Contents

- 电子邮件
- 文件系统帮助程序
- 手电筒
- 地理编码
- 地理位置
- 陀螺仪
- 启动器
- 磁力计
- 主线程
- 地图
- 打开浏览器
- 方向传感器
- 电话拨号程序
- 幂
- 首选项
- 屏幕锁定
- 安全存储
- SMS
- 文本到语音转换
- 版本跟踪
- 振动
- 疑难解答

数据和云服务

- 了解示例
- 使用 Web 服务
  - ASMX
  - WCF
  - REST
  - Azure 移动应用
- 对 Web 服务的访问进行身份验证
  - REST
  - OAuth

Android 生成错误:“LinkAssemblies”任务意外失败

为何 Xamarin.Forms.Maps Android 项目失败，且随附“COMPILETODALVIK:意外顶级错误”？

发行说明

示例

使用 Xamarin.Forms 书籍创建移动应用

企业应用程序模式电子书

Xamarin.Forms 中的 SkiaSharp Graphics

Xamarin.Forms 入门

2018/10/26 • • Edit Online

Xamarin.Forms 是一个跨平台 UI 工具包，允许开发人员有效创建可跨 iOS、Android、通用 Windows 平台应用共享的本机用户界面布局。此系列介绍 Xamarin.Forms 开发的基础知识，并涵盖如何构建多平台和多屏幕的应用程序。

有关适用于跨平台开发的安装和设置实践的概述，请参阅 Xamarin.Forms 要求和安装。

生成首个应用

要求

概述了 Xamarin.Forms 开发应用的平台要求，以及在 Visual Studio for Mac 和 Visual Studio 中使用 Xamarin.Forms 进行开发的最低系统要求。

生成首个应用

观看视频，然后按照分布说明生成并测试首个 Xamarin.Forms 应用。

Hello，Xamarin.Forms

本指南介绍如何使用 Visual Studio for Mac 或 Visual Studio 开发 Xamarin.Forms 应用程序。主题涵盖生成和部署 Xamarin.Forms 应用程序所需的工具、概念和步骤。

了解 Xamarin.Forms 多屏显示

本指南将通过引入第二页的导航，扩展之前创建的应用程序。主题涵盖数据绑定和执行导航。

Xamarin.Forms 简介

本文介绍关于使用 Xamarin.Forms 开发应用程序的关键概念，包括视图和布局、ListView 控件、数据绑定和导航。

Xamarin University 入门

通过 Xamarin University 使用 Xamarin for Visual Studio 生成第一个 Xamarin.Forms 应用

相关链接

- 免费自学教程（视频）
- Xamarin 入门（视频）

Xamarin.Forms 要求

2018/11/2 • • Edit Online

Xamarin.Forms 平台和开发系统要求。

请参阅安装一文中跨平台应用的安装和设置概述。

目标平台

可对以下操作系统编写 Xamarin.Forms 应用程序：

- iOS 8 或更高版本
- Android 4.4 (API 19) 或更高版本（详细信息）
- Windows 10 通用 Windows 平台（详细信息）

假定开发人员熟悉 .NET Standard 和共享项目。

其他平台支持

这些平台状态可在 Xamarin.Forms GitHub 上找到：

- Samsung Tizen
- macOS
- GTK#
- WPF

早期版本平台

使用 Xamarin.Forms 3.0 时，不支持这些平台：

- *Windows 8.1 / Windows Phone 8.1 WinRT*
- *Windows Phone 8 Silverlight*

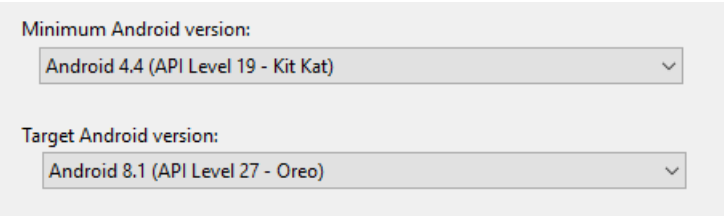
Android

应安装最新的 Android SDK 工具和 Android API 平台。可使用 Android SDK 管理器更新到最新版本。

此外，Android 项目的目标/编译版本必须设置为“使用最新安装的平台”。但是，最低版本可设置为 API 19，因此可继续支持使用 Android 4.4（以及更高版本）的设备。在“项目选项”中可设置这些值：

- Visual Studio
- Visual Studio for Mac

“项目选项”>“应用程序”>“应用程序属性”



开发系统要求

可在 macOS 和 Windows 上开发 Xamarin.Forms 应用。但是，生成 Windows 版本的应用需要使用 Windows 和 Visual Studio。

Mac 系统要求

可使用 Visual Studio for Mac 在 OS X El Capitan (10.11) 或更高版本上开发 Xamarin.Forms 应用。若要开发 iOS 应用，建议至少安装 iOS 10 SDK 和 Xcode 8。

> **NOTE**
> 不能在 macOS 上开发 Windows 应用。

Windows 系统要求

任何支持 Xamarin 开发的 Windows 安装上都可生成适用于 iOS 和 Android 的 Xamarin.Forms 应用。这要求 Visual Studio 2017 或更高版本在 Windows 7 或更高版本上运行。iOS 开发需要使用联网的 Mac。

通用 Windows 平台 (UWP)

为 UWP 开发 Xamarin.Forms 应用需要：

- Windows 10（建议使用 Fall Creators Update）
- Visual Studio 2017
- Windows 10 SDK

UWP 项目包含在 Visual Studio 2017 中所创建的 Xamarin.Forms 解决方案中，而不包含在 Visual Studio for Mac 中创建的解决方案中。可以在现有的 Xamarin.Forms 解决方案中随时添加通用 Windows 平台 (UWP) 应用。

生成第一个 Xamarin.Forms 应用

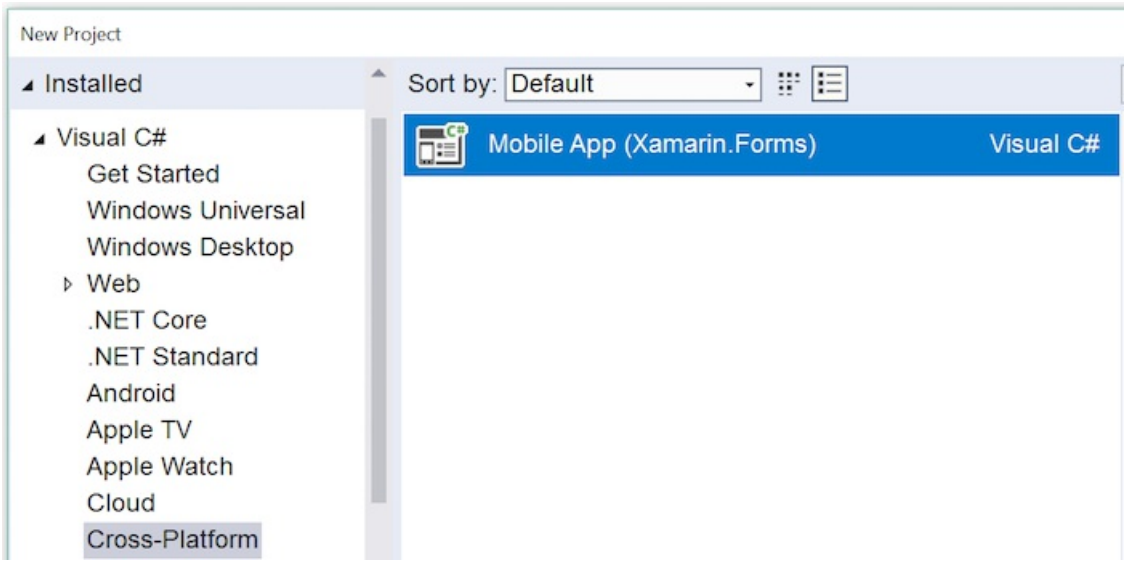
2018/11/2 • • Edit Online

观看此视频，然后按照视频中的步骤使用 Xamarin.Forms 创建第一个移动应用。

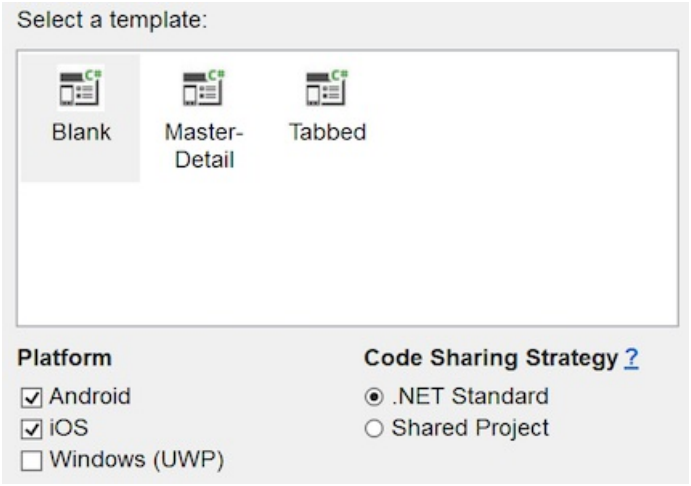
Windows 分步说明

请按照以下步骤以及上面的视频操作：

1. 选择“文件”>“新建”>“项目...”或按“创建新项目...”按钮，然后选择“Visual C#”>“跨平台”>“移动应用(Xamarin.Forms)”：



2. 请确保选中“Android”和“iOS”且勾选了“.NET Standard”代码共享策略：



3. 等到 NuGet 包还原（状态栏中将出现“还原已完成”消息）。

4. 按调试按钮（或“调试”>“开始调试”菜单项）启动 Android Emulator。

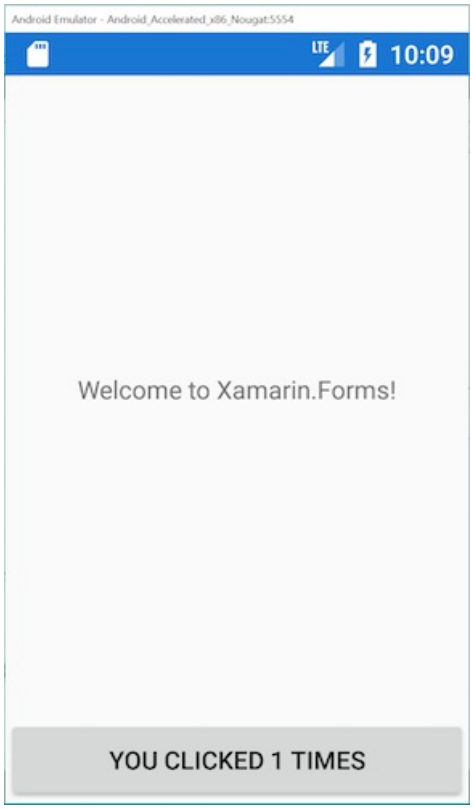
5. 编辑 MainPage.xaml，在 `</StackPanel>` 结束之前添加此 XAML：

```
<Button Text="Click Me" Clicked="Button_Clicked" />
```

6. 编辑 MainPage.xaml.cs，将此代码添加到类的末尾：

```
int count = 0;
void Button_Clicked(object sender, System.EventArgs e)
{
    count++;
    ((Button)sender).Text = $"You clicked {count} times.";
}
```

7. 调试 Android 上的应用：

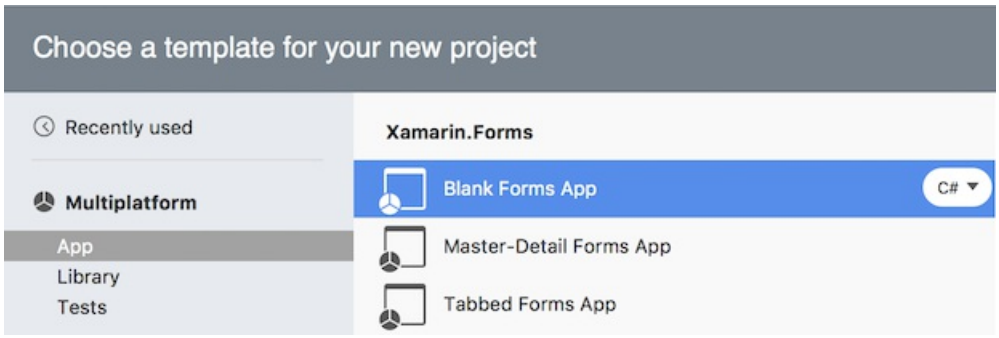


> **TIP**
> 可使用联网的 Mac 计算机从 Visual Studio 生成和调试 iOS 应用。有关详细信息，请参阅安装说明。

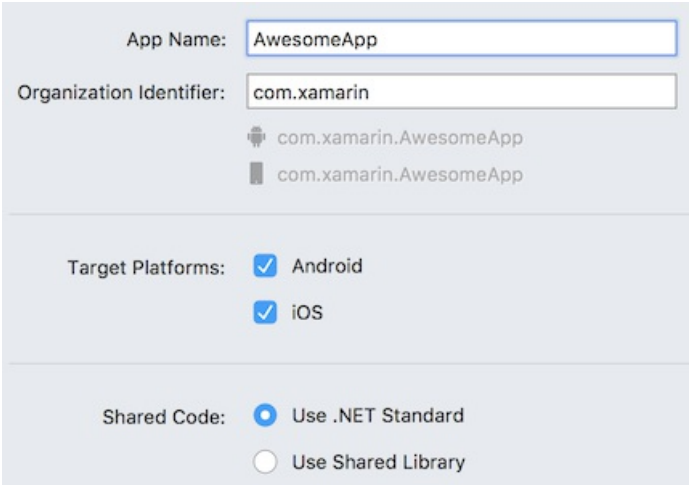
Mac 分步说明

请按照以下步骤以及上面的视频操作：

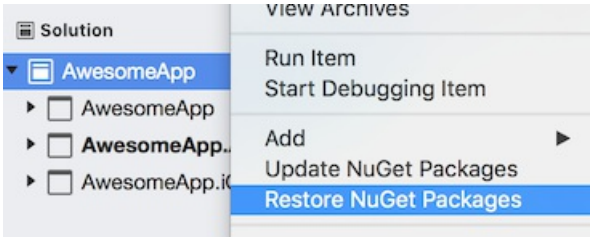
1. 选择“文件”>“新建解决方案...”或按“新建项目...”按钮，然后选择“多平台”>“应用”>“空白窗体应用”：



2. 请确保选中“Android”和“iOS”且勾选了“.NET Standard”代码共享策略：



3. 右键单击解决方案，还原 NuGet 包：



4. 按调试按钮（或“运行”>“开始调试”）启动 Android Emulator。

5. 编辑 MainPage.xaml，在 `</StackPanel>` 结束之前添加此 XAML：

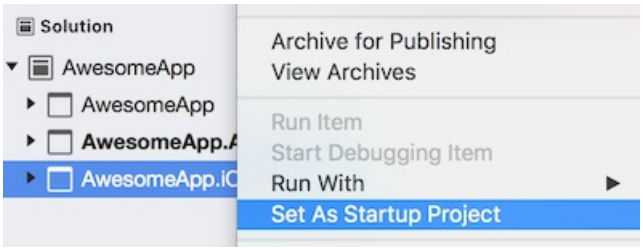
```
<Button Text="Click Me" Clicked="Handle_Clicked" />
```

6. 编辑 MainPage.xaml.cs，将此代码添加到类的末尾：

```
int count = 0;
void Handle_Clicked(object sender, System.EventArgs e)
{
    count++;
    ((Button)sender).Text = $"You clicked {count} times.";
}
```

7. 调试 Android 上的应用：

8. 右键单击，将 iOS 设置为“启动项目”：



9. 调试 iOS 上的应用：

可从示例库下载或在 GitHub 上查看完整代码。

后续步骤

- 了解 Xamarin.Forms – 生成功能更强大的应用。
- 了解 Xamarin.Forms 多屏显示 – 生成在两个屏幕之间导航的应用。
- Xamarin.Forms 示例 – 下载并运行代码示例和示例应用。
- Xamarin.Forms 简介– Xamarin.Forms 的初学者指南。
- 创建移动应用电子书 – 深入介绍 Xamarin.Forms 开发的章节（PDF 格式），包括数百个其他示例。

Hello, Xamarin.Forms

2018/10/19 • • Edit Online

本指南介绍如何使用 Visual Studio for Mac 或 Visual Studio 开发 Xamarin.Forms 应用程序，以及使用 Xamarin.Forms 开发应用程序的基础知识。主题涵盖生成和部署 Xamarin.Forms 应用程序所需的工具、概念和步骤。

首先，查看 Xamarin.Forms 系统要求。

第 1 部分：快速入门

本指南的第一部分演示如何创建一个应用程序，它将用户输入的字母数字电话号码转换为数字电话号码，然后呼叫该号码。

第 2 部分：深度分析

本指南的第二部分对已生成的内容进行回顾，以深入了解有关 Xamarin.Forms 应用程序工作原理的基础知识。

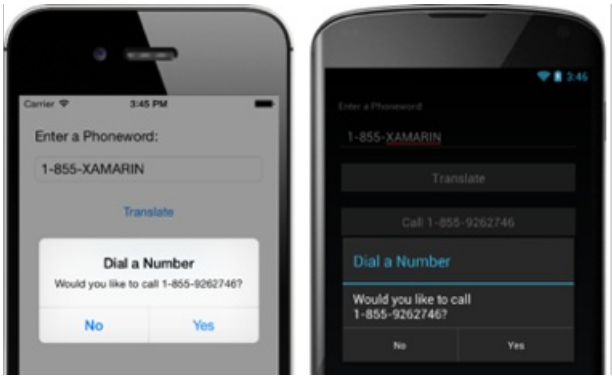
相关链接

- Xamarin.Forms 简介
- 在 Visual Studio 中进行调试
- Visual Studio for Mac 方案 - 调试
- 免费自学教程（视频）
- Xamarin 入门（视频）

Xamarin.Forms 快速入门

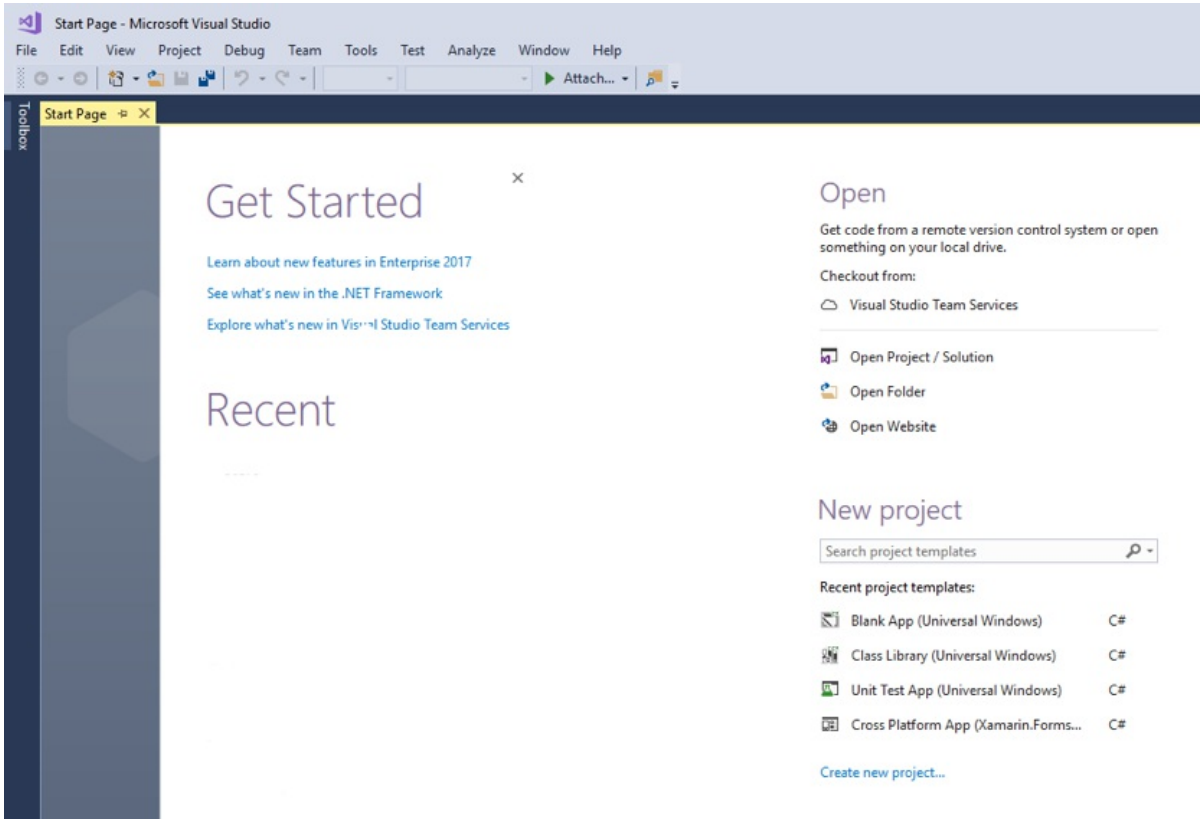
2018/11/2 • • Edit Online

本演练介绍如何创建一个应用程序，它将字母数字电话号码(由用户输入)转换为数字电话号码，然后呼叫该号码。最终的应用程序如下所示：

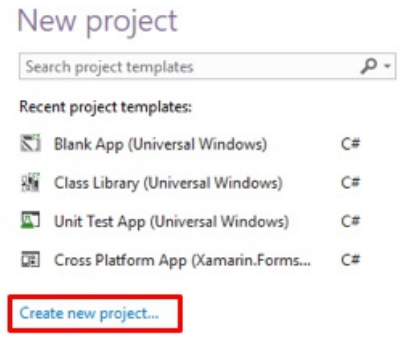


Visual Studio 入门

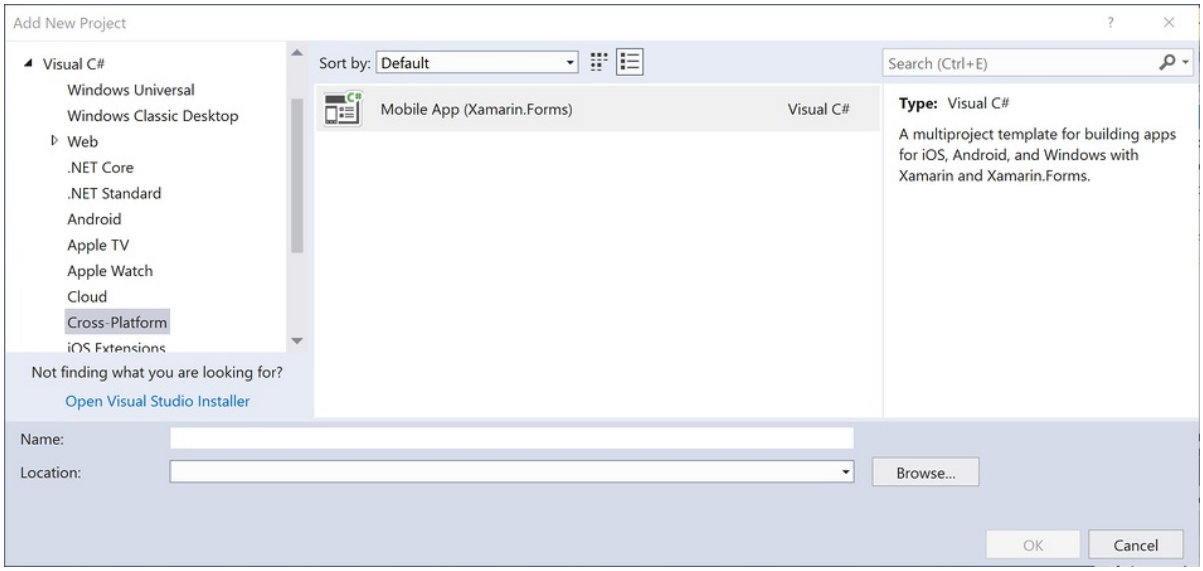
1. 在"开始"屏幕中，启动 Visual Studio。这会打开起始页：



2. 在 Visual Studio 中，单击"创建新项目..."以创建新项目：

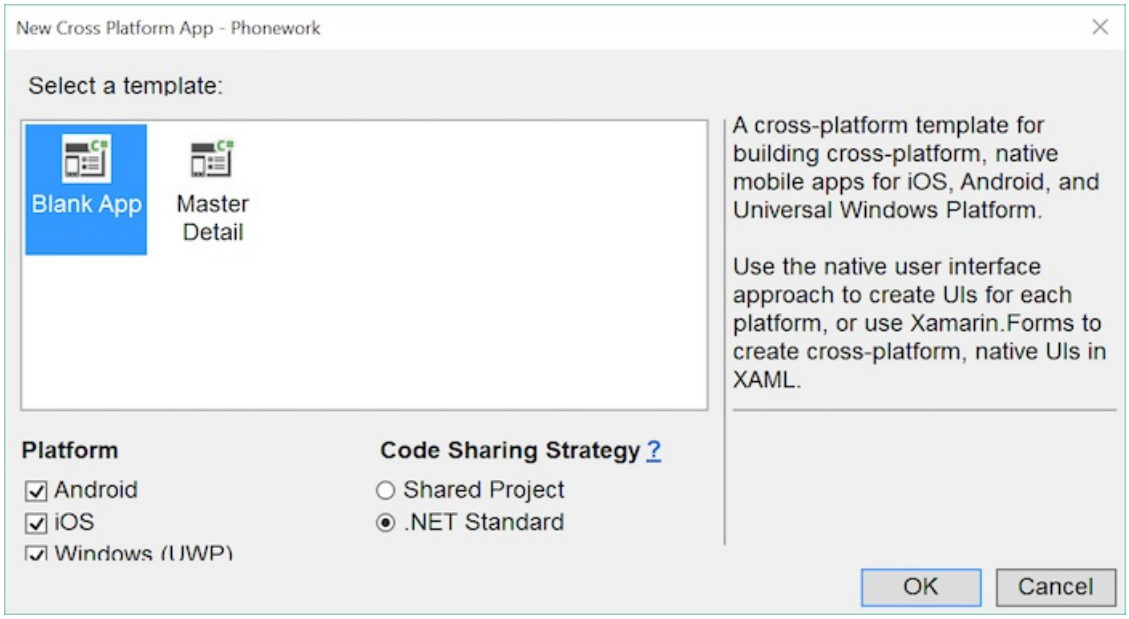


3. 在"新建项目"对话框中，单击"跨平台"，选择"移动应用(Xamarin.Forms)"模板，将"名称"设为"Phoneword"，为项目选择合适的位置，然后单击"确定"按钮：

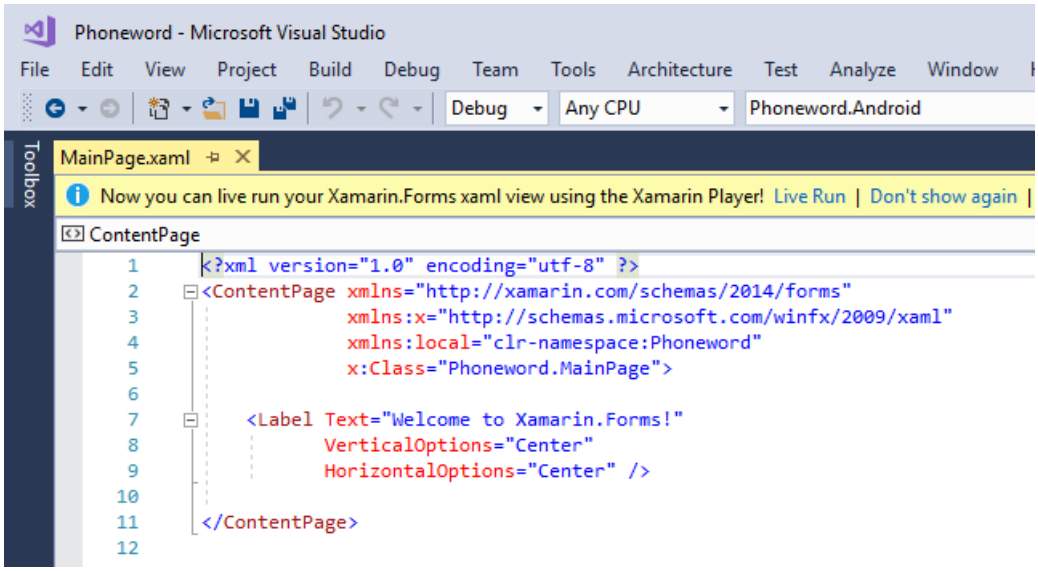


> **NOTE**
>
> 本快速入门中的 C# 和 XAML 片段要求将解决方案命名为"Phoneword"。将这些指令中的代码复制到项目中时，使用不同的解决方案名称将导致大量生成错误。

4. 在"新的跨平台应用"对话框中，单击"空白应用"，选择".NET Standard"作为代码共享策略，然后单击"确定"按钮：



5. 在“解决方案资源管理器”的“Phoneword”项目中，双击 **MainPage.xaml** 将其打开：

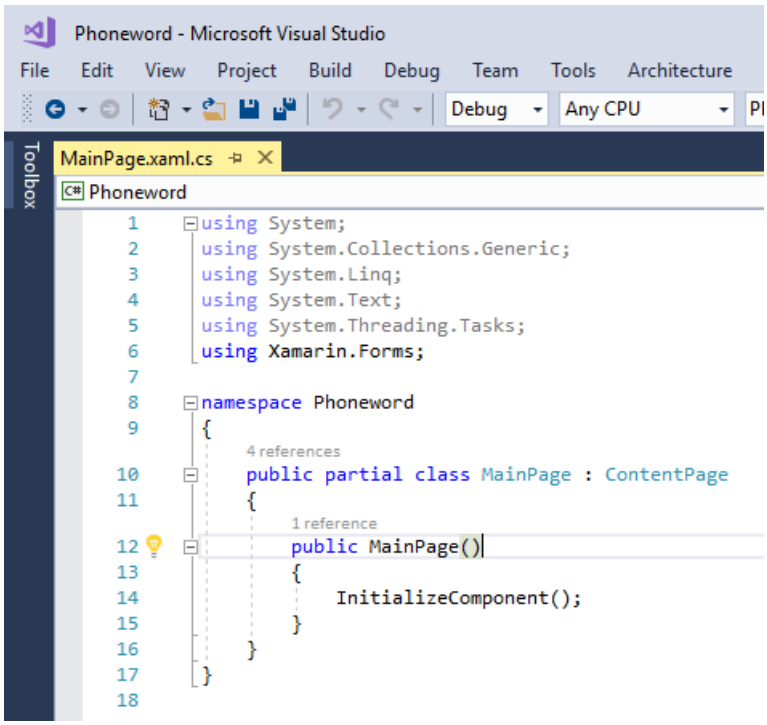


6. 在“MainPage.xaml”中，删除所有模板代码并将其替换为以下代码。此代码以声明方式定义页面上的用户界面：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="Phoneword.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
      <Label Text="Enter a Phoneword:" />
      <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
      <Button Text="Translate" Clicked="OnTranslate" />
      <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>
```

按 **Ctrl+S**，保存对 **MainPage.xaml** 所做的更改，然后关闭文件。

7. 在“解决方案资源管理器”中，展开“MainPage.xaml”，然后双击 **MainPage.xaml.cs** 将其打开：



8. 在“MainPage.xaml.cs”中，删除所有模板代码并将其替换为以下代码。如果分别在用户界面中单击“翻译”和“调用”按钮，作为响应，将分别执行 `OnTranslate` 和 `OnCall` 方法：

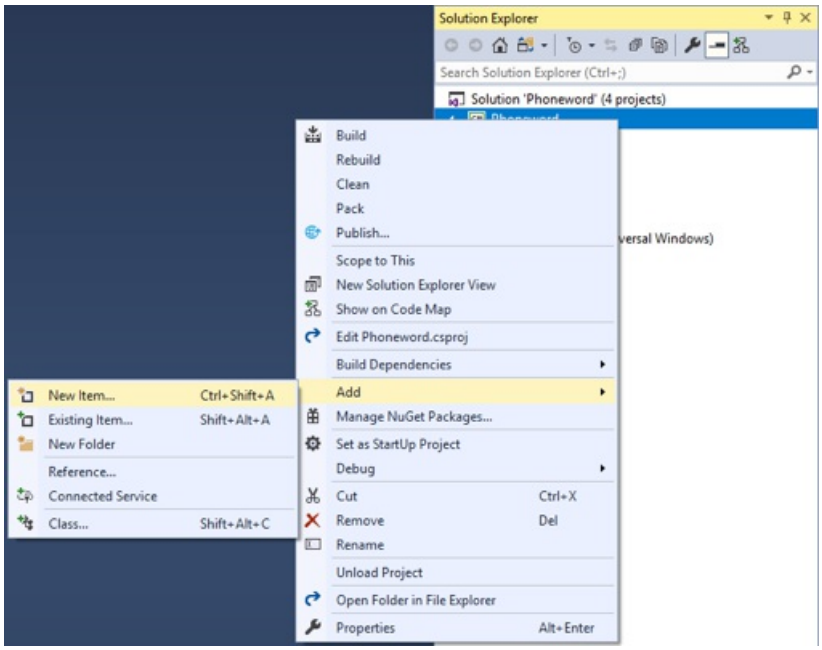
```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        string translatedNumber;

        public MainPage ()
        {
            InitializeComponent ();
        }

        void OnTranslate (object sender, EventArgs e)
        {
            translatedNumber = PhonewordTranslator.ToNumber (phoneNumberText.Text);
            if (!string.IsNullOrWhiteSpace (translatedNumber)) {
                callButton.IsEnabled = true;
                callButton.Text = "Call " + translatedNumber;
            } else {
                callButton.IsEnabled = false;
                callButton.Text = "Call";
            }
        }

        async void OnCall (object sender, EventArgs e)
        {
            if (await this.DisplayAlert (
                    "Dial a Number",
                    "Would you like to call " + translatedNumber + "?",
                    "Yes",
                    "No")) {
                var dialer = DependencyService.Get<IDialer> ();
                if (dialer != null)
                    dialer.Dial (translatedNumber);
            }
        }
    }
}
```

NOTE

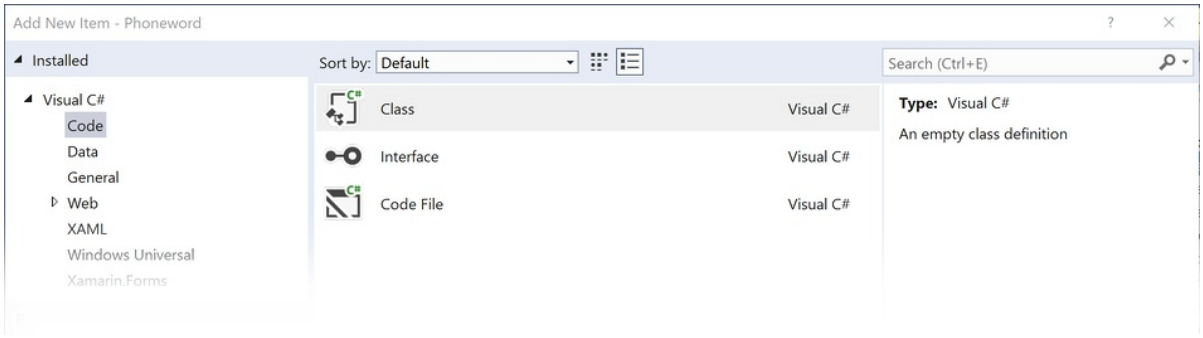
尝试在此时构建应用程序将导致稍后要修复的错误。

按 **Ctrl+S**，保存对 **MainPage.xaml.cs** 所做的更改，然后关闭文件。

9. 在“解决方案资源管理器”中，右键单击“Phoneword”项目，然后选择“添加”>“新建项...”：



10. 在"添加新项"对话框中，选择"Visual C#">"代码">"类"，将新文件命名为 **PhoneTranslator**，然后单击"添加"按钮：

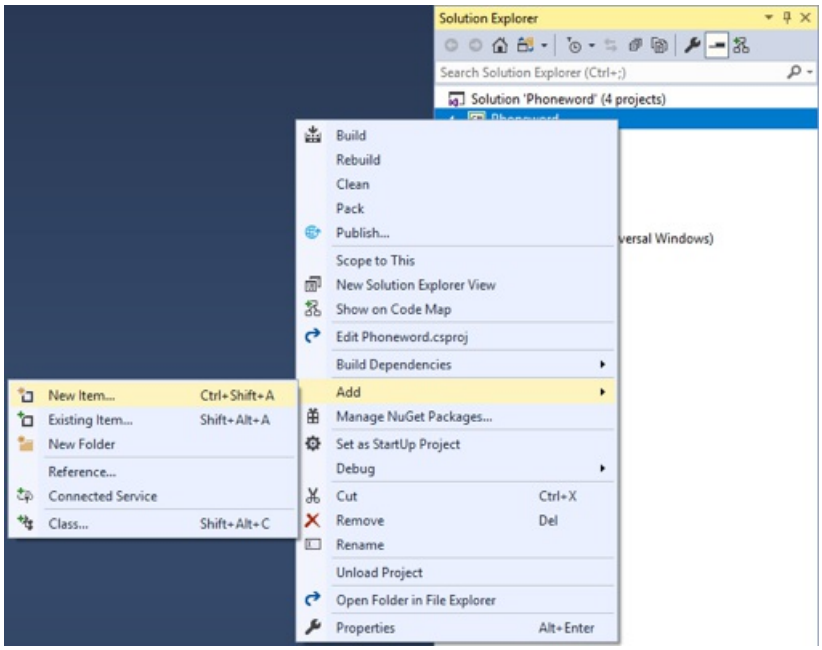


11. 在"PhoneTranslator.cs"中，删除所有模板代码并将其替换为以下代码。此代码会将手机词翻译为电话号码：

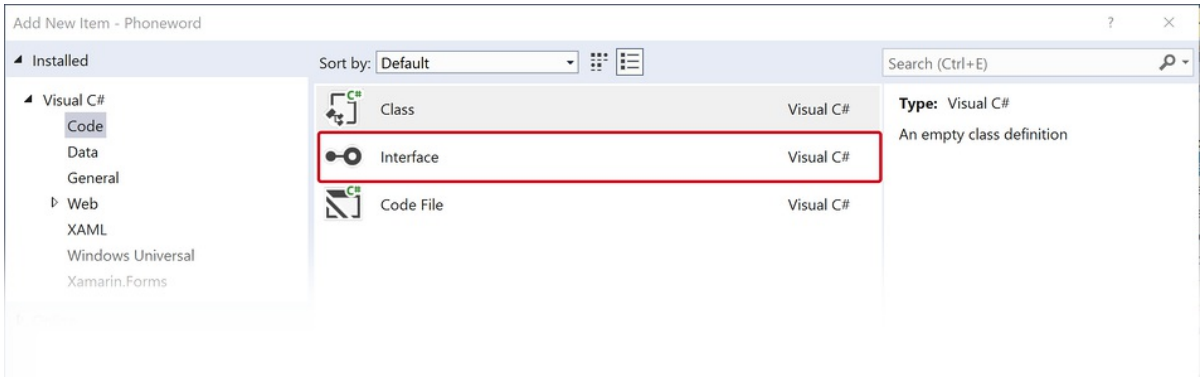
```
using System.Text;

namespace Phoneword
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return null;

            raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                    newNumber.Append(c);
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                    // Bad character?
                    else
                        return null;
                }
            }
            return newNumber.ToString();
        }

        static bool Contains(this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }

        static readonly string[] digits = {
            "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ"
        };

        static int? TranslateToNumber(char c)
        {
            for (int i = 0; i < digits.Length; i++)
            {
                if (digits[i].Contains(c))
                    return 2 + i;
            }
            return null;
        }
    }
}
```

通过按 **Ctrl+S**，保存对 **PhoneTranslator.cs** 所做的更改，然后关闭文件。

12. 在“解决方案资源管理器”中，右键单击“Phoneword”项目，然后选择“添加”>“新建项...”：



13. 在“添加新项”对话框中，选择“Visual C#”>“代码”>“界面”，将新文件命名为 **IDialer**，然后单击“添加”按钮：



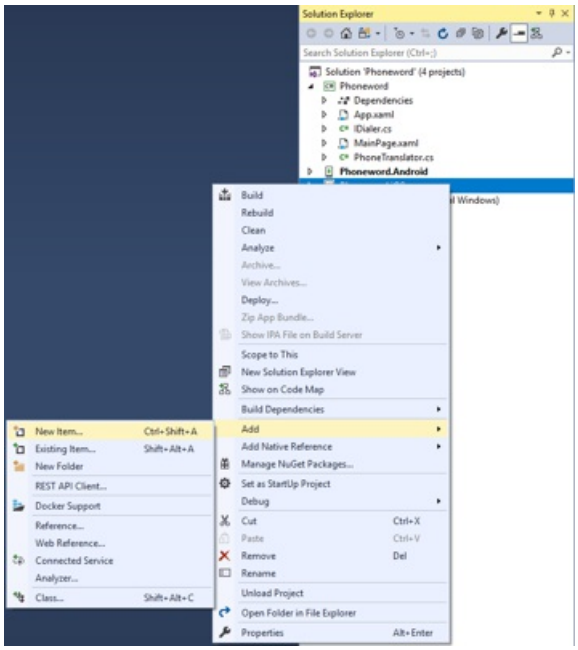
14. 在“IDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将定义 `Dial` 方法，必须在每个平台上实现此方法，才可拨打翻译后的电话号码：

```
namespace Phoneword
{
    public interface IDialer
    {
        bool Dial(string number);
    }
}
```

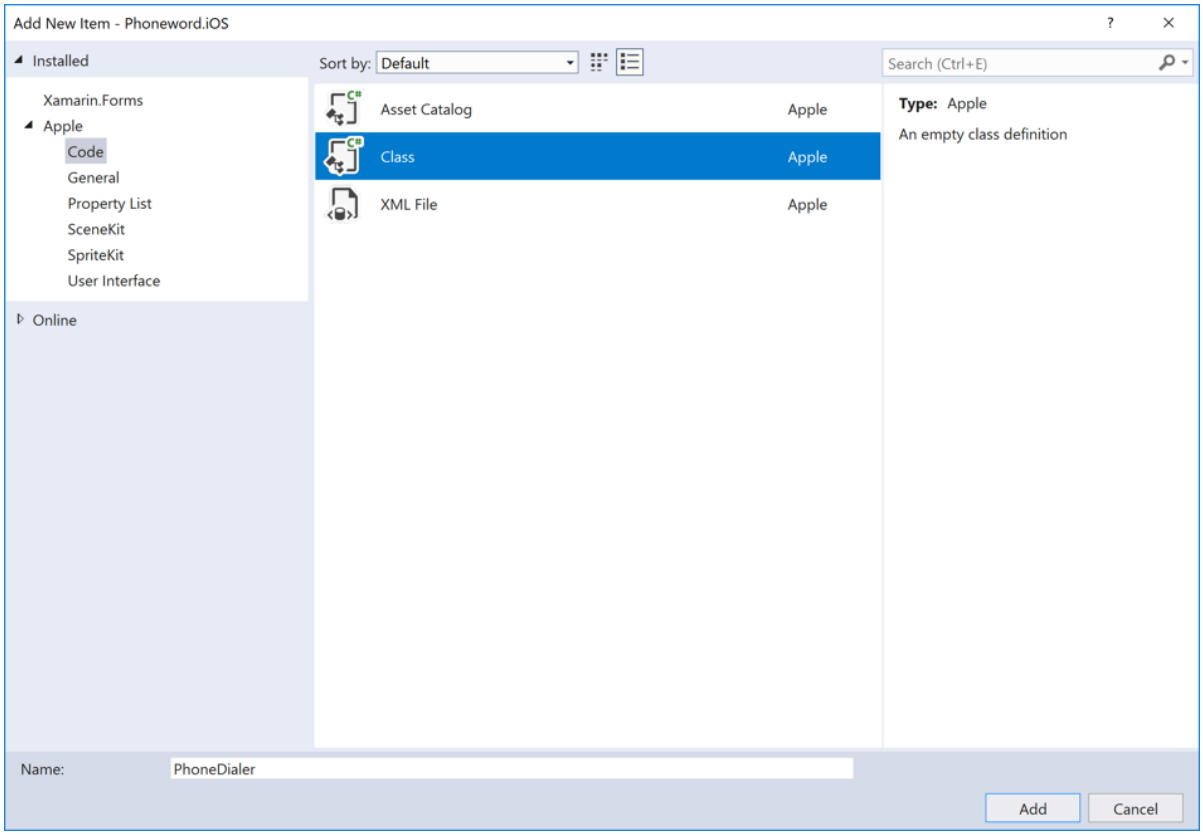
通过按 **Ctrl+S**，保存对 **IDialer.cs** 所做的更改，然后关闭文件。

> **NOTE**
>
> 此时完成了应用程序的常用代码。此时，可将特定于平台的电话拨号程序实现为 DependencyService。

15. 在“解决方案资源管理器”中，右键单击“Phoneword.iOS”项目，然后选择“添加”>“新建项...”：



16. 在“添加新项”对话框中，选择“Apple”>“代码”>“类”，将新文件命名为 **PhoneDialer**，然后单击“添加”按钮：

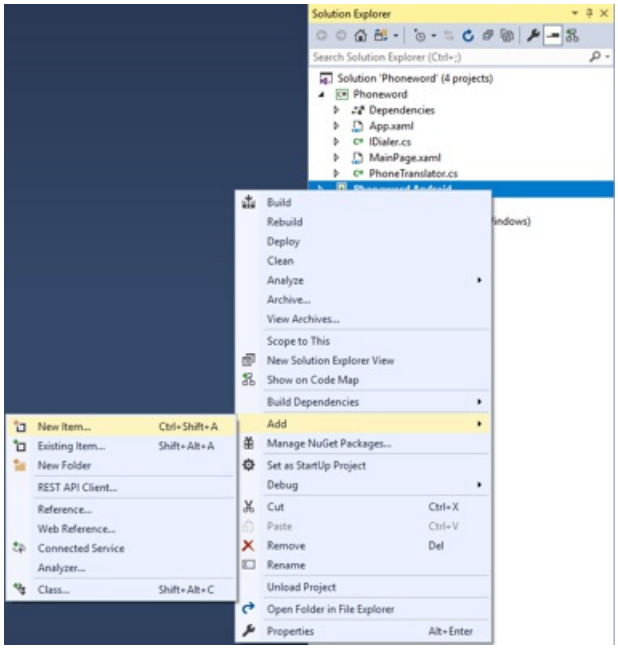


17. 在“PhoneDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将创建 `Dial` 方法，此方法将在 iOS 平台上用于拨打翻译后的电话号码：

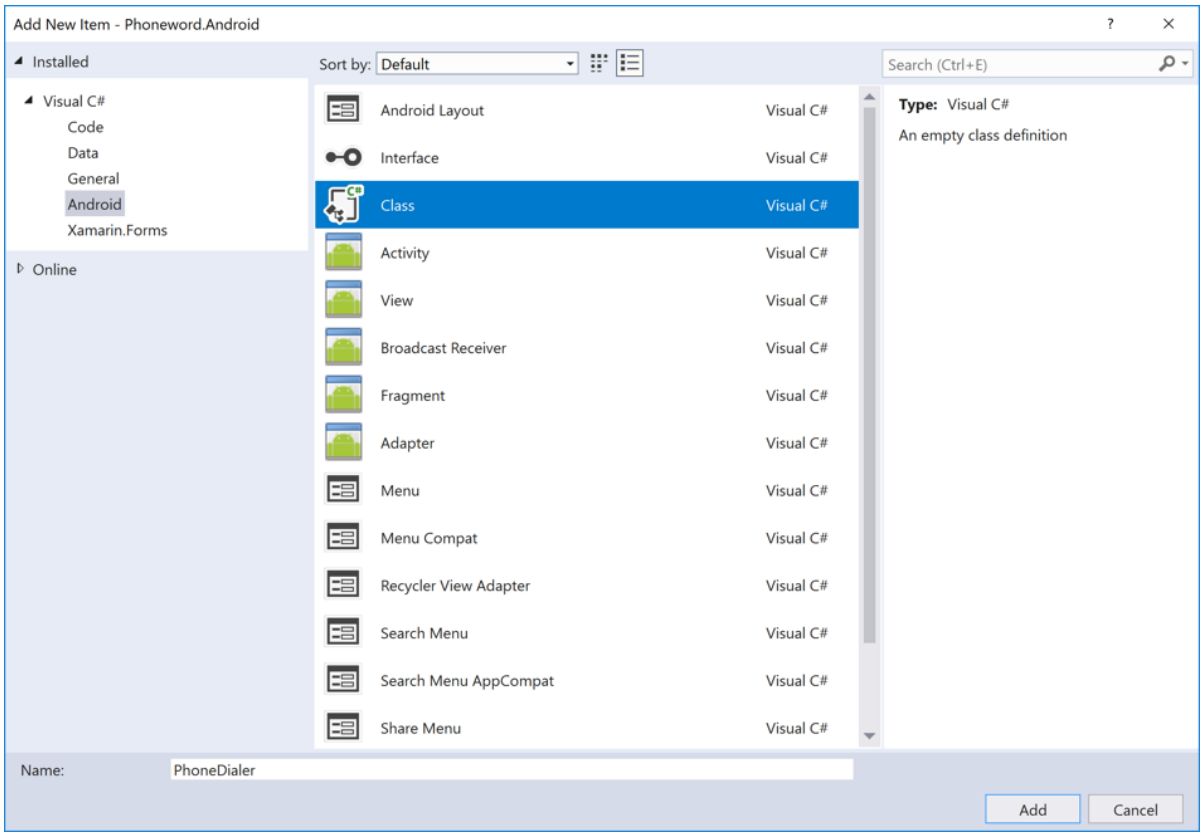
```
using Foundation;
using Phoneword.iOS;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.iOS
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            return UIApplication.SharedApplication.OpenUrl (
                new NSUrl ("tel:" + number));
        }
    }
}
```

通过按 **Ctrl+S**，保存对 **PhoneDialer.cs** 所做的更改，然后关闭文件。

18. 在“解决方案资源管理器”中，右键单击“Phoneword.Android”项目，然后选择“添加”>“新建项...”：



19. 在“添加新项”对话框中，选择“Visual C#”>“Android”>“类”，将新文件命名为 **PhoneDialer**，然后单击“添加”按钮：



20. 在“PhoneDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将创建 `Dial` 方法，此方法将在 Android 平台上用于拨打翻译后的电话号码：

```
using Android.Content;
using Android.Telephony;
using Phoneword.Droid;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.Droid
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            var context = MainActivity.Instance;
            if (context == null)
                return false;

            var intent = new Intent (Intent.ActionDial);
            intent.SetData (Uri.Parse ("tel:" + number));

            if (IsIntentAvailable (context, intent)) {
                context.StartActivity (intent);
                return true;
            }

            return false;
        }

        public static bool IsIntentAvailable(Context context, Intent intent)
        {
            var packageManager = context.PackageManager;

            var list = packageManager.QueryIntentServices (intent, 0)
                .Union (packageManager.QueryIntentActivities (intent, 0));

            if (list.Any ())
                return true;

            var manager = TelephonyManager.FromContext (context);
            return manager.PhoneType != PhoneType.None;
        }
    }
}
```

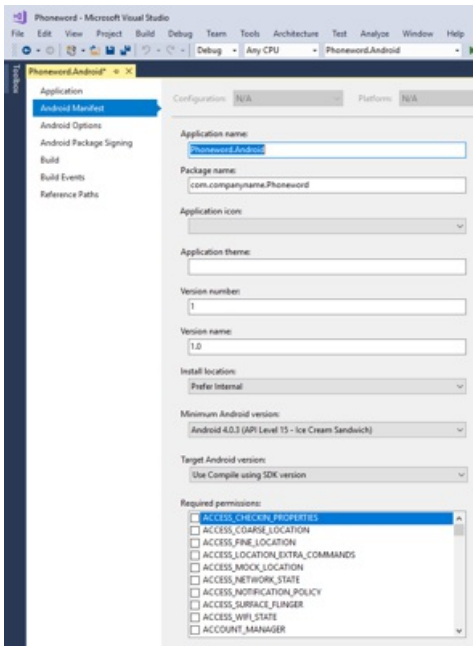
请注意，若要更好地理解此代码，需要使用最新 Android API。通过按 **Ctrl+S**，保存对 **PhoneDialer.cs** 所做的更改，然后关闭文件。

21. 在“解决方案资源管理器”的“Phoneword.Android”项目中，双击“MainActivity.cs”将其打开，然后删除所有模板代码并将其替换成下列代码：

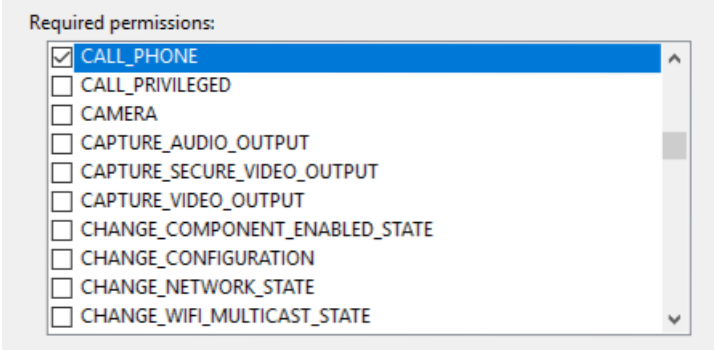
```
using Android.App;
using Android.Content.PM;
using Android.OS;

namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword", Icon = "@mipmap/icon", Theme = "@style/MainTheme", MainLauncher =
true,
              ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}
```

按 Ctrl+S，保存对 MainActivity.cs 所做的更改，然后关闭文件。

22. 在“解决方案资源管理器”的“Phoneword.Android”项目中，双击“属性”，然后选择“Android 清单”选项卡：



23. 在“所需的权限”部分中，启用“CALL_PHONE”权限。这将向应用程序授予进行电话呼叫的权限：



通过按 **Ctrl+S**，保存对清单所做的更改，然后关闭文件。

24. 右键单击 Android 应用程序项目并选择“设为启动项目”。

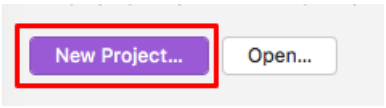
25. 使用“绿色箭头”工具栏按钮运行 Android 应用或在菜单中选择“调试”>“开始调试”。

> **WARNING**
> 所有模拟器都不支持电话呼叫，因此该功能可能无效。

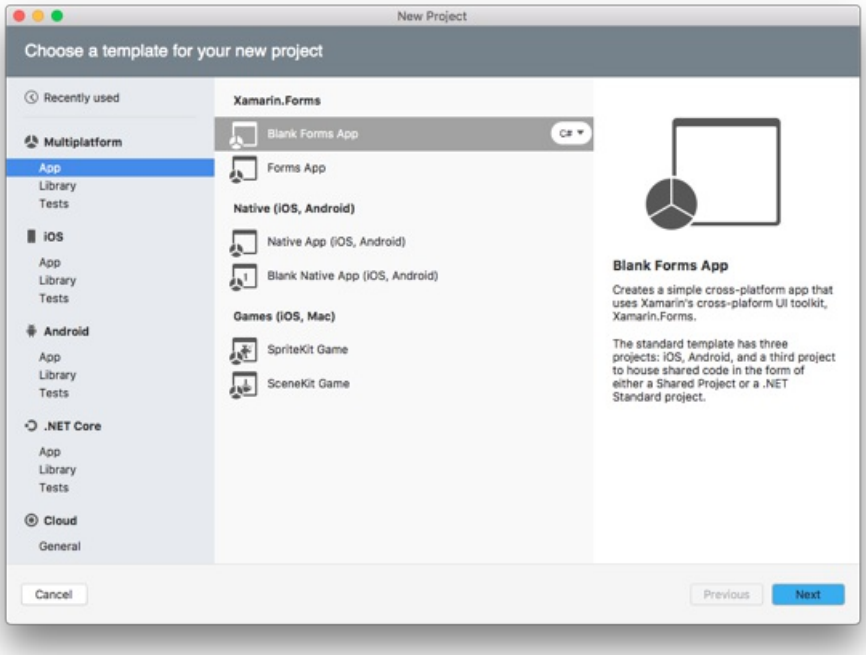
26. 如果拥有 iOS 设备并符合 Xamarin.Forms 开发的 Mac 系统要求，请使用类似技术将应用部署到 iOS 设备。或者，将应用部署到 iOS 远程模拟器。

Visual Studio for Mac 入门

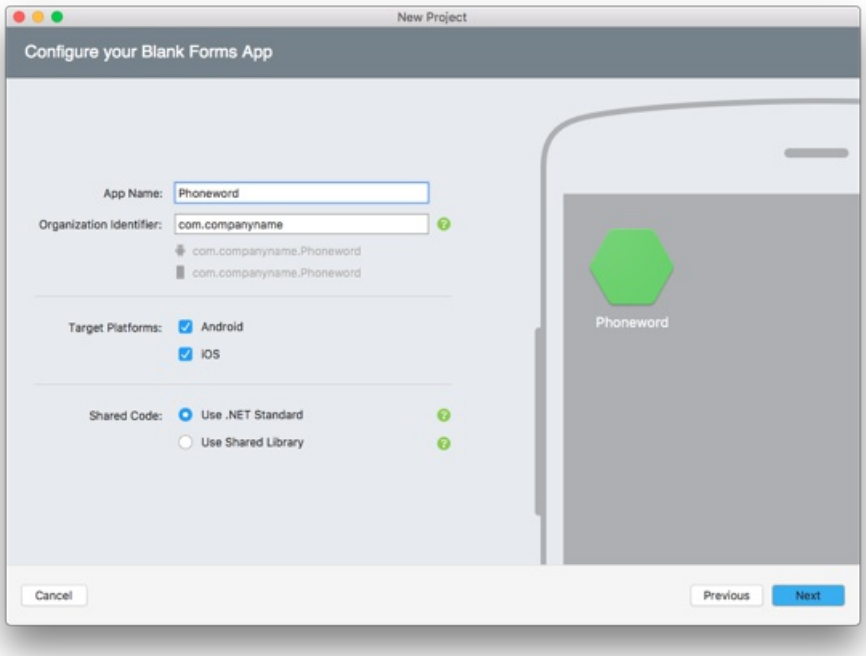
1. 启动 Visual Studio for Mac，然后在起始页上单击“新建项目...”创建新项目：



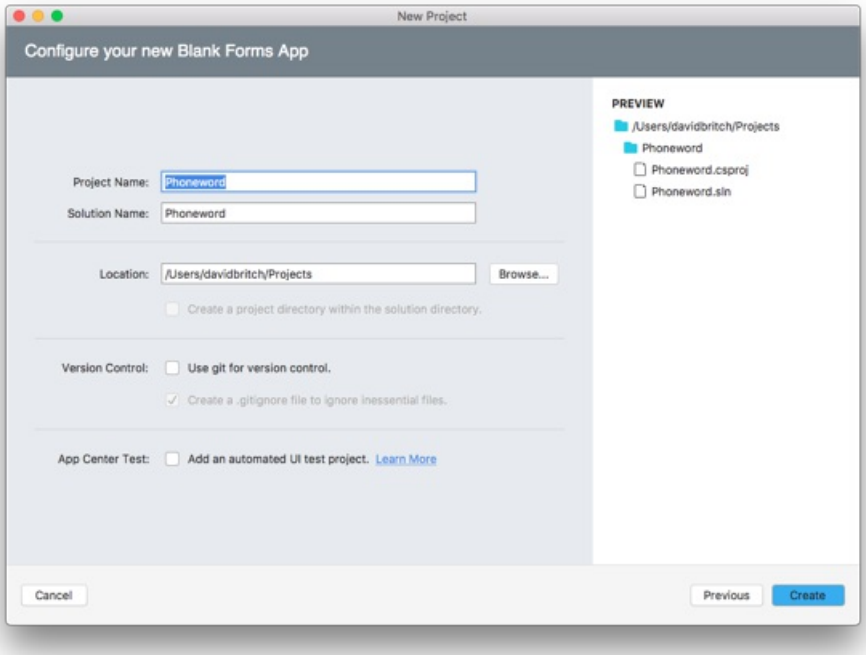
2. 在“为新项目选择一个模板”对话框中，单击“多平台”>“应用”，选择“空白窗体应用”模板，然后单击“下一步”按钮：



3. 在“配置空白表单应用”对话框中，将新应用命名为“Phoneword”，确保选中“使用 .NET Standard”单选按钮，然后单击“下一步”按钮：

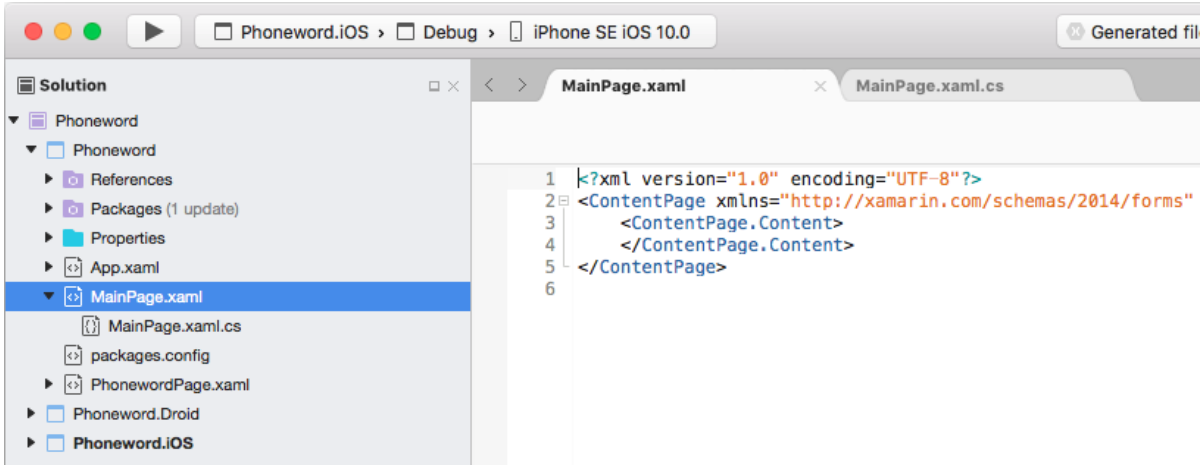


4. 在"配置新空白表单应用"对话框中，将"解决方案"和"项目"名称保留设置为"Phoneword"，为项目选择合适的位置，然后单击"创建"按钮创建项目：



> **NOTE**
> 本快速入门中的 C# 和 XAML 片段要求将解决方案命名为"Phoneword"。将这些指令中的代码复制到项目中时，使用不同的解决方案名称将导致大量生成错误。

5. 在"Solution Pad"中，双击 **MainPage.xaml**将其打开：

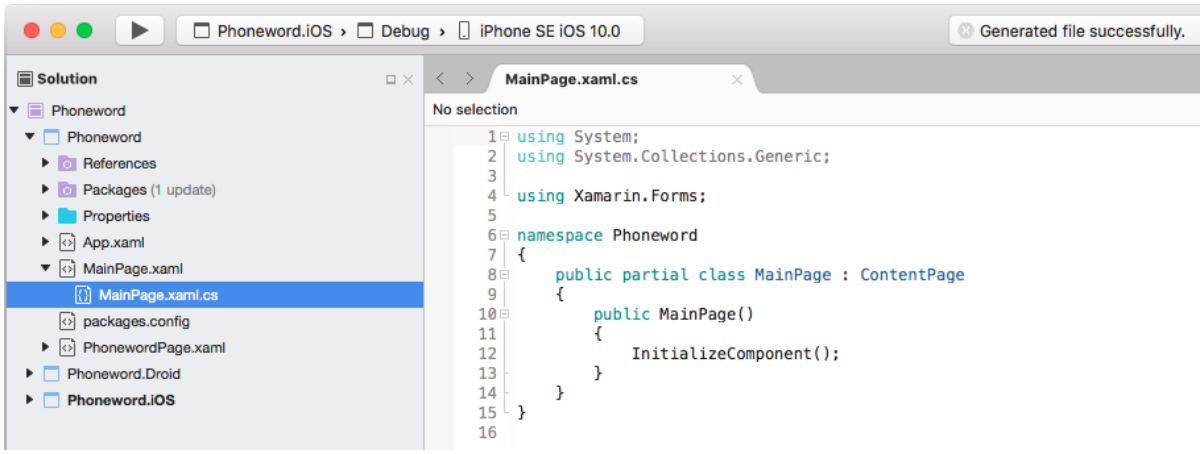


6. 在“MainPage.xaml”中，删除所有模板代码并将其替换为以下代码。此代码以声明方式定义页面上的用户界面：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="Phoneword.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
      <Label Text="Enter a Phoneword:" />
      <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
      <Button Text="Translate" Clicked="OnTranslate" />
      <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
    </StackLayout>
</ContentPage>
```

通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **MainPage.xaml** 所做的更改，然后关闭文件。

7. 在“Solution Pad”中，双击 **MainPage.xaml.cs** 将其打开：



8. 在“MainPage.xaml.cs”中，删除所有模板代码并将其替换为以下代码。如果分别在用户界面上单击“翻译”和“调用”按钮，作为响应，将分别执行 `OnTranslate` 和 `OnCall` 方法：

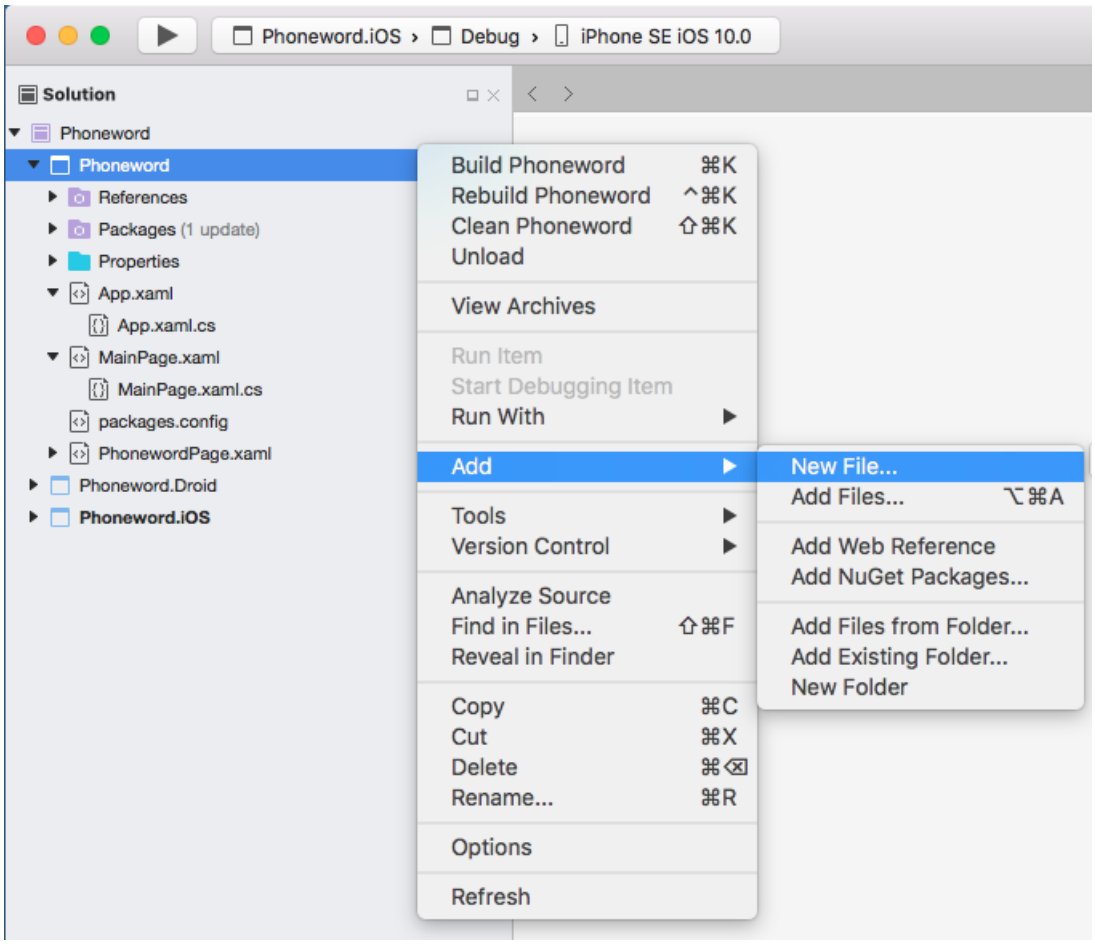
```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        string translatedNumber;

        public MainPage ()
        {
            InitializeComponent ();
        }

        void OnTranslate (object sender, EventArgs e)
        {
            translatedNumber = PhonewordTranslator.ToNumber (phoneNumberText.Text);
            if (!string.IsNullOrWhiteSpace (translatedNumber)) {
                callButton.IsEnabled = true;
                callButton.Text = "Call " + translatedNumber;
            } else {
                callButton.IsEnabled = false;
                callButton.Text = "Call";
            }
        }

        async void OnCall (object sender, EventArgs e)
        {
            if (await this.DisplayAlert (
                    "Dial a Number",
                    "Would you like to call " + translatedNumber + "?",
                    "Yes",
                    "No")) {
                var dialer = DependencyService.Get<IDialer> ();
                if (dialer != null)
                    dialer.Dial (translatedNumber);
            }
        }
    }
}
```

NOTE

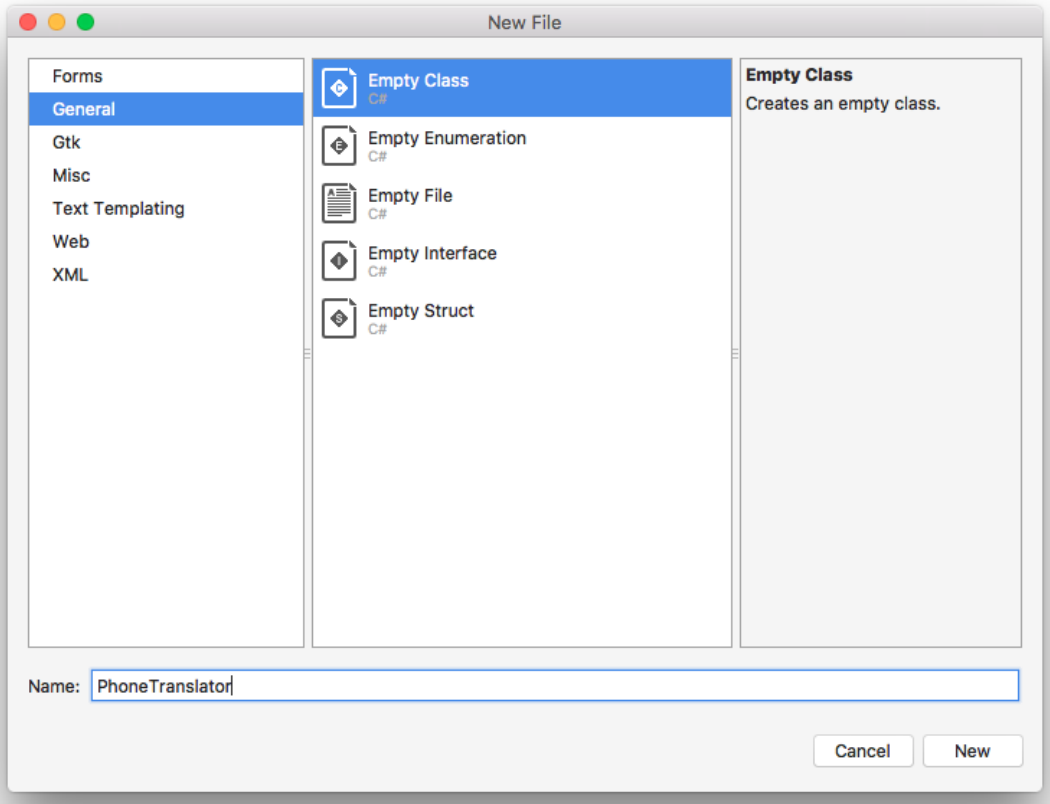
尝试在此时构建应用程序将导致稍后要修复的错误。

通过选择“文件”>“保存”，或按 ⌘ + S，保存对 **MainPage.xaml.cs** 所做的更改，然后关闭文件。

9. 在**解决方案面板**中，选择“Phoneword”项目，右键单击并选择“添加”>“新文件...”：



10. 在"新建文件"对话框中，选择"常规">"空类"，将新文件命名为 **PhoneTranslator**，然后单击"新建"按钮：

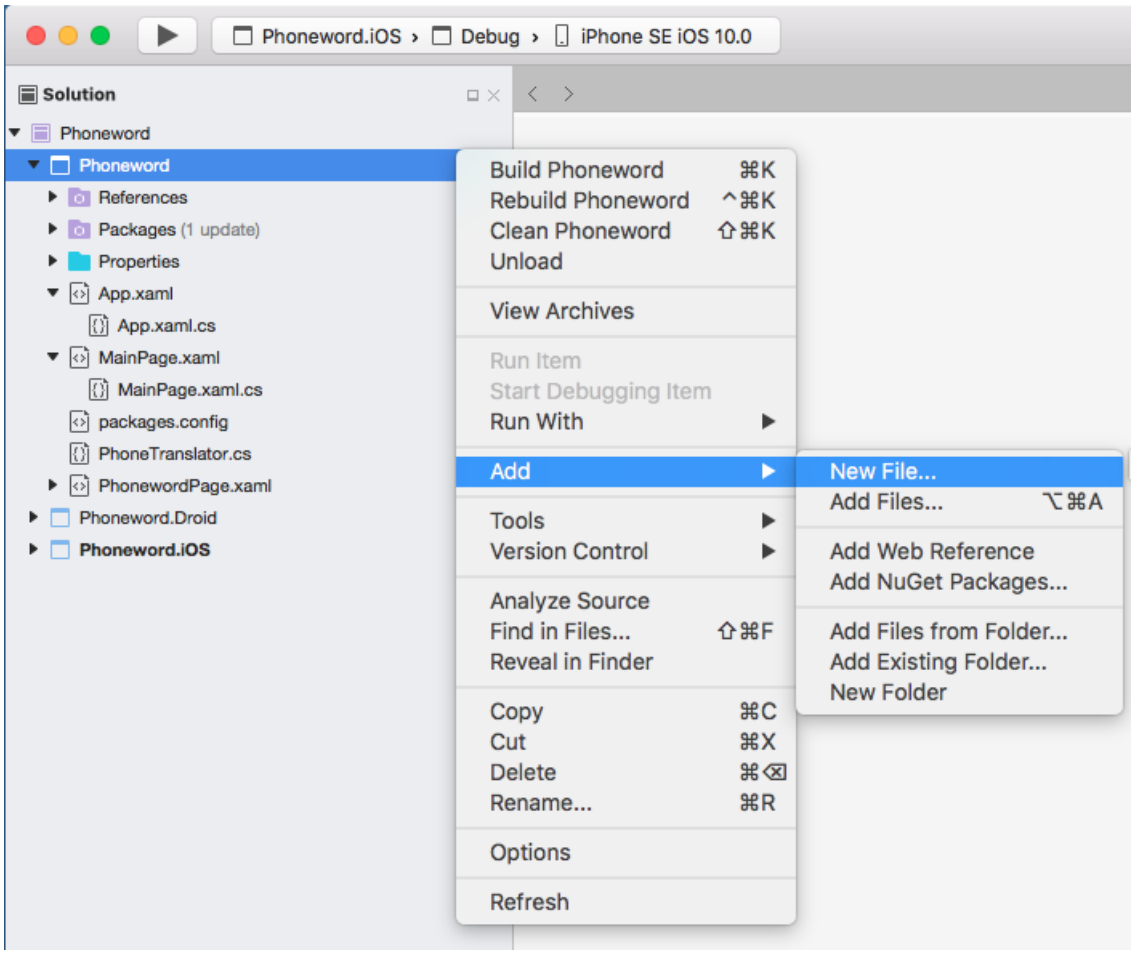


11. 在"PhoneTranslator.cs"中，删除所有模板代码并将其替换为以下代码。此代码会将手机词翻译为电话号码：

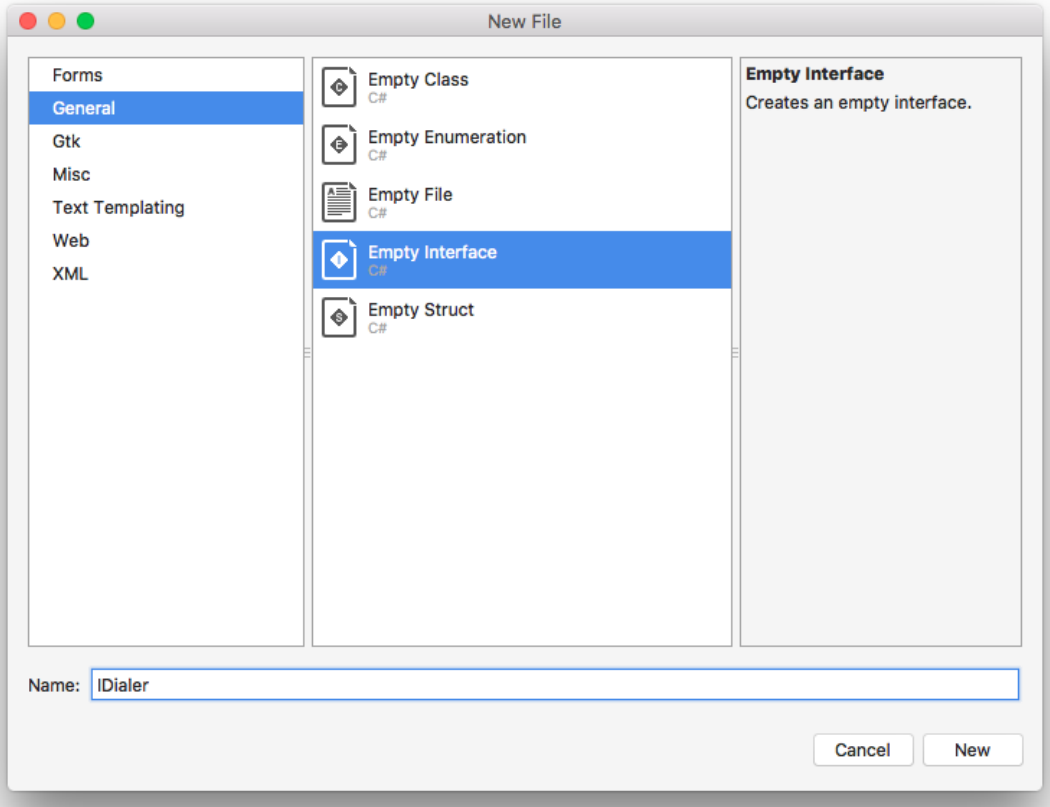
```
using System.Text;

namespace Phoneword
{
    public static class PhonewordTranslator
    {
        public static string ToNumber(string raw)
        {
            if (string.IsNullOrWhiteSpace(raw))
                return null;

            raw = raw.ToUpperInvariant();

            var newNumber = new StringBuilder();
            foreach (var c in raw)
            {
                if (" -0123456789".Contains(c))
                    newNumber.Append(c);
                else
                {
                    var result = TranslateToNumber(c);
                    if (result != null)
                        newNumber.Append(result);
                    // Bad character?
                    else
                        return null;
                }
            }
            return newNumber.ToString();
        }

        static bool Contains(this string keyString, char c)
        {
            return keyString.IndexOf(c) >= 0;
        }

        static readonly string[] digits = {
            "ABC", "DEF", "GHI", "JKL", "MNO", "PQRS", "TUV", "WXYZ"
        };

        static int? TranslateToNumber(char c)
        {
            for (int i = 0; i < digits.Length; i++)
            {
                if (digits[i].Contains(c))
                    return 2 + i;
            }
            return null;
        }
    }
}
```

通过选择“**文件**”>“**保存**”，或按 ⌘ **+ S**，保存对 **PhoneTranslator.cs** 所做的更改，然后关闭文件。

12. 在**解决方案面板**中，选择“Phoneword”项目，右键单击并选择“**添加**”>“**新文件...**”：



13. 在“新建文件”对话框中，选择“常规”>“空界面”，将新文件命名为 **IDialer**，然后单击“新建”按钮：



14. 在“IDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将定义 `Dial` 方法，必须在每个平台上

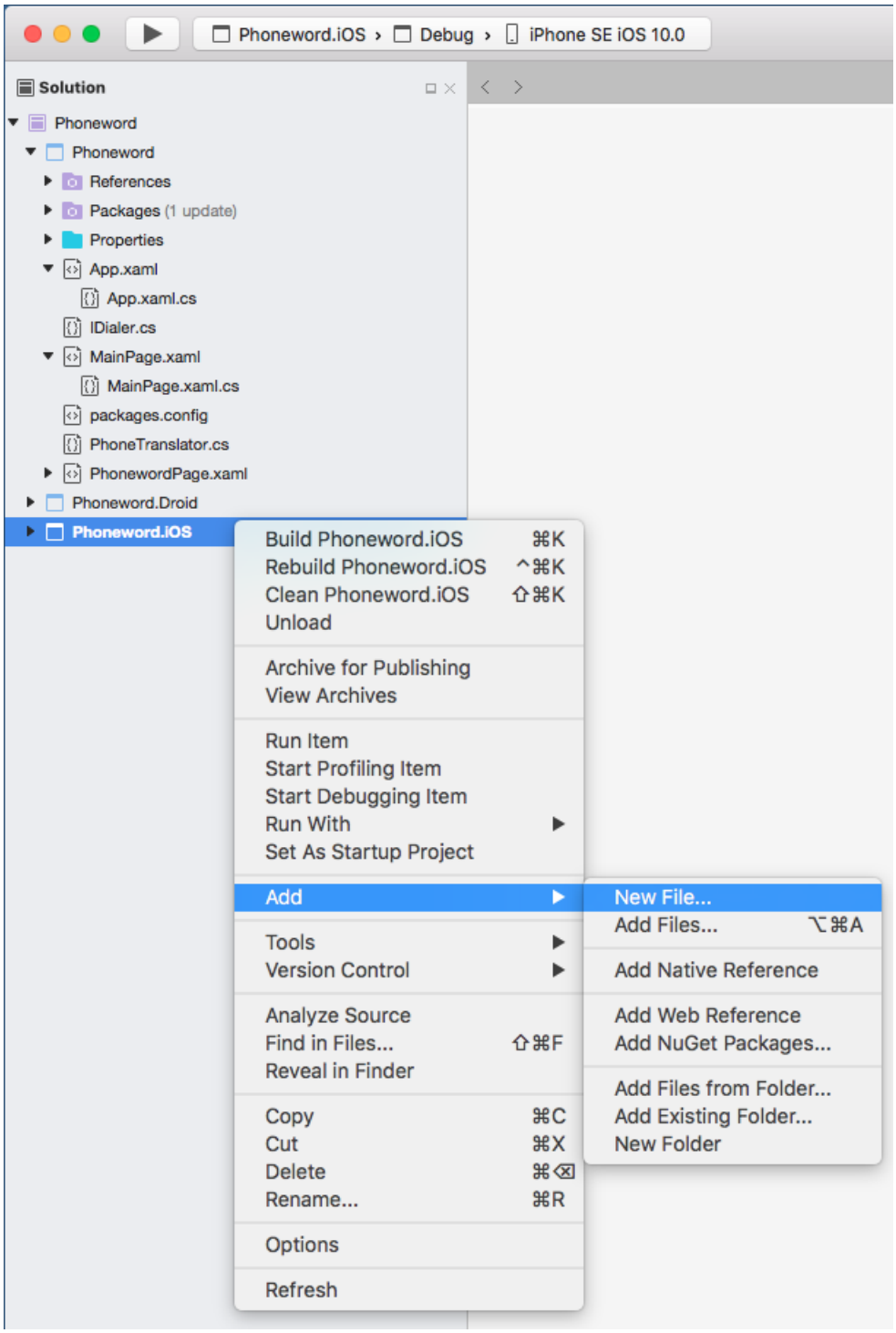
实现此方法，才可拨打翻译后的电话号码：

```
namespace Phoneword
{
    public interface IDialer
    {
        bool Dial(string number);
    }
}
```

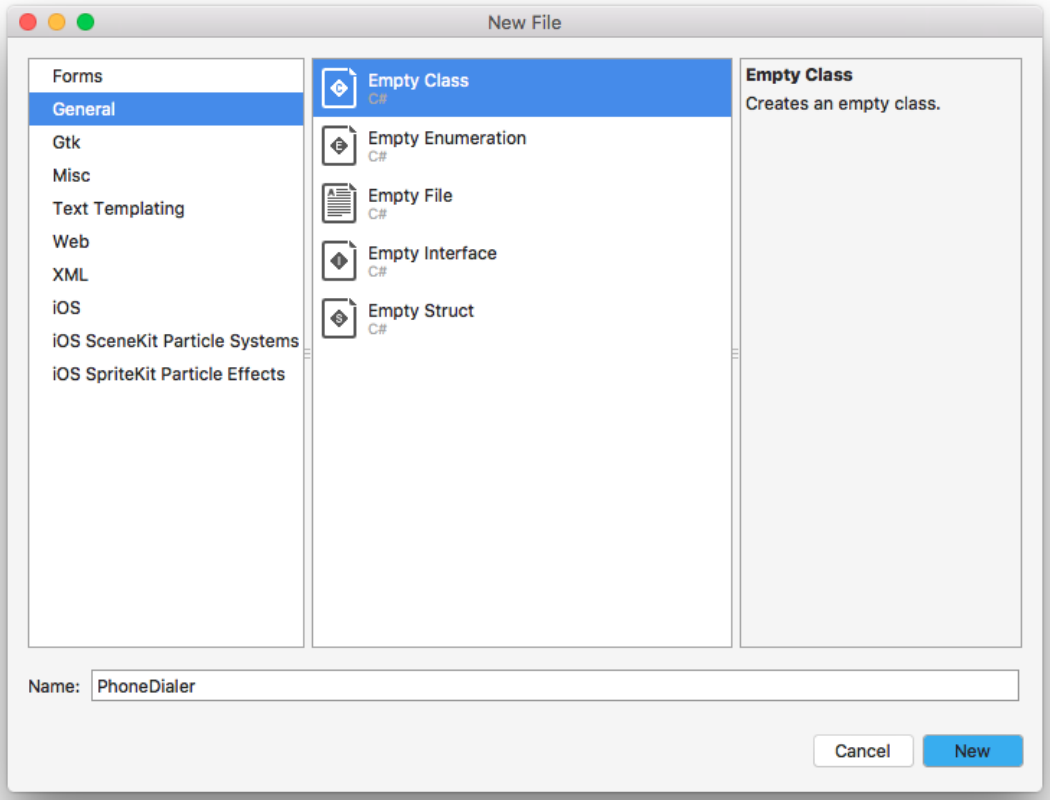
通过选择"文件">"保存"，或按 ⌘ + S，保存对 **IDialer.cs** 所做的更改，然后关闭文件。

> **NOTE**
>
> 此时完成了应用程序的常用代码。此时，可将特定于平台的电话拨号程序实现为 DependencyService。

15. 在"Solution Pad"中，选择"Phoneword.iOS"项目，右键单击并选择"添加">"新建文件...":



16. 在"新建文件"对话框中，选择"常规">"空类"，将新文件命名为 **PhoneDialer**，然后单击"新建"按钮：

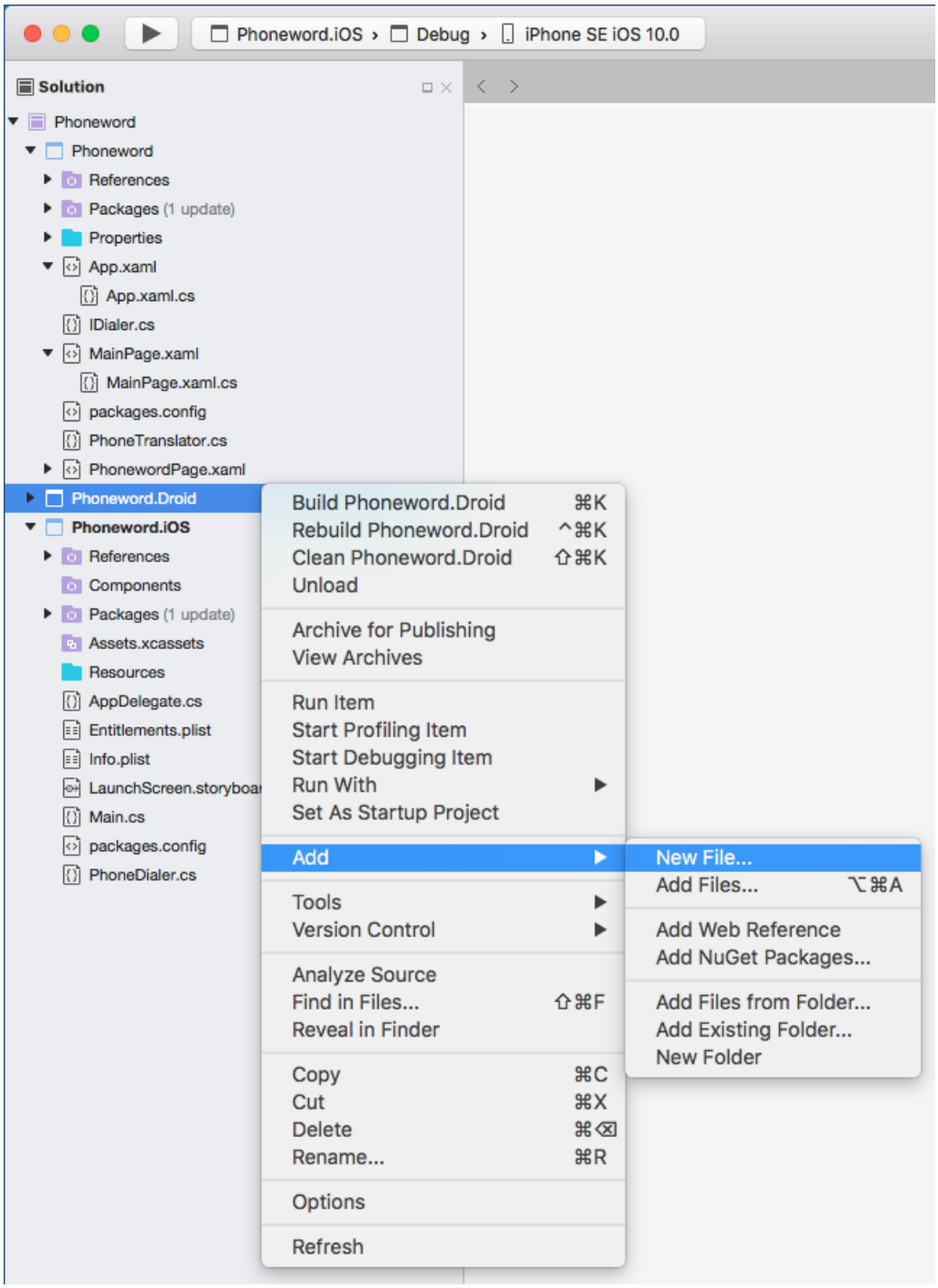


17. 在“PhoneDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将创建 `Dial` 方法，此方法将在 iOS 平台上用于拨打翻译后的电话号码：

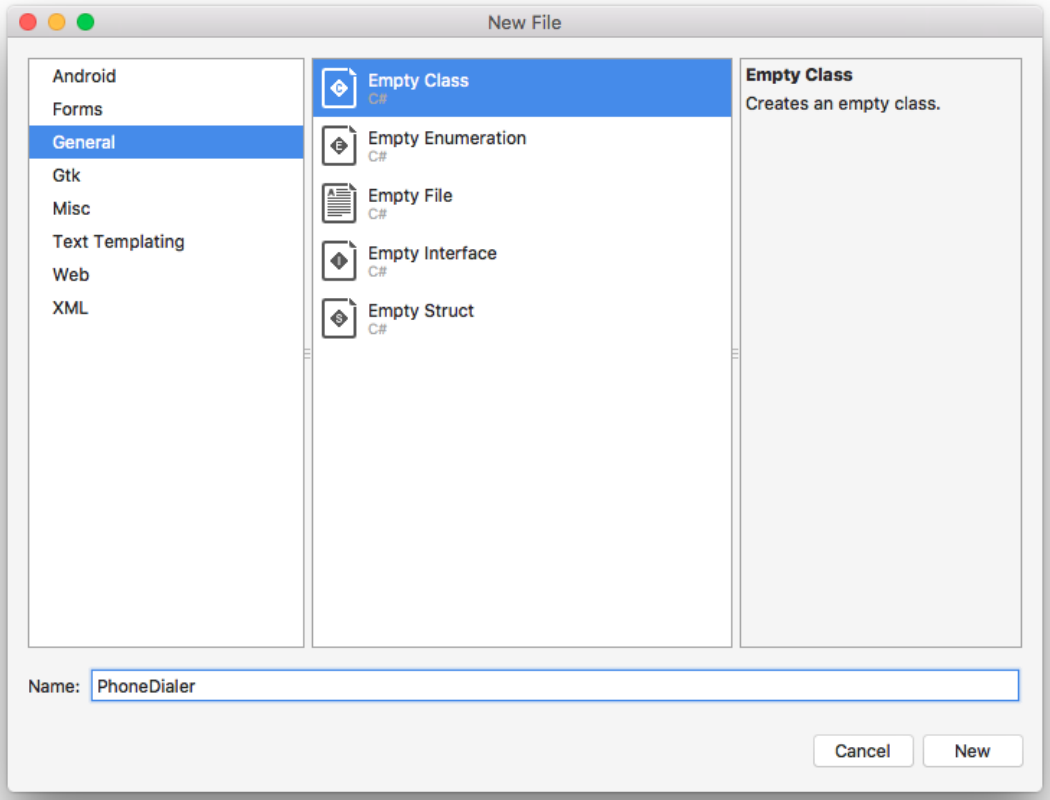
```
using Foundation;
using Phoneword.iOS;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.iOS
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            return UIApplication.SharedApplication.OpenUrl (
                new NSUrl ("tel:" + number));
        }
    }
}
```

通过选择“文件”>“保存”，或按 ⌘ + S，保存对 **PhoneDialer.cs** 所做的更改，然后关闭文件。

18. 在“Solution Pad”中，选择“Phoneword.Droid”项目，右键单击并选择“添加”>“新建文件...”：



19. 在“新建文件”对话框中，选择“常规”>“空类”，将新文件命名为 **PhoneDialer**，然后单击“新建”按钮：



20. 在“PhoneDialer.cs”中，删除所有模板代码并将其替换为以下代码。此代码将创建 `Dial` 方法，此方法将在 Android 平台上用于拨打翻译后的电话号码：

```
using Android.Content;
using Android.Telephony;
using Phoneword.Droid;
using System.Linq;
using Xamarin.Forms;
using Uri = Android.Net.Uri;

[assembly: Dependency(typeof(PhoneDialer))]
namespace Phoneword.Droid
{
    public class PhoneDialer : IDialer
    {
        public bool Dial(string number)
        {
            var context = MainActivity.Instance;
            if (context == null)
                return false;

            var intent = new Intent (Intent.ActionDial);
            intent.SetData (Uri.Parse ("tel:" + number));

            if (IsIntentAvailable (context, intent)) {
                context.StartActivity (intent);
                return true;
            }

            return false;
        }

        public static bool IsIntentAvailable(Context context, Intent intent)
        {
            var packageManager = context.PackageManager;

            var list = packageManager.QueryIntentServices (intent, 0)
                .Union (packageManager.QueryIntentActivities (intent, 0));

            if (list.Any ())
                return true;

            var manager = TelephonyManager.FromContext (context);
            return manager.PhoneType != PhoneType.None;
        }
    }
}
```

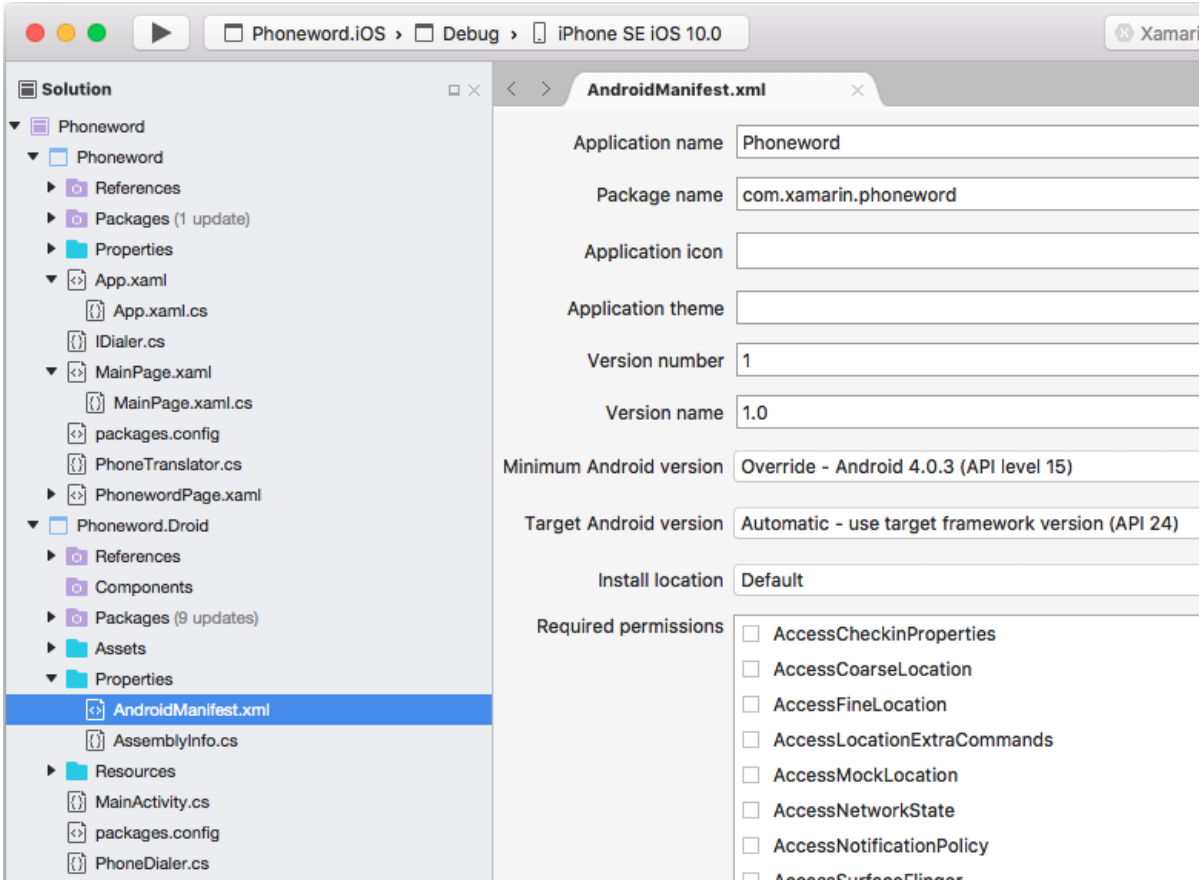
请注意，若要更好地理解此代码，需要使用最新 Android API。通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **PhoneDialer.cs** 所做的更改，然后关闭文件。

21. 在“Solution Pad”的“Phoneword.Droid”项目中，双击“MainActivity.cs”将其打开，然后删除所有模板代码并将其替换成下列代码：

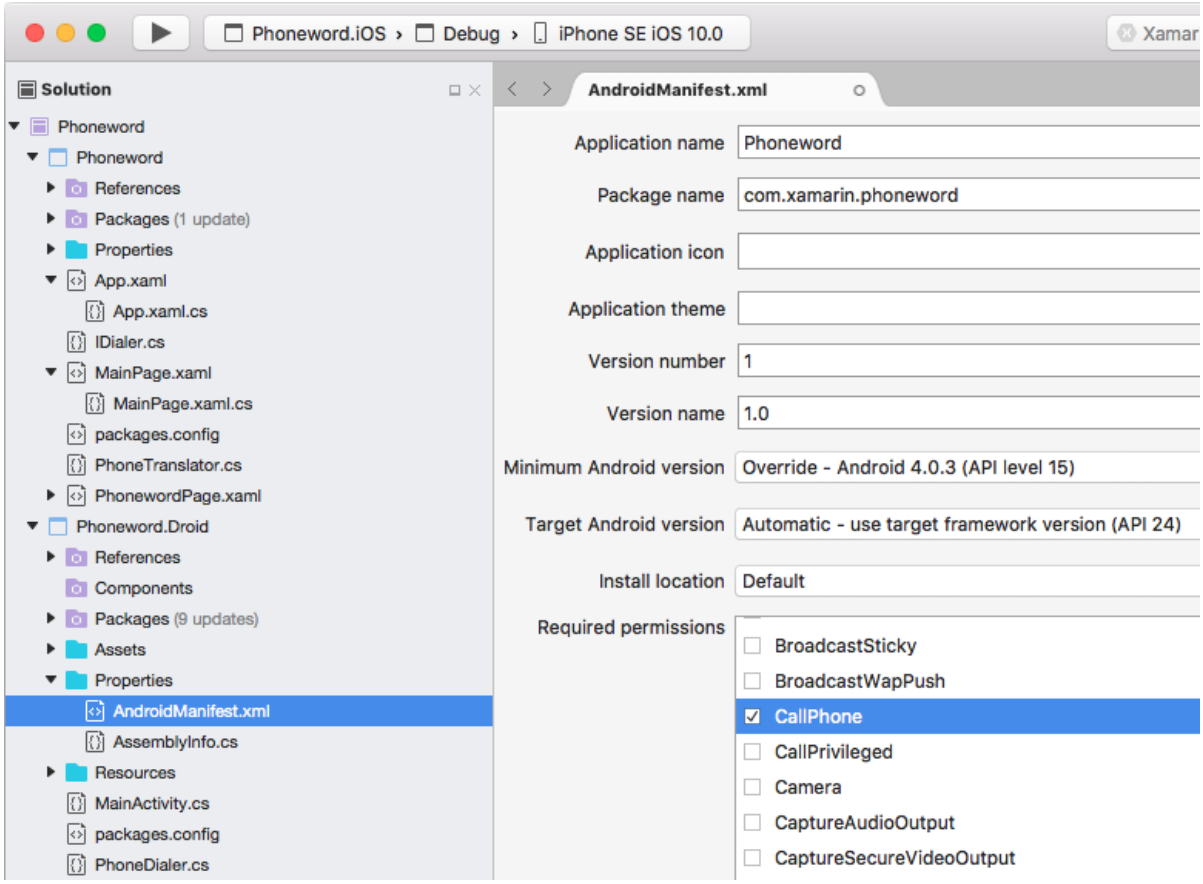
```
using Android.App;
using Android.Content.PM;
using Android.OS;

namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword", Icon = "@mipmap/icon", Theme = "@style/MainTheme", MainLauncher =
true,
              ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}
```

选择"文件">"保存"(或按 ⌘+S),保存对 MainActivity.cs 所做的更改,然后关闭文件。

22. 在"Solution Pad"中,展开"属性"文件夹,然后右键单击"AndroidManifest.xml"文件:



23. 在"所需的权限"部分中,启用"CallPhone"权限。这将向应用程序授予进行电话呼叫的权限:

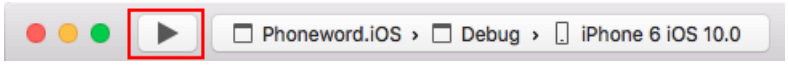


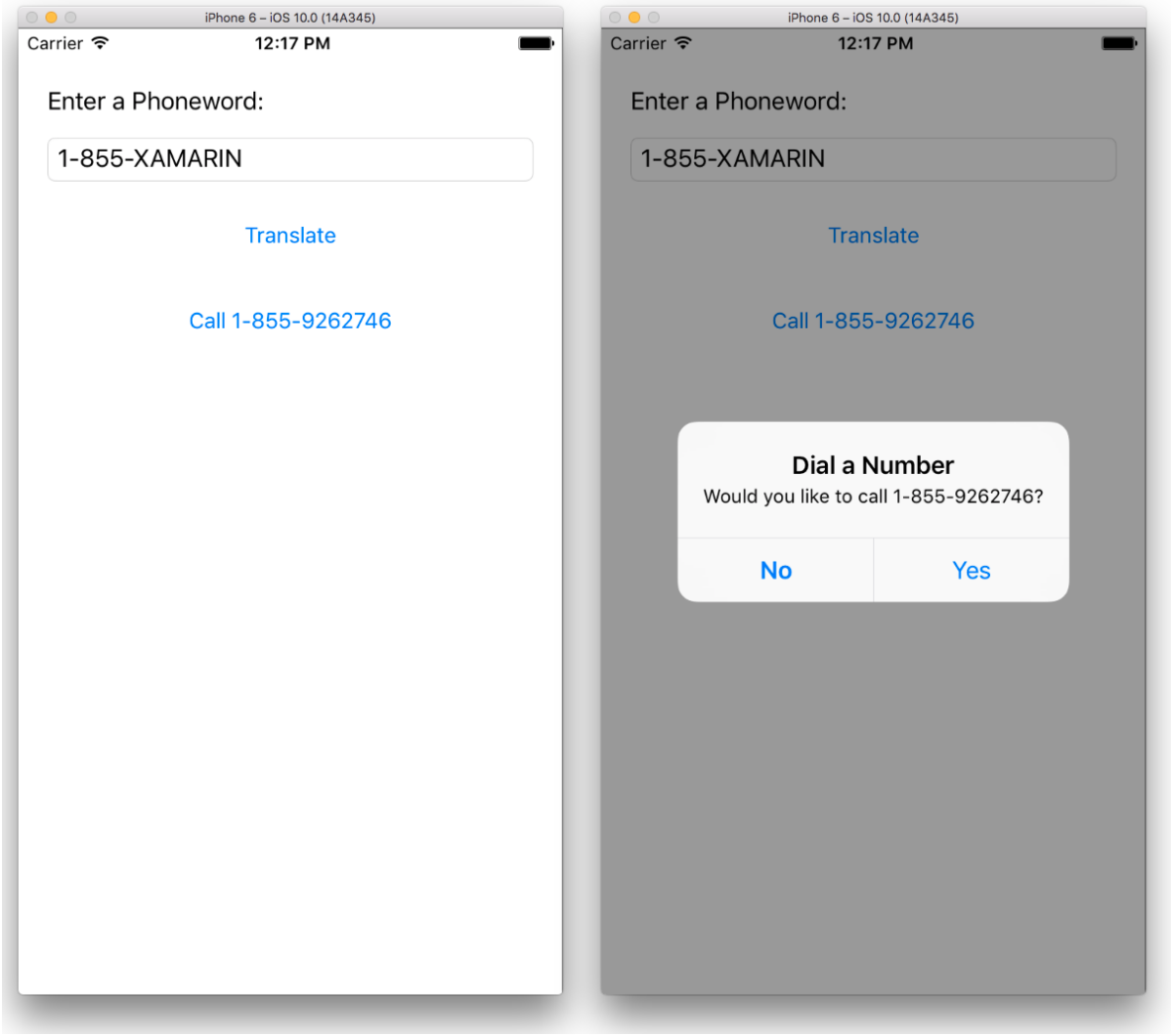
通过选择"文件">"保存"，或按 ⌘ + **S**，保存对 **AndroidManifest.xml** 所做的更改，然后关闭文件。

24. 在 Visual Studio for Mac 中，选择"生成">"生成所有"菜单项，或按 ⌘ + **B**。应用程序将生成，并在 Visual Studio for Mac 工具栏中显示一条成功消息。

25. 如果发生错误，请重复前面的步骤并更正任何错误，直到成功生成应用程序。

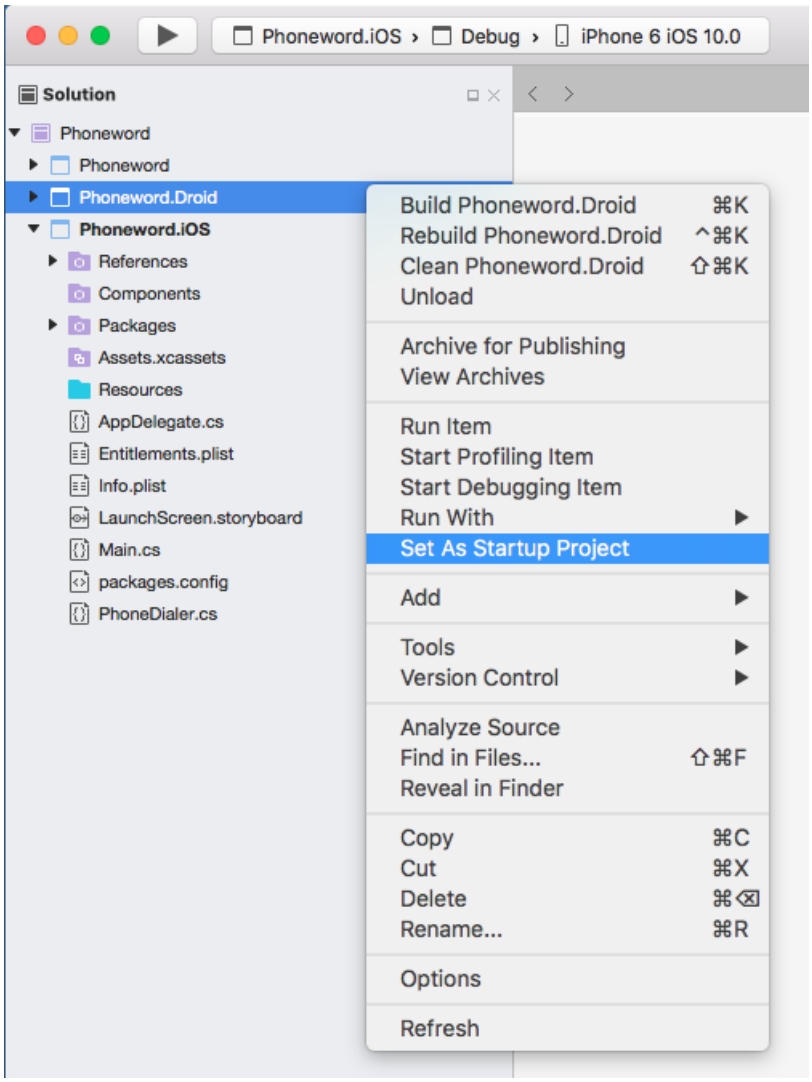
26. 在 Visual Studio for Mac 工具栏中，按"开始"按钮（类似"播放"按钮的三角形按钮），启动 iOS 模拟器内的应用程序：



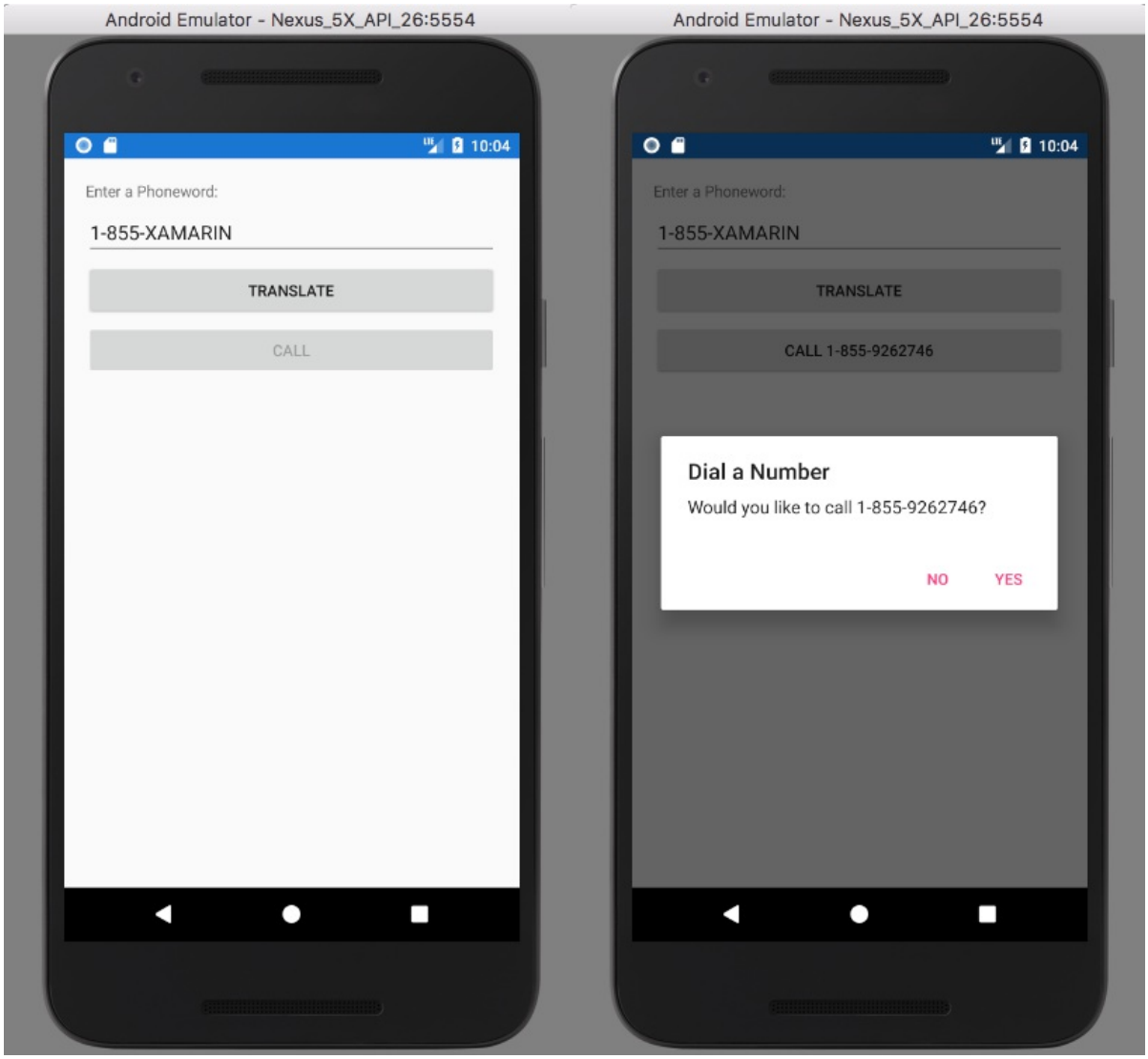


注意：iOS 模拟器不支持电话呼叫。

27. 在**解决方案面板**中，选择"Phoneword.Droid"项目，右键单击并选择"设为启动项目"：



28. 在 Visual Studio for Mac 工具栏中，按“开始”按钮（类似“播放”按钮的三角形按钮），启动 Android 模拟器内的应用程序：



> **WARNING**
> Android Emulator 不支持电话呼叫。

祝贺你完成 Xamarin.Forms 应用程序。本指南的下一个主题将回顾此演练中采用的步骤，以深入了解使用 Xamarin.Forms 开发应用程序的基础知识。

相关链接

- 通过 DependencyService 访问本机功能
- Phoneword（示例）

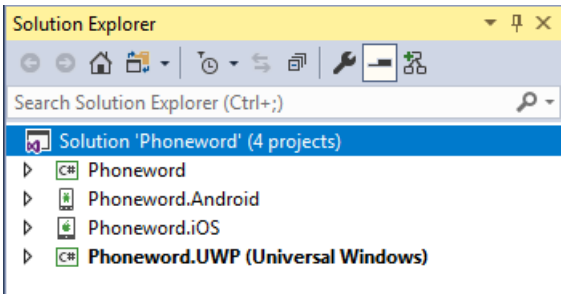
Xamarin.Forms 深度解析

2018/11/2 • • Edit Online

在 Xamarin.Forms 快速入门中，生成了 Phoneword 应用程序。本文对已生成的内容进行回顾，以深入了解有关 Xamarin.Forms 应用程序工作原理的基础知识。

Visual Studio 简介

Visual Studio 将代码组织为解决方案和项目。解决方案是可以容纳一个或多个项目的容器。项目可以是应用程序、支持库、测试应用程序等。Phoneword 应用程序包含 1 个内附 4 个项目的解决方案，如以下屏幕截图所示：

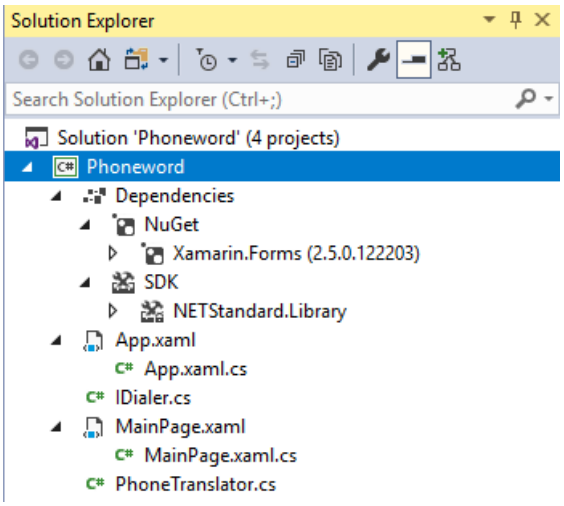


这些项目如下：

- Phoneword - 此项目是 .NET Standard 库项目，其中包含所有共享代码和共享 UI。
- Phoneword.Android - 此项目包含 Android 特定代码，是 Android 应用程序的入口点。
- Phoneword.iOS - 此项目包含 iOS 特定代码，是 iOS 应用程序的入口点。
- Phoneword.UWP - 此项目包含通用 Windows 平台特定代码，是 UWP 应用程序的入口点。

Xamarin.Forms 应用程序剖析

以下屏幕截图显示 Visual Studio 中 Phoneword .NET Standard 库项目的内容：

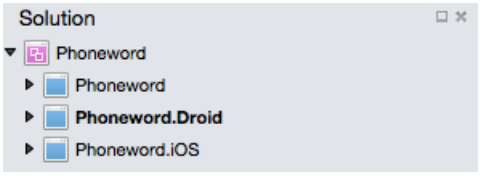


项目具有包含 NuGet 和 SDK 节点的依赖项节点：

- **NuGet**– 已添加到项目中的 Xamarin.Forms NuGet 包。
- **SDK** – `NETStandard.Library` 元包，它引用定义 .NET Standard 的一整套 NuGet 包。

Visual Studio for Mac 简介

Visual Studio for Mac 遵循将代码组织为解决方案和项目的 Visual Studio 做法。解决方案是可以容纳一个或多个项目的容器。项目可以是应用程序、支持库、测试应用程序等。Phoneword 应用程序包含 1 个内附 3 个项目的解决方案，如以下屏幕截图所示：

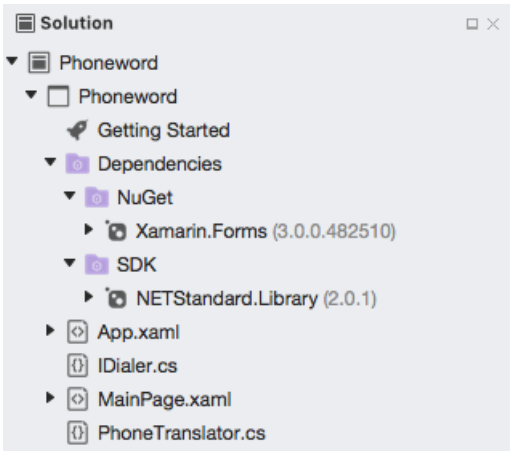


这些项目如下：

- Phoneword - 此项目是 .NET Standard 库项目，其中包含所有共享代码和共享 UI。
- Phoneword.Droid - 此项目包含 Android 特定代码，是 Android 应用程序的入口点。
- Phoneword.iOS - 此项目包含 iOS 特定代码，是 iOS 应用程序的入口点。

Xamarin.Forms 应用程序剖析

以下屏幕截图显示 Visual Studio for Mac 中 Phoneword .NET Standard 库项目的内容：



项目具有包含 NuGet 和 SDK 节点的依赖项节点：

- **NuGet**– 已添加到项目中的 Xamarin.Forms NuGet 包。
- **SDK** – `NETStandard.Library` 元包，它引用定义 .NET Standard 的一整套 NuGet 包。

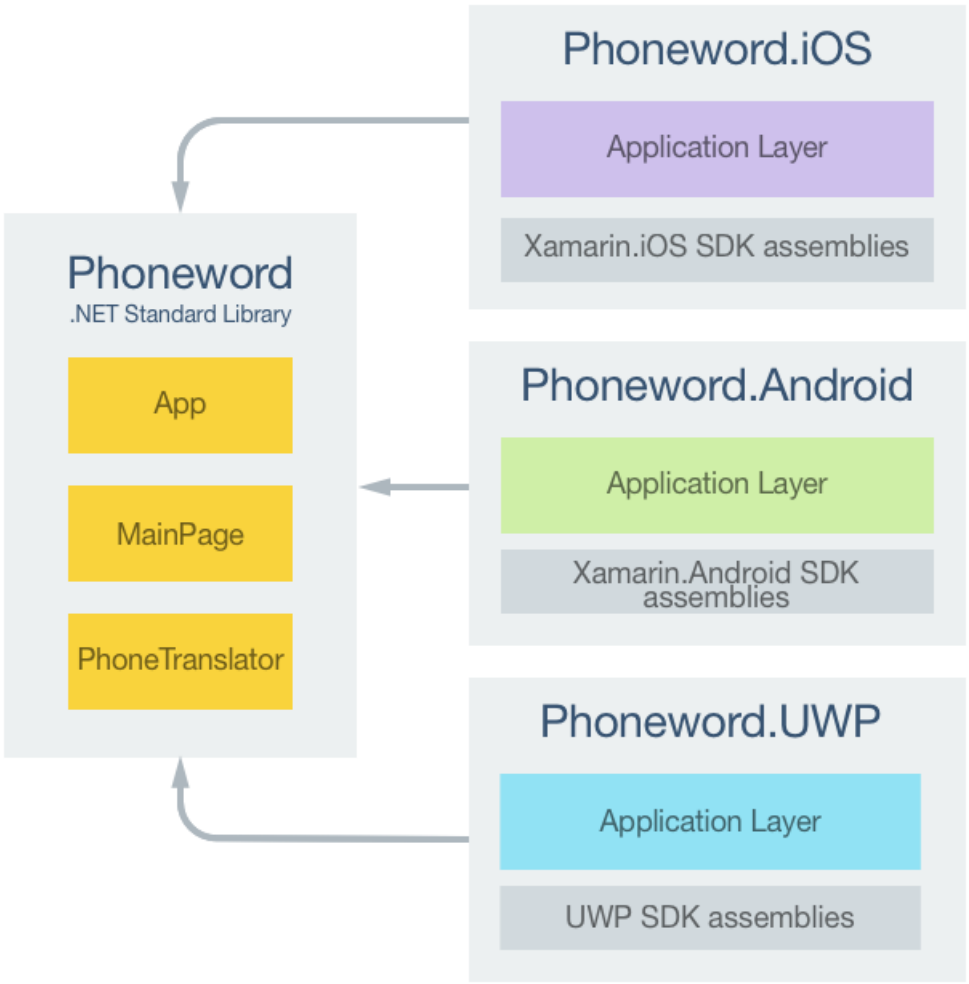
项目还包括多个文件：

- **App.xaml** - `App` 类的 XAML 标记，该类定义应用程序的资源字典。
- **App.xaml.cs** - `App` 类的代码隐藏，该类负责实例化应用程序在每个平台上将显示的首页，并处理应用程序生命周期事件。
- IDialer.cs – `IDialer` 接口，该接口指定 `Dial` 方法必须由任何实现类提供。
- **MainPage.xaml** - `MainPage` 类的 XAML 标记，该类定义应用程序启动时所显示页的 UI。
- **MainPage.xaml.cs** - `MainPage` 类的代码隐藏，该类包含用户与页面交互时执行的业务逻辑。
- **PhoneTranslator.cs** - 负责将电话文字转换为电话号码的业务逻辑，此逻辑从 **MainPage.xaml.cs** 调用。

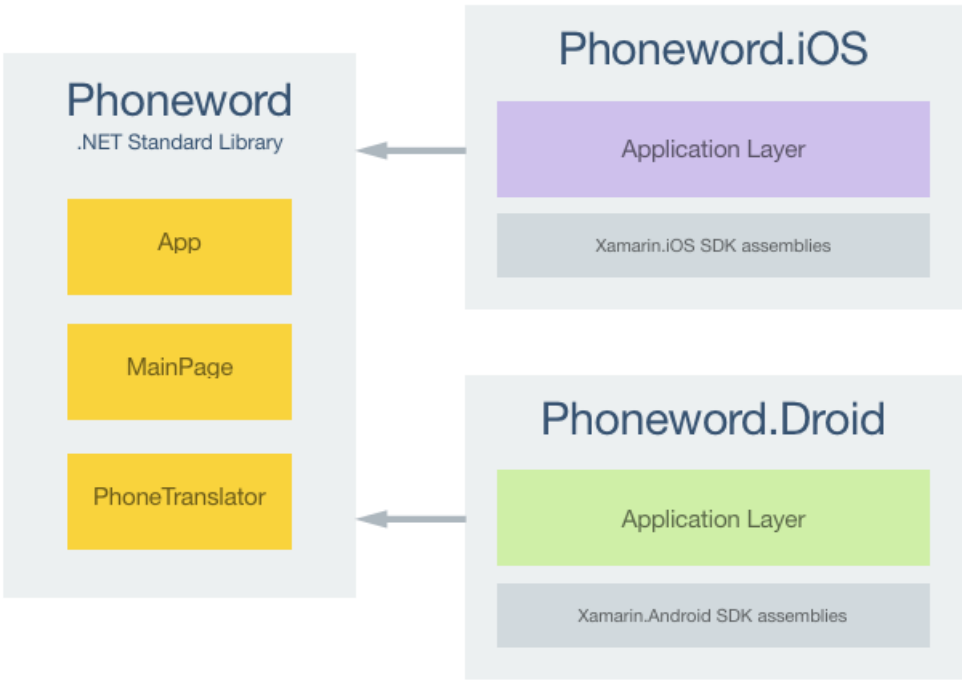
有关 Xamarin.iOS 应用程序剖析的详细信息，请参阅 Xamarin.iOS 应用程序剖析。有关 Xamarin.Android 应用程序剖析的详细信息，请参阅 Xamarin Android 应用程序剖析。

体系结构和应用程序基础知识

Xamarin.Forms 应用程序采用与传统跨平台应用程序相同的构建方式。共享代码通常位于 .NET Standard 库中，平台特定应用程序将使用此共享代码。下图概要演示了 Phoneword 应用程序的这种关系：



Xamarin.Forms 应用程序采用与传统跨平台应用程序相同的构建方式。共享代码通常位于 .NET Standard 库中，平台特定应用程序将使用此共享代码。下图概要演示了 Phoneword 应用程序的这种关系：



若要最大限度重用启动代码，Xamarin.Forms 应用程序需有一个名为 `App` 的单个类，该类负责实例化应用程序在每个平台上将显示的首页，如以下代码示例所示：

```
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public App()
        {
            InitializeComponent();
            MainPage = new MainPage();
        }
        ...
    }
}
```

此代码将 `App` 类的 `MainPage` 属性设置为 `MainPage` 类的一个新实例。此外，`XamlCompilation` 属性可打开 XAML 编译器，以使 XAML 直接编译为中间语言。有关详细信息，请参阅 XAML 编译。

在每个平台上启动应用程序

iOS

若要在 iOS 中启动 Xamarin.Forms 初始页面，Phoneword.iOS 项目应包括继承自 `FormsApplicationDelegate` 类的 `AppDelegate` 类，如以下代码示例所示：

```
namespace Phoneword.iOS
{
    [Register ("AppDelegate")]
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        public override bool FinishedLaunching (UIApplication app, NSDictionary options)
        {
            global::Xamarin.Forms.Forms.Init ();
            LoadApplication (new App ());
            return base.FinishedLaunching (app, options);
        }
    }
}
```

通过调用 `Init` 方法，`FinishedLaunching` 替代初始化 Xamarin.Forms 框架。这会导致在调用将根视图控制器设置为 `LoadApplication` 方法之前，将特定于 iOS 的 Xamarin.Forms 实现加载到应用程序。

Android

要在 Android 中启动 Xamarin.Forms 初始页面，Phoneword.Droid 项目应包括使用 `MainLauncher` 属性创建 `Activity` 的代码，以及继承自 `FormsAppCompatActivity` 类的活动，如以下代码示例所示：

```
namespace Phoneword.Droid
{
    [Activity(Label = "Phoneword",
              Icon = "@mipmap/icon",
              Theme = "@style/MainTheme",
              MainLauncher = true,
              ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        internal static MainActivity Instance { get; private set; }

        protected override void OnCreate(Bundle bundle)
        {
            TabLayoutResource = Resource.Layout.Tabbar;
            ToolbarResource = Resource.Layout.Toolbar;

            base.OnCreate(bundle);
            Instance = this;
            global::Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication(new App());
        }
    }
}
```

通过调用 `Init` 方法，`OnCreate` 替代初始化 Xamarin.Forms 框架。这会导致在加载 Xamarin.Forms 应用程序之前，将特定于 Android 的 Xamarin.Forms 实现加载到应用程序。此外，`MainActivity` 类将在 `Instance` 属性中存储对其自身的引用。引用自 `PhoneDialer` 类的 `Instance` 属性被称为本地上下文。

通用 Windows 平台

在通用 Windows 平台 (UWP) 应用程序中，可从 `App` 类调用初始化 Xamarin.Forms 框架的 `Init` 方法：

```
Xamarin.Forms.Forms.Init (e);

if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
  ...
}
```

这会将特定于 UWP 的 Xamarin.Forms 实现加载到应用程序。可通过 `MainPage` 类启动 Xamarin.Forms 初始页面，如以下代码示例所示：

```
namespace Phoneword.UWP
{
    public sealed partial class MainPage
    {
        public MainPage()
        {
            this.InitializeComponent();
            this.LoadApplication(new Phoneword.App());
        }
    }
}
```

可通过 `LoadApplication` 方法加载 Xamarin.Forms 应用程序。

> **NOTE**
>
> 通用 Windows 平台 (UWP) 应用可以使用 Xamarin.Forms 生成，但只能在 Windows 上使用 Visual Studio。

用户界面

可使用 4 个主要控件组创建 Xamarin.Forms 应用程序的用户界面。

1. **页面** - Xamarin.Forms 页呈现跨平台移动应用程序屏幕。Phoneword 应用程序使用 `ContentPage` 类显示单个屏幕。有关页面的详细信息，请参阅 Xamarin.Forms 页面。
2. **布局** - Xamarin.Forms 布局是用于将视图组合到逻辑结构的容器。Phoneword 应用程序使用 `StackLayout` 类以垂直堆叠方式排列控件。有关布局的详细信息，请参阅 Xamarin.Forms 布局。
3. **视图** - Xamarin.Forms 视图是显示在用户界面上的控件，如标签、按钮和文本输入框。Phoneword 应用程序使用 `Label`、`Entry` 和 `Button` 控件。有关视图的详细信息，请参阅 Xamarin.Forms 视图。
4. **单元格** - Xamarin.Forms 单元格是专门用于列表中的项的元素，描述列表中每个项的绘制方式。Phoneword 应用程序不会使用任何单元格。有关单元格的详细信息，请参阅 Xamarin.Forms 单元格。

在运行时，每个控件都会映射到其本身的本机等效项(即呈现的内容)。

在任一平台运行 Phoneword 应用程序时，此应用会显示对应于 Xamarin.Forms 中 `Page` 的单一屏幕。`Page` 在 Android 中表示为一个 *ViewGroup*，在 iOS 中表示为视图控制器，在 Windows 通用平台中则表示为一个页面。Phoneword 应用程序也会实例化表示 `MainPage` 类的 `ContentPage` 对象，该类的 XAML 标记如以下代码示例所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
                         xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                         x:Class="Phoneword.MainPage">
        ...
        <StackLayout>
            <Label Text="Enter a Phoneword:" />
            <Entry x:Name="phoneNumberText" Text="1-855-XAMARIN" />
            <Button x:Name="translateButton" Text="Translate" Clicked="OnTranslate" />
            <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
        </StackLayout>
</ContentPage>
```

`MainPage` 类使用 `StackLayout` 控件在屏幕上自动排列控件，而不考虑屏幕大小。根据添加顺序，以垂直方式逐个放置每个子元素。`StackLayout` 控件包含 1 个用于在页面上显示文本的 `Label` 控件、1 个用于接收文本用户输入的 `Entry` 控件和 2 个用于在响应触摸事件时执行代码的 `Button` 控件。

有关 Xamarin.Forms 中 XAML 的详细信息，请参阅 Xamarin.Forms XAML 基础知识。

响应用户交互

XAML 中定义的对象可触发在隐藏文件中处理的事件。以下代码示例演示了 `MainPage` 类的代码隐藏中的 `OnTranslate` 方法，该类在响应 `Clicked` 事件触发"转换"按钮时执行。

```
void OnTranslate(object sender, EventArgs e)
{
    translatedNumber = Core.PhonewordTranslator.ToNumber (phoneNumberText.Text);
    if (!string.IsNullOrWhiteSpace (translatedNumber)) {
        callButton.IsEnabled = true;
        callButton.Text = "Call " + translatedNumber;
    } else {
        callButton.IsEnabled = false;
        callButton.Text = "Call";
    }
}
```

`OnTranslate` 方法将 phoneword 转换为其相应的电话号码，并在响应时设置调用按钮上的属性。XAML 类的代码隐藏文件可使用为其分配的、具有 `x:Name` 属性的名称访问 XAML 中定义的对象。分配给此属性的值与 C# 变量

的规则相同，因为该值必须以字母或下划线开头，且不包含嵌入的空格。

对 `OnTranslate` 方法的切换按钮布线会在 `MainPage` 类的 XAML 标记中出现：

```
<Button x:Name="translateButton" Text="Translate" Clicked="OnTranslate" />
```

Phoneword 中引入的其他概念

用于 Xamarin.Forms 的 Phoneword 应用程序引入了多个本文未提及的概念。这些概念包括：

- 启用和禁用按钮。通过更改 `Button` 的 `IsEnabled` 属性，可将其打开或关闭。例如，以下代码示例禁用 `callButton`：

  ```
  callButton.IsEnabled = false;
  ```

- 显示警报对话框。用户按呼叫按钮时，Phoneword 应用程序会显示“警报”对话框，其中包含发出或取消呼叫的选项。`DisplayAlert` 方法用于创建该对话框，如以下代码示例所示：

