger>(): IScopedOperation    [ c878...Changes only with scope
]
Scope 1-Call 2 .GetRequiredService<OperationLogger>(): ISingletonOperation [ 1586...Always the same
]

Scope 2-Call 1 .GetRequiredService<OperationLogger>(): ITransientOperation [ f9af...Always different
]
Scope 2-Call 1 .GetRequiredService<OperationLogger>(): IScopedOperation    [ 2bd0...Changes only with scope
]
Scope 2-Call 1 .GetRequiredService<OperationLogger>(): ISingletonOperation [ 1586...Always the same
]
...
Scope 2-Call 2 .GetRequiredService<OperationLogger>(): ITransientOperation [ fa65...Always different
]
Scope 2-Call 2 .GetRequiredService<OperationLogger>(): IScopedOperation    [ 2bd0...Changes only with scope
]
Scope 2-Call 2 .GetRequiredService<OperationLogger>(): ISingletonOperation [ 1586...Always the same
]
```

在应用输出中，可看到：

- Transient 操作总是不同，每次检索服务时，都会创建一个新实例。
- Scoped 仅随着新范围更改，但在一个范围中是相同的实例。
- Singleton 操作总是相同，新实例仅被创建一次。

另请参阅

- 依赖关系注入指南
- ASP.NET Core 中的依赖注入

依赖关系注入指南

2021/11/16 •

本文介绍在 .NET 应用程序中实现依赖关系注入的一般准则和最佳做法。

设计能够进行依赖关系注入的服务

在设计能够进行依赖注入的服务时：

- 避免有状态的、静态类和成员。通过将应用设计为使用单一实例服务，避免创建全局状态。
- 避免在服务中直接实例化依赖类。直接实例化会将代码耦合到特定实现。
- 不在服务中包含过多内容，确保设计规范，并易于测试。

如果一个类有很多已注入的依赖关系，这可能表明该类拥有过多的责任，并且违反了单一责任原则 (SRP)。尝试通过将某些职责移动到一个新类来重构类。

服务释放

容器负责清除它创建的类型，并在 IDisposable 实例上调用 Dispose。从容器中解析的服务绝对不应由开发人员释放。如果类型或工厂注册为单一实例，则容器自动释放单一实例。

在下面的示例中，服务由服务容器创建，并自动释放：

```
using System;

namespace ConsoleDisposable.Example
{
    public class TransientDisposable : IDisposable
    {
        public void Dispose() => Console.WriteLine($"{nameof(TransientDisposable)}.Dispose()");
    }
}
```

前面的可释放服务应具有暂时性生存期。

```
using System;

namespace ConsoleDisposable.Example
{
    public class ScopedDisposable : IDisposable
    {
        public void Dispose() => Console.WriteLine($"{nameof(ScopedDisposable)}.Dispose()");
    }
}
```

前面的可释放服务应具有作用域内生存期。

```
using System;

namespace ConsoleDisposable.Example
{
    public class SingletonDisposable : IDisposable
    {
        public void Dispose() => Console.WriteLine($"{nameof(SingletonDisposable)}.Dispose()");
    }
}
```

前面的可释放服务应具有单一实例生存期。

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace ConsoleDisposable.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            ExemplifyDisposableScoping(host.Services, "Scope 1");
            Console.WriteLine();

            ExemplifyDisposableScoping(host.Services, "Scope 2");
            Console.WriteLine();

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((_, services) =>
                    services.AddTransient<TransientDisposable>()
                            .AddScoped<ScopedDisposable>()
                            .AddSingleton<SingletonDisposable>());

        static void ExemplifyDisposableScoping(IServiceProvider services, string scope)
        {
            Console.WriteLine($"{scope}...");

            using IServiceScope serviceScope = services.CreateScope();
            IServiceProvider provider = serviceScope.ServiceProvider;

            _ = provider.GetRequiredService<TransientDisposable>();
            _ = provider.GetRequiredService<ScopedDisposable>();
            _ = provider.GetRequiredService<SingletonDisposable>();
        }
    }
```

调试控制台在运行后显示下面的示例输出：

```
Scope 1...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

Scope 2...
ScopedDisposable.Dispose()
TransientDisposable.Dispose()

info: Microsoft.Hosting.Lifetime[0]
      Application started.Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
     Hosting environment: Production
info: Microsoft.Hosting.Lifetime[0]
     Content root path: .\configuration\console-di-disposable\bin\Debug\net5.0
info: Microsoft.Hosting.Lifetime[0]
     Application is shutting down...
SingletonDisposable.Dispose()
```

不由服务容器创建的服务

考虑下列代码：

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton(new ExampleService());

    // ...
}
```

在上述代码中：

- `ExampleService` 实例不是由服务容器创建的。
- 框架不会自动释放服务。
- 开发人员负责释放服务。

暂时和共享实例的 IDisposable 指南

暂时、有限的生存期

方案

应用需要一个 IDisposable 实例，该实例在以下任一情况下具有暂时性生存期：

- 在根范围（根容器）内解析实例。
- 应在作用域结束之前释放实例。

解决方案

使用工厂模式在父作用域外创建实例。在这种情况下，应用通常会使用一个 `Create` 方法，该方法直接调用最终类型的构造函数。如果最终类型具有其他依赖项，则工厂可以：

- 在其构造函数中接收 IServiceProvider。
- 使用 ActivatorUtilities.CreateInstance 在容器外部实例化实例，同时将容器用于其依赖项。

共享实例，有限的生存期

方案

应用需要跨多个服务的共享 IDisposable 实例，但 IDisposable 实例应具有有限的生存期。

解决方案

为实例注册作用域生存期。使用 IServiceScopeFactory.CreateScope 创建新 IServiceScope。使用作用域的 IServiceProvider 获取所需的服务。如果不再需要范围，请将其释放。

通用 `IDisposable` 准则

- 不要为 IDisposable 实例注册暂时性生存期。请改用工厂模式。
- 不要在根范围内解析具有暂时性或范围内生存期的 IDisposable 实例。唯一的例外是应用创建/重新创建并释放 IServiceProvider 的情况，但这不是理想模式。
- 通过 DI 接收 IDisposable 依赖项不要求接收方自行实现 IDisposable。IDisposable 依赖项的接收方不能对该依赖项调用 Dispose。
- 使用范围控制服务的生存期。作用域不区分层次，并且在各作用域之间没有特定联系。

有关资源清理的详细信息，请参阅实现 `Dispose` 方法或实现 `DisposeAsync` 方法。另外，请考虑容器捕获的可释放的暂时性服务方案，因为它与资源清理相关。

默认服务容器替换

内置的服务容器旨在满足框架和大多数消费者应用的需求。我们建议使用内置容器，除非你需要的特定功能不受它支持，例如：

- 属性注入
- 基于名称的注入
- 子容器
- 自定义生存期管理
- 对迟缓初始化的 `Func<T>` 支持
- 基于约定的注册

以下第三方容器可用于 ASP.NET Core 应用：

- Autofac
- Dryloc
- Grace
- LightInject
- Lamar
- Stashbox
- Unity
- Simple Injector

线程安全

创建线程安全的单一实例服务。如果单一实例服务依赖于一个暂时服务，那么暂时服务可能也需要线程安全，具体取决于单一实例使用它的方式。

单一实例服务的工厂方法（例如 AddSingleton<TService>(IServiceCollection, Func<IServiceProvider,TService>) 的第二个参数）不必是线程安全的。像类型 (`static`) 构造函数一样，它保证仅由单个线程调用一次。

建议

- 不支持基于 `async/await` 和 `Task` 的服务解析。由于 C# 不支持异步构造函数，因此请在同步解析服务后使用异步方法。
- 避免在服务容器中直接存储数据和配置。例如，用户的购物车通常不应添加到服务容器中。配置应使用 选项模型。同样，避免“数据持有者”对象，也就是仅仅为实现对另一个对象的访问而存在的对象。最好通过 DI 请求实际项。
- 避免静态访问服务。例如，避免将 IApplicationBuilder.ApplicationServices 捕获为静态字段或属性以便在其他地方使用。
- 使 DI 工厂保持快速且同步。

- 避免使用服务定位器模式。例如，可以改为使用 DI 时，不要调用 GetService 来获取服务实例。
- 要避免的另一个服务定位器变体是注入需在运行时解析依赖项的工厂。这两种做法混合了控制反转策略。
- 避免在 `ConfigureServices` 中调用 BuildServiceProvider。当开发人员想要在 `ConfigureServices` 中解析服务时，通常会调用 `BuildServiceProvider`。
- 可释放的暂时性服务由容器捕获以进行释放。如果从顶级容器解析，这会变为内存泄漏。
- 启用范围验证，确保应用没有捕获范围内服务的单一实例。有关详细信息，请参阅作用域验证。

像任何一组建议一样，你可能会遇到需要忽略某建议的情况。例外情况很少见，主要是框架本身内部的特殊情况。

DI 是静态/全局对象访问模式的替代方法。如果将其与静态对象访问混合使用，则可能无法意识到 DI 的优点。

示例反模式

除了本文中介绍的指导原则之外，还应避免几种反模式。其中的某些反模式是开发运行时本身的知识。

> **WARNING**
> 这些是示例反模式，请不要复制代码，不要使用这些模式，并不惜一切代价避免使用这些模式。

由容器捕获的可释放的暂时性服务

当你注册可实现 IDisposable 的暂时性服务时，默认情况下，DI 容器将捕获这些引用，而不是这些引用的 Dispose()，如果它们是从容器中解析的，则直到应用程序停止且容器被释放，如果它们是从作用域中解析的，则直到作用域被释放。如果从容器级解析，这会变为内存泄漏。

```
static void TransientDisposablesWithoutDispose()
{
    var services = new ServiceCollection();
    services.AddTransient<ExampleDisposable>();
    ServiceProvider serviceProvider = services.BuildServiceProvider();

    for (int i = 0; i < 1000; ++ i)
    {
        _ = serviceProvider.GetRequiredService<ExampleDisposable>();
    }

    // serviceProvider.Dispose();
}
```

在上述反模式中，1,000 个 `ExampleDisposable` 对象将被实例化并为根对象。在释放 `serviceProvider` 实例之前，它们不会被释放。

有关调试内存泄漏的详细信息，请参阅调试 .NET 中的内存泄漏。

异步 DI 工厂可能导致死锁

术语"DI 工厂"指的是在调用 `Add{LIFETIME}` 时存在的重载方法。有一些重载接受 `Func<IServiceProvider, T>`，其中 `T` 是要注册的服务，参数命名为 `implementationFactory`。`implementationFactory` 可以作为 Lambda 表达式、局部函数或方法提供。如果工厂是异步的，并且你使用 Task<TResult>.Result，则会导致死锁。

```
static void DeadLockWithAsyncFactory()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>(implementationFactory: provider =>
    {
        Bar bar = GetBarAsync(provider).Result;
        return new Foo(bar);
    });

    services.AddSingleton<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    _ = serviceProvider.GetRequiredService<Foo>();
}
```

在前面的代码中，为 `implementationFactory` 指定了一个 Lambda 表达式，其中主体在 `Task<Bar>` 返回方法上调用 Task<TResult>.Result。这会导致死锁。`GetBarAsync` 方法只使用 Task.Delay 模拟异步工作操作，然后调用 GetRequiredService<T>(IServiceProvider)。

```
static async Task<Bar> GetBarAsync(IServiceProvider serviceProvider)
{
    // Emulate asynchronous work operation
    await Task.Delay(1000);

    return serviceProvider.GetRequiredService<Bar>();
}
```

有关异步指南的详细信息，请参阅异步编程：重要信息和建议。有关调试死锁的详细信息，请参阅调试 .NET 中的死锁。

在运行此反模式并发生死锁时，可以从 Visual Studio 的"并行堆栈"窗口查看等待的两个线程。有关详细信息，请参阅在"并行堆栈"窗口中查看线程和任务。

捕获依赖项

术语"捕获依赖项"由 Mark Seemann 提出，指的是服务生存期的配置不正确，其中具有较长生存期的服务捕获了具有较短生存期的服务。

```
static void CaptiveDependency()
{
    var services = new ServiceCollection();
    services.AddSingleton<Foo>();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider();
    // Enable scope validation
    // using ServiceProvider serviceProvider = services.BuildServiceProvider(validateScopes: true);

    _ = serviceProvider.GetRequiredService<Foo>();
```

在前面的代码中，`Foo` 注册为单一实例，并确定了 `Bar` 的作用域，这在表面上看似乎是有效的。不过，请考虑 `Foo` 的实现。

```
namespace DependencyInjection.AntiPatterns
{
    public class Foo
    {
        public Foo(Bar bar)
        {
        }
    }
}
```

`Foo` 对象需要一个 `Bar` 对象，因为 `Foo` 是单一实例，并且已确定 `Bar` 的作用域 - 这是错误的配置。因此，将只会实例化 `Foo` 一次，并且它会在自己的生存期内捕获 `Bar`，这比所需的作用域内的 `Bar` 生存期要长。应考虑通过将 `validateScopes: true` 传递到 BuildServiceProvider(IServiceCollection, Boolean) 来验证作用域。验证作用域时，你将收到 InvalidOperationException 和一条消息，类似于“无法使用来自单一实例‘Foo’的作用域内的服务‘Bar’”。

有关详细信息，请参阅作用域验证。

作为单一实例的作用域服务

使用作用域内服务时，如果你不是在现有作用域内创建作用域，则该服务将成为单一实例。

```
static void ScopedServiceBecomesSingleton()
{
    var services = new ServiceCollection();
    services.AddScoped<Bar>();

    using ServiceProvider serviceProvider = services.BuildServiceProvider(validateScopes: true);
    using (IServiceScope scope = serviceProvider.CreateScope())
    {
        // Correctly scoped resolution
        Bar correct = scope.ServiceProvider.GetRequiredService<Bar>();
    }

    // Not within a scope, becomes a singleton
    Bar avoid = serviceProvider.GetRequiredService<Bar>();
}
```

在前面的代码中，在 IServiceScope 中检索 `Bar`，这是正确的。反模式是作用域外的 `Bar` 检索，变量命名为 `avoid` 以显示不正确的示例检索。

另请参阅

- .NET 中的依赖关系注入
- 教程：在 .NET 中使用依赖关系注入

.NET 中的配置

2021/11/16 •

.NET 中的配置是使用一个或多个配置提供程序执行的。配置提供程序使用各种配置源从键值对读取配置数据：

- 设置文件，例如 appsettings.json
- 环境变量
- Azure Key Vault
- Azure 应用配置
- 命令行参数
- 已安装或已创建的自定义提供程序
- 目录文件
- 内存中的 .NET 对象

配置控制台应用

在默认情况下，使用 dotnet new 或 Visual Studio 新建的 .NET 控制台应用程序不会公开配置功能。若要在新的 .NET 控制台应用程序中添加配置，请添加对 `Microsoft.Extensions.Hosting` 的包引用。修改 Program.cs 文件，使其与以下代码相匹配：

```
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;

namespace Console.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args);
    }
}
```

Host.CreateDefaultBuilder(String[]) 方法按照以下顺序为应用提供默认配置：

1. ChainedConfigurationProvider：添加现有的 `IConfiguration` 作为源。
2. 使用 JSON 配置提供程序通过 *appsettings.json* 提供。
3. 使用 JSON 配置提供程序通过 appsettings.`Environment`.json 提供 。例如，appsettings.Production.json 和 appsettings.Development.json 。
4. 应用在 `Development` 环境中运行时的应用机密。
5. 使用环境变量配置提供程序通过环境变量提供。
6. 使用命令行配置提供程序通过命令行参数提供。

后来添加的配置提供程序会替代之前的密钥设置。例如，如果在 appsettings.json 和环境中设置了 `SomeKey`，则

会使用环境值。通过默认配置提供程序，命令行配置提供程序将替代其他所有提供程序。

绑定

.NET 中的配置的其中一个关键优点是，可将配置值绑定到 .NET 对象的实例。例如，JSON 配置提供程序可用于将 appsettings.json 文件映射到 .NET 对象中并与依赖注入一起使用。此可实现选项模式，后者使用类来提供对相关设置组的强类型访问。

配置提供程序

下表显示了 .NET Core 应用可用的配置提供程序。

[illegible]	[illegible]
Azure 应用配置提供程序	Azure 应用程序配置
Azure Key Vault 配置提供程序	Azure Key Vault
命令行配置提供程序	命令行参数
自定义配置提供程序	自定义源
环境变量配置提供程序	环境变量
文件配置提供程序	JSON、XML 和 INI 文件
Key-per-file 配置提供程序	目录文件
内存配置提供程序	内存中集合
应用机密(机密管理器)	用户配置文件目录中的文件

若要详细了解各种配置提供程序，请查看 .NET 中的配置提供程序。

另请参阅

- .NET 中的配置提供程序
- 实现自定义配置提供程序
- 配置 bug 应在 github.com/dotnet/extensions 存储库中创建

.NET 中的配置提供程序

2021/11/16 •

可通过配置提供程序进行 .NET 配置。存在几种类型的提供程序，它们依赖于不同的配置源。本文详细介绍了所有不同的配置提供程序及其相应的源。

- 文件配置提供程序
- 环境变量配置提供程序
- 命令行配置提供程序
- Key-per-file 配置提供程序
- 内存配置提供程序

文件配置提供程序

FileConfigurationProvider 是从文件系统加载配置的基类。以下配置提供程序派生自 `FileConfigurationProvider`：

- JSON 配置提供程序
- XML 配置提供程序
- INI 配置提供程序

JSON 配置提供程序

JsonConfigurationProvider 类从 JSON 文件加载配置。安装 NuGet 包 `Microsoft.Extensions.Configuration.Json`。

重载可以指定：

- 文件是否可选
- 如果文件更改，是否重载配置

考虑下列代码：

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

namespace ConsoleJson.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((hostingContext, configuration) =>
                {
                    configuration.Sources.Clear();

                    IHostEnvironment env = hostingContext.HostingEnvironment;

                    configuration
                        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
                        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", true, true);

                    IConfigurationRoot configurationRoot = configuration.Build();

                    TransientFaultHandlingOptions options = new();
                    configurationRoot.GetSection(nameof(TransientFaultHandlingOptions))
                                     .Bind(options);

                    Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
                    Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay=
{options.AutoRetryDelay}");
                });
    }
}
```

前面的代码：

- 清除 CreateDefaultBuilder(String[]) 方法中默认添加的所有现有的配置提供程序。
- 将 JSON 配置提供程序配置为使用以下选项加载 appsettings.json 和 appsettings.`Environment`.json 文件：
  - `optional: true`：文件是可选的。
  - `reloadOnChange: true`：保存更改后会重载文件。

JSON 设置会被环境变量配置提供程序和命令行配置提供程序中的设置替代。

下面是一个具有各种配置设置的示例 appsettings.json 文件：

```
{
    "SecretKey": "Secret key value",
    "TransientFaultHandlingOptions": {
        "Enabled": true,
        "AutoRetryDelay": "00:00:07"
    },
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

从 IConfigurationBuilder 实例中，在添加配置提供程序后，可调用 IConfigurationBuilder.Build() 来获取 IConfigurationRoot 对象。配置根表示配置层次结构的根。可将配置中的节绑定到 .NET 对象的实例，稍后再通过依赖注入将其作为 IOptions<TOptions> 提供。

请考虑如下定义的 `TransientFaultHandlingOptions` 类：

```
using System;

namespace ConsoleJson.Example
{
    public class TransientFaultHandlingOptions
    {
        public bool Enabled { get; set; }
        public TimeSpan AutoRetryDelay { get; set; }
    }
}
```

下面的代码会生成配置根，将一个节绑定到类 `TransientFaultHandlingOptions` 类型，并将绑定值输出到控制台窗口：

```
IConfigurationRoot configurationRoot = configuration.Build();

TransientFaultHandlingOptions options = new();
configurationRoot.GetSection(nameof(TransientFaultHandlingOptions))
                 .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={options.AutoRetryDelay}");
```

应用程序会编写以下示例输出：

```
// Sample output:
//    TransientFaultHandlingOptions.Enabled=True
//    TransientFaultHandlingOptions.AutoRetryDelay=00:00:07
```

XML 配置提供程序

XmlConfigurationProvider 类在运行时从 XML 文件加载配置。安装 NuGet 包 `Microsoft.Extensions.Configuration.Xml`。

下面的代码演示如何使用 XML 配置提供程序配置 XML 文件。

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

namespace ConsoleXml.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((hostingContext, configuration) =>
                {
                    configuration.Sources.Clear();

                    configuration
                        .AddXmlFile("appsettings.xml", optional: true, reloadOnChange: true)
                        .AddXmlFile("repeating-example.xml", optional: true, reloadOnChange: true);

                    configuration.AddEnvironmentVariables();

                    if (args is { Length: > 0 })
                    {
                        configuration.AddCommandLine(args);
                    }
                });
    }
}
```

上述代码：

- 清除 CreateDefaultBuilder(String[]) 方法中默认添加的所有现有的配置提供程序。
- 将 XML 配置提供程序配置为使用以下选项加载 appsettings.xml 和 repeating-example.xml 文件：
  - `optional: true`：文件是可选的。
  - `reloadOnChange: true`：保存更改后会重载文件。
- 配置环境变量配置提供程序。
- 如果给定的 `args` 包含自变量，则配置命令行配置提供程序。

XML 设置会被环境变量配置提供程序和命令行配置提供程序中的设置替代。

下面是一个具有各种配置设置的示例 appsettings.xml 文件：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <SecretKey>Secret key value</SecretKey>
  <TransientFaultHandlingOptions>
    <Enabled>true</Enabled>
    <AutoRetryDelay>00:00:07</AutoRetryDelay>
  </TransientFaultHandlingOptions>
  <Logging>
    <LogLevel>
      <Default>Information</Default>
      <Microsoft>Warning</Microsoft>
    </LogLevel>
  </Logging>
</configuration>
```

在 .NET 5 及更早版本中，添加 `name` 属性以区分使用同一元素名称的重复元素。在 .NET 6 及更高版本中，XML 配置提供程序会自动为重复元素编制索引。这意味着你不必指定 `name` 属性，除非你希望键中有“0”索引，并且只有一个元素。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <section name="section0">
    <key name="key0">value 00</key>
    <key name="key1">value 01</key>
  </section>
  <section name="section1">
    <key name="key0">value 10</key>
    <key name="key1">value 11</key>
  </section>
</configuration>
```

以下代码会读取前面的配置文件并显示键和值：

```
IConfigurationRoot configurationRoot = configuration.Build();

string key00 = "section:section0:key:key0";
string key01 = "section:section0:key:key1";
string key10 = "section:section1:key:key0";
string key11 = "section:section1:key:key1";

string val00 = configurationRoot[key00];
string val01 = configurationRoot[key01];
string val10 = configurationRoot[key10];
string val11 = configurationRoot[key11];

Console.WriteLine($"{key00} = {val00}");
Console.WriteLine($"{key01} = {val01}");
Console.WriteLine($"{key10} = {val10}");
Console.WriteLine($"{key10} = {val11}");
```

该应用程序会编写以下示例输出：

```
// Sample output:
//    section:section0:key:key0 = value 00
//    section:section0:key:key1 = value 01
//    section:section1:key:key0 = value 10
//    section:section1:key:key0 = value 11
```

属性可用于提供值：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <key attribute="value" />
  <section>
    <key attribute="value" />
  </section>
</configuration>
```

以前的配置文件使用 `value` 加载以下键：

- `key:attribute`
- `section:key:attribute`

INI 配置提供程序

IniConfigurationProvider 类在运行时从 INI 文件加载配置。安装 NuGet 包 `Microsoft.Extensions.Configuration.Ini`。

以下代码将清除所有配置提供程序，并添加具有两个 INI 文件的 `IniConfigurationProvider` 作为源：

```
using System;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

namespace ConsoleIni.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((hostingContext, configuration) =>
                {
                    configuration.Sources.Clear();

                    IHostEnvironment env = hostingContext.HostingEnvironment;

                    configuration
                        .AddIniFile("appsettings.ini", optional: true, reloadOnChange: true)
                        .AddIniFile($"appsettings.{env.EnvironmentName}.ini", true, true);

                    foreach ((string key, string value) in
                        configuration.Build().AsEnumerable().Where(t => t.Value is not null))
                    {
                        Console.WriteLine($"{key}={value}");
                    }
                });
    }
}
```

下面是一个具有各种配置设置的示例 appsettings.ini 文件：

```
SecretKey="Secret key value"

[TransientFaultHandlingOptions]
Enabled=True
AutoRetryDelay="00:00:07"

[Logging:LogLevel]
Default=Information
Microsoft=Warning
```

以下代码通过将上述配置设置写入控制台窗口来显示这些设置：

```
foreach ((string key, string value) in
    configuration.Build().AsEnumerable().Where(t => t.Value is not null))
{
    Console.WriteLine($"{key}={value}");
}
```

该应用程序会编写以下示例输出：

```
// Sample output:
//    TransientFaultHandlingOptions:Enabled=True
//    TransientFaultHandlingOptions:AutoRetryDelay=00:00:07
//    SecretKey=Secret key value
//    Logging:LogLevel:Microsoft=Warning
//    Logging:LogLevel:Default=Information
```

环境变量配置提供程序

通过默认配置，EnvironmentVariablesConfigurationProvider 会在读取 appsettings.json、appsettings.`Environment`.json 和机密管理器后，从环境变量键值对加载配置 。因此，从环境中读取的键值会替代从 appsettings.json、appsettings.`Environment`.json 和机密管理器中读取的值 。

所有平台上的环境变量分层键都不支持 `:` 分隔符。双下划线 (`__`) 会自动替换为 `:` 且受各大平台支持。例如，Bash 不支持 `:` 分隔符，但支持 `__`。

以下 `set` 命令：

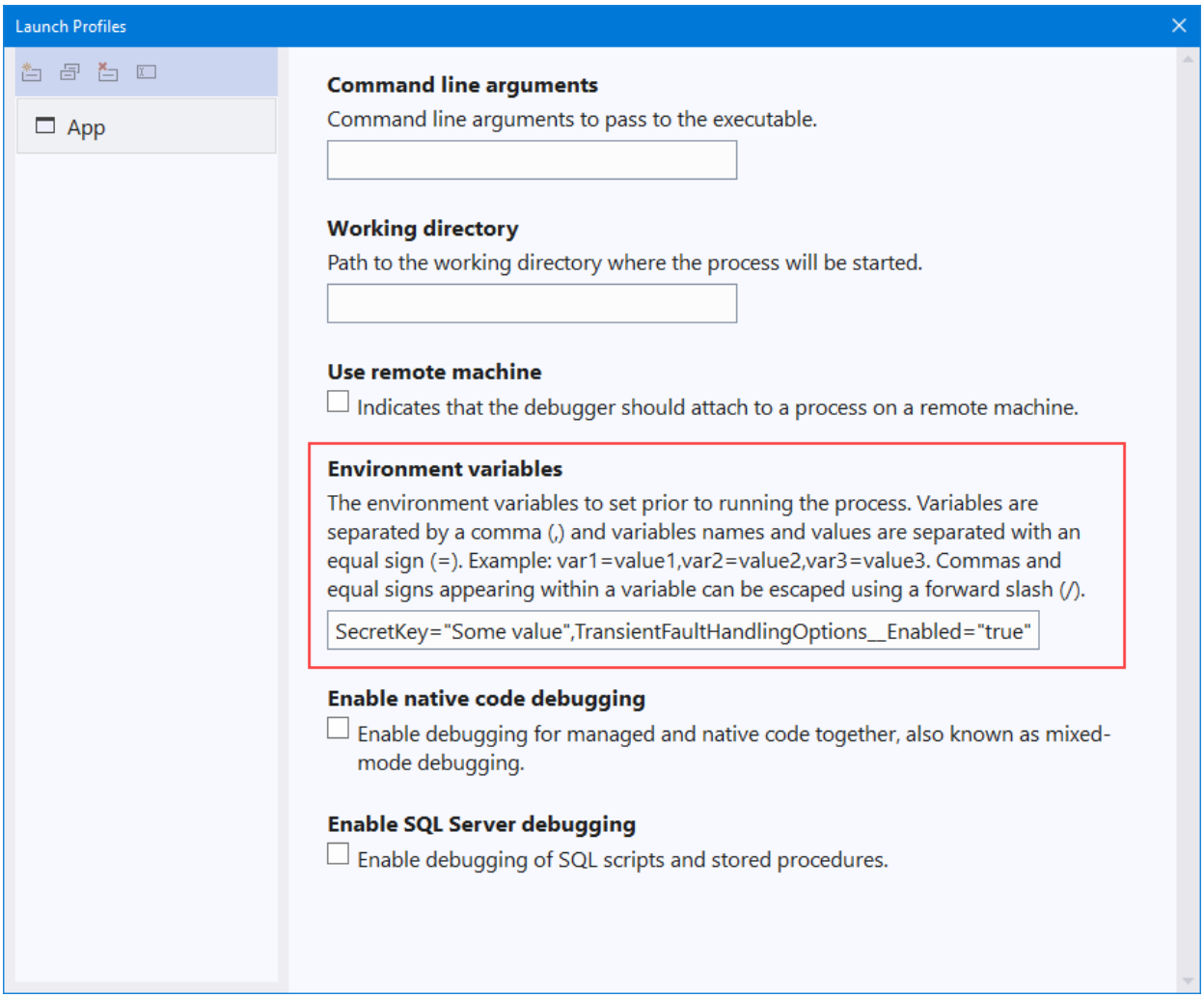
- 在 Windows 上设置上述示例的环境键和值。
- 通过将设置更改为非默认值来对其进行测试。`dotnet run` 命令必须在项目目录中运行。

```
set SecretKey="Secret key from environment"
set TransientFaultHandlingOptions__Enabled="true"
set TransientFaultHandlingOptions__AutoRetryDelay="00:00:13"

dotnet run
```

前面的环境设置：

- 仅在进程中设置，这些进程是从设置进程的命令窗口启动的。
- 不会由通过 Visual Studio 启动的 Web 应用读取。

在 Visual Studio 2019 版本 16.10 预览版 4 及更高版本中，可以使用“启动配置文件”对话框指定环境变量。



以下 setx 命令可用于在 Windows 上设置环境键和值。与 `set` 不同，`setx` 设置是持久的。`/M` 在系统环境中设置变量。如果未使用 `/M` 开关，则会设置用户环境变量。

```
setx SecretKey "Secret key from setx environment" /M
setx TransientFaultHandlingOptions__Enabled "true" /M
setx TransientFaultHandlingOptions__AutoRetryDelay "00:00:05" /M

dotnet run
```

测试前面的命令是否会替代 appsettings.json 和 appsettings.`Environment`.json：

- 使用 Visual Studio：退出并重启 Visual Studio。
- 使用 CLI：启动新的命令窗口并输入 `dotnet run`。

使用字符串调用 AddEnvironmentVariables 以指定环境变量的前缀：

```
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

namespace ConsoleEnvironment.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((hostingContext, configuration) =>
                    configuration.AddEnvironmentVariables(
                        prefix: "CustomPrefix_"));
    }
}
```

在上述代码中：

- `config.AddEnvironmentVariables(prefix: "CustomPrefix_")` 被添加到默认配置提供程序的后面。有关对配置提供程序进行排序的示例，请查看 XML 配置提供程序。
- 设有 `CustomPrefix_` 前缀的环境变量会替代默认配置提供程序。这包括没有前缀的环境变量。

前缀会在读取配置键值对时被去除。

以下命令对自定义前缀进行测试：

```
set CustomPrefix__SecretKey="Secret key with CustomPrefix_ environment"
set CustomPrefix__TransientFaultHandlingOptions__Enabled=true
set CustomPrefix__TransientFaultHandlingOptions__AutoRetryDelay=00:00:21

dotnet run
```

默认配置会加载前缀为 `DOTNET_` 的环境变量和命令行自变量。`DOTNET_` 前缀由 .NET 用于主机和应用配置，但不用于用户配置。

有关主机和应用配置的详细信息，请参阅 .NET 通用主机。

在 Azure 应用服务上，选择“设置”>“配置”页面上的“新应用程序设置”。Azure 应用服务应用程序设置：

- 已静态加密且通过加密的通道进行传输。
- 已作为环境变量公开。

连接字符串前缀

对于四个连接字符串环境变量，配置 API 具有特殊的处理规则。这些连接字符串涉及了为应用环境配置 Azure 连接字符串。使用默认配置或没有向 `AddEnvironmentVariables` 应用前缀时，具有表中所示前缀的环境变量将加载到应用中。

[illegible]	[illegible]
`CUSTOMCONNSTR_`	自定义提供程序

[illegible]	[illegible]
`MYSQLCONNSTR_`	MySQL
`SQLAZURECONNSTR_`	Azure SQL 数据库
`SQLCONNSTR_`	SQL Server

当发现环境变量并使用表中所示的四个前缀中的任何一个加载到配置中时：

- 通过删除环境变量前缀并添加配置键节（`ConnectionStrings`）来创建配置键。
- 创建一个新的配置键值对，表示数据库连接提供程序（`CUSTOMCONNSTR_` 除外，它没有声明的提供程序）。

[illegible]	[illegible]	[illegible]
`CUSTOMCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	配置条目未创建。
`MYSQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	键：`ConnectionStrings:{KEY}_ProviderName` ： 值：`MySql.Data.MySqlClient`
`SQLAZURECONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	键：`ConnectionStrings:{KEY}_ProviderName` ： 值：`System.Data.SqlClient`
`SQLCONNSTR_{KEY}`	`ConnectionStrings:{KEY}`	键：`ConnectionStrings:{KEY}_ProviderName` ： 值：`System.Data.SqlClient`

在 launchSettings.json 中设置的环境变量

在 launchSettings.json 中设置的环境变量将替代在系统环境中设置的变量。

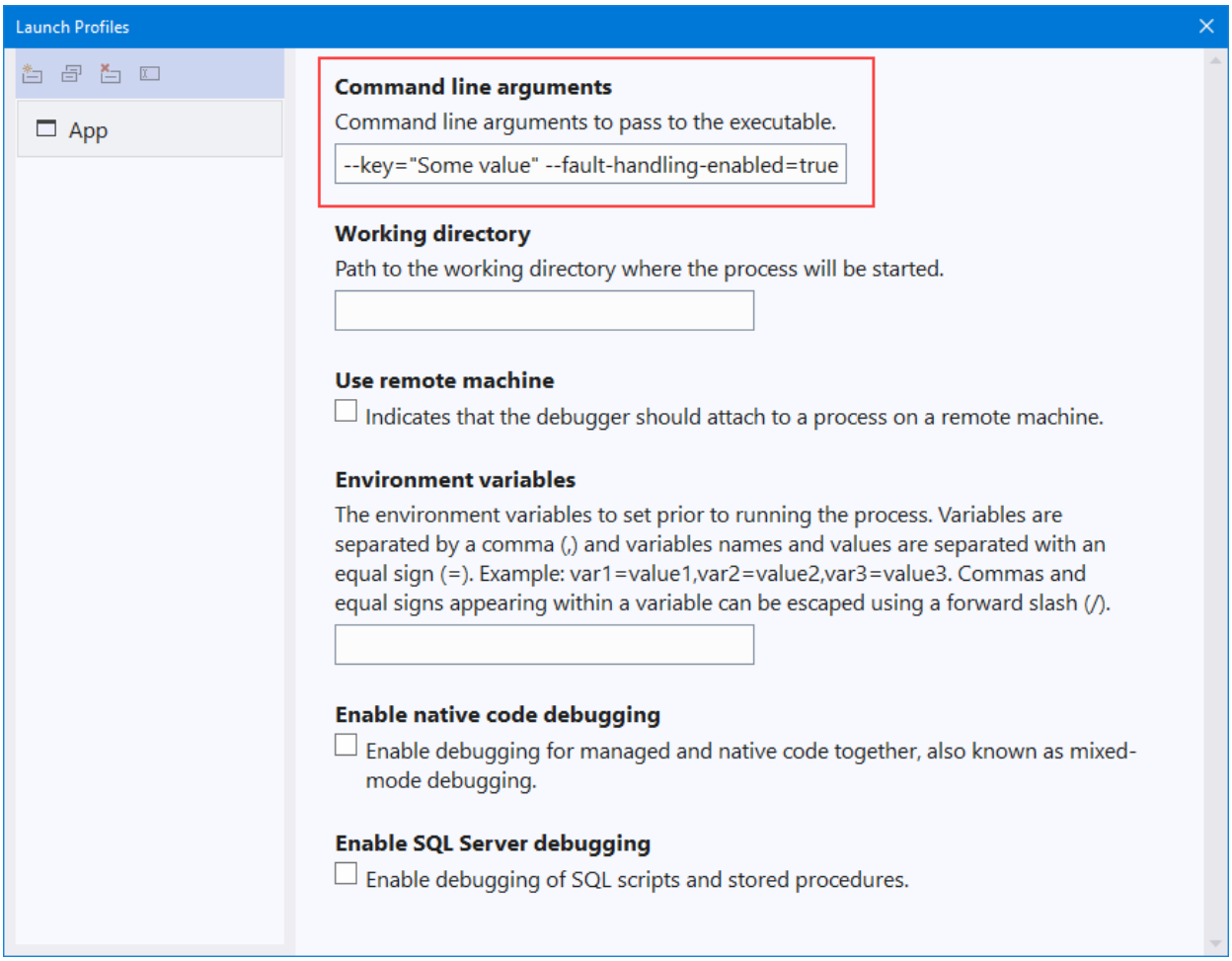
命令行配置提供程序

使用默认配置，CommandLineConfigurationProvider 会从以下配置源后的命令行参数键值对中加载配置：

- appsettings.json 和 appsettings.`Environment`.json 文件 。
- `Development` 环境中的应用机密（机密管理器）。
- 环境变量。

默认情况下，在命令行上设置的配置值会替代通过所有其他配置提供程序设置的配置值。

在 Visual Studio 2019 版本 16.10 预览版 4 及更高版本中，可以使用“启动配置文件”对话框指定命令行参数。



命令行参数

以下命令使用 `=` 设置键和值：

```
dotnet run SecretKey="Secret key from command line"
```

以下命令使用 `/` 设置键和值：

```
dotnet run /SecretKey "Secret key set from forward slash"
```

以下命令使用 `--` 设置键和值：

```
dotnet run --SecretKey "Secret key set from double hyphen"
```

键值：

- 必须后跟 `=`，或者当值后跟一个空格时，键必须具有一个 `--` 或 `/` 的前缀。
- 如果使用 `=`，则不是必需的。例如 `SomeKey=`。

在同一命令中，请勿将使用 `=` 的命令行参数键值对与使用空格的键值对混合使用。

Key-per-file 配置提供程序

KeyPerFileConfigurationProvider 使用目录的文件作为配置键值对。该键是文件名。该值是文件的内容。Key-per-file 配置提供程序用于 Docker 托管方案。

若要激活 Key-per-file 配置，请在 ConfigurationBuilder 的实例上调用 AddKeyPerFile 扩展方法。文件的 `directoryPath` 必须是绝对路径。

重载允许指定：

- 配置源的 `Action<KeyPerFileConfigurationSource>` 委托。
- 目录是否可选以及目录的路径。

双下划线字符 (`__`) 用作文件名中的配置键分隔符。例如，文件名 `Logging__LogLevel__System` 生成配置键 `Logging:LogLevel:System`。

构建主机时调用 `ConfigureAppConfiguration` 以指定应用的配置：

```
.ConfigureAppConfiguration((_, configuration) =>
{
    var path = Path.Combine(
        Directory.GetCurrentDirectory(), "path/to/files");

    configuration.AddKeyPerFile(directoryPath: path, optional: true);
})
```

内存配置提供程序

MemoryConfigurationProvider 使用内存中集合作为配置键值对。

以下代码将内存集合添加到配置系统中：

```
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

namespace ConsoleMemory.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((_, configuration) =>
                    configuration.AddInMemoryCollection(
                        new Dictionary<string, string>
                        {
                            ["SecretKey"] = "Dictionary MyKey Value",
                            ["TransientFaultHandlingOptions:Enabled"] = bool.TrueString,
                            ["TransientFaultHandlingOptions:AutoRetryDelay"] = "00:00:07",
                            ["Logging:LogLevel:Default"] = "Warning"
                        }));
    }
}
```

在上述代码中，MemoryConfigurationBuilderExtensions.AddInMemoryCollection(IConfigurationBuilder, IEnumerable<KeyValuePair<String,String>>) 会在默认配置提供程序之后添加内存提供程序。有关对配置提供程序进行排序的示例，请查看 XML 配置提供程序。

另请参阅

- .NET 中的配置
- .NET 通用主机
- 实现自定义配置提供程序

在 .NET 中实现自定义配置提供程序

2021/11/16 •

有许多配置提供程序可用于常见的配置源，如 JSON、XML 和 INI 文件。如果某个可用的提供程序不满足你的应用程序需要，说明你可能需要实现自定义配置提供程序。在本文中，你将了解如何实现一个依赖数据库作为配置源的自定义配置提供程序。

自定义配置提供程序

此示例应用演示了如何使用实体框架 (EF) Core 创建从数据库读取配置键值对的基本配置提供程序。

提供程序具有以下特征：

- EF 内存中数据库用于演示目的。
  - 若要使用需要连接字符串的数据库，请从临时配置中获取连接字符串。
- 提供程序在启动时将数据库表读入配置。提供程序不会基于每个键查询数据库。
- 未实现更改时重载，因此在应用启动后更新数据库对应用的配置没有任何影响。

定义一个 `Settings` 记录类型实体，用于在数据库中存储配置值。例如，可以将 Settings.cs 文件添加到“Models”文件夹中：

```
namespace CustomProvider.Example.Models
{
    public record Settings(string Id, string Value);
}
```

有关记录类型的信息，请参阅 C# 9 中的记录类型。

添加 `EntityConfigurationContext` 以存储和访问配置的值。

Providers/EntityConfigurationContext.cs：

```
using CustomProvider.Example.Models;
using Microsoft.EntityFrameworkCore;

namespace CustomProvider.Example.Providers
{
    public class EntityConfigurationContext : DbContext
    {
        private readonly string _connectionString;

        public DbSet<Settings> Settings { get; set; }

        public EntityConfigurationContext(string connectionString) =>
            _connectionString = connectionString;

        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            _ = _connectionString switch
            {
                { Length: > 0 } => optionsBuilder.UseSqlServer(_connectionString),
                _ => optionsBuilder.UseInMemoryDatabase("InMemoryDatabase")
            };
        }
    }
}
```

通过重写 OnConfiguring(DbContextOptionsBuilder)，可使用相应的数据库连接。例如，如果提供了连接字符串，则可连接到 SQL Server，否则可能要依赖内存数据库。

创建用于实现 IConfigurationSource 的类。

Providers/EntityConfigurationSource.cs：

```
using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers
{
    public class EntityConfigurationSource : IConfigurationSource
    {
        private readonly string _connectionString;

        public EntityConfigurationSource(string connectionString) =>
            _connectionString = connectionString;

        public IConfigurationProvider Build(IConfigurationBuilder builder) =>
            new EntityConfigurationProvider(_connectionString);
    }
}
```

通过从 ConfigurationProvider 继承来创建自定义配置提供程序。当数据库为空时，配置提供程序将对其进行初始化。由于配置密钥不区分大小写，因此用来初始化数据库的字典是用不区分大小写的比较程序 (StringComparer.OrdinalIgnoreCase) 创建的。

Providers/EntityConfigurationProvider.cs：

```
using System;
using System.Collections.Generic;
using System.Linq;
using CustomProvider.Example.Models;
using Microsoft.Extensions.Configuration;

namespace CustomProvider.Example.Providers
{
    public class EntityConfigurationProvider : ConfigurationProvider
    {
        private readonly string _connectionString;

        public EntityConfigurationProvider(string connectionString) =>
            _connectionString = connectionString;

        public override void Load()
        {
            using var dbContext = new EntityConfigurationContext(_connectionString);

            dbContext.Database.EnsureCreated();

            Data = dbContext.Settings.Any()
                ? dbContext.Settings.ToDictionary(c => c.Id, c => c.Value)
                : CreateAndSaveDefaultValues(dbContext);
        }

        static IDictionary<string, string> CreateAndSaveDefaultValues(
            EntityConfigurationContext context)
        {
            var settings = new Dictionary<string, string>(
                StringComparer.OrdinalIgnoreCase)
            {
                ["WidgetOptions:EndpointId"] = "b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67",
                ["WidgetOptions:DisplayLabel"] = "Widgets Incorporated, LLC.",
                ["WidgetOptions:WidgetRoute"] = "api/widgets"
            };

            context.Settings.AddRange(
                settings.Select(kvp => new Settings(kvp.Key, kvp.Value))
                        .ToArray());

            context.SaveChanges();

            return settings;
        }
    }
}
```

可以使用 `AddEntityConfiguration` 扩展方法将配置源添加到 IConfigurationBuilder 实例中。

Extensions/ConfigurationBuilderExtensions.cs：

```
using CustomProvider.Example.Providers;

namespace Microsoft.Extensions.Configuration
{
    public static class ConfigurationBuilderExtensions
    {
        public static IConfigurationBuilder AddEntityConfiguration(
            this IConfigurationBuilder builder)
        {
            var tempConfig = builder.Build();
            var connectionString =
                tempConfig.GetConnectionString("WidgetConnectionString");

            return builder.Add(new EntityConfigurationSource(connectionString));
        }
    }
}
```

IMPORTANT

使用临时配置源获取连接字符串非常重要。当前的 `builder` 通过调用 IConfigurationBuilder.Build() 和 GetConnectionString 临时构造其配置。获取连接字符串后，`builder` 将根据 `connectionString` 添加 `EntityConfigurationSource`。

下面的代码演示如何在 Program.cs 中使用自定义的 `EntityConfigurationProvider`：

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

namespace CustomProvider.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            var options = host.Services.GetRequiredService<IOptions<WidgetOptions>>().Value;
            Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
            Console.WriteLine($"EndpointId={options.EndpointId}");
            Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

            await host.RunAsync();
        }
        // Sample output:
        //    WidgetRoute=api/widgets
        //    EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
        //    DisplayLabel=Widgets Incorporated, LLC.

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((_, configuration) =>
                {
                    configuration.Sources.Clear();
                    configuration.AddEntityConfiguration();
                })
                .ConfigureServices((context, services) =>
                    services.Configure<WidgetOptions>(
                        context.Configuration.GetSection("WidgetOptions")));
    }
}
```

使用提供程序

若要使用自定义配置提供程序，可使用选项模式。准备好示例应用后，定义一个 options 对象来表示小组件设置。

```
using System;

namespace CustomProvider.Example
{
    public class WidgetOptions
    {
        public Guid EndpointId { get; set; }

        public string DisplayLabel { get; set; }

        public string WidgetRoute { get; set; }
    }
}
```

调用 ConfigureServices 可配置 options 的映射。

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

namespace CustomProvider.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            var options = host.Services.GetRequiredService<IOptions<WidgetOptions>>().Value;
            Console.WriteLine($"DisplayLabel={options.DisplayLabel}");
            Console.WriteLine($"EndpointId={options.EndpointId}");
            Console.WriteLine($"WidgetRoute={options.WidgetRoute}");

            await host.RunAsync();
        }
        // Sample output:
        //    WidgetRoute=api/widgets
        //    EndpointId=b3da3c4c-9c4e-4411-bc4d-609e2dcc5c67
        //    DisplayLabel=Widgets Incorporated, LLC.

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureAppConfiguration((_, configuration) =>
                {
                    configuration.Sources.Clear();
                    configuration.AddEntityConfiguration();
                })
                .ConfigureServices((context, services) =>
                    services.Configure<WidgetOptions>(
                        context.Configuration.GetSection("WidgetOptions")));
    }
}
```

前面的代码从配置的 `"WidgetOptions"` 部分配置 `WidgetOptions` 对象。这将启用选项模式，同时公开 EF 设置的依赖关系注入就绪 `IOptions<WidgetOptions>` 表示形式。这些选项最终是通过自定义配置提供程序提供的。

另请参阅

- .NET 中的配置
- .NET 中的配置提供程序
- .NET 中的选项模式
- .NET 中的依赖关系注入

.NET 中的选项模式

2021/11/16 •

选项模式使用类来提供对相关设置组的强类型访问。当配置设置由方案隔离到单独的类时，应用遵循两个重要软件工程原则：

- 接口分隔原则 (ISP) 或封装：依赖于配置设置的方案（类）仅依赖于其使用的配置设置。
- 关注点分离：应用的不同部件的设置不彼此依赖或相互耦合。

选项还提供验证配置数据的机制。有关详细信息，请参阅选项验证部分。

绑定分层配置

读取相关配置值的首选方法是使用选项模式。选项模式可以通过 IOptions<TOptions> 接口实现，其中泛型类型参数 `TOptions` 被约束为 `class`。以后可以通过依赖关系注入来提供 `IOptions<TOptions>`。有关详细信息，请参阅 .NET 中的依赖关系注入。

例如，若要读取以下配置值，请执行以下操作：

```
"TransientFaultHandlingOptions": {
    "Enabled": true,
    "AutoRetryDelay": "00:00:07"
},
```

创建以下 `TransientFaultHandlingOptions` 类：

```
public class TransientFaultHandlingOptions
{
    public bool Enabled { get; set; }
    public TimeSpan AutoRetryDelay { get; set; }
}
```

在使用选项模式时，选项类：

- 必须是非抽象类，有一个公共无参数构造函数
- 包含要绑定的公共读写属性（字段不绑定）

下面的代码：

- 调用 ConfigurationBinder.Bind 将 `TransientFaultHandlingOptions` 类绑定到 `"TransientFaultHandlingOptions"` 部分。
- 显示配置数据。

```
IConfigurationRoot configurationRoot = configuration.Build();

TransientFaultHandlingOptions options = new();
configurationRoot.GetSection(nameof(TransientFaultHandlingOptions))
                 .Bind(options);

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={options.AutoRetryDelay}");
```

在上面的代码中，已读取在应用启动后对 JSON 配置文件所做的更改。

`ConfigurationBinder.Get<T>` 绑定并返回指定的类型。使用 `ConfigurationBinder.Get<T>` 可能比使用 `ConfigurationBinder.Bind` 更方便。下面的代码演示如何将 `ConfigurationBinder.Get<T>` 与 `TransientFaultHandlingOptions` 类配合使用：

```
IConfigurationRoot configurationRoot = configuration.Build();

var options =
    configurationRoot.GetSection(nameof(TransientFaultHandlingOptions))
                     .Get<TransientFaultHandlingOptions>();

Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={options.AutoRetryDelay}");
```

在上面的代码中，已读取在应用启动后对 JSON 配置文件所做的更改。

> **IMPORTANT**
>
> ConfigurationBinder 类公开了几个 API，例如不被约束为 `class` 的 `.Bind(object instance)` 和 `.Get<T>()`。使用任何一个选项接口时，都必须遵守前面提到的选项类约束。

使用选项模式时的另一种方法是绑定 `"TransientFaultHandlingOptions"` 部分，并将其添加到依赖关系注入服务容器中。在以下代码中，`TransientFaultHandlingOptions` 已通过 Configure 被添加到了服务容器并已绑定到了配置：

```
services.Configure<TransientFaultHandlingOptions>(
    configurationRoot.GetSection(
        key: nameof(TransientFaultHandlingOptions)));
```

要访问 `services` 和 `configurationRoot` 对象，必须使用 ConfigureServices 方法 — IConfiguration 作为 HostBuilderContext.Configuration 属性提供。

```
Host.CreateDefaultBuilder(args)
    .ConfigureServices((context, services) =>
    {
        var configurationRoot = context.Configuration;
        services.Configure<TransientFaultHandlingOptions>(
            configurationRoot.GetSection(nameof(TransientFaultHandlingOptions)));
    });
```

> **TIP**
>
> `key` 参数是要搜索的配置部分的名称。它不必与代表它的类型名称相匹配。例如，你可以有一个名为 `"FaultHandling"` 的部分，该部分可以由 `TransientFaultHandlingOptions` 类来表示。在这种情况下，可以改为将 `"FaultHandling"` 传递到 GetSection 函数。在命名部分与其对应的类型相匹配时，为了方便，使用 `nameof` 运算符。

通过使用前面的代码，以下代码将读取位置选项：

```
using System;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example
{
    public class ExampleService
    {
        private readonly TransientFaultHandlingOptions _options;

        public ExampleService(IOptions<TransientFaultHandlingOptions> options) =>
            _options = options.Value;

        public void DisplayValues()
        {
            Console.WriteLine($"TransientFaultHandlingOptions.Enabled={_options.Enabled}");
            Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={_options.AutoRetryDelay}");
        }
    }
}
```

在上面的代码中，不会读取在应用启动后对 JSON 配置文件所做的更改。若要读取在应用启动后的更改，请使用 IOptionsSnapshot。

选项接口

IOptions<TOptions>:

- 不支持：
  - 在应用启动后读取配置数据。
  - 命名选项
- 注册为单一实例且可以注入到任何服务生存期。

IOptionsSnapshot<TOptions>:

- 对于在一定范围内或暂时生存期中应对每个注入解析重新计算选项的场景，它非常有用。有关详细信息，请参阅使用 IOptionsSnapshot 读取已更新的数据。
- 注册为范围内，因此无法注入到单一实例服务。
- 支持命名选项

IOptionsMonitor<TOptions>:

- 用于检索选项并管理 `TOptions` 实例的选项通知。
- 注册为单一实例且可以注入到任何服务生存期。
- 支持：
  - 更改通知
  - 命名选项
  - 可重载配置
  - 选择性选项失效 (IOptionsMonitorCache<TOptions>)

IOptionsFactory<TOptions> 负责新建选项实例。它具有单个 Create 方法。默认实现采用所有已注册 IConfigureOptions<TOptions> 和 IPostConfigureOptions<TOptions> 并首先运行所有配置，然后才进行后期配置。它区分 IConfigureNamedOptions<TOptions> 和 IConfigureOptions<TOptions> 且仅调用适当的接口。

IOptionsMonitorCache<TOptions> 由 IOptionsMonitor<TOptions> 用于缓存 `TOptions` 实例。IOptionsMonitorCache<TOptions> 可使监视器中的选项实例无效，以便重新计算值 (TryRemove)。可以通过 TryAdd 手动引入值。在应按需重新创建所有命名实例时使用 Clear 方法。

选项接口优点

使用泛型包装器类型，可以将选项的生存期从 DI 容器中解耦出来。IOptions<TOptions>.Value 接口对选项类型提供了一个抽象层，包括泛型约束。这提供了以下好处：

- `T` 配置实例的评估推迟到在访问 IOptions<TOptions>.Value 时进行，而不是在注入时进行。这一点很重要，因为你可以从各种不同的位置使用 `T` 选项，并选择生存期语义，而无需更改关于 `T` 的任何内容。
- 注册 `T` 类型的选项时，不需要显式注册 `T` 类型。如果你使用简单的默认值创建库，并且不想强制调用方将选项注册到具有特定生存期的 DI 容器中，这可以带来很多便利。
- 从 API 的角度来看，它允许对类型 `T` 进行约束（在本例中，`T` 被约束为引用类型）。

使用 IOptionsSnapshot 读取已更新的数据

当你使用 IOptionsSnapshot<TOptions> 时，在访问每个请求时会计算一次选项，并在请求的生存期内缓存这些选项。当使用支持读取已更新的配置值的配置提供程序时，将在应用启动后读取对配置所做的更改。

`IOptionsMonitor` 和 `IOptionsSnapshot` 之间的区别在于：

- `IOptionsMonitor` 是一种单一示例服务，可随时检索当前选项值，这在单一实例依赖项中尤其有用。
- `IOptionsSnapshot` 是一种作用域服务，并在构造 `IOptionsSnapshot<T>` 对象时提供选项的快照。选项快照旨在用于暂时性和有作用域的依赖项。

以下代码使用 IOptionsSnapshot<TOptions>。

```
using System;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example
{
    public class ScopedService
    {
        private readonly TransientFaultHandlingOptions _options;

        public ScopedService(IOptionsSnapshot<TransientFaultHandlingOptions> options) =>
            _options = options.Value;

        public void DisplayValues()
        {
            Console.WriteLine($"TransientFaultHandlingOptions.Enabled={_options.Enabled}");
            Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={_options.AutoRetryDelay}");
        }
    }
}
```

以下代码注册 `TransientFaultHandlingOptions` 绑定的配置实例：

```
services.Configure<TransientFaultHandlingOptions>(
    configurationRoot.GetSection(
        nameof(TransientFaultHandlingOptions)));
```

在上面的代码中，已读取在应用启动后对 JSON 配置文件所做的更改。

IOptionsMonitor

以下代码注册 `TransientFaultHandlingOptions` 绑定的配置实例。

```
services.Configure<TransientFaultHandlingOptions>(
    configurationRoot.GetSection(
        nameof(TransientFaultHandlingOptions)));
```

下面的示例使用 IOptionsMonitor<TOptions>：

```
using System;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example
{
    public class MonitorService
    {
        private readonly IOptionsMonitor<TransientFaultHandlingOptions> _monitor;

        public MonitorService(IOptionsMonitor<TransientFaultHandlingOptions> monitor) =>
            _monitor = monitor;

        public void DisplayValues()
        {
            TransientFaultHandlingOptions options = _monitor.CurrentValue;

            Console.WriteLine($"TransientFaultHandlingOptions.Enabled={options.Enabled}");
            Console.WriteLine($"TransientFaultHandlingOptions.AutoRetryDelay={options.AutoRetryDelay}");
        }
    }
}
```

在上面的代码中，已读取在应用启动后对 JSON 配置文件所做的更改。

> **TIP**
>
> 某些文件系统（例如 Docker 容器和网络共享）可能无法可靠地发送更改通知。在这些环境中使用 IOptionsMonitor<TOptions> 接口时，请将 `DOTNET_USE_POLLING_FILE_WATCHER` 环境变量设置为 `1` 或 `true`，以便轮询文件系统的更改。轮询更改的时间间隔是四秒，此间隔不可配置。
>
> 有关 Docker 容器的详细信息，请参阅容器化 .NET 应用。

命名选项支持使用 IConfigureNamedOptions

命名选项：

- 当多个配置节绑定到同一属性时有用。
- 区分大小写。

请考虑使用以下 appsettings.json 文件：

```
{
  "Features": {
    "Personalize": {
      "Enabled": true,
      "ApiKey": "aGEgaGEgeW91IHRob3VnaHQgdGhhdCB3YXMgcmVhbGx5IHNvbWV0aGluZw=="
    },
    "WeatherStation": {
      "Enabled": true,
      "ApiKey": "QXJlIHlvdSBhdHRlbXB0aW5nIHRvIGhhY2sgdXM/"
    }
  }
}
```

下面的类用于每个节，而不是创建两个类来绑定 `Features:Personalize` 和 `Features:WeatherStation`：

```
public class Features
{
    public const string Personalize = nameof(Personalize);
    public const string WeatherStation = nameof(WeatherStation);

    public bool Enabled { get; set; }
    public string ApiKey { get; set; }
}
```

下面的代码将配置命名选项：

```
ConfigureServices(services =>
{
    services.Configure<Features>(
        Features.Personalize,
        Configuration.GetSection("Features:Personalize"));

    services.Configure<Features>(
        Features.WeatherStation,
        Configuration.GetSection("Features:WeatherStation"));
});
```

下面的代码将显示命名选项：

```
public class Service
{
    private readonly Features _personalizeFeature;
    private readonly Features _weatherStationFeature;

    public Service(IOptionsSnapshot<Features> namedOptionsAccessor)
    {
        _personalizeFeature = namedOptionsAccessor.Get(Features.Personalize);
        _weatherStationFeature = namedOptionsAccessor.Get(Features.WeatherStation);
    }
}
```

所有选项都是命名实例。IConfigureOptions<TOptions> 实例将被视为面向 `Options.DefaultName` 实例，即 `string.Empty`。IConfigureNamedOptions<TOptions> 还可实现 IConfigureOptions<TOptions>。IOptionsFactory<TOptions> 的默认实现具有适当地使用每个实例的逻辑。`null` 命名选项用于面向所有命名实例，而不是某一特定命名实例。ConfigureAll 和 PostConfigureAll 使用此约定。

OptionsBuilder API

OptionsBuilder<TOptions> 用于配置 `TOptions` 实例。`OptionsBuilder` 简化了创建命名选项的过程，因为它只是初始 `AddOptions<TOptions>(string optionsName)` 调用的单个参数，而不会出现在所有后续调用中。选项验证和接受服务依赖关系的 `ConfigureOptions` 重载仅可通过 `OptionsBuilder` 获得。

`OptionsBuilder` 在选项验证部分中使用。

使用 DI 服务配置选项

在配置选项时，可以通过以下两种方式通过依赖关系注入访问服务：

- 将配置委托传递给 OptionsBuilder<TOptions> 上的 Configure。`OptionsBuilder<TOptions>` 提供 Configure 的重载，该重载允许使用最多五个服务来配置选项：

```
services.AddOptions<MyOptions>("optionalName")
    .Configure<ExampleService, ScopedService, MonitorService>(
        (options, es, ss, ms) =>
            options.Property = DoSomethingWith(es, ss, ms));
```

- 创建实现 IConfigureOptions<TOptions> 或 IConfigureNamedOptions<TOptions> 的类型，并将该类型注册为服务。

建议将配置委托传递给 Configure，因为创建服务较复杂。在调用 Configure 时，创建类型等效于框架执行的操作。调用 Configure 会注册临时泛型 IConfigureNamedOptions<TOptions>，它具有接受指定的泛型服务类型的构造函数。

选项验证

通过选项验证，可以验证选项值。

请考虑使用以下 appsettings.json 文件：

```
{
  "MyCustomSettingsSection": {
    "SiteTitle": "Amazing docs from Awesome people!",
    "Scale": 10,
    "VerbosityLevel": 32
  }
}
```

下面的类绑定到 `"MyCustomSettingsSection"` 配置节，并应用若干 `DataAnnotations` 规则：

```
using System;
using System.ComponentModel.DataAnnotations;

namespace ConsoleJson.Example
{
    public class SettingsOptions
    {
        public const string ConfigurationSectionName = "MyCustomSettingsSection";

        [RegularExpression(@"^[a-zA-Z''-'\s]{1,40}$")]
        public string SiteTitle { get; set; }

        [Range(0, 1000,
            ErrorMessage = "Value for {0} must be between {1} and {2}.")]
        public int Scale { get; set; }

        public int VerbosityLevel { get; set; }
    }
}
```

在前面的 `SettingsOptions` 类中，`ConfigurationSectionName` 属性包含要绑定到的配置部分的名称。在此方案中，选项对象提供其配置部分的名称。

> **TIP**
> 配置部分名称独立于所绑定到的配置对象。换句话说，名为 `"FooBarOptions"` 的配置部分可以绑定到名为 `ZedOptions` 的选项对象。虽然为二者提供相同的名称很常见，但这并 *不* 是必要的，且可能会导致名称冲突。

下面的代码：

- 调用 AddOptions 以获取绑定到 `SettingsOptions` 类的 OptionsBuilder<TOptions>。
- 调用 ValidateDataAnnotations 以使用 `DataAnnotations` 启用验证。

```
services.AddOptions<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations();
```

`ValidateDataAnnotations` 扩展方法在 Microsoft.Extensions.Options.DataAnnotations NuGet 包中定义。

下面的代码显示配置值或验证错误：

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example
{
    public class ValidationService
    {
        private readonly ILogger<ValidationService> _logger;
        private readonly IOptions<SettingsOptions> _config;

        public ValidationService(
            ILogger<ValidationService> logger,
            IOptions<SettingsOptions> config)
        {
            _config = config;
            _logger = logger;

            try
            {
                SettingsOptions options = _config.Value;
            }
            catch (OptionsValidationException ex)
            {
                foreach (string failure in ex.Failures)
                {
                    _logger.LogError(failure);
                }
            }
        }
    }
}
```

下面的代码使用委托应用更复杂的验证规则：

```
services.AddOptions<SettingsOptions>()
    .Bind(Configuration.GetSection(SettingsOptions.ConfigurationSectionName))
    .ValidateDataAnnotations()
    .Validate(config =>
    {
        if (config.Scale != 0)
        {
            return config.VerbosityLevel > config.Scale;
        }

        return true;
    }, "VerbosityLevel must be > than Scale.");
```

用于复杂验证的 IValidateOptions

下面的类实现了 IValidateOptions<TOptions>：

```
using System.Text.RegularExpressions;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Options;

namespace ConsoleJson.Example
{
    class ValidateSettingsOptions : IValidateOptions<SettingsOptions>
    {
        public SettingsOptions _settings { get; private set; }

        public ValidateSettingsOptions(IConfiguration config) =>
            _settings = config.GetSection(SettingsOptions.ConfigurationSectionName)
                              .Get<SettingsOptions>();

        public ValidateOptionsResult Validate(string name, SettingsOptions options)
        {
            string result = "";
            var rx = new Regex(@"^[a-zA-Z''-'\s]{1,40}$");
            Match match = rx.Match(options.SiteTitle);
            if (string.IsNullOrEmpty(match.Value))
            {
                result += $"{options.SiteTitle} doesn't match RegEx\n";
            }

            if (options.Scale < 0 || options.Scale > 1000)
            {
                result += $"{options.Scale} isn't within Range 0 - 1000\n";
            }

            if (_settings.Scale is 0 && _settings.VerbosityLevel <= _settings.Scale)
            {
                result += "VerbosityLevel must be > than Scale.";
            }

            return result != null
                ? ValidateOptionsResult.Fail(result)
                : ValidateOptionsResult.Success;
        }
    }
}
```

`IValidateOptions` 允许将验证代码移入类中。

使用前面的代码，使用以下代码在 `ConfigureServices` 中启用验证：

```
services.Configure<SettingsOptions>(
    Configuration.GetSection(
        SettingsOptions.ConfigurationSectionName));
services.TryAddEnumerable(
    ServiceDescriptor.Singleton
        <IValidateOptions<SettingsOptions>, ValidateSettingsOptions>());
```

选项后期配置

使用 IPostConfigureOptions<TOptions> 设置后期配置。后期配置在所有 IConfigureOptions<TOptions> 配置发生后运行，在需要重写配置的场景中非常有用：

```
services.PostConfigure<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

PostConfigure 可用于对命名选项进行后期配置：

```
services.PostConfigure<CustomOptions>("named_options_1", customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

使用 PostConfigureAll 对所有配置实例进行后期配置：

```
services.PostConfigureAll<CustomOptions>(customOptions =>
{
    customOptions.Option1 = "post_configured_option1_value";
});
```

另请参阅

- .NET 中的配置
- 面向 .NET 库创建者的选项模式指南

面向 .NET 库创建者的选项模式指南

2021/11/16 •

借助依赖关系注入，在注册服务及其相应的配置时，可以使用选项模式。通过选项模式，库（和服务）的使用者可以要求提供选项接口的实例，其中 `TOptions` 是你的选项类。通过强类型对象使用配置选项有助于确保值表示方法的一致性，让你无需再费力手动分析字符串值。有许多配置提供程序可供库使用者使用。借助这些提供程序，使用者可以通过多种方式配置库。

作为 .NET 库的创建者，你将了解有关如何正确向库的使用者公开选项模式的一般指南。有很多种方法都会达到同样的效果，并有几个注意事项。

命名约定

按照约定，负责注册服务的扩展方法被命名为 `Add{Service}`，其中 `{Service}` 是一个有意义的描述性名称。服务注册可能附带 `Use{Service}` 扩展方法，具体取决于包。`Use{Service}` 扩展方法在 ASP.NET Core 中是通用的。

✔ 考虑使用将你的服务与其他产品/服务区分开来的名称。

✘ 不要使用已是 Microsoft 官方包中 .NET 生态系统的一部分的名称。

✔ 考虑将公开扩展方法的静态类命名为 `{Type}Extensions`，其中 `{Type}` 是你要扩展的类型。

命名空间指南

Microsoft 包利用 `Microsoft.Extensions.DependencyInjection` 命名空间来统一各种服务产品/服务的注册。

✔ 考虑使用清楚标识包产品/服务的命名空间。

✘ 不要将 `Microsoft.Extensions.DependencyInjection` 命名空间用于非官方的 Microsoft 包。

无参数

如果你的服务可以用最少的显式配置或不需要显式配置来工作，请考虑使用无参数扩展方法。

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection
{
    public static class ServiceCollectionExtensions
    {
        public static IServiceCollection AddMyLibraryService(
            this IServiceCollection services)
        {
            services.AddOptions<LibraryOptions>()
                .Configure(options =>
                {
                    // Specify default option values
                });

            // Register lib services here...
            // services.AddScoped<ILibraryService, DefaultLibraryService>();

            return services;
        }
    }
}
```

在上述代码中，`AddMyLibraryService` 执行以下操作：

- 扩展 IServiceCollection 的实例
- 调用类型参数为 `LibraryOptions` 的 OptionsServiceCollectionExtensions.AddOptions<TOptions>(IServiceCollection)
- 链接对 Configure 的调用，用来指定默认选项值

`IConfiguration` 参数

在创建向使用者公开许多选项的库时，可能需要考虑要求使用 `IConfiguration` 参数扩展方法。应使用 IConfiguration.GetSection 函数，将预期的 `IConfiguration` 实例的作用域限定为此配置的已命名部分。

```
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection
{
    public static class ServiceCollectionExtensions
    {
        public static IServiceCollection AddMyLibraryService(
          this IServiceCollection services,
          IConfiguration namedConfigurationSection)
        {
            // Default library options are overridden
            // by bound configuration values.
            services.Configure<LibraryOptions>(namedConfigurationSection);

            // Register lib services here...
            // services.AddScoped<ILibraryService, DefaultLibraryService>();

            return services;
        }
    }
}
```

TIP

Configure<TOptions>(IServiceCollection, IConfiguration) 方法是 `Microsoft.Extensions.Options.ConfigurationExtensions` NuGet 包的一部分。

在上述代码中，`AddMyLibraryService` 执行以下操作：

- 扩展 IServiceCollection 的实例
- 定义 IConfiguration 参数 `namedConfigurationSection`
- 调用要传递 `LibraryOptions` 的泛型类型参数和 `namedConfigurationSection` 实例的 Configure<TOptions>(IServiceCollection, IConfiguration) 以进行配置

此模式中的使用者提供已命名部分已限定作用域的 `IConfiguration` 实例：

```
using System.Threading.Tasks;
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Options.ConfigParam
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((context, services) =>
                {
                    services.AddMyLibraryService(
                        context.Configuration.GetSection("LibraryOptions"));
                });
    }
}
```

对 `.AddMyLibraryService` 的调用是在 ConfigureServices 方法中进行的。同样地，使用 `Startup` 类时，会在 `ConfigureServices` 中添加注册的服务。

作为库创建者，由你指定默认值。

> **NOTE**
>
> 可以将配置绑定到选项实例。但是，存在名称冲突的风险，冲突将导致错误。此外，当以这种方式手动绑定时，将选项模式的使用限制为读取一次。对设置的更改将不会被重新绑定，因为这样的话，使用者就无法使用 IOptionsMonitor 接口。
>
> ```
> services.AddOptions<LibraryOptions>()
>     .Configure<IConfiguration>(
>         (options, configuration) =>
>             configuration.GetSection("LibraryOptions").Bind(options));
> ```

`Action<TOptions>` 参数

库的使用者可能有兴趣提供一个 Lambda 表达式来生成选项类的实例。在此场景中，你在你的扩展方法中定义了一个 `Action<LibraryOptions>` 参数。

```
using System;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection
{
    public static class ServiceCollectionExtensions
    {
        public static IServiceCollection AddMyLibraryService(
          this IServiceCollection services,
          Action<LibraryOptions> configureOptions)
        {
            services.Configure(configureOptions);

            // Register lib services here...
            // services.AddScoped<ILibraryService, DefaultLibraryService>();

            return services;
        }
    }
}
```

在上述代码中，`AddMyLibraryService` 执行以下操作：

- 扩展 IServiceCollection 的实例
- 定义一个 Action<T> 参数 `configureOptions`，其中 `T` 为 `LibraryOptions`
- 根据 `configureOptions` 操作调用 Configure

此模式下的使用者提供一个 Lambda 表达式（或一个符合 `Action<LibraryOptions>` 参数的委托）：

```
using System.Threading.Tasks;
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Options.Action
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices(services =>
                {
                    services.AddMyLibraryService(options =>
                    {
                        // User defined option values
                        // options.SomePropertyValue = ...
                    });
                });
    }
}
```

选项实例参数

库的使用者可能更倾向于提供一个内联的选项实例。在此场景中，你公开一个扩展方法，此方法采用选项对象的

实例 `LibraryOptions` 。

```
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection
{
    public static class ServiceCollectionExtensions
    {
        public static IServiceCollection AddMyLibraryService(
          this IServiceCollection services,
          LibraryOptions userOptions)
        {
            services.AddOptions<LibraryOptions>()
                .Configure(options =>
                {
                    // Overwrite default option values
                    // with the user provided options.
                    // options.SomeValue = userOptions.SomeValue;
                });

            // Register lib services here...
            // services.AddScoped<ILibraryService, DefaultLibraryService>();

            return services;
        }
    }
}
```

在上述代码中，`AddMyLibraryService` 执行以下操作：

- 扩展 IServiceCollection 的实例
- 调用类型参数为 `LibraryOptions` 的 OptionsServiceCollectionExtensions.AddOptions<TOptions> (IServiceCollection)
- 链接对 Configure 的调用，指定可从给定 `userOptions` 实例中替代的默认选项值

此模式下的使用者提供 `LibraryOptions` 类的实例，以内联方式定义所需的属性值：

```
using System.Threading.Tasks;
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Options.Object
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices(services =>
                {
                    services.AddMyLibraryService(new LibraryOptions
                    {
                        // Specify option values
                        // SomePropertyValue = ...
                    });
                });
    }
}
```

配置后

绑定或指定所有配置选项值后，便可以使用发布配置功能。通过公开前面详述的同一 `Action<TOptions>` 参数，可以选择调用 PostConfigure。发布配置在进行所有 `.Configure` 调用之后运行。

```
using System;
using Microsoft.Extensions.DependencyInjection;

namespace ExampleLibrary.Extensions.DependencyInjection
{
    public static class ServiceCollectionExtensions
    {
        public static IServiceCollection AddMyLibraryService(
          this IServiceCollection services,
          Action<LibraryOptions> configureOptions)
        {
            services.PostConfigure(configureOptions);

            // Register lib services here...
            // services.AddScoped<ILibraryService, DefaultLibraryService>();

            return services;
        }
    }
}
```

在上述代码中，`AddMyLibraryService` 执行以下操作：

- 扩展 IServiceCollection 的实例
- 定义一个 Action<T> 参数 `configureOptions`，其中 `T` 为 `LibraryOptions`
- 根据 `configureOptions` 操作调用 PostConfigure

此模式下的使用者提供一个 Lambda 表达式(或一个符合 `Action<LibraryOptions>` 参数的委托)，就如同在非发布配置场景中，使用者使用 `Action<TOptions>` 参数提供一样：

```
using System.Threading.Tasks;
using ExampleLibrary.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Options.PostConfig
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices(services =>
                {
                    services.AddMyLibraryService(options =>
                    {
                        // Specify option values
                        // options.SomePropertyValue = ...
                    });
                });
    }
}
```

另请参阅

- .NET 中的选项模式
- .NET 中的依赖关系注入
- 依赖关系注入指南

.NET 中的日志记录

2021/11/16 •

.NET 支持适用于各种内置和第三方日志记录提供程序的日志记录 API。本文介绍了如何将日志记录 API 与内置提供程序一起使用。本文中提供的大多数代码示例都适用于使用通用主机的 .NET 应用。对于不使用通用主机的应用，请参阅非主机控制台应用。

> **TIP**
> 所有日志记录示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的日志记录。

创建日志

若要创建日志，请使用依赖关系注入 (DI) 中的 ILogger<TCategoryName> 对象。

如下示例中：

- 创建一个记录器 `ILogger<Worker>`，该记录器使用类型为 `Worker` 的完全限定名称的日志类别。日志类别是与每个日志关联的字符串。
- 调用 LogInformation 以在 `Information` 级别登录。日志"级别"代表所记录事件的严重程度 。

```
public class Worker : BackgroundService
{
    private readonly ILogger<Worker> _logger;

    public Worker(ILogger<Worker> logger) =>
        _logger = logger;

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Worker running at: {time}", DateTimeOffset.UtcNow);
            await Task.Delay(1000, stoppingToken);
        }
    }
}
```

本文稍后部分将更详细地介绍级别和类别。

配置日志记录

日志配置通常由 appsettings `{Environment}`.json 文件的 `Logging` 部分提供 。以下 appsettings.Development.json 文件由 .NET 辅助角色服务模板生成：

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
}
```

在上述 JSON 中：

- 指定了 `"Default"`、`"Microsoft"` 和 `"Microsoft.Hosting.Lifetime"` 类别。
- `"Microsoft"` 类别适用于以 `"Microsoft"` 开头的所有类别。
- `"Microsoft"` 类别在日志级别 `Warning` 或更高级别记录。
- `"Microsoft.Hosting.Lifetime"` 类别比 `"Microsoft"` 类别更具体，因此 `"Microsoft.Hosting.Lifetime"` 类别在日志级别“Information”和更高级别记录。
- 未指定特定的日志提供程序，因此 `LogLevel` 适用于所有启用的日志记录提供程序，但 Windows EventLog 除外。

`Logging` 属性可以具有 LogLevel 和日志提供程序属性。`LogLevel` 指定要针对所选类别进行记录的最低级别。在前面的 JSON 中，指定了 `Information` 和 `Warning` 日志级别。`LogLevel` 表示日志的严重性，范围为 0 到 6：

`Trace` = 0、`Debug` = 1、`Information` = 2、`Warning` = 3、`Error` = 4、`Critical` = 5 和 `None` = 6。

指定 `LogLevel` 时，将为指定级别和更高级别的消息启用日志记录。在前面的 JSON 中，记录了 `Information` 及更高级别的 `Default` 类别。例如，记录了 `Information`、`Warning`、`Error` 和 `Critical` 消息。如果未指定 `LogLevel`，则日志记录默认为 `Information` 级别。有关详细信息，请参阅日志级别。

提供程序属性可以指定 `LogLevel` 属性。提供程序下的 `LogLevel` 指定要为该提供程序记录的级别，并替代非提供程序日志设置。请考虑使用以下 appsettings.json 文件：

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Error",
            "Microsoft": "Warning"
        },
        "Debug": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft.Hosting": "Trace"
            }
        },
        "EventSource": {
            "LogLevel": {
                "Default": "Warning"
            }
        }
    }
}
```

`Logging.{ProviderName}.LogLevel` 中的设置将替代 `Logging.LogLevel` 中的设置。在前面的 JSON 中，`Debug` 提供程序的默认日志级别设置为 `Information`：

`Logging:Debug:LogLevel:Default:Information`

前面的设置为每个 `Logging:Debug:` 类别（`Microsoft.Hosting` 除外）指定 `Information` 日志级别。当列出特定类别时，该特定类别将替代默认类别。在前面的 JSON 中，`Logging:Debug:LogLevel` 类别 `"Microsoft.Hosting"` 和

`"Default"` 替代 `Logging:LogLevel` 中的设置

可以为以下任何一项指定最低日志级别：

- 特定提供程序：例如，`Logging:EventSource:LogLevel:Default:Information`
- 特定类别：例如，`Logging:LogLevel:Microsoft:Warning`
- 所有提供程序和所有类别：`Logging:LogLevel:Default:Warning`

低于最低级别的任何日志均不会执行以下操作：

- 传递到提供程序。
- 记录或显示。

要阻止所有日志，请指定 LogLevel.None。`LogLevel.None` 的值为 6，该值高于 `LogLevel.Critical` (5)。

如果提供程序支持日志作用域，则 `IncludeScopes` 将指示是否启用这些域。有关详细信息，请参阅日志范围

以下 appsettings.json 文件包含所有内置提供程序的设置：

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Error",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Warning"
        },
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft.Extensions.Hosting": "Warning",
                "Default": "Information"
            }
        },
        "EventSource": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "EventLog": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "AzureAppServicesFile": {
            "IncludeScopes": true,
            "LogLevel": {
                "Default": "Warning"
            }
        },
        "AzureAppServicesBlob": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft": "Information"
            }
        },
        "ApplicationInsights": {
            "LogLevel": {
                "Default": "Information"
            }
        }
    }
}
```

在上述示例中：

- 类别和级别不是建议的值。提供该示例是为了显示所有默认提供程序。
- `Logging.{ProviderName}.LogLevel` 中的设置将替代 `Logging.LogLevel` 中的设置。例如，`Debug.LogLevel.Default` 中的级别将替代 `LogLevel.Default` 中的级别。
- 将使用每个提供程序的别名。每个提供程序都定义了一个别名；可在配置中使用该别名来代替完全限定的类型名称 。内置提供程序的别名包括：
  - 控制台
  - 调试
  - EventSource
  - EventLog
  - AzureAppServicesFile
  - AzureAppServicesBlob

- ApplicationInsights

通过命令行、环境变量和其他配置设置日志级别

日志级别可以由任何配置提供程序设置。例如，可以创建一个名为 `Logging:LogLevel:Microsoft` 且值为 `Information` 的持久性环境变量。

- 命令行
- PowerShell
- Bash

在给定日志级别值的情况下，创建并分配持久性环境变量。

```
:: Assigns the env var to the value
setx "Logging__LogLevel__Microsoft" "Information" /M
```

在命令提示符的新实例中，读取环境变量。

```
:: Prints the env var value
echo %Logging__LogLevel__Microsoft%
```

前面的环境设置会在环境中持续存在。若要在使用通过 .NET 辅助角色服务模板创建的应用时测试这些设置，请在分配环境变量后，在项目目录中使用 `dotnet run` 命令。

```
dotnet run
```

> **TIP**
> 设置环境变量后，请重启集成开发环境 (IDE)，以确保新添加的环境变量可用。

在 Azure 应用服务上，选择"设置">"配置"页面上的"新应用程序设置"。Azure 应用服务应用程序设置：

- 已静态加密且通过加密的通道进行传输。
- 已作为环境变量公开。

若要详细了解如何使用环境变量设置 .NET 配置值，请参阅环境变量。

如何应用筛选规则

创建 ILogger<TCategoryName> 对象时，ILoggerFactory 对象将根据提供程序选择一条规则，将其应用于该记录器。将按所选规则筛选 `ILogger` 实例写入的所有消息。从可用规则中为每个提供程序和类别对选择最具体的规则。

在为给定的类别创建 `ILogger` 时，以下算法将用于每个提供程序：

- 选择匹配提供程序或其别名的所有规则。如果找不到任何匹配项，则选择提供程序为空的所有规则。
- 根据上一步的结果，选择具有最长匹配类别前缀的规则。如果找不到任何匹配项，则选择未指定类别的所有规则。
- 如果选择了多条规则，则采用最后一条。
- 如果未选择任何规则，请使用 LoggingBuilderExtensions.SetMinimumLevel(ILoggingBuilder, LogLevel) 指定最小日志记录级别。

日志类别

创建 `ILogger` 对象时，将指定类别。该类别包含在由此 `ILogger` 实例创建的每条日志消息中。类别字符串是任意的，但约定将使用类名称。例如，在服务定义类似于以下对象的应用程序中，类别可能为 `"Example.DefaultService"`：

```
namespace Example
{
    public class DefaultService : IService
    {
        private readonly ILogger<DefaultService> _logger;

        public DefaultService(ILogger<DefaultService> logger) =>
            _logger = logger;

        // ...
    }
}
```

要显式指定类别，请调用 LoggerFactory.CreateLogger：

```
namespace Example
{
    public class DefaultService : IService
    {
        private readonly ILogger _logger;

        public DefaultService(ILoggerFactory loggerFactory) =>
            _logger = loggerFactory.CreateLogger("CustomCategory");

        // ...
    }
}
```

在多个类/类型中使用时，使用固定名称调用 `CreateLogger` 很有用，这样可以按类别组织事件。

`ILogger<T>` 相当于使用 `T` 的完全限定类型名称来调用 `CreateLogger`。

日志级别

下表列出了 LogLevel 值、方便的 `Log{LogLevel}` 扩展方法以及建议的用法：

LOGLEVEL	“I”	II	II
Trace	0	LogTrace	包含最详细的消息。这些消息可能包含敏感的应用数据。这些消息默认情况下处于禁用状态，并且不应在生产中启用。
调试	1	LogDebug	用于调试和开发。由于量大，请在生产中小心使用。
信息	2	LogInformation	跟踪应用的常规流。可能具有长期值。
警告	3	LogWarning	对于异常事件或意外事件。通常包括不会导致应用失败的错误或情况。

LOGLEVEL	"["	[[	[[
错误	4	LogError	表示无法处理的错误和异常。这些消息表示当前操作或请求失败，而不是整个应用失败。
严重	5	LogCritical	需要立即关注的失败。例如数据丢失、磁盘空间不足。
无	6		指定不应写入任何消息。

在上表中，`LogLevel` 按严重性由低到高的顺序列出。

Log 方法的第一个参数 LogLevel 指示日志的严重性。大多数开发人员调用 Log{LogLevel} 扩展方法，而不调用 `Log(LogLevel, ...)`。`Log{LogLevel}` 扩展方法调用 Log 方法并指定 LogLevel。例如，以下两个日志记录调用功能相同，并生成相同的日志：

```
public void LogDetails()
{
    var logMessage = "Details for log.";

    _logger.Log(LogLevel.Information, AppLogEvents.Details, logMessage);
    _logger.LogInformation(AppLogEvents.Details, logMessage);
}
```

`AppLogEvents.Details` 为事件 ID，用常量 Int32 值隐式表示。`AppLogEvents` 是一个公开各种命名的标识符常量并显示在日志事件 ID 部分中的类。

下面的代码会创建 `Information` 和 `Warning` 日志：

```
public async Task<T> GetAsync<T>(string id)
{
    _logger.LogInformation(AppLogEvents.Read, "Reading value for {Id}", id);

    var result = await _repository.GetAsync(id);
    if (result is null)
    {
        _logger.LogWarning(AppLogEvents.ReadNotFound, "GetAsync({Id}) not found", id);
    }

    return result;
}
```

在前面的代码中，第一个 `Log{LogLevel}` 参数 `AppLogEvents.Read` 是日志事件 ID。第二个参数是消息模板，其中的占位符用于填写剩余方法形参提供的实参值。稍后将在本文的消息模板部分介绍方法参数。

配置适当的日志级别并调用正确的 `Log{LogLevel}` 方法来控制写入特定存储介质的日志输出量。例如：

- 生产中：
  - 在 `Trace` 或 `Information` 级别记录日志会产生大量详细的日志消息。为了控制成本且不超过数据存储限制，请将 `Trace` 和 `Information` 级别消息记录到容量大、成本低的数据存储中。考虑将 `Trace` 和 `Information` 限制为特定类别。
  - 从 `Warning` 到 `Critical` 级别的日志记录应该很少产生日志消息。
    - 成本和存储限制通常不是问题。
    - 很少有日志可以为数据存储选择提供更大的灵活性。
- 在开发过程中：

- 设置为 `Warning`。
- 在进行故障排除时，添加 `Trace` 或 `Information` 消息。若要限制输出，请仅对正在调查的类别设置 `Trace` 或 `Information`。

以下 JSON 设置了 `Logging:Console:LogLevel:Microsoft:Information`：

```
{
    "Logging": {
        "LogLevel": {
            "Microsoft": "Warning"
        },
        "Console": {
            "LogLevel": {
                "Microsoft": "Information"
            }
        }
    }
}
```

日志事件 ID

每个日志可指定一个事件标识符，EventId 是一个具有 `Id` 和可选 `Name` 只读属性的结构。示例源代码使用 `AppLogEvents` 类来定义事件 ID：

```
internal static class AppLogEvents
{
    internal const int Create = 1000;
    internal const int Read = 1001;
    internal const int Update = 1002;
    internal const int Delete = 1003;

    internal const int Details = 3000;
    internal const int Error = 3001;

    internal const int ReadNotFound = 4000;
    internal const int UpdateNotFound = 4001;

    // ...
}
```

事件 ID 与一组事件相关联。例如，与从存储库读取值相关的所有日志都可能是 `1001`。

日志记录提供程序可将事件 ID 记录在 ID 字段中，记录在日志记录消息中，或者不进行记录。调试提供程序不显示事件 ID。控制台提供程序在类别后的括号中显示事件 ID：

```
info: Example.DefaultService.GetAsync[1001]
      Reading value for a1b2c3
warn: Example.DefaultService.GetAsync[4000]
      GetAsync(a1b2c3) not found
```

一些日志记录提供程序将事件 ID 存储在一个字段中，该字段允许对 ID 进行筛选。

日志消息模板

每个日志 API 都使用一个消息模板。消息模板可包含要填写参数的占位符。请在占位符中使用名称而不是数字。占位符的顺序（而非其名称）决定了为其提供值的参数。在以下代码中，消息模板中的参数名称不按顺序排列：

```
string p1 = "param1";
string p2 = "param2";
_logger.LogInformation("Parameter values: {p2}, {p1}", p1, p2);
```

上面的代码按顺序通过参数值创建日志消息：

```
Parameter values: param1, param2
```

此方法允许日志记录提供程序实现语义或结构化日志记录。参数本身会传递给日志记录系统，而不仅仅是格式化的消息模板。这使日志记录提供程序可以将参数值存储为字段。请考虑使用以下记录器方法：

```
_logger.LogInformation("Getting item {Id} at {RunTime}", id, DateTime.Now);
```

例如，登录到 Azure 表存储时：

- 每个 Azure 表实体都可以有 `ID` 和 `RunTime` 属性。
- 具有属性的表简化了对记录数据的查询。例如，查询可以找到特定 `RunTime` 范围内的所有日志，而不必分析文本消息中的时间。

记录异常

记录器方法的重载采用异常参数：

```
public void Test(string id)
{
    try
    {
        if (id == "none")
        {
            throw new Exception("Default Id detected.");
        }
    }
    catch (Exception ex)
    {
        _logger.LogWarning(
            AppLogEvents.Error, ex,
            "Failed to process iteration: {Id}", id);
    }
}
```

异常日志记录是特定于提供程序的。

默认日志级别

如果未设置默认日志级别，则默认的日志级别值为 `Information`。

例如，请考虑以下辅助角色服务应用：

- 使用 .NET 辅助角色模板创建的应用。
- 已删除 appsettings.json 和 appsettings.Development.json 或对其进行重命名 。

使用上述设置，导航到隐私或主页会生成许多 `Trace`、`Debug` 和 `Information` 消息，并在类别名称中包含 `Microsoft`。

如果未在配置中设置默认日志级别，以下代码会设置默认日志级别：

```
class Program
{
    static Task Main(string[] args) =>
        CreateHostBuilder(args).Build().RunAsync();

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging => logging.SetMinimumLevel(LogLevel.Warning));
}
```

筛选器函数

对配置或代码没有向其分配规则的所有提供程序和类别调用筛选器函数：

```
class Program
{
    static Task Main(string[] args) =>
        CreateHostBuilder(args).Build().RunAsync();

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter((provider, category, logLevel) =>
                {
                    return provider.Contains("ConsoleLoggerProvider")
                        && (category.Contains("Example") || category.Contains("Microsoft"))
                        && logLevel >= LogLevel.Information;
                }));
}
```

如果类别包含 `Example` 或 `Microsoft`，并且日志级别为 `Information` 或更高级别，以上代码会显示控制台日志。

日志作用域

“作用域”可对一组逻辑操作分组 。此分组可用于将相同的数据附加到作为集合的一部分而创建的每个日志。例如，在处理事务期间创建的每个日志都可包括事务 ID。

范围：

- 是 BeginScope 方法返回的 IDisposable 类型。
- 持续到处置完毕。

以下提供程序支持范围：

- `Console`
- AzureAppServicesFile 和 AzureAppServicesBlob

要使用作用域，请在 `using` 块中包装记录器调用：

```
public async Task<T> GetAsync<T>(string id)
{
    T result;

    using (_logger.BeginScope("using block message"))
    {
        _logger.LogInformation(
            AppLogEvents.Read, "Reading value for {Id}", id);

        var result = await _repository.GetAsync(id);
        if (result is null)
        {
            _logger.LogWarning(
                AppLogEvents.ReadNotFound, "GetAsync({Id}) not found", id);
        }
    }

    return result;
}
```

以下 JSON 为控制台提供程序启用范围：

```
{
    "Logging": {
        "Debug": {
            "LogLevel": {
                "Default": "Information"
            }
        },
        "Console": {
            "IncludeScopes": true,
            "LogLevel": {
                "Microsoft": "Warning",
                "Default": "Information"
            }
        },
        "LogLevel": {
            "Default": "Debug"
        }
    }
}
```

下列代码为控制台提供程序启用作用域：

```
class Program
{
    static Task Main(string[] args) =>
        CreateHostBuilder(args).Build().RunAsync();

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging((_, logging) =>
                logging.ClearProviders()
                    .AddConsole(options => options.IncludeScopes = true));
}
```

非托管控制台应用

对于没有通用主机的应用，日志记录代码在添加提供程序和创建记录器的方式上有所不同。在非主机控制台应用中，在创建 `LoggerFactory` 时调用提供程序的 `Add{provider name}` 扩展方法：

```
class Program
{
    static void Main(string[] args)
    {
        using var loggerFactory = LoggerFactory.Create(builder =>
        {
            builder
                .AddFilter("Microsoft", LogLevel.Warning)
                .AddFilter("System", LogLevel.Warning)
                .AddFilter("LoggingConsoleApp.Program", LogLevel.Debug)
                .AddConsole();
        });

        ILogger logger = loggerFactory.CreateLogger<Program>();
        logger.LogInformation("Example log message");
    }
}
```

`loggerFactory` 对象用于创建 ILogger 实例。

在 Main 中创建日志

以下代码通过在构建主机之后从 DI 获取 `ILogger` 实例来登录 `Main`：

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

class Program
{
    static Task Main(string[] args)
    {
        IHost host = Host.CreateDefaultBuilder(args).Build();

        var logger = host.Services.GetRequiredService<ILogger<Program>>();
        logger.LogInformation("Host created.");

        return host.RunAsync();
    }
}
```

没有异步记录器方法

日志记录应该会很快，不值得牺牲性能来使用异步代码。如果日志记录数据存储很慢，请不要直接写入它。考虑先将日志消息写入快速存储，然后再将其移至慢速存储。例如，登录到 SQL Server 时，请勿直接使用 `Log` 方法登录，因为 `Log` 方法是同步的。相反，你会将日志消息同步添加到内存中的队列，并让后台辅助线程从队列中拉出消息，以完成将数据推送到 SQL Server 的异步工作。

更改正在运行的应用中的日志级别

不可使用日志记录 API 在应用运行时更改日志记录。但是，一些配置提供程序可重新加载配置，这将对日志记录配置立即产生影响。例如，文件配置提供程序默认情况下会重载日志记录配置。如果在应用运行时在代码中更改了配置，则该应用可调用 IConfigurationRoot.Reload 来更新应用的日志记录配置。

NuGet 包

ILogger<TCategoryName> 和 ILoggerFactory 接口和实现都包含在 .NET SDK 中。它们还可以通过以下 NuGet 包获得：

- 这些接口位于 Microsoft.Extensions.Logging.Abstractions 中。
- 默认实现位于 Microsoft.Extensions.Logging 中。

在代码中应用日志筛选器规则

设置日志筛选器规则的首选方法是使用配置。

下面的示例演示了如何在代码中注册筛选规则：

```
class Program
{
    static Task Main(string[] args) =>
        CreateHostBuilder(args).Build().RunAsync();

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
               logging.AddFilter("System", LogLevel.Debug)
                  .AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)
                  .AddFilter<ConsoleLoggerProvider>("Microsoft", LogLevel.Trace));
}
```

`logging.AddFilter("System", LogLevel.Debug)` 指定 `System` 类别和日志级别 `Debug`。筛选器将应用于所有提供程序，因为未配置特定的提供程序。

`AddFilter<DebugLoggerProvider>("Microsoft", LogLevel.Information)` 指定以下项：

- `Debug` 日志记录提供程序。
- 日志级别 `Information` 及更高级别。
- 以 `"Microsoft"` 开头的所有类别。

另请参阅

- .NET 中的日志记录提供程序
- 在 .NET 中实现自定义日志记录提供程序
- 控制台日志格式设置
- .NET 中的高性能日志记录
- 应在 github.com/dotnet/extensions 存储库中创建日志记录 bug

.NET 中的日志记录提供程序

2021/11/16 •

日志提供程序会保留日志，但 `Console` 提供程序除外，后者仅将日志显示为标准输出。例如，Azure Application Insights 提供程序将日志存储在 Azure Application Insights 中。可以启用多个提供程序。

默认的 .NET 辅助角色应用模板：

- 使用通用主机。
- 调用 CreateDefaultBuilder，这将添加以下日志记录提供程序：
  - 控制台
  - 调试
  - EventSource
  - EventLog（仅限 Windows）

```
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;

namespace Console.Example
{
    class Program
    {
        static async Task Main(string[] args)
        {
            using IHost host = CreateHostBuilder(args).Build();

            // Application code should start here.

            await host.RunAsync();
        }

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args);
    }
}
```

上面的代码显示了使用 .NET 辅助角色应用模板创建的 `Program` 类。接下来的几个部分提供了一些示例，它们基于使用通用主机的 .NET 辅助角色应用模板。

若要替代 `Host.CreateDefaultBuilder` 添加的默认日志记录提供程序集，请调用 `ClearProviders` 并添加所需的日志记录提供程序。例如，以下代码：

- 调用 ClearProviders 以从生成器中删除所有 ILoggerProvider 实例。
- 添加控制台日志记录提供程序。

```
static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureLogging(logging =>
        {
            logging.ClearProviders();
            logging.AddConsole();
        });
```

有关其他提供程序，请参阅：

- 内置日志记录提供程序。
- 第三方日志记录提供程序。

配置依赖于 ILogger 的服务

若要配置依赖于 `ILogger<T>` 的服务，请使用构造函数注入或提供工厂方法。只有在没有其他选择的情况下，才建议使用工厂方法。例如，假设某个服务需要由 DI 提供的 `ILogger<T>` 实例：

```
.ConfigureServices(services =>
    services.AddSingleton<IExampleService>(container =>
        new DefaultExampleService
        {
            Logger = container.GetRequiredService<ILogger<IExampleService>>()
        }));
```

上述代码是 Func<IServiceProvider, IExampleService>，它在 DI 容器首次需要构造 `IExampleService` 实例时运行。可以用这种方式访问任何已注册的服务。

内置日志记录提供程序

Microsoft 扩展包含以下日志记录提供程序作为运行时库的一部分：

- 控制台
- 调试
- EventSource
- EventLog

以下日志记录提供程序由 Microsoft 提供，但不是运行时库的一部分。它们必须作为附加 NuGet 包安装。

- AzureAppServicesFile 和 AzureAppServicesBlob
- ApplicationInsights

控制台

`Console` 提供程序将输出记录到控制台。

调试

`Debug` 提供程序使用 System.Diagnostics.Debug 类，特别是通过 Debug.WriteLine 方法写入日志输出。DebugLoggerProvider 创建 DebugLogger 实例，这些实例是 `ILogger` 接口的实现。

在 Linux 上，`Debug` 提供程序日志位置取决于分发，并且可以是以下位置之一：

- */var/log/message*
- */var/log/syslog*

事件来源

`EventSource` 提供程序写入名称为 `Microsoft-Extensions-Logging` 的跨平台事件源。在 Windows 上，提供程序使用的是 ETW。

dotnet 跟踪工具

dotnet-trace 工具是一种跨平台 CLI 全局工具，可用于收集正在运行的进程的 .NET Core 跟踪。该工具会使用 LoggingEventSource 收集 Microsoft.Extensions.Logging.EventSource 提供程序数据。

有关安装说明，请参阅 dotnet-trace。有关使用 `dotnet-trace` 的诊断教程，请查看在 .NET Core 中调试 CPU 使用率高的问题。

Windows 事件日志

`EventLog` 提供程序将日志输出发送到 Windows 事件日志。与其他提供程序不同，`EventLog` 提供程序不继承默认的非提供程序设置。如果未指定 `EventLog` 日志设置，则它们默认为 `LogLevel.Warning`。

若要记录低于 LogLevel.Warning 的事件，请显式设置日志级别。以下示例将事件日志的默认日志级别设置为 LogLevel.Information：

```
"Logging": {
  "EventLog": {
    "LogLevel": {
      "Default": "Information"
    }
  }
}
```

AddEventLog 重载可以传入 EventLogSettings。如果为 `null` 或未指定，则使用以下默认设置：

- `LogName`："Application"
- `SourceName`："".NET Runtime"
- `MachineName`：使用本地计算机名称。

以下代码将 `SourceName` 从默认值 `".NET Runtime"` 更改为 `CustomLogs`：

```
public class Program
{
    static async Task Main(string[] args)
    {
        using IHost host = CreateHostBuilder(args).Build();

        // Application code should start here.

        await host.RunAsync();
    }

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddEventLog(configuration =>
                    configuration.SourceName = "CustomLogs"));
}
```

Azure 应用服务

Microsoft.Extensions.Logging.AzureAppServices 提供程序包将日志写入 Azure App Service 应用的文件系统，以及 Azure 存储帐户中的 blob 存储。

运行时库中不包括该提供程序包。若要使用提供程序，请将提供程序包添加到项目。

要配置提供程序设置，请使用 AzureFileLoggerOptions 和 AzureBlobLoggerOptions，如以下示例所示：

```
class Program
{
    static async Task Main(string[] args)
    {
        using IHost host = CreateHostBuilder(args).Build();

        // Application code should start here.

        await host.RunAsync();
    }

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddAzureWebAppDiagnostics())
            .ConfigureServices(services =>
                services.Configure<AzureFileLoggerOptions>(options =>
                {
                    options.FileName = "azure-diagnostics-";
                    options.FileSizeLimit = 50 * 1024;
                    options.RetainedFileCountLimit = 5;
                })
                .Configure<AzureBlobLoggerOptions>(options =>
                {
                    options.BlobName = "log.txt";
                }));
}
```

部署到 Azure 应用服务时，应用使用 Azure 门户的“应用服务”页面的应用服务日志部分中的设置。更新以下设置后，更改立即生效，无需重启或重新部署应用。

- 应用程序日志记录(Filesystem)
- 应用程序日志记录(Blob)

日志文件的默认位置是 D:\home\LogFiles\Application 文件夹，默认文件名为 diagnostics-yyyymmdd.txt 。默认文件大小上限为 10 MB，默认最大保留文件数为 2。默认 blob 名为 {app-name}{timestamp}/yyyy/mm/dd/hh/{guid}-applicationLog.txt 。

仅当项目在 Azure 环境中运行时，此提供程序才记录日志。

Azure 日志流式处理

Azure 日志流式处理支持从以下位置实时查看日志活动：

- 应用服务器
- Web 服务器
- 请求跟踪失败

要配置 Azure 日志流式处理，请执行以下操作：

- 从应用的门户页导航到“应用服务日志”页。
- 将“应用程序日志记录(Filesystem)”设置为“开”。
- 选择日志级别 。此设置仅适用于 Azure 日志流式处理。

导航到“日志流”页面以查看日志。记录的消息使用 `ILogger` 接口进行记录。

Azure Application Insights

Microsoft.Extensions.Logging.ApplicationInsights 提供程序包将日志写入 Azure Application Insights。Application Insights 是一项服务，可监视 Web 应用并提供用于查询和分析遥测数据的工具。如果使用此提供程序，则可以使用 Application Insights 工具来查询和分析日志。

有关更多信息，请参见以下资源：

- Application Insights 概述
- .NET Core ILogger 日志的 ApplicationInsightsLoggerProvider - 如果想要在没有其他 Application Insights 遥测的情况下实现日志记录提供程序，请从这里开始。
- Application Insights 日志记录适配器。
- 安装、配置和初始化 Application Insights SDK - Microsoft Learn 网站上的交互式教程。

记录提供程序设计注意事项

如果你计划开发自己的 ILoggerProvider 接口实现和相应的 ILogger 自定义实现，请考虑以下几点：

- ILogger.Log 方法是同步方法。
- 不应假定日志状态和对象的生存期。

`ILoggerProvider` 的实现将通过其 ILoggerProvider.CreateLogger 方法创建 `ILogger`。如果实现将日志记录消息以非阻止方式排队，则应首先具体化消息，或将用于具体化日志条目的对象状态序列化。这样做可以避免已释放的对象出现潜在的异常。

有关详细信息，请参阅在 .NET 中实现自定义日志记录提供程序。

第三方日志记录提供程序

下面是适用于 .NET 工作负载的一些第三方日志记录提供程序：

- elmah.io（GitHub 存储库）
- Gelf（GitHub 存储库）
- JSNLog（GitHub 存储库）
- KissLog.net（GitHub 存储库）
- Log4Net（GitHub 存储库）
- NLog（GitHub 存储库）
- NReco.Logging（GitHub 存储库）
- Sentry（GitHub 存储库）
- Serilog（GitHub 存储库）
- Stackdriver（GitHub 存储库）

某些第三方框架可以执行语义日志记录（又称结构化日志记录）。

使用第三方框架类似于使用以下内置提供程序之一：

1. 将 NuGet 包添加到你的项目。
2. 调用日志记录框架提供的 `ILoggerFactory` 或 `ILoggingBuilder` 扩展方法。

有关详细信息，请参阅各提供程序的相关文档。Microsoft 不支持第三方日志记录提供程序。

另请参阅

- .NET 中的日志记录。
- 在 .NET 中实现自定义日志记录提供程序。
- .NET 中的高性能日志记录。

编译时日志记录源生成

2021/11/16 •

> **NOTE**
>
> 本文中的 API 是新增的 API。它们将在 .NET 6 中公开发布，但可能会有变化。
>
> - 使用 .NET 6 预览版 4 时，这些 API 是 Microsoft.Extensions.Logging 包的一部分。
> - 使用 .NET 夜间生成时，这些 API 是 Microsoft.Extensions.Logging.Abstractions 包的一部分。

.NET 6 引入了 `LoggerMessageAttribute` 类型。此属性是 `Microsoft.Extensions.Logging` 命名空间的一部分，使用时，它会以源生成的方式生成高性能的日志记录 API。源生成日志记录支持旨在为新式 .NET 应用程序提供高度可用且高性能的日志记录解决方案。自动生成的源代码依赖于 ILogger 接口和 LoggerMessage.Define 功能。

在 `partial` 日志记录方法上使用 `LoggerMessageAttribute` 时，系统会触发源生成器。触发后，它既可以自动生成其修饰的 `partial` 方法的实现，也可以生成包含正确用法提示的编译时诊断。与现有的日志记录方法相比，编译时日志记录解决方案在运行时通常要快得多。这是因为它最大限度地消除了装箱、临时分配和副本。

基本用法

若要使用 `LoggerMessageAttribute`，使用的类和方法必须为 `partial`。代码生成器在编译时触发，并生成 `partial` 方法的实现。

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{hostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger, string hostName);
}
```

在上面的示例中，日志记录方法为 `static`，日志级别在属性定义中指定。在静态上下文中使用该属性时，需要提供 `ILogger` 实例作为参数。你也可以选择在非静态上下文中使用该属性。请参考以下示例，其中日志记录方法声明为实例方法。在此上下文中，日志记录方法通过访问包含类中的 `ILogger` 字段来获取记录器。

```
public partial class InstanceLoggingExample
{
    private readonly ILogger _logger;

    public InstanceLoggingExample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 0,
        Level = LogLevel.Critical,
        Message = "Could not open socket to `{hostName}`")]
    public partial void CouldNotOpenSocket(string hostName);
}
```

有时，日志级别必须是动态的，而不是静态内置于代码中。为此，可以省略该属性中的日志级别，改为要求将它作为日志记录方法的参数。

```
public static partial class Log
{
    [LoggerMessage(
        EventId = 0,
        Message = "Could not open socket to `{hostName}`")]
    public static partial void CouldNotOpenSocket(
        ILogger logger,
        LogLevel level, /* Dynamic log level as parameter, rather than defined in attribute. */
        string hostName);
}
```

你可以省略日志记录消息，系统将为该消息提供 String.Empty。状态将包含格式化为键值对的参数。

```
using System.Text.Json;
using Microsoft.Extensions.Logging;

ILogger<SampleObject> logger = LoggerFactory.Create(
    builder =>
    builder.AddJsonConsole(
        options =>
        options.JsonWriterOptions = new JsonWriterOptions()
        {
            Indented = true
        }))
    .CreateLogger<SampleObject>();

logger.CustomLogEvent(LogLevel.Information, "Liana", "California");

public static partial class SampleObject
{
    [LoggerMessage(EventId = 23)]
    public static partial void CustomLogEvent(
        this ILogger logger, LogLevel logLevel,
        string name, string state);
}
```

请参考使用 `JsonConsole` 格式化程序时的日志记录输出示例。

```
{
  "EventId": 23,
  "LogLevel": "Information",
  "Category": "ConsoleApp.SampleObject",
  "Message": "",
  "State": {
    "Message": "",
    "name": "Liana",
    "state": "California",
    "{OriginalFormat}": ""
  }
}
```

日志方法约束

在记录方法上使用 `LoggerMessageAttribute` 时，必须遵循一些约束：

- 日志记录方法必须为 `static`、`partial`，并返回 `void`。
- 日志记录方法名称不得以下划线开头。

- 日志记录方法的参数名称不得以下划线开头。
- 日志记录方法不得在嵌套类型中定义。
- 日志记录方法不能是泛型方法。

代码生成模型依赖于使用新式 C# 编译器 9 或更高版本编译的代码。.NET 5 提供了 C# 9.0 编译器。若要升级到新式 C# 编译器，请编辑项目文件以面向 C# 9.0。

```
<PropertyGroup>
  <LangVersion>9.0</LangVersion>
</PropertyGroup>
```

有关详细信息，请参阅 C# 语言版本控制。

日志方法剖析

ILogger.Log 签名接受 LogLevel，还可以接受 Exception，如下所示。

```
public interface ILogger
{
    void Log<TState>(
        Microsoft.Extensions.Logging.LogLevel logLevel,
        Microsoft.Extensions.Logging.EventId eventId,
        TState state,
        System.Exception? exception,
        Func<TState, System.Exception?, string> formatter);
}
```

通常，在源生成器的日志方法签名中，`ILogger`、`LogLevel` 和 `Exception` 的第一个实例将得到特殊处理。后续实例则被当作消息模板的正常参数进行处理：

```
// This is a valid attribute usage
[LoggerMessage(
    EventId = 110, Level = LogLevel.Debug, Message = "M1 {ex3} {ex2}")]
public static partial void ValidLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2,
    Exception ex3);

// This causes a warning
[LoggerMessage(
    EventId = 0, Level = LogLevel.Debug, Message = "M1 {ex} {ex2}")]
public static partial void WarningLogMethod(
    ILogger logger,
    Exception ex,
    Exception ex2);
```

IMPORTANT

发出的警告提供有关 `LoggerMessageAttribute` 的正确用法的详细信息。在上面的示例中，`WarningLogMethod` 将报告 `SYSLIB0025` 的 `DiagnosticSeverity.Warning`。

```
Don't include a template for ex in the logging message since it is implicitly taken care of.
```

不区分大小写的模板名称支持

生成器会在消息模板中的项与日志消息中的参数名称之间进行不区分大小写的比较。这意味着当 `ILogger` 枚举

状态时，消息模板会获取参数，这可以使日志更好使用：

```
public partial class LoggingExample
{
    private readonly ILogger _logger;

    public LoggingExample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 10,
        Level = LogLevel.Information,
        Message = "Welcome to {City} {Province}!")]
    public partial void LogMethodSupportsPascalCasingOfNames(
        string city, string province);

    public void TestLogging()
    {
        LogMethodSupportsPascalCasingOfNames("Vancouver", "BC");
    }
}
```

请参考使用 `JsonConsole` 格式化程序时的日志记录输出示例。

```
{
  "EventId": 13,
  "LogLevel": "Information",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "City": "Vancouver",
    "Province": "BC",
    "{OriginalFormat}": "Welcome to {City} {Province}!"
  }
}
```

不确定的参数顺序

日志方法参数的排序没有约束。开发人员可以将 `ILogger` 定义为最后一个参数，尽管看起来可能有些突兀。

```
[LoggerMessage(
    EventId = 110,
    Level = LogLevel.Debug,
    Message = "M1 {ex3} {ex2}")]
static partial void LogMethod(
    Exception ex,
    Exception ex2,
    Exception ex3,
    ILogger logger);
```

TIP

日志方法上参数的顺序不需要与模板占位符的顺序相对应。而模板中的占位符名称应与参数匹配。请参考以下 `JsonConsole` 输出和错误顺序。

```
{
  "EventId": 110,
  "LogLevel": "Debug",
  "Category": "ConsoleApp.Program",
  "Message": "M1 System.Exception: Third time's the charm. System.Exception: This is the second error.",
  "State": {
    "Message": "M1 System.Exception: Third time's the charm. System.Exception: This is the second
error.",
    "ex2": "System.Exception: This is the second error.",
    "ex3": "System.Exception: Third time's the charm.",
    "{OriginalFormat}": "M1 {ex3} {ex2}"
  }
}
```

其他日志记录示例

以下示例演示如何：

- `LogWithCustomEventName`：通过 `LoggerMessage` 属性检索事件名称。
- `LogWithDynamicLogLevel`：动态设置日志级别，以允许基于配置输入设置日志级别。
- `UsingFormatSpecifier`：使用格式说明符来格式化日志记录参数。

```
public partial class LoggingSample
{
    private readonly ILogger _logger;

    public LoggingSample(ILogger logger)
    {
        _logger = logger;
    }

    [LoggerMessage(
        EventId = 20,
        Level = LogLevel.Critical,
        Message = "Value is {value:E}")]
    public static partial void UsingFormatSpecifier(
        ILogger logger, double value);

    [LoggerMessage(
        EventId = 9,
        Level = LogLevel.Trace,
        Message = "Fixed message",
        EventName = "CustomEventName")]
    public partial void LogWithCustomEventName();

    [LoggerMessage(
        EventId = 10,
        Message = "Welcome to {city} {province}!")]
    public partial void LogWithDynamicLogLevel(
        string city, LogLevel level, string province);

    public void TestLogging()
    {
        LogWithCustomEventName();

        LogWithDynamicLogLevel("Vancouver", LogLevel.Warning, "BC");
        LogWithDynamicLogLevel("Vancouver", LogLevel.Information, "BC");

        UsingFormatSpecifier(logger, 12345.6789);
    }
}
```

请参考使用 `SimpleConsole` 格式化程序时的日志记录输出示例。

```
trce: LoggingExample[9]
      Fixed message
warn: LoggingExample[10]
      Welcome to Vancouver BC!
info: LoggingExample[10]
      Welcome to Vancouver BC!
crit: LoggingExample[20]
      Value is 1.234568E+004
```

请参考使用 `JsonConsole` 格式化程序时的日志记录输出示例。

```
{
  "EventId": 9,
  "LogLevel": "Trace",
  "Category": "LoggingExample",
  "Message": "Fixed message",
  "State": {
    "Message": "Fixed message",
    "{OriginalFormat}": "Fixed message"
  }
}
{
  "EventId": 10,
  "LogLevel": "Warning",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "city": "Vancouver",
    "province": "BC",
    "{OriginalFormat}": "Welcome to {city} {province}!"
  }
}
{
  "EventId": 10,
  "LogLevel": "Information",
  "Category": "LoggingExample",
  "Message": "Welcome to Vancouver BC!",
  "State": {
    "Message": "Welcome to Vancouver BC!",
    "city": "Vancouver",
    "province": "BC",
    "{OriginalFormat}": "Welcome to {city} {province}!"
  }
}
{
  "EventId": 20,
  "LogLevel": "Critical",
  "Category": "LoggingExample",
  "Message": "Value is 1.234568E+004",
  "State": {
    "Message": "Value is 1.234568E+004",
    "value": 12345.6789,
    "{OriginalFormat}": "Value is {value:E}"
  }
}
```


总结

随着 C# 源生成器的出现，编写高性能的日志记录 API 变得更加容易。使用源生成器方法有几个主要好处：

- 允许保留日志记录结构，并启用消息模板所需的确切格式语法。
- 允许为模板占位符提供替代名称，允许使用格式说明符。
- 允许按原样传递所有原始数据，在对其进行处理之前，不需要进行任何复杂的存储（除了创建 `string`）。
- 提供特定于日志记录的诊断，针对重复的事件 ID 发出警告。

与手动使用 LoggerMessage.Define 相比，还有一些好处：

- 语法更短、更简单：使用声明性属性，而不是对样本进行编码。
- 引导式开发人员体验：生成器会发出警告，帮助开发人员做正确的事。
- 支持任意数量的日志记录参数。`LoggerMessage.Define` 最多支持六个。
- 支持动态日志级别。单独使用 `LoggerMessage.Define` 不可能做到这一点。

另请参阅

- .NET 中的日志记录
- .NET 中的高性能日志记录
- 控制台日志格式设置
- NuGet：Microsoft.Extensions.Logging.Abstractions

在 .NET 中实现自定义日志记录提供程序

2021/11/16 •

有很多日志记录提供程序可用于常见日志记录需求。如果某个可用的提供程序不满足你的应用程序需要，则你可能需要实现自定义的 ILoggerProvider。在本文中，你将学习如何实现可用于在控制台中为日志着色的自定义日志记录提供程序。

> **TIP**
>
> 所有日志记录示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的日志记录。

示例自定义记录器配置

此示例会使用以下配置类型为每个日志级别和事件 ID 创建不同的颜色控制台条目：

```
using System;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;

public class ColorConsoleLoggerConfiguration
{
    public int EventId { get; set; }

    public Dictionary<LogLevel, ConsoleColor> LogLevels { get; set; } = new()
    {
        [LogLevel.Information] = ConsoleColor.Green
    };
}
```

前面的代码将默认级别设置为 `Information`，将颜色设置为 `Green`，而且 `EventId` 隐式设置为 `0`。

创建自定义记录器

`ILogger` 实现类别名称通常是日志记录源。例如，创建记录器的类型：

```
using System;
using Microsoft.Extensions.Logging;

public class ColorConsoleLogger : ILogger
{
    private readonly string _name;
    private readonly Func<ColorConsoleLoggerConfiguration> _getCurrentConfig;

    public ColorConsoleLogger(
        string name,
        Func<ColorConsoleLoggerConfiguration> getCurrentConfig) =>
        (_name, _getCurrentConfig) = (name, getCurrentConfig);

    public IDisposable BeginScope<TState>(TState state) => default;

    public bool IsEnabled(LogLevel logLevel) =>
        _getCurrentConfig().LogLevels.ContainsKey(logLevel);

    public void Log<TState>(
        LogLevel logLevel,
        EventId eventId,
        TState state,
        Exception exception,
        Func<TState, Exception, string> formatter)
    {
        if (!IsEnabled(logLevel))
        {
            return;
        }

        ColorConsoleLoggerConfiguration config = _getCurrentConfig();
        if (config.EventId == 0 || config.EventId == eventId.Id)
        {
            ConsoleColor originalColor = Console.ForegroundColor;

            Console.ForegroundColor = config.LogLevels[logLevel];
            Console.WriteLine($"[{eventId.Id,2}: {logLevel,-12}]");

            Console.ForegroundColor = originalColor;
            Console.WriteLine($"     {_name} - {formatter(state, exception)}");
        }
    }
}
```

前面的代码：

- 为每个类别名称创建一个记录器实例。
- 在 `IsEnabled` 中检查 `_getCurrentConfig().LogLevels.ContainsKey(logLevel)`，因此每个 `logLevel` 都有一个唯一的记录器。还应为所有更高的日志级别启用记录器：

```
public bool IsEnabled(LogLevel logLevel) =>
    _getCurrentConfig().LogLevels.ContainsKey(logLevel);
```

记录器使用 `name` 和 `Func<ColorConsoleLoggerConfiguration>` 进行实例化，这将返回当前配置，这会处理对通过 IOptionsMonitor<TOptions>.OnChange 回调监视的配置值的更新。

> **IMPORTANT**
>
> ILogger.Log 实现检查是否设置了 `config.EventId` 值。未设置 `config.EventId` 或与确切的 `logEntry.EventId` 匹配时，记录器会以颜色记录。

自定义记录器提供程序

`ILoggerProvider` 对象负责创建记录器实例。不需要为每个类别创建一个记录器实例，但这对于某些记录器（例如 NLog 或 log4net）来说是需要的。借助此策略可以为每个类别选择不同的日志记录输出目标，如以下示例中所示：

```
using System;
using System.Collections.Concurrent;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Options;

public sealed class ColorConsoleLoggerProvider : ILoggerProvider
{
    private readonly IDisposable _onChangeToken;
    private ColorConsoleLoggerConfiguration _currentConfig;
    private readonly ConcurrentDictionary<string, ColorConsoleLogger> _loggers = new();

    public ColorConsoleLoggerProvider(
        IOptionsMonitor<ColorConsoleLoggerConfiguration> config)
    {
        _currentConfig = config.CurrentValue;
        _onChangeToken = config.OnChange(updatedConfig => _currentConfig = updatedConfig);
    }

    public ILogger CreateLogger(string categoryName) =>
        _loggers.GetOrAdd(categoryName, name => new ColorConsoleLogger(name, GetCurrentConfig));

    private ColorConsoleLoggerConfiguration GetCurrentConfig() => _currentConfig;

    public void Dispose()
    {
        _loggers.Clear();
        _onChangeToken.Dispose();
    }
}
```

在前面的代码中，CreateLogger 会为每个类别名称创建一个 `ColorConsoleLogger` 实例并将其存储在 `ConcurrentDictionary<TKey,TValue>` 中。此外，还需要 IOptionsMonitor<TOptions> 接口才能更新对基础 `ColorConsoleLoggerConfiguration` 对象的更改。

自定义记录器的使用和注册

根据约定，在应用程序启动例程中注册服务以进行依赖项注入。注册在 `Program` 类中进行，还可能委托给 `Startup` 类。本示例将直接从 Program.cs 进行注册。

若要添加自定义日志记录提供程序和相应的记录器，请从 HostingHostBuilderExtensions.ConfigureLogging(IHostBuilder, Action<ILoggingBuilder>) 使用 ILoggingBuilder 添加 ILoggerProvider：

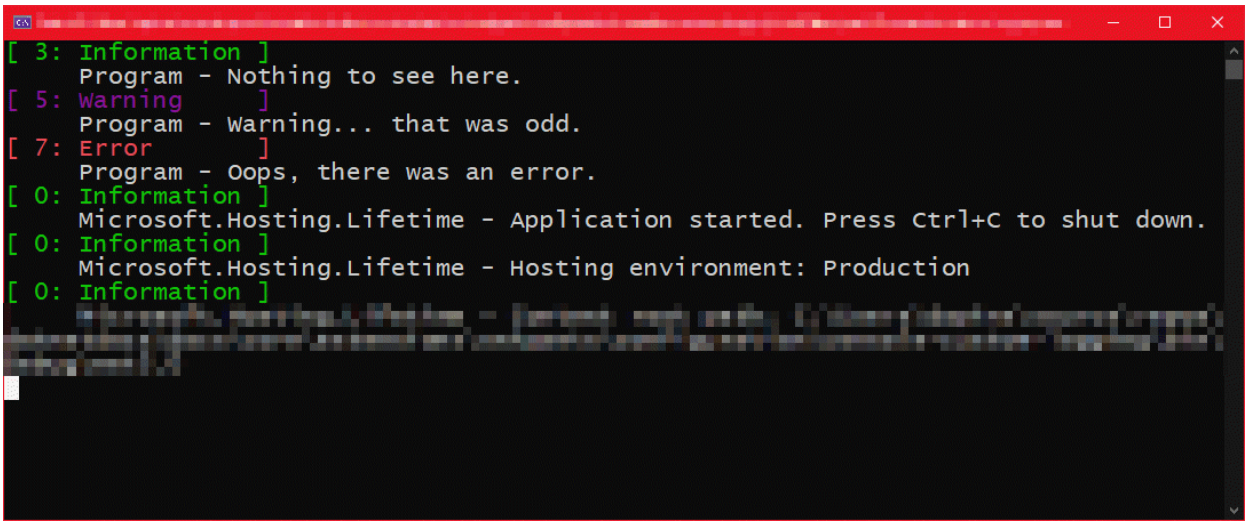
```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

class Program
{
    static async Task Main(string[] args)
    {
        using IHost host = CreateHostBuilder(args).Build();

        var logger = host.Services.GetRequiredService<ILogger<Program>>();

        logger.LogDebug(1, "Does this line get hit?");    // Not logged
        logger.LogInformation(3, "Nothing to see here."); // Logs in ConsoleColor.Green
        logger.LogWarning(5, "Warning... that was odd."); // Logs in ConsoleColor.DarkMagenta
        logger.LogError(7, "Oops, there was an error.");  // Logs in ConsoleColor.Red

        await host.RunAsync();
    }

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(builder =>
                builder.ClearProviders()
                    .AddColorConsoleLogger(configuration =>
                    {
                        configuration.LogLevels.Add(
                            LogLevel.Warning, ConsoleColor.DarkMagenta);
                        configuration.LogLevels.Add(
                            LogLevel.Error, ConsoleColor.Red);
                    }));
}
```

`ILoggingBuilder` 创建一个或多个 `ILogger` 实例。框架使用 `ILogger` 实例记录信息。

按照约定，`ILoggingBuilder` 上的扩展方法用于注册自定义提供程序：

```
using System;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.DependencyInjection.Extensions;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Configuration;

public static class ColorConsoleLoggerExtensions
{
    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder)
    {
        builder.AddConfiguration();

        builder.Services.TryAddEnumerable(
            ServiceDescriptor.Singleton<ILoggerProvider, ColorConsoleLoggerProvider>());

        LoggerProviderOptions.RegisterProviderOptions
            <ColorConsoleLoggerConfiguration, ColorConsoleLoggerProvider>(builder.Services);

        return builder;
    }

    public static ILoggingBuilder AddColorConsoleLogger(
        this ILoggingBuilder builder,
        Action<ColorConsoleLoggerConfiguration> configure)
    {
        builder.AddColorConsoleLogger();
        builder.Services.Configure(configure);

        return builder;
    }
}
```

运行此简单应用程序将把颜色输出呈现到控制台窗口，如下图所示：



另请参阅

- .NET 中的日志记录
- .NET 中的日志记录提供程序
- .NET 中的依赖关系注入
- .NET 中的高性能日志记录

.NET 中的高性能日志记录

2021/11/16 •

LoggerMessage 类公开了用于创建可缓存委托的功能，该功能比记录器扩展方法（例如 LogInformation 和 LogDebug）需要的对象分配和计算开销要少。对于高性能日志记录方案，请使用 LoggerMessage 模式。

与记录器扩展方法相比，LoggerMessage 具有以下性能优势：

- 记录器扩展方法需要将值类型（例如 `int`）“装箱”（转换）到 `object` 中。LoggerMessage 模式使用带强类型参数的静态 Action 字段和扩展方法来避免装箱。
- 记录器扩展方法每次写入日志消息时必须分析消息模板（命名的格式字符串）。如果已定义消息，那么 LoggerMessage 只需分析一次模板即可。

示例应用展示了具有优先级队列处理辅助角色服务的 LoggerMessage 功能。应用按优先级顺序处理工作项。发生这些操作时，通过 LoggerMessage 模式生成日志消息。

> **TIP**
> 所有日志记录示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的日志记录。

定义记录器消息

使用 Define(LogLevel, EventId, String) 创建一个用于记录消息的 Action 委托。Define 重载允许向命名的格式字符串（模板）传递最多六个类型参数。

提供给 Define 方法的字符串是一个模板，而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称，建议使用帕斯卡拼写法。例如：`{Item}`、`{DateTime}`。

每条日志消息都是一个 Action，保存在由 LoggerMessage.Define 创建的静态字段中。例如，示例应用创建一个字段来描述处理工作项的日志消息：

```
private static readonly Action<ILogger, Exception> _failedToProcessWorkItem;
```

对于 Action，指定：

- 日志级别。
- 具有静态扩展方法名称的唯一事件标识符 (EventId)。
- 消息模板（命名的格式字符串）。

由于对工作项处理取消排队，辅助角色服务应用会进行下列设置：

- 将日志级别设置为 LogLevel.Critical。
- 将事件 ID 设置为具有 `FailedToProcessWorkItem` 方法名称的 `13`。
- 将消息模板（命名的格式字符串）设置为字符串。

```
_failedToProcessWorkItem = LoggerMessage.Define(
    LogLevel.Critical,
    new EventId(13, nameof(FailedToProcessWorkItem)),
    "Epic failure processing item!");
```

结构化日志记录存储可以使用事件名称(当它获得事件 ID 时)来丰富日志记录。例如，Serilog 使用该事件名称。

通过强类型扩展方法调用 Action。`PriorityItemProcessed` 方法会在每次处理工作项时记录一条消息。而在(如果)发生异常时则调用 `FailedToProcessWorkItem`：

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope = _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem = _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _logger.FailedToProcessWorkItem(ex);
        }

        await Task.Delay(1000, stoppingToken);
    }
}
```

检查应用的控制台输出：

```
crit: WorkerServiceOptions.Example.Worker[13]
      Epic failure processing item!
      System.Exception: Failed to verify communications.
         at WorkerServiceOptions.Example.Worker.ExecuteAsync(CancellationToken stoppingToken) in
         ..\Worker.cs:line 27
```

要将参数传递给日志消息，创建静态字段时最多定义六种类型。在通过定义 Action 字段的 `WorkItem` 类型来处理项时，示例应用会记录工作项的详细信息：

```
private static readonly Action<ILogger, WorkItem, Exception> _processingPriorityItem;
```

委托的日志消息模板从提供的类型接收其占位符值。示例应用定义一个委托，用于在 item 参数是 `WorkItem` 的位置添加一个工作项：

```
_processingPriorityItem = LoggerMessage.Define<WorkItem>(
    LogLevel.Information,
    new EventId(1, nameof(PriorityItemProcessed)),
    "Processing priority item: {Item}");
```

用于记录正在处理工作项的静态扩展方法 `PriorityItemProcessed` 接收工作项参数值并将其传递给 Action 委托：

```
public static void PriorityItemProcessed(
    this ILogger logger, WorkItem workItem) =>
    _processingPriorityItem(logger, workItem, default!);
```

在辅助角色服务的 `ExecuteAsync` 方法中，调用 `PriorityItemProcessed` 以记录消息：

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope = _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem = _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _logger.FailedToProcessWorkItem(ex);
        }

        await Task.Delay(1000, stoppingToken);
    }
}
```

检查应用的控制台输出：

```
info: WorkerServiceOptions.Example.Worker[1]
      Processing priority item: Priority-Extreme (50db062a-9732-4418-936d-110549ad79e4): 'Verify
communications'
```

指定记录器消息作用域

DefineScope(string) 方法创建一个用于定义日志作用域的 Func<TResult> 委托。DefineScope 重载允许向命名的格式字符串（模板）传递最多三个类型参数。

Define 方法也一样，提供给 DefineScope 方法的字符串是一个模板，而不是内插字符串。占位符按照指定类型的顺序填充。模板中的占位符名称在各个模板中应当具备描述性和一致性。它们在结构化的日志数据中充当属性名称。对于占位符名称，建议使用帕斯卡拼写法。例如：`{Item}`、`{DateTime}`。

使用 DefineScope 方法定义一个日志作用域，以应用到一系列日志消息中。在 appsettings.json 的控制台记录器部分启用 `IncludeScopes`：

```
{
    "Logging": {
        "Console": {
            "IncludeScopes": true
        },
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        }
    }
}
```

要创建日志作用域，请添加一个字段来保存该作用域的 Func<TResult> 委托。示例应用创建一个名为 `_processingWorkScope` (Internal/LoggerExtensions.cs) 的字段：

```
private static Func<ILogger, DateTime, IDisposable> _processingWorkScope;
```

使用 DefineScope 来创建委托。调用委托时最多可以指定三种类型作为模板参数使用。示例应用使用包含处理开始的日期时间的消息模板：

```
_processingWorkScope =
    LoggerMessage.DefineScope<DateTime>(
        "Processing work, started at: {DateTime}");
```

为日志消息提供一种静态扩展方法。包含已命名属性的任何类型参数（这些参数出现在消息模板中）。示例应用接受自定义时间戳的 `DateTime` 以记录和返回 `_processingWorkScope`：

```
public static IDisposable ProcessingWorkScope(
    this ILogger logger, DateTime time) =>
    _processingWorkScope(logger, time);
```

该作用域将日志记录扩展调用包装在 using 块中：

```
protected override async Task ExecuteAsync(
    CancellationToken stoppingToken)
{
    using IDisposable? scope = _logger.ProcessingWorkScope(DateTime.Now);
    while (!stoppingToken.IsCancellationRequested)
    {
        WorkItem? nextItem = _priorityQueue.ProcessNextHighestPriority();
        try
        {
            if (nextItem is not null)
            {
                _logger.PriorityItemProcessed(nextItem);
            }
        }
        catch (Exception ex)
        {
            _logger.FailedToProcessWorkItem(ex);
        }

        await Task.Delay(1000, stoppingToken);
    }
}
```

检查应用控制台输出中的日志消息。以下结果显示日志消息的优先级排序，其中包括日志作用域消息：

```
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Extreme (f5090ede-a337-4041-b914-f6bc0db5ae64): 'Verify
communications'
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: ..\worker-service-options
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-High (496d440f-2007-4391-b179-09d75ab52373): 'Validate collection'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Medium (dea9e3f4-d7df-46d2-b7cd-5e0232eb98a5): 'Propagate
selections'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Medium (089d7f0d-da72-4b55-92fe-57b147838056): 'Enter pooling
[contention]'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Low (6e68c4be-089f-4450-9080-1ea63fcbb686): 'Health check network'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Deferred (6f324134-6bb6-455f-81d4-553ab307c421): 'Ping weather
service'
info: WorkerServiceOptions.Example.Worker[1]
      => Processing work, started at: 09/25/2020 14:30:45
      Processing priority item: Priority-Deferred (37bf736c-7a26-4a2a-9e56-e89bcf3b8f35): 'Set process
state'
```

日志级防护优化

在调用相应的 `Log*` 方法之前，可通过使用 ILogger.IsEnabled(LogLevel) 检查 LogLevel 来进行额外的性能优化。未为给定的 `LogLevel` 配置日志记录时，以下语句为 true：

- 未调用 ILogger.Log。
- 避免分配表示参数的 `object[]`。
- 避免值类型装箱。

参考信息：

- .NET 运行时中的微基准
- 日志级别检查的背景和动机

另请参阅

- .NET 中的日志记录

控制台日志格式设置

2021/11/16 •

在 .NET 5 中，已向 `Microsoft.Extensions.Logging.Console` 命名空间中的控制台日志添加了对自定义格式设置的支持。有三种预定义的格式设置选项可供选择：`Simple`、`Systemd` 和 `Json`。

> **IMPORTANT**
>
> 以前，ConsoleLoggerFormat 枚举允许选择所需的日志格式，可以是人工可读取的（`Default`），也可以是单行（也称为 `Systemd`）。不过，这些都是不能进行自定义的，现已弃用。

本文介绍了控制台日志格式化程序。示例源代码演示了如何执行以下操作：

- 注册新的格式化程序
- 选择要使用的已注册格式化程序
  - 通过代码或配置
- 实现自定义格式化程序
  - 通过 IOptionsMonitor<TOptions> 更新配置
  - 启用自定义颜色格式设置

> **TIP**
>
> 所有日志记录示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的日志记录。

注册格式化程序

`Console` 日志记录提供程序有几个预定义的格式化程序，并公开了可供你编写自己的自定义格式化程序的功能。要注册任何可用的格式化程序，请使用相应的 `Add{Type}Console` 扩展方法：

[illegible]	[illegible]
ConsoleFormatterNames.Json	ConsoleLoggerExtensions.AddJsonConsole
ConsoleFormatterNames.Simple	ConsoleLoggerExtensions.AddSimpleConsole
ConsoleFormatterNames.Systemd	ConsoleLoggerExtensions.AddSystemdConsole

简单

要使用 `Simple` 控制台格式化程序，请将其注册到 `AddSimpleConsole`：

```
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Simple
{
    class Program
    {
        static void Main()
        {
            using ILoggerFactory loggerFactory =
                LoggerFactory.Create(builder =>
                    builder.AddSimpleConsole(options =>
                    {
                        options.IncludeScopes = true;
                        options.SingleLine = true;
                        options.TimestampFormat = "hh:mm:ss ";
                    }));

            ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
            using (logger.BeginScope("[scope is enabled]"))
            {
                logger.LogInformation("Hello World!");
                logger.LogInformation("Logs contain timestamp and log level.");
                logger.LogInformation("Each log message is fit in a single line.");
            }
        }
    }
}
```

在前面的示例源代码中，已经注册了 ConsoleFormatterNames.Simple 格式化程序。它不仅为日志提供了在每条日志消息中包装信息（如时间和日志级别）的功能，而且还允许 ANSI 颜色嵌入和消息缩进。

Systemd

ConsoleFormatterNames.Systemd 控制台记录器：

- 使用“Syslog”日志级别格式和严重性
- 不为消息设置颜色格式
- 始终将消息记录在单行中

这对于容器来说通常很有用，因为容器经常使用 `Systemd` 控制台日志记录。在 .NET 5 中，`Simple` 控制台记录器还可以实现以单行方式进行记录的紧凑版本，并且还允许禁用颜色，如前面的示例所示。

```
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Systemd
{
    class Program
    {
        static void Main()
        {
            using ILoggerFactory loggerFactory =
                LoggerFactory.Create(builder =>
                    builder.AddSystemdConsole(options =>
                    {
                        options.IncludeScopes = true;
                        options.TimestampFormat = "hh:mm:ss ";
                    }));

            ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
            using (logger.BeginScope("[scope is enabled]"))
            {
                logger.LogInformation("Hello World!");
                logger.LogInformation("Logs contain timestamp and log level.");
                logger.LogInformation("Systemd console logs never provide color options.");
                logger.LogInformation("Systemd console logs always appear in a single line.");
            }
        }
    }
}
```

Json

要以 JSON 格式编写日志，需要使用 `Json` 控制台格式化程序。示例源代码展示了 ASP.NET Core 应用如何注册格式化程序。使用 `webapp` 模板，用 dotnet new 命令创建一个新的 ASP.NET Core 应用：

```
dotnet new webapp -o Console.ExampleFormatters.Json
```

运行此应用时，使用模板代码会获得以下默认日志格式：

```
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: https://localhost:5001
```

默认情况下，默认配置中会选择 `Simple` 控制台日志格式化程序。可以通过在 Program.cs 中调用 `AddJsonConsole` 来更改此设置：

```
using System.Text.Json;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Json
{
    class Program
    {
        static Task Main(string[] args) =>
            CreateHostBuilder(args).Build().RunAsync();

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(builder => builder.UseStartup<Startup>())
                .ConfigureLogging(builder =>
                    builder.AddJsonConsole(options =>
                    {
                        options.IncludeScopes = false;
                        options.TimestampFormat = "hh:mm:ss ";
                        options.JsonWriterOptions = new JsonWriterOptions
                        {
                            Indented = true
                        };
                    }));
    }
}
```

再次运行此应用，进行上述更改后，现在日志消息的格式为 JSON：

```
{
    "Timestamp": "09:08:33 ",
    "EventId": 0,
    "LogLevel": "Information",
    "Category": "Microsoft.Hosting.Lifetime",
    "Message": "Now listening on: https://localhost:5001",
    "State": {
        "Message": "Now listening on: https://localhost:5001",
        "address": "https://localhost:5001",
        "{OriginalFormat}": "Now listening on: {address}"
    }
}
```

TIP

默认情况下，`Json` 控制台格式化程序将每条消息记录在单行中。为了在配置格式化程序时使其更具可读性，请将 JsonWriterOptions.Indented 设置为 `true`。

通过配置设置格式化程序

前面的示例展示了如何以编程方式注册格式化程序。另外，也可以通过配置来完成。考虑到之前的 Web 应用示例源代码，如果更新 appsettings.json 文件，而不是调用 Program.cs 文件中的 `ConfigureLogging`，可以得到相同的结果。更新后的 `appsettings.json` 文件将对格式化程序进行如下配置：

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
            "FormatterName": "json",
            "FormatterOptions": {
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

需要设置的两个键值为 `"FormatterName"` 和 `"FormatterOptions"`。如果已经注册了值设置为 `"FormatterName"` 的格式化程序，则会选择此格式化程序，并且只要在 `"FormatterOptions"` 节点中将其属性作为键提供，就可以配置属性。预定义的格式化程序名称保留在 ConsoleFormatterNames 下：

- ConsoleFormatterNames.Json
- ConsoleFormatterNames.Simple
- ConsoleFormatterNames.Systemd

实现自定义格式化程序

要实现自定义格式化程序，需要执行以下操作：

- 创建一个 ConsoleFormatter 的子类，这代表你的自定义格式化程序
- 注册以下自定义格式化程序
  - ConsoleLoggerExtensions.AddConsole
  - ConsoleLoggerExtensions.AddConsoleFormatter<TFormatter,TOptions>(ILoggingBuilder, Action<TOptions>)

创建一个扩展方法来为你进行处理：

```
using Microsoft.Extensions.Logging;
using System;

namespace Console.ExampleFormatters.Custom
{
    public static class ConsoleLoggerExtensions
    {
        public static ILoggingBuilder AddCustomFormatter(
            this ILoggingBuilder builder,
            Action<CustomOptions> configure) =>
            builder.AddConsole(options => options.FormatterName = "customName")
                .AddConsoleFormatter<CustomFormatter, CustomOptions>(configure);
    }
}
```

`CustomOptions` 的定义如下：

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom
{
    public class CustomOptions : ConsoleFormatterOptions
    {
        public string CustomPrefix { get; set; }
    }
}
```

在前面的代码中，选项是 ConsoleFormatterOptions 的子类。

`AddConsoleFormatter` API 会执行以下操作：

- 注册 `ConsoleFormatter` 的子类
- 处理配置：
  - 基于选项模式和 IOptionsMonitor 接口，使用更改令牌同步更新

```
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.Custom
{
    class Program
    {
        static void Main()
        {
            using ILoggerFactory loggerFactory =
                LoggerFactory.Create(builder =>
                    builder.AddCustomFormatter(options =>
                        options.CustomPrefix = " ~~~~~ "));

            ILogger<Program> logger = loggerFactory.CreateLogger<Program>();
            using (logger.BeginScope("TODO: Add logic to enable scopes"))
            {
                logger.LogInformation("Hello World!");
                logger.LogInformation("TODO: Add logic to enable timestamp and log level info.");
            }
        }
    }
}
```

定义 `ConsoleFormatter` 的 `CustomerFormatter` 子类：

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;
using System;
using System.IO;

namespace Console.ExampleFormatters.Custom
{
    public sealed class CustomFormatter : ConsoleFormatter, IDisposable
    {
        private readonly IDisposable _optionsReloadToken;
        private CustomOptions _formatterOptions;

        public CustomFormatter(IOptionsMonitor<CustomOptions> options)
            // Case insensitive
            : base("customName") =>
            (_optionsReloadToken, _formatterOptions) =
                (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

        private void ReloadLoggerOptions(CustomOptions options) =>
            _formatterOptions = options;

        public override void Write<TState>(
            in LogEntry<TState> logEntry,
            IExternalScopeProvider scopeProvider,
            TextWriter textWriter)
        {
            string message =
                logEntry.Formatter(
                    logEntry.State, logEntry.Exception);

            if (message == null)
            {
                return;
            }

            CustomLogicGoesHere(textWriter);
            textWriter.WriteLine(message);
        }

        private void CustomLogicGoesHere(TextWriter textWriter)
        {
            textWriter.Write(_formatterOptions.CustomPrefix);
        }

        public void Dispose() => _optionsReloadToken?.Dispose();
    }
}
```

前面的 `CustomFormatter.Write<TState>` API 指明了每个日志消息包装的文本类型。一个标准的 `ConsoleFormatter` 应至少能包装日志的范围、时间戳和严重性级别。此外，你还可以对日志消息中的 ANSI 颜色进行编码，并提供所需的缩进。`CustomFormatter.Write<TState>` 的实现缺少这些功能。

有关进一步自定义格式设置的灵感，请参阅 `Microsoft.Extensions.Logging.Console` 命名空间中的现有实现：

- SimpleConsoleFormatter。
- SystemdConsoleFormatter
- JsonConsoleFormatter

自定义配置选项

若要进一步自定义日志记录可扩展性，可从任何配置提供程序配置派生的 ConsoleFormatterOptions 类。例如，可以使用 JSON 配置提供程序来定义自定义选项。首先定义 ConsoleFormatterOptions 子类。

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.CustomWithConfig
{
    public class CustomWrappingConsoleFormatterOptions : ConsoleFormatterOptions
    {
        public string CustomPrefix { get; set; }

        public string CustomSuffix { get; set; }
    }
}
```

前述控制台格式化程序选项类定义了两个自定义属性，表示前缀和后缀。接下来，定义将配置控制台格式化程序选项的 appsettings.json 文件。

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Information",
            "Microsoft": "Warning",
            "Microsoft.Hosting.Lifetime": "Information"
        },
        "Console": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            },
            "FormatterName": "CustomTimePrefixingFormatter",
            "FormatterOptions": {
                "CustomPrefix": "|-<[",
                "CustomSuffix": "]>-|",
                "SingleLine": true,
                "IncludeScopes": true,
                "TimestampFormat": "HH:mm:ss.ffff ",
                "UseUtcTimestamp": true,
                "JsonWriterOptions": {
                    "Indented": true
                }
            }
        }
    },
    "AllowedHosts": "*"
}
```

在前述 JSON 配置文件中：

- `"Logging"` 节点定义 `"Console"`。
- `"Console"` 节点指定 `"CustomTimePrefixingFormatter"` 的 `"FormatterName"`，它映射到自定义格式化程序。
- `"FormatterOptions"` 节点定义 `"CustomPrefix"` 和 `"CustomSuffix"`，以及其他一些派生的选项。

> **TIP**
>
> `$.Logging.Console.FormatterOptions` JSON 路径已保留，使用 AddConsoleFormatter 扩展方法添加时，它将映射到自定义 ConsoleFormatterOptions。除了可用属性之外，这还能定义自定义属性。

假设为以下 `CustomDatePrefixingFormatter`：

```
using System;
using System.IO;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;

namespace Console.ExampleFormatters.CustomWithConfig
{
    public sealed class CustomTimePrefixingFormatter : ConsoleFormatter, IDisposable
    {
        private readonly IDisposable _optionsReloadToken;
        private CustomWrappingConsoleFormatterOptions _formatterOptions;

        public CustomTimePrefixingFormatter(IOptionsMonitor<CustomWrappingConsoleFormatterOptions> options)
            // Case insensitive
            : base(nameof(CustomTimePrefixingFormatter)) =>
            (_optionsReloadToken, _formatterOptions) =
                (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

        private void ReloadLoggerOptions(CustomWrappingConsoleFormatterOptions options) =>
            _formatterOptions = options;

        public override void Write<TState>(
            in LogEntry<TState> logEntry,
            IExternalScopeProvider scopeProvider,
            TextWriter textWriter)
        {
            string message =
                logEntry.Formatter(
                    logEntry.State, logEntry.Exception);

            if (message == null)
            {
                return;
            }

            WritePrefix(textWriter);
            textWriter.Write(message);
            WriteSuffix(textWriter);
        }

        private void WritePrefix(TextWriter textWriter)
        {
            DateTime now = _formatterOptions.UseUtcTimestamp
                ? DateTime.UtcNow
                : DateTime.Now;

            textWriter.Write($"{_formatterOptions.CustomPrefix}
{now.ToString(_formatterOptions.TimestampFormat)}");
        }

        private void WriteSuffix(TextWriter textWriter) => textWriter.WriteLine($"
{_formatterOptions.CustomSuffix}");

        public void Dispose() => _optionsReloadToken?.Dispose();
    }
}
```

在前述格式化程序实现中：

- 监视 `CustomWrappingConsoleFormatterOptions` 是否有所更改，并相应地进行更新。
- 写入的消息将用配置的前缀和后缀进行包装。
- 使用配置的 ConsoleFormatterOptions.UseUtcTimestamp 和 ConsoleFormatterOptions.TimestampFormat 值在前缀后面、消息之前添加时间戳。

若要在自定义格式化程序实现中使用自定义配置选项，请在调用 ConfigureLogging(IHostBuilder, Action<HostBuilderContext,ILoggingBuilder>) 时添加。

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace Console.ExampleFormatters.CustomWithConfig
{
    class Program
    {
        static void Main(string[] args)
        {
            using IHost host = Host.CreateDefaultBuilder(args)
                .ConfigureLogging(builder =>
                {
                    builder.AddConsole()
                        .AddConsoleFormatter
                            <CustomTimePrefixingFormatter, CustomWrappingConsoleFormatterOptions>();
                })
                .Build();

            ILoggerFactory loggerFactory = host.Services.GetRequiredService<ILoggerFactory>();
            ILogger<Program> logger = loggerFactory.CreateLogger<Program>();

            using (logger.BeginScope("Logging scope"))
            {
                logger.LogInformation("Hello World!");
                logger.LogInformation("The .NET developer community happily welcomes you.");
            }
        }
    }
}
```

以下控制台输出类似于你使用此 `CustomTimePrefixingFormatter` 时可能想要看到的内容。

```
|-<[ 15:03:15.6179 Hello World! ]>-|
|-<[ 15:03:15.6347 The .NET developer community happily welcomes you. ]>-|
```

实现自定义颜色格式设置

为了在自定义日志记录格式化程序中正确地启用颜色功能，可以扩展 SimpleConsoleFormatterOptions，因为它具有 SimpleConsoleFormatterOptions.ColorBehavior 属性，这个属性对于在日志中启用颜色非常有用。

创建一个从 `SimpleConsoleFormatterOptions` 派生的 `CustomColorOptions`：

```
using Microsoft.Extensions.Logging.Console;

namespace Console.ExampleFormatters.Custom
{
    public class CustomColorOptions : SimpleConsoleFormatterOptions
    {
        public string CustomPrefix { get; set; }
    }
}
```

接下来，在 `TextWriterExtensions` 类中编写一些扩展方法，以便在经过格式设置的日志消息中方便地嵌入 ANSI 编码的颜色：

```
using System;
```

```
using System.IO;

namespace Console.ExampleFormatters.Custom
{
    public static class TextWriterExtensions
    {
        const string DefaultForegroundColor = "\x1B[39m\x1B[22m";
        const string DefaultBackgroundColor = "\x1B[49m";

        public static void WriteWithColor(
            this TextWriter textWriter,
            string message,
            ConsoleColor? background,
            ConsoleColor? foreground)
        {
            // Order:
            //   1. background color
            //   2. foreground color
            //   3. message
            //   4. reset foreground color
            //   5. reset background color

            var backgroundColor = background.HasValue ? GetBackgroundColorEscapeCode(background.Value) :
null;
            var foregroundColor = foreground.HasValue ? GetForegroundColorEscapeCode(foreground.Value) :
null;

            if (backgroundColor != null)
            {
                textWriter.Write(backgroundColor);
            }
            if (foregroundColor != null)
            {
                textWriter.Write(foregroundColor);
            }

            textWriter.WriteLine(message);

            if (foregroundColor != null)
            {
                textWriter.Write(DefaultForegroundColor);
            }
            if (backgroundColor != null)
            {
                textWriter.Write(DefaultBackgroundColor);
            }
        }

        static string GetForegroundColorEscapeCode(ConsoleColor color) =>
            color switch
            {
                ConsoleColor.Black =>       "\x1B[30m",
                ConsoleColor.DarkRed =>     "\x1B[31m",
                ConsoleColor.DarkGreen =>   "\x1B[32m",
                ConsoleColor.DarkYellow =>  "\x1B[33m",
                ConsoleColor.DarkBlue =>    "\x1B[34m",
                ConsoleColor.DarkMagenta => "\x1B[35m",
                ConsoleColor.DarkCyan =>    "\x1B[36m",
                ConsoleColor.Gray =>        "\x1B[37m",
                ConsoleColor.Red =>         "\x1B[1m\x1B[31m",
                ConsoleColor.Green =>       "\x1B[1m\x1B[32m",
                ConsoleColor.Yellow =>      "\x1B[1m\x1B[33m",
                ConsoleColor.Blue =>        "\x1B[1m\x1B[34m",
                ConsoleColor.Magenta =>     "\x1B[1m\x1B[35m",
                ConsoleColor.Cyan =>        "\x1B[1m\x1B[36m",
                ConsoleColor.White =>       "\x1B[1m\x1B[37m",

                _ => DefaultForegroundColor
            };
```

```
        static string GetBackgroundColorEscapeCode(ConsoleColor color) =>
            color switch
            {
                ConsoleColor.Black =>       "\x1B[40m",
                ConsoleColor.DarkRed =>     "\x1B[41m",
                ConsoleColor.DarkGreen =>   "\x1B[42m",
                ConsoleColor.DarkYellow =>  "\x1B[43m",
                ConsoleColor.DarkBlue =>    "\x1B[44m",
                ConsoleColor.DarkMagenta => "\x1B[45m",
                ConsoleColor.DarkCyan =>    "\x1B[46m",
                ConsoleColor.Gray =>        "\x1B[47m",

                _ => DefaultBackgroundColor
            };
    }
}
```

可以定义一个处理应用自定义颜色的自定义颜色格式化程序，如下所示：

```
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.Logging.Abstractions;
using Microsoft.Extensions.Logging.Console;
using Microsoft.Extensions.Options;
using System;
using System.IO;

namespace Console.ExampleFormatters.Custom
{
    public sealed class CustomColorFormatter : ConsoleFormatter, IDisposable
    {
        private readonly IDisposable _optionsReloadToken;
        private CustomColorOptions _formatterOptions;

        private bool ConsoleColorFormattingEnabled =>
            _formatterOptions.ColorBehavior == LoggerColorBehavior.Enabled ||
            _formatterOptions.ColorBehavior == LoggerColorBehavior.Default &&
            System.Console.IsOutputRedirected == false;

        public CustomColorFormatter(IOptionsMonitor<CustomColorOptions> options)
            // Case insensitive
            : base("customName") =>
            (_optionsReloadToken, _formatterOptions) =
                (options.OnChange(ReloadLoggerOptions), options.CurrentValue);

        private void ReloadLoggerOptions(CustomColorOptions options) =>
            _formatterOptions = options;

        public override void Write<TState>(
            in LogEntry<TState> logEntry,
            IExternalScopeProvider scopeProvider,
            TextWriter textWriter)
        {
            if (logEntry.Exception is null)
            {
                return;
            }

            string message =
                logEntry.Formatter(
                    logEntry.State, logEntry.Exception);

            if (message == null)
            {
                return;
            }
```

```
            CustomLogicGoesHere(textWriter);
            textWriter.WriteLine(message);
        }

        private void CustomLogicGoesHere(TextWriter textWriter)
        {
            if (ConsoleColorFormattingEnabled)
            {
                textWriter.WriteWithColor(
                    _formatterOptions.CustomPrefix,
                    ConsoleColor.Black,
                    ConsoleColor.Green);
            }
            else
            {
                textWriter.Write(_formatterOptions.CustomPrefix);
            }
        }

        public void Dispose() => _optionsReloadToken?.Dispose();
    }
}
```

当你运行应用程序时，日志会在 `FormatterOptions.ColorBehavior` 为 `Enabled` 时用绿色显示 `CustomPrefix` 消息。

> **NOTE**
>
> 当 LoggerColorBehavior 为 `Disabled` 时，日志消息不会解释日志消息中嵌入的 ANSI 颜色代码。相反，它们会输出原始消息。例如，请考虑如下事项：
>
> ```
> logger.LogInformation("Random log \x1B[42mwith green background\x1B[49m message");
> ```
>
> 这会输出逐字字符串，且不会着色。
>
> ```
> Random log \x1B[42mwith green background\x1B[49m message
> ```

另请参阅

- .NET 中的日志记录
- 在 .NET 中实现自定义日志记录提供程序
- .NET 中的高性能日志记录

.NET 通用主机

2021/11/16 •

辅助角色服务模板会创建一个 .NET 通用主机 HostBuilder。通用主机可用于其他类型的 .NET 应用程序，如控制台应用。

主机是封装应用资源和生存期功能的对象，例如：

- 依赖关系注入 (DI)
- Logging
- Configuration
- 应用关闭
- `IHostedService` 实现

当主机启动时，它将对在托管服务的服务容器集合中注册的 IHostedService 的每个实现调用 IHostedService.StartAsync。在辅助角色服务应用中，包含 BackgroundService 实例的所有 `IHostedService` 实现都调用其 BackgroundService.ExecuteAsync 方法。

一个对象中包含所有应用的相互依赖资源的主要原因是生存期管理：控制应用启动和正常关闭。这是通过 Microsoft.Extensions.Hosting NuGet 包实现的。

设置主机

主机通常由 `Program` 类中的代码配置、生成和运行。`Main` 方法：

- 调用 CreateDefaultBuilder() 方法以创建和配置生成器对象。
- 调用 Build() 以创建 IHost 实例。
- 对主机对象调用 Run 或 RunAsync 方法。

.NET 辅助角色服务模板会生成以下代码来创建通用主机：

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            });
}
```

默认生成器设置

CreateDefaultBuilder 方法：

- 将内容根路径设置为由 GetCurrentDirectory() 返回的路径。
- 通过以下对象加载主机配置：
  - 前缀为 `DOTNET_` 的环境变量。

    - 命令行参数。
- 通过以下对象加载应用配置：
    - appsettings.json。
    - appsettings.{Environment}.json。
    - 密钥管理器 当应用在 `Development` 环境中运行时。
    - 环境变量。
    - 命令行参数。
- 添加以下日志记录提供程序：
    - 控制台
    - 调试
    - EventSource
    - EventLog（仅当在 Windows 上运行时）
- 当环境为 `Development` 时，启用范围验证和依赖关系验证。

`ConfigureServices` 方法公开了向 Microsoft.Extensions.DependencyInjection.IServiceCollection 实例添加服务的功能。以后，可以通过依赖关系注入获取这些服务。

框架提供的服务

自动注册以下服务：

- IHostApplicationLifetime
- IHostLifetime
- IHostEnvironment

IHostApplicationLifetime

将 IHostApplicationLifetime 服务注入任何类以处理启动后和正常关闭任务。接口上的三个属性是用于注册应用启动和应用停止事件处理程序方法的取消令牌。该接口还包括 StopApplication() 方法。

以下示例是注册 `IHostApplicationLifetime` 事件的 `IHostedService` 实现：

```
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace AppLifetime.Example
{
    public class ExampleHostedService : IHostedService
    {
        private readonly ILogger _logger;

        public ExampleHostedService(
            ILogger<ExampleHostedService> logger,
            IHostApplicationLifetime appLifetime)
        {
            _logger = logger;

            appLifetime.ApplicationStarted.Register(OnStarted);
            appLifetime.ApplicationStopping.Register(OnStopping);
            appLifetime.ApplicationStopped.Register(OnStopped);
        }

        public Task StartAsync(CancellationToken cancellationToken)
        {
            _logger.LogInformation("1. StartAsync has been called.");

            return Task.CompletedTask;
        }

        public Task StopAsync(CancellationToken cancellationToken)
        {
            _logger.LogInformation("4. StopAsync has been called.");

            return Task.CompletedTask;
        }

        private void OnStarted()
        {
            _logger.LogInformation("2. OnStarted has been called.");
        }

        private void OnStopping()
        {
            _logger.LogInformation("3. OnStopping has been called.");
        }

        private void OnStopped()
        {
            _logger.LogInformation("5. OnStopped has been called.");
        }
    }
}
```

可以修改辅助角色服务模板以添加 `ExampleHostedService` 实现：

```
using System.Threading.Tasks;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace AppLifetime.Example
{
    class Program
    {
        static Task Main(string[] args) =>
            CreateHostBuilder(args).Build().RunAsync();

        static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((_, services) =>
                    services.AddHostedService<ExampleHostedService>());
    }
}
```

应用程序会编写以下示例输出：

```
// Sample output:
//     info: ExampleHostedService[0]
//           1. StartAsync has been called.
//     info: ExampleHostedService[0]
//           2. OnStarted has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application started.Press Ctrl+C to shut down.
//     info: Microsoft.Hosting.Lifetime[0]
//           Hosting environment: Production
//     info: Microsoft.Hosting.Lifetime[0]
//           Content root path: ..\app-lifetime\bin\Debug\net5.0
//     info: ExampleHostedService[0]
//           3. OnStopping has been called.
//     info: Microsoft.Hosting.Lifetime[0]
//           Application is shutting down...
//     info: ExampleHostedService[0]
//           4. StopAsync has been called.
//     info: ExampleHostedService[0]
//           5. OnStopped has been called.
```

IHostLifetime

IHostLifetime 实现控制主机何时启动和何时停止。使用了已注册的最后一个实现。`Microsoft.Extensions.Hosting.Internal.ConsoleLifetime` 是默认的 `IHostLifetime` 实现。有关关闭的生存期机制的详细信息，请参阅主机关闭。

IHostEnvironment

将 IHostEnvironment 服务注册到一个类，获取关于以下设置的信息：

- IHostEnvironment.ApplicationName
- IHostEnvironment.ContentRootFileProvider
- IHostEnvironment.ContentRootPath
- IHostEnvironment.EnvironmentName

主机配置

主机配置用于配置 IHostEnvironment 实现的属性。

主机配置在 ConfigureAppConfiguration 方法的 HostBuilderContext.Configuration 中可用。调用 `ConfigureAppConfiguration` 方法时，`HostBuilderContext` 和 `IConfigurationBuilder` 将传递到 `configureDelegate` 中。`configureDelegate` 被定义为 `Action<HostBuilderContext, IConfigurationBuilder>`。主机生成器上下文公开 `.Configuration` 属性，该属性是 `IConfiguration` 的一个实例。它表示从主机生成的配置，而 `IConfigurationBuilder` 是用于配置应用的生成器对象。

> **TIP**
> 在调用 `ConfigureAppConfiguration` 后，`HostBuilderContext.Configuration` 被替换为应用配置。

若要添加主机配置，请对 `IHostBuilder` 调用 ConfigureHostConfiguration。可多次调用 `ConfigureHostConfiguration`，并得到累计结果。主机使用上一次在一个给定键上设置值的选项。

以下示例创建主机配置：

```
using System.IO;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;

class Program
{
    static async Task Main(string[] args)
    {
        using IHost host = CreateHostBuilder(args).Build();

        // Application code should start here.

        await host.RunAsync();
    }

    static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureHostConfiguration(configHost =>
            {
                configHost.SetBasePath(Directory.GetCurrentDirectory());
                configHost.AddJsonFile("hostsettings.json", optional: true);
                configHost.AddEnvironmentVariables(prefix: "PREFIX_");
                configHost.AddCommandLine(args);
            });
}
```

应用配置

通过对 `IHostBuilder` 调用 ConfigureAppConfiguration 创建应用配置。可多次调用 `ConfigureAppConfiguration`，并得到累计结果。应用使用上一次在一个给定键上设置值的选项。

可以在 HostBuilderContext.Configuration 中获取由 `ConfigureAppConfiguration` 创建的配置以用于后续操作，并将其作为 DI 的服务。主机配置也会添加到应用配置。

有关详细信息，请参阅 .NET 中的配置。

主机关闭

在下列情况中，可以停止托管服务进程：

- 有人未调用 Run 或 HostingAbstractionsHostExtensions.WaitForShutdown，且应用在 `Main` 完成时正常退出。

- 应用出现故障。
- 使用 SIGKILL（或 CTRL+Z）强制关闭应用。

以上场景都不是由主机代码直接处理的。进程所有者需要以处理应用程序的相同方式来处理它们。在其他几种情况下也可以停止托管服务进程：

- 使用 `ConsoleLifetime` 时，它将侦听以下信号，并尝试正常停止主机。
  - SIGINT（或 CTRL+C）。
  - SIGQUIT（或 Windows 上的 CTRL+BREAK，Unix 上的 CTRL+\）。
  - SIGTERM（由其他应用发送，如 `docker stop`）。
- 应用调用 Environment.Exit。

以上场景都是由内置的主机逻辑（具体来说，是 `ConsoleLifetime` 类）处理的。`ConsoleLifetime` 尝试处理“关闭”信号 SIGINT、SIGQUIT 和 SIGTERM，以支持正常退出应用程序。

在 .NET 6 之前，.NET 代码无法正常地处理 SIGTERM。为了绕开此限制，`ConsoleLifetime` 需要订阅 System.AppDomain.ProcessExit。如果引发了 `ProcessExit`，`ConsoleLifetime` 会发出信号通知主机停止并阻止 `ProcessExit` 线程，等待主机停止。这样，应用程序中的清理代码便可运行，例如 IHost.StopAsync 和 `Main` 方法中 HostingAbstractionsHostExtensions.Run 之后的代码。

这会导致其他问题，因为 SIGTERM 不是引发 `ProcessExit` 的唯一方法。它还由应用程序中调用 `Environment.Exit` 的代码引发。`Environment.Exit` 不是在 `Microsoft.Extensions.Hosting` 应用模型中正常关闭进程的方法。它将引发 `ProcessExit` 事件，并退出该进程。`Main` 方法的末尾不会执行。后台和前台线程会终止，并且 `finally` 块不会执行。

由于 `ConsoleLifetime` 阻止 `ProcessExit` 等待主机关闭，此行为导致 `Environment.Exit` 死锁，也阻止等待 `ProcessExit` 返回。此外，由于 SIGTERM 处理尝试正常关闭进程，因此 `ConsoleLifetime` 会将 ExitCode 设置为 `0`，这会强制改写传递给 `Environment.Exit` 的用户退出代码。

在 .NET 6 中，支持 POSIX 信号并可对其进行处理。这样，`ConsoleLifetime` 可正常处理 SIGTERM，并在调用 `Environment.Exit` 时不再涉及其中。

> **TIP**
> 对于 .NET 6+，`ConsoleLifetime` 不再具有处理场景 `Environment.Exit` 的逻辑。调用 `Environment.Exit` 并需要执行清理逻辑的应用可以自行订阅 `ProcessExit`。在这种情况下，主机代码将不再尝试正常停止主机。

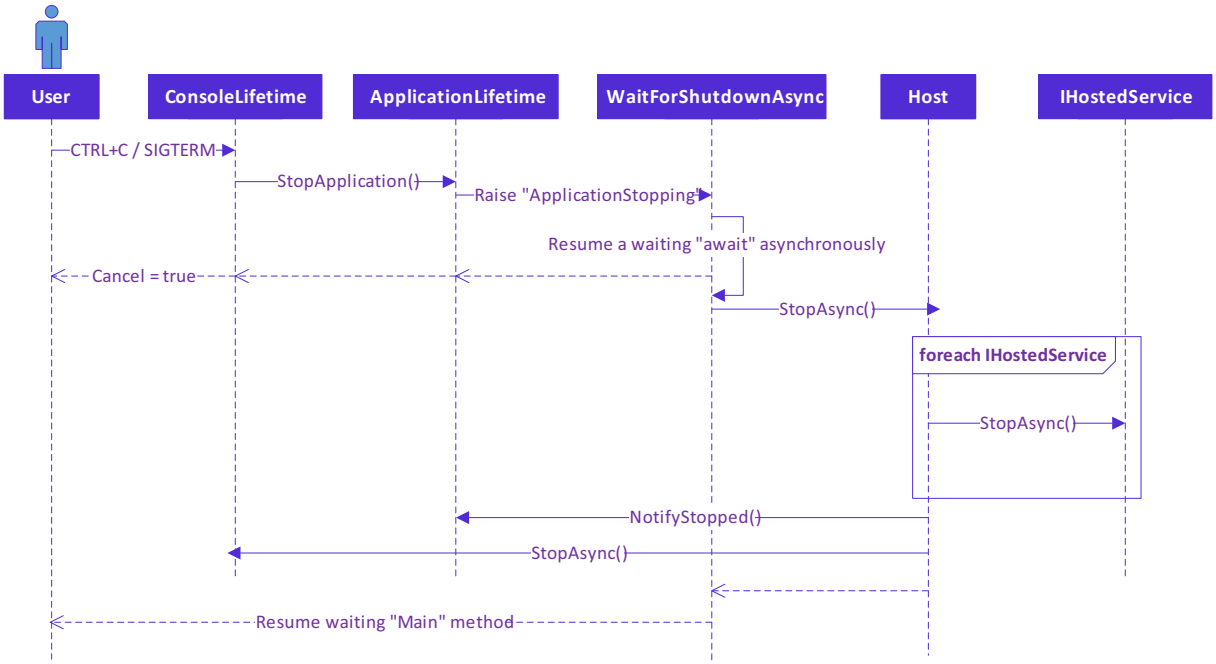
如果应用程序使用主机代码，并且你希望正常停止主机，可调用 IHostApplicationLifetime.StopApplication 而不是 `Environment.Exit`。

主机代码关闭过程

下面的顺序图演示了如何在主机代码中内部处理信号。大多数用户不需要了解此过程。但对于需要深入了解的开发人员而言，这有助于入门。

启动主机后，用户调用 `Run` 或 `WaitForShutdown` 时，处理程序会注册 IApplicationLifetime.ApplicationStopping。执行在 `WaitForShutdown` 中暂停，等待引发 `ApplicationStopping` 事件。因此，`Main` 方法不会立即返回，并且应用会一直运行，直到 `Run` 或 `WaitForShutdown` 返回。

信号发送到进程时，它将启动以下序列：



1. 控制从 `ConsoleLifetime` 流向 `ApplicationLifetime`，引发 `ApplicationStopping` 事件。这会指示 `WaitForShutdownAsync` 解除阻止 `Main` 执行代码。同时，由于 POSIX 信号已经过处理，POSIX 信号处理程序返回 `Cancel = true`。
2. `Main` 执行代码将再次开始执行，并指示主机 `StopAsync()`，进而停止所有托管服务，并引发任何其他已停止的事件。
3. 最后，`WaitForShutdown` 退出，使任何应用程序清理代码可执行且 `Main` 方法可正常退出。

另请参阅

- .NET 中的依赖关系注入
- .NET 中的日志记录
- .NET 中的配置
- .NET 中的辅助角色服务
- ASP.NET Core Web 主机
- 应在 github.com/dotnet/extensions 存储库中创建通用主机 bug

使用 .NET 的 HTTP

2021/11/16 •

在本文中，你将了解如何将 `IHttpClientFactory` 和 `HttpClient` 类型与各种 .NET 基础知识结合使用，例如依赖项注入 (DI)、日志记录和配置。HttpClient 类型是在 2012 年发布的 .NET Framework 4.5 中引入的。换句话说，它已经存在一段时间了。`HttpClient` 用于从由 Uri 标识的网络资源发出 HTTP 请求和处理 HTTP 响应。HTTP 协议占所有 Internet 流量的绝大部分。

根据推动最佳做法的新式应用程序开发原则，IHttpClientFactory 充当工厂抽象，可以使用自定义配置创建 `HttpClient` 实例。.NET Core 2.1 中引入了 IHttpClientFactory。常见的基于 HTTP 的 .NET 工作负载可以轻松利用可复原和瞬态故障处理第三方中间件。

探索 `IHttpClientFactory` 类型

本文中的所有示例源代码都依赖于 `Microsoft.Extensions.Http` NuGet 包。此外，Internet Chuck Norris 数据库免费 API 用于为“怪僻的”笑话发出 HTTP GET 请求。

调用任何 AddHttpClient 扩展方法时，将 `IHttpClientFactory` 和相关服务添加到 IServiceCollection。`IHttpClientFactory` 类型具有以下优点：

- 将 `HttpClient` 类公开为 DI 就绪类型。
- 提供一个中心位置，用于命名和配置逻辑 `HttpClient` 实例。
- 通过 `HttpClient` 中的委托处理程序来编码出站中间件的概念。
- 提供基于 Polly 的中间件的扩展方法，以利用 `HttpClient` 中的委托处理程序。
- 管理基础 HttpClientHandler 实例的池和生存期。自动管理可避免手动管理 `HttpClient` 生存期时出现的常见域名系统 (DNS) 问题。
- （通过 ILogger）添加可配置的记录体验，以处理工厂创建的客户端发送的所有请求。

消耗模式

在应用中可以通过以下多种方式使用 `IHttpClientFactory`：

- 基本用法
- 命名客户端
- 类型化客户端
- 生成的客户端

最佳方法取决于应用要求。

基本用法

若要注册 `IHttpClientFactory`，请调用 `AddHttpClient`：

```
using BasicHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHttpClient();
        services.AddTransient<JokeService>();
    })
    .Build();
```

使用服务可能需要 `IHttpClientFactory` 作为带有 DI 的构造函数参数。以下代码使用 `IHttpClientFactory` 来创建 `HttpClient` 实例：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Shared;

namespace BasicHttp.Example
{
    public class JokeService
    {
        private readonly IHttpClientFactory _httpClientFactory = null!;
        private readonly ILogger<JokeService> _logger = null!;

        public JokeService(
            IHttpClientFactory httpClientFactory,
            ILogger<JokeService> logger) =>
            (_httpClientFactory, _logger) = (httpClientFactory, logger);

        public async Task<string> GetRandomJokeAsync()
        {
            // Create the client
            HttpClient client = _httpClientFactory.CreateClient();

            try
            {
                // Make HTTP GET request
                // Parse JSON response deserialize into ChuckNorrisJoke type
                ChuckNorrisJoke? result = await client.GetFromJsonAsync<ChuckNorrisJoke>(
                    "https://api.icndb.com/jokes/random?limitTo=[nerdy]",
                    DefaultJsonSerialization.Options);

                if (result?.Value?.Joke is not null)
                {
                    return result.Value.Joke;
                }
            }
            catch (Exception ex)
            {
                _logger.LogError("Error getting something fun to say: {Error}", ex);
            }

            return "Oops, something has gone wrong - that's not funny at all!";
        }
    }
}
```

像前面的示例一样，使用 `IHttpClientFactory` 是重构现有应用的好方法。这不会影响 `HttpClient` 的使用方式。

在现有应用中创建 `HttpClient` 实例的位置，使用对 CreateClient 的调用替换这些匹配项。

命名客户端

在以下情况下，命名客户端是一个不错的选择：

- 应用需要 `HttpClient` 的许多不同用法。
- 许多 `HttpClient` 实例具有不同的配置。

可以在 `ConfigureServices` 中注册时指定命名 `HttpClient` 的配置：

```
using NamedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((context, services) =>
    {
        string httpClientName = context.Configuration["JokeHttpClientName"];
        services.AddHttpClient(
            httpClientName,
            client =>
            {
                // Set the base address of the named client.
                client.BaseAddress = new Uri("https://api.icndb.com/");

                // Add a user-agent default request header.
                client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
            });
        services.AddTransient<JokeService>();
    })
    .Build();
```

在上述代码中，客户端配置如下：

- 从 `"JokeHttpClientName"` 下的配置中提取的名称。
- 基址为 `https://api.icndb.com/`。
- 一个 `"User-Agent"` 标头。

可以使用配置来指定 HTTP 客户端名称，这有助于避免在添加和创建时误命名客户端。在本例中，appsettings.json 文件用于配置 HTTP 客户端名称：

```
{
    "JokeHttpClientName": "ChuckNorrisJokeApi"
}
```

可以轻松扩展此配置，并存储有关你希望 HTTP 客户端如何工作的更多详细信息。有关详细信息，请参阅 .NET 中的配置。

创建客户端

每次调用 CreateClient 时：

- 创建 `HttpClient` 的新实例。
- 调用配置操作。

要创建命名客户端，请将其名称传递到 `CreateClient` 中：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Shared;

namespace NamedHttp.Example
{
    public class JokeService
    {
        private readonly IHttpClientFactory _httpClientFactory = null!;
        private readonly IConfiguration _configuration = null!;
        private readonly ILogger<JokeService> _logger = null!;

        public JokeService(
            IHttpClientFactory httpClientFactory,
            IConfiguration configuration,
            ILogger<JokeService> logger) =>
            (_httpClientFactory, _configuration, _logger) =
                (httpClientFactory, configuration, logger);

        public async Task<string> GetRandomJokeAsync()
        {
            // Create the client
            string httpClientName = _configuration["JokeHttpClientName"];
            HttpClient client = _httpClientFactory.CreateClient(httpClientName);

            try
            {
                // Make HTTP GET request
                // Parse JSON response deserialize into ChuckNorrisJoke type
                ChuckNorrisJoke? result = await client.GetFromJsonAsync<ChuckNorrisJoke>(
                    "jokes/random?limitTo=[nerdy]",
                    DefaultJsonSerialization.Options);

                if (result?.Value?.Joke is not null)
                {
                    return result.Value.Joke;
                }
            }
            catch (Exception ex)
            {
                _logger.LogError("Error getting something fun to say: {Error}", ex);
            }

            return "Oops, something has gone wrong - that's not funny at all!";
        }
    }
}
```

在上述代码中，HTTP 请求不需要指定主机名。代码可以仅传递路径，因为采用了为客户端配置的基址。

类型化客户端

类型化客户端：

- 提供与命名客户端一样的功能，不需要将字符串用作密钥。
- 在使用客户端时提供 IntelliSense 和编译器帮助。
- 提供单个位置来配置特定 `HttpClient` 并与其进行交互。例如，可以使用单个类型化客户端：
  - 对于单个后端终结点。
  - 封装处理终结点的所有逻辑。
- 使用 DI 且可以被注入到应用中需要的位置。

类型化客户端在构造函数中接受 `HttpClient` 参数：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;
using Shared;

namespace TypedHttp.Example
{
    public sealed class JokeService
    {
        private readonly HttpClient _httpClient = null!;
        private readonly ILogger<JokeService> _logger = null!;

        public JokeService(
            HttpClient httpClient,
            ILogger<JokeService> logger) =>
            (_httpClient, _logger) = (httpClient, logger);

        public async Task<string> GetRandomJokeAsync()
        {
            try
            {
                // Make HTTP GET request
                // Parse JSON response deserialize into ChuckNorrisJoke type
                ChuckNorrisJoke? result = await _httpClient.GetFromJsonAsync<ChuckNorrisJoke>(
                    "https://api.icndb.com/jokes/random?limitTo=[nerdy]",
                    DefaultJsonSerialization.Options);

                if (result?.Value?.Joke is not null)
                {
                    return result.Value.Joke;
                }
            }
            catch (Exception ex)
            {
                _logger.LogError("Error getting something fun to say: {Error}", ex);
            }

            return "Oops, something has gone wrong - that's not funny at all!";
        }
    }
}
```

在上述代码中：

- 该配置是在将类型化的客户端添加到服务集合时设置的。
- `HttpClient` 被分配为类范围变量（字段），并与公开的 API 一起使用。

可以创建特定于 API 的方法来公开 `HttpClient` 功能。例如，创建 `GetRandomJokeAsync` 方法来封装代码以检索随机笑话。

以下代码调用 `ConfigureServices` 中的 AddHttpClient 来注册类型化客户端类：

```
using TypedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHttpClient<JokeService>(
            client =>
            {
                // Set the base address of the named client.
                client.BaseAddress = new Uri("https://api.icndb.com/");

                // Add a user-agent default request header.
                client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
            });
        services.AddTransient<JokeService>();
    })
    .Build();
```

使用 DI 将类型客户端注册为暂时客户端。在上述代码中，`AddHttpClient` 将 `JokeService` 注册为暂时性服务。此注册使用工厂方法执行以下操作：

1. 创建 `HttpClient` 的实例。
2. 创建 `JokeService` 的实例，将 `HttpClient` 的实例传入其构造函数。

> **TIP**
> 调用 `AddHttpClient<TClient>` 不会将 `TClient` 服务添加到 `IServiceCollection`。仍需要使用 `Add{ServiceLifetime}` 显式添加它。

生成的客户端

`IHttpClientFactory` 可结合第三方库（例如 Refit）使用。Refit 是.NET 的 REST 库。它允许声明性 REST API 定义，将接口方法映射到终结点。`RestService` 动态生成该接口的实现，使用 `HttpClient` 进行外部 HTTP 调用。

请考虑以下 `record` 类型：

```
namespace Shared
{
    public record IdentifiableJokeValue(
        int Id, string Joke);
}
```

```
namespace Shared
{
    public record ChuckNorrisJoke(
        string Type,
        IdentifiableJokeValue Value);
}
```

以下示例依赖于 `Refit.HttpClientFactory` NuGet 包，并且是一个简单的接口：

```
using System.Threading.Tasks;
using Refit;
using Shared;

namespace GeneratedHttp.Example
{
    public interface IJokeService
    {
        [Get("/jokes/random?limitTo=[nerdy]")]
        Task<ChuckNorrisJoke> GetRandomJokeAsync();
    }
}
```

前面的 C# 接口：

- 定义一个名为 `GetRandomJokeAsync` 的方法，该方法返回一个 `Task<ChuckNorrisJoke>` 实例。
- 使用外部 API 的路径和查询字符串声明 `Refit.GetAttribute` 属性。

可以添加类型化客户端，使用 Refit 生成实现：

```
using System;
using GeneratedHttp.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using Refit;
using Shared;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddRefitClient<IJokeService>()
            .ConfigureHttpClient(client =>
            {
                // Set the base address of the named client.
                client.BaseAddress = new Uri("https://api.icndb.com/");

                // Add a user-agent default request header.
                client.DefaultRequestHeaders.UserAgent.ParseAdd("dotnet-docs");
            });
    })
    .Build();
```

可以在必要时使用定义的接口，以及由 DI 和 Refit 提供的实现。

发出 POST、PUT 和 DELETE 请求

在前面的示例中，所有 HTTP 请求均使用 GET HTTP 谓词。`HttpClient` 还支持其他 HTTP 谓词，其中包括：

- POST
- PUT
- 删除
- PATCH

有关受支持的 HTTP 谓词的完整列表，请参阅 HttpMethod。

下面的示例演示如何发出 HTTP POST 请求：

```
public async Task CreateItemAsync(Item item)
{
    using StringContent json = new(
        JsonSerializer.Serialize(item, DefaultJsonSerialization.Options),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await _httpClient.PostAsync("/api/items", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

在前面的代码中，`CreateItemAsync` 方法：

- 使用 `System.Text.Json` 将 `Item` 参数序列化为 JSON。这将使用 JsonSerializerOptions 的实例来配置序列化过程。
- 创建 StringContent 的实例，以打包序列化的 JSON 以便在 HTTP 请求的正文中发送。
- 调用 PostAsync 将 JSON 内容发送到指定的 URL。这是添加到 HttpClient.BaseAddress 的相对 URL。
- 如果响应状态代码不指示成功，则调用 EnsureSuccessStatusCode 引发异常。

`HttpClient` 还支持其他类型的内容。例如，MultipartContent 和 StreamContent。有关受支持的内容的完整列表，请参阅 HttpContent。

下面的示例演示了一个 HTTP PUT 请求：

```
public async Task UpdateItemAsync(Item item)
{
    using StringContent json = new(
        JsonSerializer.Serialize(item, DefaultJsonSerialization.Options),
        Encoding.UTF8,
        MediaTypeNames.Application.Json);

    using HttpResponseMessage httpResponse =
        await _httpClient.PutAsync($"/api/items/{item.Id}", json);

    httpResponse.EnsureSuccessStatusCode();
}
```

前面的代码与 POST 示例非常相似。`UpdateItemAsync` 方法调用 PutAsync 而不是 `PostAsync`。

下面的示例演示了一个 HTTP DELETE 请求：

```
public async Task DeleteItemAsync(Guid id)
{
    using HttpResponseMessage httpResponse =
        await _httpClient.DeleteAsync($"/api/items/{id}");

    httpResponse.EnsureSuccessStatusCode();
}
```

在前面的代码中，`DeleteItemAsync` 方法调用 DeleteAsync。由于 HTTP DELETE 请求通常不包含正文，因此 `DeleteAsync` 方法不提供接受 `HttpContent` 实例的重载。

要详细了解如何将不同的 HTTP 谓词用于 `HttpClient`，请参阅 HttpClient。

`HttpClient` 生存期管理

每次对 `IHttpClientFactory` 调用 `CreateClient` 都会返回一个新 `HttpClient` 实例。每个客户端创建一个 HttpClientHandler 实例。工厂管理 `HttpClientHandler` 实例的生存期。

`IHttpClientFactory` 将工厂创建的 `HttpClientHandler` 实例汇集到池中，以减少资源消耗。新建 `HttpClient` 实例时，可能会重用池中的 `HttpClientHandler` 实例（如果生存期尚未到期的话）。

由于每个处理程序通常管理自己的基础 HTTP 连接，因此需要池化处理程序。创建超出必要数量的处理程序可能会导致连接延迟。部分处理程序还保持连接无期限地打开，这样可以防止处理程序对 DNS 更改作出反应。

处理程序的默认生存期为两分钟。要替代默认值，请在 `IServiceCollection` 上为每个客户端调用 SetHandlerLifetime：

```
services.AddHttpClient("Named.Client")
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

> **IMPORTANT**
> 通常可以将 `HttpClient` 实例视为不需要处置的对象。处置既取消传出请求，又保证在调用 Dispose 后无法使用给定的 `HttpClient` 实例。`IHttpClientFactory` 跟踪和处置 `HttpClient` 实例使用的资源。

保持各个 `HttpClient` 实例长时间处于活动状态是在 `IHttpClientFactory` 推出前使用的常见模式。迁移到 `IHttpClientFactory` 后，就无需再使用此模式。

配置 `HttpMessageHandler`

控制客户端使用的内部 `HttpMessageHandler` 的配置是有必要的。

在添加命名客户端或类型化客户端时，会返回 IHttpClientBuilder。ConfigurePrimaryHttpMessageHandler 扩展方法可以用于在 `IServiceCollection` 上定义委托。委托用于创建和配置客户端使用的主要 `HttpMessageHandler`：

```
services.AddHttpClient("Named.Client")
    .ConfigurePrimaryHttpMessageHandler(() =>
    {
        return new HttpClientHandler
        {
            AllowAutoRedirect = false,
            UseDefaultCredentials = true
        };
    });
```

其他配置

有几个额外的配置选项可用于控制 `IHttpClientHandler`：

"	"
AddHttpMessageHandler	为已命名的 `HttpClient` 添加附加消息处理程序。
AddTypedClient	配置 `TClient` 与已命名的 `HttpClient`（与 `IHttpClientBuilder` 关联）之间的绑定。
ConfigureHttpClient	添加用于配置已命名的 `HttpClient` 的委托。

“	“
ConfigureHttpMessageHandlerBuilder	添加一个委托，该委托将用于使用 HttpMessageHandlerBuilder 为已命名的 `HttpClient` 配置消息处理程序。
ConfigurePrimaryHttpMessageHandler	从已命名的 `HttpClient` 的依赖关系注入容器中配置主要 `HttpMessageHandler`。
RedactLoggedHeaders	设置其值应在记录之前进行修正的 HTTP 标头名称的集合。
SetHandlerLifetime	设置可重复使用 `HttpMessageHandler` 实例的时长。每个已命名的客户端都可自行配置处理程序生存期值。

另请参阅

- .NET 中的依赖关系注入
- .NET 中的日志记录
- .NET 中的配置
- IHttpClientFactory
- HttpClient
- 实现使用指数退避算法的 HTTP 重试

将 HTTP/3 与 HttpClient 结合使用

2021/11/16 •

HTTP/3 是 HTTP 的第三个即将发布的主要版本。HTTP/3 使用与 HTTP/1.1 和 HTTP/2 相同的语义：相同的请求方法、状态代码和消息字段适用于所有版本。差异在于基础传输。HTTP/1.1 和 HTTP/2 都将 TCP 用作其传输协议。HTTP/3 使用的是与 HTTP/3 同时开发的一种新传输技术，称为 QUIC。

与 HTTP/1.1 和 HTTP/2 相比，HTTP/3 和 QUIC 具有很多优势：

- 第一个请求的响应时间更短。QUIC 和 HTTP/3 在客户端和服务器之间以较少的往返次数协商连接。第一个请求会更快地到达服务器。
- 改进了发生连接数据包丢失时的体验。HTTP/2 通过一个 TCP 连接多路复用多个请求。如果在连接时发生数据包丢失，会影响所有请求。这个问题称为“队头阻塞”。由于 QUIC 提供本机多路复用，因此丢失的数据包只会影响已丢失数据的请求。
- 支持在网络之间转换。此功能对于移动设备非常有用，因为在移动设备更改位置时，在 WIFI 和移动电话网络之间切换是很常见的。目前，在切换网络时，HTTP/1.1 和 HTTP/2 连接会失败并提示错误。应用或 Web 浏览器必须重试任何失败的 HTTP 请求。HTTP/3 让应用或 Web 浏览器在网络发生更改时可以无缝地继续。HttpClient 和 Kestrel 不支持 .NET 6 中的网络转换。它可能在未来版本中可用。

> **IMPORTANT**
>
> HTTP/3 在 .NET 6 中作为预览功能提供，因为 HTTP/3 规范还没有最终确定，并且在 .NET 6 中，HTTP/3 可能存在行为或性能问题。
>
> 有关预览功能的详细信息，请参阅预览功能规范。
>
> 配置为利用 HTTP/3 的应用应设计为也支持 HTTP/1.1 和 HTTP/2。如果在 HTTP/3 中标识了问题，建议禁用 HTTP/3，直到问题在 .NET 的未来版本中得到解决。

HttpClient 设置

HTTP/3 支持目前以预览版提供，需要通过一个配置标志启用该功能，可以使用以下代码在项目中设置该标志：

```
<ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Net.SocketsHttpHandler.Http3Support" Value="true" />
</ItemGroup>
```

或使用 `AppContext.SetSwitch`。

可以通过将 `HttpRequestMessage.Version` 设置为 3.0 来配置 HTTP 版本。但是，由于并非所有路由器、防火墙和代理都能正确地支持 HTTP/3，我们建议将 HTTP/3 与 HTTP/1.1 和 HTTP/2 一起配置。在 HttpClient 中，可以通过以下操作来实现此目的：

- 将 `HttpRequestMessage.Version` 指定为 1.1。
- 将 `HttpRequestMessage.VersionPolicy` 指定为 `HttpVersionPolicy.RequestVersionOrHigher`。

需要为 HTTP/3 设置配置标志的原因是为了防止将来在使用版本策略 `RequestVersionOrHigher` 时应用发生中断。调用当前使用 HTTP/1.1 和 HTTP/2 的服务器时，如果服务器随后升级到 HTTP/3，则客户端会尝试使用 HTTP/3 并且有可能不兼容，因为标准并不是最终版本，因此在 .NET 6 发布后可能会更改。

平台依赖项

HTTP/3 将 QUIC 用作其传输协议。HTTP/3 的 .NET 实现使用 MsQuic 来提供 QUIC 功能。MsQuic 包含在 Windows 的特定版本中，并作为 Linux 的一个库。如果运行 HttpClient 的平台不满足 HTTP/3 的所有要求，则会将它禁用。

Windows

- Windows 11 内部版本 22000（版本 21H2）或更高版本。
- TLS 1.3 或更高版本的连接。

Linux

在 Linux 上，libmsquic 是通过 Microsoft 官方 Linux 包存储库 packages.microsoft.com 发布的。为了使用它，必须手动添加。请参阅 Microsoft 产品的 Linux 软件存储库。配置包源后，可以通过发行版的包管理器安装它，例如，对于 Ubuntu：

```
sudo apt install libmsquic
```

macOS

HTTP/3 目前在 macOS 上不受支持，但可能会在未来版本中提供。

使用 HttpClient

在项目文件中包括以下内容，以启用 HTTP/3 和 HttpClient：

```
<ItemGroup>
    <RuntimeHostConfigurationOption Include="System.Net.SocketsHttpHandler.Http3Support" Value="true" />
</ItemGroup>
```

以下代码示例使用顶级语句，演示如何在请求中指定 HTTP3：

```
// See https://aka.ms/new-console-template for more information
using System.Net;

var client = new HttpClient()
{
    DefaultRequestVersion =  HttpVersion.Version30,
    DefaultVersionPolicy = HttpVersionPolicy.RequestVersionExact
};

Console.WriteLine("--- Localhost:5001 ---");
HttpResponseMessage resp = await client.GetAsync("https://localhost:5001/");
string body = await resp.Content.ReadAsStringAsync();
Console.WriteLine(
    $"status: {resp.StatusCode}, version: {resp.Version}, " +
    $"body: {body.Substring(0, Math.Min(100, body.Length))}");
```

HTTP/3 服务器

.NET 6 中具有 Kestrel 服务器的 ASP.NET 支持 HTTP/3。有关详细信息，请参阅对 ASP.NET Core Kestrel Web 服务器使用 HTTP/3。

公测服务器

Cloudflare 托管了一个可用于在 https://cloudflare-quic.com/ 上测试客户端的 HTTP/3 站点

另请参阅

- HttpClient
- Kestrel 中的 HTTP/3 支持

.NET 中的文件通配

2021/11/16 •

在本文中，了解如何将文件通配与 `Microsoft.Extensions.FileSystemGlobbing` NuGet 包一起使用。glob 是一种术语，用于定义基于通配符匹配文件名和目录名的模式。通配是定义一个或多个 glob 模式，并从包含或排除的匹配项中生成文件的操作。

模式

若要基于用户定义的模式匹配文件系统中的文件，请通过实例化 Matcher 对象开始。`Matcher` 可在没有参数的情况下进行实例化，也可以使用 System.StringComparison 参数进行实例化，该参数在内部用于比较模式和文件名。`Matcher` 公开以下附加方法：

- Matcher.AddExclude
- Matcher.AddInclude

`AddExclude` 和 `AddInclude` 方法都可以调用任意多次，以添加各种要从结果中排除或包含文件名模式。在实例化了 `Matcher` 并添加了模式后，它用于通过 Matcher.Execute 方法从起始目录评估匹配项。

扩展方法

`Matcher` 对象具有多个扩展方法。

多个排除项

若要添加多个排除模式，可使用：

```
Matcher matcher = new();
matcher.AddExclude("*.txt");
matcher.AddExclude("*.asciidoc");
matcher.AddExclude("*.md");
```

或者，可使用 MatcherExtensions.AddExcludePatterns(Matcher, IEnumerable<String>[]) 在单个调用中添加多个排除模式：

```
Matcher matcher = new();
matcher.AddExcludePatterns(new [] { "*.txt", "*.asciidoc", "*.md" });
```

此扩展方法循环访问所有代你调用 AddExclude 的模式。

多个包含项

若要添加多个包含模式，可使用：

```
Matcher matcher = new();
matcher.AddInclude("*.txt");
matcher.AddInclude("*.asciidoc");
matcher.AddInclude("*.md");
```

或者，可使用 MatcherExtensions.AddIncludePatterns(Matcher, IEnumerable<String>[]) 在单个调用中添加多个包含模式：

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });
```

此扩展方法循环访问所有代你调用 AddInclude 的模式。

获取所有匹配文件

若要获取所有匹配文件，必须直接或间接调用 Matcher.Execute(DirectoryInfoBase)。若要直接调用它，需要一个搜索目录：

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });

string searchDirectory = "../starting-folder/";

PatternMatchingResult result = matcher.Execute(
    new DirectoryInfoWrapper(
        new DirectoryInfo(searchDirectory)));

// Use result.HasMatches and results.Files.
// The files in the results object are file paths relative to the search directory.
```

上述 C# 代码：

- 实例化 Matcher 对象。
- 调用 AddIncludePatterns(Matcher, IEnumerable<String>[]) 以添加几个要包含的文件名模式。
- 声明并分配搜索目录值。
- 从给定的 DirectoryInfo 实例化 `searchDirectory`。
- 从它包装的 DirectoryInfoWrapper 中实例化 `DirectoryInfo`。
- 在给定 `DirectoryInfoWrapper` 实例的情况下调用 `Execute` 以生成 PatternMatchingResult 对象。

> **NOTE**
> `DirectoryInfoWrapper` 类型在 `Microsoft.Extensions.FileSystemGlobbing.Abstractions` 命名空间中定义，而 `DirectoryInfo` 类型在 `System.IO` 命名空间中定义。若要避免不必要的 `using` 语句，可使用提供的扩展方法。

还有另一种扩展方法可生成表示匹配文件的 `IEnumerable<string>`：

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "*.txt", "*.asciidoc", "*.md" });

string searchDirectory = "../starting-folder/";

IEnumerable<string> matchingFiles = matcher.GetResultsInFullPath(searchDirectory);

// Use matchingFiles if there are any found.
// The files in this collection are fully qualified file system paths.
```

上述 C# 代码：

- 实例化 Matcher 对象。
- 调用 AddIncludePatterns(Matcher, IEnumerable<String>[]) 以添加几个要包含的文件名模式。
- 声明并分配搜索目录值。
- 在给定 `searchDirectory` 值的情况下调用 `GetResultsInFullPath` 以将所有匹配文件生成为 `IEnumerable<string>`。

匹配重载

PatternMatchingResult 对象表示 FilePatternMatch 实例的集合，并公开一个 `boolean` 值，该值指示结果是否有匹配项 —PatternMatchingResult.HasMatches。

使用 `Matcher` 实例，可调用各种 `Match` 重载中的任何一个来获取模式匹配结果。`Match` 方法反转了调用方的责任，即提供一个在其中评估匹配项的文件或文件集合。换言之，调用方负责传递要匹配的文件。

> **IMPORTANT**
> 使用任何 `Match` 重载时，都不涉及文件系统 I/O。所有文件通配都是在内存中通过 `matcher` 实例的包含和排除模式完成的。`Match` 重载的参数不必是完全限定的路径。如果未指定，将使用当前目录 (Directory.GetCurrentDirectory())。

匹配单个文件：

```
Matcher matcher = new();
matcher.AddInclude("**/*.md");

PatternMatchingResult result = matcher.Match("file.md");
```

上述 C# 代码：

- 在任意目录深度匹配任何带有 .md文件扩展名的文件。
- 如果名为 file.md 的文件存在于当前目录的子目录中：
  - `result.HasMatches` 将为 `true`。
  - 且 `result.Files` 将有一个匹配项。

其他 `Match` 重载的工作方式类似。

模式格式

`AddExclude` 和 `AddInclude` 方法中指定的模式可使用以下格式来匹配多个文件或目录。

- 确切的目录名或文件名
  - `some-file.txt`
  - `path/to/file.txt`
- 文件名和目录名中的通配符 `*`，表示零到多个字符，不包括分隔符。

[	[[
`*.txt`	具有 .txt 文件扩展名的所有文件。
`*.*`	具有一个扩展名的所有文件。
`*`	顶层目录中的所有文件。
`.*`	以“.”开头的文件名称。
`*word*`	文件名中包含“word”的所有文件。
`readme.*`	所有带有任何文件扩展名且名为“readme”的文件。
`styles/*.css`	目录“styles/”中扩展名为“.css”的所有文件。

[	[[
`scripts/*/*`	“scripts/”中的或“scripts/”下一级子目录中的所有文件。
`images*/*`	文件夹中名称为“images”或名称以“images”开头的所有文件。

- 任意目录深度（`/**/`）。

[	[[
`**/*`	任何子目录中的所有文件。
`dir/**/*`	“dir/”下任何子目录中的所有文件。

- 相对路径。

  若要将同级名为“shared”的目录中的所有文件与指定给 Matcher.Execute(DirectoryInfoBase) 的基本目录相匹配，请使用 `../shared/*`。

示例

请考虑下面的示例目录，以及相应文件夹中的每个文件。

```
📁 parent
│    file.md
│    README.md
│
└──📁 child
    │    file.MD
    │    index.js
    │    more.md
    │    sample.mtext
    │
    ├──📁 assets
    │        image.png
    │        image.svg
    │
    └──📁 grandchild
             file.md
             style.css
             sub.text
```

> **TIP**
> 某些文件扩展名采用大写形式，而另一些扩展名采用小写形式。默认使用 StringComparer.OrdinalIgnoreCase。若要指定不同的字符串比较行为，请使用 Matcher.Matcher(StringComparison) 构造函数。

获取所有文件扩展名为 .md 或 .mtext 的 Markdown 文件（不考虑字符大小写）：

```
Matcher matcher = new();
matcher.AddIncludePatterns(new[] { "**/*.md", "**/*.mtext" });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

运行该应用程序将输出类似于以下内容的结果：

```
C:\app\parent\file.md
C:\app\parent\README.md
C:\app\parent\child\file.MD
C:\app\parent\child\more.md
C:\app\parent\child\sample.mtext
C:\app\parent\child\grandchild\file.md
```

在任意深度获取 assets 目录中的任何文件：

```
Matcher matcher = new();
matcher.AddInclude("**/assets/**/*");

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

运行该应用程序将输出类似于以下内容的结果：

```
C:\app\parent\child\assets\image.png
C:\app\parent\child\assets\image.svg
```

获取目录名在任意深度包含单词 child 且文件扩展名不是 .md、.text 或 .mtext 的任何文件：

```
Matcher matcher = new();
matcher.AddInclude("**/*child/**/*");
matcher.AddExcludePatterns(
    new[]
    {
        "**/*.md", "**/*.text", "**/*.mtext"
    });

foreach (string file in matcher.GetResultsInFullPath("parent"))
{
    Console.WriteLine(file);
}
```

运行该应用程序将输出类似于以下内容的结果：

```
C:\app\parent\child\index.js
C:\app\parent\child\assets\image.png
C:\app\parent\child\assets\image.svg
C:\app\parent\child\grandchild\style.css
```

请参阅

- 运行时库概述
- 文件和流 I/O

基元：适用于 .NET 的扩展库

2021/11/16 •

本文介绍 Microsoft.Extensions.Primitives 库。本文中的基元不会与来自 BCL 的 .NET 基元类型或 C# 语言的基元类型混淆。相反，基元库中的类型用作某些外围 .NET NuGet 包的构建基块，如：

- `Microsoft.Extensions.Configuration`
- `Microsoft.Extensions.Configuration.FileExtensions`
- `Microsoft.Extensions.FileProviders.Composite`
- `Microsoft.Extensions.FileProviders.Physical`
- `Microsoft.Extensions.Logging.EventSource`
- `Microsoft.Extensions.Options`
- `System.Text.Json`

更改通知

当发生更改时传播通知是编程中的基本概念。对象的观察状态在更多时候是可以改变的。发生更改时，可以使用 Microsoft.Extensions.Primitives.IChangeToken 接口的实现将上述更改通知到相关各方。可用的实现如下：

- CancellationChangeToken
- CompositeChangeToken

作为开发人员，你也可以自由地实现自己的类型。IChangeToken 接口会定义几个属性：

- IChangeToken.HasChanged：获取一个指示是否发生更改的值。
- IChangeToken.ActiveChangeCallbacks：指示令牌是否将主动引发回叫。如果为 `false`，则令牌使用者必须轮询 `HasChanged` 以检测更改。

基于实例的功能

请看下面 `CancellationChangeToken` 中的使用实例：

```
CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken = new(cancellationTokenSource.Token);

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}");

Action<object?> callback = _ => Console.WriteLine("The callback was invoked.");

using (IDisposable subscription =
    cancellationChangeToken.RegisterChangeCallback(callback, null))
{
    cancellationTokenSource.Cancel();
}

Console.WriteLine($"HasChanged: {cancellationChangeToken.HasChanged}\n");

// Outputs:
//     HasChanged: False
//     The callback was invoked.
//     HasChanged: True
```

在前面的示例中，CancellationTokenSource 进行实例化，并且它的 Token 被传递给 CancellationChangeToken 构

造函数。`HasChanged` 的初始状态会写入控制台。创建了一个 `Action<object?> callback`，当调用回调时，它会写到控制台。在给定 `callback` 的情况下，调用该令牌的 RegisterChangeCallback(Action<Object>, Object) 方法。在 `using` 语句中，`cancellationTokenSource` 已取消。这会触发回叫，并再次将 `HasChanged` 的状态写入控制台。

如果需要从多个更改源执行操作，请使用 CompositeChangeToken。此实现聚合一个或多个更改令牌，而且无论触发更改的次数如何，每个已注册的回叫都仅引发一次。请考虑以下示例：

```
CancellationTokenSource firstCancellationTokenSource = new();
CancellationChangeToken firstCancellationChangeToken = new(firstCancellationTokenSource.Token);

CancellationTokenSource secondCancellationTokenSource = new();
CancellationChangeToken secondCancellationChangeToken = new(secondCancellationTokenSource.Token);

CancellationTokenSource thirdCancellationTokenSource = new();
CancellationChangeToken thirdCancellationChangeToken = new(thirdCancellationTokenSource.Token);

var compositeChangeToken =
    new CompositeChangeToken(
        new IChangeToken[]
        {
            firstCancellationChangeToken,
            secondCancellationChangeToken,
            thirdCancellationChangeToken
        });

Action<object?> callback = state => Console.WriteLine($"The {state} callback was invoked.");

// 1st, 2nd, 3rd, and 4th.
compositeChangeToken.RegisterChangeCallback(callback, "1st");
compositeChangeToken.RegisterChangeCallback(callback, "2nd");
compositeChangeToken.RegisterChangeCallback(callback, "3rd");
compositeChangeToken.RegisterChangeCallback(callback, "4th");

// It doesn't matter which cancellation source triggers the change.
// If more than one trigger the change, each callback is only fired once.
Random random = new();
int index = random.Next(3);
CancellationTokenSource[] sources = new[]
{
    firstCancellationTokenSource,
    secondCancellationTokenSource,
    thirdCancellationTokenSource
};
sources[index].Cancel();

Console.WriteLine();

// Outputs:
//     The 4th callback was invoked.
//     The 3rd callback was invoked.
//     The 2nd callback was invoked.
//     The 1st callback was invoked.
```

在前面的 C# 代码中，创建了三个 CancellationTokenSource 对象实例，并将其与相应的 CancellationChangeToken 实例配对。通过将令牌的数组传递给 CompositeChangeToken 构造函数来实例化复合标记。创建了 `Action<object?> callback`，但是这次，`state` 对象是作为已设置格式的消息来使用并写入控制台。回叫注册了四次，每次都有一个略微不同的状态对象参数。该代码使用伪随机数生成器选取一个更改令牌源（具体是哪个并不重要）并调用其 Cancel() 方法。这会触发更改，同时仅调用一次每个已注册的回叫。

替代的 `static` 方法

作为调用 `RegisterChangeCallback` 的替代方法，可以使用 Microsoft.Extensions.Primitives.ChangeToken 静态类。

来看看以下使用模式：

```
CancellationTokenSource cancellationTokenSource = new();
CancellationChangeToken cancellationChangeToken = new(cancellationTokenSource.Token);

Func<IChangeToken> producer = () =>
{
    // The producer factory should always return a new change token.
    // If the token's already fired, get a new token.
    if (cancellationTokenSource.IsCancellationRequested)
    {
        cancellationTokenSource = new();
        cancellationChangeToken = new(cancellationTokenSource.Token);
    }

    return cancellationChangeToken;
};

Action consumer = () => Console.WriteLine("The callback was invoked.");

using (ChangeToken.OnChange(producer, consumer))
{
    cancellationTokenSource.Cancel();
}

// Outputs:
//     The callback was invoked.
```

与前面的示例非常类似，你需要一个由 `changeTokenProducer` 生成的 `IChangeToken` 的实现。生成方被定义为 `Func<IChangeToken>`，预计每次调用都会返回一个新的令牌。当不使用 `state` 时，`consumer` 是一个 `Action`，或者是一个 `Action<TState>`，其中泛型类型 `TState` 流经更改通知。

字符串 tokenizer、段和值

在应用程序开发中，与字符串交互是非常常见的。对字符串的各种表示形式进行分析、拆分或循环访问。基元库提供了几种选择类型，有助于使与字符串的交互更加完善和高效。来看看以下类型：

- StringSegment：子字符串的优化表示形式。
- StringTokenizer：将 `string` 标记为 `StringSegment` 实例。
- StringValues：以有效方式表示 `null`、零个、一个或多个字符串。

`StringSegment` 类型

在本部分中，你将了解称为 StringSegment `struct` 类型的子字符串的优化表示形式。来看看以下 C# 代码示例，其中显示了一些 `StringSegment` 属性和 `AsSpan` 方法：

```
var segment =
    new StringSegment(
        "This a string, within a single segment representation.",
        14, 25);

Console.WriteLine($"Buffer: \"{segment.Buffer}\"");
Console.WriteLine($"Offset: {segment.Offset}");
Console.WriteLine($"Length: {segment.Length}");
Console.WriteLine($"Value: \"{segment.Value}\"");

Console.Write("Span: \"");
foreach (char @char in segment.AsSpan())
{
    Console.Write(@char);
}
Console.Write("\"\n");

// Outputs:
//     Buffer: "This a string, within a single segment representation."
//     Offset: 14
//     Length: 25
//     Value: " within a single segment "
//     " within a single segment "
```

在给定 `string` 值、`offset` 和 `length` 的情况下，前面的代码对 `StringSegment` 进行实例化。StringSegment.Buffer 是原始字符串参数，StringSegment.Value 是基于 StringSegment.Offset 和 StringSegment.Length 值的子字符串。

`StringSegment` 结构提供了许多方法，用于与段进行交互。

`StringTokenizer` 类型

StringTokenizer 对象是一个结构类型，用于将 `string` 标记到 `StringSegment` 实例中。较大字符串的标记化通常涉及将字符串拆分开并循环访问它。说到这个，可能会想到 String.Split。这些 API 都是类似的，但通常情况下，StringTokenizer 提供的性能更好。首先来看下面的示例：

```
var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ' });

foreach (StringSegment segment in tokenizer)
{
    // Interact with segment
}
```

在前面的代码中，在给定 900 个自动生成的 Lorem Ipsum 文本段落和一个带有空白字符 `' '` 的单个值的数组的情况下，创建了 `StringTokenizer` 类型的一个实例。tokenizer 中的每个值都表示为 `StringSegment`。代码循环访问段，允许使用者与每个 `segment` 进行交互。

对 `StringTokenizer` 和 `string.Split` 进行基准比较

由于有各种切分字符串的方法，感觉用一个基准来比较两种方法比较合适。使用 BenchmarkDotNet NuGet 包，可以考虑以下两个基准方法：

1. 使用 StringTokenizer：

```
StringBuilder buffer = new();

var tokenizer =
    new StringTokenizer(
        s_nineHundredAutoGeneratedParagraphsOfLoremIpsum,
        new[] { ' ', '.' });

foreach (StringSegment segment in tokenizer)
{
    buffer.Append(segment.Value);
}
```

2. 使用 String.Split：

```
StringBuilder buffer = new();

string[] tokenizer =
    s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(
        new[] { ' ', '.' });

foreach (string segment in tokenizer)
{
    buffer.Append(segment);
}
```

这两种方法在 API 图区上看起来相似，它们都可以将大型字符串拆分为多个块。下面的基准结果显示，`StringTokenizer` 方法几乎快了三倍，但结果可能会有所不同。与所有性能注意事项一样，应评估具体用例。

‖	‖‖	‖	‖‖‖	‖	‖	‖‖‖‖
`Tokenizer`	6.306 毫秒	0.1481 毫秒	0.4179 毫秒	6.175 毫秒	0.37	0.04
`Split`	16.966 毫秒	0.6164 毫秒	1.8079 毫秒	16.862 毫秒	1.00	0.00

图例

- 均值：所有度量值的算术平均值
- 错误：99.9% 的置信区间的一半
- 标准偏差：所有度量值的标准偏差
- 中值：此值将所有度量值中较高的一半分隔开（第 50 个百分位）
- 比率：比率分布的平均值（当前/基线）
- 比率标准偏差：比率分布的标准偏差（当前/基线）
- 1 ms：1 毫秒（0.001 秒）

有关 .NET 基准测试的详细信息，请参阅 BenchmarkDotNet。

`StringValues` 类型

StringValues 对象是一个 `struct` 类型，以有效方式表示 `null`、零个、一个或多个字符串。可以通过以下语法之一构造 `StringValues` 类型：`string?` 或 `string?[]?`。使用上一示例中的文本，考虑以下 C# 代码：

```
StringValues values =
    new(s_nineHundredAutoGeneratedParagraphsOfLoremIpsum.Split(
        new[] { '\n' }));

Console.WriteLine($"Count = {values.Count:#,#}");

foreach (string? value in values)
{
    // Interact with the value
}
// Outputs:
//     Count = 1,799
```

前面的代码在给定字符串值数组的情况下实例化 `StringValues` 对象。StringValues.Count 被写入控制台。

`StringValues` 类型是以下集合类型的一个实现：

- `IList<string>`
- `ICollection<string>`
- `IEnumerable<string>`
- `IEnumerable`
- `IReadOnlyList<string>`
- `IReadOnlyCollection<string>`

因此，它可以被循环访问，每个 `value` 都可以根据需要进行交互。

另请参阅

- .NET 中的选项模式
- .NET 中的配置
- .NET 中的日志记录提供程序

全球化和本地化 .NET 应用程序

2021/11/16 •

开发全球通用的应用程序(包括可本地化为一种或多种语言的应用程序)分为三步:全球化、可本地化性审核和本地化。

全球化

此步骤包括设计和编码非特定区域性和非特定语言的应用程序,以及设计和编码支持所有用户的本地化用户界面和区域数据的应用程序。它涉及做出不基于区域性特定的假设的设计和编程决策。虽然全球化应用程序未本地化,但它经过设计和编写,因此随后可相对容易地将其本地化为一种或多种语言。

本地化评审

此步骤包括评审应用程序的代码和设计,以确保能轻松本地化应用程序和标识本地化的潜在障碍,并验证应用程序的可执行代码是否与其资源分隔开。如果全球化阶段有效,则本地化检查将确认全球化过程中做出的设计和编码选择。本地化阶段还可以标识任何遗留问题,以便在本地化阶段不必修改应用程序的源代码。

本地化

此步骤包括为特定区域性或区域自定义应用程序。如果已正确执行全球化和本地化分析这两个步骤,则本地化主要包括用户界面的翻译。

遵循这三个步骤有两大好处:

- 让你无需改进为支持单个区域性(如美国英语)而专门设计的应用程序以支持其他区域性。英语,以支持其他区域性。
- 这会产生更加稳定并有少量 Bug 的本地化应用程序。

.NET 为开发全球通用和本地化的应用程序提供了广泛支持。特别是,.NET 类库中许多类成员通过返回可反映当前用户区域性或指定区域性约定的值来帮助全球化。此外,.NET 支持附属程序集,这可推动本地化应用程序的过程。

本节内容

全球化

讨论创建全球通用的应用程序的第一个阶段,该阶段包括设计和编码为非特定区域性和非特定语言的应用程序。

.NET 全球化和 ICU

介绍了 .NET 全球化如何使用 Unicode 国际组件 (ICU)。

本地化评审

讨论创建本地化应用程序的第二个阶段,该阶段包含标识本地化的潜在路障。

本地化

讨论创建本地化应用程序的最后阶段,该阶段包含自定义应用程序的特定区域或区域性的用户界面。

不区分区域性的字符串操作

描述默认情况下如何使用区分区域性的 .NET 方法和类来获得不区分区域性的结果。

开发全球通用应用程序的最佳做法

描述在全球化、本地化和开发全球通用的 ASP.NET 应用程序时遵循的最佳做法。

参考

- System.Globalization 命名空间

  包含定义区域性相关信息的类，这些信息包括语言、国家/地区、正在使用的日历、日期的格式模式、货币、数字以及字符串的排序顺序。

- System.Resources 命名空间

  提供用于创建、操作和使用资源的类。

- System.Text 命名空间

  包含表示 ASCII、ANSI、Unicode 以及其他字符编码的类。

- Resgen.exe（资源文件生成器）

  描述如何使用 Resgen.exe 将 .txt 文件和基于 XML 的资源格式 (.resx) 文件转换为公共语言运行时二进制 .resources 文件。

- Winres.exe（Windows 窗体资源编辑器）

  描述如何使用 Winres.exe 本地化 Windows 窗体。

全球化

2021/11/16 •

全球化涉及到设计和开发世界通用的应用，这些应用支持本地化界面和区域数据，供位于多个区域性的用户使用。在设计阶段开始之前，应确定应用将支持哪些区域性。虽然应用以单一区域性或区域作为默认目标，但可设计和编写应用，使其能够轻松地供其他区域性或区域的用户使用。

作为开发人员，我们对由区域性组成的用户界面和数据都做出过假设。例如，对于说英语的美国开发人员来说，将日期和时间数据序列化为采用 `MM/dd/yyyy hh:mm:ss` 格式的字符串似乎是相当合理的。但是，如果在处于不同区域性的系统上将该字符串进行反序列化，则可能会引发 FormatException 异常或生成不准确的数据。全球化使我们能够识别这些特定于区域性的假设，并确保它们不会影响到应用的设计和编码。

本文将讨论在全球化应用中处理字符串、日期和时间值以及数值时，应考虑的一些主要问题和遵从的最佳做法。

字符串

处理字符和字符串的是全球化的重点，因为每个区域性或区域可能使用不同的字符和字符集，且排序方式也不同。本节提供在全球化应用中使用字符串的一些建议。

在内部使用 Unicode

默认情况下，.NET 使用 Unicode 字符串。一个 Unicode 字符串由零个、一个或多个 Char 对象组成，其中每个对象表示一个 UTF-16 代码单元。对于每个字符集中的几乎每个字符来说，都有一个在全球范围内使用的 Unicode 表达式。

许多应用程序和操作系统（包括 Windows 操作系统）也可以使用代码页来表示字符集。代码页通常包含从 0x00 到 0x7F 的标准 ASCII 值，并将其他字符映射到从 0x80 到 0xFF 的剩余值。从 0x80 到 0xFF 的值的解释取决于具体的代码页。因此，如有可能，应避免在全球化应用中使用代码页。

以下示例阐释了当系统上的默认代码页与保存数据的代码页不同时，解释代码页数据的危险。（若要模拟此场景，示例应明确指定不同的代码页。）首先，示例定义了一个由希腊字母表的大写字符组成的数组。然后使用代码页 737（也称为 MS-DOS 希腊语）将其编码成一个字节数组，并保存到文件。如果检索该文件并使用代码页 737 对其字节数组进行解码，则会还原原始字符。但是，如果检索该文件并使用代码页 1252（或按拉丁字母表来表示字符的 Windows-1252）对其字节数组进行解码，原始数据则会丢失。

```
using System;
using System.IO;
using System.Text;

public class Example
{
   public static void Main()
   {
      // Represent Greek uppercase characters in code page 737.
      char[] greekChars = { 'Α', 'Β', 'Γ', 'Δ', 'Ε', 'Ζ', 'Η', 'Θ',
                            'Ι', 'Κ', 'Λ', 'Μ', 'Ν', 'Ξ', 'Ο', 'Π',
                            'Ρ', 'Σ', 'Τ', 'Υ', 'Φ', 'Χ', 'Ψ', 'Ω' };

      Encoding cp737 = Encoding.GetEncoding(737);
      int nBytes = cp737.GetByteCount(greekChars);
      byte[] bytes737 = new byte[nBytes];
      bytes737 = cp737.GetBytes(greekChars);
      // Write the bytes to a file.
      FileStream fs = new FileStream(@".\\CodePageBytes.dat", FileMode.Create);
      fs.Write(bytes737, 0, bytes737.Length);
      fs.Close();

      // Retrieve the byte data from the file.
      fs = new FileStream(@".\\CodePageBytes.dat", FileMode.Open);
      byte[] bytes1 = new byte[fs.Length];
      fs.Read(bytes1, 0, (int)fs.Length);
      fs.Close();

      // Restore the data on a system whose code page is 737.
      string data = cp737.GetString(bytes1);
      Console.WriteLine(data);
      Console.WriteLine();

      // Restore the data on a system whose code page is 1252.
      Encoding cp1252 = Encoding.GetEncoding(1252);
      data = cp1252.GetString(bytes1);
      Console.WriteLine(data);
   }
}
// The example displays the following output:
//       ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩ
//       €‚ƒ„…†‡ˆ‰Š‹ŒŽ‘’""•–—
```

```
Imports System.IO
Imports System.Text

Module Example
    Public Sub Main()
        ' Represent Greek uppercase characters in code page 737.
        Dim greekChars() As Char = {"A"c, "B"c, "Γ"c, "Δ"c, "E"c, "Z"c, "H"c, "Θ"c,
                                    "I"c, "K"c, "Λ"c, "M"c, "N"c, "Ξ"c, "O"c, "Π"c,
                                    "P"c, "Σ"c, "T"c, "Y"c, "Φ"c, "X"c, "Ψ"c, "Ω"c}

        Dim cp737 As Encoding = Encoding.GetEncoding(737)
        Dim nBytes As Integer = CInt(cp737.GetByteCount(greekChars))
        Dim bytes737(nBytes - 1) As Byte
        bytes737 = cp737.GetBytes(greekChars)
        ' Write the bytes to a file.
        Dim fs As New FileStream(".\CodePageBytes.dat", FileMode.Create)
        fs.Write(bytes737, 0, bytes737.Length)
        fs.Close()

        ' Retrieve the byte data from the file.
        fs = New FileStream(".\CodePageBytes.dat", FileMode.Open)
        Dim bytes1(CInt(fs.Length - 1)) As Byte
        fs.Read(bytes1, 0, CInt(fs.Length))
        fs.Close()

        ' Restore the data on a system whose code page is 737.
        Dim data As String = cp737.GetString(bytes1)
        Console.WriteLine(data)
        Console.WriteLine()

        ' Restore the data on a system whose code page is 1252.
        Dim cp1252 As Encoding = Encoding.GetEncoding(1252)
        data = cp1252.GetString(bytes1)
        Console.WriteLine(data)
    End Sub
End Module
' The example displays the following output:
'       ABΓΔEZHΘIKΛMNΞOΠPΣTYΦXΨΩ
'       €,ƒ„…†‡ˆ‰Š‹Œ‘’""•–—
```

使用 Unicode 可确保相同的代码单元始终能映射到相同的字符，并且相同的字符始终能映射到相同的字节数组。

使用资源文件

即使在开发以单一区域性或区域为目标的应用时，也应使用资源文件存储显示在用户界面中的字符串和其他资源。切勿将它们直接添加到代码中。使用资源文件有许多优点：

- 所有字符串都在一个位置。不必搜索整个源代码即可识别要为特定的语言或区域性修改的字符串。
- 不需要重复字符串。不使用资源文件的开发人员常常在多个源代码文件中定义同一字符串。此类重复增加了在修改字符串时忽略一个或多个实例的可能性。
- 可以将非字符串资源（如图像或二进制数据）包含在资源文件中以便于检索，而不将它们存储在单独的独立文件中。

对于创建本地化应用来说，使用资源文件具有独特优势。在附属程序集中部署资源时，公共语言运行时会基于由 CultureInfo.CurrentUICulture 属性定义的用户当前 UI 区域性来自动选择适合区域性的资源。只要提供了相应的区域性特定资源并正确示例化了 ResourceManager 对象或使用了强类型的资源类，运行时就会负责检索适合的资源。

若要详细了解如何创建资源文件，请参阅创建资源文件。若要了解如何创建和部署附属程序集，请参阅创建附属程序集以及打包和部署资源。

搜索和比较字符串

应尽可能地将字符串按整个字符串处理，而不是按一系列单个字符进行处理。尤其重要的一点是在排序或搜索子字符串时，要防止出现与分析组合字符相关的问题。

> **TIP**
> 可使用 StringInfo 类与文本元素配合使用，而不使用字符串中的单个字符。

在字符串搜索和比较中，常见的错误是将字符串作为字符的集合，其中每个字符由 Char 对象表示。实际上，单个字符串可能由一个、两个或多个 Char 对象组成。此类字符在一些字符串中出现得最频繁，这些字符串位于其字母表是由 Unicode 基本拉丁字符范围（从 U+0021 到 U+007E）以外的字符所组成的区域性中。以下示例尝试在字符串中查找 LATIN CAPITAL LETTER A WITH GRAVE 字符 (U+00C0) 的索引。但是，此字符有两种表示方法：单个代码单元 (U+00C0) 或复合字符（两个代码单元：U+0041 和 U+0300）。在这种情况下，字符在字符串示例中用两个 Char 对象（U+0041 和 U+0300）表示。示例代码调用 String.IndexOf(Char) 和 String.IndexOf(String) 重载以查找此字符在字符串实例中的位置，但返回了不同的结果 。第一个方法调用拥有 Char 参数，它执行的是序号比较，因此无法找到匹配项。第二个调用拥有 String 参数，它执行的是区分区域性的比较，因此找到了匹配项。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("pl-PL");
      string composite = "\u0041\u0300";
      Console.WriteLine("Comparing using Char:   {0}", composite.IndexOf('\u00C0'));
      Console.WriteLine("Comparing using String: {0}", composite.IndexOf("\u00C0"));
   }
}
// The example displays the following output:
//       Comparing using Char:   -1
//       Comparing using String: 0
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("pl-PL")
        Dim composite As String = ChrW(&h0041) + ChrW(&H0300)
        Console.WriteLine("Comparing using Char:   {0}", composite.IndexOf(ChrW(&h00C0)))
        Console.WriteLine("Comparing using String: {0}", composite.IndexOf(ChrW(&h00C0).ToString()))
    End Sub
End Module
' The example displays the following output:
'       Comparing using Char:   -1
'       Comparing using String: 0
```

可以通过调用包含 StringComparison 参数（如 String.IndexOf(String, StringComparison) 或 String.LastIndexOf(String, StringComparison) 方法）的重载来避免此示例中出现的一些多义性（对方法的两个相似重载的调用返回了不同的结果）。

但是，搜索并不总是区分区域性的。如果搜索的目的在于做出安全性决策或是允许或禁止访问某些资源，应进行序号比较，此主题将在下一节中讨论。

测试字符串的相等性

如果要测试两个字符串是否相等，而不是确定如何按排序顺序进行比较，则使用 String.Equals 方法，而不是字符串比较方法，如 String.Compare 或 CompareInfo.Compare。

通常比较相等性用于条件性地访问某些资源。例如，可能需要比较相等性以验证密码或确认文件是否存在。此类非语义比较应始终为序号比较，而不是区分区域性的比较。一般情况下，应使用值为 StringComparison.Ordinal 的字符串（如密码）和值为 StringComparison.OrdinalIgnoreCase 的字符串（如文件名或 URI）来调用实例 String.Equals(String, StringComparison) 方法或静态的 String.Equals(String, String, StringComparison) 方法。

相等性的比较有时会涉及到搜索或子字符串比较，而不是对 String.Equals 方法的调用。在某些情况下，可以使用子字符串搜索以确定子字符串是否与另一字符串相等。如果比较的目的是非语义的，那么搜索也应该为序号搜索，而不区分区域性。

以下示例阐释了对非语义数据进行区分区域性搜索的危险。`AccessesFileSystem` 方法旨在禁止文件系统访问以子字符串“FILE”开头的 URI。为此，它对以字符串“FILE”开头的 URI 执行区分区域性、不区分大小写的比较。由于访问文件系统的 URI 可以“FILE:”或“file:”开头，因此隐式假设“i”(U+0069) 始终为小写且等效于“I”(U+0049)。但是，在土耳其语和阿塞拜疆语中，“i”的大写为“İ”(U+0130)。由于存在此差异，因此区分区域性的比较在应禁止的情况下仍允许进行文件系统访问。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("tr-TR");
      string uri = @"file:\\c:\users\username\Documents\bio.txt";
      if (! AccessesFileSystem(uri))
         // Permit access to resource specified by URI
         Console.WriteLine("Access is allowed.");
      else
         // Prohibit access.
         Console.WriteLine("Access is not allowed.");
   }

   private static bool AccessesFileSystem(string uri)
   {
      return uri.StartsWith("FILE", true, CultureInfo.CurrentCulture);
   }
}
// The example displays the following output:
//         Access is allowed.
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("tr-TR")
        Dim uri As String = "file:\\c:\users\username\Documents\bio.txt"
        If Not AccessesFileSystem(uri) Then
            ' Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.")
        Else
            ' Prohibit access.
            Console.WriteLine("Access is not allowed.")
        End If
    End Sub

    Private Function AccessesFileSystem(uri As String) As Boolean
        Return uri.StartsWith("FILE", True, CultureInfo.CurrentCulture)
    End Function
End Module
' The example displays the following output:
'       Access is allowed.
```

因此，可执行忽视大小写的序号比较来避免出现此问题，如下例所示。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("tr-TR");
      string uri = @"file:\\c:\users\username\Documents\bio.txt";
      if (! AccessesFileSystem(uri))
         // Permit access to resource specified by URI
         Console.WriteLine("Access is allowed.");
      else
         // Prohibit access.
         Console.WriteLine("Access is not allowed.");
   }

   private static bool AccessesFileSystem(string uri)
   {
      return uri.StartsWith("FILE", StringComparison.OrdinalIgnoreCase);
   }
}
// The example displays the following output:
//         Access is not allowed.
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("tr-TR")
        Dim uri As String = "file:\\c:\users\username\Documents\bio.txt"
        If Not AccessesFileSystem(uri) Then
            ' Permit access to resource specified by URI
            Console.WriteLine("Access is allowed.")
        Else
            ' Prohibit access.
            Console.WriteLine("Access is not allowed.")
        End If
    End Sub

    Private Function AccessesFileSystem(uri As String) As Boolean
        Return uri.StartsWith("FILE", StringComparison.OrdinalIgnoreCase)
    End Function
End Module
' The example displays the following output:
'       Access is not allowed.
```

顺序和排序字符串

通常，要在用户界面中显示的已排列字符串应根据区域性进行排序。大多数情况下，在调用排序字符串的方法（如 Array.Sort 和 List<T>.Sort）时，此类字符串比较是由 .NET 隐式处理的。默认情况下，使用当前区域性的排序约定对字符串进行排序。以下示例阐释了使用英语(美国)区域性和瑞典语(瑞典)区域性的约定对一组字符串进行排序时的差异。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] values = { "able", "ångström", "apple", "Æble",
                          "Windows", "Visual Studio" };
      // Change thread to en-US.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      // Sort the array and copy it to a new array to preserve the order.
      Array.Sort(values);
      string[] enValues = (String[]) values.Clone();

      // Change culture to Swedish (Sweden).
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("sv-SE");
      Array.Sort(values);
      string[] svValues = (String[]) values.Clone();

      // Compare the sorted arrays.
      Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US", "sv-SE");
      for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
         Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr], svValues[ctr]);
   }
}
// The example displays the following output:
//       Position en-US           sv-SE
//
//       0        able            able
//       1        Æble            Æble
//       2        ångström        apple
//       3        apple           Windows
//       4        Visual Studio   Visual Studio
//       5        Windows         ångström
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim values() As String = {"able", "ångström", "apple", _
                                  "Æble", "Windows", "Visual Studio"}
        ' Change thread to en-US.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        ' Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values)
        Dim enValues() As String = CType(values.Clone(), String())

        ' Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture = New CultureInfo("sv-SE")
        Array.Sort(values)
        Dim svValues() As String = CType(values.Clone(), String())

        ' Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}", "Position", "en-US", "sv-SE")
        Console.WriteLine()
        For ctr As Integer = 0 To values.GetUpperBound(0)
            Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues(ctr), svValues(ctr))
        Next
    End Sub
End Module
' The example displays the following output:
'       Position en-US           sv-SE
'
'       0        able            able
'       1        Æble            Æble
'       2        ångström        apple
'       3        apple           Windows
'       4        Visual Studio   Visual Studio
'       5        Windows         ångström
```

区分区域性的字符串比较是由 CompareInfo 对象定义的，该对象由每个区域性的 CultureInfo.CompareInfo 属性返回。使用 String.Compare 方法重载的区分区域性的字符串比较也使用 CompareInfo 对象。

.NET 使用表格对字符串数据进行区分区域性的排序。这些表格的内容包含了数据的排序权重和字符串标准化，而这些内容是通过 .NET 的特定版本实现的 Unicode 标准版确定的。下表列出了通过 .NET 指定版本所实现的 Unicode 版本。受支持的 Unicode 版本列表仅适用于字符比较和排序；不适用于按类别分类的 Unicode 字符。有关详细信息，请参阅 String 文章中"字符串和 Unicode 标准"部分。

.NET Framework		Unicode
.NET Framework 2.0	所有操作系统	Unicode 4.1
.NET Framework 3.0	所有操作系统	Unicode 4.1
.NET Framework 3.5	所有操作系统	Unicode 4.1
.NET Framework 4	所有操作系统	Unicode 5.0
Windows 7 上的 .NET Framework 4.5 及更高版本	Unicode 5.0	
Windows 8 和更高版本的操作系统上的 .NET Framework 4.5 及更高版本		Unicode 6.3.0

.NET Framework 版本	说明	Unicode 版本
.NET Core 和 .NET 5+		取决于基础操作系统支持的 Unicode 标准版本。

从 .NET Framework 4.5 起以及在 .NET Core 和 .NET 5+ 的所有版本中，字符串比较和排序取决于操作系统。在 Windows 7 上运行的 .NET Framework 4.5 及更高版本从其自身实现 Unicode 5.0 的表中检索数据。在 Windows 8 及更高版本上运行的 .NET Framework 4.5 及更高版本从实现 Unicode 6.3 的操作系统表中检索数据。在 .NET Core 和 .NET 5+ 上，受支持的 Unicode 版本取决于基础操作系统。如果对区分区域性的已排序数据进行序列化，可使用 SortVersion 类来确定何时需要对序列化数据进行排序，使其与 .NET 和操作系统的排序顺序保持一致。有关示例，请参阅 SortVersion 类主题。

如果应用对字符串数据执行大量特定于区域性的排序，则可使用 SortKey 类来比较字符串。排序关键字反映了特定字符串的特定于区域性的排序权重，包括字母顺序、大小写和音调符号权重。由于使用排序关键字的比较为二进制，因此与显示或隐式使用 CompareInfo 对象的比较相比，这类比较速度更快。可通过将字符串传递给 CompareInfo.GetSortKey 方法为特定字符串创建区分区域性的排序关键字。

以下示例与前一个示例类似。但是此示例没有调用 Array.Sort(Array) 方法（其隐式调用了 CompareInfo.Compare 方法），而是定义了对排序关键字进行比较的 System.Collections.Generic.IComparer<T> 实现，它会进行实例化并传递到 Array.Sort<T>(T[], IComparer<T>) 方法。

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Threading;

public class SortKeyComparer : IComparer<String>
{
   public int Compare(string str1, string str2)
   {
      SortKey sk1, sk2;
      sk1 = CultureInfo.CurrentCulture.CompareInfo.GetSortKey(str1);
      sk2 = CultureInfo.CurrentCulture.CompareInfo.GetSortKey(str2);
      return SortKey.Compare(sk1, sk2);
   }
}

public class Example
{
   public static void Main()
   {
      string[] values = { "able", "ångström", "apple", "Æble",
                          "Windows", "Visual Studio" };
      SortKeyComparer comparer = new SortKeyComparer();

      // Change thread to en-US.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      // Sort the array and copy it to a new array to preserve the order.
      Array.Sort(values, comparer);
      string[] enValues = (String[]) values.Clone();

      // Change culture to Swedish (Sweden).
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("sv-SE");
      Array.Sort(values, comparer);
      string[] svValues = (String[]) values.Clone();

      // Compare the sorted arrays.
      Console.WriteLine("{0,-8} {1,-15} {2,-15}\n", "Position", "en-US", "sv-SE");
      for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
         Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues[ctr], svValues[ctr]);
   }
}
// The example displays the following output:
//       Position en-US           sv-SE
//
//       0        able            able
//       1        Æble            Æble
//       2        ångström        apple
//       3        apple           Windows
//       4        Visual Studio   Visual Studio
//       5        Windows         ångström
```

```
Imports System.Collections.Generic
Imports System.Globalization
Imports System.Threading

Public Class SortKeyComparer : Implements IComparer(Of String)
    Public Function Compare(str1 As String, str2 As String) As Integer _
           Implements IComparer(Of String).Compare
        Dim sk1, sk2 As SortKey
        sk1 = CultureInfo.CurrentCulture.CompareInfo.GetSortKey(str1)
        sk2 = CultureInfo.CurrentCulture.CompareInfo.GetSortKey(str2)
        Return SortKey.Compare(sk1, sk2)
    End Function
End Class

Module Example
    Public Sub Main()
        Dim values() As String = {"able", "ångström", "apple", _
                                  "Æble", "Windows", "Visual Studio"}
        Dim comparer As New SortKeyComparer()

        ' Change thread to en-US.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        ' Sort the array and copy it to a new array to preserve the order.
        Array.Sort(values, comparer)
        Dim enValues() As String = CType(values.Clone(), String())

        ' Change culture to Swedish (Sweden).
        Thread.CurrentThread.CurrentCulture = New CultureInfo("sv-SE")
        Array.Sort(values, comparer)
        Dim svValues() As String = CType(values.Clone(), String())

        ' Compare the sorted arrays.
        Console.WriteLine("{0,-8} {1,-15} {2,-15}", "Position", "en-US", "sv-SE")
        Console.WriteLine()
        For ctr As Integer = 0 To values.GetUpperBound(0)
            Console.WriteLine("{0,-8} {1,-15} {2,-15}", ctr, enValues(ctr), svValues(ctr))
        Next
    End Sub
End Module
' The example displays the following output:
'       Position en-US           sv-SE
'
'       0        able            able
'       1        Æble            Æble
'       2        ångström        apple
'       3        apple           Windows
'       4        Visual Studio   Visual Studio
'       5        Windows         ångström
```

避免字符串串联

如有可能，请避免使用在运行时从串联词组中生成的复合字符串。复合字符串难以本地化，因为它们往往以应用的原始语言假设语法顺序，而此顺序并不适用于其他本地化语言。

处理日期和时间

如何处理日期和时间值取决于它们是要显示在用户界面中还是保留。本节将讨论这两种用法。同时也将讨论在处理日期和时间时应如何处理时区差异和算术运算。

显示日期和时间

通常，如果日期和时间要显示在用户界面中，应使用用户区域性的格式约定，此约定是由 CultureInfo.CurrentCulture 属性以及 `CultureInfo.CurrentCulture.DateTimeFormat` 属性返回的 DateTimeFormatInfo 对象定义的。在使用以下 3 种方法之一设置日期的格式时会自动使用当前区域性的格式约

定：

- 无参数的 DateTime.ToString() 方法
- DateTime.ToString(String) 方法，其中包含一个格式字符串
- 无参数的 DateTimeOffset.ToString() 方法
- DateTimeOffset.ToString(String)，其中包含一个格式字符串
- 复合格式功能（与日期配合使用时）

以下示例显示了两次 2012 年 10 月 11 日的日出和日落数据。它首先将当前区域性设置为克罗地亚语(克罗地亚)，然后是英语(英国)。在每个用例中，日期和时间以适合当地区域性的格式显示。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   static DateTime[] dates = { new DateTime(2012, 10, 11, 7, 06, 0),
                       new DateTime(2012, 10, 11, 18, 19, 0) };

   public static void Main()
   {
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("hr-HR");
      ShowDayInfo();
      Console.WriteLine();
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
      ShowDayInfo();
   }

   private static void ShowDayInfo()
   {
      Console.WriteLine("Date: {0:D}", dates[0]);
      Console.WriteLine("   Sunrise: {0:T}", dates[0]);
      Console.WriteLine("   Sunset:  {0:T}", dates[1]);
   }
}
// The example displays the following output:
//       Date: 11. listopada 2012.
//          Sunrise: 7:06:00
//          Sunset:  18:19:00
//
//       Date: 11 October 2012
//          Sunrise: 07:06:00
//          Sunset:  18:19:00
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Dim dates() As Date = {New Date(2012, 10, 11, 7, 06, 0),
                           New Date(2012, 10, 11, 18, 19, 0)}

    Public Sub Main()
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("hr-HR")
        ShowDayInfo()
        Console.WriteLine()
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB")
        ShowDayInfo()
    End Sub

    Private Sub ShowDayInfo()
        Console.WriteLine("Date: {0:D}", dates(0))
        Console.WriteLine("   Sunrise: {0:T}", dates(0))
        Console.WriteLine("   Sunset:  {0:T}", dates(1))
    End Sub
End Module
' The example displays the following output:
'       Date: 11. listopada 2012.
'          Sunrise: 7:06:00
'          Sunset:  18:19:00
'
'       Date: 11 October 2012
'          Sunrise: 07:06:00
'          Sunset:  18:19:00
```

存留日期和时间

切勿将日期和时间数据保留为随区域性而异的格式。这是常见的编程错误，会导致数据损坏或运行时异常。以下示例使用英语(美国)区域性的格式约定将两个日期（2013 年 1 月 9 日和 2013 年 8 月 18 日）序列化为字符串。在使用英语(美国)区域性的约定检索和分析该数据时，它会成功还原。但在使用英语(英国)区域性的约定进行检索和分析时，第一个日期被错误地解释为 9 月 1 日，并且无法分析第二个日期，因为公历中没有第 18 个月。

```
using System;
using System.IO;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Persist two dates as strings.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      DateTime[] dates = { new DateTime(2013, 1, 9),
                           new DateTime(2013, 8, 18) };
      StreamWriter sw = new StreamWriter("dateData.dat");
      sw.Write("{0:d}|{1:d}", dates[0], dates[1]);
      sw.Close();

      // Read the persisted data.
      StreamReader sr = new StreamReader("dateData.dat");
      string dateData = sr.ReadToEnd();
      sr.Close();
      string[] dateStrings = dateData.Split('|');

      // Restore and display the data using the conventions of the en-US culture.
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var dateStr in dateStrings) {
         DateTime restoredDate;
         if (DateTime.TryParse(dateStr, out restoredDate))
            Console.WriteLine("The date is {0:D}", restoredDate);
         else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
      }
      Console.WriteLine();

      // Restore and display the data using the conventions of the en-GB culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var dateStr in dateStrings) {
         DateTime restoredDate;
         if (DateTime.TryParse(dateStr, out restoredDate))
            Console.WriteLine("The date is {0:D}", restoredDate);
         else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
      }
   }
}
// The example displays the following output:
//       Current Culture: English (United States)
//       The date is Wednesday, January 09, 2013
//       The date is Sunday, August 18, 2013
//
//       Current Culture: English (United Kingdom)
//       The date is 01 September 2013
//       ERROR: Unable to parse 8/18/2013
```

```
Imports System.Globalization
Imports System.IO
Imports System.Threading

Module Example
    Public Sub Main()
        ' Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim dates() As DateTime = {New DateTime(2013, 1, 9),
                                   New DateTime(2013, 8, 18)}
        Dim sw As New StreamWriter("dateData.dat")
        sw.Write("{0:d}|{1:d}", dates(0), dates(1))
        sw.Close()

        ' Read the persisted data.
        Dim sr AS New StreamReader("dateData.dat")
        Dim dateData As String = sr.ReadToEnd()
        sr.Close()
        Dim dateStrings() As String = dateData.Split("|"c)

        ' Restore and display the data using the conventions of the en-US culture.
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each dateStr In dateStrings
            Dim restoredDate As Date
            If Date.TryParse(dateStr, restoredDate) Then
                Console.WriteLine("The date is {0:D}", restoredDate)
            Else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr)
            End If
        Next
        Console.WriteLine()

        ' Restore and display the data using the conventions of the en-GB culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB")
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each dateStr In dateStrings
            Dim restoredDate As Date
            If Date.TryParse(dateStr, restoredDate) Then
                Console.WriteLine("The date is {0:D}", restoredDate)
            Else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       Current Culture: English (United States)
'       The date is Wednesday, January 09, 2013
'       The date is Sunday, August 18, 2013
'
'       Current Culture: English (United Kingdom)
'       The date is 01 September 2013
'       ERROR: Unable to parse 8/18/2013
```

可通过以下 3 种方法之一来避免此问题：

- 以二进制格式对日期和时间进行序列化，而不是序列化为字符串。
- 不考虑用户的区域性，使用同一自定义格式字符串保存和分析日期和时间的字符串表示形式。
- 使用固定区域性的格式约定保存字符串。

以下示例演示了第 3 种方法。它使用静态 CultureInfo.InvariantCulture 属性返回的固定区域性的格式约定。

```
using System;
using System.IO;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Persist two dates as strings.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      DateTime[] dates = { new DateTime(2013, 1, 9),
                           new DateTime(2013, 8, 18) };
      StreamWriter sw = new StreamWriter("dateData.dat");
      sw.Write(String.Format(CultureInfo.InvariantCulture,
                             "{0:d}|{1:d}", dates[0], dates[1]));
      sw.Close();

      // Read the persisted data.
      StreamReader sr = new StreamReader("dateData.dat");
      string dateData = sr.ReadToEnd();
      sr.Close();
      string[] dateStrings = dateData.Split('|');

      // Restore and display the data using the conventions of the en-US culture.
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var dateStr in dateStrings) {
         DateTime restoredDate;
         if (DateTime.TryParse(dateStr, CultureInfo.InvariantCulture,
                               DateTimeStyles.None, out restoredDate))
            Console.WriteLine("The date is {0:D}", restoredDate);
         else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
      }
      Console.WriteLine();

      // Restore and display the data using the conventions of the en-GB culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var dateStr in dateStrings) {
         DateTime restoredDate;
         if (DateTime.TryParse(dateStr,  CultureInfo.InvariantCulture,
                               DateTimeStyles.None, out restoredDate))
            Console.WriteLine("The date is {0:D}", restoredDate);
         else
            Console.WriteLine("ERROR: Unable to parse {0}", dateStr);
      }
   }
}
// The example displays the following output:
//       Current Culture: English (United States)
//       The date is Wednesday, January 09, 2013
//       The date is Sunday, August 18, 2013
//
//       Current Culture: English (United Kingdom)
//       The date is 09 January 2013
//       The date is 18 August 2013
```

```
Imports System.Globalization
Imports System.IO
Imports System.Threading

Module Example
    Public Sub Main()
        ' Persist two dates as strings.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim dates() As DateTime = {New DateTime(2013, 1, 9),
                                   New DateTime(2013, 8, 18)}
        Dim sw As New StreamWriter("dateData.dat")
        sw.Write(String.Format(CultureInfo.InvariantCulture,
                               "{0:d}|{1:d}", dates(0), dates(1)))
        sw.Close()

        ' Read the persisted data.
        Dim sr AS New StreamReader("dateData.dat")
        Dim dateData As String = sr.ReadToEnd()
        sr.Close()
        Dim dateStrings() As String = dateData.Split("|"c)

        ' Restore and display the data using the conventions of the en-US culture.
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each dateStr In dateStrings
            Dim restoredDate As Date
            If Date.TryParse(dateStr, CultureInfo.InvariantCulture,
                             DateTimeStyles.None, restoredDate) Then
                Console.WriteLine("The date is {0:D}", restoredDate)
            Else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr)
            End If
        Next
        Console.WriteLine()

        ' Restore and display the data using the conventions of the en-GB culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB")
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each dateStr In dateStrings
            Dim restoredDate As Date
            If Date.TryParse(dateStr, CultureInfo.InvariantCulture,
                             DateTimeStyles.None, restoredDate) Then
                Console.WriteLine("The date is {0:D}", restoredDate)
            Else
                Console.WriteLine("ERROR: Unable to parse {0}", dateStr)
            End If
        Next
    End Sub
End Module
' The example displays the following output:
'       Current Culture: English (United States)
'       The date is Wednesday, January 09, 2013
'       The date is Sunday, August 18, 2013
'
'       Current Culture: English (United Kingdom)
'       The date is 09 January 2013
'       The date is 18 August 2013
```

序列化和时区感知

一个日期和时间值可能有多个解释，从常规时间（“商店于 2013 年 1 月 2 日上午 9 点开门。”）到某个特定时刻（“出生日期：2013 年 1 月 2 日上午 6:32:00。”）。当时间值表示某个特定时刻并且将它从序列化的值中还原时，无论用户处于哪个地理位置或时区，都应确保它表示的是同一时刻。

以下示例阐释了此问题。它将一个本地日期和时间值保存为字符串，采用 3 种标准格式（"G" 表示常规日期长时

间，"s" 表示可排序日期/时间，"o" 表示往返日期/时间）以及二进制格式。

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public class Example
{
   public static void Main()
   {
      BinaryFormatter formatter = new BinaryFormatter();

      DateTime dateOriginal = new DateTime(2013, 3, 30, 18, 0, 0);
      dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local);