  ```
  await this.DisplayAlert (
          "Dial a Number",
          "Would you like to call " + translatedNumber + "?",
          "Yes",
          "No");
  ```

- 通过 `DependencyService` 类访问本机功能。Phoneword 应用程序使用 `DependencyService` 类将 `IDialer` 接口解析到特定于平台的电话拨号实现中，如以下 Phoneword 项目中的代码示例所示：

  ```
  async void OnCall (object sender, EventArgs e)
  {
      ...
      var dialer = DependencyService.Get<IDialer> ();
      ...
  }
  ```

  有关 `DependencyService` 类的详细信息，请参阅通过 DependencyService 访问本机功能。

- 通过 URL 发出电话呼叫。Phoneword 应用程序使用 `OpenURL` 启动系统电话应用。URL 包含 `tel:` 前缀，后跟要呼叫的电话号码，如以下 iOS 项目中的代码示例所示：

  ```
  return UIApplication.SharedApplication.OpenUrl (new NSUrl ("tel:" + number));
  ```

- 调整平台布局。使用 `Device` 类，开发人员能够根据每个平台自定义应用程序布局和功能，如以下代码示例所示（此示例使用不同平台上的另一个 `Padding` 值正确显示每一页）：

  ```
  <ContentPage xmlns="http://xamarin.com/schemas/2014/forms" ... >
      <ContentPage.Padding>
          <OnPlatform x:TypeArguments="Thickness">
              <On Platform="iOS" Value="20, 40, 20, 20" />
              <On Platform="Android, UWP" Value="20" />
          </OnPlatform>
      </ContentPage.Padding>
      ...
  </ContentPage>
  ```

有关平台调整的详细信息，请参阅设备类。

测试和部署

Visual Studio for Mac 和 Visual Studio 均提供许多用于测试和部署应用程序的选项。调试应用程序是应用程序开发生命周期的普遍一环，有助于诊断代码问题。有关详细信息，请参阅设置断点、逐步执行代码和日志窗口的输出信息。

模拟器是开始部署和测试应用程序的有利位置，其提供用于测试应用程序的有用功能。但是，用户不会在模拟器中使用最终应用程序，因此应尽早并经常在实际设备上测试应用程序。有关 iOS 设备预配的详细信息，请参阅设备预配。有关 Android 设备预配的详细信息，请参阅设置设备进行开发。

总结

本文介绍了使用 Xamarin.Forms 开发应用程序的基础知识。涵盖的主题包括：Xamarin.Forms 应用程序剖析、体系结构和应用程序基础知识以及用户界面。

在本指南的下一部分将介绍如何扩展应用程序以包含多个屏幕，从而探索更高级的 Xamarin.Forms 体系结构和概念。

了解 Xamarin.Forms 多屏显示

2018/7/28 • • Edit Online

本指南将扩展在《了解 Xamarin.Forms》指南中创建的 Phoneword 应用程序以导航到第二个屏幕。涵盖的主题包括页面导航和将数据绑定到集合。

第 1 部分：快速入门

本指南的第一部分演示如何将第二个屏幕添加到 Phoneword 应用程序，以跟踪应用程序的历史调用记录。

第 2 部分：深度分析

本指南的第二部分对已生成的内容进行回顾，以深入了解 Xamarin.Forms 应用程序中的页面导航和数据绑定。

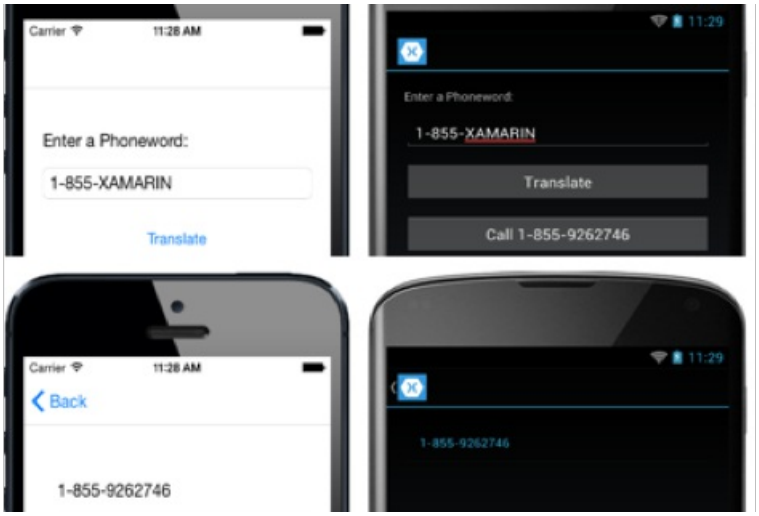
相关链接

- Xamarin.Forms 简介
- 在 Visual Studio 中进行调试
- Visual Studio for Mac 方案 - 调试
- 免费自学教程（视频）
- Xamarin 入门（视频）

Xamarin.Forms 多屏显示快速入门

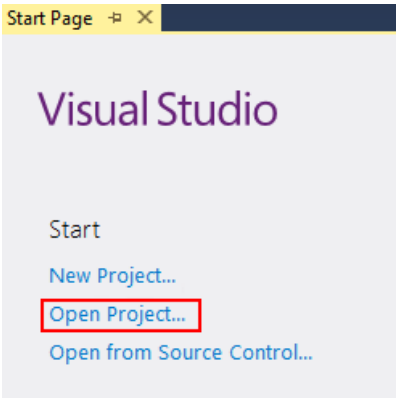
2018/11/2 • • Edit Online

本快速入门演示了如何通过添加第二个屏幕来扩展 Phoneword 应用程序，以跟踪应用程序的呼叫历史记录。最终的应用程序如下所示：

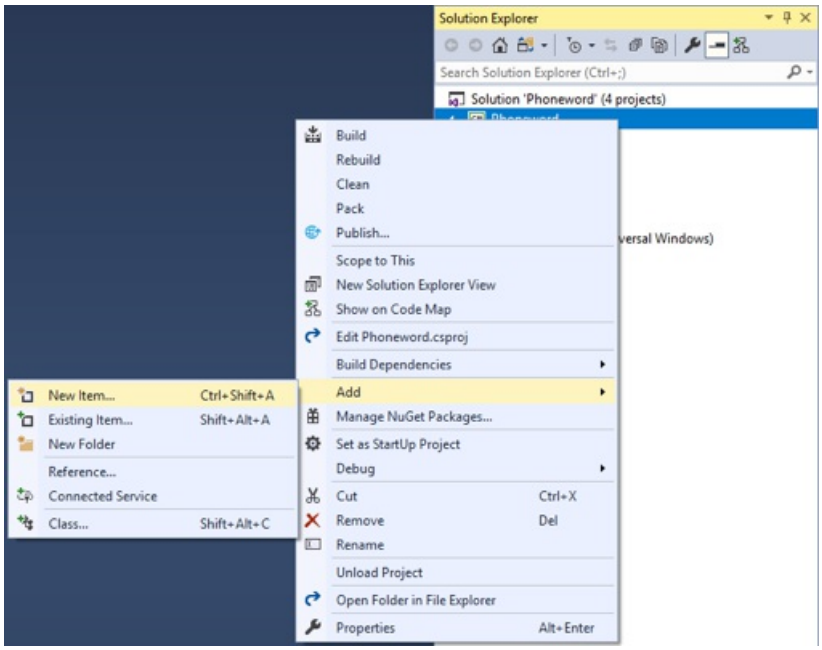


使用 Visual Studio 更新应用

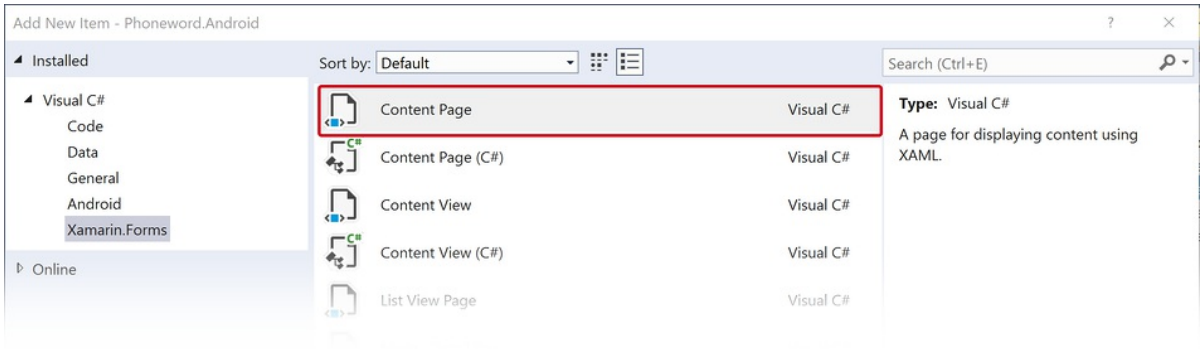
1. 启动 Visual Studio。在起始页上，单击“打开项目...”，然后在“打开项目”对话框中选择 Phoneword 项目的解决方案文件：



2. 在“解决方案资源管理器”中，右键单击“Phoneword”项目，然后选择“添加”>“新建项...”：



3. 在“添加新项”对话框中，选择“Visual C# 项”>“Xamarin.Forms”>“内容页”，将新项命名为“CallHistoryPage”，然后单击“添加”按钮。这会将一个名为 **CallHistoryPage** 的页面添加到项目：

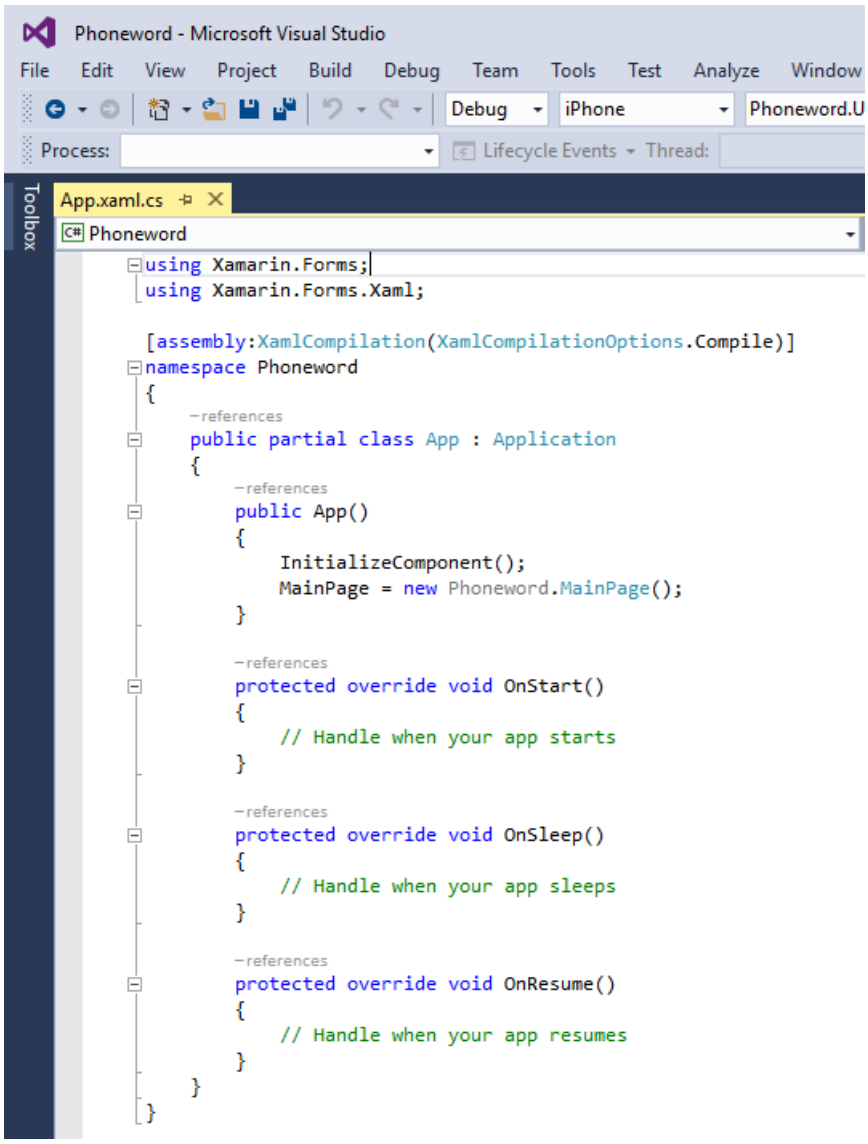


4. 在 **CallHistoryPage.xaml** 中，删除所有模板代码并将其替换为以下代码。此代码以声明方式定义页面上的用户界面：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="Phoneword.CallHistoryPage"
                  Title="Call History">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
      <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
    </StackLayout>
</ContentPage>
```

通过按 **Ctrl+S**，保存对 **CallHistoryPage.xaml** 所做的更改，然后关闭文件。

5. 在“解决方案资源管理器”中，双击共享“Phoneword”项目中的“App.xaml.cs”文件打开该文件：

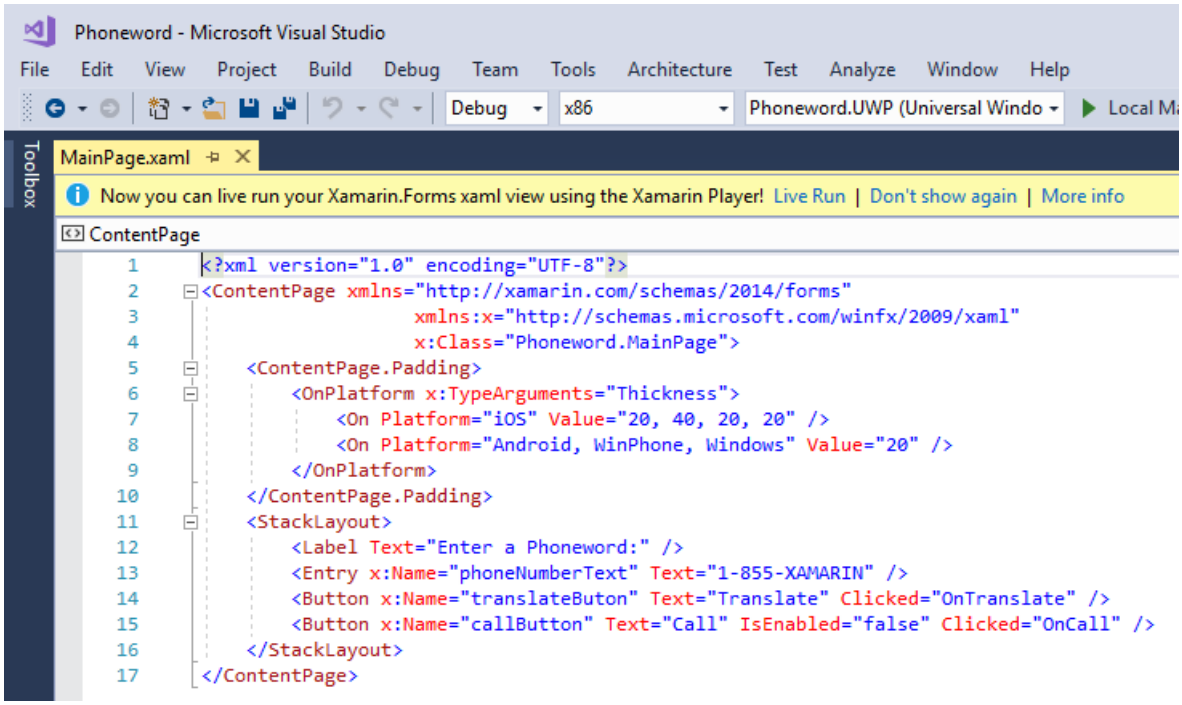


6. 在 **App.xaml.cs** 中，导入 `System.Collections.Generic` 命名空间，添加 `PhoneNumbers` 属性的声明，初始化 `App` 构造函数中的属性，并将 `MainPage` 属性初始化为 `NavigationPage`。使用 `PhoneNumbers` 集合存储应用程序呼叫的每个已翻译的电话号码列表：

```
using System.Collections.Generic;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public static IList<string> PhoneNumbers { get; set; }

        public App()
        {
            InitializeComponent();
            PhoneNumbers = new List<string>();
            MainPage = new NavigationPage(new MainPage());
        }
        ...
    }
}
```

通过按 **Ctrl+S**，保存对 **App.xaml.cs** 所做的更改，然后关闭文件。

7. 在“解决方案资源管理器”中，双击共享“Phoneword”项目中的“MainPage.xaml”文件打开该文件：

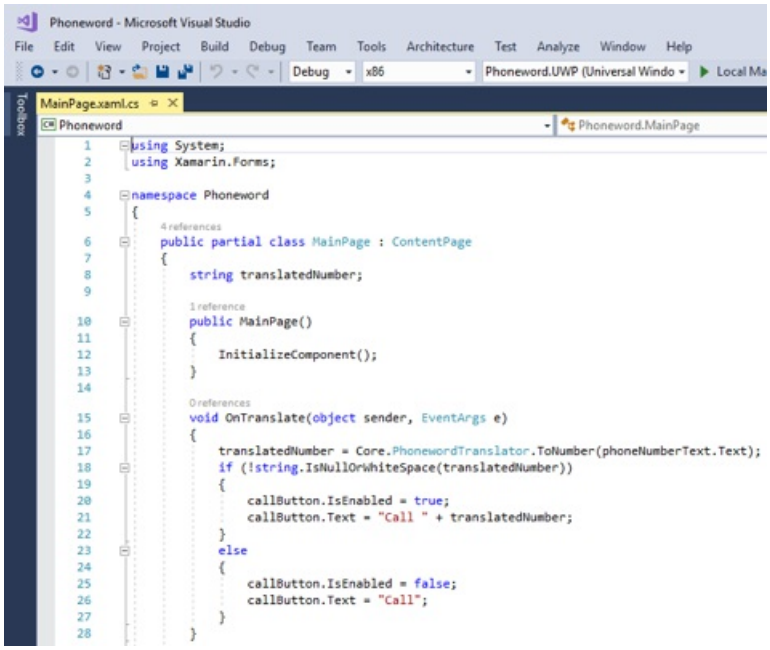


8. 在 **MainPage.xaml** 中，在 `StackLayout` 控件末尾处添加 `Button` 控件。此按钮用于导航到呼叫历史记录页：

```
<StackLayout VerticalOptions="FillAndExpand"
             HorizontalOptions="FillAndExpand"
             Orientation="Vertical"
             Spacing="15">
  ...
  <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
  <Button x:Name="callHistoryButton" Text="Call History" IsEnabled="false"
          Clicked="OnCallHistory" />
</StackLayout>
```

按 **Ctrl+S**，保存对 **MainPage.xaml** 所做的更改，然后关闭文件。

9. 在“解决方案资源管理器”中，双击 **MainPage.xaml.cs** 将其打开：



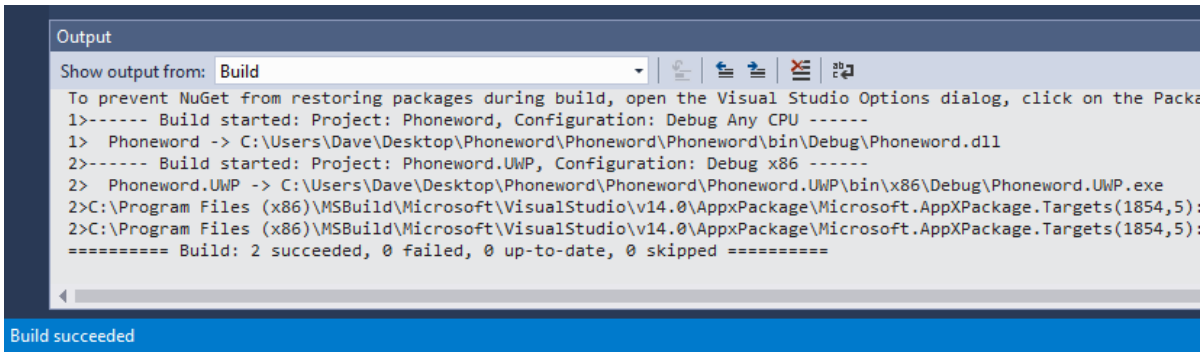
10. 在 **MainPage.xaml.cs** 中，添加 `OnCallHistory` 事件处理程序方法，并修改 `OnCall` 事件处理程序方法，从

而将已翻译的电话号码添加到 `App.PhoneNumbers` 集合并启用 `callHistoryButton`（前提是 `dialer` 变量不为 `null`）：

```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        ...

        async void OnCall(object sender, EventArgs e)
        {
            ...
            if (dialer != null) {
                App.PhoneNumbers.Add (translatedNumber);
                callHistoryButton.IsEnabled = true;
                dialer.Dial (translatedNumber);
            }
            ...
        }

        async void OnCallHistory(object sender, EventArgs e)
        {
            await Navigation.PushAsync (new CallHistoryPage ());
        }
    }
}
```

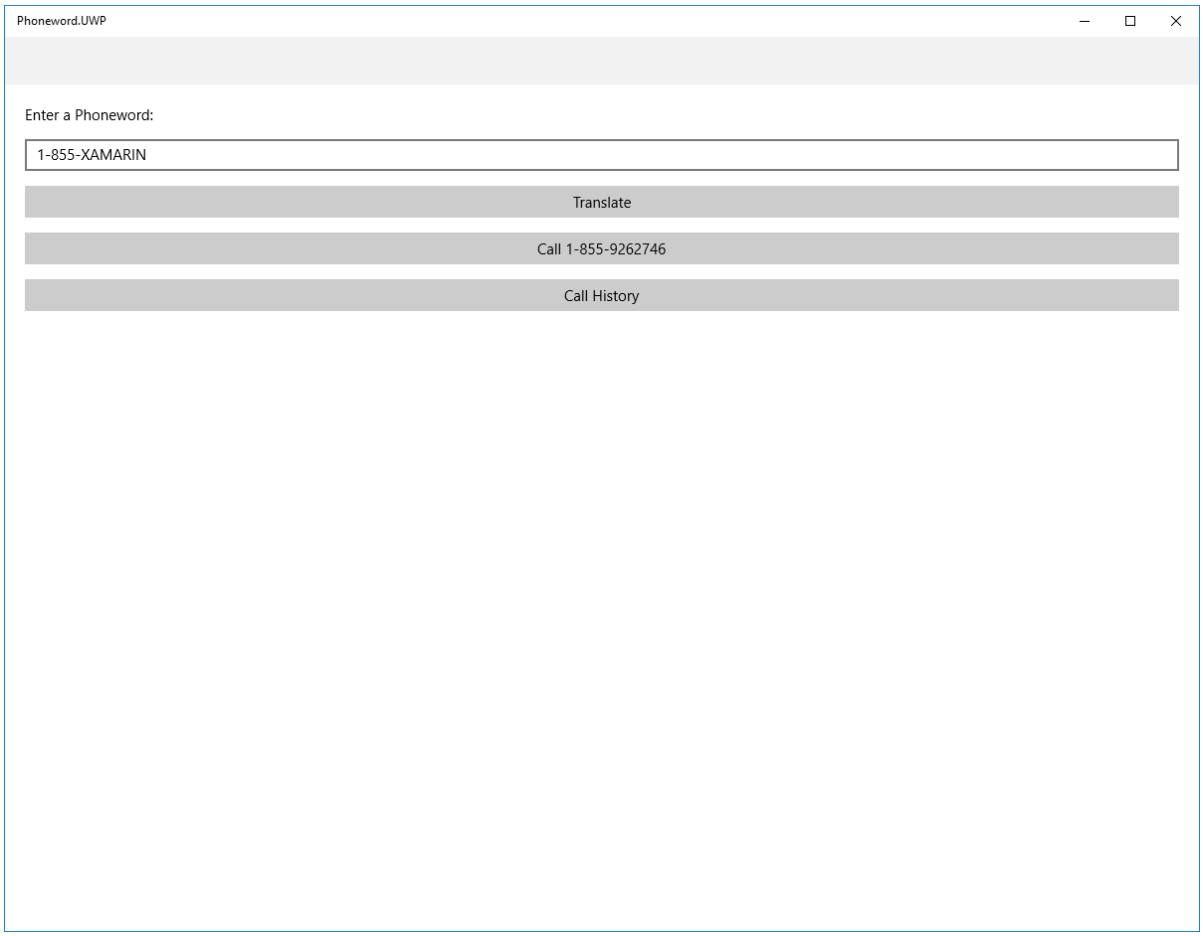
按 **Ctrl+S**，保存对 **MainPage.xaml.cs** 所做的更改，然后关闭文件。

11. 在 Visual Studio 中，选择“生成”>“生成解决方案”菜单项，或按 **Ctrl+Shift+B**。将生成应用程序，Visual Studio 状态栏中将显示一条成功消息：



如果发生错误，请重复前面的步骤并更正任何错误，直到成功生成应用程序。

12. 在 Visual Studio 工具栏中，按“开始”按钮（类似“播放”按钮的三角形按钮），启动应用程序：

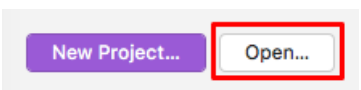


13. 在“解决方案资源管理器”中，右键单击“Phoneword.Droid”项目，然后选择“设为启用项目”。
14. 在 Visual Studio 工具栏中，按“开始”按钮（类似“播放”按钮的三角形按钮），启动 Android 模拟器内的应用程序。
15. 如果拥有 iOS 设备并符合 Xamarin.Forms 开发的 Mac 系统要求，请使用类似技术将应用部署到 iOS 设备。或者，将应用部署到 iOS 远程模拟器。

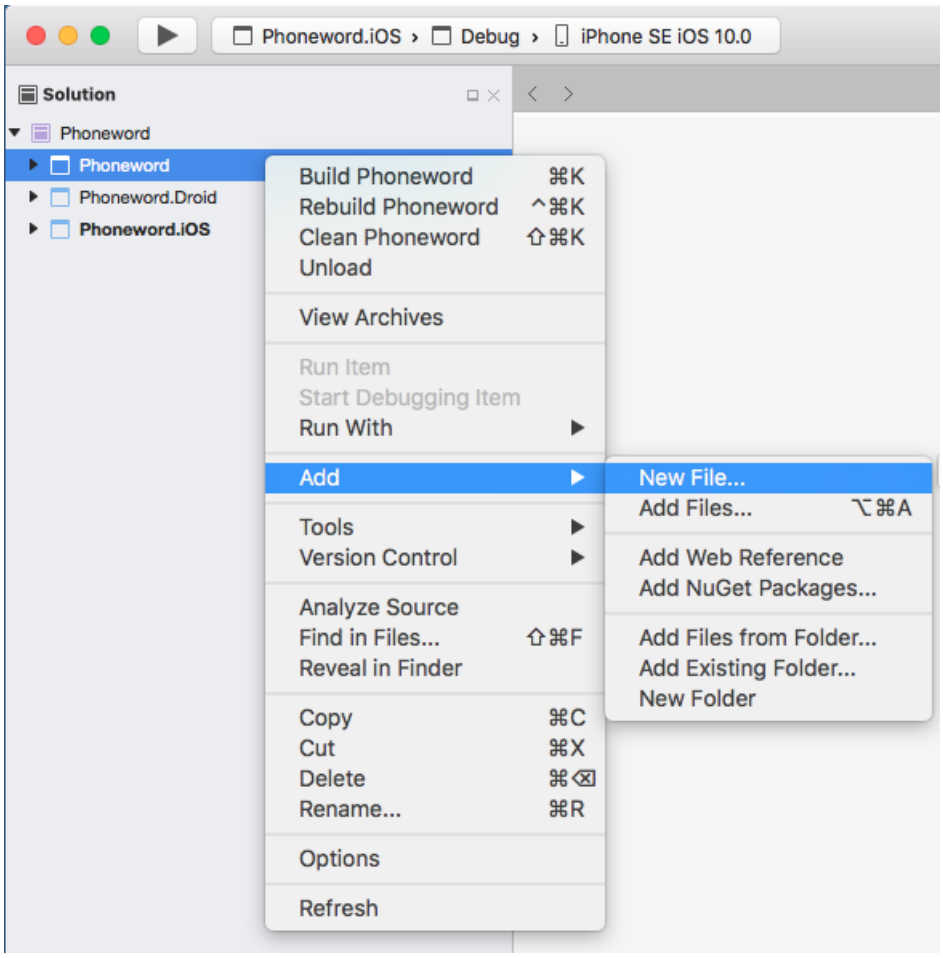
> **NOTE**
> 设备仿真器不支持电话呼叫。

使用 Visual Studio for Mac 更新应用

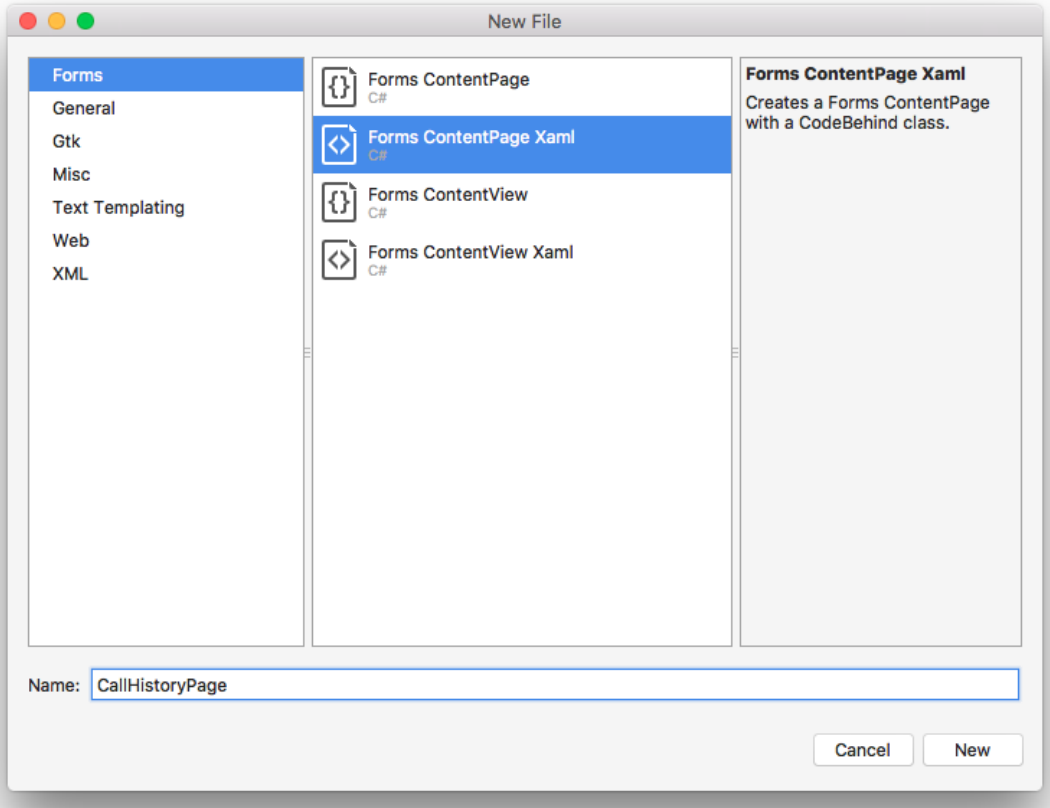
1. 启动 Visual Studio for Mac。在起始页上单击“打开...”，然后在对话框中选择 Phoneword 项目的解决方案文件：



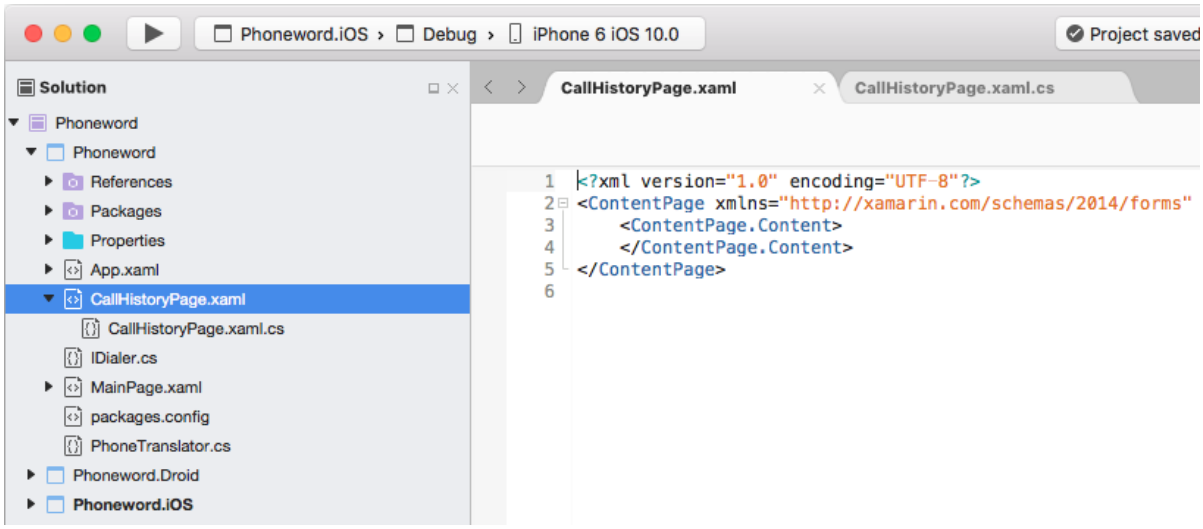
2. 在解决方案面板中，选择“Phoneword”项目，右键单击并选择“添加”>“新文件...”：



3. 在"新建文件"对话框中，选择"Forms">"Forms ContentPage Xaml"，将新文件命名为 **CallHistoryPage**，然后单击"新建"按钮。这会将一个名为 **CallHistoryPage** 的页面添加到项目：



4. 在“Solution Pad”中，双击 **CallHistoryPage.xaml** 将其打开：

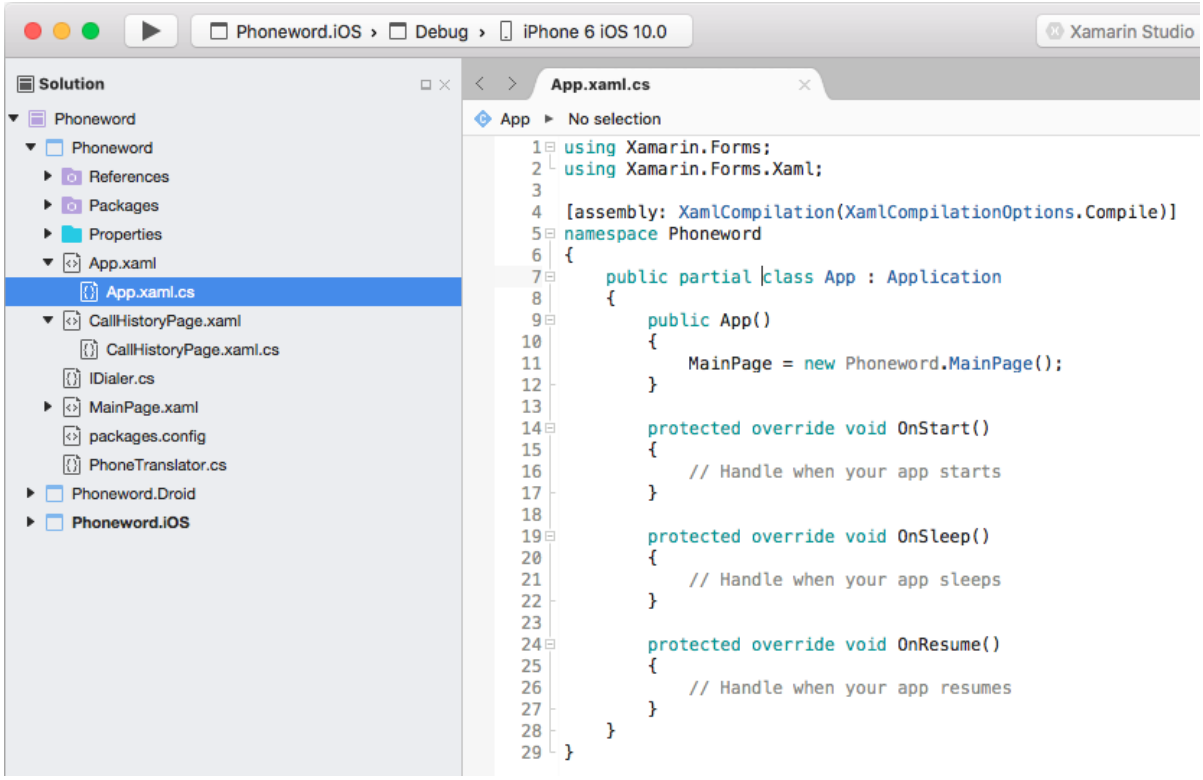


5. 在 **CallHistoryPage.xaml** 中，删除所有模板代码并将其替换为以下代码。此代码以声明方式定义页面上的用户界面：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="Phoneword.CallHistoryPage"
                  Title="Call History">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="20, 40, 20, 20" />
            <On Platform="Android, UWP" Value="20" />
        </OnPlatform>
    </ContentPage.Padding>
    <StackLayout>
      <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
    </StackLayout>
</ContentPage>
```

通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **CallHistoryPage.xaml** 所做的更改，然后关闭文件。

6. 在“Solution Pad”中，双击 **App.xaml.cs**将其打开：

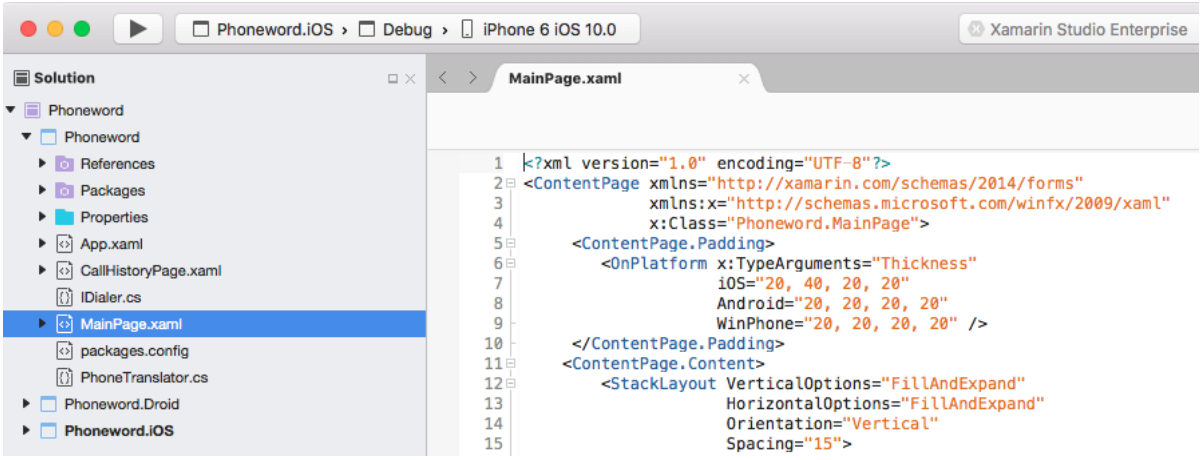


7. 在 **App.xaml.cs** 中，导入 `System.Collections.Generic` 命名空间，添加 `PhoneNumbers` 属性的声明，初始化 `App` 构造函数中的属性，并将 `MainPage` 属性初始化为 `NavigationPage`。使用 `PhoneNumbers` 集合存储应用程序呼叫的每个已翻译的电话号码列表：

```
using System.Collections.Generic;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

[assembly: XamlCompilation(XamlCompilationOptions.Compile)]
namespace Phoneword
{
    public partial class App : Application
    {
        public static IList<string> PhoneNumbers { get; set; }

        public App()
        {
            InitializeComponent();
            PhoneNumbers = new List<string>();
            MainPage = new NavigationPage(new MainPage());
        }
        ...
    }
}
```

通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **App.xaml.cs** 所做的更改，然后关闭文件。

8. 在“Solution Pad”中，双击 **MainPage.xaml**将其打开：

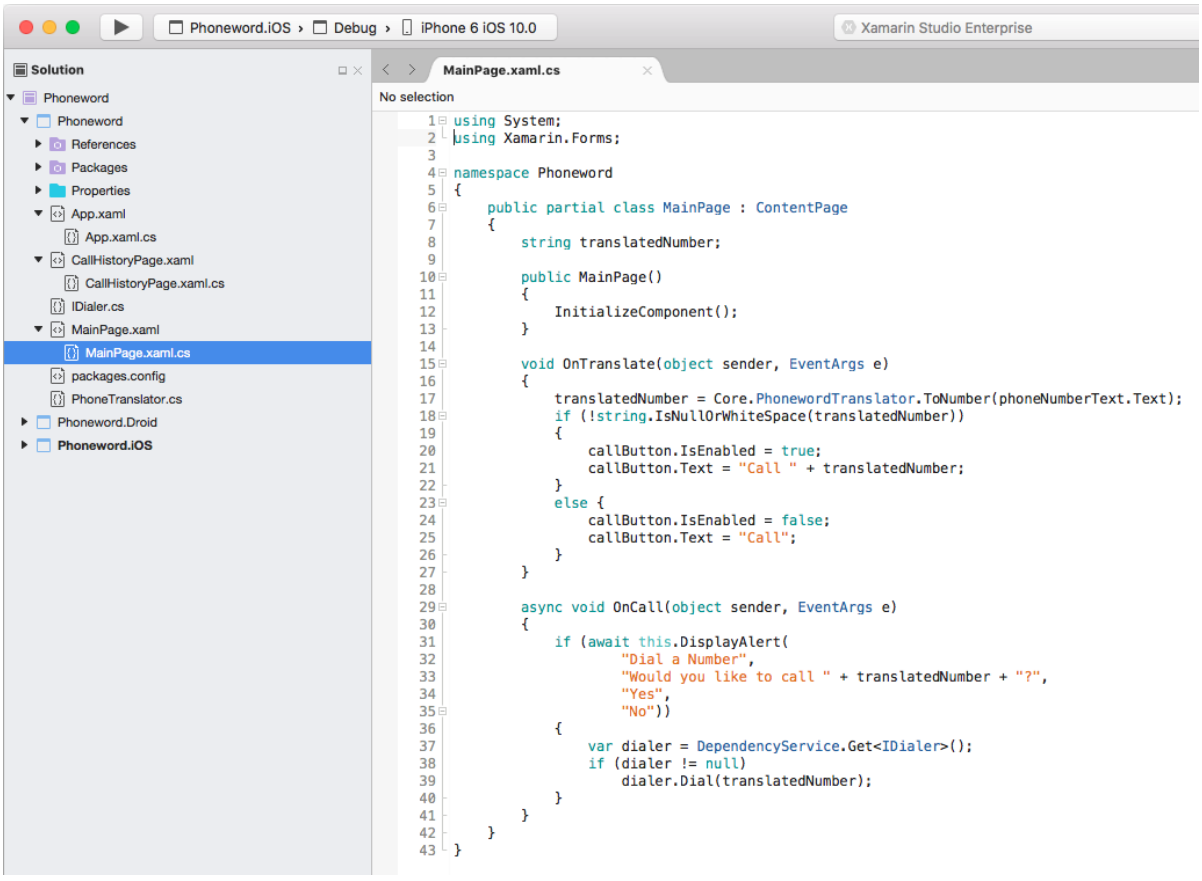


9. 在 **MainPage.xaml** 中，在 `StackLayout` 控件末尾处添加 `Button` 控件。此按钮用于导航到呼叫历史记录页：

```
<StackLayout VerticalOptions="FillAndExpand"
             HorizontalOptions="FillAndExpand"
             Orientation="Vertical"
             Spacing="15">
  ...
  <Button x:Name="callButton" Text="Call" IsEnabled="false" Clicked="OnCall" />
  <Button x:Name="callHistoryButton" Text="Call History" IsEnabled="false"
          Clicked="OnCallHistory" />
</StackLayout>
```

通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **MainPage.xaml** 所做的更改，然后关闭文件。

10. 在“Solution Pad”中，双击 **MainPage.xaml.cs** 将其打开：



11. 在 **MainPage.xaml.cs** 中，添加 `OnCallHistory` 事件处理程序方法，并修改 `OnCall` 事件处理程序方法，从而将已翻译的电话号码添加到 `App.PhoneNumbers` 集合并启用 `callHistoryButton`（前提是 `dialer` 变量不为

`null`）：

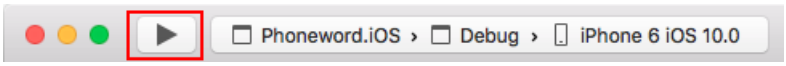
```
using System;
using Xamarin.Forms;

namespace Phoneword
{
    public partial class MainPage : ContentPage
    {
        ...

        async void OnCall(object sender, EventArgs e)
        {
            ...
            if (dialer != null) {
                App.PhoneNumbers.Add (translatedNumber);
                callHistoryButton.IsEnabled = true;
                dialer.Dial (translatedNumber);
            }
            ...
        }

        async void OnCallHistory(object sender, EventArgs e)
        {
            await Navigation.PushAsync (new CallHistoryPage ());
        }
    }
}
```

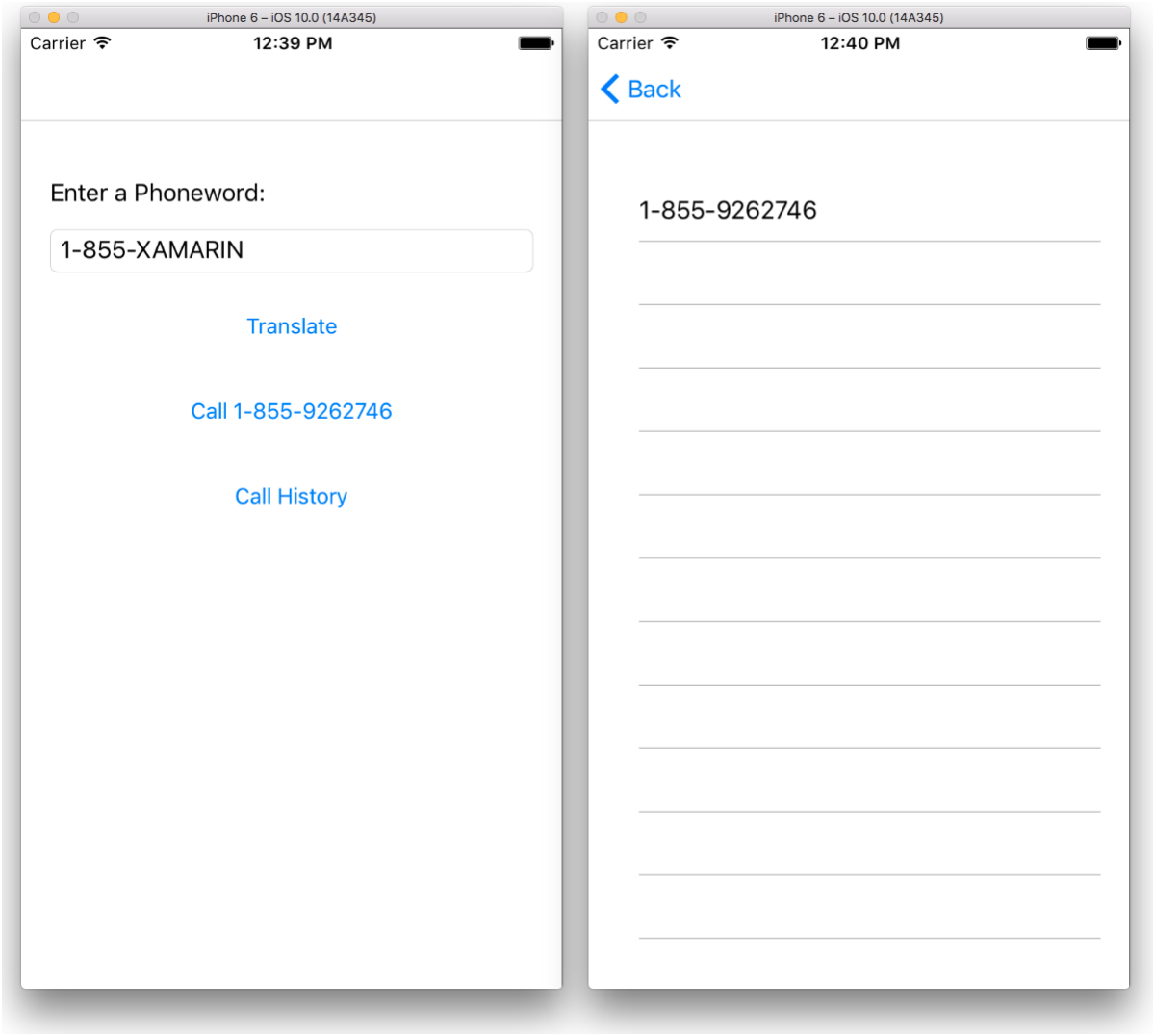
通过选择“文件”>“保存”，或按 ⌘ + **S**，保存对 **MainPage.xaml.cs** 所做的更改，然后关闭文件。

12. 在 Visual Studio for Mac 中，选择“生成”>“生成所有”菜单项，或按 ⌘ + **B**。应用程序将生成，并会在 Visual Studio for Mac 工具栏中显示一条成功消息：

如果发生错误，请重复前面的步骤并更正任何错误，直到成功生成应用程序。

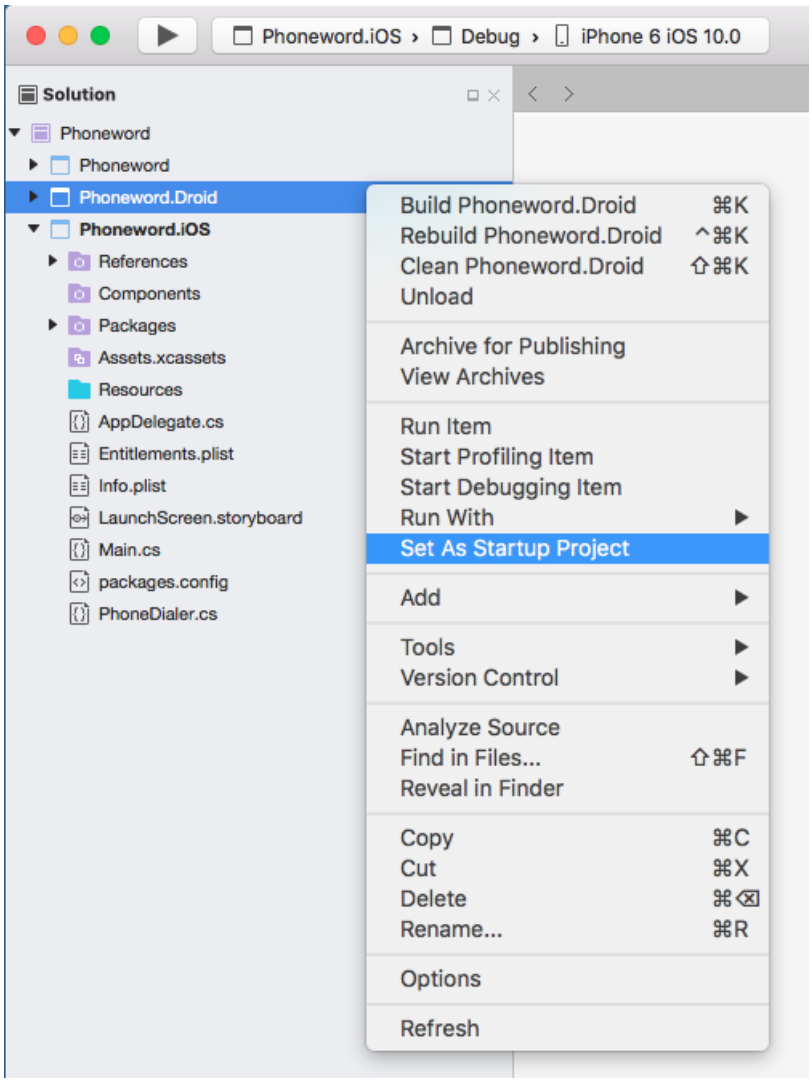
13. 在 Visual Studio for Mac 工具栏中，按“开始”按钮（类似“播放”按钮的三角形按钮），启动 iOS 模拟器内的应用程序：



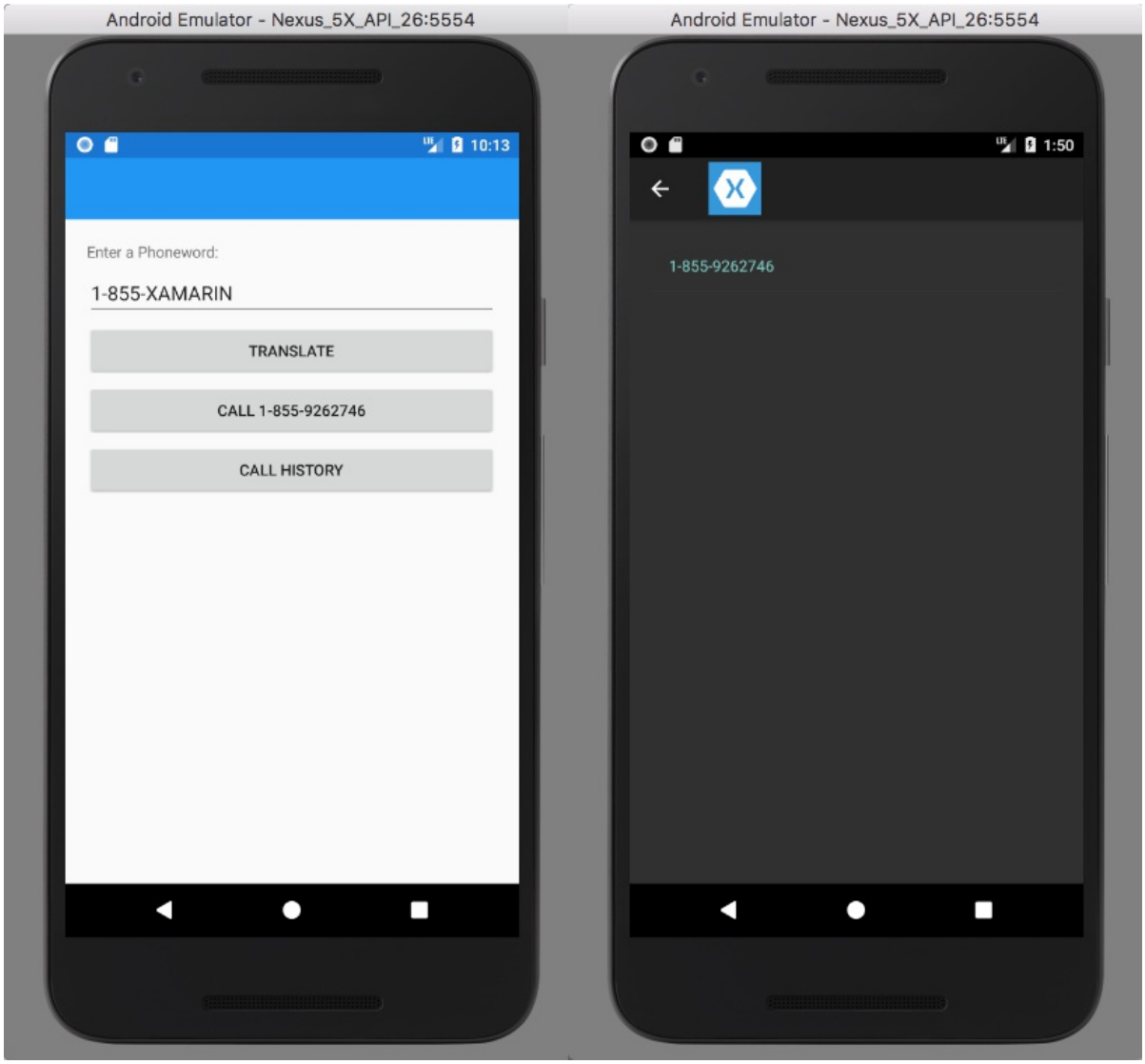


注意：iOS 模拟器不支持电话呼叫。

14. 在**解决方案面板**中，选择"Phoneword.Droid"项目，右键单击并选择"设为启动项目"：



15. 在 Visual Studio for Mac 工具栏中，按“开始”按钮（类似“播放”按钮的三角形按钮），启动 Android 模拟器内的应用程序：



> **NOTE**
> 设备仿真器不支持电话呼叫。

祝贺你！你已完成多屏幕 Xamarin.Forms 应用程序。本指南的下一个主题将回顾此演练中采用的步骤，以深入了解使用 Xamarin.Forms 进行页面导航和数据绑定的知识。

相关链接

- Phoneword（示例）
- PhonewordMultiscreen（示例）

Xamarin.Forms 多屏显示深度解析

2018/7/13 • • Edit Online

在 Xamarin.Forms 多屏显示快速入门中，Phoneword 应用程序扩展到包括第二个屏幕，它可跟踪应用程序的调用历史记录。本文对已生成的内容进行回顾，以深入了解 Xamarin.Forms 应用程序中的页面导航和数据绑定。

导航

Xamarin.Forms 提供了内置导航模型，用于管理页面堆栈的导航和用户体验。此模型实现了 `Page` 对象的后进先出 (LIFO) 堆栈。若要从一个页面移动到另一个页面，应用程序会将新页面推送到此堆栈上。若要返回上一页，应用程序将从堆栈弹出当前页。

Xamarin.Forms 具有 `NavigationPage` 类，用于管理 `Page` 对象的堆栈。`NavigationPage` 类还会将导航栏添加到页面顶部，此页面显示标题和平台相应的“返回”按钮，通过此按钮可返回上一页。以下代码示例演示如何围绕应用程序的第一页包装 `NavigationPage`：

```
public App ()
{
    ...
    MainPage = new NavigationPage (new MainPage ());
}
```

所有 `ContentPage` 实例都具有 `Navigation` 属性，可提供修改页面堆栈的方法。应仅当应用程序包括 `NavigationPage` 时，才调用这些方法。若要导航到 `CallHistoryPage`，则必须调用 `PushAsync` 方法，如下面的代码示例中所示：

```
async void OnCallHistory(object sender, EventArgs e)
{
    await Navigation.PushAsync (new CallHistoryPage ());
}
```

这将导致新的 `CallHistoryPage` 对象推送到导航堆栈上。若要以编程方式返回原始页，`CallHistoryPage` 对象必须调用 `PopAsync` 方法，如下面的代码示例中所示：

```
await Navigation.PopAsync();
```

但是，在 Phoneword 应用程序中无需此代码，因为 `NavigationPage` 类将导航栏添加到页面顶部，此页面包括平台相应的“返回”按钮，通过此按钮可返回上一页。

数据绑定

数据绑定用于简化 Xamarin.Forms 应用程序显示及其与数据的交互方式。它将在用户界面和基础应用程序之间建立连接。`BindableObject` 类包含大部分基础结构以支持数据绑定。

数据绑定可定义两个对象之间的关系。源对象提供数据。目标对象使用(并通常显示)源对象中的数据。在 Phoneword 应用程序中，绑定目标是显示电话号码的 `ListView` 控件，`PhoneNumbers` 集合是绑定源。

`PhoneNumbers` 集合在 `App` 类中进行声明和初始化，如下面的代码示例中所示：

```
public partial class App : Application
{
   public static List<string> PhoneNumbers { get; set; }

   public App ()
   {
     PhoneNumbers = new List<string>();
     ...
   }
   ...
}
```

`ListView` 实例在 `CallHistoryPage` 类中进行声明和初始化，如下面的代码示例中所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage ...
             xmlns:local="clr-namespace:Phoneword;assembly=Phoneword"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             ...>
    ...
    <ContentPage.Content>
       ...
       <ListView ItemsSource="{x:Static local:App.PhoneNumbers}" />
       ...
    </ContentPage.Content>
</ContentPage>
```

在此示例中，`ListView` 控件将显示 `ItemsSource` 属性绑定到的数据的 `IEnumerable` 集合。数据集合可以是任何类型的对象，但默认情况下，`ListView` 将使用每个项的 `ToString` 方法来显示该项。`x:Static` 标记扩展用于指示 `ItemsSource` 属性将绑定到 `App` 类（可在 `local` 命名空间中找到）的静态 `PhoneNumbers` 属性。

若要深入了解数据绑定，请参阅数据绑定基本知识。若要深入了解 XAML 标记扩展，请参阅 XAML 标记扩展。

Phoneword 中引入的其他概念

`ListView` 负责在屏幕上显示项的集合。单个单元格中包含 `ListView` 中的每个项。若要深入了解如何使用 `ListView` 控件，请参阅 ListView。

总结

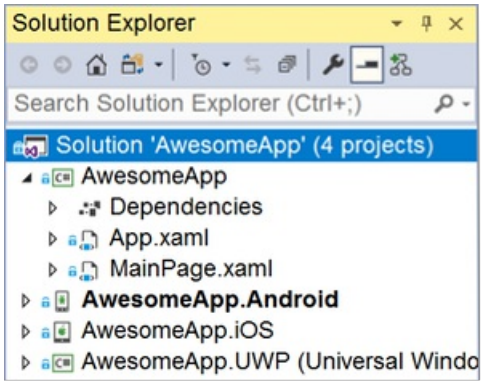
本文介绍了 Xamarin.Forms 应用程序中的页面导航和数据绑定，并演示了如何在多屏幕跨平台应用程序中使用它们。

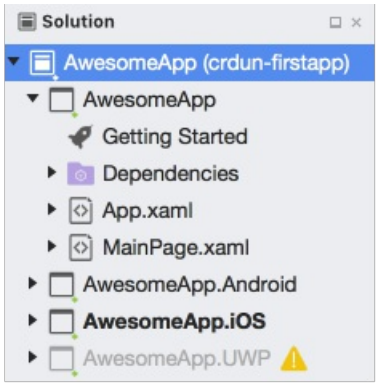
Xamarin.Forms 简介

2018/11/2 • • Edit Online

Xamarin.Forms 是一种框架，开发人员可以使用它生成适用于 Android、iOS 和 Windows 的跨平台应用程序。平台之间将共享代码和用户界面定义，但使用本机控件呈现它们。本文介绍了 Xamarin.Forms 以及如何开始在 Visual Studio 中使用 C# 和 XAML 编写应用程序。

Xamarin.Forms 应用程序使用 .NET Standard 项目包含共享代码，然后将应用程序项目分离，以便使用共享代码并生成各个平台所需的输出。创建新的 Xamarin.Forms 应用时，解决方案将包含共享代码项目（包括 C# 和 XAML 文件）及特定于平台的项目，如此屏幕截图所示：

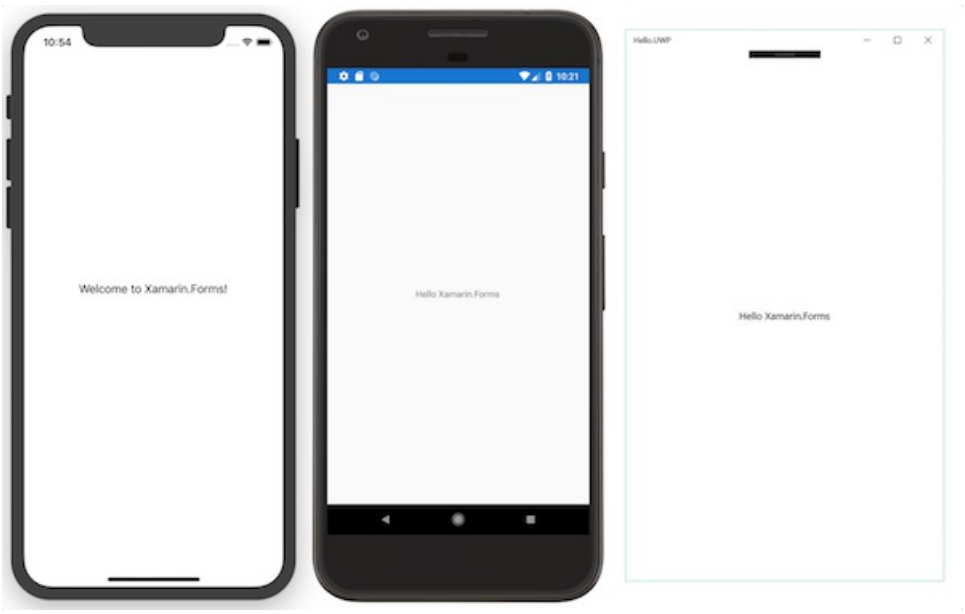




编写 Xamarin.Forms 应用时，代码和用户界面将添加到顶部的 .NET Standard 项目中，Android、iOS 和 UWP 项目均引用此项目。生成并运行 Android、iOS 和 UWP 项目以测试和部署应用。

检查 Xamarin.Forms 应用程序

Visual Studio 中的默认 Xamarin.Forms 应用模板显示单个文本标签。如果运行该应用程序，它应类似于以下屏幕截图：



屏幕截图中的每个屏幕对应于 Xamarin.Forms 中的一个页面。`Page` 在 Android 表示为一个活动，在 iOS 中表示为一个视图控制器，在 Windows 通用平台 (UWP) 中则表示为一个页面。以上屏幕截图中的示例实例化 `ContentPage` 对象，并使用该对象显示 `Label`。

为了最大限度重用启动代码，Xamarin.Forms 应用程序有一个名为 `App` 的单个类，该类负责实例化第一个将要

显示的 `Page`。`App` 类的示例可以在以下代码中看到(在 App.xaml.cs 中):

```
public partial class App : Application
{
  public App ()
  {
    InitializeComponent();
    MainPage = new MainPage(); // sets the App.MainPage property to an instance of the MainPage class
  }
}
```

此代码实例化名称为 `MainPage` 的新 `ContentPage` 对象,该对象将在页面上垂直和水平居中显示单个 `Label`。MainPage.xaml 文件中的 XAML 如下所示:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:AwesomeApp"
x:Class="AwesomeApp.MainPage">
    <StackLayout>
        <Label Text="Hello Xamarin.Forms"
           HorizontalOptions="Center"
           VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

在每个平台上启动 Xamarin.Forms 初始页面

> **TIP**
>
> 在此部分中提供了特定于平台的信息,以便于了解 Xamarin.Forms 的工作方式。项目模板已包含这些类;无需自己为它们编码。
>
> 可以跳至用户界面部分,稍后再阅读此部分。

若要在应用程序内部使用某个页面(如上面示例中的 MainPage),每个平台应用程序启动时必须初始化 Xamarin.Forms 框架并提供该页面的一个实例。初始化步骤因平台而异,此内容将在以下部分讨论。

iOS

若要在 iOS 中启动 Xamarin.Forms 初始页面,平台项目应包括 `AppDelegate` 类(该类从 `Xamarin.Forms.Platform.iOS.FormsApplicationDelegate` 类继承),如以下代码示例所示:

```
[Register("AppDelegate")]
public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
{
    public override bool FinishedLaunching(UIApplication app, NSDictionary options)
    {
      global::Xamarin.Forms.Forms.Init ();
      LoadApplication (new App ());
      return base.FinishedLaunching (app, options);
    }
}
```

通过调用 `Init` 方法,`FinishedLaunching` 替代初始化 Xamarin.Forms 框架。这会导致在调用将根视图控制器设置为 `LoadApplication` 方法之前,将特定于 iOS 的 Xamarin.Forms 实现加载到应用程序。

Android

若要在 Android 中启动 Xamarin.Forms 初始页面,平台项目应包括可使用 `MainLauncher` 属性创建 `Activity` 的代码,且具有从 `FormsAppCompatActivity` 类继承的活动,如以下代码示例所示:

```
namespace HelloXamarinFormsWorld.Android
{
    [Activity(Label = "HelloXamarinFormsWorld", Theme = "@style/MainTheme", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
    public class MainActivity : FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);
            Xamarin.Forms.Forms.Init(this, bundle);
            LoadApplication (new App ());
        }
    }
}
```

通过调用 `Init` 方法，`OnCreate` 替代初始化 Xamarin.Forms 框架。这会导致在加载 Xamarin.Forms 应用程序之前，将特定于 Android 的 Xamarin.Forms 实现加载到应用程序。

通用 Windows 平台 (UWP)

在通用 Windows 平台 (UWP) 应用程序中，可从 `App` 类调用初始化 Xamarin.Forms 框架的 `Init` 方法：

```
Xamarin.Forms.Forms.Init (e);

if (e.PreviousExecutionState == ApplicationExecutionState.Terminated)
{
  ...
}
```

这会将特定于 UWP 的 Xamarin.Forms 实现加载到应用程序。可通过 `MainPage` 类启动 Xamarin.Forms 初始页面，如以下代码示例所示：

```
public partial class MainPage
{
    public MainPage()
    {
      this.InitializeComponent();
      this.LoadApplication(new HelloXamarinFormsWorld.App());
    }
}
```

可通过 `LoadApplication` 方法加载 Xamarin.Forms 应用程序。创建新的 Xamarin.Forms 项目时，Visual Studio 会添加以上所有代码。

用户界面

在 Xamarin.Forms 中创建用户界面可以采用两种方法：

- 完全使用 C# 源代码创建用户界面。
- Extensible Application Markup Language (XAML)，一种用于描述用户界面的声明性标记语言。

无论使用哪种方法，均可获得相同的结果（并且下文对两种方法均提供了说明）。有关 Xamarin.Forms XAML 的详细信息，请参阅 XAML 基础。

视图和布局

可使用 4 个主要控件组创建 Xamarin.Forms 应用程序的用户界面。

- 页面 - Xamarin.Forms 页呈现跨平台移动应用程序屏幕。有关页面的详细信息，请参阅 Xamarin.Forms 页面。

- **布局** - Xamarin.Forms 布局是用于将视图组合到逻辑结构的容器。有关布局的详细信息，请参阅 Xamarin.Forms 布局。
- **视图** - Xamarin.Forms 视图是显示在用户界面上的控件，如标签、按钮和文本输入框。有关视图的详细信息，请参阅 Xamarin.Forms 视图。
- **单元格** - Xamarin.Forms 单元格是专门用于列表中的项的元素，描述列表中每个项的绘制方式。有关单元格的详细信息，请参阅 Xamarin.Forms 单元格。

在运行时，每个控件都会映射到其本身的本机等效项（即屏幕呈现的内容）。

控件在布局内部进行托管。下文介绍了一种常用的布局 – `StackLayout` 类。

StackLayout

`StackLayout` 在屏幕上自动排列控件而不考虑屏幕大小，从而简化了跨平台应用程序开发。根据添加顺序，以垂直方式或水平方式逐个放置每个子元素。`StackLayout` 使用的空间大小取决于 `HorizontalOptions` 和 `VerticalOptions` 属性的设置方式，但默认情况下，`StackLayout` 尝试使用整个屏幕。

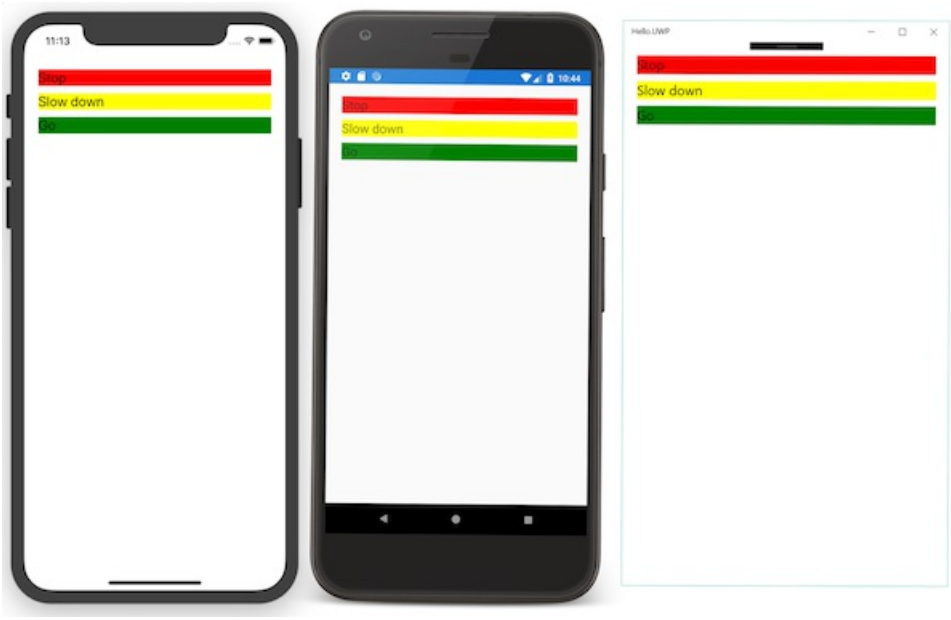
以下 XAML 代码举例说明了如何使用 `StackLayout` 排列三个 `Label` 控件：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample1" Padding="20">
  <StackLayout Spacing="10">
    <Label Text="Stop" BackgroundColor="Red" Font="20" />
    <Label Text="Slow down" BackgroundColor="Yellow" Font="20" />
    <Label Text="Go" BackgroundColor="Green" Font="20" />
  </StackLayout>
</ContentPage>
```

以下代码示例显示相应的 C# 代码：

```
public class StackLayoutExample : ContentPage
{
    public StackLayoutExample()
    {
        Padding = new Thickness(20);
        var red = new Label
        {
            Text = "Stop", BackgroundColor = Color.Red, FontSize = 20
        };
        var yellow = new Label
        {
            Text = "Slow down", BackgroundColor = Color.Yellow, FontSize = 20
        };
        var green = new Label
        {
            Text = "Go", BackgroundColor = Color.Green, FontSize = 20
        };

        Content = new StackLayout
        {
            Spacing = 10,
            Children = { red, yellow, green }
        };
    }
}
```

默认情况下，`StackLayout` 采用垂直方向，如以下屏幕截图所示：



可以将 `StackLayout` 更改为水平方向，如以下 XAML 代码示例所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample2" Padding="20">
  <StackLayout Spacing="10" VerticalOptions="End" Orientation="Horizontal" HorizontalOptions="Start">
    <Label Text="Stop" BackgroundColor="Red" Font="20" />
    <Label Text="Slow down" BackgroundColor="Yellow" Font="20" />
    <Label Text="Go" BackgroundColor="Green" Font="20" />
  </StackLayout>
</ContentPage>
```

以下代码示例显示相应的 C# 代码：

```
public class StackLayoutExample: ContentPage
{
    public StackLayoutExample()
    {
        // Code that creates red, yellow, green labels removed for clarity (see above)
        Content = new StackLayout
        {
            Spacing = 10,
            VerticalOptions = LayoutOptions.End,
            Orientation = StackOrientation.Horizontal,
            HorizontalOptions = LayoutOptions.Start,
            Children = { red, yellow, green }
        };
    }
}
```

以下屏幕截图显示布局结果：

可通过 `HeightRequest` 和 `WidthRequest` 属性设置控件大小，如以下 XAML 代码示例所示：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
x:Class="HelloXamarinFormsWorldXaml.StackLayoutExample3" Padding="20">
  <StackLayout Spacing="10" VerticalOptions="End" Orientation="Horizontal" HorizontalOptions="Start">
    <Label Text="Stop" BackgroundColor="Red" Font="20" WidthRequest="100" />
    <Label Text="Slow down" BackgroundColor="Yellow" Font="20" WidthRequest="100" />
    <Label Text="Go" BackgroundColor="Green" Font="20" WidthRequest="200" />
  </StackLayout>
</ContentPage>
```

以下代码示例显示相应的 C# 代码：

```
var red = new Label
{
    Text = "Stop", BackgroundColor = Color.Red, FontSize = 20, WidthRequest = 100
};
var yellow = new Label
{
    Text = "Slow down", BackgroundColor = Color.Yellow, FontSize = 20, WidthRequest = 100
};
var green = new Label
{
    Text = "Go", BackgroundColor = Color.Green, FontSize = 20, WidthRequest = 200
};

Content = new StackLayout
{
    Spacing = 10,
    VerticalOptions = LayoutOptions.End,
    Orientation = StackOrientation.Horizontal,
    HorizontalOptions = LayoutOptions.Start,
    Children = { red, yellow, green }
};
```

以下屏幕截图显示布局结果：

有关 `StackLayout` 类的详细信息，请参阅 StackLayout。

Xamarin.Forms 中的列表

`ListView` 控件负责在屏幕上显示项集合 - `ListView` 中的每一个项都包含在单个单元格中。默认情况下，`ListView` 使用内置 `TextCell` 模板并呈现单行文本。

以下代码示例演示一个简单的 `ListView` 示例：

```
var listView = new ListView
{
    RowHeight = 40
};
listView.ItemsSource = new string []
{
    "Buy pears", "Buy oranges", "Buy mangos", "Buy apples", "Buy bananas"
};
Content = new StackLayout
{
    VerticalOptions = LayoutOptions.FillAndExpand,
    Children = { listView }
};
```

以下屏幕截图显示生成的 `ListView`：

有关 `ListView` 控件的详细信息，请参阅 ListView。

绑定到自定义类

`ListView` 控件还可以通过默认的 `TextCell` 模板显示自定义对象。

以下代码示例演示 `TodoItem` 类：

```
public class TodoItem
{
    public string Name { get; set; }
    public bool Done { get; set; }
}
```

可按如下代码示例所示填充 `ListView` 控件：

```
listView.ItemsSource = new TodoItem [] {
    new TodoItem { Name = "Buy pears" },
    new TodoItem { Name = "Buy oranges", Done=true} ,
    new TodoItem { Name = "Buy mangos" },
    new TodoItem { Name = "Buy apples", Done=true },
    new TodoItem { Name = "Buy bananas", Done=true }
};
```

可以创建一个绑定以设置 `ListView` 要显示的 `TodoItem` 属性类型，如以下代码示例所示：

```
listView.ItemTemplate = new DataTemplate(typeof(TextCell));
listView.ItemTemplate.SetBinding(TextCell.TextProperty, "Name");
```

这会创建一个绑定，指定 `TodoItem.Name` 属性的路径，并生成前面显示的屏幕截图。

有关绑定到自定义类的详细信息，请参阅 ListView 数据源。

选择 ListView 中的项

为响应用户触摸 `ListView` 中的单元格，应处理 `ItemSelected` 事件，如以下代码示例所示：

```
listView.ItemSelected += async (sender, e) => {
    await DisplayAlert("Tapped!", e.SelectedItem + " was tapped.", "OK");
};
```

当包含在 `NavigationPage` 中时，`PushAsync` 方法可用来打开具有内置后退导航的新页面。`ItemSelected` 事件可以访问通过 `e.SelectedItem` 属性与单元格关联的对象，将其绑定到新页面并使用 `PushAsync` 显示新页面，如以下代码示例所示：

```
listView.ItemSelected += async (sender, e) => {
    var todoItem = (TodoItem)e.SelectedItem;
    var todoPage = new TodoItemPage(todoItem); // so the new page shows correct data
    await Navigation.PushAsync(todoPage);
};
```

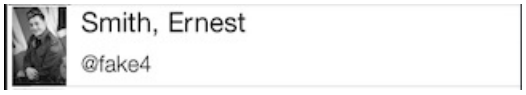
每个平台实现内置后退导航的方法都各不相同。有关详细信息，请参阅导航。

有关 `ListView` 选择的详细信息，请参阅 ListView 交互性。

自定义单元格的外观

通过子类化 `ViewCell` 类，并将该类的类型设置为 `ListView` 的 `ItemTemplate` 属性，可以自定义单元格的外观。

以下屏幕截图所示的单元格由一个 `Image` 和两个 `Label` 控件组成：



若要创建此自定义布局，应子类化 `ViewCell` 类，如以下代码示例所示：

```
class EmployeeCell : ViewCell
{
    public EmployeeCell()
    {
        var image = new Image
        {
            HorizontalOptions = LayoutOptions.Start
        };
        image.SetBinding(Image.SourceProperty, new Binding("ImageUri"));
        image.WidthRequest = image.HeightRequest = 40;

        var nameLayout = CreateNameLayout();
        var viewLayout = new StackLayout()
        {
           Orientation = StackOrientation.Horizontal,
           Children = { image, nameLayout }
        };
        View = viewLayout;
    }

    static StackLayout CreateNameLayout()
    {
        var nameLabel = new Label
        {
            HorizontalOptions= LayoutOptions.FillAndExpand
        };
        nameLabel.SetBinding(Label.TextProperty, "DisplayName");

        var twitterLabel = new Label
        {
           HorizontalOptions = LayoutOptions.FillAndExpand,
           Font = Fonts.Twitter
        };
        twitterLabel.SetBinding(Label.TextProperty, "Twitter");

        var nameLayout = new StackLayout()
        {
           HorizontalOptions = LayoutOptions.StartAndExpand,
           Orientation = StackOrientation.Vertical,
           Children = { nameLabel, twitterLabel }
        };
        return nameLayout;
    }
}
```

此段代码执行下列任务：

- 添加 `Image` 控件并将其绑定到 `Employee` 对象的 `ImageUri` 属性。若要深入了解数据绑定，请参阅数据绑定。
- 创建垂直方向的 `StackLayout` 来存放两个 `Label` 控件。`Label` 控件被绑定到 `DisplayName` 和 `Employee` 对象的 `Twitter` 属性。
- 创建 `StackLayout` 用于托管现有的 `Image` 和 `StackLayout`。使用水平方向排列其子级。

创建好自定义单元格后，可以通过将其包装在 `DataTemplate` 中由 `ListView` 控件使用，如以下代码示例所示：

```
List<Employee> myListOfEmployeeObjects = GetAListOfAllEmployees();
var listView = new ListView
{
    RowHeight = 40
};
listView.ItemsSource = myListOfEmployeeObjects;
listView.ItemTemplate = new DataTemplate(typeof(EmployeeCell));
```

此代码为 `ListView` 提供 `Employee` 的 `List`。可使用 `EmployeeCell` 类呈现每个单元格。`ListView` 将 `Employee` 对象作为其 `BindingContext` 传递给 `EmployeeCell`。

有关自定义单元格的外观的详细信息，请参阅单元格的外观。

使用 XAML 创建和自定义列表

以下代码示例演示上一部分中 `ListView` 的 XAML 等效项：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:XamarinFormsXamlSample;assembly=XamarinFormsXamlSample"
             xmlns:constants="clr-namespace:XamarinFormsSample;assembly=XamarinFormsXamlSample"
             x:Class="XamarinFormsXamlSample.Views.EmployeeListPage"
             Title="Employee List">
  <ListView x:Name="listView" IsVisible="false" ItemsSource="{x:Static local:App.Employees}"
ItemSelected="EmployeeListOnItemSelected">
    <ListView.ItemTemplate>
      <DataTemplate>
        <ViewCell>
          <ViewCell.View>
            <StackLayout Orientation="Horizontal">
              <Image Source="{Binding ImageUri}" WidthRequest="40" HeightRequest="40" />
              <StackLayout Orientation="Vertical" HorizontalOptions="StartAndExpand">
                <Label Text="{Binding DisplayName}" HorizontalOptions="FillAndExpand" />
                <Label Text="{Binding Twitter}" Font="{x:Static constants:Fonts.Twitter}"/>
              </StackLayout>
            </StackLayout>
          </ViewCell.View>
        </ViewCell>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</ContentPage>
```

此 XAML 定义包含 `ListView` 的 `ContentPage`。可通过 `ItemsSource` 属性设置 `ListView` 的数据源。在 `ListView.ItemTemplate` 元素内定义 `ItemsSource` 中每一行的布局。

数据绑定

数据绑定连接两个对象，即源和目标。源对象提供数据。目标对象使用（并经常显示）来自源对象的数据。例如，`Label`（目标对象）通常会将其 `Text` 属性绑定到源对象中的公共 `string` 属性。下图说明了这种绑定关系：

数据绑定的主要优点是让你无需再担心视图和数据源之间的数据同步。幕后的绑定框架源会将源对象中的更改自动推送到目标对象，且目标对象中的更改可选择性地推送回源对象。

创建数据绑定只需两个步骤：

- 目标对象的 `BindingContext` 属性必须设置为源。
- 必须在目标和源之间建立绑定。在 XAML 中，此过程可通过使用 `Binding` 标记扩展实现。在 C# 中，此过程可通过 `SetBinding` 方法实现。

若要深入了解数据绑定，请参阅数据绑定基本知识。

XAML

以下代码示例说明如何在 XAML 中执行数据绑定：

```
<Entry Text="{Binding FirstName}" ... />
```

在 `Entry.Text` 属性和源对象的 `FirstName` 属性之间建立绑定。`Entry` 控件中所做的更改将自动传播到 `employeeToDisplay` 对象。同样，如果更改了 `employeeToDisplay.FirstName` 属性，Xamarin.Forms 绑定引擎也会更新 `Entry` 控件的内容。这称为双向绑定。为了使双向绑定发挥作用，模型类必须实现 `INotifyPropertyChanged` 接口。

尽管可以在 XAML 中设置 `EmployeeDetailPage` 类的 `BindingContext` 属性，但此处在代码隐藏中设置 `Employee` 对象的实例：

```
public EmployeeDetailPage(Employee employee)
{
    InitializeComponent();
    this.BindingContext = employee;
}
```

虽然可以分别设置每个目标对象的 `BindingContext` 属性，但没有必要。`BindingContext` 是特殊属性，其所有子级都会继承该属性。因此，当 `ContentPage` 上的 `BindingContext` 设置为 `Employee` 实例时，`ContentPage` 的所有子级都具有相同的 `BindingContext`，并且都可绑定到 `Employee` 对象的公共属性。

C#

以下代码示例说明如何在 C# 中执行数据绑定：

```
public EmployeeDetailPage(Employee employeeToDisplay)
{
    this.BindingContext = employeeToDisplay;
    var firstName = new Entry()
    {
        HorizontalOptions = LayoutOptions.FillAndExpand
    };
    firstName.SetBinding(Entry.TextProperty, "FirstName");
    ...
}
```

向 `ContentPage` 构造函数传递 `Employee` 对象的一个实例，并将 `BindingContext` 设置为要绑定到的对象。实例化 `Entry` 控件，并在 `Entry.Text` 属性和源对象的 `FirstName` 属性之间设置绑定。`Entry` 控件中所做的更改将自动传播到 `employeeToDisplay` 对象。同样，如果更改了 `employeeToDisplay.FirstName` 属性，Xamarin.Forms 绑定引擎也会更新 `Entry` 控件的内容。这称为双向绑定。为了使双向绑定发挥作用，模型类必须实现 `INotifyPropertyChanged` 接口。

`SetBinding` 方法采用两个参数。第一个参数指定绑定类型的信息。第二个参数提供绑定内容或绑定方式的信息。大多数情况下，第二个参数只是一个字符串，持有 `BindingContext` 上的属性名称。使用下列语法可直接绑定到 `BindingContext`：

```
someLabel.SetBinding(Label.TextProperty, new Binding("."));
```

使用点语法指示 Xamarin.Forms 使用 `BindingContext`（而非 `BindingContext` 上的属性）作为数据源。如果 `BindingContext` 是简单类型（例如 `string` 或 `int`），此语法非常有用。

属性更改通知

创建绑定后，默认情况下目标对象只接收源对象的值。若要使 UI 与数据源同步，当源对象发生更改时，必须通过一种方法来通知目标对象。`INotifyPropertyChanged` 接口就提供了这种机制。当基础属性值发生更改时，实现此

接口可以通知任何数据绑定控件。

当它的某个属性更新为新值时，实现 `INotifyPropertyChanged` 的对象必须引发 `PropertyChanged` 事件，如以下代码示例所示：

```
public class MyObject : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    string _firstName;
    public string FirstName
    {
        get { return _firstName; }
        set
        {
            if (value.Equals(_firstName, StringComparison.Ordinal))
            {
                // Nothing to do - the value hasn't changed;
                return;
            }
            _firstName = value;
            OnPropertyChanged();

        }
    }

    void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        var handler = PropertyChanged;
        if (handler != null)
        {
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

当 `MyObject.FirstName` 属性发生更改时，会调用 `OnPropertyChanged` 法，从而引发 `PropertyChanged` 事件。为了避免不必要的事件触发，如果属性值没有发生更改，则不引发 `PropertyChanged` 事件。

请注意，在 `OnPropertyChanged` 方法中，`propertyName` 参数标有 `CallerMemberName` 属性。这可确保当使用 `null` 值调用 `OnPropertyChanged` 方法时，`CallerMemberName` 属性提供调用 `OnPropertyChanged` 的方法的名称。

导航

Xamarin.Forms 提供多种不同的页导航体验，具体取决于使用的 `Page` 类型。对于 `ContentPage` 实例，提供两种导航体验：

- 分层导航
- 模式导航

`CarouselPage` `MasterDetailPage` 和 `TabbedPage` 类提供替代导航体验。有关详细信息，请参阅导航。

分层导航

`NavigationPage` 类提供分层导航体验，用户可以随心所欲地向前或向后导航页面。此类将导航实现为 `Page` 对象的后进先出 (LIFO) 堆栈。

在分层导航中，使用 `NavigationPage` 类在 `ContentPage` 对象的堆栈内进行导航。若要从一页移动到另一页，应用程序会将新页推送到导航堆栈中，在堆栈中，该页会变为活动页。若要返回到前一页，应用程序会从导航堆栈弹出当前页，而使最顶层的页成为活动页。

添加到导航堆栈中的第一页称为应用程序的根页，以下代码示例显示了实现此过程的方法：

```
public App ()
{
    MainPage = new NavigationPage(new EmployeeListPage());
}
```

若要导航到 `LoginPage`，需要调用当前页的 `Navigation` 属性上的 `PushAsync` 方法，如以下代码示例所示：

```
await Navigation.PushAsync(new LoginPage());
```

这会将新的 `LoginPage` 对象推送到导航堆栈中，在堆栈中，它成为活动页。

通过设备上的返回按钮（无论是设备上的物理按钮还是屏幕按钮），可以从导航堆栈中弹出活动页。若要以编程方式返回到前一页，`LoginPage` 实例必须调用 `PopAsync` 方法，如以下代码示例所示：

```
await Navigation.PopAsync();
```

有关分层导航的详细信息，请参阅分层导航。

模式导航

Xamarin.Forms 支持模式页面。模式页面鼓励用户完成独立任务，在完成或取消该任务之前，不允许导航离开该任务。

模式页面可以是 Xamarin.Forms 支持的任何 `Page` 类型。若要显示模式页面，应用程序会将页面推送到导航堆栈中，在堆栈中，该页会变为活动页。若要返到回前一页，应用程序会从导航堆栈弹出当前页面，而使最顶层的页成为活动页。

可以由任何 `Page` 派生类型上的 `Navigation` 属性公开模式导航方法。也可使用 `Navigation` 属性公开 `ModalStack` 属性，并从中获得导航堆栈中的模式页面。但是，在模式导航中没有执行模式堆栈操作或弹出到根页的概念。这是因为基础平台普遍都不支持这些操作。

> **NOTE**
> 执行模式页面导航无需具有 `NavigationPage` 实例。

若要以模式方式导航到 `LoginPage`，需要调用当前页的 `Navigation` 属性上的 `PushModalAsync` 方法，如以下代码示例所示：

```
await Navigation.PushModalAsync(new LoginPage());
```

这会将 `LoginPage` 实例推送到导航堆栈中，在堆栈中，它成为活动页。

通过设备上的返回按钮（无论是设备上的物理按钮还是屏幕按钮），可以从导航堆栈中弹出活动页。若要以编程方式返回原始页，`LoginPage` 实例必须调用 `PopModalAsync` 方法，如以下代码示例所示：

```
await Navigation.PopModalAsync();
```

这会从导航堆栈中删除 `LoginPage` 实例，而使最顶层的页成为活动页。

有关模式导航的详细信息，请参阅模式页面。

后续步骤

本文介绍了如何开始编写 Xamarin.Forms 应用程序。建议的后续步骤包括了解以下功能：

- 控件模板让你在运行时能够轻松设计或重新设计应用程序页面的主题。有关详细信息，请参阅控件模板。
- 数据模板让你可以在支持的控件上定义数据表示形式。有关详细信息，请参阅数据模板。
- 共享代码可通过 `DependencyService` 类访问本机功能。有关详细信息，请参阅通过 DependencyService 访问本机功能。
- Xamarin.Forms 具有简单的消息传送服务，用于发送和接收消息，减少两个类之间的耦合。有关详细信息，请参阅通过 MessagingCenter 发布和订阅。
- 通过 `Renderer` 类可以在每个平台上以不同方式呈现每个页面、布局和控件，反过来又可以创建本机控件，在屏幕上排列该控件，并添加共享代码中指定的行为。开发人员可以实现自定义 `Renderer` 类，以自定义控件的外观和/或行为。有关详细信息，请参阅自定义呈现器。
- 还可以自定义每个平台上的本机控件的效果。通过子类化 `PlatformEffect` 控件在特定于平台的项目中创建效果，并将其附加到相应的 Xamarin.Forms 控件中使用。有关详细信息，请参阅效果。

此外，也可以阅读 Charles Petzold 撰写的使用 Xamarin.Forms 创建移动应用一书，了解有关 Xamarin.Forms 的详细信息。可获取此书的 PDF 版本或多种电子书格式的版本。

相关链接

- XAML 基础知识
- 控件引用
- 用户界面
- Xamarin.Forms 示例
- 入门示例
- Xamarin.Forms API 参考
- 免费自学教程(视频)

可扩展应用程序标记语言 (XAML)

2018/11/13 • • Edit Online

XAML 是一种声明性标记语言，可用于定义用户界面。使用 XAML 语法，而在单独的代码隐藏文件中定义的运行时行为的 XML 文件中定义的用户界面。

Evolve 2016：成为 **XAML Master**

> **NOTE**
> 试用XAML 标准预览版

XAML 基础知识

XAML 允许开发人员在 Xamarin.Forms 应用程序使用标记而不是代码中定义的用户界面。XAML 永远不会需要 Xamarin.Forms 程序中，但它是工具化程度，并通常会更直观地一致和比等效的代码更简洁。XAML 是非常适合于与常用的模型-视图-视图模型 (MVVM) 应用程序体系结构一起使用：XAML 定义通过基于 XAML 的数据绑定链接到 ViewModel 代码的视图。

XAML 编译

XAML 可以根据需要使用 XAML 编译器 (XAMLC) 直接编译为中间语言 (IL)。本文介绍如何使用 XAMLC 和它的好处。

XAML 预览程序

XAML 预览程序呈现页的并排方案使用 XAML 标记，从而可以查看您的呈现将如你所键入的用户界面的实时预览。

XAML 命名空间

XAML 使用 `xmlns` XML 属性的命名空间声明。本文介绍 XAML 命名空间语法，并演示如何声明 XAML 命名空间以访问的类型。

XAML 标记扩展

XAML 包括将属性设置为值或超出什么可以使用简单的字符串表示的对象的标记扩展。其中包括常量、静态属性和字段、资源字典和数据绑定引用。

字段修饰符

`x:FieldModifier` 命名空间属性指定为命名 XAML 元素生成的字段的访问级别。

传递参数

XAML 可用于将参数传递到非默认构造函数或工厂方法。本文演示如何使用可用于将参数传递到构造函数，以调用工厂方法，并指定泛型参数的类型的 XAML 属性。

可绑定属性

在 Xamarin.Forms 中，公共语言运行时 (CLR) 属性的功能扩展可绑定属性。可绑定属性是属性的特殊类型，其中 Xamarin.Forms 属性系统跟踪属性的值。本文介绍了可绑定属性，并演示如何创建并使用它们。

附加属性

附加的属性是特殊类型的一个类中定义，但附加到其他对象的可绑定属性和可识别为属性的 XAML 中包含的类和属性名称之间以句点分隔。本文介绍了附加属性，并演示如何创建并使用它们。

资源字典

XAML 资源是可以多次使用的对象的定义。一个 `ResourceDictionary` 允许在单个位置中，定义和重新整个 Xamarin.Forms 应用程序中使用的资源。本文演示了如何创建和使用 `ResourceDictionary`，以及如何合并一个 `ResourceDictionary` 到另一个。

Xamarin.Forms XAML 基础知识

2018/11/13 • • Edit Online

XAML (Extensible Application Markup Language) 允许开发人员在 Xamarin.Forms 应用程序中使用标记(而不是代码)来定义用户界面。XAML 永远不会需要在 Xamarin.Forms 程序中，但通常会更简洁和更直观地一致比等效的代码，并可能会非常有用。XAML 是非常适合于与常用的 MVVM（模型-视图-视图）应用程序体系结构一起使用：XAML 定义通过基于 XAML 的数据绑定链接到 ViewModel 代码的视图。

XAML 基础知识内容

- 概述
- 第 1 部分：XAML 入门
- 第 2 部分：基本 XAML 语法
- 第 3 部分：XAML 标记扩展
- 第 4 部分：数据绑定基础知识
- 第 5 部分：从数据绑定到 MVVM

除了这些 XAML 基础知识文章中，您可以下载一书的章节使用 Xamarin.Forms 创建移动应用：

很多书的章节中更加详细地介绍 XAML 主题包括：

第 7 章。XAML vs。代码	下载 PDF	摘要
第 8 章。代码和 XAML 协调工作	下载 PDF	摘要
第 10 章。XAML 标记扩展	下载 PDF	摘要
第 18 章。MVVM	下载 PDF	摘要

可以是这些章节免费下载的。

概述

XAML 是基于 XML 的语言由 Microsoft 创建的作为编程代码实例化和初始化对象，并组织这些对象在父-子层次结构中的替代方法。XAML 适应多种技术在.NET framework 中，但它在定义的 Windows Presentation Foundation (WPF)、Silverlight、Windows 运行时和通用 Windows 中的用户界面布局中找到其最大的实用程序平台 (UWP)。

XAML 也是 Xamarin.Forms 的跨平台本机基于编程接口为 iOS、Android 和 UWP 的一部分移动设备。在 XAML 文件中，Xamarin.Forms 开发人员可以定义用户界面用作所有 Xamarin.Forms 视图、布局和页面，以及自定义类。可以编译或可执行文件中嵌入的 XAML 文件。无论哪种方式，分析 XAML 信息在生成时查找命名的对象，并再次在运行时来实例化和初始化对象，并建立这些对象和编程代码之间的链接。

XAML 具有几大优势，等效的代码：

- XAML 通常会更简洁和可读比等效的代码。
- 在 XML 中固有的父-子层次结构允许 XAML 来模拟更 visual 清楚用户界面对象的父-子层次结构。
- XAML 可以轻松地手动编写的程序员，但还有助于为可工具化并通过可视化设计工具生成。

当然，也有缺点，主要与是固有的标记语言的限制：

- XAML 不能包含代码。必须在代码文件中定义所有事件处理程序。
- XAML 不能包含重复处理的循环。(但是，多个 Xamarin.Forms 视觉对象 — 最值得注意的是 `ListView` — 可以生成多个子级中的对象基于其 `ItemsSource` 集合。)
- XAML 不能包含有条件处理（但是，数据绑定可以引用，可有效地处理某些条件的代码基于绑定转换器。）
- XAML 通常无法实例化类未定义无参数构造函数。（但是，没有有时会解决此限制问题的方法。）
- XAML 通常不能调用方法。（同样，此限制可以有时克服。）

还没有可视化设计器生成 XAML 在 Xamarin.Forms 应用程序。所有 XAML 都必须手动编写的但没有XAML 预览程序。新增到 XAML 的程序员可能想要频繁生成并运行其应用程序，尤其是之后可能不正确很明显的任何内容。凭借大量经验在 XAML 中甚至开发人员知道试验有价值。

XAML 基本上是 XML，但 XAML 具有一些独特的语法功能。最重要的是：

- 属性元素
- 附加的属性
- 标记扩展

这些功能都*不*XML 扩展。XAML 是完全合法的 XML。但是，这些 XAML 语法功能独特的方式使用 XML。它们详细讨论了中的文章，其中简要介绍如何使用 XAML 来实现 MVVM 得出结论。

要求

本文假定你熟悉 Xamarin.Forms 工作。读取Xamarin.Forms 简介强烈建议。

本文还假定熟悉 XML，包括了解使用 XML 命名空间声明和条款*元素*，*标记*，并*特性*。

当您熟悉使用 Xamarin.Forms 和 XML 时，开始读取第 1 部分。开始使用 XAML。

相关链接

- XamlSamples
- Xamarin.Forms 简介
- 创建移动应用书籍
- Xamarin.Forms 示例

第 1 部分。XAML 入门

2018/11/13 • • Edit Online

XAML 在 Xamarin.Forms 应用程序中，主要用于定义页面的可视内容和一起使用C#代码隐藏文件。

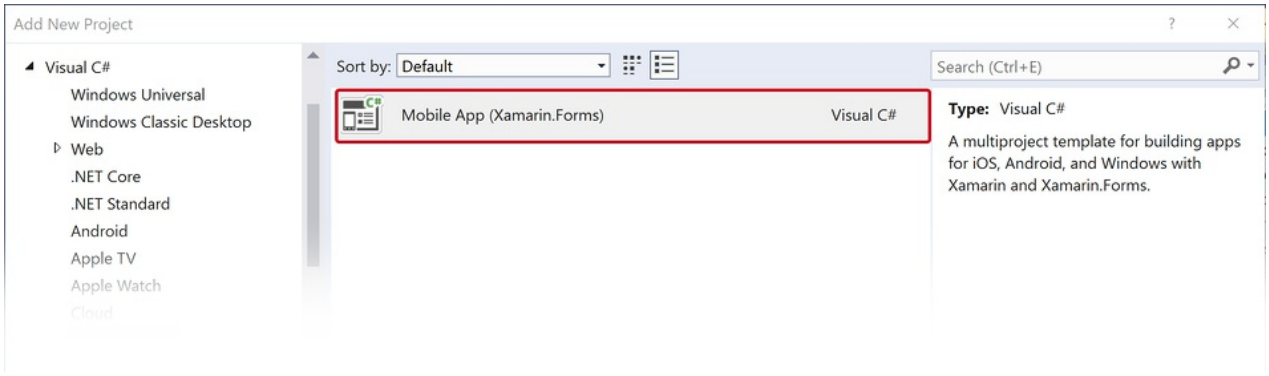
代码隐藏文件提供代码支持的标记。在一起，这两个文件会导致包含子视图和属性初始化的新类定义。在 XAML 文件中，类和属性引用的 XML 元素和属性，并建立标记和代码之间的链接。

创建解决方案

若要开始编辑第一个 XAML 文件，请使用 Visual Studio 或 Visual Studio for Mac，以创建新的 Xamarin.Forms 解决方案。（选择下面与你的环境相对应的选项卡。）

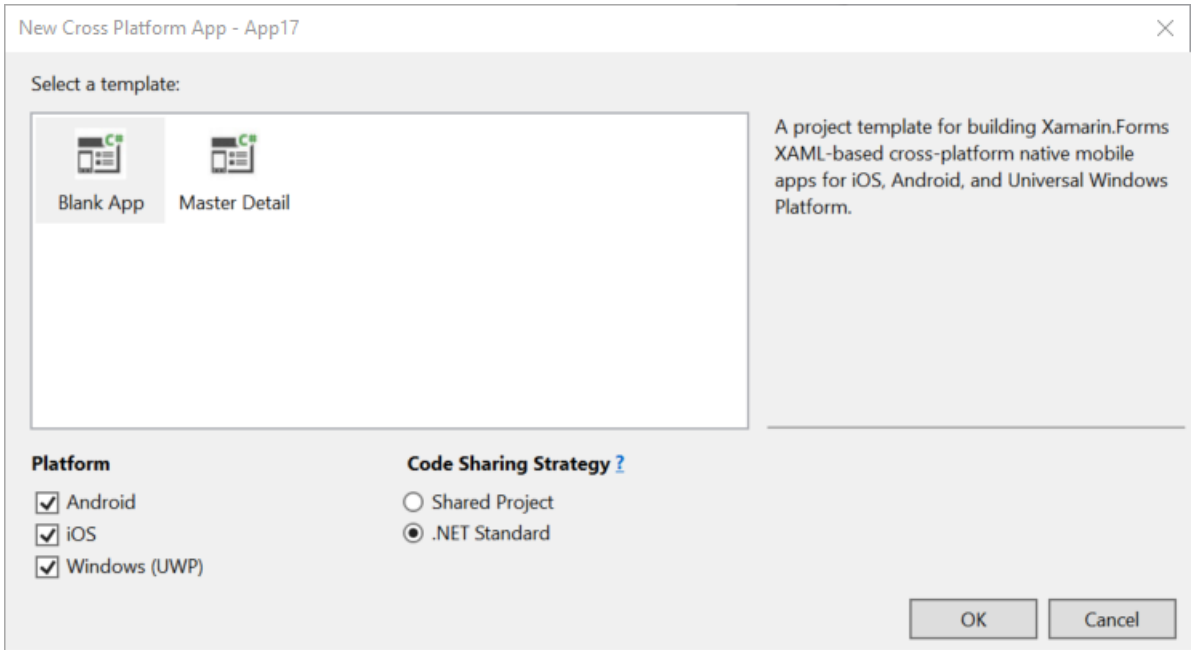
- Visual Studio
- Visual Studio for Mac

在 Windows 中，使用 Visual Studio 选择**文件 > 新建 > 项目**菜单中。在**新的项目**对话框中，选择**Visual C# > 跨平台**在左侧，然后**移动应用 (Xamarin.Forms)** 从中心的列表中。



选择该**解决方案的位置**，为其提供的名称**XamlSamples**（或所需的任意），然后按**确定**。

在下一个屏幕上，选择**空白应用模板并 .NET Standard**代码共享策略：



按**确定**。

在解决方案中创建四个项目：**XamlSamples** .NET Standard 库**XamlSamples.Android**，**XamlSamples.iOS**，和通用 Windows 平台解决方案中，**XamlSamples.UWP**。

在创建后**XamlSamples**解决方案中，你可能想要测试部署环境，通过为解决方案启动项目，选择各种平台项目和生成和部署简单的应用程序创建的phone 仿真程序或真实的设备的项目模板。

除非您需要编写特定于平台的代码共享**XamlSamples** .NET Standard 库项目是在其中你将会花费几乎所有编程时间。这些文章不将该项目外推进。

XAML 文件的剖析

内**XamlSamples** .NET 标准库是一对具有以下名称的文件：

- **App.xaml**，XAML 文件中; 和
- **App.xaml.cs**、C# *代码隐藏*与 XAML 文件相关联的文件。

将需要单击旁边的箭头**App.xaml**若要查看的代码隐藏文件。

这两**App.xaml**并**App.xaml.cs**到一个名为类参与 `App` 派生 `Application` 。XAML 文件与大多数其他类派生的类参与 `ContentPage` ; 这些文件使用 XAML 来定义整个页面的可视内容。这是中的其他两个文件，则返回 true **XamlSamples**项目：

- **MainPage.xaml**，XAML 文件中; 和
- **MainPage.xaml.cs**、C#代码隐藏文件。

MainPage.xaml（尽管可能稍有不同的格式设置），文件如下所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples"
             x:Class="XamlSamples.MainPage">

    <StackLayout>
        <!-- Place new controls here -->
        <Label Text="Welcome to Xamarin Forms!"
               VerticalOptions="Center"
               HorizontalOptions="Center" />
    </StackLayout>

</ContentPage>
```

两个 XML 命名空间 (`xmlns`) 声明将引用的 Uri，看起来在 Xamarin 的网站上的第一个和第二个在 Microsoft 的。不去费神检查哪些这些 Uri 指向。不存在。它们是只需通过 Xamarin 和 Microsoft 所拥有的 Uri 和他们基本上充当版本标识符。

第一个 XML 命名空间声明意味着在没有任何前缀的 XAML 文件中定义的标记如指在 Xamarin.Forms 中，类 `ContentPage` 。第二个命名空间声明定义前缀为 `x` 。这使用多个元素和属性的 XAML 中的内部本身和它们支持 XAML 的其他实现。但是，这些元素和属性是略有不同，具体取决于嵌入在 URI 中的年份。Xamarin.Forms 支持 2009 XAML 规范，但它不是所有。

`local` 命名空间声明，可从.NET Standard 库项目中访问其他类。

该第一个标记末尾 `x` 前缀用于名为的属性 `Class` 。因为这一点的用法 `x` 前缀是几乎通用的 XAML 命名空间，XAML 属性，如 `Class` 几乎总是称为 `x:Class` 。

`x:Class` 特性指定完全限定的.NET 类名称：`MainPage` 类中 `XamlSamples` 命名空间。这意味着此 XAML 文件定义一个名为的新类 `MainPage` 中 `XamlSamples` 派生的命名空间 `ContentPage` -在其中标记 `x:Class` 属性将会显示。

`x:Class` 属性只能出现在要定义一个派生的 XAML 文件的根元素C#类。这是在 XAML 文件中定义的唯一的新类。所有其他 XAML 文件中显示的内容是改为只需从现有的类实例化并初始化。

MainPage.xaml.cs文件如下所示 (除了未使用 `using` 指令):

```
using Xamarin.Forms;

namespace XamlSamples
{
    public partial class MainPage : ContentPage
    {
        public MainPage()
        {
            InitializeComponent();
        }
    }
}
```

`MainPage` 类派生自 `ContentPage`，但请注意 `partial` 类定义。这表明应为另一个分部类定义 `MainPage`，但之类的问题？ 和新增功能的 `InitializeComponent` 方法？

当 Visual Studio 生成项目时，它会分析要生成的 XAML 文件C#代码文件。如果查看**XamlSamples\XamlSamples\obj\Debug**目录中，您会发现名为的文件**XamlSamples.MainPage.xaml.g.cs**。G 表示生成的。这是其他分部类定义的 `MainPage`，其中包含的定义 `InitializeComponent` 方法从调用 `MainPage` 构造函数。这些两个部分 `MainPage` 然后一起编译的类定义。具体取决于是否或不编译 XAML，XAML 文件或二进制形式的 XAML 文件嵌入可执行文件中。

在运行时，代码在特定平台项目调用 `LoadApplication` 方法，并向它传递的新实例 `App` .NET Standard 库中的类。`App` 类构造函数实例化 `MainPage`。该类的构造函数调用 `InitializeComponent`，后者随后调用 `LoadFromXaml` 从.NET Standard 库中提取的 XAML 文件（或其编译的二进制文件）的方法。`LoadFromXaml` 初始化 XAML 文件中定义的所有对象、将它们连接在父-子关系中的组合在一起，将附加到 XAML 文件中设置的事件的代码中定义的事件处理程序并将对象的结果树设置页的内容。

尽管通常不需要花多长时间的生成的代码文件，但有时运行时异常会引发代码在所生成的文件，因此您应了解它们的用法。

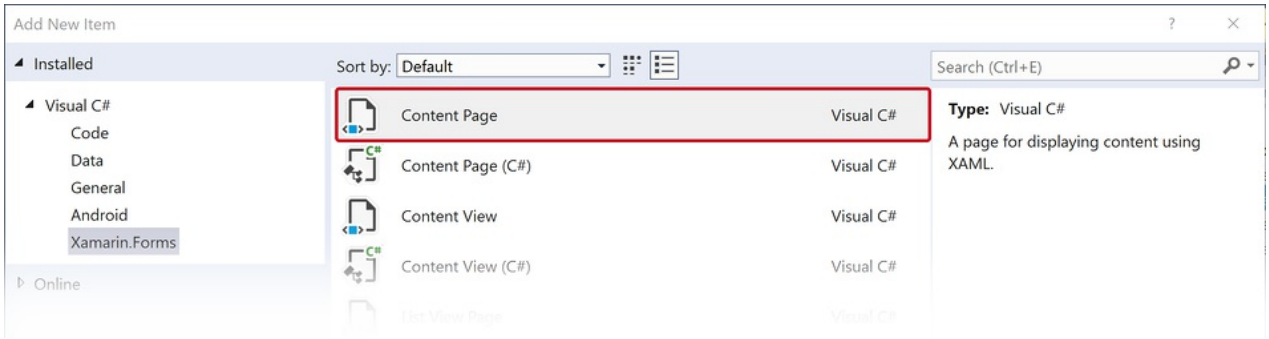
编译和运行此程序时 `Label` 元素出现在页面的中心，如 XAML 所示。从左到右的三个平台有 iOS、Android 和 UWP:

对于更有趣的视觉对象，你只需进行更多有趣的 XAML。

添加新的 XAML 页面

- Visual Studio
- Visual Studio for Mac

若要添加其他基于 XAML 的 `ContentPage` 类到你的项目，选择**XamlSamples** .NET Standard 库项目，然后调用项目 > **添加新项**菜单项。在左侧**添加新项**对话框中，选择**Visual C# 并Xamarin.Forms**。从列表中选择**内容页(不内容页 (C#)**，这将创建一个仅限代码的页面，**或内容视图**，这不是一个页面)。例如，为页面提供一个名称，**HelloXamlPage.xaml**：



两个文件添加到项目中，**HelloXamlPage.xaml**和代码隐藏文件**HelloXamlPage.xaml.cs**。

设置页面内容

编辑**HelloXamlPage.xaml**文件，以便仅标记就是那些用于 `ContentPage` 和 `ContentPage.Content`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage">
    <ContentPage.Content>

    </ContentPage.Content>
</ContentPage>
```

`ContentPage.Content` 标记是唯一的 XAML 语法的一部分。首先，它们可能会显示为无效的 XML，但它们是合法的。段不是 XML 中的特殊字符。

`ContentPage.Content` 标记称为*property 元素*标记。 `Content` 是的一个属性 `ContentPage`，并通常将设置为一个视图或具有子视图的布局。通常情况下属性变为 XAML 中的属性，但它很难设置 `Content` 属性为复杂对象。出于此原因，该属性表示为类名和句点分隔的属性名称组成的 XML 元素。现在 `Content` 属性可以设置之间 `ContentPage.Content` 标记，如下：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.HelloXamlPage"
             Title="Hello XAML Page">
    <ContentPage.Content>

        <Label Text="Hello, XAML!"
               VerticalOptions="Center"
               HorizontalTextAlignment="Center"
               Rotation="-15"
               IsVisible="true"
               FontSize="Large"
               FontAttributes="Bold"
               TextColor="Blue" />

    </ContentPage.Content>
</ContentPage>
```

另请注意，`Title`特性已设置的根标记。

在此期间，类、属性和 XML 之间的关系应该很明显：Xamarin.Forms 类（如`ContentPage`或`Label`）出现在 XML 元素 XAML 文件中。该类的属性，包括`Title`上`ContentPage`和七个属性的`Label`— 通常显示为 XML 属性。

多个快捷方式存在以便为设置这些属性的值。有些属性是基本数据类型：例如，`Title`和`Text`属性属于类型`String`，`Rotation`的类型`Double`，和`IsVisible`（这是`true`默认情况下，仅为此处设置图）的类型是`Boolean`。

`HorizontalTextAlignment`属性属于类型`TextAlignment`，这是一个枚举。对于任何枚举类型的属性，只需提供有成员名称。

对于更复杂的类型的属性中，但是，转换器用于分析 XAML。这些是在 Xamarin.Forms 中派生的类`TypeConverter`。许多都是公共类，但有些不。对于此特定的 XAML 文件，这些类的几个播放在后台角色：

- `LayoutOptionsConverter`有关`VerticalOptions`属性
- `FontSizeConverter`有关`FontSize`属性
- `ColorTypeConverter`有关`TextColor`属性

这些转换器控制允许的属性设置的语法。

`ThicknessTypeConverter`可以处理一个、两个，或由逗号分隔的四个数字。如果提供一个数字，则它将适用于所有四个边。使用两个数字，第一个是左侧和右侧填充，第二项是顶部和底部。中的顺序从左、顶部、右侧和底部有四个数字。

`LayoutOptionsConverter`可以将转换的公共静态字段的名称`LayoutOptions`结构类型的值与`LayoutOptions`。

`FontSizeConverter`可以处理`NamedSize`成员或数字字体大小。

`ColorTypeConverter`接受公共静态字段的名称`Color`结构或十六进制 RGB 值，无论 alpha 通道，前面有数字符号（#）。下面是 alpha 通道没有语法：

```
TextColor="#rrggbb"
```

每个小的字母是十六进制数字。下面是包含 alpha 通道的方式：

```
TextColor="#aarrggbb">
```

Alpha 通道，请注意 FF 完全不透明，00 是完全透明。

其他两种格式，可以指定仅为每个通道的单个十六进制数字：

`TextColor="#rgb"` `TextColor="#argb"`

在这些情况下，该数字将重复以形成的值。例如，#CF3 是 RGB 颜色抄送 FF 33。

页面导航

在运行时**XamlSamples**程序，`MainPage`显示。若要查看新`HelloXamlPage`可以作为新的启动页中设置**App.xaml.cs**文件，或导航到新页上，从`MainPage`。

若要实现导航，请首先更改中的代码**App.xaml.cs**构造函数，以便`NavigationPage`创建对象：

```
public App()
{
    InitializeComponent();
    MainPage = new NavigationPage(new MainPage());
}
```

在中**MainPage.xaml.cs**构造函数中，可以创建一个简单`Button`，并使用事件处理程序以导航到`HelloXamlPage`：

```
public MainPage()
{
    InitializeComponent();

    Button button = new Button
    {
        Text = "Navigate!",
        HorizontalOptions = LayoutOptions.Center,
        VerticalOptions = LayoutOptions.Center
    };

    button.Clicked += async (sender, args) =>
    {
        await Navigation.PushAsync(new HelloXamlPage());
    };

    Content = button;
}
```

设置 `Content` 属性页的替换设置 `Content` XAML 文件中的属性。在编译和部署此程序的新版本时，在屏幕上会出现一个按钮。因此按下导航到 `HelloXamlPage` 。下面是在 iPhone、Android 和 UWP 结果页：

您可以导航回 `MainPage` 使用 **< 返回**在 iOS 上，使用向左的箭头在页的顶部或底部的手机在 Android 上，或在 Windows 10 上的页的顶部使用向左的箭头按钮。

随意尝试不同的方式来呈现 XAML `Label` 。如果你需要在文本中嵌入的任何 Unicode 字符，可以使用标准的 XML 语法。例如，若要将间候语弯引号中，使用：

```
<Label Text="&#x201C;Hello, XAML!&#x201D;" … />
```

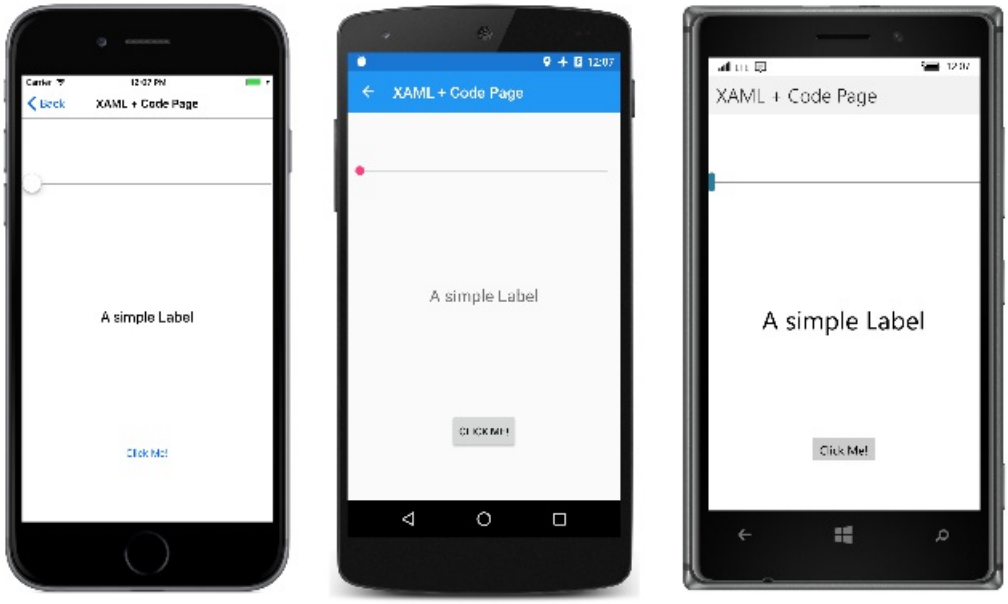
下面是如下所示：

XAML 和代码交互

HelloXamlPage示例仅包含单个 `Label` 的页上，但这是很少。大多数 `ContentPage` 派生类集 `Content` 布局的某些属性进行排序，如 `StackLayout`。 `Children` 的属性 `StackLayout` 定义的类型为 `IList<View>` 但实际类型的对象 `ElementCollection<View>`，并可以与多个视图或其他布局填充集合。在 XAML 中，这些父-子关系被建立与普通 XML 层次结构。下面是一个名为的新页的 XAML 文件**XamlPlusCodePage**:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="CenterAndExpand" />

        <Label Text="A simple Label"
               Font="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Click Me!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

此 XAML 文件是语法上完成，而以下是如下所示：



但是，您很可能要考虑此应用程序是在功能上不足之处。也许 `Slider` 应该会导致 `Label` 若要显示的当前值和 `Button` 可能用于执行一些在程序中操作。

正如您将看到在第 4 部分。数据绑定基础知识，显示的作业 `Slider` 值使用 `Label` 包含数据绑定可以完全在 XAML 中处理。但首先看到的代码解决方案很有用。即便如此，处理 `Button` 单击绝对需要的代码。这意味着的代码隐藏文件 `XamlPlusCodePage` 必须包含的处理程序 `ValueChanged` 的事件 `Slider` 并且 `Clicked` 的事件 `Button` 。让我们添加它们：

```
namespace XamlSamples
{
    public partial class XamlPlusCodePage
    {
        public XamlPlusCodePage()
        {
            InitializeComponent();
        }

        void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
        {

        }

        void OnButtonClicked(object sender, EventArgs args)
        {

        }
    }
}
```

这些事件处理程序不需要是公共的。

返回在 XAML 文件中，`Slider` 并 `Button` 标记需要包括的属性 `ValueChanged` 和 `Clicked` 引用这些处理程序的事件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
    <StackLayout>
        <Slider VerticalOptions="CenterAndExpand"
                ValueChanged="OnSliderValueChanged" />

        <Label Text="A simple Label"
               Font="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Click Me!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Clicked="OnButtonClicked" />
    </StackLayout>
</ContentPage>
```

请注意，将一个处理程序分配到一个事件具有与分配到属性的值相同的语法。

如果处理程序 `ValueChanged` 的事件 `Slider` 将使用 `Label` 若要显示的当前值，处理程序需要在代码中引用该对象。 `Label` 需要一个名称，使用指定 `x:Name` 属性。

```
<Label x:Name="valueLabel"
       Text="A simple Label"
       Font="Large"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand" />
```

`x` 前缀的 `x:Name` 属性指示此属性为 XAML 中的内部。

名称将分配给 `x:Name` 属性具有相同的规则C#变量的名称。例如，它必须以字母或下划线开头并包含任何嵌入的空格。

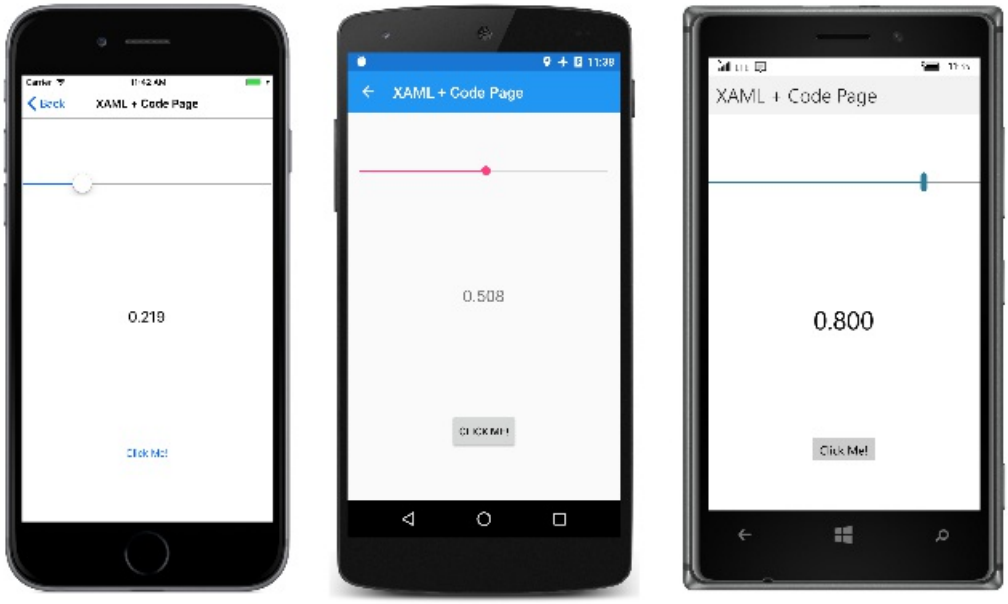
现在 `ValueChanged` 事件处理程序可以设置 `Label` 以显示新 `Slider` 值。新值是从事件参数可用：

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    valueLabel.Text = args.NewValue.ToString("F3");
}
```

或处理程序无法获取 `Slider` 生成此事件的对象 `sender` 自变量，并获取 `Value` 中的属性：

```
void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
{
    valueLabel.Text = ((Slider)sender).Value.ToString("F3");
}
```

首次运行该程序时 `Label` 不会显示 `Slider` 值，因为 `ValueChanged` 尚未尚未触发事件。但任何操作 `Slider` 导致要显示的值：



现在为 `Button` 。我们来模拟的响应 `Clicked` 通过显示的警报事件 `Text` 的按钮。事件处理程序可以安全地强制转换 `sender` 自变量 `Button` ，然后访问其属性：

```
async void OnButtonClicked(object sender, EventArgs args)
{
    Button button = (Button)sender;
    await DisplayAlert("Clicked!",
        "The button labeled '" + button.Text + "' has been clicked",
        "OK");
}
```

该方法定义为 `async` 因为 `DisplayAlert` 方法是异步的应在其前面加 `await` 运算符，该方法完成时返回。因为此方法获取 `Button` 触发此事件从 `sender` 参数，同一处理程序无法用于多个按钮。

您已了解在 XAML 中定义的对象可触发在代码隐藏文件中，处理的事件和代码隐藏文件，可以访问使用分配给与它的名称的 XAML 中定义的对象 `x:Name` 属性。这些是代码和 XAML 进行交互的两种基本方法。

一些更深入了解如何通过检查新生成扔掉 XAML 的工作原理**XamlPlusCode.xaml.g.cs 文件**，它现在包括任何名称分配给任何 `x:Name` 为私有字段的属性。下面是该文件的简化的版本：

```
public partial class XamlPlusCodePage : ContentPage {

    private Label valueLabel;

    private void InitializeComponent() {
        this.LoadFromXaml(typeof(XamlPlusCodePage));
        valueLabel = this.FindByName<Label>("valueLabel");
    }
}
```

此字段的声明允许将自由地在任意位置使用该变量 `XamlPlusCodePage` 下您所在地区的分部类文件。在运行时，该字段之后，分配已分析 XAML。这意味着 `valueLabel` 字段是 `null` 时 `XamlPlusCodePage` 构造函数开始后但有效 `InitializeComponent` 调用。

之后 `InitializeComponent` 返回控件返回到构造函数，就像它们具有已实例化并在代码中初始化已构造的页的视觉对象。XAML 文件不能再在类中扮演任何角色。你能够在任何你想，例如，通过添加到视图的方式上这些对象 `StackLayout` ，或设置 `Content` 为其他页的属性完全。你可以"遍历树"通过检查 `Content` 页面中的项的属性 `Children` 布局的集合。可以在这种方式访问的视图上设置属性或向其动态分配事件处理程序。

可以自由。它是您的页面，并且 XAML 是仅用于生成其内容的工具。

总结

使用这个简介中，您已了解到某个类定义，如何影响 XAML 文件和代码文件和 XAML 和代码的文件如何交互。但是，XAML 还具有其自己独特的语法功能，使其能够以非常灵活的方式使用。你可以开始浏览这些第 2 部分。基本 XAML 语法。

相关链接

- XamlSamples
- 第 2 部分：基本 XAML 语法
- 第 3 部分：XAML 标记扩展
- 第 4 部分：数据绑定基础知识
- 第 5 部分：从数据绑定到 MVVM

第 2 部分。基本 XAML 语法

2018/11/13 • • Edit Online

XAML 主要用于实例化和初始化对象。但通常情况下，属性必须设置为复杂对象不能轻松地表示为 XML 字符串，并有时一个类定义的属性必须设置上一个子类。这些两个需要属性元素和附加的属性的基本 XAML 语法的功能。

属性元素

在 XAML 中，类的属性通常设置为 XML 属性：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large"
       TextColor="Aqua" />
```

但是，没有在 XAML 中设置属性的替代方法。若要尝试使用此替代方法 `TextColor`，首先删除现有 `TextColor` 设置：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large" />
```

打开空元素 `Label` 分隔到开始和结束标记来标记：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">

</Label>
```

这些标记内添加的类名和句点分隔的属性名称组成的开始和结束标记：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">
    <Label.TextColor>

    </Label.TextColor>
</Label>
```

将属性值设置为这些新的标记，像这样的内容：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center"
       FontAttributes="Bold"
       FontSize="Large">
    <Label.TextColor>
        Aqua
    </Label.TextColor>
</Label>
```

若要指定这两种方式 `TextColor` 属性在功能上等效，但不要因为这会有效地进行设置的属性两次，并可能会不明确的同一属性使用的两种方法。

使用此新语法，可以引入了一些便利的术语：

- `Label` 是*对象元素*。它是 Xamarin.Forms 对象表示为一个 XML 元素。
- `Text``VerticalOptions`，`FontAttributes` 并 `FontSize` 是*属性的特性*。它们是 Xamarin.Forms 属性表示为 XML 属性。
- 在此最后一段中，`TextColor` 变得*property 元素*。它是 Xamarin.Forms 属性，但它现在是一个 XML 元素。

定义的属性元素可能在第一次似乎是违反了 XML 语法，但它不是。期间必须在 XML 中没有特殊含义。对 XML 解码器，`Label.TextColor` 是只是正常的子元素。

但是，在 XAML 中，此语法是非常特殊的。属性元素的规则之一是，没有其他可以出现在 `Label.TextColor` 标记。属性的值始终定义为属性元素的开始和结束标记之间的内容。

在多个属性，可以使用属性元素语法：

```
<Label Text="Hello, XAML!"
       VerticalOptions="Center">
    <Label.FontAttributes>
        Bold
    </Label.FontAttributes>
    <Label.FontSize>
        Large
    </Label.FontSize>
    <Label.TextColor>
        Aqua
    </Label.TextColor>
</Label>
```

也可以使用属性元素语法的所有属性：

```
<Label>
    <Label.Text>
        Hello, XAML!
    </Label.Text>
    <Label.FontAttributes>
        Bold
    </Label.FontAttributes>
    <Label.FontSize>
        Large
    </Label.FontSize>
    <Label.TextColor>
        Aqua
    </Label.TextColor>
    <Label.VerticalOptions>
        Center
    </Label.VerticalOptions>
</Label>
```

首先，属性元素语法可能看起来是不必要的过于繁琐替换好像某些内容相对来说非常简单，并在这些示例中这的确是这种情况。

但是，属性元素语法将变得重要太复杂，无法表示为一个简单的字符串属性的值时。属性元素标记内可以实例化另一个对象，并设置其属性。例如，你可以显式设置属性如 `VerticalOptions` 到 `LayoutOptions` 属性设置的值：

```
<Label>
    ...
    <Label.VerticalOptions>
        <LayoutOptions Alignment="Center" />
    </Label.VerticalOptions>
</Label>
```

另一个示例：`Grid` 具有两个属性名为 `RowDefinitions` 和 `ColumnDefinitions` 。这两个属性属于类型 `RowDefinitionCollection` 并 `ColumnDefinitionCollection` ，这是集合的 `RowDefinition` 和 `ColumnDefinition` 对象。需要使用属性元素语法来设置这些集合。

下面是为 XAML 文件的开头 `GridDemoPage` 类，其中显示的属性元素标记 `RowDefinitions` 和 `ColumnDefinitions` 集合：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.GridDemoPage"
             Title="Grid Demo Page">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>
        ...
    </Grid>
</ContentPage>
```

请注意，用于定义自动调整大小的单元格的像素宽度和高度，以及星型设置单元格的缩写的语法。

附加属性

您刚才已了解 `Grid` 要求的属性元素 `RowDefinitions` 和 `ColumnDefinitions` 集合来定义的行和列。但是，还必须对程序员来说，以指示行和列的某种方式其中的每个子 `Grid` 驻留。

中的每个子级的标记 `Grid` 指定行和列的该子项，使用以下属性：

- `Grid.Row`
- `Grid.Column`

这些属性的默认值为 0。您还可以指示是否子跨越多个行或列具有这些属性：

- `Grid.RowSpan`
- `Grid.ColumnSpan`

这两个属性具有默认值为 1。

下面是完整的 GridDemoPage.xaml 文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.GridDemoPage"
             Title="Grid Demo Page">

    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="100" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="100" />
        </Grid.ColumnDefinitions>

        <Label Text="Autosized cell"
               Grid.Row="0" Grid.Column="0"
               TextColor="White"
               BackgroundColor="Blue" />

        <BoxView Color="Silver"
                 HeightRequest="0"
                 Grid.Row="0" Grid.Column="1" />

        <BoxView Color="Teal"
                 Grid.Row="1" Grid.Column="0" />

        <Label Text="Leftover space"
               Grid.Row="1" Grid.Column="1"
               TextColor="Purple"
               BackgroundColor="Aqua"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center" />

        <Label Text="Span two rows (or more if you want)"
               Grid.Row="0" Grid.Column="2" Grid.RowSpan="2"
               TextColor="Yellow"
               BackgroundColor="Blue"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center" />

        <Label Text="Span two columns"
               Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="2"
               TextColor="Blue"
               BackgroundColor="Yellow"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center" />

        <Label Text="Fixed 100x100"
               Grid.Row="2" Grid.Column="2"
               TextColor="Aqua"
               BackgroundColor="Red"
               HorizontalTextAlignment="Center"
               VerticalTextAlignment="Center" />