      // Serialize a date.
      if (! File.Exists("DateInfo.dat")) {
         StreamWriter sw = new StreamWriter("DateInfo.dat");
         sw.Write("{0:G}|{0:s}|{0:o}", dateOriginal);
         sw.Close();
         Console.WriteLine("Serialized dates to DateInfo.dat");
      }
      // Serialize the data as a binary value.
      if (! File.Exists("DateInfo.bin")) {
         FileStream fsIn = new FileStream("DateInfo.bin", FileMode.Create);
         formatter.Serialize(fsIn, dateOriginal);
         fsIn.Close();
         Console.WriteLine("Serialized date to DateInfo.bin");
      }
      Console.WriteLine();

      // Restore the date from string values.
      StreamReader sr = new StreamReader("DateInfo.dat");
      string datesToSplit = sr.ReadToEnd();
      string[] dateStrings = datesToSplit.Split('|');
      foreach (var dateStr in dateStrings) {
         DateTime newDate = DateTime.Parse(dateStr);
         Console.WriteLine("'{0}' --> {1} {2}",
                           dateStr, newDate, newDate.Kind);
      }
      Console.WriteLine();

      // Restore the date from binary data.
      FileStream fsOut = new FileStream("DateInfo.bin", FileMode.Open);
      DateTime restoredDate = (DateTime) formatter.Deserialize(fsOut);
      Console.WriteLine("{0} {1}", restoredDate, restoredDate.Kind);
   }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Module Example
    Public Sub Main()
        Dim formatter As New BinaryFormatter()

        ' Serialize a date.
        Dim dateOriginal As Date = #03/30/2013 6:00PM#
        dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local)
        ' Serialize the date in string form.
        If Not File.Exists("DateInfo.dat") Then
            Dim sw As New StreamWriter("DateInfo.dat")
            sw.Write("{0:G}|{0:s}|{0:o}", dateOriginal)
            sw.Close()
            Console.WriteLine("Serialized dates to DateInfo.dat")
        End If
        ' Serialize the date as a binary value.
        If Not File.Exists("DateInfo.bin") Then
            Dim fsIn As New FileStream("DateInfo.bin", FileMode.Create)
            formatter.Serialize(fsIn, dateOriginal)
            fsIn.Close()
            Console.WriteLine("Serialized date to DateInfo.bin")
        End If
        Console.WriteLine()

        ' Restore the date from string values.
        Dim sr As New StreamReader("DateInfo.dat")
        Dim datesToSplit As String = sr.ReadToEnd()
        Dim dateStrings() As String = datesToSplit.Split("|"c)
        For Each dateStr In dateStrings
            Dim newDate As DateTime = DateTime.Parse(dateStr)
            Console.WriteLine("'{0}' --> {1} {2}", _
                              dateStr, newDate, newDate.Kind)
        Next
        Console.WriteLine()

        ' Restore the date from binary data.
        Dim fsOut As New FileStream("DateInfo.bin", FileMode.Open)
        Dim restoredDate As Date = DirectCast(formatter.Deserialize(fsOut), DateTime)
        Console.WriteLine("{0} {1}", restoredDate, restoredDate.Kind)
    End Sub
End Module
```

当还原数据的系统与对其进行序列化的系统位于同一时区时，反序列化的日期和时间值能准确地反映原始值，输出如下：

```
'3/30/2013 6:00:00 PM' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00.0000000-07:00' --> 3/30/2013 6:00:00 PM Local

3/30/2013 6:00:00 PM Local
```

但是，如果在处于其他时区的系统上还原数据，仅格式为“o”（往返）标准格式字符串的日期和时间值会保留时区信息，因此它会表示同一时刻。当在处于罗马标准时区的系统上还原日期和时间数据时，输出如下：

```
'3/30/2013 6:00:00 PM' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00' --> 3/30/2013 6:00:00 PM Unspecified
'2013-03-30T18:00:00.0000000-07:00' --> 3/31/2013 3:00:00 AM Local

3/30/2013 6:00:00 PM Local
```

若要准确反映表示单个时刻的日期和时间值，而无需考虑对数据进行反序列化的系统所在时区，则可以执行以下任一操作：

- 使用"o"（往返）标准格式字符串将值保存为字符串。然后在目标系统上进行反序列化。
- 将其转换为 UTC，并使用"r"(RFC1123) 标准格式字符串将其保存为一个字符串。然后在目标系统上进行反序列化，并将其转换为本地时间。
- 将其转换为 UTC，并使用"u"（通用可排序）标准格式字符串将其保存为一个字符串。然后在目标系统上进行反序列化，并将其转换为本地时间。
- 将其转换为 UTC，并保存为二进制格式。然后在目标系统上进行反序列化，并将其转换为本地时间。

以下示例演示了每种方法。

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

public class Example
{
   public static void Main()
   {
      BinaryFormatter formatter = new BinaryFormatter();

      // Serialize a date.
      DateTime dateOriginal = new DateTime(2013, 3, 30, 18, 0, 0);
      dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local);

      // Serialize the date in string form.
      if (! File.Exists("DateInfo2.dat")) {
         StreamWriter sw = new StreamWriter("DateInfo2.dat");
         sw.Write("{0:o}|{1:r}|{1:u}", dateOriginal,
                                       dateOriginal.ToUniversalTime());
         sw.Close();
         Console.WriteLine("Serialized dates to DateInfo.dat");
      }
      // Serialize the date as a binary value.
      if (! File.Exists("DateInfo2.bin")) {
         FileStream fsIn = new FileStream("DateInfo2.bin", FileMode.Create);
         formatter.Serialize(fsIn, dateOriginal.ToUniversalTime());
         fsIn.Close();
         Console.WriteLine("Serialized date to DateInfo.bin");
      }
      Console.WriteLine();

      // Restore the date from string values.
      StreamReader sr = new StreamReader("DateInfo2.dat");
      string datesToSplit = sr.ReadToEnd();
      string[] dateStrings = datesToSplit.Split('|');
      for (int ctr = 0; ctr < dateStrings.Length; ctr++) {
         DateTime newDate = DateTime.Parse(dateStrings[ctr]);
         if (ctr == 1) {
            Console.WriteLine("'{0}' --> {1} {2}",
                              dateStrings[ctr], newDate, newDate.Kind);
         }
         else {
            DateTime newLocalDate = newDate.ToLocalTime();
            Console.WriteLine("'{0}' --> {1} {2}",
                              dateStrings[ctr], newLocalDate, newLocalDate.Kind);
         }
      }
      Console.WriteLine();

      // Restore the date from binary data.
      FileStream fsOut = new FileStream("DateInfo2.bin", FileMode.Open);
      DateTime restoredDate = (DateTime) formatter.Deserialize(fsOut);
      restoredDate = restoredDate.ToLocalTime();
      Console.WriteLine("{0} {1}", restoredDate, restoredDate.Kind);
   }
}
```

```
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary

Module Example
    Public Sub Main()
        Dim formatter As New BinaryFormatter()

        ' Serialize a date.
        Dim dateOriginal As Date = #03/30/2013 6:00PM#
        dateOriginal = DateTime.SpecifyKind(dateOriginal, DateTimeKind.Local)

        ' Serialize the date in string form.
        If Not File.Exists("DateInfo2.dat") Then
            Dim sw As New StreamWriter("DateInfo2.dat")
            sw.Write("{0:o}|{1:r}|{1:u}", dateOriginal, _
                                          dateOriginal.ToUniversalTime())
            sw.Close()
            Console.WriteLine("Serialized dates to DateInfo.dat")
        End If
        ' Serialize the date as a binary value.
        If Not File.Exists("DateInfo2.bin") Then
            Dim fsIn As New FileStream("DateInfo2.bin", FileMode.Create)
            formatter.Serialize(fsIn, dateOriginal.ToUniversalTime())
            fsIn.Close()
            Console.WriteLine("Serialized date to DateInfo.bin")
        End If
        Console.WriteLine()

        ' Restore the date from string values.
        Dim sr As New StreamReader("DateInfo2.dat")
        Dim datesToSplit As String = sr.ReadToEnd()
        Dim dateStrings() As String = datesToSplit.Split("|"c)
        For ctr As Integer = 0 To dateStrings.Length - 1
            Dim newDate As DateTime = DateTime.Parse(dateStrings(ctr))
            If ctr = 1 Then
                Console.WriteLine("'{0}' --> {1} {2}", _
                                  dateStrings(ctr), newDate, newDate.Kind)
            Else
                Dim newLocalDate As DateTime = newDate.ToLocalTime()
                Console.WriteLine("'{0}' --> {1} {2}", _
                                  dateStrings(ctr), newLocalDate, newLocalDate.Kind)
            End If
        Next
        Console.WriteLine()

        ' Restore the date from binary data.
        Dim fsOut As New FileStream("DateInfo2.bin", FileMode.Open)
        Dim restoredDate As Date = DirectCast(formatter.Deserialize(fsOut), DateTime)
        restoredDate = restoredDate.ToLocalTime()
        Console.WriteLine("{0} {1}", restoredDate, restoredDate.Kind)
    End Sub
End Module
```

当在位于太平洋标准时区的系统上和在位于罗马标准时区的系统上对数据进行序列化时，该示例将显示以下输出：

```
'2013-03-30T18:00:00.0000000-07:00' --> 3/31/2013 3:00:00 AM Local
'Sun, 31 Mar 2013 01:00:00 GMT' --> 3/31/2013 3:00:00 AM Local
'2013-03-31 01:00:00Z' --> 3/31/2013 3:00:00 AM Local

3/31/2013 3:00:00 AM Local
```

有关详细信息，请参阅转换时区时间。

执行日期和时间算法

DateTime 和 DateTimeOffset 类型都支持算术运算。可以计算两个日期值之差，或者将日期值与特定的时间间隔相加或相减。但是，对日期和时间值进行的算术运算时不考虑时区和时区调整规则。因此，计算表示时刻的日期和时间值可能会返回错误结果。

例如，从太平洋标准时到太平洋夏令时的转换发生在 3 月的第二个星期日，即 2013 年 3 月 10 日。如下面的示例所示，如果计算的日期和时间比太平洋标准时区系统上的 2013 年 3 月 9 日上午 10:30 晚 48 小时，2013 年 3 月 11 日上午 10:30 这一结果不会考虑干预时间调整。

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10, 30, 0),
                                            DateTimeKind.Local);
      TimeSpan interval = new TimeSpan(48, 0, 0);
      DateTime date2 = date1 + interval;
      Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                        date1, interval.TotalHours, date2);
   }
}
// The example displays the following output:
//       3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 10:30 AM
```

```
Module Example
    Public Sub Main()
        Dim date1 As Date = DateTime.SpecifyKind(#3/9/2013 10:30AM#,
                                                 DateTimeKind.Local)
        Dim interval As New TimeSpan(48, 0, 0)
        Dim date2 As Date = date1 + interval
        Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                          date1, interval.TotalHours, date2)
    End Sub
End Module
' The example displays the following output:
'       3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 10:30 AM
```

若要确保日期和时间值的算术运算生成精准的结果，请执行以下步骤：

1. 将源时区中的时间转换为 UTC。

2. 执行算术运算。

3. 如果结果为日期和时间值，则将它从 UTC 转换成源时区中的时间。

以下示例与前一个示例类似，不同的是，它按照这 3 个步骤在 2013 年 3 月 9 日上午 10 点 30 分的基础上恰当添加了 48 个小时。

```
using System;

public class Example
{
   public static void Main()
   {
      TimeZoneInfo pst = TimeZoneInfo.FindSystemTimeZoneById("Pacific Standard Time");
      DateTime date1 = DateTime.SpecifyKind(new DateTime(2013, 3, 9, 10, 30, 0),
                                            DateTimeKind.Local);
      DateTime utc1 = date1.ToUniversalTime();
      TimeSpan interval = new TimeSpan(48, 0, 0);
      DateTime utc2 = utc1 + interval;
      DateTime date2 = TimeZoneInfo.ConvertTimeFromUtc(utc2, pst);
      Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                        date1, interval.TotalHours, date2);
   }
}
// The example displays the following output:
//        3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 11:30 AM
```

```
Module Example
    Public Sub Main()
        Dim pst As TimeZoneInfo = TimeZoneInfo.FindSystemTimeZoneByID("Pacific Standard Time")
        Dim date1 As Date = DateTime.SpecifyKind(#3/9/2013 10:30AM#,
                                                 DateTimeKind.Local)
        Dim utc1 As Date = date1.ToUniversalTime()
        Dim interval As New TimeSpan(48, 0, 0)
        Dim utc2 As Date = utc1 + interval
        Dim date2 As Date = TimeZoneInfo.ConvertTimeFromUtc(utc2, pst)
        Console.WriteLine("{0:g} + {1:N1} hours = {2:g}",
                          date1, interval.TotalHours, date2)
    End Sub
End Module
' The example displays the following output:
'       3/9/2013 10:30 AM + 48.0 hours = 3/11/2013 11:30 AM
```

有关详细信息，请参阅执行日期和时间算术运算。

对日期元素使用区分区域性的名称

应用可能需要显示月份的名称或星期几。为此，常使用以下代码。

```
using System;

public class Example
{
   public static void Main()
   {
      DateTime midYear = new DateTime(2013, 7, 1);
      Console.WriteLine("{0:d} is a {1}.", midYear, GetDayName(midYear));
   }

   private static string GetDayName(DateTime date)
   {
      return date.DayOfWeek.ToString("G");
   }
}
// The example displays the following output:
//        7/1/2013 is a Monday.
```

```
Module Example
    Public Sub Main()
        Dim midYear As Date = #07/01/2013#
        Console.WriteLine("{0:d} is a {1}.", midYear, GetDayName(midYear))
    End Sub

    Private Function GetDayName(dat As Date) As String
        Return dat.DayOfWeek.ToString("G")
    End Function
End Module
' The example displays the following output:
'       7/1/2013 is a Monday.
```

但是，此代码始终以英语返回一周中某天的名称。提取月份名称的代码通常更加固定。它常常采用特定语言的月份名称来假设十二月历。

使用自定义日期和时间格式字符串或 DateTimeFormatInfo 对象的属性，可以轻松提取字符串，以反映用户区域性中的星期几或月份名称，如下面的示例所示。它将当前区域性更改为法语(法国)，并为 2013 年 7 月 1 日显示一周中某天的名称和月份的名称。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Set the current thread culture to French (France).
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");

      DateTime midYear = new DateTime(2013, 7, 1);
      Console.WriteLine("{0:d} is a {1}.", midYear, DateUtilities.GetDayName(midYear));
      Console.WriteLine("{0:d} is a {1}.", midYear, DateUtilities.GetDayName((int) midYear.DayOfWeek));
      Console.WriteLine("{0:d} is in {1}.", midYear, DateUtilities.GetMonthName(midYear));
      Console.WriteLine("{0:d} is in {1}.", midYear, DateUtilities.GetMonthName(midYear.Month));
   }
}

public static class DateUtilities
{
   public static string GetDayName(int dayOfWeek)
   {
      if (dayOfWeek < 0 | dayOfWeek > DateTimeFormatInfo.CurrentInfo.DayNames.Length)
         return String.Empty;
      else
         return DateTimeFormatInfo.CurrentInfo.DayNames[dayOfWeek];
   }

   public static string GetDayName(DateTime date)
   {
      return date.ToString("dddd");
   }

   public static string GetMonthName(int month)
   {
      if (month < 1 | month > DateTimeFormatInfo.CurrentInfo.MonthNames.Length - 1)
         return String.Empty;
      else
         return DateTimeFormatInfo.CurrentInfo.MonthNames[month - 1];
   }

   public static string GetMonthName(DateTime date)
   {
      return date.ToString("MMMM");
   }
}
// The example displays the following output:
//       01/07/2013 is a lundi.
//       01/07/2013 is a lundi.
//       01/07/2013 is in juillet.
//       01/07/2013 is in juillet.
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        ' Set the current thread culture to French (France).
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR")

        Dim midYear As Date = #07/01/2013#
        Console.WriteLine("{0:d} is a {1}.", midYear, DateUtilities.GetDayName(midYear))
        Console.WriteLine("{0:d} is a {1}.", midYear, DateUtilities.GetDayName(midYear.DayOfWeek))
        Console.WriteLine("{0:d} is in {1}.", midYear, DateUtilities.GetMonthName(midYear))
        Console.WriteLine("{0:d} is in {1}.", midYear, DateUtilities.GetMonthName(midYear.Month))
    End Sub
End Module

Public Class DateUtilities
    Public Shared Function GetDayName(dayOfWeek As Integer) As String
        If dayOfWeek < 0 Or dayOfWeek > DateTimeFormatInfo.CurrentInfo.DayNames.Length Then
            Return String.Empty
        Else
            Return DateTimeFormatInfo.CurrentInfo.DayNames(dayOfWeek)
        End If
    End Function

    Public Shared Function GetDayName(dat As Date) As String
        Return dat.ToString("dddd")
    End Function

    Public Shared Function GetMonthName(month As Integer) As String
        If month < 1 Or month > DateTimeFormatInfo.CurrentInfo.MonthNames.Length - 1 Then
            Return String.Empty
        Else
            Return DateTimeFormatInfo.CurrentInfo.MonthNames(month - 1)
        End If
    End Function

    Public Shared Function GetMonthName(dat As Date) As String
        Return dat.ToString("MMMM")
    End Function
End Class
' The example displays the following output:
'       01/07/2013 is a lundi.
'       01/07/2013 is a lundi.
'       01/07/2013 is in juillet.
'       01/07/2013 is in juillet.
```

数字值

数字的处理方式取决于它们是显示在用户界面中还是保留。本节将讨论这两种用法。

> **NOTE**
> 在分析和设置格式时，.NET 仅将 0 到 9（U+0030 到 U+0039）的基本拉丁字符识别为数字。

显示数字值

通常，如果数字显示在用户界面中，应使用用户区域性的格式约定，此约定由 CultureInfo.CurrentCulture 属性以及 `CultureInfo.CurrentCulture.NumberFormat` 属性返回的 NumberFormatInfo 对象定义。在使用以下任意方法设置日期的格式时会自动使用当前区域性的格式约定：

- 任何数值类型无参数的 `ToString` 方法

- 任何数值类型的 `ToString(String)` 方法，其中将格式字符串作为参数
- 复合格式功能（与数值配合使用时）

以下示例显示法国巴黎每月的平均气温。在显示数据之前，它首先将当前区域性设置为法语(法国)，然后再设置为英语(美国)。在每个用例中，月份名称和气温以适合当地区域性的格式显示。请注意，两个区域性使用不同的小数分隔符以分隔气温值。另请注意，该示例使用“MMMM”自定义日期和时间格式字符串以显示完整的月份名称，并且它通过确定 DateTimeFormatInfo.MonthNames 数组中最长月份名称的长度为结果字符串中的月份名称分配了足够的空间。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      DateTime dateForMonth = new DateTime(2013, 1, 1);
      double[] temperatures = {  3.4, 3.5, 7.6, 10.4, 14.5, 17.2,
                                 19.9, 18.2, 15.9, 11.3, 6.9, 5.3 };

      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");
      Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName);
      // Build the format string dynamically so we allocate enough space for the month name.
      string fmtString = "{0,-" + GetLongestMonthNameLength().ToString() + ":MMMM}     {1,4}";
      for (int ctr = 0; ctr < temperatures.Length; ctr++)
         Console.WriteLine(fmtString,
                           dateForMonth.AddMonths(ctr),
                           temperatures[ctr]);

      Console.WriteLine();

      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName);
      fmtString = "{0,-" + GetLongestMonthNameLength().ToString() + ":MMMM}     {1,4}";
      for (int ctr = 0; ctr < temperatures.Length; ctr++)
         Console.WriteLine(fmtString,
                           dateForMonth.AddMonths(ctr),
                           temperatures[ctr]);
   }

   private static int GetLongestMonthNameLength()
   {
      int length = 0;
      foreach (var nameOfMonth in DateTimeFormatInfo.CurrentInfo.MonthNames)
         if (nameOfMonth.Length > length) length = nameOfMonth.Length;

      return length;
   }
}
// The example displays the following output:
//    Current Culture: French (France)
//       janvier        3,4
//       février        3,5
//       mars           7,6
//       avril         10,4
//       mai           14,5
//       juin          17,2
//       juillet       19,9
//       août          18,2
//       septembre     15,9
//       octobre       11,3
//       novembre       6,9
//       décembre       5,3
//
```

```
//       Current Culture: English (United States)
//       January        3.4
//       February       3.5
//       March          7.6
//       April         10.4
//       May           14.5
//       June          17.2
//       July          19.9
//       August        18.2
//       September     15.9
//       October       11.3
//       November       6.9
//       December       5.3
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim dateForMonth As Date = #1/1/2013#
        Dim temperatures() As Double = {3.4, 3.5, 7.6, 10.4, 14.5, 17.2,
                                        19.9, 18.2, 15.9, 11.3, 6.9, 5.3}

        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR")
        Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName)
        Dim fmtString As String = "{0,-" + GetLongestMonthNameLength().ToString() + ":MMMM}     {1,4}"
        For ctr = 0 To temperatures.Length - 1
            Console.WriteLine(fmtstring,
                              dateForMonth.AddMonths(ctr),
                              temperatures(ctr))
        Next
        Console.WriteLine()

        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName)
        ' Build the format string dynamically so we allocate enough space for the month name.
        fmtString = "{0,-" + GetLongestMonthNameLength().ToString() + ":MMMM}     {1,4}"
        For ctr = 0 To temperatures.Length - 1
            Console.WriteLine(fmtstring,
                              dateForMonth.AddMonths(ctr),
                              temperatures(ctr))
        Next
    End Sub

    Private Function GetLongestMonthNameLength() As Integer
        Dim length As Integer
        For Each nameOfMonth In DateTimeFormatInfo.CurrentInfo.MonthNames
            If nameOfMonth.Length > length Then length = nameOfMonth.Length
        Next
        Return length
    End Function
End Module
' The example displays the following output:
'       Current Culture: French (France)
'       janvier        3,4
'       février        3,5
'       mars           7,6
'       avril         10,4
'       mai           14,5
'       juin          17,2
'       juillet       19,9
'       août          18,2
'       septembre     15,9
'       octobre       11,3
'       novembre       6,9
'       décembre       5,3
'
'       Current Culture: English (United States)
'       January        3.4
'       February       3.5
'       March          7.6
'       April         10.4
'       May           14.5
'       June          17.2
'       July          19.9
'       August        18.2
'       September     15.9
'       October       11.3
'       November       6.9
'       December       5.3
```

存留数字值

切勿将数值数据保留为特定于区域性的格式。这是常见的编程错误，会导致数据损坏或运行时异常。以下示例随机生成了十个浮点数，然后使用英语(美国)区域性的格式约定将它们序列化为字符串。在使用英语(美国)区域性的约定检索和分析该数据时，它会成功还原。但是，当使用法语(法国)区域性的约定进行检索和分析时，无法分析任何数字，因为区域性使用了不同的小数分隔符。

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Create ten random doubles.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      double[] numbers = GetRandomNumbers(10);
      DisplayRandomNumbers(numbers);

      // Persist the numbers as strings.
      StreamWriter sw = new StreamWriter("randoms.dat");
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         sw.Write("{0:R}{1}", numbers[ctr], ctr < numbers.Length - 1 ? "|" : "");

      sw.Close();

      // Read the persisted data.
      StreamReader sr = new StreamReader("randoms.dat");
      string numericData = sr.ReadToEnd();
      sr.Close();
      string[] numberStrings = numericData.Split('|');

      // Restore and display the data using the conventions of the en-US culture.
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var numberStr in numberStrings) {
         double restoredNumber;
         if (Double.TryParse(numberStr, out restoredNumber))
            Console.WriteLine(restoredNumber.ToString("R"));
         else
            Console.WriteLine("ERROR: Unable to parse '{0}'", numberStr);
      }
      Console.WriteLine();

      // Restore and display the data using the conventions of the fr-FR culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      foreach (var numberStr in numberStrings) {
         double restoredNumber;
         if (Double.TryParse(numberStr, out restoredNumber))
            Console.WriteLine(restoredNumber.ToString("R"));
         else
            Console.WriteLine("ERROR: Unable to parse '{0}'", numberStr);
      }
   }

   private static double[] GetRandomNumbers(int n)
   {
      Random rnd = new Random();
      double[] numbers = new double[n];
      for (int ctr = 0; ctr < n; ctr++)
         numbers[ctr] = rnd.NextDouble() * 1000;
      return numbers;
   }
```

```
   private static void DisplayRandomNumbers(double[] numbers)
   {
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         Console.WriteLine(numbers[ctr].ToString("R"));
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       487.0313743534644
//       674.12000879371533
//       498.72077885024288
//       42.3034229512808
//       970.57311049223563
//       531.33717716268131
//       587.82905693530529
//       562.25210175023039
//       600.7711019370571
//       299.46113717717174
//
//       Current Culture: English (United States)
//       487.0313743534644
//       674.12000879371533
//       498.72077885024288
//       42.3034229512808
//       970.57311049223563
//       531.33717716268131
//       587.82905693530529
//       562.25210175023039
//       600.7711019370571
//       299.46113717717174
//
//       Current Culture: French (France)
//       ERROR: Unable to parse '487.0313743534644'
//       ERROR: Unable to parse '674.12000879371533'
//       ERROR: Unable to parse '498.72077885024288'
//       ERROR: Unable to parse '42.3034229512808'
//       ERROR: Unable to parse '970.57311049223563'
//       ERROR: Unable to parse '531.33717716268131'
//       ERROR: Unable to parse '587.82905693530529'
//       ERROR: Unable to parse '562.25210175023039'
//       ERROR: Unable to parse '600.7711019370571'
//       ERROR: Unable to parse '299.46113717717174'
```

```
Imports System.Globalization
Imports System.IO
Imports System.Threading

Module Example
    Public Sub Main()
        ' Create ten random doubles.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim numbers() As Double = GetRandomNumbers(10)
        DisplayRandomNumbers(numbers)

        ' Persist the numbers as strings.
        Dim sw As New StreamWriter("randoms.dat")
        For ctr As Integer = 0 To numbers.Length - 1
            sw.Write("{0:R}{1}", numbers(ctr), If(ctr < numbers.Length - 1, "|", ""))
        Next
        sw.Close()

        ' Read the persisted data.
        Dim sr AS New StreamReader("randoms.dat")
        Dim numericData As String = sr.ReadToEnd()
        sr.Close()
        Dim numberStrings() As String = numericData.Split("|"c)
```

```
        ' Restore and display the data using the conventions of the en-US culture.
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each numberStr In numberStrings
            Dim restoredNumber As Double
            If Double.TryParse(numberStr, restoredNumber) Then
                Console.WriteLine(restoredNumber.ToString("R"))
            Else
                Console.WriteLine("ERROR: Unable to parse '{0}'", numberStr)
            End If
        Next
        Console.WriteLine()

        ' Restore and display the data using the conventions of the fr-FR culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR")
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        For Each numberStr In numberStrings
            Dim restoredNumber As Double
            If Double.TryParse(numberStr, restoredNumber) Then
                Console.WriteLine(restoredNumber.ToString("R"))
            Else
                Console.WriteLine("ERROR: Unable to parse '{0}'", numberStr)
            End If
        Next
    End Sub

    Private Function GetRandomNumbers(n As Integer) As Double()
        Dim rnd As New Random()
        Dim numbers(n - 1) As Double
        For ctr As Integer = 0 To n - 1
            numbers(ctr) = rnd.NextDouble * 1000
        Next
        Return numbers
    End Function

    Private Sub DisplayRandomNumbers(numbers As Double())
        For ctr As Integer = 0 To numbers.Length - 1
            Console.WriteLine(numbers(ctr).ToString("R"))
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays output like the following:
'       487.0313743534644
'       674.12000879371533
'       498.72077885024288
'       42.3034229512808
'       970.57311049223563
'       531.33717716268131
'       587.82905693530529
'       562.25210175023039
'       600.7711019370571
'       299.46113717717174
'
'       Current Culture: English (United States)
'       487.0313743534644
'       674.12000879371533
'       498.72077885024288
'       42.3034229512808
'       970.57311049223563
'       531.33717716268131
'       587.82905693530529
'       562.25210175023039
'       600.7711019370571
'       299.46113717717174
'
'       Current Culture: French (France)
```

```
'       ERROR: Unable to parse '487.0313743534644'
'       ERROR: Unable to parse '674.12000879371533'
'       ERROR: Unable to parse '498.72077885024288'
'       ERROR: Unable to parse '42.3034229512808'
'       ERROR: Unable to parse '970.57311049223563'
'       ERROR: Unable to parse '531.33717716268131'
'       ERROR: Unable to parse '587.82905693530529'
'       ERROR: Unable to parse '562.25210175023039'
'       ERROR: Unable to parse '600.7711019370571'
'       ERROR: Unable to parse '299.46113717717174'
```

若要避免此问题，可使用以下方法之一：

- 不考虑用户的区域性，使用同一自定义格式字符串保存和分析数字的字符串表示形式。
- 使用固定区域性的格式约定将数字保存为字符串，此约定是由 CultureInfo.InvariantCulture 属性返回的。
- 以二进制格式序列化数字，而不采用字符串格式。

以下示例演示了第 3 种方法。它对 Double 值的数组进行序列化，然后使用英语(美国)和法语(法国)区域性的格式约定进行反序列化并显示。

```
using System;
using System.Globalization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Create ten random doubles.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      double[] numbers = GetRandomNumbers(10);
      DisplayRandomNumbers(numbers);

      // Serialize the array.
      FileStream fsIn = new FileStream("randoms.dat", FileMode.Create);
      BinaryFormatter formatter = new BinaryFormatter();
      formatter.Serialize(fsIn, numbers);
      fsIn.Close();

      // Read the persisted data.
      FileStream fsOut = new FileStream("randoms.dat", FileMode.Open);
      double[] numbers1 = (Double[]) formatter.Deserialize(fsOut);
      fsOut.Close();

      // Display the data using the conventions of the en-US culture.
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      DisplayRandomNumbers(numbers1);

      // Display the data using the conventions of the fr-FR culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR");
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      DisplayRandomNumbers(numbers1);
   }

   private static double[] GetRandomNumbers(int n)
   {
      Random rnd = new Random();
      double[] numbers = new double[n];
      for (int ctr = 0; ctr < n; ctr++)
         numbers[ctr] = rnd.NextDouble() * 1000;
      return numbers;
```

```
      return numbers;
   }

   private static void DisplayRandomNumbers(double[] numbers)
   {
      for (int ctr = 0; ctr < numbers.Length; ctr++)
         Console.WriteLine(numbers[ctr].ToString("R"));
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       932.10070623648392
//       96.868112262742642
//       857.111520067375
//       771.37727233179726
//       262.65733840999064
//       387.00796914613244
//       557.49389788019187
//       83.79498919648816
//       957.31006048494487
//       996.54487892824454
//
//       Current Culture: English (United States)
//       932.10070623648392
//       96.868112262742642
//       857.111520067375
//       771.37727233179726
//       262.65733840999064
//       387.00796914613244
//       557.49389788019187
//       83.79498919648816
//       957.31006048494487
//       996.54487892824454
//
//       Current Culture: French (France)
//       932,10070623648392
//       96,868112262742642
//       857,111520067375
//       771,37727233179726
//       262,65733840999064
//       387,00796914613244
//       557,49389788019187
//       83,79498919648816
//       957,31006048494487
//       996,54487892824454
```

```
Imports System.Globalization
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.Threading

Module Example
    Public Sub Main()
        ' Create ten random doubles.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim numbers() As Double = GetRandomNumbers(10)
        DisplayRandomNumbers(numbers)

        ' Serialize the array.
        Dim fsIn As New FileStream("randoms.dat", FileMode.Create)
        Dim formatter As New BinaryFormatter()
        formatter.Serialize(fsIn, numbers)
        fsIn.Close()

        ' Read the persisted data.
        Dim fsOut AS New FileStream("randoms.dat", FileMode.Open)
        Dim numbers1() As Double = DirectCast(formatter.Deserialize(fsOut), Double())
        fsOut.Close()
```

```
        ' Display the data using the conventions of the en-US culture.
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        DisplayRandomNumbers(numbers1)

        ' Display the data using the conventions of the fr-FR culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("fr-FR")
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        DisplayRandomNumbers(numbers1)
    End Sub

    Private Function GetRandomNumbers(n As Integer) As Double()
        Dim rnd As New Random()
        Dim numbers(n - 1) As Double
        For ctr As Integer = 0 To n - 1
            numbers(ctr) = rnd.NextDouble * 1000
        Next
        Return numbers
    End Function

    Private Sub DisplayRandomNumbers(numbers As Double())
        For ctr As Integer = 0 To numbers.Length - 1
            Console.WriteLine(numbers(ctr).ToString("R"))
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays output like the following:
'       932.10070623648392
'       96.868112262742642
'       857.111520067375
'       771.37727233179726
'       262.65733840999064
'       387.00796914613244
'       557.49389788019187
'       83.79498919648816
'       957.31006048494487
'       996.54487892824454
'
'       Current Culture: English (United States)
'       932.10070623648392
'       96.868112262742642
'       857.111520067375
'       771.37727233179726
'       262.65733840999064
'       387.00796914613244
'       557.49389788019187
'       83.79498919648816
'       957.31006048494487
'       996.54487892824454
'
'       Current Culture: French (France)
'       932,10070623648392
'       96,868112262742642
'       857,111520067375
'       771,37727233179726
'       262,65733840999064
'       387,00796914613244
'       557,49389788019187
'       83,79498919648816
'       957,31006048494487
'       996,54487892824454
```

货币值的序列化是一种特殊情况。由于货币值取决于表示它的货币单位，因此将它视为独立的数值没有什么意义。但是如果将货币值保存为包含货币符号的格式化字符串，则无法在其默认区域性使用不同货币符号的系统

上对其进行反序列化，如下例所示。

```
using System;
using System.Globalization;
using System.IO;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Display the currency value.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      Decimal value = 16039.47m;
      Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName);
      Console.WriteLine("Currency Value: {0:C2}", value);

      // Persist the currency value as a string.
      StreamWriter sw = new StreamWriter("currency.dat");
      sw.Write(value.ToString("C2"));
      sw.Close();

      // Read the persisted data using the current culture.
      StreamReader sr = new StreamReader("currency.dat");
      string currencyData = sr.ReadToEnd();
      sr.Close();

      // Restore and display the data using the conventions of the current culture.
      Decimal restoredValue;
      if (Decimal.TryParse(currencyData, out restoredValue))
         Console.WriteLine(restoredValue.ToString("C2"));
      else
         Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData);
      Console.WriteLine();

      // Restore and display the data using the conventions of the en-GB culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
      Console.WriteLine("Current Culture: {0}",
                        Thread.CurrentThread.CurrentCulture.DisplayName);
      if (Decimal.TryParse(currencyData, NumberStyles.Currency, null, out restoredValue))
         Console.WriteLine(restoredValue.ToString("C2"));
      else
         Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData);
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       Current Culture: English (United States)
//       Currency Value: $16,039.47
//       ERROR: Unable to parse '$16,039.47'
//
//       Current Culture: English (United Kingdom)
//       ERROR: Unable to parse '$16,039.47'
```

```
Imports System.Globalization
Imports System.IO
Imports System.Threading

Module Example
    Public Sub Main()
        ' Display the currency value.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim value As Decimal = 16039.47d
        Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName)
        Console.WriteLine("Currency Value: {0:C2}", value)

        ' Persist the currency value as a string.
        Dim sw As New StreamWriter("currency.dat")
        sw.Write(value.ToString("C2"))
        sw.Close()

        ' Read the persisted data using the current culture.
        Dim sr AS New StreamReader("currency.dat")
        Dim currencyData As String = sr.ReadToEnd()
        sr.Close()

        ' Restore and display the data using the conventions of the current culture.
        Dim restoredValue As Decimal
        If Decimal.TryParse(currencyData, restoredValue) Then
            Console.WriteLine(restoredvalue.ToString("C2"))
        Else
            Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData)
        End If
        Console.WriteLine()

        ' Restore and display the data using the conventions of the en-GB culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB")
        Console.WriteLine("Current Culture: {0}",
                          Thread.CurrentThread.CurrentCulture.DisplayName)
        If Decimal.TryParse(currencyData, NumberStyles.Currency, Nothing, restoredValue) Then
            Console.WriteLine(restoredvalue.ToString("C2"))
        Else
            Console.WriteLine("ERROR: Unable to parse '{0}'", currencyData)
        End If
        Console.WriteLine()
    End Sub
End Module
' The example displays output like the following:
'       Current Culture: English (United States)
'       Currency Value: $16,039.47
'       ERROR: Unable to parse '$16,039.47'
'
'       Current Culture: English (United Kingdom)
'       ERROR: Unable to parse '$16,039.47'
```

相反，应将数值和一些区域性信息一起序列化，如区域性的名称，这样数值和其货币符号才可在当前区域性中独立地进行反序列化。以下示例通过定义带有两个参数（Decimal 值和值所属的区域性的名称）的 `CurrencyValue` 结构来实现这一点。

```
using System;
using System.Globalization;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using System.Threading;

public class Example
{
   public static void Main()
   {
      // Display the currency value.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US");
      Decimal value = 16039.47m;
      Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName);
      Console.WriteLine("Currency Value: {0:C2}", value);

      // Serialize the currency data.
      BinaryFormatter bf = new BinaryFormatter();
      FileStream fw = new FileStream("currency.dat", FileMode.Create);
      CurrencyValue data = new CurrencyValue(value, CultureInfo.CurrentCulture.Name);
      bf.Serialize(fw, data);
      fw.Close();
      Console.WriteLine();

      // Change the current thread culture.
      Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB");
      Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName);

      // Deserialize the data.
      FileStream fr = new FileStream("currency.dat", FileMode.Open);
      CurrencyValue restoredData = (CurrencyValue) bf.Deserialize(fr);
      fr.Close();

      // Display the original value.
      CultureInfo culture = CultureInfo.CreateSpecificCulture(restoredData.CultureName);
      Console.WriteLine("Currency Value: {0}", restoredData.Amount.ToString("C2", culture));
   }
}

[Serializable] internal struct CurrencyValue
{
   public CurrencyValue(Decimal amount, string name)
   {
      this.Amount = amount;
      this.CultureName = name;
   }

   public Decimal Amount;
   public string CultureName;
}
// The example displays the following output:
//       Current Culture: English (United States)
//       Currency Value: $16,039.47
//
//       Current Culture: English (United Kingdom)
//       Currency Value: $16,039.47
```

```
Imports System.Globalization
Imports System.IO
Imports System.Runtime.Serialization.Formatters.Binary
Imports System.Threading

<Serializable> Friend Structure CurrencyValue
    Public Sub New(amount As Decimal, name As String)
        Me.Amount = amount
        Me.CultureName = name
    End Sub

    Public Amount As Decimal
    Public CultureName As String
End Structure

Module Example
    Public Sub Main()
        ' Display the currency value.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-US")
        Dim value As Decimal = 16039.47d
        Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName)
        Console.WriteLine("Currency Value: {0:C2}", value)

        ' Serialize the currency data.
        Dim bf As New BinaryFormatter()
        Dim fw As New FileStream("currency.dat", FileMode.Create)
        Dim data As New CurrencyValue(value, CultureInfo.CurrentCulture.Name)
        bf.Serialize(fw, data)
        fw.Close()
        Console.WriteLine()

        ' Change the current thread culture.
        Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture("en-GB")
        Console.WriteLine("Current Culture: {0}", CultureInfo.CurrentCulture.DisplayName)

        ' Deserialize the data.
        Dim fr AS New FileStream("currency.dat", FileMode.Open)
        Dim restoredData As CurrencyValue = CType(bf.Deserialize(fr), CurrencyValue)
        fr.Close()

        ' Display the original value.
        Dim culture As CultureInfo = CultureInfo.CreateSpecificCulture(restoredData.CultureName)
        Console.WriteLine("Currency Value: {0}", restoredData.Amount.ToString("C2", culture))
    End Sub
End Module
' The example displays the following output:
'       Current Culture: English (United States)
'       Currency Value: $16,039.47
'
'       Current Culture: English (United Kingdom)
'       Currency Value: $16,039.47
```

使用特定于区域性的设置

在 .NET 中，CultureInfo 类表示特定的区域性或区域。其中一些属性返回提供有关某些区域性方面的特定信息的对象：

- CultureInfo.CompareInfo 属性返回 CompareInfo 对象，该对象包含有关如何比较区域性和排列字符串的信息。
- CultureInfo.DateTimeFormat 属性返回 DateTimeFormatInfo 对象，该对象提供用于设置日期和时间数据格式的区域性特定信息。
- CultureInfo.NumberFormat 属性返回 NumberFormatInfo 对象，该对象提供用于设置数值数据格式的区

域性特定信息。

- CultureInfo.TextInfo 属性返回 TextInfo 对象，该对象提供有关区域性写入系统的信息。

一般情况下，不要对特定的 CultureInfo 属性及其相关对象的值作出任何假设。相反，应将区域性特定的数据视为可更改的，原因如下：

- 当数据损坏、有更好的数据可用或区域性特定的约定更改时，各个属性值是可更改且可修订。
- 各个属性值在各个 .NET 版本或操作系统版本中可能会有所不同。
- .NET 支持替换区域性。由此可定义补充现有标准区域性或完全替换现有标准区域性的新的自定义区域性。
- 在 Windows 系统上，用户可使用“控制面板”中的“区域和语言”应用，自定义区域性专用设置。在实例化 CultureInfo 对象时，可调用 CultureInfo(String, Boolean) 构造函数来确定它是否反射这些用户自定义。通常，对最终用户应用而言，你应考虑用户首选项，以用户期望的格式呈现数据。

请参阅

- 全球化和本地化
- 有关使用字符串的最佳实践

.NET 全球化和 ICU

2021/11/16 •

过去，.NET 全球化 API 在不同的平台上使用不同的基础库。在 Unix 上，API 使用 Unicode 国际组件 (ICU)，在 Windows 上，API 使用 区域语言支持 (NLS)。这导致在不同平台上运行应用程序时，在少数全球化 API 中存在一些行为差异。以下方面就存在明显的行为差异：

- 区域性和区域性数据
- 字符串大小写
- 字符串排序和搜索
- 排序关键字
- 字符串规范化
- 国际化域名 (IDN) 支持
- Linux 上的时区显示名称

从 .NET 5.0 开始，开发人员可以更好地控制使用哪个基础库，从而使应用程序可以避免不同平台之间的差异。

Windows 上的 ICU

Windows 2019 年 5 月 10 日更新及更高版本将 icu.dll 作为 OS 的一部分，并且 .NET 5.0 和更高版本默认使用 ICU。在 Windows 上运行时，.NET 5.0 和更高版本尝试加载 `icu.dll`，如果此库可用，将使用它进行全球化实现。如果无法找到或无法加载 ICU 库，如在较早版本的 Windows 上运行时，.NET 5.0 和更高版本将回退到基于 NLS 的实现。

> **NOTE**
> 即使使用 ICU，`CurrentCulture`、`CurrentUICulture` 和 `CurrentRegion` 成员仍使用 Windows 操作系统 API 来遵从用户设置。

行为差异

如果你将应用升级到目标 .NET 5，即使你不知道正在使用全球化设施，你也可能会在应用中看到更改。本部分列出了你可能会看到的行为更改之一，但还有其他一些行为更改。

String.IndexOf

请考虑使用以下代码，它调用 String.IndexOf(String) 来查找字符串中的换行符索引。

```
string s = "Hello\r\nworld!";
int idx = s.IndexOf("\n");
Console.WriteLine(idx);
```

- 在 Windows 上的早期版本的 .NET 中，代码片段打印 `6`。
- 在 Windows 10 2019 年 5 月更新和更高版本上的 .NET 5.0 及更高版本中，代码片段打印 `-1`。

若要通过执行序号搜索而不是区分区域性的搜索来修复此代码，请调用 IndexOf(String, StringComparison) 重载，并传入 StringComparison.Ordinal 作为参数。

可以运行代码分析规则 CA1307：为了清晰起见，请指定 StringComparison 和 CA1309：使用序号 StringComparison 在代码中查找这些调用站点。

有关详细信息，请参阅在 .NET 5 及更高版本中比较字符串时的行为更改。

使用 NLS 而不是 ICU

使用 ICU 代替 NLS 可能会导致与一些与全球化相关的操作存在行为差异。若要恢复为使用 NLS，开发人员可以选择退出 ICU 实现。应用程序可以通过以下任意方式启用 NLS 模式：

- 在项目文件中：

```
<ItemGroup>
  <RuntimeHostConfigurationOption Include="System.Globalization.UseNls" Value="true" />
</ItemGroup>
```

- 在 `runtimeconfig.json` 文件中：

```
{
  "runtimeOptions": {
     "configProperties": {
       "System.Globalization.UseNls": true
      }
  }
}
```

- 通过将环境变量 `DOTNET_SYSTEM_GLOBALIZATION_USENLS` 设置为值 `true` 或 `1`。

> **NOTE**
> 在项目或 `runtimeconfig.json` 文件中设置的值优先于环境变量。

有关详细信息，请参阅运行时配置设置。

应用本地 ICU

每个版本的 ICU 都可能附带了 bug 修复以及描述世界语言的更新公共区域设置数据存储库 (CLDR) 数据。当涉及与全球化相关的操作时，在 ICU 版本间移动可能会对应用行为产生细微影响。为了帮助应用程序开发人员确保所有部署之间的一致性，.NET 5.0 和更高版本使 Windows 和 Unix 上的应用能够携带和使用其自己的 ICU 副本。

应用程序可以通过以下任一方式选择使用应用本地 ICU 实现模式：

- 在项目文件中：

```
<ItemGroup>
  <RuntimeHostConfigurationOption Include="System.Globalization.AppLocalIcu" Value="<suffix>:
<version> or <version>" />
</ItemGroup>
```

- 在 `runtimeconfig.json` 文件中：

```
{
  "runtimeOptions": {
     "configProperties": {
       "System.Globalization.AppLocalIcu": "<suffix>:<version> or <version>"
     }
  }
}
```

- 通过将环境变量 `DOTNET_SYSTEM_GLOBALIZATION_APPLOCALICU` 设置为值 `<suffix>:<version>` 或 `<version>`。

`<suffix>`：长度小于 36 个字符的可选后缀，遵循公共 ICU 打包约定。在生成自定义 ICU 时，可以对其自定义以生成 lib 名称并导出符号名称以包含后缀，例如 `libicuucmyapp`，其中 `myapp` 是后缀。

`<version>`：有效的 ICU 版本，例如 67.1。此版本用于加载二进制文件和获取导出的符号。

为了在设置应用本地开关时加载 ICU，.NET 使用 NativeLibrary.TryLoad 方法探测多个路径。该方法首先尝试在 `NATIVE_DLL_SEARCH_DIRECTORIES` 属性中查找库，该属性由 dotnet 主机基于应用的 `deps.json` 文件创建。有关更多信息，请参阅默认探测。

对于独立应用，用户不需要执行任何特殊操作，只需确保 ICU 位于应用目录中（对于独立应用，工作目录默认为 `NATIVE_DLL_SEARCH_DIRECTORIES`）。

如果通过 NuGet 包使用 ICU，则可在依赖框架的应用程序中使用。NuGet 解析本机资产，并将它们包含在 `deps.json` 文件和 `runtimes` 目录下应用程序的输出目录中。.NET 从这里加载它。

对于在本地版本使用 ICU 的依赖框架的应用（非独立应用），必须执行额外的步骤。.NET SDK 尚不具有用于将“松散”的本机二进制文件合并到 `deps.json` 中的功能（请参阅此 SDK 问题）。相反，你可以通过将其他信息添加到应用程序的项目文件来启用此功能。例如：

```
<ItemGroup>
  <IcuAssemblies Include="icu\*.so*" />
  <RuntimeTargetsCopyLocalItems Include="@(IcuAssemblies)" AssetType="native" CopyLocal="true"
DestinationSubDirectory="runtimes/linux-x64/native/" DestinationSubPath="%(FileName)%(Extension)"
RuntimeIdentifier="linux-x64" NuGetPackageId="System.Private.Runtime.UnicodeData" />
</ItemGroup>
```

必须为支持的运行时的所有 ICU 二进制文件执行此操作。此外，`RuntimeTargetsCopyLocalItems` 项组中的 `NuGetPackageId` 元数据需要与项目实际引用的 NuGet 包匹配。

macOS 行为

`macOS` 关于使用 `match-o` 文件中指定的加载命令解析依赖的动态库的行为与 Linux 加载程序不同。在 Linux 加载程序中，.NET 可以尝试使用 `libicudata`、`libicuuc` 和 `libicui18n`（按此顺序）来满足 ICU 依赖项关系图。但在 macOS 上，这不起作用。在 macOS 上生成 ICU 时，默认情况下，可以使用 `libicuuc` 中的这些加载命令获取动态库。以下代码片段演示了一个示例。

```
~/ % otool -L /Users/santifdezm/repos/icu-build/icu/install/lib/libicuuc.67.1.dylib
/Users/santifdezm/repos/icu-build/icu/install/lib/libicuuc.67.1.dylib:
 libicuuc.67.dylib (compatibility version 67.0.0, current version 67.1.0)
 libicudata.67.dylib (compatibility version 67.0.0, current version 67.1.0)
 /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1281.100.1)
 /usr/lib/libc++.1.dylib (compatibility version 1.0.0, current version 902.1.0)
```

这些命令只引用 ICU 其他组件的依赖库的名称。加载程序将按照 `dlopen` 约定执行搜索，这涉及到在系统目录中包含这些库，或设置 `LD_LIBRARY_PATH` env var，或在应用级目录中包含 ICU。如果无法设置 `LD_LIBRARY_PATH` 或无法确保 ICU 二进制文件位于应用级目录中，则需要执行一些额外的操作。

对于加载程序，有一些指令（如 `@loader_path`）会告知加载程序使用加载命令在与二进制文件相同的目录中搜索该依赖项。可通过两种方式实现此目的：

- `install_name_tool -change`

  运行以下命令：

```
install_name_tool -change "libicudata.67.dylib" "@loader_path/libicudata.67.dylib"
/path/to/libicuuc.67.1.dylib
install_name_tool -change "libicudata.67.dylib" "@loader_path/libicudata.67.dylib"
/path/to/libicui18n.67.1.dylib
install_name_tool -change "libicuuc.67.dylib" "@loader_path/libicuuc.67.dylib"
/path/to/libicui18n.67.1.dylib
```

- 修补 ICU 以生成含有 `@loader_path` 的安装名称

  在运行 autoconf (`./runConfigureICU`) 之前，将这些行更改为以下内容：

```
LD_SONAME = -Wl,-compatibility_version -Wl,$(SO_TARGET_VERSION_MAJOR) -Wl,-current_version -
Wl,$(SO_TARGET_VERSION) -install_name @loader_path/$(notdir $(MIDDLE_SO_TARGET))
```

WebAssembly 上的 ICU

存在专门针对 WebAssembly 工作负荷的 ICU 版本。此版本提供与桌面配置文件的全球化兼容性。若要将 ICU 数据文件大小从 24 MB 减小到 1.4 MB（如果使用 Brotli 压缩，可压缩到约 0.3 MB），则此工作负荷有少量限制。

不支持以下 API：

- CultureInfo.EnglishName
- CultureInfo.NativeName
- DateTimeFormatInfo.NativeCalendarName
- RegionInfo.NativeName

支持以下 API，但有一些限制：

- String.Normalize(NormalizationForm) 和 String.IsNormalized(NormalizationForm) 不支持很少使用的 FormKC 和 FormKD 形式。
- RegionInfo.CurrencyNativeName 返回与 RegionInfo.CurrencyEnglishName 相同的值。

此外，还可以在 dotnet/icu 存储库中找到受支持区域设置的列表。

本地化评审

2021/11/16 •

本地化分析检查是全球通用应用程序开发中的一个中间步骤。它验证全球化应用程序是否已准备好进行本地化，以及是否能够识别需要特别处理的所有代码或所有用户界面元素。此步骤还有助于确保本地化过程不会将任何功能缺陷引入应用程序。一旦本地化分析检查提出的所有问题都得到解决，就意味着可以对应用程序进行本地化了。如果本地化分析评审详尽彻底，则在本地化过程中应该不需要修改任何源代码。

本地化分析检查包括以下三项检查：

- 是否已实现全球化建议？
- 是否已正确处理区域性敏感型功能？
- 是否已使用国际数据测试应用？

实现全球化建议

如果在设计和开发应用时考虑了本地化因素，并且遵循了全球化一文中给出的建议，那么可本地化评审在很大程度上就会成为质量保证关口。否则，请在此阶段评审并实现全球化建议，修复源代码中妨碍本地化的错误。

处理区分区域性的功能

.NET 在许多方面都不提供编程支持，而且各区域性之间差别很大。大多数情况下，你必须编写自定义代码来处理诸如以下方面的功能特性：

- 地址
- 电话号码
- 纸张大小
- 用于长度、重量、面积、体积和温度的度量单位

  虽然 .NET 不对度量单位之间的转换提供内置支持，但可以使用 RegionInfo.IsMetric 属性确定特定国家或地区是否使用公制，如下面的示例所示。

```
using System;
using System.Globalization;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "en-GB", "fr-FR",
                                "ne-NP", "es-BO", "ig-NG" };
      foreach (var cultureName in cultureNames) {
         RegionInfo region = new RegionInfo(cultureName);
         Console.WriteLine("{0} {1} the metric system.", region.EnglishName,
                           region.IsMetric ? "uses" : "does not use");
      }
   }
}
// The example displays the following output:
//       United States does not use the metric system.
//       United Kingdom uses the metric system.
//       France uses the metric system.
//       Nepal uses the metric system.
//       Bolivia uses the metric system.
//       Nigeria uses the metric system.
```

```
Imports System.Globalization

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-US", "en-GB", "fr-FR",
                                        "ne-NP", "es-BO", "ig-NG"}
        For Each cultureName In cultureNames
            Dim region As New RegionInfo(cultureName)
            Console.WriteLine("{0} {1} the metric system.", region.EnglishName,
                              If(region.IsMetric, "uses", "does not use"))
        Next
    End Sub
End Module
' The example displays the following output:
'       United States does not use the metric system.
'       United Kingdom uses the metric system.
'       France uses the metric system.
'       Nepal uses the metric system.
'       Bolivia uses the metric system.
'       Nigeria uses the metric system.
```

测试应用程序

在本地化应用程序之前，应当使用国际数据在操作系统的国际版本上对其进行测试。虽然此时不会对大部分用户界面进行本地化，但可以检测到如下问题：

- 在操作系统版本之间无法正确执行反序列化的序列化数据。
- 不反映当前区域性的约定的数值数据。例如，显示的数字可能带有错误的组分隔符、小数分隔符或货币符号。
- 不反映当前区域性的约定的日期和时间数据。例如，表示月和日的数字可能会以错误的顺序出现，日期分隔符可能不正确，或者时区信息可能不正确。
- 找不到的资源，因为尚未确定应用程序的默认区域性。
- 以特定区域性中的异常顺序显示的字符串。

- 返回意外结果的字符串比较或相等比较。

如果在开发应用时遵循了全球化建议，并正确处理了区域性敏感型功能，同时还发现并解决了测试期间出现的本地化问题，就可以执行下一步本地化。

请参阅

- 全球化和本地化
- 本地化
- 全球化
- .NET 应用中的资源

.NET 中的本地化

2021/11/16 •

本地化是针对应用支持的每个区域性，将应用资源转换为本地化版本的过程。只有在完成本地化评审步骤，以验证全球化应用是否做好本地化准备后，才应继续执行本地化步骤。

可以开始进行本地化的应用程序分为两个概念块：一个是包含所有用户界面元素的块，另一个是包含可执行代码的块。用户界面块仅包含可本地化的用户界面元素，如字符串、错误消息、对话框、菜单、嵌入的对象资源等区域性中性元素。代码块仅包含所有支持的区域性要使用的应用代码。公共语言运行时支持附属程序集资源模型，用于将应用的可执行代码与资源分隔开来。若要详细了解如何实现此模型，请参阅 .NET 中的资源。

对于应用的每个本地化版本，请添加新的附属程序集，其中包含转换为目标区域性的相应语言的本地化用户界面块。所有区域性的代码块应保持不变。用户界面块的本地化版本和代码块组合生成了应用的本地化版本。

在本文中，你将了解如何使用 IStringLocalizer<T> 和 IStringLocalizerFactory 实现。本文中的所有示例源代码都依赖于 `Microsoft.Extensions.Localization` 和 `Microsoft.Extensions.Hosting` NuGet 包。有关主机的详细信息，请参阅 .NET 通用主机。

资源文件

隔离可本地化字符串的主要机制是使用资源文件。资源文件是具有 .resx 文件扩展名的 XML 文件。资源文件在执行使用应用程序之前被转换 — 换句话说，它们表示静态的已转换内容。资源文件名通常包含区域设置标识符，并采用以下格式：

```
<FullTypeName><.Locale>.resx
```

其中：

- `<FullTypeName>` 表示特定类型的可本地化资源。
- 可选 `<.Locale>` 表示资源文件内容的区域设置。

指定区域设置

区域设置至少应该定义语言，但也可以定义区域性（方言），甚至是国家/地区。这些段通常由 `-` 字符分隔。通过添加区域性特异性，在为最佳匹配项设置优先级的地方应用“区域性回退”规则。区域设置应该映射到已知语言标记。有关详细信息，请参阅 CultureInfo.Name。

区域性回退场景

假设你的本地化应用支持不同的塞尔维亚区域设置，并且其 `MessageService` 具有以下资源文件：

[1]	DIALECT	[1]/[2]
MessageService.sr-Cyrl-RS.resx	（西里尔语，塞尔维亚）	RS
MessageService.sr-Cyrl.resx	西里尔语	
MessageService.sr-Latn-BA.resx	（拉丁语，波斯尼亚和黑塞哥维那）	BA
MessageService.sr-Latn-ME.resx	（拉丁语，黑山共和国）	ME
MessageService.sr-Latn-RS.resx	（拉丁语，塞尔维亚）	RS

☐☐	DIALECT	☐☐/☐☐☐☐
MessageService.sr-Latn.resx	拉丁语	
MessageService.sr.resx	† 拉丁语	
MessageService.resx		

† *语言的默认方言。*

当应用运行时，将 CultureInfo.CurrentCulture 设置为 `"sr-Cyrl-RS"` 本地化的区域性，尝试按以下顺序解析文件：

1. *MessageService.sr-Cyrl-RS.resx*
2. *MessageService.sr-Cyrl.resx*
3. *MessageService.sr.resx*
4. *MessageService.resx*

但是，如果应用运行时，将 CultureInfo.CurrentCulture 设置为 `"sr-Latn-BA"` 本地化的区域性，则尝试按以下顺序解析文件：

1. *MessageService.sr-Latn-BA.resx*
2. *MessageService.sr-Latn.resx*
3. *MessageService.sr.resx*
4. *MessageService.resx*

如果没有相应的匹配项，"区域性回退"规则将忽略区域设置，这意味着如果找不到匹配项，则选择资源文件编号 4。如果区域性设置为 `"fr-FR"`，本地化最终会落到 MessageService.resx 文件，这会造成问题。有关详细信息，请参阅资源回退过程。

资源查找

资源文件会在查找例程中自动解析。如果项目文件名不同于项目的根命名空间，则程序集名称可能不同。这可能会阻止资源查找成功。要解决这种不匹配问题，请使用 RootNamespaceAttribute 向本地化服务提供提示。提供以后，它将在资源查找期间使用。

示例项目名为 example.csproj，它会创建 example.dll 和 example.exe —，但是会使用 `Localization.Example` 命名空间 。应用 `assembly` 级别属性来更正这种不匹配问题：

```
[assembly: RootNamespace("Localization.Example")]
```

注册本地化服务

要注册本地化服务，请在服务配置期间调用其中一个 AddLocalization 扩展方法。这将启用以下类型的依赖关系注入 (DI)：

- Microsoft.Extensions.Localization.IStringLocalizer<T>
- Microsoft.Extensions.Localization.IStringLocalizerFactory

配置本地化选项

AddLocalization(IServiceCollection, Action<LocalizationOptions>) 重载接受类型为 `Action<LocalizationOptions>` 的 `setupAction` 参数。这使你可以配置本地化选项。

```
using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddLocalization(options =>
        {
            options.ResourcesPath = "Resources";
        });
    });

// Omitted for brevity.
```

资源文件可以存在于项目中的任何位置，但是有些常见做法已经被证明是成功的。通常情况下，会选择阻力最小的路径。上述 C# 代码：

- 创建默认主机生成器。
- 使用 `IServiceCollection` 重载调用 HostBuilder.ConfigureServices。
- 将 `AddLocalization` 调用到服务集合，并将 LocalizationOptions.ResourcesPath 指定为 `"Resources"`。

这会使本地化服务在“Resources”目录中查找资源文件。

使用 `IStringLocalizer<T>` 和 `IStringLocalizerFactory`

在注册（并选择性配置）本地化服务后，可以将以下类型和 DI 结合使用：

- IStringLocalizer<T>
- IStringLocalizerFactory

要创建能够返回本地化字符串的消息服务，请考虑使用以下 `MessageService`：

```
using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example
{
    public class MessageService
    {
        private readonly IStringLocalizer<MessageService> _localizer = null!;

        public MessageService(IStringLocalizer<MessageService> localizer) =>
            _localizer = localizer;

        [return: NotNullIfNotNull("_localizer")]
        public string? GetGreetingMessage()
        {
            LocalizedString localizedString = _localizer["GreetingMessage"];
            return localizedString;
        }
    }
}
```

在前述 C# 代码中：

- 声明 `IStringLocalizer<MessageService> _localizer` 字段。
- 构造函数采用 `IStringLocalizer<MessageService>` 参数并将其分配给 `_localizer` 字段。
- `GetGreetingMessage` 方法调用 IStringLocalizer.Item[String]，将 `"GreetingMessage"` 作为参数传递。

`IStringLocalizer` 还支持参数化字符串资源，请考虑使用以下 `ParameterizedMessageService`：

```
using System;
using System.Diagnostics.CodeAnalysis;
using Microsoft.Extensions.Localization;

namespace Localization.Example
{
    public class ParameterizedMessageService
    {
        private readonly IStringLocalizer _localizer = null!;

        public ParameterizedMessageService(IStringLocalizerFactory factory) =>
            _localizer = factory.Create(typeof(ParameterizedMessageService));

        [return: NotNullIfNotNull("_localizer")]
        public string? GetFormattedMessage(DateTime dateTime, double dinnerPrice)
        {
            LocalizedString localizedString = _localizer["DinnerPriceFormat", dateTime, dinnerPrice];
            return localizedString;
        }
    }
}
```

在前述 C# 代码中：

- 声明 `IStringLocalizer _localizer` 字段。
- 构造函数采用 `IStringLocalizerFactory` 参数，该参数用于从 `ParameterizedMessageService` 类型创建 `IStringLocalizer`，并将其分配给 `_localizer` 字段。
- `GetFormattedMessage` 方法调用 IStringLocalizer.Item[String, Object[]]，将 `"DinnerPriceFormat"`（一种 `dateTime` 对象）和 `dinnerPrice` 作为参数传递。

> **IMPORTANT**
>
> `IStringLocalizerFactory` 不是必需的。而使用服务最好要求使用 IStringLocalizer<T>。

两个 IStringLocalizer.Item[] 索引器都返回 LocalizedString，它们具有向 `string?` 的隐式转换。

将其放在一起

若要举例说明使用两种消息服务以及本地化和资源文件的应用，请考虑使用以下 Program.cs 文件：

```
using System;
using System.Globalization;
using Localization.Example;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Localization;
using Microsoft.Extensions.Logging;
using static System.Console;
using static System.Text.Encoding;

[assembly: RootNamespace("Localization.Example")]

OutputEncoding = Unicode;

if (args is { Length: 1 })
{
    CultureInfo.CurrentCulture =
        CultureInfo.CurrentUICulture =
            CultureInfo.GetCultureInfo(args[0]);
}

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddLocalization();
        services.AddTransient<MessageService>();
        services.AddTransient<ParameterizedMessageService>();
    })
    .ConfigureLogging(options => options.SetMinimumLevel(LogLevel.Warning))
    .Build();

IServiceProvider services = host.Services;

ILogger logger =
    services.GetRequiredService<ILoggerFactory>()
        .CreateLogger("Localization.Example");

MessageService messageService =
    services.GetRequiredService<MessageService>();
logger.LogWarning(
    messageService.GetGreetingMessage());

ParameterizedMessageService parameterizedMessageService =
    services.GetRequiredService<ParameterizedMessageService>();
logger.LogWarning(
    parameterizedMessageService.GetFormattedMessage(
        DateTime.Today.AddDays(-3), 37.63));

await host.RunAsync();
```

在前述 C# 代码中：

- RootNamespaceAttribute 将 `"Localization.Example"` 设置为根命名空间。
- Console.OutputEncoding 分配给 Encoding.Unicode。
- 单个参数传递给 `args` 时，CultureInfo.CurrentCulture 和 CultureInfo.CurrentUICulture 分配有 `arg[0]` 给定的 CultureInfo.GetCultureInfo(String) 的结果。
- Host 使用默认值创建。
- 本地化服务 `MessageService` 和 `ParameterizedMessageService` 注册到 DI 的 `IServiceCollection`。
- 为了消除干扰，日志记录被配置为忽略低于警告的任何日志级别。
- `MessageService` 是从 `IServiceProvider` 实例解析的，其生成的消息被记录。
- `ParameterizedMessageService` 是从 `IServiceProvider` 实例解析的，其生成的已设置格式的消息被记录。

每个 `*MessageService` 类都定义一组 .resx 文件，其中每个文件都有一个条目。下面是 `MessageService` 资源文件的示例内容，从 MessageService.resx 开始：

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Hi friends, the ".NET" developer community is excited to see you here!</value>
  </data>
</root>
```

MessageService.sr-Cyrl-RS.resx:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види овде!</value>
  </data>
</root>
```

MessageService.sr-Latn.resx:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="GreetingMessage" xml:space="preserve">
    <value>Zdravo prijatelji, ".NET" developer zajednica je uzbuđena što vas vidi ovde!</value>
  </data>
</root>
```

下面是 `ParameterizedMessageService` 资源文件的示例内容，从 ParameterizedMessageService.resx 开始：

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>On {0:D} my dinner cost {1:C}.</value>
  </data>
</root>
```

ParameterizedMessageService.sr-Cyrl-RS.resx:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>У {0:D} моја вечера је коштала {1:C}.</value>
  </data>
</root>
```

ParameterizedMessageService.sr-Latn.resx:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <data name="DinnerPriceFormat" xml:space="preserve">
    <value>U {0:D} moja večera je koštala {1:C}.</value>
  </data>
</root>
```

> **TIP**
> 为简洁起见，有意省略所有资源文件 XML 注释、架构和 `<resheader>` 元素。

示例运行

以下示例运行显示给定目标区域设置的各种本地化输出。

以 `"sr-Latn"` 为例：

```
dotnet run --project .\example\example.csproj sr-Latn

warn: Localization.Example[0]
      Zdravo prijatelji, ".NET" developer zajednica je uzbuđena što vas vidi ovde!
warn: Localization.Example[0]
      U utorak, 03. avgust 2021. moja večera je koštala 37,63 ¤.
```

当省略运行项目的 .NET CLI 参数时，使用默认系统区域性 — 在本例中为 `"en-US"`：

```
dotnet run --project .\example\example.csproj

warn: Localization.Example[0]
      Hi friends, the ".NET" developer community is excited to see you here!
warn: Localization.Example[0]
      On Tuesday, August 3, 2021 my dinner cost $37.63.
```

传递 `"sr-Cryl-RS"` 时，将找到正确的相应资源文件并应用本地化：

```
dotnet run --project .\example\example.csproj sr-Cryl-RS

warn: Localization.Example[0]
      Здраво пријатељи, ".NET" девелопер заједница је узбуђена што вас види овде!
warn: Localization.Example[0]
      У уторак, 03. август 2021. моја вечера је коштала 38 RSD.
```

示例应用程序不提供 `"fr-CA"` 的资源文件，但在使用该区域性调用时，将使用非本地化的资源文件。

> **WARNING**
> 由于已找到区域性，但未找到正确的资源文件，因此应用格式设置时，最终会实现部分本地化：
>
> ```
> dotnet run --project .\example\example.csproj fr-CA
>
> warn: Localization.Example[0]
>      Hi friends, the ".NET" developer community is excited to see you here!
> warn: Localization.Example[0]
>      On mardi 3 août 2021 my dinner cost 37,63 $.
> ```

另请参阅

- 对 .NET 应用程序进行全球化和本地化
- 打包和部署 .NET 应用中的资源
- `Microsoft.Extensions.Localization`
- .NET 中的依赖关系注入
- .NET 中的日志记录

- ASP.NET Core 本地化

执行不区分区域性的字符串操作

2021/11/16 •

如果要创建旨在按区域性向用户显示结果的应用程序，则区分区域性的字符串操作无疑是一个有利条件。默认情况下，区分区域性的方法从当前线程的 CurrentCulture 属性中获得要使用的区域性。

有时候，并不需要执行区分区域性的字符串操作。当结果不应依赖于区域性时，使用区分区域性的操作可能会导致应用程序代码在遇到自定义事例映射和排序规则的区域性时失败。相关示例，请参阅使用字符串的最佳做法中的使用当前区域性的字符串比较部分。

字符串操作是否应该区分区域性取决于应用程序使用结果的方式。向用户显示结果的字符串操作通常应是区分区域性的。例如，如果应用程序在列表框中显示本地化字符串的排序列表，则应用程序应执行区分区域性的排序操作。

内部使用的字符串操作的结果通常应该是不区分区域性的。一般而言，如果应用程序使用的是不向用户显示的文件名、持久性格式或符号信息，则字符串操作的结果不应因区域性而异。例如，如果应用程序比较字符串以确定它是否是可识别的 XML 标记，则这种比较不应是区分区域性的。此外，如果安全决策基于字符串比较或大小写更改操作的结果，则操作应该不区分区域性，以确保结果不受 CurrentCulture 值的影响。

大多数默认情况下执行区分区域性的字符串操作的 .NET 方法还提供可保证获得不区分区域性结果的重载。使用 CultureInfo 参数的重载允许消除大小写映射和排序规则中的区域性差异。对于不区分区域性的字符串操作，将区域性指定为 CultureInfo.InvariantCulture。

本节内容

本节中的文章说明如何使用默认区分区域性的 .NET 方法执行不区分区域性的字符串操作。

执行不区分区域性的字符串比较
介绍了如何使用 String.Compare 和 String.CompareTo 方法执行非区域性敏感型字符串比较。

执行不区分区域性的大小写更改
介绍了如何使用 String.ToUpper、String.ToLower、Char.ToUpper 和 Char.ToLower 方法执行非区域性敏感型大小写更改。

在集合中执行不区分区域性的字符串操作
介绍了如何使用 CaseInsensitiveComparer、CaseInsensitiveHashCodeProvider 类、SortedList、ArrayList.Sort 和 CollectionsUtil.CreateCaseInsensitiveHashtable 在集合中执行非区域性敏感型操作。

在数组中执行不区分区域性的字符串操作
介绍了如何使用 Array.Sort 和 Array.BinarySearch 方法在数组中执行非区域性敏感型操作。

另请参阅

- 排序权重表(适用于 Windows 系统上的 .NET)
- 默认 Unicode 排序元素表(适用于 Linux 和 macOS 上 .NET Core)

执行不区分区域性的字符串比较

2021/11/16 •

默认情况下，String.Compare 方法执行区分区域性和区分大小写的比较。此方法还包括多个重载，这些重载提供了一个 `culture` 参数和一个 `comparisonType` 参数，前者允许你指定要使用的区域性，后者允许你指定要使用的比较规则。调用这些方法（而非调用默认重载）将消除与特定方法调用中使用的规则相关的任何歧义，并阐明某个特定比较是区分区域性的还是不区分区域性的。

> **NOTE**
>
> String.CompareTo 方法的两种重载都执行区分区域性且区分大小写的比较；你不能使用此方法来执行不区分区域性的比较。为了使代码简单明了，建议你改用 String.Compare 方法。

对于区分区域性的操作，请将 StringComparison.CurrentCulture 或 StringComparison.CurrentCultureIgnoreCase 枚举值指定为 `comparisonType` 参数。若要使用除当前区域性之外的指定区域性来执行区域性敏感型比较，请将表示相应区域性的 CultureInfo 对象指定为 `culture` 参数。

String.Compare 方法所支持的不区分区域性的字符串比较可以是语义的（基于固定区域性的排序约定）或非语义的（基于字符串中字符的序号值）。大多数不区分区域性的字符串比较是非语义的。对于这些比较，请将 StringComparison.Ordinal 或 StringComparison.OrdinalIgnoreCase 枚举值指定为 `comparisonType` 参数。例如，如果安全决策（例如，用户名或密码比较）基于字符串比较的结果，则操作应不区分区域性且是非语义的，以确保结果不受特定区域性或语言的约定的影响。

如果你希望以一致方式处理来自多个区域性的语义相关字符串，请使用不区分区域性的语义字符串比较。例如，如果你的应用程序在列表框中显示使用多个字符集的字词，则不管当前区域性如何，你可能都需要按相同的顺序来显示这些字词。对于不区分区域性的语义比较，.NET 将定义一个基于英语的语义约定的固定区域性。若要执行不区分区域性的语义比较，请将 StringComparison.InvariantCulture 或 StringComparison.InvariantCultureIgnoreCase 指定为 `comparisonType` 参数。

下面的示例将执行两个不区分区域性的非语义字符串比较。第一个比较区分大小写，而第二个比较不区分大小写。

```
using System;

public class CompareSample
{
    public static void Main()
    {
        string string1 = "file";
        string string2 = "FILE";
        int compareResult = 0;

        compareResult = String.Compare(string1, string2,
                                       StringComparison.Ordinal);
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.Ordinal, string1, string2,
                          compareResult);

        compareResult = String.Compare(string1, string2,
                                       StringComparison.OrdinalIgnoreCase);
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.OrdinalIgnoreCase, string1, string2,
                          compareResult);
    }
}
// The example displays the following output:
//    Ordinal comparison of 'file' and 'FILE': 32
//    OrdinalIgnoreCase comparison of 'file' and 'FILE': 0
```

```
Public Class CompareSample
    Public Shared Sub Main()
        Dim string1 As String = "file"
        Dim string2 As String = "FILE"
        Dim compareResult As Integer

        compareResult = String.Compare(string1, string2, _
                                       StringComparison.Ordinal)
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.Ordinal, string1, string2,
                          compareResult)

        compareResult = String.Compare(string1, string2,
                                       StringComparison.OrdinalIgnoreCase)
        Console.WriteLine("{0} comparison of '{1}' and '{2}': {3}",
                          StringComparison.OrdinalIgnoreCase, string1, string2,
                          compareResult)
    End Sub
End Class
' The example displays the following output:
'    Ordinal comparison of 'file' and 'FILE': 32
'    OrdinalIgnoreCase comparison of 'file' and 'FILE': 0
```

可以下载排序权重表，这是一组文本文件，其中包含有关 Windows 操作系统排序和比较操作中所使用的字符权重的信息，也可以下载默认 Unicode 排序元素表，这是适用于 Linux 和 macOS 的排序权重表。

另请参阅

- String.Compare
- String.CompareTo
- 执行不区分区域性的字符串操作
- 有关使用字符串的最佳实践

执行不区分区域性的大小写更改

2021/11/16 •

String.ToUpper、String.ToLower、Char.ToUpper 和 Char.ToLower 方法提供不接受任何参数的重载。默认情况下，这些不带参数的重载根据 CultureInfo.CurrentCulture 的值执行大小写更改。这样生成的结果（区分大小写）可能会因区域性而异。为了明确希望大小写更改是区域性敏感型，还是非区域性敏感型，应使用这些要求显式指定 `culture` 参数的方法重载。对于区域性敏感型大小写更改，请为 `culture` 参数指定 `CultureInfo.CurrentCulture`。对于非区域性敏感型大小写更改，请为 `culture` 参数指定 `CultureInfo.InvariantCulture`。

通常情况下，字符串会转换为标准大小写，以方便稍后查找。如果按这种方式使用字符串，应为 `culture` 参数指定 `CultureInfo.InvariantCulture`，因为 Thread.CurrentCulture 的值可能会在大小写更改和执行查找时之间变化。

如果安全决策以大小写更改操作为依据，操作应为非区域性敏感型，以确保结果不受 `CultureInfo.CurrentCulture` 值的影响。有关展示了区域性敏感型字符串操作如何产生不一致结果的示例，请参阅字符串使用最佳做法的“使用当前区域性的字符串比较”部分。

使用 String.ToUpper 和 String.ToLower 方法

为了代码清楚起见，建议始终使用 `String.ToUpper` 和 `String.ToLower` 方法重载，以便显式指定 `culture` 参数。例如，下面的代码执行标识符查找。默认情况下，`key.ToLower` 操作为区域性敏感型，但此行为并未通过读取代码明确。

示例

```
Shared Function LookupKey(key As String) As Object
   Return internalHashtable(key.ToLower())
End Function
```

```
static object LookupKey(string key)
{
    return internalHashtable[key.ToLower()];
}
```

如果希望 `key.ToLower` 操作为非区域性敏感型，应按照以下所述更改上一示例，以便在更改大小写时显式使用 `CultureInfo.InvariantCulture`。

```
Shared Function LookupKey(key As String) As Object
    Return internalHashtable(key.ToLower(CultureInfo.InvariantCulture))
End Function
```

```
static object LookupKey(string key)
{
    return internalHashtable[key.ToLower(CultureInfo.InvariantCulture)];
}
```

使用 Char.ToUpper 和 Char.ToLower 方法

尽管 `Char.ToUpper` 和 `Char.ToLower` 方法的特性与 `String.ToUpper` 和 `String.ToLower` 方法相同，但受影响的区

域性只有土尔其语(土尔其)和阿塞拜疆语(拉丁，阿塞拜疆)。这些是唯一两个存在单字符大小写差异的区域性。有关此唯一大小写映射的详细信息，请参阅 String 类主题中的"大小写"部分。为了代码清楚起见，并确保结果一致，建议始终使用这些方法重载，以便显式指定 `culture` 参数。

另请参阅

- String.ToUpper
- String.ToLower
- Char.ToUpper
- Char.ToLower
- 执行不区分区域性的字符串操作

在集合中执行不区分区域性的字符串运算

2021/11/16 •

System.Collections 命名空间中包含默认提供区域性敏感型行为的类和成员。CaseInsensitiveComparer 和 CaseInsensitiveHashCodeProvider 类的无参数构造函数使用 Thread.CurrentCulture 属性初始化新实例。默认情况下，CollectionsUtil.CreateCaseInsensitiveHashtable 方法的所有重载都会使用 `Thread.CurrentCulture` 属性新建 Hashtable 类的实例。默认情况下，ArrayList.Sort 方法重载使用 `Thread.CurrentCulture` 执行区域性敏感型排序。将字符串用作键时，SortedList 中的排序和查找可能会受 `Thread.CurrentCulture` 影响。请按本节提供的用法建议，在 `Collections` 命名空间中的这些类和方法中获取不区分区域性的结果。

> **NOTE**
>
> 向比较方法传递 CultureInfo.InvariantCulture 确实会执行非区域性敏感型比较。但是，这不会导致对文件路径、注册表项、环境变量等进行非语义比较。也不支持基于比较结果的安全决策。若要进行非语义比较或支持基于结果的安全决策，应用应使用接受 StringComparison 值的比较方法。然后，应用应传递 StringComparison。

请使用 `CaseInsensitiveComparer` 和 `CaseInsensitiveHashCodeProvider` 类

`CaseInsensitiveHashCodeProvider` 和 `CaseInsensitiveComparer` 的无参数构造函数使用 `Thread.CurrentCulture` 来初始化类的新实例，产生区域性敏感行为。下面的代码示例演示区域性敏感 `Hashtable` 的构造函数，因为该构造函数使用 `CaseInsensitiveHashCodeProvider` 和 `CaseInsensitiveComparer` 的无参数构造函数。

```
internalHashtable = New Hashtable(CaseInsensitiveHashCodeProvider.Default, CaseInsensitiveComparer.Default)
```

```
internalHashtable = new Hashtable(CaseInsensitiveHashCodeProvider.Default, CaseInsensitiveComparer.Default);
```

若要使用 `CaseInsensitiveComparer` 和 `CaseInsensitiveHashCodeProvider` 类创建非区域性敏感型 `Hashtable`，请使用接受 `culture` 参数的构造函数，初始化这些类的新实例。对于 `culture` 参数，请指定 CultureInfo.InvariantCulture。下面的代码示例演示不区分区域性的 `Hashtable` 的构造函数。

```
internalHashtable = New Hashtable(New
    CaseInsensitiveHashCodeProvider(CultureInfo.InvariantCulture),
    New CaseInsensitiveComparer(CultureInfo.InvariantCulture))
```

```
internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider
    (CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

使用 `CollectionsUtil.CreateCaseInsensitiveHashTable` 方法

若要为 `Hashtable` 类创建一个忽略字符串大小写的新实例，`CollectionsUtil.CreateCaseInsensitiveHashTable` 方法是一种十分有用的快捷方式。但是，`CollectionsUtil.CreateCaseInsensitiveHashTable` 方法的所有重载都区分区域性，因为这些重载使用 `Thread.CurrentCulture` 属性。不能使用此方法创建不区分区域性的 `Hashtable`。若要创建不区分区域性的 `Hashtable`，请使用接受 `culture` 参数的 `Hashtable` 构造函数。对于 `culture` 参数，请指定 `CultureInfo.InvariantCulture`。下面的代码示例演示不区分区域性的 `Hashtable` 的构造函数。

```
internalHashtable = New Hashtable(New
    CaseInsensitiveHashCodeProvider(CultureInfo.InvariantCulture),
    New CaseInsensitiveComparer(CultureInfo.InvariantCulture))
```

```
internalHashtable = new Hashtable(new CaseInsensitiveHashCodeProvider
    (CultureInfo.InvariantCulture),
    new CaseInsensitiveComparer(CultureInfo.InvariantCulture));
```

使用 `SortedList` 类

`SortedList` 表示键值对的集合，这些键值对按键排序，并可按照键和索引进行访问。在使用以字符串作为键的 `SortedList` 时，排序和查找会受 `Thread.CurrentCulture` 属性的影响。若要从 `SortedList` 获取不区分区域性的行为，请使用一个接受 `comparer` 参数的构造函数来创建 `SortedList`。`comparer` 参数指定要在比较键时使用的 IComparer 实现。对于该参数，请指定使用 `CultureInfo.InvariantCulture` 来比较键的自定义比较器类。下面的示例说明一个不区分区域性的自定义比较器类，可将该比较器类指定为 `SortedList` 构造函数的 `comparer` 参数。

```
Imports System.Collections
Imports System.Globalization

Friend Class InvariantComparer
    Implements IComparer
    Private m_compareInfo As CompareInfo
    Friend Shared [Default] As New InvariantComparer()

    Friend Sub New()
        m_compareInfo = CultureInfo.InvariantCulture.CompareInfo
    End Sub

    Public Function Compare(a As Object, b As Object) As Integer _
            Implements IComparer.Compare
        Dim sa As String = CType(a, String)
        Dim sb As String = CType(b, String)
        If Not (sa Is Nothing) And Not (sb Is Nothing) Then
            Return m_compareInfo.Compare(sa, sb)
        Else
            Return Comparer.Default.Compare(a, b)
        End If
    End Function
End Class
```

```
using System;
using System.Collections;
using System.Globalization;

internal class InvariantComparer : IComparer
{
    private CompareInfo _compareInfo;
    internal static readonly InvariantComparer Default = new
        InvariantComparer();

    internal InvariantComparer()
    {
        _compareInfo = CultureInfo.InvariantCulture.CompareInfo;
    }

    public int Compare(Object a, Object b)
    {
        if (a is string sa && b is string sb)
            return _compareInfo.Compare(sa, sb);
        else
            return Comparer.Default.Compare(a,b);
    }
}
```

一般而言，如果对字符串使用 `SortedList` 而不指定自定义固定比较器，填充列表后，更改 `Thread.CurrentCulture` 会使列表失效。

使用 `ArrayList.Sort` 方法

默认情况下，`ArrayList.Sort` 方法的重载使用 `Thread.CurrentCulture` 属性来执行区分区域性的排序。由于排序顺序不同，结果可能会因区域性而异。若要消除区分区域性的行为，请使用接受 `IComparer` 实现的此方法的重载。对于 `comparer` 参数，请指定使用 `CultureInfo.InvariantCulture` 的自定义固定比较器类。使用 SortedList 类主题中提供了自定义固定比较器类的示例。

另请参阅

- CaseInsensitiveComparer
- CaseInsensitiveHashCodeProvider
- ArrayList.Sort
- SortedList
- Hashtable
- IComparer
- 执行不区分区域性的字符串操作
- CollectionsUtil.CreateCaseInsensitiveHashtable

在数组中执行不区分区域性的字符串运算

2021/11/16 •

默认情况下，Array.Sort 和 Array.BinarySearch 方法重载使用 Thread.CurrentCulture 属性执行区域性敏感型排序。由于排序顺序不同，因此这些方法返回的区域性敏感型结果可能会因区域性而异。若要消除区域性敏感型行为，请使用需要使用 `comparer` 参数的此方法重载之一。`comparer` 参数指定要在比较数组元素时使用的 IComparer 实现。对于参数，指定使用 CultureInfo.InvariantCulture 的自定义固定比较器类。在集合中执行非区域性敏感型字符串运算主题的"使用 SortedList 类"子主题提供了自定义固定比较器类的示例。

> **NOTE**
>
> 向比较方法传递 CultureInfo.InvariantCulture 确实会执行非区域性敏感型比较。但是，这不会导致对文件路径、注册表项、环境变量等进行非语义比较。也不支持基于比较结果的安全决策。若要进行非语义比较或支持基于结果的安全决策，应用应使用接受 StringComparison 值的比较方法。然后，应用应传递 Ordinal。

另请参阅

- Array.Sort
- Array.BinarySearch
- IComparer
- 执行不区分区域性的字符串操作

开发全球通用应用程序的最佳做法

2021/11/16 •

本节描述在开发全球通用的应用程序时应遵循的最佳做法。

全球化最佳做法

1. 在内部使应用程序代码成为 Unicode。
2. 使用 System.Globalization 命名空间提供的区域性识别类来操作和格式化数据。
   - 对于排序，使用 SortKey 类和 CompareInfo 类。
   - 对于字符串比较，使用 CompareInfo 类。
   - 对于日期和时间格式化，使用 DateTimeFormatInfo 类。
   - 对于数字格式化，使用 NumberFormatInfo 类。
   - 对于公历和非公历，使用 Calendar 类或特定的日历实现之一。
3. 在适当的情况下，使用 System.Globalization.CultureInfo 类提供的区域性属性设置。使用 CultureInfo.CurrentCulture 属性来执行格式化任务，如日期和时间或数字的格式化。使用 CultureInfo.CurrentUICulture 属性来检索资源。请注意，`CurrentCulture` 和 `CurrentUICulture` 属性可以通过线程进行设置。
4. 通过使用 System.Text 命名空间中的编码类，使应用程序能够与各种编码相互进行数据读写。不要采用 ASCII 数据。假定在用户可以输入文本的任何位置都将提供国际字符。例如，应用程序应接受服务器名、目录、文件名、用户名和 URL 中的国际字符。
5. 使用 UTF8Encoding 类时，出于安全原因，应使用此类提供的错误检测功能。为了打开错误检测功能，请使用具有一个 `throwOnInvalidBytes` 参数的构造函数来创建该类的一个实例，并将该参数的值设置为 `true`。
6. 尽可能将字符串按整个字符串处理，而不是按一系列个别字符处理。这在排序或搜索子字符串时尤为重要。这可以防止与分析组合字符有关的问题。还可以通过 System.Globalization.StringInfo 类使用文本单元而不是单个字符。
7. 使用 System.Drawing 命名空间提供的类来显示文本。
8. 为保持操作系统间的一致性，不要允许用户设置重写 CultureInfo。使用接受 `CultureInfo` 参数的 `useUserOverride` 构造函数并将此参数设置为 `false`。
9. 在国际操作系统版本上使用国际数据来测试应用程序功能。
10. 如果安全决策基于字符串比较或大小写更改操作的结果，请使用不区分区域性的字符串操作。这种做法可确保结果不会受 `CultureInfo.CurrentCulture` 值的影响。有关展示了区分区域性字符串比较如何产生不一致结果的示例，请参阅字符串使用最佳做法的“使用当前区域性的字符串比较”部分。

本地化最佳做法

1. 将所有可本地化的资源移动到单独的纯资源 DLL 中。可本地化的资源包括用户界面元素，如字符串、错误消息、对话框、菜单以及嵌入的对象资源。
2. 不要对字符串或用户界面资源进行硬编码。

3. 不要将不可本地化的资源放在纯资源 DLL 中。否则会使翻译人员产生困惑。
4. 不要使用在运行时从串联词组生成的复合字符串。复合字符串难以本地化，因为它们往往采用英语语法顺序，而此顺序并不适用于所有语言。
5. 避免不明确的结构，如“Empty Folder”，因为根据字符串组成部分的语法规则，这些字符串可能产生不同的翻译。例如，“empty”既可以是一个动词，也可以是一个形容词，因此在诸如意大利语或法语等语言中就可能导致不同的翻译。
6. 避免在应用程序中使用包含文本的图像和图标。本地化这些图像和图标的成本是很大的。
7. 允许在用户界面中为字符串长度的扩展保留足够的空间。在某些语言中，词组所需的空间可能比在其他语言中多 50-75%。
8. 使用 System.Resources.ResourceManager 类来根据区域性检索资源。
9. 使用 Visual Studio 创建 Windows 窗体对话框，以便可以使用 Windows 窗体资源编辑器 (Winres.exe) 对它们进行本地化。不要对 Windows 窗体对话框进行手动编码。
10. 安排进行专业本地化工作（翻译）。
11. 有关创建并本地化资源的完整说明，请参阅.NET 应用中的资源。

ASP.NET 应用程序的全球化最佳做法

1. 在应用程序中显式设置 CurrentUICulture 和 CurrentCulture 属性。不要依赖于默认设置。
2. 请注意，ASP.NET 应用程序是托管应用程序，因此可以使用与其他托管应用程序相同的类，以根据区域性检索、显示和操作信息。
3. 注意在 ASP.NET 中可以指定以下三种编码类型：
   - requestEncoding 指定从客户端浏览器接收的编码。
   - responseEncoding 指定要发送到客户端浏览器的编码。在大多数情形下，此编码应该与为 requestEncoding 指定的编码相同。
   - fileEncoding 指定用于 .aspx、.asmx 和 .asax 文件分析的默认编码。
4. 在 ASP.NET 应用程序中的以下三个位置指定 requestEncoding、responseEncoding、fileEncoding、culture 和 uiCulture 特性的值：
   - 在 Web.config 文件的全球化一节中。此文件是 ASP.NET 应用程序的外部文件。有关详细信息，请参阅 <globalization> 元素。
   - 在页面指令中。请注意，当应用程序在页面中时，文件已经被读取。因此，指定 fileEncoding 和 requestEncoding 为时已晚。只有 uiCulture、Culture 和 responseEncoding 可以在页面指令中指定。
   - 在应用程序代码中以编程方式指定。该设置可能随请求的不同而不同。同页面指令一样，到打开应用程序代码时，指定 fileEncoding 和 requestEncoding 为时已晚。只有 uiCulture、Culture 和 responseEncoding 可以在应用程序代码中指定。
5. 请注意，uiCulture 值可以设置为浏览器接受的语言。

请参阅

- 全球化和本地化
- .NET 应用中的资源

.NET 应用中的资源

2021/11/16 •

几乎每一个生产性应用都需要使用资源。资源是随应用以逻辑方式部署的任何不可执行的数据。资源可以在应用中作为错误消息显示，或者作为用户界面的一部分显示。资源可以包含多种形式的数据，包括字符串、图像和持久的对象。（持久化对象必须是可序列化的，才能将这些对象写入到资源文件。）通过在资源文件中存储数据，可以更改这些数据，而无需重新编译整个应用。还可以将数据存储在一个位置，而无需依赖存储在多个位置的硬编码数据。

.NET 为资源的创建和本地化提供全面的支持。此外，.NET 还支持一种用于打包和部署本地化资源的简单模型。

创建和本地化资源

在非本地化的应用中，可以使用资源文件作为应用数据的存储库，特别用于存储本来可能在源代码中的多个位置为硬编码的字符串。通常以文本 (.txt) 或 XML (.resx) 文件形式创建资源，并使用 Resgen.exe（资源文件生成器）将其编译为二进制 .resources 文件。随后，这些文件可由语言编译器嵌入到应用的可执行文件中。有关创建资源的详细信息，请参阅创建资源文件。

您还可以按特定的区域性对应用的资源进行本地化。这样可以生成应用的本地化（翻译）版本。在开发使用本地化资源的应用时，可以指定一个区域性作为非特定或回退区域性，以在没有合适的资源可用时使用该区域性的资源。通常，非特定区域性的资源存储在应用的可执行文件中。其余各本地化区域性的资源存储在单独的附属程序集中。有关详细信息，请参阅创建附属程序集。

打包和部署资源

本地化的应用资源部署在附属程序集中。附属程序集包含单个区域性的资源；它不包含任何应用代码。在附属程序集部署模型中，所创建的应用包含一个默认程序集（通常是主程序集），对于该应用支持的每种区域性，还包含一个附属程序集。因为附属程序集不是主程序集的一部分，所以您不必替换该应用的主程序集，即可很容易地替换或更新与特定区域性相对应的资源。

在确定哪些资源将构成应用的默认资源程序集时要谨慎。因为默认资源程序集是主程序集的一部分，所以对它做任何更改都会要求你替换主程序集。如果没有提供默认资源，则在资源回退进程尝试查找默认资源时会引发异常。在设计良好的应用中，使用资源应永远不会引发异常。

有关详细信息，请参阅打包和部署资源一文。

检索资源

在运行时，应用会基于 CultureInfo.CurrentUICulture 属性指定的区域性为每个线程加载相应的本地化资源。此属性的值按如下方式派生：

- 向 CultureInfo 属性直接分配一个表示本地化区域性的 Thread.CurrentUICulture 对象。
- 如果没有明确分配区域性，则从 CultureInfo.DefaultThreadCurrentUICulture 属性检索默认线程 UI 区域性。
- 如果没有明确分配默认线程 UI 区域性，则在本地计算机上检索当前用户的区域性。在 Windows 上运行的 .NET 实现通过调用 Windows `GetUserDefaultUILanguage` 函数来完成此操作。

有关如何设置当前 UI 区域性的详细信息，请参阅 CultureInfo 和 CultureInfo.CurrentUICulture 参考页。

随后，通过使用 System.Resources.ResourceManager 类，可以为当前的 UI 区域性或特定区域性检索资源。虽然 ResourceManager 类最常用于检索资源，但 System.Resources 命名空间包含可用于检索资源的其他类型。这些

方法包括：

- ResourceReader 类 - 可用于枚举嵌入在程序集或存储于独立二进制 .resources 文件中的资源。当您不知道运行时可用资源的准确名称时，这将会很有用。
- ResXResourceReader 类 - 可用于从 XML (.resx) 文件中检索资源。
- ResourceSet 类 - 可用于检索特定区域性的资源，而不遵循回退规则。资源可以存储在程序集或独立的二进制 .resources 文件中。您还可以开发 IResourceReader 实现以使用 ResourceSet 类从某个其他源中检索资源。
- ResXResourceSet 类 - 可用于将 XML 资源文件中的所有项目都检索到内存中。

请参阅

- CultureInfo
- CultureInfo.CurrentUICulture
- 创建资源文件
- 打包和部署资源
- 创建附属程序集
- 检索资源
- .NET 中的本地化

为 .NET 应用创建资源文件

2021/11/16 •

可以将字符串、图像或对象数据等资源包含在资源文件中，方便应用程序使用。.NET Framework 提供了五种创建资源文件的方法：

- 创建一个包含字符串资源的文本文件。可以使用资源文件生成器 (Resgen.exe) 将文本文件转换成二进制资源 (.resources) 文件。然后使用语言编译器将这个二进制资源文件嵌入可执行应用程序或应用程序库，或者使用程序集链接器 (Al.exe) 将这个二进制资源文件嵌入附属程序集。有关详细信息，请参阅文本文件中的资源部分。
- 创建一个包含字符串、图像或对象数据的 XML 资源 (.resx) 文件。可以使用资源文件生成器 (Resgen.exe) 将 .resx 文件转换成二进制资源 (.resources) 文件。然后使用语言编译器将这个二进制资源文件嵌入可执行应用程序或应用程序库，或者使用程序集链接器 (Al.exe) 将这个二进制资源文件嵌入附属程序集。有关详细信息，请参阅 .resx 文件中的资源部分。
- 使用 System.Resources 命名空间中的类型以编程方式创建一个 XML 资源 (.resx) 文件。可以创建一个 .resx 文件、枚举其资源并按名称检索特定资源。有关详细信息，请参阅以编程方式使用 .resx 文件。
- 以编程方式创建一个二进制资源 (.resources) 文件。然后使用语言编译器将该文件嵌入可执行应用程序或应用程序库，或者使用程序集链接器 (Al.exe) 将这个二进制资源文件嵌入附属程序集。有关详细信息，请参阅 .resources 文件中的资源部分。
- 使用 Visual Studio 创建一个资源文件并将其包含在项目中。Visual Studio 提供一个资源编辑器，借助该编辑器，可添加、删除和修改资源。编译时，资源文件会自动转换成二进制 .resources 文件，并嵌入应用程序程序集或附属程序集中。有关详细信息，请参阅 Visual Studio 中的资源文件部分。

文本文件中的资源

文本(.txt 或 .restext)文件只能用于存储字符串资源。对于非字符串资源，使用 .resx 文件或以编程方式创建它们。包含字符串资源的文本文件使用以下格式：

```
# This is an optional comment.
name = value

; This is another optional comment.
name = value

; The following supports conditional compilation if X is defined.
#ifdef X
name1=value1
name2=value2
#endif

# The following supports conditional compilation if Y is undefined.
#if !Y
name1=value1
name2=value2
#endif
```

.txt 和 .restext 文件的资源文件格式是相同的。.restext 文件扩展名仅用于明确区分文本文件和基于文本的资源文件。

字符串资源显示为名称/值对，其中名称是标识资源的字符串，值是在将名称传递给资源检索方法(例如

ResourceManager.GetString)时返回的资源字符串。名称和值必须用等号 (=) 分隔开。例如:

```
FileMenuName=File
EditMenuName=Edit
ViewMenuName=View
HelpMenuName=Help
```

Caution

请勿使用资源文件存储密码、安全敏感信息或私人数据。

在文本文件中,允许存在空字符串(即值为 String.Empty 的资源)。例如:

```
EmptyString=
```

从 .NET Framework 4.5 开始并在所有版本的 .NET Core 中,文本文件支持使用 `#ifdef` symbol ... `#endif` 和 `#if !` symbol ... `#endif` 构造的传统编译。还可以通过资源文件生成器 (resgen.exe) 使用 `/define` 开关来定义符号。每个资源都需要其自己的 `#ifdef` symbol... `#endif` 或 `#if !` symbol... `#endif` 构造。如果使用的是 `#ifdef` 语句,并且定义了 symbol,则关联的资源将包括在 .resources 文件中;否则,将不包括此资源。如果使用的是 `#if !` 语句,并且没有定义 symbol,则关联的资源将包括在 .resources 文件中;否则,将不包括此资源。

注释在文本文件中为可选项,并且在每行开头使用分号 (;) 或者井号 (#) 开头。包含注释的行可位于文件中的任何位置。使用资源文件生成器 (Resgen.exe) 创建的已编译 .resources 文件中不包含注释。

文本文件中的所有空白行将视为空格并忽略。

下面的示例定义名为 `OKButton` 和 `CancelButton` 的两个字符串资源。

```
#Define resources for buttons in the user interface.
OKButton=OK
CancelButton=Cancel
```

如果文本文件包含名称的重复匹配项,资源文件生成器 (Resgen.exe) 将显示一条警告,并忽略第二个名称。

值不能包含换行符,但可以使用 C 语言样式的转义符,例如使用 `\n` 表示新行,使用 `\t` 表示制表符。还可以使用经过转义的反斜杠字符(例如,"\\")。此外,允许使用空字符串。

使用 little-endian 或 big-endian 字节顺序的 UTF-8 编码或 UTF-16 编码以文本文件格式保存资源。但是在默认情况下,将 .txt 文件转换为 .resources 文件的资源文件生成器 (Resgen.exe) 会将文件默认视为 UTF-8 文件。如果希望 Resgen.exe 识别使用 UTF-16 编码的文件,必须在文件开头包含 Unicode 字节顺序标记 (U + FEFF)。

若要将文本格式的资源文件嵌入到 .NET 程序集,必须使用资源文件生成器 (Resgen.exe) 将文件转换为二进制资源 (.resources) 文件。然后可使用语言编译器将 .resources 文件嵌入 .NET 程序集,或者使用程序集链接器 (Al.exe) 将其嵌入附属程序集。

下面的示例在简单的"Hello World"控制台应用程序中使用了一个名为 GreetingResources.txt 的文本格式的资源文件。该文本文件定义了 `prompt` 和 `greeting` 两个字符串,分别用于提示用户输入其名字和显示一条问候。

```
# GreetingResources.txt
# A resource file in text format for a "Hello World" application.
#
# Initial prompt to the user.
prompt=Enter your name:
# Format string to display the result.
greeting=Hello, {0}!
```

使用以下命令可以将该文本文件转换成 .resources 文件:

```
resgen GreetingResources.txt
```

下面的示例展示了某控制台应用程序中的源代码，该控制台应用程序使用 .resources 文件向用户显示信息。

```
using System;
using System.Reflection;
using System.Resources;

public class Example
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("GreetingResources",
                               typeof(Example).Assembly);
      Console.Write(rm.GetString("prompt"));
      string name = Console.ReadLine();
      Console.WriteLine(rm.GetString("greeting"), name);
   }
}
// The example displays output like the following:
//       Enter your name: Wilberforce
//       Hello, Wilberforce!
```

```
Imports System.Reflection
Imports System.Resources

Module Example
    Public Sub Main()
        Dim rm As New ResourceManager("GreetingResources",
                                      GetType(Example).Assembly())
        Console.Write(rm.GetString("prompt"))
        Dim name As String = Console.ReadLine()
        Console.WriteLine(rm.GetString("greeting"), name)
    End Sub
End Module
' The example displays output like the following:
'       Enter your name: Wilberforce
'       Hello, Wilberforce!
```

如果使用的是 Visual Basic，且源代码文件名为 Greeting.vb，以下命令将创建一个包含嵌入的 .resources 文件的可执行文件：

```
vbc greeting.vb -resource:GreetingResources.resources
```

如果使用的是 C#，并且将源代码文件命名为 Greeting.cs，以下命令将创建一个包含嵌入 .resources 文件的可执行文件：

```
csc greeting.cs -resource:GreetingResources.resources
```

.resx 文件中的资源

与只能存储字符串资源的文本文件不同，XML 资源 (.resx) 文件可以存储字符串、二进制数据（图像、图标和音频剪辑等）以及编程对象。.resx 文件包含一个标准标头，用以描述资源条目的格式，并指定用于解析数据的 XML 的版本信息。资源文件数据跟在 XML 标头之后。每个数据项由包含在 `data` 标记中的一个名称/值对构成。其 `name` 属性定义资源名称，而嵌套的 `value` 标记包含资源值。对于字符串数据，`value` 标记包含字符串。

例如，以下 `data` 标记定义了一个名为 `prompt` 且值为“Enter your name”的字符串资源。

```
<data name="prompt" xml:space="preserve">
  <value>Enter your name:</value>
</data>
```

> **WARNING**
> 请勿使用资源文件存储密码、安全敏感信息或私人数据。

对于资源对象，data 标记中包含了 `type` 属性，用于指示资源的数据类型 。对于由二进制数据构成的对象，`data` 标记还包含 `mimetype` 属性，用以指示二进制数据的 `base64` 类型。

> **NOTE**
> 所有的 .resx 文件都使用二进制序列化格式化程序来生成和分析指定类型的二进制数据。因此如果对象的二进制序列化格式以不兼容的方式发生更改，.resx 文件可能失效。

以下示例显示了部分包含 Int32 资源和位图图像的 .resx 文件。

```
<data name="i1" type="System.Int32, mscorlib">
  <value>20</value>
</data>

<data name="flag" type="System.Drawing.Bitmap, System.Drawing,
    Version=1.0.5000.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    mimetype="application/x-microsoft.net.object.bytearray.base64">
  <value>
    AAEAAAD/////AQAAAAAAAAAMAgAAADtTeX…
  </value>
</data>
```

> **IMPORTANT**
> 由于 .resx 文件必须由采用预定义格式的格式标准的 XML 构成，不建议手动使用 .resx 文件，尤其是当 .resx 文件包含非字符串资源时。相反，Visual Studio 提供一个用于创建和操作 .resx 文件的透明接口。有关详细信息，请参阅 Visual Studio 中的资源文件部分。还可以通过编程方式创建和操作 .resx 文件。有关详细信息，请参阅以编程方式使用 .resx 文件。

.resources 文件中的资源

可以使用 System.Resources.ResourceWriter 类以编程方式从代码中直接创建二进制资源 (.resources) 文件。还可以使用资源文件生成器 (Resgen.exe) 从文本文件或 .resx 文件创建 .resources 文件。.resources 文件除了可以包含字符串数据之外，还可以包含二进制数据（字节数组）和对象数据。以编程方式创建 .resources 文件需要执行下列步骤：

1. 创建一个具有唯一文件名的 ResourceWriter 对象。可以通过将文件名或文件流指定为 ResourceWriter 类构造函数来实现这此操作。
2. 调用 ResourceWriter.AddResource 方法的重载之一以便将每个已命名的资源添加到文件。该资源可以是字符串、对象或二进制数据（字节数组）集合。
3. 通过调用 ResourceWriter.Close 方法将资源写入文件并关闭 ResourceWriter 对象。

> **NOTE**
> 请勿使用资源文件存储密码、安全敏感信息或私人数据。

下面的示例以编程方式创建了一个名为 CarResources.resources 的 .resources 文件，该文件存储了六个字符串、一个图标和两个由应用程序定义的对象（两个 `Automobile` 对象）。示例中定义并实例化的 `Automobile` 类，使用 SerializableAttribute 属性进行标记，这样二进制序列化格式化程序便可持久保留该类。

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
   private string carMake;
   private string carModel;
   private int carYear;
   private int carDoors;
   private int carCylinders;

   public Automobile(string make, string model, int year) :
                     this(make, model, year, 0, 0)
   { }

   public Automobile(string make, string model, int year,
                     int doors, int cylinders)
   {
      this.carMake = make;
      this.carModel = model;
      this.carYear = year;
      this.carDoors = doors;
      this.carCylinders = cylinders;
   }

   public string Make {
      get { return this.carMake; }
   }

   public string Model {
      get { return this.carModel; }
   }

   public int Year {
      get { return this.carYear; }
   }

   public int Doors {
      get {
         return this.carDoors; }
   }

   public int Cylinders {
      get {
         return this.carCylinders; }
   }
}

public class Example
{
   public static void Main()
   {
      // Instantiate an Automobile object.
      Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
      Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
      // Define a resource file named CarResources.resx.
```

```
      using (ResourceWriter rw = new ResourceWriter(@".\CarResources.resources"))
      {
         rw.AddResource("Title", "Classic American Cars");
         rw.AddResource("HeaderString1", "Make");
         rw.AddResource("HeaderString2", "Model");
         rw.AddResource("HeaderString3", "Year");
         rw.AddResource("HeaderString4", "Doors");
         rw.AddResource("HeaderString5", "Cylinders");
         rw.AddResource("Information", SystemIcons.Information);
         rw.AddResource("EarlyAuto1", car1);
         rw.AddResource("EarlyAuto2", car2);
      }
   }
}
```

```
Imports System.Drawing
Imports System.Resources

<Serializable()> Public Class Automobile
    Private carMake As String
    Private carModel As String
    Private carYear As Integer
    Private carDoors AS Integer
    Private carCylinders As Integer

    Public Sub New(make As String, model As String, year As Integer)
        Me.New(make, model, year, 0, 0)
    End Sub

    Public Sub New(make As String, model As String, year As Integer,
                   doors As Integer, cylinders As Integer)
        Me.carMake = make
        Me.carModel = model
        Me.carYear = year
        Me.carDoors = doors
        Me.carCylinders = cylinders
    End Sub

    Public ReadOnly Property Make As String
        Get
            Return Me.carMake
        End Get
    End Property

    Public ReadOnly Property Model As String
        Get
            Return Me.carModel
        End Get
    End Property

    Public ReadOnly Property Year As Integer
        Get
            Return Me.carYear
        End Get
    End Property

    Public ReadOnly Property Doors As Integer
        Get
            Return Me.carDoors
        End Get
    End Property

    Public ReadOnly Property Cylinders As Integer
        Get
            Return Me.carCylinders
        End Get
    End Property
```

```
End Class

Module Example
    Public Sub Main()
        ' Instantiate an Automobile object.
        Dim car1 As New Automobile("Ford", "Model N", 1906, 0, 4)
        Dim car2 As New Automobile("Ford", "Model T", 1909, 2, 4)
        ' Define a resource file named CarResources.resx.
        Using rw As New ResourceWriter(".\CarResources.resources")
            rw.AddResource("Title", "Classic American Cars")
            rw.AddResource("HeaderString1", "Make")
            rw.AddResource("HeaderString2", "Model")
            rw.AddResource("HeaderString3", "Year")
            rw.AddResource("HeaderString4", "Doors")
            rw.AddResource("HeaderString5", "Cylinders")
            rw.AddResource("Information", SystemIcons.Information)
            rw.AddResource("EarlyAuto1", car1)
            rw.AddResource("EarlyAuto2", car2)
        End Using
    End Sub
End Module
```

创建 .resources 文件后，可以通过含入语言编译器的 `/resource` 开关将其嵌入运行时可执行文件或库中，或者通过使用程序集链接器 (Al.exe) 将其嵌入附属程序集。

Visual Studio 中的资源文件

将资源文件添加到 Visual Studio 项目时，Visual Studio 会在项目目录中创建一个 .resx 文件。Visual Studio 会提供资源编辑器，可用于添加字符串、图像和二进制对象。编辑器只能用于处理静态数据，因此不能用于储存编程对象；必须以编程方式将对象数据写入 .resx 文件或 .resources 文件。有关详细信息，请参阅以编程方式使用 .resx 文件和 .resources 文件中的资源部分。

如果要添加本地化资源，请为它们提供与主资源文件相同的根文件名称。还应在文件名中指定其区域性。例如，如果要添加一个名为 Resources.resx 的资源文件，或许还需要创建名为 Resources.en-US.resx 和 Resources.fr-fr.resx 的资源文件，分别用以保存英语(美国)和法语(法国)区域性的本地化资源。还应该指定应用程序的默认区域性。找不到特定区域性的本地化资源时，将使用此默认区域性的资源。若要指定默认区域性，请在 Visual Studio 的解决方案资源管理器中，右键单击项目名称，指向“应用程序”，单击“程序集信息”，然后在“非特定语言”列表中选择合适的语言/区域性 。

编译时，Visual Studio 首先将项目中的 .resx 文件转换为二进制资源 (.resources) 文件，并将其存储在项目 obj 目录的子目录中。Visual Studio 会将不包含本地化资源的所有资源文件嵌入项目生成的主程序集中。如果资源文件包含本地化资源，Visual Studio 会将其嵌入用于每个本地化区域性的单独的附属程序集中。然后将每个附属程序集存储到名称与本地化区域性相对应的目录中。例如，本地化的英语(美国)资源存储在 en-US 子目录的附属程序集中。

请参阅

- System.Resources
- .NET 应用中的资源
- 打包和部署资源

以编程方式使用 .resx 文件

2021/11/16 •

由于 XML 资源 (.resx) 文件必须由定义完善的 XML 组成，这些 XML 的标头必须遵循特定架构（后跟名称/值对的数据），因此你会发现手动创建这些文件很容易出错。作为一种替代方法，可以使用 .NET 类库中的类型和成员以编程方式创建 .resx 文件。还可以使用 .NET 类库来检索存储在 .resx 文件中的资源。本文说明如何使用 System.Resources 命名空间的类型和成员来操作 .resx 文件。

本文讨论的是如何操作包含资源的 XML (.resx) 文件。有关操作嵌入程序集中的二进制资源文件的信息，请参阅 ResourceManager。

> **WARNING**
> 除编程方式以外还可以使用其他方式操作 .resx 文件。如果将资源文件添加到 Visual Studio 项目，那么 Visual Studio 会提供一个用于创建和维护 .resx 文件的接口，并且在编译时自动将 .resx 文件转换为 .resources 文件。你还可以使用文本编辑器来直接操作 .resx 文件。但是，若要避免破坏文件，请注意不要修改存储在文件中的任何二进制信息。

创建 .resx 文件

你可以使用 System.Resources.ResXResourceWriter 类以编程方式创建 .resx 文件，步骤如下：

1. 通过调用 ResXResourceWriter 方法和提供 .resx 文件的名称实例化 ResXResourceWriter(String) 对象。此文件名必须包括 .resx 扩展名。如果实例化位于 ResXResourceWriter 块中的 `using` 对象，则无需在步骤 3 中显式调用 ResXResourceWriter.Close 方法。

2. 为每个要添加到此文件中的资源调用 ResXResourceWriter.AddResource 方法。使用此方法的重载添加字符串、对象和二进制（字节数组）数据。如果资源是一个对象，则它必须是可序列化的。

3. 调用 ResXResourceWriter.Close 方法以生成资源文件并释放所有资源。如果 ResXResourceWriter 对象是在 `using` 块中创建的，那么资源将写入 .resx 文件，并且在 ResXResourceWriter 块的末尾释放 `using` 对象所使用的资源。

生成的 .resx 文件具有相应的标头，并且由 `data` 方法添加的每个资源都有一个 ResXResourceWriter.AddResource 标记。

> **WARNING**
> 请勿使用资源文件存储密码、安全敏感信息或私人数据。

下面的示例创建了一个名为 CarResources.resx 的 .resx 文件，该文件存储了六个字符串、一个图标和两个由应用程序定义的对象（两个 `Automobile` 对象）。本示例中定义并实例化的 `Automobile` 类使用 SerializableAttribute 属性进行标记。

```
using System;
using System.Drawing;
using System.Resources;

[Serializable()] public class Automobile
{
   private string carMake;
   private string carModel;
   private int carYear;
```

```
   private int carDoors;
   private int carCylinders;

   public Automobile(string make, string model, int year) :
                     this(make, model, year, 0, 0)
   { }

   public Automobile(string make, string model, int year,
                     int doors, int cylinders)
   {
      this.carMake = make;
      this.carModel = model;
      this.carYear = year;
      this.carDoors = doors;
      this.carCylinders = cylinders;
   }

   public string Make {
      get { return this.carMake; }
   }

   public string Model {
      get {return this.carModel; }
   }

   public int Year {
      get { return this.carYear; }
   }

   public int Doors {
      get { return this.carDoors; }
   }

   public int Cylinders {
      get { return this.carCylinders; }
   }
}

public class Example
{
   public static void Main()
   {
      // Instantiate an Automobile object.
      Automobile car1 = new Automobile("Ford", "Model N", 1906, 0, 4);
      Automobile car2 = new Automobile("Ford", "Model T", 1909, 2, 4);
      // Define a resource file named CarResources.resx.
      using (ResXResourceWriter resx = new ResXResourceWriter(@".\CarResources.resx"))
      {
         resx.AddResource("Title", "Classic American Cars");
         resx.AddResource("HeaderString1", "Make");
         resx.AddResource("HeaderString2", "Model");
         resx.AddResource("HeaderString3", "Year");
         resx.AddResource("HeaderString4", "Doors");
         resx.AddResource("HeaderString5", "Cylinders");
         resx.AddResource("Information", SystemIcons.Information);
         resx.AddResource("EarlyAuto1", car1);
         resx.AddResource("EarlyAuto2", car2);
      }
   }
}
```

```
Imports System.Drawing
Imports System.Resources

<Serializable()> Public Class Automobile
    Private carMake As String
    Private carModel As String
    Private carYear As Integer
```

```
    Private carYear AS Integer
    Private carDoors AS Integer
    Private carCylinders As Integer

    Public Sub New(make As String, model As String, year As Integer)
        Me.New(make, model, year, 0, 0)
    End Sub

    Public Sub New(make As String, model As String, year As Integer,
                   doors As Integer, cylinders As Integer)
        Me.carMake = make
        Me.carModel = model
        Me.carYear = year
        Me.carDoors = doors
        Me.carCylinders = cylinders
    End Sub

    Public ReadOnly Property Make As String
        Get
            Return Me.carMake
        End Get
    End Property

    Public ReadOnly Property Model As String
        Get
            Return Me.carModel
        End Get
    End Property

    Public ReadOnly Property Year As Integer
        Get
            Return Me.carYear
        End Get
    End Property

    Public ReadOnly Property Doors As Integer
        Get
            Return Me.carDoors
        End Get
    End Property

    Public ReadOnly Property Cylinders As Integer
        Get
            Return Me.carCylinders
        End Get
    End Property
End Class

Module Example
    Public Sub Main()
        ' Instantiate an Automobile object.
        Dim car1 As New Automobile("Ford", "Model N", 1906, 0, 4)
        Dim car2 As New Automobile("Ford", "Model T", 1909, 2, 4)
        ' Define a resource file named CarResources.resx.
        Using resx As New ResXResourceWriter(".\CarResources.resx")
            resx.AddResource("Title", "Classic American Cars")
            resx.AddResource("HeaderString1", "Make")
            resx.AddResource("HeaderString2", "Model")
            resx.AddResource("HeaderString3", "Year")
            resx.AddResource("HeaderString4", "Doors")
            resx.AddResource("HeaderString5", "Cylinders")
            resx.AddResource("Information", SystemIcons.Information)
            resx.AddResource("EarlyAuto1", car1)
            resx.AddResource("EarlyAuto2", car2)
        End Using
    End Sub
End Module
```

TIP

还可以使用 Visual Studio 创建 .resx 文件。在编译时，Visual Studio 使用 资源文件生成器 (Resgen.exe) 将 .resx 文件转换为二进制资源 (.resources) 文件，然后还将其嵌入应用程序集或附属程序集。

你无法将 .resx 文件嵌入运行时可执行文件中或将其编译到附属程序集。必须使用 资源文件生成器 (Resgen.exe)将 .resx 文件转换为二进制资源 (.resources) 文件 。然后可以将生成的 .resources 文件嵌入应用程序集或附属程序集。有关详细信息，请参阅创建资源文件。

枚举资源

在某些情况下，你可能想要从 .resx 文件中检索所有资源，而不是某个特定资源。若要执行此操作，可以使用 System.Resources.ResXResourceReader 类，该类为 .resx 文件中的所有资源提供枚举器。System.Resources.ResXResourceReader 类将实现 IDictionaryEnumerator，并返回 DictionaryEntry 对象，该对象代表用于循环的每个迭代的特定资源。此对象的 DictionaryEntry.Key 属性返回资源的键，DictionaryEntry.Value 属性返回资源的值。

下面的示例为前面的示例中创建的 CarResources.resx 文件创建 ResXResourceReader 对象，并在整个资源文件中进行迭代。该示例将资源文件中定义的两个 `Automobile` 对象添加到 System.Collections.Generic.List<T> 对象，以及将六个字符串中的五个添加到 SortedList 对象。SortedList 对象中的值将转换为一个参数数组，该参数数组用于向控制台显示列标题。还将向控制台显示 `Automobile` 属性值。

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Resources;

public class Example
{
   public static void Main()
   {
      string resxFile = @".\CarResources.resx";
      List<Automobile> autos = new List<Automobile>();
      SortedList headers = new SortedList();

      using (ResXResourceReader resxReader = new ResXResourceReader(resxFile))
      {
         foreach (DictionaryEntry entry in resxReader) {
            if (((string) entry.Key).StartsWith("EarlyAuto"))
               autos.Add((Automobile) entry.Value);
            else if (((string) entry.Key).StartsWith("Header"))
               headers.Add((string) entry.Key, (string) entry.Value);
         }
      }
      string[] headerColumns = new string[headers.Count];
      headers.GetValueList().CopyTo(headerColumns, 0);
      Console.WriteLine("{0,-8} {1,-10} {2,-4}   {3,-5}   {4,-9}\n",
                        headerColumns);
      foreach (var auto in autos)
         Console.WriteLine("{0,-8} {1,-10} {2,4}   {3,5}   {4,9}",
                           auto.Make, auto.Model, auto.Year,
                           auto.Doors, auto.Cylinders);
   }
}
// The example displays the following output:
//       Make     Model      Year   Doors   Cylinders
//
//       Ford     Model N    1906       0           4
//       Ford     Model T    1909       2           4
```

```
Imports System.Collections
Imports System.Collections.Generic
Imports System.Resources

Module Example
    Public Sub Main()
        Dim resxFile As String = ".\CarResources.resx"
        Dim autos As New List(Of Automobile)
        Dim headers As New SortedList()

        Using resxReader As New ResXResourceReader(resxFile)
            For Each entry As DictionaryEntry In resxReader
                If CType(entry.Key, String).StartsWith("EarlyAuto") Then
                    autos.Add(CType(entry.Value, Automobile))
                Else If CType(entry.Key, String).StartsWith("Header") Then
                    headers.Add(CType(entry.Key, String), CType(entry.Value, String))
                End If
            Next
        End Using
        Dim headerColumns(headers.Count - 1) As String
        headers.GetValueList().CopyTo(headerColumns, 0)
        Console.WriteLine("{0,-8} {1,-10} {2,-4}   {3,-5}   {4,-9}",
                          headerColumns)
        Console.WriteLine()
        For Each auto In autos
            Console.WriteLine("{0,-8} {1,-10} {2,4}   {3,5}   {4,9}",
                              auto.Make, auto.Model, auto.Year,
                              auto.Doors, auto.Cylinders)
        Next
    End Sub
End Module
' The example displays the following output:
'       Make     Model      Year   Doors   Cylinders
'
'       Ford     Model N    1906       0           4
'       Ford     Model T    1909       2           4
```

检索特定的资源

除了枚举 .resx 文件中的项，你可以使用 System.Resources.ResXResourceSet 类按名称检索特定资源。ResourceSet.GetString(String) 方法用于检索命名字符串资源的值。ResourceSet.GetObject(String) 方法用于检索命名对象或二进制数据的值。该方法返回的对象之后必须转换为(C# 中的 cast 或 Visual Basic 中的 convert)相应类型的对象。

以下示例按资源名称检索窗体的标题字符串和图标，还检索之前示例中使用的应用程序定义的 `Automobile` 对象，并在 DataGridView 控件中显示这些对象。

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Resources;
using System.Windows.Forms;

public class CarDisplayApp : Form
{
   private const string resxFile = @".\CarResources.resx";
   Automobile[] cars;

   public static void Main()
   {
      CarDisplayApp app = new CarDisplayApp();
      Application.Run(app);
   }

   public CarDisplayApp()
   {
      // Instantiate controls.
      PictureBox pictureBox = new PictureBox();
      pictureBox.Location = new Point(10, 10);
      this.Controls.Add(pictureBox);
      DataGridView grid = new DataGridView();
      grid.Location = new Point(10, 60);
      this.Controls.Add(grid);

      // Get resources from .resx file.
      using (ResXResourceSet resxSet = new ResXResourceSet(resxFile))
      {
         // Retrieve the string resource for the title.
         this.Text = resxSet.GetString("Title");
         // Retrieve the image.
         Icon image = (Icon) resxSet.GetObject("Information", true);
         if (image != null)
            pictureBox.Image = image.ToBitmap();

         // Retrieve Automobile objects.
         List<Automobile> carList = new List<Automobile>();
         string resName = "EarlyAuto";
         Automobile auto;
         int ctr = 1;
         do {
            auto = (Automobile) resxSet.GetObject(resName + ctr.ToString());
            ctr++;
            if (auto != null)
               carList.Add(auto);
         } while (auto != null);
         cars = carList.ToArray();
         grid.DataSource = cars;
      }
   }
}
```

```
Imports System.Collections.Generic
Imports System.Drawing
Imports System.Resources
Imports System.Windows.Forms

Public Class CarDisplayApp : Inherits Form
    Private Const resxFile As String = ".\CarResources.resx"
    Dim cars() As Automobile

    Public Shared Sub Main()
        Dim app As New CarDisplayApp()
        Application.Run(app)
    End Sub

    Public Sub New()
        ' Instantiate controls.
        Dim pictureBox As New PictureBox()
        pictureBox.Location = New Point(10, 10)
        Me.Controls.Add(pictureBox)
        Dim grid As New DataGridView()
        grid.Location = New Point(10, 60)
        Me.Controls.Add(grid)

        ' Get resources from .resx file.
        Using resxSet As New ResXResourceSet(resxFile)
            ' Retrieve the string resource for the title.
            Me.Text = resxSet.GetString("Title")
            ' Retrieve the image.
            Dim image As Icon = CType(resxSet.GetObject("Information", True), Icon)
            If image IsNot Nothing Then
                pictureBox.Image = image.ToBitmap()
            End If

            ' Retrieve Automobile objects.
            Dim carList As New List(Of Automobile)
            Dim resName As String = "EarlyAuto"
            Dim auto As Automobile
            Dim ctr As Integer = 1
            Do
                auto = CType(resxSet.GetObject(resName + ctr.ToString()), Automobile)
                ctr += 1
                If auto IsNot Nothing Then carList.Add(auto)
            Loop While auto IsNot Nothing
            cars = carList.ToArray()
            grid.DataSource = cars
        End Using
    End Sub
End Class
```

将 .resx 文件转换为二进制 .resources 文件

将 .resx 文件转换为嵌入式二进制资源 (.resources) 文件可以带来明显的好处。虽然 .resx 文件容易阅读，并且在应用程序开发期间易于维护，但是完成的应用程序很少包含这些文件。如果这些文件随应用程序分发，那么它们将作为单独文件存在，与应用程序可执行文件和随附的库分开存放。与此相反，.resources 文件则嵌入应用程序可执行文件或其随附的程序集中。另外，对于在运行时依赖 .resx 文件的本地化应用程序，开发人员负责处理资源回退。与此相反，如果创建了一组包含嵌入式 .resources 文件的附属程序集，则公共语言运行时将处理资源回退进程。

要将 .resx 文件转换为 .resources 文件，请使用资源文件生成器 (Resgen.exe)，它具有以下基本语法：

```
resgen.exe .resxFilename
```

这会生成一个具有与 .resx 文件相同的根文件名的二进制资源文件，并且该文件的扩展名为 .resources。然后，可在编译时将该文件编译为可执行文件或库。如果你使用的是 Visual Basic 编译器，则使用以下语法以在应用程序的可执行文件中嵌入一个 .resources 文件：

```
vbc filename .vb -resource: .resourcesFilename
```

如果使用的是 C#，则语法如下所示：

```
 csc filename .cs -resource: .resourcesFilename
```

还可以使用程序集链接器 (AL.exe) 在附属程序集中嵌入 .resources 文件，基本语法如下：

```
al resourcesFilename -out: assemblyFilename
```

另请参阅

- 创建资源文件
- 资源文件生成器 (Resgen.exe)
- 程序集链接器 (al.exe)

为 .NET 应用创建附属程序集

2021/11/16 •

资源文件在本地化的应用程序中具有核心作用。通过资源文件，应用程序可以使用用户自己的语言和区域性显示字符串、图像及其他数据，并且在用户自己的语言或区域性资源不可用时，提供备用数据。.NET Framework 使用中心辐射型模型来查找和检索已本地化的资源。中心即主程序集，包含不可本地化的可执行代码和单个区域性（称作非特定区域性或默认区域性）的资源。默认区域性是应用程序的回退区域性；没有任何已本地化的资源可用时，则使用默认区域性。使用 NeutralResourcesLanguageAttribute 属性来指定应用程序默认区域性的区域性。每条轮辐均连接到一个附属程序集，该附属程序集包含单个本地化区域性的资源，但不包含任何代码。因为附属程序集不是主程序集的一部分，所以不必替换该应用程序的主程序集即可轻松更新或替换与特定区域性相对应的资源。

> **NOTE**
> 应用程序默认区域性的资源也可以存储在附属程序集中。为此，可为 NeutralResourcesLanguageAttribute 属性分配一个 UltimateResourceFallbackLocation.Satellite 值。

附属程序集名称和位置

中心辐射型模型要求将资源放在特定位置，以便轻松查找和使用资源。如果未按预期编译和命名资源，或未将其放在正确的位置，则公共语言运行时将无法定位它们，并改为使用默认区域性的资源。.NET Framework 资源管理器由 ResourceManager 对象表示，用于自动访问本地化的资源。该资源管理器的相关要求如下：

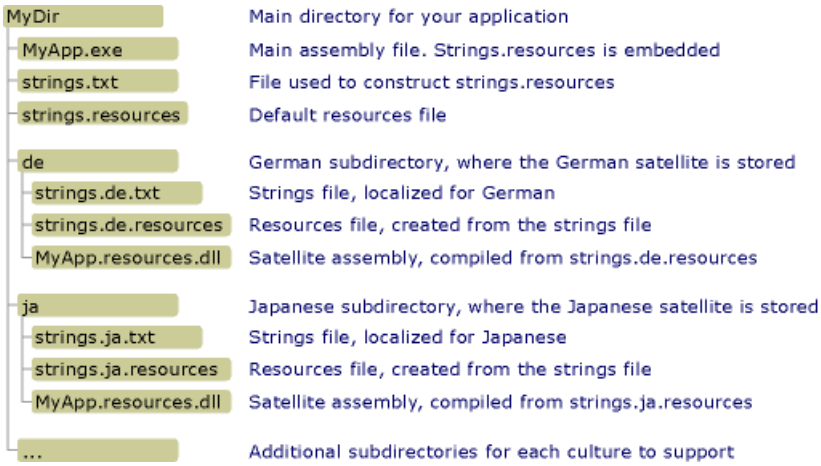
- 单个附属程序集必须包含特定区域性的所有资源。换言之，应该将多个 .txt 或 .resx 文件编译成单个二进制 .resources 文件。
- 存储区域性资源的每个本地化区域性的应用程序目录中必须有一个单独的子目录。该子目录名称必须与区域性名称相同。或者，也可以将附属程序集存储在全局程序集缓存中。在这种情况下，程序集强名称的区域性信息组件必须指明其区域性。有关详细信息，请参阅在全局程序集缓存中安装附属程序集。

  > **NOTE**
  > 如果应用程序中包含了子区域性的资源，则将每个子区域性放在应用程序目录下的单独子目录中。不要将子区域性放在其主区域性目录下的子目录中。

- 附属程序集的名称必须与应用程序相同，并且必须使用文件扩展名“.resources.dll”。例如，如果应用程序名为 Example.exe，则每个附属程序集的名称应该为 Example.resources.dll 。请注意附属程序集名称不指示其资源文件的区域性。但是，附属程序集会显示在不指定区域性的目录中。
- 附属程序集的区域性的相关信息必须包括在程序集的元数据中。若要将区域性名称存储在附属程序集的元数据中，则在使用程序集链接器将资源嵌入附属程序集时指定 `/culture` 选项。

下图显示未安装在全局程序集缓存中的应用程序的示例目录结构和位置要求。具有 .txt 和 .resources 扩展名的项不会随附在最终应用程序中。这些是用于创建最终附属资源程序集的中间资源文件。在此示例中，可以将 .txt 文件替换为 .resx 文件。有关详细信息，请参阅打包和部署资源。

下图展示了附属程序集目录：



编译附属程序集

使用资源文件生成器 (resgen.exe) 将包含资源的文本文件或 XML (.resx) 文件编译为二进制 .resources 文件 。然后使用程序集链接器 (al.exe) 将 .resources 文件编译到附属程序集中 。al.exe 从指定的 .resources 文件创建程序集 。附属程序集只能包含资源，而不能包含任何可执行代码。

下面的 al.exe 命令从德语资源文件 strings.de.resources 为 `Example` 应用程序创建了一个附属程序集 。

```
al -target:lib -embed:strings.de.resources -culture:de -out:Example.resources.dll
```

下面的 al.exe 命令也从文件 strings.de.resources 为 `Example` 应用程序创建了一个附属程序集 。/Template 选项会导致附属程序集从父程序集 (Example.dll) 继承除区域性信息之外的所有程序集元数据。

```
al -target:lib -embed:strings.de.resources -culture:de -out:Example.resources.dll -template:Example.dll
```

下表详细展示了这些命令中使用的 al.exe 选项：

"	"
`-target:lib`	指定将附属程序集编译成库 (.dll) 文件。因为附属程序集不包含可执行代码，并且不是应用程序的主程序集，所以必须将附属程序集另存为 DLL。
`-embed:strings.de.resources`	指定在 Al.exe 编译程序集时要嵌入的资源文件名。可在附属程序集中嵌入多个 .resources 文件，但如果依循中心辐射模型，则必须为每个区域性编译一个附属程序集。但是，可以为字符串和对象创建单独的 .resources 文件。
`-culture:de`	指定要编译的资源的区域性。公共语言运行时搜索特定区域性的资源时会使用此信息。如果省略此选项，Al.exe 仍然会编译资源，但当用户请求资源时，运行时将无法找到该资源。
`-out:Example.resources.dll`	指定输出文件的名称。名称必须遵循命名标准 baseName.resources.extension，其中 baseName 是主程序集的名称，extension 是有效的文件扩展名（例如 .dll）。请注意，运行时无法根据输出文件名确定附属程序集的区域性，必须使用 /culture 选项指定 。
`-template:Example.dll`	指定程序集，附属程序集将从该程序集继承除区域性字段之外的所有程序集元数据。仅当指定了具有强名称的程序集时，此选项才会影响附属程序集。

有关 al.exe 可用选项的完整列表，请参阅程序集链接器 (al.exe) 。

附属程序集示例

下面是一个简单的“Hello world”示例，该示例展示了一个包含本地化的问候语的消息框。此示例包含了英语(美国)、法语(法国)和俄语(俄罗斯)区域性的资源，并且其回退区域性为英语。要创建示例，请执行以下操作：

1. 创建一个名为 Greeting.resx 或 Greeting.txt 的资源文件，使其包含默认区域性的资源。在此文件中存储一个名为 `HelloString`，并且值为“Hello world!”的 单个字符串。

2. 为指示英语 (en) 是该应用程序的默认区域性，请将以下 System.Resources.NeutralResourcesLanguageAttribute 属性添加到应用程序的 AssemblyInfo 文件或主源代码文件中，该文件之后会编译到应用程序的主程序集中。

   ```
   [assembly: NeutralResourcesLanguage("en")]
   ```

   ```
   <Assembly: NeutralResourcesLanguage("en")>
   ```

3. 向应用程序添加对其他区域性的支持（`en-US`、`fr-FR` 和 `ru-RU`），如下所示：

   - 若要支持 `en-US` 或英语(美国)区域性，创建一个名为 Greeting.en-US.resx 或 Greeting.en-US.txt 的资源文件，并在其中存储一个名为 `HelloString` 且值为“Hi world!”的单个字符串 。
   - 若要支持 `fr-FR` 或法语(法国)区域性，创建一个名为 Greeting.fr-FR.resx 或 Greeting.fr-FR.txt 的资源文件，并在其中存储一个名为 `HelloString` 且值为“Salut tout le monde!”的单字符串 。
   - 若要支持 `ru-RU` 或俄语(俄罗斯)区域性，创建一个名为 Greeting.ru-RU.resx 或 Greeting.ru-RU.txt 的资源文件，并在其中存储一个名为 `HelloString` 且值为“Всем привет!”的单字符串 。

4. 使用 resgen.exe 将每个文本文件或 XML 资源文件编译为二进制 .resources 文件 。其输出是一组与 .resx 或 .txt 文件具有相同根文件名的文件，这些文件的扩展名为 .resources 。如果使用 Visual Studio 创建该示例，将自动处理编译过程。如果不使用 Visual Studio，则运行以下命令将 .resx 文件编译为 .resources 文件：

   ```
   resgen Greeting.resx
   resgen Greeting.en-us.resx
   resgen Greeting.fr-FR.resx
   resgen Greeting.ru-RU.resx
   ```

   如果资源位于文本文件而非 XML 文件中，则将 .resx 扩展名替换为 .txt 。

5. 将以下源代码和默认区域性的资源编译到应用程序的主程序集中：

   > **IMPORTANT**
   > 如果使用的是命令行而不是 Visual Studio 来创建此示例，则应将对 ResourceManager 类构造函数的调用修改为后列内容：`ResourceManager rm = new ResourceManager("Greeting", typeof(Example).Assembly);`

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;
using System.Windows.Forms;

class Example
{
   static void Main()
   {
      // Create array of supported cultures
      string[] cultures = {"en-CA", "en-US", "fr-FR", "ru-RU"};
      Random rnd = new Random();
      int cultureNdx = rnd.Next(0, cultures.Length);
      CultureInfo originalCulture = Thread.CurrentThread.CurrentCulture;

      try {
         CultureInfo newCulture = new CultureInfo(cultures[cultureNdx]);
         Thread.CurrentThread.CurrentCulture = newCulture;
         Thread.CurrentThread.CurrentUICulture = newCulture;
         ResourceManager rm = new ResourceManager("Example.Greeting",
                                                  typeof(Example).Assembly);
         string greeting = String.Format("The current culture is {0}.\n{1}",
                                         Thread.CurrentThread.CurrentUICulture.Name,
                                         rm.GetString("HelloString"));

         MessageBox.Show(greeting);
      }
      catch (CultureNotFoundException e) {
         Console.WriteLine("Unable to instantiate culture {0}", e.InvalidCultureName);
      }
      finally {
         Thread.CurrentThread.CurrentCulture = originalCulture;
         Thread.CurrentThread.CurrentUICulture = originalCulture;
      }
   }
}
```

```
Imports System.Globalization
Imports System.Resources
Imports System.Threading

Module Module1

    Sub Main()
        ' Create array of supported cultures
        Dim cultures() As String = {"en-CA", "en-US", "fr-FR", "ru-RU"}
        Dim rnd As New Random()
        Dim cultureNdx As Integer = rnd.Next(0, cultures.Length)
        Dim originalCulture As CultureInfo = Thread.CurrentThread.CurrentCulture

        Try
            Dim newCulture As New CultureInfo(cultures(cultureNdx))
            Thread.CurrentThread.CurrentCulture = newCulture
            Thread.CurrentThread.CurrentUICulture = newCulture
            Dim greeting As String = String.Format("The current culture is {0}.{1}{2}",
                                                   Thread.CurrentThread.CurrentUICulture.Name,
                                                   vbCrLf, My.Resources.Greetings.HelloString)

            MsgBox(greeting)
        Catch e As CultureNotFoundException
            Console.WriteLine("Unable to instantiate culture {0}", e.InvalidCultureName)
        Finally
            Thread.CurrentThread.CurrentCulture = originalCulture
            Thread.CurrentThread.CurrentUICulture = originalCulture
        End Try
    End Sub
End Module
```

如果应用程序名为 Example，并从命令行进行编译，则适用于 C# 编译器的命令为：

```
csc Example.cs -res:Greeting.resources
```

相应的 Visual Basic 编译器命令为：

```
vbc Example.vb -res:Greeting.resources
```

6. 在主应用程序目录中为应用程序支持的每个本地化的区域性创建一个子目录。应该创建一个 en-US、fr-FR 和 ru-RU 子目录 。Visual Studio 将在编译过程中自动创建这些子目录。

7. 将某个区域性特定的 .resources 文件嵌入附属程序集，并将其保存到相应的目录 。用于为每个 .resources 文件执行此操作的命令是 ：

   ```
   al -target:lib -embed:Greeting.culture.resources -culture:culture -out:culture\Example.resources.dll
   ```

   其中 culture 是由附属程序集包含了资源的区域性的名称 。Visual Studio 会自动处理这一过程。

然后便可运行该示例。它会随机将某种支持的区域性设为当前区域性，并显示本地化的问候语。

在全局程序集缓存中安装附属程序集

可以在全局程序集缓存(而不是本地应用程序子目录)中安装程序集。当具有供多个应用程序使用的类库和类库资源程序集时，这种方式非常有用。

在全局程序集缓存中安装程序集要求程序集具有强名称。具有强名称的程序集使用有效公钥/私钥对进行签名。它们包含版本信息，运行时会使用这些版本信息来确定使用哪个程序集来满足绑定要求。有关强名称和版本控

制的详细信息，请参阅程序集版本控制。有关强名称的详细信息，请参阅具有强名称的程序集。

开发应用程序时，不可能具有对最终公钥/私钥对的访问权限。为在全局程序集缓存中安装附属程序集，并确保附属程序集工作正常，可使用延迟签名技术。延迟为程序集签名时，生成时可在文件中保留用于强名称签名的空间。实际签名将延迟到以后的某个时间进行，即当最后的公钥/私钥对可用时。有关延迟签名的详细信息，请参阅延迟为程序集签名。

获取公钥

若要延迟程序集签名，必须具有公钥访问权限。可以从公司中将进行最终签名的组织处获取真正的公钥，也可以使用强名称工具 (Sn.exe) 创建一个公钥。

下面的 Sn.exe 命令创建一个测试公钥/私钥对 。–k 选项指定 Sn.exe 应新建一个密钥对，并将其保存在 TestKeyPair.snk 文件中 。

```
sn –k TestKeyPair.snk
```

可以从包含测试密钥对的文件中提取公钥。以下命令从 TestKeyPair.snk 中提取公钥，并将其保存在 PublicKey.snk 中 ：

```
sn –p TestKeyPair.snk PublicKey.snk
```

延迟为程序集签名

获取或创建公钥后，使用程序集链接器 (al.exe) 编译程序集，并指定延迟签名。

下面的 al.exe 命令从 strings.ja.resources 文件为应用程序 StringLibrary 创建了一个强名称附属程序集 ：

```
al -target:lib -embed:strings.ja.resources -culture:ja -out:StringLibrary.resources.dll -delay+ -
keyfile:PublicKey.snk
```

-delay+ 选项指定程序集链接器应延迟对程序集签名 。-keyfile 选项指定密钥文件的名称，该文件包含用以延迟程序集签名的公钥 。

对程序集重新签名

部署应用程序之前，必须使用真正的密钥对对延迟签名的附属程序集重新签名。可以使用 Sn.exe 执行此操作 。

下面的 Sn.exe 命令使用 RealKeyPair.snk 文件中存储的密钥对对 StringLibrary.resources.dll 进行签名 。– R 选项指定对之前已签名或延迟签名的程序集进行重新签名 。

```
sn –R StringLibrary.resources.dll RealKeyPair.snk
```

在全局程序集缓存中安装附属程序集

运行时在资源回退进程中搜索资源时，首先会在全局程序集缓存中查找。（有关详细信息，请参阅包和部署资源主题的"资源回退过程"部分）使用强名称对附属程序集签名后，即可使用全局程序集缓存工具 (gacutil.exe) 在全局程序集缓存中安装该程序集。

下面的 Gacutil.exe 命令在全局程序集缓存中安装了 StringLibrary.resources.dll ：

```
gacutil -i:StringLibrary.resources.dll
```

/I 选项指定 Gacutil.exe 应将指定的程序集安装到全局程序集缓存中 。在缓存中安装了附属程序集之后，要使用附属程序集的应用程序便能够使用该附属程序集所包含的资源。

全局程序集缓存中的资源：示例

以下示例使用 .NET Framework 类库中的方法从资源文件中提取和返回本地化的问候语。在全局程序集缓存中注册库及其资源。示例包括了英语(美国)、法语(法国)、俄语(俄罗斯)和英语区域性的资源。英语是默认区域性;其资源存储在主程序集中。此示例最初使用公钥延迟对库及其附属程序集签名，然后使用公钥/私钥对为其重新签名。要创建示例，请执行以下操作:

1. 如果不是使用 Visual Studio，则使用以下强名称工具 (Sn.exe) 命令创建名为 ResKey.snk 的公钥/私钥对 :

   ```
   sn -k ResKey.snk
   ```

   如果使用的是 Visual Studio，则使用项目"属性"对话框中的"签名"选项卡生成密钥文件 。

2. 使用以下强名称工具 (Sn.exe) 命令创建名为 PublicKey.snk 的公钥文件 :

   ```
   sn -p ResKey.snk PublicKey.snk
   ```

3. 创建一个名为 Strings.resx 的资源文件，使其包含默认区域性的资源 。在此文件中存储一个名为 `Greeting`，并且值为"How do you do?"的 单个字符串。

4. 为指示 "en" 是该应用程序的默认区域性，请将以下 System.Resources.NeutralResourcesLanguageAttribute 属性添加到应用程序的 AssemblyInfo 文件或主源代码文件中，该文件之后会编译到应用程序的主程序集中:

   ```
   [assembly:NeutralResourcesLanguageAttribute("en")]
   ```

   ```
   <Assembly: NeutralResourcesLanguageAttribute("en")>
   ```

5. 向应用程序添加对其他区域性的支持(en-US、fr-FR 和 ru-RU 区域性)，如下所示:

   - 若要支持"en-US"或英语(美国)区域性，则创建一个名为 Strings.en-US.resx 或 Strings.en-US.txt 的资源文件，并在其中存储一个名为 `Greeting` 且值为"Hello"的单个字符串 。
   - 若要支持"fr-FR"或法语(法国)区域性，创建一个名为 Strings.fr-FR.resx 或 Strings.fr-FR.txt 的资源文件，并在其中存储一个值为"Bon jour!"的 `Greeting` 单字符串 。
   - 若要支持"ru-RU"或俄语(俄罗斯)区域性，创建一个名为 Strings.ru-RU.resx 或 Strings.ru-RU.txt 的资源文件，并在其中存储一个值为"Привет"的 `Greeting` 单字符串 。

6. 使用 resgen.exe 将每个文本文件或 XML 资源文件编译为二进制 .resources 文件。其输出是一组与 .resx 或 .txt 文件具有相同根文件名的文件，这些文件的扩展名为 .resources 。如果使用 Visual Studio 创建该示例，将自动处理编译过程。如果不使用 Visual Studio，则运行以下命令将 .resx 文件编译进 .resources 文件 :

   ```
   resgen filename
   ```

   其中 filename 是 .resx 或文本文件的可选路径、文件名和扩展名 。

7. 将以下 StringLibrary.vb 或 StringLibrary.cs 的源代码连同默认区域性的资源编译到名为 StringLibrary.dll 的延迟签名库程序集中 :

> **IMPORTANT**
>
> 如果使用命令行而不是 Visual Studio 来创建此示例，则将对 ResourceManager 类构造函数的调用修改为 `ResourceManager rm = new ResourceManager("Strings", typeof(Example).Assembly);`。

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using System.Threading;

[assembly:NeutralResourcesLanguageAttribute("en")]

public class StringLibrary
{
   public string GetGreeting()
   {
      ResourceManager rm = new ResourceManager("Strings",
                           Assembly.GetAssembly(typeof(StringLibrary)));
      string greeting = rm.GetString("Greeting");
      return greeting;
   }
}
```

```
Imports System.Globalization
Imports System.Reflection
Imports System.Resources
Imports System.Threading

<Assembly: NeutralResourcesLanguageAttribute("en")>

Public Class StringLibrary
    Public Function GetGreeting() As String
        Dim rm As New ResourceManager("Strings", _
                                      Assembly.GetAssembly(GetType(StringLibrary)))
        Dim greeting As String = rm.GetString("Greeting")
        Return greeting
    End Function
End Class
```

适用于 C# 编译器的命令：

```
csc -t:library -resource:Strings.resources -delaysign+ -keyfile:publickey.snk StringLibrary.cs
```

相应的 Visual Basic 编译器命令为：

```
vbc -t:library -resource:Strings.resources -delaysign+ -keyfile:publickey.snk StringLibrary.vb
```

8. 在主应用程序目录中为应用程序支持的每个本地化的区域性创建一个子目录。应该创建一个 en-US、fr-FR 和 ru-RU 子目录 。Visual Studio 将在编译过程中自动创建这些子目录。由于所有附属程序集都具有相同的文件名，所以使用子目录存储单个区域性特定的附属程序集，直到使用公钥/私钥对为附属程序集签名为止。

9. 将某个特定区域性的 .resources 文件嵌入延迟签名的附属程序集，并将其保存到相应的目录 。用于为每个 .resources 文件执行此操作的命令是 ：

```
al -target:lib -embed:Strings.culture.resources -culture:culture -
out:culture\StringLibrary.resources.dll -delay+ -keyfile:publickey.snk
```

其中 culture 是区域性的名称 。在此示例中，区域性名称是 en-US、fr-FR 和 ru-RU。

10. 使用强名称工具 (sn.exe) 对 StringLibrary.dll 重新签名，如下所示：

```
sn –R StringLibrary.dll RealKeyPair.snk
```

11. 对单个附属程序集重新签名。要执行此操作，请按如下所示对每个附属程序集使用强名称工具 (sn.exe)：

```
sn –R StringLibrary.resources.dll RealKeyPair.snk
```

12. 使用以下命令在全局程序集缓存中注册 StringLibrary.dll 及其每个附属程序集：

```
gacutil -i filename
```

其中 filename 是要注册的文件的名称 。

13. 如果使用的是 Visual Studio，则新建一个名为 `Example` 的控制台应用程序项目，向其添加对 StringLibrary.dll 的引用和以下源代码，然后进行编译 。

```
using System;
using System.Globalization;
using System.Threading;

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-GB", "en-US", "fr-FR", "ru-RU" };
      Random rnd = new Random();
      string cultureName = cultureNames[rnd.Next(0, cultureNames.Length)];
      Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(cultureName);
      Console.WriteLine("The current UI culture is {0}",
                        Thread.CurrentThread.CurrentUICulture.Name);
      StringLibrary strLib = new StringLibrary();
      string greeting = strLib.GetGreeting();
      Console.WriteLine(greeting);
   }
}
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-GB", "en-US", "fr-FR", "ru-RU"}
        Dim rnd As New Random()
        Dim cultureName As String = cultureNames(rnd.Next(0, cultureNames.Length))
        Thread.CurrentThread.CurrentUICulture = CultureInfo.CreateSpecificCulture(cultureName)
        Console.WriteLine("The current UI culture is {0}",
                          Thread.CurrentThread.CurrentUICulture.Name)
        Dim strLib As New StringLibrary()
        Dim greeting As String = strLib.GetGreeting()
        Console.WriteLine(greeting)
    End Sub
End Module
```

若要通过命令行进行编译，请在 C# 编译器中使用以下命令：

```
csc Example.cs -r:StringLibrary.dll
```

适用于 Visual Basic 编译器的命令行为：

```
vbc Example.vb -r:StringLibrary.dll
```

14. 运行 Example.exe 。

请参阅

- 打包和部署资源
- 延迟为程序集签名
- al.exe（程序集链接器）
- sn.exe（强名称工具）
- gacutil.exe（全局程序集缓存工具）
- .NET 中的资源

打包和部署 .NET 应用中的资源

2021/11/16 •

应用程序依靠 .NET Framework Resource Manager（由 ResourceManager 类表示）来检索已本地化的资源。Resource Manager 假定使用中枢轮辐式模型来打包和部署资源。中枢即主程序集，包含不可本地化的可执行代码和单个区域性（称作非特定区域性或默认区域性）的资源。默认区域性是应用程序的回退区域性；如果找不到已本地化的资源，则使用默认区域性的资源。每条轮辐均连接到一个附属程序集，该附属程序集包含单个区域性的资源，但不包含任何代码。

此模型有以下优点：

- 部署应用程序后，可针对新区域性以增量方式添加资源。因为特定于区域性资源的后续开发可能需要大量时间，这使你能够先发布主要应用程序，并在以后发布特定于区域性的资源。
- 可更新和更改应用程序的附属程序集，无需重新编译应用程序。
- 应用程序只需加载那些包含特定区域性所需资源的附属程序集。这可显著减少系统资源的使用。

但是，该模型也有以下缺点：

- 必须管理多组资源。
- 测试应用程序的初始成本增加，因为必须测试多种配置。请注意，从长远角度看，测试使用多个附属程序集测试多个核心应用程序会比测试并维护多个并行国际版本更轻松，成本更低。

资源命名约定

在打包应用程序的资源时，必须使用公共语言运行时所要求的资源命名约定对其进行命名。运行时可按其区域性名称标识资源。每个区域性均被赋予唯一名称，通常是与语言相关的两个小写字母的区域性名称和必要情况下，与国家或地区相关的两个大写子母的区域性名称的组合。子区域性名称跟在区域性名称后，以短划线 (-) 隔开。例如：ja-JP 表示日本日语，en-US 表示美国英语，de-DE 表示德国德语，de-AT 表示奥地利德语。请参阅 Windows 支持的语言/区域名称列表中的“语言标记” 列。列名遵循 BCP 47 定义的标准。

> **NOTE**
> 两字母区域性名称有一些例外，如表示中文（简体）的 `zh-Hans`。

有关详细信息，请参阅创建资源文件和创建附属程序集。

资源回退进程

用于打包和部署资源的中枢轮辐式模型可使用回退过程来定位相应的资源。如果应用程序请求不可用的已本地化资源，则公共语言运行时会在区域性层次结构中查找与用户应用程序请求最匹配的合适回退资源，并且只在最后迫不得已的情况下才引发异常。如果在层次结构的任意级别找到了合适的资源，运行时就会使用该资源。如果未找到资源，则将继续在下一级别中搜索。

若要提高查找性能，可将 NeutralResourcesLanguageAttribute 属性应用到主程序集，并向其传递将与主程序集一起使用的非特定语言的名称。

.NET Framework 资源回退进程

.NET Framework 资源回退进程包含以下步骤：

TIP

可以使用 <relativeBindForResources> 配置元素来优化资源回退过程和运行时针对资源程序集探测所依据的过程。有关详细信息，请参阅优化资源回退过程。

1. 运行时首先检查全局程序集缓存，找到与为应用程序请求的区域性匹配的程序集。

   全局程序集缓存可存储由许多应用程序共享的资源程序集。这使你无需在创建的每个应用程序的目录结构中包括特定的资源集。如果运行时找到了对程序集的引用，它将搜索程序集，查找请求的资源。如果在程序集中找到了该项，则使用请求的资源。如果找不到该项，将继续搜索。

2. 接下来，运行时将检查当前正在执行程序集的目录，查找与请求的区域性匹配的子目录。如果找到该子目录，它将搜索该子目录以找到请求的区域性的有效附属程序集。然后，运行时将搜索附属程序集，寻找请求的资源。如果在程序集中找到该资源，则使用它。如果找不到该资源，将继续搜索。

3. 接下来，运行时将查询 Windows Installer，以确定是否要按需安装附属程序集。如果是，它将处理安装，加载程序集，以及搜索它或搜索请求的资源。如果在程序集中找到该资源，则使用它。如果找不到该资源，将继续搜索。

4. 运行时引发 AppDomain.AssemblyResolve 事件以指示找不到附属程序集。如果选择对事件进行处理，事件处理程序可以返回对其资源将用于查找的附属程序集的引用。否则，事件处理程序将返回 `null`，搜索继续。

5. 接下来，运行时将再次搜索全局程序集缓存，这次是为了请求的区域性的父程序集。如果全局程序集缓存中存在父程序集，运行时将搜索程序集，寻找请求的资源。

   父区域性被定义为合适的回退区域性。将父区域性视为回退候选项，因为提供任何资源都比引发异常更可取。此过程还允许重复使用资源。仅当子无需本地化所请求的资源时，才应在父级别包含特定资源。例如，如果提供 `en`（非特定英语）的附属程序集：`en-GB`（英国英语）和 `en-US`（美国英语），则 `en` 附属程序集应包含公共术语，并且 `en-GB` 和 `en-US` 附属程序集可能只对那些不同的术语提供替代项。

6. 接下来，运行时将检查当前正在执行程序集的目录，查看是否包含父目录。如果存在父目录，则运行时将搜索该目录，寻找父区域性的有效附属程序集。如果找到该程序集，运行时将搜索程序集，寻找请求的资源。如果找到该资源，则使用它。如果找不到该资源，将继续搜索。

7. 接下来，运行时将查询 Windows Installer，以确定是否要按需安装父级附属程序集。如果是，它将处理安装，加载程序集，以及搜索它或搜索请求的资源。如果在程序集中找到该资源，则使用它。如果找不到该资源，将继续搜索。

8. 运行时引发 AppDomain.AssemblyResolve 事件以指示找不到合适的回退资源。如果选择对事件进行处理，事件处理程序可以返回对其资源将用于查找的附属程序集的引用。否则，事件处理程序将返回 `null`，搜索继续。

9. 接下来，如前前面三个步骤所述，运行时将通过许多可能的级别搜索父程序集。每个区域性只有一个父区域性（由 CultureInfo.Parent 属性定义），但一个父区域性可能还有其自己的父区域性。如果区域性的 Parent 属性返回 CultureInfo.InvariantCulture，父区域性搜索将停止；对于资源回退，固定区域性不被视为父区域性，也不被视为具有资源的区域性。

10. 如果区域性是最初指定的，且已搜索所有父级，但仍未找到资源，则使用默认（回退）区域性的资源。通常，默认区域性的资源包含在主应用程序集中。但是，可指定 NeutralResourcesLanguageAttribute 特性的 Location 属性的值为 Satellite，以指定资源的最终回退位置是附属程序集，而不是主程序集。

> **NOTE**
> 默认资源是唯一可与主程序集一起编译的资源。除非通过使用 NeutralResourcesLanguageAttribute 属性指定附属程序集，否则为最终回退(最终父级)。因此，建议在主程序集中始终包含一组默认资源。这有助于防止引发异常。通过包含默认资源文件，可为所有资源提供回退，并确保始终向用户呈现至少一种资源，即使该资源不是特定于区域性的。

11. 最后，如果运行时找不到默认(回退)区域性的资源，将引发 MissingManifestResourceException 或 MissingSatelliteAssemblyException 异常，指示找不到该资源。

例如，假定应用程序请求本地化为墨西哥西班牙语(`es-MX` 区域性)所需的资源。运行时将首先搜索全局程序集缓存，寻找匹配 `es-MX` 的程序集，但找不到。然后，运行时将搜索当前正在执行的程序集，寻找 `es-MX` 目录。如果失败，运行时将再次搜索全局程序集缓存，寻找反映相应回退区域性的父程序集 — 在本例中为 `es`(西班牙语)。如果找不到父程序集，运行时将针对 `es-MX` 区域性搜索父程序集的所有可能级别，直到它找到对应的资源。如果找不到资源，运行时将使用默认区域性的资源。

优化 .NET Framework 资源回退进程

在下列情况下，可以按运行时搜索附属程序集中的资源所依据的内容优化进程

- 附属程序集部署在与代码程序集相同的位置。如果代码程序集安装在全局程序集缓存中，则附属程序集也会安装到全局程序集缓存中。如果代码程序集安装在一个目录中，则附属程序集安装在该目录的特定于区域性的文件夹中。
- 附属程序集不会按需进行安装。
- 应用程序代码不会处理 AppDomain.AssemblyResolve 事件。

可通过在应用程序配置文件中包含 <relativeBindForResources> 元素并将其 `enabled` 属性设为 `true`，优化附属程序集探测，如下例所示。

```
<configuration>
   <runtime>
      <relativeBindForResources enabled="true" />
   </runtime>
</configuration>
```

优化的附属程序集探测是选择加入功能。也就是说，运行时遵循资源回退过程中记录的步骤，除非 <relativeBindForResources> 元素位于应用程序的配置文件中，并且其 `enabled` 属性已设为 `true`。如果出现这种情况，附属程序集探测过程将进行如下修改：

- 运行时使用父代码程序集位置来探测附属程序集。如果父程序集安装在全局程序集缓存中，则运行时在缓存，而不是在应用程序的目录中探测。如果父程序集安装在应用程序目录中，则运行时在应用程序目录，而不是在全局程序集缓存中探测。
- 运行时不会查询 Windows Installer，寻找附属程序集的按需安装。
- 如果特定资源程序集探测失败，运行时将不会引发 AppDomain.AssemblyResolve 事件。

.NET Core 资源回退进程

.NET Core 资源回退进程包含以下步骤：

1. 运行时尝试加载请求的区域性的附属程序集。
   - 检查当前正在执行程序集的目录，查找与请求的区域性匹配的子目录。如果找到该子目录，它将搜索该子目录以找到请求的区域性的有效附属程序集并加载该程序集。

> **NOTE**
> 在具有区分大小写的文件系统(即 Linux 和 macOS)的操作系统上，区域性名称子目录搜索区分大小写。子目录名称必须与 CultureInfo.Name 的大小写完全匹配(例如，`es` 或 `es-MX`)。

> **NOTE**
> 如果程序员从 AssemblyLoadContext 派生了自定义程序集加载上下文，则情况会很复杂。如果将正在执行程序集加载到自定义上下文中，则运行时会将附属程序集加载到自定义上下文中。相关详细信息超出了本文档所涉及的范围。请参阅AssemblyLoadContext。

- 如果未找到附属程序集，AssemblyLoadContext 引发 AssemblyLoadContext.Resolving 事件以指示找不到附属程序集。如果选择对事件进行处理，事件处理程序可以加载引用并返回对附属程序集的引用。
- 如果仍未找到附属程序集，则 AssemblyLoadContext 会导致 AppDomain 触发 AppDomain.AssemblyResolve 事件以指示找不到附属程序集。如果选择对事件进行处理，事件处理程序可以加载引用并返回对附属程序集的引用。

2. 如果找到附属程序集，则运行时将搜索请求的资源的附属程序集。如果在程序集中找到该资源，则使用它。如果找不到该资源，将继续搜索。

   > **NOTE**
   > 要在附属程序集中查找资源，运行时将搜索 ResourceManager 为当前 CultureInfo.Name 请求的资源文件。在资源文件中，它搜索所请求的资源名称。如果找不到上述任何一个，则资源将被视为未找到。

3. 接下来，运行时通过许多潜在级别搜索父区域性程序集，每次均重复步骤 1 和 2。

   父区域性被定义为合适的回退区域性。将父区域性视为回退候选项，因为提供任何资源都比引发异常更可取。此过程还允许重复使用资源。仅当子无需本地化所请求的资源时，才应在父级别包含特定资源。例如，如果提供 `en`(非特定英语)的附属程序集：`en-GB`(英国英语)和 `en-US`(美国英语)，则 `en` 附属程序集应包含公共术语，并且 `en-GB` 和 `en-US` 附属程序集只对那些不同的术语提供替代项。

   每个区域性只有一个父区域性(由 CultureInfo.Parent 属性定义)，但一个父区域性可能还有其自己的父区域性。当区域性的 Parent 属性返回 CultureInfo.InvariantCulture 时，父区域性搜索将停止。对于资源回退，固定区域性不被视为父区域性，也不被视为具有资源的区域性。

4. 如果区域性是最初指定的，且已搜索所有父级，但仍未找到资源，则使用默认(回退)区域性的资源。通常，默认区域性的资源包含在主应用程序集中。但是，可指定 Location 属性的值为 Satellite，以指示资源的最终回退位置是附属程序集，而不是主程序集。

   > **NOTE**
   > 默认资源是唯一可与主程序集一起编译的资源。除非通过使用 NeutralResourcesLanguageAttribute 属性指定附属程序集，否则为最终回退(最终父级)。因此，建议在主程序集中始终包含一组默认资源。这有助于防止引发异常。通过包含默认资源文件，可为所有资源提供回退，并确保始终向用户呈现至少一种资源，即使该资源不是特定于区域性的。

5. 最后，如果运行时找不到默认(回退)区域性的资源文件，将引发 MissingManifestResourceException 或 MissingSatelliteAssemblyException 异常，指示找不到该资源。如果找到资源文件但是请求的资源不存在，则请求返回 `null`。

最终回退到附属程序集

可选择从主程序集中删除资源，并指定运行时应加载对应于特定区域性的附属程序集中的最终回退资源。若要控制回退进程，请使用 NeutralResourcesLanguageAttribute(String, UltimateResourceFallbackLocation) 构造函数并提供 UltimateResourceFallbackLocation 参数的值，用于指定 Resource Manager 是应从主程序集还是应从附属程序集中提取回退资源。

下面的 .NET Framework 示例使用 NeutralResourcesLanguageAttribute 属性将应用程序回退资源存储在法语 (`fr`) 语言的附属程序集中。本示例介绍了两个基于文本的资源文件，这两个文件用于定义名为 `Greeting` 的单个字符串资源。第一个文件 resources.fr.txt 包含法语资源。

```
Greeting=Bon jour!
```

第二个文件 resources.ru.txt 包含俄语资源。

```
Greeting=Добрый день
```

从命令行运行资源文件生成器 (resgen.exe) 可将这两个文件编译为 .resources 文件。对于法语资源，命令为：

```
resgen.exe resources.fr.txt
```

对于俄语资源，命令为：

```
resgen.exe resources.ru.txt
```

对于法语资源，从命令行运行程序集连接器 (al.exe)，将 .resources 文件嵌入动态链接库，如下所示：

```
al /t:lib /embed:resources.fr.resources /culture:fr /out:fr\Example1.resources.dll
```

而对于俄语资源，则为如下所示：

```
al /t:lib /embed:resources.ru.resources /culture:ru /out:ru\Example1.resources.dll
```

应用程序源代码位于名为 Example1.cs 或 Example1.vb 的文件中。它包括 NeutralResourcesLanguageAttribute 属性，以指示默认应用程序资源位于 fr 子目录中。它可实例化 Resource Manager，检索 `Greeting` 资源的值，并将其显示到控制台。

```
using System;
using System.Reflection;
using System.Resources;

[assembly:NeutralResourcesLanguage("fr", UltimateResourceFallbackLocation.Satellite)]

public class Example
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("resources",
                                               typeof(Example).Assembly);
      string greeting = rm.GetString("Greeting");
      Console.WriteLine(greeting);
   }
}
```

```
Imports System.Reflection
Imports System.Resources

<Assembly: NeutralResourcesLanguage("fr", UltimateResourceFallbackLocation.Satellite)>
Module Example
    Public Sub Main()
        Dim rm As New ResourceManager("resources", GetType(Example).Assembly)
        Dim greeting As String = rm.GetString("Greeting")
        Console.WriteLine(greeting)
    End Sub
End Module
```

然后，可以从命令行编译 C# 源代码，如下所示：

```
csc Example1.cs
```

用于 Visual Basic 编译器的命令十分相似：

```
vbc Example1.vb
```

由于没有嵌入在主程序集中的资源，因此无需使用 `/resource` 切换进行编译。

当从不是俄语的任何系统运行示例时，它将显示以下输出：

```
Bon jour!
```

建议的打包替代项

由于时间或预算约束，可能无法为应用程序支持的每个子区域性均创建一组资源。但可以为所有相关子区域性可用的父区域性创建单个附属程序集。例如，可以提供单个英语附属程序集 (en)，请求特定于区域的英语资源的用户将检索该程序集，并且为请求特定于区域的德语资源的用户创建单个德语附属程序集 (de)。例如，对德国德语 (de-DE)、奥地利德语 (de-AT) 和瑞士德语 (de-CH) 的请求均会回退到德语附属程序集 (de)。默认资源是最终回退资源，因而应是大多数应用程序用户将请求的资源，因此应仔细选择这些资源。此方法可部署区域性特定性较低，但可显著减少应用程序本地化成本的资源。

请参阅

- .NET 应用中的资源
- 全局程序集缓存
- 创建资源文件
- 创建附属程序集

检索.NET 应用中的资源

2021/11/16 •

使用 .NET 应用中的本地化资源时，最好用主程序集打包默认或非特定区域性的资源，并为应用支持的每种语言或区域性单独创建附属程序集。可以使用下一节中介绍的 ResourceManager 类访问已命名的资源。如果选择不在主程序集和附属程序集中嵌入资源，也可以按本文后面的从 .resources 文件中检索资源一节中所述直接访问二进制 .resources 文件。

从程序集中检索资源

ResourceManager 类提供对运行时资源的访问权限。使用 ResourceManager.GetString 方法检索字符串资源和 ResourceManager.GetObject 或使用 ResourceManager.GetStream 方法检索非字符串资源。每个方法都有两种重载：

- 单一参数是包含资源名称的字符串的重载。该方法尝试为当前线程区域性检索资源。有关详细信息，请参阅 GetString(String)、GetObject(String)和 GetStream(String) 方法。
- 具有两个参数的重载：一个字符串包含资源名称，一个 CultureInfo 对象表示要对其检索资源的区域性。如果找不到该区域性的资源集，资源管理器将使用回退规则检索相应的资源。有关详细信息，请参阅 GetString(String, CultureInfo)、GetObject(String, CultureInfo)和 GetStream(String, CultureInfo) 方法。

资源管理器使用资源回退进程控制应用检索区域性特定资源的方式。有关详细信息，请参阅打包和部署资源中的“资源回退进程”一节。有关实例化 ResourceManager 对象的详细信息，请参阅 ResourceManager 类主题中的“实例化 ResourceManager 对象”一节。

检索字符串数据示例

下面的示例调用 GetString(String) 方法检索当前 UI 区域性的字符串资源。它包括英语(美国)区域性的非特定字符串资源和法语(法国)和俄语(俄罗斯)区域性的本地化资源。下面的英语(美国)资源位于名为 Strings.txt 的文件中：

```
TimeHeader=The current time is
```

法语(法国)资源位于名为 Strings.fr-FR.txt 的文件中：

```
TimeHeader=L'heure actuelle est
```

俄语(俄罗斯)资源位于名为 Strings.ru-RU.txt 的文件中：

```
TimeHeader=Текущее время —
```

此示例的源代码(在代码的 C# 版本中位于名为 GetString.cs 的文件中，在 Visual Basic 版本中位于名为 GetString.vb 的文件中)定义包含四种区域性名称(资源可用的三种区域性和西班牙语(西班牙)区域性)的字符串数组。一个随机执行 5 次的循环选择其中一种区域性并将其分配到 Thread.CurrentCulture 和 CultureInfo.CurrentUICulture 属性。然后调用 GetString(String) 方法检索与一天当中的时间一起显示的本地化的字符串。

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguageAttribute("en-US")]

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "fr-FR", "ru-RU", "es-ES" };
      Random rnd = new Random();
      ResourceManager rm = new ResourceManager("Strings",
                               typeof(Example).Assembly);

      for (int ctr = 0; ctr <= cultureNames.Length; ctr++) {
         string cultureName = cultureNames[rnd.Next(0, cultureNames.Length)];
         CultureInfo culture = CultureInfo.CreateSpecificCulture(cultureName);
         Thread.CurrentThread.CurrentCulture = culture;
         Thread.CurrentThread.CurrentUICulture = culture;

         Console.WriteLine("Current culture: {0}", culture.NativeName);
         string timeString = rm.GetString("TimeHeader");
         Console.WriteLine("{0} {1:T}\n", timeString, DateTime.Now);
      }
   }
}
// The example displays output like the following:
//    Current culture: English (United States)
//    The current time is 9:34:18 AM
//
//    Current culture: Español (España, alfabetización internacional)
//    The current time is 9:34:18
//
//    Current culture: русский (Россия)
//    Текущее время — 9:34:18
//
//    Current culture: français (France)
//    L'heure actuelle est 09:34:18
//
//    Current culture: русский (Россия)
//    Текущее время — 9:34:18
```

```
Imports System.Globalization
Imports System.Resources
Imports System.Threading

<Assembly: NeutralResourcesLanguageAttribute("en-US")>

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-US", "fr-FR", "ru-RU", "es-ES"}
        Dim rnd As New Random()
        Dim rm As New ResourceManager("Strings", GetType(Example).Assembly)

        For ctr As Integer = 0 To cultureNames.Length
            Dim cultureName As String = cultureNames(rnd.Next(0, cultureNames.Length))
            Dim culture As CultureInfo = CultureInfo.CreateSpecificCulture(cultureName)
            Thread.CurrentThread.CurrentCulture = culture
            Thread.CurrentThread.CurrentUICulture = culture

            Console.WriteLine("Current culture: {0}", culture.NativeName)
            Dim timeString As String = rm.GetString("TimeHeader")
            Console.WriteLine("{0} {1:T}", timeString, Date.Now)
            Console.WriteLine()
        Next
    End Sub
End Module
' The example displays output similar to the following:
'    Current culture: English (United States)
'    The current time is 9:34:18 AM
'
'    Current culture: Español (España, alfabetización internacional)
'    The current time is 9:34:18
'
'    Current culture: русский (Россия)
'    Текущее время — 9:34:18
'
'    Current culture: français (France)
'    L'heure actuelle est 09:34:18
'
'    Current culture: русский (Россия)
'    Текущее время — 9:34:18
```

以下批处理 (.bat) 文件编译该示例，并在相应的目录中生成附属程序集。为 C# 语言和编译器提供了命令。对于 Visual Basic，将 `csc` 更改为 `vbc`，并将 `GetString.cs` 更改为 `GetString.vb`。

```
resgen strings.txt
csc GetString.cs -resource:strings.resources

resgen strings.fr-FR.txt
md fr-FR
al -embed:strings.fr-FR.resources -culture:fr-FR -out:fr-FR\GetString.resources.dll

resgen strings.ru-RU.txt
md ru-RU
al -embed:strings.ru-RU.resources -culture:ru-RU -out:ru-RU\GetString.resources.dll
```

当前 UI 区域性为西班牙语(西班牙)时，请注意该示例会显示英语语言资源，因为西班牙语语言资源不可用，而英语是该示例的默认区域性。

检索对象数据示例

可以使用 GetObject 和 GetStream 方法检索对象数据。这包括基元数据类型、可序列化对象和以二进制格式存储的对象(如图像)。

下面的示例使用 GetStream(String) 方法检索应用启动初始窗口中使用的位图。以下源代码位于名为

CreateResources.cs 的文件中(C# 版本)或位于名为 CreateResources.vb 的文件中(Visual Basic 版本)，它能生成包含序列化图像的 .resx 文件。在这种情况下，图片从一个名为 SplashScreen.jpg 的文件中加载；可以修改文件名以替换你自己的图像。

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Resources;

public class Example
{
   public static void Main()
   {
      Bitmap bmp = new Bitmap(@".\SplashScreen.jpg");
      MemoryStream imageStream = new MemoryStream();
      bmp.Save(imageStream, ImageFormat.Jpeg);

      ResXResourceWriter writer = new ResXResourceWriter("AppResources.resx");
      writer.AddResource("SplashScreen", imageStream);
      writer.Generate();
      writer.Close();
   }
}
```

```
Imports System.Drawing
Imports System.Drawing.Imaging
Imports System.IO
Imports System.Resources

Module Example
    Public Sub Main()
        Dim bmp As New Bitmap(".\SplashScreen.jpg")
        Dim imageStream As New MemoryStream()
        bmp.Save(imageStream, ImageFormat.Jpeg)

        Dim writer As New ResXResourceWriter("AppResources.resx")
        writer.AddResource("SplashScreen", imageStream)
        writer.Generate()
        writer.Close()
    End Sub
End Module
```

以下代码将检索该资源，并在 PictureBox 控件中显示图像。

```
using System;
using System.Drawing;
using System.IO;
using System.Resources;
using System.Windows.Forms;

public class Example
{
   public static void Main()
   {
      ResourceManager rm = new ResourceManager("AppResources", typeof(Example).Assembly);
      Bitmap screen = (Bitmap) Image.FromStream(rm.GetStream("SplashScreen"));

      Form frm = new Form();
      frm.Size = new Size(300, 300);

      PictureBox pic = new PictureBox();
      pic.Bounds = frm.RestoreBounds;
      pic.BorderStyle = BorderStyle.Fixed3D;
      pic.Image = screen;
      pic.SizeMode = PictureBoxSizeMode.StretchImage;

      frm.Controls.Add(pic);
      pic.Anchor = AnchorStyles.Top | AnchorStyles.Bottom |
                   AnchorStyles.Left | AnchorStyles.Right;

      frm.ShowDialog();
   }
}
```

```
Imports System.Drawing
Imports System.IO
Imports System.Resources
Imports System.Windows.Forms

Module Example
    Public Sub Main()
        Dim rm As New ResourceManager("AppResources", GetType(Example).Assembly)
        Dim screen As Bitmap = CType(Image.FromStream(rm.GetStream("SplashScreen")), Bitmap)

        Dim frm As New Form()
        frm.Size = new Size(300, 300)

        Dim pic As New PictureBox()
        pic.Bounds = frm.RestoreBounds
        pic.BorderStyle = BorderStyle.Fixed3D
        pic.Image = screen
        pic.SizeMode = PictureBoxSizeMode.StretchImage

        frm.Controls.Add(pic)
        pic.Anchor = AnchorStyles.Top Or AnchorStyles.Bottom Or
                     AnchorStyles.Left Or AnchorStyles.Right

        frm.ShowDialog()
    End Sub
End Module
```

可以使用以下批处理文件生成 C# 示例。对于 Visual Basic，将 `csc` 更改为 `vbc`，并将源代码文件的扩展名由 `.cs` 更改为 `.vb`。

```
csc CreateResources.cs
CreateResources

resgen AppResources.resx

csc GetStream.cs -resource:AppResources.resources
```

下面的示例使用 ResourceManager.GetObject(String) 方法反序列化一个自定义对象。该示例包含一个名为 UIElements.cs（对于 Visual Basic 则为 UIElements.vb）的源代码文件，用于定义以下名为 `PersonTable` 的结构。此结构应由显示表列的本地化名称的常规表显示例程使用。请注意，`PersonTable` 结构标有 SerializableAttribute 属性。

```
using System;

[Serializable] public struct PersonTable
{
   public readonly int nColumns;
   public readonly string column1;
   public readonly string column2;
   public readonly string column3;
   public readonly int width1;
   public readonly int width2;
   public readonly int width3;

   public PersonTable(string column1, string column2, string column3,
                  int width1, int width2, int width3)
   {
      this.column1 = column1;
      this.column2 = column2;
      this.column3 = column3;
      this.width1 = width1;
      this.width2 = width2;
      this.width3 = width3;
      this.nColumns = typeof(PersonTable).GetFields().Length / 2;
   }
}
```

```
<Serializable> Public Structure PersonTable
    Public ReadOnly nColumns As Integer
    Public Readonly column1 As String
    Public ReadOnly column2 As String
    Public ReadOnly column3 As String
    Public ReadOnly width1 As Integer
    Public ReadOnly width2 As Integer
    Public ReadOnly width3 As Integer

    Public Sub New(column1 As String, column2 As String, column3 As String,
                   width1 As Integer, width2 As Integer, width3 As Integer)
        Me.column1 = column1
        Me.column2 = column2
        Me.column3 = column3
        Me.width1 = width1
        Me.width2 = width2
        Me.width3 = width3
        Me.nColumns = Me.GetType().GetFields().Count \ 2
    End Sub
End Structure
```

下面的代码来自名为 CreateResources.cs（对于 Visual Basic 则为 CreateResources.vb）的文件，该代码创建一个名为 UIResources.resx 的 XML 资源文件，该文件存储有表标题和包含已针对英语语言本地化的应用的信息的 `PersonTable` 对象。

```
using System;
using System.Resources;

public class CreateResource
{
   public static void Main()
   {
      PersonTable table = new PersonTable("Name", "Employee Number",
                                          "Age", 30, 18, 5);
      ResXResourceWriter rr = new ResXResourceWriter(@".\UIResources.resx");
      rr.AddResource("TableName", "Employees of Acme Corporation");
      rr.AddResource("Employees", table);
      rr.Generate();
      rr.Close();
   }
}
```

```
Imports System.Resources

Module CreateResource
    Public Sub Main()
        Dim table As New PersonTable("Name", "Employee Number", "Age", 30, 18, 5)
        Dim rr As New ResXResourceWriter(".\UIResources.resx")
        rr.AddResource("TableName", "Employees of Acme Corporation")
        rr.AddResource("Employees", table)
        rr.Generate()
        rr.Close()
    End Sub
End Module
```

下面的代码位于名为 GetObject.cs (GetObject.vb) 的源代码文件中，然后检索资源并将其显示在控制台上。

```
using System;
using System.Resources;

[assembly: NeutralResourcesLanguageAttribute("en")]

public class Example
{
   public static void Main()
   {
      string fmtString = String.Empty;
      ResourceManager rm = new ResourceManager("UIResources", typeof(Example).Assembly);
      string title = rm.GetString("TableName");
      PersonTable tableInfo = (PersonTable) rm.GetObject("Employees");

      if (! String.IsNullOrEmpty(title)) {
         fmtString = "{0," + ((Console.WindowWidth + title.Length) / 2).ToString() + "}";
         Console.WriteLine(fmtString, title);
         Console.WriteLine();
      }

      for (int ctr = 1; ctr <= tableInfo.nColumns; ctr++) {
         string columnName = "column"  + ctr.ToString();
         string widthName = "width" + ctr.ToString();
         string value = tableInfo.GetType().GetField(columnName).GetValue(tableInfo).ToString();
         int width = (int) tableInfo.GetType().GetField(widthName).GetValue(tableInfo);
         fmtString = "{0,-" + width.ToString() + "}";
         Console.Write(fmtString, value);
      }
      Console.WriteLine();
   }
}
```

```
Imports System.Resources

<Assembly: NeutralResourcesLanguageAttribute("en")>

Module Example
    Public Sub Main()
        Dim fmtString As String = String.Empty
        Dim rm As New ResourceManager("UIResources", GetType(Example).Assembly)
        Dim title As String = rm.GetString("TableName")
        Dim tableInfo As PersonTable = DirectCast(rm.GetObject("Employees"), PersonTable)

        If Not String.IsNullOrEmpty(title) Then
            fmtString = "{0," + ((Console.WindowWidth + title.Length) \ 2).ToString() + "}"
            Console.WriteLine(fmtString, title)
            Console.WriteLine()
        End If

        For ctr As Integer = 1 To tableInfo.nColumns
            Dim columnName As String = "column" + ctr.ToString()
            Dim widthName As String = "width" + ctr.ToString()
            Dim value As String = CStr(tableInfo.GetType().GetField(columnName).GetValue(tableInfo))
            Dim width As Integer = CInt(tableInfo.GetType().GetField(widthName).GetValue(tableInfo))
            fmtString = "{0,-" + width.ToString() + "}"
            Console.Write(fmtString, value)
        Next
        Console.WriteLine()
    End Sub
End Module
```

可以生成必要的资源文件和程序集，并通过执行以下批处理文件运行该应用。必须使用 `/r` 选项提供具有对 UIElements.dll 的引用的 Resgen.exe，以便其能够访问有关 `PersonTable` 结构的信息。如果使用 C#，请将 `vbc` 编译器名称替换为 `csc`，并将 `.vb` 扩展名替换为 `.cs`。

```
vbc -t:library UIElements.vb
vbc CreateResources.vb -r:UIElements.dll
CreateResources

resgen UIResources.resx  -r:UIElements.dll
vbc GetObject.vb -r:UIElements.dll -resource:UIResources.resources

GetObject.exe
```

附属程序集的版本支持

默认情况下，ResourceManager 对象检索请求的资源时，会寻找版本号与主程序集版本号相匹配的附属程序集。部署应用后，建议更新主程序集或特定资源附属程序集。.NET Framework 提供对主程序集和附属程序集的版本控制支持。

SatelliteContractVersionAttribute 属性提供对主程序集的版本控制支持。在应用的主程序集上指定此属性，无需更新主程序集的附属程序集即可更新和重新部署主程序集。更新主程序集后，递增主程序集的版本号，但附属协定版本号保持不变。资源管理器检索请求的资源时，会加载此属性指定的附属程序集版本。

发行者策略程序集提供对附属程序集的版本控制支持。你可以更新并重新部署附属程序集，而不用更新主程序集。更新附属程序集后，递增其版本号，并将其附带到发行者策略程序集中。在发行者策略程序集中，指定新附属程序集为后向兼容其之前版本。资源管理器会使用 SatelliteContractVersionAttribute 属性确定附属程序集的版本，但程序集加载程序将绑定到发行者策略所指定的附属程序集版本。有关发行者策略程序集的详细信息，请参阅 创建发行者策略文件。

若要启用完全的程序集版本控制支持，建议你在 全局程序集缓存 中部署具有强名称的程序集，并将不具有强名称的程序集部署在应用程序目录中。若在应用程序目录中部署具有强名称的程序集，则无法在更新程序集时递

增附属程序集的版本号。相反，必须在使用更新的代码替换现有代码处执行就地更新，并保持相同的版本号。例如，若要使用完全指定的程序集名称 "myApp.resources, Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a" 更新版本 1.0.0.0 的附属程序集，请使用已编译同一个完全指定的程序集名称 "myApp.resources, Version=1.0.0.0, Culture=de, PublicKeyToken=b03f5f11d50a3a" 的更新的 myApp.resources.dll 来覆盖它。请注意，在附属程序集文件上使用就地更新会使应用难以准确确定附属程序集的版本。

有关程序集版本控制的详细信息，请参阅 程序集版本控制 和 运行时如何定位程序集。

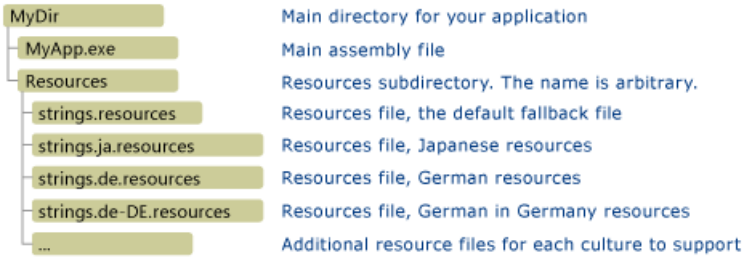
从 .resources 文件中检索资源

如果选择不在附属程序集中部署资源，你仍可以使用 ResourceManager 对象直接访问 .resources 文件中的资源。要执行此操作，必须正确部署.resources 文件。然后使用 ResourceManager.CreateFileBasedResourceManager 方法实例化 ResourceManager 对象，并指定包含独立 .resources 文件的目录。

部署 .resources 文件

将 .resources 文件嵌入应用程序程序集和附属程序集后，每个附属程序集都具有相同的文件名，但被放在反射附属程序集区域性的子目录中。与此相反，从 .resources 文件直接访问资源时，可以将所有 .resources 文件放在单一目录（通常为应用程序目录的子目录）中。应用的默认 .resources 文件名称仅包含一个根名称，不带有其区域性的指示（例如 strings.resources）。每个本地化的区域性资源存储在名称包含根名称，后带有区域性标记所组成的文件中（例如 strings.ja.resources 或 strings.de-DE.resources）。

下图显示资源文件应被放置在目录结构中的何处。它还提供了.resource 文件的命名约定。



使用资源管理器

创建资源并将其放置在相应的目录中后，通过调用 ResourceManager 方法创建 CreateFileBasedResourceManager(String, String, Type) 对象以使用资源。第一个参数指定应用的默认 .resources 文件的根名称（在上一节的示例中为 "strings"）。第二个参数指定的资源的位置（上一个示例中为 "Resources"）。第三个参数指定要使用的 ResourceSet 实现。如果第三个参数为 `null`，则使用默认运行时 ResourceSet 。

> **NOTE**
> 请勿使用独立 .resources 文件部署 ASP.NET 应用。这可能会导致锁定问题并破坏 XCOPY 部署。建议部署附属程序集中的 ASP.NET 资源。有关更多信息，请参见 ASP.NET Web Page Resources Overview。

实例化 ResourceManager 对象后，使用前文所述的 GetString、GetObject和 GetStream 方法检索资源。但是，直接从 .resources 文件中检索资源与从程序集中检索嵌入的资源有所不同。从 .resources 文件中检索资源时，GetString(String)、GetObject(String)和 GetStream(String) 方法总是忽略当前区域性检索默认区域性的资源。若要检索应用的当前区域性资源或指定区域性的资源，必须调用 GetString(String, CultureInfo)、GetObject(String, CultureInfo)或 GetStream(String, CultureInfo) 方法并指定要检索资源的区域性。若要检索当前区域性的资源，请将 CultureInfo.CurrentCulture 属性的值指定为 `culture` 参数。如果资源管理器无法检索 `culture` 的资源，则使用标准资源回退规则检索相应的资源。

示例

下面的示例说明资源管理器如何直接从 .resources 文件中检索资源。此示例由三个基于文本的资源文件组成，

区域性分别为英语(美国)、法语(法国)和俄语(俄罗斯)。英语(美国)为示例的默认区域性。其资源存储在以下名为 Strings.txt 的文件中：

```
Greeting=Hello
Prompt=What is your name?
```

法语(法国) 区域性的资源存储在以下名为 Strings.fr-FR.txt 的文件中：

```
Greeting=Bon jour
Prompt=Comment vous appelez-vous?
```

俄语(俄罗斯) 区域性的资源存储在以下名为 Strings.ru-RU.txt 的文件中：

```
Greeting=Здравствуйте
Prompt=Как вас зовут?
```

以下是该实例的源代码。该示例为英语(美国)、英语(加拿大)、法语(法国)和俄语(俄罗斯)区域性实例化 CultureInfo 对象，并将以上每一种语言作为当前区域性。ResourceManager.GetString(String, CultureInfo) 方法提供 CultureInfo.CurrentCulture 属性的值作为 `culture` 参数来检索相应的区域性指定资源。

```
using System;
using System.Globalization;
using System.Resources;
using System.Threading;

[assembly: NeutralResourcesLanguage("en-US")]

public class Example
{
   public static void Main()
   {
      string[] cultureNames = { "en-US", "en-CA", "ru-RU", "fr-FR" };
      ResourceManager rm = ResourceManager.CreateFileBasedResourceManager("Strings", "Resources", null);

      foreach (var cultureName in cultureNames) {
         Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName);
         string greeting = rm.GetString("Greeting", CultureInfo.CurrentCulture);
         Console.WriteLine("\n{0}!", greeting);
         Console.Write(rm.GetString("Prompt", CultureInfo.CurrentCulture));
         string name = Console.ReadLine();
         if (! String.IsNullOrEmpty(name))
            Console.WriteLine("{0}, {1}!", greeting, name);
      }
      Console.WriteLine();
   }
}
// The example displays output like the following:
//       Hello!
//       What is your name? Dakota
//       Hello, Dakota!
//
//       Hello!
//       What is your name? Koani
//       Hello, Koani!
//
//       Здравствуйте!
//       Как вас зовут?Samuel
//       Здравствуйте, Samuel!
//
//       Bon jour!
//       Comment vous appelez-vous?Yiska
//       Bon jour, Yiska!
```

```
Imports System.Globalization
Imports System.Resources
Imports System.Threading

<Assembly: NeutralResourcesLanguageAttribute("en-US")>

Module Example
    Public Sub Main()
        Dim cultureNames() As String = {"en-US", "en-CA", "ru-RU", "fr-FR"}
        Dim rm As ResourceManager = ResourceManager.CreateFileBasedResourceManager("Strings", "Resources",
Nothing)

        For Each cultureName In cultureNames
            Console.WriteLine()
            Thread.CurrentThread.CurrentCulture = CultureInfo.CreateSpecificCulture(cultureName)
            Dim greeting As String = rm.GetString("Greeting", CultureInfo.CurrentCulture)
            Console.WriteLine("{0}!", greeting)
            Console.Write(rm.GetString("Prompt", CultureInfo.CurrentCulture))
            Dim name As String = Console.ReadLine()
            If Not String.IsNullOrEmpty(name) Then
                Console.WriteLine("{0}, {1}!", greeting, name)
            End If
        Next
        Console.WriteLine()
    End Sub
End Module
' The example displays output like the following:
'       Hello!
'       What is your name? Dakota
'       Hello, Dakota!
'
'       Hello!
'       What is your name? Koani
'       Hello, Koani!
'
'       Здравствуйте!
'       Как вас зовут?Samuel
'       Здравствуйте, Samuel!
'
'       Bon jour!
'       Comment vous appelez-vous?Yiska
'       Bon jour, Yiska!
```

可以通过运行以下批处理文件编译该示例的 C# 版本。如果使用 Visual Basic，请将 `csc` 替换为 `vbc`，并将 `.cs` 扩展名替换为 `.vb`。

```
md Resources
resgen Strings.txt Resources\Strings.resources
resgen Strings.fr-FR.txt Resources\Strings.fr-FR.resources
resgen Strings.ru-RU.txt Resources\Strings.ru-RU.resources

csc Example.cs
```

请参阅

- ResourceManager
- .NET 应用中的资源
- 打包和部署资源
- 运行时如何定位程序集

.NET 中的辅助角色服务

2021/11/16 •

创建长时间运行的服务的原因有很多，例如：

- 处理 CPU 密集型数据。
- 在后台对工作项进行排队。
- 按计划执行基于时间的操作。

后台服务处理通常不涉及用户界面 (UI)，但可以围绕它们来构建 UI。在早期使用 .NET Framework 时，Windows 开发人员可能基于这些原因创建 Windows 服务。使用 .NET，你可以使用 BackgroundService —（它是 IHostedService 的实现）或实现自己的服务。

使用 .NET，你将不再局限于 Windows。你可以开发跨平台的后台服务。托管服务支持日志记录、配置和依赖项注入 (DI)。它们是库扩展套件的一部分，这意味着它们是所有使用通用主机的 .NET 工作负载的基础。

术语

有许多术语被误用为同义词。在本部分，我们提供了其中一些术语的定义，以使这些术语的意图更为直观。

- 后台服务：引用 BackgroundService 类型。
- 托管服务：实现 IHostedService 或引用 IHostedService 本身。
- 长时间运行的服务：持续运行的任何服务。
- Windows 服务：Windows 服务基础结构，最初以 .NET Framework 为中心，但现在可通过 .NET 访问。
- 辅助角色服务：引用辅助角色服务模板。

辅助角色服务模板

辅助角色服务模板可用于 .NET CLI 和 Visual Studio。有关详细信息，请参阅 .NET CLI，`dotnet new worker` - 模板。模板由 `Program` 和 `Worker` 类组成。

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace App.WorkerService
{
    public class Program
    {
        public static void Main(string[] args)
        {
            CreateHostBuilder(args).Build().Run();
        }

        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureServices((hostContext, services) =>
                {
                    services.AddHostedService<Worker>();
                });
    }
}
```

前面的 `Program` 类：

- 创建默认 IHostBuilder。
- 调用 ConfigureServices 以使用 AddHostedService 将 `Worker` 类添加为托管服务。
- 从生成器生成 IHost。
- 在运行应用的 `host` 实例上调用 `Run`。

可以使用顶级语句重写模板中的 Program.cs 文件：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using App.WorkerService;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((hostContext, services) =>
    {
        services.AddHostedService<Worker>();
    })
    .Build();

await host.RunAsync();
```

这在功能上等效于原始模板。有关 C# 9 功能的详细信息，请参阅 C# 9.0 中的新增功能。

对于 `Worker`，模板提供了一个简单的实现。

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.WorkerService
{
    public class Worker : BackgroundService
    {
        private readonly ILogger<Worker> _logger;

        public Worker(ILogger<Worker> logger)
        {
            _logger = logger;
        }

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogInformation("Worker running at: {time}", DateTimeOffset.Now);
                try
                {
                    await Task.Delay(1000, stoppingToken);
                }
                catch (OperationCanceledException)
                {
                    break;
                }
            }
        }
    }
}
```

前面的 `Worker` 类是 BackgroundService 的子类，用于实现 IHostedService。BackgroundService 是一个 `abstract class`，需要子类来实现 BackgroundService.ExecuteAsync(CancellationToken)。在模板实现中，`ExecuteAsync` 每秒循环一次，记录当前日期和时间，直到进程收到取消信号。

项目文件

辅助角色服务模板依赖于以下项目文件 `Sdk`：

```
<Project Sdk="Microsoft.NET.Sdk.Worker">
```

有关更多信息，请参阅 .NET 项目 SDK。

NuGet 包

基于辅助角色服务模板的应用使用 `Microsoft.NET.Sdk.Worker` SDK，并且具有对 Microsoft.Extensions.Hosting 包的显式包引用。

容器和云的可采用性

对于大多数新式 .NET 工作负载，容器是一个可行的选择。从 Visual Studio 中的辅助角色服务模板创建长时间运行的服务时，可以选择加入 Docker 支持。这将创建一个 Dockerfile，可用于容器化 .NET 应用。*Dockerfile* 是生成映像的一组指令。对于 .NET 应用，Dockerfile 通常位于解决方案文件旁边目录的根目录中。

```
# See https://aka.ms/containerfastmode to understand how Visual Studio uses this
# Dockerfile to build your images for faster debugging.

FROM mcr.microsoft.com/dotnet/runtime:5.0 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["background-service/App.WorkerService.csproj", "background-service/"]
RUN dotnet restore "background-service/App.WorkerService.csproj"
COPY . .
WORKDIR "/src/background-service"
RUN dotnet build "App.WorkerService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.WorkerService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.WorkerService.dll"]
```

前面的 Dockerfile 步骤包括：

- 将 `mcr.microsoft.com/dotnet/runtime:5.0` 中的基映像设置为别名 `base`。
- 将工作目录更改为 /app。
- 设置 `mcr.microsoft.com/dotnet/sdk:5.0` 映像中的 `build` 别名。
- 将工作目录更改为 /src。
- 复制内容并发布 .NET 应用：
  - 应用使用 `dotnet publish` 命令发布。
- 从 `mcr.microsoft.com/dotnet/runtime:5.0`（`base` 别名）中继 .NET SDK 映像。
- 从 /publish 复制已发布的生成输出。
- 定义委托给 `dotnet App.BackgroundService.dll` 的入口点。

> **TIP**
>
> `mcr.microsoft.com` 中的 MCR 代表“Microsoft Container Registry”，是 Microsoft 官方 Docker 中心的联合容器目录。Microsoft 联合容器目录一文包含更多详细信息。

将 Docker 作为 .NET 辅助角色服务的部署策略时，项目文件中有几个注意事项：

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>App.WorkerService</RootNamespace>
    <DockerDefaultTargetOS>Linux</DockerDefaultTargetOS>
    <Nullable>enable</Nullable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="5.0.0" />
    <PackageReference Include="Microsoft.VisualStudio.Azure.Containers.Tools.Targets" Version="1.10.14" />
  </ItemGroup>
</Project>
```

在上一个项目文件中，`<DockerDefaultTargetOS>` 元素将 `Linux` 指定为其目标。若要面向 Windows 容器，请改用 `Windows`。从模板中选择 Docker 支持时，`Microsoft.VisualStudio.Azure.Containers.Tools.Targets` NuGet 包会自

动添加为包引用。

有关在 Docker 上使用 .NET 的详细信息，请参阅教程：容器化 .NET 应用。有关部署到 Azure 的详细信息，请参阅教程：将辅助角色服务部署到 Azure。

托管服务扩展性

IHostedService 接口定义两种方法：

- IHostedService.StartAsync(CancellationToken)
- IHostedService.StopAsync(CancellationToken)

这两种方法充当生命周期方法 - 它们分别在主机启动和停止事件期间被调用。

> **IMPORTANT**
> 接口充当 AddHostedService<THostedService>(IServiceCollection) 扩展方法的泛型类型参数约束，这意味着只允许实现。可以将提供的 BackgroundService 与子类一起使用，或完全实现自己的子类。

另请参阅

- BackgroundService 子类教程：
  - 在 .NET 中创建队列服务
  - 在 .NET 中的 `BackgroundService` 内使用作用域服务
  - 在 .NET 中使用 `BackgroundService` 创建 Windows 服务
- 自定义 IHostedService 实现：
  - 在 .NET 中实现 `IHostedService` 接口

创建队列服务

2021/11/16 •

队列服务是很好的长时间运行服务示例，在此示例中，工作项可以排队并按顺序处理，因为之前的工作项已完成。依靠辅助角色服务模板，你将在 BackgroundService 上构建一些新功能。

在本教程中，你将了解如何执行以下操作：

- 创建队列服务。
- 将工作委托给任务队列。
- 从 IHostApplicationLifetime 事件注册控制台密钥侦听器。

> **TIP**
> 所有".NET 中的辅助角色"示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的辅助角色。

先决条件

- .NET 5.0 SDK 或更高版本
- .NET 集成开发环境 (IDE)
  - 随意使用 Visual Studio

创建新项目

若要使用 Visual Studio 创建新的辅助角色服务项目，请选择"文件" > "新建" > "项目..." 。从"创建新项目"对话框搜索"辅助角色服务"，并选择辅助角色服务模板。如果你想要使用 .NET CLI，请在工作目录中打开你最喜欢的终端。运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

> **TIP**
> 如果使用 Visual Studio Code，则可以从集成终端运行 .NET CLI 命令。有关详细信息，请参阅 Visual Studio Code：集成终端。

创建队列服务

你可能会熟悉 `System.Web.Hosting` 命名空间中的 QueueBackgroundWorkItem(Func<CancellationToken,Task>) 功能。若要为受此功能激发的服务建模，请先将 `IBackgroundTaskQueue` 接口添加到项目中：

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.QueueService
{
    public interface IBackgroundTaskQueue
    {
        ValueTask QueueBackgroundWorkItemAsync(
            Func<CancellationToken, ValueTask> workItem);

        ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
            CancellationToken cancellationToken);
    }
}
```

有两种方法，一种用于公开队列功能，另一种用于将之前排队的工作项移出队列。工作项是一个 `Func<CancellationToken, ValueTask>`。接下来，将默认实现添加到项目。

```
using System;
using System.Threading;
using System.Threading.Channels;
using System.Threading.Tasks;

namespace App.QueueService
{
    public class DefaultBackgroundTaskQueue : IBackgroundTaskQueue
    {
        private readonly Channel<Func<CancellationToken, ValueTask>> _queue;

        public DefaultBackgroundTaskQueue(int capacity)
        {
            BoundedChannelOptions options = new(capacity)
            {
                FullMode = BoundedChannelFullMode.Wait
            };
            _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>(options);
        }

        public async ValueTask QueueBackgroundWorkItemAsync(
            Func<CancellationToken, ValueTask> workItem)
        {
            if (workItem is null)
            {
                throw new ArgumentNullException(nameof(workItem));
            }

            await _queue.Writer.WriteAsync(workItem);
        }

        public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
            CancellationToken cancellationToken)
        {
            Func<CancellationToken, ValueTask>? workItem =
                await _queue.Reader.ReadAsync(cancellationToken);

            return workItem;
        }
    }
}
```

上述实现依赖 Channel<T> 作为队列。使用显式容量调用 BoundedChannelOptions(Int32)。应根据预期的应用程序负载和访问队列的并发线程数进行容量设置。BoundedChannelFullMode.Wait 将导致调用 ChannelWriter<T>.WriteAsync 返回一个任务，该任务仅在空间可用时才会完成。这会导致背压，以防过多的发

布服务器/调用开始累积。

重写辅助角色类

在以下 `QueueHostedService` 示例中：

- `ProcessTaskQueueAsync` 方法在 `ExecuteAsync` 中返回 Task。
- 在 `ProcessTaskQueueAsync` 中，取消排队并执行队列中的后台任务。
- 服务在 `StopAsync` 中停止之前，将等待工作项。

将现有 `Worker` 类替换为以下 C# 代码，并将该文件重命名为“QueueHostedService.cs”。

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.QueueService
{
    public sealed class QueuedHostedService : BackgroundService
    {
        private readonly IBackgroundTaskQueue _taskQueue;
        private readonly ILogger<QueuedHostedService> _logger;

        public QueuedHostedService(
            IBackgroundTaskQueue taskQueue,
            ILogger<QueuedHostedService> logger) =>
            (_taskQueue, _logger) = (taskQueue, logger);

        protected override Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(QueuedHostedService)} is running.{Environment.NewLine}" +
                $"{Environment.NewLine}Tap W to add a work item to the " +
                $"background queue.{Environment.NewLine}");

            return ProcessTaskQueueAsync(stoppingToken);
        }

        private async Task ProcessTaskQueueAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                try
                {
                    Func<CancellationToken, ValueTask>? workItem =
                        await _taskQueue.DequeueAsync(stoppingToken);

                    await workItem(stoppingToken);
                }
                catch (OperationCanceledException)
                {
                    // Prevent throwing if stoppingToken was signaled
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Error occurred executing task work item.");
                }
            }
        }

        public override async Task StopAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(QueuedHostedService)} is stopping.");

            await base.StopAsync(stoppingToken);
        }
    }
}
```

每当在输入设备上选择 `w` 键时，`MonitorLoop` 服务将处理托管服务的排队任务：

- `IBackgroundTaskQueue` 注入到 `MonitorLoop` 服务中。
- 调用 `IBackgroundTaskQueue.QueueBackgroundWorkItemAsync` 来将工作项排入队列。
- 工作项模拟长时间运行的后台任务：

- 将执行三次 5 秒的延迟 (Delay)。
- 如果任务已取消，`try-catch` 语句将捕获 OperationCanceledException。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace App.QueueService
{
    public class MonitorLoop
    {
        private readonly IBackgroundTaskQueue _taskQueue;
        private readonly ILogger<MonitorLoop> _logger;
        private readonly CancellationToken _cancellationToken;

        public MonitorLoop(
            IBackgroundTaskQueue taskQueue,
            ILogger<MonitorLoop> logger,
            IHostApplicationLifetime applicationLifetime)
        {
            _taskQueue = taskQueue;
            _logger = logger;
            _cancellationToken = applicationLifetime.ApplicationStopping;
        }

        public void StartMonitorLoop()
        {
            _logger.LogInformation($"{nameof(MonitorAsync)} loop is starting.");

            // Run a console user input loop in a background thread
            Task.Run(async () => await MonitorAsync());
        }

        private async ValueTask MonitorAsync()
        {
            while (!_cancellationToken.IsCancellationRequested)
            {
                var keyStroke = Console.ReadKey();
                if (keyStroke.Key == ConsoleKey.W)
                {
                    // Enqueue a background work item
                    await _taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItemAsync);
                }
            }
        }

        private async ValueTask BuildWorkItemAsync(CancellationToken token)
        {
            // Simulate three 5-second tasks to complete
            // for each enqueued work item

            int delayLoop = 0;
            var guid = Guid.NewGuid();

            _logger.LogInformation("Queued work item {Guid} is starting.", guid);

            while (!token.IsCancellationRequested && delayLoop < 3)
            {
                try
                {
                    await Task.Delay(TimeSpan.FromSeconds(5), token);
                }
                catch (OperationCanceledException)
                {
                    // Prevent throwing if the Delay is cancelled
```

```
                }

                ++ delayLoop;

                _logger.LogInformation("Queued work item {Guid} is running. {DelayLoop}/3", guid,
delayLoop);
            }

            string format = delayLoop switch
            {
                3 => "Queued Background Task {Guid} is complete.",
                _ => "Queued Background Task {Guid} was cancelled."
            };

            _logger.LogInformation(format, guid);
        }
    }
}
```

将现有 `Program` 内容替换为以下 C# 代码：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using App.QueueService;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices((context, services) =>
    {
        services.AddSingleton<MonitorLoop>();
        services.AddHostedService<QueuedHostedService>();
        services.AddSingleton<IBackgroundTaskQueue>(_ =>
        {
            if (!int.TryParse(context.Configuration["QueueCapacity"], out var queueCapacity))
            {
                queueCapacity = 100;
            }

            return new DefaultBackgroundTaskQueue(queueCapacity);
        });
    })
    .Build();

MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();

await host.RunAsync();
```

已在 `IHostBuilder.ConfigureServices` (*Program.cs*) 中注册这些服务。已使用 `AddHostedService` 扩展方法注册托管服务。`MonitorLoop` 在 Program.cs 顶级语句中启动：

```
MonitorLoop monitorLoop = host.Services.GetRequiredService<MonitorLoop>()!;
monitorLoop.StartMonitorLoop();
```

有关注册服务的详细信息，请参阅 .NET 中的依赖关系注入。

验证服务功能

若要从 Visual Studio 运行应用程序，请选择 F5 或选择“调试” > “开始调试”菜单选项。如果使用的是 .NET CLI，请从工作目录运行 `dotnet run` 命令：

```
dotnet run
```

有关 .NET CLI run 命令的详细信息，请参阅 dotnet run。

出现提示时，请至少输入 `w`（或 `W`）一次，以便对模拟的工作项进行排队。你将看到与下面类似的输出：

```
info: App.QueueService.MonitorLoop[0]
      MonitorAsync loop is starting.
info: App.QueueService.QueuedHostedService[0]
      QueuedHostedService is running.

      Tap W to add a work item to the background queue.

info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\queue-service
winfo: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is starting.
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 1/3
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 2/3
info: App.QueueService.MonitorLoop[0]
      Queued work item 8453f845-ea4a-4bcb-b26e-c76c0d89303e is running. 3/3
info: App.QueueService.MonitorLoop[0]
      Queued Background Task 8453f845-ea4a-4bcb-b26e-c76c0d89303e is complete.
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.QueueService.QueuedHostedService[0]
      QueuedHostedService is stopping.
```

如果从 Visual Studio 内部运行应用程序，请选择“调试” > “停止调试...”。或者，从控制台窗口中选择“Ctrl” + “C”，以发送取消信号。

另请参阅

- .NET 中的辅助角色服务
- 在 `BackgroundService` 内使用作用域服务
- 使用 `BackgroundService` 创建 Windows 服务
- 实现 `IHostedService` 接口

在 `BackgroundService` 内使用作用域服务

2021/11/16 •

当使用任意 AddHostedService 扩展方法注册 IHostedService 的实现时，该服务被注册为单一实例。在某些情况下，你可能想要依赖于作用域服务。有关详细信息，请参阅 .NET 服务生存期中的依赖关系注入。

在本教程中，你将了解如何执行以下操作：

- 解析单一实例 BackgroundService 中的作用域依赖关系。
- 将工作委托给作用域服务。
- 实现 BackgroundService.StopAsync(CancellationToken) 的 `override`。

> **TIP**
> 所有".NET 中的辅助角色"示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的辅助角色。

先决条件

- .NET 5.0 SDK 或更高版本
- .NET 集成开发环境 (IDE)
  - 随意使用 Visual Studio

创建新项目

若要使用 Visual Studio 创建新的辅助角色服务项目，请选择"文件" > "新建" > "项目..." 。从"创建新项目"对话框搜索"辅助角色服务"，并选择辅助角色服务模板。如果你想要使用 .NET CLI，请在工作目录中打开你最喜欢的终端。运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

> **TIP**
> 如果使用 Visual Studio Code，则可以从集成终端运行 .NET CLI 命令。有关详细信息，请参阅 Visual Studio Code：集成终端。

创建作用域服务

要在 `BackgroundService` 中使用有作用域的服务，请创建一个作用域。默认情况下，不会为托管服务创建作用域。作用域后台服务包含后台任务的逻辑。

```
using System.Threading;
using System.Threading.Tasks;

namespace App.ScopedService
{
    public interface IScopedProcessingService
    {
        Task DoWorkAsync(CancellationToken stoppingToken);
    }
}
```

前面的接口定义了一个 `DoWorkAsync` 方法。定义默认实现：

- 服务是异步的。`DoWorkAsync` 方法返回 `Task`。出于演示目的，在 `DoWorkAsync` 方法中等待 10 秒的延迟。
- ILogger 注入到服务中：

```
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Logging;

namespace App.ScopedService
{
    public class DefaultScopedProcessingService : IScopedProcessingService
    {
        private int _executionCount;
        private readonly ILogger<DefaultScopedProcessingService> _logger;

        public DefaultScopedProcessingService(
            ILogger<DefaultScopedProcessingService> logger) =>
            _logger = logger;

        public async Task DoWorkAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                ++ _executionCount;

                _logger.LogInformation(
                    "{ServiceName} working, execution count: {Count}",
                    nameof(DefaultScopedProcessingService),
                    _executionCount);

                await Task.Delay(10_000, stoppingToken);
            }
        }
    }
}
```

托管服务创建一个作用域来解决作用域后台服务以调用其 `DoWorkAsync` 方法。`DoWorkAsync` 返回 `ExecuteAsync` 等待的 `Task`：

重写辅助角色类

将现有 `Worker` 类替换为以下 C# 代码，并将该文件重命名为“ScopedBackgroundService.cs”：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.ScopedService
{
    public sealed class ScopedBackgroundService : BackgroundService
    {
        private readonly IServiceProvider _serviceProvider;
        private readonly ILogger<ScopedBackgroundService> _logger;

        public ScopedBackgroundService(
            IServiceProvider serviceProvider,
            ILogger<ScopedBackgroundService> logger) =>
            (_serviceProvider, _logger) = (serviceProvider, logger);

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(ScopedBackgroundService)} is running.");

            await DoWorkAsync(stoppingToken);
        }

        private async Task DoWorkAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(ScopedBackgroundService)} is working.");

            using (IServiceScope scope = _serviceProvider.CreateScope())
            {
                IScopedProcessingService scopedProcessingService =
                    scope.ServiceProvider.GetRequiredService<IScopedProcessingService>();

                await scopedProcessingService.DoWorkAsync(stoppingToken);
            }
        }

        public override async Task StopAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(ScopedBackgroundService)} is stopping.");

            await base.StopAsync(stoppingToken);
        }
    }
}
```

在前面的代码中，将创建一个显式作用域，并通过依赖关系注入服务提供程序解析 `IScopedProcessingService` 实现。解析后的服务实例是有作用域的，它的 `DoWorkAsync` 方法正在等待。

将模板 Program.cs 文件内容替换为以下 C# 代码：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using App.ScopedService;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHostedService<ScopedBackgroundService>();
        services.AddScoped<IScopedProcessingService, DefaultScopedProcessingService>();
    })
    .Build();

await host.RunAsync();
```

已在 `IHostBuilder.ConfigureServices` (*Program.cs*) 中注册这些服务。已使用 `AddHostedService` 扩展方法注册托管服务。

有关注册服务的详细信息，请参阅 .NET 中的依赖关系注入。

验证服务功能

若要从 Visual Studio 运行应用程序，请选择 F5 或选择“调试” > “开始调试”菜单选项。如果使用的是 .NET CLI，请从工作目录运行 `dotnet run` 命令：

```
dotnet run
```

有关 .NET CLI run 命令的详细信息，请参阅 dotnet run。

让应用程序运行一段时间，以生成多个执行计数增量。你将看到与下面类似的输出：

```
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is running.
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is working.
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 1
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\scoped-service
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 2
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 3
info: App.ScopedService.DefaultScopedProcessingService[0]
      DefaultScopedProcessingService working, execution count: 4
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.ScopedService.ScopedBackgroundService[0]
      ScopedBackgroundService is stopping.
```

如果从 Visual Studio 内部运行应用程序，请选择“调试” > “停止调试...”。或者，从控制台窗口中选择“Ctrl” + “C”，以发送取消信号。

另请参阅

- .NET 中的辅助角色服务

- 创建队列服务
- 使用 `BackgroundService` 创建 Windows 服务
- 实现 `IHostedService` 接口

使用 `BackgroundService` 创建 Windows 服务

2021/11/16 •

.NET Framework 开发人员可能熟悉 Windows 服务应用。在 .NET Core 和 .NET 5+ 之前，依赖 .NET Framework 的开发人员可能会创建 Windows 服务来执行后台任务或执行长时间运行的进程。此功能仍然可用，你可以创建作为 Windows 服务运行的辅助角色服务。

本教程介绍以下操作：

- 将 .NET 辅助角色应用作为单个文件可执行文件发布。
- 创建 Windows 服务。
- 将 `BackgroundService` 应用创建为 Windows 服务。
- 启动和停止 Windows 服务。
- 查看事件日志。
- 删除 Windows 服务。

> **TIP**
> 所有“.NET 中的辅助角色”示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的辅助角色。

先决条件

- .NET 5.0 SDK 或更高版本
- Windows OS
- .NET 集成开发环境 (IDE)
  - 随意使用 Visual Studio

创建新项目

若要使用 Visual Studio 创建新的辅助角色服务项目，请选择“文件” > “新建” > “项目...” 。从“创建新项目”对话框搜索“辅助角色服务”，并选择辅助角色服务模板。如果你想要使用 .NET CLI，请在工作目录中打开你最喜欢的终端。运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

> **TIP**
> 如果使用 Visual Studio Code，则可以从集成终端运行 .NET CLI 命令。有关详细信息，请参阅 Visual Studio Code：集成终端。

安装 NuGet 包

为了与 .NET IHostedService 实现中的本机 Windows 服务互操作，你需要安装 `Microsoft.Extensions.Hosting.WindowsServices` NuGet 包。

若要从 Visual Studio 安装此包，请使用“管理 NuGet 包...”对话框。搜索“Microsoft.Extensions.Hosting.WindowsServices”，然后安装它。如果要使用 .NET CLI，请运行 `dotnet add package` 命令：

```
dotnet add package Microsoft.Extensions.Hosting.WindowsServices
```

作为本教程的示例源代码的一部分，你还需要安装 `Microsoft.Extensions.Http` NuGet 包。

```
dotnet add package Microsoft.Extensions.Http
```

有关 .NET CLI add package 命令的详细信息，请参阅 `dotnet add package`。

成功添加包后，你的项目文件现在应包含以下包引用：

```
<ItemGroup>
  <PackageReference Include="Microsoft.Extensions.Hosting" Version="5.0.0" />
  <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices" Version="5.0.1" />
  <PackageReference Include="Microsoft.Extensions.Http" Version="5.0.0" />
</ItemGroup>
```

更新项目文件

此辅助角色项目使用 C# 的可为 null 的引用类型。若要为整个项目启用这些类型，请对项目文件进行相应的更新：

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>App.WindowsService</RootNamespace>
    <Nullable>enable</Nullable>
    <OutputType>exe</OutputType>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="5.0.0" />
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices" Version="5.0.1" />
    <PackageReference Include="Microsoft.Extensions.Http" Version="5.0.0" />
  </ItemGroup>
</Project>
```

上述项目文件更改添加了 `<Nullable>enable<Nullable>` 节点。有关详细信息，请参阅设置可为 null 的上下文。

创建服务

将新类添加到名为 JokeService.cs 的项目，并将其内容替换为以下 C# 代码：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Text.Json;
using System.Threading.Tasks;

namespace App.WindowsService
{
    public class JokeService
    {
        private readonly HttpClient _httpClient;
        private readonly JsonSerializerOptions _options = new()
        {
            PropertyNameCaseInsensitive = true
        };

        private const string JokeApiUrl =
            "https://karljoke.herokuapp.com/jokes/programming/random";

        public JokeService(HttpClient httpClient) => _httpClient = httpClient;

        public async Task<string> GetJokeAsync()
        {
            try
            {
                // The API returns an array with a single entry.
                Joke[]? jokes = await _httpClient.GetFromJsonAsync<Joke[]>(
                    JokeApiUrl, _options);

                Joke? joke = jokes?[0];

                return joke is not null
                    ? $"{joke.Setup}{Environment.NewLine}{joke.Punchline}"
                    : "No joke here...";
            }
            catch (Exception ex)
            {
                return $"That's not funny! {ex}";
            }
        }
    }

    public record Joke(int Id, string Type, string Setup, string Punchline);
}
```

前面的玩笑服务源代码公开了单个功能，即 `GetJokeAsync` 方法。这是一个 Task<TResult> 返回方法，其中 `T` 是一个 `string`，它表示随机的编程玩笑。将 HttpClient 注入构造函数并分配给类范围 `_httpClient` 变量。

TIP

玩笑 API 来自 GitHub 上的开源项目。它用于演示目的，我们不保证将来可以使用它。若要快速测试 API，请在浏览器中打开以下 URL：

```
https://karljoke.herokuapp.com/jokes/programming/random.
```

重写辅助角色类

将模板中的现有 `Worker` 替换为以下 C# 代码，并将该文件重命名为 WindowsBackgroundService.cs：

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.WindowsService
{
    public sealed class WindowsBackgroundService : BackgroundService
    {
        private readonly JokeService _jokeService;
        private readonly ILogger<WindowsBackgroundService> _logger;

        public WindowsBackgroundService(
            JokeService jokeService,
            ILogger<WindowsBackgroundService> logger) =>
            (_jokeService, _logger) = (jokeService, logger);

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                string joke = await _jokeService.GetJokeAsync();
                _logger.LogWarning(joke);

                await Task.Delay(TimeSpan.FromMinutes(1), stoppingToken);
            }
        }
    }
}
```

在前面的代码中，将 `JokeService` 与 `ILogger` 一起注入。这两者都作为 `private readonly` 字段提供给类。在 `ExecuteAsync` 方法中，玩笑服务请求一个玩笑，并将其写入记录器。在这种情况下，记录器由 Windows 事件日志实现 - Microsoft.Extensions.Logging.EventLog.EventLogLogger。日志已写入，并可在事件查看器中查看。

> **NOTE**
>
> 默认情况下，事件日志严重性为 Warning。可对此进行配置，但出于演示目的，`WindowsBackgroundService` 使用 LogWarning 扩展方法进行记录。若要专门针对 `EventLog` 级别，请在 appsettings.{Environment}.json 中添加一个条目或提供一个 EventLogSettings.Filter 值。
>
> ```
> "Logging": {
>   "EventLog": {
>     "LogLevel": {
>       "Default": "Information"
>     }
>   }
> }
> ```
>
> 有关配置日志级别的详细信息，请参阅 .NET 中的日志记录提供程序：配置 Windows 事件日志。

重写 Program 类

将模板 Program.cs 文件内容替换为以下 C# 代码：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using App.WindowsService;

using IHost host = Host.CreateDefaultBuilder(args)
    .UseWindowsService(options =>
    {
        options.ServiceName = ".NET Joke Service";
    })
    .ConfigureServices(services =>
    {
        services.AddHostedService<WindowsBackgroundService>();
        services.AddHttpClient<JokeService>();
    })
    .Build();

await host.RunAsync();
```

UseWindowsService(IHostBuilder) 扩展方法将应用配置为作为 Windows 服务工作。服务名称设置为 `".NET Joke Service"`。已注册托管服务，并将 `HttpClient` 注册到 `JokeService` 进行依赖关系注入。

有关注册服务的详细信息，请参阅 .NET 中的依赖关系注入。

发布应用

若要将 .NET 辅助角色服务应用创建为 Windows 服务，建议将应用作为单个可执行文件发布。拥有一个独立式可执行文件不太容易出错，因为文件系统周围没有任何依赖文件。但是你也可以选择其他发布形式，这是完全可以接受的，只要创建的 *.exe 文件可以作为 Windows 服务控制管理器的目标。

> **IMPORTANT**
> 另一种发布方法是生成 *.dll（而不是 *.exe），当使用 Windows 服务控制管理器安装已发布的应用时，委托给 .NET CLI 并传递 DLL 。有关详细信息，请参阅 .NET CLI：dotnet 命令。
>
> ```
> sc.exe create ".NET Joke Service" binpath="C:\Path\To\dotnet.exe C:\Path\To\App.WindowsService.dll"
> ```

```
<Project Sdk="Microsoft.NET.Sdk.Worker">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <RootNamespace>App.WindowsService</RootNamespace>
    <Nullable>enable</Nullable>
    <OutputType>exe</OutputType>
    <PublishSingleFile>true</PublishSingleFile>
    <RuntimeIdentifier>win-x64</RuntimeIdentifier>
    <PlatformTarget>x64</PlatformTarget>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting" Version="5.0.0" />
    <PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices" Version="5.0.1" />
    <PackageReference Include="Microsoft.Extensions.Http" Version="5.0.0" />
  </ItemGroup>
</Project>
```

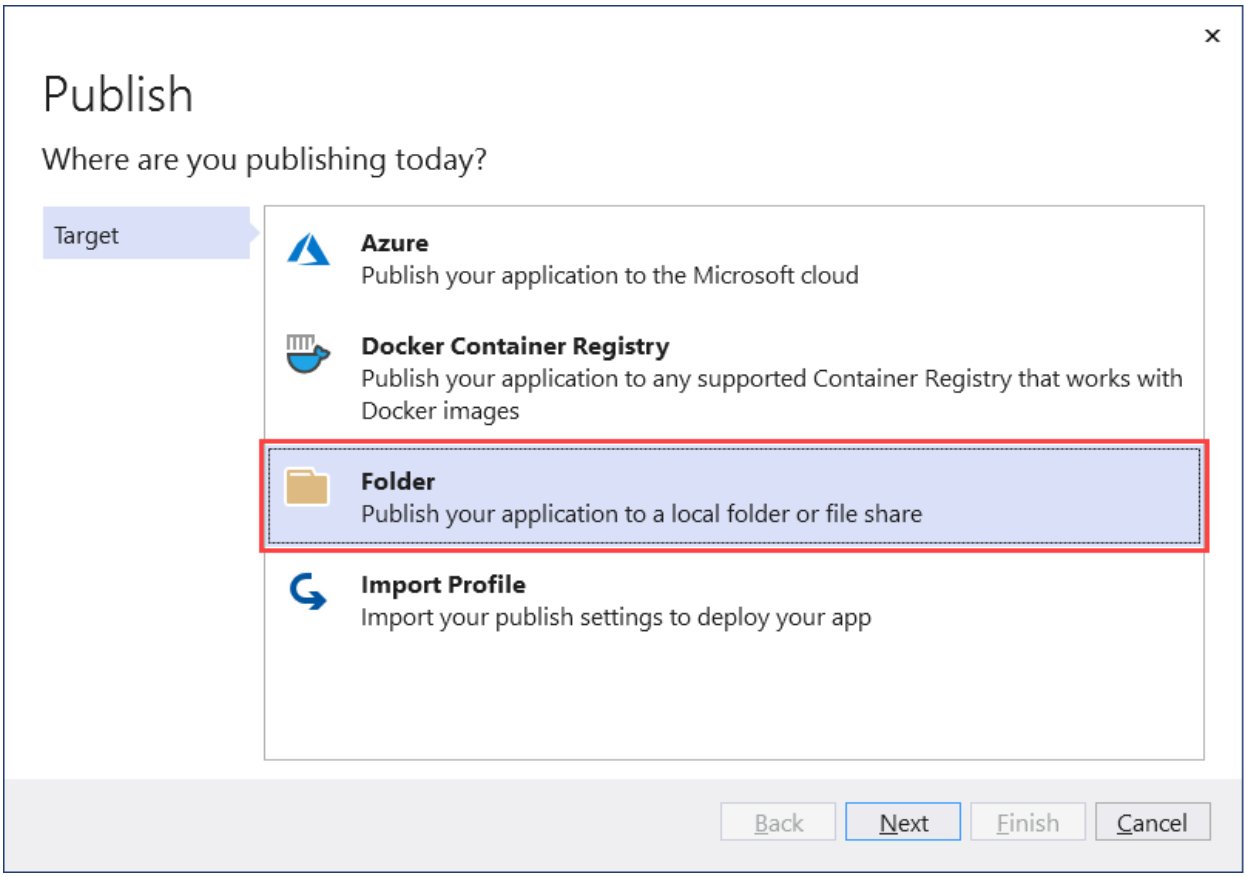
项目文件前面突出显示的行定义了以下行为：

- `<OutputType>exe</OutputType>`：创建控制台应用程序。

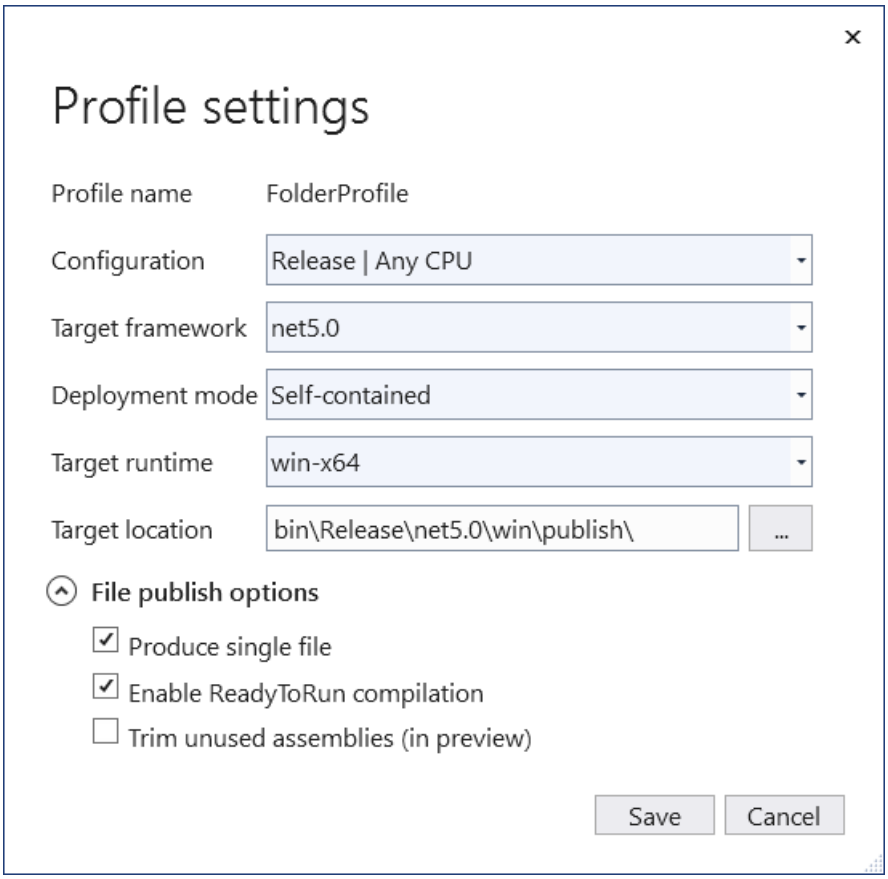
- `<PublishSingleFile>true</PublishSingleFile>`：启用单文件发布。
- `<RuntimeIdentifier>win-x64</RuntimeIdentifier>`：指定 `win-x64` 的 RID。
- `<PlatformTarget>x64</PlatformTarget>`：指定 64 位的目标平台 CPU。

若要从 Visual Studio 发布应用，可以创建一个持久保留的发布配置文件。发布配置文件基于 XML，其文件扩展名为 .pubxml。Visual Studio 使用此配置文件来隐式发布应用，而如果你使用 .NET CLI，则必须显式指定发布配置文件才能使用该配置文件。

右键单击“解决方案资源管理器”中的项目，选择“发布...”。然后，选择“添加发布配置文件”创建配置文件。从“发布”对话框中，选择“文件夹”作为“目标”。



保留默认“位置”，然后选择“完成”。创建配置文件后，选择“显示所有设置”，然后验证“配置文件设置”。



确保指定了以下设置：

- 部署模式：自包含
- 生成单个文件：已选中
- 启用 ReadyToRun 编译：已选中
- 剪裁未使用的程序集(预览版)：未选中

最后，选择“发布”。将编译应用，并将生成的 .exe 文件发布到 /publish 输出目录。

或者，可以使用 .NET CLI 发布应用：

```
dotnet publish --output "C:\custom\publish\directory"
```

有关详细信息，请参阅 `dotnet publish`。

创建 Windows 服务

若要创建 Windows 服务，请使用本机 Windows 服务控制管理器 (sc.exe) create 命令。以管理员身份运行 PowerShell。

```
sc.exe create ".NET Joke Service" binpath="C:\Path\To\App.WindowsService.exe"
```

TIP

如果需要更改主机配置的内容根源，可以在指定 `binpath` 时将其作为命令行参数传递：

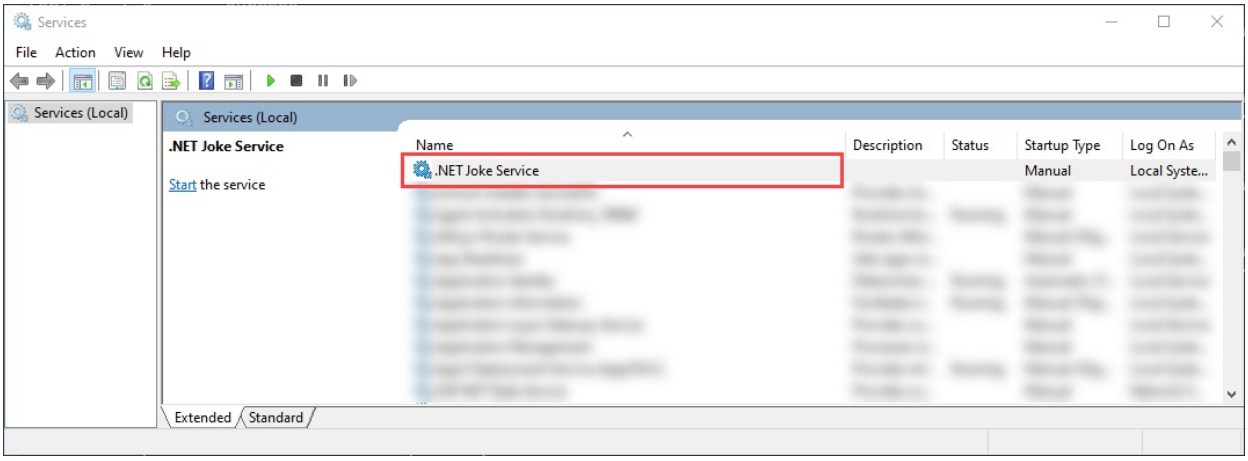
```
sc.exe create "Svc Name" binpath="C:\Path\To\App.exe --contentRoot C:\Other\Path"
```

你将看到以下输出消息：

```
[SC] CreateService SUCCESS
```

有关详细信息，请参阅 sc.exe create。

若要查看创建为 Windows 服务的应用，请打开“服务”。选择 Windows 键（或 Ctrl + Esc），然后从“服务”进行搜索。从“服务”应用，你应该能够按名称找到你的服务。



验证服务功能

若要验证服务是否按预期运行，需要执行以下操作：

- 启动服务
- 查看日志
- 停止服务

启动 Windows 服务

若要启动 Windows 服务，请使用 `sc.exe start` 命令：

```
sc.exe start ".NET Joke Service"
```

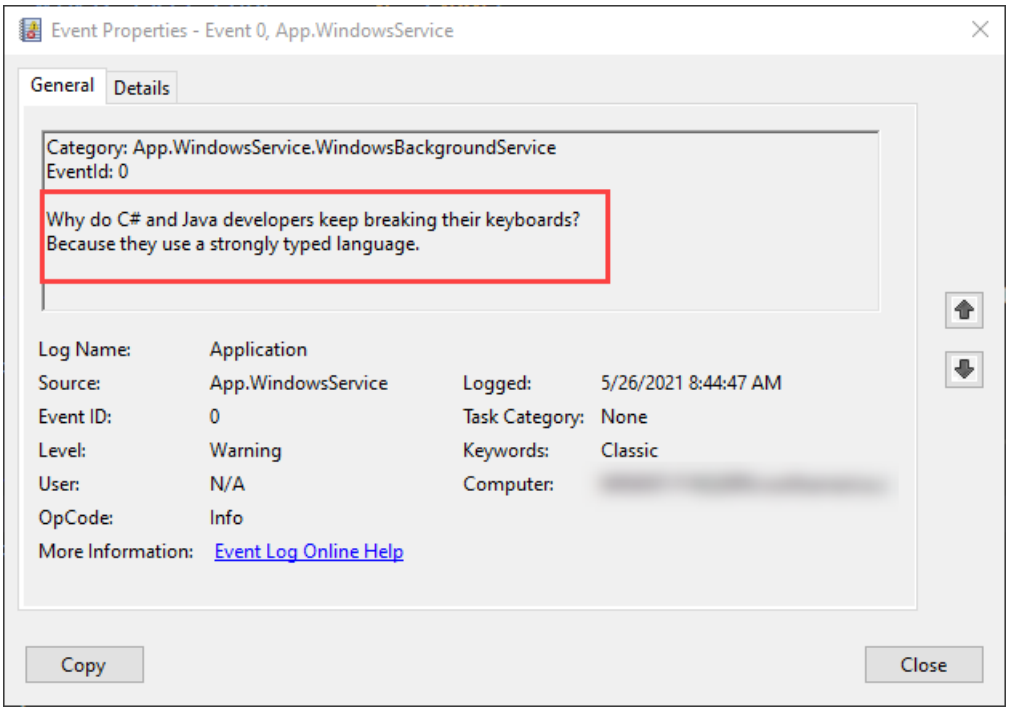
将显示类似于下面的输出：

```
SERVICE_NAME: .NET Joke Service
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 2  START_PENDING
                            (NOT_STOPPABLE, NOT_PAUSABLE, IGNORES_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x7d0
    PID                : 37636
    FLAGS
```

服务“状态”将从 `START_PENDING` 转换到“正在运行”。

查看日志

若要查看日志，请打开“事件查看器”。选择 Windows 键（或 Ctrl + Esc），然后搜索 `"Event Viewer"`。选择“事件查看器(本地)” > “Windows 日志” > “应用程序”节点。你应该会看到“源”与应用命名空间匹配的“警告”级别条目。双击该条目，或右键单击并选择“事件属性”以查看详细信息。



查看"事件日志"中的日志后，应停止该服务。它设计为每分钟记录一次随机玩笑。这是有意的行为，但不适用于生产服务。

停止 Windows 服务

若要停止 Windows 服务，请使用 `sc.exe stop` 命令：

```
sc.exe stop ".NET Joke Service"
```

将显示类似于下面的输出：

```
SERVICE_NAME: .NET Joke Service
    TYPE               : 10  WIN32_OWN_PROCESS
    STATE              : 3  STOP_PENDING
                            (STOPPABLE, NOT_PAUSABLE, ACCEPTS_SHUTDOWN)
    WIN32_EXIT_CODE    : 0  (0x0)
    SERVICE_EXIT_CODE  : 0  (0x0)
    CHECKPOINT         : 0x0
    WAIT_HINT          : 0x0
```

服务"状态"将从 `STOP_PENDING` 转换到"已停止"。

删除 Windows 服务

若要删除 Windows 服务，请使用本机 Windows 服务控制管理器 (sc.exe) delete 命令。以管理员身份运行 PowerShell。

> **IMPORTANT**
> 如果服务未处于"已停止"状态，将不会立即删除它。在发出删除命令之前，请确保服务已停止。

```
sc.exe delete ".NET Joke Service"
```

你将看到以下输出消息：

```
[SC] DeleteService SUCCESS
```

有关详细信息，请参阅 sc.exe delete。

另请参阅

- .NET 中的辅助角色服务
- 创建队列服务
- 在 `BackgroundService` 内使用作用域服务
- 实现 `IHostedService` 接口

实现 `IHostedService` 接口

2021/11/16 •

当需要超出提供的 BackgroundService 的有限控制，可实现你自己的 IHostedService。IHostedService 接口是 .NET 中所有长期运行的服务的基础。已使用 AddHostedService<THostedService>(IServiceCollection) 扩展方法注册自定义实现。

在本教程中，你将了解如何执行以下操作：

- 实现 IHostedService 和 IAsyncDisposable 接口。
- 创建基于计时器的服务。
- 使用依赖项注入和日志记录注册自定义注入。

> **TIP**
> 所有“.NET 中的辅助角色”示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的辅助角色。

先决条件

- .NET 5.0 SDK 或更高版本
- .NET 集成开发环境 (IDE)
  - 随意使用 Visual Studio

创建新项目

若要使用 Visual Studio 创建新的辅助角色服务项目，请选择“文件” > “新建” > “项目...” 。从“创建新项目”对话框搜索“辅助角色服务”，并选择辅助角色服务模板。如果你想要使用 .NET CLI，请在工作目录中打开你最喜欢的终端。运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

> **TIP**
> 如果使用 Visual Studio Code，则可以从集成终端运行 .NET CLI 命令。有关详细信息，请参阅 Visual Studio Code：集成终端。

创建计时器服务

基于计时器的后台服务使用 System.Threading.Timer 类。计时器触发 `DoWork` 方法。在 IHostLifetime.StopAsync(CancellationToken) 上禁用计时器，并在 IAsyncDisposable.DisposeAsync() 上处置服务容器时处置计时器：

将模板中 `Worker` 的内容替换为以下 C# 代码，并将文件重命名为“TimerService.cs”：

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;
using System;
using System.Threading;
using System.Threading.Tasks;

namespace App.TimerHostedService
{
    public sealed class TimerService : IHostedService, IAsyncDisposable
    {
        private readonly Task _completedTask = Task.CompletedTask;
        private readonly ILogger<TimerService> _logger;
        private int _executionCount = 0;
        private Timer? _timer;

        public TimerService(ILogger<TimerService> logger) => _logger = logger;

        public Task StartAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation($"{nameof(TimerHostedService)} is running.");
            _timer = new Timer(DoWork, null, TimeSpan.Zero, TimeSpan.FromSeconds(5));

            return _completedTask;
        }

        private void DoWork(object? state)
        {
            int count = Interlocked.Increment(ref _executionCount);

            _logger.LogInformation(
                $"{nameof(TimerHostedService)} is working, execution count: {{Count:#,0}}",
                count);
        }

        public Task StopAsync(CancellationToken stoppingToken)
        {
            _logger.LogInformation(
                $"{nameof(TimerHostedService)} is stopping.");

            _timer?.Change(Timeout.Infinite, 0);

            return _completedTask;
        }

        public async ValueTask DisposeAsync()
        {
            if (_timer is IAsyncDisposable timer)
            {
                await timer.DisposeAsync();
            }

            _timer = null;
        }
    }
}
```

IMPORTANT

`Worker` 是 BackgroundService 的子类。现在，`TimerService` 可同时实现 IHostedService 和 IAsyncDisposable 接口。

`TimerService` 为 `sealed`，并级联来自其 `_timer` 实例的 `DisposeAsync` 调用。有关"级联释放模式"详细信息，请参阅实现 `DisposeAsync` 方法。

调用 StartAsync 时，将实例化计时器，从而启动计时器。

> **TIP**
> Timer 不等待先前的 `DoWork` 执行完成，因此所介绍的方法可能并不适用于所有场景。使用 Interlocked.Increment 以原子操作的形式执行计数器递增，这可确保多个线程不会并行更新 `_executionCount`。

将现有 `Program` 内容替换为以下 C# 代码：

```
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using App.TimerHostedService;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddHostedService<TimerService>();
    })
    .Build();

await host.RunAsync();
```

已使用 `AddHostedService` 扩展方法在 `IHostBuilder.ConfigureServices` (Program.cs) 中注册该服务。这是注册 BackgroundService 子类时使用的相同扩展方法，因为它们均实现 IHostedService 接口。

有关注册服务的详细信息，请参阅 .NET 中的依赖关系注入。

验证服务功能

若要从 Visual Studio 运行应用程序，请选择 F5 或选择“调试” > “开始调试”菜单选项。如果使用的是 .NET CLI，请从工作目录运行 `dotnet run` 命令：

```
dotnet run
```

有关 .NET CLI run 命令的详细信息，请参阅 dotnet run。

让应用程序运行一段时间，以生成多个执行计数增量。你将看到与下面类似的输出：

```
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is running.
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: .\timer-service
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 1
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 2
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 3
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is working, execution count: 4
info: Microsoft.Hosting.Lifetime[0]
      Application is shutting down...
info: App.TimerHostedService.TimerService[0]
      TimerHostedService is stopping.
```

如果从 Visual Studio 内部运行应用程序，请选择“调试” > “停止调试...”。或者，从控制台窗口中选择“Ctrl” + “C”，

以发送取消信号。

另请参阅

需要考虑几个相关教程：

- .NET 中的辅助角色服务
- 创建队列服务
- 在 `BackgroundService` 内使用作用域服务
- 使用 `BackgroundService` 创建 Windows 服务

将辅助角色服务部署到 Azure

2021/11/16 •

本文介绍如何将 .NET 辅助角色服务部署到 Azure。当辅助角色作为 Azure 容器注册表 (ACR) 中的 Azure 容器实例 (ACI) 运行时，它可以充当云中的微服务。长时间运行的服务有许多用例，这正是使用辅助角色服务的原因。

在本教程中，你将了解如何执行以下操作：

- 创建辅助角色服务。
- 创建容器注册表资源。
- 将映像推送到容器注册表。
- 作为容器实例部署。
- 验证辅助角色服务功能。

> **TIP**
> 所有“.NET 中的辅助角色”示例源代码都可以在示例浏览器中下载。有关详细信息，请参阅浏览代码示例：.NET 中的辅助角色。

先决条件

- .NET 5.0 SDK 或更高版本。
- Docker Desktop（Windows 或 Mac）。
- 具有活动订阅的 Azure 帐户。免费创建帐户。
- 根据你选择的开发人员环境：
  - Visual Studio、Visual Studio Code 或 Visual Studio for Mac。
  - .NET CLI
  - Azure CLI。

创建新项目

若要使用 Visual Studio 创建新的辅助角色服务项目，请选择“文件” > “新建” > “项目...” 。从“创建新项目”对话框搜索“辅助角色服务”，并选择辅助角色服务模板。输入所需的项目名称，选择相应的位置，然后选择“下一步”。在“其他信息”页上，对于“目标框架”，请选择 `.NET 5.0`，并选中“启用 Docker”选项以启用 Docker 支持。选择所需的 Docker OS。

若要使用 Visual Studio Code 创建新的辅助角色服务项目，可以从集成终端运行 .NET CLI 命令。有关详细信息，请参阅 Visual Studio Code：集成终端。

打开集成终端并运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

若要使用 .NET CLI 创建新的辅助角色服务项目，请在工作目录中打开你最喜欢的终端。运行 `dotnet new` 命令，将 `<Project.Name>` 替换为所需的项目名称。

```
dotnet new worker --name <Project.Name>
```

有关 .NET CLI 新建辅助角色服务项目命令的详细信息，请参阅 dotnet new 辅助角色。

构建应用程序以确保它还原依赖包，并且编译时不会出错。

若要从 Visual Studio 生成应用程序，请选择 F6 或选择“生成” > “生成解决方案”菜单选项。

若要从 Visual Studio Code 生成应用程序，请打开集成终端窗口，然后从工作目录运行 `dotnet build` 命令。

```
dotnet build
```

有关 .NET CLI build 命令的详细信息，请参阅 `dotnet build`。

若要从 .NET CLI 生成应用程序，请从工作目录运行 `dotnet build` 命令。

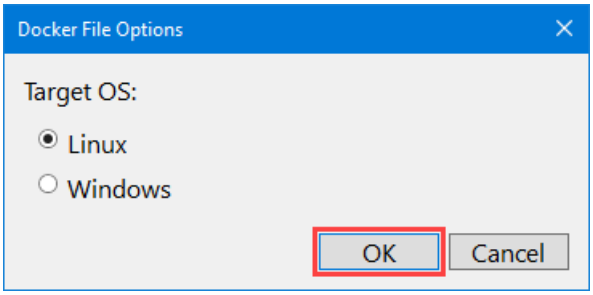
```
dotnet build <path/to/project.csproj>
```

指定 `<path/to/project.csproj>` 值，该值是要生成的项目文件的路径。有关 .NET CLI build 命令的详细信息，请参阅 `dotnet build`。

添加 Docker 支持

如果在创建新的 Worker 项目时正确选择了“启用 Docker”复选框，请跳至生成 Docker 映像步骤。

如果未选择此选项，不用担心，你现在仍然可以添加它。在 Visual Studio 中，右键单击“解决方案资源管理器”中的项目节点，然后选择“添加” > “Docker 支持”。系统将提示你选择“目标 OS”，选择“确定”使用默认操作系统。



在 Visual Studio Code 中，需要安装 Docker 扩展和 Azure 帐户扩展。打开命令面板，选择“Docker: 将 Docker 文件添加到工作区”选项。当系统提示“选择应用程序平台”时，请选择“.NET: Core 控制台”。当系统提示“选择项目”时，请选择你创建的辅助角色服务项目。当系统提示“选择操作系统”时，选择列出的第一个操作系统。当系统提示是否“包括可选 Docker Compose 文件”时，请选择“否”。

Docker 支持需要 Dockerfile。此文件是一整套指令，用于将 .NET 辅助角色服务生成为 Docker 映像。Dockerfile 是一个没有文件扩展名的文件。下面是一个 Dockerfile 的示例，应存在于项目文件的根目录中。

使用 CLI，Dockerfile 不是为你创建的。需要将其内容复制到一个名为 Dockerfile 的新文件中，并且该文件应再次位于项目的根目录中。

```
FROM mcr.microsoft.com/dotnet/runtime:5.0 AS base
WORKDIR /app

# Creates a non-root user with an explicit UID and adds permission to access the /app folder
# For more info, please refer to https://aka.ms/vscode-docker-dotnet-configure-containers
RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app
USER appuser

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["App.CloudService.csproj", "./"]
RUN dotnet restore "App.CloudService.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "App.CloudService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "App.CloudService.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "App.CloudService.dll"]
```

> **NOTE**
> 需要更新 Dockerfile 中引用 *App.CloudService(将其替换为项目名称)的各行。

有关官方 .NET 映像的详细信息，请参阅 Docker Hub：.NET 运行时和 Docker Hub：.NET SDK。

生成 Docker 映像

若要生成 Docker 映像，Docker 引擎必须运行。

> **IMPORTANT**
> 使用 Docker Desktop 和 Visual Studio 时，若要避免与卷共享相关的错误，请确保已启用该功能。
>
> 1. 在 Docker Desktop 中的“设置”屏幕上，选择“共享驱动器”。
> 2. 选择包含项目文件的驱动器。
>
> 有关详细信息，请参阅使用 Docker 排查 Visual Studio 开发方面的问题。

右键单击解决方案资源管理器中的“Dockerfile”，然后选择“生成 Docker 映像”。此时将显示“输出”窗口，用于报告 `docker build` 命令进度。

右键单击资源管理器中的“Dockerfile”，然后选择“生成映像”。当系统提示“将映像标记为”时，请输入 `appcloudservice:latest`。此时将显示“Docker 任务”输出终端，用于报告 Docker 生成命令进度。

> **NOTE**
>
> 如果系统未提示标记映像，则可能是 Visual Studio Code 依赖于现有的 tasks.json。如果使用的不是所需标记，可以对其进行更改，方法是更新 `tasks` 数组中 `docker-build` 配置项的 `dockerBuild/tag` 值。请考虑以下示例配置部分：
>
> ```
> {
>   "type": "docker-build",
>   "label": "docker-build: release",
>   "dependsOn": [
>     "build"
>   ],
>   "dockerBuild": {
>     "tag": "appcloudservice:latest",
>     "dockerfile": "${workspaceFolder}/cloud-service/Dockerfile",
>     "context": "${workspaceFolder}",
>     "pull": true
>   },
>   "netCore": {
>     "appProject": "${workspaceFolder}/cloud-service/App.CloudService.csproj"
>   }
> }
> ```

在 Dockerfile 的根目录中打开终端窗口，并运行以下 docker 命令：

```
docker build -t appcloudservice:latest -f Dockerfile .
```

`docker build` 命令运行时，它会将 Dockerfile 中的每一行处理为一个指令步骤。此命令生成映像，并创建一个指向该映像的本地存储库“appcloudservice”。

> **TIP**
>
> 生成的 Dockerfile 因开发环境不同而不同。例如，如果从 Visual Studio 中添加 Docker 支持，则在尝试从 Visual Studio Code 生成 Docker 映像时可能会遇到问题 — 因为 Dockerfile 步骤有所不同。最好选择一个开发环境并在本教程中始终使用该环境。

创建容器注册表

利用 Azure 容器注册表 (ACR) 资源，你可以在专用注册表中生成、存储和管理容器映像与项目。若要创建容器注册表，需要在 Azure 门户中创建一个新资源。

1. 选择“订阅”和相应的“资源组”（或创建一个新资源组）。
2. 输入注册表名称。
3. 选择“位置”。
4. 选择适当的 SKU，例如“基本”。
5. 选择“查看 + 创建”。
6. 系统显示“验证已通过”后，选择“创建”。

> **IMPORTANT**
>
> 若要在创建容器实例时使用此容器注册表，必须启用“管理员用户”。选择“访问密钥”，然后启用“管理员用户”。

利用 Azure 容器注册表 (ACR) 资源，你可以在专用注册表中生成、存储和管理容器映像与项目。在 Dockerfile 的根目录中打开终端窗口，并运行以下 Azure CLI 命令：

IMPORTANT

要从 Azure CLI 与 Azure 资源交互，必须对终端会话进行身份验证。若要进行身份验证，请使用 `az login` 命令：

```
az login
```

登录后，如果具有多个订阅且未设置默认订阅，请使用 `az account set` 命令指定你的订阅。

```
az account set --subscription <subscription name or id>
```

登录到 Azure CLI 后，会话可以相应地与资源进行交互。

如果还没有想要将工作器服务与之关联的资源组，请使用 `az group create` 命令创建一个：

```
az group create -n <resource group> -l <location>
```

提供 `<resource group>` 名称和 `<location>`。若要创建容器注册表，需要调用 `az acr create` 命令。

```
az acr create -n <registry name> -g <resource group> --sku <sku> --admin-enabled true
```

将占位符替换为自己的适当值：

- `<registry name>`：注册表的名称。
- `<resource group>`：上面使用的资源组名称。
- `<sku>`：接受的值、基本、经典、高级或标准。

上述命令：

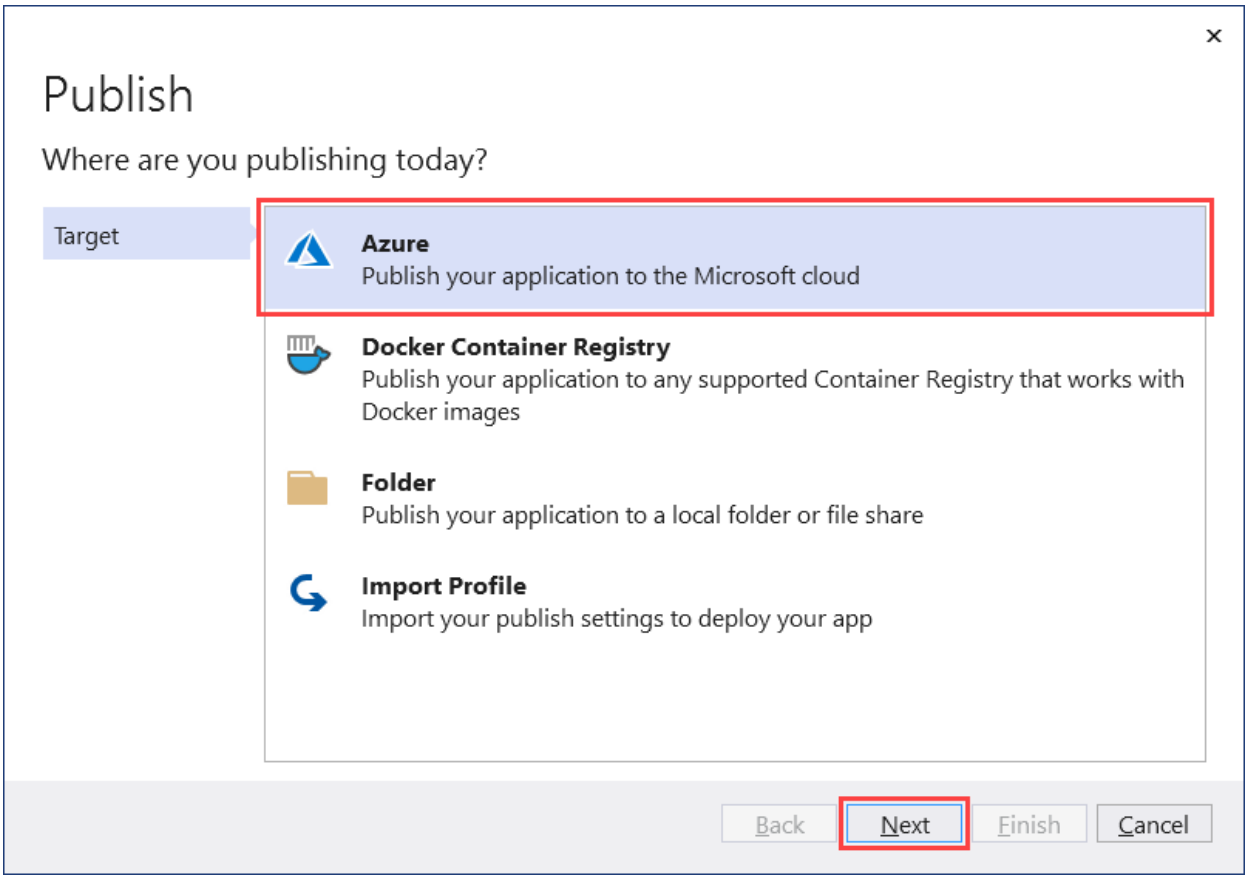
- 在指定的资源组中创建一个给定了注册表名称的 Azure 容器注册表。
- 已启用管理员用户，这是 Azure 容器实例所必需的。

有关详细信息，请参阅快速入门：创建 Azure 容器注册表。

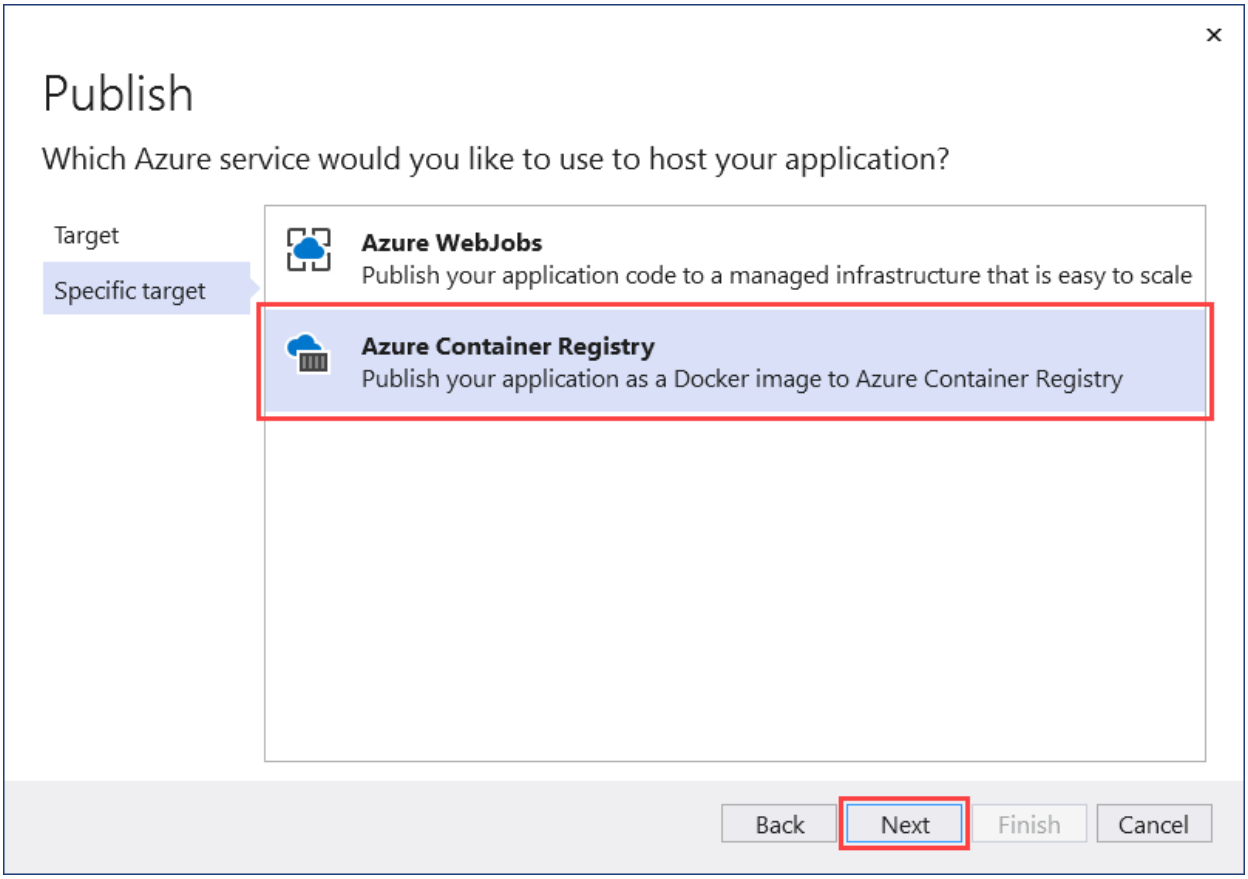
将映像推送到容器注册表

生成 .NET Docker 映像并创建容器注册表资源后，现在就可以将映像推送到容器注册表。

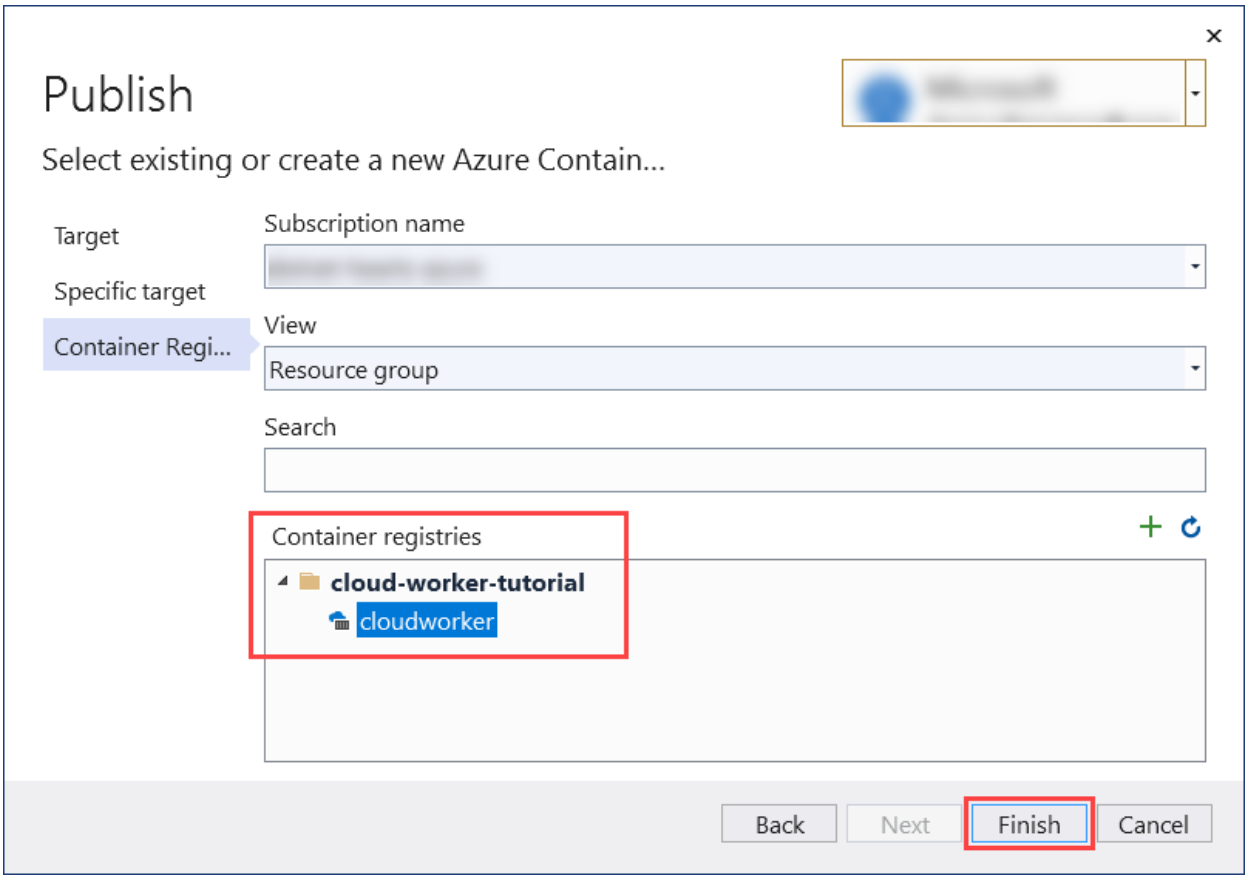
在解决方案资源管理器中，右键单击该项目并选择“发布”。此时会显示“发布”对话框。对于“目标”，选择“Azure”，然后选择“下一步”。



对于“特定目标”，选择“Azure 容器注册表”，然后选择“下一步”。

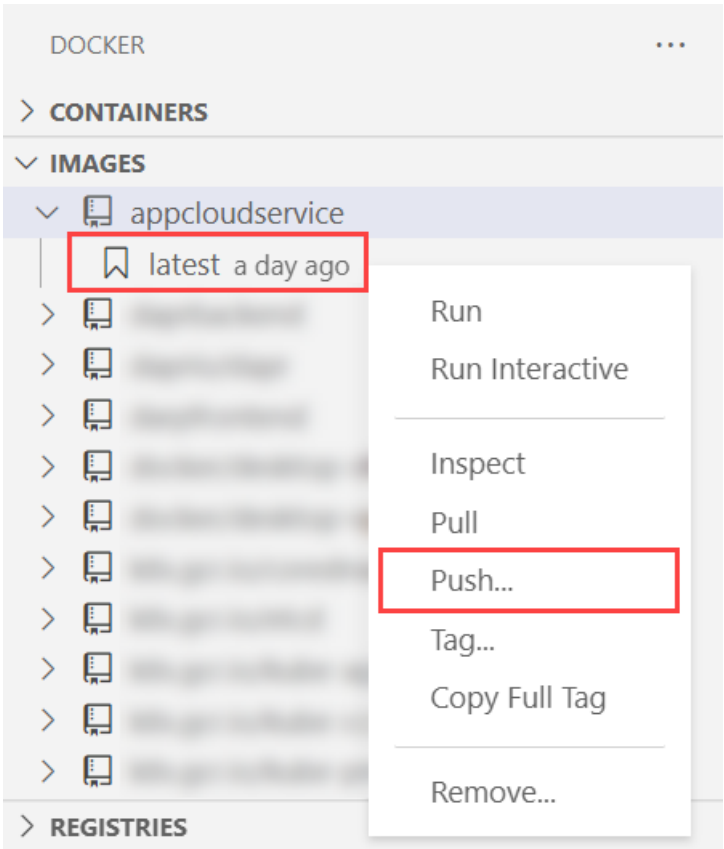


接下来，对于“容器注册表”，选择曾用于创建 ACR 资源的“订阅名称”。在“容器注册表”选择区域，选择创建的容器注册表，然后选择“完成”。



该操作会创建一个发布配置文件，可用于将映像发布到容器注册表。选择“发布”按钮将映像推送到容器注册表，“输出”窗口将报告发布进度，操作成功完成后，将显示“已成功发布”的消息。

从 Visual Studio Code 中的“活动栏”中选择“Docker”。展开“图像”树状视图，然后展开 `appcloudservice` 映像节点并右键单击 `latest` 标记。



集成终端窗口将向容器注册表报告 `docker push` 命令的进度。

若要将映像推送到容器注册表，需要先登录注册表：

```
az acr login -n <registry name>
```

`az acr login` 命令将通过 Docker CLI 登录到容器注册表。若要将映像推送到容器注册表，请使用 az acr build 命令，容器注册表名称为 `<registry name>`：

```
az acr build -r <registry name> -t appcloudservice .
```

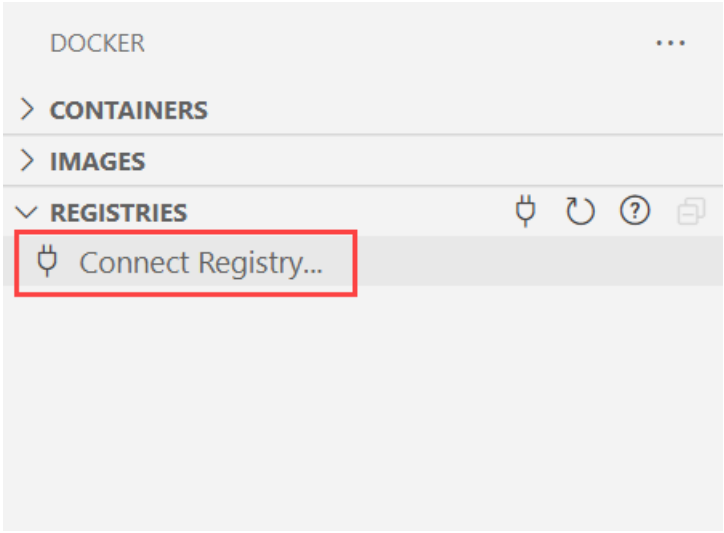
上述命令：

- 将源打包到 tar 文件。
- 将其上传到容器注册表。
- 容器注册表将解压缩 tar 文件。
- 在容器注册表资源中对 Dockerfile 运行 `docker build` 命令。
- 将映像添加到容器注册表。

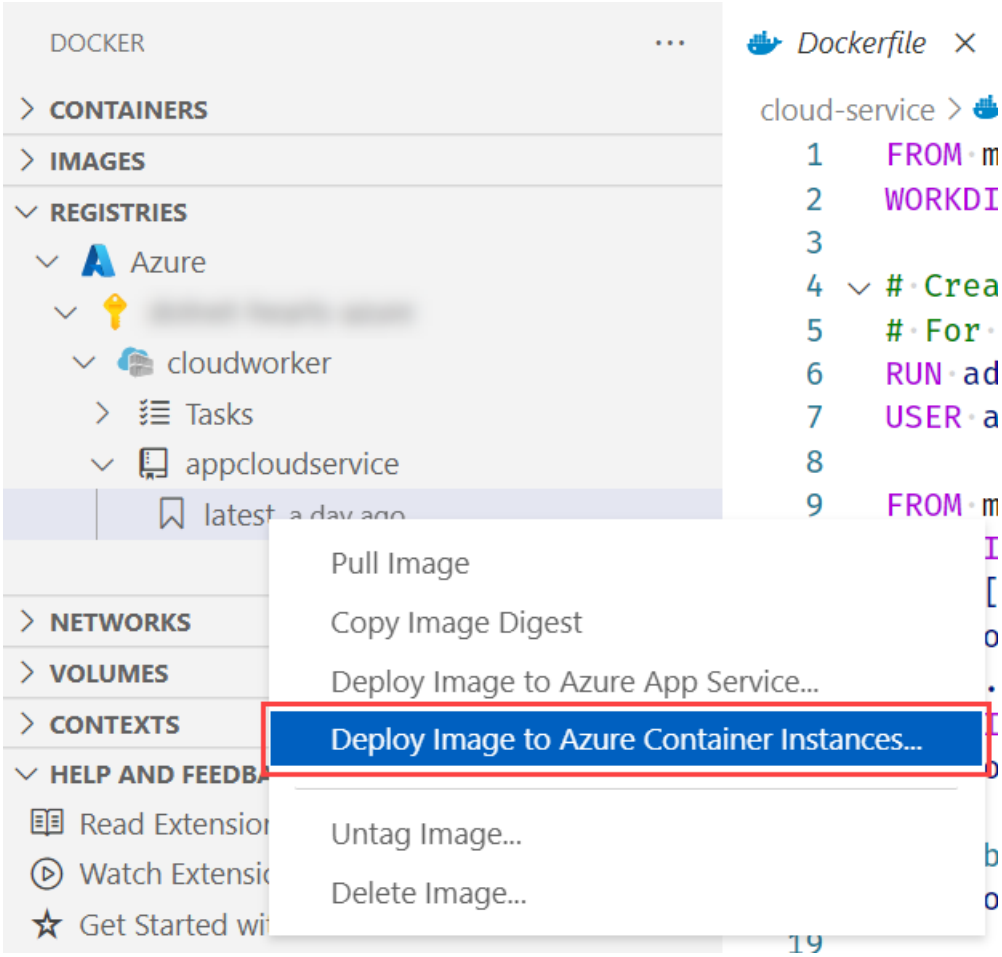
若要验证映像是否已成功推送到容器注册表，请导航到 Azure 门户。打开容器注册表资源，选择“服务”下的“存储库”。你应该会看到该映像。

作为容器实例部署

从 Visual Studio Code 中的“活动栏”中选择“Docker”。展开“注册表”节点，再选择“连接注册表”。当系统提示时，选择“Azure”并登录（如果需要）。



展开“注册表”节点，选择“Azure”，再选择“订阅”>“容器注册表”>“映像”，然后右键单击标记。选择“将映像部署到 Azure 容器实例”。



若要创建容器实例，需要使用 `az container create` 命令创建容器组。

```
az container create -g <resource group> \
  --name <instance name> \
  --image <registry name>.azurecr.io/<image name>:latest \
  --registry-password <password>
```

请提供适当的值：

- `<resource group>`：在本教程中使用的资源组名称。
- `<instance name>`：容器实例的名称。
- `<registry name>`：容器注册表的名称。
- `<image name>`：映像的名称。
- `<password>`：容器注册表的密码，可以从 Azure 门户的容器注册表资源 >“访问密钥”获取此密码。

若要创建容器实例，需要在 Azure 门户中创建一个新资源。

1. 选择与上一部分相同的“订阅”和相应的“资源组”。
2. 输入容器名称— `appcloudservice-container`。
3. 选择与上一个“位置”选择相对应的“区域”。
4. 对于映像源，选择“Azure 容器注册表”。
5. 按上一步骤中提供的名称选择“注册表”。
6. 选择“映像”和“映像标记”。
7. 选择“查看 + 创建”。
8. 假定“验证已通过”，选择“创建”。

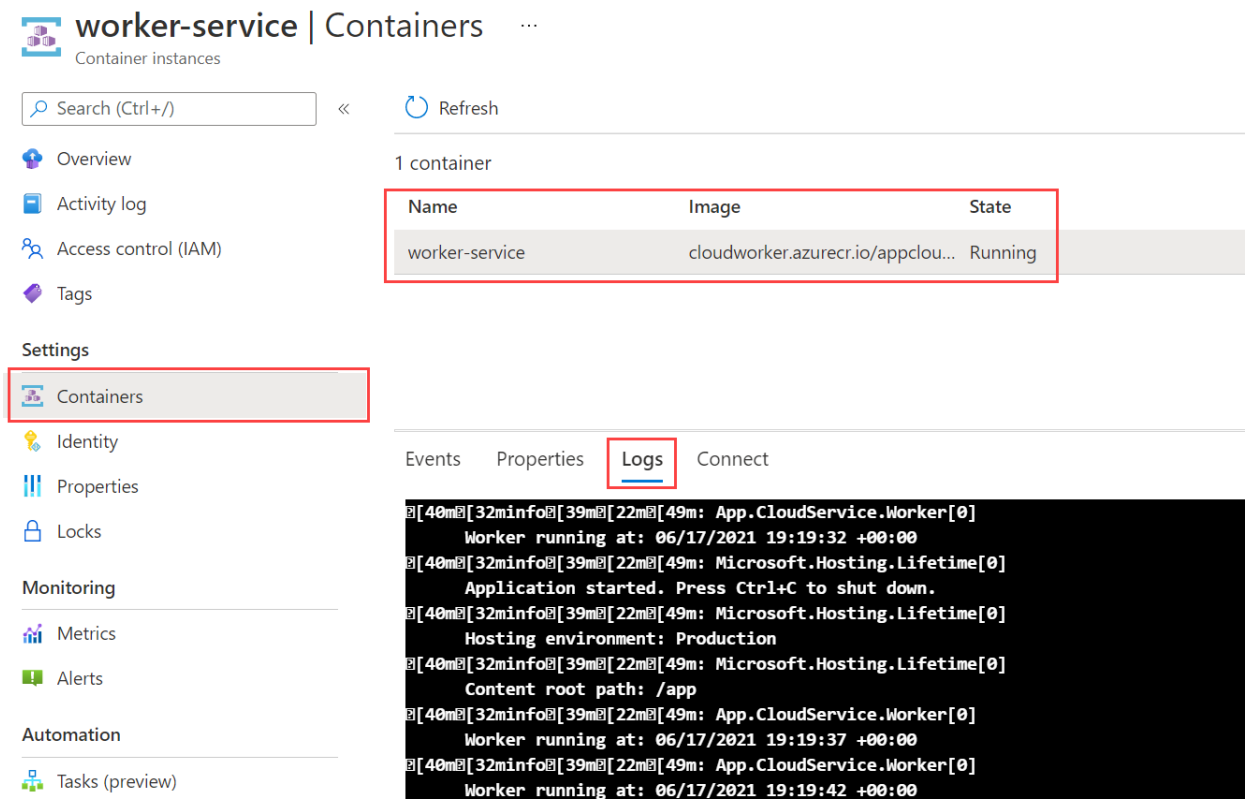
创建资源可能需要一些时间，一旦创建，请选择“转到资源”按钮。

有关详细信息，请参阅快速入门：创建 Azure 容器实例。

验证服务功能

容器实例创建后，它会立即开始运行。

若要验证辅助角色服务是否正常运行，请导航到容器实例资源中的 Azure 门户，选择“容器”选项。



你将看到容器及其当前状态。在本示例中，其状态为“正在运行”。选择“日志”查看 .NET 辅助角色服务输出。

若要验证辅助角色服务是否正常运行，可以查看正在运行的应用程序的日志。使用 `az container logs` 命令：

```
az container logs -g <resource group> --name <instance name>
```

请提供适当的值：

- `<resource group>`：在本教程中使用的资源组名称。
- `<instance name>`：容器实例的名称。

将看到 .NET 辅助角色服务输出日志，这意味着已成功将容器化应用部署到 ACI。

另请参阅

- .NET 中的辅助角色服务
- 在 `BackgroundService` 内使用作用域服务
- 使用 `BackgroundService` 创建 Windows 服务
- 实现 `IHostedService` 接口
- 教程：使 .NET Core 应用程序容器化

.NET 中的缓存

2021/11/16 •

本文介绍各种缓存机制。缓存指在中间层中存储数据的行为，该行为可使后续数据检索更快。从概念上讲，缓存是一种性能优化策略和设计考虑因素。缓存可以显著提高应用性能，方法是提高不常更改（或检索成本高）的数据的就绪性。本文介绍两种主要的缓存，并提供这两种的示例源代码：

- `Microsoft.Extensions.Caching.Memory`
- `Microsoft.Extensions.Caching.Distributed`

> **IMPORTANT**
> .NET 有两个 `MemoryCache` 类，一个在 `System.Runtime.Caching` 命名空间中，另一个在 `Microsoft.Extensions.Caching` 命名空间中：
>
> - System.Runtime.Caching.MemoryCache
> - Microsoft.Extensions.Caching.Memory.MemoryCache
>
> 虽然本文重点介绍缓存，但不包括 `System.Runtime.Caching` NuGet 包。所有对 `MemoryCache` 的引用都在 `Microsoft.Extensions.Caching` 命名空间内。

所有 `Microsoft.Extensions.*` 包都具有依赖项注入 (DI) 就绪性，并且 IMemoryCache 和 IDistributedCache 接口都可以用作服务。

内存中缓存

本部分将介绍 `Microsoft.Extensions.Caching.Memory` 包。IMemoryCache 的当前实现是 ConcurrentDictionary<TKey,TValue> 的包装器，公开功能丰富的 API。缓存中的项由 ICacheEntry 表示，可以是任何 `object`。内存中缓存解决方案适用于在单个服务器中运行的应用，其中所有缓存数据在应用进程中租用内存。

> **TIP**
> 对于多服务器缓存场景，请考虑使用分布式缓存方法替代内存中缓存。

内存中缓存 API

缓存的使用者可控制可调过期和绝对过期：

- ICacheEntry.AbsoluteExpiration
- ICacheEntry.AbsoluteExpirationRelativeToNow
- ICacheEntry.SlidingExpiration

设置过期后，如果未在过期时间安排内访问缓存中的项，将导致这些项被逐出。使用者可通过 MemoryCacheEntryOptions 使用其他选项来控制缓存项。每个 ICacheEntry 都与 MemoryCacheEntryOptionMemoryCacheEntryOptions 配对，后者使用 IChangeToken 公开过期逐出功能，使用 CacheItemPriority 设置优先级，并控制 ICacheEntry.Size。请考虑以下扩展方法：

- MemoryCacheEntryExtensions.AddExpirationToken
- MemoryCacheEntryExtensions.RegisterPostEvictionCallback
- MemoryCacheEntryExtensions.SetSize

- MemoryCacheEntryExtensions.SetPriority

内存中缓存示例

若要使用默认 IMemoryCache 实现，请调用 AddMemoryCache 扩展方法，以向 DI 注册所有必需的服务。在下面的代码示例中，泛型主机用于公开 ConfigureServices 功能：

```
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services => services.AddMemoryCache())
    .Build();
```

你可以以不同的方式访问 `IMemoryCache`，例如构造函数注入，具体取决于你的 .NET 工作负载。在此示例中，在 `host` 上使用 `IServiceProvider` 实例，并调用泛型 GetRequiredService<T>(IServiceProvider) 扩展方法：

```
IMemoryCache cache =
    host.Services.GetRequiredService<IMemoryCache>();
```

注册内存中缓存服务并通过 DI 解析后，即可开始缓存。此示例循环访问英文字母表“A”到“Z”中的字母。存在一个 `record`，它保存对字母的引用并生成消息。

```
record AlphabetLetter(char Letter)
{
    internal string Message =>
        $"The '{Letter}' character is the {Letter - 64} letter in the English alphabet.";
}
```

该示例包含一个帮助程序函数，该函数会循环访问字母表字母：

```
static async ValueTask IterateAlphabetAsync(
    Func<char, Task> asyncFunc)
{
    for (char letter = 'A'; letter <= 'Z'; ++ letter)
    {
        await asyncFunc(letter);
    }

    Console.WriteLine();
}
```

在前述 C# 代码中：

- `Func<char, Task> asyncFunc` 在每次迭代时等待，并传递当前 `letter`。
- 处理完所有字母后，一个空白行会写入到控制台。

若要将项添加到缓存，请调用 `Create` 或 `Set` API：

```
await IterateAlphabetAsync(letter =>
{
    MemoryCacheEntryOptions options = new()
    {
        AbsoluteExpirationRelativeToNow =
            TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
    };

    _ = options.RegisterPostEvictionCallback(OnPostEviction);

    AlphabetLetter alphabetLetter =
        cache.Set(
            letter, new AlphabetLetter(letter), options);

    Console.WriteLine($"{alphabetLetter.Letter} was cached.");

    return Task.Delay(
        TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});
```

在前述 C# 代码中：

- 等待 `IterateAlphabetAsync` 的调用。
- `Func<char, Task> asyncFunc` 使用 lambda 来表示。
- `MemoryCacheEntryOptions` 是以相对于现在的绝对过期来实例化的。
- 逐出后回叫已注册。
- `AlphabetLetter` 对象已实例化，并随 `letter` 和 `options` 一起传递到 Set 中。
- 该字母缓存时写入控制台。
- 最后，将返回 Task.Delay。

对于字母表中的每个字母，将写入一个包含过期和逐出后回叫的缓存项。

逐出后回叫会将逐出的值的详细信息写入控制台：

```
static void OnPostEviction(
    object key, object letter, EvictionReason reason, object state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{alphabetLetter.Letter} was evicted for {reason}.");
    }
};
```

填充缓存后，将等待另一个对 `IterateAlphabetAsync` 的调用，但这次将调用 IMemoryCache.TryGetValue：

```
await IterateAlphabetAsync(letter =>
{
    if (cache.TryGetValue(letter, out object? value) &&
        value is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{letter} is still in cache. {alphabetLetter.Message}");
    }

    return Task.CompletedTask;
});
```

如果 `cache` 包含 `letter` 键，并且 `value` 是 `AlphabetLetter` 的实例，该键将写入控制台。如果 `letter` 键不在缓存中，会逐出该键并调用其逐出后回叫。

其他扩展方法

`IMemoryCache` 具有许多方便的扩展方法，其中包括异步 `GetOrCreateAsync`：

- CacheExtensions.Get
- CacheExtensions.GetOrCreate
- CacheExtensions.GetOrCreateAsync
- CacheExtensions.Set
- CacheExtensions.TryGetValue

将其放在一起

整个示例应用源代码是一个顶级程序，需要两个 NuGet 包：

- `Microsoft.Extensions.Caching.Memory`
- `Microsoft.Extensions.Hosting`

```
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services => services.AddMemoryCache())
    .Build();

IMemoryCache cache =
    host.Services.GetRequiredService<IMemoryCache>();

const int MillisecondsDelayAfterAdd = 50;
const int MillisecondsAbsoluteExpiration = 750;

static void OnPostEviction(
    object key, object letter, EvictionReason reason, object state)
{
    if (letter is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{alphabetLetter.Letter} was evicted for {reason}.");
    }
};

static async ValueTask IterateAlphabetAsync(
    Func<char, Task> asyncFunc)
{
    for (char letter = 'A'; letter <= 'Z'; ++ letter)
    {
        await asyncFunc(letter);
    }

    Console.WriteLine();
}

await IterateAlphabetAsync(letter =>
{
    MemoryCacheEntryOptions options = new()
    {
        AbsoluteExpirationRelativeToNow =
            TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
    };

    _ = options.RegisterPostEvictionCallback(OnPostEviction);

    AlphabetLetter alphabetLetter =
        cache.Set(
            letter, new AlphabetLetter(letter), options);
```

```
    Console.WriteLine($"{alphabetLetter.Letter} was cached.");

    return Task.Delay(
        TimeSpan.FromMilliseconds(MillisecondsDelayAfterAdd));
});

await IterateAlphabetAsync(letter =>
{
    if (cache.TryGetValue(letter, out object? value) &&
        value is AlphabetLetter alphabetLetter)
    {
        Console.WriteLine($"{letter} is still in cache. {alphabetLetter.Message}");
    }

    return Task.CompletedTask;
});

await host.RunAsync();

record AlphabetLetter(char Letter)
{
    internal string Message =>
        $"The '{Letter}' character is the {Letter - 64} letter in the English alphabet.";
}
```

可以调整 `MillisecondsDelayAfterAdd` 和 `MillisecondsAbsoluteExpiration` 值，以观察缓存项过期和逐出行为的变化。下面是运行此代码的示例输出，由于 .NET 事件具有不确定性，无法保证输出完全相同。

```
A was cached.
B was cached.
C was cached.
D was cached.
E was cached.
F was cached.
G was cached.
H was cached.
I was cached.
J was cached.
K was cached.
L was cached.
M was cached.
N was cached.
O was cached.
P was cached.
Q was cached.
R was cached.
S was cached.
T was cached.
U was cached.
V was cached.
W was cached.
X was cached.
Y was cached.
Z was cached.

Q is still in cache. The 'Q' character is the 17 letter in the English alphabet.
R is still in cache. The 'R' character is the 18 letter in the English alphabet.
S is still in cache. The 'S' character is the 19 letter in the English alphabet.
T is still in cache. The 'T' character is the 20 letter in the English alphabet.
U is still in cache. The 'U' character is the 21 letter in the English alphabet.
D was evicted for Expired.
C was evicted for Expired.
G was evicted for Expired.
E was evicted for Expired.
F was evicted for Expired.
B was evicted for Expired.
M was evicted for Expired.
V is still in cache. The 'V' character is the 22 letter in the English alphabet.
H was evicted for Expired.
I was evicted for Expired.
J was evicted for Expired.
K was evicted for Expired.
L was evicted for Expired.
A was evicted for Expired.
N was evicted for Expired.
W is still in cache. The 'W' character is the 23 letter in the English alphabet.
O was evicted for Expired.
P was evicted for Expired.
X is still in cache. The 'X' character is the 24 letter in the English alphabet.
Y is still in cache. The 'Y' character is the 25 letter in the English alphabet.
Z is still in cache. The 'Z' character is the 26 letter in the English alphabet.
```

由于已设置绝对过期 (MemoryCacheEntryOptions.AbsoluteExpirationRelativeToNow)，因此最终将逐出所有缓存项。

辅助角色服务缓存

缓存数据的一种常见策略是独立于使用数据服务更新缓存。辅助角色服务模板是一个很好的示例，因为 BackgroundService 独立于其他应用程序代码(或在后台)运行。当托管 IHostedService 实现的应用程序开始运行时，相应的实现(在这种情况下为 `BackgroundService` 或"辅助角色")开始在同一进程中运行。这些托管服务通过 AddHostedService<THostedService>(IServiceCollection) 扩展方法向 DI 注册为单一实例。可以使用任何服务生存期向 DI 注册其他服务。

> **IMPORTANT**
>
> 请务必要了解服务生存期。调用 AddMemoryCache 以注册所有内存中缓存服务时，服务将注册为单一实例。

照片服务场景

假设你在开发依赖于可通过 HTTP 访问的第三方 API 的照片服务。这种照片数据不会经常更改，但数据量很大。每张照片都由一个简单的 `record` 表示：

```
namespace CachingExamples.Memory
{
    public record Photo(
        int AlbumId,
        int Id,
        string Title,
        string Url,
        string ThumbnailUrl);
}
```

在下面的示例中，你将看到若干在 DI 中注册的服务。每个服务承担单一责任。

```
using System.Collections.Generic;
using CachingExamples.Memory;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

using IHost host = Host.CreateDefaultBuilder(args)
    .ConfigureServices(services =>
    {
        services.AddMemoryCache();
        services.AddHttpClient<CacheWorker>();
        services.AddHostedService<CacheWorker>();
        services.AddScoped<PhotoService>();
        services.AddSingleton(typeof(CacheSignal<>));
    })
    .Build();

await host.StartAsync();
```

在前述 C# 代码中：

- 泛型主机使用默认值创建。
- 内存中缓存服务使用 AddMemoryCache 注册。
- 使用 AddHttpClient<TClient>(IServiceCollection) 为 `CacheWorker` 类注册了一个 `HttpClient` 实例。
- `CacheWorker` 类使用 AddHostedService<THostedService>(IServiceCollection) 注册。
- `PhotoService` 类使用 AddScoped<TService>(IServiceCollection) 注册。
- `CacheSignal<T>` 类使用 AddSingleton 注册。
- `host` 由生成器实例化，并异步启动。

`PhotoService` 负责获取符合给定条件（或 `filter`）的照片：

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Logging;

namespace CachingExamples.Memory
{
    public sealed class PhotoService
    {
        private readonly IMemoryCache _cache;
        private readonly CacheSignal<Photo> _cacheSignal;
        private readonly ILogger<PhotoService> _logger;

        public PhotoService(
            IMemoryCache cache,
            CacheSignal<Photo> cacheSignal,
            ILogger<PhotoService> logger) =>
            (_cache, _cacheSignal, _logger) = (cache, cacheSignal, logger);

        public async IAsyncEnumerable<Photo> GetPhotosAsync(Func<Photo, bool>? filter = default)
        {
            try
            {
                await _cacheSignal.WaitAsync();

                Photo[] photos =
                    await _cache.GetOrCreateAsync(
                        "Photos", _ =>
                        {
                            _logger.LogWarning("This should never happen!");

                            return Task.FromResult(Array.Empty<Photo>());
                        });

                // If no filter is provided, use a pass-thru.
                filter ??= _ => true;

                foreach (Photo? photo in photos)
                {
                    if (photo is not null && filter(photo))
                    {
                        yield return photo;
                    }
                }
            }
            finally
            {
                _cacheSignal.Release();
            }
        }
    }
}
```

在前述 C# 代码中：

- 构造函数需要 `IMemoryCache`、`CacheSignal<Photo>` 和 `ILogger`。
- `GetPhotosAsync` 方法：
  - 定义 `Func<Photo, bool> filter` 参数，并返回 `IAsyncEnumerable<Photo>`。
  - 调用 `_cacheSignal.WaitAsync()` 并等待它释放，这可确保在访问缓存前先填充缓存。
  - 调用 `_cache.GetOrCreateAsync()`，异步获取缓存中的所有照片。
  - `factory` 参数记录警告，并返回空照片数组 - 这应该永远不会发生。
  - 缓存中的每张照片都使用 `yield return` 进行循环访问、筛选和具体化。

- 最后，缓存信号已重置。

此服务的使用者可以调用 `GetPhotosAsync` 方法，并相应地处理照片。不需要 `HttpClient`，因为缓存包含照片。

`CacheWorker` 是 BackgroundService 的子类：

```
using System;
using System.Net.Http;
using System.Net.Http.Json;
using System.Threading;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Memory;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace CachingExamples.Memory
{
    public class CacheWorker : BackgroundService
    {
        private readonly ILogger<CacheWorker> _logger;
        private readonly HttpClient _httpClient;
        private readonly CacheSignal<Photo> _cacheSignal;
        private readonly IMemoryCache _cache;
        private readonly TimeSpan _updateInterval = TimeSpan.FromHours(3);

        private const string Url = "https://jsonplaceholder.typicode.com/photos";

        public CacheWorker(
            ILogger<CacheWorker> logger,
            HttpClient httpClient,
            CacheSignal<Photo> cacheSignal,
            IMemoryCache cache) =>
            (_logger, _httpClient, _cacheSignal, _cache) = (logger, httpClient, cacheSignal, cache);

        public override async Task StartAsync(CancellationToken cancellationToken)
        {
            await _cacheSignal.WaitAsync();
            await base.StartAsync(cancellationToken);
        }

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            while (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogInformation("Updating cache.");

                try
                {
                    Photo[]? photos =
                        await _httpClient.GetFromJsonAsync<Photo[]>(
                            Url, stoppingToken);

                    if (photos is { Length: > 0 })
                    {
                        _cache.Set("Photos", photos);
                        _logger.LogInformation(
                            "Cache updated with {Count:#,#} photos.", photos.Length);
                    }
                    else
                    {
                        _logger.LogWarning(
                            "Unable to fetch photos to update cache.");
                    }
                }
                finally
                {
                    _cacheSignal.Release();
                }
```

```
                }

                try
                {
                    _logger.LogInformation(
                        "Will attempt to update the cache in {Hours} hours from now.",
                        _updateInterval.Hours);

                    await Task.Delay(_updateInterval, stoppingToken);
                }
                catch (OperationCanceledException)
                {
                    _logger.LogWarning("Cancellation acknowledged: shutting down.");
                    break;
                }
            }
        }
    }
}
```

> **IMPORTANT**
>
> 需要 `override` BackgroundService.StartAsync 并调用 `await _cacheSignal.WaitAsync()`，防止在 `CacheWorker` 的启动和调用 `PhotoService.GetPhotosAsync` 之间出现争用条件。

在前述 C# 代码中：

- 构造函数需要 `ILogger`、`HttpClient``CacheSignal<Photo>` 和 `IMemoryCache`。
- 将 `_updateInterval` 定义为 3 小时。
- `ExecuteAsync` 方法：
  - 在应用运行时循环。
  - 向 `"https://jsonplaceholder.typicode.com/photos"` 发出 HTTP 请求，将响应映射为一组 `Photo` 对象。
  - 照片组放置在 `"Photos"` 键下的 `IMemoryCache` 中。
  - `_cacheSignal.Release()` 被调用，释放任何正在等待信号的使用者。
  - 根据更新间隔，等待对 Task.Delay 的调用。
  - 延迟 3 小时后，缓存将再次更新。

异步信号在一个泛型类型受约束的单一实例中基于一个封装的 SemaphoreSlim 实例。`CacheSignal<T>` 依赖于 `SemaphoreSlim` 实例：

```
using System.Threading;
using System.Threading.Tasks;

namespace CachingExamples.Memory
{
    public sealed class CacheSignal<T>
    {
        private readonly SemaphoreSlim _semaphore = new(1, 1);

        /// <summary>
        /// Exposes a <see cref="Task"/> that represents the asynchronous wait operation.
        /// When signaled (consumer calls <see cref="Release"/>), the
        /// <see cref="Task.Status"/> is set as <see cref="TaskStatus.RanToCompletion"/>.
        /// </summary>
        public Task WaitAsync() => _semaphore.WaitAsync();

        /// <summary>
        /// Exposes the ability to signal the release of the <see cref="WaitAsync"/>'s operation.
        /// Callers who were waiting, will be able to continue.
        /// </summary>
        public void Release() => _semaphore.Release();
    }
}
```

在前述 C# 代码中，修饰器模式用于包装 `SemaphoreSlim` 的实例。由于 `CacheSignal<T>` 被注册为单一实例，它可以在所有服务生存期中与任何泛型类型（在这种情况下为 `Photo`）一起使用。它负责发出缓存的种子设定的信号。

分布式缓存

在某些场景中，需要使用分布式缓存 — 例如有多个应用服务器的情况。分布式缓存支持比内存中缓存方法更广的横向扩展。使用分布式缓存将缓存内存卸载到外部进程，但确实需要额外的网络 I/O 并会引入更多一点的延迟（即使是名义上）。

分布式缓存抽象是 `Microsoft.Extensions.Caching.Memory` NuGet 包的一部分，甚至还存在一个 `AddDistributedMemoryCache` 扩展方法。

Caution

AddDistributedMemoryCache 只应在开发和/或测试场景中使用，它不是可行的生产实现。

请考虑以下包中 `IDistributedCache` 的任何可用实现：

- `Microsoft.Extensions.Caching.SqlServer`
- `Microsoft.Extensions.Caching.StackExchangeRedis`
- `NCache.Microsoft.Extensions.Caching.OpenSource`

分布式缓存 API

分布式缓存 API 比对应的内存中缓存 API 更原始一些。键值对更基本一些。内存中缓存键基于 `object`，而分布式键基于 `string`。对于内存中缓存，值可以是任何强类型的泛型，而分布式缓存中的值将保存为 `byte[]`。这并不是说各种实现不会公开强类型的泛型值，而是公开实现的详细信息。

创建值

若要在分布式缓存中创建值，请调用其中一个 set API：

- IDistributedCache.SetAsync
- IDistributedCache.Set

通过使用内存中缓存示例中的 `AlphabetLetter` 记录，你可以将对象串行化为 JSON，然后将 `string` 编码为 `byte[]`：

```
DistributedCacheEntryOptions options = new()
{
    AbsoluteExpirationRelativeToNow =
        TimeSpan.FromMilliseconds(MillisecondsAbsoluteExpiration)
};

AlphabetLetter alphabetLetter = new(letter);
string json = JsonSerializer.Serialize(alphabetLetter);
byte[] bytes = Encoding.UTF8.GetBytes(json);

await cache.SetAsync(letter.ToString(), bytes, options);
```

与内存中缓存非常类似，缓存项可能有一些选项有助于微调它们在缓存中的存在 — 在这种情况下为 DistributedCacheEntryOptions。

创建扩展方法

有几种便利的扩展方法可用于创建值，这些方法有助于避免将对象的 `string` 表示形式编码为 `byte[]`：

- DistributedCacheExtensions.SetStringAsync
- DistributedCacheExtensions.SetString

读取值

若要从分布式缓存读取值，请调用其中一个 Get API：

- IDistributedCache.GetAsync
- IDistributedCache.Get

```
AlphabetLetter? alphabetLetter = null;
byte[]? bytes = await cache.GetAsync(letter.ToString());
if (bytes is { Length: > 0 })
{
    string json = Encoding.UTF8.GetString(bytes);
    alphabetLetter = JsonSerializer.Deserialize<AlphabetLetter>(json);
}
```

从缓存读取缓存项后，可以从 `string` 中获取 UTF8 编码的 `byte[]` 表示形式

读取扩展方法

有几种便利的扩展方法可用于读取值，这些方法有助于避免将 `byte[]` 解码为对象的 `string` 表示形式：

- DistributedCacheExtensions.GetStringAsync
- DistributedCacheExtensions.GetString

更新值

无法使用单个 API 调用实际更新分布式缓存中的值，但值可以使用其中一个 Refresh API 重置其可调过期：

- IDistributedCache.RefreshAsync
- IDistributedCache.Refresh

如果需要更新实际值，则必须删除值，然后重新添加。

删除值

若要删除分布式缓存中的值，请调用其中一个 Remove API：

- IDistributedCache.RemoveAsync
- IDistributedCache.Remove

> **TIP**
> 尽管存在上述 API 的同步版本，但请注意分布式缓存的实现依赖于网络 I/O。因此，在更多情况下，更倾向于使用异步 API。

另请参阅

- .NET 中的依赖关系注入
- .NET 通用主机
- .NET 中的辅助角色服务
- 面向 .NET 开发人员的 Azure
- ASP.NET Core 中的内存中缓存
- ASP.NET Core 中的分布式缓存

LINQ 概述

2021/11/16 •

语言集成查询 (LINQ) 为 C# 和 Visual Basic 提供语言级查询功能和高阶函数 API，让你能够编写具有很高表达力度的声明性代码。

语言级查询语法

语言级查询语法如下：

```
var linqExperts = from p in programmers
                  where p.IsNewToLINQ
                  select new LINQExpert(p);
```

```
Dim linqExperts = From p in programmers
                  Where p.IsNewToLINQ
                  Select New LINQExpert(p)
```

同一个示例使用 `IEnumerable<T>` API 的情况如下：

```
var linqExperts = programmers.Where(p => p.IsNewToLINQ)
                             .Select(p => new LINQExpert(p));
```

```
Dim linqExperts = programmers.Where(Function(p) p.IsNewToLINQ).
                              Select(Function(p) New LINQExpert(p))
```

LINQ 具有很高的表达力度

假设你有一份宠物列表，但想要将它转换为字典，以便可以使用宠物的 `RFID` 值直接访问宠物信息。

传统的命令性代码如下：

```
var petLookup = new Dictionary<int, Pet>();

foreach (var pet in pets)
{
    petLookup.Add(pet.RFID, pet);
}
```

```
Dim petLookup = New Dictionary(Of Integer, Pet)()

For Each pet in pets
    petLookup.Add(pet.RFID, pet)
Next
```

代码的意图不是创建新的 `Dictionary<int, Pet>` 并通过循环在其中添加条目，而是将现有列表转换为字典！ LINQ 维持这种意图，而命令性代码则不会。

等效的 LINQ 表达式如下：

```
var petLookup = pets.ToDictionary(pet => pet.RFID);
```

```
Dim petLookup = pets.ToDictionary(Function(pet) pet.RFID)
```

使用 LINQ 的代码非常有效，因为在程序员的推理过程中，LINQ 能够在意图与代码之间找到合理的平衡。另一个好处就是精简代码。想像一下，如果能够像上面一样将大部分的基本代码减掉 1/3，情况会怎样？ 真好，对吧？

LINQ 提供程序简化数据访问

对于生产环境中的软件，其重要功能块的任务不外乎就是来自某些源（数据库、JSON、XML 等）的数据。通常，这就需要用户学习每个数据源的新 API，而这是一个枯燥的过程。LINQ 可将用于数据访问的常用元素抽象化成查询语法，不过你选择哪种数据源，这种语法看上去都是相同的，因而简化了此任务。

这将查找具有特定属性值的所有 XML 元素：

```
public static IEnumerable<XElement> FindAllElementsWithAttribute(XElement documentRoot, string elementName,
                                        string attributeName, string value)
{
    return from el in documentRoot.Elements(elementName)
           where (string)el.Element(attributeName) == value
           select el;
}
```

```
Public Shared Function FindAllElementsWithAttribute(documentRoot As XElement, elementName As String,
                                        attributeName As String, value As String) As IEnumerable(Of
XElement)
    Return From el In documentRoot.Elements(elementName)
           Where el.Element(attributeName).ToString() = value
           Select el
End Function
```

为了执行此任务而编写代码来手动遍历 XML 文档会带来重重困难。

LINQ 提供程序的作用不仅仅是与 XML 交互。Linq to SQL 是适用于 MSSQL Server 数据库的极其简练的对象关系映射器 (ORM)。使用 Json.NET 库可以通过 LINQ 有效遍历 JSON 文档。此外，如果没有哪个库可以解决你的需要，你还可以编写自己的 LINQ 提供程序！

使用查询语法的理由

为什么要使用查询语法？ 这是用户经常提出的一个问题。无论如何，对于下面的代码：

```
var filteredItems = myItems.Where(item => item.Foo);
```

```
Dim filteredItems = myItems.Where(Function(item) item.Foo)
```

要比下面的代码简洁得多：

```
var filteredItems = from item in myItems
                    where item.Foo
                    select item;
```

```
Dim filteredItems = From item In myItems
                    Where item.Foo
                    Select item
```

难道 API 语法不比查询语法更简洁吗？

不是。查询语法允许使用 let 子句，这样，便可以在表达式的作用域内引入和绑定变量，然后在表达式的后续片段中使用该变量。只使用 API 语法重现相同的代码也是可行的，不过，这很可能会导致代码难以阅读。

那么，问题来了，**只使用查询语法可以吗？**

在以下情况下，此问题的答案是可以：

- 现有的基本代码已使用查询语法。
- 由于复杂性的问题，需要在查询中限定变量的作用域。
- 你偏好使用查询语法，并且它不会使基本代码变得混乱。

在以下情况下，此问题的答案是 **不可以**...

- 现有的基本代码已使用 API 语法
- 不需要在查询中限定变量的作用域
- 你偏好使用 API 语法，并且它不会使基本代码变得混乱

基本 LINQ

有关 LINQ 示例的完整列表，请访问 101 个 LINQ 示例。

以下示例简单演示了 LINQ 的一些重要片段。没有办法演示完整的代码，因为 LINQ 提供的功能比此处演示的要多。

语句构成 - `Where`、`Select` 和 `Aggregate`

```
// Filtering a list.
var germanShepherds = dogs.Where(dog => dog.Breed == DogBreed.GermanShepherd);

// Using the query syntax.
var queryGermanShepherds = from dog in dogs
                           where dog.Breed == DogBreed.GermanShepherd
                           select dog;

// Mapping a list from type A to type B.
var cats = dogs.Select(dog => dog.TurnIntoACat());

// Using the query syntax.
var queryCats = from dog in dogs
                select dog.TurnIntoACat();

// Summing the lengths of a set of strings.
int seed = 0;
int sumOfStrings = strings.Aggregate(seed, (s1, s2) => s1.Length + s2.Length);
```

```
' Filtering a list.
Dim germanShepherds = dogs.Where(Function(dog) dog.Breed = DogBreed.GermanShepherd)

' Using the query syntax.
Dim queryGermanShepherds = From dog In dogs
                           Where dog.Breed = DogBreed.GermanShepherd
                           Select dog

' Mapping a list from type A to type B.
Dim cats = dogs.Select(Function(dog) dog.TurnIntoACat())

' Using the query syntax.
Dim queryCats = From dog In dogs
                Select dog.TurnIntoACat()

' Summing the lengths of a set of strings.
Dim seed As Integer = 0
Dim sumOfStrings As Integer = strings.Aggregate(seed, Function(s1, s2) s1.Length + s2.Length)
```

平展列表的列表

```
// Transforms the list of kennels into a list of all their dogs.
var allDogsFromKennels = kennels.SelectMany(kennel => kennel.Dogs);
```

```
' Transforms the list of kennels into a list of all their dogs.
Dim allDogsFromKennels = kennels.SelectMany(Function(kennel) kennel.Dogs)
```

两个集之间的联合(使用自定义比较运算符)

```
public class DogHairLengthComparer : IEqualityComparer<Dog>
{
    public bool Equals(Dog a, Dog b)
    {
        if (a == null && b == null)
        {
            return true;
        }
        else if ((a == null && b != null) ||
                 (a != null && b == null))
        {
            return false;
        }
        else
        {
            return a.HairLengthType == b.HairLengthType;
        }
    }

    public int GetHashCode(Dog d)
    {
        // Default hashcode is enough here, as these are simple objects.
        return d.GetHashCode();
    }
}
...

// Gets all the short-haired dogs between two different kennels.
var allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, new DogHairLengthComparer());
```

```
Public Class DogHairLengthComparer
  Inherits IEqualityComparer(Of Dog)

  Public Function Equals(a As Dog,b As Dog) As Boolean
      If a Is Nothing AndAlso b Is Nothing Then
          Return True
      ElseIf (a Is Nothing AndAlso b IsNot Nothing) OrElse (a IsNot Nothing AndAlso b Is Nothing) Then
          Return False
      Else
          Return a.HairLengthType = b.HairLengthType
      End If
  End Function

  Public Function GetHashCode(d As Dog) As Integer
      ' Default hashcode is enough here, as these are simple objects.
      Return d.GetHashCode()
  End Function
End Class

...

' Gets all the short-haired dogs between two different kennels.
Dim allShortHairedDogs = kennel1.Dogs.Union(kennel2.Dogs, New DogHairLengthComparer())
```

两个集之间的交集

```
// Gets the volunteers who spend share time with two humane societies.
var volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     new VolunteerTimeComparer());
```

```
' Gets the volunteers who spend share time with two humane societies.
Dim volunteers = humaneSociety1.Volunteers.Intersect(humaneSociety2.Volunteers,
                                                     New VolunteerTimeComparer())
```

中间件排序

```
// Get driving directions, ordering by if it's toll-free before estimated driving time.
var results = DirectionsProcessor.GetDirections(start, end)
              .OrderBy(direction => direction.HasNoTolls)
              .ThenBy(direction => direction.EstimatedTime);
```

```
' Get driving directions, ordering by if it's toll-free before estimated driving time.
Dim results = DirectionsProcessor.GetDirections(start, end).
                OrderBy(Function(direction) direction.HasNoTolls).
                ThenBy(Function(direction) direction.EstimatedTime)
```

实例属性的相等性

最后，我们演示一个更高级的示例：确定相同类型的两个实例的属性值是否相等（该示例摘自此 StackOverflow 文章，不过已做修改）：

```
public static bool PublicInstancePropertiesEqual<T>(this T self, T to, params string[] ignore) where T :
class
{
    if (self == null || to == null)
    {
        return self == to;
    }

    // Selects the properties which have unequal values into a sequence of those properties.
    var unequalProperties = from property in typeof(T).GetProperties(BindingFlags.Public |
BindingFlags.Instance)
                            where !ignore.Contains(property.Name)
                            let selfValue = property.GetValue(self, null)
                            let toValue = property.GetValue(to, null)
                            where !Equals(selfValue, toValue)
                            select property;
    return !unequalProperties.Any();
}
```

```
<System.Runtime.CompilerServices.Extension()>
Public Function PublicInstancePropertiesEqual(Of T As Class)(self As T, [to] As T, ParamArray ignore As
String()) As Boolean
    If self Is Nothing OrElse [to] Is Nothing Then
        Return self Is [to]
    End If

    ' Selects the properties which have unequal values into a sequence of those properties.
    Dim unequalProperties = From [property] In GetType(T).GetProperties(BindingFlags.Public Or
BindingFlags.Instance)
                            Where Not ignore.Contains([property].Name)
                            Let selfValue = [property].GetValue(self, Nothing)
                            Let toValue = [property].GetValue([to], Nothing)
                            Where Not Equals(selfValue, toValue) Select [property]
    Return Not unequalProperties.Any()
End Function
```

PLINQ

PLINQ(又称并行 LINQ)是 LINQ 表达式的并行执行引擎。换言之，LINQ 正则表达式可能会没有意义地在任意数量的线程之间并行化。为此，可以调用表达式前面的 `AsParallel()` 。

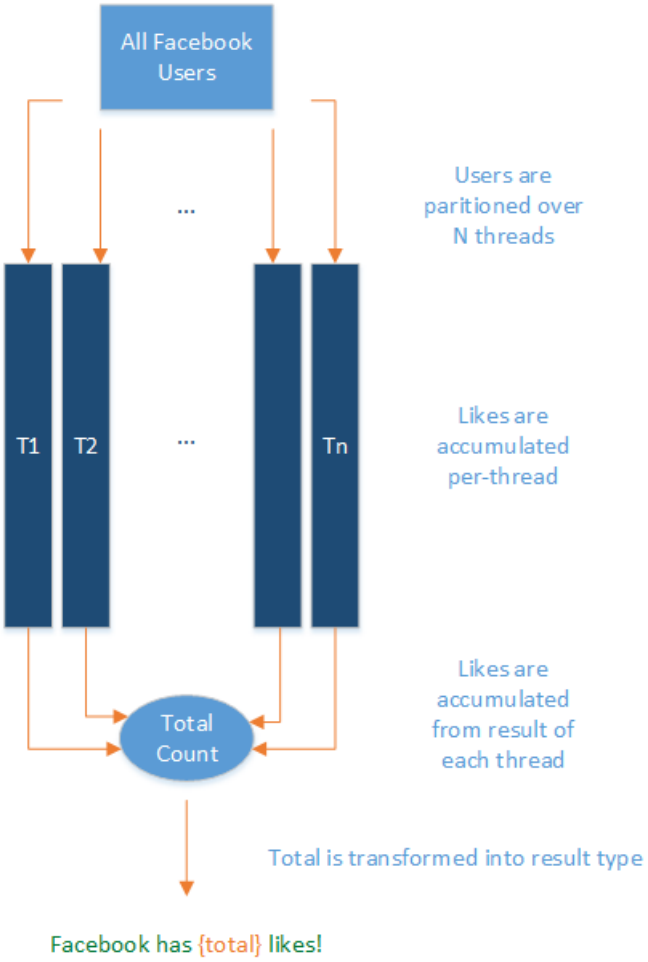
考虑以下情况：

```
public static string GetAllFacebookUserLikesMessage(IEnumerable<FacebookUser> facebookUsers)
{
    var seed = default(UInt64);

    Func<UInt64, UInt64, UInt64> threadAccumulator = (t1, t2) => t1 + t2;
    Func<UInt64, UInt64, UInt64> threadResultAccumulator = (t1, t2) => t1 + t2;
    Func<Uint64, string> resultSelector = total => $"Facebook has {total} likes!";

    return facebookUsers.AsParallel()
                        .Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector);
}
```

```
Public Shared GetAllFacebookUserLikesMessage(facebookUsers As IEnumerable(Of FacebookUser)) As String
{
    Dim seed As UInt64 = 0

    Dim threadAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim threadResultAccumulator As Func(Of UInt64, UInt64, UInt64) = Function(t1, t2) t1 + t2
    Dim resultSelector As Func(Of Uint64, string) = Function(total) $"Facebook has {total} likes!"

    Return facebookUsers.AsParallel().
                        Aggregate(seed, threadAccumulator, threadResultAccumulator, resultSelector)
}
```

此代码将会根据需要在系统线程之间将 `facebookUsers` 分区，累加每个并行线程上的类似项总计，累加每个线程计算的结果，然后将该结果投影为一个合理的字符串。

图示：



可通过 LINQ 能够轻松表达的可并行化 CPU 密集型作业（即，没有副作用的纯函数）非常适合使用 PLINQ 来处理。对于 *确实* 有副作用的作业，请考虑使用 任务并行库。

更多资源

- 101 LINQ 示例
- Linqpad，适用于 C#/F#/Visual Basic 的演练环境和数据库查询引擎
- EduLinq，帮助用户了解如何实现 LINQ 到对象的电子书

XML 文档和数据

2021/11/16 •

.NET Framework 提供了一组全面而集成的类，可用来方便地生成可以识别 XML 的应用程序。通过以下命名空间中的类，可以分析和编写 XML，编辑内存中的 XML 数据，进行数据验证以及 XSLT 转换。

- System.Xml
- System.Xml.XPath
- System.Xml.Xsl
- System.Xml.Schema
- System.Xml.Linq

若要查看完整的列表，请在 .NET API 浏览器上搜索“System.Xml”。

这些命名空间中的类支持万维网联合会 (W3C) 建议。例如：

- System.Xml.XmlDocument 类可实现 W3C 文档对象模型 (DOM) 级别 1 核心和 DOM 级别 2 核心建议。
- System.Xml.XmlReader 和 System.Xml.XmlWriter 类支持 W3C XML 1.0 和 XML 中的命名空间建议。
- System.Xml.Schema.XmlSchemaSet 类中的架构支持 W3C XML 架构第 1 部分：结构和 XML 架构第 2 部分：数据类型建议。
- System.Xml.Xsl 命名空间中的类支持符合 W3C XSLT 1.0 建议的 XSLT 转换。

.NET Framework 中的 XML 类具有以下优点：

- **高效率。**通过 LINQ to XML (C#) 和 LINQ to XML (Visual Basic)，能够更轻松地使用 XML 编程，并且能够得到与 SQL 类似的查询体验。
- **扩展性。**.NET Framework 中的 XML 类可使用抽象基类和虚拟方法进行扩展。例如，您可以创建 XmlUrlResolver 类的一个派生类，用以将缓存流存储到本地磁盘。
- **可插入的体系结构。**.NET Framework 提供了一种体系结构，其中的组件可以相互利用，数据可以在各组件之间传送。例如，可以使用 XPathDocument 类来转换数据存储（例如，XmlDocument 或 XslCompiledTransform 对象），然后可将输出传送到另一个存储或作为 Web 服务的流返回。
- **性能。**为获得更佳应用程序性能，.NET Framework 中设计的某些 XML 类支持基于流的模型并具有以下特性：
  - 只进、拉出模型分析使用最小缓存 (XmlReader)。
  - 只进验证 (XmlReader)。
  - 游标式导航，可使创建的节点减少到单个虚拟节点，同时提供对文档的随机访问 (XPathNavigator)。

  为了在需要进行 XSLT 处理时都获得更佳性能，您可以使用 XPathDocument 类，这是一个用于 XPath 查询的经过优化的只读存储，旨在高效地与 XslCompiledTransform 类结合使用。
- **与 ADO.NET 集成。**XML 类和 ADO.NET 紧密集成，将关系数据和 XML 组合在一起。DataSet 类是从数据库中检索到的数据在内存中的缓存。DataSet 类能够使用 XmlReader 和 XmlWriter 类读取和写入 XML，以 XML 架构 (XSD) 形式保持其内部关系架构结构，并可以推断 XML 文档的架构结构。

本节内容

XML 处理选项 讨论用于处理 XML 数据的选项。

处理内存中 XML 数据 讨论用于处理内存中 XML 数据的三种模型：LINQ to XML (C#) 和 LINQ to XML (Visual Basic)、XmlDocument 类（基于 W3C 文档对象模型）以及 XPathDocument 类（基于 XPath 数据模型）。

XSLT 转换
描述如何使用 XSLT 处理器。

XML 架构对象模型 (SOM)
描述用于通过提供 XmlSchema 类加载和编辑架构来生成和处理 XML 架构 (XSD) 的类。

关系数据和 ADO.NET 的 XML 集成
描述 .NET Framework 如何通过 DataSet 对象和 XmlDataDocument 对象启用对数据的关系和分层表示形式的实时同步访问。

管理 XML 文档中的命名空间
描述 XmlNamespaceManager 类如何用于存储和维护命名空间信息。

System.Xml 类中的类型支持
描述如何将 XML 数据类型映射到 CLR 类型，如何转换 XML 类型，并描述 System.Xml 类中的其它类型支持功能。

相关章节

ADO.NET
提供如何使用 ADO.NET 访问数据的信息。

安全性
提供对 .NET Framework 安全系统的概述。

Microsoft.Data.Sqlite 概述

2021/11/16 •

Microsoft.Data.Sqlite 是用于 SQLite 的轻型 ADO.NET 提供程序。用于 SQLite 的 Entity Framework Core 提供程序就是基于此库而构建。但它还可以单独使用，也可以与其他数据访问库一起使用。

安装

可从 NuGet 获取最新的稳定版本。

- .NET Core CLI
- Visual Studio

```
dotnet add package Microsoft.Data.Sqlite
```

用法

此库实现了用于连接、命令、数据读取器等的常见 ADO.NET 抽象。

```
using (var connection = new SqliteConnection("Data Source=hello.db"))
{
    connection.Open();

    var command = connection.CreateCommand();
    command.CommandText =
    @"
        SELECT name
        FROM user
        WHERE id = $id
    ";
    command.Parameters.AddWithValue("$id", id);

    using (var reader = command.ExecuteReader())
    {
        while (reader.Read())
        {
            var name = reader.GetString(0);

            Console.WriteLine($"Hello, {name}!");
        }
    }
}
```

请参阅

- 连接字符串
- API 参考
- SQL 语法

.NET 中的并行处理、并发和异步编程

2021/11/16 •

借助 .NET，可以通过多种方式来编写异步代码，以提高应用程序的用户响应速度；并能编写并行代码，以使用多个执行线程来最大限度地提升用户计算机的性能。

本节内容

异步编程

介绍了 .NET 提供的异步编程机制。

并行编程

描述基于任务的编程模型，该模型简化了并行开发，使你能够通过固有方法编写高效、细化且可伸缩的并行代码，而不必直接处理线程或线程池。

线程处理

介绍了 .NET 提供的基本并发和同步机制。

异步概述

2021/11/16 •

不久前，人们通过购买更新的电脑和服务器来加快应用的速度，但是这种趋势已经停止了。事实上，趋势逆转了。手机配备 1ghz 单核 ARM 芯片，而服务器工作负荷转向 VM。用户仍青睐响应式 UI，企业所有者则希望拥有能随着其业务缩放的服务器。向手机和云的转变以及超过 30 亿使用 Internet 的人口引领着新软件模式的形成。

- 有了高应用存储率，客户端应用有望保持始终开启，始终连接的状态，并且可持续响应用户交互（例如，触摸）！
- 用户期望服务能够通过平稳地扩展和收缩来应对流量高峰。

异步编程是一项关键技术，可以直接处理多个核心上的阻塞 I/O 和并发操作。通过 C#、Visual Basic 和 F# 中易于使用的语言级异步编程模型，.NET 可为应用和服务提供使其变得可响应且富有弹性。

为什么要编写异步代码？

新型应用广泛使用文件和网络 I/O。默认情况下 I/O API 一般会阻塞，导致糟糕的用户体验和硬件利用率，除非希望学习和使用富有挑战的模式。基于任务的异步 API 和语言级异步编程模型改变了这种模型，只需了解几个新概念就可默认进行异步执行。

异步代码具有以下特点：

- 等待 I/O 请求返回的同时，可通过生成处理更多请求的线程，处理更多的服务器请求。
- 等待 I/O 请求的同时生成 UI 交互线程，并通过将长时间运行的工作转换到其他 CPU 核心，让 UI 的响应速度更快。
- 许多较新的 .NET APIs 都是异步的。
- 在 .NET 中编写异步代码很简单！

后续步骤

有关详细信息，请参阅异步深度剖析主题。

异步编程模式主题概述了 .NET 支持的三个异步编程模式：

- 异步编程模型 (APM)（旧版）
- 基于事件的异步模式 (EAP)（旧版）
- 基于任务的异步模式 (TAP)（建议用于新开发）

若要详细了解推荐的基于任务的编程模型，请参阅基于任务的异步编程主题。

深入了解异步

2021/11/16 •

使用基于 .NET 任务的异步模型可直接编写绑定 I/O 和 CPU 的异步代码。该模型由 `Task` 和 `Task<T>` 类型以及 C# 和 Visual Basic 中的 `async` 和 `await` 关键字公开。（有关特定语言的资源，请参见另请参阅部分。）本文解释如何使用 .NET 异步，并深入介绍其中使用的异步框架。

任务和 Task<T>

任务是用于实现称之为并发 Promise 模型的构造。简单地说，它们“承诺”，会在稍后完成工作，让你使用干净的 API 与 promise 协作。

- `Task` 表示不返回值的单个操作。
- `Task<T>` 表示返回 `T` 类型的值的单个操作。

请务必将任务理解为工作的异步抽象，而非 在线程之上的抽象。默认情况下，任务在当前线程上执行，且在适当时会将工作委托给操作系统。可选择性地通过 `Task.Run` API 显式请求任务在独立线程上运行。

任务会公开一个 API 协议来监视、等候和访问任务的结果值（如 `Task<T>`）。含有 `await` 关键字的语言集成可提供高级别抽象来使用任务。

任务运行时，使用 `await` 在任务完成前将控制让步于其调用方，可让应用程序和服务执行有用工作。任务完成后代码无需依靠回调或事件便可继续执行。语言和任务 API 集成会为你完成此操作。如果正在使用 `Task<T>`，任务完成时，`await` 关键字还将“打开”返回的值。下面进一步详细介绍了此工作原理。

可在基于任务的异步模式 (TAP) 主题中了解有关任务以及与任务交互的不同方法的详细信息。

深入了解针对绑定 I/O 的操作的任务

以下部分介绍了使用典型异步 I/O 调用时会出现的各种情况。让我们先看以下类的几个例子。

第一个示例方法 `GetHtmlAsync()` 调用异步方法，并返回一个活动任务，很可能尚未完成。第二个示例方法 `GetFirstCharactersCountAsync()` 还使用了 `async` 和 `await` 关键字对任务进行操作。

```
class DotNetFoundationClient
{
    // HttpClient is intended to be instantiated once per application, rather than per-use.
    private static readonly HttpClient s_client = new HttpClient();

    public Task<string> GetHtmlAsync()
    {
        // Execution is synchronous here
        var uri = new Uri("https://www.dotnetfoundation.org");

        return s_client.GetStringAsync(uri);
    }

    public async Task<string> GetFirstCharactersCountAsync(int count)
    {
        // Execution is synchronous here
        var uri = new Uri("https://www.dotnetfoundation.org");

        // Execution of GetFirstCharactersCountAsync() is yielded to the caller here
        // GetStringAsync returns a Task<string>, which is *awaited*
        var page = await s_client.GetStringAsync(uri);

        // Execution resumes when the client.GetStringAsync task completes,
        // becoming synchronous again.

        if (count > page.Length)
        {
            return page;
        }
        else
        {
            return page.Substring(0, count);
        }
    }
}
```

对 `GetStringAsync()` 的调用通过低级别 .NET 库进行(可能是调用其他异步方法),直到其到达 P/Invoke 互操作调用,进入本机网络库。本机库随后可能会调入系统 API 调用(例如 Linux 上套接字的 `write()`)。可能会使用 TaskCompletionSource 在本机/托管边界创建一个任务对象。将通过层向上传递任务对象,对其进行操作或直接返回,最后返回到初始调用方。

在上述第二个示例方法 `GetFirstCharactersCountAsync()` 中,`Task<T>` 对象直接从 `GetStringAsync` 返回。由于使用了 `await` 关键字,因此该方法会返回一个新建的任务对象。在 `GetFirstCharactersCountAsync` 方法中,控制权从此位置返回给调用方。Task<T> 对象的方法和属性使调用者能够监视任务的进度。GetFirstCharactersCountAsync 中剩余的代码执行完毕时,该任务便完成。

调用系统 API 后,请求位于内核空间,一路来到操作系统的网络子系统(例如 Linux 内核中的 `/net`)。此处操作系统将对网络请求进行异步 处理。所用操作系统不同,细节可能有所不同(可能会将设备驱动程序调用安排为发送回运行时的信号,或者会执行设备驱动程序调用然后 有一个信号发送回来),但最终都会通知运行时网络请求正在进行中。此时,设备驱动程序工作处于已计划、正在进行或是已完成(请求已"通过网络"发出),但由于这些均为异步进行,设备驱动程序可立即着手处理其他事项!

例如,在 Windows 中操作系统线程调用网络设备驱动程序并要求它通过表示操作的中断请求数据包 (IRP) 执行网络操作。设备驱动程序接收 IRP,调用网络,将 IRP 标记为"待定",并返回到操作系统。由于现在操作系统线程了解到 IRP 为"待定",因此无需再为此作业进行进一步操作,将其"返回",这样它就可用于完成其他工作。

请求完成且数据通过设备驱动程序返回后,会经由中断通知 CPU 新接收到的数据。处理中断的方式因操作系统不同而有所不同,但最终都会通过操作系统将数据传递到系统互操作调用(例如,Linux 中的中断处理程序将安排 IRQ 的下半部分通过操作系统异步向上传递数据)。这也是异步发生的! 在下一个可用线程能执行异步方法且"解包"已完成任务的结果前,结果会排入队列。

在整个过程中，关键点在于 **没有线程专用于运行任务**。尽管需要在一些上下文中执行工作（即，操作系统确实必须将数据传递到设备驱动程序并响应中断），但没有专用于 *等待* 数据从请求返回的线程。这让系统能处理更多的工作而不是等待某些 I/O 调用结束。

虽然这看上去需要完成许多工作，但以实际时间来计量，这远少于执行实际 I/O 工作所花费的时间。虽然不是完全精确，但此类调用可能的时间线如下所示：

0-1
——
–2-3

- 从点 `0` 到 `1` 所花费时间很长，直到异步方法将控制让步于其调用方才结束。
- 从点 `1` 到点 `2` 所用时间是花费在 I/O 上的时间，且 CPU 没有耗时。
- 最后，点 `2` 到点 `3` 所花费时间用于将控制（和可能的值）传递回异步方法，此时将再次执行。

这对服务器方案而言意味着什么？

此模型可很好地处理典型的服务器方案工作负荷。由于没有专用于阻止未完成任务的线程，因此服务器线程池可服务更多的 Web 请求。

考虑使用两个服务器：一个运行异步代码，一个不运行异步代码。对于本例，每个服务器只有 5 个线程可用于服务请求。此数字太小，不切合实际，仅供演示。

假设这两个服务器都接收 6 个并发请求。每个请求执行一个 I/O 操作。未运行异步代码的服务器必须对第 6 个请求排队，直到 5 个线程中的一个完成了 I/O 密集型工作并编写了响应。此时收到了第 20 个请求，由于队列过长，服务器可能会开始变慢。

运行有异步代码的服务器也需对第 6 个请求排队，但由于使用了 `async` 和 `await`，I/O 密集型工作开始时，每个线程都会得到释放，无需等到工作结束。收到第 20 个请求时，传入请求队列将变得很小（如果其中还有请求的话），且服务器不会变慢。

尽管这是一个人为想象的示例，但在现实世界中其工作方式与此类似。事实上，相比服务器将线程专用于接收到的每个请求，使用 `async` 和 `await` 能够使服务器多处理一个数量级的请求。

这对客户端方案而言意味着什么？

使用 `async` 和 `await` 对客户端应用带来的最大好处在于提高了响应能力。尽管可以手动生成线程让应用响应，但相比仅使用 `async` 和 `await`，生成线程的操作更加昂贵。特别是对于手机游戏等应用而言，在涉及 I/O 时尽可能少地影响 UI 线程，这点至关重要。

更重要的是，由于绑定 I/O 的工作在 CPU 上几乎没有耗时，所以将整个 CPU 线程专用于执行几乎没有任何作用的工作将是一种资源浪费。

此外，使用 `async` 方法将工作调度到 UI 线程（例如更新 UI）十分简单，且无需额外的工作（例如调用线程安全的委托）。

深入了解绑定 CPU 的操作的任务和 Task<T>

绑定 CPU 的 `async` 代码与绑定 I/O 的 `async` 代码有些许不同。由于工作在 CPU 上执行，无法解决线程专用于计算的问题。`async` 和 `await` 的运用使得可以与后台线程交互并让异步方法调用方可响应。请注意这不会为共享数据提供任何保护。如果正在使用共享数据，仍需要采用合适的同步策略。

这里详细介绍了绑定 CPU 的异步调用的方方面面：

```
public async Task<int> CalculateResult(InputData data)
{
    // This queues up the work on the threadpool.
    var expensiveResultTask = Task.Run(() => DoExpensiveCalculation(data));

    // Note that at this point, you can do some other work concurrently,
    // as CalculateResult() is still executing!

    // Execution of CalculateResult is yielded here!
    var result = await expensiveResultTask;

    return result;
}
```

`CalculateResult()` 在调用它的线程上执行。调用 `Task.Run` 时，它会在线程池上对昂贵的绑定 CPU 的操作 `DoExpensiveCalculation()` 进行排队，并收到一个 `Task<int>` 句柄。`DoExpensiveCalculation()` 最终在下一个可用线程上并行运行(很可能在另一个 CPU 内核上)。当 `DoExpensiveCalculation()` 在另一线程处理任务时，由于调用 `CalculateResult()` 的线程仍在执行，这时可能会出现并行工作的情况。

一旦遇到 `await`，`CalculateResult()` 执行会让步于调用方，在 `DoExpensiveCalculation()` 执行运算的同时，允许其他任务在当前线程执行。`DoExpensiveCalculation()` 完成后，结果会在主线程上排队等待运行。最后，主线程将返回执行得到 `DoExpensiveCalculation()` 结果的 `CalculateResult()`，。

异步为什么在此处会起作用？

`async` 和 `await` 是在需要可响应性时管理绑定 CPU 的工作的最佳实践。存在多个可将异步用于绑定 CPU 的工作的模式。请务必注意，使用异步成本有少许费用，不推荐紧凑循环使用它。如何编写此新功能的代码完全取决于你。

请参阅

- C# 中的异步编程
- 使用 Async 和 Await 的异步编程 (C#)
- F# 中的异步编程
- 使用 Async 和 Await 的异步编程 (Visual Basic)

异步编程模式

2021/11/16 •

.NET 提供了执行异步操作的三种模式：

- **基于任务的异步模式 (TAP)**，该模式使用单一方法表示异步操作的开始和完成。TAP 是在 .NET Framework 4 中引入的。这**是在 .NET 中进行异步编程的推荐方法。**C# 中的 async 和 await 关键词以及 Visual Basic 中的 Async 和 Await 运算符为 TAP 添加了语言支持。有关详细信息，请参阅基于任务的异步模式 (TAP)。
- 基于事件的异步模式 (EAP)，是提供异步行为的基于事件的旧模型。这种模式需要后缀为 `Async` 的方法，以及一个或多个事件、事件处理程序委托类型和 `EventArg` 派生类型。EAP 是在 .NET Framework 2.0 中引入的。建议新开发中不再使用这种模式。有关详细信息，请参阅基于事件的异步模式 (EAP)。
- 异步编程模型 (APM) 模式（也称为 IAsyncResult 模式），这是使用 IAsyncResult 接口提供异步行为的旧模型。在这种模式下，同步操作需要 `Begin` 和 `End` 方法（例如，`BeginWrite` 和 `EndWrite` 以实现异步写入操作）。不建议新的开发使用此模式。有关详细信息，请参阅异步编程模型 (APM)。

模式的比较

为了快速比较这三种模式的异步操作方式，请考虑使用从指定偏移量处起将指定量数据读取到提供的缓冲区中的 `Read` 方法：

```
public class MyClass
{
    public int Read(byte [] buffer, int offset, int count);
}
```

此方法对应的 TAP 将公开以下单个 `ReadAsync` 方法：

```
public class MyClass
{
    public Task<int> ReadAsync(byte [] buffer, int offset, int count);
}
```

对应的 EAP 将公开以下类型和成员的集：

```
public class MyClass
{
    public void ReadAsync(byte [] buffer, int offset, int count);
    public event ReadCompletedEventHandler ReadCompleted;
}
```

对应的 APM 将公开 `BeginRead` 和 `EndRead` 方法：

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

请参阅

- 深入了解异步
- C# 中的异步编程
- F# 中的异步编程
- 使用 Async 和 Await 的异步编程 (Visual Basic)

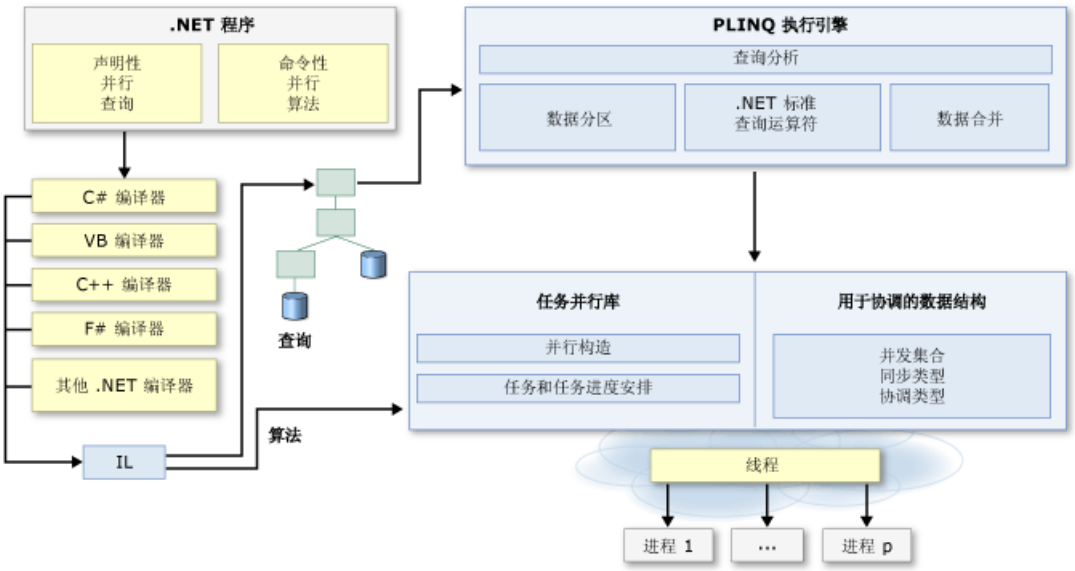
.NET 中的并行编程

2021/11/16 •

许多个人计算机和工作站都有多个 CPU 内核，以便多个线程能够同时执行。为了利用硬件，你可以对代码进行并行化，以将工作分摊在多个处理器上。

过去，并行化需要线程和锁的低级操作。Visual Studio 和 .NET 提供了运行时、类库类型和诊断工具，从而增强了对并行编程的支持。.NET Framework 4 中引入的这些功能简化了并行开发。你可以通过固有方法编写高效、细化且可伸缩的并行代码，而不必直接处理线程或线程池。

下图简要概述了 .NET 中的并行编程体系结构。



相关主题

任务并行库 (TPL)	提供针对 System.Threading.Tasks.Parallel 类的文档(包括 `For` 和 `ForEach` 循环的并行版本)，还提供了针对 System.Threading.Tasks.Task 类的文档(描绘了表示异步操作的首选方式)。
并行 LINQ (PLINQ)	LINQ to Objects 的并行实现，该实现显著提高了许多情况下的性能。
用于并行编程的数据结构	提供一些链接，这些链接指向有关线程安全集合类、轻量同步类型以及延迟初始化类型的文档。
并行诊断工具	提供一些链接，这些链接指向任务和并行堆栈的 Visual Studio 调试器窗口和并发可视化工具的文档。
PLINQ 和 TPL 的自定义分区程序	描述分区程序的工作方式，以及如何配置默认分区程序或创建新的分区程序。
任务计划程序	描述计划程序的工作方式，以及如何配置默认计划程序。

"	"
PLINQ 和 TPL 中的 Lambda 表达式	简要概述 C# 和 Visual Basic 中的 Lambda 表达式，并演示如何在 PLINQ 和任务并行库中使用这些表达式。
其他阅读材料	收录了指向其他信息以及 .NET 中并行编程示例资源的链接。

请参阅

- 异步概述
- 托管线程

任务并行库 (TPL)

2021/11/16 •

任务并行库 (TPL) 是 System.Threading 和 System.Threading.Tasks 空间中的一组公共类型和 API。TPL 的目的是通过简化将并行和并发添加到应用程序的过程来提高开发人员的工作效率。TPL 动态缩放并发的程度以最有效地使用所有可用的处理器。此外，TPL 还处理工作分区、ThreadPool 上的线程调度、取消支持、状态管理以及其他低级别的细节操作。通过使用 TPL，你可以在将精力集中于程序要完成的工作，同时最大程度地提高代码的性能。

自 .NET Framework 4 起，首选 TPL 编写多线程代码和并行代码。但是，并不是所有代码都适合并行化。例如，如果某个循环在每次迭代时只执行少量工作，或它在很多次迭代时都不运行，那么并行化的开销可能导致代码运行更慢。此外，像任何多线程代码一样，并行化会增加程序执行的复杂性。尽管 TPL 简化了多线程方案，但我们建议你对线程处理概念（例如，锁、死锁和争用条件）进行基本的了解，以便能够有效地使用 TPL。

相关文章

TITLE	ıı
数据并行	描述如何创建并行的 `for` 和 `foreach` 循环（在 Visual Basic 中为 `For` 和 `For Each`）。
基于任务的异步编程	描述如何通过使用 Parallel.Invoke 隐式创建和运行任务，或通过直接使用 Task 对象显式创建和运行任务。
数据流	描述如何使用 TPL 数据流库中的数据流组件处理多项运算，这些运算必须彼此通信，或在数据可用时处理数据。
数据和任务并行的潜在问题	描述一些常见缺陷以及如何避免它们。
并行 LINQ (PLINQ)	描述如何使用 LINQ 查询实现数据并行化。
并行编程	.NET 并行编程的顶级节点。

请参阅

- 使用 .NET Core 和 .NET Standard 并行编程的示例

数据并行（任务并行库）

2021/11/16 •

*数据并行*指的是对源集合或数组的元素同时（即，并行）执行相同操作的场景。在数据并行操作中，对源集合进行分区，以便多个线程能够同时在不同的网段上操作。

任务并行库 (TPL) 支持通过 System.Threading.Tasks.Parallel 类实现的数据并行。此类对 for 循环和 foreach 循环（Visual Basic 中的 `For` 和 `For Each`）提供了基于方法的并行执行。你为 Parallel.For 或 Parallel.ForEach 循环编写的循环逻辑与编写连续循环的相似。无需创建线程或列工作项。在基本循环中，不需要加锁。TPL 为你处理所有低级别的工作。有关使用 Parallel.For 和 Parallel.ForEach 的详细信息，请下载文档并行编程模式：了解并应用与 .NET Framework 4 的并行模式。下面的代码示例演示了一个简单的 `foreach` 循环及其并行等效项。

> **NOTE**
> 本文档使用 lambda 表达式在 TPL 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

```
// Sequential version
foreach (var item in sourceCollection)
{
    Process(item);
}

// Parallel equivalent
Parallel.ForEach(sourceCollection, item => Process(item));
```

```
' Sequential version
For Each item In sourceCollection
    Process(item)
Next

' Parallel equivalent
Parallel.ForEach(sourceCollection, Sub(item) Process(item))
```

并行循环运行时，TPL 将数据源进行分区，以便该循环可以同时对多个部分进行作用。在后台，任务计划程序基于系统资源和工作负荷来划分任务。如有可能，如果工作负荷变得不平衡了，计划程序将重新分配多个线程与处理器之间的工作。

> **NOTE**
> 你也可以提供你自己的自定义分区程序或计划程序。有关详细信息，请参阅 PLINQ 和 TPL 的自定义分区程序和任务计划程序。

Parallel.For 和 Parallel.ForEach 方法都有多个过载，可让你停止或中断循环执行，监视其它线程上循环的状态，保持本地线程状态，完成本地线程对象，控制并发程度等等。启用此功能的帮助器类型包括 ParallelLoopState、ParallelOptions、ParallelLoopResult、CancellationToken 和 CancellationTokenSource。

有关详细信息，请参阅并行编程模式：了解并应用与 .NET Framework 4 的并行模式。

PLINQ 支持使用声明性查询语法的数据并行。有关详细信息，请参阅并行 LINQ (PLINQ)。

相关主题

TITLE	ıı
如何：编写简单的 Parallel.For 循环	描述如何编写遍历任何数组或可变址 IEnumerable<T> 源集合的 For 循环。
如何：编写简单的 Parallel.ForEach 循环	描述如何编写遍历任何 IEnumerable<T> 源集合的 ForEach 循环。
如何：从 Parallel.For 循环停止或中断	描述如何停止或中断并行循环，以便所有线程都获得该操作的通知。
如何：编写具有线程局部变量的 Parallel.For 循环	描述如何编写 For 循环，该循环中每个线程都维持有对其它任何线程不可见的私有变量，以及如何在循环完成时，同步所有线程的结果。
如何：使用分区本地变量编写 Parallel.ForEach 循环	描述如何编写 ForEach 循环，该循环中每个线程都维持有对其它任何线程不可见的私有变量，以及如何在循环完成时，同步所有线程的结果。
如何：取消 Parallel.For 或 ForEach 循环	描述如何通过使用 System.Threading.CancellationToken 取消并行循环
如何：加快小型循环主体的速度	描述在循环主体极小时加快执行速度的方法。
任务并行库 (TPL)	提供任务并行库的概述。
并行编程	介绍.NET Framework 中的并行编程。

请参阅

- 并行编程

如何：编写简单的 Parallel.For 循环

2021/11/16 •

本主题包含两个示例，这两个示例阐释了 Parallel.For 方法。第一个示例使用 Parallel.For(Int64, Int64, Action<Int64>) 方法重载，而第二个示例使用 Parallel.For(Int32, Int32, Action<Int32>) 重载，它们是 Parallel.For 方法最简单的两个重载。如果不需要取消循环、中断循环迭代或保持任何线程本地状态，则可以使用 Parallel.For 方法的这两个重载。

> **NOTE**
> 本文档使用 lambda 表达式在 TPL 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

第一个示例计算单个目录中文件的大小。第二个示例计算两个矩阵的乘积。

目录大小示例

本示例是一个简单的命令行实用工具，用于计算一个目录中的文件总大小。它需要将单个目录路径作为参数，并报告该目录中文件的数量和总大小。在验证目录存在后，它会使用 Parallel.For 方法来枚举目录中的文件并确定其文件大小。然后，将每个文件大小添加到 `totalSize` 变量。请注意，此加法操作是通过调用 Interlocked.Add 来执行的，因此它是作为原子操作来执行的。否则，多个任务可能会同时尝试更新 `totalSize` 变量。

```
using System;
using System.IO;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main(string[] args)
   {
      long totalSize = 0;

      if (args.Length == 0) {
         Console.WriteLine("There are no command line arguments.");
         return;
      }
      if (! Directory.Exists(args[0])) {
         Console.WriteLine("The directory does not exist.");
         return;
      }

      String[] files = Directory.GetFiles(args[0]);
      Parallel.For(0, files.Length,
                   index => { FileInfo fi = new FileInfo(files[index]);
                              long size = fi.Length;
                              Interlocked.Add(ref totalSize, size);
                   } );
      Console.WriteLine("Directory '{0}':", args[0]);
      Console.WriteLine("{0:N0} files, {1:N0} bytes", files.Length, totalSize);
   }
}
// The example displaysoutput like the following:
//       Directory 'c:\windows\':
//       32 files, 6,587,222 bytes
```

```
Imports System.IO
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim totalSize As Long = 0

        Dim args() As String = Environment.GetCommandLineArgs()
        If args.Length = 1 Then
            Console.WriteLine("There are no command line arguments.")
            Return
        End If
        If Not Directory.Exists(args(1))
            Console.WriteLine("The directory does not exist.")
            Return
        End If

        Dim files() As String = Directory.GetFiles(args(1))
        Parallel.For(0, files.Length,
                     Sub(index As Integer)
                         Dim fi As New FileInfo(files(index))
                         Dim size As Long = fi.Length
                         Interlocked.Add(totalSize, size)
                     End Sub)
        Console.WriteLine("Directory '{0}':", args(1))
        Console.WriteLine("{0:N0} files, {1:N0} bytes", files.Length, totalSize)
    End Sub
End Module
' The example displays output like the following:
'       Directory 'c:\windows\':
'       32 files, 6,587,222 bytes
```

矩阵和秒表示例

本示例使用 Parallel.For 方法来计算两个矩阵的乘积。它还演示如何使用 System.Diagnostics.Stopwatch 类来比较并行循环和非并行循环的性能。请注意，由于本示例可能会生成大量输出，因此它允许将输出重定向到一个文件。

```
using System;
using System.Diagnostics;
using System.Threading.Tasks;

class MultiplyMatrices
{
    #region Sequential_Loop
    static void MultiplyMatricesSequential(double[,] matA, double[,] matB,
                                           double[,] result)
    {
        int matACols = matA.GetLength(1);
        int matBCols = matB.GetLength(1);
        int matARows = matA.GetLength(0);

        for (int i = 0; i < matARows; i++)
        {
            for (int j = 0; j < matBCols; j++)
            {
                double temp = 0;
                for (int k = 0; k < matACols; k++)
                {
                   temp += matA[i, k] * matB[k, j];
                }
                result[i, j] += temp;
            }
```

```
        }
    }
    #endregion

    #region Parallel_Loop
    static void MultiplyMatricesParallel(double[,] matA, double[,] matB, double[,] result)
    {
        int matACols = matA.GetLength(1);
        int matBCols = matB.GetLength(1);
        int matARows = matA.GetLength(0);

        // A basic matrix multiplication.
        // Parallelize the outer loop to partition the source array by rows.
        Parallel.For(0, matARows, i =>
        {
            for (int j = 0; j < matBCols; j++)
            {
                double temp = 0;
                for (int k = 0; k < matACols; k++)
                {
                    temp += matA[i, k] * matB[k, j];
                }
                result[i, j] = temp;
            }
        }); // Parallel.For
    }
    #endregion

    #region Main
    static void Main(string[] args)
    {
        // Set up matrices. Use small values to better view
        // result matrix. Increase the counts to see greater
        // speedup in the parallel loop vs. the sequential loop.
        int colCount = 180;
        int rowCount = 2000;
        int colCount2 = 270;
        double[,] m1 = InitializeMatrix(rowCount, colCount);
        double[,] m2 = InitializeMatrix(colCount, colCount2);
        double[,] result = new double[rowCount, colCount2];

        // First do the sequential version.
        Console.Error.WriteLine("Executing sequential loop...");
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        MultiplyMatricesSequential(m1, m2, result);
        stopwatch.Stop();
        Console.Error.WriteLine("Sequential loop time in milliseconds: {0}",
                                stopwatch.ElapsedMilliseconds);

        // For the skeptics.
        OfferToPrint(rowCount, colCount2, result);

        // Reset timer and results matrix.
        stopwatch.Reset();
        result = new double[rowCount, colCount2];

        // Do the parallel loop.
        Console.Error.WriteLine("Executing parallel loop...");
        stopwatch.Start();
        MultiplyMatricesParallel(m1, m2, result);
        stopwatch.Stop();
        Console.Error.WriteLine("Parallel loop time in milliseconds: {0}",
                                stopwatch.ElapsedMilliseconds);
        OfferToPrint(rowCount, colCount2, result);

        // Keep the console window open in debug mode.
        Console.Error.WriteLine("Press any key to exit.");
```

```
        Console.ReadKey();
    }
    #endregion

    #region Helper_Methods
    static double[,] InitializeMatrix(int rows, int cols)
    {
        double[,] matrix = new double[rows, cols];

        Random r = new Random();
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                matrix[i, j] = r.Next(100);
            }
        }
        return matrix;
    }

    private static void OfferToPrint(int rowCount, int colCount, double[,] matrix)
    {
        Console.Error.Write("Computation complete. Print results (y/n)? ");
        char c = Console.ReadKey(true).KeyChar;
        Console.Error.WriteLine(c);
        if (Char.ToUpperInvariant(c) == 'Y')
        {
            if (! Console.IsOutputRedirected) Console.WindowWidth = 180;
            Console.WriteLine();
            for (int x = 0; x < rowCount; x++)
            {
                Console.WriteLine("ROW {0}: ", x);
                for (int y = 0; y < colCount; y++)
                {
                    Console.Write("{0:#.##} ", matrix[x, y]);
                }
                Console.WriteLine();
            }
        }
    }
    #endregion
}
```

```
Imports System.Diagnostics
Imports System.Threading.Tasks

Module MultiplyMatrices
#Region "Sequential_Loop"
    Sub MultiplyMatricesSequential(ByVal matA As Double(,), ByVal matB As Double(,), ByVal result As
Double(,))
        Dim matACols As Integer = matA.GetLength(1)
        Dim matBCols As Integer = matB.GetLength(1)
        Dim matARows As Integer = matA.GetLength(0)

        For i As Integer = 0 To matARows - 1
            For j As Integer = 0 To matBCols - 1
                Dim temp As Double = 0
                For k As Integer = 0 To matACols - 1
                    temp += matA(i, k) * matB(k, j)
                Next
                result(i, j) += temp
            Next
        Next
    End Sub
#End Region

#Region "Parallel_Loop"
```

```
    Private Sub MultiplyMatricesParallel(ByVal matA As Double(,), ByVal matB As Double(,), ByVal result As
Double(,))
        Dim matACols As Integer = matA.GetLength(1)
        Dim matBCols As Integer = matB.GetLength(1)
        Dim matARows As Integer = matA.GetLength(0)

        ' A basic matrix multiplication.
        ' Parallelize the outer loop to partition the source array by rows.
        Parallel.For(0, matARows, Sub(i)
                                      For j As Integer = 0 To matBCols - 1
                                          Dim temp As Double = 0
                                          For k As Integer = 0 To matACols - 1
                                              temp += matA(i, k) * matB(k, j)
                                          Next
                                          result(i, j) += temp
                                      Next
                                  End Sub)
    End Sub
#End Region

#Region "Main"
    Sub Main(ByVal args As String())
        ' Set up matrices. Use small values to better view
        ' result matrix. Increase the counts to see greater
        ' speedup in the parallel loop vs. the sequential loop.
        Dim colCount As Integer = 180
        Dim rowCount As Integer = 2000
        Dim colCount2 As Integer = 270
        Dim m1 As Double(,) = InitializeMatrix(rowCount, colCount)
        Dim m2 As Double(,) = InitializeMatrix(colCount, colCount2)
        Dim result As Double(,) = New Double(rowCount - 1, colCount2 - 1) {}

        ' First do the sequential version.
        Console.Error.WriteLine("Executing sequential loop...")
        Dim stopwatch As New Stopwatch()
        stopwatch.Start()

        MultiplyMatricesSequential(m1, m2, result)
        stopwatch.[Stop]()
        Console.Error.WriteLine("Sequential loop time in milliseconds: {0}", stopwatch.ElapsedMilliseconds)

        ' For the skeptics.
        OfferToPrint(rowCount, colCount2, result)

        ' Reset timer and results matrix.
        stopwatch.Reset()
        result = New Double(rowCount - 1, colCount2 - 1) {}

        ' Do the parallel loop.
        Console.Error.WriteLine("Executing parallel loop...")
        stopwatch.Start()
        MultiplyMatricesParallel(m1, m2, result)
        stopwatch.[Stop]()
        Console.Error.WriteLine("Parallel loop time in milliseconds: {0}", stopwatch.ElapsedMilliseconds)
        OfferToPrint(rowCount, colCount2, result)

        ' Keep the console window open in debug mode.
        Console.Error.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub
#End Region

#Region "Helper_Methods"
    Function InitializeMatrix(ByVal rows As Integer, ByVal cols As Integer) As Double(,)
        Dim matrix As Double(,) = New Double(rows - 1, cols - 1) {}

        Dim r As New Random()
        For i As Integer = 0 To rows - 1
            For j As Integer = 0 To cols - 1
```

```
                matrix(i, j) = r.[Next](100)
            Next
        Next
        Return matrix
    End Function

    Sub OfferToPrint(ByVal rowCount As Integer, ByVal colCount As Integer, ByVal matrix As Double(,))
        Console.Error.Write("Computation complete. Display results (y/n)? ")
        Dim c As Char = Console.ReadKey(True).KeyChar
        Console.Error.WriteLine(c)
        If Char.ToUpperInvariant(c) = "Y"c Then
            If Not Console.IsOutputRedirected Then Console.WindowWidth = 168
            Console.WriteLine()
            For x As Integer = 0 To rowCount - 1
                Console.WriteLine("ROW {0}: ", x)
                For y As Integer = 0 To colCount - 1
                    Console.Write("{0:#.##} ", matrix(x, y))
                Next
                Console.WriteLine()
            Next
        End If
    End Sub
#End Region
End Module
```

在对任何代码(包括循环)进行并行化时，一个重要的目标是利用尽可能多的处理器，而不会过度并行化到并行处理的开销使任何性能优势消耗殆尽的程度。在本特定示例中，只会对外部循环进行并行化，原因是不会在内部循环中执行太多工作。少量工作和不良缓存影响的组合可能会导致嵌套并行循环的性能降低。因此，仅并行化外部循环是在大多数系统上最大程度地发挥并发优势的最佳方式。

委托

For 的此重载的第三个参数是类型为 `Action<int>`(C# 中)或 `Action(Of Integer)`(Visual Basic 中)的委托。不管 `Action` 委托具有零个、一个或十六个类型参数，它都始终返回 void。在 Visual Basic 中，`Action` 的行为是用 `Sub` 定义的。示例使用 lambda 表达式来创建委托，但也可以用其他方式创建委托。有关详细信息，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

迭代值

委托采用其值为当前迭代的单一输入参数。此迭代值由运行时提供，并且其起始值为正在当前线程上处理的源的片段(分区)上第一个元素的索引。

如果需要更好地控制并发级别，请使用采用 System.Threading.Tasks.ParallelOptions 输入参数的重载之一，例如：Parallel.For(Int32, Int32, ParallelOptions, Action<Int32,ParallelLoopState>)。

返回值和异常处理

当所有线程均已完成时，For 会返回一个 System.Threading.Tasks.ParallelLoopResult 对象。当手动停止或中断循环迭代时，此返回值特别有用，因为 ParallelLoopResult 存储诸如完成运行的最后一个迭代等信息。如果某个线程上出现一个或多个异常，则将会引发 System.AggregateException。

在本示例的代码中，未使用 For 的返回值。

分析和性能

可以使用性能向导来查看计算机上的 CPU 使用情况。进行试验，增加矩阵中的列数和行数。矩阵越大，并行计算和顺序计算之间的性能差异就越大。当矩阵很小时，由于设置并行循环时会产生开销，因此顺序计算将运行更快。

同步调用共享资源(如控制台或文件系统)将大幅降低并行循环的性能。在衡量性能时，请尝试避免在循环内进行诸如 Console.WriteLine 等调用。

编译代码

将此代码复制并粘贴到 Visual Studio 项目中。

请参阅

- For
- ForEach
- 数据并行
- 并行编程

如何：编写简单的 Parallel.ForEach 循环

2021/11/16 •

此示例展示了如何使用 Parallel.ForEach 循环，对任何 System.Collections.IEnumerable 或 System.Collections.Generic.IEnumerable<T> 数据源启用数据并行。

> **NOTE**
> 本文档使用 lambda 表达式在 PLINQ 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

示例

此示例假定 C:\Users\Public\Pictures\Sample Pictures 文件夹中有几个 .jpg 文件，并创建名为“Modified”的新子文件夹。运行该示例时，它会旋转示例图片中的每个 .jpg 图像并将其保存到“Modified”文件夹 可以根据需要修改这两个路径。

```
using System;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;

namespace ParallelExample
{
    class Program
    {
        static void Main()
        {
            // 2 million
            var limit = 2_000_000;
            var numbers = Enumerable.Range(0, limit).ToList();

            var watch = Stopwatch.StartNew();
            var primeNumbersFromForeach = GetPrimeList(numbers);
            watch.Stop();

            var watchForParallel = Stopwatch.StartNew();
            var primeNumbersFromParallelForeach = GetPrimeListWithParallel(numbers);
            watchForParallel.Stop();

            Console.WriteLine($"Classical foreach loop | Total prime numbers :
{primeNumbersFromForeach.Count} | Time Taken : {watch.ElapsedMilliseconds} ms.");
            Console.WriteLine($"Parallel.ForEach loop  | Total prime numbers :
{primeNumbersFromParallelForeach.Count} | Time Taken : {watchForParallel.ElapsedMilliseconds} ms.");

            Console.WriteLine("Press any key to exit.");
            Console.ReadLine();
        }

        /// <summary>
        /// GetPrimeList returns Prime numbers by using sequential ForEach
        /// </summary>
        /// <param name="inputs"></param>
        /// <returns></returns>
        private static IList<int> GetPrimeList(IList<int> numbers) => numbers.Where(IsPrime).ToList();
```

```
        /// <summary>
        /// GetPrimeListWithParallel returns Prime numbers by using Parallel.ForEach
        /// </summary>
        /// <param name="numbers"></param>
        /// <returns></returns>
        private static IList<int> GetPrimeListWithParallel(IList<int> numbers)
        {
            var primeNumbers = new ConcurrentBag<int>();

            Parallel.ForEach(numbers, number =>
            {
                if (IsPrime(number))
                {
                    primeNumbers.Add(number);
                }
            });

            return primeNumbers.ToList();
        }

        /// <summary>
        /// IsPrime returns true if number is Prime, else false.(https://en.wikipedia.org/wiki/Prime_number)
        /// </summary>
        /// <param name="number"></param>
        /// <returns></returns>
        private static bool IsPrime(int number)
        {
            if (number < 2)
            {
                return false;
            }

            for (var divisor = 2; divisor <= Math.Sqrt(number); divisor++)
            {
                if (number % divisor == 0)
                {
                    return false;
                }
            }
            return true;
        }
    }
}
```

```
Imports System.Collections.Concurrent

Namespace ParallelExample
    Class Program
        Shared Sub Main()
            ' 2 million
            Dim limit = 2_000_000
            Dim numbers = Enumerable.Range(0, limit).ToList()

            Dim watch = Stopwatch.StartNew()
            Dim primeNumbersFromForeach = GetPrimeList(numbers)
            watch.Stop()

            Dim watchForParallel = Stopwatch.StartNew()
            Dim primeNumbersFromParallelForeach = GetPrimeListWithParallel(numbers)
            watchForParallel.Stop()

            Console.WriteLine($"Classical foreach loop | Total prime numbers :
{primeNumbersFromForeach.Count} | Time Taken : {watch.ElapsedMilliseconds} ms.")
            Console.WriteLine($"Parallel.ForEach loop  | Total prime numbers :
{primeNumbersFromParallelForeach.Count} | Time Taken : {watchForParallel.ElapsedMilliseconds} ms.")

            Console.WriteLine("Press any key to exit.")
            Console.ReadLine()
        End Sub

        ' GetPrimeList returns Prime numbers by using sequential ForEach
        Private Shared Function GetPrimeList(numbers As IList(Of Integer)) As IList(Of Integer)
            Return numbers.Where(AddressOf IsPrime).ToList()
        End Function

        ' GetPrimeListWithParallel returns Prime numbers by using Parallel.ForEach
        Private Shared Function GetPrimeListWithParallel(numbers As IList(Of Integer)) As IList(Of Integer)
            Dim primeNumbers = New ConcurrentBag(Of Integer)()
            Parallel.ForEach(numbers, Sub(number)

                                          If IsPrime(number) Then
                                              primeNumbers.Add(number)
                                          End If
                                      End Sub)
            Return primeNumbers.ToList()
        End Function

        ' IsPrime returns true if number is Prime, else false.(https://en.wikipedia.org/wiki/Prime_number)
        Private Shared Function IsPrime(number As Integer) As Boolean
            If number < 2 Then
                Return False
            End If

            For divisor = 2 To Math.Sqrt(number)

                If number Mod divisor = 0 Then
                    Return False
                End If
            Next

            Return True
        End Function
    End Class
End Namespace
```

Parallel.ForEach 循环的工作原理类似 Parallel.For 循环。该循环对源集合进行分区，并根据系统环境在多个线程上安排工作。系统上的处理器越多，并行方法的运行速度就越快。对于一些源集合，有序循环可能会更快，具体视源大小以及该循环要执行的工作类型而定。有关性能的详细信息，请参阅数据和任务并行的潜在问题。

若要详细了解并行循环，请参阅如何：编写简单的 Parallel.For 循环。

若要将 Parallel.ForEach 与非泛型集合结合使用，可以使用 Enumerable.Cast 扩展方法，将集合转换为泛型集合，如下面的示例所示：

```
Parallel.ForEach(nonGenericCollection.Cast<object>(),
    currentElement =>
    {
    });
```

```
Parallel.ForEach(nonGenericCollection.Cast(Of Object), _
                 Sub(currentElement)
                     ' ... work with currentElement
                 End Sub)
```

还可以使用并行 LINQ (PLINQ) 并行处理 IEnumerable<T> 数据源。借助 PLINQ，可以使用声明性查询语法来表达循环行为。有关详细信息，请参阅并行 LINQ (PLINQ)。

编译并运行代码

可以作为 .NET Framework 的控制台应用程序或 .NET Core 的控制台应用程序编译代码。

Visual Studio 中有适用于 Windows 桌面和 .NET Core 的 Visual Basic 和 C# 控制台应用程序模板。

从命令行，可使用 .NET Core CLI 命令（例如 `dotnet new console` 或 `dotnet new console -lang vb`），或者可创建文件并使用 .NET Framework 应用程序提供的命令行编译器。

对于.NET Core 项目，必须引用 System.Drawing.Common NuGet 包。在 Visual Studio 中，使用 NuGet 包管理器安装该包。或者，也可以在 *.csproj 或 *.vbproj 文件中添加对包的引用：

```
<ItemGroup>
     <PackageReference Include="System.Drawing.Common" Version="4.5.1" />
</ItemGroup>
```

要从命令行运行 .NET Core 控制台应用程序，请使用包含该应用程序的文件夹中的 `dotnet run`。

要从 Visual Studio 中运行控制台应用程序，请按 F5。

请参阅

- 数据并行
- 并行编程
- 并行 LINQ (PLINQ)

如何：编写具有线程局部变量的 Parallel.For 循环

2021/11/16 •

此示例演示如何使用线程本地变量来存储和检索由 For 循环创建的每个单独任务中的状态。通过使用线程本地数据，你可以避免将大量的访问同步为共享状态的开销。在任务的所有迭代完成之前，你将计算和存储值，而不是写入每个迭代上的共享资源。然后，你可以将最终结果一次性写入共享资源，或将其传递到另一个方法。

示例

以下示例调用 For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) 方法以计算在包含一百万个元素的数组中值的总和。每个元素的值等于其索引。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Test
{
    static void Main()
    {
        int[] nums = Enumerable.Range(0, 1000000).ToArray();
        long total = 0;

        // Use type parameter to make subtotal a long, not an int
        Parallel.For<long>(0, nums.Length, () => 0, (j, loop, subtotal) =>
        {
            subtotal += nums[j];
            return subtotal;
        },
            (x) => Interlocked.Add(ref total, x)
        );

        Console.WriteLine("The total is {0:N0}", total);
        Console.WriteLine("Press any key to exit");
        Console.ReadKey();
    }
}
```

```
'How to: Write a Parallel.For Loop That Has Thread-Local Variables

Imports System.Threading
Imports System.Threading.Tasks

Module ForWithThreadLocal

    Sub Main()
        Dim nums As Integer() = Enumerable.Range(0, 1000000).ToArray()
        Dim total As Long = 0

        ' Use type parameter to make subtotal a Long type. Function will overflow otherwise.
        Parallel.For(Of Long)(0, nums.Length, Function() 0, Function(j, [loop], subtotal)
                                                                subtotal += nums(j)
                                                                Return subtotal
                                                            End Function, Function(x) Interlocked.Add(total,
x))

        Console.WriteLine("The total is {0:N0}", total)
        Console.WriteLine("Press any key to exit")
        Console.ReadKey()
    End Sub

End Module
```

每个 For 方法的前两个参数都指定起始迭代值和结束迭代值。在方法的此重载中，第三个参数是在其中初始化本地状态的参数。在此上下文中，"本地状态"是指其生存期恰好从当前线程上循环的第一个迭代之前延伸至最后一个迭代之后的变量。

第三个参数的类型为 Func<TResult>，其中 `TResult` 是将存储线程本地状态的变量的类型。其类型由在调用泛型 For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) 方法时提供的泛型类型参数定义，在此情况下为 Int64。该类型参数告诉编译器将用于存储线程本地状态的临时变量的类型。在此示例中，表达式 `() => 0`（在 Visual Basic 中为 `Function() 0`）将线程本地变量初始化为零。如果泛型类型参数是引用类型或用户定义的值类型，表达式将如下所示：

```
() => new MyClass()
```

```
Function() new MyClass()
```

第四个参数定义循环逻辑。它必须是一个委托或 lambda 表达式，其签名在 C# 中为 `Func<int, ParallelLoopState, long, long>`，在 Visual Basic 中为 `Func(Of Integer, ParallelLoopState, Long, Long)`。第一个参数是针对循环的该次迭代的循环计数器的值。第二个参数是可用于中断循环的 ParallelLoopState 对象；此对象由 Parallel 类提供给循环的每个匹配项。第三个参数是线程本地变量。最后一个参数是返回类型。在此情况下，该类型为 Int64，因为那是我们在 For 类型参数中指定的类型。该变量名为 `subtotal` 并且由 lambda 表达式返回。返回值用于在循环的每个后续迭代上初始化 `subtotal`。你也可以将最后一个参数看作传递到每个迭代，然后在最后一个迭代完成时传递到 `localFinally` 委托的值。

第五个参数定义在特定线程上的所有迭代都完成后，将调用一次的方法。输入参数的类型同样也对应于 For<TLocal>(Int32, Int32, Func<TLocal>, Func<Int32,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) 方法的类型参数，以及主体 lambda 表达式返回的类型。在此示例中，通过调用 Interlocked.Add 方法，采用线程安全的方式在类范围将值添加到变量。通过使用线程本地变量，我们避免了在循环的每个迭代上写入此类变量。

若要详细了解如何使用 Lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

请参阅

- 数据并行
- 并行编程
- 任务并行库 (TPL)
- PLINQ 和 TPL 中的 Lambda 表达式

如何：使用分区本地变量编写 Parallel.ForEach 循环

2021/11/16 •

下面的示例演示如何编写使用分区本地变量的 ForEach 方法。当 ForEach 循环执行时，它会将其源集合划分为多个分区。每个分区都有自己的分区本地变量的副本。分区本地变量类似于线程本地变量，只是单个线程上可以运行多个分区。

此示例中的代码和参数非常类似于对应的 For 方法。有关详细信息，请参阅如何：编写具有线程局部变量的 Parallel.For 循环。

若要在 ForEach 循环中使用分区本地变量，必须调用采用两个类型参数的其中一个方法重载。第一个类型参数 `TSource` 指定源元素的类型，第二个类型参数 `TLocal` 指定分区本地变量的类型。

示例

以下示例调用 Parallel.ForEach<TSource,TLocal>(IEnumerable<TSource>, Func<TLocal>, Func<TSource,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) 重载以计算一百万个元素的数组的总和。此重载具有四个参数：

- `source`，也就是数据源。它必须实现 IEnumerable<T>。在我们的示例中，数据源是由 `IEnumerable<Int32>` 方法返回的一百万个成员 Enumerable.Range 对象。
- `localInit`，或初始化分区本地变量的函数。为每个在其中执行 Parallel.ForEach 操作的分区调用此函数一次。我们的示例将分区本地变量初始化为零。
- `body`，由并行循环对循环的每个迭代调用的 Func<T1,T2,T3,TResult>。其签名为 `Func\<TSource, ParallelLoopState, TLocal, TLocal>`。你为委托提供代码，并且循环将传入输入参数，它们是：
  - IEnumerable<T> 的当前元素。
  - 你可以在委托的代码中用来检查循环状态的 ParallelLoopState 变量。
  - 分区本地变量。

  你的委托返回分区本地变量，然后将此变量传递到在该特定分区中执行的循环的下一次迭代。每个循环分区维护此变量的一个单独实例。

  在该示例中，委托将每个整数的值添加到分区本地变量，该变量维护该分区中整数元素值的不断变化着的总数。
- `localFinally`，当在每个分区中的循环操作完成时，`Action<TLocal>` 调用的 Parallel.ForEach 委托。Parallel.ForEach 方法将此循环分区的分区本地变量的最终值传递给 `Action<TLocal>` 委托，并且你提供代码，以执行合并来自此分区的结果与其他分区的结果所需的操作。此委托可以由多个任务并行调用。因此，该示例使用 Interlocked.Add(Int32, Int32) 方法以同步对 `total` 变量的访问。由于委托类型为 Action<T>，因此不存在返回值。

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Test
{
    static void Main()
    {
        int[] nums = Enumerable.Range(0, 1000000).ToArray();
        long total = 0;

        // First type parameter is the type of the source elements
        // Second type parameter is the type of the thread-local variable (partition subtotal)
        Parallel.ForEach<int, long>(nums, // source collection
                                    () => 0, // method to initialize the local variable
                                    (j, loop, subtotal) => // method invoked by the loop on each iteration
                                    {
                                        subtotal += j; //modify local variable
                                        return subtotal; // value to be passed to next iteration
                                    },
            // Method to be executed when each partition has completed.
            // finalResult is the final value of subtotal for a particular partition.
                                    (finalResult) => Interlocked.Add(ref total, finalResult)
                                    );

        Console.WriteLine("The total from Parallel.ForEach is {0:N0}", total);
    }
}
// The example displays the following output:
//        The total from Parallel.ForEach is 499,999,500,000
```

```
' How to: Write a Parallel.ForEach Loop That Has Thread-Local Variables

Imports System.Threading
Imports System.Threading.Tasks

Module ForEachThreadLocal
    Sub Main()

        Dim nums() As Integer = Enumerable.Range(0, 1000000).ToArray()
        Dim total As Long = 0

        ' First type parammemter is the type of the source elements
        ' Second type parameter is the type of the thread-local variable (partition subtotal)
        Parallel.ForEach(Of Integer, Long)(nums, Function() 0,
                                           Function(elem, loopState, subtotal)
                                               subtotal += elem
                                               Return subtotal
                                           End Function,
                                            Sub(finalResult)
                                                Interlocked.Add(total, finalResult)
                                            End Sub)

        Console.WriteLine("The result of Parallel.ForEach is {0:N0}", total)
    End Sub
End Module
' The example displays the following output:
'       The result of Parallel.ForEach is 499,999,500,000
```

请参阅

- 数据并行

- 如何：编写具有线程局部变量的 Parallel.For 循环
- PLINQ 和 TPL 中的 Lambda 表达式

如何：取消 Parallel.For 或 ForEach Loop

2021/11/16 •

Parallel.For 和 Parallel.ForEach 方法支持通过使用取消令牌进行取消。若要详细了解取消的大致信息，请参阅取消。在并行循环中，将 CancellationToken 提供给 ParallelOptions 参数中的方法，再将并行调用封闭到 try-catch 块中。

示例

下面的示例展示了如何取消调用 Parallel.ForEach。可以采用相同的方法来取消调用 Parallel.For。

```
namespace CancelParallelLoops
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;

    class Program
    {
        static void Main()
        {
            int[] nums = Enumerable.Range(0, 10000000).ToArray();
            CancellationTokenSource cts = new CancellationTokenSource();

           // Use ParallelOptions instance to store the CancellationToken
            ParallelOptions po = new ParallelOptions();
            po.CancellationToken = cts.Token;
            po.MaxDegreeOfParallelism = System.Environment.ProcessorCount;
            Console.WriteLine("Press any key to start. Press 'c' to cancel.");
            Console.ReadKey();

            // Run a task so that we can cancel from another thread.
            Task.Factory.StartNew(() =>
            {
                if (Console.ReadKey().KeyChar == 'c')
                    cts.Cancel();
                Console.WriteLine("press any key to exit");
            });

            try
            {
                Parallel.ForEach(nums, po, (num) =>
                {
                    double d = Math.Sqrt(num);
                    Console.WriteLine("{0} on {1}", d, Thread.CurrentThread.ManagedThreadId);
                    po.CancellationToken.ThrowIfCancellationRequested();
                });
            }
            catch (OperationCanceledException e)
            {
                Console.WriteLine(e.Message);
            }
            finally
            {
                cts.Dispose();
            }

            Console.ReadKey();
        }
    }
}
```

```
' How to: Cancel a Parallel.For or ForEach Loop

Imports System.Threading
Imports System.Threading.Tasks

Module CancelParallelLoops
    Sub Main()
        Dim nums() As Integer = Enumerable.Range(0, 10000000).ToArray()
        Dim cts As New CancellationTokenSource

        ' Use ParallelOptions instance to store the CancellationToken
        Dim po As New ParallelOptions
        po.CancellationToken = cts.Token
        po.MaxDegreeOfParallelism = System.Environment.ProcessorCount
        Console.WriteLine("Press any key to start. Press 'c' to cancel.")
        Console.ReadKey()

        ' Run a task so that we can cancel from another thread.
        Dim t As Task = Task.Factory.StartNew(Sub()
                                                  If Console.ReadKey().KeyChar = "c"c Then
                                                      cts.Cancel()
                                                  End If
                                                  Console.WriteLine(vbCrLf & "Press any key to exit.")
                                              End Sub)

        Try

            ' The error "Exception is unhandled by user code" will appear if "Just My Code"
            ' is enabled. This error is benign. You can press F5 to continue, or disable Just My Code.
            Parallel.ForEach(nums, po, Sub(num)
                                           Dim d As Double = Math.Sqrt(num)
                                           Console.CursorLeft = 0
                                           Console.Write("{0:##.##} on {1}", d,
Thread.CurrentThread.ManagedThreadId)
                                           po.CancellationToken.ThrowIfCancellationRequested()
                                       End Sub)

        Catch e As OperationCanceledException
            Console.WriteLine(e.Message)
        Finally
            cts.Dispose()
        End Try

        Console.ReadKey()

    End Sub
End Module
```

如果发送取消信号的令牌与ParallelOptions 实例中指定的令牌相同，并行循环会在取消时抛出一个 OperationCanceledException。如果导致取消发生的是其他一些令牌，循环会抛出带 OperationCanceledException 的 AggregateException 作为 InnerException。

请参阅

- 数据并行
- PLINQ 和 TPL 中的 Lambda 表达式

如何：处理并行循环中的异常

2021/11/16 •

Parallel.For 和 Parallel.ForEach 重载没有任何用于处理可能引发的异常的特殊机制。在这一方面，它们类似于常规 `for` 和 `foreach` 循环（在 Visual Basic 中为 `For` 和 `For Each`）；未处理的异常会导致循环在当前运行的迭代完成后立即终止。

向并行循环添加自己的异常处理逻辑时，将处理类似于在多个线程上同时引发相似异常的情况，以及一个线程上引发异常导致另一个线程上引发另一个异常的情况。你可以通过将循环中的所有异常包装到一个 System.AggregateException 中处理这两种情况。下面的示例演示了一种可能的方法。

> **NOTE**
>
> 某些情况下，当启用“仅我的代码”后，Visual Studio 会在引发异常的行中断运行并显示一条错误消息，该消息显示“用户代码未处理异常”。此错误是良性的。可以按 F5 继续运行，并请参阅下面示例中所示的异常处理行为。为了阻止 Visual Studio 在第一个错误出现时中断，只需依次转到“工具”、“选项”、“调试”、“常规”下，取消选中“仅我的代码”复选框即可。

示例

在此示例中，所有异常都被捕获，并包装到引发的 System.AggregateException 中。调用方可以决定要处理哪些异常。

```
class ExceptionDemo2
{
    static void Main(string[] args)
    {
        // Create some random data to process in parallel.
        // There is a good probability this data will cause some exceptions to be thrown.
        byte[] data = new byte[5000];
        Random r = new Random();
        r.NextBytes(data);

        try
        {
            ProcessDataInParallel(data);
        }
        catch (AggregateException ae)
        {
            var ignoredExceptions = new List<Exception>();
            // This is where you can choose which exceptions to handle.
            foreach (var ex in ae.Flatten().InnerExceptions)
            {
                if (ex is ArgumentException)
                    Console.WriteLine(ex.Message);
                else
                    ignoredExceptions.Add(ex);
            }
            if (ignoredExceptions.Count > 0) throw new AggregateException(ignoredExceptions);
        }

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    private static void ProcessDataInParallel(byte[] data)
    {
        // Use ConcurrentQueue to enable safe enqueueing from multiple threads.
        var exceptions = new ConcurrentQueue<Exception>();

        // Execute the complete loop and capture all exceptions.
        Parallel.ForEach(data, d =>
        {
            try
            {
                // Cause a few exceptions, but not too many.
                if (d < 3)
                    throw new ArgumentException($"Value is {d}. Value must be greater than or equal to 3.");
                else
                    Console.Write(d + " ");
            }
            // Store the exception and continue with the loop.
            catch (Exception e)
            {
                exceptions.Enqueue(e);
            }
        });
        Console.WriteLine();

        // Throw the exceptions here after the loop completes.
        if (exceptions.Count > 0) throw new AggregateException(exceptions);
    }
}
```

```
' How to: Handle Exceptions in Parallel Loops

Imports System.Collections.Concurrent
Imports System.Collections.Generic
Imports System.Threading.Tasks

Module ExceptionsInLoops

    Sub Main()

        ' Create some random data to process in parallel.
        ' There is a good probability this data will cause some exceptions to be thrown.
        Dim data(1000) As Byte
        Dim r As New Random()
        r.NextBytes(data)

        Try
            ProcessDataInParallel(data)
        Catch ae As AggregateException
            Dim ignoredExceptions As New List(Of Exception)
            ' This is where you can choose which exceptions to handle.
            For Each ex As Exception In ae.Flatten().InnerExceptions
                If (TypeOf (ex) Is ArgumentException) Then
                    Console.WriteLine(ex.Message)
                Else
                    ignoredExceptions.Add(ex)
                End If
            Next
            If ignoredExceptions.Count > 0 Then
                Throw New AggregateException(ignoredExceptions)
            End If
        End Try
        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub
    Sub ProcessDataInParallel(ByVal data As Byte())

        ' Use ConcurrentQueue to enable safe enqueueing from multiple threads.
        Dim exceptions As New ConcurrentQueue(Of Exception)

        ' Execute the complete loop and capture all exceptions.
        Parallel.ForEach(Of Byte)(data, Sub(d)
                                            Try
                                                ' Cause a few exceptions, but not too many.
                                                If d < 3 Then
                                                    Throw New ArgumentException($"Value is {d}. Value must
be greater than or equal to 3")
                                                Else
                                                    Console.Write(d & " ")
                                                End If
                                            Catch ex As Exception
                                                ' Store the exception and continue with the loop.
                                                exceptions.Enqueue(ex)
                                            End Try
                                        End Sub)
        Console.WriteLine()
        ' Throw the exceptions here after the loop completes.
        If exceptions.Count > 0 Then
            Throw New AggregateException(exceptions)
        End If
    End Sub
End Module
```

请参阅

- 数据并行
- PLINQ 和 TPL 中的 Lambda 表达式

如何：加快小型循环体的速度

2021/11/16 •

如果 Parallel.For 循环的主体很小，它的执行速度可能慢于相当的顺序循环，如 C# 中的 for 循环和 Visual Basic 中的 For 循环。性能下降是由数据分区中的开销和在每个循环迭代上调用委托的成本所引起的。若要解决这种情况下，Partitioner 类提供了 Partitioner.Create 方法，使你能够为委托主体提供一个顺序循环，以便每个分区仅调用一次委托，而不是每个迭代调用一次委托。有关详细信息，请参阅 PLINQ 和 TPL 的自定义分区程序。

示例

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {

        // Source must be array or IList.
        var source = Enumerable.Range(0, 100000).ToArray();

        // Partition the entire source array.
        var rangePartitioner = Partitioner.Create(0, source.Length);

        double[] results = new double[source.Length];

        // Loop over the partitions in parallel.
        Parallel.ForEach(rangePartitioner, (range, loopState) =>
        {
            // Loop over each range element without a delegate invocation.
            for (int i = range.Item1; i < range.Item2; i++)
            {
                results[i] = source[i] * Math.PI;
            }
        });

        Console.WriteLine("Operation complete. Print results? y/n");
        char input = Console.ReadKey().KeyChar;
        if (input == 'y' || input == 'Y')
        {
            foreach(double d in results)
            {
                Console.Write("{0} ", d);
            }
        }
    }
}
```

```
Imports System.Threading.Tasks
Imports System.Collections.Concurrent

Module PartitionDemo

    Sub Main()
        ' Source must be array or IList.
        Dim source = Enumerable.Range(0, 100000).ToArray()

        ' Partition the entire source array.
        ' Let the partitioner size the ranges.
        Dim rangePartitioner = Partitioner.Create(0, source.Length)

        Dim results(source.Length - 1) As Double

        ' Loop over the partitions in parallel. The Sub is invoked
        ' once per partition.
        Parallel.ForEach(rangePartitioner, Sub(range, loopState)

                                               ' Loop over each range element without a delegate invocation.
                                               For i As Integer = range.Item1 To range.Item2 - 1
                                                   results(i) = source(i) * Math.PI
                                               Next
                                           End Sub)
        Console.WriteLine("Operation complete. Print results? y/n")
        Dim input As Char = Console.ReadKey().KeyChar
        If input = "y"c Or input = "Y"c Then
            For Each d As Double In results
                Console.Write("{0} ", d)
            Next
        End If

    End Sub
End Module
```

在循环执行最少量的工作时，此示例中演示的方法很有用。随着工作变得更占用计算资源，通过默认分区程序，使用 For 或 ForEach 循环，很有可能会获得相同或更好的性能。

另请参阅

- 数据并行
- PLINQ 和 TPL 的自定义分区程序
- 迭代器 (C#)
- 迭代器 (Visual Basic)
- PLINQ 和 TPL 中的 Lambda 表达式

如何：使用并行类循环访问文件目录

2021/11/16 •

在许多情况下，文件迭代是可以轻松并行执行的操作。主题如何：使用 PLINQ 循环访问文件目录介绍了如何在许多情况下以最简单的方式执行此任务。不过，如果代码必须处理访问文件系统时可能会出现的多种异常，可能会带来麻烦。下面的示例展示了一种解决此问题的方法。它使用基于堆栈的迭代遍历指定目录下的所有文件和文件夹，并让代码能够捕获和处理各种异常。当然，如何处理异常还是取决于自己的选择。

示例

下面的示例按顺序循环访问目录，但会并行处理文件。这可能是文件与目录比很大时的最佳方法。也可以并行执行目录迭代，并顺序访问每个文件。并行执行两个循环的效率可能并不高，除非专门定目标到有大量处理器的计算机。不过，与所有情况一样，应彻底测试应用，以确定最佳方法。

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Security;
using System.Threading;
using System.Threading.Tasks;

class Program
{
   static void Main()
   {
      try {
         TraverseTreeParallelForEach(@"C:\Program Files", (f) =>
         {
            // Exceptions are no-ops.
            try {
               // Do nothing with the data except read it.
               byte[] data = File.ReadAllBytes(f);
            }
            catch (FileNotFoundException) {}
            catch (IOException) {}
            catch (UnauthorizedAccessException) {}
            catch (SecurityException) {}
            // Display the filename.
            Console.WriteLine(f);
         });
      }
      catch (ArgumentException) {
         Console.WriteLine(@"The directory 'C:\Program Files' does not exist.");
      }

      // Keep the console window open.
      Console.ReadKey();
   }

   public static void TraverseTreeParallelForEach(string root, Action<string> action)
   {
      //Count of files traversed and timer for diagnostic output
      int fileCount = 0;
      var sw = Stopwatch.StartNew();

      // Determine whether to parallelize file processing on each folder based on processor count.
      int procCount = System.Environment.ProcessorCount;

      // Data structure to hold names of subfolders to be examined for files.
```

```
// Data structure to hold names of subfolders to be examined for files.
Stack<string> dirs = new Stack<string>();

if (!Directory.Exists(root)) {
      throw new ArgumentException();
}
dirs.Push(root);

while (dirs.Count > 0) {
   string currentDir = dirs.Pop();
   string[] subDirs = {};
   string[] files = {};

   try {
      subDirs = Directory.GetDirectories(currentDir);
   }
   // Thrown if we do not have discovery permission on the directory.
   catch (UnauthorizedAccessException e) {
      Console.WriteLine(e.Message);
      continue;
   }
   // Thrown if another process has deleted the directory after we retrieved its name.
   catch (DirectoryNotFoundException e) {
      Console.WriteLine(e.Message);
      continue;
   }

   try {
      files = Directory.GetFiles(currentDir);
   }
   catch (UnauthorizedAccessException e) {
      Console.WriteLine(e.Message);
      continue;
   }
   catch (DirectoryNotFoundException e) {
      Console.WriteLine(e.Message);
      continue;
   }
   catch (IOException e) {
      Console.WriteLine(e.Message);
      continue;
   }

   // Execute in parallel if there are enough files in the directory.
   // Otherwise, execute sequentially.Files are opened and processed
   // synchronously but this could be modified to perform async I/O.
   try {
      if (files.Length < procCount) {
         foreach (var file in files) {
            action(file);
            fileCount++;
         }
      }
      else {
         Parallel.ForEach(files, () => 0, (file, loopState, localCount) =>
                                       { action(file);
                                         return (int) ++localCount;
                                       },
                          (c) => {
                                    Interlocked.Add(ref fileCount, c);
                          });
      }
   }
   catch (AggregateException ae) {
      ae.Handle((ex) => {
                  if (ex is UnauthorizedAccessException) {
                     // Here we just output a message and go on.
                     Console.WriteLine(ex.Message);
                     return true;
```

```
                        }
                        // Handle other exceptions here if necessary...

                        return false;
            });
         }

         // Push the subdirectories onto the stack for traversal.
         // This could also be done before handing the files.
         foreach (string str in subDirs)
            dirs.Push(str);
      }

      // For diagnostic purposes.
      Console.WriteLine("Processed {0} files in {1} milliseconds", fileCount, sw.ElapsedMilliseconds);
   }
}
```

```
Imports System.Collections.Generic
Imports System.Diagnostics
Imports System.IO
Imports System.Security
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Sub Main()
        Try
            TraverseTreeParallelForEach("C:\Program Files",
                                        Sub(f)
                                         ' Exceptions are No-ops.
                                         Try
                                             ' Do nothing with the data except read it.
                                            Dim data() As Byte = File.ReadAllBytes(f)
                                             ' In the event the file has been deleted.
                                         Catch e As FileNotFoundException

                                             ' General I/O exception, especially if the file is in use.
                                         Catch e As IOException

                                             ' Lack of adequate permissions.
                                         Catch e As UnauthorizedAccessException

                                             ' Lack of adequate permissions.
                                         Catch e As SecurityException

                                            End Try
                                         ' Display the filename.
                                         Console.WriteLine(f)
                                        End Sub)
        Catch e As ArgumentException
            Console.WriteLine("The directory 'C:\Program Files' does not exist.")
        End Try
        ' Keep the console window open.
        Console.ReadKey()
    End Sub

    Public Sub TraverseTreeParallelForEach(ByVal root As String, ByVal action As Action(Of String))
        'Count of files traversed and timer for diagnostic output
        Dim fileCount As Integer = 0
        Dim sw As Stopwatch = Stopwatch.StartNew()

        ' Determine whether to parallelize file processing on each folder based on processor count.
        Dim procCount As Integer = System.Environment.ProcessorCount

        ' Data structure to hold names of subfolders to be examined for files.
        Dim dirs As New Stack(Of String)
```

```
        If Not Directory.Exists(root) Then Throw New ArgumentException()

        dirs.Push(root)

        While (dirs.Count > 0)
            Dim currentDir As String = dirs.Pop()
            Dim subDirs() As String = Nothing
            Dim files() As String = Nothing

            Try
                subDirs = Directory.GetDirectories(currentDir)
                ' Thrown if we do not have discovery permission on the directory.
            Catch e As UnauthorizedAccessException
                Console.WriteLine(e.Message)
                Continue While
                ' Thrown if another process has deleted the directory after we retrieved its name.
            Catch e As DirectoryNotFoundException
                Console.WriteLine(e.Message)
                Continue While
            End Try

            Try
                files = Directory.GetFiles(currentDir)
            Catch e As UnauthorizedAccessException
                Console.WriteLine(e.Message)
                Continue While
            Catch e As DirectoryNotFoundException
                Console.WriteLine(e.Message)
                Continue While
            Catch e As IOException
                Console.WriteLine(e.Message)
                Continue While
            End Try

            ' Execute in parallel if there are enough files in the directory.
            ' Otherwise, execute sequentially.Files are opened and processed
            ' synchronously but this could be modified to perform async I/O.
            Try
                If files.Length < procCount Then
                    For Each file In files
                        action(file)
                        fileCount += 1
                    Next
                Else
                    Parallel.ForEach(files, Function() 0, Function(file, loopState, localCount)
                                                              action(file)
                                                              localCount = localCount + 1
                                                              Return localCount
                                                          End Function,
                                     Sub(c)
                                         Interlocked.Add(fileCount, c)
                                     End Sub)
                End If
            Catch ae As AggregateException
                ae.Handle(Function(ex)

                              If TypeOf (ex) Is UnauthorizedAccessException Then

                                  ' Here we just output a message and go on.
                                  Console.WriteLine(ex.Message)
                                  Return True
                              End If
                              ' Handle other exceptions here if necessary...

                              Return False
                          End Function)
            End Try
            ' Push the subdirectories onto the stack for traversal.
            ' This could be done before handing the files
```

```
            ' This could also be done before handing the files.
            For Each str As String In subDirs
                dirs.Push(str)
            Next

            ' For diagnostic purposes.
            Console.WriteLine("Processed {0} files in {1} milliseconds", fileCount, sw.ElapsedMilliseconds)
        End While
    End Sub
End Module
```

在此示例中，文件 I/O 是同步执行。若要处理大文件或网络连接速度慢，最好异步访问文件。可以将异步 I/O 技术与并行迭代结合使用。有关详细信息，请参阅 TPL 和传统 .NET 异步编程。

此示例使用局部 `fileCount` 变量维护已处理的总文件数的计数。由于多个任务可能并发访问此变量，可以调用 Interlocked.Add 方法来同步对它的访问。

请注意，如果主线程抛出异常，ForEach 方法启动的线程可能会继续运行。若要停止这些线程，可以在异常处理程序中设置布尔变量，并在并行循环每次迭代时检查此变量的值。如果此值指明异常已抛出，请使用 ParallelLoopState 变量停止或中断循环。有关详细信息，请参阅如何：停止或中断 Parallel.For 循环。

另请参阅

- 数据并行

基于任务的异步编程

2021/11/16 •

任务并行库 (TPL) 以“任务”的概念为基础，后者表示异步操作。在某些方面，任务类似于线程或 ThreadPool 工作项，但是抽象级别更高。术语“任务并行”是指一个或多个独立的任务同时运行。任务提供两个主要好处：

- 系统资源的使用效率更高，可伸缩性更好。

  在后台，任务排队到已使用算法增强的 ThreadPool，这些算法能够确定线程数并随之调整，提供负载平衡以实现吞吐量最大化。这会使任务相对轻量，你可以创建很多任务以启用细化并行。

- 对于线程或工作项，可以使用更多的编程控件。

  任务和围绕它们生成的框架提供了一组丰富的 API，这些 API 支持等待、取消、继续、可靠的异常处理、详细状态、自定义计划等功能。

出于这两个原因，在 .NET 中，TPL 是用于编写多线程、异步和并行代码的首选 API。

隐式创建和运行任务

Parallel.Invoke 方法提供了一种简便方式，可同时运行任意数量的任意语句。只需为每个工作项传入 Action 委托即可。创建这些委托的最简单方式是使用 lambda 表达式。lambda 表达式可调用指定的方法，或提供内联代码。下面的示例演示一个基本的 Invoke 调用，该调用创建并启动同时运行的两个任务。第一个任务由调用名为 `DoSomeWork` 的方法的 lambda 表达式表示，第二个任务由调用名为 `DoSomeOtherWork` 的方法的 lambda 表达式表示。

> **NOTE**
> 本文档使用 lambda 表达式在 TPL 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

```
Parallel.Invoke(() => DoSomeWork(), () => DoSomeOtherWork());
```

```
Parallel.Invoke(Sub() DoSomeWork(), Sub() DoSomeOtherWork())
```

> **NOTE**
> Task 在后台创建的 Invoke 实例数不一定与所提供的委托数相等。TPL 可能会使用各种优化，特别是对于大量的委托。

有关详细信息，请参阅如何：使用 Parallel.Invoke 来执行并行操作。

为了更好地控制任务执行或从任务返回值，必须更加显式地使用 Task 对象。

显式创建和运行任务

不返回值的任务由 System.Threading.Tasks.Task 类表示。返回值的任务由 System.Threading.Tasks.Task<TResult> 类表示，该类从 Task 继承。任务对象处理基础结构详细信息，并提供可在任务的整个生存期内从调用线程访问的方法和属性。例如，可以随时访问任务的 Status 属性，以确定它是已开始运行、已完成运行、已取消还是引发了异常。状态由 TaskStatus 枚举表示。

在创建任务时，你赋予它一个用户委托，该委托封装该任务将执行的代码。该委托可以表示为命名的委托、匿名方法或 lambda 表达式。lambda 表达式可以包含对命名方法的调用，如下面的示例所示。请注意，该示例包含对 Task.Wait 方法的调用，以确保任务在控制台模式应用程序结束之前完成执行。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.Name = "Main";

      // Create a task and supply a user delegate by using a lambda expression.
      Task taskA = new Task( () => Console.WriteLine("Hello from taskA."));
      // Start the task.
      taskA.Start();

      // Output a message from the calling thread.
      Console.WriteLine("Hello from thread '{0}'.",
                        Thread.CurrentThread.Name);
      taskA.Wait();
   }
}
// The example displays output like the following:
//       Hello from thread 'Main'.
//       Hello from taskA.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        ' Create a task and supply a user delegate by using a lambda expression.
        Dim taskA = New Task(Sub() Console.WriteLine("Hello from taskA."))
        ' Start the task.
        taskA.Start()

        ' Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name)
        taskA.Wait()
    End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

你还可以使用 Task.Run 方法通过一个操作创建并启动任务。无论是哪个任务计划程序与当前线程关联，Run 方法都将使用默认的任务计划程序来管理任务。不需要对任务的创建和计划进行更多控制时，首选 Run 方法创建并启动任务。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      Thread.CurrentThread.Name = "Main";

      // Define and run the task.
      Task taskA = Task.Run( () => Console.WriteLine("Hello from taskA."));

      // Output a message from the calling thread.
      Console.WriteLine("Hello from thread '{0}'.",
                        Thread.CurrentThread.Name);
      taskA.Wait();
   }
}
// The example displays output like the following:
//       Hello from thread 'Main'.
//       Hello from taskA.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Thread.CurrentThread.Name = "Main"

        Dim taskA As Task = Task.Run(Sub() Console.WriteLine("Hello from taskA."))

        ' Output a message from the calling thread.
        Console.WriteLine("Hello from thread '{0}'.",
                          Thread.CurrentThread.Name)
        taskA.Wait()
    End Sub
End Module
' The example displays output like the following:
'    Hello from thread 'Main'.
'    Hello from taskA.
```

你还可以使用 TaskFactory.StartNew 方法在一个操作中创建并启动任务。不必将创建和计划分开并且需要其他任务创建选项或使用特定计划程序时，或者需要将其他状态传递到可以通过 Task.AsyncState 属性检索到的任务时，请使用此方法，如下例所示。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                               },
                                               new CustomData() {Name = i, CreationTime =
DateTime.Now.Ticks} );
      }
      Task.WaitAll(taskArray);
      foreach (var task in taskArray) {
         var data = task.AsyncState as CustomData;
         if (data != null)
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum);
      }
   }
}
// The example displays output like the following:
//       Task #0 created at 635116412924597583 on thread #3.
//       Task #1 created at 635116412924607584 on thread #4.
//       Task #3 created at 635116412924607584 on thread #4.
//       Task #4 created at 635116412924607584 on thread #4.
//       Task #2 created at 635116412924607584 on thread #3.
//       Task #6 created at 635116412924607584 on thread #3.
//       Task #5 created at 635116412924607584 on thread #4.
//       Task #8 created at 635116412924607584 on thread #4.
//       Task #7 created at 635116412924607584 on thread #3.
//       Task #9 created at 635116412924607584 on thread #4.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
    Public CreationTime As Long
    Public Name As Integer
    Public ThreadNum As Integer
End Class

Module Example
    Public Sub Main()
        Dim taskArray(9) As Task
        For i As Integer = 0 To taskArray.Length - 1
            taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                     Dim data As CustomData = TryCast(obj, CustomData)
                                                     If data Is Nothing Then Return

                                                     data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                                 End Sub,
                                                 New CustomData With {.Name = i, .CreationTime =
DateTime.Now.Ticks})
        Next
        Task.WaitAll(taskArray)

        For Each task In taskArray
            Dim data = TryCast(task.AsyncState, CustomData)
            If data IsNot Nothing Then
                Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                                  data.Name, data.CreationTime, data.ThreadNum)
            End If
        Next
    End Sub
End Module
' The example displays output like the following:
'    Task #0 created at 635116451245250515, ran on thread #3, RanToCompletion
'    Task #1 created at 635116451245270515, ran on thread #4, RanToCompletion
'    Task #2 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #3 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #4 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #5 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #6 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #7 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #8 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #9 created at 635116451245270515, ran on thread #3, RanToCompletion
```

Task 和 Task<TResult> 均公开静态 Factory 属性，该属性返回 TaskFactory 的默认实例，因此你可以调用该方法为 `Task.Factory.StartNew()` 。此外，在以下示例中，由于任务的类型为 System.Threading.Tasks.Task<TResult>，因此每个任务都具有包含计算结果的公共 Task<TResult>.Result 属性。任务以异步方式运行，可以按任意顺序完成。如果在计算完成之前访问 Result 属性，则该属性将阻止调用线程，直到值可用为止。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
        Task<Double>[] taskArray = { Task<Double>.Factory.StartNew(() => DoComputation(1.0)),
                                     Task<Double>.Factory.StartNew(() => DoComputation(100.0)),
                                     Task<Double>.Factory.StartNew(() => DoComputation(1000.0)) };

        var results = new Double[taskArray.Length];
        Double sum = 0;

        for (int i = 0; i < taskArray.Length; i++) {
            results[i] = taskArray[i].Result;
            Console.Write("{0:N1} {1}", results[i],
                          i == taskArray.Length - 1 ? "= " : "+ ");
            sum += results[i];
        }
        Console.WriteLine("{0:N1}", sum);
   }

   private static Double DoComputation(Double start)
   {
      Double sum = 0;
      for (var value = start; value <= start + 10; value += .1)
         sum += value;

      return sum;
   }
}
// The example displays the following output:
//        606.0 + 10,605.0 + 100,495.0 = 111,706.0
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim taskArray() = {Task(Of Double).Factory.StartNew(Function() DoComputation(1.0)),
                           Task(Of Double).Factory.StartNew(Function() DoComputation(100.0)),
                           Task(Of Double).Factory.StartNew(Function() DoComputation(1000.0))}

        Dim results(taskArray.Length - 1) As Double
        Dim sum As Double

        For i As Integer = 0 To taskArray.Length - 1
            results(i) = taskArray(i).Result
            Console.Write("{0:N1} {1}", results(i),
                          If(i = taskArray.Length - 1, "= ", "+ "))
            sum += results(i)
        Next
        Console.WriteLine("{0:N1}", sum)
    End Sub

    Private Function DoComputation(start As Double) As Double
        Dim sum As Double
        For value As Double = start To start + 10 Step .1
            sum += value
        Next
        Return sum
    End Function
End Module
' The example displays the following output:
'       606.0 + 10,605.0 + 100,495.0 = 111,706.0
```

有关详细信息，请参阅如何：从任务中返回值。

使用 lambda 表达式创建委托时，你有权访问源代码中当时可见的所有变量。然而，在某些情况下，特别是在循环中，lambda 不按照预期的方式捕获变量。它仅捕获最终值，而不是它每次迭代后更改的值。以下示例演示了该问题。它将循环计数器传递给实例化 `CustomData` 对象并使用循环计数器作为对象标识符的 lambda 表达式。如示例输出所示，每个 `CustomData` 对象都具有相同的标识符。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      // Create the task object by using an Action(Of Object) to pass in the loop
      // counter. This produces an unexpected result.
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj) => {
                                                 var data = new CustomData() {Name = i, CreationTime =
DateTime.Now.Ticks};
                                                 data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                                 Console.WriteLine("Task #{0} created at {1} on thread #
{2}.",
                                                                   data.Name, data.CreationTime,
data.ThreadNum);
                                               },
                                              i );
      }
      Task.WaitAll(taskArray);
   }
}
// The example displays output like the following:
//       Task #10 created at 635116418427727841 on thread #4.
//       Task #10 created at 635116418427737842 on thread #4.
//       Task #10 created at 635116418427737842 on thread #4.
//       Task #10 created at 635116418427737842 on thread #4.
//       Task #10 created at 635116418427737842 on thread #4.
//       Task #10 created at 635116418427737842 on thread #4.
//       Task #10 created at 635116418427727841 on thread #3.
//       Task #10 created at 635116418427747843 on thread #3.
//       Task #10 created at 635116418427747843 on thread #3.
//       Task #10 created at 635116418427737842 on thread #4.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
    Public CreationTime As Long
    Public Name As Integer
    Public ThreadNum As Integer
End Class

Module Example
    Public Sub Main()
        ' Create the task object by using an Action(Of Object) to pass in the loop
        ' counter. This produces an unexpected result.
        Dim taskArray(9) As Task
        For i As Integer = 0 To taskArray.Length - 1
            taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                     Dim data As New CustomData With {.Name = i,
.CreationTime = DateTime.Now.Ticks}
                                                     data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                                     Console.WriteLine("Task #{0} created at {1} on thread #
{2}.",
                                                                       data.Name, data.CreationTime,
data.ThreadNum)
                                                 End Sub,
                                                 i)
        Next
        Task.WaitAll(taskArray)
    End Sub
End Module
' The example displays output like the following:
'       Task #10 created at 635116418427727841 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427737842 on thread #4.
'       Task #10 created at 635116418427727841 on thread #3.
'       Task #10 created at 635116418427747843 on thread #3.
'       Task #10 created at 635116418427747843 on thread #3.
'       Task #10 created at 635116418427737842 on thread #4.
```

通过使用构造函数向任务提供状态对象，可以在每次迭代时访问该值。以下示例在上一示例的基础上做了修改，在创建 `CustomData` 对象时使用循环计数器，该对象继而传递给 lambda 表达式。如示例输出所示，每个 `CustomData` 对象现在都具有唯一的一个标识符，该标识符基于该对象实例化时循环计数器的值。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      // Create the task object by using an Action(Of Object) to pass in custom data
      // to the Task constructor. This is useful when you need to capture outer variables
      // from within a loop.
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                                  Console.WriteLine("Task #{0} created at {1} on thread #
{2}.",
                                                                    data.Name, data.CreationTime,
data.ThreadNum);
                                               },
                                               new CustomData() {Name = i, CreationTime =
DateTime.Now.Ticks} );
      }
      Task.WaitAll(taskArray);
   }
}
// The example displays output like the following:
//       Task #0 created at 635116412924597583 on thread #3.
//       Task #1 created at 635116412924607584 on thread #4.
//       Task #3 created at 635116412924607584 on thread #4.
//       Task #4 created at 635116412924607584 on thread #4.
//       Task #2 created at 635116412924607584 on thread #3.
//       Task #6 created at 635116412924607584 on thread #3.
//       Task #5 created at 635116412924607584 on thread #4.
//       Task #8 created at 635116412924607584 on thread #4.
//       Task #7 created at 635116412924607584 on thread #3.
//       Task #9 created at 635116412924607584 on thread #4.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
    Public CreationTime As Long
    Public Name As Integer
    Public ThreadNum As Integer
End Class

Module Example
    Public Sub Main()
        ' Create the task object by using an Action(Of Object) to pass in custom data
        ' to the Task constructor. This is useful when you need to capture outer variables
        ' from within a loop.
        Dim taskArray(9) As Task
        For i As Integer = 0 To taskArray.Length - 1
            taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                     Dim data As CustomData = TryCast(obj, CustomData)
                                                     If data Is Nothing Then Return

                                                     data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                                     Console.WriteLine("Task #{0} created at {1} on thread #
{2}.",
                                                                       data.Name, data.CreationTime,
data.ThreadNum)
                                                 End Sub,
                                                 New CustomData With {.Name = i, .CreationTime =
DateTime.Now.Ticks})
        Next
        Task.WaitAll(taskArray)
    End Sub
End Module
' The example displays output like the following:
'       Task #0 created at 635116412924597583 on thread #3.
'       Task #1 created at 635116412924607584 on thread #4.
'       Task #3 created at 635116412924607584 on thread #4.
'       Task #4 created at 635116412924607584 on thread #4.
'       Task #2 created at 635116412924607584 on thread #3.
'       Task #6 created at 635116412924607584 on thread #3.
'       Task #5 created at 635116412924607584 on thread #4.
'       Task #8 created at 635116412924607584 on thread #4.
'       Task #7 created at 635116412924607584 on thread #3.
'       Task #9 created at 635116412924607584 on thread #4.
```

此状态作为参数传递给任务委托，并且可通过使用 Task.AsyncState 属性从任务对象访问。以下示例在上一示例的基础上演变而来。它使用 AsyncState 属性显示关于传递到 lambda 表达式的 `CustomData` 对象的信息。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class CustomData
{
   public long CreationTime;
   public int Name;
   public int ThreadNum;
}

public class Example
{
   public static void Main()
   {
      Task[] taskArray = new Task[10];
      for (int i = 0; i < taskArray.Length; i++) {
         taskArray[i] = Task.Factory.StartNew( (Object obj ) => {
                                                  CustomData data = obj as CustomData;
                                                  if (data == null)
                                                     return;

                                                  data.ThreadNum = Thread.CurrentThread.ManagedThreadId;
                                               },
                                               new CustomData() {Name = i, CreationTime =
DateTime.Now.Ticks} );
      }
      Task.WaitAll(taskArray);
      foreach (var task in taskArray) {
         var data = task.AsyncState as CustomData;
         if (data != null)
            Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                              data.Name, data.CreationTime, data.ThreadNum);
      }
   }
}
// The example displays output like the following:
//       Task #0 created at 635116412924597583 on thread #3.
//       Task #1 created at 635116412924607584 on thread #4.
//       Task #3 created at 635116412924607584 on thread #4.
//       Task #4 created at 635116412924607584 on thread #4.
//       Task #2 created at 635116412924607584 on thread #3.
//       Task #6 created at 635116412924607584 on thread #3.
//       Task #5 created at 635116412924607584 on thread #4.
//       Task #8 created at 635116412924607584 on thread #4.
//       Task #7 created at 635116412924607584 on thread #3.
//       Task #9 created at 635116412924607584 on thread #4.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Class CustomData
    Public CreationTime As Long
    Public Name As Integer
    Public ThreadNum As Integer
End Class

Module Example
    Public Sub Main()
        Dim taskArray(9) As Task
        For i As Integer = 0 To taskArray.Length - 1
            taskArray(i) = Task.Factory.StartNew(Sub(obj As Object)
                                                     Dim data As CustomData = TryCast(obj, CustomData)
                                                     If data Is Nothing Then Return

                                                     data.ThreadNum = Thread.CurrentThread.ManagedThreadId
                                                 End Sub,
                                                 New CustomData With {.Name = i, .CreationTime =
DateTime.Now.Ticks})
        Next
        Task.WaitAll(taskArray)

        For Each task In taskArray
            Dim data = TryCast(task.AsyncState, CustomData)
            If data IsNot Nothing Then
                Console.WriteLine("Task #{0} created at {1}, ran on thread #{2}.",
                                  data.Name, data.CreationTime, data.ThreadNum)
            End If
        Next
    End Sub
End Module
' The example displays output like the following:
'    Task #0 created at 635116451245250515, ran on thread #3, RanToCompletion
'    Task #1 created at 635116451245270515, ran on thread #4, RanToCompletion
'    Task #2 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #3 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #4 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #5 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #6 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #7 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #8 created at 635116451245270515, ran on thread #3, RanToCompletion
'    Task #9 created at 635116451245270515, ran on thread #3, RanToCompletion
```

任务 ID

每个任务都获得一个在应用程序域中唯一标识自己的整数 ID，可以使用 Task.Id 属性访问该 ID。该 ID 可有效用于在 Visual Studio 调试器的"并行堆栈"和"任务"窗口中查看任务信息。该 ID 是惰式创建的，这意味着它不会在被请求之前创建；因此每次运行该程序时，任务可能具有不同的 ID。有关如何在调试器中查看任务 ID 的详细信息，请参阅使用任务窗口和使用并行堆栈窗口。

任务创建选项

创建任务的大多数 API 提供接受 TaskCreationOptions 参数的重载。通过指定下列某个或多个选项，可指示任务计划程序在线程池中安排任务计划的方式。可以使用位 OR 运算组合选项。

下面的示例演示一个具有 LongRunning 和 PreferFairness 选项的任务。

```
var task3 = new Task(() => MyLongRunningMethod(),
                     TaskCreationOptions.LongRunning | TaskCreationOptions.PreferFairness);
task3.Start();
```

```
Dim task3 = New Task(Sub() MyLongRunningMethod(),
                         TaskCreationOptions.LongRunning Or TaskCreationOptions.PreferFairness)
task3.Start()
```

任务、线程和区域性

每个线程都具有一个关联的区域性和 UI 区域性，分别由 Thread.CurrentCulture 和 Thread.CurrentUICulture 属性定义。线程的区域性用在诸如格式、分析、排序和字符串比较操作中。线程的 UI 区域性用于查找资源。

除非使用 CultureInfo.DefaultThreadCurrentCulture 和 CultureInfo.DefaultThreadCurrentUICulture 属性在应用程序域中为所有线程指定默认区域性，线程的默认区域性和 UI 区域性则由系统区域性定义。如果你显式设置线程的区域性并启动新线程，则新线程不会继承正在调用的线程的区域性；相反，其区域性就是默认系统区域性。但是，在基于任务的编程中，任务使用调用线程的区域性，即使任务在不同线程上以异步方式运行也是如此。

下面的示例提供了简单的演示。它将应用的当前区域性更改为 French (France)；或者，如果 French (France) 已为当前区域性，则将其更改为 English (United States)。然后，调用一个名为 `formatDelegate` 的委托，该委托返回在新区域性中格式化为货币值的数字。无论委托是由任务同步调用还是异步调用，该任务都将使用调用线程的区域性。

```
using System;
using System.Globalization;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
       decimal[] values = { 163025412.32m, 18905365.59m };
       string formatString = "C2";
       Func<String> formatDelegate = () => { string output = String.Format("Formatting using the {0} culture
on thread {1}.\n",
                                                                          CultureInfo.CurrentCulture.Name,

Thread.CurrentThread.ManagedThreadId);
                                             foreach (var value in values)
                                                output += String.Format("{0}   ",
value.ToString(formatString));

                                             output += Environment.NewLine;
                                             return output;
                                           };

       Console.WriteLine("The example is running on thread {0}",
                         Thread.CurrentThread.ManagedThreadId);
       // Make the current culture different from the system culture.
       Console.WriteLine("The current culture is {0}",
                         CultureInfo.CurrentCulture.Name);
       if (CultureInfo.CurrentCulture.Name == "fr-FR")
          Thread.CurrentThread.CurrentCulture = new CultureInfo("en-US");
       else
          Thread.CurrentThread.CurrentCulture = new CultureInfo("fr-FR");

       Console.WriteLine("Changed the current culture to {0}.\n",
                         CultureInfo.CurrentCulture.Name);
```

```
       // Execute the delegate synchronously.
       Console.WriteLine("Executing the delegate synchronously:");
       Console.WriteLine(formatDelegate());

       // Call an async delegate to format the values using one format string.
       Console.WriteLine("Executing a task asynchronously:");
       var t1 = Task.Run(formatDelegate);
       Console.WriteLine(t1.Result);

       Console.WriteLine("Executing a task synchronously:");
       var t2 = new Task<String>(formatDelegate);
       t2.RunSynchronously();
       Console.WriteLine(t2.Result);
   }
}
// The example displays the following output:
//         The example is running on thread 1
//         The current culture is en-US
//         Changed the current culture to fr-FR.
//
//         Executing the delegate synchronously:
//         Formatting using the fr-FR culture on thread 1.
//         163 025 412,32 €   18 905 365,59 €
//
//         Executing a task asynchronously:
//         Formatting using the fr-FR culture on thread 3.
//         163 025 412,32 €   18 905 365,59 €
//
//         Executing a task synchronously:
//         Formatting using the fr-FR culture on thread 1.
//         163 025 412,32 €   18 905 365,59 €
```

```
Imports System.Globalization
Imports System.Threading

Module Example
    Public Sub Main()
        Dim values() As Decimal = {163025412.32D, 18905365.59D}
        Dim formatString As String = "C2"
        Dim formatDelegate As Func(Of String) = Function()
                                                    Dim output As String = String.Format("Formatting using
the {0} culture on thread {1}.",

CultureInfo.CurrentCulture.Name,

Thread.CurrentThread.ManagedThreadId)
                                                    output += Environment.NewLine
                                                    For Each value In values
                                                        output += String.Format("{0}   ",
value.ToString(formatString))
                                                    Next
                                                    output += Environment.NewLine
                                                    Return output
                                                End Function

        Console.WriteLine("The example is running on thread {0}",
                          Thread.CurrentThread.ManagedThreadId)
        ' Make the current culture different from the system culture.
        Console.WriteLine("The current culture is {0}",
                          CultureInfo.CurrentCulture.Name)
        If CultureInfo.CurrentCulture.Name = "fr-FR" Then
            Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
        Else
            Thread.CurrentThread.CurrentCulture = New CultureInfo("fr-FR")
        End If
        Console.WriteLine("Changed the current culture to {0}.",
                          CultureInfo.CurrentCulture.Name)
```

```
        Console.WriteLine()

        ' Execute the delegate synchronously.
        Console.WriteLine("Executing the delegate synchronously:")
        Console.WriteLine(formatDelegate())

        ' Call an async delegate to format the values using one format string.
        Console.WriteLine("Executing a task asynchronously:")
        Dim t1 = Task.Run(formatDelegate)
        Console.WriteLine(t1.Result)

        Console.WriteLine("Executing a task synchronously:")
        Dim t2 = New Task(Of String)(formatDelegate)
        t2.RunSynchronously()
        Console.WriteLine(t2.Result)
    End Sub
End Module

' The example displays the following output:
'
'          The example is running on thread 1
'          The current culture is en-US
'          Changed the current culture to fr-FR.
'
'          Executing the delegate synchronously:
'          Formatting Imports the fr-FR culture on thread 1.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task asynchronously:
'          Formatting Imports the fr-FR culture on thread 3.
'          163 025 412,32 €   18 905 365,59 €
'
'          Executing a task synchronously:
'          Formatting Imports the fr-FR culture on thread 1.
'          163 025 412,32 €   18 905 365,59 €
```

NOTE

在 .NET Framework 4.6 之前的 .NET Framework 版本中，任务的区域性由它在其上运行的线程区域性确定，而不是调用线程的区域性 。对于异步任务，这意味着任务使用的区域性可能不同于调用线程的区域性。

有关异步任务和区域性的详细信息，请参阅 CultureInfo 主题中的“区域性和基于异步任务的操作”部分。

创建任务延续

使用 Task.ContinueWith 和 Task<TResult>.ContinueWith 方法，可以指定要在先行任务完成时启动的任务。延续任务的委托已传递了对先行任务的引用，因此它可以检查先行任务的状态，并通过检索 Task<TResult>.Result 属性的值将先行任务的输出用作延续任务的输入。

在下面的示例中，`getData` 任务通过调用 TaskFactory.StartNew<TResult>(Func<TResult>) 方法来启动。当 `processData` 完成时，`getData` 任务自动启动，当 `displayData` 完成时，`processData` 启动。`getData` 产生一个整数数组，通过 `processData` 任务的 `getData` 属性，Task<TResult>.Result 任务可访问该数组。`processData` 任务处理该数组并返回结果，结果的类型从传递到 Task<TResult>.ContinueWith<TNewResult>(Func<Task<TResult>,TNewResult>) 方法的 Lambda 表达式的返回类型推断而来。`displayData` 完成时，`processData` 任务自动执行，而 Tuple<T1,T2,T3> 任务可通过 `processData` 任务的 `displayData` 属性访问由 `processData` lambda 表达式返回的 Task<TResult>.Result 对象。`displayData` 任务采用 `processData` 任务的结果，继而得出自己的结果，其类型以相似方式推断而来，且可由程序中的 Result 属性使用。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var getData = Task.Factory.StartNew(() => {
                                             Random rnd = new Random();
                                             int[] values = new int[100];
                                             for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
                                                values[ctr] = rnd.Next();

                                             return values;
                                          } );
      var processData = getData.ContinueWith((x) => {
                                                int n = x.Result.Length;
                                                long sum = 0;
                                                double mean;

                                                for (int ctr = 0; ctr <= x.Result.GetUpperBound(0); ctr++)
                                                   sum += x.Result[ctr];

                                                mean = sum / (double) n;
                                                return Tuple.Create(n, sum, mean);
                                             } );
      var displayData = processData.ContinueWith((x) => {
                                                    return String.Format("N={0:N0}, Total = {1:N0}, Mean =
{2:N2}",
                                                                         x.Result.Item1, x.Result.Item2,
                                                                         x.Result.Item3);
                                                 } );
      Console.WriteLine(displayData.Result);
   }
}
// The example displays output similar to the following:
//    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim getData = Task.Factory.StartNew(Function()
                                                Dim rnd As New Random()
                                                Dim values(99) As Integer
                                                For ctr = 0 To values.GetUpperBound(0)
                                                    values(ctr) = rnd.Next()
                                                Next
                                                Return values
                                            End Function)
        Dim processData = getData.ContinueWith(Function(x)
                                                   Dim n As Integer = x.Result.Length
                                                   Dim sum As Long
                                                   Dim mean As Double

                                                   For ctr = 0 To x.Result.GetUpperBound(0)
                                                       sum += x.Result(ctr)
                                                   Next
                                                   mean = sum / n
                                                   Return Tuple.Create(n, sum, mean)
                                               End Function)
        Dim displayData = processData.ContinueWith(Function(x)
                                                       Return String.Format("N={0:N0}, Total = {1:N0}, Mean
= {2:N2}",
                                                                            x.Result.Item1, x.Result.Item2,
                                                                            x.Result.Item3)
                                                   End Function)
        Console.WriteLine(displayData.Result)
    End Sub
End Module
' The example displays output like the following:
'   N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

因为 Task.ContinueWith 是实例方法，所以你可以将方法调用链接在一起，而不是为每个先行任务去实例化 Task<TResult> 对象。以下示例与上一示例在功能上等同，唯一的不同在于它将对 Task.ContinueWith 方法的调用链接在一起。请注意，通过方法调用链返回的 Task<TResult> 对象是最终延续任务。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var displayData = Task.Factory.StartNew(() => {
                                              Random rnd = new Random();
                                              int[] values = new int[100];
                                              for (int ctr = 0; ctr <= values.GetUpperBound(0); ctr++)
                                                 values[ctr] = rnd.Next();

                                              return values;
                                            } ).
                        ContinueWith((x) => {
                                       int n = x.Result.Length;
                                       long sum = 0;
                                       double mean;

                                       for (int ctr = 0; ctr <= x.Result.GetUpperBound(0); ctr++)
                                          sum += x.Result[ctr];

                                       mean = sum / (double) n;
                                       return Tuple.Create(n, sum, mean);
                                     } ).
                        ContinueWith((x) => {
                                       return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}",
                                                            x.Result.Item1, x.Result.Item2,
                                                            x.Result.Item3);
                                     } );
      Console.WriteLine(displayData.Result);
   }
}
// The example displays output similar to the following:
//    N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim displayData = Task.Factory.StartNew(Function()
                                                    Dim rnd As New Random()
                                                    Dim values(99) As Integer
                                                    For ctr = 0 To values.GetUpperBound(0)
                                                        values(ctr) = rnd.Next()
                                                    Next
                                                    Return values
                                                End Function). _
                          ContinueWith(Function(x)
                                           Dim n As Integer = x.Result.Length
                                           Dim sum As Long
                                           Dim mean As Double

                                           For ctr = 0 To x.Result.GetUpperBound(0)
                                               sum += x.Result(ctr)
                                           Next
                                           mean = sum / n
                                           Return Tuple.Create(n, sum, mean)
                                       End Function). _
                          ContinueWith(Function(x)
                                           Return String.Format("N={0:N0}, Total = {1:N0}, Mean = {2:N2}",
                                                                x.Result.Item1, x.Result.Item2,
                                                                x.Result.Item3)
                                       End Function)
        Console.WriteLine(displayData.Result)
    End Sub
End Module
' The example displays output like the following:
'   N=100, Total = 110,081,653,682, Mean = 1,100,816,536.82
```

使用 ContinueWhenAll 和 ContinueWhenAny 方法，可以从多个任务继续。

有关详细信息，请参阅使用延续任务链接任务。

创建分离的子任务

如果在任务中运行的用户代码创建一个新任务，且未指定 AttachedToParent 选项，则该新任务不采用任何特殊方式与父任务同步。这种不同步的任务类型称为“分离的嵌套任务”或“分离的子任务”。以下示例展示了创建一个分离子任务的任务。

```
var outer = Task.Factory.StartNew(() =>
{
    Console.WriteLine("Outer task beginning.");

    var child = Task.Factory.StartNew(() =>
    {
        Thread.SpinWait(5000000);
        Console.WriteLine("Detached task completed.");
    });
});

outer.Wait();
Console.WriteLine("Outer task completed.");
// The example displays the following output:
//    Outer task beginning.
//    Outer task completed.
//    Detached task completed.
```

```
Dim outer = Task.Factory.StartNew(Sub()
                                      Console.WriteLine("Outer task beginning.")
                                      Dim child = Task.Factory.StartNew(Sub()
                                                                            Thread.SpinWait(5000000)
                                                                            Console.WriteLine("Detached task
completed.")
                                                                        End Sub)
                                  End Sub)
outer.Wait()
Console.WriteLine("Outer task completed.")
' The example displays the following output:
'     Outer task beginning.
'     Outer task completed.
'    Detached child completed.
```

请注意，父任务不会等待分离子任务完成。

创建子任务

如果任务中运行的用户代码在创建任务时指定了 AttachedToParent 选项，新任务就称为父任务的附加子任务。因为父任务隐式地等待所有附加子任务完成，所以你可以使用 AttachedToParent 选项表示结构化的任务并行。以下示例展示了创建十个附加子任务的父任务。请注意，虽然此示例调用 Task.Wait 方法等待父任务完成，但不必显式等待附加子任务完成。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
                      Console.WriteLine("Parent task beginning.");
                      for (int ctr = 0; ctr < 10; ctr++) {
                         int taskNo = ctr;
                         Task.Factory.StartNew((x) => {
                                                  Thread.SpinWait(5000000);
                                                  Console.WriteLine("Attached child #{0} completed.",
                                                                    x);
                                               },
                                               taskNo, TaskCreationOptions.AttachedToParent);
                      }
                   });

      parent.Wait();
      Console.WriteLine("Parent task completed.");
   }
}
// The example displays output like the following:
//       Parent task beginning.
//       Attached child #9 completed.
//       Attached child #0 completed.
//       Attached child #8 completed.
//       Attached child #1 completed.
//       Attached child #7 completed.
//       Attached child #2 completed.
//       Attached child #6 completed.
//       Attached child #3 completed.
//       Attached child #5 completed.
//       Attached child #4 completed.
//       Parent task completed.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim parent = Task.Factory.StartNew(Sub()
                                               Console.WriteLine("Parent task beginning.")
                                               For ctr As Integer = 0 To 9
                                                   Dim taskNo As Integer = ctr
                                                   Task.Factory.StartNew(Sub(x)
                                                                             Thread.SpinWait(5000000)
                                                                             Console.WriteLine("Attached
child #{0} completed.",
                                                                                               x)
                                                                         End Sub,
                                                                         taskNo,
TaskCreationOptions.AttachedToParent)
                                               Next
                                           End Sub)
        parent.Wait()
        Console.WriteLine("Parent task completed.")
    End Sub
End Module
' The example displays output like the following:
'       Parent task beginning.
'       Attached child #9 completed.
'       Attached child #0 completed.
'       Attached child #8 completed.
'       Attached child #1 completed.
'       Attached child #7 completed.
'       Attached child #2 completed.
'       Attached child #6 completed.
'       Attached child #3 completed.
'       Attached child #5 completed.
'       Attached child #4 completed.
'       Parent task completed.
```

父任务可使用 TaskCreationOptions.DenyChildAttach 选项阻止其他任务附加到父任务。有关详细信息，请参阅附加和分离的子任务。

等待任务完成

System.Threading.Tasks.Task 和 System.Threading.Tasks.Task<TResult> 类型提供了 Task.Wait 方法的若干重载，以便能够等待任务完成。此外，使用静态 Task.WaitAll 和 Task.WaitAny 方法的重载可以等待一批任务中的任一任务或所有任务完成。

通常，会出于以下某个原因等待任务：

- 主线程依赖于任务计算的最终结果。
- 你必须处理可能从任务引发的异常。
- 应用程序可以在所有任务执行完毕之前终止。例如，执行 `Main`（应用程序入口点）中的所有同步代码后，控制台应用程序将立即终止。

下面的示例演示不包含异常处理的基本模式。

```
Task[] tasks = new Task[3]
{
    Task.Factory.StartNew(() => MethodA()),
    Task.Factory.StartNew(() => MethodB()),
    Task.Factory.StartNew(() => MethodC())
};

//Block until all tasks complete.
Task.WaitAll(tasks);

// Continue on this thread...
```

```
Dim tasks() =
{
    Task.Factory.StartNew(Sub() MethodA()),
    Task.Factory.StartNew(Sub() MethodB()),
    Task.Factory.StartNew(Sub() MethodC())
}

' Block until all tasks complete.
Task.WaitAll(tasks)

' Continue on this thread...
```

有关演示异常处理的示例，请参见异常处理。

某些重载允许你指定超时，而其他重载采用额外的 CancellationToken 作为输入参数，以便可以通过编程方式或根据用户输入来取消等待。

等待任务时，其实是在隐式等待使用 TaskCreationOptions.AttachedToParent 选项创建的该任务的所有子级。Task.Wait 在该任务已完成时立即返回。Task.Wait 方法将抛出由某任务引发的任何异常，即使 Task.Wait 方法是在该任务完成之后调用的。

组合任务

Task 类和 Task<TResult> 类提供多种方法，这些方法能够帮助你组合多个任务以实现常见模式，并更好地使用由 C#、Visual Basic 和 F# 提供的异步语言功能。本节介绍了 WhenAll、WhenAny、Delay 和 FromResult 方法。

Task.WhenAll

Task.WhenAll 方法异步等待多个 Task 或 Task<TResult> 对象完成。通过它提供的重载版本可以等待非均匀任务组。例如，你可以等待多个 Task 和 Task<TResult> 对象在一个方法调用中完成。

Task.WhenAny

Task.WhenAny 方法异步等待多个 Task 或 Task<TResult> 对象中的一个完成。与在 Task.WhenAll 方法中一样，该方法提供重载版本，让你能等待非均匀任务组。WhenAny 方法在下列情境中尤其有用。

- 冗余运算。请考虑可以用多种方式执行的算法或运算。你可使用 WhenAny 方法来选择先完成的运算，然后取消剩余的运算。
- 交叉运算。你可启动必须全部完成的多项运算，并使用 WhenAny 方法在每项运算完成时处理结果。在一项运算完成后，可以启动一个或多个其他任务。
- 受限制的运算。你可使用 WhenAny 方法通过限制并发运算的数量来扩展前面的情境。
- 过期的运算。你可使用 WhenAny 方法在一个或多个任务与特定时间后完成的任务（例如 Delay 方法返回的任务）间进行选择。下节描述了 Delay 方法。

Task.Delay

Task.Delay 方法将生成在指定时间后完成的 Task 对象。你可使用此方法来生成偶尔轮询数据的循环，引入超时，

将对用户输入的处理延迟预定的一段时间等。

Task(T).FromResult

通过使用 Task.FromResult 方法，你可以创建包含预计算结果的 Task<TResult> 对象。执行返回 Task<TResult> 对象的异步运算，且已计算该 Task<TResult> 对象的结果时，此方法将十分有用。有关使用 FromResult 检索缓存中包含的异步下载运算结果的示例，请参阅如何：创建预先计算的任务。

处理任务中的异常

当某个任务抛出一个或多个异常时，异常包装在 AggregateException 异常中。该异常传播回与该任务联接的线程，通常该线程正在等待该任务完成或该线程访问 Result 属性。此行为用于强制实施 .NET Framework 策略 - 默认所有未处理的异常应终止进程。调用代码可以通过使用 `try` / `catch` 块中的以下任意方法来处理异常：

- Wait 方法
- WaitAll 方法
- WaitAny 方法
- Result 属性

联接线程也可以通过在对任务进行垃圾回收之前访问 Exception 属性来处理异常。通过访问此属性，可防止未处理的异常在对象完成时触发终止进程的异常传播行为。

有关异常和任务的的详细信息，请参阅异常处理。

取消任务

Task 类支持协作取消，并与 .NET Framework 4 中新增的 System.Threading.CancellationTokenSource 类和 System.Threading.CancellationToken 类完全集成。System.Threading.Tasks.Task 类中的大多数构造函数采用 CancellationToken 对象作为输入参数。许多 StartNew 和 Run 重载还包括 CancellationToken 参数。

你可以创建标记，并使用 CancellationTokenSource 类在以后某一时间发出取消请求。可以将该标记作为参数传递给 Task，还可以在执行响应取消请求的工作的用户委托中引用同一标记。

有关详细信息，请参阅任务取消和如何：取消任务及其子级。

TaskFactory 类

TaskFactory 类提供静态方法，这些方法封装了用于创建和启动任务和延续任务的一些常用模式。

- 最常用模式为 StartNew，它在一个语句中创建并启动任务。
- 如果通过多个先行任务创建延续任务，请使用 ContinueWhenAll 方法或 ContinueWhenAny 方法，或它们在 Task<TResult> 类中的相当方法。有关详细信息，请参阅使用延续任务链接任务。
- 若要在 `BeginX` 或 `EndX` 实例中封装异步编程模型 Task 和 Task<TResult> 方法，请使用 FromAsync 方法。有关详细信息，请参阅 TPL 和传统 .NET Framework 异步编程。

默认的 TaskFactory 可作为 Task 类或 Task<TResult> 类上的静态属性访问。你还可以直接实例化 TaskFactory 并指定各种选项，包括 CancellationToken、TaskCreationOptions 选项、TaskContinuationOptions 选项或 TaskScheduler。创建任务工厂时所指定的任何选项将应用于它创建的所有任务，除非 Task 是通过使用 TaskCreationOptions 枚举创建的(在这种情况下，任务的选项重写任务工厂的选项)。

无委托的任务

在某些情况下，可能需要使用 Task 封装由外部组件(而不是你自己的用户委托)执行的某个异步操作。如果该操作基于异步编程模型 Begin/End 模式，你可以使用 FromAsync 方法。如果不是这种情况，你可以使用

TaskCompletionSource<TResult> 对象将该操作包装在任务中，并因而获得 Task 可编程性的一些好处，例如对异常传播和延续的支持。有关详细信息，请参阅 TaskCompletionSource<TResult>。

自定义计划程序

大多数应用程序或库开发人员并不关心任务在哪个处理器上运行、任务如何将其工作与其他任务同步以及如何在 System.Threading.ThreadPool 中计划任务。他们只需要它在主机上尽可能高效地执行。如果需要对计划细节进行更细化的控制，可以使用任务并行库在默认任务计划程序上配置一些设置，甚至是提供自定义计划程序。有关详细信息，请参阅 TaskScheduler。

相关数据结构

TPL 有几种在并行和顺序方案中都有用的新公共类型。它们包括 System.Collections.Concurrent 命名空间中的一些线程安全的、快速且可缩放的集合类，还包括一些新的同步类型（例如 System.Threading.Semaphore 和 System.Threading.ManualResetEventSlim），对特定类型的工作负荷而言，这些新同步类型比旧的同步类型效率更高。.NET Framework 4 中的其他新类型（例如 System.Threading.Barrier 和 System.Threading.SpinLock）提供了早期版本中未提供的功能。有关详细信息，请参阅用于并行编程的数据结构。

自定义任务类型

建议不要从 System.Threading.Tasks.Task 或 System.Threading.Tasks.Task<TResult> 继承。相反，我们建议你使用 AsyncState 属性将其他数据或状态与 Task 或 Task<TResult> 对象相关联。还可以使用扩展方法扩展 Task 和 Task<TResult> 类的功能。有关扩展方法的详细信息，请参阅扩展方法和扩展方法。

如果必须从 Task 或 Task<TResult> 继承，则不能使用 Run 或 System.Threading.Tasks.TaskFactory，System.Threading.Tasks.TaskFactory<TResult> 或 System.Threading.Tasks.TaskCompletionSource<TResult> 类创建自定义任务类型的实例，因为这些类仅创建 Task 和 Task<TResult> 对象。此外，不能使用 Task、Task<TResult>、TaskFactory 和 TaskFactory<TResult> 提供的任务延续机制创建自定义任务类型的实例，因为这些机制也只创建 Task 和 Task<TResult> 对象。

相关主题

TITLE	II
使用延续任务来链接任务	描述延续任务的工作方式。
附加和分离的子任务	描述附加子任务和分离子任务之间的差异。
任务取消	描述在 Task 对象中内置的取消支持。
异常处理	描述如何处理并行线程中的异常。
如何：使用 Parallel.Invoke 来执行并行操作	描述如何使用 Invoke。
如何：从任务中返回值	描述如何从任务中返回值。
如何：取消任务及其子级	描述如何取消任务。
如何：创建预先计算的任务	描述如何使用 Task.FromResult 方法去检索缓存中包含的异步下载运算结果。
如何：使用并行任务遍历二叉树	描述如何使用任务遍历二叉树。

TITLE	𝐂
如何：解除嵌套任务的包装	演示如何使用 Unwrap 扩展方法。
数据并行	描述如何使用 For 和 ForEach 来创建循环访问数据的并行循环。
并行编程	.NET Framework 并行编程的顶级节点。

请参阅

- 并行编程
- 使用 .NET Core 和 .NET Standard 并行编程的示例

使用延续任务来链接任务

2021/11/16 •

在异步编程中，一个异步操作在完成时调用另一个操作的情况较常见。延续使后续操作可以使用第一次操作的结果。传统上，延续性是通过使用回调方法完成的。在任务并行库中，*延续任务* 提供了同样的功能。延续任务（也简称为“延续”）是一个异步任务，在完成时由另一个任务（称为“先行任务”）调用。

尽管延续相对容易使用，但也十分强大和灵活。例如，你可以：

- 将数据从前面的任务传递到延续。
- 指定将调用或不调用延续所依据的精确条件。
- 在延续启动之前取消延续，或在延续正在运行时以协作方式取消延续。
- 提供有关应如何计划延续的提示。
- 从同一前面的任务中调用多个延续。
- 在多个前面的任务中的全部或任意任务完成时调用一个延续。
- 将延续依次相连，形成任意长度。
- 使用延续来处理前面的任务所引发的异常。

关于延续

延续创建时的状态为 WaitingForActivation 。在一个或多个前面的任务完成时，它将自动激活。若在用户代码中对延续调用 Task.Start ，将引发 System.InvalidOperationException 异常。

延续本身是 Task ，并不阻止它在其上启动的线程。调用 Task.Wait 方法进行阻止，直到延续任务完成。

为一个先行任务创建延续

通过调用 Task.ContinueWith 方法创建在其前面的任务完成时执行的延续。下面的示例演示基本模式（为清楚起见，省略了异常处理）。它会执行一个先行任务 - `taskA`，将返回一个 DayOfWeek 对象，指示当天为周几。前面的任务完成时，将向延续任务 `continuation` 传递前面的任务，并显示包含其结果的字符串。

> **NOTE**
>
> 本文中的 C# 示例利用 `Main` 方法的 `async` 修饰符。此功能在 C# 7.1 及更高版本中提供。以前的版本在编译此示例代码时生成 `CS5001` 。需要将语言版本设置为 C#7.1 或更高版本。可以通过有关配置语言版本的文章了解如何配置语言版本。

```
using System;
using System.Threading.Tasks;

public class SimpleExample
{
    public static async Task Main()
    {
        // Declare, assign, and start the antecedent task.
        Task<DayOfWeek> taskA = Task.Run(() => DateTime.Today.DayOfWeek);

        // Execute the continuation when the antecedent finishes.
        await taskA.ContinueWith(antecedent => Console.WriteLine($"Today is {antecedent.Result}."));
    }
}
// The example displays the following output:
//       Today is Monday.
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        ' Execute the antecedent.
        Dim taskA As Task(Of DayOfWeek) = Task.Run(Function() DateTime.Today.DayOfWeek)

        ' Execute the continuation when the antecedent finishes.
        Dim continuation As Task = taskA.ContinueWith(Sub(antecedent)
                                                          Console.WriteLine("Today is {0}.",
antecedent.Result)
                                                      End Sub)
        continuation.Wait()
    End Sub
End Module
' The example displays output like the following output:
'       Today is Monday.
```

为多个先行任务创建延续

还可以创建一个将在一组任务中的任意或全部任务完成时运行的延续任务。若要在所有前面的任务都完成后执行延续，则可以调用静态（在 Visual Basic 中为 `Shared` ）Task.WhenAll 方法或实例 TaskFactory.ContinueWhenAll 方法。若要在多个前面的任务中的任意任务完成时执行延续，则可以调用静态（在 Visual Basic 中为 `Shared` ）Task.WhenAny 方法或实例 TaskFactory.ContinueWhenAny 方法。

调用 Task.WhenAll 和 Task.WhenAny 重载不会阻止调用线程。不过，通常调用除 Task.WhenAll(IEnumerable<Task>) 和 Task.WhenAll(Task[]) 方法外的其他所有方法来检索返回的 Task<TResult>.Result 属性，这样不会阻止调用线程。

下面的示例调用 Task.WhenAll(IEnumerable<Task>) 方法来创建反映其 10 个前面的任务的结果的延续任务。每个前面的任务计算从 1 到 10 的索引值的平方值。如果前面的任务成功完成（其 Task.Status 属性为 TaskStatus.RanToCompletion），则延续任务的 Task<TResult>.Result 属性为由每个前面的任务返回的 Task<TResult>.Result 值组成的数组。该示例计算它们的总和，得出 1 到 10 之间的所有数字的平方和。

```
using System.Collections.Generic;
using System;
using System.Threading.Tasks;

public class WhenAllExample
{
    public static async Task Main()
    {
        var tasks = new List<Task<int>>();
        for (int ctr = 1; ctr <= 10; ctr++)
        {
            int baseValue = ctr;
            tasks.Add(Task.Factory.StartNew(b => (int)b * (int)b, baseValue));
        }

        var results = await Task.WhenAll(tasks);

        int sum = 0;
        for (int ctr = 0; ctr <= results.Length - 1; ctr++)
        {
            var result = results[ctr];
            Console.Write($"{result} {((ctr == results.Length - 1) ? "=" : "+")} ");
            sum += result;
        }

        Console.WriteLine(sum);
    }
}
// The example displays the similar output:
//    1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

```
Imports System.Collections.Generic
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim tasks As New List(Of Task(Of Integer))()
        For ctr As Integer = 1 To 10
            Dim baseValue As Integer = ctr
            tasks.Add(Task.Factory.StartNew(Function(b)
                                                Dim i As Integer = CInt(b)
                                                Return i * i
                                            End Function, baseValue))
        Next
        Dim continuation = Task.WhenAll(tasks)

        Dim sum As Long = 0
        For ctr As Integer = 0 To continuation.Result.Length - 1
            Console.Write("{0} {1} ", continuation.Result(ctr),
                          If(ctr = continuation.Result.Length - 1, "=", "+"))
            sum += continuation.Result(ctr)
        Next
        Console.WriteLine(sum)
    End Sub
End Module
' The example displays the following output:
'       1 + 4 + 9 + 16 + 25 + 36 + 49 + 64 + 81 + 100 = 385
```

延续选项

在创建单任务延续时，你可以使用 ContinueWith 重载，该重载采用 System.Threading.Tasks.TaskContinuationOptions 枚举值来指定启动延续所依据的条件。例如，可以将延续指定为仅在前面的任务已完成运行时运行，或仅在前面的任务完成时处于错误状态时运行。如果该条件在前面的任

务准备调用延续时未得到满足，则延续将直接转换为 TaskStatus.Canceled 状态，之后将无法启动。

许多多任务延续方法（如 TaskFactory.ContinueWhenAll 方法的重载）也包括 System.Threading.Tasks.TaskContinuationOptions 参数。但是，只有所有 System.Threading.Tasks.TaskContinuationOptions 枚举成员的一个子集有效。你可以指定在 System.Threading.Tasks.TaskContinuationOptions 枚举中具有对应的值的 System.Threading.Tasks.TaskCreationOptions 值，如 TaskContinuationOptions.AttachedToParent、TaskContinuationOptions.LongRunning和 TaskContinuationOptions.PreferFairness。如果为多任务延续指定 `NotOn` 或 `OnlyOn` 选项中的任意一个，则在运行时将引发 ArgumentOutOfRangeException 异常。

有关任务延续选项的详细信息，请参阅 TaskContinuationOptions 主题。

将数据传递到延续

Task.ContinueWith 方法将对前面的任务的引用以参数形式传递到延续的用户委托。如果前面的任务是一个 System.Threading.Tasks.Task<TResult> 对象，并且任务在完成前保持运行，则延续可以访问任务的 Task<TResult>.Result 属性。

在任务完成之前，将阻止 Task<TResult>.Result 属性。但是，如果任务已取消或出错，则尝试访问 Result 属性将引发 AggregateException 异常。可通过使用 OnlyOnRanToCompletion 选项避免此问题，如下面的示例所示。

```
using System;
using System.Threading.Tasks;

public class ResultExample
{
    public static async Task Main()
    {
       var task = Task.Run(
           () =>
           {
                DateTime date = DateTime.Now;
                return date.Hour > 17
                    ? "evening"
                    : date.Hour > 12
                        ? "afternoon"
                        : "morning";
            });

        await task.ContinueWith(
            antecedent =>
            {
                Console.WriteLine($"Good {antecedent.Result}!");
                Console.WriteLine($"And how are you this fine {antecedent.Result}?");
            }, TaskContinuationOptions.OnlyOnRanToCompletion);
   }
}
// The example displays the similar output:
//       Good afternoon!
//       And how are you this fine afternoon?
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim t = Task.Run(Function()
                             Dim dat As DateTime = DateTime.Now
                             If dat = DateTime.MinValue Then
                                 Throw New ArgumentException("The clock is not working.")
                             End If

                             If dat.Hour > 17 Then
                                 Return "evening"
                             Else If dat.Hour > 12 Then
                                 Return "afternoon"
                             Else
                                 Return "morning"
                             End If
                         End Function)
        Dim c = t.ContinueWith(Sub(antecedent)
                                   Console.WriteLine("Good {0}!",
                                                     antecedent.Result)
                                   Console.WriteLine("And how are you this fine {0}?",
                                                     antecedent.Result)
                               End Sub, TaskContinuationOptions.OnlyOnRanToCompletion)
        c.Wait()
    End Sub
End Module
' The example displays output like the following:
'       Good afternoon!
'       And how are you this fine afternoon?
```

如果希望延续即使在前面的任务未完成运行时也运行，则必须防止出现异常。一种方法是测试前面的任务的 Task.Status 属性，并且仅在状态不是 Result 或 Faulted 时才尝试访问 Canceled属性。也可以检查前面的任务的 Exception 属性。有关详细信息，请参阅异常处理。下面的示例将之前的示例修改为仅在前面的任务的状态为 Task<TResult>.Result 时，才访问该任务的 TaskStatus.RanToCompletion属性。

```
using System;
using System.Threading.Tasks;

public class ResultTwoExample
{
    public static async Task Main() =>
        await Task.Run(
            () =>
            {
                DateTime date = DateTime.Now;
                return date.Hour > 17
                   ? "evening"
                   : date.Hour > 12
                       ? "afternoon"
                       : "morning";
            })
            .ContinueWith(
                antecedent =>
                {
                    if (antecedent.Status == TaskStatus.RanToCompletion)
                    {
                        Console.WriteLine($"Good {antecedent.Result}!");
                        Console.WriteLine($"And how are you this fine {antecedent.Result}?");
                    }
                    else if (antecedent.Status == TaskStatus.Faulted)
                    {
                        Console.WriteLine(antecedent.Exception.GetBaseException().Message);
                    }
                });
}
// The example displays output like the following:
//       Good afternoon!
//       And how are you this fine afternoon?
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim t = Task.Run(Function()
                             Dim dat As DateTime = DateTime.Now
                             If dat = DateTime.MinValue Then
                                 Throw New ArgumentException("The clock is not working.")
                             End If

                             If dat.Hour > 17 Then
                                 Return "evening"
                             Else If dat.Hour > 12 Then
                                 Return "afternoon"
                             Else
                                 Return "morning"
                             End If
                         End Function)
        Dim c = t.ContinueWith(Sub(antecedent)
                                   If t.Status = TaskStatus.RanToCompletion Then
                                       Console.WriteLine("Good {0}!",
                                                         antecedent.Result)
                                       Console.WriteLine("And how are you this fine {0}?",
                                                         antecedent.Result)
                                   Else If t.Status = TaskStatus.Faulted Then
                                       Console.WriteLine(t.Exception.GetBaseException().Message)
                                   End If
                               End Sub)
    End Sub
End Module
' The example displays output like the following:
'       Good afternoon!
'       And how are you this fine afternoon?
```

取消延续

在以下情况下，延续的 Task.Status 属性将设置为 TaskStatus.Canceled：

- 延续引发 OperationCanceledException 以响应取消请求。就像任何任务一样，如果异常包含已传递到延续的相同标记，则会将其视为确认协作取消。
- 将 System.Threading.CancellationToken 属性为 IsCancellationRequested 的 `true` 传递到延续。在这种情况下，延续不会启动，并且将转换为 TaskStatus.Canceled 状态。
- 延续由于其 TaskContinuationOptions 参数设置的条件未得到满足而从不运行。例如，如果前面的任务进入 TaskStatus.Faulted 状态，则该任务被传递了 TaskContinuationOptions.NotOnFaulted 选项的延续将不会运行，而是将转换为 Canceled 状态。

如果一项任务及其延续表示同一逻辑操作的两个部分，则可以将相同的取消标记传递到这两个任务，如下面的示例所示。它包含的前面的任务可生成由可被 33 的整数组成的列表，并将该列表传递给延续。而延续反过来显示该列表。前面的任务和延续任务都将定期以随机间隔暂停。此外，System.Threading.Timer 对象用于在五秒的超时间隔后执行 `Elapsed` 方法。此示例调用 CancellationTokenSource.Cancel 方法，从而将导致当前正在执行的任务调用 CancellationToken.ThrowIfCancellationRequested 方法。是否在前面的任务或其延续正在执行时调用 CancellationTokenSource.Cancel 方法取决于随机生成的暂停的持续时间。如果已取消前面的任务，则延续将不会启动。如果未取消前面的任务，则仍然可以使用标记来取消延续。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

public class CancellationExample
```

```
public class CancellationExample
{
    static readonly Random s_random = new Random((int)DateTime.Now.Ticks);

    public static async Task Main()
    {
        using var cts = new CancellationTokenSource();
        CancellationToken token = cts.Token;
        var timer = new Timer(Elapsed, cts, 5000, Timeout.Infinite);

        var task = Task.Run(
            async () =>
            {
                var product33 = new List<int>();
                for (int index = 1; index < short.MaxValue; index++)
                {
                    if (token.IsCancellationRequested)
                    {
                        Console.WriteLine("\nCancellation requested in antecedent...\n");
                        token.ThrowIfCancellationRequested();
                    }
                    if (index % 2000 == 0)
                    {
                        int delay = s_random.Next(16, 501);
                        await Task.Delay(delay);
                    }
                    if (index % 33 == 0)
                    {
                        product33.Add(index);
                    }
                }

                return product33.ToArray();
            }, token);

        Task<double> continuation = task.ContinueWith(
            async antecedent =>
            {
                Console.WriteLine("Multiples of 33:\n");
                int[] array = antecedent.Result;
                for (int index = 0; index < array.Length; index++)
                {
                    if (token.IsCancellationRequested)
                    {
                        Console.WriteLine("\nCancellation requested in continuation...\n");
                        token.ThrowIfCancellationRequested();
                    }
                    if (index % 100 == 0)
                    {
                        int delay = s_random.Next(16, 251);
                        await Task.Delay(delay);
                    }

                    Console.Write($"{array[index]:N0}{(index != array.Length - 1 ? ", " : "")}");

                    if (Console.CursorLeft >= 74)
                    {
                        Console.WriteLine();
                    }
                }
                Console.WriteLine();
                return array.Average();
            }, token).Unwrap();

        try
        {
            await task;
            double result = await continuation;
        }
        catch (Exception ex)
```

```
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }

        Console.WriteLine("\nAntecedent Status: {0}", task.Status);
        Console.WriteLine("Continuation Status: {0}", continuation.Status);
    }

    static void Elapsed(object state)
    {
        if (state is CancellationTokenSource cts)
        {
            cts.Cancel();
            Console.WriteLine("\nCancellation request issued...\n");
        }
    }
}
// The example displays the similar output:
//     Multiples of 33:
//
//     33, 66, 99, 132, 165, 198, 231, 264, 297, 330, 363, 396, 429, 462, 495, 528,
//     561, 594, 627, 660, 693, 726, 759, 792, 825, 858, 891, 924, 957, 990, 1,023,
//     1,056, 1,089, 1,122, 1,155, 1,188, 1,221, 1,254, 1,287, 1,320, 1,353, 1,386,
//     1,419, 1,452, 1,485, 1,518, 1,551, 1,584, 1,617, 1,650, 1,683, 1,716, 1,749,
//     1,782, 1,815, 1,848, 1,881, 1,914, 1,947, 1,980, 2,013, 2,046, 2,079, 2,112,
//     2,145, 2,178, 2,211, 2,244, 2,277, 2,310, 2,343, 2,376, 2,409, 2,442, 2,475,
//     2,508, 2,541, 2,574, 2,607, 2,640, 2,673, 2,706, 2,739, 2,772, 2,805, 2,838,
//     2,871, 2,904, 2,937, 2,970, 3,003, 3,036, 3,069, 3,102, 3,135, 3,168, 3,201,
//     3,234, 3,267, 3,300, 3,333, 3,366, 3,399, 3,432, 3,465, 3,498, 3,531, 3,564,
//     3,597, 3,630, 3,663, 3,696, 3,729, 3,762, 3,795, 3,828, 3,861, 3,894, 3,927,
//     3,960, 3,993, 4,026, 4,059, 4,092, 4,125, 4,158, 4,191, 4,224, 4,257, 4,290,
//     4,323, 4,356, 4,389, 4,422, 4,455, 4,488, 4,521, 4,554, 4,587, 4,620, 4,653,
//     4,686, 4,719, 4,752, 4,785, 4,818, 4,851, 4,884, 4,917, 4,950, 4,983, 5,016,
//     5,049, 5,082, 5,115, 5,148, 5,181, 5,214, 5,247, 5,280, 5,313, 5,346, 5,379,
//     5,412, 5,445, 5,478, 5,511, 5,544, 5,577, 5,610, 5,643, 5,676, 5,709, 5,742,
//     Cancellation request issued...
//
//     5,775,
//     Cancellation requested in continuation...
//
//     The operation was canceled.
//
//     Antecedent Status: RanToCompletion
//     Continuation Status: Canceled
```

```
Imports System.Collections.Generic
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim rnd As New Random()
        Dim lockObj As New Object()
        Dim cts As New CancellationTokenSource()
        Dim token As CancellationToken = cts.Token
        Dim timer As New Timer(AddressOf Elapsed, cts, 5000, Timeout.Infinite)

        Dim t = Task.Run(Function()
                             Dim product33 As New List(Of Integer)()
                             For ctr As Integer = 1 To Int16.MaxValue
                                 ' Check for cancellation.
                                 If token.IsCancellationRequested Then
                                     Console.WriteLine("\nCancellation requested in antecedent...\n")
                                     token.ThrowIfCancellationRequested()
                                 End If
                                 ' Introduce a delay.
                                 If ctr Mod 2000 = 0 Then
```

```
                                If ctr Mod 2000 = 0 Then
                                    Dim delay As Integer
                                    SyncLock lockObj
                                        delay = rnd.Next(16, 501)
                                    End SyncLock
                                    Thread.Sleep(delay)
                                End If

                                ' Determine if this is a multiple of 33.
                                If ctr Mod 33 = 0 Then product33.Add(ctr)
                            Next
                            Return product33.ToArray()
                        End Function, token)

        Dim continuation = t.ContinueWith(Sub(antecedent)
                                              Console.WriteLine("Multiples of 33:" + vbCrLf)
                                              Dim arr = antecedent.Result
                                              For ctr As Integer = 0 To arr.Length - 1
                                                  If token.IsCancellationRequested Then
                                                      Console.WriteLine("{0}Cancellation requested in
continuation...{0}",
                                                                        vbCrLf)
                                                      token.ThrowIfCancellationRequested()
                                                  End If

                                                  If ctr Mod 100 = 0 Then
                                                      Dim delay As Integer
                                                      SyncLock lockObj
                                                          delay = rnd.Next(16, 251)
                                                      End SyncLock
                                                      Thread.Sleep(delay)
                                                  End If
                                                  Console.Write("{0:N0}{1}", arr(ctr),
                                                                If(ctr <> arr.Length - 1, ", ", ""))
                                                  If Console.CursorLeft >= 74 Then Console.WriteLine()
                                              Next
                                              Console.WriteLine()
                                          End Sub, token)

        Try
            continuation.Wait()
        Catch e As AggregateException
            For Each ie In e.InnerExceptions
                Console.WriteLine("{0}: {1}", ie.GetType().Name,
                                  ie.Message)
            Next
        Finally
            cts.Dispose()
        End Try

        Console.WriteLine(vbCrLf + "Antecedent Status: {0}", t.Status)
        Console.WriteLine("Continuation Status: {0}", continuation.Status)
    End Sub

    Private Sub Elapsed(state As Object)
        Dim cts As CancellationTokenSource = TryCast(state, CancellationTokenSource)
        If cts Is Nothing Then return

        cts.Cancel()
        Console.WriteLine("{0}Cancellation request issued...{0}", vbCrLf)
    End Sub
End Module
' The example displays output like the following:
'    Multiples of 33:
'
'    33, 66, 99, 132, 165, 198, 231, 264, 297, 330, 363, 396, 429, 462, 495, 528,
'    561, 594, 627, 660, 693, 726, 759, 792, 825, 858, 891, 924, 957, 990, 1,023,
'    1,056, 1,089, 1,122, 1,155, 1,188, 1,221, 1,254, 1,287, 1,320, 1,353, 1,386,
'    1,419, 1,452, 1,485, 1,518, 1,551, 1,584, 1,617, 1,650, 1,683, 1,716, 1,749,
```

```
'    1,782, 1,815, 1,848, 1,881, 1,914, 1,947, 1,980, 2,013, 2,046, 2,079, 2,112,
'    2,145, 2,178, 2,211, 2,244, 2,277, 2,310, 2,343, 2,376, 2,409, 2,442, 2,475,
'    2,508, 2,541, 2,574, 2,607, 2,640, 2,673, 2,706, 2,739, 2,772, 2,805, 2,838,
'    2,871, 2,904, 2,937, 2,970, 3,003, 3,036, 3,069, 3,102, 3,135, 3,168, 3,201,
'    3,234, 3,267, 3,300, 3,333, 3,366, 3,399, 3,432, 3,465, 3,498, 3,531, 3,564,
'    3,597, 3,630, 3,663, 3,696, 3,729, 3,762, 3,795, 3,828, 3,861, 3,894, 3,927,
'    3,960, 3,993, 4,026, 4,059, 4,092, 4,125, 4,158, 4,191, 4,224, 4,257, 4,290,
'    4,323, 4,356, 4,389, 4,422, 4,455, 4,488, 4,521, 4,554, 4,587, 4,620, 4,653,
'    4,686, 4,719, 4,752, 4,785, 4,818, 4,851, 4,884, 4,917, 4,950, 4,983, 5,016,
'    5,049, 5,082, 5,115, 5,148, 5,181, 5,214, 5,247, 5,280, 5,313, 5,346, 5,379,
'    5,412, 5,445, 5,478, 5,511, 5,544, 5,577, 5,610, 5,643, 5,676, 5,709, 5,742,
'    5,775, 5,808, 5,841, 5,874, 5,907, 5,940, 5,973, 6,006, 6,039, 6,072, 6,105,
'    6,138, 6,171, 6,204, 6,237, 6,270, 6,303, 6,336, 6,369, 6,402, 6,435, 6,468,
'    6,501, 6,534, 6,567, 6,600, 6,633, 6,666, 6,699, 6,732, 6,765, 6,798, 6,831,
'    6,864, 6,897, 6,930, 6,963, 6,996, 7,029, 7,062, 7,095, 7,128, 7,161, 7,194,
'    7,227, 7,260, 7,293, 7,326, 7,359, 7,392, 7,425, 7,458, 7,491, 7,524, 7,557,
'    7,590, 7,623, 7,656, 7,689, 7,722, 7,755, 7,788, 7,821, 7,854, 7,887, 7,920,
'    7,953, 7,986, 8,019, 8,052, 8,085, 8,118, 8,151, 8,184, 8,217, 8,250, 8,283,
'    8,316, 8,349, 8,382, 8,415, 8,448, 8,481, 8,514, 8,547, 8,580, 8,613, 8,646,
'    8,679, 8,712, 8,745, 8,778, 8,811, 8,844, 8,877, 8,910, 8,943, 8,976, 9,009,
'    9,042, 9,075, 9,108, 9,141, 9,174, 9,207, 9,240, 9,273, 9,306, 9,339, 9,372,
'    9,405, 9,438, 9,471, 9,504, 9,537, 9,570, 9,603, 9,636, 9,669, 9,702, 9,735,
'    9,768, 9,801, 9,834, 9,867, 9,900, 9,933, 9,966, 9,999, 10,032, 10,065, 10,098,
'    10,131, 10,164, 10,197, 10,230, 10,263, 10,296, 10,329, 10,362, 10,395, 10,428,
'    10,461, 10,494, 10,527, 10,560, 10,593, 10,626, 10,659, 10,692, 10,725, 10,758,
'    10,791, 10,824, 10,857, 10,890, 10,923, 10,956, 10,989, 11,022, 11,055, 11,088,
'    11,121, 11,154, 11,187, 11,220, 11,253, 11,286, 11,319, 11,352, 11,385, 11,418,
'    11,451, 11,484, 11,517, 11,550, 11,583, 11,616, 11,649, 11,682, 11,715, 11,748,
'    11,781, 11,814, 11,847, 11,880, 11,913, 11,946, 11,979, 12,012, 12,045, 12,078,
'    12,111, 12,144, 12,177, 12,210, 12,243, 12,276, 12,309, 12,342, 12,375, 12,408,
'    12,441, 12,474, 12,507, 12,540, 12,573, 12,606, 12,639, 12,672, 12,705, 12,738,
'    12,771, 12,804, 12,837, 12,870, 12,903, 12,936, 12,969, 13,002, 13,035, 13,068,
'    13,101, 13,134, 13,167, 13,200, 13,233, 13,266,
'    Cancellation requested in continuation...
'
'
'    Cancellation request issued...
'
'    TaskCanceledException: A task was canceled.
'
'    Antecedent Status: RanToCompletion
'    Continuation Status: Canceled
```

通过在创建延续时指定 TaskContinuationOptions.NotOnCanceled 选项，在不向延续提供取消标记而取消其前面的任务的情况下，你也可阻止延续的执行。下面是一个简单的示例。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class CancellationTwoExample
{
    public static async Task Main()
    {
        using var cts = new CancellationTokenSource();
        CancellationToken token = cts.Token;
        cts.Cancel();

        var task = Task.FromCanceled(token);
        Task continuation =
            task.ContinueWith(
                antecedent => Console.WriteLine("The continuation is running."),
                TaskContinuationOptions.NotOnCanceled);

        try
        {
            await task;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"{ex.GetType().Name}: {ex.Message}");
            Console.WriteLine();
        }

        Console.WriteLine($"Task {task.Id}: {task.Status:G}");
        Console.WriteLine($"Task {continuation.Id}: {continuation.Status:G}");
    }
}
// The example displays the similar output:
//       TaskCanceledException: A task was canceled.
//
//       Task 1: Canceled
//       Task 2: Canceled
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim cts As New CancellationTokenSource()
        Dim token As CancellationToken = cts.Token
        cts.Cancel()

        Dim t As Task = Task.FromCanceled(token)
        Dim continuation As Task = t.ContinueWith(Sub(antecedent)
                                                      Console.WriteLine("The continuation is running.")
                                                  End Sub, TaskContinuationOptions.NotOnCanceled)
        Try
            t.Wait()
        Catch e As AggregateException
            For Each ie In e.InnerExceptions
                Console.WriteLine("{0}: {1}", ie.GetType().Name, ie.Message)
            Next
            Console.WriteLine()
        Finally
            cts.Dispose()
        End Try

        Console.WriteLine("Task {0}: {1:G}", t.Id, t.Status)
        Console.WriteLine("Task {0}: {1:G}", continuation.Id,
                          continuation.Status)
    End Sub
End Module
' The example displays the following output:
'       TaskCanceledException: A task was canceled.
'
'       Task 1: Canceled
'       Task 2: Canceled
```

在延续转换为 Canceled 状态后，它可能会影响后面的延续，具体情况取决于为这些延续指定的 TaskContinuationOptions 。

已释放的延续将不会启动。

延续和子任务

在前面的任务及其所有附加的子任务完成之前，延续将不会运行。延续不会等待分离的子任务完成。以下两个示例阐释了两个子任务，其中一个附加到创建了延续的前面的任务，另一个从创建了延续的前面的任务中分离。在下面的示例中，延续仅在所有子任务都完成后才会运行，并且多次运行该示例时，每次生成的输出相同。该示例通过调用 TaskFactory.StartNew 方法来启动前面的任务，因为在默认情况下， Task.Run 方法将创建一个默认任务创建选项为 TaskCreationOptions.DenyChildAttach的父任务。

```
using System;
using System.Threading.Tasks;

public class AttachedExample
{
    public static async Task Main()
    {
        await Task.Factory
                  .StartNew(
            () =>
            {
                Console.WriteLine($"Running antecedent task {Task.CurrentId}...");
                Console.WriteLine("Launching attached child tasks...");
                for (int ctr = 1; ctr <= 5; ctr++)
                {
                    int index = ctr;
                    Task.Factory.StartNew(async value =>
                    {
                        Console.WriteLine($"   Attached child task #{value} running");
                        await Task.Delay(1000);
                    }, index, TaskCreationOptions.AttachedToParent);
                }
                Console.WriteLine("Finished launching attached child tasks...");
            }).ContinueWith(
                antecedent =>
                    Console.WriteLine($"Executing continuation of Task {antecedent.Id}"));
    }
}
// The example displays the similar output:
//     Running antecedent task 1...
//     Launching attached child tasks...
//     Finished launching attached child tasks...
//        Attached child task #1 running
//        Attached child task #5 running
//        Attached child task #3 running
//        Attached child task #2 running
//        Attached child task #4 running
//     Executing continuation of Task 1
```

```
Imports System.Threading
Imports System.Threading.Tasks

Public Module Example
    Public Sub Main()
        Dim t = Task.Factory.StartNew(Sub()
                                          Console.WriteLine("Running antecedent task {0}...",
                                                            Task.CurrentId)
                                          Console.WriteLine("Launching attached child tasks...")
                                          For ctr As Integer = 1 To 5
                                              Dim index As Integer = ctr
                                              Task.Factory.StartNew(Sub(value)
                                                                        Console.WriteLine("   Attached child
task #{0} running",
                                                                                          value)
                                                                        Thread.Sleep(1000)
                                                                    End Sub, index,
TaskCreationOptions.AttachedToParent)
                                          Next
                                          Console.WriteLine("Finished launching attached child tasks...")
                                      End Sub)
        Dim continuation = t.ContinueWith(Sub(antecedent)
                                              Console.WriteLine("Executing continuation of Task {0}",
                                                                antecedent.Id)
                                          End Sub)
        continuation.Wait()
    End Sub
End Module
' The example displays the following output:
'       Running antecedent task 1...
'       Launching attached child tasks...
'       Finished launching attached child tasks...
'          Attached child task #5 running
'          Attached child task #1 running
'          Attached child task #2 running
'          Attached child task #3 running
'          Attached child task #4 running
'       Executing continuation of Task 1
```

但是，如果子任务与前面的任务分离，则前面的任务任务一旦终止，延续就将立即开始运行，而无论子任务的状态。因此，多次运行下面的示例可能生成可变输出，具体取决于任务计划程序处理每个子任务的方式。

```
using System;
using System.Threading.Tasks;

public class DetachedExample
{
    public static async Task Main()
    {
        Task task =
            Task.Factory.StartNew(
                () =>
                {
                    Console.WriteLine($"Running antecedent task {Task.CurrentId}...");
                    Console.WriteLine("Launching attached child tasks...");
                    for (int ctr = 1; ctr <= 5; ctr++)
                    {
                        int index = ctr;
                        Task.Factory.StartNew(
                            async value =>
                            {
                                Console.WriteLine($"   Attached child task #{value} running");
                                await Task.Delay(1000);
                            }, index);
                    }
                    Console.WriteLine("Finished launching detached child tasks...");
                }, TaskCreationOptions.DenyChildAttach);

        Task continuation =
            task.ContinueWith(
                antecedent =>
                    Console.WriteLine($"Executing continuation of Task {antecedent.Id}"));

        await continuation;

        Console.ReadLine();
    }
}
// The example displays the similar output:
//     Running antecedent task 1...
//     Launching attached child tasks...
//     Finished launching detached child tasks...
//     Executing continuation of Task 1
//        Attached child task #1 running
//        Attached child task #5 running
//        Attached child task #2 running
//        Attached child task #3 running
//        Attached child task #4 running
```

```
Imports System.Threading
Imports System.Threading.Tasks

Public Module Example
    Public Sub Main()
        Dim t = Task.Factory.StartNew(Sub()
                                          Console.WriteLine("Running antecedent task {0}...",
                                                            Task.CurrentId)
                                          Console.WriteLine("Launching attached child tasks...")
                                          For ctr As Integer = 1 To 5
                                              Dim index As Integer = ctr
                                              Task.Factory.StartNew(Sub(value)
                                                                        Console.WriteLine("   Attached child
task #{0} running",
                                                                                          value)
                                                                        Thread.Sleep(1000)
                                                                    End Sub, index)
                                          Next
                                          Console.WriteLine("Finished launching detached child tasks...")
                                      End Sub, TaskCreationOptions.DenyChildAttach)
        Dim continuation = t.ContinueWith(Sub(antecedent)
                                              Console.WriteLine("Executing continuation of Task {0}",
                                                                antecedent.Id)
                                          End Sub)
        continuation.Wait()
    End Sub
End Module
' The example displays output like the following:
'       Running antecedent task 1...
'       Launching attached child tasks...
'       Finished launching detached child tasks...
'          Attached child task #1 running
'          Attached child task #2 running
'          Attached child task #5 running
'          Attached child task #3 running
'       Executing continuation of Task 1
'          Attached child task #4 running
```

前面的任务的最终状态取决于任何附加的子任务的最终状态。分离的子任务的状态不影响父级。有关详细信息，请参阅附加和分离的子任务。

将状态与延续关联

可以将任意状态与任务延续关联。ContinueWith 方法提供重载版本，每个重载版本都带有一个表示延续状态的 Object 值。可以之后通过使用 Task.AsyncState 属性访问此状态对象。如果未提供值，则此状态对象为 `null` 。

将使用 异步编程模型 (APM) 的现有代码转换为使用 TPL 时，延续状态非常有用。在 APM 中，通常在 Begin*Method* 方法中提供对象状态，并在之后通过使用 IAsyncResult.AsyncState 属性访问该状态。通过使用 ContinueWith 方法，你可在将使用 APM 的代码转换为使用 TPL 时保留此状态。

在 Visual Studio 调试器中处理 Task 对象时，延续状态也非常有用。例如，在“并行任务”窗口中，“任务”列显示每个任务的状态对象的字符串表示形式。有关“并行任务”窗口的详细信息，请参阅使用并行任务窗口。

下面的示例演示如何使用延续状态。它将创建一个延续任务链。每个任务都将为 DateTime 方法的 `state` 参数提供当前时间，即一个 ContinueWith 对象。每个 DateTime 对象都表示创建延续任务的时间。每个任务都将生成第二个 DateTime 对象作为其结果，该对象表示任务的完成时间。所有任务都完成后，本示例将显示每个延续任务的创建时间和完成时间。

```
using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

class ContinuationStateExample
{
    static DateTime DoWork()
    {
        Thread.Sleep(2000);

        return DateTime.Now;
    }

    static async Task Main()
    {
        Task<DateTime> task = Task.Run(() => DoWork());

        var continuations = new List<Task<DateTime>>();
        for (int i = 0; i < 5; i++)
        {
            task = task.ContinueWith((antecedent, _) => DoWork(), DateTime.Now);
            continuations.Add(task);
        }

        await task;

        foreach (Task<DateTime> continuation in continuations)
        {
            DateTime start = (DateTime)continuation.AsyncState;
            DateTime end = continuation.Result;

            Console.WriteLine($"Task was created at {start.TimeOfDay} and finished at {end.TimeOfDay}.");
        }

        Console.ReadLine();
    }
}
// The example displays the similar output:
//     Task was created at 10:56:21.1561762 and finished at 10:56:25.1672062.
//     Task was created at 10:56:21.1610677 and finished at 10:56:27.1707646.
//     Task was created at 10:56:21.1610677 and finished at 10:56:29.1743230.
//     Task was created at 10:56:21.1610677 and finished at 10:56:31.1779883.
//     Task was created at 10:56:21.1610677 and finished at 10:56:33.1837083.
```

```
Imports System.Collections.Generic
Imports System.Threading
Imports System.Threading.Tasks

' Demonstrates how to associate state with task continuations.
Public Module ContinuationState
    ' Simluates a lengthy operation and returns the time at which
    ' the operation completed.
    Public Function DoWork() As Date
        ' Simulate work by suspending the current thread
        ' for two seconds.
        Thread.Sleep(2000)

        ' Return the current time.
        Return Date.Now
    End Function

    Public Sub Main()
        ' Start a root task that performs work.
        Dim t As Task(Of Date) = Task(Of Date).Run(Function() DoWork())

        ' Create a chain of continuation tasks, where each task is
        ' followed by another task that performs work.
        Dim continuations As New List(Of Task(Of DateTime))()
        For i As Integer = 0 To 4
            ' Provide the current time as the state of the continuation.
            t = t.ContinueWith(Function(antecedent, state) DoWork(), DateTime.Now)
            continuations.Add(t)
        Next

        ' Wait for the last task in the chain to complete.
        t.Wait()

        ' Display the creation time of each continuation (the state object)
        ' and the completion time (the result of that task) to the console.
        For Each continuation In continuations
            Dim start As DateTime = CDate(continuation.AsyncState)
            Dim [end] As DateTime = continuation.Result

            Console.WriteLine("Task was created at {0} and finished at {1}.",
               start.TimeOfDay, [end].TimeOfDay)
        Next
    End Sub
End Module
' The example displays output like the following:
'       Task was created at 10:56:21.1561762 and finished at 10:56:25.1672062.
'       Task was created at 10:56:21.1610677 and finished at 10:56:27.1707646.
'       Task was created at 10:56:21.1610677 and finished at 10:56:29.1743230.
'       Task was created at 10:56:21.1610677 and finished at 10:56:31.1779883.
'       Task was created at 10:56:21.1610677 and finished at 10:56:33.1837083.
```

返回 Task 类型的延续

有时，可能需要链接可返回 Task 类型的延续。这些任务称为嵌套任务，它们很常见。当父任务调用 Task<TResult>.ContinueWith，并提供作为任务返回的 `continuationFunction` 时，你可以调用 Unwrap 来创建表示 `<Task<Task<T>>>` 或 `Task(Of Task(Of T))` (Visual Basic) 的异步操作的代理任务。

下面的示例展示如何使用包装附加任务返回函数的延续。每个延续都可以进行解包，并公开已包装的内部任务。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class UnwrapExample
{
    public static async Task Main()
    {
        Task<int> taskOne = RemoteIncrement(0);
        Console.WriteLine("Started RemoteIncrement(0)");

        Task<int> taskTwo = RemoteIncrement(4)
            .ContinueWith(t => RemoteIncrement(t.Result))
            .Unwrap().ContinueWith(t => RemoteIncrement(t.Result))
            .Unwrap().ContinueWith(t => RemoteIncrement(t.Result))
            .Unwrap();

        Console.WriteLine("Started RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)");

        try
        {
            await taskOne;
            Console.WriteLine("Finished RemoteIncrement(0)");

            await taskTwo;
            Console.WriteLine("Finished RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)");
        }
        catch (Exception e)
        {
            Console.WriteLine($"A task has thrown the following (unexpected) exception:\n{e}");
        }
    }

    static Task<int> RemoteIncrement(int number) =>
        Task<int>.Factory.StartNew(
            obj =>
            {
                Thread.Sleep(1000);

                int x = (int)obj;
                Console.WriteLine("Thread={0}, Next={1}", Thread.CurrentThread.ManagedThreadId, ++x);
                return x;
            },
            number);
}

// The example displays the similar output:
//     Started RemoteIncrement(0)
//     Started RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)
//     Thread=4, Next=1
//     Finished RemoteIncrement(0)
//     Thread=5, Next=5
//     Thread=6, Next=6
//     Thread=6, Next=7
//     Thread=6, Next=8
//     Finished RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)
```

```
Imports System.Threading

Module UnwrapExample
    Sub Main()
        Dim taskOne As Task(Of Integer) = RemoteIncrement(0)
        Console.WriteLine("Started RemoteIncrement(0)")

        Dim taskTwo As Task(Of Integer) = RemoteIncrement(4).
            ContinueWith(Function(t) RemoteIncrement(t.Result)).
            Unwrap().ContinueWith(Function(t) RemoteIncrement(t.Result)).
            Unwrap().ContinueWith(Function(t) RemoteIncrement(t.Result)).
            Unwrap()

        Console.WriteLine("Started RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)")

        Try
            taskOne.Wait()
            Console.WriteLine("Finished RemoteIncrement(0)")

            taskTwo.Wait()
            Console.WriteLine("Finished RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)")
        Catch e As AggregateException
            Console.WriteLine($"A task has thrown the following (unexpected) exception:{vbLf}{e}")
        End Try
    End Sub

    Function RemoteIncrement(ByVal number As Integer) As Task(Of Integer)
        Return Task(Of Integer).Factory.StartNew(
            Function(obj)
                Thread.Sleep(1000)

                Dim x As Integer = CInt(obj)
                Console.WriteLine("Thread={0}, Next={1}", Thread.CurrentThread.ManagedThreadId,
Interlocked.Increment(x))
                Return x
            End Function, number)
    End Function
End Module

' The example displays the similar output:
'     Started RemoteIncrement(0)
'     Started RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)
'     Thread=4, Next=1
'     Finished RemoteIncrement(0)
'     Thread=5, Next=5
'     Thread=6, Next=6
'     Thread=6, Next=7
'     Thread=6, Next=8
'     Finished RemoteIncrement(...(RemoteIncrement(RemoteIncrement(4))...)
```

有关使用 Unwrap 的详细信息，请参阅如何：展开嵌套任务。

处理从延续中引发的异常

前面的任务与延续之间的关系不是父/子关系。由延续引发的异常不会传播到前面的任务。因此，请按在任何其他任务中处理异常的方式来处理由延续引发的异常，如下所示：

- 你可以使用 Wait、WaitAll或 WaitAny 法或其对应的泛型方法来等待延续。你可以在同一 `try` 语句中等待前面的任务及其延续，如下面的示例所示。

```
using System;
using System.Threading.Tasks;

public class ExceptionExample
{
    public static async Task Main()
    {
        Task<int> task = Task.Run(
            () =>
            {
                Console.WriteLine($"Executing task {Task.CurrentId}");
                return 54;
            });

        var continuation = task.ContinueWith(
            antecedent =>
            {
                Console.WriteLine($"Executing continuation task {Task.CurrentId}");
                Console.WriteLine($"Value from antecedent: {antecedent.Result}");

                throw new InvalidOperationException();
            });

        try
        {
            await task;
            await continuation;
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
// The example displays the similar output:
//       Executing task 1
//       Executing continuation task 2
//       Value from antecedent: 54
//       Operation is not valid due to the current state of the object.
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task(Of Integer).Run(Function()
                                             Console.WriteLine("Executing task {0}",
                                                               Task.CurrentId)
                                             Return 54
                                         End Function)
        Dim continuation = task1.ContinueWith(Sub(antecedent)
                                                  Console.WriteLine("Executing continuation task {0}",
                                                                    Task.CurrentId)
                                                  Console.WriteLine("Value from antecedent: {0}",
                                                                    antecedent.Result)
                                                  Throw New InvalidOperationException()
                                              End Sub)

        Try
            task1.Wait()
            continuation.Wait()
        Catch ae As AggregateException
            For Each ex In ae.InnerExceptions
                Console.WriteLine(ex.Message)
            Next
        End Try
    End Sub
End Module
' The example displays the following output:
'       Executing task 1
'       Executing continuation task 2
'       Value from antecedent: 54
'       Operation is not valid due to the current state of the object.
```

- 可以使用第二个延续来观察第一个延续的 Exception 属性。在下面的示例中，某个任务尝试从不存在的文件中进行读取。然后，延续将显示有关前面的任务中的异常的信息。

```
using System;
using System.IO;
using System.Linq;
using System.Threading.Tasks;

public class ExceptionTwoExample
{
    public static async Task Main()
    {
        var task = Task.Run(
            () =>
            {
                string fileText = File.ReadAllText(@"C:\NonexistentFile.txt");
                return fileText;
            });

        Task continuation = task.ContinueWith(
            antecedent =>
            {
                var fileNotFound =
                    antecedent.Exception
                        ?.InnerExceptions
                        ?.FirstOrDefault(e => e is FileNotFoundException) as FileNotFoundException;

                if (fileNotFound != null)
                {
                    Console.WriteLine(fileNotFound.Message);
                }
            }, TaskContinuationOptions.OnlyOnFaulted);

        await continuation;

        Console.ReadLine();
    }
}
// The example displays the following output:
//        Could not find file 'C:\NonexistentFile.txt'.
```

```
Imports System.IO
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim t = Task.Run(Function()
                             Dim s As String = File.ReadAllText("C:\NonexistentFile.txt")
                             Return s
                         End Function)

        Dim c = t.ContinueWith(Sub(antecedent)
                                   ' Get the antecedent's exception information.
                                   For Each ex In antecedent.Exception.InnerExceptions
                                       If TypeOf ex Is FileNotFoundException
                                           Console.WriteLine(ex.Message)
                                       End If
                                   Next
                               End Sub, TaskContinuationOptions.OnlyOnFaulted)

        c.Wait()
    End Sub
End Module
' The example displays the following output:
'       Could not find file 'C:\NonexistentFile.txt'.
```

因为使用 TaskContinuationOptions.OnlyOnFaulted 选项运行，所以仅当前面的任务中发生异常时才会执行延续，

并且可以假设前面的任务的 Exception 属性不为 `null`。如果无论前面的任务中是否引发了异常都将执行延续，则必须先检查前面的任务的 Exception 属性是否不为 `null`，然后再尝试处理该异常，如以下代码片段所示。

```
var fileNotFound =
    antecedent.Exception
        ?.InnerExceptions
        ?.FirstOrDefault(e => e is FileNotFoundException) as FileNotFoundException;

if (fileNotFound != null)
{
    Console.WriteLine(fileNotFound.Message);
}
```

```
' Determine whether an exception occurred.
If antecedent.Exception IsNot Nothing Then
    ' Get the antecedent's exception information.
    For Each ex In antecedent.Exception.InnerExceptions
        If TypeOf ex Is FileNotFoundException
            Console.WriteLine(ex.Message)
        End If
    Next
End If
```

有关详细信息，请参阅异常处理。

- 如果延续为附加子任务并且是通过使用 TaskContinuationOptions.AttachedToParent 选项创建的，则父级会将该延续的异常传播回调用线程，就像任何其他附加子级的情况一样。有关详细信息，请参阅附加和分离的子任务。

请参阅

- 任务并行库 (TPL)

已附加和已分离的子任务

2021/11/16 •

子任务(或嵌套任务)是在另一个任务(称为"父任务")的用户委托中创建的 System.Threading.Tasks.Task 实例。可以分离或附加子任务。分离的子任务 是独立于父级而执行的任务。附加的子任务 是使用 TaskCreationOptions.AttachedToParent 选项创建的嵌套任务，父级不显式或默认禁止附加任务。一个任务可以创建任意数量的附加和分离子任务，这仅受系统资源限制。

下表列出了两种子任务之间的基本差异。

ㄖ	ㄖㄖㄖㄖ	ㄖㄖㄖㄖ
父级将等待子任务完成。	No	是
父级将传播由子任务引发的异常。	No	是
父级的状态取决于子级的状态。	No	是

在大多数情况下，我们建议你使用分离子任务，因为它们与其他任务之间的关系不太复杂。这就是父任务内创建的任务会默认分离的原因，并且必须显式指定 TaskCreationOptions.AttachedToParent 选项来创建附加子任务。

分离子任务

尽管子任务是由父任务创建的，但在默认情况下，它独立于父任务。在以下示例中，父任务创建了一个简单的子任务。如果多次运行该示例的代码，你可能会注意到该示例的输出与所演示的输出不同，并且该输出可能在每次运行代码时，会发生更改。发生这种情况的原因是父任务和子任务彼此独立执行;子任务是一个分离任务。该示例仅等待父任务完成，并且子任务在控制台应用终止之前，可能无法执行或完成。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
         Console.WriteLine("Outer task executing.");

         var child = Task.Factory.StartNew(() => {
            Console.WriteLine("Nested task starting.");
            Thread.SpinWait(500000);
            Console.WriteLine("Nested task completing.");
         });
      });

      parent.Wait();
      Console.WriteLine("Outer has completed.");
   }
}
// The example produces output like the following:
//        Outer task executing.
//        Nested task starting.
//        Outer has completed.
//        Nested task completing.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim parent = Task.Factory.StartNew(Sub()
                                               Console.WriteLine("Outer task executing.")
                                               Dim child = Task.Factory.StartNew(Sub()

Console.WriteLine("Nested task starting.")
                                                                                     Thread.SpinWait(500000)

Console.WriteLine("Nested task completing.")
                                                                                 End Sub)
                                           End Sub)
        parent.Wait()
        Console.WriteLine("Outer task has completed.")
    End Sub
End Module
' The example produces output like the following:
'   Outer task executing.
'   Nested task starting.
'   Outer task has completed.
'   Nested task completing.
```

如果该子任务由 Task<TResult> 对象，而不是 Task 对象表示，则你可以通过访问子任务的 Task<TResult>.Result 属性，确保父任务将等待子任务完成，即使该子任务是一个分离子任务。Result 属性在其任务完成前会进行阻止，如以下示例所示。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Example
{
   static void Main()
   {
      var outer = Task<int>.Factory.StartNew(() => {
            Console.WriteLine("Outer task executing.");

            var nested = Task<int>.Factory.StartNew(() => {
                  Console.WriteLine("Nested task starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Nested task completing.");
                  return 42;
            });

            // Parent will wait for this detached child.
            return nested.Result;
      });

      Console.WriteLine("Outer has returned {0}.", outer.Result);
   }
}
// The example displays the following output:
//       Outer task executing.
//       Nested task starting.
//       Nested task completing.
//       Outer has returned 42.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim parent = Task(Of Integer).Factory.StartNew(Function()
                                                           Console.WriteLine("Outer task executing.")
                                                           Dim child = Task(Of
Integer).Factory.StartNew(Function()

Console.WriteLine("Nested task starting.")

Thread.SpinWait(5000000)

Console.WriteLine("Nested task completing.")

Return 42
                                                                                                    End
Function)
                                                           Return child.Result


                                                       End Function)
        Console.WriteLine("Outer has returned {0}", parent.Result)
    End Sub
End Module
' The example displays the following output:
'       Outer task executing.
'       Nested task starting.
'       Detached task completing.
'       Outer has returned 42
```

附加子任务

不同于分离子任务，附加子任务与父任务紧密同步。可以通过使用任务创建语句中的 TaskCreationOptions.AttachedToParent 选项，将之前示例中的分离子任务更改为附加子任务，如以下示例中所示。在此代码中，附加子任务会在父任务之前完成。因此，每次运行代码时，该示例的输出都是相同的。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Factory.StartNew(() => {
            Console.WriteLine("Parent task executing.");
            var child = Task.Factory.StartNew(() => {
                  Console.WriteLine("Attached child starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Attached child completing.");
            }, TaskCreationOptions.AttachedToParent);
      });
      parent.Wait();
      Console.WriteLine("Parent has completed.");
   }
}
// The example displays the following output:
//       Parent task executing.
//       Attached child starting.
//       Attached child completing.
//       Parent has completed.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim parent = Task.Factory.StartNew(Sub()
                                               Console.WriteLine("Parent task executing")
                                               Dim child = Task.Factory.StartNew(Sub()

Console.WriteLine("Attached child starting.")

Thread.SpinWait(5000000)

Console.WriteLine("Attached child completing.")
                                                                                   End Sub,
TaskCreationOptions.AttachedToParent)
                                           End Sub)
        parent.Wait()
        Console.WriteLine("Parent has completed.")
    End Sub
End Module
' The example displays the following output:
'       Parent task executing.
'       Attached child starting.
'       Attached child completing.
'       Parent has completed.
```

可以使用附加子任务，创建异步操作的紧密同步关系图。

但是，子任务仅在其父任务不会阻止附加子任务时，才可以附加到其父任务。通过在父任务类构造函数中指定 TaskCreationOptions.DenyChildAttach选项或 TaskFactory.StartNew 方法，父任务可以显式阻止子任务附加到其中。如果父任务是通过调用 Task.Run 方法而创建的，则可以隐式阻止子任务附加到其中。下面的示例阐释了这一点。这与上述示例相同，除了该父任务是通过调用 Task.Run(Action) 方法，而不是 TaskFactory.StartNew(Action) 方法创建的。因为子任务不能附加到其父任务，则该示例的输出是不可预知的。因为 Task.Run 重载的默认任务创建选项包括 TaskCreationOptions.DenyChildAttach，所以本示例在功能上等效于"分离子任务"部分中的第一个示例。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var parent = Task.Run(() => {
            Console.WriteLine("Parent task executing.");
            var child = Task.Factory.StartNew(() => {
                  Console.WriteLine("Attached child starting.");
                  Thread.SpinWait(5000000);
                  Console.WriteLine("Attached child completing.");
            }, TaskCreationOptions.AttachedToParent);
      });
      parent.Wait();
      Console.WriteLine("Parent has completed.");
   }
}
// The example displays output like the following:
//       Parent task executing.
//       Parent has completed.
//       Attached child starting.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim parent = Task.Run(Sub()
                                  Console.WriteLine("Parent task executing.")
                                  Dim child = Task.Factory.StartNew(Sub()
                                                                        Console.WriteLine("Attached child
starting.")
                                                                        Thread.SpinWait(5000000)
                                                                        Console.WriteLine("Attached child
completing.")
                                                                    End Sub,
TaskCreationOptions.AttachedToParent)
                              End Sub)
        parent.Wait()
        Console.WriteLine("Parent has completed.")
    End Sub
End Module
' The example displays output like the following:
'       Parent task executing.
'       Parent has completed.
'       Attached child starting.
```

子任务中的异常

如果分离子任务引发了异常，则该异常必须直接在父任务中进行观察和处理，正如任何非嵌套任务一样。如果附加子任务引发了异常，则该异常会自动传播到父任务，并返回到等待或尝试访问任务的 Task<TResult>.Result 属性的线程。因此，通过使用附加子任务，可以一次性处理调用线程上对 Task.Wait 的调用中的所有异常。有关详细信息，请参阅异常处理。

取消和子任务

任务取消需要彼此协作。也就是说，若要取消任务，则每个附加或分离的子任务必须监视取消标记的状态。如果想要通过使用一个取消请求来取消父任务及其所有子任务，则需要将作为参数的相同令牌传递到所有的任务，并在每个任务中提供逻辑，以对每个任务中的请求作出响应。有关详细信息，请参阅任务取消和如何：取消任务及其子任务。

当父任务取消时

如果父任务在其子任务开始前取消了自身，则子任务将永远不会开始。如果父任务在其子任务已开始后取消了自身，则子任务将完成运行，除非它自己具有取消逻辑。有关详细信息，请参阅任务取消。

当分离子任务取消时

如果分离子任务使用传递到父任务的相同标记取消自身，且父任务不会等待子任务，则不会传播异常，因为该异常将被视为良性协作取消。此行为与任何顶级任务的行为相同。

当附加子任务取消时

当附加子任务使用传递到其父任务的相同标记取消自身时，TaskCanceledException 将传播到 AggregateException 中的联接线程。必须等待父任务，以便你除了所有通过附加子任务的图形传播的错误异常之外，还可以处理所有良性异常。

有关详细信息，请参阅异常处理。

阻止子任务附加到其父任务

由子任务引发的未经处理的异常将传播到父任务中。可以使用此行为，从一个根任务而无需遍历任务树来观察所有子任务异常。但是，当父任务不需要其他代码的附件时，异常传播可能会产生问题。例如，设想下从 Task 对

象调用第三方库组件的应用。如果第三方库组件也创建一个 Task 对象，并指定 TaskCreationOptions.AttachedToParent 以将其附加到父任务中，则子任务中出现的任何未经处理的异常将会传播到父任务。这可能会导致主应用中出现意外行为。

若要防止子任务附加到其父任务，请在创建父任务 Task 或 Task<TResult> 对象时，指定 TaskCreationOptions.DenyChildAttach 选项。当某项任务尝试附加到其父任务，且其父任务指定了 TaskCreationOptions.DenyChildAttach 选项时，则子任务将不能附加到父任务，并且将像未指定 TaskCreationOptions.AttachedToParent 选项一样进行执行。

可能还想要防止子任务在没有及时完成时附加到其父任务。因为父任务只有在所有子任务完成后才会完成，所以长时间运行的子任务会使整个应用执行得非常缓慢。有关展示了如何通过防止子任务附加到父任务来提升应用性能的示例，请参阅如何：防止子任务附加到父任务。

另请参阅

- 并行编程
- 数据并行

任务取消

2021/11/16 •

System.Threading.Tasks.Task 和 System.Threading.Tasks.Task<TResult> 类支持通过使用取消标记进行取消。有关详细信息，请参阅托管线程中的取消。在 Task 类中，取消涉及用户委托间的协作，这表示可取消的操作和请求取消的代码。成功取消涉及调用 CancellationTokenSource.Cancel 方法的请求代码，以及及时终止操作的用户委托。可以使用以下选项之一终止操作：

- 简单地从委托中返回。在许多情况下，这样已足够；但是，采用这种方式取消的任务实例会转换为 TaskStatus.RanToCompletion 状态，而不是 TaskStatus.Canceled 状态。
- 引发 OperationCanceledException，并将其传递到在其上请求了取消的标记。完成此操作的首选方式是使用 ThrowIfCancellationRequested 方法。采用这种方式取消的任务会转换为 Canceled 状态，调用代码可使用该状态来验证任务是否响应了其取消请求。

下面的示例演示引发异常的任务取消的基本模式。请注意，标记将传递到用户委托和任务实例本身。

```
using System;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        var tokenSource2 = new CancellationTokenSource();
        CancellationToken ct = tokenSource2.Token;

        var task = Task.Run(() =>
        {
            // Were we already canceled?
            ct.ThrowIfCancellationRequested();

            bool moreToDo = true;
            while (moreToDo)
            {
                // Poll on this property if you have to do
                // other cleanup before throwing.
                if (ct.IsCancellationRequested)
                {
                    // Clean up here, then...
                    ct.ThrowIfCancellationRequested();
                }
            }
        }, tokenSource2.Token); // Pass same token to Task.Run.

        tokenSource2.Cancel();

        // Just continue on this thread, or await with try-catch:
        try
        {
            await task;
        }
        catch (OperationCanceledException e)
        {
            Console.WriteLine($"{nameof(OperationCanceledException)} thrown with message: {e.Message}");
        }
        finally
        {
            tokenSource2.Dispose();
        }

        Console.ReadKey();
    }
}
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Test
    Sub Main()
        Dim tokenSource2 As New CancellationTokenSource()
        Dim ct As CancellationToken = tokenSource2.Token

        Dim t2 = Task.Factory.StartNew(Sub()
                                           ' Were we already canceled?
                                           ct.ThrowIfCancellationRequested()

                                           Dim moreToDo As Boolean = True
                                           While moreToDo = True
                                               ' Poll on this property if you have to do
                                               ' other cleanup before throwing.
                                               If ct.IsCancellationRequested Then

                                                   ' Clean up here, then...
                                                   ct.ThrowIfCancellationRequested()
                                               End If

                                           End While
                                       End Sub _
        , tokenSource2.Token) ' Pass same token to StartNew.

        ' Cancel the task.
        tokenSource2.Cancel()

        ' Just continue on this thread, or Wait/WaitAll with try-catch:
        Try
            t2.Wait()

        Catch e As AggregateException

            For Each item In e.InnerExceptions
                Console.WriteLine(e.Message & " " & item.Message)
            Next
        Finally
            tokenSource2.Dispose()
        End Try

        Console.ReadKey()
    End Sub
End Module
```

有关更完整的示例，请参见如何：取消任务及其子级。

当任务实例观察到用户代码引发的 OperationCanceledException 时，它会将该异常的标记与其关联的标记（传递到创建任务的 API 的标记）进行比较。如果这两个标记相同，并且标记的 IsCancellationRequested 属性返回 true，则任务会将此解释为确认取消并转换为 Canceled 状态。如果您不使用 Wait 或 WaitAll 方法来等待任务，则任务只会将其状态设置为 Canceled。

如果你在等待转换为 Canceled 状态的任务，则会引发 System.Threading.Tasks.TaskCanceledException 异常（包装在 AggregateException 异常中）。请注意，此异常指示成功的取消，而不是有错误的情况。因此，任务的 Exception 属性返回 `null`。

如果标记的 IsCancellationRequested 属性返回 false，或者异常的标记与任务的标记不匹配，则会将 OperationCanceledException 按照普通的异常来处理，从而导致任务转换为 Faulted 状态。另外还要注意，其他异常的存在将也会导致任务转换为 Faulted 状态。您可以在 Status 属性中获取已完成任务的状态。

在请求取消操作之后，任务可能还可以继续处理一些项目。

请参阅

- 托管线程中的取消
- 如何：取消任务及其子级

异常处理（任务并行库）

2021/11/16 •

由在任务内部运行的用户代码引发的未处理异常会传播回调用线程，但本主题稍后部分介绍的某些情况除外。如果使用静态或实例 Task.Wait 方法之一，异常会传播，异常处理方法为将调用封闭到 `try` / `catch` 语句中。如果任务是所附加子任务的父级，或在等待多个任务，那么可能会引发多个异常。

为了将所有异常传播回调用线程，任务基础结构会将这些异常包装在 AggregateException 实例中。AggregateException 异常具有 InnerExceptions 属性，可枚举该属性来检查引发的所有原始异常，并单独处理（或不处理）每个异常。也可以使用 AggregateException.Handle 方法处理原始异常。

即使只引发了一个异常，仍会将该异常包装在 AggregateException 中，如以下示例所示。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      try
      {
          task1.Wait();
      }
      catch (AggregateException ae)
      {
          foreach (var e in ae.InnerExceptions) {
              // Handle the custom exception.
              if (e is CustomException) {
                  Console.WriteLine(e.Message);
              }
              // Rethrow any other exception.
              else {
                  throw e;
              }
          }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

        Try
            task1.Wait()
        Catch ae As AggregateException
            For Each ex In ae.InnerExceptions
                ' Handle the custom exception.
                If TypeOf ex Is CustomException Then
                    Console.WriteLine(ex.Message)
                    ' Rethrow any other exception.
                Else
                    Throw ex
                End If
            Next
        End Try
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

可以通过只捕获 AggregateException 而不观察任何内部异常来避免未处理的异常。但是，我们建议你不要这样做，因为这样相当于在非并行情况下捕获基 Exception 类型。捕获异常而不采取具体措施从中恢复可能会使程序进入不确定状态。

如果不想调用 Task.Wait 方法来等待任务完成，也可以通过任务的 Exception 属性检索 AggregateException 异常，如下面的示例所示。有关详细信息，请参阅本主题中的通过使用 Task.Exception 属性观察异常部分。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      while(! task1.IsCompleted) {}

      if (task1.Status == TaskStatus.Faulted) {
          foreach (var e in task1.Exception.InnerExceptions) {
              // Handle the custom exception.
              if (e is CustomException) {
                  Console.WriteLine(e.Message);
              }
              // Rethrow any other exception.
              else {
                  throw e;
              }
          }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

        While Not task1.IsCompleted
        End While

        If task1.Status = TaskStatus.Faulted Then
            For Each ex In task1.Exception.InnerExceptions
                ' Handle the custom exception.
                If TypeOf ex Is CustomException Then
                    Console.WriteLine(ex.Message)
                    ' Rethrow any other exception.
                Else
                    Throw ex
                End If
            Next
        End If
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

如果不等待传播异常的任务，或要访问其 Exception 属性，则会在对该任务进行垃圾回收时根据 .NET 异常策略提升异常。

如果允许异常向上冒泡回到联接线程，则一个任务也许可以在引发异常后继续处理一些项。

> **NOTE**
> 某些情况下，当启用“仅我的代码”后，Visual Studio 会在引发异常的行中断运行并显示一条错误消息，该消息显示“用户代码未处理异常”。此错误是良性的。可以按 F5 继续并查看在这些示例中演示的异常处理行为。若要阻止 Visual Studio 在出现第一个错误时中断运行，只需在“工具”->“选项”->“调试”->“常规” 下取消选中“启用‘仅我的代码’” 复选框即可。

附加子任务和嵌套 AggregateExceptions

如果某个任务具有引发异常的附加子任务，则会在将该异常传播到父任务之前将其包装在 AggregateException 中，父任务将该异常包装在自己的 AggregateException 中，然后再将其传播回调用线程。在这种情况下，在 Task.Wait、WaitAny、或 WaitAll 方法处捕获的 AggregateException 异常的 InnerExceptions 属性包含一个或多个 AggregateException 实例，而不包含导致错误的原始异常。为了避免必须循环访问嵌套 AggregateException 异常，可以使用 Flatten 方法删除所有嵌套 AggregateException 异常，以便 AggregateException.InnerExceptions 属性包含原始异常。在下面的示例中，嵌套 AggregateException 实例已经平展，并且仅在一个循环中处理。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Factory.StartNew(() => {
                     var child1 = Task.Factory.StartNew(() => {
                        var child2 = Task.Factory.StartNew(() => {
                            // This exception is nested inside three AggregateExceptions.
                            throw new CustomException("Attached child2 faulted.");
                        }, TaskCreationOptions.AttachedToParent);

                        // This exception is nested inside two AggregateExceptions.
                        throw new CustomException("Attached child1 faulted.");
                     }, TaskCreationOptions.AttachedToParent);
      });

      try {
         task1.Wait();
      }
      catch (AggregateException ae) {
         foreach (var e in ae.Flatten().InnerExceptions) {
            if (e is CustomException) {
               Console.WriteLine(e.Message);
            }
            else {
               throw;
            }
         }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//    Attached child1 faulted.
//    Attached child2 faulted.
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Factory.StartNew(Sub()
                                              Dim child1 = Task.Factory.StartNew(Sub()
                                                                                     Dim child2 =
Task.Factory.StartNew(Sub()

Throw New CustomException("Attached child2 faulted.")