    </Grid>
</ContentPage>
```

`Grid.Row` 和 `Grid.Column` 设置为 0 不是必需的但通常包括为清晰起见。

下面是如下所示在所有三个平台上：

仅从语法，判断这些`Grid.Row`，`Grid.Column`，`Grid.RowSpan`，和`Grid.ColumnSpan`似乎是静态字段或属性的属性`Grid`，但有趣的是，`Grid`不定义任何名为`Row`，`Column`，`RowSpan`，或`ColumnSpan`。

相反，`Grid`定义名为四个可绑定属性`RowProperty`，`ColumnProperty`，`RowSpanProperty`，和`ColumnSpanProperty`。这些是特殊类型的可绑定属性称为*附加属性*。由定义`Grid`上的子级但设置类`Grid`。

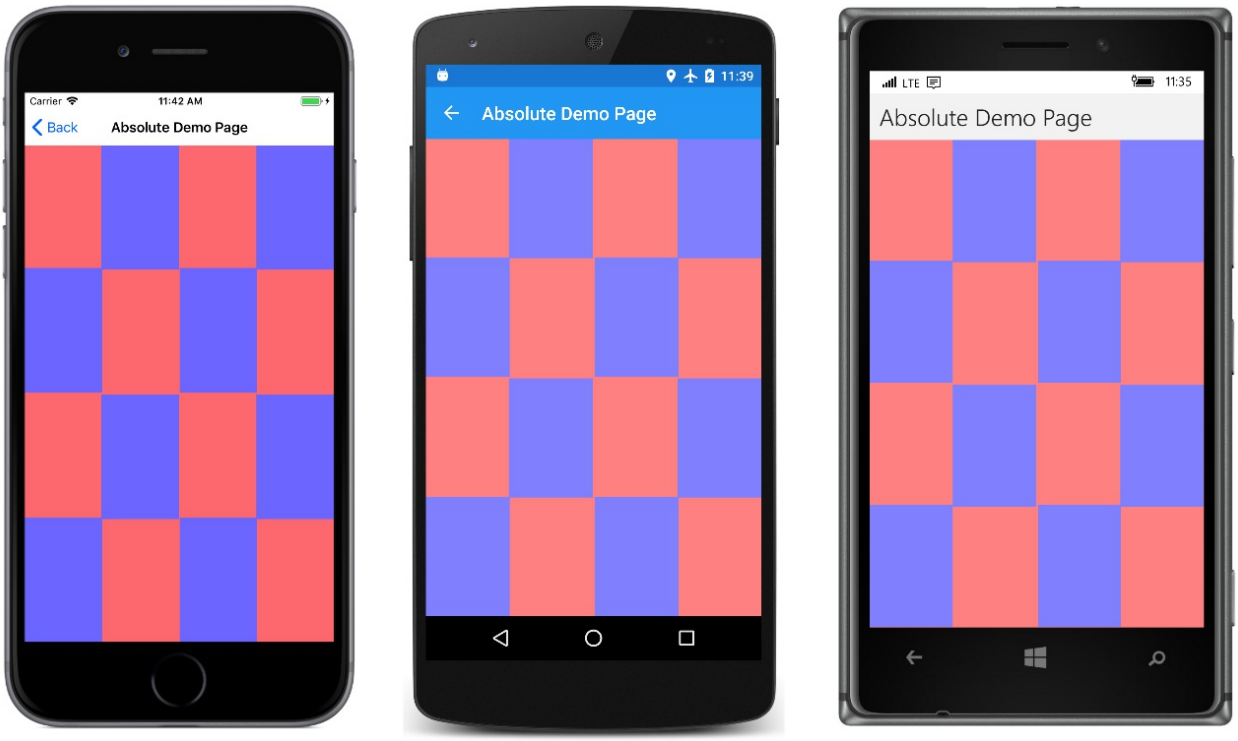
当你希望使用这些附加属性在代码中，`Grid`类提供了名为的静态方法`SetRow`，`GetColumn`，依次类推。但在 XAML 中，这些附加的属性设置为中的子级的属性`Grid`使用简单的属性名称。

附加的属性的形式始终可识别在 XAML 文件中包含的类和一个句点分隔的属性名称的属性。它们被称为*附加属性*因为它们由一个类定义 (在这种情况下，`Grid`)，但附加到其他对象 (在本例中为子级的`Grid`)。在布局期间`Grid`可以询问以了解在哪里放置每个子这些附加属性的值。

`AbsoluteLayout`类定义了名为两个附加的属性`LayoutBounds`和`LayoutFlags`。下面是棋盘图案意识到使用按比例定位和调整大小功能`AbsoluteLayout`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.AbsoluteDemoPage"
             Title="Absolute Demo Page">

    <AbsoluteLayout BackgroundColor="#FF8080">
        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0.33, 0, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="1, 0, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0, 0.33, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0.67, 0.33, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0.33, 0.67, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="1, 0.67, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0, 1, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

        <BoxView Color="#8080FF"
                 AbsoluteLayout.LayoutBounds="0.67, 1, 0.25, 0.25"
                 AbsoluteLayout.LayoutFlags="All" />

  </AbsoluteLayout>
</ContentPage>
```

和其内容如下：



此类的某些内容，可能会质疑使用 XAML 的智慧。当然，重复和的规律性 `LayoutBounds` 矩形所示，它可能会更好地实现在代码中。

这的确是合法的关注点，并且没有与平衡使用代码和标记定义用户界面时没问题。很容易地在 XAML 中定义一些视觉对象，然后使用代码隐藏文件的构造函数添加一些可能会更好地生成在循环中的多个视觉对象。

内容属性

在上一示例中， `StackLayout` ， `Grid` ，并 `AbsoluteLayout` 对象设置为 `Content` 属性 `ContentPage` ，这些布局的子级是实际中的项和 `Children` 集合。但这些 `Content` 和 `Children` 属性却无处 XAML 文件中。

肯定可以包括 `Content` 并 `Children` 属性为属性的元素，如在**XamlPlusCode**示例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.XamlPlusCodePage"
             Title="XAML + Code Page">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout.Children>
                <Slider VerticalOptions="CenterAndExpand"
                        ValueChanged="OnSliderValueChanged" />

                <Label x:Name="valueLabel"
                       Text="A simple Label"
                       FontSize="Large"
                       HorizontalOptions="Center"
                       VerticalOptions="CenterAndExpand" />

                <Button Text="Click Me!"
                      HorizontalOptions="Center"
                      VerticalOptions="CenterAndExpand"
                      Clicked="OnButtonClicked" />
            </StackLayout.Children>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

真正的问题是：为什么是这些属性的元素不所需的 XAML 文件中？

允许在 XAML 中使用 Xamarin.Forms 中定义的元素中被标记了一个属性 `ContentProperty` 类中的属性。如果您查找 `ContentPage` 类在联机的 Xamarin.Forms 文档，您将看到此属性：

```
[Xamarin.Forms.ContentProperty("Content")]
public class ContentPage : TemplatedPage
```

这意味着，`Content` 属性元素标记不是必需的。出现的开始和结束之间的任何 XML 内容 `ContentPage` 标记假定要分配给 `Content` 属性。

`StackLayout``Grid`， `AbsoluteLayout`，并 `RelativeLayout` 都派生 `Layout<View>`，并且如果您查找 `Layout<T>` Xamarin.Forms 文档中，在您将看到另一个 `ContentProperty` 属性：

```
[Xamarin.Forms.ContentProperty("Children")]
public abstract class Layout<T> : Layout ...
```

允许内容的布局以会自动添加到 `Children` 集合，而无需显式 `Children` 属性元素标记。

其他类也有 `ContentProperty` 属性定义。例如，内容的属性 `Label` 是 `Text`。检查其他人的 API 文档。

使用 OnPlatform 平台差异

在单页面应用程序中很常见设置 `Padding` 页后，可以避免覆盖 iOS 状态栏上的属性。在代码中，可以使用 `Device.RuntimePlatform` 属性实现此目的：

```
if (Device.RuntimePlatform == Device.iOS)
{
    Padding = new Thickness(0, 20, 0, 0);
}
```

你还可以执行在 XAML 中使用类似的内容 `OnPlatform` 和 `On` 类。首先添加为属性元素 `Padding` 属性页的顶部附近：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>

    </ContentPage.Padding>
    ...
</ContentPage>
```

在这些标记包括 `OnPlatform` 标记。 `OnPlatform` 是一个泛型类。您需要指定泛型类型参数，在这种情况下，`Thickness`，这是一种 `Padding` 属性。幸运的是，没有专门用于定义名为的泛型参数的 XAML 属性 `x:TypeArguments`。这应与要设置的属性的类型匹配：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">

        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

`OnPlatform` 有一个名为 `Platforms` 也就是说 `IList` 的 `On` 对象。该属性使用属性元素标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <OnPlatform.Platforms>

            </OnPlatform.Platforms>
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

现在，添加 `On` 元素。为每个设置 `Platform` 属性和 `Value` 属性设置为标记 `Thickness` 属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <OnPlatform.Platforms>
                <On Platform="iOS" Value="0, 20, 0, 0" />
                <On Platform="Android" Value="0, 0, 0, 0" />
                <On Platform="UWP" Value="0, 0, 0, 0" />
            </OnPlatform.Platforms>
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

可以简化此标记。内容属性的 `OnPlatform` 是 `Platforms`，因此可以删除这些属性元素标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
            <On Platform="Android" Value="0, 0, 0, 0" />
            <On Platform="UWP" Value="0, 0, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

`Platform` 的属性 `On` 属于类型 `IList<string>`，因此，如果值是相同的则可以包含多个平台：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
            <On Platform="Android, UWP" Value="0, 0, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

由于 Android 和 UWP 设置的默认值为 `Padding`，可以删除标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

这是设置一个与平台相关的标准方法 `Padding` 在 XAML 中的属性。如果 `Value` 设置不能由单个字符串，您可以为其定义属性元素：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="...">

    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS">
                <On.Value>
                    0, 20, 0, 0
                </On.Value>
            </On>
        </OnPlatform>
    </ContentPage.Padding>
  ...
</ContentPage>
```

总结

使用属性元素和附加的属性，很多基本 XAML 语法已建立。但是，有时您需要将属性设置为对象以间接方式，例如，从资源字典。在下一部分中，一部分介绍了此方法3。XAML 标记扩展。

相关链接

- XamlSamples
- 第 1 部分：XAML 入门
- 第 3 部分：XAML 标记扩展

- 第 4 部分：数据绑定基础知识
- 第 5 部分：从数据绑定到 MVVM

第 3 部分。XAML 标记扩展

2018/11/13 • • Edit Online

XAML 标记扩展构成允许将属性设置为对象或从其他源间接引用的值的 XAML 中的重要功能。XAML 标记扩展是特别重要的共享对象，并引用整个应用程序，使用的常量，但它们在数据绑定中查找其最大的实用程序。

XAML 标记扩展

一般情况下，使用 XAML 将对象的属性设置为显式值，如字符串、数字、枚举成员或转换为在后台值的字符串。

有时，但是，属性必须改为引用其他，某一位置定义的值，或可能由代码在运行时需要很少的处理。出于这些目的，XAML *标记扩展*可用。

这些 XAML 标记扩展不是扩展的 XML。XAML 是完全合法的 XML。它们称为"扩展"因为它们由代码中实现的类支持 `IMarkupExtension` 。您可以编写自己的自定义标记扩展。

在许多情况下，XAML 标记扩展是 XAML 文件中立即认出因为它们将显示由大括号分隔的属性设置为: {和}，但有时标记扩展将出现在标记为传统的元素。

共享的资源

某些 XAML 页属性设置为相同的值包含多个视图。例如，许多这些属性设置 `Button` 对象是否相同:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

    <StackLayout>
        <Button Text="Do this!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                BorderWidth="3"
                Rotation="-15"
                TextColor="Red"
                FontSize="24" />

        <Button Text="Do that!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                BorderWidth="3"
                Rotation="-15"
                TextColor="Red"
                FontSize="24" />

        <Button Text="Do the other thing!"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                BorderWidth="3"
                Rotation="-15"
                TextColor="Red"
                FontSize="24" />

    </StackLayout>
</ContentPage>
```

如果其中一个属性需要更改，可能想要进行的更改只需一次而不是三次。如果这是代码，您将可能使用常量和静

态只读对象来帮助保持一致且轻松地修改此类值。

在 XAML，一种受欢迎的解决方案是将存储此类值或对象中*资源字典*。`VisualElement` 类定义一个名为属性 `Resources` 类型的 `ResourceDictionary`，这是类型的键的字典 `string` 的类型和值 `object`。可以将对象放入此字典，然后从标记中，所有在 XAML 中引用它们。

若要在页面上使用的资源字典，包含一对 `Resources` 属性元素标记。它是最方便的方式将这些页面的顶部：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

    <ContentPage.Resources>

    </ContentPage.Resources>
    ...
</ContentPage>
```

还有必要显式包括 `ResourceDictionary` 标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>

        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>
```

现在可以对资源字典添加对象和各种类型的值。这些类型必须是可实例化。它们不能为抽象类，例如。这些类型还必须具有公共无参数构造函数。每个项需要一个使用指定的字典键 `x:Key` 属性。例如：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions"
                           Alignment="Center" />

            <LayoutOptions x:Key="vertOptions"
                           Alignment="Center"
                           Expands="True" />
        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>
```

这两项是结构类型的值 `LayoutOptions`，并且每个包含唯一键，另一个或两个属性设置。在代码和标记，它是更常见，若要使用的静态字段 `LayoutOptions`，但下面是更方便地设置属性。

现在需要设置 `HorizontalOptions` 并 `VerticalOptions` 对这些资源，这些按钮的属性和为此，执行 `StaticResource` XAML 标记扩展：

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="3"
        Rotation="-15"
        TextColor="Red"
        FontSize="24" />
```

`StaticResource`标记扩展始终用大括号分隔，并且包括字典的键。

名称`StaticResource`使它有别于`DynamicResource`，它还支持 Xamarin.Forms。`DynamicResource`适用于在运行时，可能会更改的值与关联的字典键时`StaticResource`只是当构造页上的元素后从字典访问元素。

有关`BorderWidth`属性，它是必需的字典中存储一个双精度值。XAML 可以方便地定义对等的常见数据类型的标记`x:Double`和`x:Int32`：

```
<ContentPage.Resources>
    <ResourceDictionary>
        <LayoutOptions x:Key="horzOptions"
                       Alignment="Center" />

        <LayoutOptions x:Key="vertOptions"
                       Alignment="Center"
                       Expands="True" />

        <x:Double x:Key="borderWidth">
            3
        </x:Double>
    </ResourceDictionary>
</ContentPage.Resources>
```

不需要将其放在三个行。此旋转角度此字典条目仅占用一行：

```
<ContentPage.Resources>
    <ResourceDictionary>
        <LayoutOptions x:Key="horzOptions"
                       Alignment="Center" />

        <LayoutOptions x:Key="vertOptions"
                       Alignment="Center"
                       Expands="True" />

         <x:Double x:Key="borderWidth">
            3
         </x:Double>

        <x:Double x:Key="rotationAngle">-15</x:Double>
    </ResourceDictionary>
</ContentPage.Resources>
```

这两个资源可以引用方式与`LayoutOptions`值：

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="{StaticResource borderWidth}"
        Rotation="{StaticResource rotationAngle}"
        TextColor="Red"
        FontSize="24" />
```

类型的资源 `Color`，可以使用直接分配这些类型的属性时使用的相同字符串表示形式。创建资源时，将调用类型转换器。下面是类型的资源 `Color`：

```
<Color x:Key="textColor">Red</Color>
```

通常情况下，程序集 `FontSize` 属性绑定到的成员 `NamedSize` 如枚举 `Large`。 `FontSizeConverter` 类在后台将其转换为依赖于平台的值使用的工作原理 `Device.GetNamedSized` 方法。但是，在定义的字体大小资源时，它更有意义使用数字值，显示为此处 `x:Double` 类型：

```
<x:Double x:Key="fontSize">24</x:Double>
```

现在所有的属性除外 `Text` 由资源设置定义：

```
<Button Text="Do this!"
        HorizontalOptions="{StaticResource horzOptions}"
        VerticalOptions="{StaticResource vertOptions}"
        BorderWidth="{StaticResource borderWidth}"
        Rotation="{StaticResource rotationAngle}"
        TextColor="{StaticResource textColor}"
        FontSize="{StaticResource fontSize}" />
```

还有可能要使用 `OnPlatform` 来定义适用于平台的不同值的资源字典中。下面是如何 `OnPlatform` 对象可以是不同的文本颜色的资源字典的一部分：

```
<OnPlatform x:Key="textColor"
            x:TypeArguments="Color">
    <On Platform="iOS" Value="Red" />
    <On Platform="Android" Value="Aqua" />
    <On Platform="UWP" Value="#80FF80" />
</OnPlatform>
```

请注意，`OnPlatform` 获取同时 `x:Key` 属性，因为它是在字典中的对象和一个 `x:TypeArguments` 属性，因为它是一个泛型类。 `iOS`， `Android`，并 `UWP` 特性转换为 `Color` 值在初始化对象时。

下面是最终完成 XAML 文件具有访问共享的六个值的三个按钮：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SharedResourcesPage"
             Title="Shared Resources Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <LayoutOptions x:Key="horzOptions"
                           Alignment="Center" />

            <LayoutOptions x:Key="vertOptions"
                           Alignment="Center"
                           Expands="True" />

            <x:Double x:Key="borderWidth">3</x:Double>

            <x:Double x:Key="rotationAngle">-15</x:Double>

            <OnPlatform x:Key="textColor"
                        x:TypeArguments="Color">
                <On Platform="iOS" Value="Red" />
                <On Platform="Android" Value="Aqua" />
                <On Platform="UWP" Value="#80FF80" />
            </OnPlatform>

            <x:String x:Key="fontSize">Large</x:String>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <Button Text="Do this!"
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                Rotation="{StaticResource rotationAngle}"
                TextColor="{StaticResource textColor}"
                FontSize="{StaticResource fontSize}" />

        <Button Text="Do that!"
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                Rotation="{StaticResource rotationAngle}"
                TextColor="{StaticResource textColor}"
                FontSize="{StaticResource fontSize}" />

        <Button Text="Do the other thing!"
                HorizontalOptions="{StaticResource horzOptions}"
                VerticalOptions="{StaticResource vertOptions}"
                BorderWidth="{StaticResource borderWidth}"
                Rotation="{StaticResource rotationAngle}"
                TextColor="{StaticResource textColor}"
                FontSize="{StaticResource fontSize}" />

    </StackLayout>
</ContentPage>
```

屏幕截图验证**一致**样**式和依**赖**于平台的**样**式**:

尽管最常见的定义 `Resources` 集合顶部的页上，请记住，`Resources` 属性定义由 `VisualElement`，并且可以具有 `Resources` 上页面上其他元素的集合。例如，请添加一个到 `StackLayout` 在此示例中：

```
<StackLayout>
    <StackLayout.Resources>
        <ResourceDictionary>
            <Color x:Key="textColor">Blue</Color>
        </ResourceDictionary>
    </StackLayout.Resources>
    ...
</StackLayout>
```

你会发现这些按钮的文本颜色现在为蓝色。基本上，只要 XAML 分析程序遇到 `StaticResource` 标记扩展，它在可视树搜索，并使用第一个 `ResourceDictionary` 遇到包含该密钥。

最常见的资源字典中存储的对象类型之一是 Xamarin.Forms `Style`，其定义的属性设置的集合。样式将在本文中讨论样式。

有时开发人员熟悉 XAML 想知道是否他们可如将可视元素 `Label` 或 `Button` 中 `ResourceDictionary`。虽然肯定可能，但它没有太大意义。用途 `ResourceDictionary` 是共享的对象。不能共享一个可视元素。同一个实例不能出现两次在单个页面上。

X:static 标记扩展

尽管其名称的相似之处 `x:Static` 和 `StaticResource` 有很大差异。`StaticResource` 从资源字典时返回的对象 `x:Static` 访问以下项之一：

- 公共静态字段
- 公共静态属性
- 公共常量字段
- 枚举成员。

`StaticResource` 标记扩展支持的 XAML 实现定义的资源字典，虽然 `x:Static` 属于内部函数 XAML，作为 `x` 前缀显示。

下面是几个示例，演示如何 `x:Static` 可以显式引用静态字段和枚举成员：

```
<Label Text="Hello, XAML!"
       VerticalOptions="{x:Static LayoutOptions.Start}"
       HorizontalTextAlignment="{x:Static TextAlignment.Center}"
       TextColor="{x:Static Color.Aqua}" />
```

到目前为止，这并不令人印象非常深刻。但 `x:Static` 标记扩展还可以引用静态字段或属性从你自己的代码。例如，下面是 `AppConstants` 类，其中包含一些您可能想要在整个应用程序的多个页面上使用的静态字段：

```
using System;
using Xamarin.Forms;

namespace XamlSamples
{
    static class AppConstants
    {
        public static readonly Thickness PagePadding;

        public static readonly Font TitleFont;

        public static readonly Color BackgroundColor = Color.Aqua;

        public static readonly Color ForegroundColor = Color.Brown;

        static AppConstants()
        {
            switch (Device.RuntimePlatform)
            {
                case Device.iOS:
                    PagePadding = new Thickness(5, 20, 5, 0);
                    TitleFont = Font.SystemFontOfSize(35, FontAttributes.Bold);
                    break;

                case Device.Android:
                    PagePadding = new Thickness(5, 0, 5, 0);
                    TitleFont = Font.SystemFontOfSize(40, FontAttributes.Bold);
                    break;

                case Device.UWP:
                    PagePadding = new Thickness(5, 0, 5, 0);
                    TitleFont = Font.SystemFontOfSize(50, FontAttributes.Bold);
                    break;
            }
        }
    }
}
```

若要引用的 XAML 文件中此类的静态字段，将需要设法指出此文件的所在位置在 XAML 文件中。使用 XML 命名空间声明来执行此操作。

回想一下作为标准 Xamarin.Forms XAML 模板的一部分创建的 XAML 文件包含两个 XML 命名空间声明：一个用于访问 Xamarin.Forms 类和另一个用于引用标记和属性到 XAML 内部函数：

```
xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

你将需要额外的 XML 命名空间声明，若要访问其他类。每个其他的 XML 命名空间声明定义一个新的前缀。若要访问的共享的应用程序的.NET 标准库的本地类，例如 `AppConstants`，XAML 编程人员通常使用该前缀 `local`。命名空间声明必须指示 CLR（公共语言运行时）命名空间名称，也称为.NET 命名空间名称，名称将显示在C# `namespace` 定义中或在 `using` 指令：

```
xmlns:local="clr-namespace:XamlSamples"
```

此外可以在.NET Standard 库引用任何程序集中定义.NET 命名空间的 XML 命名空间的声明。例如，下面是 `sys` 的标准.NET 前缀 `System` 命名空间，即**mscorlib程序集**，这对于"Microsoft 通用对象运行时库，"花了一次，但现在表示"多语言标准常见对象运行时库。" 由于这是另一个程序集，您还必须指定程序集名称，在这种情况下**mscorlib**:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

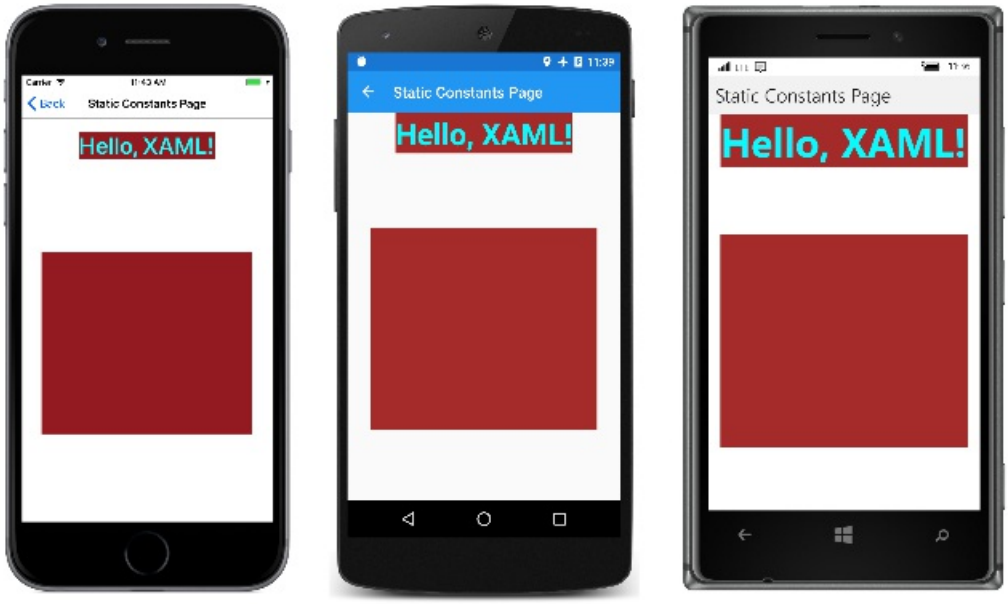
请注意，在关键字 `clr-namespace` 后跟一个冒号，然后.NET 命名空间名称后，接分号，关键字 `assembly`，等号和程序集名称。

是的加上冒号 `clr-namespace` 但等号后面 `assembly`。语法定义在此方式故意：最 XML 命名空间声明引用一个 URI，如开始一个 URI 方案名称 `http`，这始终后跟一个冒号。 `clr-namespace` 此字符串的一部分用于模拟这种约定。

中包含以下两个命名空间声明**StaticConstantsPage**示例。请注意， `BoxView` 尺寸设置为 `Math.PI` 和 `Math.E`，但缩放到原来的 100:

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             x:Class="XamlSamples.StaticConstantsPage"
             Title="Static Constants Page"
             Padding="{x:Static local:AppConstants.PagePadding}">

    <StackLayout>
       <Label Text="Hello, XAML!"
              TextColor="{x:Static local:AppConstants.BackgroundColor}"
              BackgroundColor="{x:Static local:AppConstants.ForegroundColor}"
              Font="{x:Static local:AppConstants.TitleFont}"
              HorizontalOptions="Center" />

      <BoxView WidthRequest="{x:Static sys:Math.PI}"
               HeightRequest="{x:Static sys:Math.E}"
               Color="{x:Static local:AppConstants.ForegroundColor}"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               Scale="100" />
    </StackLayout>
</ContentPage>
```

大小所产生的 `BoxView` 相对于屏幕是依赖于平台的:



其他标准标记扩展

多个标记扩展是固有的 XAML 和在 Xamarin.Forms XAML 文件中受支持。其中一些不常使用，但在需要时至关重要：

- 如果属性具有非 `null` 值的默认值，但你想要将其设置为 `null`，将其设置为 `{x:Null}` 标记扩展。
- 如果某个属性属于类型 `Type`，你可以将其分配给 `Type` 对象使用标记扩展 `{x:Type someClass}`。
- 可在 XAML 中使用定义数组 `x:Array` 标记扩展。此标记扩展具有所需的属性名为 `Type`，该值指示数组中元素的类型。
- `Binding` 标记扩展中讨论了第 4 部分。数据绑定基础知识。

ConstraintExpression 标记扩展

标记扩展可以有属性，但它们不将设置等的 XML 属性。在标记扩展中，属性设置以逗号分隔，并在大括号内显示没有引号。

这可以用名为 Xamarin.Forms 标记扩展说明 `ConstraintExpression`，用于与 `RelativeLayout` 类。作为常量，或相对于父级或其他命名的视图，可以指定的位置或子视图的大小。语法 `ConstraintExpression` 允许您设置的位置或视图使用的大小 `Factor` 属性的另一个视图，加上一次 `Constant`。比这更复杂的任何内容需要代码。

以下是一个示例：

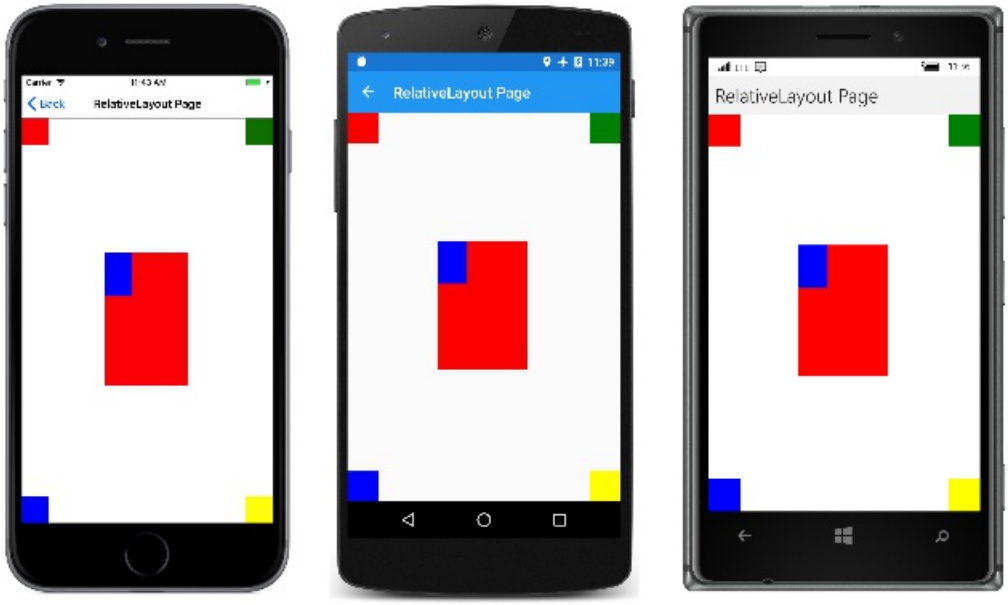
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.RelativeLayoutPage"
             Title="RelativeLayout Page">

    <RelativeLayout>

        <!-- Upper left -->
        <BoxView Color="Red"
                 RelativeLayout.XConstraint=
                     "{ConstraintExpression Type=Constant,
                                            Constant=0}"
                 RelativeLayout.YConstraint=
                     "{ConstraintExpression Type=Constant,
                                            Constant=0}" />
        <!-- Upper right -->
        <BoxView Color="Green"
                 RelativeLayout.XConstraint=
                     "{ConstraintExpression Type=RelativeToParent,
```

```
                                    Property=Width,
                                    Factor=1,
                                    Constant=-40}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=Constant,
                                    Constant=0}" />
        <!-- Lower left -->
        <BoxView Color="Blue"
            RelativeLayout.XConstraint=
                "{ConstraintExpression Type=Constant,
                                    Constant=0}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=1,
                                    Constant=-40}" />
        <!-- Lower right -->
        <BoxView Color="Yellow"
            RelativeLayout.XConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=1,
                                    Constant=-40}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=1,
                                    Constant=-40}" />

        <!-- Centered and 1/3 width and height of parent -->
        <BoxView x:Name="oneThird"
            Color="Red"
            RelativeLayout.XConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=0.33}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=0.33}"
            RelativeLayout.WidthConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Width,
                                    Factor=0.33}"
            RelativeLayout.HeightConstraint=
                "{ConstraintExpression Type=RelativeToParent,
                                    Property=Height,
                                    Factor=0.33}"  />

        <!-- 1/3 width and height of previous -->
        <BoxView Color="Blue"
            RelativeLayout.XConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=X}"
            RelativeLayout.YConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Y}"
            RelativeLayout.WidthConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Width,
                                    Factor=0.33}"
            RelativeLayout.HeightConstraint=
                "{ConstraintExpression Type=RelativeToView,
                                    ElementName=oneThird,
                                    Property=Height,
                                    Factor=0.33}"  />
```

```
    </RelativeLayout>
</ContentPage>
```

可能需要与此示例的最重要经验是标记扩展的语法：没有引号必须出现在标记扩展的大括号内。键入时标记扩展在 XAML 文件中，是自然地想要将属性的值括在引号。忍不住 ！

下面是运行的程序：



总结

如下所示的 XAML 标记扩展提供 XAML 文件的重要支持。但也许是最有价值的 XAML 标记扩展`Binding`，其中并未包括在本系列的下一部分第 4 部分。数据绑定基础知识。

相关链接

- XamlSamples
- 第 1 部分：XAML 入门
- 第 2 部分：基本 XAML 语法
- 第 4 部分：数据绑定基础知识
- 第 5 部分：从数据绑定到 MVVM

第 4 部分。数据绑定基础知识

2018/11/13 • • Edit Online

数据绑定使两个用于链接，这样一个导致更改中的其他对象的属性。这是一个非常有价值的工具，并可以完全在代码中定义数据绑定，而 XAML 提供快捷方式和便利。因此，在 Xamarin.Forms 中最重要的标记扩展之一绑定。

数据绑定

数据绑定连接两个对象，调用的属性*源*并*目标*。在代码中，两个步骤是必需的：`BindingContext` 的目标对象的属性必须设置为源对象，并 `SetBinding` 方法（通常与结合使用 `Binding` 类）必须要绑定的属性的目标对象上调用源对象的属性对象。

目标属性必须是可绑定属性，这意味着目标对象必须派生自 `BindableObject`。联机 Xamarin.Forms 文档将指示哪个属性是可绑定属性。属性 `Label` 如 `Text` 与可绑定属性关联 `TextProperty`。

在标记中，您还必须执行相同的两个步骤所需在代码中，只不过 `Binding` 标记扩展可以代替 `SetBinding` 调用和 `Binding` 类。

但是，在 XAML 中定义数据绑定时，有多种方法来设置 `BindingContext` 的目标对象。有时从代码隐藏文件中，有时使用设置 `StaticResource` 或 `x:Static` 标记扩展的内容有时 `BindingContext` 属性元素标记。

绑定最常使用与基础数据模型，通常在 MVVM（模型-视图-视图）的应用程序体系结构、实现连接程序的视觉对象中所述第 5 部分。从数据绑定到 MVVM，但可能会出现其他情形。

视图-视图绑定

您可以定义要链接的同一页面上的两个视图的属性的数据绑定。在这种情况下，设置 `BindingContext` 的目标对象使用 `x:Reference` 标记扩展。

下面是包含的 XAML 文件 `Slider` 和两个 `Label` 视图，其中之一旋转 `Slider` 值，其中显示了另一个 `Slider` 值：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SliderBindingsPage"
             Title="Slider Bindings Page">

    <StackLayout>
        <Label Text="ROTATION"
               BindingContext="{x:Reference Name=slider}"
               Rotation="{Binding Path=Value}"
               FontAttributes="Bold"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                Maximum="360"
                VerticalOptions="CenterAndExpand" />

        <Label BindingContext="{x:Reference slider}"
               Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
               FontAttributes="Bold"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

`Slider` 包含 `x:Name` 这两个引用的属性 `Label` 视图使用 `x:Reference` 标记扩展。

`x:Reference` 绑定扩展定义名为的属性 `Name` 若要在这种情况下设置为引用的元素的名称 `slider` 。但是，`ReferenceExtension` 定义的类 `x:Reference` 标记扩展还定义 `ContentProperty` 属性 `Name` ，这意味着它不是明确要求。仅针对各种，第一个 `x:Reference` 包括"名称 =", 但第二个不：

```
BindingContext="{x:Reference Name=slider}"
…
BindingContext="{x:Reference slider}"
```

`Binding` 标记扩展本身可以有多个属性，就像 `BindingBase` 和 `Binding` 类。 `ContentProperty` 有关 `Binding` 是 `Path` ，但"路径 ="可以省略标记扩展的一部分，如果路径中的第一项 `Binding` 标记扩展。第一个示例具有"路径 =", 但第二个示例省略它：

```
Rotation="{Binding Path=Value}"
…
Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
```

属性可以是在同一行或分成多行：

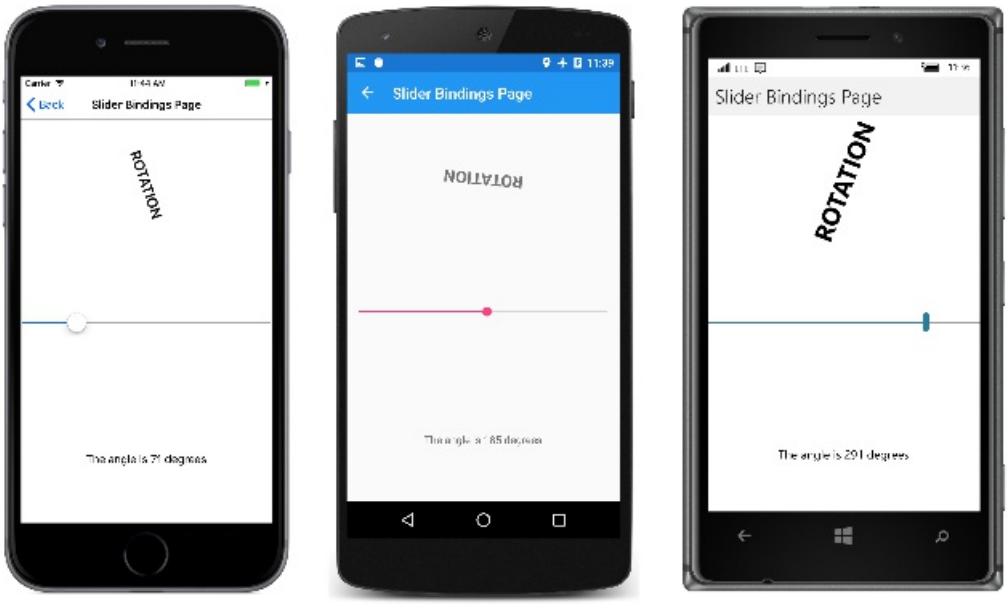
```
Text="{Binding Value,
               StringFormat='The angle is {0:F0} degrees'}"
```

执行任何方便。

请注意 `StringFormat` 在第二个属性 `Binding` 标记扩展。在 Xamarin.Forms 中，绑定不执行任何隐式类型转换，并且如果您需要将非字符串对象显示为一个字符串，你必须提供类型转换器或使用 `StringFormat` 。在后台，静态 `String.Format` 方法用于实现 `StringFormat` 。这可能是问题，因为.NET 格式规范涉及大括号，还用来分隔标记扩展。这将创建令人混乱 XAML 分析器的风险。若要避免这种情况，请用单引号引起来将整个的格式设置字符串：

```
Text="{Binding Value, StringFormat='The angle is {0:F0} degrees'}"
```

下面是正在运行的程序：



绑定模式

单个视图对其几个属性具有数据绑定。但是，每个视图只能有一个 `BindingContext`，因此，在该视图上的多个数据绑定必须全部引用同一对象的属性。

向此服务器和其他问题的解决方案涉及 `Mode` 属性设置为的成员 `BindingMode` 枚举：

- `Default`
- `OneWay` -值从源传输到目标
- `OneWayToSource` -值从目标传输到源
- `TwoWay` -值传输源和目标之间的这两种方式

下面的程序说明的一个常见用途 `OneWayToSource` 和 `TwoWay` 绑定模式。四个 `Slider` 视图都为了控制 `Scale`，`Rotate`， `RotateX`，并 `RotateY` 属性的 `Label`。首先，它看起来像的这四个属性 `Label` 应为数据绑定目标，因为每个所设置 `Slider`。但是，`BindingContext` 的 `Label` 可以是只有一个对象，并且有四个不同的滑块。

出于此原因，所有绑定中都设置似乎是向后的方法：`BindingContext` 的每四个滑块都设置为 `Label`，和上都设置绑定 `Value` 滑块的属性。通过使用 `OneWayToSource` 和 `TwoWay` 模式下，这些 `Value` 属性可以设置的源属性，即 `Scale`， `Rotate`， `RotateX`，并 `RotateY` 的属性 `Label`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="XamlSamples.SliderTransformsPage"
             Padding="5"
             Title="Slider Transforms Page">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
```

```
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <!-- Scaled and rotated Label -->
        <Label x:Name="label"
               Text="TEXT"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <!-- Slider and identifying Label for Scale -->
        <Slider x:Name="scaleSlider"
                BindingContext="{x:Reference label}"
                Grid.Row="1" Grid.Column="0"
                Maximum="10"
                Value="{Binding Scale, Mode=TwoWay}" />

        <Label BindingContext="{x:Reference scaleSlider}"
               Text="{Binding Value, StringFormat='Scale = {0:F1}'}"
               Grid.Row="1" Grid.Column="1"
               VerticalTextAlignment="Center" />

        <!-- Slider and identifying Label for Rotation -->
        <Slider x:Name="rotationSlider"
                BindingContext="{x:Reference label}"
                Grid.Row="2" Grid.Column="0"
                Maximum="360"
                Value="{Binding Rotation, Mode=OneWayToSource}" />

        <Label BindingContext="{x:Reference rotationSlider}"
               Text="{Binding Value, StringFormat='Rotation = {0:F0}'}"
               Grid.Row="2" Grid.Column="1"
               VerticalTextAlignment="Center" />

        <!-- Slider and identifying Label for RotationX -->
        <Slider x:Name="rotationXSlider"
                BindingContext="{x:Reference label}"
                Grid.Row="3" Grid.Column="0"
                Maximum="360"
                Value="{Binding RotationX, Mode=OneWayToSource}" />

        <Label BindingContext="{x:Reference rotationXSlider}"
               Text="{Binding Value, StringFormat='RotationX = {0:F0}'}"
               Grid.Row="3" Grid.Column="1"
               VerticalTextAlignment="Center" />

        <!-- Slider and identifying Label for RotationY -->
        <Slider x:Name="rotationYSlider"
                BindingContext="{x:Reference label}"
                Grid.Row="4" Grid.Column="0"
                Maximum="360"
                Value="{Binding RotationY, Mode=OneWayToSource}" />

        <Label BindingContext="{x:Reference rotationYSlider}"
               Text="{Binding Value, StringFormat='RotationY = {0:F0}'}"
               Grid.Row="4" Grid.Column="1"
               VerticalTextAlignment="Center" />
    </Grid>
</ContentPage>
```

对其中三个绑定 `Slider` 视图是 `OneWayToSource`，这意味着，`Slider` 值的属性中将导致更改其 `BindingContext`，即 `Label` 名为 `label`。这三个 `Slider` 视图会改变 `Rotate`、`RotateX`，和 `RotateY` 的属性 `Label`。

但是，对于绑定 `Scale` 属性是 `TwoWay`。这是因为 `Scale` 属性具有默认值为 1，并使用 `TwoWay` 绑定的原因 `Slider` 初始值设置为 1 而不是 0。如果该绑定 `OneWayToSource`，则 `Scale` 属性将最初设置为从 0 `Slider` 默认值。`Label` 会显示，并向用户可能导致一些混淆。

> **NOTE**
> `VisualElement` 类还具有 `ScaleX` 并 `ScaleY` 属性，缩放 `VisualElement` x 轴和 y 轴上分别。

绑定和集合

执行任何操作演示如何将优于模板化的 XAML 和数据绑定功能 `ListView`。

`ListView` 定义 `ItemsSource` 类型的属性 `IEnumerable`，和它在该集合中显示的项。这些项可以是任何类型的对象。默认情况下 `ListView` 使用 `ToString` 方法的每个项来显示该项。有时这是正是您所希望，但在许多情况下，`ToString` 返回对象的完全限定的类名称。

但是中的项 `ListView` 集合可以显示希望使用的方式*模板*，其中包括派生的类 `Cell`。在模板中每个项目都会被克隆 `ListView`，并在模板已设置的数据绑定被传输到单个的克隆。

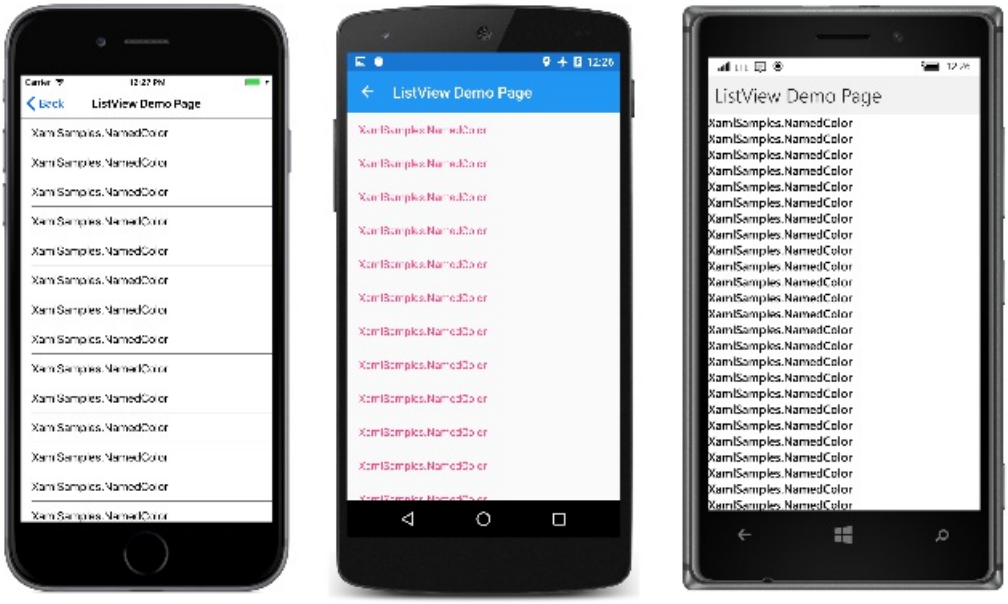
通常，你将想要创建使用这些项的自定义单元格 `ViewCell` 类。此过程是太好理解在代码中，但在 XAML 中变得非常简单。

XamlSamples 中包含的项目是一个名为类 `NamedColor`。每个 `NamedColor` 对象具有 `Name` 并 `FriendlyName` 类型的属性 `string`，和一个 `Color` 类型的属性 `Color`。此外，`NamedColor` 具有 141 静态只读字段的类型 `Color` 对应于 Xamarin.Forms 中定义的颜色 `Color` 类。静态构造函数创建 `IEnumerable<NamedColor>` 集合，其中包含 `NamedColor` 对象对应于这些静态字段，并将其分配给它的公共静态 `All` 属性。

设置静态 `NamedColor.All` 属性设置为 `ItemsSource` 的 `ListView` 轻松使用 `x:Static` 标记扩展：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
             x:Class="XamlSamples.ListViewDemoPage"
             Title="ListView Demo Page">

    <ListView ItemsSource="{x:Static local:NamedColor.All}" />

</ContentPage>
```

结果显示建立的类型的真正项 `XamlSamples.NamedColor`：

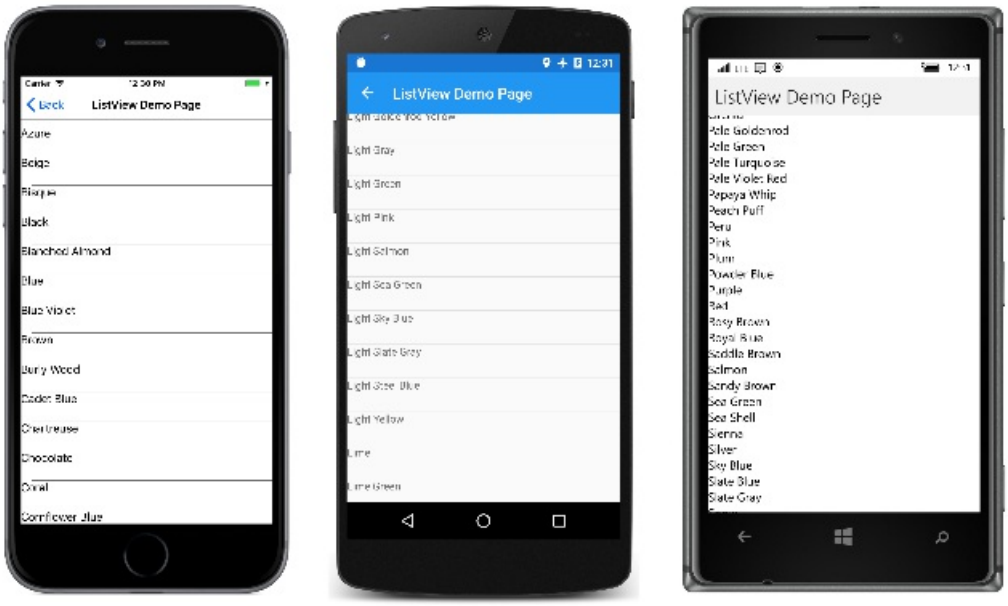


它不是太多信息，但 `ListView` 是可滚动和可选择。

若要定义的项模板，您需要跳出 `ItemTemplate` 属性设置为属性元素中，将其设置为 `DataTemplate`，然后引用 `ViewCell`。向 `View` 属性的 `ViewCell` 可以定义一个或多个视图以显示每个项的布局。下面是一个简单的示例：

```
<ListView ItemsSource="{x:Static local:NamedColor.All}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <ViewCell.View>
                    <Label Text="{Binding FriendlyName}" />
                </ViewCell.View>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

`Label` 元素设置为 `View` 属性的 `ViewCell`。（`ViewCell.View` 不需要标记，因为 `View` 属性是 content 属性 `ViewCell`。）此标记将显示 `FriendlyName` 每个属性 `NamedColor` 对象：



好多了。现在需要的是项模板的详细信息和实际颜色条理。若要支持此模板，某些值的对象定义和页面的资源字典中：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples"
             x:Class="XamlSamples.ListViewDemoPage"
             Title="ListView Demo Page">

    <ContentPage.Resources>
        <ResourceDictionary>
            <OnPlatform x:Key="boxSize"
                        x:TypeArguments="x:Double">
                <On Platform="iOS, Android, UWP" Value="50" />
            </OnPlatform>

            <OnPlatform x:Key="rowHeight"
                        x:TypeArguments="x:Int32">
                <On Platform="iOS, Android, UWP" Value="60" />
            </OnPlatform>

            <local:DoubleToIntConverter x:Key="intConverter" />

        </ResourceDictionary>
    </ContentPage.Resources>

    <ListView ItemsSource="{x:Static local:NamedColor.All}"
              RowHeight="{StaticResource rowHeight}">
        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <StackLayout Padding="5, 5, 0, 5"
                                 Orientation="Horizontal"
                                 Spacing="15">

                        <BoxView WidthRequest="{StaticResource boxSize}"
                                 HeightRequest="{StaticResource boxSize}"
                                 Color="{Binding Color}" />

                        <StackLayout Padding="5, 0, 0, 0"
                                     VerticalOptions="Center">

                            <Label Text="{Binding FriendlyName}"
                                   FontAttributes="Bold"
                                   FontSize="Medium" />

                            <StackLayout Orientation="Horizontal"
                                         Spacing="0">
                                <Label Text="{Binding Color.R,
                                       Converter={StaticResource intConverter},
                                       ConverterParameter=255,
                                       StringFormat='R={0:X2}'}" />

                                <Label Text="{Binding Color.G,
                                       Converter={StaticResource intConverter},
                                       ConverterParameter=255,
                                       StringFormat=', G={0:X2}'}" />

                                <Label Text="{Binding Color.B,
                                       Converter={StaticResource intConverter},
                                       ConverterParameter=255,
                                       StringFormat=', B={0:X2}'}" />
                            </StackLayout>
                        </StackLayout>
                    </StackLayout>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

请注意，使用 `OnPlatform` 来定义的大小 `BoxView` 和的高度 `ListView` 行。尽管所有三个平台的值是相同的标记可能容易就可以调整为其他值，以微调显示。

绑定值转换器

以前**ListView 演示**XAML 文件显示各个 `R`， `G`，并 `B` Xamarin.Forms 属性 `Color` 结构。这些属性属于类型 `double` 和 0 到 1 范围。如果你想要显示的十六进制值，不能只使用 `StringFormat` 与格式规范"X2"。仅用于范围内的整数，而，除了 `double` 需要乘以 255 的值。

这个小问题已解决了与*值转换器*，也称为*绑定转换器*。这是一个类实现 `IValueConverter` 接口，这意味着它具有两个方法名为 `Convert` 和 `ConvertBack` 。 `Convert` 值从源传输到目标; 当调用方法 `ConvertBack` 方法调用进行传输从目标到源中 `OneWayToSource` 或 `TwoWay` 绑定：

```
using System;
using System.Globalization;
using Xamarin.Forms;

namespace XamlSamples
{
    class DoubleToIntConverter : IValueConverter
    {
        public object Convert(object value, Type targetType,
                              object parameter, CultureInfo culture)
        {
            double multiplier;

            if (!Double.TryParse(parameter as string, out multiplier))
                multiplier = 1;

            return (int)Math.Round(multiplier * (double)value);
        }

        public object ConvertBack(object value, Type targetType,
                                  object parameter, CultureInfo culture)
        {
            double divider;

            if (!Double.TryParse(parameter as string, out divider))
                divider = 1;

            return ((double)(int)value) / divider;
        }
    }
}
```

`ConvertBack` 因为绑定仅一种方法从源到目标方法无法在此程序中播放一个角色。

绑定引用绑定转换器，且有 `Converter` 属性。绑定转换器还可以接受使用指定的参数 `ConverterParameter` 属性。对于一些更加通用，这是如何指定乘数。绑定转换器检查是否存在有效的转换器参数 `double` 值。

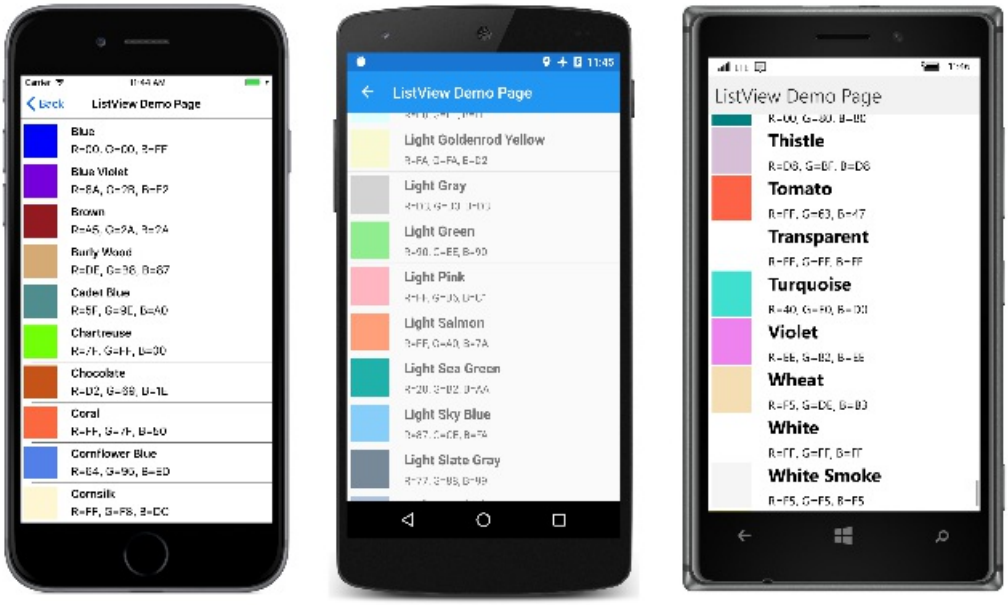
转换器是实例化资源字典中，因此它可以在多个绑定之间共享：

```
<local:DoubleToIntConverter x:Key="intConverter" />
```

三个数据绑定引用此单一实例。请注意， `Binding` 标记扩展包含一个嵌入 `StaticResource` 标记扩展：

```
<Label Text="{Binding Color.R,
                      Converter={StaticResource intConverter},
                      ConverterParameter=255,
                      StringFormat='R={0:X2}'}" />
```

下面是结果：



`ListView`中处理动态如果执行某些步骤在基础数据，但是仅限中可能出现的更改非常完善。如果项的集合分配给`ItemsSource`的属性`ListView`在运行时更改 — 的是，如果可以将项添加到或从集合中移除 — 使用`ObservableCollection`这些项的类。`ObservableCollection`实现`INotifyCollectionChanged`接口，并`ListView`将安装的处理程序`CollectionChanged`事件。

如果在运行时，更改项目本身的属性，则集合中的项应实现`INotifyPropertyChanged`对使用的属性值的接口和信号更改`PropertyChanged`事件。在本系列的下一部分对此进行了演示第 5 部分。从数据绑定到 MVVM。

总结

数据绑定提供了用于链接在页面内的两个对象之间或视觉对象之间的属性和基础数据的强大机制。但在应用程序开始使用数据源时，常用的应用程序体系结构模式开始显现出来，它作为一种有用的模式。中对此进行了第 5 部分。从数据绑定到 MVVM。

相关链接

- XamlSamples
- 第 1 部分：开始使用 XAML（示例）
- 第 2 部分：基本 XAML 语法（示例）
- 第 3 部分：XAML 标记扩展（示例）
- 第 5 部分：从数据绑定到 MVVM（示例）

第 5 部分。从数据绑定到 MVVM

2018/11/13 • • Edit Online

模型-视图-视图模型 (MVVM) 体系结构模式的发明时与 XAML 记住。该模式会强制实施三个软件层之间的隔离，XAML 用户界面，称为视图;基础数据，称为模型;并且在视图和模型之间的中介调用 ViewModel。通过 XAML 文件中定义的数据绑定通常连接中的视图和 ViewModel。视图 BindingContext 通常是 ViewModel 的实例。

简单的 ViewModel

为 Viewmodel 的简介，让我们先来看一下不存在的程序。前面你已了解如何在定义新的 XML 命名空间声明，以允许到其他程序集中的引用类的 XAML 文件。下面是一个程序，用于定义 XML 命名空间声明为 `System` 命名空间：

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

该程序可以使用 `x:Static` 若要获取当前日期和时间从静态 `DateTime.Now` 属性设置该 `DateTime` 值设置为 `BindingContext` 上 `StackLayout`：

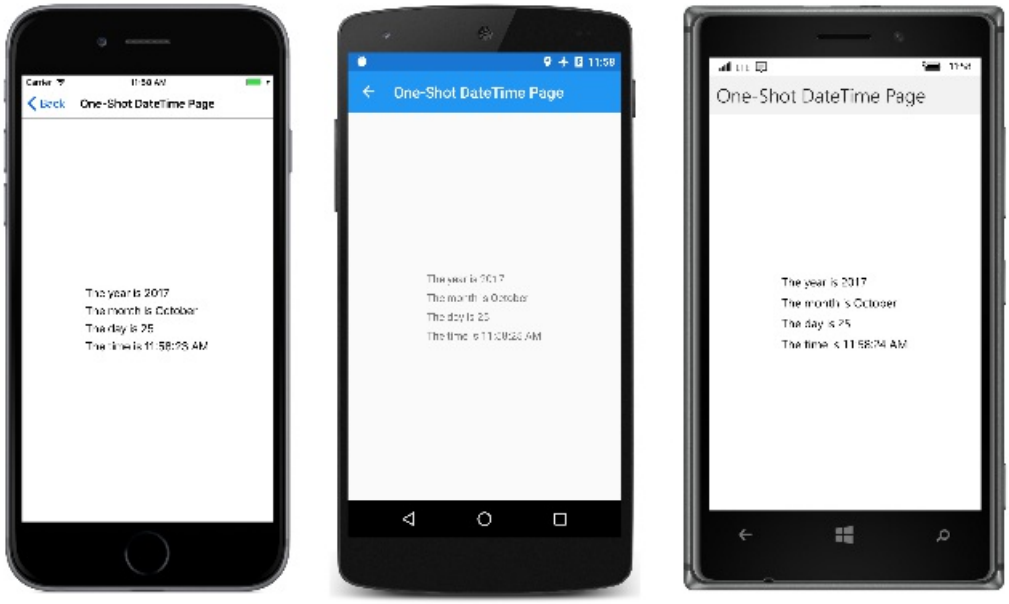
```
<StackLayout BindingContext="{x:Static sys:DateTime.Now}" …>
```

`BindingContext` 是一个非常特殊属性：当设置 `BindingContext` 某个元素上，它由继承该元素的所有子级。这意味着所有子级 `StackLayout` 具有此相同 `BindingContext`，并且它们可以包含简单绑定到该对象的属性。

在中**One-Shot DateTime**程序，两个子级包含到这些属性的绑定 `DateTime` 值，但两个其他的子级包含似乎缺少绑定路径的绑定。这意味着 `DateTime` 本身的值用于 `StringFormat`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             x:Class="XamlSamples.OneShotDateTimePage"
             Title="One-Shot DateTime Page">

    <StackLayout BindingContext="{x:Static sys:DateTime.Now}"
                 HorizontalOptions="Center"
                 VerticalOptions="Center">

        <Label Text="{Binding Year, StringFormat='The year is {0}'}" />
        <Label Text="{Binding StringFormat='The month is {0:MMMM}'}" />
        <Label Text="{Binding Day, StringFormat='The day is {0}'}" />
        <Label Text="{Binding StringFormat='The time is {0:T}'}" />

    </StackLayout>
</ContentPage>
```

当然，最大的问题是日期和时间是集后首次生成页面时，并且永远不会更改：



XAML 文件可以显示时钟始终会显示当前时间，但它需要一些代码来帮助解决问题。当 MVVM、模型和 ViewModel 方面的想法是完全用代码编写的类。视图通常是 XAML 引用的文件，通过数据绑定在 ViewModel 中定义的属性。

正确的模型是未知的 ViewModel 中，并适当的 ViewModel 是未知的视图。但是，通常一名程序员通过公开 ViewModel 与特定用户界面相关联的数据类型的数据类型。例如，如果模型访问数据库，其中包含 8 位字符的 ASCII 字符串时，ViewModel 会需要这些字符串转换为 Unicode 字符串，以适应独占使用的用户界面中的 Unicode 之间转换。

在简单示例中的 MVVM（如那些如下所示），通常没有模型，和的模式涉及只是一个视图和 ViewModel 与数据绑定链接。

下面是具有只是单个属性名为时钟的 ViewModel `DateTime`，但该更新的`DateTime`属性每秒：

```
using System;
using System.ComponentModel;
using Xamarin.Forms;

namespace XamlSamples
{
    class ClockViewModel : INotifyPropertyChanged
    {
        DateTime dateTime;

        public event PropertyChangedEventHandler PropertyChanged;

        public ClockViewModel()
        {
            this.DateTime = DateTime.Now;

            Device.StartTimer(TimeSpan.FromSeconds(1), () =>
                {
                    this.DateTime = DateTime.Now;
                    return true;
                });
        }

        public DateTime DateTime
        {
            set
            {
                if (dateTime != value)
                {
                    dateTime = value;

                    if (PropertyChanged != null)
                    {
                        PropertyChanged(this, new PropertyChangedEventArgs("DateTime"));
                    }
                }
            }
            get
            {
                return dateTime;
            }
        }
    }
}
```

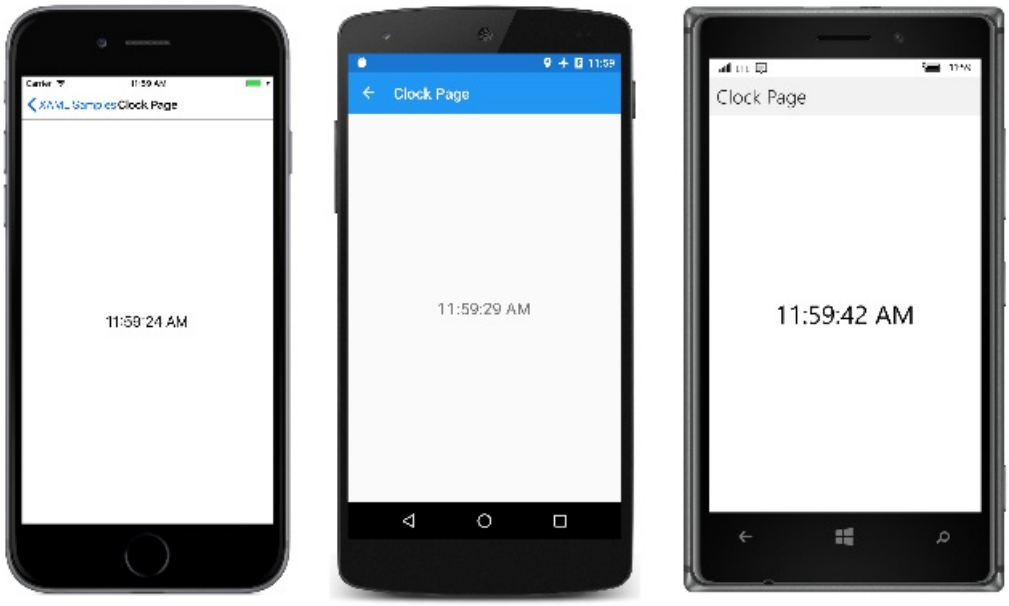
通常实现 Viewmodel `INotifyPropertyChanged` 接口，这意味着，将引发类 `PropertyChanged` 事件的一个属性发生更改时。在 Xamarin.Forms 中的数据绑定机制将处理程序附加到此 `PropertyChanged` 事件以便其属性更改时可以收到通知并保留目标更新为新值。

基于此 ViewModel 时钟可以是简单，如下：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
             x:Class="XamlSamples.ClockPage"
             Title="Clock Page">

    <Label Text="{Binding DateTime, StringFormat='{0:T}'}"
           FontSize="Large"
           HorizontalOptions="Center"
           VerticalOptions="Center">
        <Label.BindingContext>
            <local:ClockViewModel />
        </Label.BindingContext>
    </Label>
</ContentPage>
```

请注意如何 `ClockViewModel` 设置为 `BindingContext` 的 `Label` 使用属性元素标记。或者，您可以实例化 `ClockViewModel` 中 `Resources` 集合并将其设置为 `BindingContext` 通过 `StaticResource` 标记扩展。或者，代码隐藏文件可以实例化 ViewModel。

`Binding` 上的标记扩展 `Text` 的属性 `Label` 格式 `DateTime` 属性。下面是显示：



还有可能访问的各个属性 `DateTime` 用句点分隔属性 ViewModel 属性：

```
<Label Text="{Binding DateTime.Second, StringFormat='{0}'}" … >
```

交互式 MVVM

MVVM 是经常用于交互式视图基于的基础数据模型的双向数据绑定。

下面是一个名为类 `HslViewModel`，用于将 `Color` 值到 `Hue`， `Saturation`，和 `Luminosity` 值，反之亦然：

```
using System;
using System.ComponentModel;
using Xamarin.Forms;

namespace XamlSamples
{
    public class HslViewModel : INotifyPropertyChanged
    {
        double hue, saturation, luminosity;
```

```
        Color color;

        public event PropertyChangedEventHandler PropertyChanged;

        public double Hue
        {
            set
            {
                if (hue != value)
                {
                    hue = value;
                    OnPropertyChanged("Hue");
                    SetNewColor();
                }
            }
            get
            {
                return hue;
            }
        }

        public double Saturation
        {
            set
            {
                if (saturation != value)
                {
                    saturation = value;
                    OnPropertyChanged("Saturation");
                    SetNewColor();
                }
            }
            get
            {
                return saturation;
            }
        }

        public double Luminosity
        {
            set
            {
                if (luminosity != value)
                {
                    luminosity = value;
                    OnPropertyChanged("Luminosity");
                    SetNewColor();
                }
            }
            get
            {
                return luminosity;
            }
        }

        public Color Color
        {
            set
            {
                if (color != value)
                {
                    color = value;
                    OnPropertyChanged("Color");

                    Hue = value.Hue;
                    Saturation = value.Saturation;
                    Luminosity = value.Luminosity;
                }
            }
```

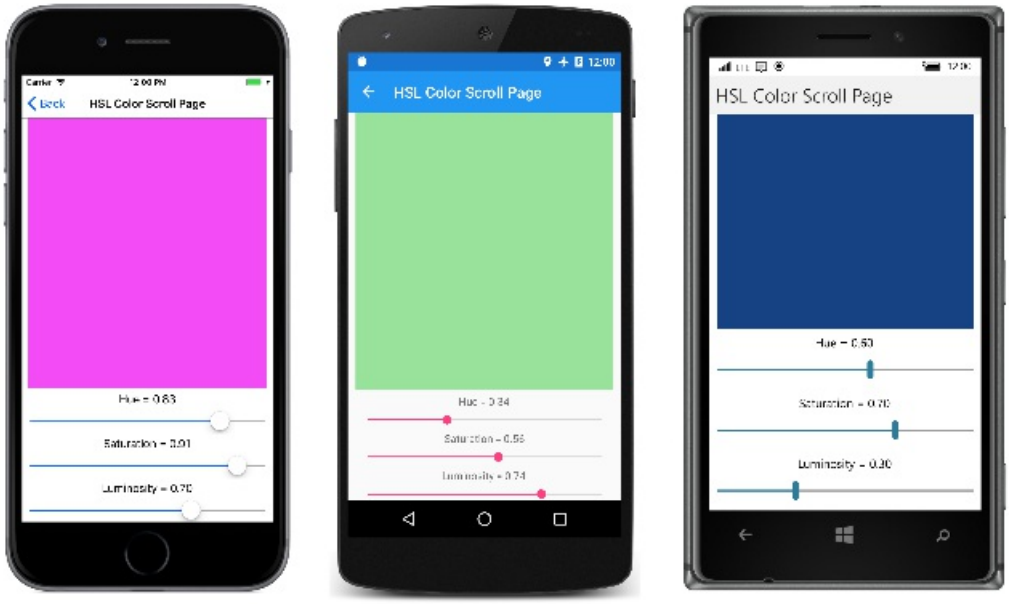
```
            }
            get
            {
                return color;
            }
        }

        void SetNewColor()
        {
            Color = Color.FromHsla(Hue, Saturation, Luminosity);
        }

        protected virtual void OnPropertyChanged(string propertyName)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

将更改为`Hue`，`Saturation`，并`Luminosity`属性的原因`Color`属性更改，并对更改`Color`会导致其他三个属性以更改。这可能看起来好像无限循环，只不过不会调用类`PropertyChanged`事件除非实际上已更改的属性。这样便禁止了到否则为无法控制反馈循环。

下面的 XAML 文件包含`BoxView`其`Color`属性绑定到`Color` ViewModel 和三个属性`Slider`和三个`Label`视图绑定到`Hue`，`Saturation`，和`Luminosity`属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
             x:Class="XamlSamples.HslColorScrollPage"
             Title="HSL Color Scroll Page">
    <ContentPage.BindingContext>
        <local:HslViewModel Color="Aqua" />
    </ContentPage.BindingContext>

    <StackLayout Padding="10, 0">
        <BoxView Color="{Binding Color}"
                 VerticalOptions="FillAndExpand" />

        <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}"
               HorizontalOptions="Center" />

        <Slider Value="{Binding Hue, Mode=TwoWay}" />

        <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}"
               HorizontalOptions="Center" />

        <Slider Value="{Binding Saturation, Mode=TwoWay}" />

        <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}"
               HorizontalOptions="Center" />

        <Slider Value="{Binding Luminosity, Mode=TwoWay}" />
    </StackLayout>
</ContentPage>
```

每个绑定`Label`是默认`OneWay`。它只需要显示的值。但每个绑定`Slider`是`TwoWay`。这允许`Slider`从 ViewModel 进行初始化。请注意，`Color`属性设置为`Aqua` ViewModel 实例化时。但在更改`Slider`还需要在 ViewModel，然后计算新颜色中设置属性的新值。



与 Viewmodel 命令

在许多情况下，MVVM 模式被限制为数据项的操作：在视图中的用户界面对象的并行 ViewModel 中的数据对象。

但是，有时视图需要包含触发 ViewModel 中的各种操作的按钮。但不能包含 ViewModel `Clicked` 按钮处理程序，可能会阻塞到特定的用户界面范例 ViewModel 因为。

若要允许 Viewmodel 更独立于特定用户界面对象，但仍允许在 ViewModel 中，调用的方法*命令*存在接口。此命令接口支持在 Xamarin.Forms 中的以下元素：

- `Button`
- `MenuItem`
- `ToolbarItem`
- `SearchBar`
- `TextCell`（因此也 `ImageCell`）
- `ListView`
- `TapGestureRecognizer`

除 `SearchBar` 和 `ListView` 元素，这些元素定义两个属性：

- `Command` 类型 `System.Windows.Input.ICommand`
- `CommandParameter` 类型 `Object`

`SearchBar` 定义 `SearchCommand` 并 `SearchCommandParameter` 属性，而 `ListView` 定义 `RefreshCommand` 类型的属性 `ICommand`。

`ICommand` 接口定义两个方法和一个事件：

- `void Execute(object arg)`
- `bool CanExecute(object arg)`
- `event EventHandler CanExecuteChanged`

ViewModel 可以定义类型的属性 `ICommand`。然后可以将绑定到这些属性 `Command` 每个属性 `Button` 或其他元素或可能是实现此接口的自定义视图。可以选择性地设置 `CommandParameter` 属性标识单个 `Button` 对象（或其他元素）的绑定到此 ViewModel 属性。在内部，`Button` 调用 `Execute` 方法，只要用户点击 `Button`，并传递到 `Execute` 方法及其 `CommandParameter`。

`CanExecute` 方法和 `CanExecuteChanged` 事件的情况下使用其中 `Button` 点击也可能是当前无效，在这种情况下 `Button` 应禁用其自身。 `Button` 调用 `CanExecute` 时 `Command` 先设置属性和每当 `CanExecuteChanged` 触发事件。如果 `CanExecute` 将返回 `false`，则 `Button` 禁用其自身，不会生成 `Execute` 调用。

有关添加到将 Viewmodel 命令的帮助，Xamarin.Forms 定义了两个类实现 `ICommand`：`Command` 并 `Command<T>` 其中 `T` 是函数的参数的类型 `Execute` 和 `CanExecute` 。这两个类定义多个构造函数加上 `ChangeCanExecute` ViewModel 可调用以强制方法 `Command` 对象以触发 `CanExecuteChanged` 事件。

下面是一个简单的键盘，用于输入电话号码的 ViewModel。请注意， `Execute` 和 `CanExecute` 方法被定义为构造函数中的 lambda 函数权限：

```
using System;
using System.ComponentModel;
using System.Windows.Input;
using Xamarin.Forms;

namespace XamlSamples
{
    class KeypadViewModel : INotifyPropertyChanged
    {
        string inputString = "";
        string displayText = "";
        char[] specialChars = { '*', '#' };

        public event PropertyChangedEventHandler PropertyChanged;

        // Constructor
        public KeypadViewModel()
        {
            AddCharCommand = new Command<string>((key) =>
                {
                    // Add the key to the input string.
                    InputString += key;
                });

            DeleteCharCommand = new Command(() =>
                {
                    // Strip a character from the input string.
                    InputString = InputString.Substring(0, InputString.Length - 1);
                },
                () =>
                {
                    // Return true if there's something to delete.
                    return InputString.Length > 0;
                });
        }

        // Public properties
        public string InputString
        {
            protected set
            {
                if (inputString != value)
                {
                    inputString = value;
                    OnPropertyChanged("InputString");
                    DisplayText = FormatText(inputString);

                    // Perhaps the delete button must be enabled/disabled.
                    ((Command)DeleteCharCommand).ChangeCanExecute();
                }
            }

            get { return inputString; }
        }
```

```
        public string DisplayText
        {
            protected set
            {
                if (displayText != value)
                {
                    displayText = value;
                    OnPropertyChanged("DisplayText");
                }
            }
            get { return displayText; }
        }

        // ICommand implementations
        public ICommand AddCharCommand { protected set; get; }

        public ICommand DeleteCharCommand { protected set; get; }

        string FormatText(string str)
        {
            bool hasNonNumbers = str.IndexOfAny(specialChars) != -1;
            string formatted = str;

            if (hasNonNumbers || str.Length < 4 || str.Length > 10)
            {
            }
            else if (str.Length < 8)
            {
                formatted = String.Format("{0}-{1}",
                                          str.Substring(0, 3),
                                          str.Substring(3));
            }
            else
            {
                formatted = String.Format("({0}) {1}-{2}",
                                          str.Substring(0, 3),
                                          str.Substring(3, 3),
                                          str.Substring(6));
            }
            return formatted;
        }

        protected void OnPropertyChanged(string propertyName)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

此 ViewModel 假定 `AddCharCommand` 属性绑定到 `Command` 属性的多个按钮（或任何其他命令接口的操作），其中每个由 `CommandParameter` 。这些按钮添加到字符 `InputString` 属性，然后格式化的电话号码为 `DisplayText` 属性。

此外，还有第二个类型的属性 `ICommand` 名为 `DeleteCharCommand` 。这绑定到一个后间距按钮，但如果没有要删除的字符，则应禁用按钮。

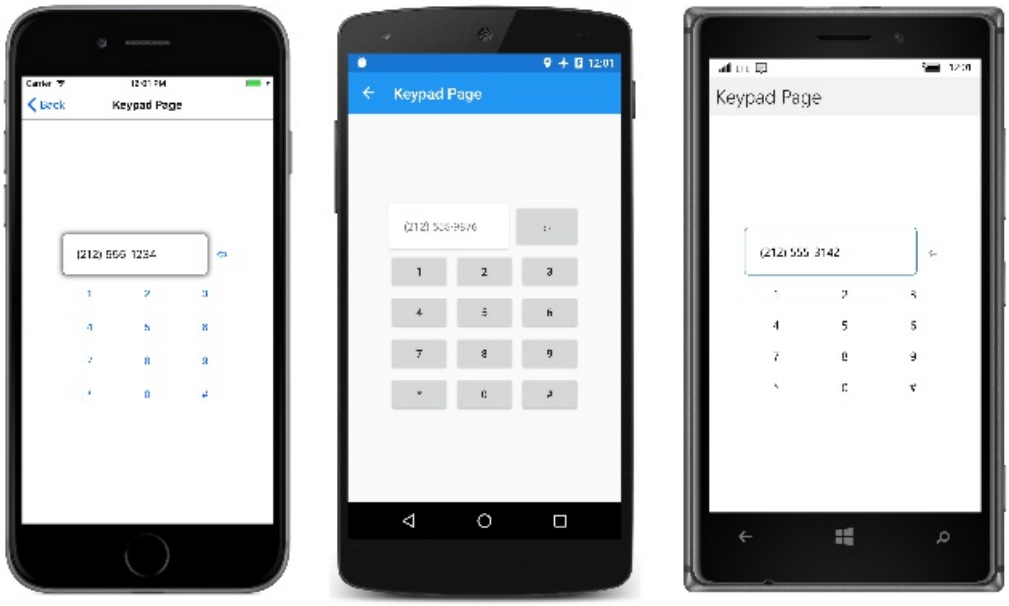
因为这可能是以下键盘不是如直观复杂。相反，标记已减至最少以演示更清楚地命令接口的使用：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples;assembly=XamlSamples"
             x:Class="XamlSamples.KeypadPage"
             Title="Keypad Page">

    <Grid HorizontalOptions="Center"
          VerticalOptions="Center">
        <Grid.BindingContext>
```

```
    <local:KeypadViewModel />
</Grid.BindingContext>

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="80" />
    <ColumnDefinition Width="80" />
    <ColumnDefinition Width="80" />
</Grid.ColumnDefinitions>

<!-- Internal Grid for top row of items -->
<Grid Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Frame Grid.Column="0"
           OutlineColor="Accent">
        <Label Text="{Binding DisplayText}" />
    </Frame>

    <Button Text="&#x21E6;"
            Command="{Binding DeleteCharCommand}"
            Grid.Column="1"
            BorderWidth="0" />
</Grid>

<Button Text="1"
        Command="{Binding AddCharCommand}"
        CommandParameter="1"
        Grid.Row="1" Grid.Column="0" />

<Button Text="2"
        Command="{Binding AddCharCommand}"
        CommandParameter="2"
        Grid.Row="1" Grid.Column="1" />

<Button Text="3"
        Command="{Binding AddCharCommand}"
        CommandParameter="3"
        Grid.Row="1" Grid.Column="2" />

<Button Text="4"
        Command="{Binding AddCharCommand}"
        CommandParameter="4"
        Grid.Row="2" Grid.Column="0" />

<Button Text="5"
        Command="{Binding AddCharCommand}"
        CommandParameter="5"
        Grid.Row="2" Grid.Column="1" />

<Button Text="6"
        Command="{Binding AddCharCommand}"
        CommandParameter="6"
        Grid.Row="2" Grid.Column="2" />

<Button Text="7"
        Command="{Binding AddCharCommand}"
        CommandParameter="7"
        Grid.Row="3" Grid.Column="0" />
```

```
        <Button Text="8"
                Command="{Binding AddCharCommand}"
                CommandParameter="8"
                Grid.Row="3" Grid.Column="1" />

        <Button Text="9"
                Command="{Binding AddCharCommand}"
                CommandParameter="9"
                Grid.Row="3" Grid.Column="2" />

        <Button Text="*"
                Command="{Binding AddCharCommand}"
                CommandParameter="*"
                Grid.Row="4" Grid.Column="0" />

        <Button Text="0"
                Command="{Binding AddCharCommand}"
                CommandParameter="0"
                Grid.Row="4" Grid.Column="1" />

        <Button Text="#"
                Command="{Binding AddCharCommand}"
                CommandParameter="#"
                Grid.Row="4" Grid.Column="2" />
    </Grid>
</ContentPage>
```

`Command`属性的第一个`Button`出现在此标记绑定到`DeleteCharCommand`；其余绑定到`AddCharCommand`与`CommandParameter`显示的字符，它是相同`Button`人脸。下面是该程序的操作：



调用异步方法

命令还可以调用异步方法。这通过使用实现`async`并`await`关键字指定时`Execute`方法：

```
DownloadCommand = new Command (async () => await DownloadAsync ());
```

这指示`DownloadAsync`方法是`Task`和应处于等待状态：

```
async Task DownloadAsync ()
{
    await Task.Run (() => Download ());
}

void Download ()
{
    ...
}
```

实现导航菜单

XamlSamples包含在本系列的文章中的所有源代码的程序使用 ViewModel 其主页。此 ViewModel 是具有名为三个属性的简短类的定义 `Type`、`Title`，和 `Description` 的包含类型的每个示例页面、标题和简短说明。此外，ViewModel 定义名为的静态属性 `All`，它是在程序中的所有页的集合：

```
public class PageDataViewModel
{
    public PageDataViewModel(Type type, string title, string description)
    {
        Type = type;
        Title = title;
        Description = description;
    }

    public Type Type { private set; get; }

    public string Title { private set; get; }

    public string Description { private set; get; }

    static PageDataViewModel()
    {
        All = new List<PageDataViewModel>
        {
            // Part 1. Getting Started with XAML
            new PageDataViewModel(typeof(HelloXamlPage), "Hello, XAML",
                                  "Display a Label with many properties set"),

            new PageDataViewModel(typeof(XamlPlusCodePage), "XAML + Code",
                                  "Interact with a Slider and Button"),

            // Part 2. Essential XAML Syntax
            new PageDataViewModel(typeof(GridDemoPage), "Grid Demo",
                                  "Explore XAML syntax with the Grid"),

            new PageDataViewModel(typeof(AbsoluteDemoPage), "Absolute Demo",
                                  "Explore XAML syntax with AbsoluteLayout"),

            // Part 3. XAML Markup Extensions
            new PageDataViewModel(typeof(SharedResourcesPage), "Shared Resources",
                                  "Using resource dictionaries to share resources"),

            new PageDataViewModel(typeof(StaticConstantsPage), "Static Constants",
                                  "Using the x:Static markup extensions"),

            new PageDataViewModel(typeof(RelativeLayoutPage), "Relative Layout",
                                  "Explore XAML markup extensions"),

            // Part 4. Data Binding Basics
            new PageDataViewModel(typeof(SliderBindingsPage), "Slider Bindings",
                                  "Bind properties of two views on the page"),

            new PageDataViewModel(typeof(SliderTransformsPage), "Slider Transforms",
```

```
                                "Use Sliders with reverse bindings"),

            new PageDataViewModel(typeof(ListViewDemoPage), "ListView Demo",
                                "Use a ListView with data bindings"),

            // Part 5. From Data Bindings to MVVM
            new PageDataViewModel(typeof(OneShotDateTimePage), "One-Shot DateTime",
                                "Obtain the current DateTime and display it"),

            new PageDataViewModel(typeof(ClockPage), "Clock",
                                "Dynamically display the current time"),

            new PageDataViewModel(typeof(HslColorScrollPage), "HSL Color Scroll",
                                "Use a view model to select HSL colors"),

            new PageDataViewModel(typeof(KeypadPage), "Keypad",
                                "Use a view model for numeric keypad logic")
        };
    }

    public static IList<PageDataViewModel> All { private set; get; }
}
```

XAML 文件 `MainPage` 定义 `ListBox` 其 `ItemsSource` 属性设置为该 `All` 属性，它包含 `TextCell` 用于显示 `Title` 和 `Description` 每一页的属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:XamlSamples"
             x:Class="XamlSamples.MainPage"
             Padding="5, 0"
             Title="XAML Samples">

    <ListView ItemsSource="{x:Static local:PageDataViewModel.All}"
              ItemSelected="OnListViewItemSelected">
        <ListView.ItemTemplate>
            <DataTemplate>
                <TextCell Text="{Binding Title}"
                          Detail="{Binding Description}" />
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

可滚动列表中显示页面：

当用户选择某个项时触发的代码隐藏文件中的处理程序。处理程序集 `SelectedItem` 的属性 `ListBox` 回 `null` 然后实例化所选的页面并导航到它：

```
private async void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
{
    (sender as ListView).SelectedItem = null;

    if (args.SelectedItem != null)
    {
        PageDataViewModel pageData = args.SelectedItem as PageDataViewModel;
        Page page = (Page)Activator.CreateInstance(pageData.Type);
        await Navigation.PushAsync(page);
    }
}
```

视频

可以轻松地 Xamarin.Forms 和 Prism Xamarin Evolve 2016: MVVM

总结

XAML 是用于在 Xamarin.Forms 应用程序，尤其是在数据绑定中定义的用户界面的强大工具，使用 MVVM。结果是在代码中的所有后台支持具有的用户界面的干净、巧妙，且可能可工具化表示形式。

相关链接

- XamlSamples
- 第 1 部分：XAML 入门
- 第 2 部分：基本 XAML 语法
- 第 3 部分：XAML 标记扩展
- 第 4 部分：数据绑定基础知识

在 Xamarin.Forms 中 XAML 编译

2018/11/13 • • Edit Online

XAML 可根据需要编译为中间语言 (IL) 使用 XAML 编译器 (XAMLC) 直接。

XAML 编译提供了很多好处：

- 它会执行 XAML 的编译时检查，从而可向用户通知任何错误。
- 它会消除 XAML 元素的某些负载和实例化时间。
- 它通过不再包含 .xaml 文件，来帮助减小最终程序集的文件大小。

若要确保向后兼容性的默认情况下禁用 XAML 编译。它可以在程序集和类级别启用通过添加 `XamlCompilation` 属性。

下面的代码示例演示如何在程序集级别启用 XAML 编译：

```
using Xamarin.Forms.Xaml;
...
[assembly: XamlCompilation (XamlCompilationOptions.Compile)]
namespace PhotoApp
{
  ...
}
```

在此示例中，编译时检查的程序集内包含的所有 XAML 将执行，但在编译时而不是在运行时报告的 XAML 出现错误。因此，`assembly` 为前缀 `XamlCompilation` 属性指定该属性应用于整个程序集。

> **NOTE**
> `XamlCompilation` 属性和 `XamlCompilationOptions` 枚举位于 `Xamarin.Forms.Xaml` 命名空间，必须导入使用它们。

下面的代码示例演示了在类级别启用 XAML 编译：

```
using Xamarin.Forms.Xaml;
...
[XamlCompilation (XamlCompilationOptions.Compile)]
public class HomePage : ContentPage
{
  ...
}
```

在此示例中，编译时检查的 XAML 的 `HomePage` 类将在执行和错误报告作为编译过程的一部分。

> **NOTE**
> 可以启用已编译的绑定，以提高 Xamarin.Forms 应用程序中的数据绑定性能。有关详细信息，请参阅编译绑定。

相关链接

- `XamlCompilation`
- `XamlCompilationOptions`

Xamarin.Forms XAML 工具箱

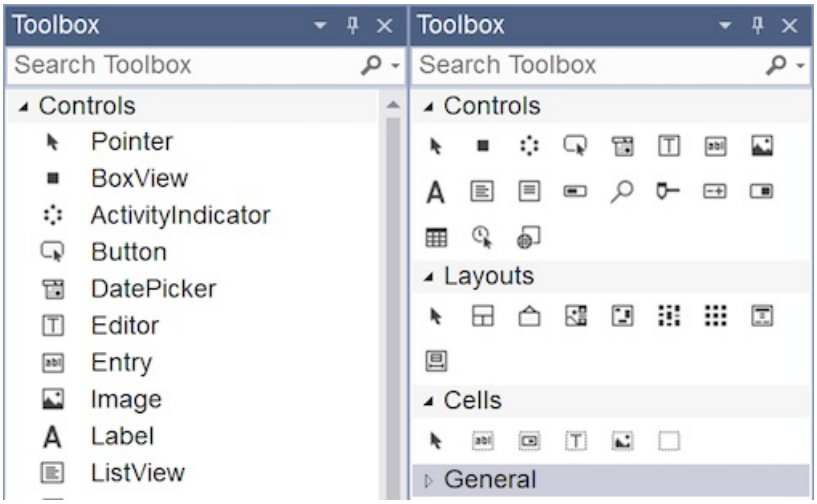
2018/10/26 • • Edit Online

Visual Studio 2017 版本 15.8 和 Visual Studio for Mac 7.6 现在具有工具箱可编辑 Xamarin.Forms XAML 文件时。工具箱中包含所有内置的 Xamarin.Forms 控件和布局，可以拖动到 XAML 编辑器。

- Visual Studio
- Visual Studio for Mac

在 Visual Studio 2017 中，打开 Xamarin.Forms XAML 文件进行编辑。工具箱可通过按所示**Ctrl + W**，**X**键盘，或选择**视图 > 工具箱**菜单项。

可以隐藏，像在 Visual Studio 2017 中，使用右上角或上下文菜单中的图标，其他窗格停靠工具箱。Xamarin.Forms XAML 工具箱具有自定义视图选项，可通过右键单击每个部分更改。切换**列表**视图选项列表和 compact 视图之间进行切换：



Xamarin.Forms XAML 文件打开进行编辑时，将任何控件或布局从工具箱拖到该文件，然后利用 Intellisense 自定义用户界面。

适用于 Xamarin.Forms 的 XAML 预览程序

2018/11/13 • • Edit Online

请参阅的呈现将如你所键入的 Xamarin.Forms 布局 ！

要求

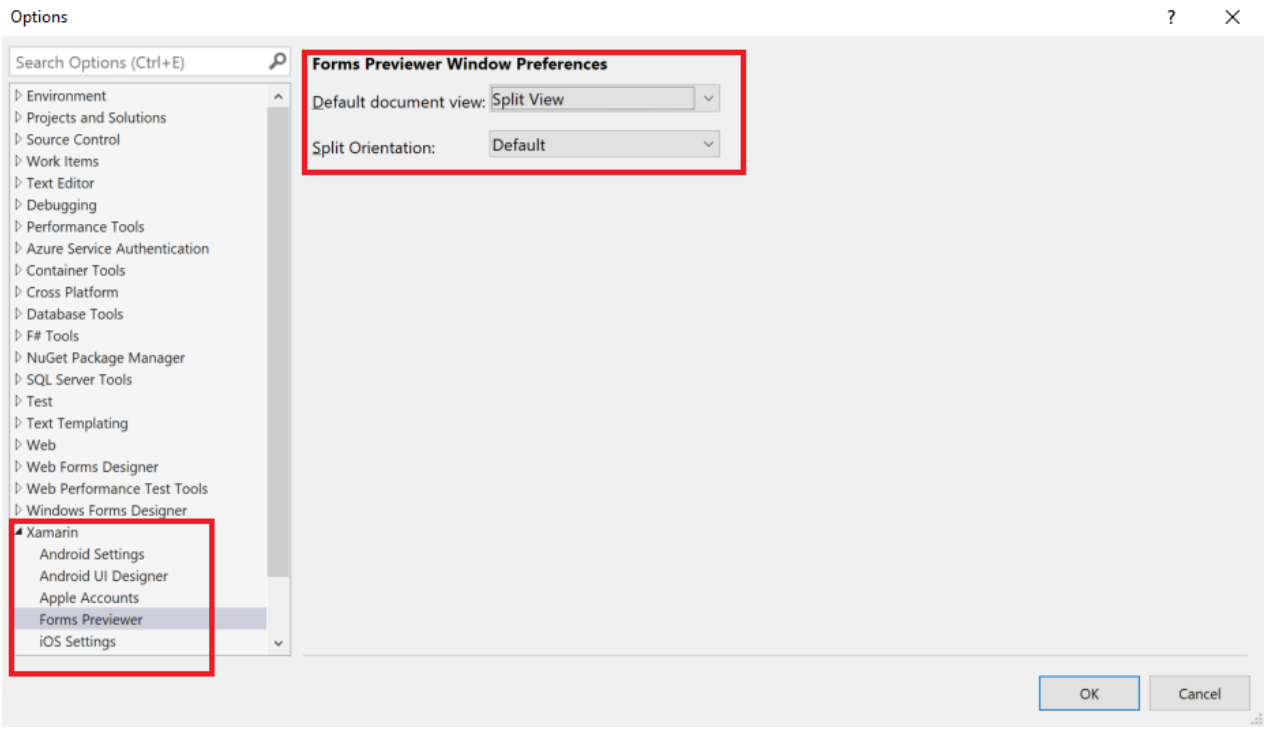
项目需要 XAML 预览程序若要运行的最新 Xamarin.Forms NuGet 包。预览 Android 应用程序需要JDK 1.8 x64。

中的详细信息发行说明。

入门

- Visual Studio
- Visual Studio for Mac

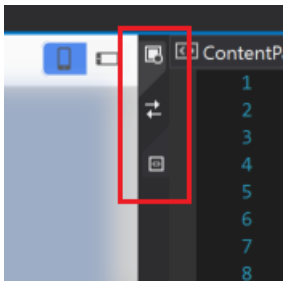
XAML 预览程序默认情况下，可以控制从**工具** > 选项 > **Xamarin** > **窗体**预览**程序**对话框。在此对话框可以选择默认文档视图和拆分方向。



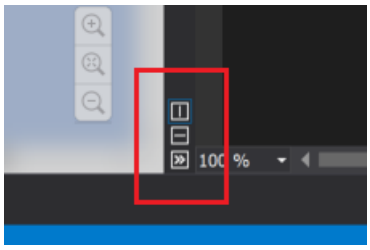
在设置中选择打开 XAML 页面编辑器将拆分的基于**工具** > 选项 > **Xamarin** > **窗体**预览**程序**对话框。但是，可以在编辑器窗口中更改这些首选项。

XAML 预览控件

编辑器窗口的顶部具有要选择哪个窗格正在使用中，切换到设计窗格顶部的按钮和切换到源窗格的底部按钮的按钮。中间的按钮交换窗格顺序。



编辑器窗口的底部有水平和垂直方向拆分窗格，以展开或折叠当前子窗格的按钮。



XAML 预览选项

预览窗格顶部的选项包括：

- **Phone** – 在电话大小屏幕中呈现
- **平板电脑**– 呈现在平板电脑大小屏幕（请注意在窗格的右下角有缩放控件）
- **Android** – 显示在屏幕的 Android 版本
- **iOS** – 显示在屏幕的 iOS 版本
- Portait（图标）– 使用纵向方向预览版
- **横向**（图标）– 使用横向方向预览版

添加设计时数据

某些布局可能很难直观显示不包含任何数据绑定到用户界面控件。若要使预览更有用，分配一些静态数据的控件进行硬编码的绑定上下文（不管是在代码隐藏中或使用 XAML）。

请参阅 James Montemagno博客文章上添加设计时数据若要了解如何将绑定到 XAML 中静态 ViewModel。

检测设计模式

静态 `DesignMode.IsDesignModeEnabled` 属性可以检查以确定是否在预览程序中运行应用程序。这使您可以指定在预览程序中运行应用程序时，将仅执行的代码：

```
if (DesignMode.IsDesignModeEnabled)
{
  // Previewer only code
}
```

疑难解答

检查以下问题并Xamarin 论坛，如果遇到问题。

未显示 XAML 预览

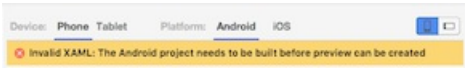
请检查是否属于以下情况：

- 项目应构建（编译）然后再尝试预览 XAML 文件。
- 设计器代理必须安装第一次预览 XAML 文件-进度指示器将显示在预览程序，以及进度消息，直到准备就绪。

- 请尝试关闭并重新打开 XAML 文件。
- 确保你 `App` 类具有无参数构造函数。

XAML 无效：需要在创建预览之前生成 Android 项目

XAML 预览程序需要在呈现页面之前生成该项目。如果在预览窗格的顶部会出现以下错误，重新生成应用程序，然后重试。



在 Xamarin.Forms 中 XAML 命名空间

2018/11/13 • • Edit Online

XAML 使用的命名空间声明的 xmlns XML 属性。本文介绍 XAML 命名空间语法，并演示如何声明 XAML 命名空间以访问的类型。

概述

有两个都是 XAML 文件的根元素中的 XAML 命名空间声明。第一个定义默认命名空间，如下面的 XAML 代码示例中所示：

```
xmlns="http://xamarin.com/schemas/2014/forms"
```

默认命名空间指定在没有任何前缀的 XAML 文件中定义的元素引用 Xamarin.Forms 类，如 `ContentPage` 。

第二个命名空间声明使用 `x` 前缀，如下面的 XAML 代码示例中所示：

```
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

XAML 使用前缀来引用命名空间中的类型时要使用的前缀与声明非默认命名空间。 `x` 命名空间声明指定带前缀的 XAML 中定义元素 `x` 用于元素和属性是固有的 XAML（特别是 2009 XAML 规范）。

下表概括了 `x` Xamarin.Forms 支持的命名空间属性：

构造	描述
`x:Arguments`	指定非默认构造函数或工厂方法对象声明的构造函数自变量。
`x:Class`	指定在 XAML 中定义的类的命名空间和类。类名必须与匹配的代码隐藏文件的类名。请注意，此构造只能出现在 XAML 文件的根元素。
`x:DataType`	指定的 XAML 元素，它的子级，将绑定到的对象的类型。
`x:FactoryMethod`	指定可用于初始化对象的工厂方法。
`x:FieldModifier`	指定的命名 XAML 元素生成的字段的访问级别。
`x:Key`	指定的每个资源的唯一用户键 `ResourceDictionary` 。键的值用于检索 XAML 资源，并通常用作参数 `StaticResource` 标记扩展。
`x:Name`	指定的 XAML 元素的运行时对象名称。设置 `x:Name` 类似于在代码中声明变量。
`x:TypeArguments`	指定的泛型类型参数的泛型类型构造函数。

有关详细信息 `x:DataType` 属性，请参阅编译绑定。有关详细信息 `x:FieldModifier` 属性，请参阅字段修饰符。有关详细信息 `x:Arguments` ， `x:FactoryMethod` ，并 `x:TypeArguments` 特性，请参见在 XAML 中传递参数。

在 XAML 中，命名空间声明从父元素继承到子元素。因此，在 XAML 文件的根元素中定义一个命名空间，该文件中的所有元素都继承的命名空间声明。

类型声明的命名空间

可以使用公共语言运行时 (CLR) 命名空间名称和程序集名称（可选）指定的命名空间声明中声明具有前缀的 XAML 命名空间，在 XAML 中引用类型。这被通过定义以下中的关键字命名空间声明的值：

- **clr 命名空间：或使用：** – 在包含要将公开为 XAML 元素的类型的程序集内的 CLR 命名空间声明。此关键字是必需的。
- **程序集 =** – 包含所引用的 CLR 命名空间的程序集。此值为程序集，不带文件扩展名的名称。应为包含将引用程序集的 XAML 文件的项目文件中的引用建立对程序集的路径。此关键字，则可省略**clr 命名空间**值是引用类型的应用程序代码在同一程序集中。

请注意，字符分隔 `clr-namespace` 或 `using` 标记和其值是一个冒号，而字符分隔 `assembly` 标记和其值是一个等号。要使用两个标记之间的字符是分号。

下面的代码示例演示了一个 XAML 命名空间声明：

```
<ContentPage ... xmlns:local="clr-namespace:HelloWorld" ...>
  ...
</ContentPage>
```

或者，这可以编写为：

```
<ContentPage ... xmlns:local="using:HelloWorld" ...>
  ...
</ContentPage>
```

`local` 前缀是用来指示命名空间中的类型是应用程序的本地约定。或者，如果类型属于不同程序集，程序集名称也应被定义在命名空间声明中，如以下 XAML 代码示例所示：

```
<ContentPage ... xmlns:behaviors="clr-namespace:Behaviors;assembly=BehaviorsLibrary" ...>
  ...
</ContentPage>
```

当将从导入的命名空间，类型的实例声明为以下 XAML 代码示例所示，然后指定命名空间前缀：

```
<ListView ...>
  <ListView.Behaviors>
    <behaviors:EventToCommandBehavior EventName="ItemSelected" ... />
  </ListView.Behaviors>
</ListView>
```

总结

本文引入 XAML 命名空间语法，并说明了如何以声明要访问的类型的 XAML 命名空间。XAML 使用 `xmlns` XML 属性的命名空间声明和类型，可以引用在 XAML 中声明具有前缀的 XAML 命名空间。

相关链接

- 在 XAML 中传递自变量

XAML 标记扩展

2018/11/13 • • Edit Online

XAML 标记扩展可帮助元素特性，若要从文本字符串以外的源设置，从而扩展的功能和灵活性的 XAML。

例如，正常情况下设置 `Color` 属性的 `BoxView` 如下所示：

```
<BoxView Color="Blue" />
```

或者，可以将其设置为十六进制的 RGB 颜色值：

```
<BoxView Color="#FF0080" />
```

在任一情况下，文本字符串设置为 `Color` 属性转换为 `Color` 的值 `ColorTypeConverter` 类。

您可能希望改为设置 `Color` 属性从存储中使用的资源字典的值或已创建的类的静态属性的值或类型的属性 `Color` 的页上，另一个元素或从构造分隔色调、饱和度和亮度值。

所有这些选项都可能使用 XAML 标记扩展。但不要让短语"标记扩展"被吓：XAML 标记扩展是*不*XML 的扩展。即使使用 XAML 标记扩展，XAML 始终是合法的 XML。

标记扩展是实际上只是以不同的方式来表示一个元素的属性。XAML 标记扩展是通常可识别括在大括号中的属性设置：

```
<BoxView Color="{StaticResource themeColor}" />
```

在大括号中的任何属性设置为*始终*XAML 标记扩展。但是，正如您将看到，XAML 标记扩展还可以引用而不使用大括号。

本文分为两个部分：

使用 XAML 标记扩展

使用 Xamarin.Forms 中定义的 XAML 标记扩展。

创建 XAML 标记扩展

编写您自己自定义的 XAML 标记扩展。

相关链接

- 标记扩展（示例）
- 从 Xamarin.Forms 书籍的 XAML 标记扩展一章
- 资源字典
- 动态样式
- 数据绑定

使用 XAML 标记扩展

2018/11/13 • • Edit Online

XAML 标记扩展帮助元素特性，若要设置从各种源，从而增强的功能和灵活性的 XAML。多个 XAML 标记扩展是 XAML 2009 规范的一部分。这些通常在与 XAML 文件中出现 `x` 命名空间前缀，且通常被引用到具有此前缀。本文讨论了以下标记扩展：

- `x:Static` – 引用静态属性、字段或枚举成员。
- `x:Reference` – 名为页面上的元素的引用。
- `x:Type` – 将属性设置为 `System.Type` 对象。
- `x:Array` – 构造特定类型的对象的数组。
- `x:Null` – 将属性设置为 `null` 值。
- `OnPlatform` -自定义根据每个平台的 UI 外观。
- `OnIdiom` -自定义 UI 外观基于的设备运行应用程序的惯用语法。

其他 XAML 标记扩展从历史上看其他 XAML 实现中，通过受支持和 Xamarin.Forms 还支持。这些更全面介绍了其他文章：

- `StaticResource` – 资源字典中引用的对象，如本文所述**资源字典**。
- `DynamicResource` – 响应中使用的资源字典中的对象的更改，如本文所述**动态样式**。
- `Binding` – 建立两个对象的属性之间的链接文章中所述**数据绑定**。
- `TemplateBinding` – 从控件模板，执行数据绑定，如本文所述**控件模板中绑定**。

`RelativeLayout` 布局使用自定义标记扩展 `ConstraintExpression` 。文章中介绍了此标记扩展 **RelativeLayout**。

x:Static 标记扩展

`x:Static` 标记扩展受 `StaticExtension` 类。类具有名为的单个属性 `Member` 类型的 `string` 设置为公共常量、静态属性、静态字段或枚举成员的名称。

一种常用的方式来使用 `x:Static` 是第一次定义类与某些常量或静态变量，如这小 `AppConstants` 类 **MarkupExtensions** 程序：

```
static class AppConstants
{
    public static double NormalFontSize = 18;
}
```

X：静态演示页上演示了多种方式来使用 `x:Static` 标记扩展。最繁琐的方法实例化 `StaticExtension` 类之间 `Label.FontSize` 属性元素标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             xmlns:local="clr-namespace:MarkupExtensions"
             x:Class="MarkupExtensions.StaticDemoPage"
             Title="x:Static Demo">
    <StackLayout Margin="10, 0">
        <Label Text="Label No. 1">
            <Label.FontSize>
                <x:StaticExtension Member="local:AppConstants.NormalFontSize" />
            </Label.FontSize>
        </Label>

        ...

    </StackLayout>
</ContentPage>
```

此 XAML 分析器还允许 `StaticExtension` 类，以缩写为 `x:Static`：

```
<Label Text="Label No. 2">
    <Label.FontSize>
        <x:Static Member="local:AppConstants.NormalFontSize" />
    </Label.FontSize>
</Label>
```

这可以更进一步简化，但更改，将引入某些新的语法：它包含将置于 `StaticExtension` 类和设置在大括号中的成员。所生成的表达式将直接为 `FontSize` 属性：

```
<Label Text="Label No. 3"
       FontSize="{x:StaticExtension Member=local:AppConstants.NormalFontSize}" />
```

请注意，存在*没有*引号引起来的大括号内。`Member` 属性的 `StaticExtension` 不再是一个 XML 属性。而是表达式的，它是表达式的标记扩展的一部分。

正如你可以将缩写 `x:StaticExtension` 到 `x:Static` 时您将其用作对象元素，还可以简化它的大括号内表达式中：

```
<Label Text="Label No. 4"
       FontSize="{x:Static Member=local:AppConstants.NormalFontSize}" />
```

`StaticExtension` 类具有 `ContentProperty` 属性引用的属性 `Member`，这会将此属性作为类的默认内容属性。对于使用大括号表示 XAML 标记扩展，则可以消除 `Member=` 表达式的一部分：

```
<Label Text="Label No. 5"
       FontSize="{x:Static local:AppConstants.NormalFontSize}" />
```

这是最常见的形式 `x:Static` 标记扩展。

静态演示页包含两个其他示例。XAML 文件的根标记的.net 中包含的 XML 命名空间声明 `System` 命名空间：

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

这允许 `Label` 字体大小设置为静态字段 `Math.PI`。该操作会相当小文本，因此 `Scale` 属性设置为 `Math.E`：

```
<Label Text="&#x03C0; &#x00D7; E sized text"
       FontSize="{x:Static sys:Math.PI}"
       Scale="{x:Static sys:Math.E}"
       HorizontalOptions="Center" />
```

最后一个示例显示 `Device.RuntimePlatform` 值。 `Environment.NewLine` 静态属性用来插入新行字符两者之间 `Span` 对象：

```
<Label HorizontalTextAlignment="Center"
       FontSize="{x:Static local:AppConstants.NormalFontSize}">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Runtime Platform: " />
            <Span Text="{x:Static sys:Environment.NewLine}" />
            <Span Text="{x:Static Device.RuntimePlatform}" />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

下面是在所有三个平台上运行的示例：

x:Reference 标记扩展

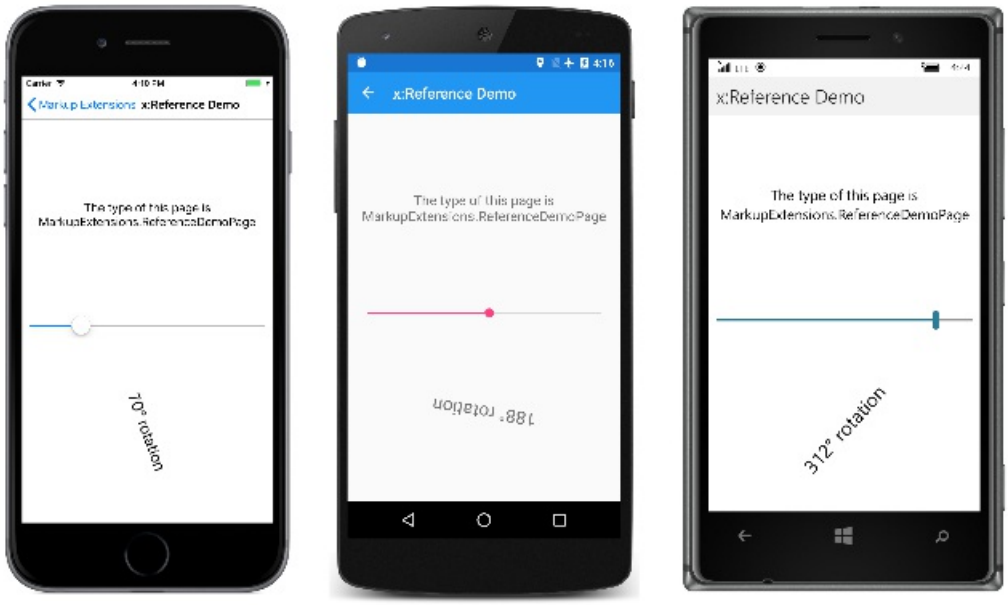
`x:Reference` 标记扩展受 `ReferenceExtension` 类。类具有名为的单个属性 `Name` 类型的 `string` 设置为它已被授予与名称页上的元素名称 `x:Name` 。这 `Name` 属性是内容的属性 `ReferenceExtension` ，因此 `Name=` 时不需要 `x:Reference` 在大括号中显示。

`x:Reference` 标记扩展以独占方式用于数据绑定，一文中的更详细地介绍**数据绑定**。

X:reference 演示页显示的两种用法 `x:Reference` 使用数据绑定，它用于设置的第一个 `Source` 属性 `Binding` 对象，并且它用于设置第二个 `BindingContext` 对于两个数据绑定的属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MarkupExtensions.ReferenceDemoPage"
             x:Name="page"
             Title="x:Reference Demo">

    <StackLayout Margin="10, 0">

        <Label Text="{Binding Source={x:Reference page},
                              StringFormat='The type of this page is {0}'}"
               FontSize="18"
               VerticalOptions="CenterAndExpand"
               HorizontalTextAlignment="Center" />

        <Slider x:Name="slider"
                Maximum="360"
                VerticalOptions="Center" />

        <Label BindingContext="{x:Reference slider}"
               Text="{Binding Value, StringFormat='{0:F0}&#x00B0; rotation'}"
               Rotation="{Binding Value}"
               FontSize="24"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

    </StackLayout>
</ContentPage>
```

这两`x:Reference`表达式使用的缩写的形式`ReferenceExtension`类名称，并消除`Name=`表达式的一部分。在第一个示例中，`x:Reference`标记扩展嵌入在`Binding`标记扩展。请注意，`Source`和`StringFormat`设置用逗号分隔。下面是在所有三个平台上运行的程序：



x:Type 标记扩展

`x:Type`标记扩展是 C# 的 XAML 等效项`typeof`关键字。它受`TypeExtension`类，该类定义一个名为属性`TypeName`类型的`string`设置为类或结构的名称。`x:Type`标记扩展返回`System.Type`类或结构的对象。`TypeName`是的 content 属性`TypeExtension`，因此`TypeName=`时不需要`x:Type`大括号，会显示。

在 Xamarin.Forms 中，有多个属性具有类型的自变量的`Type`。示例包括`TargetType`的属性`Style`，并X:typearguments用来在泛型类中指定参数属性。但是，XAML 分析器不会执行`typeof`操作，自动和`x:Type`在这些情况下不使用标记扩展。

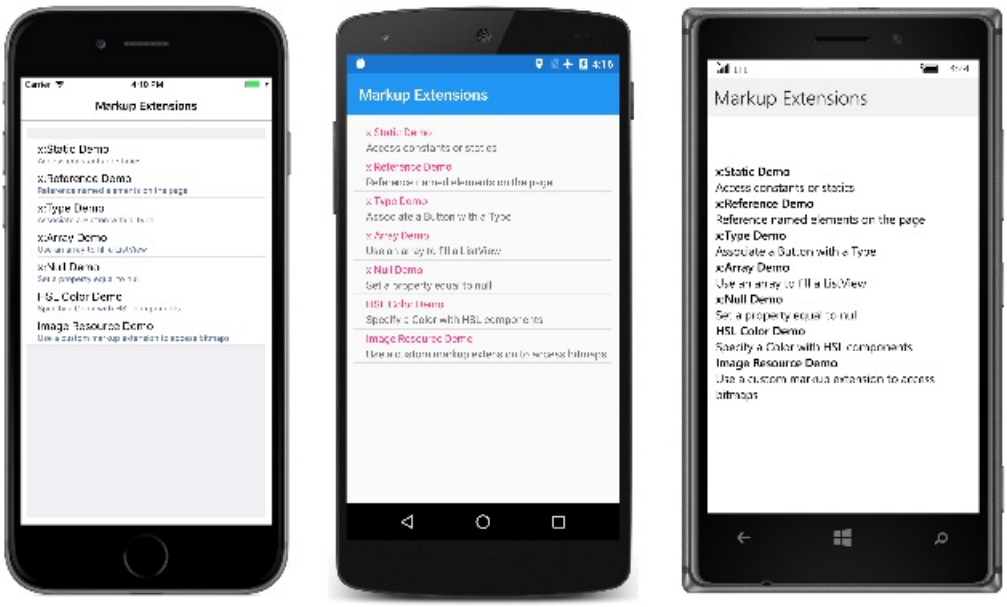
一个位置其中 `x:Type` 是要求是使用 `x:Array` 中所述的标记扩展下一节。

`x:Type` 构造一个菜单，其中每个菜单项对应于特定类型的对象时，标记扩展也是很有用。你可以将关联 `Type` 对象与每个菜单项，，然后选择菜单项时实例化对象。

这是如何在导航菜单 `MainPage` 中标记扩展编程的工作原理。**MainPage.xaml**文件包含 `TableView` 与每个 `TextCell` 对应于程序中的特定页：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MarkupExtensions"
             x:Class="MarkupExtensions.MainPage"
             Title="Markup Extensions"
             Padding="10">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection>
                <TextCell Text="x:Static Demo"
                          Detail="Access constants or statics"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:StaticDemoPage}" />

                <TextCell Text="x:Reference Demo"
                          Detail="Reference named elements on the page"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:ReferenceDemoPage}" />

                <TextCell Text="x:Type Demo"
                          Detail="Associate a Button with a Type"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:TypeDemoPage}" />

                <TextCell Text="x:Array Demo"
                          Detail="Use an array to fill a ListView"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:ArrayDemoPage}" />

                ...

        </TableRoot>
    </TableView>
</ContentPage>
```

下面是在中打开主页标记扩展：



每个 `CommandParameter` 属性设置为 `x:Type` 引用了一个其他页面的标记扩展。 `Command` 属性绑定到一个名为属性 `NavigateCommand` 。此属性定义在 `MainPage` 代码隐藏文件：

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(async (Type pageType) =>
        {
            Page page = (Page)Activator.CreateInstance(pageType);
            await Navigation.PushAsync(page);
        });

        BindingContext = this;
    }

    public ICommand NavigateCommand { private set; get; }
}
```

`NavigateCommand` 属性是 `Command` 对象，它实现包含类型的自变量的 execute 命令 `Type` —的值 `CommandParameter` 。该方法使用 `Activator.CreateInstance` 来实例化页，然后转到它。最后，通过设置构造函数 `BindingContext` 到其自身页，这使得 `Binding` 上 `Command` 工作。请参阅**数据绑定**一文，特别是 **Commanding** 一文，了解有关此类型的代码的更多详细信息。

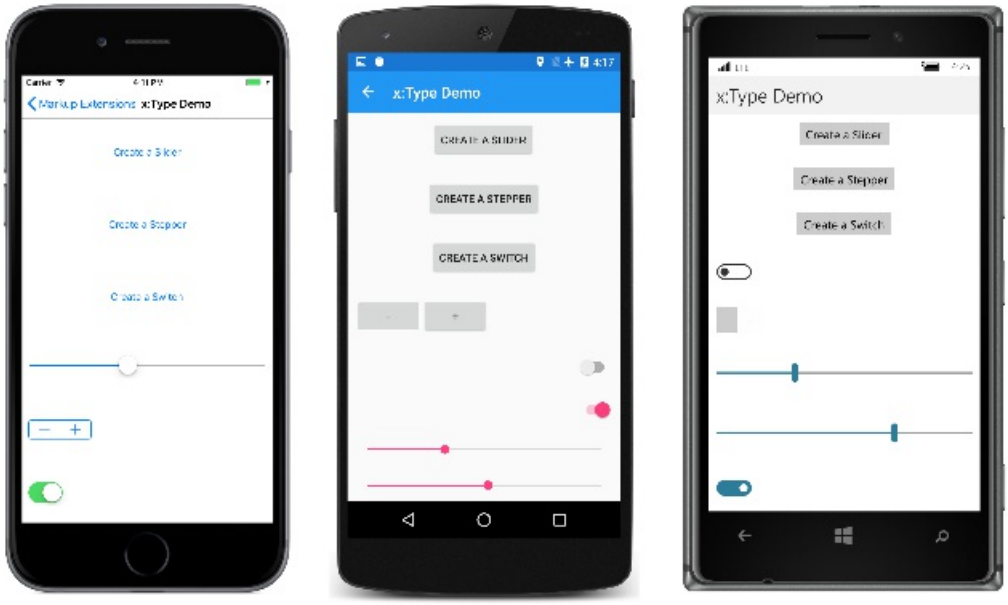
X：类型演示页使用类似的技术来实例化 Xamarin.Forms 元素，并将其添加到 `StackLayout` 。该 XAML 文件最初由三个 `Button` 元素使用其 `Command` 属性设置为 `Binding` 和 `CommandParameter` 属性设置为三个 Xamarin.Forms 视图的类型：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MarkupExtensions.TypeDemoPage"
             Title="x:Type Demo">

    <StackLayout x:Name="stackLayout"
                 Padding="10, 0">

        <Button Text="Create a Slider"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Command="{Binding CreateCommand}"
                CommandParameter="{x:Type Slider}" />

        <Button Text="Create a Stepper"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Command="{Binding CreateCommand}"
                CommandParameter="{x:Type Stepper}" />

        <Button Text="Create a Switch"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                Command="{Binding CreateCommand}"
                CommandParameter="{x:Type Switch}" />
    </StackLayout>
</ContentPage>
```

代码隐藏文件定义和初始化 `CreateCommand` 属性：

```
public partial class TypeDemoPage : ContentPage
{
    public TypeDemoPage()
    {
        InitializeComponent();

        CreateCommand = new Command<Type>((Type viewType) =>
        {
            View view = (View)Activator.CreateInstance(viewType);
            view.VerticalOptions = LayoutOptions.CenterAndExpand;
            stackLayout.Children.Add(view);
        });

        BindingContext = this;
    }

    public ICommand CreateCommand { private set; get; }
}
```

是该方法时执行 `Button` 被按下创建的自变量的新实例，设置其 `VerticalOptions` 属性，并将其添加到 `StackLayout` 。这三个 `Button` 元素然后动态创建的视图与共享页：



x:Array 标记扩展

`x:Array` 标记扩展，可在标记中定义一个数组。它受 `ArrayExtension` 类，该类定义两个属性：

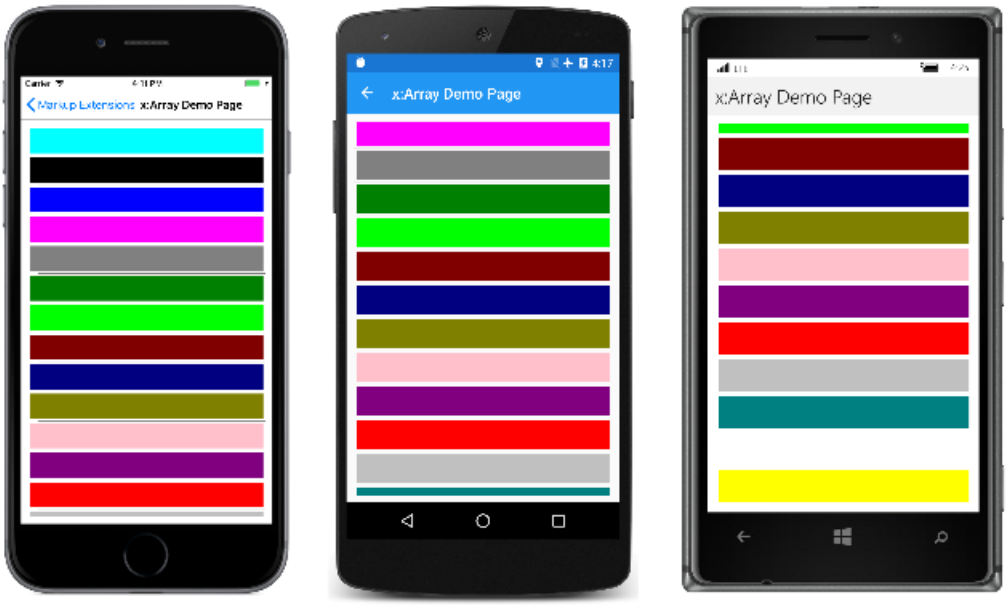
- `Type` 类型的 `Type` ，指示数组中元素的类型。
- `Items` 类型的 `IList` ，它是项目本身的集合。这是内容属性的 `ArrayExtension` 。

`x:Array` 标记扩展本身永远不会显示在大括号中。相反， `x:Array` 开始和结束标记分隔的项的列表。设置 `Type` 属性设置为 `x:Type` 标记扩展。

X:array 演示页显示了如何使用 `x:Array` 若要将项添加到 `ListView` 通过设置 `ItemsSource` 到一个数组的属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MarkupExtensions.ArrayDemoPage"
             Title="x:Array Demo Page">
    <ListView Margin="10">
        <ListView.ItemsSource>
            <x:Array Type="{x:Type Color}">
                <Color>Aqua</Color>
                <Color>Black</Color>
                <Color>Blue</Color>
                <Color>Fuchsia</Color>
                <Color>Gray</Color>
                <Color>Green</Color>
                <Color>Lime</Color>
                <Color>Maroon</Color>
                <Color>Navy</Color>
                <Color>Olive</Color>
                <Color>Pink</Color>
                <Color>Purple</Color>
                <Color>Red</Color>
                <Color>Silver</Color>
                <Color>Teal</Color>
                <Color>White</Color>
                <Color>Yellow</Color>
            </x:Array>
        </ListView.ItemsSource>

        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <BoxView Color="{Binding}"
                             Margin="3" />
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

`ViewCell`创建一个简单`BoxView`为每个颜色条目：



有几种方法来指定对单个`Color`此数组中的项。可以使用`x:Static`标记扩展：

```
<x:Static Member="Color.Blue" />
```

或者，可以使用 `StaticResource` 从资源字典中检索一种颜色：

```
<StaticResource Key="myColor" />
```

这篇文章后，你将看到还会创建一个新的颜色值的自定义 XAML 标记扩展：

```
<local:HslColor H="0.5" S="1.0" L="0.5" />
```

在定义字符串或数字等常见类型的数组时，使用标记中列出**传递构造函数自变量**文章来分隔的值。

x:Null 标记扩展

`x:Null` 标记扩展受 `NullExtension` 类。它没有属性，并且只需 XAML 等效于 C# `null` 关键字。

`x:Null` 标记扩展是很少需要和很少使用，但如果找到它需要，您会很高兴它存在。

X:null 演示页说明了一种情况时 `x:Null` 可能非常方便。假设您定义一种隐式 `Style` 有关 `Label` 其中包括 `Setter`，用于设置 `FontFamily` 属性设置为依赖于平台的系列名称：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="MarkupExtensions.NullDemoPage"
             Title="x:Null Demo">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="48" />
                <Setter Property="FontFamily">
                    <Setter.Value>
                        <OnPlatform x:TypeArguments="x:String">
                            <On Platform="iOS" Value="Times New Roman" />
                            <On Platform="Android" Value="serif" />
                            <On Platform="UWP" Value="Times New Roman" />
                        </OnPlatform>
                    </Setter.Value>
                </Setter>
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ContentPage.Content>
        <StackLayout Padding="10, 0">
            <Label Text="Text 1" />
            <Label Text="Text 2" />

            <Label Text="Text 3"
                   FontFamily="{x:Null}" />

            <Label Text="Text 4" />
            <Label Text="Text 5" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

然后你将发现之一 `Label` 元素，所需的所有属性设置的隐式 `Style` 除 `FontFamily`，想要将默认值。您可以定义另一个 `Style` 为此目的，但更简单的方法是只需设置 `FontFamily` 属性的特定 `Label` 到 `x:Null`，如下所示在中心 `Label`。

下面是三个平台上运行的程序：

请注意，该四个的 `Label` 元素具有衬线字体，但在中心 `Label` 具有默认的 sans-serif 字体。

OnPlatform 标记扩展

`OnPlatform` 标记扩展允许您自定义根据每个平台的 UI 外观。它提供了相同的功能 `OnPlatform` 并 `On` 类，但具有更简洁表示形式。

`OnPlatform` 标记扩展受 `OnPlatformExtension` 类，该类定义以下属性：

- `Default` 类型的 `object`，设置为默认值应用于表示平台的属性。
- `Android` 类型的 `object`，设置一个值为要应用在 Android 上。
- `GTK` 类型的 `object`，设置一个值为要应用于 GTK 平台。
- `iOS` 类型的 `object`，设置为值用于在 iOS 上应用。
- `macOS` 类型的 `object`，设置为值用于在 macOS 上应用。
- `Tizen` 类型的 `object`，设置一个值为要应用于 Tizen 平台。
- `UWP` 类型的 `object`，设置一个值为通用 Windows 平台上应用。
- `WPF` 类型的 `object`，设置一个值为要应用于 Windows Presentation Foundation 平台。
- `Converter` 类型的 `IValueConverter`，设置为 `IValueConverter` 实现。
- `ConverterParameter` 类型的 `object`，设置一个值为要传递给 `IValueConverter` 实现。

> **NOTE**
> 此 XAML 分析器允许 `OnPlatformExtension` 类以缩写为 `OnPlatform`。

`Default` 属性是 content 属性 `OnPlatformExtension`。因此，对于使用大括号表示 XAML 标记表达式，则可以消除 `Default=` 表达式的一部分提供，它是第一个参数。

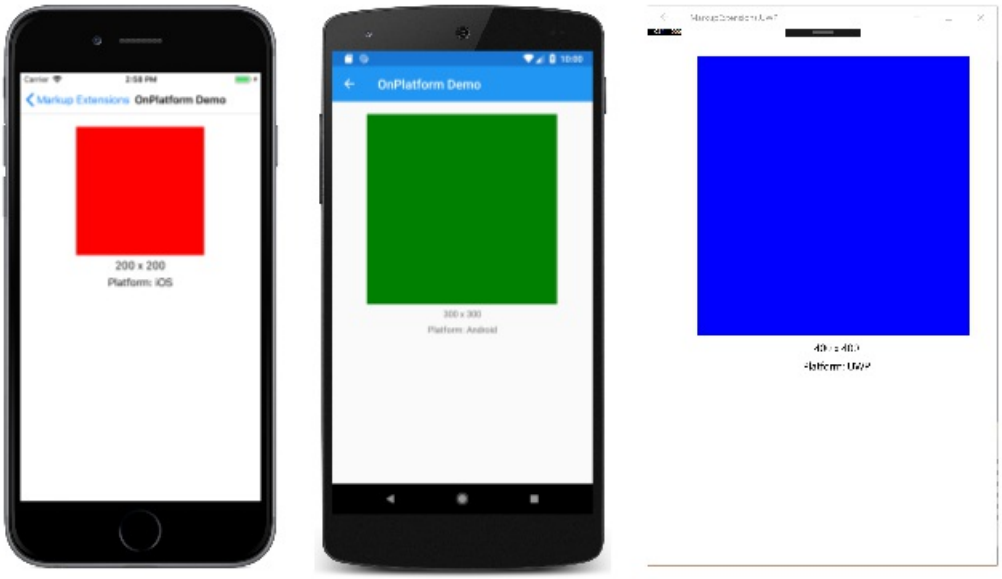
> **IMPORTANT**
> XAML 分析器要求正确类型的值将会提供给属性使用 `OnPlatform` 标记扩展。如果类型转换，则有必要，`OnPlatform` 标记扩展将尝试执行它使用 Xamarin.Forms 提供的默认值转换器。但是，有不能通过默认转换器，在这些情况下执行某些类型转换 `Converter` 属性应设置为 `IValueConverter` 实现。

OnPlatform 演示页显示了如何使用 `OnPlatform` 标记扩展：

```
<BoxView Color="{OnPlatform Yellow, iOS=Red, Android=Green, UWP=Blue}"
         WidthRequest="{OnPlatform 250, iOS=200, Android=300, UWP=400}"
         HeightRequest="{OnPlatform 250, iOS=200, Android=300, UWP=400}"
         HorizontalOptions="Center" />
```

在此示例中，所有这三个 `OnPlatform` 表达式使用的缩写的形式 `OnPlatformExtension` 类名。这三个 `OnPlatform` 标记扩展集 `Color`，`WidthRequest`，并且 `HeightRequest` 属性 `BoxView` 为 iOS、Android 和 UWP 上不同的值。标记扩展还为未指定，同时消除了在平台上的这些属性提供默认值 `Default=` 表达式的一部分。请注意，由逗号分隔的标记扩展属性的设置。

下面是在所有三个平台上运行的程序：



OnIdiom 标记扩展

`OnIdiom` 标记扩展，可自定义 UI 外观基于的设备运行应用程序的惯用语法。它受 `OnIdiomExtension` 类，该类定义以下属性：

- `Default` 类型的 `object`，设置为默认值应用于设备的惯用语言表示的属性。
- `Phone` 类型的 `object`，设置为值用于在手机上应用。
- `Tablet` 类型的 `object`，若要在平板电脑上应用的值设置。
- `Desktop` 类型的 `object`，设置为值用于在桌面平台上应用。
- `TV` 类型的 `object`，设置一个值为要应用于电视平台。
- `Watch` 类型的 `object`，设置为值应用于监视平台。
- `Converter` 类型的 `IValueConverter`，设置为 `IValueConverter` 实现。
- `ConverterParameter` 类型的 `object`，设置一个值为要传递给 `IValueConverter` 实现。

> **NOTE**
> 此 XAML 分析器允许 `OnIdiomExtension` 类以缩写为 `OnIdiom`。

`Default` 属性是 content 属性 `OnIdiomExtension`。因此，对于使用大括号表示 XAML 标记表达式，则可以消除 `Default=` 表达式的一部分提供，它是第一个参数。

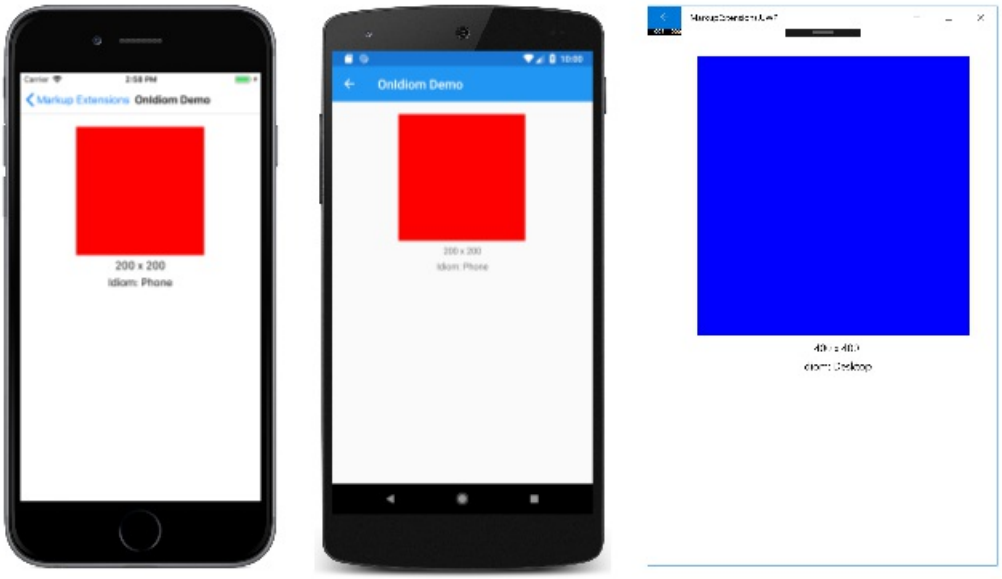
> **IMPORTANT**
>
> XAML 分析器要求正确类型的值将会提供给属性使用 `OnIdiom` 标记扩展。如果类型转换，则有必要，`OnIdiom` 标记扩展将尝试执行它使用 Xamarin.Forms 提供的默认值转换器。但是，有不能通过默认转换器，在这些情况下执行某些类型转换 `Converter` 属性应设置为 `IValueConverter` 实现。

OnIdiom 演示页显示了如何使用 `OnIdiom` 标记扩展：

```
<BoxView Color="{OnIdiom Yellow, Phone=Red, Tablet=Green, Desktop=Blue}"
         WidthRequest="{OnIdiom 100, Phone=200, Tablet=300, Desktop=400}"
         HeightRequest="{OnIdiom 100, Phone=200, Tablet=300, Desktop=400}"
         HorizontalOptions="Center" />
```

在此示例中，所有这三个 `OnIdiom` 表达式使用的缩写的形式 `OnIdiomExtension` 类名。这三个 `OnIdiom` 标记扩展集 `Color`，`WidthRequest`，并且 `HeightRequest` 属性 `BoxView` 为手机、平板电脑和桌面的惯用语言在不同的值。标记扩展还提供默认值为这些属性在未指定，同时消除习惯用语 `Default=` 表达式的一部分。请注意，由逗号分隔的标记扩展属性的设置。

下面是在所有三个平台上运行的程序：



定义你自己的标记扩展

如果遇到过需要在 Xamarin.Forms 中不可用的 XAML 标记扩展，则可以创建您自己。

相关链接

- 标记扩展（示例）
- 从 Xamarin.Forms 书籍的 XAML 标记扩展一章
- 资源字典
- 动态样式
- 数据绑定

创建 XAML 标记扩展

2018/11/13 • • Edit Online

在以编程方式的级别中，XAML 标记扩展是实现的类 `IMarkupExtension` 或 `IMarkupExtension<T>` 接口。你可以浏览在如下所述的标准标记扩展的源代码 **MarkupExtensions** directory的 Xamarin.Forms GitHub 存储库。

还有可能通过派生自定义您自己自定义的 XAML 标记扩展 `IMarkupExtension` 或 `IMarkupExtension<T>` 。如果标记扩展获取特定类型的值，使用泛型窗体。这是 Xamarin.Forms 标记扩展的多个用例：

- `TypeExtension` 派生自 `IMarkupExtension<Type>`
- `ArrayExtension` 派生自 `IMarkupExtension<Array>`
- `DynamicResourceExtension` 派生自 `IMarkupExtension<DynamicResource>`
- `BindingExtension` 派生自 `IMarkupExtension<BindingBase>`
- `ConstraintExpression` 派生自 `IMarkupExtension<Constraint>`

这两个 `IMarkupExtension` 接口定义每个，只有一个方法名为 `ProvideValue`：

```
public interface IMarkupExtension
{
    object ProvideValue(IServiceProvider serviceProvider);
}

public interface IMarkupExtension<out T> : IMarkupExtension
{
    new T ProvideValue(IServiceProvider serviceProvider);
}
```

由于 `IMarkupExtension<T>` 派生自 `IMarkupExtension` 并包含 `new` 上的关键字 `ProvideValue` ，它同时包含 `ProvideValue` 方法。

通常，XAML 标记扩展的返回值定义参与的属性。(明显的例外是 `NullExtension` ，在其中 `ProvideValue` 只需返回 `null` 。) `ProvideValue` 方法具有一个参数类型的 `IServiceProvider` 将稍后在本文中讨论的。

用于指定颜色标记扩展

下面的 XAML 标记扩展使你可以构造 `Color` 值使用色调、饱和度和亮度的组件。它定义的颜色，包括 alpha 分量，将初始化为 1 的四个组件的四个属性。类派生自 `IMarkupExtension<Color>` 以指示 `Color` 返回值：

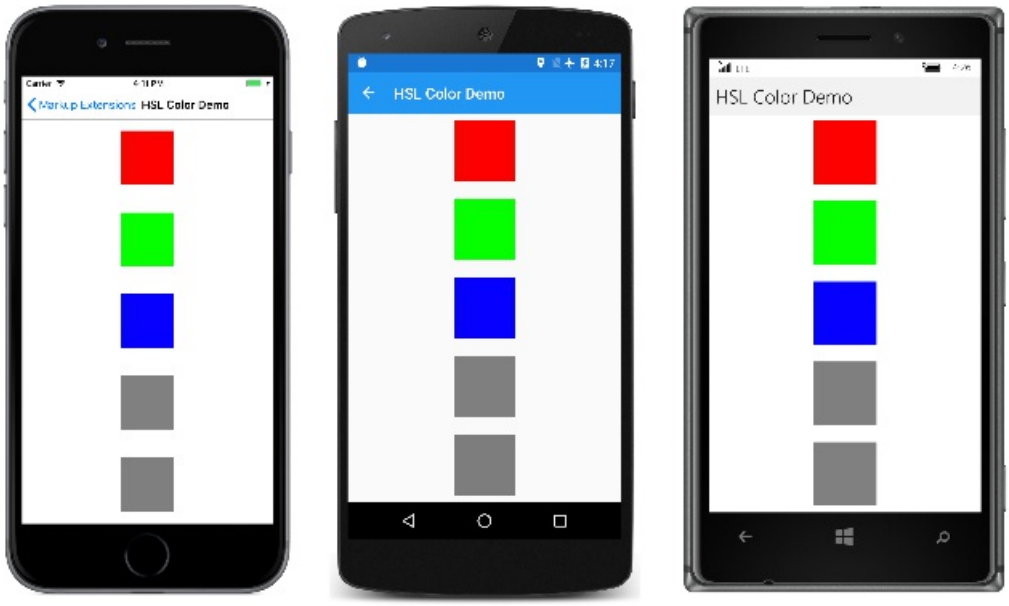
```
public class HslColorExtension : IMarkupExtension<Color>
{
    public double H { set; get; }

    public double S { set; get; }

    public double L { set; get; }

    public double A { set; get; } = 1.0;

    public Color ProvideValue(IServiceProvider serviceProvider)
    {
        return Color.FromHsla(H, S, L, A);
    }

    object IMarkupExtension.ProvideValue(IServiceProvider serviceProvider)
    {
        return (this as IMarkupExtension<Color>).ProvideValue(serviceProvider);
    }
}
```

因为`IMarkupExtension<T>`派生自`IMarkupExtension`，类必须包含两个`ProvideValue`方法，即：返回`Color`，另一个返回`object`，但第二种方法只需调用第一种方法。

HSL 颜色演示页上显示不同的方式`HslColorExtension`可以在指定的颜色的 XAML 文件中出现`BoxView`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MarkupExtensions"
             x:Class="MarkupExtensions.HslColorDemoPage"
             Title="HSL Color Demo">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="WidthRequest" Value="80" />
                <Setter Property="HeightRequest" Value="80" />
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <BoxView>
            <BoxView.Color>
                <local:HslColorExtension H="0" S="1" L="0.5" A="1" />
            </BoxView.Color>
        </BoxView>

        <BoxView>
            <BoxView.Color>
                <local:HslColor H="0.33" S="1" L="0.5" />
            </BoxView.Color>
        </BoxView>

        <BoxView Color="{local:HslColorExtension H=0.67, S=1, L=0.5}" />

        <BoxView Color="{local:HslColor H=0, S=0, L=0.5}" />

        <BoxView Color="{local:HslColor A=0.5}" />
    </StackLayout>
</ContentPage>
```

请注意，当 `HslColorExtension` 是一个 XML 标记，四个属性设置为属性，但它出现时大括号之间，由不带引号的逗号分隔的四个属性。默认值为 `H`， `S`，并 `L` 为 0 和的默认值 `A` 为 1，因此可以省略这些属性，如果你希望它们设置为默认值。最后一个示例演示其中亮度为 0，这通常会导致黑色，但 alpha 通道是 0.5，因此它是半双工透明的将显示一个示例与白色背景页的灰色：



用于访问位图标记扩展

参数 `ProvideValue` 是一个对象，实现 `IServiceProvider` 接口，在.NET 中定义 `System` 命名空间。此接口具有一个成员，一个名为方法 `GetService` 与 `Type` 参数。

`ImageResourceExtension` 如下所示的类演示的可能用途之一 `IServiceProvider` 并 `GetService` 若要获取 `IXmlLineInfoProvider` 对象可提供行和字符，该值指示在其中检测到特定错误的信息。在这种情况下，会引发的异常时 `Source` 未设置属性：

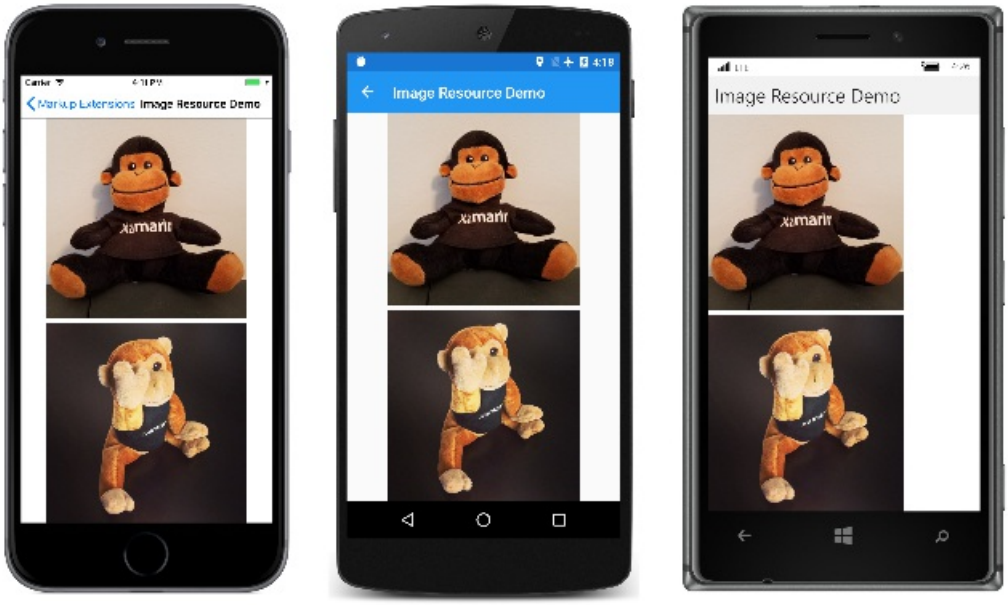
```
[ContentProperty("Source")]
class ImageResourceExtension : IMarkupExtension<ImageSource>
{
    public string Source { set; get; }

    public ImageSource ProvideValue(IServiceProvider serviceProvider)
    {
        if (String.IsNullOrEmpty(Source))
        {
            IXmlLineInfoProvider lineInfoProvider = serviceProvider.GetService(typeof(IXmlLineInfoProvider))
as IXmlLineInfoProvider;
            IXmlLineInfo lineInfo = (lineInfoProvider != null) ? lineInfoProvider.XmlLineInfo : new
XmlLineInfo();
            throw new XamlParseException("ImageResourceExtension requires Source property to be set",
lineInfo);
        }

        string assemblyName = GetType().GetTypeInfo().Assembly.GetName().Name;

        return ImageSource.FromResource(assemblyName + "." + Source);
    }

    object IMarkupExtension.ProvideValue(IServiceProvider serviceProvider)
    {
        return (this as IMarkupExtension<ImageSource>).ProvideValue(serviceProvider);
    }
}
```

`ImageResourceExtension` 当需要访问作为嵌入资源的.NET Standard 库项目中存储的图像文件的 XAML 文件时非常有用。它使用 `Source` 属性来调用静态 `ImageSource.FromResource` 方法。此方法要求完全限定资源名称，其中包含的程序集名称、文件夹名称和用句点分隔的文件名。`ImageResourceExtension` 不需要的程序集名称部分因为获取使用反射的程序集名称，并将前面添加到 `Source` 属性。无论如何，`ImageSource.FromResource` 必须从包含位图，这意味着除非图像还在该库中，此 XAML 资源扩展不能为外部库的一部分的程序集调用。(请参阅**嵌入图像**访问位图作为嵌入资源存储的详细信息的文章。)

尽管 `ImageResourceExtension` 需要 `Source` 属性设置，`Source` 属性指示在属性中为类的内容属性。这意味着，`Source=` 可以省略大括号中的表达式的一部分。在中图像资源演示页上，`Image` 元素提取使用文件夹名称和文件名用句点分隔的两个映像：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MarkupExtensions"
             x:Class="MarkupExtensions.ImageResourceDemoPage"
             Title="Image Resource Demo">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Image Source="{local:ImageResource Images.SeatedMonkey.jpg}"
               Grid.Row="0" />

        <Image Source="{local:ImageResource Images.FacePalm.jpg}"
               Grid.Row="1" />

    </Grid>
</ContentPage>
```

下面是在所有三个平台上运行的程序：



服务提供商

通过使用 `IServiceProvider` 自变量 `ProvideValue`，XAML 标记扩展有权访问在其中使用这些 XAML 文件有关的有用信息。但若要使用 `IServiceProvider` 参数成功，您需要知道什么类型的服务是在特定上下文中可用。若要了解此功能的最佳方法是通过研究现有 XAML 标记扩展中的源代码 **MarkupExtensions文件夹**在 GitHub 上的 Xamarin.Forms 存储库中。请注意某些类型的服务是 Xamarin.Forms 的内部。

在某些 XAML 标记扩展中，此服务可能会很有用：

```
 IProvideValueTarget provideValueTarget = serviceProvider.GetService(typeof(IProvideValueTarget)) as
IProvideValueTarget;
```

`IProvideValueTarget` 接口定义两个属性 `TargetObject` 和 `TargetProperty`。在获得此信息 `ImageResourceExtension` 类，`TargetObject` 是 `Image` 并 `TargetProperty` 是 `BindableProperty` 对象 `Source` 属性 `Image`。这是在其设置 XAML 标记扩展的属性。

`GetService` 使用的自变量调用 `typeof(IProvideValueTarget)` 实际返回的类型的对象 `SimpleValueTargetProvider`，其定义中 `Xamarin.Forms.Xaml.Internals` 命名空间。如果强制转换的返回值 `GetService` 为该类型，你可以访问 `ParentObjects` 属性，它是一个数组，包含 `Image` 元素，`Grid` 父项，并 `ImageResourceDemoPage` 的父级 `Grid`。

结束语

XAML 标记扩展通过扩展从各种源中设置属性的功能，在 XAML 中扮演重要角色。此外，如果现有的 XAML 标记扩展未提供所需内容，您还可以编写您自己。

相关链接

- 标记扩展（示例）
- 从 Xamarin.Forms 书籍的 XAML 标记扩展一章

Xamarin.Forms 中的 XAML 字段修饰符

2018/6/20 • • Edit Online

`x:FieldModifier` *命名空间属性指定的生成字段的命名 XAML 元素的访问级别。*

概述

该属性的有效值为:

- `Public` – 指定的 XAML 元素所生成的字段是 `public`。
- `NotPublic` – 指定的 XAML 元素所生成的字段是 `internal` 对程序集。

如果未设置属性的值，元素生成的字段将 `private`。

必须满足以下条件的 `x:FieldModifier` 要处理的属性:

- XAML 中的顶级元素必须是有效 `x:Class`。
- 当前的 XAML 元素具有 `x:Name` 指定。

下面的 XAML 演示设置该属性的示例:

```
<Label x:Name="privateLabel" />
<Label x:Name="internalLabel" x:FieldModifier="NotPublic" />
<Label x:Name="publicLabel" x:FieldModifier="Public" />
```

NOTE

`x:FieldModifier` 属性不能用于指定 XAML 类的访问级别。

在 XAML 中传递自变量

2018/11/13 • • Edit Online

本文演示如何使用可用于将参数传递到非默认构造函数，以调用工厂方法，并指定泛型参数的类型的 XAML 属性。

概述

通常是使用构造函数需要参数，或通过调用静态创建方法的对象的实例化所必需的。这可以通过在 XAML 中使用 `x:Arguments` 和 `x:FactoryMethod` 属性：

- `x:Arguments` 特性用于指定非默认构造函数或工厂方法对象声明的构造函数自变量。有关详细信息，请参阅传递构造函数参数。
- `x:FactoryMethod` 特性用于指定可用于初始化对象的工厂方法。有关详细信息，请参阅调用工厂方法。

此外，`x:TypeArguments` 属性可用于指定泛型类型的构造函数的泛型类型参数。有关详细信息，请参阅指定泛型类型参数。

传递构造函数自变量

参数可以传递到非默认构造函数使用 `x:Arguments` 属性。表示自变量的类型的 XML 元素中，每个构造函数自变量必须进行分隔。Xamarin.Forms 的基本类型支持以下元素：

- `x:Object`
- `x:Boolean`
- `x:Byte`
- `x:Int16`
- `x:Int32`
- `x:Int64`
- `x:Single`
- `x:Double`
- `x:Decimal`
- `x:Char`
- `x:String`
- `x:TimeSpan`
- `x:Array`
- `x:DateTime`

下面的代码示例演示了如何使用 `x:Arguments` 有三个特性 `Color` 构造函数：

```
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color>
      <x:Arguments>
        <x:Double>0.9</x:Double>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color>
      <x:Arguments>
        <x:Double>0.25</x:Double>
        <x:Double>0.5</x:Double>
        <x:Double>0.75</x:Double>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color>
      <x:Arguments>
        <x:Double>0.8</x:Double>
        <x:Double>0.5</x:Double>
        <x:Double>0.2</x:Double>
        <x:Double>0.5</x:Double>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
```

内的元素数 `x:Arguments` 标记，以及这些元素的类型必须与之一匹配 `Color` 构造函数。`Color` 构造函数使用单个参数需要灰度值从 0（黑色）设置为 1（白色）。`Color` 构造函数带有三个参数需要红色、绿色和蓝色值范围从 0 到 1。`Color` 构造函数带四个参数将作为第四个参数添加 alpha 通道。

以下屏幕截图显示了每个调用的结果 `Color` 构造函数使用指定的参数值：

调用工厂方法

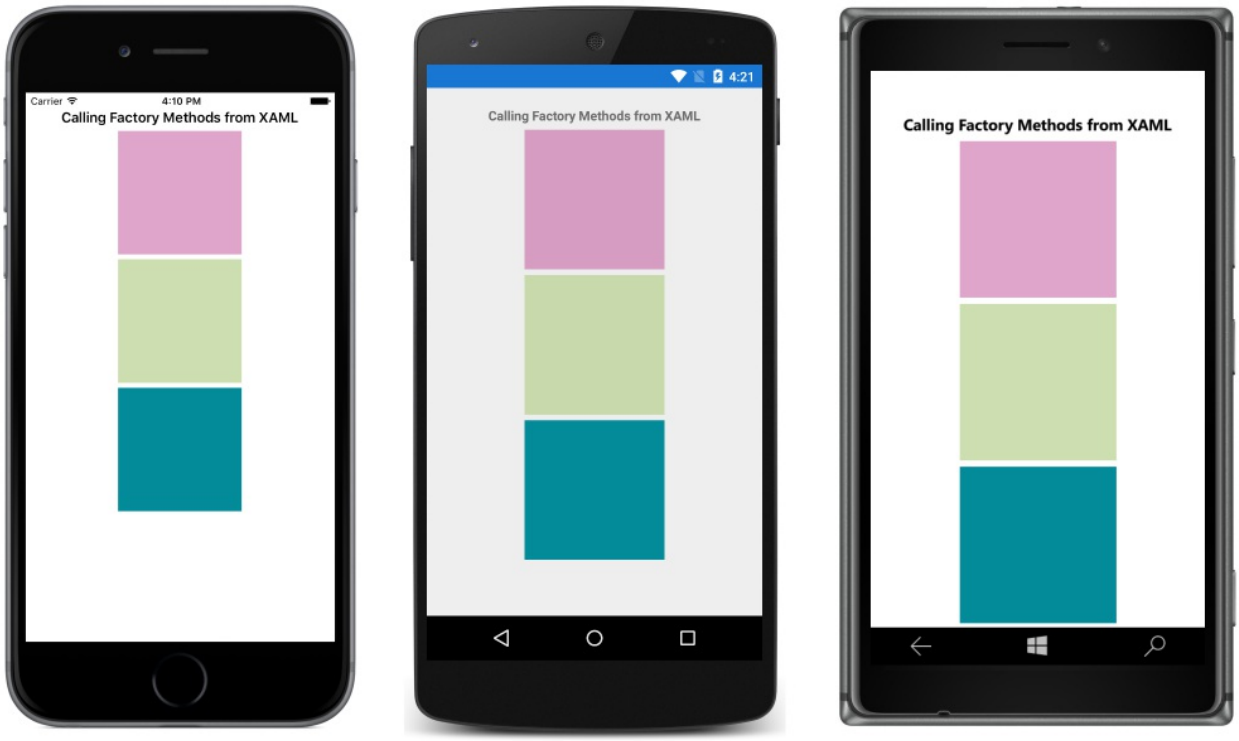
可以在 XAML 中通过指定方法的调用工厂方法采用 `x:FactoryMethod` 属性，并使用其自变量 `x:Arguments` 属性。工厂方法是 `public static` 返回对象或值类或结构，它定义的方法的类型相同的方法。

`Color` 结构定义的工厂方法数和下面的代码示例演示如何调用这三种页脚：

```
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromRgba">
      <x:Arguments>
        <x:Int32>192</x:Int32>
        <x:Int32>75</x:Int32>
        <x:Int32>150</x:Int32>
        <x:Int32>128</x:Int32>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromHsla">
      <x:Arguments>
        <x:Double>0.23</x:Double>
        <x:Double>0.42</x:Double>
        <x:Double>0.69</x:Double>
        <x:Double>0.7</x:Double>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
<BoxView HeightRequest="150" WidthRequest="150" HorizontalOptions="Center">
  <BoxView.Color>
    <Color x:FactoryMethod="FromHex">
      <x:Arguments>
        <x:String>#FF048B9A</x:String>
      </x:Arguments>
    </Color>
  </BoxView.Color>
</BoxView>
```

内的元素数 `x:Arguments` 标记，以及这些元素的类型必须匹配要调用的工厂方法的参数。`FromRgba` 工厂方法需要四个 `Int32` 表示红色、绿色、蓝色和 alpha 值分别范围从 0 到 255 之间的参数。`FromHsla` 工厂方法需要四个 `Double` 表示色调、饱和度、亮度和 alpha 值，分别范围从 0 到 1 的参数。`FromHex` 工厂方法需要 `String` 表示十六进制 (A) 代表 RGB 颜色。

以下屏幕截图显示了每个调用的结果 `Color` 工厂方法与指定的参数值：



指定泛型类型参数

可以使用指定泛型类型参数的泛型类型的构造函数 `x:TypeArguments` 属性，如下面的代码示例中所示：

```
<ContentPage ...>
  <StackLayout>
    <StackLayout.Margin>
      <OnPlatform x:TypeArguments="Thickness">
        <On Platform="iOS" Value="0,20,0,0" />
        <On Platform="Android" Value="5, 10" />
        <On Platform="UWP" Value="10" />
      </OnPlatform>
    </StackLayout.Margin>
  </StackLayout>
</ContentPage>
```

`OnPlatform` 类是一个泛型类，必须使用实例化 `x:TypeArguments` 匹配目标类型的属性。在中 `On` 类， `Platform` 属性可以接受单个 `string` 值或以逗号分隔的多个 `string` 值。在此示例中， `StackLayout.Margin` 属性设置为特定于平台的 `Thickness` 。

总结

本文演示了使用的 XAML 特性，可用于将参数传递到非默认构造函数，以调用工厂方法，并指定泛型参数的类型。

相关链接

- XAML 命名空间
- 传递构造函数自变量（示例）
- 调用工厂方法（示例）

可绑定属性

2018/11/13 • • Edit Online

在 Xamarin.Forms 中，公共语言运行时 (CLR) 属性的功能扩展可绑定属性。可绑定属性是属性的特殊类型，其中 Xamarin.Forms 属性系统跟踪属性的值。本文介绍了可绑定属性，并演示如何创建并使用它们。

概述

可绑定属性扩展通过将数据备份具有的属性的 CLR 属性功能 `BindableProperty` 类型，而不是支持使用字段属性。可绑定属性的用途是提供支持数据绑定、样式、模板的属性系统，并通过父-子关系的值设置。此外，可绑定属性可以提供默认值，验证属性值和监视属性更改回调。

属性应作为可绑定属性，以支持一个或多个以下功能：

- 作为一个有效*目标*数据绑定的属性。
- 通过将属性设置样式。
- 提供默认属性值不同于属性的类型的默认值。
- 正在验证属性的值。
- 监视属性更改。

Xamarin.Forms 的可绑定属性的示例包括 `Label.Text`，`Button.BorderRadius`，以及 `StackLayout.Orientation`。每个可绑定属性都有一个相应 `public static readonly` 类型的属性 `BindableProperty` 相同的类上公开，可绑定属性的标识符。例如，相应的可绑定属性标识符 `Label.Text` 属性是 `Label.TextProperty`。

创建和使用可绑定属性

创建可绑定属性的过程如下所示：

1. 创建 `BindableProperty` 具有的一个实例 `BindableProperty.Create` 方法重载。
2. 定义属性的访问器 `BindableProperty` 实例。

请注意，所有 `BindableProperty` 必须在 UI 线程上创建实例。这意味着在 UI 线程运行的代码可以获取或设置可绑定属性的值。但是，`BindableProperty` 实例可以从其他线程封送到 UI 线程中访问 `Device.BeginInvokeOnMainThread` 方法。

创建属性

若要创建 `BindableProperty` 实例，包含的类必须派生自 `BindableObject` 类。但是，`BindableObject` 类是高类层次结构中，因此大多数类用于用户界面功能支持可绑定属性。

可以通过声明创建可绑定的属性 `public static readonly` 类型的属性 `BindableProperty`。可绑定属性应设置为返回值之一 `BindableProperty.Create` 方法重载。声明应为体内 `BindableObject` 派生的类，但之外的任何成员定义。

创建时，则必须指定至少一个标识符 `BindableProperty`，以及以下参数：

- 名称 `BindableProperty`。
- 属性的类型。
- 所属对象的类型。
- 属性的默认值。这可确保该属性总是返回特定的默认值，未设置，并且它可以不同于属性的类型的默认值。默认值将是还原何时 `ClearValue` 可绑定属性上调用方法。

下面的代码演示具有标识符和四个必需参数的值的可绑定属性的示例：

```
public static readonly BindableProperty EventNameProperty =
  BindableProperty.Create ("EventName", typeof(string), typeof(EventToCommandBehavior), null);
```

这将创建 `BindableProperty` 实例名为 `EventName`，类型的 `string`。该属性归 `EventToCommandBehavior` 类，并具有默认值为 `null`。可绑定属性的命名约定是可绑定的属性标识符中指定属性名称必须匹配 `Create` 方法，使用"属性"追加到它。因此，在上面的示例中，可绑定的属性标识符是 `EventNameProperty`。

（可选）在创建时 `BindableProperty` 实例，下面可以指定参数：

- 绑定模式。这用于指定将属性值更改将传播的方向。在默认绑定模式中，更改将传播从*源*到*目标*。
- 设置属性值时，将调用一个验证委托。有关详细信息，请参阅验证回叫。
- 属性更改将属性值已更改时调用的委托。有关详细信息，请参阅检测属性更改。
- 一个属性，更改将属性值将更改时调用的委托。此委托具有与属性更改委托相同的签名。
- 属性值已更改时，将调用一个强制值委托。有关详细信息，请参阅强制值回叫。
- 一个 `Func`，用来初始化默认属性值。有关详细信息，请参阅使用 Func 创建默认值。

创建访问器

属性访问器需要使用属性语法来访问可绑定的属性。`Get` 访问器应返回相应的可绑定属性中包含的值。这可以通过调用来实现 `GetValue` 方法，传入要获取的值的可绑定的属性标识符，然后将结果转换为所需的类型。`Set` 访问器应设置相应的可绑定属性的值。这可以通过调用来实现 `SetValue` 方法，传入在其上设置值和要设置的值的可绑定的属性标识符。

下面的代码示例演示的访问器 `EventName` 可绑定的属性：

```
public string EventName {
  get { return (string)GetValue (EventNameProperty); }
  set { SetValue (EventNameProperty, value); }
}
```

使用可绑定属性

一旦创建可绑定属性后，可以从 XAML 或代码使用它。在 XAML，这被通过使用命名空间声明，该值指示 CLR 命名空间名称和（可选）程序集名称声明具有前缀的命名空间。有关详细信息，请参阅XAML 命名空间。

下面的代码示例演示了包含所引用的自定义类型的应用程序代码在同一程序集中定义的可绑定属性的自定义类型的 XAML 命名空间：

```
<ContentPage ... xmlns:local="clr-namespace:EventToCommandBehavior" ...>
  ...
</ContentPage>
```

设置时使用命名空间声明 `EventName` 可绑定属性，如下面的 XAML 代码示例所示：

```
<ListView ...>
  <ListView.Behaviors>
    <local:EventToCommandBehavior EventName="ItemSelected" ... />
  </ListView.Behaviors>
</ListView>
```

以下代码示例显示相应的 C# 代码：

```
var listView = new ListView ();
listView.Behaviors.Add (new EventToCommandBehavior {
  EventName = "ItemSelected",
  ...
});
```

高级的方案

创建时 `BindableProperty` 实例时，有多种可设为启用高级可绑定属性方案的可选参数。本部分探讨这些方案。

检测属性更改

一个 `static` 属性更改回调方法可以通过指定注册与可绑定属性 `propertyChanged` 参数 `BindableProperty.Create` 方法。可绑定属性的值更改时，将调用指定的回调方法。

下面的代码示例演示如何 `EventName` 可绑定属性寄存器 `OnEventNameChanged` 作为属性更改回调方法的方法：

```
public static readonly BindableProperty EventNameProperty =
  BindableProperty.Create (
    "EventName", typeof(string), typeof(EventToCommandBehavior), null, propertyChanged: OnEventNameChanged);
...

static void OnEventNameChanged (BindableObject bindable, object oldValue, object newValue)
{
  // Property changed implementation goes here
}
```

在属性更改回调方法中， `BindableObject` 参数用于表示一项更改和两个值所属的类的实例已报告 `object` 参数表示的新旧值可绑定属性。

验证回叫

一个 `static` 验证回叫方法可通过指定注册到可绑定的属性 `validateValue` 参数 `BindableProperty.Create` 方法。设置可绑定属性的值时，将调用指定的回调方法。

下面的代码示例演示如何 `Angle` 可绑定属性寄存器 `IsValidValue` 作为验证回调方法的方法：

```
public static readonly BindableProperty AngleProperty =
  BindableProperty.Create ("Angle", typeof(double), typeof(HomePage), 0.0, validateValue: IsValidValue);
...

static bool IsValidValue (BindableObject view, object value)
{
  double result;
  bool isDouble = double.TryParse (value.ToString (), out result);
  return (result >= 0 && result <= 360);
}
```

验证回叫提供了一个值，并应返回 `true` 的值是否有效的属性，否则 `false` 。如果验证回调返回，将会引发异常 `false` ，应由开发人员。可绑定属性设置时，验证回叫方法的典型用法约束整数或双精度型的值。例如， `IsValidValue` 方法检查属性值是 `double` 0 到 360 之间的范围内。

强制值回叫

一个 `static` 强制值回叫方法可通过指定注册到可绑定的属性 `coerceValue` 参数 `BindableProperty.Create` 方法。可绑定属性的值更改时，将调用指定的回调方法。

强制的值回叫用于强制重新计算的可绑定属性的属性的值更改时。例如，可以使用强制值回叫，以确保一个可绑定属性的值不大于另一个可绑定属性的值。

下面的代码示例演示如何 `Angle` 可绑定属性寄存器 `CoerceAngle` 方法作为强制值回叫方法：

```
public static readonly BindableProperty AngleProperty = BindableProperty.Create (
  "Angle", typeof(double), typeof(HomePage), 0.0, coerceValue: CoerceAngle);
public static readonly BindableProperty MaximumAngleProperty = BindableProperty.Create (
  "MaximumAngle", typeof(double), typeof(HomePage), 360.0);
...

static object CoerceAngle (BindableObject bindable, object value)
{
  var homePage = bindable as HomePage;
  double input = (double)value;

  if (input > homePage.MaximumAngle) {
    input = homePage.MaximumAngle;
  }

  return input;
}
```

`CoerceAngle` 方法检查的值 `MaximumAngle` 属性，并且如果 `Angle` 属性值大于它，它强制转换为值 `MaximumAngle` 属性值。

使用 Func 创建默认值

一个 `Func` 可用于初始化的默认值的可绑定的属性，如下面的代码示例中所示：

```
public static readonly BindableProperty SizeProperty =
  BindableProperty.Create ("Size", typeof(double), typeof(HomePage), 0.0,
  defaultValueCreator: bindable => Device.GetNamedSize (NamedSize.Large, (Label)bindable));
```

`defaultValueCreator` 参数设置为 `Func`，它调用 `Device.GetNamedSize` 方法以返回 `double` 表示使用的字体的命名的大小 `Label` 原生平台上。

总结

本文介绍了可绑定属性，并演示了如何创建并使用它们。可绑定属性是属性的特殊类型，其中 Xamarin.Forms 属性系统跟踪属性的值。

相关链接

- XAML 命名空间
- 事件到命令行为（示例）
- 验证回叫（示例）
- 强制值回叫（示例）
- BindableProperty
- BindableObject

附加属性

2018/11/13 • • Edit Online

附加的属性是特殊类型的一个类中定义，但附加到其他对象的可绑定属性和可识别为属性的 XAML 中包含的类和属性名称之间以句点分隔。本文介绍了附加属性，并演示如何创建并使用它们。

概述

附加属性启用要为其自己的类未定义的属性分配值的对象。例如，子元素可使用附加属性，以通知其父元素它们将在用户界面中显示。`Grid` 控件允许的行和子级通过设置指定的列 `Grid.Row` 和 `Grid.Column` 附加属性。`Grid.Row` 并 `Grid.Column` 是附加的属性，因为它们的子级的元素上设置 `Grid`，而不是在 `Grid` 本身。

可绑定属性应作为附加属性在以下方案中实现：

- 定义类需要而不具有用于类可用的属性设置机制时。
- 当类表示一种服务，需要与其他类轻松地集成。

可绑定属性的详细信息，请参阅可绑定属性。

创建和使用附加的属性

创建附加的属性的过程如下所示：

1. 创建 `BindableProperty` 具有的一个实例 `CreateAttached` 方法重载。
2. 提供 `static` `Get` *PropertyName*并 `Set` *PropertyName*方法访问器作为附加属性。

创建属性

在创建时使用的附加的属性在其他类型中，创建属性的类不需要派生自 `BindableObject`。但是，*目标*访问器的属性应是的或派生自，`BindableObject`。

可以通过声明创建附加的属性 `public static readonly` 类型的属性 `BindableProperty`。可绑定属性应设置为返回值之一 `BindableProperty.CreateAttached` 方法重载。声明应在其所属的类，但之外的任何成员定义的正文内。

下面的代码演示附加属性的示例：

```
public static readonly BindableProperty HasShadowProperty =
  BindableProperty.CreateAttached ("HasShadow", typeof(bool), typeof(ShadowEffect), false);
```

这将创建一个名为的附加的属性 `HasShadow`，类型的 `bool`。该属性归 `ShadowEffect` 类，并具有默认值为 `false`。附加属性的命名约定是附加的属性标识符中指定属性名称必须匹配 `CreateAttached` 方法，使用"属性"追加到它。因此，在上面的示例中，附加的属性标识符是 `HasShadowProperty`。

有关创建可绑定属性，包括可以在创建期间，指定的参数的详细信息请参阅创建和使用可绑定属性。

创建访问器

静态 `Get` *PropertyName*并 `Set` *PropertyName*方法都是必需的因为附加属性的访问器，否则将无法使用属性系统附加的属性。`Get` *PropertyName*访问器应符合以下签名：

```
public static valueType GetPropertyName(BindableObject target)
```

`Get` *PropertyName*取值函数的返回值包含在相应应 `BindableProperty` 字段的附加属性。这可以通过调用来实现

`GetValue` 方法，传入要获取的值的可绑定的属性标识符，然后将结果值输出到所需的类型强制转换。

`Set` *PropertyName*访问器应符合以下签名：

```
public static void SetPropertyName(BindableObject target, valueType value)
```

`Set` *PropertyName*访问器应将相应的值设置 `BindableProperty` 字段的附加属性。这可以通过调用来实现 `SetValue` 方法，传入在其上设置值和要设置的值的可绑定的属性标识符。

这两个访问器的*目标*对象应具有的或者派生 `BindableObject` 。

下面的代码示例演示的访问器 `HasShadow` 附加属性：

```
public static bool GetHasShadow (BindableObject view)
{
  return (bool)view.GetValue (HasShadowProperty);
}

public static void SetHasShadow (BindableObject view, bool value)
{
  view.SetValue (HasShadowProperty, value);
}
```

使用附加的属性

一旦创建附加的属性后，可以从 XAML 或代码使用它。在 XAML，这被通过使用命名空间声明，该值指示公共语言运行时 (CLR) 命名空间名称和（可选）程序集名称声明具有前缀的命名空间。有关详细信息，请参阅XAML 命名空间。

下面的代码示例演示了包含所引用的自定义类型的应用程序代码在同一程序集中定义的附加的属性的自定义类型的 XAML 命名空间：

```
<ContentPage ... xmlns:local="clr-namespace:EffectsDemo" ...>
  ...
</ContentPage>
```

当特定控件上的附加的属性设置为以下 XAML 代码示例所示，然后使用命名空间声明：

```
<Label Text="Label Shadow Effect" local:ShadowEffect.HasShadow="true" />
```

以下代码示例显示相应的 C# 代码：

```
var label = new Label { Text = "Label Shadow Effect" };
ShadowEffect.SetHasShadow (label, true);
```

使用附加的属性的方式

由样式之前，还可以将附加的属性添加到控件。以下 XAML 代码示例所示*显式*使用的样式 `HasShadow` 附加属性，可以应用于 `Label` 控件：

```
<Style x:Key="ShadowEffectStyle" TargetType="Label">
  <Style.Setters>
    <Setter Property="local:ShadowEffect.HasShadow" Value="true" />
  </Style.Setters>
</Style>
```

`Style` 可应用于 `Label` 通过设置其 `Style` 属性设置为 `Style` 实例使用 `StaticResource` 标记扩展，如下面的代码示例中所示：

```
<Label Text="Label Shadow Effect" Style="{StaticResource ShadowEffectStyle}" />
```

有关样式的详细信息，请参阅样式。

高级的方案

在创建附加的属性时，有大量可选参数，可设置为启用高级附加的属性的方案。这包括检测属性更改、验证属性值，以及将强制转换属性值。有关详细信息，请参阅高级方案。

总结

本文介绍了附加属性，并演示了如何创建并使用它们。附加的属性是一种特殊的可绑定属性，但附加到其他对象，并可识别在 XAML 中的一个类中定义为包含的类和一个句点分隔的属性名称的属性。

相关链接

- 可绑定属性
- XAML 命名空间
- 阴影效果（示例）
- BindableProperty
- BindableObject

资源字典

2018/11/13 • • Edit Online

XAML 资源是可以共享和重复使用 Xamarin.Forms 应用程序在整个对象的定义。

这些资源对象存储在资源字典中。本文介绍如何创建和使用的资源字典，以及如何合并资源字典。

概述

一个 `ResourceDictionary` 是 Xamarin.Forms 应用程序所使用的资源的存储库。中存储的典型资源 `ResourceDictionary` 包括样式，控件模板，数据模板，颜色和转换器。

在 XAML，资源存储在 `ResourceDictionary` 可检索和使用应用于元素 `StaticResource` 标记扩展。在 C# 中，资源也可以定义在 `ResourceDictionary` 然后检索并应用到通过使用基于字符串的索引器元素。但是，没有什么优势使用 `ResourceDictionary` 在 C# 中，如共享的对象可以只需存储为字段或属性，并不直接访问不会第一个检索这些字典中。

创建和使用 ResourceDictionary

资源定义中 `ResourceDictionary` ，则设置为下列任一 `Resources` 属性：

- `Resources` 派生自任何类的属性 `Application`
- `Resources` 派生自任何类的属性 `VisualElement`

Xamarin.Forms 程序仅包含一个类派生自 `Application` 通常使用许多派生自的类，但 `VisualElement` ，包括页面、布局和控件。任何这些对象可以具有其 `Resources` 属性设置为 `ResourceDictionary` 。选择在何处放置特定 `ResourceDictionary` ，可以使用的资源的影响：

- 中的资源 `ResourceDictionary` 如附加到视图 `Button` 或 `Label` 只能应用于该特定对象，因此这不是很有用。
- 中的资源 `ResourceDictionary` 如附加到布局 `StackLayout` 或 `Grid` 可应用于的布局和布局的所有子级。
- 中的资源 `ResourceDictionary` 定义的页级别可应用到的页和到其所有子项。
- 中的资源 `ResourceDictionary` 定义在应用程序级别可以应用于整个应用程序。

以下 XAML 演示应用程序级别中定义的资源 `ResourceDictionary` 中**App.xaml**标准 Xamarin.Forms 程序的一部分创建的文件：

```
<Application ...>
    <Application.Resources>
        <ResourceDictionary>
            <Color x:Key="PageBackgroundColor">Yellow</Color>
            <Color x:Key="HeadingTextColor">Black</Color>
            <Color x:Key="NormalTextColor">Blue</Color>
            <Style x:Key="LabelPageHeadingStyle" TargetType="Label">
                <Setter Property="FontAttributes" Value="Bold" />
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="TextColor" Value="{StaticResource HeadingTextColor}" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

这 `ResourceDictionary` 定义了三个 `Color` 资源和一个 `Style` 资源。有关详细信息 `App` 类，请参阅App 类。

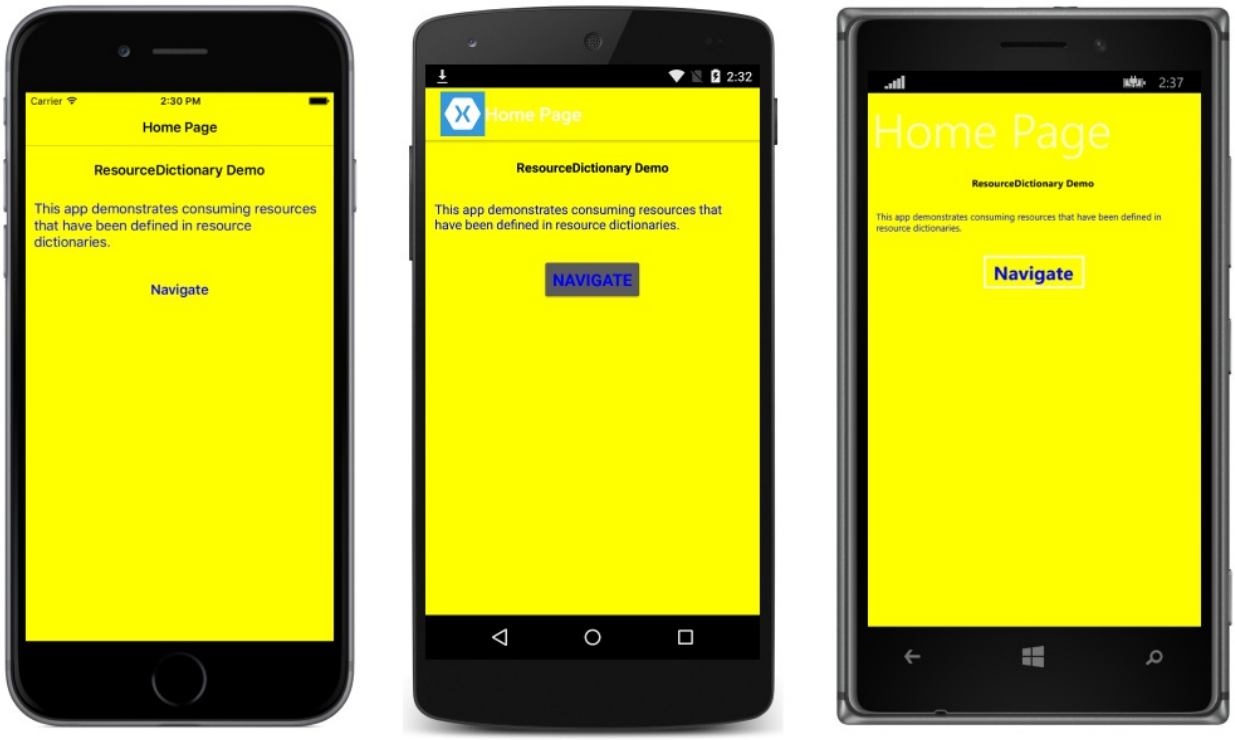
Xamarin.Forms 从 3.0 开始，显式 `ResourceDictionary` 标记不是必需的。 `ResourceDictionary` 对象自动创建的并且可以插入资源之间直接 `Resources` 属性元素标记：

```
<Application ...>
    <Application.Resources>
        <Color x:Key="PageBackgroundColor">Yellow</Color>
        <Color x:Key="HeadingTextColor">Black</Color>
        <Color x:Key="NormalTextColor">Blue</Color>
        <Style x:Key="LabelPageHeadingStyle" TargetType="Label">
            <Setter Property="FontAttributes" Value="Bold" />
            <Setter Property="HorizontalOptions" Value="Center" />
            <Setter Property="TextColor" Value="{StaticResource HeadingTextColor}" />
        </Style>
    </Application.Resources>
</Application>
```

每个资源都有一个使用指定的密钥 `x:Key` 属性，将它作为字典键中的 `ResourceDictionary` 。该密钥用于检索从一个资源 `ResourceDictionary` 由 `StaticResource` 标记扩展，如下面显示了其他资源中定义的 XAML 代码示例中所示 `StackLayout` ：

```
<StackLayout Margin="0,20,0,0">
  <StackLayout.Resources>
    <ResourceDictionary>
      <Style x:Key="LabelNormalStyle" TargetType="Label">
        <Setter Property="TextColor" Value="{StaticResource NormalTextColor}" />
      </Style>
      <Style x:Key="MediumBoldText" TargetType="Button">
        <Setter Property="FontSize" Value="Medium" />
        <Setter Property="FontAttributes" Value="Bold" />
      </Style>
    </ResourceDictionary>
  </StackLayout.Resources>
  <Label Text="ResourceDictionary Demo" Style="{StaticResource LabelPageHeadingStyle}" />
    <Label Text="This app demonstrates consuming resources that have been defined in resource dictionaries."
           Margin="10,20,10,0"
           Style="{StaticResource LabelNormalStyle}" />
    <Button Text="Navigate"
            Clicked="OnNavigateButtonClicked"
            TextColor="{StaticResource NormalTextColor}"
            Margin="0,20,0,0"
            HorizontalOptions="Center"
            Style="{StaticResource MediumBoldText}" />
</StackLayout>
```

第一个 `Label` 实例检索并使用 `LabelPageHeadingStyle` 中的应用程序级别定义的资源 `ResourceDictionary` ，与第二个 `Label` 实例检索和使用 `LabelNormalStyle` 控制级别中定义的资源 `ResourceDictionary` 。同样， `Button` 实例检索并使用 `NormalTextColor` 中的应用程序级别定义的资源 `ResourceDictionary` ，和 `MediumBoldText` 中的控制级别定义资源 `ResourceDictionary` 。这会导致下面的屏幕截图中所示的外观：



> **NOTE**
>
> 不应在应用程序级别的资源字典中，这种情况时所需的页面资源将随后在应用程序启动，而不是可以解析包含特定于单个页面的资源。有关详细信息，请参阅减小应用程序资源字典大小。

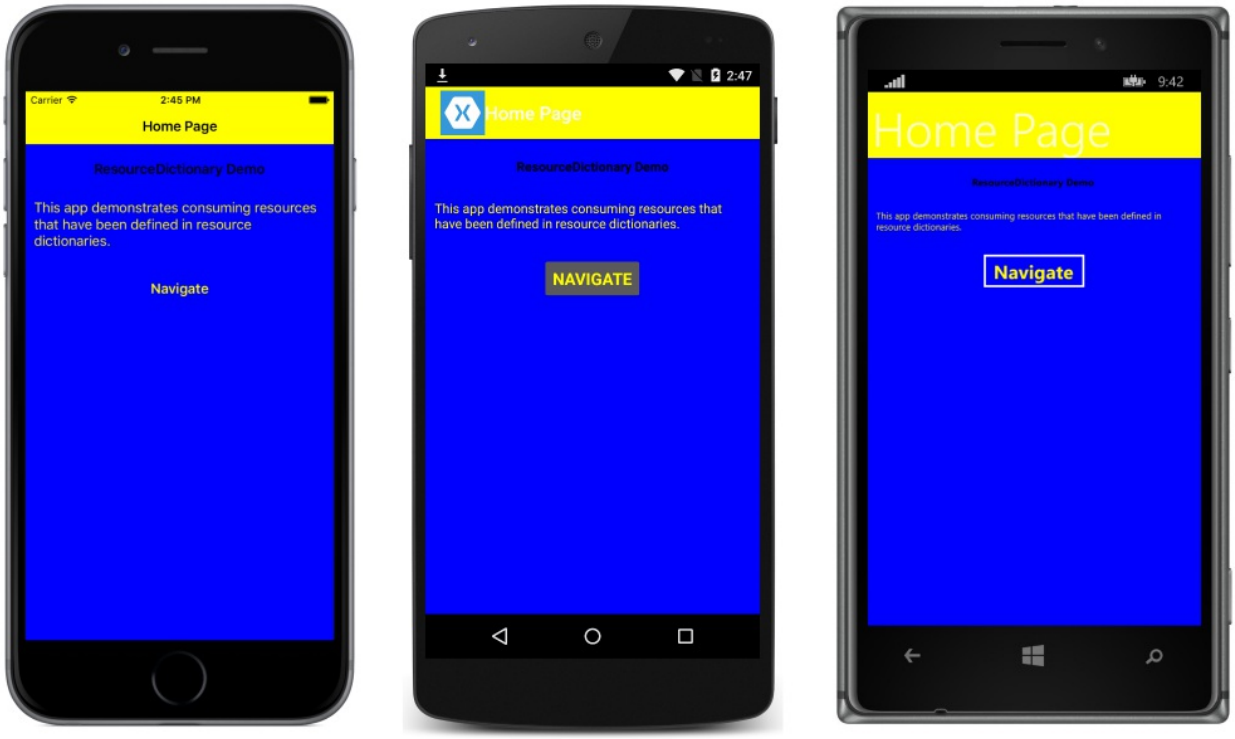
重写的资源

当 `ResourceDictionary` 资源共享 `x:Key` 属性值的视图层次结构中更低时定义的资源将优先于更高版本定义了。例如，设置 `PageBackgroundColor` 资源 `Blue` 在应用程序级别将会重写页面级 `PageBackgroundColor` 资源设置为 `Yellow` 。同样，页面级 `PageBackgroundColor` 资源将被重写由控件级别 `PageBackgroundColor` 资源。下面的 XAML 代码示例演示此优先顺序：

```
<ContentPage ... BackgroundColor="{StaticResource PageBackgroundColor}">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Color x:Key="PageBackgroundColor">Blue</Color>
            <Color x:Key="NormalTextColor">Yellow</Color>
        </ResourceDictionary>
    </ContentPage.Resources>
    <StackLayout Margin="0,20,0,0">
        ...
        <Label Text="ResourceDictionary Demo" Style="{StaticResource LabelPageHeadingStyle}" />
        <Label Text="This app demonstrates consuming resources that have been defined in resource
dictionaries."
               Margin="10,20,10,0"
               Style="{StaticResource LabelNormalStyle}" />
        <Button Text="Navigate"
                Clicked="OnNavigateButtonClicked"
                TextColor="{StaticResource NormalTextColor}"
                Margin="0,20,0,0"
                HorizontalOptions="Center"
                Style="{StaticResource MediumBoldText}" />
    </StackLayout>
</ContentPage>
```

原始 `PageBackgroundColor` 并 `NormalTextColor` 的情况下，应用程序级别定义将取代 `PageBackgroundColor` 和

`NormalTextColor` 页级别定义的实例。因此，页面背景颜色变为蓝色，并且页面上的文本将变为黄色，如以下屏幕截图中所示：



但请注意，背景栏 `NavigationPage` 仍为黄色，因为 `BarBackgroundColor` 属性设置的值为 `PageBackgroundColor` 应用程序中定义的资源级别 `ResourceDictionary` 。

下面是另一种方法来考虑一下 `ResourceDictionary` 优先级：时 XAML 分析程序遇到 `StaticResource` ，它会通过体验在可视化树向上搜索匹配的键，它找到使用第一个匹配项。如果此搜索结束的页上且仍未找到键，XAML 分析器将搜索 `ResourceDictionary` 附加到 `App` 对象。如果仍未找到该键，则引发异常。

独立的资源字典

一个类派生自 `ResourceDictionary` 也可以在单独的独立文件。(更准确地说，派生自的类 `ResourceDictionary` 通常需要_对_的文件由于在 XAML 文件，但具有的代码隐藏文件中定义资源 `InitializeComponent` 调用也是有必要。)然后，可以在应用程序间共享生成的文件。

若要创建此类文件，添加一个新**内容**视图或**内容**页项与项目 (但不是**内容**视图或者**内容**页与仅 C# 文件)。在 XAML 文件和 C# 文件上，从基类的名称更改 `ContentView` 或 `ContentPage` 到 `ResourceDictionary` 。在 XAML 文件中，基类的名称是顶级元素。

下面的 XAML 示例演示 `ResourceDictionary` 名为 `MyResourceDictionary` ：

```
<ResourceDictionary xmlns="http://xamarin.com/schemas/2014/forms"
                    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                    x:Class="ResourceDictionaryDemo.MyResourceDictionary">
    <DataTemplate x:Key="PersonDataTemplate">
        <ViewCell>
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="0.5*" />
                    <ColumnDefinition Width="0.2*" />
                    <ColumnDefinition Width="0.3*" />
                </Grid.ColumnDefinitions>
                <Label Text="{Binding Name}" TextColor="{StaticResource NormalTextColor}"
FontAttributes="Bold" />
                <Label Grid.Column="1" Text="{Binding Age}" TextColor="{StaticResource NormalTextColor}" />
                <Label Grid.Column="2" Text="{Binding Location}" TextColor="{StaticResource NormalTextColor}"
HorizontalTextAlignment="End" />
            </Grid>
        </ViewCell>
    </DataTemplate>
</ResourceDictionary>
```

这 `ResourceDictionary` 包含单个资源，这是一个对象类型 `DataTemplate` 。

可以实例化 `MyResourceDictionary` 通过将它置于一对之间 `Resources` 属性元素标记，例如，在 `ContentPage` ：

```
<ContentPage ...>
    <ContentPage.Resources>
        <local:MyResourceDictionary />
    </ContentPage.Resources>
    ...
</ContentPage>
```

实例 `MyResourceDictionary` 设置为 `Resources` 属性的 `ContentPage` 对象。

但是，此方法具有一些限制： `Resources` 的属性 `ContentPage` 引用仅此一 `ResourceDictionary` 。在大多数情况下，所需的选项包括其他 `ResourceDictionary` 实例和可能是其他资源。

此任务需要合并的资源字典。

合并资源字典

合并的资源字典合并一个或多 `ResourceDictionary` 到另一个实例 `ResourceDictionary` 。可以通过设置执行此 XAML 文件中 `MergedDictionaries` 属性设置为一个或多个将合并到应用程序、页或控件级别的资源字典 `ResourceDictionary` 。

> **IMPORTANT**
>
> `ResourceDictionary` 此外定义了 `MergedWith` 属性。不使用此属性;它开始已弃用 Xamarin.Forms 3.0。

实例 `MyResourceDictionary` 可以合并到任何应用程序、页或控件级别 `ResourceDictionary` 。下面的 XAML 代码示例显示了其合并为一个页级别 `ResourceDictionary` 使用 `MergedDictionaries` 属性：

```
<ContentPage ...>
    <ContentPage.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <local:MyResourceDictionary />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>
```

该标记显示了仅的实例 `MyResourceDictionary` 添加到 `ResourceDictionary` 但你还可以引用其他 `ResourceDictionary` 实例内 `MergedDictionary` 属性元素标记，并位于这些标记之外的其他资源：

```
<ContentPage ...>
    <ContentPage.Resources>
        <ResourceDictionary>

            <!-- Add more resources here -->

            <ResourceDictionary.MergedDictionaries>

                <!-- Add more resource dictionaries here -->

                <local:MyResourceDictionary />

                <!-- Add more resource dictionaries here -->

            </ResourceDictionary.MergedDictionaries>

            <!-- Add more resources here -->

        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>
```

可能只有一个 `MergedDictionaries` 主题中 `ResourceDictionary`，但您可以将任意多个 `ResourceDictionary` 根据需要在这里实例。

合并时 `ResourceDictionary` 资源共享相同 `x:Key` 属性值，Xamarin.Forms 使用以下资源优先级：

1. 资源字典的本地资源中。
2. 合并资源字典中包含的资源通过已弃用 `MergedWith` 属性。
3. 已通过合并资源字典中包含的资源 `MergedDictionaries` 集合中的列出顺序反向 `MergedDictionaries` 属性。

> **NOTE**
> 搜索资源字典可以是计算密集型任务，如果应用程序包含多个大型资源字典。因此，若要避免不需要的搜索，您应确保应用程序中的每一页仅使用适用于页的资源字典。

Xamarin.Forms 3.0 中的合并字典

Xamarin.Forms 3.0，合并在一起的过程开头 `ResourceDictionary` 实例已成为某种程度上更轻松、更灵活。`MergedDictionaries` 属性元素标记不再需要。相反，您将添加到资源字典另 `ResourceDictionary` 与新的标记 `Source` 属性设置为使用的资源的 XAML 文件的文件名：

```
<ContentPage ...>
    <ContentPage.Resources>
        <ResourceDictionary>

            <!-- Add more resources here -->

            <ResourceDictionary Source="MyResourceDictionary.xaml" />

            <!-- Add more resources here -->

        </ResourceDictionary>
    </ContentPage.Resources>
    ...
</ContentPage>
```

因为 Xamarin.Forms 3.0 自动实例化 `ResourceDictionary`，这两个外部 `ResourceDictionary` 标记不再需要：

```
<ContentPage ...>
    <ContentPage.Resources>

        <!-- Add more resources here -->

        <ResourceDictionary Source="MyResourceDictionary.xaml" />

        <!-- Add more resources here -->

    </ContentPage.Resources>
    ...
</ContentPage>
```

这种新语法 does_不_实例化 `MyResourceDictionary` 类。相反，它引用的 XAML 文件。有关原因代码隐藏文件 (**MyResourceDictionary.xaml.cs**) 不再需要。您还可以删除 `x:Class` 的根标记的特性**MyResourceDictionary.xaml**文件。

总结

本文介绍了如何创建和使用 `ResourceDictionary`，以及如何合并资源字典。一个 `ResourceDictionary` 允许在单个位置中，定义和重新整个 Xamarin.Forms 应用程序中使用的资源。

相关链接

- 资源字典（示例）
- 样式
- ResourceDictionary

XAML 标准（预览）

2018/11/13 • • Edit Online

请执行以下步骤，尝试使用 Xamarin.Forms 中 XAML 标准：

- Visual Studio
- Visual Studio for Mac

1. 下载预览此处 NuGet 包。
2. 添加**Xamarin.Forms.Alias**到 Xamarin.Forms.NET Standard 和平台项目的 NuGet 包。
3. 初始化具有包 `Alias.Init()`
4. 添加 `xmlns:a` 引用 `xmlns:a="clr-namespace:Xamarin.Forms.Alias;assembly=Xamarin.Forms.Alias"`
5. 在 XAML 中使用的类型-请参阅控件参考有关详细信息。

以下 XAML 演示了在 Xamarin.Forms 中使用某些 XAML 标准控件 `ContentPage`：

```
<?xml version="1.0" encoding="utf-8"?>
<ContentPage
  xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:a="clr-namespace:Xamarin.Forms.Alias;assembly=Xamarin.Forms.Alias"
  x:Class="XAMLStandardSample.ItemsPage"
  Title="{Binding Title}" x:Name="BrowseItemsPage">
  <ContentPage.ToolbarItems>
    <ToolbarItem Text="Add" Clicked="AddItem_Clicked" />
  </ContentPage.ToolbarItems>
  <ContentPage.Content>
    <a:StackPanel>
      <ListView x:Name="ItemsListView" ItemsSource="{Binding Items}" VerticalOptions="FillAndExpand"
HasUnevenRows="true" RefreshCommand="{Binding LoadItemsCommand}" IsPullToRefreshEnabled="true" IsRefreshing="
{Binding IsBusy, Mode=OneWay}" CachingStrategy="RecycleElement" ItemSelected="OnItemSelected">
        <ListView.ItemTemplate>
          <DataTemplate>
            <ViewCell>
              <StackLayout Padding="10">
                <a:TextBlock Text="{Binding Text}" LineBreakMode="NoWrap" Style="{DynamicResource
ListItemTextStyle}" FontSize="16" />
                <a:TextBlock Text="{Binding Description}" LineBreakMode="NoWrap" Style="{DynamicResource
ListItemDetailTextStyle}" FontSize="13" />
              </StackLayout>
            </ViewCell>
          </DataTemplate>
        </ListView.ItemTemplate>
      </ListView>
    </a:StackPanel>
  </ContentPage.Content>
</ContentPage>
```

> **NOTE**
>
> 需要 xmlns `a:` XAML 标准控件上的前缀是当前的预览版的限制。

相关链接

- 预览 NuGet
- 控件引用

XAML 标准（预览）控件

2018/11/13 • • Edit Online

此页列出了在预览版中，其等效的 Xamarin.Forms 控件一起提供的 XAML 标准控件。

此外，还有在 XAML 标准了新的属性和枚举名称的控件的列表。

控件

XAMARIN.FORMS	XAML 标准
Frame	Border
选取器	组合框
ActivityIndicator	ProgressRing
StackLayout	StackPanel
Label	TextBlock
条目	文本框
开关	ToggleSwitch
ContentView	用户控件

属性和枚举

具有更新的属性的 XAMARIN.FORMS 控件	XAMARIN.FORMS 属性或枚举	XAML 标准等效项
按钮、入口、标签、DatePicker、编辑器、SearchBar、TimePicker	TextColor	Foreground
VisualElement	BackgroundColor	背景 *
选取器中按钮	边框颜色 OutlineColor	BorderBrush
Button	BorderWidth	边框的粗细
ProgressBar	进度	“值”
按钮、入口、标签、编辑器、SearchBar、范围、字体	FontAttributesBold、斜体、无	FontStyleItalic 正常

具有更新的属性的 XAMARIN.FORMS 控件	XAMARIN.FORMS 属性或枚举	XAML 标准等效项
按钮、入口、标签、编辑器、SearchBar、范围、字体	FontAttributes	FontWeights * 加粗、正常
InputView	KeyboardDefault、Url、数字、电话、文本、聊天、电子邮件	InputScopeNameValue * 默认值、Url、数字、TelephoneNumber、文本、聊天、EmailNameOrAddress
StackPanel	StackOrientation	方向 *

IMPORTANT

项标有 * 是在当前的预览版不完整

相关链接

- 预览 NuGet

Xamarin.Forms 应用程序基础知识

2018/7/13 • • Edit Online

辅助功能

若要使用 Xamarin.Forms 合并可访问功能（如支持屏幕阅读工具）的提示。

应用类

`Application` 类是适用于 Xamarin.Forms 的起始点 – 每个应用程序需要实现一个子类 `App` 设置初始页。它还提供了 `Properties` 集合进行简单的数据存储。可以在 C# 或 XAML 中对其进行定义。

应用生命周期

`Application` 类 `OnStart`， `OnSleep`，和 `OnResume` 方法，以及模式导航事件，使您能够处理的自定义代码的应用程序生命周期事件。

行为

用户界面控件可以轻松地扩展而无需子类化使用行为来添加功能。

自定义呈现器

自定义呈现器让开发人员 override Xamarin.Forms 控件以自定义其外观和行为（如果需要使用本机 Sdk）每个平台上的默认呈现。

数据绑定

数据绑定链接两个对象的属性允许在一个属性，才会自动反映在另一个属性中的更改。数据绑定是模型-视图-视图不可或缺的一部分 (MVVM) 应用程序体系结构。

依赖关系服务

`DependencyService` 提供了一个简单的定位符，以便可以共享代码中的代码到接口，并提供特定于平台的实现会自动得到解决，轻松地引用 Xamarin.Forms 中的特定于平台的功能。

效果

效果允许进行自定义，每个平台上的本机控件，通常用于较小的样式更改。

文件

可以使用代码在.NET Standard 库中，或通过使用嵌入的资源来实现处理用于 Xamarin.Forms 的文件。

手势

Xamarin.Forms `GestureRecognizer` 类上的用户界面控件支持 tap、pinch 和平移手势。

本地化

可以使用内置的.NET 本地化框架构建跨平台使用 Xamarin.Forms 的多语言应用程序。

本地数据库

Xamarin.Forms 支持使用 SQLite 数据库引擎，这使得可以加载并将对象保存在共享代码中的数据库驱动的应用程序。

消息中心

Xamarin.Forms `MessagingCenter` 启用视图模型和其他组件以与通信而无需知道有关彼此的任何除了简单的消息约定。

导航

Xamarin.Forms 提供了许多不同的页导航体验，具体取决于 `Page` 键入正在使用。

模板

控件模板让你可以轻松设计或 re 主题在运行时，应用程序页时数据模板提供的功能支持的控件上定义的数据表示形式。

触发器

通过响应属性更改和 XAML 中的事件来更新控件。

相关链接

- Xamarin.Forms 简介

Xamarin.Forms 可访问性

2018/10/26 • • Edit Online

构建可访问的应用程序可确保应用程序是否可由方法通过一系列需求和体验的用户界面的人员。

使 Xamarin.Forms 应用程序可访问意味着考虑的布局和设计的许多用户界面元素。要考虑的问题的指导原则，请参阅可访问性清单。Xamarin.Forms Api 已可解决很多可访问性问题，如大字体和合适的颜色和对比度设置。

Android 可访问性并iOS 可访问性指南包含的 Xamarin，公开的本机 Api 的详细信息并UWP MSDN 上的可访问性指南说明在该平台上的本机方法。这些 Api 用于完全实现每个平台上的可访问应用程序。

当前不具有 Xamarin.Forms*内置*支持所有可访问性 Api 适用于每个基础平台。但是，它支持设置自动化属性在用户界面元素以支持屏幕读取器和导航帮助工具，这是构建可访问应用程序的最重要的部分之一。有关详细信息，请参阅自动化属性。

Xamarin.Forms 应用程序还可以指定的控件的 tab 键顺序。有关详细信息，请参阅的键盘导航。

其他辅助功能 Api (如在 iOS 上的 PostNotification) 可能会更好地适合于 `DependencyService` 或自定义呈现器实现。这些不是在本指南中介绍。

测试辅助功能

Xamarin.Forms 应用程序通常针对多个平台，这意味着测试根据平台的可访问性功能。单击这些链接可了解如何测试每个平台上的辅助功能：

- **iOS 测试**
- **Android 测试**
- **Windows AccScope (MSDN)**

相关链接

- 跨平台可访问性
- 自动化属性
- 键盘辅助功能

在 Xamarin.Forms 中的自动化属性

2018/10/26 • • Edit Online

Xamarin.Forms 允许使用附加的属性从 AutomationProperties 类，后者又设置本机可访问性的值在用户界面元素上设置的可访问性值。本文介绍如何使用 AutomationProperties 类，以便屏幕阅读器可以说出是有关元素的页上。

Xamarin.Forms，可以通过以下附加属性的用户界面元素上设置的自动化属性：

- `AutomationProperties.IsInAccessibleTree` – 指示元素是否可用于可访问的应用程序。有关详细信息，请参阅AutomationProperties.IsInAccessibleTree。
- `AutomationProperties.Name` – 可作为元素的 speakable 标识符的元素的简短说明。有关详细信息，请参阅AutomationProperties.Name。
- `AutomationProperties.HelpText` – 元素，它可以视为与元素关联的工具提示文本的详细说明。有关详细信息，请参阅AutomationProperties.HelpText。
- `AutomationProperties.LabeledBy` – 允许另一个元素定义当前元素的可访问性信息。有关详细信息，请参阅AutomationProperties.LabeledBy。

这些附加属性设置本机可访问性的值，以便屏幕阅读器可以说出的元素。有关附加属性的详细信息，请参阅附加属性。

> **IMPORTANT**
>
> 使用 `AutomationProperties` 附加的属性可能会影响在 Android 上的 UI 测试执行。`AutomationId`，`AutomationProperties.Name` 并 `AutomationProperties.HelpText` 属性将设置本机 `ContentDescription` 属性，与 `AutomationProperties.Name` 和 `AutomationProperties.HelpText` 属性值优先于 `AutomationId` 值（如果这两个 `AutomationProperties.Name` 和 `AutomationProperties.HelpText` 进行设置，则值将连接）。这意味着，任何测试寻找 `AutomationId` 如果将失败 `AutomationProperties.Name` 或 `AutomationProperties.HelpText` 还在元素上设置。在此方案中，应更改的 UI 测试来查找的值 `AutomationProperties.Name` 或 `AutomationProperties.HelpText`，或这两者的串联。

每个平台都有不同的屏幕读取器旁白的可访问性值：

- iOS 具有语音朗读。有关详细信息，请参阅VoiceOver 与你的设备上测试可访问性developer.apple.com 上。
- Android 有 TalkBack。有关详细信息，请参阅测试您的应用程序的可访问性developer.android.com 上。
- Windows 具有讲述人。有关详细信息，请参阅使用讲述人验证主要应用方案。

但是，屏幕阅读器的具体行为取决于对软件和它的用户的配置。例如，大多数屏幕读取器读取时接收焦点，与控件关联的文本使用户能够定位本身，因为它们在页上控件之间移动。某些屏幕阅读器，请参阅整个应用程序用户界面时将显示一个页面，这使用户能够在尝试导航它之前收到所有页面的可用信息性内容。

屏幕阅读器，请参阅其他可访问性值。在示例应用程序：

- 将读取 voiceOver `Placeholder` 的值 `Entry` 后，跟使用该控件的说明。
- 将读取 talkBack `Placeholder` 的值 `Entry` 后，跟 `AutomationProperties.HelpText` 值后，跟使用该控件的说明。
- 讲述人将读取 `AutomationProperties.LabeledBy` 的值 `Entry` 后，跟使用控件的说明。

此外，讲述人将确定优先级 `AutomationProperties.Name`，`AutomationProperties.LabeledBy`，然后 `AutomationProperties.HelpText`。在 Android 上，可以将合并 TalkBack `AutomationProperties.Name` 和 `AutomationProperties.HelpText` 值。因此，建议，全面的可访问性测试上执行每个平台，以确保获得最佳体验。

AutomationProperties.IsInAccessibleTree

`AutomationProperties.IsInAccessibleTree` 附加的属性是 `boolean`，它确定如果该元素是可访问，并且因此可见，对屏幕读取器。它必须设置为 `true` 若要使用其他可访问性附加属性。这可以实现在 XAML 中，如下所示：

```
<Entry AutomationProperties.IsInAccessibleTree="true" />
```

或者，可以将其设置 C# 中，如下所示：

```
var entry = new Entry();
AutomationProperties.SetIsInAccessibleTree(entry, true);
```

> **NOTE**
> 请注意，`SetValue` 方法还可用于设置 `AutomationProperties.IsInAccessibleTree` 附加属性 – `entry.SetValue(AutomationProperties.IsInAccessibleTreeProperty, true);`

AutomationProperties.Name

`AutomationProperties.Name` 附加的属性值应为屏幕读取器使用地宣布推出一个元素的较短的描述性文本字符串。应为具有对于了解内容或与用户界面进行交互非常重要的含义的元素设置此属性。这可以实现在 XAML 中，如下所示：

```
<ActivityIndicator AutomationProperties.IsInAccessibleTree="true"
                   AutomationProperties.Name="Progress indicator" />
```

或者，可以将其设置 C# 中，如下所示：

```
var activityIndicator = new ActivityIndicator();
AutomationProperties.SetIsInAccessibleTree(activityIndicator, true);
AutomationProperties.SetName(activityIndicator, "Progress indicator");
```

> **NOTE**
> 请注意，`SetValue` 方法还可用于设置 `AutomationProperties.Name` 附加属性 – `activityIndicator.SetValue(AutomationProperties.NameProperty, "Progress indicator");`

AutomationProperties.HelpText

`AutomationProperties.HelpText` 附加属性应设置为文本描述用户界面元素，并可以与元素关联的作为工具提示文本的想法。这可以实现在 XAML 中，如下所示：

```
<Button Text="Toggle ActivityIndicator"
        AutomationProperties.IsInAccessibleTree="true"
        AutomationProperties.HelpText="Tap to toggle the activity indicator" />
```

或者，可以将其设置 C# 中，如下所示：

```
var button = new Button { Text = "Toggle ActivityIndicator" };
AutomationProperties.SetIsInAccessibleTree(button, true);
AutomationProperties.SetHelpText(button, "Tap to toggle the activity indicator");
```

NOTE

请注意，`SetValue` 方法还可用于设置 `AutomationProperties.HelpText` 附加属性 –

```
button.SetValue(AutomationProperties.HelpTextProperty, "Tap to toggle the activity indicator");
```

在某些平台上以进行编辑控件，如 `Entry`，则 `HelpText` 属性有时可以省略，替换为占位符文本。例如，"输入您的姓名"是的良好候选项 `Entry.Placeholder` 将文本放在用户的实际输入之前在控件中的属性。

AutomationProperties.LabeledBy

`AutomationProperties.LabeledBy` 附加的属性允许另一个元素定义当前元素的可访问性信息。例如，`Label` 旁边 `Entry` 可以用于描述什么 `Entry` 表示。这可以实现在 XAML 中，如下所示：

```
<Label x:Name="label" Text="Enter your name: " />
<Entry AutomationProperties.IsInAccessibleTree="true"
       AutomationProperties.LabeledBy="{x:Reference label}" />
```

或者，可以将其设置 C# 中，如下所示：

```
var nameLabel = new Label { Text = "Enter your name: " };
var entry = new Entry();
AutomationProperties.SetIsInAccessibleTree(entry, true);
AutomationProperties.SetLabeledBy(entry, nameLabel);
```

NOTE

请注意，`SetValue` 方法还可用于设置 `AutomationProperties.IsInAccessibleTree` 附加属性 –

```
entry.SetValue(AutomationProperties.LabeledByProperty, nameLabel);
```

相关链接

- 附加属性
- 可访问性（示例）

在 Xamarin.Forms 中的键盘导航

2018/10/26 • • Edit Online

某些用户都可以使用不提供相应的键盘访问的应用程序引起的困难。指定控件的 tab 键顺序使键盘导航并准备应用程序页，以按特定顺序接收输入。

默认情况下，控件的 tab 键顺序是它们是列入 XAML，或以编程方式添加到子集合的相同顺序。此顺序是在其中控件将导航到与键盘、顺序和此默认顺序通常是最佳顺序。但是，默认顺序并不总是与预期的顺序相同，在下面的 XAML 代码示例所示：

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.5*" />
        <ColumnDefinition Width="0.5*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Label Text="You"
           HorizontalOptions="Center" />
    <Label Grid.Column="1"
           Text="Manager"
           HorizontalOptions="Center" />
    <Entry Grid.Row="1"
           Placeholder="Enter forename" />
    <Entry Grid.Column="1"
           Grid.Row="1"
           Placeholder="Enter forename" />
    <Entry Grid.Row="2"
           Placeholder="Enter surname" />
    <Entry Grid.Column="1"
           Grid.Row="2"
           Placeholder="Enter surname" />
</Grid>
```

下面的屏幕截图显示了此代码示例的默认选项卡顺序：

此处的 tab 键顺序是基于行，控件 XAML 中的列出顺序。因此，按 Tab 键导航名 `Entry` 情况下后，跟姓氏 `Entry` 实例。但是，更直观的体验是使用列的第一个选项卡导航栏中，以便按 Tab 键导航名姓氏对。这可以通过指定的输入控件的 tab 键顺序来实现。

> **NOTE**
>
> 通用 Windows 平台上的键盘快捷方式可定义，使应用程序的可见 UI 通过键盘，而不是通过触摸或鼠标用户可以快速导航并进行交互以直观的方式。有关详细信息，请参阅设置 VisualElement 访问密钥。

设置 tab 键顺序

`VisualElement.TabIndex` 属性用于指示的顺序 `VisualElement` 实例时用户通过按 Tab 键导航控件接收焦点。该属性的默认值为 0，并将它设置为任何 `int` 值。

以下规则适用时使用默认 tab 键顺序，或通过 `TabIndex` 属性：

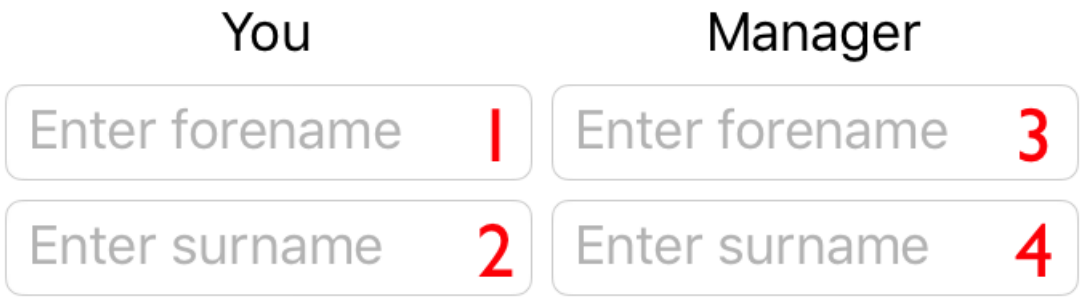
- `VisualElement` 实例 `TabIndex` 等于 0 添加到基于其在 XAML 或子集合中的声明顺序的 tab 键顺序。
- `VisualElement` 实例 `TabIndex` 大于 0 添加到 tab 键顺序基于其 `TabIndex` 值。
- `VisualElement` 实例 `TabIndex` 小于 0 添加到 tab 键顺序，并且出现前面任何零值。
- 在冲突 `TabIndex` 通过按声明顺序来解析。

在定义后 tab 键顺序，按 Tab 键将循环移动焦点控件按升序 `TabIndex` 顺序，一旦达到最终的控件周围包装到开头。

下面的 XAML 示例演示 `TabIndex` 设置输入控件的启用列第一个选项卡导航属性：

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.5*" />
        <ColumnDefinition Width="0.5*" />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Label Text="You"
           HorizontalOptions="Center" />
    <Label Grid.Column="1"
           Text="Manager"
           HorizontalOptions="Center" />
    <Entry Grid.Row="1"
           Placeholder="Enter forename"
           TabIndex="1" />
    <Entry Grid.Column="1"
           Grid.Row="1"
           Placeholder="Enter forename"
           TabIndex="3" />
    <Entry Grid.Row="2"
           Placeholder="Enter surname"
           TabIndex="2" />
    <Entry Grid.Column="1"
           Grid.Row="2"
           Placeholder="Enter surname"
           TabIndex="4" />
</Grid>
```

下面的屏幕截图显示了此代码示例的选项卡顺序：



此处的 tab 键顺序是基于列的。因此，按 Tab 键导航名姓氏 `Entry` 对。

不从 tab 键顺序包括控件

除了设置控件的 tab 键顺序，可能需要从 tab 键顺序中排除控件。这是通过设置一种方法的实现 `IsEnabled` 属性的控件，以便 `false`，因为已禁用的控件不在 tab 键顺序。

但是，可能需要从 tab 键顺序中排除控件，即使它们不已禁用。这可以通过 `VisualElement.IsTapStop` 属性，用于指示是否 `VisualElement` 包含在 tab 导航。其默认值是 `true`，并且其值为时 `false` 如果控件将被忽略的 tab 键导航基础结构，而无论 `TabIndex` 设置。

支持的控件

`TabIndex` 和 `IsTapStop` 接受键盘输入一个或多个平台上的以下控件支持属性：

- `Button`
- `DatePicker`
- `Editor`
- `Entry`
- `NavigationPage`
- `Picker`
- `ProgressBar`
- `SearchBar`
- `Slider`
- `Stepper`
- `Switch`
- `TabbedPage`
- `TimePicker`

> **NOTE**
> 每个控件不是每个平台上可获得焦点的选项卡。

相关链接

- 可访问性（示例）

Xamarin.Forms 应用程序类

2018/11/1 • • Edit Online

`Application` 基类提供以下功能，它们在默认的项目中公开 `App` 子类：

- 一个 `MainPage` 属性，它是在何处设置应用的初始页。
- 持久 `Properties` 字典在生命周期状态更改存储简单值。
- 静态 `Current` 属性，其中包含对当前应用程序对象的引用。

它还公开生命周期方法如 `OnStart`， `OnSleep`，和 `OnResume` 以及模式导航的事件。

具体的模板取决于您选择，`App` 类可以定义在两种方式之一：

- **C#**，或
- **XAML 和 C#**

若要创建应用程序类使用 XAML，默认应用类必须替换 XAML应用类和关联的代码隐藏，如下面的代码示例中所示：

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Photos.App">

</Application>
```

下面的代码示例显示了关联的代码隐藏：

```
public partial class App : Application
{
    public App ()
    {
        InitializeComponent ();
        MainPage = new HomePage ();
    }
    ...
}
```

设置以及 `MainPage` 属性，还必须调用代码隐藏 `InitializeComponent` 方法来加载和分析相关联的 XAML。

MainPage 属性

`MainPage` 属性上的 `Application` 类设置的根页的应用程序。

例如，可以创建在逻辑应用 `App` 类，以显示不同的页，具体取决于是否用户登录或未。

`MainPage` 属性应设置 `App` 构造函数，

```
public class App : Xamarin.Forms.Application
{
    public App ()
    {
        MainPage = new ContentPage { Title = "App Lifecycle Sample" }; // your page here
    }
}
```

属性字典

`Application`子类具有静态`Properties`可以使用字典来存储数据，特别是在中使用`OnStart`， `OnSleep`，和`OnResume`方法。这可以从访问，你的代码使用 Xamarin.Forms 中的任意位置`Application.Current.Properties`。

`Properties` Dictionary 使用`string`密钥和存储`object`值。

例如，您可以设置永久`"id"`在代码中的任意位置的属性（当选择项，在页面的`OnDisappearing`方法，或在`OnSleep`方法）如下所示：

```
Application.Current.Properties ["id"] = someClass.ID;
```

在中`OnStart`或`OnResume`方法可以使用此值以重新创建以某种方式的用户的体验。`Properties`字典存储`object`s，因此您需要其值强制转换之前使用它。

```
if (Application.Current.Properties.ContainsKey("id"))
{
    var id = Application.Current.Properties ["id"] as int;
    // do something with id
}
```

始终检查存在的密钥才能访问它以防止出现意外的错误。

> **NOTE**
>
> `Properties`字典可以仅序列化的存储基元类型。尝试存储的其他类型（如`List<string>`）可能会以静默方式失败。

持久性

`Properties`字典自动保存到设备。当应用程序将返回从后台或甚至重新启动之后，将可添加到字典中的数据。

Xamarin.Forms 1.4 上引入的其他方法`Application`类的`SavePropertiesAsync()`-这可以调用以进行主动保留`Properties`字典。这将允许您保存后的重要更新的属性，而不是它们不获取序列化出由于故障或丢失的 os 的风险。

您可以查找对使用的引用`Properties`字典中的使用 Xamarin.Forms 创建移动应用书籍章节6， 15，和20，与关联示例。

Application 类

完整`Application`类的实现如下所示的引用：

```
public class App : Xamarin.Forms.Application
{
    public App ()
    {
        MainPage = new ContentPage { Title = "App Lifecycle Sample" }; // your page here
    }

    protected override void OnStart()
    {
        // Handle when your app starts
        Debug.WriteLine ("OnStart");
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
        Debug.WriteLine ("OnSleep");
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
        Debug.WriteLine ("OnResume");
    }
}
```

然后在每个特定于平台的项目中实例化并传递给此类 `LoadApplication` 这是 where 方法 `MainPage` 加载并显示给用户。以下各节中显示为每个平台的代码。最新的 Xamarin.Forms 解决方案模板已包含所有这些代码，为您的应用程序预先配置的。

iOS 项目

IOS `AppDelegate` 类继承自 `FormsApplicationDelegate` 。它应当：

- 调用 `LoadApplication` 的实例与 `App` 类。
- 始终返回 `base.FinishedLaunching (app, options);` 。

```
[Register ("AppDelegate")]
public partial class AppDelegate :
    global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate // superclass new in 1.3
{
    public override bool FinishedLaunching (UIApplication app, NSDictionary options)
    {
        global::Xamarin.Forms.Forms.Init ();

        LoadApplication (new App ());  // method is new in 1.3

        return base.FinishedLaunching (app, options);
    }
}
```

Android 项目

Android `MainActivity` 继承 `FormsAppCompatActivity` 。在中 `OnCreate` 重写 `LoadApplication` 使用的实例调用方法 `App` 类。

```
[Activity (Label = "App Lifecycle Sample", Icon = "@drawable/icon", Theme = "@style/MainTheme", MainLauncher
= true,
    ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
public class MainActivity : FormsAppCompatActivity
{
    protected override void OnCreate (Bundle bundle)
    {
        base.OnCreate (bundle);

        global::Xamarin.Forms.Forms.Init (this, bundle);

        LoadApplication (new App ()); // method is new in 1.3
    }
}
```

适用于 Windows 10 的通用 Windows 项目 (UWP)

请参阅安装 Windows 项目有关 Xamarin.Forms 中的 UWP 支持信息。

在 UWP 项目的主页应继承自 `WindowsPage`：

```
<forms:WindowsPage
   ...
   xmlns:forms="using:Xamarin.Forms.Platform.UWP"
   ...>
</forms:WindowsPage>
```

C#代码隐藏构造必须调用 `LoadApplication` 创建一个实例在 Xamarin.Forms `App`。请注意，它是显式使用应用程序的命名空间限定的好办法 `App` 因为 UWP 应用程序也有其自己 `App` 与 Xamarin.Forms 不相关的类。

```
public sealed partial class MainPage
{
    public MainPage()
    {
        InitializeComponent();

        LoadApplication(new YOUR_NAMESPACE.App());
    }
 }
```

请注意，`Forms.Init()` 必须调用**App.xaml.cs**围绕第 63 行。

Xamarin.Forms 应用程序生命周期

2018/11/1 • • Edit Online

`Application` 基类提供了以下功能：

- 生命周期方法 `OnStart`， `OnSleep`，和 `OnResume`。
- 页导航事件 `PageAppearing`， `PageDisappearing`。
- 模式导航事件 `ModalPushing`， `ModalPushed`， `ModalPopping`，和 `ModalPopped`。

生命周期方法

`Application` 类包含三个可以重写以处理生命周期方法的虚拟方法：

- **OnStart** -应用程序启动时调用。
- **OnSleep** -应用程序转到在后台每次调用。
- **OnResume** -时恢复应用程序时，发送到后台之后，调用。

请注意，*没有*应用程序终止时的方法。从在正常情况下（即不崩溃）应用程序终止时将发生*OnSleep*状态，而无需任何其他通知到您的代码。

若要观察调用这些方法时，实现 `WriteLine`（如下所示），在每个调用和每个平台上测试。

```
protected override void OnStart()
{
    Debug.WriteLine ("OnStart");
}
protected override void OnSleep()
{
    Debug.WriteLine ("OnSleep");
}
protected override void OnResume()
{
    Debug.WriteLine ("OnResume");
}
```

更新时 *旧*Xamarin.Forms 应用程序（例如。创建与 Xamarin.Forms 1.3 或更低版本），请确保 Android 主活动，包括 `ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation` 在 `[Activity()]` 属性。如果此项不存在将观察 `OnStart` 上旋转以及应用程序首次启动时调用方法。此属性自动包含在当前的 Xamarin.Forms 应用程序模板。

页面导航事件

上有两个事件 `Application` 提供的页面出现然后消失，通知的类：

- `PageAppearing` -要在屏幕上显示一个页面时引发。
- `PageDisappearing` -当一个页面是要从屏幕上消失时引发。

这些事件可以在你想要跟踪页，因为它们在屏幕上出现在方案中使用。

NOTE

`PageAppearing` 并 `PageDisappearing` 从引发事件 `Page` 基类后立即 `Page.Appearing` 并 `Page.Disappearing` 事件，分别。

模式导航事件

上有四个事件 `Application` 类，每个都有其自己的事件参数，可响应模式页面所显示并已解除的：

- **ModalPushing** - `ModalPushingEventArgs`
- **ModalPushed** - `ModalPushedEventArgs`
- **ModalPopping** - `ModalPoppingEventArgs` 类包含 `Cancel` 属性。当 `Cancel` 设置为 `true` 模式 pop 将被取消。
- **ModalPopped** - `ModalPoppedEventArgs`

NOTE

若要实现的应用程序生命周期方法和模式导航事件，所有预 `Application` 创建 Xamarin.Forms 应用的方法 (即编写的应用程序在版本 1.2 或更低版本中，使用静态 `GetMainPage` 方法) 已更新，以创建默认值 `Application` 将其设置为父级的 `MainPage`。

使用这一旧行为的 Xamarin.Forms 应用程序必须更新到 `Application` 子类上所述应用程序类页。

Xamarin.Forms 行为

2018/7/13 • • Edit Online

行为，可以将功能添加到用户界面控件，而它们不必子类。在代码中编写并添加到 XAML 或代码中的控件行为。

行为简介

行为，可实现您通常必须编写为代码隐藏中，因为它直接与 API 的方式，可以将它用于简明地附加到该控件的控件进行交互的代码。本文介绍的行为。

附加行为

附加的行为是 `static` 具有一个或多个附加属性的类。本文演示如何创建和使用附加的行为。

Xamarin.Forms 行为

Xamarin.Forms 行为创建的派生自 `Behavior` 或 `Behavior<T>` 类。本文演示如何创建和使用 Xamarin.Forms 行为。

可重用行为

行为是可重复使用跨多个应用程序。这些文章介绍如何创建有用的行为来执行常用的功能。

行为简介

2018/7/13 • • Edit Online

行为使您可以将功能添加到用户界面控件，而它们不必子类。相反，该功能是行为类中实现并附加到控件，就像它是控件本身的一部分。本文介绍的行为。

行为，可实现您通常必须编写为代码隐藏中，因为它直接与 API 中，它可以是用于简明地附加到控件，并且跨多个应用程序打包以供重复使用的方式的控件进行交互的代码。它们可用于提供一系列完整的功能提供给控件，例如：

- 添加到电子邮件验证程序 `Entry` 。
- 创建分级控件使用的点击手势识别程序。
- 控制动画。
- 添加到控件的效果。

行为还可实现更高级的方案。中的上下文*发出命令*，行为是用于连接到命令的控件很有用的方法。此外，它们可以用于将命令与不设计与命令进行交互的控件相关联。例如，它们可用于调用以响应事件触发命令。

Xamarin.Forms 支持两种不同样式的行为：

- **Xamarin.Forms 行为**– 派生的类 `Behavior` 或 `Behavior<T>` 类，其中 `T` 是该控件的类型行为应该应用。有关 Xamarin.Forms 行为的详细信息，请参阅Xamarin.Forms 行为并可重用行为。
- **附加行为**– `static` 具有一个或多个附加属性的类。有关附加行为的详细信息，请参阅附加行为。

本指南重点 Xamarin.Forms 行为，因为它们是行为构造的首选的方法。

相关链接

- 行为
- 行为<T>

附加的行为

2018/7/13 • • Edit Online

附加的行为是具有一个或多个附加属性的静态类。本文演示如何创建和使用附加的行为。

概述

附加的属性是一种特殊的可绑定属性。它们是一个类中定义，但附加到其他对象，并且它们是可识别在 XAML 中作为包含的类和一个句点分隔的属性名称的属性。

可以定义附加的属性 `propertyChanged` 属性的值发生更改，例如当属性设置在控件上时将执行的委托。当 `propertyChanged` 委托执行时，已通过对附加的控件在其上它正在和包含该属性的旧的和新值的参数的引用。此委托可用于将新功能添加到该属性附加到通过操作中，按如下所示传递的引用的控件：

1. `propertyChanged` 委托将强制转换控件引用，即作为接收 `BindableObject` 到的控件类型的行为是用于增强。
2. `propertyChanged` 委托修改属性的控件的控件，来实现的核心行为功能公开的事件的控件或注册事件处理程序调用方法。

附加行为的一个问题是它们在定义 `static` 类，与 `static` 属性和方法。这使得难以创建具有状态的附加的行为。此外，Xamarin.Forms 行为已替换为行为构造的首选方法的附加的行为。有关 Xamarin.Forms 行为的详细信息，请参阅Xamarin.Forms 行为并可重用行为。

创建附加的行为

该示例应用程序演示 `NumericValidationBehavior`，其中突出显示了通过到用户输入的值 `Entry` 控件中以红色，如果不是 `double`。行为是在下面的代码示例所示：

```
public static class NumericValidationBehavior
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached (
            "AttachBehavior",
            typeof(bool),
            typeof(NumericValidationBehavior),
            false,
            propertyChanged:OnAttachBehaviorChanged);

    public static bool GetAttachBehavior (BindableObject view)
    {
        return (bool)view.GetValue (AttachBehaviorProperty);
    }

    public static void SetAttachBehavior (BindableObject view, bool value)
    {
        view.SetValue (AttachBehaviorProperty, value);
    }

    static void OnAttachBehaviorChanged (BindableObject view, object oldValue, object newValue)
    {
        var entry = view as Entry;
        if (entry == null) {
            return;
        }

        bool attachBehavior = (bool)newValue;
        if (attachBehavior) {
            entry.TextChanged += OnEntryTextChanged;
        } else {
            entry.TextChanged -= OnEntryTextChanged;
        }
    }

    static void OnEntryTextChanged (object sender, TextChangedEventArgs args)
    {
        double result;
        bool isValid = double.TryParse (args.NewTextValue, out result);
        ((Entry)sender).TextColor = isValid ? Color.Default : Color.Red;
    }
}
```

`NumericValidationBehavior` 类包含一个名为的附加的属性 `AttachBehavior` 与 `static` getter 和 setter，它控制的添加或删除到要附加的控件的行为。此附加属性寄存器 `OnAttachBehaviorChanged` 属性的值发生更改时将执行的方法。此方法注册或取消注册的事件处理程序 `TextChanged` 事件，值的基础 `AttachBehavior` 附加属性。提供的核心功能的行为 `OnEntryTextChanged` 方法，将值输入到分析 `Entry` 的用户和组 `TextColor` 属性设置为红色，如果该值不 `double` 。

使用附加的行为

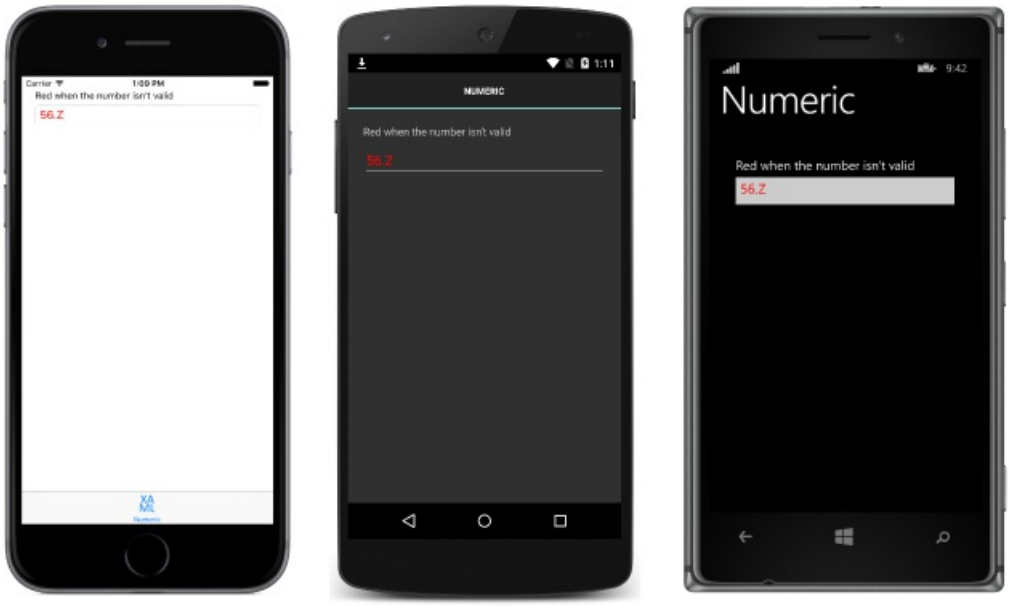
`NumericValidationBehavior` 类可供添加 `AttachBehavior` 附加到的属性 `Entry` 控制，如以下 XAML 代码示例所示：

```
<ContentPage ... xmlns:local="clr-namespace:WorkingWithBehaviors;assembly=WorkingWithBehaviors" ...>
    ...
    <Entry Placeholder="Enter a System.Double" local:NumericValidationBehavior.AttachBehavior="true" />
    ...
</ContentPage>
```

等效于 `Entry` C# 中所示下面的代码示例：

```
var entry = new Entry { Placeholder = "Enter a System.Double" };
NumericValidationBehavior.SetAttachBehavior (entry, true);
```

在运行时，行为将根据行为实现响应与该控件的交互。下面的屏幕截图演示了如何对无效输入进行响应的附加的行为：



> **NOTE**
> 附加的行为针对特定的控件类型（或适用于许多控件的超类）编写的它们只应添加到一个兼容的控件。尝试将行为附加到不兼容的控件将导致未知行为，并取决于行为实现。

从控件中删除附加的行为

`NumericValidationBehavior` 类可以通过设置控件中移除 `AttachBehavior` 附加属性设置为 `false`，如以下 XAML 代码示例所示：

```
<Entry Placeholder="Enter a System.Double" local:NumericValidationBehavior.AttachBehavior="false" />
```

等效于 `Entry` C# 中所示下面的代码示例：

```
var entry = new Entry { Placeholder = "Enter a System.Double" };
NumericValidationBehavior.SetAttachBehavior (entry, false);
```

在运行时，`OnAttachBehaviorChanged` 方法将时执行的值 `AttachBehavior` 附加的属性设置为 `false`。 `OnAttachBehaviorChanged` 方法将然后取消注册的事件处理程序 `TextChanged` 事件，确保当用户与控件交互时，不会执行行为。

总结

本文演示了如何创建和使用附加的行为。附加的行为是 `static` 具有一个或多个附加属性的类。

相关链接

- 附加的行为（示例）

Xamarin.Forms 行为

2018/7/13 • • Edit Online

Xamarin.Forms 行为创建的派生自的行为或行为类。本文演示如何创建和使用 Xamarin.Forms 行为。

概述

创建 Xamarin.Forms 行为的过程如下所示：

1. 创建一个类继承自 `Behavior` 或 `Behavior<T>` 类，其中 `T` 是一种行为要应用到的控件。
2. 重写 `OnAttachedTo` 方法来执行任何所需的设置。
3. 重写 `OnDetachingFrom` 方法来执行任何所需的清理。
4. 实现的核心功能的行为。

这会导致在下面的代码示例所示的结构：

```
public class CustomBehavior : Behavior<View>
{
    protected override void OnAttachedTo (View bindable)
    {
        base.OnAttachedTo (bindable);
        // Perform setup
    }

    protected override void OnDetachingFrom (View bindable)
    {
        base.OnDetachingFrom (bindable);
        // Perform clean up
    }

    // Behavior implementation
}
```

`OnAttachedTo` 行为附加到控件后将立即激发方法。此方法将接收到它已附加，并可用于注册事件处理程序或执行其他安装程序支持行为功能所需的控件的引用。例如，您可以订阅一个控件中的事件。然后将该事件在事件处理程序实现行为功能。

`OnDetachingFrom` 方法在从控件中移除行为时被激发。此方法将接收到它已附加，并用于执行任何所需的清理控件的引用。例如，可以从上一个控件，以防止内存泄漏事件取消订阅。

然后可以通过将附加到已使用行为 `Behaviors` 相应的控件的集合。

创建 Xamarin.Forms 行为

该示例应用程序演示 `NumericValidationBehavior`，其中突出显示了通过到用户输入的值 `Entry` 控件中以红色，如果不是 `double`。行为是在下面的代码示例所示：

```
public class NumericValidationBehavior : Behavior<Entry>
{
    protected override void OnAttachedTo(Entry entry)
    {
        entry.TextChanged += OnEntryTextChanged;
        base.OnAttachedTo(entry);
    }

    protected override void OnDetachingFrom(Entry entry)
    {
        entry.TextChanged -= OnEntryTextChanged;
        base.OnDetachingFrom(entry);
    }

    void OnEntryTextChanged(object sender, TextChangedEventArgs args)
    {
        double result;
        bool isValid = double.TryParse (args.NewTextValue, out result);
        ((Entry)sender).TextColor = isValid ? Color.Default : Color.Red;
    }
}
```

`NumericValidationBehavior` 派生自 `Behavior<T>` 类，其中 `T` 是 `Entry` 。`OnAttachedTo` 方法注册的事件处理程序 `TextChanged` 事件，与 `OnDetachingFrom` 方法取消注册 `TextChanged` 事件，以防止内存泄漏。提供的核心功能的行为 `OnEntryTextChanged` 方法，将为用户输入的值解析 `Entry` ，并设置 `TextColor` 属性设置为红色，如果该值不 `double` 。

> **NOTE**
>
> Xamarin.Forms 不会设置 `BindingContext` 的行为，因为可以共享行为，并将其应用于多个控件通过样式。

使用 Xamarin.Forms 行为

每个 Xamarin.Forms 控件具有 `Behaviors` ，添加到集合的一个或多个行为可以如以下 XAML 代码示例所示：

```
<Entry Placeholder="Enter a System.Double">
    <Entry.Behaviors>
        <local:NumericValidationBehavior />
    </Entry.Behaviors>
</Entry>
```

等效于 `Entry` C# 中所示下面的代码示例：

```
var entry = new Entry { Placeholder = "Enter a System.Double" };
entry.Behaviors.Add (new NumericValidationBehavior ());
```

在运行时行为将根据行为实现响应与该控件的交互。下面的屏幕截图演示了如何对无效输入进行响应的行为：

> **NOTE**
>
> 行为针对特定的控件类型（或适用于许多控件的超类）编写的它们只应添加到一个兼容的控件。尝试将行为附加到不兼容的控件将导致引发异常。

使用 Xamarin.Forms 行为的方式

显式或隐式样式也可以使用行为。但是，创建设置样式 `Behaviors` 控件的属性不能，因为该属性是只读的。解决方案是将附加的属性添加到控件添加和删除行为的行为类。过程如下所示：

1. 将附加的属性添加到将用于控制添加或删除行为将附加的控件的行为的行为类。请确保附加的属性注册 `propertyChanged` 属性的值发生更改时将执行的委托。
2. 创建 `static` getter 和 setter 的附加属性。
3. 实现中的逻辑 `propertyChanged` 委托来添加和删除行为。

下面的代码示例显示了附加的属性，用于控制添加和删除 `NumericValidationBehavior`：

```
public class NumericValidationBehavior : Behavior<Entry>
{
    public static readonly BindableProperty AttachBehaviorProperty =
        BindableProperty.CreateAttached ("AttachBehavior", typeof(bool), typeof(NumericValidationBehavior),
false, propertyChanged: OnAttachBehaviorChanged);

    public static bool GetAttachBehavior (BindableObject view)
    {
        return (bool)view.GetValue (AttachBehaviorProperty);
    }

    public static void SetAttachBehavior (BindableObject view, bool value)
    {
        view.SetValue (AttachBehaviorProperty, value);
    }

    static void OnAttachBehaviorChanged (BindableObject view, object oldValue, object newValue)
    {
        var entry = view as Entry;
        if (entry == null) {
            return;
        }

        bool attachBehavior = (bool)newValue;
        if (attachBehavior) {
            entry.Behaviors.Add (new NumericValidationBehavior ());
        } else {
            var toRemove = entry.Behaviors.FirstOrDefault (b => b is NumericValidationBehavior);
            if (toRemove != null) {
                entry.Behaviors.Remove (toRemove);
            }
        }
    }
    ...
}
```

`NumericValidationBehavior` 类包含一个名为的附加的属性 `AttachBehavior` 与 `static` getter 和 setter，它控制的添加或删除到要附加的控件的行为。此附加属性寄存器 `OnAttachBehaviorChanged` 属性的值发生更改时将执行的方法。此方法添加或删除到值的基础的控件行为 `AttachBehavior` 附加属性。

下面的代码示例演示*显式*的样式 `NumericValidationBehavior`，它使用 `AttachBehavior` 附加属性，并将其应用于 `Entry` 控件：

```
<Style x:Key="NumericValidationStyle" TargetType="Entry">
    <Style.Setters>
        <Setter Property="local:NumericValidationBehavior.AttachBehavior" Value="true" />
    </Style.Setters>
</Style>
```

`Style` 可应用于 `Entry` 通过设置其 `Style` 属性设置为 `Style` 使用实例的步骤 `StaticResource` 标记扩展，如下面的代码示例中所示：

```
<Entry Placeholder="Enter a System.Double" Style="{StaticResource NumericValidationStyle}">
```

有关样式的详细信息，请参阅样式。

> **NOTE**
> 虽然您可以添加可绑定属性设置或查询中 XAML，如果执行操作，创建行为的行为将状态它们不应共享中的控件之间 `Style` 在 `ResourceDictionary` 。

从控件中删除行为

`OnDetachingFrom` 方法触发行为已从一个控件，以及用于执行任何所需的清理如取消订阅事件，以防止内存泄漏。但是，行为不会隐式删除从控件除非控件的 `Behaviors` 通过修改集合 `Remove` 或 `Clear` 方法。下面的代码示例演示了如何从控件的移除特定行为 `Behaviors` 集合：

```
var toRemove = entry.Behaviors.FirstOrDefault (b => b is NumericValidationBehavior);
if (toRemove != null) {
    entry.Behaviors.Remove (toRemove);
}
```

或者，该控件的 `Behaviors` 可以清除集合，如下面的代码示例中所示：

```
entry.Behaviors.Clear();
```

此外，请注意，行为不会隐式从删除控件时从导航堆栈中弹出页。相反，它们之前，必须显式删除超出作用域的页。

总结

本文演示了如何创建和使用 Xamarin.Forms 行为。Xamarin.Forms 行为创建的派生自 `Behavior` 或 `Behavior<T>` 类。

相关链接

- Xamarin.Forms 行为（示例）
- Xamarin.Forms 行为应用具有样式（示例）
- 行为
- 行为

可重用行为

2018/11/2 • • Edit Online

行为是可重复使用跨多个应用程序。这些文章介绍如何创建有用的行为来执行常用的功能。

可重用 EffectBehavior

行为是将效果添加到控件时，删除处理代码中的代码隐藏文件的样板效果很有用的方法。本文演示如何创建和使用 Xamarin.Forms 行为添加到控件的效果。

可重用 EventToCommandBehavior

行为可用于将命令与不设计与命令进行交互的控件相关联。本文演示如何创建和使用 Xamarin.Forms 行为以事件激发时调用命令。

可重用 EffectBehavior

2018/11/2 • • Edit Online

行为是将效果添加到控件时，删除处理代码中的代码隐藏文件的样板效果很有用的方法。本文演示如何创建和使用 Xamarin.Forms 行为添加到控件的效果。

概述

`EffectBehavior` 类是一个可重用的 Xamarin.Forms 自定义行为，将添加 `Effect` 实例与控件时的行为附加到控件，并且删除 `Effect` 实例时的行为是从控件中分离。

以下行为属性必须设置为使用行为：

- 组– 的值 `ResolutionGroupName` 影响类的属性。
- **名称**– 的值 `ExportEffect` 影响类的属性。

有关效果的详细信息，请参阅效果。

> **NOTE**
> `EffectBehavior` 是一个自定义类，可以位于效果行为示例，并不是 Xamarin.Forms 的一部分。

创建行为

`EffectBehavior` 类派生自 `Behavior<T>` 类，其中 `T` 是 `View` 。这意味着，`EffectBehavior` 类可以附加到任何 Xamarin.Forms 控件。

实现可绑定属性

`EffectBehavior` 类定义了两个 `BindableProperty` 实例，用于添加 `Effect` 到控件时行为附加到控件。这些属性下面的代码示例所示：

```
public class EffectBehavior : Behavior<View>
{
  public static readonly BindableProperty GroupProperty =
    BindableProperty.Create ("Group", typeof(string), typeof(EffectBehavior), null);
  public static readonly BindableProperty NameProperty =
    BindableProperty.Create ("Name", typeof(string), typeof(EffectBehavior), null);

  public string Group {
    get { return (string)GetValue (GroupProperty); }
    set { SetValue (GroupProperty, value); }
  }

  public string Name {
    get { return(string)GetValue (NameProperty); }
    set { SetValue (NameProperty, value); }
  }
  ...
}
```

当 `EffectBehavior` 用 `Group` 属性应设置为的值 `ResolutionGroupName` 效果的属性。此外，`Name` 属性应设置为值 `ExportEffect` 影响类的属性。

实现重写

`EffectBehavior` 类将重写 `OnAttachedTo` 并 `OnDetachingFrom` 方法 `Behavior<T>` 类，如以下代码所示示例：

```
public class EffectBehavior : Behavior<View>
{
  ...
  protected override void OnAttachedTo (BindableObject bindable)
  {
    base.OnAttachedTo (bindable);
    AddEffect (bindable as View);
  }

  protected override void OnDetachingFrom (BindableObject bindable)
  {
    RemoveEffect (bindable as View);
    base.OnDetachingFrom (bindable);
  }
  ...
}
```

`OnAttachedTo` 方法执行安装程序通过调用 `AddEffect` 方法，在附加的控件作为参数传递。`OnDetachingFrom` 方法通过调用执行清理 `RemoveEffect` 方法，在附加的控件作为参数传递。

实现行为功能

行为的目的是要添加 `Effect` 中定义 `Group` 并 `Name` 属性，以控制时的行为附加到控件，并且删除 `Effect` 时的行为是从控件中分离。核心行为功能下面的代码示例所示：

```
public class EffectBehavior : Behavior<View>
{
  ...
  void AddEffect (View view)
  {
    var effect = GetEffect ();
    if (effect != null) {
      view.Effects.Add (GetEffect ());
    }
  }

  void RemoveEffect (View view)
  {
    var effect = GetEffect ();
    if (effect != null) {
      view.Effects.Remove (GetEffect ());
    }
  }

  Effect GetEffect ()
  {
    if (!string.IsNullOrWhiteSpace (Group) && !string.IsNullOrWhiteSpace (Name)) {
      return Effect.Resolve (string.Format ("{0}.{1}", Group, Name));
    }
    return null;
  }
}
```

`AddEffect` 方法执行响应 `EffectBehavior` 附加到一个控件，并接收作为参数的附加的控件。该方法然后将检索到的效果添加到控件的 `Effects` 集合。`RemoveEffect` 方法执行响应 `EffectBehavior` 正在分离从一个控件，并接收作为参数的附加的控件。方法然后从控件的删除效果 `Effects` 集合。

`GetEffect` 方法使用 `Effect.Resolve` 方法来检索 `Effect` 。效果是通过串联所在 `Group` 和 `Name` 属性值。如果一个平台不提供效果 `Effect.Resolve` 方法将返回一个非 `null` 值。

使用行为

`EffectBehavior`类可以附加到 `Behaviors` 集合的一个控件，如以下 XAML 代码示例所示：

```
<Label Text="Label Shadow Effect" ...>
  <Label.Behaviors>
    <local:EffectBehavior Group="Xamarin" Name="LabelShadowEffect" />
  </Label.Behaviors>
</Label>
```

以下代码示例显示相应的 C# 代码：

```
var label = new Label {
  Text = "Label Shadow Effect",
  ...
};
label.Behaviors.Add (new EffectBehavior {
  Group = "Xamarin",
  Name = "LabelShadowEffect"
});
```

`Group`并`Name`行为的属性设置为的值 `ResolutionGroupName` 并 `ExportEffect` 中每个特定于平台的影响类的属性项目。

在运行时，行为附加到时 `Label` 控件，`Xamarin.LabelShadowEffect`将添加到控件的 `Effects` 集合。这会导致要添加到显示的文本阴影`Label`控制，如以下屏幕截图中所示：

使用此行为来添加和删除效果从控件的优点是样板效果处理代码可从代码隐藏文件。

总结

本文演示了使用行为添加到控件的效果。`EffectBehavior`类是一个可重用的 Xamarin.Forms 自定义行为，将添加 `Effect` 实例与控件时的行为附加到控件，并且删除`Effect`实例时的行为是从控件中分离。

相关链接

- 效果
- 影响行为（示例）
- 行为
- 行为

可重用 EventToCommandBehavior

2018/11/13 • • Edit Online

行为可用于将命令与不设计与命令进行交互的控件相关联。本文演示如何创建和使用 Xamarin.Forms 行为以事件激发时调用命令。

概述

`EventToCommandBehavior` 类是在响应中执行的命令可重复使用 Xamarin.Forms 自定义行为*任何*事件触发。默认情况下，事件参数的事件将传递到该命令，并且可以根据需要通过转换 `IValueConverter` 实现。

以下行为属性必须设置为使用行为：

- **EventName** – 行为侦听事件的名称。
- **命令**– `ICommand` 要执行。行为期望找到 `ICommand` 实例上 `BindingContext` 可能继承自父元素的附加控件。

此外可以设置以下可选行为属性：

- **CommandParameter** – `object` ，将传递到该命令。
- **转换器**– `IValueConverter` 实现，它将更改的事件参数数据的格式，如之间传递*源*和*目标*由绑定引擎。

> **NOTE**
> `EventToCommandBehavior` 是一个自定义类，可以位于EventToCommand 行为示例，并不是 Xamarin.Forms 的一部分。

创建行为

`EventToCommandBehavior` 类派生自 `BehaviorBase<T>` 类，后者又派生 `Behavior<T>` 类。目的 `BehaviorBase<T>` 类是为需要的任何 Xamarin.Forms 行为提供基类 `BindingContext` 设置为附加的控件的行为。这可确保该行为可以绑定到或执行 `ICommand` 指定的 `Command` 属性时使用行为。

`BehaviorBase<T>` 类提供了可重写 `OnAttachedTo` 方法以设置 `BindingContext` 的行为，并且可重写 `OnDetachingFrom` 方法清除的 `BindingContext` 。此外，类存储中的附加控件的引用 `AssociatedObject` 属性。

实现可绑定属性

`EventToCommandBehavior` 类定义了四个 `BindableProperty` 情况下，执行用户定义事件激发时的命令。这些属性下面的代码示例所示：

```
public class EventToCommandBehavior : BehaviorBase<View>
{
  public static readonly BindableProperty EventNameProperty =
    BindableProperty.Create ("EventName", typeof(string), typeof(EventToCommandBehavior), null,
propertyChanged: OnEventNameChanged);
  public static readonly BindableProperty CommandProperty =
    BindableProperty.Create ("Command", typeof(ICommand), typeof(EventToCommandBehavior), null);
  public static readonly BindableProperty CommandParameterProperty =
    BindableProperty.Create ("CommandParameter", typeof(object), typeof(EventToCommandBehavior), null);
  public static readonly BindableProperty InputConverterProperty =
    BindableProperty.Create ("Converter", typeof(IValueConverter), typeof(EventToCommandBehavior), null);

  public string EventName { ... }
  public ICommand Command { ... }
  public object CommandParameter { ... }
  public IValueConverter Converter { ...  }
  ...
}
```

当`EventToCommandBehavior`用类`Command`属性应为数据绑定到`ICommand`执行响应事件激发中定义的`EventName`属性。该行为将想要查找`ICommand`上`BindingContext`附加的控件。

默认情况下，该事件的事件参数将传递给命令。此数据可以根据需要转换之间传递*源*并*目标*由绑定引擎使用，通过指定`IValueConverter`实现作为`Converter`属性值。或者，参数可以传递给该命令通过指定`CommandParameter`属性值。

实现重写

`EventToCommandBehavior`类将重写`OnAttachedTo`并`OnDetachingFrom`方法的`BehaviorBase<T>`类，如下面的代码示例中所示：

```
public class EventToCommandBehavior : BehaviorBase<View>
{
  ...
  protected override void OnAttachedTo (View bindable)
  {
    base.OnAttachedTo (bindable);
    RegisterEvent (EventName);
  }

  protected override void OnDetachingFrom (View bindable)
  {
    DeregisterEvent (EventName);
    base.OnDetachingFrom (bindable);
  }
  ...
}
```

`OnAttachedTo`方法执行安装程序通过调用`RegisterEvent`方法，传递的值中`EventName`属性作为参数。
`OnDetachingFrom`方法通过调用执行清理`DeregisterEvent`方法，传递的值中`EventName`属性作为参数。

实现行为功能

行为的目的是执行命令定义的`Command`属性中定义的事件触发响应`EventName`属性。核心行为功能下面的代码示例所示：

```
public class EventToCommandBehavior : BehaviorBase<View>
{
  ...
  void RegisterEvent (string name)
  {
    if (string.IsNullOrWhiteSpace (name)) {
      return;
    }

    EventInfo eventInfo = AssociatedObject.GetType ().GetRuntimeEvent (name);
    if (eventInfo == null) {
      throw new ArgumentException (string.Format ("EventToCommandBehavior: Can't register the '{0}' event.",
EventName));
    }
    MethodInfo methodInfo = typeof(EventToCommandBehavior).GetTypeInfo ().GetDeclaredMethod ("OnEvent");
    eventHandler = methodInfo.CreateDelegate (eventInfo.EventHandlerType, this);
    eventInfo.AddEventHandler (AssociatedObject, eventHandler);
  }

  void OnEvent (object sender, object eventArgs)
  {
    if (Command == null) {
      return;
    }

    object resolvedParameter;
    if (CommandParameter != null) {
      resolvedParameter = CommandParameter;
    } else if (Converter != null) {
      resolvedParameter = Converter.Convert (eventArgs, typeof(object), null, null);
    } else {
      resolvedParameter = eventArgs;
    }

    if (Command.CanExecute (resolvedParameter)) {
      Command.Execute (resolvedParameter);
    }
  }
  ...
}
```

`RegisterEvent` 方法执行以响应 `EventToCommandBehavior` 要附加到控件，并且它收到的值 `EventName` 属性作为参数。然后，该方法尝试找到中定义的事件 `EventName` 属性，可在附加的控件。前提是该事件可定位，请 `OnEvent` 方法注册为事件处理程序方法。

`OnEvent` 以响应事件激发中定义的执行方法 `EventName` 属性。前提 `Command` 属性引用的有效 `ICommand`，该方法尝试检索参数要传递给 `ICommand`，如下所示：

- 如果 `CommandParameter` 属性定义了一个参数，它就会检索。
- 否则为如果 `Converter` 属性定义 `IValueConverter` 实现，该转换器执行，并将事件参数数据转换之间传递*源并目标*由绑定引擎。
- 否则，事件自变量被认为是该参数。

数据绑定 `ICommand` 然后执行，则将在参数中传递给提供程序的命令 `CanExecute` 方法将返回 `true`。

尽管没有显示在这里，但是 `EventToCommandBehavior` 还包括 `DeregisterEvent` 方法，则由 `OnDetachingFrom` 方法。`DeregisterEvent` 方法用于查找并取消注册中定义的事件 `EventName` 属性中，清除任何潜在的内存泄漏。

使用行为

`EventToCommandBehavior` 类可以附加到 `Behaviors` 集合的一个控件，如以下 XAML 代码示例所示：

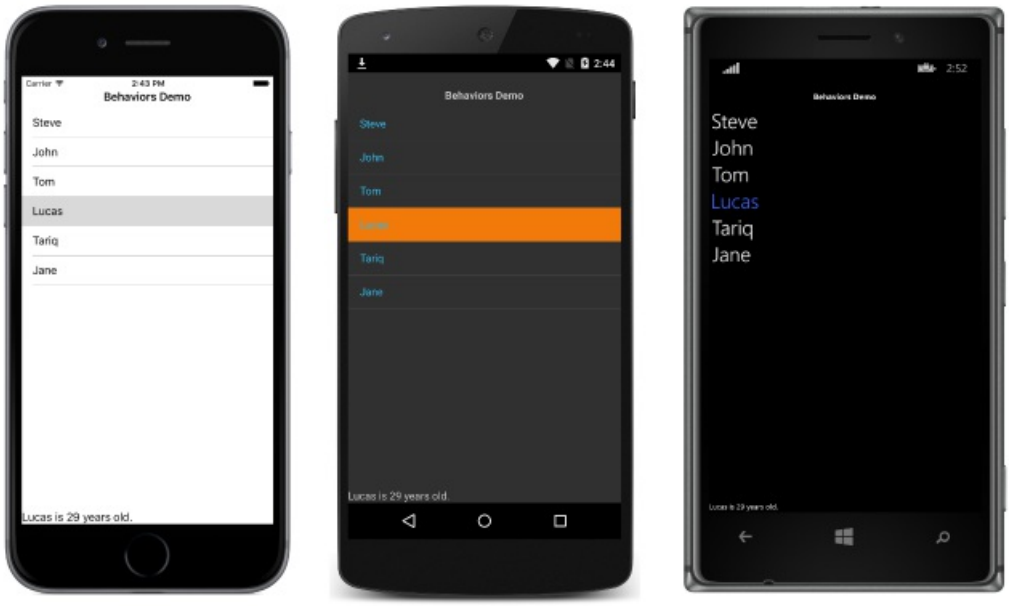
```
<ListView ItemsSource="{Binding People}">
    <ListView.ItemTemplate>
        <DataTemplate>
            <TextCell Text="{Binding Name}" />
        </DataTemplate>
    </ListView.ItemTemplate>
    <ListView.Behaviors>
        <local:EventToCommandBehavior EventName="ItemSelected" Command="{Binding OutputAgeCommand}"
Converter="{StaticResource SelectedItemConverter}" />
    </ListView.Behaviors>
</ListView>
<Label Text="{Binding SelectedItemText}" />
```

以下代码示例显示相应的 C# 代码：

```
var listView = new ListView();
listView.SetBinding(ItemsView<Cell>.ItemsSourceProperty, "People");
listView.ItemTemplate = new DataTemplate(() =>
{
    var textCell = new TextCell();
    textCell.SetBinding(TextCell.TextProperty, "Name");
    return textCell;
});
listView.Behaviors.Add(new EventToCommandBehavior
{
    EventName = "ItemSelected",
    Command = ((HomePageViewModel)BindingContext).OutputAgeCommand,
    Converter = new SelectedItemEventArgsToSelectedItemConverter()
});

var selectedItemLabel = new Label();
selectedItemLabel.SetBinding(Label.TextProperty, "SelectedItemText");
```

`Command` 行为的属性被数据绑定到 `OutputAgeCommand` 属性关联的 ViewModel，而 `Converter` 属性设置为 `SelectedItemConverter` 实例，它将返回 `SelectedItem` 的 `ListView` 从 `SelectedItemChangedEventArgs` 。

在运行时，行为将响应与控件交互。中选择一项时 `ListView` ，则 `ItemSelected` 会触发事件，其将执行 `OutputAgeCommand` ViewModel 中。这又更新 ViewModel `SelectedItemText` 属性的 `Label` 将绑定到，如以下屏幕截图中所示：



使用此行为时触发事件时，执行命令的优点是命令可以与没有专门用于使用命令进行交互的控件相关联。此外，这将从代码隐藏文件移除样板事件处理代码。

总结

本文演示了使用 Xamarin.Forms 行为事件激发时调用命令。行为可用于将命令与不设计与命令进行交互的控件相关联。

相关链接

- EventToCommand 行为（示例）
- 行为
- 行为<T>

Xamarin.Forms 自定义呈现器

2018/7/13 • • Edit Online

使用目标平台，允许保留每个平台相应的外观和感觉的 Xamarin.Forms 应用程序的本机控件呈现 Xamarin.Forms 的用户界面。自定义呈现器让开发人员重写此过程以自定义外观和行为的每个平台上的 Xamarin.Forms 控件。

自定义呈现器简介

自定义呈现器为自定义外观和行为的 Xamarin.Forms 控件提供一种有效方法。它们可用于小型样式更改或复杂的特定于平台的布局和行为自定义。本文介绍了自定义呈现器，并概述了用于创建自定义呈现器的过程。

呈现器基类和本机控件

每个 Xamarin.Forms 控件已创建的本机控件实例的每个平台随附的呈现器。本文列出了呈现器和实现每个 Xamarin.Forms 页面、布局、视图和单元格的本机控件类。

自定义项

Xamarin.Forms `Entry` 控件允许单个行的文本进行编辑。本文演示如何创建自定义呈现器 `Entry` 控件，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。

自定义 ContentPage

一个 `ContentPage` 是一个可显示一个视图，并占据屏幕的大部分可见元素。本文演示如何创建自定义呈现器 `ContentPage` 页上，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。

自定义地图

Xamarin.Forms.Maps 提供跨平台抽象，用于显示可使用本机 Api 每个平台上，以提供快速、熟悉的映射体验的地图进行用户的映射。本主题演示如何创建自定义呈现器为 `Map` 控件，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。

自定义 ListView

Xamarin.Forms `ListView` 是视图，以垂直列表的形式显示数据的集合。本文演示如何创建自定义呈现器，用于封装特定于平台的列表控件和本机单元格的布局，允许更好地控制本机列表控制性能。

自定义 ViewCell

Xamarin.Forms `ViewCell` 是可以添加到一个单元格 `ListView` 或者 `TableView`，其中包含开发人员定义的视图。本文演示如何创建自定义呈现器 `ViewCell` 位于 Xamarin.Forms `ListView` 控件。这会阻止被重复调用期间的 Xamarin.Forms 布局计算 `ListView` 滚动。

实现视图

Xamarin.Forms 自定义用户界面控件应派生自 `View` 类，该类用于将布局和屏幕上的控件。本文演示如何创建 Xamarin.Forms 自定义控件用于显示来自设备的摄像机的预览视频流的自定义呈现器。

实现 HybridWebView

本文演示如何创建自定义呈现器 `HybridWebView` 自定义控件，该示例演示了如何增强特定于平台的 web 控件，若要从 JavaScript 允许 C# 代码调用。

实现视频播放器

本文介绍如何编写呈现器来实现一个自定义 `VideoPlayer` 可以播放视频从 web、作为应用程序资源嵌入的视频或视频存储在用户的设备上的视频库中的控件。演示了几种方法，包括实施方法和只读的可绑定属性。

相关链接

- 效果
- 自定义呈现器（Xamarin University 视频）
- 自定义呈现器（Xamarin University 视频）示例

自定义呈现器简介

2018/10/26 • • Edit Online

自定义呈现器为自定义外观和行为的 Xamarin.Forms 控件提供一种有效方法。它们可用于小型样式更改或复杂的特定于平台的布局和行为自定义。本文介绍了自定义呈现器，并概述了用于创建自定义呈现器的过程。

Xamarin.Forms页面、布局和控件提供常见的 API，用于描述跨平台移动用户界面。每个页面、布局和控件的呈现不同，每个平台上使用 `Renderer` 类，该类又创建本机控件（对应于 Xamarin.Forms 表示形式），对其进行排列在屏幕上，并将添加中指定的行为共享的代码。

开发人员可以实现自定义 `Renderer` 类，以自定义控件的外观和/或行为。为给定类型的自定义呈现器可以添加到一个应用程序项目，以自定义控件在一个位置，同时允许在其他平台; 上的默认行为或不同的自定义呈现器可以添加到 iOS、Android 和通用 Windows 平台 (UWP) 上创建不同的外观和感觉的每个应用程序项目。但是，实现执行简单控件自定义的自定义呈现器类通常是重型的响应。效果简化此过程中，并通常用于较小的样式更改。有关详细信息，请参阅效果。

正在检查为何自定义呈现器有必要

更改 Xamarin.Forms 控件的外观，而无需使用自定义呈现器，是涉及到创建自定义控件子类化，然后使用自定义控件代替原始控件通过一个两步过程。下面的代码示例显示了子类化的示例 `Entry` 控件：

```
public class MyEntry : Entry
{
  public MyEntry ()
  {
    BackgroundColor = Color.Gray;
  }
}
```

`MyEntry` 控件是 `Entry` 控制在何处 `BackgroundColor` 设置为灰色，并可以通过声明其位置的命名空间和控件元素上使用的命名空间前缀在 Xaml 中引用。下面的代码示例演示如何 `MyEntry` 自定义控件可供 `ContentPage`：

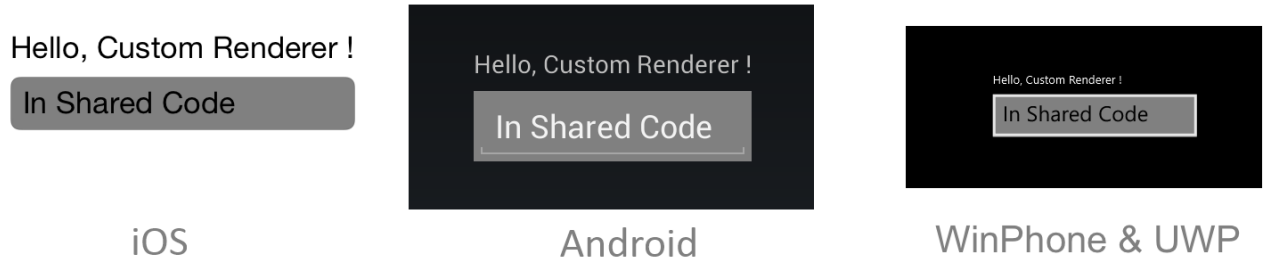
```
<ContentPage
    ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```

`local` 命名空间前缀可以是任何内容。但是，`namespace` 和 `assembly` 值必须匹配的自定义控件的详细信息。一旦声明的命名空间，前缀用于引用自定义控件。

NOTE

定义 `xmlns` 比要简单得在.NET Standard 类库项目中共享的项目。.NET Standard 库编译为程序集，因此很容易地确定什么 `assembly=CustomRenderer` 值应为。（包括 XAML）的所有共享的资源时使用共享项目，将编译到每个引用的项目，这意味着，如果 iOS、Android 和 UWP 项目具有其自己*程序集名称*就无法写入 `xmlns` 声明由于需要为每个应用程序不同的值。共享的项目的 XAML 中的自定义控件需要用相同的程序集名称来配置每个应用程序项目。

`MyEntry` 自定义控件然后呈现在每个平台，具有灰色背景上，如以下屏幕截图中所示：



仅通过将控件子类化已经完成更改每个平台上的控件的背景色。但是，这种技术中因为不能充分利用特定于平台的增强功能和自定义它可以实现限制。在需要时，必须实现自定义呈现器。

创建自定义呈现器类

创建自定义呈现器类的过程如下所示：

1. 创建呈现本机控件呈现器类的子类。
2. 重写呈现本机控件的方法并编写逻辑以自定义控件。通常情况下，`OnElementChanged` 方法用于呈现本机控件，创建相应的 Xamarin.Forms 控件时调用。
3. 添加 `ExportRenderer` 到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 控件属性。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
> 对于大多数 Xamarin.Forms 元素，它是可选提供每个平台项目中的自定义呈现器。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。但是，自定义呈现器呈现时所需的每个平台项目中视图或ViewCell元素。

本系列中的主题将提供演示并解释了此过程的不同的 Xamarin.Forms 元素。

疑难解答

如果自定义控件包含在.NET Standard 库项目已添加到解决方案（即不是.NET 标准库由 Visual Studio for Mac/Visual Studio Xamarin.Forms 应用程序项目模板创建），异常可能会发生在 iOS 中时尝试访问自定义控件。如果发生此问题，可以通过创建从自定义控件的引用来解决 `AppDelegate` 类：

```
var temp = new ClassInPCL(); // in AppDelegate, but temp not used anywhere
```

这将强制使编译器可以识别 `ClassInPCL` 通过解析它的类型。或者，`Preserve` 可以将属性添加到 `AppDelegate` 类来实现相同的结果：