End Sub,

TaskCreationOptions.AttachedToParent)
                                                                                         Throw New
CustomException("Attached child1 faulted.")
                                                                                 End Sub,

TaskCreationOptions.AttachedToParent)
                                          End Sub)

        Try
            task1.Wait()
        Catch ae As AggregateException
            For Each ex In ae.Flatten().InnerExceptions
                If TypeOf ex Is CustomException Then
                    Console.WriteLine(ex.Message)
                Else
                    Throw
                End If
            Next
        End Try
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays the following output:
'       Attached child1 faulted.
'       Attached child2 faulted.
```

还可以使用 AggregateException.Flatten 方法，通过多个任务在一个 AggregateException 实例中抛出的多个 AggregateException 实例重新抛出内部异常，如下面的示例所示。

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Threading.Tasks;
public class Example
{
   public static void Main()
   {
        try {
            ExecuteTasks();
        }
        catch (AggregateException ae) {
            foreach (var e in ae.InnerExceptions) {
                Console.WriteLine("{0}:\n   {1}", e.GetType().Name, e.Message);
            }
        }
   }

   static void ExecuteTasks()
   {
        // Assume this is a user-entered String.
        String path = @"C:\";
        List<Task> tasks = new List<Task>();

        tasks.Add(Task.Run(() => {
                             // This should throw an UnauthorizedAccessException.
                              return Directory.GetFiles(path, "*.txt",
                                                        SearchOption.AllDirectories);
                          }));

        tasks.Add(Task.Run(() => {
                              if (path == @"C:\")
                                 throw new ArgumentException("The system root is not a valid path.");
                              return new String[] { ".txt", ".dll", ".exe", ".bin", ".dat" };
                          }));

        tasks.Add(Task.Run(() => {
                               throw new NotImplementedException("This operation has not been
implemented.");
                          }));

        try {
            Task.WaitAll(tasks.ToArray());
        }
        catch (AggregateException ae) {
            throw ae.Flatten();
        }
    }
}
// The example displays the following output:
//       UnauthorizedAccessException:
//          Access to the path 'C:\Documents and Settings' is denied.
//       ArgumentException:
//          The system root is not a valid path.
//       NotImplementedException:
//          This operation has not been implemented.
```

```
Imports System.Collections.Generic
Imports System.IO
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Try
            ExecuteTasks()
        Catch ae As AggregateException
            For Each e In ae.InnerExceptions
                Console.WriteLine("{0}:{2}   {1}", e.GetType().Name, e.Message,
                                  vbCrLf)
            Next
        End Try
    End Sub

    Sub ExecuteTasks()
        ' Assume this is a user-entered String.
        Dim path = "C:\"
        Dim tasks As New List(Of Task)

        tasks.Add(Task.Run(Function()
                               ' This should throw an UnauthorizedAccessException.
                               Return Directory.GetFiles(path, "*.txt",
                                                         SearchOption.AllDirectories)
                           End Function))

        tasks.Add(Task.Run(Function()
                               If path = "C:\" Then
                                   Throw New ArgumentException("The system root is not a valid path.")
                               End If
                               Return {".txt", ".dll", ".exe", ".bin", ".dat"}
                           End Function))

        tasks.Add(Task.Run(Sub()
                               Throw New NotImplementedException("This operation has not been implemented.")
                           End Sub))

        Try
            Task.WaitAll(tasks.ToArray)
        Catch ae As AggregateException
            Throw ae.Flatten()
        End Try
    End Sub
End Module
' The example displays the following output:
'       UnauthorizedAccessException:
'          Access to the path 'C:\Documents and Settings' is denied.
'       ArgumentException:
'          The system root is not a valid path.
'       NotImplementedException:
'          This operation has not been implemented.
```

分离子任务中的异常

默认情况下，子任务在创建时处于分离状态。必须在直接父任务中处理或重新引发从分离任务引发的异常；将不会采用与附加子任务传播回异常相同的方式将这些异常传播回调用线程。最顶层的父级可以手动重新引发分离子级中的异常，以使其包装在 AggregateException 中并传播回调用线程。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run(() => {
                      var nested1 = Task.Run(() => {
                                          throw new CustomException("Detached child task faulted.");
                                    });

          // Here the exception will be escalated back to the calling thread.
          // We could use try/catch here to prevent that.
          nested1.Wait();
      });

      try {
         task1.Wait();
      }
      catch (AggregateException ae) {
         foreach (var e in ae.Flatten().InnerExceptions) {
            if (e is CustomException) {
               Console.WriteLine(e.Message);
            }
         }
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//    Detached child task faulted.
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Run(Sub()
                                 Dim nestedTask1 = Task.Run(Sub()
                                                                Throw New CustomException("Detached child
task faulted.")
                                                            End Sub)
                                 ' Here the exception will be escalated back to joining thread.
                                 ' We could use try/catch here to prevent that.
                                 nestedTask1.Wait()
                             End Sub)

        Try
            task1.Wait()
        Catch ae As AggregateException
            For Each ex In ae.Flatten().InnerExceptions
                If TypeOf ex Is CustomException Then
                    ' Recover from the exception. Here we just
                    ' print the message for demonstration purposes.
                    Console.WriteLine(ex.Message)
                End If
            Next
        End Try
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays the following output:
'       Detached child task faulted.
```

即使使用延续观察子任务中的异常，该异常仍然必须由父任务观察。

指示协作取消的异常

在任务中的用户代码响应取消请求时，正确的过程是引发传入在其上传达请求的取消标记中的 OperationCanceledException 。在尝试传播异常之前，任务实例会将异常中的标记与创建异常时传递给异常的标记进行比较。如果标记相同，则任务会传播包装在 TaskCanceledException 中的 AggregateException，并且将可以在检查内部异常时看到它。但是，如果调用线程未在等待任务，则将不会传播此特定异常。有关详细信息，请参阅任务取消。

```
var tokenSource = new CancellationTokenSource();
var token = tokenSource.Token;

var task1 = Task.Factory.StartNew(() =>
{
    CancellationToken ct = token;
    while (someCondition)
    {
        // Do some work...
        Thread.SpinWait(50000);
        ct.ThrowIfCancellationRequested();
    }
},
token);

// No waiting required.
tokenSource.Dispose();
```

```
Dim someCondition As Boolean = True
Dim tokenSource = New CancellationTokenSource()
Dim token = tokenSource.Token

Dim task1 = Task.Factory.StartNew(Sub()
                                      Dim ct As CancellationToken = token
                                      While someCondition = True
                                          ' Do some work...
                                          Thread.SpinWait(500000)
                                          ct.ThrowIfCancellationRequested()
                                      End While
                                  End Sub,
                                  token)
```

使用 Handle 方法筛选内部异常

可以使用 AggregateException.Handle 方法，筛选掉可视为“已处理”的异常，而无需进一步使用任何逻辑。在提供给 AggregateException.Handle(Func<Exception,Boolean>) 方法的用户委托中，可以检查异常类型及其 Message 属性，或可便于确定异常是否为良性的其他任何信息。在 AggregateException.Handle 方法返回结果后，便会立即在新实例 AggregateException 中重新抛出委托针对其返回 `false` 的任何异常。

下面的示例在功能上相当于本主题中的第一个示例（检查 AggregateException.InnerExceptions 集合中的所有异常）。相反，此异常处理程序对每个异常调用 AggregateException.Handle 方法对象，并仅重新抛出不是 `CustomException` 实例的异常。

```
using System;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run( () => { throw new CustomException("This exception is expected!"); } );

      try {
          task1.Wait();
      }
      catch (AggregateException ae)
      {
         // Call the Handle method to handle the custom exception,
         // otherwise rethrow the exception.
         ae.Handle(ex => { if (ex is CustomException)
                              Console.WriteLine(ex.Message);
                           return ex is CustomException;
                         });
      }
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays the following output:
//        This exception is expected!
```

```
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Run(Sub() Throw New CustomException("This exception is expected!"))

        Try
            task1.Wait()
        Catch ae As AggregateException
            ' Call the Handle method to handle the custom exception,
            ' otherwise rethrow the exception.
            ae.Handle(Function(e)
                          If TypeOf e Is CustomException Then
                              Console.WriteLine(e.Message)
                          End If
                          Return TypeOf e Is CustomException
                      End Function)
        End Try
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays the following output:
'       This exception is expected!
```

下面是更完整的示例，在枚举文件时，使用 AggregateException.Handle 方法提供 UnauthorizedAccessException 异常的特殊处理。

```
using System;
using System.IO;
using System.Threading.Tasks;

public class Example
{
    public static void Main()
    {
        // This should throw an UnauthorizedAccessException.
       try {
           var files = GetAllFiles(@"C:\");
           if (files != null)
              foreach (var file in files)
                 Console.WriteLine(file);
        }
        catch (AggregateException ae) {
           foreach (var ex in ae.InnerExceptions)
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message);
        }
        Console.WriteLine();

        // This should throw an ArgumentException.
        try {
           foreach (var s in GetAllFiles(""))
              Console.WriteLine(s);
        }
        catch (AggregateException ae) {
           foreach (var ex in ae.InnerExceptions)
               Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message);
        }
    }

    static string[] GetAllFiles(string path)
    {
       var task1 = Task.Run( () => Directory.GetFiles(path, "*.txt",
                                                      SearchOption.AllDirectories));

       try {
          return task1.Result;
       }
       catch (AggregateException ae) {
          ae.Handle( x => { // Handle an UnauthorizedAccessException
                            if (x is UnauthorizedAccessException) {
                                Console.WriteLine("You do not have permission to access all folders in this
path.");
                                Console.WriteLine("See your network administrator or try another path.");
                            }
                            return x is UnauthorizedAccessException;
                          });
          return Array.Empty<String>();
       }
   }
}
// The example displays the following output:
//       You do not have permission to access all folders in this path.
//       See your network administrator or try another path.
//
//       ArgumentException: The path is not of a legal form.
```

```
Imports System.IO
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        ' This should throw an UnauthorizedAccessException.
        Try
            Dim files = GetAllFiles("C:\")
            If files IsNot Nothing Then
                For Each file In files
                    Console.WriteLine(file)
                Next
            End If
        Catch ae As AggregateException
            For Each ex In ae.InnerExceptions
                Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message)
            Next
        End Try
        Console.WriteLine()

        ' This should throw an ArgumentException.
        Try
            For Each s In GetAllFiles("")
                Console.WriteLine(s)
            Next
        Catch ae As AggregateException
            For Each ex In ae.InnerExceptions
                Console.WriteLine("{0}: {1}", ex.GetType().Name, ex.Message)
            Next
        End Try
        Console.WriteLine()
    End Sub

    Function GetAllFiles(ByVal path As String) As String()
        Dim task1 = Task.Run(Function()
                                 Return Directory.GetFiles(path, "*.txt",
                                                           SearchOption.AllDirectories)
                             End Function)
        Try
            Return task1.Result
        Catch ae As AggregateException
            ae.Handle(Function(x)
                          ' Handle an UnauthorizedAccessException
                          If TypeOf x Is UnauthorizedAccessException Then
                              Console.WriteLine("You do not have permission to access all folders in this
path.")
                              Console.WriteLine("See your network administrator or try another path.")
                          End If
                          Return TypeOf x Is UnauthorizedAccessException
                      End Function)
        End Try
        Return Array.Empty(Of String)()
    End Function
End Module
' The example displays the following output:
'       You do not have permission to access all folders in this path.
'       See your network administrator or try another path.
'
'       ArgumentException: The path is not of a legal form.
```

通过使用 Task.Exception 属性观察异常

如果任务完成时的状态为 TaskStatus.Faulted，可以检查它的 Exception 属性，以发现是哪个异常导致错误发生。观察 Exception 属性的一个好方法是使用仅在前面的任务出错时才运行的延续，如以下示例所示。

```
using System;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
   public static void Main()
   {
      var task1 = Task.Run(() =>
                           { throw new CustomException("task1 faulted.");
      }).ContinueWith( t => { Console.WriteLine("{0}: {1}",
                                                t.Exception.InnerException.GetType().Name,
                                                t.Exception.InnerException.Message);
                            }, TaskContinuationOptions.OnlyOnFaulted);
      Thread.Sleep(500);
   }
}

public class CustomException : Exception
{
   public CustomException(String message) : base(message)
   {}
}
// The example displays output like the following:
//        CustomException: task1 faulted.
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Public Sub Main()
        Dim task1 = Task.Factory.StartNew(Sub()
                                              Throw New CustomException("task1 faulted.")
                                          End Sub).
                    ContinueWith(Sub(t)
                                     Console.WriteLine("{0}: {1}",
                                                       t.Exception.InnerException.GetType().Name,
                                                       t.Exception.InnerException.Message)
                                 End Sub, TaskContinuationOptions.OnlyOnFaulted)

        Thread.Sleep(500)
    End Sub
End Module

Class CustomException : Inherits Exception
    Public Sub New(s As String)
        MyBase.New(s)
    End Sub
End Class
' The example displays output like the following:
'       CustomException: task1 faulted.
```

在有意义的应用程序中，延续委托可能会记录有关异常的详细信息，并可能生成新任务以从异常中恢复。如果任务出错，以下表达式将引发异常：

- `await task`
- `task.Wait()`
- `task.Result`
- `task.GetAwaiter().GetResult()`

使用 `try-catch` 语句来处理和观察引发的异常。或者，通过访问 Task.Exception 属性来观察异常。

UnobservedTaskException 事件

在某些情况下(例如承载不受信任的插件时)，良性异常可能比较普遍，因此很难以手动方式观察到所有异常。在这些情况下，可以处理 TaskScheduler.UnobservedTaskException 事件。传递到处理程序的 System.Threading.Tasks.UnobservedTaskExceptionEventArgs 实例可用于阻止未观察到的异常传播回联接线程。

请参阅

- 任务并行库 (TPL)

如何：使用 Parallel.Invoke 执行并行操作

2021/11/16 •

此示例演示如何通过使用任务并行库中的 Invoke 并行操作。共享的数据源上执行三个操作。因为操作均不修改源，所以可通过直接的方式并行执行操作。

> **NOTE**
> 本文档使用 lambda 表达式在 TPL 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

示例

```
namespace ParallelTasks
{
    using System;
    using System.IO;
    using System.Linq;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    using System.Net;

    class ParallelInvoke
    {
        static void Main()
        {
            // Retrieve Goncharov's "Oblomov" from Gutenberg.org.
            string[] words = CreateWordArray(@"http://www.gutenberg.org/files/54700/54700-0.txt");

            #region ParallelTasks
            // Perform three tasks in parallel on the source array
            Parallel.Invoke(() =>
                            {
                                Console.WriteLine("Begin first task...");
                                GetLongestWord(words);
                            },  // close first Action

                            () =>
                            {
                                Console.WriteLine("Begin second task...");
                                GetMostCommonWords(words);
                            }, //close second Action

                            () =>
                            {
                                Console.WriteLine("Begin third task...");
                                GetCountForWord(words, "sleep");
                            } //close third Action
                        ); //close parallel.invoke

            Console.WriteLine("Returned from Parallel.Invoke");
            #endregion

            Console.WriteLine("Press any key to exit");
            Console.ReadKey();
        }

        #region HelperMethods
```

```
        private static void GetCountForWord(string[] words, string term)
        {
            var findWord = from word in words
                           where word.ToUpper().Contains(term.ToUpper())
                           select word;

            Console.WriteLine($@"Task 3 -- The word ""{term}"" occurs {findWord.Count()} times.");
        }

        private static void GetMostCommonWords(string[] words)
        {
            var frequencyOrder = from word in words
                                 where word.Length > 6
                                 group word by word into g
                                 orderby g.Count() descending
                                 select g.Key;

            var commonWords = frequencyOrder.Take(10);

            StringBuilder sb = new StringBuilder();
            sb.AppendLine("Task 2 -- The most common words are:");
            foreach (var v in commonWords)
            {
                sb.AppendLine("  " + v);
            }
            Console.WriteLine(sb.ToString());
        }

        private static string GetLongestWord(string[] words)
        {
            var longestWord = (from w in words
                               orderby w.Length descending
                               select w).First();

            Console.WriteLine($"Task 1 -- The longest word is {longestWord}.");
            return longestWord;
        }

        // An http request performed synchronously for simplicity.
        static string[] CreateWordArray(string uri)
        {
            Console.WriteLine($"Retrieving from {uri}");

            // Download a web page the easy way.
            string s = new WebClient().DownloadString(uri);

            // Separate string into an array of words, removing some common punctuation.
            return s.Split(
                new char[] { ' ', '\u000A', ',', '.', ';', ':', '-', '_', '/' },
                StringSplitOptions.RemoveEmptyEntries);
        }
        #endregion
    }
}
//        The example displays output like the following:
//              Retrieving from http://www.gutenberg.org/files/54700/54700-0.txt
//              Begin first task...
//              Begin second task...
//              Begin third task...
//              Task 2 -- The most common words are:
//              Oblomov
//              himself
//              Schtoltz
//              Gutenberg
//              Project
//              another
//              thought
//              Oblomov's
//              nothing
```

```
//              nothing
//              replied
//
//              Task 1 -- The longest word is incomprehensible.
//              Task 3 -- The word "sleep" occurs 57 times.
//              Returned from Parallel.Invoke
//              Press any key to exit
```

```
Imports System.Net
Imports System.Threading.Tasks

Module ParallelTasks
    Sub Main()
        ' Retrieve Goncharov's "Oblomov" from Gutenberg.org.
        Dim words As String() = CreateWordArray("http://www.gutenberg.org/files/54700/54700-0.txt")

        '#Region "ParallelTasks"
        ' Perform three tasks in parallel on the source array
        Parallel.Invoke(Sub()
                            Console.WriteLine("Begin first task...")
                            GetLongestWord(words)
                            ' close first Action
                        End Sub,
            Sub()
                Console.WriteLine("Begin second task...")
                GetMostCommonWords(words)
                'close second Action
            End Sub,
            Sub()
                Console.WriteLine("Begin third task...")
                GetCountForWord(words, "sleep")
                'close third Action
            End Sub)
        'close parallel.invoke
        Console.WriteLine("Returned from Parallel.Invoke")
        '#End Region

        Console.WriteLine("Press any key to exit")
        Console.ReadKey()
    End Sub

#Region "HelperMethods"
    Sub GetCountForWord(ByVal words As String(), ByVal term As String)
        Dim findWord = From word In words
                       Where word.ToUpper().Contains(term.ToUpper())
                       Select word

        Console.WriteLine($"Task 3 -- The word ""{term}"" occurs {findWord.Count()} times.")
    End Sub

    Sub GetMostCommonWords(ByVal words As String())
        Dim frequencyOrder = From word In words
                             Where word.Length > 6
                             Group By word
                             Into wordGroup = Group, Count()
                             Order By wordGroup.Count() Descending
                             Select wordGroup

        Dim commonWords = From grp In frequencyOrder
                          Select grp
                          Take (10)

        Dim s As String
        s = "Task 2 -- The most common words are:" & vbCrLf
        For Each v In commonWords
            s = s & v(0) & vbCrLf
        Next
        Console.WriteLine(s)
```

```
    End Sub

    Function GetLongestWord(ByVal words As String()) As String
        Dim longestWord = (From w In words
                           Order By w.Length Descending
                           Select w).First()

        Console.WriteLine($"Task 1 -- The longest word is {longestWord}.")
        Return longestWord
    End Function


    ' An http request performed synchronously for simplicity.
    Function CreateWordArray(ByVal uri As String) As String()
        Console.WriteLine($"Retrieving from {uri}")

        ' Download a web page the easy way.
        Dim s As String = New WebClient().DownloadString(uri)

        ' Separate string into an array of words, removing some common punctuation.
        Return s.Split(New Char() {" "c, ControlChars.Lf, ","c, "."c, ";"c, ":"c,
        "-"c, "_"c, "/"c}, StringSplitOptions.RemoveEmptyEntries)
    End Function
#End Region
End Module
' The exmaple displays output like the following:
'       Retrieving from http://www.gutenberg.org/files/54700/54700-0.txt
'       Begin first task...
'       Begin second task...
'       Begin third task...
'       Task 2 -- The most common words are:
'       Oblomov
'       himself
'       Schtoltz
'       Gutenberg
'       Project
'       another
'       thought
'       Oblomov's
'       nothing
'       replied
'
'       Task 1 -- The longest word is incomprehensible.
'       Task 3 -- The word "sleep" occurs 57 times.
'       Returned from Parallel.Invoke
'       Press any key to exit
```

借助 Invoke，你只需表达想同时运行的操作，运行时会处理所有线程计划详细信息（包括自动缩放至主计算机上的内核数）。

此示例并行操作，而非数据。此外，可使用 PLINQ 并行 LINQ 查询，并按顺序运行查询。或者，可使用 PLINQ 并行数据。另一个选项是并行查询和任务。尽管生成的开销在处理器相对较少的主机计算机上可能会降低性能，但在处理器较多的计算机上可更好地缩放。

编译代码

将完整示例复制和粘贴到 Microsoft Visual Studio 项目，并按 F5 键。

请参阅

- 并行编程
- 如何：取消任务及其子级
- 并行 LINQ (PLINQ)

如何：从任务中返回值

2021/11/16 •

此示例演示如何使用 System.Threading.Tasks.Task<TResult> 类型，以返回 Result 属性的值。它要求 C:\Users\Public\Pictures\Sample Pictures\ 目录存在，并且该目录包含文件。

示例

```
using System;
using System.Linq;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        // Return a value type with a lambda expression
        Task<int> task1 = Task<int>.Factory.StartNew(() => 1);
        int i = task1.Result;

        // Return a named reference type with a multi-line statement lambda.
        Task<Test> task2 = Task<Test>.Factory.StartNew(() =>
        {
            string s = ".NET";
            double d = 4.0;
            return new Test { Name = s, Number = d };
        });
        Test test = task2.Result;

        // Return an array produced by a PLINQ query
        Task<string[]> task3 = Task<string[]>.Factory.StartNew(() =>
        {
            string path = @"C:\Users\Public\Pictures\Sample Pictures\";
            string[] files = System.IO.Directory.GetFiles(path);

            var result = (from file in files.AsParallel()
                          let info = new System.IO.FileInfo(file)
                          where info.Extension == ".jpg"
                          select file).ToArray();

            return result;
        });

        foreach (var name in task3.Result)
            Console.WriteLine(name);
    }
    class Test
    {
        public string Name { get; set; }
        public double Number { get; set; }
    }
}
```

```
Imports System.Threading.Tasks

Module Module1

    Sub Main()
        ReturnAValue()

        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()

    End Sub

    Sub ReturnAValue()

        ' Return a value type with a lambda expression
        Dim task1 = Task(Of Integer).Factory.StartNew(Function() 1)
        Dim i As Integer = task1.Result

        ' Return a named reference type with a multi-line statement lambda.
        Dim task2 As Task(Of Test) = Task.Factory.StartNew(Function()
                                                               Dim s As String = ".NET"
                                                               Dim d As Integer = 4
                                                               Return New Test With {.Name = s, .Number = d}
                                                           End Function)

        Dim myTest As Test = task2.Result
        Console.WriteLine(myTest.Name & ": " & myTest.Number)

        ' Return an array produced by a PLINQ query.
        Dim task3 As Task(Of String()) = Task(Of String()).Factory.StartNew(Function()

                                                                                Dim path =
"C:\Users\Public\Pictures\Sample Pictures\"
                                                                                Dim files =
System.IO.Directory.GetFiles(path)

                                                                                Dim result = (From file In
files.AsParallel()
                                                                                              Let info = New
System.IO.FileInfo(file)
                                                                                              Where
info.Extension = ".jpg"
                                                                                              Select
file).ToArray()
                                                                                Return result
                                                                            End Function)

        For Each name As String In task3.Result
            Console.WriteLine(name)
        Next
    End Sub

    Class Test
        Public Name As String
        Public Number As Double
    End Class
End Module
```

Result 属性将阻止调用线程，直到任务完成。

若要了解如何将一个 System.Threading.Tasks.Task<TResult> 的结果传递到延续任务，请参阅使用延续任务链接任务。

请参阅

- 基于任务的异步编程
- PLINQ 和 TPL 中的 Lambda 表达式

如何：取消任务及其子级

2021/11/16 •

这些示例展示了如何执行下列任务：

1. 创建并启动可取消任务。
2. 将取消令牌传递给用户委托，并视需要传递给任务实例。
3. 注意并响应用户委托中的取消请求。
4. （可选）注意已取消任务的调用线程。

调用线程不会强制结束任务，只会提示取消请求已发出。如果任务已在运行，至于怎样才能注意请求并适当响应，取决于用户委托的选择。如果取消请求在任务运行前发出，用户委托绝不会执行，任务对象的状态会转换为“已取消”。

示例

此示例展示了如何终止 Task 及其子级，以响应取消请求。还会演示，当用户委托通过引发 TaskCanceledException 终止时，调用线程可以选择使用 Wait 方法或 WaitAll 方法来等待任务完成。在这种情况下，必须使用 `try/catch` 块来处理调用线程上的异常。

```
using System;
using System.Collections.Concurrent;
using System.Threading;
using System.Threading.Tasks;

public class Example
{
    public static async Task Main()
    {
        var tokenSource = new CancellationTokenSource();
        var token = tokenSource.Token;

        // Store references to the tasks so that we can wait on them and
        // observe their status after cancellation.
        Task t;
        var tasks = new ConcurrentBag<Task>();

        Console.WriteLine("Press any key to begin tasks...");
        Console.ReadKey(true);
        Console.WriteLine("To terminate the example, press 'c' to cancel and exit...");
        Console.WriteLine();

        // Request cancellation of a single task when the token source is canceled.
        // Pass the token to the user delegate, and also to the task so it can
        // handle the exception correctly.
        t = Task.Run(() => DoSomeWork(1, token), token);
        Console.WriteLine("Task {0} executing", t.Id);
        tasks.Add(t);

        // Request cancellation of a task and its children. Note the token is passed
        // to (1) the user delegate and (2) as the second argument to Task.Run, so
        // that the task instance can correctly handle the OperationCanceledException.
        t = Task.Run(() =>
        {
            // Create some cancelable child tasks.
            Task tc;
            for (int i = 3; i <= 10; i++)
```

```
            for (int i = 3; i <= 10; i++)
            {
                // For each child task, pass the same token
                // to each user delegate and to Task.Run.
                tc = Task.Run(() => DoSomeWork(i, token), token);
                Console.WriteLine("Task {0} executing", tc.Id);
                tasks.Add(tc);
                // Pass the same token again to do work on the parent task.
                // All will be signaled by the call to tokenSource.Cancel below.
                DoSomeWork(2, token);
            }
        }, token);

        Console.WriteLine("Task {0} executing", t.Id);
        tasks.Add(t);

        // Request cancellation from the UI thread.
        char ch = Console.ReadKey().KeyChar;
        if (ch == 'c' || ch == 'C')
        {
            tokenSource.Cancel();
            Console.WriteLine("\nTask cancellation requested.");

            // Optional: Observe the change in the Status property on the task.
            // It is not necessary to wait on tasks that have canceled. However,
            // if you do wait, you must enclose the call in a try-catch block to
            // catch the TaskCanceledExceptions that are thrown. If you do
            // not wait, no exception is thrown if the token that was passed to the
            // Task.Run method is the same token that requested the cancellation.
        }

        try
        {
            await Task.WhenAll(tasks.ToArray());
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine($"\n{nameof(OperationCanceledException)} thrown\n");
        }
        finally
        {
            tokenSource.Dispose();
        }

        // Display status of all tasks.
        foreach (var task in tasks)
            Console.WriteLine("Task {0} status is now {1}", task.Id, task.Status);
    }

    static void DoSomeWork(int taskNum, CancellationToken ct)
    {
        // Was cancellation already requested?
        if (ct.IsCancellationRequested)
        {
            Console.WriteLine("Task {0} was cancelled before it got started.",
                              taskNum);
            ct.ThrowIfCancellationRequested();
        }

        int maxIterations = 100;

        // NOTE!!! A "TaskCanceledException was unhandled
        // by user code" error will be raised here if "Just My Code"
        // is enabled on your computer. On Express editions JMC is
        // enabled and cannot be disabled. The exception is benign.
        // Just press F5 to continue executing your code.
        for (int i = 0; i <= maxIterations; i++)
        {
            // Do a bit of work. Not too much.
```

```
            var sw = new SpinWait();
            for (int j = 0; j <= 100; j++)
                sw.SpinOnce();

            if (ct.IsCancellationRequested)
            {
                Console.WriteLine("Task {0} cancelled", taskNum);
                ct.ThrowIfCancellationRequested();
            }
        }
    }
}
// The example displays output like the following:
//       Press any key to begin tasks...
//    To terminate the example, press 'c' to cancel and exit...
//
//    Task 1 executing
//    Task 2 executing
//    Task 3 executing
//    Task 4 executing
//    Task 5 executing
//    Task 6 executing
//    Task 7 executing
//    Task 8 executing
//    c
//    Task cancellation requested.
//    Task 2 cancelled
//    Task 7 cancelled
//
//    OperationCanceledException thrown
//
//    Task 2 status is now Canceled
//    Task 1 status is now RanToCompletion
//    Task 8 status is now Canceled
//    Task 7 status is now Canceled
//    Task 6 status is now RanToCompletion
//    Task 5 status is now RanToCompletion
//    Task 4 status is now RanToCompletion
//    Task 3 status is now RanToCompletion
```

```
Imports System.Collections.Concurrent
Imports System.Threading
Imports System.Threading.Tasks

Module Example
    Sub Main()
        Dim tokenSource As New CancellationTokenSource()
        Dim token As CancellationToken = tokenSource.Token

        ' Store references to the tasks so that we can wait on them and
        ' observe their status after cancellation.
        Dim t As Task
        Dim tasks As New ConcurrentBag(Of Task)()

        Console.WriteLine("Press any key to begin tasks...")
        Console.ReadKey(True)
        Console.WriteLine("To terminate the example, press 'c' to cancel and exit...")
        Console.WriteLine()

        ' Request cancellation of a single task when the token source is canceled.
        ' Pass the token to the user delegate, and also to the task so it can
        ' handle the exception correctly.
        t = Task.Factory.StartNew(Sub() DoSomeWork(1, token), token)
        Console.WriteLine("Task {0} executing", t.Id)
        tasks.Add(t)

        ' Request cancellation of a task and its children. Note the token is passed
        ' to (1) the user delegate and (2) as the second argument to StartNew, so
```

```
        ' to (1) the user delegate and (2) as the second argument to StartNew, so
        ' that the task instance can correctly handle the OperationCanceledException.
        t = Task.Factory.StartNew(Sub()
                                      ' Create some cancelable child tasks.
                                      Dim tc As Task
                                      For i As Integer = 3 To 10
                                          ' For each child task, pass the same token
                                          ' to each user delegate and to StartNew.
                                          tc = Task.Factory.StartNew(Sub(iteration) DoSomeWork(iteration,
token), i, token)

                                          Console.WriteLine("Task {0} executing", tc.Id)
                                          tasks.Add(tc)
                                          ' Pass the same token again to do work on the parent task.
                                          ' All will be signaled by the call to tokenSource.Cancel below.
                                          DoSomeWork(2, token)
                                      Next
                                  End Sub,
                                  token)

        Console.WriteLine("Task {0} executing", t.Id)
        tasks.Add(t)

        ' Request cancellation from the UI thread.
        Dim ch As Char = Console.ReadKey().KeyChar
        If ch = "c"c Or ch = "C"c Then
            tokenSource.Cancel()
            Console.WriteLine(vbCrLf + "Task cancellation requested.")

            ' Optional: Observe the change in the Status property on the task.
            ' It is not necessary to wait on tasks that have canceled. However,
            ' if you do wait, you must enclose the call in a try-catch block to
            ' catch the TaskCanceledExceptions that are thrown. If you do
            ' not wait, no exception is thrown if the token that was passed to the
            ' StartNew method is the same token that requested the cancellation.
        End If

        Try
            Task.WaitAll(tasks.ToArray())
        Catch e As AggregateException
            Console.WriteLine()
            Console.WriteLine("AggregateException thrown with the following inner exceptions:")
            ' Display information about each exception.
            For Each v In e.InnerExceptions
                If TypeOf v Is TaskCanceledException
                    Console.WriteLine("   TaskCanceledException: Task {0}",
                                      DirectCast(v, TaskCanceledException).Task.Id)
                Else
                    Console.WriteLine("   Exception: {0}", v.GetType().Name)
                End If
            Next
            Console.WriteLine()
        Finally
            tokenSource.Dispose()
        End Try

        ' Display status of all tasks.
        For Each t In tasks
            Console.WriteLine("Task {0} status is now {1}", t.Id, t.Status)
        Next
    End Sub

    Sub DoSomeWork(ByVal taskNum As Integer, ByVal ct As CancellationToken)
        ' Was cancellation already requested?
        If ct.IsCancellationRequested = True Then
            Console.WriteLine("Task {0} was cancelled before it got started.",
                              taskNum)
            ct.ThrowIfCancellationRequested()
        End If
```

```
        Dim maxIterations As Integer = 100

        ' NOTE!!! A "TaskCanceledException was unhandled
        ' by user code" error will be raised here if "Just My Code"
        ' is enabled on your computer. On Express editions JMC is
        ' enabled and cannot be disabled. The exception is benign.
        ' Just press F5 to continue executing your code.
        For i As Integer = 0 To maxIterations
            ' Do a bit of work. Not too much.
            Dim sw As New SpinWait()
            For j As Integer = 0 To 100
                sw.SpinOnce()
            Next
            If ct.IsCancellationRequested Then
                Console.WriteLine("Task {0} cancelled", taskNum)
                ct.ThrowIfCancellationRequested()
            End If
        Next
    End Sub
End Module
' The example displays output like the following:
'    Press any key to begin tasks...
'    To terminate the example, press 'c' to cancel and exit...
'
'    Task 1 executing
'    Task 2 executing
'    Task 3 executing
'    Task 4 executing
'    Task 5 executing
'    Task 6 executing
'    Task 7 executing
'    Task 8 executing
'    c
'    Task cancellation requested.
'    Task 2 cancelled
'    Task 7 cancelled
'
'    AggregateException thrown with the following inner exceptions:
'       TaskCanceledException: Task 2
'       TaskCanceledException: Task 8
'       TaskCanceledException: Task 7
'
'    Task 2 status is now Canceled
'    Task 1 status is now RanToCompletion
'    Task 8 status is now Canceled
'    Task 7 status is now Canceled
'    Task 6 status is now RanToCompletion
'    Task 5 status is now RanToCompletion
'    Task 4 status is now RanToCompletion
'    Task 3 status is now RanToCompletion
```

System.Threading.Tasks.Task 类与基于 System.Threading.CancellationTokenSource 和 System.Threading.CancellationToken 类型的取消模型完全集成。有关详细信息，请参阅托管线程中的取消和任务取消。

请参阅

- System.Threading.CancellationTokenSource
- System.Threading.CancellationToken
- System.Threading.Tasks.Task
- System.Threading.Tasks.Task<TResult>
- 基于任务的异步编程
- 附加和分离的子任务

- PLINQ 和 TPL 中的 Lambda 表达式

如何：创建预先计算的任务

2021/11/16 •

本文档介绍如何使用 Task.FromResult 方法检索缓存中包含的异步下载操作的结果。FromResult 方法返回一个将所提供的值作为其 Task<TResult> 属性的已完成的 Result 对象。执行返回 Task<TResult> 对象的异步运算，且已计算该 Task<TResult> 对象的结果时，此方法将十分有用。

示例

下面的示例从 Web 下载字符串。它定义了 `DownloadStringAsync` 方法。此方法从 Web 异步下载字符串。此示例还使用 ConcurrentDictionary<TKey,TValue> 对象来缓存先前操作的结果。如果此缓存中包含输入的地址，`DownloadStringAsync` 会使用 FromResult 方法来生成包含位于该地址的内容的 Task<TResult> 对象。否则，`DownloadStringAsync` 从 Web 下载文件并将结果添加到缓存中。

```
using System;
using System.Collections.Concurrent;
using System.Diagnostics;
using System.Linq;
using System.Net;
using System.Threading.Tasks;

// Demonstrates how to use Task<TResult>.FromResult to create a task
// that holds a pre-computed result.
class CachedDownloads
{
   // Holds the results of download operations.
   static ConcurrentDictionary<string, string> cachedDownloads =
      new ConcurrentDictionary<string, string>();

   // Asynchronously downloads the requested resource as a string.
   public static Task<string> DownloadStringAsync(string address)
   {
      // First try to retrieve the content from cache.
      string content;
      if (cachedDownloads.TryGetValue(address, out content))
      {
         return Task.FromResult<string>(content);
      }

      // If the result was not in the cache, download the
      // string and add it to the cache.
      return Task.Run(async () =>
      {
         content = await new WebClient().DownloadStringTaskAsync(address);
         cachedDownloads.TryAdd(address, content);
         return content;
      });
   }

   static void Main(string[] args)
   {
      // The URLs to download.
      string[] urls = new string[]
      {
         "http://msdn.microsoft.com",
         "http://www.contoso.com",
         "http://www.microsoft.com"
      };

      // Used to time download operations.
```

```
      // Used to time download operations.
      Stopwatch stopwatch = new Stopwatch();

      // Compute the time required to download the URLs.
      stopwatch.Start();
      var downloads = from url in urls
                      select DownloadStringAsync(url);
      Task.WhenAll(downloads).ContinueWith(results =>
      {
         stopwatch.Stop();

         // Print the number of characters download and the elapsed time.
         Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
            results.Result.Sum(result => result.Length),
            stopwatch.ElapsedMilliseconds);
      })
      .Wait();

      // Perform the same operation a second time. The time required
      // should be shorter because the results are held in the cache.
      stopwatch.Restart();
      downloads = from url in urls
                  select DownloadStringAsync(url);
      Task.WhenAll(downloads).ContinueWith(results =>
      {
         stopwatch.Stop();

         // Print the number of characters download and the elapsed time.
         Console.WriteLine("Retrieved {0} characters. Elapsed time was {1} ms.",
            results.Result.Sum(result => result.Length),
            stopwatch.ElapsedMilliseconds);
      })
      .Wait();
   }
}

/* Sample output:
Retrieved 27798 characters. Elapsed time was 1045 ms.
Retrieved 27798 characters. Elapsed time was 0 ms.
*/
```

```
Imports System.Collections.Concurrent
Imports System.Diagnostics
Imports System.Linq
Imports System.Net
Imports System.Threading.Tasks

' Demonstrates how to use Task<TResult>.FromResult to create a task
' that holds a pre-computed result.
Friend Class CachedDownloads
    ' Holds the results of download operations.
    Private Shared cachedDownloads As New ConcurrentDictionary(Of String, String)()

    ' Asynchronously downloads the requested resource as a string.
    Public Shared Function DownloadStringAsync(ByVal address As String) As Task(Of String)
        ' First try to retrieve the content from cache.
        Dim content As String
        If cachedDownloads.TryGetValue(address, content) Then
            Return Task.FromResult(Of String)(content)
        End If

        ' If the result was not in the cache, download the
        ' string and add it to the cache.
        Return Task.Run(async Function()
                            content = await New WebClient().DownloadStringTaskAsync(address)
                            cachedDownloads.TryAdd(address, content)
                            Return content
                        End Function)
```

```
                        End Function)
    End Function

    Shared Sub Main(ByVal args() As String)
        ' The URLs to download.
        Dim urls() As String = {"http://msdn.microsoft.com", "http://www.contoso.com",
"http://www.microsoft.com"}

        ' Used to time download operations.
        Dim stopwatch As New Stopwatch()

        ' Compute the time required to download the URLs.
        stopwatch.Start()
        Dim downloads = From url In urls _
                        Select DownloadStringAsync(url)
        Task.WhenAll(downloads).ContinueWith(Sub(results)
                                                 ' Print the number of characters download and the elapsed
time.
                                                 stopwatch.Stop()
                                                 Console.WriteLine("Retrieved {0} characters. Elapsed time
was {1} ms.", results.Result.Sum(Function(result) result.Length), stopwatch.ElapsedMilliseconds)
                                             End Sub).Wait()

        ' Perform the same operation a second time. The time required
        ' should be shorter because the results are held in the cache.
        stopwatch.Restart()
        downloads = From url In urls _
                    Select DownloadStringAsync(url)
        Task.WhenAll(downloads).ContinueWith(Sub(results)
                                                 ' Print the number of characters download and the elapsed
time.
                                                 stopwatch.Stop()
                                                 Console.WriteLine("Retrieved {0} characters. Elapsed time
was {1} ms.", results.Result.Sum(Function(result) result.Length), stopwatch.ElapsedMilliseconds)
                                             End Sub).Wait()
    End Sub
End Class

' Sample output:
'Retrieved 27798 characters. Elapsed time was 1045 ms.
'Retrieved 27798 characters. Elapsed time was 0 ms.
'
```

此示例计算两次下载多个字符串需要的时间。与第一组下载操作相比，第二组下载操作应该会花费较少的时间，因为结果已包含在缓存中。FromResult 方法可以使 `DownloadStringAsync` 方法创建包含这些预计算结果的 Task<TResult> 对象。

另请参阅

- 基于任务的异步编程

如何：使用并行任务遍历二叉树

2021/11/16 •

下面的示例展示了两种使用并行任务遍历树数据结构的方式。树创建本身是留给大家练练手的。

示例

```
public class TreeWalk
{
    static void Main()
    {
        Tree<MyClass> tree = new Tree<MyClass>();

        // ...populate tree (left as an exercise)

        // Define the Action to perform on each node.
        Action<MyClass> myAction = x => Console.WriteLine("{0} : {1}", x.Name, x.Number);

        // Traverse the tree with parallel tasks.
        DoTree(tree, myAction);
    }

    public class MyClass
    {
        public string Name { get; set; }
        public int Number { get; set; }
    }
    public class Tree<T>
    {
        public Tree<T> Left;
        public Tree<T> Right;
        public T Data;
    }

    // By using tasks explcitly.
    public static void DoTree<T>(Tree<T> tree, Action<T> action)
    {
        if (tree == null) return;
        var left = Task.Factory.StartNew(() => DoTree(tree.Left, action));
        var right = Task.Factory.StartNew(() => DoTree(tree.Right, action));
        action(tree.Data);

        try
        {
            Task.WaitAll(left, right);
        }
        catch (AggregateException )
        {
            //handle exceptions here
        }
    }

    // By using Parallel.Invoke
    public static void DoTree2<T>(Tree<T> tree, Action<T> action)
    {
        if (tree == null) return;
        Parallel.Invoke(
            () => DoTree2(tree.Left, action),
            () => DoTree2(tree.Right, action),
            () => action(tree.Data)
        );
    }
}
```

```
Imports System.Threading.Tasks

Public Class TreeWalk

    Shared Sub Main()

        Dim tree As Tree(Of Person) = New Tree(Of Person)()

        ' ...populate tree (left as an exercise)

        ' Define the Action to perform on each node.
        Dim myAction As Action(Of Person) = New Action(Of Person)(Sub(x)
                                                                      Console.WriteLine("{0}  : {1} ",
x.Name, x.Number)
                                                                  End Sub)

        ' Traverse the tree with parallel tasks.
        DoTree(tree, myAction)
    End Sub

    Public Class Person
        Public Name As String
        Public Number As Integer
    End Class

    Public Class Tree(Of T)
        Public Left As Tree(Of T)
        Public Right As Tree(Of T)
        Public Data As T
    End Class

    ' By using tasks explicitly.
    Public Shared Sub DoTree(Of T)(ByVal myTree As Tree(Of T), ByVal a As Action(Of T))
        If myTree Is Nothing Then
            Return
        End If
        Dim left = Task.Factory.StartNew(Sub() DoTree(myTree.Left, a))
        Dim right = Task.Factory.StartNew(Sub() DoTree(myTree.Right, a))
        a(myTree.Data)

        Try
            Task.WaitAll(left, right)
        Catch ae As AggregateException
            'handle exceptions here
        End Try
    End Sub

    ' By using Parallel.Invoke
    Public Shared Sub DoTree2(Of T)(ByVal myTree As Tree(Of T), ByVal myAct As Action(Of T))
        If myTree Is Nothing Then
            Return
        End If
        Parallel.Invoke(
            Sub() DoTree2(myTree.Left, myAct),
            Sub() DoTree2(myTree.Right, myAct),
            Sub() myAct(myTree.Data)
        )
    End Sub
End Class
```

上面的两种方法在功能上是相当的。通过使用 StartNew 方法创建和运行任务，可以从可用于等待任务和处理异常的任务中返回句柄。

另请参阅

- 任务并行库 (TPL)

如何：解除嵌套任务的包装

2021/11/16 •

可以从方法返回任务，再等待或继续执行此任务，如下面的示例所示：

```
static Task<string> DoWorkAsync()
{
    return Task<String>.Factory.StartNew(() =>
    {
       //...
        return "Work completed.";
    });
}

static void StartTask()
{
    Task<String> t = DoWorkAsync();
    t.Wait();
    Console.WriteLine(t.Result);
}
```

```
Shared Function DoWorkAsync() As Task(Of String)

    Return Task(Of String).Run(Function()
                                   '...
                                   Return "Work completed."
                               End Function)
End Function

Shared Sub StartTask()

    Dim t As Task(Of String) = DoWorkAsync()
    t.Wait()
    Console.WriteLine(t.Result)
End Sub
```

在上面的示例中，Result 属性的类型为 `string`（Visual Basic 中的 `String`）。

不过，在某些情况下，建议在另一个任务中创建任务，再返回嵌套任务。在这种情况下，封闭任务的 `TResult` 本身就是任务。在下面的示例中，Result 属性是 C# 中的 `Task<Task<string>>` 或 Visual Basic 中的 `Task(Of Task(Of String))`。

```
// Note the type of t and t2.
Task<Task<string>> t = Task.Factory.StartNew(() => DoWorkAsync());
Task<Task<string>> t2 = DoWorkAsync().ContinueWith((s) => DoMoreWorkAsync());

// Outputs: System.Threading.Tasks.Task`1[System.String]
Console.WriteLine(t.Result);
```

```
' Note the type of t and t2.
Dim t As Task(Of Task(Of String)) = Task.Run(Function() DoWorkAsync())
Dim t2 As Task(Of Task(Of String)) = DoWorkAsync().ContinueWith(Function(s) DoMoreWorkAsync())

' Outputs: System.Threading.Tasks.Task`1[System.String]
Console.WriteLine(t.Result)
```

虽然可以编写代码来取消包装外部任务并检索原始任务及其 Result 属性，但此类代码不易编写，因为必须处理异常和取消请求。在这种情况下，建议使用 Unwrap 扩展方法之一，如下面的示例所示。

```
// Unwrap the inner task.
Task<string> t3 = DoWorkAsync().ContinueWith((s) => DoMoreWorkAsync()).Unwrap();

// Outputs "More work completed."
Console.WriteLine(t.Result);
```

```
' Unwrap the inner task.
Dim t3 As Task(Of String) = DoWorkAsync().ContinueWith(Function(s) DoMoreWorkAsync()).Unwrap()

' Outputs "More work completed."
Console.WriteLine(t.Result)
```

Unwrap 方法可用于将任何 `Task<Task>` 或 `Task<Task<TResult>>`（Visual Basic 中的 `Task(Of Task)` 或 `Task(Of Task(Of TResult))`）转换为 `Task` 或 `Task<TResult>`（Visual Basic 中的 `Task(Of TResult)`）。新任务完全表示内部嵌套任务，并包含取消状态和所有异常。

示例

下面的示例展示了如何使用 Unwrap 扩展方法。

```
namespace Unwrap
{
    using System;
    using System.Collections.Generic;
    using System.Linq;
    using System.Text;
    using System.Threading;
    using System.Threading.Tasks;
    // A program whose only use is to demonstrate Unwrap.
    class Program
    {
        static void Main()
        {
            // An arbitrary threshold value.
            byte threshold = 0x40;

            // data is a Task<byte[]>
            var data = Task<byte[]>.Factory.StartNew(() =>
                {
                    return GetData();
                });

            // We want to return a task so that we can
            // continue from it later in the program.
            // Without Unwrap: stepTwo is a Task<Task<byte[]>>
            // With Unwrap: stepTwo is a Task<byte[]>
            var stepTwo = data.ContinueWith((antecedent) =>
                {
                    return Task<byte>.Factory.StartNew( () => Compute(antecedent.Result));
```

```
                })
                .Unwrap();

            // Without Unwrap: antecedent.Result = Task<byte>
            // and the following method will not compile.
            // With Unwrap: antecedent.Result = byte and
            // we can work directly with the result of the Compute method.
            var lastStep = stepTwo.ContinueWith( (antecedant) =>
                {
                    if (antecedant.Result >= threshold)
                    {
                      return Task.Factory.StartNew( () =>  Console.WriteLine("Program complete. Final =
0x{0:x} threshold = 0x{1:x}", stepTwo.Result, threshold));
                    }
                    else
                    {
                        return DoSomeOtherAsyncronousWork(stepTwo.Result, threshold);
                    }
                });

            lastStep.Wait();
            Console.WriteLine("Press any key");
            Console.ReadKey();
        }

        #region Dummy_Methods
        private static byte[] GetData()
        {
            Random rand = new Random();
            byte[] bytes = new byte[64];
            rand.NextBytes(bytes);
            return bytes;
        }

        static Task DoSomeOtherAsyncronousWork(int i, byte b2)
        {
            return Task.Factory.StartNew(() =>
                {
                    Thread.SpinWait(500000);
                    Console.WriteLine("Doing more work. Value was <= threshold");
                });
        }
        static byte Compute(byte[] data)
        {

            byte final = 0;
            foreach (byte item in data)
            {
                final ^= item;
                Console.WriteLine("{0:x}", final);
            }
            Console.WriteLine("Done computing");
            return final;
        }
        #endregion
    }
}
```

```
'How to: Unwrap a Task
Imports System.Threading
Imports System.Threading.Tasks

Module UnwrapATask2

    Sub Main()
        ' An arbitrary threshold value.
        Dim threshold As Byte = &H40
```

```
        ' myData is a Task(Of Byte())

        Dim myData As Task(Of Byte()) = Task.Factory.StartNew(Function()
                                                                  Return GetData()
                                                              End Function)
        ' We want to return a task so that we can
        ' continue from it later in the program.
        ' Without Unwrap: stepTwo is a Task(Of Task(Of Byte))
        ' With Unwrap: stepTwo is a Task(Of Byte)

        Dim stepTwo = myData.ContinueWith(Function(antecedent)
                                              Return Task.Factory.StartNew(Function()
                                                                               Return
Compute(antecedent.Result)
                                                                           End Function)
                                          End Function).Unwrap()

        Dim lastStep = stepTwo.ContinueWith(Function(antecedent)
                                                Console.WriteLine("Result = {0}", antecedent.Result)
                                                If antecedent.Result >= threshold Then
                                                    Return Task.Factory.StartNew(Sub()

Console.WriteLine("Program complete. Final = &H{1:x} threshold = &H{1:x}",

stepTwo.Result, threshold)
                                                                                     End Sub)
                                                Else
                                                    Return DoSomeOtherAsynchronousWork(stepTwo.Result,
threshold)
                                                End If
                                            End Function)
        Try
            lastStep.Wait()
        Catch ae As AggregateException
            For Each ex As Exception In ae.InnerExceptions
                Console.WriteLine(ex.Message & ex.StackTrace & ex.GetBaseException.ToString())
            Next
        End Try

        Console.WriteLine("Press any key")
        Console.ReadKey()
    End Sub

#Region "Dummy_Methods"
    Function GetData() As Byte()
        Dim rand As Random = New Random()
        Dim bytes(64) As Byte
        rand.NextBytes(bytes)
        Return bytes
    End Function

    Function DoSomeOtherAsynchronousWork(ByVal i As Integer, ByVal b2 As Byte) As Task
        Return Task.Factory.StartNew(Sub()
                                         Thread.SpinWait(500000)
                                         Console.WriteLine("Doing more work. Value was <= threshold.")
                                     End Sub)
    End Function

    Function Compute(ByVal d As Byte()) As Byte
        Dim final As Byte = 0
        For Each item As Byte In d
            final = final Xor item
            Console.WriteLine("{0:x}", final)
        Next
        Console.WriteLine("Done computing")
        Return final
    End Function
#End Region
```

```
End Module
```

另请参阅

- System.Threading.Tasks.TaskExtensions
- **基于任务的异步编程**

如何：防止子任务附加到其父任务

2021/11/16 •

本文档演示如何阻止子任务附加到父任务。在调用由第三方编写的也使用任务的组件时，阻止子任务附加到其父级是有用的。例如，使用 TaskCreationOptions.AttachedToParent 选项创建 Task 或 Task<TResult> 对象的第三方组件，如果长时间运行或引发未经处理的异常，可能会导致代码中出现问题。

示例

下面的示例对使用默认选项的效果与阻止子任务附加到其父级的效果进行比较。示例创建了一个 Task 对象，该对象可调入同时使用 Task 对象的第三方库。第三方库使用 AttachedToParent 选项创建 Task 对象。应用程序使用 TaskCreationOptions.DenyChildAttach 选项创建父任务。该选项指示运行时移除子任务中的 AttachedToParent 规范。

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

// Defines functionality that is provided by a third-party.
// In a real-world scenario, this would likely be provided
// in a separate code file or assembly.
namespace Contoso
{
   public class Widget
   {
      public Task Run()
      {
         // Create a long-running task that is attached to the
         // parent in the task hierarchy.
         return Task.Factory.StartNew(() =>
         {
            // Simulate a lengthy operation.
            Thread.Sleep(5000);
         }, TaskCreationOptions.AttachedToParent);
      }
   }
}

// Demonstrates how to prevent a child task from attaching to the parent.
class DenyChildAttach
{
   static void RunWidget(Contoso.Widget widget,
      TaskCreationOptions parentTaskOptions)
   {
      // Record the time required to run the parent
      // and child tasks.
      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      Console.WriteLine("Starting widget as a background task...");

      // Run the widget task in the background.
      Task<Task> runWidget = Task.Factory.StartNew(() =>
         {
            Task widgetTask = widget.Run();

            // Perform other work while the task runs...
            Thread.Sleep(1000);
```

```
            return widgetTask;
         }, parentTaskOptions);

      // Wait for the parent task to finish.
      Console.WriteLine("Waiting for parent task to finish...");
      runWidget.Wait();
      Console.WriteLine("Parent task has finished. Elapsed time is {0} ms.",
         stopwatch.ElapsedMilliseconds);

      // Perform more work...
      Console.WriteLine("Performing more work on the main thread...");
      Thread.Sleep(2000);
      Console.WriteLine("Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds);

      // Wait for the child task to finish.
      Console.WriteLine("Waiting for child task to finish...");
      runWidget.Result.Wait();
      Console.WriteLine("Child task has finished. Elapsed time is {0} ms.",
        stopwatch.ElapsedMilliseconds);
   }

   static void Main(string[] args)
   {
      Contoso.Widget w = new Contoso.Widget();

      // Perform the same operation two times. The first time, the operation
      // is performed by using the default task creation options. The second
      // time, the operation is performed by using the DenyChildAttach option
      // in the parent task.

      Console.WriteLine("Demonstrating parent/child tasks with default options...");
      RunWidget(w, TaskCreationOptions.None);

      Console.WriteLine();

      Console.WriteLine("Demonstrating parent/child tasks with the DenyChildAttach option...");
      RunWidget(w, TaskCreationOptions.DenyChildAttach);
   }
}

/* Sample output:
Demonstrating parent/child tasks with default options...
Starting widget as a background task...
Waiting for parent task to finish...
Parent task has finished. Elapsed time is 5014 ms.
Performing more work on the main thread...
Elapsed time is 7019 ms.
Waiting for child task to finish...
Child task has finished. Elapsed time is 7019 ms.

Demonstrating parent/child tasks with the DenyChildAttach option...
Starting widget as a background task...
Waiting for parent task to finish...
Parent task has finished. Elapsed time is 1007 ms.
Performing more work on the main thread...
Elapsed time is 3015 ms.
Waiting for child task to finish...
Child task has finished. Elapsed time is 5015 ms.
*/
```

```
Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks

' Defines functionality that is provided by a third-party.
' In a real-world scenario, this would likely be provided
' in a separate code file or assembly.
```

```
' In a separate code file or assembly:
Namespace Contoso
    Public Class Widget
        Public Function Run() As Task
            ' Create a long-running task that is attached to the
            ' parent in the task hierarchy.
            Return Task.Factory.StartNew(Sub() Thread.Sleep(5000), TaskCreationOptions.AttachedToParent)
            ' Simulate a lengthy operation.
        End Function
    End Class
End Namespace

' Demonstrates how to prevent a child task from attaching to the parent.
Friend Class DenyChildAttach
    Private Shared Sub RunWidget(ByVal widget As Contoso.Widget, ByVal parentTaskOptions As
TaskCreationOptions)
        ' Record the time required to run the parent
        ' and child tasks.
        Dim stopwatch As New Stopwatch()
        stopwatch.Start()

        Console.WriteLine("Starting widget as a background task...")

        ' Run the widget task in the background.
        Dim runWidget As Task(Of Task) = Task.Factory.StartNew(Function()
                                                                   ' Perform other work while the task
runs...
                                                                   Dim widgetTask As Task = widget.Run()
                                                                   Thread.Sleep(1000)
                                                                   Return widgetTask
                                                               End Function, parentTaskOptions)

        ' Wait for the parent task to finish.
        Console.WriteLine("Waiting for parent task to finish...")
        runWidget.Wait()
        Console.WriteLine("Parent task has finished. Elapsed time is {0} ms.",
stopwatch.ElapsedMilliseconds)

        ' Perform more work...
        Console.WriteLine("Performing more work on the main thread...")
        Thread.Sleep(2000)
        Console.WriteLine("Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds)

        ' Wait for the child task to finish.
        Console.WriteLine("Waiting for child task to finish...")
        runWidget.Result.Wait()
        Console.WriteLine("Child task has finished. Elapsed time is {0} ms.", stopwatch.ElapsedMilliseconds)
    End Sub

    Shared Sub Main(ByVal args() As String)
        Dim w As New Contoso.Widget()

        ' Perform the same operation two times. The first time, the operation
        ' is performed by using the default task creation options. The second
        ' time, the operation is performed by using the DenyChildAttach option
        ' in the parent task.

        Console.WriteLine("Demonstrating parent/child tasks with default options...")
        RunWidget(w, TaskCreationOptions.None)

        Console.WriteLine()

        Console.WriteLine("Demonstrating parent/child tasks with the DenyChildAttach option...")
        RunWidget(w, TaskCreationOptions.DenyChildAttach)
    End Sub
End Class

' Sample output:
'Demonstrating parent/child tasks with default options...
'Starting widget as a background task...
```

```
'Starting widget as a background task...
'Waiting for parent task to finish...
'Parent task has finished. Elapsed time is 5014 ms.
'Performing more work on the main thread...
'Elapsed time is 7019 ms.
'Waiting for child task to finish...
'Child task has finished. Elapsed time is 7019 ms.
'
'Demonstrating parent/child tasks with the DenyChildAttach option...
'Starting widget as a background task...
'Waiting for parent task to finish...
'Parent task has finished. Elapsed time is 1007 ms.
'Performing more work on the main thread...
'Elapsed time is 3015 ms.
'Waiting for child task to finish...
'Child task has finished. Elapsed time is 5015 ms.
'
```

因为父任务只有在所有子任务完成后才会完成，所以长时间运行的子任务会让整个应用程序执行得非常缓慢。在此示例中，当应用程序使用默认选项创建父任务时，子任务必须在父任务完成之前完成。当应用程序使用 TaskCreationOptions.DenyChildAttach 选项时，子任务未附加到父任务。因此，应用程序可以在父任务完成之后且必须等待子任务完成之前执行其他工作。

另请参阅

- 基于任务的异步编程

数据流（任务并行库）

2021/11/16 •

任务并行库 (TPL) 提供数据流组件，可帮助提高启用并发的应用程序的可靠性。这些数据流组件统称为 TPL 数据流库。这种数据流模型通过向粗粒度的数据流和管道任务提供进程内消息传递来促进基于角色的编程。数据流组件基于 TPL 的类型和计划基础结构，并集成了 C#、Visual Basic 和 F# 语言的异步编程支持。当您有必须相互异步沟通的多个操作或者想要在数据可用时对其处理时，这些数据流组件就非常有用。例如，请考虑一个处理网络摄像机图像数据的应用程序。通过使用数据流模型，当图像帧可用时，应用程序就可以处理它们。如果应用程序增强图像帧（例如执行灯光修正或消除红眼），则可以创建数据流组件的管道。管道的每个阶段可以使用更粗粒度的并行功能（例如 TPL 提供的功能）来转换图像。

本文档对 TPL 数据流库进行了概述。它介绍编程模型，预定义的数据流块类型，以及如何配置数据流块来满足应用程序的特定要求。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

编程模型

TPL 数据流库向具有高吞吐量和低滞后时间的占用大量 CPU 和 I/O 操作的应用程序的并行化和消息传递提供了基础。它还能显式控制缓存数据的方式以及在系统中移动的方式。为了更好地了解数据流编程模型，请考虑一个以异步方式从磁盘加载图像并创建复合图像的应用程序。传统编程模型通常需要使用回调和同步对象（例如锁）来协调任务和访问共享数据。通过使用数据流编程模型，您可以从磁盘读取时创建处理图像的数据流对象。在数据流模型下，您可以声明当数据可用时的处理方式，以及数据之间的所有依赖项。由于运行时管理数据之间的依赖项，因此通常可以避免这种要求来同步访问共享数据。此外，因为运行时计划基于数据的异步到达，所以数据流可以通过有效管理基础线程提高响应能力和吞吐量。有关在 Windows 窗体应用程序中使用数据流编程模型实现图像处理的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

源和目标

TPL 数据流库包括 *数据流块*，它是缓冲并处理数据的数据结构。TPL 定义了三种数据流块：源块、目标块和传播器块。源块作为数据源，可以读取。目标块作为数据接收方，可以写入。传播器块作为源块和目标块，可以读取和写入。TPL 定义 System.Threading.Tasks.Dataflow.ISourceBlock<TOutput> 接口来表示源，System.Threading.Tasks.Dataflow.ITargetBlock<TInput> 表示目标以及 System.Threading.Tasks.Dataflow.IPropagatorBlock<TInput,TOutput> 表示传播器。IPropagatorBlock<TInput,TOutput> 继承自 ISourceBlock<TOutput> 和 ITargetBlock<TInput>。

TPL 数据流库提供了多个预定义的数据流块类型，可以实现 ISourceBlock<TOutput>、ITargetBlock<TInput> 和 IPropagatorBlock<TInput,TOutput> 接口。这些数据流块类型在本文档的预定义的数据流块类型部分进行了说明。

连接块

可以连接数据流块来形成管道（这是数据流块的线性序列），或网络（这是数据流块的图形）。管道是网络的一种形式。在管道或网络中，当数据可用时源向目标异步传播数据。ISourceBlock<TOutput>.LinkTo 方法将源数据流块链接到目标块。源可以链接到零个或多个目标；目标可以从零个或多个源进行链接。您可以同时向管道或网络中添加或从其移除数据流块。预定义的数据流块类型处理所有的建立或释放链接的线程安全性。

有关连接数据流块以形成基本管道的示例，请参阅演练：创建数据流管道。有关连接数据流块以形成更复杂网络的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。有关在源向目标传递消息后从源取消目标链接的示例，请参阅如何：取消链接数据流块。

筛选

当您调用 ISourceBlock<TOutput>.LinkTo 方法将源链接到目标时，您可以根据消息的值提供一个委托来决定目标块是接受还是拒绝该消息。这种筛选机制很有用，它可以保证数据流块只接收特定值。对于大多数预定义的数据流块类型，如果源块连接到多个目标块，那么当目标块拒绝消息时，源将向下一个目标提供该消息。源向目标提供消息的顺序是按源定义的，可以根据源类型的不同而不同。一个目标接受消息后，大多数源块类型会停止提供该消息。此规则的例外情况是 BroadcastBlock<T> 类，这个类向所有目标提供每条消息，即使某些目标拒绝消息。有关使用筛选功能来仅处理特定消息的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

> **IMPORTANT**
> 由于每个预定义源数据流块类型确保了消息是按照它们接收的顺序来传播的，因此每一条消息都必须在源块可以处理下一条消息之前从源块读取。因此，当您使用筛选向一个源连接多个目标时，请确保至少一个目标块能够接收每一条消息。否则，您的应用程序可能发生死锁。

消息传递

数据流编程模型与 *消息传递* 这一概念相关，其中程序的独立组件通过发送消息相互通信。在应用组件间传播消息的一种方法是，调用 Post 和 DataflowBlock.SendAsync 方法，向目标数据流块发送消息（Post 同步运行，SendAsync 异步运行），再调用 Receive、ReceiveAsync 和 TryReceive 方法接收源数据流块发送的消息。您可以通过向头节点（目标块）发送输入数据，从管道的终端节点或网络的终端节点（一个或多个源块）接收输出数据来使用数据流管道或网络组合使用这些方法。您还可以使用 Choose 方法从提供的第一个拥有可用数据的源读取数据，并对该数据执行操作。

源数据流块通过调用方法 ITargetBlock<TInput>.OfferMessage 向目标数据流块提供数据。目标块通过以下三种方式之一来回应提供的消息：它可以接受消息，拒绝消息或推迟消息。当目标接受消息时，OfferMessage 方法会返回 Accepted。当目标拒绝消息时，OfferMessage 方法会返回 Declined。当目标要求它不再接收来自源的任何消息时，OfferMessage 会返回 DecliningPermanently。预定义的源块类型在这些返回值接收后不会向链接的目标提供消息，并且它们会自动取消这些目标的链接。

当目标块推迟消息以备日后使用时，OfferMessage 方法会返回 Postponed。推迟消息的目标块可以稍后调用 ISourceBlock<TOutput>.ReserveMessage 方法，以尝试暂留所提供的消息。此时，消息仍可用，并且可由该目标块使用，否则表明该消息已由另一个目标接收。如果目标数据流块稍后需要消息或不再需要消息，它会分别调用 ISourceBlock<TOutput>.ConsumeMessage 或 ReleaseReservation 方法。消息预留通常由以非贪婪模式运行的数据流块类型使用。非贪婪模式将在本文档的后面详细介绍。除了保留推迟的消息，目标块也可以使用 ISourceBlock<TOutput>.ConsumeMessage 方法来尝试直接使用推迟的消息。

数据流块完成

数据流块也支持完成概念。完成状态的数据流块不执行任何进一步的工作。每个数据流块都有相关的 System.Threading.Tasks.Task 对象（称为"完成任务"），表示数据流块的完成状态。因为您可以使用完成任务等待 Task 对象完成，所以您可以等待数据流网络的一个或更多终端节点来完成任务。IDataflowBlock 接口定义 Complete 方法（该方法向数据流块通知它完成的请求）和 Completion 属性（该属性返回数据流块的完成任务）。ISourceBlock<TOutput> 和 ITargetBlock<TInput> 都继承 IDataflowBlock 接口。

有两种方法来确定数据流块完成时是否没有出错、遇到一个或多个错误或已取消。第一种方法是在 `try` - `catch` 块（在 Visual Basic 中为 `Try` - `Catch`）中对完成任务调用 Task.Wait 方法。下面的示例创建一个 ActionBlock<TInput> 对象，该对象在其输入值小于零时会引发 ArgumentOutOfRangeException。当此示例在完成任务后调用 AggregateException 时，将引发 Wait。通过 ArgumentOutOfRangeException 对象的 InnerExceptions 属性来访问 AggregateException。

```
// Create an ActionBlock<int> object that prints its input
// and throws ArgumentOutOfRangeException if the input
// is less than zero.
var throwIfNegative = new ActionBlock<int>(n =>
{
   Console.WriteLine("n = {0}", n);
   if (n < 0)
   {
      throw new ArgumentOutOfRangeException();
   }
});

// Post values to the block.
throwIfNegative.Post(0);
throwIfNegative.Post(-1);
throwIfNegative.Post(1);
throwIfNegative.Post(-2);
throwIfNegative.Complete();

// Wait for completion in a try/catch block.
try
{
   throwIfNegative.Completion.Wait();
}
catch (AggregateException ae)
{
   // If an unhandled exception occurs during dataflow processing, all
   // exceptions are propagated through an AggregateException object.
   ae.Handle(e =>
   {
      Console.WriteLine("Encountered {0}: {1}",
         e.GetType().Name, e.Message);
      return true;
   });
}

/* Output:
n = 0
n = -1
Encountered ArgumentOutOfRangeException: Specified argument was out of the range
 of valid values.
*/
```

```
' Create an ActionBlock<int> object that prints its input
' and throws ArgumentOutOfRangeException if the input
' is less than zero.
Dim throwIfNegative = New ActionBlock(Of Integer)(Sub(n)
                                                      Console.WriteLine("n = {0}", n)
                                                      If n < 0 Then
                                                          Throw New ArgumentOutOfRangeException()
                                                      End If
                                                  End Sub)

' Post values to the block.
throwIfNegative.Post(0)
throwIfNegative.Post(-1)
throwIfNegative.Post(1)
throwIfNegative.Post(-2)
throwIfNegative.Complete()

' Wait for completion in a try/catch block.
Try
    throwIfNegative.Completion.Wait()
Catch ae As AggregateException
    ' If an unhandled exception occurs during dataflow processing, all
    ' exceptions are propagated through an AggregateException object.
    ae.Handle(Function(e)
                  Console.WriteLine("Encountered {0}: {1}", e.GetType().Name, e.Message)
                  Return True
              End Function)
End Try

'          Output:
'         n = 0
'         n = -1
'         Encountered ArgumentOutOfRangeException: Specified argument was out of the range
'          of valid values.
'
```

此示例演示在执行数据流块的委托中异常变成不可处理的情况。建议您在这样的块主体中处理异常。然而，你如果没能这么做，块就表现得好像是它被取消了，而且不会处理传入消息。

当显式取消数据流块时，AggregateException 对象在 OperationCanceledException 属性中包含 InnerExceptions。有关数据流取消的详细信息，请参阅启用取消部分。

第二种确定数据流块的完成状态的方法是使用延续执行完成任务，或者使用 C# 和 Visual Basic 的异步语言功能以异步方式等待完成任务。您提供给 Task.ContinueWith 方法的委托采用表示前面任务的 Task 对象。就 Completion 属性来说，延续的委托自行采用完成任务。下面的示例与前一个示例相似，不同之处在于它也使用 ContinueWith 方法创建输出整个数据流操作状态的延续任务。

```
// Create an ActionBlock<int> object that prints its input
// and throws ArgumentOutOfRangeException if the input
// is less than zero.
var throwIfNegative = new ActionBlock<int>(n =>
{
   Console.WriteLine("n = {0}", n);
   if (n < 0)
   {
      throw new ArgumentOutOfRangeException();
   }
});

// Create a continuation task that prints the overall
// task status to the console when the block finishes.
throwIfNegative.Completion.ContinueWith(task =>
{
   Console.WriteLine("The status of the completion task is '{0}'.",
      task.Status);
});

// Post values to the block.
throwIfNegative.Post(0);
throwIfNegative.Post(-1);
throwIfNegative.Post(1);
throwIfNegative.Post(-2);
throwIfNegative.Complete();

// Wait for completion in a try/catch block.
try
{
   throwIfNegative.Completion.Wait();
}
catch (AggregateException ae)
{
   // If an unhandled exception occurs during dataflow processing, all
   // exceptions are propagated through an AggregateException object.
   ae.Handle(e =>
   {
      Console.WriteLine("Encountered {0}: {1}",
         e.GetType().Name, e.Message);
      return true;
   });
}

/* Output:
n = 0
n = -1
The status of the completion task is 'Faulted'.
Encountered ArgumentOutOfRangeException: Specified argument was out of the range
 of valid values.
*/
```

```
' Create an ActionBlock<int> object that prints its input
' and throws ArgumentOutOfRangeException if the input
' is less than zero.
Dim throwIfNegative = New ActionBlock(Of Integer)(Sub(n)
                                                      Console.WriteLine("n = {0}", n)
                                                      If n < 0 Then
                                                          Throw New ArgumentOutOfRangeException()
                                                      End If
                                                  End Sub)

' Create a continuation task that prints the overall
' task status to the console when the block finishes.
throwIfNegative.Completion.ContinueWith(Sub(task) Console.WriteLine("The status of the completion task is
'{0}'.", task.Status))

' Post values to the block.
throwIfNegative.Post(0)
throwIfNegative.Post(-1)
throwIfNegative.Post(1)
throwIfNegative.Post(-2)
throwIfNegative.Complete()

' Wait for completion in a try/catch block.
Try
    throwIfNegative.Completion.Wait()
Catch ae As AggregateException
    ' If an unhandled exception occurs during dataflow processing, all
    ' exceptions are propagated through an AggregateException object.
    ae.Handle(Function(e)
                  Console.WriteLine("Encountered {0}: {1}", e.GetType().Name, e.Message)
                  Return True
              End Function)
End Try

'          Output:
'         n = 0
'         n = -1
'         The status of the completion task is 'Faulted'.
'         Encountered ArgumentOutOfRangeException: Specified argument was out of the range
'          of valid values.
'
```

您也可以使用类似延续任务主体中的属性（例如 IsCanceled）来确定有关数据流块的完成状态的其他信息。若要深入了解延续任务及其与取消和错误处理如何相关，请参阅使用延续任务链接任务、任务取消和异常处理。

预定义的数据流块类型

TPL 数据流库提供了多个预定义的数据流块类型。这些类型分为三个类别：缓冲块、执行块和分组块。以下部分描述了组成这些类别的块类型。

缓冲块

缓冲块存放的数据供数据使用者使用。TPL 数据流库提供三种缓冲块类型：System.Threading.Tasks.Dataflow.BufferBlock<T>、System.Threading.Tasks.Dataflow.BroadcastBlock<T> 和 System.Threading.Tasks.Dataflow.WriteOnceBlock<T>。

BufferBlock(T)

BufferBlock<T> 类表示一般用途的异步消息结构。此类存储先进先出 (FIFO) 消息队列，此消息队列可由多个源写入或从多个目标读取。在目标收到来自 BufferBlock<T> 对象的消息时，将从消息队列中删除此消息。因此，虽然一个 BufferBlock<T> 对象可以具有多个目标，但只有一个目标将接收每条消息。需将多条消息传递给另一个组件，且该组件必须接收每条消息时，BufferBlock<T> 类十分有用。

下面的基本示例将多个 Int32 值发送到 BufferBlock<T> 对象，然后从该对象读回这些值。

```
// Create a BufferBlock<int> object.
var bufferBlock = new BufferBlock<int>();

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   bufferBlock.Post(i);
}

// Receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(bufferBlock.Receive());
}

/* Output:
   0
   1
   2
 */
```

```
' Create a BufferBlock<int> object.
Dim bufferBlock = New BufferBlock(Of Integer)()

' Post several messages to the block.
For i As Integer = 0 To 2
    bufferBlock.Post(i)
Next i

' Receive the messages back from the block.
For i As Integer = 0 To 2
    Console.WriteLine(bufferBlock.Receive())
Next i

'          Output:
'            0
'            1
'            2
'
```

有关演示如何将消息写入到 BufferBlock<T> 对象并从该对象读取消息的完整示例，请参阅如何：将消息写入数据流块和从数据流块读取消息。

BroadcastBlock(T)

若您必须将多条消息传递给另一个组件，而该组件只需要最新的值，则 BroadcastBlock<T> 类很有用。需向多个组件广播消息时，此类也很有用。

下面的基本示例将 Double 值发送给 BroadcastBlock<T> 对象，然后多次从该对象读回该值。由于值在被读取之后不会从 BroadcastBlock<T> 对象中移除，因此每一次的可用值都相同。

```
// Create a BroadcastBlock<double> object.
var broadcastBlock = new BroadcastBlock<double>(null);

// Post a message to the block.
broadcastBlock.Post(Math.PI);

// Receive the messages back from the block several times.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(broadcastBlock.Receive());
}

/* Output:
   3.14159265358979
   3.14159265358979
   3.14159265358979
 */
```

```
' Create a BroadcastBlock<double> object.
Dim broadcastBlock = New BroadcastBlock(Of Double)(Nothing)

' Post a message to the block.
broadcastBlock.Post(Math.PI)

' Receive the messages back from the block several times.
For i As Integer = 0 To 2
    Console.WriteLine(broadcastBlock.Receive())
Next i

'          Output:
'            3.14159265358979
'            3.14159265358979
'            3.14159265358979
'
```

有关演示如何使用 BroadcastBlock<T> 来将一条消息广播给多个目标块的完整示例，请参阅如何：在数据流块中指定任务计划程序。

WriteOnceBlock(T)

WriteOnceBlock<T> 类与 BroadcastBlock<T> 类相似，不同之处在于 WriteOnceBlock<T> 对象仅可被写入一次。可以将 WriteOnceBlock<T> 视作类似于 C# 中的 readonly（Visual Basic 中的 ReadOnly）关键字，不同之处在于 WriteOnceBlock<T> 对象在收到值后（而不是在构造时）成为不可变对象。与 BroadcastBlock<T> 类相似，在目标收到来自 WriteOnceBlock<T> 对象的消息时，不会从该目标删除此消息。因此，多个目标将接收到该消息的副本。当您想要仅传播多条消息中的第一条时，WriteOnceBlock<T> 类很有用。

下面的基本示例将多个 String 值发送给 WriteOnceBlock<T> 对象，然后从该对象读回该值。由于 WriteOnceBlock<T> 对象在 WriteOnceBlock<T> 对象接收消息后只能写入一次，因此它放弃后续消息。

```
// Create a WriteOnceBlock<string> object.
var writeOnceBlock = new WriteOnceBlock<string>(null);

// Post several messages to the block in parallel. The first
// message to be received is written to the block.
// Subsequent messages are discarded.
Parallel.Invoke(
   () => writeOnceBlock.Post("Message 1"),
   () => writeOnceBlock.Post("Message 2"),
   () => writeOnceBlock.Post("Message 3"));

// Receive the message from the block.
Console.WriteLine(writeOnceBlock.Receive());

/* Sample output:
   Message 2
 */
```

```
' Create a WriteOnceBlock<string> object.
Dim writeOnceBlock = New WriteOnceBlock(Of String)(Nothing)

' Post several messages to the block in parallel. The first
' message to be received is written to the block.
' Subsequent messages are discarded.
Parallel.Invoke(Function() writeOnceBlock.Post("Message 1"), Function() writeOnceBlock.Post("Message 2"),
Function() writeOnceBlock.Post("Message 3"))

' Receive the message from the block.
Console.WriteLine(writeOnceBlock.Receive())

'          Sample output:
'            Message 2
'
```

有关演示如何使用 WriteOnceBlock<T> 接收完成的第一个操作值的完整示例，请参阅如何：取消链接数据流块。

执行块

执行块为每条接收数据调用用户提供的委托。TPL 数据流库提供三种执行块类型：ActionBlock<TInput>、System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput> 和 System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput>。

ActionBlock(T)

ActionBlock<TInput> 类在接收数据时是调用委托的目标块。将 ActionBlock<TInput> 对象视为数据可用时异步运行的委托。您提供给 ActionBlock<TInput> 对象的委托可以是类型 Action<T> 或类型 `System.Func<TInput, Task>`。当通过 ActionBlock<TInput> 使用 Action<T> 对象时，每个输入元素的处理在委托返回时视为已完成。当您通过 ActionBlock<TInput> 使用 `System.Func<TInput, Task>` 对象时，只有当返回的 Task 对象完成时，每个输入元素的处理才可以视为已完成。使用这两种机制，您可使用 ActionBlock<TInput> 同步和异步处理每个输入元素。

下面的基本示例将多个 Int32 值发送给 ActionBlock<TInput> 对象。ActionBlock<TInput> 对象将这些值输出到控制台中。然后此示例将该块设置为已完成状态，并等待所有数据流任务完成。

```
// Create an ActionBlock<int> object that prints values
// to the console.
var actionBlock = new ActionBlock<int>(n => Console.WriteLine(n));

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
   actionBlock.Post(i * 10);
}

// Set the block to the completed state and wait for all
// tasks to finish.
actionBlock.Complete();
actionBlock.Completion.Wait();

/* Output:
   0
   10
   20
 */
```

```
' Create an ActionBlock<int> object that prints values
' to the console.
Dim actionBlock = New ActionBlock(Of Integer)(Function(n) WriteLine(n))

' Post several messages to the block.
For i As Integer = 0 To 2
    actionBlock.Post(i * 10)
Next i

' Set the block to the completed state and wait for all
' tasks to finish.
actionBlock.Complete()
actionBlock.Completion.Wait()

'          Output:
'            0
'            10
'            20
'
```

有关演示如何在 ActionBlock<TInput> 类中使用委托的完整示例，请参阅如何：在数据流块收到数据时执行操作。

TransformBlock(TInput, TOutput)

TransformBlock<TInput,TOutput> 类与 ActionBlock<TInput> 类相似，不同之处在于它可以同时充当源和目标。传递给 TransformBlock<TInput,TOutput> 对象的委托返回类型为 `TOutput` 的值。您提供给 TransformBlock<TInput,TOutput> 对象的委托可以是类型 `System.Func<TInput, TOutput>` 或类型 `System.Func<TInput, Task<TOutput>>`。当您搭配使用 TransformBlock<TInput,TOutput> 和 `System.Func<TInput, TOutput>` 对象时，每个输入元素的处理在委托返回时视为已完成。当您搭配使用 TransformBlock<TInput,TOutput> 和 `System.Func<TInput, Task<TOutput>>` 对象时，只有当返回的 Task<TResult> 对象完成时，每个输入元素的处理才可以视为已完成。像 ActionBlock<TInput> 一样，通过使用这两种机制，您可使用 TransformBlock<TInput,TOutput> 同步和异步处理每个输入元素。

下面的基本示例所创建的 TransformBlock<TInput,TOutput> 对象用于计算输入的平方根。TransformBlock<TInput,TOutput> 对象采用 Int32 值作为输入并生成 Double 值作为输出。

```
// Create a TransformBlock<int, double> object that
// computes the square root of its input.
var transformBlock = new TransformBlock<int, double>(n => Math.Sqrt(n));

// Post several messages to the block.
transformBlock.Post(10);
transformBlock.Post(20);
transformBlock.Post(30);

// Read the output messages from the block.
for (int i = 0; i < 3; i++)
{
   Console.WriteLine(transformBlock.Receive());
}

/* Output:
   3.16227766016838
   4.47213595499958
   5.47722557505166
 */
```

```
' Create a TransformBlock<int, double> object that
' computes the square root of its input.
Dim transformBlock = New TransformBlock(Of Integer, Double)(Function(n) Math.Sqrt(n))

' Post several messages to the block.
transformBlock.Post(10)
transformBlock.Post(20)
transformBlock.Post(30)

' Read the output messages from the block.
For i As Integer = 0 To 2
    Console.WriteLine(transformBlock.Receive())
Next i

'          Output:
'            3.16227766016838
'            4.47213595499958
'            5.47722557505166
'
```

有关在数据流块网络(用于在 Windows 窗体应用程序中执行图像处理)中使用 TransformBlock<TInput,TOutput> 的完整示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

TransformManyBlock(TInput, TOutput)

TransformManyBlock<TInput,TOutput> 类与 TransformBlock<TInput,TOutput> 类相似，不同之处在于 TransformManyBlock<TInput,TOutput> 为每一个输入值生成零个或多个输出值，而不是为每个输入值仅生成一个输出值。您提供给 TransformManyBlock<TInput,TOutput> 对象的委托可以是类型 `System.Func<TInput, IEnumerable<TOutput>>` 或类型 `System.Func<TInput, Task<IEnumerable<TOutput>>>` 。当您搭配使用 TransformManyBlock<TInput,TOutput> 和 `System.Func<TInput, IEnumerable<TOutput>>` 对象时，每个输入元素的处理在委托返回时视为已完成。当您搭配使用 TransformManyBlock<TInput,TOutput> 和 `System.Func<TInput, Task<IEnumerable<TOutput>>>` 对象时，只有当返回的 `System.Threading.Tasks.Task<IEnumerable<TOutput>>` 对象完成时，每个输入元素的处理才可以视为已完成。

下面的基本示例所创建的 TransformManyBlock<TInput,TOutput> 对象将字符串拆分为单个字符序列。TransformManyBlock<TInput,TOutput> 对象采用 String 值作为输入并生成 Char 值作为输出。

```
// Create a TransformManyBlock<string, char> object that splits
// a string into its individual characters.
var transformManyBlock = new TransformManyBlock<string, char>(
   s => s.ToCharArray());

// Post two messages to the first block.
transformManyBlock.Post("Hello");
transformManyBlock.Post("World");

// Receive all output values from the block.
for (int i = 0; i < ("Hello" + "World").Length; i++)
{
   Console.WriteLine(transformManyBlock.Receive());
}

/* Output:
   H
   e
   l
   l
   o
   W
   o
   r
   l
   d
 */
```

```
' Create a TransformManyBlock<string, char> object that splits
' a string into its individual characters.
Dim transformManyBlock = New TransformManyBlock(Of String, Char)(Function(s) s.ToCharArray())

' Post two messages to the first block.
transformManyBlock.Post("Hello")
transformManyBlock.Post("World")

' Receive all output values from the block.
For i As Integer = 0 To ("Hello" & "World").Length - 1
    Console.WriteLine(transformManyBlock.Receive())
Next i

'          Output:
'            H
'            e
'            l
'            l
'            o
'            W
'            o
'            r
'            l
'            d
'
```

有关使用 TransformManyBlock<TInput,TOutput> 在一个数据流管道中为每一个输入生成多个独立输出的完整示例，请参阅演练：创建数据流管道。

并行度

每个 ActionBlock<TInput>、TransformBlock<TInput,TOutput> 和 TransformManyBlock<TInput,TOutput> 对象都缓冲输入消息，直到块准备处理它们。默认情况下，这些类以接收消息的顺序处理消息，一次处理一条消息。您还可以指定并行度，使 ActionBlock<TInput>、TransformBlock<TInput,TOutput> 和 TransformManyBlock<TInput,TOutput> 对象同时处理多条消息。有关并行执行的详细信息，请参阅本文档后面的“指定并行度”部分。有关设置并行度使执行数据流块能够一次处理多条消息的示例，请参阅如何：指定数据流

块中的并行度。

委托类型摘要

下表汇总了可提供给 ActionBlock<TInput>、TransformBlock<TInput,TOutput> 和 TransformManyBlock<TInput,TOutput> 对象的委托类型。此表还指出委托类型是同步执行还是异步执行。

[illegible]	[illegible]	[illegible]
ActionBlock<TInput>	`System.Action`	`System.Func<TInput, Task>`
TransformBlock<TInput,TOutput>	`System.Func<TInput, TOutput>`	`System.Func<TInput, Task<TOutput>>`
TransformManyBlock<TInput,TOutput>	`System.Func<TInput, IEnumerable<TOutput>>`	`System.Func<TInput, Task<IEnumerable<TOutput>>>`

当处理执行块类型时，还可以使用 lambda 表达式。有关演示如何使用 lambda 表达式处理执行块的示例，请参阅如何：在数据流块收到数据时执行操作。

分组块

分组块在各种约束下合并一个或多个源的数据。TPL 数据流库提供三种联接块类型：BatchBlock<T>、JoinBlock<T1,T2> 和 BatchedJoinBlock<T1,T2>。

BatchBlock(T)

BatchBlock<T> 类将一系列输入数据合并到输出数据数组，即批处理。在创建 BatchBlock<T> 对象时，指定每个批的大小。当 BatchBlock<T> 对象接收指定数量的输入元素时，它会异步传播含这些元素的数组。如果 BatchBlock<T> 对象设置为已完成状态，但不包含足够的元素形成批，则会传播包含其余输入元素的最终数组。

BatchBlock<T> 类可以在贪婪或非贪婪模式下运行。在默认贪婪模式下，BatchBlock<T> 对象接受它提供的每条消息，并在接收指定数量的元素后传播数组。在非贪婪模式下，BatchBlock<T> 对象推迟所有传入的消息，直到足够的源给块提供消息来形成批。贪婪模式处理开销较少，所以通常比非贪婪模式执行得更有效。但是，当您必须以基本方式协调来自多个源的消耗时，可以使用非贪婪模式。在 Greedy 构造函数的 `False` 参数中，通过将 `dataflowBlockOptions` 设置为 BatchBlock<T> 来指定非贪婪模式。

下面的基本示例将多个 Int32 值发送给一批中能容纳十个元素的 BatchBlock<T> 对象。为了确保所有值从 BatchBlock<T> 传播，此示例调用 Complete 方法。Complete 方法将 BatchBlock<T> 对象设置为已完成状态，因此，BatchBlock<T> 对象作为最终批传播剩余的所有元素。

```
// Create a BatchBlock<int> object that holds ten
// elements per batch.
var batchBlock = new BatchBlock<int>(10);

// Post several values to the block.
for (int i = 0; i < 13; i++)
{
   batchBlock.Post(i);
}
// Set the block to the completed state. This causes
// the block to propagate out any remaining
// values as a final batch.
batchBlock.Complete();

// Print the sum of both batches.

Console.WriteLine("The sum of the elements in batch 1 is {0}.",
   batchBlock.Receive().Sum());

Console.WriteLine("The sum of the elements in batch 2 is {0}.",
   batchBlock.Receive().Sum());

/* Output:
   The sum of the elements in batch 1 is 45.
   The sum of the elements in batch 2 is 33.
 */
```

```
' Create a BatchBlock<int> object that holds ten
' elements per batch.
Dim batchBlock = New BatchBlock(Of Integer)(10)

' Post several values to the block.
For i As Integer = 0 To 12
    batchBlock.Post(i)
Next i
' Set the block to the completed state. This causes
' the block to propagate out any remaining
' values as a final batch.
batchBlock.Complete()

' Print the sum of both batches.

Console.WriteLine("The sum of the elements in batch 1 is {0}.", batchBlock.Receive().Sum())

Console.WriteLine("The sum of the elements in batch 2 is {0}.", batchBlock.Receive().Sum())

'          Output:
'            The sum of the elements in batch 1 is 45.
'            The sum of the elements in batch 2 is 33.
'
```

有关使用 BatchBlock<T> 改进数据库插入操作效率的完整示例，请参阅演练：使用 BatchBlock 和 BatchedJoinBlock 提高效率。

JoinBlock(T1, T2, ...)

JoinBlock<T1,T2> 和 JoinBlock<T1,T2,T3> 类收集输入元素并传播包含这些元素的 System.Tuple<T1,T2> 或 System.Tuple<T1,T2,T3> 对象。JoinBlock<T1,T2> 和 JoinBlock<T1,T2,T3> 类不能从 ITargetBlock<TInput> 继承。而是提供属性 Target1、Target2 和 Target3 来实现 ITargetBlock<TInput>。

像在贪婪或非贪婪模式下运行的 BatchBlock<T>、JoinBlock<T1,T2> 和 JoinBlock<T1,T2,T3> 一样。在默认贪婪模式下，JoinBlock<T1,T2> 或 JoinBlock<T1,T2,T3> 对象在其每个目标接收至少一条消息之后接受提供的每条消息并传播元组。在非贪婪模式下，JoinBlock<T1,T2> 或 JoinBlock<T1,T2,T3> 对象推迟所有传入的消息，直到向任何目标提供了创建元组所需的数据。此时，块参与两阶段提交协议，以原子方式从源中检索所有必需的项。此

延迟使得其他实体可以同时使用数据，这使整个系统取得进展。

下面的基本示例演示 JoinBlock<T1,T2,T3> 对象需要多个数据来计算值的情况。此示例创建了需要两个 JoinBlock<T1,T2,T3> 值和一个 Int32 值来执行算术运算的 Char 对象。

```
// Create a JoinBlock<int, int, char> object that requires
// two numbers and an operator.
var joinBlock = new JoinBlock<int, int, char>();

// Post two values to each target of the join.

joinBlock.Target1.Post(3);
joinBlock.Target1.Post(6);

joinBlock.Target2.Post(5);
joinBlock.Target2.Post(4);

joinBlock.Target3.Post('+');
joinBlock.Target3.Post('-');

// Receive each group of values and apply the operator part
// to the number parts.

for (int i = 0; i < 2; i++)
{
   var data = joinBlock.Receive();
   switch (data.Item3)
   {
      case '+':
         Console.WriteLine("{0} + {1} = {2}",
            data.Item1, data.Item2, data.Item1 + data.Item2);
         break;
      case '-':
         Console.WriteLine("{0} - {1} = {2}",
            data.Item1, data.Item2, data.Item1 - data.Item2);
         break;
      default:
         Console.WriteLine("Unknown operator '{0}'.", data.Item3);
         break;
   }
}

/* Output:
   3 + 5 = 8
   6 - 4 = 2
 */
```

```
' Create a JoinBlock<int, int, char> object that requires
' two numbers and an operator.
Dim joinBlock = New JoinBlock(Of Integer, Integer, Char)()

' Post two values to each target of the join.

joinBlock.Target1.Post(3)
joinBlock.Target1.Post(6)

joinBlock.Target2.Post(5)
joinBlock.Target2.Post(4)

joinBlock.Target3.Post("+"c)
joinBlock.Target3.Post("-"c)

' Receive each group of values and apply the operator part
' to the number parts.

For i As Integer = 0 To 1
    Dim data = joinBlock.Receive()
    Select Case data.Item3
        Case "+"c
            Console.WriteLine("{0} + {1} = {2}", data.Item1, data.Item2, data.Item1 + data.Item2)
        Case "-"c
            Console.WriteLine("{0} - {1} = {2}", data.Item1, data.Item2, data.Item1 - data.Item2)
        Case Else
            Console.WriteLine("Unknown operator '{0}'.", data.Item3)
    End Select
Next i

'          Output:
'            3 + 5 = 8
'            6 - 4 = 2
'
```

有关在非贪婪模式下使用 JoinBlock<T1,T2> 对象合作共享资源的完整示例，请参阅如何：使用 JoinBlock 从多个源读取数据。

BatchedJoinBlock(T1, T2, ...)

BatchedJoinBlock<T1,T2> 和 BatchedJoinBlock<T1,T2,T3> 类收集各批输入元素，并传播包含这些元素的 `System.Tuple(IList(T1), IList(T2))` 或 `System.Tuple(IList(T1), IList(T2), IList(T3))` 对象。将 BatchedJoinBlock<T1,T2> 视为 BatchBlock<T> 和 JoinBlock<T1,T2> 的组合。在创建 BatchedJoinBlock<T1,T2> 对象时，指定每个批的大小。BatchedJoinBlock<T1,T2> 还提供了属性 Target1 和 Target2 来实现 ITargetBlock<TInput>。当从所有目标收到指定数量的输入元素时，BatchedJoinBlock<T1,T2> 对象会异步传播包含这些元素的 `System.Tuple(IList(T1), IList(T2))` 对象。

下面的基本示例创建了一个包含结果、BatchedJoinBlock<T1,T2> 值和 Int32 对象错误的 Exception 对象。此示例执行多个操作，然后将结果写入 Target1 属性，将错误写入 Target2 对象的 BatchedJoinBlock<T1,T2> 属性。由于成功和失败操作的计数事先是未知的，因此 IList<T> 对象使每个目标都能收到零个或多个值。

```
// For demonstration, create a Func<int, int> that
// returns its argument, or throws ArgumentOutOfRangeException
// if the argument is less than zero.
Func<int, int> DoWork = n =>
{
   if (n < 0)
      throw new ArgumentOutOfRangeException();
   return n;
};

// Create a BatchedJoinBlock<int, Exception> object that holds
// seven elements per batch.
var batchedJoinBlock = new BatchedJoinBlock<int, Exception>(7);

// Post several items to the block.
foreach (int i in new int[] { 5, 6, -7, -22, 13, 55, 0 })
{
   try
   {
      // Post the result of the worker to the
      // first target of the block.
      batchedJoinBlock.Target1.Post(DoWork(i));
   }
   catch (ArgumentOutOfRangeException e)
   {
      // If an error occurred, post the Exception to the
      // second target of the block.
      batchedJoinBlock.Target2.Post(e);
   }
}

// Read the results from the block.
var results = batchedJoinBlock.Receive();

// Print the results to the console.

// Print the results.
foreach (int n in results.Item1)
{
   Console.WriteLine(n);
}
// Print failures.
foreach (Exception e in results.Item2)
{
   Console.WriteLine(e.Message);
}

/* Output:
   5
   6
   13
   55
   0
   Specified argument was out of the range of valid values.
   Specified argument was out of the range of valid values.
 */
```

```
' For demonstration, create a Func<int, int> that
' returns its argument, or throws ArgumentOutOfRangeException
' if the argument is less than zero.
Dim DoWork As Func(Of Integer, Integer) = Function(n)
                                              If n < 0 Then
                                                  Throw New ArgumentOutOfRangeException()
                                              End If
                                              Return n
                                          End Function

' Create a BatchedJoinBlock<int, Exception> object that holds
' seven elements per batch.
Dim batchedJoinBlock = New BatchedJoinBlock(Of Integer, Exception)(7)

' Post several items to the block.
For Each i As Integer In New Integer() {5, 6, -7, -22, 13, 55, 0}
    Try
        ' Post the result of the worker to the
        ' first target of the block.
        batchedJoinBlock.Target1.Post(DoWork(i))
    Catch e As ArgumentOutOfRangeException
        ' If an error occurred, post the Exception to the
        ' second target of the block.
        batchedJoinBlock.Target2.Post(e)
    End Try
Next i

' Read the results from the block.
Dim results = batchedJoinBlock.Receive()

' Print the results to the console.

' Print the results.
For Each n As Integer In results.Item1
    Console.WriteLine(n)
Next n
' Print failures.
For Each e As Exception In results.Item2
    Console.WriteLine(e.Message)
Next e

'          Output:
'            5
'            6
'            13
'            55
'            0
'            Specified argument was out of the range of valid values.
'            Specified argument was out of the range of valid values.
'
```

有关使用 BatchedJoinBlock<T1,T2> 捕获程序从数据库中读取数据时发生的结果和任何异常的完整示例，请参阅演练：使用 BatchBlock 和 BatchedJoinBlock 提高效率。

配置数据流块行为

可以通过给数据流块类型的构造函数提供 System.Threading.Tasks.Dataflow.DataflowBlockOptions 对象来启用其他选项。这些选项控制这类管理基础任务和并行度的调度程序的行为。DataflowBlockOptions 还包含派生类型，用以指定特定于某些数据流块类型的行为。下表汇总了与每个数据流块类型相关的选项类型。

[illegible]	DATAFLOWBLOCKOPTIONS [illegible]
BufferBlock<T>	DataflowBlockOptions

类型	DataflowBlockOptions 类型
BroadcastBlock<T>	DataflowBlockOptions
WriteOnceBlock<T>	DataflowBlockOptions
ActionBlock<TInput>	ExecutionDataflowBlockOptions
TransformBlock<TInput,TOutput>	ExecutionDataflowBlockOptions
TransformManyBlock<TInput,TOutput>	ExecutionDataflowBlockOptions
BatchBlock<T>	GroupingDataflowBlockOptions
JoinBlock<T1,T2>	GroupingDataflowBlockOptions
BatchedJoinBlock<T1,T2>	GroupingDataflowBlockOptions

以下各节提供了有关重要数据流块选项(通过 System.Threading.Tasks.Dataflow.DataflowBlockOptions、System.Threading.Tasks.Dataflow.ExecutionDataflowBlockOptions 和 System.Threading.Tasks.Dataflow.GroupingDataflowBlockOptions 类提供)的其他信息。

指定任务计划程序

每个预定义的数据流块在数据可用时使用 TPL 任务计划机制执行一些活动，例如，将数据传播到目标、接收来自源的数据并运行用户定义的委托。TaskScheduler 是抽象类，表示将任务排队成线程的任务计划程序。默认任务计划程序 Default 使用 ThreadPool 类进行排队并执行工作。构造数据流块对象时，您可以通过设置 TaskScheduler 属性重写默认任务计划程序。

当同一个任务计划程序管理多个数据流块时，它可在它们之间强制实施策略。例如，如果多个数据流块分别配置为面向同一 ConcurrentExclusiveSchedulerPair 对象的独占计划程序，则会序列化这些块间运行的所有工作。同样，如果这些块配置为面向同一 ConcurrentExclusiveSchedulerPair 对象的并发计划程序，而该计划程序配置为具有最大并发级，则这些块中所有的工作都会受到并发操作数的限制。有关使用 ConcurrentExclusiveSchedulerPair 类启用并发读取操作，但以排除所有其他操作的独占方式执行写入操作的示例，请参阅如何：在数据流块中指定任务计划程序。有关 TPL 中的任务计划程序的详细信息，请参阅 TaskScheduler 类主题。

指定并行度

默认情况下，TPL 数据流库提供三种执行块类型(ActionBlock<TInput>、TransformBlock<TInput,TOutput> 和 TransformManyBlock<TInput,TOutput>)，一次处理一条消息。这些数据流块类型也会按照接收消息的顺序对消息进行处理。若要使这些数据流块同时处理该消息，请在构造数据流对象块时设置 ExecutionDataflowBlockOptions.MaxDegreeOfParallelism 属性。

MaxDegreeOfParallelism 的默认值为 1，这保证了数据流块一次处理一条消息。将该属性设置为大于 1 的值将使数据流块可以同时处理多条消息。将该属性设置为 DataflowBlockOptions.Unbounded 将使基础任务计划程序管理最大并发程度。

> **IMPORTANT**
> 当指定大于 1 的最大并行度时，会同时处理多条消息，因此，消息可能不会按照接收的顺序进行处理。然而，从块输出消息的顺序与接收消息的顺序相同。

由于 MaxDegreeOfParallelism 属性表示最大并行度，因此数据流块执行时的并行度可能小于指定的值。为了达到功能要求或因为缺少可用的系统资源，数据流块可能使用较小的并行度。数据流块选择的并行度不会超过您

指定的值。

MaxDegreeOfParallelism 属性的值对于每个数据流块对象而言，都是特有的。例如，如果四个数据流对象块中的每一个都指定 1 作为最大并行度，则所有四个数据流对象块可以并行运行。

有关设置最大并行度以启用并行冗长操作的示例，请参阅如何：指定数据流块中的并行度。

指定每个任务的消息数

预定义的数据流块类型使用任务来处理多个输入元素。这有助于最大限度地减少需要处理数据的任务对象数，从而使应用程序可以更有效地运行。但是，当一个数据流块集合中的任务处理数据时，其他数据流块的任务可能需要按照队列消息等待处理时间。若要使数据流任务更加公平，请设置 MaxMessagesPerTask 属性。当 MaxMessagesPerTask 设置为 DataflowBlockOptions.Unbounded 默认值时，数据流块使用的任务会处理尽可能多的消息。当 MaxMessagesPerTask 设置为 Unbounded 以外的值时，数据流块为每个 Task 对象至多处理这个数量的消息。虽然设置 MaxMessagesPerTask 属性可以提高任务间的公平性，但它可能会导致该系统创建多个非必要的任务，这会降低性能。

启用取消

TPL 提供了一种机制，能使任务以一种合作的方式协调取消。若要启用数据流块参与此取消机制，请设置 CancellationToken 属性。当此 CancellationToken 对象设置为已取消状态时，所有监视该标记的数据流块都会完成当前项目的执行，但不会开始处理后续项。这些数据流块也会清除所有缓冲的消息，释放所有源和目标块的连接，并转换为已取消状态。通过转换为已取消状态，Completion 属性具有设置为 Status 的 Canceled 属性，除非在处理过程中出现异常。在这种情况下，Status 会设置为 Faulted。

有关演示如何在 Windows 窗体应用程序中使用取消的示例，请参阅如何：取消数据流块。若要深入了解 TPL 中的取消，请参阅任务取消。

指定贪婪与非贪婪行为

几个分组数据流块类型可以在贪婪或非贪婪模式下运行。默认情况下，预定义的数据流块类型在贪婪模式下运行。

对于联接块类型（如 JoinBlock<T1,T2>），贪婪模式意味着块立即接受数据，即使相应的数据联接不可用。非贪婪模式意味着块推迟所有传入的消息，直到在其每个目标上有一个可完成联接。如果任何推迟的消息不再可用，则联接块会释放所有推迟的消息并重新启动该过程。对于 BatchBlock<T> 类，贪婪和非贪婪行为非常相似，不同之处在于在非贪婪模式下，BatchBlock<T> 对象推迟所有传入的消息，直到不同源中有足够消息可用于完成批作业。

若要为数据流块指定非贪婪模式，请将 Greedy 设置为 `False`。有关演示如何使用非贪婪模式使多个联接块更高效地共享数据源的示例，请参阅如何：使用 JoinBlock 从多个源读取数据。

自定义数据流块

尽管 TPL 数据流库提供了许多预定义块类型，但是您还是可以创建执行自定义行为的其他块类型。直接实现 ISourceBlock<TOutput> 或 ITargetBlock<TInput> 接口或使用 Encapsulate 方法生成封装现有块类型行为的复杂块。有关演示如何实现自定义数据流块功能的示例，请参阅演练：创建自定义数据流块类型。

相关主题

TITLE	
如何：将消息写入数据流块和从数据流块读取消息	演示如何向 BufferBlock<T> 对象写入和读取消息。
如何：实现制造者-使用者数据流模式	描述如何使用数据流模型实现制造者-使用方模式，在这个模型中制造者向数据流块发送消息，而使用方从该块中读取消息。

TITLE	描述
如何：在数据流块收到数据时执行操作	描述如何向执行数据流块类型（ActionBlock<TInput>、TransformBlock<TInput,TOutput> 和 TransformManyBlock<TInput,TOutput>）提供委托。
演练：创建数据流管道	描述如何创建从 Web 下载文本并对该文本执行操作的数据流管道。
如何：取消链接数据流块	展示了如何在源向目标提供消息后，使用 LinkTo 方法取消链接源数据流块和目标数据流块。
演练：在 Windows 窗体应用程序中使用数据流	演示如何创建在 Windows 窗体应用程序中执行图像处理的数据流块网络。
如何：取消数据流块	演示如何在 Windows 窗体应用程序中使用取消。
如何：使用 JoinBlock 从多个源读取数据	解释如何在多个源的数据可用时使用 JoinBlock<T1,T2> 类执行操作，以及如何使用非贪婪模式使多个联接块更有效地共享数据源。
如何：指定数据流块中的并行度	描述如何设置 MaxDegreeOfParallelism 属性使执行数据流块一次处理多条消息。
如何：在数据流块中指定任务计划程序	演示在应用程序中使用数据流时如何关联特定任务计划程序。
演练：使用 BatchBlock 和 BatchedJoinBlock 提高效率	描述如何使用 BatchBlock<T> 类改进数据库插入操作的效率，以及如何使用 BatchedJoinBlock<T1,T2> 类获取程序从数据库中读取数据时产生的结果和发生的任何异常。
演练：创建自定义数据流块类型	演示创建实现自定义行为的数据流块类型的两种方法。
任务并行库 (TPL)	介绍 TPL，一个可在 .NET Framework 应用程序里简化并行和并发编程的库。

如何：将消息写入数据流块和从数据流块读取消息

2021/11/16 •

本文介绍如何使用任务并行库 (TPL) 数据流库将消息写入数据流块和从数据流块读取消息。TPL 数据流库同时提供用于从数据流块写入和读取消息的同步和异步方法。本文介绍如何使用 System.Threading.Tasks.Dataflow.BufferBlock<T> 类。BufferBlock<T> 类可缓冲消息，既充当消息源，又充当消息目标。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

同步写入和读取

下面的示例使用 Post 方法写入 BufferBlock<T> 数据流块，使用 Receive 方法从同一对象读取。

```
var bufferBlock = new BufferBlock<int>();

// Post several messages to the block.
for (int i = 0; i < 3; i++)
{
    bufferBlock.Post(i);
}

// Receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(bufferBlock.Receive());
}

// Output:
//   0
//   1
//   2
```

```
Dim bufferBlock = New BufferBlock(Of Integer)()

' Post several messages to the block.
For i As Integer = 0 To 2
    bufferBlock.Post(i)
Next i

' Receive the messages back from the block.
For i As Integer = 0 To 2
    Console.WriteLine(bufferBlock.Receive())
Next i

' Output:
'   0
'   1
'   2
```

如以下示例所示，还可以使用 TryReceive 方法从数据流块读取。TryReceive 方法不会阻止当前线程，而且在偶尔轮询数据时非常有用。

```
// Post more messages to the block.
for (int i = 0; i < 3; i++)
{
    bufferBlock.Post(i);
}

// Receive the messages back from the block.
while (bufferBlock.TryReceive(out int value))
{
    Console.WriteLine(value);
}

// Output:
//   0
//   1
//   2
```

```
' Post more messages to the block.
For i As Integer = 0 To 2
    bufferBlock.Post(i)
Next i

' Receive the messages back from the block.
Dim value As Integer
Do While bufferBlock.TryReceive(value)
    Console.WriteLine(value)
Loop

' Output:
'   0
'   1
'   2
```

由于 Post 方法是同步运行的，前面示例中的 BufferBlock<T> 对象会在第二个循环读取数据之前接收所有数据。下面的示例对第一个示例进行了扩展，使用 Task.WhenAll(Task[]) 同时读取和写入消息块。由于 WhenAll 并发执行操作，因此不会按任何特定的顺序将值写入 BufferBlock<T> 对象。

```
// Write to and read from the message block concurrently.
var post01 = Task.Run(() =>
{
    bufferBlock.Post(0);
    bufferBlock.Post(1);
});
var receive = Task.Run(() =>
{
    for (int i = 0; i < 3; i++)
    {
        Console.WriteLine(bufferBlock.Receive());
    }
});
var post2 = Task.Run(() =>
{
    bufferBlock.Post(2);
});

await Task.WhenAll(post01, receive, post2);

// Output:
//   0
//   1
//   2
```

```
' Write to and read from the message block concurrently.
Dim post01 = Task.Run(Sub()
                          bufferBlock.Post(0)
                          bufferBlock.Post(1)
                      End Sub)
Dim receive = Task.Run(Sub()
                           For i As Integer = 0 To 2
                               Console.WriteLine(bufferBlock.Receive())
                           Next i
                       End Sub)
Dim post2 = Task.Run(Sub() bufferBlock.Post(2))
Task.WaitAll(post01, receive, post2)

' Output:
'   0
'   1
'   2
```

异步写入和读取

下面的示例使用 SendAsync 方法异步写入 BufferBlock<T> 对象，使用 ReceiveAsync 方法从同一对象异步读取。本示例使用 async 和 await 运算符（Visual Basic 中为 Async 和 Await）以异步方式向目标块发送数据以及从中读取数据。必须启用数据流块来推迟消息时，SendAsync 方法很有用。希望在数据可用时对此数据进行操作时，ReceiveAsync 方法很有用。有关消息在消息块之间如何传播的详细信息，请参阅数据流中的“消息传递”一节。

```
// Post more messages to the block asynchronously.
for (int i = 0; i < 3; i++)
{
    await bufferBlock.SendAsync(i);
}

// Asynchronously receive the messages back from the block.
for (int i = 0; i < 3; i++)
{
    Console.WriteLine(await bufferBlock.ReceiveAsync());
}

// Output:
//   0
//   1
//   2
```

```
' Post more messages to the block asynchronously.
For i As Integer = 0 To 2
    await bufferBlock.SendAsync(i)
Next i

' Asynchronously receive the messages back from the block.
For i As Integer = 0 To 2
    Console.WriteLine(await bufferBlock.ReceiveAsync())
Next i

' Output:
'   0
'   1
'   2
```

完整示例

以下示例显示本文中的所有代码。

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates a how to write to and read from a dataflow block.
class DataflowReadWrite
{
    // Demonstrates asynchronous dataflow operations.
    static async Task AsyncSendReceive(BufferBlock<int> bufferBlock)
    {
        // Post more messages to the block asynchronously.
        for (int i = 0; i < 3; i++)
        {
            await bufferBlock.SendAsync(i);
        }

        // Asynchronously receive the messages back from the block.
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(await bufferBlock.ReceiveAsync());
        }

        // Output:
        //   0
        //   1
        //   2
    }
```

```
    static async Task Main()
    {
        var bufferBlock = new BufferBlock<int>();

        // Post several messages to the block.
        for (int i = 0; i < 3; i++)
        {
            bufferBlock.Post(i);
        }

        // Receive the messages back from the block.
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(bufferBlock.Receive());
        }

        // Output:
        //   0
        //   1
        //   2

        // Post more messages to the block.
        for (int i = 0; i < 3; i++)
        {
            bufferBlock.Post(i);
        }

        // Receive the messages back from the block.
        while (bufferBlock.TryReceive(out int value))
        {
            Console.WriteLine(value);
        }

        // Output:
        //   0
        //   1
        //   2

        // Write to and read from the message block concurrently.
        var post01 = Task.Run(() =>
        {
            bufferBlock.Post(0);
            bufferBlock.Post(1);
        });
        var receive = Task.Run(() =>
        {
            for (int i = 0; i < 3; i++)
            {
                Console.WriteLine(bufferBlock.Receive());
            }
        });
        var post2 = Task.Run(() =>
        {
            bufferBlock.Post(2);
        });

        await Task.WhenAll(post01, receive, post2);

        // Output:
        //   0
        //   1
        //   2

        // Demonstrate asynchronous dataflow operations.
        await AsyncSendReceive(bufferBlock);
    }
}
```

```
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

' Demonstrates a how to write to and read from a dataflow block.
Friend Class DataflowReadWrite
    ' Demonstrates asynchronous dataflow operations.
    Private Shared async Function AsyncSendReceive(ByVal bufferBlock As BufferBlock(Of Integer)) As Task
        ' Post more messages to the block asynchronously.
        For i As Integer = 0 To 2
            await bufferBlock.SendAsync(i)
        Next i

        ' Asynchronously receive the messages back from the block.
        For i As Integer = 0 To 2
            Console.WriteLine(await bufferBlock.ReceiveAsync())
        Next i

        ' Output:
        '   0
        '   1
        '   2
    End Function

    Shared Sub Main(ByVal args() As String)
        Dim bufferBlock = New BufferBlock(Of Integer)()

        ' Post several messages to the block.
        For i As Integer = 0 To 2
            bufferBlock.Post(i)
        Next i

        ' Receive the messages back from the block.
        For i As Integer = 0 To 2
            Console.WriteLine(bufferBlock.Receive())
        Next i

        ' Output:
        '   0
        '   1
        '   2

        ' Post more messages to the block.
        For i As Integer = 0 To 2
            bufferBlock.Post(i)
        Next i

        ' Receive the messages back from the block.
        Dim value As Integer
        Do While bufferBlock.TryReceive(value)
            Console.WriteLine(value)
        Loop

        ' Output:
        '   0
        '   1
        '   2

        ' Write to and read from the message block concurrently.
        Dim post01 = Task.Run(Sub()
                                  bufferBlock.Post(0)
                                  bufferBlock.Post(1)
                              End Sub)
        Dim receive = Task.Run(Sub()
                                   For i As Integer = 0 To 2
                                       Console.WriteLine(bufferBlock.Receive())
                                   Next i
                               End Sub)
        Dim post2 = Task.Run(Sub() bufferBlock.Post(2))
```

```
        Task.WaitAll(post01, receive, post2)

        ' Output:
        '   0
        '   1
        '   2

        ' Demonstrate asynchronous dataflow operations.
        AsyncSendReceive(bufferBlock).Wait()
    End Sub

End Class
```

后续步骤

本示例演示如何直接从消息块读取和写入。还可以连接数据流块来形成管道（这是数据流块的线性序列）或网络（这是数据流块的图形）。在管道或网络中，当数据可用时源向目标异步传播数据。有关创建基本数据流管道的示例，请参阅演练：创建数据流管道。有关创建更复杂的数据流网络的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

另请参阅

- 数据流（任务并行库）

如何：实现生成方-使用方数据流模式

2021/11/16 •

本文介绍如何使用 TPL 数据流库实现生成方-使用方模式。在此模式下，制造者向消息块发送消息，使用者从该块读取消息。

> **NOTE**
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

示例

下面的示例演示使用数据流的基本制造者-使用者模型。`Produce` 方法将包含随机字节数据的数组写入 System.Threading.Tasks.Dataflow.ITargetBlock<TInput> 对象，而 `Consume` 方法从 System.Threading.Tasks.Dataflow.ISourceBlock<TOutput> 对象中读取字节。通过对 ISourceBlock<TOutput> 和 ITargetBlock<TInput> 接口（而非其派生类型）的操作，可以编写可对多种数据流块类型执行的可重用代码。此示例使用 BufferBlock<T> 类。由于 BufferBlock<T> 类既充当源块又充当目标块，制造者和使用者可以使用共享对象来传输数据。

`Produce` 方法在循环中调用 Post 方法，以将数据同步写入目标块。在 `Produce` 方法将所有数据写入目标块后，它将调用 Complete 方法来表明此块将不再有其他数据可用。`Consume` 方法使用 async 和 await 运算符（Visual Basic 中的 Async 和 Await），异步计算从 ISourceBlock<TOutput> 对象收到的总字节数。为了异步操作，`Consume` 方法会调用 OutputAvailableAsync 方法，以便在源块有可用数据和源块不再有其他可用数据时收到通知。

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

class DataflowProducerConsumer
{
    static void Produce(ITargetBlock<byte[]> target)
    {
        var rand = new Random();

        for (int i = 0; i < 100; ++ i)
        {
            var buffer = new byte[1024];
            rand.NextBytes(buffer);
            target.Post(buffer);
        }

        target.Complete();
    }

    static async Task<int> ConsumeAsync(ISourceBlock<byte[]> source)
    {
        int bytesProcessed = 0;

        while (await source.OutputAvailableAsync())
        {
            byte[] data = await source.ReceiveAsync();
            bytesProcessed += data.Length;
        }

        return bytesProcessed;
    }

    static async Task Main()
    {
        var buffer = new BufferBlock<byte[]>();
        var consumerTask = ConsumeAsync(buffer);
        Produce(buffer);

        var bytesProcessed = await consumerTask;

        Console.WriteLine($"Processed {bytesProcessed:#,#} bytes.");
    }
}

// Sample  output:
//     Processed 102,400 bytes.
```

```
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

Friend Class DataflowProducerConsumer
    Private Shared Sub Produce(ByVal target As ITargetBlock(Of Byte()))
        Dim rand As New Random()

        For i As Integer = 0 To 99
            Dim buffer(1023) As Byte
            rand.NextBytes(buffer)
            target.Post(buffer)
        Next i

        target.Complete()
    End Sub

    Private Shared Async Function ConsumeAsync(
        ByVal source As ISourceBlock(Of Byte())) As Task(Of Integer)
        Dim bytesProcessed As Integer = 0

        Do While Await source.OutputAvailableAsync()
            Dim data() As Byte = Await source.ReceiveAsync()
            bytesProcessed += data.Length
        Loop

        Return bytesProcessed
    End Function

    Shared Sub Main()
        Dim buffer = New BufferBlock(Of Byte())()
        Dim consumer = ConsumeAsync(buffer)
        Produce(buffer)

        Dim result = consumer.GetAwaiter().GetResult()

        Console.WriteLine($"Processed {result:#,#} bytes.")
    End Sub
End Class

' Sample output:
'     Processed 102,400 bytes.
```

可靠编程

前面的示例只使用一个使用者来处理源数据。如果您的应用程序中有多个使用者，请使用 TryReceive 方法从源块读取数据，如以下示例所示。

```
static async Task<int> ConsumeAsync(IReceivableSourceBlock<byte[]> source)
{
    int bytesProcessed = 0;
    while (await source.OutputAvailableAsync())
    {
        while (source.TryReceive(out byte[] data))
        {
            bytesProcessed += data.Length;
        }
    }
    return bytesProcessed;
}
```

```
Private Shared Async Function ConsumeAsync(
    ByVal source As IReceivableSourceBlock(Of Byte())) As Task(Of Integer)
    Dim bytesProcessed As Integer = 0

    Do While Await source.OutputAvailableAsync()
        Dim data() As Byte
        Do While source.TryReceive(data)
            bytesProcessed += data.Length
        Loop
    Loop

    Return bytesProcessed
End Function
```

没有可用数据时，TryReceive 方法将返回 `False` 。当多个使用者必须并发访问源块时，此机制可确保在调用 OutputAvailableAsync 后数据仍然可用。

请参阅

- 数据流

如何：在数据流块收到数据时执行操作

2021/11/16 •

在接收数据时，执行数据流块类型会调用用户提供的委托。System.Threading.Tasks.Dataflow.ActionBlock<TInput>、System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput> 和 System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput> 类是执行数据流块类型。当为执行数据流块提供工作函数时，可以使用 `delegate` 关键字（Visual Basic 中为 `Sub`）、Action<T>、Func<T,TResult> 或 lambda 表达式。本文档描述如何使用 Func<T,TResult> 和 lambda 表达式在执行块中执行操作。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

示例

下面的示例使用数据流从磁盘读取一个文件并计算该文件中等于零的字节的数量。它使用 TransformBlock<TInput,TOutput> 读取文件并计算零字节的数量，并使用 ActionBlock<TInput> 将零字节的数量输出到控制台。当块收到数据时，TransformBlock<TInput,TOutput> 对象会指定一个 Func<T,TResult> 对象来执行工作。ActionBlock<TInput> 对象使用 lambda 表达式将读取到的零字节的数量输出到控制台。

```
using System;
using System.IO;
using System.Linq;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to provide delegates to exectution dataflow blocks.
class DataflowExecutionBlocks
{
   // Computes the number of zero bytes that the provided file
   // contains.
   static int CountBytes(string path)
   {
      byte[] buffer = new byte[1024];
      int totalZeroBytesRead = 0;
      using (var fileStream = File.OpenRead(path))
      {
         int bytesRead = 0;
         do
         {
            bytesRead = fileStream.Read(buffer, 0, buffer.Length);
            totalZeroBytesRead += buffer.Count(b => b == 0);
         } while (bytesRead > 0);
      }

      return totalZeroBytesRead;
   }

   static void Main(string[] args)
   {
      // Create a temporary file on disk.
      string tempFile = Path.GetTempFileName();
```

```
      // Write random data to the temporary file.
      using (var fileStream = File.OpenWrite(tempFile))
      {
         Random rand = new Random();
         byte[] buffer = new byte[1024];
         for (int i = 0; i < 512; i++)
         {
            rand.NextBytes(buffer);
            fileStream.Write(buffer, 0, buffer.Length);
         }
      }

      // Create an ActionBlock<int> object that prints to the console
      // the number of bytes read.
      var printResult = new ActionBlock<int>(zeroBytesRead =>
      {
         Console.WriteLine("{0} contains {1} zero bytes.",
            Path.GetFileName(tempFile), zeroBytesRead);
      });

      // Create a TransformBlock<string, int> object that calls the
      // CountBytes function and returns its result.
      var countBytes = new TransformBlock<string, int>(
         new Func<string, int>(CountBytes));

      // Link the TransformBlock<string, int> object to the
      // ActionBlock<int> object.
      countBytes.LinkTo(printResult);

      // Create a continuation task that completes the ActionBlock<int>
      // object when the TransformBlock<string, int> finishes.
      countBytes.Completion.ContinueWith(delegate { printResult.Complete(); });

      // Post the path to the temporary file to the
      // TransformBlock<string, int> object.
      countBytes.Post(tempFile);

      // Requests completion of the TransformBlock<string, int> object.
      countBytes.Complete();

      // Wait for the ActionBlock<int> object to print the message.
      printResult.Completion.Wait();

      // Delete the temporary file.
      File.Delete(tempFile);
   }
}

/* Sample output:
tmp4FBE.tmp contains 2081 zero bytes.
*/
```

```
Imports System.IO
Imports System.Linq
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to provide delegates to exectution dataflow blocks.
Friend Class DataflowExecutionBlocks
    ' Computes the number of zero bytes that the provided file
    ' contains.
    Private Shared Function CountBytes(ByVal path As String) As Integer
        Dim buffer(1023) As Byte
        Dim totalZeroBytesRead As Integer = 0
        Using fileStream = File.OpenRead(path)
            Dim bytesRead As Integer = 0
```

```
            Do
                bytesRead = fileStream.Read(buffer, 0, buffer.Length)
                totalZeroBytesRead += buffer.Count(Function(b) b = 0)
            Loop While bytesRead > 0
        End Using

        Return totalZeroBytesRead
    End Function

    Shared Sub Main(ByVal args() As String)
        ' Create a temporary file on disk.
        Dim tempFile As String = Path.GetTempFileName()

        ' Write random data to the temporary file.
        Using fileStream = File.OpenWrite(tempFile)
            Dim rand As New Random()
            Dim buffer(1023) As Byte
            For i As Integer = 0 To 511
                rand.NextBytes(buffer)
                fileStream.Write(buffer, 0, buffer.Length)
            Next i
        End Using

        ' Create an ActionBlock<int> object that prints to the console
        ' the number of bytes read.
        Dim printResult = New ActionBlock(Of Integer)(Sub(zeroBytesRead) Console.WriteLine("{0} contains {1}
zero bytes.", Path.GetFileName(tempFile), zeroBytesRead))

        ' Create a TransformBlock<string, int> object that calls the
        ' CountBytes function and returns its result.
        Dim countBytes = New TransformBlock(Of String, Integer)(New Func(Of String, Integer)(AddressOf
DataflowExecutionBlocks.CountBytes))

        ' Link the TransformBlock<string, int> object to the
        ' ActionBlock<int> object.
        countBytes.LinkTo(printResult)

        ' Create a continuation task that completes the ActionBlock<int>
        ' object when the TransformBlock<string, int> finishes.
        countBytes.Completion.ContinueWith(Sub() printResult.Complete())

        ' Post the path to the temporary file to the
        ' TransformBlock<string, int> object.
        countBytes.Post(tempFile)

        ' Requests completion of the TransformBlock<string, int> object.
        countBytes.Complete()

        ' Wait for the ActionBlock<int> object to print the message.
        printResult.Completion.Wait()

        ' Delete the temporary file.
        File.Delete(tempFile)
    End Sub
End Class

' Sample output:
'tmp4FBE.tmp contains 2081 zero bytes.
'
```

虽然可以为 TransformBlock<TInput,TOutput> 对象提供 Lambda 表达式，但是本示例使用 Func<T,TResult> 来允许其他代码使用 `CountBytes` 方法。因为要执行的工作是此任务特有的，使用其他代码不可能有用，所以 ActionBlock<TInput> 对象使用 lambda 表达式。有关 lambda 表达式如何在任务并行库中工作的详细信息，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

Dataflow 文档中的“委托类型摘要”汇总了可以提供给 ActionBlock<TInput>、TransformBlock<TInput,TOutput>

和 TransformManyBlock<TInput,TOutput> 对象的委托类型。该表还指出委托类型是同步执行还是异步执行。

可靠编程

此示例为 Func<T,TResult> 对象提供类型 TransformBlock<TInput,TOutput> 的委托，以同步执行数据流块的任务。为了使数据流块异步执行操作，请为数据流块提供类型 Func<TResult> 的委托。当数据流块异步执行操作时，数据流块的任务只有在返回的 Task<TResult> 对象完成时才会完成。下面的示例修改了 `CountBytes` 方法，并使用 async 和 await 运算符（Visual Basic 中为 Async 和 Await）异步计算所提供文件中为零的字节的总数。ReadAsync 方法异步执行文件读取操作。

```
// Asynchronously computes the number of zero bytes that the provided file
// contains.
static async Task<int> CountBytesAsync(string path)
{
   byte[] buffer = new byte[1024];
   int totalZeroBytesRead = 0;
   using (var fileStream = new FileStream(
      path, FileMode.Open, FileAccess.Read, FileShare.Read, 0x1000, true))
   {
      int bytesRead = 0;
      do
      {
         // Asynchronously read from the file stream.
         bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length);
         totalZeroBytesRead += buffer.Count(b => b == 0);
      } while (bytesRead > 0);
   }

   return totalZeroBytesRead;
}
```

```
' Asynchronously computes the number of zero bytes that the provided file
' contains.
Private Shared async Function CountBytesAsync(ByVal path As String) As Task(Of Integer)
    Dim buffer(1023) As Byte
    Dim totalZeroBytesRead As Integer = 0
    Using fileStream = New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, &H1000, True)
        Dim bytesRead As Integer = 0
        Do
            ' Asynchronously read from the file stream.
            bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length)
            totalZeroBytesRead += buffer.Count(Function(b) b = 0)
        Loop While bytesRead > 0
    End Using

    Return totalZeroBytesRead
End Function
```

还可以使用异步的 lambda 表达式在执行数据流块中执行操作。下面的示例修改了上一示例中使用的 TransformBlock<TInput,TOutput> 对象，以便使用 lambda 表达式异步执行工作。

```
// Create a TransformBlock<string, int> object that calls the
// CountBytes function and returns its result.
var countBytesAsync = new TransformBlock<string, int>(async path =>
{
   byte[] buffer = new byte[1024];
   int totalZeroBytesRead = 0;
   using (var fileStream = new FileStream(
      path, FileMode.Open, FileAccess.Read, FileShare.Read, 0x1000, true))
   {
      int bytesRead = 0;
      do
      {
         // Asynchronously read from the file stream.
         bytesRead = await fileStream.ReadAsync(buffer, 0, buffer.Length);
         totalZeroBytesRead += buffer.Count(b => b == 0);
      } while (bytesRead > 0);
   }

   return totalZeroBytesRead;
});
```

```
' Create a TransformBlock<string, int> object that calls the
' CountBytes function and returns its result.
Dim countBytesAsync = New TransformBlock(Of String, Integer)(async Function(path)
                                                                 ' Asynchronously read from the file stream.
                                                                 Dim buffer(1023) As Byte
                                                                 Dim totalZeroBytesRead As Integer = 0
                                                                 Using fileStream = New FileStream(path,
FileMode.Open, FileAccess.Read, FileShare.Read, &H1000, True)

                                                                     Dim bytesRead As Integer = 0
                                                                     Do
                                                                         bytesRead = await
fileStream.ReadAsync(buffer, 0, buffer.Length)

                                                                         totalZeroBytesRead +=
buffer.Count(Function(b) b = 0)

                                                                     Loop While bytesRead > 0
                                                                 End Using
                                                                 Return totalZeroBytesRead
                                                             End Function)
```

另请参阅

- 数据流

演练：创建数据流管道

2021/11/16 •

尽管可以使用 DataflowBlock.Receive、DataflowBlock.ReceiveAsync 和 DataflowBlock.TryReceive 方法从源块接收消息，但也可以连接消息块来形成一个 *数据流管道*。数据流管道是一系列组件或"数据流块"，每个组件或数据流块执行一个有助于实现更大目标的特定任务。数据流管道中的每个数据流块会在收到来自另一数据流块的消息时执行工作。这就好比是汽车制造装配线。每辆汽车通过装配线时，一站组装车架，下一站则安装引擎，以此类推。因为装配线可以同时装配多辆汽车，所以比一次装配整辆车拥有更高的产出。

本文档演示了一个数据流管道，用于从网站上下载书籍《The Iliad of Homer》并搜索文本以将各个单词与反转第一个单词字符的单词相匹配。本文档中数据流管道的形成包括以下步骤：

1. 创建参与管道的数据流块。
2. 连接每个数据流块与管道中的下一个块。每个块将管道中前一个块的输出作为输入接收。
3. 对每个数据流块，创建一个延续任务，该延续任务在上一个块完成后将下一个块的状态设置为已完成状态。
4. 将数据发布到管道的开头。
5. 将管道的开头标记为已完成。
6. 等待管道完成所有工作。

先决条件

开始本演练之前，请阅读数据流。

创建控制台应用程序

在 Visual Studio 中，创建 Visual C# 或 Visual Basic"控制台应用程序"项目。安装 System.Threading.Tasks.Dataflow NuGet 包。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择"项目"菜单中的"管理 NuGet 包"，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

将以下代码添加到项目中以创建基本应用程序。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a basic dataflow pipeline.
// This program downloads the book "The Iliad of Homer" by Homer from the Web
// and finds all reversed words that appear in that book.
static class Program
{
   static void Main()
   {
   }
}
```

```
Imports System.Net.Http
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a basic dataflow pipeline.
' This program downloads the book "The Iliad of Homer" by Homer from the Web
' and finds all reversed words that appear in that book.
Module DataflowReversedWords

    Sub Main()
    End Sub

End Module
```

创建数据流块

将以下代码添加到 `Main` 方法以创建参与管道的数据流块。下表总结了管道的每个成员的角色。

```
//
// Create the members of the pipeline.
//

// Downloads the requested resource as a string.
var downloadString = new TransformBlock<string, string>(async uri =>
{
   Console.WriteLine("Downloading '{0}'...", uri);

   return await new HttpClient(new HttpClientHandler{ AutomaticDecompression =
System.Net.DecompressionMethods.GZip }).GetStringAsync(uri);
});

// Separates the specified text into an array of words.
var createWordList = new TransformBlock<string, string[]>(text =>
{
   Console.WriteLine("Creating word list...");

   // Remove common punctuation by replacing all non-letter characters
   // with a space character.
   char[] tokens = text.Select(c => char.IsLetter(c) ? c : ' ').ToArray();
   text = new string(tokens);

   // Separate the text into an array of words.
   return text.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
});

// Removes short words and duplicates.
var filterWordList = new TransformBlock<string[], string[]>(words =>
{
   Console.WriteLine("Filtering word list...");

   return words
      .Where(word => word.Length > 3)
      .Distinct()
      .ToArray();
});

// Finds all words in the specified collection whose reverse also
// exists in the collection.
var findReversedWords = new TransformManyBlock<string[], string>(words =>
{
   Console.WriteLine("Finding reversed words...");

   var wordsSet = new HashSet<string>(words);

   return from word in words.AsParallel()
          let reverse = new string(word.Reverse().ToArray())
          where word != reverse && wordsSet.Contains(reverse)
          select word;
});

// Prints the provided reversed words to the console.
var printReversedWords = new ActionBlock<string>(reversedWord =>
{
   Console.WriteLine("Found reversed words {0}/{1}",
      reversedWord, new string(reversedWord.Reverse().ToArray()));
});
```

```
'
' Create the members of the pipeline.
'

' Downloads the requested resource as a string.
Dim downloadString = New TransformBlock(Of String, String)(
    Async Function(uri)
        Console.WriteLine("Downloading '{0}'...", uri)

        Return Await New HttpClient().GetStringAsync(uri)
    End Function)

' Separates the specified text into an array of words.
Dim createWordList = New TransformBlock(Of String, String())(
   Function(text)
       Console.WriteLine("Creating word list...")

     ' Remove common punctuation by replacing all non-letter characters
     ' with a space character.
     Dim tokens() As Char = text.Select(Function(c) If(Char.IsLetter(c), c, " "c)).ToArray()
       text = New String(tokens)

     ' Separate the text into an array of words.
     Return text.Split(New Char() {" "c}, StringSplitOptions.RemoveEmptyEntries)
   End Function)

' Removes short words and duplicates.
Dim filterWordList = New TransformBlock(Of String(), String())(
   Function(words)
       Console.WriteLine("Filtering word list...")

       Return words.Where(Function(word) word.Length > 3).Distinct().ToArray()
   End Function)

' Finds all words in the specified collection whose reverse also
' exists in the collection.
Dim findReversedWords = New TransformManyBlock(Of String(), String)(
   Function(words)

       Dim wordsSet = New HashSet(Of String)(words)

       Return From word In words.AsParallel()
              Let reverse = New String(word.Reverse().ToArray())
              Where word <> reverse AndAlso wordsSet.Contains(reverse)
              Select word
   End Function)

' Prints the provided reversed words to the console.
Dim printReversedWords = New ActionBlock(Of String)(
   Sub(reversedWord)
       Console.WriteLine("Found reversed words {0}/{1}", reversedWord, New
String(reversedWord.Reverse().ToArray()))
   End Sub)
```

|
 |
 |
 |
| --- | --- | --- |
| `downloadString` | TransformBlock<TInput,TOutput> | 从 Web 下载该书的文本。 |
| `createWordList` | TransformBlock<TInput,TOutput> | 将该书的文本分成单词的数组。 |
| `filterWordList` | TransformBlock<TInput,TOutput> | 删除短词和单词数组中的重复项。 |

“	“	“
`findReversedWords`	TransformManyBlock<TInput,TOutput>	查找经过筛选的单词数组集合中所有将字母反转后仍在单词数组中出现的单词。
`printReversedWords`	ActionBlock<TInput>	向控制台显示单词及对应的倒序词。

虽然可以将本示例中数据流管道的多个步骤合并为一个步骤，不过本示例阐释了组合多个独立数据流任务来执行较大任务的概念。示例通过使用 TransformBlock<TInput,TOutput> 使管道的每个成员能对其输入数据执行操作并将结果发送到管道中的下一步骤。管道的 `findReversedWords` 成员是一个 TransformManyBlock<TInput,TOutput> 对象，因为该成员会为每个输入生成多个独立输出。管道的结尾 `printReversedWords` 是一个 ActionBlock<TInput> 对象，因为它会对其输入执行一个操作，但不产生结果。

形成管道

添加以下代码将每个块与管道中的下一个块连接。

当您调用 LinkTo 方法将源数据流块连接到目标数据流块时，源数据流块会在数据可用时将数据传播到目标块。如果你还提供 DataflowLinkOptions，并将 PropagateCompletion 设置为 true，则在管道中成功或未成功完成一个块都将导致管道中下一个块的完成。

```
//
// Connect the dataflow blocks to form a pipeline.
//

var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };

downloadString.LinkTo(createWordList, linkOptions);
createWordList.LinkTo(filterWordList, linkOptions);
filterWordList.LinkTo(findReversedWords, linkOptions);
findReversedWords.LinkTo(printReversedWords, linkOptions);
```

```
'
' Connect the dataflow blocks to form a pipeline.
'

Dim linkOptions = New DataflowLinkOptions With {.PropagateCompletion = True}

downloadString.LinkTo(createWordList, linkOptions)
createWordList.LinkTo(filterWordList, linkOptions)
filterWordList.LinkTo(findReversedWords, linkOptions)
findReversedWords.LinkTo(printReversedWords, linkOptions)
```

将数据发布到管道

添加以下代码，以将《The Iliad of Homer》一书的 URL 发布到数据流管道的开头。

```
// Process "The Iliad of Homer" by Homer.
downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt");
```

```
' Process "The Iliad of Homer" by Homer.
downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt")
```

此示例使用 DataflowBlock.Post 将数据同步发送到管道的开头。在必须将数据异步发送到数据流节点时，请使用

DataflowBlock.SendAsync 方法。

完成管道活动

添加以下代码将管道的开头标记为已完成。管道的开头在处理了所有缓冲的消息后传播其完成。

```
// Mark the head of the pipeline as complete.
downloadString.Complete();
```

```
' Mark the head of the pipeline as complete.
downloadString.Complete()
```

此示例通过要处理的数据流管道发送一个 URL。如果要通过管道发送多个输入，请在提交了所有输入后调用 IDataflowBlock.Complete 方法。如果您的应用程序没有表示数据不再可用或应用程序不必等待管道完成的定义完善的点，则可以忽略此步骤。

等待管道完成

添加以下代码以等待管道完成。管道的结尾完成时完成整个操作。

```
// Wait for the last block in the pipeline to process all messages.
printReversedWords.Completion.Wait();
```

```
' Wait for the last block in the pipeline to process all messages.
printReversedWords.Completion.Wait()
```

可以同时等待任一线程或多个线程的数据流完成。

完整示例

下面的示例显示此演练的完整代码。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a basic dataflow pipeline.
// This program downloads the book "The Iliad of Homer" by Homer from the Web
// and finds all reversed words that appear in that book.
static class DataflowReversedWords
{
   static void Main()
   {
      //
      // Create the members of the pipeline.
      //

      // Downloads the requested resource as a string.
      var downloadString = new TransformBlock<string, string>(async uri =>
      {
         Console.WriteLine("Downloading '{0}'...", uri);

         return await new HttpClient(new HttpClientHandler{ AutomaticDecompression =
System.Net.DecompressionMethods.GZip }).GetStringAsync(uri);
      });
```

```
      // Separates the specified text into an array of words.
      var createWordList = new TransformBlock<string, string[]>(text =>
      {
         Console.WriteLine("Creating word list...");

         // Remove common punctuation by replacing all non-letter characters
         // with a space character.
         char[] tokens = text.Select(c => char.IsLetter(c) ? c : ' ').ToArray();
         text = new string(tokens);

         // Separate the text into an array of words.
         return text.Split(new[] { ' ' }, StringSplitOptions.RemoveEmptyEntries);
      });

      // Removes short words and duplicates.
      var filterWordList = new TransformBlock<string[], string[]>(words =>
      {
         Console.WriteLine("Filtering word list...");

         return words
            .Where(word => word.Length > 3)
            .Distinct()
            .ToArray();
      });

      // Finds all words in the specified collection whose reverse also
      // exists in the collection.
      var findReversedWords = new TransformManyBlock<string[], string>(words =>
      {
         Console.WriteLine("Finding reversed words...");

         var wordsSet = new HashSet<string>(words);

         return from word in words.AsParallel()
                let reverse = new string(word.Reverse().ToArray())
                where word != reverse && wordsSet.Contains(reverse)
                select word;
      });

      // Prints the provided reversed words to the console.
      var printReversedWords = new ActionBlock<string>(reversedWord =>
      {
         Console.WriteLine("Found reversed words {0}/{1}",
            reversedWord, new string(reversedWord.Reverse().ToArray()));
      });

      //
      // Connect the dataflow blocks to form a pipeline.
      //

      var linkOptions = new DataflowLinkOptions { PropagateCompletion = true };

      downloadString.LinkTo(createWordList, linkOptions);
      createWordList.LinkTo(filterWordList, linkOptions);
      filterWordList.LinkTo(findReversedWords, linkOptions);
      findReversedWords.LinkTo(printReversedWords, linkOptions);

      // Process "The Iliad of Homer" by Homer.
      downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt");

      // Mark the head of the pipeline as complete.
      downloadString.Complete();

      // Wait for the last block in the pipeline to process all messages.
      printReversedWords.Completion.Wait();
   }
}
/* Sample output:
```

```
   Downloading 'http://www.gutenberg.org/cache/epub/16452/pg16452.txt'...
   Creating word list...
   Filtering word list...
   Finding reversed words...
   Found reversed words doom/mood
   Found reversed words draw/ward
   Found reversed words aera/area
   Found reversed words seat/taes
   Found reversed words live/evil
   Found reversed words port/trop
   Found reversed words sleek/keels
   Found reversed words area/aera
   Found reversed words tops/spot
   Found reversed words evil/live
   Found reversed words mood/doom
   Found reversed words speed/deeps
   Found reversed words moor/room
   Found reversed words trop/port
   Found reversed words spot/tops
   Found reversed words spots/stops
   Found reversed words stops/spots
   Found reversed words reed/deer
   Found reversed words keels/sleek
   Found reversed words deeps/speed
   Found reversed words deer/reed
   Found reversed words taes/seat
   Found reversed words room/moor
   Found reversed words ward/draw
*/
```

```
Imports System.Net.Http
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a basic dataflow pipeline.
' This program downloads the book "The Iliad of Homer" by Homer from the Web
' and finds all reversed words that appear in that book.
Module DataflowReversedWords

    Sub Main()
        '
        ' Create the members of the pipeline.
        '

        ' Downloads the requested resource as a string.
        Dim downloadString = New TransformBlock(Of String, String)(
            Async Function(uri)
                Console.WriteLine("Downloading '{0}'...", uri)

                Return Await New HttpClient().GetStringAsync(uri)
            End Function)

        ' Separates the specified text into an array of words.
        Dim createWordList = New TransformBlock(Of String, String())(
           Function(text)
               Console.WriteLine("Creating word list...")

             ' Remove common punctuation by replacing all non-letter characters
             ' with a space character.
             Dim tokens() As Char = text.Select(Function(c) If(Char.IsLetter(c), c, " "c)).ToArray()
               text = New String(tokens)

             ' Separate the text into an array of words.
             Return text.Split(New Char() {" "c}, StringSplitOptions.RemoveEmptyEntries)
           End Function)

        ' Removes short words and duplicates.
        Dim filterWordList = New TransformBlock(Of String(), String())(
```

```
            Function(words)
                Console.WriteLine("Filtering word list...")

                Return words.Where(Function(word) word.Length > 3).Distinct().ToArray()
            End Function)

        ' Finds all words in the specified collection whose reverse also
        ' exists in the collection.
        Dim findReversedWords = New TransformManyBlock(Of String(), String)(
            Function(words)

                Dim wordsSet = New HashSet(Of String)(words)

                Return From word In words.AsParallel()
                       Let reverse = New String(word.Reverse().ToArray())
                       Where word <> reverse AndAlso wordsSet.Contains(reverse)
                       Select word
            End Function)

        ' Prints the provided reversed words to the console.
        Dim printReversedWords = New ActionBlock(Of String)(
            Sub(reversedWord)
                Console.WriteLine("Found reversed words {0}/{1}", reversedWord, New
String(reversedWord.Reverse().ToArray()))
            End Sub)

        '
        ' Connect the dataflow blocks to form a pipeline.
        '

        Dim linkOptions = New DataflowLinkOptions With {.PropagateCompletion = True}

        downloadString.LinkTo(createWordList, linkOptions)
        createWordList.LinkTo(filterWordList, linkOptions)
        filterWordList.LinkTo(findReversedWords, linkOptions)
        findReversedWords.LinkTo(printReversedWords, linkOptions)

        ' Process "The Iliad of Homer" by Homer.
        downloadString.Post("http://www.gutenberg.org/cache/epub/16452/pg16452.txt")

        ' Mark the head of the pipeline as complete.
        downloadString.Complete()

        ' Wait for the last block in the pipeline to process all messages.
        printReversedWords.Completion.Wait()
    End Sub

End Module

' Sample output:
'Downloading 'http://www.gutenberg.org/cache/epub/16452/pg16452.txt'...
'Creating word list...
'Filtering word list...
'Finding reversed words...
'Found reversed words aera/area
'Found reversed words doom/mood
'Found reversed words draw/ward
'Found reversed words live/evil
'Found reversed words seat/taes
'Found reversed words area/aera
'Found reversed words port/trop
'Found reversed words sleek/keels
'Found reversed words tops/spot
'Found reversed words evil/live
'Found reversed words speed/deeps
'Found reversed words mood/doom
'Found reversed words moor/room
'Found reversed words spot/tops
'Found reversed words spots/stops
```

```
'Found reversed words trop/port
'Found reversed words stops/spots
'Found reversed words reed/deer
'Found reversed words deeps/speed
'Found reversed words deer/reed
'Found reversed words taes/seat
'Found reversed words keels/sleek
'Found reversed words room/moor
'Found reversed words ward/draw
```

后续步骤

此示例发送一个通过数据流管道处理的 URL。如果要通过管道发送多个输入值，可以将并行的形式引入应用程序，这与零件在汽车厂中移动的方式类似。当管道的第一个成员将其结果发送给第二个成员时，它可以在第二个成员处理第一个结果时并行处理另一个项。

通过使用数据流管道实现的并行称为 *粗粒度并行*，因为它通常由几个较大的任务组成。此外，你也可以在数据流管道中对短时间运行的较小任务使用 *粒度较细的并行*。在本示例中，管道的 `findReversedWords` 成员使用 PLINQ 并行处理工作列表中的多个项。在粗粒度的管道中使用细粒度并行可以提高总吞吐量。

另外，还可以将数据流块连接到多个目标块，以创建"数据流网络"。LinkTo 方法采用一个 Predicate<T> 对象，该对象定义了目标块是否根据其值来接受每个消息。大多数充当源的数据流块类型按目标块连接的顺序向所有已连接的目标块提供消息，直到其中一个块接受此消息。通过使用此筛选机制，您可以创建已连接数据流块的系统，指示某些数据通过一条路径，其他数据通过另一条路径。有关使用筛选来创建数据流网络的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

请参阅

- 数据流

如何：取消链接数据流块

2021/11/16 •

本文档介绍如何取消目标数据流块与其源的链接。

> **NOTE**
> TPL 数据流库(System.Threading.Tasks.Dataflow 命名空间)不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

示例

下面的示例创建了三个 TransformBlock<TInput,TOutput> 对象，每个对象调用 `TrySolution` 方法来计算值。此示例只需使用第一次调用 `TrySolution` 的结果即可完成。

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to unlink dataflow blocks.
class DataflowReceiveAny
{
   // Receives the value from the first provided source that has
   // a message.
   public static T ReceiveFromAny<T>(params ISourceBlock<T>[] sources)
   {
      // Create a WriteOnceBlock<T> object and link it to each source block.
      var writeOnceBlock = new WriteOnceBlock<T>(e => e);
      foreach (var source in sources)
      {
         // Setting MaxMessages to one instructs
         // the source block to unlink from the WriteOnceBlock<T> object
         // after offering the WriteOnceBlock<T> object one message.
         source.LinkTo(writeOnceBlock, new DataflowLinkOptions { MaxMessages = 1 });
      }
      // Return the first value that is offered to the WriteOnceBlock object.
      return writeOnceBlock.Receive();
   }

   // Demonstrates a function that takes several seconds to produce a result.
   static int TrySolution(int n, CancellationToken ct)
   {
      // Simulate a lengthy operation that completes within three seconds
      // or when the provided CancellationToken object is cancelled.
      SpinWait.SpinUntil(() => ct.IsCancellationRequested,
         new Random().Next(3000));

      // Return a value.
      return n + 42;
   }

   static void Main(string[] args)
   {
      // Create a shared CancellationTokenSource object to enable the
      // TrySolution method to be cancelled.
      var cts = new CancellationTokenSource();
```

```
      // Create three TransformBlock<int, int> objects.
      // Each TransformBlock<int, int> object calls the TrySolution method.
      Func<int, int> action = n => TrySolution(n, cts.Token);
      var trySolution1 = new TransformBlock<int, int>(action);
      var trySolution2 = new TransformBlock<int, int>(action);
      var trySolution3 = new TransformBlock<int, int>(action);

      // Post data to each TransformBlock<int, int> object.
      trySolution1.Post(11);
      trySolution2.Post(21);
      trySolution3.Post(31);

      // Call the ReceiveFromAny<T> method to receive the result from the
      // first TransformBlock<int, int> object to finish.
      int result = ReceiveFromAny(trySolution1, trySolution2, trySolution3);

      // Cancel all calls to TrySolution that are still active.
      cts.Cancel();

      // Print the result to the console.
      Console.WriteLine("The solution is {0}.", result);

      cts.Dispose();
   }
}

/* Sample output:
The solution is 53.
*/
```

```
Imports System.Threading
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to unlink dataflow blocks.
Friend Class DataflowReceiveAny
    ' Receives the value from the first provided source that has
    ' a message.
    Public Shared Function ReceiveFromAny(Of T)(ParamArray ByVal sources() As ISourceBlock(Of T)) As T
        ' Create a WriteOnceBlock<T> object and link it to each source block.
        Dim writeOnceBlock = New WriteOnceBlock(Of T)(Function(e) e)
        For Each source In sources
            ' Setting MaxMessages to one instructs
            ' the source block to unlink from the WriteOnceBlock<T> object
            ' after offering the WriteOnceBlock<T> object one message.
            source.LinkTo(writeOnceBlock, New DataflowLinkOptions With {.MaxMessages = 1})
        Next source
        ' Return the first value that is offered to the WriteOnceBlock object.
        Return writeOnceBlock.Receive()
    End Function

    ' Demonstrates a function that takes several seconds to produce a result.
    Private Shared Function TrySolution(ByVal n As Integer, ByVal ct As CancellationToken) As Integer
        ' Simulate a lengthy operation that completes within three seconds
        ' or when the provided CancellationToken object is cancelled.
        SpinWait.SpinUntil(Function() ct.IsCancellationRequested, New Random().Next(3000))

        ' Return a value.
        Return n + 42
    End Function

    Shared Sub Main(ByVal args() As String)
        ' Create a shared CancellationTokenSource object to enable the
        ' TrySolution method to be cancelled.
        Dim cts = New CancellationTokenSource()

        ' Create three TransformBlock<int, int> objects.
        ' Each TransformBlock<int, int> object calls the TrySolution method.
        Dim action As Func(Of Integer, Integer) = Function(n) TrySolution(n, cts.Token)
        Dim trySolution1 = New TransformBlock(Of Integer, Integer)(action)
        Dim trySolution2 = New TransformBlock(Of Integer, Integer)(action)
        Dim trySolution3 = New TransformBlock(Of Integer, Integer)(action)

        ' Post data to each TransformBlock<int, int> object.
        trySolution1.Post(11)
        trySolution2.Post(21)
        trySolution3.Post(31)

        ' Call the ReceiveFromAny<T> method to receive the result from the
        ' first TransformBlock<int, int> object to finish.
        Dim result As Integer = ReceiveFromAny(trySolution1, trySolution2, trySolution3)

        ' Cancel all calls to TrySolution that are still active.
        cts.Cancel()

        ' Print the result to the console.
        Console.WriteLine("The solution is {0}.", result)

        cts.Dispose()
    End Sub
End Class

' Sample output:
'The solution is 53.
'
```

为了接收完成的第一个 TransformBlock<TInput,TOutput> 对象的值，该示例定义了 `ReceiveFromAny(T)` 方法。

`ReceiveFromAny(T)` 方法接受一个 ISourceBlock<TOutput> 对象的数组，并将其中每个对象链接到 WriteOnceBlock<T> 对象。当使用 LinkTo 方法将源数据流块链接到目标块时，源会在数据可用时将消息传播到目标。因为 WriteOnceBlock<T> 类只接受为其提供的第一条消息，`ReceiveFromAny(T)` 方法通过调用 Receive 方法来生成其结果。这将生成提供给 WriteOnceBlock<T> 对象的第一条消息。LinkTo 方法有一个重载版本，其含有一个具有 MaxMessages 属性的 DataflowLinkOptions 对象，当该属性设置为 `1` 时，则指示源块在目标收到来自源的一条消息后取消与目标的链接。取消 WriteOnceBlock<T> 对象与其源的链接非常重要，因为当 WriteOnceBlock<T> 对象收到一条消息后，便不再需要源数组与 WriteOnceBlock<T> 对象之间的关系。

为了使 `TrySolution` 的剩余调用能够在其中一个调用计算了一个值后结束，`TrySolution` 方法采用一个 CancellationToken 对象，该对象在对 `ReceiveFromAny(T)` 的调用返回后将被取消。当此 SpinUntil 对象取消时，CancellationToken 方法将返回。

另请参阅

- 数据流

演练：在 Windows 窗体应用程序中使用数据流

2021/11/16 •

本文档演示如何创建在 Windows 窗体应用程序中执行图像处理的数据流块网络。

此示例从指定的文件夹加载图像文件、创建复合图像，并显示结果。本示例使用数据流模型通过网络路由图像。在数据流模型中，程序的独立组件之间通过发送消息进行通信。某个组件收到一条消息时，它会执行某项操作，然后将结果传递给另一个组件。相比之下，在控制流模型中，应用程序使用控制结构（例如条件语句和循环等等）控制程序中操作的顺序。

先决条件

开始本演练之前，请阅读数据流。

> **NOTE**
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

部分

本演练包含以下各节：

- 创建 Windows 窗体应用程序
- 创建数据流网络
- 将数据流网络连接到用户界面
- 完整示例

创建 Windows 窗体应用程序

本节介绍如何创建基本 Windows 窗体应用程序并将控件添加到主窗体。

创建 Windows 窗体应用程序

1. 在 Visual Studio 中，创建 Visual C# 或 Visual Basic“Windows 窗体应用程序”项目。在本文档中，该项目名为 `CompositeImages`。
2. 在主窗体的窗体设计器中，Form1.cs（对于 Visual Basic，则为 Form1.vb）添加了 ToolStrip 控件。
3. 向 ToolStrip 控件添加 ToolStripButton 控件。将 DisplayStyle 属性设置为 Text，并将 Text 属性设置为“Choose Folder”。
4. 向 ToolStrip 控件再添加一个 ToolStripButton 控件。将 DisplayStyle 属性设置为 Text，将 Text 属性设置为“Cancel”，并将 Enabled 属性设置为 `False`。
5. 向主窗体添加 PictureBox 对象。将 Dock 属性设置为 Fill。

创建数据流网络

本节介绍如何创建执行图像处理的数据流网络。

创建数据流网络

1. 向项目中添加对 System.Threading.Tasks.Dataflow.dll 的引用。

2. 确保 Form1.cs(对于 Visual Basic，则为 Form1.vb)包含以下 `using`(Visual Basic 中为 `Using`)语句：

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

3. 将以下数据成员添加到 `Form1` 类：

```
// The head of the dataflow network.
ITargetBlock<string> headBlock = null;

// Enables the user interface to signal cancellation to the network.
CancellationTokenSource cancellationTokenSource;
```

4. 将下面的 `CreateImageProcessingNetwork` 方法添加到 `Form1` 类。此方法创建图像处理网络。

```
// Creates the image processing dataflow network and returns the
// head node of the network.
ITargetBlock<string> CreateImageProcessingNetwork()
{
   //
   // Create the dataflow blocks that form the network.
   //

   // Create a dataflow block that takes a folder path as input
   // and returns a collection of Bitmap objects.
   var loadBitmaps = new TransformBlock<string, IEnumerable<Bitmap>>(path =>
      {
         try
         {
            return LoadBitmaps(path);
         }
         catch (OperationCanceledException)
         {
            // Handle cancellation by passing the empty collection
            // to the next stage of the network.
            return Enumerable.Empty<Bitmap>();
         }
      });

   // Create a dataflow block that takes a collection of Bitmap objects
   // and returns a single composite bitmap.
   var createCompositeBitmap = new TransformBlock<IEnumerable<Bitmap>, Bitmap>(bitmaps =>
      {
         try
         {
            return CreateCompositeBitmap(bitmaps);
         }
         catch (OperationCanceledException)
```

```
      {
         // Handle cancellation by passing null to the next stage
         // of the network.
         return null;
      }
   });

// Create a dataflow block that displays the provided bitmap on the form.
var displayCompositeBitmap = new ActionBlock<Bitmap>(bitmap =>
   {
      // Display the bitmap.
      pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
      pictureBox1.Image = bitmap;

      // Enable the user to select another folder.
      toolStripButton1.Enabled = true;
      toolStripButton2.Enabled = false;
      Cursor = DefaultCursor;
   },
   // Specify a task scheduler from the current synchronization context
   // so that the action runs on the UI thread.
   new ExecutionDataflowBlockOptions
   {
       TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
   });

// Create a dataflow block that responds to a cancellation request by
// displaying an image to indicate that the operation is cancelled and
// enables the user to select another folder.
var operationCancelled = new ActionBlock<object>(delegate
   {
      // Display the error image to indicate that the operation
      // was cancelled.
      pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
      pictureBox1.Image = pictureBox1.ErrorImage;

      // Enable the user to select another folder.
      toolStripButton1.Enabled = true;
      toolStripButton2.Enabled = false;
      Cursor = DefaultCursor;
   },
   // Specify a task scheduler from the current synchronization context
   // so that the action runs on the UI thread.
   new ExecutionDataflowBlockOptions
   {
      TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
   });

//
// Connect the network.
//

// Link loadBitmaps to createCompositeBitmap.
// The provided predicate ensures that createCompositeBitmap accepts the
// collection of bitmaps only if that collection has at least one member.
loadBitmaps.LinkTo(createCompositeBitmap, bitmaps => bitmaps.Count() > 0);

// Also link loadBitmaps to operationCancelled.
// When createCompositeBitmap rejects the message, loadBitmaps
// offers the message to operationCancelled.
// operationCancelled accepts all messages because we do not provide a
// predicate.
loadBitmaps.LinkTo(operationCancelled);

// Link createCompositeBitmap to displayCompositeBitmap.
// The provided predicate ensures that displayCompositeBitmap accepts the
// bitmap only if it is non-null.
createCompositeBitmap.LinkTo(displayCompositeBitmap, bitmap => bitmap != null);
```

```
   // Also link createCompositeBitmap to operationCancelled.
   // When displayCompositeBitmap rejects the message, createCompositeBitmap
   // offers the message to operationCancelled.
   // operationCancelled accepts all messages because we do not provide a
   // predicate.
   createCompositeBitmap.LinkTo(operationCancelled);

   // Return the head of the network.
   return loadBitmaps;
}
```

5. 实现 `LoadBitmaps` **方法**。

```
// Loads all bitmap files that exist at the provided path.
IEnumerable<Bitmap> LoadBitmaps(string path)
{
   List<Bitmap> bitmaps = new List<Bitmap>();

   // Load a variety of image types.
   foreach (string bitmapType in
      new string[] { "*.bmp", "*.gif", "*.jpg", "*.png", "*.tif" })
   {
      // Load each bitmap for the current extension.
      foreach (string fileName in Directory.GetFiles(path, bitmapType))
      {
         // Throw OperationCanceledException if cancellation is requested.
         cancellationTokenSource.Token.ThrowIfCancellationRequested();

         try
         {
            // Add the Bitmap object to the collection.
            bitmaps.Add(new Bitmap(fileName));
         }
         catch (Exception)
         {
            // TODO: A complete application might handle the error.
         }
      }
   }
   return bitmaps;
}
```

6. 实现 `CreateCompositeBitmap` **方法**。

```
// Creates a composite bitmap from the provided collection of Bitmap objects.
// This method computes the average color of each pixel among all bitmaps
// to create the composite image.
Bitmap CreateCompositeBitmap(IEnumerable<Bitmap> bitmaps)
{
   Bitmap[] bitmapArray = bitmaps.ToArray();

   // Compute the maximum width and height components of all
   // bitmaps in the collection.
   Rectangle largest = new Rectangle();
   foreach (var bitmap in bitmapArray)
   {
      if (bitmap.Width > largest.Width)
         largest.Width = bitmap.Width;
      if (bitmap.Height > largest.Height)
         largest.Height = bitmap.Height;
   }

   // Create a 32-bit Bitmap object with the greatest dimensions.
   Bitmap result = new Bitmap(largest.Width, largest.Height,
      PixelFormat.Format32bppArgb);
```

```
// Lock the result Bitmap.
var resultBitmapData = result.LockBits(
   new Rectangle(new Point(), result.Size), ImageLockMode.WriteOnly,
   result.PixelFormat);

// Lock each source bitmap to create a parallel list of BitmapData objects.
var bitmapDataList = (from bitmap in bitmapArray
                      select bitmap.LockBits(
                        new Rectangle(new Point(), bitmap.Size),
                        ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb))
                     .ToList();

// Compute each column in parallel.
Parallel.For(0, largest.Width, new ParallelOptions
{
   CancellationToken = cancellationTokenSource.Token
},
i =>
{
   // Compute each row.
   for (int j = 0; j < largest.Height; j++)
   {
      // Counts the number of bitmaps whose dimensions
      // contain the current location.
      int count = 0;

      // The sum of all alpha, red, green, and blue components.
      int a = 0, r = 0, g = 0, b = 0;

      // For each bitmap, compute the sum of all color components.
      foreach (var bitmapData in bitmapDataList)
      {
         // Ensure that we stay within the bounds of the image.
         if (bitmapData.Width > i && bitmapData.Height > j)
         {
            unsafe
            {
               byte* row = (byte*)(bitmapData.Scan0 + (j * bitmapData.Stride));
               byte* pix = (byte*)(row + (4 * i));
               a += *pix; pix++;
               r += *pix; pix++;
               g += *pix; pix++;
               b += *pix;
            }
            count++;
         }
      }

      //prevent divide by zero in bottom right pixelless corner
      if (count == 0)
          break;

      unsafe
      {
         // Compute the average of each color component.
         a /= count;
         r /= count;
         g /= count;
         b /= count;

         // Set the result pixel.
         byte* row = (byte*)(resultBitmapData.Scan0 + (j * resultBitmapData.Stride));
         byte* pix = (byte*)(row + (4 * i));
         *pix = (byte)a; pix++;
         *pix = (byte)r; pix++;
         *pix = (byte)g; pix++;
         *pix = (byte)b;
      }
```

```
        }
    });

    // Unlock the source bitmaps.
    for (int i = 0; i < bitmapArray.Length; i++)
    {
        bitmapArray[i].UnlockBits(bitmapDataList[i]);
    }

    // Unlock the result bitmap.
    result.UnlockBits(resultBitmapData);

    // Return the result.
    return result;
}
```

NOTE

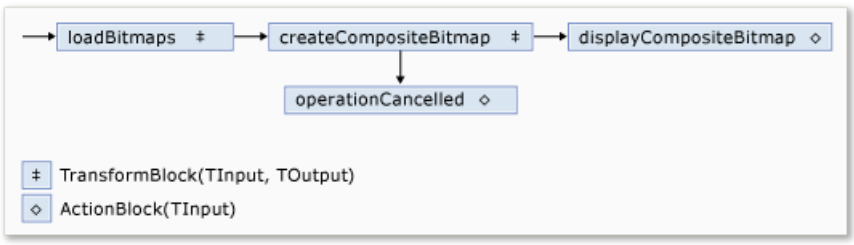
C# 版本的 `CreateCompositeBitmap` 方法使用指针启用高效处理 System.Drawing.Bitmap 对象。因此，若要使用 unsafe 关键字，必须在项目中启用"允许不安全代码"选项。有关如何在 Visual C# 项目中启用不安全代码的详细信息，请参阅"项目设计器"->"生成"页 (C#)。

下表描述了网络的成员。

`loadBitmaps`	TransformBlock<TInput,TOutput>	将文件夹路径用作输入，并生成一组 Bitmap 对象作为输出。
`createCompositeBitmap`	TransformBlock<TInput,TOutput>	将一组 Bitmap 对象用作输入，并生成复合位图作为输出。
`displayCompositeBitmap`	ActionBlock<TInput>	在窗体上显示复合位图。
`operationCancelled`	ActionBlock<TInput>	显示图像以表示操作取消并使用户能够选择其他文件夹。

为了连接数据流块以形成网络，此示例使用 LinkTo 方法。LinkTo 方法包含重载版本，需要使用 Predicate<T> 对象确定目标数据流块是接受还是拒绝消息。此筛选机制使消息块只接收特定值。在此示例中，网络能以两种方式进行分支。主分支从磁盘加载图像，创建复合图像并在窗体上显示该图像。备用分支取消当前操作。借助 Predicate<T> 对象，主分支的数据流块可以拒绝特定消息，从而切换到替换分支。例如，如果用户取消了操作，数据流块 `createCompositeBitmap` 将生成 `null`（在 Visual Basic 中为 `Nothing`）作为其输出。数据流块 `displayCompositeBitmap` 拒绝 `null` 输入值，因此该消息将传递到 `operationCancelled`。数据流块 `operationCancelled` 接受所有消息，并因此显示图像以表示操作取消。

下图显示图像处理网络：



因为 `displayCompositeBitmap` 和 `operationCancelled` 数据流块是在用户界面上操作，所以这些操作要在用户界面线程上执行，这一点很重要。为此，在构造期间，每个对象都提供将 TaskScheduler 属性设置为

TaskScheduler.FromCurrentSynchronizationContext 的 ExecutionDataflowBlockOptions 对象。 TaskScheduler.FromCurrentSynchronizationContext 方法会创建一个在当前同步上下文中执行工作的 TaskScheduler 对象。因为 `CreateImageProcessingNetwork` 方法是通过“选择文件夹”按钮的处理程序调用的，而该处理程序在用户界面线程上运行，所以 `displayCompositeBitmap` 和 `operationCancelled` 数据流块的操作也在用户界面线程上运行。

此示例使用共享的取消令牌，而不是设置 CancellationToken 属性，因为 CancellationToken 属性永久取消数据流块执行。利用取消标记，此示例可多次重复使用同一数据流网络，即使用户取消一个或多个操作也是如此。有关使用 CancellationToken 永久取消数据流块执行的示例，请参阅如何：取消数据流块。

将数据流网络连接到用户界面

本节介绍如何将数据流网络连接到用户界面。复合图像创建和操作取消都是从“选择文件夹”和“取消”按钮启动的。用户选择以上任一按钮时，都会以异步方式启动相应操作。

将数据流网络连接到用户界面

1. 在主窗体的窗体设计器中，创建“选择文件夹”按钮的 Click 事件的事件处理程序。

2. 实现“选择文件夹”按钮的 Click 事件。

```
// Event handler for the Choose Folder button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // Create a FolderBrowserDialog object to enable the user to
   // select a folder.
   FolderBrowserDialog dlg = new FolderBrowserDialog
   {
      ShowNewFolderButton = false
   };

   // Set the selected path to the common Sample Pictures folder
   // if it exists.
   string initialDirectory = Path.Combine(
      Environment.GetFolderPath(Environment.SpecialFolder.CommonPictures),
      "Sample Pictures");
   if (Directory.Exists(initialDirectory))
   {
      dlg.SelectedPath = initialDirectory;
   }

   // Show the dialog and process the dataflow network.
   if (dlg.ShowDialog() == DialogResult.OK)
   {
      // Create a new CancellationTokenSource object to enable
      // cancellation.
      cancellationTokenSource = new CancellationTokenSource();

      // Create the image processing network if needed.
      headBlock ??= CreateImageProcessingNetwork();

      // Post the selected path to the network.
      headBlock.Post(dlg.SelectedPath);

      // Enable the Cancel button and disable the Choose Folder button.
      toolStripButton1.Enabled = false;
      toolStripButton2.Enabled = true;

      // Show a wait cursor.
      Cursor = Cursors.WaitCursor;
   }
}
```

3. 在主窗体的窗体设计器中，创建“取消”按钮的 Click 事件的事件处理程序。

4. 实现“取消”按钮的 Click 事件。

```
// Event handler for the Cancel button.
private void toolStripButton2_Click(object sender, EventArgs e)
{
   // Signal the request for cancellation. The current component of
   // the dataflow network will respond to the cancellation request.
   cancellationTokenSource.Cancel();
}
```

完整示例

下面的示例显示此演练的完整代码。

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace CompositeImages
{
   public partial class Form1 : Form
   {
      // The head of the dataflow network.
      ITargetBlock<string> headBlock = null;

      // Enables the user interface to signal cancellation to the network.
      CancellationTokenSource cancellationTokenSource;

      public Form1()
      {
         InitializeComponent();
      }

      // Creates the image processing dataflow network and returns the
      // head node of the network.
      ITargetBlock<string> CreateImageProcessingNetwork()
      {
         //
         // Create the dataflow blocks that form the network.
         //

         // Create a dataflow block that takes a folder path as input
         // and returns a collection of Bitmap objects.
         var loadBitmaps = new TransformBlock<string, IEnumerable<Bitmap>>(path =>
            {
               try
               {
                  return LoadBitmaps(path);
               }
               catch (OperationCanceledException)
               {
                  // Handle cancellation by passing the empty collection
                  // to the next stage of the network.
                  return Enumerable.Empty<Bitmap>();
               }
```

```
   });

// Create a dataflow block that takes a collection of Bitmap objects
// and returns a single composite bitmap.
var createCompositeBitmap = new TransformBlock<IEnumerable<Bitmap>, Bitmap>(bitmaps =>
   {
      try
      {
         return CreateCompositeBitmap(bitmaps);
      }
      catch (OperationCanceledException)
      {
         // Handle cancellation by passing null to the next stage
         // of the network.
         return null;
      }
   });

// Create a dataflow block that displays the provided bitmap on the form.
var displayCompositeBitmap = new ActionBlock<Bitmap>(bitmap =>
   {
      // Display the bitmap.
      pictureBox1.SizeMode = PictureBoxSizeMode.StretchImage;
      pictureBox1.Image = bitmap;

      // Enable the user to select another folder.
      toolStripButton1.Enabled = true;
      toolStripButton2.Enabled = false;
      Cursor = DefaultCursor;
   },
   // Specify a task scheduler from the current synchronization context
   // so that the action runs on the UI thread.
   new ExecutionDataflowBlockOptions
   {
       TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
   });

// Create a dataflow block that responds to a cancellation request by
// displaying an image to indicate that the operation is cancelled and
// enables the user to select another folder.
var operationCancelled = new ActionBlock<object>(delegate
   {
      // Display the error image to indicate that the operation
      // was cancelled.
      pictureBox1.SizeMode = PictureBoxSizeMode.CenterImage;
      pictureBox1.Image = pictureBox1.ErrorImage;

      // Enable the user to select another folder.
      toolStripButton1.Enabled = true;
      toolStripButton2.Enabled = false;
      Cursor = DefaultCursor;
   },
   // Specify a task scheduler from the current synchronization context
   // so that the action runs on the UI thread.
   new ExecutionDataflowBlockOptions
   {
      TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
   });

//
// Connect the network.
//

// Link loadBitmaps to createCompositeBitmap.
// The provided predicate ensures that createCompositeBitmap accepts the
// collection of bitmaps only if that collection has at least one member.
loadBitmaps.LinkTo(createCompositeBitmap, bitmaps => bitmaps.Count() > 0);

// Also link loadBitmaps to operationCancelled.
```

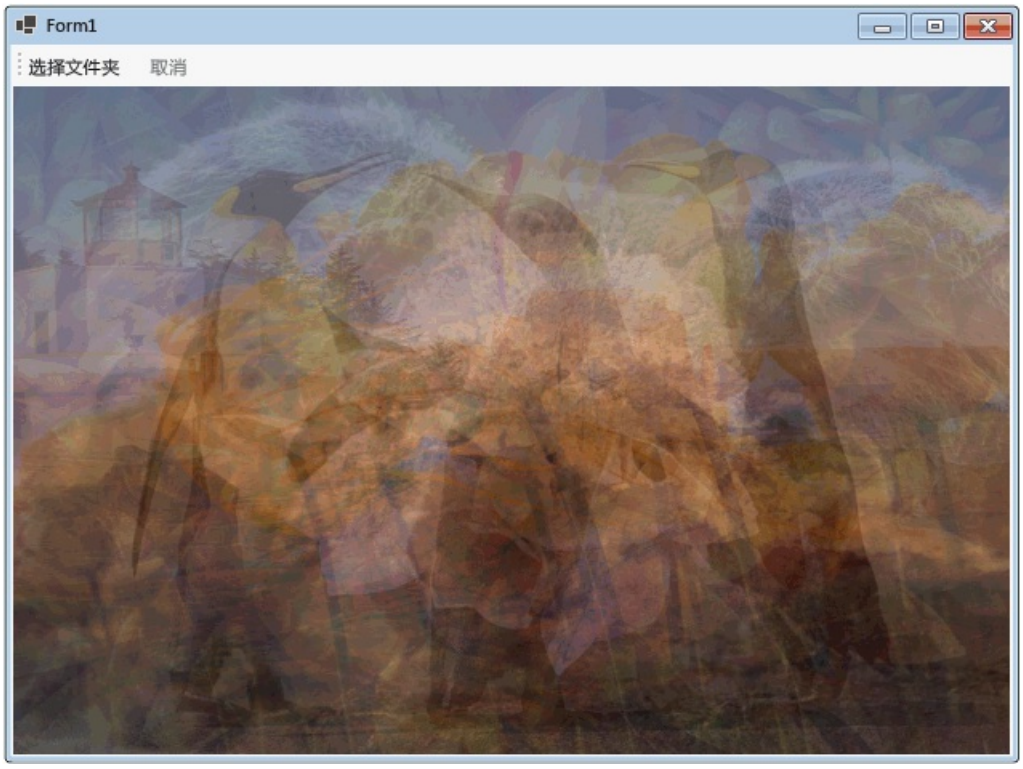
```
    // When createCompositeBitmap rejects the message, loadBitmaps
    // offers the message to operationCancelled.
    // operationCancelled accepts all messages because we do not provide a
    // predicate.
    loadBitmaps.LinkTo(operationCancelled);

    // Link createCompositeBitmap to displayCompositeBitmap.
    // The provided predicate ensures that displayCompositeBitmap accepts the
    // bitmap only if it is non-null.
    createCompositeBitmap.LinkTo(displayCompositeBitmap, bitmap => bitmap != null);

    // Also link createCompositeBitmap to operationCancelled.
    // When displayCompositeBitmap rejects the message, createCompositeBitmap
    // offers the message to operationCancelled.
    // operationCancelled accepts all messages because we do not provide a
    // predicate.
    createCompositeBitmap.LinkTo(operationCancelled);

    // Return the head of the network.
    return loadBitmaps;
}

// Loads all bitmap files that exist at the provided path.
IEnumerable<Bitmap> LoadBitmaps(string path)
{
    List<Bitmap> bitmaps = new List<Bitmap>();

    // Load a variety of image types.
    foreach (string bitmapType in
        new string[] { "*.bmp", "*.gif", "*.jpg", "*.png", "*.tif" })
    {
        // Load each bitmap for the current extension.
        foreach (string fileName in Directory.GetFiles(path, bitmapType))
        {
            // Throw OperationCanceledException if cancellation is requested.
            cancellationTokenSource.Token.ThrowIfCancellationRequested();

            try
            {
                // Add the Bitmap object to the collection.
                bitmaps.Add(new Bitmap(fileName));
            }
            catch (Exception)
            {
                // TODO: A complete application might handle the error.
            }
        }
    }
    return bitmaps;
}

// Creates a composite bitmap from the provided collection of Bitmap objects.
// This method computes the average color of each pixel among all bitmaps
// to create the composite image.
Bitmap CreateCompositeBitmap(IEnumerable<Bitmap> bitmaps)
{
    Bitmap[] bitmapArray = bitmaps.ToArray();

    // Compute the maximum width and height components of all
    // bitmaps in the collection.
    Rectangle largest = new Rectangle();
    foreach (var bitmap in bitmapArray)
    {
        if (bitmap.Width > largest.Width)
            largest.Width = bitmap.Width;
        if (bitmap.Height > largest.Height)
            largest.Height = bitmap.Height;
    }
```

```
// Create a 32-bit Bitmap object with the greatest dimensions.
Bitmap result = new Bitmap(largest.Width, largest.Height,
   PixelFormat.Format32bppArgb);

// Lock the result Bitmap.
var resultBitmapData = result.LockBits(
   new Rectangle(new Point(), result.Size), ImageLockMode.WriteOnly,
   result.PixelFormat);

// Lock each source bitmap to create a parallel list of BitmapData objects.
var bitmapDataList = (from bitmap in bitmapArray
                      select bitmap.LockBits(
                        new Rectangle(new Point(), bitmap.Size),
                        ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb))
                     .ToList();

// Compute each column in parallel.
Parallel.For(0, largest.Width, new ParallelOptions
{
   CancellationToken = cancellationTokenSource.Token
},
i =>
{
   // Compute each row.
   for (int j = 0; j < largest.Height; j++)
   {
      // Counts the number of bitmaps whose dimensions
      // contain the current location.
      int count = 0;

      // The sum of all alpha, red, green, and blue components.
      int a = 0, r = 0, g = 0, b = 0;

      // For each bitmap, compute the sum of all color components.
      foreach (var bitmapData in bitmapDataList)
      {
         // Ensure that we stay within the bounds of the image.
         if (bitmapData.Width > i && bitmapData.Height > j)
         {
            unsafe
            {
               byte* row = (byte*)(bitmapData.Scan0 + (j * bitmapData.Stride));
               byte* pix = (byte*)(row + (4 * i));
               a += *pix; pix++;
               r += *pix; pix++;
               g += *pix; pix++;
               b += *pix;
            }
            count++;
         }
      }

      //prevent divide by zero in bottom right pixelless corner
      if (count == 0)
          break;

      unsafe
      {
         // Compute the average of each color component.
         a /= count;
         r /= count;
         g /= count;
         b /= count;

         // Set the result pixel.
         byte* row = (byte*)(resultBitmapData.Scan0 + (j * resultBitmapData.Stride));
         byte* pix = (byte*)(row + (4 * i));
         *pix = (byte)a; pix++;
         *pix = (byte)r; pix++;
```

```
                *pix = (byte)r; pix++;
                *pix = (byte)g; pix++;
                *pix = (byte)b;
            }
        }
    });

    // Unlock the source bitmaps.
    for (int i = 0; i < bitmapArray.Length; i++)
    {
        bitmapArray[i].UnlockBits(bitmapDataList[i]);
    }

    // Unlock the result bitmap.
    result.UnlockBits(resultBitmapData);

    // Return the result.
    return result;
}

// Event handler for the Choose Folder button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
    // Create a FolderBrowserDialog object to enable the user to
    // select a folder.
    FolderBrowserDialog dlg = new FolderBrowserDialog
    {
        ShowNewFolderButton = false
    };

    // Set the selected path to the common Sample Pictures folder
    // if it exists.
    string initialDirectory = Path.Combine(
        Environment.GetFolderPath(Environment.SpecialFolder.CommonPictures),
        "Sample Pictures");
    if (Directory.Exists(initialDirectory))
    {
        dlg.SelectedPath = initialDirectory;
    }

    // Show the dialog and process the dataflow network.
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        // Create a new CancellationTokenSource object to enable
        // cancellation.
        cancellationTokenSource = new CancellationTokenSource();

        // Create the image processing network if needed.
        headBlock ??= CreateImageProcessingNetwork();

        // Post the selected path to the network.
        headBlock.Post(dlg.SelectedPath);

        // Enable the Cancel button and disable the Choose Folder button.
        toolStripButton1.Enabled = false;
        toolStripButton2.Enabled = true;

        // Show a wait cursor.
        Cursor = Cursors.WaitCursor;
    }
}

// Event handler for the Cancel button.
private void toolStripButton2_Click(object sender, EventArgs e)
{
    // Signal the request for cancellation. The current component of
    // the dataflow network will respond to the cancellation request.
    cancellationTokenSource.Cancel();
}
```

```
        ~Form1()
        {
            cancellationTokenSource.Dispose();
        }
    }
}
```

下图显示公共 \Sample Pictures\ 文件夹的典型输出。



请参阅

- 数据流

如何：取消数据流块

2021/11/16 •

本文档介绍如何在应用程序中启用取消。此示例使用 Windows 窗体显示数据流管道中工作项的活动位置以及取消的效果。

> **NOTE**
> TPL 数据流库(System.Threading.Tasks.Dataflow 命名空间)不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

创建 Windows 窗体应用程序

1. 创建一个 C# 或 Visual Basic **Windows 窗体应用程序** 项目。在以下步骤中，该项目命名为 `CancellationWinForms`。
2. 在主窗体的窗体设计器中，Form1.cs(对于 Visual Basic，则为 Form1.vb)添加了 ToolStrip 控件。
3. 向 ToolStrip 控件添加 ToolStripButton 控件。将 DisplayStyle 属性设置为 Text，并将 Text 属性设置为“添加工作项”。
4. 向 ToolStrip 控件再添加一个 ToolStripButton 控件。将 DisplayStyle 属性设置为 Text，将 Text 属性设置为“Cancel”，并将 Enabled 属性设置为 `False`。
5. 向 ToolStrip 控件添加四个 ToolStripProgressBar 对象。

创建数据流管道

本部分介绍如何创建数据流管道，用以处理工作项以及更新进度条。

创建数据流管道

1. 在项目中，添加对 System.Threading.Tasks.Dataflow.dll 的引用。
2. 确保 Form1.cs(对于 Visual Basic 则为 Form1.vb)包含以下 `using` 语句(Visual Basic 中为 `Imports`)。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

```
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow
```

3. 将 `WorkItem` 类添加为 `Form1` 类的内部类型。

```
// A placeholder type that performs work.
class WorkItem
{
   // Performs work for the provided number of milliseconds.
   public void DoWork(int milliseconds)
   {
      // For demonstration, suspend the current thread.
      Thread.Sleep(milliseconds);
   }
}
```

```
' A placeholder type that performs work.
Private Class WorkItem
    ' Performs work for the provided number of milliseconds.
    Public Sub DoWork(ByVal milliseconds As Integer)
        ' For demonstration, suspend the current thread.
        Thread.Sleep(milliseconds)
    End Sub
End Class
```

4. 将以下数据成员添加到 `Form1` 类。

```
// Enables the user interface to signal cancellation.
CancellationTokenSource cancellationSource;

// The first node in the dataflow pipeline.
TransformBlock<WorkItem, WorkItem> startWork;

// The second, and final, node in the dataflow pipeline.
ActionBlock<WorkItem> completeWork;

// Increments the value of the provided progress bar.
ActionBlock<ToolStripProgressBar> incrementProgress;

// Decrements the value of the provided progress bar.
ActionBlock<ToolStripProgressBar> decrementProgress;

// Enables progress bar actions to run on the UI thread.
TaskScheduler uiTaskScheduler;
```

```
' Enables the user interface to signal cancellation.
Private cancellationSource As CancellationTokenSource

' The first node in the dataflow pipeline.
Private startWork As TransformBlock(Of WorkItem, WorkItem)

' The second, and final, node in the dataflow pipeline.
Private completeWork As ActionBlock(Of WorkItem)

' Increments the value of the provided progress bar.
Private incrementProgress As ActionBlock(Of ToolStripProgressBar)

' Decrements the value of the provided progress bar.
Private decrementProgress As ActionBlock(Of ToolStripProgressBar)

' Enables progress bar actions to run on the UI thread.
Private uiTaskScheduler As TaskScheduler
```

5. 将下面的 `CreatePipeline` 方法添加到 `Form1` 类。

```
// Creates the blocks that participate in the dataflow pipeline.
```

```
// Creates the blocks that participate in the dataflow pipeline.
private void CreatePipeline()
{
   // Create the cancellation source.
   cancellationSource = new CancellationTokenSource();

   // Create the first node in the pipeline.
   startWork = new TransformBlock<WorkItem, WorkItem>(workItem =>
   {
      // Perform some work.
      workItem.DoWork(250);

      // Decrement the progress bar that tracks the count of
      // active work items in this stage of the pipeline.
      decrementProgress.Post(toolStripProgressBar1);

      // Increment the progress bar that tracks the count of
      // active work items in the next stage of the pipeline.
      incrementProgress.Post(toolStripProgressBar2);

      // Send the work item to the next stage of the pipeline.
      return workItem;
   },
   new ExecutionDataflowBlockOptions
   {
      CancellationToken = cancellationSource.Token
   });

   // Create the second, and final, node in the pipeline.
   completeWork = new ActionBlock<WorkItem>(workItem =>
   {
      // Perform some work.
      workItem.DoWork(1000);

      // Decrement the progress bar that tracks the count of
      // active work items in this stage of the pipeline.
      decrementProgress.Post(toolStripProgressBar2);

      // Increment the progress bar that tracks the overall
      // count of completed work items.
      incrementProgress.Post(toolStripProgressBar3);
   },
   new ExecutionDataflowBlockOptions
   {
      CancellationToken = cancellationSource.Token,
      MaxDegreeOfParallelism = 2
   });

   // Connect the two nodes of the pipeline. When the first node completes,
   // set the second node also to the completed state.
   startWork.LinkTo(
      completeWork, new DataflowLinkOptions { PropagateCompletion = true });

   // Create the dataflow action blocks that increment and decrement
   // progress bars.
   // These blocks use the task scheduler that is associated with
   // the UI thread.

   incrementProgress = new ActionBlock<ToolStripProgressBar>(
      progressBar => progressBar.Value++,
      new ExecutionDataflowBlockOptions
      {
         CancellationToken = cancellationSource.Token,
         TaskScheduler = uiTaskScheduler
      });

   decrementProgress = new ActionBlock<ToolStripProgressBar>(
      progressBar => progressBar.Value--,
      new ExecutionDataflowBlockOptions
      {
```

```
        {
            CancellationToken = cancellationSource.Token,
            TaskScheduler = uiTaskScheduler
        });
}
```

```
' Creates the blocks that participate in the dataflow pipeline.
Private Sub CreatePipeline()
    ' Create the cancellation source.
    cancellationSource = New CancellationTokenSource()

    ' Create the first node in the pipeline.
    startWork = New TransformBlock(Of WorkItem, WorkItem)(Function(workItem)
                                                              ' Perform some work.
                                                              ' Decrement the progress bar that
tracks the count of
                                                              ' active work items in this stage of
the pipeline.
                                                              ' Increment the progress bar that
tracks the count of
                                                              ' active work items in the next stage
of the pipeline.
                                                              ' Send the work item to the next stage
of the pipeline.
                                                              workItem.DoWork(250)

decrementProgress.Post(toolStripProgressBar1)

incrementProgress.Post(toolStripProgressBar2)
                                                              Return workItem
                                                          End Function,
    New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token})

    ' Create the second, and final, node in the pipeline.
    completeWork = New ActionBlock(Of WorkItem)(Sub(workItem)
                                                    ' Perform some work.
                                                    ' Decrement the progress bar that tracks the
count of
                                                    ' active work items in this stage of the
pipeline.
                                                    ' Increment the progress bar that tracks the
overall
                                                    ' count of completed work items.
                                                    workItem.DoWork(1000)
                                                    decrementProgress.Post(toolStripProgressBar2)
                                                    incrementProgress.Post(toolStripProgressBar3)
                                                End Sub,
    New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                            .MaxDegreeOfParallelism = 2})

    ' Connect the two nodes of the pipeline. When the first node completes,
    ' set the second node also to the completed state.
    startWork.LinkTo(
       completeWork, New DataflowLinkOptions With {.PropagateCompletion = true})

    ' Create the dataflow action blocks that increment and decrement
    ' progress bars.
    ' These blocks use the task scheduler that is associated with
    ' the UI thread.

    incrementProgress = New ActionBlock(Of ToolStripProgressBar)(
       Sub(progressBar) progressBar.Value += 1,
       New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                               .TaskScheduler = uiTaskScheduler})

    decrementProgress = New ActionBlock(Of ToolStripProgressBar)(
       Sub(progressBar) progressBar.Value -= 1,
       New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                               .TaskScheduler = uiTaskScheduler})

End Sub
```

因为 `incrementProgress` 和 `decrementProgress` 数据流块是在用户界面上操作，所以这些操作要在用户界面线程上执行，这一点很重要。为此，在构造期间，每个对象都提供将 TaskScheduler 属性设置为 TaskScheduler.FromCurrentSynchronizationContext 的 ExecutionDataflowBlockOptions 对象。TaskScheduler.FromCurrentSynchronizationContext 方法会创建一个在当前同步上下文中执行工作的 TaskScheduler 对象。因为 `Form1` 构造函数是从用户界面线程中调用的，所以 `incrementProgress` 和 `decrementProgress` 数据流块的操作也会在用户界面线程上运行。

此示例在构造管道成员时设置 CancellationToken 属性。由于 CancellationToken 属性永久取消数据流块执行，因此如果用户在取消操作后又想再将更多工作项添加到管道中，必须重新创建整个管道。有关演示使用另一种方法取消数据流块以便在取消操作后可以执行其他工作的示例，请参阅演练：在 Windows 窗体应用程序中使用数据流。

将数据流管道连接到用户界面

本节介绍如何将数据流管道连接到用户界面。创建管道以及将工作项添加到管道中都由“添加工作项”按钮的事件处理程序控制。通过“取消”按钮启动取消操作。用户单击以上任一按钮时，都会以异步方式启动相应操作。

将数据流管道连接到用户界面

1. 在主窗体的窗体设计器中，创建“添加工作项”按钮的 Click 事件的事件处理程序。

2. 实现“添加工作项”按钮的 Click 事件。

```
// Event handler for the Add Work Items button.
private void toolStripButton1_Click(object sender, EventArgs e)
{
   // The Cancel button is disabled when the pipeline is not active.
   // Therefore, create the pipeline and enable the Cancel button
   // if the Cancel button is disabled.
   if (!toolStripButton2.Enabled)
   {
      CreatePipeline();

      // Enable the Cancel button.
      toolStripButton2.Enabled = true;
   }

   // Post several work items to the head of the pipeline.
   for (int i = 0; i < 5; i++)
   {
      toolStripProgressBar1.Value++;
      startWork.Post(new WorkItem());
   }
}
```

```
' Event handler for the Add Work Items button.
Private Sub toolStripButton1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton1.Click
    ' The Cancel button is disabled when the pipeline is not active.
    ' Therefore, create the pipeline and enable the Cancel button
    ' if the Cancel button is disabled.
    If Not toolStripButton2.Enabled Then
        CreatePipeline()

        ' Enable the Cancel button.
        toolStripButton2.Enabled = True
    End If

    ' Post several work items to the head of the pipeline.
    For i As Integer = 0 To 4
        toolStripProgressBar1.Value += 1
        startWork.Post(New WorkItem())
    Next i
End Sub
```

3. 在主窗体的窗体设计器中，创建“取消”按钮的 Click 事件的事件处理程序。

4. 实现“取消”按钮的 Click 事件处理程序。

```
// Event handler for the Cancel button.
private async void toolStripButton2_Click(object sender, EventArgs e)
{
   // Disable both buttons.
   toolStripButton1.Enabled = false;
   toolStripButton2.Enabled = false;

   // Trigger cancellation.
   cancellationSource.Cancel();

   try
   {
      // Asynchronously wait for the pipeline to complete processing and for
      // the progress bars to update.
      await Task.WhenAll(
         completeWork.Completion,
         incrementProgress.Completion,
         decrementProgress.Completion);
   }
   catch (OperationCanceledException)
   {
   }

   // Increment the progress bar that tracks the number of cancelled
   // work items by the number of active work items.
   toolStripProgressBar4.Value += toolStripProgressBar1.Value;
   toolStripProgressBar4.Value += toolStripProgressBar2.Value;

   // Reset the progress bars that track the number of active work items.
   toolStripProgressBar1.Value = 0;
   toolStripProgressBar2.Value = 0;

   // Enable the Add Work Items button.
   toolStripButton1.Enabled = true;
}
```

```
' Event handler for the Cancel button.
Private Async Sub toolStripButton2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton2.Click
    ' Disable both buttons.
    toolStripButton1.Enabled = False
    toolStripButton2.Enabled = False

    ' Trigger cancellation.
    cancellationSource.Cancel()

    Try
        ' Asynchronously wait for the pipeline to complete processing and for
        ' the progress bars to update.
        Await Task.WhenAll(completeWork.Completion, incrementProgress.Completion,
decrementProgress.Completion)
    Catch e1 As OperationCanceledException
    End Try

    ' Increment the progress bar that tracks the number of cancelled
    ' work items by the number of active work items.
    toolStripProgressBar4.Value += toolStripProgressBar1.Value
    toolStripProgressBar4.Value += toolStripProgressBar2.Value

    ' Reset the progress bars that track the number of active work items.
    toolStripProgressBar1.Value = 0
    toolStripProgressBar2.Value = 0

    ' Enable the Add Work Items button.
    toolStripButton1.Enabled = True
End Sub
```

示例

下面的示例演示 Form1.cs（对于 Visual Basic 则为 Form1.vb）的完整代码。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace CancellationWinForms
{
   public partial class Form1 : Form
   {
      // A placeholder type that performs work.
      class WorkItem
      {
         // Performs work for the provided number of milliseconds.
         public void DoWork(int milliseconds)
         {
            // For demonstration, suspend the current thread.
            Thread.Sleep(milliseconds);
         }
      }

      // Enables the user interface to signal cancellation.
      CancellationTokenSource cancellationSource;

      // The first node in the dataflow pipeline.
      TransformBlock<WorkItem, WorkItem> startWork;

      // The second, and final, node in the dataflow pipeline.
      ActionBlock<WorkItem> completeWork;
```

```
        // Increments the value of the provided progress bar.
        ActionBlock<ToolStripProgressBar> incrementProgress;

        // Decrements the value of the provided progress bar.
        ActionBlock<ToolStripProgressBar> decrementProgress;

        // Enables progress bar actions to run on the UI thread.
        TaskScheduler uiTaskScheduler;

        public Form1()
        {
           InitializeComponent();

           // Create the UI task scheduler from the current sychronization
           // context.
           uiTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext();
        }

        // Creates the blocks that participate in the dataflow pipeline.
        private void CreatePipeline()
        {
           // Create the cancellation source.
           cancellationSource = new CancellationTokenSource();

           // Create the first node in the pipeline.
           startWork = new TransformBlock<WorkItem, WorkItem>(workItem =>
           {
              // Perform some work.
              workItem.DoWork(250);

              // Decrement the progress bar that tracks the count of
              // active work items in this stage of the pipeline.
              decrementProgress.Post(toolStripProgressBar1);

              // Increment the progress bar that tracks the count of
              // active work items in the next stage of the pipeline.
              incrementProgress.Post(toolStripProgressBar2);

              // Send the work item to the next stage of the pipeline.
              return workItem;
           },
           new ExecutionDataflowBlockOptions
           {
              CancellationToken = cancellationSource.Token
           });

           // Create the second, and final, node in the pipeline.
           completeWork = new ActionBlock<WorkItem>(workItem =>
           {
              // Perform some work.
              workItem.DoWork(1000);

              // Decrement the progress bar that tracks the count of
              // active work items in this stage of the pipeline.
              decrementProgress.Post(toolStripProgressBar2);

              // Increment the progress bar that tracks the overall
              // count of completed work items.
              incrementProgress.Post(toolStripProgressBar3);
           },
           new ExecutionDataflowBlockOptions
           {
              CancellationToken = cancellationSource.Token,
              MaxDegreeOfParallelism = 2
           });

           // Connect the two nodes of the pipeline. When the first node completes,
           // set the second node also to the completed state.
           startWork.LinkTo(
```

```
            completeWork, new DataflowLinkOptions { PropagateCompletion = true });

        // Create the dataflow action blocks that increment and decrement
        // progress bars.
        // These blocks use the task scheduler that is associated with
        // the UI thread.

        incrementProgress = new ActionBlock<ToolStripProgressBar>(
            progressBar => progressBar.Value++,
            new ExecutionDataflowBlockOptions
            {
                CancellationToken = cancellationSource.Token,
                TaskScheduler = uiTaskScheduler
            });

        decrementProgress = new ActionBlock<ToolStripProgressBar>(
            progressBar => progressBar.Value--,
            new ExecutionDataflowBlockOptions
            {
                CancellationToken = cancellationSource.Token,
                TaskScheduler = uiTaskScheduler
            });
    }

    // Event handler for the Add Work Items button.
    private void toolStripButton1_Click(object sender, EventArgs e)
    {
        // The Cancel button is disabled when the pipeline is not active.
        // Therefore, create the pipeline and enable the Cancel button
        // if the Cancel button is disabled.
        if (!toolStripButton2.Enabled)
        {
            CreatePipeline();

            // Enable the Cancel button.
            toolStripButton2.Enabled = true;
        }

        // Post several work items to the head of the pipeline.
        for (int i = 0; i < 5; i++)
        {
            toolStripProgressBar1.Value++;
            startWork.Post(new WorkItem());
        }
    }

    // Event handler for the Cancel button.
    private async void toolStripButton2_Click(object sender, EventArgs e)
    {
        // Disable both buttons.
        toolStripButton1.Enabled = false;
        toolStripButton2.Enabled = false;

        // Trigger cancellation.
        cancellationSource.Cancel();

        try
        {
            // Asynchronously wait for the pipeline to complete processing and for
            // the progress bars to update.
            await Task.WhenAll(
                completeWork.Completion,
                incrementProgress.Completion,
                decrementProgress.Completion);
        }
        catch (OperationCanceledException)
        {
        }
```

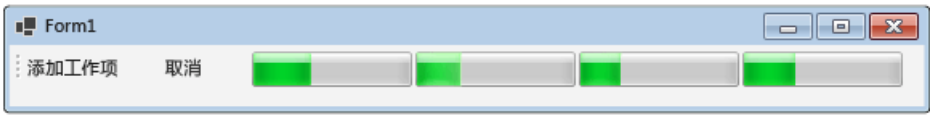
```
         // Increment the progress bar that tracks the number of cancelled
         // work items by the number of active work items.
         toolStripProgressBar4.Value += toolStripProgressBar1.Value;
         toolStripProgressBar4.Value += toolStripProgressBar2.Value;

         // Reset the progress bars that track the number of active work items.
         toolStripProgressBar1.Value = 0;
         toolStripProgressBar2.Value = 0;

         // Enable the Add Work Items button.
         toolStripButton1.Enabled = true;
      }

      ~Form1()
      {
         cancellationSource.Dispose();
      }
   }
}
```

```
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow


Namespace CancellationWinForms
    Partial Public Class Form1
        Inherits Form
        ' A placeholder type that performs work.
        Private Class WorkItem
            ' Performs work for the provided number of milliseconds.
            Public Sub DoWork(ByVal milliseconds As Integer)
                ' For demonstration, suspend the current thread.
                Thread.Sleep(milliseconds)
            End Sub
        End Class

        ' Enables the user interface to signal cancellation.
        Private cancellationSource As CancellationTokenSource

        ' The first node in the dataflow pipeline.
        Private startWork As TransformBlock(Of WorkItem, WorkItem)

        ' The second, and final, node in the dataflow pipeline.
        Private completeWork As ActionBlock(Of WorkItem)

        ' Increments the value of the provided progress bar.
        Private incrementProgress As ActionBlock(Of ToolStripProgressBar)

        ' Decrements the value of the provided progress bar.
        Private decrementProgress As ActionBlock(Of ToolStripProgressBar)

        ' Enables progress bar actions to run on the UI thread.
        Private uiTaskScheduler As TaskScheduler

        Public Sub New()
            InitializeComponent()

            ' Create the UI task scheduler from the current sychronization
            ' context.
            uiTaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
        End Sub

        ' Creates the blocks that participate in the dataflow pipeline.
        Private Sub CreatePipeline()
            ' Create the cancellation source.
            cancellationSource = New CancellationTokenSource()
```

```
            ' Create the first node in the pipeline.
            startWork = New TransformBlock(Of WorkItem, WorkItem)(Function(workItem)
                                                                      ' Perform some work.
                                                                      ' Decrement the progress bar that
tracks the count of
                                                                      ' active work items in this stage of
the pipeline.
                                                                      ' Increment the progress bar that
tracks the count of
                                                                      ' active work items in the next stage
of the pipeline.
                                                                      ' Send the work item to the next stage
of the pipeline.
                                                                      workItem.DoWork(250)

decrementProgress.Post(toolStripProgressBar1)

incrementProgress.Post(toolStripProgressBar2)
                                                                      Return workItem
                                                                  End Function,
            New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token})

            ' Create the second, and final, node in the pipeline.
            completeWork = New ActionBlock(Of WorkItem)(Sub(workItem)
                                                            ' Perform some work.
                                                            ' Decrement the progress bar that tracks the
count of
                                                            ' active work items in this stage of the
pipeline.
                                                            ' Increment the progress bar that tracks the
overall
                                                            ' count of completed work items.
                                                            workItem.DoWork(1000)
                                                            decrementProgress.Post(toolStripProgressBar2)
                                                            incrementProgress.Post(toolStripProgressBar3)
                                                        End Sub,
            New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                    .MaxDegreeOfParallelism = 2})

            ' Connect the two nodes of the pipeline. When the first node completes,
            ' set the second node also to the completed state.
            startWork.LinkTo(
               completeWork, New DataflowLinkOptions With {.PropagateCompletion = true})

            ' Create the dataflow action blocks that increment and decrement
            ' progress bars.
            ' These blocks use the task scheduler that is associated with
            ' the UI thread.

            incrementProgress = New ActionBlock(Of ToolStripProgressBar)(
               Sub(progressBar) progressBar.Value += 1,
               New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                       .TaskScheduler = uiTaskScheduler})

            decrementProgress = New ActionBlock(Of ToolStripProgressBar)(
               Sub(progressBar) progressBar.Value -= 1,
               New ExecutionDataflowBlockOptions With {.CancellationToken = cancellationSource.Token,
                                                       .TaskScheduler = uiTaskScheduler})

        End Sub

        ' Event handler for the Add Work Items button.
        Private Sub toolStripButton1_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton1.Click
            ' The Cancel button is disabled when the pipeline is not active.
            ' Therefore, create the pipeline and enable the Cancel button
            ' if the Cancel button is disabled.
            If Not toolStripButton2.Enabled Then
```

```
                If Not toolStripButton2.Enabled Then
                    CreatePipeline()

                    ' Enable the Cancel button.
                    toolStripButton2.Enabled = True
                End If

                ' Post several work items to the head of the pipeline.
                For i As Integer = 0 To 4
                    toolStripProgressBar1.Value += 1
                    startWork.Post(New WorkItem())
                Next i
            End Sub

            ' Event handler for the Cancel button.
            Private Async Sub toolStripButton2_Click(ByVal sender As Object, ByVal e As EventArgs) Handles
toolStripButton2.Click
                ' Disable both buttons.
                toolStripButton1.Enabled = False
                toolStripButton2.Enabled = False

                ' Trigger cancellation.
                cancellationSource.Cancel()

                Try
                    ' Asynchronously wait for the pipeline to complete processing and for
                    ' the progress bars to update.
                    Await Task.WhenAll(completeWork.Completion, incrementProgress.Completion,
decrementProgress.Completion)
                Catch e1 As OperationCanceledException
                End Try

                ' Increment the progress bar that tracks the number of cancelled
                ' work items by the number of active work items.
                toolStripProgressBar4.Value += toolStripProgressBar1.Value
                toolStripProgressBar4.Value += toolStripProgressBar2.Value

                ' Reset the progress bars that track the number of active work items.
                toolStripProgressBar1.Value = 0
                toolStripProgressBar2.Value = 0

                ' Enable the Add Work Items button.
                toolStripButton1.Enabled = True
            End Sub

            Protected Overrides Sub Finalize()
                cancellationSource.Dispose()
                MyBase.Finalize()
            End Sub
        End Class
    End Namespace
```

下图显示正在运行的应用程序。



另请参阅

- 数据流

演练：创建自定义数据流块类型

2021/11/16 •

尽管 TPL 数据流库提供了多种可启用各种功能的数据流块类型，但也可以创建自定义块类型。本文档介绍了如何创建实现自定义行为的数据流块类型。

系统必备

阅读本文档前，请先阅读数据流。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

定义滑动窗口数据流块

假设数据流应用要求缓冲输入值，并以滑动窗口方式输出值。例如，如果输入值为 {0, 1, 2, 3, 4, 5}，且窗口大小为 3，那么滑动窗口数据流块生成输出数组 {0, 1, 2}、{1, 2, 3}、{2, 3, 4} 和 {3, 4, 5}。下面各部分介绍了两种方式，用于创建实现此自定义行为的数据流块类型。第一种方式是，使用 Encapsulate 方法将 ISourceBlock<TOutput> 对象和 ITargetBlock<TInput> 对象的功能合并到一个传播器块中。第二种方式是，定义派生自 IPropagatorBlock<TInput,TOutput> 的类，并结合现有功能执行自定义行为。

使用 Encapsulate 方法定义滑动窗口数据流块

下面的示例使用 Encapsulate 方法通过目标和源创建传播器块。使用传播器块，可以将源块和目标块用作数据的接收方和发送方。

如果需要自定义数据流功能，但不需要提供其他方法、属性或字段的类型，此方式就很有用。

```
// Creates a IPropagatorBlock<T, T[]> object propagates data in a
// sliding window fashion.
public static IPropagatorBlock<T, T[]> CreateSlidingWindow<T>(int windowSize)
{
   // Create a queue to hold messages.
   var queue = new Queue<T>();

   // The source part of the propagator holds arrays of size windowSize
   // and propagates data out to any connected targets.
   var source = new BufferBlock<T[]>();

   // The target part receives data and adds them to the queue.
   var target = new ActionBlock<T>(item =>
   {
      // Add the item to the queue.
      queue.Enqueue(item);
      // Remove the oldest item when the queue size exceeds the window size.
      if (queue.Count > windowSize)
         queue.Dequeue();
      // Post the data in the queue to the source block when the queue size
      // equals the window size.
      if (queue.Count == windowSize)
         source.Post(queue.ToArray());
   });

   // When the target is set to the completed state, propagate out any
   // remaining data and set the source to the completed state.
   target.Completion.ContinueWith(delegate
   {
      if (queue.Count > 0 && queue.Count < windowSize)
         source.Post(queue.ToArray());
      source.Complete();
   });

   // Return a IPropagatorBlock<T, T[]> object that encapsulates the
   // target and source blocks.
   return DataflowBlock.Encapsulate(target, source);
}
```

```
' Creates a IPropagatorBlock<T, T[]> object propagates data in a
' sliding window fashion.
Public Shared Function CreateSlidingWindow(Of T)(ByVal windowSize As Integer) As IPropagatorBlock(Of T, T())
    ' Create a queue to hold messages.
    Dim queue = New Queue(Of T)()

    ' The source part of the propagator holds arrays of size windowSize
    ' and propagates data out to any connected targets.
    Dim source = New BufferBlock(Of T())()

    ' The target part receives data and adds them to the queue.
    Dim target = New ActionBlock(Of T)(Sub(item)
                                           ' Add the item to the queue.
                                           ' Remove the oldest item when the queue size exceeds the window
size.
                                           ' Post the data in the queue to the source block when the queue
size
                                           ' equals the window size.
                                           queue.Enqueue(item)
                                           If queue.Count > windowSize Then
                                               queue.Dequeue()
                                           End If
                                           If queue.Count = windowSize Then
                                               source.Post(queue.ToArray())
                                           End If
                                       End Sub)

    ' When the target is set to the completed state, propagate out any
    ' remaining data and set the source to the completed state.
    target.Completion.ContinueWith(Sub()
                                       If queue.Count > 0 AndAlso queue.Count < windowSize Then
                                           source.Post(queue.ToArray())
                                       End If
                                       source.Complete()
                                   End Sub)

    ' Return a IPropagatorBlock<T, T[]> object that encapsulates the
    ' target and source blocks.
    Return DataflowBlock.Encapsulate(target, source)
End Function
```

派生自 IPropagatorBlock 以定义滑动窗口数据流块

下面的示例展示了 `SlidingWindowBlock` 类。此类派生自 IPropagatorBlock<TInput,TOutput>，可用作数据的源和目标。与上一示例中所述一样，`SlidingWindowBlock` 类是在现有数据流块类型的基础之上构建而成。不同之处在于，`SlidingWindowBlock` 类还实现了 ISourceBlock<TOutput>、ITargetBlock<TInput> 和 IDataflowBlock 接口所需的方法。这些方法全都将工作转发给预定义的数据流块类型成员。例如，`Post` 方法将工作转交给也是 ITargetBlock<TInput> 对象的 `m_target` 数据成员。

如果需要自定义数据流功能，还需要提供其他方法、属性或字段的类型，此方式就很有用。例如，`SlidingWindowBlock` 类也派生自 IReceivableSourceBlock<TOutput>，这样就可以提供 TryReceive 和 TryReceiveAll 方法了。`SlidingWindowBlock` 类还具有扩展性，体现在提供 `WindowSize` 属性，以检索滑动窗口中的元素数量。

```
// Propagates data in a sliding window fashion.
public class SlidingWindowBlock<T> : IPropagatorBlock<T, T[]>,
                                     IReceivableSourceBlock<T[]>
{
   // The size of the window.
   private readonly int m_windowSize;
   // The target part of the block.
   private readonly ITargetBlock<T> m_target;
```

```
// The source part of the block.
private readonly IReceivableSourceBlock<T[]> m_source;

// Constructs a SlidingWindowBlock object.
public SlidingWindowBlock(int windowSize)
{
   // Create a queue to hold messages.
   var queue = new Queue<T>();

   // The source part of the propagator holds arrays of size windowSize
   // and propagates data out to any connected targets.
   var source = new BufferBlock<T[]>();

   // The target part receives data and adds them to the queue.
   var target = new ActionBlock<T>(item =>
   {
      // Add the item to the queue.
      queue.Enqueue(item);
      // Remove the oldest item when the queue size exceeds the window size.
      if (queue.Count > windowSize)
         queue.Dequeue();
      // Post the data in the queue to the source block when the queue size
      // equals the window size.
      if (queue.Count == windowSize)
         source.Post(queue.ToArray());
   });

   // When the target is set to the completed state, propagate out any
   // remaining data and set the source to the completed state.
   target.Completion.ContinueWith(delegate
   {
      if (queue.Count > 0 && queue.Count < windowSize)
         source.Post(queue.ToArray());
      source.Complete();
   });

   m_windowSize = windowSize;
   m_target = target;
   m_source = source;
}

// Retrieves the size of the window.
public int WindowSize { get { return m_windowSize; } }

#region IReceivableSourceBlock<TOutput> members

// Attempts to synchronously receive an item from the source.
public bool TryReceive(Predicate<T[]> filter, out T[] item)
{
   return m_source.TryReceive(filter, out item);
}

// Attempts to remove all available elements from the source into a new
// array that is returned.
public bool TryReceiveAll(out IList<T[]> items)
{
   return m_source.TryReceiveAll(out items);
}

#endregion

#region ISourceBlock<TOutput> members

// Links this dataflow block to the provided target.
public IDisposable LinkTo(ITargetBlock<T[]> target, DataflowLinkOptions linkOptions)
{
   return m_source.LinkTo(target, linkOptions);
}
```

```
   // Called by a target to reserve a message previously offered by a source
   // but not yet consumed by this target.
   bool ISourceBlock<T[]>.ReserveMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target)
   {
      return m_source.ReserveMessage(messageHeader, target);
   }

   // Called by a target to consume a previously offered message from a source.
   T[] ISourceBlock<T[]>.ConsumeMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target, out bool messageConsumed)
   {
      return m_source.ConsumeMessage(messageHeader,
         target, out messageConsumed);
   }

   // Called by a target to release a previously reserved message from a source.
   void ISourceBlock<T[]>.ReleaseReservation(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target)
   {
      m_source.ReleaseReservation(messageHeader, target);
   }

   #endregion

   #region ITargetBlock<TInput> members

   // Asynchronously passes a message to the target block, giving the target the
   // opportunity to consume the message.
   DataflowMessageStatus ITargetBlock<T>.OfferMessage(DataflowMessageHeader messageHeader,
      T messageValue, ISourceBlock<T> source, bool consumeToAccept)
   {
      return m_target.OfferMessage(messageHeader,
         messageValue, source, consumeToAccept);
   }

   #endregion

   #region IDataflowBlock members

   // Gets a Task that represents the completion of this dataflow block.
   public Task Completion { get { return m_source.Completion; } }

   // Signals to this target block that it should not accept any more messages,
   // nor consume postponed messages.
   public void Complete()
   {
      m_target.Complete();
   }

   public void Fault(Exception error)
   {
      m_target.Fault(error);
   }

   #endregion
}
```

```
    ' Propagates data in a sliding window fashion.
    Public Class SlidingWindowBlock(Of T)
        Implements IPropagatorBlock(Of T, T()), IReceivableSourceBlock(Of T())
        ' The size of the window.
        Private ReadOnly m_windowSize As Integer
        ' The target part of the block.
        Private ReadOnly m_target As ITargetBlock(Of T)
        ' The source part of the block.
        Private ReadOnly m_source As IReceivableSourceBlock(Of T())
```

```
        ' Constructs a SlidingWindowBlock object.
        Public Sub New(ByVal windowSize As Integer)
            ' Create a queue to hold messages.
            Dim queue = New Queue(Of T)()

            ' The source part of the propagator holds arrays of size windowSize
            ' and propagates data out to any connected targets.
            Dim source = New BufferBlock(Of T())()

            ' The target part receives data and adds them to the queue.
            Dim target = New ActionBlock(Of T)(Sub(item)
                                                   ' Add the item to the queue.
                                                   ' Remove the oldest item when the queue size exceeds the
window size.
                                                   ' Post the data in the queue to the source block when the
queue size
                                                   ' equals the window size.
                                                   queue.Enqueue(item)
                                                   If queue.Count > windowSize Then
                                                       queue.Dequeue()
                                                   End If
                                                   If queue.Count = windowSize Then
                                                       source.Post(queue.ToArray())
                                                   End If
                                               End Sub)

            ' When the target is set to the completed state, propagate out any
            ' remaining data and set the source to the completed state.
            target.Completion.ContinueWith(Sub()
                                               If queue.Count > 0 AndAlso queue.Count < windowSize Then
                                                   source.Post(queue.ToArray())
                                               End If
                                               source.Complete()
                                           End Sub)

            m_windowSize = windowSize
            m_target = target
            m_source = source
        End Sub

        ' Retrieves the size of the window.
        Public ReadOnly Property WindowSize() As Integer
            Get
                Return m_windowSize
            End Get
        End Property

        '#Region "IReceivableSourceBlock<TOutput> members"

        ' Attempts to synchronously receive an item from the source.
        Public Function TryReceive(ByVal filter As Predicate(Of T()), <System.Runtime.InteropServices.Out()>
ByRef item() As T) As Boolean Implements IReceivableSourceBlock(Of T()).TryReceive
            Return m_source.TryReceive(filter, item)
        End Function

        ' Attempts to remove all available elements from the source into a new
        ' array that is returned.
        Public Function TryReceiveAll(<System.Runtime.InteropServices.Out()> ByRef items As IList(Of T()))
As Boolean Implements IReceivableSourceBlock(Of T()).TryReceiveAll
            Return m_source.TryReceiveAll(items)
        End Function

        '#End Region

#Region "ISourceBlock<TOutput> members"

        ' Links this dataflow block to the provided target.
        Public Function LinkTo(ByVal target As ITargetBlock(Of T()), ByVal linkOptions As
```

```
DataflowLinkOptions) As IDisposable Implements ISourceBlock(Of T()).LinkTo
            Return m_source.LinkTo(target, linkOptions)
        End Function

        ' Called by a target to reserve a message previously offered by a source
        ' but not yet consumed by this target.
        Private Function ReserveMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) As Boolean Implements ISourceBlock(Of T()).ReserveMessage
            Return m_source.ReserveMessage(messageHeader, target)
        End Function

        ' Called by a target to consume a previously offered message from a source.
        Private Function ConsumeMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T()), ByRef messageConsumed As Boolean) As T() Implements ISourceBlock(Of
T()).ConsumeMessage
            Return m_source.ConsumeMessage(messageHeader, target, messageConsumed)
        End Function

        ' Called by a target to release a previously reserved message from a source.
        Private Sub ReleaseReservation(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) Implements ISourceBlock(Of T()).ReleaseReservation
            m_source.ReleaseReservation(messageHeader, target)
        End Sub

#End Region

#Region "ITargetBlock<TInput> members"

        ' Asynchronously passes a message to the target block, giving the target the
        ' opportunity to consume the message.
        Private Function OfferMessage(ByVal messageHeader As DataflowMessageHeader, ByVal messageValue As T,
ByVal source As ISourceBlock(Of T), ByVal consumeToAccept As Boolean) As DataflowMessageStatus Implements
ITargetBlock(Of T).OfferMessage
            Return m_target.OfferMessage(messageHeader, messageValue, source, consumeToAccept)
        End Function

#End Region

#Region "IDataflowBlock members"

        ' Gets a Task that represents the completion of this dataflow block.
        Public ReadOnly Property Completion() As Task Implements IDataflowBlock.Completion
            Get
                Return m_source.Completion
            End Get
        End Property

        ' Signals to this target block that it should not accept any more messages,
        ' nor consume postponed messages.
        Public Sub Complete() Implements IDataflowBlock.Complete
            m_target.Complete()
        End Sub

        Public Sub Fault(ByVal [error] As Exception) Implements IDataflowBlock.Fault
            m_target.Fault([error])
        End Sub

#End Region
    End Class
```

完整示例

下面的示例显示此演练的完整代码。它还展示了如何在对块执行写入和读取操作并将结果打印到控制台的方法中使用两个滑动窗口块。

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to create a custom dataflow block type.
class Program
{
   // Creates a IPropagatorBlock<T, T[]> object propagates data in a
   // sliding window fashion.
   public static IPropagatorBlock<T, T[]> CreateSlidingWindow<T>(int windowSize)
   {
      // Create a queue to hold messages.
      var queue = new Queue<T>();

      // The source part of the propagator holds arrays of size windowSize
      // and propagates data out to any connected targets.
      var source = new BufferBlock<T[]>();

      // The target part receives data and adds them to the queue.
      var target = new ActionBlock<T>(item =>
      {
         // Add the item to the queue.
         queue.Enqueue(item);
         // Remove the oldest item when the queue size exceeds the window size.
         if (queue.Count > windowSize)
            queue.Dequeue();
         // Post the data in the queue to the source block when the queue size
         // equals the window size.
         if (queue.Count == windowSize)
            source.Post(queue.ToArray());
      });

      // When the target is set to the completed state, propagate out any
      // remaining data and set the source to the completed state.
      target.Completion.ContinueWith(delegate
      {
         if (queue.Count > 0 && queue.Count < windowSize)
            source.Post(queue.ToArray());
         source.Complete();
      });

      // Return a IPropagatorBlock<T, T[]> object that encapsulates the
      // target and source blocks.
      return DataflowBlock.Encapsulate(target, source);
   }

   // Propagates data in a sliding window fashion.
   public class SlidingWindowBlock<T> : IPropagatorBlock<T, T[]>,
                                        IReceivableSourceBlock<T[]>
   {
      // The size of the window.
      private readonly int m_windowSize;
      // The target part of the block.
      private readonly ITargetBlock<T> m_target;
      // The source part of the block.
      private readonly IReceivableSourceBlock<T[]> m_source;

      // Constructs a SlidingWindowBlock object.
      public SlidingWindowBlock(int windowSize)
      {
         // Create a queue to hold messages.
         var queue = new Queue<T>();

         // The source part of the propagator holds arrays of size windowSize
         // and propagates data out to any connected targets.
         var source = new BufferBlock<T[]>();

         // The target part receives data and adds them to the queue.
```

```
      // The target part receives data and adds them to the queue.
      var target = new ActionBlock<T>(item =>
      {
         // Add the item to the queue.
         queue.Enqueue(item);
         // Remove the oldest item when the queue size exceeds the window size.
         if (queue.Count > windowSize)
            queue.Dequeue();
         // Post the data in the queue to the source block when the queue size
         // equals the window size.
         if (queue.Count == windowSize)
            source.Post(queue.ToArray());
      });

      // When the target is set to the completed state, propagate out any
      // remaining data and set the source to the completed state.
      target.Completion.ContinueWith(delegate
      {
         if (queue.Count > 0 && queue.Count < windowSize)
            source.Post(queue.ToArray());
         source.Complete();
      });

      m_windowSize = windowSize;
      m_target = target;
      m_source = source;
   }

   // Retrieves the size of the window.
   public int WindowSize { get { return m_windowSize; } }

   #region IReceivableSourceBlock<TOutput> members

   // Attempts to synchronously receive an item from the source.
   public bool TryReceive(Predicate<T[]> filter, out T[] item)
   {
      return m_source.TryReceive(filter, out item);
   }

   // Attempts to remove all available elements from the source into a new
   // array that is returned.
   public bool TryReceiveAll(out IList<T[]> items)
   {
      return m_source.TryReceiveAll(out items);
   }

   #endregion

   #region ISourceBlock<TOutput> members

   // Links this dataflow block to the provided target.
   public IDisposable LinkTo(ITargetBlock<T[]> target, DataflowLinkOptions linkOptions)
   {
      return m_source.LinkTo(target, linkOptions);
   }

   // Called by a target to reserve a message previously offered by a source
   // but not yet consumed by this target.
   bool ISourceBlock<T[]>.ReserveMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target)
   {
      return m_source.ReserveMessage(messageHeader, target);
   }

   // Called by a target to consume a previously offered message from a source.
   T[] ISourceBlock<T[]>.ConsumeMessage(DataflowMessageHeader messageHeader,
      ITargetBlock<T[]> target, out bool messageConsumed)
   {
      return m_source.ConsumeMessage(messageHeader,
         target, out messageConsumed);
```

```
            target, out messageConsumed);
      }

      // Called by a target to release a previously reserved message from a source.
      void ISourceBlock<T[]>.ReleaseReservation(DataflowMessageHeader messageHeader,
         ITargetBlock<T[]> target)
      {
         m_source.ReleaseReservation(messageHeader, target);
      }

      #endregion

      #region ITargetBlock<TInput> members

      // Asynchronously passes a message to the target block, giving the target the
      // opportunity to consume the message.
      DataflowMessageStatus ITargetBlock<T>.OfferMessage(DataflowMessageHeader messageHeader,
         T messageValue, ISourceBlock<T> source, bool consumeToAccept)
      {
         return m_target.OfferMessage(messageHeader,
            messageValue, source, consumeToAccept);
      }

      #endregion

      #region IDataflowBlock members

      // Gets a Task that represents the completion of this dataflow block.
      public Task Completion { get { return m_source.Completion; } }

      // Signals to this target block that it should not accept any more messages,
      // nor consume postponed messages.
      public void Complete()
      {
         m_target.Complete();
      }

      public void Fault(Exception error)
      {
         m_target.Fault(error);
      }

      #endregion
   }

   // Demonstrates usage of the sliding window block by sending the provided
   // values to the provided propagator block and printing the output of
   // that block to the console.
   static void DemonstrateSlidingWindow<T>(IPropagatorBlock<T, T[]> slidingWindow,
      IEnumerable<T> values)
   {
      // Create an action block that prints arrays of data to the console.
      string windowComma = string.Empty;
      var printWindow = new ActionBlock<T[]>(window =>
      {
         Console.Write(windowComma);
         Console.Write("{");

         string comma = string.Empty;
         foreach (T item in window)
         {
            Console.Write(comma);
            Console.Write(item);
            comma = ",";
         }
         Console.Write("}");

         windowComma = ", ";
      });
```

```
      // Link the printer block to the sliding window block.
      slidingWindow.LinkTo(printWindow);

      // Set the printer block to the completed state when the sliding window
      // block completes.
      slidingWindow.Completion.ContinueWith(delegate { printWindow.Complete(); });

      // Print an additional newline to the console when the printer block completes.
      var completion = printWindow.Completion.ContinueWith(delegate { Console.WriteLine(); });

      // Post the provided values to the sliding window block and then wait
      // for the sliding window block to complete.
      foreach (T value in values)
      {
         slidingWindow.Post(value);
      }
      slidingWindow.Complete();

      // Wait for the printer to complete and perform its final action.
      completion.Wait();
   }

   static void Main(string[] args)
   {

      Console.Write("Using the DataflowBlockExtensions.Encapsulate method ");
      Console.WriteLine("(T=int, windowSize=3):");
      DemonstrateSlidingWindow(CreateSlidingWindow<int>(3), Enumerable.Range(0, 10));

      Console.WriteLine();

      var slidingWindow = new SlidingWindowBlock<char>(4);

      Console.Write("Using SlidingWindowBlock<T> ");
      Console.WriteLine("(T=char, windowSize={0}):", slidingWindow.WindowSize);
      DemonstrateSlidingWindow(slidingWindow, from n in Enumerable.Range(65, 10)
                                              select (char)n);
   }
}

/* Output:
Using the DataflowBlockExtensions.Encapsulate method (T=int, windowSize=3):
{0,1,2}, {1,2,3}, {2,3,4}, {3,4,5}, {4,5,6}, {5,6,7}, {6,7,8}, {7,8,9}

Using SlidingWindowBlock<T> (T=char, windowSize=4):
{A,B,C,D}, {B,C,D,E}, {C,D,E,F}, {D,E,F,G}, {E,F,G,H}, {F,G,H,I}, {G,H,I,J}
 */
```

```
Imports System.Collections.Generic
Imports System.Linq
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to create a custom dataflow block type.
Friend Class Program
    ' Creates a IPropagatorBlock<T, T[]> object propagates data in a
    ' sliding window fashion.
    Public Shared Function CreateSlidingWindow(Of T)(ByVal windowSize As Integer) As IPropagatorBlock(Of T,
T())
        ' Create a queue to hold messages.
        Dim queue = New Queue(Of T)()

        ' The source part of the propagator holds arrays of size windowSize
        ' and propagates data out to any connected targets.
        Dim source = New BufferBlock(Of T())()

        ' The target part receives data and adds them to the queue.
```

```
        ' The target part receives data and adds them to the queue.
        Dim target = New ActionBlock(Of T)(Sub(item)
                                               ' Add the item to the queue.
                                               ' Remove the oldest item when the queue size exceeds the
window size.
                                               ' Post the data in the queue to the source block when the
queue size
                                               ' equals the window size.
                                               queue.Enqueue(item)
                                               If queue.Count > windowSize Then
                                                   queue.Dequeue()
                                               End If
                                               If queue.Count = windowSize Then
                                                   source.Post(queue.ToArray())
                                               End If
                                           End Sub)

        ' When the target is set to the completed state, propagate out any
        ' remaining data and set the source to the completed state.
        target.Completion.ContinueWith(Sub()
                                           If queue.Count > 0 AndAlso queue.Count < windowSize Then
                                               source.Post(queue.ToArray())
                                           End If
                                           source.Complete()
                                       End Sub)

        ' Return a IPropagatorBlock<T, T[]> object that encapsulates the
        ' target and source blocks.
        Return DataflowBlock.Encapsulate(target, source)
    End Function

    ' Propagates data in a sliding window fashion.
    Public Class SlidingWindowBlock(Of T)
        Implements IPropagatorBlock(Of T, T()), IReceivableSourceBlock(Of T())
        ' The size of the window.
        Private ReadOnly m_windowSize As Integer
        ' The target part of the block.
        Private ReadOnly m_target As ITargetBlock(Of T)
        ' The source part of the block.
        Private ReadOnly m_source As IReceivableSourceBlock(Of T())

        ' Constructs a SlidingWindowBlock object.
        Public Sub New(ByVal windowSize As Integer)
            ' Create a queue to hold messages.
            Dim queue = New Queue(Of T)()

            ' The source part of the propagator holds arrays of size windowSize
            ' and propagates data out to any connected targets.
            Dim source = New BufferBlock(Of T())()

            ' The target part receives data and adds them to the queue.
            Dim target = New ActionBlock(Of T)(Sub(item)
                                                   ' Add the item to the queue.
                                                   ' Remove the oldest item when the queue size exceeds the
window size.
                                                   ' Post the data in the queue to the source block when the
queue size
                                                   ' equals the window size.
                                                   queue.Enqueue(item)
                                                   If queue.Count > windowSize Then
                                                       queue.Dequeue()
                                                   End If
                                                   If queue.Count = windowSize Then
                                                       source.Post(queue.ToArray())
                                                   End If
                                               End Sub)

            ' When the target is set to the completed state, propagate out any
            ' remaining data and set the source to the completed state.
```

```
            target.Completion.ContinueWith(Sub()
                                               If queue.Count > 0 AndAlso queue.Count < windowSize Then
                                                   source.Post(queue.ToArray())
                                               End If
                                               source.Complete()
                                           End Sub)

            m_windowSize = windowSize
            m_target = target
            m_source = source
        End Sub

        ' Retrieves the size of the window.
        Public ReadOnly Property WindowSize() As Integer
            Get
                Return m_windowSize
            End Get
        End Property

        '#Region "IReceivableSourceBlock<TOutput> members"

        ' Attempts to synchronously receive an item from the source.
        Public Function TryReceive(ByVal filter As Predicate(Of T()), <System.Runtime.InteropServices.Out()>
ByRef item() As T) As Boolean Implements IReceivableSourceBlock(Of T()).TryReceive
            Return m_source.TryReceive(filter, item)
        End Function

        ' Attempts to remove all available elements from the source into a new
        ' array that is returned.
        Public Function TryReceiveAll(<System.Runtime.InteropServices.Out()> ByRef items As IList(Of T()))
As Boolean Implements IReceivableSourceBlock(Of T()).TryReceiveAll
            Return m_source.TryReceiveAll(items)
        End Function

        '#End Region

#Region "ISourceBlock<TOutput> members"

        ' Links this dataflow block to the provided target.
        Public Function LinkTo(ByVal target As ITargetBlock(Of T()), ByVal linkOptions As
DataflowLinkOptions) As IDisposable Implements ISourceBlock(Of T()).LinkTo
            Return m_source.LinkTo(target, linkOptions)
        End Function

        ' Called by a target to reserve a message previously offered by a source
        ' but not yet consumed by this target.
        Private Function ReserveMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) As Boolean Implements ISourceBlock(Of T()).ReserveMessage
            Return m_source.ReserveMessage(messageHeader, target)
        End Function

        ' Called by a target to consume a previously offered message from a source.
        Private Function ConsumeMessage(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T()), ByRef messageConsumed As Boolean) As T() Implements ISourceBlock(Of
T()).ConsumeMessage
            Return m_source.ConsumeMessage(messageHeader, target, messageConsumed)
        End Function

        ' Called by a target to release a previously reserved message from a source.
        Private Sub ReleaseReservation(ByVal messageHeader As DataflowMessageHeader, ByVal target As
ITargetBlock(Of T())) Implements ISourceBlock(Of T()).ReleaseReservation
            m_source.ReleaseReservation(messageHeader, target)
        End Sub

#End Region

#Region "ITargetBlock<TInput> members"

        ' Asynchronously passes a message to the target block, giving the target the
```

```
        ' opportunity to consume the message.
        Private Function OfferMessage(ByVal messageHeader As DataflowMessageHeader, ByVal messageValue As T,
ByVal source As ISourceBlock(Of T), ByVal consumeToAccept As Boolean) As DataflowMessageStatus Implements
ITargetBlock(Of T).OfferMessage
            Return m_target.OfferMessage(messageHeader, messageValue, source, consumeToAccept)
        End Function

#End Region

#Region "IDataflowBlock members"

        ' Gets a Task that represents the completion of this dataflow block.
        Public ReadOnly Property Completion() As Task Implements IDataflowBlock.Completion
            Get
                Return m_source.Completion
            End Get
        End Property

        ' Signals to this target block that it should not accept any more messages,
        ' nor consume postponed messages.
        Public Sub Complete() Implements IDataflowBlock.Complete
            m_target.Complete()
        End Sub

        Public Sub Fault(ByVal [error] As Exception) Implements IDataflowBlock.Fault
            m_target.Fault([error])
        End Sub

#End Region
    End Class

    ' Demonstrates usage of the sliding window block by sending the provided
    ' values to the provided propagator block and printing the output of
    ' that block to the console.
    Private Shared Sub DemonstrateSlidingWindow(Of T)(ByVal slidingWindow As IPropagatorBlock(Of T, T()),
ByVal values As IEnumerable(Of T))
        ' Create an action block that prints arrays of data to the console.
        Dim windowComma As String = String.Empty
        Dim printWindow = New ActionBlock(Of T())(Sub(window)
                                                      Console.Write(windowComma)
                                                      Console.Write("{")
                                                      Dim comma As String = String.Empty
                                                      For Each item As T In window
                                                          Console.Write(comma)
                                                          Console.Write(item)
                                                          comma = ","
                                                      Next item
                                                      Console.Write("}")
                                                      windowComma = ", "
                                                  End Sub)

        ' Link the printer block to the sliding window block.
        slidingWindow.LinkTo(printWindow)

        ' Set the printer block to the completed state when the sliding window
        ' block completes.
        slidingWindow.Completion.ContinueWith(Sub() printWindow.Complete())

        ' Print an additional newline to the console when the printer block completes.
        Dim completion = printWindow.Completion.ContinueWith(Sub() Console.WriteLine())

        ' Post the provided values to the sliding window block and then wait
        ' for the sliding window block to complete.
        For Each value As T In values
            slidingWindow.Post(value)
        Next value
        slidingWindow.Complete()

        ' Wait for the printer to complete and perform its final action.
```

```
        completion.Wait()
    End Sub

    Shared Sub Main(ByVal args() As String)

        Console.Write("Using the DataflowBlockExtensions.Encapsulate method ")
        Console.WriteLine("(T=int, windowSize=3):")
        DemonstrateSlidingWindow(CreateSlidingWindow(Of Integer)(3), Enumerable.Range(0, 10))

        Console.WriteLine()

        Dim slidingWindow = New SlidingWindowBlock(Of Char)(4)

        Console.Write("Using SlidingWindowBlock<T> ")
        Console.WriteLine("(T=char, windowSize={0}):", slidingWindow.WindowSize)
        DemonstrateSlidingWindow(slidingWindow, _
            From n In Enumerable.Range(65, 10) _
            Select ChrW(n))
    End Sub
End Class

' Output:
'Using the DataflowBlockExtensions.Encapsulate method (T=int, windowSize=3):
'{0,1,2}, {1,2,3}, {2,3,4}, {3,4,5}, {4,5,6}, {5,6,7}, {6,7,8}, {7,8,9}
'
'Using SlidingWindowBlock<T> (T=char, windowSize=4):
'{A,B,C,D}, {B,C,D,E}, {C,D,E,F}, {D,E,F,G}, {E,F,G,H}, {F,G,H,I}, {G,H,I,J}
'
```

另请参阅

- 数据流

如何：使用 JoinBlock 从多个源读取数据

2021/11/16 •

本文档介绍如何在来自多个源的数据可用时使用 JoinBlock<T1,T2> 类执行操作。还演示了如何使用非贪婪模式使多个联接块更有效地共享数据源。

> **NOTE**
> TPL 数据流库(System.Threading.Tasks.Dataflow 命名空间)不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

示例

下面的示例定义了三种资源类型(`NetworkResource`、`FileResource` 和 `MemoryResource`)并在资源可用时执行操作。此示例需要使用 `NetworkResource` 和 `MemoryResource` 对才能执行第一个操作，需要使用 `FileResource` 和 `MemoryResource` 对才能执行第二个操作。为了在所有所需资源可用时启用这些操作，此示例使用 JoinBlock<T1,T2> 类。当 JoinBlock<T1,T2> 对象从所有源接收数据时，它会将该数据传播到其目标，其目标在本示例中为 ActionBlock<TInput> 对象。两个 JoinBlock<T1,T2> 对象都从 `MemoryResource` 对象的共享池中读取。

```
using System;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to use non-greedy join blocks to distribute
// resources among a dataflow network.
class Program
{
   // Represents a resource. A derived class might represent
   // a limited resource such as a memory, network, or I/O
   // device.
   abstract class Resource
   {
   }

   // Represents a memory resource. For brevity, the details of
   // this class are omitted.
   class MemoryResource : Resource
   {
   }

   // Represents a network resource. For brevity, the details of
   // this class are omitted.
   class NetworkResource : Resource
   {
   }

   // Represents a file resource. For brevity, the details of
   // this class are omitted.
   class FileResource : Resource
   {
   }

   static void Main(string[] args)
   {
```

```
{
   // Create three BufferBlock<T> objects. Each object holds a different
   // type of resource.
   var networkResources = new BufferBlock<NetworkResource>();
   var fileResources = new BufferBlock<FileResource>();
   var memoryResources = new BufferBlock<MemoryResource>();

   // Create two non-greedy JoinBlock<T1, T2> objects.
   // The first join works with network and memory resources;
   // the second pool works with file and memory resources.

   var joinNetworkAndMemoryResources =
      new JoinBlock<NetworkResource, MemoryResource>(
         new GroupingDataflowBlockOptions
         {
            Greedy = false
         });

   var joinFileAndMemoryResources =
      new JoinBlock<FileResource, MemoryResource>(
         new GroupingDataflowBlockOptions
         {
            Greedy = false
         });

   // Create two ActionBlock<T> objects.
   // The first block acts on a network resource and a memory resource.
   // The second block acts on a file resource and a memory resource.

   var networkMemoryAction =
      new ActionBlock<Tuple<NetworkResource, MemoryResource>>(
         data =>
         {
            // Perform some action on the resources.

            // Print a message.
            Console.WriteLine("Network worker: using resources...");

            // Simulate a lengthy operation that uses the resources.
            Thread.Sleep(new Random().Next(500, 2000));

            // Print a message.
            Console.WriteLine("Network worker: finished using resources...");

            // Release the resources back to their respective pools.
            networkResources.Post(data.Item1);
            memoryResources.Post(data.Item2);
         });

   var fileMemoryAction =
      new ActionBlock<Tuple<FileResource, MemoryResource>>(
         data =>
         {
            // Perform some action on the resources.

            // Print a message.
            Console.WriteLine("File worker: using resources...");

            // Simulate a lengthy operation that uses the resources.
            Thread.Sleep(new Random().Next(500, 2000));

            // Print a message.
            Console.WriteLine("File worker: finished using resources...");

            // Release the resources back to their respective pools.
            fileResources.Post(data.Item1);
            memoryResources.Post(data.Item2);
         });
```

```
      // Link the resource pools to the JoinBlock<T1, T2> objects.
      // Because these join blocks operate in non-greedy mode, they do not
      // take the resource from a pool until all resources are available from
      // all pools.

      networkResources.LinkTo(joinNetworkAndMemoryResources.Target1);
      memoryResources.LinkTo(joinNetworkAndMemoryResources.Target2);

      fileResources.LinkTo(joinFileAndMemoryResources.Target1);
      memoryResources.LinkTo(joinFileAndMemoryResources.Target2);

      // Link the JoinBlock<T1, T2> objects to the ActionBlock<T> objects.

      joinNetworkAndMemoryResources.LinkTo(networkMemoryAction);
      joinFileAndMemoryResources.LinkTo(fileMemoryAction);

      // Populate the resource pools. In this example, network and
      // file resources are more abundant than memory resources.

      networkResources.Post(new NetworkResource());
      networkResources.Post(new NetworkResource());
      networkResources.Post(new NetworkResource());

      memoryResources.Post(new MemoryResource());

      fileResources.Post(new FileResource());
      fileResources.Post(new FileResource());
      fileResources.Post(new FileResource());

      // Allow data to flow through the network for several seconds.
      Thread.Sleep(10000);
   }
}

/* Sample output:
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
File worker: using resources...
File worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
Network worker: using resources...
Network worker: finished using resources...
File worker: using resources...
*/
```

```
Imports System.Threading
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to use non-greedy join blocks to distribute
' resources among a dataflow network.
Friend Class Program
    ' Represents a resource. A derived class might represent
    ' a limited resource such as a memory, network, or I/O
    ' device.
    Private MustInherit Class Resource
    End Class

    ' Represents a memory resource. For brevity, the details of
    ' this class are omitted.
```

```
    ' this class are omitted.
    Private Class MemoryResource
        Inherits Resource
    End Class

    ' Represents a network resource. For brevity, the details of
    ' this class are omitted.
    Private Class NetworkResource
        Inherits Resource
    End Class

    ' Represents a file resource. For brevity, the details of
    ' this class are omitted.
    Private Class FileResource
        Inherits Resource
    End Class

    Shared Sub Main(ByVal args() As String)
        ' Create three BufferBlock<T> objects. Each object holds a different
        ' type of resource.
        Dim networkResources = New BufferBlock(Of NetworkResource)()
        Dim fileResources = New BufferBlock(Of FileResource)()
        Dim memoryResources = New BufferBlock(Of MemoryResource)()

        ' Create two non-greedy JoinBlock<T1, T2> objects.
        ' The first join works with network and memory resources;
        ' the second pool works with file and memory resources.

        Dim joinNetworkAndMemoryResources = New JoinBlock(Of NetworkResource, MemoryResource)(New
GroupingDataflowBlockOptions With {.Greedy = False})

        Dim joinFileAndMemoryResources = New JoinBlock(Of FileResource, MemoryResource)(New
GroupingDataflowBlockOptions With {.Greedy = False})

        ' Create two ActionBlock<T> objects.
        ' The first block acts on a network resource and a memory resource.
        ' The second block acts on a file resource and a memory resource.

        Dim networkMemoryAction = New ActionBlock(Of Tuple(Of NetworkResource, MemoryResource))(Sub(data)
                                                                                                '
Perform some action on the resources.
                                                                                                ' Print
a message.
                                                                                                '
Simulate a lengthy operation that uses the resources.
                                                                                                ' Print
a message.
                                                                                                '
Release the resources back to their respective pools.

Console.WriteLine("Network worker: using resources...")

Thread.Sleep(New Random().Next(500, 2000))

Console.WriteLine("Network worker: finished using resources...")

networkResources.Post(data.Item1)

memoryResources.Post(data.Item2)
                                                                                                  End Sub)

        Dim fileMemoryAction = New ActionBlock(Of Tuple(Of FileResource, MemoryResource))(Sub(data)
                                                                                                ' Perform some
action on the resources.
                                                                                                ' Print a
message.
                                                                                                ' Simulate a
lengthy operation that uses the resources.
                                                                                                ' Print a
```

```
message.                                                                                    ' Release the
resources back to their respective pools.

Console.WriteLine("File worker: using resources...")

Thread.Sleep(New Random().Next(500, 2000))

Console.WriteLine("File worker: finished using resources...")

fileResources.Post(data.Item1)

memoryResources.Post(data.Item2)
                                                                                End Sub)

        ' Link the resource pools to the JoinBlock<T1, T2> objects.
        ' Because these join blocks operate in non-greedy mode, they do not
        ' take the resource from a pool until all resources are available from
        ' all pools.

        networkResources.LinkTo(joinNetworkAndMemoryResources.Target1)
        memoryResources.LinkTo(joinNetworkAndMemoryResources.Target2)

        fileResources.LinkTo(joinFileAndMemoryResources.Target1)
        memoryResources.LinkTo(joinFileAndMemoryResources.Target2)

        ' Link the JoinBlock<T1, T2> objects to the ActionBlock<T> objects.

        joinNetworkAndMemoryResources.LinkTo(networkMemoryAction)
        joinFileAndMemoryResources.LinkTo(fileMemoryAction)

        ' Populate the resource pools. In this example, network and
        ' file resources are more abundant than memory resources.

        networkResources.Post(New NetworkResource())
        networkResources.Post(New NetworkResource())
        networkResources.Post(New NetworkResource())

        memoryResources.Post(New MemoryResource())

        fileResources.Post(New FileResource())
        fileResources.Post(New FileResource())
        fileResources.Post(New FileResource())

        ' Allow data to flow through the network for several seconds.
        Thread.Sleep(10000)

    End Sub

End Class

' Sample output:
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'File worker: using resources...
'File worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'Network worker: using resources...
'Network worker: finished using resources...
'File worker: using resources...
```

```
'
```