```
[assembly: Preserve (typeof (ClassInPCL))]
```

这将创建对引用 `ClassInPCL` 指示它已在运行时所需的类型。有关详细信息，请参阅保留代码。

总结

本文提供了自定义呈现器的简介，并概述了在创建自定义呈现器的过程。自定义呈现器为自定义外观和行为的 Xamarin.Forms 控件提供一种有效方法。它们可用于小型样式更改或复杂的特定于平台的布局和行为自定义。

相关链接

- 效果

呈现器基类和本机控件

2018/7/13 • • Edit Online

每个 Xamarin.Forms 控件已创建的本机控件实例的每个平台随附的呈现器。本文列出了呈现器和实现每个 Xamarin.Forms 页面、布局、视图和单元格的本机控件类。

除 `MapRenderer` 类，可以在以下命名空间中找到特定于平台的呈现器：

- **iOS** – Xamarin.Forms.Platform.iOS
- **Android** – Xamarin.Forms.Platform.Android
- **Android (AppCompat)** – Xamarin.Forms.Platform.Android.AppCompat
- **通用 Windows 平台 (UWP)** – Xamarin.Forms.Platform.UWP

`MapRenderer` 类可在以下命名空间：

- **iOS** – Xamarin.Forms.Maps.iOS
- **Android** – Xamarin.Forms.Maps.Android
- **通用 Windows 平台 (UWP)** – Xamarin.Forms.Maps.UWP

页数

下表列出的呈现器和本机控件类的实现每个 Xamarin.Forms页类型：

页	呈现器	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
`ContentPage`	PageRenderer	UIViewController	视图分组		FrameworkElement
`MasterDetailPage`	PhoneMasterDetailRenderer (iOS – 电话)、TabletMasterDetailPageRenderer (iOS – 平板电脑)、MasterDetailRenderer (Android)、MasterDetailPageRenderer (Android AppCompat)、MasterDetailPageRenderer (UWP)	UIViewController (Phone) UISplitViewController（平板电脑）	DrawerLayout (v4)	DrawerLayout (v4)	FrameworkElement（自定义控件）
`NavigationPage`	NavigationRenderer（iOS 和 Android）、NavigationPageRenderer (Android AppCompat)、NavigationPageRenderer (UWP)	UIToolbar	视图分组	视图分组	FrameworkElement（自定义控件）

页	呈现器	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
`TabbedPage`	TabbedRenderer（iOS 和 Android）、TabbedPageRenderer (Android AppCompat)、TabbedPageRenderer (UWP)	UIView	ViewPager	ViewPager	FrameworkElement（转动）
`TemplatedPage`	PageRenderer	UIViewController	视图分组		FrameworkElement
`CarouselPage`	CarouselPageRenderer	UIScrollView	ViewPager	ViewPager	FrameworkElement (FlipView)

布局

下表列出的呈现器和本机控件类的实现每个 Xamarin.Forms布局类型：

布局	呈现器	IOS	ANDROID	UWP
`ContentPresenter`	ViewRenderer	UIView	视图	FrameworkElement
`ContentView`	ViewRenderer	UIView	视图	FrameworkElement
`Frame`	FrameRenderer	UIView	视图分组	Border
`ScrollView`	ScrollViewRenderer	UIScrollView	ScrollView	ScrollViewer
`TemplatedView`	ViewRenderer	UIView	视图	FrameworkElement
`AbsoluteLayout`	ViewRenderer	UIView	视图	FrameworkElement
`Grid`	ViewRenderer	UIView	视图	FrameworkElement
`RelativeLayout`	ViewRenderer	UIView	视图	FrameworkElement
`StackLayout`	ViewRenderer	UIView	视图	FrameworkElement

视图

下表列出的呈现器和本机控件类的实现每个 Xamarin.Forms视图类型：

视图	呈现器	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
`ActivityIndicator`	ActivityIndicatorRenderer	UIActivityIndicator	ProgressBar		ProgressBar

视图	呈现器	IOS	ANDROID	ANDROID (APPCOMPAT)	UWP
`BoxView`	BoxRenderer（iOS 和 Android）、BoxViewRenderer (UWP)	UIView	视图分组		矩形
`Button`	ButtonRenderer	UIButton	Button	AppCompatButton	Button
`DatePicker`	DatePickerRenderer	UITextField	EditText		DatePicker
`Editor`	EditorRenderer	UITextView	EditText		文本框
`Entry`	EntryRenderer	UITextField	EditText		文本框
`Image`	ImageRenderer	UIImageView	ImageView		图像
`Label`	LabelRenderer	UILabel	TextView		TextBlock
`ListView`	ListViewRenderer	UITableView	ListView		ListView
`Map`	MapRenderer	MKMapView	MapView		MapControl
`Picker`	PickerRenderer	UITextField	EditText	EditText	组合框
`ProgressBar`	ProgressBarRenderer	UIProgressView	ProgressBar		ProgressBar
`SearchBar`	SearchBarRenderer	UISearchBar	程序标签		AutoSuggestBox
`Slider`	SliderRenderer	UISlider	也		Slider
`Stepper`	StepperRenderer	UIStepper	LinearLayout		控件
`Switch`	SwitchRenderer	UISwitch	开关	SwitchCompat	ToggleSwitch
`TableView`	TableViewRenderer	UITableView	ListView		ListView
`TimePicker`	TimePickerRenderer	UITextField	EditText		TimePicker
`WebView`	WebViewRenderer	UIWebView	Web 视图		Web 视图

单元格

下表列出的呈现器和本机控件类的实现每个 Xamarin.Forms单元格类型：

单元格	呈现器	IOS	ANDROID	UWP
`EntryCell`	EntryCellRenderer	使用 UITextField UITableViewCell	TextView 和 EditText LinearLayout	带有文本框的 DataTemplate
`SwitchCell`	SwitchCellRenderer	使用 UISwitch UITableViewCell	开关	使用网格包含 TextBlock 和 ToggleSwitch 的 DataTemplate
`TextCell`	TextCellRenderer	UITableViewCell	使用两个 TextViews LinearLayout	使用包含两个 Textblock StackPanel 的 DataTemplate
`ImageCell`	ImageCellRenderer	使用 UIImage UITableViewCell	使用两个 TextViews 和 ImageView LinearLayout	使用网格包含图像和两个 Textblock 的 DataTemplate
`ViewCell`	ViewCellRenderer	UITableViewCell	视图	用 ContentPresenter 的 DataTemplate

总结

本文已列出的呈现器和本机控件类的实现每个 Xamarin.Forms 页面、布局、视图和单元格。每个 Xamarin.Forms 控件已创建的本机控件实例的每个平台随附的呈现器。

相关链接

- 自定义呈现器（Xamarin University 视频）

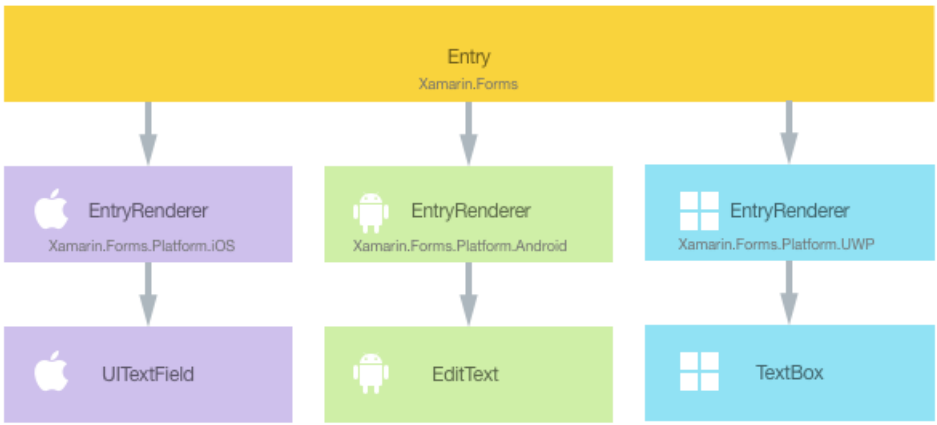
自定义项

2018/7/13 • • Edit Online

Xamarin.Forms 条目控制允许单个行的文本进行编辑。本文演示如何创建自定义呈现器对于项控件，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。

每个 Xamarin.Forms 控件已创建的本机控件实例的每个平台随附的呈现器。当 `Entry` Xamarin.Forms 应用程序，在 iOS 中呈现控件 `EntryRenderer` 类实例化时，后者又在实例化本机 `UITextField` 控件。在 Android 平台上 `EntryRenderer` 类实例化 `EditText` 控件。在通用 Windows 平台 (UWP)，`EntryRenderer` 类实例化 `TextBox` 控件。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `Entry` 控制和相应的本机控件可实现它：



渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的自定义 `Entry` 每个平台上的控件。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义控件。
2. 使用Xamarin.Forms 中的自定义控件。
3. 创建每个平台上的控件的自定义呈现器。

每个项将现在讨论反过来，实现 `Entry` 控件的每个平台有不同的背景色。

创建自定义项控件

自定义 `Entry` 控制可以创建通过子类化 `Entry` 控制，如下面的代码示例中所示：

```
public class MyEntry : Entry
{
}
```

`MyEntry` 控件中创建.NET Standard 库项目，只需 `Entry` 控件。自定义控件将执行中的自定义呈现器，因此在不需要进行任何其他实现 `MyEntry` 控件。

使用自定义控件

`MyEntry` 控件可以在 XAML 中引用.NET Standard 库项目通过声明其位置的命名空间和控件元素上使用的命名空间前缀。下面的代码示例演示如何将 `MyEntry` 控件可供 XAML 页：

```
<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <local:MyEntry Text="In Shared Code" />
    ...
</ContentPage>
```

`local`命名空间前缀可以命名任何内容。但是，`clr-namespace`和`assembly`值必须匹配的自定义控件的详细信息。一旦声明的命名空间前缀用于引用自定义控件。

下面的代码示例演示如何将`MyEntry`控件可供 C# 页：

```
public class MainPage : ContentPage
{
  public MainPage ()
  {
    Content = new StackLayout {
      Children = {
        new Label {
          Text = "Hello, Custom Renderer !",
        },
        new MyEntry {
          Text = "In Shared Code",
        }
      },
      VerticalOptions = LayoutOptions.CenterAndExpand,
      HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
  }
}
```

此代码实例化新`ContentPage`对象，它将显示`Label`和`MyEntry`控件居中垂直和水平页上。

自定义呈现器现在可以添加到每个平台上的控件的外观进行自定义的每个应用程序项目。

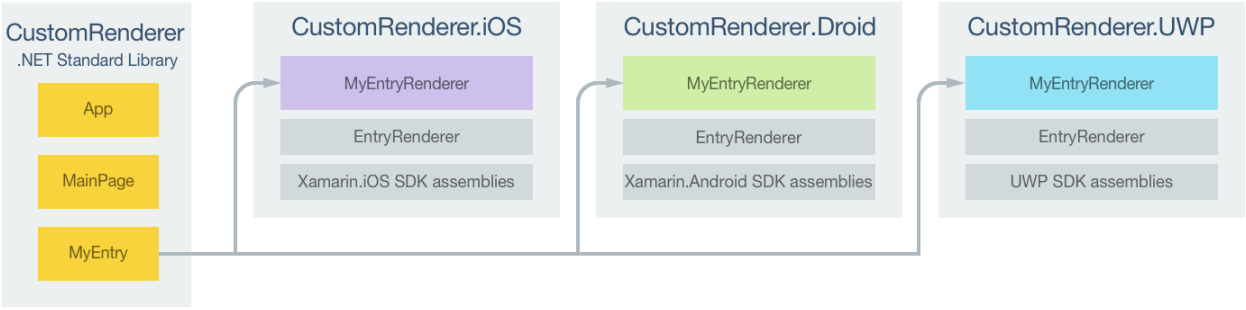
在每个平台上创建自定义呈现器

创建自定义呈现器类的过程如下所示：

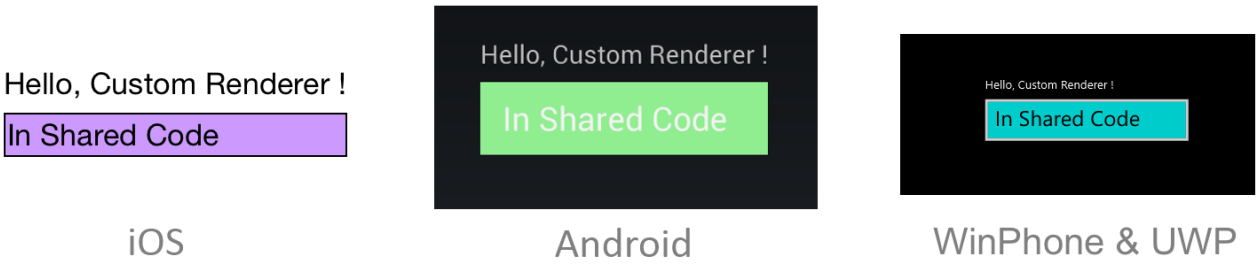
1. 创建一个子类`EntryRenderer`呈现本机控件的类。
2. 重写`OnElementChanged`呈现本机控件和写入逻辑自定义控件的方法。创建相应的 Xamarin.Forms 控件时，调用此方法。
3. 添加`ExportRenderer`到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 控件属性。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
> 若要提供每个平台项目中的自定义呈现器可以选择它。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`MyEntry` 由特定于平台的呈现控件 `MyEntryRenderer` 类，它们都派生自 `EntryRenderer` 为每个平台的类。这会导致每个 `MyEntry` 控制呈现具有特定于平台的背景色，如以下屏幕截图中所示：



`EntryRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 控件以呈现相应的本机控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序 `OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `MyEntry` 控件。

重写的版本 `OnElementChanged` 中的方法 `MyEntryRenderer` 类是执行的本机控件自定义的位置。可以通过访问对正在使用在平台上的本机控件的类型化的引用 `Control` 属性。此外，通过获取对所呈现的 Xamarin.Forms 控件的引用 `Element` 属性，尽管在示例应用程序不使用它。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 控件的类型名称和自定义呈现器的类型名称。 `assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论的每个特定于平台的实现 `MyEntryRenderer` 自定义呈现器类。

在 ios 设备上创建自定义呈现器

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer (typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.iOS
{
    public class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged (e);

            if (Control != null) {
                // do whatever you want to the UITextField here!
                Control.BackgroundColor = UIColor.FromRGB (204, 153, 255);
                Control.BorderStyle = UITextBorderStyle.Line;
            }
        }
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 iOS `UITextField` 控件，与分配给呈现器的控件的引用 `Control` 属性。然后，将背景色设置为与淡紫色 `UIColor.FromRGB` 方法。

在 Android 上创建自定义呈现器

下面的代码示例显示了 Android 平台的自定义呈现器：

```
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.Android
{
    class MyEntryRenderer : EntryRenderer
    {
        public MyEntryRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);

            if (Control != null)
            {
                Control.SetBackgroundColor(global::Android.Graphics.Color.LightGreen);
            }
        }
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 Android `EditText` 控件，与分配给呈现器的控件的引用 `Control` 属性。然后，将背景色设置为与浅绿色 `Control.SetBackgroundColor` 方法。

在 UWP 上创建自定义呈现器

下面的代码示例显示了适用于 UWP 的自定义呈现器：

```
[assembly: ExportRenderer(typeof(MyEntry), typeof(MyEntryRenderer))]
namespace CustomRenderer.UWP
{
    public class MyEntryRenderer : EntryRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Entry> e)
        {
            base.OnElementChanged(e);

            if (Control != null)
            {
                Control.Background = new SolidColorBrush(Colors.Cyan);
            }
        }
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 `TextBox` 控件，与分配给呈现器的控件的引用 `Control` 属性。背景色设置为青色通过创建 `SolidColorBrush` 实例。

总结

本文演示了如何创建自定义控件呈现器适用于 Xamarin.Forms `Entry` 控件，使开发人员能够重写默认本机呈现具有其自己特定于平台的呈现。自定义呈现器提供自定义的 Xamarin.Forms 控件外观的强大方法。它们可用于小型样式更改或复杂的特定于平台的布局和行为自定义。

相关链接

- CustomRendererEntry（示例）

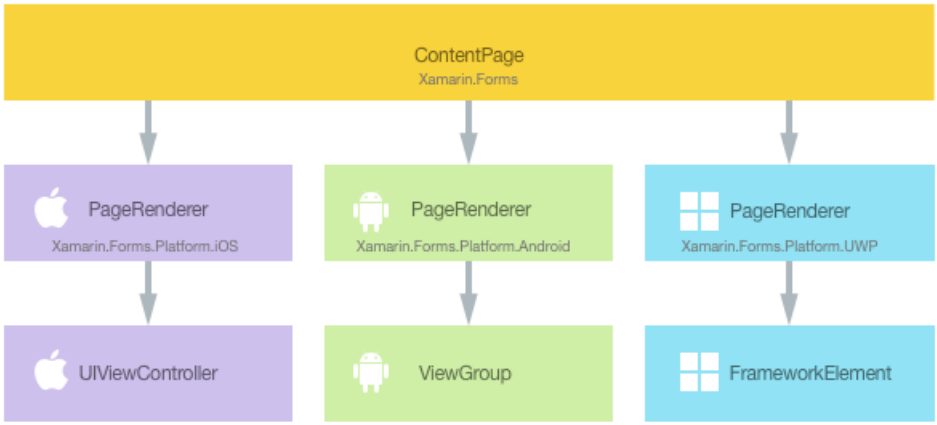
自定义 ContentPage

2018/9/1 • • Edit Online

内容页是显示一个视图并占据屏幕的大部分的可视元素。本文演示如何创建自定义呈现器对于 ContentPage 页中，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。

每个 Xamarin.Forms 控件已创建的本机控件实例的每个平台随附的呈现器。当 `ContentPage` Xamarin.Forms 应用程序，在 iOS 中呈现 `PageRenderer` 类实例化时，这反过来实例化本机 `UIViewController` 控件。在 Android 平台上 `PageRenderer` 类实例化 `ViewGroup` 控件。在通用 Windows 平台 (UWP)，`PageRenderer` 类实例化 `FrameworkElement` 控件。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `ContentPage` 和相应的本机控件实现它：



渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的自定义 `ContentPage` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 页面。
2. 使用Xamarin.Forms 页。
3. 创建每个平台上的页的自定义呈现器。

每个项将现在讨论反过来，若要实现 `CameraPage`，它提供了实时源的相机捕获照片的功能。

创建 Xamarin.Forms 页面

未更改 `ContentPage` 可以添加到共享的 Xamarin.Forms 项目中，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="CustomRenderer.CameraPage">
    <ContentPage.Content>
    </ContentPage.Content>
</ContentPage>
```

同样，有关的代码隐藏文件 `ContentPage` 还应保持不变的如下面的代码示例中所示：

```
public partial class CameraPage : ContentPage
{
    public CameraPage ()
    {
        // A custom renderer is used to display the camera UI
        InitializeComponent ();
    }
}
```

下面的代码示例演示如何在 C# 中创建页面：

```
public class CameraPageCS : ContentPage
{
    public CameraPageCS ()
    {
    }
}
```

实例 `CameraPage` 将用于显示实时相机源每个平台上。自定义控件将执行中的自定义呈现器，因此在不需要进行任何其他实现 `CameraPage` 类。

使用 Xamarin.Forms 页面

空 `CameraPage` Xamarin.Forms 应用程序必须显示。上的按钮时将发生这种情况 `MainPage` 点击实例，它将依次执行 `OnTakePhotoButtonClicked` 方法，如下面的代码示例中所示：

```
async void OnTakePhotoButtonClicked (object sender, EventArgs e)
{
    await Navigation.PushAsync (new CameraPage ());
}
```

此代码只需导航到 `CameraPage`，哪些自定义呈现器将自定义每个平台上的页面的外观。

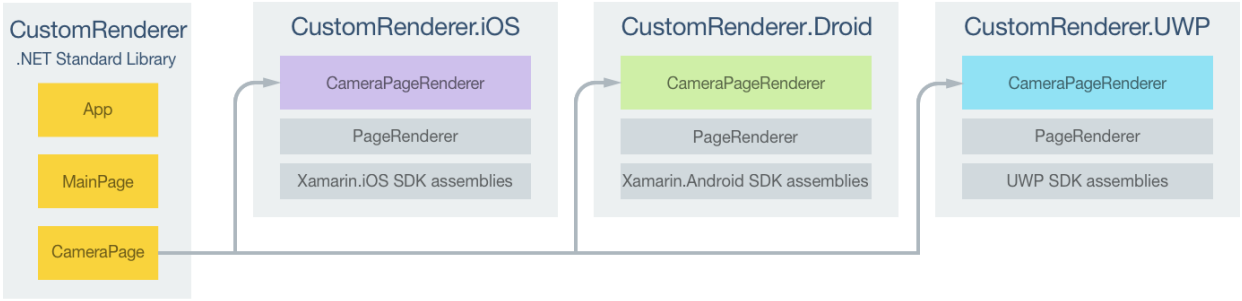
在每个平台上创建页的呈现器

创建自定义呈现器类的过程如下所示：

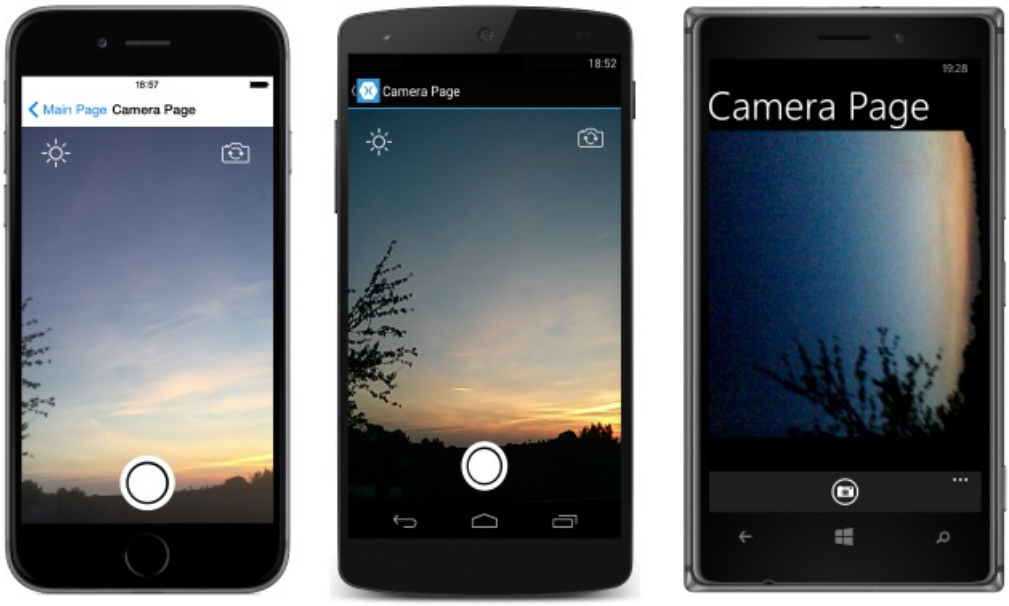
1. 创建一个子类 `PageRenderer` 类。
2. 重写 `OnElementChanged` 呈现的本机的页和写入逻辑，以自定义页面的方法。 `OnElementChanged` 创建相应的 Xamarin.Forms 控件时调用方法。
3. 添加 `ExportRenderer` 属性到页呈现器类，以指定它将用于呈现 Xamarin.Forms 页面。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
> 若要提供每个平台项目中的页呈现器可以选择它。如果未注册页的呈现器，则将使用默认的呈现器的页。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`CameraPage` 实例由特定于平台的呈现 `CameraPageRenderer` 类，它们都派生自 `PageRenderer` 该平台的类。这会导致每个 `CameraPage` 实例呈现使用实时相机源，如以下屏幕截图中所示：



`PageRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 页呈现相应的本机控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序 `OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `CameraPage` 实例。

重写的版本 `OnElementChanged` 中的方法 `CameraPageRenderer` 类是执行本机页自定义的位置。可以通过获取对所呈现的 Xamarin.Forms 页实例的引用 `Element` 属性。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 页的类型名称和自定义呈现器的类型名称。 `assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论的实现 `CameraPageRenderer` 为每个平台的自定义呈现器。

在 ios 设备上创建页的呈现器

下面的代码示例显示了为 iOS 平台页的呈现器：

```
[assembly:ExportRenderer (typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.iOS
{
    public class CameraPageRenderer : PageRenderer
    {
        ...

        protected override void OnElementChanged (VisualElementChangedEventArgs e)
        {
            base.OnElementChanged (e);

            if (e.OldElement != null || Element == null) {
                return;
            }

            try {
                SetupUserInterface ();
                SetupEventHandlers ();
                SetupLiveCameraStream ();
                AuthorizeCameraUse ();
            } catch (Exception ex) {
                System.Diagnostics.Debug.WriteLine (@"            ERROR: ", ex.Message);
            }
        }
        ...
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 iOS `UIViewController` 控件。前提被呈现器不已附加到现有的 Xamarin.Forms 元素，且前提是存在的页实例，所呈现的自定义呈现器仅呈现实时照相机流。

通过一系列的使用方法然后自定义页面 `AVCapture` Api 以提供来自照相机和捕获照片的功能的实时流。

创建在 Android 上的页的呈现器

下面的代码示例显示了 Android 平台的页的呈现器：

```
[assembly: ExportRenderer(typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.Droid
{
    public class CameraPageRenderer : PageRenderer, TextureView.ISurfaceTextureListener
    {
        ...
        public CameraPageRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Page> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            try
            {
                SetupUserInterface();
                SetupEventHandlers();
                AddView(view);
            }
            catch (Exception ex)
            {
                System.Diagnostics.Debug.WriteLine(@"            ERROR: ", ex.Message);
            }
        }
        ...
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 Android `ViewGroup` 控件，它是一组视图。前提被呈现器不已附加到现有的 Xamarin.Forms 元素，且前提是存在的页实例，所呈现的自定义呈现器仅呈现实时照相机流。

然后通过调用一系列的方法使用的自定义页面 `Camera` API 以提供来自照相机和之前捕获照片的功能的实时流 `AddView` 会调用方法来添加实时照相机流式传输到 UI `ViewGroup` 。请注意，在 Android 上它也不必重写 `OnLayout` 方法以执行在视图上的度量值和布局操作。有关详细信息，请参阅ContentPage 呈现器示例。

在 UWP 上创建页的呈现器

下面的代码示例显示了适用于 UWP 的页呈现器：

```
[assembly: ExportRenderer(typeof(CameraPage), typeof(CameraPageRenderer))]
namespace CustomRenderer.UWP
{
    public class CameraPageRenderer : PageRenderer
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.Page> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null || Element == null)
            {
                return;
            }

            try
            {
                ...
                SetupUserInterface();
                SetupBasedOnStateAsync();

                this.Children.Add(page);
            }
            ...
        }

        protected override Size ArrangeOverride(Size finalSize)
        {
            page.Arrange(new Windows.Foundation.Rect(0, 0, finalSize.Width, finalSize.Height));
            return finalSize;
        }
        ...
    }
}
```

对基类的调用 `OnElementChanged` 方法实例化 `FrameworkElement` 在其呈现页的控件。前提被呈现器不已附加到现有的 Xamarin.Forms 元素，且前提是存在的页实例，所呈现的自定义呈现器仅呈现实时照相机流。然后通过调用一系列的方法使用的自定义页面 `MediaCapture` API 以提供来自照相机和自定义的页面添加到之前捕获照片的功能的实时流 `Children` 显示的集合。

实现派生的自定义呈现器时 `PageRenderer` UWP 上，`ArrangeOverride` 方法还应实现来排列页面控件，因为基呈现器不知道应如何处理它们。否则，生成空白页。因此，在此示例 `ArrangeOverride` 方法调用 `Arrange` 方法 `Page` 实例。

> **NOTE**
>
> 若要停止和释放的对象，它提供对 UWP 应用程序中相机的访问至关重要。如果不这样做可能会影响其他应用程序尝试访问设备的照相机。有关详细信息，请参阅显示摄像头预览。

总结

本文演示了如何创建自定义呈现器 `ContentPage` 页上，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。一个 `ContentPage` 是一个可显示一个视图，并占据屏幕的大部分可见元素。

相关链接

- CustomRendererContentPage（示例）

自定义 Xamarin.Forms 映射

2018/6/9 • • Edit Online

Xamarin.Forms.Maps 提供跨平台抽象显示为用户可使用每个平台上的 Api 以提供快速且熟悉映射体验本机地图的地图。

自定义图钉

此文章介绍了如何创建自定义呈现器 `Map` 控件，每个平台将显示自定义的 pin 与 pin 数据的自定义的视图的本机映射。

突出显示地图上的圆形区域

此文章介绍了如何将循环覆盖添加到一个图，以突出显示的地图的圆形区域。

突出显示地图上的区域

此文章介绍了如何添加一个代码图，以突出显示在地图上的区域的多边形覆盖。多边形是闭合的形状并具有其内部填写。

突出显示地图上的路线

此文章介绍了如何在向地图添加折线覆盖。折线覆盖是一系列连接的线条通常用于上一个代码图，显示一个路由或窗体具有必需的任何形状的段。

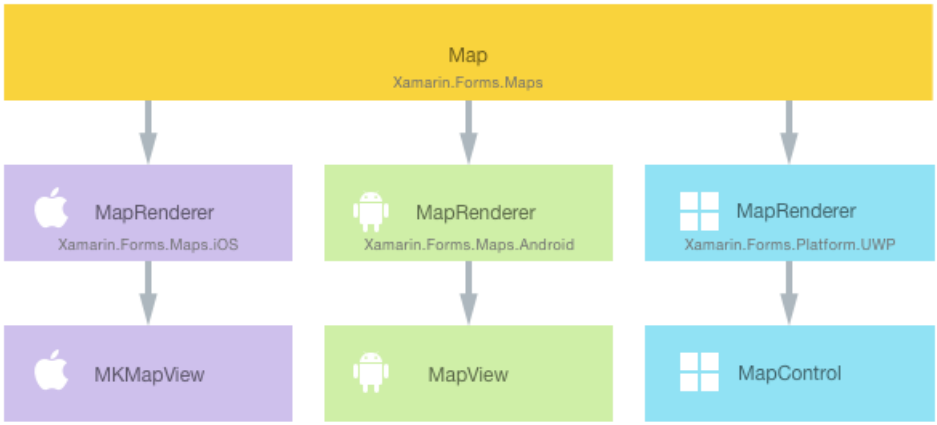
自定义图钉

2018/7/24 • • Edit Online

本文演示如何创建地图控件，每个平台上将显示具有自定义的 pin 和固定数据的自定义的视图的本机映射的自定义呈现器。

每个 Xamarin.Forms 视图都会随附的呈现器为每个平台创建本机控件的实例。当 `Map` Xamarin.Forms 应用程序在 iOS 中，呈现 `MapRenderer` 类实例化时，这反过来实例化本机 `MKMapView` 控件。在 Android 平台上 `MapRenderer` 类实例化本机 `MapView` 控件。在通用 Windows 平台 (UWP)，`MapRenderer` 类实例化本机 `MapControl` 。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `Map` 和相应的本机控件实现它：



呈现过程可用于通过创建自定义呈现器为实现特定于平台的自定义 `Map` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义地图。
2. 使用Xamarin.Forms 中的自定义映射。
3. 创建映射每个平台上的自定义呈现器。

每个项将现在讨论反过来，若要实现 `CustomMap` 呈现器的每个平台上将显示具有自定义的 pin 和固定数据的自定义的视图的本机映射。

> **NOTE**
> `Xamarin.Forms.Maps` 必须初始化和使用之前配置。有关详细信息，请参阅 `Maps Control` 。

创建自定义地图

可以通过子类化创建一个自定义地图控件 `Map` 类，如下面的代码示例中所示：

```
public class CustomMap : Map
{
  public List<CustomPin> CustomPins { get; set; }
}
```

`CustomMap` 控制在.NET Standard 库项目中创建和定义自定义地图的 API。自定义地图公开 `CustomPins` 表示的集合

属性 `CustomPin` 将呈现的每个平台上本机地图控件的对象。 `CustomPin` 类下面的代码示例中所示：

```
public class CustomPin : Pin
{
  public string Url { get; set; }
}
```

此类定义 `CustomPin` 为继承的属性 `Pin` 类，并添加 `Url` 属性。

使用自定义地图

`CustomMap` 控件可以在 XAML 中引用.NET Standard 库项目通过声明其位置的命名空间和自定义地图控件上使用的命名空间前缀。下面的代码示例演示如何将 `CustomMap` 控件可供 XAML 页：

```
<ContentPage ...
             xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer">
    <ContentPage.Content>
        <local:CustomMap x:Name="myMap" MapType="Street"
          WidthRequest="{x:Static local:App.ScreenWidth}"
          HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>
```

`local` 命名空间前缀可以命名任何内容。但是， `clr-namespace` 和 `assembly` 值必须匹配的自定义地图的详细信息。一旦声明的命名空间，前缀用于引用自定义映射。

下面的代码示例演示如何将 `CustomMap` 控件可供 C# 页：

```
public class MapPageCS : ContentPage
{
  public MapPageCS ()
  {
    var customMap = new CustomMap {
      MapType = MapType.Street,
      WidthRequest = App.ScreenWidth,
      HeightRequest = App.ScreenHeight
    };
    ...

    Content = customMap;
  }
}
```

`CustomMap` 实例将用于显示每个平台上的本机映射。它具有 `MapType` 属性设置的显示样式 `Map` ，与可能的值中定义 `MapType` 枚举。对于 iOS 和 Android，宽度和高度的代码图设置的属性通过 `App` 初始化特定于平台的项目中的类。

映射和球瓶的位置它包含，如下面的代码示例中所示进行初始化：

```
public MapPage ()
{
  ...
  var pin = new CustomPin {
    Type = PinType.Place,
    Position = new Position (37.79752, -122.40183),
    Label = "Xamarin San Francisco Office",
    Address = "394 Pacific Ave, San Francisco CA",
    Id = "Xamarin",
    Url = "http://xamarin.com/about/"
  };

  customMap.CustomPins = new List<CustomPin> { pin };
  customMap.Pins.Add (pin);
  customMap.MoveToRegion (MapSpan.FromCenterAndRadius (
    new Position (37.79752, -122.40183), Distance.FromMiles (1.0)));
}
```

此初始化添加自定义的 pin，并将置于具有地图的视图 `MoveToRegion` 方法，以通过创建更改的位置和地图的缩放级别 `MapSpan` 从 `Position` 和一个 `Distance` 。

自定义呈现器现在可以添加到自定义本机地图控件的每个应用程序项目。

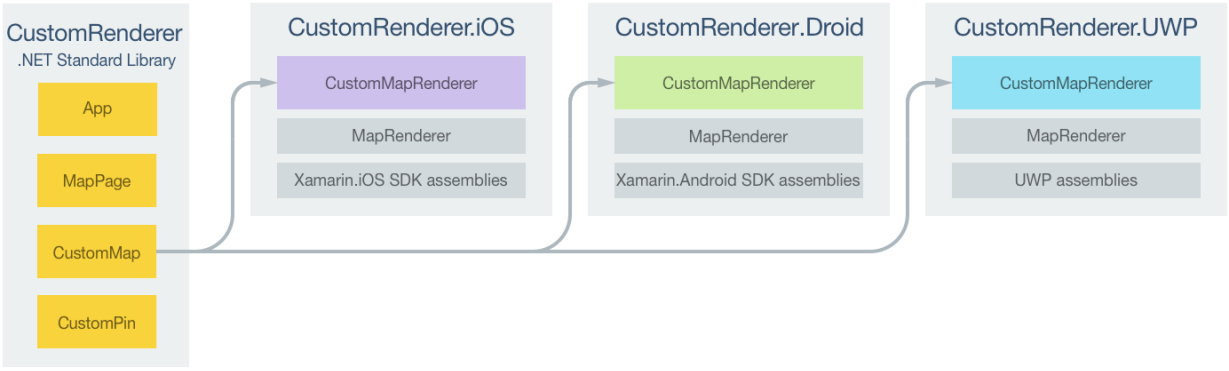
在每个平台上创建自定义呈现器

创建自定义呈现器类的过程如下所示：

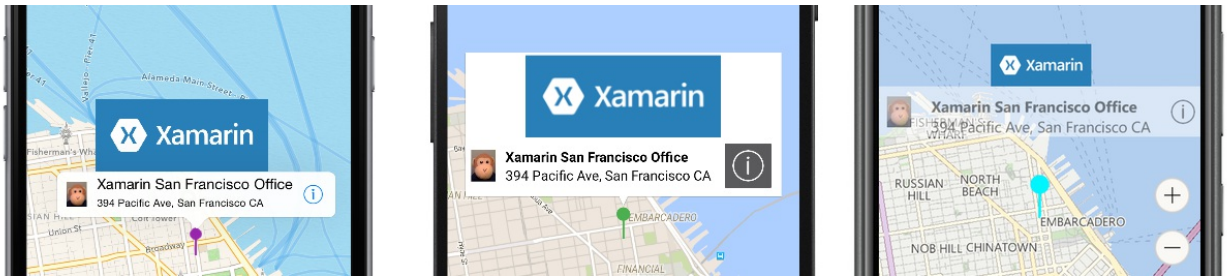
1. 创建一个子类 `MapRenderer` 呈现自定义地图的类。
2. 重写 `OnElementChanged` 呈现的自定义地图和写入逻辑来其进行自定义的方法。创建相应的 Xamarin.Forms 自定义地图时，调用此方法。
3. 添加 `ExportRenderer` 到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 自定义映射特性。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
> 若要提供每个平台项目中的自定义呈现器可以选择它。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`CustomMap` 控件呈现的特定于平台的呈现器类，派生自 `MapRenderer` 为每个平台的类。这会导致每个 `CustomMap` 控制呈现与特定于平台的控件，如以下屏幕截图中所示：



`MapRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 自定义地图来呈现相应的本机控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序 `OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `CustomMap` 实例。

重写的版本 `OnElementChanged` 中每个特定于平台的呈现器类，方法是执行的本机控件自定义的位置。可以通过访问对正在使用在平台上的本机控件的类型化的引用 `Control` 属性。此外，通过获取对所呈现的 Xamarin.Forms 控件的引用 `Element` 属性。

订阅中的事件处理程序时必须小心 `OnElementChanged` 方法，如下面的代码示例中所示：

```
protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
{
  base.OnElementChanged (e);

  if (e.OldElement != null) {
    // Unsubscribe from event handlers
  }

  if (e.NewElement != null) {
    // Configure the native control and subscribe to event handlers
  }
}
```

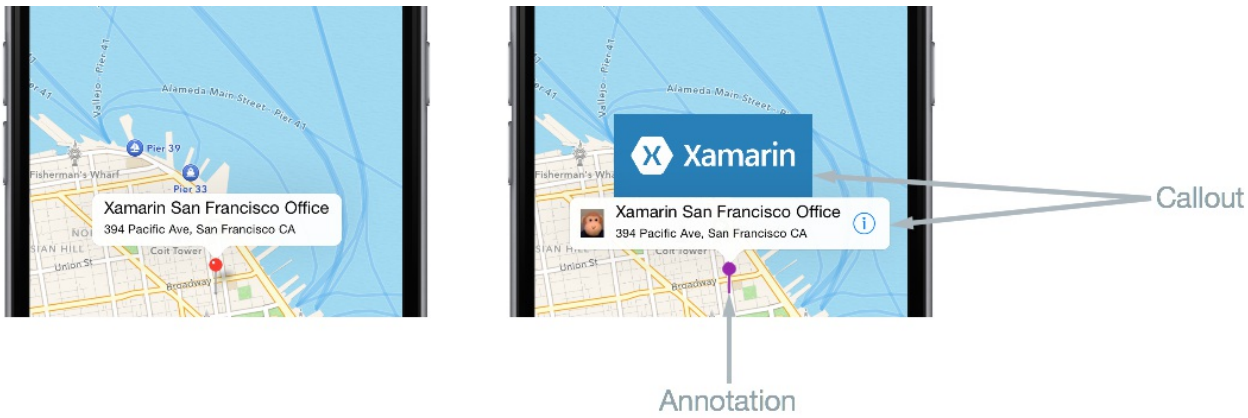
应配置本机控件，并仅在自定义呈现器附加到新 Xamarin.Forms 元素时事件处理程序订阅。同样，任何已订阅的事件处理程序应仅当呈现器附加到的元素发生更改时取消订阅。采用此方法将有助于创建不会遭受内存泄漏的自定义呈现器。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 自定义控件的类型名称和自定义呈现器的类型名称。 `assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论每个特定于平台的自定义呈现器类的实现。

在 ios 设备上创建自定义呈现器

下面的屏幕截图显示地图之前和之后自定义项：



在 iOS 上调用 pin*批注*，可以是自定义映像或不同颜色的系统定义的 pin。可以选择性地显示批注*标注*，其中显示

以响应用户选择批注。显示标注 `Label` 并 `Address` 的属性 `Pin` 具有可选左和右附件视图实例。在上面的屏幕截图，左附件视图的 monkey、图像是右附件视图所*信息*按钮。

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        UIView customPinView;
        List<CustomPin> customPins;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveAnnotations(nativeMap.Annotations);
                    nativeMap.GetViewForAnnotation = null;
                    nativeMap.CalloutAccessoryControlTapped -= OnCalloutAccessoryControlTapped;
                    nativeMap.DidSelectAnnotationView -= OnDidSelectAnnotationView;
                    nativeMap.DidDeselectAnnotationView -= OnDidDeselectAnnotationView;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                customPins = formsMap.CustomPins;

                nativeMap.GetViewForAnnotation = GetViewForAnnotation;
                nativeMap.CalloutAccessoryControlTapped += OnCalloutAccessoryControlTapped;
                nativeMap.DidSelectAnnotationView += OnDidSelectAnnotationView;
                nativeMap.DidDeselectAnnotationView += OnDidDeselectAnnotationView;
            }
        }
        ...
    }
}
```

`OnElementChanged` 方法执行的以下配置 `MKMapView` 实例，前提是自定义呈现器附加到新 Xamarin.Forms 元素：

- `GetViewForAnnotation` 属性设置为 `GetViewForAnnotation` 方法。调用此方法时批注的位置将成为图上可见，和用于自定义批注前显示。
- 事件处理程序 `CalloutAccessoryControlTapped`， `DidSelectAnnotationView`，和 `DidDeselectAnnotationView` 注册事件。这些事件时触发用户点击的标注正确附件，以及何时用户选择和取消选择批注，分别。仅当元素呈现器附加到更改时，已取消订阅事件。

显示批注

`GetViewForAnnotation` 时的批注位置变得可见在映射中，以及用于自定义批注前显示调用方法。批注由两部分组成：

- `MkAnnotation` – 包括标题、副标题和批注的位置。
- `MkAnnotationView` – 包含图像来表示该批注，和（可选）显示了当用户点击批注的标注。

`GetViewForAnnotation` 方法接受 `IMKAnnotation`，其中包含批注的数据，返回 `MKAnnotationView` 显示在映射中，并在下面的代码示例所示：

```
protected override MKAnnotationView GetViewForAnnotation(MKMapView mapView, IMKAnnotation annotation)
{
    MKAnnotationView annotationView = null;

    if (annotation is MKUserLocation)
        return null;

    var customPin = GetCustomPin(annotation as MKPointAnnotation);
    if (customPin == null) {
        throw new Exception("Custom pin not found");
    }

    annotationView = mapView.DequeueReusableAnnotation(customPin.Id.ToString());
    if (annotationView == null) {
        annotationView = new CustomMKAnnotationView(annotation, customPin.Id.ToString());
        annotationView.Image = UIImage.FromFile("pin.png");
        annotationView.CalloutOffset = new CGPoint(0, 0);
        annotationView.LeftCalloutAccessoryView = new UIImageView(UIImage.FromFile("monkey.png"));
        annotationView.RightCalloutAccessoryView = UIButton.FromType(UIButtonType.DetailDisclosure);
        ((CustomMKAnnotationView)annotationView).Id = customPin.Id.ToString();
        ((CustomMKAnnotationView)annotationView).Url = customPin.Url;
    }
    annotationView.CanShowCallout = true;

    return annotationView;
}
```

此方法可确保该批注将显示为自定义映像，而不是作为系统定义的 pin，且点击批注标注会显示包含左侧和右侧的批注标题和地址的其他内容. 实现这一点，如下所示：

1. `GetCustomPin` 调用方法以返回批注的自定义的 pin 数据。
2. 为了节省内存，批注的视图已入池以供重复使用通过调用 `DequeueReusableAnnotation` 。
3. `CustomMKAnnotationView` 类用于扩展 `MKAnnotationView` 类与 `Id` 并 `Url` 对应于中的相同属性的属性 `CustomPin` 实例。新实例 `CustomMKAnnotationView` 创建，前提是该批注是 `null`：
   - `CustomMKAnnotationView.Image` 属性设置为将表示在地图上的批注的图像。
   - `CustomMKAnnotationView.CalloutOffset` 属性设置为 `CGPoint` ，它指定将在标注居中上面批注。
   - `CustomMKAnnotationView.LeftCalloutAccessoryView` 属性设置为批注标题和地址的左侧将出现 monkey 的映像。
   - `CustomMKAnnotationView.RightCalloutAccessoryView` 属性设置为*信息*批注标题和地址右侧将显示的按钮。
   - `CustomMKAnnotationView.Id` 属性设置为 `CustomPin.Id` 属性返回 `GetCustomPin` 方法。这样，使其可以识别的批注可进一步自定义标注，如果所需的。
   - `CustomMKAnnotationView.Url` 属性设置为 `CustomPin.Url` 属性返回 `GetCustomPin` 方法。URL 将导航到时用户点击右标注附件视图中显示的按钮。
4. `MKAnnotationView.CanShowCallout` 属性设置为 `true` 以便标注批注点击时显示。
5. 批注返回要显示在代码图上。

选择批注

当用户点击该批注，`DidSelectAnnotationView` 事件触发时，后者将执行 `OnDidSelectAnnotationView` 方法：

```
void OnDidSelectAnnotationView (object sender, MKAnnotationViewEventArgs e)
{
  var customView = e.View as CustomMKAnnotationView;
  customPinView = new UIView ();

  if (customView.Id == "Xamarin") {
    customPinView.Frame = new CGRect (0, 0, 200, 84);
    var image = new UIImageView (new CGRect (0, 0, 200, 84));
    image.Image = UIImage.FromFile ("xamarin.png");
    customPinView.AddSubview (image);
    customPinView.Center = new CGPoint (0, -(e.View.Frame.Height + 75));
    e.View.AddSubview (customPinView);
  }
}
```

此方法通过添加扩展（其中包含向左和右附件视图）的现有标注 `UIView` 实例与它包含 Xamarin 徽标的图像，前提是所选的批注都有其 `Id` 属性设置为 `Xamarin`。这样可以对不同批注显示不同标注的位置的方案。 `UIView` 将上面的现有标注中间位置显示实例。

点击右标注附件视图

当在用户点击*信息*在右标注附件视图中，按钮 `CalloutAccessoryControlTapped` 事件触发时，后者将执行 `OnCalloutAccessoryControlTapped` 方法：

```
void OnCalloutAccessoryControlTapped (object sender, MKMapViewAccessoryTappedEventArgs e)
{
  var customView = e.View as CustomMKAnnotationView;
  if (!string.IsNullOrWhiteSpace (customView.Url)) {
    UIApplication.SharedApplication.OpenUrl (new Foundation.NSUrl (customView.Url));
  }
}
```

此方法打开 web 浏览器并导航到存储中的地址 `CustomMKAnnotationView.Url` 属性。请注意，在创建时已定义地址 `CustomPin` .NET Standard 库项目中的集合。

取消选中批注

该批注显示并在映射中，在用户点击 `DidDeselectAnnotationView` 事件触发时，后者将执行 `OnDidDeselectAnnotationView` 方法：

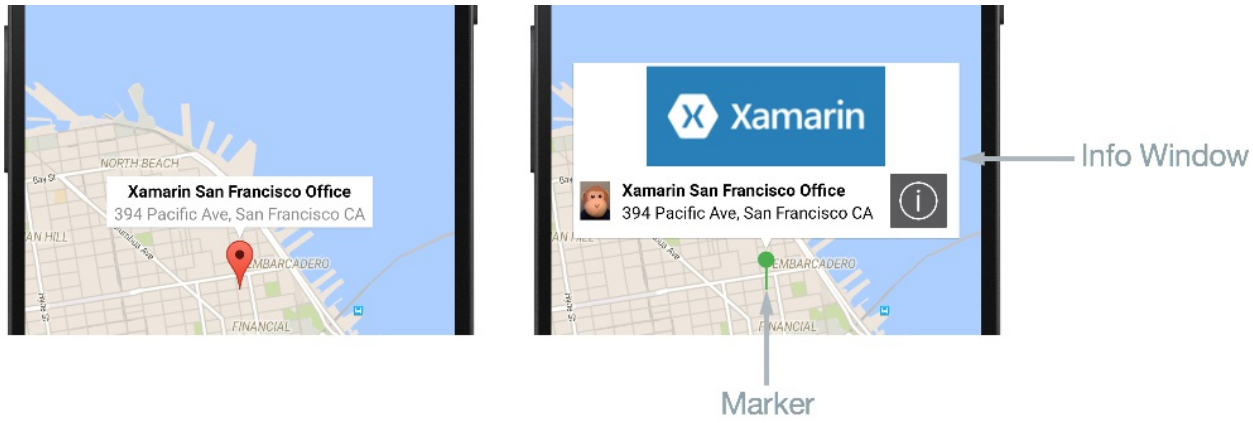
```
void OnDidDeselectAnnotationView (object sender, MKAnnotationViewEventArgs e)
{
  if (!e.View.Selected) {
    customPinView.RemoveFromSuperview ();
    customPinView.Dispose ();
    customPinView = null;
  }
}
```

此方法可确保在未选择现有的标注、标注（Xamarin 徽标的图像）的扩展的部分也将停止正在显示，并将释放其资源时。

有关自定义的详细信息 `MKMapView` 实例，请参阅iOS 地图。

在 Android 上创建自定义呈现器

下面的屏幕截图显示地图之前和之后自定义项：



在 Android 上调用 pin*标记*，并且可以自定义映像或系统定义的各种颜色标记。标记可以显示*信息窗口*，其中显示在用户点击针对标记的响应。信息窗口显示 `Label` 并 `Address` 的属性 `Pin` 实例，并可自定义以包括其他内容。但是，只能有一个信息窗口可以显示在一次。

下面的代码示例显示了 Android 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.Droid
{
    public class CustomMapRenderer : MapRenderer, GoogleMap.IInfoWindowAdapter
    {
        List<CustomPin> customPins;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                NativeMap.InfoWindowClick -= OnInfoWindowClick;
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                customPins = formsMap.CustomPins;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(GoogleMap map)
        {
            base.OnMapReady(map);

            NativeMap.InfoWindowClick += OnInfoWindowClick;
            NativeMap.SetInfoWindowAdapter(this);
        }
        ...
    }
}
```

提供自定义呈现器附加到新 Xamarin.Forms 元素，`OnElementChanged` 方法调用 `MapView.GetMapAsync` 方法，获取基础 `GoogleMap` 与视图。一次 `GoogleMap` 实例不可用，`OnMapReady` 将调用重写。此方法注册的事件处理程序 `InfoWindowClick` 事件，则会激发任务栏信息窗口，并仅当元素呈现器附加到更改时从取消订阅。`OnMapReady` 重写还会调用 `SetInfoWindowAdapter` 方法，以指定的 `CustomMapRenderer` 类实例将提供方法以自定义信息窗口。

`CustomMapRenderer` 类实现 `GoogleMap.IInfoWindowAdapter` 接口自定义的信息窗口。此接口指定必须实现以下方法：

- `public Android.Views.View GetInfoWindow(Marker marker)` – 此方法调用返回一个标记的自定义的信息窗口。如果它返回 `null`，则将使用默认的窗口呈现。如果它返回 `View`，然后，`View` 置于信息窗口框架。
- `public Android.Views.View GetInfoContents(Marker marker)` – 此方法调用以返回 `View` 包含内容的信息窗口中，并将只有当调用 `GetInfoWindow` 方法将返回 `null`。如果它返回 `null`，则将使用的信息窗口内容的默认呈现。

在示例应用程序，自定义信息的窗口内容，因此 `GetInfoWindow` 方法将返回 `null` 要启用此功能。

自定义标记

可以通过调用自定义用于表示标记的图标 `MarkerOptions.SetIcon` 方法。这可以通过重写 `CreateMarker` 方法，为每个调用 `Pin`，添加到映射：

```
protected override MarkerOptions CreateMarker(Pin pin)
{
    var marker = new MarkerOptions();
    marker.SetPosition(new LatLng(pin.Position.Latitude, pin.Position.Longitude));
    marker.SetTitle(pin.Label);
    marker.SetSnippet(pin.Address);
    marker.SetIcon(BitmapDescriptorFactory.FromResource(Resource.Drawable.pin));
    return marker;
}
```

此方法创建一个新 `MarkerOption` 为每个实例 `Pin` 实例。设置位置、标签和标记的地址后，其图标设置为 `SetIcon` 方法。此方法采用 `BitmapDescriptor` 对象，其中包含必要的数据以与呈现图标 `BitmapDescriptorFactory` 类，用于提供帮助器方法来简化创建 `BitmapDescriptor`。

有关使用详细信息 `BitmapDescriptorFactory` 类，以自定义标记，请参阅自定义标记。

自定义信息窗口

在用户点击标记上时 `GetInfoContents` 执行方法时，提供的 `GetInfoWindow` 方法将返回 `null`。下面的代码示例演示 `GetInfoContents` 方法：

```
public Android.Views.View GetInfoContents (Marker marker)
{
  var inflater = Android.App.Application.Context.GetSystemService (Context.LayoutInflaterService) as
Android.Views.LayoutInflater;
  if (inflater != null) {
    Android.Views.View view;

    var customPin = GetCustomPin (marker);
    if (customPin == null) {
      throw new Exception ("Custom pin not found");
    }

    if (customPin.Id.ToString() == "Xamarin") {
      view = inflater.Inflate (Resource.Layout.XamarinMapInfoWindow, null);
    } else {
      view = inflater.Inflate (Resource.Layout.MapInfoWindow, null);
    }

    var infoTitle = view.FindViewById<TextView> (Resource.Id.InfoWindowTitle);
    var infoSubtitle = view.FindViewById<TextView> (Resource.Id.InfoWindowSubtitle);

    if (infoTitle != null) {
      infoTitle.Text = marker.Title;
    }
    if (infoSubtitle != null) {
      infoSubtitle.Text = marker.Snippet;
    }

    return view;
  }
  return null;
}
```

此方法返回 `View` 包含信息窗口的内容。实现这一点，如下所示：

- 一个 `LayoutInflater` 检索实例。这用于实例化到其对应的布局 XML 文件 `View` 。
- `GetCustomPin` 调用方法以返回信息窗口中的自定义的 pin 数据。
- `XamarinMapInfoWindow` 如果被放大布局 `CustomPin.Id` 属性等于 `Xamarin` 。否则为 `MapInfoWindow` 放大布局。这样可以显示为不同的标记的信息不同窗口布局的位置的方案。
- `InfoWindowTitle` 和 `InfoWindowSubtitle` 膨胀的布局中，从检索资源及其 `Text` 属性设置为相应的数据从 `Marker` 实例，前提是不是资源 `null` 。
- `View` 实例返回要显示在代码图上。

> **NOTE**
> 为非活动的信息窗口 `View` 。相反，Android 会将转换 `View` 为静态位图和显示的为图像。这意味着一个 click 事件，它不能响应任何触控事件或手势，并在信息窗口中的各个控件不能响应到其自身可以响应的信息窗口时，单击事件。

单击信息窗口

当用户在信息窗口中，单击 `InfoWindowClick` 事件触发时，后者将执行 `OnInfoWindowClick` 方法：

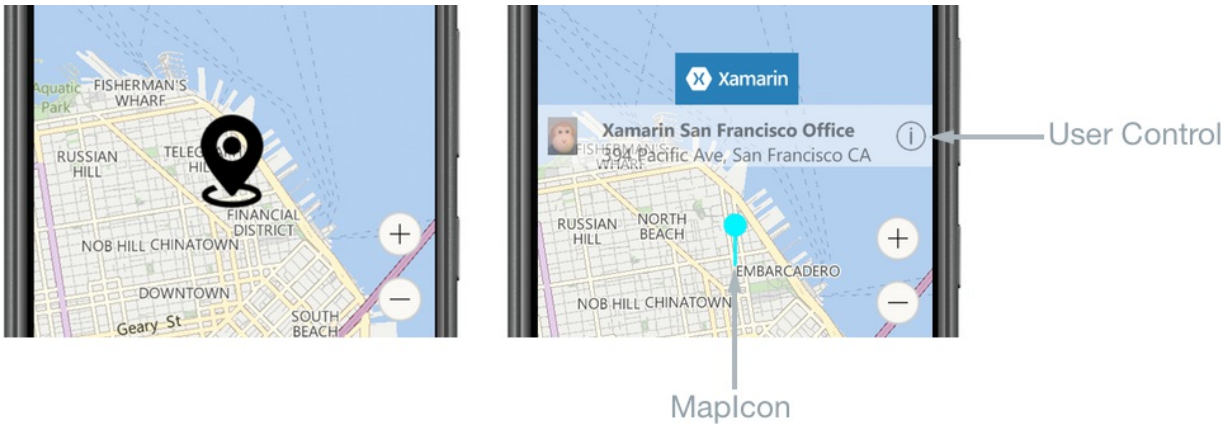
```
void OnInfoWindowClick (object sender, GoogleMap.InfoWindowClickEventArgs e)
{
  var customPin = GetCustomPin (e.Marker);
  if (customPin == null) {
    throw new Exception ("Custom pin not found");
  }

  if (!string.IsNullOrWhiteSpace (customPin.Url)) {
    var url = Android.Net.Uri.Parse (customPin.Url);
    var intent = new Intent (Intent.ActionView, url);
    intent.AddFlags (ActivityFlags.NewTask);
    Android.App.Application.Context.StartActivity (intent);
  }
}
```

此方法打开 web 浏览器并导航到存储中的地址 `Url` 检索到的属性 `CustomPin` 实例 `Marker`。请注意，在创建时已定义地址 `CustomPin` .NET Standard 库项目中的集合。

有关自定义的详细信息 `MapView` 实例，请参阅地图 API。

在通用 Windows 平台上创建自定义呈现器

下面的屏幕截图显示地图之前和之后自定义项：



在 UWP 上调用 pin *结构图图标*，并且可以自定义映像或系统定义的默认图像。图图标可以显示 `UserControl`，以响应用户点击的地图图标上显示。`UserControl` 可以显示任何内容，包括 `Label` 并 `Address` 的属性 `Pin` 实例。

下面的代码示例显示了 UWP 自定义呈现器：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace CustomRenderer.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        MapControl nativeMap;
        List<CustomPin> customPins;
        XamarinMapOverlay mapOverlay;
        bool xamarinOverlayShown = false;

        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                nativeMap.MapElementClick -= OnMapElementClick;
                nativeMap.Children.Clear();
                mapOverlay = null;
                nativeMap = null;
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                nativeMap = Control as MapControl;
                customPins = formsMap.CustomPins;

                nativeMap.Children.Clear();
                nativeMap.MapElementClick += OnMapElementClick;

                foreach (var pin in customPins)
                {
                    var snPosition = new BasicGeoposition { Latitude = pin.Pin.Position.Latitude, Longitude =
pin.Pin.Position.Longitude };
                    var snPoint = new Geopoint(snPosition);

                    var mapIcon = new MapIcon();
                    mapIcon.Image = RandomAccessStreamReference.CreateFromUri(new Uri("ms-appx:///pin.png"));
                    mapIcon.CollisionBehaviorDesired = MapElementCollisionBehavior.RemainVisible;
                    mapIcon.Location = snPoint;
                    mapIcon.NormalizedAnchorPoint = new Windows.Foundation.Point(0.5, 1.0);

                    nativeMap.MapElements.Add(mapIcon);
                }
            }
        }
        ...
    }
}
```

`OnElementChanged` 方法，提供自定义呈现器附加到新 Xamarin.Forms 元素将执行以下操作：

- 它将清除 `MapControl.Children` 若要从映射中，在注册的事件处理程序之前删除任何现有的用户界面元素的集合 `MapElementClick` 事件。当用户在点击或单击时将引发此事件 `MapElement` 上 `MapControl`，并仅当元素呈现器附加到更改时从取消订阅。
- 中的每个 pin `customPins` 集合显示在地图上的正确地理位置，如下所示：
  - Pin 的位置创建为 `Geopoint` 实例。
  - 一个 `MapIcon` 创建实例来表示 pin。
  - 图像用于表示 `MapIcon` 通过设置指定 `MapIcon.Image` 属性。但是，地图图标的图像是不能始终保证要显示，因为它可能会被在地图上其他元素遮挡。因此，映射图标 `CollisionBehaviorDesired` 属性设置为 `MapElementCollisionBehavior.RemainVisible`，以确保它仍显示。

- 位置 `MapIcon` 通过设置指定 `MapIcon.Location` 属性。
- `MapIcon.NormalizedAnchorPoint` 属性设置为将指针悬停在图像的近似位置。如果此属性保持不变的 (0, 0)，它表示图像的左上的角，其默认值中的地图的缩放级别的更改可能会导致指向其他位置的映像。
- `MapIcon` 实例添加到 `MapControl.MapElements` 集合。这会导致上显示的地图图标 `MapControl` 。

> **NOTE**
> 对于多个地图图标，使用相同的映像时 `RandomAccessStreamReference` 为了获得最佳性能，应在页面或应用程序级别声明实例。

显示此用户控件

在用户点击映射图标时 `OnMapElementClick` 执行方法。下面的代码示例显示了此方法：

```
private void OnMapElementClick(MapControl sender, MapElementClickEventArgs args)
{
    var mapIcon = args.MapElements.FirstOrDefault(x => x is MapIcon) as MapIcon;
    if (mapIcon != null)
    {
        if (!xamarinOverlayShown)
        {
            var customPin = GetCustomPin(mapIcon.Location.Position);
            if (customPin == null)
            {
                throw new Exception("Custom pin not found");
            }

            if (customPin.Id.ToString() == "Xamarin")
            {
                if (mapOverlay == null)
                {
                    mapOverlay = new XamarinMapOverlay(customPin);
                }

                var snPosition = new BasicGeoposition { Latitude = customPin.Pin.Position.Latitude, Longitude
= customPin.Pin.Position.Longitude };
                var snPoint = new Geopoint(snPosition);

                nativeMap.Children.Add(mapOverlay);
                MapControl.SetLocation(mapOverlay, snPoint);
                MapControl.SetNormalizedAnchorPoint(mapOverlay, new Windows.Foundation.Point(0.5, 1.0));
                xamarinOverlayShown = true;
            }
        }
        else
        {
            nativeMap.Children.Remove(mapOverlay);
            xamarinOverlayShown = false;
        }
    }
}
```

此方法创建 `UserControl` 显示 pin 有关的信息的实例。实现这一点，如下所示：

- `MapIcon` 检索实例。
- `GetCustomPin` 调用方法以返回要显示的自定义的 pin 数据。
- 一个 `XamarinMapOverlay` 创建实例以显示自定义的 pin 数据。此类是一个用户控件。
- 要显示的地理位置 `XamarinMapOverlay` 实例上 `MapControl` 将创建为 `Geopoint` 实例。
- `XamarinMapOverlay` 实例添加到 `MapControl.Children` 集合。此集合包含将地图显示的 XAML 用户界面元素。
- 地理位置 `XamarinMapOverlay` 通过调用设置在地图上的实例 `SetLocation` 方法。

- 上的相对位置 `XamarinMapOverlay` 实例中，对应于指定位置，将通过调用 `SetNormalizedAnchorPoint` 方法。这可确保，在中的映射结果的缩放级别更改 `XamarinMapOverlay` 实例始终显示在正确位置。

或者，如果已经在地图上显示有关 pin 信息，在地图上的点击中移除 `XamarinMapOverlay` 实例与 `MapControl.Children` 集合。

点击信息按钮

当在用户点击*信息*按钮 `XamarinMapOverlay` 用户控件 `Tapped` 事件触发时，后者将执行 `OnInfoButtonTapped` 方法：

```
private async void OnInfoButtonTapped(object sender, TappedRoutedEventArgs e)
{
    await Launcher.LaunchUriAsync(new Uri(customPin.Url));
}
```

此方法打开 web 浏览器并导航到存储中的地址 `Url` 属性的 `CustomPin` 实例。请注意，在创建时已定义地址 `CustomPin` .NET Standard 库项目中的集合。

有关自定义的详细信息 `MapControl` 实例，请参阅地图和位置概述MSDN 上。

总结

本文演示了如何创建自定义呈现器 `Map` 控件，使开发人员能够重写其自己特定于平台的自定义的默认本机呈现。Xamarin.Forms.Maps 提供跨平台抽象，用于显示可使用本机 Api 每个平台上，以提供快速、熟悉的映射体验的地图进行用户的映射。

相关链接

- 地图控件
- iOS 地图
- 地图 API
- 自定义的 Pin（示例）

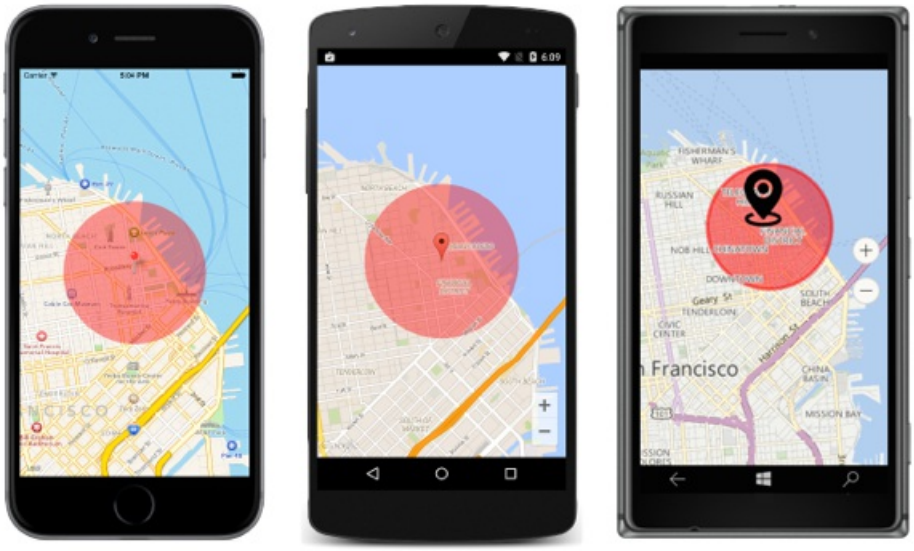
突出显示地图上的圆形区域

2018/7/13 • • Edit Online

本文介绍如何将循环覆盖添加到映射中，以突出显示地图的圆形区域。

概述

覆盖是在地图上的分层的图形。覆盖层支持绘制图形的内容，它被放大随着与该映射。以下屏幕截图显示将循环覆盖添加到映射的结果：



当 `Map` Xamarin.Forms 应用程序，在 iOS 中呈现控件 `MapRenderer` 类实例化时，这反过来实例化本机 `MKMapView` 控件。在 Android 平台上 `MapRenderer` 类实例化本机 `MapView` 控件。在通用 Windows 平台 (UWP)，`MapRenderer` 类实例化本机 `MapControl` 。渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的映射自定义 `Map` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义地图。
2. 使用Xamarin.Forms 中的自定义映射。
3. 自定义通过每个平台上创建代码图的自定义呈现器映射。

> **NOTE**
>
> `Xamarin.Forms.Maps` 必须初始化和使用之前配置。有关详细信息，请参阅 `Maps Control`

有关自定义使用自定义呈现器的映射的信息，请参阅自定义图钉。

创建自定义地图

创建 `CustomCircle` 类具有 `Position` 和 `Radius` 属性：

```
public class CustomCircle
{
  public Position Position { get; set; }
  public double Radius { get; set; }
}
```

然后，创建一个子类 `Map` 类，将类型的属性添加 `CustomCircle` ：

```
public class CustomMap : Map
{
  public CustomCircle Circle { get; set; }
}
```

使用自定义地图

使用`CustomMap`控件通过声明它的实例中的 XAML 页实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
             x:Class="MapOverlay.MapPage">
    <ContentPage.Content>
        <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>
```

或者，使用`CustomMap`控件通过声明它的实例中的 C# 页实例：

```
public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}
```

初始化`CustomMap`控件：

```
public partial class MapPage : ContentPage
{
  public MapPage ()
  {
    ...
    var pin = new Pin {
      Type = PinType.Place,
      Position = new Position (37.79752, -122.40183),
      Label = "Xamarin San Francisco Office",
      Address = "394 Pacific Ave, San Francisco CA"
    };

    var position = new Position (37.79752, -122.40183);
    customMap.Circle = new CustomCircle {
      Position = position,
      Radius = 1000
    };

    customMap.Pins.Add (pin);
    customMap.MoveToRegion (MapSpan.FromCenterAndRadius (position, Distance.FromMiles (1.0)));
  }
}
```

此初始化添加`Pin`并`CustomCircle`实例到自定义映射，并将具有地图的视图`MoveToRegion`方法，以更改的位置和

缩放通过创建映射的级别 `MapSpan` 从 `Position` 和一个 `Distance` 。

自定义地图

自定义呈现器必须现在添加到要向地图添加循环覆盖每个应用程序项目中。

在 ios 设备上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法中添加循环覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKCircleRenderer circleRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    circleRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                var circle = formsMap.Circle;

                nativeMap.OverlayRenderer = GetOverlayRenderer;

                var circleOverlay = MKCircle.Circle(new
CoreLocation.CLLocationCoordinate2D(circle.Position.Latitude, circle.Position.Longitude), circle.Radius);
                nativeMap.AddOverlay(circleOverlay);
            }
        }
        ...
    }
}
```

此方法执行下面的配置，前提是自定义呈现器附加到新 Xamarin.Forms 元素：

- `MKMapView.OverlayRenderer` 属性设置为对应的委托。
- 通过设置静态创建圆形 `MKCircle` 对象，以米为单位指定圆的中心和圆的半径。
- 该圆形被添加到映射通过调用 `MKMapView.AddOverlay` 方法。

然后，实现 `GetOverlayRenderer` 方法以自定义在覆盖区上的呈现：

```
public class CustomMapRenderer : MapRenderer
{
  MKCircleRenderer circleRenderer;
  ...

  MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
  {
      if (circleRenderer == null && !Equals(overlayWrapper, null)) {
          var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
          circleRenderer = new MKCircleRenderer(overlay as MKCircle) {
              FillColor = UIColor.Red,
              Alpha = 0.4f
          };
      }
      return circleRenderer;
  }
}
```

在 Android 上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 和 `OnMapReady` 添加循环覆盖的方法：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        CustomCircle circle;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void
OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Xamarin.Forms.Maps.Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                circle = formsMap.Circle;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var circleOptions = new CircleOptions();
            circleOptions.InvokeCenter(new LatLng(circle.Position.Latitude, circle.Position.Longitude));
            circleOptions.InvokeRadius(circle.Radius);
            circleOptions.InvokeFillColor(0X66FF0000);
            circleOptions.InvokeStrokeColor(0X66FF0000);
            circleOptions.InvokeStrokeWidth(0);

            NativeMap.AddCircle(circleOptions);
        }
    }
}
```

`OnElementChanged` 方法调用 `MapView.GetMapAsync` 方法，获取基础 `GoogleMap` 的，提供自定义呈现器附加到新 Xamarin.Forms 元素绑定到视图。一次 `GoogleMap` 实例不可用，`OnMapReady` 将调用方法，其中通过实例化创建圆形 `CircleOptions` 对象，以米为单位指定圆的中心和圆的半径。该圆形然后通过调用添加到映射 `NativeMap.AddCircle` 方法。

在通用 Windows 平台上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法中添加循环覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        const int EarthRadiusInMeteres = 6371000;

        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }
            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;
                var circle = formsMap.Circle;

                var coordinates = new List<BasicGeoposition>();
                var positions = GenerateCircleCoordinates(circle.Position, circle.Radius);
                foreach (var position in positions)
                {
                    coordinates.Add(new BasicGeoposition { Latitude = position.Latitude, Longitude =
position.Longitude });
                }

                var polygon = new MapPolygon();
                polygon.FillColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeThickness = 5;
                polygon.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polygon);
            }
        }
        // GenerateCircleCoordinates helper method (below)
    }
}
```

提供自定义呈现器附加到新 Xamarin.Forms 元素，此方法将执行以下操作：

- 从检索的圆圈位置和半径 `CustomMap.Circle` 属性，并传递给 `GenerateCircleCoordinates` 方法，后者生成纬度和经度坐标圆圈外围。此帮助器方法的代码如下所示。
- 圆形外围坐标转换为 `List` 的 `BasicGeoposition` 坐标。
- 通过实例化创建圆形 `MapPolygon` 对象。 `MapPolygon` 类用于在地图上显示多点形状，通过设置其 `Path` 属性设置为 `Geopath` 对象，其中包含的形状坐标。
- 将其添加到地图上呈现多边形 `MapControl.MapElements` 集合。

```
List<Position> GenerateCircleCoordinates(Position position, double radius)
{
    double latitude = position.Latitude.ToRadians();
    double longitude = position.Longitude.ToRadians();
    double distance = radius / EarthRadiusInMeteres;
    var positions = new List<Position>();

    for (int angle = 0; angle <=360; angle++)
    {
        double angleInRadians = ((double)angle).ToRadians();
        double latitudeInRadians = Math.Asin(Math.Sin(latitude) * Math.Cos(distance) + Math.Cos(latitude) *
Math.Sin(distance) * Math.Cos(angleInRadians));
        double longitudeInRadians = longitude + Math.Atan2(Math.Sin(angleInRadians) * Math.Sin(distance) *
Math.Cos(latitude), Math.Cos(distance) - Math.Sin(latitude) * Math.Sin(latitudeInRadians));

        var pos = new Position(latitudeInRadians.ToDegrees(), longitudeInRadians.ToDegrees());
        positions.Add(pos);
    }

    return positions;
}
```

总结

本文介绍了如何将循环覆盖添加到映射，以突出显示地图的圆形区域。

相关链接

- 循环映射 Ovlerlay（示例）
- 自定义图钉
- Xamarin.Forms.Maps

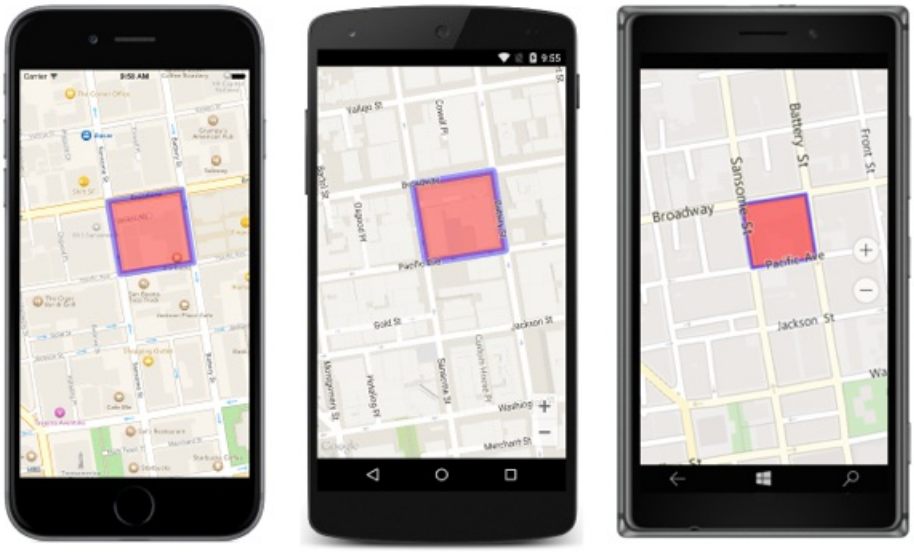
突出显示地图上的区域

2018/7/13 • • Edit Online

本文介绍了如何将多边形覆盖添加到映射，以突出显示地图上的区域。多边形是一个闭合的形状和中填充其内部。

概述

覆盖是在地图上的分层的图形。覆盖层支持绘制图形的内容，它被放大随着与该映射。以下屏幕截图显示向地图添加多边形覆盖的结果：



当 `Map` Xamarin.Forms 应用程序，在 iOS 中呈现控件 `MapRenderer` 类实例化时，这反过来实例化本机 `MKMapView` 控件。在 Android 平台上 `MapRenderer` 类实例化本机 `MapView` 控件。在通用 Windows 平台 (UWP)，`MapRenderer` 类实例化本机 `MapControl` 。渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的映射自定义 `Map` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义地图。
2. 使用Xamarin.Forms 中的自定义映射。
3. 自定义通过每个平台上创建代码图的自定义呈现器映射。

> **NOTE**
>
> `Xamarin.Forms.Maps` 必须初始化和使用之前配置。有关详细信息，请参阅 `Maps Control` 。

有关自定义使用自定义呈现器的映射的信息，请参阅自定义图钉。

创建自定义地图

创建一个子类 `Map` 类，它将 `ShapeCoordinates` 属性：

```
public class CustomMap : Map
{
  public List<Position> ShapeCoordinates { get; set; }

  public CustomMap ()
  {
    ShapeCoordinates = new List<Position> ();
  }
}
```

`ShapeCoordinates` 属性将存储的定义以突出显示的区域的坐标集合。

使用自定义地图

使用 `CustomMap` 控件通过声明它的实例中的 XAML 页实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
             x:Class="MapOverlay.MapPage">
    <ContentPage.Content>
        <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>
```

或者，使用 `CustomMap` 控件通过声明它的实例中的 C# 页实例：

```
public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}
```

初始化 `CustomMap` 控件：

```
public partial class MapPage : ContentPage
{
  public MapPage ()
  {
    ...
    customMap.ShapeCoordinates.Add (new Position (37.797513, -122.402058));
    customMap.ShapeCoordinates.Add (new Position (37.798433, -122.402256));
    customMap.ShapeCoordinates.Add (new Position (37.798582, -122.401071));
    customMap.ShapeCoordinates.Add (new Position (37.797658, -122.400888));

    customMap.MoveToRegion (MapSpan.FromCenterAndRadius (new Position (37.79752, -122.40183),
Distance.FromMiles (0.1)));
  }
}
```

此初始化指定一系列的纬度和经度坐标定义的映射会突出显示的区域。然后将定位具有地图的视图 `MoveToRegion` 方法，以通过创建更改的位置和地图的缩放级别 `MapSpan` 从 `Position` 和一个 `Distance` 。

自定义地图

现在必须为要向地图添加多边形覆盖每个应用程序项目添加自定义呈现器。

在 ios 设备上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法添加多边形覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKPolygonRenderer polygonRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    polygonRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;

                nativeMap.OverlayRenderer = GetOverlayRenderer;

                CLLocationCoordinate2D[] coords = new CLLocationCoordinate2D[formsMap.ShapeCoordinates.Count];

                int index = 0;
                foreach (var position in formsMap.ShapeCoordinates)
                {
                    coords[index] = new CLLocationCoordinate2D(position.Latitude, position.Longitude);
                    index++;
                }

                var blockOverlay = MKPolygon.FromCoordinates(coords);
                nativeMap.AddOverlay(blockOverlay);
            }
        }
        ...
    }
}
```

此方法执行下面的配置，前提是自定义呈现器附加到新 Xamarin.Forms 元素：

- `MKMapView.OverlayRenderer` 属性设置为对应的委托。
- 从纬度和经度坐标的集合中检索 `CustomMap.ShapeCoordinates` 属性和数组的形式存储 `CLLocationCoordinate2D` 实例。
- 通过调用静态创建多边形 `MKPolygon.FromCoordinates` 方法，后者指定的纬度和经度的每个点。
- 多边形被添加到映射通过调用 `MKMapView.AddOverlay` 方法。此方法会自动关闭多边形通过绘制一条连接的第一个和最后一个点线。

然后，实现 `GetOverlayRenderer` 方法以自定义在覆盖区上的呈现：

```
public class CustomMapRenderer : MapRenderer
{
  MKPolygonRenderer polygonRenderer;
  ...

  MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
  {
      if (polygonRenderer == null && !Equals(overlayWrapper, null)) {
          var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
          polygonRenderer = new MKPolygonRenderer(overlay as MKPolygon) {
              FillColor = UIColor.Red,
              StrokeColor = UIColor.Blue,
              Alpha = 0.4f,
              LineWidth = 9
          };
      }
      return polygonRenderer;
  }
}
```

在 Android 上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 和 `OnMapReady` 添加多边形覆盖的方法：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        List<Position> shapeCoordinates;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                shapeCoordinates = formsMap.ShapeCoordinates;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var polygonOptions = new PolygonOptions();
            polygonOptions.InvokeFillColor(0x66FF0000);
            polygonOptions.InvokeStrokeColor(0x660000FF);
            polygonOptions.InvokeStrokeWidth(30.0f);

            foreach (var position in shapeCoordinates)
            {
                polygonOptions.Add(new LatLng(position.Latitude, position.Longitude));
            }
            NativeMap.AddPolygon(polygonOptions);
        }
    }
}
```

`OnElementChanged` 方法检索集合中的纬度和经度坐标 `CustomMap.ShapeCoordinates` 属性并将它们存储在成员变量中。然后，它调用 `MapView.GetMapAsync` 方法，获取基础 `GoogleMap` 的，提供自定义呈现器附加到新 Xamarin.Forms 元素绑定到视图。一次 `GoogleMap` 实例不可用，`OnMapReady` 将调用方法，其中通过实例化创建多边形 `PolygonOptions` 对象，它指定的纬度和经度的每个点。多边形然后通过调用添加到映射 `NativeMap.AddPolygon` 方法。此方法会自动关闭多边形通过绘制一条连接的第一个和最后一个点线。

在通用 Windows 平台上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法添加多边形覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
               // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;

                var coordinates = new List<BasicGeoposition>();
                foreach (var position in formsMap.ShapeCoordinates)
                {
                    coordinates.Add(new BasicGeoposition() { Latitude = position.Latitude, Longitude =
position.Longitude });
                }

                var polygon = new MapPolygon();
                polygon.FillColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polygon.StrokeColor = Windows.UI.Color.FromArgb(128, 0, 0, 255);
                polygon.StrokeThickness = 5;
                polygon.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polygon);
            }
        }
    }
}
```

提供自定义呈现器附加到新 Xamarin.Forms 元素，此方法将执行以下操作：

- 从纬度和经度坐标的集合中检索 `CustomMap.ShapeCoordinates` 属性和转换为 `List` 的 `BasicGeoposition` 坐标。
- 多边形创建的实例化 `MapPolygon` 对象。 `MapPolygon` 类用于在地图上显示多点形状，通过设置其 `Path` 属性设置为 `Geopath` 对象，其中包含的形状坐标。
- 将其添加到地图上呈现多边形 `MapControl.MapElements` 集合。请注意，多边形通过绘制一条连接的第一个和最后一个点线将为自动关闭。

总结

本文介绍了如何将多边形覆盖添加到映射，以突出显示的地图区域。多边形是一个闭合的形状和中填充其内部。

相关链接

- 多边形映射覆盖（示例）
- 自定义图钉
- Xamarin.Forms.Maps

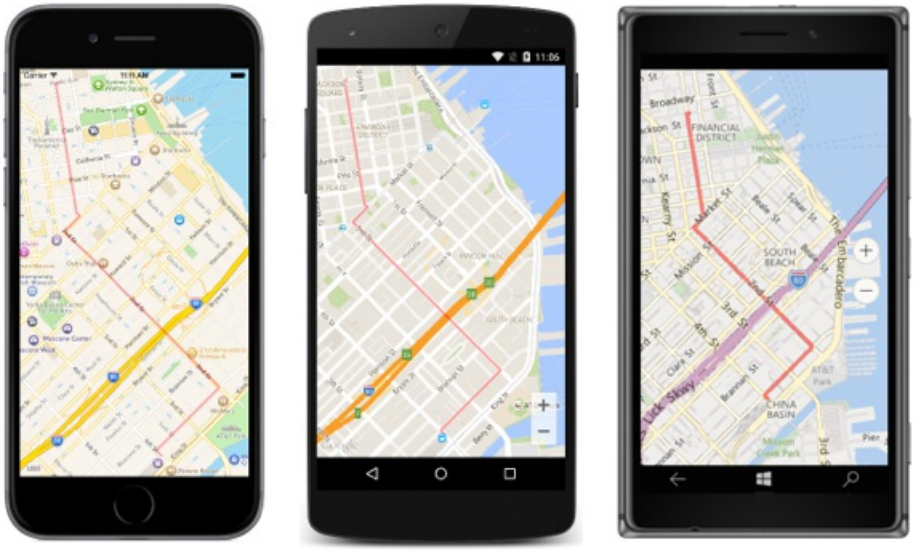
突出显示地图上的路由

2018/7/13 • • Edit Online

本文介绍如何向映射添加 polyline 覆盖。折线覆盖是一系列连接的直线段通常用来在地图上显示路线或窗体所需的任何形状。

概述

覆盖是在地图上的分层的图形。覆盖层支持绘制图形的内容，它被放大随着与该映射。以下屏幕截图显示将折线覆盖添加到映射的结果：



当 `Map` Xamarin.Forms 应用程序，在 iOS 中呈现控件 `MapRenderer` 类实例化时，这反过来实例化本机 `MKMapView` 控件。在 Android 平台上 `MapRenderer` 类实例化本机 `MapView` 控件。在通用 Windows 平台 (UWP)，`MapRenderer` 类实例化本机 `MapControl` 。渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的映射自定义 `Map` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义地图。
2. 使用Xamarin.Forms 中的自定义映射。
3. 自定义通过每个平台上创建代码图的自定义呈现器映射。

> **NOTE**
>
> `Xamarin.Forms.Maps` 必须初始化和使用之前配置。有关详细信息，请参阅 `Maps Control` 。

有关自定义使用自定义呈现器的映射的信息，请参阅自定义图钉。

创建自定义地图

创建一个子类 `Map` 类，它将 `RouteCoordinates` 属性：

```
public class CustomMap : Map
{
  public List<Position> RouteCoordinates { get; set; }

  public CustomMap ()
  {
    RouteCoordinates = new List<Position> ();
  }
}
```

`RouteCoordinates` 属性将存储这些坐标定义的路由被突出显示的集合。

使用自定义地图

使用 `CustomMap` 控件通过声明它的实例中的 XAML 页实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:MapOverlay;assembly=MapOverlay"
             x:Class="MapOverlay.MapPage">
    <ContentPage.Content>
        <local:CustomMap x:Name="customMap" MapType="Street" WidthRequest="{x:Static local:App.ScreenWidth}"
HeightRequest="{x:Static local:App.ScreenHeight}" />
    </ContentPage.Content>
</ContentPage>
```

或者，使用 `CustomMap` 控件通过声明它的实例中的 C# 页实例：

```
public class MapPageCS : ContentPage
{
    public MapPageCS ()
    {
        var customMap = new CustomMap {
            MapType = MapType.Street,
            WidthRequest = App.ScreenWidth,
            HeightRequest = App.ScreenHeight
        };
        ...
        Content = customMap;
    }
}
```

初始化 `CustomMap` 控件：

```
public partial class MapPage : ContentPage
{
  public MapPage ()
  {
    ...
    customMap.RouteCoordinates.Add (new Position (37.785559, -122.396728));
    customMap.RouteCoordinates.Add (new Position (37.780624, -122.390541));
    customMap.RouteCoordinates.Add (new Position (37.777113, -122.394983));
    customMap.RouteCoordinates.Add (new Position (37.776831, -122.394627));

    customMap.MoveToRegion (MapSpan.FromCenterAndRadius (new Position (37.79752, -122.40183),
Distance.FromMiles (1.0)));
  }
}
```

此初始化指定一系列的纬度和经度坐标来突出显示地图上定义的路由。然后将定位具有地图的视图 `MoveToRegion` 方法，以通过创建更改的位置和地图的缩放级别 `MapSpan` 从 `Position` 和一个 `Distance` 。

自定义地图

自定义呈现器必须现在添加到要添加到映射的折线覆盖每个应用程序项目中。

在 ios 设备上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法添加 polyline 覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.iOS
{
    public class CustomMapRenderer : MapRenderer
    {
        MKPolylineRenderer polylineRenderer;

        protected override void OnElementChanged(ElementChangedEventArgs<View> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null) {
                var nativeMap = Control as MKMapView;
                if (nativeMap != null) {
                    nativeMap.RemoveOverlays(nativeMap.Overlays);
                    nativeMap.OverlayRenderer = null;
                    polylineRenderer = null;
                }
            }

            if (e.NewElement != null) {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MKMapView;
                nativeMap.OverlayRenderer = GetOverlayRenderer;

                CLLocationCoordinate2D[] coords = new CLLocationCoordinate2D[formsMap.RouteCoordinates.Count];
                int index = 0;
                foreach (var position in formsMap.RouteCoordinates)
                {
                    coords[index] = new CLLocationCoordinate2D(position.Latitude, position.Longitude);
                    index++;
                }

                var routeOverlay = MKPolyline.FromCoordinates(coords);
                nativeMap.AddOverlay(routeOverlay);
            }
        }
        ...
    }
}
```

此方法执行下面的配置，前提是自定义呈现器附加到新 Xamarin.Forms 元素：

- `MKMapView.OverlayRenderer` 属性设置为对应的委托。
- 从纬度和经度坐标的集合中检索 `CustomMap.RouteCoordinates` 属性和数组的形式存储 `CLLocationCoordinate2D` 实例。
- 通过调用静态创建 polyline `MKPolyline.FromCoordinates` 方法，后者指定的纬度和经度的每个点。
- 线的折线被添加到映射通过调用 `MKMapView.AddOverlay` 方法。

然后，实现 `GetOverlayRenderer` 方法以自定义在覆盖区上的呈现：

```
public class CustomMapRenderer : MapRenderer
{
  MKPolylineRenderer polylineRenderer;
  ...

  MKOverlayRenderer GetOverlayRenderer(MKMapView mapView, IMKOverlay overlayWrapper)
  {
      if (polylineRenderer == null && !Equals(overlayWrapper, null)) {
          var overlay = Runtime.GetNSObject(overlayWrapper.Handle) as IMKOverlay;
          polylineRenderer = new MKPolylineRenderer(overlay as MKPolyline) {
              FillColor = UIColor.Blue,
              StrokeColor = UIColor.Red,
              LineWidth = 3,
              Alpha = 0.4f
          };
      }
      return polylineRenderer;
  }
}
```

在 Android 上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 和 `OnMapReady` 添加 polyline 覆盖的方法：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.Droid
{
    public class CustomMapRenderer : MapRenderer
    {
        List<Position> routeCoordinates;

        public CustomMapRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(Xamarin.Forms.Platform.Android.ElementChangedEventArgs<Map>
e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                routeCoordinates = formsMap.RouteCoordinates;
                Control.GetMapAsync(this);
            }
        }

        protected override void OnMapReady(Android.Gms.Maps.GoogleMap map)
        {
            base.OnMapReady(map);

            var polylineOptions = new PolylineOptions();
            polylineOptions.InvokeColor(0x66FF0000);

            foreach (var position in routeCoordinates)
            {
                polylineOptions.Add(new LatLng(position.Latitude, position.Longitude));
            }

            NativeMap.AddPolyline(polylineOptions);
        }
    }
}
```

`OnElementChanged` 方法检索集合中的纬度和经度坐标 `CustomMap.RouteCoordinates` 属性并将它们存储在成员变量中。然后，它调用 `MapView.GetMapAsync` 方法，获取基础 `GoogleMap` 的，提供自定义呈现器附加到新 Xamarin.Forms 元素绑定到视图。一次 `GoogleMap` 实例不可用，`OnMapReady` 将调用方法，其中通过实例化创建 polyline `PolylineOptions` 对象，它指定的纬度和经度的每个点。线的折线然后通过调用添加到映射 `NativeMap.AddPolyline` 方法。

在通用 Windows 平台上创建自定义呈现器

创建一个子类 `MapRenderer` 类并重写其 `OnElementChanged` 方法添加 polyline 覆盖：

```
[assembly: ExportRenderer(typeof(CustomMap), typeof(CustomMapRenderer))]
namespace MapOverlay.UWP
{
    public class CustomMapRenderer : MapRenderer
    {
        protected override void OnElementChanged(ElementChangedEventArgs<Map> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // Unsubscribe
            }

            if (e.NewElement != null)
            {
                var formsMap = (CustomMap)e.NewElement;
                var nativeMap = Control as MapControl;

                var coordinates = new List<BasicGeoposition>();
                foreach (var position in formsMap.RouteCoordinates)
                {
                    coordinates.Add(new BasicGeoposition() { Latitude = position.Latitude, Longitude =
position.Longitude });
                }

                var polyline = new MapPolyline();
                polyline.StrokeColor = Windows.UI.Color.FromArgb(128, 255, 0, 0);
                polyline.StrokeThickness = 5;
                polyline.Path = new Geopath(coordinates);
                nativeMap.MapElements.Add(polyline);
            }
        }
    }
}
```

提供自定义呈现器附加到新 Xamarin.Forms 元素，此方法将执行以下操作：

- 从纬度和经度坐标的集合中检索 `CustomMap.RouteCoordinates` 属性和转换为 `List` 的 `BasicGeoposition` 坐标。
- 通过实例化创建 polyline `MapPolyline` 对象。 `MapPolygon` 类用于在地图上显示一条线，通过设置其 `Path` 属性设置为 `Geopath` 对象，其中包含的行坐标。
- 将其添加到地图上呈现 polyline `MapControl.MapElements` 集合。

总结

本文介绍了如何将折线覆盖添加到映射中，若要在地图上显示路线或窗体所需的任何形状。

相关链接

- 折线映射 Ovlerlay（示例）
- 自定义图钉
- Xamarin.Forms.Maps

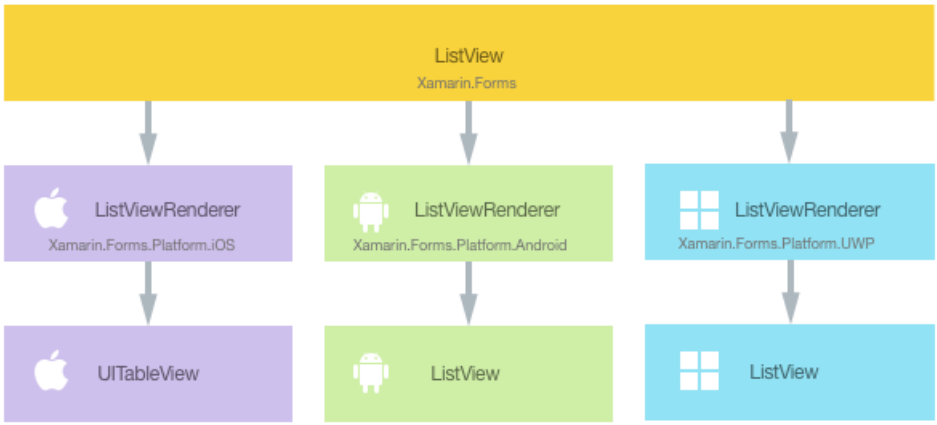
自定义 ListView

2018/7/13 • • Edit Online

Xamarin.Forms ListView 是视图，以垂直列表的形式显示数据的集合。本文演示如何创建自定义呈现器，用于封装特定于平台的列表控件和本机单元格的布局，允许更好地控制本机列表控制性能。

每个 Xamarin.Forms 视图都会随附的呈现器为每个平台创建本机控件的实例。当 `ListView` Xamarin.Forms 应用程序，在 iOS 中呈现 `ListViewRenderer` 类实例化时，这反过来实例化本机 `UITableView` 控件。在 Android 平台上 `ListViewRenderer` 类实例化本机 `ListView` 控件。在通用 Windows 平台 (UWP)，`ListViewRenderer` 类实例化本机 `ListView` 控件。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `ListView` 控制和相应的本机控件可实现它：



渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的自定义 `ListView` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义控件。
2. 使用Xamarin.Forms 中的自定义控件。
3. 创建每个平台上的控件的自定义呈现器。

每个项将现在讨论反过来，若要实现 `NativeListView` 充分利用特定于平台的列表控件和本机单元格布局的呈现器。移植现有本机应用，其中包含列表和可重复使用的单元格代码时，此方案非常有用。此外，它允许详细自定义可能会影响性能，例如数据虚拟化的列表控件功能。

创建自定义 ListView 控件

自定义 `ListView` 控制可以创建通过子类化 `ListView` 类，如下面的代码示例中所示：

```
public class NativeListView : ListView
{
  public static readonly BindableProperty ItemsProperty =
    BindableProperty.Create ("Items", typeof(IEnumerable<DataSource>), typeof(NativeListView), new
List<DataSource> ());

  public IEnumerable<DataSource> Items {
    get { return (IEnumerable<DataSource>)GetValue (ItemsProperty); }
    set { SetValue (ItemsProperty, value); }
  }

  public event EventHandler<SelectedItemChangedEventArgs> ItemSelected;

  public void NotifyItemSelected (object item)
  {
    if (ItemSelected != null) {
      ItemSelected (this, new SelectedItemChangedEventArgs (item));
    }
  }
}
```

`NativeListView` .NET Standard 库项目中创建和定义自定义控件的 API。此控件公开 `Items` 属性，可用于填充 `ListView` 提供数据，并可以是数据绑定到用于显示目的。它还公开了 `ItemSelected` 变化的特定于平台的本机列表控件中选择一项均会触发的事件。若要深入了解数据绑定，请参阅数据绑定基本知识。

使用自定义控件

`NativeListView` 自定义控件，可以引用在 Xaml 中的.NET Standard 库项目中声明其位置的命名空间并在控件上使用的命名空间前缀。下面的代码示例演示如何将 `NativeListView` 自定义控件可供 XAML 页：

```
<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <ContentPage.Content>
          <Grid>
            <Grid.RowDefinitions>
              <RowDefinition Height="Auto"/>
              <RowDefinition Height="*" />
            </Grid.RowDefinitions>
          <Label Text="{x:Static local:App.Description}" HorizontalTextAlignment="Center" />
            <local:NativeListView Grid.Row="1" x:Name="nativeListView" ItemSelected="OnItemSelected"
VerticalOptions="FillAndExpand" />
          </Grid>
      </ContentPage.Content>
</ContentPage>
```

`local` 命名空间前缀可以命名任何内容。但是，`clr-namespace` 和 `assembly` 值必须匹配的自定义控件的详细信息。一旦声明的命名空间，前缀用于引用自定义控件。

下面的代码示例演示如何将 `NativeListView` 自定义控件可供 C# 页：

```
public class MainPageCS : ContentPage
{
    NativeListView nativeListView;

    public MainPageCS()
    {
        nativeListView = new NativeListView
        {
            Items = DataSource.GetList(),
            VerticalOptions = LayoutOptions.FillAndExpand
        };

        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
                Padding = new Thickness(0, 20, 0, 0);
                break;
            case Device.Android:
            case Device.UWP:
                Padding = new Thickness(0);
                break;
        }

        Content = new Grid
        {
            RowDefinitions = {
                new RowDefinition { Height = GridLength.Auto },
                new RowDefinition { Height = new GridLength (1, GridUnitType.Star) }
            },
            Children = {
                new Label { Text = App.Description, HorizontalTextAlignment = TextAlignment.Center },
                nativeListView
            }
        };
        nativeListView.ItemSelected += OnItemSelected;
    }
    ...
}
```

`NativeListView`自定义控件使用特定于平台的自定义呈现器来显示数据，通过填充一系列`Items`属性。在列表中的每一行都包含三个项的数据-名称、类别和图像文件名。在列表中每行的布局定义特定于平台的自定义呈现器。

> **NOTE**
>
> 因为`NativeListView`可使用特定于平台的列表控件包含滚动功能的呈现自定义控件、自定义控件应该不托管在可滚动的布局控件等`ScrollView`。

自定义呈现器现在可以添加到每个应用程序项目来创建特定于平台的列表控件和本机单元格布局。

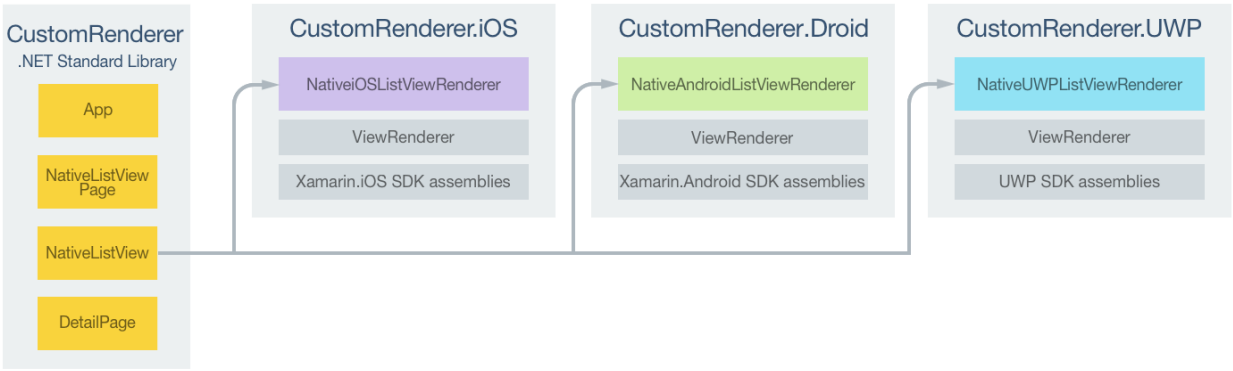
在每个平台上创建自定义呈现器

创建自定义呈现器类的过程如下所示：

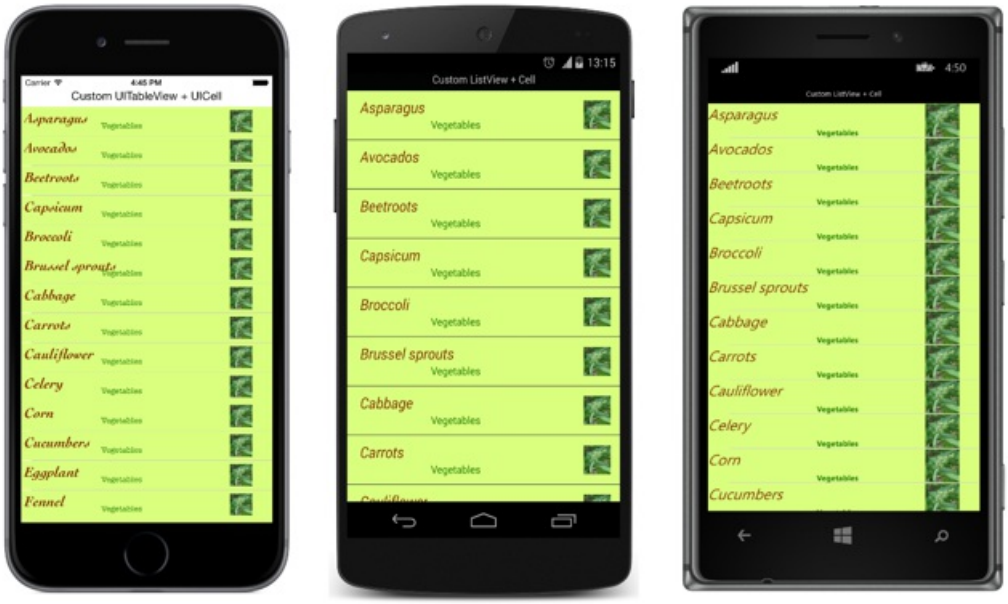
1. 创建一个子类`ListViewRenderer`呈现自定义控件的类。
2. 重写`OnElementChanged`呈现其进行自定义的自定义控件和写入逻辑的方法。此方法时调用相应的 Xamarin.Forms `ListView`创建。
3. 添加`ExportRenderer`到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 自定义控件属性。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
> 若要提供每个平台项目中的自定义呈现器可以选择它。如果未注册的自定义呈现器，则将使用默认的呈现器的单元格的基类。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`NativeListView` 自定义控件呈现的特定于平台的呈现器类，它们都派生自 `ListViewRenderer` 为每个平台的类。这会导致每个 `NativeListView` 自定义控件呈现特定于平台的列表控件和本机单元格布局，如以下屏幕截图中所示：



`ListViewRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 自定义控件来呈现相应的本机控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序，`OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `NativeListView` 实例。

重写的版本 `OnElementChanged` 中每个特定于平台的呈现器类，方法是执行的本机控件自定义的位置。可以通过访问对正在使用在平台上的本机控件的类型化的引用 `Control` 属性。此外，通过获取对所呈现的 Xamarin.Forms 控件的引用 `Element` 属性。

订阅中的事件处理程序时必须小心 `OnElementChanged` 方法，如下面的代码示例中所示：

```
protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
{
  base.OnElementChanged (e);

  if (e.OldElement != null) {
    // Unsubscribe from event handlers and cleanup any resources
  }

  if (e.NewElement != null) {
    // Configure the native control and subscribe to event handlers
  }
}
```

本机控件应仅配置和自定义呈现器附加到新 Xamarin.Forms 元素时订阅到事件处理程序。同样，任何已订阅的事件处理程序应取消订阅仅元素呈现器附加到更改时。采用此方法将有助于创建不会遭受内存泄漏的自定义呈现器。

重写的版本 `OnElementPropertyChanged` 中每个特定于平台的呈现器类，方法是对 Xamarin.Forms 自定义控件上的可绑定属性更改做出响应的位置。始终应进行检查更改的属性，如可以多次调用此重写。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 自定义控件的类型名称和自定义呈现器的类型名称。 `assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论每个特定于平台的自定义呈现器类的实现。

在 ios 设备上创建自定义呈现器

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
[assembly: ExportRenderer (typeof(NativeListView), typeof(NativeiOSListViewRenderer))]
namespace CustomRenderer.iOS
{
    public class NativeiOSListViewRenderer : ListViewRenderer
    {
        protected override void OnElementChanged (ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged (e);

            if (e.OldElement != null) {
                // Unsubscribe
            }

            if (e.NewElement != null) {
                Control.Source = new NativeiOSListViewSource (e.NewElement as NativeListView);
            }
        }
    }
}
```

`UITableView` 通过在创建实例的配置控制 `NativeiOSListViewSource` 类，前提是自定义呈现器附加到新 Xamarin.Forms 元素。此类提供到数据 `UITableView` 通过重写控件 `RowsInSection` 并 `GetCell` 方法从 `UITableViewSource` 类，并通过公开 `Items` 属性，其中包含要显示的数据列表。类还提供了 `RowSelected` 调用的方法重写 `ItemSelected` 事件提供的 `NativeListView` 自定义控件。重写的方法的相关详细信息，请参阅子类化 UITableViewSource。 `GetCell` 方法将返回 `UITableCellView`，使用在列表中，每行数据填充，并在下面的代码示例所示：

```
public override UITableViewCell GetCell (UITableView tableView, NSIndexPath indexPath)
{
  // request a recycled cell to save memory
  NativeiOSListViewCell cell = tableView.DequeueReusableCell (cellIdentifier) as NativeiOSListViewCell;

  // if there are no cells to reuse, create a new one
  if (cell == null) {
    cell = new NativeiOSListViewCell (cellIdentifier);
  }

  if (String.IsNullOrWhiteSpace (tableItems [indexPath.Row].ImageFilename)) {
    cell.UpdateCell (tableItems [indexPath.Row].Name
      , tableItems [indexPath.Row].Category
      , null);
  } else {
    cell.UpdateCell (tableItems [indexPath.Row].Name
      , tableItems [indexPath.Row].Category
      , UIImage.FromFile ("Images/" + tableItems [indexPath.Row].ImageFilename + ".jpg"));
  }

  return cell;
}
```

此方法创建 `NativeiOSListViewCell` 实例将在屏幕显示的数据的每一行。 `NativeiOSCell` 实例定义的每个单元格和单元格的数据布局。当单元格从由于滚动屏幕消失后时，该单元格将可供重复使用。这可以避免通过确保仅有的浪费内存 `NativeiOSCell` 要显示在屏幕上，而不是所有在列表中的数据的数据的实例。有关单元格重复使用的详细信息，请参阅单元格重用。 `GetCell` 方法还会读取 `ImageFilename` 属性的数据，提供它存在，并读取图像，并将其存储为每一行 `UIImage` 实例，然后更新 `NativeiOSListViewCell` 实例（名称、类别和图像）的数据行。

`NativeiOSListViewCell` 类定义每个单元格的布局，并在下面的代码示例所示：

```
public class NativeiOSListViewCell : UITableViewCell
{
  UILabel headingLabel, subheadingLabel;
  UIImageView imageView;

  public NativeiOSListViewCell (NSString cellId) : base (UITableViewCellStyle.Default, cellId)
  {
    SelectionStyle = UITableViewCellSelectionStyle.Gray;

    ContentView.BackgroundColor = UIColor.FromRGB (218, 255, 127);

    imageView = new UIImageView ();

    headingLabel = new UILabel () {
      Font = UIFont.FromName ("Cochin-BoldItalic", 22f),
      TextColor = UIColor.FromRGB (127, 51, 0),
      BackgroundColor = UIColor.Clear
    };

    subheadingLabel = new UILabel () {
      Font = UIFont.FromName ("AmericanTypewriter", 12f),
      TextColor = UIColor.FromRGB (38, 127, 0),
      TextAlignment = UITextAlignment.Center,
      BackgroundColor = UIColor.Clear
    };

    ContentView.Add (headingLabel);
    ContentView.Add (subheadingLabel);
    ContentView.Add (imageView);
  }

  public void UpdateCell (string caption, string subtitle, UIImage image)
  {
    headingLabel.Text = caption;
    subheadingLabel.Text = subtitle;
    imageView.Image = image;
  }

  public override void LayoutSubviews ()
  {
    base.LayoutSubviews ();

    headingLabel.Frame = new CoreGraphics.CGRect (5, 4, ContentView.Bounds.Width - 63, 25);
    subheadingLabel.Frame = new CoreGraphics.CGRect (100, 18, 100, 20);
    imageView.Frame = new CoreGraphics.CGRect (ContentView.Bounds.Width - 63, 5, 33, 33);
  }
}
```

此类定义用于呈现该单元格的内容和其布局的控件。`NativeiOSListViewCell` 构造函数创建的实例 `UILabel` 和 `UIImageView` 控件，并初始化它们的外观。这些控件用于显示与每个行的数据，而 `UpdateCell` 用来设置此数据方法 `UILabel` 和 `UIImageView` 实例。这些实例的位置设置的重写 `LayoutSubviews` 方法，通过指定单元格内的其坐标。

响应的自定义控件上的属性更改

如果 `NativeListView.Items` 属性发生更改，由于添加到的项或从列表中删除，自定义呈现器需要通过显示所做的更改做出响应。这可以通过重写实现 `OnElementPropertyChanged` 方法，在下面的代码示例所示：

```
protected override void OnElementPropertyChanged (object sender,
System.ComponentModel.PropertyChangedEventArgs e)
{
  base.OnElementPropertyChanged (sender, e);

  if (e.PropertyName == NativeListView.ItemsProperty.PropertyName) {
    Control.Source = new NativeiOSListViewSource (Element as NativeListView);
  }
}
```

方法创建的新实例 `NativeiOSListViewSource` 向提供数据的类 `UITableView` 控件，提供的可绑定 `NativeListView.Items` 属性已更改。

在 Android 上创建自定义呈现器

下面的代码示例显示了 Android 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(NativeListView), typeof(NativeAndroidListViewRenderer))]
namespace CustomRenderer.Droid
{
    public class NativeAndroidListViewRenderer : ListViewRenderer
    {
        Context _context;

        public NativeAndroidListViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged(e);

            if (e.OldElement != null)
            {
                // unsubscribe
                Control.ItemClick -= OnItemClick;
            }

            if (e.NewElement != null)
            {
                // subscribe
                Control.Adapter = new NativeAndroidListViewAdapter(_context as Android.App.Activity,
e.NewElement as NativeListView);
                Control.ItemClick += OnItemClick;
            }
        }
        ...

        void OnItemClick(object sender, Android.Widget.AdapterView.ItemClickEventArgs e)
        {
            ((NativeListView)Element).NotifyItemSelected(((NativeListView)Element).Items.ToList()[e.Position -
1]);
        }
    }
}
```

本机 `ListView` 控件配置为提供的自定义呈现器附加到新 Xamarin.Forms 元素。此配置涉及到创建的实例 `NativeAndroidListViewAdapter` 类，该类提供数据的本机 `ListView` 控件，并注册事件处理程序来处理 `ItemClick` 事件。反过来，此处理程序将调用 `ItemSelected` 事件提供的 `NativeListView` 自定义控件。 `ItemClick` 如果 Xamarin.Forms 元素呈现器附加到更改从订阅的事件，将取消。

`NativeAndroidListViewAdapter` 派生自 `BaseAdapter` 类并公开 `Items` 属性，其中包含要显示数据的列表，以及重写

`Count`, `GetView`, `GetItemId`, 和 `this[int]` 方法。有关这些方法重写的详细信息，请参阅实现 ListAdapter。 `GetView` 方法返回填充了数据，每个行的视图，并在下面的代码示例所示：

```
public override View GetView (int position, View convertView, ViewGroup parent)
{
  var item = tableItems [position];

  var view = convertView;
  if (view == null) {
    // no view to re-use, create new
    view = context.LayoutInflater.Inflate (Resource.Layout.NativeAndroidListViewCell, null);
  }
  view.FindViewById<TextView> (Resource.Id.Text1).Text = item.Name;
  view.FindViewById<TextView> (Resource.Id.Text2).Text = item.Category;

  // grab the old image and dispose of it
  if (view.FindViewById<ImageView> (Resource.Id.Image).Drawable != null) {
    using (var image = view.FindViewById<ImageView> (Resource.Id.Image).Drawable as BitmapDrawable) {
      if (image != null) {
        if (image.Bitmap != null) {
          //image.Bitmap.Recycle ();
          image.Bitmap.Dispose ();
        }
      }
    }
  }

  // If a new image is required, display it
  if (!String.IsNullOrWhiteSpace (item.ImageFilename)) {
    context.Resources.GetBitmapAsync (item.ImageFilename).ContinueWith ((t) => {
      var bitmap = t.Result;
      if (bitmap != null) {
        view.FindViewById<ImageView> (Resource.Id.Image).SetImageBitmap (bitmap);
        bitmap.Dispose ();
      }
    }, TaskScheduler.FromCurrentSynchronizationContext ());
  } else {
    // clear the image
    view.FindViewById<ImageView> (Resource.Id.Image).SetImageBitmap (null);
  }

  return view;
}
```

`GetView` 调用方法以返回要呈现的单元格作为 `View`，列表中的数据的每一行。它会创建 `View` 实例将使用的外观在屏幕显示的数据的每一行 `View` 布局文件中定义的实例。当单元格从由于滚动屏幕消失后时，该单元格将可供重复使用。这可以避免通过确保仅有的浪费内存 `View` 要显示在屏幕上，而不是所有在列表中的数据的数据的实例。有关视图重用的详细信息，请参阅重复使用行视图。

`GetView` 方法还填充 `View` 具有数据，包括从在指定的文件名中读取图像数据实例 `ImageFilename` 属性。

通过本机每个单元格 dispayed 布局 `ListView` 中定义 `NativeAndroidListViewCell.axml` 布局文件，它被放大 `LayoutInflater.Inflate` 方法。下面的代码示例显示了布局定义：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:background="@drawable/CustomSelector">
    <LinearLayout
        android:id="@+id/Text"
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="10dip">
        <TextView
            android:id="@+id/Text1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dip"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/Text2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="14dip"
            android:textColor="#FF267F00"
            android:paddingLeft="100dip" />
    </LinearLayout>
    <ImageView
        android:id="@+id/Image"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:padding="5dp"
        android:src="@drawable/icon"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

此布局指定两个 `TextView` 控件和一个 `ImageView` 控件用于显示单元格的内容。这两个 `TextView` 控件是在垂直方向 `LinearLayout` 控件，与中所包含的所有控件 `RelativeLayout` 。

响应的自定义控件上的属性更改

如果 `NativeListView.Items` 属性发生更改，由于添加到的项或从列表中删除，自定义呈现器需要通过显示所做的更改做出响应。这可以通过重写实现 `OnElementPropertyChanged` 方法，在下面的代码示例所示：

```
protected override void OnElementPropertyChanged (object sender,
System.ComponentModel.PropertyChangedEventArgs e)
{
  base.OnElementPropertyChanged (sender, e);

  if (e.PropertyName == NativeListView.ItemsProperty.PropertyName) {
    Control.Adapter = new NativeAndroidListViewAdapter (_context as Android.App.Activity, Element as
NativeListView);
  }
}
```

方法创建的新实例 `NativeAndroidListViewAdapter` 类，该类提供数据的本机 `ListView` 控件，提供的可绑定 `NativeListView.Items` 属性已更改。

在 UWP 上创建自定义呈现器

下面的代码示例显示了适用于 UWP 的自定义呈现器：

```
[assembly: ExportRenderer(typeof(NativeListView), typeof(NativeUWPListViewRenderer))]
namespace CustomRenderer.UWP
{
    public class NativeUWPListViewRenderer : ListViewRenderer
    {
        ListView listView;

        protected override void OnElementChanged(ElementChangedEventArgs<Xamarin.Forms.ListView> e)
        {
            base.OnElementChanged(e);

            listView = Control as ListView;

            if (e.OldElement != null)
            {
                // Unsubscribe
                listView.SelectionChanged -= OnSelectedItemChanged;
            }

            if (e.NewElement != null)
            {
                listView.SelectionMode = ListViewSelectionMode.Single;
                listView.IsItemClickEnabled = false;
                listView.ItemsSource = ((NativeListView)e.NewElement).Items;
                listView.ItemTemplate = App.Current.Resources["ListViewItemTemplate"] as
Windows.UI.Xaml.DataTemplate;
                // Subscribe
                listView.SelectionChanged += OnSelectedItemChanged;
            }
        }

        void OnSelectedItemChanged(object sender, SelectionChangedEventArgs e)
        {
            ((NativeListView)Element).NotifyItemSelected(listView.SelectedItem);
        }
    }
}
```

本机 `ListView` 控件配置为提供的自定义呈现器附加到新 Xamarin.Forms 元素。此配置涉及到设置如何本机 `ListView` 控件将响应所选择的项填充数据显示由该控件的外观和每个单元格的内容定义和注册事件处理程序来处理 `SelectionChanged` 事件。反过来，此处理程序将调用 `ItemSelected` 事件提供的 `NativeListView` 自定义控件。 `SelectionChanged` 如果 Xamarin.Forms 元素呈现器附加到更改从订阅的事件，将取消。

外观和内容的每个本机 `ListView` 由定义单元格 `DataTemplate` 名为 `ListViewItemTemplate` 。这 `DataTemplate` 存储在应用程序级资源字典中，并在下面的代码示例所示：

```
<DataTemplate x:Key="ListViewItemTemplate">
    <Grid Background="#DAFF7F">
        <Grid.Resources>
            <local:ConcatImageExtensionConverter x:Name="ConcatImageExtensionConverter" />
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.40*"/>
            <ColumnDefinition Width="0.20*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.ColumnSpan="2" Foreground="#7F3300" FontStyle="Italic" FontSize="22"
VerticalAlignment="Top" Text="{Binding Name}" />
        <TextBlock Grid.RowSpan="2" Grid.Column="1" Foreground="#267F00" FontWeight="Bold" FontSize="12"
VerticalAlignment="Bottom" Text="{Binding Category}" />
        <Image Grid.RowSpan="2" Grid.Column="2" HorizontalAlignment="Left" VerticalAlignment="Center" Source="
{Binding ImageFilename, Converter={StaticResource ConcatImageExtensionConverter}}" Width="50" Height="50" />
        <Line Grid.Row="1" Grid.ColumnSpan="3" X1="0" X2="1" Margin="30,20,0,0" StrokeThickness="1"
Stroke="LightGray" Stretch="Fill" VerticalAlignment="Bottom" />
    </Grid>
</DataTemplate>
```

`DataTemplate` 指定用来显示该单元格，其布局和外观的内容的控件。两个 `TextBlock` 控件和一个 `Image` 控件用于显示通过数据绑定的单元格的内容。此外，实例 `ConcatImageExtensionConverter` 用于连接 `.jpg` 文件扩展名为每个图像文件名称。这可确保 `Image` 控件可以加载并呈现图像时 `Source` 属性设置。

响应的自定义控件上的属性更改

如果 `NativeListView.Items` 属性发生更改，由于添加到的项或从列表中删除，自定义呈现器需要通过显示所做的更改做出响应。这可以通过重写实现 `OnElementPropertyChanged` 方法，在下面的代码示例所示：

```
protected override void OnElementPropertyChanged(object sender, System.ComponentModel.PropertyChangedEventArgs
e)
{
    base.OnElementPropertyChanged(sender, e);

    if (e.PropertyName == NativeListView.ItemsProperty.PropertyName)
    {
        listView.ItemsSource = ((NativeListView)Element).Items;
    }
}
```

该方法将重新填充本机 `ListView` 控件的已更改的数据，提供的可绑定 `NativeListView.Items` 属性已更改。

总结

本文演示了如何创建自定义呈现器，用于封装特定于平台的列表控件和本机单元格的布局，允许更好地控制本机列表控制性能。

相关链接

- CustomRendererListView（示例）

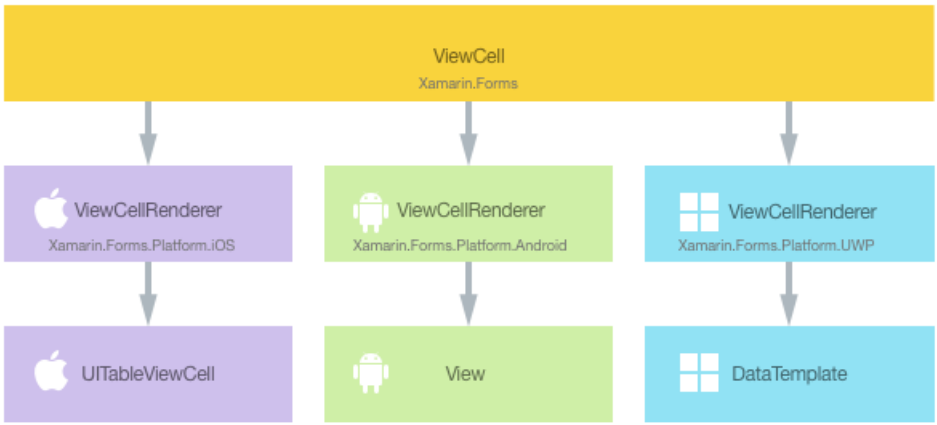
自定义 ViewCell

2018/7/13 • • Edit Online

Xamarin.Forms ViewCell 是可以添加到 ListView 或 TableView，其中包含开发人员定义的视图的单元格。本文演示如何创建 ViewCell Xamarin.Forms ListView 控件中托管的自定义呈现器。这将停止 Xamarin.Forms 布局计算正在从 ListView 滚动期间重复调用。

每个 Xamarin.Forms 单元格已创建的本机控件实例的每个平台随附的呈现器。当 `ViewCell` Xamarin.Forms 应用程序，在 iOS 中呈现 `ViewCellRenderer` 类实例化时，这反过来实例化本机 `UITableViewCell` 控件。在 Android 平台上 `ViewCellRenderer` 类实例化本机 `View` 控件。在通用 Windows 平台 (UWP)，`ViewCellRenderer` 类实例化本机 `DataTemplate`。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `ViewCell` 和相应的本机控件实现它：



渲染过程时可以执行利用通过创建自定义呈现器为实现特定于平台的自定义 `ViewCell` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义单元格。
2. 使用Xamarin.Forms 中的自定义单元格。
3. 创建每个平台上的单元格的自定义呈现器。

每个项将现在讨论反过来，实现 `NativeCell` 呈现器都使用了 Xamarin.Forms 中承载的每个单元的特定于平台的布局 `ListView` 控件。这会阻止被重复调用期间的 Xamarin.Forms 布局计算 `ListView` 滚动。

创建自定义单元格

可以通过子类化创建一个自定义单元格控件 `ViewCell` 类，如下面的代码示例中所示：

```
public class NativeCell : ViewCell
{
  public static readonly BindableProperty NameProperty =
    BindableProperty.Create ("Name", typeof(string), typeof(NativeCell), "");

  public string Name {
    get { return (string)GetValue (NameProperty); }
    set { SetValue (NameProperty, value); }
  }

  public static readonly BindableProperty CategoryProperty =
    BindableProperty.Create ("Category", typeof(string), typeof(NativeCell), "");

  public string Category {
    get { return (string)GetValue (CategoryProperty); }
    set { SetValue (CategoryProperty, value); }
  }

  public static readonly BindableProperty ImageFilenameProperty =
    BindableProperty.Create ("ImageFilename", typeof(string), typeof(NativeCell), "");

  public string ImageFilename {
    get { return (string)GetValue (ImageFilenameProperty); }
    set { SetValue (ImageFilenameProperty, value); }
  }
}
```

`NativeCell`类在.NET Standard 库项目中创建和定义自定义单元格的 API。自定义单元格公开`Name`、`Category`和`ImageFilename`可以通过数据绑定显示的属性。若要深入了解数据绑定，请参阅数据绑定基本知识。

使用自定义单元格

`NativeCell`自定义单元格可以是在 Xaml 中引用.NET Standard 库项目中通过声明其位置的命名空间和自定义单元格元素上使用的命名空间前缀。下面的代码示例演示如何将`NativeCell`自定义单元格可供 XAML 页：

```
<ContentPage ...
    xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
    ...>
    ...
    <ContentPage.Content>
          <StackLayout>
              <Label Text="Xamarin.Forms native cell" HorizontalTextAlignment="Center" />
              <ListView x:Name="listView" CachingStrategy="RecycleElement" ItemSelected="OnItemSelected">
                  <ListView.ItemTemplate>
                      <DataTemplate>
                          <local:NativeCell Name="{Binding Name}" Category="{Binding Category}"
ImageFilename="{Binding ImageFilename}" />
                      </DataTemplate>
                  </ListView.ItemTemplate>
              </ListView>
          </StackLayout>
      </ContentPage.Content>
</ContentPage>
```

`local`命名空间前缀可以命名任何内容。但是，`clr-namespace`和`assembly`值必须匹配的自定义控件的详细信息。一旦声明的命名空间，前缀用于引用自定义单元格。

下面的代码示例演示如何将`NativeCell`自定义单元格可供 C# 页：

```
public class NativeCellPageCS : ContentPage
{
    ListView listView;

    public NativeCellPageCS()
    {
        listView = new ListView(ListViewCachingStrategy.RecycleElement)
        {
            ItemsSource = DataSource.GetList(),
            ItemTemplate = new DataTemplate(() =>
            {
                var nativeCell = new NativeCell();
                nativeCell.SetBinding(NativeCell.NameProperty, "Name");
                nativeCell.SetBinding(NativeCell.CategoryProperty, "Category");
                nativeCell.SetBinding(NativeCell.ImageFilenameProperty, "ImageFilename");

                return nativeCell;
            })
        };

        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
                Padding = new Thickness(0, 20, 0, 0);
                break;
            case Device.Android:
            case Device.UWP:
                Padding = new Thickness(0);
                break;
        }

        Content = new StackLayout
        {
            Children = {
                new Label { Text = "Xamarin.Forms native cell", HorizontalTextAlignment =
TextAlignment.Center },
                listView
            }
        };
        listView.ItemSelected += OnItemSelected;
    }
    ...
}
```

Xamarin.Forms `ListView` 控件用于显示数据，通过填充一系列 `ItemSource` 属性。`RecycleElement` 缓存策略尝试最大程度减少 `ListView` 通过回收列表单元格占用的内存和执行速度。有关详细信息，请参阅缓存策略。

在列表中的每一行都包含三个项的数据-名称、类别和图像文件名。在列表中每行的布局由定义 `DataTemplate` 通过引用 `ListView.ItemTemplate` 可绑定属性。`DataTemplate` 定义会在列表中的数据的每个行 `NativeCell`，它显示其 `Name`， `Category`，和 `ImageFilename` 通过数据绑定的属性。有关详细信息 `ListView` 控件，请参阅ListView。

自定义呈现器现在可以添加到自定义的每个单元的特定于平台的布局的每个应用程序项目。

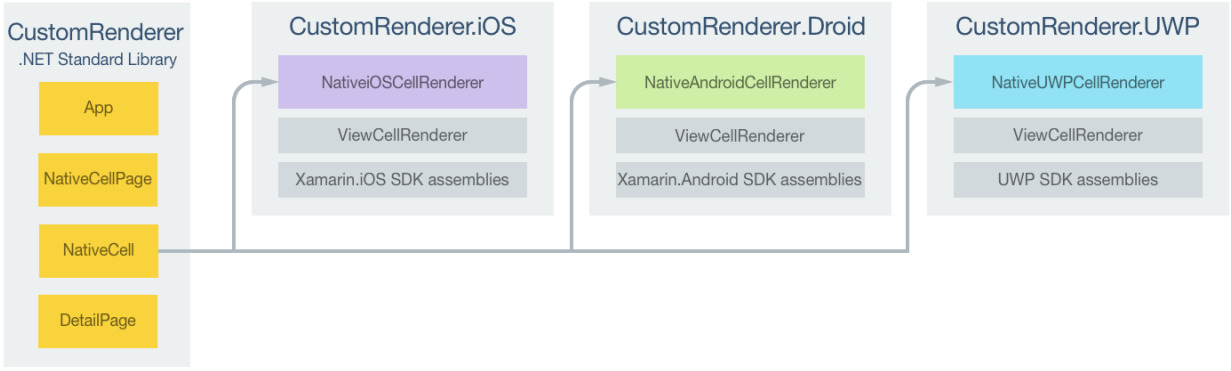
在每个平台上创建自定义呈现器

创建自定义呈现器类的过程如下所示：

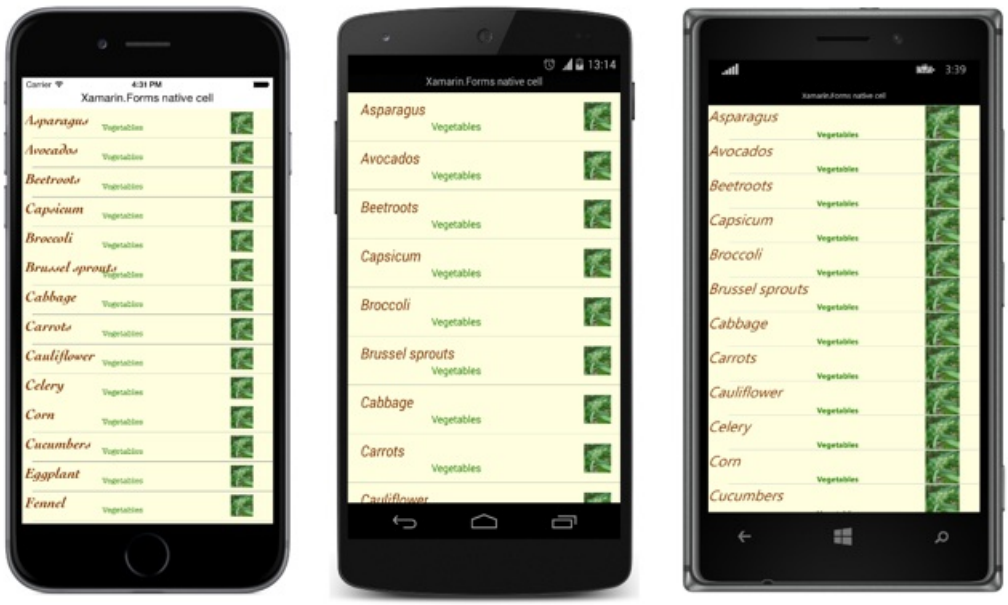
1. 创建一个子类 `ViewCellRenderer` 呈现自定义单元格的类。
2. 重写呈现自定义单元格的特定于平台的方法并编写逻辑以其进行自定义。
3. 添加 `ExportRenderer` 属性到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 自定义单元格。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
>
> 对于大多数 Xamarin.Forms 元素，它是可选提供每个平台项目中的自定义呈现器。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。但是，自定义呈现器呈现时所需的每个平台项目中ViewCell元素。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`NativeCell`自定义单元格呈现的特定于平台的呈现器类，它们都派生自`ViewCellRenderer`为每个平台的类。这会导致每个`NativeCell`自定义单元格呈现特定于平台的布局，如以下屏幕截图中所示：



`ViewCellRenderer`类公开特定于平台的方法来呈现自定义单元格。这是`GetCell` iOS 平台上的方法`GetCellCore` Android 平台上的方法和`GetTemplate` UWP 上的方法。

每个自定义呈现器类用修饰`ExportRenderer`与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数-Xamarin.Forms 单元格所呈现的类型名称和自定义呈现器的类型名称。`assembly`到的属性的前缀指定特性应用于整个程序集。

以下各节讨论每个特定于平台的自定义呈现器类的实现。

在 ios 设备上创建自定义呈现器

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeiOSCellRenderer))]
namespace CustomRenderer.iOS
{
    public class NativeiOSCellRenderer : ViewCellRenderer
    {
        NativeiOSCell cell;

        public override UITableViewCell GetCell(Cell item, UITableViewCell reusableCell, UITableView tv)
        {
            var nativeCell = (NativeCell)item;

            cell = reusableCell as NativeiOSCell;
            if (cell == null)
                cell = new NativeiOSCell(item.GetType().FullName, nativeCell);
            else
                cell.NativeCell.PropertyChanged -= OnNativeCellPropertyChanged;

            nativeCell.PropertyChanged += OnNativeCellPropertyChanged;
            cell.UpdateCell(nativeCell);
            return cell;
        }
        ...
    }
}
```

`GetCell` 调用方法来生成要显示每个单元格。每个单元格是 `NativeiOSCell` 实例，用于定义单元和其数据的布局。操作 `GetCell` 方法所依赖 `ListView` 缓存策略：

- 当 `ListView` 缓存策略是 `RetainElement` ，则 `GetCell` 方法将调用的每个单元。一个 `NativeiOSCell` 将为每个创建实例 `NativeCell` 最初在屏幕显示的实例。当用户通过滚动 `ListView` ， `NativeiOSCell` 实例将重新使用。有关 iOS 单元格重复使用的详细信息，请参阅单元格重用。

  > **NOTE**
  > 此自定义呈现器代码将执行一些单元格重复使用，即使 `ListView` 设置保留单元格。

  每个显示的数据 `NativeiOSCell` 实例，新创建的或重复使用，是否将更新从每个数据 `NativeCell` 实例通过 `UpdateCell` 方法。

  > **NOTE**
  > `OnNativeCellPropertyChanged` 方法将永远不会调用何时 `ListView` 缓存策略设置为保留的单元格。

- 当 `ListView` 缓存策略是 `RecycleElement` ，则 `GetCell` 最初在屏幕上显示每个单元格将调用方法。一个 `NativeiOSCell` 将为每个创建实例 `NativeCell` 最初在屏幕显示的实例。每个显示的数据 `NativeiOSCell` 中的数据将更新实例 `NativeCell` 实例通过 `UpdateCell` 方法。但是， `GetCell` 不会调用方法，当用户滚动通过 `ListView` 。相反， `NativeiOSCell` 实例将重新使用。 `PropertyChanged` 将在引发事件 `NativeCell` 实例时数据发生更改，并 `OnNativeCellPropertyChanged` 事件处理程序将更新的数据在每个重复使用 `NativeiOSCell` 实例。

下面的代码示例演示 `OnNativeCellPropertyChanged` 方法调用时 `PropertyChanged` 引发事件：

```
namespace CustomRenderer.iOS
{
    public class NativeiOSCellRenderer : ViewCellRenderer
    {
        ...

        void OnNativeCellPropertyChanged(object sender, PropertyChangedEventArgs e)
        {
            var nativeCell = (NativeCell)sender;
            if (e.PropertyName == NativeCell.NameProperty.PropertyName)
            {
                cell.HeadingLabel.Text = nativeCell.Name;
            }
            else if (e.PropertyName == NativeCell.CategoryProperty.PropertyName)
            {
                cell.SubheadingLabel.Text = nativeCell.Category;
            }
            else if (e.PropertyName == NativeCell.ImageFilenameProperty.PropertyName)
            {
                cell.CellImageView.Image = cell.GetImage(nativeCell.ImageFilename);
            }
        }
    }
}
```

此方法将更新所显示的数据重新使用 `NativeiOSCell` 实例。检查已更改的属性变得很，因为可以多次调用方法。

`NativeiOSCell` 类定义每个单元格的布局，并在下面的代码示例所示：

```
internal class NativeiOSCell : UITableViewCell, INativeElementView
{
  public UILabel HeadingLabel { get; set; }
  public UILabel SubheadingLabel { get; set; }
  public UIImageView CellImageView { get; set; }

  public NativeCell NativeCell { get; private set; }
  public Element Element => NativeCell;

  public NativeiOSCell(string cellId, NativeCell cell) : base(UITableViewCellStyle.Default, cellId)
  {
    NativeCell = cell;

    SelectionStyle = UITableViewCellSelectionStyle.Gray;
    ContentView.BackgroundColor = UIColor.FromRGB(255, 255, 224);
    CellImageView = new UIImageView();

    HeadingLabel = new UILabel()
    {
      Font = UIFont.FromName("Cochin-BoldItalic", 22f),
      TextColor = UIColor.FromRGB(127, 51, 0),
      BackgroundColor = UIColor.Clear
    };

    SubheadingLabel = new UILabel()
    {
      Font = UIFont.FromName("AmericanTypewriter", 12f),
      TextColor = UIColor.FromRGB(38, 127, 0),
      TextAlignment = UITextAlignment.Center,
      BackgroundColor = UIColor.Clear
    };

    ContentView.Add(HeadingLabel);
    ContentView.Add(SubheadingLabel);
    ContentView.Add(CellImageView);
  }

  public void UpdateCell(NativeCell cell)
  {
    HeadingLabel.Text = cell.Name;
    SubheadingLabel.Text = cell.Category;
    CellImageView.Image = GetImage(cell.ImageFilename);
  }

  public UIImage GetImage(string filename)
  {
    return (!string.IsNullOrWhiteSpace(filename)) ? UIImage.FromFile("Images/" + filename + ".jpg") : null;
  }

  public override void LayoutSubviews()
  {
    base.LayoutSubviews();

    HeadingLabel.Frame = new CGRect(5, 4, ContentView.Bounds.Width - 63, 25);
    SubheadingLabel.Frame = new CGRect(100, 18, 100, 20);
    CellImageView.Frame = new CGRect(ContentView.Bounds.Width - 63, 5, 33, 33);
  }
}
```

此类定义用于呈现该单元格的内容和其布局的控件。类实现 `INativeElementView` 接口，这是需要 `ListView` 使用 `RecycleElement` 缓存策略。此接口指定的类必须实现 `Element` 属性，它应返回回收单元格的自定义单元格数据。

`NativeiOSCell` 构造函数初始化的外观 `HeadingLabel`， `SubheadingLabel`，和 `CellImageView` 属性。这些属性用于显示中存储的数据 `NativeCell` 实例，与 `UpdateCell` 方法被调用来设置每个属性的值。此外，当 `ListView` 使用 `RecycleElement` 缓存策略，所显示的数据 `HeadingLabel`， `SubheadingLabel`，和 `CellImageView` 属性可以是通过更新

`OnNativeCellPropertyChanged` 中自定义呈现器的方法。

通过执行单元格布局 `LayoutSubviews` 重写，该设置的坐标 `HeadingLabel`， `SubheadingLabel`，和 `CellImageView` 单元格内。

在 Android 上创建自定义呈现器

下面的代码示例显示了 Android 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeAndroidCellRenderer))]
namespace CustomRenderer.Droid
{
    public class NativeAndroidCellRenderer : ViewCellRenderer
    {
        NativeAndroidCell cell;

        protected override Android.Views.View GetCellCore(Cell item, Android.Views.View convertView,
ViewGroup parent, Context context)
        {
            var nativeCell = (NativeCell)item;
            Console.WriteLine("\t\t" + nativeCell.Name);

            cell = convertView as NativeAndroidCell;
            if (cell == null)
            {
                cell = new NativeAndroidCell(context, nativeCell);
            }
            else
            {
                cell.NativeCell.PropertyChanged -= OnNativeCellPropertyChanged;
            }

            nativeCell.PropertyChanged += OnNativeCellPropertyChanged;

            cell.UpdateCell(nativeCell);
            return cell;
        }
        ...
    }
}
```

`GetCellCore` 调用方法来生成要显示每个单元格。每个单元格是 `NativeAndroidCell` 实例，用于定义单元和其数据的布局。操作 `GetCellCore` 方法所依赖 `ListView` 缓存策略：

- 当 `ListView` 缓存策略是 `RetainElement`，则 `GetCellCore` 方法将调用的每个单元。一个 `NativeAndroidCell` 将为每个创建 `NativeCell` 最初在屏幕显示的实例。当用户通过滚动 `ListView`， `NativeAndroidCell` 实例将重新使用。有关 Android 单元格重复使用的详细信息，请参阅重复使用行视图。

  > **NOTE**
  > 请注意此自定义呈现器代码将执行一些单元格重复使用，即使 `ListView` 设置保留单元格。

  每个显示的数据 `NativeAndroidCell` 实例，新创建的或重复使用，是否将更新从每个数据 `NativeCell` 实例通过 `UpdateCell` 方法。

  > **NOTE**
  > 请注意，当 `OnNativeCellPropertyChanged` 方法将调用何时 `ListView` 是设置为保留单元，它将不会更新 `NativeAndroidCell` 属性值。

- 当 `ListView` 缓存策略是 `RecycleElement`，则 `GetCellCore` 最初在屏幕上显示每个单元格将调用方法。一个 `NativeAndroidCell` 将为每个创建实例 `NativeCell` 最初在屏幕显示的实例。每个显示的数据 `NativeAndroidCell` 中的数据将更新实例 `NativeCell` 实例通过 `UpdateCell` 方法。但是，`GetCellCore` 不会调用方法，当用户滚动通过 `ListView` 。相反，`NativeAndroidCell` 实例将重新使用。`PropertyChanged` 将在引发事件 `NativeCell` 实例时数据发生更改，并 `OnNativeCellPropertyChanged` 事件处理程序将更新的数据在每个重复使用 `NativeAndroidCell` 实例。

下面的代码示例演示 `OnNativeCellPropertyChanged` 方法调用时 `PropertyChanged` 引发事件：

```
namespace CustomRenderer.Droid
{
    public class NativeAndroidCellRenderer : ViewCellRenderer
    {
        ...

        void OnNativeCellPropertyChanged(object sender, PropertyChangedEventArgs e)
        {
            var nativeCell = (NativeCell)sender;
            if (e.PropertyName == NativeCell.NameProperty.PropertyName)
            {
                cell.HeadingTextView.Text = nativeCell.Name;
            }
            else if (e.PropertyName == NativeCell.CategoryProperty.PropertyName)
            {
                cell.SubheadingTextView.Text = nativeCell.Category;
            }
            else if (e.PropertyName == NativeCell.ImageFilenameProperty.PropertyName)
            {
                cell.SetImage(nativeCell.ImageFilename);
            }
        }
    }
}
```

此方法将更新所显示的数据重新使用 `NativeAndroidCell` 实例。检查已更改的属性变得很，因为可以多次调用方法。

`NativeAndroidCell` 类定义每个单元格的布局，并在下面的代码示例所示：

```
internal class NativeAndroidCell : LinearLayout, INativeElementView
{
  public TextView HeadingTextView { get; set; }
  public TextView SubheadingTextView { get; set; }
  public ImageView ImageView { get; set; }

  public NativeCell NativeCell { get; private set; }
  public Element Element => NativeCell;

  public NativeAndroidCell(Context context, NativeCell cell) : base(context)
  {
    NativeCell = cell;

    var view = (context as Activity).LayoutInflater.Inflate(Resource.Layout.NativeAndroidCell, null);
    HeadingTextView = view.FindViewById<TextView>(Resource.Id.HeadingText);
    SubheadingTextView = view.FindViewById<TextView>(Resource.Id.SubheadingText);
    ImageView = view.FindViewById<ImageView>(Resource.Id.Image);

    AddView(view);
  }

  public void UpdateCell(NativeCell cell)
  {
    HeadingTextView.Text = cell.Name;
    SubheadingTextView.Text = cell.Category;

    // Dispose of the old image
    if (ImageView.Drawable != null)
    {
      using (var image = ImageView.Drawable as BitmapDrawable)
      {
        if (image != null)
        {
          if (image.Bitmap != null)
          {
            image.Bitmap.Dispose();
          }
        }
      }
    }

    SetImage(cell.ImageFilename);
  }

  public void SetImage(string filename)
  {
    if (!string.IsNullOrWhiteSpace(filename))
    {
      // Display new image
      Context.Resources.GetBitmapAsync(filename).ContinueWith((t) =>
      {
        var bitmap = t.Result;
        if (bitmap != null)
        {
          ImageView.SetImageBitmap(bitmap);
          bitmap.Dispose();
        }
      }, TaskScheduler.FromCurrentSynchronizationContext());
    }
    else
    {
      // Clear the image
      ImageView.SetImageBitmap(null);
    }
  }
}
```

此类定义用于呈现该单元格的内容和其布局的控件。类实现 `INativeElementView` 接口，这是需要 `ListView` 使用 `RecycleElement` 缓存策略。此接口指定的类必须实现 `Element` 属性，它应返回回收单元格的自定义单元格数据。

`NativeAndroidCell` 构造函数增大 `NativeAndroidCell` 布局，并将初始化 `HeadingTextView`， `SubheadingTextView`，和 `ImageView` 夸大布局中控件的属性。这些属性用于显示中存储的数据 `NativeCell` 实例，与 `UpdateCell` 方法被调用来设置每个属性的值。此外，当 `ListView` 使用 `RecycleElement` 缓存策略，所显示的数据 `HeadingTextView`，`SubheadingTextView`，和 `ImageView` 属性可以是通过更新 `OnNativeCellPropertyChanged` 中自定义呈现器的方法。

下面的代码示例显示的布局定义 `NativeAndroidCell.axml` 布局文件：

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:padding="8dp"
    android:background="@drawable/CustomSelector">
    <LinearLayout
        android:id="@+id/Text"
        android:orientation="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:paddingLeft="10dip">
        <TextView
            android:id="@+id/HeadingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="#FF7F3300"
            android:textSize="20dip"
            android:textStyle="italic" />
        <TextView
            android:id="@+id/SubheadingText"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textSize="14dip"
            android:textColor="#FF267F00"
            android:paddingLeft="100dip" />
    </LinearLayout>
    <ImageView
        android:id="@+id/Image"
        android:layout_width="48dp"
        android:layout_height="48dp"
        android:padding="5dp"
        android:src="@drawable/icon"
        android:layout_alignParentRight="true" />
</RelativeLayout>
```

此布局指定两个 `TextView` 控件和一个 `ImageView` 控件用于显示单元格的内容。这两个 `TextView` 控件是在垂直方向 `LinearLayout` 控件，与中所包含的所有控件 `RelativeLayout` 。

在 UWP 上创建自定义呈现器

下面的代码示例显示了适用于 UWP 的自定义呈现器：

```
[assembly: ExportRenderer(typeof(NativeCell), typeof(NativeUWPCellRenderer))]
namespace CustomRenderer.UWP
{
    public class NativeUWPCellRenderer : ViewCellRenderer
    {
        public override Windows.UI.Xaml.DataTemplate GetTemplate(Cell cell)
        {
            return App.Current.Resources["ListViewItemTemplate"] as Windows.UI.Xaml.DataTemplate;
        }
    }
}
```

`GetTemplate` 调用方法以返回要呈现列表中的数据的每一行的单元格。它会创建 `DataTemplate` 为每个 `NativeCell` 实例将与显示在屏幕上，`DataTemplate` 定义外观和单元格的内容。

`DataTemplate` 存储在应用程序级资源字典中，并在下面的代码示例所示：

```
<DataTemplate x:Key="ListViewItemTemplate">
    <Grid Background="LightYellow">
        <Grid.Resources>
            <local:ConcatImageExtensionConverter x:Name="ConcatImageExtensionConverter" />
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.40*" />
            <ColumnDefinition Width="0.40*"/>
            <ColumnDefinition Width="0.20*" />
        </Grid.ColumnDefinitions>
        <TextBlock Grid.ColumnSpan="2" Foreground="#7F3300" FontStyle="Italic" FontSize="22"
VerticalAlignment="Top" Text="{Binding Name}" />
        <TextBlock Grid.RowSpan="2" Grid.Column="1" Foreground="#267F00" FontWeight="Bold" FontSize="12"
VerticalAlignment="Bottom" Text="{Binding Category}" />
        <Image Grid.RowSpan="2" Grid.Column="2" HorizontalAlignment="Left" VerticalAlignment="Center"
Source="{Binding ImageFilename, Converter={StaticResource ConcatImageExtensionConverter}}" Width="50"
Height="50" />
        <Line Grid.Row="1" Grid.ColumnSpan="3" X1="0" X2="1" Margin="30,20,0,0" StrokeThickness="1"
Stroke="LightGray" Stretch="Fill" VerticalAlignment="Bottom" />
    </Grid>
</DataTemplate>
```

`DataTemplate` 指定用来显示该单元格，其布局和外观的内容的控件。两个 `TextBlock` 控件和一个 `Image` 控件用于显示通过数据绑定的单元格的内容。此外，实例 `ConcatImageExtensionConverter` 用于连接 `.jpg` 文件扩展名为每个图像文件名称。这可确保 `Image` 控件可以加载并呈现图像时 `Source` 属性设置。

总结

本文演示了如何创建自定义呈现器 `ViewCell` 位于 Xamarin.Forms `ListView` 控件。这会阻止被重复调用期间的 Xamarin.Forms 布局计算 `ListView` 滚动。

相关链接

- ListView 性能
- CustomRendererViewCell（示例）

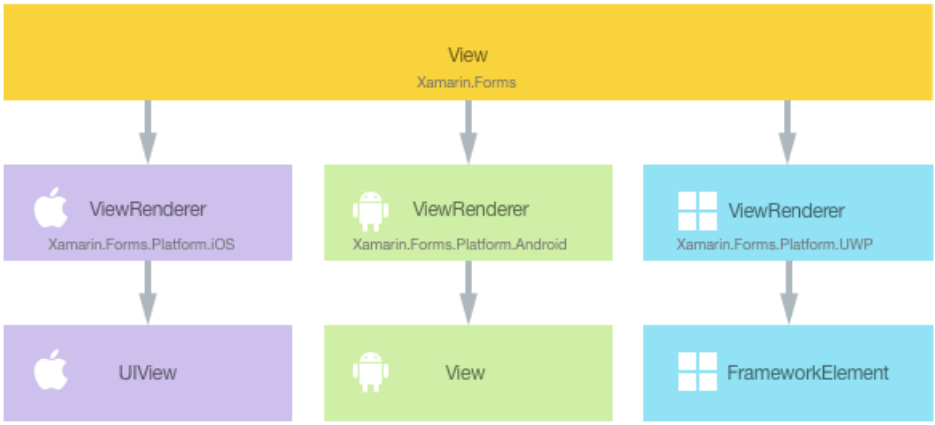
实现视图

2018/7/13 • • Edit Online

Xamarin.Forms 自定义用户界面控件应派生视图类，用于放置布局和屏幕上的控件。本文演示如何创建 Xamarin.Forms 自定义控件用于显示来自设备的摄像机的预览视频流的自定义呈现器。

每个 Xamarin.Forms 视图都会随附的呈现器为每个平台创建本机控件的实例。当 `View` Xamarin.Forms 应用程序在 iOS 中，呈现 `ViewRenderer` 类实例化时，这反过来实例化本机 `UIView` 控件。在 Android 平台上 `ViewRenderer` 类实例化本机 `View` 控件。在通用 Windows 平台 (UWP)，`ViewRenderer` 类实例化本机 `FrameworkElement` 控件。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `View` 和相应的本机控件实现它：



呈现过程可用于通过创建自定义呈现器为实现特定于平台的自定义 `View` 每个平台上。执行此操作的过程如下所示：

1. 创建Xamarin.Forms 自定义控件。
2. 使用Xamarin.Forms 中的自定义控件。
3. 创建每个平台上的控件的自定义呈现器。

每个项将现在讨论反过来，若要实现 `CameraPreview` 显示来自设备的摄像机的预览视频流的呈现器。点击视频流将停止并启动它。

创建自定义控件

可以通过子类化创建一个自定义控件 `View` 类，如下面的代码示例中所示：

```
public class CameraPreview : View
{
  public static readonly BindableProperty CameraProperty = BindableProperty.Create (
    propertyName: "Camera",
    returnType: typeof(CameraOptions),
    declaringType: typeof(CameraPreview),
    defaultValue: CameraOptions.Rear);

  public CameraOptions Camera {
    get { return (CameraOptions)GetValue (CameraProperty); }
    set { SetValue (CameraProperty, value); }
  }
}
```

`CameraPreview`自定义控件的可移植类库 (PCL) 项目中创建和定义用于控制的 API。自定义控件公开`Camera`用于控制是否应从前面或背面摄像头设备上的显示视频流的属性。如果指定的值不是`Camera`属性创建控件时，它默认为指定背面摄像头。

使用自定义控件

`CameraPreview`自定义控件可以在 XAML 中引用 PCL 项目中通过声明其位置的命名空间和自定义控件元素上使用的命名空间前缀。下面的代码示例演示如何将`CameraPreview`自定义控件可供 XAML 页：

```
<ContentPage ...
             xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
             ...>
    <ContentPage.Content>
        <StackLayout>
            <Label Text="Camera Preview:" />
            <local:CameraPreview Camera="Rear"
              HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`local`命名空间前缀可以命名任何内容。但是，`clr-namespace`和`assembly`值必须匹配的自定义控件的详细信息。一旦声明的命名空间，前缀用于引用自定义控件。

下面的代码示例演示如何将`CameraPreview`自定义控件可供 C# 页：

```
public class MainPageCS : ContentPage
{
  public MainPageCS ()
  {
    ...
    Content = new StackLayout {
      Children = {
        new Label { Text = "Camera Preview:" },
        new CameraPreview {
          Camera = CameraOptions.Rear,
          HorizontalOptions = LayoutOptions.FillAndExpand,
          VerticalOptions = LayoutOptions.FillAndExpand
        }
      }
    };
  }
}
```

实例`CameraPreview`自定义控件用于显示来自设备的摄像机的预览视频流。除了可以选择指定的值`Camera`属性，自定义控件将执行自定义呈现器中。

自定义呈现器现在可以添加到每个应用程序项目来创建特定于平台的相机预览控件。

在每个平台上创建自定义呈现器

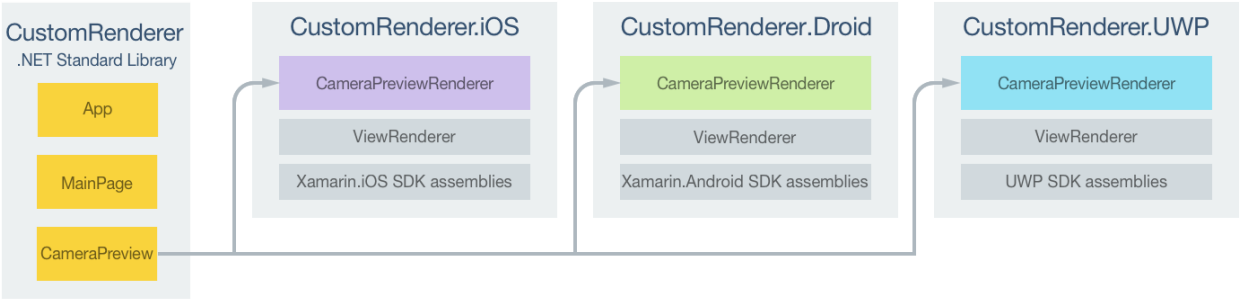
创建自定义呈现器类的过程如下所示：

1. 创建一个子类`ViewRenderer<T1,T2>`呈现自定义控件的类。第一个类型参数应为自定义控件呈现器，在这种情况下是`CameraPreview`。第二个类型参数应将实现自定义控件的本机控件。
2. 重写`OnElementChanged`呈现其进行自定义的自定义控件和写入逻辑的方法。创建相应的 Xamarin.Forms 控件时，调用此方法。
3. 添加`ExportRenderer`到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 自定义控件属性。此属性用于向

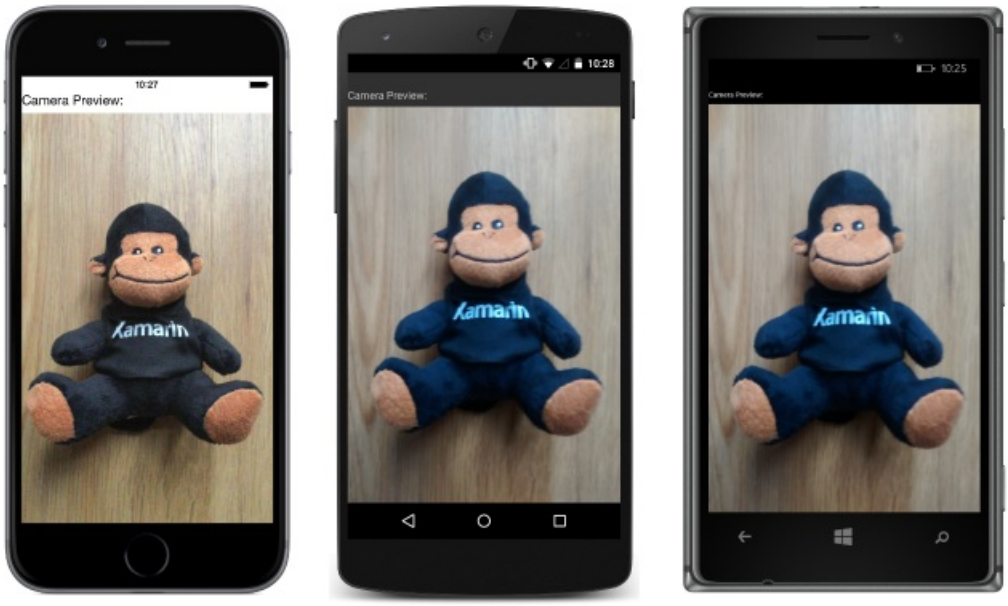
Xamarin.Forms 注册自定义呈现器。

> **NOTE**
>
> 对于大多数 Xamarin.Forms 元素，它是可选提供每个平台项目中的自定义呈现器。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。但是，自定义呈现器呈现时所需的每个平台项目中视图元素。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`CameraPreview` 自定义控件呈现的特定于平台的呈现器类，它们都派生自 `ViewRenderer` 为每个平台的类。这会导致每个 `CameraPreview` 自定义控件呈现与特定于平台的控件，如以下屏幕截图中所示：



`ViewRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 自定义控件来呈现相应的本机控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序，`OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `CameraPreview` 实例。

重写的版本 `OnElementChanged` 中每个特定于平台的呈现器类，方法是执行的本机控件实例化和自定义的位置。`SetNativeControl` 应使用方法来实例化本机控件，此方法还会将分配到的控件引用 `Control` 属性。此外，通过获取对所呈现的 Xamarin.Forms 控件的引用 `Element` 属性。

在某些情况下，`OnElementChanged` 可以多次调用方法。因此，若要防止内存泄漏，必须格外小心实例化新的本机控件时。下面的代码示例中演示了在自定义呈现器中实例化新的本机控件时要使用的方法：

```
protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
  base.OnElementChanged (e);

  if (Control == null) {
    // Instantiate the native control and assign it to the Control property with
    // the SetNativeControl method
  }

  if (e.OldElement != null) {
    // Unsubscribe from event handlers and cleanup any resources
  }

  if (e.NewElement != null) {
    // Configure the control and subscribe to event handlers
  }
}
```

当 `Control` 属性是 `null` 时，新的本机控件只应实例化一次。仅当自定义呈现器附加到新 Xamarin.Forms 元素时，才应配置该控件并订阅事件处理程序。同样，任何已订阅的事件处理程序只应呈现器附加到的元素发生更改时取消订阅。采用此方法将有助于创建不会遭受内存泄漏的高性能自定义呈现器。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 自定义控件的类型名称和自定义呈现器的类型名称。`assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论每个特定于平台的自定义呈现器类的实现。

在 ios 设备上创建自定义呈现器

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
[assembly: ExportRenderer (typeof(CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.iOS
{
    public class CameraPreviewRenderer : ViewRenderer<CameraPreview, UICameraPreview>
    {
        UICameraPreview uiCameraPreview;

        protected override void OnElementChanged (ElementChangedEventArgs<CameraPreview> e)
        {
            base.OnElementChanged (e);

            if (Control == null) {
                uiCameraPreview = new UICameraPreview (e.NewElement.Camera);
                SetNativeControl (uiCameraPreview);
            }
            if (e.OldElement != null) {
                // Unsubscribe
                uiCameraPreview.Tapped -= OnCameraPreviewTapped;
            }
            if (e.NewElement != null) {
                // Subscribe
                uiCameraPreview.Tapped += OnCameraPreviewTapped;
            }
        }

        void OnCameraPreviewTapped (object sender, EventArgs e)
        {
            if (uiCameraPreview.IsPreviewing) {
                uiCameraPreview.CaptureSession.StopRunning ();
                uiCameraPreview.IsPreviewing = false;
            } else {
                uiCameraPreview.CaptureSession.StartRunning ();
                uiCameraPreview.IsPreviewing = true;
            }
        }
        ...
    }
}
```

前提`Control`属性是`null`，则`SetNativeControl`调用方法来实例化一个新`UICameraPreview`控件并将分配给它的引用`Control`属性。`UICameraPreview`控件是使用一个特定于平台的自定义控件`AVCapture` Api 以提供从相机预览流。它公开`Tapped`处理的事件，`OnCameraPreviewTapped`方法来停止和启动视频预览点击它时。`Tapped`自定义呈现器附加到新 Xamarin.Forms 元素，并且已取消订阅仅元素呈现器附加到更改时，对订阅的事件。

在 Android 上创建自定义呈现器

下面的代码示例显示了 Android 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(CustomRenderer.CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.Droid
{
    public class CameraPreviewRenderer : ViewRenderer<CustomRenderer.CameraPreview,
CustomRenderer.Droid.CameraPreview>
    {
        CameraPreview cameraPreview;

        public CameraPreviewRenderer(Context context) : base(context)
        {
        }

        protected override void OnElementChanged(ElementChangedEventArgs<CustomRenderer.CameraPreview> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                cameraPreview = new CameraPreview(Context);
                SetNativeControl(cameraPreview);
            }

            if (e.OldElement != null)
            {
                // Unsubscribe
                cameraPreview.Click -= OnCameraPreviewClicked;
            }
            if (e.NewElement != null)
            {
                Control.Preview = Camera.Open((int)e.NewElement.Camera);

                // Subscribe
                cameraPreview.Click += OnCameraPreviewClicked;
            }
        }

        void OnCameraPreviewClicked(object sender, EventArgs e)
        {
            if (cameraPreview.IsPreviewing)
            {
                cameraPreview.Preview.StopPreview();
                cameraPreview.IsPreviewing = false;
            }
            else
            {
                cameraPreview.Preview.StartPreview();
                cameraPreview.IsPreviewing = true;
            }
        }
        ...
    }
}
```

前提 `Control` 属性是 `null`，则 `SetNativeControl` 调用方法来实例化一个新 `CameraPreview` 控制并为其分配一个引用 `Control` 属性。 `CameraPreview` 控件是使用一个特定于平台的自定义控件 `Camera` API 以提供从相机预览流。 `CameraPreview` 然后配置控制，前提是自定义呈现器附加到新 Xamarin.Forms 元素。此配置涉及到创建一个新的本机 `Camera` 对象，以便访问特定的硬件照相机，并注册事件处理程序来处理 `Click` 事件。反过来此处理程序将停止和启动视频预览，点击它时。 `Click` 如果 Xamarin.Forms 元素呈现器附加到更改从订阅的事件，将取消。

在 UWP 上创建自定义呈现器

下面的代码示例显示了适用于 UWP 的自定义呈现器：

```
[assembly: ExportRenderer(typeof(CameraPreview), typeof(CameraPreviewRenderer))]
namespace CustomRenderer.UWP
{
    public class CameraPreviewRenderer : ViewRenderer<CameraPreview, Windows.UI.Xaml.Controls.CaptureElement>
    {
        ...
        CaptureElement _captureElement;
        bool _isPreviewing;

        protected override void OnElementChanged(ElementChangedEventArgs<CameraPreview> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                ...
                _captureElement = new CaptureElement();
                _captureElement.Stretch = Stretch.UniformToFill;

                SetupCamera();
                SetNativeControl(_captureElement);
            }
            if (e.OldElement != null)
            {
                // Unsubscribe
                Tapped -= OnCameraPreviewTapped;
                ...
            }
            if (e.NewElement != null)
            {
                // Subscribe
                Tapped += OnCameraPreviewTapped;
            }
        }

        async void OnCameraPreviewTapped(object sender, TappedRoutedEventArgs e)
        {
            if (_isPreviewing)
            {
                await StopPreviewAsync();
            }
            else
            {
                await StartPreviewAsync();
            }
        }
        ...
    }
}
```

前提`Control`属性是`null`，一个新`CaptureElement`实例化并`SetupCamera`调用方法时，使用`MediaCapture` API 以提供从相机预览流。`SetNativeControl`然后，调用方法的引用分配`CaptureElement`实例向`Control`属性。`CaptureElement`控件公开`Tapped`处理的事件，`OnCameraPreviewTapped`方法来停止和启动视频预览点击它时。`Tapped`自定义呈现器附加到新 Xamarin.Forms 元素，并且已取消订阅仅元素呈现器附加到更改时，对订阅的事件。

> **NOTE**
>
> 若要停止和释放的对象，它提供对 UWP 应用程序中相机的访问至关重要。如果不这样做可能会影响其他应用程序尝试访问设备的照相机。有关详细信息，请参阅显示摄像头预览。

总结

本文演示了如何创建 Xamarin.Forms 自定义控件用于显示来自设备的摄像机的预览视频流的自定义呈现器。Xamarin.Forms 自定义用户界面控件应派生自 `View` 类，该类用于将布局和屏幕上的控件。

相关链接

- CustomRendererView（示例）

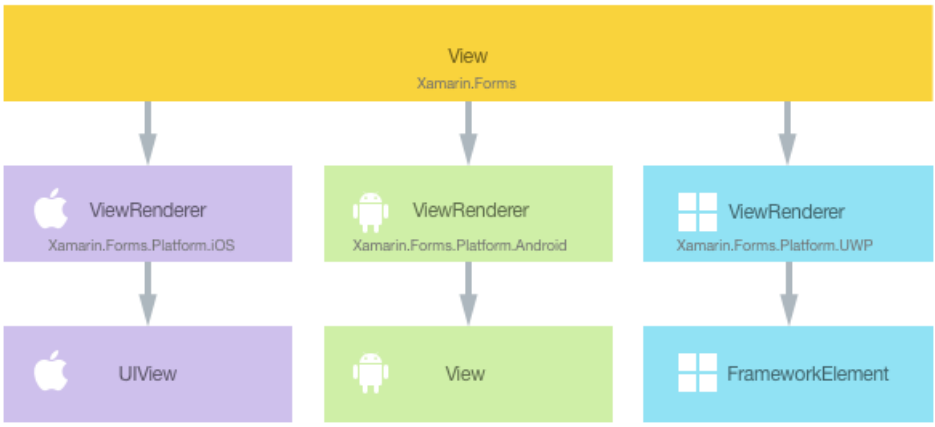
实现 HybridWebView

2018/10/26 • • Edit Online

Xamarin.Forms 自定义用户界面控件应派生视图类，用于放置布局和屏幕上的控件。本文演示如何创建 HybridWebView 自定义控件，该示例演示了如何增强特定于平台的 web 控件，以便 C# 代码调用 JavaScript 中的自定义呈现器。

每个 Xamarin.Forms 视图都会随附的呈现器为每个平台创建本机控件的实例。当 `View` Xamarin.Forms 应用程序在 iOS 中，呈现 `ViewRenderer` 类实例化时，这反过来实例化本机 `UIView` 控件。在 Android 平台上 `ViewRenderer` 类实例化 `View` 控件。在通用 Windows 平台 (UWP)，`ViewRenderer` 类实例化本机 `FrameworkElement` 控件。有关呈现器和 Xamarin.Forms 控件映射到的本机控件类的详细信息，请参阅呈现器基类和本机控件。

下图说明了之间的关系 `View` 和相应的本机控件实现它：



呈现过程可用于通过创建自定义呈现器为实现特定于平台的自定义 `View` 每个平台上。执行此操作的过程如下所示：

1. 创建 `HybridWebView` 自定义控件。
2. 占用 `HybridWebView` 从 Xamarin.Forms。
3. 创建的自定义呈现器 `HybridWebView` 每个平台上。

每个项将现在讨论反过来实现 `HybridWebView` 增强了特定于平台的 web 控件，以便 C# 代码调用 JavaScript 中的呈现器。`HybridWebView` 实例将用于显示要求用户输入其名称的 HTML 页。然后，当用户单击 HTML 按钮，将调用 JavaScript 函数的 C# `Action` 显示一个弹出窗口，其中包含用户名称。

调用 C# 从 JavaScript 中为该过程的详细信息，请参阅调用 C# 从 JavaScript。有关 HTML 页的详细信息，请参阅创建 Web 页。

创建 HybridWebView

`HybridWebView` 可以通过子类化创建自定义控件 `View` 类，如下面的代码示例中所示：

```
public class HybridWebView : View
{
  Action<string> action;
  public static readonly BindableProperty UriProperty = BindableProperty.Create (
    propertyName: "Uri",
    returnType: typeof(string),
    declaringType: typeof(HybridWebView),
    defaultValue: default(string));

  public string Uri {
    get { return (string)GetValue (UriProperty); }
    set { SetValue (UriProperty, value); }
  }

  public void RegisterAction (Action<string> callback)
  {
    action = callback;
  }

  public void Cleanup ()
  {
    action = null;
  }

  public void InvokeAction (string data)
  {
    if (action == null || data == null) {
      return;
    }
    action.Invoke (data);
  }
}
```

`HybridWebView`自定义控件在.NET Standard 库项目中创建和定义控件的以下 API:

- 一个`Uri`属性，它指定要加载的网页的地址。
- 一个`RegisterAction`方法是否注册`Action`与该控件。已注册的操作将从通过引用该 HTML 文件中包含的 JavaScript 调用`Uri`属性。
- 一个`CleanUp`删除对已注册的引用的方法`Action`。
- `InvokeAction`方法调用的已注册`Action`。从每个特定于平台的项目中的自定义呈现器将调用此方法。

使用 HybridWebView

`HybridWebView`自定义控件可以在 XAML 中引用.NET Standard 库项目通过声明其位置的命名空间和自定义控件上使用的命名空间前缀。下面的代码示例演示如何将`HybridWebView`自定义控件可供 XAML 页：

```
<ContentPage ...
             xmlns:local="clr-namespace:CustomRenderer;assembly=CustomRenderer"
             x:Class="CustomRenderer.HybridWebViewPage"
             Padding="0,20,0,0">
    <ContentPage.Content>
        <local:HybridWebView x:Name="hybridWebView" Uri="index.html"
          HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand" />
    </ContentPage.Content>
</ContentPage>
```

`local`命名空间前缀可以命名任何内容。但是，`clr-namespace`和`assembly`值必须匹配的自定义控件的详细信息。一旦声明的命名空间，前缀用于引用自定义控件。

下面的代码示例演示如何将`HybridWebView`自定义控件可供 C# 页：

```
public class HybridWebViewPageCS : ContentPage
{
  public HybridWebViewPageCS ()
  {
    var hybridWebView = new HybridWebView {
      Uri = "index.html",
      HorizontalOptions = LayoutOptions.FillAndExpand,
      VerticalOptions = LayoutOptions.FillAndExpand
    };
    ...
    Padding = new Thickness (0, 20, 0, 0);
    Content = hybridWebView;
  }
}
```

`HybridWebView`实例将用于显示每个平台上的本机 web 控件。它具有`Uri`属性设置为某一 HTML 文件存储在每个特定于平台的项目，并将由本机 web 控件。呈现的 HTML 会要求用户输入其名称，使用 JavaScript 函数调用 C# `Action` HTML 按钮单击响应中。

`HybridWebViewPage`注册要从 JavaScript 调用的操作，如下面的代码示例中所示：

```
public partial class HybridWebViewPage : ContentPage
{
  public HybridWebViewPage ()
  {
    ...
    hybridWebView.RegisterAction (data => DisplayAlert ("Alert", "Hello " + data, "OK"));
  }
}
```

此操作会调用`DisplayAlert`方法来显示模式弹出框，其中的情况下显示的 HTML 页中输入的名称`HybridWebView`实例。

自定义呈现器现在可以添加到每个应用程序项目，以增强特定于平台的 web 控件，因为它允许 C# 代码从 JavaScript 调用。

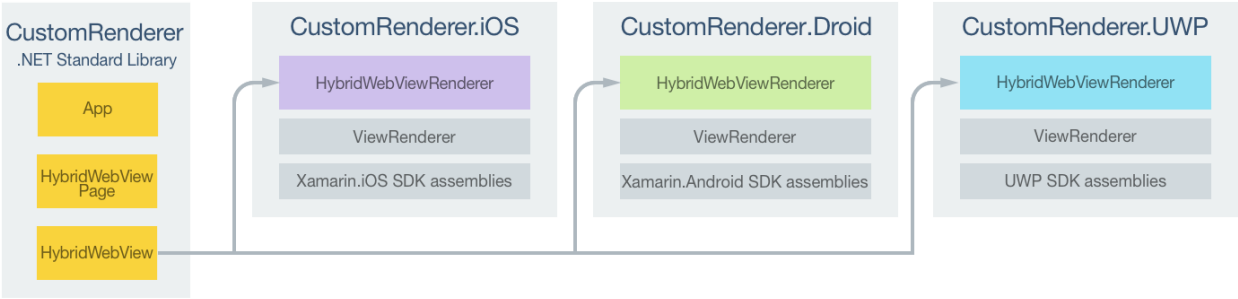
在每个平台上创建自定义呈现器

创建自定义呈现器类的过程如下所示：

1. 创建一个子类`ViewRenderer<T1,T2>`呈现自定义控件的类。第一个类型参数应为自定义控件呈现器，在这种情况下是`HybridWebView`。第二个类型参数应将实现自定义视图的本机控件。
2. 重写`OnElementChanged`呈现其进行自定义的自定义控件和写入逻辑的方法。创建相应的 Xamarin.Forms 自定义控件时，调用此方法。
3. 添加`ExportRenderer`到自定义呈现器类，以指定它将用于呈现 Xamarin.Forms 自定义控件属性。此属性用于向 Xamarin.Forms 注册自定义呈现器。

> **NOTE**
>
> 对于大多数 Xamarin.Forms 元素，它是可选提供每个平台项目中的自定义呈现器。如果未注册的自定义呈现器，则将使用默认的呈现器的控件的基类。但是，自定义呈现器呈现时所需的每个平台项目中视图元素。

下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`HybridWebView` 自定义控件呈现的特定于平台的呈现器类，它们都派生自 `ViewRenderer` 为每个平台的类。这会导致每个 `HybridWebView` 自定义控件呈现与特定于平台的 web 控件，如以下屏幕截图中所示：

`ViewRenderer` 类公开 `OnElementChanged` 创建 Xamarin.Forms 自定义控件以呈现相应的本机 web 控件时调用的方法。此方法采用 `ElementChangedEventArgs` 参数，其中包含 `OldElement` 和 `NewElement` 属性。这些属性表示 Xamarin.Forms 元素的呈现器*已*附加到，和 Xamarin.Forms 元素的呈现器*是*附加到分别。在示例应用程序 `OldElement` 属性将为 `null` 并 `NewElement` 属性将包含对引用 `HybridWebView` 实例。

重写的版本 `OnElementChanged` 中每个特定于平台的呈现器类，方法是执行的本机 web 控件实例化和自定义的位置。`SetNativeControl` 应使用方法进行实例化本机 web 控件，并且此方法还会将分配到的控件引用 `Control` 属性。此外，通过获取对所呈现的 Xamarin.Forms 控件的引用 `Element` 属性。

在某些情况下 `OnElementChanged` 可以多次调用方法。因此，若要防止内存泄漏，必须格外小心实例化新的本机控件时。下面的代码示例中演示了在自定义呈现器中实例化新的本机控件时要使用的方法：

```
protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
  base.OnElementChanged (e);

  if (Control == null) {
    // Instantiate the native control and assign it to the Control property with
    // the SetNativeControl method
  }

  if (e.OldElement != null) {
    // Unsubscribe from event handlers and cleanup any resources
  }

  if (e.NewElement != null) {
    // Configure the control and subscribe to event handlers
  }
}
```

当 `Control` 属性是 `null` 时，新的本机控件只应实例化一次。仅当自定义呈现器附加到新 Xamarin.Forms 元素时，才应配置该控件并订阅事件处理程序。同样，仅当呈现器所附加到的元素更改时，才应取消订阅任何订阅的事件处理程序。采用此方法将有助于创建不会遭受内存泄漏的高性能自定义呈现器。

每个自定义呈现器类用修饰 `ExportRenderer` 与 Xamarin.Forms 结合注册呈现器的属性。该属性采用两个参数 – 正在呈现的 Xamarin.Forms 自定义控件的类型名称和自定义呈现器的类型名称。 `assembly` 到的属性的前缀指定特性应用于整个程序集。

以下各节讨论每个本机 web 控件、的过程调用 C# 从 JavaScript 中，并在每个特定于平台的自定义呈现器类中的此实现由加载 web 页的结构。

创建 Web 页

下面的代码示例显示了将显示的网页 `HybridWebView` 自定义控件：

```
<html>
<body>
<script src="http://code.jquery.com/jquery-1.11.0.min.js"></script>
<h1>HybridWebView Test</h1>
<br/>
Enter name: <input type="text" id="name">
<br/>
<br/>
<button type="button" onclick="javascript:invokeCSCode($('#name').val());">Invoke C# Code</button>
<br/>
<p id="result">Result:</p>
<script type="text/javascript">
function log(str)
{
    $('#result').text($('#result').text() + " " + str);
}

function invokeCSCode(data) {
    try {
        log("Sending Data:" + data);
        invokeCSharpAction(data);
    }
    catch (err){
          log(err);
    }
}
</script>
</body>
</html>
```

在 web 页上，用户输入其名称中的 `input` 元素，并提供 `button` 将调用 C# 代码时单击的元素。实现此目的的过程如下所示：

- 当用户单击 `button` 元素中，`invokeCSCode` 调用 JavaScript 函数时，值为 `input` 元素传递给函数。
- `invokeCSCode` 函数调用 `log` 函数来显示的数据发送到 C# `Action`。然后，它调用 `invokeCSharpAction` 方法以调用 C# `Action`，将从接收到该参数传递 `input` 元素。

`invokeCSharpAction` JavaScript 函数未定义在 web 页中，并将每个自定义呈现器注入到它。

调用 C# 从 JavaScript

调用 C# 从 JavaScript 的过程是在每个平台上完全相同：

- 自定义呈现器创建本机 web 控件，并加载指定的 HTML 文件 `HybridWebView.Uri` 属性。
- 自定义呈现器网页加载后，会注入 `invokeCSharpAction` 到的网页的 JavaScript 函数。
- 当用户输入其名称并单击 HTML `button` 元素中，`invokeCSCode` 调用函数时，这反过来调用 `invokeCSharpAction` 函数。
- `invokeCSharpAction` 函数将调用方法中调用的自定义呈现器 `HybridWebView.InvokeAction` 方法。
- `HybridWebView.InvokeAction` 方法调用的已注册 `Action`。

以下各节将讨论此过程的每个平台上的实现方式。

在 ios 设备上创建自定义呈现器

下面的代码示例显示了为 iOS 平台的自定义呈现器：

```
[assembly: ExportRenderer (typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.iOS
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, WKWebView>, IWKScriptMessageHandler
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data)
{window.webkit.messageHandlers.invokeAction.postMessage(data);}";
        WKUserContentController userController;

        protected override void OnElementChanged (ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged (e);

            if (Control == null) {
                userController = new WKUserContentController ();
                var script = new WKUserScript (new NSString (JavaScriptFunction),
WKUserScriptInjectionTime.AtDocumentEnd, false);
                userController.AddUserScript (script);
                userController.AddScriptMessageHandler (this, "invokeAction");

                var config = new WKWebViewConfiguration { UserContentController = userController };
                var webView = new WKWebView (Frame, config);
                SetNativeControl (webView);
            }
            if (e.OldElement != null) {
                userController.RemoveAllUserScripts ();
                userController.RemoveScriptMessageHandler ("invokeAction");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup ();
            }
            if (e.NewElement != null) {
                string fileName = Path.Combine (NSBundle.MainBundle.BundlePath, string.Format ("Content/{0}",
Element.Uri));
                Control.LoadRequest (new NSUrlRequest (new NSUrl (fileName, false)));
            }
        }

        public void DidReceiveScriptMessage (WKUserContentController userContentController, WKScriptMessage
message)
        {
            Element.InvokeAction (message.Body.ToString ());
        }
    }
}
```

`HybridWebViewRenderer` 类加载 web 页中指定 `HybridWebView.Uri` 属性转换成一个本机 `WKWebView` 控件，并 `invokeCSharpAction` JavaScript 函数注入到 web 页。用户输入其名称并单击 HTML 后 `button` 元素中，`invokeCSharpAction` 执行的 JavaScript 函数时，使用 `DidReceiveScriptMessage` 从网页上收到一条消息后调用的方法。反过来，此方法将调用 `HybridWebView.InvokeAction` 方法，将调用已注册的操作，以显示弹出窗口中。

此功能来实现，如下所示：

- 前提 `Control` 属性是 `null`，执行以下操作：
  - 一个 `WKUserContentController` 创建实例后，它允许将消息发布并将用户脚本注入到 web 页。
  - 一个 `WKUserScript` 创建实例注入 `invokeCSharpAction` 到加载 web 页之后发送的网页的 JavaScript 函数。
  - `WKUserContentController.AddScript` 方法将添加 `WKUserScript` 到内容的控制器实例。
  - `WKUserContentController.AddScriptMessageHandler` 方法将添加名为脚本消息处理程序 `invokeAction` 到 `WKUserContentController` 实例，这将导致 JavaScript 函数 `window.webkit.messageHandlers.invokeAction.postMessage(data)` 中所有定义将使用的所有 web 视图中的帧 `WKUserContentController` 实例。
  - 一个 `WKWebViewConfiguration` 创建实例后，使用 `WKUserContentController` 实例设置为内容的控制器。

    - 一个 `WKWebView` 实例化控件，并 `SetNativeControl` 调用方法来分配对引用 `WKWebView` 控制对 `Control` 属性。
- 提供自定义呈现器附加到新 Xamarin.Forms 元素：
    - `WKWebView.LoadRequest` 方法将加载指定的 HTML 文件 `HybridWebView.Uri` 属性。该代码指定该文件存储在 `Content` 项目文件夹中的。后显示的网页， `invokeCSharpAction` JavaScript 函数将注入到 web 页。
- 当元素呈现器附加到的更改：
    - 释放资源。

> **NOTE**
>
> `WKWebView` 类仅支持 iOS 8 及更高版本。

在 Android 上创建自定义呈现器

下面的代码示例显示了 Android 平台的自定义呈现器：

```
[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.Droid
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, Android.Webkit.WebView>
    {
        const string JavascriptFunction = "function invokeCSharpAction(data){jsBridge.invokeAction(data);}";
        Context _context;

        public HybridWebViewRenderer(Context context) : base(context)
        {
            _context = context;
        }

        protected override void OnElementChanged(ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                var webView = new Android.Webkit.WebView(_context);
                webView.Settings.JavaScriptEnabled = true;
                webView.SetWebViewClient(new JavascriptWebViewClient($"javascript: {JavascriptFunction}"));
                SetNativeControl(webView);
            }
            if (e.OldElement != null)
            {
                Control.RemoveJavascriptInterface("jsBridge");
                var hybridWebView = e.OldElement as HybridWebView;
                hybridWebView.Cleanup();
            }
            if (e.NewElement != null)
            {
                Control.AddJavascriptInterface(new JSBridge(this), "jsBridge");
                Control.LoadUrl($"file:///android_asset/Content/{Element.Uri}");
            }
        }
    }
}
```

`HybridWebViewRenderer` 类加载 web 页中指定 `HybridWebView.Uri` 属性转换成一个本机 `WebView` 控件，并 `invokeCSharpAction` JavaScript 函数注入到网页，网页后完成加载后， `OnPageFinished` 重写中 `JavascriptWebViewClient` 类：

```
public class JavascriptWebViewClient : WebViewClient
{
    string _javascript;

    public JavascriptWebViewClient(string javascript)
    {
        _javascript = javascript;
    }

    public override void OnPageFinished(WebView view, string url)
    {
        base.OnPageFinished(view, url);
        view.EvaluateJavascript(_javascript, null);
    }
}
```

用户输入其名称并单击 HTML 后 `button` 元素，`invokeCSharpAction` 执行 JavaScript 函数。此功能来实现，如下所示：

- 前提 `Control` 属性是 `null`，执行以下操作：
  - 一个本机 `WebView` 创建实例，在该控件中启用 JavaScript 和一个 `JavascriptWebViewClient` 实例设置的实现为 `WebViewClient`。
  - `SetNativeControl` 调用方法来分配对本机引用 `WebView` 控制对 `Control` 属性。
- 提供自定义呈现器附加到新 Xamarin.Forms 元素：
  - `WebView.AddJavascriptInterface` 方法将一个新中注入 `JSBridge` WebView 的 JavaScript 上下文，其命名的实例到主框架 `jsBridge`。这允许在方法 `JSBridge` 类从 JavaScript 访问。
  - `WebView.LoadUrl` 方法将加载指定的 HTML 文件 `HybridWebView.Uri` 属性。该代码指定该文件存储在 `Content` 项目文件夹中的。
  - 在中 `JavascriptWebViewClient` 类，`invokeCSharpAction` 页面加载完毕后，JavaScript 函数注入到 web 页。
- 当元素呈现器附加到的更改：
  - 释放资源。

当 `invokeCSharpAction` 执行的 JavaScript 函数，它又调用 `JSBridge.InvokeAction` 方法，在下面的代码示例所示：

```
public class JSBridge : Java.Lang.Object
{
  readonly WeakReference<HybridWebViewRenderer> hybridWebViewRenderer;

  public JSBridge (HybridWebViewRenderer hybridRenderer)
  {
    hybridWebViewRenderer = new WeakReference <HybridWebViewRenderer> (hybridRenderer);
  }

  [JavascriptInterface]
  [Export ("invokeAction")]
  public void InvokeAction (string data)
  {
    HybridWebViewRenderer hybridRenderer;

    if (hybridWebViewRenderer != null && hybridWebViewRenderer.TryGetTarget (out hybridRenderer))
    {
      hybridRenderer.Element.InvokeAction (data);
    }
  }
}
```

类必须派生自 `Java.Lang.Object`，并向 JavaScript 公开的方法必须使用修饰 `[JavascriptInterface]` 和 `[Export]` 属性。因此，当 `invokeCSharpAction` 注入到 web 页和执行 JavaScript 功能，它将调用 `JSBridge.InvokeAction` 方法，因

为正在使用修饰 `[JavascriptInterface]` 和 `[Export("invokeAction")]` 属性。依次 `InvokeAction` 方法将调用 `HybridWebView.InvokeAction` 方法，它将调用已注册的操作，以显示弹出窗口中。

> **NOTE**
> 项目使用 `[Export]` 属性必须包含对引用 `Mono.Android.Export`，或将导致编译器错误。

请注意，`JSBridge` 类维护 `WeakReference` 到 `HybridWebViewRenderer` 类。这是为了避免创建两个类之间的循环引用。有关详细信息请参阅弱引用MSDN 上。

在 UWP 上创建自定义呈现器

下面的代码示例显示了适用于 UWP 的自定义呈现器:

```
[assembly: ExportRenderer(typeof(HybridWebView), typeof(HybridWebViewRenderer))]
namespace CustomRenderer.UWP
{
    public class HybridWebViewRenderer : ViewRenderer<HybridWebView, Windows.UI.Xaml.Controls.WebView>
    {
        const string JavaScriptFunction = "function invokeCSharpAction(data){window.external.notify(data);}";

        protected override void OnElementChanged(ElementChangedEventArgs<HybridWebView> e)
        {
            base.OnElementChanged(e);

            if (Control == null)
            {
                SetNativeControl(new Windows.UI.Xaml.Controls.WebView());
            }
            if (e.OldElement != null)
            {
                Control.NavigationCompleted -= OnWebViewNavigationCompleted;
                Control.ScriptNotify -= OnWebViewScriptNotify;
            }
            if (e.NewElement != null)
            {
                Control.NavigationCompleted += OnWebViewNavigationCompleted;
                Control.ScriptNotify += OnWebViewScriptNotify;
                Control.Source = new Uri(string.Format("ms-appx-web:///Content//{0}", Element.Uri));
            }
        }

        async void OnWebViewNavigationCompleted(WebView sender, WebViewNavigationCompletedEventArgs args)
        {
            if (args.IsSuccess)
            {
                // Inject JS script
                await Control.InvokeScriptAsync("eval", new[] { JavaScriptFunction });
            }
        }

        void OnWebViewScriptNotify(object sender, NotifyEventArgs e)
        {
            Element.InvokeAction(e.Value);
        }
    }
}
```

`HybridWebViewRenderer` 类加载 web 页中指定 `HybridWebView.Uri` 属性转换成一个本机 `WebView` 控件，并 `invokeCSharpAction` JavaScript 函数注入到 web 页上，加载 web 页之后，使用 `WebView.InvokeScriptAsync` 方法。用户输入其名称并单击 HTML 后 `button` 元素中，`invokeCSharpAction` 执行的 JavaScript 函数时，使用 `OnWebViewScriptNotify` 从网页上收到通知后被调用方法。反过来，此方法将调用 `HybridWebView.InvokeAction` 方法，

将调用已注册的操作，以显示弹出窗口中。

此功能来实现，如下所示：

- 前提 `Control` 属性是 `null`，执行以下操作：
  - `SetNativeControl` 调用方法来实例化一个新的本机 `WebView` 控件并为其分配一个引用 `Control` 属性。
- 提供自定义呈现器附加到新 Xamarin.Forms 元素：
  - 事件处理程序 `NavigationCompleted` 和 `ScriptNotify` 注册事件。 `NavigationCompleted` 时将激发事件位于本机 `WebView` 控件完成加载当前内容或导航已失败。 `ScriptNotify` 事件时将触发在本机内容 `WebView` 控件使用 JavaScript 来将字符串传递给应用程序。Web 页面触发 `ScriptNotify` 通过调用事件 `window.external.notify` 而传递 `string` 参数。
  - `WebView.Source` 属性设置为指定的 HTML 文件的 URI `HybridWebView.Uri` 属性。此代码假定该文件存储在 `Content` 项目文件夹中的。显示的网页，一旦 `NavigationCompleted` 会触发事件和 `OnWebViewNavigationCompleted` 将调用方法。 `invokeCSharpAction` JavaScript 函数然后会注入与网页 `WebView.InvokeScriptAsync` 方法，前提是已成功完成的导航。
- 当元素呈现器附加到的更改：
  - 事件已取消订阅。

总结

本文演示了如何创建自定义呈现器 `HybridWebView` 演示了如何增强特定于平台的 web 控件，以便 C# 代码调用 JavaScript 中的自定义的控件。

相关链接

- CustomRendererHybridWebView（示例）
- 从 JavaScript 中调用 C#

实现视频播放器

2018/6/9 • • Edit Online

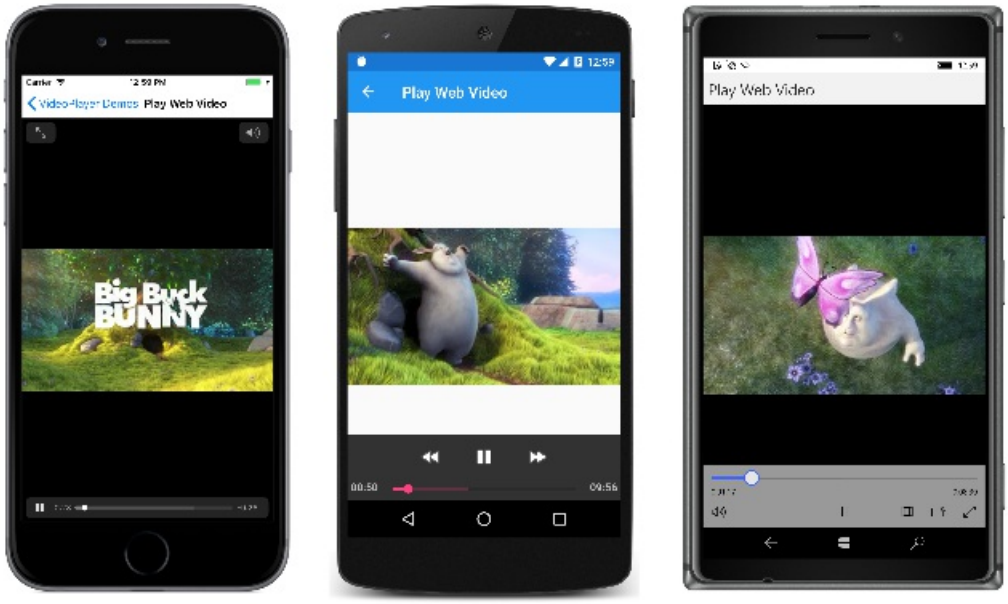
因此，有时需要播放在 Xamarin.Forms 应用程序中的视频文件。此系列文章讨论如何编写适用于 iOS、Android 和一个名为的 Xamarin.Forms 类通用 Windows 平台 (UWP) 的自定义呈现器 `VideoPlayer`。

在 **VideoPlayerDemos** 取样，请实现并支持所有文件 `VideoPlayer` 在文件夹中名为 `FormsVideoLibrary` 使用命名空间的标识 `FormsVideoLibrary` 或命名空间开始 `FormsVideoLibrary`。此组织命名应进行轻松地将视频播放器文件复制到 Xamarin.Forms 解决方案。

`VideoPlayer` 可以播放从三种类型的源的视频文件：

- 使用 URL 在 Internet
- 在平台应用程序中嵌入的资源
- 设备的视频库

视频播放器需要*传输控件*、它们用于播放和暂停视频中，按钮和定位栏，显示了通过视频的进度，并允许用户快速跳到不同的位置。`VideoPlayer` 可以使用传输控件和定位栏提供的平台（如下所示），也可以提供自定义传输控件和定位栏。此处是在 iOS、Android 和通用 Windows 平台下运行的程序：



当然，你可以打开手机上，以进行较大的视图。

更复杂的视频播放器将具有一些其他功能，如卷控制、电话呼叫时，通过中断视频的机制和一种在播放期间保持屏幕活动。

以下一系列文章渐进式演示如何生成的平台呈现器和支持类：

创建平台视频播放器

每个平台需要 `VideoPlayerRenderer` 类创建和维护包含平台支持的视频播放器控件。这篇文章演示呈现器的结构类，以及如何创建播放器。

播放 Web 视频

可能的视频的视频播放器最常见来源是 Internet。本文介绍如何引用和用作视频播放器源 Web 视频。

绑定到播放器的视频源

本文章将使用`ListView`来呈现视频播放的集合。一个程序显示的代码隐藏文件如何可以设置视频源的视频播放器，但第二个程序演示如何使用数据绑定之间`ListView`和视频播放器。

加载应用程序资源视频

在平台项目中，可以作为资源嵌入视频。这篇文章演示如何存储这些资源和更高版本将它们加载到程序中要播放的视频播放器。

访问设备的视频库

使用设备的照相机创建视频时，视频文件存储在设备的映像库中。这篇文章演示如何访问设备的图像选取器，以便选择的视频，并随后使用视频播放器播放。

自定义视频传输控件

尽管每个平台上的视频播放器提供自己的按钮的窗体中的传输控制播放和暂停，可以禁止显示这些按钮，还可以提供你自己。本文介绍如何。

自定义视频定位

每个平台视频播放器都有一个位置栏，显示视频的进度，并允许你以跳到特定位置的向前或向后。本文演示如何可以将该位置栏替换为自定义控件。

相关链接

- 视频播放器演示（示例）

创建平台视频播放器

2018/6/9 • • Edit Online

VideoPlayerDemos 解决方案包含所有代码，以实现 Xamarin.Forms 视频播放器。它还包括演示如何使用视频播放器应用程序中的一系列页面。所有 `VideoPlayer` 代码和其平台呈现器位于项目文件夹名为 `FormsVideoLibrary`，并且也使用命名空间 `FormsVideoLibrary`。这应使轻松地将文件复制到自己的应用程序和引用类。

视频播放器

`VideoPlayer` 类是一部分**VideoPlayerDemos**在平台间共享的标准.NET 库。它派生自 `View`：

```
using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
    }
}
```

此类的成员 (和 `IVideoPlayerController` 接口) 按照文章所述。

每三个平台都包含一个名为类 `VideoPlayerRenderer`，其中包含特定于平台的代码来实现视频播放器。此呈现器的主要任务是创建该平台的视频播放器。

IOS 播放器视图控制器

在 iOS 中实现视频播放器时涉及多个类。应用程序首次创建 `AVPlayerViewController`，然后将设置 `Player` 类型的对象属性 `AVPlayer`。播放机分配的视频源时，其他类是必需的。

如所有呈现器，iOS `VideoPlayerRenderer` 包含 `ExportRenderer` 该属性标识 `VideoPlayer` 与呈现器的视图：

```
using System;
using System.ComponentModel;
using System.IO;

using AVFoundation;
using AVKit;
using CoreMedia;
using Foundation;
using UIKit;

using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
                          typeof(FormsVideoLibrary.iOS.VideoPlayerRenderer))]

namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
    }
}
```

设置的平台控件的呈现器通常派生自 `ViewRenderer<View, NativeView>` 类，其中 `View` 方法是 Xamarin.Forms `View` 派生（在这种情况下，`VideoPlayer`）和 `NativeView` 是 iOS `UIView` 呈现器类派生。对于此呈现器，该泛型自变量只需设置为 `UIView`，很快就会看到的原因。

当呈现器根据 `UIViewController` 派生（与此是），则应重写类 `ViewController` 属性，并返回视图控制器，在此情况下 `AVPlayerViewController`。它是的用途 `_playerViewController` 字段：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        AVPlayerItem playerItem;
        AVPlayerViewController _playerViewController;       // solely for ViewController property

        public override UIViewController ViewController => _playerViewController;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    // Create AVPlayerViewController
                    _playerViewController = new AVPlayerViewController();

                    // Set Player property to AVPlayer
                    player = new AVPlayer();
                    _playerViewController.Player = player;

                    // Use the View from the controller as the native control
                    SetNativeControl(_playerViewController.View);
                }
                ...
            }
        }
        ...
    }
}
```

主要责任 `OnElementChanged` 替代是检查如果 `Control` 属性是 `null` 并且，如果是这样，创建平台控件，并将其传递到 `SetNativeControl` 方法。在这种情况下，该对象是仅可从 `View` 属性 `AVPlayerViewController` 。 `UIView` 派生恰好是一个名为的专用类 `AVPlayerView` ，但因为它是私有的不能将它显式指定为第二个泛型参数 `ViewRenderer` 。

通常 `Control` 呈现器类的属性此后指 `UIView` 用来实现呈现器，但在这种情况下 `Control` 未在其他位置使用属性。

Android 视频视图

Android 呈现器 `VideoPlayer` 基于 Android `VideoView` 类。但是，如果 `VideoView` 可供本身用于播放的区域分配的在 Xamarin.Forms 应用中，视频填充视频 `VideoPlayer` 而无需维护正确的纵横比。此原因（如很快就会看到），`VideoView` 进行的 Android 子 `RelativeLayout` 。A `using` 指令定义 `ARelativeLayout` 区分开来 Xamarin.Forms `RelativeLayout` ，和第二个泛型参数，位于 `ViewRenderer` ：

```
using System;
using System.ComponentModel;
using System.IO;

using Android.Content;
using Android.Media;
using Android.Widget;
using ARelativeLayout = Android.Widget.RelativeLayout;

using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
                          typeof(FormsVideoLibrary.Droid.VideoPlayerRenderer))]

namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        public VideoPlayerRenderer(Context context) : base(context)
        {
        }
        ...
    }
}
```

从 Xamarin.Forms 2.5 开始，Android 呈现器应包括的构造函数 `Context` 自变量。

`OnElementChanged` 重写创建同时 `VideoView` 和 `RelativeLayout` 和设置的布局参数 `VideoView` 以使其内居中 `RelativeLayout` 。

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        MediaController mediaController;    // Used to display transport controls
        bool isPrepared;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    // Save the VideoView for future reference
                    videoView = new VideoView(Context);

                    // Put the VideoView in a RelativeLayout
                    ARelativeLayout relativeLayout = new ARelativeLayout(Context);
                    relativeLayout.AddView(videoView);

                    // Center the VideoView in the RelativeLayout
                    ARelativeLayout.LayoutParams layoutParams =
                        new ARelativeLayout.LayoutParams(LayoutParams.MatchParent, LayoutParams.MatchParent);
                    layoutParams.AddRule(LayoutRules.CenterInParent);
                    videoView.LayoutParameters = layoutParams;

                    // Handle a VideoView event
                    videoView.Prepared += OnVideoViewPrepared;

                    // Use the RelativeLayout as the native control
                    SetNativeControl(relativeLayout);
                }
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null && videoView != null)
            {
                videoView.Prepared -= OnVideoViewPrepared;
            }
            base.Dispose(disposing);
        }

        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            isPrepared = true;
            ...
        }
        ...
    }
}
```

处理程序 `Prepared` 此方法中附加和分离中事件 `Dispose` 方法。此事件激发时 `VideoView` 有足够的信息，以开始播放视频文件。

UWP 媒体元素

在通用 Windows 平台 (UWP)，最常见的视频播放器是 `MediaElement` 。该文档的 `MediaElement` 指示 `MediaPlayerElement` 应改为使用时才需要支持版本的 Windows 10 版本 1607年开头。

`OnElementChanged` 替代需要创建 `MediaElement`、设置几个事件处理程序，并将传递 `MediaElement` 对象传递给 `SetNativeControl`：

```
using System;
using System.ComponentModel;

using Windows.Storage;
using Windows.Storage.Streams;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;

using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ExportRenderer(typeof(FormsVideoLibrary.VideoPlayer),
                          typeof(FormsVideoLibrary.UWP.VideoPlayerRenderer))]

namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    MediaElement mediaElement = new MediaElement();
                    SetNativeControl(mediaElement);

                    mediaElement.MediaOpened += OnMediaElementMediaOpened;
                    mediaElement.CurrentStateChanged += OnMediaElementCurrentStateChanged;
                }
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null)
            {
                Control.MediaOpened -= OnMediaElementMediaOpened;
                Control.CurrentStateChanged -= OnMediaElementCurrentStateChanged;
            }

            base.Dispose(disposing);
        }
        ...
    }
}
```

两个事件处理程序中分离 `Dispose` 呈现器的事件。

显示传输控件

包含三个平台中的所有视频播放器支持一组默认的传输包括用于播放和暂停，以及一个栏，以指示视频中中的当前位置，并将移到新位置的按钮的控件。

`VideoPlayer` 类定义一个名为属性 `AreTransportControlsEnabled` 并将默认值设置为 `true`：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // AreTransportControlsEnabled property
        public static readonly BindableProperty AreTransportControlsEnabledProperty =
            BindableProperty.Create(nameof(AreTransportControlsEnabled), typeof(bool), typeof(VideoPlayer),
true);

        public bool AreTransportControlsEnabled
        {
            set { SetValue(AreTransportControlsEnabledProperty, value); }
            get { return (bool)GetValue(AreTransportControlsEnabledProperty); }
        }
        ...
    }
}
```

尽管此属性同时具有 `set` 和 `get` 访问器，呈现器必须处理情况下，仅在设置属性。`get` 访问器只返回该属性的当前值。

属性，如 `AreTransportControlsEnabled` 平台呈现器两种方式处理：

- 第一次是当 Xamarin.Forms 创建 `VideoPlayer` 元素。这指示在 `OnElementChanged` 呈现器的重写时 `NewElement` 属性不是 `null`。在此期间，呈现器可以设置中定义为从该属性的初始值自己平台视频播放器 `VideoPlayer`。
- 如果中的属性 `VideoPlayer` 更高版本发生更改，则 `OnElementPropertyChanged` 调用中呈现器的方法。这样的呈现器，更新基于新的属性设置的平台视频播放器。

下面是如何 `AreTransportControlsEnabled` 属性处理三个平台中：

iOS 播放控件

IOS 的属性 `AVPlayerViewController` 控制显示的传输控制是 `ShowsPlaybackControls`。下面是如何在 iOS 中设置该属性 `VideoViewRenderer`：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        AVPlayerViewController _playerViewController;       // solely for ViewController property

        public override UIViewController ViewController => _playerViewController;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            ((AVPlayerViewController)ViewController).ShowsPlaybackControls =
Element.AreTransportControlsEnabled;
        }
        ...
    }
}
```

`Element` 呈现器的属性是指 `VideoPlayer` 类。

Android 媒体控制器

在 Android 中，显示传输控件需要创建 `MediaController` 对象并将其与关联 `VideoView` 对象。机制所示 `SetAreTransportControlsEnabled` 方法：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        MediaController mediaController;    // Used to display transport controls
        bool isPrepared;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            if (Element.AreTransportControlsEnabled)
            {
                mediaController = new MediaController(Context);
                mediaController.SetMediaPlayer(videoView);
                videoView.SetMediaController(mediaController);
            }
            else
            {
                videoView.SetMediaController(null);

                if (mediaController != null)
                {
                    mediaController.SetMediaPlayer(null);
                    mediaController = null;
                }
            }
        }
        ...
    }
}
```

UWP 传输控件属性

UWP `MediaElement` 定义名为的属性 `AreTransportControlsEnabled` ，以便从设置属性 `VideoPlayer` 具有相同名称的属性：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetAreTransportControlsEnabled();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(sender, args);

            if (args.PropertyName == VideoPlayer.AreTransportControlsEnabledProperty.PropertyName)
            {
                SetAreTransportControlsEnabled();
            }
            ...
        }

        void SetAreTransportControlsEnabled()
        {
            Control.AreTransportControlsEnabled = Element.AreTransportControlsEnabled;
        }
        ...
    }
}
```

一个属性是开始播放视频所需：这一点至关重要`Source`属性，引用视频文件。实现`Source`属性描述在下一步的文章中，播放 Web 视频。

相关链接

- 视频播放器演示（示例）

播放 Web 视频

2018/7/13 • • Edit Online

`VideoPlayer` 类定义 `Source` 属性用于指定的源视频文件，以及 `AutoPlay` 属性。 `AutoPlay` 其默认设置为 `true`，这意味着视频应开始播放之后自动 `Source` 已设置：

```
using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Source property
        public static readonly BindableProperty SourceProperty =
            BindableProperty.Create(nameof(Source), typeof(VideoSource), typeof(VideoPlayer), null);

        [TypeConverter(typeof(VideoSourceConverter))]
        public VideoSource Source
        {
            set { SetValue(SourceProperty, value); }
            get { return (VideoSource)GetValue(SourceProperty); }
        }

        // AutoPlay property
        public static readonly BindableProperty AutoPlayProperty =
            BindableProperty.Create(nameof(AutoPlay), typeof(bool), typeof(VideoPlayer), true);

        public bool AutoPlay
        {
            set { SetValue(AutoPlayProperty, value); }
            get { return (bool)GetValue(AutoPlayProperty); }
        }
        ...
    }
}
```

`Source` 属性属于类型 `VideoSource`，这模仿 Xamarin.Forms `ImageSource` 抽象类，并且其三个派生 `UriImageSource`， `FileImageSource`，并 `StreamImageSource` 。没有流选项仅适用于 `VideoPlayer` 但是，因为 iOS 和 Android 不支持播放从流视频。

视频源

抽象 `VideoSource` 类只包含三个实例化派生自的三个类的静态方法 `VideoSource`：

```
namespace FormsVideoLibrary
{
    [TypeConverter(typeof(VideoSourceConverter))]
    public abstract class VideoSource : Element
    {
        public static VideoSource FromUri(string uri)
        {
            return new UriVideoSource() { Uri = uri };
        }

        public static VideoSource FromFile(string file)
        {
            return new FileVideoSource() { File = file };
        }

        public static VideoSource FromResource(string path)
        {
            return new ResourceVideoSource() { Path = path };
        }
    }
}
```

`UriVideoSource`类用于与 URI 指定一个可下载的视频文件。它定义类型的单个属性`string`:

```
namespace FormsVideoLibrary
{
    public class UriVideoSource : VideoSource
    {
        public static readonly BindableProperty UriProperty =
            BindableProperty.Create(nameof(Uri), typeof(string), typeof(UriVideoSource));

        public string Uri
        {
            set { SetValue(UriProperty, value); }
            get { return (string)GetValue(UriProperty); }
        }
    }
}
```

处理类型的对象`UriVideoSource`如下所述。

`ResourceVideoSource`类用于访问作为平台应用程序还使用指定的嵌入资源存储的视频文件`string`属性:

```
namespace FormsVideoLibrary
{
    public class ResourceVideoSource : VideoSource
    {
        public static readonly BindableProperty PathProperty =
            BindableProperty.Create(nameof(Path), typeof(string), typeof(ResourceVideoSource));

        public string Path
        {
            set { SetValue(PathProperty, value); }
            get { return (string)GetValue(PathProperty); }
        }
    }
}
```

处理类型的对象`ResourceVideoSource`文章中介绍了加载应用程序资源视频。`VideoPlayer`类具有任何工具可以加载为.NET Standard 库中的资源存储的视频文件。

`FileVideoSource`类用于从设备的视频库访问视频文件。单个属性的类型也是`string`:

```
namespace FormsVideoLibrary
{
    public class FileVideoSource : VideoSource
    {
        public static readonly BindableProperty FileProperty =
                  BindableProperty.Create(nameof(File), typeof(string), typeof(FileVideoSource));

        public string File
        {
            set { SetValue(FileProperty, value); }
            get { return (string)GetValue(FileProperty); }
        }
    }
}
```

处理类型的对象 `FileVideoSource` 文章中介绍了访问设备的视频库。

`VideoSource` 类包括 `TypeConverter` 引用的属性 `VideoSourceConverter`：

```
namespace FormsVideoLibrary
{
    [TypeConverter(typeof(VideoSourceConverter))]
    public abstract class VideoSource : Element
    {
        ...
    }
}
```

调用此类型转换器时 `Source` 属性设置为在 XAML 中的字符串。下面是 `VideoSourceConverter` 类：

```
namespace FormsVideoLibrary
{
    public class VideoSourceConverter : TypeConverter
    {
        public override object ConvertFromInvariantString(string value)
        {
            if (!String.IsNullOrWhiteSpace(value))
            {
                Uri uri;
                return Uri.TryCreate(value, UriKind.Absolute, out uri) && uri.Scheme != "file" ?
                                VideoSource.FromUri(value) : VideoSource.FromResource(value);
            }

            throw new InvalidOperationException("Cannot convert null or whitespace to ImageSource");
        }
    }
}
```

`ConvertFromInvariantString` 方法尝试将字符串转换为 `Uri` 对象。如果成功，并且该方案不是 `file:`，则该方法返回 `UriVideoSource`。否则，它将返回 `ResourceVideoSource`。

设置视频源

单个平台呈现器被实现所有其他逻辑相关的视频源。以下部分介绍了如何平台呈现器播放视频时 `Source` 属性设置为 `UriVideoSource` 对象。

IOS 视频源

两个部分 `VideoPlayerRenderer` 参与视频播放器将视频源设置。Xamarin.Forms 首先创建类型的对象 `VideoPlayer`，则 `OnElementChanged` 方法调用与 `NewElement` 自变量对象的属性设置为的 `VideoPlayer`。 `OnElementChanged` 方法调用

`SetSource`：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            ...
        }
        ...
    }
}
```

更高版本上，当`Source`更改属性时，`OnElementPropertyChanged`方法调用与`PropertyName`"源"属性和`SetSource`再次调用。

若要播放视频文件在 iOS 中，类型的对象`AVAsset`首次创建封装该视频文件，并用于创建`AVPlayerItem`，其中然后交给`AVPlayer`对象。下面是如何`SetSource`方法将处理`Source`属性的类型时`UriVideoSource`：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        AVPlayerItem playerItem;
        ...
        void SetSource()
        {
            AVAsset asset = null;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    asset = AVAsset.FromUrl(new NSUrl(uri));
                }
            }
            ...
            if (asset != null)
            {
                playerItem = new AVPlayerItem(asset);
            }
            else
            {
                playerItem = null;
            }

            player.ReplaceCurrentItemWithPlayerItem(playerItem);

            if (playerItem != null && Element.AutoPlay)
            {
                player.Play();
            }
        }
        ...
    }
}
```

`AutoPlay`属性中的 iOS 视频类，具有不相似之处，因此属性检查的末尾`SetSource`方法来调用`Play`方法`AVPlayer`对象。

在某些情况下，视频继续与页之后播放`VideoPlayer`导航回主页。若要停止视频中，`ReplaceCurrentItemWithPlayerItem`还在设置`Dispose`重写：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void Dispose(bool disposing)
        {
            base.Dispose(disposing);

            if (player != null)
            {
                player.ReplaceCurrentItemWithPlayerItem(null);
            }
        }
        ...
    }
}
```

Android 的视频源

Android `VideoPlayerRenderer` 需要在设置播放机的视频源时 `VideoPlayer` 是首次创建及更高版本时 `Source` 属性更改：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                ...
            }
        }
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            ...
        }
        ...
    }
}
```

`SetSource` 方法可处理类型的对象 `UriVideoSource` 通过调用 `SetVideoUri` 上 `VideoView` 与 Android `Uri` 从 URI 的字符串创建对象。 `Uri` 类完全限定此处以便将其从.NET `Uri` 类：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    videoView.SetVideoURI(Android.Net.Uri.Parse(uri));
                    hasSetSource = true;
                }
            }
            ...

            if (hasSetSource && Element.AutoPlay)
            {
                videoView.Start();
            }
        }
        ...
    }
}
```

Android `VideoView` 无相应 `AutoPlay` 属性，因此 `Start` 调用方法时如果已设置新视频。

如果存在是有区别的行为的 ios 和 Android 的呈现器 `Source` 属性 `VideoPlayer` 设置为 `null`，或者如果 `Uri` 属性 `UriVideoSource` 设置为 `null` 或空白字符串。如果 iOS 视频播放器当前播放的视频，并 `Source` 设置为 `null`(或字符串是 `null` 或保留为空)，`ReplaceCurrentItemWithPlayerItem` 使用调用 `null` 值。当前的视频将被替换，并停止播放。

Android 不支持类似的工具。如果 `Source` 属性设置为 `null`，则 `SetSource` 方法只需忽略它，并将持续播放，当前的视频。

UWP 视频源

UWP `MediaElement` 定义 `AutoPlay` 属性，它在与任何其他属性一样呈现器中处理：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                SetSource();
                SetAutoPlay();
                ...
            }
        }

        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.SourceProperty.PropertyName)
            {
                SetSource();
            }
            else if (args.PropertyName == VideoPlayer.AutoPlayProperty.PropertyName)
            {
                SetAutoPlay();
            }
            ...
        }
        ...
    }
}
```

`SetSource`属性句柄`UriVideoSource`通过设置对象`Source`的属性`MediaElement`到.NET`Uri`值，或设置为`null`如果`Source`属性`VideoPlayer`设置为`null`：

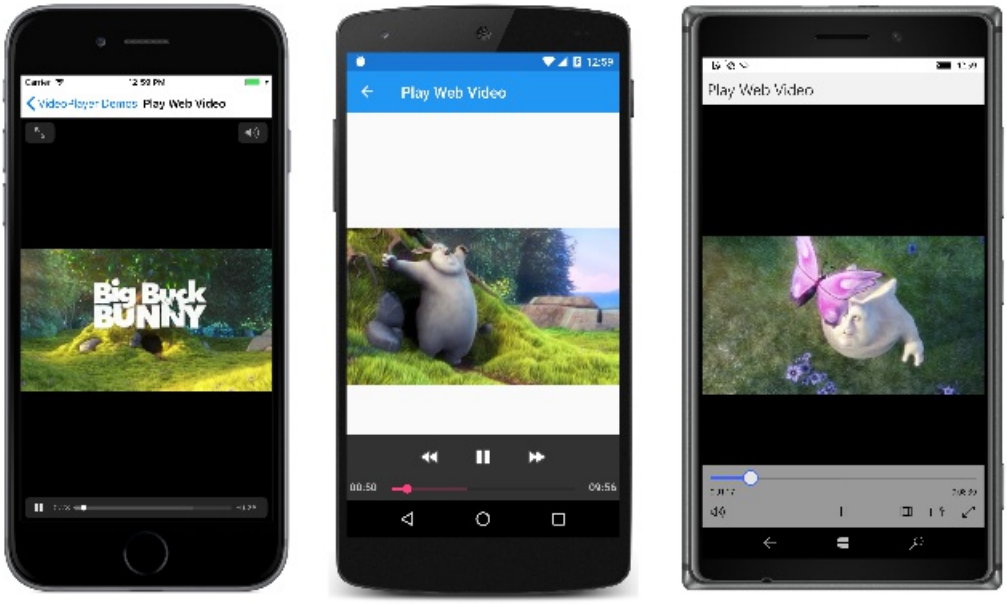
```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        async void SetSource()
        {
            bool hasSetSource = false;

            if (Element.Source is UriVideoSource)
            {
                string uri = (Element.Source as UriVideoSource).Uri;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    Control.Source = new Uri(uri);
                    hasSetSource = true;
                }
            }
            ...
            if (!hasSetSource)
            {
                Control.Source = null;
            }
        }

        void SetAutoPlay()
        {
            Control.AutoPlay = Element.AutoPlay;
        }
        ...
    }
}
```

URL 源设置

使用三个呈现器中的这些属性的实现，就可能要播放的视频 URL 的来源。**播放 Web 视频**页面 **VideoPlayDemos** 程序由下面的 XAML 文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.PlayWebVideoPage"
             Title="Play Web Video">

    <video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />

</ContentPage>
```

`VideoSourceConverter` 类将字符串转换为 `UriVideoSource`。导航到**播放 Web 视频**页上，视频开始加载和播放下载足够数量的数据并将其缓冲时启动。视频为大约 10 分钟的长度：



在每个三个平台上传输控件淡出如果它们不会用到，但可以还原以查看通过点击该视频。

可以阻止自动启动设置视频 `AutoPlay` 属性设置为 `false`：

```
<video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
                   AutoPlay="false" />
```

你将需要按**播放**按钮以开始播放视频。

类似地，可以通过设置隐含的传输控件显示 `AreTransportControlsEnabled` 属性设置为 `false`：

```
<video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
                   AreTransportControlsEnabled="False" />
```

如果这两个属性设置为 `false`，然后视频不会开始播放，并将启动该服务没有办法 ！ 可能需要调用 `Play` 从代码隐藏文件，或创建你自己的传输控件，如本文所述实现自定义视频传输控件。

App.xaml文件包含资源的两个其他视频：

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.App">
    <Application.Resources>
        <ResourceDictionary>

            <video:UriVideoSource x:Key="ElephantsDream"
                                  Uri="https://archive.org/download/ElephantsDream/ed_hd_512kb.mp4" />

            <video:UriVideoSource x:Key="BigBuckBunny"
                                  Uri="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4"
/>

            <video:UriVideoSource x:Key="Sintel"
                                  Uri="https://archive.org/download/Sintel/sintel-2048-stereo_512kb.mp4" />

        </ResourceDictionary>
    </Application.Resources>
</Application>
```

可以为这些其他电影之一的引用，替换中的显式 URL **PlayWebVideo.xaml**文件具有 `StaticResource` 标记扩展，

这种情况下的`VideoSourceConverter`不需要创建`UriVideoSource`对象：

```
<video:VideoPlayer Source="{StaticResource ElephantsDream}" />
```

或者，可以设置`Source`中的视频文件的属性`ListView`，如在下一篇文章中所述绑定到播放器的视频源。

相关链接

- 视频播放机演示（示例）

绑定到播放器的视频源

2018/6/9 • • Edit Online

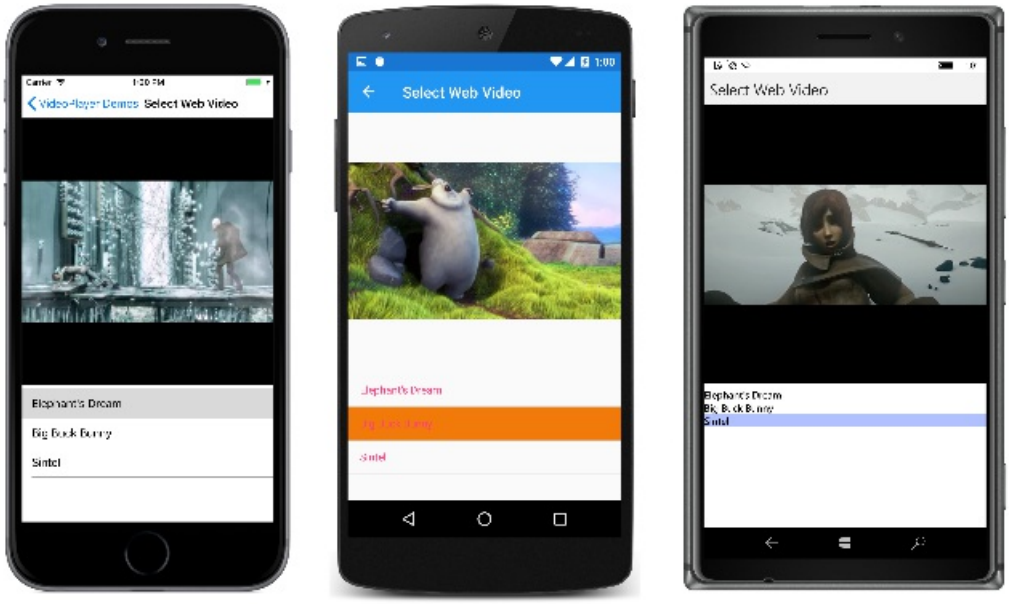
当 `Source` 属性 `VideoPlayer` 视图设为新的视频文件，现有视频停止播放并开始新的播放视频。说明了这一点通过选择 **Web 视频**页 **VideoPlayerDemos** 示例。该页面包括 `ListView` 与从引用的三个视频标题**App.xaml**文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.SelectWebVideoPage"
             Title="Select Web Video">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
                           Grid.Row="0" />

        <ListView Grid.Row="1"
                  ItemSelected="OnListViewItemSelected">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type x:String}">
                    <x:String>Elephant's Dream</x:String>
                    <x:String>Big Buck Bunny</x:String>
                    <x:String>Sintel</x:String>
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </Grid>
</ContentPage>
```

选择视频时，`ItemSelected` 执行的代码隐藏文件中的事件处理程序。该处理程序从标题中删除任何空格和撇号，并使用该值作为键来获取中定义的资源之一**App.xaml**文件。`UriVideoSource` 然后将对象设置为 `Source` 属性 `VideoPlayer` 。

```
namespace VideoPlayerDemos
{
    public partial class SelectWebVideoPage : ContentPage
    {
        public SelectWebVideoPage()
        {
            InitializeComponent();
        }

        void OnListViewItemSelected(object sender, SelectedItemChangedEventArgs args)
        {
            if (args.SelectedItem != null)
            {
                string key = ((string)args.SelectedItem).Replace(" ", "").Replace("'", "");
                videoPlayer.Source = (UriVideoSource)Application.Current.Resources[key];
            }
        }
    }
}
```

在首次加载页时未选定任何项 `ListView` ，因此您必须选择一个用于开始播放视频：



`Source`属性`VideoPlayer`由可绑定的属性，这意味着，它可以是数据绑定的目标。说明了这一点通过将绑定到**VideoPlayer**页。中的标记**BindToVideoPlayer.xaml**由封装的视频和相应的标题的以下类支持文件`VideoSource`对象：

```
namespace VideoPlayerDemos
{
    public class VideoInfo
    {
        public string DisplayName { set; get; }

        public VideoSource VideoSource { set; get; }

        public override string ToString()
        {
            return DisplayName;
        }
    }
}
```

`ListView`中**BindToVideoPlayer.xaml**文件包含的这些数组`VideoInfo`对象，使用视频标题初始化每个与`UriVideoSource`从资源字典中的对象**App.xaml**：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:VideoPlayerDemos"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.BindToVideoPlayerPage"
             Title="Bind to VideoPlayer">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="2*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
                           Grid.Row="0"
                           Source="{Binding Source={x:Reference listView},
                                            Path=SelectedItem.VideoSource}" />

        <ListView x:Name="listView"
                  Grid.Row="1">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:VideoInfo}">
                    <local:VideoInfo DisplayName="Elephant's Dream"
                                     VideoSource="{StaticResource ElephantsDream}" />

                    <local:VideoInfo DisplayName="Big Buck Bunny"
                                     VideoSource="{StaticResource BigBuckBunny}" />

                    <local:VideoInfo DisplayName="Sintel"
                                     VideoSource="{StaticResource Sintel}" />
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </Grid>
</ContentPage>
```

`Source` 属性 `VideoPlayer` 绑定到 `ListView`。 `Path` 绑定的指定为 `SelectedItem.VideoSource`，这是包含两个属性的复合路径：`SelectedItem` 是的一个属性 `ListView`。所选的项的类型是 `VideoInfo`，它具有 `VideoSource` 属性。

第一个与选择 **Web** 视频页上，未选定任何项最初从 `ListView`，因此你需要先选择视频之一，然后它将开始播放。

相关链接

- 视频播放器演示（示例）

加载应用程序资源视频

2018/6/9 • • Edit Online

有关自定义呈现器 `VideoPlayer` 视图是能够播放应用程序资源作为单个平台项目中嵌入的视频文件。但是，当前版本的 `VideoPlayer` 无法访问标准.NET 库中嵌入的资源。

若要加载这些资源，创建的实例 `ResourceVideoSource` 通过设置 `Path` 到文件名（或的文件夹和文件名）的资源的属性。或者，可以调用静态 `VideoSource.FromResource` 方法可以引用该资源。然后，设置 `ResourceVideoSource` 对象传递给 `Source` 属性 `VideoPlayer` 。

存储的视频文件

在平台项目中存储的视频文件时，为将不同用于三个平台。

iOS 视频资源

在 iOS 项目中，你可以存储在视频资源文件夹或子文件夹资源文件夹。视频文件必须具有 `Build Action` 的 `BundleResource` 。设置 `Path` 属性 `ResourceVideoSource` 为文件名，例如， **MyFile.mp4**文件资源文件夹，或**MyFolder/MyFile.mp4**，其中**myfolder 文件**夹是子文件夹的资源。

在**VideoPlayerDemos**解决方案， **VideoPlayerDemos.iOS**项目包含的子文件夹资源名为视频包含一个名为文件**iOSApiVideo.mp4**。这是简短的视频演示如何使用 Xamarin 网站上查找适用于 iOS 的文档 `AVPlayerViewController` 类。

Android 的视频资源

在 Android 项目中，视频必须存储在子文件夹资源名为**原始**。**原始**文件夹不能包含子文件夹。为视频文件提供 `Build Action` 的 `AndroidResource` 。设置 `Path` 属性 `ResourceVideoSource` 为文件名，例如， **MyFile.mp4**。

VideoPlayerDemos.Android项目包含的子文件夹资源名为**原始**，其中包含名为的文件**AndroidApiVideo.mp4**.

UWP 视频资源

在通用 Windows 平台项目中，你可以存储在项目中的任何文件夹中的视频。为此文件提供 `Build Action` 的 `Content` 。设置 `Path` 属性 `ResourceVideoSource` 到的文件夹和文件名，例如， **MyFolder/MyVideo.mp4**。

VideoPlayerDemos.UWP项目包含名为的文件夹视频文件**UWPApiVideo.mp4**。

加载视频文件

每个平台呈现器类包含中的代码其 `SetSource` 加载视频文件作为资源存储方法。

iOS 资源加载

IOS 版本 `VideoPlayerRenderer` 使用 `GetUrlForResource` 方法 `NSBundle` 用于加载资源。完整路径必须划分为文件名、扩展名和目录。该代码使用 `Path` 中.NET 类 `System.IO` 对于划分到这些组件的文件路径的命名空间：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void SetSource()
        {
            AVAsset asset = null;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string path = (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhitespace(path))
                {
                    string directory = Path.GetDirectoryName(path);
                    string filename = Path.GetFileNameWithoutExtension(path);
                    string extension = Path.GetExtension(path).Substring(1);
                    NSUrl url = NSBundle.MainBundle.GetUrlForResource(filename, extension, directory);
                    asset = AVAsset.FromUrl(url);
                }
            }
            ...
        }
        ...
    }
}
```

Android 资源加载

Android `VideoPlayerRenderer` 文件名和包名称用于构造 `Uri` 对象。包名称是应用程序的名称在这种情况下**VideoPlayerDemos.Android**，从中可以获取从静态 `Context.PackageName` 属性。所产生的 `Uri` 对象随后会传递给 `SetVideoURI` 方法 `VideoView`：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string package = Context.PackageName;
                string path = (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhiteSpace(path))
                {
                    string filename = Path.GetFileNameWithoutExtension(path).ToLowerInvariant();
                    string uri = "android.resource://" + package + "/raw/" + filename;
                    videoView.SetVideoURI(Android.Net.Uri.Parse(uri));
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}
```

UWP 资源加载

UWP `VideoPlayerRenderer` 构造 `Uri` 路径的对象并将它设置为 `Source` 属性 `MediaElement`：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        async void SetSource()
        {
            bool hasSetSource = false;
            ...
            else if (Element.Source is ResourceVideoSource)
            {
                string path = "ms-appx:///" + (Element.Source as ResourceVideoSource).Path;

                if (!String.IsNullOrWhiteSpace(path))
                {
                    Control.Source = new Uri(path);
                    hasSetSource = true;
                }
            }
        }
        ...
    }
}
```

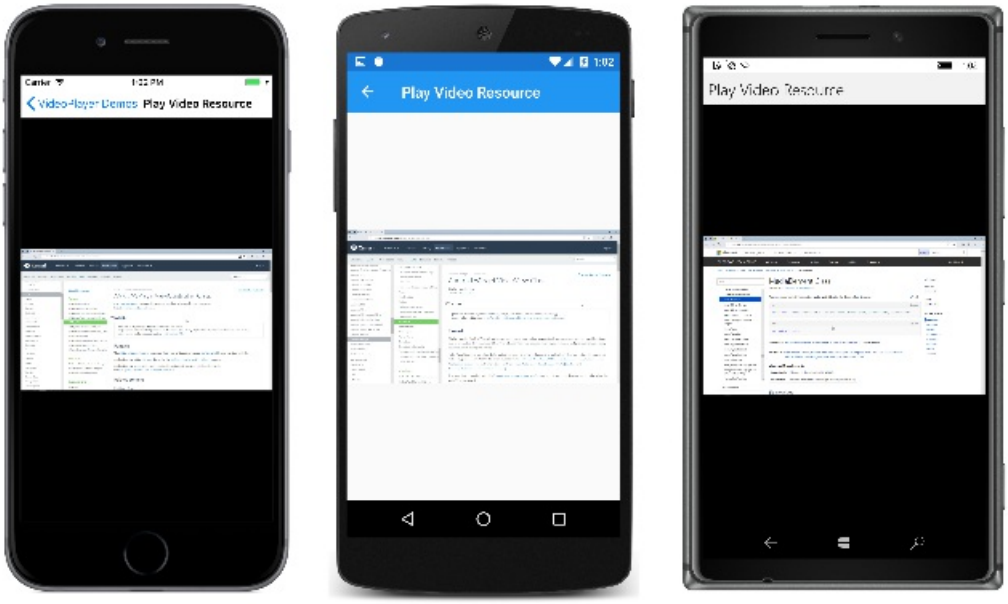
播放的资源文件

播放视频资源页面VideoPlayerDemos解决方案使用 `OnPlatform` 类可指定用于每个平台的视频文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.PlayVideoResourcePage"
             Title="Play Video Resource">
    <video:VideoPlayer>
        <video:VideoPlayer.Source>
            <video:ResourceVideoSource>
                <video:ResourceVideoSource.Path>
                    <OnPlatform x:TypeArguments="x:String">
                        <On Platform="iOS" Value="Videos/iOSApiVideo.mp4" />
                        <On Platform="Android" Value="AndroidApiVideo.mp4" />
                        <On Platform="UWP" Value="Videos/UWPApiVideo.mp4" />
                    </OnPlatform>
                </video:ResourceVideoSource.Path>
            </video:ResourceVideoSource>
        </video:VideoPlayer.Source>
    </video:VideoPlayer>
</ContentPage>
```

如果 iOS 资源存储在资源文件夹，并且如果 UWP 资源存储在项目的根文件夹中，可以用于三个平台使用相同的文件名。如果是这样，则您可以将该名称直接为 `Source` 属性 `VideoPlayer`。

下面是三个平台上运行该页面：



现在，你已了解如何从 Web URI 加载视频以及如何播放嵌入的资源。此外，你可以从设备的视频库加载视频。

相关链接

- 视频播放器演示（示例）

访问设备的视频库

2018/6/9 • • Edit Online

大多数现代移动设备和台式计算机能够使用设备的摄像头的视频记录。然后，用户创建的视频将存储为设备上的文件。可以从映像库中检索这些文件，通过播放 `VideoPlayer` 就像任何其他视频的类。

照片选取器依赖关系服务

其中每三个平台都包含一种功能，允许用户从设备的映像库中选择的照片或视频。播放视频设备的映像库中的第一步生成时，将调用每个平台上的映像选取器的依赖项服务。如下所述的依赖项服务是非常类似于中定义一个**从图片库中选取照片**文章，只不过视频选取器返回文件名而不是 `Stream` 对象。

标准.NET 类库项目定义一个名为的接口 `IVideoPicker` 依赖项服务：

```
namespace FormsVideoLibrary
{
    public interface IVideoPicker
    {
        Task<string> GetVideoFileAsync();
    }
}
```

每三个平台都包含一个名为类 `VideoPicker` 实现此接口。

IOS 视频选取器

IOS `VideoPicker` 使用 iOS `UIImagePickerController` 访问映像库，指定它应被限制为视频（称为"电影"）iOS 中 `MediaType` 属性。请注意，`VideoPicker` 显式实现 `IVideoPicker` 接口。另请注意 `Dependency` 标识作为一种依赖服务的此类的属性。这些是允许 Xamarin.Forms 平台项目中查找依赖项服务的两个要求：

```
using System;
using System.Threading.Tasks;
using UIKit;
using Xamarin.Forms;

[assembly: Dependency(typeof(FormsVideoLibrary.iOS.VideoPicker))]

namespace FormsVideoLibrary.iOS
{
    public class VideoPicker : IVideoPicker
    {
        TaskCompletionSource<string> taskCompletionSource;
        UIImagePickerController imagePicker;

        public Task<string> GetVideoFileAsync()
        {
            // Create and define UIImagePickerController
            imagePicker = new UIImagePickerController
            {
                SourceType = UIImagePickerControllerSourceType.SavedPhotosAlbum,
                MediaTypes = new string[] { "public.movie" }
            };

            // Set event handlers
            imagePicker.FinishedPickingMedia += OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled += OnImagePickerCancelled;

            // Present UIImagePickerController;
            UIWindow window = UIApplication.SharedApplication.KeyWindow;
            var viewController = window.RootViewController;
            viewController.PresentModalViewController(imagePicker, true);

            // Return Task object
            taskCompletionSource = new TaskCompletionSource<string>();
            return taskCompletionSource.Task;
        }

        void OnImagePickerFinishedPickingMedia(object sender, UIImagePickerMediaPickedEventArgs args)
        {
            if (args.MediaType == "public.movie")
            {
                taskCompletionSource.SetResult(args.MediaUrl.AbsoluteString);
            }
            else
            {
                taskCompletionSource.SetResult(null);
            }
            imagePicker.DismissModalViewController(true);
        }

        void OnImagePickerCancelled(object sender, EventArgs args)
        {
            taskCompletionSource.SetResult(null);
            imagePicker.DismissModalViewController(true);
        }
    }
}
```

Android 视频选取器

Android 实现 `IVideoPicker` 需要属于应用程序的活动的回调方法。为此，`MainActivity` 类定义两个属性、字段和回调方法：

```
namespace VideoPlayerDemos.Droid
{
    ...
    public class MainActivity : global::Xamarin.Forms.Platform.Android.FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            Current = this;
            ...
        }

        // Field, properties, and method for Video Picker
        public static MainActivity Current { private set; get; }

        public static readonly int PickImageId = 1000;

        public TaskCompletionSource<string> PickImageTaskCompletionSource { set; get; }

        protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
        {
            base.OnActivityResult(requestCode, resultCode, data);

            if (requestCode == PickImageId)
            {
                if ((resultCode == Result.Ok) && (data != null))
                {
                    // Set the filename as the completion of the Task
                    PickImageTaskCompletionSource.SetResult(data.DataString);
                }
                else
                {
                    PickImageTaskCompletionSource.SetResult(null);
                }
            }
        }
    }
}
```

`OnCreate`中的方法`MainActivity`将自己的实例存储在静态`Current`属性。这样的实现`IVideoPicker`获取`MainActivity`用于启动实例选择视频选择器：

```
using System;
using System.Threading.Tasks;
using Android.Content;
using Xamarin.Forms;

// Need application's MainActivity
using VideoPlayerDemos.Droid;

[assembly: Dependency(typeof(FormsVideoLibrary.Droid.VideoPicker))]

namespace FormsVideoLibrary.Droid
{
    public class VideoPicker : IVideoPicker
    {
        public Task<string> GetVideoFileAsync()
        {
            // Define the Intent for getting images
            Intent intent = new Intent();
            intent.SetType("video/*");
            intent.SetAction(Intent.ActionGetContent);

            // Get the MainActivity instance
            MainActivity activity = MainActivity.Current;

            // Start the picture-picker activity (resumes in MainActivity.cs)
            activity.StartActivityForResult(
                Intent.CreateChooser(intent, "Select Video"),
                MainActivity.PickImageId);

            // Save the TaskCompletionSource object as a MainActivity property
            activity.PickImageTaskCompletionSource = new TaskCompletionSource<string>();

            // Return Task object
            return activity.PickImageTaskCompletionSource.Task;
        }
    }
}
```

上述元素添加到`MainActivity`对象是中的唯一代码 **VideoPlayerDemos** 解决方案普通的应用程序代码需要更改以支持`FormsVideoLibrary`类。

UWP 视频选取器

UWP 实现`IVideoPicker`界面使用 UWP `FileOpenPicker` 。它开始图片库中，使用文件搜索，并将文件类型限制为 MP4 和 WMV（Windows Media 视频）：

```
using System;
using System.Threading.Tasks;
using Windows.Storage;
using Windows.Storage.Pickers;
using Xamarin.Forms;

[assembly: Dependency(typeof(FormsVideoLibrary.UWP.VideoPicker))]

namespace FormsVideoLibrary.UWP
{
    public class VideoPicker : IVideoPicker
    {
        public async Task<string> GetVideoFileAsync()
        {
            // Create and initialize the FileOpenPicker
            FileOpenPicker openPicker = new FileOpenPicker
            {
                ViewMode = PickerViewMode.Thumbnail,
                SuggestedStartLocation = PickerLocationId.PicturesLibrary
            };

            openPicker.FileTypeFilter.Add(".wmv");
            openPicker.FileTypeFilter.Add(".mp4");

            // Get a file and return the path
            StorageFile storageFile = await openPicker.PickSingleFileAsync();
            return storageFile?.Path;
        }
    }
}
```

调用依赖服务

播放视频库页 **VideoPlayerDemos** 程序演示如何使用视频选取器依赖关系服务。XAML 文件中包含 `VideoPlayer` 实例和一个 `Button` 标记为显示视频库：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.PlayLibraryVideoPage"
             Title="Play Library Video">
    <StackLayout>
        <video:VideoPlayer x:Name="videoPlayer"
                           VerticalOptions="FillAndExpand" />

        <Button Text="Show Video Library"
                Margin="10"
                HorizontalOptions="Center"
                Clicked="OnShowVideoLibraryClicked" />
    </StackLayout>
</ContentPage>
```

代码隐藏文件包含 `Clicked` 处理程序 `Button` 。调用依赖服务需要调用 `DependencyService.Get` 若要获取的实现 `IVideoPicker` 平台项目中的接口。 `GetVideoFileAsync` 然后在该实例上调用方法：

```
namespace VideoPlayerDemos
{
    public partial class PlayLibraryVideoPage : ContentPage
    {
        public PlayLibraryVideoPage()
        {
            InitializeComponent();
        }

       async void OnShowVideoLibraryClicked(object sender, EventArgs args)
        {
            Button btn = (Button)sender;
            btn.IsEnabled = false;

            string filename = await DependencyService.Get<IVideoPicker>().GetVideoFileAsync();

            if (!String.IsNullOrWhiteSpace(filename))
            {
                videoPlayer.Source = new FileVideoSource
                {
                    File = filename
                };
            }

            btn.IsEnabled = true;
        }
    }
}
```

`Clicked` 处理然后使用该文件名来创建 `FileVideoSource` 对象并将它设置为 `Source` 属性 `VideoPlayer` 。

每个 `VideoPlayerRenderer` 类包含中的代码其 `SetSource` 类型的对象的方法 `FileVideoSource` 。这些如下所示：

处理 iOS 文件

IOS 版本 `VideoPlayerRenderer` 进程 `FileVideoSource` 使用静态对象 `Asset.FromUrl` 文件名的方法。这创建 `AVAsset` 表示设备的映像库中文件的对象：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void SetSource()
        {
            AVAsset asset = null;
            ...
            else if (Element.Source is FileVideoSource)
            {
                string uri = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(uri))
                {
                    asset = AVAsset.FromUrl(new NSUrl(uri));
                }
            }
            ...
        }
        ...
    }
}
```

处理 Android 文件

在处理类型的对象时 `FileVideoSource` 的 Android 实现 `VideoPlayerRenderer` 使用 `SetVideoPath` 方法 `VideoView` 设备

的映像库中指定的文件：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void SetSource()
        {
            isPrepared = false;
            bool hasSetSource = false;
            ...
            else if (Element.Source is FileVideoSource)
            {
                string filename = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(filename))
                {
                    videoView.SetVideoPath(filename);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}
```

处理 UWP 文件

在处理类型的对象时 `FileVideoSource` 的 UWP 实现 `SetSource` 方法需要创建 `StorageFile` 对象、打开该文件进行读取，并将传递到的流对象 `SetSource` 方法 `MediaElement`：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
        async void SetSource()
        {
            bool hasSetSource = false;
            ...
            else if (Element.Source is FileVideoSource)
            {
                // Code requires Pictures Library in Package.appxmanifest Capabilities to be enabled
                string filename = (Element.Source as FileVideoSource).File;

                if (!String.IsNullOrWhiteSpace(filename))
                {
                    StorageFile storageFile = await StorageFile.GetFileFromPathAsync(filename);
                    IRandomAccessStreamWithContentType stream = await storageFile.OpenReadAsync();
                    Control.SetSource(stream, storageFile.ContentType);
                    hasSetSource = true;
                }
            }
            ...
        }
        ...
    }
}
```

对于所有三个平台，视频开始播放视频之后，几乎可以立即设置源，因为该文件在设备上，不需要下载。

相关链接

- 视频播放器演示（示例）
- 从图片库中选取照片

自定义视频传输控件

2018/7/13 • • Edit Online

视频播放器的传输控件包括执行功能的按钮**播放**，**暂停**，**并停止**。这些按钮通常带有熟悉图标而不是文本，**并播放并暂停**函数通常组合成一个按钮。

默认情况下，`VideoPlayer` 显示传输控件支持的每个平台。当您将设置 `AreTransportControlsEnabled` 属性设置为 `false`，这些控件被抑制。然后，你可以控制 `VideoPlayer` 以编程方式或提供你自己的传输控件。

播放、暂停和停止方法

`VideoPlayer` 类定义了三个方法名为 `Play`， `Pause`，和 `Stop` 实现的触发事件：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        public event EventHandler PlayRequested;

        public void Play()
        {
            PlayRequested?.Invoke(this, EventArgs.Empty);
        }

        public event EventHandler PauseRequested;

        public void Pause()
        {
            PauseRequested?.Invoke(this, EventArgs.Empty);
        }

        public event EventHandler StopRequested;

        public void Stop()
        {
            StopRequested?.Invoke(this, EventArgs.Empty);
        }
    }
}
```

这些事件的事件处理程序设置的 `VideoPlayerRenderer` 类中每个平台，如下所示：

iOS 传输实现

IOS 版本的 `VideoPlayerRenderer` 使用 `OnElementChanged` 方法以设置的这三个事件处理程序时 `NewElement` 属性不是 `null` 并分离事件处理程序时 `OldElement` 不是 `null`：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        AVPlayer player;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }
        ...
        // Event handlers to implement methods
        void OnPlayRequested(object sender, EventArgs args)
        {
            player.Play();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            player.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            player.Pause();
            player.Seek(new CMTime(0, 1));
        }
    }
}
```

事件处理程序通过调用的方法实现 `AVPlayer` 对象。没有任何 `Stop` 方法 `AVPlayer`，因此它模拟通过暂停视频并将位置移到开头。

Android 传输实现

Android 实现是类似于 iOS 实现。三个函数的处理程序时设置 `NewElement` 不是 `null` 并且分离出何时 `OldElement` 不是 `null`：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }
        ...
        void OnPlayRequested(object sender, EventArgs args)
        {
            videoView.Start();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            videoView.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            videoView.StopPlayback();
        }
    }
}
```

三个函数调用方法定义的 `VideoView`。

UWP 传输实现

三个传输函数的 UWP 实现是非常类似于 iOS 和 Android 的实现：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.PlayRequested += OnPlayRequested;
                args.NewElement.PauseRequested += OnPauseRequested;
                args.NewElement.StopRequested += OnStopRequested;
            }

            if (args.OldElement != null)
            {
                ...
                args.OldElement.PlayRequested -= OnPlayRequested;
                args.OldElement.PauseRequested -= OnPauseRequested;
                args.OldElement.StopRequested -= OnStopRequested;
            }
        }
        ...
        // Event handlers to implement methods
        void OnPlayRequested(object sender, EventArgs args)
        {
            Control.Play();
        }

        void OnPauseRequested(object sender, EventArgs args)
        {
            Control.Pause();
        }

        void OnStopRequested(object sender, EventArgs args)
        {
            Control.Stop();
        }
    }
}
```

视频播放器状态

实现播放，暂停，并停止函数还不足以支持传输控件。通常播放并暂停使用同一按钮来更改其外观来指示视频播放或暂停当前是实现命令。此外，仅当尚未加载视频时，甚至不应启用按钮。

这些要求意味着，视频播放器需要提供当前状态，指示如果它是播放或暂停，或者如果尚未准备好播放视频。（三个平台还支持属性，指示是否视频可以暂停，也可以移动到新位置，但这些属性是适用于流式处理视频而不是视频文件，因此它们不支持在 `VideoPlayer` 此处所述。）

VideoPlayerDemos项目包括 `VideoStatus` 包含三个成员的枚举：

```
namespace FormsVideoLibrary
{
    public enum VideoStatus
    {
        NotReady,
        Playing,
        Paused
    }
}
```

`VideoPlayer` 类定义了名为的仅限实际的可绑定属性 `Status` 类型的 `VideoStatus` 。此属性被定义为只读的因为它仅应设置为从平台呈现器：

```
using System;
using Xamarin.Forms;

namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Status read-only property
        private static readonly BindablePropertyKey StatusPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Status), typeof(VideoStatus), typeof(VideoPlayer),
VideoStatus.NotReady);

        public static readonly BindableProperty StatusProperty = StatusPropertyKey.BindableProperty;

        public VideoStatus Status
        {
            get { return (VideoStatus)GetValue(StatusProperty); }
        }

        VideoStatus IVideoPlayerController.Status
        {
            set { SetValue(StatusPropertyKey, value); }
            get { return Status; }
        }
        ...
    }
}
```

通常情况下，只读的可绑定的属性将具有私有 `set` 访问器上的 `Status` 属性以使其能够从类中设置。有关 `View` 支持的呈现器，但是，派生类的属性必须设置从类外部的但只能由平台呈现器。

出于此原因，另一个属性定义的名称 `IVideoPlayerController.Status` 。这是一个显式接口实现，并可生成 `IVideoPlayerController` 接口的 `VideoPlayer` 类实现：

```
namespace FormsVideoLibrary
{
    public interface IVideoPlayerController
    {
        VideoStatus Status { set; get; }

        TimeSpan Duration { set; get; }
    }
}
```

它类似于如何 `WebView` 控件使用 `IWebViewController` 要实现的接口 `CanGoBack` 和 `CanGoForward` 属性。（请参阅的源代码 `WebView` 和其呈现器的详细信息。）

这样就可以为类的外部 `VideoPlayer` 若要设置 `Status` 通过引用属性 `IVideoPlayerController` 接口。（稍后您将看到代码。）该属性可以设置从其他类，但不太可能会无意中设置。最重要的是， `Status` 属性不能通过数据绑定设置。

若要帮助在此数字保持呈现器 `Status` 属性已更新， `VideoPlayer` 类定义 `UpdateStatus` 事件时触发每隔 10 秒：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        public event EventHandler UpdateStatus;

        public VideoPlayer()
        {
            Device.StartTimer(TimeSpan.FromMilliseconds(100), () =>
            {
                UpdateStatus?.Invoke(this, EventArgs.Empty);
                return true;
            });
        }
        ...
    }
}
```

IOS 状态设置

IOS `VideoPlayerRenderer` 设置的处理程序 `UpdateStatus` 事件（并分离该处理程序时的基础 `VideoPlayer` 元素不存在），并使用该处理程序来设置 `Status` 属性：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                ...
                args.NewElement.UpdateStatus += OnUpdateStatus;
                ...
            }

            if (args.OldElement != null)
            {
                args.OldElement.UpdateStatus -= OnUpdateStatus;
                ...
            }
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            VideoStatus videoStatus = VideoStatus.NotReady;

            switch (player.Status)
            {
                case AVPlayerStatus.ReadyToPlay:
                    switch (player.TimeControlStatus)
                    {
                        case AVPlayerTimeControlStatus.Playing:
                            videoStatus = VideoStatus.Playing;
                            break;

                        case AVPlayerTimeControlStatus.Paused:
                            videoStatus = VideoStatus.Paused;
                            break;
                    }
                    break;
                }
            }

            ((IVideoPlayerController)Element).Status = videoStatus;
            ...
        }
        ...
    }
}
```

两个属性`AVPlayer`必须访问：`Status`类型的属性`AVPlayerStatus`并`TimeControlStatus`类型的属性`AVPlayerTimeControlStatus`。请注意，`Element`属性（即`VideoPlayer`）必须强制转换为`IVideoPlayerController`若要设置`Status`属性。

Android 状态设置

`IsPlaying` Android 属性`VideoView`是一个布尔值，仅指示视频是否播放或暂停。若要确定是否`VideoView`都不 play 也不能暂停视频还，`Prepared`事件的`VideoView`必须处理。这些两个处理程序中设置`OnElementChanged`方法，并分离期间`Dispose`重写：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        bool isPrepared;

        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            ...
            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    ...
                    videoView.Prepared += OnVideoViewPrepared;
                    ...
                }
                ...
                args.NewElement.UpdateStatus += OnUpdateStatus;
                ...
            }

            if (args.OldElement != null)
            {
                args.OldElement.UpdateStatus -= OnUpdateStatus;
                ...
            }

        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null && videoView != null)
            {
                videoView.Prepared -= OnVideoViewPrepared;
            }
            if (Element != null)
            {
                Element.UpdateStatus -= OnUpdateStatus;
            }

            base.Dispose(disposing);
        }
        ...
    }
}
```

`UpdateStatus` 处理程序使用 `isPrepared` 字段（在中设置 `Prepared` 处理程序）和 `IsPlaying` 属性来设置 `Status` 属性：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        VideoView videoView;
        ...
        bool isPrepared;
        ...
        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            isPrepared = true;
            ...
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            VideoStatus status = VideoStatus.NotReady;

            if (isPrepared)
            {
                status = videoView.IsPlaying ? VideoStatus.Playing : VideoStatus.Paused;
            }
            ...
        }
        ...
    }
}
```

UWP 状态设置

UWP `VideoPlayerRenderer` 利用 `UpdateStatus` 事件，但它不需要它用于设置 `Status` 属性。 `MediaElement` 定义 `CurrentStateChanged` 允许呈现器的事件时通知 `CurrentState` 属性已更改。在分离属性 `Dispose` 重写：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        protected override void OnElementChanged(ElementChangedEventArgs<VideoPlayer> args)
        {
            base.OnElementChanged(args);

            if (args.NewElement != null)
            {
                if (Control == null)
                {
                    ...
                    mediaElement.CurrentStateChanged += OnMediaElementCurrentStateChanged;
                };
                ...
            }
            ...
        }

        protected override void Dispose(bool disposing)
        {
            if (Control != null)
            {
                ...
                Control.CurrentStateChanged -= OnMediaElementCurrentStateChanged;
            }

            base.Dispose(disposing);
        }
        ...
    }
}
```

`CurrentState`属性属于类型`MediaElementState`，并轻松地将映射到`VideoStatus`：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        void OnMediaElementCurrentStateChanged(object sender, RoutedEventArgs args)
        {
            VideoStatus videoStatus = VideoStatus.NotReady;

            switch (Control.CurrentState)
            {
                case MediaElementState.Playing:
                    videoStatus = VideoStatus.Playing;
                    break;

                case MediaElementState.Paused:
                case MediaElementState.Stopped:
                    videoStatus = VideoStatus.Paused;
                    break;
            }

            ((IVideoPlayerController)Element).Status = videoStatus;
        }
        ...
    }
}
```

播放、暂停和停止按钮

使用 Unicode 字符的符号**播放**，暂**停**，**并停止**图像会出现问题。Miscellaneous 技术部分 Unicode 标准定义了三个看似适合于此用途的符号字符。这些是：

- 0x23F5（黑色中等右指三角形）或□有关**播放**
- 0x23F8（双竖线）或□有关暂**停**
- 0x23F9（黑色方框）或□有关**停止**

而不考虑如何在浏览器中显示这些符号（以及不同的浏览器处理这些不同的方式），它们不一致地显示通过 Xamarin.Forms 支持的平台上。IOS 和 UWP 设备上暂**停并停止**字符有图形的外观，与蓝色 3D 背景和白色的前景色。这不是在 Android 上，符号是只是蓝色的情况。但是，对于 0x23F5 codepoint**播放**不具有相同的外观，UWP，和它甚至不支持在 iOS 和 Android。

为此，0x23F5 码位不能用于**播放**。是一个不错的替代品：

- 0x25B6（黑色右指三角形）或▶有关**播放**

这支持的所有三个平台，只不过它是普通的黑色三角形不像的三维效果暂**停并停止**。一种可能性是遵循 0x25B6 码位，变体的代码：

- 0x25B6 跟 0xFE0F（变量 16）或▶□有关**播放**

这是如下所示的标记中的用途。在 iOS 上，这样**播放**符号作为相同的三维外观暂**停并停止**按钮，但将该变量不在 Android 和 UWP 上起作用。

自定义传输页上设置**AreTransportControlsEnabled**属性设置为**false**并包括 `ActivityIndicator` 加载视频时，显示和两个按钮。 `DataTrigger` 对象用于启用和禁用 `ActivityIndicator` 和按钮，并将第一个按钮之间切换**播放**并暂**停**：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.CustomTransportPage"
             Title="Custom Transport">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
                           Grid.Row="0"
                           AutoPlay="False"
                           AreTransportControlsEnabled="False"
                           Source="{StaticResource BigBuckBunny}" />

        <ActivityIndicator Grid.Row="0"
                           Color="Gray"
                           IsVisible="False">
            <ActivityIndicator.Triggers>
                <DataTrigger TargetType="ActivityIndicator"
                             Binding="{Binding Source={x:Reference videoPlayer},
                                               Path=Status}"
                             Value="{x:Static video:VideoStatus.NotReady}">
                    <Setter Property="IsVisible" Value="True" />
                    <Setter Property="IsRunning" Value="True" />
                </DataTrigger>
            </ActivityIndicator.Triggers>
        </ActivityIndicator>

        <StackLayout Grid.Row="1"
                     Orientation="Horizontal"
                     Margin="0, 10"
```

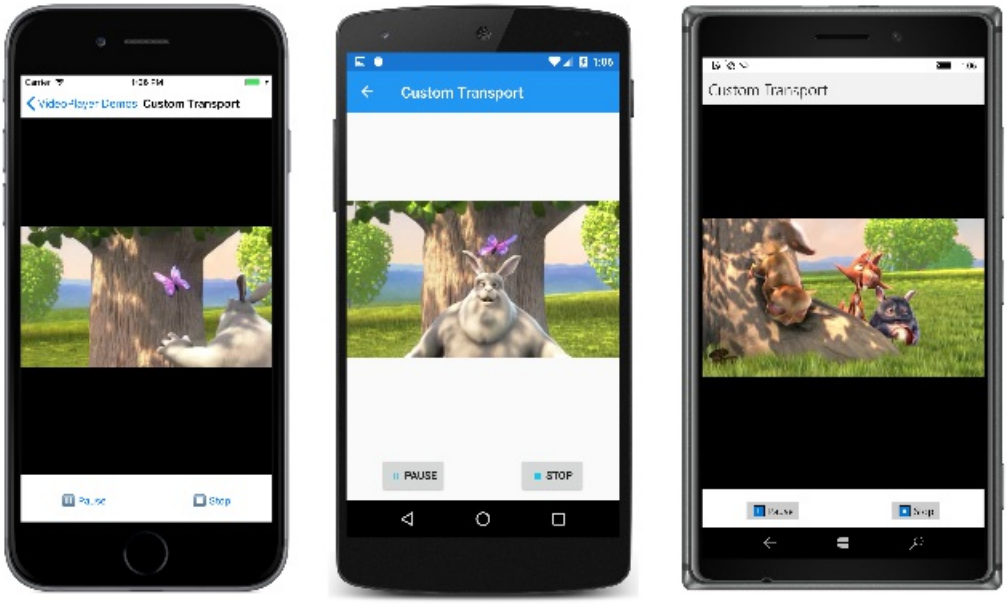
```
                    BindingContext="{x:Reference videoPlayer}">

            <Button Text="&#x25B6;&#xFE0F; Play"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnPlayPauseButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                                 Binding="{Binding Status}"
                                 Value="{x:Static video:VideoStatus.Playing}">
                        <Setter Property="Text" Value="&#x23F8; Pause" />
                    </DataTrigger>

                    <DataTrigger TargetType="Button"
                                 Binding="{Binding Status}"
                                 Value="{x:Static video:VideoStatus.NotReady}">
                        <Setter Property="IsEnabled" Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>

            <Button Text="&#x23F9; Stop"
                    HorizontalOptions="CenterAndExpand"
                    Clicked="OnStopButtonClicked">
                <Button.Triggers>
                    <DataTrigger TargetType="Button"
                                 Binding="{Binding Status}"
                                 Value="{x:Static video:VideoStatus.NotReady}">
                        <Setter Property="IsEnabled" Value="False" />
                    </DataTrigger>
                </Button.Triggers>
            </Button>
        </StackLayout>
    </Grid>
</ContentPage>
```

一文中详细描述了数据触发器数据触发器。

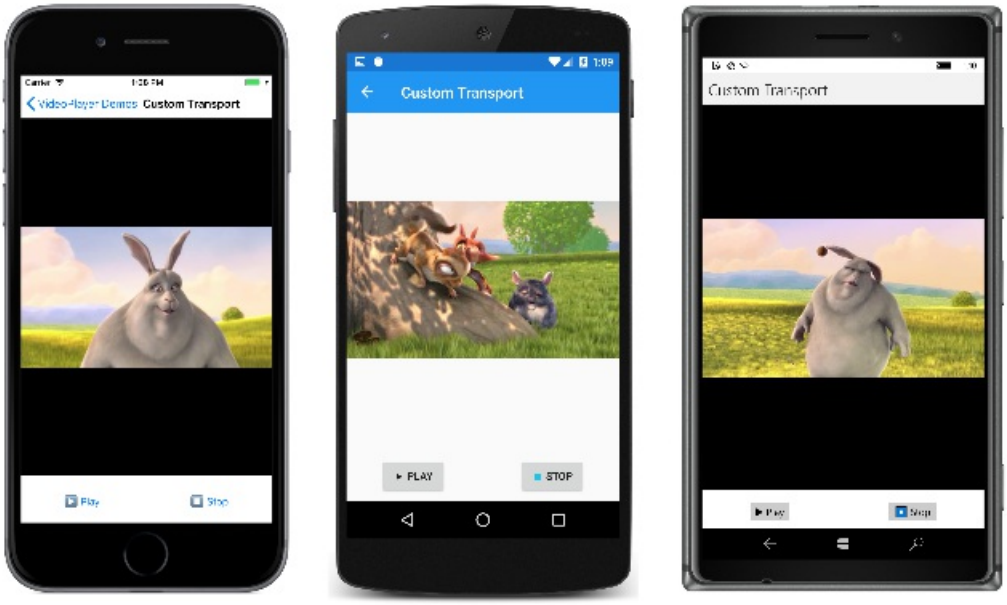
代码隐藏文件具有按钮处理程序 `Clicked` 事件：

```
namespace VideoPlayerDemos
{
    public partial class CustomTransportPage : ContentPage
    {
        public CustomTransportPage()
        {
            InitializeComponent();
        }

        void OnPlayPauseButtonClicked(object sender, EventArgs args)
        {
            if (videoPlayer.Status == VideoStatus.Playing)
            {
                videoPlayer.Pause();
            }
            else if (videoPlayer.Status == VideoStatus.Paused)
            {
                videoPlayer.Play();
            }
        }

        void OnStopButtonClicked(object sender, EventArgs args)
        {
            videoPlayer.Stop();
        }
    }
}
```

因为 `AutoPlay` 设置为 `false` 中**CustomTransport.xaml**文件中，你将需要按**播放**按钮时它将成为能够开始视频。按钮定义，因此上面讨论的 Unicode 字符都伴随其文本等效项。播放视频时，按钮在每个平台上具有一致的外观：



但在 Android 和 UWP，**播放**暂停视频时，按钮看起来很不同：



在生产应用程序中，您可能需要使用您自己的位图图像的按钮来实现 visual 一致性。

相关链接

- 视频播放机演示（示例）

自定义视频定位

2018/6/9 • • Edit Online

由每个平台实现的传输控制包括位置栏。此栏类似于滑块或滚动条，并在其总持续时间内显示视频的当前位置。此外，用户可以操作的位置栏将向前或向后移动到视频中的新位置。

这篇文章演示如何实现你自己的自定义位置栏。

持续时间属性

信息的一个项，`VideoPlayer` 需要支持自定义位置栏是视频的持续时间。`VideoPlayer` 定义只读 `Duration` 类型的属性 `TimeSpan`：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Duration read-only property
        private static readonly BindablePropertyKey DurationPropertyKey =
            BindableProperty.CreateReadOnly(nameof(Duration), typeof(TimeSpan), typeof(VideoPlayer), new
TimeSpan(),
                propertyChanged: (bindable, oldValue, newValue) => ((VideoPlayer)bindable).SetTimeToEnd());

        public static readonly BindableProperty DurationProperty = DurationPropertyKey.BindableProperty;

        public TimeSpan Duration
        {
            get { return (TimeSpan)GetValue(DurationProperty); }
        }

        TimeSpan IVideoPlayerController.Duration
        {
            set { SetValue(DurationPropertyKey, value); }
            get { return Duration; }
        }
        ...
    }
}
```

如 `Status` 属性中所述上一篇文章，则此 `Duration` 属性是只读的。定义专用 `BindablePropertyKey` 并且仅可通过引用设置 `IVideoPlayerController` 接口，其中包括这 `Duration` 属性：

```
namespace FormsVideoLibrary
{
    public interface IVideoPlayerController
    {
        VideoStatus Status { set; get; }

        TimeSpan Duration { set; get; }
    }
}
```

此外会注意到的属性更改的处理程序调用一个名为方法 `SetTimeToEnd` 描述此文章中更高版本。

视频的持续时间是*不*可用后立即 `Source` 属性 `VideoPlayer` 设置。部分必须下载视频文件，才能基础视频播放器可以

确定其持续时间。

下面是每个平台呈现器如何获取视频的持续时间：

在 iOS 中的视频持续时间

在 iOS 视频的持续时间获取从 `Duration` 属性 `AVPlayerItem`，但不是立即晚于 `AVPlayerItem` 创建。可以设置的 iOS 观察者 `Duration` 属性，但 `VideoPlayerRenderer` 获取的持续时间以 `UpdateStatus` 方法，即每秒 10 次：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            if (playerItem != null)
            {
                ((IVideoPlayerController)Element).Duration = ConvertTime(playerItem.Duration);
                ...
            }
        }

        TimeSpan ConvertTime(CMTime cmTime)
        {
            return TimeSpan.FromSeconds(Double.IsNaN(cmTime.Seconds) ? 0 : cmTime.Seconds);

        }
        ...
    }
}
```

`ConvertTime` 方法将 `CMTime` 对象传递给 `TimeSpan` 值。

在 Android 中的视频持续时间

`Duration` Android 属性 `VideoView` 报告以毫秒为单位的有效持续时间时 `Prepared` 事件 `VideoView` 激发。Android `VideoPlayerRenderer` 类使用该处理程序来获取 `Duration` 属性：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void OnVideoViewPrepared(object sender, EventArgs args)
        {
            ...
            ((IVideoPlayerController)Element).Duration = TimeSpan.FromMilliseconds(videoView.Duration);
        }
        ...
    }
}
```

在 UWP 视频持续时间

`NaturalDuration` 属性 `MediaElement` 是 `TimeSpan` 值并成为有效 `MediaElement` 激发 `MediaOpened` 事件：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        void OnMediaElementMediaOpened(object sender, RoutedEventArgs args)
        {
            ((IVideoPlayerController)Element).Duration = Control.NaturalDuration.TimeSpan;
        }
        ...
    }
}
```

位置属性

`VideoPlayer` 此外需要 `Position` 属性可提高从零到 `Duration` 播放视频。`VideoPlayer` 实现此属性，如 `Position` UWP 中的属性 `MediaElement`，这是正常的可绑定属性的公共 `set` 和 `get` 访问器：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        // Position property
        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(VideoPlayer), new TimeSpan(),
                propertyChanged: (bindable, oldValue, newValue) => ((VideoPlayer)bindable).SetTimeToEnd());

        public TimeSpan Position
        {
            set { SetValue(PositionProperty, value); }
            get { return (TimeSpan)GetValue(PositionProperty); }
        }
        ...
    }
}
```

`get` 访问器返回的视频当前位置播放它时，但 `set` 访问器旨在通过向前或向后移动的视频的位置的位置栏的用户的操作做出响应。

在 iOS 和 Android 中，获取当前的位置的属性仅具有 `get` 访问器和 `Seek` 方法是可用于执行此第二个任务。如果你认为一下，单独 `Seek` 方法似乎是一个更明智的做法比单个 `Position` 属性。单个 `Position` 属性具有固有的问题：在视频播放时 `Position` 属性必须持续更新，以反映新的位置。但你不希望对大多数更改 `Position` 属性以使视频播放器将移到视频中的新位置。如果发生这种情况，通过查找到最后一个值以响应视频播放器会 `Position` 不向前移动属性和视频。

尽管的实现难度 `Position` 具有属性 `set` 和 `get` 访问器中，因为它是与 UWP 一致这种方法选择 `MediaElement`，并且具有与数据绑定一起很大的优势：`Position` 属性 `VideoPlayer` 可以绑定到用于同时显示的位置并要查找到新位置的滑块。但是，一些预防措施是必需的实现这时 `Position` 属性以避免反馈循环。

设置和获取 iOS 位置

在 iOS 中，`CurrentTime` 属性 `AVPlayerItem` 对象指示播放视频的当前位置。IOS `VideoPlayerRenderer` 设置 `Position` 中的属性 `UpdateStatus` 同时它所设置的处理程序 `Duration` 属性：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            if (playerItem != null)
            {
                ...
                ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty,
ConvertTime(playerItem.CurrentTime));
            }
        }
        ...
    }
}
```

呈现器检测到时 `Position` 属性设置从 `VideoPlayer` 中已更改 `OnElementPropertyChanged` 重写，并使用该新值来调用 `Seek` 方法 `AVPlayer` 对象：

```
namespace FormsVideoLibrary.iOS
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, UIView>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                TimeSpan controlPosition = ConvertTime(player.CurrentTime);

                if (Math.Abs((controlPosition - Element.Position).TotalSeconds) > 1)
                {
                    player.Seek(CMTime.FromSeconds(Element.Position.TotalSeconds, 1));
                }
            }
        }
        ...
    }
}
```

请记住，每次 `Position` 中的属性 `VideoPlayer` 设置从 `OnUpdateStatus` 处理程序， `Position` 属性激发 `PropertyChanged` 事件，在中检测到 `OnElementPropertyChanged` 重写。有关这些更改中的大多数 `OnElementPropertyChanged` 方法不执行任何操作。否则，视频的位置中的每个更改，它将移到刚达到的相同位置 ！

若要避免这种反馈循环， `OnElementPropertyChanged` 方法仅调用 `Seek` 时之间的差异 `Position` 属性和的当前位置 `AVPlayer` 大于 1 秒。

设置和获取 Android 位置

IOS 呈现器，Android 中一样 `VideoPlayerRenderer` 设置的新值 `Position` 中的属性 `OnUpdateStatus` 处理程序。 `CurrentPosition` 属性 `VideoView` 包含单元中的毫秒的新位置：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ...
            TimeSpan timeSpan = TimeSpan.FromMilliseconds(videoView.CurrentPosition);
            ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty, timeSpan);
        }
        ...
    }
}
```

此外，iOS 呈现器中一样 Android 呈现器调用 `SeekTo` 方法 `VideoView` 时 `Position` 属性已更改，但仅限于时更改为多个为一秒从不同 `CurrentPosition` 值 `VideoView`：

```
namespace FormsVideoLibrary.Droid
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, ARelativeLayout>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                if (Math.Abs(videoView.CurrentPosition - Element.Position.TotalMilliseconds) > 1000)
                {
                    videoView.SeekTo((int)Element.Position.TotalMilliseconds);
                }
            }
        }
        ...
    }
}
```

设置和获取 UWP 位置

UWP `VideoPlayerRenderer` 句柄 `Position` 中一样的 iOS 和 Android 的呈现器，但是，由于 `Position` UWP 属性 `MediaElement` 也 `TimeSpan` 值，则不必进行转换：

```
namespace FormsVideoLibrary.UWP
{
    public class VideoPlayerRenderer : ViewRenderer<VideoPlayer, MediaElement>
    {
        ...
        protected override void OnElementPropertyChanged(object sender, PropertyChangedEventArgs args)
        {
            ...
            else if (args.PropertyName == VideoPlayer.PositionProperty.PropertyName)
            {
                if (Math.Abs((Control.Position - Element.Position).TotalSeconds) > 1)
                {
                    Control.Position = Element.Position;
                }
            }
        }
        ...
        void OnUpdateStatus(object sender, EventArgs args)
        {
            ((IElementController)Element).SetValueFromRenderer(VideoPlayer.PositionProperty,
Control.Position);
        }
        ...
    }
}
```

计算 TimeToEnd 属性

有时视频播放器显示视频中的剩余时间。此值匹配位置始于当视频开始，并减少到零，视频结束时的视频的持续时间。

`VideoPlayer` 包括的只读 `TimeToEnd` 完全在类处理的属性基于更改 `Duration` 和 `Position` 属性：

```
namespace FormsVideoLibrary
{
    public class VideoPlayer : View, IVideoPlayerController
    {
        ...
        private static readonly BindablePropertyKey TimeToEndPropertyKey =
            BindableProperty.CreateReadOnly(nameof(TimeToEnd), typeof(TimeSpan), typeof(VideoPlayer), new
TimeSpan());

        public static readonly BindableProperty TimeToEndProperty = TimeToEndPropertyKey.BindableProperty;

        public TimeSpan TimeToEnd
        {
            private set { SetValue(TimeToEndPropertyKey, value); }
            get { return (TimeSpan)GetValue(TimeToEndProperty); }
        }

        void SetTimeToEnd()
        {
            TimeToEnd = Duration - Position;
        }
        ...
    }
}
```

`SetTimeToEnd` 从这两者的属性更改的处理程序调用方法 `Duration` 和 `Position`。

一个自定义的滑块，用于视频

可以编写一个位置栏中，自定义控件或使用 Xamarin.Forms `Slider` 或派生自类 `Slider`，如下所示 `PositionSlider` 类。类定义名为的两个新属性 `Duration` 和 `Position` 类型的 `TimeSpan` 想要将数据绑定到中的相同名称的两个属性 `VideoPlayer`。请注意，对默认的绑定模式 `Position` 属性是双向：

```
namespace FormsVideoLibrary
{
    public class PositionSlider : Slider
    {
        public static readonly BindableProperty DurationProperty =
            BindableProperty.Create(nameof(Duration), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(1),
                                    propertyChanged: (bindable, oldValue, newValue) =>
                                    {
                                        double seconds = ((TimeSpan)newValue).TotalSeconds;
                                        ((PositionSlider)bindable).Maximum = seconds <= 0 ? 1 : seconds;
                                    });

        public TimeSpan Duration
        {
            set { SetValue(DurationProperty, value); }
            get { return (TimeSpan)GetValue(DurationProperty); }
        }

        public static readonly BindableProperty PositionProperty =
            BindableProperty.Create(nameof(Position), typeof(TimeSpan), typeof(PositionSlider), new
TimeSpan(0),
                                    defaultBindingMode: BindingMode.TwoWay,
                                    propertyChanged: (bindable, oldValue, newValue) =>
                                    {
                                        double seconds = ((TimeSpan)newValue).TotalSeconds;
                                        ((PositionSlider)bindable).Value = seconds;
                                    });

        public TimeSpan Position
        {
            set { SetValue(PositionProperty, value); }
            get { return (TimeSpan)GetValue(PositionProperty); }
        }

        public PositionSlider()
        {
            PropertyChanged += (sender, args) =>
            {
                if (args.PropertyName == "Value")
                {
                    TimeSpan newPosition = TimeSpan.FromSeconds(Value);

                    if (Math.Abs(newPosition.TotalSeconds - Position.TotalSeconds) / Duration.TotalSeconds >
0.01)
                    {
                        Position = newPosition;
                    }
                }
            };
        }
    }
}
```

属性更改处理程序 `Duration` 属性集 `Maximum` 基础属性 `Slider` 到 `TotalSeconds` 属性 `TimeSpan` 值。同样的属性更改处理程序 `Position` 设置 `Value` 属性 `Slider`。这样，基础 `Slider` 跟踪的位置 `PositionSlider`。

`PositionSlider` 从基础更新 `Slider` 只有一个实例中：当用户操作 `Slider` 以指示视频应高级或反转到新位置。这中检测到 `PropertyChanged` 的构造函数中的处理程序 `PositionSlider`。处理程序检查中的更改 `Value` 属性，以及是否不同于 `Position` 属性，则 `Position` 属性设置从 `Value` 属性。

从理论上讲，内部 `if` 语句无法编写如下：

```
if (newPosition.Seconds != Position.Seconds)
{
    Position = newPosition;
}
```

但是的 Android 实现 `Slider` 具有仅 1,000 离散步骤而不考虑 `Minimum` 和 `Maximum` 设置。该视频的总长度超过 1,000 秒，然后两种不同 `Position` 值将对应于相同 `Value` 设置 `Slider` ，，这 `if` 语句会触发用户操作的假正 `Slider` 。它会改为检查新的位置和现有的位置大于总持续时间的百分之一更加安全。

使用 PositionSlider

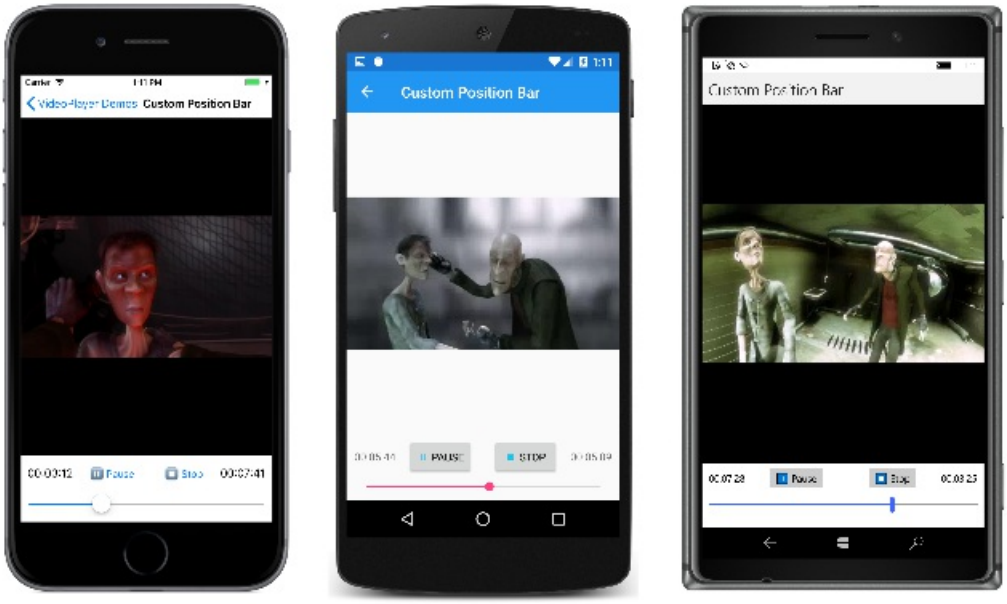
适用于 UWP 文档 `MediaElement` 警告有关绑定到 `Position` 属性由于属性频繁更新。文档建议，使用计时器查询 `Position` 属性。

这是一个不错的建议，但这三种 `VideoPlayerRenderer` 类已间接使用计时器来更新 `Position` 属性。 `Position` 的处理程序中更改属性 `UpdateStatus` 事件，激发每秒仅 10 次。

因此， `Position` 属性 `VideoPlayer` 可以绑定到 `Position` 属性 `PositionSlider` 不会出现性能问题，如中所示自定义位置栏页：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             x:Class="VideoPlayerDemos.CustomPositionBarPage"
             Title="Custom Position Bar">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <video:VideoPlayer x:Name="videoPlayer"
                           Grid.Row="0"
                           AreTransportControlsEnabled="False"
                           Source="{StaticResource ElephantsDream}" />

        ...