为了实现 `MemoryResource` 对象共享池的高效使用，此示例指定了一个 GroupingDataflowBlockOptions 对象，该对象将 Greedy 属性设置为 `False` 以创建在非贪婪模式下运行的 JoinBlock<T1,T2> 对象。非贪婪联接块会推迟所有传入的消息，直至从每个源收到一条消息。如果任何推迟的消息由另一个块接受，联接块将重新启动该进程。非贪婪模式使联接块能够共享一个或多个源块，以便在其他块等待数据时能够使进程向前推进。在此示例中，如果将 `MemoryResource` 对象添加到 `memoryResources` 池中，那么要接收第二个数据源的第一个联接块可以将进程向前推进。如果此示例使用贪婪模式(默认模式)，一个联接块可能会接受 `MemoryResource` 对象，然后等待第二个资源变为可用。但是，如果另一个联接块有自己的第二个可用数据源，则它无法使进程向前推进，因为 `MemoryResource` 对象已被另一联接块占用。

可靠编程

使用非贪婪联接还有助于防止应用程序中出现死锁。在软件应用中，如果两个或多个进程分别留有资源，且相互等待另一进程释放其他资源，就会发生死锁 。考虑一个定义两个 JoinBlock<T1,T2> 对象的应用程序。两个对象都从两个共享源块读取数据。在贪婪模式下，如果一个联接块从第一个源读取，第二个联接块从第二个源读取，则应用程序可能发生死锁，原因是两个联接块相互等待另一个联接块释放其资源。在非贪婪模式下，每个联接块只在所有数据可用时才从其源读取，因此消除了死锁风险。

另请参阅

- 数据流

如何：指定数据流块中的并行度

2021/11/16 •

本文档介绍如何设置 ExecutionDataflowBlockOptions.MaxDegreeOfParallelism 属性使执行数据流块一次处理多条消息。当数据流块需要执行长时间运行的计算并且可从并行处理消息中获益时，这种做法很有用。此示例使用 System.Threading.Tasks.Dataflow.ActionBlock<TInput> 类并发执行多个数据流操作；但是，您可以对 TPL 数据流库提供的任何预定义的执行块类型（ActionBlock<TInput>、System.Threading.Tasks.Dataflow.TransformBlock<TInput,TOutput> 和 System.Threading.Tasks.Dataflow.TransformManyBlock<TInput,TOutput>）指定最大并行度。

> **NOTE**
>
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

示例

下面的示例执行两个数据流计算并输出每个计算所需的运行时间。第一个计算指定最大并行度为 1，这是默认值。最大并行度 1 会使数据流块按顺序处理消息。第二个计算与第一个类似，但指定的最大并行度与可用处理器的数量相等。这使数据流块能够并行执行多个操作。

```
using System;
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to specify the maximum degree of parallelism
// when using dataflow.
class Program
{
   // Performs several computations by using dataflow and returns the elapsed
   // time required to perform the computations.
   static TimeSpan TimeDataflowComputations(int maxDegreeOfParallelism,
      int messageCount)
   {
      // Create an ActionBlock<int> that performs some work.
      var workerBlock = new ActionBlock<int>(
         // Simulate work by suspending the current thread.
         millisecondsTimeout => Thread.Sleep(millisecondsTimeout),
         // Specify a maximum degree of parallelism.
         new ExecutionDataflowBlockOptions
         {
            MaxDegreeOfParallelism = maxDegreeOfParallelism
         });

      // Compute the time that it takes for several messages to
      // flow through the dataflow block.

      Stopwatch stopwatch = new Stopwatch();
      stopwatch.Start();

      for (int i = 0; i < messageCount; i++)
      {
         workerBlock.Post(1000);
      }
```

```
      }
      workerBlock.Complete();

      // Wait for all messages to propagate through the network.
      workerBlock.Completion.Wait();

      // Stop the timer and return the elapsed number of milliseconds.
      stopwatch.Stop();
      return stopwatch.Elapsed;
   }
   static void Main(string[] args)
   {
      int processorCount = Environment.ProcessorCount;
      int messageCount = processorCount;

      // Print the number of processors on this computer.
      Console.WriteLine("Processor count = {0}.", processorCount);

      TimeSpan elapsed;

      // Perform two dataflow computations and print the elapsed
      // time required for each.

      // This call specifies a maximum degree of parallelism of 1.
      // This causes the dataflow block to process messages serially.
      elapsed = TimeDataflowComputations(1, messageCount);
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " +
         "elapsed time = {2}ms.", 1, messageCount, (int)elapsed.TotalMilliseconds);

      // Perform the computations again. This time, specify the number of
      // processors as the maximum degree of parallelism. This causes
      // multiple messages to be processed in parallel.
      elapsed = TimeDataflowComputations(processorCount, messageCount);
      Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " +
         "elapsed time = {2}ms.", processorCount, messageCount, (int)elapsed.TotalMilliseconds);
   }
}

/* Sample output:
Processor count = 4.
Degree of parallelism = 1; message count = 4; elapsed time = 4032ms.
Degree of parallelism = 4; message count = 4; elapsed time = 1001ms.
*/
```

```
Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks.Dataflow

' Demonstrates how to specify the maximum degree of parallelism
' when using dataflow.
Friend Class Program
    ' Performs several computations by using dataflow and returns the elapsed
    ' time required to perform the computations.
    Private Shared Function TimeDataflowComputations(ByVal maxDegreeOfParallelism As Integer, ByVal
messageCount As Integer) As TimeSpan
        ' Create an ActionBlock<int> that performs some work.
        Dim workerBlock = New ActionBlock(Of Integer)(Function(millisecondsTimeout)
Pause(millisecondsTimeout), New ExecutionDataflowBlockOptions() With {.MaxDegreeOfParallelism =
maxDegreeOfParallelism})
        ' Simulate work by suspending the current thread.
        ' Specify a maximum degree of parallelism.

        ' Compute the time that it takes for several messages to
        ' flow through the dataflow block.

        Dim stopwatch As New Stopwatch()
        stopwatch.Start()
```

```
        For i As Integer = 0 To messageCount - 1
            workerBlock.Post(1000)
        Next i
        workerBlock.Complete()

        ' Wait for all messages to propagate through the network.
        workerBlock.Completion.Wait()

        ' Stop the timer and return the elapsed number of milliseconds.
        stopwatch.Stop()
        Return stopwatch.Elapsed
    End Function

    Private Shared Function Pause(ByVal obj As Object)
        Thread.Sleep(obj)
        Return Nothing
    End Function
    Shared Sub Main(ByVal args() As String)
        Dim processorCount As Integer = Environment.ProcessorCount
        Dim messageCount As Integer = processorCount

        ' Print the number of processors on this computer.
        Console.WriteLine("Processor count = {0}.", processorCount)

        Dim elapsed As TimeSpan

        ' Perform two dataflow computations and print the elapsed
        ' time required for each.

        ' This call specifies a maximum degree of parallelism of 1.
        ' This causes the dataflow block to process messages serially.
        elapsed = TimeDataflowComputations(1, messageCount)
        Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " & "elapsed time = {2}ms.", 1,
messageCount, CInt(Fix(elapsed.TotalMilliseconds)))

        ' Perform the computations again. This time, specify the number of
        ' processors as the maximum degree of parallelism. This causes
        ' multiple messages to be processed in parallel.
        elapsed = TimeDataflowComputations(processorCount, messageCount)
        Console.WriteLine("Degree of parallelism = {0}; message count = {1}; " & "elapsed time = {2}ms.",
processorCount, messageCount, CInt(Fix(elapsed.TotalMilliseconds)))
    End Sub
End Class

' Sample output:
'Processor count = 4.
'Degree of parallelism = 1; message count = 4; elapsed time = 4032ms.
'Degree of parallelism = 4; message count = 4; elapsed time = 1001ms.
'
```

可靠编程

默认情况下，每个预定义的数据流块会按照接收消息的顺序将消息传播出去。虽然在指定的最大并行度大于 1 时会同时处理多条消息，但这些消息仍然按被接收的顺序进行传播。

由于 MaxDegreeOfParallelism 属性表示最大并行度，因此数据流块执行时的并行度可能小于指定的值。为了满足功能需求或需要考虑可用系统资源不足的问题时，数据流块可以使用较小的并行度。数据流块选择的并行度从不会大于您指定的值。

另请参阅

- 数据流

如何：在数据流块中指定任务计划程序

2021/11/16 •

本文档演示在应用程序中使用数据流时如何关联特定任务计划程序。示例在 Windows 窗体应用程序中使用 System.Threading.Tasks.ConcurrentExclusiveSchedulerPair 类来显示读取器任务处于活动状态的时间和编写器任务处于活动状态的时间。它还使用 TaskScheduler.FromCurrentSynchronizationContext 方法使数据流块能够在用户界面线程上运行。

> **NOTE**
>
> TPL 数据流库(System.Threading.Tasks.Dataflow 命名空间)不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

创建 Windows 窗体应用程序

1. 创建一个 Visual C# 或 Visual Basic“Windows 窗体应用程序”项目 。在以下步骤中，该项目命名为 `WriterReadersWinForms`。
2. 在主窗体的窗体设计器中，Form1.cs(对于 Visual Basic 则为 Form1.vb)添加了四个 CheckBox 控件。将 `checkBox1`、`checkBox2`、`checkBox3`、`checkBox4` 的 Text 属性分别设置为“读取器 1”、“读取器 2”、“读取器 3”和“编写器”。将每个控件的 Enabled 属性设置为 `False`。
3. 在窗体上添加一个 Timer 控件。将 Interval 属性设置为 `2500`。

添加数据流功能

本节介绍如何创建参与应用程序的数据流块以及如何将每个数据流块与任务计划程序关联。

在应用程序中添加数据流功能

1. 在项目中，添加对 System.Threading.Tasks.Dataflow.dll 的引用。
2. 确保 Form1.cs(对于 Visual Basic 则为 Form1.vb)包含以下 `using` 语句(Visual Basic 中为 `Imports`)。

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;
```

```
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow
```

3. 将 BroadcastBlock<T> 数据成员添加到 `Form1` 类。

```
// Broadcasts values to an ActionBlock<int> object that is associated
// with each check box.
BroadcastBlock<int> broadcaster = new BroadcastBlock<int>(null);
```

```
' Broadcasts values to an ActionBlock<int> object that is associated
' with each check box.
Private broadcaster As New BroadcastBlock(Of Integer)(Nothing)
```

4. 在 `Form1` 构造函数中，在调用 `InitializeComponent` 后，创建一个 ActionBlock<TInput> 对象，该对象将切换 CheckBox 对象的状态。

```
// Create an ActionBlock<CheckBox> object that toggles the state
// of CheckBox objects.
// Specifying the current synchronization context enables the
// action to run on the user-interface thread.
var toggleCheckBox = new ActionBlock<CheckBox>(checkBox =>
{
   checkBox.Checked = !checkBox.Checked;
},
new ExecutionDataflowBlockOptions
{
   TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
});
```

```
' Create an ActionBlock<CheckBox> object that toggles the state
' of CheckBox objects.
' Specifying the current synchronization context enables the
' action to run on the user-interface thread.
Dim toggleCheckBox = New ActionBlock(Of CheckBox)(Sub(checkBox) checkBox.Checked = Not
checkBox.Checked, New ExecutionDataflowBlockOptions With {.TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext()})
```

5. 在 `Form1` 构造函数中，创建一个 ConcurrentExclusiveSchedulerPair 对象和四个 ActionBlock<TInput> 对象，每个 ActionBlock<TInput> 对象对应一个 CheckBox 对象。对每个 ActionBlock<TInput> 对象，指定一个 ExecutionDataflowBlockOptions 对象，该对象将读取器的 TaskScheduler 属性设置为 ConcurrentScheduler 属性，将编写器的设置为 ExclusiveScheduler 属性。

```
// Create a ConcurrentExclusiveSchedulerPair object.
// Readers will run on the concurrent part of the scheduler pair.
// The writer will run on the exclusive part of the scheduler pair.
var taskSchedulerPair = new ConcurrentExclusiveSchedulerPair();

// Create an ActionBlock<int> object for each reader CheckBox object.
// Each ActionBlock<int> object represents an action that can read
// from a resource in parallel to other readers.
// Specifying the concurrent part of the scheduler pair enables the
// reader to run in parallel to other actions that are managed by
// that scheduler.
var readerActions =
   from checkBox in new CheckBox[] {checkBox1, checkBox2, checkBox3}
   select new ActionBlock<int>(milliseconds =>
   {
      // Toggle the check box to the checked state.
      toggleCheckBox.Post(checkBox);

      // Perform the read action. For demonstration, suspend the current
      // thread to simulate a lengthy read operation.
      Thread.Sleep(milliseconds);

      // Toggle the check box to the unchecked state.
      toggleCheckBox.Post(checkBox);
   },
   new ExecutionDataflowBlockOptions
   {
      TaskScheduler = taskSchedulerPair.ConcurrentScheduler
   });

// Create an ActionBlock<int> object for the writer CheckBox object.
// This ActionBlock<int> object represents an action that writes to
// a resource, but cannot run in parallel to readers.
// Specifying the exclusive part of the scheduler pair enables the
// writer to run in exclusively with respect to other actions that are
// managed by the scheduler pair.
var writerAction = new ActionBlock<int>(milliseconds =>
{
   // Toggle the check box to the checked state.
   toggleCheckBox.Post(checkBox4);

   // Perform the write action. For demonstration, suspend the current
   // thread to simulate a lengthy write operation.
   Thread.Sleep(milliseconds);

   // Toggle the check box to the unchecked state.
   toggleCheckBox.Post(checkBox4);
},
new ExecutionDataflowBlockOptions
{
   TaskScheduler = taskSchedulerPair.ExclusiveScheduler
});

// Link the broadcaster to each reader and writer block.
// The BroadcastBlock<T> class propagates values that it
// receives to all connected targets.
foreach (var readerAction in readerActions)
{
   broadcaster.LinkTo(readerAction);
}
broadcaster.LinkTo(writerAction);
```

```
' Create a ConcurrentExclusiveSchedulerPair object.
' Readers will run on the concurrent part of the scheduler pair.
' The writer will run on the exclusive part of the scheduler pair.
Dim taskSchedulerPair = New ConcurrentExclusiveSchedulerPair()

' Create an ActionBlock<int> object for each reader CheckBox object.
' Each ActionBlock<int> object represents an action that can read
' from a resource in parallel to other readers.
' Specifying the concurrent part of the scheduler pair enables the
' reader to run in parallel to other actions that are managed by
' that scheduler.
Dim readerActions = From checkBox In New CheckBox() {checkBox1, checkBox2, checkBox3} _
                    Select New ActionBlock(Of Integer)(Sub(milliseconds)
                                           ' Toggle the check box to the checked state.
                                           ' Perform the read action. For demonstration, suspend the
current
                                           ' thread to simulate a lengthy read operation.
                                           ' Toggle the check box to the unchecked state.
                                           toggleCheckBox.Post(checkBox)
                                                        Thread.Sleep(milliseconds)
                                                        toggleCheckBox.Post(checkBox)
                                                    End Sub, New ExecutionDataflowBlockOptions
With {.TaskScheduler = taskSchedulerPair.ConcurrentScheduler})

' Create an ActionBlock<int> object for the writer CheckBox object.
' This ActionBlock<int> object represents an action that writes to
' a resource, but cannot run in parallel to readers.
' Specifying the exclusive part of the scheduler pair enables the
' writer to run in exclusively with respect to other actions that are
' managed by the scheduler pair.
Dim writerAction = New ActionBlock(Of Integer)(Sub(milliseconds)
                                                   ' Toggle the check box to the checked state.
                                                   ' Perform the write action. For demonstration,
suspend the current
                                                   ' thread to simulate a lengthy write operation.
                                                   ' Toggle the check box to the unchecked state.
                                                   toggleCheckBox.Post(checkBox4)
                                                   Thread.Sleep(milliseconds)
                                                   toggleCheckBox.Post(checkBox4)
                                               End Sub, New ExecutionDataflowBlockOptions With
{.TaskScheduler = taskSchedulerPair.ExclusiveScheduler})

' Link the broadcaster to each reader and writer block.
' The BroadcastBlock<T> class propagates values that it
' receives to all connected targets.
For Each readerAction In readerActions
    broadcaster.LinkTo(readerAction)
Next readerAction
broadcaster.LinkTo(writerAction)
```

6. 在 `Form1` 构造函数中，启动 Timer 对象。

```
// Start the timer.
timer1.Start();
```

```
' Start the timer.
timer1.Start()
```

7. 在主窗体的窗体设计器中，为计时器的 Tick 事件创建事件处理程序。

8. 实现计时器的 Tick 事件。

```
// Event handler for the timer.
private void timer1_Tick(object sender, EventArgs e)
{
   // Post a value to the broadcaster. The broadcaster
   // sends this message to each target.
   broadcaster.Post(1000);
}
```

```
' Event handler for the timer.
Private Sub timer1_Tick(ByVal sender As Object, ByVal e As EventArgs) Handles timer1.Tick
    ' Post a value to the broadcaster. The broadcaster
    ' sends this message to each target.
    broadcaster.Post(1000)
End Sub
```

因为 `toggleCheckBox` 数据流块是在用户界面上操作，所以此操作要在用户界面线程上执行，这一点很重要。为此，在构造期间，此对象提供一个将 ExecutionDataflowBlockOptions 属性设置为 TaskScheduler 的 TaskScheduler.FromCurrentSynchronizationContext 对象。FromCurrentSynchronizationContext 方法会创建一个在当前同步上下文中执行工作的 TaskScheduler 对象。因为 `Form1` 构造函数是从用户界面线程中调用的，所以 `toggleCheckBox` 数据流块的操作也会在用户线程中运行。

对于在同一 ConcurrentExclusiveSchedulerPair 对象上运行的所有其他数据流块，此示例还使用了 ConcurrentExclusiveSchedulerPair 类使某些数据流块能够并发操作，而使其他数据流块能够单独执行。当多个数据流块共享资源，但有些需要对该资源独占访问时，这种技术很有用，因为它不需要手动同步对该资源的访问。手动同步的消除会使代码更高效。

示例

下面的示例演示 Form1.cs（对于 Visual Basic 则为 Form1.vb）的完整代码。

```
using System;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;
using System.Windows.Forms;

namespace WriterReadersWinForms
{
   public partial class Form1 : Form
   {
      // Broadcasts values to an ActionBlock<int> object that is associated
      // with each check box.
      BroadcastBlock<int> broadcaster = new BroadcastBlock<int>(null);

      public Form1()
      {
         InitializeComponent();

         // Create an ActionBlock<CheckBox> object that toggles the state
         // of CheckBox objects.
         // Specifying the current synchronization context enables the
         // action to run on the user-interface thread.
         var toggleCheckBox = new ActionBlock<CheckBox>(checkBox =>
         {
            checkBox.Checked = !checkBox.Checked;
         },
         new ExecutionDataflowBlockOptions
         {
            TaskScheduler = TaskScheduler.FromCurrentSynchronizationContext()
         });
```

```
   });

   // Create a ConcurrentExclusiveSchedulerPair object.
   // Readers will run on the concurrent part of the scheduler pair.
   // The writer will run on the exclusive part of the scheduler pair.
   var taskSchedulerPair = new ConcurrentExclusiveSchedulerPair();

   // Create an ActionBlock<int> object for each reader CheckBox object.
   // Each ActionBlock<int> object represents an action that can read
   // from a resource in parallel to other readers.
   // Specifying the concurrent part of the scheduler pair enables the
   // reader to run in parallel to other actions that are managed by
   // that scheduler.
   var readerActions =
      from checkBox in new CheckBox[] {checkBox1, checkBox2, checkBox3}
      select new ActionBlock<int>(milliseconds =>
      {
         // Toggle the check box to the checked state.
         toggleCheckBox.Post(checkBox);

         // Perform the read action. For demonstration, suspend the current
         // thread to simulate a lengthy read operation.
         Thread.Sleep(milliseconds);

         // Toggle the check box to the unchecked state.
         toggleCheckBox.Post(checkBox);
      },
      new ExecutionDataflowBlockOptions
      {
         TaskScheduler = taskSchedulerPair.ConcurrentScheduler
      });

   // Create an ActionBlock<int> object for the writer CheckBox object.
   // This ActionBlock<int> object represents an action that writes to
   // a resource, but cannot run in parallel to readers.
   // Specifying the exclusive part of the scheduler pair enables the
   // writer to run in exclusively with respect to other actions that are
   // managed by the scheduler pair.
   var writerAction = new ActionBlock<int>(milliseconds =>
   {
      // Toggle the check box to the checked state.
      toggleCheckBox.Post(checkBox4);

      // Perform the write action. For demonstration, suspend the current
      // thread to simulate a lengthy write operation.
      Thread.Sleep(milliseconds);

      // Toggle the check box to the unchecked state.
      toggleCheckBox.Post(checkBox4);
   },
   new ExecutionDataflowBlockOptions
   {
      TaskScheduler = taskSchedulerPair.ExclusiveScheduler
   });

   // Link the broadcaster to each reader and writer block.
   // The BroadcastBlock<T> class propagates values that it
   // receives to all connected targets.
   foreach (var readerAction in readerActions)
   {
      broadcaster.LinkTo(readerAction);
   }
   broadcaster.LinkTo(writerAction);

   // Start the timer.
   timer1.Start();
}

// Event handler for the timer.
private void timer1_Tick(object sender, EventArgs e)
```

```
      private void timer1_Tick(object sender, EventArgs e)
      {
         // Post a value to the broadcaster. The broadcaster
         // sends this message to each target.
         broadcaster.Post(1000);
      }
   }
}
```

```
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Threading.Tasks.Dataflow


Namespace WriterReadersWinForms
    Partial Public Class Form1
        Inherits Form
        ' Broadcasts values to an ActionBlock<int> object that is associated
        ' with each check box.
        Private broadcaster As New BroadcastBlock(Of Integer)(Nothing)

        Public Sub New()
            InitializeComponent()

            ' Create an ActionBlock<CheckBox> object that toggles the state
            ' of CheckBox objects.
            ' Specifying the current synchronization context enables the
            ' action to run on the user-interface thread.
            Dim toggleCheckBox = New ActionBlock(Of CheckBox)(Sub(checkBox) checkBox.Checked = Not
checkBox.Checked, New ExecutionDataflowBlockOptions With {.TaskScheduler =
TaskScheduler.FromCurrentSynchronizationContext()})

            ' Create a ConcurrentExclusiveSchedulerPair object.
            ' Readers will run on the concurrent part of the scheduler pair.
            ' The writer will run on the exclusive part of the scheduler pair.
            Dim taskSchedulerPair = New ConcurrentExclusiveSchedulerPair()

            ' Create an ActionBlock<int> object for each reader CheckBox object.
            ' Each ActionBlock<int> object represents an action that can read
            ' from a resource in parallel to other readers.
            ' Specifying the concurrent part of the scheduler pair enables the
            ' reader to run in parallel to other actions that are managed by
            ' that scheduler.
            Dim readerActions = From checkBox In New CheckBox() {checkBox1, checkBox2, checkBox3} _
                                Select New ActionBlock(Of Integer)(Sub(milliseconds)
                                                       ' Toggle the check box to the checked state.
                                                       ' Perform the read action. For demonstration, suspend
the current
                                                       ' thread to simulate a lengthy read operation.
                                                       ' Toggle the check box to the unchecked state.
                                                       toggleCheckBox.Post(checkBox)
                                                                   Thread.Sleep(milliseconds)
                                                                   toggleCheckBox.Post(checkBox)
                                                                End Sub, New
ExecutionDataflowBlockOptions With {.TaskScheduler = taskSchedulerPair.ConcurrentScheduler})

            ' Create an ActionBlock<int> object for the writer CheckBox object.
            ' This ActionBlock<int> object represents an action that writes to
            ' a resource, but cannot run in parallel to readers.
            ' Specifying the exclusive part of the scheduler pair enables the
            ' writer to run in exclusively with respect to other actions that are
            ' managed by the scheduler pair.
            Dim writerAction = New ActionBlock(Of Integer)(Sub(milliseconds)
                                                               ' Toggle the check box to the checked state.
                                                               ' Perform the write action. For
demonstration, suspend the current
                                                               ' thread to simulate a lengthy write
operation
```

```
operation.
                                                                    ' Toggle the check box to the unchecked
state.
                                                                    toggleCheckBox.Post(checkBox4)
                                                                    Thread.Sleep(milliseconds)
                                                                    toggleCheckBox.Post(checkBox4)
                                                                End Sub, New ExecutionDataflowBlockOptions With
{.TaskScheduler = taskSchedulerPair.ExclusiveScheduler})

            ' Link the broadcaster to each reader and writer block.
            ' The BroadcastBlock<T> class propagates values that it
            ' receives to all connected targets.
            For Each readerAction In readerActions
                broadcaster.LinkTo(readerAction)
            Next readerAction
            broadcaster.LinkTo(writerAction)

            ' Start the timer.
            timer1.Start()
        End Sub

        ' Event handler for the timer.
        Private Sub timer1_Tick(ByVal sender As Object, ByVal e As EventArgs) Handles timer1.Tick
            ' Post a value to the broadcaster. The broadcaster
            ' sends this message to each target.
            broadcaster.Post(1000)
        End Sub
    End Class
End Namespace
```

另请参阅

- 数据流

演练：使用 BatchBlock 和 BatchedJoinBlock 提高效率

2021/11/16 •

TPL 数据流库提供 System.Threading.Tasks.Dataflow.BatchBlock<T> 和 System.Threading.Tasks.Dataflow.BatchedJoinBlock<T1,T2> 类，以便可以接收和缓冲一个或多个源的数据，再将缓冲的数据作为一个集合传播出去。如果从一个或多个源收集数据，再批处理多个数据元素，就会发现这种批处理机制非常有用。例如，假设应用使用数据流将记录插入数据库。如果同时插入多项，而不是顺序一次插入一个，此操作可能会更高效。本文档介绍了如何使用 BatchBlock<T> 类，提高此类数据库插入操作的效率。它还介绍了如何使用 BatchedJoinBlock<T1,T2> 类，捕获程序从数据库读取数据时的结果和发生的任何异常。

> **NOTE**
> TPL 数据流库（System.Threading.Tasks.Dataflow 命名空间）不随 .NET 一起分发。若要在 Visual Studio 中安装 System.Threading.Tasks.Dataflow 命名空间，请打开项目，选择“项目”菜单中的“管理 NuGet 包”，再在线搜索 `System.Threading.Tasks.Dataflow` 包。或者，若要使用 .NET Core CLI 进行安装，请运行 `dotnet add package System.Threading.Tasks.Dataflow`。

系统必备

1. 开始本演练前，请先阅读数据流文档中的“联接块”部分。
2. 确保计算机上有 Northwind 数据库的副本 Northwind.sdf。此文件通常位于 %Program Files%\Microsoft SQL Server Compact Edition\v3.5\Samples\ 文件夹中。

> **IMPORTANT**
> 在一些版本的 Windows 中，如果以非管理员模式运行 Visual Studio，便无法连接到 Northwind.sdf。若要连接到 Northwind.sdf，请在“以管理员身份运行”模式下启动 Visual Studio 或 Visual Studio 开发人员命令提示 。

本演练包含以下各节：

- 创建控制台应用
- 定义 Employee 类
- 定义员工数据库操作
- 不使用缓冲将员工数据添加到数据库
- 使用缓冲将员工数据添加到数据库
- 使用已缓冲联接读取数据库中的员工数据
- 完整示例

创建控制台应用

1. 在 Visual Studio 中，创建 Visual C# 或 Visual Basic“控制台应用程序”项目。在本文档中，该项目名为 `DataflowBatchDatabase`。

2. 在项目中，添加对 System.Data.SqlServerCe.dll 和 System.Threading.Tasks.Dataflow.dll 的引用。

3. 确保 Form1.cs（对于 Visual Basic，则为 Form1.vb）包含以下 `using`（Visual Basic 中为 `Imports`）语句。

```
using System;
using System.Collections.Generic;
using System.Data.SqlServerCe;
using System.Diagnostics;
using System.IO;
using System.Threading.Tasks.Dataflow;
```

```
Imports System.Collections.Generic
Imports System.Data.SqlServerCe
Imports System.Diagnostics
Imports System.IO
Imports System.Threading.Tasks.Dataflow
```

4. 将以下数据成员添加到 `Program` 类。

```
// The number of employees to add to the database.
// TODO: Change this value to experiment with different numbers of
// employees to insert into the database.
static readonly int insertCount = 256;

// The size of a single batch of employees to add to the database.
// TODO: Change this value to experiment with different batch sizes.
static readonly int insertBatchSize = 96;

// The source database file.
// TODO: Change this value if Northwind.sdf is at a different location
// on your computer.
static readonly string sourceDatabase =
   @"C:\Program Files\Microsoft SQL Server Compact Edition\v3.5\Samples\Northwind.sdf";

// TODO: Change this value if you require a different temporary location.
static readonly string scratchDatabase =
   @"C:\Temp\Northwind.sdf";
```

```
' The number of employees to add to the database.
' TODO: Change this value to experiment with different numbers of
' employees to insert into the database.
Private Shared ReadOnly insertCount As Integer = 256

' The size of a single batch of employees to add to the database.
' TODO: Change this value to experiment with different batch sizes.
Private Shared ReadOnly insertBatchSize As Integer = 96

' The source database file.
' TODO: Change this value if Northwind.sdf is at a different location
' on your computer.
Private Shared ReadOnly sourceDatabase As String = "C:\Program Files\Microsoft SQL Server Compact
Edition\v3.5\Samples\Northwind.sdf"

' TODO: Change this value if you require a different temporary location.
Private Shared ReadOnly scratchDatabase As String = "C:\Temp\Northwind.sdf"
```

定义 Employee 类

向 `Program` 类添加 `Employee` 类。

```
// Describes an employee. Each property maps to a
// column in the Employees table in the Northwind database.
// For brevity, the Employee class does not contain
// all columns from the Employees table.
class Employee
{
   public int EmployeeID { get; set; }
   public string LastName { get; set; }
   public string FirstName { get; set; }

   // A random number generator that helps tp generate
   // Employee property values.
   static Random rand = new Random(42);

   // Possible random first names.
   static readonly string[] firstNames = { "Tom", "Mike", "Ruth", "Bob", "John" };
   // Possible random last names.
   static readonly string[] lastNames = { "Jones", "Smith", "Johnson", "Walker" };

   // Creates an Employee object that contains random
   // property values.
   public static Employee Random()
   {
      return new Employee
      {
         EmployeeID = -1,
         LastName = lastNames[rand.Next() % lastNames.Length],
         FirstName = firstNames[rand.Next() % firstNames.Length]
      };
   }
}
```

```
' Describes an employee. Each property maps to a
' column in the Employees table in the Northwind database.
' For brevity, the Employee class does not contain
' all columns from the Employees table.
Private Class Employee
    Public Property EmployeeID() As Integer
    Public Property LastName() As String
    Public Property FirstName() As String

    ' A random number generator that helps tp generate
    ' Employee property values.
    Private Shared rand As New Random(42)

    ' Possible random first names.
    Private Shared ReadOnly firstNames() As String = {"Tom", "Mike", "Ruth", "Bob", "John"}
    ' Possible random last names.
    Private Shared ReadOnly lastNames() As String = {"Jones", "Smith", "Johnson", "Walker"}

    ' Creates an Employee object that contains random
    ' property values.
    Public Shared Function Random() As Employee
        Return New Employee With {.EmployeeID = -1, .LastName = lastNames(rand.Next() Mod lastNames.Length),
.FirstName = firstNames(rand.Next() Mod firstNames.Length)}
    End Function
End Class
```

`Employee` 类包含下面三个属性：`EmployeeID`、`LastName` 和 `FirstName`。这些属性对应于 Northwind 数据库中 `Employees` 表的 `Employee ID`、`Last Name` 和 `First Name` 列。在展示的此示例中，`Employee` 类还定义了 `Random` 方法，用于创建属性值为随机值的 `Employee` 对象。

定义员工数据库操作

向 `Program` 添加 `InsertEmployees`、`GetEmployeeCount` 和 `GetEmployeeID` 方法。

```
// Adds new employee records to the database.
static void InsertEmployees(Employee[] employees, string connectionString)
{
   using (SqlCeConnection connection =
      new SqlCeConnection(connectionString))
   {
      try
      {
         // Create the SQL command.
         SqlCeCommand command = new SqlCeCommand(
            "INSERT INTO Employees ([Last Name], [First Name])" +
            "VALUES (@lastName, @firstName)",
            connection);

         connection.Open();
         for (int i = 0; i < employees.Length; i++)
         {
            // Set parameters.
            command.Parameters.Clear();
            command.Parameters.Add("@lastName", employees[i].LastName);
            command.Parameters.Add("@firstName", employees[i].FirstName);

            // Execute the command.
            command.ExecuteNonQuery();
         }
      }
      finally
      {
         connection.Close();
      }
   }
}

// Retrieves the number of entries in the Employees table in
// the Northwind database.
static int GetEmployeeCount(string connectionString)
{
   int result = 0;
   using (SqlCeConnection sqlConnection =
      new SqlCeConnection(connectionString))
   {
      SqlCeCommand sqlCommand = new SqlCeCommand(
         "SELECT COUNT(*) FROM Employees", sqlConnection);

      sqlConnection.Open();
      try
      {
         result = (int)sqlCommand.ExecuteScalar();
      }
      finally
      {
         sqlConnection.Close();
      }
   }
   return result;
}

// Retrieves the ID of the first employee that has the provided name.
static int GetEmployeeID(string lastName, string firstName,
   string connectionString)
{
   using (SqlCeConnection connection =
      new SqlCeConnection(connectionString))
```

```
    {
        SqlCeCommand command = new SqlCeCommand(
            string.Format(
                "SELECT [Employee ID] FROM Employees " +
                "WHERE [Last Name] = '{0}' AND [First Name] = '{1}'",
                lastName, firstName),
            connection);

        connection.Open();
        try
        {
            return (int)command.ExecuteScalar();
        }
        finally
        {
            connection.Close();
        }
    }
}
```

```
' Adds new employee records to the database.
Private Shared Sub InsertEmployees(ByVal employees() As Employee, ByVal connectionString As String)
    Using connection As New SqlCeConnection(connectionString)
        Try
            ' Create the SQL command.
            Dim command As New SqlCeCommand("INSERT INTO Employees ([Last Name], [First Name])" & "VALUES
(@lastName, @firstName)", connection)

            connection.Open()
            For i As Integer = 0 To employees.Length - 1
                ' Set parameters.
                command.Parameters.Clear()
                command.Parameters.Add("@lastName", employees(i).LastName)
                command.Parameters.Add("@firstName", employees(i).FirstName)

                ' Execute the command.
                command.ExecuteNonQuery()
            Next i
        Finally
            connection.Close()
        End Try
    End Using
End Sub

' Retrieves the number of entries in the Employees table in
' the Northwind database.
Private Shared Function GetEmployeeCount(ByVal connectionString As String) As Integer
    Dim result As Integer = 0
    Using sqlConnection As New SqlCeConnection(connectionString)
        Dim sqlCommand As New SqlCeCommand("SELECT COUNT(*) FROM Employees", sqlConnection)

        sqlConnection.Open()
        Try
            result = CInt(Fix(sqlCommand.ExecuteScalar()))
        Finally
            sqlConnection.Close()
        End Try
    End Using
    Return result
End Function

' Retrieves the ID of the first employee that has the provided name.
Private Shared Function GetEmployeeID(ByVal lastName As String, ByVal firstName As String, ByVal
connectionString As String) As Integer
    Using connection As New SqlCeConnection(connectionString)
        Dim command As New SqlCeCommand(String.Format("SELECT [Employee ID] FROM Employees " & "WHERE [Last
Name] = '{0}' AND [First Name] = '{1}'", lastName, firstName), connection)

        connection.Open()
        Try
            Return CInt(Fix(command.ExecuteScalar()))
        Finally
            connection.Close()
        End Try
    End Using
End Function
```

`InsertEmployees` 方法将新员工记录添加到数据库。`GetEmployeeCount` 方法检索 `Employees` 表中的条目数。`GetEmployeeID` 方法检索与所提供姓名匹配的首位员工的标识符。这三个方法全都需要对 Northwind 数据库使用连接字符串，并使用 `System.Data.SqlServerCe` 命名空间中的功能与数据库进行通信。

不使用缓冲将员工数据添加到数据库

向 `Program` 类添加 `AddEmployees` 和 `PostRandomEmployees` 方法。

```
// Posts random Employee data to the provided target block.
static void PostRandomEmployees(ITargetBlock<Employee> target, int count)
{
   Console.WriteLine("Adding {0} entries to Employee table...", count);

   for (int i = 0; i < count; i++)
   {
      target.Post(Employee.Random());
   }
}

// Adds random employee data to the database by using dataflow.
static void AddEmployees(string connectionString, int count)
{
   // Create an ActionBlock<Employee> object that adds a single
   // employee entry to the database.
   var insertEmployee = new ActionBlock<Employee>(e =>
      InsertEmployees(new Employee[] { e }, connectionString));

   // Post several random Employee objects to the dataflow block.
   PostRandomEmployees(insertEmployee, count);

   // Set the dataflow block to the completed state and wait for
   // all insert operations to complete.
   insertEmployee.Complete();
   insertEmployee.Completion.Wait();
}
```

```
' Posts random Employee data to the provided target block.
Private Shared Sub PostRandomEmployees(ByVal target As ITargetBlock(Of Employee), ByVal count As Integer)
    Console.WriteLine("Adding {0} entries to Employee table...", count)

    For i As Integer = 0 To count - 1
        target.Post(Employee.Random())
    Next i
End Sub

' Adds random employee data to the database by using dataflow.
Private Shared Sub AddEmployees(ByVal connectionString As String, ByVal count As Integer)
    ' Create an ActionBlock<Employee> object that adds a single
    ' employee entry to the database.
    Dim insertEmployee = New ActionBlock(Of Employee)(Sub(e) InsertEmployees(New Employee() {e},
connectionString))

    ' Post several random Employee objects to the dataflow block.
    PostRandomEmployees(insertEmployee, count)

    ' Set the dataflow block to the completed state and wait for
    ' all insert operations to complete.
    insertEmployee.Complete()
    insertEmployee.Completion.Wait()
End Sub
```

`AddEmployees` 方法使用数据流将随机员工数据添加到数据库。此方法创建 ActionBlock<TInput> 对象，以调用 `InsertEmployees` 方法将员工条目添加到数据库。然后，`AddEmployees` 方法调用 `PostRandomEmployees` 方法，将多个 `Employee` 对象发布到 ActionBlock<TInput> 对象。然后，`AddEmployees` 方法等待所有插入操作完成。

使用缓冲将员工数据添加到数据库

向 `Program` 类添加 `AddEmployeesBatched` 方法。

```
// Adds random employee data to the database by using dataflow.
// This method is similar to AddEmployees except that it uses batching
// to add multiple employees to the database at a time.
static void AddEmployeesBatched(string connectionString, int batchSize,
   int count)
{
   // Create a BatchBlock<Employee> that holds several Employee objects and
   // then propagates them out as an array.
   var batchEmployees = new BatchBlock<Employee>(batchSize);

   // Create an ActionBlock<Employee[]> object that adds multiple
   // employee entries to the database.
   var insertEmployees = new ActionBlock<Employee[]>(a =>
      InsertEmployees(a, connectionString));

   // Link the batch block to the action block.
   batchEmployees.LinkTo(insertEmployees);

   // When the batch block completes, set the action block also to complete.
   batchEmployees.Completion.ContinueWith(delegate { insertEmployees.Complete(); });

   // Post several random Employee objects to the batch block.
   PostRandomEmployees(batchEmployees, count);

   // Set the batch block to the completed state and wait for
   // all insert operations to complete.
   batchEmployees.Complete();
   insertEmployees.Completion.Wait();
}
```

```
' Adds random employee data to the database by using dataflow.
' This method is similar to AddEmployees except that it uses batching
' to add multiple employees to the database at a time.
Private Shared Sub AddEmployeesBatched(ByVal connectionString As String, ByVal batchSize As Integer, ByVal
count As Integer)
    ' Create a BatchBlock<Employee> that holds several Employee objects and
    ' then propagates them out as an array.
    Dim batchEmployees = New BatchBlock(Of Employee)(batchSize)

    ' Create an ActionBlock<Employee[]> object that adds multiple
    ' employee entries to the database.
    Dim insertEmployees = New ActionBlock(Of Employee())(Sub(a) Program.InsertEmployees(a,
connectionString))

    ' Link the batch block to the action block.
    batchEmployees.LinkTo(insertEmployees)

    ' When the batch block completes, set the action block also to complete.
    batchEmployees.Completion.ContinueWith(Sub() insertEmployees.Complete())

    ' Post several random Employee objects to the batch block.
    PostRandomEmployees(batchEmployees, count)

    ' Set the batch block to the completed state and wait for
    ' all insert operations to complete.
    batchEmployees.Complete()
    insertEmployees.Completion.Wait()
End Sub
```

此方法类似于 `AddEmployees`，不同之处在于它还先使用 BatchBlock<T> 类缓冲多个 `Employee` 对象，然后再将这些对象发送给 ActionBlock<TInput> 对象。由于 BatchBlock<T> 类将多个元素以集合形式传播出去，因此 ActionBlock<TInput> 对象被修改为对 `Employee` 对象的数组执行操作。与 `AddEmployees` 方法类似，`AddEmployeesBatched` 调用 `PostRandomEmployees` 方法发布多个 `Employee` 对象；不同之处在于，

`AddEmployeesBatched` 将这些对象发布到 BatchBlock<T> 对象。`AddEmployeesBatched` 方法还等待所有插入操作完成。

使用已缓冲联接读取数据库中的员工数据

向 `Program` 类添加 `GetRandomEmployees` 方法。

```
// Displays information about several random employees to the console.
static void GetRandomEmployees(string connectionString, int batchSize,
   int count)
{
   // Create a BatchedJoinBlock<Employee, Exception> object that holds
   // both employee and exception data.
   var selectEmployees = new BatchedJoinBlock<Employee, Exception>(batchSize);

   // Holds the total number of exceptions that occurred.
   int totalErrors = 0;

   // Create an action block that prints employee and error information
   // to the console.
   var printEmployees =
      new ActionBlock<Tuple<IList<Employee>, IList<Exception>>>(data =>
      {
         // Print information about the employees in this batch.
         Console.WriteLine("Received a batch...");
         foreach (Employee e in data.Item1)
         {
            Console.WriteLine("Last={0} First={1} ID={2}",
               e.LastName, e.FirstName, e.EmployeeID);
         }

         // Print the error count for this batch.
         Console.WriteLine("There were {0} errors in this batch...",
            data.Item2.Count);

         // Update total error count.
         totalErrors += data.Item2.Count;
      });

   // Link the batched join block to the action block.
   selectEmployees.LinkTo(printEmployees);

   // When the batched join block completes, set the action block also to complete.
   selectEmployees.Completion.ContinueWith(delegate { printEmployees.Complete(); });

   // Try to retrieve the ID for several random employees.
   Console.WriteLine("Selecting random entries from Employees table...");
   for (int i = 0; i < count; i++)
   {
      try
      {
         // Create a random employee.
         Employee e = Employee.Random();

         // Try to retrieve the ID for the employee from the database.
         e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString);

         // Post the Employee object to the Employee target of
         // the batched join block.
         selectEmployees.Target1.Post(e);
      }
      catch (NullReferenceException e)
      {
         // GetEmployeeID throws NullReferenceException when there is
         // no such employee with the given name. When this happens,
         // post the Exception object to the Exception target of
```

```
         // post the Exception object to the Exception target of
         // the batched join block.
         selectEmployees.Target2.Post(e);
      }
   }

   // Set the batched join block to the completed state and wait for
   // all retrieval operations to complete.
   selectEmployees.Complete();
   printEmployees.Completion.Wait();

   // Print the total error count.
   Console.WriteLine("Finished. There were {0} total errors.", totalErrors);
}
```

```
' Displays information about several random employees to the console.
Private Shared Sub GetRandomEmployees(ByVal connectionString As String, ByVal batchSize As Integer, ByVal
count As Integer)
    ' Create a BatchedJoinBlock<Employee, Exception> object that holds
    ' both employee and exception data.
    Dim selectEmployees = New BatchedJoinBlock(Of Employee, Exception)(batchSize)

    ' Holds the total number of exceptions that occurred.
    Dim totalErrors As Integer = 0

    ' Create an action block that prints employee and error information
    ' to the console.
    Dim printEmployees = New ActionBlock(Of Tuple(Of IList(Of Employee), IList(Of Exception)))(Sub(data)
                                                                                                  ' Print
information about the employees in this batch.
                                                                                                  ' Print
the error count for this batch.
                                                                                                  ' Update
total error count.

Console.WriteLine("Received a batch...")
                                                                                                  For Each
e As Employee In data.Item1

Console.WriteLine("Last={0} First={1} ID={2}", e.LastName, e.FirstName, e.EmployeeID)
                                                                                                  Next e

Console.WriteLine("There were {0} errors in this batch...", data.Item2.Count)

totalErrors += data.Item2.Count
                                                                                              End Sub)

    ' Link the batched join block to the action block.
    selectEmployees.LinkTo(printEmployees)

    ' When the batched join block completes, set the action block also to complete.
    selectEmployees.Completion.ContinueWith(Sub() printEmployees.Complete())

    ' Try to retrieve the ID for several random employees.
    Console.WriteLine("Selecting random entries from Employees table...")
    For i As Integer = 0 To count - 1
        Try
            ' Create a random employee.
            Dim e As Employee = Employee.Random()

            ' Try to retrieve the ID for the employee from the database.
            e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString)

            ' Post the Employee object to the Employee target of
            ' the batched join block.
            selectEmployees.Target1.Post(e)
        Catch e As NullReferenceException
            ' GetEmployeeID throws NullReferenceException when there is
```

```
            ' no such employee with the given name. When this happens,
            ' post the Exception object to the Exception target of
            ' the batched join block.
            selectEmployees.Target2.Post(e)
        End Try
    Next i

    ' Set the batched join block to the completed state and wait for
    ' all retrieval operations to complete.
    selectEmployees.Complete()
    printEmployees.Completion.Wait()

    ' Print the total error count.
    Console.WriteLine("Finished. There were {0} total errors.", totalErrors)
End Sub
```

此方法将随机员工信息打印到控制台中。它会创建多个随机 `Employee` 对象，并调用 `GetEmployeeID` 方法检索每个对象的唯一标识符。由于 `GetEmployeeID` 方法在找不到与给定姓氏和名字匹配的员工时抛出异常，因此 `GetRandomEmployees` 方法使用 BatchedJoinBlock<T1,T2> 类存储 `GetEmployeeID` 成功调用对应的 `Employee` 对象，以及失败调用对应的 System.Exception 对象。此示例中的 ActionBlock<TInput> 对象对保留 `Employee` 对象列表和 Exception 对象列表的 Tuple<T1,T2> 对象执行操作。如果收到的 `Employee` 和 Exception 对象数总和等于批大小，BatchedJoinBlock<T1,T2> 对象就会传播出此类数据。

完整示例

以下示例显示了完整的代码。`Main` 方法比较执行批量数据库插入和非批量数据库插入所需的时间。它还展示了如何使用已缓冲联接，读取数据库中的员工数据并报告错误。

```
using System;
using System.Collections.Generic;
using System.Data.SqlServerCe;
using System.Diagnostics;
using System.IO;
using System.Threading.Tasks.Dataflow;

// Demonstrates how to use batched dataflow blocks to improve
// the performance of database operations.
namespace DataflowBatchDatabase
{
   class Program
   {
      // The number of employees to add to the database.
      // TODO: Change this value to experiment with different numbers of
      // employees to insert into the database.
      static readonly int insertCount = 256;

      // The size of a single batch of employees to add to the database.
      // TODO: Change this value to experiment with different batch sizes.
      static readonly int insertBatchSize = 96;

      // The source database file.
      // TODO: Change this value if Northwind.sdf is at a different location
      // on your computer.
      static readonly string sourceDatabase =
         @"C:\Program Files\Microsoft SQL Server Compact Edition\v3.5\Samples\Northwind.sdf";

      // TODO: Change this value if you require a different temporary location.
      static readonly string scratchDatabase =
         @"C:\Temp\Northwind.sdf";

      // Describes an employee. Each property maps to a
      // column in the Employees table in the Northwind database.
      // For brevity, the Employee class does not contain
```

```
// all columns from the Employees table.
class Employee
{
   public int EmployeeID { get; set; }
   public string LastName { get; set; }
   public string FirstName { get; set; }

   // A random number generator that helps tp generate
   // Employee property values.
   static Random rand = new Random(42);

   // Possible random first names.
   static readonly string[] firstNames = { "Tom", "Mike", "Ruth", "Bob", "John" };
   // Possible random last names.
   static readonly string[] lastNames = { "Jones", "Smith", "Johnson", "Walker" };

   // Creates an Employee object that contains random
   // property values.
   public static Employee Random()
   {
      return new Employee
      {
         EmployeeID = -1,
         LastName = lastNames[rand.Next() % lastNames.Length],
         FirstName = firstNames[rand.Next() % firstNames.Length]
      };
   }
}

// Adds new employee records to the database.
static void InsertEmployees(Employee[] employees, string connectionString)
{
   using (SqlCeConnection connection =
      new SqlCeConnection(connectionString))
   {
      try
      {
         // Create the SQL command.
         SqlCeCommand command = new SqlCeCommand(
            "INSERT INTO Employees ([Last Name], [First Name])" +
            "VALUES (@lastName, @firstName)",
            connection);

         connection.Open();
         for (int i = 0; i < employees.Length; i++)
         {
            // Set parameters.
            command.Parameters.Clear();
            command.Parameters.Add("@lastName", employees[i].LastName);
            command.Parameters.Add("@firstName", employees[i].FirstName);

            // Execute the command.
            command.ExecuteNonQuery();
         }
      }
      finally
      {
         connection.Close();
      }
   }
}

// Retrieves the number of entries in the Employees table in
// the Northwind database.
static int GetEmployeeCount(string connectionString)
{
   int result = 0;
   using (SqlCeConnection sqlConnection =
      new SqlCeConnection(connectionString))
```

```
            new SqlCeConnection(connectionString))
        {
            SqlCeCommand sqlCommand = new SqlCeCommand(
                "SELECT COUNT(*) FROM Employees", sqlConnection);

            sqlConnection.Open();
            try
            {
                result = (int)sqlCommand.ExecuteScalar();
            }
            finally
            {
                sqlConnection.Close();
            }
        }
        return result;
    }

    // Retrieves the ID of the first employee that has the provided name.
    static int GetEmployeeID(string lastName, string firstName,
        string connectionString)
    {
        using (SqlCeConnection connection =
            new SqlCeConnection(connectionString))
        {
            SqlCeCommand command = new SqlCeCommand(
                string.Format(
                    "SELECT [Employee ID] FROM Employees " +
                    "WHERE [Last Name] = '{0}' AND [First Name] = '{1}'",
                    lastName, firstName),
                connection);

            connection.Open();
            try
            {
                return (int)command.ExecuteScalar();
            }
            finally
            {
                connection.Close();
            }
        }
    }

    // Posts random Employee data to the provided target block.
    static void PostRandomEmployees(ITargetBlock<Employee> target, int count)
    {
        Console.WriteLine("Adding {0} entries to Employee table...", count);

        for (int i = 0; i < count; i++)
        {
            target.Post(Employee.Random());
        }
    }

    // Adds random employee data to the database by using dataflow.
    static void AddEmployees(string connectionString, int count)
    {
        // Create an ActionBlock<Employee> object that adds a single
        // employee entry to the database.
        var insertEmployee = new ActionBlock<Employee>(e =>
            InsertEmployees(new Employee[] { e }, connectionString));

        // Post several random Employee objects to the dataflow block.
        PostRandomEmployees(insertEmployee, count);

        // Set the dataflow block to the completed state and wait for
        // all insert operations to complete.
        insertEmployee.Complete();
        insertEmployee.Completion.Wait();
```

```
   insertEmployee.Completion.Wait();
}

// Adds random employee data to the database by using dataflow.
// This method is similar to AddEmployees except that it uses batching
// to add multiple employees to the database at a time.
static void AddEmployeesBatched(string connectionString, int batchSize,
   int count)
{
   // Create a BatchBlock<Employee> that holds several Employee objects and
   // then propagates them out as an array.
   var batchEmployees = new BatchBlock<Employee>(batchSize);

   // Create an ActionBlock<Employee[]> object that adds multiple
   // employee entries to the database.
   var insertEmployees = new ActionBlock<Employee[]>(a =>
      InsertEmployees(a, connectionString));

   // Link the batch block to the action block.
   batchEmployees.LinkTo(insertEmployees);

   // When the batch block completes, set the action block also to complete.
   batchEmployees.Completion.ContinueWith(delegate { insertEmployees.Complete(); });

   // Post several random Employee objects to the batch block.
   PostRandomEmployees(batchEmployees, count);

   // Set the batch block to the completed state and wait for
   // all insert operations to complete.
   batchEmployees.Complete();
   insertEmployees.Completion.Wait();
}

// Displays information about several random employees to the console.
static void GetRandomEmployees(string connectionString, int batchSize,
   int count)
{
   // Create a BatchedJoinBlock<Employee, Exception> object that holds
   // both employee and exception data.
   var selectEmployees = new BatchedJoinBlock<Employee, Exception>(batchSize);

   // Holds the total number of exceptions that occurred.
   int totalErrors = 0;

   // Create an action block that prints employee and error information
   // to the console.
   var printEmployees =
      new ActionBlock<Tuple<IList<Employee>, IList<Exception>>>(data =>
      {
         // Print information about the employees in this batch.
         Console.WriteLine("Received a batch...");
         foreach (Employee e in data.Item1)
         {
            Console.WriteLine("Last={0} First={1} ID={2}",
               e.LastName, e.FirstName, e.EmployeeID);
         }

         // Print the error count for this batch.
         Console.WriteLine("There were {0} errors in this batch...",
            data.Item2.Count);

         // Update total error count.
         totalErrors += data.Item2.Count;
      });

   // Link the batched join block to the action block.
   selectEmployees.LinkTo(printEmployees);

   // When the batched join block completes, set the action block also to complete.
   selectEmployees.Completion.ContinueWith(delegate { printEmployees.Complete(); });
```

```
      selectEmployees.Completion.ContinueWith(delegate { printEmployees.Complete(); });

      // Try to retrieve the ID for several random employees.
      Console.WriteLine("Selecting random entries from Employees table...");
      for (int i = 0; i < count; i++)
      {
         try
         {
            // Create a random employee.
            Employee e = Employee.Random();

            // Try to retrieve the ID for the employee from the database.
            e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString);

            // Post the Employee object to the Employee target of
            // the batched join block.
            selectEmployees.Target1.Post(e);
         }
         catch (NullReferenceException e)
         {
            // GetEmployeeID throws NullReferenceException when there is
            // no such employee with the given name. When this happens,
            // post the Exception object to the Exception target of
            // the batched join block.
            selectEmployees.Target2.Post(e);
         }
      }

      // Set the batched join block to the completed state and wait for
      // all retrieval operations to complete.
      selectEmployees.Complete();
      printEmployees.Completion.Wait();

      // Print the total error count.
      Console.WriteLine("Finished. There were {0} total errors.", totalErrors);
   }

   static void Main(string[] args)
   {
      // Create a connection string for accessing the database.
      // The connection string refers to the temporary database location.
      string connectionString = string.Format(@"Data Source={0}",
         scratchDatabase);

      // Create a Stopwatch object to time database insert operations.
      Stopwatch stopwatch = new Stopwatch();

      // Start with a clean database file by copying the source database to
      // the temporary location.
      File.Copy(sourceDatabase, scratchDatabase, true);

      // Demonstrate multiple insert operations without batching.
      Console.WriteLine("Demonstrating non-batched database insert operations...");
      Console.WriteLine("Original size of Employee table: {0}.",
         GetEmployeeCount(connectionString));
      stopwatch.Start();
      AddEmployees(connectionString, insertCount);
      stopwatch.Stop();
      Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
         GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds);

      Console.WriteLine();

      // Start again with a clean database file.
      File.Copy(sourceDatabase, scratchDatabase, true);

      // Demonstrate multiple insert operations, this time with batching.
      Console.WriteLine("Demonstrating batched database insert operations...");
      Console.WriteLine("Original size of Employee table: {0}.",
```

```
            GetEmployeeCount(connectionString));
         stopwatch.Restart();
         AddEmployeesBatched(connectionString, insertBatchSize, insertCount);
         stopwatch.Stop();
         Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
            GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds);

         Console.WriteLine();

         // Start again with a clean database file.
         File.Copy(sourceDatabase, scratchDatabase, true);

         // Demonstrate multiple retrieval operations with error reporting.
         Console.WriteLine("Demonstrating batched join database select operations...");
         // Add a small number of employees to the database.
         AddEmployeesBatched(connectionString, insertBatchSize, 16);
         // Query for random employees.
         GetRandomEmployees(connectionString, insertBatchSize, 10);
      }
   }
}
/* Sample output:
Demonstrating non-batched database insert operations...
Original size of Employee table: 15.
Adding 256 entries to Employee table...
New size of Employee table: 271; elapsed insert time: 11035 ms.

Demonstrating batched database insert operations...
Original size of Employee table: 15.
Adding 256 entries to Employee table...
New size of Employee table: 271; elapsed insert time: 197 ms.

Demonstrating batched join database insert operations...
Adding 16 entries to Employee table...
Selecting items from Employee table...
Received a batch...
Last=Jones First=Tom ID=21
Last=Jones First=John ID=24
Last=Smith First=Tom ID=26
Last=Jones First=Tom ID=21
There were 4 errors in this batch...
Received a batch...
Last=Smith First=Tom ID=26
Last=Jones First=Mike ID=28
There were 0 errors in this batch...
Finished. There were 4 total errors.
*/
```

```
Imports System.Collections.Generic
Imports System.Data.SqlServerCe
Imports System.Diagnostics
Imports System.IO
Imports System.Threading.Tasks.Dataflow


' Demonstrates how to use batched dataflow blocks to improve
' the performance of database operations.
Namespace DataflowBatchDatabase
    Friend Class Program
        ' The number of employees to add to the database.
        ' TODO: Change this value to experiment with different numbers of
        ' employees to insert into the database.
        Private Shared ReadOnly insertCount As Integer = 256

        ' The size of a single batch of employees to add to the database.
        ' TODO: Change this value to experiment with different batch sizes.
        Private Shared ReadOnly insertBatchSize As Integer = 96
```

```
        ' The source database file.
        ' TODO: Change this value if Northwind.sdf is at a different location
        ' on your computer.
        Private Shared ReadOnly sourceDatabase As String = "C:\Program Files\Microsoft SQL Server Compact
Edition\v3.5\Samples\Northwind.sdf"

        ' TODO: Change this value if you require a different temporary location.
        Private Shared ReadOnly scratchDatabase As String = "C:\Temp\Northwind.sdf"

        ' Describes an employee. Each property maps to a
        ' column in the Employees table in the Northwind database.
        ' For brevity, the Employee class does not contain
        ' all columns from the Employees table.
        Private Class Employee
            Public Property EmployeeID() As Integer
            Public Property LastName() As String
            Public Property FirstName() As String

            ' A random number generator that helps tp generate
            ' Employee property values.
            Private Shared rand As New Random(42)

            ' Possible random first names.
            Private Shared ReadOnly firstNames() As String = {"Tom", "Mike", "Ruth", "Bob", "John"}
            ' Possible random last names.
            Private Shared ReadOnly lastNames() As String = {"Jones", "Smith", "Johnson", "Walker"}

            ' Creates an Employee object that contains random
            ' property values.
            Public Shared Function Random() As Employee
                Return New Employee With {.EmployeeID = -1, .LastName = lastNames(rand.Next() Mod
lastNames.Length), .FirstName = firstNames(rand.Next() Mod firstNames.Length)}
            End Function
        End Class

        ' Adds new employee records to the database.
        Private Shared Sub InsertEmployees(ByVal employees() As Employee, ByVal connectionString As String)
            Using connection As New SqlCeConnection(connectionString)
                Try
                    ' Create the SQL command.
                    Dim command As New SqlCeCommand("INSERT INTO Employees ([Last Name], [First Name])" &
"VALUES (@lastName, @firstName)", connection)

                    connection.Open()
                    For i As Integer = 0 To employees.Length - 1
                        ' Set parameters.
                        command.Parameters.Clear()
                        command.Parameters.Add("@lastName", employees(i).LastName)
                        command.Parameters.Add("@firstName", employees(i).FirstName)

                        ' Execute the command.
                        command.ExecuteNonQuery()
                    Next i
                Finally
                    connection.Close()
                End Try
            End Using
        End Sub

        ' Retrieves the number of entries in the Employees table in
        ' the Northwind database.
        Private Shared Function GetEmployeeCount(ByVal connectionString As String) As Integer
            Dim result As Integer = 0
            Using sqlConnection As New SqlCeConnection(connectionString)
                Dim sqlCommand As New SqlCeCommand("SELECT COUNT(*) FROM Employees", sqlConnection)

                sqlConnection.Open()
                Try
```

```
                    result = CInt(Fix(sqlCommand.ExecuteScalar()))
                Finally
                    sqlConnection.Close()
                End Try
            End Using
            Return result
        End Function

        ' Retrieves the ID of the first employee that has the provided name.
        Private Shared Function GetEmployeeID(ByVal lastName As String, ByVal firstName As String, ByVal
connectionString As String) As Integer
            Using connection As New SqlCeConnection(connectionString)
                Dim command As New SqlCeCommand(String.Format("SELECT [Employee ID] FROM Employees " &
"WHERE [Last Name] = '{0}' AND [First Name] = '{1}'", lastName, firstName), connection)

                connection.Open()
                Try
                    Return CInt(Fix(command.ExecuteScalar()))
                Finally
                    connection.Close()
                End Try
            End Using
        End Function

        ' Posts random Employee data to the provided target block.
        Private Shared Sub PostRandomEmployees(ByVal target As ITargetBlock(Of Employee), ByVal count As
Integer)
            Console.WriteLine("Adding {0} entries to Employee table...", count)

            For i As Integer = 0 To count - 1
                target.Post(Employee.Random())
            Next i
        End Sub

        ' Adds random employee data to the database by using dataflow.
        Private Shared Sub AddEmployees(ByVal connectionString As String, ByVal count As Integer)
            ' Create an ActionBlock<Employee> object that adds a single
            ' employee entry to the database.
            Dim insertEmployee = New ActionBlock(Of Employee)(Sub(e) InsertEmployees(New Employee() {e},
connectionString))

            ' Post several random Employee objects to the dataflow block.
            PostRandomEmployees(insertEmployee, count)

            ' Set the dataflow block to the completed state and wait for
            ' all insert operations to complete.
            insertEmployee.Complete()
            insertEmployee.Completion.Wait()
        End Sub

        ' Adds random employee data to the database by using dataflow.
        ' This method is similar to AddEmployees except that it uses batching
        ' to add multiple employees to the database at a time.
        Private Shared Sub AddEmployeesBatched(ByVal connectionString As String, ByVal batchSize As Integer,
ByVal count As Integer)
            ' Create a BatchBlock<Employee> that holds several Employee objects and
            ' then propagates them out as an array.
            Dim batchEmployees = New BatchBlock(Of Employee)(batchSize)

            ' Create an ActionBlock<Employee[]> object that adds multiple
            ' employee entries to the database.
            Dim insertEmployees = New ActionBlock(Of Employee())(Sub(a) Program.InsertEmployees(a,
connectionString))

            ' Link the batch block to the action block.
            batchEmployees.LinkTo(insertEmployees)

            ' When the batch block completes, set the action block also to complete.
            batchEmployees.Completion.ContinueWith(Sub() insertEmployees.Complete())
```

```
            ' Post several random Employee objects to the batch block.
            PostRandomEmployees(batchEmployees, count)

            ' Set the batch block to the completed state and wait for
            ' all insert operations to complete.
            batchEmployees.Complete()
            insertEmployees.Completion.Wait()
        End Sub

        ' Displays information about several random employees to the console.
        Private Shared Sub GetRandomEmployees(ByVal connectionString As String, ByVal batchSize As Integer,
ByVal count As Integer)
            ' Create a BatchedJoinBlock<Employee, Exception> object that holds
            ' both employee and exception data.
            Dim selectEmployees = New BatchedJoinBlock(Of Employee, Exception)(batchSize)

            ' Holds the total number of exceptions that occurred.
            Dim totalErrors As Integer = 0

            ' Create an action block that prints employee and error information
            ' to the console.
            Dim printEmployees = New ActionBlock(Of Tuple(Of IList(Of Employee), IList(Of Exception)))
(Sub(data)
                                                                                                                '
Print information about the employees in this batch.
                                                                                                                '
Print the error count for this batch.
                                                                                                                '
Update total error count.

Console.WriteLine("Received a batch...")

For Each e As Employee In data.Item1

Console.WriteLine("Last={0} First={1} ID={2}", e.LastName, e.FirstName, e.EmployeeID)

Next e

Console.WriteLine("There were {0} errors in this batch...", data.Item2.Count)

totalErrors += data.Item2.Count
                                                                                                    End
Sub)

            ' Link the batched join block to the action block.
            selectEmployees.LinkTo(printEmployees)

            ' When the batched join block completes, set the action block also to complete.
            selectEmployees.Completion.ContinueWith(Sub() printEmployees.Complete())

            ' Try to retrieve the ID for several random employees.
            Console.WriteLine("Selecting random entries from Employees table...")
            For i As Integer = 0 To count - 1
                Try
                    ' Create a random employee.
                    Dim e As Employee = Employee.Random()

                    ' Try to retrieve the ID for the employee from the database.
                    e.EmployeeID = GetEmployeeID(e.LastName, e.FirstName, connectionString)

                    ' Post the Employee object to the Employee target of
                    ' the batched join block.
                    selectEmployees.Target1.Post(e)
                Catch e As NullReferenceException
                    ' GetEmployeeID throws NullReferenceException when there is
                    ' no such employee with the given name. When this happens,
                    ' post the Exception object to the Exception target of
                    ' the batched join block.
```

```
                    selectEmployees.Target2.Post(e)
                End Try
            Next i

            ' Set the batched join block to the completed state and wait for
            ' all retrieval operations to complete.
            selectEmployees.Complete()
            printEmployees.Completion.Wait()

            ' Print the total error count.
            Console.WriteLine("Finished. There were {0} total errors.", totalErrors)
        End Sub

        Shared Sub Main(ByVal args() As String)
            ' Create a connection string for accessing the database.
            ' The connection string refers to the temporary database location.
            Dim connectionString As String = String.Format("Data Source={0}", scratchDatabase)

            ' Create a Stopwatch object to time database insert operations.
            Dim stopwatch As New Stopwatch()

            ' Start with a clean database file by copying the source database to
            ' the temporary location.
            File.Copy(sourceDatabase, scratchDatabase, True)

            ' Demonstrate multiple insert operations without batching.
            Console.WriteLine("Demonstrating non-batched database insert operations...")
            Console.WriteLine("Original size of Employee table: {0}.", GetEmployeeCount(connectionString))
            stopwatch.Start()
            AddEmployees(connectionString, insertCount)
            stopwatch.Stop()
            Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds)

            Console.WriteLine()

            ' Start again with a clean database file.
            File.Copy(sourceDatabase, scratchDatabase, True)

            ' Demonstrate multiple insert operations, this time with batching.
            Console.WriteLine("Demonstrating batched database insert operations...")
            Console.WriteLine("Original size of Employee table: {0}.", GetEmployeeCount(connectionString))
            stopwatch.Restart()
            AddEmployeesBatched(connectionString, insertBatchSize, insertCount)
            stopwatch.Stop()
            Console.WriteLine("New size of Employee table: {0}; elapsed insert time: {1} ms.",
GetEmployeeCount(connectionString), stopwatch.ElapsedMilliseconds)

            Console.WriteLine()

            ' Start again with a clean database file.
            File.Copy(sourceDatabase, scratchDatabase, True)

            ' Demonstrate multiple retrieval operations with error reporting.
            Console.WriteLine("Demonstrating batched join database select operations...")
            ' Add a small number of employees to the database.
            AddEmployeesBatched(connectionString, insertBatchSize, 16)
            ' Query for random employees.
            GetRandomEmployees(connectionString, insertBatchSize, 10)
        End Sub
    End Class
End Namespace
' Sample output:
'Demonstrating non-batched database insert operations...
'Original size of Employee table: 15.
'Adding 256 entries to Employee table...
'New size of Employee table: 271; elapsed insert time: 11035 ms.
'
'Demonstrating batched database insert operations...
```

```
'Original size of Employee table: 15.
'Adding 256 entries to Employee table...
'New size of Employee table: 271; elapsed insert time: 197 ms.
'
'Demonstrating batched join database insert operations...
'Adding 16 entries to Employee table...
'Selecting items from Employee table...
'Received a batch...
'Last=Jones First=Tom ID=21
'Last=Jones First=John ID=24
'Last=Smith First=Tom ID=26
'Last=Jones First=Tom ID=21
'There were 4 errors in this batch...
'Received a batch...
'Last=Smith First=Tom ID=26
'Last=Jones First=Mike ID=28
'There were 0 errors in this batch...
'Finished. There were 4 total errors.
'
```

另请参阅

- 数据流

TPL 和传统 .NET 异步编程

2021/11/16 •

.NET 提供了以下两种标准模式，用于执行 I/O 密集型和计算密集型异步操作：

- 异步编程模型 (APM)，在该模式中异步操作由一对 Begin/End 方法表示。例如 FileStream.BeginRead 和 Stream.EndRead。
- 基于事件的异步模式 (EAP)，在该模式中异步操作由名为 `<OperationName>Async` 和 `<OperationName>Completed` 的方法/事件对表示。例如 WebClient.DownloadStringAsync 和 WebClient.DownloadStringCompleted。

任务并行库 (TPL) 可采用各种方法与任一异步模式协同使用。可将 APM 和 EAP 操作作为 `Task` 对象向库使用者公开，也可以公开 APM 模式但用 `Task` 对象在内部实现它们。在这两种情况下，可使用 `Task` 对象简化代码以及利用以下有用的功能：

- 在任务开始后随时以任务延续形式注册回调。
- 使用 ContinueWhenAll 和 ContinueWhenAny 方法，或者 WaitAll 和 WaitAny 方法并列为响应 `Begin_` 方法而执行的多个操作。
- 封装同一 `Task` 对象中的异步 I/O 密集型和计算密集型操作。
- 监视 `Task` 对象的状态。
- 使用 TaskCompletionSource<TResult> 将操作状态封送处理至 `Task` 对象。

在 Task 中包装 APM 操作

System.Threading.Tasks.TaskFactory 和 System.Threading.Tasks.TaskFactory<TResult> 类都提供了 TaskFactory.FromAsync 和 TaskFactory<TResult>.FromAsync 方法的几个重载，由此可将 APM begin/end 方法对封装在 Task 或 Task<TResult> 实例中。各种重载都可容纳任何具有零至三个输入参数的 begin/end 方法对。

对于具有返回值（在 Visual Basic 中为 `Function`）的 `End` 方法的对，使用 TaskFactory<TResult> 中创建 Task<TResult> 的方法。对于具有返回 void（在 Visual Basic 中为 `Sub`）的 `End` 方法，使用 TaskFactory 中创建 Task 的方法。

在极少情况下，如果 `Begin` 方法具有三个以上参数或包含 `ref` 或 `out` 参数，则提供仅封装 `End` 方法的其他 `FromAsync` 重载。

下面的示例显示了匹配 FileStream.BeginRead 和 FileStream.EndRead 方法的 `FromAsync` 重载的签名。

```
public Task<TResult> FromAsync<TArg1, TArg2, TArg3>(
    Func<TArg1, TArg2, TArg3, AsyncCallback, object, IAsyncResult> beginMethod, //BeginRead
     Func<IAsyncResult, TResult> endMethod, //EndRead
     TArg1 arg1, // the byte[] buffer
     TArg2 arg2, // the offset in arg1 at which to start writing data
     TArg3 arg3, // the maximum number of bytes to read
     object state // optional state information
    )
```

```
Public Function FromAsync(Of TArg1, TArg2, TArg3)(
                ByVal beginMethod As Func(Of TArg1, TArg2, TArg3, AsyncCallback, Object, IAsyncResult),
                ByVal endMethod As Func(Of IAsyncResult, TResult),
                ByVal dataBuffer As TArg1,
                ByVal byteOffsetToStartAt As TArg2,
                ByVal maxBytesToRead As TArg3,
                ByVal stateInfo As Object)
```

此重载采用三个输入参数，如下所示。第一个参数是匹配 FileStream.BeginRead 方法签名的 Func<T1,T2,T3,T4,T5,TResult> 委托。第二个参数使用 IAsyncResult 并返回 `TResult` 的 Func<T,TResult> 委托。由于 EndRead 返回一个整数，因此编译器会将 `TResult` 类型推断为 Int32 并将任务类型推断为 Task。最后第四个参数与 FileStream.BeginRead 方法中的参数相同：

- 存储文件数据的缓冲区。
- 开始写入数据的缓冲区的偏移量。
- 要从文件中读取的最大数据量。
- 存储要传递至回调的用户定义状态数据的可选对象。

使用 ContinueWith 执行回调功能

如果需要访问文件中的数据，而不仅仅访问字节数，则 FromAsync 方法不能满足此操作。请改用 Task，其 `Result` 属性包含文件数据。可以通过向原始任务添加延续来实现这种操作。延续执行通常由 AsyncCallback 委托执行的任务。先前任务完成且填充了数据缓冲区后调用此操作。（FileStream 对象应在返回前关闭。）

下面的示例演示如何返回封装 FileStream 类的 `BeginRead` / `EndRead` 对的 Task。

```
const int MAX_FILE_SIZE = 14000000;
public static Task<string> GetFileStringAsync(string path)
{
    FileInfo fi = new FileInfo(path);
    byte[] data = null;
    data = new byte[fi.Length];

    FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, data.Length, true);

    //Task<int> returns the number of bytes read
    Task<int> task = Task<int>.Factory.FromAsync(
            fs.BeginRead, fs.EndRead, data, 0, data.Length, null);

    // It is possible to do other work here while waiting
    // for the antecedent task to complete.
    // ...

    // Add the continuation, which returns a Task<string>.
    return task.ContinueWith((antecedent) =>
    {
        fs.Close();

        // Result = "number of bytes read" (if we need it.)
        if (antecedent.Result < 100)
        {
            return "Data is too small to bother with.";
        }
        else
        {
            // If we did not receive the entire file, the end of the
            // data buffer will contain garbage.
            if (antecedent.Result < data.Length)
                Array.Resize(ref data, antecedent.Result);

            // Will be returned in the Result property of the Task<string>
            // at some future point after the asynchronous file I/O operation completes.
            return new UTF8Encoding().GetString(data);
        }
    });
}
```

```
Const MAX_FILE_SIZE As Integer = 14000000
Shared Function GetFileStringAsync(ByVal path As String) As Task(Of String)
    Dim fi As New FileInfo(path)
    Dim data(fi.Length - 1) As Byte

    Dim fs As FileStream = New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, data.Length,
True)

    ' Task(Of Integer) returns the number of bytes read
    Dim myTask As Task(Of Integer) = Task(Of Integer).Factory.FromAsync(
        AddressOf fs.BeginRead, AddressOf fs.EndRead, data, 0, data.Length, Nothing)

    ' It is possible to do other work here while waiting
    ' for the antecedent task to complete.
    ' ...

    ' Add the continuation, which returns a Task<string>.
    Return myTask.ContinueWith(Function(antecedent)
                                   fs.Close()
                                   If (antecedent.Result < 100) Then
                                       Return "Data is too small to bother with."
                                   End If
                                   ' If we did not receive the entire file, the end of the
                                   ' data buffer will contain garbage.
                                   If (antecedent.Result < data.Length) Then
                                       Array.Resize(data, antecedent.Result)
                                   End If

                                   ' Will be returned in the Result property of the Task<string>
                                   ' at some future point after the asynchronous file I/O operation
completes.
                                   Return New UTF8Encoding().GetString(data)
                               End Function)

End Function
```

然后可调用此方法，如下所示。

```
Task<string> t = GetFileStringAsync(path);

// Do some other work:
// ...

try
{
     Console.WriteLine(t.Result.Substring(0, 500));
}
catch (AggregateException ae)
{
    Console.WriteLine(ae.InnerException.Message);
}
```

```
Dim myTask As Task(Of String) = GetFileStringAsync(path)

' Do some other work
' ...

Try
    Console.WriteLine(myTask.Result.Substring(0, 500))
Catch ex As AggregateException
    Console.WriteLine(ex.InnerException.Message)
End Try
```

提供自定义状态数据

在通常的 IAsyncResult 操作中，如果AsyncCallback 委托需要一些自定义状态数据，则必须通过 `Begin` 方法中的最后一个参数将它传入，以便可将数据打包到最终要传递至回调方法的 IAsyncResult 对象中。当使用 `FromAsync` 方法时，通常无需此操作。如果延续知道自定义数据，可直接在延续委托中捕获它。下面的示例与以前的示例类似，但延续检查此延续的用户委托可直接访问的自定义状态数据，而不是检查历史任务的 `Result` 属性。

```
public Task<string> GetFileStringAsync2(string path)
{
    FileInfo fi = new FileInfo(path);
    byte[] data = new byte[fi.Length];
    MyCustomState state = GetCustomState();
    FileStream fs = new FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, data.Length, true);
    // We still pass null for the last parameter because
    // the state variable is visible to the continuation delegate.
    Task<int> task = Task<int>.Factory.FromAsync(
            fs.BeginRead, fs.EndRead, data, 0, data.Length, null);

    return task.ContinueWith((antecedent) =>
    {
        // It is safe to close the filestream now.
        fs.Close();

        // Capture custom state data directly in the user delegate.
        // No need to pass it through the FromAsync method.
        if (state.StateData.Contains("New York, New York"))
        {
            return "Start spreading the news!";
        }
        else
        {
            // If we did not receive the entire file, the end of the
            // data buffer will contain garbage.
            if (antecedent.Result < data.Length)
                Array.Resize(ref data, antecedent.Result);

            // Will be returned in the Result property of the Task<string>
            // at some future point after the asynchronous file I/O operation completes.
            return new UTF8Encoding().GetString(data);
        }
    });
}
```

```
Public Function GetFileStringAsync2(ByVal path As String) As Task(Of String)
    Dim fi = New FileInfo(path)
    Dim data(fi.Length - 1) As Byte
    Dim state As New MyCustomState()

    Dim fs As New FileStream(path, FileMode.Open, FileAccess.Read, FileShare.Read, data.Length, True)
    ' We still pass null for the last parameter because
    ' the state variable is visible to the continuation delegate.
    Dim myTask As Task(Of Integer) = Task(Of Integer).Factory.FromAsync(
            AddressOf fs.BeginRead, AddressOf fs.EndRead, data, 0, data.Length, Nothing)

    Return myTask.ContinueWith(Function(antecedent)
                                   fs.Close()
                                   ' Capture custom state data directly in the user delegate.
                                   ' No need to pass it through the FromAsync method.
                                   If (state.StateData.Contains("New York, New York")) Then
                                       Return "Start spreading the news!"
                                   End If

                                   ' If we did not receive the entire file, the end of the
                                   ' data buffer will contain garbage.
                                   If (antecedent.Result < data.Length) Then
                                       Array.Resize(data, antecedent.Result)
                                   End If
                                   '/ Will be returned in the Result property of the Task<string>
                                   '/ at some future point after the asynchronous file I/O operation
completes.
                                   Return New UTF8Encoding().GetString(data)
                               End Function)

End Function
```

同步多个 FromAsync 任务

当结合使用 `FromAsync` 方法时，静态 ContinueWhenAll 和 ContinueWhenAny 方法具有更大的灵活性。下面的示例显示如何启动多个异步 I/O 操作，然后等待所有这些操作都完成后再执行延续。

```
public Task<string> GetMultiFileData(string[] filesToRead)
{
    FileStream fs;
    Task<string>[] tasks = new Task<string>[filesToRead.Length];
    byte[] fileData = null;
    for (int i = 0; i < filesToRead.Length; i++)
    {
        fileData = new byte[0x1000];
        fs = new FileStream(filesToRead[i], FileMode.Open, FileAccess.Read, FileShare.Read, fileData.Length,
true);

        // By adding the continuation here, the
        // Result of each task will be a string.
        tasks[i] = Task<int>.Factory.FromAsync(
                 fs.BeginRead, fs.EndRead, fileData, 0, fileData.Length, null)
                 .ContinueWith((antecedent) =>
                     {
                         fs.Close();

                         // If we did not receive the entire file, the end of the
                         // data buffer will contain garbage.
                         if (antecedent.Result < fileData.Length)
                             Array.Resize(ref fileData, antecedent.Result);

                         // Will be returned in the Result property of the Task<string>
                         // at some future point after the asynchronous file I/O operation completes.
                         return new UTF8Encoding().GetString(fileData);
                     });
    }

    // Wait for all tasks to complete.
    return Task<string>.Factory.ContinueWhenAll(tasks, (data) =>
    {
        // Propagate all exceptions and mark all faulted tasks as observed.
        Task.WaitAll(data);

        // Combine the results from all tasks.
        StringBuilder sb = new StringBuilder();
        foreach (var t in data)
        {
            sb.Append(t.Result);
        }
        // Final result to be returned eventually on the calling thread.
        return sb.ToString();
    });
}
```

```
Public Function GetMultiFileData(ByVal filesToRead As String()) As Task(Of String)
    Dim fs As FileStream
    Dim tasks(filesToRead.Length - 1) As Task(Of String)
    Dim fileData() As Byte = Nothing
    For i As Integer = 0 To filesToRead.Length
        fileData(&H1000) = New Byte()
        fs = New FileStream(filesToRead(i), FileMode.Open, FileAccess.Read, FileShare.Read, fileData.Length,
True)

        ' By adding the continuation here, the
        ' Result of each task will be a string.
        tasks(i) = Task(Of Integer).Factory.FromAsync(AddressOf fs.BeginRead,
                                                      AddressOf fs.EndRead,
                                                      fileData,
                                                      0,
                                                      fileData.Length,
                                                      Nothing).
                                                  ContinueWith(Function(antecedent)
                                                                   fs.Close()
                                                                   'If we did not receive the entire file,
the end of the
                                                                   ' data buffer will contain garbage.
                                                                   If (antecedent.Result < fileData.Length)
Then
                                                                       ReDim Preserve
fileData(antecedent.Result)
                                                                   End If

                                                                   'Will be returned in the Result property
of the Task<string>
                                                                   ' at some future point after the
asynchronous file I/O operation completes.
                                                                   Return New
UTF8Encoding().GetString(fileData)
                                                               End Function)
    Next

    Return Task(Of String).Factory.ContinueWhenAll(tasks, Function(data)

                                                           ' Propagate all exceptions and mark all
faulted tasks as observed.
                                                           Task.WaitAll(data)

                                                           ' Combine the results from all tasks.
                                                           Dim sb As New StringBuilder()
                                                           For Each t As Task(Of String) In data
                                                               sb.Append(t.Result)
                                                           Next
                                                           ' Final result to be returned eventually on
the calling thread.
                                                           Return sb.ToString()
                                                       End Function)
End Function
```

仅用于 End 方法的 FromAsync 任务

在极少情况下，如果 `Begin` 方法需要三个以上的输入参数，或具有 `ref` 或 `out` 参数，可以使用仅表示 `End` 方法的 `FromAsync` 重载，例如，TaskFactory<TResult>.FromAsync(IAsyncResult, Func<IAsyncResult,TResult>)。这些方法还可用于传递 IAsyncResult 并将其封装到 Task 的任何方案中。

```
static Task<String> ReturnTaskFromAsyncResult()
{
    IAsyncResult ar = DoSomethingAsynchronously();
    Task<String> t = Task<string>.Factory.FromAsync(ar, _ =>
        {
            return (string)ar.AsyncState;
        });

    return t;
}
```

```
Shared Function ReturnTaskFromAsyncResult() As Task(Of String)
    Dim ar As IAsyncResult = DoSomethingAsynchronously()
    Dim t As Task(Of String) = Task(Of String).Factory.FromAsync(ar, Function(res) CStr(res.AsyncState))
    Return t
End Function
```

开始和取消 FromAsync 任务

`FromAsync` 方法返回的任务具有 `WaitingForActivation` 状态，并在创建任务后在某个时刻由操作系统启动。如果尝试调用此类任务上的“启动”，将引发异常。

无法取消 `FromAsync` 任务，因为基础 .NET API 目前不支持取消正在进行中的文件或网络 I/O。可以将取消功能添加到封装 `FromAsync` 调用的方法中，但只能在调用 `FromAsync` 之前或在调用完成之后响应取消（例如，在延续任务中）。

一些支持 EAP 的类（如 WebClient）不支持取消，但可以通过使用取消标记集成该本机取消功能。

将复杂的 EAP 操作公开为任务

TPL 不提供任何专用于以 `FromAsync` 系列方法包装 IAsyncResult 模式相同的方式封装基于事件的异步操作的方法。但是，TPL 会提供 System.Threading.Tasks.TaskCompletionSource<TResult> 类，此类可用于将任意一组操作表示为 Task<TResult>。这些操作可能同步、可能异步，可能是 I/O 密集型、也可能是计算密集型，还可能两者都是。

下面的示例显示如何使用 TaskCompletionSource<TResult> 将一组异步 WebClient 操作作为基础 Task<TResult> 向客户端代码公开。此方法允许输入 Web URL 数组和术语或名称来进行搜索，然后返回每个站点搜索字词出现的次数。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Threading;
using System.Threading.Tasks;

public class SimpleWebExample
{
  public Task<string[]> GetWordCountsSimplified(string[] urls, string name,
                                                CancellationToken token)
  {
      TaskCompletionSource<string[]> tcs = new TaskCompletionSource<string[]>();
      WebClient[] webClients = new WebClient[urls.Length];
      object m_lock = new object();
      int count = 0;
      List<string> results = new List<string>();

      // If the user cancels the CancellationToken, then we can use the
      // WebClient's ability to cancel its own async operations.
      token.Register(() =>
      {
```

```
        {
            foreach (var wc in webClients)
            {
                if (wc != null)
                    wc.CancelAsync();
            }
        });

        for (int i = 0; i < urls.Length; i++)
        {
            webClients[i] = new WebClient();

            #region callback
            // Specify the callback for the DownloadStringCompleted
            // event that will be raised by this WebClient instance.
            webClients[i].DownloadStringCompleted += (obj, args) =>
            {

                // Argument validation and exception handling omitted for brevity.

                // Split the string into an array of words,
                // then count the number of elements that match
                // the search term.
                string[] words = args.Result.Split(' ');
                string NAME = name.ToUpper();
                int nameCount = (from word in words.AsParallel()
                                 where word.ToUpper().Contains(NAME)
                                 select word)
                                .Count();

                // Associate the results with the url, and add new string to the array that
                // the underlying Task object will return in its Result property.
                lock (m_lock)
                {
                   results.Add(String.Format("{0} has {1} instances of {2}", args.UserState, nameCount,
name));

                   // If this is the last async operation to complete,
                   // then set the Result property on the underlying Task.
                   count++;
                   if (count == urls.Length)
                   {
                      tcs.TrySetResult(results.ToArray());
                   }
                }
            };
            #endregion

            // Call DownloadStringAsync for each URL.
            Uri address = null;
            address = new Uri(urls[i]);
            webClients[i].DownloadStringAsync(address, address);
        } // end for

        // Return the underlying Task. The client code
        // waits on the Result property, and handles exceptions
        // in the try-catch block there.
        return tcs.Task;
    }
}
```

```
Imports System.Collections.Generic
Imports System.Net
Imports System.Threading
Imports System.Threading.Tasks

Public Class SimpleWebExample
    Dim tcs As New TaskCompletionSource(Of String())
```

```
    Dim token As CancellationToken
    Dim results As New List(Of String)
    Dim m_lock As New Object()
    Dim count As Integer
    Dim addresses() As String
    Dim nameToSearch As String

    Public Function GetWordCountsSimplified(ByVal urls() As String, ByVal str As String,
                                            ByVal token As CancellationToken) As Task(Of String())
        addresses = urls
        nameToSearch = str

        Dim webClients(urls.Length - 1) As WebClient

        ' If the user cancels the CancellationToken, then we can use the
        ' WebClient's ability to cancel its own async operations.
        token.Register(Sub()
                           For Each wc As WebClient In webClients
                               If wc IsNot Nothing Then
                                   wc.CancelAsync()
                               End If
                           Next
                       End Sub)

        For i As Integer = 0 To urls.Length - 1
            webClients(i) = New WebClient()

            ' Specify the callback for the DownloadStringCompleted
            ' event that will be raised by this WebClient instance.
            AddHandler webClients(i).DownloadStringCompleted, AddressOf WebEventHandler

            Dim address As New Uri(urls(i))
            ' Pass the address, and also use it for the userToken
            ' to identify the page when the delegate is invoked.
            webClients(i).DownloadStringAsync(address, address)
        Next

        ' Return the underlying Task. The client code
        ' waits on the Result property, and handles exceptions
        ' in the try-catch block there.
        Return tcs.Task
    End Function

    Public Sub WebEventHandler(ByVal sender As Object, ByVal args As DownloadStringCompletedEventArgs)

        If args.Cancelled = True Then
            tcs.TrySetCanceled()
            Return
        ElseIf args.Error IsNot Nothing Then
            tcs.TrySetException(args.Error)
            Return
        Else
            ' Split the string into an array of words,
            ' then count the number of elements that match
            ' the search term.
            Dim words() As String = args.Result.Split(" "c)

            Dim name As String = nameToSearch.ToUpper()
            Dim nameCount = (From word In words.AsParallel()
                             Where word.ToUpper().Contains(name)
                             Select word).Count()

            ' Associate the results with the url, and add new string to the array that
            ' the underlying Task object will return in its Result property.
            SyncLock (m_lock)
                results.Add(String.Format("{0} has {1} instances of {2}", args.UserState, nameCount,
nameToSearch))
                count = count + 1
                If (count = addresses.Length) Then
```

```
                    tcs.TrySetResult(results.ToArray())
                End If
            End SyncLock
        End If
    End Sub
End Class
```

有关包括其他异常处理且展示了如何通过客户端代码调用方法的更完整示例，请参阅如何：在任务中包装 EAP 模式。

请记住，通过 TaskCompletionSource<TResult> 创建的任何任务均由 `TaskCompletionSource` 启动，因此用户代码不应在此任务中调用 `Start` 方法。

使用任务实现 APM 模式

在某些情况下，可能需要通过使用 API 中 begin/end 方法对直接公开 IAsyncResult 模式。例如，可能想要与现有的 API 保持一致，或者可能具有需要这种模式的自动化工具。在这种情况下，可使用 `Task` 对象来简化在内部实现 APM 模式的方式。

下面的示例显示如何使用任务实现长时间运行计算密集型方法的 APM begin/end 方法对。

```
class Calculator
{
    public IAsyncResult BeginCalculate(int decimalPlaces, AsyncCallback ac, object state)
    {
        Console.WriteLine("Calling BeginCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId);
        Task<string> f = Task<string>.Factory.StartNew(_ => Compute(decimalPlaces), state);
        if (ac != null) f.ContinueWith((res) => ac(f));
        return f;
    }

    public string Compute(int numPlaces)
    {
        Console.WriteLine("Calling compute on thread {0}", Thread.CurrentThread.ManagedThreadId);

        // Simulating some heavy work.
        Thread.SpinWait(500000000);

        // Actual implemenation left as exercise for the reader.
        // Several examples are available on the Web.
        return "3.14159265358979323846264338327950288";
    }

    public string EndCalculate(IAsyncResult ar)
    {
        Console.WriteLine("Calling EndCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId);
        return ((Task<string>)ar).Result;
    }
}

public class CalculatorClient
{
    static int decimalPlaces = 12;
    public static void Main()
    {
        Calculator calc = new Calculator();
        int places = 35;

        AsyncCallback callBack = new AsyncCallback(PrintResult);
        IAsyncResult ar = calc.BeginCalculate(places, callBack, calc);

        // Do some work on this thread while the calulator is busy.
        Console.WriteLine("Working...");
        Thread.SpinWait(500000);
        Console.ReadLine();
    }

    public static void PrintResult(IAsyncResult result)
    {
        Calculator c = (Calculator)result.AsyncState;
        string piString = c.EndCalculate(result);
        Console.WriteLine("Calling PrintResult on thread {0}; result = {1}",
                    Thread.CurrentThread.ManagedThreadId, piString);
    }
}
```

```
Class Calculator
    Public Function BeginCalculate(ByVal decimalPlaces As Integer, ByVal ac As AsyncCallback, ByVal state As
Object) As IAsyncResult
        Console.WriteLine("Calling BeginCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId)
        Dim myTask = Task(Of String).Factory.StartNew(Function(obj) Compute(decimalPlaces), state)
        myTask.ContinueWith(Sub(antedecent) ac(myTask))

    End Function
    Private Function Compute(ByVal decimalPlaces As Integer)
        Console.WriteLine("Calling compute on thread {0}", Thread.CurrentThread.ManagedThreadId)

        ' Simulating some heavy work.
        Thread.SpinWait(500000000)

        ' Actual implemenation left as exercise for the reader.
        ' Several examples are available on the Web.
        Return "3.14159265358979323846264338327950288"
    End Function

    Public Function EndCalculate(ByVal ar As IAsyncResult) As String
        Console.WriteLine("Calling EndCalculate on thread {0}", Thread.CurrentThread.ManagedThreadId)
        Return CType(ar, Task(Of String)).Result
    End Function
End Class

Class CalculatorClient
    Shared decimalPlaces As Integer
    Shared Sub Main()
        Dim calc As New Calculator
        Dim places As Integer = 35
        Dim callback As New AsyncCallback(AddressOf PrintResult)
        Dim ar As IAsyncResult = calc.BeginCalculate(places, callback, calc)

        ' Do some work on this thread while the calulator is busy.
        Console.WriteLine("Working...")
        Thread.SpinWait(500000)
        Console.ReadLine()
    End Sub

    Public Shared Sub PrintResult(ByVal result As IAsyncResult)
        Dim c As Calculator = CType(result.AsyncState, Calculator)
        Dim piString As String = c.EndCalculate(result)
        Console.WriteLine("Calling PrintResult on thread {0}; result = {1}",
                    Thread.CurrentThread.ManagedThreadId, piString)
    End Sub

End Class
```

使用 StreamExtensions 示例代码

.NET Standard parallel extensions extras 存储库中的 StreamExtensions.cs 文件包含将 `Task` 对象用于异步文件和网络 I/O 的若干参考实现。

请参阅

- 任务并行库 (TPL)

如何：在任务中包装 EAP 模式

2021/11/16 •

下面的示例演示如何通过使用 TaskCompletionSource<TResult> 将任意基于事件的异步模式 (EAP) 操作序列公开为一个任务。此示例还展示了如何使用 CancellationToken 对 WebClient 对象调用内置的取消方法。

示例

```
class WebDataDownloader
{

    static void Main()
    {
        WebDataDownloader downloader = new WebDataDownloader();
        string[] addresses = { "http://www.msnbc.com", "http://www.yahoo.com",
                               "http://www.nytimes.com", "http://www.washingtonpost.com",
                               "http://www.latimes.com", "http://www.newsday.com" };
        CancellationTokenSource cts = new CancellationTokenSource();

        // Create a UI thread from which to cancel the entire operation
        Task.Factory.StartNew(() =>
        {
            Console.WriteLine("Press c to cancel");
            if (Console.ReadKey().KeyChar == 'c')
                cts.Cancel();
        });

        // Using a neutral search term that is sure to get some hits.
        Task<string[]> webTask = downloader.GetWordCounts(addresses, "the", cts.Token);

        // Do some other work here unless the method has already completed.
        if (!webTask.IsCompleted)
        {
            // Simulate some work.
            Thread.SpinWait(5000000);
        }

        string[] results = null;
        try
        {
            results = webTask.Result;
        }
        catch (AggregateException e)
        {
            foreach (var ex in e.InnerExceptions)
            {
                OperationCanceledException oce = ex as OperationCanceledException;
                if (oce != null)
                {
                    if (oce.CancellationToken == cts.Token)
                    {
                        Console.WriteLine("Operation canceled by user.");
                    }
                }
                else
                {
                    Console.WriteLine(ex.Message);
                }
            }
        }
        finally
```

```
        {
            cts.Dispose();
        }
        if (results != null)
        {
            foreach (var item in results)
                Console.WriteLine(item);
        }
        Console.ReadKey();
    }

    Task<string[]> GetWordCounts(string[] urls, string name, CancellationToken token)
    {
        TaskCompletionSource<string[]> tcs = new TaskCompletionSource<string[]>();
        WebClient[] webClients = new WebClient[urls.Length];

        // If the user cancels the CancellationToken, then we can use the
        // WebClient's ability to cancel its own async operations.
        token.Register(() =>
            {
                foreach (var wc in webClients)
                {
                    if (wc != null)
                        wc.CancelAsync();
                }
            });

        object m_lock = new object();
        int count = 0;
        List<string> results = new List<string>();
        for (int i = 0; i < urls.Length; i++)
        {
            webClients[i] = new WebClient();

            #region callback
            // Specify the callback for the DownloadStringCompleted
            // event that will be raised by this WebClient instance.
            webClients[i].DownloadStringCompleted += (obj, args) =>
             {
                 if (args.Cancelled == true)
                 {
                     tcs.TrySetCanceled();
                     return;
                 }
                 else if (args.Error != null)
                 {
                     // Pass through to the underlying Task
                     // any exceptions thrown by the WebClient
                     // during the asynchronous operation.
                     tcs.TrySetException(args.Error);
                     return;
                 }
                 else
                 {
                     // Split the string into an array of words,
                     // then count the number of elements that match
                     // the search term.
                     string[] words = null;
                     words = args.Result.Split(' ');
                     string NAME = name.ToUpper();
                     int nameCount = (from word in words.AsParallel()
                                      where word.ToUpper().Contains(NAME)
                                      select word)
                                     .Count();

                     // Associate the results with the url, and add new string to the array that
                     // the underlying Task object will return in its Result property.
                     results.Add(String.Format("{0} has {1} instances of {2}", args.UserState, nameCount,
name));
```

```
            }

            // If this is the last async operation to complete,
            // then set the Result property on the underlying Task.
            lock (m_lock)
            {
                count++;
                if (count == urls.Length)
                {
                    tcs.TrySetResult(results.ToArray());
                }
            }
         };
        #endregion

        // Call DownloadStringAsync for each URL.
        Uri address = null;
        try
        {
            address = new Uri(urls[i]);
            // Pass the address, and also use it for the userToken
            // to identify the page when the delegate is invoked.
            webClients[i].DownloadStringAsync(address, address);
        }

        catch (UriFormatException ex)
        {
            // Abandon the entire operation if one url is malformed.
            // Other actions are possible here.
            tcs.TrySetException(ex);
            return tcs.Task;
        }
    }

    // Return the underlying Task. The client code
    // waits on the Result property, and handles exceptions
    // in the try-catch block there.
    return tcs.Task;
}
```

```
Class WebDataDownLoader

    Dim tcs As New TaskCompletionSource(Of String())
    Dim nameToSearch As String
    Dim token As CancellationToken
    Dim results As New List(Of String)
    Dim m_lock As Object
    Dim count As Integer
    Dim addresses() As String

    Shared Sub Main()

        Dim downloader As New WebDataDownLoader()
        downloader.addresses = {"http://www.msnbc.com", "http://www.yahoo.com", _
                                "http://www.nytimes.com", "http://www.washingtonpost.com", _
                                "http://www.latimes.com", "http://www.newsday.com"}
        Dim cts As New CancellationTokenSource()

        ' Create a UI thread from which to cancel the entire operation
        Task.Factory.StartNew(Sub()
                                  Console.WriteLine("Press c to cancel")
                                  If Console.ReadKey().KeyChar = "c"c Then
                                      cts.Cancel()
                                  End If
                              End Sub)

        ' Using a neutral search term that is sure to get some hits on English web sites.
```

```
        ' Please substitute your favorite search term.
        downloader.nameToSearch = "the"
        Dim webTask As Task(Of String()) = downloader.GetWordCounts(downloader.addresses,
downloader.nameToSearch, cts.Token)

        ' Do some other work here unless the method has already completed.
        If (webTask.IsCompleted = False) Then
            ' Simulate some work
            Thread.SpinWait(5000000)
        End If

        Dim results As String() = Nothing
        Try
            results = webTask.Result
        Catch ae As AggregateException
            For Each ex As Exception In ae.InnerExceptions
                If (TypeOf (ex) Is OperationCanceledException) Then
                    Dim oce As OperationCanceledException = CType(ex, OperationCanceledException)
                    If oce.CancellationToken = cts.Token Then
                        Console.WriteLine("Operation canceled by user.")
                    End If
                Else
                    Console.WriteLine(ex.Message)
                End If

            Next
        Finally
            cts.Dispose()
        End Try

        If (Not results Is Nothing) Then
            For Each item As String In results
                Console.WriteLine(item)
            Next
        End If

        Console.WriteLine("Press any key to exit")
        Console.ReadKey()
    End Sub

    Public Function GetWordCounts(ByVal urls() As String, ByVal str As String, ByVal token As
CancellationToken) As Task(Of String())

        Dim webClients() As WebClient
        ReDim webClients(urls.Length)
        m_lock = New Object()

        ' If the user cancels the CancellationToken, then we can use the
        ' WebClient's ability to cancel its own async operations.
        token.Register(Sub()
                           For Each wc As WebClient In webClients
                               If Not wc Is Nothing Then
                                   wc.CancelAsync()
                               End If
                           Next
                       End Sub)


        For i As Integer = 0 To urls.Length - 1
            webClients(i) = New WebClient()

            ' Specify the callback for the DownloadStringCompleted
            ' event that will be raised by this WebClient instance.
            AddHandler webClients(i).DownloadStringCompleted, AddressOf WebEventHandler

            Dim address As Uri = Nothing
            Try
                address = New Uri(urls(i))
                ' Pass the address, and also use it for the userToken
```

```
                ' to identify the page when the delegate is invoked.
                webClients(i).DownloadStringAsync(address, address)
            Catch ex As UriFormatException
                tcs.TrySetException(ex)
                Return tcs.Task
            End Try

        Next

        ' Return the underlying Task. The client code
        ' waits on the Result property, and handles exceptions
        ' in the try-catch block there.
        Return tcs.Task
    End Function



    Public Sub WebEventHandler(ByVal sender As Object, ByVal args As DownloadStringCompletedEventArgs)

        If args.Cancelled = True Then
            tcs.TrySetCanceled()
            Return
        ElseIf Not args.Error Is Nothing Then
            tcs.TrySetException(args.Error)
            Return
        Else
            ' Split the string into an array of words,
            ' then count the number of elements that match
            ' the search term.
            Dim words() As String = args.Result.Split(" "c)
            Dim NAME As String = nameToSearch.ToUpper()
            Dim nameCount = (From word In words.AsParallel()
                             Where word.ToUpper().Contains(NAME)
                             Select word).Count()

            ' Associate the results with the url, and add new string to the array that
            ' the underlying Task object will return in its Result property.
            results.Add(String.Format("{0} has {1} instances of {2}", args.UserState, nameCount,
nameToSearch))
        End If

        SyncLock (m_lock)
            count = count + 1
            If (count = addresses.Length) Then
                tcs.TrySetResult(results.ToArray())
            End If
        End SyncLock
    End Sub
```

另请参阅

- TPL 和传统 .NET 异步编程

数据并行和任务并行中的潜在缺陷

2021/11/16 •

在许多情况下，与普通的顺序循环相比，Parallel.For 和 Parallel.ForEach 可以显著提升性能。但是，对循环进行并行化的工作增加了复杂性，可能会导致在顺序代码中出现不常见或根本不会遇到的问题。本主题列出了一些在编写并行循环时要避免的做法。

不要假定并行的速度始终更快

在某些情况下，并行循环可能比它等效的顺序循环的运行速度更慢。基本的经验法则是具有较少迭代和快速用户委托的并行循环未必会快很多。但是，由于性能会涉及到很多因素，因此我们建议始终衡量实际的结果。

避免写入共享内存位置

在顺序代码中，从静态变量或类字段中读取或写入静态变量或类字段的情况很常见。但是，每当多个线程同时访问此类变量时，则很有可能会出现争用条件。即使可以使用锁来同步对变量的访问，但同步开销可能会对性能造成损害。因此，我们建议尽可能地避免在一个并行循环中访问共享状态，或至少限制对共享状态的访问。为此，最好使用 Parallel.For 和 Parallel.ForEach 的重载，以便在循环执行期间使用 System.Threading.ThreadLocal<T> 变量存储线程本地状态。有关详细信息，请参阅如何：编写具有线程局部变量的 Parallel.For 循环和如何：使用分区本地变量编写 Parallel.ForEach 循环。

避免过度并行化

通过使用并行循环，将会产生对源集合进行分区和同步工作线程的开销成本。计算机上的处理器数量进一步限制了并行化的优点。仅在一个处理器上运行多个受计算限制的线程时，速度并不会得到提升。因此，必须要小心，不要对循环进行过度并行化。

在嵌套的循环中，最有可能发生过度并行化的情况。在大多数情况下，除非满足以下一个或多个条件，否则最好仅对外部循环进行并行化：

- 已知内部循环非常长。
- 正在对每个订单执行开销极大的计算。（示例中所示的操作开销不大。）
- 已知目标系统具有足够的处理器来处理通过对 `cust.Orders` 上的查询进行并行化所产生的线程数。

在所有情况下，确定最佳查询形式的最好方法是进行测试和测量。

避免调用非线程安全方法

如果从并行循环中写入非线程安全实例方法，可能会导致出现程序可能检测到也可能检测不到的数据损坏。还可能会导致异常。在下面的示例中，多个线程尝试同时调用 FileStream.WriteByte 方法，类并不支持这样做。

```
FileStream fs = File.OpenWrite(path);
byte[] bytes = new Byte[10000000];
// ...
Parallel.For(0, bytes.Length, (i) => fs.WriteByte(bytes[i]));
```

```
Dim fs As FileStream = File.OpenWrite(filepath)
Dim bytes() As Byte
ReDim bytes(1000000)
' ...init byte array
Parallel.For(0, bytes.Length, Sub(n) fs.WriteByte(bytes(n)))
```

限制调用线程安全方法

.NET 中的大多数静态方法是线程安全的，并且可以同时从多个线程中调用。但是，即使在这些情况下，所涉及到的同步也可能会导致查询速度大幅度下降。

> **NOTE**
> 可以自行对此进行测试，具体方法是在查询中插入一些 WriteLine 调用。尽管出于演示目的，在文档示例中使用了此方法，但除非必要，否则不要在并行循环中使用它。

注意线程关联问题

某些技术(例如，单线程单元 (STA) 组件的 COM 互操作性、Windows 窗体以及 Windows Presentation Foundation (WPF))具有要求代码在特定线程上运行的线程关联限制。例如，在 Windows 窗体和 WPF 中，只能在创建控件的线程上访问该控件。举例来说，这意味着，除非将线程调度器配置为仅将工作安排在 UI 线程上，否则你将无法从并行循环中更新列表控件。有关详细信息，请参阅指定同步上下文。

在由 Parallel.Invoke 调用的委托中等待时请谨慎使用

在某些情况下，任务并行库将对任务进行内联操作，这意味着它将在当前正在执行的线程上的任务上运行。(有关详细信息，请参阅任务计划程序。)此性能优化在某些情况下可能会导致死锁。例如，两个任务可能运行相同的委托代码，该代码在事件发生时将发出信号，然后等待另一个任务发出信号。如果在相同线程上将第二个任务内联为第一个，并且第一个任务进入等待状态，则第二个任务将永远无法发出其事件信号。为了避免发生这种情况，可以在等待操作上指定超时，或使用显式线程构造函数来帮助确保一个任务无法阻止另一个任务。

不要假定 ForEach、For 和 ForAll 的迭代始终并行执行

请务必注意，For、ForEach 或 ForAll 循环中的各个迭代可能会(但不需要)并行执行。因此，应避免编写任何依赖于迭代并行执行的正确性或依赖于按任何特定顺序执行迭代的代码。例如，此代码有可能会死锁：

```
ManualResetEventSlim mre = new ManualResetEventSlim();
Enumerable.Range(0, Environment.ProcessorCount * 100)
    .AsParallel()
    .ForAll((j) =>
        {
            if (j == Environment.ProcessorCount)
            {
                Console.WriteLine("Set on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Set();
            }
            else
            {
                Console.WriteLine("Waiting on {0} with value of {1}",
                    Thread.CurrentThread.ManagedThreadId, j);
                mre.Wait();
            }
        }); //deadlocks
```

```
Dim mres = New ManualResetEventSlim()
Enumerable.Range(0, Environment.ProcessorCount * 100) _
.AsParallel() _
.ForAll(Sub(j)

            If j = Environment.ProcessorCount Then
                Console.WriteLine("Set on {0} with value of {1}",
                                  Thread.CurrentThread.ManagedThreadId, j)
                mres.Set()
            Else
                Console.WriteLine("Waiting on {0} with value of {1}",
                                  Thread.CurrentThread.ManagedThreadId, j)
                mres.Wait()
            End If
        End Sub) ' deadlocks
```

在此示例中，一个迭代设置一个事件，而所有的其他迭代则等待该事件。在设置事件的迭代完成之前，任何等待迭代均无法完成。但是，在设置事件的迭代有机会执行之前，等待迭代可能会阻止用于执行并行循环的所有线程。这将导致死锁 – 设置事件的迭代将永不会执行，并且等待迭代将永远不会醒来。

具体而言，并行循环的一个迭代绝不应该等待循环的另一个迭代来继续执行。如果并行循环决定按相反的顺序安排迭代，则会发生死锁。

避免在 UI 线程上执行并行循环

务必要使应用程序的用户界面 (UI) 保持响应状态。如果操作包含足够的工作来保证并行化，则可能不应在 UI 线程上运行该操作。相反，而是应卸载要在后台线程上运行的该操作。例如，如果要使用并行循环来计算随后应在 UI 控件中呈现的某些数据，则应考虑在任务实例内执行循环，而不是直接在 UI 事件处理程序中执行。只有先当核心计算完成后，才应将 UI 更新封送回 UI 线程。

如果确实要在 UI 线程上运行并行循环，请当心，避免从循环内更新 UI 控件。如果尝试从在 UI 线程上执行的并行循环内更新 UI 控件，将可能会导致状态损坏、异常、更新延迟甚至死锁，具体将取决于 UI 更新的调用方式。在下面的示例中，在所有的迭代完成之前，并行循环将阻止它在其上执行的 UI 线程。不过，如果循环的迭代是对后台线程运行（就像 For 一样），调用 Invoke 可能会导致将消息提交到 UI 线程，并阻止等待处理相应消息。由于 UI 线程受阻止而无法运行 For，因此消息永远无法得到处理，且 UI 线程死锁。

```
private void button1_Click(object sender, EventArgs e)
{
    Parallel.For(0, N, i =>
    {
        // do work for i
        button1.Invoke((Action)delegate { DisplayProgress(i); });
    });
}
```

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click

    Dim iterations As Integer = 20
    Parallel.For(0, iterations, Sub(x)
                                    Button1.Invoke(Sub()
                                                       DisplayProgress(x)
                                                   End Sub)
                                End Sub)
End Sub
```

下面的示例演示如何通过在任务实例内运行循环来避免死锁。循环不会阻止 UI 线程，并且消息可得到处理。

```
private void button1_Click(object sender, EventArgs e)
{
    Task.Factory.StartNew(() =>
        Parallel.For(0, N, i =>
        {
            // do work for i
            button1.Invoke((Action)delegate { DisplayProgress(i); });
        })
         );
}
```

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click

    Dim iterations As Integer = 20
    Task.Factory.StartNew(Sub() Parallel.For(0, iterations, Sub(x)
                                                                Button1.Invoke(Sub()
                                                                                   DisplayProgress(x)
                                                                               End Sub)
                                                            End Sub))
End Sub
```

请参阅

- 并行编程
- PLINQ 的潜在问题
- 并行编程模式：了解并使用 .NET Framework 4 应用并行模式

PLINQ 介绍

2021/11/16 •

并行 LINQ (PLINQ) 是语言集成查询 (LINQ) 模式的并行实现。PLINQ 将整套 LINQ 标准查询运算符实现为 System.Linq 命名空间的扩展方法，并提供适用于并行操作的其他运算符。PLINQ 将 LINQ 语法的简洁和可靠性与并行编程的强大功能结合在一起。

> **TIP**
> 如果不熟悉 LINQ，则它具有统一的模型，用于以类型安全方式查询任何可枚举数据源。LINQ to Objects 是针对内存中集合（如 List<T> 和数组）运行的 LINQ 查询的名称。本文假定你对 LINQ 有基本的了解。有关详细信息，请参阅语言集成查询 (LINQ)。

什么是并行查询？

一个 PLINQ 查询的许多方面都类似于非并行的 LINQ to Objects 查询。与顺序 LINQ 查询一样，PLINQ 查询对任何内存中 IEnumerable 或 IEnumerable<T> 数据源执行操作，并且推迟了执行，即在枚举查询前不会开始执行。主要区别在于，PLINQ 会尝试充分利用系统上的所有处理器。方法是将数据源分区成片段，然后在多个处理器上针对单独工作线程上的每个片段执行并行查询。在许多情况下，并行执行意味着查询运行速度显著提高。

通过并行执行，通常只需向数据源添加 AsParallel 查询操作，PLINQ 即可显著提升性能（与某些类型查询的旧代码相比）。但是，并行可能会引入其自身的复杂性，因此并非所有的查询操作的运行速度在 PLINQ 中都更快。事实上，并行实际上会降低某些查询的速度。因此，应了解排序等问题将如何对并行查询产生影响。有关详细信息，请参阅了解 PLINQ 中的加速。

> **NOTE**
> 本文档使用 lambda 表达式在 PLINQ 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

本文的其余部分将概述主 PLINQ 类，并讨论如何创建 PLINQ 查询。每部分包含指向更详细信息以及代码示例的链接。

ParallelEnumerable 类

System.Linq.ParallelEnumerable 类公开了几乎所有的 PLINQ 功能。它和 System.Linq 命名空间类型的其余部分一起被编译到 System.Core.dll 程序集中。Visual Studio 中默认的 C# 和 Visual Basic 项目均会引用该程序集并导入该命名空间。

尽管 ParallelEnumerable 实现了 LINQ to Objects 支持的所有标准查询运算符，但它不会尝试并行执行每个实现。如果你不熟悉 LINQ，请参阅 LINQ (C#) 简介和 LINQ (Visual Basic) 简介。

除了标准查询运算符外，ParallelEnumerable 类还包含一组启用并行执行专用行为的方法。下表中列出了这些特定于 PLINQ 的方法。

PARALLELENUMERABLE	
AsParallel	PLINQ 的入口点。指定如果可能，应并行化查询的其余部分。

PARALLELENUMERABLE 成员	说明
AsSequential	指定查询的其余部分应像非并行的 LINQ 查询一样按顺序运行。
AsOrdered	指定 PLINQ 应为查询的其余部分保留源序列的排序，或直到例如通过使用 orderby（在 Visual Basic 中为 Order By）子句更改排序为止。
AsUnordered	指定保留源序列的排序不需要查询其余部分的 PLINQ。
WithCancellation	指定 PLINQ 应定期监视请求取消时所提供的取消标记的状态以及取消执行。
WithDegreeOfParallelism	指定 PLINQ 应用于并行化查询的处理器的最大数量。
WithMergeOptions	提供有关 PLINQ 应如何（如果可能）将并行结果合并回使用线程上的一个序列的提示。
WithExecutionMode	指定 PLINQ 应如何并行化查询（即使是当默认行为是按顺序运行查询时）。
ForAll	一种多线程枚举方法，与循环访问查询结果不同，它允许在不首先合并回使用者线程的情况下并行处理结果。
Aggregate 重载	对于 PLINQ 唯一的重载，它启用对线程本地分区的中间聚合以及一个用于合并所有分区结果的最终聚合函数。

选择使用模型

编写查询时，请对数据源调用 ParallelEnumerable.AsParallel 扩展方法，以选择使用 PLINQ，如下面的示例所示。

```
var source = Enumerable.Range(1, 10000);

// Opt in to PLINQ with AsParallel.
var evenNums = from num in source.AsParallel()
               where num % 2 == 0
               select num;
Console.WriteLine("{0} even numbers out of {1} total",
                  evenNums.Count(), source.Count());
// The example displays the following output:
//       5000 even numbers out of 10000 total
```

```
Dim source = Enumerable.Range(1, 10000)

' Opt in to PLINQ with AsParallel
Dim evenNums = From num In source.AsParallel()
               Where num Mod 2 = 0
               Select num
Console.WriteLine("{0} even numbers out of {1} total",
                  evenNums.Count(), source.Count())
' The example displays the following output:
'       5000 even numbers out of 10000 total
```

AsParallel 扩展方法将后续查询运算符（在此示例中为 `where` 和 `select`）绑定到 System.Linq.ParallelEnumerable 实现。

执行模式

默认情况下，PLINQ 是保守的。在运行时，PLINQ 基础结构将分析查询的总体结构。如果通过并行可能会提高查询速度，PLINQ 则将源序列分区为可以同时运行的任务。如果并行化查询不安全，PLINQ 则只会按顺序运行查询。如果 PLINQ 可以在可能会较昂贵的并行算法或成本较低的顺序算法之间进行选择，它会默认选择顺序算法。可以使用 WithExecutionMode 方法和 System.Linq.ParallelExecutionMode 枚举指示 PLINQ 选择并行算法。如果你通过测试和测量知道特定查询以并行方式执行得更快时，此做法非常有用。有关详细信息，请参阅如何：在 PLINQ 中指定执行模式。

并行度

默认情况下，PLINQ 使用主机计算机上的所有处理器。可以使用 WithDegreeOfParallelism 方法指示 PLINQ 使用不超过指定数量的处理器。当你要确保计算机上运行的其他进程收到一定的 CPU 时间量时，此做法将非常有用。下面的片段将查询限制为最多使用两个处理器。

```
var query = from item in source.AsParallel().WithDegreeOfParallelism(2)
            where Compute(item) > 42
            select item;
```

```
Dim query = From item In source.AsParallel().WithDegreeOfParallelism(2)
            Where Compute(item) > 42
            Select item
```

在查询要执行大量非受计算限制的工作（如文件 I/O）的情况下，最好指定比计算机上的内核数要大的并行度。

已排序和未排序的并行查询

在某些查询中，一个查询运算符必须产生保留源序列排序的结果。为此，PLINQ 提供了 AsOrdered 运算符。AsOrdered 不同于 AsSequential。尽管仍并行处理 AsOrdered 序列，但会缓冲和排序它的结果。由于顺序暂留通常涉及额外的工作，因此处理 AsOrdered 序列可能比处理默认 AsUnordered 序列更慢。特定的已排序并行操作是否比操作的顺序版本更快取决于许多因素。

下面的代码示例演示了如何选择使用顺序保留。

```
var evenNums =
    from num in numbers.AsParallel().AsOrdered()
    where num % 2 == 0
    select num;
```

```
Dim evenNums = From num In numbers.AsParallel().AsOrdered()
               Where num Mod 2 = 0
               Select num
```

有关详细信息，请参阅 PLINQ 中的顺序保留。

并行和顺序查询

某些操作要求按顺序提供源数据。必要时，ParallelEnumerable 查询运算符自动还原为顺序模式。对于要求顺序执行的用户定义的查询运算符和用户委托，PLINQ 提供了 AsSequential 方法。使用 AsSequential 时，查询中的所有后续运算符都会顺序执行，直到再次调用 AsParallel。有关详细信息，请参阅如何：合并并行和顺序 LINQ 查询。

合并查询结果的选项

当一个 PLINQ 查询并行执行时，它从每个工作线程得到的结果必须合并回到主线程上，以便由 `foreach` 循环（在 Visual Basic 中为 `For Each`）使用或插入到列表或数组中。例如在某些情况下，指定一个特定类型的合并操作可能会有好处，以更快地开始产生结果。为此，PLINQ 支持 WithMergeOptions 方法和 ParallelMergeOptions 枚举。有关详细信息，请参阅 PLINQ 中的合并选项。

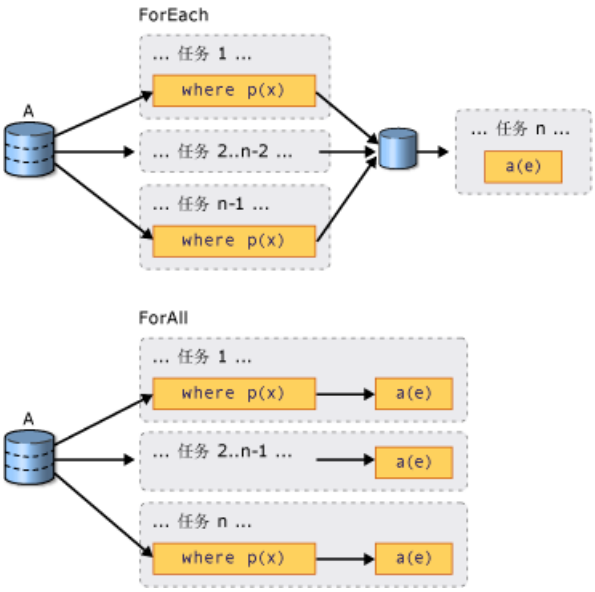
ForAll 运算符

在顺序 LINQ 查询中，执行一直延迟到在 `foreach`（Visual Basic 中为 `For Each`）循环中或通过调用 ToList、ToArray 或 ToDictionary 等方法枚举查询。在 PLINQ 中，还可以使用 `foreach` 执行查询以及循环访问结果。但是，`foreach` 本身不会并行运行，因此，它要求将所有并行任务的输出合并回该循环正在上面运行的线程中。在 PLINQ 中，在必须保留查询结果的最终排序，以及以按串行方式处理结果时，例如当为每个元素调用 `Console.WriteLine` 时，则可以使用 `foreach`。为了在无需顺序暂留以及可自行并行处理结果时更快地执行查询，请使用 ForAll 方法执行 PLINQ 查询。ForAll 不执行最终的这一合并步骤。下面的代码示例说明如何使用 ForAll 方法。此处使用 System.Collections.Concurrent.ConcurrentBag<T> 是因为它已优化，可以同时添加多个线程，而无需尝试移除任何项。

```
var nums = Enumerable.Range(10, 10000);
var query =
    from num in nums.AsParallel()
    where num % 10 == 0
    select num;

// Process the results as each thread completes
// and add them to a System.Collections.Concurrent.ConcurrentBag(Of Int)
// which can safely accept concurrent add operations
query.ForAll(e => concurrentBag.Add(Compute(e)));
```

```
Dim nums = Enumerable.Range(10, 10000)
Dim query = From num In nums.AsParallel()
            Where num Mod 10 = 0
            Select num

' Process the results as each thread completes
' and add them to a System.Collections.Concurrent.ConcurrentBag(Of Int)
' which can safely accept concurrent add operations
query.ForAll(Sub(e) concurrentBag.Add(Compute(e)))
```

下图展示了 `foreach` 与 ForAll 在查询执行方面的区别。



取消

PLINQ 在 .NET 中与取消类型集成在一起。(有关详细信息，请参阅托管线程中的取消。)因此，与顺序 LINQ to Objects 查询不同，可以取消 PLINQ 查询。若要创建可取消 PLINQ 查询，请在查询中使用 WithCancellation 运算符，并提供 CancellationToken 实例作为参数。如果令牌上的 IsCancellationRequested 属性设置为 true，PLINQ 就会注意到它，停止处理所有线程并抛出 OperationCanceledException。

在设置取消标记后，PLINQ 查询还可能会继续处理一些元素。

为了提高响应速度，还可以在长时间运行的用户委托中响应取消请求。有关详细信息，请参阅如何：取消 PLINQ 查询。

异常

当一个 PLINQ 查询执行时，可能会同时从不同的线程引发多个异常。此外，处理异常的代码可能与引发异常的代码处于不同的线程上。PLINQ 使用 AggregateException 类型封装查询抛出的所有异常，并将这些异常封送回调用线程。在调用线程上，只需要一个 try-catch 块。不过，可以循环访问在 AggregateException 中封装的所有异常，并捕获任何可以安全恢复的异常。在极少数情况下，可能会抛出未在 AggregateException 中包装的一些异常，ThreadAbortException 也没有进行包装。

如果允许异常向上冒泡回到联接线程，则查询也许可以在引发异常后继续处理一些项。

有关详细信息，请参阅如何：处理 PLINQ 查询中的异常。

自定义分区程序

在某些情况下，可以通过编写利用源数据的某些特征的自定义分区程序来提高查询性能。在查询中，自定义分区程序本身是被查询的可枚举对象。

```
int[] arr = new int[9999];
Partitioner<int> partitioner = new MyArrayPartitioner<int>(arr);
var query = partitioner.AsParallel().Select(SomeFunction);
```

```
Dim arr(10000) As Integer
Dim partitioner As Partitioner(Of Integer) = New MyArrayPartitioner(Of Integer)(arr)
Dim query = partitioner.AsParallel().Select(Function(x) SomeFunction(x))
```

PLINQ 支持固定数量的分区(尽管在运行时期间为了负载均衡可能会将数据重新动态分配到这些分区)。For 和 ForEach 仅支持动态分区。也就是说，分区数在运行时发生变化。有关详细信息，请参阅 PLINQ 和 TPL 的自定义分区程序。

衡量 PLINQ 性能

在很多情况下，可以并行化查询，但是设置并行查询的开销可能会超出获得的性能收益。如果查询不执行大量的计算，或者如果数据源较小，则 PLINQ 查询的速度可能比顺序 LINQ to Objects 查询的速度慢。可以在 Visual Studio Team Server 中使用并行性能分析器比较各种查询的性能，查找处理瓶颈，以及确定查询是并行运行还是按顺序运行。有关详细信息，请参阅并发可视化工具 SDK和如何：衡量 PLINQ 查询性能。

请参阅

- 并行 LINQ (PLINQ)
- 了解 PLINQ 中的加速

了解 PLINQ 中的加速

2021/11/16 •

PLINQ 的主要用途是，在多核计算机上并行执行查询委托，以加速执行 LINQ to Objects 查询。如果单独处理源集合中的每个元素，且各个代理之间不涉及共享状态，PLINQ 的性能最佳。此类操作在 LINQ to Objects 和 PLINQ 中很常见，通常称为“适合并行”，因为它们可以轻松适应计划多个线程的工作。不过，并非所有查询完全都由适合并行操作组成；在大多数情况下，查询涉及一些无法并行执行或减慢并行执行的运算符。即使查询完全都由适合并行组成，PLINQ 仍必须对数据源进行分区，并计划线程工作，通常还需要在查询完成时合并结果。所有这些操作都增加了并行执行的计算成本；增加并行执行而产生的这些成本称为“开销”。为了实现 PLINQ 查询的最佳性能，目标是最大限度地增加适合并行执行的部分，并尽量减少需要开销的部分。本文有助于确保编写的 PLINQ 查询尽可能高效，且仍能产生正确结果。

PLINQ 查询性能的影响因素

下面各部分列出了并行查询性能的一些最重要的影响因素。这些都是一般性说明，本身并不足以用于在所有情况下预测查询性能。和以往一样，请务必在具有一系列代表性配置和负载的计算机上度量特定查询的实际性能。

1. 整体工作的计算成本。

   为了实现加速，PLINQ 查询必须有足够多的适合并行操作来抵消开销。工作可以表示为每个委托的计算成本乘以源集合中的元素数量。假设操作可以并行执行，它的计算成本越高，加速的机会就越大。例如，如果函数的执行时间为 1 毫秒，那么超过 1000 个元素的顺序查询需要 1 秒的时间才能执行此操作，而在四核计算机上，并行查询可能只需要 250 毫秒就能完成。这就产生 750 毫秒的加速。如果函数执行每个元素需要 1 秒，就会产生 750 秒的加速。如果委托成本很高，PLINQ 可能会让速度显著提升，前提是源集合中只有几项。相反，包含最简单的委托的小型源集合通常不适合执行 PLINQ。

   在下面的示例中，queryA 可能很适合执行 PLINQ，前提是它的 Select 函数涉及很多工作。queryB 可能不适合执行 PLINQ，因为 Select 语句中没有足够多的工作，并行开销会抵消大部分或全部加速。

   ```
   Dim queryA = From num In numberList.AsParallel()
                Select ExpensiveFunction(num); 'good for PLINQ

   Dim queryB = From num In numberList.AsParallel()
                Where num Mod 2 > 0
                Select num; 'not as good for PLINQ
   ```

   ```
   var queryA = from num in numberList.AsParallel()
                select ExpensiveFunction(num); //good for PLINQ

   var queryB = from num in numberList.AsParallel()
                where num % 2 > 0
                select num; //not as good for PLINQ
   ```

2. 系统上的逻辑内核数量（并行度）。

   这一点是上一部分的必然结果，在具有更多内核的计算机上，适合并行查询运行得更快，这是因为可以在更多并发线程之间划分工作。加速总量取决于查询整体工作的并行度百分比。不过，不要认为所有查询在八核计算机上的运行速度都是在四核计算机上的两倍。优化查询以实现最佳性能时，请务必在具有不同数量内核的计算机上度量实际结果。这一点与第 1 点相关：需要更大的数据集，才能利用更多的计算资源。

3. 操作的数量和种类。

如果有必要维护源序列中的元素顺序，PLINQ 提供 AsOrdered 运算符。虽然排序有相关成本，但此成本通常还算低。GroupBy 和 Join 操作同样也会产生开销。如果允许按任意顺序处理源集合中的元素，并在准备就绪后立即将它们传递给下一个运算符，PLINQ 的性能最佳。有关详细信息，请参阅 PLINQ 中的顺序保留。

4. 查询执行形式。

   若要通过调用 ToArray 或 ToList 存储查询结果，所有并行线程的结果都必须合并到一个数据结构中。这就涉及不可避免的计算成本。同样，如果使用 foreach（Visual Basic 中的 For Each）循环来循环访问结果，工作线程的结果必须串行化到枚举器线程。不过，如果只想根据每个线程的结果执行某操作，可以使用 ForAll 方法对多个线程执行此操作。

5. 合并选项类型。

   PLINQ 可以配置为缓冲输出并在生成整个结果集后分块区生成或一次性全部生成，也可以配置为在各个结果生成时流式传输它们。前一个导致总体执行时间减少，后一个导致所生成元素之间的延迟减少。尽管合并选项不一定会对总体查询性能造成重大影响，但它们可能会影响感知性能，因为它们控制用户在看到结果前必须等待的时间。有关详细信息，请参阅 PLINQ 中的合并选项。

6. 分区种类。

   在某些情况下，对可索引源集合执行 PLINQ 查询可能会导致工作负载不平衡。如果发生这种情况，可以创建自定义分区程序，从而提升查询性能。有关详细信息，请参阅 PLINQ 和 TPL 的自定义分区程序。

如果 PLINQ 选择顺序模式

PLINQ 始终都会尝试至少像顺序运行查询一样快地执行查询。虽然 PLINQ 没有考虑用户委托的计算成本或输入源大小，但它确实会查找特定查询“形状”。具体来说，它会查找通常会减慢查询在并行模式下的执行速度的查询运算符或运算符组合。如果找到此类形状，PLINQ 默认会回退到顺序模式。

不过，在度量特定查询的性能后，可以确定它在并行模式下的实际运行速度更快。在这种情况下，可以通过 WithExecutionMode 方法使用 ParallelExecutionMode.ForceParallelism 标志来指示 PLINQ 并行执行查询。有关详细信息，请参阅如何：在 PLINQ 中指定执行模式。

下面列出了 PLINQ 在顺序模式下默认执行的查询形状：

- 在删除或重新排列了原始索引的排序或筛选运算符后面，包含 Select、已编制索引 Where、已编制索引 SelectMany 或 ElementAt 子句的查询。
- 包含 Take、TakeWhile、Skip、SkipWhile 运算符且源序列中的索引不是原始顺序的查询。
- 包含 Zip 或 SequenceEquals 的查询，除非其中一个数据源具有按原始顺序排列的索引，并且另一个数据源是可索引的（即，数组或 IList(T)）。
- 包含 Concat 的查询，除非应用于可索引的数据源。
- 包含 Reverse 的查询，除非应用于可索引的数据源。

请参阅

- 并行 LINQ (PLINQ)

PLINQ 中的顺序保留

2021/11/16 •

在 PLINQ 中，目标是在保持正确性的同时，最大限度地提升性能。虽然查询应尽可能快地运行，但仍应生成正确结果。在某些情况下，为了满足正确性要求，必须暂留源序列的顺序；不过，顺序暂留的计算成本可能非常高。因此，默认情况下，PLINQ 不暂留源序列的顺序。在这方面，PLINQ 类似于 LINQ to SQL，但与确实暂留顺序的 LINQ to Objects 不同。

若要替代默认行为，可以对源序列使用 AsOrdered 运算符，启用顺序暂留。稍后，可以使用 AsUnordered 方法，在查询中禁用顺序暂留。使用这两种方法时，查询的处理依据为，确定是并行执行还是顺序执行查询的启发。有关详细信息，请参阅了解 PLINQ 中的加速。

下面的示例展示了无序并行查询，以筛选符合条件的所有元素，而完全不尝试对结果进行排序。

```
var cityQuery =
    (from city in cities.AsParallel()
     where city.Population > 10000
     select city).Take(1000);
```

```
Dim cityQuery = From city In cities.AsParallel()
                Where city.Population > 10000
                Take (1000)
```

此查询不一定会生成源序列中符合条件的前 1000 个城市，而会生成符合条件的 1000 个城市中的一部分。PLINQ 查询运算符将源序列划分为多个子序列，并将它们作为并发任务进行处理。如果未指定顺序暂留，每个分区生成的结果以任意顺序传递到查询的下一个阶段。此外，在继续处理其余元素前，分区可能还会生成一部分结果。因此，顺序可能每次都不同。应用无法对此进行控制，因为它取决于操作系统如何将线程排入计划。

下面的示例对源序列使用 AsOrdered 运算符，以替代默认行为。这样可确保 Take 方法返回源序列中符合条件的前 1000 个城市。

```
var orderedCities =
    (from city in cities.AsParallel().AsOrdered()
     where city.Population > 10000
     select city).Take(1000);
```

```
Dim orderedCities = From city In cities.AsParallel().AsOrdered()
                    Where city.Population > 10000
                    Take (1000)
```

不过，此查询可能不会像无序版本那样快速运行，因为它必须跟踪整个分区中的原始顺序，并且在合并时确保顺序是一致的。因此，建议仅在需要时，才将 AsOrdered用于需要它的查询部分。如果不再需要顺序暂留，请使用 AsUnordered 禁用它。为此，下面的示例撰写了两个查询。

```
var orderedCities2 =
    (from city in cities.AsParallel().AsOrdered()
     where city.Population > 10000
     select city).Take(1000);

var finalResult =
    from city in orderedCities2.AsUnordered()
    join p in people.AsParallel()
    on city.Name equals p.CityName into details
    from c in details
    select new
    {
        city.Name,
        Pop = city.Population,
        c.Mayor
    };

foreach (var city in finalResult) { /*...*/ }
```

```
Dim orderedCities2 = From city In cities.AsParallel().AsOrdered()
                     Where city.Population > 10000
                     Select city
                     Take (1000)

Dim finalResult = From city In orderedCities2.AsUnordered()
                  Join p In people.AsParallel() On city.Name Equals p.CityName
                  Select New With {.Name = city.Name, .Pop = city.Population, .Mayor = city.Mayor}

For Each city In finalResult
    Console.WriteLine(city.Name & ":" & city.Pop & ":" & city.Mayor)
Next
```

请注意，PLINQ 暂留查询其余部分的顺序强制施加运算符生成的序列顺序。也就是说，OrderBy 和 ThenBy 等运算符被视为后跟 AsOrdered 调用。

查询运算符和顺序

下面的查询运算符将顺序暂留引入查询中的所有后续操作，或一直运行到 AsUnordered 获得调用：

- OrderBy
- OrderByDescending
- ThenBy
- ThenByDescending

在某些情况下，下面的 PLINQ 查询运算符可能需要有序的源序列，才能生成正确结果：

- Reverse
- SequenceEqual
- TakeWhile
- SkipWhile
- Zip

一些 PLINQ 查询运算符的行为因源序列是有序还是无序而异。下表列出了这些运算符。

ttt	ttttttttttt	ttttttttttt
Aggregate	非关联或非交换操作的非确定性输出	非关联或非交换操作的非确定性输出
All	不适用	不适用
Any	不适用	不适用
AsEnumerable	不适用	不适用
Average	非关联或非交换操作的非确定性输出	非关联或非交换操作的非确定性输出
Cast	有序结果	无序结果
Concat	有序结果	无序结果
Count	不适用	不适用
DefaultIfEmpty	不适用	不适用
Distinct	有序结果	无序结果
ElementAt	返回指定元素	任意元素
ElementAtOrDefault	返回指定元素	任意元素
Except	无序结果	无序结果
First	返回指定元素	任意元素
FirstOrDefault	返回指定元素	任意元素
ForAll	非确定性并行执行	非确定性并行执行
GroupBy	有序结果	无序结果
GroupJoin	有序结果	无序结果
Intersect	有序结果	无序结果
Join	有序结果	无序结果
Last	返回指定元素	任意元素
LastOrDefault	返回指定元素	任意元素
LongCount	不适用	不适用
Min	不适用	不适用
OrderBy	对序列重新排序	开始新的有序部分

[illegible]	[illegible]	[illegible]
OrderByDescending	对序列重新排序	开始新的有序部分
Range	暂无(默认值与 AsParallel 相同)	不适用
Repeat	暂无(默认值与 AsParallel 相同)	不适用
Reverse	反转	不执行任何操作
Select	有序结果	无序结果
Select(已编入索引)	有序结果	无序结果。
SelectMany	有序结果。	无序结果
SelectMany(已编入索引)	有序结果。	无序结果。
SequenceEqual	有序比较	无序比较
Single	不适用	不适用
SingleOrDefault	不适用	不适用
Skip	跳过第一个 n 元素	跳过所有 n 元素
SkipWhile	有序结果。	非确定性。以当前任意顺序执行 SkipWhile
Sum	非关联或非交换操作的非确定性输出	非关联或非交换操作的非确定性输出
Take	获取前 `n` 个元素	获取任意 `n` 个元素
TakeWhile	有序结果	非确定性。以当前任意顺序执行 TakeWhile
ThenBy	补充 `OrderBy`	补充 `OrderBy`
ThenByDescending	补充 `OrderBy`	补充 `OrderBy`
ToArray	有序结果	无序结果
ToDictionary	不适用	不适用
ToList	有序结果	无序结果
ToLookup	有序结果	无序结果
Union	有序结果	无序结果
Where	有序结果	无序结果

[illegible]	[illegible]	[illegible]
Where(已编入索引)	有序结果	无序结果
Zip	有序结果	无序结果

无序结果并不是主动随机无序;就是没有应用任何特殊顺序逻辑。在某些情况下，无序查询可能会暂留源序列的顺序。对于使用已编入索引的 Select 运算符的查询，PLINQ 保证输出元素按索引升序排列，但不保证向元素分配哪些索引。

另请参阅

- 并行 LINQ (PLINQ)
- 并行编程

PLINQ 中的合并选项

2021/11/16 •

如果并行执行查询，PLINQ 对源序列进行分区，以便多个线程能够并发处理不同部分，通常是在不同的线程中。如果要在一个线程（例如，`foreach`（Visual Basic 中的 `For Each`）循环）中使用结果，必须将每个线程的结果合并回一个序列中。PLINQ 执行的合并类型具体视查询中的运算符而定。例如，对结果强制施加新顺序的运算符必须缓冲所有线程中的全部元素。从使用线程（以及应用用户）的角度来看，完全缓冲查询可能会运行很长时间，才能生成第一个结果。默认情况下，其他运算符进行部分缓冲，并分批生成结果。默认不缓冲的一个运算符是 ForAll。它会立即生成所有线程中的所有元素。

使用 WithMergeOptions 方法（如下面的示例所示），可以向 PLINQ 提供提示，指明要执行的合并类型。

```
var scanLines = from n in nums.AsParallel()
                    .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                where n % 2 == 0
                select ExpensiveFunc(n);
```

```
Dim scanlines = From n In nums.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
                Where n Mod 2 = 0
                Select ExpensiveFunc(n)
```

有关完整的示例，请参阅如何：在 PLINQ 中指定合并选项。

如果特定查询无法支持请求执行的选项，将会直接忽略此选项。大多数情况下，无需为 PLINQ 查询指定合并选项。不过，在某些情况下，通过测试和度量，可以发现查询在非默认模式下执行效果最佳。这种做法的常见用途是，强制区块合并运算符流式传输结果，以提升用户界面的响应速度。

ParallelMergeOptions

ParallelMergeOptions 枚举包括以下选项，用于针对受支持的查询形状，指定在一个线程上使用结果时如何生成查询的最终输出：

- `Not Buffered`

  使用 NotBuffered 选项，每个处理过的元素一生成就会由各个线程返回。这种行为类同于“流式传输”输出。如果查询中有 AsOrdered 运算符，`NotBuffered` 暂留源元素的顺序。虽然 `NotBuffered` 一有结果就立即开始生成，但生成所有结果的总时间可能仍长于使用其他合并选项之一的用时。

- `Auto Buffered`

  通过 AutoBuffered 选项，查询将元素收集到缓冲区中，并定期一次性将所有缓冲内容生成到使用线程中。这类同于分“区块”生成源数据，而不是使用 `NotBuffered` 的“流式传输”行为。`AutoBuffered` 可能需要花费比 `NotBuffered` 更长的时间，才能在使用线程中生成第一个元素。缓冲大小和确切的生成行为不可配置，并因与查询相关的各种因素而异。

- `FullyBuffered`

  使用 FullyBuffered 选项，可以先缓冲整个查询的输出，再生成任何元素。使用此选项时，虽然在使用线程中生成第一个元素的耗时更长，但完整结果的生成时间可能仍短于使用其他选项的用时。

支持合并选项的查询运算符

下表列出了支持所有合并选项模式的运算符（受指定的限制约束）。

AsEnumerable	None
Cast	None
Concat	只包含 Array 或 List 源的无序查询。
DefaultIfEmpty	None
OfType	None
Reverse	只包含 Array 或 List 源的无序查询。
Select	None
SelectMany	None
Skip	None
Take	None
Where	None

其他所有 PLINQ 查询运算符可能会忽略用户提供的合并选项。一些查询运算符（例如，Reverse 和 OrderBy）在生成并重新排序所有元素之前，无法生成任何元素。因此，如果在还包含 Reverse 等运算符的查询中使用 ParallelMergeOptions，除非运算符生成了结果，否则将不会在查询中应用合并行为。

一些运算符处理合并选项的能力，取决于源序列的类型，以及之前是否在查询中使用过 AsOrdered 运算符。ForAll 始终为 NotBuffered；它立即生成元素。OrderBy 始终为 FullyBuffered；它必须先对整个列表进行排序，再生成元素。

请参阅

- 并行 LINQ (PLINQ)
- 如何：在 PLINQ 中指定合并选项

PLINQ 的潜在缺陷

2021/11/16 •

在许多情况下，与顺序 LINQ to Objects 查询相比，PLINQ 可以显著提升性能。不过，并行执行查询增加了工作复杂性，可能会导致在顺序代码中不常见或根本不会遇到的问题。本主题列出了一些在编写 PLINQ 查询时要避免的做法。

不要假定并行速度总是更快

并行执行有时会导致 PLINQ 查询比相当的 LINQ to Objects 慢。基本原则是，源元素很少且用户委托速度快的查询不太可能会加速很多。不过，由于影响性能的因素有很多，因此建议在决定是否使用 PLINQ 前，先度量一下实际结果。有关详细信息，请参阅了解 PLINQ 中的加速。

不要写入共享内存位置

在顺序代码中，从静态变量或类字段中读取或写入静态变量或类字段的情况很常见。但是，每当多个线程同时访问此类变量时，则很有可能会出现争用条件。即使可以使用锁来同步对变量的访问，但同步开销可能会对性能造成损害。因此，建议尽量避免在 PLINQ 查询中访问共享状态，或至少限制对共享状态的访问。

不要过度并行化

使用 `AsParallel` 方法会产生对源集合进行分区和同步工作线程的开销成本。计算机上的处理器数量进一步限制了并行化的优点。仅在一个处理器上运行多个受计算限制的线程时，速度并不会得到提升。因此，必须要小心，不要过度并行执行查询。

过度并行执行的最常见方案是嵌套查询，如下面的代码片段所示。

```
var q = from cust in customers.AsParallel()
        from order in cust.Orders.AsParallel()
        where order.OrderDate > date
        select new { cust, order };
```

```
Dim q = From cust In customers.AsParallel()
        From order In cust.Orders.AsParallel()
        Where order.OrderDate > aDate
        Select New With {cust, order}
```

在这种情况下，最好仅并行执行外部数据源（“客户”），除非满足下面的一个或多个条件：

- 内部数据源 (cust.Orders) 已知非常长。
- 正在对每个订单执行开销极大的计算。（示例中所示的操作开销不大。）
- 已知目标系统具有足够的处理器来处理通过对 `cust.Orders` 上的查询进行并行化所产生的线程数。

在所有情况下，确定最佳查询形式的最好方法是进行测试和测量。有关详细信息，请参阅如何：衡量 PLINQ 查询性能。

避免调用非线程安全方法

如果通过 PLINQ 查询对非线程安全实例方法执行写入操作，可能会导致数据损坏，程序可能会或可能不会检测

不到它。还可能会导致异常。在下面的示例中，多个线程尝试同时调用 `FileStream.Write` 方法，类并不支持这样做。

```
Dim fs As FileStream = File.OpenWrite(…)
a.AsParallel().Where(...).OrderBy(...).Select(...).ForAll(Sub(x) fs.Write(x))
```

```
FileStream fs = File.OpenWrite(...);
a.AsParallel().Where(...).OrderBy(...).Select(...).ForAll(x => fs.Write(x));
```

限制对线程安全方法的调用

.NET 中的大多数静态方法是线程安全的，并且可以同时从多个线程中调用。但是，即使在这些情况下，所涉及到的同步也可能会导致查询速度大幅度下降。

> **NOTE**
> 可以自行对此进行测试，具体方法是在查询中插入一些 WriteLine 调用。文档示例中使用的此方法只为了方便本文演示，请勿在 PLINQ 查询中使用它。

避免不必要的排序操作

并行执行查询时，PLINQ 将源序列划分为多个分区，可以在多个线程上并发运行。默认情况下，分区的处理顺序和结果顺序是不可预测的（`OrderBy` 等运算符除外）。虽然可以指示 PLINQ 暂留任何源序列的顺序，但这会对性能产生不利影响。最佳做法是，尽量将查询的结构设计为不依赖顺序暂留。有关详细信息，请参阅 PLINQ 中的顺序保留。

尽量首选 ForAll（而不是 ForEach）

尽管 PLINQ 对多个线程执行查询，但如果在 `foreach` 循环（Visual Basic 中的 `For Each`）中使用结果，查询结果必须合并回一个线程，并由枚举器串行访问。在某些情况下，这是不可避免的；不过，应尽量使用 `ForAll` 方法，让每个线程输出自己的结果。例如，通过对线程安全集合（如 System.Collections.Concurrent.ConcurrentBag<T>）执行写入操作。

此问题还适用于 Parallel.ForEach。换言之，强烈建议 `source.AsParallel().Where().ForAll(...)` 首选 `Parallel.ForEach(source.AsParallel().Where(), ...)`。

注意线程关联问题

某些技术（例如，单线程单元 (STA) 组件的 COM 互操作性、Windows 窗体以及 Windows Presentation Foundation (WPF)）具有要求代码在特定线程上运行的线程关联限制。例如，在 Windows 窗体和 WPF 中，只能在创建控件的线程上访问该控件。如果尝试在 PLINQ 查询中访问 Windows 窗体控件的共享状态，且在调试器中运行查询，就会导致异常抛出。（可以禁用此设置。）不过，如果对 UI 线程使用查询，可以通过枚举查询结果的 `foreach` 循环来访问控件，因为代码只对一个线程执行。

不要假定 ForEach、For 和 ForAll 的迭代始终并行执行

请务必注意，Parallel.For、Parallel.ForEach 或 ForAll 循环中的各个迭代可能会（但不需要）并行执行。因此，应避免编写任何依赖于迭代并行执行的正确性或依赖于按任何特定顺序执行迭代的代码。

例如，此代码有可能会死锁：

```
Dim mre = New ManualResetEventSlim()
Enumerable.Range(0, Environment.ProcessorCount * 100).AsParallel().ForAll(Sub(j)
   If j = Environment.ProcessorCount Then
       Console.WriteLine("Set on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j)
       mre.Set()
   Else
       Console.WriteLine("Waiting on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j)
       mre.Wait()
   End If
End Sub) ' deadlocks
```

```
ManualResetEventSlim mre = new ManualResetEventSlim();
Enumerable.Range(0, Environment.ProcessorCount * 100).AsParallel().ForAll((j) =>
{
    if (j == Environment.ProcessorCount)
    {
        Console.WriteLine("Set on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j);
        mre.Set();
    }
    else
    {
        Console.WriteLine("Waiting on {0} with value of {1}", Thread.CurrentThread.ManagedThreadId, j);
        mre.Wait();
    }
}); //deadlocks
```

在此示例中，一个迭代设置一个事件，而所有的其他迭代则等待该事件。在设置事件的迭代完成之前，任何等待迭代均无法完成。但是，在设置事件的迭代有机会执行之前，等待迭代可能会阻止用于执行并行循环的所有线程。这将导致死锁 – 设置事件的迭代将永不会执行，并且等待迭代将永远不会醒来。

具体而言，并行循环的一个迭代绝不应该等待循环的另一个迭代来继续执行。如果并行循环决定按相反的顺序安排迭代，则会发生死锁。

请参阅

- 并行 LINQ (PLINQ)

如何：创建并执行简单的 PLINQ 查询

2021/11/16 •

本文中的示例演示如何通过对源序列使用 ParallelEnumerable.AsParallel 扩展方法来创建一个简单的并行语言集成查询 (LINQ) 查询，并使用 ParallelEnumerable.ForAll 方法执行该查询。

> **NOTE**
> 本文档使用 lambda 表达式在 PLINQ 中定义委托。如果不熟悉 C# 或 Visual Basic 中的 lambda 表达式，请参阅 PLINQ 和 TPL 中的 Lambda 表达式。

示例

```
using System;
using System.Linq;

class ExampleForAll
{
    public static void Main()
    {
        var source = Enumerable.Range(100, 20000);

        // Result sequence might be out of order.
        var parallelQuery =
            from num in source.AsParallel()
            where num % 10 == 0
            select num;

        // Process result sequence in parallel
        parallelQuery.ForAll((e) => DoSomething(e));

        // Or use foreach to merge results first.
        foreach (var n in parallelQuery)
        {
            Console.WriteLine(n);
        }

        // You can also use ToArray, ToList, etc as with LINQ to Objects.
        var parallelQuery2 =
            (from num in source.AsParallel()
             where num % 10 == 0
             select num).ToArray();

        // Method syntax is also supported
        var parallelQuery3 =
            source.AsParallel()
                .Where(n => n % 10 == 0)
                .Select(n => n);

        Console.WriteLine("\nPress any key to exit...");
        Console.ReadLine();
    }

    static void DoSomething(int _) { }
}
```

```
Public Class Program
    Public Shared Sub Main()
        Dim source = Enumerable.Range(100, 20000)

        ' Result sequence might be out of order.
        Dim parallelQuery = From num In source.AsParallel()
                            Where num Mod 10 = 0
                            Select num

        ' Process result sequence in parallel
        parallelQuery.ForAll(Sub(e)
                                 DoSomething(e)
                             End Sub)

        ' Or use For Each to merge results first
        ' as in this example, Where results must
        ' be serialized sequentially through static Console method.
        For Each n In parallelQuery
            Console.Write("{0} ", n)
        Next

        ' You can also use ToArray, ToList, etc, as with LINQ to Objects.
        Dim parallelQuery2 = (From num In source.AsParallel()
                              Where num Mod 10 = 0
                              Select num).ToArray()

        ' Method syntax is also supported
        Dim parallelQuery3 =
            source.AsParallel().Where(Function(n)
                                          Return (n Mod 10) = 0
                                      End Function).Select(Function(n)
                                                               Return n
                                                           End Function)

        For Each i As Integer In parallelQuery3
            Console.Write($"{i} ")
        Next


        Console.WriteLine()
        Console.WriteLine("Press any key to exit...")
        Console.ReadLine()
    End Sub

    ' A toy function to demonstrate syntax. Typically you need a more
    ' computationally expensive method to see speedup over sequential queries.
    Shared Sub DoSomething(ByVal i As Integer)
        Console.Write($"{Math.Sqrt(i):###.##} ")
    End Sub
End Class
```

此示例演示用于在结果序列的排序不重要的情况下创建和执行任何并行 LINQ 查询的基本模式. 未排序的查询通常比已排序的查询快。查询将源分区为多个任务，这些任务将在多个线程上异步执行。每个任务的完成顺序不仅取决于处理分区中的元素所涉及的工作量，还取决于诸如操作系统如何调度每个线程之类的外部因素。本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。有关如何在查询中保留元素排序的详细信息，请参阅如何：在 PLINQ 查询中控制排序。

请参阅

- 并行 LINQ (PLINQ)

如何：在 PLINQ 查询中控制排序

2021/11/16 •

这些示例展示了如何使用 AsOrdered 扩展方法控制 PLINQ 查询中的顺序。

> **WARNING**
> 这些示例主要用于演示用法，可能会或可能不会比相当的顺序 LINQ to Objects 查询快。

示例

下面的示例暂留了源序列的顺序。有时，这样做是有必要的；例如，一些查询运算符需要有序的源序列，才能生成正确结果。

```
var source = Enumerable.Range(9, 10000);

// Source is ordered; let's preserve it.
var parallelQuery =
    from num in source.AsParallel().AsOrdered()
    where num % 3 == 0
    select num;

// Use foreach to preserve order at execution time.
foreach (var item in parallelQuery)
{
    Console.Write($"{item} ");
}

// Some operators expect an ordered source sequence.
var lowValues = parallelQuery.Take(10);
```

```
Sub OrderedQuery()

    Dim source = Enumerable.Range(9, 10000)

    ' Source is ordered let's preserve it.
    Dim parallelQuery = From num In source.AsParallel().AsOrdered()
                        Where num Mod 3 = 0
                        Select num

    ' Use For Each to preserve order at execution time.
    For Each item In parallelQuery
        Console.Write("{0} ", item)
    Next

    ' Some operators expect an ordered source sequence.
    Dim lowValues = parallelQuery.Take(10)

End Sub
```

示例

下面的示例展示了一些可能要求源序列应为有序的查询运算符。虽然这些运算符可以处理无序序列，但可能会生成意外结果。

```
// Paste into PLINQDataSample class.
static void SimpleOrdering()
{

    var customers = GetCustomers();

    // Take the first 20, preserving the original order
    var firstTwentyCustomers = customers
                               .AsParallel()
                               .AsOrdered()
                               .Take(20);

    foreach (var c in firstTwentyCustomers)
        Console.Write("{0} ", c.CustomerID);

    // All elements in reverse order.
    var reverseOrder = customers
                       .AsParallel()
                       .AsOrdered()
                       .Reverse();

    foreach (var v in reverseOrder)
        Console.Write("{0} ", v.CustomerID);

    // Get the element at a specified index.
    var cust = customers.AsParallel()
                       .AsOrdered()
                       .ElementAt(48);

    Console.WriteLine("Element #48 is: {0}", cust.CustomerID);
}
```

```
' Paste into PLINQDataSample class
Shared Sub SimpleOrdering()
    Dim customers As List(Of Customer) = GetCustomers().ToList()

    ' Take the first 20, preserving the original order

    Dim firstTwentyCustomers = customers _
                               .AsParallel() _
                               .AsOrdered() _
                               .Take(20)

    Console.WriteLine("Take the first 20 in original order")
    For Each c As Customer In firstTwentyCustomers
        Console.Write(c.CustomerID & " ")
    Next

    ' All elements in reverse order.
    Dim reverseOrder = customers _
                       .AsParallel() _
                       .AsOrdered() _
                       .Reverse()

    Console.WriteLine(vbCrLf & "Take all elements in reverse order")
    For Each c As Customer In reverseOrder
        Console.Write("{0} ", c.CustomerID)
    Next
    ' Get the element at a specified index.
    Dim cust = customers.AsParallel() _
                       .AsOrdered() _
                       .ElementAt(48)

    Console.WriteLine("Element #48 is: " & cust.CustomerID)

End Sub
```

若要运行此方法，请将它粘贴到 PLINQ 数据样本项目的 PLINQDataSample 类中，再按 F5。

示例

下面的示例展示了如何暂留查询第一部分的顺序，再删除顺序以提升 join 子句的性能，并对最终结果序列重新应用顺序。

```
// Paste into PLINQDataSample class.
static void OrderedThenUnordered()
{

    var orders = GetOrders();
    var orderDetails = GetOrderDetails();

    var q2 = orders.AsParallel()
       .Where(o => o.OrderDate < DateTime.Parse("07/04/1997"))
       .Select(o => o)
       .OrderBy(o => o.CustomerID) // Preserve original ordering for Take operation.
       .Take(20)
       .AsUnordered()  // Remove ordering constraint to make join faster.
       .Join(
              orderDetails.AsParallel(),
              ord => ord.OrderID,
              od => od.OrderID,
              (ord, od) =>
              new
              {
                  ID = ord.OrderID,
                  Customer = ord.CustomerID,
                  Product = od.ProductID
              }
             )
       .OrderBy(i => i.Product); // Apply new ordering to final result sequence.

    foreach (var v in q2)
        Console.WriteLine("{0} {1} {2}", v.ID, v.Customer, v.Product);
}
```

```
' Paste into PLINQDataSample class
Sub OrderedThenUnordered()
    Dim Orders As IEnumerable(Of Order) = GetOrders()
    Dim orderDetails As IEnumerable(Of OrderDetail) = GetOrderDetails()

    ' Sometimes it's easier to create a query
    ' by composing two subqueries
    Dim query1 = From ord In Orders.AsParallel()
                 Where ord.OrderDate < DateTime.Parse("07/04/1997")
                 Select ord
                 Order By ord.CustomerID
                 Take 20

    Dim query2 = From ord In query1.AsUnordered()
                 Join od In orderDetails.AsParallel() On ord.OrderID Equals od.OrderID
                 Order By od.ProductID
                 Select New With {ord.OrderID, ord.CustomerID, od.ProductID}


    For Each item In query2
        Console.WriteLine("{0} {1} {2}", item.OrderID, item.CustomerID, item.ProductID)
    Next
End Sub
```

若要运行此方法，请将它粘贴到 PLINQ 数据样本项目的 PLINQDataSample 类中，再按 F5。

另请参阅

- ParallelEnumerable
- 并行 LINQ (PLINQ)

如何：合并并行和顺序 LINQ 查询

2021/11/16 •

此示例展示了如何使用 AsSequential 方法，指示 PLINQ 顺序处理查询中的所有后续运算符。尽管顺序处理通常比并行处理慢，但有时却是生成正确结果的必要条件。

> **NOTE**
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

下面的示例展示了一种需要 AsSequential 的情况，即暂留在旧查询子句中建立的顺序。

```
// Paste into PLINQDataSample class.
static void SequentialDemo()
{
    var orders = GetOrders();
    var query = (from order in orders.AsParallel()
                 orderby order.OrderID
                 select new
                 {
                     order.OrderID,
                     OrderedOn = order.OrderDate,
                     ShippedOn = order.ShippedDate
                 })
                 .AsSequential().Take(5);
}
```

```
' Paste into PLINQDataSample class
Shared Sub SequentialDemo()

    Dim orders = GetOrders()
    Dim query = From ord In orders.AsParallel()
                Order By ord.OrderID
                Select New With
                {
                    ord.OrderID,
                    ord.OrderDate,
                    ord.ShippedDate
                }

    Dim query2 = query.AsSequential().Take(5)

    For Each item In query2
        Console.WriteLine("{0}, {1}, {2}", item.OrderDate, item.OrderID, item.ShippedDate)
    Next
End Sub
```

编译代码

若要编译并运行此代码，请将它粘贴到 PLINQ 数据样本项目中，添加用于从 `Main` 调用方法的代码行，再按 F5。

请参阅

- 并行 LINQ (PLINQ)

如何：处理 PLINQ 查询中的异常

2021/11/16 •

本主题中的第一个示例展示了如何处理 PLINQ 查询在执行时可能会抛出的 System.AggregateException。第二个示例展示了如何将 try-catch 块置于委托中，并尽可能靠近抛出异常的代码位置。这样一来，可以在异常发生时尽快捕获它们，并继续执行查询。如果允许异常向上冒泡回到联接线程，则查询也许可以在引发异常后继续处理一些项。

如果 PLINQ 回退到顺序执行且发生异常，异常可能会直接传播，而不会被包装在 AggregateException 中。此外，ThreadAbortException 始终都会直接传播。

> **NOTE**
>
> 选中“仅我的代码”后，Visual Studio 会在抛出异常的代码行处中断，并显示错误消息“用户代码未处理异常”。此错误是良性的。可以按 F5 继续运行，并请参阅下面示例中所示的异常处理行为。为了阻止 Visual Studio 在第一个错误出现时中断，只需依次转到“工具”、“选项”、“调试”、“常规” 下，取消选中“仅我的代码”复选框即可。
>
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

此示例展示了如何将 try-catch 块置于执行查询以捕获抛出的任何 System.AggregateException 的代码附近。

```
// Paste into PLINQDataSample class.
static void PLINQExceptions_1()
{
    // Using the raw string array here. See PLINQ Data Sample.
    string[] customers = GetCustomersAsStrings().ToArray();

    // First, we must simulate some currupt input.
    customers[54] = "###";

    var parallelQuery = from cust in customers.AsParallel()
                        let fields = cust.Split(',')
                        where fields[3].StartsWith("C") //throw indexoutofrange
                        select new { city = fields[3], thread = Thread.CurrentThread.ManagedThreadId };
    try
    {
        // We use ForAll although it doesn't really improve performance
        // since all output is serialized through the Console.
        parallelQuery.ForAll(e => Console.WriteLine("City: {0}, Thread:{1}", e.city, e.thread));
    }

    // In this design, we stop query processing when the exception occurs.
    catch (AggregateException e)
    {
        foreach (var ex in e.InnerExceptions)
        {
            Console.WriteLine(ex.Message);
            if (ex is IndexOutOfRangeException)
                Console.WriteLine("The data source is corrupt. Query stopped.");
        }
    }
}
```

```
' Paste into PLINQDataSample class
Shared Sub PLINQExceptions_1()

    ' Using the raw string array here. See PLINQ Data Sample.
    Dim customers As String() = GetCustomersAsStrings().ToArray()

    ' First, we must simulate some currupt input.
    customers(20) = "###"

    'throws indexoutofrange
    Dim query = From cust In customers.AsParallel()
                Let fields = cust.Split(","c)
                Where fields(3).StartsWith("C")
                Select fields
    Try
        ' We use ForAll although it doesn't really improve performance
        ' since all output is serialized through the Console.
        query.ForAll(Sub(e)
                         Console.WriteLine("City: {0}, Thread:{1}")
                     End Sub)
    Catch e As AggregateException

        ' In this design, we stop query processing when the exception occurs.
        For Each ex In e.InnerExceptions
            Console.WriteLine(ex.Message)
            If TypeOf ex Is IndexOutOfRangeException Then
                Console.WriteLine("The data source is corrupt. Query stopped.")
            End If
        Next
    End Try
End Sub
```

在此示例中，查询无法在异常抛出后继续运行。在应用代码捕获到异常时，PLINQ 已停止对所有线程运行查询。

示例

下面的示例展示了如何将 try-catch 块置于委托中，以便可以捕获异常并继续执行查询。

```
// Paste into PLINQDataSample class.
static void PLINQExceptions_2()
{
    var customers = GetCustomersAsStrings().ToArray();
    // Using the raw string array here.
    // First, we must simulate some currupt input
    customers[54] = "###";

    // Assume that in this app, we expect malformed data
    // occasionally and by design we just report it and continue.
    static bool IsTrue(string[] f, string c)
    {
        try
        {
            string s = f[3];
            return s.StartsWith(c);
        }
        catch (IndexOutOfRangeException)
        {
            Console.WriteLine($"Malformed cust: {f}");
            return false;
        }
    };

    // Using the raw string array here
    var parallelQuery =
        from cust in customers.AsParallel()
        let fields = cust.Split(',')
        where IsTrue(fields, "C") //use a named delegate with a try-catch
        select new { City = fields[3] };

    try
    {
        // We use ForAll although it doesn't really improve performance
        // since all output must be serialized through the Console.
        parallelQuery.ForAll(e => Console.WriteLine(e.City));
    }

    // IndexOutOfRangeException will not bubble up
    // because we handle it where it is thrown.
    catch (AggregateException e)
    {
        foreach (var ex in e.InnerExceptions)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

```
' Paste into PLINQDataSample class
Shared Sub PLINQExceptions_2()

    Dim customers() = GetCustomersAsStrings().ToArray()
    ' Using the raw string array here.
    ' First, we must simulate some currupt input
    customers(20) = "###"

    ' Create a delegate with a lambda expression.
    ' Assume that in this app, we expect malformed data
    ' occasionally and by design we just report it and continue.
    Dim isTrue As Func(Of String(), String, Boolean) = Function(f, c)

                                                           Try

                                                               Dim s As String = f(3)
                                                               Return s.StartsWith(c)

                                                           Catch e As IndexOutOfRangeException

                                                               Console.WriteLine("Malformed cust: {0}", f)
                                                               Return False
                                                           End Try
                                                       End Function

    ' Using the raw string array here
    Dim query = From cust In customers.AsParallel()
                Let fields = cust.Split(",","c)
                Where isTrue(fields, "C")
                Select New With {.City = fields(3)}
    Try
        ' We use ForAll although it doesn't really improve performance
        ' since all output must be serialized through the Console.
        query.ForAll(Sub(e) Console.WriteLine(e.City))


        ' IndexOutOfRangeException will not bubble up
        ' because we handle it where it is thrown.
    Catch e As AggregateException
        For Each ex In e.InnerExceptions
            Console.WriteLine(ex.Message)
        Next
    End Try
End Sub
```

编译代码

- 若要编译和运行这些示例，请将它们复制到“PLINQ 数据样本”示例中，并通过 Main 调用此方法。

可靠编程

仅在确定如何处理异常时，才捕获异常，以免破坏程序状态。

另请参阅

- ParallelEnumerable
- 并行 LINQ (PLINQ)

如何：取消 PLINQ 查询

2021/11/16 •

下面的示例展示了取消 PLINQ 查询的两种方法。第一个示例展示了如何取消主要由数据遍历组成的查询。第二个示例展示了如何取消包含计算成本很高的用户函数的查询。

> **NOTE**
>
> 选中“仅我的代码”后，Visual Studio 会在抛出异常的代码行处中断，并显示错误消息“用户代码未处理异常”。此错误是良性的。可以按 F5 继续运行，并请参阅下面示例中所示的异常处理行为。为了阻止 Visual Studio 在第一个错误出现时中断，只需依次转到“工具”、“选项”、“调试”、“常规” 下，取消选中“仅我的代码”复选框即可。
>
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

```
namespace PLINQCancellation_1
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
    using static System.Console;

    class Program
    {
        static void Main()
        {
            int[] source = Enumerable.Range(1, 10000000).ToArray();
            using CancellationTokenSource cts = new();

            // Start a new asynchronous task that will cancel the
            // operation from another thread. Typically you would call
            // Cancel() in response to a button click or some other
            // user interface event.
            Task.Factory.StartNew(() =>
            {
                UserClicksTheCancelButton(cts);
            });

            int[] results = null;
            try
            {
                results =
                    (from num in source.AsParallel().WithCancellation(cts.Token)
                     where num % 3 == 0
                     orderby num descending
                     select num).ToArray();
            }
            catch (OperationCanceledException e)
            {
                WriteLine(e.Message);
            }
            catch (AggregateException ae)
            {
                if (ae.InnerExceptions != null)
                {
                    foreach (Exception e in ae.InnerExceptions)
```

```
                    foreach (Exception e in ae.InnerExceptions)
                    {
                        WriteLine(e.Message);
                    }
                }
            }

            foreach (var item in results ?? Array.Empty<int>())
            {
                WriteLine(item);
            }
            WriteLine();
            ReadKey();
        }

        static void UserClicksTheCancelButton(CancellationTokenSource cts)
        {
            // Wait between 150 and 500 ms, then cancel.
            // Adjust these values if necessary to make
            // cancellation fire while query is still executing.
            Random rand = new();
            Thread.Sleep(rand.Next(150, 500));
            cts.Cancel();
        }
    }
}
```

```
Class Program
    Private Shared Sub Main(ByVal args As String())
        Dim source As Integer() = Enumerable.Range(1, 10000000).ToArray()
        Dim cs As New CancellationTokenSource()

        ' Start a new asynchronous task that will cancel the
        ' operation from another thread. Typically you would call
        ' Cancel() in response to a button click or some other
        ' user interface event.
        Task.Factory.StartNew(Sub()
                                  UserClicksTheCancelButton(cs)
                              End Sub)

        Dim results As Integer() = Nothing
        Try

            results = (From num In source.AsParallel().WithCancellation(cs.Token) _
                       Where num Mod 3 = 0 _
                       Order By num Descending _
                       Select num).ToArray()
        Catch e As OperationCanceledException

            Console.WriteLine(e.Message)
        Catch ae As AggregateException

            If ae.InnerExceptions IsNot Nothing Then
                For Each e As Exception In ae.InnerExceptions
                    Console.WriteLine(e.Message)
                Next
            End If
        Finally
            cs.Dispose()
        End Try

        If results IsNot Nothing Then
            For Each item In results
                Console.WriteLine(item)
            Next
        End If
        Console.WriteLine()

        Console.ReadKey()
    End Sub

    Private Shared Sub UserClicksTheCancelButton(ByVal cs As CancellationTokenSource)
        ' Wait between 150 and 500 ms, then cancel.
        ' Adjust these values if necessary to make
        ' cancellation fire while query is still executing.
        Dim rand As New Random()
        Thread.Sleep(rand.[Next](150, 350))
        cs.Cancel()
    End Sub
End Class
```

PLINQ 框架不会将一个 OperationCanceledException 滚动到 System.AggregateException 中；必须在单独的 catch 块中处理 OperationCanceledException。如果一个或多个用户委托抛出 OperationCanceledException(externalCT)（通过使用外部 System.Threading.CancellationToken），但没有其他任何异常，且查询被定义为 `AsParallel().WithCancellation(externalCT)`，那么 PLINQ 会抛出 OperationCanceledException (externalCT)，而不是 System.AggregateException。不过，如果一个用户委托抛出 OperationCanceledException，另一个委托抛出另一种类型的异常，那么这两个异常都会滚动到 AggregateException 中。

关于取消的一般性指南如下：

1. 如果执行用户委托取消，应将外部 CancellationToken 告知给 PLINQ，并引发 OperationCanceledException(externalCT)。

2. 如果发生取消且没有引发其他任何异常，则处理 OperationCanceledException，而不是 AggregateException。

示例

下面的示例展示了如何在用户代码中使用计算成本高的函数时处理取消。

```
namespace PLINQCancellation_2
{
    using System;
    using System.Linq;
    using System.Threading;
    using System.Threading.Tasks;
    using static System.Console;

    class Program
    {
        static void Main(string[] args)
        {
            int[] source = Enumerable.Range(1, 10000000).ToArray();
            using CancellationTokenSource cts = new();

            // Start a new asynchronous task that will cancel the
            // operation from another thread. Typically you would call
            // Cancel() in response to a button click or some other
            // user interface event.
            Task.Factory.StartNew(() =>
            {
                UserClicksTheCancelButton(cts);
            });

            double[] results = null;
            try
            {
                results =
                    (from num in source.AsParallel().WithCancellation(cts.Token)
                     where num % 3 == 0
                     select Function(num, cts.Token)).ToArray();
            }
            catch (OperationCanceledException e)
            {
                WriteLine(e.Message);
            }
            catch (AggregateException ae)
            {
                if (ae.InnerExceptions != null)
                {
                    foreach (Exception e in ae.InnerExceptions)
                        WriteLine(e.Message);
                }
            }

            foreach (var item in results ?? Array.Empty<double>())
            {
                WriteLine(item);
            }
            WriteLine();
            ReadKey();
        }

        // A toy method to simulate work.
        static double Function(int n, CancellationToken ct)
        {
```

```
            // If work is expected to take longer than 1 ms
            // then try to check cancellation status more
            // often within that work.
            for (int i = 0; i < 5; i++)
            {
                // Work hard for approx 1 millisecond.
                Thread.SpinWait(50000);

                // Check for cancellation request.
                ct.ThrowIfCancellationRequested();
            }
            // Anything will do for our purposes.
            return Math.Sqrt(n);
        }

        static void UserClicksTheCancelButton(CancellationTokenSource cts)
        {
            // Wait between 150 and 500 ms, then cancel.
            // Adjust these values if necessary to make
            // cancellation fire while query is still executing.
            Random rand = new();
            Thread.Sleep(rand.Next(150, 500));
            WriteLine("Press 'c' to cancel");
            if (ReadKey().KeyChar == 'c')
            {
                cts.Cancel();
            }
        }
    }
}
```

```
Class Program2
    Private Shared Sub Main(ByVal args As String())


        Dim source As Integer() = Enumerable.Range(1, 10000000).ToArray()
        Dim cs As New CancellationTokenSource()

        ' Start a new asynchronous task that will cancel the
        ' operation from another thread. Typically you would call
        ' Cancel() in response to a button click or some other
        ' user interface event.
        Task.Factory.StartNew(Sub()

                                  UserClicksTheCancelButton(cs)
                              End Sub)

        Dim results As Double() = Nothing
        Try

            results = (From num In source.AsParallel().WithCancellation(cs.Token) _
                       Where num Mod 3 = 0 _
                       Select [Function](num, cs.Token)).ToArray()
        Catch e As OperationCanceledException


            Console.WriteLine(e.Message)
        Catch ae As AggregateException
            If ae.InnerExceptions IsNot Nothing Then
                For Each e As Exception In ae.InnerExceptions
                    Console.WriteLine(e.Message)
                Next
            End If
        Finally
            cs.Dispose()
        End Try
```

```
        If results IsNot Nothing Then
            For Each item In results
                Console.WriteLine(item)
            Next
        End If
        Console.WriteLine()

        Console.ReadKey()
    End Sub

    ' A toy method to simulate work.
    Private Shared Function [Function](ByVal n As Integer, ByVal ct As CancellationToken) As Double
        ' If work is expected to take longer than 1 ms
        ' then try to check cancellation status more
        ' often within that work.
        For i As Integer = 0 To 4
            ' Work hard for approx 1 millisecond.
            Thread.SpinWait(50000)

            ' Check for cancellation request.
            If ct.IsCancellationRequested Then
                Throw New OperationCanceledException(ct)
            End If
        Next
        ' Anything will do for our purposes.
        Return Math.Sqrt(n)
    End Function

    Private Shared Sub UserClicksTheCancelButton(ByVal cs As CancellationTokenSource)
        ' Wait between 150 and 500 ms, then cancel.
        ' Adjust these values if necessary to make
        ' cancellation fire while query is still executing.
        Dim rand As New Random()
        Thread.Sleep(rand.[Next](150, 350))
        Console.WriteLine("Press 'c' to cancel")
        If Console.ReadKey().KeyChar = "c"c Then
            cs.Cancel()

        End If
    End Sub
End Class
```

在用户代码中处理取消时，无需在查询定义中使用 WithCancellation。不过，之所以建议使用 WithCancellation 是因为，WithCancellation 对查询性能没有影响，并让取消由查询运算符和用户代码进行处理。

为了确保系统响应速度，建议每毫秒检查一次取消；不过，只要不超过每 10 毫秒一次，任何频率都认为是可接受的。此频率不得对代码性能产生不利影响。

如果枚举器已遭清理（例如，当代码跳出循环访问查询结果的 foreach（Visual Basic 中的 For Each）循环时），查询就会被取消，但不会引发异常。

请参阅

- ParallelEnumerable
- 并行 LINQ (PLINQ)
- 托管线程中的取消

如何：编写自定义 PLINQ 聚合函数

2021/11/16 •

此示例展示了如何使用 Aggregate 方法，将自定义聚合函数应用于源序列。

> **WARNING**
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

下面的示例计算整数序列的标准偏差。

```
namespace PLINQAggregation
{
    using System;
    using System.Linq;

    class aggregation
    {
        static void Main(string[] args)
        {

            // Create a data source for demonstration purposes.
            int[] source = new int[100000];
            Random rand = new Random();
            for (int x = 0; x < source.Length; x++)
            {
                // Should result in a mean of approximately 15.0.
                source[x] = rand.Next(10, 20);
            }

            // Standard deviation calculation requires that we first
            // calculate the mean average. Average is a predefined
            // aggregation operator, along with Max, Min and Count.
            double mean = source.AsParallel().Average();

            // We use the overload that is unique to ParallelEnumerable. The
            // third Func parameter combines the results from each thread.
            double standardDev = source.AsParallel().Aggregate(
                // initialize subtotal. Use decimal point to tell
                // the compiler this is a type double. Can also use: 0d.
                0.0,

                 // do this on each thread
                 (subtotal, item) => subtotal + Math.Pow((item - mean), 2),

                 // aggregate results after all threads are done.
                 (total, thisThread) => total + thisThread,

                // perform standard deviation calc on the aggregated result.
                (finalSum) => Math.Sqrt((finalSum / (source.Length - 1)))
            );
            Console.WriteLine("Mean value is = {0}", mean);
            Console.WriteLine("Standard deviation is {0}", standardDev);
            Console.ReadLine();
        }
    }
}
```

```
Class aggregation
    Private Shared Sub Main(ByVal args As String())

        ' Create a data source for demonstration purposes.
        Dim source As Integer() = New Integer(99999) {}
        Dim rand As New Random()
        For x As Integer = 0 To source.Length - 1
            ' Should result in a mean of approximately 15.0.
            source(x) = rand.[Next](10, 20)
        Next

        ' Standard deviation calculation requires that we first
        ' calculate the mean average. Average is a predefined
        ' aggregation operator, along with Max, Min and Count.
        Dim mean As Double = source.AsParallel().Average()


        ' We use the overload that is unique to ParallelEnumerable. The
        ' third Func parameter combines the results from each thread.
        ' initialize subtotal. Use decimal point to tell
        ' the compiler this is a type double. Can also use: 0d.

        ' do this on each thread

        ' aggregate results after all threads are done.

        ' perform standard deviation calc on the aggregated result.
        Dim standardDev As Double = source.AsParallel().Aggregate(0.0R, Function(subtotal, item) subtotal +
Math.Pow((item - mean), 2), Function(total, thisThread) total + thisThread, Function(finalSum)
Math.Sqrt((finalSum / (source.Length - 1))))
        Console.WriteLine("Mean value is = {0}", mean)
        Console.WriteLine("Standard deviation is {0}", standardDev)

        Console.ReadLine()
    End Sub
End Class
```

此示例使用 PLINQ 独有的聚合标准查询运算符的重载。此重载需要使用额外的 System.Func<T1,T2,TResult> 作为第三个输入参数。此委托先合并所有线程的结果，再对聚合结果执行最终计算。此示例将所有线程的总和结果相加到一起。

请注意，如果 Lambda 表达式主体由一个表达式组成，System.Func<T,TResult> 委托的返回值就是表达式值。

另请参阅

- ParallelEnumerable
- 并行 LINQ (PLINQ)

如何：在 PLINQ 中指定执行模式

2021/11/16 •

此示例展示了如何强制 PLINQ 规避默认启发，同时并行执行查询，无论查询的形状如何。

> **NOTE**
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

```
// Paste into PLINQDataSample class.
static void ForceParallel()
{
    var customers = GetCustomers();
    var parallelQuery = (from cust in customers.AsParallel()
                            .WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                         where cust.City == "Berlin"
                         select cust.CustomerName)
                        .ToList();
}
```

```
Private Shared Sub ForceParallel()
    Dim customers = GetCustomers()
    Dim parallelQuery = (From cust In
customers.AsParallel().WithExecutionMode(ParallelExecutionMode.ForceParallelism)
                         Where cust.City = "Berlin"
                         Select cust.CustomerName).ToList()
End Sub
```

PLINQ 旨在利用机会并行执行查询。不过，并非所有查询都受益于并行执行。例如，如果查询包含一个工作量较小的用户委托，顺序运行查询的速度通常更快。顺序执行速度更快是因为相较所实现的加速，启用并行执行所涉及的开销更加昂贵。因此，PLINQ 并不自动并行执行每个查询。它先检查查询的形状和所包含的各种运算符。根据此分析，默认执行模式下的 PLINQ 可能会决定顺序执行部分或全部查询。不过，在某些情况下，你对查询更加了解，所做决定可能优于 PLINQ 根据分析所做的决定。例如，你可能知道委托比较昂贵，且查询一定会受益于并行执行。在这种情况下，可以使用 WithExecutionMode 方法并指定 ForceParallelism 值，以指示 PLINQ 始终并行运行查询。

编译代码

将此代码剪切并粘贴到 PLINQ 数据样本中，并通过 `Main` 调用方法。

请参阅

- AsSequential
- 并行 LINQ (PLINQ)

如何：在 PLINQ 中指定合并选项

2021/11/16 •

此示例展示了如何指定应用于 PLINQ 查询中所有后续运算符的合并选项。虽然无需显式设置合并选项，但这样做可以提升性能。若要详细了解合并选项，请参阅 PLINQ 中的合并选项。

> **WARNING**
> 本示例旨在演示用法，运行速度可能不如等效的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

示例

下面的示例展示了合并选项在基本方案中的行为，此方案包含无序源，并将高成本函数应用于每个元素。

```
namespace MergeOptions
{
    using System;
    using System.Diagnostics;
    using System.Linq;
    using System.Threading;

    class Program
    {
        static void Main(string[] args)
        {

            var nums = Enumerable.Range(1, 10000);

            // Replace NotBuffered with AutoBuffered
            // or FullyBuffered to compare behavior.
            var scanLines = from n in nums.AsParallel()
                                .WithMergeOptions(ParallelMergeOptions.NotBuffered)
                            where n % 2 == 0
                            select ExpensiveFunc(n);

            Stopwatch sw = Stopwatch.StartNew();
            foreach (var line in scanLines)
            {
                Console.WriteLine(line);
            }

            Console.WriteLine("Elapsed time: {0} ms. Press any key to exit.",
                            sw.ElapsedMilliseconds);
            Console.ReadKey();
        }

        // A function that demonstrates what a fly
        // sees when it watches television :-)
        static string ExpensiveFunc(int i)
        {
            Thread.SpinWait(2000000);
            return string.Format("{0} ****************************************", i);
        }
    }
}
```

```
Class MergeOptions2
    Sub DoMergeOptions()

        Dim nums = Enumerable.Range(1, 10000)

        ' Replace NotBuffered with AutoBuffered
        ' or FullyBuffered to compare behavior.
        Dim scanLines = From n In nums.AsParallel().WithMergeOptions(ParallelMergeOptions.NotBuffered)
                        Where n Mod 2 = 0
                        Select ExpensiveFunc(n)

        Dim sw = Stopwatch.StartNew()
        For Each line In scanLines
            Console.WriteLine(line)
        Next

        Console.WriteLine("Elapsed time: {0} ms. Press any key to exit.")
        Console.ReadKey()

    End Sub
    ' A function that demonstrates what a fly
    ' sees when it watches television :-)
    Function ExpensiveFunc(ByVal i As Integer) As String
        Threading.Thread.SpinWait(2000000)
        Return String.Format("{0} ****************************************", i)
    End Function
End Class
```

如果 AutoBuffered 选项在第一个元素生成前导致不利延迟发生，请尝试使用 NotBuffered 选项，更快更顺畅地生成结果元素。

另请参阅

- ParallelMergeOptions
- 并行 LINQ (PLINQ)

如何：使用 PLINQ 循环访问文件目录

2021/11/16 •

本文展示了两种方法，以对文件目录平行执行操作。第一个查询使用 GetFiles 方法，在数组中填充目录和所有子目录中的文件名。在整个数组填充完成前，此方法不会返回数组，所以可能会在操作开始时引入延迟。不过，在填充数组后，PLINQ 可以快速地并行处理数组。

第二个查询使用立即开始返回结果的静态 EnumerateDirectories 和 EnumerateFiles 方法。循环访问大型目录树时，这种方法可能会比第一个示例更快，不过处理时间取决于很多因素。

> **NOTE**
> 这些示例用于演示用法，可能不会比相当的顺序 LINQ to Objects 查询快。若要详细了解加速，请参阅了解 PLINQ 中的加速。

GetFiles 示例

此示例展示了如何在以下简单方案中循环访问文件目录：有权访问树中的所有目录，文件大小不大，访问时间也不是很长。这种方法在最初构造文件名数组时有一段延迟。

```
// Use Directory.GetFiles to get the source sequence of file names.
public static void FileIterationOne(string path)
{
    var sw = Stopwatch.StartNew();
    int count = 0;
    string[] files = null;
    try
    {
        files = Directory.GetFiles(path, "*.*", SearchOption.AllDirectories);
    }
    catch (UnauthorizedAccessException)
    {
        Console.WriteLine("You do not have permission to access one or more folders in this directory
tree.");
        return;
    }
    catch (FileNotFoundException)
    {
        Console.WriteLine($"The specified directory {path} was not found.");
    }

    var fileContents =
        from FileName in files.AsParallel()
        let extension = Path.GetExtension(FileName)
        where extension == ".txt" || extension == ".htm"
        let Text = File.ReadAllText(FileName)
        select new
        {
            Text,
            FileName
        };

    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine($"{Path.GetFileName(item.FileName)}:{item.Text.Length}");
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle(ex =>
        {
            if (ex is UnauthorizedAccessException uae)
            {
                Console.WriteLine(uae.Message);
                return true;
            }
            return false;
        });
    }

    Console.WriteLine($"FileIterationOne processed {count} files in {sw.ElapsedMilliseconds} milliseconds");
}
```

EnumerateFiles 示例

此示例展示了如何在以下简单方案中循环访问文件目录：有权访问树中的所有目录，文件大小不大，访问时间也不是很长。这种方法比上一示例更快地开始生成结果。

```
public static void FileIterationTwo(string path) //225512 ms
{
    var count = 0;
    var sw = Stopwatch.StartNew();
    var fileNames =
        from dir in Directory.EnumerateFiles(path, "*.*", SearchOption.AllDirectories)
        select dir;

    var fileContents =
        from FileName in fileNames.AsParallel()
        let extension = Path.GetExtension(FileName)
        where extension == ".txt" || extension == ".htm"
        let Text = File.ReadAllText(FileName)
        select new
        {
            Text,
            FileName
        };
    try
    {
        foreach (var item in fileContents)
        {
            Console.WriteLine($"{Path.GetFileName(item.FileName)}:{item.Text.Length}");
            count++;
        }
    }
    catch (AggregateException ae)
    {
        ae.Handle(ex =>
        {
            if (ex is UnauthorizedAccessException uae)
            {
                Console.WriteLine(uae.Message);
                return true;
            }
            return false;
        });
    }

    Console.WriteLine($"FileIterationTwo processed {count} files in {sw.ElapsedMilliseconds} milliseconds");
}
```

使用 GetFiles 时，请确保有权访问树中的所有目录。否则，将引发异常，且不会返回任何结果。如果在 PLINQ 查询中使用 EnumerateDirectories，棘手的是合理处理 I/O 异常，以便能够继续循环访问。如果代码必须处理 I/O 或未经授权的访问异常，应考虑使用如何：使用并行类循环访问文件目录中介绍的方法。

如果 I/O 延迟造成问题（例如，对于通过网络的文件 I/O），请考虑使用 TPL 和传统 .NET 异步编程和这篇博客文章中介绍的某种异步 I/O 方法。

请参阅

- 并行 LINQ (PLINQ)

如何：衡量 PLINQ 查询性能

2021/11/16 •

此示例展示了如何使用 Stopwatch 类度量 PLINQ 查询的执行时间。

示例

此示例使用空 `foreach` 循环(Visual Basic 中为 `For Each`)来衡量查询执行所用的时间。在实际代码中，循环通常会包含其他处理步骤，这会增加查询总执行时间。请注意，秒表会恰好在循环开始之前启动，因为此时查询开始执行。如果需要更精细的度量，可以使用 `ElapsedTicks` 属性，而不是 `ElapsedMilliseconds`。

```
using System;
using System.Diagnostics;
using System.Linq;

class ExampleMeasure
{
    static void Main()
    {
        var source = Enumerable.Range(0, 3000000);

        var queryToMeasure =
             from num in source.AsParallel()
             where num % 3 == 0
             select Math.Sqrt(num);

        Console.WriteLine("Measuring...");

        // The query does not run until it is enumerated.
        // Therefore, start the timer here.
        var sw = Stopwatch.StartNew();

        // For pure query cost, enumerate and do nothing else.
        foreach (var n in queryToMeasure) { }

        sw.Stop();
        long elapsed = sw.ElapsedMilliseconds; // or sw.ElapsedTicks
        Console.WriteLine("Total query time: {0} ms", elapsed);

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

```
Module ExampleMeasure
    Sub Main()
        Dim source = Enumerable.Range(0, 3000000)
        ' Define parallel and non-parallel queries.
        Dim queryToMeasure = From num In source.AsParallel()
                             Where num Mod 3 = 0
                             Select Math.Sqrt(num)

        Console.WriteLine("Measuring...")

        ' The query does not run until it is enumerated.
        ' Therefore, start the timer here.
        Dim sw = Stopwatch.StartNew()

        ' For pure query cost, enumerate and do nothing else.
        For Each n As Double In queryToMeasure
        Next

        sw.Stop()
        Dim elapsed As Long = sw.ElapsedMilliseconds  ' or sw.ElapsedTicks
        Console.WriteLine($"Total query time: {elapsed} ms.")

        Console.WriteLine("Press any key to exit.")
        Console.ReadKey()
    End Sub
End Module
```

试验查询实现时，总执行时间是有用的指标，但它并不总是能说明一切。为了更深入全面地了解查询线程相互之间以及查询线程和其他运行进程之前的交互，请使用并发可视化工具。

请参阅

- 并行 LINQ (PLINQ)

PLINQ 数据示例

2021/11/16 •

此示例包含 .csv 格式的示例数据，以及将它转换为"客户"、"产品"、"订单"和"订单详情"内存中集合的方法。若要进一步试验 PLINQ，可以将其他一些主题中的代码示例粘贴到本主题的代码中，并通过 `Main` 方法调用它。也可以将此类数据用于自己的 PLINQ 查询。

数据表示 Northwind 数据库中的一部分。其中包含五十 (50) 个客户记录，但并不包含所有字段。此外，还包含"订单"中的一部分行和每个客户的相应 Order_Detail 数据。所有产品都包含在内。

> **NOTE**
> 数据集不是非常大，无法证明对于仅包含基本 `where` 和 `select` 子句的查询，PLINQ 比 LINQ to Objects 快。为了观察此类小型数据集的加速，请使用包含对数据集中每个元素执行计算成本高的操作的查询。

设置此示例

1. 创建 Visual Basic 或 Visual C# 控制台应用项目。
2. 通过运行下面这些步骤后面的代码，替换 Module1.vb 或 Program.cs 的内容。
3. 在"项目"菜单上，单击"添加新项"。选择"文本文件"，再单击"确定"。复制此主题中的数据，再将它粘贴到新的文本文件中。在"文件"菜单上，单击"保存"，将文件命名为"Plinqdata.csv"，再将它保存到包含源代码文件的文件夹中。
4. 按 F5 以验证项目是否正确生成和运行。下面的输出应显示在控制台窗口中。

```
Customer count: 50
Product count: 77
Order count: 190
Order Details count: 483
Press any key to exit.
```

```
// This class contains a subset of data from the Northwind database
// in the form of string arrays. The methods such as GetCustomers, GetOrders, and so on
// transform the strings into object arrays that you can query in an object-oriented way.
// Many of the code examples in the PLINQ How-to topics are designed to be pasted into
// the PLINQDataSample class and invoked from the Main method.
partial class PLINQDataSample
{

    public static void Main()
    {
        ////Call methods here.
        TestDataSource();

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    static void TestDataSource()
    {
        Console.WriteLine("Customer count: {0}", GetCustomers().Count());
        Console.WriteLine("Product count: {0}", GetProducts().Count());
        Console.WriteLine("Order count: {0}", GetOrders().Count());
        Console.WriteLine("Order Details count: {0}", GetOrderDetails().Count());
    }
```

```
}

#region DataClasses
public class Order
{
    private Lazy<OrderDetail[]> _orderDetails;
    public Order()
    {
        _orderDetails = new Lazy<OrderDetail[]>(() => GetOrderDetailsForOrder(OrderID));
    }
    public int OrderID { get; set; }
    public string CustomerID { get; set; }
    public DateTime OrderDate { get; set; }
    public DateTime ShippedDate { get; set; }
    public OrderDetail[] OrderDetails { get { return _orderDetails.Value; } }
}

public class Customer
{
    private Lazy<Order[]> _orders;
    public Customer()
    {
        _orders = new Lazy<Order[]>(() => GetOrdersForCustomer(CustomerID));
    }
    public string CustomerID { get; set; }
    public string CustomerName { get; set; }
    public string Address { get; set; }
    public string City { get; set; }
    public string PostalCode { get; set; }
    public Order[] Orders
    {
        get
        {
            return _orders.Value;
        }
    }
}

public class Product
{
    public string ProductName { get; set; }
    public int ProductID { get; set; }
    public double UnitPrice { get; set; }
}

public class OrderDetail
{
    public int OrderID { get; set; }
    public int ProductID { get; set; }
    public double UnitPrice { get; set; }
    public double Quantity { get; set; }
    public double Discount { get; set; }
}
#endregion

public static IEnumerable<string> GetCustomersAsStrings()
{
    return System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                    .SkipWhile((line) => line.StartsWith("CUSTOMERS") == false)
                                     .Skip(1)
                                    .TakeWhile((line) => line.StartsWith("END CUSTOMERS") == false);
}

public static IEnumerable<Customer> GetCustomers()
{
    var customers = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                    .SkipWhile((line) => line.StartsWith("CUSTOMERS") == false)
                                    .Skip(1)
                                    .TakeWhile((line) => line.StartsWith("END CUSTOMERS") == false);
    return (from line in customers
```

```
    return (from line in customers
            let fields = line.Split(',')
            let custID = fields[0].Trim()
            select new Customer()
            {
                CustomerID = custID,
                CustomerName = fields[1].Trim(),
                Address = fields[2].Trim(),
                City = fields[3].Trim(),
                PostalCode = fields[4].Trim()
            });
}

public static Order[] GetOrdersForCustomer(string id)
{
    // Assumes we copied the file correctly!
    var orders = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                .SkipWhile((line) => line.StartsWith("ORDERS") == false)
                                 .Skip(1)
                                .TakeWhile((line) => line.StartsWith("END ORDERS") == false);
    var orderStrings = from line in orders
                       let fields = line.Split(',')
                       where fields[1].CompareTo(id) == 0
                       select new Order()
                       {
                           OrderID = Convert.ToInt32(fields[0]),
                           CustomerID = fields[1].Trim(),
                           OrderDate = DateTime.Parse(fields[2]),
                           ShippedDate = DateTime.Parse(fields[3])
                       };
    return orderStrings.ToArray();
}

//  "10248, VINET, 7/4/1996 12:00:00 AM, 7/16/1996 12:00:00 AM
public static IEnumerable<Order> GetOrders()
{
    // Assumes we copied the file correctly!
    var orders = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                .SkipWhile((line) => line.StartsWith("ORDERS") == false)
                                 .Skip(1)
                                .TakeWhile((line) => line.StartsWith("END ORDERS") == false);
    return from line in orders
           let fields = line.Split(',')

           select new Order()
           {
               OrderID = Convert.ToInt32(fields[0]),
               CustomerID = fields[1].Trim(),
               OrderDate = DateTime.Parse(fields[2]),
               ShippedDate = DateTime.Parse(fields[3])
           };
}

public static IEnumerable<Product> GetProducts()
{
    // Assumes we copied the file correctly!
    var products = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                .SkipWhile((line) => line.StartsWith("PRODUCTS") == false)
                                 .Skip(1)
                                .TakeWhile((line) => line.StartsWith("END PRODUCTS") == false);
    return from line in products
           let fields = line.Split(',')
           select new Product()
           {
               ProductID = Convert.ToInt32(fields[0]),
               ProductName = fields[1].Trim(),
               UnitPrice = Convert.ToDouble(fields[2])
           };
}
```

```
    public static IEnumerable<OrderDetail> GetOrderDetails()
    {
        // Assumes we copied the file correctly!
        var orderDetails = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("ORDER DETAILS") == false)
                                          .Skip(1)
                                         .TakeWhile((line) => line.StartsWith("END ORDER DETAILS") == false);

        return from line in orderDetails
               let fields = line.Split(',')
               select new OrderDetail()
               {
                   OrderID = Convert.ToInt32(fields[0]),
                   ProductID = Convert.ToInt32(fields[1]),
                   UnitPrice = Convert.ToDouble(fields[2]),
                   Quantity = Convert.ToDouble(fields[3]),
                   Discount = Convert.ToDouble(fields[4])
               };
    }

    public static OrderDetail[] GetOrderDetailsForOrder(int id)
    {
        // Assumes we copied the file correctly!
        var orderDetails = System.IO.File.ReadAllLines(@"..\..\plinqdata.csv")
                                         .SkipWhile((line) => line.StartsWith("ORDER DETAILS") == false)
                                          .Skip(1)
                                         .TakeWhile((line) => line.StartsWith("END ORDER DETAILS") == false);

        var orderDetailStrings = from line in orderDetails
                                 let fields = line.Split(',')
                                 let ordID = Convert.ToInt32(fields[0])
                                 where ordID == id
                                 select new OrderDetail()
                                 {
                                     OrderID = ordID,
                                     ProductID = Convert.ToInt32(fields[1]),
                                     UnitPrice = Convert.ToDouble(fields[2]),
                                     Quantity = Convert.ToDouble(fields[3]),
                                     Discount = Convert.ToDouble(fields[4])
                                 };

        return orderDetailStrings.ToArray();
    }
}
```

```
    ' This class contains a subset of data from the Northwind database
    ' in the form of string arrays. The methods such as GetCustomers, GetOrders, and so on
    ' transform the strings into object arrays that you can query in an object-oriented way.
    ' Many of the code examples in the PLINQ How-to topics are designed to be pasted into
    ' the PLINQDataSample class and invoked from the Main method.
    ' IMPORTANT: This data set is not large enough for meaningful comparisons of PLINQ vs. LINQ performance.
    Class PLINQDataSample
        Shared Sub Main(ByVal args As String())

            'Call methods here.
            TestDataSource()

            Console.WriteLine("Press any key to exit.")
            Console.ReadKey()
        End Sub

        Shared Sub TestDataSource()
            Console.WriteLine("Customer count: {0}", GetCustomers().Count())
            Console.WriteLine("Product count: {0}", GetProducts().Count())
            Console.WriteLine("Order count: {0}", GetOrders().Count())
            Console.WriteLine("Order Details count: {0}", GetOrderDetails().Count())
```

```
        End Sub

#Region "DataClasses"
        Class Order
            Public Sub New()
                _OrderDetails = New Lazy(Of OrderDetail())(Function() GetOrderDetailsForOrder(OrderID))
            End Sub
            Private _OrderID As Integer
            Public Property OrderID() As Integer
                Get
                    Return _OrderID
                End Get
                Set(ByVal value As Integer)
                    _OrderID = value
                End Set
            End Property
            Private _CustomerID As String
            Public Property CustomerID() As String
                Get
                    Return _CustomerID
                End Get
                Set(ByVal value As String)
                    _CustomerID = value
                End Set
            End Property
            Private _OrderDate As DateTime
            Public Property OrderDate() As DateTime
                Get
                    Return _OrderDate
                End Get
                Set(ByVal value As DateTime)
                    _OrderDate = value
                End Set
            End Property
            Private _ShippedDate As DateTime
            Public Property ShippedDate() As DateTime
                Get
                    Return _ShippedDate
                End Get
                Set(ByVal value As DateTime)
                    _ShippedDate = value
                End Set
            End Property
            Private _OrderDetails As Lazy(Of OrderDetail())
            Public ReadOnly Property OrderDetails As OrderDetail()
                Get
                    Return _OrderDetails.Value
                End Get
            End Property
        End Class

        Class Customer

            Private _Orders As Lazy(Of Order())
            Public Sub New()
                _Orders = New Lazy(Of Order())(Function() GetOrdersForCustomer(_CustomerID))
            End Sub

            Private _CustomerID As String
            Public Property CustomerID() As String
                Get
                    Return _CustomerID
                End Get
                Set(ByVal value As String)
                    _CustomerID = value
                End Set
            End Property
            Private _CustomerName As String
            Public Property CustomerName() As String
```

```
            Get
                Return _CustomerName
            End Get
            Set(ByVal value As String)
                _CustomerName = value
            End Set
        End Property
        Private _Address As String
        Public Property Address() As String
            Get
                Return _Address
            End Get
            Set(ByVal value As String)
                _Address = value
            End Set
        End Property
        Private _City As String
        Public Property City() As String
            Get
                Return _City
            End Get
            Set(ByVal value As String)
                _City = value
            End Set
        End Property
        Private _PostalCode As String
        Public Property PostalCode() As String
            Get
                Return _PostalCode
            End Get
            Set(ByVal value As String)
                _PostalCode = value
            End Set
        End Property
        Public ReadOnly Property Orders() As Order()
            Get
                Return _Orders.Value
            End Get
        End Property
    End Class

    Class Product
        Private _ProductName As String
        Public Property ProductName() As String
            Get
                Return _ProductName
            End Get
            Set(ByVal value As String)
                _ProductName = value
            End Set
        End Property
        Private _ProductID As Integer
        Public Property ProductID() As Integer
            Get
                Return _ProductID
            End Get
            Set(ByVal value As Integer)
                _ProductID = value
            End Set
        End Property
        Private _UnitPrice As Double
        Public Property UnitPrice() As Double
            Get
                Return _UnitPrice
            End Get
            Set(ByVal value As Double)
                _UnitPrice = value
            End Set
        End Property
```

```
        End Class

        Class OrderDetail
            Private _OrderID As Integer
            Public Property OrderID() As Integer
                Get
                    Return _OrderID
                End Get
                Set(ByVal value As Integer)
                    _OrderID = value
                End Set
            End Property
            Private _ProductID As Integer
            Public Property ProductID() As Integer
                Get
                    Return _ProductID
                End Get
                Set(ByVal value As Integer)
                    _ProductID = value
                End Set
            End Property
            Private _UnitPrice As Double
            Public Property UnitPrice() As Double
                Get
                    Return _UnitPrice
                End Get
                Set(ByVal value As Double)
                    _UnitPrice = value
                End Set
            End Property
            Private _Quantity As Double
            Public Property Quantity() As Double
                Get
                    Return _Quantity
                End Get
                Set(ByVal value As Double)
                    _Quantity = value
                End Set
            End Property
            Private _Discount As Double
            Public Property Discount() As Double
                Get
                    Return _Discount
                End Get
                Set(ByVal value As Double)
                    _Discount = value
                End Set
            End Property


        End Class
#End Region

        Shared Function GetCustomersAsStrings() As IEnumerable(Of String)

            Return IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("CUSTOMERS") = False).
                Skip(1).
                TakeWhile(Function(line) line.StartsWith("END CUSTOMERS") = False)
        End Function
        Shared Function GetCustomers() As IEnumerable(Of Customer)

            Dim customers = IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("CUSTOMERS") = False).
                Skip(1).
                TakeWhile(Function(line) line.StartsWith("END CUSTOMERS") = False)

            Return From line In customers
                   Let fields = line.Split(","c)
```

```
            Select New Customer With {
             .CustomerID = fields(0).Trim(),
             .CustomerName = fields(1).Trim(),
             .Address = fields(2).Trim(),
             .City = fields(3).Trim(),
             .PostalCode = fields(4).Trim()}
End Function

Shared Function GetOrders() As IEnumerable(Of Order)

    Dim orders = IO.File.ReadAllLines("..\..\plinqdata.csv").
                    SkipWhile(Function(line) line.StartsWith("ORDERS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END ORDERS") = False)

    Return From line In orders
           Let fields = line.Split(","c)
           Select New Order With {
            .OrderID = CType(fields(0).Trim(), Integer),
            .CustomerID = fields(1).Trim(),
            .OrderDate = DateTime.Parse(fields(2)),
            .ShippedDate = DateTime.Parse(fields(3))}

End Function

Shared Function GetOrdersForCustomer(ByVal id As String) As Order()

    Dim orders = IO.File.ReadAllLines("..\..\plinqdata.csv").
                    SkipWhile(Function(line) line.StartsWith("ORDERS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END ORDERS") = False)

    Dim orderStrings = From line In orders
                       Let fields = line.Split(","c)
                       Let custID = fields(1).Trim()
                       Where custID = id
                       Select New Order With {
                        .OrderID = CType(fields(0).Trim(), Integer),
                        .CustomerID = custID,
                        .OrderDate = DateTime.Parse(fields(2)),
                        .ShippedDate = DateTime.Parse(fields(3))}

    Return orderStrings.ToArray()

End Function
Shared Function GetProducts() As IEnumerable(Of Product)

    Dim products = IO.File.ReadAllLines("..\..\plinqdata.csv").
        SkipWhile(Function(line) line.StartsWith("PRODUCTS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END PRODUCTS") = False)

    Return From line In products
           Let fields = line.Split(","c)
           Select New Product With {
               .ProductID = CType(fields(0), Integer),
                .ProductName = fields(1).Trim(),
                .UnitPrice = CType(fields(2), Double)}
End Function

Shared Function GetOrderDetailsForOrder(ByVal orderID As Integer) As OrderDetail()
    Dim orderDetails = IO.File.ReadAllLines("..\..\plinqdata.csv").
        SkipWhile(Function(line) line.StartsWith("ORDER DETAILS") = False).
                    Skip(1).
                    TakeWhile(Function(line) line.StartsWith("END ORDER DETAILS") = False)

    Dim ordDetailStrings = From line In orderDetails
                           Let fields = line.Split(","c)
                           Let ordID = Convert.ToInt32(fields(0))
```

```
                                Where ordID = orderID
                                Select New OrderDetail With {
                                 .OrderID = ordID,
                                 .ProductID = CType(fields(1), Integer),
                                 .UnitPrice = CType(fields(2), Double),
                                 .Quantity = CType(fields(3), Double),
                                 .Discount = CType(fields(4), Double)}
            Return ordDetailStrings.ToArray()

        End Function

        Shared Function GetOrderDetails() As IEnumerable(Of OrderDetail)
            Dim orderDetails = IO.File.ReadAllLines("..\..\plinqdata.csv").
                SkipWhile(Function(line) line.StartsWith("ORDER DETAILS") = False).
                            Skip(1).
                            TakeWhile(Function(line) line.StartsWith("END ORDER DETAILS") = False)

            Return From line In orderDetails
                   Let fields = line.Split(","c)
                   Select New OrderDetail With {
                    .OrderID = CType(fields(0), Integer),
                    .ProductID = CType(fields(1), Integer),
                    .UnitPrice = CType(fields(2), Double),
                    .Quantity = CType(fields(3), Double),
                    .Discount = CType(fields(4), Double)}
        End Function

    End Class
```

data

```
CUSTOMERS
ALFKI,Alfreds Futterkiste,Obere Str. 57,Berlin,12209
ANATR,Ana Trujillo Emparedados y helados,Avda. de la Constitución 2222,México D.F.,05021
ANTON,Antonio Moreno Taquería,Mataderos  2312,México D.F.,05023
AROUT,Around the Horn,120 Hanover Sq.,London,WA1 1DP
BERGS,Berglunds snabbköp,Berguvsvägen  8,Luleå,S-958 22
BLAUS,Blauer See Delikatessen,Forsterstr. 57,Mannheim,68306
BLONP,Blondesddsl père et fils,24, place Kléber,Strasbourg,67000
BOLID,Bólido Comidas preparadas,C/ Araquil, 67,Madrid,28023
BONAP,Bon app',12, rue des Bouchers,Marseille,13008
BOTTM,Bottom-Dollar Markets,23 Tsawassen Blvd.,Tsawassen,T2F 8M4
BSBEV,B's Beverages,Fauntleroy Circus,London,EC2 5NT
CACTU,Cactus Comidas para llevar,Cerrito 333,Buenos Aires,1010
CENTC,Centro comercial Moctezuma,Sierras de Granada 9993,México D.F.,05022
CHOPS,Chop-suey Chinese,Hauptstr. 29,Bern,3012
COMMI,Comércio Mineiro,Av. dos Lusíadas, 23,Sao Paulo,05432-043
CONSH,Consolidated Holdings,Berkeley Gardens 12  Brewery,London,WX1 6LT
DRACD,Drachenblut Delikatessen,Walserweg 21,Aachen,52066
DUMON,Du monde entier,67, rue des Cinquante Otages,Nantes,44000
EASTC,Eastern Connection,35 King George,London,WX3 6FW
ERNSH,Ernst Handel,Kirchgasse 6,Graz,8010
FAMIA,Familia Arquibaldo,Rua Orós, 92,Sao Paulo,05442-030
FISSA,FISSA Fabrica Inter. Salchichas S.A.,C/ Moralzarzal, 86,Madrid,28034
FOLIG,Folies gourmandes,184, chaussée de Tournai,Lille,59000
FOLKO,Folk och fä HB,Åkergatan 24,Bräcke,S-844 67
FRANK,Frankenversand,Berliner Platz 43,München,80805
FRANR,France restauration,54, rue Royale,Nantes,44000
FRANS,Franchi S.p.A.,Via Monte Bianco 34,Torino,10100
FURIB,Furia Bacalhau e Frutos do Mar,Jardim das rosas n. 32,Lisboa,1675
GALED,Galería del gastrónomo,Rambla de Cataluña, 23,Barcelona,08022
GODOS,Godos Cocina Típica,C/ Romero, 33,Sevilla,41101
GOURL,Gourmet Lanchonetes,Av. Brasil, 442,Campinas,04876-786
GREAL,Great Lakes Food Market,2732 Baker Blvd.,Eugene,97403
GROSR,GROSELLA-Restaurante,5ª Ave. Los Palos Grandes,Caracas,1081
```

```
HANAR,Hanari Carnes,Rua do Paço, 67,Rio de Janeiro,05454-876
HILAA,HILARION-Abastos,Carrera 22 con Ave. Carlos Soublette #8-35,San Cristóbal,5022
HUNGC,Hungry Coyote Import Store,City Center Plaza 516 Main St.,Elgin,97827
HUNGO,Hungry Owl All-Night Grocers,8 Johnstown Road,Cork,
ISLAT,Island Trading,Garden House Crowther Way,Cowes,PO31 7PJ
KOENE,Königlich Essen,Maubelstr. 90,Brandenburg,14776
LACOR,La corne d'abondance,67, avenue de l'Europe,Versailles,78000
LAMAI,La maison d'Asie,1 rue Alsace-Lorraine,Toulouse,31000
LAUGB,Laughing Bacchus Wine Cellars,1900 Oak St.,Vancouver,V3F 2K1
LAZYK,Lazy K Kountry Store,12 Orchestra Terrace,Walla Walla,99362
LEHMS,Lehmanns Marktstand,Magazinweg 7,Frankfurt a.M.,60528
LETSS,Let's Stop N Shop,87 Polk St. Suite 5,San Francisco,94117
LILAS,LILA-Supermercado,Carrera 52 con Ave. Bolívar #65-98 Llano Largo,Barquisimeto,3508
LINOD,LINO-Delicateses,Ave. 5 de Mayo Porlamar,I. de Margarita,4980
LONEP,Lonesome Pine Restaurant,89 Chiaroscuro Rd.,Portland,97219
MAGAA,Magazzini Alimentari Riuniti,Via Ludovico il Moro 22,Bergamo,24100
MAISD,Maison Dewey,Rue Joseph-Bens 532,Bruxelles,B-1180
END CUSTOMERS

ORDERS
10250,HANAR,7/8/1996 12:00:00 AM,7/12/1996 12:00:00 AM
10253,HANAR,7/10/1996 12:00:00 AM,7/16/1996 12:00:00 AM
10254,CHOPS,7/11/1996 12:00:00 AM,7/23/1996 12:00:00 AM
10257,HILAA,7/16/1996 12:00:00 AM,7/22/1996 12:00:00 AM
10258,ERNSH,7/17/1996 12:00:00 AM,7/23/1996 12:00:00 AM
10263,ERNSH,7/23/1996 12:00:00 AM,7/31/1996 12:00:00 AM
10264,FOLKO,7/24/1996 12:00:00 AM,8/23/1996 12:00:00 AM
10265,BLONP,7/25/1996 12:00:00 AM,8/12/1996 12:00:00 AM
10267,FRANK,7/29/1996 12:00:00 AM,8/6/1996 12:00:00 AM
10275,MAGAA,8/7/1996 12:00:00 AM,8/9/1996 12:00:00 AM
10278,BERGS,8/12/1996 12:00:00 AM,8/16/1996 12:00:00 AM
10279,LEHMS,8/13/1996 12:00:00 AM,8/16/1996 12:00:00 AM
10280,BERGS,8/14/1996 12:00:00 AM,9/12/1996 12:00:00 AM
10283,LILAS,8/16/1996 12:00:00 AM,8/23/1996 12:00:00 AM
10284,LEHMS,8/19/1996 12:00:00 AM,8/27/1996 12:00:00 AM
10289,BSBEV,8/26/1996 12:00:00 AM,8/28/1996 12:00:00 AM
10290,COMMI,8/27/1996 12:00:00 AM,9/3/1996 12:00:00 AM
10296,LILAS,9/3/1996 12:00:00 AM,9/11/1996 12:00:00 AM
10297,BLONP,9/4/1996 12:00:00 AM,9/10/1996 12:00:00 AM
10298,HUNGO,9/5/1996 12:00:00 AM,9/11/1996 12:00:00 AM
10300,MAGAA,9/9/1996 12:00:00 AM,9/18/1996 12:00:00 AM
10303,GODOS,9/11/1996 12:00:00 AM,9/18/1996 12:00:00 AM
10307,LONEP,9/17/1996 12:00:00 AM,9/25/1996 12:00:00 AM
10309,HUNGO,9/19/1996 12:00:00 AM,10/23/1996 12:00:00 AM
10315,ISLAT,9/26/1996 12:00:00 AM,10/3/1996 12:00:00 AM
10317,LONEP,9/30/1996 12:00:00 AM,10/10/1996 12:00:00 AM
10318,ISLAT,10/1/1996 12:00:00 AM,10/4/1996 12:00:00 AM
10321,ISLAT,10/3/1996 12:00:00 AM,10/11/1996 12:00:00 AM
10323,KOENE,10/7/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10325,KOENE,10/9/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10327,FOLKO,10/11/1996 12:00:00 AM,10/14/1996 12:00:00 AM
10328,FURIB,10/14/1996 12:00:00 AM,10/17/1996 12:00:00 AM
10330,LILAS,10/16/1996 12:00:00 AM,10/28/1996 12:00:00 AM
10331,BONAP,10/16/1996 12:00:00 AM,10/21/1996 12:00:00 AM
10335,HUNGO,10/22/1996 12:00:00 AM,10/24/1996 12:00:00 AM
10337,FRANK,10/24/1996 12:00:00 AM,10/29/1996 12:00:00 AM
10340,BONAP,10/29/1996 12:00:00 AM,11/8/1996 12:00:00 AM
10342,FRANK,10/30/1996 12:00:00 AM,11/4/1996 12:00:00 AM
10343,LEHMS,10/31/1996 12:00:00 AM,11/6/1996 12:00:00 AM
10347,FAMIA,11/6/1996 12:00:00 AM,11/8/1996 12:00:00 AM
10350,LAMAI,11/11/1996 12:00:00 AM,12/3/1996 12:00:00 AM
10351,ERNSH,11/11/1996 12:00:00 AM,11/20/1996 12:00:00 AM
10352,FURIB,11/12/1996 12:00:00 AM,11/18/1996 12:00:00 AM
10355,AROUT,11/15/1996 12:00:00 AM,11/20/1996 12:00:00 AM
10357,LILAS,11/19/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10358,LAMAI,11/20/1996 12:00:00 AM,11/27/1996 12:00:00 AM
10360,BLONP,11/22/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10362,BONAP,11/25/1996 12:00:00 AM,11/28/1996 12:00:00 AM
10363,DRACD,11/26/1996 12:00:00 AM,12/4/1996 12:00:00 AM
```

10363,DRACD,11/26/1996 12:00:00 AM,12/4/1996 12:00:00 AM
10364,EASTC,11/26/1996 12:00:00 AM,12/4/1996 12:00:00 AM
10365,ANTON,11/27/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10366,GALED,11/28/1996 12:00:00 AM,12/30/1996 12:00:00 AM
10368,ERNSH,11/29/1996 12:00:00 AM,12/2/1996 12:00:00 AM
10370,CHOPS,12/3/1996 12:00:00 AM,12/27/1996 12:00:00 AM
10371,LAMAI,12/3/1996 12:00:00 AM,12/24/1996 12:00:00 AM
10373,HUNGO,12/5/1996 12:00:00 AM,12/11/1996 12:00:00 AM
10375,HUNGC,12/6/1996 12:00:00 AM,12/9/1996 12:00:00 AM
10378,FOLKO,12/10/1996 12:00:00 AM,12/19/1996 12:00:00 AM
10380,HUNGO,12/12/1996 12:00:00 AM,1/16/1997 12:00:00 AM
10381,LILAS,12/12/1996 12:00:00 AM,12/13/1996 12:00:00 AM
10382,ERNSH,12/13/1996 12:00:00 AM,12/16/1996 12:00:00 AM
10383,AROUT,12/16/1996 12:00:00 AM,12/18/1996 12:00:00 AM
10384,BERGS,12/16/1996 12:00:00 AM,12/20/1996 12:00:00 AM
10386,FAMIA,12/18/1996 12:00:00 AM,12/25/1996 12:00:00 AM
10389,BOTTM,12/20/1996 12:00:00 AM,12/24/1996 12:00:00 AM
10391,DRACD,12/23/1996 12:00:00 AM,12/31/1996 12:00:00 AM
10394,HUNGC,12/25/1996 12:00:00 AM,1/3/1997 12:00:00 AM
10395,HILAA,12/26/1996 12:00:00 AM,1/3/1997 12:00:00 AM
10396,FRANK,12/27/1996 12:00:00 AM,1/6/1997 12:00:00 AM
10400,EASTC,1/1/1997 12:00:00 AM,1/16/1997 12:00:00 AM
10404,MAGAA,1/3/1997 12:00:00 AM,1/8/1997 12:00:00 AM
10405,LINOD,1/6/1997 12:00:00 AM,1/22/1997 12:00:00 AM
10408,FOLIG,1/8/1997 12:00:00 AM,1/14/1997 12:00:00 AM
10410,BOTTM,1/10/1997 12:00:00 AM,1/15/1997 12:00:00 AM
10411,BOTTM,1/10/1997 12:00:00 AM,1/21/1997 12:00:00 AM
10413,LAMAI,1/14/1997 12:00:00 AM,1/16/1997 12:00:00 AM
10414,FAMIA,1/14/1997 12:00:00 AM,1/17/1997 12:00:00 AM
10415,HUNGC,1/15/1997 12:00:00 AM,1/24/1997 12:00:00 AM
10422,FRANS,1/22/1997 12:00:00 AM,1/31/1997 12:00:00 AM
10423,GOURL,1/23/1997 12:00:00 AM,2/24/1997 12:00:00 AM
10425,LAMAI,1/24/1997 12:00:00 AM,2/14/1997 12:00:00 AM
10426,GALED,1/27/1997 12:00:00 AM,2/6/1997 12:00:00 AM
10431,BOTTM,1/30/1997 12:00:00 AM,2/7/1997 12:00:00 AM
10434,FOLKO,2/3/1997 12:00:00 AM,2/13/1997 12:00:00 AM
10436,BLONP,2/5/1997 12:00:00 AM,2/11/1997 12:00:00 AM
10444,BERGS,2/12/1997 12:00:00 AM,2/21/1997 12:00:00 AM
10445,BERGS,2/13/1997 12:00:00 AM,2/20/1997 12:00:00 AM
10449,BLONP,2/18/1997 12:00:00 AM,2/27/1997 12:00:00 AM
10453,AROUT,2/21/1997 12:00:00 AM,2/26/1997 12:00:00 AM
10456,KOENE,2/25/1997 12:00:00 AM,2/28/1997 12:00:00 AM
10457,KOENE,2/25/1997 12:00:00 AM,3/3/1997 12:00:00 AM
10460,FOLKO,2/28/1997 12:00:00 AM,3/3/1997 12:00:00 AM
10464,FURIB,3/4/1997 12:00:00 AM,3/14/1997 12:00:00 AM
10466,COMMI,3/6/1997 12:00:00 AM,3/13/1997 12:00:00 AM
10467,MAGAA,3/6/1997 12:00:00 AM,3/11/1997 12:00:00 AM
10468,KOENE,3/7/1997 12:00:00 AM,3/12/1997 12:00:00 AM
10470,BONAP,3/11/1997 12:00:00 AM,3/14/1997 12:00:00 AM
10471,BSBEV,3/11/1997 12:00:00 AM,3/18/1997 12:00:00 AM
10473,ISLAT,3/13/1997 12:00:00 AM,3/21/1997 12:00:00 AM
10476,HILAA,3/17/1997 12:00:00 AM,3/24/1997 12:00:00 AM
10480,FOLIG,3/20/1997 12:00:00 AM,3/24/1997 12:00:00 AM
10484,BSBEV,3/24/1997 12:00:00 AM,4/1/1997 12:00:00 AM
10485,LINOD,3/25/1997 12:00:00 AM,3/31/1997 12:00:00 AM
10486,HILAA,3/26/1997 12:00:00 AM,4/2/1997 12:00:00 AM
10488,FRANK,3/27/1997 12:00:00 AM,4/2/1997 12:00:00 AM
10490,HILAA,3/31/1997 12:00:00 AM,4/3/1997 12:00:00 AM
10491,FURIB,3/31/1997 12:00:00 AM,4/8/1997 12:00:00 AM
10492,BOTTM,4/1/1997 12:00:00 AM,4/11/1997 12:00:00 AM
10494,COMMI,4/2/1997 12:00:00 AM,4/9/1997 12:00:00 AM
10497,LEHMS,4/4/1997 12:00:00 AM,4/7/1997 12:00:00 AM
10501,BLAUS,4/9/1997 12:00:00 AM,4/16/1997 12:00:00 AM
10507,ANTON,4/15/1997 12:00:00 AM,4/22/1997 12:00:00 AM
10509,BLAUS,4/17/1997 12:00:00 AM,4/29/1997 12:00:00 AM
10511,BONAP,4/18/1997 12:00:00 AM,4/21/1997 12:00:00 AM
10512,FAMIA,4/21/1997 12:00:00 AM,4/24/1997 12:00:00 AM
10519,CHOPS,4/28/1997 12:00:00 AM,5/1/1997 12:00:00 AM
10521,CACTU,4/29/1997 12:00:00 AM,5/2/1997 12:00:00 AM

10522,LEHMS,4/30/1997 12:00:00 AM,5/6/1997 12:00:00 AM
10528,GREAL,5/6/1997 12:00:00 AM,5/9/1997 12:00:00 AM
10529,MAISD,5/7/1997 12:00:00 AM,5/9/1997 12:00:00 AM
10532,EASTC,5/9/1997 12:00:00 AM,5/12/1997 12:00:00 AM
10535,ANTON,5/13/1997 12:00:00 AM,5/21/1997 12:00:00 AM
10538,BSBEV,5/15/1997 12:00:00 AM,5/16/1997 12:00:00 AM
10539,BSBEV,5/16/1997 12:00:00 AM,5/23/1997 12:00:00 AM
10541,HANAR,5/19/1997 12:00:00 AM,5/29/1997 12:00:00 AM
10544,LONEP,5/21/1997 12:00:00 AM,5/30/1997 12:00:00 AM
10550,GODOS,5/28/1997 12:00:00 AM,6/6/1997 12:00:00 AM
10551,FURIB,5/28/1997 12:00:00 AM,6/6/1997 12:00:00 AM
10558,AROUT,6/4/1997 12:00:00 AM,6/10/1997 12:00:00 AM
10568,GALED,6/13/1997 12:00:00 AM,7/9/1997 12:00:00 AM
10573,ANTON,6/19/1997 12:00:00 AM,6/20/1997 12:00:00 AM
10581,FAMIA,6/26/1997 12:00:00 AM,7/2/1997 12:00:00 AM
10582,BLAUS,6/27/1997 12:00:00 AM,7/14/1997 12:00:00 AM
10589,GREAL,7/4/1997 12:00:00 AM,7/14/1997 12:00:00 AM
10600,HUNGC,7/16/1997 12:00:00 AM,7/21/1997 12:00:00 AM
10614,BLAUS,7/29/1997 12:00:00 AM,8/1/1997 12:00:00 AM
10616,GREAL,7/31/1997 12:00:00 AM,8/5/1997 12:00:00 AM
10617,GREAL,7/31/1997 12:00:00 AM,8/4/1997 12:00:00 AM
10621,ISLAT,8/5/1997 12:00:00 AM,8/11/1997 12:00:00 AM
10629,GODOS,8/12/1997 12:00:00 AM,8/20/1997 12:00:00 AM
10634,FOLIG,8/15/1997 12:00:00 AM,8/21/1997 12:00:00 AM
10635,MAGAA,8/18/1997 12:00:00 AM,8/21/1997 12:00:00 AM
10638,LINOD,8/20/1997 12:00:00 AM,9/1/1997 12:00:00 AM
10643,ALFKI,8/25/1997 12:00:00 AM,9/2/1997 12:00:00 AM
10645,HANAR,8/26/1997 12:00:00 AM,9/2/1997 12:00:00 AM
10649,MAISD,8/28/1997 12:00:00 AM,8/29/1997 12:00:00 AM
10652,GOURL,9/1/1997 12:00:00 AM,9/8/1997 12:00:00 AM
10656,GREAL,9/4/1997 12:00:00 AM,9/10/1997 12:00:00 AM
10660,HUNGC,9/8/1997 12:00:00 AM,10/15/1997 12:00:00 AM
10662,LONEP,9/9/1997 12:00:00 AM,9/18/1997 12:00:00 AM
10665,LONEP,9/11/1997 12:00:00 AM,9/17/1997 12:00:00 AM
10677,ANTON,9/22/1997 12:00:00 AM,9/26/1997 12:00:00 AM
10685,GOURL,9/29/1997 12:00:00 AM,10/3/1997 12:00:00 AM
10690,HANAR,10/2/1997 12:00:00 AM,10/3/1997 12:00:00 AM
10692,ALFKI,10/3/1997 12:00:00 AM,10/13/1997 12:00:00 AM
10697,LINOD,10/8/1997 12:00:00 AM,10/14/1997 12:00:00 AM
10702,ALFKI,10/13/1997 12:00:00 AM,10/21/1997 12:00:00 AM
10707,AROUT,10/16/1997 12:00:00 AM,10/23/1997 12:00:00 AM
10709,GOURL,10/17/1997 12:00:00 AM,11/20/1997 12:00:00 AM
10710,FRANS,10/20/1997 12:00:00 AM,10/23/1997 12:00:00 AM
10726,EASTC,11/3/1997 12:00:00 AM,12/5/1997 12:00:00 AM
10729,LINOD,11/4/1997 12:00:00 AM,11/14/1997 12:00:00 AM
10731,CHOPS,11/6/1997 12:00:00 AM,11/14/1997 12:00:00 AM
10734,GOURL,11/7/1997 12:00:00 AM,11/12/1997 12:00:00 AM
10746,CHOPS,11/19/1997 12:00:00 AM,11/21/1997 12:00:00 AM
10753,FRANS,11/25/1997 12:00:00 AM,11/27/1997 12:00:00 AM
10760,MAISD,12/1/1997 12:00:00 AM,12/10/1997 12:00:00 AM
10763,FOLIG,12/3/1997 12:00:00 AM,12/8/1997 12:00:00 AM
10782,CACTU,12/17/1997 12:00:00 AM,12/22/1997 12:00:00 AM
10789,FOLIG,12/22/1997 12:00:00 AM,12/31/1997 12:00:00 AM
10797,DRACD,12/25/1997 12:00:00 AM,1/5/1998 12:00:00 AM
10807,FRANS,12/31/1997 12:00:00 AM,1/30/1998 12:00:00 AM
10819,CACTU,1/7/1998 12:00:00 AM,1/16/1998 12:00:00 AM
10825,DRACD,1/9/1998 12:00:00 AM,1/14/1998 12:00:00 AM
10835,ALFKI,1/15/1998 12:00:00 AM,1/21/1998 12:00:00 AM
10853,BLAUS,1/27/1998 12:00:00 AM,2/3/1998 12:00:00 AM
10872,GODOS,2/5/1998 12:00:00 AM,2/9/1998 12:00:00 AM
10874,GODOS,2/6/1998 12:00:00 AM,2/11/1998 12:00:00 AM
10881,CACTU,2/11/1998 12:00:00 AM,2/18/1998 12:00:00 AM
10887,GALED,2/13/1998 12:00:00 AM,2/16/1998 12:00:00 AM
10892,MAISD,2/17/1998 12:00:00 AM,2/19/1998 12:00:00 AM
10896,MAISD,2/19/1998 12:00:00 AM,2/27/1998 12:00:00 AM
10928,GALED,3/5/1998 12:00:00 AM,3/18/1998 12:00:00 AM
10937,CACTU,3/10/1998 12:00:00 AM,3/13/1998 12:00:00 AM
10952,ALFKI,3/16/1998 12:00:00 AM,3/24/1998 12:00:00 AM
10969,COMMI,3/23/1998 12:00:00 AM,3/30/1998 12:00:00 AM

```
10987,EASTC,3/31/1998 12:00:00 AM,4/6/1998 12:00:00 AM
11026,FRANS,4/15/1998 12:00:00 AM,4/28/1998 12:00:00 AM
11036,DRACD,4/20/1998 12:00:00 AM,4/22/1998 12:00:00 AM
11042,COMMI,4/22/1998 12:00:00 AM,5/1/1998 12:00:00 AM
END ORDERS

ORDER DETAILS
10250,41,7.7000,10,0
10250,51,42.4000,35,0.15
10250,65,16.8000,15,0.15
10253,31,10.0000,20,0
10253,39,14.4000,42,0
10253,49,16.0000,40,0
10254,24,3.6000,15,0.15
10254,55,19.2000,21,0.15
10254,74,8.0000,21,0
10257,27,35.1000,25,0
10257,39,14.4000,6,0
10257,77,10.4000,15,0
10258,2,15.2000,50,0.2
10258,5,17.0000,65,0.2
10258,32,25.6000,6,0.2
10263,16,13.9000,60,0.25
10263,24,3.6000,28,0
10263,30,20.7000,60,0.25
10263,74,8.0000,36,0.25
10264,2,15.2000,35,0
10264,41,7.7000,25,0.15
10265,17,31.2000,30,0
10265,70,12.0000,20,0
10267,40,14.7000,50,0
10267,59,44.0000,70,0.15
10267,76,14.4000,15,0.15
10275,24,3.6000,12,0.05
10275,59,44.0000,6,0.05
10278,44,15.5000,16,0
10278,59,44.0000,15,0
10278,63,35.1000,8,0
10278,73,12.0000,25,0
10279,17,31.2000,15,0.25
10280,24,3.6000,12,0
10280,55,19.2000,20,0
10280,75,6.2000,30,0
10283,15,12.4000,20,0
10283,19,7.3000,18,0
10283,60,27.2000,35,0
10283,72,27.8000,3,0
10284,27,35.1000,15,0.25
10284,44,15.5000,21,0
10284,60,27.2000,20,0.25
10284,67,11.2000,5,0.25
10289,3,8.0000,30,0
10289,64,26.6000,9,0
10290,5,17.0000,20,0
10290,29,99.0000,15,0
10290,49,16.0000,15,0
10290,77,10.4000,10,0
10296,11,16.8000,12,0
10296,16,13.9000,30,0
10296,69,28.8000,15,0
10297,39,14.4000,60,0
10297,72,27.8000,20,0
10298,2,15.2000,40,0
10298,36,15.2000,40,0.25
10298,59,44.0000,30,0.25
10298,62,39.4000,15,0
10300,66,13.6000,30,0
10300,68,10.0000,20,0
10303,40,14.7000,40,0.1
```

```
10303,65,16.8000,30,0.1
10303,68,10.0000,15,0.1
10307,62,39.4000,10,0
10307,68,10.0000,3,0
10309,4,17.6000,20,0
10309,6,20.0000,30,0
10309,42,11.2000,2,0
10309,43,36.8000,20,0
10309,71,17.2000,3,0
10315,34,11.2000,14,0
10315,70,12.0000,30,0
10317,1,14.4000,20,0
10318,41,7.7000,20,0
10318,76,14.4000,6,0
10321,35,14.4000,10,0
10323,15,12.4000,5,0
10323,25,11.2000,4,0
10323,39,14.4000,4,0
10325,6,20.0000,6,0
10325,13,4.8000,12,0
10325,14,18.6000,9,0
10325,31,10.0000,4,0
10325,72,27.8000,40,0
10327,2,15.2000,25,0.2
10327,11,16.8000,50,0.2
10327,30,20.7000,35,0.2
10327,58,10.6000,30,0.2
10328,59,44.0000,9,0
10328,65,16.8000,40,0
10328,68,10.0000,10,0
10330,26,24.9000,50,0.15
10330,72,27.8000,25,0.15
10331,54,5.9000,15,0
10335,2,15.2000,7,0.2
10335,31,10.0000,25,0.2
10335,32,25.6000,6,0.2
10335,51,42.4000,48,0.2
10337,23,7.2000,40,0
10337,26,24.9000,24,0
10337,36,15.2000,20,0
10337,37,20.8000,28,0
10337,72,27.8000,25,0
10340,18,50.0000,20,0.05
10340,41,7.7000,12,0.05
10340,43,36.8000,40,0.05
10342,2,15.2000,24,0.2
10342,31,10.0000,56,0.2
10342,36,15.2000,40,0.2
10342,55,19.2000,40,0.2
10343,64,26.6000,50,0
10343,68,10.0000,4,0.05
10343,76,14.4000,15,0
10347,25,11.2000,10,0
10347,39,14.4000,50,0.15
10347,40,14.7000,4,0
10347,75,6.2000,6,0.15
10350,50,13.0000,15,0.1
10350,69,28.8000,18,0.1
10351,38,210.8000,20,0.05
10351,41,7.7000,13,0
10351,44,15.5000,77,0.05
10351,65,16.8000,10,0.05
10352,24,3.6000,10,0
10352,54,5.9000,20,0.15
10355,24,3.6000,25,0
10355,57,15.6000,25,0
10357,10,24.8000,30,0.2
10357,26,24.9000,16,0
10357,60,27.2000,8,0.2
```

```
10358,24,3.6000,10,0.05
10358,34,11.2000,10,0.05
10358,36,15.2000,20,0.05
10360,28,36.4000,30,0
10360,29,99.0000,35,0
10360,38,210.8000,10,0
10360,49,16.0000,35,0
10360,54,5.9000,28,0
10362,25,11.2000,50,0
10362,51,42.4000,20,0
10362,54,5.9000,24,0
10363,31,10.0000,20,0
10363,75,6.2000,12,0
10363,76,14.4000,12,0
10364,69,28.8000,30,0
10364,71,17.2000,5,0
10365,11,16.8000,24,0
10366,65,16.8000,5,0
10366,77,10.4000,5,0
10368,21,8.0000,5,0.1
10368,28,36.4000,13,0.1
10368,57,15.6000,25,0
10368,64,26.6000,35,0.1
10370,1,14.4000,15,0.15
10370,64,26.6000,30,0
10370,74,8.0000,20,0.15
10371,36,15.2000,6,0.2
10373,58,10.6000,80,0.2
10373,71,17.2000,50,0.2
10375,14,18.6000,15,0
10375,54,5.9000,10,0
10378,71,17.2000,6,0
10380,30,20.7000,18,0.1
10380,53,26.2000,20,0.1
10380,60,27.2000,6,0.1
10380,70,12.0000,30,0
10381,74,8.0000,14,0
10382,5,17.0000,32,0
10382,18,50.0000,9,0
10382,29,99.0000,14,0
10382,33,2.0000,60,0
10382,74,8.0000,50,0
10383,13,4.8000,20,0
10383,50,13.0000,15,0
10383,56,30.4000,20,0
10384,20,64.8000,28,0
10384,60,27.2000,15,0
10386,24,3.6000,15,0
10386,34,11.2000,10,0
10389,10,24.8000,16,0
10389,55,19.2000,15,0
10389,62,39.4000,20,0
10389,70,12.0000,30,0
10391,13,4.8000,18,0
10394,13,4.8000,10,0
10394,62,39.4000,10,0
10395,46,9.6000,28,0.1
10395,53,26.2000,70,0.1
10395,69,28.8000,8,0
10396,23,7.2000,40,0
10396,71,17.2000,60,0
10396,72,27.8000,21,0
10400,29,99.0000,21,0
10400,35,14.4000,35,0
10400,49,16.0000,30,0
10404,26,24.9000,30,0.05
10404,42,11.2000,40,0.05
10404,49,16.0000,30,0.05
10405,3,8.0000,50,0
```

```
10408,37,20.8000,10,0
10408,54,5.9000,6,0
10408,62,39.4000,35,0
10410,33,2.0000,49,0
10410,59,44.0000,16,0
10411,41,7.7000,25,0.2
10411,44,15.5000,40,0.2
10411,59,44.0000,9,0.2
10413,1,14.4000,24,0
10413,62,39.4000,40,0
10413,76,14.4000,14,0
10414,19,7.3000,18,0.05
10414,33,2.0000,50,0
10415,17,31.2000,2,0
10415,33,2.0000,20,0
10422,26,24.9000,2,0
10423,31,10.0000,14,0
10423,59,44.0000,20,0
10425,55,19.2000,10,0.25
10425,76,14.4000,20,0.25
10426,56,30.4000,5,0
10426,64,26.6000,7,0
10431,17,31.2000,50,0.25
10431,40,14.7000,50,0.25
10431,47,7.6000,30,0.25
10434,11,16.8000,6,0
10434,76,14.4000,18,0.15
10436,46,9.6000,5,0
10436,56,30.4000,40,0.1
10436,64,26.6000,30,0.1
10436,75,6.2000,24,0.1
10444,17,31.2000,10,0
10444,26,24.9000,15,0
10444,35,14.4000,8,0
10444,41,7.7000,30,0
10445,39,14.4000,6,0
10445,54,5.9000,15,0
10449,10,24.8000,14,0
10449,52,5.6000,20,0
10449,62,39.4000,35,0
10453,48,10.2000,15,0.1
10453,70,12.0000,25,0.1
10456,21,8.0000,40,0.15
10456,49,16.0000,21,0.15
10457,59,44.0000,36,0
10460,68,10.0000,21,0.25
10460,75,6.2000,4,0.25
10464,4,17.6000,16,0.2
10464,43,36.8000,3,0
10464,56,30.4000,30,0.2
10464,60,27.2000,20,0
10466,11,16.8000,10,0
10466,46,9.6000,5,0
10467,24,3.6000,28,0
10467,25,11.2000,12,0
10468,30,20.7000,8,0
10468,43,36.8000,15,0
10470,18,50.0000,30,0
10470,23,7.2000,15,0
10470,64,26.6000,8,0
10471,7,24.0000,30,0
10471,56,30.4000,20,0
10473,33,2.0000,12,0
10473,71,17.2000,12,0
10476,55,19.2000,2,0.05
10476,70,12.0000,12,0
10480,47,7.6000,30,0
10480,59,44.0000,12,0
10484,21,8.0000,14,0
```

```
10484,40,14.7000,10,0
10484,51,42.4000,3,0
10485,2,15.2000,20,0.1
10485,3,8.0000,20,0.1
10485,55,19.2000,30,0.1
10485,70,12.0000,60,0.1
10486,11,16.8000,5,0
10486,51,42.4000,25,0
10486,74,8.0000,16,0
10488,59,44.0000,30,0
10488,73,12.0000,20,0.2
10490,59,44.0000,60,0
10490,68,10.0000,30,0
10490,75,6.2000,36,0
10491,44,15.5000,15,0.15
10491,77,10.4000,7,0.15
10492,25,11.2000,60,0.05
10492,42,11.2000,20,0.05
10494,56,30.4000,30,0
10497,56,30.4000,14,0
10497,72,27.8000,25,0
10497,77,10.4000,25,0
10501,54,7.4500,20,0
10507,43,46.0000,15,0.15
10507,48,12.7500,15,0.15
10509,28,45.6000,3,0
10511,4,22.0000,50,0.15
10511,7,30.0000,50,0.15
10511,8,40.0000,10,0.15
10512,24,4.5000,10,0.15
10512,46,12.0000,9,0.15
10512,47,9.5000,6,0.15
10512,60,34.0000,12,0.15
10519,10,31.0000,16,0.05
10519,56,38.0000,40,0
10519,60,34.0000,10,0.05
10521,35,18.0000,3,0
10521,41,9.6500,10,0
10521,68,12.5000,6,0
10522,1,18.0000,40,0.2
10522,8,40.0000,24,0
10522,30,25.8900,20,0.2
10522,40,18.4000,25,0.2
10528,11,21.0000,3,0
10528,33,2.5000,8,0.2
10528,72,34.8000,9,0
10529,55,24.0000,14,0
10529,68,12.5000,20,0
10529,69,36.0000,10,0
10532,30,25.8900,15,0
10532,66,17.0000,24,0
10535,11,21.0000,50,0.1
10535,40,18.4000,10,0.1
10535,57,19.5000,5,0.1
10535,59,55.0000,15,0.1
10538,70,15.0000,7,0
10538,72,34.8000,1,0
10539,13,6.0000,8,0
10539,21,10.0000,15,0
10539,33,2.5000,15,0
10539,49,20.0000,6,0
10541,24,4.5000,35,0.1
10541,38,263.5000,4,0.1
10541,65,21.0500,36,0.1
10541,71,21.5000,9,0.1
10544,28,45.6000,7,0
10544,67,14.0000,7,0
10550,17,39.0000,8,0.1
```

```
10550,19,9.2000,10,0
10550,21,10.0000,6,0.1
10550,61,28.5000,10,0.1
10551,16,17.4500,40,0.15
10551,35,18.0000,20,0.15
10551,44,19.4500,40,0
10558,47,9.5000,25,0
10558,51,53.0000,20,0
10558,52,7.0000,30,0
10558,53,32.8000,18,0
10558,73,15.0000,3,0
10568,10,31.0000,5,0
10573,17,39.0000,18,0
10573,34,14.0000,40,0
10573,53,32.8000,25,0
10581,75,7.7500,50,0.2
10582,57,19.5000,4,0
10582,76,18.0000,14,0
10589,35,18.0000,4,0
10600,54,7.4500,4,0
10600,73,15.0000,30,0
10614,11,21.0000,14,0
10614,21,10.0000,8,0
10614,39,18.0000,5,0
10616,38,263.5000,15,0.05
10616,56,38.0000,14,0
10616,70,15.0000,15,0.05
10616,71,21.5000,15,0.05
10617,59,55.0000,30,0.15
10621,19,9.2000,5,0
10621,23,9.0000,10,0
10621,70,15.0000,20,0
10621,71,21.5000,15,0
10629,29,123.7900,20,0
10629,64,33.2500,9,0
10634,7,30.0000,35,0
10634,18,62.5000,50,0
10634,51,53.0000,15,0
10634,75,7.7500,2,0
10635,4,22.0000,10,0.1
10635,5,21.3500,15,0.1
10635,22,21.0000,40,0
10638,45,9.5000,20,0
10638,65,21.0500,21,0
10638,72,34.8000,60,0
10643,28,45.6000,15,0.25
10643,39,18.0000,21,0.25
10643,46,12.0000,2,0.25
10645,18,62.5000,20,0
10645,36,19.0000,15,0
10649,28,45.6000,20,0
10649,72,34.8000,15,0
10652,30,25.8900,2,0.25
10652,42,14.0000,20,0
10656,14,23.2500,3,0.1
10656,44,19.4500,28,0.1
10656,47,9.5000,6,0.1
10660,20,81.0000,21,0
10662,68,12.5000,10,0
10665,51,53.0000,20,0
10665,59,55.0000,1,0
10665,76,18.0000,10,0
10677,26,31.2300,30,0.15
10677,33,2.5000,8,0.15
10685,10,31.0000,20,0
10685,41,9.6500,4,0
10685,47,9.5000,15,0
10690,56,38.0000,20,0.25
10690,77,13.0000,30,0.25
```

```
10692,63,43.9000,20,0
10697,19,9.2000,7,0.25
10697,35,18.0000,9,0.25
10697,58,13.2500,30,0.25
10697,70,15.0000,30,0.25
10702,3,10.0000,6,0
10702,76,18.0000,15,0
10707,55,24.0000,21,0
10707,57,19.5000,40,0
10707,70,15.0000,28,0.15
10709,8,40.0000,40,0
10709,51,53.0000,28,0
10709,60,34.0000,10,0
10710,19,9.2000,5,0
10710,47,9.5000,5,0
10726,4,22.0000,25,0
10726,11,21.0000,5,0
10729,1,18.0000,50,0
10729,21,10.0000,30,0
10729,50,16.2500,40,0
10731,21,10.0000,40,0.05
10731,51,53.0000,30,0.05
10734,6,25.0000,30,0
10734,30,25.8900,15,0
10734,76,18.0000,20,0
10746,13,6.0000,6,0
10746,42,14.0000,28,0
10746,62,49.3000,9,0
10746,69,36.0000,40,0
10753,45,9.5000,4,0
10753,74,10.0000,5,0
10760,25,14.0000,12,0.25
10760,27,43.9000,40,0
10760,43,46.0000,30,0.25
10763,21,10.0000,40,0
10763,22,21.0000,6,0
10763,24,4.5000,20,0
10782,31,12.5000,1,0
10789,18,62.5000,30,0
10789,35,18.0000,15,0
10789,63,43.9000,30,0
10789,68,12.5000,18,0
10797,11,21.0000,20,0
10807,40,18.4000,1,0
10819,43,46.0000,7,0
10819,75,7.7500,20,0
10825,26,31.2300,12,0
10825,53,32.8000,20,0
10835,59,55.0000,15,0
10835,77,13.0000,2,0.2
10853,18,62.5000,10,0
10872,55,24.0000,10,0.05
10872,62,49.3000,20,0.05
10872,64,33.2500,15,0.05
10872,65,21.0500,21,0.05
10874,10,31.0000,10,0
10881,73,15.0000,10,0
10887,25,14.0000,5,0
10892,59,55.0000,40,0.05
10896,45,9.5000,15,0
10896,56,38.0000,16,0
10928,47,9.5000,5,0
10928,76,18.0000,5,0
10937,28,45.6000,8,0
10937,34,14.0000,20,0
10952,6,25.0000,16,0.05
10952,28,45.6000,2,0
10969,46,12.0000,9,0
10987,7,30.0000,60,0
```

```
10987,43,46.0000,6,0
10987,72,34.8000,20,0
11026,18,62.5000,8,0
11026,51,53.0000,10,0
11036,13,6.0000,7,0
11036,59,55.0000,30,0
11042,44,19.4500,15,0
11042,61,28.5000,4,0
END ORDER DETAILS

PRODUCTS
1,Chai,18.0000
2,Chang,19.0000
3,Aniseed Syrup,10.0000
4,Chef Anton's Cajun Seasoning,22.0000
5,Chef Anton's Gumbo Mix,21.3500
6,Grandma's Boysenberry Spread,25.0000
7,Uncle Bob's Organic Dried Pears,30.0000
8,Northwoods Cranberry Sauce,40.0000
9,Mishi Kobe Niku,97.0000
10,Ikura,31.0000
11,Queso Cabrales,21.0000
12,Queso Manchego La Pastora,38.0000
13,Konbu,6.0000
14,Tofu,23.2500
15,Genen Shouyu,15.5000
16,Pavlova,17.4500
17,Alice Mutton,39.0000
18,Carnarvon Tigers,62.5000
19,Teatime Chocolate Biscuits,9.2000
20,Sir Rodney's Marmalade,81.0000
21,Sir Rodney's Scones,10.0000
22,Gustaf's Knäckebröd,21.0000
23,Tunnbröd,9.0000
24,Guaraná Fantástica,4.5000
25,NuNuCa Nuß-Nougat-Creme,14.0000
26,Gumbär Gummibärchen,31.2300
27,Schoggi Schokolade,43.9000
28,Rössle Sauerkraut,45.6000
29,Thüringer Rostbratwurst,123.7900
30,Nord-Ost Matjeshering,25.8900
31,Gorgonzola Telino,12.5000
32,Mascarpone Fabioli,32.0000
33,Geitost,2.5000
34,Sasquatch Ale,14.0000
35,Steeleye Stout,18.0000
36,Inlagd Sill,19.0000
37,Gravad lax,26.0000
38,Côte de Blaye,263.5000
39,Chartreuse verte,18.0000
40,Boston Crab Meat,18.4000
41,Jack's New England Clam Chowder,9.6500
42,Singaporean Hokkien Fried Mee,14.0000
43,Ipoh Coffee,46.0000
44,Gula Malacca,19.4500
45,Rogede sild,9.5000
46,Spegesild,12.0000
47,Zaanse koeken,9.5000
48,Chocolade,12.7500
49,Maxilaku,20.0000
50,Valkoinen suklaa,16.2500
51,Manjimup Dried Apples,53.0000
52,Filo Mix,7.0000
53,Perth Pasties,32.8000
54,Tourtière,7.4500
55,Pâté chinois,24.0000
56,Gnocchi di nonna Alice,38.0000
57,Ravioli Angelo,19.5000
58,Escargots de Bourgogne,13.2500
```

```
59,Raclette Courdavault,55.0000
60,Camembert Pierrot,34.0000
61,Sirop d'érable,28.5000
62,Tarte au sucre,49.3000
63,Vegie-spread,43.9000
64,Wimmers gute Semmelknödel,33.2500
65,Louisiana Fiery Hot Pepper Sauce,21.0500
66,Louisiana Hot Spiced Okra,17.0000
67,Laughing Lumberjack Lager,14.0000
68,Scottish Longbreads,12.5000
69,Gudbrandsdalsost,36.0000
70,Outback Lager,15.0000
71,Flotemysost,21.5000
72,Mozzarella di Giovanni,34.8000
73,Röd Kaviar,15.0000
74,Longlife Tofu,10.0000
75,Rhönbräu Klosterbier,7.7500
76,Lakkalikööri,18.0000
77,Original Frankfurter grüne Soße,13.0000
END PRODUCTS
```