        <StackLayout Grid.Row="1"
                     Orientation="Horizontal"
                     Margin="10, 0"
                     BindingContext="{x:Reference videoPlayer}">

            <Label Text="{Binding Path=Position,
                                  StringFormat='{0:hh\\:mm\\:ss}'}"
                   VerticalOptions="Center"/>

            ...

            <Label Text="{Binding Path=TimeToEnd,
                                  StringFormat='{0:hh\\:mm\\:ss}'}"
                   VerticalOptions="Center" />
        </StackLayout>

        <video:PositionSlider Grid.Row="2"
                              Margin="10, 0, 10, 10"
                              BindingContext="{x:Reference videoPlayer}"
                              Duration="{Binding Duration}"
                              Position="{Binding Position}">
            <video:PositionSlider.Triggers>
                <DataTrigger TargetType="video:PositionSlider"
                             Binding="{Binding Status}"
                             Value="{x:Static video:VideoStatus.NotReady}">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </video:PositionSlider.Triggers>
        </video:PositionSlider>
    </Grid>
</ContentPage>
```

第一个省略号（…）隐藏 `ActivityIndicator`；它是与以前相同自定义传输页。请注意两个 `Label` 元素显示 `Position` 和 `TimeToEnd` 属性。这两个之间省略号 `Label` 元素隐藏这两个 `Button` 元素中所示自定义传输页面的播放、暂停和停止。代码隐藏逻辑也是相同自定义传输页。



到此结束的讨论 `VideoPlayer` 。

相关链接

- 视频播放器演示（示例）

Xamarin.Forms 数据绑定

2018/10/25 • • Edit Online

数据绑定是链接的两个对象的属性，使一个属性中的更改会自动反映在另一个属性的技术。数据绑定是模型-视图-视图模型 (MVVM) 应用程序体系结构的重要组成部分。

数据链接问题

Xamarin.Forms 应用程序包含一个或多个页面，其中每个通常包含多个名为的用户界面对象*视图*。该程序的首要任务之一是保持同步，这些视图并来跟踪不同的值或它们所代表的选择。视图通常从基础数据源，代表值和用户操作这些视图，可以更改该数据。基础数据视图更改时，必须反映该更改，同样，当基础数据更改时，所做的更改必须反映在视图中。

若要成功地处理此作业，必须在这些视图或基础数据中的更改的通知程序。常见的解决方案是定义发生更改时发出信号的事件。事件处理程序随后进行安装，这些更改的通知。它通过将数据从一个对象传输到另一个响应。但是，当存在多个视图时，必须也有很多事件处理程序，并获取涉及大量的代码。

数据绑定解决方案

数据绑定自动执行此作业，并将事件处理程序呈现不必要。（这些事件是仍有必要，但是，因为数据绑定基础结构使用它们。）在代码或 XAML，可以实现数据绑定，但它们是在它们有助于减少代码隐藏文件的大小的 XAML 中更常见。通过将声明性代码或标记替换为事件处理程序中的程序代码，该应用程序是简化和阐明了。

数据绑定中涉及的两个对象之一几乎始终是派生的元素 `View` 和窗体页面的可视界面的一部分。另一个对象是：

- 另一个 `View` 派生，通常在同一页上。
- 代码文件中的对象。

在如中的演示程序 **DataBindingDemos** 示例，两个数据绑定 `View` 派生类通常显示为的清晰和简洁的目的。但是，相同的原则可以应用于之间的数据绑定 `View` 和其他对象。当使用模型-视图-视图模型 (MVVM) 体系结构生成应用程序时，具有基础数据的类通常称为 ViewModel。

在以下一系列文章中探讨了数据绑定：

基本绑定

了解数据绑定目标和源之间的差异并查看代码和 XAML 中的简单数据绑定。

绑定模式

了解如何绑定模式，可以控制两个对象之间的数据流。

字符串格式设置

使用数据绑定进行格式化和显示为字符串的对象。

绑定路径

深入了解 `Path` 数据绑定来访问子属性和集合成员的属性。

绑定值转换器

绑定值转换器用于变更数据绑定中的值。

绑定回退

使数据绑定通过定义要使用在绑定过程失败时的回退值更稳健。

命令界面

实现 `Command` 具有数据绑定属性。

已编译的绑定

使用已编译的绑定，以提高数据绑定性能。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节
- XAML 标记扩展

Xamarin.Forms 基本绑定

2018/11/1 • • Edit Online

Xamarin.Forms 数据绑定链接一对两个对象，在至少一个通常是用户界面对象之间的属性。这两个对象调用*目标*并*源*:

- *目标*位于对象（或属性）设置数据绑定。
- *源*是由数据绑定引用的对象（和属性）。

这种区别有时可能有点令人困惑：在最简单的情况下，数据流从源到目标，这意味着，来自源属性的值设置目标属性的值。但是，在某些情况下，可以或者数据流从目标到源，或在两个方向。为避免混淆，请记住，目标始终是即使它是提供数据，而非接收数据在其设置数据绑定的对象。

与绑定上下文的绑定

尽管完全在 XAML 中通常指定数据绑定，很有意义，若要查看代码中的数据绑定。**基本代码绑定**页包含的 XAML 文件 `Label` 和一个 `Slider`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.BasicCodeBindingPage"
             Title="Basic Code Binding">
    <StackLayout Padding="10, 0">
        <Label x:Name="label"
               Text="TEXT"
               FontSize="48"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                Maximum="360"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

`Slider` 设置 0 到 360 之间的范围。此计划的目的是旋转 `Label` 通过操作 `Slider`。

无数据绑定，则会将设置 `ValueChanged` 的事件 `Slider` 事件处理程序访问 `Value` 的属性 `Slider` 并将该值设置为 `Rotation` 属性 `Label`。数据绑定自动执行该作业;事件处理程序和在其中的代码不再是必需的。

可以在任何派生的类的实例上设置绑定 `BindableObject`，其中包括 `Element`， `VisualElement`， `View`，和 `View` 派生类。始终在目标对象上设置绑定。绑定引用的源对象。若要设置数据绑定，请使用以下两个成员的目标类：

- `BindingContext` 属性指定的源对象。
- `SetBinding` 方法指定的目标属性和源属性。

在此示例中，`Label` 是绑定目标和 `Slider` 是绑定源。中的更改 `Slider` 源影响的旋转 `Label` 目标。数据源中的流复制到目标。

`SetBinding` 方法定义的 `BindableObject` 具有一个类型的参数 `BindingBase` 从中 `Binding` 类派生而来，但有其他 `SetBinding` 方法定义由 `BindableObjectExtensions` 类。中的代码隐藏文件**基本代码绑定**示例使用更简单 `SetBinding` 从此类的扩展方法。

```
public partial class BasicCodeBindingPage : ContentPage
{
    public BasicCodeBindingPage()
    {
        InitializeComponent();

        label.BindingContext = slider;
        label.SetBinding(Label.RotationProperty, "Value");
    }
}
```

`Label` 对象是绑定目标，因此这是在设置此属性，在调用该方法的对象。`BindingContext` 属性指示绑定源，即 `Slider`。

`SetBinding` 方法对绑定目标调用，但指定的目标属性和源属性。目标属性被指定为 `BindableProperty` 对象：`Label.RotationProperty`。源属性指定为字符串，并指示 `Value` 属性的 `Slider`。

`SetBinding` 方法揭示了其中一个数据绑定的最重要规则：

目标属性必须受可绑定的属性。

此规则表示目标对象必须是派生的类的实例 `BindableObject`。请参阅**可绑定属性**一文，了解概述的可绑定对象并可绑定属性。

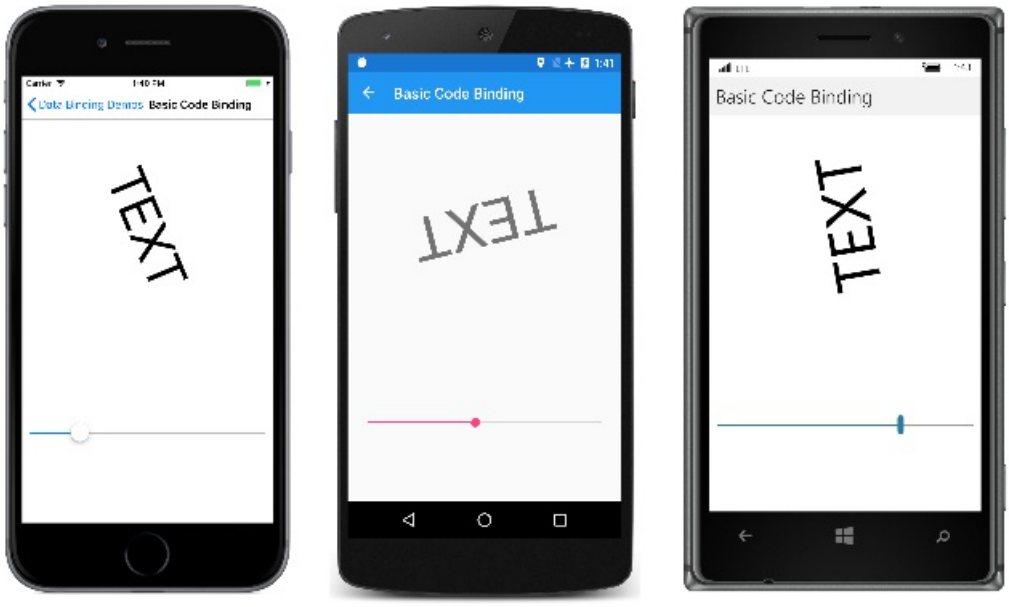
没有此类规则的源属性，指定为字符串。在内部，使用反射来访问实际的属性。在此特定情况，但是，`Value` 属性还受可绑定的属性。

代码可以是某种程度上简化：`RotationProperty` 通过定义可绑定属性 `VisualElement`，并由继承 `Label` 和 `ContentPage`，因此不需要的类名中 `SetBinding` 调用：

```
label.SetBinding(RotationProperty, "Value");
```

但是，包括类名称是很好的目标对象的提醒。

在操作 `Slider`，则 `Label` 相应地旋转：



基本 Xaml 绑定页等同于**基本代码绑定**不同之处在于它在 XAML 中定义的整个数据绑定：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.BasicXamlBindingPage"
             Title="Basic XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
               FontSize="80"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               BindingContext="{x:Reference Name=slider}"
               Rotation="{Binding Path=Value}" />

        <Slider x:Name="slider"
                Maximum="360"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

只需如下所示的代码中，是目标对象上设置数据绑定 `Label` 。涉及两个 XAML 标记扩展。这些是立即认出由大括号分隔符：

- `x:Reference` 需要标记扩展引用的源对象，即 `Slider` 名为 `slider` 。
- `Binding` 标记扩展链接 `Rotation` 的属性 `Label` 到 `Value` 属性 `Slider` 。

请参阅文章XAML 标记扩展XAML 标记扩展有关的详细信息。 `x:Reference` 标记扩展受 `ReferenceExtension` 类; `Binding` 受 `BindingExtension` 类。因为 XML 命名空间前缀指示， `x:Reference` 是 XAML 2009 规范的一部分而 `Binding` 是 Xamarin.Forms 的一部分。请注意，在大括号内显示没有引号。

它很容易忘记 `x:Reference` 标记扩展设置时 `BindingContext` 。常见错误地将属性设置为此类的绑定源的名称直接的是：

```
BindingContext="slider"
```

但这不正确。该标记将设置 `BindingContext` 属性设置为 `string` 其字符拼写"slider"的对象 ！

请注意，使用指定的源属性 `Path` 的属性 `BindingExtension` ，这对应于 `Path` 属性 `Binding` 类。

上显示的标记基本 XAML 绑定页可以简化：XAML 标记扩展，例如 `x:Reference` 并 `Binding` 可以*内容属性*属性定义，这对于 XAML标记扩展表示的属性名称不需要出现。 `Name` 属性是内容的属性 `x:Reference` ，并 `Path` 属性是 content 属性 `Binding` ，这意味着它们可以消除从表达式：

```
<Label Text="TEXT"
       FontSize="80"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand"
       BindingContext="{x:Reference slider}"
       Rotation="{Binding Value}" />
```

无需绑定上下文的绑定

`BindingContext` 属性是数据绑定的一个重要组成部分，但它并不总是需要。可以改为按指定的源对象 `SetBinding` 调用或 `Binding` 标记扩展。

了这一点替代方法代码绑定示例。XAML 文件是类似于基本代码绑定示例不同之处在于 `Slider` 为控件定义 `Scale` 属性 `Label` 。因此， `Slider` 为一系列设置–2 到 2：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.AlternativeCodeBindingPage"
             Title="Alternative Code Binding">
    <StackLayout Padding="10, 0">
        <Label x:Name="label"
               Text="TEXT"
               FontSize="40"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                Minimum="-2"
                Maximum="2"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

代码隐藏文件设置的绑定 `SetBinding` 方法定义的 `BindableObject`。参数是构造函数有关 `Binding` 类：

```
public partial class AlternativeCodeBindingPage : ContentPage
{
    public AlternativeCodeBindingPage()
    {
        InitializeComponent();

        label.SetBinding(Label.ScaleProperty, new Binding("Value", source: slider));
    }
}
```

`Binding` 构造函数具有 6 参数，因此 `source` 参数指定了命名参数。参数是 `slider` 对象。

运行此程序可能会感到有点惊讶：

在左侧的 iOS 屏幕显示屏幕页首次出现时的外观。其中是 `Label`？

问题在于 `Slider` 初始值为 0。这将导致 `Scale` 属性的 `Label` 也设置为 0，重写其默认值为 1。这会导致 `Label` 正在最初不可见。如 Android 和通用 Windows 平台 (UWP) 的屏幕截图所示，你能够 `Slider` 使 `Label` 再次出现，但其初始的亮点是令其不安。

您会发现在下一篇文章如何避免此问题通过初始化 `Slider` 默认值为 `Scale` 属性。

> **NOTE**
> `VisualElement` 类还定义 `ScaleX` 并 `ScaleY` 属性，可以缩放 `VisualElement` 中按不同方式水平和垂直方向。

替代 XAML 绑定页完全在 XAML 中显示等效的绑定：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.AlternativeXamlBindingPage"
             Title="Alternative XAML Binding">
    <StackLayout Padding="10, 0">
        <Label Text="TEXT"
               FontSize="40"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand"
               Scale="{Binding Source={x:Reference slider},
                               Path=Value}" />

        <Slider x:Name="slider"
                Minimum="-2"
                Maximum="2"
                VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

现在 `Binding` 标记扩展有两个属性设置，请 `Source` 和 `Path`，由逗号分隔。这些可以如果您愿意，在同一行上显示：

```
Scale="{Binding Source={x:Reference slider}, Path=Value}" />
```

`Source` 属性设置为嵌入 `x:Reference` 否则具有相同的语法与设置的标记扩展 `BindingContext`。请注意，大括号出现没有引号，必须用逗号分隔的两个属性。

内容属性的 `Binding` 标记扩展 `Path`，但 `Path=` 可以仅取消标记扩展的一部分，如果它是在表达式中的第一个属性。若要消除 `Path=` 过程中，您需要来交换两个属性：

```
Scale="{Binding Value, Source={x:Reference slider}}" />
```

尽管 XAML 标记扩展通常由大括号分隔，但它们还可以表示为对象元素：

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Source="{x:Reference slider}"
                 Path="Value" />
    </Label.Scale>
</Label>
```

现在 `Source` 和 `Path` 属性是常规 XAML 属性：显示在引号内的值以及属性不由逗号分隔。`x:Reference` 标记扩展可以成为对象元素：

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand">
    <Label.Scale>
        <Binding Path="Value">
            <Binding.Source>
                <x:Reference Name="slider" />
            </Binding.Source>
        </Binding>
    </Label.Scale>
</Label>
```

此语法并不常见，但有时有必要时涉及到复杂的对象。

到目前为止所示的示例设置 `BindingContext` 属性和 `Source` 的属性 `Binding` 到 `x:Reference` 标记扩展引用的页上的另一个视图。这两个属性属于类型 `Object`，并且它们可以设置为包括适用于绑定源的属性的任何对象。

在继续操作文章中，您会发现，可以设置 `BindingContext` 或 `Source` 属性设置为 `x:Static` 标记扩展引用的静态属性或字段中，值或 `StaticResource` 标记扩展引用中存储的对象资源字典中，或直接到对象，这是通常（但并非总是如此）的 ViewModel 实例。

`BindingContext` 属性也设置为 `Binding` 对象，以便 `Source` 并 `Path` 的属性 `Binding` 定义的绑定上下文。

绑定上下文继承

在本文中，你已了解，你可以指定源对象使用 `BindingContext` 属性或 `Source` 属性的 `Binding` 对象。如果两者都设置 `Source` 的属性 `Binding` 优先于 `BindingContext`。

`BindingContext` 属性具有非常重要的特征：

设置 `BindingContext` 通过可视树继承属性。

正如您将看到，这会非常方便，用于简化绑定表达式，并在某些情况下—尤其是在模型-视图-视图模型 (MVVM) 方案—很重要。

绑定上下文继承示例是简单的绑定上下文的继承演示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.BindingContextInheritancePage"
             Title="BindingContext Inheritance">
    <StackLayout Padding="10">

        <StackLayout VerticalOptions="FillAndExpand"
                     BindingContext="{x:Reference slider}">

            <Label Text="TEXT"
                   FontSize="80"
                   HorizontalOptions="Center"
                   VerticalOptions="EndAndExpand"
                   Rotation="{Binding Value}" />

            <BoxView Color="#800000FF"
                     WidthRequest="180"
                     HeightRequest="40"
                     HorizontalOptions="Center"
                     VerticalOptions="StartAndExpand"
                     Rotation="{Binding Value}" />
        </StackLayout>

        <Slider x:Name="slider"
                Maximum="360" />

    </StackLayout>
</ContentPage>
```

`BindingContext`的属性`StackLayout`设置为`slider`对象。由已继承此绑定上下文`Label`并`BoxView`，这两个都没有其`Rotation`属性设置为`Value`属性`Slider`：

在中下一篇文章，你将看到如何*绑定模式*可以更改目标和源对象之间的数据流。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 绑定模式

2018/11/13 • • Edit Online

中前一篇文章，则**替代方法代码绑定并替代 XAML 绑定特色的**页 `Label` 使用其 `Scale` 属性绑定到 `Value` 属性的 `Slider` 。因为 `Slider` 初始值为 0，这导致 `Scale` 的属性 `Label` 设置为 0，而不是 1，和 `Label` 消失。

在中 **DataBindingDemos** 示例中，**反向绑定**页是类似于前一篇文章中的程序，只不过上定义的数据绑定 `Slider` 而不是在 `Label` ：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DataBindingDemos.ReverseBindingPage"
             Title="Reverse Binding">
    <StackLayout Padding="10, 0">

        <Label x:Name="label"
               Text="TEXT"
               FontSize="80"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                VerticalOptions="CenterAndExpand"
                Value="{Binding Source={x:Reference label},
                                Path=Opacity}" />
    </StackLayout>
</ContentPage>
```

首先，这看起来可能向后：现在 `Label` 是数据绑定源和 `Slider` 是目标。绑定引用 `Opacity` 属性的 `Label` ，其中包含默认值为 1。

正如您所料， `Slider` 初始化为值 1 从初始 `Opacity` 的值 `Label` 。在左侧 iOS 屏幕截图所示：

但您可能会很吃惊， `Slider` 仍将有效，如 Android 和 UWP 的屏幕截图所示。这似乎表明，数据绑定的工作方式更好时 `Slider` 是绑定目标而不是 `Label` 因为初始化工作方式与我们所料。

之间的差异**反向**绑定示例和更早的示例涉及*绑定模式*。

默认绑定模式

绑定模式指定的成员 `BindingMode` 枚举：

- `Default`
- `TwoWay` – 数据源和目标之间出现这两种方式
- `OneWay` – 数据源中发送到目标
- `OneWayToSource` – 数据从目标发送到源
- `OneTime` – 数据源从转到目标，但仅当 `BindingContext` 更改（新 Xamarin.Forms 3.0）

每个可绑定属性具有默认值绑定时创建的可绑定属性，设置的模式和可以在中找到 `DefaultBindingMode` 属性的 `BindableProperty` 对象。当该属性是数据绑定目标时实际上这一默认绑定模式指示的模式。

如大多数属性的默认绑定模式 `Rotation`， `Scale`，并 `Opacity` 是 `OneWay`。当这些属性是数据绑定目标时，则目标属性设置从源。

但是的默认绑定模式 `Value` 的属性 `Slider` 是 `TwoWay`。这意味着，当 `Value` 属性是数据绑定目标，则目标（像往常一样）设置从源但从目标还设置源。这样做可允许 `Slider` 若要从初始设置 `Opacity` 值。

此双向绑定可能看起来创建无限循环，但未按预期进行。可绑定属性不发出信号属性更改，除非该属性实际上发生变化。这可以防止无限循环。

双向绑定

最可绑定属性具有的默认绑定模式 `OneWay` 但以下属性具有默认绑定模式的 `TwoWay`：

- `Date` 属性 `DatePicker`
- `Text` 属性的 `Editor`， `Entry`， `SearchBar`，和 `EntryCell`
- `IsRefreshing` 属性 `ListView`
- `SelectedItem` 属性 `MultiPage`
- `SelectedIndex` 和 `SelectedItem` 的属性 `Picker`
- `Value` 属性的 `Slider` 和 `Stepper`
- `IsToggled` 属性 `Switch`
- `On` 属性 `SwitchCell`
- `Time` 属性 `TimePicker`

这些特定的属性定义为 `TwoWay` 非常充分的理由：

当使用模型-视图-视图模型 (MVVM) 应用程序体系结构中使用数据绑定时，ViewModel 类是数据绑定源和视图，其中包含视图如 `Slider`，是数据绑定的目标。MVVM 绑定类似于反向绑定大于上一示例中的绑定的示例。它是属性的很有可能您想要使用在 ViewModel 中，相应的值进行初始化的页面上的每个视图，但在视图中的更改应还影响 ViewModel 属性。

使用默认绑定模式的属性 `TwoWay` 是最有可能在 MVVM 方案中使用这些属性。

一个单向到源绑定

只读的可绑定属性具有默认绑定模式的 `OneWayToSource`。设置了默认绑定模式的仅一个读/写可绑定属性 `OneWayToSource`：

- `SelectedItem` 属性 `ListView`

基本原理是，在绑定 `SelectedItem` 属性应导致绑定源设置。在本文后面的示例重写该行为。

一次性绑定

多个属性具有默认绑定模式的 `OneTime`。这些是：

- `IsTextPredictionEnabled` 属性 `Entry`
- `Text``BackgroundColor`，并 `Style` 的属性 `Span`。

绑定模式的属性为目标 `OneTime` 仅绑定上下文更改时，会更新。对于这些目标属性的绑定，这将简化绑定基础结构，因为不需要监视的源属性中的更改。

Viewmodel 和属性更改通知

简单颜色选择器页演示如何使用简单的 ViewModel。数据绑定允许用户选择颜色使用三个 `Slider` 元素的色调、饱和度和亮度。

ViewModel 是数据绑定源。ViewModel does不来定义可绑定属性，但它实现一种通知机制，允许绑定基础结构的属性值更改时得到通知。此通知机制 `INotifyPropertyChanged` 接口，定义一个名为的单个属性 `PropertyChanged`。通常实现此接口的类激发事件时的一个公共属性值的改变。事件不需要，如果该属性永远不会更改触发。(`INotifyPropertyChanged` 也会实现接口 `BindableObject` 和一个 `PropertyChanged` 可绑定属性值更改时触发事件。)

`HslColorViewModel` 类定义了五个属性：`Hue`， `Saturation`， `Luminosity`，和 `Color` 相互关联的属性。当任何一个的三个颜色组件更改值时 `Color` 属性将重新计算，和 `PropertyChanged` 的事件触发的所有四个属性：

```
public class HslColorViewModel : INotifyPropertyChanged
{
    Color color;
    string name;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Hue
    {
        set
        {
            if (color.Hue != value)
            {
                Color = Color.FromHsla(value, color.Saturation, color.Luminosity);
            }
        }
        get
        {
            return color.Hue;
        }
    }

    public double Saturation
    {
        set
        {
            if (color.Saturation != value)
            {
                Color = Color.FromHsla(color.Hue, value, color.Luminosity);
            }
        }
        get
        {
            return color.Saturation;
        }
    }

    public double Luminosity
    {
        set
        {
            if (color.Luminosity != value)
            {
                Color = Color.FromHsla(color.Hue, color.Saturation, value);
            }
```

```
            }
        }
        get
        {
            return color.Luminosity;
        }
    }

    public Color Color
    {
        set
        {
            if (color != value)
            {
                color = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Hue"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Saturation"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Luminosity"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));

                Name = NamedColor.GetNearestColorName(color);
            }
        }
        get
        {
            return color;
        }
    }

    public string Name
    {
        private set
        {
            if (name != value)
            {
                name = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
            }
        }
        get
        {
            return name;
        }
    }
}
```

当 `Color` 属性更改、静态 `GetNearestColorName` 中的方法 `NamedColor` 类 (也包含在**DataBindingDemos**解决方案) 获取最接近已命名的颜色，并设置 `Name` 属性。这 `Name` 属性有一个私有 `set` 访问器，因此它不能将设置从类外部的。

当 ViewModel 设置用作绑定源时，绑定基础结构附加到一个处理程序 `PropertyChanged` 事件。这样一来的绑定可以对属性的更改的通知和能够更改后的值中设置目标属性。

但是，当目标属性 (或 `Binding` 目标属性上的定义) 具有 `BindingMode` 的 `OneTime`，则没有必要的绑定基础结构上附加一个处理程序 `PropertyChanged` 事件。更新目标属性时，才 `BindingContext` 更改，并不在源属性本身更改。

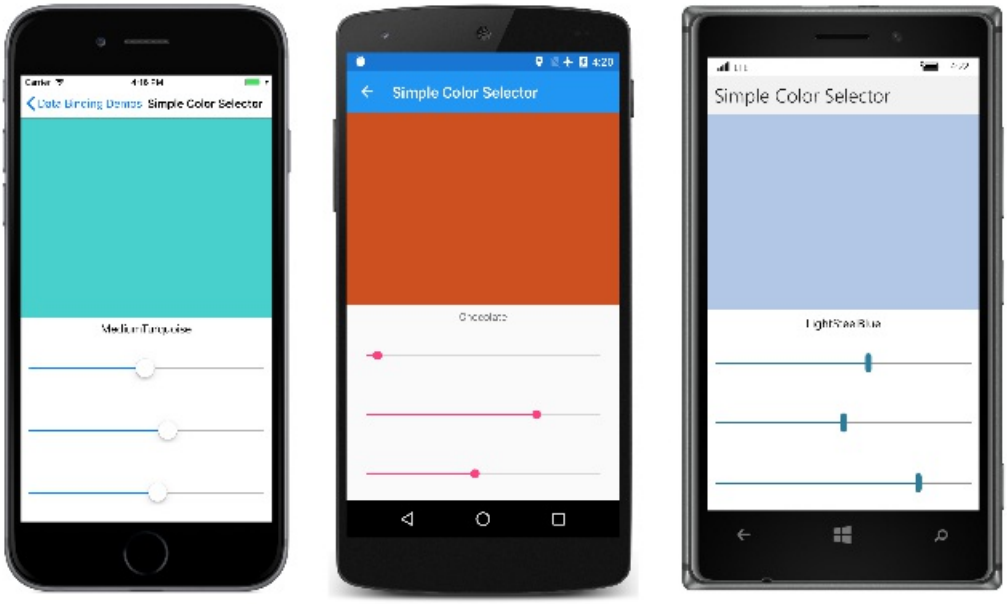
简单颜色选择器XAML 文件实例化 `HslColorViewModel` 中页面的资源字典并将初始化 `Color` 属性。
`BindingContext` 的属性 `Grid` 设置为 `StaticResource` 绑定扩展来引用该资源：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.SimpleColorSelectorPage">

    <ContentPage.Resources>
        <ResourceDictionary>
            <local:HslColorViewModel x:Key="viewModel"
                                     Color="MediumTurquoise" />

            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <Grid BindingContext="{StaticResource viewModel}">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <BoxView Color="{Binding Color}"
                 Grid.Row="0" />

        <StackLayout Grid.Row="1"
                     Margin="10, 0">

            <Label Text="{Binding Name}"
                   HorizontalTextAlignment="Center" />

            <Slider Value="{Binding Hue}" />

            <Slider Value="{Binding Saturation}" />

            <Slider Value="{Binding Luminosity}" />
        </StackLayout>
    </Grid>
</ContentPage>
```

`BoxView`， `Label`，和三个`Slider`视图继承从绑定上下文`Grid`。这些视图是引用 ViewModel 中的源属性的所有绑定目标。有关`Color`的属性`BoxView`，和`Text`属性`Label`，数据绑定是`OneWay`：从 ViewModel 中的属性设置视图中的属性。

`Value`的属性`Slider`，，但`TwoWay`。这样，每个`Slider`若要从 ViewModel 中，以及若要设置从每个 viewmodel 设置`Slider`。

首次运行该程序时，`BoxView`， `Label`，和三个`Slider`元素是从基于初始 ViewModel 的所有组`Color`时实例化 ViewModel 设置的属性。在左侧 iOS 屏幕截图所示：



操控滑块 `BoxView` 和 `Label` 相应地，Android 和 UWP 的屏幕截图所示更新。

实例化 ViewModel 中的资源字典是一个常用方法。它也是可以实例化 ViewModel 中的属性元素标记 `BindingContext` 属性。在中简单颜色选择器XAML 文件中，请尝试删除 `HslColorViewModel` 从资源字典并将其设置为 `BindingContext` 属性 `Grid` 如下所示：

```
<Grid>
    <Grid.BindingContext>
        <local:HslColorViewModel Color="MediumTurquoise" />
    </Grid.BindingContext>

    ...

</Grid>
```

可以通过多种方式中设置的绑定上下文。有时，代码隐藏文件实例化 ViewModel 并将其设置为 `BindingContext` 页属性。这些是所有有效的方法。

重写绑定模式

如果在目标属性上的默认绑定模式不适用于特定的数据绑定，则可以通过设置替代 `Mode` 的属性 `Binding`（或 `Mode` 的属性 `Binding` 标记扩展）的成员之一 `BindingMode` 枚举。

但是，将设置 `Mode` 属性设置为 `TwoWay` 始终无法按您所料。例如，请尝试修改替代 **XAML** 绑定XAML 文件以包括 `TwoWay` 绑定定义中：

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand"
       Scale="{Binding Source={x:Reference slider},
                       Path=Value,
                       Mode=TwoWay}" />
```

它可能会发生的 `Slider` 将初始化为初始值的 `Scale` 属性，它是 1，但未按预期进行。当 `TwoWay` 初始化绑定，目标从源中设置第一次，这意味着 `Scale` 属性设置为 `Slider` 默认值为 0。当 `TwoWay` 上设置绑定 `Slider`，则 `Slider` 从源最初设置。

您可以将绑定模式设置为 `OneWayToSource` 中替代 **XAML** 绑定示例：

```
<Label Text="TEXT"
       FontSize="40"
       HorizontalOptions="Center"
       VerticalOptions="CenterAndExpand"
       Scale="{Binding Source={x:Reference slider},
                       Path=Value,
                       Mode=OneWayToSource}" />
```

现在 `Slider` 初始化为 1 (默认值 `Scale`) 但操作 `Slider` 不会影响 `Scale` 属性，因此这不是很有用。

> **NOTE**
> `VisualElement` 类还定义 `ScaleX` 并 `ScaleY` 属性，可以缩放 `VisualElement` 中按不同方式水平和垂直方向。

重写的默认绑定模式非常有用的应用程序 `TwoWay` 涉及 `SelectedItem` 属性的 `ListView`。默认绑定模式是 `OneWayToSource`。如果在设置数据绑定 `SelectedItem` 属性来引用 ViewModel 中的源属性，则该 source 属性将设置从 `ListView` 所选内容。但是，在某些情况下，你可能还想 `ListView` 从 ViewModel 进行初始化。

的示例设置页演示此技术。此页表示应用程序设置，经常在 ViewModel 中，这种中定义的简单实现 `SampleSettingsViewModel` 文件：

```
public class SampleSettingsViewModel : INotifyPropertyChanged
{
    string name;
    DateTime birthDate;
    bool codesInCSharp;
    double numberOfCopies;
    NamedColor backgroundNamedColor;

    public event PropertyChangedEventHandler PropertyChanged;

    public SampleSettingsViewModel(IDictionary<string, object> dictionary)
    {
        Name = GetDictionaryEntry<string>(dictionary, "Name");
        BirthDate = GetDictionaryEntry(dictionary, "BirthDate", new DateTime(1980, 1, 1));
        CodesInCSharp = GetDictionaryEntry<bool>(dictionary, "CodesInCSharp");
        NumberOfCopies = GetDictionaryEntry(dictionary, "NumberOfCopies", 1.0);
        BackgroundNamedColor = NamedColor.Find(GetDictionaryEntry(dictionary, "BackgroundNamedColor",
"White"));
    }

    public string Name
    {
        set { SetProperty(ref name, value); }
        get { return name; }
    }

    public DateTime BirthDate
    {
        set { SetProperty(ref birthDate, value); }
        get { return birthDate; }
    }

    public bool CodesInCSharp
    {
        set { SetProperty(ref codesInCSharp, value); }
        get { return codesInCSharp; }
    }

    public double NumberOfCopies
    {
        set { SetProperty(ref numberOfCopies, value); }
        get { return numberOfCopies; }
```

```
        get { return numberOfCopies; }
    }

    public NamedColor BackgroundNamedColor
    {
        set
        {
            if (SetProperty(ref backgroundNamedColor, value))
            {
                OnPropertyChanged("BackgroundColor");
            }
        }
        get { return backgroundNamedColor; }
    }

    public Color BackgroundColor
    {
        get { return BackgroundNamedColor?.Color ?? Color.White; }
    }

    public void SaveState(IDictionary<string, object> dictionary)
    {
        dictionary["Name"] = Name;
        dictionary["BirthDate"] = BirthDate;
        dictionary["CodesInCSharp"] = CodesInCSharp;
        dictionary["NumberOfCopies"] = NumberOfCopies;
        dictionary["BackgroundNamedColor"] = BackgroundNamedColor.Name;
    }

    T GetDictionaryEntry<T>(IDictionary<string, object> dictionary, string key, T defaultValue = default(T))
    {
        return dictionary.ContainsKey(key) ? (T)dictionary[key] : defaultValue;
    }

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

每个应用程序设置是一个属性，将保存到一个名为方法中的 Xamarin.Forms 属性字典 `SaveState` 以及从该字典的构造函数中加载。类的底部是两个方法，可以帮助简化 Viewmodel，使它们更不易出错。 `OnPropertyChanged` 底部方法具有一个可选参数，设置为调用属性。以字符串形式指定的属性名称时，这可以避免拼写错误。

`SetProperty` 类中的方法执行更多：它将被存储为一个字段的值设置为属性值进行比较，并仅调用 `OnPropertyChanged` 两个值不相等。

`SampleSettingsViewModel` 类定义的背景颜色的两个属性：`BackgroundNamedColor` 属性属于类型 `NamedColor` ，其中一个类也包含在**DataBindingDemos**解决方案。 `BackgroundColor` 属性属于类型 `Color` ，并从获取 `Color` 属性的 `NamedColor` 对象。

`NamedColor` 类使用.NET 反射来枚举在 Xamarin.Forms 中的所有静态公共字段 `Color` 结构，并将其存储在可从静态访问集合中其名称与 `All` 属性：

```
public class NamedColor : IEquatable<NamedColor>, IComparable<NamedColor>
```

```
{
    // Instance members
    private NamedColor()
    {
    }

    public string Name { private set; get; }

    public string FriendlyName { private set; get; }

    public Color Color { private set; get; }

    public string RgbDisplay { private set; get; }

    public bool Equals(NamedColor other)
    {
        return Name.Equals(other.Name);
    }

    public int CompareTo(NamedColor other)
    {
        return Name.CompareTo(other.Name);
    }

    // Static members
    static NamedColor()
    {
        List<NamedColor> all = new List<NamedColor>();
        StringBuilder stringBuilder = new StringBuilder();

        // Loop through the public static fields of the Color structure.
        foreach (FieldInfo fieldInfo in typeof(Color).GetRuntimeFields())
        {
            if (fieldInfo.IsPublic &&
                fieldInfo.IsStatic &&
                fieldInfo.FieldType == typeof(Color))
            {
                // Convert the name to a friendly name.
                string name = fieldInfo.Name;
                stringBuilder.Clear();
                int index = 0;

                foreach (char ch in name)
                {
                    if (index != 0 && Char.IsUpper(ch))
                    {
                        stringBuilder.Append(' ');
                    }
                    stringBuilder.Append(ch);
                    index++;
                }

                // Instantiate a NamedColor object.
                Color color = (Color)fieldInfo.GetValue(null);

                NamedColor namedColor = new NamedColor
                {
                    Name = name,
                    FriendlyName = stringBuilder.ToString(),
                    Color = color,
                    RgbDisplay = String.Format("{0:X2}-{1:X2}-{2:X2}",
                                               (int)(255 * color.R),
                                               (int)(255 * color.G),
                                               (int)(255 * color.B))
                };

                // Add it to the collection.
                all.Add(namedColor);
            }
```

```
        }
        all.TrimExcess();
        all.Sort();
        All = all;
    }

    public static IList<NamedColor> All { private set; get; }

    public static NamedColor Find(string name)
    {
        return ((List<NamedColor>)All).Find(nc => nc.Name == name);
    }

    public static string GetNearestColorName(Color color)
    {
        double shortestDistance = 1000;
        NamedColor closestColor = null;

        foreach (NamedColor namedColor in NamedColor.All)
        {
            double distance = Math.Sqrt(Math.Pow(color.R - namedColor.Color.R, 2) +
                                        Math.Pow(color.G - namedColor.Color.G, 2) +
                                        Math.Pow(color.B - namedColor.Color.B, 2));

            if (distance < shortestDistance)
            {
                shortestDistance = distance;
                closestColor = namedColor;
            }
        }
        return closestColor.Name;
    }
}
```

`App`类中**DataBindingDemos**项目定义一个名为属性`Settings`类型的`SampleSettingsViewModel`。初始化此属性时`App`类实例化，并`SaveState`时调用方法`OnSleep`调用方法：

```
public partial class App : Application
{
    public App()
    {
        InitializeComponent();

        Settings = new SampleSettingsViewModel(Current.Properties);

        MainPage = new NavigationPage(new MainPage());
    }

    public SampleSettingsViewModel Settings { private set; get; }

    protected override void OnStart()
    {
        // Handle when your app starts
    }

    protected override void OnSleep()
    {
        // Handle when your app sleeps
        Settings.SaveState(Current.Properties);
    }

    protected override void OnResume()
    {
        // Handle when your app resumes
    }
}
```

应用程序生命周期方法的详细信息，请参阅文章**应用程序生命周期**。

几乎所有其他项中处理**SampleSettingsPage.xaml**文件。`BindingContext`页的设置使用`Binding`标记扩展：绑定源是静态`Application.Current`属性，它是该实例的`App`类在项目中，并且`Path`设置为`Settings`属性，它是`SampleSettingsViewModel`对象：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.SampleSettingsPage"
             Title="Sample Settings"
             BindingContext="{Binding Source={x:Static Application.Current},
                                      Path=Settings}">

    <StackLayout BackgroundColor="{Binding BackgroundColor}"
                 Padding="10"
                 Spacing="10">

        <StackLayout Orientation="Horizontal">
            <Label Text="Name: "
                   VerticalOptions="Center" />

            <Entry Text="{Binding Name}"
                   Placeholder="your name"
                   HorizontalOptions="FillAndExpand"
                   VerticalOptions="Center" />
        </StackLayout>

        <StackLayout Orientation="Horizontal">
            <Label Text="Birth Date: "
                   VerticalOptions="Center" />

            <DatePicker Date="{Binding BirthDate}"
                        HorizontalOptions="FillAndExpand"
                        VerticalOptions="Center" />
```

```
        </StackLayout>

        <StackLayout Orientation="Horizontal">
            <Label Text="Do you code in C#? "
                   VerticalOptions="Center" />

            <Switch IsToggled="{Binding CodesInCSharp}"
                    VerticalOptions="Center" />
        </StackLayout>

        <StackLayout Orientation="Horizontal">
            <Label Text="Number of Copies: "
                   VerticalOptions="Center" />

            <Stepper Value="{Binding NumberOfCopies}"
                     VerticalOptions="Center" />

            <Label Text="{Binding NumberOfCopies}"
                   VerticalOptions="Center" />
        </StackLayout>

        <Label Text="Background Color:" />

        <ListView x:Name="colorListView"
                  ItemsSource="{x:Static local:NamedColor.All}"
                  SelectedItem="{Binding BackgroundNamedColor, Mode=TwoWay}"
                  VerticalOptions="FillAndExpand"
                  RowHeight="40">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <BoxView Color="{Binding Color}"
                                     HeightRequest="32"
                                     WidthRequest="32"
                                     VerticalOptions="Center" />

                            <Label Text="{Binding FriendlyName}"
                                   FontSize="24"
                                   VerticalOptions="Center" />
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

页的所有子项都继承的绑定上下文。大部分此页上的其他绑定到中的属性都是 `SampleSettingsViewModel`。 `BackgroundColor` 属性用于设置 `BackgroundColor` 的属性 `StackLayout`，和 `Entry`， `DatePicker`， `Switch`，和 `Stepper` 属性被绑定到 ViewModel 中的其他属性。

`ItemsSource` 的属性 `ListView` 设置为静态 `NamedColor.All` 属性。这样就会填写 `ListView` 所有 `NamedColor` 实例。中每一项 `ListView`，该项目的绑定上下文设置为 `NamedColor` 对象。 `BoxView` 并 `Label` 中 `ViewCell` 绑定到属性中 `NamedColor`。

`SelectedItem` 的属性 `ListView` 属于类型 `NamedColor`，并将其绑定到 `BackgroundNamedColor` 属性 `SampleSettingsViewModel`：

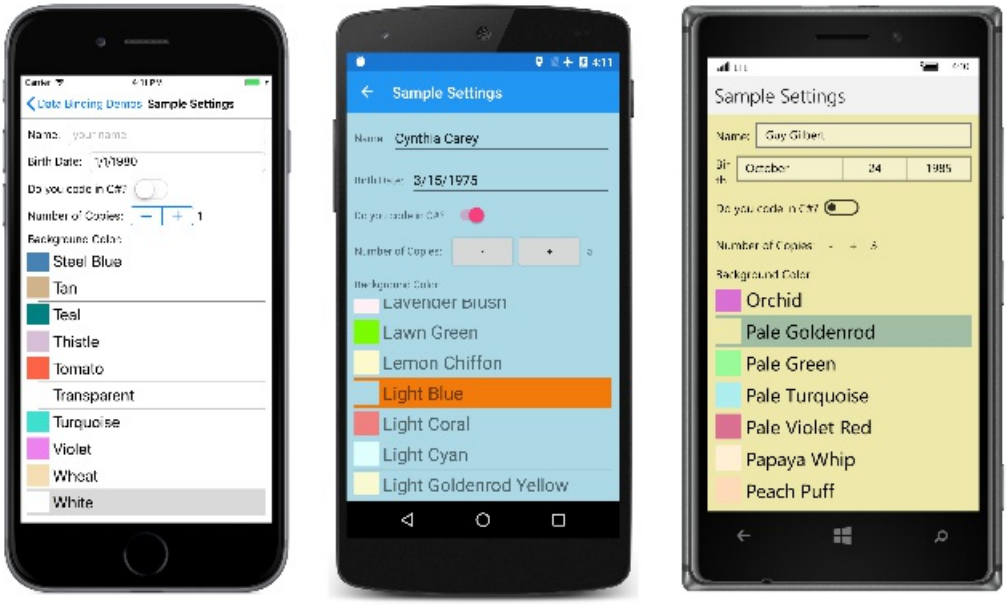
```
SelectedItem="{Binding BackgroundNamedColor, Mode=TwoWay}"
```

默认绑定模式 `SelectedItem` 是 `OneWayToSource`，用于从所选的项设置的 ViewModel 属性。 `TwoWay` 模式允许 `SelectedItem` 从 ViewModel 进行初始化。

但是，当 `SelectedItem` 这种方式，设置 `ListView` 不会自动滚动以显示所选的项。在代码隐藏文件中的一小段代码，有必要：

```
public partial class SampleSettingsPage : ContentPage
{
    public SampleSettingsPage()
    {
        InitializeComponent();

        if (colorListView.SelectedItem != null)
        {
            colorListView.ScrollTo(colorListView.SelectedItem,
                                   ScrollToPosition.MakeVisible,
                                   false);
        }
    }
}
```

当首次运行时，在左侧的 iOS 屏幕快照显示了的程序。中的构造函数 `SampleSettingsViewModel` 初始化的背景色为白色，并且在选择的内容 `ListView`：



其他两个屏幕快照显示更改后的设置。在与此页进行试验，记得将进入睡眠状态或终止在设备或仿真程序认为其正在运行的程序。终止该程序从 Visual Studio 调试器将不会导致 `OnSleep` 重写中 `App` 类来调用。

将在下一篇文章中了解如何指定**字符串格式设置**上设置的数据绑定 `Text` 属性 `Label`。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 字符串格式设置

2018/7/13 • • Edit Online

有时很方便地使用数据绑定来显示对象或值的字符串表示形式。例如，你可能想要使用 `Label` 若要显示的当前值 `Slider`。在此数据绑定中，`Slider` 是源和目标数据库中均 `Text` 属性的 `Label`。

在代码中显示字符串，该功能最强大的工具时，静态 `String.Format` 方法。格式设置字符串包含格式设置代码特定于不同类型的对象，并可包含要设置格式的值以及其他文本。请参阅.NET 中的格式设置类型一文，了解字符串格式设置的详细信息。

StringFormat 属性

此功能被传送到数据绑定：您设置 `StringFormat` 的属性 `Binding`（或 `StringFormat` 属性 `Binding` 标记扩展）到标准.NET 格式设置字符串包含一个占位符：

```
<Slider x:Name="slider" />
<Label Text="{Binding Source={x:Reference slider},
                      Path=Value,
                      StringFormat='The slider value is {0:F2}'}" />
```

请注意，由以帮助避免将大括号视为另一个 XAML 标记扩展的 XAML 分析器的单引号（撇号）字符分隔格式设置字符串。否则，此字符串没有单引号字符是相同的字符串将显示一个浮点值对的调用中 `String.Format`。格式规范 `F2` 导致带两位小数显示的值。

`StringFormat` 属性仅在如果目标属性的类型有意义 `string`，并且绑定模式是 `OneWay` 或 `TwoWay`。对于双向绑定，`StringFormat` 是仅适用于从源向目标传递的值。

正如您将在下一篇文章中看到上绑定路径，数据绑定可能会变得非常复杂而又费解。在调试时这些数据绑定，可以添加 `Label` 到 XAML 文件与 `StringFormat` 显示某些中间结果。即使使用仅用于显示对象的类型，可帮助你。

字符串格式设置页说明了几种用法的 `StringFormat` 属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:sys="clr-namespace:System;assembly=mscorlib"
             x:Class="DataBindingDemos.StringFormattingPage"
             Title="String Formatting">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <Style TargetType="BoxView">
                <Setter Property="Color" Value="Blue" />
                <Setter Property="HeightRequest" Value="2" />
                <Setter Property="Margin" Value="0, 5" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Margin="10">
        <Slider x:Name="slider" />
        <Label Text="{Binding Source={x:Reference slider},
                              Path=Value,
                              StringFormat='The slider value is {0:F2}'}" />

        <BoxView />

        <TimePicker x:Name="timePicker" />
        <Label Text="{Binding Source={x:Reference timePicker},
                              Path=Time,
                              StringFormat='The TimeSpan is {0:c}'}" />

        <BoxView />

        <Entry x:Name="entry" />
        <Label Text="{Binding Source={x:Reference entry},
                              Path=Text,
                              StringFormat='The Entry text is &amp;quot;{0}&amp;quot;'}" />

        <BoxView />

        <StackLayout BindingContext="{x:Static sys:DateTime.Now}">
            <Label Text="{Binding}" />
            <Label Text="{Binding Path=Ticks,
                                  StringFormat='{0:N0} ticks since 1/1/1'}" />
            <Label Text="{Binding StringFormat='The {{0:MMMM}} specifier produces {0:MMMM}'}" />
            <Label Text="{Binding StringFormat='The long date is {0:D}'}" />
        </StackLayout>

        <BoxView />

        <StackLayout BindingContext="{x:Static sys:Math.PI}">
            <Label Text="{Binding}" />
            <Label Text="{Binding StringFormat='PI to 4 decimal points = {0:F4}'}" />
            <Label Text="{Binding StringFormat='PI in scientific notation = {0:E7}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

上的绑定 `Slider` 并 `TimePicker` 显示了如何使用格式规范的特定于 `double` 和 `TimeSpan` 数据类型。 `StringFormat` 显示从文本 `Entry` 视图演示如何通过使用在格式设置字符串中指定两个双引号 `&quot;` HTML 实体。

XAML 文件中的下一个部分是 `StackLayout` 与 `BindingContext` 设置为 `x:Static` 标记扩展引用静态 `DateTime.Now` 属性。第一个绑定具有任何属性：

```
<Label Text="{Binding}" />
```

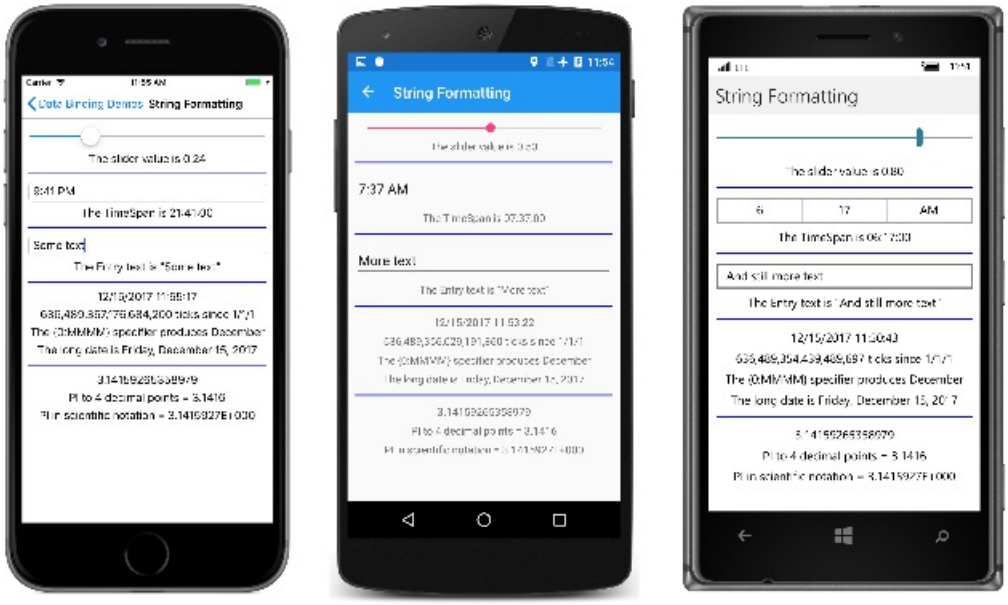
这只是显示 `DateTime` 的值 `BindingContext` 使用默认格式设置。第二个绑定显示 `Ticks` 的属性 `DateTime`，而其他两个绑定显示 `DateTime` 本身具有特定格式设置。请注意，这 `StringFormat`：

```
<Label Text="{Binding StringFormat='The {{0:MMMM}} specifier produces {0:MMMM}'}" />
```

如果需要显示在格式设置字符串中的向左或右大括号，只需使用其中一对。

最后一个部分集 `BindingContext` 的值 `Math.PI`，并使用默认格式设置和两个不同类型的数字格式显示。

下面是在所有三个平台上运行的程序：



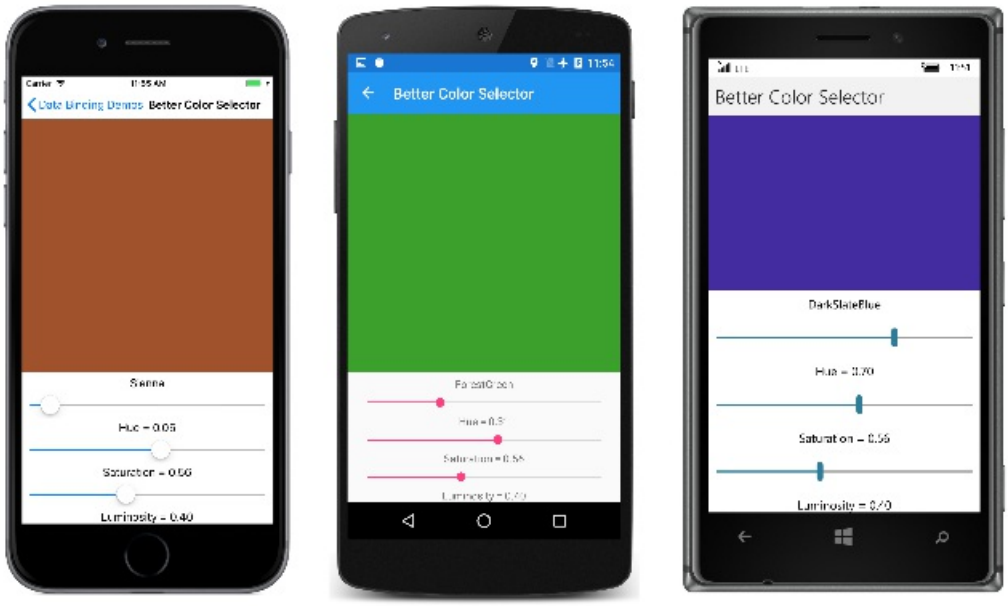
Viewmodel 和字符串格式设置

当使用 `Label` 并 `StringFormat` 若要显示的一个视图，它也是 ViewModel 的目标值，可以定义从视图到绑定 `Label` 或从 ViewModel 到达 `Label`。一般情况下，第二种方法是最佳的因为它将验证的视图和 ViewModel 之间的绑定都正常工作。

这种方法所示更好的颜色选择器示例中，使用同一个 ViewModel 作为简单颜色选择器中所示的程序绑定模式文章：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.BetterColorSelectorPage"
             Title="Better Color Selector">

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <StackLayout.BindingContext>
            <local:HslColorViewModel Color="Sienna" />
        </StackLayout.BindingContext>

        <BoxView Color="{Binding Color}"
                 VerticalOptions="FillAndExpand" />

        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />

            <Slider Value="{Binding Hue}" />
            <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />

            <Slider Value="{Binding Saturation}" />
            <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />

            <Slider Value="{Binding Luminosity}" />
            <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

现在有三对 `Slider` 并 `Label` 元素绑定到相同的源中的属性 `HslColorViewModel` 对象。唯一的区别在于 `Label` 已 `StringFormat` 属性来显示每个 `Slider` 值。



您可能想知道如何可以在传统两位数字的十六进制格式中显示 RGB（红色、绿色、蓝色）值。这些整数值不从直

接可用 `Color` 结构。一种解决方案是计算颜色组件是在 ViewModel 中的整数值并将其作为属性公开。你可以然后格式化它们使用 `X2` 格式规范。

另一种方法是更多常规：你可以编写*绑定值转换器*更高版本的文章中所述**绑定值转换器**。

下一篇文章中，但是，探讨**绑定路径**更详细介绍，并且显示了如何使用它来引用子属性和集合中的项。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 绑定路径

2018/10/27 • • Edit Online

在所有以前的数据绑定示例中，`Path` 的属性 `Binding` 类 (或 `Path` 属性 `Binding` 标记扩展) 已设置向单个属性。实际上可能会设置 `Path` 到*子属性*(的属性的属性)，或者与集合的成员。

例如，假设您的页面包含 `TimePicker`：

```
<TimePicker x:Name="timePicker">
```

`Time` 的属性 `TimePicker` 属于类型 `TimeSpan`，但可能是你想要创建数据绑定引用 `TotalSeconds` 属性的 `TimeSpan` 值。下面是数据绑定：

```
{Binding Source={x:Reference timePicker},
         Path=Time.TotalSeconds}
```

`Time` 属性属于类型 `TimeSpan`，其中包含 `TotalSeconds` 属性。`Time` 和 `TotalSeconds` 属性是只需连接的周期。中的项 `Path` 字符串始终引用属性而不属于这些属性的类型。

在显示的示例和其他几种路径变体页：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:globe="clr-namespace:System.Globalization;assembly=mscorlib"
             x:Class="DataBindingDemos.PathVariationsPage"
             Title="Path Variations"
             x:Name="page">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="HorizontalTextAlignment" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Margin="10, 0">
        <TimePicker x:Name="timePicker" />

        <Label Text="{Binding Source={x:Reference timePicker},
                              Path=Time.TotalSeconds,
                              StringFormat='{0} total seconds'}" />

        <Label Text="{Binding Source={x:Reference page},
                              Path=Content.Children.Count,
                              StringFormat='There are {0} children in this StackLayout'}" />

        <Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
                              Path=DateTimeFormat.DayNames[3],
                              StringFormat='The middle day of the week is {0}'}" />

        <Label>
            <Label.Text>
                <Binding Path="DateTimeFormat.DayNames[3]"
                         StringFormat="The middle day of the week in France is {0}">
                    <Binding.Source>
                        <globe:CultureInfo>
                            <x:Arguments>
                                <x:String>fr-FR</x:String>
                            </x:Arguments>
                        </globe:CultureInfo>
                    </Binding.Source>
                </Binding>
            </Label.Text>
        </Label>

        <Label Text="{Binding Source={x:Reference page},
                              Path=Content.Children[1].Text.Length,
                              StringFormat='The second Label has {0} characters'}" />
    </StackLayout>
</ContentPage>
```

在第二个 `Label`，绑定源是页面本身。`Content` 属性属于类型 `StackLayout`，其中包含 `Children` 类型的属性 `IList<View>`，其中包含 `Count` 属性，指示的子级的个数。

使用索引器的路径

在第三个绑定 `Label` 中路径变体的引用 `CultureInfo` 类 `System.Globalization` 命名空间：

```
<Label Text="{Binding Source={x:Static globe:CultureInfo.CurrentCulture},
                      Path=DateTimeFormat.DayNames[3],
                      StringFormat='The middle day of the week is {0}'}" />
```

源设置为静态 `CultureInfo.CurrentCulture` 属性，它是类型的对象 `CultureInfo` 。类定义一个名为属性 `DateTimeFormat` 类型的 `DateTimeFormatInfo` ，其中包含 `DayNames` 集合。索引选择第四个项。

第四个 `Label` 执行类似但区域性与法国相关联。 `Source` 绑定的属性设置为 `CultureInfo` 对象和构造函数：

```
<Label>
    <Label.Text>
        <Binding Path="DateTimeFormat.DayNames[3]"
                 StringFormat="The middle day of the week in France is {0}">
            <Binding.Source>
                <globe:CultureInfo>
                    <x:Arguments>
                        <x:String>fr-FR</x:String>
                    </x:Arguments>
                </globe:CultureInfo>
            </Binding.Source>
        </Binding>
    </Label.Text>
</Label>
```

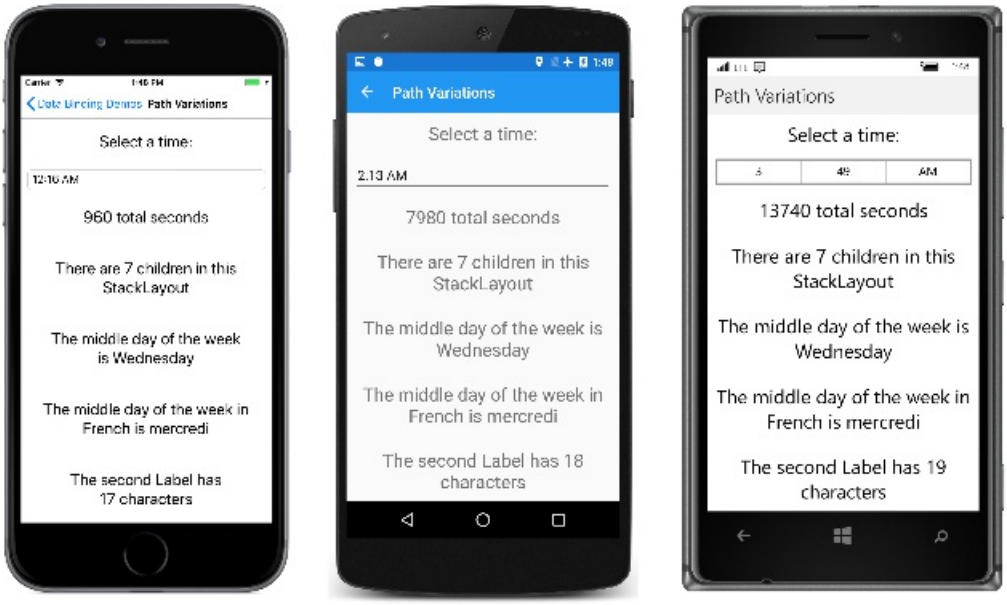
请参阅传递构造函数参数有关在 XAML 中指定构造函数自变量的详细信息。

最后，最后一个示例是类似于第二个，不同之处在于它所引用的一个子级的 `StackLayout` ：

```
<Label Text="{Binding Source={x:Reference page},
                      Path=Content.Children[1].Text.Length,
                      StringFormat='The first Label has {0} characters'}" />
```

该子级是 `Label` ，其中包含 `Text` 类型的属性 `String` ，其中包含 `Length` 属性。第一个 `Label` 报表 `TimeSpan` 集中 `TimePicker` ，因此该文本更改时，最终 `Label` 也将发生更改。

下面是在所有三个平台上运行的程序：



调试复杂的路径

复杂路径定义可能很难构造：您需要知道每个子属性的类型或要正确地添加下一个子属性的集合中的项的类型，但这些类型本身并不显示在路径中。一个好方法是以增量方式生成路径和查看的中间结果。对于最后一个示例，可以开始否 `Path` 根本的定义：

```
<Label Text="{Binding Source={x:Reference page},
                      StringFormat='{0}'}" />
```

显示绑定源的类型或 `DataBindingDemos.PathVariationsPage`。您知道 `PathVariationsPage` 派生自 `ContentPage`，因此它有 `Content` 属性：

```
<Label Text="{Binding Source={x:Reference page},
                      Path=Content,
                      StringFormat='{0}'}" />
```

类型 `Content` 属性现在显示为 `Xamarin.Forms.StackLayout`。添加 `Children` 属性设置为 `Path` 且类型为 `Xamarin.Forms.ElementCollection'1[Xamarin.Forms.View]`，Xamarin.Forms 中，但显然是集合类型的内部的类。将索引添加到该且类型为 `Xamarin.Forms.Label`。继续以这种方式。

在 Xamarin.Forms 处理绑定路径，它将安装 `PropertyChanged` 实现的路径中的任何对象上的处理程序 `INotifyPropertyChanged` 接口。最后一个绑定例如，在第一个更改做出反应 `Label` 因为 `Text` 属性更改。

如果绑定路径中的属性不实现 `INotifyPropertyChanged`，对该属性的任何更改将被忽略。某些更改可能完全使指针无效的绑定路径，因此仅当属性和子属性的字符串永远不会变为无效时，应使用此方法。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 绑定值转换器

2018/7/13 • • Edit Online

数据绑定到目标属性，并在某些情况下从目标属性到源属性，通常将数据传输源属性中。当源和目标属性都是相同的类型，或一个类型可以转换为另一种类型的隐式转换通过时，此传输非常简单。如果不是这种情况，类型转换必须执行的位置。

在中**字符串格式设置**文章中，你已了解如何使用 `StringFormat` 数据绑定来转换为字符串的任何类型的属性。对于其他类型的转换，需要编写一些专门的代码中实现的类 `IValueConverter` 接口。(通用 Windows 平台包含一个名为的相似类 `IValueConverter` 中 `Windows.UI.Xaml.Data` 命名空间，但这 `IValueConverter` 处于 `Xamarin.Forms` 命名空间。)类实现 `IValueConverter` 称为*值转换器*，但它们通常也称为*绑定转换器*或*绑定值转换器*。

IValueConverter 接口

假设你想要定义其中源属性是类型的数据绑定 `int` 但目标属性为 `bool` 。你想要生成此数据绑定 `false` 值等于 0，整数源时和 `true` 否则为。

可以使用实现的类来执行此操作 `IValueConverter` 接口：

```
public class IntToBoolConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)value != 0;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (bool)value ? 1 : 0;
    }
}
```

设置到此类的实例 `Converter` 的属性 `Binding` 类或设置为 `Converter` 属性 `Binding` 标记扩展。此类将成为数据绑定的一部分。

`Convert` 时数据将从源系统移到目标中调用方法 `OneWay` 或 `TwoWay` 绑定。 `value` 参数是对象或数据绑定源中的值。该方法必须返回的数据绑定目标类型的值。如下所示的强制转换的方法 `value` 参数 `int` ，然后将它与 0 进行比较 `bool` 返回值。

`ConvertBack` 时数据从目标移动到的源中调用方法 `TwoWay` 或 `OneWayToSource` 绑定。 `ConvertBack` 执行相反的转换：它假定 `value` 参数是 `bool` 目标，并将其转换为 `int` 返回源值。

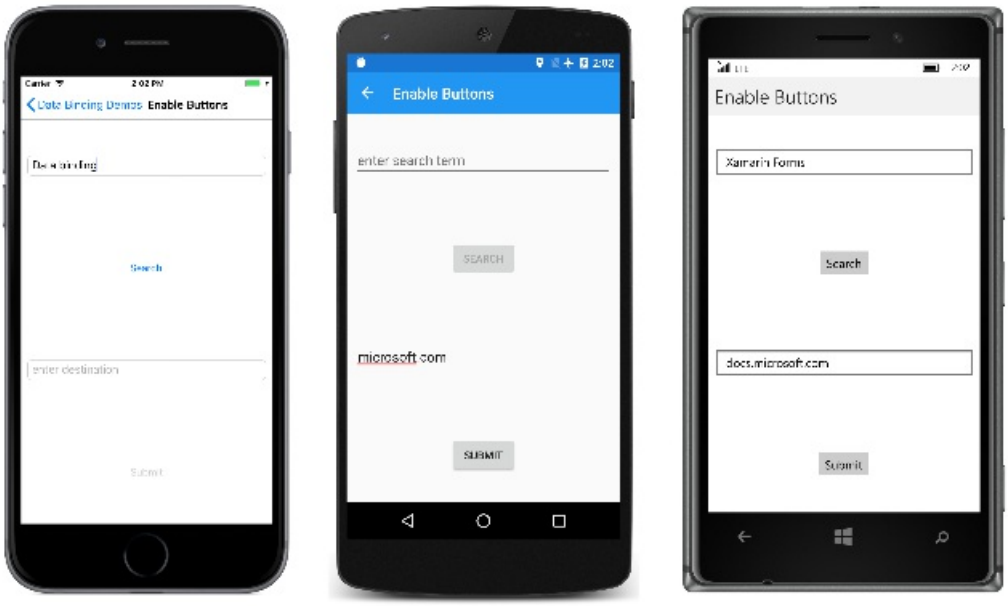
如果数据绑定还包括 `StringFormat` 设置，值转换器之前调用结果格式化为字符串。

启用按钮页面**数据绑定演示**示例演示如何在数据绑定中使用此值转换器。 `IntToBoolConverter` 在页面的资源字典中实例化。它然后使用引用 `StaticResource` 标记扩展来设置 `Converter` 中两个数据绑定属性。它是共享的页上的多个数据绑定中的数据转换器非常常见：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.EnableButtonsPage"
             Title="Enable Buttons">
    <ContentPage.Resources>
        <ResourceDictionary>
            <local:IntToBoolConverter x:Key="intToBool" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Padding="10, 0">
        <Entry x:Name="entry1"
               Text=""
               Placeholder="enter search term"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Search"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                IsEnabled="{Binding Source={x:Reference entry1},
                                    Path=Text.Length,
                                    Converter={StaticResource intToBool}}" />

        <Entry x:Name="entry2"
               Text=""
               Placeholder="enter destination"
               VerticalOptions="CenterAndExpand" />

        <Button Text="Submit"
                HorizontalOptions="Center"
                VerticalOptions="CenterAndExpand"
                IsEnabled="{Binding Source={x:Reference entry2},
                                    Path=Text.Length,
                                    Converter={StaticResource intToBool}}" />
    </StackLayout>
</ContentPage>
```

如果你的应用程序的多个页中使用值转换器，则您可以在实例化它中的资源字典**App.xaml**文件。

启用按钮页说明时需要一种常见 `Button` 执行一个操作基于文本的用户键入到 `Entry` 视图。如果不具有已键入 `Entry`，则 `Button` 应禁用。每个 `Button` 包含数据绑定在其 `IsEnabled` 属性。数据绑定源是 `Length` 的属性 `Text` 的相应属性 `Entry`。如果该 `Length` 属性不是 0，则返回值转换器 `true` 和 `Button` 已启用：



请注意，`Text`中每个属性`Entry`初始化为空字符串。`Text`属性是`null`默认情况下，数据绑定不会在这种情况下。

某些值转换器编写专门为特定应用程序，而其他人已通用化。如果您知道值转换器将只应用于`OneWay`绑定，则`ConvertBack`方法可以只返回`null`。

`Convert`隐式上面所示方法假定`value`自变量的类型是`int`并且返回值必须是类型`bool`。同样，`ConvertBack`方法假设`value`自变量的类型是`bool`和返回值是`int`。如果这不是这种情况，将发生运行时异常。

您可以编写更通用化并接受多个不同类型的数据值转换器。`Convert`并`ConvertBack`方法可使用`as`或`is`运算符`value`参数，也可以调用`GetType`上该参数，以确定其类型，然后再执行一些适当的。每个方法的返回值的预期的类型由`targetType`参数。有时，使用数据绑定的不同的目标类型; 使用值转换器可以使用值转换器`targetType`参数来执行转换为正确的类型。

如果正在执行的转换为不同的区域性不同，请使用`culture`实现此目的的参数。`parameter`自变量`Convert`和`ConvertBack`本文稍后介绍。

绑定转换器属性

值转换器类可以具有属性和泛型参数。此特定值转换器转换`bool`从源类型的对象到`T`目标：

```
public class BoolToObjectConverter<T> : IValueConverter
{
    public T TrueObject { set; get; }

    public T FalseObject { set; get; }

    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (bool)value ? TrueObject : FalseObject;
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return ((T)value).Equals(TrueObject);
    }
}
```

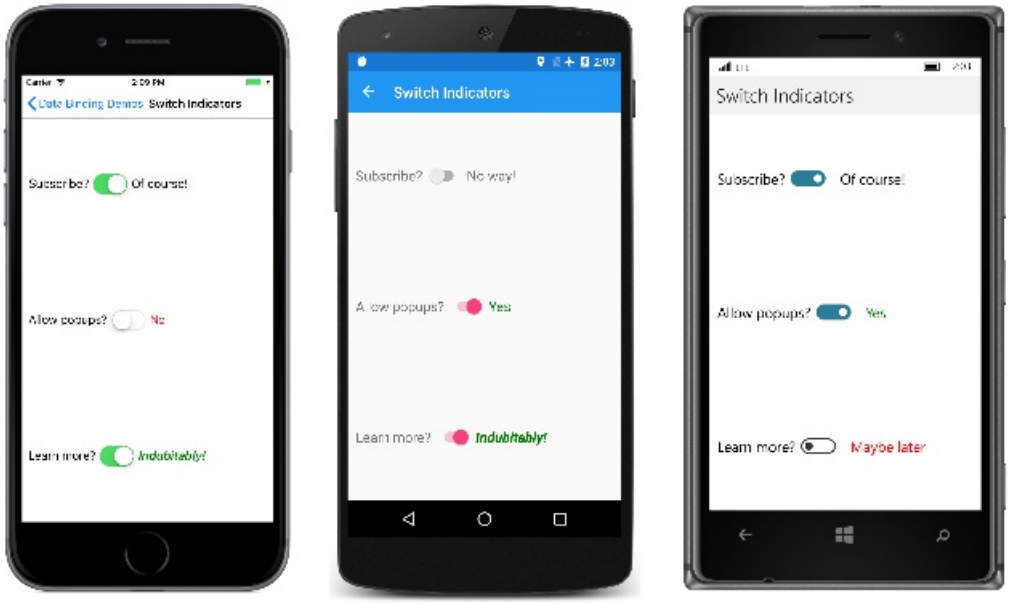
交换机指示器页说明如何使用它来显示的值`Switch`视图。虽然通常会作为资源字典中的资源实例化的值转换器，但此页将演示一种替代方法：之间实例化每个值转换器`Binding.Converter`属性元素标记。`x:TypeArguments`指示泛型自变量，并`TrueObject`和`FalseObject`均设置为该类型的对象：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.SwitchIndicatorsPage"
             Title="Switch Indicators">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="FontSize" Value="18" />
                <Setter Property="VerticalOptions" Value="Center" />
            </Style>

            <Style TargetType="Switch">
                <Setter Property="VerticalOptions" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Padding="10, 0">
        <StackLayout Orientation="Horizontal"
                     VerticalOptions="CenterAndExpand">
```

```
                VerticalOptions="CenterAndExpand" >
    <Label Text="Subscribe?" />
    <Switch x:Name="switch1" />
    <Label>
        <Label.Text>
            <Binding Source="{x:Reference switch1}"
                     Path="IsToggled">
                <Binding.Converter>
                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                 TrueObject="Of course!"
                                                 FalseObject="No way!" />
                </Binding.Converter>
            </Binding>
        </Label.Text>
    </Label>
</StackLayout>

<StackLayout Orientation="Horizontal"
             VerticalOptions="CenterAndExpand">
    <Label Text="Allow popups?" />
    <Switch x:Name="switch2" />
    <Label>
        <Label.Text>
            <Binding Source="{x:Reference switch2}"
                     Path="IsToggled">
                <Binding.Converter>
                    <local:BoolToObjectConverter x:TypeArguments="x:String"
                                                 TrueObject="Yes"
                                                 FalseObject="No" />
                </Binding.Converter>
            </Binding>
        </Label.Text>
        <Label.TextColor>
            <Binding Source="{x:Reference switch2}"
                     Path="IsToggled">
                <Binding.Converter>
                    <local:BoolToObjectConverter x:TypeArguments="Color"
                                                 TrueObject="Green"
                                                 FalseObject="Red" />
                </Binding.Converter>
            </Binding>
        </Label.TextColor>
    </Label>
</StackLayout>

<StackLayout Orientation="Horizontal"
             VerticalOptions="CenterAndExpand">
    <Label Text="Learn more?" />
    <Switch x:Name="switch3" />
    <Label FontSize="18"
           VerticalOptions="Center">
        <Label.Style>
            <Binding Source="{x:Reference switch3}"
                     Path="IsToggled">
                <Binding.Converter>
                    <local:BoolToObjectConverter x:TypeArguments="Style">
                        <local:BoolToObjectConverter.TrueObject>
                            <Style TargetType="Label">
                                <Setter Property="Text" Value="Indubitably!" />
                                <Setter Property="FontAttributes" Value="Italic, Bold" />
                                <Setter Property="TextColor" Value="Green" />
                            </Style>
                        </local:BoolToObjectConverter.TrueObject>

                        <local:BoolToObjectConverter.FalseObject>
                            <Style TargetType="Label">
                                <Setter Property="Text" Value="Maybe later" />
                                <Setter Property="FontAttributes" Value="None" />
                                <Setter Property="TextColor" Value="Red" />
```

```
                                    </Style>
                                </local:BoolToObjectConverter.FalseObject>
                            </local:BoolToObjectConverter>
                        </Binding.Converter>
                    </Binding>
                </Label.Style>
            </Label>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

在过去三个 `Switch` 并 `Label` 对，泛型自变量设置为 `Style`，和整个 `Style` 对象提供的值 `TrueObject` 和 `FalseObject`。这些重写的隐式样式 `Label` 在资源字典中设置，因此该样式中的属性显式分配给 `Label`。切换 `Switch` 会导致相应 `Label` 以反映此更改：



它也是可以使用 `Triggers` 基于其他视图的用户界面中实现类似的更改。

绑定转换器参数

`Binding` 类定义 `ConverterParameter` 属性，并且 `Binding` 标记扩展还定义 `ConverterParameter` 属性。如果设置此属性，则该值将传递到 `Convert` 并 `ConvertBack` 方法，例如 `parameter` 参数。即使在多个数据绑定，间共享的值转换器实例 `ConverterParameter` 可以不同执行稍有不同的转换。

使用 `ConverterParameter` 演示了颜色选择程序。在这种情况下，`RgbColorViewModel` 有三个属性的类型 `double` 名为 `Red`，`Green`，并 `Blue` 它用来构造 `Color` 值：

```
public class RgbColorViewModel : INotifyPropertyChanged
{
    Color color;
    string name;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Red
    {
        set
        {
            if (color.R != value)
            {
                Color = new Color(value, color.G, color.B);
            }
        }
        get
```

```
        {
            return color.R;
        }
    }

    public double Green
    {
        set
        {
            if (color.G != value)
            {
                Color = new Color(color.R, value, color.B);
            }
        }
        get
        {
            return color.G;
        }
    }

    public double Blue
    {
        set
        {
            if (color.B != value)
            {
                Color = new Color(color.R, color.G, value);
            }
        }
        get
        {
            return color.B;
        }
    }

    public Color Color
    {
        set
        {
            if (color != value)
            {
                color = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Red"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Green"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Blue"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));

                Name = NamedColor.GetNearestColorName(color);
            }
        }
        get
        {
            return color;
        }
    }

    public string Name
    {
        private set
        {
            if (name != value)
            {
                name = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Name"));
            }
        }
        get
        {
            return name;
```

```
        }
    }
}
```

`Red`, `Green`, 和 `Blue` 属性范围介于 0 和 1 之间。但是，你可能倾向，组件显示为两位数字的十六进制值。

若要在 XAML 中显示为十六进制值，它们必须乘以 255，转换为整数，然后使用"X2"的规范格式中 `StringFormat` 属性。值转换器可以处理的前两个任务（乘以 255 和将转换为整数）。若要使值转换器以尽可能通用，乘法因子可以指定与 `ConverterParameter` 属性，这意味着它将进入 `Convert` 并 `ConvertBack` 的方法作为 `parameter` 参数：

```
public class DoubleToIntConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)Math.Round((double)value * GetParameter(parameter));
    }

    public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
    {
        return (int)value / GetParameter(parameter);
    }

    double GetParameter(object parameter)
    {
        if (parameter is double)
            return (double)parameter;

        else if (parameter is int)
            return (int)parameter;

        else if (parameter is string)
            return double.Parse((string)parameter);

        return 1;
    }
}
```

`Convert` 将从转换 `double` 到 `int` 时乘以 `parameter` 值; `ConvertBack` 除以整数 `value` 自变量 `parameter`，并返回 `double` 结果。（在程序中如下所示，值转换器仅用于字符串格式设置，因此 `ConvertBack` 不使用。）

类型 `parameter` 参数是可能不同，具体取决于是否在代码或 XAML 中定义的数据绑定。如果 `ConverterParameter` 属性的 `Binding` 设置在代码中，很可能设置为数字值：

```
binding.ConverterParameter = 255;
```

`ConverterParameter` 属性属于类型 `Object`，因此，C# 编译器将文字 255 解释为一个整数，并且将属性设置为该值。

在 XAML，但是，`ConverterParameter` 很可能设置如下：

```
<Label Text="{Binding Red,
                      Converter={StaticResource doubleToInt},
                      ConverterParameter=255,
                      StringFormat='Red = {0:X2}'}" />
```

255 看起来像一个数字，但，因为 `ConverterParameter` 属于类型 `Object`，XAML 分析器将 255 视为字符串。

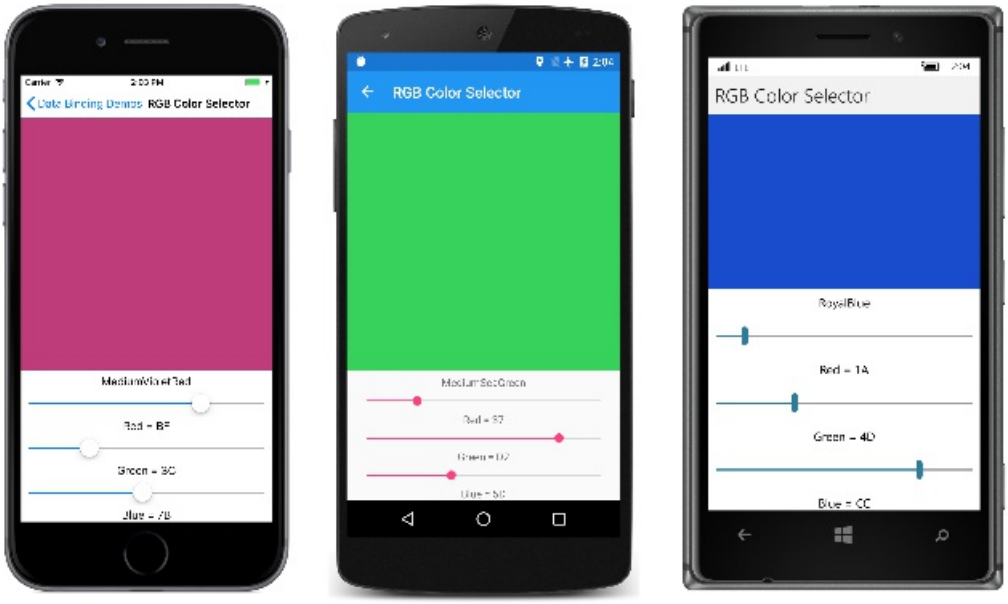
为此，如上所示的值转换器包含一个单独 `GetParameter` 处理的情况下方法 `parameter` 类型的正在 `double`， `int`，或 `string`。

RGB 颜色选择器页上实例化 `DoubleToIntConverter` 在以下两个隐式样式的定义及其资源字典中：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.RgbColorSelectorPage"
             Title="RGB Color Selector">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Slider">
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>

            <local:DoubleToIntConverter x:Key="doubleToInt" />
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout>
        <StackLayout.BindingContext>
            <local:RgbColorViewModel Color="Gray" />
        </StackLayout.BindingContext>

        <BoxView Color="{Binding Color}"
                 VerticalOptions="FillAndExpand" />

        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />

            <Slider Value="{Binding Red}" />
            <Label Text="{Binding Red,
                                 Converter={StaticResource doubleToInt},
                                 ConverterParameter=255,
                                 StringFormat='Red = {0:X2}'}" />

            <Slider Value="{Binding Green}" />
            <Label Text="{Binding Green,
                                 Converter={StaticResource doubleToInt},
                                 ConverterParameter=255,
                                 StringFormat='Green = {0:X2}'}" />

            <Slider Value="{Binding Blue}" />
            <Label>
                <Label.Text>
                    <Binding Path="Blue"
                             StringFormat="Blue = {0:X2}"
                             Converter="{StaticResource doubleToInt}">
                        <Binding.ConverterParameter>
                            <x:Double>255</x:Double>
                        </Binding.ConverterParameter>
                    </Binding>
                </Label.Text>
            </Label>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

值 `Red` 并 `Green` 属性将显示与 `Binding` 标记扩展。 `Blue` 属性，但是，实例化 `Binding` 类，以演示如何使用显式 `double` 可以将值设置为 `ConverterParameter` 属性。

下面是结果：



相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 绑定回退

2018/10/26 • • Edit Online

有时数据绑定失败，因为无法解析，绑定源，或者因为绑定成功，但返回 `null` 值。虽然可以使用值转换器或其他额外的代码处理这些情况下，数据绑定可实现更稳健通过定义要使用在绑定过程失败时的回退值。这可以通过定义来实现 `FallbackValue` 并 `TargetNullValue` 绑定表达式中的属性。因为这些属性位于 `BindingBase` 类，它们可以使用绑定，已编译的绑定，以及使用 `Binding` 标记扩展。

> **NOTE**
> 利用 `FallbackValue` 并 `TargetNullValue` 绑定表达式中的属性是可选的。

定义一个回退值

`FallbackValue` 属性，可将使用定义的回退值时绑定*源*无法解析。设置此属性的常见方案是绑定到不同类型的绑定的集合中的所有对象可能不存在的源属性时。

MonkeyDetail页说明了设置 `FallbackValue` 属性：

```
<Label Text="{Binding Population, FallbackValue='Population size unknown'}"
       ... />
```

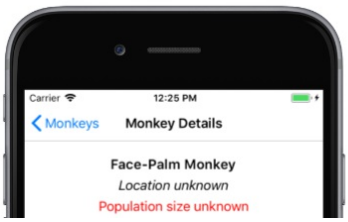
上的绑定 `Label` 定义 `FallbackValue` 绑定源无法解析，则将在目标系统设置的值。因此，定义的值 `FallbackValue` 将显示属性，如果 `Population` 属性不存在对绑定对象。请注意，此处 `FallbackValue` 由单引号（撇号）字符分隔属性值。

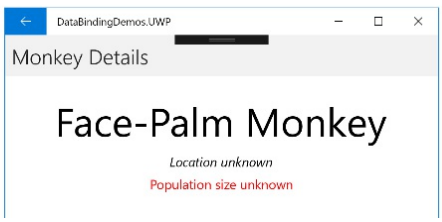
而不是定义 `FallbackValue` 内联属性值，我们建议将它们定义为中的资源 `ResourceDictionary` 。此方法的优点是，此类值在单个位置中，定义一次并且可以进行更轻松地本地化。然后可以使用检索资源 `StaticResource` 标记扩展：

```
<Label Text="{Binding Population, FallbackValue={StaticResource populationUnknown}}"
       ... />
```

> **NOTE**
> 不能设置 `FallbackValue` 使用绑定表达式的属性。

下面是在所有三个平台上运行的程序：





当 `FallbackValue` 属性未设置在绑定表达式和绑定路径或路径的一部分进行解析， `BindableProperty.DefaultValue` 在目标上设置。但是，当 `FallbackValue` 属性设置了绑定路径或路径的一部分进行解析，值 `FallbackValue` value 属性设置在目标系统上。因此，在**MonkeyDetail**页面 `Label` 显示"未知的总体大小"，因为缺少所需的绑定的对象

`Population` 属性。

> **IMPORTANT**
> 绑定表达式中不执行已定义的值转换器时 `FallbackValue` 属性设置。

定义 null 替换值

`TargetNullValue` 属性，可替换值以定义用于将时绑定*源*得到解决，但值 `null` 。设置此属性的常见方案是绑定到可能的源属性时 `null` 绑定集合中。

猴页说明了设置 `TargetNullValue` 属性：

```
<ListView ItemsSource="{Binding Monkeys}"
          ...>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <Grid>
                    ...
                    <Image Source="{Binding ImageUrl,
TargetNullValue='https://upload.wikimedia.org/wikipedia/commons/2/20/Point_d_interrogation.jpg'}"
                           ... />
                    ...
                    <Label Text="{Binding Location, TargetNullValue='Location unknown'}"
                           ... />
                </Grid>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

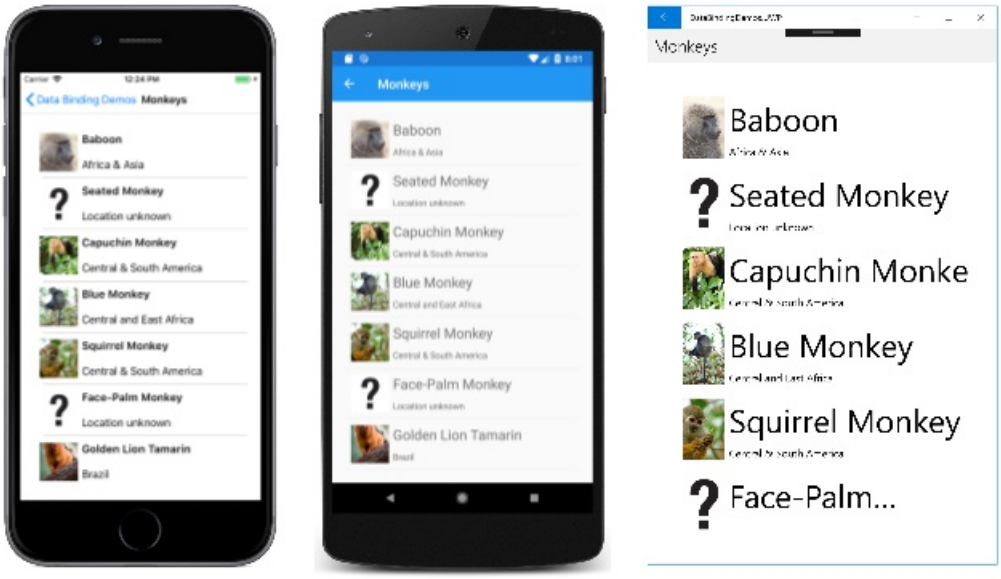
上的绑定 `Image` 并 `Label` 都定义 `TargetNullValue` 值的绑定路径返回的情况下应用 `null` . 因此，定义的值 `TargetNullValue` 属性将显示集合中的任何对象的位置 `ImageUrl` 和 `Location` 未定义属性。请注意，此处 `TargetNullValue` 由单引号（撇号）字符分隔属性值。

而不是定义 `TargetNullValue` 内联属性值，我们建议将它们定义为中的资源 `ResourceDictionary` 。此方法的优点是，此类值在单个位置中，定义一次并且可以进行更轻松地本地化。然后可以使用检索资源 `StaticResource` 标记扩展：

```
<Image Source="{Binding ImageUrl, TargetNullValue={StaticResource fallbackImageUrl}}"
       ... />
<Label Text="{Binding Location, TargetNullValue={StaticResource locationUnknown}}"
       ... />
```

> **NOTE**
> 不能设置 `TargetNullValue` 使用绑定表达式的属性。

下面是在所有三个平台上运行的程序：



时`TargetNullValue`属性未设置在绑定表达式中，源值`null`将被转换，如果定义了值转换器，如果格式化`StringFormat`定义，然后将结果设置在目标系统上。但是，当`TargetNullValue`属性设置，源值`null`如果定义了值转换器，并且如果它仍是将转换`null`完成转换的值后`TargetNullValue`属性设置在目标系统上。

> **IMPORTANT**
> 绑定表达式中不应用格式设置字符串时`TargetNullValue`属性设置。

相关链接

- 数据绑定演示（示例）

Xamarin.Forms 命令界面

2018/11/1 • • Edit Online

在模型-视图-视图模型 (MVVM) 体系结构中，定义数据绑定的 ViewModel，这通常是派生的类中属性之间 `INotifyPropertyChanged`，并在视图中，这通常是 XAML 文件的属性。有时应用程序能够通过要求用户若要启动命令会影响在 ViewModel 中的某些内容超越这些属性绑定的需求。这些命令通常通过单击按钮来发出信号或手指点击，以及传统上在代码隐藏文件中的处理程序中处理这些请求 `Clicked` 的事件 `Button` 或 `Tapped` 事件的 `TapGestureRecognizer`。

命令接口提供对 MVVM 体系结构更适合一种方法来实现命令。ViewModel 本身可以包含命令，如在视图中的特定活动的反应中执行的方法 `Button` 单击。这些命令之间定义数据绑定和 `Button`。

若要允许之间的数据绑定 `Button` 和 ViewModel，`Button` 定义两个属性：

- `Command` 类型 `System.Windows.Input.ICommand`
- `CommandParameter` 类型 `Object`

若要使用命令接口，您可以定义数据绑定面向 `Command` 的属性 `Button` 其中源是类型的 ViewModel 中的属性 `ICommand`。ViewModel 包含与该代码 `ICommand` 时单击该按钮时执行的属性。可以设置 `CommandParameter` 到同一个绑定到任意数据来区分多个按钮，如果所有 `ICommand` ViewModel 中的属性。

`Command` 和 `CommandParameter` 属性也由以下类定义：

- `MenuItem` 因此，`ToolbarItem`，又派生自 `MenuItem`
- `TextCell` 因此，`ImageCell`，又派生自 `TextCell`
- `TapGestureRecognizer`

`SearchBar` 定义 `SearchCommand` 类型的属性 `ICommand` 和一个 `SearchCommandParameter` 属性。`RefreshCommand` 的属性 `ListView` 类型的也是 `ICommand`。

在中的 ViewModel 不依赖于在视图中的特定用户界面对象的方式处理所有这些命令。

ICommand 接口

`System.Windows.Input.ICommand` 接口不是 Xamarin.Forms 的一部分。而定义在System.Windows.Input命名空间，它包括两个方法和一个事件：

```
public interface ICommand
{
    public void Execute (Object parameter);

    public bool CanExecute (Object parameter);

    public event EventHandler CanExecuteChanged;
}
```

若要使用命令接口，将 ViewModel 包含类型的属性 `ICommand`：

```
public ICommand MyCommand { private set; get; }
```

ViewModel 还必须引用实现的类 `ICommand` 接口。此类将稍后所述。在视图中，`Command` 属性的 `Button` 绑定到该属性：

```
<Button Text="Execute command"
        Command="{Binding MyCommand}" />
```

当用户按 `Button`，则 `Button` 调用 `Execute` 中的方法 `ICommand` 对象绑定到其 `Command` 属性。是接口的最简单的命令的一部分。

`CanExecute` 方法是更复杂。绑定上的第一个定义当 `Command` 的属性 `Button`，并以某种方式更改数据绑定时 `Button` 调用 `CanExecute` 中的方法 `ICommand` 对象。如果 `CanExecute` 将返回 `false`，则 `Button` 禁用其自身。这表示特定命令目前不可用或无效。

`Button` 还会将一个处理程序附加上 `CanExecuteChanged` 事件的 `ICommand`。从在 ViewModel 中激发事件。当触发该事件时，`Button` 调用 `CanExecute` 试。`Button` 情况下启用自身 `CanExecute` 返回 `true` 和禁用它自己 `CanExecute` 返回 `false`。

> **IMPORTANT**
> 不要使用 `IsEnabled` 属性的 `Button` 如果使用的命令接口。

命令类

当将 ViewModel 定义类型的属性 `ICommand`，ViewModel 也必须包含或引用实现的类 `ICommand` 接口。此类必须包含或引用 `Execute` 并 `CanExecute` 方法和激发 `CanExecuteChanged` 事件每当 `CanExecute` 方法可能返回不同的值。

可以自己，编写此类也可以使用其他人编写了一个类。因为 `ICommand` 是一部分的 Microsoft Windows，它具有已使用多年与 Windows MVVM 应用程序。使用的 Windows 类，实现 `ICommand` 使你可共享 Windows 应用程序和 Xamarin.Forms 应用程序之间将 Viewmodel。

如果共享 Viewmodel 之间 Windows 和 Xamarin.Forms 不是关键因素，则可以使用 `Command` 或 `Command<T>` 类包含在 Xamarin.Forms 中实现 `ICommand` 接口。这些类允许您指定的正文 `Execute` 和 `CanExecute` 类构造函数中的方法。使用 `Command<T>` 使用时 `CommandParameter` 属性来区分多个视图绑定到同一 `ICommand` 属性，且更简单 `Command` 类不需要时。

基本命令

人的项页面**数据绑定演示**程序演示了一些简单的命令在 ViewModel 中实现。

`PersonViewModel` 定义名为三个属性 `Name`，`Age`，和 `Skills` 定义一个人。此类可进行不包含任何 `ICommand` 属性：

```
public class PersonViewModel : INotifyPropertyChanged
{
    string name;
    double age;
    string skills;

    public event PropertyChangedEventHandler PropertyChanged;

    public string Name
    {
        set { SetProperty(ref name, value); }
        get { return name; }
    }

    public double Age
    {
        set { SetProperty(ref age, value); }
        get { return age; }
    }

    public string Skills
    {
        set { SetProperty(ref skills, value); }
        get { return skills; }
    }

    public override string ToString()
    {
        return Name + ", age " + Age;
    }

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

`PersonCollectionViewModel`所示下面创建新对象的类型`PersonViewModel`，并允许用户以填充数据。为此，该类定义了属性`IsEditing`类型的`bool`并`PersonEdit`类型的`PersonViewModel`。此外，该类定义了三个属性的类型`ICommand`和一个名为属性`Persons`类型的`IList<PersonViewModel>`：

```
public class PersonCollectionViewModel : INotifyPropertyChanged
{
    PersonViewModel personEdit;
    bool isEditing;

    public event PropertyChangedEventHandler PropertyChanged;

    ...

    public bool IsEditing
    {
        private set { SetProperty(ref isEditing, value); }
        get { return isEditing; }
    }

    public PersonViewModel PersonEdit
    {
        set { SetProperty(ref personEdit, value); }
        get { return personEdit; }
    }

    public ICommand NewCommand { private set; get; }

    public ICommand SubmitCommand { private set; get; }

    public ICommand CancelCommand { private set; get; }

    public IList<PersonViewModel> Persons { get; } = new ObservableCollection<PersonViewModel>();

    bool SetProperty<T>(ref T storage, T value, [CallerMemberName] string propertyName = null)
    {
        if (Object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

此缩写的列表不包含类的构造函数，该类型的三个属性 `ICommand` 定义，这将会显示。请注意，更改为三种类型的属性 `ICommand` 并 `Persons` 属性不会导致 `PropertyChanged` 所激发事件。这些属性都设置为，首次创建类时，并且此后不会更改。

之前检查的构造函数 `PersonCollectionViewModel` 类中，让我们看一下 XAML 文件人的项程序。此文件包含 `Grid` 使用其 `BindingContext` 属性设置为 `PersonCollectionViewModel` 。 `Grid` 包含 `Button` 包含文本新建使用其 `Command` 属性绑定到 `NewCommand` ViewModel 中的属性，使用属性的输入窗体绑定到 `IsEditing` 属性，为属性以及 `PersonViewModel` ，并将其绑定到的两个按钮 `SubmitCommand` 和 `CancelCommand` ViewModel 的属性。最后一个 `ListView` 显示人员已经输入的集合：

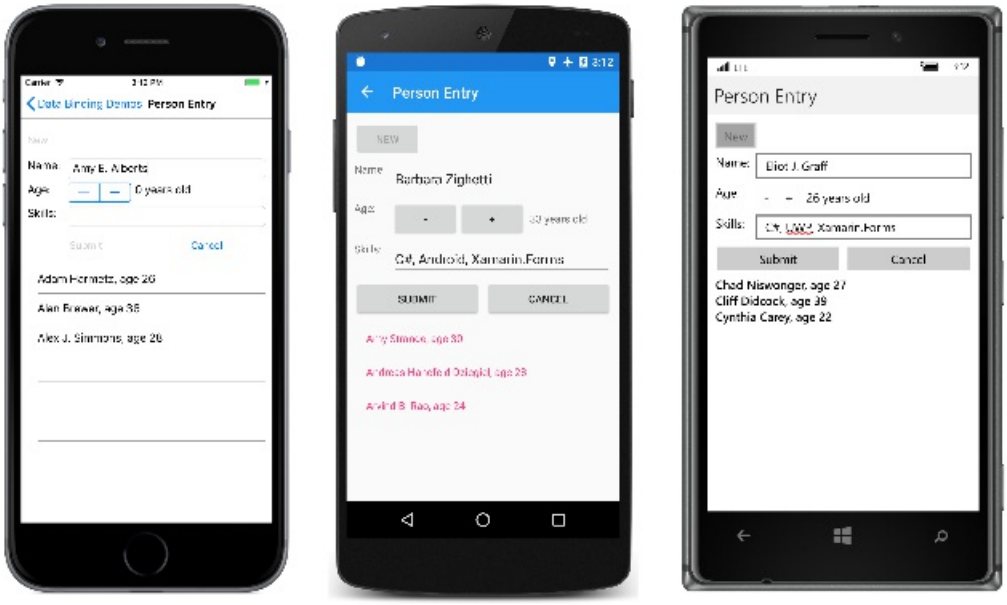
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.PersonEntryPage"
             Title="Person Entry">
    <Grid Margin="10">
        <Grid.BindingContext>
            <local:PersonCollectionViewModel />
        </Grid.BindingContext>
```

```
</ContentPage.BindingContext>

<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<!-- New Button -->
<Button Text="New"
        Grid.Row="0"
        Command="{Binding NewCommand}"
        HorizontalOptions="Start" />

<!-- Entry Form -->
<Grid Grid.Row="1"
      IsEnabled="{Binding IsEditing}">

    <Grid BindingContext="{Binding PersonEdit}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Label Text="Name: " Grid.Row="0" Grid.Column="0" />
        <Entry Text="{Binding Name}"
               Grid.Row="0" Grid.Column="1" />

        <Label Text="Age: " Grid.Row="1" Grid.Column="0" />
        <StackLayout Orientation="Horizontal"
                     Grid.Row="1" Grid.Column="1">
            <Stepper Value="{Binding Age}"
                     Maximum="100" />
            <Label Text="{Binding Age, StringFormat='{0} years old'}"
                   VerticalOptions="Center" />
        </StackLayout>

        <Label Text="Skills: " Grid.Row="2" Grid.Column="0" />
        <Entry Text="{Binding Skills}"
               Grid.Row="2" Grid.Column="1" />

    </Grid>
</Grid>

<!-- Submit and Cancel Buttons -->
<Grid Grid.Row="2">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Button Text="Submit"
            Grid.Column="0"
            Command="{Binding SubmitCommand}"
            VerticalOptions="CenterAndExpand" />

    <Button Text="Cancel"
            Grid.Column="1"
            Command="{Binding CancelCommand}"
            VerticalOptions="CenterAndExpand" />
</Grid>

<!-- List of Persons -->
```

```
        <!-- List of Persons -->
        <ListView Grid.Row="3"
                  ItemsSource="{Binding Persons}" />
    </Grid>
</ContentPage>
```

下面是它的工作原理：用户首次按下**新建**按钮。这使输入窗体，但禁用**新建**按钮。用户然后输入一个名称、年龄和技能。在编辑时，用户可以按**取消**按钮以重新开始。仅当已输入的名称和有效期限**提交**按钮已启用。按这**提交**按钮将此人传输到由显示的集合 `ListView` 。之后可以**取消**或**提交**按下按钮，清除输入窗体并**新建**按钮已再次启用。

输入有效的期限之前，在左侧的 iOS 屏幕显示的布局。Android 和 UWP 屏幕显示**提交**设置年龄后启用按钮：



该程序没有任何工具，以便编辑现有条目，并在导航离开页面时不保存这些条目。

所有的逻辑**新建**，**提交**，并**取消**中处理按钮 `PersonCollectionViewModel` 通过定义 `NewCommand` ， `SubmitCommand` ，和 `CancelCommand` 属性。构造函数 `PersonCollectionViewModel` 将这三个属性设置为类型的对象 `Command` 。

一个构造函数的 `Command` 类，可将类型的自变量传递 `Action` 并 `Func<bool>` 对应于 `Execute` 和 `CanExecute` 方法。它是最简单的方法将这些操作和函数定义为 lambda 函数直接在 `Command` 构造函数。下面是定义 `Command` 对象 `NewCommand` 属性：

```
public class PersonCollectionViewModel : INotifyPropertyChanged
{

    ...

    public PersonCollectionViewModel()
    {
        NewCommand = new Command(
            execute: () =>
            {
                PersonEdit = new PersonViewModel();
                PersonEdit.PropertyChanged += OnPersonEditPropertyChanged;
                IsEditing = true;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return !IsEditing;
            });

        ...

    }

    void OnPersonEditPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        (SubmitCommand as Command).ChangeCanExecute();
    }

    void RefreshCanExecutes()
    {
        (NewCommand as Command).ChangeCanExecute();
        (SubmitCommand as Command).ChangeCanExecute();
        (CancelCommand as Command).ChangeCanExecute();
    }

    ...

}
```

当用户单击**新建按钮**，`execute` 函数传递给 `Command` 执行构造函数。这将创建一个新 `PersonViewModel` 对象，该对象上设置的处理程序 `PropertyChanged` 事件，设置 `IsEditing` 到 `true`，并调用 `RefreshCanExecutes` 后构造函数定义的方法。

除了实现 `ICommand` 接口，`Command` 类还定义了一个名为方法 `ChangeCanExecute` 。应调用到 ViewModel `ChangeCanExecute` 有关 `ICommand` 属性时，发生任何可能会更改的返回值 `CanExecute` 方法。调用 `ChangeCanExecute` 会导致 `Command` 类，以触发 `CanExecuteChanged` 方法。 `Button` 已附加该事件的处理程序并响应通过调用 `CanExecute` 同样，然后启用本身基于该方法的返回值。

时 `execute` 方法 `NewCommand` 调用 `RefreshCanExecutes` ，则 `NewCommand` 属性获取调用 `ChangeCanExecute` ，和 `Button` 调用 `canExecute` 方法，现在返回 `false` 因为 `IsEditing` 属性现 `true` 。

`PropertyChanged` 处理程序的新 `PersonViewModel` 对象调用 `ChangeCanExecute` 方法的 `SubmitCommand` 。下面是该命令属性的实现方式：

```
public class PersonCollectionViewModel : INotifyPropertyChanged
{

    ...

    public PersonCollectionViewModel()
    {

        ...

        SubmitCommand = new Command(
            execute: () =>
            {
                Persons.Add(PersonEdit);
                PersonEdit.PropertyChanged -= OnPersonEditPropertyChanged;
                PersonEdit = null;
                IsEditing = false;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return PersonEdit != null &&
                       PersonEdit.Name != null &&
                       PersonEdit.Name.Length > 1 &&
                       PersonEdit.Age > 0;
            });

        ...
    }

    ...

}
```

`canExecute` 函数 `SubmitCommand` 每次在中更改某个属性调用 `PersonViewModel` 对象正在编辑。它将返回 `true` 仅当 `Name` 属性是至少包含一个字符和 `Age` 大于 0。此时，**提交**按钮将变为启用状态。

`execute` 函数**提交**移除属性更改处理程序从 `PersonViewModel`，将对象添加到 `Persons` 集合，并返回到初始条件的所有内容。

`execute` 函数**取消**按钮执行所有操作的**提交** does 除外按钮添加到集合的对象：

```
public class PersonCollectionViewModel : INotifyPropertyChanged
{

    ...

    public PersonCollectionViewModel()
    {

        ...

        CancelCommand = new Command(
            execute: () =>
            {
                PersonEdit.PropertyChanged -= OnPersonEditPropertyChanged;
                PersonEdit = null;
                IsEditing = false;
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return IsEditing;
            });
    }

    ...

}
```

`canExecute` 方法将返回 `true` 随时 `PersonViewModel` 正在编辑。

这些技术可能是适用于更复杂的方案：中的属性 `PersonCollectionViewModel` 无法绑定到 `SelectedItem` 的属性 `ListView` 用于编辑现有项和一个删除无法添加按钮以删除这些项。

但并不需要定义 `execute` 和 `canExecute` 作为 lambda 函数的方法。您可以将其写入 ViewModel 中为常规的私有方法以及中引用它们 `Command` 构造函数。但是，这种方法 does 往往会导致大量的 ViewModel 中只有一次引用的方法。

使用命令参数

可以很方便的一个或多个按钮（或其他用户界面对象）来共用同一个 `ICommand` ViewModel 中的属性。在这种情况下，使用 `CommandParameter` 属性来区分按钮。

你可以继续使用 `Command` 类，这些共享 `ICommand` 属性。类定义备用构造函数接受 `execute` 并 `canExecute` 带参数的类型的方法 `Object` 。这是如何 `CommandParameter` 传递给这些方法。

但是，使用时 `CommandParameter` ，它是最容易使用泛型 `Command<T>` 类，以指定的对象设置为类型 `CommandParameter` 。 `execute` 和 `canExecute` 您指定的方法具有该类型的参数。

十进制键盘页通过显示如何实现用于输入十进制数字键盘来演示此方法。 `BindingContext` 有关 `Grid` 是 `DecimalKeypadViewModel` 。 `Entry` 此 ViewModel 属性绑定到 `Text` 属性的 `Label` 。所有 `Button` 对象绑定到 ViewModel 中的各种命令： `ClearCommand` ， `BackspaceCommand` ，和 `DigitCommand` ：

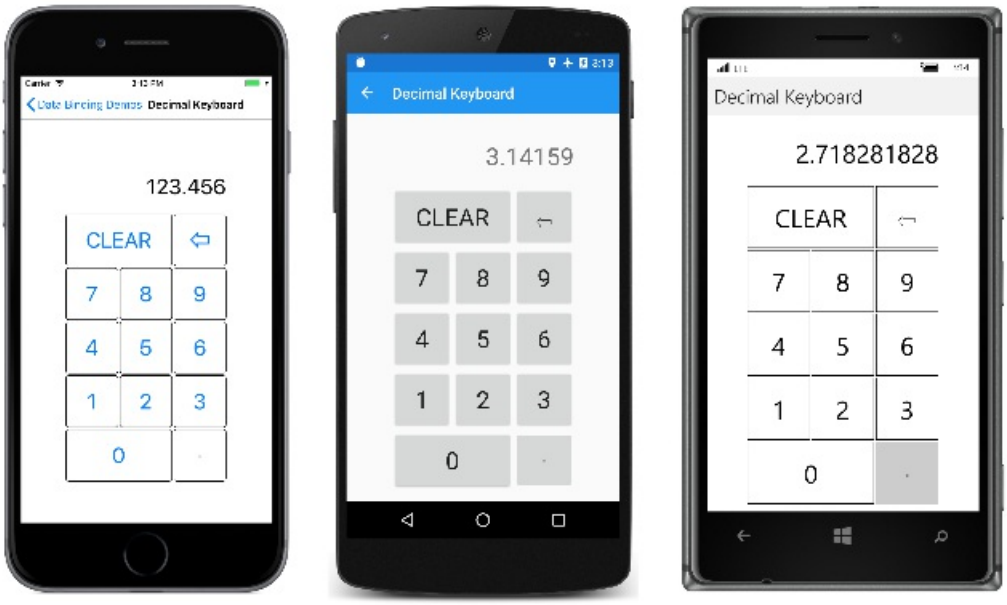
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.DecimalKeypadPage"
             Title="Decimal Keyboard">

    <Grid WidthRequest="240"
          HeightRequest="480"
          ColumnSpacing="2"
          RowSpacing="2"
```

```
  RowSpacing="2"
  HorizontalOptions="Center"
  VerticalOptions="Center">

<Grid.BindingContext>
    <local:DecimalKeypadViewModel />
</Grid.BindingContext>

<Grid.Resources>
    <ResourceDictionary>
        <Style TargetType="Button">
            <Setter Property="FontSize" Value="32" />
            <Setter Property="BorderWidth" Value="1" />
            <Setter Property="BorderColor" Value="Black" />
        </Style>
    </ResourceDictionary>
</Grid.Resources>

<Label Text="{Binding Entry}"
       Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3"
       FontSize="32"
       LineBreakMode="HeadTruncation"
       VerticalTextAlignment="Center"
       HorizontalTextAlignment="End" />

<Button Text="CLEAR"
        Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="2"
        Command="{Binding ClearCommand}" />

<Button Text="&#x21E6;"
        Grid.Row="1" Grid.Column="2"
        Command="{Binding BackspaceCommand}" />

<Button Text="7"
        Grid.Row="2" Grid.Column="0"
        Command="{Binding DigitCommand}"
        CommandParameter="7" />

<Button Text="8"
        Grid.Row="2" Grid.Column="1"
        Command="{Binding DigitCommand}"
        CommandParameter="8" />

<Button Text="9"
        Grid.Row="2" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="9" />

<Button Text="4"
        Grid.Row="3" Grid.Column="0"
        Command="{Binding DigitCommand}"
        CommandParameter="4" />

<Button Text="5"
        Grid.Row="3" Grid.Column="1"
        Command="{Binding DigitCommand}"
        CommandParameter="5" />

<Button Text="6"
        Grid.Row="3" Grid.Column="2"
        Command="{Binding DigitCommand}"
        CommandParameter="6" />

<Button Text="1"
        Grid.Row="4" Grid.Column="0"
        Command="{Binding DigitCommand}"
        CommandParameter="1" />

<Button Text="2"
```

```
                Grid.Row="4" Grid.Column="1"
                Command="{Binding DigitCommand}"
                CommandParameter="2" />

        <Button Text="3"
                Grid.Row="4" Grid.Column="2"
                Command="{Binding DigitCommand}"
                CommandParameter="3" />

        <Button Text="0"
                Grid.Row="5" Grid.Column="0" Grid.ColumnSpan="2"
                Command="{Binding DigitCommand}"
                CommandParameter="0" />

        <Button Text="&#x00B7;"
                Grid.Row="5" Grid.Column="2"
                Command="{Binding DigitCommand}"
                CommandParameter="." />
    </Grid>
</ContentPage>
```

10 位数字和小数点的 11 按钮共享绑定到 `DigitCommand`。 `CommandParameter` 区分这些按钮。将值设置为 `CommandParameter` 通常是与除小数点，它为清晰起见使用中间的点字符显示的按钮显示的文本相同。

下面是该程序的操作：



请注意，所有三个屏幕截图中的十进制点按钮被禁用，因为输入的数字中已包含小数点。

`DecimalKeypadViewModel` 定义 `Entry` 类型的属性 `string`（这是唯一的属性触发 `PropertyChanged` 事件）和类型的三个属性 `ICommand`：

```
public class DecimalKeypadViewModel : INotifyPropertyChanged
{
    string entry = "0";

    public event PropertyChangedEventHandler PropertyChanged;

    ...

    public string Entry
    {
        private set
        {
            if (entry != value)
            {
                entry = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Entry"));
            }
        }
        get
        {
            return entry;
        }
    }

    public ICommand ClearCommand { private set; get; }

    public ICommand BackspaceCommand { private set; get; }

    public ICommand DigitCommand { private set; get; }
}
```

按钮对应于 `ClearCommand` 始终处于启用状态，并只需设置回"0"的条目：

```
public class DecimalKeypadViewModel : INotifyPropertyChanged
{

    ...

    public DecimalKeypadViewModel()
    {
        ClearCommand = new Command(
            execute: () =>
            {
                Entry = "0";
                RefreshCanExecutes();
            });

        ...

    }

    void RefreshCanExecutes()
    {
        ((Command)BackspaceCommand).ChangeCanExecute();
        ((Command)DigitCommand).ChangeCanExecute();
    }

    ...

}
```

由于始终启用按钮，因此不需要指定 `canExecute` 中的参数 `Command` 构造函数。

输入数字和退格的逻辑是有些棘手，因为如果已不输入任何数字，则 `Entry` 属性是字符串"0"。如果用户键入更多

的零，则 `Entry` 仍包含只是一个零。如果用户键入任何其他数字，该数字将替换为零。但是，如果用户键入小数点前任何其他数字，然后 `Entry` 是字符串"0。

退格符仅当项的长度大于 1，或如果启用按钮 `Entry` 不等于字符串"0":

```
public class DecimalKeypadViewModel : INotifyPropertyChanged
{

    ...

    public DecimalKeypadViewModel()
    {

        ...

        BackspaceCommand = new Command(
            execute: () =>
            {
                Entry = Entry.Substring(0, Entry.Length - 1);
                if (Entry == "")
                {
                    Entry = "0";
                }
                RefreshCanExecutes();
            },
            canExecute: () =>
            {
                return Entry.Length > 1 || Entry != "0";
            });

        ...

    }

    ...

}
```

逻辑 `execute` 函数**退格符**按钮可确保 `Entry` 是至少一个"0"的字符串。

`DigitCommand` 属性将绑定到 11 按钮，其中每个标识本身与 `CommandParameter` 属性。 `DigitCommand` 可以将设置为常规的实例 `Command` ，但它的更轻松地使用 `Command<T>` 泛型类。使用 XAML，命令接口时 `CommandParameter` 属性通常是字符串，并且这是泛型参数的类型。 `execute` 并 `canExecute` 函数然后具有类型的自变量 `string` :

```
public class DecimalKeypadViewModel : INotifyPropertyChanged
{

    ...

    public DecimalKeypadViewModel()
    {

        ...

        DigitCommand = new Command<string>(
            execute: (string arg) =>
            {
                Entry += arg;
                if (Entry.StartsWith("0") && !Entry.StartsWith("0."))
                {
                    Entry = Entry.Substring(1);
                }
                RefreshCanExecutes();
            },
            canExecute: (string arg) =>
            {
                return !(arg == "." && Entry.Contains("."));
            });
    }

    ...