另请参阅

- 并行 LINQ (PLINQ)

用于并行编程的数据结构

2021/11/16 •

.NET 提供了几种对并行编程非常有用的类型，包括一组并发集合类、轻型同步基元和用于迟缓初始化的类型。 可以将这些类型与任何多线程应用代码（包括任务并行库和 PLINQ）结合使用。

并发回收类

System.Collections.Concurrent 命名空间中的回收类提供线程安全的添加和删除操作，以尽可能地避免锁定，并在需要锁定时使用细粒度锁定。并发集合类在访问项时不需要用户代码执行任何锁定。如果多个线程在回收中添加和删除项，并发回收类可以显著提高 System.Collections.ArrayList 和 System.Collections.Generic.List<T>（具有用户实现的锁定）等类型的性能。

下表列出了并发集合类：

System.Collections.Concurrent.BlockingCollection<T>	为实现 System.Collections.Concurrent.IProducerConsumerCollection<T> 的线程安全集合提供阻塞和限制功能。如果没有槽可用或回收已满，阻止制作者线程。如果回收为空，阻止使用者线程。此类型还支持使用者和制作者执行非阻止访问。可以将 BlockingCollection<T> 用作基类或后备存储，以便为支持 IEnumerable<T> 的任何回收类提供阻止和绑定。
System.Collections.Concurrent.ConcurrentBag<T>	提供可缩放的添加和获取操作的线程安全包实现。
System.Collections.Concurrent.ConcurrentDictionary<TKey,TValue>	可缩放的并发字典类型。
System.Collections.Concurrent.ConcurrentQueue<T>	可缩放的并发 FIFO 队列。
System.Collections.Concurrent.ConcurrentStack<T>	可缩放的并发 LIFO 堆栈。

有关详细信息，请参阅线程安全集合。

同步基元

通过消除旧多线程处理代码中高昂的锁定机制，System.Threading 命名空间中的同步基元实现了细粒度并发和更快速的性能。

下表列出了同步类型：

System.Threading.Barrier	通过让每个任务可以在某一点指示自己已到达，并一直阻止到部分或全部任务已到达，让多个线程可以并行处理算法。有关详细信息，请参阅 Barrier。
System.Threading.CountdownEvent	通过提供简单的回收机制，简化分支和联接方案。有关详细信息，请参阅 CountdownEvent。

"	"
System.Threading.ManualResetEventSlim	类似于 System.Threading.ManualResetEvent 的同步基元。虽然 ManualResetEventSlim 是轻型基元，但只能用于进程内通信。
System.Threading.SemaphoreSlim	限制可同时访问资源或资源池的线程数的同步基元。有关详细信息，请参阅 Semaphore 和 SemaphoreSlim。
System.Threading.SpinLock	互斥锁基元，导致尝试获取锁的线程先在循环中等待或旋转一段时间，再生成量程。在应缩短锁等待时间的情况下，SpinLock 的性能优于其他形式的锁定。有关详细信息，请参阅 SpinLock。
System.Threading.SpinWait	小的轻型类型，它会旋转一段指定的时间，并最终将线程置于等待状态(如果超出旋转计数的话)。有关详细信息，请参阅 SpinWait。

有关详细信息，请参阅：

- 如何：使用 SpinLock 进行低级别同步
- 如何：使用屏障同步并发操作。

迟缓初始化类

通过迟缓初始化，除非需要，否则不分配对象内存。迟缓初始化可以提升性能，具体是通过在整个程序生存期内均匀分布对象分配。若要为任何自定义类型启用迟缓初始化，可以包装类型 Lazy<T>。

下表列出了迟缓初始化类型：

"	"
System.Lazy<T>	提供线程安全的轻型迟缓初始化。
System.Threading.ThreadLocal<T>	每线程提供迟缓初始化值，其中每线程迟缓调用初始化函数。
System.Threading.LazyInitializer	提供静态方法，避免出现需要分配专用迟缓初始化实例的情况。相反，它们使用引用是为了确保目标在获得访问时已初始化。

若要了解详细信息，请参阅迟缓初始化

聚合异常

System.AggregateException 类型可用于捕获对各个线程并发抛出的多个异常，并将它们作为一个异常返回给联接线程。为此，System.Threading.Tasks.Task 和 System.Threading.Tasks.Parallel 类型以及 PLINQ 大量使用 AggregateException。有关详细信息，请参阅异常处理和如何：处理 PLINQ 查询中的异常。

请参阅

- System.Collections.Concurrent
- System.Threading
- 并行编程

并行诊断工具

2021/11/16 •

Visual Studio 为调试和分析多线程应用程序提供了广泛的支持。

调试

Visual Studio 调试器添加了用于调试并行应用程序的新窗口。有关详细信息，请参阅下列主题：

- 使用“并行堆栈”窗口
- 使用“任务”窗口
- 演练：调试并行应用程序。

分析

可以利用“并发可视化工具”报告视图直观显示并行程序中的线程如何彼此进行交互，以及如何与系统上其他进程中的线程进行交互。有关详细信息，请参阅并发可视化工具。

另请参阅

- 并行编程

PLINQ 和 TPL 的自定义分区程序

2021/11/16 •

若要并行执行对数据源的操作，关键步骤之一是，将数据源分区 成多个部分，以供多个线程同时访问。PLINQ 和任务并行库 (TPL) 提供了默认分区程序，在用户编写并行查询或 ForEach 循环时透明运行。对于更高级的方案，可以插入自己的分区程序。

分区的种类

对数据源进行分区的方法有很多种。最高效的方法是，多个线程一起协作，共同处理原始源序列，而不是将数据源实际分割成多个子序列。对于长度提前已知的数组和其他索引源（如 IList 集合），范围分区 是最简单的分区种类。每个线程都会收到唯一起始和结束索引，这样就可以处理范围内的数据源，而又不会覆盖其他任何线程或被其他任何线程覆盖。范围分区涉及的唯一开销是，创建范围这项初始工作；之后就无需执行其他任何同步工作了。因此，只要工作负载是均分的，就可以确保实现良好性能。范围分区的缺点是，即使某线程提前完成，也无法帮助其他线程完成工作。

对于长度未知的链接列表或其他集合，可以使用区块分区 。在区块分区中，并行循环或查询中的每个线程或任务都会使用并处理一个区块中的若干源元素，再返回检索其他元素。分区程序可确保所有元素均已分发，且没有重复项。区块可为任意大小。例如，如何：实现动态分区中展示的分区程序创建的区块就只包含一个元素。只要区块不是太大，这类分区就一定会执行负载均衡，因为向线程分配的元素不是预先确定的。不过，每当线程需要获取其他区块时，分区程序就要承担一次同步开销。在这种情况下产生的同步量与区块大小成反比。

通常情况下，范围分区只有在以下情况下才会更快：委托的执行时间为小到中等，数据源有大量元素，且每个分区的工作总量大致相等。因此，大多数情况下，区块分区通常更快。如果数据源有少量元素或委托的执行时间较长，那么区块分区和范围分区的性能大致相同。

TPL 分区程序还支持动态数量的分区。也就是说，可以在 ForEach 循环生成新任务时（举个例子）快速创建分区。借助此功能，分区程序可以与循环本身一起缩放。动态分区程序也一定会执行负载均衡。创建自定义分区程序时，必须支持可通过 ForEach 循环使用的动态分区。

配置 PLINQ 负载均衡分区程序

借助 Partitioner.Create 方法的一些重载，可以为数组或 IList 源创建分区程序，并指定是否应尝试均衡各线程的工作负载。如果分区程序被配置为执行负载均衡，那么使用的就是区块分区，元素会根据请求以小区块的形式分配到每个分区。这种方法有助于确保在整个循环或查询完成前，所有分区都有元素可供处理。附加重载可用于提供任何 IEnumerable 源的负载均衡分区。

负载均衡通常要求分区相对频繁地从分区程序请求获取元素。相比之下，执行静态分区的分区程序可以使用范围分区或区块分区，将元素一次性全部分配给每个分区程序。虽然这样做产生的开销少于负载均衡，但如果一个线程的工作量最终大大多于其他线程，那么执行时间可能就会变长。默认情况下，如果传入的是 IList 或数组，PLINQ 始终都会使用不执行负载均衡的范围分区。若要为 PLINQ 启用负载均衡，请使用 `Partitioner.Create` 方法，如下面的示例所示。

```
// Static partitioning requires indexable source. Load balancing
// can use any IEnumerable.
var nums = Enumerable.Range(0, 100000000).ToArray();

// Create a load-balancing partitioner. Or specify false for static partitioning.
Partitioner<int> customPartitioner = Partitioner.Create(nums, true);

// The partitioner is the query's data source.
var q = from x in customPartitioner.AsParallel()
        select x * Math.PI;

q.ForAll((x) =>
{
    ProcessData(x);
});
```

```
' Static number of partitions requires indexable source.
Dim nums = Enumerable.Range(0, 100000000).ToArray()

' Create a load-balancing partitioner. Or specify false For  Shared partitioning.
Dim customPartitioner = Partitioner.Create(nums, True)

' The partitioner is the query's data source.
Dim q = From x In customPartitioner.AsParallel()
        Select x * Math.PI

q.ForAll(Sub(x) ProcessData(x))
```

在任何给定方案中确定是否使用负载均衡的最佳方式是，在有代表性的负载和计算机配置下，试验并度量操作需要多长时间才能完成。例如，如果是只有几个内核的多核计算机，静态分区可以让速度显著提升；但如果是内核相对较多的计算机，静态分区可能会导致速度降低。

下表列出了 Create 方法的可用重载。这些分区程序不仅限于在 PLINQ 或 Task 中使用。还可用于任何自定义并行构造。

重载	默认负载均衡
Create<TSource>(IEnumerable<TSource>)	Always
Create<TSource>(TSource[], Boolean)	将布尔参数指定为 true 时
Create<TSource>(IList<TSource>, Boolean)	将布尔参数指定为 true 时
Create(Int32, Int32)	Never
Create(Int32, Int32, Int32)	Never
Create(Int64, Int64)	Never
Create(Int64, Int64, Int64)	Never

配置 Parallel.ForEach 静态范围分区程序

在 For 循环中，循环的主体作为委托提供给方法。调用此委托的成本与调用虚拟方法大致相同。在某些情况下，并行循环的主体可能非常小，这就会导致对每个循环迭代调用委托的成本变得十分高昂。在这种情况下，可以使用 Create 重载之一，对数据源元素创建范围分区的 IEnumerable<T>。然后，可以将此范围集合传递到主体包含常规 `for` 循环的 ForEach 方法。这种方法的优势在于，委托调用成本在每个范围内只产生一次，而不是每个元

素都产生一次。下面的示例展示了基本模式。

```
using System;
using System.Collections.Concurrent;
using System.Linq;
using System.Threading;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {

        // Source must be array or IList.
        var source = Enumerable.Range(0, 100000).ToArray();

        // Partition the entire source array.
        var rangePartitioner = Partitioner.Create(0, source.Length);

        double[] results = new double[source.Length];

        // Loop over the partitions in parallel.
        Parallel.ForEach(rangePartitioner, (range, loopState) =>
        {
            // Loop over each range element without a delegate invocation.
            for (int i = range.Item1; i < range.Item2; i++)
            {
                results[i] = source[i] * Math.PI;
            }
        });

        Console.WriteLine("Operation complete. Print results? y/n");
        char input = Console.ReadKey().KeyChar;
        if (input == 'y' || input == 'Y')
        {
            foreach(double d in results)
            {
                Console.Write("{0} ", d);
            }
        }
    }
}
```

```
Imports System.Threading.Tasks
Imports System.Collections.Concurrent

Module PartitionDemo

    Sub Main()
        ' Source must be array or IList.
        Dim source = Enumerable.Range(0, 100000).ToArray()

        ' Partition the entire source array.
        ' Let the partitioner size the ranges.
        Dim rangePartitioner = Partitioner.Create(0, source.Length)

        Dim results(source.Length - 1) As Double

        ' Loop over the partitions in parallel. The Sub is invoked
        ' once per partition.
        Parallel.ForEach(rangePartitioner, Sub(range, loopState)

                                               ' Loop over each range element without a delegate invocation.
                                               For i As Integer = range.Item1 To range.Item2 - 1
                                                   results(i) = source(i) * Math.PI
                                               Next
                                           End Sub)
        Console.WriteLine("Operation complete. Print results? y/n")
        Dim input As Char = Console.ReadKey().KeyChar
        If input = "y"c Or input = "Y"c Then
            For Each d As Double In results
                Console.Write("{0} ", d)
            Next
        End If

    End Sub
End Module
```

循环中的每个线程都会收到自己的 Tuple<T1,T2>，其中包含指定子范围中的起始和结束索引值。内循环 `for` 使用 `fromInclusive` 和 `toExclusive` 值直接循环访问数组或 IList。

借助 Create 重载之一，可以指定分区大小和分区数量。此重载适用于以下情况：每个元素的工作量很少，甚至每个元素调用一个虚拟方法都会对性能产生显著影响。

自定义分区程序

在某些情况下，实现自己的分区程序可能是值得尝试的，或甚至有必要这样做。例如，假设有一个自定义集合类，可以根据对类内部结构的了解情况进行更有效的分区（与默认分区程序相比）。或者，不妨根据对源集合中不同位置的元素处理时长的了解情况，创建不同大小的范围分区。

若要创建基本自定义分区程序，请从 System.Collections.Concurrent.Partitioner<TSource> 派生类，并重写虚拟方法，如下表所述。

GetPartitions	此方法由主线程调用一次，并返回 IList(IEnumerator(TSource))。循环或查询中的每个工作线程都可以对列表调用 `GetEnumerator`，以在不同的分区中检索 IEnumerator<T>。
SupportsDynamicPartitions	如果实现 GetDynamicPartitions，返回 `true`；否则，返回 `false`。

GetDynamicPartitions	如果 SupportsDynamicPartitions 是 `true`，可以视需要选择调用此方法(而不是 GetPartitions)。

如果结果必须可排序，或需要对元素进行索引访问，请从 System.Collections.Concurrent.OrderablePartitioner<TSource> 派生类，并重写虚拟方法，如下表所述。

GetPartitions	此方法由主线程调用一次，并返回 `IList(IEnumerator(TSource))`。循环或查询中的每个工作线程都可以对列表调用 `GetEnumerator`，以在不同的分区中检索 IEnumerator<T>。
SupportsDynamicPartitions	如果实现 GetDynamicPartitions，返回 `true`；否则，返回 false。
GetDynamicPartitions	这通常直接调用 GetOrderableDynamicPartitions。
GetOrderableDynamicPartitions	如果 SupportsDynamicPartitions 是 `true`，可以视需要选择调用此方法(而不是 GetPartitions)。

下表详细介绍了三种负载均衡分区程序是如何实现 OrderablePartitioner<TSource> 类的。

成员	IList/成员(数组)	IList/成员(其他)	IEnumerable
GetOrderablePartitions	使用范围分区	使用更适合列表的区块分区(partitionCount 已指定)	通过创建静态数量的分区来使用区块分区。
OrderablePartitioner<TSource>.GetOrderableDynamicPartitions	抛出不支持异常	使用更适合列表和动态分区的区块分区	通过创建动态数量的分区来使用区块分区。
KeysOrderedInEachPartition	返回 `true`	返回 `true`	返回 `true`
KeysOrderedAcrossPartitions	返回 `true`	返回 `false`	返回 `false`
KeysNormalized	返回 `true`	返回 `true`	返回 `true`
SupportsDynamicPartitions	返回 `false`	返回 `true`	返回 `true`

动态分区

若要在 ForEach 方法中使用分区程序，必须能够返回动态数量的分区。也就是说，分区程序可以在循环执行期间随时按需提供新分区的枚举器。这基本上意味着，每当循环添加新并行任务时，就会请求获取此任务的新分区。如果要求数据必须可排序，请从 System.Collections.Concurrent.OrderablePartitioner<TSource> 派生类，这样就可以为所有分区中的每个项都分配一个唯一索引。

有关详细信息及示例，请参阅如何：实现动态分区。

分区程序合同

实现自定义分区程序时，请遵循以下指南，它们有助于确保与 PLINQ 和 TPL 中的 ForEach 进行正确交互：

- 如果调用 GetPartitions 时 `partitionsCount` 参数值等于或小于零，抛出 ArgumentOutOfRangeException。虽然 PLINQ 和 TPL 绝不会传入等于 0 的 `partitionCount`，但仍建议防范这种可能性。

- GetPartitions 和 GetOrderablePartitions 应始终返回分区的 `partitionsCount` 数。如果分区程序用尽数据，且无法根据请求创建任意多个分区，方法应为剩余的每个分区返回空枚举器。否则，PLINQ 和 TPL 都会抛出 InvalidOperationException。
- GetPartitions、GetOrderablePartitions、GetDynamicPartitions 和 GetOrderableDynamicPartitions 不得返回 `null`（在 Visual Basic 中为 `Nothing`）。如果返回，PLINQ/TPL 会抛出 InvalidOperationException。
- 返回分区的方法应始终返回可完全且唯一枚举数据源的分区。除非分区程序在设计上有特别要求，否则数据源或跳过的项不得有重复项。如果未遵循此规则，输出顺序可能会出现混乱。
- 为了让输出顺序不出现混乱，下面的布尔 Getter 必须始终准确返回以下值：
  - `KeysOrderedInEachPartition`：每个分区返回密钥索引递增的元素。
  - `KeysOrderedAcrossPartitions`：对于返回的所有分区，分区 i 中的密钥索引大于分区 i -1 中的密钥索引。
  - `KeysNormalized`：所有密钥索引从零开始不间断单调递增。
- 所有索引都必须是唯一的。不得有重复索引。如果未遵循此规则，输出顺序可能会出现混乱。
- 所有索引都必须为非负索引。如果未遵循此规则，PLINQ/TPL 可能会抛出异常。

另请参阅

- 并行编程
- 如何：实现动态分区
- 如何：实现静态分区程序

如何：实现动态分区

2021/11/16 •

下面的示例展示了如何实现自定义 System.Collections.Concurrent.OrderablePartitioner<TSource>，以实现动态分区，并可以通过特定重载 ForEach 和 PLINQ 进行使用。

示例

每当分区对枚举器调用 MoveNext，枚举器都会为此分区提供一个列表元素。如果是 PLINQ 和 ForEach，分区是 Task 实例。由于请求在多个线程上并发，因此对当前索引的访问是同步的。

```
//
// An orderable dynamic partitioner for lists
//
using System;
using System.Collections;
using System.Collections.Concurrent;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Xml.Linq;
using System.Numerics;

class OrderableListPartitioner<TSource> : OrderablePartitioner<TSource>
{
    private readonly IList<TSource> m_input;

    // Must override to return true.
    public override bool SupportsDynamicPartitions => true;

    public OrderableListPartitioner(IList<TSource> input) : base(true, false, true) =>
        m_input = input;

    public override IList<IEnumerator<KeyValuePair<long, TSource>>> GetOrderablePartitions(int
partitionCount)
    {
        var dynamicPartitions = GetOrderableDynamicPartitions();
        var partitions =
            new IEnumerator<KeyValuePair<long, TSource>>[partitionCount];

        for (int i = 0; i < partitionCount; i++)
        {
            partitions[i] = dynamicPartitions.GetEnumerator();
        }
        return partitions;
    }

    public override IEnumerable<KeyValuePair<long, TSource>> GetOrderableDynamicPartitions() =>
        new ListDynamicPartitions(m_input);

    private class ListDynamicPartitions : IEnumerable<KeyValuePair<long, TSource>>
    {
        private IList<TSource> m_input;
        private int m_pos = 0;

        internal ListDynamicPartitions(IList<TSource> input) =>
            m_input = input;
```

```
        public IEnumerator<KeyValuePair<long, TSource>> GetEnumerator()
        {
            while (true)
            {
                // Each task gets the next item in the list. The index is
                // incremented in a thread-safe manner to avoid races.
                int elemIndex = Interlocked.Increment(ref m_pos) - 1;

                if (elemIndex >= m_input.Count)
                {
                    yield break;
                }

                yield return new KeyValuePair<long, TSource>(
                    elemIndex, m_input[elemIndex]);
            }
        }

        IEnumerator IEnumerable.GetEnumerator() =>
            ((IEnumerable<KeyValuePair<long, TSource>>)this).GetEnumerator();
    }
}

class ConsumerClass
{
    static void Main()
    {
        var nums = Enumerable.Range(0, 10000).ToArray();
        OrderableListPartitioner<int> partitioner = new OrderableListPartitioner<int>(nums);

        // Use with Parallel.ForEach
        Parallel.ForEach(partitioner, (i) => Console.WriteLine(i));

        // Use with PLINQ
        var query = from num in partitioner.AsParallel()
                    where num % 2 == 0
                    select num;

        foreach (var v in query)
            Console.WriteLine(v);
    }
}
```

```
Imports System.Threading
Imports System.Threading.Tasks
Imports System.Collections.Concurrent

Module Module1
    Public Class OrderableListPartitioner(Of TSource)
        Inherits OrderablePartitioner(Of TSource)


        Private ReadOnly m_input As IList(Of TSource)

        Public Sub New(ByVal input As IList(Of TSource))
            MyBase.New(True, False, True)
            m_input = input
        End Sub

        ' Must override to return true.
        Public Overrides ReadOnly Property SupportsDynamicPartitions As Boolean
            Get
                Return True
            End Get
        End Property

        Public Overrides Function GetOrderablePartitions(ByVal partitionCount As Integer) As IList(Of
```

```
        Public Overrides Function GetOrderablePartitions(ByVal partitionCount As Integer) As IList(Of
IEnumerator(Of KeyValuePair(Of Long, TSource)))
            Dim dynamicPartitions = GetOrderableDynamicPartitions()
            Dim partitions(partitionCount - 1) As IEnumerator(Of KeyValuePair(Of Long, TSource))

            For i = 0 To partitionCount - 1
                partitions(i) = dynamicPartitions.GetEnumerator()
            Next

            Return partitions
        End Function

        Public Overrides Function GetOrderableDynamicPartitions() As IEnumerable(Of KeyValuePair(Of Long,
TSource))
            Return New ListDynamicPartitions(m_input)
        End Function

        Private Class ListDynamicPartitions
            Implements IEnumerable(Of KeyValuePair(Of Long, TSource))

            Private m_input As IList(Of TSource)

            Friend Sub New(ByVal input As IList(Of TSource))
                m_input = input
            End Sub

            Public Function GetEnumerator() As IEnumerator(Of KeyValuePair(Of Long, TSource)) Implements
IEnumerable(Of KeyValuePair(Of Long, TSource)).GetEnumerator
                Return New ListDynamicPartitionsEnumerator(m_input)
            End Function

            Public Function GetEnumerator1() As IEnumerator Implements IEnumerable.GetEnumerator
                Return CType(Me, IEnumerable).GetEnumerator()
            End Function
        End Class

        Private Class ListDynamicPartitionsEnumerator
            Implements IEnumerator(Of KeyValuePair(Of Long, TSource))

            Private m_input As IList(Of TSource)
            Shared m_pos As Integer = 0
            Private m_current As KeyValuePair(Of Long, TSource)

            Public Sub New(ByVal input As IList(Of TSource))
                m_input = input
                m_pos = 0
                Me.disposedValue = False
            End Sub

            Public ReadOnly Property Current As KeyValuePair(Of Long, TSource) Implements IEnumerator(Of
KeyValuePair(Of Long, TSource)).Current
                Get
                    Return m_current
                End Get
            End Property

            Public ReadOnly Property Current1 As Object Implements IEnumerator.Current
                Get
                    Return Me.Current
                End Get
            End Property

            Public Function MoveNext() As Boolean Implements IEnumerator.MoveNext
                Dim elemIndex = Interlocked.Increment(m_pos) - 1
                If elemIndex >= m_input.Count Then
                    Return False
                End If

                m_current = New KeyValuePair(Of Long, TSource)(elemIndex, m_input(elemIndex))
                Return True
```

```
                Return True
            End Function

            Public Sub Reset() Implements IEnumerator.Reset
                m_pos = 0
            End Sub

            Private disposedValue As Boolean ' To detect redundant calls

            Protected Overridable Sub Dispose(ByVal disposing As Boolean)
                If Not Me.disposedValue Then
                    m_input = Nothing
                    m_current = Nothing
                End If
                Me.disposedValue = True
            End Sub

            Public Sub Dispose() Implements IDisposable.Dispose
                Dispose(True)
                GC.SuppressFinalize(Me)
            End Sub

        End Class

    End Class

    Class ConsumerClass

        Shared Sub Main()

            Console.BufferHeight = 20000
            Dim nums = Enumerable.Range(0, 2000).ToArray()

            Dim partitioner = New OrderableListPartitioner(Of Integer)(nums)

            ' Use with Parallel.ForEach
            Parallel.ForEach(partitioner, Sub(i) Console.Write("{0}:{1}  ", i,
Thread.CurrentThread.ManagedThreadId))

            Console.WriteLine("PLINQ ----------------------------------")


            ' create a new partitioner, since Enumerators are not reusable.
            Dim partitioner2 = New OrderableListPartitioner(Of Integer)(nums)
            ' Use with PLINQ
            Dim query = From num In partitioner2.AsParallel()
                        Where num Mod 8 = 0
                        Select num

            For Each v In query
                Console.Write("{0}  ", v)
            Next

            Console.WriteLine("press any key")
            Console.ReadKey()
        End Sub
    End Class

End Module
```

例如，每个区块包含一个元素的区块分区。通过一次提供更多元素，可以减少对锁的争用，并(从理论上说)实现更快的性能。不过，在某个时间点，较大的区块可能需要额外的负载均衡逻辑，才能确保所有线程在全部工作完成前一直处于忙碌状态。

另请参阅

- PLINQ 和 TPL 的自定义分区程序
- 如何：实现静态分区程序

如何：实现静态分区的分区程序

2021/11/16 •

下面的示例展示了一种为执行静态分区的 PLINQ 实现简单自定义分区程序的方法。由于分区程序不支持动态分区，因此无法通过 Parallel.ForEach 使用它。对于每个元素需要越来越多处理时间的数据源，此分区程序可能会让速度提升（与默认范围分区程序相比）。

示例

```
// A static range partitioner for sources that require
// a linear increase in processing time for each succeeding element.
// The range sizes are calculated based on the rate of increase
// with the first partition getting the most elements and the
// last partition getting the least.
class MyPartitioner : Partitioner<int>
{
    int[] source;
    double rateOfIncrease = 0;

    public MyPartitioner(int[] source, double rate)
    {
        this.source = source;
        rateOfIncrease = rate;
    }

    public override IEnumerable<int> GetDynamicPartitions()
    {
        throw new NotImplementedException();
    }

    // Not consumable from Parallel.ForEach.
    public override bool SupportsDynamicPartitions
    {
        get
        {
            return false;
        }
    }

    public override IList<IEnumerator<int>> GetPartitions(int partitionCount)
    {
        List<IEnumerator<int>> _list = new List<IEnumerator<int>>();
        int end = 0;
        int start = 0;
        int[] nums = CalculatePartitions(partitionCount, source.Length);

        for (int i = 0; i < nums.Length; i++)
        {
            start = nums[i];
            if (i < nums.Length - 1)
                end = nums[i + 1];
            else
                end = source.Length;

            _list.Add(GetItemsForPartition(start, end));

            // For demonstratation.
            Console.WriteLine("start = {0} b (end) = {1}", start, end);
        }
        return (IList<IEnumerator<int>>)_list;
    }
```

```
    }
    /*
     *
     *
     *                                                          B
      // Model increasing workloads as a right triangle         /  |
         divided into equal areas along vertical lines.       / |  |
         Each partition  is taller and skinnier              /   |  |
         than the last.                                    / |   |  |
                                                         /   |   |  |
                                                       /     |   |  |
                                                     /  |    |   |  |
                                                   /    |    |   |  |
                                          A     /_____|____|___|__| C
     */
    private int[] CalculatePartitions(int partitionCount, int sourceLength)
    {
        // Corresponds to the opposite side of angle A, which corresponds
        // to an index into the source array.
        int[] partitionLimits = new int[partitionCount];
        partitionLimits[0] = 0;

        // Represent total work as rectangle of source length times "most expensive element"
        // Note: RateOfIncrease can be factored out of equation.
        double totalWork = sourceLength * (sourceLength * rateOfIncrease);
        // Divide by two to get the triangle whose slope goes from zero on the left to "most"
        // on the right. Then divide by number of partitions to get area of each partition.
        totalWork /= 2;
        double partitionArea = totalWork / partitionCount;

        // Draw the next partitionLimit on the vertical coordinate that gives
        // an area of partitionArea * currentPartition.
        for (int i = 1; i < partitionLimits.Length; i++)
        {
            double area = partitionArea * i;

           // Solve for base given the area and the slope of the hypotenuse.
            partitionLimits[i] = (int)Math.Floor(Math.Sqrt((2 * area) / rateOfIncrease));
        }
        return partitionLimits;
    }

    IEnumerator<int> GetItemsForPartition(int start, int end)
    {
        // For demonstration purpsoes. Each thread receives its own enumerator.
        Console.WriteLine("called on thread {0}", Thread.CurrentThread.ManagedThreadId);
        for (int i = start; i < end; i++)
            yield return source[i];
    }
}

class Consumer
{
    public static void Main2()
    {
        var source = Enumerable.Range(0, 10000).ToArray();

        Stopwatch sw = Stopwatch.StartNew();
        MyPartitioner partitioner = new MyPartitioner(source, .5);

        var query = from n in partitioner.AsParallel()
                    select ProcessData(n);

        foreach (var v in query) { }
        Console.WriteLine("Processing time with custom partitioner {0}", sw.ElapsedMilliseconds);

        var source2 = Enumerable.Range(0, 10000).ToArray();

        sw = Stopwatch.StartNew();
```

```
        var query2 = from n in source2.AsParallel()
                     select ProcessData(n);

        foreach (var v in query2) { }
        Console.WriteLine("Processing time with default partitioner {0}", sw.ElapsedMilliseconds);
    }

    // Consistent processing time for measurement purposes.
    static int ProcessData(int i)
    {
        Thread.SpinWait(i * 1000);
        return i;
    }
}
```

此示例中的分区都基于每个元素的处理时间呈线性增加的假设。实际上，按这种方式预测处理时间可能会很困难。如果将静态分区程序与特定数据源结合使用，可以优化源的分区公式，也可以添加负载均衡逻辑，或使用区块分区方法（如如何：实现动态分区中所述）。

另请参阅

- PLINQ 和 TPL 的自定义分区程序

PLINQ 和 TPL 中的 Lambda 表达式

2021/11/16 •

任务并行库 (TPL) 包含许多方法，需要使用 System.Func<TResult> 或 System.Action 系列委托之一作为输入参数。使用这些委托将自定义程序逻辑传入到并行循环、任务或查询中。TPL 以及 PLINQ 的代码示例使用 lambda 表达式以内联代码块的形式创建这些委托的实例。本主题简要介绍 Func 和 Action，并演示如何在任务并行库和 PLINQ 中使用 lambda 表达式。

> **NOTE**
>
> 有关委托的更多常规信息，请参阅委托和委托。有关 C# 和 Visual Basic 中的 lambda 表达式的详细信息，请参阅 Lambda 表达式和 Lambda 表达式。

Func 委托

`Func` 委托封装返回值的一个方法。在 `Func` 签名中，最后或最右侧的类型参数始终指定返回类型。导致编译器错误出现的常见原因之一是，尝试将两个输入参数传入 System.Func<T,TResult>；此类型其实只需要使用一个输入参数。.NET 定义 `Func` 的 17 个版本：System.Func<TResult>、System.Func<T,TResult>、System.Func<T1,T2,TResult>，依此类推直到 System.Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>。

Action 委托

System.Action 委托封装了不返回值的方法（Visual Basic 中的 Sub）。在 `Action` 类型签名中，类型参数仅表示输入参数。与 `Func` 一样，.NET 定义了 `Action` 的 17 个版本，从没有类型参数的版本到具有 16 个类型参数的版本。

示例

下面的 Parallel.ForEach<TSource,TLocal>(IEnumerable<TSource>, Func<TLocal>, Func<TSource,ParallelLoopState,TLocal,TLocal>, Action<TLocal>) 方法示例展示了如何使用 Lambda 表达式表达 Func 和 Action 委托。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

class ForEachWithThreadLocal
{
    // Demonstrated features:
    //   Parallel.ForEach()
    //  Thread-local state
    // Expected results:
    //      This example sums up the elements of an int[] in parallel.
    //      Each thread maintains a local sum. When a thread is initialized, that local sum is set to 0.
    //      On every iteration the current element is added to the local sum.
    //      When a thread is done, it safely adds its local sum to the global sum.
    //      After the loop is complete, the global sum is printed out.
    // Documentation:
    //  http://msdn.microsoft.com/library/dd990270(VS.100).aspx
    static void Main()
    {
        // The sum of these elements is 40.
        int[] input = { 4, 1, 6, 2, 9, 5, 10, 3 };
        int sum = 0;

        try
        {
            Parallel.ForEach(
                    input,             // source collection
                    () => 0,             // thread local initializer
                    (n, loopState, localSum) =>  // body
                    {
                        localSum += n;
                        Console.WriteLine("Thread={0}, n={1}, localSum={2}",
Thread.CurrentThread.ManagedThreadId, n, localSum);
                        return localSum;
                    },
                    (localSum) => Interlocked.Add(ref sum, localSum)     // thread local aggregator
                );

            Console.WriteLine("\nSum={0}", sum);
        }
        // No exception is expected in this example, but if one is still thrown from a task,
        // it will be wrapped in AggregateException and propagated to the main thread.
        catch (AggregateException e)
        {
            Console.WriteLine("Parallel.ForEach has thrown an exception. THIS WAS NOT EXPECTED.\n{0}", e);
        }
    }
}
```

```
Imports System.Threading
Imports System.Threading.Tasks

Module ForEachDemo

    ' Demonstrated features:
    '   Parallel.ForEach()
    '   Thread-local state
    ' Expected results:
    '   This example sums up the elements of an int[] in parallel.
    '   Each thread maintains a local sum. When a thread is initialized, that local sum is set to 0.
    '   On every iteration the current element is added to the local sum.
    '   When a thread is done, it safely adds its local sum to the global sum.
    '   After the loop is complete, the global sum is printed out.
    ' Documentation:
    '   http://msdn.microsoft.com/library/dd990270(VS.100).aspx
    Private Sub ForEachDemo()
        ' The sum of these elements is 40.
        Dim input As Integer() = {4, 1, 6, 2, 9, 5, _
        10, 3}
        Dim sum As Integer = 0

        Try
            ' source collection
            Parallel.ForEach(input,
                             Function()
                                 ' thread local initializer
                                 Return 0
                             End Function,
                             Function(n, loopState, localSum)
                                 ' body
                                 localSum += n
                                 Console.WriteLine("Thread={0}, n={1}, localSum={2}",
Thread.CurrentThread.ManagedThreadId, n, localSum)
                                 Return localSum
                             End Function,
                             Sub(localSum)
                                 ' thread local aggregator
                                 Interlocked.Add(sum, localSum)
                             End Sub)

            Console.WriteLine(vbLf & "Sum={0}", sum)
        Catch e As AggregateException
            ' No exception is expected in this example, but if one is still thrown from a task,
            ' it will be wrapped in AggregateException and propagated to the main thread.
            Console.WriteLine("Parallel.ForEach has thrown an exception. THIS WAS NOT EXPECTED." & vbLf & "
{0}", e)
        End Try
    End Sub


End Module
```

另请参阅

- 并行编程

其他阅读材料（并行编程）

2021/11/16 •

下面的资源详细介绍了 .NET 中的并行编程：

- 并行编程模式：通过 .NET Framework 4 了解和应用并行模式一文介绍了利用这些模式开发并行组件的常见并行模式和最佳做法。
- 《多核体系结构的分解和协调设计模式》一书介绍了利用 .NET Framework 4 中引入的并行编程支持的并行编程模式。
- 使用 .NET 并行编程博客包含许多深入介绍了 .NET 中并行编程的文章。
- 使用 .NET Core 和 .NET Standard 并行编程的示例页面包含许多示例，展示了中级和高级并行编程技巧。

请参阅

- 并行计算开发人员中心
- Visual C++ 中的并行编程

托管线程

2021/11/16 •

无论是要为具有一个还是多个处理器的计算机进行开发，你都希望应用程序能够提供响应最为迅速的用户交互，即使应用程序当前正在执行其他操作，也不例外。使用多线程执行是让应用程序一直迅速响应用户的最有效方式，同时也是在用户事件之间或在用户事件期间使用处理器的最有效方式。虽然本部分介绍的是线程基本概念，但将会重点介绍托管线程概念和如何使用托管线程。

> **NOTE**
>
> 自 .NET Framework 4 起，由于出现了 System.Threading.Tasks.Parallel 和 System.Threading.Tasks.Task 类、并行 LINQ (PLINQ)、System.Collections.Concurrent 命名空间中的并发集合类以及基于任务（而非线程）概念的编程模型，多线程编程大大得到了简化。有关详细信息，请参阅并行编程。

本节内容

托管线程处理基本知识
概述了托管线程以及何时使用多线程。

使用线程和线程处理
介绍了如何创建、启动、暂停、恢复和中止线程。

托管线程处理的最佳做法
介绍了同步级别、如何避免死锁和争用条件，以及其他线程问题。

线程处理对象和功能
介绍了可用于同步在不同线程上访问的线程活动和对象数据的托管类，并概述了线程池线程。

参考

System.Threading
收录了用于使用和同步托管线程的类。

System.Collections.Concurrent
收录了可安全用于多线程的集合类。

System.Threading.Tasks
收录了用于创建和计划并发处理任务的类。

相关章节

应用程序域
概述了应用程序域及其在公共语言基础结构中的应用。

异步文件 I/O
描述异步 I/O 的性能优势和基本操作。

基于任务的异步模式 (TAP)
概述了推荐的 .NET 异步编程模式。

使用异步方式调用同步方法
介绍了如何使用委托的内置功能对线程池线程调用方法。

并行编程

介绍了并行编程库，其简化了在应用程序中使用多线程。

并行 LINQ (PLINQ)

介绍了为利用多个处理器而并行运行查询的系统。

在 .NET 中测试

2021/11/16 •

本文介绍了测试概念，并说明了如何使用不同类型的测试来验证代码。有多种工具可用于测试 .NET 应用程序，如 .NET CLI 或集成开发环境 (IDE)。

测试类型

使用自动测试是确保应用程序代码按作者期望执行操作的一种绝佳方式。本文介绍了单元测试、集成测试和负载测试。

单元测试

单元测试是一种试验单个软件组件或方法（也称为“工作单元”）的测试。单元测试仅应测试开发人员控件内的代码。它们不测试基础结构问题。基础结构问题包括与数据库、文件系统和网络资源的交互。

有关创建单元测试的详细信息，请参阅测试工具。

集成测试

集成测试 与单元测试的不同之处在于，它试验两个或更多软件组件一同工作（也称为其“集成”）的能力。这些测试在更广泛范围的受测系统上运行，而单元测试则侧重于单个组件。通常，集成测试会包括对基础结构问题的测试。

负载测试

负载测试旨在确定系统是否可以处理指定的负载，例如，使用应用程序的并发用户数和应用程序响应性处理交互的能力。有关 Web 应用程序负载测试的详细信息，请参阅 ASP.NET Core 负载/压力测试。

测试注意事项

请记住，可以使用编写测试的最佳做法。例如，测试驱动开发 (TDD) 是指先编写单元测试，再编写该单元测试要检查的代码。TDD 就像先编写书籍大纲，再编写该书籍。它旨在帮助开发人员编写更简单、更具可读性的高效代码。

测试工具

.NET 是一个多语言开发平台，可以为 C#、F# 和 Visual Basic 编写各种测试类型。对于每种语言，可以在几个测试框架中进行选择。

xUnit

xUnit 是一个适用于 .NET 的免费、开源、面向社区的单元测试工具。xUnit.net 由 NUnit v2 的原发明者编写，是针对单元测试 .NET 应用的最新技术。xUnit.net 适用于 ReSharper、CodeRush、TestDriven.NET 和 Xamarin。它是 .NET Foundation 的项目，并在其行为准则下运行。

有关更多信息，请参见以下资源：

- 使用 C# 执行单元测试
- 使用 F# 执行单元测试
- 使用 Visual Basic 执行单元测试

NUnit

NUnit 是适用于所有 .NET 语言的单元测试框架。最初从 JUnit 移植而来，当前生产版本已被重写，添加了许多新功能和对各种 .NET 平台的支持。它是 .NET Foundation 的项目。

有关更多信息，请参见以下资源：

- 使用 C# 执行单元测试
- 使用 F# 执行单元测试
- 使用 Visual Basic 执行单元测试

MSTest

MSTest 是适用于所有 .NET 语言的 Microsoft 测试框架。它可以扩展，并且适用于 .NET CLI 和 Visual Studio。有关更多信息，请参见以下资源：

- 使用 C# 执行单元测试
- 使用 F# 执行单元测试
- 使用 Visual Basic 执行单元测试

.NET CLI

你可以使用 dotnet test 命令，从 .NET CLI 运行解决方案单元测试。.NET CLI 公开了 集成开发环境 (IDE) 通过用户界面提供的大部分功能。.NET CLI 是跨平台的，可作为持续集成和交付管道的一部分使用。.NET CLI 用于脚本化进程，以自动执行常见任务。

IDE

无论使用的是 Visual Studio、Visual Studio for Mac 还是 Visual Studio Code，都有用于测试功能的图形用户界面。IDE 提供了比 CLI 更多的功能，例如 Live Unit Testing。有关详细信息，请参阅 Visual Studio 中的包含与排除测试。

另请参阅

有关详细信息，请参阅以下文章：

- 使用 .NET 执行单元测试的最佳实践
- ASP.NET Core 中的集成测试
- 运行选择性单元测试
- 将代码覆盖率用于单元测试

.NET Core 和 .NET Standard 单元测试最佳做法

2021/11/16 •

编写单元测试有许多好处;它们有助于回归、提供文档和促进良好的设计。然而，难懂且脆弱的单元测试会对代码库造成严重破坏。本文介绍一些有关 .NET Core 和 .NET Standard 项目的单元测试设计的最佳做法。

本指南将介绍一些在编写单元测试时的最佳做法，使测试具有弹性且易于理解。

作者是 John Reese 且特别感谢 Roy Osherove

为什么要执行单元测试？

比执行功能测试节省时间

功能测试费用高。它们通常涉及打开应用程序并执行一系列你(或其他人)必须遵循的步骤，以验证预期的行为。测试人员可能并不总是了解这些步骤，这意味着为了执行测试，他们必须联系更熟悉该领域的人。对于细微更改，测试本身可能需要几秒钟，对于较大更改，可能需要几分钟。最后，在系统中所做的每项更改都必须重复此过程。

而单元测试只需按一下按钮即可运行，只需要几毫秒时间，且无需测试人员了解整个系统。测试通过与否取决于测试运行程序，而非测试人员。

防止回归

回归缺陷是在对应用程序进行更改时引入的缺陷。测试人员通常不仅测试新功能，还要测试预先存在的功能，以验证先前实现的功能是否仍按预期运行。

使用单元测试，可在每次生成后，甚至在更改一行代码后重新运行整套测试。让你确信新代码不会破坏现有功能。

可执行文档

在给定某个输入的情况下，特定方法的作用或行为可能并不总是很明显。你可能会想知道:如果我将空白字符串传递给它，此方法会有怎样的行为？ Null?

如果你有一套命名正确的单元测试，每个测试应能够清楚地解释给定输入的预期输出。此外，它应该能够验证其确实有效。

减少耦合代码

当代码紧密耦合时，可能难以进行单元测试。如果不为编写的代码创建单元测试，则耦合可能不太明显。

为代码编写测试会自然地解耦代码，因为采用其他方法测试会更困难。

优质单元测试的特征

- 快速。对成熟项目进行数千次单元测试，这很常见。应花非常少的时间来运行单元测试。几毫秒。
- 独立。单元测试是独立的，可以单独运行，并且不依赖文件系统或数据库等任何外部因素。
- 可重复。运行单元测试的结果应该保持一致，也就是说，如果在运行期间不更改任何内容，总是返回相同的结果。
- 自检查。测试应该能够在没有任何人工交互的情况下，自动检测测试是否通过。
- 及时。与要测试的代码相比，编写单元测试不应花费过多不必要的时间。如果发现测试代码与编写代码相比需要花费大量的时间，请考虑一种更易测试的设计。

代码覆盖率

高代码覆盖率百分比通常与较高的代码质量相关联。但该度量值本身无法确定代码的质量。设置过高的代码覆盖率百分比目标可能会适得其反。假设一个复杂的项目有数千个条件分支，并且假设你设定了一个 95% 代码覆盖率的目标。该项目当前维保持 90% 的代码覆盖率。要覆盖剩余 5% 的所有边缘事例，需要花费巨大的工作量，而且价值主张会迅速降低。

高代码覆盖率百分比不指示成功，也不意味着高代码质量。它仅仅表示单元测试所涵盖的代码量。有关详细信息，请参阅单元测试代码覆盖率。

让我们使用相同的术语

遗憾的是，当谈到测试时，术语“mock”经常被滥用。以下几点定义了编写单元测试时最常见的 fake 类型：

Fake - Fake 是一个通用术语，可用于描述 stub 或 mock 对象。它是 stub 还是 mock 取决于使用它的上下文。也就是说，Fake 可以是 stub 或 mock。

Mock - Mock 对象是系统中的 fake 对象，用于确定单元测试是否通过。Mock 起初为 Fake，直到对其断言。

Stub - Stub 是系统中现有依赖项（或协作者）的可控制替代项。通过使用 Stub，可以在无需使用依赖项的情况下直接测试代码。默认情况下，存根起初为 fake。

请思考以下代码片段：

```
var mockOrder = new MockOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

这是 Stub 被引用为 mock 的一个示例。在本例中，它是 Stub。只是将 Order 作为实例化 `Purchase`（被测系统）的一种方法传递。名称 `MockOrder` 也具有误导性，因为同样地 order 不是 mock。

更好的方法是

```
var stubOrder = new FakeOrder();
var purchase = new Purchase(stubOrder);

purchase.ValidateOrders();

Assert.True(purchase.CanBeShipped);
```

通过将类重命名为 `FakeOrder`，使类更通用，类可以用作 mock 或 stub。以更适合测试用例者为准。在上述示例中，`FakeOrder` 用作 stub。在断言期间，没有以任何形状或形式使用 `FakeOrder`。`FakeOrder` 传递到 `Purchase` 类，以满足构造函数的要求。

要将其用作 Mock，可执行如下操作

```
var mockOrder = new FakeOrder();
var purchase = new Purchase(mockOrder);

purchase.ValidateOrders();

Assert.True(mockOrder.Validated);
```

在这种情况下，检查 Fake 上的属性（对其进行断言），因此在以上代码片段中，`mockOrder` 是 Mock。

> **IMPORTANT**
> 正确理解此术语至关重要。如果将 stub 称为 mock，其他开发人员会对你的意图做出错误的判断。

关于 mock 与 stub 需要注意的一个重点是，mock 与 stub 很像，但可以针对 mock 对象进行断言，而不针对 stub 进行断言。

最佳实践

编写单元测试时，尽量不要引入基础结构依赖项。这些依赖项会降低测试速度，使测试更加脆弱，应将其保留供集成测试使用。可以通过遵循 Explicit Dependencies Principle（显式依赖项原则）和使用 Dependency Injection（依赖项注入）避免应用程序中的这些依赖项。还可以将单元测试保留在单独的项目中，与集成测试相分隔。这可确保单元测试项目没有引用或依赖于基础结构包。

为测试命名

测试的名称应包括三个部分：

- 要测试的方法的名称。
- 测试的方案。
- 调用方案时的预期行为。

为什么？

- 命名标准非常重要，因为它们明确地表达了测试的意图。

测试不仅能确保代码有效，还能提供文档。只需查看单元测试套件，就可以在不查看代码本身的情况下推断代码的行为。此外，测试失败时，可以确切地看到哪些方案不符合预期。

不佳：

```
[Fact]
public void Test_Single()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

良好：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

安排测试

“Arrange、Act、Assert”是单元测试时的常见模式。顾名思义，它包含三个主要操作：

- 安排对象，根据需要对其进行创建和设置。
- 作用于对象。
- 断言某些项按预期进行。

为什么？

- 明确地将要测试的内容从“arrange”和“assert”步骤分开 。
- 降低将断言与“Act”代码混杂的可能性。

可读性是编写测试时最重要的方面之一。在测试中分离这些操作会明确地突出显示调用代码所需的依赖项、调用代码的方式以及尝试断言的内容。虽然可以组合一些步骤并减小测试的大小，但主要目标是使测试尽可能可读。

不佳：

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Assert
    Assert.Equal(0, stringCalculator.Add(""));
}
```

良好：

```
[Fact]
public void Add_EmptyString_ReturnsZero()
{
    // Arrange
    var stringCalculator = new StringCalculator();

    // Act
    var actual = stringCalculator.Add("");

    // Assert
    Assert.Equal(0, actual);
}
```

以最精简方式编写通过测试

单元测试中使用的输入应为最简单的输入，以便验证当前正在测试的行为。

为什么？

- 测试对代码库的未来更改更具弹性。
- 更接近于测试行为而非实现。

包含比通过测试所需信息更多信息的测试更可能将错误引入测试，并且可能使测试的意图变得不太明确。编写测试时需要将重点放在行为上。在模型上设置额外的属性或在不需要时使用非零值，只会偏离所要证明的内容。

不佳：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("42");

    Assert.Equal(42, actual);
}
```

良好：

```
[Fact]
public void Add_SingleNumber_ReturnsSameNumber()
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add("0");

    Assert.Equal(0, actual);
}
```

避免魔幻字符串

单元测试中的命名变量和生产代码中的命名变量同样重要。单元测试不应包含魔幻字符串。

为什么？

- 测试读者无需检查生产代码即可了解值的特殊之处。
- 明确地显示所要证明的内容，而不是显示要完成的内容 。

魔幻字符串可能会让测试读者感到困惑。如果字符串看起来不寻常，他们可能想知道为什么为某个参数或返回值选择了某个值。这可能会使他们仔细查看实现细节，而不是专注于测试。

> **TIP**
> 编写测试时，应力求表达尽可能多的意图。对于魔幻字符串，一种很好的方法是将这些值赋给常量。

不佳:

```
[Fact]
public void Add_BigNumber_ThrowsException()
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add("1001");

    Assert.Throws<OverflowException>(actual);
}
```

良好:

```
[Fact]
void Add_MaximumSumResult_ThrowsOverflowException()
{
    var stringCalculator = new StringCalculator();
    const string MAXIMUM_RESULT = "1001";

    Action actual = () => stringCalculator.Add(MAXIMUM_RESULT);

    Assert.Throws<OverflowException>(actual);
}
```

在测试中应避免逻辑

编写单元测试时，请避免手动字符串串联和逻辑条件，例如 `if`、`while`、`for` 和 `switch` 等等。

为什么？

- 降低在测试中引入 bug 的可能性。
- 专注于最终结果，而不是实现细节。

将逻辑引入测试套件中时，引入 bug 的可能性大幅度增加。你最不希望测试套件中出现 bug。你应该对测试工作充满高度的自信，否则你将不会信任它们。你不信任的测试不会带来任何价值。当测试失败时，你希望意识到

代码存在问题且无法忽略该问题。

> **TIP**
> 如果不可避免地要在测试中使用逻辑，请考虑将测试分成两个或多个不同的测试。

不佳：

```
[Fact]
public void Add_MultipleNumbers_ReturnsCorrectResults()
{
    var stringCalculator = new StringCalculator();
    var expected = 0;
    var testCases = new[]
    {
        "0,0,0",
        "0,1,2",
        "1,2,3"
    };

    foreach (var test in testCases)
    {
        Assert.Equal(expected, stringCalculator.Add(test));
        expected += 3;
    }
}
```

良好：

```
[Theory]
[InlineData("0,0,0", 0)]
[InlineData("0,1,2", 3)]
[InlineData("1,2,3", 6)]
public void Add_MultipleNumbers_ReturnsSumOfNumbers(string input, int expected)
{
    var stringCalculator = new StringCalculator();

    var actual = stringCalculator.Add(input);

    Assert.Equal(expected, actual);
}
```

更偏好 helper 方法而非 setup 和 teardown

如果测试需要类似的对象或状态，那么比起使用 `Setup` 和 `Teardown` 属性（如果存在），更偏好使用 helper 方法。

为什么？

- 读者阅读测试时产生的困惑减少，因为每个测试中都可以看到所有代码。
- 给定测试的设置过多或过少的可能性降低。
- 在测试之间共享状态（这会在测试之间创建不需要的依赖项）的可能性降低。

在单元测试框架中，在测试套件的每个单元测试之前调用 `Setup`。虽然有些人可能会将其视为有用的工具，但它通常最终导致庞大且难懂的测试。每个测试通常有不同的要求，以使测试启动并运行。遗憾的是，`Setup` 迫使你对每个测试使用完全相同的要求。

> **NOTE**
> 自版本 2.x 起，xUnit 已删除 SetUp 和 TearDown

不佳:

```
private readonly StringCalculator stringCalculator;
public StringCalculatorTests()
{
    stringCalculator = new StringCalculator();
}
```

```
// more tests...
```

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var result = stringCalculator.Add("0,1");

    Assert.Equal(1, result);
}
```

良好:

```
[Fact]
public void Add_TwoNumbers_ReturnsSumOfNumbers()
{
    var stringCalculator = CreateDefaultStringCalculator();

    var actual = stringCalculator.Add("0,1");

    Assert.Equal(1, actual);
}
```

```
// more tests...
```

```
private StringCalculator CreateDefaultStringCalculator()
{
    return new StringCalculator();
}
```

避免多个断言

在编写测试时，请尝试每次测试只包含一个 Assert。仅使用一个 assert 的常用方法包括：

- 为每个 assert 创建单独的测试。
- 使用参数化测试。

为什么？

- 如果一个 Assert 失败，将不计算后续 Assert。
- 确保在测试中没有断言多个事例。
- 让你从整体上了解测试失败原因。

将多个断言引入测试用例时，不能保证所有断言都会执行。在大多数单元测试框架中，一旦断言在单元测试中失败，则进行中的测试会自动被视为失败。这可能会令人困惑，因为正在运行的功能将显示为失败。

> **NOTE**
>
> 此规则的一个常见例外是对对象进行断言。在这种情况下，通常可以对每个属性进行多次断言，以确保对象处于所预期的状态。

不佳：

```
[Fact]
public void Add_EdgeCases_ThrowsArgumentExceptions()
{
    Assert.Throws<ArgumentException>(() => stringCalculator.Add(null));
    Assert.Throws<ArgumentException>(() => stringCalculator.Add("a"));
}
```

良好：

```
[Theory]
[InlineData(null)]
[InlineData("a")]
public void Add_InputNullOrAlphabetic_ThrowsArgumentException(string input)
{
    var stringCalculator = new StringCalculator();

    Action actual = () => stringCalculator.Add(input);

    Assert.Throws<ArgumentException>(actual);
}
```

通过单元测试公共方法验证专有方法

在大多数情况下，不需要测试专用方法。专用方法是实现细节。可以这样认为：专用方法永远不会孤立存在。在某些时候，存在调用专用方法作为其实现的一部分的面向公共的方法。你应关心的是调用到专用方法的公共方法的最终结果。

请考虑下列情形

```
public string ParseLogLine(string input)
{
    var sanitizedInput = TrimInput(input);
    return sanitizedInput;
}

private string TrimInput(string input)
{
    return input.Trim();
}
```

你的第一反应可能是开始为 `TrimInput` 编写测试，因为想要确保该方法按预期工作。但是，`ParseLogLine` 完全有可能以一种你所不期望的方式操纵 `sanitizedInput`，使得对 `TrimInput` 的测试变得毫无用处。

真正的测试应该针对面向公共的方法 `ParseLogLine` 进行，因为这是你最终应该关心的。

```
public void ParseLogLine_StartsAndEndsWithSpace_ReturnsTrimmedResult()
{
    var parser = new Parser();

    var result = parser.ParseLogLine(" a ");

    Assert.Equals("a", result);
}
```

由此，如果看到一个专用方法，可以找到公共方法并针对该方法编写测试。不能仅仅因为专用方法返回预期结果就认为最终调用专用方法的系统正确地使用结果。

Stub 静态引用

单元测试的原则之一是其必须完全控制被测试的系统。当生产代码包含对静态引用（例如 `DateTime.Now`）的调用时，这可能会存在问题。考虑下列代码

```
public int GetDiscountedPrice(int price)
{
    if (DateTime.Now.DayOfWeek == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

如何对此代码进行单元测试？ 可以尝试一种方法，例如

```
public void GetDiscountedPrice_NotTuesday_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(2, actual)
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();

    var actual = priceCalculator.GetDiscountedPrice(2);

    Assert.Equals(1, actual);
}
```

遗憾的是，你会很快意识到你的测试存在一些问题。

- 如果在星期二运行测试套件，则第二个测试将通过，但第一个测试将失败。
- 如果在任何其他日期运行测试套件，则第一个测试将通过，但第二个测试将失败。

要解决这些问题，需要将“seam”引入生产代码中。一种方法是在接口中包装需要控制的代码，并使生产代码依赖于该接口。

```
public interface IDateTimeProvider
{
    DayOfWeek DayOfWeek();
}

public int GetDiscountedPrice(int price, IDateTimeProvider dateTimeProvider)
{
    if (dateTimeProvider.DayOfWeek() == DayOfWeek.Tuesday)
    {
        return price / 2;
    }
    else
    {
        return price;
    }
}
```

你的测试套件现在变得

```
public void GetDiscountedPrice_NotTuesday_ReturnsFullPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Monday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(2, actual);
}

public void GetDiscountedPrice_OnTuesday_ReturnsHalfPrice()
{
    var priceCalculator = new PriceCalculator();
    var dateTimeProviderStub = new Mock<IDateTimeProvider>();
    dateTimeProviderStub.Setup(dtp => dtp.DayOfWeek()).Returns(DayOfWeek.Tuesday);

    var actual = priceCalculator.GetDiscountedPrice(2, dateTimeProviderStub);

    Assert.Equals(1, actual);
}
```

现在，测试套件可以完全控制 `DateTime.Now`，并且在调用方法时可以存根任何值。

使用 dotnet test 和 xUnit 在 .NET Core 中进行 C# 单元测试

2021/11/16 •

本教程演示如何生成包含单元测试项目和源代码项目的解决方案。若要使用预构建解决方案学习本教程，请查看或下载示例代码。有关下载说明，请参阅示例和教程。

创建解决方案

在本部分中，将创建包含源和测试项目的解决方案。已完成的解决方案具有以下目录结构：

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        PrimeService.cs
        PrimeService.csproj
    /PrimeService.Tests
        PrimeService_IsPrimeShould.cs
        PrimeServiceTests.csproj
```

以下说明提供了创建测试解决方案的步骤。有关通过一个步骤创建测试解决方案的说明，请参阅用于创建测试解决方案的命令。

- 打开 shell 窗口。
- 运行下面的命令：

  ```
  dotnet new sln -o unit-testing-using-dotnet-test
  ```

  `dotnet new sln` 命令用于在 unit-testing-using-dotnet-test 目录中创建新的解决方案。
- 将目录更改为 unit-testing-using-dotnet-test 文件夹。
- 运行下面的命令：

  ```
  dotnet new classlib -o PrimeService
  ```

  `dotnet new classlib` 命令用于在 PrimeService 文件夹中创建新的类库项目。新的类库将包含要测试的代码。
- 将 *Class1.cs* 重命名为 *PrimeService.cs*。
- 将 PrimeService.cs 中的代码替换为以下代码：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Not implemented.");
        }
    }
}
```

- 前面的代码：
  - 引发 NotImplementedException，其中包含一条消息，指示未实现。
  - 稍后在教程中更新。
- 在 unit-testing-using-dotnet-test 目录下运行以下命令，向解决方案添加类库项目：

```
dotnet sln add ./PrimeService/PrimeService.csproj
```

- 运行以下命令创建 PrimeService.Tests 项目：

```
dotnet new xunit -o PrimeService.Tests
```

- 上面的命令：
  - 在 PrimeService.Tests 目录中创建 PrimeService.Tests 项目 。测试项目将 xUnit 用作测试库。
  - 通过将以下 `<PackageReference />` 元素添加到项目文件来配置测试运行程序：
    - `Microsoft.NET.Test.Sdk`
    - `xunit`
    - `xunit.runner.visualstudio`
    - `coverlet.collector`
- 运行以下命令将测试项目添加到解决方案文件：

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

- 将 `PrimeService` 类库作为一个依赖项添加到 PrimeService.Tests 项目中：

```
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference
./PrimeService/PrimeService.csproj
```

用于创建解决方案的命令

本部分汇总了上一部分中的所有命令。如果已完成上一部分中的步骤，请跳过本部分。

以下命令用于在 Windows 计算机上创建测试解决方案。对于 macOS 和 Unix，请将 `ren` 命令更新为 OS 版本的 `ren` 以重命名文件：

```
dotnet new sln -o unit-testing-using-dotnet-test
cd unit-testing-using-dotnet-test
dotnet new classlib -o PrimeService
ren .\PrimeService\Class1.cs PrimeService.cs
dotnet sln add ./PrimeService/PrimeService.csproj
dotnet new xunit -o PrimeService.Tests
dotnet add ./PrimeService.Tests/PrimeService.Tests.csproj reference ./PrimeService/PrimeService.csproj
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

请按照上一部分中的“将 PrimeService.cs 中的代码替换为以下代码”的说明进行操作。

创建测试

测试驱动开发 (TDD) 中的一种常用方法是在实现目标代码之前编写测试。本教程使用 TDD 方法。`IsPrime` 方法可调用，但未实现。对 `IsPrime` 的测试调用失败。对于 TDD，会编写已知失败的测试。更新目标代码使测试通过。你可以重复使用此方法，编写失败的测试，然后更新目标代码使测试通过。

更新 PrimeService.Tests 项目：

- 删除 PrimeService.Tests/UnitTest1.cs。
- 创建 PrimeService.Tests/PrimeService_IsPrimeShould.cs 文件。
- 将 PrimeService_IsPrimeShould.cs 中的代码替换为以下代码：

```
using Xunit;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    public class PrimeService_IsPrimeShould
    {
        [Fact]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var primeService = new PrimeService();
            bool result = primeService.IsPrime(1);

            Assert.False(result, "1 should not be prime");
        }
    }
}
```

`[Fact]` 属性声明由测试运行程序运行的测试方法。从 PrimeService.Tests 文件夹运行 `dotnet test`。dotnet test 命令生成两个项目并运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用单元测试项目启动测试运行程序。

测试失败，因为尚未实现 `IsPrime`。使用 TDD 方法，只需编写足够的代码即可使此测试通过。使用以下代码更新 `IsPrime`：

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Not fully implemented.");
}
```

运行 `dotnet test`。测试通过。

添加更多测试

为 0 和 -1 添加素数测试。可以复制在上一步中创建的测试，并复制以下代码以测试 0 和 -1。但请不要这样做，因为有更好的方法。

```
var primeService = new PrimeService();
bool result = primeService.IsPrime(1);

Assert.False(result, "1 should not be prime");
```

仅当参数更改代码重复和测试膨胀中的结果时复制测试代码。以下 xUnit 属性允许编写类似测试套件：

- `[Theory]` 表示执行相同代码，但具有不同输入参数的测试套件。
- `[InlineData]` 属性指定这些输入的值。

可以不使用上述 xUnit 属性创建新测试，而是用来创建单个索引。替换以下代码：

```
[Fact]
public void IsPrime_InputIs1_ReturnFalse()
{
    var primeService = new PrimeService();
    bool result = primeService.IsPrime(1);

    Assert.False(result, "1 should not be prime");
}
```

替换为以下代码：

```
[Theory]
[InlineData(-1)]
[InlineData(0)]
[InlineData(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.False(result, $"{value} should not be prime");
}
```

在前面的代码中，`[Theory]` 和 `[InlineData]` 允许测试多个小于 2 的值。2 是最小的素数。

在类声明之后和 `[Theory]` 属性之前添加以下代码：

```
private readonly PrimeService _primeService;

public PrimeService_IsPrimeShould()
{
    _primeService = new PrimeService();
}
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，请使用以下代码更新 `IsPrime` 方法：

```
public bool IsPrime(int candidate)
{
    if (candidate < 2)
    {
        return false;
    }
    throw new NotImplementedException("Not fully implemented.");
}
```

遵循 TDD 方法，添加更多失败的测试，然后更新目标代码。请参阅已完成的测试版本和库的完整实现。

已完成的 `IsPrime` 方法不是用于测试素性的有效算法。

其他资源

- xUnit.net 官方网站
- ASP.NET Core 中的测试控制器逻辑
- `dotnet add reference`

使用 dotnet test 和 xUnit 在 .NET Core 中进行 F# 库单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。这样便于管理类库和单元测试项目。在解决方案库中，创建 MathService 目录。目录和文件结构目前如下所示：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

将 MathService 作为当前目录，然后运行 `dotnet new classlib -lang "F#"` 以创建源项目。创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行 `dotnet sln add .\MathService\MathService.fsproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用 `dotnet new xunit -lang "F#"` 创建一个新项目。这会创建将 xUnit 用作测试库的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="xunit" Version="2.2.0" />
  <PackageReference Include="xunit.runner.visualstudio" Version="2.2.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在以前的步骤中已添加 xUnit 和 xUnit 运行程序。现在，

将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

在 unit-testing-with-fsharp 目录中执行 `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj`。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 Tests.fs 并添加以下代码：

```
[<Fact>]
let ``My test`` () =
    Assert.True(true)

[<Fact>]
let ``Fail every time`` () = Assert.True(false)
```

`[<Fact>]` 属性表示由测试运行程序运行的测试方法。在 unit-testing-with-fsharp中，执行 `dotnet test` 以构建测试和类库，然后运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方序列。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```
[<Fact>]
let ``Sequence of Evens returns empty collection`` () =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

测试失败。尚未创建实现。在起作用的 `MathService` 类中编写最简单的代码，使此测试通过：

```
let squaresOfOdds xs =
    Seq.empty<int>
```

在 unit-testing-with-fsharp 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目，然后构建 `MathService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

完成要求

你已经通过了一个测试，现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 `1`。数值 1 较为简单，因为 1 的平方是 1。下一个测试如下所示：

```
[<Fact>]
let ``Sequences of Ones and Evens returns Ones`` () =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.Equal<Collections.Generic.IEnumerable<int>>(expected, actual)
```

执行 `dotnet test` 以运行测试，并显示新测试失败。现在更新 `squaresOfOdds` 方法，以处理此新测试。筛选出序列中的所有偶数值，以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的：

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

还要执行一个步骤：计算每个奇数的平方值。从编写新测试开始：

```
[<Fact>]
let ``SquaresOfOdds works`` () =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.Equal(expected, actual)
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方，以此方式来修复测试的缺陷：

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- dotnet new
- dotnet sln
- dotnet add reference
- dotnet test

使用 dotnet test 和 xUnit 进行 Visual Basic .NET Core 库单元测试

2021/11/16 •

本教程演示如何生成包含单元测试项目和库项目的解决方案。若要使用预构建解决方案学习本教程，请查看或下载示例代码。有关下载说明，请参阅示例和教程。

创建解决方案

在本部分中，将创建包含源和测试项目的解决方案。已完成的解决方案具有以下目录结构：

```
/unit-testing-using-dotnet-test
    unit-testing-using-dotnet-test.sln
    /PrimeService
        PrimeService.vb
        PrimeService.vbproj
    /PrimeService.Tests
        PrimeService_IsPrimeShould.vb
        PrimeServiceTests.vbproj
```

以下说明提供了创建测试解决方案的步骤。有关通过一个步骤创建测试解决方案的说明，请参阅用于创建测试解决方案的命令。

- 打开 shell 窗口。
- 运行下面的命令：

  ```
  dotnet new sln -o unit-testing-using-dotnet-test
  ```

  `dotnet new sln` 命令用于在 unit-testing-using-dotnet-test 目录中创建新的解决方案。
- 将目录更改为 unit-testing-using-dotnet-test 文件夹。
- 运行下面的命令：

  ```
  dotnet new classlib -o PrimeService --lang VB
  ```

  `dotnet new classlib` 命令用于在 PrimeService 文件夹中创建新的类库项目。新的类库将包含要测试的代码。
- 将 Class1.vb 重命名为 PrimeService.vb。
- 将 PrimeService.vb 中的代码替换为以下代码：

```
Imports System

Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Not implemented.")
        End Function
    End Class
End Namespace
```

- 前面的代码：
  - 引发 NotImplementedException，其中包含一条消息，指示未实现。
  - 稍后在教程中更新。
- 在 unit-testing-using-dotnet-test 目录下运行以下命令，向解决方案添加类库项目：

```
dotnet sln add ./PrimeService/PrimeService.vbproj
```

- 运行以下命令创建 PrimeService.Tests 项目：

```
dotnet new xunit -o PrimeService.Tests
```

- 上面的命令：
  - 在 PrimeService.Tests 目录中创建 PrimeService.Tests 项目 。测试项目将 xUnit 用作测试库。
  - 通过将以下 `<PackageReference />` 元素添加到项目文件来配置测试运行程序：
    - “Microsoft.NET.Test.Sdk”
    - “xunit”
    - “xunit.runner.visualstudio”
- 运行以下命令将测试项目添加到解决方案文件：

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.vbproj
```

- 将 `PrimeService` 类库作为一个依赖项添加到 PrimeService.Tests 项目中：

```
dotnet add ./PrimeService.Tests/PrimeService.Tests.vbproj reference
./PrimeService/PrimeService.vbproj
```

用于创建解决方案的命令

本部分汇总了上一部分中的所有命令。如果已完成上一部分中的步骤，请跳过本部分。

以下命令用于在 Windows 计算机上创建测试解决方案。对于 macOS 和 Unix，请将 `ren` 命令更新为 OS 版本的 `ren` 以重命名文件：

```
dotnet new sln -o unit-testing-using-dotnet-test
cd unit-testing-using-dotnet-test
dotnet new classlib -o PrimeService
ren .\PrimeService\Class1.vb PrimeService.vb
dotnet sln add ./PrimeService/PrimeService.vbproj
dotnet new xunit -o PrimeService.Tests
dotnet add ./PrimeService.Tests/PrimeService.Tests.vbproj reference ./PrimeService/PrimeService.vbproj
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.vbproj
```

请按照上一部分中的“将 PrimeService.vb 中的代码替换为以下代码”的说明进行操作。

创建测试

测试驱动开发 (TDD) 中的一种常用方法是在实现目标代码之前编写测试。本教程使用 TDD 方法。`IsPrime` 方法可调用，但未实现。对 `IsPrime` 的测试调用失败。对于 TDD，会编写已知失败的测试。更新目标代码使测试通过。你可以重复使用此方法，编写失败的测试，然后更新目标代码使测试通过。

更新 PrimeService.Tests 项目：

- 删除 PrimeService.Tests/UnitTest1.vb。
- 创建 PrimeService.Tests/PrimeService_IsPrimeShould.vb 文件。
- 将 PrimeService_IsPrimeShould.vb 中的代码替换为以下代码：

```
Imports Xunit

Namespace PrimeService.Tests
    Public Class PrimeService_IsPrimeShould
        Private ReadOnly _primeService As Prime.Services.PrimeService

        Public Sub New()
            _primeService = New Prime.Services.PrimeService()
        End Sub


        <Fact>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

`[Fact]` 属性声明由测试运行程序运行的测试方法。从 PrimeService.Tests 文件夹运行 `dotnet test`。dotnet test 命令生成两个项目并运行测试。xUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用单元测试项目启动测试运行程序。

测试失败，因为尚未实现 `IsPrime`。使用 TDD 方法，只需编写足够的代码即可使此测试通过。使用以下代码更新 `IsPrime`：

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Not implemented.")
End Function
```

运行 `dotnet test`。测试通过。

添加更多测试

为 0 和 -1 添加素数测试。你可以复制上述测试并将以下代码更改为使用 0 和 -1：

```
Dim result As Boolean = _primeService.IsPrime(1)

Assert.False(result, "1 should not be prime")
```

仅当参数更改代码重复和测试膨胀中的结果时复制测试代码。以下 xUnit 属性允许编写类似测试套件：

- `[Theory]` 表示执行相同代码，但具有不同输入参数的测试套件。
- `[InlineData]` 属性指定这些输入的值。

可以不使用上述 xUnit 属性创建新测试，而是用来创建单个索引。替换以下代码：

```
<Fact>
Sub IsPrime_InputIs1_ReturnFalse()
    Dim result As Boolean = _primeService.IsPrime(1)

    Assert.False(result, "1 should not be prime")
End Sub
```

替换为以下代码：

```
<Theory>
<InlineData(-1)>
<InlineData(0)>
<InlineData(1)>
Sub IsPrime_ValuesLessThan2_ReturnFalse(ByVal value As Integer)
    Dim result As Boolean = _primeService.IsPrime(value)

    Assert.False(result, $"{value} should not be prime")
End Sub
```

在前面的代码中，`[Theory]` 和 `[InlineData]` 允许测试多个小于 2 的值。2 是最小的素数。

运行 `dotnet test`，其中两个测试失败。若要使所有测试通过，请使用以下代码更新 `IsPrime` 方法：

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate < 2 Then
        Return False
    End If
    Throw New NotImplementedException("Not fully implemented.")
End Function
```

遵循 TDD 方法，添加更多失败的测试，然后更新目标代码。请参阅已完成的测试版本和库的完整实现。

已完成的 `IsPrime` 方法不是用于测试素性的有效算法。

其他资源

- xUnit.net 官方网站
- ASP.NET Core 中的测试控制器逻辑
- `dotnet add reference`

使用 .NET CLI 组织和测试项目

2021/11/16 •

本教程遵循教程：使用 Visual Studio Code 通过 .NET 创建控制台应用程序，不仅介绍如何创建简单的控制台应用，还将介绍如何开发高级且结构完善的应用程序。在演示如何使用文件夹来组织代码后，本教程还将说明如何使用 xUnit 测试框架扩展控制台应用程序。

使用文件夹组织代码

如要要在控制台应用中引入新类型，可向该应用添加包含该类型的文件。例如，如果向项目添加包含 `AccountInformation` 和 `MonthlyReportRecords` 类型的文件，则项目文件结构是平面的，且易于导航：

```
/MyProject
|__AccountInformation.cs
|__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

但是，仅在项目规模相对较小时，此方法才适用。你能否想象在项目中添加 20 个类型时会发生什么？ 项目的根目录中会散落许多文件，这样的项目必然会难以导航和维护。

要组织项目，请创建一个名为 Models 新文件夹，将其用于保存类型文件。将类型文件放入 Models 文件夹中：

```
/MyProject
|__/Models
   |__AccountInformation.cs
   |__MonthlyReportRecords.cs
|__MyProject.csproj
|__Program.cs
```

按逻辑将文件分组到文件夹的项目易于导航和维护。在下一节中，将创建一个更复杂的示例，它包含文件夹和单元测试。

使用 NewTypes Pets 示例进行组织和测试

必备知识

- .NET 5.0 SDK 或更高版本。

生成示例

对于下列步骤，可使用 NewTypes Pets 示例进行相关操作，也可以创建自己的文件与文件夹进行操作。各类型按逻辑组织为文件夹结构，允许日后加入更多类型，测试也按逻辑放置在文件夹中，允许日后加入更多测试。

此示例包含两种类型 `Dog` 和 `Cat`，并使它们实现一个公共接口 `IPet`。对于 `NewTypes` 项目，目标是将与宠物相关的类型组织到 Pets 文件夹中。如果之后添加了另一组类型（例如 WildAnimals），则将其与 Pets 文件夹一同放在 NewTypes 文件夹中。WildAnimals 文件夹可包含不属于宠物的动物类型，如 `Squirrel` 和 `Rabbit` 类型。按照这种方式添加类型，不会破坏项目的良好组织。

创建以下文件夹结构，并指明文件内容：

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
```

IPet.cs:

```
using System;

namespace Pets
{
    public interface IPet
    {
        string TalkToOwner();
    }
}
```

Dog.cs:

```
using System;

namespace Pets
{
    public class Dog : IPet
    {
        public string TalkToOwner() => "Woof!";
    }
}
```

Cat.cs:

```
using System;

namespace Pets
{
    public class Cat : IPet
    {
        public string TalkToOwner() => "Meow!";
    }
}
```

Program.cs:

```
using System;
using Pets;
using System.Collections.Generic;

namespace ConsoleApplication
{
    public class Program
    {
        public static void Main(string[] args)
        {
            List<IPet> pets = new List<IPet>
            {
                new Dog(),
                new Cat()
            };

            foreach (var pet in pets)
            {
                Console.WriteLine(pet.TalkToOwner());
            }
        }
    }
}
```

NewTypes.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

</Project>
```

请执行以下命令：

```
dotnet run
```

获得以下输出：

```
Woof!
Meow!
```

可选练习：可通过扩展此项目来添加新的宠物类型，例如 `Bird`。使鸟的 `TalkToOwner` 方法向所有者发出 `Tweet!`。再次运行应用。输出将包含 `Tweet!`

测试示例

`NewTypes` 项目已准备就绪，与宠物相关的类型均置于一个文件夹中，因此具有良好的组织。接下来，创建测试项目，并使用 xUnit 测试框架开始编写测试。使用单元测试，可自动检查宠物类型的行为，确认其正常运行。

导航回 src 文件夹并创建“test”文件夹，后者包含 NewTypesTests 文件夹。在 NewTypesTests 文件夹的命令提示符中，执行 `dotnet new xunit`。这将生成两个文件：NewTypesTests.csproj 和 UnitTest1.cs。

测试项目当前无法测试 `NewTypes` 中的类型，并且需要对 `NewTypes` 项目的项目引用。要添加项目引用，请使用 `dotnet add reference` 命令：

```
dotnet add reference ../../src/NewTypes/NewTypes.csproj
```

或者，可以选择向 NewTypesTests.csproj 文件添加 `<ItemGroup>` 节点，手动添加项目引用：

```
<ItemGroup>
  <ProjectReference Include="../../src/NewTypes/NewTypes.csproj" />
</ItemGroup>
```

NewTypesTests.csproj:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3" />
  </ItemGroup>

  <ItemGroup>
    <ProjectReference Include="../../src/NewTypes/NewTypes.csproj"/>
  </ItemGroup>

</Project>
```

NewTypesTests.csproj 文件包含下列内容：

- 对 .NET 测试基础结构 `Microsoft.NET.Test.Sdk` 的包引用
- 对 xUnit 测试框架 `xunit` 的包引用
- 对测试运行程序 `xunit.runner.visualstudio` 的包引用
- 对要测试的代码 `NewTypes` 的项目引用

将 UnitTest1.cs 的名称更改为 PetTests.cs，并将文件中的代码替换为下列内容：

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.NotEqual(expected, actual);
    }
}
```

可选练习：如果先前向所有者添加了生成 `Tweet!` 的 `Bird` 类型，请向 PetTests.cs 文件 `BirdTalkToOwnerReturnsTweet` 添加测试方法，以检查对于 `Bird` 类型，`TalkToOwner` 方法是否正常工作。

> **NOTE**
> 尽管期望 `expected` 和 `actual` 值相等，但使用 `Assert.NotEqual` 检查的初始断言表明这些值并不相等。务必最初创建一个失败的测试，以检查测试的逻辑是否正确。确认测试失败后，调整断言，使测试通过。

下面演示了完整的项目结构：

```
/NewTypes
|__/src
   |__/NewTypes
      |__/Pets
         |__Dog.cs
         |__Cat.cs
         |__IPet.cs
      |__Program.cs
      |__NewTypes.csproj
|__/test
   |__NewTypesTests
      |__PetTests.cs
      |__NewTypesTests.csproj
```

在 test/NewTypesTests 目录中开始。使用 `dotnet test` 命令运行测试。此命令启动项目文件中指定的测试运行程序。

测试按预期失败，控制台显示以下输出：

```
Test run for C:\Source\dotnet\docs\samples\snippets\core\tutorials\testing-with-
cli\csharp\test\NewTypesTests\bin\Debug\net5.0\NewTypesTests.dll (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.8.1
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
[xUnit.net 00:00:00.50]     PetTests.DogTalkToOwnerReturnsWoof [FAIL]
  Failed PetTests.DogTalkToOwnerReturnsWoof [6 ms]
  Error Message:
   Assert.NotEqual() Failure
Expected: Not "Woof!"
Actual:   "Woof!"
  Stack Trace:
     at PetTests.DogTalkToOwnerReturnsWoof() in
C:\Source\dotnet\docs\samples\snippets\core\tutorials\testing-with-
cli\csharp\test\NewTypesTests\PetTests.cs:line 13

Failed!  - Failed:     1, Passed:     1, Skipped:     0, Total:     2, Duration: 8 ms - NewTypesTests.dll
(net5.0)
```

将测试的断言从 `Assert.NotEqual` 更改为 `Assert.Equal`：

```
using System;
using Xunit;
using Pets;

public class PetTests
{
    [Fact]
    public void DogTalkToOwnerReturnsWoof()
    {
        string expected = "Woof!";
        string actual = new Dog().TalkToOwner();

        Assert.Equal(expected, actual);
    }

    [Fact]
    public void CatTalkToOwnerReturnsMeow()
    {
        string expected = "Meow!";
        string actual = new Cat().TalkToOwner();

        Assert.Equal(expected, actual);
    }
}
```

使用 `dotnet test` 命令重新运行测试，并获得以下输出：

```
Test run for C:\Source\dotnet\docs\samples\snippets\core\tutorials\testing-with-
cli\csharp\test\NewTypesTests\bin\Debug\net5.0\NewTypesTests.dll (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.8.1
Copyright (c) Microsoft Corporation.  All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed!  - Failed:     0, Passed:     2, Skipped:     0, Total:     2, Duration: 2 ms - NewTypesTests.dll
(net5.0)
```

测试通过。在与所有者谈话时，宠物类型的方法返回正确的值。

你已了解使用 xUnit 来组织和测试项目的方法。继续使用这些方法，将它们应用于自己的项目中。祝你编码愉快！

使用 NUnit 和 .NET Core 进行 C# 单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

先决条件

- .NET Core 2.1 SDK 或更高版本。
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-using-nunit 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 PrimeService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
```

将 PrimeService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib
```

将 *Class1.cs* 重命名为 *PrimeService.cs*。创建 `PrimeService` 类的失败实现：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

将目录更改回 unit-testing-using-nunit 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add PrimeService/PrimeService.csproj
```

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit
```

dotnet new 命令可创建一个将 NUnit 用作测试库的测试项目。生成的模板在 PrimeService.Tests.csproj 文件中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.13.2" />
  <PackageReference Include="NUnit3TestAdapter" Version="4.1.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，`dotnet new` 已添加 Microsoft 测试 SDK、NUnit 测试框架和 NUnit 测试适配器。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

下图显示了最终的解决方案布局：

```
/unit-testing-using-nunit
    unit-testing-using-nunit.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.csproj
```

在 unit-testing-using-nunit 目录中执行以下命令：

```
dotnet sln add ./PrimeService.Tests/PrimeService.Tests.csproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。在 PrimeService.Tests 目录中，将 UnitTest1.cs 文件重命名为 PrimeService_IsPrimeShould.cs，并将其整个内容替换为以下代码：

```
using NUnit.Framework;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestFixture]
    public class PrimeService_IsPrimeShould
    {
        private PrimeService _primeService;

        [SetUp]
        public void SetUp()
        {
            _primeService = new PrimeService();
        }

        [Test]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            var result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}
```

`[TestFixture]` 属性表示包含单元测试的类。`[Test]` 属性指示方法是测试方法。

保存此文件并执行 `dotnet test` 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

在 unit-testing-using-nunit 目录中再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0，-1。可以添加具有 `[Test]` 属性的新测试，但这很快就会变得枯燥乏味。还有其他 NUnit 属性可用于编写一套类似的测试。`[TestCase]` 属性用于创建一套可执行相同代码但具有不同输入参数的测试。可以使用 `[TestCase]` 属性来指定这些输入的值。

无需创建新的测试，而是应用此属性来创建数据驱动的单个测试。数据驱动的测试方法用于测试多个小于 2（即最小质数）的值：

```
[TestCase(-1)]
[TestCase(0)]
[TestCase(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以在 PrimeService.cs 文件中更改 `Main` 方法开头的 `if` 子句：

```
if (candidate < 2)
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

使用 dotnet test 和 NUnit 在 .NET Core 中进行 F# 库的单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

先决条件

- .NET Core 2.1 SDK 或更高版本。
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 MathService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

将 MathService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib -lang "F#"
```

创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add .\MathService\MathService.fsproj
```

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit -lang "F#"
```

这会创建一个将 NUnit 用作测试框架的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="NUnit" Version="3.9.0" />
  <PackageReference Include="NUnit3TestAdapter" Version="3.9.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，`dotnet new` 已添加 NUnit 和 NUnit 测试适配器。现在，将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathService.Tests.fsproj
```

在 unit-testing-with-fsharp 目录中执行以下命令：

```
dotnet sln add .\MathService.Tests\MathService.Tests.fsproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 UnitTest1.fs 并添加以下代码：

```
namespace MathService.Tests

open System
open NUnit.Framework
open MathService

[<TestFixture>]
type TestClass () =

    [<Test>]
    member this.TestMethodPassing() =
        Assert.True(true)

    [<Test>]
     member this.FailEveryTime() = Assert.True(false)
```

`[<TestFixture>]` 属性表示包含测试的类。`[<Test>]` 属性表示由测试运行程序运行的测试方法。在 unit-testing-with-fsharp 目录中，执行 `dotnet test` 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方序列。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```
[<Test>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int>
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

请注意已将 `expected` 序列转换为列表。NUnit 框架依赖于许多标准 .NET 类型。此依赖关系表示公共接口和预期结果支持 ICollection，而非 IEnumerable。

运行此测试时，会看到测试失败。尚未创建实现。在起作用的 MathService 项目的 Library.fs 类中编写最简单的代码，使此测试通过：

```
let squaresOfOdds xs =
    Seq.empty<int>
```

在 unit-testing-with-fsharp 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目，然后构建 `MathService.Tests` 项目。构建这两个项目后，该命令将运行测试。现在两个测试通过。

完成要求

你已经通过了一个测试，现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 `1`。数值 1 较为简单，因为 1 的平方是 1。下一个测试如下所示：

```
[<Test>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

如果执行 `dotnet test`，新测试将失败。必须更新 `squaresOfOdds` 方法才能处理此新测试。必须筛选出序列中的所有偶数值，以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的：

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
```

注意对 `Seq.toList` 的调用。它会创建一个列表，此列表将实现 ICollection 接口。

还要执行一个步骤：计算每个奇数的平方值。从编写新测试开始：

```
[<Test>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.That(actual, Is.EqualTo(expected))
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方，以此方式来修复测试的缺陷：

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- dotnet add reference
- dotnet test

使用 dotnet test 和 NUnit 进行 Visual Basic .NET Core 库的单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

先决条件

- .NET Core 2.1 SDK 或更高版本。
- 按需选择的文本编辑器或代码编辑器。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-vb-nunit 的目录，以保留该解决方案。在此新目录中，运行以下命令，为类库和测试项目创建新的解决方案文件：

```
dotnet new sln
```

接下来，创建 PrimeService 目录。下图显示了当前的文件结构：

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
```

将 PrimeService 作为当前目录，并运行以下命令以创建源项目：

```
dotnet new classlib -lang VB
```

将 Class1.VB 重命名为 PrimeService.VB。创建 `PrimeService` 类的失败实现：

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first.")
        End Function
    End Class
End Namespace
```

将目录更改回 unit-testing-vb-using-mstest 目录。运行以下命令，向解决方案添加类库项目：

```
dotnet sln add .\PrimeService\PrimeService.vbproj
```

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

将 PrimeService.Tests 目录作为当前目录，并使用以下命令创建一个新项目：

```
dotnet new nunit -lang VB
```

dotnet new 命令可创建一个将 NUnit 用作测试库的测试项目。生成的模板在 PrimeServiceTests.vbproj 文件中配置了测试运行程序：

```
<ItemGroup>
  <PackageReference Include="nunit" Version="3.13.2" />
  <PackageReference Include="NUnit3TestAdapter" Version="4.1.0" />
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。在上一步中，`dotnet new` 已添加 NUnit 和 NUnit 测试适配器。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-vb-nunit
    unit-testing-vb-nunit.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeService.Tests.vbproj
```

在 unit-testing-vb-nunit 目录中执行以下命令：

```
dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj
```

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。在 PrimeService.Tests 目录中，将 UnitTest1.vb 文件重命名为 PrimeService_IsPrimeShould.VB，并将其整个内容替换为以下代码：

```
Imports NUnit.Framework

Namespace PrimeService.Tests
    <TestFixture>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <Test>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.False(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

`<TestFixture>` 属性指示包含测试的类。`<Test>` 属性表示由测试运行程序运行的方法。在 unit-testing-vb-nunit 中，执行 `dotnet test` 以构建测试和类库，然后运行测试。NUnit 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

在 unit-testing-vb-nunit 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0，-1。可以将这些情况添加为具有 `<Test>` 属性的新测试，但这很快就会变得枯燥乏味。还有其他 xUnit 属性，可使你编写类似测试套件。`<TestCase>` 属性表示执行相同代码，但具有不同输入参数一系列测试。可以使用 `<TestCase>` 属性来指定这些输入的值。

无需创建新测试，而是应用这两个属性来创建一系列测试，用于测试小于 2（最小质数）的几个值：

```
<TestFixture>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <TestCase(-1)>
    <TestCase(0)>
    <TestCase(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <TestCase(2)>
    <TestCase(3)>
    <TestCase(5)>
    <TestCase(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <TestCase(4)>
    <TestCase(6)>
    <TestCase(8)>
    <TestCase(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以在 PrimeServices.cs 文件中更改 `Main` 方法开头的 `if` 子句：

```
if candidate < 2
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

使用 MSTest 和 .NET 进行 C# 单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

先决条件

- .NET 5.0 SDK 或更高版本

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-using-mstest 的目录，用以保存解决方案。在此新目录中，运行 `dotnet new sln` 为类库和测试项目创建新的解决方案文件。创建 PrimeService 目录。下图显示了当前的目录和文件结构：

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
```

将 *PrimeService* 作为当前目录，然后运行 `dotnet new classlib` 以创建源项目。将 *Class1.cs* 重命名为 *PrimeService.cs*。将文件中的代码替换为以下代码，以创建 `PrimeService` 类的失败实现：

```
using System;

namespace Prime.Services
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            throw new NotImplementedException("Please create a test first.");
        }
    }
}
```

将目录更改回 unit-testing-using-mstest 目录。运行 `dotnet sln add` 以向解决方案添加类库项目：

```
dotnet sln add PrimeService/PrimeService.csproj
```

创建测试项目

创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
```

将 *PrimeService.Tests* 目录作为当前目录，并使用 `dotnet new mstest` 创建一个新项目。dotnet 新命令会创建一个将 MSTest 用作测试库的测试项目。模板在 PrimeServiceTests.csproj 文件中配置测试运行器：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="16.7.1" />
  <PackageReference Include="MSTest.TestAdapter" Version="2.1.1" />
  <PackageReference Include="MSTest.TestFramework" Version="2.1.1" />
  <PackageReference Include="coverlet.collector" Version="1.3.0" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。上一步中的 `dotnet new` 添加了 MSTest SDK、MSTest 测试框架、MSTest 运行器和 Coverlet 进行代码覆盖率报告。

将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.csproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

下图显示了最终的解决方案布局：

```
/unit-testing-using-mstest
    unit-testing-using-mstest.sln
    /PrimeService
        Source Files
        PrimeService.csproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.csproj
```

切换到 unit-testing-using-mstest 目录，并运行 `dotnet sln add`：

```
dotnet sln add .\PrimeService.Tests\PrimeService.Tests.csproj
```

创建第一个测试

编写失败测试，使其通过，然后重复此过程。从 *PrimeService.Tests* 目录删除 *UnitTest1.cs*，并创建一个名为 *PrimeService_IsPrimeShould.cs* 且包含以下内容的新 C# 文件：

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Prime.Services;

namespace Prime.UnitTests.Services
{
    [TestClass]
    public class PrimeService_IsPrimeShould
    {
        private readonly PrimeService _primeService;

        public PrimeService_IsPrimeShould()
        {
            _primeService = new PrimeService();
        }

        [TestMethod]
        public void IsPrime_InputIs1_ReturnFalse()
        {
            bool result = _primeService.IsPrime(1);

            Assert.IsFalse(result, "1 should not be prime");
        }
    }
}
```

TestClass 属性表示包含单元测试的类。TestMethod 属性指示方法是测试方法。

保存此文件并执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```
public bool IsPrime(int candidate)
{
    if (candidate == 1)
    {
        return false;
    }
    throw new NotImplementedException("Please create a test first.");
}
```

在 unit-testing-using-mstest 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0，-1。可使用 TestMethod 属性添加新测试，但这很快就会变得枯燥乏味。还有其他 MSTest 属性，使用这些属性可编写类似测试的套件。测试方法可以执行相同的代码，但具有不同的输入参数。可以使用 DataRow 属性来指定这些输入的值。

可以不使用这两个属性创建新测试，而用来创建单个数据驱动的测试。数据驱动的测试方法用于测试多个小于 2（即最小质数）的值。在 PrimeService_IsPrimeShould.cs 中添加新的测试方法：

```
[TestMethod]
[DataRow(-1)]
[DataRow(0)]
[DataRow(1)]
public void IsPrime_ValuesLessThan2_ReturnFalse(int value)
{
    var result = _primeService.IsPrime(value);

    Assert.IsFalse(result, $"{value} should not be prime");
}
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以在 PrimeService.cs 文件中更改 `IsPrime` 方法开头的 `if` 子句：

```
if (candidate < 2)
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- Microsoft.VisualStudio.TestTools.UnitTesting
- 在单元测试中使用 MSTest 框架
- MSTest V2 测试框架文档

使用 dotnet test 和 MSTest 在 .NET Core 中进行 F# 库单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-with-fsharp 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。这样便于管理类库和单元测试项目。在解决方案库中，创建 MathService 目录。目录和文件结构目前如下所示：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
```

将 MathService 作为当前目录，然后运行 `dotnet new classlib -lang "F#"` 以创建源项目。创建数学服务的失败实现：

```
module MyMath =
    let squaresOfOdds xs = raise (System.NotImplementedException("You haven't written a test yet!"))
```

将目录更改回 unit-testing-with-fsharp 目录。运行 `dotnet sln add .\MathService\MathService.fsproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 MathService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
```

将 MathService.Tests 目录作为当前目录，并使用 `dotnet new mstest -lang "F#"` 创建一个新项目。这会创建一个将 MSTest 用作测试框架的测试项目。生成的模板在 MathServiceTests.fsproj 中配置测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.3.0-preview-20170628-02" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在前面的步骤中已添加 MSTest 和 MSTest 运行程序。现

在，将 `MathService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../MathService/MathService.fsproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-with-fsharp
    unit-testing-with-fsharp.sln
    /MathService
        Source Files
        MathService.fsproj
    /MathService.Tests
        Test Source Files
        MathServiceTests.fsproj
```

在 unit-testing-with-fsharp 目录中执行 `dotnet sln add .\MathService.Tests\MathService.Tests.fsproj`。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。打开 Tests.fs 并添加以下代码：

```
namespace MathService.Tests

open System
open Microsoft.VisualStudio.TestTools.UnitTesting
open MathService

[<TestClass>]
type TestClass () =

    [<TestMethod>]
    member this.TestMethodPassing() =
        Assert.IsTrue(true)

    [<TestMethod>]
     member this.FailEveryTime() = Assert.IsTrue(false)
```

`[<TestClass>]` 属性表示包含测试的类。`[<TestMethod>]` 属性表示由测试运行程序运行的测试方法。在 unit-testing-with-fsharp 目录中，执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

这两个测试演示了最基本的已通过测试和未通过测试。`My test` 通过，而 `Fail every time` 未通过。现在创建针对 `squaresOfOdds` 方法的测试。`squaresOfOdds` 方法返回输入序列中所有奇整数值的平方列表。可以以迭代的方式创建可验证此功能的测试，而非尝试同时写入所有的函数。若要让每个测试都通过，意味着要针对此方法创建必要的功能。

可以编写的最简单的测试是调用包含所有偶数的 `squaresOfOdds`，它的结果应该是一个空整数序列。此测试如下所示：

```
[<TestMethod>]
member this.TestEvenSequence() =
    let expected = Seq.empty<int> |> Seq.toList
    let actual = MyMath.squaresOfOdds [2; 4; 6; 8; 10]
    Assert.AreEqual(expected, actual)
```

请注意已将 `expected` 序列转换为列表。MSTest 库依赖于许多标准 .NET 类型。此依赖关系表示公共接口和预期结果支持 ICollection，而非 IEnumerable。

运行此测试时，会看到测试失败。尚未创建实现。在起作用的 `Mathservice` 类中编写最简单的代码，使此测试通过：

```
let squaresOfOdds xs =
    Seq.empty<int> |> Seq.toList
```

在 unit-testing-with-fsharp 目录中，再次运行 `dotnet test`。`dotnet test` 命令构建 `MathService` 项目，然后构建 `MathService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

完成要求

你已经通过了一个测试，现在可以编写更多测试。下一个简单示例使用的序列包含的唯一奇数为 `1`。数值 1 较为简单，因为 1 的平方是 1。下一个测试如下所示：

```
[<TestMethod>]
member public this.TestOnesAndEvens() =
    let expected = [1; 1; 1; 1]
    let actual = MyMath.squaresOfOdds [2; 1; 4; 1; 6; 1; 8; 1; 10]
    Assert.AreEqual(expected, actual)
```

如果执行 `dotnet test`，新测试将失败。必须更新 `squaresOfOdds` 方法才能处理此新测试。必须筛选出序列中的所有偶数值，以使此测试通过。可以编写一个小筛选器函数并使用 `Seq.filter` 来实现此目的：

```
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd |> Seq.toList
```

注意对 `Seq.toList` 的调用。它会创建一个列表，此列表将实现 ICollection 接口。

还要执行一个步骤：计算每个奇数的平方值。从编写新测试开始：

```
[<TestMethod>]
member public this.TestSquaresOfOdds() =
    let expected = [1; 9; 25; 49; 81]
    let actual = MyMath.squaresOfOdds [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]
    Assert.AreEqual(expected, actual)
```

可以通过映射操作传递经过筛选的序列来计算每个奇数的平方，以此方式来修复测试的缺陷：

```
let private square x = x * x
let private isOdd x = x % 2 <> 0

let squaresOfOdds xs =
    xs
    |> Seq.filter isOdd
    |> Seq.map square
    |> Seq.toList
```

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

另请参阅

- dotnet new
- dotnet sln
- dotnet add reference
- dotnet test

使用 dotnet test 和 MSTest 进行 Visual Basic .NET Core 库单元测试

2021/11/16 •

本教程介绍分步构建示例解决方案的交互式体验，以了解单元测试概念。如果希望使用预构建解决方案学习本教程，请在开始前查看或下载示例代码。有关下载说明，请参阅示例和教程。

本文介绍如何测试 .NET Core 项目。如果要测试 ASP.NET Core 项目，请参阅 ASP.NET Core 中的集成测试。

创建源项目

打开 shell 窗口。创建一个名为 unit-testing-vb-mstest 的目录，以保留该解决方案。在此新目录中，运行 `dotnet new sln` 创建新的解决方案。此做法便于管理类库和单元测试项目。在解决方案目录中，创建 PrimeService 目录。目前目录和文件结构如下所示：

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
```

将 *PrimeService* 作为当前目录，然后运行 `dotnet new classlib -lang VB` 以创建源项目。将 Class1.VB 重命名为 PrimeService.VB。创建 `PrimeService` 类的失败实现：

```
Namespace Prime.Services
    Public Class PrimeService
        Public Function IsPrime(candidate As Integer) As Boolean
            Throw New NotImplementedException("Please create a test first")
        End Function
    End Class
End Namespace
```

将目录更改回 unit-testing-vb-using-mstest 目录。运行 `dotnet sln add .\PrimeService\PrimeService.vbproj` 向解决方案添加类库项目。

创建测试项目

接下来，创建 PrimeService.Tests 目录。下图显示了它的目录结构：

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
```

将 *PrimeService.Tests* 目录作为当前目录，并使用 `dotnet new mstest -lang VB` 创建一个新项目。此命令会创建一个将 MSTest 用作测试库的测试项目。生成的模板在 PrimeServiceTests.vbproj 中配置了测试运行程序：

```
<ItemGroup>
  <PackageReference Include="Microsoft.NET.Test.Sdk" Version="15.5.0" />
  <PackageReference Include="MSTest.TestAdapter" Version="1.1.18" />
  <PackageReference Include="MSTest.TestFramework" Version="1.1.18" />
</ItemGroup>
```

测试项目需要其他包创建和运行单元测试。`dotnet new` 在前面的步骤中已添加 MSTest 和 MSTest 运行程序。现在，将 `PrimeService` 类库作为另一个依赖项添加到项目中。使用 `dotnet add reference` 命令：

```
dotnet add reference ../PrimeService/PrimeService.vbproj
```

可以在 GitHub 上的示例存储库中看到整个文件。

最终的解决方案布局将如下所示：

```
/unit-testing-vb-mstest
    unit-testing-vb-mstest.sln
    /PrimeService
        Source Files
        PrimeService.vbproj
    /PrimeService.Tests
        Test Source Files
        PrimeServiceTests.vbproj
```

在 unit-testing-vb-mstest 目录中执行 `dotnet sln add .\PrimeService.Tests\PrimeService.Tests.vbproj`。

创建第一个测试

编写一个失败测试，使其通过，然后重复此过程。从 PrimeService.Tests 目录删除 UnitTest1.vb，并创建一个名为 PrimeService_IsPrimeShould.VB 的新 Visual Basic 文件。添加以下代码：

```
Imports Microsoft.VisualStudio.TestTools.UnitTesting

Namespace PrimeService.Tests
    <TestClass>
    Public Class PrimeService_IsPrimeShould
        Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

        <TestMethod>
        Sub IsPrime_InputIs1_ReturnFalse()
            Dim result As Boolean = _primeService.IsPrime(1)

            Assert.IsFalse(result, "1 should not be prime")
        End Sub

    End Class
End Namespace
```

`<TestClass>` 属性指示包含测试的类。`<TestMethod>` 属性表示由测试运行程序运行的方法。在 unit-testing-vb-mstest 中，执行 `dotnet test` 以构建测试和类库，然后运行测试。MSTest 测试运行程序包含要运行测试的程序入口点。`dotnet test` 使用已创建的单元测试项目启动测试运行程序。

测试失败。尚未创建实现。在起作用的 `PrimeService` 类中编写最简单的代码，使此测试通过：

```
Public Function IsPrime(candidate As Integer) As Boolean
    If candidate = 1 Then
        Return False
    End If
    Throw New NotImplementedException("Please create a test first.")
End Function
```

在 unit-testing-vb-mstest 目录中，再次运行 `dotnet test`。 `dotnet test` 命令构建 `PrimeService` 项目，然后构建 `PrimeService.Tests` 项目。构建这两个项目后，该命令将运行此单项测试。测试通过。

添加更多功能

你已经通过了一个测试，现在可以编写更多测试。质数有其他几种简单情况：0，-1。可以将这些情况添加为具有 `<TestMethod>` 属性的新测试，但这很快就会变得枯燥乏味。还有其他 MSTest 属性，使用这些属性可编写类似测试的套件。 `<DataTestMethod>` 属性表示执行相同代码，但具有不同输入参数一系列测试。可以使用 `<DataRow>` 属性来指定这些输入的值。

可以不使用这两个属性创建新测试，而用来创建单个索引。此索引是测试多个小于 2（即最小的质数）的值的方法：

```
<TestClass>
Public Class PrimeService_IsPrimeShould
    Private _primeService As Prime.Services.PrimeService = New Prime.Services.PrimeService()

    <DataTestMethod>
    <DataRow(-1)>
    <DataRow(0)>
    <DataRow(1)>
    Sub IsPrime_ValuesLessThan2_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub

    <DataTestMethod>
    <DataRow(2)>
    <DataRow(3)>
    <DataRow(5)>
    <DataRow(7)>
    Public Sub IsPrime_PrimesLessThan10_ReturnTrue(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsTrue(result, $"{value} should be prime")
    End Sub

    <DataTestMethod>
    <DataRow(4)>
    <DataRow(6)>
    <DataRow(8)>
    <DataRow(9)>
    Public Sub IsPrime_NonPrimesLessThan10_ReturnFalse(value As Integer)
        Dim result As Boolean = _primeService.IsPrime(value)

        Assert.IsFalse(result, $"{value} should not be prime")
    End Sub
End Class
```

运行 `dotnet test`，两项测试均失败。若要使所有测试通过，可以更改方法开头的 `if` 子句：

```
if candidate < 2
```

通过在主库中添加更多测试、理论和代码继续循环访问。你将拥有已完成的测试版本和库的完整实现。

你已生成一个小型库和该库的一组单元测试。你已将解决方案结构化，使添加新包和新测试成为了正常工作流的一部分。你已将多数的时间和精力集中在解决应用程序的目标上。

运行选定的单元测试

2021/11/16 •

借助 .NET Core 中的 `dotnet test` 命令，可以使用筛选表达式来运行选定的测试。本文演示如何筛选测试。本示例使用 `dotnet test`。如果使用的是 `vstest.console.exe`，请将 `--filter` 替换成 `--testcasefilter:`。

语法

```
dotnet test --filter <Expression>
```

- 表达式的格式为 `<Property><Operator><Value>[|&<Expression>]`。

  表达式可与布尔运算符结合使用：`|` 表示 or 布尔运算符，`&` 表示 and 布尔运算符 。

  表达式可以用括号括起来。例如：`(Name~MyClass) | (Name~MyClass2)`。

  没有任何运算符的表达式被解释为 `FullyQualifiedName` 属性上的 contains 。例如，`dotnet test --filter xyz` 和 `dotnet test --filter FullyQualifiedName~xyz` 相同。

- 属性是 `Test Case` 的一个特性。例如，常用单元测试框架支持以下属性。

[illegible]	[illegible]

MSTest	`FullyQualifiedName`
	`Name`
	`ClassName`
	`Priority`
	`TestCategory`

xUnit	`FullyQualifiedName`
	`DisplayName`
	`Traits`

Nunit	`FullyQualifiedName`
	`Name`
	`Priority`
	`TestCategory`

- **运算符**
  - `=` 完全匹配
  - `!=` 非完全匹配
  - `~` 包含
  - `!~` 不包含
- 值是一个字符串。所有查找都不区分大小写。

字符转义

若要在 Linux 或 macOS 上的筛选表达式中使用感叹号 (`!`)，请在其前面加一个反斜杠 (`\!`) 对其进行转义。例如，以下筛选器跳过包含 `IntegrationTests` 的命名空间中的所有测试：

```
dotnet test --filter FullyQualifiedName\!~IntegrationTests
```

对于包含泛型类型参数的逗号的 `FullyQualifiedName` 值，请使用 `%2C` 来转义逗号。例如：

```
dotnet test --filter
"FullyQualifiedName=MyNamespace.MyTestsClass<ParameterType1%2CParameterType2>.MyTestMethod"
```

MSTest 示例

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MSTestNamespace
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod, Priority(1), TestCategory("CategoryA")]
        public void TestMethod1()
        {
        }

        [TestMethod, Priority(2)]
        public void TestMethod2()
        {
        }
    }
}
```

■■■	■■
`dotnet test --filter Method`	运行 FullyQualifiedName 包含 `Method` 的测试。
`dotnet test --filter Name~TestMethod1`	运行名称包含 `TestMethod1` 的测试。
`dotnet test --filter ClassName=MSTestNamespace.UnitTest1`	运行属于类 `MSTestNamespace.UnitTest1` 的测试。 ■：由于 `ClassName` 值应有命名空间，因此 `ClassName=UnitTest1` 无效。
`dotnet test --filter FullyQualifiedName!=MSTestNamespace.UnitTest1.TestMethod1`	运行除 `MSTestNamespace.UnitTest1.TestMethod1` 之外的其他所有测试。
`dotnet test --filter TestCategory=CategoryA`	运行含 `[TestCategory("CategoryA")]` 批注的测试。
`dotnet test --filter Priority=2`	运行含 `[Priority(2)]` 批注的测试。

使用条件运算符 `|` 和 `&` 的示例：

- 运行 FullyQualifiedName 中包含 `UnitTest1` 或 TestCategoryAttribute 为 `"CategoryA"` 的测试。

  ```
  dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"
  ```

- 运行 FullyQualifiedName 中包含 `UnitTest1` 且 TestCategoryAttribute 为 `"CategoryA"` 的测试。

```
dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"
```

- 运行 FullyQualifiedName 中包含 `UnitTest1` 且 TestCategoryAttribute 为 `"CategoryA"` 或 PriorityAttribute 的优先级为 `1` 的测试。

```
dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"
```

xUnit 示例

```
using Xunit;

namespace XUnitNamespace
{
    public class TestClass1
    {
        [Fact, Trait("Priority", "1"), Trait("Category", "CategoryA")]
        public void Test1()
        {
        }

        [Fact, Trait("Priority", "2")]
        public void Test2()
        {
        }
    }
}
```

`dotnet test --filter DisplayName=XUnitNamespace.TestClass1.Test1`	仅运行一个测试，即 `XUnitNamespace.TestClass1.Test1`。
`dotnet test --filter FullyQualifiedName!=XUnitNamespace.TestClass1.Test1`	运行除 `XUnitNamespace.TestClass1.Test1` 之外的其他所有测试。
`dotnet test --filter DisplayName~TestClass1`	运行显示名称包含 `TestClass1` 的测试。

在代码示例中，包含键 `"Category"` 和 `"Priority"` 的已定义特征可用于筛选。

`dotnet test --filter XUnit`	运行 FullyQualifiedName 包含 `XUnit` 的测试。
`dotnet test --filter Category=CategoryA`	运行包含 `[Trait("Category", "CategoryA")]` 的测试。

使用条件运算符 `|` 和 `&` 的示例：

- 运行 FullyQualifiedName 中包含 `TestClass1` 或 `Trait` 的键为 `"Category"` 且值为 `"CategoryA"` 的测试。

```
dotnet test --filter "FullyQualifiedName~TestClass1|Category=CategoryA"
```

- 运行 FullyQualifiedName 中包含 `TestClass1` 且 `Trait` 的键为 `"Category"` 且值为 `"CategoryA"` 的测试。

```
dotnet test --filter "FullyQualifiedName~TestClass1&Category=CategoryA"
```

- 运行 FullyQualifiedName 中包含 `TestClass1` 且 `Trait` 的键为 `"Category"` 且值为 `"CategoryA"` 或 `Trait` 的键为 `"Priority"` 且值为 `1` 的测试。

```
dotnet test --filter "(FullyQualifiedName~TestClass1&Category=CategoryA)|Priority=1"
```

NUnit 示例

```
using NUnit.Framework;

namespace NUnitNamespace
{
    public class UnitTest1
    {
        [Test, Property("Priority", 1), Category("CategoryA")]
        public void TestMethod1()
        {
        }

        [Test, Property("Priority", 2)]
        public void TestMethod2()
        {
        }
    }
}
```

`dotnet test --filter Method`	运行 FullyQualifiedName 包含 `Method` 的测试。
`dotnet test --filter Name~TestMethod1`	运行名称包含 `TestMethod1` 的测试。
`dotnet test --filter FullyQualifiedName~NUnitNamespace.UnitTest1`	运行属于类 `NUnitNamespace.UnitTest1` 的测试。
`dotnet test --filter FullyQualifiedName!=NUnitNamespace.UnitTest1.TestMethod1`	运行除 `NUnitNamespace.UnitTest1.TestMethod1` 之外的其他所有测试。
`dotnet test --filter TestCategory=CategoryA`	运行含 `[Category("CategoryA")]` 批注的测试。
`dotnet test --filter Priority=2`	运行含 `[Priority(2)]` 批注的测试。

使用条件运算符 `|` 和 `&` 的示例：

运行 FullyQualifiedName 中包含 `UnitTest1` 或 `Category` 为 `"CategoryA"` 的测试。

```
dotnet test --filter "FullyQualifiedName~UnitTest1|TestCategory=CategoryA"
```

运行 FullyQualifiedName 中包含 `UnitTest1` 且 `Category` 为 `"CategoryA"` 的测试。

```
dotnet test --filter "FullyQualifiedName~UnitTest1&TestCategory=CategoryA"
```

运行 FullyQualifiedName 中包含 `UnitTest1` 且 `Category` 为 `"CategoryA"` 或 `Property` 的 `"Priority"` 为 `1` 的测试。

```
dotnet test --filter "(FullyQualifiedName~UnitTest1&TestCategory=CategoryA)|Priority=1"
```

有关详细信息，请参阅 TestCase 筛选器

请参阅

- dotnet test
- dotnet test -- 筛选器

后续步骤

对单元测试排序

对单元测试排序

2021/11/16 •

有时，你可能希望按特定顺序运行单元测试。理想情况下，单元测试的运行顺序不重要，最佳做法是避免对单元测试排序。但无论如何，可能会有需要这样做。为此，本文将演示如何对测试运行进行排序。

如果你更喜欢浏览源代码，请参阅对 .NET Core 单元测试排序示例存储库。

> **TIP**
> 除了本文中所述的排序功能，还应考虑将使用 Visual Studio 创建自定义播放列表作为替代方法。

按字母顺序排序

使用 MSTest，测试将按其测试名称自动排序。

> **NOTE**
> 名为 `Test14` 的测试将在 `Test2` 之前运行，即使数字 `2` 小于 `14` 也是如此。这是因为，测试名称排序使用的是测试的文本名称。

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace MSTest.Project
{
    [TestClass]
    public class ByAlphabeticalOrder
    {
        public static bool Test1Called;
        public static bool Test2Called;
        public static bool Test3Called;

        [TestMethod]
        public void Test2()
        {
            Test2Called = true;

            Assert.IsTrue(Test1Called);
            Assert.IsFalse(Test3Called);
        }

        [TestMethod]
        public void Test1()
        {
            Test1Called = true;

            Assert.IsFalse(Test2Called);
            Assert.IsFalse(Test3Called);
        }

        [TestMethod]
        public void Test3()
        {
            Test3Called = true;

            Assert.IsTrue(Test1Called);
            Assert.IsTrue(Test2Called);
        }
    }
}
```

xUnit 测试框架允许对测试运行顺序进行更细致的控制。可以实现 `ITestCaseOrderer` 和 `ITestCollectionOrderer` 接口，以控制类或测试集合的测试用例的顺序。

按测试用例的字母顺序排序

若要按其方法名称对测试用例排序，可以实现 `ITestCaseOrderer` 并提供排序机制。

```
using System.Collections.Generic;
using System.Linq;
using Xunit.Abstractions;
using Xunit.Sdk;

namespace XUnit.Project.Orderers
{
    public class AlphabeticalOrderer : ITestCaseOrderer
    {
        public IEnumerable<TTestCase> OrderTestCases<TTestCase>(
            IEnumerable<TTestCase> testCases) where TTestCase : ITestCase =>
            testCases.OrderBy(testCase => testCase.TestMethod.Method.Name);
    }
}
```

然后，在测试类中，使用 `TestCaseOrdererAttribute` 设置测试用例的顺序。

```
using Xunit;

namespace XUnit.Project
{
    [TestCaseOrderer("XUnit.Project.Orderers.AlphabeticalOrderer", "XUnit.Project")]
    public class ByAlphabeticalOrder
    {
        public static bool Test1Called;
        public static bool Test2Called;
        public static bool Test3Called;

        [Fact]
        public void Test1()
        {
            Test1Called = true;

            Assert.False(Test2Called);
            Assert.False(Test3Called);
        }

        [Fact]
        public void Test2()
        {
            Test2Called = true;

            Assert.True(Test1Called);
            Assert.False(Test3Called);
        }

        [Fact]
        public void Test3()
        {
            Test3Called = true;

            Assert.True(Test1Called);
            Assert.True(Test2Called);
        }
    }
}
```

按集合的字母顺序排序

若要按其显示名称对测试集合排序，可以实现 `ITestCollectionOrderer` 并提供排序机制。

```
using System.Collections.Generic;
using System.Linq;
using Xunit;
using Xunit.Abstractions;

namespace XUnit.Project.Orderers
{
    public class DisplayNameOrderer : ITestCollectionOrderer
    {
        public IEnumerable<ITestCollection> OrderTestCollections(
            IEnumerable<ITestCollection> testCollections) =>
            testCollections.OrderBy(collection => collection.DisplayName);
    }
}
```

由于测试集合可能会并行运行，因此必须使用 `CollectionBehaviorAttribute` 显式禁用集合的测试并行化。然后，将实现指定到 `TestCollectionOrdererAttribute`。

```
using Xunit;

// Need to turn off test parallelization so we can validate the run order
[assembly: CollectionBehavior(DisableTestParallelization = true)]
[assembly: TestCollectionOrderer("XUnit.Project.Orderers.DisplayNameOrderer", "XUnit.Project")]

namespace XUnit.Project
{
    [Collection("Xzy Test Collection")]
    public class TestsInCollection1
    {
        public static bool Collection1Run;

        [Fact]
        public static void Test()
        {
            Assert.True(TestsInCollection2.Collection2Run);     // Abc
            Assert.True(TestsInCollection3.Collection3Run);     // Mno
            Assert.False(TestsInCollection1.Collection1Run);    // Xyz

            Collection1Run = true;
        }
    }

    [Collection("Abc Test Collection")]
    public class TestsInCollection2
    {
        public static bool Collection2Run;

        [Fact]
        public static void Test()
        {
            Assert.False(TestsInCollection2.Collection2Run);    // Abc
            Assert.False(TestsInCollection3.Collection3Run);    // Mno
            Assert.False(TestsInCollection1.Collection1Run);    // Xyz

            Collection2Run = true;
        }
    }

    [Collection("Mno Test Collection")]
    public class TestsInCollection3
    {
        public static bool Collection3Run;

        [Fact]
        public static void Test()
        {
            Assert.True(TestsInCollection2.Collection2Run);     // Abc
            Assert.False(TestsInCollection3.Collection3Run);    // Mno
            Assert.False(TestsInCollection1.Collection1Run);    // Xyz

            Collection3Run = true;
        }
    }
}
```

按自定义属性排序

若要使用自定义属性对 xUnit 测试排序，首先需要一个要依赖的属性。按如下所示定义 `TestPriorityAttribute`：

```
using System;

namespace XUnit.Project.Attributes
{
    [AttributeUsage(AttributeTargets.Method, AllowMultiple = false)]
    public class TestPriorityAttribute : Attribute
    {
        public int Priority { get; private set; }

        public TestPriorityAttribute(int priority) => Priority = priority;
    }
}
```

接下来，请考虑 `ITestCaseOrderer` 接口的以下 `PriorityOrderer` 实现。

```
using System.Collections.Generic;
using System.Linq;
using Xunit.Abstractions;
using Xunit.Sdk;
using XUnit.Project.Attributes;

namespace XUnit.Project.Orderers
{
    public class PriorityOrderer : ITestCaseOrderer
    {
        public IEnumerable<TTestCase> OrderTestCases<TTestCase>(
            IEnumerable<TTestCase> testCases) where TTestCase : ITestCase
        {
            string assemblyName = typeof(TestPriorityAttribute).AssemblyQualifiedName!;
            var sortedMethods = new SortedDictionary<int, List<TTestCase>>();
            foreach (TTestCase testCase in testCases)
            {
                int priority = testCase.TestMethod.Method
                    .GetCustomAttributes(assemblyName)
                    .FirstOrDefault()
                    ?.GetNamedArgument<int>(nameof(TestPriorityAttribute.Priority)) ?? 0;

                GetOrCreate(sortedMethods, priority).Add(testCase);
            }

            foreach (TTestCase testCase in
                sortedMethods.Keys.SelectMany(
                    priority => sortedMethods[priority].OrderBy(
                        testCase => testCase.TestMethod.Method.Name)))
            {
                yield return testCase;
            }
        }

        private static TValue GetOrCreate<TKey, TValue>(
            IDictionary<TKey, TValue> dictionary, TKey key)
            where TKey : struct
            where TValue : new() =>
            dictionary.TryGetValue(key, out TValue result)
                ? result
                : (dictionary[key] = new TValue());
    }
}
```

然后，在测试类中，使用 `TestCaseOrdererAttribute` 将测试用例的顺序设置为 `PriorityOrderer`。

```
using Xunit;
using XUnit.Project.Attributes;

namespace XUnit.Project
{
    [TestCaseOrderer("XUnit.Project.Orderers.PriorityOrderer", "XUnit.Project")]
    public class ByPriorityOrder
    {
        public static bool Test1Called;
        public static bool Test2ACalled;
        public static bool Test2BCalled;
        public static bool Test3Called;

        [Fact, TestPriority(5)]
        public void Test3()
        {
            Test3Called = true;

            Assert.True(Test1Called);
            Assert.True(Test2ACalled);
            Assert.True(Test2BCalled);
        }

        [Fact, TestPriority(0)]
        public void Test2B()
        {
            Test2BCalled = true;

            Assert.True(Test1Called);
            Assert.True(Test2ACalled);
            Assert.False(Test3Called);
        }

        [Fact]
        public void Test2A()
        {
            Test2ACalled = true;

            Assert.True(Test1Called);
            Assert.False(Test2BCalled);
            Assert.False(Test3Called);
        }

        [Fact, TestPriority(-5)]
        public void Test1()
        {
            Test1Called = true;

            Assert.False(Test2ACalled);
            Assert.False(Test2BCalled);
            Assert.False(Test3Called);
        }
    }
}
```

按优先级排序

为了显式对测试排序，NUnit 提供了 `OrderAttribute`。具有此属性的测试先于没有此属性的测试启动。顺序值用于确定运行单元测试的顺序。

```
using NUnit.Framework;

namespace NUnit.Project
{
    public class ByOrder
    {
        public static bool Test1Called;
        public static bool Test2ACalled;
        public static bool Test2BCalled;
        public static bool Test3Called;

        [Test, Order(5)]
        public void Test1()
        {
            Test3Called = true;

            Assert.IsTrue(Test1Called);
            Assert.IsFalse(Test2ACalled);
            Assert.IsTrue(Test2BCalled);
        }

        [Test, Order(0)]
        public void Test2B()
        {
            Test2BCalled = true;

            Assert.IsTrue(Test1Called);
            Assert.IsFalse(Test2ACalled);
            Assert.IsFalse(Test3Called);
        }

        [Test]
        public void Test2A()
        {
            Test2ACalled = true;

            Assert.IsTrue(Test1Called);
            Assert.IsTrue(Test2BCalled);
            Assert.IsTrue(Test3Called);
        }

        [Test, Order(-5)]
        public void Test3()
        {
            Test1Called = true;

            Assert.IsFalse(Test2ACalled);
            Assert.IsFalse(Test2BCalled);
            Assert.IsFalse(Test3Called);
        }
    }
}
```

后续步骤

单元测试代码覆盖率

将代码覆盖率用于单元测试

2021/11/16 •

单元测试有助于确保功能的正常运行，并为重构工作提供一种验证方法。代码覆盖率是单元测试运行的代码量（行、分支或方法）的度量值。例如，如果你有一个简单的应用程序，其中只有两个条件分支（分支 a 和分支 b），则验证条件分支 a 的单元测试将报告 50% 的分支代码覆盖率。

本文介绍如何通过 Coverlet 在单元测试中使用代码覆盖率和使用 ReportGenerator 生成报表。尽管本文重点介绍 C# 和 xUnit 作为测试框架，但 MSTest 和 NUnit 也适用。Coverlet 是 GitHub 上的开源项目，可为 C# 提供跨平台代码覆盖率框架。Coverlet 是 .NET Foundation 的一部分。Coverlet 收集 Cobertura 覆盖率测试运行数据，用于生成报表。

此外，本文详细介绍如何使用从 Coverlet 测试运行收集的代码覆盖率信息来生成报表。可以使用另一个 GitHub 上的开源项目 - ReportGenerator 来生成报表。ReportGenerator 将由 Cobertura 生成的覆盖率报表转换为各种格式的用户可读的报表。

本文基于示例浏览器中提供的示例源代码项目。

测试中的系统

“测试中的系统”指的是要对其编写单元测试的代码，这可能是对象、服务或其他任何公开可测试功能的内容。为了本文的目的，你将创建一个类库（它将成为测试中的系统），以及两个对应的单元测试项目。

创建类库

在名为 `UnitTestingCodeCoverage` 的新目录中的命令提示符下，使用 `dotnet new classlib` 命令创建新的 .NET 标准类库：

```
dotnet new classlib -n Numbers
```

下面的代码段定义了一个简单的 `PrimeService` 类，该类提供了用于检查数值是否为质数的功能。复制下面的代码片段，并替换在“编号”目录中自动创建的“Class1.cs”文件的内容 。将“Class1.cs”文件重命名为“PrimeService.cs”。

```
namespace System.Numbers
{
    public class PrimeService
    {
        public bool IsPrime(int candidate)
        {
            if (candidate < 2)
            {
                return false;
            }

            for (int divisor = 2; divisor <= Math.Sqrt(candidate); ++divisor)
            {
                if (candidate % divisor == 0)
                {
                    return false;
                }
            }
            return true;
        }
    }
}
```

TIP

值得一提的是，`Numbers` 类库是有意添加到 `System` 命名空间中的。因此，无需 `using System;` 命名空间声明即可访问 System.Math。有关详细信息，请参阅命名空间（C# 参考）。

创建测试项目

使用 `dotnet new xunit` 命令，在同一命令提示符下创建两个新的“xUnit 测试项目(.NET Core)”模板：

```
dotnet new xunit -n XUnit.Coverlet.Collector
```

```
dotnet new xunit -n XUnit.Coverlet.MSBuild
```

这两个新创建的 xUnit 测试项目都需要添加 Numbers 类库的项目引用。这是为了使测试项目有权访问 PrimeService 以便进行测试。在命令提示符下，使用 `dotnet add` 命令：

```
dotnet add XUnit.Coverlet.Collector\XUnit.Coverlet.Collector.csproj reference Numbers\Numbers.csproj
```

```
dotnet add XUnit.Coverlet.MSBuild\XUnit.Coverlet.MSBuild.csproj reference Numbers\Numbers.csproj
```

MSBuild 项目命名正确，因为它依赖于 coverlet.msbuild NuGet 包。通过运行 `dotnet add package` 命令添加此包依赖项：

```
cd XUnit.Coverlet.MSBuild && dotnet add package coverlet.msbuild && cd ..
```

之前的命令更改了有效作用于 MSBuild 测试项目的目录，然后添加了 NuGet 包。完成此操作后，它会更改目录，向上执行一个级别。

打开两个 UnitTest1.cs 文件，并将其内容替换为以下代码片段。将 UnitTest1.cs 文件重命名为 PrimeServiceTests.cs 。

```
using System.Numbers;
using Xunit;

namespace XUnit.Coverlet
{
    public class PrimeServiceTests
    {
        readonly PrimeService _primeService;

        public PrimeServiceTests() => _primeService = new PrimeService();

        [
            Theory,
            InlineData(-1), InlineData(0), InlineData(1)
        ]
        public void IsPrime_ValuesLessThan2_ReturnFalse(int value) =>
            Assert.False(_primeService.IsPrime(value), $"{value} should not be prime");

        [
            Theory,
            InlineData(2), InlineData(3), InlineData(5), InlineData(7)
        ]
        public void IsPrime_PrimesLessThan10_ReturnTrue(int value) =>
            Assert.True(_primeService.IsPrime(value), $"{value} should be prime");

        [
            Theory,
            InlineData(4), InlineData(6), InlineData(8), InlineData(9)
        ]
        public void IsPrime_NonPrimesLessThan10_ReturnFalse(int value) =>
            Assert.False(_primeService.IsPrime(value), $"{value} should not be prime");
    }
}
```

创建解决方案

在命令提示符下，创建一个用于封装类库和两个测试项目的新解决方案。使用 `dotnet sln` 命令：

```
dotnet new sln -n XUnit.Coverage
```

这会在 UnitTestingCodeCoverage 目录中创建新的解决方案文件名 `XUnit.Coverage` 。将项目添加到解决方案的根。

- Linux
- Windows

```
dotnet sln XUnit.Coverage.sln add **/*.csproj --in-root
```

使用 `dotnet build` 命令生成解决方案：

```
dotnet build
```

如果生成成功，则已创建了三个项目，正确引用了项目和包，并正确更新了源代码。做得不错！

工具

代码覆盖率工具有两种类型：

- **数据收集器**：数据收集器监视测试执行并收集有关测试运行的信息。它们以各种输出格式(例如 XML 和 JSON)报告收集的信息。有关详细信息，请参阅第一个数据收集器。
- **报表生成器**：使用从测试运行收集的数据生成报表，通常为带样式的 HTML。

本部分重点介绍数据收集器工具。若要通过 Coverlet 获得代码覆盖率，现有单元测试项目必须具有相应的包依赖项，或者依赖于 .NET 全局工具和对应的 coverlet.console NuGet 包。

与 .NET 测试集成

默认情况下，xUnit 测试项目模板已与 coverlet.collector 集成。在命令提示符下，将目录更改为 XUnit.Coverlet.Collector 项目，并运行 `dotnet test` 命令：

```
cd XUnit.Coverlet.Collector && dotnet test --collect:"XPlat Code Coverage"
```

> **NOTE**
> `"XPlat Code Coverage"` 参数是与 Coverlet 中的数据收集器对应的易记名称。此名称是必需的，但不区分大小写。

作为 `dotnet test` 运行的一部分，生成的 coverage.cobertura.xml 文件输出到 TestResults 目录 。该 XML 文件包含结果。这是一个依赖于 .NET CLI 的跨平台选项，非常适用于不可使用 MSBuild 的生成系统。

下面是 coverage.cobertura.xml 文件的示例 。

```
<?xml version="1.0" encoding="utf-8"?>
<coverage line-rate="1" branch-rate="1" version="1.9" timestamp="1592248008"
          lines-covered="12" lines-valid="12" branches-covered="6" branches-valid="6">
  <sources>
    <source>C:\</source>
  </sources>
  <packages>
    <package name="Numbers" line-rate="1" branch-rate="1" complexity="6">
      <classes>
        <class name="Numbers.PrimeService" line-rate="1" branch-rate="1" complexity="6"
               filename="Numbers\PrimeService.cs">
          <methods>
            <method name="IsPrime" signature="(System.Int32)" line-rate="1"
                    branch-rate="1" complexity="6">
              <lines>
                <line number="8" hits="11" branch="False" />
                <line number="9" hits="11" branch="True" condition-coverage="100% (2/2)">
                  <conditions>
                    <condition number="7" type="jump" coverage="100%" />
                  </conditions>
                </line>
                <line number="10" hits="3" branch="False" />
                <line number="11" hits="3" branch="False" />
                <line number="14" hits="22" branch="True" condition-coverage="100% (2/2)">
                  <conditions>
                    <condition number="57" type="jump" coverage="100%" />
                  </conditions>
                </line>
                <line number="15" hits="7" branch="False" />
                <line number="16" hits="7" branch="True" condition-coverage="100% (2/2)">
                  <conditions>
                    <condition number="27" type="jump" coverage="100%" />
                  </conditions>
                </line>
                <line number="17" hits="4" branch="False" />
                <line number="18" hits="4" branch="False" />
                <line number="20" hits="3" branch="False" />
                <line number="21" hits="4" branch="False" />
```

```
            <line number="23" hits="11" branch="False" />
          </lines>
        </method>
      </methods>
      <lines>
        <line number="8" hits="11" branch="False" />
        <line number="9" hits="11" branch="True" condition-coverage="100% (2/2)">
          <conditions>
            <condition number="7" type="jump" coverage="100%" />
          </conditions>
        </line>
        <line number="10" hits="3" branch="False" />
        <line number="11" hits="3" branch="False" />
        <line number="14" hits="22" branch="True" condition-coverage="100% (2/2)">
          <conditions>
            <condition number="57" type="jump" coverage="100%" />
          </conditions>
        </line>
        <line number="15" hits="7" branch="False" />
        <line number="16" hits="7" branch="True" condition-coverage="100% (2/2)">
          <conditions>
            <condition number="27" type="jump" coverage="100%" />
          </conditions>
        </line>
        <line number="17" hits="4" branch="False" />
        <line number="18" hits="4" branch="False" />
        <line number="20" hits="3" branch="False" />
        <line number="21" hits="4" branch="False" />
        <line number="23" hits="11" branch="False" />
      </lines>
    </class>
  </classes>
 </package>
</packages>
</coverage>
```

TIP

有一种替代方法：如果生成系统已使用 MSBuild，则你可以使用 MSBuild 包。在命令提示符下，将目录更改为 XUnit.Coverlet.MSBuild 项目，并运行 `dotnet test` 命令：

```
dotnet test /p:CollectCoverage=true /p:CoverletOutputFormat=cobertura
```

生成的 coverage.cobertura.xml 文件为输出。可按照此处的 msbuild 集成指南操作

生成报告

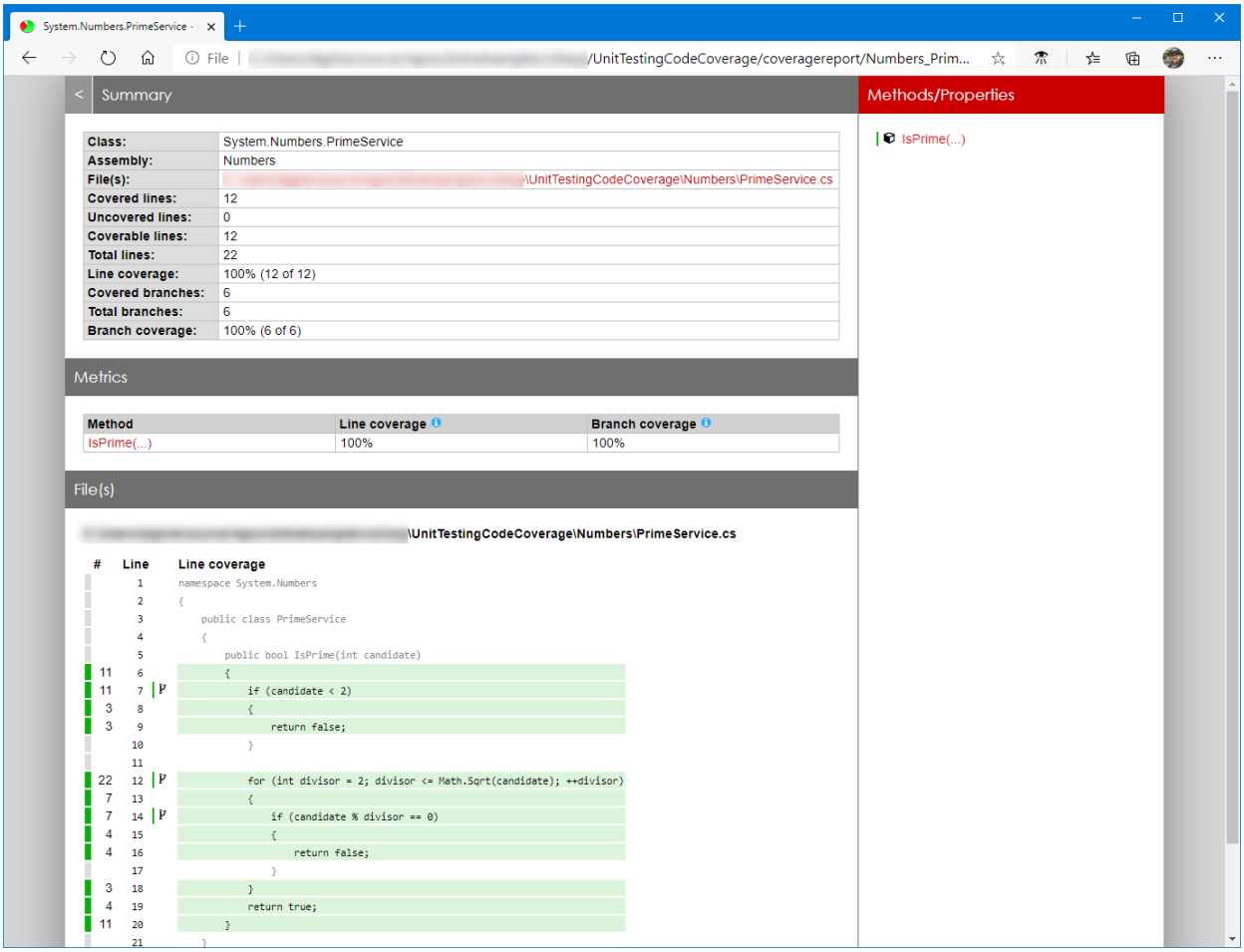
现在，你既可从单元测试运行收集数据，就可以使用 ReportGenerator 来生成报表。若要将 ReportGenerator NuGet 包安装为 .NET 全局工具，请使用 `dotnet tool install` 命令：

```
dotnet tool install -g dotnet-reportgenerator-globaltool
```

给定先前的测试运行得到的输出 coverage.cobertura.xml 文件，运行该工具并提供所需的选项。

```
reportgenerator
-reports:"Path\To\TestProject\TestResults\{guid}\coverage.cobertura.xml"
-targetdir:"coveragereport"
-reporttypes:Html
```

运行此命令后，HTML 文件表示生成的报表。



请参阅

- Visual Studio 单元测试代码覆盖率
- GitHub - Coverlet 存储库
- GitHub - ReportGenerator 存储库
- ReportGenerator 项目网站
- .NET CLI 测试命令
- dotnet-coverage
- 示例源代码

后续步骤

单元测试最佳做法

通过 dotnet vstest 测试已发布的输出

2021/11/16 •

可以使用 `dotnet vstest` 命令测试已发布的输出。这将适用于 xUnit、MSTest 和 NUnit 测试。只需找到属于已发布输出的 DLL 文件，然后运行：

```
dotnet vstest <MyPublishedTests>.dll
```

其中，`<MyPublishedTests>` 是已发布的测试项目的名称。

示例

下面的命令演示在已发布的 DLL 上运行测试。

```
dotnet new mstest -o MyProject.Tests
cd MyProject.Tests
dotnet publish -o out
dotnet vstest out/MyProject.Tests.dll
```

> **NOTE**
> 注意：如果你的应用面向 `netcoreapp` 之外的框架，仍可通过传入带有框架标志的目标框架来运行 `dotnet vstest` 命令。例如，`dotnet vstest <MyPublishedTests>.dll --Framework:".NETFramework,Version=v4.6"`。在 Visual Studio 2017 Update 5 及更高版本中，自动检测所需的框架。

另请参阅

- 使用 dotnet 测试和 xUnit 进行单元测试
- 使用 dotnet 测试和 NUnit 进行单元测试
- 使用 dotnet test 和 MSTest 进行单元测试

.NET 中的安全性

2021/11/16 •

公共语言运行时和 .NET 提供了许多有用的类和服务，使开发人员能够编写安全代码、使用加密以及实现基于角色的安全性。

本节内容

- 安全性的基础概念
  概述公共语言运行时的安全功能。
- 基于角色的安全性
  描述如何在代码中与基于角色的安全性进行交互。
- 加密模型
  概述 .NET 提供的加密服务。
- 安全编码准则
  介绍创建可靠的 .NET 应用程序的一些最佳做法。

相关章节

开发指南
提供了有关应用程序开发的所有关键技术区域和任务（包括创建、配置、调试、保护和部署应用程序）的指南，以及有关动态编程、互操作性、扩展性、内存管理和线程处理的信息。

什么是"托管代码"？

2021/11/16 •

使用 .NET 时，我们经常会遇到"托管代码"这个术语。本文档解释这个术语的含义及其更多相关信息。

简而言之，托管代码就是执行过程交由运行时管理的代码。在这种情况下，相关的运行时称为公共语言运行时 (CLR)，不管使用的是哪种实现（例如 Mono、.NET Framework 或 .NET Core/.NET 5+）。CLR 负责提取托管代码、将其编译成机器代码，然后执行它。除此之外，运行时还提供多个重要服务，例如自动内存管理、安全边界、类型安全，等等。

相反，如果运行 C/C++ 程序，则运行的代码也称为"非托管代码"。在非托管环境中，程序员需要亲自负责处理相当多的事情。实际的程序在本质上是操作系统 (OS) 载入内存，然后启动的二进制代码。其他任何工作 - 从内存管理到安全考虑因素 - 对于程序员来说是一个不小的负担。

托管代码是使用可在 .NET 上运行的一种高级语言（例如 C#、Visual Basic、F# 等）编写的。使用相应的编译器编译以这些语言编写的代码时，无法获得机器代码，而是获得 **中间语言** 代码，然后运行时会对其进行编译并将其执行。C++ 是这条规则的一个例外，因为它也能够生成可在 Windows 上运行的本机非托管二进制代码。

中间语言和执行

什么是"中间语言"（简称 IL）？ 中间语言是编译使用高级 .NET 语言编写的代码后获得的结果。对使用其中一种语言编写的代码进行编译后，即可获得 IL 所生成的二进制代码。必须注意，IL 独立于在运行时顶层运行的任何特定语言；行业甚至为它单独制定了规范，如果有需要，你可以阅读该规范。

从高级代码生成 IL 后，你很有可能想要运行它。CLR 此时将接管工作，启动 实时 (JIT) 编译过程，或者将代码从 IL 实时 编译成可以真正在 CPU 上运行的机器代码。这样，CLR 就能确切地知道代码的作用，并可以有效地 *管理* 代码。

中间语言有时也称为公共中间语言 (CIL) 或 Microsoft 中间语言 (MSIL)。

托管代码互操作性

当然，CLR 允许越过托管与非托管环境之间的边界，同时，即使在基类库中，也有很多代码可以做到这一点。这称为 **互操作性**，简称 **interop**。例如，使用这些机制可以包装某个非托管库以及调用该库。但是，请务必注意，如果采取这种方法，当代码越过运行时的边界时，实际的执行管理将再次交接到托管代码，因而需要遵守相同的限制。

与此类似，C# 语言可让你利用所谓的 **不安全上下文**（指定执行过程不由 CLR 管理的代码片段），在代码中直接使用非托管构造，例如指针。

更多资源

- .NET Framework 概述
- 不安全代码和指针
- 本机互操作性

自动内存管理

2021/11/16 •

自动内存管理是公共语言运行时在托管执行过程中提供的服务之一。公共语言运行时的垃圾回收器为应用程序管理内存的分配和释放。对开发人员而言，这就意味着在开发托管应用程序时不必编写执行内存管理任务的代码。自动内存管理可解决常见问题，例如，忘记释放对象并导致内存泄漏，或尝试访问已释放对象的内存。本节描述垃圾回收器如何分配和释放内存。

分配内存

初始化新进程时，运行时会为进程保留一个连续的地址空间区域。这个保留的地址空间被称为托管堆。托管堆维护着一个指针，用它指向将在堆中分配的下一个对象的地址。最初，该指针设置为指向托管堆的基址。托管堆上包含了所有引用类型。应用程序创建第一个引用类型时，将为托管堆的基址中的类型分配内存。应用程序创建下一个对象时，垃圾回收器在紧接第一个对象后面的地址空间内为它分配内存。只要地址空间可用，垃圾回收器就会继续以这种方式为新对象分配空间。

从托管堆中分配内存要比非托管内存分配速度快。由于运行时通过为指针添加值来为对象分配内存，所以这几乎和从堆栈中分配内存一样快。另外，由于连续分配的新对象在托管堆中是连续存储，所以应用程序可以快速访问这些对象。

释放内存

垃圾回收器的优化引擎根据所执行的分配决定执行回收的最佳时间。垃圾回收器在执行回收时，会释放应用程序不再使用的对象的内存。它通过检查应用程序的根来确定不再使用的对象。每个应用程序都有一组根。每个根或者引用托管堆中的对象，或者设置为空。应用程序的根包含线程堆栈上的静态字段、局部变量和参数以及 CPU 寄存器。垃圾回收器可以访问由实时 (JIT) 编译器和运行时维护的活动根的列表。垃圾回收器对照此列表检查应用程序的根，并在此过程中创建一个图表，在其中包含所有可从这些根中访问的对象。

不在该图表中的对象将无法从应用程序的根中访问。垃圾回收器会考虑无法访问的对象垃圾，并释放为它们分配的内存。在回收中，垃圾回收器检查托管堆，查找无法访问对象所占据的地址空间块。发现无法访问的对象时，它就使用内存复制功能来压缩内存中可以访问的对象，释放分配给不可访问对象的地址空间块。在压缩了可访问对象的内存后，垃圾回收器就会做出必要的指针更正，以便应用程序的根指向新地址中的对象。它还将托管堆指针定位至最后一个可访问对象之后。请注意，只有在回收发现大量的无法访问的对象时，才会压缩内存。如果托管堆中的所有对象均未被回收，则不需要压缩内存。

为了改进性能，运行时为单独堆中的大型对象分配内存。垃圾回收器会自动释放大型对象的内存。但是，为了避免移动内存中的大型对象，不会压缩此内存。

级别和性能

为优化垃圾回收器的性能，将托管堆分为三代：第 0 代、第 1 代和第 2 代。运行时的垃圾回收算法基于以下几个普遍原理，这些垃圾回收方案的原理已在计算机软件业通过实验得到了证实。首先，压缩托管堆的一部分内存要比压缩整个托管堆速度快。其次，较新的对象生存期较短，而较旧的对象生存期则较长。最后，较新的对象趋向于相互关联，并且大致同时由应用程序访问。

运行时的垃圾回收器将新对象存储在第 0 级中。在应用程序生存期的早期创建的对象如果未被回收，则被升级并存储在第 1 级和第 2 级中。本主题中稍后介绍了对象升级过程。因为压缩托管堆的一部分要比压缩整个托管堆速度快，所以此方案允许垃圾回收器在每次执行回收时释放特定级别的内存，而不是整个托管堆的内存。

实际上，垃圾回收器在第 0 级托管堆已满时执行回收。如果应用程序在第 0 级托管堆已满时尝试新建对象，垃圾回收器将会发现第 0 级托管堆中没有可分配给该对象的剩余地址空间。垃圾回收器执行回收，尝试为对象释放

第 0 级托管堆中的地址空间。垃圾回收器从检查第 0 级托管堆中的对象(而不是托管堆中的所有对象)开始执行回收。这是最有效的途径，因为新对象的生存期往往较短，并且期望在执行回收时，应用程序不再使用第 0 级托管堆中的许多对象。另外，单独回收第 0 级托管堆通常可以回收足够的内存，这样，应用程序便可以继续创建新对象。

垃圾回收器执行第 0 级托管堆的回收后，会压缩可访问对象的内存，如本主题前面的释放内存中所述。然后，垃圾回收器升级这些对象，并考虑第 1 级托管堆的这一部分。因为未被回收的对象往往具有较长的生存期，所以将它们升级至更高的级别很有意义。因此，垃圾回收器在每次执行第 0 级托管堆的回收时，不必重新检查第 1 级和第 2 级托管堆中的对象。

在执行第 0 级托管堆的首次回收并把可访问的对象升级至第 1 级托管堆后，垃圾回收器将考虑第 0 级托管堆的其余部分。它将继续为第 0 级托管堆中的新对象分配内存，直至第 0 级托管堆已满并需执行另一回收为止。这时，垃圾回收器的优化引擎会决定是否需要检查较旧的级别中的对象。例如，如果第 0 级托管堆的回收没有回收足够的内存，不能使应用程序成功完成创建新对象的尝试，垃圾回收器就会先执行第 1 级托管堆的回收，然后再执行第 2 级托管堆的回收。如果这样仍不能回收足够的内存，垃圾回收器将执行第 2、1 和 0 级托管堆的回收。每次回收后，垃圾回收器都会压缩第 0 级托管堆中的可访问对象并将它们升级至第 1 级托管堆。第 1 级托管堆中未被回收的对象将会升级至第 2 级托管堆。由于垃圾回收器只支持三个级别，因此第 2 级托管堆中未被回收的对象会继续保留在第 2 级托管堆中，直到在将来的回收中确定它们为无法访问为止。

为非托管资源释放内存

对于应用程序创建的大多数对象，可以依赖垃圾回收器自动执行必要的内存管理任务。但是，非托管资源需要显式清除。最常用的非托管资源类型是包装操作系统资源的对象，例如，文件句柄、窗口句柄或网络连接。虽然垃圾回收器可以跟踪封装非托管资源的托管对象的生存期，但却无法具体了解如何清理资源。创建封装非托管资源的对象时，建议在公共 Dispose 方法中提供必要的代码以清理非托管资源。通过提供 Dispose 方法，对象的用户可以在使用完对象后显式释放其内存。使用封装非托管资源的对象时，应该了解 Dispose 并在必要时调用它。有关清理非托管资源的详细信息和实现 Dispose 的设计模式示例，请参见 垃圾回收。

请参阅

- GC
- 垃圾回收
- 托管执行过程

清理未托管资源

2021/11/16 •

对于应用创建的大多数对象，可以依赖 .NET 垃圾回收器来进行内存管理。但是，如果创建包含非托管资源的对象，则当你使用完非托管资源后，必须显式释放这些资源。最常用的非托管资源类型是包装操作系统资源的对象，如文件、窗口、网络连接或数据库连接。虽然垃圾回收器可以跟踪封装非托管资源的对象的生存期，但无法了解如何发布并清理这些非托管资源。

如果你的类型使用非托管资源，则应执行以下操作：

- 实现清理模式。这要求你提供 IDisposable.Dispose 实现以启用非托管资源的确定性释放。当不再需要此对象（或其使用的资源）时，类型使用者可调用 Dispose。Dispose 方法立即释放非托管资源。
- 在类型使用者忘记调用 Dispose 的情况下，请提供一种方法来释放非托管资源。有两种方法可以实现此目的：
  - 使用安全句柄包装非托管资源。这是推荐采用的方法。安全句柄派生自 System.Runtime.InteropServices.SafeHandle 抽象类，并包含可靠的 Finalize 方法。在使用安全句柄时，只需实现 IDisposable 接口并在 Dispose 实现中调用安全句柄的 IDisposable.Dispose 方法。如果未调用安全句柄的 Dispose 方法，则垃圾回收器将自动调用安全句柄的终结器。

    —或—

  - 重写 Object.Finalize 方法。当类型使用者无法调用 IDisposable.Dispose 以确定性地释放非托管资源时，终止会启用对非托管资源的非确定性释放。通过重写 Object.Finalize 方法来定义终结器。

> **WARNING**
> 但是，由于对象终止是一项复杂且易出错的操作，建议你使用安全句柄，而不是提供你自己的终结器。

然后，类型使用者可直接调用 IDisposable.Dispose 实现以释放非托管资源使用的内存。在正确实现 Dispose 方法时，安全句柄的 Finalize 方法或 Object.Finalize 方法的重写会在未调用 Dispose 方法的情况下阻止清理资源。

本节内容

实现 Dispose 方法介绍如何实现用于释放非托管资源的释放模式。

使用实现 `IDisposable` 的对象介绍类型使用者如何确保调用其 Dispose 实现。建议使用 C# `using`（或 Visual Basic `Using`）语句来执行此操作。

参考

类/结构	说明
System.IDisposable	定义用于释放非托管资源的 Dispose 方法。
Object.Finalize	如果 Dispose 方法未释放非托管资源，则准备对象终止。
GC.SuppressFinalize	取消终止。通常，从 `Dispose` 方法调用此方法来阻止执行终结器。

实现 Dispose 方法

2021/11/16 •

实现 Dispose 方法主要用于释放非托管资源。处理 IDisposable 实现的实例成员时，通常会级联 Dispose 调用。实现 Dispose 有其他原因，例如，为了释放已分配的内存、删除已添加到集合中的项，或发出释放已获取的锁的信号。

.NET 垃圾回收器不会分配或释放非托管内存。对象释放模式（称为“释放模式”）会对对象生存期强制施加顺序。释放模式用于实现 IDisposable 接口的对象，在与文件和管道句柄、注册表句柄、等待句柄或指向非托管内存块的指针交互时较为常见。这是因为垃圾回收器无法回收非托管对象。

若要帮助确保始终适当地清理资源，Dispose 方法应为幂等，这样可以多次调用而不引发异常。此外，Dispose 的后续调用不应执行任何操作。

为 GC.KeepAlive 方法提供的代码示例演示了垃圾回收如何引起终结器运行，而对该对象或其成员的非托管引用仍在使用中。利用 GC.KeepAlive 使对象从当前例程开始到调用此方法的那一刻为止都不适合进行垃圾回收，这是可行的。

安全句柄

编写对象终结器的代码是一项复杂的任务，如果处理不好可能会出现问题。因此，建议你构造 System.Runtime.InteropServices.SafeHandle 对象，而非实现终结器。

System.Runtime.InteropServices.SafeHandle 是一种抽象托管类型，该类型包装了可标识非托管资源的 System.IntPtr。在 Windows 上，它可能标识一个句柄，而在 Unix 上则可能标识一个文件描述符。它提供了所有必要的逻辑，以确保在处理 `SafeHandle` 或删除对 `SafeHandle` 的所有引用并最终完成 `SafeHandle` 实例时，只释放该资源一次。

System.Runtime.InteropServices.SafeHandle 是抽象基类。派生类会为不同类型的句柄提供特定实例。这些派生类可验证 System.IntPtr 的哪些值被视为无效，以及如何实际释放句柄。例如，SafeFileHandle 派生自 `SafeHandle` 以包装可标识打开的文件句柄/描述符的 `IntPtrs`，并重写其 SafeHandle.ReleaseHandle() 方法来关闭它（通过 Unix 上的 `close` 函数或 Windows 上的 `CloseHandle` 函数）。.NET 库中创建非托管资源的大多数 API 会将其包装在 `SafeHandle` 中，并根据需要返回此 `SafeHandle`，而不是返回原始指针。在与非托管组件进行交互并获取非托管资源的 `IntPtr` 的情况下，你可以创建自己的 `SafeHandle` 类型进行包装。因此，极少数非 `SafeHandle` 类型需要实现终结器；大多数可释放模式实现最终只包装其他受管理资源，其中某些资源可能是 `SafeHandle`。

Microsoft.Win32.SafeHandles 命名空间中的以下派生类提供安全句柄：

- 用于文件、内存映射文件和管道的 SafeFileHandle、SafeMemoryMappedFileHandle 和 SafePipeHandle 类。
- 用于内存视图的 SafeMemoryMappedViewHandle 类。
- 用于加密构造的 SafeNCryptKeyHandle、SafeNCryptProviderHandle 和 SafeNCryptSecretHandle 类。
- 用于注册表项的 SafeRegistryHandle 类。
- 用于等待句柄的 SafeWaitHandle 类。

Dispose() 和 Dispose(bool)

IDisposable 接口需要实现单个无参数的方法 Dispose。此外，任何非密封类都应具有要实现的附加 `Dispose(bool)` 重载方法：

- 一种没有参数的 `public` 非虚拟的（Visual Basic 中的 `NonInheritable`）IDisposable.Dispose 实现。

- `protected virtual`（Visual Basic 中为 `Overridable`）`Dispose` 方法，其签名为：

```
protected virtual void Dispose(bool disposing)
{
}
```

```
Protected Overridable Sub Dispose(disposing As Boolean)
End Sub
```

> **IMPORTANT**
> 从终结器调用时，`disposing` 参数应为 `false`，从 IDisposable.Dispose 方法调用时应为 `true`。换言之，确定情况下调用时为 `true`，而在不确定情况下调用时为 `false`。

Dispose() 方法

由于 `public`、非虚拟（Visual Basic 中为 `NonInheritable`）、无参数的 `Dispose` 方法由该类型的使用者调用，因此其用途是释放非托管资源，执行常规清理，以及指示终结器（如果存在）不必运行。释放与托管对象关联的实际内存始终是垃圾回收器的域。因此，它具有标准实现：

```
public void Dispose()
{
   // Dispose of unmanaged resources.
   Dispose(true);
   // Suppress finalization.
   GC.SuppressFinalize(this);
}
```

```
Public Sub Dispose() _
           Implements IDisposable.Dispose
    ' Dispose of unmanaged resources.
    Dispose(True)
    ' Suppress finalization.
    GC.SuppressFinalize(Me)
End Sub
```

`Dispose` 方法执行所有对象清理，使垃圾回收器不再需要调用对象的 Object.Finalize 重写。因此，调用 SuppressFinalize 方法会阻止垃圾回收器运行终结器。如果类型没有终结器，则对 GC.SuppressFinalize 的调用不起作用。请注意，实际清除由 `Dispose(bool)` 方法重载执行。

Dispose(bool) 方法重载

在重载中，`disposing` 参数是一个 Boolean，它指示方法调用是来自 Dispose 方法（其值为 `true`）还是来自终结器（其值为 `false`）。

方法的主体包含两个代码块：

- 释放非托管资源的块。无论 `disposing` 参数的值如何，都会执行此块。
- 释放托管资源的条件块。如果 `disposing` 的值为 `true`，则执行此块。它释放的托管资源可包括：
  - 实现 **IDisposable 的托管对象。**可用于调用其 Dispose 实现（级联释放）的条件块。如果你已使用 System.Runtime.InteropServices.SafeHandle 的派生类来包装非托管资源，则应在此处调用 SafeHandle.Dispose() 实现。
  - **占用大量内存或使用短缺资源的托管对象。**将大型托管对象引用分配到 `null`，使它们更有可能

无法访问。相比以非确定性方式回收它们，这样做释放的速度更快，此操作通常在条件块之外完成。

如果方法调用来自终结器，则应仅执行释放非托管资源的代码。实施者负责确保假路径不会与可能已被回收的托管对象交互。这一点很重要，因为垃圾回收器在终止期间销毁托管对象的顺序是不确定的。

级联释放调用

如果你的类拥有一个字段或属性，并且其类型实现 IDisposable，则包含类本身还应实现 IDisposable。实例化 IDisposable 实现并将其存储为实例成员的类，也负责清理。这是为了帮助确保引用的可释放类型可通过 Dispose 方法明确执行清理。在本例中，类为 `sealed`（Visual Basic 中为 `NotInheritable`）。

```
public sealed class Foo : IDisposable
{
    private readonly IDisposable _bar;

    public Foo()
    {
        _bar = new Bar();
    }

    public void Dispose()
    {
        _bar?.Dispose();
    }
}
```

```
Public NotInheritable Class Foo
    Implements IDisposable

    Private ReadOnly _bar As IDisposable

    Public Sub New()
        _bar = New Bar()
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        _bar.Dispose()
    End Sub
End Class
```

实现释放模式

所有非密封类或（未修改为 `NotInheritable` 的 Visual Basic 类）都应被视为潜在的基类，因为它们可以被继承。如果为任何潜在基类实现释放模式，则必须提供以下内容：

- 调用 Dispose 方法的 `Dispose(bool)` 实现。
- 执行实际清理的 `Dispose(bool)` 方法。
- 从包装非托管资源的 SafeHandle 派生的类（推荐），或对 Object.Finalize 方法的重写。SafeHandle 类提供了终结器，因此你无需自行编写。

> **IMPORTANT**
> 基类可以只引用托管对象，并实现释放模式。在这些情况下，不需要终结器。仅当直接引用非托管资源时，才需要终结器。

以下是一个常规模式，用于实现使用安全句柄的基类的释放模式：

NOTE

上一个示例使用 SafeFileHandle 对象阐释模式；可以使用派生自 SafeHandle 的任何对象来替代。请注意，该示例不会正确实例化其 SafeFileHandle 对象。

以下是一个常规模式，用于实现重写 Object.Finalize 的基类的释放模式。

TIP

在 C# 中，通过重写 Object.Finalize 创建一个终结器。在 Visual Basic 中，这是通过 `Protected Overrides Sub Finalize()` 完成的。

实现派生类的释放模式

从实现 IDisposable 接口的类派生的类不应实现 IDisposable，因为 IDisposable.Dispose 的基类实现由其派生类继承。若要清理派生类，请提供以下内容：

- `protected override void Dispose(bool)` 方法，用于替代基类方法并执行派生类的实际清理。此方法还必须调用基类的 `base.Dispose(bool)`（Visual Basic 中为 `MyBase.Dispose(bool)`）方法，并传递参数的释放状态。
- 从包装非托管资源的 SafeHandle 派生的类（推荐），或对 Object.Finalize 方法的重写。SafeHandle 类提供了一个使你无需编写代码的终结器。如果你提供了终结器，它必须调用 `disposing` 参数为 `false` 的 `Dispose(bool)` 重载。

以下是一个常规模式，用于实现使用安全句柄的派生类的释放模式：

NOTE

上一个示例使用 SafeFileHandle 对象阐释模式；可以使用派生自 SafeHandle 的任何对象来替代。请注意，该示例不会正确实例化其 SafeFileHandle 对象。

以下是一个常规模式，用于实现重写 Object.Finalize 的派生类的释放模式：

使用安全句柄实现释放模式

下面的示例阐释了基类 `DisposableStreamResource` 的释放模式，此模式使用安全句柄封装非托管资源。它定义

`DisposableStreamResource` 类，该类使用 SafeFileHandle 包装表示打开的文件的 Stream 对象。此类还包含一个属性 `Size`，该属性返回文件流中的总字节数。

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

public class DisposableStreamResource : IDisposable
{
    // Define constants.
    protected const uint GENERIC_READ = 0x80000000;
    protected const uint FILE_SHARE_READ = 0x00000001;
    protected const uint OPEN_EXISTING = 3;
    protected const uint FILE_ATTRIBUTE_NORMAL = 0x80;
    private const int INVALID_FILE_SIZE = unchecked((int)0xFFFFFFFF);

    // Define Windows APIs.
    [DllImport("kernel32.dll", EntryPoint = "CreateFileW", CharSet = CharSet.Unicode)]
    protected static extern SafeFileHandle CreateFile(
        string lpFileName, uint dwDesiredAccess,
        uint dwShareMode, IntPtr lpSecurityAttributes,
        uint dwCreationDisposition, uint dwFlagsAndAttributes,
        IntPtr hTemplateFile);

    [DllImport("kernel32.dll")]
    private static extern int GetFileSize(
        SafeFileHandle hFile, out int lpFileSizeHigh);

    // Define locals.
    private bool _disposed = false;
    private readonly SafeFileHandle _safeHandle;
    private readonly int _upperWord;

    public DisposableStreamResource(string fileName)
    {
        if (string.IsNullOrWhiteSpace(fileName))
        {
            throw new ArgumentException("The fileName cannot be null or an empty string");
        }

        _safeHandle = CreateFile(
            fileName, GENERIC_READ, FILE_SHARE_READ, IntPtr.Zero,
            OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, IntPtr.Zero);

        // Get file size.
        Size = GetFileSize(_safeHandle, out _upperWord);
        if (Size == INVALID_FILE_SIZE)
        {
            Size = -1;
        }
        else if (_upperWord > 0)
        {
            Size = (((long)_upperWord) << 32) + Size;
        }
    }

    public long Size { get; }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (_disposed)
```

```
        {
            return;
        }

        // Dispose of managed resources here.
        if (disposing)
        {
            _safeHandle?.Dispose();
        }

        // Dispose of any unmanaged resources not wrapped in safe handles.

        _disposed = true;
    }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.IO

Public Class DisposableStreamResource : Implements IDisposable
    ' Define constants.
    Protected Const GENERIC_READ As UInteger = &H80000000UI
    Protected Const FILE_SHARE_READ As UInteger = &H0I
    Protected Const OPEN_EXISTING As UInteger = 3
    Protected Const FILE_ATTRIBUTE_NORMAL As UInteger = &H80
    Private Const INVALID_FILE_SIZE As Integer = &HFFFFFFFF

    ' Define Windows APIs.
    Protected Declare Function CreateFile Lib "kernel32" Alias "CreateFileA" (
        lpFileName As String, dwDesiredAccess As UInt32,
        dwShareMode As UInt32, lpSecurityAttributes As IntPtr,
        dwCreationDisposition As UInt32, dwFlagsAndAttributes As UInt32,
        hTemplateFile As IntPtr) As SafeFileHandle

    Private Declare Function GetFileSize Lib "kernel32" (
        hFile As SafeFileHandle, ByRef lpFileSizeHigh As Integer) As Integer

    ' Define locals.
    Private disposed As Boolean = False
    Private ReadOnly safeHandle As SafeFileHandle
    Private ReadOnly upperWord As Integer

    Public Sub New(fileName As String)
        If String.IsNullOrWhiteSpace(fileName) Then
            Throw New ArgumentNullException("The fileName cannot be null or an empty string")
        End If

        safeHandle = CreateFile(
            fileName, GENERIC_READ, FILE_SHARE_READ, IntPtr.Zero,
            OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, IntPtr.Zero)

        ' Get file size.
        Size = GetFileSize(safeHandle, upperWord)
        If Size = INVALID_FILE_SIZE Then
            Size = -1
        ElseIf upperWord > 0 Then
            Size = (CLng(upperWord) << 32) + Size
        End If
    End Sub

    Public ReadOnly Property Size As Long

    Public Sub Dispose() _
              Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub

    Protected Overridable Sub Dispose(disposing As Boolean)
        If disposed Then Exit Sub

        ' Dispose of managed resources here.
        If disposing Then
            safeHandle.Dispose()
        End If

        ' Dispose of any unmanaged resources not wrapped in safe handles.

        disposed = True
    End Sub
End Class
```

使用安全句柄实现派生类的释放模式

下面的示例阐释派生类 `DisposableStreamResource2` 的释放模式，该类继承自上一个示例中显示的 `DisposableStreamResource` 类。此类额外添加一种方法（即 `WriteFileInfo`），并使用 SafeFileHandle 对象包装可写文件的句柄。

```
using Microsoft.Win32.SafeHandles;
using System;
using System.Runtime.InteropServices;

public class DisposableStreamResource2 : DisposableStreamResource
{
    // Define additional constants.
    protected const uint GENERIC_WRITE = 0x40000000;
    protected const uint OPEN_ALWAYS = 4;

    // Define additional APIs.
    [DllImport("kernel32.dll")]
    protected static extern bool WriteFile(
        SafeFileHandle safeHandle, string lpBuffer,
        int nNumberOfBytesToWrite, out int lpNumberOfBytesWritten,
        IntPtr lpOverlapped);

    // To detect redundant calls
    private bool _disposed = false;
    private bool _created = false;
    private SafeFileHandle _safeHandle;
    private readonly string _fileName;

    public DisposableStreamResource2(string fileName) : base(fileName) => _fileName = fileName;

    public void WriteFileInfo()
    {
        if (!_created)
        {
            _safeHandle = CreateFile(
                @".\FileInfo.txt", GENERIC_WRITE, 0, IntPtr.Zero,
                OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, IntPtr.Zero);

            _created = true;
        }

        string output = $"{_fileName}: {Size:N0} bytes\n";
        _ = WriteFile(_safeHandle, output, output.Length, out _, IntPtr.Zero);
    }

    protected override void Dispose(bool disposing)
    {
        if (_disposed)
        {
            return;
        }

        // Release any managed resources here.
        if (disposing)
        {
            // Dispose managed state (managed objects).
            _safeHandle?.Dispose();
        }

        // TODO: free unmanaged resources (unmanaged objects) and override a finalizer below.
        // TODO: set large fields to null.

        _disposed = true;

        // Call the base class implementation.
        base.Dispose(disposing);
    }
}
```

```
Imports Microsoft.Win32.SafeHandles
Imports System.IO

Public Class DisposableStreamResource2 : Inherits DisposableStreamResource
    ' Define additional constants.
    Protected Const GENERIC_WRITE As Integer = &H40000000
    Protected Const OPEN_ALWAYS As Integer = 4

    ' Define additional APIs.
    Protected Declare Function WriteFile Lib "kernel32.dll" (
        safeHandle As SafeFileHandle, lpBuffer As String,
        nNumberOfBytesToWrite As Integer, ByRef lpNumberOfBytesWritten As Integer,
        lpOverlapped As Object) As Boolean

    ' Define locals.
    Private disposed As Boolean = False
    Private created As Boolean = False
    Private safeHandle As SafeFileHandle
    Private ReadOnly filename As String

    Public Sub New(filename As String)
        MyBase.New(filename)
        Me.filename = filename
    End Sub

    Public Sub WriteFileInfo()
        If Not created Then
            safeHandle = CreateFile(
                ".\FileInfo.txt", GENERIC_WRITE, 0, IntPtr.Zero,
                OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, IntPtr.Zero)
            created = True
        End If

        Dim output As String = $"{filename }: {Size:N0} bytes {vbCrLf }"
        Dim result = WriteFile(safeHandle, output, output.Length, 0&, Nothing)
    End Sub

    Protected Overridable Overloads Sub Dispose(disposing As Boolean)
        If disposed Then Exit Sub

        ' Release any managed resources here.
        If disposing Then
            safeHandle?.Dispose()
        End If

        disposed = True
        ' Release any unmanaged resources not wrapped by safe handles here.

        ' Call the base class implementation.
        MyBase.Dispose(disposing)
    End Sub
End Class
```

请参阅

- SuppressFinalize
- IDisposable
- IDisposable.Dispose
- Microsoft.Win32.SafeHandles
- System.Runtime.InteropServices.SafeHandle
- Object.Finalize
- 定义和使用类和结构 (C++/CLI)

实现 DisposeAsync 方法

2021/11/16 •

已将 System.IAsyncDisposable 接口作为 C# 8.0 的一部分引入。需要执行资源清理时，可以实现 IAsyncDisposable.DisposeAsync() 方法，就像实现 Dispose 方法一样。但是，其中一个主要区别是，此实现允许异步清理操作。DisposeAsync() 返回表示异步释放操作的 ValueTask。

通常，当实现 IAsyncDisposable 接口时，类还将实现 IDisposable 接口。IAsyncDisposable 接口的一种良好实现模式是为同步或异步释放做好准备。用于实现释放模式的所有指南也适用于异步实现。本文假设你已熟悉如何实现 Dispose 方法。

DisposeAsync() 和 DisposeAsyncCore()

IAsyncDisposable 接口声明单个无参数方法 DisposeAsync()。任何非密封类都应具有另外一个也返回 ValueTask 的 `DisposeAsyncCore()` 方法。

- 没有参数的 `public` IAsyncDisposable.DisposeAsync() 实现。
- 一个 `protected virtual ValueTask DisposeAsyncCore()` 方法，其签名为：

```
protected virtual ValueTask DisposeAsyncCore()
{
}
```

DisposeAsync() 方法

`public` 无参数的 `DisposeAsync()` 方法在 `await using` 语句中隐式调用，其用途是释放非托管资源，执行常规清理，以及指示终结器（如果存在）不必运行。释放与托管对象关联的内存始终是垃圾回收器的域。因此，它具有标准实现：

```
public async ValueTask DisposeAsync()
{
    // Perform async cleanup.
    await DisposeAsyncCore();

    // Dispose of unmanaged resources.
    Dispose(false);
    // Suppress finalization.
    GC.SuppressFinalize(this);
}
```

> **NOTE**
> 与释放模式相比，异步释放模式的主要差异在于，从 DisposeAsync() 到 `Dispose(bool)` 重载方法的调用被赋予 `false` 作为参数。但实现 IDisposable.Dispose() 方法时，改为传递 `true`。这有助于确保与同步释放模式的功能等效性，并进一步确保仍调用终结器代码路径。换句话说，`DisposeAsyncCore()` 方法将异步释放托管资源，因此不希望也同步释放这些资源。因此，调用 `Dispose(false)` 而非 `Dispose(true)`。

DisposeAsyncCore() 方法

`DisposeAsyncCore()` 方法旨在执行受管理资源的异步清理，或对 `DisposeAsync()` 执行级联调用。当子类继承作为 IAsyncDisposable 的实现的基类时，它会封装常见的异步清理操作。`DisposeAsyncCore()` 方法是 `virtual`，以便派生类可以在其重写中定义其他清理。

TIP

如果 IAsyncDisposable 的实现是 `sealed`，则不需要 `DisposeAsyncCore()` 方法，异步清理可直接在 IAsyncDisposable.DisposeAsync() 方法中执行。

实现异步释放模式

所有非密封类都应被视为潜在的基类，因为它们可以被继承。如果为任何潜在基类实现异步释放模式，则必须提供 `protected virtual ValueTask DisposeAsyncCore()` 方法。下面是使用 System.Text.Json.Utf8JsonWriter 的异步释放模式的实现示例。

```
using System;
using System.IO;
using System.Text.Json;
using System.Threading.Tasks;

public class ExampleAsyncDisposable : IAsyncDisposable, IDisposable
{
    private Utf8JsonWriter _jsonWriter = new Utf8JsonWriter(new MemoryStream());

    public void Dispose()
    {
        Dispose(disposing: true);
        GC.SuppressFinalize(this);
    }

    public async ValueTask DisposeAsync()
    {
        await DisposeAsyncCore();

        Dispose(disposing: false);
        GC.SuppressFinalize(this);
    }

    protected virtual void Dispose(bool disposing)
    {
        if (disposing)
        {
            _jsonWriter?.Dispose();
        }

        _jsonWriter = null;
    }

    protected virtual async ValueTask DisposeAsyncCore()
    {
        if (_jsonWriter is not null)
        {
            await _jsonWriter.DisposeAsync().ConfigureAwait(false);
        }

        _jsonWriter = null;
    }
}
```

前面的示例使用 Utf8JsonWriter。有关 `System.Text.Json` 的详细信息，请参阅如何从 Newtonsoft.Json 迁移到 System.Text.Json。

同时实现释放模式和异步释放模式

可能需要同时实现 IDisposable 和 IAsyncDisposable 接口，尤其是当类范围包含这些实现的实例时。这样做可确

保你可以正确地级联清理调用。下面是一个示例类，它实现两个接口并演示清理的正确指导。

```
using System;
using System.IO;
using System.Threading.Tasks;

namespace Samples
{
    public class CustomDisposable : IDisposable, IAsyncDisposable
    {
        IDisposable _disposableResource = new MemoryStream();
        IAsyncDisposable _asyncDisposableResource = new MemoryStream();

        public void Dispose()
        {
            Dispose(disposing: true);
            GC.SuppressFinalize(this);
        }

        public async ValueTask DisposeAsync()
        {
            await DisposeAsyncCore();

            Dispose(disposing: false);
            GC.SuppressFinalize(this);
        }

        protected virtual void Dispose(bool disposing)
        {
            if (disposing)
            {
                _disposableResource?.Dispose();
                (_asyncDisposableResource as IDisposable)?.Dispose();
            }

            _disposableResource = null;
            _asyncDisposableResource = null;
        }

        protected virtual async ValueTask DisposeAsyncCore()
        {
            if (_asyncDisposableResource is not null)
            {
                await _asyncDisposableResource.DisposeAsync().ConfigureAwait(false);
            }

            if (_disposableResource is IAsyncDisposable disposable)
            {
                await disposable.DisposeAsync().ConfigureAwait(false);
            }
            else
            {
                _disposableResource.Dispose();
            }

            _asyncDisposableResource = null;
            _disposableResource = null;
        }
    }
}
```

IDisposable.Dispose() 和 IAsyncDisposable.DisposeAsync() 实现都是简单的样板代码。

在 `Dispose(bool)` 重载方法中，如果 IDisposable 实例不为 `null`，则有条件地将其释放。IAsyncDisposable 实例被强制转换为 IDisposable，如果该实例也不为 `null`，也将被释放。然后，将这两个实例都分配给 `null`。

使用 `DisposeAsyncCore()` 方法时，遵循相同的逻辑方法。如果 IAsyncDisposable 实例不为 `null`，则等待其对 `DisposeAsync().ConfigureAwait(false)` 的调用。如果 IDisposable 实例也是 IAsyncDisposable 的实现，也将其异步释放。然后，将这两个实例都分配给 `null`。

使用异步释放

要正确使用实现 IAsyncDisposable 接口的对象，请将 await和 using 关键字结合使用。请考虑以下示例，其中 `ExampleAsyncDisposable` 类进行了实例化，然后包装在 `await using` 语句中。

```
using System;
using System.Threading.Tasks;

class ExampleConfigureAwaitProgram
{
    static async Task Main()
    {
        var exampleAsyncDisposable = new ExampleAsyncDisposable();
        await using (exampleAsyncDisposable.ConfigureAwait(false))
        {
            // Interact with the exampleAsyncDisposable instance.
        }

        Console.ReadLine();
    }
}
```

> **IMPORTANT**
> 使用 IAsyncDisposable 接口的 ConfigureAwait(IAsyncDisposable, Boolean) 扩展方法配置延续任务在其原始上下文或计划程序上的封送方式。有关 `ConfigureAwait` 的详细信息，请参阅 ConfigureAwait FAQ。

对于不需要使用 `ConfigureAwait` 的情况，可以按如下所示简化 `await using` 语句：

```
using System;
using System.Threading.Tasks;

class ExampleProgram
{
    static async Task Main()
    {
        await using (var exampleAsyncDisposable = new ExampleAsyncDisposable())
        {
            // Interact with the exampleAsyncDisposable instance.
        }

        Console.ReadLine();
    }
}
```

此外，它还可以编写为使用 using 声明的隐式范围。

```
using System;
using System.Threading.Tasks;

class ExampleUsingDeclarationProgram
{
    static async Task Main()
    {
        await using var exampleAsyncDisposable = new ExampleAsyncDisposable();

        // Interact with the exampleAsyncDisposable instance.

        Console.ReadLine();
    }
}
```

堆叠的 using

在创建和使用实现 IAsyncDisposable 的多个对象的情况下，残存错误条件中具有 ConfigureAwait 的堆叠 `await using` 语句可能会阻止调用 DisposeAsync()。若要确保始终调用 DisposeAsync()，应避免堆叠。下面的三个代码示例显示要改用的可接受模式。

可接受的模式一

```
using System;
using System.Threading.Tasks;

class ExampleOneProgram
{
    static async Task Main()
    {
        var objOne = new ExampleAsyncDisposable();
        await using (objOne.ConfigureAwait(false))
        {
            // Interact with the objOne instance.

            var objTwo = new ExampleAsyncDisposable();
            await using (objTwo.ConfigureAwait(false))
            {
                // Interact with the objOne and/or objTwo instance(s).
            }
        }

        Console.ReadLine();
    }
}
```

在前面的示例中，每个异步清理操作的范围都显式地限定在 `await using` 块下。外部范围由 `objOne` 设置其大括号的方法来定义；若将 `objTwo` 括起来，这样就会先处理 `objTwo`，然后处理 `objOne`。这两个 `IAsyncDisposable` 实例都使三种 DisposeAsync() 方法等待，从而执行其异步清理操作。嵌套调用，而不是堆叠调用。

可接受的模式二

```
class ExampleTwoProgram
{
    static async Task Main()
    {
        var objOne = new ExampleAsyncDisposable();
        await using (objOne.ConfigureAwait(false))
        {
            // Interact with the objOne instance.
        }

        var objTwo = new ExampleAsyncDisposable();
        await using (objTwo.ConfigureAwait(false))
        {
            // Interact with the objTwo instance.
        }

        Console.ReadLine();
    }
}
```

在前面的示例中，每个异步清理操作的范围都显式地限定在 `await using` 块下。在每个块的末尾，相应的 `IAsyncDisposable` 实例使其 DisposeAsync() 方法等待，从而执行其异步清理操作。按顺序排列调用，而不是堆叠调用。在此场景中，首先处理 `objOne`，然后处理 `objTwo`。

可接受的模式三

```
class ExampleThreeProgram
{
    static async Task Main()
    {
        var objOne = new ExampleAsyncDisposable();
        await using var ignored1 = objOne.ConfigureAwait(false);

        var objTwo = new ExampleAsyncDisposable();
        await using var ignored2 = objTwo.ConfigureAwait(false);

        // Interact with objOne and/or objTwo instance(s).

        Console.ReadLine();
    }
}
```

在前面的示例中，每个异步清理操作都通过包含的方法主体隐式限定了范围。在封闭块的末尾，`IAsyncDisposable` 实例执行其异步清理操作。此运行顺序与它们声明的顺序相反，这意味着 `objTwo` 在 `objOne` 之前被处理。

无法接受的模式

如果从 `AnotherAsyncDisposable` 构造函数引发异常，则 `objOne` 不会得到正确处理：

```
class DoNotDoThisProgram
{
    static async Task Main()
    {
        var objOne = new ExampleAsyncDisposable();
        // Exception thrown on .ctor
        var objTwo = new AnotherAsyncDisposable();

        await using (objOne.ConfigureAwait(false))
        await using (objTwo.ConfigureAwait(false))
        {
            // Neither object has its DisposeAsync called.
        }

        Console.ReadLine();
    }
}
```

TIP

避免此模式，因为它可能导致意外行为。

请参阅

有关 `IDisposable` 和 `IAsyncDisposable` 的双重实现示例，请参阅 GitHub 上的 Utf8JsonWriter 源代码。

- IAsyncDisposable
- IAsyncDisposable.DisposeAsync()
- ConfigureAwait(IAsyncDisposable, Boolean)

使用实现 IDisposable 的对象

2021/11/16 •

公共语言运行时的垃圾回收器回收托管对象使用的内存，而使用非托管资源的类型则实现 IDisposable 接口，以允许回收这些非托管资源所需的内存。在使用完实现 IDisposable 的对象后，应调用该对象的 IDisposable.Dispose 实现。可以通过两种方法执行此操作：

- 使用 C# `using` 语句（Visual Basic 中为 `Using`）。
- 通过实现 `try/finally` 块，并调用 `finally` 中的 IDisposable.Dispose。

using 语句

C# 中的 `using` 语句和 Visual Basic 中的 `Using` 语句可以简化为清理对象而必须编写的代码。`using` 语句获取一个或多个资源，执行指定的语句，并自动处置对象。但是，`using` 语句仅对在用于构造对象的方法的范围内使用的对象有用。

以下示例使用 `using` 语句创建并发布 System.IO.StreamReader 对象。

```
using System;
using System.IO;

class Example
{
   static void Main()
   {
      char[] buffer = new char[50];
      using var streamReader = new StreamReader("file1.txt");

      int charsRead = 0;
      while (streamReader.Peek() != -1)
      {
         charsRead = streamReader.Read(buffer, 0, buffer.Length);
         //
         // Process characters read.
         //
      }
   }
}
```

```
Imports System.IO

Module Example
    Public Sub Main()
        Dim buffer(49) As Char
        Using s As New StreamReader("File1.txt")
            Dim charsRead As Integer
            Do While s.Peek() <> -1
                charsRead = s.Read(buffer, 0, buffer.Length)
                '
                ' Process characters read.
                '
            Loop
        End Using
    End Sub
End Module
```

虽然 StreamReader 类实现了 IDisposable 接口(这说明它使用的是非托管资源)，但此示例不显式调用 StreamReader.Dispose 方法。当 C# 或 Visual Basic 编译器遇到 `using` 语句时，它会发出与以下显式包含 `try/finally` 块的代码等效的中间语言 (IL)。

```
using System;
using System.IO;

class Example
{
    static void Main()
    {
        char[] buffer = new char[50];
        var streamReader = new StreamReader("file1.txt");
        try
        {
            int charsRead = 0;
            while (streamReader.Peek() != -1)
            {
                charsRead = streamReader.Read(buffer, 0, buffer.Length);
                //
                // Process characters read.
                //
            }
        }
        finally
        {
            if (streamReader != null)
            {
                ((IDisposable)streamReader).Dispose();
            }
        }
    }
}
```

```
Imports System.IO

Module Example
    Public Sub Main()
        Dim buffer(49) As Char
        ''      Dim s As New StreamReader("File1.txt")
        With s As New StreamReader("File1.txt")
      Try
                Dim charsRead As Integer
                Do While s.Peek() <> -1
                    charsRead = s.Read(buffer, 0, buffer.Length)
                    '
                    ' Process characters read.
                    '
                Loop
            Finally
                If s IsNot Nothing Then DirectCast(s, IDisposable).Dispose()
            End Try
        End With
    End Sub
End Module
```

使用 C# `using` 语句，还可以在一个语句(在内部相当于嵌套语句 `using`)中获取多个资源。下面的示例实例化两个 StreamReader 对象以读取两个不同文件的内容。

```
using System.IO;

class Example
{
    static void Main()
    {
        char[] buffer1 = new char[50];
        char[] buffer2 = new char[50];

        using StreamReader version1 = new StreamReader("file1.txt"),
                           version2 = new StreamReader("file2.txt");

        int charsRead1, charsRead2 = 0;
        while (version1.Peek() != -1 && version2.Peek() != -1)
        {
            charsRead1 = version1.Read(buffer1, 0, buffer1.Length);
            charsRead2 = version2.Read(buffer2, 0, buffer2.Length);
            //
            // Process characters read.
            //
        }
    }
}
```

Try/finally 块

可以选择直接实现 `try/finally` 块，而不是将 `try/finally` 块包装在 `using` 语句中。它可以是私有编码样式，或者你可能出于下列原因之一需要这样做：

- 包含 `catch` 块以处理 `try` 块中引发的异常。否则，不会处理 `using` 语句中引发的任何异常。
- 实例化实现 IDisposable（其范围对于声明它的块是非本地的）的对象。

下面的示例与上一示例类似，不同之处在于此示例使用 `try/catch/finally` 块实例化、使用和清理 StreamReader 对象，同时处理 StreamReader 构造函数及其 ReadToEnd 方法抛出的任何异常。`finally` 块中的代码检查实现 IDisposable 的对象在其调用 Dispose 方法之前不为 `null`。此操作失败会导致运行时发生 NullReferenceException 异常。

```
using System;
using System.Globalization;
using System.IO;

class Example
{
    static void Main()
    {
        StreamReader? streamReader = null;
        try
        {
            streamReader = new StreamReader("file1.txt");
            string contents = streamReader.ReadToEnd();
            var info = new StringInfo(contents);
            Console.WriteLine($"The file has {info.LengthInTextElements} text elements.");
        }
        catch (FileNotFoundException)
        {
            Console.WriteLine("The file cannot be found.");
        }
        catch (IOException)
        {
            Console.WriteLine("An I/O error has occurred.");
        }
        catch (OutOfMemoryException)
        {
            Console.WriteLine("There is insufficient memory to read the file.");
        }
        finally
        {
            streamReader?.Dispose();
        }
    }
}
```

```
Imports System.Globalization
Imports System.IO

Module Example
    Sub Main()
        Dim streamReader As StreamReader = Nothing
        Try
            streamReader = New StreamReader("file1.txt")
            Dim contents As String = streamReader.ReadToEnd()
            Dim info As StringInfo = New StringInfo(contents)
            Console.WriteLine($"The file has {info.LengthInTextElements} text elements.")
        Catch e As FileNotFoundException
            Console.WriteLine("The file cannot be found.")
        Catch e As IOException
            Console.WriteLine("An I/O error has occurred.")
        Catch e As OutOfMemoryException
            Console.WriteLine("There is insufficient memory to read the file.")
        Finally
            If streamReader IsNot Nothing Then streamReader.Dispose()
        End Try
    End Sub
End Module
```

如果选择实现或必须实现 `try/finally` 块，可以遵循此基本模式，因为编程语言不支持 `using` 语句，但允许直接调用 Dispose 方法。

IDisposable 实例成员

如果类将 IDisposable 实现保存为实例成员(字段或属性)，类还应实现 IDisposable。有关详细信息，请参阅实现级联释放。

请参阅

- 清理未托管资源
- using 语句(C# 参考)
- Using 语句 (Visual Basic)

垃圾回收

2021/11/16 •

.NET 的垃圾回收器管理应用程序的内存分配和释放。每当有对象新建时，公共语言运行时都会从托管堆为对象分配内存。只要托管堆中有地址空间，运行时就会继续为新对象分配空间。不过，内存并不是无限的。垃圾回收器最终必须执行垃圾回收来释放一些内存。垃圾回收器的优化引擎会根据所执行的分配来确定执行回收的最佳时机。执行回收时，垃圾回收器会在托管堆中检查应用程序不再使用的对象，然后执行必要的操作来回收其内存。

本节内容

TITLE	ıı
垃圾回收的基本知识	描述垃圾回收的工作原理、如何在托管堆上分配对象，以及其他核心概念。
工作站和服务器垃圾回收	描述了客户端应用的工作站垃圾回收与服务器应用的服务器垃圾回收之间的区别。
后台垃圾回收	描述了后台垃圾回收，它是在进行第二代回收时对第 0 代和第 1 代对象的回收。
大型对象堆	描述了大型对象堆 (LOH) 及其垃圾回收方式。
垃圾回收和性能	介绍了可用来诊断垃圾回收和性能问题的性能检查。
已引发回收	描述如何完成垃圾回收。
延迟模式	描述确定垃圾回收侵入性的模式。
针对共享 Web 承载优化	介绍了如何在多个小网站共用的服务器上优化垃圾回收。
垃圾回收通知	介绍了如何确定全面垃圾回收的开始时间和结束时间。
应用程序域资源监视	介绍了如何监视应用程序域的 CPU 和内存使用情况。
弱引用	描述允许应用程序访问对象的同时也允许垃圾回收器收集相应对象的功能。

参考

- System.GC
- System.GCCollectionMode
- System.GCNotificationStatus
- System.Runtime.GCLatencyMode
- System.Runtime.GCSettings
- GCSettings.LargeObjectHeapCompactionMode
- Object.Finalize

- System.IDisposable

请参阅

- **清理非托管资源**

垃圾回收的基本知识

2021/11/16 •

在公共语言运行时 (CLR) 中，垃圾回收器 (GC) 用作自动内存管理器。垃圾回收器管理应用程序的内存分配和释放。对于使用托管代码的开发人员而言，这就意味着不必编写执行内存管理任务的代码。自动内存管理可解决常见问题，例如，忘记释放对象并导致内存泄漏，或尝试访问已释放对象的内存。

本文章介绍垃圾回收的核心概念。

优点

垃圾回收器具有以下优点：

- 开发人员不必手动释放内存。
- 有效分配托管堆上的对象。
- 回收不再使用的对象，清除它们的内存，并保留内存以用于将来分配。托管对象会自动获取干净的内容来开始，因此，它们的构造函数不必对每个数据字段进行初始化。
- 通过确保对象不能使用另一个对象的内容来提供内存安全。

内存基础知识

下面的列表总结了重要的 CLR 内存概念。

- 每个进程都有其自己单独的虚拟地址空间。同一台计算机上的所有进程共享相同的物理内存和页文件（如果有）。
- 默认情况下，32 位计算机上的每个进程都具有 2 GB 的用户模式虚拟地址空间。
- 作为一名应用程序开发人员，你只能使用虚拟地址空间，请勿直接操控物理内存。垃圾回收器为你分配和释放托管堆上的虚拟内存。

  如果你编写的是本机代码，请使用 Windows 函数处理虚拟地址空间。这些函数为你分配和释放本机堆上的虚拟内存。
- 虚拟内存有三种状态：

"	"
Free	该内存块没有引用关系，可用于分配。
保留	内存块可供你使用，并且不能用于任何其他分配请求。但是，在该内存块提交之前，你无法将数据存储到其中。
已提交	内存块已指派给物理存储。

- 可能会存在虚拟地址空间碎片。就是说地址空间中存在一些被称为孔的可用块。当请求虚拟内存分配时，虚拟内存管理器必须找到满足该分配请求的足够大的单个可用块。即使有 2 GB 可用空间，2 GB 分配请求也会失败，除非所有这些可用空间都位于一个地址块中。
- 如果没有足够的可供保留的虚拟地址空间或可供提交的物理空间，则可能会用尽内存。

  即使在物理内存压力（即物理内存的需求）较低的情况下也会使用页文件。首次出现物理内存压力较高的

情况时，操作系统必须在物理内存中腾出空间来存储数据，并将物理内存中的部分数据备份到页文件中。该数据只会在需要时进行分页，所以在物理内存压力较低的情况下也可能会进行分页。

内存分配

初始化新进程时，运行时会为进程保留一个连续的地址空间区域。这个保留的地址空间被称为托管堆。托管堆维护着一个指针，用它指向将在堆中分配的下一个对象的地址。最初，该指针设置为指向托管堆的基址。托管堆上部署了所有引用类型。应用程序创建第一个引用类型时，将为托管堆的基址中的类型分配内存。应用程序创建下一个对象时，垃圾回收器在紧接第一个对象后面的地址空间内为它分配内存。只要地址空间可用，垃圾回收器就会继续以这种方式为新对象分配空间。

从托管堆中分配内存要比非托管内存分配速度快。由于运行时通过为指针添加值来为对象分配内存，所以这几乎和从堆栈中分配内存一样快。另外，由于连续分配的新对象在托管堆中是连续存储，所以应用程序可以快速访问这些对象。

内存释放

垃圾回收器的优化引擎根据所执行的分配决定执行回收的最佳时间。垃圾回收器在执行回收时，会释放应用程序不再使用的对象的内存。它通过检查应用程序的根来确定不再使用的对象。应用程序的根包含线程堆栈上的静态字段、局部变量、CPU 寄存器、GC 句柄和终结队列。每个根或者引用托管堆中的对象，或者设置为空。垃圾回收器可以为这些根请求其余运行时。垃圾回收器使用此列表创建一个图表，其中包含所有可从这些根中访问的对象。

不在该图表中的对象将无法从应用程序的根中访问。垃圾回收器会考虑无法访问的对象垃圾，并释放为它们分配的内存。在回收中，垃圾回收器检查托管堆，查找无法访问对象所占据的地址空间块。发现无法访问的对象时，它就使用内存复制功能来压缩内存中可以访问的对象，释放分配给不可访问对象的地址空间块。在压缩了可访问对象的内存后，垃圾回收器就会做出必要的指针更正，以便应用程序的根指向新地址中的对象。它还将托管堆指针定位至最后一个可访问对象之后。

只有在回收发现大量的无法访问的对象时，才会压缩内存。如果托管堆中的所有对象均未被回收，则不需要压缩内存。

为了改进性能，运行时为单独堆中的大型对象分配内存。垃圾回收器会自动释放大型对象的内存。但是，为了避免移动内存中的大型对象，通常不会压缩此内存。

垃圾回收的条件

当满足以下条件之一时将发生垃圾回收：

- 系统具有低的物理内存。这是通过 OS 的内存不足通知或主机指示的内存不足检测出来。
- 由托管堆上已分配的对象使用的内存超出了可接受的阈值。随着进程的运行，此阈值会不断地进行调整。
- 调用 GC.Collect 方法。几乎在所有情况下，你都不必调用此方法，因为垃圾回收器会持续运行。此方法主要用于特殊情况和测试。

托管堆

在垃圾回收器由 CLR 初始化之后，它会分配一段内存用于存储和管理对象。此内存称为托管堆（与操作系统中的本机堆相对）。

每个托管进程都有一个托管堆。进程中的所有线程都在同一堆上为对象分配内存。

若要保留内存，垃圾回收器会调用 Windows VirtualAlloc 函数，并且每次为托管应用保留一个内存段。垃圾回收器还会根据需要保留内存段，并调用 Windows VirtualFree 函数，将内存段释放回操作系统（在清除所有对象的内存段后）。

> **IMPORTANT**
> 垃圾回收器分配的段大小特定于实现，并且随时可能更改（包括定期更新）。应用程序不应假设特定段的大小或依赖于此大小，也不应尝试配置段分配可用的内存量。

堆上分配的对象越少，垃圾回收器必须执行的工作就越少。分配对象时，请勿使用超出你需求的舍入值，例如在仅需要 15 个字节的情况下分配了 32 个字节的数组。

当触发垃圾回收时，垃圾回收器将回收由非活动对象占用的内存。回收进程会对活动对象进行压缩，以便将它们一起移动，并移除死空间，从而使堆更小一些。这将确保一起分配的对象全都位于托管堆上，从而保留它们的局部性。

垃圾回收的侵入性（频率和持续时间）是由分配的数量和托管堆上保留的内存数量决定的。

此堆可视为两个堆的累计：大对象堆和小对象堆。大对象堆包含大小不少于 85,000 个字节的对象，这些对象通常是数组。非常大的实例对象是很少见的。

> **TIP**
> 可以配置阈值大小，以使对象能够进入大型对象堆。

代数

GC 算法基于几个注意事项：

- 压缩托管堆的一部分内存要比压缩整个托管堆速度快。
- 较新的对象生存期较短，而较旧的对象生存期则较长。
- 较新的对象趋向于相互关联，并且大致同时由应用程序访问。

垃圾回收主要在回收短生存期对象时发生。为优化垃圾回收器的性能，将托管堆分为三代：第 0 代、第 1 代和第 2 代，因此它可以单独处理长生存期和短生存期对象。垃圾回收器将新对象存储在第 0 代中。在应用程序生存期的早期创建的对象如果未被回收，则被升级并存储在第 1 级和第 2 级中。因为压缩托管堆的一部分要比压缩整个托管堆速度快，所以此方案允许垃圾回收器在每次执行回收时释放特定级别的内存，而不是整个托管堆的内存。

- **第 0 代**。这是最年轻的代，其中包含短生存期对象。短生存期对象的一个示例是临时变量。垃圾回收最常发生在此代中。

  新分配的对象构成新一代对象，并隐式地成为第 0 代集合。但是，如果它们是大型对象，它们将延续到大型对象堆 (LOH)，这有时称为第 3 代。第 3 代是在第 2 代中逻辑收集的物理生成。

  大多数对象通过第 0 代中的垃圾回收进行回收，不会保留到下一代。

  如果应用程序在第 0 代托管堆已满时尝试创建新对象，垃圾回收器将执行收集，以尝试为该对象释放地址空间。垃圾回收器从检查第 0 级托管堆中的对象（而不是托管堆中的所有对象）开始执行回收。单独回收第 0 代托管堆通常可以回收足够的内存，这样，应用程序便可以继续创建新对象。

- **第 1 代**。这一代包含短生存期对象并用作短生存期对象和长生存期对象之间的缓冲区。

  垃圾回收器执行第 0 代托管堆的回收后，会压缩可访问对象的内存，并将其升级到第 1 代。因为未被回收的对象往往具有较长的生存期，所以将它们升级至更高的级别很有意义。垃圾回收器不必在每次执行第 0 代托管堆的回收时，都重新检查第 1 代和第 2 代托管堆中的对象。

  如果第 0 代托管堆的回收没有回收足够的内存供应用程序创建新对象，垃圾回收器就会先执行第 1 代托管堆的回收，然后再执行第 2 代托管堆的回收。第 1 级托管堆中未被回收的对象将会升级至第 2 级托管堆。

- **第 2 代**。这一代包含长生存期对象。长生存期对象的一个示例是服务器应用程序中的一个包含在进程期间处于活动状态的静态数据的对象。

  第 2 代托管堆中未被回收的对象会继续保留在第 2 代托管堆中，直到在将来的回收中确定它们无法访问为止。

  大型对象堆上的对象（有时称为 *第 3 代*）也在第 2 代中收集。

当条件得到满足时，垃圾回收将在特定代上发生。回收某个代意味着回收此代中的对象及其所有更年轻的代。第 2 代垃圾回收也称为完整垃圾回收，因为它回收所有代中的对象（即，托管堆中的所有对象）。

幸存和提升

垃圾回收中未回收的对象也称为幸存者，并会被提升到下一代：

- 第 0 代垃圾回收中未被回收的对象将会升级至第 1 代。
- 第 1 代垃圾回收中未被回收的对象将会升级至第 2 代。
- 第 2 代垃圾回收中未被回收的对象将仍保留在第 2 代。

当垃圾回收器检测到某个代中的幸存率很高时，它会增加该代的分配阈值。下次回收将回收非常大的内存。CLR 持续在以下两个优先级之间进行平衡：不允许通过延迟垃圾回收，让应用程序的工作集获取太大内存，以及不允许垃圾回收过于频繁地运行。

暂时代和暂时段

因为第 0 代和第 1 代中的对象的生存期较短，因此，这些代被称为“暂时代”。

暂时代在称为“暂时段”的内存段中进行分配。垃圾回收器获取的每个新段将成为新的暂时段，并包含在第 0 代垃圾回收中幸存的对象。旧的暂时段将成为新的第 2 代段。

根据系统为 32 位还是 64 位以及它正在哪种类型的垃圾回收器（工作站或服务器 GC）上运行，暂时段的大小发生相应变化。下表显示了暂时段的默认大小。

[illegible]/[illegible] GC	32 [illegible]	64 [illegible]
工作站 GC	16 MB	256 MB
服务器 GC	64 MB	4 GB
服务器 GC（具有 4 个以上的逻辑 CPU）	32 MB	2 GB
服务器 GC（具有 8 个以上的逻辑 CPU）	16 MB	1 GB

暂时段可以包含第 2 代对象。第 2 代对象可使用多个段（在内存允许的情况下进程所需的任意数量）。

从暂时垃圾回收中释放的内存量限制为暂时段的大小。释放的内存量与死对象占用的空间成比例。

垃圾回收过程中发生的情况

垃圾回收分为以下几个阶段：

- 标记阶段，找到并创建所有活动对象的列表。
- 重定位阶段，用于更新对将要压缩的对象的引用。
- 压缩阶段，用于回收由死对象占用的空间，并压缩幸存的对象。压缩阶段将垃圾回收中幸存下来的对象移至段中时间较早的一端。

  因为第 2 代回收可以占用多个段，所以可以将已提升到第 2 代中的对象移动到时间较早的段中。可以将第 1 代幸存者和第 2 代幸存者都移动到不同的段，因为它们已被提升到第 2 代。

通常，由于复制大型对象会造成性能代偿，因此不会压缩大型对象堆 (LOH)。但是，在 .NET Core 和 .NET Framework 4.5.1 及更高版本中，可以根据需要使用 GCSettings.LargeObjectHeapCompactionMode 属性按需压缩大型对象堆。此外，当通过指定以下任一项设置硬限制时，将自动压缩 LOH：

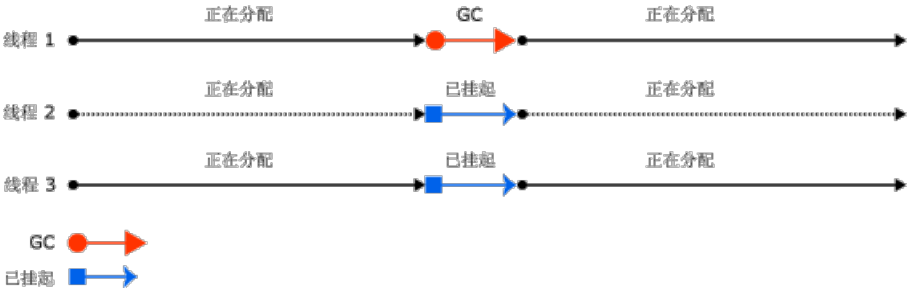
- 针对容器的内存限制。
- GCHeapHardLimit 或 GCHeapHardLimitPercent 运行时配置选项。

垃圾回收器使用以下信息来确定对象是否为活动对象：

- **堆栈根**。由实时 (JIT) 编译器和堆栈查看器提供的堆栈变量。JIT 优化可以延长或缩短报告给垃圾回收器的堆栈变量内的代码的区域。
- **垃圾回收句柄**。指向托管对象且可由用户代码或公共语言运行时分配的句柄。
- **静态数据**。应用程序域中可能引用其他对象的静态对象。每个应用程序域都会跟踪其静态对象。

在垃圾回收启动之前，除了触发垃圾回收的线程以外的所有托管线程均会挂起。

下图演示了触发垃圾回收并导致其他线程挂起的线程。



非托管资源

对于应用程序创建的大多数对象，可以依赖垃圾回收自动执行必要的内存管理任务。但是，非托管资源需要显式清除。最常用的非托管资源类型是包装操作系统资源的对象，例如，文件句柄、窗口句柄或网络连接。虽然垃圾回收器可以跟踪封装非托管资源的托管对象的生存期，但却无法具体了解如何清理资源。

创建封装非托管资源的对象时，建议在公共 `Dispose` 方法中提供必要的代码以清理非托管资源。通过提供 `Dispose` 方法，对象的用户可以在使用完对象后显式释放其内存。使用封装非托管资源的对象时，务必要在需要时调用 `Dispose`。

还必须提供一种释放非托管资源的方法，以防类型使用者忘记调用 `Dispose`。可以使用安全句柄来包装非托管资源，也可以重写 Object.Finalize() 方法。

有关清理非托管资源的详细信息，请参阅清理非托管资源。

请参阅

- 工作站和服务器垃圾回收
- 后台垃圾回收
- GC 的配置选项
- 垃圾回收

工作站和服务器垃圾回收

2021/11/16 •

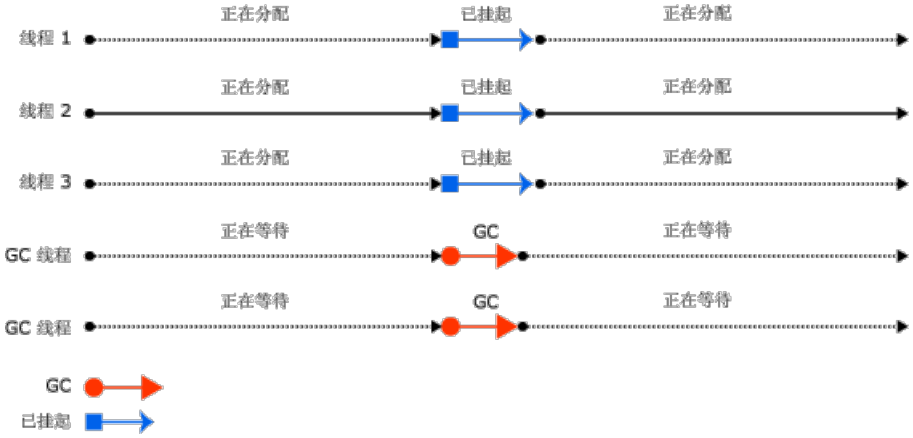
垃圾回收器可自行优化并且适用于多种方案。不过，你可以基于工作负载的特征设置垃圾回收的类型。CLR 提供了以下类型的垃圾回收：

- 工作站垃圾回收 (GC) 是为客户端应用设计的。它是独立应用的默认 GC 风格。对于托管应用（例如由 ASP.NET 托管的应用），由主机确定默认 GC 风格。

  工作站垃圾回收既可以是并发的，也可以是非并发的。并发（或后台）垃圾回收使托管线程能够在垃圾回收期间继续操作。后台垃圾回收替换 .NET Framework 4 及更高版本中的并行垃圾回收。

- 服务器垃圾回收，用于需要高吞吐量和可伸缩性的服务器应用程序。
  - 在 .NET Core 中，服务器垃圾回收既可以是非并发也可以是后台执行。
  - 在 .NET Framework 4.5 和更高版本中，服务器垃圾回收既可以是非并发也可以是后台执行。在 .NET Framework 4 和以前的版本中，服务器垃圾回收非并行运行。

下图演示了服务器上执行垃圾回收的专用线程：



性能注意事项

工作站 GC

以下是工作站垃圾回收的线程处理和性能注意事项：

- 回收发生在触发垃圾回收的用户线程上，并保留相同优先级。因为用户线程通常以普通优先级运行，所以垃圾回收器（在普通优先级线程上运行）必须与其他线程竞争 CPU 时间。（运行本机代码的线程不会由于服务器或工作站垃圾回收而挂起。）
- 工作站垃圾回收始终用于只有一个处理器的计算机，无论配置设置如何。

服务器 GC

以下是服务器垃圾回收的线程处理和性能注意事项：

- 回收发生在以 `THREAD_PRIORITY_HIGHEST` 优先级运行的多个专用线程上。
- 为每个 CPU 提供一个用于执行垃圾回收的一个堆和专用线程，并将同时回收这些堆。每个堆都包含一个小对象堆和一个大对象堆，并且所有的堆都可由用户代码访问。不同堆上的对象可以相互引用。
- 因为多个垃圾回收线程一起工作，所以对于相同大小的堆，服务器垃圾回收比工作站垃圾回收更快一些。

- 服务器垃圾回收通常具有更大的段。但是，这是通常情况：段大小特定于实现且可能更改。调整应用程序时，不要假设垃圾回收器分配的段大小。
- 服务器垃圾回收会占用大量资源。例如，假设在一台有 4 个处理器的计算机上，运行着 12 个使用服务器 GC 的进程。如果所有进程碰巧同时回收垃圾，它们会相互干扰，因为将在同一个处理器上调度 12 个线程。如果进程处于活动状态，则最好不要让它们都使用服务器 GC。

如果运行应用程序的数百个实例，请考虑使用工作站垃圾回收并禁用并发垃圾回收。这可以减少上下文切换，从而提高性能。

请参阅

- 后台垃圾回收
- 用于垃圾回收的运行时配置选项

后台垃圾回收

2021/11/16 •

在后台垃圾回收 (GC) 中，在进行第 2 代回收的过程中，将会根据需要收集暂时代（第 0 代和第 1 代）。后台垃圾回收是在一个或多个专用线程上执行的，具体取决于它是后台还是服务器 GC，它只适用于第 2 代回收。

默认启用后台垃圾回收。可以在 .NET Framework 应用中使用 gcConcurrent 配置设置或 .NET Core 和 .NET 5 及更高版本应用中的 System.GC.Concurrent 来启用或禁用后台垃圾回收。

> **NOTE**
> 后台垃圾回收替换在 .NET Framework 4 及更高版本中可用的并行垃圾回收。在 .NET Framework 4 中，仅支持工作站垃圾回收。从 .NET Framework 4.5 开始，后台垃圾回收可用于工作站和服务器垃圾回收 。

后台垃圾回收期间对暂时代的回收称为“前台”垃圾回收。发生前台垃圾回收时，所有托管线程都将被挂起。

当后台垃圾回收正在进行并且你已在第 0 代中分配了足够的对象时，CLR 将执行第 0 代或第 1 代前台垃圾回收。专用的后台垃圾回收线程将在常见的安全点上进行检查以确定是否存在对前台垃圾回收的请求。如果存在，则后台回收将挂起自身以便前台垃圾回收可以发生。在前台垃圾回收完成之后，专用的后台垃圾回收线程和用户线程将继续。

后台垃圾回收可以消除并发垃圾回收所带来的分配限制，因为在后台垃圾回收期间，可发生暂时垃圾回收。后台垃圾回收可以删除暂存世代中的死对象。如果需要，它还可以在第 1 代垃圾回收期间扩展堆。

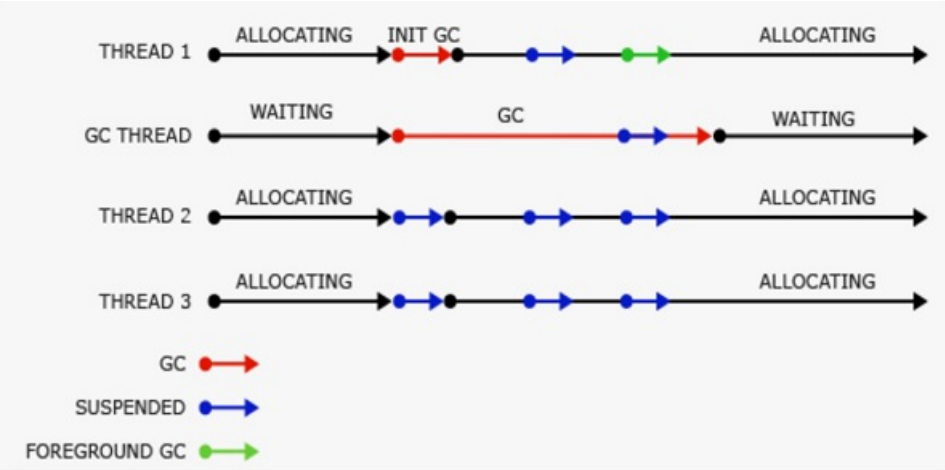
后台工作站与服务器 GC

从 .NET Framework 4.5 开始，后台垃圾回收可用于服务器 GC。服务器 GC 是服务器垃圾回收的默认模式。

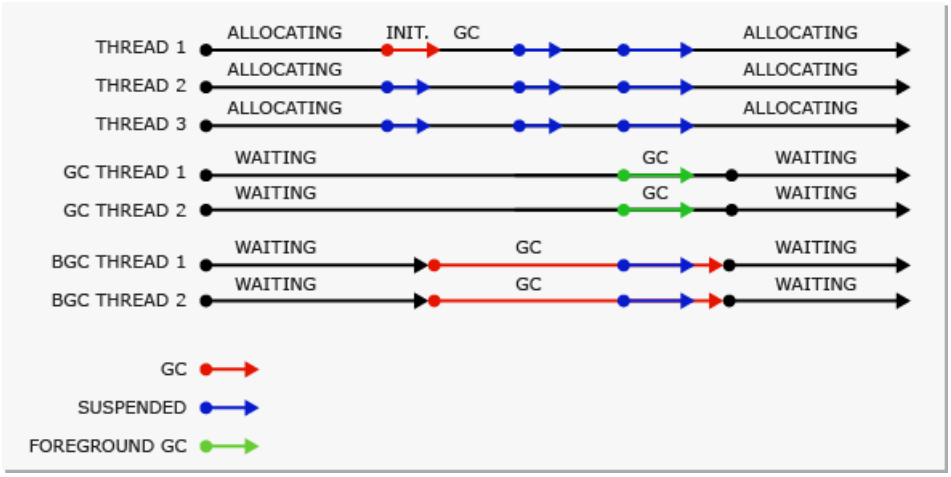
后台服务器垃圾回收与后台工作站垃圾回收具有类似功能，但有一些不同之处：

- 后台工作区域垃圾回收使用一个专用的后台垃圾回收线程，而后台服务器垃圾回收使用多个线程。通常一个逻辑处理器有一个专用线程。
- 不同于工作站后台垃圾回收线程，这些后台服务器 GC 线程不会超时。

下图显示对独立专用线程执行的后台工作站垃圾回收：



下图显示对独立专用线程执行的后台服务器垃圾回收：



并行垃圾回收

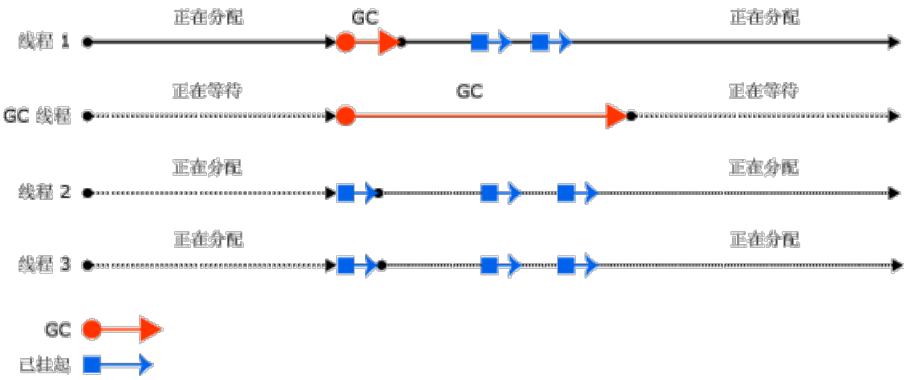
> **TIP**
> 本部分仅适用于：
>
> - 用于工作站垃圾回收的 .NET Framework 3.5 及更早版本
> - 用于服务器垃圾回收的 .NET Framework 4 及更早版本
>
> 在更高的版本中，后台垃圾回收取代了并行垃圾回收。

在工作站或服务器垃圾回收中，你可以启用并发垃圾回收，以便在大多数回收期间，让各线程与执行垃圾回收的专用线程并发运行。此选项只影响第 2 代中的垃圾回收；第 0 代和第 1 代中的垃圾回收始终是非并发的，因为它们完成的速度很快。

并发垃圾回收通过最大程度地减少因回收引起的暂停，使交互应用程序能够更快地响应。在运行并发垃圾回收线程的大多数时间，托管线程可以继续运行。此设计使得在发生垃圾回收时的暂停时间更短。

并发垃圾回收在一个专用线程上执行。默认情况下，CLR 将运行工作站垃圾回收，并在单处理器和多处理器计算机上同时启用并发垃圾回收。

下图演示了在单独的专用线程上执行的并发垃圾回收。



请参阅

- 工作站和服务器垃圾回收
- 用于垃圾回收的运行时配置选项

Windows 系统上的大型对象堆

2021/11/16 •

.NET 垃圾回收器 (GC) 将对象分为小型和大型对象。如果是大型对象，它的某些特性将比对象较小时显得更为重要。例如，压缩大型对象—(也就是在内存中将其复制到堆上的其他位置)—的费用相当高。因此，垃圾回收器将大型对象放置在大型对象堆 (LOH) 上。本文将讨论符合什么条件的对象才能称之为大型对象，如何回收大型对象，以及大型对象具备哪些性能意义。

> **IMPORTANT**
> 本文仅讨论 .NET Framework 中的大型对象堆和 Windows 系统上运行的 .NET Core。不包括在其他平台上的 .NET 实现上运行的 LOH。

对象如何在 LOH 上结束

如果对象的大小大于或等于 85,000 字节，将被视为大型对象。此数字根据性能优化确定。对象分配请求为 85,000 字节或更大时，运行时会将其分配到大型对象堆。

若要了解其意义，可查看垃圾回收器的部分相关基础知识。

垃圾回收器是分代回收器。它包含三代：第 0 代、第 1 代和第 2 代。包含 3 代的原因是，在优化良好的应用中，大部分对象都在第 0 代就清除了。例如，在服务器应用中，与每个请求相关的分配应在请求完成后清除。仍存在的分配请求将转到第 1 代，并在那里进行清除。从本质上讲，第 1 代是新对象区域与生存期较长的对象区域之间的缓冲区。

小型对象始终在第 0 代中进行分配，或者根据它们的生存期，可能会提升为第 1 代或第 2 代。大型对象始终在第 2 代中进行分配。

大型对象属于第 2 代，因为只有在第 2 代回收期间才能回收它们。回收一代时，同时也会回收它前面的所有代。例如，执行第 1 代 GC 时，将同时回收第 1 代和第 0 代。执行第 2 代 GC 时，将回收整个堆。因此，第 2 代 GC 还可称为“完整 GC”。本文引用第 2 代 GC 而不是完整 GC，但这两个术语是可以互换的。

代可提供 GC 堆的逻辑视图。实际上，对象存在于托管堆段中。托管堆段是 GC 通过调用 VirtualAlloc 功能代表托管代码在操作系统上保留的内存块。加载 CLR 时，GC 分配两个初始堆段：一个用于小型对象(小型对象堆或 SOH)，一个用于大型对象(大型对象堆)。

然后，通过将托管对象置于这些托管堆段上来满足分配请求。如果该对象小于 85,000 字节，则将它置于 SOH 的段上，否则，将它置于 LOH 段。随着分配到各段上的对象越来越多，会以较小块的形式提交这些段。对于 SOH，GC 未处理的对象将提升为下一代。第 0 代回收未处理的对象现在视为第 1 代对象，以此类推。但是，最后一代回收未处理的对象仍会被视为最后一代中的对象。也就是说，第 2 代垃圾回收未处理的对象仍是第 2 代对象；LOH 未处理的对象仍是 LOH 对象(由第 2 代回收)。

用户代码只能在第 0 代(小型对象)或 LOH(大型对象)中分配。只有 GC 可以在第 1 代(通过提升第 0 代回收未处理的对象)和第 2 代(通过提升第 1 代和第 2 代回收未处理的对象)中“分配”对象。

触发垃圾回收后，GC 将寻找存在的对象并将它们压缩。但是由于压缩费用很高，GC 会扫过 LOH，列出没有被清除的对象列表以供以后重新使用，从而满足大型对象的分配请求。相邻的被清除对象将组成一个自由对象。

.NET Core 和 .NET Framework(从 .NET Framework 4.5.1 开始)包括 GCSettings.LargeObjectHeapCompactionMode 属性，该属性可让用户指定在下一完整阻止 GC 期间压缩 LOH。并且在以后，.NET 可能会自动决定压缩 LOH。这就意味着，如果分配了大型对象并希望确保它们不被移动，则应将其固定起来。

图 1 说明了一种情况，在第一次第 0 代 GC 后 GC 形成了第 1 代，其中 `Obj1` 和 `Obj3` 被清除；在第一次第 1 代 GC 后形成了第 2 代，其中 `Obj2` 和 `Obj5` 被清除。请注意此图和下图仅用于说明，它们只包含能更好展示堆上的情况的极少几个对象。实际上，GC 中通常包含更多的对象。

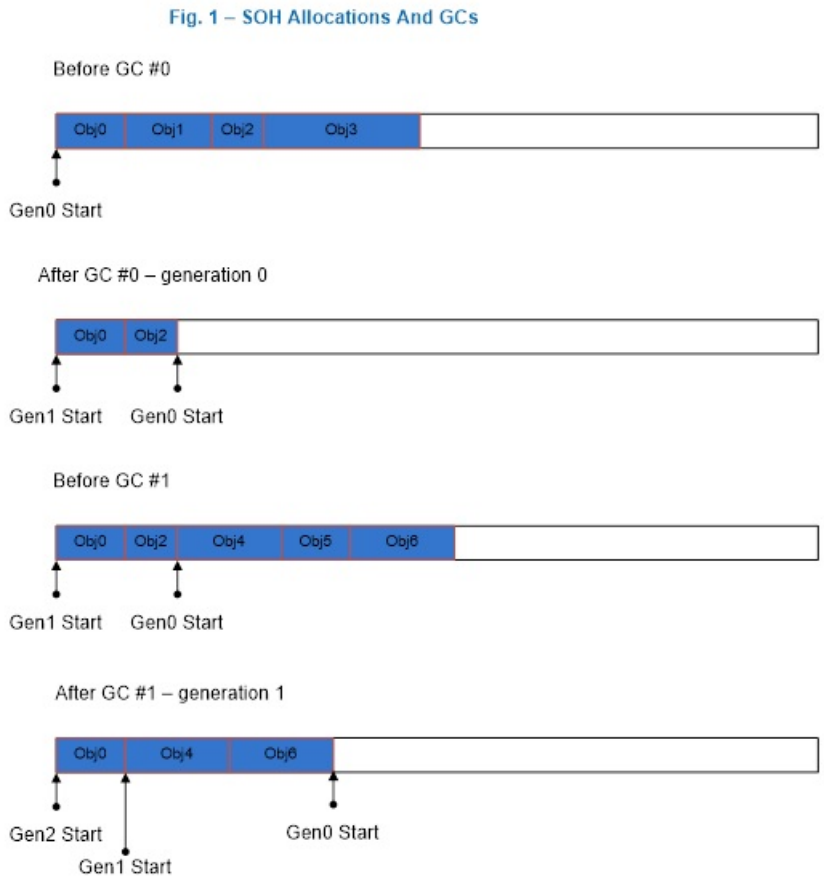


图 1：第 0 代和第 1 代 GC。

图 2 显示了第 2 代 GC 发现 `Obj1` 和 `Obj2` 被清除后，GC 在内存中形成了相邻的可用空间，由 `Obj1` 和 `Obj2` 占用，然后用于满足 `Obj4` 的分配要求。从最后一个对象 `Obj3` 到此段末尾的空间仍可用于满足分配请求。

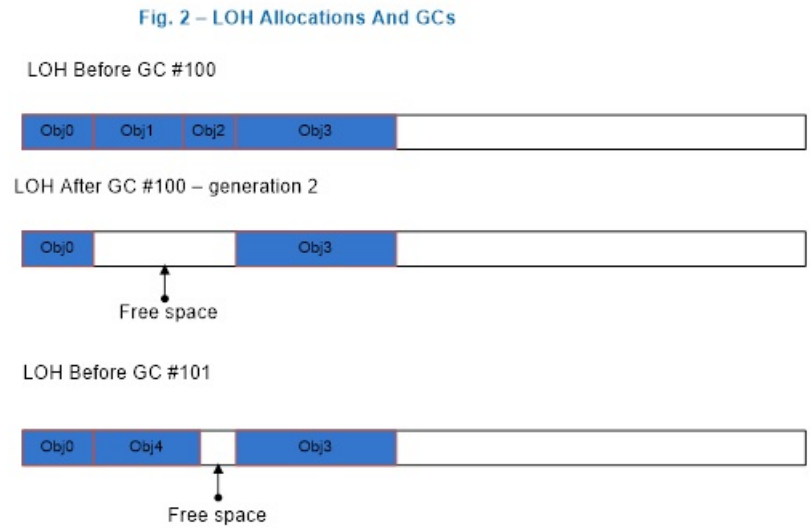


图 2：第 2 代 GC 后

如果没有足够的可用空间来容纳大型对象分配请求，GC 首先尝试从操作系统获取更多段。如果失败了，它将触发第 2 代 GC，试图释放部分空间。

在第 1 代或第 2 代 GC 期间，垃圾回收器会通过调用 VirtualFree 功能将不包含活动对象的段释放回操作系统。将退回最后一个活动对象到段末尾的空间（第 0 代/第 1 代存在的短暂段上的空间除外，垃圾回收器会在该段上会保存部分提交内容，因为应用程序将在其中立即分配）。而且，尽管已重置可用空间，但仍会提交它们，这意味

着操作系统无需将其中的数据重新写入磁盘。

由于 LOH 仅在第 2 代 GC 期间进行回收，所以 LOH 段仅在此类 GC 期间可用。图 3 说明了一种情况，在此情况下，垃圾回收器将某段（段 2）释放回操作系统并且退回剩余段上更多的空间。如果需要使用该段末尾的已退回空间来满足大型对象分配请求，它会再次提交该内存。（有关提交/退回的解释说明，请参阅 VirtualAlloc 的文档）。

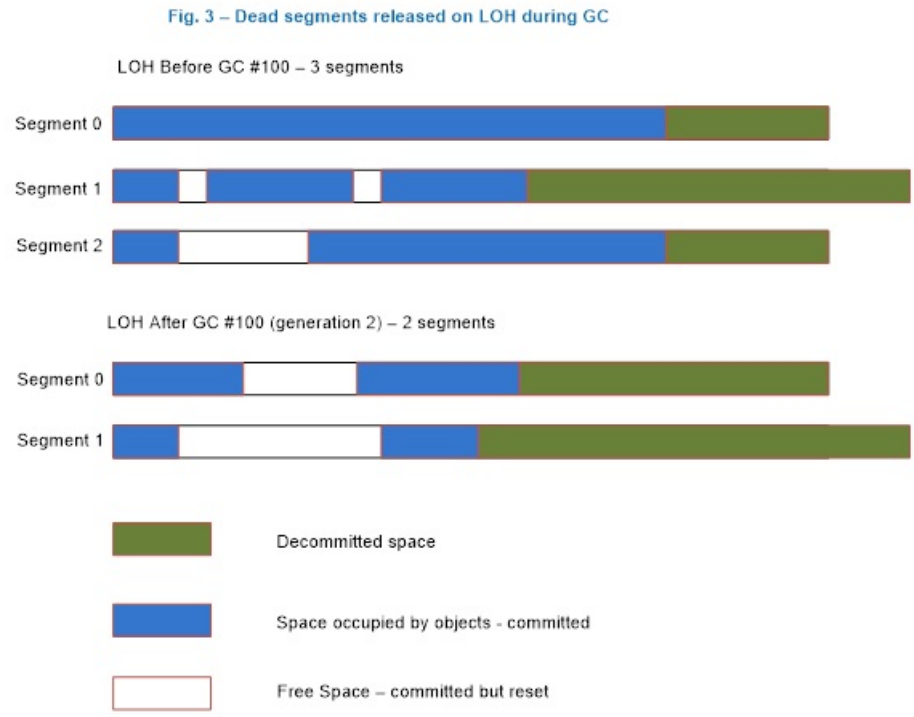


图 3：第 2 代 GC 后的 LOH

何时收集大型对象？

通常情况下，出现以下三种情形中的任一情况，都会执行 GC：

- 分配超出第 0 代或大型对象阈值。

  阈值是某代的属性。垃圾回收器在其中分配对象时，会为代设置阈值。超出阈值后，会在该代上触发 GC。因此，分配小型或大型对象时，需要分别使用第 0 代和 LOH 的阈值。当垃圾回收器分配到第 1 代和第 2 代中时，将使用它们的阈值。运行此程序时，会动态调整这些阈值。

  这是典型情况，大部分 GC 执行都因为托管堆上的分配。

- 调用 GC.Collect 方法。

  如果调用无参数 GC.Collect() 方法，或另一个重载作为参数传递到 GC.MaxGeneration，将会一起收集 LOH 和剩余的托管堆。

- 系统处于内存不足的状况。

  垃圾回收器收到来自操作系统 的高内存通知时，会发生以上情况。如果垃圾回收器认为执行第 2 代 GC 会有效率，它将触发第 2 代。

LOH 性能意义

大型对象堆上的分配通过以下几种方式影响性能。

- 分配成本。

  CLR 确保清除了它提供的每个新对象的内存。这意味着大型对象的分配成本完全由清理的内存（除非触发了 GC）决定。如果需要 2 轮才能清除一个字节，即需要 170,000 轮才能清除最小的大型对象。清除 2GHz 计算机上 16MB 对象的内存大约需要 16ms。这些成本相当大。

- 回收成本。

  因为 LOH 和第 2 代一起回收，如果超出了它们之中任何一个的阈值，则触发第 2 代回收。如果由于 LOH 触发第 2 代回收，第 2 代没有必要在 GC 后变得更小。如果第 2 代上数据不多，则影响较小。但是，如果第 2 代很大，则触发多次第 2 代 GC 可能会产生性能问题。如果很多大型对象都在非常短暂的基础上进行分配，并且拥有大型 SOH，则可能会花费太多时间来执行 GC。除此之外，如果连续分配并且释放真正的大型对象，那么分配成本可能会增加。

- 具有引用类型的数组元素。

  LOH 上的特大型对象通常是数组（很少会有非常大的实例对象）。如果数组的元素有丰富的引用，则可能产生成本；如果元素没有丰富的引用，将不会产生此类成本。如果元素不包含任何引用，则垃圾回收器根本无需处理此数组。例如，如果使用数组存储二进制树中的节点，一种实现方法是按实际节点引用某个节点的左侧节点和右侧节点：

  ```
  class Node
  {
     Data d;
     Node left;
     Node right;
  };

  Node[] binary_tr = new Node [num_nodes];
  ```

  如果 `num_nodes` 非常大，则垃圾回收器需要处理每个元素的至少两个引用。另一种方法是存储左侧节点和右侧节点的索引：

  ```
  class Node
  {
     Data d;
     uint left_index;
     uint right_index;
  } ;
  ```

  不要将左侧节点的数据引用为 `left.d`，而是将其引用为 `binary_tr[left_index].d`。而垃圾回收器无需查看左侧节点和右侧节点的任何引用。

在这三种因素中，前两个通常比第三个更重要。因此，建议分配重复使用的大型对象池，而不是分配临时大型对象。

收集 LOH 的性能数据

收集特定区域的性能数据之前，应完成以下操作：

1. 找到应查看此区域的证据。

2. 排查你知道的其他区域，确保未发现可解释上述性能问题的内容。

参阅博客尝试找出解决方案之前先了解问题获取内存和 CPU 的基础知识的详细信息。

可使用以下工具来收集 LOH 性能数据：

- .NET CLR 内存性能计数器
- ETW 事件
- 调试器

.NET CLR 内存性能计数器

这些性能计数器通常是调查性能问题的第一步(但是推荐使用 ETW 事件)。通过添加所需计数器配置性能监视器，如图 4 所示。与 LOH 相关的是：

- 第 2 代回收次数

  显示自进程开始起第 2 代 GC 发生的次数。此计数器在第 2 代回收结束时递增(也称为完整垃圾回收)。此计数器显示上次观测的值。

- 大型对象堆大小

  以字节显示当前大小，包括 LOH 的可用空间。此计数器在垃圾回收结束时更新，不在每次分配时更新。

查看性能计数器的常用方法是使用性能监视器 (perfmon.exe)。使用"添加计数器"可为关注的进程添加感兴趣的计数器。可将性能计数器数据保存在日志文件中，如图 4 所示：

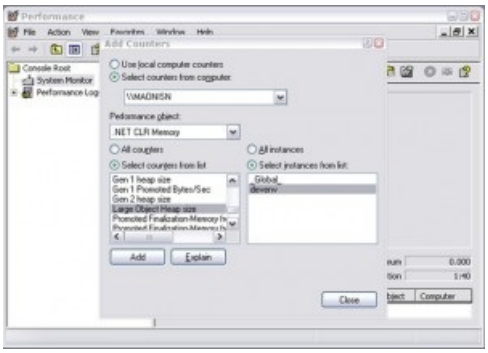


图 4：第 2 代 GC 后的 LOH

也可以编程方式查询性能计数器。大部分人在例行测试过程中都采用此方式进行收集。如果发现计数器显示的值不正常，则可以使用其他方法获得更多详细信息以帮助调查。

> **NOTE**
> 建议使用 ETW 事件代替性能计数，因为 ETW 提供更丰富的信息。

ETW 事件

垃圾回收器提供丰富的 ETW 事件集，帮助了解堆的工作内容和工作原理。以下博客文章演示了如何使用 ETW 收集和了解 GC 事件：

- GC ETW 事件 - 1
- GC ETW 事件 - 2
- GC ETW 事件 - 3
- GC ETW 事件 - 4

若要标识由临时 LOH 分配造成的过多第 2 代 GC 次数，请查看 GC 的"触发原因"列。有关仅分配临时大型对象的简单测试，可使用以下 PerfView 命令行收集 ETW 事件的信息：

```
perfview /GCCollectOnly /AcceptEULA /nogui collect
```

结果类似于以下类容：

Gen 2 for Process 10720: testlarge

GC Events by Time
All times are in msec. Hover over columns for help.

GC Index	Pause Start	Trigger Reason	Gen	Suspend Msec	Pause MSec	% Pause Time	% GC	Gen0 Alloc MB	Gen0 Alloc Rate MB/sec	Peak MB	After MB	Ratio Peak/After	Promoted MB	Gen0 MB	Gen0 Survival Rate %	Gen0 Frag %	Gen1 MB	Gen1 Survival Rate %	Gen1 Frag %	Gen2 MB	Gen2 Survival Rate %	Gen2 Frag %	LOH MB	LOH Survival Rate %	LOH Frag %	Finalizable Surv MB	Pinned Obj
1	2,248.409	AllocLarge	2N	0.007	0.422	0.0	NaN	0.000	0.00	3.244	0.043	76.19	0.041	0.000	85	0.00	0.007	0	15.25	0.000	0	0.00	0.035	1	0.36	0.00	5
2	2,248.907	AllocLarge	2N	0.002	0.056	43.3	NaN	0.000	0.00	3.244	0.043	76.14	0.041	0.000	0	0.00	0.000	100	0.00	0.007	0	15.53	0.035	1	0.36	0.00	5
3	2,249.003	AllocLarge	2N	0.001	0.052	52.9	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
4	2,249.101	AllocLarge	2N	0.001	0.050	51.8	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
5	2,249.197	AllocLarge	2N	0.001	0.051	51.4	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
6	2,249.293	AllocLarge	2N	0.001	0.051	52.5	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
7	2,249.389	AllocLarge	2N	0.001	0.052	53.1	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
8	2,249.485	AllocLarge	2N	0.001	0.051	52.7	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
9	2,249.572	AllocLarge	2N	0.001	0.043	53.4	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
10	2,249.651	AllocLarge	2N	0.001	0.043	53.2	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
11	2,249.748	AllocLarge	2N	0.001	0.048	46.3	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
12	2,249.834	AllocLarge	2N	0.001	0.068	63.7	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
13	2,249.937	AllocLarge	2N	0.001	0.068	64.9	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
14	2,250.060	AllocLarge	2N	0.001	0.037	39.6	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
15	2,250.143	AllocLarge	2N	0.001	0.051	52.2	NaN	0.000	0.00	3.244	0.043	76.15	0.041	0.000	0	0.00	0.000	0	0.00	0.007	100	15.53	0.035	1	0.36	0.00	5
16	2,250.232	AllocLarge	2N	0.001	0.042	51.2	NaN	0.005	121.73	3.249	0.047	68.47	0.046	0.000	99	0.00	0.005	0	0.49	0.007	100	15.53	0.035	1	0.36	0.00	5
17	2,250.309	AllocLarge	2N	0.001	0.040	52.0	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	99	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
18	2,250.384	AllocLarge	2N	0.001	0.039	51.7	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
19	2,250.459	AllocLarge	2N	0.001	0.038	50.6	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
20	2,250.534	AllocLarge	2N	0.001	0.039	50.9	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
21	2,250.609	AllocLarge	2N	0.001	0.042	53.1	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
22	2,250.686	AllocLarge	2N	0.001	0.042	52.7	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
23	2,250.763	AllocLarge	2N	0.001	0.044	54.6	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5
24	2,250.842	AllocLarge	2N	0.001	0.039	51.3	NaN	0.000	0.00	3.249	0.047	68.43	0.046	0.000	0	0.00	0.000	0	0.00	0.012	100	9.69	0.035	1	0.36	0.00	5

图 5：使用 PerfView 显示的 ETW 事件

如下所示，所有 GC 都是第 2 代 GC，并且都由 AllocLarge 触发，这表示分配大型对象会触发此 GC。我们知道这些分配是临时的，因为“LOH 未清理率 %”列显示为 1%。

可以收集显示分配这些大写对象的人员的其他 ETW 事件。以下命令行：

```
perfview /GCOnly /AcceptEULA /nogui collect
```

收集 AllocationTick 事件，大约每 10 万次分配就会触发该事件。换句话说，每次分配大型对象都会触发事件。然后可查看某个 GC 堆分配视图，该视图显示分配大型对象的调用堆栈：

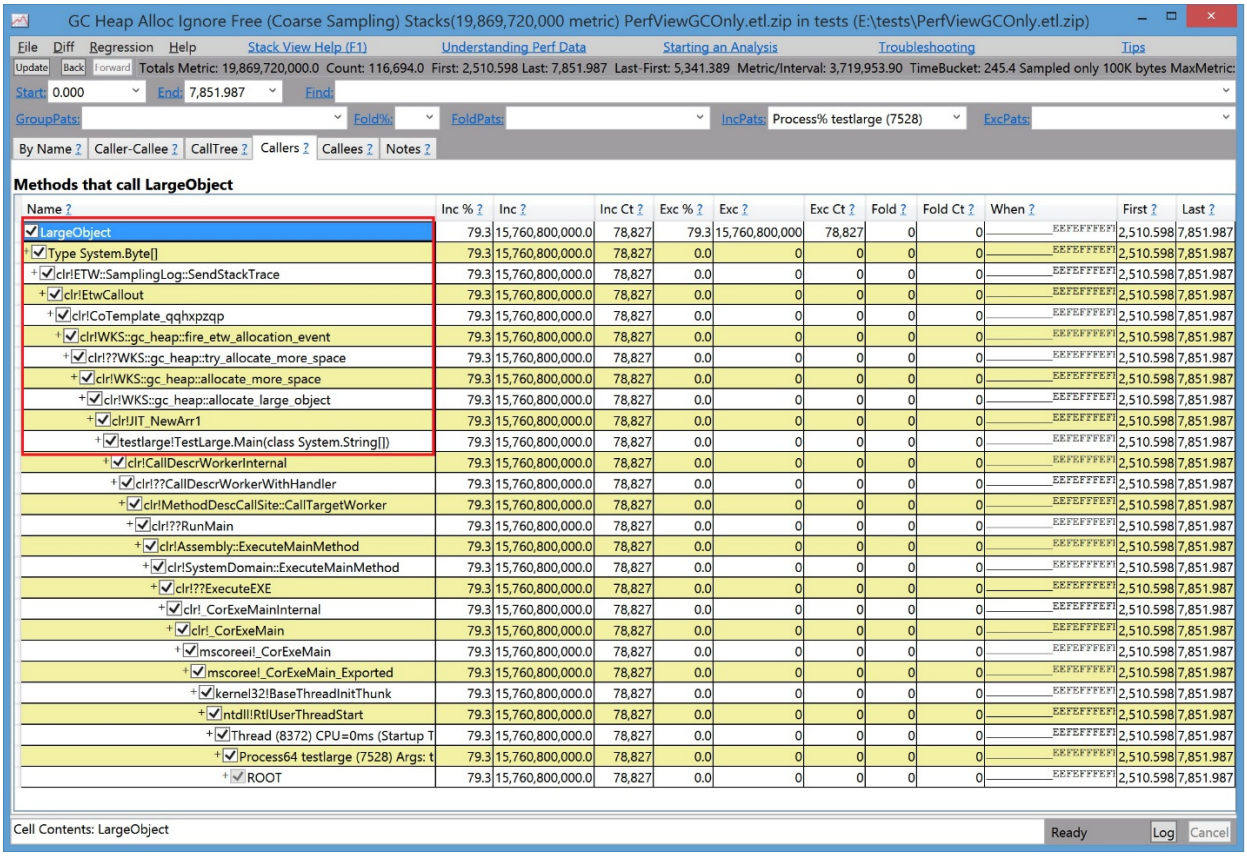


图 6：GC 堆分配视图

如图所示，这是从 `Main` 方法分配大型对象的简单测试。

调试器

如果只有内存转储，则需要查看 LOH 上实际有哪些对象，你可使用 .NET 提供的 SoS 调试器扩展来查看。

> **NOTE**
> 此部分提到的调试命令适用于 Windows 调试器。

以下内容显示了分析 LOH 的示例输出：

```
0:003> .loadby sos mscorwks
0:003> !eeheap -gc
Number of GC Heaps: 1
generation 0 starts at 0x013e35ec
sdgeneration 1 starts at 0x013e1b6c
generation 2 starts at 0x013e1000
ephemeral segment allocation context: none
segment   begin allocated     size
0018f2d0 790d5588 790f4b38 0x0001f5b0(128432)
013e0000 013e1000 013e35f8 0x000025f8(9720)
Large object heap starts at 0x023e1000
segment   begin allocated     size
023e0000 023e1000 033db630 0x00ffa630(16754224)
033e0000 033e1000 043cdf98 0x00fecf98(16699288)
043e0000 043e1000 05368b58 0x00f87b58(16284504)
Total Size 0x2f90cc8(49876168)
------------------------------
GC Heap Size 0x2f90cc8(49876168)
0:003> !dumpheap -stat 023e1000 033db630
total 133 objects
Statistics:
MT    Count    TotalSize Class Name
001521d0       66     2081792     Free
7912273c       63     6663696 System.Byte[]
7912254c       4     8008736 System.Object[]
Total 133 objects
```

LOH 堆大小为 (16,754,224 + 16,699,288 + 16,284,504) = 49,738,016 字节。在地址 023e1000 和地址 033db630 之间，8,008,736 字节由 System.Object 对象的数组占用，6,663,696 字节由 System.Byte 对象的数组占用，2,081,792 字节由可用空间占用。

有时，调试器显示 LOH 的总大小少于 85,000 个字节。这是由于运行时本身使用 LOH 分配某些小于大型对象的对象引起的。

因为不会压缩 LOH，有时会怀疑 LOH 是碎片源。碎片表示：

- 托管堆的碎片由托管对象之间的可用空间量来表示。在 SoS 中，`!dumpheap -type Free` 命令显示托管对象之间的可用空间量。
- 虚拟内存 (VM) 地址空间的碎片是标识为 `MEM_FREE` 的内存。可在 windbg 中使用各种调试器命令来获取碎片。

  以下示例显示 VM 空间中的碎片：

```
0:000> !address
00000000 : 00000000 - 00010000
Type     00000000
Protect 00000001 PAGE_NOACCESS
State   00010000 MEM_FREE
Usage   RegionUsageFree
00010000 : 00010000 - 00002000
Type     00020000 MEM_PRIVATE
Protect 00000004 PAGE_READWRITE
State   00001000 MEM_COMMIT
Usage   RegionUsageEnvironmentBlock
00012000 : 00012000 - 0000e000
Type     00000000
Protect 00000001 PAGE_NOACCESS
State   00010000 MEM_FREE
Usage   RegionUsageFree
… [omitted]
-------------------- Usage SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Pct(Busy)   Usage
701000 (   7172) : 00.34%   20.69%   : RegionUsageIsVAD
7de15000 ( 2062420) : 98.35%   00.00%   : RegionUsageFree
1452000 (   20808) : 00.99%   60.02%   : RegionUsageImage
300000 (   3072) : 00.15%   08.86%   : RegionUsageStack
3000 (     12) : 00.00%   00.03%   : RegionUsageTeb
381000 (   3588) : 00.17%   10.35%   : RegionUsageHeap
0 (       0) : 00.00%   00.00%   : RegionUsagePageHeap
1000 (       4) : 00.00%   00.01%   : RegionUsagePeb
1000 (       4) : 00.00%   00.01%   : RegionUsageProcessParametrs
2000 (       8) : 00.00%   00.02%   : RegionUsageEnvironmentBlock
Tot: 7fff0000 (2097088 KB) Busy: 021db000 (34668 KB)

-------------------- Type SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Usage
7de15000 ( 2062420) : 98.35%   : <free>
1452000 (   20808) : 00.99%   : MEM_IMAGE
69f000 (   6780) : 00.32%   : MEM_MAPPED
6ea000 (   7080) : 00.34%   : MEM_PRIVATE

-------------------- State SUMMARY --------------------------
TotSize (     KB)   Pct(Tots) Usage
1a58000 (   26976) : 01.29%   : MEM_COMMIT
7de15000 ( 2062420) : 98.35%   : MEM_FREE
783000 (   7692) : 00.37%   : MEM_RESERVE

Largest free region: Base 01432000 - Size 707ee000 (1843128 KB)
```

通常看到的更多是由临时大型对象导致的 VM 碎片，这些对象要求垃圾回收器频繁从操作系统获取新的托管堆段，并将空托管堆段释放回操作系统。

要验证 LOH 是否会生成 VM 碎片，可在 VirtualAlloc 和 VirtualFree 上设置一个断点，查看是谁调用了它们。例如，如果想知道谁曾尝试从操作系统分配大于 8MBB 的虚拟内存块，可按以下方式设置断点：

```
bp kernel32!virtualalloc "j (dwo(@esp+8)>800000) 'kb';'g'"
```

只有在分配大小大于 8MB (0x800000) 的情况下调用 VirtualAlloc 时，此命令才会进入调试器并显示调用堆栈。

CLR 2.0 增加了称为“VM 囤积”的功能，用于频繁获取和释放段（包括在大型和小型对象堆上）的情况。若要指定 VM 囤积，可通过托管 API 指定称为 `STARTUP_HOARD_GC_VM` 的启动标记。CLR 退回这些段上的内存并将其添加到备用列表中，而不会将该空段释放回操作系统。（请注意 CLR 不会针对太大型的段执行此操作。）CLR 稍后将使用这些段来满足新段请求。下一次应用需要新段时，CLR 将使用此备用列表中的某个足够大的段。

VM 囤积还可用于想要保存已获取段的应用程序（例如属于系统上运行的主要应用的部分服务器应用），以避免内

存不足的异常。

强烈建议你在使用此功能时认真测试应用程序，以确保应用程序的内存使用情况比较稳定。

垃圾回收和性能

2021/11/16 •

本主题介绍与垃圾回收和内存使用情况相关的问题。它解决了关于托管堆的问题，并解释了如何最小化垃圾回收对应用程序的影响。每个问题具有访问可用来调查问题的过程的链接。

性能分析工具

以下各节介绍了可用于调查内存使用情况和垃圾回收问题的工具。本主题中稍后提供的过程将引用这些工具。

内存性能计数器

可以使用性能计数器来收集性能数据。有关说明，请参阅运行时分析。如 .NET 中的性能计数器中所述，性能计数器的 .NET CLR 内存类别提供有关垃圾回收器的信息。

用 SOS 调试

可以使用 Windows 调试器 (WinDbg) 检查托管堆上的对象。

若要安装 WinDbg，请从下载 Windows 调试工具页安装 Windows 调试工具。

垃圾回收 ETW 事件

Windows 事件跟踪 (ETW) 是一个跟踪系统，对由 .NET 提供的分析和调试支持提供补充。从 .NET Framework 4 开始，垃圾回收 ETW 事件将捕获有用信息，用于从统计的角度来分析托管堆。例如，在将要发生垃圾回收时引发的 `GCStart_V1` 事件提供了以下信息：

- 正在收集哪一代对象。
- 是什么触发了垃圾回收。
- 垃圾回收的类型（并发或非并发）。

ETW 事件日志有效，且不会掩盖与垃圾回收相关的任何性能问题。一个进程可以通过结合 ETW 事件来提供其自身的事件。登录后，可以关联应用程序事件和垃圾回收事件，以确定如何以及何时出现堆问题。例如，服务器应用程序可以在客户端请求开始和结束时提供事件。

分析 API

公共语言运行时 (CLR) 分析接口将提供有关垃圾回收期间受影响对象的详细信息。垃圾回收开始和结束时，可以通知探查器。它可以提供有关托管堆上对象的报告，其中包括每一代对象的标识。有关详细信息，请参阅分析概述。

探查器可以提供全面的信息。但是，复杂的探查器可能会修改应用程序的行为。

应用程序域资源监控

从 .NET Framework 4 开始，应用程序域资源监视 (ARM) 使主机可以通过应用程序域监视 CPU 和内存使用情况。有关详细信息，请参阅应用程序域资源监控。

故障排除性能问题

第一步是确定问题是否确实为垃圾回收。如果确定是，则从以下列表进行选择，以解决该问题。

- 引发内存不足异常
- 进程占用过多内存
- 垃圾回收器回收对象的速度不够快

- 托管堆太零碎
- 垃圾回收暂停时间太长
- 第 0 代太大
- 垃圾回收期间的 CPU 使用率太高

问题：抛出内存不足异常

对于引发的托管 OutOfMemoryException，存在以下两种合理的情况：

- 虚拟内存不足。

  垃圾回收器按预先确定大小的分段来分配系统内存。如果分配需要其他段，但在进程的虚拟内存空间中没有剩余的连续可用块了，则托管堆的分配将失败。

- 没有足够的物理内存来分配。

tttt
确定是否已托管内存不足异常。 确定可保留的虚拟内存量。 确定是否有足够的物理内存。

如果确定异常不合法，请使用以下信息与 Microsoft 客户服务和支持联系：

- 带有托管内存不足异常的堆栈。
- 完整内存转储。
- 证明这不是合法内存不足异常的数据包括显示虚拟或物理内存不是问题的数据。

问题：进程占用过多内存

通常会假设 Windows 任务管理器"性能"选项卡上的内存使用量显示可以指示何时使用了太多内存。然而，该显示与工作集相关；它不提供有关虚拟内存使用量的信息。

如果确定问题是托管堆引发的，必须测量一段时间的托管堆，以确定模式。

如果确定问题不是托管堆引发的，则必须使用本地调试。

tttt
确定可保留的虚拟内存量。 确定托管堆的内存提交量。 确定托管堆的内存保留量。 确定第 2 代中的大型对象。 确定对对象的引用。

问题：垃圾回收器回收对象的速度不够快

当出现对象好像未按垃圾回收的预期进行回收的情况时，必须确定是否存在任何对这些对象的强引用。

如果没有对包含死对象的一代进行垃圾回收，这表示尚未运行死对象的终结器，你也可能会遇到以上问题。例

如，当正在运行一个单线程单元 (STA) 应用程序并且服务终结器队列的线程不能调用至其中时，可能发生这种问题。

检查对对象的引用。 确定是否已运行终结器。 确定是否存在等待终结的对象。

问题：**托管堆太零碎**

碎片级别将计算为可用空间占这一代已分配的总内存的比率。对于第 2 代，可接受的碎片级别不能超过 20%。因为第 2 代可以变得很大，所以碎片的比率比绝对值更重要。

第 0 代中存在大量可用空间，这不是问题，因为新的对象将在其中进行分配。

碎片始终出现在大型对象堆中，因为它没有进行压缩。相邻的可用对象会自然地折叠至一个单个的空间，以满足大型对象的分配请求。

在第 1 代和第 2 代中，碎片可能会成为问题。如果它们在垃圾回收后还有大量的可用空间，则应用程序对象的使用可能需要进行修改，并且应考虑重新评估长期对象的生存期。

固定对象过多可能会增加碎片。如果碎片太多，则可以固定许多对象。

如果虚拟内存的碎片阻止垃圾回收器添加段，原因可能是下列之一：

- 频繁加载和卸载许多小的程序集。
- 与非托管代码互操作时，保留了太多对 COM 对象的引用。
- 大型暂时性对象的创建会导致大型对象堆频繁分配和释放堆段。

  当承载 CLR 时，应用程序可以请求垃圾回收器保留其片段。这将减少段分配的频率。通过使用 STARTUP_FLAGS 枚举中的 STARTUP_HOARD_GC_VM 标志来完成。

确定托管堆中的可用空间量。 确定固定对象的数目。

如果认为没有出现碎片的合理原因，请与 Microsoft 客户服务和支持联系。

问题：**垃圾回收暂停时间太长**

由于垃圾回收软实时操作，因此应用程序必须能够容忍暂停。软实时的一个衡量标准是 95% 的操作必须按时完成。

在并发垃圾回收中，允许托管线程在一个回收过程中运行，这意味着暂停时间会非常短。

短暂的垃圾回收（第 0 代和第 1 代）只会持续几毫秒，所以减少暂停时间通常是不可行的。然而，你可以通过更改应用程序的分配请求的模式，在第 2 代回收中减少暂停。

另一个更准确的方法是使用垃圾回收 ETW 事件。可以通过为某个事件序列添加时间戳的差异来查找回收的计时。整个集合序列包括暂停执行引擎、垃圾回收本身以及恢复执行引擎。

可以使用垃圾回收通知，确定服务器是否将要进行第 2 代回收，以及将请求重新路由到另一个服务器是否可以减轻任何暂停问题。

""

确定垃圾回收中的时长。

确定导致垃圾回收的原因。

问题：**第 0 代太大**

第 0 代可能在 64 位系统上有更多的对象，尤其是当使用服务器垃圾回收而不是工作站垃圾回收时。这是因为触发 0 代垃圾回收的阈值在这些环境中更高，且 0 代回收可以变得更大。触发垃圾回收之前，当应用程序分配更多的内存时，性能将会提高。

问题：**垃圾回收期间的 CPU 使用率太高**

在垃圾回收期间，CPU 的使用率会很高。如果在垃圾回收中花费大量的处理时间，则回收的数量将过于频繁或回收的持续时间将过长。托管堆上增加的对象分配率将导致垃圾回收更频繁地发生。减少分配速率可减少垃圾回收的频率。

可以通过使用 `Allocated Bytes/second` 性能计数器来监视分配速率。有关更多信息，请参阅 .NET 中的性能计数器。

收集的持续时间是分配后幸存对象数量的主要因素。如果有许多对象仍需收集，则垃圾回收器必须要检查大量的内存。压缩幸存对象的工作很耗时。若要确定回收期间处理对象的数量，请在指定代的垃圾回收结束时，在调试器中设置一个断点。

""

确定 CPU 的使用率过高是否由垃圾回收引起。

在垃圾回收结束时，设置一个断点。

故障排除指南

本部分介绍在开始调查时应考虑的准则。

工作站或服务器垃圾回收

确定是否正在使用正确的垃圾回收类型。如果应用程序使用多个线程和对象实例，则使用服务器垃圾回收，而不是工作站垃圾回收。服务器垃圾回收在多个线程上进行操作，而工作站垃圾回收则需要应用程序的多个实例运行它们自己的垃圾回收线程并争取 CPU 时间。

低负载且不常在后台(如服务)执行任务的应用程序，可以在禁用并发垃圾回收的情况下使用工作站垃圾回收。

何时衡量托管堆的大小

除非使用探查器，否则必须建立一致的测量模式，以有效地诊断性能问题。若要建立一个计划，请考虑以下几点：

- 如果在第 2 代垃圾回收后测量，则整个托管的堆将不再存在垃圾(死对象)。
- 如果在一个 0 代垃圾回收后立即进行测量，则尚不会收集第 1 代和 2 中的对象。
- 如果在垃圾回收之前立即进行测量，则你将在垃圾回收启动之前，测量尽可能多的分配。
- 在垃圾回收期间进行测量会出现问题，因为垃圾回收器的数据结构对于遍历是无效状态，可能不能提供完整的结果。这是设计使然。
- 当与并发垃圾回收一起使用工作站垃圾回收时，回收的对象不会进行压缩，因此，堆的大小可能会相同或更大(碎片可以使它看起来更大)。

- 第 2 代上的并发垃圾回收在物理内存加载过高时，将被延迟。

以下过程介绍如何设置一个断点，以便测量托管堆。

若要在垃圾回收结束时设置一个断点

- 在加载了 SOS 调试器扩展的 WinDbg 中，键入以下命令：

  ```
  bp mscorwks!WKS::GCHeap::RestartEE "j
  (dwo(mscorwks!WKS::GCHeap::GcCondemnedGeneration)==2) 'kb';'g'"
  ```

  其中，GcCondemnedGeneration 设置为所需的代。此命令要求私有符号。

  如果在已回收第 2 代对象以进行垃圾回收后执行 RestartEE，则此命令会强制中断。

  在服务器垃圾回收中，只有一个线程调用 RestartEE，因此在第 2 代垃圾回收期间，此断点只会出现一次。

性能检查过程

本部分将介绍下列过程，以避免造成性能问题的原因：

- 确定问题是否由垃圾回收引起。
- 确定是否已托管内存不足异常。
- 确定可保留的虚拟内存量。
- 确定是否有足够的物理内存。
- 确定托管堆的内存提交量。
- 确定托管堆的内存保留量。
- 确定第 2 代中的大型对象。
- 确定对对象的引用。
- 确定是否已运行终结器。
- 确定是否存在等待终结的对象。
- 确定托管堆中的可用空间量。
- 确定固定对象的数目。
- 确定垃圾回收中的时长。
- 确定触发垃圾回收的原因。
- 确定 CPU 的使用率过高是否由垃圾回收引起。

若要确定问题是否是垃圾回收引起

- 请检查以下两个内存性能计数器：
  - **GC 所占时间百分比**。显示执行最后一个垃圾回收周期后，执行垃圾回收所用运行时间的百分比。使用此计数器确定垃圾回收器是否花费太多时间来使托管堆空间可用。如果垃圾回收所用的时间相对较短，这可能表示托管堆之外存在资源问题。当涉及并发或后台垃圾回收时，此计数器可能不准确。
  - **已提交的字节总数**。显示垃圾回收器当前已提交的虚拟内存量。使用此计数器确定垃圾回收器所占用的内存是否是应用程序所使用的内存的过多部分。

大多数的内存性能计数器会在每次垃圾回收结束时进行更新。因此，它们可能不会反映你希望了解的当前情况。

若要确定是否已托管内存不足异常

1. 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入打印异常 (**pe**) 命令：

   !pe

   如果已托管异常，OutOfMemoryException 将显示为异常类型，如以下示例中所示。

   ```
   Exception object: 39594518
   Exception type: System.OutOfMemoryException
   Message: <none>
   InnerException: <none>
   StackTrace (generated):
   ```

2. 如果输出没有指定异常，则必须确定内存不足异常来自哪个线程。在调试器中键入以下命令，以显示所有带调用堆栈的线程：

   ~*kb

   具有存在异常调用的堆栈的线程会由 `RaiseTheException` 参数进行指示。这是托管异常对象。

   ```
   28adfb44 7923918f 5b61f2b4 00000000 5b61f2b4 mscorwks!RaiseTheException+0xa0
   ```

3. 可以使用以下命令来转储嵌套的异常。

   !pe -nested

   如果找不到任何异常，则非托管代码将产生内存不足异常。

若要确定可保留的虚拟内存量

- 在加载了 SOS 调试器扩展的 WinDbg 中键入以下命令，以获取最大的可用区域：

  !address -summary

  最大可用区域将如以下输出所示进行显示。

  ```
  Largest free region: Base 54000000 - Size 0003A980
  ```

  在此示例中，最大可用区域的大小大约为 24000 KB（按十六进制形式则为 3A980）。此区域比垃圾回收器对分段所需的大小要小得多。

  - 或 -

- 使用 **vmstat** 命令：

  !vmstat

  最大可用区域是 MAXIMUM 列中的最大值，如以下输出所示。

```
TYPE        MINIMUM   MAXIMUM     AVERAGE   BLK COUNT   TOTAL
~~~~        ~~~~~~~   ~~~~~~~     ~~~~~~~   ~~~~~~~~~~  ~~~~
Free:
Small       8K        64K         46K       36          1,671K
Medium      80K       864K        349K      3           1,047K
Large       1,384K    1,278,848K  151,834K  12          1,822,015K
Summary     8K        1,278,848K  35,779K   51          1,824,735K
```

若要确定是否有足够的物理内存

1. 则启动 Windows 任务管理器。
2. 在"性能"选项卡上，查看已提交的值。（在 Windows 7 中，查看"系统组"中的"提交 (KB)"。）

   如果"总数"接近"限值"，则物理内存将不足。

若要确定托管堆的内存提交量

- 使用 `# Total committed bytes` 内存性能计数器获取托管堆提交的字节数。垃圾回收器根据需要在某个段上提交区块，但不会全部在同一时间进行。

  > **NOTE**
  > 请不要使用 `# Bytes in all Heaps` 性能计数器，因为它不表示托管堆的实际内存使用情况。代的大小包括在此值中，且实际上是其阈值大小，即如果代以对象进行填充，将引发垃圾回收的大小。因此，此值通常为零。

若要确定托管堆的内存保留量

- 使用 `# Total reserved bytes` 内存性能计数器。

  垃圾回收器在段中保留内存，你可以使用 **eeheap** 命令确定段的开始位置。

  > **IMPORTANT**
  > 尽管可以确定垃圾回收器为每个段分配的内存量，但是段的大小是特定于实现的，并可能会在任何时间(包括在定期更新中)进行更改。应用程序不应假设特定段的大小或依赖于此大小，也不应尝试配置段分配可用的内存量。

- 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令：

  !eeheap -gc

  结果如下所示：

```
Number of GC Heaps: 2
------------------------------
Heap 0 (002db550)
generation 0 starts at 0x02abe29c
generation 1 starts at 0x02abdd08
generation 2 starts at 0x02ab0038
ephemeral segment allocation context: none
  segment    begin allocated     size
02ab0000 02ab0038  02aceff4 0x0001efbc(126908)
Large object heap starts at 0x0aab0038
  segment    begin allocated     size
0aab0000 0aab0038  0aab2278 0x00002240(8768)
Heap Size   0x211fc(135676)
------------------------------
Heap 1 (002dc958)
generation 0 starts at 0x06ab1bd8
generation 1 starts at 0x06ab1bcc
generation 2 starts at 0x06ab0038
ephemeral segment allocation context: none
  segment    begin allocated     size
06ab0000 06ab0038  06ab3be4 0x00003bac(15276)
Large object heap starts at 0x0cab0038
  segment    begin allocated     size
0cab0000 0cab0038  0cab0048 0x00000010(16)
Heap Size    0x3bbc(15292)
------------------------------
GC Heap Size   0x24db8(150968)
```

由“段”指示的地址是段的起始地址。

若要确定第 2 代中的大型对象

- 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令：

  !dumpheap –stat

  如果托管堆很大，则 **dumpheap** 可能需要一段时间才能完成。

  你可以从输出的最后几行开始分析，因为它们列出了占用了大多数空间的对象。例如：

```
2c6108d4   173712     14591808 DevExpress.XtraGrid.Views.Grid.ViewInfo.GridCellInfo
00155f80      533     15216804      Free
7a747c78   791070     15821400 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700930     19626040 System.Collections.Specialized.ListDictionary
2c64e36c    78644     20762016 DevExpress.XtraEditors.ViewInfo.TextEditViewInfo
79124228   121143     29064120 System.Object[]
035f0ee4    81626     35588936 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
791242ec    40182     90664128 System.Collections.Hashtable+bucket[]
790fa3e0  3154024    137881448 System.String
Total 8454945 objects
```

  所列出的最后一个对象是一个字符串，且占用的空间最多。可以检查应用程序，以查看如何优化字符串对象。若要查看 150 到 200 个字节之间的字符串，请键入以下命令：

  !dumpheap -type System.String -min 150 -max 200

  如下所示是结果的一个示例。

```
Address  MT           Size  Gen
1875d2c0 790fa3e0      152    2 System.String HighlightNullStyle_Blotter_PendingOrder-
11_Blotter_PendingOrder-11
…
```

对 ID 使用整数而非字符串，这样可能会更有效。如果数千次重复相同的字符串，请考虑字符串暂留。有关字符串暂留的详细信息，请参阅 String.Intern 方法的参考主题。

若要确定对对象的引用

- 在加载了 SOS 调试器扩展的 WinDbg 中，键入以下命令，以列出对对象的引用：

  !gcroot

  `-or-`

- 若要确定对特定对象的引用，包括地址：

  !gcroot 1c37b2ac

  在堆栈上找到的根可能是误报。有关详细信息，请参阅命令 `!help gcroot`。

```
ebx:Root:19011c5c(System.Windows.Forms.Application+ThreadContext)->
19010b78(DemoApp.FormDemoApp)->
19011158(System.Windows.Forms.PropertyStore)->
… [omitted]
1c3745ec(System.Data.DataTable)->
1c3747a8(System.Data.DataColumnCollection)->
1c3747f8(System.Collections.Hashtable)->
1c376590(System.Collections.Hashtable+bucket[])->
1c376c98(System.Data.DataColumn)->
1c37b270(System.Data.Common.DoubleStorage)->
1c37b2ac(System.Double[])
Scan Thread 0 OSTHread 99c
Scan Thread 6 OSTHread 484
```

gcroot 命令可能需要很长时间才能完成。任何不通过垃圾回收进行回收的对象是活动对象。这意味着，某些根直接或间接地保留于该对象，因此 gcroot 应将路径信息返回到该对象。应检查返回的关系图，并查看仍然引用这些对象的原因。

若要确定是否已运行终结器

- 则运行包含以下代码的测试程序：

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

如果测试解决了此问题，这意味着垃圾回收器未回收对象，因为这些对象的终结器已被挂起。GC.WaitForPendingFinalizers 方法将启用这些终结器来完成其任务，并解决问题。

若要确定是否存在等待被终结的对象

1. 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令：

   !finalizequeue

   查看已准备好进行终结的对象的数目。如果数目很多，则必须检查这些终结器完全没有进展或进展速度不够快的原因。

2. 若要获取线程的输出，请键入以下命令：

   threads -special

   此命令提供如下所示的输出。

```
   OSID     Special thread type
2    cd0    DbgHelper
3    c18    Finalizer
4    df0    GC SuspendEE
```

   终结器线程将指示当前正在运行的终结器（如果存在）。当终结器线程没有运行任何终结器时，则它正在等待一个事件告诉它进行工作。大多数情况下，你将看到此状态中的终结器线程，因为它在 THREAD_HIGHEST_PRIORITY 处运行，并应快速完成运行终结器（如果存在）。

若要确定托管堆中的可用空间量

- 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令：

  !dumpheap -type Free -stat

  此命令将显示托管堆上所有可用对象的总大小，如以下示例中所示。

```
total 230 objects
Statistics:
      MT    Count    TotalSize Class Name
00152b18      230     40958584      Free
Total 230 objects
```

- 若要确定第 0 代中的可用空间，请键入以下命令以获取代的内存使用信息：

  !eeheap -gc

  该命令将显示类似以下所示的输出。最后一行将显示暂时段。

```
Heap 0 (0015ad08)
generation 0 starts at 0x49521f8c
generation 1 starts at 0x494d7f64
generation 2 starts at 0x007f0038
ephemeral segment allocation context: none
segment  begin     allocated  size
00178250 7a80d84c  7a82f1cc   0x00021980(137600)
00161918 78c50e40  78c7056c   0x0001f72c(128812)
007f0000 007f0038  047eed28   0x03ffecf0(67103984)
3a120000 3a120038  3a3e84f8   0x002c84c0(2917568)
46120000 46120038  49e05d04   0x03ce5ccc(63855820)
```

- 计算 0 代使用的空间：

  ?49e05d04-0x49521f8c

  结果如下所示：0 代大约为 9 MB。

```
Evaluate expression: 9321848 = 008e3d78
```

- 以下命令将转储 0 代范围内的可用空间：

  !dumpheap -type Free -stat 0x49521f8c 49e05d04

结果如下所示：

```
------------------------------
Heap 0
total 409 objects
------------------------------
Heap 1
total 0 objects
------------------------------
Heap 2
total 0 objects
------------------------------
Heap 3
total 0 objects
------------------------------
total 409 objects
Statistics:
      MT    Count TotalSize Class Name
0015a498      409   7296540      Free
Total 409 objects
```

此输出显示堆的 0 代部分正在对对象使用 9 MB 的空间并且有 7 MB 可用。此分析显示了 0 代对碎片的贡献程度。此堆的使用量应从总量中扣除，作为长期对象所产生的碎片的原因。

若要确定固定对象的数目

- 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令：

  !gchandles

  显示的统计信息包括固定句柄的数量，如以下示例所示。

```
GC Handle Statistics:
Strong Handles:      29
Pinned Handles:      10
```

若要确定垃圾回收中的时间

- 检查 `% Time in GC` 内存性能计数器。

  通过使用采样间隔时间来计算值。因为该计数器在每次垃圾回收结束时进行更新，所以如果在间隔期间没有产生任何回收，则当前的示例将具有与之前的示例相同的值。

  回收时间是通过将采样间隔时间乘以百分比值获取的。

  以下数据显示了为时 8 秒的研究的 4 个采样，彼此间隔 2 秒。`Gen0`、`Gen1` 和 `Gen2` 列显示了该代的间隔期间发生的垃圾回收数。

```
Interval    Gen0    Gen1    Gen2    % Time in GC
        1       9       3       1              10
        2      10       3       1               1
        3      11       3       1               3
        4      11       3       1               3
```

当垃圾回收发生时，将不会显示此信息，但可以确定时间间隔中发生的垃圾回收数。假设最坏的情况下，第 10 个 0 代垃圾回收在第 2 个间隔开始时完成，且第 11 个 0 代垃圾回收在第 5 个间隔结束时完成。第 10 个和第 11 个垃圾回收结束时之间的时间约为 2 秒钟，并且性能计数器显示为 3%，因此第 11 个 0 代垃圾回收的持续时间为(2 秒 * 3%= 60 毫秒)。

在此示例中，存在 5 个周期。

```
Interval    Gen0    Gen1    Gen2     % Time in GC
        1       9       3       1                3
        2      10       3       1                1
        3      11       4       2                1
        4      11       4       2                1
        5      11       4       2               20
```

第 2 个第 2代垃圾回收在第 3 个间隔期间开始并在第 5 个间隔处完成。假设最坏情况下，最后一次垃圾回收是针对在第 2 个间隔开始时完成的 0 代回收，且第 2 代垃圾回收在第 5 个间隔结束时完成。因此，第 0 代垃圾回收结束和第 2 代垃圾回收结束之间的时间是 4 秒。因为 `% Time in GC` 计数器为 20%，所以第 2 代垃圾回收可能使用的最长时间为(4 秒 * 20%= 800 毫秒)。

- 或者，可以通过使用垃圾回收 ETW 事件，确定垃圾回收的时长，并分析此信息以确定垃圾回收的持续时间。

  例如，以下数据显示了一个发生在非并发垃圾回收期间的事件序列。

```
Timestamp    Event name
513052        GCSuspendEEBegin_V1
513078        GCSuspendEEEnd
513090        GCStart_V1
517890        GCEnd_V1
517894        GCHeapStats
517897        GCRestartEEBegin
517918        GCRestartEEEnd
```

  挂起托管线程花费了 26us (`GCSuspendEEEnd` – `GCSuspendEEBegin_V1`)。

  实际的垃圾回收花费了 4.8 毫秒 (`GCEnd_V1` – `GCStart_V1`)。

  回复执行托管线程花费了 21us (`GCRestartEEEnd` – `GCRestartEEBegin`)。

  以下输出为后台垃圾回收提供了一个示例，并包括进程、线程和事件字段。(没有显示所有数据。)

```
timestamp(us)    event name           process    thread    event field
42504385        GCSuspendEEBegin_V1    Test.exe    4372            1
42504648        GCSuspendEEEnd         Test.exe    4372
42504816        GCStart_V1             Test.exe    4372       102019
42504907        GCStart_V1             Test.exe    4372       102020
42514170        GCEnd_V1               Test.exe    4372
42514204        GCHeapStats            Test.exe    4372       102020
42832052        GCRestartEEBegin       Test.exe    4372
42832136        GCRestartEEEnd         Test.exe    4372
63685394        GCSuspendEEBegin_V1    Test.exe    4744            6
63686347        GCSuspendEEEnd         Test.exe    4744
63784294        GCRestartEEBegin       Test.exe    4744
63784407        GCRestartEEEnd         Test.exe    4744
89931423        GCEnd_V1               Test.exe    4372       102019
89931464        GCHeapStats            Test.exe    4372
```

  42504816 处的 `GCStart_V1` 事件指示此为一个后台垃圾回收，因为最后一个字段是 `1` 。这将变为垃圾回收 No. 102019。

  将发生 `GCStart` 事件，因为在开始后台垃圾回收之前，需要一个暂时垃圾回收。这将变为垃圾回收 No. 102020。

  在 42514170 处，垃圾回收 No. 102020 结束。此时，将重新启动托管线程。这将在触发此后台垃圾回收的线程 4372 上完成。

在线程 4744 上，发生了一个挂起。这是唯一一次后台垃圾回收不得不挂起托管线程。此持续时间为大约 99 毫秒 ((63784407-63685394)/1000)。

后台垃圾回收的 `GCEnd` 事件位于 89931423。这意味着后台垃圾回收持续了大约 47 秒 ((89931423-42504816)/1000)。

托管线程运行时，可以查看发生的任意数量的暂时垃圾回收。

若要确定触发垃圾回收的原因

- 在加载了 SOS 调试器扩展的 WinDbg 或 Visual Studio 调试器中，键入以下命令，以显示所有带调用堆栈的线程：

  ~*kb

  该命令将显示类似以下所示的输出。

```
0012f3b0 79ff0bf8 mscorwks!WKS::GCHeap::GarbageCollect
0012f454 30002894 mscorwks!GCInterface::CollectGeneration+0xa4
0012f490 79fa22bd fragment_ni!request.Main(System.String[])+0x48
```

  如果垃圾回收是操作系统的内存不足通知引起的，则调用堆栈会非常相似，除了线程是终结器线程之外。终结器线程将获取异步内存不足的通知，并引发垃圾回收。

  如果垃圾回收是内存分配引起的，则堆栈显示如下：

```
0012f230 7a07c551 mscorwks!WKS::GCHeap::GarbageCollectGeneration
0012f2b8 7a07cba8 mscorwks!WKS::gc_heap::try_allocate_more_space+0x1a1
0012f2d4 7a07cefb mscorwks!WKS::gc_heap::allocate_more_space+0x18
0012f2f4 7a02a51b mscorwks!WKS::GCHeap::Alloc+0x4b
0012f310 7a02ae4c mscorwks!Alloc+0x60
0012f364 7a030e46 mscorwks!FastAllocatePrimitiveArray+0xbd
0012f424 300027f4 mscorwks!JIT_NewArr1+0x148
000af70f 3000299f fragment_ni!request..ctor(Int32, Single)+0x20c
0000002a 79fa22bd fragment_ni!request.Main(System.String[])+0x153
```

  实时帮助程序 (`JIT_New*`) 最终调用 `GCHeap::GarbageCollectGeneration`。如果确定第 2 代垃圾回收是分配引起的，则必须确定第 2 代垃圾回收所分配的对象以及如何避免它们。也就是说，想要确定第 2 代垃圾回收的开始和结束之间的差异，以及引发第 2 代回收的对象。

  例如，在调试器中键入以下命令，以显示第 2 代回收的开始：

  !dumpheap –stat

  输出示例（经过删减以显示使用的最多空间的对象）：

```
79124228    31857      9862328 System.Object[]
035f0384    25668     11601936 Toolkit.TlkPosition
00155f80    21248     12256296      Free
79103b6c   297003     13068132 System.Threading.ReaderWriterLock
7a747ad4   708732     14174640 System.Collections.Specialized.HybridDictionary
7a747c78   786498     15729960 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700298     19608344 System.Collections.Specialized.ListDictionary
035f0ee4    89192     38887712 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
7912c444    91616     71887080 System.Double[]
791242ec    32451     82462728 System.Collections.Hashtable+bucket[]
790fa3e0  2459154    112128436 System.String
Total 6471774 objects
```

在第 2 代结束时，重复该命令：

!dumpheap –stat

输出示例（经过删减以显示使用的最多空间的对象）：

```
79124228    26648      9314256 System.Object[]
035f0384    25668     11601936 Toolkit.TlkPosition
79103b6c   296770     13057880 System.Threading.ReaderWriterLock
7a747ad4   708730     14174600 System.Collections.Specialized.HybridDictionary
7a747c78   786497     15729940 System.Collections.Specialized.ListDictionary+DictionaryNode
7a747bac   700298     19608344 System.Collections.Specialized.ListDictionary
00155f80    13806     34007212      Free
035f0ee4    89187     38885532 Toolkit.TlkOrder
00fcae40     6193     44911636 WaveBasedStrategy.Tick_Snap[]
791242ec    32370     82359768 System.Collections.Hashtable+bucket[]
790fa3e0  2440020    111341808 System.String
Total 6417525 objects
```

`double[]` 对象从输出的末尾消失，这意味着它们被回收了。这些对象大约占 70 MB。剩余的对象没有太多变化。因此，这些 `double[]` 对象是第 2 代垃圾回收发生的原因。下一步是确定 `double[]` 对象存在以及他们最后死亡的原因。可以询问代码开发人员这些对象的来源，或使用 gcroot 命令。

若要确定 CPU 的使用率高是否是垃圾回收引起的

- 将 `% Time in GC` 内存性能计数器的值与处理时间相关联。

  如果 `% Time in GC` 值在与处理时间同时达到峰值，则垃圾回收将造成 CPU 使用率过高。否则，配置应用程序，以查找发生使用率过高的位置。

请参阅

- 垃圾回收

被动回收

2021/11/16 •

在大多数情况下，垃圾回收器可以确定执行回收的最佳时间，应让其独立运行。在某些不常见的情况下，强制回收可以提高应用程序的性能。在这种情况下，可以使用 GC.Collect 方法强制执行垃圾回收，从而诱导垃圾回收。

如果应用代码中特定点使用的内存量大量减少，请使用 GC.Collect 方法。例如，如果应用使用包含多个控件的复杂对话框，那么在对话框关闭时调用 Collect 可以立即回收对话框占用的内存，从而提升性能。请确保应用程序不会过于频繁地引发垃圾回收，否则当垃圾回收器无效率地尝试回收对象时，可能会使性能降低。可以向 Collect 方法提供 GCCollectionMode.Optimized 枚举值，以便仅在回收能够提高效率时才进行回收，如下一部分所述。

GC 回收模式

可以使用包含 GCCollectionMode 值的 GC.Collect 方法重载之一，指定强制回收的行为，如下所示。

`GCCOLLECTIONMODE` [	[[
Default	对正在运行的 .NET 版本使用默认的垃圾回收设置。
Forced	强制立即执行垃圾回收。这相当于调用 GC.Collect() 重载。它会导致对所有分代进行完全阻塞回收。 强制执行即时完全阻止式垃圾回收前，还可以将 GCSettings.LargeObjectHeapCompactionMode 属性设置为 GCLargeObjectHeapCompactionMode.CompactOnce，从而压缩大型对象堆。
Optimized	使垃圾回收器可以确定当前时间是否是回收对象的最佳时间。 垃圾回收器可能判定回收效率不够高，因此回收不合理，在这种情况下将返回而不回收对象。

后台回收或阻塞回收

可以调用 GC.Collect(Int32, GCCollectionMode, Boolean) 方法重载，指定诱导回收是否是阻止式。执行的回收类型取决于此方法的 `mode` 和 `blocking` 参数组合。`mode` 是 GCCollectionMode 枚举的成员，且 `blocking` 值为 Boolean。下表汇总了 `mode` 和 `blocking` 参数的交互。

`MODE`	`BLOCKING` = `TRUE`	`BLOCKING` = `FALSE`
Forced 或 Default	尽快执行阻塞回收。如果后台回收正在进行且分代为 0 或 1，Collect(Int32, GCCollectionMode, Boolean) 方法会立即触发阻止式回收，并在回收完成后返回结果。如果后台回收正在进行且 `generation` 参数为 2，此方法会等到后台回收完成，再触发第 2 代阻止式回收，然后返回结果。	尽快执行回收。Collect(Int32, GCCollectionMode, Boolean) 方法请求执行后台回收，但这并没有保证；阻止式回收仍可执行，具体视环境而定。如果后台回收正在进行，该方法将立即返回。

`MODE`	`BLOCKING` = `TRUE`	`BLOCKING` = `FALSE`
Optimized	可能会执行阻止式回收，具体视垃圾回收器的状态和 `generation` 参数而定。垃圾回收器会尽量提供最佳性能。	根据垃圾回收器的状态，有时可执行回收。Collect(Int32, GCCollectionMode, Boolean) 方法请求执行后台回收，但这并没有保证；阻止式回收仍可执行，具体视环境而定。垃圾回收器会尽量提供最佳性能。如果后台回收正在进行，该方法将立即返回。

另请参阅

- 延迟模式
- 垃圾回收

延迟模式

2021/11/16 •

若要回收对象，垃圾回收器 (GC) 必须停止应用程序中所有正在执行的线程。垃圾回收器处于活动状态的时间段称为延迟 。

在某些情况下（例如当应用程序检索数据或显示内容时），关键时刻可能发生完整的垃圾回收，从而妨碍性能。可以通过将 GCSettings.LatencyMode 属性设置为其中一个 System.Runtime.GCLatencyMode 值来调节垃圾回收的干扰。

低延迟设置

使用"低"延迟设置意味着垃圾回收器对应用程序的干扰较少。垃圾回收在回收内存方面较为保守。

System.Runtime.GCLatencyMode 枚举提供两种低延迟设置：

- GCLatencyMode.LowLatency 禁止第 2 代回收，仅执行第 0 代和第 1 代回收。只能在短时间内使用。在更长时间内，如果系统处于内存压力下，垃圾回收器将触发一次回收，这样会暂时暂停应用程序并中断对时间要求很急的操作。此设置仅对工作站垃圾回收可用。
- GCLatencyMode.SustainedLowLatency 禁止第 2 代前台回收，仅执行第 0 代、第 1 代回收和第 2 代后台回收。它可以长时间使用，并对工作站和服务器垃圾回收都可用。如果后台垃圾回收已禁用，则无法使用此设置。

在低延迟期间，除非发生以下情况，否则禁止第 2 代回收：

- 系统收到操作系统的低内存通知。
- 应用程序代码通过调用 GC.Collect 方法并将 `generation` 参数指定为 2 来包含回收。

方案

下表列出了使用 GCLatencyMode 值的应用程序方案：

tttt	tttttt
Batch	对于不具有用户界面 (UI) 或服务器端操作的应用程序。 禁用后台垃圾回收后，这将是工作站和服务器垃圾回收的默认模式。Batch 模式还会替代 gcConcurrent 设置，即它会阻止后台或并发回收。
Interactive	对于具有 UI 的大多数应用程序。 这是工作站和服务器垃圾回收的默认模式。但是，如果托管了某个应用，则优先考虑托管进程的垃圾回收器设置。
LowLatency	对于具有短期时效性操作（操作期间垃圾回收器的干扰可能会引起中断）的应用程序。例如，呈现动画或数据采集功能的应用程序。

[illegible]	[illegible]
SustainedLowLatency	适用于在有限但有可能更长的时间内具有时效性操作并且在此期间垃圾回收器中断具有破坏性的应用程序。例如，需要随着交易时间内的市场数据变化做出快速响应的应用程序。 此模式会比其他模式产生更大的托管堆大小。由于它不压缩托管堆，因此可能产生更多碎片。确保有足够的可用内存。

低延迟使用指南

使用 GCLatencyMode.LowLatency 模式时，请注意以下指导原则：

- 尽可能地缩短低延迟时段。
- 避免在低延迟时段分配大量内存。由于垃圾回收回收的对象较少可能出现低内存通知。
- 在低延迟模式下，最大限度减少新的分配次数，尤其是分配到大型对象堆和固定对象的次数。
- 知道可以分配的线程。由于 LatencyMode 属性设置属于进程范围的设置，因此可以在分配的任何线程上生成 OutOfMemoryException 异常。
- 将低延迟代码包装在受约束的执行区域中。有关详细信息，请参阅受约束的执行区域。
- 在低延迟期间，可以通过调用 GC.Collect(Int32, GCCollectionMode) 方法强制进行第 2 代回收。

另请参阅

- System.GC
- 已引发回收
- 垃圾回收

针对共享 Web 承载优化

2021/11/16 •

如果是通过托管多个小型网站进行共享的服务器的管理员，可以将下列 `gcTrimCommitOnLowMemory` 设置添加到 .NET 目录中 Aspnet.config 文件内的 `runtime` 节点，从而优化性能和增加网站容量：

```
<gcTrimCommitOnLowMemory enabled="true|false"/>
```

> **NOTE**
> 此建议设置仅适用于共享 Web 托管方案。

由于垃圾回收器保留内存以供将来分配，因此它提交的空间可能会超过真正所需。可以减少此空间来适应系统内存负载过重的情况。减少提交的此空间可提升性能，并将容量扩展为托管更多网站。

如果启用 `gcTrimCommitOnLowMemory` 设置，垃圾回收器会计算系统内存负载，并在负载达到 90% 时进入修整模式。除非负载下降到不到 85%，否则会一直处于修整模式。

如果条件允许，垃圾回收器可以决定 `gcTrimCommitOnLowMemory` 设置对当前应用没有帮助并忽略它。

示例

下面的 XML 片段展示了如何启用 `gcTrimCommitOnLowMemory` 设置。省略号表示 `runtime` 节点中会有其他设置。

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <runtime>
    . . .
    <gcTrimCommitOnLowMemory enabled="true"/>
    </runtime>
    . . .
</configuration>
```

另请参阅

- 垃圾回收

垃圾回收通知

2021/11/16 •

在有些情况下，公共语言运行时执行的完整垃圾回收（即第 2 代回收）可能会对性能产生负面影响。特别是，处理大量请求的服务器可能会出现此问题；在这种情况下，长时间垃圾回收会导致请求超时。为了防止在关键时期发生完全回收，可以接收即将执行完全垃圾回收的通知，再采取措施将工作负载重定向到另一个服务器实例。也可以自行诱导回收，前提是当前服务器实例不需要处理请求。

RegisterForFullGCNotification 方法注册为，在运行时检测到即将执行完全垃圾回收时发出通知。此通知分为两个部分：完全垃圾回收何时即将执行，以及完全垃圾回收何时完成。

> **WARNING**
> 只有阻止垃圾回收会引发通知。如果 <gcConcurrent> 配置元素已启用，后台垃圾回收不会发出通知。

若要确定何时发出通知，请使用 WaitForFullGCApproach 和 WaitForFullGCComplete 方法。通常，在 `while` 循环中使用这些方法，以持续获取表示通知状态的 GCNotificationStatus 枚举。如果值为 Succeeded，可以执行下列操作：

- 为了响应使用 WaitForFullGCApproach 方法获得的通知，可以重定向工作负载，并能自行诱导回收。
- 为了响应使用 WaitForFullGCComplete 方法获得的通知，可以让当前服务器实例再次用于处理请求。也可以收集信息。例如，可以使用 CollectionCount 方法记录回收次数。

WaitForFullGCApproach 和 WaitForFullGCComplete 方法要配合使用。使用一个方法，而不使用另一个方法，可能会生成不确定的结果。

完全垃圾回收

如果发生下列任一情况，运行时就会执行完全垃圾回收：

- 足够多的内存已提升到第 2 代，导致执行下一个第 2 代回收。
- 足够多的内存已提升到大型对象堆，导致执行下一个第 2 代回收。
- 由于其他因素，导致第 1 代回收升级为第 2 代回收。

在 RegisterForFullGCNotification 方法中指定的阈值适用于前两种情况。不过，在第一种情况下，不一定会在与指定的阈值相称的时间收到通知，原因有下面两个：

- 运行时不检查每个小型对象分配（出于性能考虑）。
- 只有第 1 代回收将内存提升到第 2 代。

第三种情况也加剧了通知接收时间的不确定性。可以在此期间重定向请求，或在可以更好适应时自行诱导回收，从而减轻不合时宜的完全垃圾回收造成的影响。尽管并不保证有效，但确实证明这是非常实用的方法。

通知阈值参数

RegisterForFullGCNotification 方法包含两个参数，用于指定第 2 代对象和大型对象堆的阈值。如果达到这些值，就应发出垃圾回收通知。下表介绍了这些参数。

“	“
`maxGenerationThreshold`	介于 1 和 99 之间的数字，指定根据在第 2 代中提升的对象，应何时发出通知。
`largeObjectHeapThreshold`	介于 1 和 99 之间的数字，指定根据大型对象堆中分配的对象，应何时发出通知。

如果指定的值过高，很可能会出现的情况是，将会收到通知，但在运行时执行回收前等待的时间太长。如果自行诱导回收，回收的对象可能会多于在运行时执行回收时回收的对象。

如果指定的值过低，在运行时执行回收前等待通知的时间可能不够长。

示例

描述

在下面的示例中，一组服务器处理传入的 Web 请求。为了模拟处理请求的工作负载，将字节数组添加到 List<T> 集合中。每个服务器都会注册获取垃圾回收通知，再对 `WaitForFullGCProc` 用户方法启动线程，以持续监视 WaitForFullGCApproach 和 WaitForFullGCComplete 方法返回的 GCNotificationStatus 枚举。

在通知发出时，WaitForFullGCApproach 和 WaitForFullGCComplete 方法调用它们各自的事件处理用户方法：

- `OnFullGCApproachNotify`

  此方法调用 `RedirectRequests` 用户方法，指示请求队列服务器暂停向服务器发送请求。具体模拟方式是，将类级别变量 `bAllocate` 设置为 `false`，这样就不会再分配对象。

  接下来，调用 `FinishExistingRequests` 用户方法，完成处理挂起的服务器请求。具体模拟方式是，清除 List<T> 集合。

  最后，由于工作负载很轻，诱导垃圾回收。

- `OnFullGCCompleteNotify`

  此方法调用用户方法 `AcceptRequests` 以继续接受请求，因为服务器不再易受完全垃圾回收影响。此操作的具体模拟方式是，将 `bAllocate` 变量设置为 `true`，以便能够继续将对象添加到 List<T> 集合。

下面的代码包含示例的 `Main` 方法。

```
using namespace System;
using namespace System::Collections::Generic;
using namespace System::Threading;

namespace GCNotify
{
    ref class Program
    {
    private:
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;

        // Variable for ending the example.
        static bool finalExit = false;
```

```
        // Collection for objects that
        // simulate the server request workload.
        static List<array<Byte>^>^ load = gcnew List<array<Byte>^>();


    public:
        static void Main()
        {
            try
            {
                // Register for a notification.
                GC::RegisterForFullGCNotification(10, 10);
                Console::WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;

                // Start a thread using WaitForFullGCProc.
                Thread^ thWaitForFullGC = gcnew Thread(gcnew ThreadStart(&WaitForFullGCProc));
                thWaitForFullGC->Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {
                    int lastCollCount = 0;
                    int newCollCount = 0;


                    while (true)
                    {
                        if (bAllocate)
                        {
                            load->Add(gcnew array<Byte>(1000));
                            newCollCount = GC::CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console::WriteLine("Gen 2 collection count: {0}",
GC::CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }

                }
                catch (OutOfMemoryException^)
                {
                    Console::WriteLine("Out of memory.");
                }


                finalExit = true;
                checkForNotify = false;
                GC::CancelFullGCNotification();

            }
            catch (InvalidOperationException^ invalidOp)
            {

                Console::WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
```

```
                + invalidOp->Message);
        }
    }

public:
    static void OnFullGCApproachNotify()
    {
        Console::WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC::Collect();
        Console::WriteLine("Induced a collection.");

    }


public:
    static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console::WriteLine("Accepting requests again.");
    }

public:
    static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC::WaitForFullGCApproach();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notifiction raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                s = GC::WaitForFullGCComplete();
```

```
                    s = GC::WaitForFullGCComplete();
                    if (s == GCNotificationStatus::Succeeded)
                    {
                        Console::WriteLine("GC Notification raised.");
                        OnFullGCCompleteEndNotify();
                    }
                    else if (s == GCNotificationStatus::Canceled)
                    {
                        Console::WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // Could be a time out.
                        Console::WriteLine("GC Notification not applicable.");
                        break;
                    }
                }


                Thread::Sleep(500);
                // FinalExit is set to true right before
                // the main thread cancelled notification.
                if (finalExit)
                {
                    break;
                }
            }
        }

    private:
        static void RedirectRequests()
        {
            // Code that sends requests
            // to other servers.

            // Suspend work.
            bAllocate = false;

        }

        static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load->Clear();

        }

        static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    };
}

int main()
{
    GCNotify::Program::Main();
}
```

```
public static void Main(string[] args)
{
    try
    {
        // Register for a notification.
        GC.RegisterForFullGCNotification(10, 10);
        Console.WriteLine("Registered for GC notification.");

        checkForNotify = true;
        bAllocate = true;

        // Start a thread using WaitForFullGCProc.
        Thread thWaitForFullGC = new Thread(new ThreadStart(WaitForFullGCProc));
        thWaitForFullGC.Start();

        // While the thread is checking for notifications in
        // WaitForFullGCProc, create objects to simulate a server workload.
        try
        {

            int lastCollCount = 0;
            int newCollCount = 0;

            while (true)
            {
                if (bAllocate)
                {
                    load.Add(new byte[1000]);
                    newCollCount = GC.CollectionCount(2);
                    if (newCollCount != lastCollCount)
                    {
                        // Show collection count when it increases:
                        Console.WriteLine("Gen 2 collection count: {0}", GC.CollectionCount(2).ToString());
                        lastCollCount = newCollCount;
                    }

                    // For ending the example (arbitrary).
                    if (newCollCount == 500)
                    {
                        finalExit = true;
                        checkForNotify = false;
                        break;
                    }
                }
            }
        }
        catch (OutOfMemoryException)
        {
            Console.WriteLine("Out of memory.");
        }

        finalExit = true;
        checkForNotify = false;
        GC.CancelFullGCNotification();
    }
    catch (InvalidOperationException invalidOp)
    {

        Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
            + invalidOp.Message);
    }
}
```

```
Imports System.Collections.Generic
Imports System.Threading
```

```
Class Program
    ' Variables for continual checking in the
    ' While loop in the WaitForFullGcProc method.
    Private Shared checkForNotify As Boolean = False

    ' Variable for suspending work
    ' (such as servicing allocated server requests)
    ' after a notification is received and then
    ' resuming allocation after inducing a garbage collection.
    Private Shared bAllocate As Boolean = False

    ' Variable for ending the example.
    Private Shared finalExit As Boolean = False

    ' Collection for objects that
    ' simulate the server request workload.
    Private Shared load As New List(Of Byte())


    Public Shared Sub Main(ByVal args() As String)
        Try
            ' Register for a notification.
            GC.RegisterForFullGCNotification(10, 10)
            Console.WriteLine("Registered for GC notification.")

            bAllocate = True
            checkForNotify = True

            ' Start a thread using WaitForFullGCProc.
            Dim thWaitForFullGC As Thread = _
                New Thread(New ThreadStart(AddressOf WaitForFullGCProc))
            thWaitForFullGC.Start()

            ' While the thread is checking for notifications in
            ' WaitForFullGCProc, create objects to simulate a server workload.
            Try
                Dim lastCollCount As Integer = 0
                Dim newCollCount As Integer = 0


                While (True)
                    If bAllocate = True Then

                        load.Add(New Byte(1000) {})
                        newCollCount = GC.CollectionCount(2)
                        If (newCollCount <> lastCollCount) Then
                            ' Show collection count when it increases:
                            Console.WriteLine("Gen 2 collection count: {0}", _
                                              GC.CollectionCount(2).ToString)
                            lastCollCount = newCollCount
                        End If

                        ' For ending the example (arbitrary).
                        If newCollCount = 500 Then
                            finalExit = True
                            checkForNotify = False
                            bAllocate = False
                            Exit While
                        End If

                    End If
                End While

            Catch outofMem As OutOfMemoryException
                Console.WriteLine("Out of memory.")
            End Try

            finalExit = True
            checkForNotify = False
```

```
            GC.CancelFullGCNotification()

        Catch invalidOp As InvalidOperationException
            Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled." _
                              & vbLf & invalidOp.Message)
        End Try
    End Sub

    Public Shared Sub OnFullGCApproachNotify()
        Console.WriteLine("Redirecting requests.")

        ' Method that tells the request queuing
        ' server to not direct requests to this server.
        RedirectRequests()

        ' Method that provides time to
        ' finish processing pending requests.
        FinishExistingRequests()

        ' This is a good time to induce a GC collection
        ' because the runtime will induce a ful GC soon.
        ' To be very careful, you can check precede with a
        ' check of the GC.GCCollectionCount to make sure
        ' a full GC did not already occur since last notified.
        GC.Collect()
        Console.WriteLine("Induced a collection.")
    End Sub

    Public Shared Sub OnFullGCCompleteEndNotify()
        ' Method that informs the request queuing server
        ' that this server is ready to accept requests again.
        AcceptRequests()
        Console.WriteLine("Accepting requests again.")
    End Sub

    Public Shared Sub WaitForFullGCProc()

        While True
            ' CheckForNotify is set to true and false in Main.

            While checkForNotify
                ' Check for a notification of an approaching collection.
                Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notification raised.")
                    OnFullGCApproachNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' This can occur if a timeout period
                    ' is specified for WaitForFullGCApproach(Timeout)
                    ' or WaitForFullGCComplete(Timeout)
                    ' and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

                ' Check for a notification of a completed collection.
                s = GC.WaitForFullGCComplete
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notifiction raised.")
                    OnFullGCCompleteEndNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' Could be a time out.
                    Console.WriteLine("GC Notification not applicable.")
```

```
                    Exit While
                End If

            End While
            Thread.Sleep(500)
            ' FinalExit is set to true right before
            ' the main thread cancelled notification.
            If finalExit Then
                Exit While
            End If

        End While
    End Sub

    Private Shared Sub RedirectRequests()
        ' Code that sends requests
        ' to other servers.

        ' Suspend work.
        bAllocate = False
    End Sub

    Private Shared Sub FinishExistingRequests()
        ' Code that waits a period of time
        ' for pending requests to finish.

        ' Clear the simulated workload.
        load.Clear()

    End Sub

    Private Shared Sub AcceptRequests()
        ' Code that resumes processing
        ' requests on this server.

        ' Resume work.
        bAllocate = True
    End Sub
End Class
```

下面的代码包含 `WaitForFullGCProc` 用户方法，其中包括持续 while 循环，用于检查是否有垃圾回收通知。

```
public:
    static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC::WaitForFullGCApproach();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notifiction raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                s = GC::WaitForFullGCComplete();
                if (s == GCNotificationStatus::Succeeded)
                {
                    Console::WriteLine("GC Notification raised.");
                    OnFullGCCompleteEndNotify();
                }
                else if (s == GCNotificationStatus::Canceled)
                {
                    Console::WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // Could be a time out.
                    Console::WriteLine("GC Notification not applicable.");
                    break;
                }
            }


            Thread::Sleep(500);
            // FinalExit is set to true right before
            // the main thread cancelled notification.
            if (finalExit)
            {
                break;
            }
        }
    }
```

```
public static void WaitForFullGCProc()
{
    while (true)
    {
        // CheckForNotify is set to true and false in Main.
        while (checkForNotify)
        {
            // Check for a notification of an approaching collection.
            GCNotificationStatus s = GC.WaitForFullGCApproach();
            if (s == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCApproachNotify();
            }
            else if (s == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // This can occur if a timeout period
                // is specified for WaitForFullGCApproach(Timeout)
                // or WaitForFullGCComplete(Timeout)
                // and the time out period has elapsed.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }

            // Check for a notification of a completed collection.
            GCNotificationStatus status = GC.WaitForFullGCComplete();
            if (status == GCNotificationStatus.Succeeded)
            {
                Console.WriteLine("GC Notification raised.");
                OnFullGCCompleteEndNotify();
            }
            else if (status == GCNotificationStatus.Canceled)
            {
                Console.WriteLine("GC Notification cancelled.");
                break;
            }
            else
            {
                // Could be a time out.
                Console.WriteLine("GC Notification not applicable.");
                break;
            }
        }

        Thread.Sleep(500);
        // FinalExit is set to true right before
        // the main thread cancelled notification.
        if (finalExit)
        {
            break;
        }
    }
}
```

```
Public Shared Sub WaitForFullGCProc()

    While True
        ' CheckForNotify is set to true and false in Main.

        While checkForNotify
            ' Check for a notification of an approaching collection.
            Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
            If (s = GCNotificationStatus.Succeeded) Then
                Console.WriteLine("GC Notification raised.")
                OnFullGCApproachNotify()
            ElseIf (s = GCNotificationStatus.Canceled) Then
                Console.WriteLine("GC Notification cancelled.")
                Exit While
            Else
                ' This can occur if a timeout period
                ' is specified for WaitForFullGCApproach(Timeout)
                ' or WaitForFullGCComplete(Timeout)
                ' and the time out period has elapsed.
                Console.WriteLine("GC Notification not applicable.")
                Exit While
            End If

            ' Check for a notification of a completed collection.
            s = GC.WaitForFullGCComplete
            If (s = GCNotificationStatus.Succeeded) Then
                Console.WriteLine("GC Notifiction raised.")
                OnFullGCCompleteEndNotify()
            ElseIf (s = GCNotificationStatus.Canceled) Then
                Console.WriteLine("GC Notification cancelled.")
                Exit While
            Else
                ' Could be a time out.
                Console.WriteLine("GC Notification not applicable.")
                Exit While
            End If

        End While
        Thread.Sleep(500)
        ' FinalExit is set to true right before
        ' the main thread cancelled notification.
        If finalExit Then
            Exit While
        End If

    End While
End Sub
```

下面的代码包含 `OnFullGCApproachNotify` 方法，调用自

`WaitForFullGCProc` 方法。

```
public:
    static void OnFullGCApproachNotify()
    {
        Console::WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC::Collect();
        Console::WriteLine("Induced a collection.");

    }
```

```
public static void OnFullGCApproachNotify()
{

    Console.WriteLine("Redirecting requests.");

    // Method that tells the request queuing
    // server to not direct requests to this server.
    RedirectRequests();

    // Method that provides time to
    // finish processing pending requests.
    FinishExistingRequests();

    // This is a good time to induce a GC collection
    // because the runtime will induce a full GC soon.
    // To be very careful, you can check precede with a
    // check of the GC.GCCollectionCount to make sure
    // a full GC did not already occur since last notified.
    GC.Collect();
    Console.WriteLine("Induced a collection.");
}
```

```
Public Shared Sub OnFullGCApproachNotify()
    Console.WriteLine("Redirecting requests.")

    ' Method that tells the request queuing
    ' server to not direct requests to this server.
    RedirectRequests()

    ' Method that provides time to
    ' finish processing pending requests.
    FinishExistingRequests()

    ' This is a good time to induce a GC collection
    ' because the runtime will induce a ful GC soon.
    ' To be very careful, you can check precede with a
    ' check of the GC.GCCollectionCount to make sure
    ' a full GC did not already occur since last notified.
    GC.Collect()
    Console.WriteLine("Induced a collection.")
End Sub
```

下面的代码包含 `OnFullGCApproachComplete` 方法，调用自 `WaitForFullGCProc` 方法。

```
public:
    static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console::WriteLine("Accepting requests again.");
    }
```

```
public static void OnFullGCCompleteEndNotify()
{
    // Method that informs the request queuing server
    // that this server is ready to accept requests again.
    AcceptRequests();
    Console.WriteLine("Accepting requests again.");
}
```

```
Public Shared Sub OnFullGCCompleteEndNotify()
    ' Method that informs the request queuing server
    ' that this server is ready to accept requests again.
    AcceptRequests()
    Console.WriteLine("Accepting requests again.")
End Sub
```

下面的代码包含调用自 `OnFullGCApproachNotify` 和 `OnFullGCCompleteNotify` 方法的用户方法。用户方法重定向请求，完成现有请求，再在发生完全垃圾回收后继续执行请求。

```
private:
    static void RedirectRequests()
    {
        // Code that sends requests
        // to other servers.

        // Suspend work.
        bAllocate = false;

    }

    static void FinishExistingRequests()
    {
        // Code that waits a period of time
        // for pending requests to finish.

        // Clear the simulated workload.
        load->Clear();

    }

    static void AcceptRequests()
    {
        // Code that resumes processing
        // requests on this server.

        // Resume work.
        bAllocate = true;
    }
```

```
private static void RedirectRequests()
{
    // Code that sends requests
    // to other servers.

    // Suspend work.
    bAllocate = false;
}

private static void FinishExistingRequests()
{
    // Code that waits a period of time
    // for pending requests to finish.

    // Clear the simulated workload.
    load.Clear();
}

private static void AcceptRequests()
{
    // Code that resumes processing
    // requests on this server.

    // Resume work.
    bAllocate = true;
}
```

```
Private Shared Sub RedirectRequests()
    ' Code that sends requests
    ' to other servers.

    ' Suspend work.
    bAllocate = False
End Sub

Private Shared Sub FinishExistingRequests()
    ' Code that waits a period of time
    ' for pending requests to finish.

    ' Clear the simulated workload.
    load.Clear()

End Sub

Private Shared Sub AcceptRequests()
    ' Code that resumes processing
    ' requests on this server.

    ' Resume work.
    bAllocate = True
End Sub
```

整个代码示例如下所示：

```
using namespace System;
using namespace System::Collections::Generic;
using namespace System::Threading;

namespace GCNotify
{
    ref class Program
    {
    private:
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;

        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<array<Byte>^>^ load = gcnew List<array<Byte>^>();


    public:
        static void Main()
        {
            try
            {
                // Register for a notification.
                GC::RegisterForFullGCNotification(10, 10);
                Console::WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;
```

```
                // Start a thread using WaitForFullGCProc.
                Thread^ thWaitForFullGC = gcnew Thread(gcnew ThreadStart(&WaitForFullGCProc));
                thWaitForFullGC->Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {
                    int lastCollCount = 0;
                    int newCollCount = 0;


                    while (true)
                    {
                        if (bAllocate)
                        {
                            load->Add(gcnew array<Byte>(1000));
                            newCollCount = GC::CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console::WriteLine("Gen 2 collection count: {0}",
GC::CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }

                }
                catch (OutOfMemoryException^)
                {
                    Console::WriteLine("Out of memory.");
                }


                finalExit = true;
                checkForNotify = false;
                GC::CancelFullGCNotification();

            }
            catch (InvalidOperationException^ invalidOp)
            {

                Console::WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                    + invalidOp->Message);
            }
        }

    public:
        static void OnFullGCApproachNotify()
        {
            Console::WriteLine("Redirecting requests.");

            // Method that tells the request queuing
            // server to not direct requests to this server.
            RedirectRequests();

            // Method that provides time to
            // finish processing pending requests.
            FinishExistingRequests();
```

```
            // This is a good time to induce a GC collection
            // because the runtime will induce a full GC soon.
            // To be very careful, you can check precede with a
            // check of the GC.GCCollectionCount to make sure
            // a full GC did not already occur since last notified.
            GC::Collect();
            Console::WriteLine("Induced a collection.");

        }


    public:
        static void OnFullGCCompleteEndNotify()
        {
            // Method that informs the request queuing server
            // that this server is ready to accept requests again.
            AcceptRequests();
            Console::WriteLine("Accepting requests again.");
        }

    public:
        static void WaitForFullGCProc()
        {
            while (true)
            {
                // CheckForNotify is set to true and false in Main.
                while (checkForNotify)
                {
                    // Check for a notification of an approaching collection.
                    GCNotificationStatus s = GC::WaitForFullGCApproach();
                    if (s == GCNotificationStatus::Succeeded)
                    {
                        Console::WriteLine("GC Notifiction raised.");
                        OnFullGCApproachNotify();
                    }
                    else if (s == GCNotificationStatus::Canceled)
                    {
                        Console::WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // This can occur if a timeout period
                        // is specified for WaitForFullGCApproach(Timeout)
                        // or WaitForFullGCComplete(Timeout)
                        // and the time out period has elapsed.
                        Console::WriteLine("GC Notification not applicable.");
                        break;
                    }

                    // Check for a notification of a completed collection.
                    s = GC::WaitForFullGCComplete();
                    if (s == GCNotificationStatus::Succeeded)
                    {
                        Console::WriteLine("GC Notification raised.");
                        OnFullGCCompleteEndNotify();
                    }
                    else if (s == GCNotificationStatus::Canceled)
                    {
                        Console::WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // Could be a time out.
                        Console::WriteLine("GC Notification not applicable.");
                        break;
                    }
                }
```

```
                }


                Thread::Sleep(500);
                // FinalExit is set to true right before
                // the main thread cancelled notification.
                if (finalExit)
                {
                    break;
                }
            }
        }

    private:
        static void RedirectRequests()
        {
            // Code that sends requests
            // to other servers.

            // Suspend work.
            bAllocate = false;

        }

        static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load->Clear();

        }

        static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    };
}

int main()
{
    GCNotify::Program::Main();
}
```

```
using System;
using System.Collections.Generic;
using System.Threading;

namespace GCNotify
{
    class Program
    {
        // Variable for continual checking in the
        // While loop in the WaitForFullGCProc method.
        static bool checkForNotify = false;

        // Variable for suspending work
        // (such servicing allocated server requests)
        // after a notification is received and then
        // resuming allocation after inducing a garbage collection.
        static bool bAllocate = false;
```

```
        // Variable for ending the example.
        static bool finalExit = false;

        // Collection for objects that
        // simulate the server request workload.
        static List<byte[]> load = new List<byte[]>();

        public static void Main(string[] args)
        {
            try
            {
                // Register for a notification.
                GC.RegisterForFullGCNotification(10, 10);
                Console.WriteLine("Registered for GC notification.");

                checkForNotify = true;
                bAllocate = true;

                // Start a thread using WaitForFullGCProc.
                Thread thWaitForFullGC = new Thread(new ThreadStart(WaitForFullGCProc));
                thWaitForFullGC.Start();

                // While the thread is checking for notifications in
                // WaitForFullGCProc, create objects to simulate a server workload.
                try
                {

                    int lastCollCount = 0;
                    int newCollCount = 0;

                    while (true)
                    {
                        if (bAllocate)
                        {
                            load.Add(new byte[1000]);
                            newCollCount = GC.CollectionCount(2);
                            if (newCollCount != lastCollCount)
                            {
                                // Show collection count when it increases:
                                Console.WriteLine("Gen 2 collection count: {0}",
GC.CollectionCount(2).ToString());
                                lastCollCount = newCollCount;
                            }

                            // For ending the example (arbitrary).
                            if (newCollCount == 500)
                            {
                                finalExit = true;
                                checkForNotify = false;
                                break;
                            }
                        }
                    }
                }
                catch (OutOfMemoryException)
                {
                    Console.WriteLine("Out of memory.");
                }

                finalExit = true;
                checkForNotify = false;
                GC.CancelFullGCNotification();
            }
            catch (InvalidOperationException invalidOp)
            {

                Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled.\n"
                    + invalidOp.Message);
            }
```

```
    }

    public static void OnFullGCApproachNotify()
    {

        Console.WriteLine("Redirecting requests.");

        // Method that tells the request queuing
        // server to not direct requests to this server.
        RedirectRequests();

        // Method that provides time to
        // finish processing pending requests.
        FinishExistingRequests();

        // This is a good time to induce a GC collection
        // because the runtime will induce a full GC soon.
        // To be very careful, you can check precede with a
        // check of the GC.GCCollectionCount to make sure
        // a full GC did not already occur since last notified.
        GC.Collect();
        Console.WriteLine("Induced a collection.");
    }

    public static void OnFullGCCompleteEndNotify()
    {
        // Method that informs the request queuing server
        // that this server is ready to accept requests again.
        AcceptRequests();
        Console.WriteLine("Accepting requests again.");
    }

    public static void WaitForFullGCProc()
    {
        while (true)
        {
            // CheckForNotify is set to true and false in Main.
            while (checkForNotify)
            {
                // Check for a notification of an approaching collection.
                GCNotificationStatus s = GC.WaitForFullGCApproach();
                if (s == GCNotificationStatus.Succeeded)
                {
                    Console.WriteLine("GC Notification raised.");
                    OnFullGCApproachNotify();
                }
                else if (s == GCNotificationStatus.Canceled)
                {
                    Console.WriteLine("GC Notification cancelled.");
                    break;
                }
                else
                {
                    // This can occur if a timeout period
                    // is specified for WaitForFullGCApproach(Timeout)
                    // or WaitForFullGCComplete(Timeout)
                    // and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.");
                    break;
                }

                // Check for a notification of a completed collection.
                GCNotificationStatus status = GC.WaitForFullGCComplete();
                if (status == GCNotificationStatus.Succeeded)
                {
                    Console.WriteLine("GC Notification raised.");
                    OnFullGCCompleteEndNotify();
                }
                else if (status == GCNotificationStatus.Canceled)
```

```
                    else if (status == GCNotificationStatus.Canceled)
                    {
                        Console.WriteLine("GC Notification cancelled.");
                        break;
                    }
                    else
                    {
                        // Could be a time out.
                        Console.WriteLine("GC Notification not applicable.");
                        break;
                    }
                }

                Thread.Sleep(500);
                // FinalExit is set to true right before
                // the main thread cancelled notification.
                if (finalExit)
                {
                    break;
                }
            }
        }

        private static void RedirectRequests()
        {
            // Code that sends requests
            // to other servers.

            // Suspend work.
            bAllocate = false;
        }

        private static void FinishExistingRequests()
        {
            // Code that waits a period of time
            // for pending requests to finish.

            // Clear the simulated workload.
            load.Clear();
        }

        private static void AcceptRequests()
        {
            // Code that resumes processing
            // requests on this server.

            // Resume work.
            bAllocate = true;
        }
    }
}
```

```
Imports System.Collections.Generic
Imports System.Threading

Class Program
    ' Variables for continual checking in the
    ' While loop in the WaitForFullGcProc method.
    Private Shared checkForNotify As Boolean = False

    ' Variable for suspending work
    ' (such as servicing allocated server requests)
    ' after a notification is received and then
    ' resuming allocation after inducing a garbage collection.
    Private Shared bAllocate As Boolean = False

    ' Variable for ending the example.
    Private Shared finalExit As Boolean = False
```

```
Private Shared finalExit As Boolean = False

' Collection for objects that
' simulate the server request workload.
Private Shared load As New List(Of Byte())


Public Shared Sub Main(ByVal args() As String)
    Try
        ' Register for a notification.
        GC.RegisterForFullGCNotification(10, 10)
        Console.WriteLine("Registered for GC notification.")

        bAllocate = True
        checkForNotify = True

        ' Start a thread using WaitForFullGCProc.
        Dim thWaitForFullGC As Thread = _
            New Thread(New ThreadStart(AddressOf WaitForFullGCProc))
        thWaitForFullGC.Start()

        ' While the thread is checking for notifications in
        ' WaitForFullGCProc, create objects to simulate a server workload.
        Try
            Dim lastCollCount As Integer = 0
            Dim newCollCount As Integer = 0


            While (True)
                If bAllocate = True Then

                    load.Add(New Byte(1000) {})
                    newCollCount = GC.CollectionCount(2)
                    If (newCollCount <> lastCollCount) Then
                        ' Show collection count when it increases:
                        Console.WriteLine("Gen 2 collection count: {0}", _
                                          GC.CollectionCount(2).ToString)
                        lastCollCount = newCollCount
                    End If

                    ' For ending the example (arbitrary).
                    If newCollCount = 500 Then
                        finalExit = True
                        checkForNotify = False
                        bAllocate = False
                        Exit While
                    End If

                End If
            End While

        Catch outofMem As OutOfMemoryException
            Console.WriteLine("Out of memory.")
        End Try

        finalExit = True
        checkForNotify = False
        GC.CancelFullGCNotification()

    Catch invalidOp As InvalidOperationException
        Console.WriteLine("GC Notifications are not supported while concurrent GC is enabled." _
                          & vbLf & invalidOp.Message)
    End Try
End Sub

Public Shared Sub OnFullGCApproachNotify()
    Console.WriteLine("Redirecting requests.")

    ' Method that tells the request queuing
    ' server to not direct requests to this server.
```

```
        ' server to not direct requests to this server.
        RedirectRequests()

        ' Method that provides time to
        ' finish processing pending requests.
        FinishExistingRequests()

        ' This is a good time to induce a GC collection
        ' because the runtime will induce a ful GC soon.
        ' To be very careful, you can check precede with a
        ' check of the GC.GCCollectionCount to make sure
        ' a full GC did not already occur since last notified.
        GC.Collect()
        Console.WriteLine("Induced a collection.")
    End Sub

    Public Shared Sub OnFullGCCompleteEndNotify()
        ' Method that informs the request queuing server
        ' that this server is ready to accept requests again.
        AcceptRequests()
        Console.WriteLine("Accepting requests again.")
    End Sub

    Public Shared Sub WaitForFullGCProc()

        While True
            ' CheckForNotify is set to true and false in Main.

            While checkForNotify
                ' Check for a notification of an approaching collection.
                Dim s As GCNotificationStatus = GC.WaitForFullGCApproach
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notification raised.")
                    OnFullGCApproachNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' This can occur if a timeout period
                    ' is specified for WaitForFullGCApproach(Timeout)
                    ' or WaitForFullGCComplete(Timeout)
                    ' and the time out period has elapsed.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

                ' Check for a notification of a completed collection.
                s = GC.WaitForFullGCComplete
                If (s = GCNotificationStatus.Succeeded) Then
                    Console.WriteLine("GC Notifiction raised.")
                    OnFullGCCompleteEndNotify()
                ElseIf (s = GCNotificationStatus.Canceled) Then
                    Console.WriteLine("GC Notification cancelled.")
                    Exit While
                Else
                    ' Could be a time out.
                    Console.WriteLine("GC Notification not applicable.")
                    Exit While
                End If

            End While
            Thread.Sleep(500)
            ' FinalExit is set to true right before
            ' the main thread cancelled notification.
            If finalExit Then
                Exit While
            End If

        End While
```

```
    End Sub

    Private Shared Sub RedirectRequests()
        ' Code that sends requests
        ' to other servers.

        ' Suspend work.
        bAllocate = False
    End Sub

    Private Shared Sub FinishExistingRequests()
        ' Code that waits a period of time
        ' for pending requests to finish.

        ' Clear the simulated workload.
        load.Clear()

    End Sub

    Private Shared Sub AcceptRequests()
        ' Code that resumes processing
        ' requests on this server.

        ' Resume work.
        bAllocate = True
    End Sub
End Class
```

请参阅

- 垃圾回收

应用程序域资源监控

2021/11/16 •

借助应用域资源监视 (ARM)，主机可以通过应用域监视 CPU 和内存使用情况。对于在长时间运行的进程中使用多个应用域的 ASP.NET 等主机，这就很有用。主机可以卸载对整个进程的性能有不利影响的应用的应用域，但仅当能够发现有问题的应用时，才可以这样做。ARM 提供的信息就有助于作出此类决定。

例如，托管服务可能有多个应用在 ASP.NET 服务器上运行。如果进程中的一个应用开始占用太多内存或过长的处理器时间，托管服务就可以使用 ARM 发现导致问题发生的应用域。

ARM 是轻型服务，足可用于实际应用。若要访问信息，可以使用 Windows 事件跟踪 (ETW)，或直接使用托管或本机 API。

启用资源监视

可以通过下列四种方法启用 ARM：在公共语言运行时 (CLR) 启动时提供配置文件、使用非托管宿主 API、使用托管代码或侦听 ARM ETW 事件。

一旦启用，ARM 就会开始收集进程中所有应用域的相关数据。如果在启用 ARM 前就创建了应用域，累积数据会在 ARM 启用时启动，而不是在应用域创建时启动。一旦启用，ARM 便无法再禁用。

- 可以在 CLR 启动时启用 ARM，具体操作是向配置文件添加 <appDomainResourceMonitoring> 元素，并将 `enabled` 属性设置为 `true`。值 `false`（默认值）只表示不在启动时启用 ARM；稍后可以使用其他激活机制之一来激活它。
- 主机可以请求获取 ICLRAppDomainResourceMonitor 托管接口来启用 ARM。成功获取此接口后，就会启用 ARM。
- 托管代码可以将静态 AppDomain.MonitoringIsEnabled 属性（Visual Basic 中的 `Shared`）设置为 `true`，从而启用 ARM。设置此属性后，就会启用 ARM。
- 启动后，可以通过侦听 ETW 事件来启用 ARM。使用 `AppDomainResourceManagementKeyword` 关键字启用公共提供程序 `Microsoft-Windows-DotNETRuntime` 后，ARM 便会启用，并开始抛出所有应用域的事件。若要将数据与应用域及线程相关联，还必须使用 `ThreadingKeyword` 关键字启用 `Microsoft-Windows-DotNETRuntimeRundown` 提供程序。

使用 ARM

ARM 提供应用域使用的总处理器时间，以及关于内存使用情况的三种信息。

- **应用域使用的总处理器时间（以秒为单位）**：此时间的计算方式为，将操作系统报告的线程时间相加得出，包括在生存期内在应用域中执行的所有线程。受阻止或处于睡眠状态的线程不使用处理器时间。如果线程调用本机代码，线程在本机代码中花费的时间计入执行调用的应用域的总处理器时间。
  - 托管 API：AppDomain.MonitoringTotalProcessorTime 属性。
  - 宿主 API：ICLRAppDomainResourceMonitor::GetCurrentCpuTime 方法。
  - ETW 事件：`ThreadCreated`、`ThreadAppDomainEnter` 和 `ThreadTerminated` 事件。若要了解提供程序和关键字，请参阅 CLR ETW 事件 中的“应用域资源监视事件”。
- **应用程序域在其生命周期内进行的托管分配总量（以字节为单位）**：总分配并不总是反映应用程序域的内存使用情况，因为所分配的对象可能是短期的。不过，如果应用分配并释放大量对象，分配成本可能会非常高。

- 托管 API：AppDomain.MonitoringTotalAllocatedMemorySize 属性。
- 宿主 API：ICLRAppDomainResourceMonitor::GetCurrentAllocated 方法。
- ETW 事件：`AppDomainMemAllocated` 事件、`Allocated` 字段。

- **应用域引用且在最新执行的完全阻止式回收后保留的托管内存（以字节为单位）**：此数字只有在执行完全阻止式回收后才准确。（这与并发回收相反，后者发生在后台，不会阻止应用。）例如，GC.Collect() 方法重载导致执行完全阻止式回收。
  - 托管 API：AppDomain.MonitoringSurvivedMemorySize 属性。
  - 宿主 API：ICLRAppDomainResourceMonitor::GetCurrentSurvived 方法、`pAppDomainBytesSurvived` 参数。
  - ETW 事件：`AppDomainMemSurvived` 事件、`Survived` 字段。
- **进程引用且在最新执行的完全阻止式回收后保留的托管内存总量（以字节为单位）**：可将为单个应用程序域保留的内存与此数字进行比较。
  - 托管 API：AppDomain.MonitoringSurvivedProcessMemorySize 属性。
  - 宿主 API：ICLRAppDomainResourceMonitor::GetCurrentSurvived 方法、`pTotalBytesSurvived` 参数。
  - ETW 事件：`AppDomainMemSurvived` 事件、`ProcessSurvived` 字段。

确定何时发生完全阻止式回收

若要确定何时保留的内存计数是准确的，只需知道何时发生了完全阻止式回收即可。执行此操作的方法取决于用来检查 ARM 统计信息的 API。

托管 API

如果使用 AppDomain 类的属性，可以使用 GC.RegisterForFullGCNotification 方法来注册获取完全回收的通知。使用的阈值并不重要，因为正在等待回收完成，而不是回收的方法完成。然后，可以调用 GC.WaitForFullGCComplete 方法，一直阻止到完全回收完成。可以创建线程，用于在循环中调用此方法，并在此方法返回结果时执行任何所需的分析。

也可以定期调用 GC.CollectionCount 方法，以确定第 2 代回收计数是否已增加。此方法可能无法准确指明完全回收的发生，具体视轮询频率而定。

宿主 API

如果使用非托管宿主 API，主机必须向 CLR 传递 IHostGCManager 接口实现。如果 CLR 恢复执行在回收发生时被暂停的线程，便会调用 IHostGCManager::SuspensionEnding 方法。CLR 将生成的已完成回收作为方法参数进行传递，以便主机能够确定回收是完全回收还是部分回收。实现 IHostGCManager::SuspensionEnding 方法可以查询保留的内存，以确保在内存更新时立即检索计数。

请参阅

- AppDomain.MonitoringIsEnabled
- ICLRAppDomainResourceMonitor 接口
- <appDomainResourceMonitoring>
- CLR ETW 事件

弱引用

2021/11/16 •

如果应用程序的代码可以访问一个正由该程序使用的对象，垃圾回收器就不能回收该对象，那么，就认为应用程序对该对象具有强引用。

弱引用允许应用程序访问对象，同时也允许垃圾回收器收集相应的对象。如果不存在强引用，则弱引用的有限期只限于收集对象前的一个不确定的时间段。使用弱引用时，应用程序仍可对该对象进行强引用，这样做可防止该对象被收集。但始终存在这样的风险：垃圾回收器在重新建立强引用之前先处理该对象。

占用大量内存，但通过垃圾回收功能回收以后很容易重新创建的对象特别适合使用弱引用。

假设 Windows 窗体应用中的树状视图向用户显示层次结构复杂的选项。如果基础数据量很大，则用户使用应用程序中的其他部分时，在内存中保留该树会导致效率低下。

当用户切换到应用的其他部分时，可以使用 WeakReference 类创建对树的弱引用，并销毁所有强引用。当用户切换回该树时，应用程序会尝试获得对该树的强引用，如果成功，就不必重新构造该树。

若要对某对象建立弱引用，请使用要跟踪的对象实例创建 WeakReference。然后将 Target 属性设置为该对象，将该对象的原始引用设置为 `null`。有关代码示例，请参阅类库中的 WeakReference。

短弱引用和长弱引用

可以创建短弱引用或长弱引用：

- Short

  垃圾回收功能回收对象后，短弱引用的目标会变为 `null`。弱引用本身是托管对象，与其他任何托管对象一样需要经过垃圾回收。短弱引用是 WeakReference 的无参数构造函数。

- Long

  在对象的 Finalize 方法已调用后，长弱引用获得保留。这样，便可以重新创建该对象，但该对象仍保持不可预知的状态。若要使用长引用，请在 WeakReference 构造函数中指定 `true`。

  如果对象类型不包含 Finalize 方法，应用的是短弱引用功能。弱引用只在目标被收集前有效，运行终结器后可以随时收集目标。

若要建立强引用并重新使用对象，请将 WeakReference 的 Target 属性强制转换为对象类型。如果 Target 属性返回 `null`，表示对象已被收集；否则，可继续使用对象，因为应用已重新获得对它的强引用。

使用弱引用的准则

仅在必要时使用长弱引用，因为在终结后对象的状态不可预知。

避免对小对象使用弱引用，因为指针本身可能和对象一样大，或者比对象还大。

避免将弱引用作为内存管理问题的自动解决方案，而应开发一个有效的缓存策略来处理应用程序的对象。

请参阅

- 垃圾回收

内存和跨度相关类型

2021/11/16 •

从 .NET Core 2.1 开始，.NET 包含多个相互关联的类型，它们表示任意内存的相邻强类型区域。这些方法包括：

- System.Span<T>，用于访问连续内存区域的类型。Span<T> 实例可由一组 `T` 类型、一个 String、一个使用 stackalloc 分配的缓冲区或一个指向非托管内存的指针提供支持。由于它必须在堆栈上进行分配，因此存在诸多限制。例如，类中的字段不能是 Span<T> 类型，跨度类型也不能在异步操作中使用。
- System.ReadOnlySpan<T>，Span<T> 结构的不可变版本。
- System.Memory<T>：连续内存区域的包装器。Memory<T> 实例可以由 `T` 类型数组、String 或内存管理器提供支持。因为 Memory<T> 可以存储在托管堆上，所以它没有任何 Span<T> 限制。
- System.ReadOnlyMemory<T>，Memory<T> 结构的不可变版本。
- System.Buffers.MemoryPool<T>，它将强类型内存块从内存池分配给所有者。IMemoryOwner<T> 实例可以通过调用 MemoryPool<T>.Rent 从池中租用，并通过调用 MemoryPool<T>.Dispose() 将其释放回池中。
- System.Buffers.IMemoryOwner<T>，表示内存块的所有者并控制其生存期管理。
- MemoryManager<T>，一个抽象基类，可用于替换 Memory<T> 的实现，以便 Memory<T> 可以由其他类型（如安全句柄）提供支持。MemoryManager<T> 适用于高级方案。
- ArraySegment<T>，从特定索引开始的特定数量数组元素的包装器。
- System.MemoryExtensions，用于将字符串、数组和数组段转换为 Memory<T> 块的扩展方法集合。

System.Span<T>、System.Memory<T> 及其只读对等体被设计为，允许创建算法来避免不必要地复制内存或在托管堆上进行分配。创建它们（通过 `Slice` 或它们的构造函数）并不涉及复制基础缓冲：只更新代表已包装内存的“视图”的相关引用和偏移。

> **NOTE**
> 对于早期框架，Span<T> 和 Memory<T> 在 System.Memory NuGet 包中提供。

有关更多信息，请参见 System.Buffers 命名空间。

使用内存和跨度

由于内存和跨度相关类型通常用于在处理管道中存储数据，因此开发人员在使用 Span<T>、Memory<T> 和相关类型时要务必遵循一套最佳做法。内存<T>> 和跨度<T>> 使用准则中介绍了这些最佳做法。

请参阅

- System.Memory<T>
- System.ReadOnlyMemory<T>
- System.Span<T>
- System.ReadOnlySpan<T>
- System.Buffers

内存<T>和跨度<T>使用准则

2021/11/16 •

.NET Core 包括多个表示内存的任意连续区域的类型。.NET Core 2.0 引入了 Span<T> 和 ReadOnlySpan<T>，它们是可由托管或非托管内存提供支持的轻量级内存缓冲区。由于这些类型只能存储在堆栈上，因此它们不适用于多种方案，包括异步方法调用。.NET Core 2.1 添加了其他一些类型，包括 Memory<T>、ReadOnlyMemory<T>、IMemoryOwner<T> 和 MemoryPool<T>。与 Span<T> 相同，Memory<T> 及其相关类型可以由托管和非托管内存提供支持。与 Span<T> 不同，Memory<T> 可以存储在托管堆上。

Span<T> 和 Memory<T> 都是可用于管道的结构化数据的缓冲区。也就是说，它们设计的目的是将某些或所有数据有效地传递到管道中的组件，这些组件可以对其进行处理并(可选)修改缓冲区。由于 Memory<T> 及其相关类型可由多个组件或多个线程访问，因此开发人员必须遵循一些标准使用准则才能生成可靠的代码。

所有者、使用者和生存期管理

由于可以在各个 API 之间传送缓冲区，以及由于缓冲区有时可以从多个线程进行访问，因此请务必考虑生存期管理。下面介绍三个核心概念：

- **所有权**。缓冲区实例的所有者负责生存期管理，包括在不再使用缓冲区时将其销毁。所有缓冲区都拥有一个所有者。通常，所有者是创建缓冲区或从工厂接收缓冲区的组件。所有权也可以转让；组件 A 可以将缓冲区的控制权转让给组件 B，此时组件 A 就无法再使用该缓冲区，组件 B 将负责在不再使用缓冲区时将其销毁。
- **使用**。允许缓冲区实例的使用者通过从中读取并可能写入其中来使用缓冲区实例。缓冲区一次可以拥有一个使用者，除非提供了某些外部同步机制。缓冲区的当前使用者不一定是缓冲区的所有者。
- **租用**。租用是允许特定组件成为缓冲区使用者的时长。

以下伪代码示例阐释了这三个概念。它包括实例化类型为 Char 的 Memory<T> 缓冲区的 `Main` 方法，调用 `WriteInt32ToBuffer` 方法以将整数的字符串表示形式写入缓冲区，然后调用 `DisplayBufferToConsole` 方法以显示缓冲区的值。

```
using System;

class Program
{
    // Write 'value' as a human-readable string to the output buffer.
    void WriteInt32ToBuffer(int value, Buffer buffer);

    // Display the contents of the buffer to the console.
    void DisplayBufferToConsole(Buffer buffer);

    // Application code
    static void Main()
    {
        var buffer = CreateBuffer();
        try
        {
            int value = Int32.Parse(Console.ReadLine());
            WriteInt32ToBuffer(value, buffer);
            DisplayBufferToConsole(buffer);
        }
        finally
        {
            buffer.Destroy();
        }
    }
}
```

`Main` 方法创建缓冲区(在此示例中为 Span<T> 实例),因此它是其所有者。因此,`Main` 将负责在不再使用缓冲区时将其销毁。这是通过调用缓冲区的 Span<T>.Clear() 方法完成的。(此处的 Clear() 方法实际上会清除缓冲区的内存;Span<T> 结构实际上没有销毁缓冲区的方法。)

缓冲区有两个使用者:`WriteInt32ToBuffer` 和 `DisplayBufferToConsole`。一次只能有一个使用者(先是 `WriteInt32ToBuffer`,然后是 `DisplayBufferToConsole`),这两个使用者都不拥有缓冲区。另请注意,此上下文中的"使用者"并不意味着以只读形式查看缓冲区;如果提供了以读/写形式查看缓冲区的权限,则使用者可以像 `WriteInt32ToBuffer` 那样修改缓冲区的内容。

`WriteInt32ToBuffer` 方法在方法调用的开始时间和方法返回的时间之间会租用(可以使用)缓冲区。同样,`DisplayBufferToConsole` 在执行时会租用缓冲区,回退该方法时将解除租用。(没有用于租用管理的 API;"租用"是概念性内容。)

Memory<T> 和所有者/使用者模型

如所有者、使用者和生存期管理部分中指出,缓冲区始终都有一个所有者。.NET Core 支持以下两种所有权模型:

- 支持单个所有权的模型。缓冲区在其整个生存期内拥有单个所有者。
- 支持所有权转让的模型。缓冲区的所有权可以从其原始所有者(其创建者)转让给其他组件,该组件随后将负责缓冲区的生存期管理。该所有者可以反过来将所有权转让给其他组件等。

使用 System.Buffers.IMemoryOwner<T> 接口显式管理缓冲区的所有权。IMemoryOwner<T> 支持两种所有权模型。具有 IMemoryOwner<T> 引用的组件拥有缓冲区。以下示例使用 IMemoryOwner<T> 实例反映 Memory<T> 缓冲区的所有权。

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent();

        Console.Write("Enter a number: ");
        try {
            var value = Int32.Parse(Console.ReadLine());

            var memory = owner.Memory;

            WriteInt32ToBuffer(value, memory);

            DisplayBufferToConsole(owner.Memory.Slice(0, value.ToString().Length));
        }
        catch (FormatException) {
            Console.WriteLine("You did not enter a valid number.");
        }
        catch (OverflowException) {
            Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
        }
        finally {
            owner?.Dispose();
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Span;
        for (int ctr = 0; ctr < strValue.Length; ctr++)
            span[ctr] = strValue[ctr];
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

也可以使用 `using` 编写此示例：

```
using System;
using System.Buffers;

class Example
{
    static void Main()
    {
        using (IMemoryOwner<char> owner = MemoryPool<char>.Shared.Rent())
        {
            Console.Write("Enter a number: ");
            try {
                var value = Int32.Parse(Console.ReadLine());

                var memory = owner.Memory;
                WriteInt32ToBuffer(value, memory);
                DisplayBufferToConsole(memory.Slice(0, value.ToString().Length));
            }
            catch (FormatException) {
                Console.WriteLine("You did not enter a valid number.");
            }
            catch (OverflowException) {
                Console.WriteLine($"You entered a number less than {Int32.MinValue:N0} or greater than
{Int32.MaxValue:N0}.");
            }
        }
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();

        var span = buffer.Slice(0, strValue.Length).Span;
        strValue.AsSpan().CopyTo(span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

在此代码中：

- `Main` 方法保留对 IMemoryOwner<T> 实例的引用，因此 `Main` 方法是缓冲区的所有者。
- `WriteInt32ToBuffer` 和 `DisplayBufferToConsole` 方法接受 Memory<T> 作为公共 API。因此，它们是缓冲区的使用者。并且它们一次仅使用一个。

尽管 `WriteInt32ToBuffer` 方法用于将值写入缓冲区，但 `DisplayBufferToConsole` 方法并不如此。若要反映此情况，可以接受类型为 ReadOnlyMemory<T> 的参数。有关 ReadOnlyMemory<T> 的详细信息，请参阅规则 2：如果缓冲区应为只读，则使用 ReadOnlySpan<T> 或 ReadOnlyMemory<T>。

“无所有者”Memory<T> 实例

无需使用 IMemoryOwner<T> 即可创建 Memory<T> 实例。在这种情况下，缓冲区的所有权是隐式的而不是显式的，并且仅支持单所有者模型。可以通过以下方式达到此目的：

- 直接调用 Memory<T> 构造函数之一，传入 `T[]`，如下面的示例所示。
- 调用 String.AsMemory 扩展方法以生成 `ReadOnlyMemory<char>` 实例。

```
using System;

class Example
{
    static void Main()
    {
        Memory<char> memory = new char[64];

        Console.Write("Enter a number: ");
        var value = Int32.Parse(Console.ReadLine());

        WriteInt32ToBuffer(value, memory);
        DisplayBufferToConsole(memory);
    }

    static void WriteInt32ToBuffer(int value, Memory<char> buffer)
    {
        var strValue = value.ToString();
        strValue.AsSpan().CopyTo(buffer.Slice(0, strValue.Length).Span);
    }

    static void DisplayBufferToConsole(Memory<char> buffer) =>
        Console.WriteLine($"Contents of the buffer: '{buffer}'");
}
```

最初创建 Memory<T> 实例的方法是缓冲区的隐式所有者。无法将所有权转让给任何其他组件，因为没有 IMemoryOwner<T> 实例可用于进行转让。（或者，也可以假设运行时的垃圾回收器拥有缓冲区，而所有方法仅使用缓冲区。）

使用准则

由于拥有内存块，但打算传递到多个组件，因此其中一些组件可能会同时在特定的内存块上运行，请务必建立使用 Memory<T> 和 Span<T> 的准则。准则是必需的，因为：

- 组件可以在内存块的所有者发布内存块之后保留对该内存块的引用。
- 组件可以同时在运行其他组件的缓冲区上运行，在该过程中会损坏缓冲区中的数据。
- 虽然 Span<T> 的堆栈分配特性优化了性能并使 Span<T> 成为在内存块上运行的首选类型，但它也使 Span<T> 受一些主要限制的约束。请务必了解何时使用 Span<T> 以及何时使用 Memory<T>。

下面介绍成功使用 Memory<T> 及其相关类型的建议。除非另有明确说明，否则适用于 Memory<T> 和 Span<T> 的指南也适用于 ReadOnlyMemory<T> 和 ReadOnlySpan<T>。

规则 1：对于同步 API，如有可能，请使用 Span<T>（而不是 Memory<T>）作为参数。

Span<T> 比 Memory<T> 更通用，可以表示更多种类的连续内存缓冲区。Span<T> 还提供比 Memory<T> 更好的性能。最后，尽管无法进行 Span<T> 到 Memory<T> 的转换，但可以使用 Memory<T>.Span 属性将 Memory<T> 实例转换为 Span<T>。因此，如果调用方恰好具有 Memory<T> 实例，则它们不管怎样都可以使用 Span<T> 参数调用你的方法。

使用类型为 Span<T>（而不是类型为 Memory<T>）的参数还可以帮助你编写正确的使用方法实现。你将自动进行编译时检查，以确保不尝试访问方法租用之外的缓冲区（后续部分将对此进行详细介绍）。

有时，必须使用 Memory<T> 参数（而不是 Span<T> 参数），即使完全同步也是如此。所依赖的 API 可能仅接受 Memory<T> 参数。这没有问题，但应注意同步使用 Memory<T> 时所涉及的权衡取舍。

规则 2：如果缓冲区应为只读，则使用 ReadOnlySpan<T> 或 ReadOnlyMemory<T>。

在前面的示例中，`DisplayBufferToConsole` 方法仅从缓冲区读取；它不会修改缓冲区的内容。方法签名应进行如下更改。

```
void DisplayBufferToConsole(ReadOnlyMemory<char> buffer);
```

事实上，如果我们结合使用此规则和规则 1，我们可以做得更好，并按如下所示重写方法签名：

```
void DisplayBufferToConsole(ReadOnlySpan<char> buffer);
```

`DisplayBufferToConsole` 方法现在几乎适用于每一个能够想到的缓冲区类型：`T[]`，使用 stackalloc 分配的存储等。甚至可以向其直接传递 String！

规则 3：如果方法接受 Memory<T> 并返回 `void`，则在方法返回之后不得使用 Memory<T> 实例。

这与前面提到的"租用"概念相关。返回 void 的方法对 Memory<T> 实例的租用将在进入该方法时开始，并在退出该方法时结束。请考虑以下示例，该示例会基于控制台中的输入在循环中调用 `Log`。

```
using System;
using System.Buffers;

public class Example
{
    // implementation provided by third party
    static extern void Log(ReadOnlyMemory<char> message);

    // user code
    public static void Main()
    {
        using (var owner = MemoryPool<char>.Shared.Rent())
        {
            var memory = owner.Memory;
            var span = memory.Span;
            while (true)
            {
                int value = Int32.Parse(Console.ReadLine());
                if (value < 0)
                    return;

                int numCharsWritten = ToBuffer(value, span);
                Log(memory.Slice(0, numCharsWritten));
            }
        }
    }

    private static int ToBuffer(int value, Span<char> span)
    {
        string strValue = value.ToString();
        int length = strValue.Length;
        strValue.AsSpan().CopyTo(span.Slice(0, length));
        return length;
    }
}
```

如果 `Log` 是完全同步的方法，则此代码将按预期运行，因为在任何给定时间只有一个活动的内存实例使用者。但请假设 `Log` 具有此实现。

```
// !!! INCORRECT IMPLEMENTATION !!!
static void Log(ReadOnlyMemory<char> message)
{
    // Run in background so that we don't block the main thread while performing IO.
    Task.Run(() =>
    {
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(message);
    });
}
```

在此实现中，`Log` 违反了其租用，因为它在返回原始方法之后仍尝试在后台使用 Memory<T> 示例。`Main` 方法可能会在 `Log` 尝试从缓冲区进行读取时更改缓冲区，这可能导致数据损坏。

有多种方法可解决此问题：

- `Log` 方法可以按 `Log` 方法的以下实现所示返回 Task，而不是 `void`。

  ```
  // An acceptable implementation.
  static Task Log(ReadOnlyMemory<char> message)
  {
      // Run in the background so that we don't block the main thread while performing IO.
      return Task.Run(() => {
                  StreamWriter sw = File.AppendText(@".\input-numbers.dat");
          sw.WriteLine(message);
          sw.Flush();
      });
  }
  ```

- 可以改为按如下所示实现 `Log`：

  ```
  // An acceptable implementation.
  static void Log(ReadOnlyMemory<char> message)
  {
      string defensiveCopy = message.ToString();
      // Run in the background so that we don't block the main thread while performing IO.
      Task.Run(() => {
          StreamWriter sw = File.AppendText(@".\input-numbers.dat");
          sw.WriteLine(defensiveCopy);
          sw.Flush();
      });
  }
  ```

规则 4：如果方法接受 Memory<T> 并返回某个任务，则在该任务转换为终止状态之后不得使用 Memory<T> 实例。

这只是规则 3 的异步变体。可以按如下所示编写前面示例中的 `Log` 方法以遵守此规则：

```
// An acceptable implementation.
static void Log(ReadOnlyMemory<char> message)
{
    // Run in the background so that we don't block the main thread while performing IO.
    Task.Run(() => {
        string defensiveCopy = message.ToString();
        StreamWriter sw = File.AppendText(@".\input-numbers.dat");
        sw.WriteLine(defensiveCopy);
        sw.Flush();
    });
}
```

此处的"终止状态"表示任务转换为已完成、已出错或已取消状态。换而言之，"终止状态"表示"导致等待引发或继续执行的任何状态"。

此指南适用于返回 Task、Task<TResult>、ValueTask<TResult> 或任何类似类型的方法。

规则 5：如果构造函数接受 Memory<T> 作为参数，则假定构造对象上的实例方法是 Memory<T> 实例的使用者。

请看下面的示例：

```
class OddValueExtractor
{
    public OddValueExtractor(ReadOnlyMemory<int> input);
    public bool TryReadNextOddValue(out int value);
}

void PrintAllOddValues(ReadOnlyMemory<int> input)
{
    var extractor = new OddValueExtractor(input);
    while (extractor.TryReadNextOddValue(out int value))
    {
      Console.WriteLine(value);
    }
}
```

此处的 `OddValueExtractor` 构造函数接受 `ReadOnlyMemory<int>` 作为构造函数参数，因此构造函数本身是 `ReadOnlyMemory<int>` 实例的使用者，并且返回值上的所有实例方法也是原始 `ReadOnlyMemory<int>` 实例的使用者。这意味着 `TryReadNextOddValue` 使用 `ReadOnlyMemory<int>` 实例，即使该实例未直接传递到 `TryReadNextOddValue` 方法也是如此。

规则 6：如果类型具有可设置的 Memory<T> 类型的属性（或等效实例方法），则假定该对象上的实例方法是 Memory<T> 实例的使用者。

这实际上只是规则 5 的变体。存在此规则的原因是假定属性 setter 或等效方法捕获并保留其输入，因此同一对象上的实例方法可能会使用已捕获状态。

下面的示例将触发以下规则：

```
class Person
{
    // Settable property.
    public Memory<char> FirstName { get; set; }

    // alternatively, equivalent "setter" method
    public SetFirstName(Memory<char> value);

    // alternatively, a public settable field
    public Memory<char> FirstName;
}
```

规则 7：如果具有 IMemoryOwner<T> 引用，则必须在某些时候对其进行处理或转让其所有权（但不同时执行两个操作）。

由于 Memory<T> 实例可能由托管或非托管内存提供支持，因此在对 Memory<T> 实例执行的工作完成之后，所有者必须调用 MemoryPool<T>.Dispose。此外，所有者可能会将 IMemoryOwner<T> 实例的所有权转让给其他组件，同时获取组件将负责在适当时间调用 MemoryPool<T>.Dispose（稍后将对此进行详细介绍）。

调用 Dispose 方法失败可能会导致非托管内存泄漏或其他性能降低。

此规则也适用于调用工厂方法（如 MemoryPool<T>.Rent）的代码。调用方将成为返回的 IMemoryOwner<T> 的

所有者，并负责在完成后处理实例。

规则 8：如果 API 接口中具有 IMemoryOwner<T> 参数，即表示你接受该实例的所有权。

接受此类型的实例表示组件打算获取此实例的所有权。组件将负责根据规则 7 进行正确处理。

在方法调用完成后，将 IMemoryOwner<T> 实例的所有权转让给其他组件的任何组件应不再使用该实例。

> **IMPORTANT**
> 如果构造函数接受 IMemoryOwner<T> 作为参数，则其类型应实现 IDisposable，并且 Dispose 方法应调用 MemoryPool<T>.Dispose。

规则 9：如果正在包装同步 P/Invoke 方法，则 API 应接受 Span<T> 作为参数。

根据规则 1，Span<T> 通常是用于同步 API 的正确类型。可以通过 `fixed` 关键字固定 Span<T> 实例，如下面的示例所示。

```
using System.Runtime.InteropServices;

[DllImport(...)]
private static extern unsafe int ExportedMethod(byte* pbData, int cbData);

public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        int retVal = ExportedMethod(pbData, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

在上一示例中，如果输入跨度为空，则 `pbData` 可以为 Null。如果导出的方法绝对需要 `pbData` 为非 Null，即使 `cbData` 为 0，则可以按如下所示实现该方法：

```
public unsafe int ManagedWrapper(Span<byte> data)
{
    fixed (byte* pbData = &MemoryMarshal.GetReference(data))
    {
        byte dummy = 0;
        int retVal = ExportedMethod((pbData != null) ? pbData : &dummy, data.Length);

        /* error checking retVal goes here */

        return retVal;
    }
}
```

规则 10：如果正在包装异步 P/Invoke 方法，则 API 应接受 Memory<T> 作为参数。

由于不能在异步操作中使用 `fixed` 关键字，因此使用 Memory<T>.Pin 方法固定 Memory<T> 实例，无论该实例表示的连续内存类型为何。下面的示例演示了如何使用此 API 执行异步 P/Invoke 调用。

```
using System.Runtime.InteropServices;

[UnmanagedFunctionPointer(...)]
private delegate void OnCompletedCallback(IntPtr state, int result);
```

```
[DllImport(...)]
private static extern unsafe int ExportedAsyncMethod(byte* pbData, int cbData, IntPtr pState, IntPtr
lpfnOnCompletedCallback);

private static readonly IntPtr _callbackPtr = GetCompletionCallbackPointer();

public unsafe Task<int> ManagedWrapperAsync(Memory<byte> data)
{
    // setup
    var tcs = new TaskCompletionSource<int>();
    var state = new MyCompletedCallbackState
    {
        Tcs = tcs
    };
    var pState = (IntPtr)GCHandle.Alloc(state);

    var memoryHandle = data.Pin();
    state.MemoryHandle = memoryHandle;

    // make the call
    int result;
    try
    {
        result = ExportedAsyncMethod((byte*)memoryHandle.Pointer, data.Length, pState, _callbackPtr);
    }
    catch
    {
        ((GCHandle)pState).Free(); // cleanup since callback won't be invoked
        memoryHandle.Dispose();
        throw;
    }

    if (result != PENDING)
    {
        // Operation completed synchronously; invoke callback manually
        // for result processing and cleanup.
        MyCompletedCallbackImplementation(pState, result);
    }

    return tcs.Task;
}

private static void MyCompletedCallbackImplementation(IntPtr state, int result)
{
    GCHandle handle = (GCHandle)state;
    var actualState = (MyCompletedCallbackState)(handle.Target);
    handle.Free();
    actualState.MemoryHandle.Dispose();

    /* error checking result goes here */

    if (error)
    {
        actualState.Tcs.SetException(...);
    }
    else
    {
        actualState.Tcs.SetResult(result);
    }
}

private static IntPtr GetCompletionCallbackPointer()
{
    OnCompletedCallback callback = MyCompletedCallbackImplementation;
    GCHandle.Alloc(callback); // keep alive for lifetime of application
    return Marshal.GetFunctionPointerForDelegate(callback);
}
```

```
private class MyCompletedCallbackState
{
    public TaskCompletionSource<int> Tcs;
    public MemoryHandle MemoryHandle;
}
```

请参阅

- System.Memory<T>
- System.Buffers.IMemoryOwner<T>
- System.Span<T>

本机互操作性

2021/11/16 •

以下文章介绍在 .NET 中实现"本机互操作性"的各种方法。

调用本机代码的原因有以下几种：

- 操作系统附带的许多 API 并不在托管类库中。该应用场景最好的例子就是对硬件或操作系统管理功能的访问。
- 与其他具有或可生成 C 式 ABI（本机 ABI）的组件通信，如通过 Java 本机接口 (JNI) 公开的 Java 代码或可生成本机组件的任何其他托管语言。
- 在 Windows 上，安装的大部分软件（例如 Microsoft Office 套件）会注册 COM 组件，用于代表软件的程序，并使开发人员能够自动化或者使用它们。在这种情况下，也需要本机互操作性。

前述列表并未包括开发人员想要/偏向于/需要与本机组件交互的所有可能场合与情境。例如，.NET 类库使用本机互操作性支持来实现其相当多的 API，如控制台支持和操作、文件系统访问，等等。但务必应知道，在需要的情况下是有其他选择的。

> **NOTE**
> 本部分中的大多示例是针对 .NET Core 支持的所有三个平台（Windows、Linux 和 macOS）提供的。但是，在某些简短的演示性示例中，只显示了一个使用 Windows 文件名和扩展名（即，库的扩展名"dll"）的例子。这并不意味着这些功能不能在 Linux 或 macOS 上使用，只是出于方便而未提到这些平台。

请参阅

- 平台调用 (P/Invoke)
- 类型封送
- 本机互操作性最佳做法

平台调用 (P/Invoke)

2021/11/16 •

P/Invoke 是可用于从托管代码访问非托管库中的结构、回调和函数的一种技术。大多数 P/Invoke API 包含在以下两个命名空间中：`System` 和 `System.Runtime.InteropServices`。使用这两个命名空间可提供用于描述如何与本机组件通信的工具。

我们从最常见的示例着手。该示例在托管代码中调用非托管函数。让我们从命令行应用程序显示一个消息框：

```
using System;
using System.Runtime.InteropServices;

public class Program
{
    // Import user32.dll (containing the function we need) and define
    // the method corresponding to the native function.
    [DllImport("user32.dll", CharSet = CharSet.Unicode, SetLastError = true)]
    private static extern int MessageBox(IntPtr hWnd, string lpText, string lpCaption, uint uType);

    public static void Main(string[] args)
    {
        // Invoke the function as a regular managed method.
        MessageBox(IntPtr.Zero, "Command-line message box", "Attention!", 0);
    }
}
```

上述示例非常简单，但确实演示了从托管代码调用非托管函数所需执行的操作。让我们逐步分析该示例：

- 第 1 行显示 `System.Runtime.InteropServices` 命名空间（用于保存全部所需项）的 using 语句。
- 第 7 行引入 `DllImport` 属性。此特性至关重要，因为它告诉运行时要加载非托管 DLL。传入的字符串是目标函数所在的 DLL。此外，它还指定哪些字符集用于封送字符串。最后，它指定此函数调用 SetLastError，且运行时应捕获相应错误代码，以便用户能够通过 Marshal.GetLastWin32Error() 检索它。
- 第 8 行显示了 P/Invoke 的关键作用。它定义了一个托管方法，该方法的签名与非托管方法**完全相同**。可以看到，声明中包含一个新关键字 `extern`，告诉运行时这是一个外部方法。调用该方法时，运行时应在 `DllImport` 特性中指定的 DLL 内查找该方法。

该示例的剩余部分无非就是调用该方法，就像调用其他任何托管方法一样。

在 macOS 上也可以使用类似的示例。需要更改 `DllImport` 属性中的库名称，因为 macOS 使用不同的方案来命名动态库。下面的示例使用 `getpid(2)` 函数获取应用程序的进程 ID，然后控制台上列显该 ID：

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libSystem shared library and define the method
        // corresponding to the native function.
        [DllImport("libSystem.dylib")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

它在 Linux 上也是类似的。函数名称相同，因为 `getpid(2)` 是标准 POSIX 系统调用。

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Import the libc shared library and define the method
        // corresponding to the native function.
        [DllImport("libc.so.6")]
        private static extern int getpid();

        public static void Main(string[] args)
        {
            // Invoke the function and get the process ID.
            int pid = getpid();
            Console.WriteLine(pid);
        }
    }
}
```

从非托管代码调用托管代码

运行时允许通信流量双向流通，这样，便可以使用函数指针从本机函数回调托管代码。在托管代码中，与函数指针最接近的功能就是委托，正是凭借这个功能，才能从本机代码回调托管代码。

此功能的使用方式类似于上面所述的从托管代码调用本机进程。对于给定的回调，需要定义一个与签名匹配的委托，并将其传入外部方法。运行时将负责处理所有剩余工作。

```
using System;
using System.Runtime.InteropServices;

namespace ConsoleApplication1
{
    public static class Program
    {
        // Define a delegate that corresponds to the unmanaged function.
        private delegate bool EnumWC(IntPtr hwnd, IntPtr lParam);

        // Import user32.dll (containing the function we need) and define
        // the method corresponding to the native function.
        [DllImport("user32.dll")]
        private static extern int EnumWindows(EnumWC lpEnumFunc, IntPtr lParam);

        // Define the implementation of the delegate; here, we simply output the window handle.
        private static bool OutputWindow(IntPtr hwnd, IntPtr lParam)
        {
            Console.WriteLine(hwnd.ToInt64());
            return true;
        }

        public static void Main(string[] args)
        {
            // Invoke the method; note the delegate as a first parameter.
            EnumWindows(OutputWindow, IntPtr.Zero);
        }
    }
}
```

在演练示例之前，最好是回顾一下所要使用的非托管函数的签名。要调用以枚举所有窗口的函数具有以下签名：
`BOOL EnumWindows (WNDENUMPROC lpEnumFunc, LPARAM lParam);`

第一个参数是回调。该回调具有以下签名：`BOOL CALLBACK EnumWindowsProc (HWND hwnd, LPARAM lParam);`

现在，让我们来演练示例：

- 示例中的第 9 行定义与非托管代码中回调签名匹配的委托。请注意如何在托管代码中使用 `IntPtr` 表示 LPARAM 和 HWND 类型。
- 第 13 和 14 行从 user32.dll 库中引入 `EnumWindows` 函数。
- 第 17 - 20 行实现该委托。在这个简单的示例中，我们只要将句柄输出到控制台。
- 最后，第 24 行调用外部方法并传入委托。

下面显示了 Linux 和 macOS 示例。在这些平台上，我们可以使用 C 库 `libc` 中的 `ftw` 函数。此函数用于遍历目录层次结构，它使用指向某个函数的指针作为其参数之一。该函数具有以下签名：
`int (*fn) (const char *fpath, const struct stat *sb, int typeflag)` 。

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libc.so.6")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code. You can find more information
    // about this in the section on marshalling below.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

macOS 示例使用相同的函数，唯一的差别在于 `DllImport` 特性的自变量，因为 macOS 将 `libc` 保留在不同的位置。

```
using System;
using System.Runtime.InteropServices;

namespace PInvokeSamples
{
    public static class Program
    {
        // Define a delegate that has the same signature as the native function.
        private delegate int DirClbk(string fName, StatClass stat, int typeFlag);

        // Import the libc and define the method to represent the native function.
        [DllImport("libSystem.dylib")]
        private static extern int ftw(string dirpath, DirClbk cl, int descriptors);

        // Implement the above DirClbk delegate;
        // this one just prints out the filename that is passed to it.
        private static int DisplayEntry(string fName, StatClass stat, int typeFlag)
        {
            Console.WriteLine(fName);
            return 0;
        }

        public static void Main(string[] args)
        {
            // Call the native function.
            // Note the second parameter which represents the delegate (callback).
            ftw(".", DisplayEntry, 10);
        }
    }

    // The native callback takes a pointer to a struct. The below class
    // represents that struct in managed code.
    [StructLayout(LayoutKind.Sequential)]
    public class StatClass
    {
        public uint DeviceID;
        public uint InodeNumber;
        public uint Mode;
        public uint HardLinks;
        public uint UserID;
        public uint GroupID;
        public uint SpecialDeviceID;
        public ulong Size;
        public ulong BlockSize;
        public uint Blocks;
        public long TimeLastAccess;
        public long TimeLastModification;
        public long TimeLastStatusChange;
    }
}
```

上面两个示例都依赖于参数，在这两种情况下，参数是作为托管类型提供的。运行时将采取“适当的措施”，在另一个平台上将这些代码处理成等效的代码。类型封送页介绍了如何将类型封送到本机代码。

更多资源

- PInvoke.net wiki 是一个出色的 Wiki 站点，其中提供了有关常用 Windows API 以及如何调用这些 API 的信息。
- C++/CLI 中的 P/Invoke
- 有关 P/Invoke 的 Mono 文档

类型封送

2021/11/16 •

封送 是当类型需要在托管代码和本机代码之间切换时转换类型的过程。

需要封送的原因在于托管代码与非托管代码中的类型并不相同。例如，在托管代码中，可指定 `String`。但在非托管环境中，字符串类型可以是 Unicode（“宽型”）、非 Unicode、null 结尾、ASCII 等。默认情况下，P/Invoke 子系统会尝试基于默认行为执行正确的操作，如本文中所述。但是，如果需要额外的控制，可以使用 MarshalAs 属性指定要在非托管端上使用的预期类型。例如，如果想要将字符串作为以 null 结尾的 ANSI 字符串发送，则可以执行如下操作：

```
[DllImport("somenativelibrary.dll")]
static extern int MethodA([MarshalAs(UnmanagedType.LPStr)] string parameter);
```

封送通用类型的默认规则

通常，运行时会尝试在封送时执行“正确的操作”，从而最大限地减少用户的工作。以下表格介绍了每种类型在用于参数或字段时的默认封送方式。C99/C++11 固定宽度的整数和字符类型用于确保下表适用于所有平台。可以使用与这些类型具有相同的对齐和大小要求的任何本机类型。

第一个表介绍了其封送与 P/Invoke 和字段封送相同的各种类型的映射。

.NET 类型	本机类型
`byte`	`uint8_t`
`sbyte`	`int8_t`
`short`	`int16_t`
`ushort`	`uint16_t`
`int`	`int32_t`
`uint`	`uint32_t`
`long`	`int64_t`
`ulong`	`uint64_t`
`char`	`char` 或 `char16_t` 依赖于 P/Invoke 或结构的 `CharSet`。请参阅字符集文档。
`string`	`char*` 或 `char16_t*` 依赖于 P/Invoke 或结构的 `CharSet`。请参阅字符集文档。
`System.IntPtr`	`intptr_t`

.NET 类型	默认封送
`System.UIntPtr`	`uintptr_t`
.NET 指针类型（例如，`void*`）	`void*`
从 `System.Runtime.InteropServices.SafeHandle` 派生的类型	`void*`
从 `System.Runtime.InteropServices.CriticalHandle` 派生的类型	`void*`
`bool`	Win32 `BOOL` 类型
`decimal`	COM `DECIMAL` 结构
.NET 委托	本机函数指针
`System.DateTime`	Win32 `DATE` 类型
`System.Guid`	Win32 `GUID` 类型

如果作为参数或结构进行封送，则部分封送类别具有不同的默认设置。

.NET 类型	默认值(类型)	默认值(类型)
.NET 数组	指向数组元素的本机表示形式的数组开头的指针。	不允许不带 `[MarshalAs]` 属性
`LayoutKind` 为 `Sequential` 或 `Explicit` 的类	指向类的本机表示形式的指针	类的本机表示形式

下表包含仅适用于 Windows 的默认封送规则。在非 Windows 平台上，无法封送这些类型。

.NET 类型	默认值(类型)	默认值(类型)
`object`	`VARIANT`	`IUnknown*`
`System.Array`	COM 接口	不允许不带 `[MarshalAs]` 属性
`System.ArgIterator`	`va_list`	不允许
`System.Collections.IEnumerator`	`IEnumVARIANT*`	不允许
`System.Collections.IEnumerable`	`IDispatch*`	不允许
`System.DateTimeOffset`	`int64_t` 表示自 1601 年 1 月 1 日午夜以来的时钟周期数	`int64_t` 表示自 1601 年 1 月 1 日午夜以来的时钟周期数

某些类型只能作为参数封送，而不能作为字段封送。下表列出了这些类型：

.NET 类型	默认封送(非 COM)
`System.Text.StringBuilder`	`char*` 或 `char16_t*` 依赖于 P/Invoke 的 `CharSet`。请参阅字符集文档。
`System.ArgIterator`	`va_list`（仅在 Windows x86/x64/arm64 上）
`System.Runtime.InteropServices.ArrayWithOffset`	`void*`
`System.Runtime.InteropServices.HandleRef`	`void*`

如果这些默认设置不执行所需的操作，则可以自定义参数的封送方式。参数封送一文介绍自定义不同参数类型的封送方式的具体步骤。

COM 方案中的默认封送

在 .NET 中调用 COM 对象上的方法时，.NET 运行时会更改默认封送规则，以匹配常见的 COM 语义。下表列出了 .NET 运行时在 COM 方案中使用的规则：

.NET 类型	默认封送(COM 非类)
`bool`	`VARIANT_BOOL`
`StringBuilder`	`LPWSTR`
`string`	`BSTR`
委托类型	在 .NET Framework 中为 `_Delegate*`。.NET Core 和 .NET 5+ 中不允许使用。
`System.Drawing.Color`	`OLECOLOR`
.NET 数组	`SAFEARRAY`
`string[]`	`BSTR` 的 `SAFEARRAY`

封送类和结构

有关类型封送的另一个问题是如何将结构传入非托管方法。例如，某些非托管方法需要使用结构作为参数。在这种情况下，需要在环境的托管部分中创建相应的结构或类，以便将它用作参数。不过，仅仅是定义类并不足够，还需要告知封送处理程序如何将类中的字段映射到非托管结构。在此处，`StructLayout` 属性将会很有用。

```
[DllImport("kernel32.dll")]
static extern void GetSystemTime(SystemTime systemTime);

[StructLayout(LayoutKind.Sequential)]
class SystemTime {
    public ushort Year;
    public ushort Month;
    public ushort DayOfWeek;
    public ushort Day;
    public ushort Hour;
    public ushort Minute;
    public ushort Second;
    public ushort Milsecond;
}

public static void Main(string[] args) {
    SystemTime st = new SystemTime();
    GetSystemTime(st);
    Console.WriteLine(st.Year);
}
```

前面的代码演示了如何调用 `GetSystemTime()` 函数的简单示例。值得注意的部分是第 4 行。该属性指定应按顺序将类的字段映射到另一端（非托管端）上的结构。这意味着，字段的命名并不重要，唯一重要的是字段顺序，因为这种顺序需对应于非托管结构，如下面的示例所示：

```
typedef struct _SYSTEMTIME {
  WORD wYear;
  WORD wMonth;
  WORD wDayOfWeek;
  WORD wDay;
  WORD wHour;
  WORD wMinute;
  WORD wSecond;
  WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

有时，默认结构封送不执行所需的操作。自定义结构封送一文介绍如何自定义结构的封送方式。

自定义结构封送

2021/11/16 •

有时，结构的默认封送规则无法完全满足要求。.NET 运行时提供了几个扩展点以自定义结构的布局和字段的封送方式。

自定义结构布局

.NET 提供了 System.Runtime.InteropServices.StructLayoutAttribute 属性和 System.Runtime.InteropServices.LayoutKind 枚举，允许用户自定义字段在内存中的放置方式。以下指南将帮助你避免常见问题。

✔ 请考虑尽量使用 `LayoutKind.Sequential`。

✔ 当本机结构还具有显式布局(如联合)时，请务必仅将 `LayoutKind.Explicit` 用于封送。

✘ 如果需要在 .NET Core 3.0 之前以运行时为目标，请避免在非 Windows 平台上封送结构时使用 `LayoutKind.Explicit`。3.0 之前的 .NET Core 运行时不支持在 Intel 或 AMD 64 位的非 Windows 系统上，按值将显式结构传递到本机函数。但是，运行时支持在所有平台上按引用传递显式结构。

自定义布尔字段封送

本机代码具有许多不同的布尔表示形式。仅在 Windows 上，有三种方式可用于表示布尔值。运行时不知道结构的本机定义，因此，它最多只能对如何封送布尔值做出猜测。.NET 运行时提供指示如何封送布尔字段的方式。下面的示例介绍如何将 .NET `bool` 封送到不同的本机布尔类型。

布尔值默认作为本机 4 字节 Win32 `BOOL` 值进行封送，如下面的示例所示：

```
public struct WinBool
{
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

如果想要明确指出，则可以使用 UnmanagedType.Bool 值获取如上所述的行为：

```
public struct WinBool
{
    [MarshalAs(UnmanagedType.Bool)]
    public bool b;
}
```

```
struct WinBool
{
    public BOOL b;
};
```

使用下面的 `UnmanagedType.U1` 或 `UnmanagedType.I1` 值，可以告知运行时将 `b` 字段作为 1 字节本机 `bool` 类型进行封送。

```
public struct CBool
{
    [MarshalAs(UnmanagedType.U1)]
    public bool b;
}
```

```
struct CBool
{
    public bool b;
};
```

在 Windows 上，可以使用 UnmanagedType.VariantBool 值告知运行时将布尔值封送到 2 字节的 `VARIANT_BOOL` 值：

```
public struct VariantBool
{
    [MarshalAs(UnmanagedType.VariantBool)]
    public bool b;
}
```

```
struct VariantBool
{
    public VARIANT_BOOL b;
};
```

> **NOTE**
> `VARIANT_BOOL` 与 `VARIANT_TRUE = -1` 和 `VARIANT_FALSE = 0` 中的大多数 bool 类型不同。此外，不等于 `VARIANT_TRUE` 的所有值都将被视为 false。

自定义数组字段封送

.NET 还包括自定义数组封送的多种方式。

默认情况下，.NET 将数组作为指向元素的连续列表的指针进行封送：

```
public struct DefaultArray
{
    public int[] values;
}
```

```
struct DefaultArray
{
    int* values;
};
```

如果要与 COM API 交互，则可能必须将数组作为 `SAFEARRAY*` 对象进行封送。可以使用 System.Runtime.InteropServices.MarshalAsAttribute 和 UnmanagedType.SafeArray 值告知运行时将数组作为 `SAFEARRAY*` 进行封送：

```
public struct SafeArrayExample
{
    [MarshalAs(UnmanagedType.SafeArray)]
    public int[] values;
}
```

```
struct SafeArrayExample
{
    SAFEARRAY* values;
};
```

如果需要自定义 `SAFEARRAY` 中的元素类型，则可以使用 MarshalAsAttribute.SafeArraySubType 和 MarshalAsAttribute.SafeArrayUserDefinedSubType 字段自定义 `SAFEARRAY` 的确切元素类型。

如果需要就地封送数组，则可以使用 UnmanagedType.ByValArray 值告知封送处理程序就地封送数组。使用此封送时，还必须为数组中的元素数对应的 MarshalAsAttribute.SizeConst 字段提供一个值，以便运行时可以正确地为结构分配空间。

```
public struct InPlaceArray
{
    [MarshalAs(UnmanagedType.ByValArray, SizeConst = 4)]
    public int[] values;
}
```

```
struct InPlaceArray
{
    int values[4];
};
```

> **NOTE**
> .NET 不支持将变长数组字段作为 C99 可变数组成员进行封送。

自定义字符串字段封送

.NET 还提供用于封送字符串字段的各种自定义。

默认情况下，.NET 将字符串作为指向以 null 结尾的字符串的指针进行封送。编码取决于 System.Runtime.InteropServices.StructLayoutAttribute 中的 StructLayoutAttribute.CharSet 字段的值。如果未指定任何属性，则编码将默认为 ANSI 编码。

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char* str;
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    public string str;
}
```

```
struct DefaultString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

如果需要对不同字段使用不同编码或只是希望在结构定义中更加明确，则可以在 System.Runtime.InteropServices.MarshalAsAttribute 属性上使用 UnmanagedType.LPStr 或 UnmanagedType.LPWStr 值。

```
public struct AnsiString
{
    [MarshalAs(UnmanagedType.LPStr)]
    public string str;
}
```

```
struct AnsiString
{
    char* str;
};
```

```
public struct UnicodeString
{
    [MarshalAs(UnmanagedType.LPWStr)]
    public string str;
}
```

```
struct UnicodeString
{
    char16_t* str; // Could also be wchar_t* on Windows.
};
```

如果想要使用 UTF-8 编码封送字符串，则可以在 MarshalAsAttribute 中使用 UnmanagedType.LPUTF8Str 值。

```
public struct UTF8String
{
    [MarshalAs(UnmanagedType.LPUTF8Str)]
    public string str;
}
```

```
struct UTF8String
{
    char* str;
};
```

> **NOTE**
> 使用 UnmanagedType.LPUTF8Str 需要 .NET Framework 4.7（或更高版本）或 .NET Core 1.1（或更高版本）。不能在 .NET Standard 2.0 中使用。

如果使用 COM API，则可能需要将字符串作为 `BSTR` 进行封送。使用 UnmanagedType.BStr 值可以将字符串作为 `BSTR` 进行封送。

```
public struct BString
{
    [MarshalAs(UnmanagedType.BStr)]
    public string str;
}
```

```
struct BString
{
    BSTR str;
};
```

使用基于 WinRT 的 API 时，可能需要将字符串作为 `HSTRING` 进行封送。使用 UnmanagedType.HString 值可以将字符串作为 `HSTRING` 进行封送。

```
public struct HString
{
    [MarshalAs(UnmanagedType.HString)]
    public string str;
}
```

```
struct BString
{
    HSTRING str;
};
```

如果 API 要求你将字符串就地传入结构，则可以使用 UnmanagedType.ByValTStr 值。务必注意，通过 `ByValTStr` 封送的字符串的编码由 `CharSet` 属性确定。此外，还需要通过 MarshalAsAttribute.SizeConst 字段传递字符串长度。

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Ansi)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char str[4];
};
```

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct DefaultString
{
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 4)]
    public string str;
}
```

```
struct DefaultString
{
    char16_t str[4]; // Could also be wchar_t[4] on Windows.
};
```

自定义十进制字段封送

如果在 Windows 上操作，则可能会遇到一些使用本机 `CY` 或 `CURRENCY` 结构的 API。默认情况下，.NET `decimal` 类型会封送到本机 `DECIMAL` 结构。但是，可以使用包含 UnmanagedType.Currency 值的 MarshalAsAttribute 来指示封送处理程序将 `decimal` 值转换为本机 `CY` 值。

```
public struct Currency
{
    [MarshalAs(UnmanagedType.Currency)]
    public decimal dec;
}
```

```
struct Currency
{
    CY dec;
};
```

封送 `System.Object`

在 Windows 上，可以将类型为 `object` 的字段封送到本机代码。可以将这些字段封送到以下三个类型之一：

- `VARIANT`
- `IUnknown*`
- `IDispatch*`

默认情况下，将类型为 `object` 的字段封送到用来包装对象的 `IUnknown*`。

```
public struct ObjectDefault
{
    public object obj;
}
```

```
struct ObjectDefault
{
    IUnknown* obj;
};
```

如果要将对象字段封送到 `IDispatch*`，请添加包含 UnmanagedType.IDispatch 值的 MarshalAsAttribute。

```
public struct ObjectDispatch
{
    [MarshalAs(UnmanagedType.IDispatch)]
    public object obj;
}
```

```
struct ObjectDispatch
{
    IDispatch* obj;
};
```

如果要将其作为 `VARIANT` 进行封送，请添加包含 UnmanagedType.Struct 值的 MarshalAsAttribute。

```
public struct ObjectVariant
{
    [MarshalAs(UnmanagedType.Struct)]
    public object obj;
}
```

```
struct ObjectVariant
{
    VARIANT obj;
};
```

下表介绍了如何将 `obj` 字段的不同运行时类型映射到存储在 `VARIANT` 中的各种类型：

.NET	VARIANT		.NET	VARIANT
`byte`	`VT_UI1`		`System.Runtime.Interop`	`VT_BSTR` `BStrWrapper`
`sbyte`	`VT_I1`		`object`	`VT_DISPATCH`
`short`	`VT_I2`		`System.Runtime.Interop`	`VT_UNKNOWN` `nownWrapper`
`ushort`	`VT_UI2`		`System.Runtime.Interop`	`VT_DISPATCH` `atchWrapper`
`int`	`VT_I4`		`System.Reflection.Miss`	`VT_ERROR`
`uint`	`VT_UI4`		`(object)null`	`VT_EMPTY`
`long`	`VT_I8`		`bool`	`VT_BOOL`
`ulong`	`VT_UI8`		`System.DateTime`	`VT_DATE`
`float`	`VT_R4`		`decimal`	`VT_DECIMAL`
`double`	`VT_R8`		`System.Runtime.Interop`	`VT_CURRENCY` `encyWrapper`
`char`	`VT_UI2`		`System.DBNull`	`VT_NULL`
`string`	`VT_BSTR`			

自定义参数封送

2021/11/16 •

当 .NET 运行时的默认参数封送行为不执行所需的操作时，可以使用 System.Runtime.InteropServices.MarshalAsAttribute 属性自定义参数的封送方式。

自定义字符串参数

.NET 具有各种可用来封送字符串的格式。这些方法分为有关 C 样式字符串和以 Windows 为中心的字符串的不同格式部分。

C 样式字符串

其中每个格式都会将以 null 结尾的字符串传递到本机代码。它们的差别在于本机字符串的编码。

`SYSTEM.RUNTIME.INTEROPSERVICES.UNMANAGEDTYPE` [	[[
LPStr	ANSI
LPUTF8Str	UTF-8
LPWStr	UTF-16
LPTStr	UTF-16

UnmanagedType.VBByRefStr 格式稍有不同。与 `LPWStr` 一样，该格式会将字符串封送到采用 UTF-16 编码的本机 C 样式字符串。但是，托管签名允许你按引用传入字符串，而匹配的本机签名会按值获取字符串。这一区别允许你使用按值获取字符串的本机 API，并对其进行就地修改而无需使用 `StringBuilder`。我们建议不要手动使用此格式，因为很容易导致与不匹配的本机和托管签名混淆。

以 Windows 为中心的字符串格式

与 COM 或 OLE 接口交互时，可能会发现本机函数将字符串用作 `BSTR` 参数。可以使用 UnmanagedType.BStr 非托管类型将字符串作为 `BSTR` 进行封送。

如果要与 WinRT API 交互，则可以使用 UnmanagedType.HString 格式将字符串作为 `HSTRING` 进行封送。

自定义数组参数

.NET 还提供多种方式来封送数组参数。如果要调用获取 C 样式数组的 API，则使用 UnmanagedType.LPArray 非托管类型。如果数组中的值需要进行自定义封送，则可以为其使用 `[MarshalAs]` 属性上的 ArraySubType 字段。

如果使用的是 COM API，则可能必须将数组参数作为 `SAFEARRAY*` 进行封送。为此，可以使用 UnmanagedType.SafeArray 非托管类型。自定义 `object` 字段上的表中显示了 `SAFEARRAY` 的元素的默认类型。可以使用 MarshalAsAttribute.SafeArraySubType 和 MarshalAsAttribute.SafeArrayUserDefinedSubType 字段自定义 `SAFEARRAY` 的确切元素类型。

自定义布尔或十进制参数

有关封送布尔或十进制参数的信息，请参阅自定义结构封送。

自定义对象参数（仅适用于 Windows）

在 Windows 上，.NET 运行时提供多种不同方式，将对象参数封送到本机代码。

作为特定的 COM 接口进行封送

如果 API 使用指向 COM 对象的指针，则可以使用类型为 `object` 的参数上的以下任一 `UnmanagedType` 格式告知 .NET 作为这些特定接口进行封送：

- `IUnknown`
- `IDispatch`
- `IInspectable`

此外，如果类型标记为 `[ComVisible(true)]`，或如果要封送 `object` 类型，则可以使用 UnmanagedType.Interface 格式将对象作为类型的 COM 视图的 COM 可调用包装器进行封送。

封送到 `VARIANT`

如果本机 API 采用 Win32 `VARIANT`，则可以使用 `object` 参数上的 UnmanagedType.Struct 格式将对象作为 `VARIANT` 进行封送。如需了解 .NET 类型与 `VARIANT` 类型之间的映射，请参阅有关自定义 `object` 字段的文档。

自定义封送处理程序

如果要将本机 COM 接口投影到其他托管类型，则可以使用 `UnmanagedType.CustomMarshaler` 格式和 ICustomMarshaler 的实现来提供自己的自定义封送处理代码。

本机互操作性最佳做法

2021/11/16 •

.NET 提供了多种方式来自定义本机互操作性代码。本文包括 Microsoft .NET 团队为实现本机互操作性而遵循的指南。

通用指南

本部分中的指南适用于所有互操作方案。

- ✔ 请务必对方法和参数使用同一命名和大小写以作为要调用的本机方法。
- ✔ 请考虑对常数值使用同一命名和大小写。
- ✔ 请务必使用映射到最接近本机类型的 .NET 类型。例如，在 C# 中，当本机类型为 `unsigned int` 时使用 `uint`。
- ✔ 请务必在所需行为与默认行为不同时仅使用 `[In]` 和 `[Out]` 属性。
- ✔ 请考虑使用 System.Buffers.ArrayPool<T> 来汇集本机数组缓冲区。
- ✔ 请考虑使用与本机库相同的名称和大小写包装类中的 P/Invoke 声明。
  - 这允许 `[DllImport]` 属性使用 C# `nameof` 语言功能传入本机库的名称，并确保本机库的名称拼写正确。

DllImport 属性设置

PreserveSig	`true`	保留默认设置	将其显式设置为 False 时，失败的 HRESULT 返回值将变为异常（因此，定义中的返回值将变为 Null）。
SetLastError	`false`	取决于 API	如果 API 使用 GetLastError，并使用 Marshal.GetLastWin32Error 获取值，则将其设置为 True。如果 API 设置一个表示其有错误的条件，则在进行其他调用之前获取错误以避免无意覆盖该错误。
CharSet	`CharSet.None`，这会退回到 `CharSet.Ansi` 行为	定义中存在字符串或字符时显式使用 `CharSet.Unicode` 或 `CharSet.Ansi`	这将指定字符串的封送行为以及为 `false` 时 `ExactSpelling` 的操作。请注意，`CharSet.Ansi` 在 Unix 上实际为 UTF8。大部分时间，Windows 使用 Unicode，而 Unix 使用 UTF8。有关更多信息，请查看有关字符集的文档。

ExactSpelling	`false`	`true`	将其设置为 True 并在运行时获得些许性能优势不会查找后缀为“A”或“W”的备用函数名称，具体取决于 `CharSet` 设置的值（“A”用于 `CharSet.Ansi`，“W”用于 `CharSet.Unicode`）。

字符串参数

当字符集为 Unicode 或参数显式标记为 `[MarshalAs(UnmanagedType.LPWSTR)]`，并且通过值（不是 `ref` 或 `out`）传递字符串时，将固定该字符串并直接由本机代码使用（而非复制）。

除非明确想要对字符串进行 ANSI 处理，否则请务必将 `[DllImport]` 标记为 `Charset.Unicode`。

✕ 不使用 `[Out] string` 参数。如果该字符串为暂存的字符串，则通过包含 `[Out]` 属性的值传递的字符串参数可能使运行时变得不稳定。请在 String.Intern 的文档中查看有关字符串暂存的详细信息。

✕ 避免使用 `StringBuilder` 参数。`StringBuilder` 封送*始终*创建本机缓冲区副本。因此，该操作的效率可能非常低。采取调用带有字符串的 Windows API 的典型方案：

1. 创建所需容量的 SB（分配托管容量）{1}
2. 调用
   a. 分配本机缓冲区 {2}
   b. 如果为 `[In]`（`StringBuilder` 参数的默认值），则复制内容
   c. 如果为 `[Out]` {3}（也是 `StringBuilder` 的默认值），则将本机缓冲区复制到新分配的托管数组中
3. `ToString()` 分配其他托管数组 {4}

这是 {4} 分配，可从本机代码中获取字符串。可用来限制此操作的最佳方法是在其他调用中重用 `StringBuilder`，但这仍只能保存 1 个分配。最好从 `ArrayPool` 使用并缓存字符缓冲区 - 然后可以在后续调用中直接获得 `ToString()` 的分配。

`StringBuilder` 的另一个问题是它始终会将返回缓冲区备份复制到第一个 Null。如果传递的返回字符串未终止或为双 Null 终止字符串，则 P/Invoke 很可能不正确。

如果使用 `StringBuilder`，则最后一个问题是容量确实不会包括隐藏的 Null，该值始终计入互操作。人们常常会犯这个错误，因为大多数 API 希望缓冲区的大小包括 Null。这可能会导致产生浪费/不必要的分配。此外，此问题会阻止运行时通过优化 `StringBuilder` 封送来最大限度地减少副本。

✔ 请考虑使用 `ArrayPool` 中的 `char[]`。

有关字符串封送的详细信息，请参阅字符串的默认封送和自定义字符串封送。

> 特定于 Windows 对于 `[Out]` 字符串，CLR 将默认使用 `CoTaskMemFree` 来释放字符串，或对于标记为 `UnmanagedType.BSTR` 的字符串，使用 `SysStringFree`。对于具有输出字符串缓冲区的大多数 API：传入的字符计数必须包括 Null。如果返回的值小于传入的字符计数，则调用成功，并且该值是不带尾随 Null 的字符数。否则，该计数是包括 Null 字符的缓冲区的所需大小。
>
> - 传入 5 个，获取 4 个：字符串包含 4 个字符，带有尾随 Null。
> - 传入 5 个，获取 6 个：字符串包含 5 个字符，需要包含 6 个字符的缓冲区来保存 Null。字符串的 Windows 数据类型

布尔参数和字段

布尔值很容易混淆。默认情况下，将 .NET `bool` 封送到 Windows `BOOL`，它在其中为包含 4 个字节的值。但

是，C 和 C++ 中的 `_Bool` 和 `bool` 类型是单 字节。这可能会导致难以跟踪 bug，因为一半的返回值将被丢弃，这样 可能只会更改结果。有关将 .NET `bool` 值封送到 C 或 C++ `bool` 类型的详细信息，请参阅有关自定义布尔字段封送的文档。

GUID

GUID 可在签名中直接使用。许多 Windows API 使用 `GUID&` 类型别名（例如，`REFIID`）。通过引用传递时，可以通过 `ref` 或使用 `[MarshalAs(UnmanagedType.LPStruct)]` 属性传递。

GUID	ıtttttt GUID
`KNOWNFOLDERID`	`REFKNOWNFOLDERID`

✕ 请勿对除 `ref` GUID 参数以外的任何参数使用 `[MarshalAs(UnmanagedType.LPStruct)]`。

Blittable 类型

Blittable 类型是托管代码和本机代码中具有相同位级别表示形式的类型。因此，无需将这些类型转换为其他格式即可往返本机代码进行封送，而且由于这样可以提高性能，应首选这些类型。

：

- `byte`，`sbyte`，`short`，`ushort`，`int`，`uint`，`long`，`ulong`，`single`，`double`
- Blittable 类型的非嵌套一维数组（例如，`int[]`）
- 具有实例字段只有 blittable 值类型的固定布局的结构和类
  - 固定的布局需要 `[StructLayout(LayoutKind.Sequential)]` 或 `[StructLayout(LayoutKind.Explicit)]`
  - 默认情况下结构为 `LayoutKind.Sequential`，类为 `LayoutKind.Auto`

不是 blittable：

- `bool`

有时为 blittable：

- `char`，`string`

通过引用传递 blittable 类型时，这些类型只会被封送处理程序固定，而不会复制到中间缓冲区。（类在本质上通过引用传递，结构在与 `ref` 或 `out` 结合使用时会通过引用传递。）

如果 `char` 位于一维数组中，或者如果它是包含使用 `CharSet = CharSet.Unicode` 的 `[StructLayout]` 显式标记的类型的一部分，则该类型为 blittable。

```
[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
public struct UnicodeCharStruct
{
    public char c;
}
```

如果 `string` 不包含在其他类型中，并且作为使用 `[MarshalAs(UnmanagedType.LPWStr)]` 标记的参数传递或 `[DllImport]` 已设置 `CharSet = CharSet.Unicode`，则该类型为 blittable。

可以通过尝试创建固定的 `GCHandle` 来查看类型是否为 blittable。如果该类型不是字符串或被视为 blittable，则 `GCHandle.Alloc` 将引发 `ArgumentException`。

✔ 尽可能使结构为 blittable。

有关详细信息，请参见：

- 可直接复制到本机结构中的类型和非直接复制到本机结构中的类型
- 类型封送

使托管对象保持活动状态

`GC.KeepAlive()` 将确保对象保持在作用域内，直到采用 KeepAlive 方法。

`HandleRef` 允许封送处理程序在 P/Invoke 的持续时间内使对象保持活动状态。方法签名中可以使用该类型，而不是 `IntPtr`。`SafeHandle` 可有效地替换此类，应改为使用此类型。

`GCHandle` 允许固定托管对象和获取指向该类型的本机指针。基本模式是：

```
GCHandle handle = GCHandle.Alloc(obj, GCHandleType.Pinned);
IntPtr ptr = handle.AddrOfPinnedObject();
handle.Free();
```

固定不是 `GCHandle` 的默认设置。其他主要模式是通过本机代码将引用传递到托管对象并返回到托管代码（通常使用回调）。模式如下：

```
GCHandle handle = GCHandle.Alloc(obj);
SomeNativeEnumerator(callbackDelegate, GCHandle.ToIntPtr(handle));

// In the callback
GCHandle handle = GCHandle.FromIntPtr(param);
object managedObject = handle.Target;

// After the last callback
handle.Free();
```

请务必注意需要显式释放 `GCHandle` 以避免内存泄漏。

常见的 Windows 数据类型

下面是 Windows API 中常用的数据类型列表以及调用 Windows 代码时要使用的 C# 类型。

以下类型在 32 位和 64 位 Windows 上具有相同大小，而不管其名称为何。

位	WINDOWS	C (WINDOWS)	C#	注
32	`BOOL`	`int`	`int`	`bool`
8	`BOOLEAN`	`unsigned char`	`byte`	`[MarshalAs(UnmanagedType.U1)] bool`
8	`BYTE`	`unsigned char`	`byte`	
8	`CHAR`	`char`	`sbyte`	
8	`UCHAR`	`unsigned char`	`byte`	
16	`SHORT`	`short`	`short`	
16	`CSHORT`	`short`	`short`	
16	`USHORT`	`unsigned short`	`ushort`	

位	Windows	C (Windows)	C#	备注
16	`WORD`	`unsigned short`	`ushort`	
16	`ATOM`	`unsigned short`	`ushort`	
32	`INT`	`int`	`int`	
32	`LONG`	`long`	`int`	
32	`ULONG`	`unsigned long`	`uint`	
32	`DWORD`	`unsigned long`	`uint`	
64	`QWORD`	`long long`	`long`	
64	`LARGE_INTEGER`	`long long`	`long`	
64	`LONGLONG`	`long long`	`long`	
64	`ULONGLONG`	`unsigned long long`	`ulong`	
64	`ULARGE_INTEGER`	`unsigned long long`	`ulong`	
32	`HRESULT`	`long`	`int`	
32	`NTSTATUS`	`long`	`int`	

以下类型(指针)遵循平台的宽度。对其使用 `IntPtr` / `UIntPtr`。

句柄(位 `INTPTR`)	指针(位 `UINTPTR`)
`HANDLE`	`WPARAM`
`HWND`	`UINT_PTR`
`HINSTANCE`	`ULONG_PTR`
`LPARAM`	`SIZE_T`
`LRESULT`	
`LONG_PTR`	
`INT_PTR`	

Windows `PVOID`，这是一个 C `void*`，可以作为 `IntPtr` 或 `UIntPtr` 进行封送，但在可能的情况下更倾向于 `void*`。

Windows 数据类型

数据类型范围

结构

托管结构是在堆栈上创建的，在返回方法之前不会将其删除。按照定义，它们是“固定的”（不会被 GC 移动）。如果本机代码不会在当前方法末尾之外使用指针，则也可以使用不安全代码块中的地址。

Blittable 结构的性能更好，因为它们可以由封送层直接使用。尝试使结构为 blittable（例如，避免 `bool`）。有关详细信息，请参阅 Blittable 类型部分。

如果结构为 blittable，请使用 `sizeof()` 而不是 `Marshal.SizeOf<MyStruct>()`，以获得更好的性能。如上所述，可以通过尝试创建固定的 `GCHandle` 来验证该类型是否为 blittable。如果该类型不是字符串或被视为 blittable，则 `GCHandle.Alloc` 将引发 `ArgumentException`。

指向定义中的结构的指针必须通过 `ref` 传递或使用 `unsafe` 和 `*`。

✔ 请尽可能将托管结构与官方平台文档或标题中使用的形状和名称匹配。

✔ 请务必使用 C# `sizeof()` 而不是 blittable 结构的 `Marshal.SizeOf<MyStruct>()`，以提高性能。

✕ 避免使用 `System.Delegate` 或 `System.MulticastDelegate` 字段来表示结构中的函数指针字段。

由于 System.Delegate 和 System.MulticastDelegate 没有必需的签名，因此它们不能保证传入的委托将与本机代码所需的签名匹配。此外，在 .NET Framework 和 .NET Core 中，如果本机表示形式的字段值不是包装托管委托的函数指针，则将包含 `System.Delegate` 或 `System.MulticastDelegate` 的结构从其本机表示形式封送到托管对象这一操作可能会导致运行时不稳定。在 .NET 5 及更高版本中，不支持将 `System.Delegate` 或 `System.MulticastDelegate` 字段从本机表示形式封送到托管对象。使用特定委托类型，而不是 `System.Delegate` 或 `System.MulticastDelegate`。

固定缓冲区

`INT_PTR Reserved1[2]` 等数组必须封送到两个 `IntPtr` 字段（`Reserved1a` 和 `Reserved1b`）。当本机数组为基元类型时，可以使用 `fixed` 关键字更明确地进行编写。例如，`SYSTEM_PROCESS_INFORMATION` 在本机标头中类似如下内容：

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG NextEntryOffset;
    ULONG NumberOfThreads;
    BYTE Reserved1[48];
    UNICODE_STRING ImageName;
...
} SYSTEM_PROCESS_INFORMATION
```

可以在 C# 中编写如下内容：

```
internal unsafe struct SYSTEM_PROCESS_INFORMATION
{
    internal uint NextEntryOffset;
    internal uint NumberOfThreads;
    private fixed byte Reserved1[48];
    internal Interop.UNICODE_STRING ImageName;
    ...
}
```

但是，固定的缓冲区有一些问题。非 blittable 类型的固定缓冲区不会正确封送，因此，就地数组需要扩大到多个单独字段。此外，在早于 3.0 的 .NET Framework 和 .NET Core 中，如果包含固定缓冲区字段的结构嵌套在非 blittable 结构中，则不会将固定缓冲区字段正确封送到本机代码。

字符集和封送

2021/11/16 •

`char` 值、`string` 对象和 `System.Text.StringBuilder` 对象的封送方式取决于 P/Invoke 或结构上的 `CharSet` 字段的值。可以通过在声明 P/Invoke 时设置 DllImportAttribute.CharSet 字段来设置 P/Invoke 的 `CharSet`。若要设置 `CharSet` 类型，请在类或结构声明中设置 StructLayoutAttribute.CharSet 字段。未设置这些属性字段时，将由语言编译器确定使用哪些 `CharSet`。C# 和 Visual Basic 默认使用 Ansi 字符集。

下表显示了每个字符集之间的映射以及字符或字符串在使用该字符集封送时的表示形式：

CHARSET	WINDOWS	UNIX 上的 .NET CORE 2.2 及更高版本	UNIX 上的 .NET CORE 3.0 及更高版本和 MONO
`Ansi`	`char`（系统默认的 Windows (ANSI) 代码页）	`char` (UTF-8)	`char` (UTF-8)
`Unicode`	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char16_t` (UTF-16)
`Auto`	`wchar_t` (UTF-16)	`char16_t` (UTF-16)	`char` (UTF-8)

确保了解选择字符集时所需的本机表示形式。

向 COM 公开 .NET Core 组件

2021/11/16 •

在 .NET Core 中，与 .NET Framework 相比，向 COM 公开 .NET 对象的过程已明显简化。下面的过程将引导你如何向 COM 公开类。本教程介绍了如何：

- 从 .NET Core 向 COM 公开类。
- 生成 COM 服务器作为构建 .NET Core 库的一部分。
- 自动为无注册表 COM 生成并行服务器清单。

先决条件

- 安装 .NET Core 3.0 SDK 或更高版本。

创建库

第一步是创建库。

1. 创建新文件夹，并在该文件夹中运行以下命令：

```
dotnet new classlib
```

2. 打开 `Class1.cs`。
3. 将 `using System.Runtime.InteropServices;` 添加到文件顶部。
4. 创建名为 `IServer` 的接口。例如：

```
using System;
using System.Runtime.InteropServices;

[ComVisible(true)]
[Guid(ContractGuids.ServerInterface)]
[InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
public interface IServer
{
    /// <summary>
    /// Compute the value of the constant Pi.
    /// </summary>
    double ComputePi();
}
```

5. 将 `[Guid("<IID>")]` 属性添加到接口，包含要实现的 COM 接口的接口 GUID。例如 `[Guid("fe103d6e-e71b-414c-80bf-982f18f6c1c7")]`。请注意，此 GUID 必须唯一，因为它是 COM 的此接口的唯一标识符。在 Visual Studio 中，可通过转到“工具”>“创建 GUID”以打开“创建 GUID”工具来生成 GUID。
6. 将 `[InterfaceType]` 属性添加到接口，并指定接口应实现的基本 COM 接口。
7. 创建用于实现 `IServer` 的名为 `Server` 的类。
8. 将 `[Guid("<CLSID>")]` 属性添加到类，包含要实现的 COM 类的类标识符 GUID。例如 `[Guid("9f35b6f5-2c05-4e7f-93aa-ee087f6e7ab6")]`。与接口 GUID 一样，此 GUID 必须唯一，因为它是 COM

的此接口的唯一标识符。

9. 将 `[ComVisible(true)]` 属性添加到接口和类。

> **IMPORTANT**
> 与 .NET Framework 不同，.NET Core 要求指定想要通过 COM 激活的任何类的 CLSID。

生成 COM 主机

1. 打开 `.csproj` 项目文件并在 `<PropertyGroup></PropertyGroup>` 标记中添加 `<EnableComHosting>true</EnableComHosting>`。
2. 生成项目。

生成的输出将具有 `ProjectName.dll`、`ProjectName.deps.json`、`ProjectName.runtimeconfig.json` 和 `ProjectName.comhost.dll` 文件。

为 COM 注册 COM 主机

打开提升的命令提示符，然后运行 `regsvr32 ProjectName.comhost.dll`。这将使用 COM 注册所有公开的 .NET 对象。

启用 RegFree COM

1. 打开 `.csproj` 项目文件并在 `<PropertyGroup></PropertyGroup>` 标记中添加 `<EnableRegFreeCom>true</EnableRegFreeCom>`。
2. 生成项目。

生成的输出现在还将具有 `ProjectName.X.manifest` 文件。此文件是用于无注册表的 COM 的并行清单。

在 COM 主机中嵌入类型库

与 .NET Framework 不同，.NET Core 或 .NET 5+ 中不支持从 .NET 程序集生成 COM 类型库 (TLB)。本指南旨在说明如何为 COM 接口的本机声明手动编写 IDL 文件或 C/C++ 标头。如果决定编写 IDL 文件，可使用 Visual C++ SDK 的 MIDL 编译器对其进行编译来生成 TLB。

在 .NET 6 预览版 5 及更高版本中，.NET SDK 支持在项目生成过程中将已编译的 TLB 嵌入 COM 主机。

若要在应用程序中嵌入类型库，请执行以下步骤：

1. 打开 `.csproj` 项目文件并在 `<ItemGroup></ItemGroup>` 标记中添加 `<ComHostTypeLibrary Include="path/to/typelib.tlb" Id="<id>" />`。
2. 将 `<id>` 替换为正整数值。在指定要嵌入 COM 主机的 TLB 中，该值必须是唯一的。
   - 如果你只将一个 `Id` 添加到项目中，则 `ComHostTypeLibrary` 属性是可选的。

例如，以下代码块将索引为 `1` 的 `Server.tlb` 类型库添加到 COM 主机：

```
<ItemGroup>
    <ComHostTypeLibrary Include="Server.tlb" Id="1" />
</ItemGroup>
```

示例

GitHub 上的 dotnet/samples 存储库中有一个正常运行的 COM 服务器示例。

附加说明

> **IMPORTANT**
>
> 在 .NET Framework 中，32 位和 64 位客户端可以使用“任何 CPU”程序集。默认情况下，在 .NET Core、.NET 5 和更高版本中，“任何 CPU”程序集附带了 64 位的 *.comhost.dll。因此，它们只能由 64 位客户端使用。这是默认的，因为这是 SDK 表示的内容。此行为与发布“自包含”功能的方式相同：默认情况下，它使用 SDK 提供的内容。
>
> `NETCoreSdkRuntimeIdentifier` MSBuild 属性确定 *.comhost.dll 的位数。正如预期的那样，托管部分的位数实际上是不可知的，而随附的本机资产默认为目标 SDK。

在激活期间，包含 COM 组件的程序集将基于程序集路径加载到单独的 AssemblyLoadContext 中。如果有一个程序集提供多个 COM 服务器，则重用 `AssemblyLoadContext`，以便该程序集中的所有服务器都位于同一加载上下文中。如果有多个程序集提供 COM 服务器，则将为每个程序集创建一个新的 `AssemblyLoadContext`，并且每个服务器位于其程序集所对应的加载上下文中。

不支持 COM 组件的自包含部署。仅支持 COM 组件的依赖框架的部署。

此外，将 .NET Framework 和 .NET Core 同时加载到同一进程具有诊断限制。主要限制是调试托管组件，因为不能同时调试 .NET Framework 和 .NET Core。此外，这两个运行时实例不共享托管程序集。这意味着无法在两个运行时之间共享实际的 .NET 类型，所有交互必须仅限于公开的 COM 接口协定。

编写自定义 .NET 主机以从本机代码控制 .NET 运行时

2021/11/16 •

像所有的托管代码一样，.NET 应用程序也由主机执行。主机负责启动运行时（包括 JIT 和垃圾回收器等组件）和调用托管的入口点。

托管 .NET 运行时是高级方案，在大多数情况下，.NET 开发人员无需担心托管问题，因为 .NET 生成过程会提供默认主机来运行 .NET 应用程序。虽然在某些特殊情况下，它对显式托管 .NET 运行时非常有用 - 无论是作为一种在本机进程中调用托管代码的方式还是为了获得对运行时工作原理更好的控制。

本文概述了从本机代码启动 .NET 运行时和在其中执行托管代码的必要步骤。

先决条件

由于主机是本机应用程序，所以本教程将介绍如何构造 C++ 应用程序以托管 .NET。将需要一个 C++ 开发环境（例如，Visual Studio 提供的环境）。

还需要生成 .NET 组件以用来测试主机，因此你应该安装 .NET SDK。

承载 API

在 .NET Core 3.0 及更高版本中托管 .NET 运行时是通过 `nethost` 和 `hostfxr` 库的 API 完成的。由这些入口点来处理查找和设置运行时进行初始化所遇到的复杂性；通过它们，还可启动托管应用程序和调用静态托管方法。

在 .NET Core 3.0 之前，托管运行时的唯一选项是通过 `coreclrhost.h` API。此托管 API 现已过时，不应用于托管 .NET Core 3.0 和更高版本的运行时。

使用 `nethost.h` 和 `hostfxr.h` 创建主机

有关展示以下教程中所述的步骤的示例主机，请访问 dotnet/samples GitHub 存储库。该示例中的注释清楚地将本教程中已编号的步骤与它们在示例中的执行位置关联。有关下载说明，请参阅示例和教程。

请记住，示例主机的用途在于提供学习指导，在纠错方面不甚严谨，其重在可读性而非效率。

以下步骤详细说明如何使用 `nethost` 和 `hostfxr` 库在本机应用程序中启动 .NET 运行时并调用托管静态方法。示例使用了随 .NET SDK 一起安装的 `nethost` 标头和库，以及 dotnet/runtime 存储库中的 `coreclr_delegates.h` 和 `hostfxr.h` 文件的副本。

步骤 1 - 加载 `hostfxr` 并获取导出的托管函数

`nethost` 库提供用于查找 `hostfxr` 库的 `get_hostfxr_path` 函数。`hostfxr` 库公开用于托管 .NET 运行时的函数。函数的完整列表可在 `hostfxr.h` 和本机托管设计文档中找到。示例和本教程使用以下函数：

- `hostfxr_initialize_for_runtime_config`：初始化主机上下文，并使用指定的运行时配置准备初始化 .NET 运行时。
- `hostfxr_get_runtime_delegate`：获取对运行时功能的委托。
- `hostfxr_close`：关闭主机上下文。

使用 `nethost` 库中的 `get_hostfxr_path` API 找到了 `hostfxr` 库。随后加载此库并检索其导出。

```
// Using the nethost library, discover the location of hostfxr and get exports
bool load_hostfxr()
{
    // Pre-allocate a large buffer for the path to hostfxr
    char_t buffer[MAX_PATH];
    size_t buffer_size = sizeof(buffer) / sizeof(char_t);
    int rc = get_hostfxr_path(buffer, &buffer_size, nullptr);
    if (rc != 0)
        return false;

    // Load hostfxr and get desired exports
    void *lib = load_library(buffer);
    init_fptr = (hostfxr_initialize_for_runtime_config_fn)get_export(lib,
"hostfxr_initialize_for_runtime_config");
    get_delegate_fptr = (hostfxr_get_runtime_delegate_fn)get_export(lib, "hostfxr_get_runtime_delegate");
    close_fptr = (hostfxr_close_fn)get_export(lib, "hostfxr_close");

    return (init_fptr && get_delegate_fptr && close_fptr);
}
```

步骤 2 - 初始化和启动 .NET 运行时

`hostfxr_initialize_for_runtime_config` 和 `hostfxr_get_runtime_delegate` 函数使用将加载的托管组件的运行时配置初始化并启动 .NET 运行时。`hostfxr_get_runtime_delegate` 函数用于获取运行时委托，允许加载托管程序集并获取指向该程序集中的静态方法的函数指针。

```
// Load and initialize .NET Core and get desired function pointer for scenario
load_assembly_and_get_function_pointer_fn get_dotnet_load_assembly(const char_t *config_path)
{
    // Load .NET Core
    void *load_assembly_and_get_function_pointer = nullptr;
    hostfxr_handle cxt = nullptr;
    int rc = init_fptr(config_path, nullptr, &cxt);
    if (rc != 0 || cxt == nullptr)
    {
        std::cerr << "Init failed: " << std::hex << std::showbase << rc << std::endl;
        close_fptr(cxt);
        return nullptr;
    }

    // Get the load assembly function pointer
    rc = get_delegate_fptr(
        cxt,
        hdt_load_assembly_and_get_function_pointer,
        &load_assembly_and_get_function_pointer);
    if (rc != 0 || load_assembly_and_get_function_pointer == nullptr)
        std::cerr << "Get delegate failed: " << std::hex << std::showbase << rc << std::endl;

    close_fptr(cxt);
    return (load_assembly_and_get_function_pointer_fn)load_assembly_and_get_function_pointer;
}
```

步骤 3 - 加载托管程序集并获取指向托管方法的函数指针

将调用运行时委托以加载托管程序集并获取指向托管方法的函数指针。委托需要程序集路径、类型名称和方法名称作为输入，并返回可用于调用托管方法的函数指针。

```
// Function pointer to managed delegate
component_entry_point_fn hello = nullptr;
int rc = load_assembly_and_get_function_pointer(
    dotnetlib_path.c_str(),
    dotnet_type,
    dotnet_type_method,
    nullptr /*delegate_type_name*/,
    nullptr,
    (void**)&hello);
```

该示例通过在调用运行时委托时将 `nullptr` 作为委托类型名称传递，对托管方法使用默认签名：

```
public delegate int ComponentEntryPoint(IntPtr args, int sizeBytes);
```

可以通过在调用运行时委托时指定委托类型名称来使用其他签名。

步骤 4 - 运行托管代码！

本机主机现在可以调用托管方法，并向其传递所需的参数。

```
lib_args args
{
    STR("from host!"),
    i
};

hello(&args, sizeof(args));
```

.NET 中的 COM 互操作

2021/11/16 •

组件对象模型 (COM) 允许对象向其他组件公开其功能并在 Windows 平台上托管应用程序。为帮助用户实现与其现有代码库的互操作，.NET Framework 始终为与 COM 库进行互操作提供强大支持。在 .NET Core 3.0 中，此支持中的很大一部分已添加到 Windows 上的 .NET Core。此处的文档说明了公共 COM 互操作技术的工作原理，以及如何利用它们与现有 COM 库进行互操作。

- COM 包装
- COM 可调用包装器
- 运行时可调用包装器
- 为 COM 互操作限定 .NET 类型

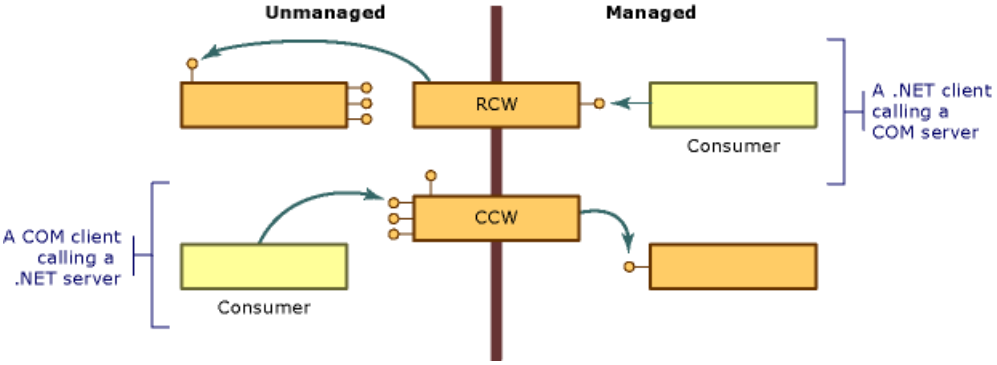
COM 包装

2021/11/16 •

COM 在以下几个重要方面与 .NET 运行时对象模型存在差别：

- COM 对象的客户端必需管理这些对象的生存期；公共语言运行时管理其环境中对象的生存期。
- COM 对象的客户端通过对提供服务的接口发出请求并取回接口指针来发现服务是否可用。.NET 对象的客户端可以使用反射来获取对象功能的说明。
- NET 对象位于由 .NET 运行时执行环境管理的内存中。出于性能原因，执行环境可以在内存中来回移动对象，并更新所移动对象的所有引用。非管理的客户端在获取指向对象的指针后，依赖于该对象以保留在相同位置。这些客户端没有机制来处理位置不固定的对象。

为了克服这些差异，运行时提供了包装类，使托管和非管理的客户端认为它们在各自的环境中调用对象。每当托管客户端对某个 COM 对象调用方法时，运行时就会创建一个运行时可调用包装器 (RCW)。除此之外，RCW 还会提取托管和非托管引用机制之间的差异。该运行时还创建了一个 COM 可调用包装器 (CCW) 来逆转此过程，使 COM 客户端能够对 .NET 对象无缝地调用方法。如下图所示，调用代码的性质确定运行时创建的包装类。



大多数情况下，运行时生成的标准 RCW 或 CCW 都可为跨 COM 和 .NET 运行时之间边界的调用提供充分的封送处理。使用自定义属性，可以选择性地调整运行时表示托管和非托管代码的方式。

请参阅

- .NET Framework 中的高级 COM 互操作性
- 运行时可调用包装器
- COM 可调用包装器
- 自定义 .NET Framework 中的标准包装器
- 如何：自定义 .NET Framework 中的运行时可调用包装器

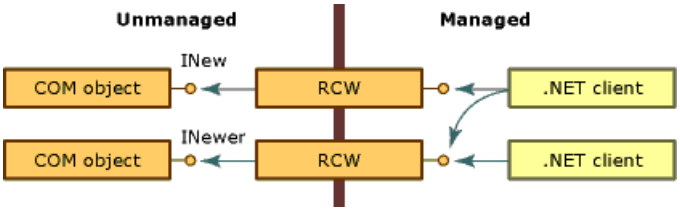
运行时可调用包装

2021/11/16 •

公共语言运行时通过名为运行时可调用包装 (RCW) 的代理公开 COM 对象。尽管 RCW 似乎是 .NET 客户端的普通对象，但它的主要功能是封送处理 .NET 客户端和 COM 对象之间的调用。

无论 COM 对象上有多少引用数目，运行时都只为每个 COM 对象创建一个 RCW。运行时针对每个对象的每个进程维护一个 RCW。如果在某个应用程序域或单元创建一个 RCW，然后传递一个其他应用程序域或单元的引用，则将使用第一个对象的代理。如下图所示，任意数量的托管客户端都可拥有一个对公开 INew 和 INewer 接口的 COM 对象的引用。

下图显示了通过运行时可调用包装器访问 COM 对象的过程：



借助从类型库派生而来的元数据，运行时创建正在调用的 COM 对象及其包装。每个 RCW 都在其包装的 COM 对象上维护一个接口指针的缓存，并且当不再需要 RCW 时释放 COM 对象上的引用。运行时在 RCW 上执行垃圾回收。

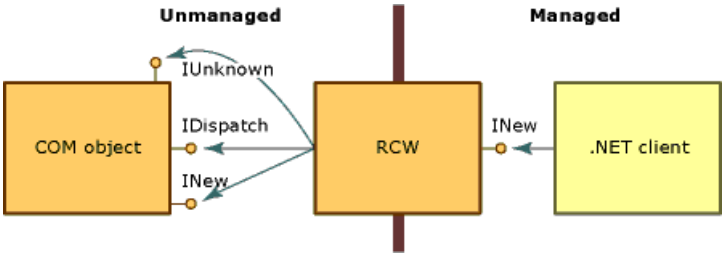
在其他活动中，RCW 代表包装的对象封送托管代码和非托管代码间的数据。具体而言，每当在客户端和服务器之间传递数据的不同表示形式时，RCW 都向方法自变量和方法返回值提供封送处理。

标准包装强制执行内置的封送处理规则。例如，当 .NET 客户端将 String 类型作为自变量的一部分传递到非托管对象时，包装会将字符串转换为 BSTR 类型。如果 COM 对象向它托管的调用方返回 BSTR，则调用方会收到 String。客户端和服务器都会发送和接收所熟悉的数据。其他类型不需要转换。例如，标准包装始终无需转换类型即可在托管代码和非托管代码间传递 4 字节整数。

封送处理所选接口

运行时可调用包装器 (RCW) 的主要目的是隐藏托管和非托管编程模型之间的差异。若要创建无缝转换，RCW 需使用选定的 COM 接口且不将其公开到 .NET 客户端，如下图所示。

下图显示了 COM 接口和运行时可调用包装器：



当创建为早期绑定对象时，RCW 为特定类型。它可实现 COM 对象实现的接口，并可公开对象接口中的方法、属性和事件。图示中，RCW 公开 INew 接口，但使用“IUnknown”和“IDispatch”接口 。此外，RCW 向 .NET 客户端公开 INew 接口的所有成员。

RCW 使用下表列出的接口，这些接口由其包装的对象公开。

“	“
IDispatch	用于通过反射后期绑定到 COM 对象。
IErrorInfo	提供以下内容的文字描述：错误、错误源、帮助文件，帮助上下文以及定义错误的接口的 GUID（.NET 类始终为 GUID_NULL）。
IProvideClassInfo	如果正在包装的 COM 对象实现 IProvideClassInfo，RCW 会提取此接口中的类型信息以提供更佳的类型标识。
IUnknown	针对对象标识、类型强制和生存期管理： - 对象标识 运行时通过比较每个对象的 IUnknown 接口的值来区分 COM 对象。 - 类型强制 RCW 识别由 QueryInterface 方法执行的动态类型发现。 - 生存期管理 借助 QueryInterface 方法，RCW 获取并保存对非托管对象的引用，直到运行时在包装器上执行会释放非托管对象的垃圾回收。

RCW 选择性地使用下表中列出的接口，这些接口由其包装的对象公开。

“	“
IConnectionPoint 和 IConnectionPointContainer	RCW 对向基于委托的事件公开连接点事件样式的对象执行转换。
IDispatchEx（仅限 .NET Framework）	如果类实现 IDispatchEx，则 RCW 实现 IExpando 。 IDispatchEx 接口是 IDispatch 接口的扩展，与 IDispatch 不同，它可枚举、添加、删除和以区分大小的方式调用成员 。
IEnumVARIANT	使支持枚举的 COM 类型可被视为集合。

请参阅

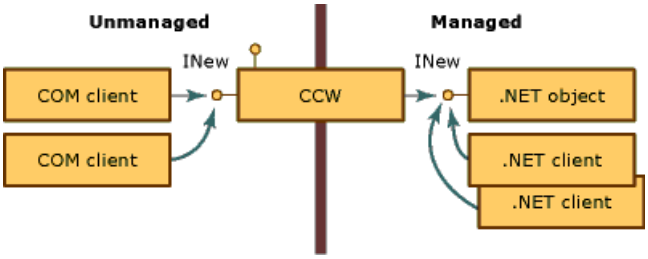
- COM 包装
- COM 可调用包装器
- 有关从类型库转换到程序集的摘要
- 将类型库作为程序集导入

COM 可调用包装

2021/11/16 •

COM 客户端调用 .NET 对象时，公共语言运行时将创建托管对象和该对象的 COM 可调用包装器 (CCW)。无法直接引用 .NET 对象，COM 客户端使用 CCW 作为托管对象的代理。

无论请求其服务的 COM 客户端数量是多少，该运行时都只为托管对象创建恰好一个 CCW。如下图所示，多个 COM 客户端可以保持对公开 INew 接口的 CCW 的引用。反之，CCW 保持对实现该接口的托管对象的单一引用，并被垃圾回收。COM 和 .NET 客户端可以同时对同一托管对象发出请求。



COM 可调用包装器对在 .NET 运行时内运行的其他类不可见。它们的主要目的是封送托管和非托管代码之间的调用；但是，CCW 还托管它们包装的托管对象的对象标识和对象生存期。

对象标识

运行时为其垃圾回收堆中的 .NET 对象分配内存，从而使运行时能根据需要在内存中移动对象。与此相反，运行时为非回收堆中的 CCW 分配内存，使 COM 客户端直接引用包装器成为可能。

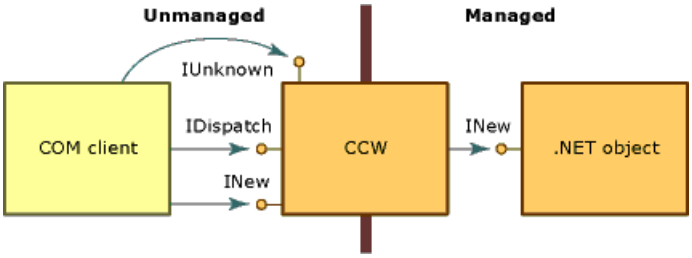
对象生存期

与其包装的 .NET 客户端不同，CCW 在传统 COM 方式中会进行引用计数。当 CCW 上的引用计数达到零时，包装器将释放其对托管对象的引用。将在下一个垃圾回收周期期间收集无剩余引用的托管对象。

模拟 COM 接口

CCW 公开所有公共的 COM 可见接口和数据类型，并以与 COM 对基于接口的交互的强制一致的方式向 COM 客户端返回值。对于 COM 客户端而言，调用 .NET 对象上的方法与调用 COM 对象上的方法相同。

若要创建这一无缝的方法，CCW 生成传统 COM 接口，如 IUnknown 和 IDispatch。如下图所示，CCW 保持对其包装的 .NET 对象的单一引用。COM 客户端和 .NET 对象都通过代理和 CCW 的存根构造进行相互交互。



除公开由托管环境中的类显式实现的接口外，.NET 运行时代表对象提供对下表中列出的 COM 接口的实现。.NET 类可以通过提供其自身对这些接口的实现而替代默认行为。但是，运行时始终提供 IUnknown 和 IDispatch 接口的实现。

“	“
IDispatch	为晚期绑定到类型提供一个机制。
IErrorInfo	提供以下内容的文字描述：错误、错误源、帮助文件，帮助上下文以及定义错误的接口的 GUID（.NET 类始终为 GUID_NULL）。
IProvideClassInfo	启用 COM 客户端，以获取对由托管类实现的 ITypeInfo 接口的访问。对于未从 COM 导入的类型，在 .NET Core 上返回 `COR_E_NOTSUPPORTED`。
ISupportErrorInfo	启用 COM 客户端，以确定托管对象是否支持 IErrorInfo 接口。如果支持，则启用客户端，以获取指向最新异常对象的指针。所有托管类型都支持 IErrorInfo 接口。
ITypeInfo（仅限 .NET Framework）	为类提供与 Tlbexp.exe 生成的类型信息完全相同的类型信息。
IUnknown	提供 IUnknown 接口的标准实现，COM 客户端使用该接口管理 CCW 的生存期并提供类型强制转换。

托管类还可以提供下表中介绍的 COM 接口。

“	“
(_classname) 类接口	该接口由运行时公开但未显式定义，它公开托管对象上显式公开的所有公共接口、方法、属性和字段。
IConnectionPoint 和 IConnectionPointContainer	以基于委托的事件（用于注册事件订阅服务器的接口）为源的对象的接口。
IDispatchEx（仅限 .NET Framework）	如果类实现 IExpando，则为由运行时提供的接口。IDispatchEx 接口是 IDispatch 接口的扩展，与 IDispatch 不同，它可枚举、添加、删除和以区分大小的方式调用成员 。
IEnumVARIANT	集合类型类的接口，如果类实现 IEnumerable，则该接口将枚举集合中的对象。

类接口简介

类接口，未在托管代码中显式定义，是公开 .NET 对象中显式公开的所有公共方法、属性、字段和事件的接口。此接口可以是双重接口或仅支持调度的接口。类接口接收 .NET 类本身的名称（前面带下划线）。例如，对于类 Mammal，类接口为 _Mammal。

对于派生类，类接口也公开基类的所有公共方法、属性和字段。派生类还公开各基类的类接口。例如，如果类 Mammal 扩展类 MammalSuperclass（MammalSuperclass 类本身扩展 System.Object），则 .NET 对象向 COM 客户端公开名为 _Mammal、_MammalSuperclass 和 _Object 的三个类接口。

例如，请考虑以下 .NET 类：

```
' Applies the ClassInterfaceAttribute to set the interface to dual.
<ClassInterface(ClassInterfaceType.AutoDual)> _
' Implicitly extends System.Object.
Public Class Mammal
    Sub Eat()
    Sub Breathe()
    Sub Sleep()
End Class
```

```
// Applies the ClassInterfaceAttribute to set the interface to dual.
[ClassInterface(ClassInterfaceType.AutoDual)]
// Implicitly extends System.Object.
public class Mammal
{
    public void Eat() {}
    public void Breathe() {}
    public void Sleep() {}
}
```

COM 客户端可以获取指向名为 `_Mammal` 的类接口的指针。在 .NET Framework 上，可使用 类型库导出程序 (Tlbexp.exe) 工具生成包含 `_Mammal` 接口定义的类型库。.NET Core 不支持类型库导出程序。如果 `Mammal` 类实现了一个或多个接口，则这些接口将出现在组件类下。

```
[odl, uuid(…), hidden, dual, nonextensible, oleautomation]
interface _Mammal : IDispatch
{
    [id(0x00000000), propget] HRESULT ToString([out, retval] BSTR*
        pRetVal);
    [id(0x60020001)] HRESULT Equals([in] VARIANT obj, [out, retval]
        VARIANT_BOOL* pRetVal);
    [id(0x60020002)] HRESULT GetHashCode([out, retval] short* pRetVal);
    [id(0x60020003)] HRESULT GetType([out, retval] _Type** pRetVal);
    [id(0x6002000d)] HRESULT Eat();
    [id(0x6002000e)] HRESULT Breathe();
    [id(0x6002000f)] HRESULT Sleep();
}
[uuid(…)]
coclass Mammal
{
    [default] interface _Mammal;
}
```

生成类接口是可选操作。默认情况下，COM 互操作会为你导出到类型库中的每个类生成仅支持调度的接口。可以通过将 ClassInterfaceAttribute 应用到你的类来阻止或修改此接口的自动创建。尽管类接口可以减轻向 COM 公开托管类的任务，但其用途相当有限。

Caution

使用类接口（而不是显式定义你自己的类接口）可以增加托管类的未来版本控制的复杂性。请在使用类接口之前阅读以下指南。

定义要使用的 COM 客户端的显式接口，而非生成类接口。

由于 COM 互操作会自动生成类接口，对类进行的后期版本更改可改变由公共语言运行时公开的类接口的布局。由于 COM 客户端通常没有准备处理接口布局中的更改，因此如果更改类的成员布局，它们将发生中断。

本指南强调了向 COM 客户端公开的接口必须保持不变这一概念。若要降低因对接口布局无意地重新排序而中断 COM 客户端的风险，请通过显式定义接口从接口布局中隔离对类的所有更改。

使用 ClassInterfaceAttribute 来取消类接口的自动生成，并实现类的显式接口，如以下代码片段所示：

```
<ClassInterface(ClassInterfaceType.None)>Public Class LoanApp
    Implements IExplicit
    Sub M() Implements IExplicit.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.None)]
public class LoanApp : IExplicit
{
    int IExplicit.M() { return 0; }
}
```

ClassInterfaceType.None 值防止类元数据导出到类型库时生成类接口。在前面的示例中，COM 客户端只能通过 `IExplicit` 接口访问 `LoanApp` 类。

避免缓存调度标识符 (DispId)

对于脚本化客户端、Microsoft Visual Basic 6.0 客户端或不缓存接口成员的 DispId 的任何后期绑定客户端，可接受使用类接口。DispId 标识接口成员，以启用后期绑定。

对于类接口，基于接口中成员的位置生成 DispId。如果更改了成员顺序并将类导出到类型库中，则将改变类接口中生成的 DispId。

若要避免在使用类接口时中断后期绑定 COM 客户端，请应用具有 ClassInterfaceAttribute 值的 ClassInterfaceType.AutoDispatch。此值实现仅支持调度的类接口，但将省略类型库中的接口说明。没有接口说明，客户端就无法在编译时缓存 DispId。尽管这是类接口的默认接口类型，但你可以显式地应用属性值。

```
<ClassInterface(ClassInterfaceType.AutoDispatch)> Public Class LoanApp
    Implements IAnother
    Sub M() Implements IAnother.M
…
End Class
```

```
[ClassInterface(ClassInterfaceType.AutoDispatch)]
public class LoanApp
{
    public int M() { return 0; }
}
```

若要在运行时获取接口成员的 DispId，COM 客户端可以调用 IDispatch.GetIdsOfNames。若要调用接口上的方法，请将返回的 DispId 作为参数传递给 IDispatch.Invoke。

限制使用类接口的双重接口选项。

双重接口通过 COM 客户端启用对接口成员的早期绑定和后期绑定。在设计时和测试期间，将类接口设置为双重可能会非常有用。对于永远不会被修改的托管类（及其基类），此选项也是可接受的。在所有其它情况下，请避免将类接口设置为双重。

自动生成的双重接口可能适合少数情况；但是，更多情况下，它将造成与版本相关的复杂性。例如，使用派生类的类接口的 COM 客户端可以通过对基类的更改轻松中断。当第三方提供基类时，类接口的布局将不受你的控制。进一步来说，与仅支持调度的接口不同，双重接口 (ClassInterfaceType.AutoDual)提供对导出的类型库中的类接口的说明。此类说明会促使后期绑定的客户端在编译时缓存 DispId。

确保所有 COM 事件通知都是后期绑定的。

默认情况下，COM 类型信息直接嵌入到托管程序集中，这会消除对主互操作程序集 (PIA) 的需要。但是，嵌入式类型信息的一个限制是它不支持通过早期绑定的 vtable 调用传递 COM 事件通知，而仅支持后期绑定的 `IDispatch::Invoke` 调用。

如果应用程序需要对 COM 事件接口方法进行早期绑定调用，则可以将 Visual Studio 中的"嵌入互操作类型"属性设置为 `true`，或在项目文件中包含以下元素：

```
<EmbedInteropTypes>True</EmbedInteropTypes>
```

请参阅

- ClassInterfaceAttribute
- COM 包装
- 向 COM 公开 .NET Framework 组件
- 向 COM 公开 .NET Core 组件
- 为互操作限定 .NET 类型
- 运行时可调用包装器

为 COM 互操作限定 .NET 类型

2021/11/16 •

若要向 COM 应用程序公开程序集中的类型，请考虑 COM 互操作在设计时的需求。如果符合以下准则，托管类型（类、接口、结构和枚举）将与 COM 类型无缝集成：

- 类应显式实现接口。

  尽管 COM 互操作提供了一种机制，用于自动生成包含类的所有成员及其基类成员的接口，但最好还是提供显式接口。自动生成的接口称为类接口。有关指南，请参阅类接口简介。

  可以使用 Visual Basic、C# 和 C++ 将接口定义合并在代码中，而无需使用接口定义语言 (IDL) 或其等效语言。有关语法的详细信息，请参见语言文档。

- 托管的类型必须是公共的。

  只有程序集中的公共类型才会注册并导出到类型库中。因此只有公共类型才对 COM 可见。

  托管类型会向其他未向 COM 公开的托管代码公开功能。例如，参数化的构造函数、静态方法和常数字段不会向 COM 客户端公开。此外，运行时在类型中和类型外封送数据时，数据可能会被复制或转换。

- 方法、属性、字段和事件必须是公共的。

  如果要对 COM 可见，公共类型的成员也必须是公共的。通过应用 ComVisibleAttribute，可以限制程序集、公共类型或公共类型的公共成员的可见性。默认情况下，所有公共类型和成员都是可见的。

- 具备公共无参数构造函数的类型才能从 COM 中激活。

  托管的公共类型对于 COM 是可见的。但是如果没有公共无参数构造函数（不带任何参数的构造函数），COM 客户端无法创建该类型。如果该类型由其他方法激活，COM 客户端仍可使用该类型。

- 类型不能是抽象的。

  COM 客户端和 .NET 客户端都不能创建抽象的类型。

导出到 COM 后，托管类型的继承层次结构将被展平。托管和非托管环境之间的版本控制也会有所不同。向 COM 公开的类型不具有其他托管类型相同的版本控制特性。

请参阅

- ComVisibleAttribute
- 向 COM 公开 .NET Framework 组件
- 类接口简介
- 应用互操作属性
- 打包用于 COM 的 .NET Framework 程序集

应用互操作特性

2021/11/16 •

System.Runtime.InteropServices 命名空间提供三类特定于互操作的特性：在设计时由你应用的特性、在转换进程中由 COM 互操作工具和 API 应用的特性以及由你或 COM 互操作应用的特性。

如果不熟悉将特性应用到托管代码的任务，请参阅利用特性扩展元数据。如其他自定义特性一样，可以将特定于互操作的特性应用于类型、方法、属性、参数、字段和其他成员。

设计时特性

可以使用设计时特性调整由 COM 互操作工具和 API 执行的转换进程的结果。下表介绍了可以应用到托管源代码的特性。有时，COM 互操作工具也可能应用此表中所述的特性。

"	"
AutomationProxyAttribute	指定应使用自动化封送处理程序还是自定义代理和存根对类型进行封送处理。
ClassInterfaceAttribute	控制为类生成的接口类型。
CoClassAttribute	标识从类型库导入的原始组件类的 CLSID。 COM 互操作工具通常应用此特性。
ComImportAttribute	指示组件类或接口定义是从 COM 类型库导入的。运行时使用此标记来确定如何对类型进行激活和封送处理。此特性禁止将类型导入回类型库。 COM 互操作工具通常应用此特性。
ComRegisterFunctionAttribute	指定在从 COM 注册程序集以使用时应调用的方法，这样在注册过程中可以执行用户编写的代码。
ComSourceInterfacesAttribute	标识类的事件源的接口。 COM 互操作工具可以应用此特性。
ComUnregisterFunctionAttribute	指示在从 COM 取消注册程序集时应调用的方法，这样用户编写的代码可以在过程中执行。
ComVisibleAttribute	当特性值为“false”时，将使类型对 COM 不可见。此特性可以应用于单个类型或整个程序集，以控制 COM 的可见性。默认情况下，所有托管的公共类型都是可见的；不需要使用此特性来使它们可见。
DispIdAttribute	指定方法或字段的 COM 调度标识符 (DISPID)。此特性包含适用于它所述的方法、字段或属性的 DISPID。 COM 互操作工具可以应用此特性。

ComDefaultInterfaceAttribute	指示在 .NET 中实现的默认 COM 类接口。 COM 互操作工具可以应用此特性。
FieldOffsetAttribute	指示当用于 StructLayoutAttribute 时类中每个字段的物理位置，并且将 LayoutKind 设置为 Explicit 。
GuidAttribute	指定类、接口或整个类型库的全局唯一标识符 (GUID)。传递到此特性的字符串必须是 System.Guid 类型可接受的构造函数参数的格式。 COM 互操作工具可以应用此特性。
IDispatchImplAttribute	指示当向 COM 公开双重接口和调度时，公共语言运行时使用哪个 IDispatch 接口实现。
InAttribute	指示数据应封送到调用方。可用于特性参数。
InterfaceTypeAttribute	控制如何向 COM 客户端公开托管接口（仅限于双重、IUnknown-derived 或 IDispatch）。 COM 互操作工具可以应用此特性。
LCIDConversionAttribute	指示非托管的方法签名应具有 LCID 参数。 COM 互操作工具可以应用此特性。
MarshalAsAttribute	指示应如何在托管和非托管代码之间封送处理字段或参数中的数据。此特性始终是可选的，因为每个数据类型都具有默认的封送处理行为。 COM 互操作工具可以应用此特性。
OptionalAttribute	指示参数是可选的。 COM 互操作工具可以应用此特性。
OutAttribute	指示字段或参数中的数据必须从调用的对象被封送回其调用方。
PreserveSigAttribute	取消一般在互操作调用过程中发生的 HRESULT 或 retval 签名转换。特性会影响封送处理以及类型库导出。 COM 互操作工具可以应用此特性。
ProgIdAttribute	指定 .NET 类的 ProgID。可用于特性类。
StructLayoutAttribute	控制类的字段的物理布局。 COM 互操作工具可以应用此特性。

转换工具特性

下表介绍了转换过程期间 COM 互操作工具应用的特性。不要在设计时应用这些特性。

ComAliasNameAttribute	指示参数或字段类型的 COM 别名。可用于特性参数、字段或返回值。
ComConversionLossAttribute	指示有关类或接口的信息从类型库被导入到程序集时丢失。
ComEventInterfaceAttribute	标识源接口和实现事件接口的方法的类。
ImportedFromTypeLibAttribute	指示程序集最初是从 COM 类型库导入的。此特性包含原始类型库的类型库定义。
TypeLibFuncAttribute	包含最初从 COM 类型库为此功能导入的 FUNCFLAGS。
TypeLibTypeAttribute	包含最初从 COM 类型库为此类型导入的 TYPEFLAGS。
TypeLibVarAttribute	包含最初从 COM 类型库为此变量导入的 VARFLAGS。

请参阅

- System.Runtime.InteropServices
- 向 COM 公开 .NET Framework 组件
- 特性
- 为互操作限定 .NET 类型
- 打包用于 COM 的 .NET Framework 程序集

在非托管代码中处理互操作异常

2021/11/16 •

仅在 Windows 平台上支持非托管代码异常互操作。非 Windows 平台上会出现可移植性问题。由于 Unix ABI 没有异常处理的定义，因此托管代码无法知道异常机制的内部工作方式。因此，异常最终可能导致不可预知的行为和故障。

Setjmp/Longjmp 行为

不支持与 `setjmp` 和 `longjmp` C 函数的互操作。无法使用 `longjmp` 跳过托管帧。

有关详细信息，请参阅 longjmp 文档。

请参阅

- 异常
- 与本机库的互操作

.NET 分发打包

2021/11/16 •

随着 .NET 5（和 .NET Core）以及更高版本在越来越多的平台上可用，了解如何对使用它的应用和库进行打包、命名和版本控制非常有用。这样，无论用户选择在哪里运行 .NET，包维护人员均可以帮助确保获得一致的体验。本文对以下用户非常有用：

- 尝试从源生成 .NET。
- 想要更改 .NET CLI，但更改可能会影响生成的布局或包。

磁盘布局

安装时，.NET 包含一些组件，这些组件在文件系统中排列如下：

```
{dotnet_root}                                   (*)
├── dotnet                          (1)
├── LICENSE.txt                     (8)
├── ThirdPartyNotices.txt           (8)
├── host                                        (*)
│   └── fxr                                     (*)
│       └── <fxr version>           (2)
├── sdk                                         (*)
│   ├── <sdk version>               (3)
│   └── NuGetFallbackFolder         (4)         (*)
├── packs                                       (*)
│   ├── Microsoft.AspNetCore.App.Ref            (*)
│   │   └── <aspnetcore ref version>    (11)
│   ├── Microsoft.NETCore.App.Ref               (*)
│   │   └── <netcore ref version>       (12)
│   ├── Microsoft.NETCore.App.Host.<rid>        (*)
│   │   └── <apphost version>           (13)
│   ├── Microsoft.WindowsDesktop.App.Ref        (*)
│   │   └── <desktop ref version>       (14)
│   └── NETStandard.Library.Ref                 (*)
│       └── <netstandard version>       (15)
├── shared                                      (*)
│   ├── Microsoft.NETCore.App                   (*)
│   │   └── <runtime version>     (5)
│   ├── Microsoft.AspNetCore.App                (*)
│   │   └── <aspnetcore version>  (6)
│   ├── Microsoft.AspNetCore.All                (*)
│   │   └── <aspnetcore version>  (6)
│   └── Microsoft.WindowsDesktop.App            (*)
│       └── <desktop app version> (7)
└── templates                                   (*)
    └── <templates version>       (17)
/
├── etc/dotnet
│       └── install_location      (16)
├── usr/share/man/man1
│       └── dotnet.1.gz           (9)
└── usr/bin
        └── dotnet                (10)
```

- (1) **dotnet 主机**（也称为“muxer”）有两个不同角色：激活运行时以启动应用程序，及激活 SDK 以向其分派命令。主机是本机可执行文件（`dotnet.exe`）。

主机只有一个，不过大部分的其他组件都在带有版本的目录中（2、3、5 和 6）。这意味着系统上可存在多个版本，

因为它们是并排安装的。

- (2) host/fxr/<fxr version> 包含了主机所使用的框架解析逻辑。主机采用已安装的最新 hostfxr。在执行 .NET 应用程序时，hostfxr 负责选择合适的运行时。例如，为 .NET Core 2.0.0 生成的应用程序会使用 2.0.5 运行时（如果可用）。同样，hostfxr 在开发期间也会选择适当的 SDK。
- (3) sdk/<sdk version> SDK（也称为“工具”）是一组托管工具，可用于编写和生成 .NET 库和应用程序。SDK 包括 .NET CLI、托管的语言编译器、MSBuild 及相关生成任务和目标、NuGet、新项目模板等。
- (4) **sdk/NuGetFallbackFolder** 包含 SDK 在还原操作期间使用的 NuGet 包的缓存，例如在运行 `dotnet restore` 或 `dotnet build` 时。此文件夹仅在 .NET Core 3.0 之前使用。不能从源生成它，因为它包含来自 `nuget.org` 的预构建二进制资产。

“共享” 文件夹包含框架。共享框架提供一组位于中心位置的库，从而让不同的应用程序使用。

- (5) shared/Microsoft.NETCore.App/<runtime version> 此框架包含.NET 运行时和支持托管库。
- (6) shared/Microsoft.AspNetCore.{App,All}/<aspnetcore version> 包含 ASP.NET Core 库。已开发且支持 `Microsoft.AspNetCore.App` 下的库（作为 .NET 项目的一部分）。`Microsoft.AspNetCore.All` 下的库是一个超集，其中还包含第三方库。
- (7) shared/Microsoft.Desktop.App/<desktop app version> 包含 Windows 桌面库。在非 Windows 平台上不包含此项。
- (8) LICENSE.txt 和 ThirdPartyNotices.txt 分别是 .NET 许可证和 .NET 中使用的第三方库的许可证。
- (9,10) **dotnet.1.gz, dotnet** `dotnet.1.gz` 是 dotnet 手册页。`dotnet` 是指向 dotnet 主机 (1) 的符号链接。这些文件安装在已知位置用于系统集成。
- (11,12) Microsoft.NETCore.App.Ref 和 Microsoft.AspNetCore.App.Ref 分别描述了 `x.y` 版本 .NET 和 ASP.NET Core 的 API。针对这些目标版本进行编译时，将使用这些包。
- (13) Microsoft.NETCore.App.Host.<rid> 包含平台 `rid` 的原生二进制文件。将 .NET 应用程序编译为适用于该平台的本机二进制文件时，将使用此二进制文件作为模板。
- (14) **Microsoft.WindowsDesktop.App.Ref** 介绍 Windows 桌面应用程序 `x.y` 版本的 API。在针对该目标进行编译时，将使用这些文件。在非 Windows 平台上不提供此项。
- (15) **NETStandard.Library.Ref** 描述了 netstandard `x.y` API。在针对该目标进行编译时，将使用这些文件。
- (16)“/etc/dotnet/install_location”是一个包含 `{dotnet_root}` 完整路径的文件 。该路径可能以换行符结尾。根路径为 `/usr/share/dotnet` 时无需添加此文件。
- (17) templates 包含 SDK 使用的模板。例如，`dotnet new` 在此处查找项目模板。

标记为 `(*)` 的文件夹被多个包使用。某些包格式（例如，`rpm`）需要对此类文件夹进行特殊处理。包维护人员必须处理这个问题。

推荐的包

.NET 版本控制基于运行时组件 `[major].[minor]` 版本号。SDK 版本采用相同的 `[major].[minor]`，并有一个独立的 `[patch]`，它为 SDK 合并了功能和修补语义。例如：SDK 版本 2.2.302 是支持 2.2 运行时的 SDK 的第 3 个功能版本的第 2 个补丁版本。有关版本控制的工作原理的详细信息，请参阅 .NET 版本控制概述。

一些包在自己的名称中就包含一部分版本号。这允许你安装特定版本。版本名称中不包含版本的剩余部分。这允许 OS 包管理器更新这些包（例如，自动安装安全修补程序）。支持的包管理器特定于 Linux。

下面列出了推荐的包：

- `dotnet-sdk-[major].[minor]` - 安装特定运行时的最新 SDK
  - **版本**：<sdk version>
  - **示例**：dotnet-sdk-2.1
  - **包含**：(3),(4)
  - **依赖项**：`dotnet-runtime-[major].[minor]`、`aspnetcore-runtime-[major].[minor]`、`dotnet-targeting-pack-[major].[minor]`、`aspnetcore-targeting-pack-[major].[minor]`、`netstandard-targeting-pack-[netstandard_major].[netstandard_minor]`、`dotnet-apphost-pack-[major].[minor]`、`dotnet-templates-[major].[minor]`
- `aspnetcore-runtime-[major].[minor]` - 安装特定 ASP.NET Core 运行时
  - **版本**：<aspnetcore runtime version>
  - **示例**：aspnetcore-runtime-2.1
  - **包含**：(6)
  - **依赖项**：`dotnet-runtime-[major].[minor]`
- `dotnet-runtime-deps-[major].[minor]` *（可选）* - 安装运行自包含应用程序的依赖项
  - **版本**：<runtime version>
  - **示例**：dotnet-runtime-deps-2.1
  - **依赖项**：*特定于分发的依赖项*
- `dotnet-runtime-[major].[minor]` - 安装特定运行时
  - **版本**：<runtime version>
  - **示例**：dotnet-runtime-2.1
  - **包含**：(5)
  - **依赖项**：`dotnet-hostfxr-[major].[minor]`、`dotnet-runtime-deps-[major].[minor]`
- `dotnet-hostfxr-[major].[minor]` - 依赖项
  - **版本**：<runtime version>
  - **示例**：dotnet-hostfxr-3.0
  - **包含**：(2)
  - **依赖项**：`dotnet-host`
- `dotnet-host` - 依赖项
  - **版本**：<runtime version>
  - **示例**：dotnet-host
  - **包含**：(1),(8),(9),(10),(16)
- `dotnet-apphost-pack-[major].[minor]` - 依赖项
  - **版本**：<runtime version>
  - **包含**：(13)
- `dotnet-targeting-pack-[major].[minor]` - 允许面向非最新的运行时
  - **版本**：<runtime version>
  - **包含**：(12)
- `aspnetcore-targeting-pack-[major].[minor]` - 允许面向非最新的运行时
  - **版本**：<aspnetcore runtime version>
  - **包含**：(11)
- `netstandard-targeting-pack-[netstandard_major].[netstandard_minor]` - 允许面向 netstandard 版本
  - **版本**：<sdk version>

    - **包含**：(15)
- `dotnet-templates-[major].[minor]`
    - **版本**：<sdk version>
    - **包含**：(15)

`dotnet-runtime-deps-[major].[minor]` 需要了解发行版特定依赖项 。因为发行版生成系统可能能够自动派生包，所以包是可选的，如果选择，会将这些依赖项直接添加到 `dotnet-runtime-[major].[minor]` 包中。

当包内容位于受版本控制的文件夹下时，包名称 `[major].[minor]` 与受版本控制的文件夹名称匹配。对于所有包（除 `netstandard-targeting-pack-[netstandard_major].[netstandard_minor]` 外），这也与 .NET 版本匹配。

包间的依赖关系应使用"等于或大于"版本要求 。例如，`dotnet-sdk-2.2:2.2.401` 要求 `aspnetcore-runtime-2.2 >= 2.2.6`。这使用户可以通过根包（例如 `dnf update dotnet-sdk-2.2`）升级其安装。

大多数分发都需要从源中构建所有项目。这对包有一些影响：

- 不能简单地从源生成 `shared/Microsoft.AspNetCore.All` 下的第三方库。因此 `aspnetcore-runtime` 包中省略了该文件夹。
- 使用 `nuget.org` 中的二进制项目填充了 `NuGetFallbackFolder`。它应保留为空。

多个 `dotnet-sdk` 包可能会为 `NuGetFallbackFolder` 提供同样的文件。若要避免包管理器出现问题，这些文件应完全相同（包括校验和、修改日期等等）。

生成包

dotnet/source-build 存储库中说明了如何生成 .NET SDK 的源 tarball 及其所有组件。源版本存储库中的输出内容符合本文第一部分中所描述的布局。

库指南

2021/11/16 •

本指南向开发人员提供了有关创建高质量的 .NET 库的建议。本文档重点介绍在构建 .NET 库时的操作内容和原因，还不是操作方式。

高质量 .NET 库的方方面面：

- 包容性 - 优秀的 .NET 库致力于支持众多平台、编程语言和应用程序。
- 稳定性：优秀的 .NET 系统在具有众多库的应用程序中运行的 .NET 生态系统中共存。
- 设计为可改进：.NET 库要随着时间的推移进行改进和演变，同时支持现有用户。
- 可调试：.NET 库要使用最新的工具，为用户打造卓越的调试体验。
- 受信任：.NET 库通过安全最佳做法发布到 NuGet，备受开发人员的信赖。

入门

建议类型

每篇文章介绍四种类型的建议："请执行"、"请考虑"、、"请避免"、和"请勿"。建议类型表示了应遵循的程度。

应始终遵循"请执行"建议。例如：

✔ 请使用 NuGet 包分发库。

在另一方面，"请考虑"建议是在一般情况下要遵循的建议，但存在该规则的合法例外，此时不遵循指南也不妨：

✔ 请考虑使用 SemVer 2.0.0 控制 NuGet 包的版本。

"请避免"建议是指在一般情况下不应执行的操作，但有时也可以打破规则：

✘ 请避免使用需要确切版本的 NuGet 包引用。

最后，"请勿"建议是指在大多数情况下不得执行的操作：

✘ 请勿发布库的强名称和非强名称版本。例如，`Contoso.Api` 和 `Contoso.Api.StrongNamed`。

框架设计准则

2021/11/16 •

本部分提供了有关设计扩展 .NET Framework 并与 .NET Framework 进行交互的库的指南。其目标是通过提供独立于用于开发的编程语言的统一编程模型，来帮助库设计人员确保 API 一致性和易用性。建议在开发扩展 .NET Framework 的类和组件时遵循这些设计准则。库设计不一致会对开发人员工作效率产生负面影响，并阻碍采用。

本指南分为几组简单的建议，其前缀分别为 `Do`、`Consider`、`Avoid` 和 `Do not`。这些准则旨在帮助类库设计人员了解不同解决方案之间的权衡。在某些情况下，良好的库设计需要你违反这些设计准则。这种情况应该很少见，因此你需要依据一个清晰且令人信服的理由作出决策。

这些准则摘自《Framework 设计准则：可重用 .NET 库的约定、惯例和模式第 2 版》，作者：Krzysztof Cwalina 和 Brad Abrams。

本节内容

命名规则
提供针对类库中的命名程序集、命名空间、类型和成员的准则。

类型设计准则
提供有关使用静态和抽象类、接口、枚举、结构和其他类型的准则。

成员设计准则
提供有关设计和使用属性、方法、构造函数、字段、事件、运算符和参数的准则。

扩展性设计
讨论扩展性机制（如子类化、使用事件、虚拟成员和回调），并说明如何选择最能满足框架要求的机制。

异常设计准则
介绍有关设计、引发和捕获异常的设计准则。

使用准则
介绍有关使用常见类型（如数组、特性和集合、支持序列化以及重载相等性运算符）的准则。

常用设计模型
提供有关选择和实现依赖项属性的准则。

请参阅

- 概述
- 开发指南

命名准则

2021/11/16 •

在框架开发中采用一致的命名约定集可能是框架可用性的主要贡献。它允许很多开发人员对广泛分隔的项目使用框架。除了表单的一致性以外，框架元素的名称也必须易于理解，并且必须传达每个元素的功能。

本章的目的是提供一组一致的命名约定，以使开发人员能够立即了解名称。

尽管采用这些命名约定作为一般代码开发准则会使代码的命名更加一致，但你只需将它们应用到公开 (公共或受保护类型和成员的 Api，并) 显式实现的接口。

本节内容

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则

大小写约定

2021/11/16 •

本章中的指导原则介绍了使用大小写的简单方法，即在应用一致的情况下，使类型、成员和参数的标识符易读。

有关标识符的首字母大写规则

若要区分标识符中的单词，请将标识符中每个单词的首字母大写。不要使用下划线来区分单词，或在标识符中的任何位置使用下划线。有两种适当的方法可以根据标识符的使用将其首字母大写：

- PascalCasing
- camelCasing

PascalCasing 约定（用于除了参数名称外的所有标识符）将每个单词（包括长度超过两个字母的首字母缩写词）的第一个字符大写，如以下示例所示：

`PropertyDescriptor` `HtmlTag`

两个字母的首字母缩写词是一种特殊情况，其中两个字母都大写，如以下标识符所示：

`IOStream`

camelCasing 约定（仅用于参数名称，将每个单词的第一个字符（除第一个单词之外）大写，如以下示例所示。如示例中所示，以字母混合形式表示的两个字母首字母缩写词均采用小写。

`propertyDescriptor` `ioStream` `htmlTag`

✔ 对于包含多个单词的所有公共成员、类型和命名空间名称，请使用 PascalCasing。

✔ 使用 camelCasing 作为参数名称。

下表描述了不同标识符类型的首字母大写规则。

命名空间	Pascal	`namespace System.Security { ... }`
类型	Pascal	`public class StreamReader { ... }`
接口	Pascal	`public interface IEnumerable { ... }`
方法	Pascal	`public class Object { public virtual string ToString(); }`
属性	Pascal	`public class String { public int Length { get; } }`

[illegible]	[illegible]	[illegible]
事件	Pascal	`public class Process { public event EventHandler Exited; }`
字段	Pascal	`public class MessageQueue { public static readonly TimeSpan InfiniteTimeout; } public struct UInt32 { public const Min = 0; }`
枚举值	Pascal	`public enum FileMode { Append, ... }`
参数	混合	`public class Convert { public static int ToInt32(string value); }`

将组合词和常见术语的首字母大写

为了实现首字母大写，大多数组合术语都被视为单个单词。

✕ 不要将所谓的"闭合形式"组合词中的每个首字母大写。

这些是以单个词（如终结点）形式编写的组合词。为了符号大小写准则，请将"闭合形式"组合词视为单个单词。使用当前字典来确定是否以闭合形式写入组合词。

PASCAL	[illegible]	NOT
`BitFlag`	`bitFlag`	`Bitflag`
`Callback`	`callback`	`CallBack`
`Canceled`	`canceled`	`Cancelled`
`DoNot`	`doNot`	`Don't`
`Email`	`email`	`EMail`
`Endpoint`	`endpoint`	`EndPoint`
`FileName`	`fileName`	`Filename`
`Gridline`	`gridline`	`GridLine`
`Hashtable`	`hashtable`	`HashTable`

PASCAL	ıı	NOT
`Id`	`id`	`ID`
`Indexes`	`indexes`	`Indices`
`LogOff`	`logOff`	`LogOut`
`LogOn`	`logOn`	`LogIn`
`Metadata`	`metadata`	`MetaData, metaData`
`Multipanel`	`multipanel`	`MultiPanel`
`Multiview`	`multiview`	`MultiView`
`Namespace`	`namespace`	`NameSpace`
`Ok`	`ok`	`OK`
`Pi`	`pi`	`PI`
`Placeholder`	`placeholder`	`PlaceHolder`
`SignIn`	`signIn`	`SignOn`
`SignOut`	`signOut`	`SignOff`
`UserName`	`userName`	`Username`
`WhiteSpace`	`whiteSpace`	`Whitespace`
`Writable`	`writable`	`Writeable`

区分大小写

可在 CLR 上运行的语言不需要支持区分大小写，但有些则不需要。即使你的语言支持，其他可能访问你框架的语言也不是如此。因此，外部可访问的任何 API 都不能单独依赖于大小写来区分同一上下文中的两个名称。

✕ 不要假设所有编程语言都区分大小写。它们不是。名称不能按大小写单独区分。

请参阅

- 框架设计指南
- 命名规则

通用命名约定

2021/11/16 •

本部分介绍与选词相关的常规命名约定、有关使用缩写和首字母缩写词的准则，以及如何避免使用特定于语言的名称的建议。

选词

✔ 选择易读的标识符名称。

例如，属性 `HorizontalAlignment` 在英文方面的可读性高于 `AlignmentHorizontal`。

✔ 可读性比简洁性更重要。

属性名称 `CanScrollHorizontally` 比 `ScrollableX`（对 X 轴的模糊引用）更好。

✕ 不要使用下划线、连字符或任何其他非字母数字字符。

✕ 不要使用匈牙利表示法。

✕ 避免使用与广泛应用的编程语言关键字冲突的标识符。

根据公共语言规范 (CLS) 的规则 4，所有符合语言都必须提供一种机制，以允许访问使用该语言的关键字作为标识符的已命名项。例如，在这种情况下，C# 使用 @ 符号作为转义机制。不过，最好是避免使用常见关键字，因为使用转义序列的方法比不使用该序列的方法更难。

使用缩写和首字母缩写词

✕ 不要使用缩写或缩写词作为标识符名称的一部分。

例如，使用 `GetWindow` 而不是 `GetWin`。

✕ 不要使用未被广泛接受的任何首字母缩写词，仅在必要时才使用。

避免使用特定于语言的名称

✔ 使用有语义的名称，而不是类型名称的特定于语言的关键字。

例如，相比 `GetInt`，`GetLength` 是更好的名称。

✔ 在极少数情况下，如果标识符没有超出其类型的语义含义，则使用泛型 CLR 类型名称，而不是特定于语言的名称。

例如，转换为 Int64 的方法应命名为 `ToInt64`，而不是 `ToLong`（因为 Int64 是特定于 C# 别名 `long` 的 CLR 名称）。下表显示了几个使用 CLR 类型名称的基本数据类型，以及 C#、Visual Basic 和 C++) 的相应类型名称。

C#	VISUAL BASIC	C++	CLR
sbyte	SByte	char	SByte
byte	Byte	unsigned char	Byte
short	Short	short	Int16

C#	VISUAL BASIC	C++	CLR
ushort	UInt16	unsigned short	UInt16
int	Integer	int	Int32
uint	UInt32	unsigned int	UInt32
long	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	■	float	■
double	■	double	■
bool	■	bool	■
char	Char	wchar_t	Char
string	■	■	■
object	Object	Object	Object

✔ 在极少数情况下，如果标识符没有语义含义，并且参数的类型并不重要，可以使用常见名称（如 `value` 或 `item`），而不是重复使用类型名称。

命名现有 API 的新版本

✔ 创建现有 API 的新版本时，使用类似于旧 API 的名称。

这有助于突出显示两个 API 之间的关系。

✔ 更喜欢添加后缀（而非前缀）来指示现有 API 的新版本。

这将有助于在浏览文档或使用 IntelliSense 时发现。旧版本的 API 将被组织到新的 Api 附近，因为大多数浏览器和 IntelliSense 都按字母顺序显示标识符。

✔ 考虑使用全新但有意义的标识符，而不是添加后缀或前缀。

✔ 使用数字后缀来指示现有 API 的新版本，尤其是当 API 的现有名称是唯一有意义的名称（例如，如果其是行业标准），并且添加任何有意义的后缀（或更改名称）并非不是合适的选择时。

✕ 不要使用标识符的“前”（或类似的）后缀，将其与同一 API 的先前版本区分开。

✔ 在引入在 64 位整数上操作的 API 版本（长整数）而不是在 32 位整数上操作时，可以使用后缀“64”。仅当存在现有的 32 位 API 时，才需要采用此方法；请勿对只有 64 位版本的新 API 使用此方法。

请参阅

- 框架设计准则
- 命名准则
- EditorConfig 适用的 .NET 命名约定

程序集和 DLL 的名称

2021/11/16 •

程序集是托管代码程序的部署和标识单元。尽管程序集可跨一个或多个文件，但通常程序集与 DLL 相互映射。因此，本节仅介绍 DLL 命名约定，然后可以将其映射到程序集命名约定。

✔为程序集 Dll 选择名称，这些名称建议了大块功能，如 System.object。

程序集和 DLL 的名称不必与命名空间名称对应，但在命名程序集时遵循命名空间名称是合理的。合理的经验法则是基于程序集中包含的命名空间的公共前缀来命名 DLL。例如，可以调用具有两个命名空间和的程序集 `MyCompany.MyTechnology.FirstFeature` `MyCompany.MyTechnology.SecondFeature` `MyCompany.MyTechnology.dll` 。

✔考虑根据以下模式命名 Dll：

```
<Company>.<Component>.dll
```

其中 `<Component>` 包含一个或多个以句点分隔的子句。例如：

`Litware.Controls.dll`.

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则
- 命名准则

命名空间的名称

2021/11/16 •

与其他命名准则一样，命名命名空间的目标是为程序员创建足够的清晰度，使其能够立即了解命名空间的内容。以下模板指定命名空间的一般规则：

```
<Company>.(<Product>|<Technology>)[.<Feature>][.<Subnamespace>]
```

下面是一些示例：

`Fabrikam.Math` `Litware.Security`

✔ 使用公司名称作为命名空间名称的前缀，以防不同公司的命名空间具有相同的名称。

✔ 在命名空间名称的第二级使用稳定的独立于版本的产品名称。

✕ 不要使用组织层次结构作为命名空间层次结构中的名称的基础，因为公司内的组名通常是短期的。围绕一组相关技术组织命名空间的层次结构。

✔ 使用 PascalCasing，并使用句点分隔命名空间组件 (例如，`Microsoft.Office.PowerPoint`)。如果品牌采用 nontraditional 大小写，则应遵循品牌定义的大小写，即使它与常规命名空间大小写不符。

✔ 考虑在适当的情况下使用复数命名空间名称。

例如，请使用 `System.Collections` 而不是 `System.Collection`。不过，品牌名称和首字母缩写是此规则的例外情况。例如，请使用 `System.IO` 而不是 `System.IOs`。

✕ 命名空间和该命名空间中的类型不要使用相同的名称。

例如，不要将 `Debug` 用作命名空间名称，并 `Debug` 在同一命名空间中提供名为的类。一些编译器需要完全限定此类类型。

命名空间和类型名称冲突

✕ 不要引入泛型类型名称 `Element`，例如、、`Node` `Log` 和 `Message`。

很多情况下，这样做会导致在常见方案中发生类型名称冲突。应限定泛型类型名称 (`FormElement`、`XmlNode`、`EventLog` `SoapMessage`)。

为避免不同类别的命名空间的类型名称冲突，有一些特定的准则。

- **应用程序模型命名空间**

  属于单个应用程序模型的命名空间经常一起使用，但几乎不能与其他应用程序模型的命名空间一起使用。例如，System.Windows.Forms 命名空间非常少与 System.Web.UI 命名空间一起使用。下面列出了众所周知的应用程序模型命名空间组：

  `System.Windows*` `System.Web.UI*`

  ✕ 不要为单个应用程序模型中的命名空间中的类型指定相同的名称。

  例如，不要将名为的类型添加 `Page` 到 System.Web.UI.Adapters 命名空间，因为该 System.Web.UI 命名空间已经包含一个名为的类型 `Page`。

- **基础结构命名空间**

  此组包含在开发常见应用程序期间很少导入的命名空间。例如，`.Design` 命名空间主要在开发编程工具时使用。避免与这些命名空间中的类型冲突并不重要。

- **核心命名空间**

  核心命名空间包括所有 `System` 命名空间(不包括应用程序模型的命名空间和基础结构命名空间)。核心命名空间包括、、、 `System` `System.IO` `System.Xml` 和 `System.Net` 。

  ✕ 不要提供会与核心命名空间中的任何类型冲突的类型名称。

  例如，永远不要将 `Stream` 用作类型名称。它会与 System.IO.Stream 非常常用的类型冲突。

- **技术命名空间组**

  此类别包括) 的前两个命名空间节点相同的所有命名空间 `(<Company>.<Technology>*` ，例如 `Microsoft.Build.Utilities` 和 `Microsoft.Build.Tasks` 。属于单个技术的类型不会相互冲突，这一点非常重要。

  ✕ 不要分配会与单个技术中的其他类型冲突的类型名称。

  ✕ 请勿引入技术命名空间和应用程序模型命名空间中的类型之间的类型名称冲突 (除非该技术不打算用于应用程序模型) 。

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则
- 命名准则

类、结构和接口的名称

2021/11/16 •

遵循的命名准则适用于常规类型命名。

✔使用 PascalCasing 将类和结构命名为名词或名词短语。

这区分了方法中的类型名称，这些类型名为带有谓词短语。

✔利用形容词短语或偶尔用名词或名词短语来命名接口。

应很少使用名词和名词短语，它们可能会指示该类型应为抽象类，而不是接口。

✕ 不要为类名称指定前缀 (例如，"C") 。

✔考虑用基类的名称结束派生类的名称。

这是非常可读的，并且清楚地说明了该关系。代码中的一些示例是:，它是一种 `ArgumentOutOfRangeException` `Exception` ，而 `SerializableAttribute` 是一种类型的 `Attribute` 。但是，在应用此准则时使用合理的判断非常重要;例如， `Button` 类是一种 `Control` 事件，但 `Control` 它的名称中没有显示。

✔使用字母 I 来为接口名称加上前缀，以指示该类型是接口。

例如， `IComponent` (描述性名词) ， `ICustomAttributeProvider` (名词短语) ， `IPersistable` (形容词) 是适当的接口名称。对于其他类型名称，请避免缩写形式。

✔确保在定义类接口对（其中类是接口的标准实现）时，接口名称上的 "I" 前缀仅有不同的名称。

泛型类型参数的名称

已将泛型添加到 .NET Framework 2.0。此功能引入了一种称为 *类型参数* 的新标识符。

✔使用描述性名称命名泛型类型参数，除非单个字母名称完全一目了然，并且描述性名称不会添加值。

✔考虑使用 `T` 作为具有一个单字母类型参数的类型的类型参数名称。

```
public int IComparer<T> { ... }
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T:struct { ... }
```

✔在中为描述性类型参数名称加上前缀 `T` 。

```
public interface ISessionChannel<TSession> where TSession : ISession {
    TSession Session { get; }
}
```

✔考虑，指示对参数名称中的类型参数施加的约束。

例如，可以调用约束为的参数 `ISession` `TSession` 。

常见类型的名称

当命名类型派生自或实现某些 .NET Framework 类型时，✔按照下表中所述的指导原则进行操作。

☐☐☐	☐☐/☐☐☐☐☐☐
`System.Attribute`	✔将后缀 "Attribute" 添加到自定义特性类的名称中。
`System.Delegate`	✔将后缀 "EventHandler" 添加到在事件中使用的委托的名称。 ✔将后缀 "Callback" 添加到作为事件处理程序使用的委托的名称。 ✕ 不要将后缀 "Delegate" 添加到委托。
`System.EventArgs`	✔添加后缀 "EventArgs"。
`System.Enum`	✕ 不要从此类派生;改为使用您的语言支持的关键字;例如，在 c # 中，使用 `enum` 关键字。 ✕ 不要添加后缀 "Enum" 或 "标志"。
`System.Exception`	✔添加后缀 "Exception"。
`IDictionary` `IDictionary<TKey,TValue>`	✔添加后缀 "Dictionary"。请注意，`IDictionary` 是一种特定类型的集合，但是此准则优先于下面的更常见的集合原则。

`IEnumerable`
`ICollection`
`IList`
`IEnumerable<T>`
`ICollection<T>`
`IList<T>`
	✔添加后缀 "Collection"。

`System.IO.Stream`	✔添加后缀 "Stream"。
`CodeAccessPermission IPermission`	✔添加后缀 "权限"。

命名枚举

通常情况下），枚举类型 (名称也称为枚举) 应遵循标准类型命名规则 (PascalCasing 等。不过，还有其他一些准则专门适用于枚举。

✔为枚举使用单数类型名称，除非其值为位域。

✔对将位域作为值的枚举使用复数类型名称，也称为标志枚举。

✕ 不要在枚举类型名称中使用 "Enum" 后缀。

✕ 不要在枚举类型名称中使用 "标记" 或 "标志" 后缀。

✕ 不要在枚举值名称上使用前缀 (例如，将 "ad" 用于 ADO 枚举，使用 "rtf" 进行丰富文本枚举等）。

另请参阅

- 框架设计准则
- 命名准则

类型成员的名称

2021/11/16 •

类型由以下成员构成：方法、属性、事件、构造函数和字段。以下各节介绍命名类型成员的准则。

方法的名称

方法是执行操作的方式，设计准则要求方法名称为谓词或谓词短语。遵循此准则，还有利于区分方法名称与属性和类型名称，后者为名词或形容词性短语。

✔ 为谓词或谓词短语指定方法名称。

```
public class String {
    public int CompareTo(...);
    public string[] Split(...);
    public string Trim();
}
```

属性的名称

与其他成员不同，应向属性给定名词性短语或形容词性名称。这是因为属性是指数据，属性的名称应反映这一点。属性名称总是采用帕斯卡大小写。

✔ 使用名词、名词短语或形容词名称属性。

✕ 没有与 "Get" 方法的名称相匹配的属性，如以下示例中所示：

`public string TextWriter { get {...} set {...} }` `public string GetTextWriter(int value) { ... }`

此模式通常意味着该属性事实上是一种方法。

✔ 使用一个复数短语来描述集合中的项，而不是使用后跟 "List" 或 "Collection" 的短语来命名集合属性。

✔ 使用赞成短语 (, `CanSeek` 而不是) 来命名布尔属性 `CantSeek` 。或者，还可以在布尔属性前面添加 "Is"、"Can" 或 "has" 前缀，但前提是在它添加值的位置。

✔ 考虑为属性提供与其类型相同的名称。

例如，以下属性可正确获取和设置名为 `Color` 的枚举值，因此属性名为 `Color` ：

```
public enum Color {...}
public class Control {
    public Color Color { get {...} set {...} }
}
```

事件的名称

事件始终指操作，可以是即将发生的，也可以是已经发生的。因此，对于方法，事件用谓词命名，并用谓词时态指示引发事件的时间。

✔ 使用动词或动词短语来命名事件。

示例包括 `Clicked` 、 `Painting` 和 `DroppedDown` 。

✔ 使用现有的和过去的时态为事件名称提供前后的概念。

例如，在窗口关闭前引发的关闭事件可命名为 `Closing`，而在窗口关闭后后引发的关闭事件可命名为 `Closed`。

✕ 不要使用 "Before" 或 "After" 前缀或 postfixes 来指示前和后事件。请如上所示使用现在时和过去时。

✔ 命名事件处理程序 (使用 "EventHandler" 后缀的事件类型) 委托，如下面的示例中所示：

```
public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

✔ 确实要 `sender` `e` 在事件处理程序中使用两个名为和的参数。

sender 参数表示引发事件的对象。sender 参数的类型通常是 `object`，即使可以使用更具体的类型。

✔ 用 "EventArgs" 后缀命名事件参数类。

字段的名称

字段命名准则适用于静态公开字段和受保护的字段。原则不涉及内部和专用字段，而成员设计准则不允许使用公开字段或受保护的实例字段。

✔ 在字段名称中使用 PascalCasing。

✔ 使用名词、名词短语或形容词来命名字段。

✕ 不要对字段名称使用前缀。

例如，请勿使用"g_"或"s_"来指示静态字段。

另请参阅

- 框架设计准则
- 命名准则

命名参数

2021/11/16 •

除了可读性的明显原因外，请务必遵循有关参数名称的准则，因为当可视化设计工具提供 Intellisense 和类浏览功能时，参数将显示在文档和设计器中。

✔ 在参数名称中使用 camelCasing。

✔ 使用描述性参数名称。

✔ 考虑使用基于参数含义而不是参数类型的名称。

命名运算符重载参数

`left` `right` 如果参数没有任何意义，✔ 确实要使用和进行二元运算符重载参数名称。

对于参数，✔ 确实使用 `value` 一元运算符重载参数名称。

✔ 考虑运算符重载参数有意义的名称，如果这样做会增加重要值。

✕ 不要对运算符重载参数名称使用缩写或数值索引。

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则
- 命名准则

命名资源

2021/11/16 •

由于可以通过某些对象来引用可本地化的资源，就像它们是属性一样，因此资源的命名准则与属性准则类似。

✔ 在资源键中使用 PascalCasing。

✔ 提供描述性而不是短标识符。

✕ 不要使用主要 CLR 语言的特定于语言的关键字。

✔ 在命名资源中仅使用字母数字字符和下划线。

✔ 对异常消息资源使用以下命名约定。

资源标识符应为异常类型名称和异常的简短标识符：

`ArgumentExceptionIllegalCharacters` `ArgumentExceptionInvalidName` `ArgumentExceptionFileNameIsMalformed`

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则
- 命名准则

类型设计准则

2021/11/16 •

从 CLR 的角度来看，只有两种类型：引用类型和值类型，但为了讨论框架设计，我们将类型分成更多的逻辑组，每个逻辑组都有自己特定的设计规则。

类是引用类型的常规用例。它们构成了大多数框架中的大部分类型。类之所以受欢迎，是因为它们支持一组丰富的面向对象的功能以及它们的普遍适用性。基类和抽象类是与扩展性相关的特殊逻辑组。

接口是可由引用类型和值类型实现的类型。因此，它们可充当引用类型和值类型的多态层次结构的根。此外，接口还可用于模拟多重继承，而这是 CLR 本机不支持的。

结构是值类型的常规用例，应为小型简单类型保留，类似于语言基元。

枚举是值类型的一种特例，用于定义短值集，如星期几、控制台颜色等。

静态类是旨在作为静态成员容器的类型。它们通常用于提供其他操作的快捷方式。

委托、异常、特性、数组和集合都是专用于特定用途的引用类型的特例，它们的设计和使用指南在本书的其他位置进行了讨论。

✔ 请务必确保每个类型都是一组定义完善的相关成员，而不只是一个随机的无关功能集合。

本节内容

在类和结构之间选择 抽象类设计 静态类设计 接口设计 结构设计 枚举设计 嵌套设计

请参阅

- 框架设计指南

在类和结构之间选择

2021/11/16 •

每个框架设计器所面临的一项基本设计决策是，是将类型设计为作为类 (引用类型) 还是作为结构 () 值类型。充分了解引用类型和值类型的行为差异对于做出此选择至关重要。

引用类型和值类型的第一个区别在于，引用类型在堆上分配并进行垃圾回收，而值类型则在堆栈上分配，或在包含类型中以内联方式分配，在堆栈展开或其包含类型释放时释放。因此，值类型的分配和释放通常比引用类型的分配和释放更便宜。

接下来，将对引用类型的数组进行内嵌分配，也就是说，数组元素只是引用驻留在堆上的引用类型的实例。值类型数组是以内联方式分配的，这意味着数组元素是值类型的实际实例。因此，值类型阵列的分配和释放比引用类型阵列的分配和释放更便宜。此外，在大多数情况下，值类型数组表现出更好的引用位置。

下一个差异与内存使用情况相关。值类型在转换为引用类型或其实现的接口之一时获得装箱。它们会在转换回值类型时进行取消装箱。因为框是在堆上分配并进行垃圾回收的对象，所以太多装箱和取消装箱会对堆、垃圾回收器和最终性能产生负面影响。与此相反，引用类型不会被强制转换。(有关详细信息，请参阅 装箱和取消装箱)。

接下来，引用类型分配复制引用，而值类型分配则复制整个值。因此，大型引用类型的分配比大值类型的分配更便宜。

最后，引用类型通过引用传递，而值类型通过值传递。对引用类型的实例所做的更改将影响指向该实例的所有引用。值类型实例在按值传递时复制。当值类型的实例发生更改时，它不会影响它的任何副本。由于这些副本不是由用户显式创建的，而是在传递参数或返回值时隐式创建的，因此，可以更改的值类型可能会给许多用户造成混淆。因此，值类型应是不可变的。

作为经验法则，框架中的大部分类型都应为类。但在某些情况下，值类型的特征使其更适合使用结构。

✔ 考虑定义结构而不是类的实例，如果该类型的实例较小且通常为短生存期，或者通常嵌入到其他对象中。

✕ 避免定义结构，除非该类型具有以下所有特性：

- 它以逻辑方式表示单个值，与) (、等的基元类型类似。`int` `double`
- 它的实例大小为16字节。
- 它是不可变的。
- 它不需要频繁装箱。

在所有其他情况下，应将类型定义为类。

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 类型设计准则
- 框架设计准则

抽象类设计

2021/11/16 •

✕ 请勿在抽象类型中定义公共或受保护的内部构造函数。

只有在用户需要创建类型的实例时，构造函数才应该是公共的。由于你无法创建抽象类型的实例，因此具有公共构造函数的抽象类型设计不正确，会引起用户的误解。

✔ 请务必在抽象类中定义一个受保护的或内部的构造函数。

受保护的构造函数更常见，在创建子类型时，它仅允许基类进行自己的初始化。

内部构造函数可用于将抽象类的具体实现限制为定义该类的程序集。

✔ 请务必提供至少一种从你交付的每个抽象类继承的具体类型。

这样做有助于验证抽象类的设计。例如，System.IO.FileStream 是 System.IO.Stream 抽象类的一个实现。

请参阅

- 类型设计准则
- 框架设计指南

静态类设计

2021/11/16 •

静态类定义为仅包含静态成员（除了从 System.Object 继承的实例成员和可能的专用构造函数之外）的类。某些语言为静态类提供内置支持。在 C# 2.0 及更高版本中，将一个类声明为静态时，它是密封的、抽象的，并且不能替代或声明任何实例成员。

静态类是纯面向对象的设计与简单性之间的折衷。它们通常用于提供其他操作（如 System.IO.File）的快捷方式、扩展方法的持有者或者无法为其确保完整的面向对象包装器的功能（如 System.Environment）。

✔ 请务必要少使用静态类。

静态类应该只用作框架的面向对象核心的支持类。

✕ 请勿将静态类视为杂项桶。

✕ 请勿在静态类中声明或替代实例成员。

✔ 如果你的编程语言没有对静态类的内置支持，请务必将静态类声明为密封的且抽象的，并添加一个专用实例构造函数。

请参阅

- 类型设计准则
- 框架设计指南

接口设计

2021/11/16 •

虽然大多数 API 最好使用类和结构进行建模，但是在某些情况下，接口更合适或者是唯一的选择。

CLR 不支持多重继承（例如，CLR 类不能从多个基类继承），但它允许类型实现一个或多个接口以及从基类继承。因此，接口常用于实现多重继承的效果。例如，IDisposable 是一个接口，该接口允许类型独立于它们要参与的任何其他继承层次结构来支持可处置性。

适合定义接口的另一种情况是创建可由多种类型（包括某些值类型）支持的通用接口。值类型不能从 ValueType 以外的类型继承，但它们可以实现接口，因此，使用接口是提供通用基类型的唯一选择。

✔ 如果你需要使一些通用 API 可由包含值类型的类型集支持，请务必定义一个接口。

✔ 如果你需要针对已从其他类型继承的类型支持一个接口的功能，请考虑定义该接口。

✕ 请避免使用标记接口（没有成员的接口）。

如果你需要将一个类标记为具有特定的特征（标记），则通常使用自定义特性而不是接口。

✔ 请务必提供至少一种作为接口的实现的类型。

这样做有助于验证接口的设计。例如，List<T> 是 IList<T> 接口的一个实现。

✔ 请务必提供至少一个使用你定义的每个接口的 API（将接口用作参数的方法或类型化为接口的属性）。

这样做有助于验证接口设计。例如，List<T>.Sort 使用 System.Collections.Generic.IComparer<T> 接口。

✕ 请勿将成员添加到之前已提供的接口。

这样做会破坏接口的实现。你应该创建新的接口，以避免版本控制问题。

除了这些准则中所述的情况外，在设计托管代码可重用库时，通常应该选择类而不是接口。

请参阅

- 类型设计准则
- 框架设计指南

结构设计

2021/11/16 •

常规用途值类型最常被称为结构，即其 C# 关键字。本部分提供常规结构设计的准则。

✕ 请勿为结构提供无参数构造函数。

若遵循此准则，便可创建结构数组，而不必对每个数组项运行该构造函数。请注意，C# 不允许结构具有无参数构造函数。

✕ 请勿定义可变值类型。

可变值类型有几个问题。例如，当属性 getter 返回值类型时，调用方会收到一个副本。由于该副本是隐式创建的，因此开发人员可能不会意识到他们正在改变副本，而不是原始值。此外，某些语言（尤其是动态语言）在使用可变值类型方面存在问题，因为即使是本地变量，在被取消引用后，也会导致生成一个副本。

✔ 请务必确保将所有实例数据设置为零、false 或 null（视情况而定）的状态有效。

这可防止在创建结构数组时意外创建无效的实例。

✔ 请务必在值类型上实现 IEquatable<T>。

值类型上的 Object.Equals 方法会导致装箱，且其默认实现效率不高，因为它使用反射。Equals 可具有更好的性能，并且可得到实施，这样就不会导致装箱。

✕ 请勿显式扩展 ValueType。事实上，大多数语言都禁止这样做。

通常，结构可能非常有用，但只应用于不会频繁装箱的小型单个不可变值。

请参阅

- 类型设计准则
- 框架设计指南
- 在类和结构之间选择

枚举设计

2021/11/16 •

枚举是一种特殊类型的值类型。有两种类型的枚举：简单枚举和标记枚举。

简单枚举表示小型关闭的选项集。简单枚举的一个常见示例是一组颜色。

标记枚举旨在支持枚举值的按位运算。标志枚举的一个常见示例是选项的列表。

✔ 确实使用枚举来表示值集的强类型参数、属性和返回值。

✔ 使用枚举而不是静态常量。

✘ 请勿对开放集使用枚举（如操作系统版本、朋友名称等）。

✘ 不要提供旨在供将来使用的保留枚举值。

你始终可以在以后的阶段将值添加到现有枚举。有关向枚举添加值的详细信息，请参阅 将值添加到枚举 。保留值只是污染一组真实值，往往导致用户错误。

✘ 避免公开只包含一个值的枚举。

若要确保未来的 C Api 可扩展性，常见的做法是将保留参数添加到方法签名。此类保留参数可表示为具有单个默认值的枚举。不应在托管 Api 中完成此操作。方法重载允许在未来版本中添加参数。

✘ 不要在枚举中包含 sentinel 值。

尽管它们有时对框架开发人员很有帮助，但对于框架的用户来说，sentinel 值会令人感到困惑。它们用于跟踪枚举的状态，而不是由枚举表示的集中的值之一。

✔ 在简单枚举上提供零值。

请考虑调用值，如 "None"。如果此类值不适合于此特定枚举，则应将基础值指定为零的最常见默认值。

✔ 考虑 Int32 在大多数编程语言中使用（默认值）作为枚举的基础类型，除非满足以下任一条件：

- 枚举是一个标志枚举，你有超过32个标志，或者预计将来会有更多的标志。
- 基础类型需要不同于 Int32 ，与需要不同大小枚举的非托管代码的互操作性更简单。
- 较小的基础类型会显著节省空间。如果希望枚举主要作为控制流的参数使用，则大小没有差别。如果存在以下情况，大小节省可能会很大：
  - 希望枚举在非常频繁实例化的结构或类中用作字段。
  - 您希望用户创建枚举实例的大型数组或集合。
  - 需要序列化大量枚举实例。

对于内存中使用，请注意，托管对象始终是 `DWORD` 一致的，因此，你可以在一个实例中有效地使用多个枚举或其他小型结构将一个较小的枚举打包，以便进行差别，因为总实例大小始终会向上舍入为 `DWORD` 。

✔ 用名词或名词短语复数和简单枚举作为名词或名词短语来命名标志枚举。

✘ 不要直接扩展 System.Enum 。

System.Enum 是 CLR 用来创建用户定义的枚举的特殊类型。大多数编程语言都提供了一个编程元素，该元素可让你访问此功能。例如，在 c # 中， `enum` 关键字用于定义枚举。

设计标志枚举

✔ 应用 System.FlagsAttribute 来标记枚举。不要将此属性应用于简单枚举。

✔ 对标志枚举值使用2的幂，以便可以使用按位 "或" 运算自由合并这些值。

✔ 考虑为常用的标志组合提供特殊的枚举值。

按位运算是一个高级概念，对于简单任务，不应是必需的。ReadWrite 这是一个特殊值的示例。

✕ 避免创建标志枚举，其中某些值的组合无效。

✕ 避免使用值为零的标记枚举值，除非该值表示 "所有标志均已清除" 并正确命名，如下一个准则所述。

✔ DO name 标记枚举的零值 `None` 。对于标志枚举，值始终为 "清除所有标志"。

向枚举添加值

很常见的情况是，在已交付枚举后，需要向其添加值。在从现有 API 返回新添加的值时存在潜在的应用程序兼容性问题，因为编写不当的应用程序可能无法正确处理新值。

✔ 考虑向枚举添加值，而不考虑较小的兼容性风险。

如果对枚举的添加操作导致应用程序不兼容，请考虑添加一个新的 API，该 API 将返回新的和旧的值，并弃用旧的 API，该 API 应继续只返回旧值。这将确保现有的应用程序保持兼容。

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 类型设计准则
- 框架设计准则

嵌套类型

2021/11/16 •

嵌套类型是在另一种类型(称为封闭类型)的范围内定义的类型。嵌套类型可访问其封闭类型的所有成员。例如，它可以访问在封闭类型中定义的专用字段，还可以访问在封闭类型的所有祖先类型中定义的受保护字段。

一般而言，应慎用嵌套类型。其原因有若干：有些开发人员并不完全熟悉概念。例如，这些开发人员可能在声明嵌套类型的变量的语法方面遇到了问题。嵌套类型也与其封闭类型紧密耦合，因此嵌套类型不适合用作通用类型。

嵌套类型最适合对其封闭类型的实现详细信息进行建模。最终用户应该很少需要声明嵌套类型的变量，也几乎永远不需要显式实例化嵌套类型。例如，集合的枚举器可以是该集合的嵌套类型。枚举器通常通过其封闭类型进行实例化，由于许多语言支持 foreach 语句，因此枚举器变量很少需要由最终用户声明。

✔ 当嵌套类型和其外部类型之间的关系使得成员可访问性语义可取时，请务必使用嵌套类型。

✕ 请勿使用公共嵌套类型作为逻辑分组构造;对此使用命名空间。

✕ 请避免使用公开暴露的嵌套类型。对此，唯一的例外情况是：只需在很少的情况(如子类化或其他高级自定义场景)下声明嵌套类型的变量时。

✕ 如果嵌套类型可能在包含类型之外被引用，则请勿使用该类型。

例如，一个传递给在类上定义的方法的枚举不应被定义为类中的嵌套类型。

✕ 如果嵌套类型需要通过客户端代码进行实例化，则请勿使用该类型。如果某个类型具有公共构造函数，则它可能不应被嵌套。

如果某个类型可以被实例化，这似乎表明该类型在框架中有自己的位置(你可以创建它，使用它并销毁它，而不必使用外部类型)，因此该类型不应被嵌套。如果内部类型与外部类型没有任何关系，则不应在外部类型之外广泛重用内部类型。

✕ 请勿将嵌套类型定义为接口的成员。许多语言不支持此类构造。

请参阅

- 类型设计准则
- 框架设计指南

成员设计准则

2021/11/16 •

方法、属性、事件、构造函数和字段统称为成员。成员最终是指向框架的最终用户公开框架功能的方式。

成员可以为虚拟或非虚拟、具体或抽象、静态或实例，并可具有多个不同的可访问性范围。所有这些组件都提供令人难以置信的表现力，但同时需要注意框架设计器的一部分。

本章提供设计任何类型的成员时应遵循的基本准则。

本节内容

成员重载
属性设计
构造函数设计
事件设计
现场设计
扩展方法
运算符重载
参数设计

经许可重印皮尔逊教育，Inc. 的作者 :从框架设计指导原则:用于可重复使用的 .Net 库的约定、惯例和模式; 第2版 By Krzysztof Cwalina，Brad Abrams，通过 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分2008发布。

另请参阅

- 框架设计准则

成员重载

2021/11/16 •

成员重载意味着针对同一类型创建两个或多个成员，这些成员仅在参数的数量或类型上不同，但具有相同的名称。例如，在以下示例中，重载了 `WriteLine` 方法：

```
public static class Console {
    public void WriteLine();
    public void WriteLine(string value);
    public void WriteLine(bool value);
    ...
}
```

由于只有方法、构造函数和索引属性可以有参数，因此只能重载这些成员。

重载是提高可重用库的可用性、生产力和可读性的最重要方法之一。通过对参数数量进行重载，可提供更简单的构造函数和方法版本。通过对参数类型进行重载，可为那些针对不同类型的选定集执行相同操作的成员使用相同的成员名称。

✔ 请务必尝试使用描述性参数名称来指示较短重载使用的默认值。

✕ 请避免在重载中随意改变参数名称。如果一个重载中的参数与另一个重载中的参数表示相同的输入，则这些参数应该具有相同的名称。

✕ 请避免重载的成员中参数的顺序不一致。在所有重载中，具有相同名称的参数应出现在相同的位置。

✔ 请确保仅使最长的重载虚拟化（如果需要扩展性）。较短的重载应直接调用较长的重载。

✕ 请勿使用 `ref` 或 `out` 修饰符来重载成员。

某些语言无法像这样解析对重载的调用。此外，此类重载通常具有完全不同的语义，可能不应该是重载，而应该是两种单独的方法。

✕ 请勿包含参数位于相同位置且类型相似但语义不同的重载。

✔ 请务必允许为可选参数传递 `null`。

✔ 请务必使用成员重载，而不是使用默认参数定义成员。

默认参数不符合 CLS。

请参阅

- 成员设计准则
- 框架设计指南

属性设计

2021/11/16 •

虽然从技术上来说，属性与方法很相似，但它们在使用方案方面大不一样。它们应被视作智能字段。它们具有字段的调用语法，同时兼备方法的灵活性。

✔ 如果调用方不能更改属性的值，则要创建 get-only 属性。

请记住，如果属性属于可变引用类型，则即使属性是 get-only，也可更改属性值。

✘ 请勿提供 set-only 属性，也不要提供其 Setter 可访问性范围比 Getter 大的属性。

例如，不要使用具有公共 Setter 和受保护的 Getter 的属性。

如果无法提供属性 Getter，请改为以方法的形式实现功能。请考虑将 `Set` 作为方法名称的开头，后跟你向属性赋予的名称。例如，AppDomain 有一个名为 `SetCachePath` 的方法，而不是有一个名为 `CachePath` 的 set-only 属性。

✔ 请为所有属性提供敏感默认值，确保默认值不会导致安全漏洞或超级低效的代码。

✔ 允许以任意顺序设置属性，即使这会导致对象的状态暂时无效也是如此。

普遍的情况是两个或更多属性在一个点上相互关联；在这个点上，鉴于同一对象上其他属性的值，某一属性的某些值可能无效。在这种情况下，因无效状态导致的异常应当延迟，直到相互关联的属性实际上由该对象一起使用为止。

✔ 如果属性 Setter 引发异常，则要保留先前的值。

✘ 避免从属性 Getter 引发异常。

属性 Getter 应为简单操作，不得具有任何前置条件。如果 Getter 可能会引发异常，则它可能会被重新设计为方法。请注意，此规则不适用于索引器；在索引器中，我们预计验证参数时会引发异常。

索引属性设计

索引属性是一种特殊属性，它可具有参数，且可使用与数组索引类似的特殊语法进行调用。

索引属性通常被称为索引器。索引器应仅在提供对逻辑集合中项目的访问的 API 中使用。例如，字符串是字符的集合，并且已添加了 System.String 上的索引器来访问其字符。

✔ 请考虑使用索引器来提供对内部数组中存储的数据的访问。

✔ 请考虑对表示项目集合的类型提供索引器。

✘ 不要使用带有多个参数的索引属性。

如果设计需要多个参数，请重新考虑属性是否真的表示逻辑集合的访问器。如果不表示此内容，请改为使用方法。请考虑以 `Get` 或 `Set` 作为方法名称的开头。

✘ 避免使用参数类型不是 System.Int32、System.Int64、System.String、System.Object 或枚举的索引器。

如果设计需要其他类型的参数，强烈建议重新评估 API 是否真的表示逻辑集合的访问器。如果不表示此内容，请使用方法。请考虑以 `Get` 或 `Set` 作为方法名称的开头。

✔ 请对索引属性使用名称 `Item`，除非有明显更好的名称（例如，参见 `System.String` 上的 Chars[] 属性）。

在 C# 中，索引器的默认名称为 Item。IndexerNameAttribute 可用于自定义此名称。

✘ 请勿同时提供在语义上等效的索引器和方法。

✕ 请勿在一个类型中提供多个系列的重载索引器。

这是由 C# 编译器实施的。

✕ 请勿使用非默认的索引属性。

这是由 C# 编译器实施的。

属性更改通知事件

有时，提供事件来通知用户属性值出现更改很有用。例如，`System.Windows.Forms.Control` 在其 `Text` 属性的值更改后会引发 `TextChanged` 事件。

✔ 请考虑在高级 API（通常是设计器组件）中的属性值修改时引发更改通知事件。

如果用户需要在某对象的某个属性出现更改时知道该情况，则该对象应对该属性引发更改通知事件。

不过，对于基类型或基础集合等低级 API，引发此类事件的开销太大，不太值得这样做。例如，当新项目添加到列表且 `Count` 属性更改时，List<T> 不会引发此类事件。

✔ 请考虑当属性的值通过外部强制操作发生更改时引发更改通知事件。

如果某属性值通过某个外部强制操作（通过在对象上调用方法之外的方式），则会引发事件，向开发人员指出该值正在更改且已经更改。文本框控件的 `Text` 属性就是一个很好的例子。当用户在 `TextBox` 中键入文本时，属性值会自动更改。

请参阅

- 成员设计准则
- 框架设计指南

构造函数设计

2021/11/16 •

有两种类型的构造函数：类型构造函数和实例构造函数。

类型构造函数是静态的构造函数，在使用类型之前由 CLR 运行。实例构造函数在创建类型的实例时运行。

类型构造函数不能采用任何参数。而实例构造函数可以。不采用任何参数的实例构造函数通常称为无参数构造函数。

构造函数是创建类型的实例的最自然方式。大多数开发人员在考虑创建实例的替代方法（如工厂方法）之前，会搜索并尝试使用构造函数。

✔ 请考虑提供简单的（理想情况下为默认设置）构造函数。

简单的构造函数包含的参数非常少，且所有参数均为基元或枚举。此类简单的构造函数可提高框架的可用性。

✔ 如果所需操作的语义没有直接映射到新实例的构造，或者如果遵循构造函数设计准则感觉不自然，请考虑使用静态工厂方法而不是构造函数。

✔ 请务必使用构造函数参数作为设置主属性的快捷方式。

使用后跟一些属性集的空构造函数和使用带有多个参数的构造函数在语义上应该不存在任何差异。

✔ 如果构造函数参数只是用来设置属性，请务必对构造函数参数和属性使用相同的名称。

此类参数和属性的唯一区别应该是大小写形式。

✔ 在构造函数中执行最少的工作。

除了捕获构造函数参数外，构造函数不应执行太多工作。任何其他处理的成本都应延迟到需要时再进行。

✔ 在适当情况下，请务必从实例构造函数引发异常。

✔ 如果需要公共无参数构造函数，请务必在类中显式声明此类构造函数。

如果未针对某个类型显式声明任何构造函数，则许多语言（如 C#）将自动添加一个公共无参数构造函数。（抽象类获取一个受保护的构造函数。）

向类添加参数化构造函数会阻止编译器添加无参数构造函数。这通常会导致意外的中断性变更。

✘ 请避免对结构显式定义无参数构造函数。

这样可以更快地创建数组，因为如果没有定义无参数构造函数，它就不必在数组的每个槽上运行。请注意，由于此原因，包许多编译器（包括 C#）不允许结构具有无参数构造函数。

✘ 请避免在对象的构造函数中对该对象调用虚拟成员。

调用虚拟成员将导致调用派生度最高的替代，即使派生度最高的类型的构造函数尚未完全运行也是如此。

类型构造函数指南

✔ 请务必使静态构造函数专用。

静态构造函数（也称为类构造函数）用于初始化类型。在创建第一个类型实例或调用该类型的任何静态成员之前，CLR 调用静态构造函数。用户无法控制调用静态构造函数的时间。如果静态构造函数不是专用的，则 CLR 以外的代码可以调用它。根据构造函数中执行的操作，这可能导致意外行为。C# 编译器会强制使静态构造函数专用。

✕ 请勿从静态构造函数引发异常。

如果从类型构造函数引发了一个异常，则该类型在当前应用程序域中不可用。

✔ 请考虑使用静态构造函数以内联方式（而非显式地）初始化静态字段，因为运行时能够优化不具有显式定义的静态构造函数的类型的性能。

请参阅

- 成员设计准则
- 框架设计指南

事件设计

2021/11/16 •

事件是最常用的回叫形式(“回叫”就是允许框架调入用户代码的构造)。其他回叫机制包括采用委托的成员、虚拟成员和基于接口的插件。可用性调查中的数据显示，大多数开发人员更喜欢使用事件，而不是使用其他回叫机制。事件与 Visual Studio 和多种语言完美集成。

值得注意的是存在两组事件:在系统状态更改前引发的事件(称为前期事件)，以及在状态更改后引发的事件(称为后期事件)。前期事件的一个例子是 `Form.Closing`，它是在窗体关闭前引发的。后期事件的一个例子是 `Form.Closed`，它是在窗体关闭后引发的。

✔ 请对事件使用“raise”(引发)这个词，而不是使用“fire”(激发)或“trigger”(触发)。

✔ 请使用 System.EventHandler<TEventArgs>，而不是手动创建新的委托来用作事件处理程序。

✔ 请考虑将 EventArgs 的子类用作事件参数，除非你完全确信事件将永不需要将任何数据传递到事件处理方法(在此情况下，你可直接使用 `EventArgs` 类型)。

如果你直接使用 `EventArgs` 来传递 API，则你永远无法在不破坏兼容性的情况下添加任何要与事件一起传递的数据。如果你使用子类，则即使最开始完全为空，你也将能够在需要时向子类添加属性。

✔ 请使用受保护的虚拟方法来引发每个事件。这仅适用于非密封类上的非静态事件，不可用于结构、密封类和静态事件。

此方法旨在使派生类能够处理使用替代项的事件。对于处理派生类中的基类事件，替代操作更加灵活、更加快速，也更自然。按照约定，方法的名称应以“On”开头，后跟事件的名称。

派生类可选择不在其替代中调用方法的基础实现。为此，请不要在为使基类正常工作而所需的方法中包含任何处理。

✔ 请将一个参数传递给会引发事件的受保护的方法。

该参数应被命名为 `e`，且其类型应为事件参数类。

✘ 在引发非静态事件时，请勿将 NULL 作为发送方传递。

✔ 在引发静态事件时，请将 NULL 作为发送方传递。

✘ 在引发事件时，请勿将 NULL 作为事件数据参数传递。

如果不想向事件处理方法传递任何数据，则应传递 `EventArgs.Empty`。开发人员希望此参数不是 NULL。

✔ 请考虑引发最终用户可取消的事件。这仅适用于前期事件。

使用 System.ComponentModel.CancelEventArgs 或其子类作为事件参数，以允许最终用户取消事件。

自定义事件处理程序设计

在有些情况下无法使用 `EventHandler<T>`，例如当框架需要与不支持泛型的较早版本的 CLR 一起使用时。在这种情况下，你可能需要设计和开发自定义事件处理程序委托。

✔ 请对事件处理程序使用返回类型 void。

事件处理程序可调用多个事件处理方法，可能是对多个对象调用。如果事件处理方法可返回值，则每个事件调用都有多个返回值。

✔ 请使用 `object` 作为事件处理程序的第一个参数的类型，并将其称为 `sender`。

✔ 请使用 System.EventArgs 或其子类作为事件处理程序的第二个参数的类型，并将其称为 `e`。

✕ 事件处理程序上请勿带有两个以上的参数。

请参阅

- 成员设计准则
- 框架设计指南

字段设计

2021/11/16 •

封装原则是面向对象的设计中最重要的概念之一。此原则表明，存储在一个对象中的数据应只能由该对象访问。

解释该原则的一个有效的方法是：应该设计一个类型，以便可以在不破坏代码的情况下对该类型的字段进行更改（名称或类型更改），而不是对该类型的成员进行更改。此解释直接暗示了所有字段都必须是专用的。

我们将常量和静态只读字段排除在这一严格限制之外，因为几乎从定义上来说，此类字段从来不需要更改。

✕ 请勿提供公共的或受保护的实例字段。

应提供用于访问字段的属性，而不是将其设为公共或受保护。

✔ 请务必对永远不会更改的常量使用常量字段。

编译器会直接将常量字段的值直接刻录到调用代码中。因此，如果没有破坏兼容性的风险，常量值永远不会更改。

✔ 请务必对预定义的对象实例使用公共静态 `readonly` 字段。

如果存在类型的预定义实例，请将它们声明为类型本身的公共只读静态字段。

✕ 请勿将可变类型的实例分配到 `readonly` 字段。

可变类型是具有实例的类型，这些实例可在实例化后进行修改。例如，数组、大多数集合和流都是可变类型，但 System.Int32、System.Uri 和 System.String 都是不可变的。引用类型字段的只读修饰符可防止替换存储在该字段中的实例，但不能防止通过调用更改实例的成员来修改字段的实例数据。

请参阅

- 成员设计准则
- 框架设计指南

扩展方法

2021/11/16 •

扩展方法是一项语言功能，允许使用实例方法调用语法来调用静态方法。这些方法必须至少采用一个参数，该参数表示方法要针对其操作的实例。

定义此类扩展方法的类称为 sponsor 类，并且必须声明为静态。若要使用扩展方法，必须导入用于定义 sponsor 类的命名空间。

✕ 请避免盲目地定义扩展方法，特别是对于不属于你的类型。

如果你确实拥有某个类型的源代码，请考虑改用常规实例方法。如果你没有某个类型的源代码，而你又想加一个方法，请务必要小心了。自由地使用扩展方法可能会使那些未设计为具有这些方法的类型的 API 混杂在一起。

✔ 请考虑在以下任何情况下使用扩展方法：

- 要提供与接口的每个实现相关的帮助程序功能（如果所述功能可根据核心接口进行编写）。这是因为不能以其他方式将具体实现分配给接口。例如，`LINQ to Objects` 运算符作为所有 IEnumerable<T> 类型的扩展方法实现。因此，任何 `IEnumerable<>` 实现都自动启用了 LINQ。
- 当实例方法会引入对某个类型的依赖项，但此类依赖项会破坏依赖项管理规则时。例如，从 String 到 System.Uri 的依赖项可能是不可取的，因此从依赖项管理的角度来看，返回 `System.Uri` 的 `String.ToUri()` 实例方法是错误的设计。返回 `System.Uri` 的静态扩展方法 `Uri.ToUri(this string str)` 是一个更好的设计。

✕ 请避免针对 System.Object 定义扩展方法。

VB 用户将无法使用扩展方法语法对对象引用调用此类方法。VB 不支持调用此类方法，因为在 VB 中，将引用声明为对象将强制对它的所有方法调用进行后期绑定（调用的实际成员是在运行时确定的），而对扩展方法的绑定是在编译时确定的（早期绑定）。

请注意，本指南适用于存在相同绑定行为的其他语言，或者不支持扩展方法的其他语言。

✕ 请勿将扩展方法置于与扩展类型相同的命名空间中，除非它用于向接口添加方法或用于依赖项管理。

✕ 请避免使用相同的签名来定义两个或多个扩展方法（即使它们驻留在不同的命名空间中也是如此）。

✔ 如果扩展类型是一个接口，并且在大多数或所有情况下都要使用扩展方法，请考虑在与扩展类型相同的命名空间中定义扩展方法。

✕ 请勿在通常与其他功能相关联的名称空间中定义实现某个功能的扩展方法。而应在与扩展方法所属的功能关联的命名空间中定义这些方法。

✕ 请避免对专用于扩展方法的命名空间进行泛型命名（例如“Extension”）。请改用描述性名称（例如“Routing”）。

请参阅

- 成员设计准则

- 框架设计指南

运算符重载

2021/11/16 •

运算符重载可以使框架类型看起来像是内置语言基元一样。

尽管运算符重载在某些情况下是允许使用的，而且很有用，但还是应该谨慎使用它们。在许多情况下，运算符重载被滥用，例如当框架设计者开始使用运算符来执行本应是简单方法的运算时。以下准则应该可帮助你决定何时以及如何使用运算符重载。

✕ 请避免定义运算符重载，除非在感觉像是基元(内置)类型的类型中。

✔ 请考虑在一个感觉像是基元类型的类型中定义运算符重载。

例如，System.String 定义了 `operator==` 和 `operator!=`。

✔ 请务必在表示数字的结构(如 System.Decimal)中定义运算符重载。

✕ 在定义运算符重载时请勿太刻意。

运算符重载在运算结果显而易见的情况下非常有用。例如，能够从另一个 `DateTime` 中减去一个 DateTime 并得到一个 TimeSpan，这是有意义的。但是，以下操作是不恰当的：使用逻辑 union 运算符来联合两个数据库查询或使用 shift 运算符写入到流中。

✕ 除非至少有一个操作数属于定义重载的类型，否则请勿提供运算符重载。

✔ 请确保以对称方式重载运算符。

例如，如果你重载了 `operator==`，则还应重载 `operator!=`。同样，如果你重载了 `operator<`，则还应重载 `operator>`，诸如此类。

✔ 请考虑为方法提供与每个重载的运算符相对应的友好名称。

许多语言不支持运算符重载。出于此原因，建议重载运算符的类型包含一个辅助方法，该方法具有适当的特定于域的名称且提供等效功能。

下表包含一个运算符列表及相应的友好方法名称。

C# [illegible]	[illegible]	[illegible]
`N/A`	`op_Implicit`	`To<TypeName>/From<TypeName>`
`N/A`	`op_Explicit`	`To<TypeName>/From<TypeName>`
`+ (binary)`	`op_Addition`	`Add`
`- (binary)`	`op_Subtraction`	`Subtract`
`* (binary)`	`op_Multiply`	`Multiply`
`/`	`op_Division`	`Divide`
`%`	`op_Modulus`	`Mod or Remainder`

C# ㄱㄱㄱㄱ	ㄱㄱㄱㄱ	ㄱㄱㄱ
`^`	`op_ExclusiveOr`	`Xor`
`& (binary)`	`op_BitwiseAnd`	`BitwiseAnd`
`|`	`op_BitwiseOr`	`BitwiseOr`
`&&`	`op_LogicalAnd`	`And`
`||`	`op_LogicalOr`	`Or`
`=`	`op_Assign`	`Assign`
`<<`	`op_LeftShift`	`LeftShift`
`>>`	`op_RightShift`	`RightShift`
`N/A`	`op_SignedRightShift`	`SignedRightShift`
`N/A`	`op_UnsignedRightShift`	`UnsignedRightShift`
`==`	`op_Equality`	`Equals`
`!=`	`op_Inequality`	`Equals`
`>`	`op_GreaterThan`	`CompareTo`
`<`	`op_LessThan`	`CompareTo`
`>=`	`op_GreaterThanOrEqual`	`CompareTo`
`<=`	`op_LessThanOrEqual`	`CompareTo`
`*=`	`op_MultiplicationAssignment`	`Multiply`
`-=`	`op_SubtractionAssignment`	`Subtract`
`^=`	`op_ExclusiveOrAssignment`	`Xor`
`<<=`	`op_LeftShiftAssignment`	`LeftShift`
`%=`	`op_ModulusAssignment`	`Mod`
`+=`	`op_AdditionAssignment`	`Add`
`&=`	`op_BitwiseAndAssignment`	`BitwiseAnd`
`|=`	`op_BitwiseOrAssignment`	`BitwiseOr`

C# [illegible]	[illegible]	[illegible]
`,`	`op_Comma`	`Comma`
`/=`	`op_DivisionAssignment`	`Divide`
`--`	`op_Decrement`	`Decrement`
`++`	`op_Increment`	`Increment`
`- (unary)`	`op_UnaryNegation`	`Negate`
`+ (unary)`	`op_UnaryPlus`	`Plus`
`~`	`op_OnesComplement`	`OnesComplement`

重载运算符 ==

重载 `operator ==` 非常复杂。该运算符的语义需要与其他一些成员(如 Object.Equals)兼容。

转换运算符

转换运算符是允许从一种类型转换为另一种类型的一元运算符。该运算符必须对操作数或返回类型定义为静态成员。有两种类型的转换运算符:隐式和显式。

✕ 如果最终用户没有明确期望进行此类转换,请勿提供转换运算符。

✕ 请勿在类型的域外定义转换运算符。

例如,Int32、Double 和 Decimal 均为数值类型,而 DateTime 则不是。因此,应该没有转换运算符能够将 `Double(long)` 转换为 `DateTime`。在这种情况下,首选使用构造函数。

✕ 如果转换可能有损,请勿提供隐式转换运算符。

例如,不应存在从 `Double` 到 `Int32` 的隐式转换,因为 `Double` 的范围比 `Int32` 大。即使转换可能会有损,也可以提供显式转换运算符。

✕ 请勿从隐式强制转换引发异常。

最终用户很难了解发生的情况,因为他们可能没有意识到正在进行转换。

✔ 如果对强制转换运算符的调用导致有损转换,而该运算符的协定不允许有损转换,请务必引发 System.InvalidCastException。

请参阅

- 成员设计准则
- 框架设计指南

参数设计

2021/11/16 •

本部分就参数设计提供广泛的指导，其中有内容介绍自变量检查指南。此外，你还应查看命名参数中描述的指南。

✔ 请使用提供成员所需功能的最低派生参数类型。

例如，假设你想要设计这样一种方法，它可枚举集合并将每个项目输出到控制台。此类方法应将 IEnumerable 用作参数，而不是使用 ArrayList 或 IList 等内容。

✕ 请勿使用保留的参数。

如果在某一将来版本中需要对某成员输入更多内容，则可添加新的重载。

✕ 请勿具有将指针、指针数组或多维数组用作参数的公开方法。

指针和多维数组相对而言很难正确使用。几乎在任何情况下，API 都可重新设计来避免将这些类型用作参数。

✔ 请替换跟在所有按值参数和 `ref` 参数后面的所有 `out` 参数（参数数组除外），即使这会导致重载之间的参数排序不一致也是如此（具体请参见成员重载）。

`out` 参数可被视为额外的返回值，将它们组合在一起可使方法签名更易于理解。

✔ 在替代成员或实现接口成员时，请在参数命名方面保持一致。

这可更好地在方法之间传达关系。

在枚举参数和布尔参数之间选择

✔ 如果不使用枚举，成员会具有两个或更多布尔参数，那么请使用枚举型。

✕ 请勿使用布尔型，除非你完全确信永远不需要两个以上的值。

枚举型让你有空间将来添加值，但你应了解向枚举添加值所带来的全部影响，具体可查看枚举设计。

✔ 请考虑为是真正的双状态值且仅用于初始化布尔属性的构造函数参数使用布尔型。

验证自变量

✔ 请验证传递到公共、受保护或已显式实现的成员的自变量。引发 System.ArgumentException；如果验证失败，则引发其某个子类。

请注意，并非一定要在公共成员或受保护成员自身进行实际验证。在某些专用或内部例程中，也可在更低级别进行验证。要点是被公开给最终用户的整个表面区域都对自变量进行检查。

✔ 如果传递了 NULL 自变量，而成员不支持 NULL 自变量，则引发 ArgumentNullException。

✔ 请验证枚举参数。

不要假设将在枚举定义的范围内的枚举自变量。CLR 允许将任何整数值强制转换为枚举值，即使值未在枚举中定义也是如此。

✕ 请勿将 Enum.IsDefined 用于枚举范围检查。

✔ 请注意可变参数在经过验证后可能会更改。

如果成员对安全性很敏感，则请进行复制，然后再验证和处理参数。

参数传递

从框架设计者的角度来看，存在三组主要的参数：按值参数、`ref` 参数和 `out` 参数。

当自变量通过按值参数传递时，成员会收到传入的实际自变量的副本。如果自变量是值类型，则会在堆栈上放置自变量的副本。如果自变量是引用类型，则会在堆栈上放置引用的副本。最常用的 CLR 语言（例如 C#、VB.NET 和 C++）默认为按值传递参数。

当自变量通过 `ref` 参数传递时，成员会收到对传入的实际自变量的引用。如果自变量是值类型，则会在堆栈上放置对自变量的引用。如果自变量是引用类型，则会在堆栈上放置对该引用的引用。`Ref` 参数可用于允许成员修改调用方传递的自变量。

`Out` 参数与 `ref` 参数类似，但有一些细微的区别。参数最初被视为未分配，在分配了某个值之前无法在成员主体中读取。此外，在成员返回之前，必须向参数分配某个值。

✕ 不要使用 `out` 和 `ref` 参数。

若要使用 `out` 或 `ref` 参数，需要具有使用指针的经验，了解值类型和引用类型的区别，还能处理具有多个返回值的方法。另外，`out` 和 `ref` 参数之间的区别并未得到广泛了解。针对一般受众进行设计的框架架构师不应指望用户能熟练运用 `out` 或 `ref` 参数。

✕ 请勿按引用来传递引用类型。

此规则存在一些有限的例外情况，例如可用于交换引用的方法。

参数数目可变的成员

可提供一个数组参数来表示采用可变数量的参数的成员。例如，String 提供以下方法：

```
public class String {
    public static string Format(string format, object[] parameters);
}
```

然后，用户可调用 String.Format 方法，如下所示：

```
String.Format("File {0} not found in {1}",new object[]{filename,directory});
```

如果向数组参数添加 C# params 关键字，会将参数更改为所谓的 params 数组参数，并提供用于创建临时数组的快捷方式。

```
public class String {
    public static string Format(string format, params object[] parameters);
}
```

这样，用户就可通过直接在自变量列表中传递数组元素来调用方法。

```
String.Format("File {0} not found in {1}",filename,directory);
```

请注意，params 这一关键字仅可添加到参数列表中的最后一个参数。

✔ 如果预计最终用户会传递具有少量元素的数组，那么请考虑向数组参数添加 params 关键字。如果在常见方案中将传递大量元素，那么用户将可能根本不以内联方式传递这些元素，因此不需要使用 params 关键字。

✕ 如果调用方几乎总是已在数组中具有输入，那么请不要使用 params 数组。

例如，几乎永远不会通过传递单个字节来调用具有字节数组参数的成员。因此，.NET Framework 中的字节数组参数不会使用 params 关键字。

✕ 如果通过采用 params 数组参数的成员修改数组，那么请勿使用 params 数组。

事实上很多编译器会将成员的自变量转换为调用站点处的临时数组，因此该数组可能是一个临时对象，这使得对该数组的所有修改都将丢失。

✔ 请考虑在简单重载中使用 params 关键字，即使更复杂的重载不可使用它也是如此。

请自我提问，了解用户即使不在所有重载中使用 params 数组，是否也会在一个重载中使用它。

✔ 请尝试对参数排序，以便可使用 params 参数。

✔ 在对性能极度敏感的 API 中，请考虑对具有少量自变量的调用提供特定重载和代码路径。

这样，当使用少量自变量调用 API 时，就能避免创建数组对象。通过采用单数形式的数组参数并添加数字后缀来创建参数的名称。

仅当要将完整代码路径用作特例时（不仅仅是创建数组和调用更常规的方法），才应这样操作。

✔ 请注意 NULL 可作为 params 数组自变量传递。

在处理之前，应验证数组是否不是 NULL。

✕ 请勿使用 `varargs` 方法（也就是省略号）。

对于传递可变参数列表，某些 CLR 语言（例如 C++）支持替代约定（称为 `varargs` 方法）。不得在框架中使用此约定，因为它不符合 CLS。

指针参数

通常，指针不得出现在设计良好的托管式代码框架的公共表面区域中。大多数情况下，应封装指针。但在某些情况下，由于可操作性原因需要使用指针，因此在这类情况下使用指针是合适的。

✔ 请对采用指针自变量的所有成员提供替代约定，原因是指针不符合 CLS。

✕ 不要对指针自变量执行自变量检查，此类检查成本高昂。

✔ 请在使用指针设计成员时遵循与指针相关的常见约定。

例如，无需传递开始索引，原因是可使用简单的指针算法实现这一效果。

请参阅

- 成员设计准则
- 框架设计指南

扩展性设计

2021/11/16 •

设计框架的一个重要方面是确保框架的扩展性得到了仔细的考虑。这需要你了解与各种扩展性机制相关的成本和优势。本章可帮助你确定哪些扩展性机制(子类化、事件、虚拟成员、回调等)可最大程度地满足你的框架的要求。

可通过多种方式在框架中提供扩展性。它们从功能较弱但价格较低到功能强大但价格昂贵不等。对于任何给定的扩展性要求，你应选择可满足要求的成本最低的扩展性机制。请记住，通常可以在之后添加更多的扩展性，但是如果不引入中断性变更，你永远也无法删除它。

本节内容

未密封类
受保护的成员
事件和回调
虚拟成员
抽象(抽象类型和接口)
用于实现抽象的基类
密封

请参阅

- 框架设计指南

未密封类

2021/11/16 •

不能从密封类继承，密封类阻止了扩展性。相比之下，可从其继承的类称为非密封类。

✔ 请考虑使用未添加虚拟成员或受保护成员的非密封类，这是向框架提供廉价但非常受欢迎的扩展性的好方法。

开发人员通常希望从非密封类继承，以便添加便捷成员，如自定义构造函数、新方法或方法重载。例如，`System.Messaging.MessageQueue` 是非密封的，因此允许用户创建默认为特定队列路径的自定义队列，或者添加自定义方法来简化特定场景的 API。

在大多数编程语言中，类是默认不密封的，这也是框架中对大多数类的推荐默认设置。非密封类型提供的扩展性很受框架用户的欢迎，并且由于与非密封类型相关联的测试成本相对较低，因此提供这种扩展性相当便宜。

请参阅

- 框架设计指南
- 扩展性设计
- 密封

受保护的成员

2021/11/16 •

受保护的成员本身不提供任何扩展性，但它们可以通过子类化使扩展性更强大。它们可用于公开高级自定义选项，而不会不必要地使主公共接口复杂化。

框架设计者需要小心处理受保护的成员，因为名称“protected”会给人一种虚假的安全感。任何人都可以对非密封类进行子类化并访问受保护的成员，因此用于公共成员的所有相同的防御性编码实践都适用于受保护的成员。

✔ 请考虑使用受保护的成员进行高级自定义。

✔ 出于安全、文档和兼容性分析的目的，请务必将非密封类中的受保护成员视为公共成员。

任何人都可从类继承并访问受保护的成员。

请参阅

- 框架设计指南
- 扩展性设计

事件和回调

2021/11/16 •

回调是允许框架通过委托回调到用户代码的扩展点。通常通过方法的参数将这些委托传递到框架。

事件是回调的一种特例，它支持便捷且一致的语法，用于提供委托（事件处理程序）。此外，Visual Studio 的语句完成和设计器还提供了有关使用基于事件的 API 的帮助。（请参阅事件设计。）

✔ 请考虑使用回调以允许用户提供要由框架执行的自定义代码。

✔ 请考虑使用事件以允许用户自定义框架的行为，而无需理解面向对象的设计。

✔ 请务必首选使用事件而非普通回调，因为它们对于更广泛的开发人员来说更熟悉，并且与 Visual Studio 语句完成进行了集成。

✘ 请避免在对性能敏感的 API 中使用回调。

✔ 在使用回调定义 API 时，请务必使用新的 `Func<...>`、`Action<...>` 或 `Expression<...>` 类型，而不是自定义委托。

`Func<...>` 和 `Action<...>` 表示泛型委托。`Expression<...>` 表示函数定义，这些定义可进行编译并随后在运行时进行调用，但也可进行序列化并传递到远程进程。

✔ 请务必衡量并了解使用 `Expression<...>`（而不是使用 `Func<...>` 和 `Action<...>` 委托）对性能的影响。

`Expression<...>` 类型在大多数情况下在逻辑上等同于 `Func<...>` 和 `Action<...>` 委托。它们之间的主要区别在于委托旨在用于本地流程场景，而表达式适用于在远程进程或计算机中对该表达式求值是有益的且可能的情况。

✔ 请务必要明白，调用委托即表示你将执行任意代码，而这可能会对安全性、正确性和兼容性产生影响。

请参阅

- 扩展性设计
- 框架设计指南

虚成员

2021/11/16 •

可替代虚拟成员，从而更改子类的行为。就它们提供的扩展性而言，它们与回调非常相似，但就执行性能和内存消耗而言，它们则更佳。此外，在需要创建一种特殊的现有类型（专用化）的场景中，虚拟成员感觉更自然。

虚拟成员的性能优于回调和事件，但并不比非虚拟方法好。

虚拟成员的主要缺点是虚拟成员的行为只能在编译时进行修改。而回调行为可在运行时进行修改。

与回调（可能不止回调）一样，虚拟成员的设计、测试和维护成本很高，因为对虚拟成员的任何调用都可能以不可预测的方式被替代，并可能执行任意代码。此外，通常还需要执行更多的工作以清楚地定义虚拟成员的协定，因此设计和记录这些成员的成本更高。

✕ 请勿使成员虚拟化，除非你有充分的理由这样做，并且知道与设计、测试和维护虚拟成员相关的所有成本。

在不破坏兼容性的情况下，虚拟成员对可对其进行的更改不太宽容。此外，它们比非虚拟成员慢，主要是因为对虚拟成员的调用没有内联。

✔ 请考虑将扩展性限制在绝对必要的范围内。

✔ 相对于虚拟成员的公共可访问性，请务必首选受保护的可访问性。公共成员应通过调用受保护的虚拟成员提供扩展性（如需要）。

类的公共成员应为该类的直接使用者提供正确的功能集。虚拟成员设计为在子类中被替代，而受保护的可访问性是将所有虚拟扩展点的范围限定在可使用这些扩展点的位置的好方法。

请参阅

- 框架设计指南
- 扩展性设计

抽象（抽象类型和接口）

2021/11/16 •

抽象是一种描述协定但不提供协定的完整实现的类型。抽象通常作为抽象类或接口实现，并且它们附带一组定义明确的引用文档，其中描述了实现协定的类型所需的语义。.NET Framework 中的一些最重要的抽象包括 Stream、IEnumerable<T> 和 Object。

你可实现支持抽象协定的具体类型并将此具体类型与使用该抽象的（针对该抽象操作的）框架 API 结合使用，从而扩展框架。

很难设计一种能够经受时间考验、有意义且有用的抽象。主要难点是获得正确的成员集，不能多也不能少。如果一个抽象有太多的成员，它就会变得难以（甚至不可能）实现。如果它的成员对于承诺的功能而言太少，则在许多令人感兴趣的场景中，它就变得毫无用处。

框架中的抽象太多也会对框架的可用性产生负面影响。如果不理解抽象如何融入具体实现及针对该抽象操作的 API 的大环境，通常就很难理解该抽象。此外，抽象及其成员的名称都必然是抽象的，在没有首先理解它们更广泛的使用上下文的情况下，这通常使得它们难以理解、令人捉摸不透。

不过，抽象提供极其强大的扩展性，这是其他扩展性机制所不能比拟的。它们是插件、控制反转 (IoC)、管道等多个体系结构模式的核心。它们对于框架的可测试性也极其重要。良好的抽象使你可以出于单元测试的目的，剔除大量的依赖项。总之，抽象是面向对象的新式框架所追求的丰富性的原因。

✕ 请勿提供抽象，除非通过开发一些具体实现及使用该抽象的 API 对它们进行了测试。

✔ 设计抽象时，请务必要在抽象类和接口之间进行仔细地选择。

✔ 考虑为抽象的具体实现提供引用测试。此类测试应该允许用户测试其实现是否正确实现了协定。

请参阅

- 框架设计指南
- 扩展性设计

用于实现抽象的基类

2021/11/16 •

严格地说，当一个类派生出另一个类时，这个类就变成了基类。然而，就本部分而言，基类是一个主要设计用来提供一个公共抽象或者让其他类通过继承来重用一些默认实现的类。基类通常位于继承层次结构的中间 - 介于层次结构根的抽象与底部的几个自定义实现之间。

它们充当用于实现抽象的实现帮助程序。例如，框架对有序项集合的一种抽象是 IList<T> 接口。实现 IList<T> 并不简单，因此框架提供了多个基类，如 Collection<T> 和 KeyedCollection<TKey,TItem>，它们充当用于实现自定义集合的帮助程序。

基类本身通常不适合充当抽象，因为它们往往包含太多的实现。例如，`Collection<T>` 基类包含许多实现，这些实现与以下事实有关：它实现非泛型的 `IList` 接口（以便更好地与非泛型集合集成），并且它是存储在内存中某个字段中的项集合。

如前所述，基类可为需要实现抽象的用户提供十分宝贵的帮助，但同时它们也是一个很大的负担。它们增加了外围应用，加深了继承层次结构的深度，因此在概念上使框架复杂化。因此，只有当基类为框架的用户提供重要的价值时，才应该使用基类。如果它们只为框架的实现者提供价值，就应该避免使用它们，在这种情况下，强烈推荐考虑委托给内部实现，而不是从基类继承。

✔ 请考虑使基类抽象，即使它们不包含任何抽象成员也是如此。这清楚地向用户传达了该类专门为从其继承而设计。

✔ 请考虑将基类置于与主流场景类型不同的命名空间中。根据定义，基类旨在用于高级扩展性场景，因此大多数用户对此不感兴趣。

✕ 如果打算在公共 API 中使用基类，请避免用“Base”后缀来命名该类。

请参阅

- 框架设计指南
- 扩展性设计

密封

2021/11/16 •

面向对象的框架的一项功能是，开发人员可采用框架设计者无法预料的方式对其进行扩展和自定义。这是可扩展设计的强大之处，也是危险之处。因此，当你设计框架时，非常重要的一点是在需要时仔细设计以实现扩展性，而在有风险时限制扩展性。

阻止扩展性的强大机制是密封的。可以密封类或单个成员。密封一个类可防止用户从该类继承。密封一个成员可防止用户替代特定成员。

✕ 若没有充分的理由，请勿密封类。

因为想不出扩展性场景而密封类不是一个充分的理由。框架用户喜欢出于各种非显着性的原因而从类中继承，比如添加便捷成员。有关用户要从某类型继承的非显着性原因的示例，请参阅非密封类。

密封一个类的充分理由包括：

- 类是一个静态类。请参阅静态类设计。
- 类在继承的受保护成员中存储对安全性敏感的机密。
- 类继承许多虚拟成员，且单独密封它们的成本将超过使该类不密封的好处。
- 类是一个需要非常快速的运行时查找的特性。密封特性的性能水平略高于非密封特性。请参阅特性。

✕ 请勿对密封类型声明受保护的成员或虚拟成员。

按照定义，不能从密封类型继承。这意味着不能调用密封类型的受保护成员，也不能替代密封类型的虚拟方法。

✔ 请考虑密封你替代的成员。

引入虚拟成员（如虚拟成员中所述）可能导致的问题也同样适用于替代（尽管程度稍低）。密封替代可以使你从继承层次结构中的这一点开始就避免这些问题。

请参阅

- 框架设计指南
- 扩展性设计
- 未密封类

异常设计准则

2021/11/16 •

与基于返回值的错误报告相比，异常处理具有很多优点。良好的框架设计可帮助应用程序开发人员实现异常的优点。本部分讨论异常的优点，并提供有效使用它们的准则。

本节内容

异常引发
使用标准异常类型
异常和性能

请参阅

- 框架设计指南

异常引发

2021/11/16 •

本部分中所述的异常引发准则要求对执行失败的含义有一个很好的定义。每当一个成员无法执行它旨在要执行的操作(成员名称所指的操作)时，执行失败就会发生。例如，如果 `OpenFile` 方法不能将打开的文件句柄返回给调用方，这将被视为执行失败。

大多数开发人员已经习惯将异常用于使用错误(例如被零除或空引用)。在框架中，异常用于所有错误情况，包括执行错误。

✕ 请勿返回错误代码。

异常是在框架中报告错误的主要方法。

✔ 请务必通过引发异常来报告执行失败。

✔ 在代码遇到无法安全地进行进一步执行的情况时，请考虑通过调用 `System.Environment.FailFast` (.NET Framework 2.0 功能)来终止进程，而不是引发一个异常。

✕ 如果可能，请勿对正常控制流使用异常。

除了可能出现争用条件的系统故障和操作之外，框架设计者还应设计 API，使用户能够编写不引发异常的代码。例如，你可提供一种在调用成员之前检查前置条件的方法，使用户可以编写不引发异常的代码。

用于检查另一个成员的前置条件的成员通常称为测试者，而实际执行该工作的成员称为执行者。

在某些情况下，“测试者-执行者”模式可能会产生不可接受的性能开销。在此类情况下，应考虑使用所谓的“尝试-分析”模式(有关详细信息，请参阅异常和性能)。

✔ 请考虑引发异常对性能的影响。每秒 100 次以上的引发率可能会显著影响大多数应用程序的性能。

✔ 请务必记录所有由可公开调用的成员因违反成员协定(而不是系统故障)而引发的异常，并将它们视为你协定的一部分。

作为协定一部分的异常不应从一个版本更改为下一个版本(即不应更改异常类型，也不应添加新异常)。

✕ 请勿包含可基于某些选项引发或不引发的公共成员。

✕ 请勿包含将异常作为返回值或 `out` 参数返回的公共成员。

从公共 API 返回异常，而不是引发异常，这会使基于异常的错误报告的许多好处无法实现。

✔ 请考虑使用异常生成器方法。

从不同的位置引发相同的异常是很常见的情况。若要避免代码膨胀，请使用创建异常并初始化其属性的帮助程序方法。

此外，引发异常的成员不会被内联。将 throw 语句移动到生成器中可能会允许该成员内联。

✕ 请勿从异常筛选器块中引发异常。

当异常筛选器引发一个异常时，CLR 将捕获该异常，并且筛选器将返回 false。此行为与筛选器显式执行并返回 false 是无法区分的，因此很难进行调试。

✕ 请避免从 finally 块显式引发异常。由于调用引发的方法而隐式引发的异常是可以接受的。

在 Pearson Education, Inc. 授权下，由 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分再版自 Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition（Framework 设计准则：可重用 .NET 库的约定、惯例和模式第 2 版），由 Krzysztof Cwalina 和 Brad Abrams 发布于 2008 年 10 月 22 日。

请参阅

- 框架设计指南
- 异常设计准则

使用标准异常类型

2021/11/16 •

本部分介绍框架提供的标准异常及其用法的详细信息。列表并不完整。请参阅 .NET Framework 参考文档，以了解其他框架异常类型的用法。

Exception 和 SystemException

✕ 不要引发 System.Exception 或 System.SystemException 。

✕ 不要捕获 `System.Exception` `System.SystemException` 框架代码中的或，除非你打算重新引发。

✕ 避免捕获 `System.Exception` 或 `System.SystemException` （顶级异常处理程序除外）。

ApplicationException

✕ 不要引发或派生自 ApplicationException 。

InvalidOperationException

InvalidOperationException如果对象处于不适当的状态，✔会引发。

ArgumentException、System.argumentnullexception 和 ArgumentOutOfRangeException

ArgumentException如果向成员传递了错误的参数，✔将引发或其子类型之一。如果适用，更喜欢派生程度最高的异常类型。

✔ `ParamName` 在引发的子类之一时设置属性 `ArgumentException` 。

此属性表示导致引发异常的参数的名称。请注意，可以使用构造函数重载之一来设置属性。

✔用于 `value` 属性 setter 的隐式值参数的名称。

NullReferenceException、IndexOutOfRangeException 和 AccessViolationException

✕ 不要允许可公开调用的 Api 显式或隐式引发 NullReferenceException 、AccessViolationException 或 IndexOutOfRangeException 。这些异常由执行引擎保留并引发，在大多数情况下表示 bug。

执行参数检查以避免引发这些异常。引发这些异常会公开方法的实现详细信息，这些详细信息可能会随时间变化。

StackOverflowException

✕ 不要显式引发 StackOverflowException 。异常只应由 CLR 显式引发。

✕ 请勿捕获 `StackOverflowException` 。

几乎无法编写在存在任意堆栈溢出时保持一致的托管代码。CLR 的非托管部分保持一致，方法是使用探测将堆栈溢出移到明确定义的位置，而不是通过从任意堆栈溢出中进行回退。

OutOfMemoryException

✕ 不要显式引发 OutOfMemoryException 。此异常仅由 CLR 基础结构引发。

ComException、SEHException 和 ExecutionEngineException

✕ 不要显式引发 COMException 、ExecutionEngineException 和 SEHException 。这些异常仅由 CLR 基础结构引发。

另请参阅

- 框架设计准则
- 异常设计准则

异常和性能

2021/11/16 •

与异常相关的一个常见问题是，如果异常用于经常失败的代码，则实现的性能是不可接受的。这是一个合理的担忧。当一个成员引发一个异常时，其性能可能会慢几个数量级。但是，在严格遵守不允许使用错误代码的异常准则的同时，也有可能获得良好的性能。本部分中所述的两种模式提供了执行此操作的方法。

✕ 因为担心异常可能会对性能产生负面影响，因此请勿使用错误代码。

若要提高性能，可以使用接下来的两个部分中所述的"测试者-执行者"模式或"尝试-分析"模式。

"测试者-执行者"模式

有时，异常引发成员的性能可以通过将该成员分成两部分来提高。让我们看看 ICollection<T> 接口的 Add 方法。

```
ICollection<int> numbers = ...
numbers.Add(1);
```

如果集合是只读的，则 `Add` 方法引发。在方法调用预计会经常失败的情况下，这可能是一个性能问题。缓解此问题的一种方法是在尝试添加值之前测试集合是否可写。

```
ICollection<int> numbers = ...
...
if (!numbers.IsReadOnly)
{
    numbers.Add(1);
}
```

用于测试条件的成员（在本示例中为 `IsReadOnly` 属性）被称为测试者。用于执行潜在引发操作的成员（在本示例中为 `Add` 方法）被称为执行者。

✔ 请考虑对可能在常见场景中引发异常的成员使用"测试者-执行者"模式，以避免与异常相关的性能问题。

"尝试-分析"模式

对于对性能极其敏感的 API，应使用比前一部分中所述的"测试者-执行者"模式更快的模式。该模式要求调整成员名称，使定义完善的测试用例成为成员语义的一部分。例如，DateTime 定义一个 Parse 方法，当字符串的分析失败时，该方法将引发异常。它还定义一个相应的 TryParse 方法，该方法尝试进行分析，但如果分析不成功，则返回 false，而如果分析成功，则使用 `out` 参数返回分析结果。

```
public struct DateTime
{
    public static DateTime Parse(string dateTime)
    {
        ...
    }
    public static bool TryParse(string dateTime, out DateTime result)
    {
        ...
    }
}
```

使用此模式时，务必严格地定义“尝试”功能。如果成员由于定义完善的“尝试”功能以外的任何原因失败，该成员仍必须引发相应的异常。

✔ 请考虑对可能在常见场景中引发异常的成员使用“尝试-分析”模式，以避免与异常相关的性能问题。

✔ 对于实现此模式的方法，请务必使用前缀“Try”和布尔返回类型。

✔ 请务必使用“尝试-分析”模式为每个成员提供一个异常引发成员。

请参阅

- 框架设计指南
- 异常设计准则

使用准则

2021/11/16 •

本部分包含有关在可公开访问的 API 中使用常见类型的准则。其中涉及内置框架类型（如序列化特性）的直接使用和常见运算符的重载。

System.IDisposable 接口在本部分未进行介绍，但在释放模式部分中进行了讨论。

> **NOTE**
> 有关其他常见内置 .NET Framework 类型的指南和附加信息，请参阅以下各项的引用主题：System.DateTime、System.DateTimeOffset、System.ICloneable、System.IComparable<T>、System.IEquatable<T>、System.Nullable<T>、System.Object、System.Uri。

本节内容

数组
特性
集合
序列化
System.Xml 使用情况
相等运算符

请参阅

- 框架设计指南

数组（.NET Framework 设计准则）

2021/11/16 •

✔ 在公共 API 中，请务必首选使用集合而非数组。集合部分提供有关如何在集合和数组之间进行选择的详细信息。

✕ 请勿使用只读数组字段。该字段本身是只读的并且不能更改，但可以更改数组中的元素。

✔ 请考虑使用交错数组而不是多维数组。

交错数组是指所包含的元素也是数组的数组。与多维数组相比，构成元素的数组可以是不同的大小，从而使某些数据集（例如稀疏矩阵）浪费的空间更少。此外，CLR 优化了针对交错数组的索引操作，因此在某些情况下，它们可能会表现出更好的运行时性能。

请参阅

- Array
- 框架设计指南
- 使用准则

属性（.NET Framework 设计准则）

2021/11/16 •

System.Attribute 是用于定义自定义特性的基类。

特性是可添加到编程元素（如程序集、类型、成员和参数）的注释。它们存储在程序集的元数据中，并且可在运行时使用反射 API 进行访问。例如，框架定义了 ObsoleteAttribute，可将其应用于类型或成员，以指示该类型或成员已弃用。

特性可具有一个或多个属性，这些属性包含与特性相关的其他数据。例如，`ObsoleteAttribute` 可能包含有关其中某个类型或成员已弃用的版本的其他信息，以及替换已过时 API 的新 API 的说明。

应用某个特性时，必须指定该特性的某些属性。它们被称为必需属性或必需参数，因为它们被表示为位置构造函数参数。例如，ConditionalAttribute 的 ConditionString 属性是一个必需属性。

在应用特性时不一定要指定的属性称为可选属性（或可选参数）。它们由可设置的属性表示。应用某个特性时，编译器提供了特殊的语法来设置这些属性。例如，AttributeUsageAttribute.Inherited 属性表示一个可选参数。

✔ 请务必使用后缀“Attribute”来命名自定义属性类。

✔ 请务必将 AttributeUsageAttribute 应用于自定义属性。

✔ 请务必提供可选参数的可设置属性。

✔ 请务必提供必需参数的仅限获取属性。

✔ 请务必提供构造函数参数来初始化对应于必需参数的属性。每个参数都应具有与相应属性相同的名称（但大小写不同）。

✘ 请避免提供构造函数参数来初始化对应于可选参数的属性。

换句话说，请勿包含可同时使用构造函数和 setter 设置的属性。此准则非常明确地说明了哪些参数是可选的，哪些参数是必需的，并避免了用两种方法来执行相同的操作。

✘ 请避免重载自定义特性构造函数。

只具有一个构造函数，这清楚地告诉了用户哪些参数是必需的，哪些参数是可选的。

✔ 如果可能，请务必密封自定义特性类。这样可以更快地查找特性。

请参阅

- 框架设计指南
- 使用准则

集合准则

2021/11/16 •

专为操作一组具有某些共同特性的对象而设计的类型都可被视为一个集合。这种类型几乎总是适合实现 IEnumerable 或 IEnumerable<T>，因此在本部分中，我们仅将实现其中一个或全部两个接口的类型视为集合。

✕ 请勿在公共 API 中使用弱类型集合。

表示集合项目的所有返回值和参数的类型都应该是确切的项目类型，而不是其任何基类型(这仅适用于集合的公共成员)。

✕ 请勿在公共 API 中使用 ArrayList 和 List<T>。

这些类型是数据结构，根据设计可用于内部实现，不可用于公共 API。`List<T>` 针对性能和功能进行了优化，代价是灵活性和 API 的简洁度。例如，如果你返回 `List<T>`，则你将永远无法在客户端代码修改集合时收到通知。此外，`List<T>` 会公开很多成员(例如 BinarySearch)，它们在很多方案中没有用或不适用。以下两部分描述了为在公共 API 中使用而专门设计的类型(抽象)。

✕ 请勿在公共 API 中使用 `Hashtable` 和 `Dictionary<TKey,TValue>`。

这些类型是数据结构，根据设计可用于内部实现。公共 API 应使用 IDictionary、`IDictionary <TKey, TValue>`，或者使用实现其中一个或全部两个接口的自定义类型。

✕ 请勿使用 IEnumerator<T> 和 IEnumerator，也不要使用其他实现上述任一接口的类型，作为 `GetEnumerator` 方法的返回类型的除外。

从 `GetEnumerator` 之外的方法返回枚举器的类型不可与 `foreach` 语句一起使用。

✕ 请勿在同一类型上同时实现 `IEnumerator<T>` 和 `IEnumerable<T>`。这同样适用于非泛型接口 `IEnumerator` 和 `IEnumerable`。

集合参数

✔ 请尽量使用专业化程度最小的类型作为参数类型。将集合作为参数的大多数成员使用的是 `IEnumerable<T>` 接口。

✕ 如果仅是为了访问 `Count` 属性，请避免使用 ICollection<T> 和 ICollection。

转而请考虑使用 `IEnumerable<T>` 或 `IEnumerable`，并动态检查对象是否实现 `ICollection<T>` 或 `ICollection`。

集合属性和返回值

✕ 请勿提供可设置的集合属性。

用户可先清除集合，然后添加新内容来替换集合的内容。如果常见的情况是替换整个集合，则请考虑对集合提供 `AddRange` 方法。

✔ 请对表示读取/写入集合的属性或返回值使用 `Collection<T>` 或 `Collection<T>` 的子类。

如果 `Collection<T>` 不满足某些要求(例如集合不得实现 IList)，请通过实现 `IEnumerable<T>`、`ICollection<T>` 或 IList<T> 来使用自定义集合。

✔ 请对表示只读集合的属性或返回值使用 `ReadOnlyCollection<T>` 的子类或 ReadOnlyCollection<T>，在极少的情况下请使用 `IEnumerable<T>`。

通常，请优先选择 `ReadOnlyCollection<T>`。如果它不满足某些要求（例如集合不得实现 `IList`），请通过实现 `IEnumerable<T>`、`ICollection<T>` 或 `IList<T>` 来使用自定义集合。如果要实现自定义只读集合，请实现 `ICollection<T>.IsReadOnly` 来返回 `true`。

如果你确定你只想支持的方案是仅向前迭代，那么使用 `IEnumerable<T>` 即可。

✔ 请考虑使用泛型基础集合的子类，而不是直接使用集合。

这样可进行更好的命名，还可添加基础集合类型上不存在的帮助程序成员。这尤其适用于高级 API。

✔ 请考虑从超级常用的方法和属性返回 `Collection<T>` 或 `ReadOnlyCollection<T>` 的子类。

这样，你就能在将来添加帮助程序方法或更改集合实现。

✔ 如果集合中存储的项目具有唯一键（名称、ID 等），请考虑使用键控集合。键控集合可同时由整数和键编制索引，它们通常通过从 `KeyedCollection<TKey,TItem>` 继承来实现。

键控集合通常会占用更大的内存，如果内存开销的权重比拥有键的优点高，那么不得使用这类集合。

✘ 请勿从集合属性返回 NULL 值，也不要从返回集合的方法中返回 NULL 值。转而请返回空集合或空数组。

一般规则是应将 NULL 集合和空（0 项目）集合/数组视为相同。

快照与活动集合

表示某时间点的某状态的集合被称为快照集合。例如，包含从数据库查询返回的行的集合是一个快照。始终表示当前状态的集合被称为活动集合。例如，`ComboBox` 项目的集合是一个活动集合。

✘ 请勿从属性返回快照集合。属性应返回活动集合。

属性 Getter 应该是非常轻量的操作。要返回快照，需要在 O(n) 操作中创建内部集合的副本。

✔ 请使用快照集合或活动 `IEnumerable<T>`（或其子类型）表示可变的集合（"可变"是指可在不显式修改集合的情况下进行更改）。

通常，表示共享资源（例如目录中的文件）的所有集合都是可变的。这种集合很难或不可能作为活动集合实现，除非实现只是一个仅前进枚举器。

在数组和集合之间进行选择

✔ 相对于数组，请优选集合。

集合可让你更大程度地控制内容，它们随时间的推移而变化，而且更易于使用。此外，不建议对只读方案使用数组，原因是克隆数组的成本过高。可用性调查显示，某些开发人员感觉更喜欢使用基于集合的 API。

不过，如果你要开发低级 API，则对读写方案使用数组可能效果更好。数组的内存占用更小，这有助于减少工作集，而且由于数组已由运行时进行优化，因此可更快地访问数组中的元素。

✔ 请考虑在低级 API 中使用数组来尽量减少内存耗用和尽量提高性能。

✔ 请使用字节数组而不是字节集合。

✘ 如果每次调用属性 Getter 时，属性都必须返回新数组（例如内部数组的副本），那么请勿对属性使用数组。

实现自定义集合

✔ 设计新集合时，请考虑从 `Collection<T>`、`ReadOnlyCollection<T>` 或 `KeyedCollection<TKey,TItem>` 中继承。

✔ 设计新集合时，请实现 `IEnumerable<T>`。请考虑实现 `ICollection<T>` 或者甚至实现 `IList<T>`（如果合理的话）。

实现此类自定义集合时，请尽可能严格遵守由 `Collection<T>` 和 `ReadOnlyCollection<T>` 建立的 API 模式。也就

是说，显式实现相同的成员，对如同这两个集合一样的参数进行命名等等。

✔ 如果集合通常会传递到将非泛型集合接口（`IList` 和 `ICollection`）视为输入的 API，那么请考虑实现这些接口。

✘ 对于具有与集合概念无关的复杂 API 的类型，请避免实现集合接口。

✘ 请勿从非泛型基础集合（例如 `CollectionBase`）进行继承。请改为使用 `Collection<T>`、`ReadOnlyCollection<T>` 和 `KeyedCollection<TKey,TItem>`。

对自定义集合命名

创建集合（实现 `IEnumerable` 的类型）的原因主要有下面两种：(1) 为了创建这样一个新的数据结构，它具有结构特定的操作且性能特征通常与现有数据结构（例如 List<T>、LinkedList<T>、Stack<T>）不同；以及 (2) 为了创建一个专用于保存特定一组项目（例如 StringCollection）的集合。数据结构最常用于应用程序和库的内部实现。专用集合主要在 API 中（作为属性和参数类型）公开。

✔ 请在实现 `IDictionary` 或 `IDictionary<TKey,TValue>` 的抽象的名称中使用“Dictionary”后缀。

✔ 请在实现 `IEnumerable`（或其任何后代）和表示项目列表的类型的名称中使用“Collection”后缀。

✔ 请为自定义数据结构使用适当的数据结构名称。

✘ 不要在集合抽象的名称中使用任何暗指特定实现的后缀，例如“LinkedList”或“Hashtable”。

✔ 请考虑对集合名称使用项目类型的名称作为前缀。例如，存储类型为 `Address`（实现 `IEnumerable<Address>`）的项目的集合应被命名为 `AddressCollection`。如果项目类型为接口，则可省略项目类型的前缀“I”。因此，可将IDisposable 项目的集合称为 `DisposableCollection`。

✔ 如果相应的可写集合可添加到框架中或者已存在于框架中，请考虑在只读集合的名称中使用“ReadOnly”前缀。

例如，应将字符串的只读集合称为 `ReadOnlyStringCollection`。

请参阅

- 框架设计指南
- 使用准则

序列化

2021/11/16 •

序列化是将对象转换为易于持久化或传输的格式的过程。例如，你可以将一个对象序列化，使用 HTTP 通过 Internet 传输该对象，并在目标计算机上对其进行反序列化。

.NET Framework 提供针对各种序列化场景进行了优化的三种主要序列化技术。下表列出了这些技术以及与这些技术相关的主要 Framework 类型。

[illegible]	[illegible]	[illegible]
[illegible]	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	常规持久性 Web 服务 JSON
XML [illegible]	XmlSerializer	可完全控制 XML 形状的 XML 格式
运行时序列化（二进制和 SOAP）	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	.NET 远程处理

✔ 设计新类型时请务必考虑序列化。

选择要支持的合适的序列化技术

✔ 如果可能需要在 Web 服务中持久保持或使用类型的实例，则应考虑支持数据协定序列化。

✔ 如果需要对序列化类型时生成的 XML 格式具有更多控制，则应考虑改为支持 XML 序列化，或者在支持数据协定序列化之外还支持 XML 序列化。

这对于某些需要使用数据协定序列化所不支持的 XML 构造（例如，用于生成 XML 特性）的互操作性场景可能是必需的。

✔ 如果类型的实例需要越过 .NET 远程处理边界传递，则应考虑支持运行时序列化。

✘ 请避免仅仅因为常规持久性原因而支持运行时序列化或 XML 序列化，而应首选数据协定序列化。

支持数据协定序列化

通过将 DataContractAttribute 应用到类型，并将 DataMemberAttribute 应用到类型的成员（字段和属性），类型可支持数据协定序列化。

✔ 如果可以在部分信任模式下使用类型，则考虑将类型的数据成员标记为公共的。

在完全信任模式下，数据协定序列化程序可对非公共类型和成员进行序列化和反序列化，但在部分信任模式下，仅可对公共成员进行序列化和反序列化。

✔ 请务必在包含 DataMemberAttribute 的所有属性上实现 getter 和 setter。数据协定序列化程序要求该类型的 getter 和 setter 都被视为是可序列化的。（在 .NET Framework 3.5 SP1 中，某些集合属性可以为仅限获取属性。）如果不会在部分信任模式下使用类型，则这两个属性访问器中的一个或两个都可以是非公共的。

✔ 对于反序列化实例的初始化，应考虑使用序列化回调。

反序列化对象时，不调用任何构造函数。（该规则存在例外情况。在反序列化期间，调用标记为 CollectionDataContractAttribute 的集合的构造函数。）因此，在正常构造期间执行的任何逻辑都需要作为一个序列化回调实现。

`OnDeserializedAttribute` 是最常用的回调属性。此系列中的其他属性还有 OnDeserializingAttribute、OnSerializingAttribute 和 OnSerializedAttribute。这些属性可分别用来标记在反序列化之前、序列化之前以及序列化之后执行的回调。

✔ 请考虑使用 KnownTypeAttribute 指示在反序列化复杂对象图时应使用的具体类型。

✔ 创建或更改可序列化的类型时，请务必考虑后向兼容性和前向兼容性。

请记住，类型的未来版本的序列化流可反序列化到类型的当前版本，反之亦然。

你一定要清楚，数据成员（甚至是私有成员和内部成员）不能更改其名称和类型，甚至不能更改其在类型的未来版本中的顺序，除非特别留意将使用显式参数的协定保存到数据协定属性。

在对可序列化的类型进行更改时，测试序列化的兼容性。尝试将新版本反序列化为旧版本，以及将旧版本反序列化为新版本。

✔ 考虑实现 IExtensibleDataObject 以允许在类型的两个不同版本之间进行往返。

序列化程序可通过此接口确保在往返期间不丢失任何数据。使用 IExtensibleDataObject.ExtensionData 属性存储来自当前版本未知的类型的未来版本的任何数据，因此该属性不能将这些数据存储在其数据成员中。在随后序列化当前版本并将其反序列化为未来版本时，可在序列化流中使用附加数据。

支持 XML 序列化

数据协定序列化是 .NET Framework 中的主要（默认）序列化技术，但也存在数据协定序列化不支持的序列化场景。例如，数据协定序列化无法让您完全控制序列化程序生成或使用的 XML 的形状。如果要求此类精细控制，则必须使用 XML 序列化，而且需要设计类型以支持此序列化技术。

✘ 请避免专门针对 XML 序列化设计类型，除非你有很充分的理由要控制所生成的 XML 的形状。此序列化技术已由上一节讨论的数据协定序列化所取代。

✔ 如果通过应用 XML 序列化属性所提供的对于序列化 XML 的形状的控制还无法满足你的需要，则可考虑实现 IXmlSerializable 接口。利用此接口的两种方法（ReadXml 和 WriteXml），你可以完全控制序列化的 XML 流。还可以通过应用 `XmlSchemaProviderAttribute` 来控制为类型生成的 XML 架构。

支持运行时序列化

运行时序列化是 .NET 远程处理所使用的一项技术。如果你认为将会使用 .NET 远程处理传输类型，则需要确保类型支持运行时序列化。

可通过应用 SerializableAttribute 来提供对运行时序列化的基本支持，更高级的场景涉及实现一个简单的运行时可序列化模式（实现 ISerializable 并提供序列化构造函数）。

✔ 如果你的类型将要用于 .NET 远程处理，则应考虑支持运行时序列化。例如，System.AddIn 命名空间使用 .NET 远程处理，因此在 `System.AddIn` 加载项之间交换的所有类型都需要支持运行时序列化。

✔ 如果需要完全控制序列化过程，则应考虑实现运行时可序列化模式。例如，如果您需要在对数据进行序列化或反序列化时转换数据。

此模式非常简单。您需要执行的全部操作就是实现 ISerializable 接口，并提供在反序列化对象时使用的特殊构造函数。

✔ 请务必保护序列化构造函数，并提供完全按照此处示例中所示类型化和命名的两个参数。

```
[Serializable]
public class Person : ISerializable
{
    protected Person(SerializationInfo info, StreamingContext context)
    {
        // ...
    }
}
```

✔ 请务必显式实现 ISerializable 成员。

✔ 请务必将链接要求应用到 ISerializable.GetObjectData 实现。这可确保只有完全受信任的核心和运行时序列化程序才有权访问成员。

在 Pearson Education, Inc. 授权下，由 Addison-Wesley Professional 作为 Microsoft Windows 开发系列的一部分再版自 Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition（Framework 设计准则：可重用 .NET 库的约定、惯例和模式第 2 版），由 Krzysztof Cwalina 和 Brad Abrams 发布于 2008 年 10 月 22 日。

请参阅

- 框架设计指南
- 使用准则

System.Xml 使用情况

2021/11/16 •

本部分讨论 System.Xml 命名空间中可用于表示 XML 数据的几种类型的用法。

✕ 请勿使用 XmlNode 或 XmlDocument 来表示 XML 数据。优选改用 IXPathNavigable、XmlReader、XmlWriter 的实例或 XNode 的子类型。`XmlNode` 和 `XmlDocument` 不是为在公共 API 中公开而设计的。

✔ 请确保使用 `XmlReader`、`IXPathNavigable` 或 `XNode` 的子类型作为接受或返回 XML 的成员的输入或输出。

使用这些抽象，而不是 `XmlDocument`、`XmlNode` 或 XPathDocument，因为这会将方法与内存中 XML 文档的特定实现分离，并允许它们与公开 `XNode`、`XmlReader` 或 XPathNavigator 的虚拟 XML 数据源一起工作。

✕ 如果要创建一个表示基础对象模型或数据源的 XML 视图的类型，请勿将 `XmlDocument` 子类化。

请参阅

- 框架设计指南
- 使用准则

相等运算符

2021/11/16 •

本部分讨论重载相等运算符，并将 `operator==` 和 `operator!=` 视为相等运算符。

✕ 请勿重载其中一个相等运算符而不重载另一个。

✔ 请务必确保 Object.Equals 和相等运算符具有完全相同的语义和相似的性能特征。

这通常意味着在重载相等运算符时，需要重写 `Object.Equals`。

✕ 请避免从相等运算符引发异常。

例如，如果其中一个参数为 null，则返回 false，而不是引发 `NullReferenceException`。

针对值类型的相等运算符

✔ 如果相等性是有意义的，请务必对值类型重载相等运算符。

在大多数编程语言中，值类型没有默认的 `operator==` 实现。

针对引用类型的相等运算符

✕ 请避免对可变引用类型重载相等运算符。

许多语言都具有用于引用类型的内置相等运算符。内置运算符通常实现引用相等性，当默认行为更改为值相等性时，很多开发人员都感到很惊讶。

对于不可变的引用类型，此问题得到了缓解，因为不可变性使发现引用相等性和值相等性之间的差异变得更加困难。

✕ 请避免对引用类型重载相等运算符（如果实现的速度远远低于引用相等性的实现速度）。

请参阅

- 框架设计指南
- 使用准则

常见设计模式

2021/11/16 •

软件模式、模式语言和对立模式有很多书籍，它们可满足各种模式的使用。因此，本章提供了与一组非常有限的模式相关的指南和讨论，这些模式经常在 .NET Framework Api 的设计中使用。

本节内容

依赖项属性
释放模式

另请参阅

- 框架设计准则

依赖项属性

2021/11/16 •

依赖属性 (DP) 是常规属性 - 例如，它将它的值存储在属性存储中，而不是将其存储在类型变量(字段)中。

附加依赖属性是一种作为静态 Get 和 Set 方法建模的依赖属性，这些方法表示描述对象及其容器之间关系(例如 `Button` 对象在 `Panel` 容器上的位置)的"属性"。

✔ 如果你需要依赖属性以支持 WPF 功能(如样式设置、触发器、数据绑定、动画、动态资源和继承)，请务必提供这些属性。

依赖属性设计

✔ 在实现依赖属性时，请务必要从 DependencyObject 或它的一个子类型继承。该类型提供了一种非常有效的属性存储实现，并自动支持 WPF 数据绑定。

✔ 请务必为每个依赖属性提供一个常规 CLR 属性和存储 System.Windows.DependencyProperty 实例的公共静态只读字段。

✔ 请务必通过调用实例方法 DependencyObject.GetValue 和 DependencyObject.SetValue 来实现依赖属性。

✔ 请务必通过在属性名称后加上"Property"后缀来命名依赖属性静态字段。

✘ 请勿在代码中显式设置依赖属性的默认值；请改为在元数据中设置它们。

如果你显式设置了一个属性默认值，则可能会阻止通过某些隐式方式(如样式设置)来设置该属性。

✘ 请勿将标准代码以外的代码置于属性访问器中来访问静态字段。

如果属性是通过隐式方式(如样式设置)设置的，则该类代码不会执行，因为样式设置直接使用静态字段。

✘ 请勿使用依赖属性来存储安全数据。即使是专用依赖属性，也可以公开地进行访问。

附加依赖属性设计

上一部分中所述的依赖属性表示声明类型的固有属性；例如，`Text` 属性是声明它的 `TextButton` 的属性。附加依赖属性是一种特殊的依赖属性。

一个附加属性的典型示例是 Grid.Column 属性。该属性表示 Button 的(而不是 Grid 的)列位置，但它只有在 Button 包含在 Grid 中时才相关，因此它通过 Grid 附加到 Button。

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Button Grid.Column="0">Click</Button>
    <Button Grid.Column="1">Clack</Button>
</Grid>
```

附加属性的定义与常规依赖属性的定义大致相同，不同之处在于访问器由静态 Get 和 Set 方法表示：

```
public class Grid {

    public static int GetColumn(DependencyObject obj) {
        return (int)obj.GetValue(ColumnProperty);
    }

    public static void SetColumn(DependencyObject obj, int value) {
        obj.SetValue(ColumnProperty,value);
    }

    public static readonly DependencyProperty ColumnProperty =
        DependencyProperty.RegisterAttached(
            "Column",
            typeof(int),
            typeof(Grid)
    );
}
```

依赖属性验证

属性通常所实现的内容就是所谓的验证。尝试更改属性的值时，将执行验证逻辑。

遗憾的是，依赖属性访问器不能包含任意的验证代码。而需要在属性注册期间指定依赖属性验证逻辑。

✕ 请勿在属性的访问器中放置依赖属性验证逻辑。而是将一个验证回调传递到 `DependencyProperty.Register` 方法。

依赖属性更改通知

✕ 请勿在依赖属性访问器中实现更改通知逻辑。依赖属性具有内置的更改通知功能，必须通过向 PropertyMetadata 提供更改通知回调来使用该功能。

依赖属性值强制转换

在实际修改属性存储之前，若提供给属性 setter 的值被该 setter 修改，就会发生属性强制转换。

✕ 请勿在依赖属性访问器中实现强制转换逻辑。

依赖属性具有内置的强制转换功能，可通过向 `PropertyMetadata` 提供强制转换回调来使用该功能。

请参阅

- 框架设计指南
- 常用设计模型

有关从 .NET Framework 移植到 .NET 的概述

2021/11/16 •

本文概述了在将代码从 .NET Framework 移植到 .NET（旧称为 .NET Core）时应考虑的事项。对于许多项目，从 .NET Framework 移植到 .NET 是相对简单的。项目的复杂性决定了在项目文件的初始迁移之后要做多少工作。

应用模型在 .NET 中可用的项目（如库、控制台应用和桌面应用）通常不需要太大的更改。需要使用新应用模型的项目（如从 ASP.NET 迁移到 ASP.NET Core）需要的工作要多一点。旧应用模型中的很多模式都有可以在转换过程中使用的等效项。

不可用的技术

.NET Framework 中有一些技术在 .NET 中是不存在：

- 应用程序域

  不支持创建额外应用程序域。对于代码隔离，将流程或容器用作备用。

- 远程处理

  远程处理用于跨不再受支持的应用程序域进行通信。对于简单的跨进程通信，可将进程间通信 (IPC) 机制视为远程处理的备用方案，如 System.IO.Pipes 类或 MemoryMappedFile 类。对于更复杂的方案，请考虑 StreamJsonRpc 或 ASP.NET Core 等框架（使用 gRPC 或 RESTful Web API 服务）。

- 代码访问安全性 (CAS)

  CAS 是受 .NET Framework 支持、但在 .NET Framework 4.0 中已停用的沙盒技术。它已被 Security Transparency 取代，并且在 .NET 中不受支持。请改用操作系统提供的安全边界，如虚拟化、容器或用户帐户。

- 安全透明度

  与 CAS 类似，这种沙盒技术不再被推荐用于 .NET Framework 应用程序，而且在 .NET 中也不受支持。请改用操作系统提供的安全边界，如虚拟化、容器或用户帐户。

- System.EnterpriseServices

  .NET 不支持 System.EnterpriseServices (COM+)。

- Windows Workflow Foundation (WF) 和 Windows Communication Foundation (WCF)

  .NET 5 及更高版本（包括 .NET Core）不支持 WF 和 WCF。有关替代方法，请参阅 CoreWF 和 CoreWCF。

若要详细了解这些不受支持的技术，请参阅 .NET Framework 技术在 .NET Core 和 .NET 5 及更高版本上不可用。

Windows 桌面技术

许多为 .NET Framework 创建的应用程序都使用桌面技术，如 Windows 窗体或 Windows Presentation Foundation (WPF)。虽然 Windows 窗体和 WPF 均已移植到 .NET 中，但这些仍是仅适用于 Windows 的技术。

在迁移 Windows 窗体或 WPF 应用程序之前，请先考虑以下依赖项：

1. 适用于 .NET 的项目文件使用与 .NET Framework 不同的格式。
2. 你的项目可能会使用在 .NET 中不可用的 API。
3. 第三方控件和库可能还没有移植到 .NET 中，仍只对 .NET Framework 可用。

4. 你的项目使用在 .NET 中不再可用的技术。

.NET 使用 Windows 窗体和 WPF 的开放源代码版本，并对 .NET Framework 进行了增强。

有关将桌面应用程序迁移到 .NET 5 的教程，请参阅以下文章之一：

- 将 .NET Framework WPF 应用迁移到 .NET
- 将 .NET Framework Windows 窗体应用迁移到 .NET

特定于 Windows 的 API

应用程序仍可以在 .NET 支持的平台上对本机库进行平台调用。这项技术并不仅限于 Windows。但是，如果你引用的库是特定于 Windows 的(如 user32.dll 或 kernel32.dll)，那么代码只能在 Windows 上正常运行。对于想要在其上运行应用的每个平台，你都必须查找特定于平台的版本，或者让你的代码足够通用以在所有平台上运行。

当将应用程序从 .NET Framework 移植到 .NET 时，应用程序可能使用了随 .NET Framework 一起分发的库。许多在 .NET Framework 中可用的 API 都没有移植到 .NET 中，因为它们依赖特定于 Windows 的技术，如 Windows Registry 或 GDI+ 绘图模型。

Windows Compatibility Pack 为 .NET 提供了大部分的 .NET Framework API 面，并通过 Microsoft.Windows.Compatibility NuGet 包提供。

有关详细信息，请参阅使用 Windows Compatibility Pack 将代码移植到 .NET 中。

.NET Framework 兼容性模式

.NET Framework 兼容性模式是在 .NET Standard 2.0 中引入的。使用此兼容性模式，.NET Standard 和 .NET 5 及更高版本(以及 .NET Core 3.1)项目可以在仅适用于 Windows 的情况下引用 .NET Framework 库。引用 .NET Framework 库不适用于所有项目(如库使用 Windows Presentation Foundation (WPF) API 时)，但它的开启了很多移植方案。有关详细信息，请参阅分析依赖项以将代码从 .NET Framework 移植到 .NET 中。

跨平台

.NET(旧称为 .NET Core)是为跨平台而设计的。如果代码不依赖特定于 Windows 的技术，那么它可以在 macOS、Linux 和 Android 等其他平台上运行。这包括如下项目类型：

- 库
- 基于控制台的工具
- 自动化
- ASP.NET 站点

.NET Framework 是仅适用于 Windows 的组件。当代码使用特定于 Windows 的技术或 API(如 Windows 窗体和 Windows Presentation Foundation (WPF))时，代码仍可以在 .NET 上运行，但不能在其他操作系统上运行。

库或基于控制台的应用程序不需要太多更改就可以跨平台使用。当移植到 .NET 时，可能需要考虑这一点，并在其他平台上测试应用程序。

.NET Standard 的未来

.NET Standard 是针对多个 .NET 实现推出的一套正式的 .NET API 规范。推出 .NET Standard 的背后动机是要提高 .NET 生态系统中的一致性。自 .NET 5 起，采用了一种不同的方法来建立一致性；使用这种新方法，在很多情况下，都不需要使用 .NET Standard。有关详细信息，请参阅 .NET 5 和 .NET Standard。

.NET Standard 2.0 是支持 .NET Framework 的最后一个版本。

移植辅助工具

可以使用不同的工具来帮助自动执行迁移的某些方面，而不是将应用程序从 .NET Framework 手动移植到 .NET 中。移植复杂的项目本身就是一个复杂的过程。这些工具可能在此过程中有所帮助。

即使你使用工具来帮助移植应用程序，也应查阅本文中的"移植时的注意事项"部分。

.NET 升级助手

.NET 升级助手是一款可以在不同类型的 .NET Framework 应用上运行的命令行工具。它旨在帮助将 .NET Framework 应用升级到 .NET 5。在运行此工具后，大多数情况下，应用将需要更多操作才能完成迁移。此工具会安装可以帮助完成迁移的分析器。此工具适用于以下类型的 .NET Framework 应用程序：

- Windows 窗体
- WPF
- ASP.NET MVC
- 控制台
- 类库

此工具使用本文中列出的其他工具，并指导迁移过程。若要详细了解此工具，请参阅 .NET 升级助手概述。

try-convert

try-convert 工具是一款 .NET 全局工具，可用于将项目或整个解决方案转换为 .NET SDK，包括将桌面应用迁移到 .NET 5。但是，如果你的项目有复杂的生成进程（如自定义任务、目标或导入），则不建议使用此工具。

有关详细信息，请参阅 try-convert GitHub 存储库。

.NET 可移植性分析器

.NET 可移植性分析器是一种工具，可分析程序集并为应用程序或库提供有关缺失的 .NET API 的详细报告，以便在指定的目标 .NET 平台上实现可移植性。

若要使用 Visual Studio 中的 .NET 可移植性分析器，请从市场中安装此扩展。

有关详细信息，请参阅 .NET 可移植性分析器。

平台兼容性分析器

平台兼容性分析器分析你是否在使用将会在运行时抛出 PlatformNotSupportedException 的 API。尽管这并不常见，但如果从 .NET Framework 4.7.2 或更高版本进行移动，最好进行检查。若要详细了解会在 .NET 上抛出异常的 API，请参阅始终在 .NET Core 上抛出异常的 API。

有关详细信息，请参阅平台兼容性分析器。

移植时的注意事项

将应用程序移植到 .NET 时，请按顺序考虑以下建议。

✔ 考虑使用 .NET 升级助手来迁移项目。尽管此工具处于预览阶段，但它自动执行本文中详细介绍的大部分手动步骤，并为你继续迁移路径提供了一个很好的起点。

✔ 考虑先检查依赖项。依赖项必须定目标到 .NET 5、.NET Standard 或 .NET Core。

✔ 务必从 NuGet packages.config 文件迁移到项目文件中的 PackageReference 设置。使用 Visual Studio 转换 package.config 文件。

✔ 考虑升级到最新的项目文件格式，即使你还不能移植应用，也不例外。.NET Framework 项目使用过时的项目格式。尽管最新的项目格式（称为"SDK 样式项目"）是为 .NET Core 及更高版本创建的，它们也适用于 .NET Framework。拥有最新格式的项目文件可以为将来移植应用打下良好的基础。

✔ 务必将 .NET Framework 项目重新定目标到 .NET Framework 4.7.2 及更高版本。在 .NET Standard 不支持现有 API 情况下，这可确保最新备用 API 的可用性。

✔ 考虑定目标到 .NET 5（而不是 .NET Core 3.1）。虽然 .NET Core 3.1 是长期支持 (LTS) 版本，但 .NET 5 是最新的，并且 .NET 6 也将在发布后成为 LTS。

✔ 务必为 Windows 窗体和 WPF 项目定目标到 .NET 5。.NET 5 包含许多对桌面应用的改进。

✔ 若要迁移也可以用于 .NET Framework 项目的库，请考虑定目标到 .NET Standard 2.0。也可以为库设定多个目标，同时定目标到 .NET Framework 和 .NET Standard。

✔ 如果迁移之后出现缺少 API 的错误，请务必添加对 Microsoft.Windows.Compatibility NuGet 包的引用。大部分 .NET Framework API 面是通过 NuGet 包提供给 .NET 的。

另请参阅

- .NET 升级助手概述
- ASP.NET 到 ASP.NET Core 迁移
- 将 .NET Framework WPF 应用迁移到 .NET
- 将 .NET Framework Windows 窗体应用迁移到 .NET
- 适用于服务器应用的 .NET 5 与 .NET Framework

.NET、MSBuild 和 Visual Studio 版本控制概述

2021/11/16 •

了解 .NET SDK 的版本控制以及它与 Visual Studio 和 MSBuild 的关系可能会造成混淆。MSBuild 版本虽与 Visual Studio 配合使用，但也包含在 .NET SDK 中。SDK 有最低版本的 MSBuild 和 Visual Studio 可供使用，并且不会在早于该最低版本的 Visual Studio 版本中加载。

版本控制

默认情况下，.NET SDK 的第一部分与其包含、运行和面向的 .NET 版本相匹配。功能带从 1 开始，并在每季度随 Visual Studio 次要版本而增加。补丁版本随每个月的服务更新递增。

例如，版本 5.0.203 随 .NET 5 一起提供，是第二个 Visual Studio 次要版本，因为 5.0.100 是第一次出现，并且是自发布 5.0.200 后的第三个补丁。

生命周期

SDK 的支持时间范围通常与其中包含的 Visual Studio 版本一致。

SDK	MSBUILD/VS		
2.1.5xx	15.9	11 月 18 日	8 月 21 日*
2.1.8xx	16.2 (No VS)	7 月 19 日	8 月 21 日
3.1.1xx	16.4	12 月 19 日	12 月 22 日
3.1.4xx	16.7	8 月 20 日	12 月 22 日
5.0.1xx	16.8	11 月 20 日	3 月 21 日
5.0.2xx	16.9	3 月 21 日	8 月 22 日
5.0.3xx	16.10	5 月 21 日	8 月 21 日
5.0.4xx	16.11	8 月 21 日	2 月 22 日*
6.0.100-rc1	17.0-preview 4	9 月	不可用
6.0.100	17.0**	11 月 21 日	

NOTE

仅在 Visual Studio 17.0+ 中正式支持将 `net6.0` 用作目标。

MSbuild/Visual Studio 的支持时间更长*

* .NET 6 发布时，目标是为了使 .NET SDK 在版本 16.11 中适用于 下级目标*。这意味着不会强制同时更新 SDK 和 Visual Studio 版本。但是，由于 16.11 版中的 6.0 功能和 C#10 功能的限制，你将无法以 .NET 6 作为目标。这种兼容性专门适用于以版本 5.0 和更低版本作为目标。

对于每个主要的 SDK 版本，预计每年至少需要进行一次 MSBuild 和 Visual Studio 新版本的中断性变更。5.0.Nxx SDK 的所有版本都将在版本 16.8 至版本 16.11 的所有 Visual Studio 和 MSBuild 版本上加载，因为在此期间没有进行中断性变更。SDK 功能（补丁）更新中不应有中断性变更。

参考

- .NET 的版本控制方式概述
- .NET 和 .NET Core 官方支持策略
- Microsoft .NET 和 .NET Core
- .NET 下载（Windows、Linux 和 macOS）

适用于服务器应用的 .NET 与 .NET Framework

2021/11/16 •

有两种支持的 .NET 实现可用于生成服务器端应用。

.NET	.NET Core 1.0 - 3.1、.NET 5 及更高版本的 .NET。
.NET Framework	.NET Framework 1.0 - 4.8

这两者共用许多相同的组件，你可在它们之间共享代码。但两者之间存在根本的差异，可根据需要实现的目标进行选择。本文介绍了在何种情况下进行选择。

在以下情况，对服务器应用程序使用 .NET：

- 用户有跨平台需求。
- 你正在以微服务为目标。
- 你正在使用 Docker 容器。
- 需要高性能和可扩展的系统。
- 需按应用程序提供并行的 .NET 版本。

在以下情况，对服务器应用程序使用 .NET Framework：

- 应用当前使用 .NET Framework（建议扩展而不是迁移）。
- 应用使用不可用于 .NET 的第三方库或 NuGet 包。
- 应用使用不可用于 .NET 的 .NET Framework 技术。
- 应用使用不支持 .NET 的平台。

选择 .NET 的情形

以下各部分更详细地说明了前面提到的通过选择 .NET Framework 选择 .NET 的原因。

跨平台需求

如果 Web 或服务应用程序需要在多个平台（例如 Windows、Linux 和 macOS）上运行，请使用 .NET。

.NET 作为开发工作站支持前面提到的操作系统。Visual Studio 提供了适用于 Windows 和 macOS 的集成开发环境 (IDE)。还可使用运行于 macOS、Linux 和 Windows 上的 Visual Studio Code。Visual Studio Code 支持 .NET，包括 IntelliSense 和调试。大多数第三方编辑器（如 Sublime、Emacs 和 VI）都可搭配 .NET 使用。这些第三方编辑器可使用 Omnisharp 获取编辑器 IntelliSense。也可不使用任何代码编辑器，直接使用适用于所有支持平台的 .NET CLI。

微服务体系结构

微服务体系结构允许跨服务边界组合使用技术。通过这种技术组合，可逐步接受 .NET 作为能与其他微服务或服务搭配使用的新微服务。例如，可组合使用微服务或使用 .NET Framework、Java、Ruby 或其他单片技术开发的服务。

可用的基础结构平台有很多。Azure Service Fabric，设计用于大型和复杂微服务系统 。Azure App Service，很适合用于无状态微服务。基于 Docker 的微服务备选方案适合任何一种微服务方法，这部分内容将在容器部分进行说明。所有这些平台都支持 .NET，是托管微服务的理想选择。

有关微服务体系结构的详细信息，请参阅 .NET 微服务 - 适用于容器化 .NET 应用程序的体系结构。

容器

容器通常与微服务体系结构结合使用。还可使用容器将遵循任何体系结构模式的 Web 应用或服务容器化。可在 Windows 容器上使用 .NET Framework，但 .NET 的模块化和轻型性质使之成为容器的更佳选择。在创建和部署容器时，使用 .NET 时容器的映像大小要远小于使用 .NET Framework 时的大小。例如，因为它是跨平台的，所以可将服务器应用部署到 Linux Docker 容器。

Docker 容器可托管在自己的 Linux 或 Windows 基础结构中，或托管在 Azure Kubernetes 服务等云服务中。Azure Kubernetes 服务可管理、协调和缩放云中基于容器的应用程序。

高性能和可扩展的系统

如果系统需要最佳的性能和可伸缩性，.NET 和 ASP.NET Core 是最佳的选择。Windows Server 和 Linux 的高性能服务器运行时使 ASP.NET Core 成为 TechEmpower 基准上性能最佳的 Web 框架。

性能和可伸缩性对微服务体系结构尤为重要，体系结构中可能正在运行数百个微服务。借助 ASP.NET Core，系统运行的服务器/虚拟机 (VM) 数要低得多。减少服务器/VM 后可节省基础结构和托管成本。

按应用程序级别并行安装 .NET 版本

若要安装含不同 .NET 版本上的依赖项的应用程序，建议使用 NET。该实现支持在同一计算机上并行安装不同版本的 .NET 运行时。并行安装允许在同一服务器上使用多项服务，每项服务位于其相应的 .NET 版本上。这还可在应用程序升级和 IT 运营时降低风险、节省成本。

.NET Framework 不支持并行安装。它是一个 Windows 组件，一次只能有一个版本存在于计算机上。.NET Framework 的每个版本均替换之前的版本。如果安装面向 .NET Framework 更高版本的新应用，则可能会中断计算机上运行的现有应用，因为替换了之前的版本。

选择 .NET Framework 的情形

.NET 对新应用程序和应用程序模式特别有用。但是在很多现有方案中依然会自然而然地选择 .NET Framework，并且对于所有服务器应用程序，.NET Framework 不会被 .NET 代替。

现有的 .NET Framework 应用程序

在大多数情况下，不需要将现有应用程序迁移到 .NET。相反，若要扩展现有的应用程序（例如，在 ASP.NET Core 中写入新的 Web 服务），建议使用 .NET。

不可用于 .NET 的第三方库或 NuGet 包

通过 .NET Standard 可跨各种 .NET 实现（包括 .NET Core/5+）共享代码。使用 .NET Standard 2.0，兼容性模式允许 .NET Standard 和 .NET 项目引用 .NET Framework 库。有关详细信息，请参阅对 .NET Framework 库的支持。

仅在以下情况下需要使用 .NET Framework：库或 NuGet 包使用 .NET Standard 或 .NET 中不提供的技术。

.NET Framework 技术不可用于 .NET

某些 .NET Framework 技术在 .NET 中不可用。以下列表显示无法在 .NET 中找到的最常见技术：

- ASP.NET Web 窗体应用程序：ASP.NET Web 窗体仅在.NET Framework 中可用。ASP.NET Core 不能用于 ASP.NET Web 窗体。
- ASP.NET 网页应用程序：ASP.NET 网页未包含在 ASP.NET Core 中。
- WCF 服务的实现。虽然 WCF 客户端库可从 .NET 使用 WCF 服务，WCF 服务器实现目前只在 .NET Framework 上可用。
- 工作流相关的服务：Windows Workflow Foundation (WF)、工作流服务（WCF + 单个服务中的 WF）和 WCF Data Services（以前称为“ADO.NET Data Services”）仅在 .NET Framework 上可用。
- 语言支持：.NET 目前支持 Visual Basic 和 F#，但不是所有项目类型都支持。有关支持的项目模板列表，请

参阅 dotnet new 的模板选项。

有关详细信息，请参阅在 .NET 中不可用的 .NET Framework 技术。

平台不支持 .NET

某些 Microsoft 或第三方平台不支持 .NET。某些 Azure 服务提供尚不可用于 .NET 的 SDK。在这种情况下，可使用等效的 REST API（而不是客户端 SDK）。

请参阅

- 在 ASP.NET 和 ASP.NET Core 之间进行选择
- 面向 .NET Framework 的 ASP.NET Core
- 目标框架
- .NET 简介
- 从 .NET Framework 移植到 .NET 5
- .NET 和 Docker 简介
- .NET 组件概述
- .NET 微服务 - 适用于容器化 .NET 应用程序的体系结构

.NET 升级助手概述

2021/11/16 •

你可能有些应用当前正在 .NET Framework 上运行，而你想将它们移植到 .NET 5。.NET 升级助手工具可帮助完成此过程。本文提供以下内容：

- .NET 升级助手概述。
- 如何安装 .NET 升级助手。

什么是 .NET 升级助手

.NET 升级助手是一款可以在不同类型的 .NET Framework 应用上运行的命令行工具。它旨在帮助将 .NET Framework 应用升级到 .NET 5。在运行此工具后，大多数情况下，应用将需要其他操作才能完成迁移。此工具会安装可以帮助完成迁移的分析器。

该工具目前支持下列 .NET Framework 应用类型：

- .NET Framework Windows 窗体应用
- .NET Framework WPF 应用
- .NET Framework ASP.NET MVC 应用
- .NET Framework 控制台应用
- .NET Framework 类库

.NET 升级助手目前为预发行版，且正在频繁接收更新。如果在使用该工具时发现问题，请在工具的 GitHub 存储库中进行报告。

如何安装 .NET 升级助手

入门教程介绍了如何安装和使用 .NET 升级助手。

先决条件

- 此工具使用 MSBuild 来处理项目文件。请确保已安装最新版本的 MSBuild。要满足此要求，一个简单的方法是安装 Visual Studio 2019。

安装步骤

可运行以下命令将该工具安装为 .NET CLI 工具：

```
dotnet tool install -g upgrade-assistant
```

同样地，由于 .NET 升级助手是作为 .NET CLI 工具安装的，可运行以下命令来轻松更新它：

```
dotnet tool update -g upgrade-assistant
```

有关详细的安装说明，请查看项目的自述文件。

另请参阅

- 将 ASP.NET MVC 应用升级到 .NET 5
- 将 WPF 应用升级到 .NET 5

- 将 Windows 窗体应用升级到 .NET 5
- .NET 升级助手 GitHub 存储库

使用 .NET 升级助手将 WPF 应用升级到 .NET 5

2021/11/16 •

.NET 升级助手是一种命令行工具，可帮助将 .NET Framework WPF 应用升级到 .NET 5。本文提供以下内容：

- 演示如何针对 .NET Framework WPF 应用运行该工具
- 故障排除提示

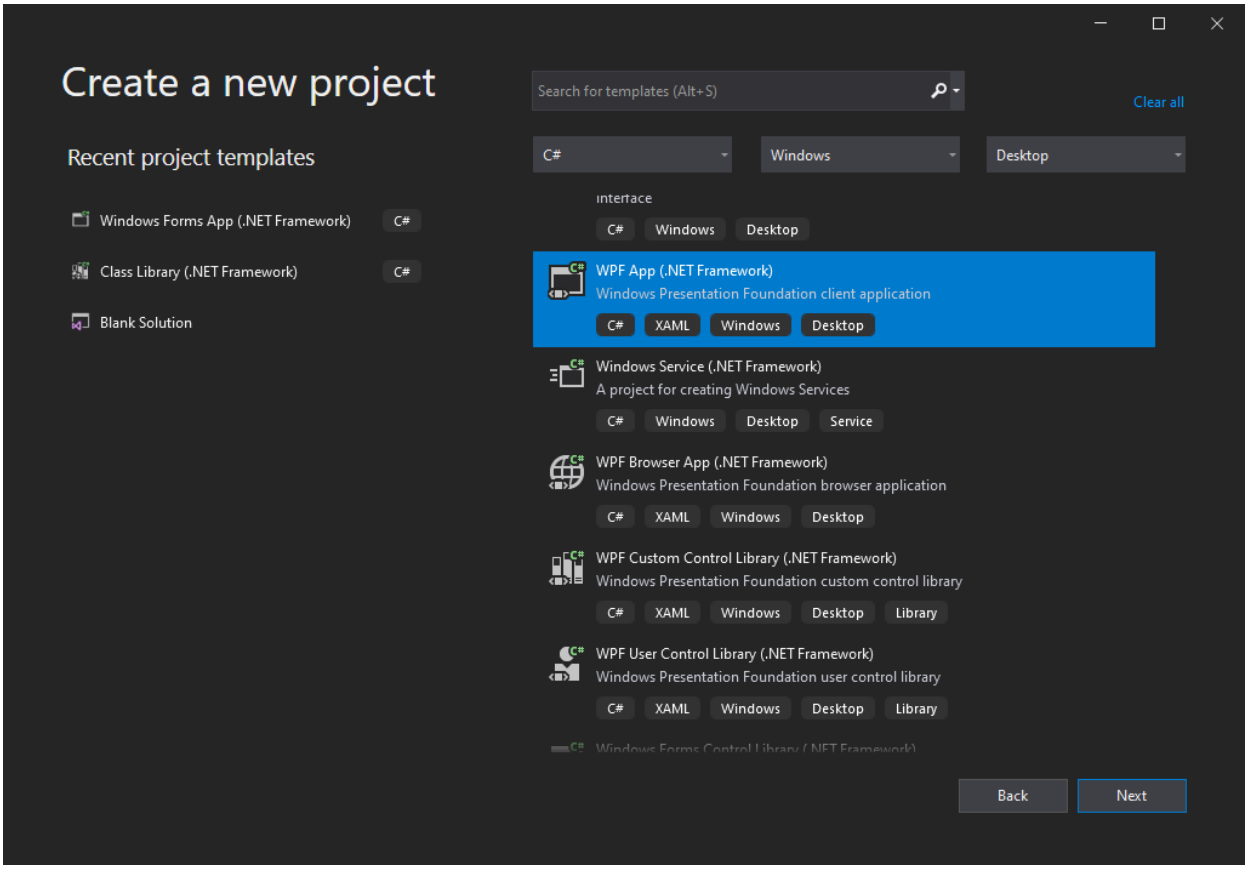
升级 .NET Framework WPF 应用

本部分演示如何针对新创建的面向 .NET Framework 4.6.1 的 WPF 应用运行 NET 升级助手。若要详细了解如何安装此工具，请查看 .NET 升级助手概述。

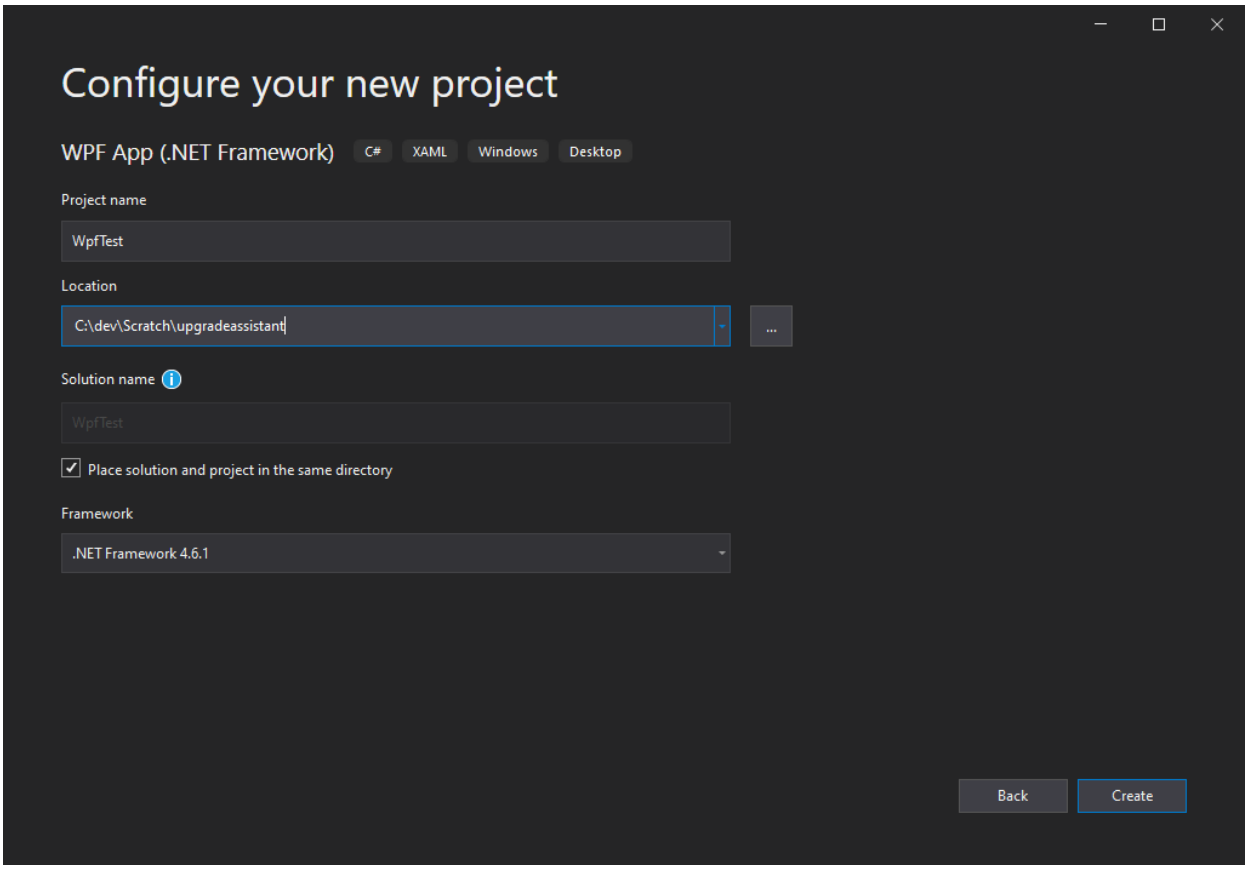
初始演示设置

如果你要针对你自己的 .NET Framework 应用运行 .NET 升级助手，可跳过此步骤。如果你只想试用一下来看看它的工作原理，可在此步骤中了解如何设置示例 .NET WPF 项目以供使用。

借助 Visual Studio，使用 .NET Framework 创建新的 WPF 应用。



将项目命名为“WpfTest”。将项目配置为使用 .NET Framework 4.6.1。



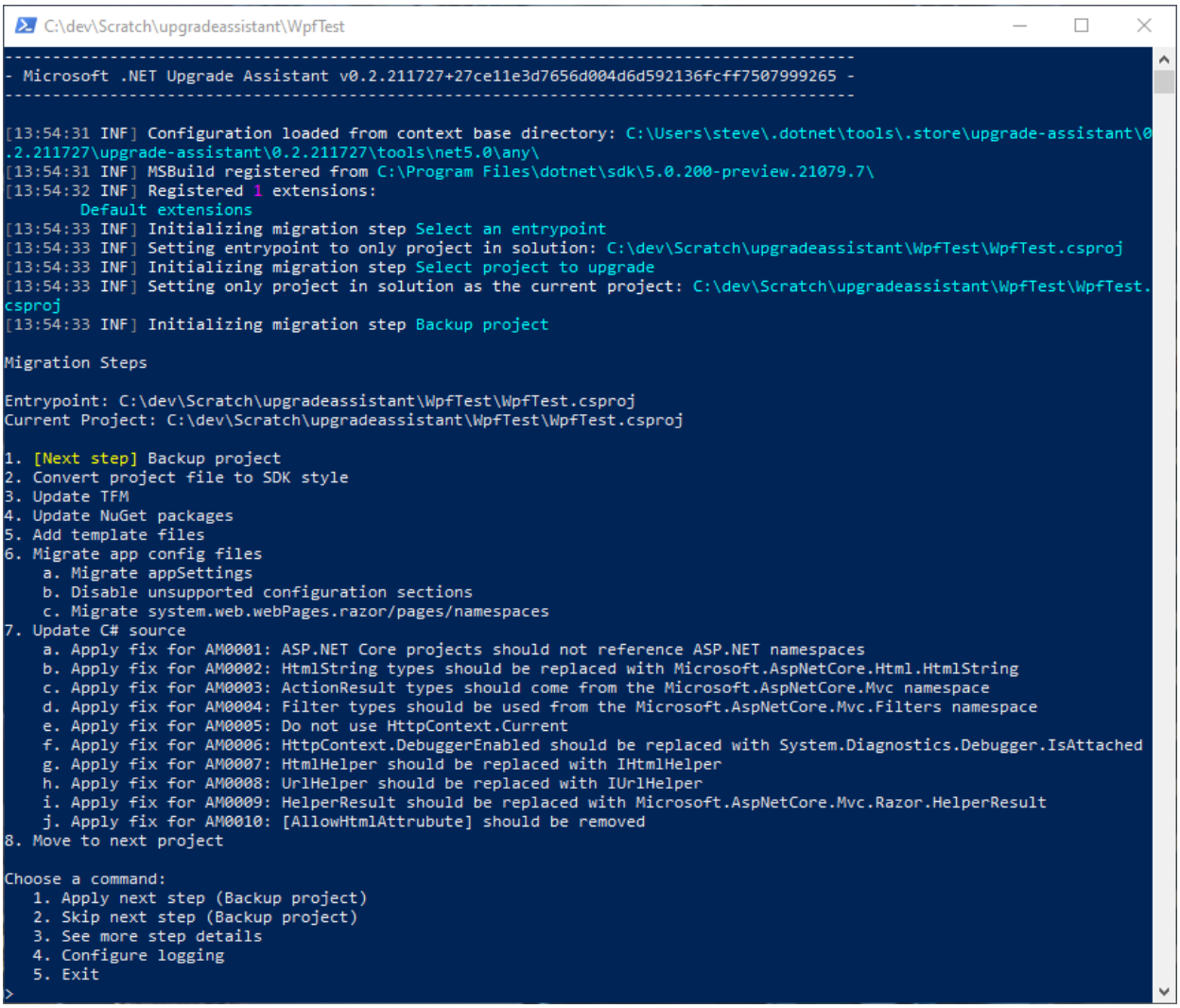
查看所创建的项目及其文件，尤其是它的项目文件。

运行升级助手

打开终端，导航到目标项目或解决方案所在的文件夹。运行 `upgrade-assistant` 命令，传入你要针对的项目的名称（可从任意位置运行该命令，只要项目文件的路径有效就行）。

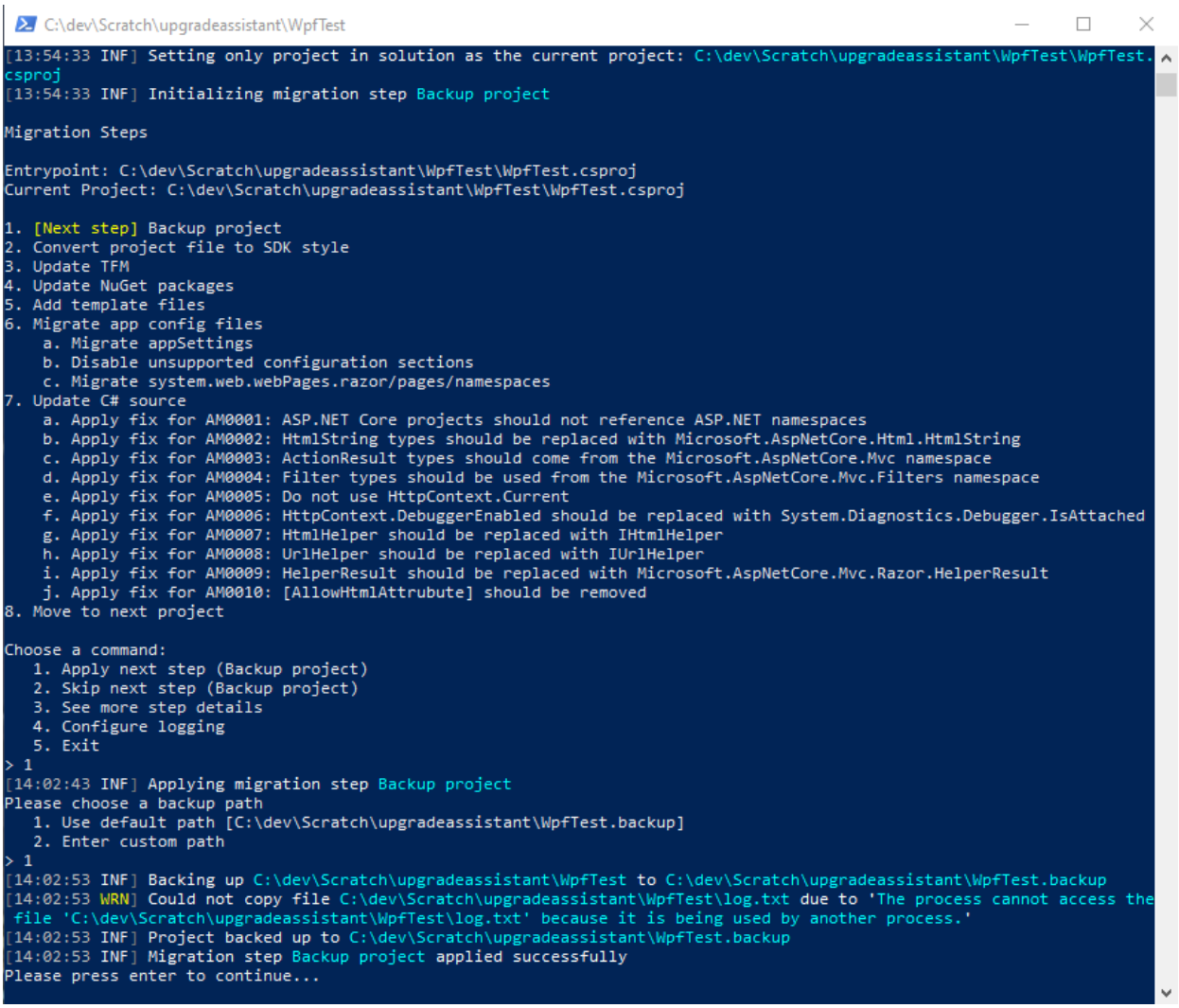
```
upgrade-assistant upgrade .\WpfTest.csproj
```

该工具将运行并显示它将执行的步骤列表。

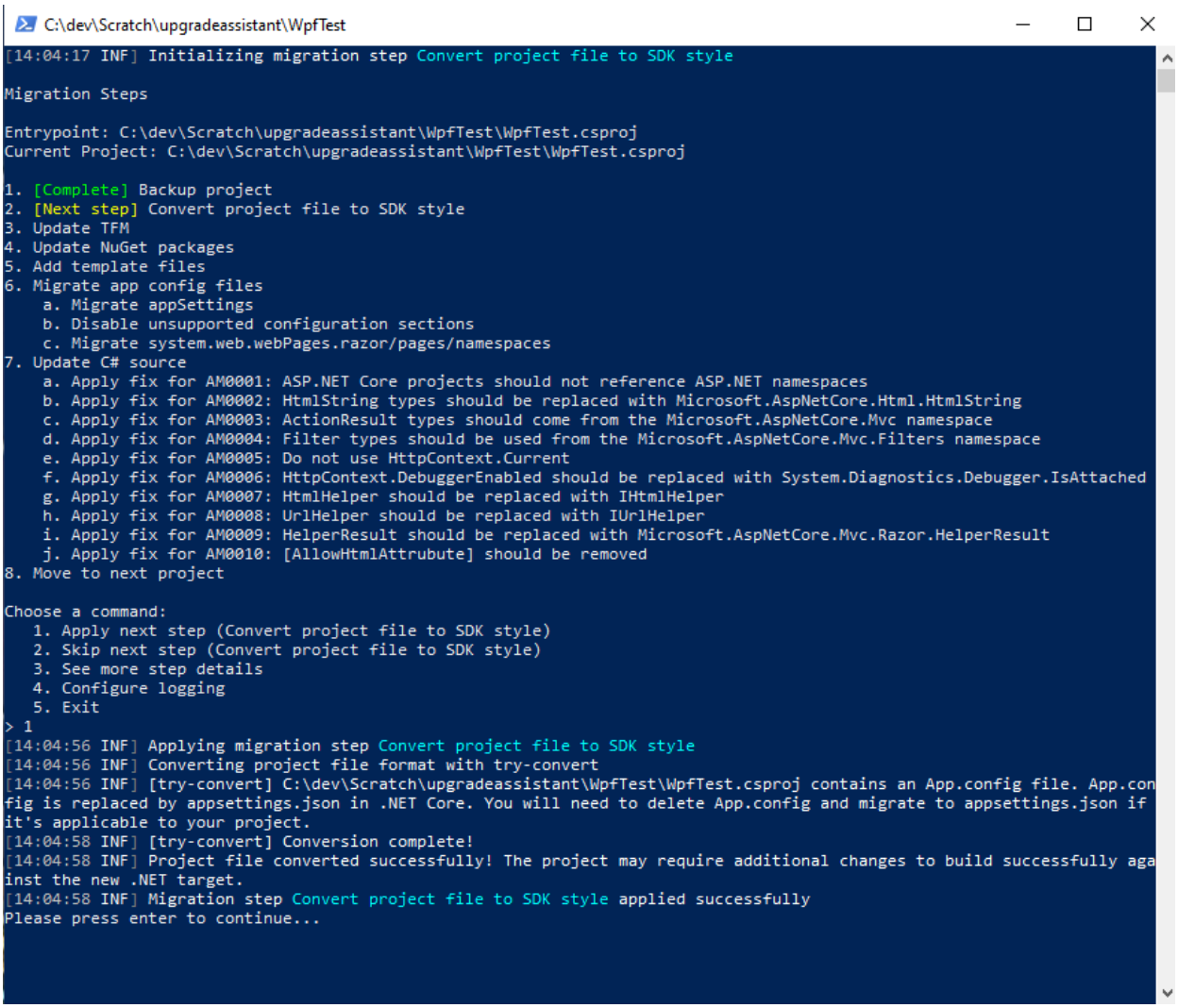


完成每个步骤后，该工具都会提供一组命令，用户可应用这些命令，也可跳过下一步骤、查看更多详细信息、配置日志记录或退出该过程。如果该工具检测到某个步骤将不执行任何操作，它会自动跳过该步骤，转到下一步骤，直到到达有要执行的操作的步骤为止。如果未进行其他任何选择，那么按 Enter 将执行下一步。

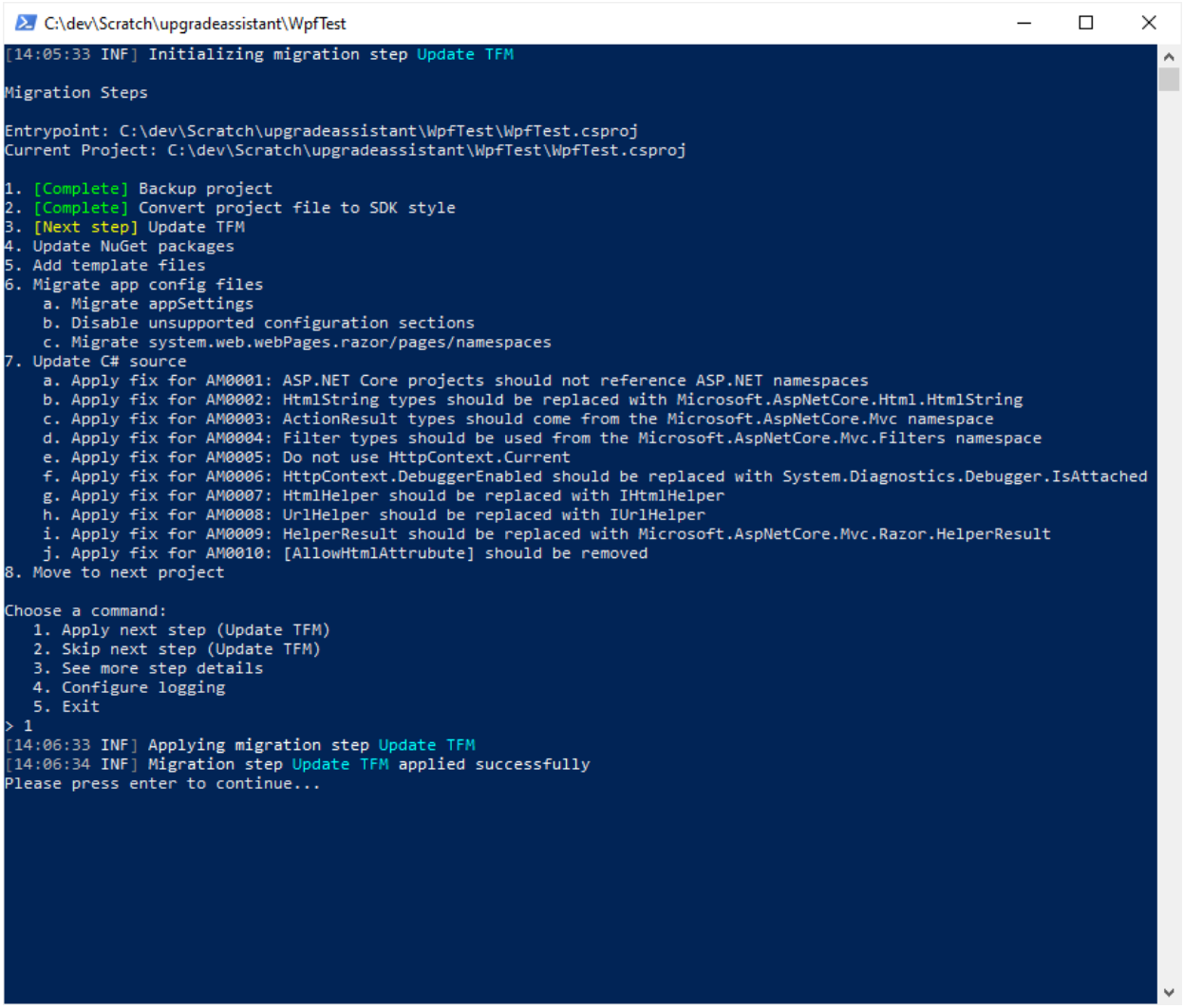
在此示例中，每次都会选择“应用”步骤。第一步是备份项目。



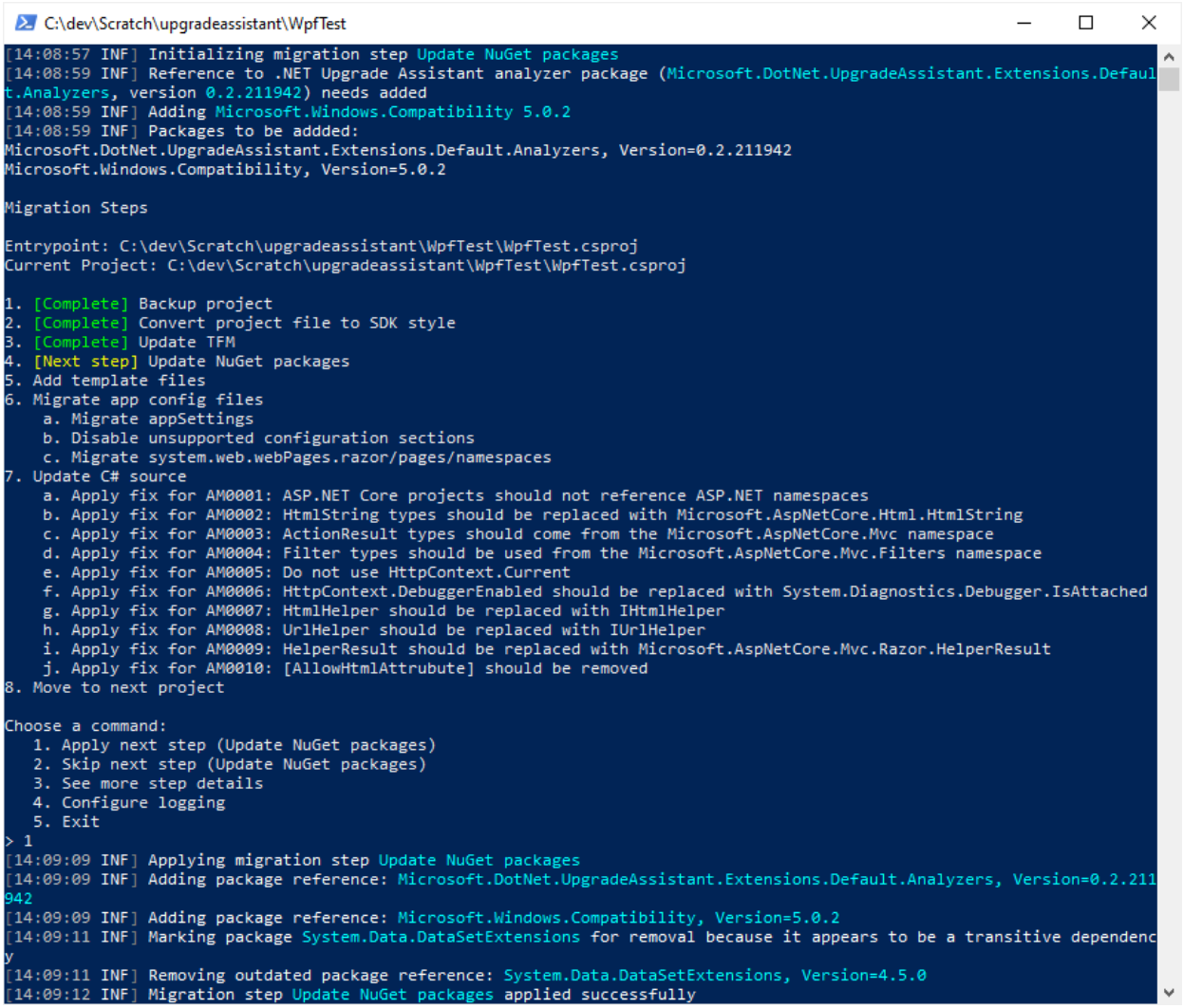
该工具会提示输入自定义路径进行备份或使用默认路径，后者会将项目备份放在具有 `.backup` 扩展名的同一文件夹中。此工具接下来做的是将项目文件转换为 SDK 样式。



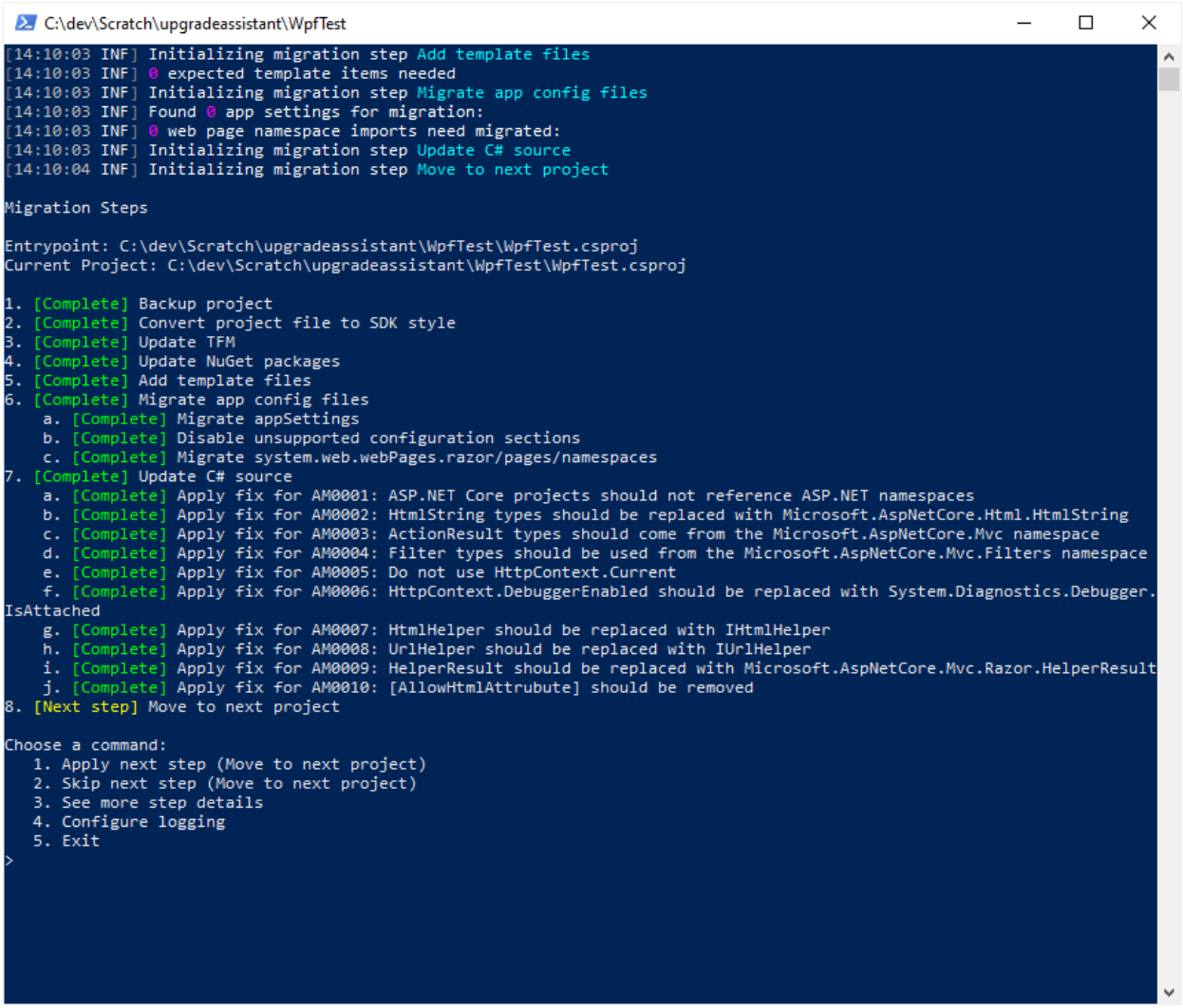
更新项目格式后，下一步是更新项目的 TFM。



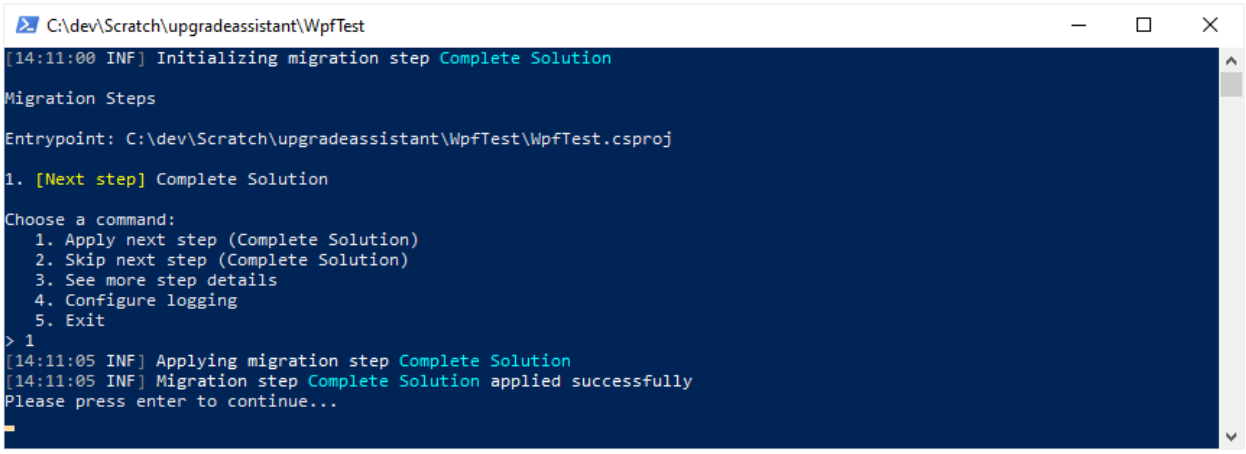
接下来，该工具会更新项目的 NuGet 包。



更新包后，接下来是添加模板文件（如果有）。在本例中，没有需要添加的模板文件。该步骤将继续，迁移应用配置文件并更新 C# 源来应用修补程序，如下所示。此项目无需任何配置文件或源代码更改，因此会自动继续这些步骤。



这是最后一个项目，因此下一步是“移动到新的项目”，它提示完成迁移整个解决方案的过程。



完成此过程后，已迁移的 WPF 项目将如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0-windows</TargetFramework>
    <OutputType>WinExe</OutputType>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <UseWPF>true</UseWPF>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.CSharp" Version="4.7.0" />
    <PackageReference Include="Microsoft.DotNet.UpgradeAssistant.Extensions.Default.Analyzers"
Version="0.2.211942">
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.Windows.Compatibility" Version="5.0.2" />
  </ItemGroup>
</Project>
```

请注意，.NET 升级助手还会向项目添加分析器，它可帮助继续执行升级过程。

故障排除提示

使用 .NET 升级助手时，可能会出现一些已知问题。某些情况下，.NET 升级助手在内部使用的 try-convert 工具会出现问题。

有关更多故障排除提示和已知问题，可查看此工具的 GitHub 存储库。

另请参阅

- 将 Windows 窗体应用升级到 .NET 5
- 将 ASP.NET MVC 应用升级到 .NET 5
- .NET 升级助手概述
- .NET 升级助手 GitHub 存储库

使用 .NET 升级助手将 Windows 窗体应用升级到 .NET 5

2021/11/16 •

.NET 升级助手是一种命令行工具，可帮助将 .NET Framework Windows 窗体 (WinForms) 应用升级到 .NET 5。本文提供以下内容：

- 演示如何针对 .NET Framework Windows 窗体应用运行该工具
- 故障排除提示

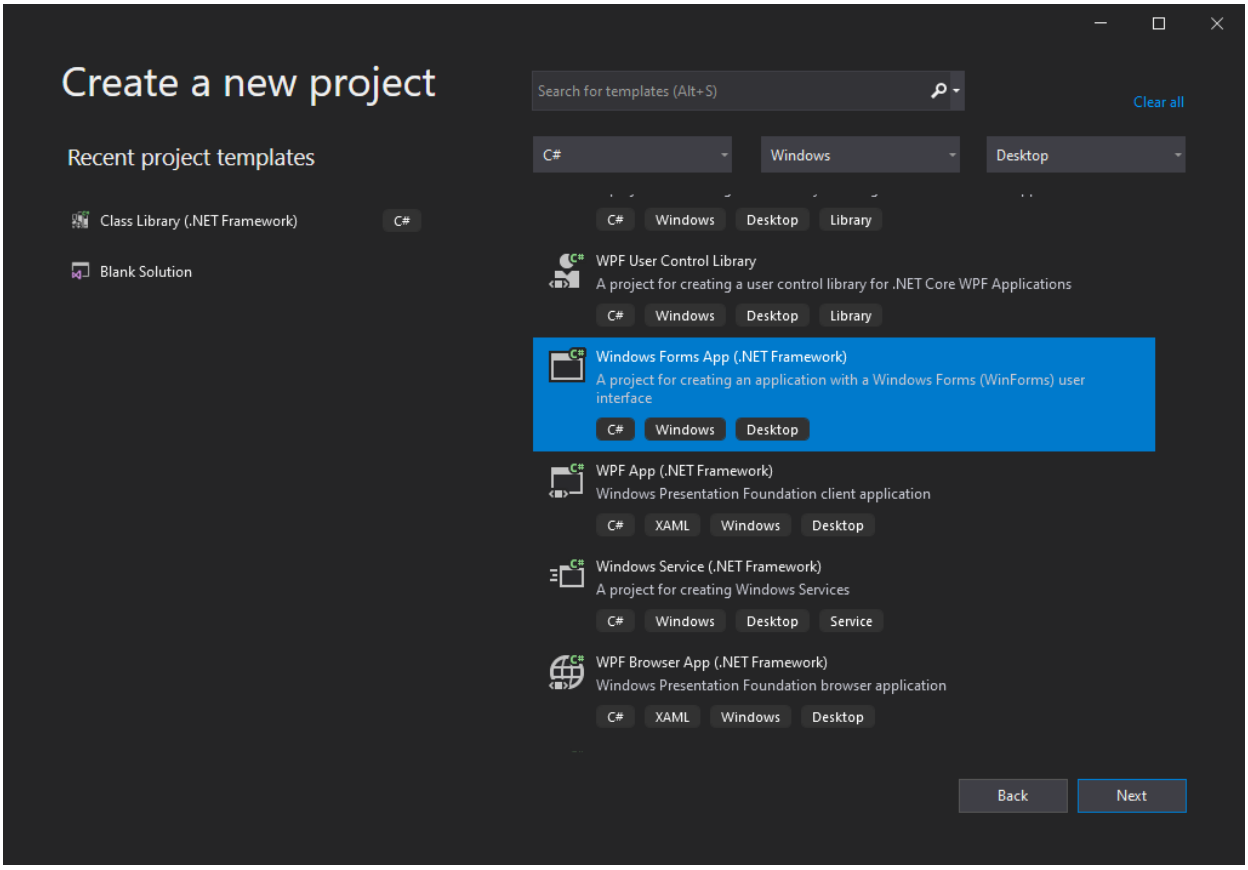
升级 .NET Framework Windows 窗体应用

本部分演示如何针对新创建的面向 .NET Framework 4.6.1 的 Windows 窗体应用运行 NET 升级助手。若要详细了解如何安装此工具，请查看 .NET 升级助手概述。

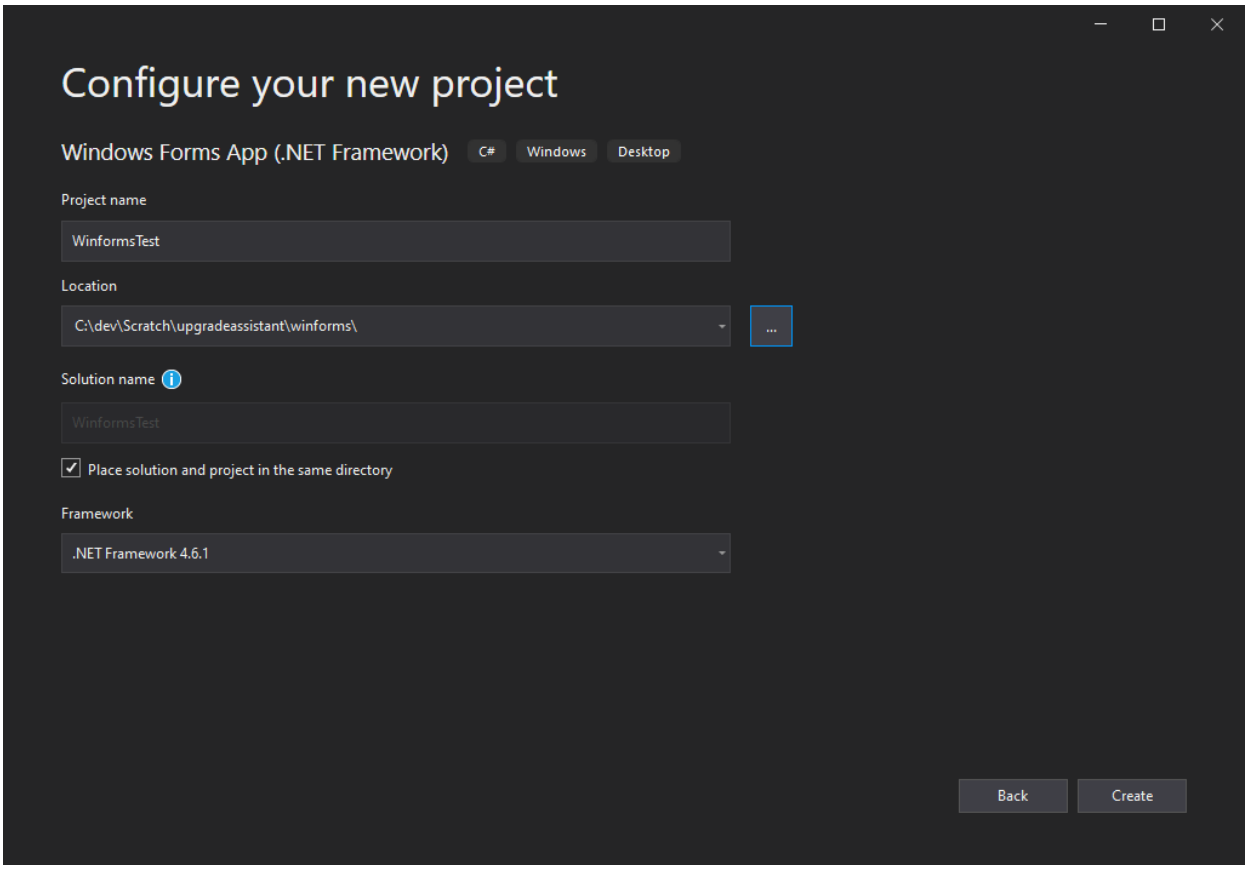
初始演示设置

如果你要针对你自己的 .NET Framework 应用运行 .NET 升级助手，可跳过此步骤。如果你只想试用一下来看看它的工作原理，可在此步骤中了解如何设置示例 .NET Windows 窗体项目以供使用。

借助 Visual Studio，使用 .NET Framework 创建一个新的 Windows 窗体项目。



将项目命名为 WinformsTest。将项目配置为使用 .NET Framework 4.6.1。



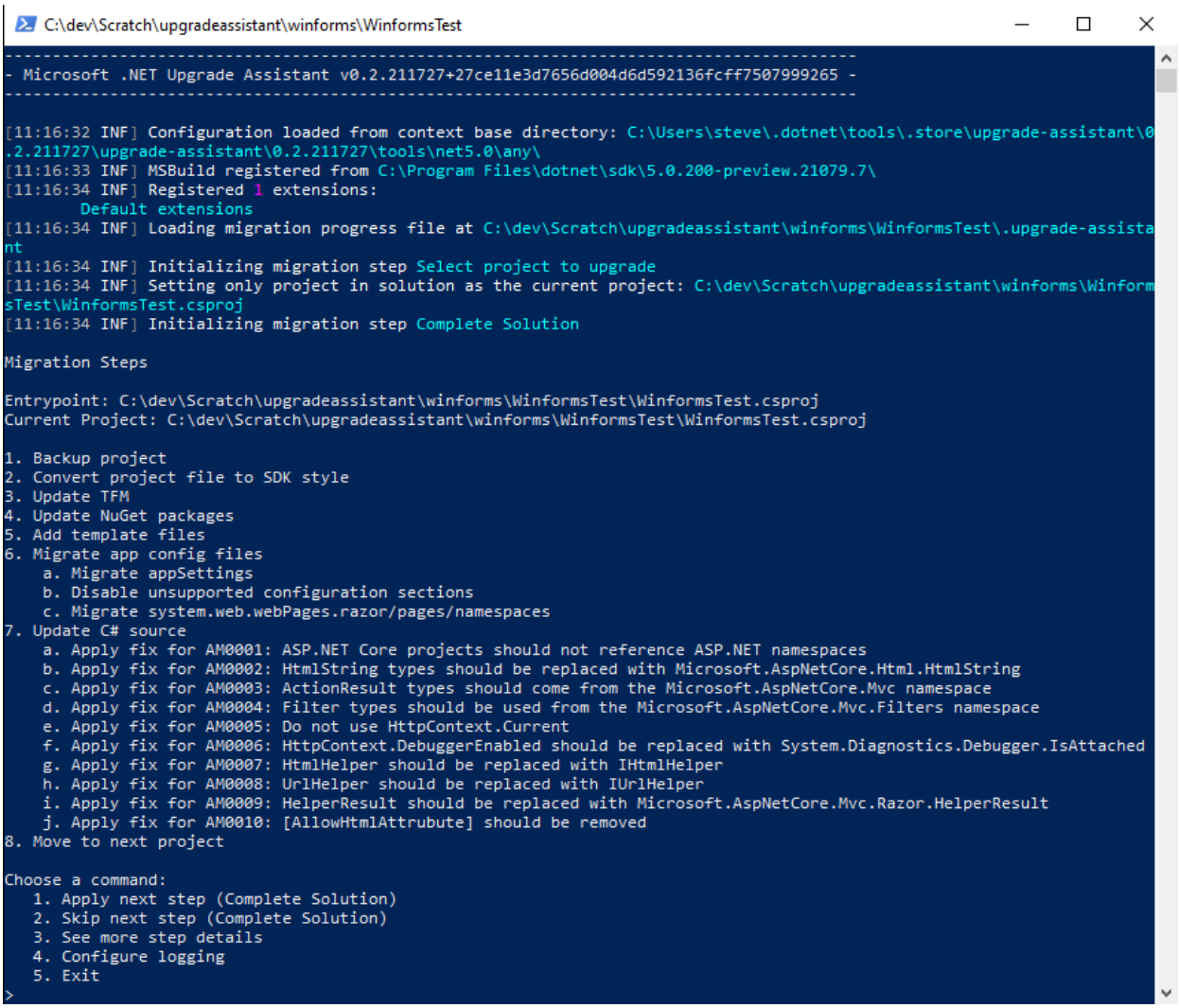
查看所创建的项目及其文件，尤其是它的项目文件。

运行升级助手

打开终端，导航到目标项目或解决方案所在的文件夹。运行 `upgrade-assistant` 命令，传入你要针对的项目的名称(可从任意位置运行该命令，只要项目文件的路径有效就行)。

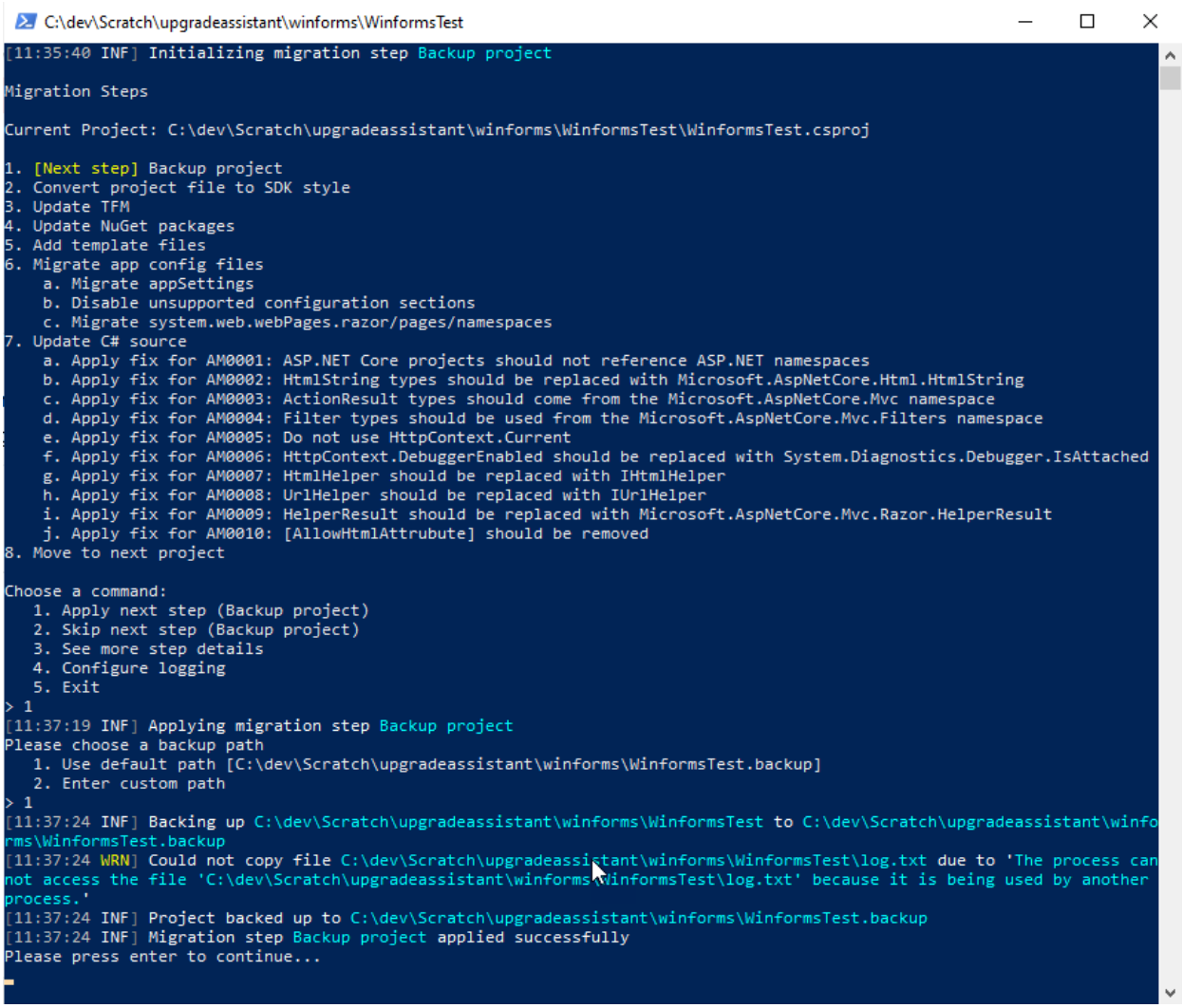
```
upgrade-assistant upgrade .\WinformsTest.csproj
```

该工具将运行并显示它将执行的步骤列表。

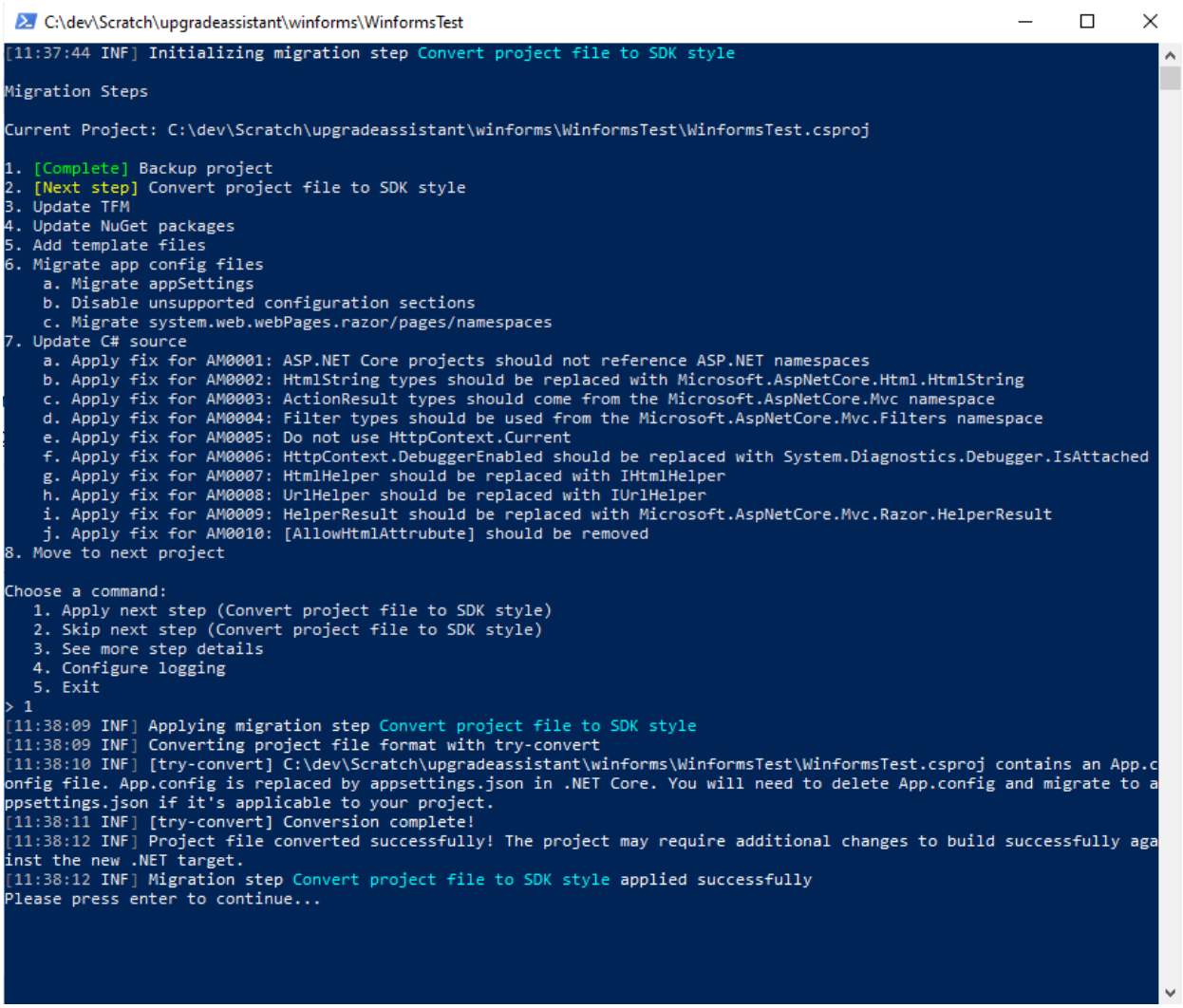


完成每个步骤后，该工具都会提供一组命令，用户可应用这些命令，也可跳过下一步骤、查看更多详细信息、配置日志记录或退出该过程。如果该工具检测到某个步骤将不执行任何操作，它会自动跳过该步骤，转到下一步骤，直到到达有要执行的操作的步骤为止。如果未进行其他任何选择，那么按 Enter 将执行下一步。

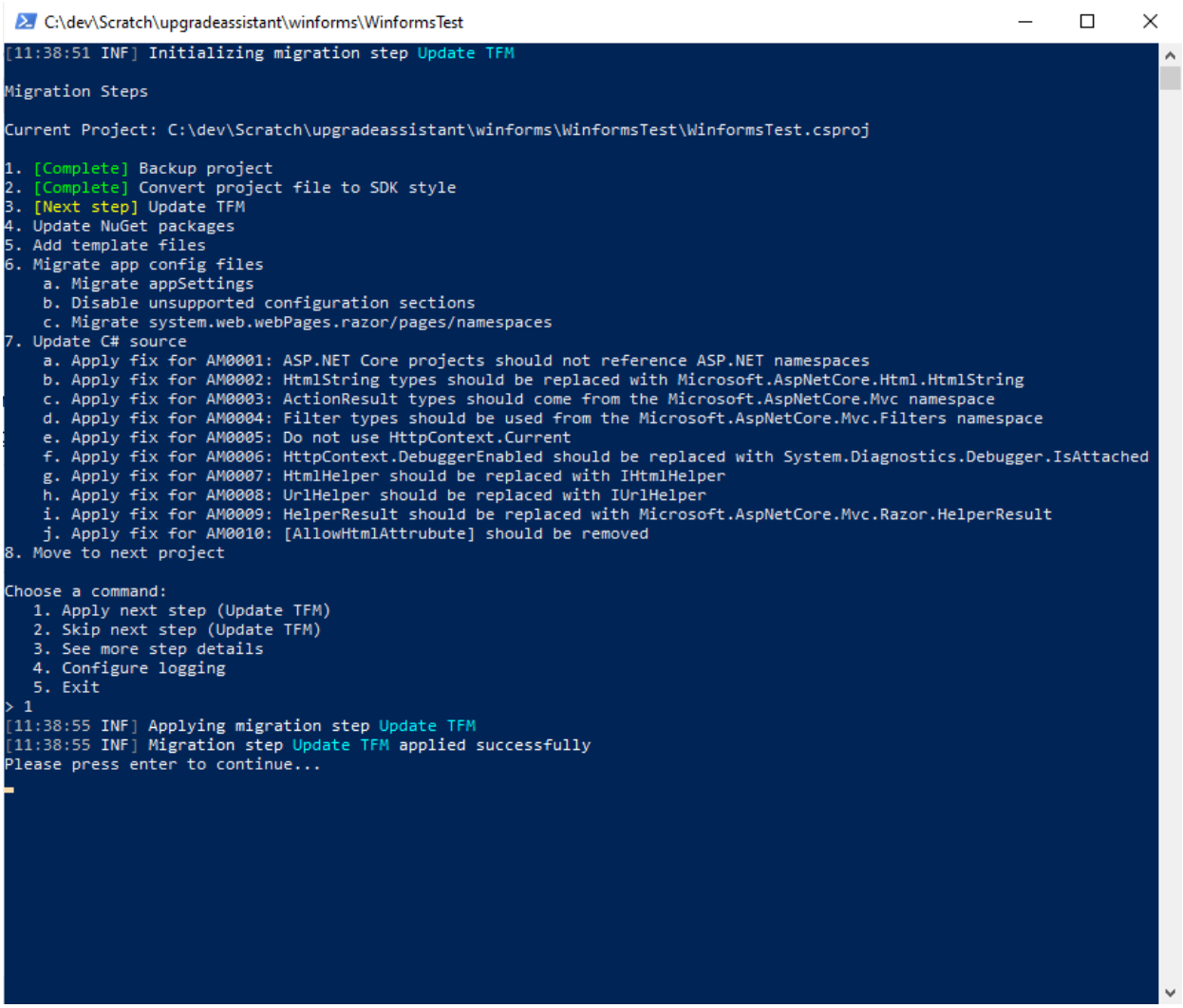
在此示例中，每次都会选择“应用”步骤。第一步是备份项目。



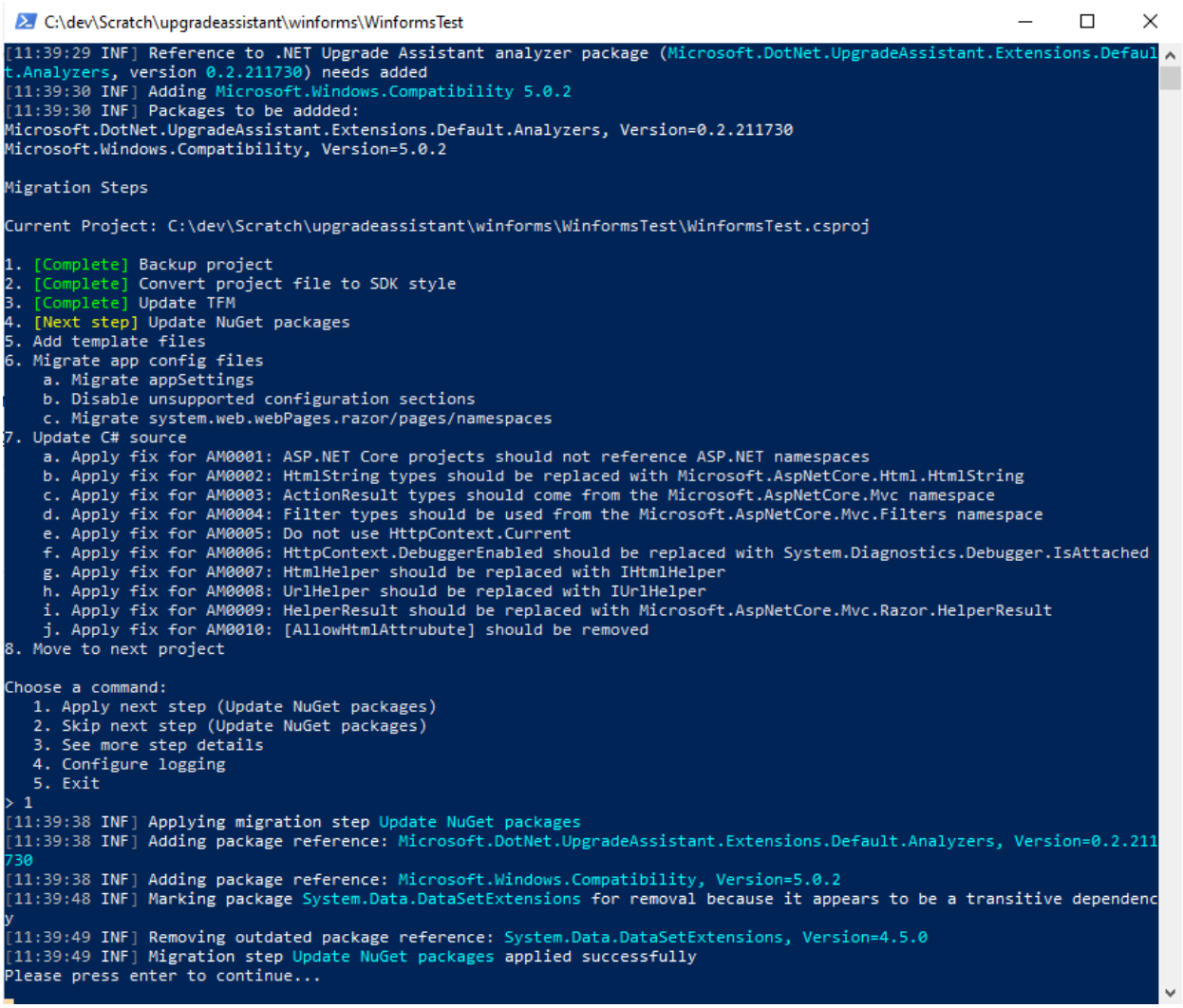
该工具会提示输入自定义路径进行备份或使用默认路径，后者会将项目备份放在具有 `.backup` 扩展名的同一文件夹中。此工具接下来做的是将项目文件转换为 SDK 样式。



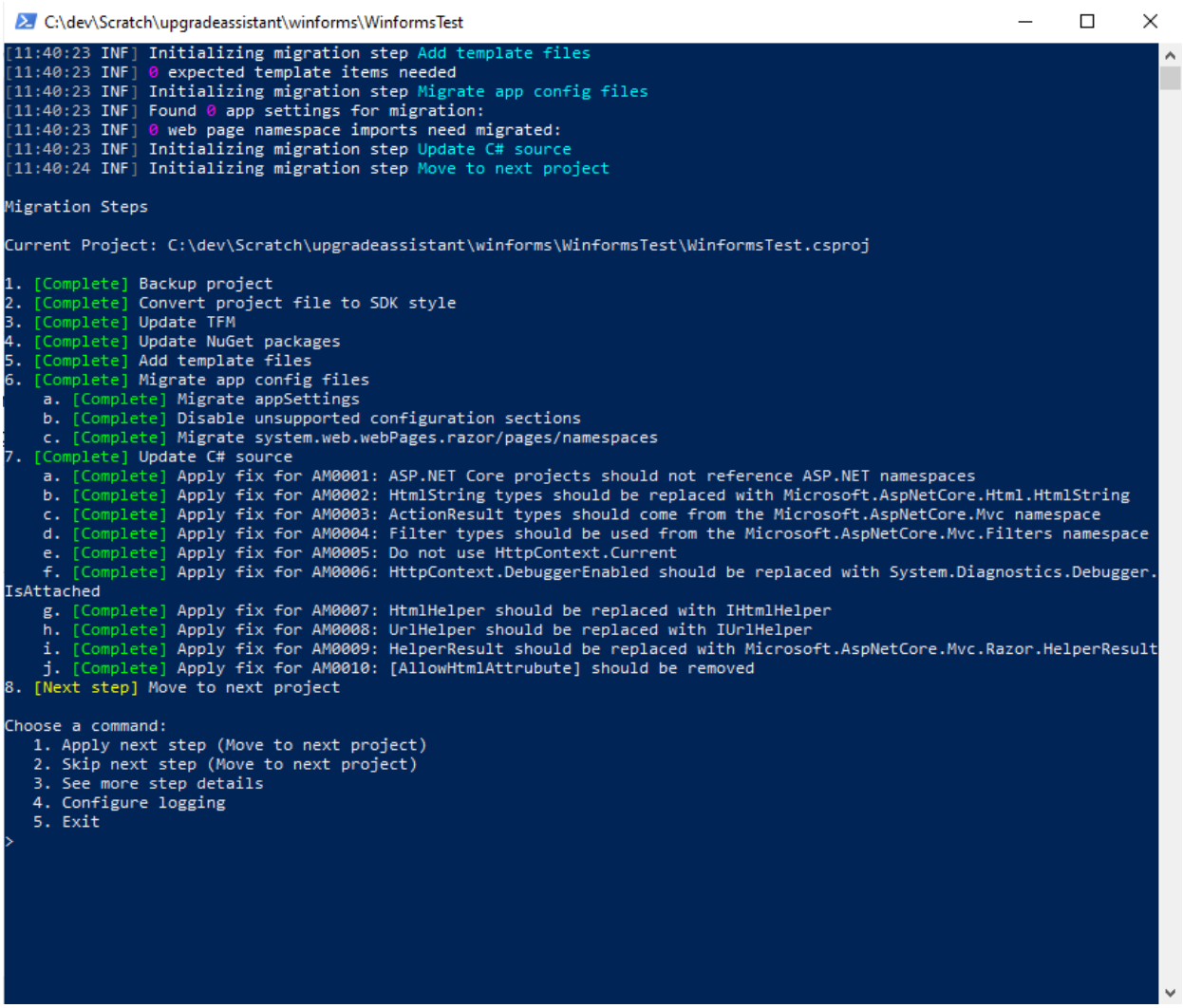
更新项目格式后，下一步是更新项目的 TFM。



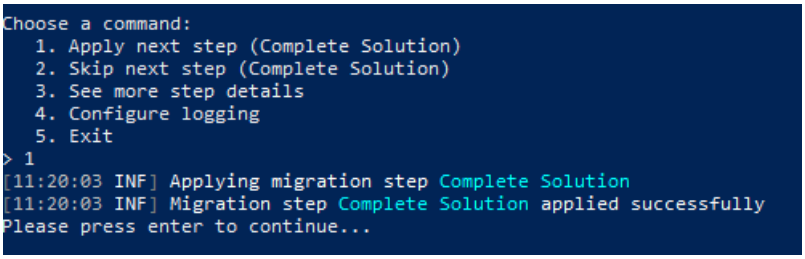
接下来，该工具会更新项目的 NuGet 包。



更新包后，接下来是添加模板文件（如果有）。在本例中，没有需要添加的模板文件。该步骤将继续，迁移应用配置文件并更新 C# 源来应用修补程序，如下所示。此项目无需任何配置文件或源代码更改，因此会自动继续这些步骤。



这是最后一个项目，因此下一步是“移动到新的项目”，它提示完成迁移整个解决方案的过程。



完成此过程后，已迁移的 Windows 窗体项目将如下所示：

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0-windows</TargetFramework>
    <OutputType>WinExe</OutputType>
    <GenerateAssemblyInfo>false</GenerateAssemblyInfo>
    <UseWindowsForms>true</UseWindowsForms>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.CSharp" Version="4.7.0" />
    <PackageReference Include="Microsoft.DotNet.UpgradeAssistant.Extensions.Default.Analyzers"
Version="0.2.211730">
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="Microsoft.Windows.Compatibility" Version="5.0.2" />
  </ItemGroup>
</Project>
```

请注意，.NET 升级助手还会向项目添加分析器，它可帮助继续执行升级过程。

故障排除提示

使用 .NET 升级助手时，可能会出现一些已知问题。某些情况下，.NET 升级助手在内部使用的 try-convert 工具会出现问题。

有关更多故障排除提示和已知问题，可查看此工具的 GitHub 存储库。

另请参阅

- 将 WPF 应用升级到 .NET 5
- 将 ASP.NET MVC 应用升级到 .NET 5
- .NET 升级助手概述
- .NET 升级助手 GitHub 存储库

使用 .NET 升级助手将 ASP.NET MVC 应用升级到 .NET 5

2021/11/16 •

.NET 升级助手是一种命令行工具，可帮助将 .NET Framework ASP.NET MVC 应用升级到 .NET 5。本文提供以下内容：

- 演示如何针对 .NET Framework ASP.NET MVC 应用运行该工具
- 故障排除提示

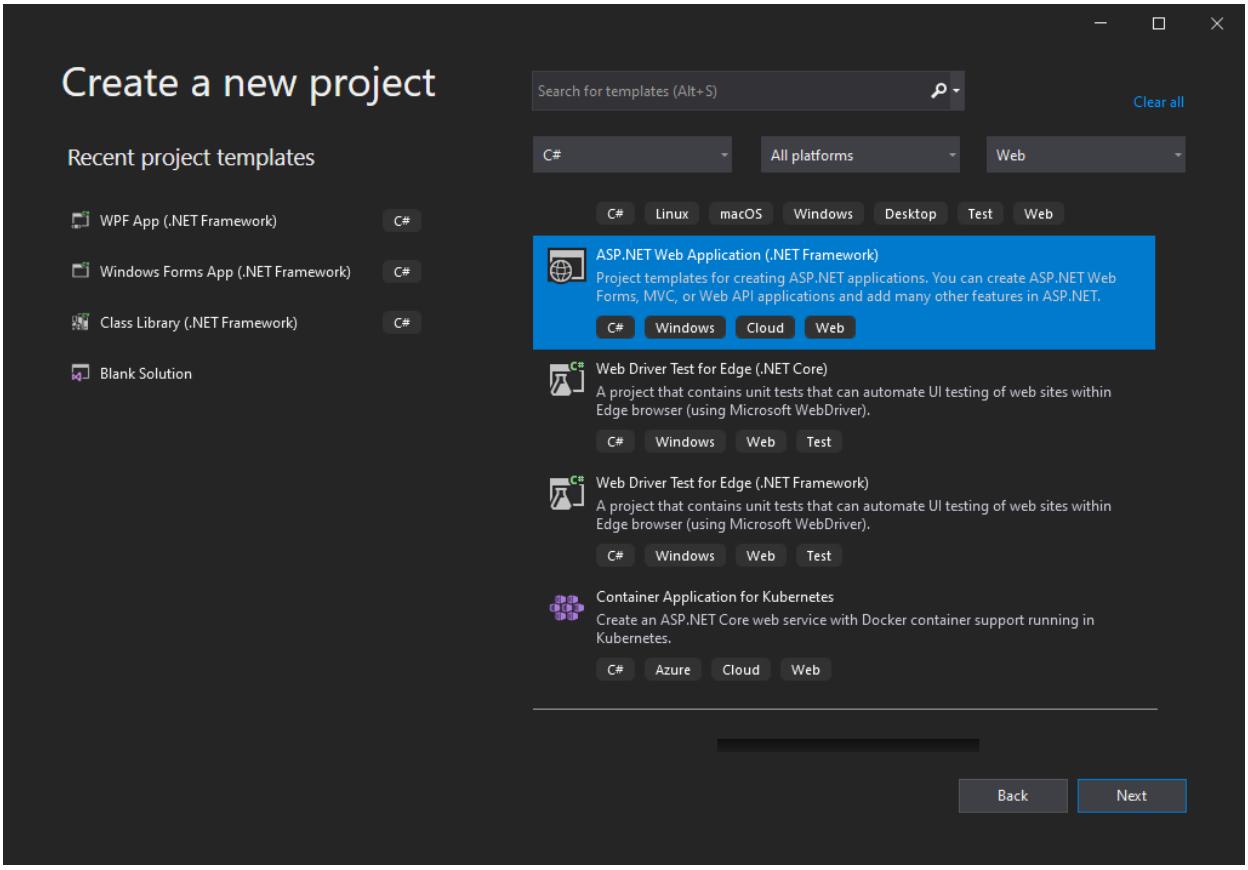
升级 .NET Framework ASP.NET MVC 应用

本部分演示如何针对新创建的面向 .NET Framework 4.6.1 的 ASP.NET MVC 应用运行 NET 升级助手。若要详细了解如何安装此工具，请查看 .NET 升级助手概述。

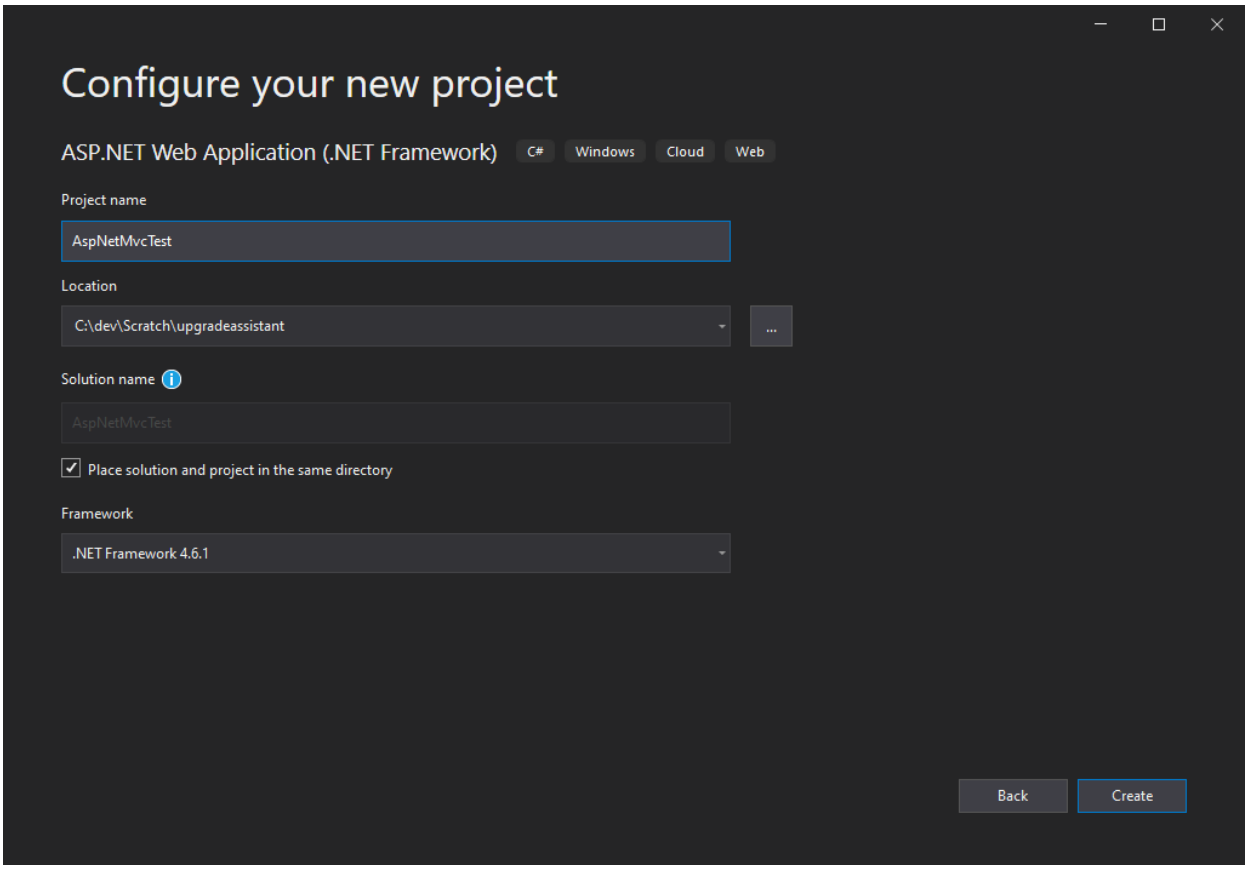
初始演示设置

如果你要针对你自己的 .NET Framework 应用运行 .NET 升级助手，可跳过此步骤。如果你只想试用一下来看看它的工作原理，可在此步骤中了解如何设置示例 ASP.NET MVC 和 Web API (.NET Framework) 项目以供使用。

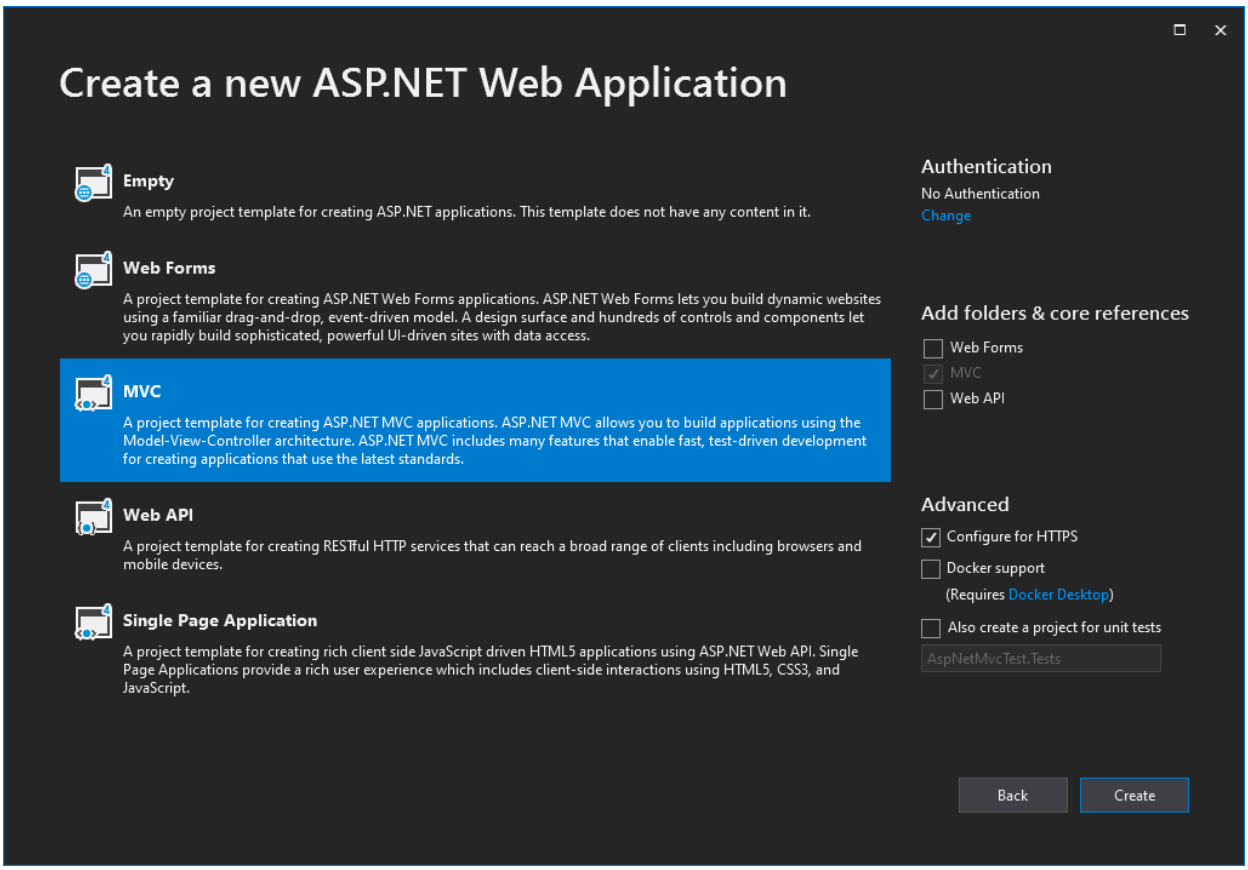
借助 Visual Studio，使用 .NET Framework 创建一个新的 ASP.NET Web 应用程序项目。



将项目命名为 AspNetMvcTest。将项目配置为使用 .NET Framework 4.6.1。



在下一对话框中，选择“MVC”应用程序，然后选择“创建”。



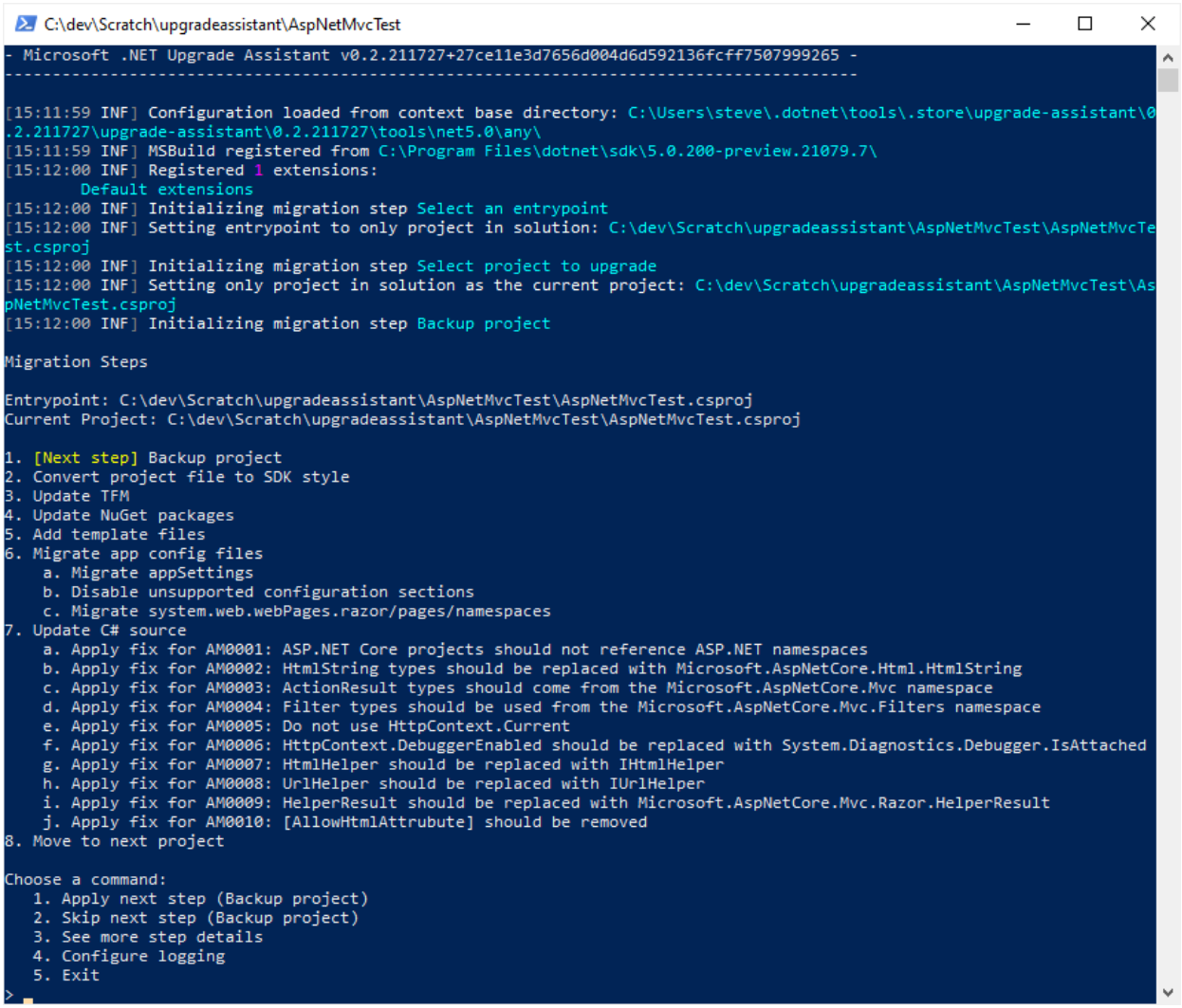
查看所创建的项目及其文件，尤其是它的项目文件。

运行升级助手

打开终端，导航到目标项目或解决方案所在的文件夹。运行 `upgrade-assistant` 命令，传入你要针对的项目的名称（可从任意位置运行该命令，只要项目文件的路径有效就行）。

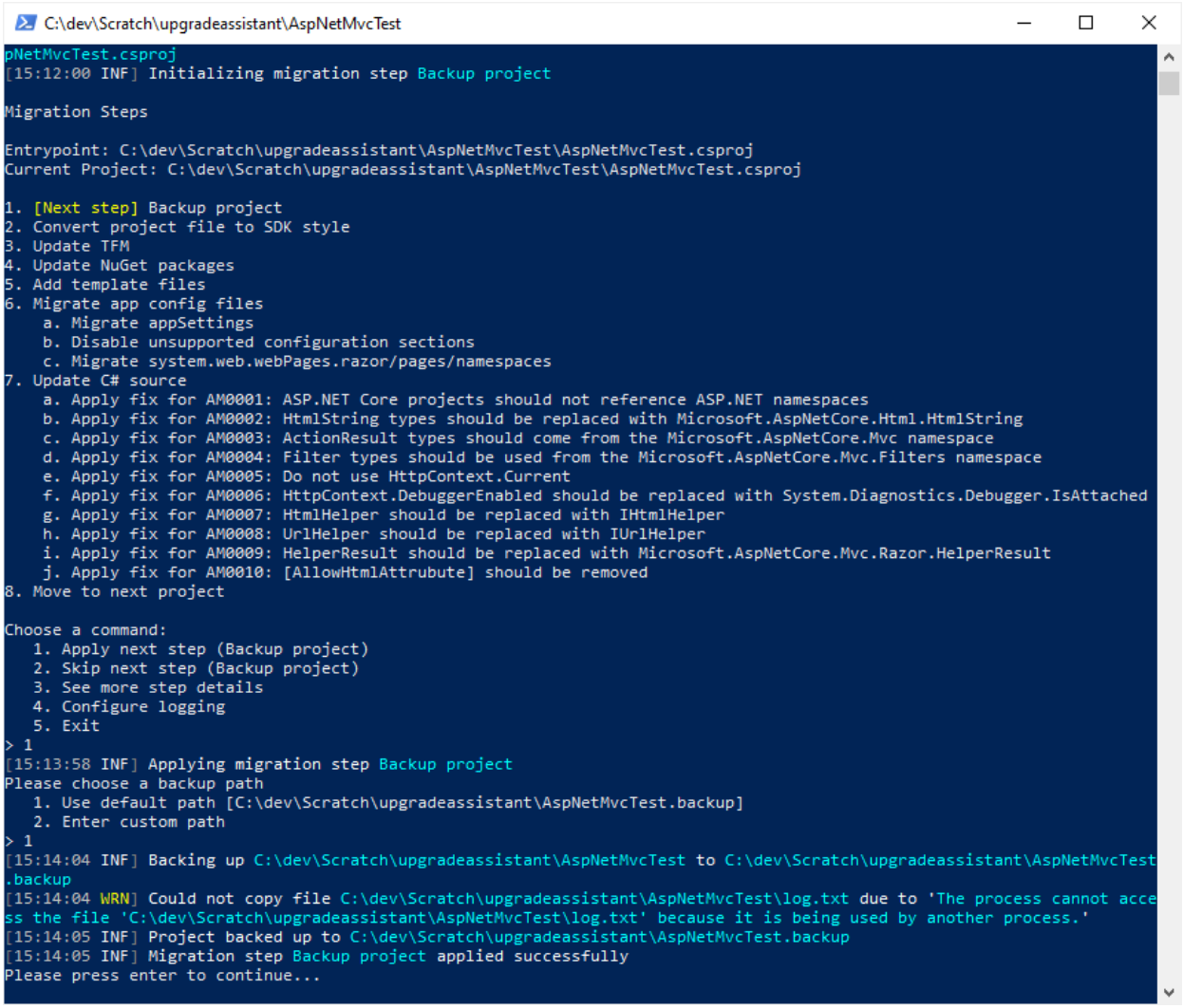
```
upgrade-assistant upgrade .\AspNetMvcTest.csproj
```

该工具将运行并显示它将执行的步骤列表。

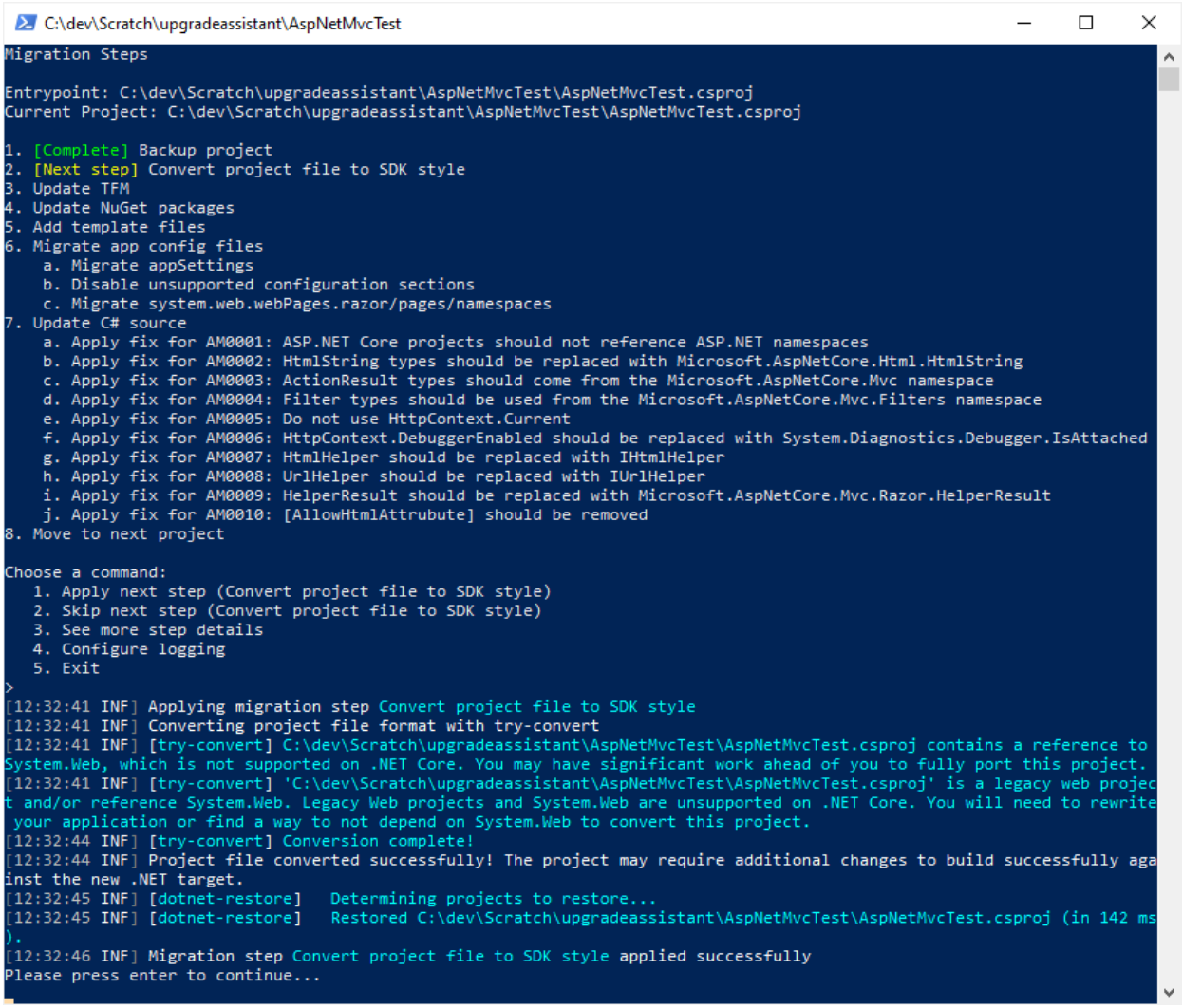


完成每个步骤后，该工具都会提供一组命令，用户可应用这些命令，也可跳过下一步骤、查看更多详细信息、配置日志记录或退出该过程。如果该工具检测到某个步骤将不执行任何操作，它会自动跳过该步骤，转到下一步骤，直到到达有要执行的操作的步骤为止。如果未进行其他任何选择，那么按 Enter 将执行下一步。

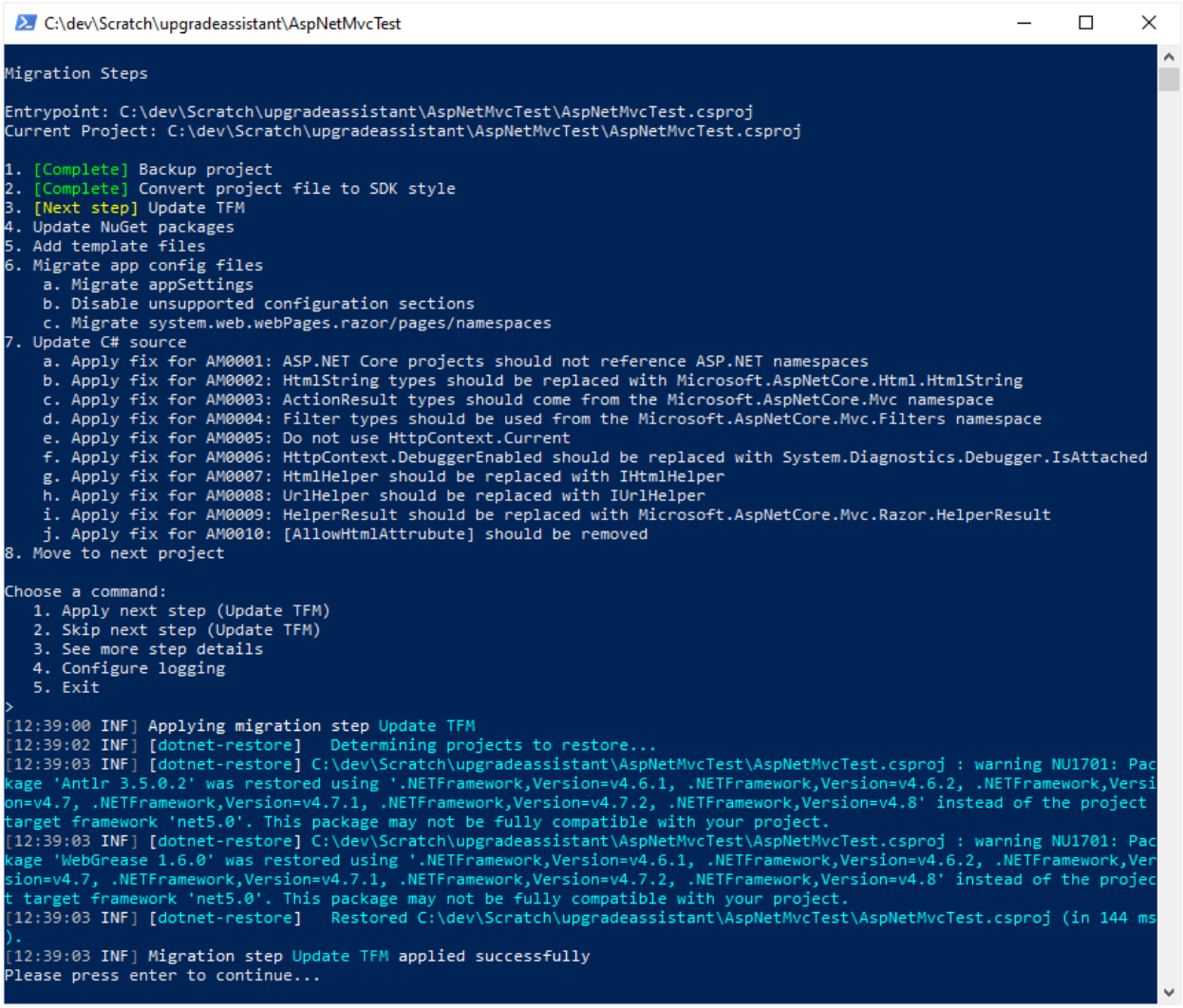
在此示例中，每次都会选择“应用”步骤。第一步是备份项目。



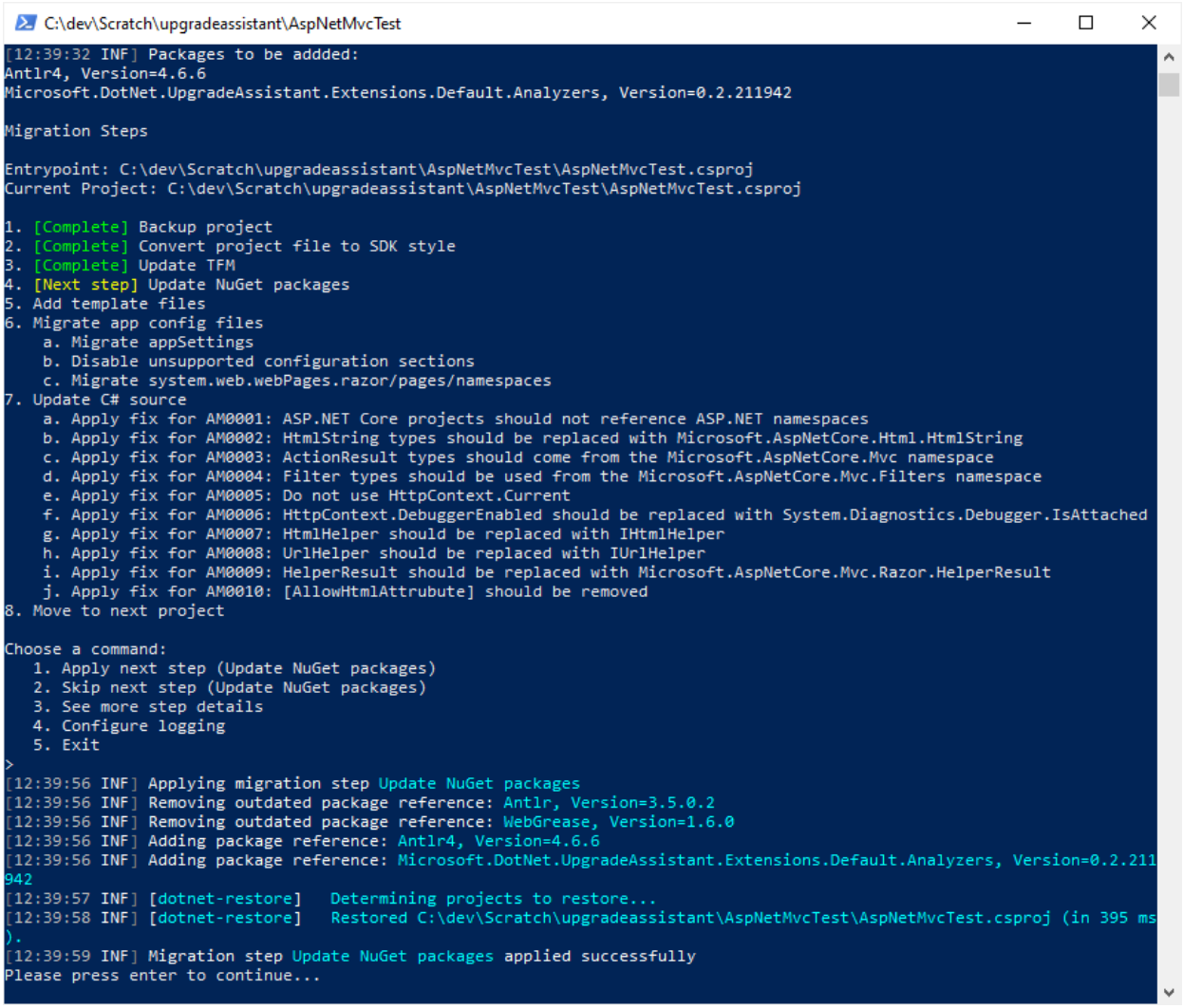
该工具会提示输入自定义路径进行备份或使用默认路径，后者会将项目备份放在具有 `.backup` 扩展名的同一文件夹中。此工具接下来做的是将项目文件转换为 SDK 样式。



更新项目格式后，下一步是更新项目的 TFM。



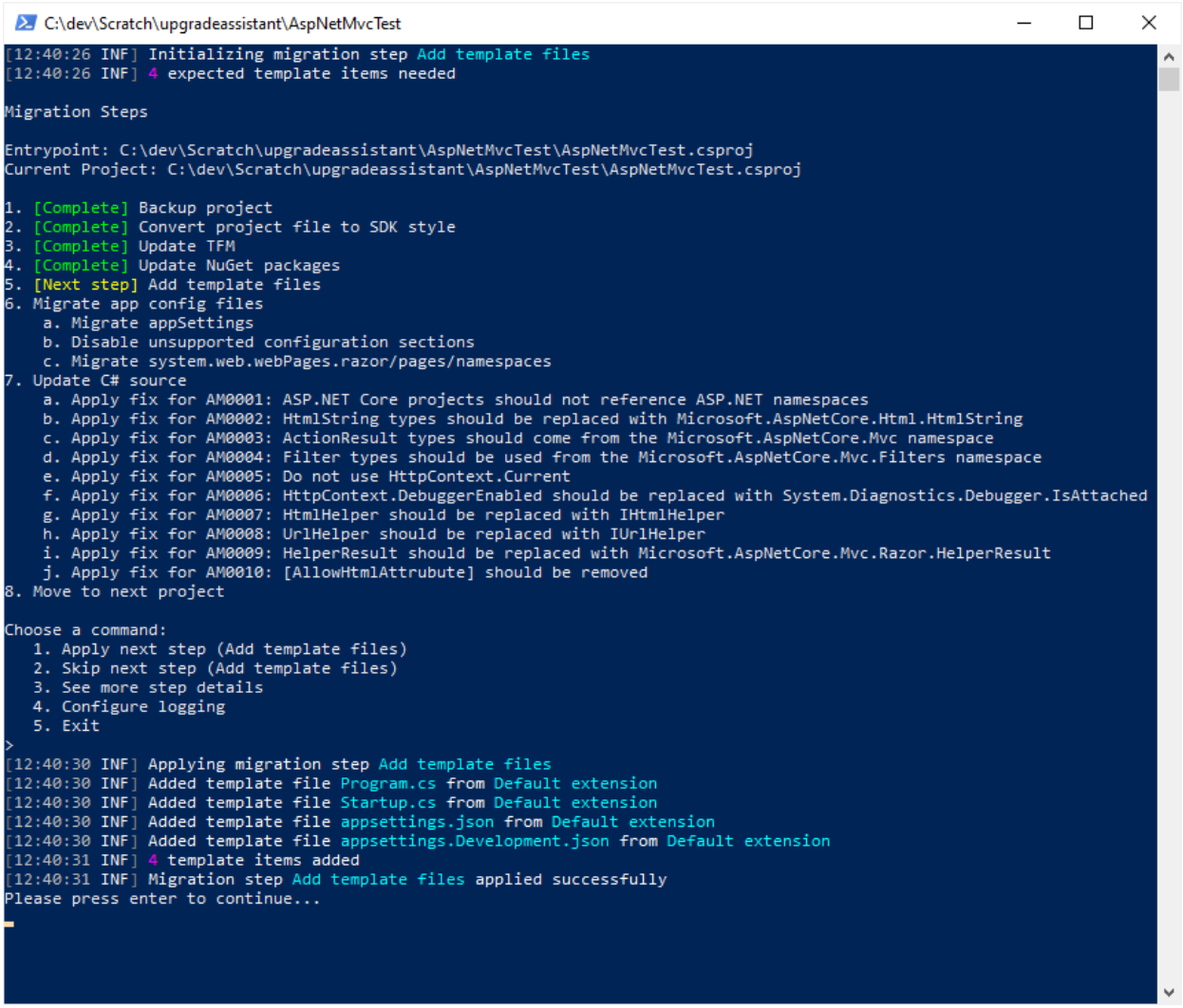
接下来，该工具会更新项目的 NuGet 包。多个包需要更新，且会添加一个新的分析器包。



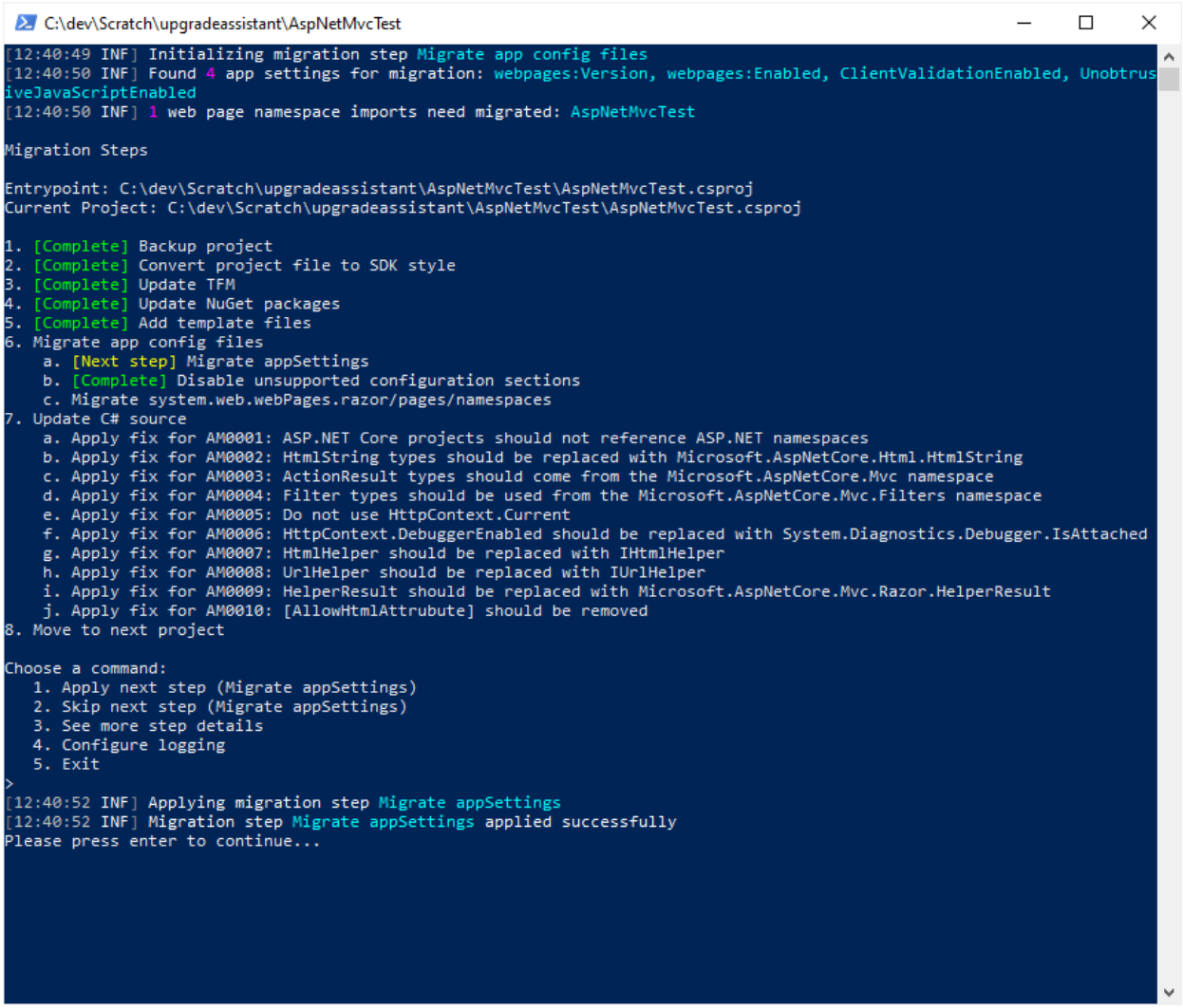
更新包后，接下来是添加模板文件（如果有）。该工具指示有 4 个必须添加的预期模板项，随后它会添加这些项。以下是模板文件的列表：

- `Program.cs`
- `Startup.cs`
- `appsettings.json`
- `appsettings.Development.json`

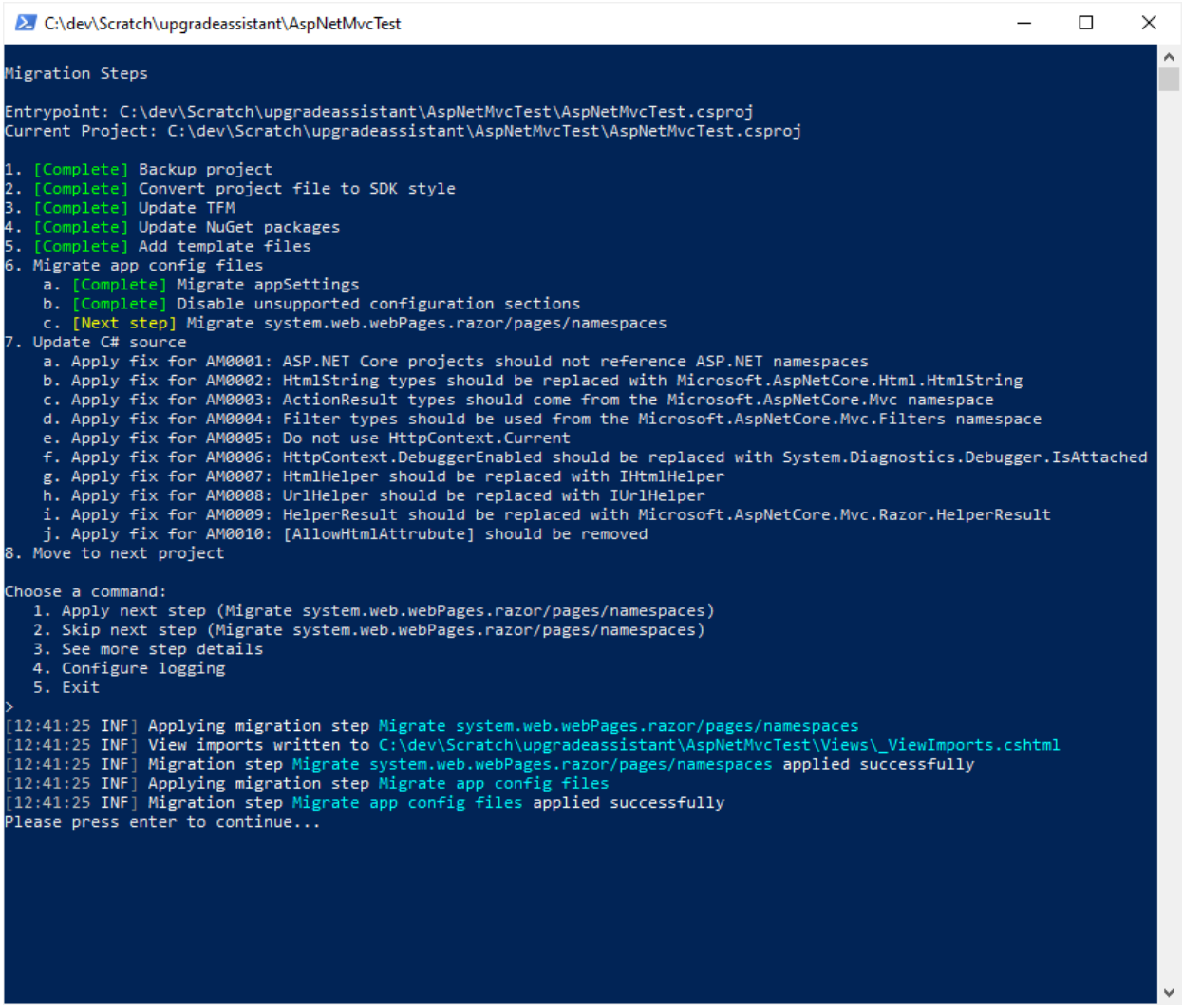
ASP.NET Core 会使用这些文件来进行应用启动和配置。



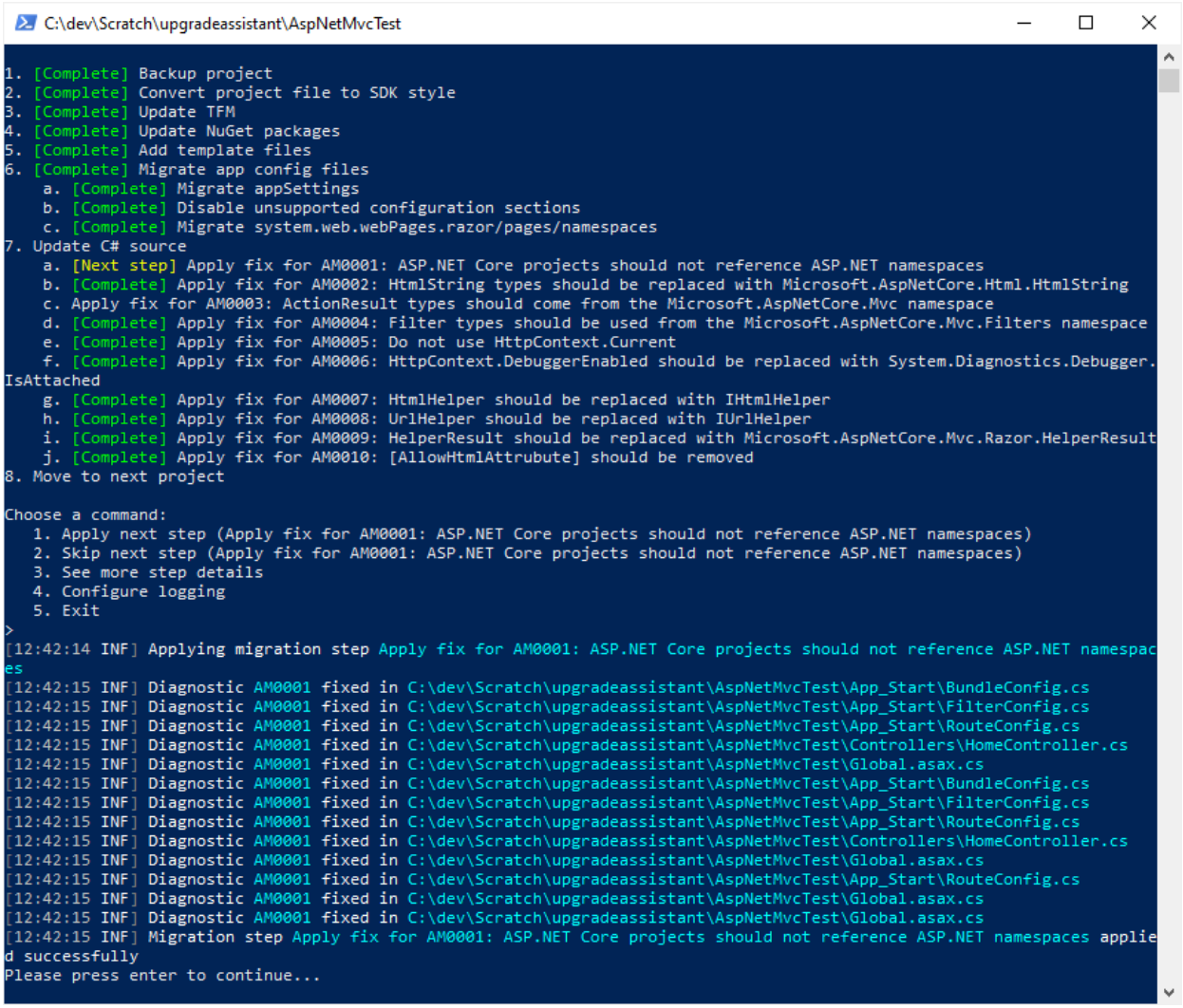
接下来，该工具会迁移配置文件。该工具会标识应用设置并禁用不受支持的配置部分，然后迁移 `appSettings` 配置值。



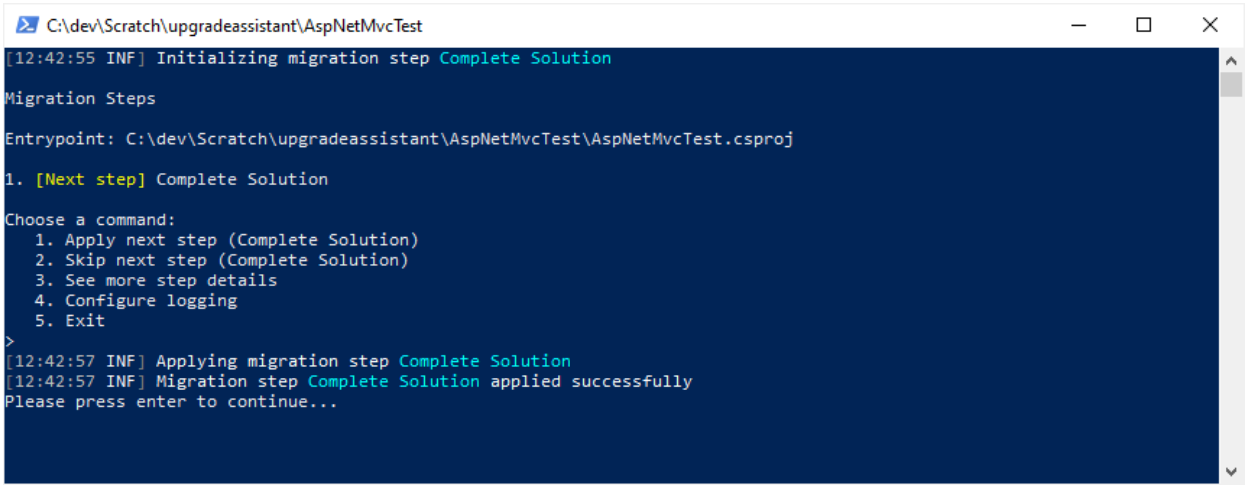
该工具通过迁移 `system.web.webPages.razor/pages/namespaces` 来完成配置文件的迁移。



该工具会应用已知的修补程序来将 C# 引用迁移到其新的对应项。



这是最后一个项目，因此下一步是"移动到新的项目"，它提示完成迁移整个解决方案的过程。



完成此过程后，打开项目文件并进行查看。查找静态文件，如下所示：

```
<ItemGroup>
  <Content Include="fonts\glyphicons-halflings-regular.woff2" />
  <Content Include="fonts\glyphicons-halflings-regular.woff" />
  <Content Include="fonts\glyphicons-halflings-regular.ttf" />
  <Content Include="fonts\glyphicons-halflings-regular.eot" />
  <Content Include="Content\bootstrap.min.css.map" />
  <Content Include="Content\bootstrap.css.map" />
  <Content Include="Content\bootstrap-theme.min.css.map" />
  <Content Include="Content\bootstrap-theme.css.map" />
  <Content Include="Scripts\jquery-3.4.1.slim.min.map" />
  <Content Include="Scripts\jquery-3.4.1.min.map" />
</ItemGroup>
```

该由 Web 服务器处理的静态文件应移动到名为 `wwwroot` 的根级别文件夹下适当的文件夹中。有关详细信息，请查看 ASP.NET Core 中的静态文件 。移动文件后，可删除项目文件中与这些文件对应的 `<Content>` 元素。事实上，可删除所有 `<Content>` 元素及其包含组。此外，应删除指向客户端库（如 `bootstrap` 或 `jQuery`）的所有 `<PackageReference>`。

默认情况下，项目将被转换为类库。请将第一行的 `Sdk` 属性更改为 `Microsoft.NET.Sdk.Web`，并将 `<TargetFramework>` 设置为 `net5.0`。编译该项目。此时，错误数应当相当小。在移植新的 ASP.NET 4.6.1 MVC 项目时，其余错误引用 `App_Start` 文件夹中的文件：

- BundleConfig.cs
- FilterConfig.cs
- RouteConfig.cs

可删除这些文件和整个 `App_Start` 文件夹。同样，可删除 `Global.asax` 和 `Global.asax.cs` 文件。

此时，只剩下与捆绑相关的错误。可通过多种方式在 SP.NET Core 中配置捆绑和缩减。选择最适合你的项目的任何内容。

故障排除提示

使用 .NET 升级助手时，可能会出现一些已知问题。某些情况下，.NET 升级助手在内部使用的 try-convert 工具会出现问题。

有关更多故障排除提示和已知问题，可查看此工具的 GitHub 存储库。

另请参阅

- 将 WPF 应用升级到 .NET 5
- 将 Windows 窗体应用升级到 .NET 5
- .NET 升级助手概述
- .NET 升级助手 GitHub 存储库

升级助手遥测

2021/11/16 •

升级助手包含收集使用情况数据的遥测功能。遥测数据用于帮助了解如何改进工具。

如何选择退出

升级助手遥测功能默认处于启用状态。要选择退出遥测功能，请将 `DOTNET_UPGRADEASSISTANT_TELEMETRY_OPTOUT` 环境变量设置为 `1` 或 `true`。

- Console
- PowerShell
- Bash

在给定值的情况下，创建并分配持久化环境变量。

```
:: Assigns the env var to the value
setx DOTNET_UPGRADEASSISTANT_TELEMETRY_OPTOUT="1"
```

在命令提示符的新实例中，读取环境变量。

```
:: Prints the env var value
echo %DOTNET_UPGRADEASSISTANT_TELEMETRY_OPTOUT%
```

公开

首次运行升级助手时，该工具将显示类似于以下内容的文本。文本可能会因运行的工具版本而略有不同。此"首次运行"体验是 Microsoft 通知用户有关数据收集信息的方式。

```
Telemetry
---------
The .NET tools collect usage data in order to help us improve your experience.
It is collected by Microsoft and shared with the community. You can opt-out of
telemetry by setting the DOTNET_UPGRADEASSISTANT_TELEMETRY_OPTOUT environment
variable to '1' or 'true' using your favorite shell.
```

若要禁止显示"首次运行"体验文本，请将 `DOTNET_UPGRADEASSISTANT_SKIP_FIRST_TIME_EXPERIENCE` 环境变量设置为 `1` 或 `true`。

数据点

遥测功能不收集以下数据：

- 收集代码示例

数据将安全地发送到 Microsoft 服务器，并以受限制的访问权限予以保存。

保护你的隐私对我们很重要。如果你怀疑遥测功能正在收集敏感数据，或怀疑数据未得到安全或妥当地处理，请执行以下操作之一：

- 在 dotnet/upgrade-assistant 存储库中发布问题。

- 发送电子邮件至 dotnet@microsoft.com 以进行调查。

遥测功能收集以下数据。

[illegible]	[illegible]
>=0.2.231802	调用时间戳。
>=0.2.231802	用于确定地理位置的三个八进制数 IP 地址。
>=0.2.231802	操作系统和版本。
>=0.2.231802	工具在其上运行的运行时 ID (RID)。
>=0.2.231802	工具是否在容器中运行。
>=0.2.231802	经过哈希处理的媒体访问控制 (MAC) 地址：计算机的加密 (SHA256) 哈希唯一 ID。
>=0.2.231802	内核版本。
>=0.2.231802	升级助手版本。
>=0.2.231802	调用的命令和参数名称。不收集实际参数值。
>=0.2.231802	使用的 MSBuild 版本。
>=0.2.231802	哈希解决方案 ID（如果没有可用 ID，则为哈希路径）。
>=0.2.231802	每个项目的哈希项目 ID（如果没有可用 ID，则为哈希路径）。
>=0.2.231802	每个入口点的哈希项目 ID（如果没有可用 ID，则为哈希路径）。
>=0.2.231802	对于每个步骤，初始化和应用步骤的时间。
>=0.2.231802	对于每个步骤，所选的决策（例如，`apply`）。

其他资源

- .NET SDK 遥测
- .NET CLI 遥测数据

移植代码时可能会发生中断性变更

2021/11/16 •

影响兼容性的更改(也称为中断性变更)将在 .NET 版本之间发生。由于某些技术不可用，从 .NET Framework 移植到 .NET 时，更改会产生影响。此外，你也可能遇到中断性变更，仅仅因为 .NET 是一种跨平台技术，而 .NET Framework 不是。

Microsoft 致力于在 .NET 版本之间保持高级别的兼容性，因此，虽然发生了中断性变更，但它们都是经过仔细考虑的。

在升级主版本之前，请查看中断性变更文档，以了解可能影响你的变更。

中断性变更类别

有几种类型的中断性变更...

- 公共协定修改
- 行为变更
- 平台支持
- 内部实现变更
- 代码更改

有关允许或不允许哪些操作的详细信息，请参阅影响兼容性的变更

兼容性类型

兼容性是指在 .NET 实现(而不是最初开发代码的版本)上编译或运行代码的能力。

变更可以通过六种不同的方式影响兼容性...

- 行为变更
- 二进制兼容性
- 源兼容性
- 设计时兼容性
- 向后兼容性
- 向前兼容性

有关详细信息，请参阅代码更改如何影响兼容性

查找中断性变更

将记录影响兼容性的更改，在从 .NET Framework 移植到 .NET 或升级到较新版本的 .NET 之前，应查看这些变更。

有关详细信息，请参阅中断性变更参考概述

分析依赖项以将代码从 .NET Framework 移植到 .NET 中

2021/11/16 •

若要确定项目中不支持的第三方依赖项，必须先了解依赖项。外部依赖项是在项目中引用但没有自行构建的 NuGet 包或 `.dll` 文件。

通过将代码移植到 .NET Standard 2.0 或更低版本，可确保该代码可用于 .NET Framework 和 .NET。但是，如果你不需要将库用于 .NET Framework，请考虑以最新版本的 .NET 为目标。

将 NuGet 包迁移到 `PackageReference`

.NET 不能使用 packages.config 文件引用 NuGet。.NET 和 .NET Framework 都可以使用 PackageReference 来指定包依赖项。如果你使用 packages.config 在项目中指定包，请将其转换为 `PackageReference` 格式。

如需了解如何迁移，请参阅从 packages.config 迁移到 PackageReference一文。

升级 NuGet 包

将项目迁移为 `PackageReference` 格式后，验证包是否与 .NET 兼容。

首先，请将包升级到可以使用的最新版本。可通过 Visual Studio 中的 NuGet 包管理器 UI 完成此操作。包依赖项的较新版本可能已经与 .NET Core 兼容。

分析包依赖项

如果尚未验证转换和升级后的包依赖项是否适用于 .NET Core，可通过以下几种方法实现此目的：

使用 nuget.org 分析 NuGet 包

可以在包页面“依赖项”部分的 nuget.org 上查看每个包所支持的目标框架名字对象 (TFM) 。

尽管使用站点验证兼容性是一种比较简单的方法，但站点上并未提供所有包的“依赖项”信息 。

使用 NuGet 包资源管理器分析 NuGet 包

NuGet 包本身就是一组包含特定于平台的程序集的文件夹。检查包中是否有包含兼容程序集的文件夹。

检查 NuGet 包文件夹最简单方法是使用 NuGet 包资源管理器工具。安装完成后，使用以下步骤查看文件夹名称：

1. 打开 NuGet 包资源管理器。
2. 单击“从在线源打开包” 。
3. 搜索包的名称。
4. 从搜索结果中选择包的名称，然后点击“打开” 。
5. 展开右侧的“lib”文件夹并查看文件夹名称 。

查找名称使用以下模式之一的文件夹：`netstandardX.Y`、`netX.Y` 或 `netcoreappX.Y`。

这些值是映射到 .NET Standard、.NET 和 .NET Core 各版本的目标框架名字对象 (TFM)，它们均与 .NET 兼容。

> **IMPORTANT**
> 在查看包支持的 TFM 时，请注意，除 `netstandard*` 以外的 TFM 面向的是 .NET 的特定实现，如 .NET 5、.NET Core 或 .NET Framework。从 .NET 5 开始，`net*` TFM（未指定操作系统）实际上取代了 `netstandard*` 成为可移植目标。例如，`net5.0` 面向 .NET 5 API 图面，并且是跨平台友好的，而 `net5.0-windows` 面向在 Windows 操作系统上实现的 .NET 5 API 图面。

.NET Framework 兼容性模式

在分析完 NuGet 包之后，你可能会发现它们仅面向 .NET Framework。

从 .NET Standard 2.0 开始，引入了 .NET Framework 兼容性模式。此兼容性模式允许 .NET Standard 和 .NET Core 项目引用 .NET Framework 库。引用 .NET Framework 库不适用于所有项目（如库使用 Windows Presentation Foundation (WPF) API 时），但它的开启了很多移植方案。

在项目中引用面向 .NET Framework 的 NuGet 包时（如 `Huitian.PowerCollections`），你会收到类似于以下示例的包回退警告 (NU1701)：

```
NU1701: Package 'Huitian.PowerCollections 1.0.0' was restored using '.NETFramework,Version=v4.6.1' instead of the project target framework '.NETStandard,Version=v2.0'. This package may not be fully compatible with your project.
```

当添加包以及每次生成时都会显示该警告，以确保在项目中测试该包。如果项目按预期工作，可通过在 Visual Studio 中编辑包属性或在最喜欢的代码编辑器中手动编辑项目文件来取消该警告。

若要通过编辑项目文件来取消警告，请找到要取消警告的包的 `PackageReference` 条目并添加 `NoWarn` 属性。`NoWarn` 属性接受所有警告 ID 的逗号分隔列表。以下示例显示如何通过手动编辑项目文件来取消 `Huitian.PowerCollections` 包的 `NU1701` 警告：

```
<ItemGroup>
  <PackageReference Include="Huitian.PowerCollections" Version="1.0.0" NoWarn="NU1701" />
</ItemGroup>
```

有关如何在 Visual Studio 中取消编译器警告的详细信息，请参阅取消 NuGet 包的警告。

如果 NuGet 包无法在 .NET 上运行

如果所依赖的 NuGet 包无法在 .NET Core 上运行，可以执行以下几项操作：

- 如果项目是开放源代码并托管在诸如 GitHub 中，则可以直接与开发人员交流。
- 可直接在 nuget.org 上联系作者。搜索包并单击包页面左侧的“联系所有者”。
- 可以搜索在 .NET Core 上运行的其他包，它与所使用的包进行的是相同的任务。
- 可以尝试自己编写包所执行的代码。
- 可以通过更改应用的功能来清除对包的依赖性，至少在该包有可用的兼容性版本之前都能这样做。

请注意，开源项目维护人员和 NuGet 包发布者通常是志愿者。他们因为关注某个特定领域而免费提供服务，通常从事另外一份职业。当联系他们请求 .NET Core 支持时请注意这点。

如果使用上述任何选项都无法解决问题，则可能需要移植到最近版本的 .NET Core。

.NET 团队需要了解哪些库最需要使用 .NET Core 支持。你可以发送电子邮件到 dotnet@microsoft.com，谈谈你想要使用的库。

分析非 NuGet 依赖项

用户可能拥有不是 NuGet 包的依赖项，如文件系统中的 DLL。确定该依赖项是否可移植的唯一方法是运行 .NET

可移植性分析器工具。该工具可以分析面向 .NET Framework 的程序集，并识别不可移植到其他 .NET 平台（如 .NET Core）的 API。可将该工具作为控制台应用程序或作为 Visual Studio 扩展来运行。

后续步骤

- 有关从 .NET Framework 移植到 .NET 的概述

使用 Windows 兼容性包将代码移植到 .NET 5+

2021/11/16 •

将现有代码从 .NET Framework 移植到 .NET 时发现的一些最常见问题是对只有在 .NET Framework 中才能找到的 API 和技术的依赖。Windows 兼容性包提供了许多这样的技术，因此可以更轻松地生成 .NET 应用程序和 .NET Standard 库。

兼容包是 .NET Standard 2.0 的逻辑扩展，它大幅扩展了 API 集。现有代码几乎不修改即可编译。为了信守 .NET Standard 的承诺（"所有 .NET 实现都提供的一组 API"），.NET Standard 不包括无法跨所有平台工作的技术，如注册表、Windows Management Instrumentation (WMI) 或反射发出 API。Windows 兼容性包位于 .NET Standard 顶部，提供对这些仅限 Windows 的技术的访问权限。对于那些想要迁移到 .NET 但至少第一步仍计划停留在 Windows 上的客户，这尤其有用。在这种情况下，可以使用仅限 Windows 的技术消除迁移障碍。

包的内容

Windows 兼容性包通过 Microsoft.Windows.Compatibility NuGet 包提供，并可从面向 .NET 或 .NET Standard 的项目引用。

它提供了约 20,000 个 API，包括仅限 Windows 的 API 以及以下技术领域中的跨平台 API：

- 代码页
- CodeDom
- 配置
- 目录服务
- 绘图
- ODBC
- 权限
- 端口
- Windows 访问控制列表 (ACL)
- Windows Communication Foundation (WCF)
- Windows 加密
- Windows 事件日志
- Windows 管理规范 (WMI)
- Windows 性能计数器
- Windows 注册表
- Windows 运行时缓存
- Windows 服务

有关详细信息，请参阅兼容包规范。

入门

1. 移植之前，请确保查看移植过程。
2. 将现有代码移植到 .NET 或 .NET Standard 时，请安装 Microsoft.Windows.Compatibility NuGet 包。

   如果要停留在 Windows 上，则已准备完毕。
3. 如果要在 Linux 或 macOS 上运行 .NET 应用程序或 .NET Standard 库，请使用平台兼容性分析器查找不会跨平台工作的 API 的使用情况。

4. 删除这些 API 的使用情况、将其替换为跨平台替代项，或使用平台检查对其实施保护，例如：

```
private static string GetLoggingPath()
{
    // Verify the code is running on Windows.
    if (RuntimeInformation.IsOSPlatform(OSPlatform.Windows))
    {
        using (var key = Registry.CurrentUser.OpenSubKey(@"Software\Fabrikam\AssetManagement"))
        {
            if (key?.GetValue("LoggingDirectoryPath") is string configuredPath)
                return configuredPath;
        }
    }

    // This is either not running on Windows or no logging path was configured,
    // so just use the path for non-roaming user-specific data files.
    var appDataPath = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData);
    return Path.Combine(appDataPath, "Fabrikam", "AssetManagement", "Logging");
}
```

有关演示，请查看 Windows 兼容性包的第 9 频道视频。

另请参阅

- 有关从 .NET Framework 移植到 .NET 的概述
- ASP.NET 到 ASP.NET Core 迁移
- 将 .NET Framework WPF 应用迁移到 .NET
- 将 .NET Framework Windows 窗体应用迁移到 .NET

.NET Framework 技术在 .NET Core 和 .NET 5+ 上不可用

2021/11/16 •

一些适用于 .NET Framework 库的技术不可用于 .NET 5 及更高版本(和 .NET Core),例如应用域、远程处理和代码访问安全性 (CAS)。如果你的库依赖于本页中列出的一项或多项技术,请考虑使用提及的替代方法。

有关 API 兼容性的详细信息,请参阅 .NET 中断性变更。

应用程序域

应用程序域 (AppDomain) 可将应用相互隔离。AppDomain 需要运行时支持,并且耗费的资源成本较高。不支持创建其他应用域,也尚未计划在将来添加此功能。对于代码隔离,将流程或容器用作备用。若要动态加载程序集,请使用 AssemblyLoadContext 类。

.NET 5+ 公开了一些 AppDomain API 曲面,以便可以更轻松地从 .NET Framework 进行代码迁移。一些 API 可正常工作(例如 AppDomain.UnhandledException),一些成员不会执行任何操作(例如 SetCachePath),也有一些会引发 PlatformNotSupportedException(例如 CreateDomain)。对照 dotnet/runtime GitHub 存储库中的 `System.AppDomain` 引用源检查所使用的类型。确保选择与已实现的版本相匹配的分支。

远程处理

.NET 5 及更高版本(和 .NET Core)不支持 .NET 远程处理。.NET 远程处理被认为是存在问题的体系结构。它用于在不再受支持的应用程序域之间进行通信。同样,远程处理也需要运行时支持,进行维护的成本较高。

对于简单的跨进程通信,可将进程间通信 (IPC) 机制视为远程处理的备用方案,如 System.IO.Pipes 类或 MemoryMappedFile 类。对于更复杂的场景,开放源代码 StreamJsonRpc 项目提供了一个跨平台的 .NET Standard 远程处理框架,该框架在现有流或管道连接的基础上工作。

对于跨计算机的通信,可将基于网络的解决方案用作备用方案。最好使用低开销纯文本协议,例如 HTTP。此处,ASP.NET Core 使用的 Web 服务器 Kestrel Web 服务器是一个选择。也可考虑将 System.Net.Sockets 用于基于网络的跨计算机的方案。前面提到的 StreamJsonRpc 可用于通过 Web 套接字进行 JSON 或二进制(通过 MessagePack)通信。

有关更多消息传送选项,请参阅 .NET 开放源代码开发人员项目:消息传送。

代码访问安全性 (CAS)

沙盒依赖为托管应用程序或库使用或运行提供资源的运行时或框架进行限制,其在 .NET Framework 上不受支持,因此在 .NET Core 和 .NET 5+ 上也不受支持。.NET Framework 和运行时中存在太多发生特权提升以继续将 CAS 视为安全边界的情况。此外,CAS 使实现更加复杂,通常会对无意使用它的应用程序造成正确性-性能影响。

可使用操作系统提供的安全边界,例如虚拟化、容器或具有最少特权集的用于运行进程的用户帐户。

安全透明度

与 CAS 相似,借助安全透明度可以通过声明性方式将沙盒代码与安全关键代码隔离,但是不再支持将它作为安全边界。Silverlight 大规模使用了此功能。

可使用操作系统提供的安全边界,例如虚拟化、容器或用于运行进程的用户帐户具有最少的一组特权。

System.EnterpriseServices

.NET Core 和 .NET 5 及更高版本不支持 System.EnterpriseServices (COM+)。

Workflow Foundation 和 WCF

.NET 5+（包括 .NET Core）不支持 Windows Workflow Foundation (WF) 和 Windows Communication Foundation (WCF)。有关替代方法，请参阅 CoreWF 和 CoreWCF。

保存反射生成的程序集

.NET 5+（包括 .NET Core）不支持保存由 System.Reflection.Emit API 生成的程序集。AssemblyBuilder.Save 方法在 .NET 5+（包括 .NET Core）中不可用。此外，AssemblyBuilderAccess 枚举的以下字段不可用：

- ReflectionOnly
- RunAndSave
- Save

作为替代方法，请考虑 ILPack 库。

有关详细信息，请参阅 dotnet/runtime 问题 15704。

加载多模块程序集

由多个模块组成的程序集（在 MSBuild 中设置为 `OutputType=Module`）在 .NET 5+（包括 .NET Core）中不受支持。

作为替代方法，请考虑将各个模块合并到单个程序集文件中。

另请参阅

- 有关从 .NET Framework 移植到 .NET 的概述

查找代码中不支持的 API

2021/11/16 •

.NET Framework 代码中的 API 可能由于多种原因而不受支持。这些原因包括：容易解决的原因（例如命名空间更改）和难以解决的原因（例如不支持整个技术）。第一步是确定不再支持哪些 API，然后确定正确的解决方法。

.NET 可移植性分析器

.NET 可移植性分析器是一种工具，可分析程序集并为应用程序或库提供有关缺失的 .NET API 的详细报告，以便在指定的目标 .NET 平台上实现可移植性。

若要使用 Visual Studio 中的 .NET 可移植性分析器，请从市场中安装此扩展。

有关详细信息，请参阅 .NET 可移植性分析器。

升级助手

.NET 升级助手是一款可以在不同类型的 .NET Framework 应用上运行的命令行工具。它旨在帮助将 .NET Framework 应用升级到 .NET 5。在运行此工具后，大多数情况下，应用将需要更多操作才能完成迁移。此工具会安装可以帮助完成迁移的分析器。此工具适用于以下类型的 .NET Framework 应用程序：

- Windows 窗体
- WPF
- ASP.NET MVC
- 控制台
- 类库

此工具其他工具中的 .NET 可移植性分析器，并指导迁移过程。若要详细了解此工具，请参阅 .NET 升级助手概述。

移植代码的先决条件

2021/11/16 •

进行所需的更改，以生成和运行 .NET 应用程序，然后开始工作以移植你的代码。仍然可以在生成和运行 .NET Framework 应用程序的同时完成这些更改。

升级到必需的工具

升级到支持将面向的 .NET 版本的 MSBuild/Visual Studio 版本。有关详细信息，请参阅 .NET SDK、MSBuild 和 VS 之间的版本控制关系。

更新 .NET Framework 目标版本

建议将 .NET Framework 应用的目标设定为版本 4.7.2 或更高版本。在 .NET Standard 不支持现有 API 情况下，这可确保最新备用 API 的可用性。

对于每个想要移植的项目，请在 Visual Studio 中执行以下操作：

1. 右键单击该项目，然后选择“属性”。
2. 在“目标框架”下拉列表中，选择“.NET Framework 4.7.2”。
3. 重新编译该项目。

因为项目现在面向 .NET Framework 4.7.2，因此可使用该版本的 .NET Framework 作为移植代码的基准。

更改为 PackageReference 格式

将所有引用转换为 PackageReference 格式。

转换为 SDK 样式项目格式

将项目转换为 SDK 样式格式。

更新依赖项

将依赖项更新为可用的最新版本，并在可能的情况下更新为 .NET Standard 版本。

后续步骤

- 创建移植计划

创建移植计划

2021/11/16 •

在直接了解代码之前，请花时间完成建议的迁移前步骤。本文让你深入了解可能遇到的问题类型，并帮助确定最有意义的方法。

移植代码

继续下一步之前，请确保遵循移植代码的先决条件。准备好决定最适合你的方法并开始移植代码。

主要处理编译器

此方法可能较适合小项目或不会用很多 .NET Framework API 的项目。此方法很简单：

1. 可选择在项目上运行 ApiPort。若运行 ApiPort，则从报告获取有关需要解决的问题的信息。
2. 将所有代码复制到新的 .NET 项目。
3. 查看可移植性报表（如果已生成）时，解决编译器错误，直至项目完全得到编译。

尽管它是非结构化的，但这种以代码为中心的方法通常可以快速解决问题。只包含数据模型的项目可能是此方法的理想选择。

可移植性问题得到解决前停留在 .NET Framework 上

如果更希望拥有在整个过程期间编译的代码，此方法可能是最佳选择。该方法如下所示：

1. 在项目上运行 ApiPort。
2. 通过使用可移植的不同 API 解决问题。
3. 记录阻止你使用直接替代方案的所有区域。
4. 对所有要移植的项目重复前面的步骤，直到确信每个项目都做好被复制到新的 .NET 项目中的准备。
5. 将代码复制到新的 .NET 项目中。
6. 解决所有已记录的不存在直接替代方案的问题。

这种谨慎的方法比单纯解决编译器错误更有条理，但相对而言，它仍以代码为中心，且优点是始终拥有编译的代码。解决不能通过只使用另一个 API 解决的某些问题的方法大不相同。你可能会发现对于某些项目，需要制定更全面的计划，这将在下一种方法中介绍。

制定全面的攻击计划

此方法可能最适合大型或更复杂的项目，在这种情况下，为支持 .NET，可能必需重构代码或将某些代码区域完全重写。该方法如下所示：

1. 在项目上运行 ApiPort。
2. 了解每个非可移植类型使用的位置以及位置对整体可移植性的影响。
   - 了解这些类型的特性。它们是否数量少，但使用频繁？ 它们是否数量大，但使用不频繁？ 它们是串联使用，还是在整个代码中传播？
   - 是否可以轻松隔离不可移植的代码，以便可以更有效地处理它？
   - 是否需要重构代码？
   - 对于这些不可移植的类型，是否存在可完成相同任务的备用 API？ 例如，如果使用的是 WebClient 类，则改用 HttpClient 类。
   - 是否存在其他可用于完成任务的可移植 API，即使它不是直接替代 API？ 例如，如果使用 XmlSchema 来分析 XML，但是无需 XML 架构发现，则可使用 System.Xml.Linq API 并自行实现分析，而不是依赖于某个 API。

3. 如果具有难以移植的程序集，是否值得将其暂时留在 .NET Framework 上？ 以下是一些需要考虑的事项：

   - 库中可能具有某些与 .NET 不兼容的功能，因为它过于依赖 .NET Framework 或 Windows 特定的功能。是否值得暂时搁置该功能，并直到资源可用于移植这些功能，才发布具有较少功能的库的临时 .NET 版本？
   - 重构是否有用？

4. 编写自己对不可用 .NET Framework API 的实现是否合理？

   可以考虑复制、修改，并使用 .NET Framework 参考源中的代码。参考源代码已在 MIT 许可证下获得许可，因此可以自由选择将此源作为自己代码的基础。只需确保在代码中正确设置 Microsoft。

5. 根据不同项目的需要，重复此过程。

分析阶段可能需要一些时间，具体取决于代码库的大小。在此阶段花费时间（尤其是在具有复杂的代码库时），全面了解所需的更改范围并制定计划，最终通常可节省时间。

计划可能包括对代码库做重大更改，同时仍面向 .NET Framework 4.7.2。这是前一种方法更有条理的版本。着手执行计划的方式具体取决于代码库。

混合方法

在每个项目的基础上，可能会将上述方法进行混合。执行哪些操作对你和代码库最有意义。

移植测试

要确保移植代码后一切正常的最佳方式是在将代码移植到 .NET 时进行测试。为此，需要使用将针对 .NET 生成和运行测试的测试框架。当前，有三个选择：

- xUnit
  - 入门
  - 将 MSTest 项目转换为 xUnit 的工具
- NUnit
  - 入门
  - 关于从 MSTest 迁移到 NUnit 的博客文章
- MSTest

推荐的方法

从根本上讲，移植工作在很大程度上取决于生成 .NET Framework 代码的方式。移植代码的一个好方法是从库的基项开始，这是代码的基础组件。这可能是数据模型或某些其他内容直接或间接使用的基本类和方法。

1. 移植测试项目，该项目测试当前正在移植的库层。
2. 将库中的基项复制到新的 .NET 项目中，然后选择想要支持的 .NET Standard 版本。
3. 进行任何所需的更改，使代码进行编译。大部分内容可能会要求将 NuGet 包依赖项添加到 csproj 文件。
4. 运行测试并进行任何所需调整。
5. 选择下一层代码进行移植，并重复前面的步骤。

如果从库的基项开始并从基项向外移动并根据需要测试每一层，移植将是一个系统化的过程，在这种情况下，问题可以一次隔离到一层代码中。

后续步骤

- .NET 升级助手概述
- 组织项目以支持 .NET Framework 和 .NET Core

组织项目以支持 .NET Framework 和 .NET

2021/11/16 •

可以创建一个并行编译 .NET Framework 和 .NET 的解决方案。本文介绍可帮助你实现此目标的多个项目组织选项。以下是决定如何使用 .NET 设置项目布局时要考虑的一些典型方案。此列表可能无法涵盖所有要求。

- 将现有项目和 .NET 项目合并为单个项目

  优点：

  - 通过编译单个项目（而非多个项目）简化生成过程，每个项目针对不同的 .NET Framework 版本或平台。
  - 由于需要管理单个项目文件，因此简化了多目标项目的源文件管理。添加或删除源文件时，需要手动将备用文件与其他项目进行同步。
  - 轻松生成可供使用的 NuGet 包。
  - 允许使用编译器指令为特定.NET Framework 版本编写代码。

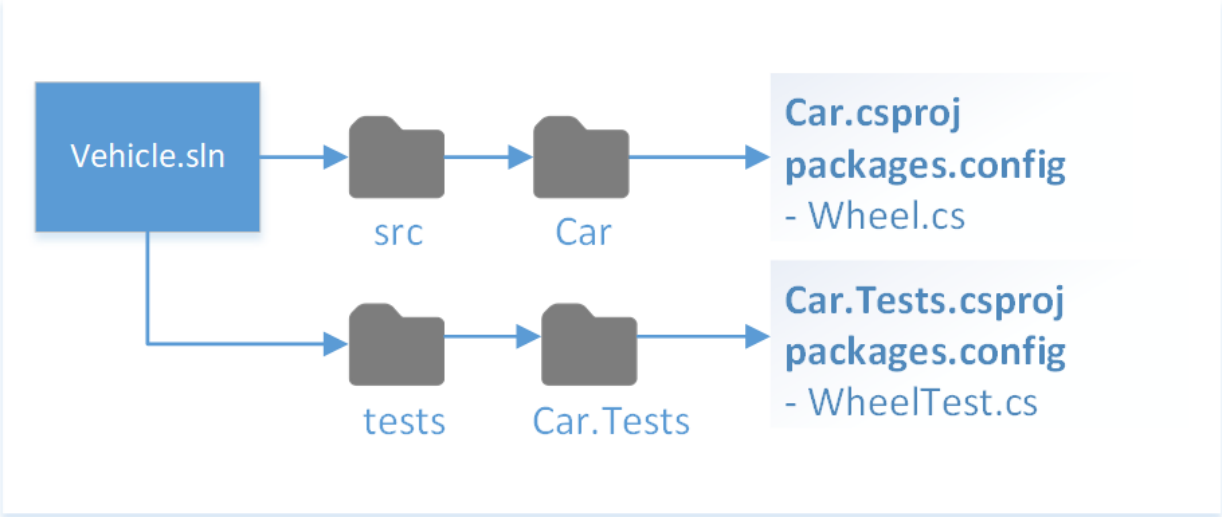
  缺点：

  - 要求开发者使用 Visual Studio 2019 或更高版本来打开现有项目。若要支持 Visual Studio 的早期版本，建议将项目文件保存在不同的文件夹中。
- 使所有项目保持独立

  优点：

  - 支持没有安装 Visual Studio 2019 或更高版本的开发人员和参与者基于现有项目开发。
  - 减少现有项目中出现新 bug 的可能性，因为这些项目中不需要进行任何代码改动。

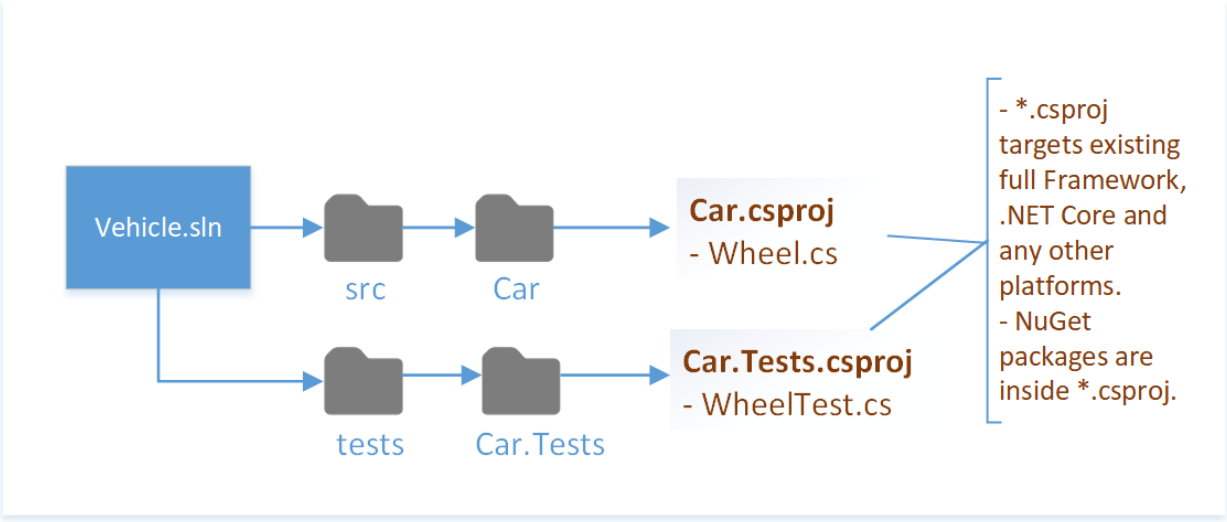
请考虑此示例 GitHub 存储库。下图显示了此存储库的布局：



以下部分介绍了基于示例存储库添加对 .NET 支持的几种方式。

将现有项目替换为多目标的 .NET 项目

重新组织存储库，以便删除任何现有的 *.csproj 文件，并创建以多个框架为目标的单一 *.csproj 文件。这是一项不错的选择，因为单个项目可以编译不同的框架。它还可以处理每个目标框架的不同编译选项和依赖项。



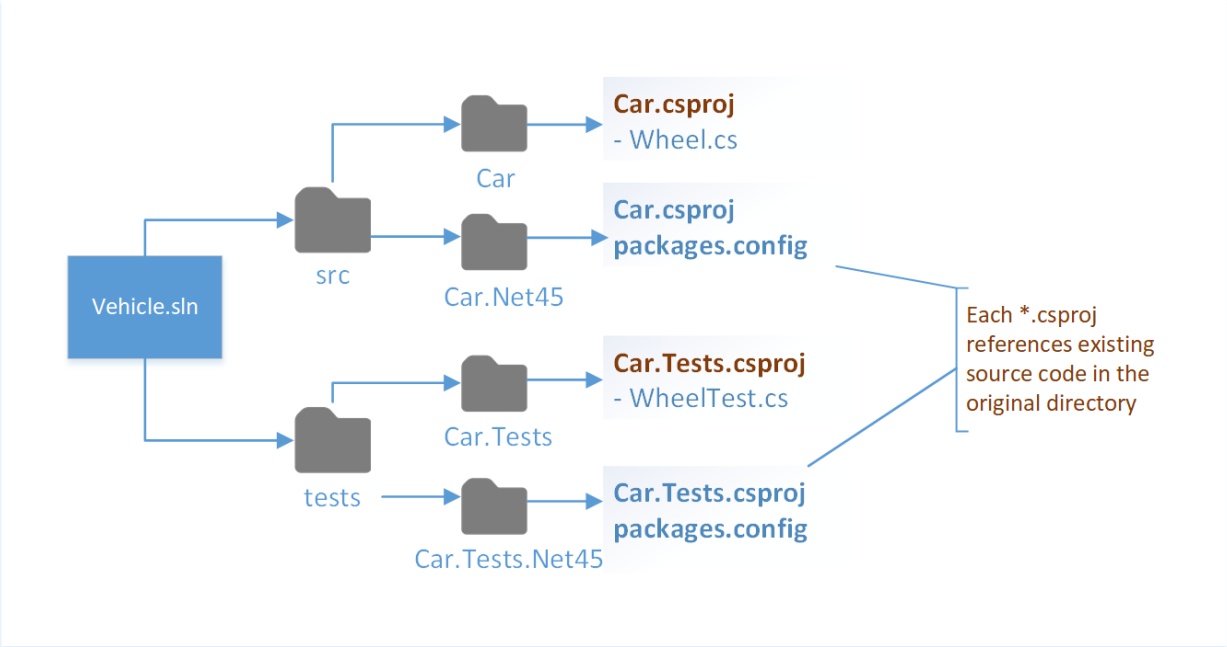
有关代码示例，请参阅 GitHub。

需注意的更改：

- 用新的 .NET *.csproj 替换 packages.config 和 *.csproj 。NuGet 包是使用 `<PackageReference> ItemGroup` 指定的。

保留现有项目并创建 .NET 项目

如果存在以较旧框架为目标的现有项目，可能需要保留这些项目并将 .NET 项目用作将来框架的目标。



有关代码示例，请参阅 GitHub。

将 .NET 项目和现有项目保存在不同的文件夹中。将项目保存在不同的文件夹中可以避免强制使用 Visual Studio 2019 或更高版本。可以创建仅打开旧项目的单独解决方案。

另请参阅

- .NET 移植文档

如何将 C++/CLI 项目移植到 .NET Core

2021/11/16 •

从 .NET Core 3.1 和 Visual Studio 2019 16.4 版开始，C++/CLI 项目可面向 .NET Core。这种支持使你能够将具有 C++/CLI 互操作层的 Windows 桌面应用程序移植到 .NET Core。本文介绍如何将 C++/CLI 项目从 .NET Framework 移植到 .NET Core 3.1。

C++/CLI .NET Core 限制

与其他语言相比，将 C++/CLI 项目移植到 .NET Core 具有一些重要限制：

- .NET Core 的 C++/CLI 支持仅适用于 Windows。
- C++/CLI 项目不能面向 .NET Standard，只能面向 .NET Core（或 .NET Framework）。
- C++/CLI 项目不支持新的 SDK 样式项目文件格式。相反，即使在面向 .NET Core 时，C++/CLI 项目也使用现有的 vcxproj 文件格式。
- C++/CLI 项目不能同时面向多个 .NET 平台。如果需要同时为 .NET Framework 和 .NET Core 生成 C++/CLI 项目，请使用单独的项目文件。
- .NET Core 不支持 `-clr:pure` 或 `-clr:safe` 编译，仅支持新的 `-clr:netcore` 选项（等效于 .NET Framework 的 `-clr`）。

移植 C++/CLI 项目

要将 C++/CLI 项目移植到 .NET Core，请对 vcxproj 文件进行以下更改。这些迁移步骤不同于其他项目类型所需的步骤，因为 C++/CLI 项目不使用 SDK 样式的项目文件。

1. 将 `<CLRSupport>true</CLRSupport>` 属性替换为 `<CLRSupport>NetCore</CLRSupport>`。此属性通常位于特定于配置的属性组中，因此你可能需要在多处替换它。
2. 将 `<TargetFrameworkVersion>` 属性替换为 `<TargetFramework>netcoreapp3.1</TargetFramework>`。
3. 删除任何 .NET Framework 引用（例如 `<Reference Include="System" />`）。使用 `<CLRSupport>NetCore</CLRSupport>` 时，将自动引用 .NET Core SDK 程序集。
4. 根据需要更新 cpp 文件中的 API 使用情况，以删除 .NET Core 不可使用的 API。由于 C++/CLI 项目通常是非常精简的互操作层，因此通常不需要进行诸多更改。.NET 可移植性分析器可用于识别 C++/CLI 二进制文件使用的不受支持的 .NET API，就像使用纯托管二进制文件一样。

WPF 和 Windows 窗体使用情况

.NET Core C++/CLI 项目可以使用 Windows 窗体和 WPF API。要使用这些 Windows 桌面 API，需要将显式框架引用添加到 UI 库。使用 Windows 桌面 API 的 SDK 样式项目通过使用 `Microsoft.NET.Sdk.WindowsDesktop` SDK 来自动引用必要的框架库。由于 C++/CLI 项目不使用 SDK 样式的项目格式，因此它们在面向 .NET Core 时需要添加显式框架引用。

要使用 Windows 窗体 API，请将此引用添加到 vcxproj 文件：

```
<!-- Reference all of Windows Forms -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App.WindowsForms" />
```

要使用 WPF API，请将此引用添加到 vcxproj 文件：

```
<!-- Reference all of WPF -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App.WPF" />
```

要同时使用 Windows 窗体和 WPF API，请将此引用添加到 vcxproj 文件：

```
<!-- Reference the entirety of the Windows desktop framework:
     Windows Forms, WPF, and the types that provide integration between them -->
<FrameworkReference Include="Microsoft.WindowsDesktop.App" />
```

目前，不能使用 Visual Studio 的引用管理器来添加这些引用。而是改为手动更新项目文件。可通过在 Visual Studio 中卸载项目，然后编辑项目文件来完成此更新。还可以使用其他编辑器，例如 VS Code。

在不使用 MSBuild 的情况下生成

还可以在不使用 MSBuild 的情况下生成 C++/CLI 项目。请按照以下步骤，使用 cl.exe 和 link.exe 直接生成适用于 .NET Core 的 C++/CLI 项目：

1. 编译时，将 `-clr:netcore` 传递给 cl.exe。
2. 引用必要的 .NET Core 引用程序集。
3. 链接时，提供 .NET Core 应用主机目录作为 `LibPath`（以便可以找到 ijwhost.lib）。
4. 将 ijwhost.dll（从 .NET Core 应用主机目录）复制到项目的输出目录。
5. 确保将运行托管代码的应用程序的第一个组件存在 runtimeconfig.json 文件。如果应用程序具有托管入口点，则将自动创建并复制 `runtime.config` 文件。不过，如果应用程序具有本机入口点，则需要为第一个 C++/CLI 库创建 `runtimeconfig.json` 文件以使用 .NET Core 运行时。

已知问题

在处理面向 .NET Core 的 C++/CLI 项目时，需要注意几个已知问题。

- .NET Core C++/CLI 项目中的 WPF 框架引用目前会导致一些有关无法导入符号的无关警告。可以安全忽略这些警告，并且应该尽快解决它们。
- 如果应用程序具有本机入口点，则首次执行托管代码的 C++/CLI 库需要 runtimeconfig.json 文件。此配置文件在 .NET Core 运行时启动时使用。C++/CLI 项目不会在生成时自动创建 `runtimeconfig.json` 文件，因此必须手动生成该文件。如果从托管入口点调用 C++/CLI 库，则 C++/CLI 库不需要 `runtimeconfig.json` 文件（因为入口点程序集将具有一个在启动运行时时使用的该文件）。下面显示了一个简单的示例 `runtimeconfig.json` 文件。有关详细信息，请参阅 GitHub 上的规范。

  ```
  {
        "runtimeOptions": {
           "tfm": "netcoreapp3.1",
           "framework": {
              "name": "Microsoft.NETCore.App",
              "version": "3.1.0"
           }
        }
  }
  ```

- 在 Windows 7 上，当入口应用程序为原生时，加载 .NET Core C++/CLI 程序集可能会表现出失败的行为。这一失败行为是由于 Windows 7 加载程序不遵循 C++/CLI 程序集的非 `mscoree.dll` 入口点。建议的操作方式是将入口应用程序转换为托管代码。Windows 7 上的所有情况都尤其不支持涉及线程本地存储 (TLS) 的场景。