}
```

`execute` 方法将追加到字符串自变量 `Entry` 属性。但是，如果结果开头零（但不是零和小数点）然后该初始零必须删除使用 `Substring` 函数。

`canExecute` 方法将返回 `false` 仅当参数为（指示十进制点是否已按下）的小数点和 `Entry` 已包含小数点。

所有 `execute` 方法将调用 `RefreshCanExecutes`，后者随后调用 `ChangeCanExecute` 两个 `DigitCommand` 和 `ClearCommand` 。这可确保小数点和退格符按钮已启用或禁用基于当前的输入的数字序列。

将命令添加到现有视图

如果你想要使用的命令的接口与不支持它的视图，则可以使用 Xamarin.Forms 行为，可将事件转换为命令。这本文所述**可重用 EventToCommandBehavior**。

异步导航菜单的命令

命令是用于实现导航菜单，例如，在方便**数据绑定演示**程序本身。下面是部分**MainPage.xaml**:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.MainPage"
             Title="Data Binding Demos"
             Padding="10">
    <TableView Intent="Menu">
        <TableRoot>
            <TableSection Title="Basic Bindings">

                <TextCell Text="Basic Code Binding"
                          Detail="Define a data-binding in code"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:BasicCodeBindingPage}" />

                <TextCell Text="Basic XAML Binding"
                          Detail="Define a data-binding in XAML"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:BasicXamlBindingPage}" />

                <TextCell Text="Alternative Code Binding"
                          Detail="Define a data-binding in code without a BindingContext"
                          Command="{Binding NavigateCommand}"
                          CommandParameter="{x:Type local:AlternativeCodeBindingPage}" />

                ...

            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>
```

使用与 XAML，发出命令时 `CommandParameter` 属性通常设置为字符串。在这种情况下，但是，XAML 标记扩展使用，以便 `CommandParameter` 属于类型 `System.Type`。

每个 `Command` 属性绑定到一个名为属性 `NavigateCommand`。属性定义在代码隐藏文件中，**MainPage.xaml.cs**：

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(
            async (Type pageType) =>
            {
                Page page = (Page)Activator.CreateInstance(pageType);
                await Navigation.PushAsync(page);
            });

        BindingContext = this;
    }

    public ICommand NavigateCommand { private set; get; }
}
```

构造函数设置 `NavigateCommand` 属性设置为 `execute` 方法实例化 `System.Type` 参数，然后转到它。因为 `PushAsync` 调用需要 `await` 运算符，`execute` 方法必须被标记为异步。这通过实现 `async` 参数列表前面的关键字。

该构造函数还设置 `BindingContext` 到其自身的页，以便绑定引用 `NavigateCommand` 此类中。

此构造函数中的代码的顺序具有重要意义：`InitializeComponent` 调用会导致 XAML 进行分析，但在该时间绑定到

属性命名 `NavigateCommand` 不能解析，因为 `BindingContext` 设置为 `null` 。如果 `BindingContext` 构造函数中设置*之前* `NavigateCommand` 设置，则绑定时，可能会解决 `BindingContext` 设置，但此时， `NavigateCommand` 仍是 `null` 。设置 `NavigateCommand` 后 `BindingContext` 将不在绑定上的没有影响，因为对更改 `NavigateCommand` 不会激发 `PropertyChanged` 事件，并绑定并不知道这 `NavigateCommand` 现在是否有效。

将两者都设置 `NavigateCommand` 并 `BindingContext` （按任何顺序）到在调用前先 `InitializeComponent` 正常工作，因为 XAML 分析器遇到绑定定义时设置的绑定的这两个组件。

数据绑定有时会很棘手，但是，如您所见本系列的文章中，功能强大且用途广泛，而且极大地帮助来分隔从用户界面的基本逻辑来组织你的代码。

相关链接

- 数据绑定演示（示例）
- 数据绑定 Xamarin.Forms 书籍章节

Xamarin.Forms 编译绑定

2018/10/26 • • Edit Online

比经典绑定，从而可以提高 Xamarin.Forms 应用程序中的数据绑定性能更快地解决已编译的绑定。

数据绑定具有两个主要问题：

1. 没有任何编译时验证的绑定表达式。相反，绑定是在运行时解析。因此，任何无效的绑定不是运行时之前或时检测到应用程序不会像预期会显示错误消息。
2. 它们并不经济高效。绑定解析在运行时使用常规用途对象检查（反射），执行此操作的开销会不同平台的。

已编译的绑定通过解析在编译时而不是在运行时绑定表达式来提高 Xamarin.Forms 应用程序中的数据绑定性能。此外，此的绑定表达式的编译时验证可实现更好的开发人员故障排除体验，因为作为生成错误报告无效的绑定。

使用已编译的绑定的过程是：

1. 启用 XAML 编译。有关 XAML 编译的详细信息，请参阅XAML 编译。
2. 设置 `x:DataType` 特性，可以在 `VisualElement` 到的对象类型的 `VisualElement`，及其子级将绑定到。请注意，可以查看层次结构中的任何位置重新定义此属性。

> **NOTE**
> 我们建议设置 `x:DataType` 作为视图层次结构中同一级别的特性 `BindingContext` 设置。

在 XAML 编译时，将报告任何无效的绑定表达式作为生成错误。但是，XAML 编译器仅将报告遇到的第一个无效的绑定表达式的生成错误。未定义任何有效的绑定表达式 `VisualElement` 或其子将已编译，无论 `BindingContext` XAML 或代码中设置。编译绑定表达式生成将在从属性获取一个值的已编译的代码*源*，并将其上设置的属性上*目标*中指定的标记。此外，具体取决于绑定表达式，生成的代码可能会观察到的值中的更改*源*属性并刷新*目标*属性，并可能会将更改推从*目标*回*源*。

> **IMPORTANT**
> 已编译的绑定当前已禁用定义任何绑定表达式 `Source` 属性。这是因为 `Source` 属性始终设置使用 `x:Reference` 标记扩展，在编译时无法解析。

使用已编译的绑定

编译的颜色选择器页演示了如何使用 Xamarin.Forms 视图和 ViewModel 属性之间的已编译的绑定：

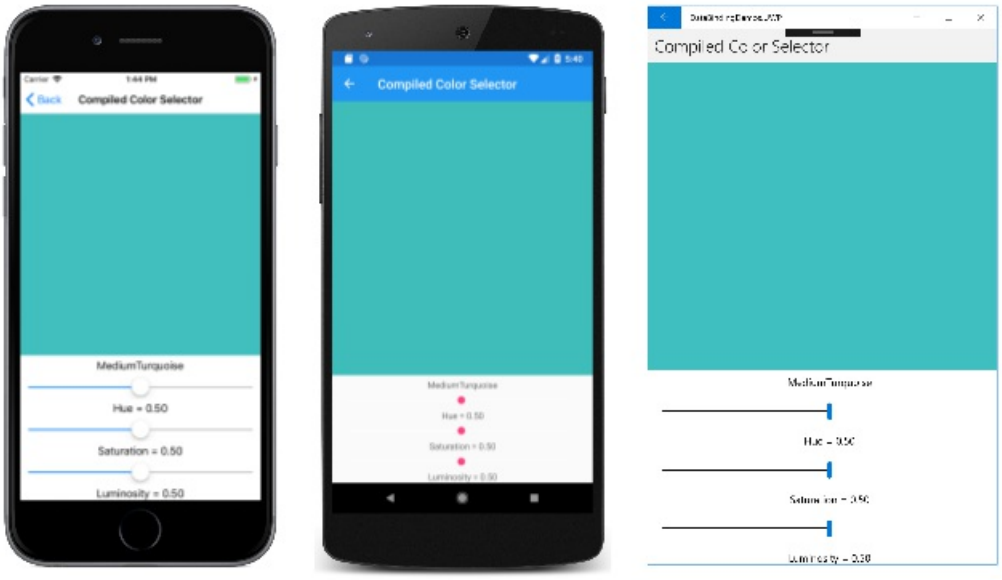
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.CompiledColorSelectorPage"
             Title="Compiled Color Selector">
    ...
    <StackLayout x:DataType="local:HslColorViewModel">
        <StackLayout.BindingContext>
            <local:HslColorViewModel Color="Sienna" />
        </StackLayout.BindingContext>
        <BoxView Color="{Binding Color}"
                 ... />
        <StackLayout Margin="10, 0">
            <Label Text="{Binding Name}" />
            <Slider Value="{Binding Hue}" />
            <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
            <Slider Value="{Binding Saturation}" />
            <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
            <Slider Value="{Binding Luminosity}" />
            <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

根 `StackLayout` 实例化 `HslColorViewModel` 并初始化 `Color` 属性的属性元素标记内 `BindingContext` 属性。此根 `StackLayout` 还定义了 `x:DataType` 属性为 ViewModel 类型，该值指示根中的任何绑定表达式 `StackLayout` 将编译视图层次结构。这可以通过更改绑定表达式绑定到不存在 ViewModel 属性，这将导致生成错误的任何验证。

> **IMPORTANT**
>
> `x:DataType` 属性可以是视图层次结构中的任何时候重新定义。

`BoxView`、`Label` 元素，以及 `Slider` 视图继承从绑定上下文 `StackLayout`. 这些视图是引用 ViewModel 中的源属性的所有绑定目标。有关 `BoxView.Color` 属性，并 `Label.Text` 属性，数据绑定是 `OneWay` – 从 ViewModel 中的属性设置视图中的属性。但是，`Slider.Value` 属性使用 `TwoWay` 绑定。这样，每个 `Slider` 若要从 ViewModel 中，以及若要设置从每个 viewmodel 设置 `Slider`。

当首次运行应用程序时，`BoxView`、`Label` 元素，以及 `Slider` 元素是从基于 ViewModel 的所有组初始 `Color` 时实例化 ViewModel 设置的属性。以下屏幕截图所示：



滑块操作，如 `BoxView` 并 `Label` 相应地更新元素。

有关此颜色选择器的详细信息，请参阅Viewmodel 和属性更改通知。

在 DataTemplate 中使用已编译的绑定

中的绑定 `DataTemplate` 要模板化的对象的上下文中解释。因此，使用编译时中的绑定 `DataTemplate`，则 `DataTemplate` 需要使用其数据对象的类型声明 `x:DataType` 属性。

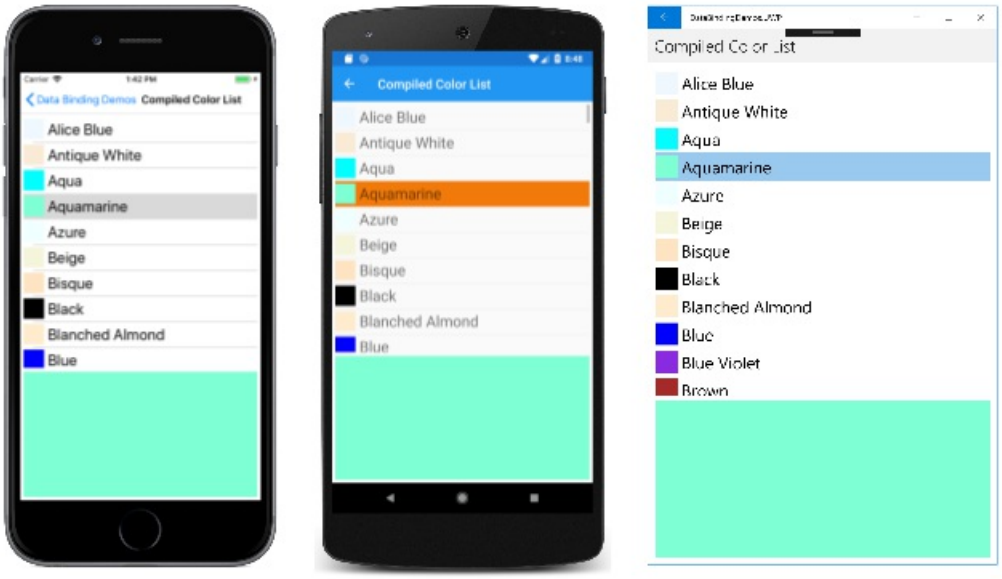
编译的颜色列表页上演示了如何使用中的已编译的绑定 `DataTemplate`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataBindingDemos"
             x:Class="DataBindingDemos.CompiledColorListPage"
             Title="Compiled Color List">
    <Grid>
        ...
        <ListView x:Name="colorListView"
                  ItemsSource="{x:Static local:NamedColor.All}"
                  ... >
            <ListView.ItemTemplate>
                <DataTemplate x:DataType="local:NamedColor">
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <BoxView Color="{Binding Color}"
                                     ... />
                            <Label Text="{Binding FriendlyName}"
                                   ... />
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
        <!-- The BoxView doesn't use compiled bindings -->
        <BoxView Color="{Binding Source={x:Reference colorListView}, Path=SelectedItem.Color}"
                 ... />
    </Grid>
</ContentPage>
```

`ListView.ItemsSource` 属性设置为静态 `NamedColor.All` 属性。 `NamedColor` 类使用.NET 反射来枚举中的所有静态公共字段 `Color` 结构，并将其存储在可从静态访问集合中其名称与 `All` 属性。因此， `ListView` 的所有填充 `NamedColor` 实例。中每一项 `ListView`，该项目的绑定上下文设置为 `NamedColor` 对象。 `BoxView` 并 `Label` 中的元素 `ViewCell` 绑定到 `NamedColor` 属性。

请注意， `DataTemplate` 定义 `x:DataType` 属性不存在 `NamedColor` 的任何指示的类型绑定表达式中的 `DataTemplate` 将编译视图层次结构。这可以通过更改绑定表达式来绑定到不存在的任何验证 `NamedColor` 属性，这将导致生成错误。

当首次运行应用程序时， `ListView` 填入 `NamedColor` 实例。中的项后 `ListView` 已选中 `BoxView.Color` 属性设置为中的选定项的颜色 `ListView`：



选择其中的其他项 `ListView` 更新的颜色 `BoxView` 。

组合编译使用经典的绑定的绑定

绑定表达式仅编译为视图层次结构的 `x:DataType` 上定义属性。相反，任何视图层次结构中的依据 `x:DataType` 属性将使用经典的绑定。因此，它是将已编译的绑定和在页面上的经典绑定。例如，在以前部分中的视图 `DataTemplate` 使用已编译的绑定，而 `BoxView` 设置为在选定的颜色 `ListView` 却没有。

仔细构造的 `x:DataType` 属性可能因此会导致使用已编译和经典模型绑定的页。或者，`x:DataType` 特性可以重新定义为视图层次结构中的任何时候 `null` 使用 `x:Null` 标记扩展。执行此操作表示视图层次结构中的任何绑定表达式将使用经典的绑定。*混合绑定*页展示这种方法：

```
<StackLayout x:DataType="local:HslColorViewModel">
    <StackLayout.BindingContext>
        <local:HslColorViewModel Color="Sienna" />
    </StackLayout.BindingContext>
    <BoxView Color="{Binding Color}"
             VerticalOptions="FillAndExpand" />
    <StackLayout x:DataType="{x:Null}"
                 Margin="10, 0">
        <Label Text="{Binding Name}" />
        <Slider Value="{Binding Hue}" />
        <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />
        <Slider Value="{Binding Saturation}" />
        <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />
        <Slider Value="{Binding Luminosity}" />
        <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
    </StackLayout>
</StackLayout>
```

根 `StackLayout` 设置 `x:DataType` 属性不存在 `HslColorViewModel` 的任何指示的类型绑定表达式的根目录中 `StackLayout` 将编译视图层次结构。但是，内部 `StackLayout` 重新定义 `x:DataType` 归于 `null` 与 `x:Null` 标记表达式。因此，绑定表达式中内部 `StackLayout` 使用经典的绑定。仅 `BoxView` ，在根目录 `StackLayout` 查看层次结构中，使用编译的绑定。

有关详细信息 `x:Null` 标记表达式，请参阅X:null 标记扩展。

性能

已编译的绑定来提高数据绑定性能，具有性能优势不同。单元测试将显示的：

- 使用属性更改通知的已编译的绑定 (即 `OneWay`、 `OneWayToSource`，或 `TwoWay` 绑定) 解析比经典绑定要快大约 8 次。
- 已编译的绑定不使用属性更改通知 (即 `OneTime` 绑定) 解析比经典绑定要快大约 20 倍。
- 设置 `BindingContext` 在已编译的绑定使用属性更改通知 (即 `OneWay`、 `OneWayToSource`，或 `TwoWay` 绑定) 比大约 5 次设置 `BindingContext` 经典绑定上。
- 设置 `BindingContext` 在已编译的绑定不会使用属性的更改通知 (即 `OneTime` 绑定) 比大约 7 次设置 `BindingContext` 经典绑定上。

这些性能差异可以正在使用的操作系统和应用程序正在其运行的设备版本放大取决于正在使用的平台的移动设备上。

相关链接

- 数据绑定演示（示例）

Xamarin.Forms DependencyService

2018/6/9 • • Edit Online

Xamarin.Forms 允许开发人员定义特定于平台的项目中的行为。DependencyService 然后查找适当的平台实现，允许访问本机功能的共享的代码。

本指南由组成的以下文章：

- **介绍** –引入了的总体体系结构 `DependencyService` 概念。
- **实现文本到语音转换** –将指导完成使用每个平台的本机文本到语音转换系统的示例。
- **检查设备方向** –将指导完成使用本机平台 Api 来确定设备的方向的示例。
- **获取电池信息** –将指导完成使用本机 Api 以获取有关电池的状态信息的示例。
- **从库中选取照片** –将指导完成使用本机 Api 从电话的图片库中选择一张照片的示例。

相关链接

- 使用 DependencyService（示例）
- DependencyService（示例）
- Xamarin.Forms 示例

DependencyService 简介

2018/11/1 • • Edit Online

概述

`DependencyService` 允许应用从共享代码调用特定于平台的功能。此功能通过 Xamarin.Forms 应用，若要执行的本机应用程序可以执行任何操作。

`DependencyService` 是服务定位器。在实践中，定义了一个接口和 `DependencyService` 查找该接口从各种平台项目的正确实现。

> **NOTE**
> 默认情况下 `DependencyService` 将唯一解析平台具有无参数构造函数的实现。但是，依赖关系解析方法可以注入到 Xamarin.Forms 使用依赖关系注入容器或工厂方法来解析平台的实现。这种方法可以用于解决平台实现个构造函数的参数。有关详细信息，请参阅Xamarin.Forms 中的依赖项解析。

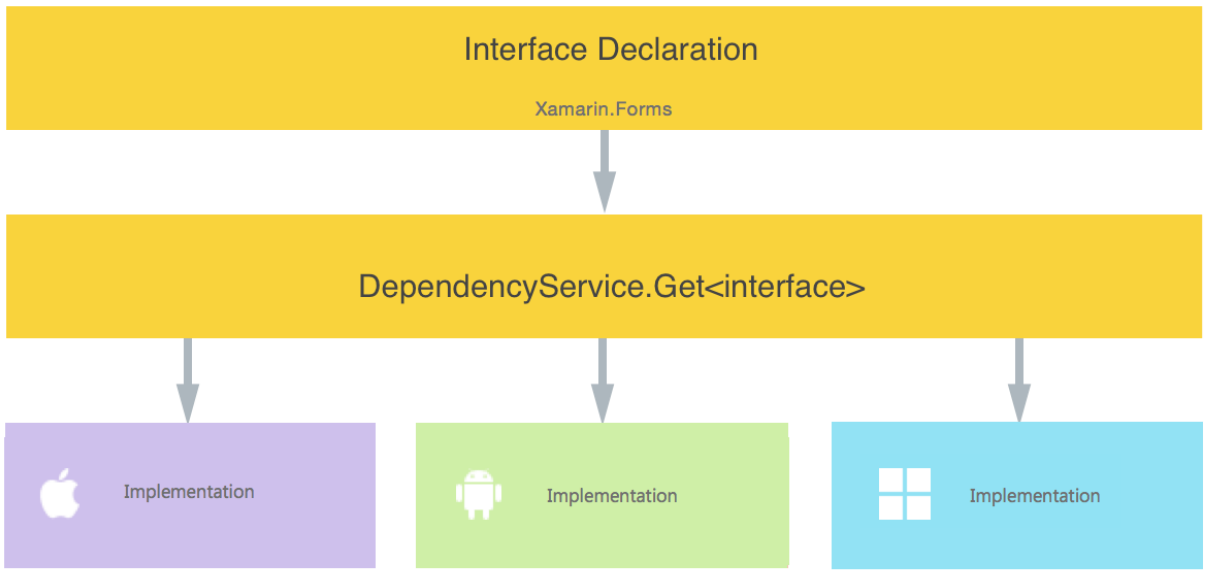
DependencyService 的工作原理

Xamarin.Forms 应用需要四个组件来使用 `DependencyService`：

- **接口**–由共享代码中的接口定义所需的功能。
- **实现每个平台**–实现接口的类必须添加到每个平台项目。
- **注册**–每个实现类必须与注册 `DependencyService` 通过元数据属性。注册启用 `DependencyService` 来找到实现类并将其提供替代该接口在运行时。
- **调用到 DependencyService** –共享代码需要显式调用 `DependencyService` 寻求接口的实现。

请注意，必须为每个平台项目在解决方案中提供实现。平台项目中的不包含实现将在运行时失败。

下图介绍了应用程序的结构：



接口

您设计的接口将定义如何与特定于平台的功能进行交互。注意如果你正在开发的组件或 NuGet 包作为共享的组件。API 设计可以执行或破坏包。下面的示例指定语音文本的允许的灵活指定字要朗读但由要为每个平台自定义的实现简单的界面：

```
public interface ITextToSpeech {
    void Speak ( string text ); //note that interface members are public by default
}
```

每个平台的实现

一旦合适接口在设计时，必须为您面向的每个平台项目中实现该接口。例如，以下类实现 `ITextToSpeech` 在 iOS 上的接口：

```
namespace UsingDependencyService.iOS
{
    public class TextToSpeech_iOS : ITextToSpeech
    {
        public void Speak (string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer ();

            var speechUtterance = new AVSpeechUtterance (text) {
                Rate = AVSpeechUtterance.MaximumSpeechRate/4,
                Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance (speechUtterance);
        }
    }
}
```

注册

每个接口的实现需要使用注册 `DependencyService` 与元数据特性。下面的代码注册适用于 iOS 的实现：

```
[assembly: Dependency (typeof (TextToSpeech_iOS))]
namespace UsingDependencyService.iOS
{
  ...
}
```

将所有组合在一起，特定于平台的实现如下所示：

```
[assembly: Dependency (typeof (TextToSpeech_iOS))]
namespace UsingDependencyService.iOS
{
    public class TextToSpeech_iOS : ITextToSpeech
    {
        public void Speak (string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer ();

            var speechUtterance = new AVSpeechUtterance (text) {
                Rate = AVSpeechUtterance.MaximumSpeechRate/4,
                Voice = AVSpeechSynthesisVoice.FromLanguage ("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance (speechUtterance);
        }
    }
}
```

注意：在命名空间级别而不是类级别执行注册。

通用 Windows 平台.NET Native 编译

使用.NET Native 编译选项的 UWP 项目应遵循略有不同的配置初始化 Xamarin.Forms 时。.NET native 编译依赖关系服务还要求略有不同的注册。

在中**App.xaml.cs**文件中，手动注册每个依赖关系服务，在 UWP 项目中使用定义 `Register<T>` 方法，如下所示：

```
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
// register the dependencies in the same
Xamarin.Forms.DependencyService.Register<TextToSpeechImplementation>();
```

注意：手动注册使用 `Register<T>` 是使用.NET Native 编译生成的版本中的才有效。如果省略此行，调试版本中仍将起作用，但发布版本将无法加载依赖关系服务。

调用到 DependencyService

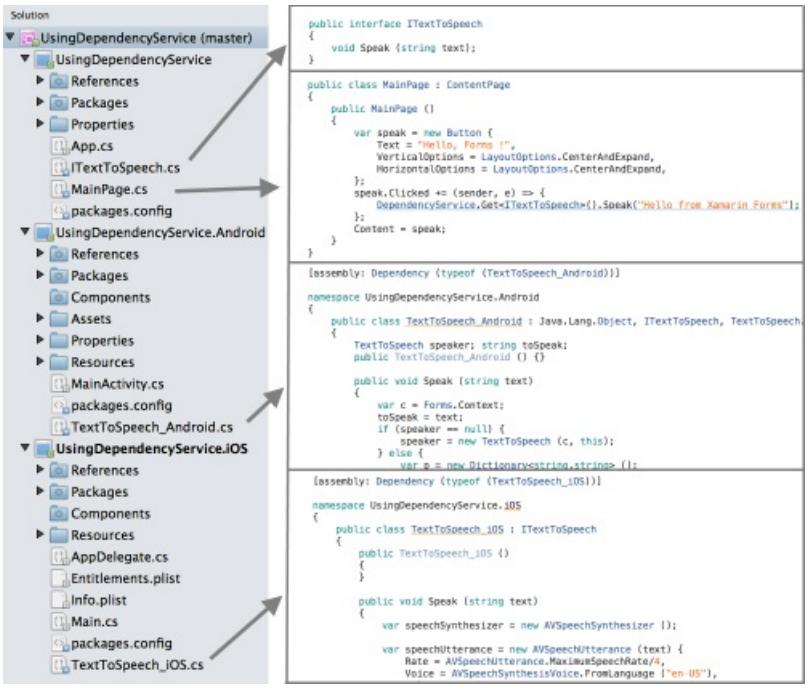
一旦项目已具有一个公共接口和实现为每个平台已设置，使用 `DependencyService` 若要在运行时获取正确的实现：

```
DependencyService.Get<ITextToSpeech>().Speak("Hello from Xamarin Forms");
```

`DependencyService.Get<T>` 将找到正确的接口实现 `T`。

解决方案结构

示例 UsingDependencyService 解决方案是适用于 iOS 和 Android 如下所示，使用上面所述的代码更改突出显示。



NOTE

您**必**须提供每个平台项目中的实现。如果不注册了任何接口实现，则 `DependencyService` 将不能解决 `Get<T>()` 方法在运行时。

相关链接

- DependencyServiceSample
- Xamarin.Forms 示例

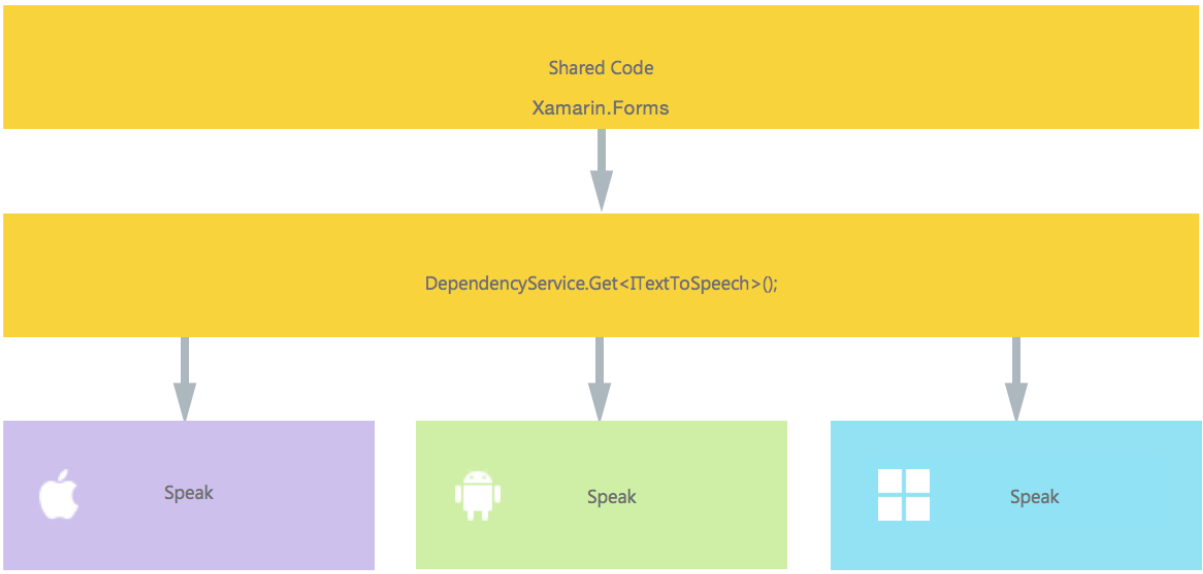
实现文本到语音转换

2018/10/19 • • Edit Online

创建使用的跨平台应用这篇文章将指导你 `DependencyService` 访问本机文本到语音转换 Api:

- **创建接口** –了解如何共享代码中创建接口。
- **iOS 实现** –了解如何在适用于 iOS 的本机代码中实现接口。
- **Android 实现** –了解如何适用于 Android 的本机代码中实现的接口。
- **UWP 实现** –了解如何为通用 Windows 平台 (UWP) 中的本机代码实现的接口。
- **在共享代码中实现** –了解如何使用 `DependencyService` 来调入本机实现从共享代码。

应用程序使用 `DependencyService` 将具有以下结构:



创建界面

首先，表达你打算实现的功能的共享代码中创建的接口。对于此示例，该接口包含一个方法， `Speak`:

```
public interface ITextToSpeech
{
    void Speak (string text);
}
```

针对此接口在共享代码中编写代码将允许 Xamarin.Forms 应用程序访问每个平台上的语音 Api。

> **NOTE**
> 实现接口的类必须具有无参数的构造函数，以使用 `DependencyService` 。

iOS 实现

必须在每个特定于平台的应用程序项目中实现该接口。请注意，类具有无参数构造函数，以便 `DependencyService` 可以创建新实例。

```
[assembly: Dependency(typeof(TextToSpeechImplementation))]
namespace DependencyServiceSample.iOS
{

    public class TextToSpeechImplementation : ITextToSpeech
    {
        public TextToSpeechImplementation() { }

        public void Speak(string text)
        {
            var speechSynthesizer = new AVSpeechSynthesizer();
            var speechUtterance = new AVSpeechUtterance(text)
            {
                Rate = AVSpeechUtterance.MaximumSpeechRate / 4,
                Voice = AVSpeechSynthesisVoice.FromLanguage("en-US"),
                Volume = 0.5f,
                PitchMultiplier = 1.0f
            };

            speechSynthesizer.SpeakUtterance(speechUtterance);
        }
    }
}
```

`[assembly]` 属性的实现作为注册类 `ITextToSpeech` 接口，这意味着 `DependencyService.Get<ITextToSpeech>()` 可用于共享代码中创建它的一个实例。

Android 实现

Android 代码将更复杂的 iOS 版本比：它需要实现的类从特定于 Android 的继承 `Java.Lang.Object` 并实现 `IOnInitListener` 接口也。它还需要访问当前的 Android 上下文，它公开由 `MainActivity.Instance` 属性。

```
[assembly: Dependency(typeof(TextToSpeechImplementation))]
namespace DependencyServiceSample.Droid
{
    public class TextToSpeechImplementation : Java.Lang.Object, ITextToSpeech, TextToSpeech.IOnInitListener
    {
        TextToSpeech speaker;
        string toSpeak;

        public void Speak(string text)
        {
            toSpeak = text;
            if (speaker == null)
            {
                speaker = new TextToSpeech(MainActivity.Instance, this);
            }
            else
            {
                speaker.Speak(toSpeak, QueueMode.Flush, null, null);
            }
        }

        public void OnInit(OperationResult status)
        {
            if (status.Equals(OperationResult.Success))
            {
                speaker.Speak(toSpeak, QueueMode.Flush, null, null);
            }
        }
    }
}
```

`[assembly]` 属性的实现作为注册类 `ITextToSpeech` 接口，这意味着 `DependencyService.Get<ITextToSpeech>()` 可用于共享代码中创建它的一个实例。

通用 Windows 平台实现

通用 Windows 平台具有中的语音 API `Windows.Media.SpeechSynthesis` 命名空间。唯一需要注意的是记住勾选**麦克风**功能在清单中，否则访问到语音 Api 被阻止。

```
[assembly:Dependency(typeof(TextToSpeechImplementation))]
public class TextToSpeechImplementation : ITextToSpeech
{
    public async void Speak(string text)
    {
        var mediaElement = new MediaElement();
        var synth = new Windows.Media.SpeechSynthesis.SpeechSynthesizer();
        var stream = await synth.SynthesizeTextToStreamAsync(text);

        mediaElement.SetSource(stream, stream.ContentType);
        mediaElement.Play();
    }
}
```

`[assembly]` 属性的实现作为注册类 `ITextToSpeech` 接口，这意味着 `DependencyService.Get<ITextToSpeech>()` 可用于共享代码中创建它的一个实例。

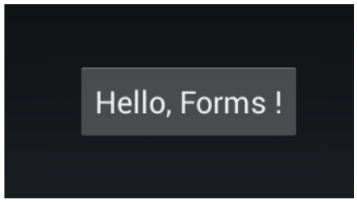
在共享代码中实现

现在我们可以编写并测试访问文本到语音转换接口的共享的代码。此简单的页面包括触发的语音功能的按钮。它使用 `DependencyService` 若要获取的实例 `ITextToSpeech` 接口–此实例将在运行时是具有完全访问权限的本机 SDK 的特定于平台的实现。

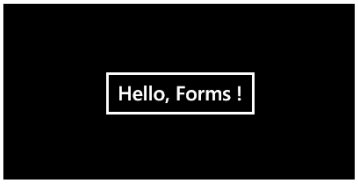
```
public MainPage ()
{
    var speak = new Button {
        Text = "Hello, Forms !",
        VerticalOptions = LayoutOptions.CenterAndExpand,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
    speak.Clicked += (sender, e) => {
        DependencyService.Get<ITextToSpeech>().Speak("Hello from Xamarin Forms");
    };
    Content = speak;
}
```

在 iOS、Android 或 UWP 上运行此应用程序并按下按钮会讲到，每个平台上使用本机语音 SDK 的应用程序。

iOS



Android



WinPhone & UWP

相关链接

- 使用 DependencyService（示例）

- DependencyServiceSample

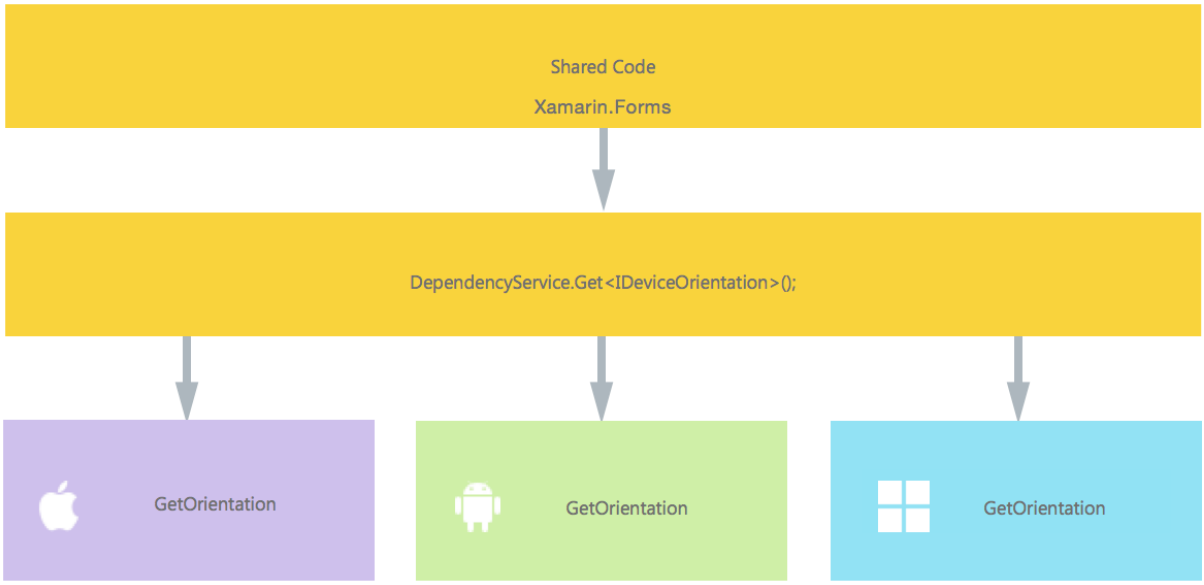
检查设备方向

2018/10/26 • • Edit Online

本文将指导你使用 `DependencyService` 检查设备方向从每个平台上使用本机 Api 的共享代码。本演练基于现有 `DeviceOrientation` 阿里 Özgür 的插件。请参阅GitHub 存储库有关详细信息。

- **创建接口** –了解如何在接口中共享代码创建。
- **iOS 实现** –了解如何在适用于 iOS 的本机代码中实现接口。
- **Android 实现** –了解如何适用于 Android 的本机代码中实现的接口。
- **UWP 实现** –了解如何为通用 Windows 平台 (UWP) 中的本机代码实现的接口。
- **在共享代码中实现** –了解如何使用 `DependencyService` 来调入本机实现从共享代码。

应用程序使用 `DependencyService` 将具有以下结构：



> **NOTE**
> 可以检测是否在设备处于纵向或横向方向中的共享代码，如中所示设备方向。在本文中所述的方法使用本机功能来获取有关方向，包括设备在正面朝下的详细信息。

创建界面

首先，表达你打算实现的功能的共享代码中创建的接口。对于此示例，该接口包含一个方法：

```
namespace DependencyServiceSample.Abstractions
{
    public enum DeviceOrientations
    {
        Undefined,
        Landscape,
        Portrait
    }

    public interface IDeviceOrientation
    {
        DeviceOrientations GetOrientation();
    }
}
```

针对此接口在共享代码中编写代码将允许 Xamarin.Forms 应用程序访问每个平台上的设备方向 Api。

> **NOTE**
> 实现接口的类必须具有无参数的构造函数，以使用 `DependencyService`。

iOS 实现

必须在每个特定于平台的应用程序项目中实现该接口。请注意，类具有无参数构造函数，以便 `DependencyService` 可以创建新实例：

```
using UIKit;
using Foundation;

namespace DependencyServiceSample.iOS
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation(){ }

        public DeviceOrientations GetOrientation()
        {
            var currentOrientation = UIApplication.SharedApplication.StatusBarOrientation;
            bool isPortrait = currentOrientation == UIInterfaceOrientation.Portrait
                || currentOrientation == UIInterfaceOrientation.PortraitUpsideDown;

            return isPortrait ? DeviceOrientations.Portrait: DeviceOrientations.Landscape;
        }
    }
}
```

最后，添加以下 `[assembly]` 包括任何所需属性该类上方（和任何已定义的命名空间之外），`using` 语句：

```
using UIKit;
using Foundation;
using DependencyServiceSample.iOS; //enables registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (DeviceOrientationImplementation))]
namespace DependencyServiceSample.iOS {
    ...
```

此属性的实现作为注册类 `IDeviceOrientation` 接口，这意味着 `DependencyService.Get<IDeviceOrientation>` 可用于共享代码中创建它的一个实例。

Android 实现

下面的代码实现 `IDeviceOrientation` 在 Android 上：

```
using DependencyServiceSample.Droid;
using Android.Hardware;

namespace DependencyServiceSample.Droid
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation() { }

        public static void Init() { }

        public DeviceOrientations GetOrientation()
        {
            IWindowManager windowManager =
Android.App.Application.Context.GetSystemService(Context.WindowService).JavaCast<IWindowManager>();

            var rotation = windowManager.DefaultDisplay.Rotation;
            bool isLandscape = rotation == SurfaceOrientation.Rotation90 || rotation ==
SurfaceOrientation.Rotation270;
            return isLandscape ? DeviceOrientations.Landscape : DeviceOrientations.Portrait;
        }
    }
}
```

添加以下 `[assembly]` 包括任何所需属性该类上方（和任何已定义的命名空间之外），`using` 语句：

```
using DependencyServiceSample.Droid; //enables registration outside of namespace
using Android.Hardware;

[assembly: Xamarin.Forms.Dependency (typeof (DeviceOrientationImplementation))]
namespace DependencyServiceSample.Droid {
    ...
```

此属性的实现作为注册类 `IDeviceOrientaiton` 接口，这意味着 `DependencyService.Get<IDeviceOrientation>` 可在共享的代码可以创建它的一个实例。

通用 Windows 平台实现

下面的代码实现 `IDeviceOrientation` 通用 Windows 平台上的接口：

```
namespace DependencyServiceSample.WindowsPhone
{
    public class DeviceOrientationImplementation : IDeviceOrientation
    {
        public DeviceOrientationImplementation() { }

        public DeviceOrientations GetOrientation()
        {
            var orientation = Windows.UI.ViewManagement.ApplicationView.GetForCurrentView().Orientation;
            if (orientation == Windows.UI.ViewManagement.ApplicationViewOrientation.Landscape) {
                return DeviceOrientations.Landscape;
            }
            else {
                return DeviceOrientations.Portrait;
            }
        }
    }
}
```

添加 `[assembly]` 包括任何所需属性该类上方（和任何已定义的命名空间之外），`using` 语句：

```
using DependencyServiceSample.WindowsPhone; //enables registration outside of namespace

[assembly: Dependency(typeof(DeviceOrientationImplementation))]
namespace DependencyServiceSample.WindowsPhone {
    ...
```

此属性的实现作为注册类 `DeviceOrientationImplementation` 接口，这意味着 `DependencyService.Get<IDeviceOrientation>` 可在共享的代码可以创建它的一个实例。

在共享代码中实现

现在，我们可以编写并测试访问的共享的代码 `IDeviceOrientation` 接口。此简单的页面包括一个按钮，更新其自己基于设备方向的文本。它使用 `DependencyService` 若要获取的实例 `IDeviceOrientation` 接口–此实例将在运行时是具有完全访问权限的本机 SDK 的特定于平台的实现：

```
public MainPage ()
{
    var orient = new Button {
        Text = "Get Orientation",
        VerticalOptions = LayoutOptions.CenterAndExpand,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
    orient.Clicked += (sender, e) => {
       var orientation = DependencyService.Get<IDeviceOrientation>().GetOrientation();
       switch(orientation){
           case DeviceOrientations.Undefined:
                orient.Text = "Undefined";
                break;
           case DeviceOrientations.Landscape:
                orient.Text = "Landscape";
                break;
           case DeviceOrientations.Portrait:
                orient.Text = "Portrait";
                break;
       }
    };
    Content = orient;
}
```

在 iOS、Android 或 Windows 平台上运行此应用程序并按下按钮将导致该按钮的文本更新设备的方向。

相关链接

- 使用 DependencyService（示例）
- DependencyService（示例）
- Xamarin.Forms 示例

正在检查电池状态

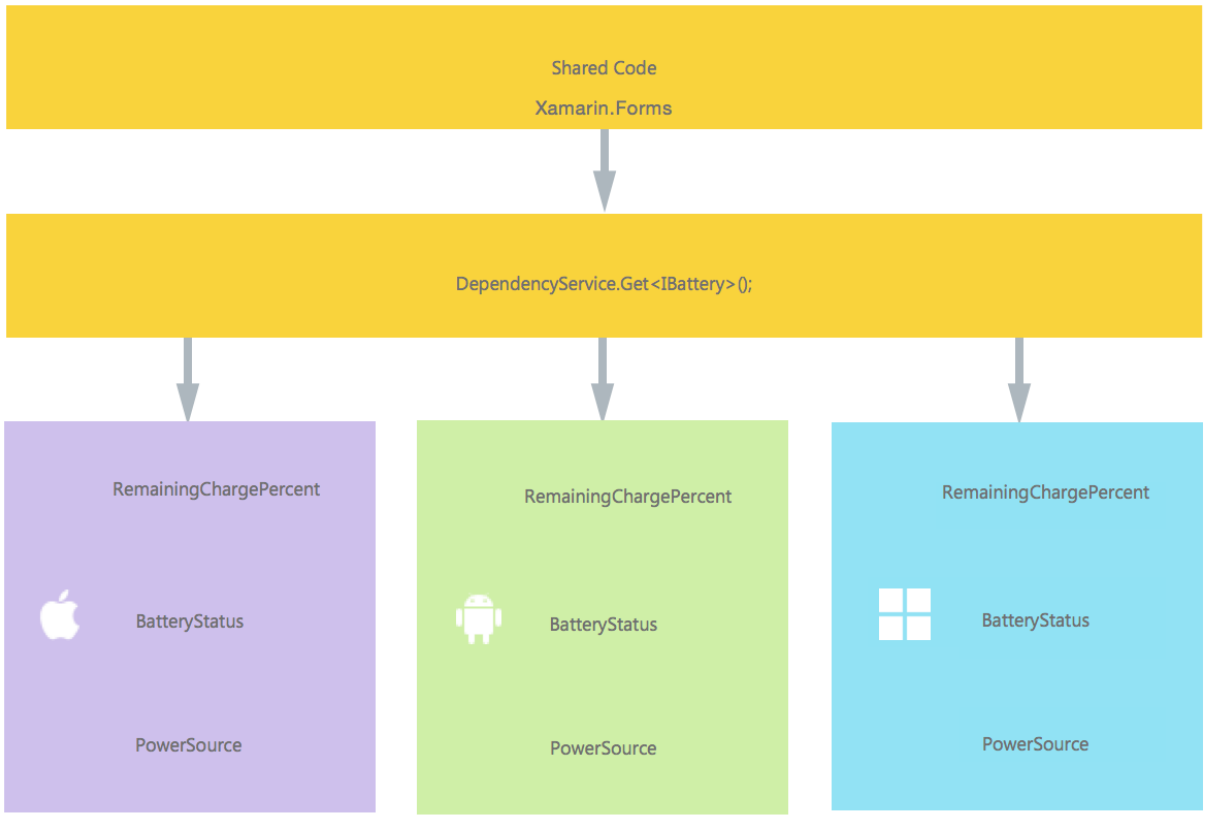
2018/7/13 • • Edit Online

本文将指导完成创建应用程序，用于检查电池状态。这篇文章基于 James montemagno 电池插件。有关详细信息，请参阅GitHub 存储库。

因为 Xamarin.Forms 不包括用于检查当前的电池状态的功能，此应用程序将需要使用 `DependencyService` 以充分利用本机 Api。本文将介绍以下步骤使用 `DependencyService`：

- **创建接口** –了解如何共享代码中创建接口。
- **iOS 实现** –了解如何在适用于 iOS 的本机代码中实现接口。
- **Android 实现** –了解如何适用于 Android 的本机代码中实现的接口。
- **通用 Windows 平台实现** –了解如何为通用 Windows 平台 (UWP) 中的本机代码实现的接口。
- **在共享代码中实现** –了解如何使用 `DependencyService` 来调入本机实现从共享代码。

完成后，应用程序使用 `DependencyService` 将具有以下结构：



创建界面

首先，创建一个接口，表示所需的功能的共享代码中。在电池检查应用程序的情况下的相关信息是电池剩余的百分比，无论设备已充电，且在设备接通电源的方式：

```
namespace DependencyServiceSample
{
  public enum BatteryStatus
  {
    Charging,
    Discharging,
    Full,
    NotCharging,
    Unknown
  }

  public enum PowerSource
  {
    Battery,
    Ac,
    Usb,
    Wireless,
    Other
  }

  public interface IBattery
  {
    int RemainingChargePercent { get; }
    BatteryStatus Status { get; }
    PowerSource PowerSource { get; }
  }
}
```

针对此接口在共享代码中编写代码将允许 Xamarin.Forms 应用程序访问每个平台上的电源管理 Api。

> **NOTE**
> 实现接口的类必须具有无参数的构造函数，以使用 `DependencyService` 。不能由接口定义构造函数。

iOS 实现

`IBattery` 接口必须实现每个特定于平台的应用程序项目中。IOS 实现将使用本机 `UIDevice` 访问电池信息的 Api。请注意，下面的类具有无参数构造函数，以便 `DependencyService` 可以创建新实例：

```
using UIKit;
using Foundation;
using DependencyServiceSample.iOS;

namespace DependencyServiceSample.iOS
{
  public class BatteryImplementation : IBattery
  {
    public BatteryImplementation()
    {
      UIDevice.CurrentDevice.BatteryMonitoringEnabled = true;
    }

    public int RemainingChargePercent
    {
      get
      {
        return (int)(UIDevice.CurrentDevice.BatteryLevel * 100F);
      }
    }

    public BatteryStatus Status
    {
      get
      {
        switch (UIDevice.CurrentDevice.BatteryState)
        {
          case UIDeviceBatteryState.Charging:
            return BatteryStatus.Charging;
          case UIDeviceBatteryState.Full:
            return BatteryStatus.Full;
          case UIDeviceBatteryState.Unplugged:
            return BatteryStatus.Discharging;
          default:
            return BatteryStatus.Unknown;
        }
      }
    }

    public PowerSource PowerSource
    {
      get
      {
        switch (UIDevice.CurrentDevice.BatteryState)
        {
          case UIDeviceBatteryState.Charging:
            return PowerSource.Ac;
          case UIDeviceBatteryState.Full:
            return PowerSource.Ac;
          case UIDeviceBatteryState.Unplugged:
            return PowerSource.Battery;
          default:
            return PowerSource.Other;
        }
      }
    }
  }
}
```

最后，添加以下 `[assembly]` 包括任何所需属性该类上方（和任何已定义的命名空间之外），`using` 语句：

```
using UIKit;
using Foundation;
using DependencyServiceSample.iOS;//necessary for registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (BatteryImplementation))]
namespace DependencyServiceSample.iOS
{
  public class BatteryImplementation : IBattery {
    ...
```

此属性的实现作为注册类 `IBattery` 接口，这意味着 `DependencyService.Get<IBattery>` 可用于共享代码中创建它的一个实例：

Android 实现

Android 实现使用 `Android.OS.BatteryManager` API。此实现起来更为复杂的 iOS 版本，需要检查，以处理缺少的电池权限：

```
using System;
using Android;
using Android.Content;
using Android.App;
using Android.OS;
using BatteryStatus = Android.OS.BatteryStatus;
using DependencyServiceSample.Droid;

namespace DependencyServiceSample.Droid
{
  public class BatteryImplementation : IBattery
  {
    private BatteryBroadcastReceiver batteryReceiver;
    public BatteryImplementation() { }

    public int RemainingChargePercent
    {
      get
      {
        try
        {
          using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
          {
            using (var battery = Application.Context.RegisterReceiver(null, filter))
            {
              var level = battery.GetIntExtra(BatteryManager.ExtraLevel, -1);
              var scale = battery.GetIntExtra(BatteryManager.ExtraScale, -1);

              return (int)Math.Floor(level * 100D / scale);
            }
          }
        }
        catch
        {
          System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
          throw;
        }

      }
    }

    public DependencyServiceSample.BatteryStatus Status
    {
      get
      {
        try
```

```
    {
      using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
      {
        using (var battery = Application.Context.RegisterReceiver(null, filter))
        {
          int status = battery.GetIntExtra(BatteryManager.ExtraStatus, -1);
          var isCharging = status == (int)BatteryStatus.Charging || status == (int)BatteryStatus.Full;

          var chargePlug = battery.GetIntExtra(BatteryManager.ExtraPlugged, -1);
          var usbCharge = chargePlug == (int)BatteryPlugged.Usb;
          var acCharge = chargePlug == (int)BatteryPlugged.Ac;
          bool wirelessCharge = false;
          wirelessCharge = chargePlug == (int)BatteryPlugged.Wireless;

          isCharging = (usbCharge || acCharge || wirelessCharge);
          if (isCharging)
            return DependencyServiceSample.BatteryStatus.Charging;

          switch(status)
          {
            case (int)BatteryStatus.Charging:
              return DependencyServiceSample.BatteryStatus.Charging;
            case (int)BatteryStatus.Discharging:
              return DependencyServiceSample.BatteryStatus.Discharging;
            case (int)BatteryStatus.Full:
              return DependencyServiceSample.BatteryStatus.Full;
            case (int)BatteryStatus.NotCharging:
              return DependencyServiceSample.BatteryStatus.NotCharging;
            default:
              return DependencyServiceSample.BatteryStatus.Unknown;
          }
        }
      }
    }
    catch
    {
      System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
      throw;
    }
  }
}

public PowerSource PowerSource
{
  get
  {
    try
    {
      using (var filter = new IntentFilter(Intent.ActionBatteryChanged))
      {
        using (var battery = Application.Context.RegisterReceiver(null, filter))
        {
          int status = battery.GetIntExtra(BatteryManager.ExtraStatus, -1);
          var isCharging = status == (int)BatteryStatus.Charging || status == (int)BatteryStatus.Full;

          var chargePlug = battery.GetIntExtra(BatteryManager.ExtraPlugged, -1);
          var usbCharge = chargePlug == (int)BatteryPlugged.Usb;
          var acCharge = chargePlug == (int)BatteryPlugged.Ac;

          bool wirelessCharge = false;
          wirelessCharge = chargePlug == (int)BatteryPlugged.Wireless;

          isCharging = (usbCharge || acCharge || wirelessCharge);

          if (!isCharging)
            return DependencyServiceSample.PowerSource.Battery;
          else if (usbCharge)
            return DependencyServiceSample.PowerSource.Usb;
          else if (acCharge)
```

```
                return DependencyServiceSample.PowerSource.Ac;
              else if (wirelessCharge)
                return DependencyServiceSample.PowerSource.Wireless;
              else
                return DependencyServiceSample.PowerSource.Other;
            }
          }
        }
        catch
        {
          System.Diagnostics.Debug.WriteLine ("Ensure you have android.permission.BATTERY_STATS");
          throw;
        }
      }
    }
  }
}
```

添加以下 `[assembly]` 包括任何所需属性该类上方（和任何已定义的命名空间之外），`using` 语句：

```
...
using BatteryStatus = Android.OS.BatteryStatus;
using DependencyServiceSample.Droid; //enables registration outside of namespace

[assembly: Xamarin.Forms.Dependency (typeof (BatteryImplementation))]
namespace DependencyServiceSample.Droid
{
  public class BatteryImplementation : IBattery {
    ...
```

此属性的实现作为注册类 `IBattery` 接口，这意味着 `DependencyService.Get<IBattery>` 可在共享的代码可以创建它的一个实例。

通用 Windows 平台实现

UWP 实现使用 `Windows.Devices.Power` Api 以获取电池状态信息：

```
using DependencyServiceSample.UWP;
using Xamarin.Forms;

[assembly: Dependency(typeof(BatteryImplementation))]
namespace DependencyServiceSample.UWP
{
    public class BatteryImplementation : IBattery
    {
        private BatteryStatus status = BatteryStatus.Unknown;
        Windows.Devices.Power.Battery battery;

        public BatteryImplementation()
        {
        }

        private Windows.Devices.Power.Battery DefaultBattery
        {
            get
            {
                return battery ?? (battery = Windows.Devices.Power.Battery.AggregateBattery);
            }
        }

        public int RemainingChargePercent
        {
            get
            {
```

```
                var finalReport = DefaultBattery.GetReport();
                var finalPercent = -1;

                if (finalReport.RemainingCapacityInMilliwattHours.HasValue &&
finalReport.FullChargeCapacityInMilliwattHours.HasValue)
                {
                    finalPercent = (int)((finalReport.RemainingCapacityInMilliwattHours.Value /
                        (double)finalReport.FullChargeCapacityInMilliwattHours.Value) * 100);
                }
                return finalPercent;
            }
        }

        public BatteryStatus Status
        {
            get
            {
                var report = DefaultBattery.GetReport();
                var percentage = RemainingChargePercent;

                if (percentage >= 1.0)
                {
                    status = BatteryStatus.Full;
                }
                else if (percentage < 0)
                {
                    status = BatteryStatus.Unknown;
                }
                else
                {
                    switch (report.Status)
                    {
                        case Windows.System.Power.BatteryStatus.Charging:
                            status = BatteryStatus.Charging;
                            break;
                        case Windows.System.Power.BatteryStatus.Discharging:
                            status = BatteryStatus.Discharging;
                            break;
                        case Windows.System.Power.BatteryStatus.Idle:
                            status = BatteryStatus.NotCharging;
                            break;
                        case Windows.System.Power.BatteryStatus.NotPresent:
                            status = BatteryStatus.Unknown;
                            break;
                    }
                }
                return status;
            }
        }

        public PowerSource PowerSource
        {
            get
            {
                if (status == BatteryStatus.Full || status == BatteryStatus.Charging)
                {
                    return PowerSource.Ac;
                }
                return PowerSource.Battery;
            }
        }
    }
}
```

`[assembly]`上面的命名空间声明的属性的实现作为注册类`IBattery`接口，这意味着`DependencyService.Get<IBattery>`可用于共享代码中创建它的一个实例。

在共享代码中实现

现在，已针对每个平台实现该接口，可以编写共享的应用程序以充分利用它。将包含应用程序具有一个按钮的页，当点击更新其文本与当前的电池状态。它使用 `DependencyService` 若要获取的实例 `IBattery` 接口。在运行时，此实例将是具有完全访问权限的本机 SDK 的特定于平台的实现。

```
public MainPage ()
{
    var button = new Button {
        Text = "Click for battery info",
        VerticalOptions = LayoutOptions.CenterAndExpand,
        HorizontalOptions = LayoutOptions.CenterAndExpand,
    };
    button.Clicked += (sender, e) => {
        var bat = DependencyService.Get<IBattery>();

        switch (bat.PowerSource){
          case PowerSource.Battery:
            button.Text = "Battery - ";
            break;
          case PowerSource.Ac:
            button.Text = "AC - ";
            break;
          case PowerSource.Usb:
            button.Text = "USB - ";
            break;
          case PowerSource.Wireless:
            button.Text = "Wireless - ";
            break;
          case PowerSource.Other:
          default:
            button.Text = "Other - ";
            break;
        }
        switch (bat.Status){
          case BatteryStatus.Charging:
            button.Text += "Charging";
            break;
          case BatteryStatus.Discharging:
            button.Text += "Discharging";
            break;
          case BatteryStatus.NotCharging:
            button.Text += "Not Charging";
            break;
          case BatteryStatus.Full:
            button.Text += "Full";
            break;
          case BatteryStatus.Unknown:
          default:
            button.Text += "Unknown";
            break;
        }
    };
    Content = button;
}
```

在 iOS 上运行此应用程序，Android 或 UWP 和按下按钮将导致更新以反映设备的当前电源状态的按钮文本。

相关链接

- DependencyService（示例）
- 使用 DependencyService（示例）
- Xamarin.Forms 示例

从图片库中选取照片

2018/10/26 • • Edit Online

本文将指导完成创建一个允许用户从电话的图片库中选取照片应用程序。因为 Xamarin.Forms 不包括此功能，它是使用所需 `DependencyService` 访问每个平台上的本机 Api。本文将介绍以下步骤使用 `DependencyService` 此任务：

- **创建接口** –了解如何共享代码中创建接口。
- **iOS 实现** –了解如何在适用于 iOS 的本机代码中实现接口。
- **Android 实现** –了解如何适用于 Android 的本机代码中实现的接口。
- **通用 Windows 平台实现** –了解如何为通用 Windows 平台 (UWP) 中的本机代码实现的接口。
- **在共享代码中实现** –了解如何使用 `DependencyService` 来调入本机实现从共享代码。

创建界面

首先，创建一个接口，表示所需的功能的共享代码中。对于照片选择应用程序，只是一种方法是必需的。在中定义的这 `IPicturePicker` .NET 标准库的示例代码中的接口：

```
namespace DependencyServiceSample
{
    public interface IPicturePicker
    {
        Task<Stream> GetImageStreamAsync();
    }
}
```

`GetImageStreamAsync` 方法定义为异步，因为该方法必须返回快速，但它不能返回 `Stream` 直到用户有浏览图片库，并选择了一个所选照片的对象。

在使用特定于平台的代码的所有平台实现此接口。

iOS 实现

IOS 实现 `IPicturePicker` 接口使用 `UIImagePickerController` 中所述**从库中选择一张照片**方案并示例代码。

IOS 实现包含在 `PicturePickerImplementation` iOS 项目中的示例代码的类。若要使此类对可见 `DependencyService` 管理器中，必须使用标识的类 [`assembly`] 类型的属性 `Dependency` ，类必须是公共的且显式实现 `IPicturePicker` 接口：

```
[assembly: Dependency (typeof (PicturePickerImplementation))]

namespace DependencyServiceSample.iOS
{
    public class PicturePickerImplementation : IPicturePicker
    {
        TaskCompletionSource<Stream> taskCompletionSource;
        UIImagePickerController imagePicker;

        public Task<Stream> GetImageStreamAsync()
        {
            // Create and define UIImagePickerController
            imagePicker = new UIImagePickerController
            {
                SourceType = UIImagePickerControllerSourceType.PhotoLibrary,
                MediaTypes =
UIImagePickerController.AvailableMediaTypes(UIImagePickerControllerSourceType.PhotoLibrary)
            };

            // Set event handlers
            imagePicker.FinishedPickingMedia += OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled += OnImagePickerCancelled;

            // Present UIImagePickerController;
            UIWindow window = UIApplication.SharedApplication.KeyWindow;
            var viewController = window.RootViewController;
            viewController.PresentModalViewController(imagePicker, true);

            // Return Task object
            taskCompletionSource = new TaskCompletionSource<Stream>();
            return taskCompletionSource.Task;
        }
        ...
    }
}
```

`GetImageStreamAsync` 方法创建 `UIImagePickerController` 和它初始化，以选择照片库中的映像。两个事件处理程序所需：一个用于当用户选择一张照片以及另一个用于当用户取消的照片库显示。 `PresentModalViewController` 然后向用户显示照片库。

在此情况下，`GetImageStreamAsync` 方法必须返回 `Task<Stream>` 对象正在调用它的代码。仅当用户完成与照片库进行交互并调用其中一个事件处理程序时，此任务已完成。为此，类情况下 `TaskCompletionSource` 类非常重要。此类提供 `Task` 要从返回的正确泛型类型的对象 `GetImageStreamAsync` 方法和类可以稍后向发出信号完成此任务。

`FinishedPickingMedia` 时用户所选图片调用事件处理程序。但是，该处理程序提供了 `UIImage` 对象和 `Task` 必须返回.NET `Stream` 对象。这是在两个步骤：`UIImage` 对象必须先转换为存储在内存中的 JPEG 文件 `NSData` 对象，然后 `NSData` 对象转换为.NET `Stream` 对象。调用 `SetResult` 方法 `TaskCompletionSource` 对象，从而完成任务 `Stream` 对象：

```
namespace DependencyServiceSample.iOS
{
    public class PicturePickerImplementation : IPicturePicker
    {
        TaskCompletionSource<Stream> taskCompletionSource;
        UIImagePickerController imagePicker;
        ...
        void OnImagePickerFinishedPickingMedia(object sender, UIImagePickerMediaPickedEventArgs args)
        {
            UIImage image = args.EditedImage ?? args.OriginalImage;

            if (image != null)
            {
                // Convert UIImage to .NET Stream object
                NSData data = image.AsJPEG(1);
                Stream stream = data.AsStream();

                UnregisterEventHandlers();

                // Set the Stream as the completion of the Task
                taskCompletionSource.SetResult(stream);
            }
            else
            {
                UnregisterEventHandlers();
                taskCompletionSource.SetResult(null);
            }
            imagePicker.DismissModalViewController(true);
        }

        void OnImagePickerCancelled(object sender, EventArgs args)
        {
            UnregisterEventHandlers();
            taskCompletionSource.SetResult(null);
            imagePicker.DismissModalViewController(true);
        }

        void UnregisterEventHandlers()
        {
            imagePicker.FinishedPickingMedia -= OnImagePickerFinishedPickingMedia;
            imagePicker.Canceled -= OnImagePickerCancelled;
        }
    }
}
```

IOS 应用程序需要访问照片库从用户的权限。以下内容添加至 `dict` Info.plist 文件的部分：

```
<key>NSPhotoLibraryUsageDescription</key>
<string>Picture Picker uses photo library</string>
```

Android 实现

Android 实现使用中所述的技术选择一个映像方案并示例代码。但是，当用户从图片库选择图像时调用的方法是 `OnActivityResult` 派生的类中重写 `Activity` 。出于此原因，普通 `MainActivity` Android 项目中的类具有补充，使用字段、属性和的重写 `OnActivityResult` 方法：

```
public class MainActivity : FormsAppCompatActivity
{
    ...
    // Field, property, and method for Picture Picker
    public static readonly int PickImageId = 1000;

    public TaskCompletionSource<Stream> PickImageTaskCompletionSource { set; get; }

    protected override void OnActivityResult(int requestCode, Result resultCode, Intent intent)
    {
        base.OnActivityResult(requestCode, resultCode, intent);

        if (requestCode == PickImageId)
        {
            if ((resultCode == Result.Ok) && (intent != null))
            {
                Android.Net.Uri uri = intent.Data;
                Stream stream = ContentResolver.OpenInputStream(uri);

                // Set the Stream as the completion of the Task
                PickImageTaskCompletionSource.SetResult(stream);
            }
            else
            {
                PickImageTaskCompletionSource.SetResult(null);
            }
        }
    }
}
```

`OnActivityResult` 重写指示选定的图片文件与 Android `Uri` 对象，但是可以转换为.NET `Stream` 对象通过调用 `OpenInputStream` 方法 `ContentResolver` 从获取的对象活动的 `ContentResolver` 属性。

Android 实现与 iOS 实现一样使用 `TaskCompletionSource` 任务完成时发出信号通知。这 `TaskCompletionSource` 对象定义中的公共属性作为 `MainActivity` 类。这允许将属性中引用 `PicturePickerImplementation` Android 项目中的类。这是与类 `GetImageStreamAsync` 方法：

```
[assembly: Dependency(typeof(PicturePickerImplementation))]

namespace DependencyServiceSample.Droid
{
    public class PicturePickerImplementation : IPicturePicker
    {
        public Task<Stream> GetImageStreamAsync()
        {
            // Define the Intent for getting images
            Intent intent = new Intent();
            intent.SetType("image/*");
            intent.SetAction(Intent.ActionGetContent);

            // Start the picture-picker activity (resumes in MainActivity.cs)
            MainActivity.Instance.StartActivityForResult(
                Intent.CreateChooser(intent, "Select Picture"),
                MainActivity.PickImageId);

            // Save the TaskCompletionSource object as a MainActivity property
            MainActivity.Instance.PickImageTaskCompletionSource = new TaskCompletionSource<Stream>();

            // Return Task object
            return MainActivity.Instance.PickImageTaskCompletionSource.Task;
        }
    }
}
```

此方法访问 `MainActivity` 类有多种用途：为 `Instance` 属性，对于 `PickImageId` 字段中，为 `TaskCompletionSource` 属性，并调用 `StartActivityForResult` 。此方法由定义 `FormsAppCompatActivity` 类，该类是类的基类的 `MainActivity` 。

UWP 实现

与 iOS 和 Android 的实现中，不同于通用 Windows 平台的照片选取器实现不需要 `TaskCompletionSource` 类。 `PicturePickerImplementation` 类使用 `FileOpenPicker` 类，以获取对照片库的访问。因为 `PickSingleFileAsync` 方法 `FileOpenPicker` 本身是异步的 `GetImageStreamAsync` 方法只需使用 `await` 使用该方法（和其他异步方法），并返回 `Stream` 对象：

```
[assembly: Dependency(typeof(PicturePickerImplementation))]

namespace DependencyServiceSample.UWP
{
    public class PicturePickerImplementation : IPicturePicker
    {
        public async Task<Stream> GetImageStreamAsync()
        {
            // Create and initialize the FileOpenPicker
            FileOpenPicker openPicker = new FileOpenPicker
            {
                ViewMode = PickerViewMode.Thumbnail,
                SuggestedStartLocation = PickerLocationId.PicturesLibrary,
            };

            openPicker.FileTypeFilter.Add(".jpg");
            openPicker.FileTypeFilter.Add(".jpeg");
            openPicker.FileTypeFilter.Add(".png");

            // Get a file and return a Stream
            StorageFile storageFile = await openPicker.PickSingleFileAsync();

            if (storageFile == null)
            {
                return null;
            }

            IRandomAccessStreamWithContentType raStream = await storageFile.OpenReadAsync();
            return raStream.AsStreamForRead();
        }
    }
}
```

在共享代码中实现

现在，已针对每个平台实现该接口，.NET Standard 库中的应用程序可以充分利用它。

`App` 类创建 `Button` 选取照片：

```
Button pickPictureButton = new Button
{
    Text = "Pick Photo",
    VerticalOptions = LayoutOptions.CenterAndExpand,
    HorizontalOptions = LayoutOptions.CenterAndExpand
};
stack.Children.Add(pickPictureButton);
```

`Clicked` 处理程序使用 `DependencyService` 类来调用 `GetImageStreamAsync`。这会导致在平台项目中的调用。如果该方法返回 `Stream` 对象，则处理程序将创建 `Image` 元素以使用该图片 `TapGestureRecognizer`，并替换 `StackLayout` 的页面上 `Image`：

```
pickPictureButton.Clicked += async (sender, e) =>
{
    pickPictureButton.IsEnabled = false;
    Stream stream = await DependencyService.Get<IPicturePicker>().GetImageStreamAsync();

    if (stream != null)
    {
        Image image = new Image
        {
            Source = ImageSource.FromStream(() => stream),
            BackgroundColor = Color.Gray
        };

        TapGestureRecognizer recognizer = new TapGestureRecognizer();
        recognizer.Tapped += (sender2, args) =>
        {
            (MainPage as ContentPage).Content = stack;
            pickPictureButton.IsEnabled = true;
        };
        image.GestureRecognizers.Add(recognizer);

        (MainPage as ContentPage).Content = image;
    }
    else
    {
        pickPictureButton.IsEnabled = true;
    }
};
```

点击 `Image` 元素将页返回到正常。

相关链接

- 从库 (iOS) 中选择一张照片
- 选择一个映像 (Android)
- DependencyService（示例）

Xamarin.Forms 效果

2018/7/13 • • Edit Online

使用目标平台，允许保留每个平台相应的外观和感觉的 Xamarin.Forms 应用程序的本机控件呈现 Xamarin.Forms 的用户界面。效果允许进行自定义，而不必求助于自定义呈现器实现每个平台上的本机控件。

效果简介

效果允许进行自定义，每个平台上的本机控件，通常用于较小的样式更改。本文介绍了影响，概述了影响与自定义呈现器之间的边界，并介绍了 `PlatformEffect` 类。

创建一种效果

效果简化控件的自定义项。本文演示如何创建更改的背景色的效果 `Entry` 控制当控件获得焦点。

将参数传递给效果

创建通过参数配置的效果使要重复使用的效果。这些文章演示了如何使用属性将参数传递给某个效果，请和更改在运行时参数。

调用影响来自事件

效果可以调用的事件。本文介绍如何创建实现低级别多点触控手指跟踪并向发出信号的触摸屏输入按下、移动和版本的应用程序的事件。

效果简介

2018/7/13 • • Edit Online

效果允许进行自定义，每个平台上的本机控件，通常用于较小的样式更改。本文介绍了影响，概述了影响与自定义呈现器之间的边界并描述 PlatformEffect 类。

Xamarin.Forms页面、布局和控件提供常见的 API，用于描述跨平台移动用户界面。每个页面、布局和控件以不同的方式使用呈现在每个平台 `Renderer` 类，该类又创建本机控件（对应于 Xamarin.Forms 表示形式），对其进行排列在屏幕上，并将添加在共享指定的行为代码。

开发人员可以实现自定义 `Renderer` 类，以自定义控件的外观和/或行为。但是，实现执行简单控件自定义的自定义呈现器类通常是重型的响应。效果简化此过程中，允许更轻松地自定义每个平台上的本机控件。

特定于平台的项目中创建通过子类化效果 `PlatformEffect` 由控件，然后效果将其附加到 Xamarin.Forms.NET 标准库或共享库项目中相应的控件。

为什么要使用通过自定义呈现器的影响？

效果简化控件的自定义、可重用，和可以进行参数化以重用，从而进一步提升。

也可以使用自定义呈现器可以实现具有某种效果的任何内容。但是，自定义呈现器提供更多灵活性和自定义效果比。以下指导原则列出在其中选择效果，而不是自定义呈现器的情况：

- 更改特定于平台的控件的属性将会获得所需的结果时，建议使用一种效果。
- 需要重写特定于平台的方法时，自定义呈现器是控件的必需的。
- 需要替换实现的 Xamarin.Forms 控件的特定于平台的控件时需要自定义呈现器。

子类化 PlatformEffect 类

下表列出的命名空间 `PlatformEffect` 上每个平台和其属性的类型的类：

平台	命名空间	容器	控件
iOS	Xamarin.Forms.Platform.iOS	UIView	UIView
Android	Xamarin.Forms.Platform.Android	视图分组	视图
通用 Windows 平台 (UWP)	Xamarin.Forms.Platform.UWP	FrameworkElement	FrameworkElement

每个特定于平台的 `PlatformEffect` 类公开以下属性：

- `Container` – 引用被用来实现布局的特定于平台的控件。
- `Control` – 引用被用来实现的 Xamarin.Forms 控件的特定于平台的控件。
- `Element` – 引用 Xamarin.Forms 控件要呈现。

效果没有容器、控件或元素与其所附加到，因为它们可以附加到任何元素的类型信息。因此，影响附加到的元素，它不支持时它应适当降级或引发异常。但是，`Container`，`Control`，和 `Element` 属性可以强制转换为其实现的类型。有关类型的这些详细信息，请参阅呈现器基类和本机控件。

每个特定于平台的 `PlatformEffect` 类公开以下方法，必须重写以实现一种效果：

- `OnAttached` – 当效果附加到 Xamarin.Forms 控件时调用。重写的版本中的每个特定于平台的效果类，此方法是执行自定义控件，以及异常处理以防效果不能应用于指定的 Xamarin.Forms 控件的位置。
- `OnDetached` – 当效果与 Xamarin.Forms 控件时调用。重写的版本中的每个特定于平台的效果类，此方法是执行任何效果清理，例如取消注册事件处理程序的位置。

此外，`PlatformEffect` 公开 `OnElementPropertyChanged` 方法，还可以重写该方法。当该元素的属性发生更改时，调用此方法。重写的版本中的每个特定于平台的效果类，此方法是对 Xamarin.Forms 控件上的可绑定的属性更改进行响应的位置。始终应进行检查更改的属性，如可以多次调用此重写。

相关链接

- 自定义呈现器

创建一种效果

2018/7/13 • • Edit Online

效果简化控件的自定义项。本文演示如何创建更改项控件的背景色，当控件获得焦点时的效果。

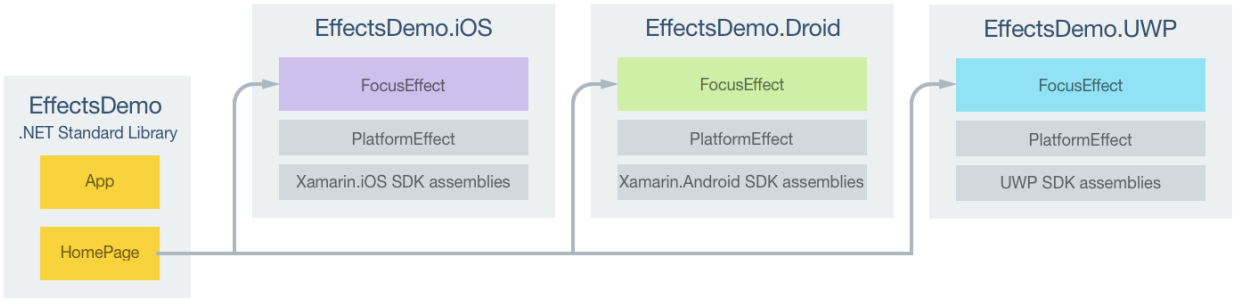
每个特定于平台的项目中创建有效的过程如下所示：

1. 创建一个子类 `PlatformEffect` 类。
2. 重写 `OnAttached` 方法和写入自定义控件的逻辑。
3. 重写 `OnDetached` 清理控件自定义，如果所需的方法和写入逻辑。
4. 添加 `ResolutionGroupName` 影响类的属性。此属性设置的效果，阻止与其他具有相同名称的效果的碰撞公司宽命名空间。请注意，此属性只能应用一次每个项目。
5. 添加 `ExportEffect` 影响类的属性。此属性用于通过 Xamarin.Forms，组名称，以及查找然后将其应用到控件的影响的唯一 ID 注册效果。该属性采用两个参数 – 效果，并将用于定位的效果，然后将其应用到控件的唯一字符串的类型名称。

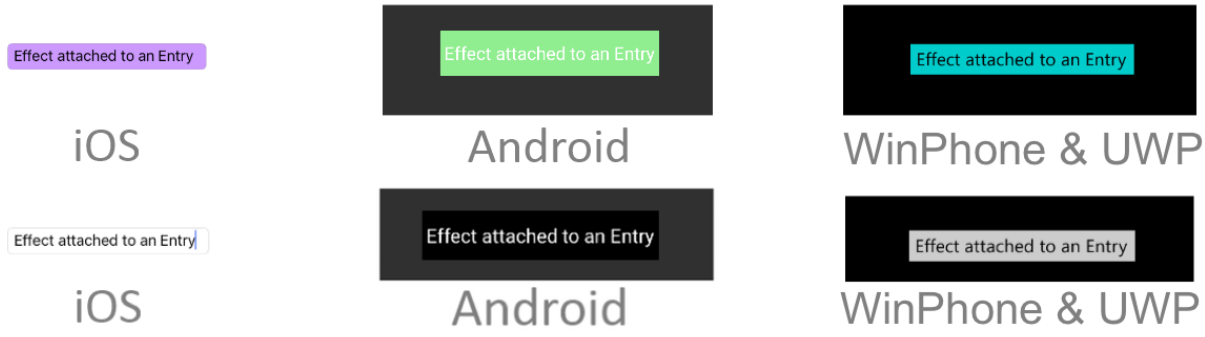
将其附加到相应的控件，然后可以使用效果。

> **NOTE**
> 它是可选提供每个平台项目中的效果。尝试使用一种效果，其中一个不注册时将返回一个非 null 值，不执行任何操作。

该示例应用程序演示 `FocusEffect` 时获得焦点，更改控件的背景色。下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



`Entry` 对的 control 权限 `HomePage` 由自定义 `FocusEffect` 每个特定于平台的项目中的类。每个 `FocusEffect` 类派生自 `PlatformEffect` 为每个平台的类。这会导致 `Entry` 控制呈现具有特定于平台的背景色，这会更改时控件获得焦点，如以下屏幕截图中所示：



创建每个平台上的效果

以下各节讨论的特定于平台的实现 `FocusEffect` 类。

iOS 项目

下面的代码示例演示 `FocusEffect` 实现针对 iOS 项目：

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;

[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.iOS
{
    public class FocusEffect : PlatformEffect
    {
        UIColor backgroundColor;

        protected override void OnAttached ()
        {
            try {
                Control.BackgroundColor = backgroundColor = UIColor.FromRGB (204, 153, 255);
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }

        protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged (args);

            try {
                if (args.PropertyName == "IsFocused") {
                    if (Control.BackgroundColor == backgroundColor) {
                        Control.BackgroundColor = UIColor.White;
                    } else {
                        Control.BackgroundColor = backgroundColor;
                    }
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}
```

`OnAttached` 方法设置 `BackgroundColor` 淡紫色与该控件的属性 `UIColor.FromRGB` 方法，并将此颜色也存储在字段中。此功能包装在 `try` / `catch` 阻止以防效果附加到的控件不具有 `BackgroundColor` 属性。通过提供任何实现 `OnDetached` 方法因为任何清理不不需要。

`OnElementPropertyChanged` 替代响应的 Xamarin.Forms 控件上的可绑定的属性更改。当 `IsFocused` 属性更改 `BackgroundColor` 如果控件有焦点的控件属性更改为白色，否则它将更改为淡紫色。此功能包装在 `try` / `catch` 阻止以防效果附加到的控件不具有 `BackgroundColor` 属性。

Android 项目

下面的代码示例演示 `FocusEffect` 实现针对 Android 项目：

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.Android;

[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.Droid
{
    public class FocusEffect : PlatformEffect
    {
        Android.Graphics.Color backgroundColor;

        protected override void OnAttached ()
        {
            try {
                backgroundColor = Android.Graphics.Color.LightGreen;
                Control.SetBackgroundColor (backgroundColor);

            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }

        protected override void OnElementPropertyChanged (System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged (args);
            try {
                if (args.PropertyName == "IsFocused") {
                    if (((Android.Graphics.Drawables.ColorDrawable)Control.Background).Color ==
backgroundColor) {
                        Control.SetBackgroundColor (Android.Graphics.Color.Black);
                    } else {
                        Control.SetBackgroundColor (backgroundColor);
                    }
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}
```

`OnAttached` 方法调用 `SetBackgroundColor` 方法以设置要打造出色的控件的背景色绿色，并且还在字段中存储此颜色。此功能包装在 `try` / `catch` 阻止以防效果附加到的控件不具有 `SetBackgroundColor` 属性。通过提供任何实现 `OnDetached` 方法因为任何清理不不需要。

`OnElementPropertyChanged` 替代响应的 Xamarin.Forms 控件上的可绑定的属性更改。当 `IsFocused` 属性更改控件的背景色更改为白色，如果控件有焦点，否则它将更改为浅绿色。此功能包装在 `try` / `catch` 阻止以防效果附加到的控件不具有 `BackgroundColor` 属性。

通用 Windows 平台项目

下面的代码示例演示 `FocusEffect` 实现通用 Windows 平台 (UWP) 项目：

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.UWP;

[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(FocusEffect), "FocusEffect")]
namespace EffectsDemo.UWP
{
    public class FocusEffect : PlatformEffect
    {
        protected override void OnAttached()
        {
            try
            {
                (Control as Windows.UI.Xaml.Controls.Control).Background = new SolidColorBrush(Colors.Cyan);
                (Control as FormsTextBox).BackgroundFocusBrush = new SolidColorBrush(Colors.White);
            }
            catch (Exception ex)
            {
                Debug.WriteLine("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached()
        {
        }
    }
}
```

`OnAttached`方法设置`Background`蓝绿色，并设置该控件的属性`BackgroundFocusBrush`属性设置为白色。此功能包装在`try` / `catch`阻止以防效果附加到控件缺少这些属性。通过提供任何实现`OnDetached`方法因为任何清理不不需要。

使用效果

使用从 Xamarin.Forms.NET Standard 库或共享库项目的效果的过程如下所示：

1. 声明将由效果自定义控件。
2. 通过将其添加到控件的附加到该控件的效果 `Effects` 集合。

> **NOTE**
> 效果实例只能附加到单个控件。因此，必须解决影响两次以在两个控件中使用它。

使用 XAML 中的效果

以下 XAML 代码示例所示 `Entry` 控件`FocusEffect`附加：

```
<Entry Text="Effect attached to an Entry" ...>
    <Entry.Effects>
        <local:FocusEffect />
    </Entry.Effects>
    ...
</Entry>
```

`FocusEffect` .NET Standard 库中的类支持中 XAML，效果消耗，并在下面的代码示例所示：

```
public class FocusEffect : RoutingEffect
{
    public FocusEffect () : base ("MyCompany.FocusEffect")
    {
    }
}
```

`FocusEffect`类子类 `RoutingEffect` 类，该类表示包装内部效果通常特定于平台的一个独立于平台的效果。`FocusEffect`类会调用基类构造函数，传入参数包含的串联的解析组名称 (使用指定 `ResolutionGroupName` 效果类中的属性)，和的唯一 ID使用指定 `ExportEffect` 效果类中的属性。因此，当 `Entry` 在运行时的新实例初始化 `MyCompany.FocusEffect` 添加到控件的 `Effects` 集合。

效果可以还附加到控件通过使用一种行为，或通过使用附加属性。有关使用行为附加到控件影响的详细信息，请参阅可重用 EffectBehavior。有关使用附加的属性将影响附加到控件的详细信息，请参阅效果传递参数。

使用 C 中的效果#

等效于 `Entry` C# 中所示下面的代码示例：

```
var entry = new Entry {
  Text = "Effect attached to an Entry",
  ...
};
```

`FocusEffect`附加到 `Entry`通过将效果添加到控件的实例 `Effects` 集合，如下面的代码示例中所示：

```
public HomePageCS ()
{
  ...
  entry.Effects.Add (Effect.Resolve ("MyCompany.FocusEffect"));
  ...
}
```

`Effect.Resolve` 返回 `Effect` 对于指定的名称，这是解析组名称的串联 (使用指定 `ResolutionGroupName` 效果类中的属性)，并使用指定的唯一 ID `ExportEffect` 效果类中的属性。如果一个平台不提供效果 `Effect.Resolve` 方法将返回一个非 `null` 值。

总结

本文演示了如何创建更改的背景色的效果 `Entry` 控制当控件获得焦点。

相关链接

- 自定义呈现器
- Effect
- PlatformEffect
- 背景颜色效果（示例）
- 焦点效果（示例）

将参数传递给效果

2018/6/22 • • Edit Online

可通过属性，启用可重复使用的效果定义影响参数。通过实例化效果时指定每个属性的值之前，然后可以将参数传递到效果。

传递效果参数作为公共语言运行时属性

公共语言运行时 (CLR) 属性可用来定义不响应运行时属性更改的效果参数。本文演示如何使用 CLR 属性将参数传递给效果。

传递效果参数作为附加的属性

附加的属性可用来定义响应运行时属性更改的效果参数。本文演示如何使用附加的属性将参数传递给起作用，以及更改在运行时参数。

公共语言运行时属性作为参数传递的效果

2018/7/13 • • Edit Online

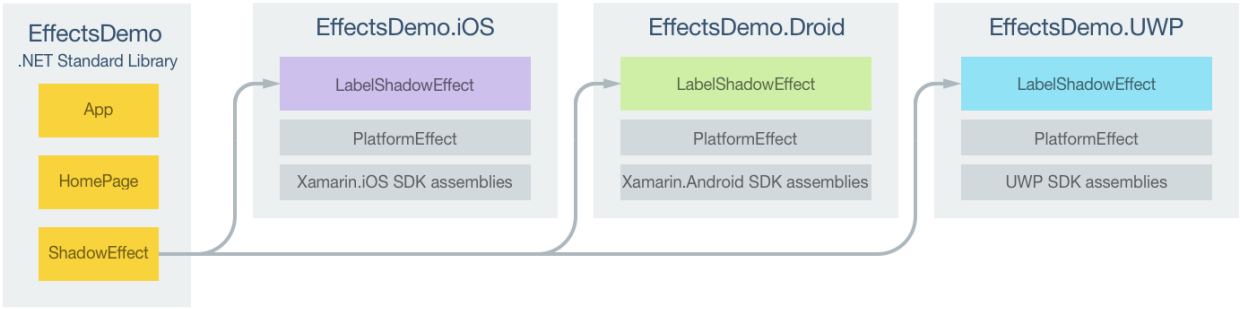
公共语言运行时 (CLR) 属性可以用于定义不响应运行时属性更改的效果参数。本文演示如何使用 CLR 属性将参数传递给效果。

创建不响应运行时属性更改的效果参数的过程如下所示：

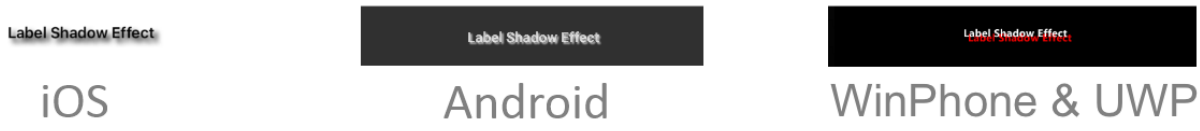
1. 创建 `public` 类的子类 `RoutingEffect` 类。 `RoutingEffect` 类表示包装内部效果通常特定于平台的一个独立于平台的效果。
2. 创建调用基类构造函数，传递解析组名称，并已在每个特定于平台的效果类中指定的唯一 ID 的串联中的构造函数。
3. 将属性添加到每个参数传递的效果的类。

通过实例化效果时指定的每个属性的值之前，然后可以是参数传递到效果。

该示例应用程序演示 `ShadowEffect`，它将到显示的文本阴影 `Label` 控件。下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



一个 `Label` 对的 control 权限 `HomePage` 由自定义 `LabelShadowEffect` 每个特定于平台的项目中。参数传递给每个 `LabelShadowEffect` 中的属性通过 `ShadowEffect` 类。每个 `LabelShadowEffect` 类派生自 `PlatformEffect` 为每个平台的类。这会导致要添加到显示的文本阴影 `Label` 控制，如以下屏幕截图中所示：



创建效果参数

一个 `public` 类的子类 `RoutingEffect` 类应创建表示效果参数，如下面的代码示例中所示：

```
public class ShadowEffect : RoutingEffect
{
  public float Radius { get; set; }

  public Color Color { get; set; }

  public float DistanceX { get; set; }

  public float DistanceY { get; set; }

  public ShadowEffect () : base ("MyCompany.LabelShadowEffect")
  {
  }
}
```

`ShadowEffect`包含四个属性表示参数传递到每个特定于平台的`LabelShadowEffect`。类构造函数调用基类构造函数，传入参数包含解析组名称，并已在每个特定于平台的效果类中指定的唯一 ID 的串联。因此的新实例`MyCompany.LabelShadowEffect`将添加到控件的`Effects`集合时`ShadowEffect`实例化。

使用效果

以下 XAML 代码示例所示`Label`控件`ShadowEffect`附加：

```
<Label Text="Label Shadow Effect" ...>
  <Label.Effects>
    <local:ShadowEffect Radius="5" DistanceX="5" DistanceY="5">
      <local:ShadowEffect.Color>
        <OnPlatform x:TypeArguments="Color">
            <On Platform="iOS" Value="Black" />
            <On Platform="Android" Value="White" />
            <On Platform="UWP" Value="Red" />
        </OnPlatform>
      </local:ShadowEffect.Color>
    </local:ShadowEffect>
  </Label.Effects>
</Label>
```

等效于`Label` C# 中所示下面的代码示例：

```
var label = new Label {
  Text = "Label Shadow Effect",
  ...
};

Color color = Color.Default;
switch (Device.RuntimePlatform)
{
    case Device.iOS:
        color = Color.Black;
        break;
    case Device.Android:
        color = Color.White;
        break;
    case Device.UWP:
        color = Color.Red;
        break;
}

label.Effects.Add (new ShadowEffect {
  Radius = 5,
  Color = color,
  DistanceX = 5,
  DistanceY = 5
});
```

在这两个代码示例中，实例 `ShadowEffect` 类实例化之前要添加到控件的每个属性指定的值与 `Effects` 集合。请注意，`ShadowEffect.Color` 属性使用特定于平台的颜色值。有关详细信息，请参阅设备类。

创建每个平台上的效果

以下各节讨论的特定于平台的实现 `LabelShadowEffect` 类。

iOS 项目

下面的代码示例演示 `LabelShadowEffect` 实现针对 iOS 项目：

```
[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                if (effect != null) {
                    Control.Layer.CornerRadius = effect.Radius;
                    Control.Layer.ShadowColor = effect.Color.ToCGColor ();
                    Control.Layer.ShadowOffset = new CGSize (effect.DistanceX, effect.DistanceY);
                    Control.Layer.ShadowOpacity = 1.0f;
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}
```

`OnAttached` 方法检索 `ShadowEffect` 实例，并设置 `Control.Layer` 属性设置为指定的属性值以创建阴影。此功能包装在 `try` / `catch` 阻止以防效果附加到该控件不具有 `Control.Layer` 属性。通过提供任何实现 `OnDetached` 方法因为任何清理不不需要。

Android 项目

下面的代码示例演示 `LabelShadowEffect` 实现针对 Android 项目：

```
[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.Droid
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                var control = Control as Android.Widget.TextView;
                var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                if (effect != null) {
                    float radius = effect.Radius;
                    float distanceX = effect.DistanceX;
                    float distanceY = effect.DistanceY;
                    Android.Graphics.Color color = effect.Color.ToAndroid ();
                    control.SetShadowLayer (radius, distanceX, distanceY, color);
                }
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}
```

`OnAttached` 方法检索 `ShadowEffect` 实例，并调用 `TextView.SetShadowLayer` 方法来创建一个使用指定的属性值的阴影。此功能包装在 `try` / `catch` 阻止以防效果附加到该控件不具有 `Control.Layer` 属性。通过提供任何实现 `OnDetached` 方法因为任何清理不不需要。

通用 Windows 平台项目

下面的代码示例演示 `LabelShadowEffect` 实现通用 Windows 平台 (UWP) 项目：

```
[assembly: ResolutionGroupName ("Xamarin")]
[assembly: ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.UWP
{
    public class LabelShadowEffect : PlatformEffect
    {
        bool shadowAdded = false;

        protected override void OnAttached ()
        {
            try {
                if (!shadowAdded) {
                    var effect = (ShadowEffect)Element.Effects.FirstOrDefault (e => e is ShadowEffect);
                    if (effect != null) {
                        var textBlock = Control as Windows.UI.Xaml.Controls.TextBlock;
                        var shadowLabel = new Label ();
                        shadowLabel.Text = textBlock.Text;
                        shadowLabel.FontAttributes = FontAttributes.Bold;
                        shadowLabel.HorizontalOptions = LayoutOptions.Center;
                        shadowLabel.VerticalOptions = LayoutOptions.CenterAndExpand;
                        shadowLabel.TextColor = effect.Color;
                        shadowLabel.TranslationX = effect.DistanceX;
                        shadowLabel.TranslationY = effect.DistanceY;

                        ((Grid)Element.Parent).Children.Insert (0, shadowLabel);
                        shadowAdded = true;
                    }
                }
            } catch (Exception ex) {
                Debug.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
    }
}
```

通用 Windows 平台不提供阴影效果，因此 `LabelShadowEffect` 两个平台上的实现模拟一个通过添加第二个偏移量 `Label` 落后于主要 `Label` 。 `OnAttached` 方法检索 `ShadowEffect` 实例，创建新 `Label` ，并在设置某些布局属性 `Label` 。然后通过设置，创建卷影 `TextColor` ， `TranslationX` ，以及 `TranslationY` 属性控制的颜色和位置 `Label` . `shadowLabel` 然后插入偏移量落后于主要 `Label` 。此功能包装在 `try` / `catch` 阻止以防效果附加到该控件不具有 `Control.Layer` 属性。通过提供任何实现 `OnDetached` 方法因为任何清理不不需要。

总结

本文演示了使用 CLR 属性将参数传递给效果。可以使用 CLR 属性定义不响应运行时属性更改的效果参数。

相关链接

- 自定义呈现器
- Effect
- PlatformEffect
- RoutingEffect
- 阴影效果（示例）

作为附加属性的效果参数传递

2018/7/13 • • Edit Online

附加的属性可用于定义响应的运行时属性更改的效果参数。本文演示如何使用附加属性，以将参数传递给某个效果，请和更改在运行时参数。

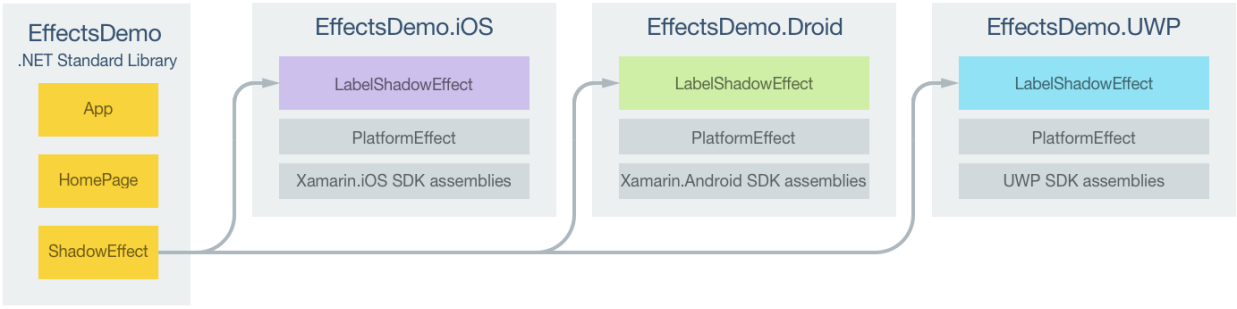
创建响应的运行时属性更改的效果参数的过程如下所示：

1. 创建 `static` 类，该类包含要传递给影响每个参数的附加的属性。
2. 将其他附加的属性添加到将用于控制添加或删除效果类将附加到的控件的类。请确保此附加属性寄存器 `propertyChanged` 属性的值发生更改时将执行的委托。
3. 创建 `static` getter 和 setter 为每个附加属性。
4. 实现中的逻辑 `propertyChanged` 委托来添加和删除效果。
5. 实现内部的嵌套的类 `static` 类，名为后的效果，其中子类 `RoutingEffect` 类。对于构造函数中，调用基类构造函数，传入解析组名称，并已在每个特定于平台的效果类中指定的唯一 ID 的串联。

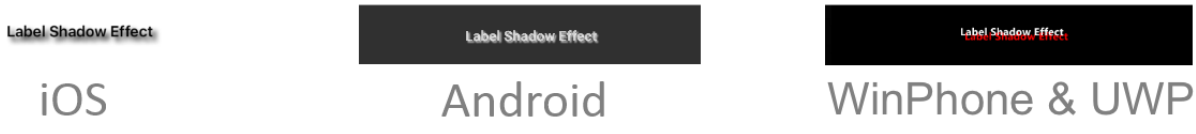
通过将附加的属性和属性值添加到相应的控件之前，然后可以将参数传递到效果。此外，还可以指定新的附加的属性值在运行时更改参数。

> **NOTE**
>
> 附加的属性是一种特殊的可绑定属性，但附加到其他对象，并可识别在 XAML 中的一个类中定义为包含的类和一个句点分隔的属性名称的属性。有关详细信息，请参阅附加属性。

该示例应用程序演示 `ShadowEffect`，它将到显示的文本阴影 `Label` 控件。此外，可以在运行时更改阴影的颜色。下图演示了示例应用程序，以及它们之间的关系的每个项目的职责：



一个 `Label` 对的 control 权限 `HomePage` 由自定义 `LabelShadowEffect` 每个特定于平台的项目中。参数传递给每个 `LabelShadowEffect` 通过中的附加属性 `ShadowEffect` 类。每个 `LabelShadowEffect` 类派生自 `PlatformEffect` 为每个平台的类。这会导致要添加到显示的文本阴影 `Label` 控制，如以下屏幕截图中所示：



创建效果参数

一个 `static` 类应创建表示效果参数，如下面的代码示例中所示：

```
public static class ShadowEffect
{
  public static readonly BindableProperty HasShadowProperty =
    BindableProperty.CreateAttached ("HasShadow", typeof(bool), typeof(ShadowEffect), false, propertyChanged:
OnHasShadowChanged);
  public static readonly BindableProperty ColorProperty =
    BindableProperty.CreateAttached ("Color", typeof(Color), typeof(ShadowEffect), Color.Default);
  public static readonly BindableProperty RadiusProperty =
    BindableProperty.CreateAttached ("Radius", typeof(double), typeof(ShadowEffect), 1.0);
  public static readonly BindableProperty DistanceXProperty =
    BindableProperty.CreateAttached ("DistanceX", typeof(double), typeof(ShadowEffect), 0.0);
  public static readonly BindableProperty DistanceYProperty =
    BindableProperty.CreateAttached ("DistanceY", typeof(double), typeof(ShadowEffect), 0.0);

  public static bool GetHasShadow (BindableObject view)
  {
    return (bool)view.GetValue (HasShadowProperty);
  }

  public static void SetHasShadow (BindableObject view, bool value)
  {
    view.SetValue (HasShadowProperty, value);
  }
  ...

  static void OnHasShadowChanged (BindableObject bindable, object oldValue, object newValue)
  {
    var view = bindable as View;
    if (view == null) {
      return;
    }

    bool hasShadow = (bool)newValue;
    if (hasShadow) {
      view.Effects.Add (new LabelShadowEffect ());
    } else {
      var toRemove = view.Effects.FirstOrDefault (e => e is LabelShadowEffect);
      if (toRemove != null) {
        view.Effects.Remove (toRemove);
      }
    }
  }

  class LabelShadowEffect : RoutingEffect
  {
    public LabelShadowEffect () : base ("MyCompany.LabelShadowEffect")
    {
    }
  }
}
```

`ShadowEffect`包含五个附加的属性，则使用`static` getter 和 setter 为每个附加属性。这些属性的四个表示参数传递给每个特定于平台的`LabelShadowEffect`。`ShadowEffect`类还定义了`HasShadow`附加属性，用于控制添加或删除到该控件的效果，`ShadowEffect`类附加到。此附加属性寄存器`OnHasShadowChanged`属性的值发生更改时将执行的方法。此方法添加或删除值的基础的效果`HasShadow`附加属性。

嵌套`LabelShadowEffect`类，其中子类`RoutingEffect`类，支持效果添加和删除。`RoutingEffect`类表示包装内部效果通常特定于平台的一个独立于平台的效果。这简化了效果删除过程，因为没有编译时访问类型信息特定于平台的效果。`LabelShadowEffect`构造函数调用基类构造函数，传入参数包含解析组名称，并已在每个特定于平台的效果类中指定的唯一 ID 的串联。这样效果添加和删除在`OnHasShadowChanged`方法，按如下所示：

- **影响加法**– 的新实例`LabelShadowEffect`添加到控件的`Effects`集合。这将替换使用`Effect.Resolve`方法来添加效果。

- **影响删除**– 的第一个实例 `LabelShadowEffect` 在该控件的 `Effects` 检索集合并将其删除。

使用效果

每个平台专属 `LabelShadowEffect` 可供添加到的附加的属性 `Label` 控制，如以下 XAML 代码示例所示：

```
<Label Text="Label Shadow Effect" ...
       local:ShadowEffect.HasShadow="true" local:ShadowEffect.Radius="5"
       local:ShadowEffect.DistanceX="5" local:ShadowEffect.DistanceY="5">
  <local:ShadowEffect.Color>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="Black" />
        <On Platform="Android" Value="White" />
        <On Platform="UWP" Value="Red" />
    </OnPlatform>
  </local:ShadowEffect.Color>
</Label>
```

等效于 `Label` C# 中所示下面的代码示例：

```
var label = new Label {
  Text = "Label Shadow Effect",
  ...
};

Color color = Color.Default;
switch (Device.RuntimePlatform)
{
    case Device.iOS:
        color = Color.Black;
        break;
    case Device.Android:
        color = Color.White;
        break;
    case Device.UWP:
        color = Color.Red;
        break;
}

ShadowEffect.SetHasShadow (label, true);
ShadowEffect.SetRadius (label, 5);
ShadowEffect.SetDistanceX (label, 5);
ShadowEffect.SetDistanceY (label, 5);
ShadowEffect.SetColor (label, color));
```

设置 `ShadowEffect.HasShadow` 附加属性设置为 `true` 执行 `ShadowEffect.OnHasShadowChanged` 方法添加或删除 `LabelShadowEffect` 到 `Label` 控件。在这两个代码示例中，`ShadowEffect.Color` 附加的属性提供特定于平台的颜色值。有关详细信息，请参阅设备类。

此外，`Button` 允许要在运行时更改的阴影颜色。当 `Button` 单击时，以下代码更改通过设置颜色阴影 `ShadowEffect.Color` 附加属性：

```
ShadowEffect.SetColor (label, Color.Teal);
```

使用样式的效果

一种样式也可以使用可供添加到控件的附加的属性的效果。以下 XAML 代码示例所示*显式*阴影效果，可以应用于样式 `Label` 控件：

```
<Style x:Key="ShadowEffectStyle" TargetType="Label">
  <Style.Setters>
    <Setter Property="local:ShadowEffect.HasShadow" Value="True" />
    <Setter Property="local:ShadowEffect.Radius" Value="5" />
    <Setter Property="local:ShadowEffect.DistanceX" Value="5" />
    <Setter Property="local:ShadowEffect.DistanceY" Value="5" />
  </Style.Setters>
</Style>
```

`Style` 可应用于 `Label` 通过设置其 `Style` 属性设置为 `Style` 实例使用 `StaticResource` 标记扩展，如下面的代码示例中所示：

```
<Label Text="Label Shadow Effect" ... Style="{StaticResource ShadowEffectStyle}" />
```

有关样式的详细信息，请参阅样式。

创建每个平台上的效果

以下各节讨论的特定于平台的实现 `LabelShadowEffect` 类。

iOS 项目

下面的代码示例演示 `LabelShadowEffect` 实现针对 iOS 项目：

```
[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached ()
        {
            try {
                UpdateRadius ();
                UpdateColor ();
                UpdateOffset ();
                Control.Layer.ShadowOpacity = 1.0f;
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateRadius ()
        {
            Control.Layer.CornerRadius = (nfloat)ShadowEffect.GetRadius (Element);
        }

        void UpdateColor ()
        {
            Control.Layer.ShadowColor = ShadowEffect.GetColor (Element).ToCGColor ();
        }

        void UpdateOffset ()
        {
            Control.Layer.ShadowOffset = new CGSize (
                (double)ShadowEffect.GetDistanceX (Element),
                (double)ShadowEffect.GetDistanceY (Element));
        }
    }
```

`OnAttached`方法调用的方法，检索附加的属性的值使用`ShadowEffect` getter，并将其设置`Control.Layer`用于创建阴影的属性值的属性。此功能包装在`try` / `catch`阻止以防效果附加到该控件不具有`Control.Layer`属性。通过提供任何实现`OnDetached`方法因为任何清理不不需要。

响应属性更改

如果任一`ShadowEffect`附加在运行时，需要通过显示所做的更改来响应该效果的属性值更改。重写的版本`OnElementPropertyChanged`方法，在特定于平台的效果类中，是对可绑定的属性更改进行响应的位置，如下面的代码示例中所示：

```
public class LabelShadowEffect : PlatformEffect
{
  ...
  protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
  {
    if (args.PropertyName == ShadowEffect.RadiusProperty.PropertyName) {
      UpdateRadius ();
    } else if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
      UpdateColor ();
    } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
               args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
      UpdateOffset ();
    }
  }
  ...
}
```

`OnElementPropertyChanged` 方法更新 radius、颜色或阴影的偏移量提供的相应 `ShadowEffect` 附加的属性值已更改。始终应进行检查更改的属性，如可以多次调用此重写。

Android 项目

下面的代码示例演示 `LabelShadowEffect` 实现针对 Android 项目：

```
[assembly:ResolutionGroupName ("MyCompany")]
[assembly:ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.Droid
{
    public class LabelShadowEffect : PlatformEffect
    {
        Android.Widget.TextView control;
        Android.Graphics.Color color;
        float radius, distanceX, distanceY;

        protected override void OnAttached ()
        {
            try {
                control = Control as Android.Widget.TextView;
                UpdateRadius ();
                UpdateColor ();
                UpdateOffset ();
                UpdateControl ();
            } catch (Exception ex) {
                Console.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateControl ()
        {
            if (control != null) {
                control.SetShadowLayer (radius, distanceX, distanceY, color);
            }
        }

        void UpdateRadius ()
        {
            radius = (float)ShadowEffect.GetRadius (Element);
        }

        void UpdateColor ()
        {
            color = ShadowEffect.GetColor (Element).ToAndroid ();
        }

        void UpdateOffset ()
        {
            distanceX = (float)ShadowEffect.GetDistanceX (Element);
            distanceY = (float)ShadowEffect.GetDistanceY (Element);
        }
    }
```

`OnAttached`方法调用的方法，检索附加的属性的值使用`ShadowEffect` getter，并调用一个方法，调用`TextView.SetShadowLayer`方法来创建阴影使用属性值。此功能包装在`try` / `catch`阻止以防效果附加到该控件不具有`Control.Layer`属性。通过提供任何实现`OnDetached`方法因为任何清理不不需要。

响应属性更改

如果任一`ShadowEffect`附加在运行时，需要通过显示所做的更改来响应该效果的属性值更改。重写的版本`OnElementPropertyChanged`方法，在特定于平台的效果类中，是对可绑定的属性更改进行响应的位置，如下面的代码示例中所示：

```
public class LabelShadowEffect : PlatformEffect
{
  ...
  protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
  {
    if (args.PropertyName == ShadowEffect.RadiusProperty.PropertyName) {
      UpdateRadius ();
      UpdateControl ();
    } else if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
      UpdateColor ();
      UpdateControl ();
    } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
               args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
      UpdateOffset ();
      UpdateControl ();
    }
  }
  ...
}
```

`OnElementPropertyChanged` 方法更新 radius、颜色或阴影的偏移量提供的相应 `ShadowEffect` 附加的属性值已更改。始终应进行检查更改的属性，如可以多次调用此重写。

通用 Windows 平台项目

下面的代码示例演示 `LabelShadowEffect` 实现通用 Windows 平台 (UWP) 项目：

```
[assembly: ResolutionGroupName ("MyCompany")]
[assembly: ExportEffect (typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace EffectsDemo.UWP
{
    public class LabelShadowEffect : PlatformEffect
    {
        Label shadowLabel;
        bool shadowAdded = false;

        protected override void OnAttached ()
        {
            try {
                if (!shadowAdded) {
                    var textBlock = Control as Windows.UI.Xaml.Controls.TextBlock;

                    shadowLabel = new Label ();
                    shadowLabel.Text = textBlock.Text;
                    shadowLabel.FontAttributes = FontAttributes.Bold;
                    shadowLabel.HorizontalOptions = LayoutOptions.Center;
                    shadowLabel.VerticalOptions = LayoutOptions.CenterAndExpand;

                    UpdateColor ();
                    UpdateOffset ();

                    ((Grid)Element.Parent).Children.Insert (0, shadowLabel);
                    shadowAdded = true;
                }
            } catch (Exception ex) {
                Debug.WriteLine ("Cannot set property on attached control. Error: ", ex.Message);
            }
        }

        protected override void OnDetached ()
        {
        }
        ...

        void UpdateColor ()
        {
            shadowLabel.TextColor = ShadowEffect.GetColor (Element);
        }

        void UpdateOffset ()
        {
            shadowLabel.TranslationX = ShadowEffect.GetDistanceX (Element);
            shadowLabel.TranslationY = ShadowEffect.GetDistanceY (Element);
        }
    }
}
```

通用 Windows 平台不提供阴影效果，因此`LabelShadowEffect`两个平台上的实现模拟一个通过添加第二个偏移量`Label`落后于主要`Label`。`OnAttached`方法创建的新`Label`，并对设置某些布局属性`Label`。然后，它调用检索使用的附加的属性值的方法`ShadowEffect` getter，并通过设置创建卷影`TextColor`，`TranslationX`，并且`TranslationY`属性来控制的颜色和位置`Label`。`shadowLabel`然后插入偏移量落后于主要`Label`。此功能包装在`try` / `catch`阻止以防效果附加到该控件不具有`Control.Layer`属性。通过提供任何实现`OnDetached`方法因为任何清理不不需要。

响应属性更改

如果任一`ShadowEffect`附加在运行时，需要通过显示所做的更改来响应该效果的属性值更改。重写的版本`OnElementPropertyChanged`方法，在特定于平台的效果类中，是对可绑定的属性更改进行响应的位置，如下面的代码示例中所示：

```
public class LabelShadowEffect : PlatformEffect
{
  ...
  protected override void OnElementPropertyChanged (PropertyChangedEventArgs args)
  {
    if (args.PropertyName == ShadowEffect.ColorProperty.PropertyName) {
      UpdateColor ();
    } else if (args.PropertyName == ShadowEffect.DistanceXProperty.PropertyName ||
                    args.PropertyName == ShadowEffect.DistanceYProperty.PropertyName) {
      UpdateOffset ();
    }
  }
  ...
}
```

`OnElementPropertyChanged` 方法将更新的颜色或阴影的偏移量提供的相应 `ShadowEffect` 附加的属性值已更改。始终应进行检查更改的属性，如可以多次调用此重写。

总结

本文演示了使用附加属性，以将参数传递给某个效果，请和更改在运行时参数。附加的属性可用于定义响应的运行时属性更改的效果参数。

相关链接

- 自定义呈现器
- Effect
- PlatformEffect
- RoutingEffect
- 阴影效果（示例）

调用效果中的事件

2018/11/13 • • Edit Online

一种效果可以定义和调用的事件，信号基础本机视图中的更改。本文介绍如何实现低级别多点触控手指跟踪，以及如何生成信号触摸活动的事件。

本文中所述的效果提供低级的触摸事件的访问权限。这些低级别的事件不是可通过现有 `GestureRecognizer` 类，但它们是对某些类型的应用程序至关重要。例如，finger-paint 跟踪各手指，因为它们在屏幕上移动的应用程序需要。音乐键盘需要检测分流点，并释放上各个项，以及从一个密钥为另一种滑奏 gliding 手指。

效果非常适用于多点触控手指跟踪，因为可以将它附加到 Xamarin.Forms 的任何元素。

平台触摸事件

IOS、Android 和通用 Windows 平台都包括一个低级别 API，允许应用程序以检测触摸活动。所有的三种基本类型之间区分这些平台触摸事件：

- *按下*，当有手指触摸屏幕
- *移动*、当移动手指触摸屏幕
- *发布*，当手指松开的屏幕

在多点触控环境中，多个手指可同时触摸屏幕。各种平台包括应用程序可用于区分多个手指标识 (ID) 号。

在 iOS 中，`UIView` 类定义了三种可重写方法 `TouchesBegan`，`TouchesMoved`，和 `TouchesEnded` 与这三个基本事件相对应。文章多点触控手指跟踪介绍如何使用这些方法。但是，iOS 程序不需要重写派生类 `UIView` 使用这些方法。IOS `UIGestureRecognizer` 还定义了这些相同三种方法，并且您可以将派生的类的实例附加 `UIGestureRecognizer` 任何 `UIView` 对象。

在 Android 中，`View` 类定义可重写方法名为 `OnTouchEvent` 来处理所有触摸活动。触摸活动的类型定义的枚举成员 `Down`，`PointerDown`，`Move`，`Up`，并 `PointerUp` 文章中所述多点触控手指跟踪。Android `View` 还定义了名为的事件 `Touch` 允许事件处理程序附加到任何 `View` 对象。

通用 Windows 平台 (UWP) 中，`UIElement` 类定义名为事件 `PointerPressed`，`PointerMoved`，和 `PointerReleased`。这些文章中所述MSDN 上的处理指针输入文章和 API 文档 `UIElement` 类。

`Pointer` 中通用 Windows 平台的 API 旨在统一鼠标、触控和笔输入。为此，`PointerMoved` 在即使在未按下鼠标按钮时，鼠标移过元素时调用事件。`PointerRoutedEventArgs` 通常会显示这些事件的对象有一个名为 `Pointer` 具有一个名为属性 `IsInContact` 指示是否按下鼠标或手指的屏幕。

此外，UWP 定义两个名为的更多事件 `PointerEntered` 和 `PointerExited`。这些错误表示当鼠标或手指移动从一个元素到另一个。例如，考虑两个相邻元素名为 A 和 b。两个元素都已安装指针事件处理的程序。当手指按下 A 上，`PointerPressed` 调用事件。当手指移动时，一个调用 `PointerMoved` 事件。如果手指移动从 A 到 B，A 调用 `PointerExited` 事件和 B 调用 `PointerEntered` 事件。如果手指然后释放，B 调用 `PointerReleased` 事件。

IOS 和 Android 平台是不同于 UWP：首先获取对的调用的视图 `TouchesBegan` 或 `OnTouchEvent` 时上方的手指触摸视图将继续获取所有触摸活动，即使手指移动到不同的视图。如果应用程序捕获该指针的 UWP 可以同样行为：在 `PointerEntered` 事件处理程序中，元素调用 `CapturePointer`，然后获取从该手指触摸屏输入的所有活动。

UWP 方法证明是非常有用，对于某些类型的应用程序，例如，音乐键盘。每个密钥可以处理该注册表项的触摸事件，当有手指已从一个项滑到另一个使用时进行检测 `PointerEntered` 和 `PointerExited` 事件。

出于此原因，本文中所述的触控跟踪效果实现 UWP 方法。

触控跟踪影响 API

接触跟踪效果演示示例包含的类（和枚举）实现低级别触控跟踪。这些类型属于命名空间 `TouchTracking` 和单词开头 `Touch`。**TouchTrackingEffectDemos** .NET Standard 库项目包括 `TouchActionType` 触控事件的类型的枚举：

```
public enum TouchActionType
{
    Entered,
    Pressed,
    Moved,
    Released,
    Exited,
    Cancelled
}
```

所有平台还都包括一个事件，指出已取消触摸事件。

`TouchEffect` .NET 标准库中的类派生自 `RoutingEffect`，并定义名为的事件 `TouchAction` 和一个名为方法 `OnTouchAction`，它调用 `TouchAction` 事件：

```
public class TouchEffect : RoutingEffect
{
    public event TouchActionEventHandler TouchAction;

    public TouchEffect() : base("XamarinDocs.TouchEffect")
    {
    }

    public bool Capture { set; get; }

    public void OnTouchAction(Element element, TouchActionEventArgs args)
    {
        TouchAction?.Invoke(element, args);
    }
}
```

另请注意 `Capture` 属性。若要捕获触摸事件，应用程序必须设置此属性为 `true` 早于 `Pressed` 事件。否则，触控事件的行为类似于通用 Windows 平台中。

`TouchActionEventArgs` .NET Standard 库中的类包含附带的每个事件的所有信息：

```
public class TouchActionEventArgs : EventArgs
{
    public TouchActionEventArgs(long id, TouchActionType type, Point location, bool isInContact)
    {
        Id = id;
        Type = type;
        Location = location;
        IsInContact = isInContact;
    }

    public long Id { private set; get; }

    public TouchActionType Type { private set; get; }

    public Point Location { private set; get; }

    public bool IsInContact { private set; get; }
}
```

应用程序可以使用 `Id` 跟踪各手指的属性。请注意 `IsInContact` 属性。此属性始终为 `true` 有关 `Pressed` 事件并

`false` 为 `Released` 事件。它也是始终 `true` 为 `Moved` iOS 和 Android 上的事件。 `IsInContact` 属性可能会 `false` 为 `Moved` 事件时在桌面上运行该程序并没有一个按钮的情况下移动鼠标指针在通用 Windows 平台上按下。

可以使用 `TouchEffect` 自己的应用程序通过在.NET Standard 库项目中的解决方案，包括该文件并添加到实例中的类 `Effects` Xamarin.Forms 的任何元素的集合。附加到处理程序 `TouchAction` 获取触控事件的事件。

若要使用 `TouchEffect` 在自己的应用程序，您将需要包含在平台的实现**TouchTrackingEffectDemos**解决方案。

触控跟踪效果实现

IOS、Android 和 UWP 的实现 `TouchEffect` 开头的最简单的实现 (UWP) 和结尾的 iOS 实现，因为它是比其他更多结构复杂如下所述。

UWP 实现

UWP 实现 `TouchEffect` 最简单的方法。像往常一样，类派生自 `PlatformEffect` 并包含两个程序集特性：

```
[assembly: ResolutionGroupName("XamarinDocs")]
[assembly: ExportEffect(typeof(TouchTracking.UWP.TouchEffect), "TouchEffect")]

namespace TouchTracking.UWP
{
    public class TouchEffect : PlatformEffect
    {
        ...
    }
}
```

`OnAttached` 重写将某些信息保存为字段和附加到所有指针事件的处理程序：

```
public class TouchEffect : PlatformEffect
{
    FrameworkElement frameworkElement;
    TouchTracking.TouchEffect effect;
    Action<Element, TouchActionEventArgs> onTouchAction;

    protected override void OnAttached()
    {
        // Get the Windows FrameworkElement corresponding to the Element that the effect is attached to
        frameworkElement = Control == null ? Container : Control;

        // Get access to the TouchEffect class in the .NET Standard library
        effect = (TouchTracking.TouchEffect)Element.Effects.
                    FirstOrDefault(e => e is TouchTracking.TouchEffect);

        if (effect != null && frameworkElement != null)
        {
            // Save the method to call on touch events
            onTouchAction = effect.OnTouchAction;

            // Set event handlers on FrameworkElement
            frameworkElement.PointerEntered += OnPointerEntered;
            frameworkElement.PointerPressed += OnPointerPressed;
            frameworkElement.PointerMoved += OnPointerMoved;
            frameworkElement.PointerReleased += OnPointerReleased;
            frameworkElement.PointerExited += OnPointerExited;
            frameworkElement.PointerCanceled += OnPointerCancelled;
        }
    }
    ...
}
```

`OnPointerPressed` 处理程序的调用来调用影响事件 `onTouchAction` 字段中 `CommonHandler` 方法：

```
public class TouchEffect : PlatformEffect
{
    ...
    void OnPointerPressed(object sender, PointerRoutedEventArgs args)
    {
        CommonHandler(sender, TouchActionType.Pressed, args);

        // Check setting of Capture property
        if (effect.Capture)
        {
            (sender as FrameworkElement).CapturePointer(args.Pointer);
        }
    }
    ...
    void CommonHandler(object sender, TouchActionType touchActionType, PointerRoutedEventArgs args)
    {
        PointerPoint pointerPoint = args.GetCurrentPoint(sender as UIElement);
        Windows.Foundation.Point windowsPoint = pointerPoint.Position;

        onTouchAction(Element, new TouchActionEventArgs(args.Pointer.PointerId,
                                                         touchActionType,
                                                         new Point(windowsPoint.X, windowsPoint.Y),
                                                         args.Pointer.IsInContact));
    }
}
```

`OnPointerPressed` 此外会检查的值 `Capture` 属性的效果类中的.NET Standard 库和调用 `CapturePointer` 若是 `true` 。

其他 UWP 事件处理程序是更简单：

```
public class TouchEffect : PlatformEffect
{
    ...
    void OnPointerEntered(object sender, PointerRoutedEventArgs args)
    {
        CommonHandler(sender, TouchActionType.Entered, args);
    }
    ...
}
```

Android 实现

Android 和 iOS 实现是一定更复杂，因为它们必须实现 `Exited` 和 `Entered` 事件时手指将从一个元素移到另一个。这两个实现结构类似。

Android `TouchEffect` 类安装的处理程序 `Touch` 事件：

```
view = Control == null ? Container : Control;
...
view.Touch += OnTouch;
```

类还定义了两个静态字典：

```
public class TouchEffect : PlatformEffect
{
    ...
    static Dictionary<Android.Views.View, TouchEffect> viewDictionary =
        new Dictionary<Android.Views.View, TouchEffect>();

    static Dictionary<int, TouchEffect> idToEffectDictionary =
        new Dictionary<int, TouchEffect>();
    ...
```

`viewDictionary` 获取每个时间的新条目 `OnAttached` 调用重写：

```
viewDictionary.Add(view, this);
```

从字典中删除该条目 `OnDetached` 。每个实例 `TouchEffect` 效果附加到的特定视图相关联。静态字典允许任何 `TouchEffect` 实例以枚举所有其他视图和其对应 `TouchEffect` 实例。这是将事件从一个视图传输到另一个允许所必需的。

Android 将分配一个 ID 代码触摸事件，允许应用程序可以跟踪各手指。 `idToEffectDictionary` 将使用此 ID 代码相关联 `TouchEffect` 实例。项添加到此字典时 `Touch` 的手指按调用处理程序：

```
void OnTouch(object sender, Android.Views.View.TouchEventArgs args)
{
    ...
    switch (args.Event.ActionMasked)
    {
        case MotionEventActions.Down:
        case MotionEventActions.PointerDown:
            FireEvent(this, id, TouchActionType.Pressed, screenPointerCoords, true);

            idToEffectDictionary.Add(id, this);

            capture = libTouchEffect.Capture;
            break;
```

该项已从 `idToEffectDictionary` 手指从屏幕的发布时。 `FireEvent` 方法只是积累调用所需的所有信息 `OnTouchAction` 方法：

```
void FireEvent(TouchEffect touchEffect, int id, TouchActionType actionType, Point pointerLocation, bool
isInContact)
{
    // Get the method to call for firing events
    Action<Element, TouchActionEventArgs> onTouchAction = touchEffect.libTouchEffect.OnTouchAction;

    // Get the location of the pointer within the view
    touchEffect.view.GetLocationOnScreen(twoIntArray);
    double x = pointerLocation.X - twoIntArray[0];
    double y = pointerLocation.Y - twoIntArray[1];
    Point point = new Point(fromPixels(x), fromPixels(y));

    // Call the method
    onTouchAction(touchEffect.formsElement,
        new TouchActionEventArgs(id, actionType, point, isInContact));
}
```

在两种不同方式处理所有其他触摸屏输入类型：如果 `Capture` 属性是 `true` ，触控事件是相当简单转换为 `TouchEffect` 信息。它变得更加复杂时 `Capture` 是 `false` 因为可能需要从一个视图移到另一个触摸事件。这负责

`CheckForBoundaryHop` move 事件期间调用的方法。此方法利用了这两个静态字典。它通过枚举 `viewDictionary` 来确定当前触摸手指，和它使用的视图 `idToEffectDictionary` 用于存储当前 `TouchEffect` 实例（并因此，当前视图）具有特定 ID 相关联：

```
void CheckForBoundaryHop(int id, Point pointerLocation)
{
    TouchEffect touchEffectHit = null;

    foreach (Android.Views.View view in viewDictionary.Keys)
    {
        // Get the view rectangle
        try
        {
            view.GetLocationOnScreen(twoIntArray);
        }
        catch // System.ObjectDisposedException: Cannot access a disposed object.
        {
            continue;
        }
        Rectangle viewRect = new Rectangle(twoIntArray[0], twoIntArray[1], view.Width, view.Height);

        if (viewRect.Contains(pointerLocation))
        {
            touchEffectHit = viewDictionary[view];
        }
    }

    if (touchEffectHit != idToEffectDictionary[id])
    {
        if (idToEffectDictionary[id] != null)
        {
            FireEvent(idToEffectDictionary[id], id, TouchActionType.Exited, pointerLocation, true);
        }
        if (touchEffectHit != null)
        {
            FireEvent(touchEffectHit, id, TouchActionType.Entered, pointerLocation, true);
        }
        idToEffectDictionary[id] = touchEffectHit;
    }
}
```

如果已更改 `idToEffectDictionary`，该方法可能会调用 `FireEvent` 有关 `Exited` 和 `Entered` 到传输到另一个视图中。但是，手指可能已被移至占用而无需附加视图区域 `TouchEffect`，或从该区域具有附加的效果的视图。

请注意 `try` 和 `catch` 访问视图时阻止。在页中，导航到，然后导航回主页上，`OnDetached` 不会调用方法和项将保留在 `viewDictionary` 但 Android 将认为它们释放。

IOS 实现

IOS 实现是类似于 Android 的实现不同之处在于 iOS `TouchEffect` 类必须实例化的派生 `UIGestureRecognizer`。这是一个类中名为的 iOS 项目 `TouchRecognizer`。此类维护存储的两个静态字典 `TouchRecognizer` 实例：

```
static Dictionary<UIView, TouchRecognizer> viewDictionary =
    new Dictionary<UIView, TouchRecognizer>();

static Dictionary<long, TouchRecognizer> idToTouchDictionary =
    new Dictionary<long, TouchRecognizer>();
```

此结构的大部分 `TouchRecognizer` 类是类似于 Android `TouchEffect` 类。

将触摸效果放到工作

TouchTrackingEffectDemos 程序包含五个测试的常见任务的触控跟踪效果的页。

字数拖动页面允许您添加 `BoxView` 元素到 `AbsoluteLayout` 然后将它们拖在屏幕上。XAML 文件实例化两个 `Button` 添加的视图 `BoxView` 元素 `AbsoluteLayout` 清除 `AbsoluteLayout` 。

中的方法代码隐藏文件，添加一个新 `BoxView` 到 `AbsoluteLayout` 还添加了 `TouchEffect` 对象传递给 `BoxView` 并将事件处理程序附加到效果：

```
void AddBoxViewToLayout()
{
    BoxView boxView = new BoxView
    {
        WidthRequest = 100,
        HeightRequest = 100,
        Color = new Color(random.NextDouble(),
                          random.NextDouble(),
                          random.NextDouble())
    };

    TouchEffect touchEffect = new TouchEffect();
    touchEffect.TouchAction += OnTouchEffectAction;
    boxView.Effects.Add(touchEffect);
    absoluteLayout.Children.Add(boxView);
}
```

`TouchAction` 事件处理程序处理所有触控事件的所有 `BoxView` 元素，但它需要一些多加小心：它不能在单个上允许两根手指 `BoxView` 因为该程序仅实现拖动，和两个手指的会相互干扰。出于此原因，页上定义的当前跟踪每个手指嵌入的类：

```
class DragInfo
{
    public DragInfo(long id, Point pressPoint)
    {
        Id = id;
        PressPoint = pressPoint;
    }

    public long Id { private set; get; }

    public Point PressPoint { private set; get; }
}

Dictionary<BoxView, DragInfo> dragDictionary = new Dictionary<BoxView, DragInfo>();
```

`dragDictionary` 包含的一项每个 `BoxView` 当前正被拖动。

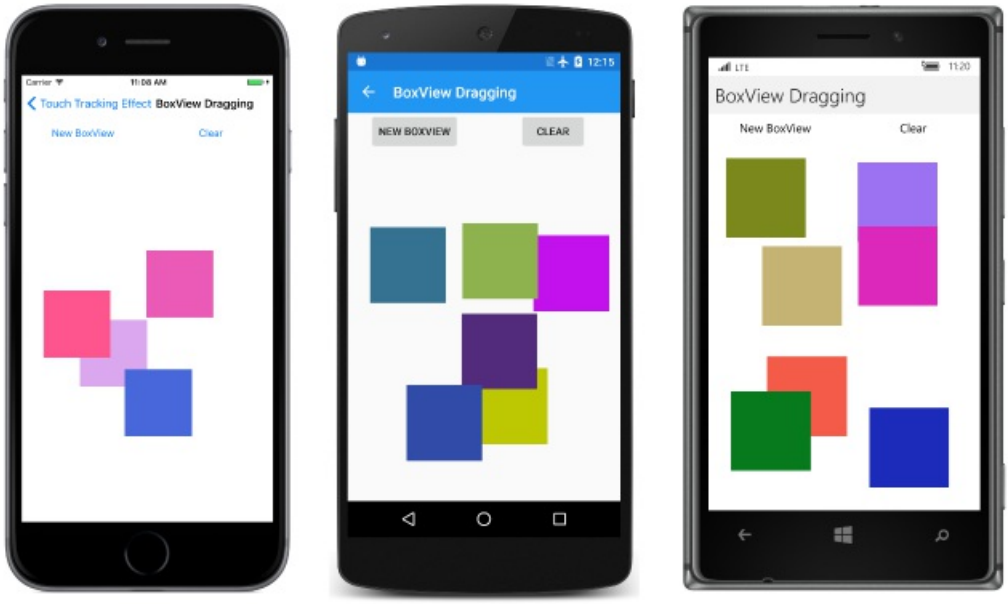
`Pressed` 触摸操作将项添加到此字典和 `Released` 操作将其删除。 `Pressed` 逻辑必须检查是否已经存在某个项的字典中 `BoxView` 。如果是这样， `BoxView` 为正被拖动，新的事件是第二个手指上的相同 `BoxView` 。有关 `Moved` 和 `Released` 操作，事件处理程序必须检查是否字典为此，有一个条目 `BoxView` 和的触摸屏输入 `Id` 属性拖动 `BoxView` 字典条目中的相匹配：

```
void OnTouchEffectAction(object sender, TouchActionEventArgs args)
{
    BoxView boxView = sender as BoxView;

    switch (args.Type)
    {
        case TouchActionType.Pressed:
            // Don't allow a second touch on an already touched BoxView
            if (!dragDictionary.ContainsKey(boxView))
            {
                dragDictionary.Add(boxView, new DragInfo(args.Id, args.Location));

                // Set Capture property to true
                TouchEffect touchEffect = (TouchEffect)boxView.Effects.FirstOrDefault(e => e is TouchEffect);
                touchEffect.Capture = true;
            }
            break;

        case TouchActionType.Moved:
            if (dragDictionary.ContainsKey(boxView) && dragDictionary[boxView].Id == args.Id)
            {
                Rectangle rect = AbsoluteLayout.GetLayoutBounds(boxView);
                Point initialLocation = dragDictionary[boxView].PressPoint;
                rect.X += args.Location.X - initialLocation.X;
                rect.Y += args.Location.Y - initialLocation.Y;
                AbsoluteLayout.SetLayoutBounds(boxView, rect);
            }
            break;

        case TouchActionType.Released:
            if (dragDictionary.ContainsKey(boxView) && dragDictionary[boxView].Id == args.Id)
            {
                dragDictionary.Remove(boxView);
            }
            break;
    }
}
```

`Pressed`逻辑集`Capture`的属性`TouchEffect`对象传递给`true`。这根手指的所有后续事件传送到同一个事件处理程序效果。

`Moved`逻辑移动`BoxView`通过更改`LayoutBounds`附加属性。`Location`属性的事件参数始终是相对于`BoxView`正在拖动，并且如果`BoxView`正被拖动的固定速度`Location`连续的事件的属性将大约相同。例如，如果按下一个手指`BoxView`中的中心，而`Pressed`操作存储区`PressPoint`属性（50，50），也仍然相同的后续事件。如果`BoxView`沿对角线方向拖动以恒定速率，后续`Location`属性期间`Moved`操作可能的值（55、55），在这种情况下`Moved`逻辑的水平和垂直位置加上5`BoxView`.这样可将移动`BoxView`，以便其中心再次是直接在手指下。

可以移动多个`BoxView`元素同时使用不同的手指。



子类化视图

通常情况下，更加容易 Xamarin.Forms 元素来处理其自身的触摸事件。**可拖**动**字数拖**动页面的功能与相同**字数拖**动页上，但用户拖动的实例的元素 `DraggableBoxView` 从派生类 `BoxView`：

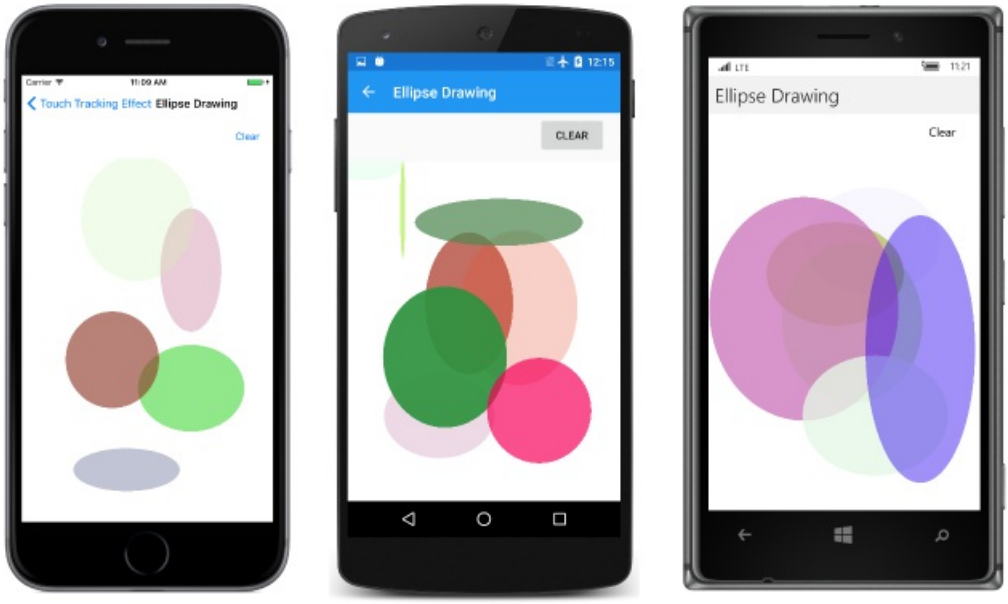
```
class DraggableBoxView : BoxView
{
    bool isBeingDragged;
    long touchId;
    Point pressPoint;

    public DraggableBoxView()
    {
        TouchEffect touchEffect = new TouchEffect
        {
            Capture = true
        };
        touchEffect.TouchAction += OnTouchEffectAction;
        Effects.Add(touchEffect);
    }

    void OnTouchEffectAction(object sender, TouchActionEventArgs args)
    {
        switch (args.Type)
        {
            case TouchActionType.Pressed:
                if (!isBeingDragged)
                {
                    isBeingDragged = true;
                    touchId = args.Id;
                    pressPoint = args.Location;
                }
                break;

            case TouchActionType.Moved:
                if (isBeingDragged && touchId == args.Id)
                {
                    TranslationX += args.Location.X - pressPoint.X;
                    TranslationY += args.Location.Y - pressPoint.Y;
                }
                break;

            case TouchActionType.Released:
                if (isBeingDragged && touchId == args.Id)
                {
                    isBeingDragged = false;
                }
                break;
        }
    }
}
```

构造函数创建并附加 `TouchEffect`，并设置 `Capture` 时首次实例化该对象的属性。没有字典是必需的因为该类本身存储 `isBeingDragged`、`pressPoint`，和 `touchId` 与每个手指关联的值。`Moved` 处理更改 `TranslationX` 并 `TranslationY` 属性，以便处理逻辑将即使的父 `DraggableBoxView` 不是 `AbsoluteLayout`。

将与 SkiaSharp 集成

接下来两个演示需要图形，并且使用 SkiaSharp 实现此目的。你可能想要了解有关Xamarin.Forms 中使用 SkiaSharp研究这些示例之前。前两篇文章（"SkiaSharp 绘制基础知识"和"SkiaSharp 线和路径"）涵盖所有需要此处。

椭圆绘图页面允许您通过轻扫手指在屏幕上的绘制一个椭圆。具体取决于如何移动手指，您可以从左上到右下角或任何其他的对角角绘制椭圆。使用随机颜色和不透明度是绘制椭圆。



如果您然后触摸省略号之一，可以将其拖到另一个位置。这要求称为"命中测试，"这涉及到搜索的特定点处的图形对象的技术。SkiaSharp 省略号不是 Xamarin.Forms 元素，因此它们不能执行其自己 `TouchEffect` 处理。 `TouchEffect` 必须应用于整个 `SKCanvasView` 对象。

EllipseDrawPage.xaml文件实例化 `SKCanvasView` 在单个单元格 `Grid` 。 `TouchEffect` 对象附加到的 `Grid` :

```
<Grid x:Name="canvasViewGrid"
        Grid.Row="1"
        BackgroundColor="White">

    <skia:SKCanvasView x:Name="canvasView"
                       PaintSurface="OnCanvasViewPaintSurface" />
    <Grid.Effects>
        <tt:TouchEffect Capture="True"
                        TouchAction="OnTouchEffectAction" />
    </Grid.Effects>
</Grid>
```

在 Android 和通用 Windows 平台， `TouchEffect` 可以直接向附加 `SKCanvasView` ，但不起作用的 iOS 上。请注意， `Capture` 属性设置为 `true` 。

由类型的对象表示 SkiaSharp 呈现每个椭圆 `EllipseDrawingFigure` :

```
class EllipseDrawingFigure
{
    SKPoint pt1, pt2;

    public EllipseDrawingFigure()
    {
    }

    public SKColor Color { set; get; }

    public SKPoint StartPoint
    {
        set
        {
            pt1 = value;
            MakeRectangle();
        }
    }

    public SKPoint EndPoint
    {
        set
        {
            pt2 = value;
            MakeRectangle();
        }
    }

    void MakeRectangle()
    {
        Rectangle = new SKRect(pt1.X, pt1.Y, pt2.X, pt2.Y).Standardized;
    }

    public SKRect Rectangle { set; get; }

    // For dragging operations
    public Point LastFingerLocation { set; get; }

    // For the dragging hit-test
    public bool IsInEllipse(SKPoint pt)
    {
        SKRect rect = Rectangle;

        return (Math.Pow(pt.X - rect.MidX, 2) / Math.Pow(rect.Width / 2, 2) +
                Math.Pow(pt.Y - rect.MidY, 2) / Math.Pow(rect.Height / 2, 2)) < 1;
    }
}
```

`StartPoint` 并 `EndPoint` 程序处理触摸输入; 时所使用属性 `Rectangle` 属性用于绘制椭圆。 `LastFingerLocation` 属性时所拖动的椭圆，发挥作用， `IsInEllipse` 方法中的命中测试的辅助工具。该方法将返回 `true` 如果点在 ellipse 内部的。

代码隐藏文件维护三个集合：

```
Dictionary<long, EllipseDrawingFigure> inProgressFigures = new Dictionary<long, EllipseDrawingFigure>();
List<EllipseDrawingFigure> completedFigures = new List<EllipseDrawingFigure>();
Dictionary<long, EllipseDrawingFigure> draggingFigures = new Dictionary<long, EllipseDrawingFigure>();
```

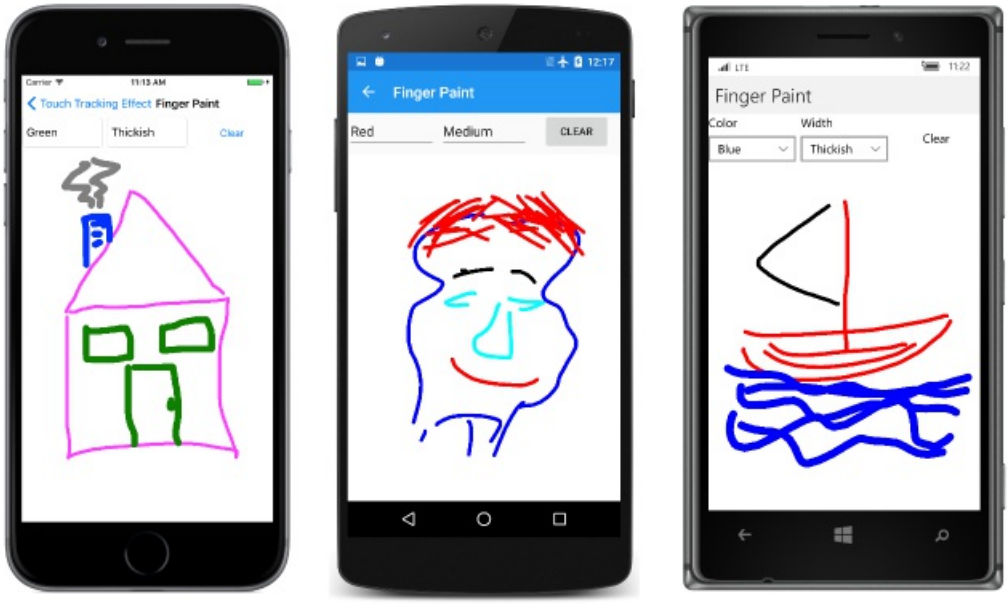
`draggingFigure` 字典包含的子集 `completedFigures` 集合。SkiaSharp `PaintSurface` 事件处理程序只需将呈现在这些对象 `completedFigures` 和 `inProgressFigures` 集合：

```
SKPaint paint = new SKPaint
{
    Style = SKPaintStyle.Fill
};
...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKCanvas canvas = args.Surface.Canvas;
    canvas.Clear();

    foreach (EllipseDrawingFigure figure in completedFigures)
    {
        paint.Color = figure.Color;
        canvas.DrawOval(figure.Rectangle, paint);
    }
    foreach (EllipseDrawingFigure figure in inProgressFigures.Values)
    {
        paint.Color = figure.Color;
        canvas.DrawOval(figure.Rectangle, paint);
    }
}
```

触摸处理的最棘手部分是`Pressed`处理。这就是执行命中测试，但如果代码检测到用户的手指下一个椭圆，如果当前未被另一个手指拖动可仅被拖动该椭圆形。如果用户的手指下没有椭圆，代码开始绘制新椭圆的过程：

```
case TouchActionType.Pressed:
    bool isDragOperation = false;

    // Loop through the completed figures
    foreach (EllipseDrawingFigure fig in completedFigures.Reverse<EllipseDrawingFigure>())
    {
        // Check if the finger is touching one of the ellipses
        if (fig.IsInEllipse(ConvertToPixel(args.Location)))
        {
            // Tentatively assume this is a dragging operation
            isDragOperation = true;

            // Loop through all the figures currently being dragged
            foreach (EllipseDrawingFigure draggedFigure in draggingFigures.Values)
            {
                // If there's a match, we'll need to dig deeper
                if (fig == draggedFigure)
                {
                    isDragOperation = false;
                    break;
                }
            }

            if (isDragOperation)
            {
                fig.LastFingerLocation = args.Location;
                draggingFigures.Add(args.Id, fig);
                break;
            }
        }
    }

    if (isDragOperation)
    {
        // Move the dragged ellipse to the end of completedFigures so it's drawn on top
        EllipseDrawingFigure fig = draggingFigures[args.Id];
        completedFigures.Remove(fig);
        completedFigures.Add(fig);
    }
    else // start making a new ellipse
    {
        // Random bytes for random color
        byte[] buffer = new byte[4];
        random.NextBytes(buffer);

        EllipseDrawingFigure figure = new EllipseDrawingFigure
        {
            Color = new SKColor(buffer[0], buffer[1], buffer[2], buffer[3]),
            StartPoint = ConvertToPixel(args.Location),
            EndPoint = ConvertToPixel(args.Location)
        };
        inProgressFigures.Add(args.Id, figure);
    }
    canvasView.InvalidateSurface();
    break;
```

其他 SkiaSharp 示例是**手指绘制**页。**可以从两个**选择**笔画颜色和笔划**宽度`Picker`视图，**然后使用一个或多个手指**绘**制**：



此示例还需要一个单独的类表示在屏幕上绘制每个行：

```
class FingerPaintPolyline
{
    public FingerPaintPolyline()
    {
        Path = new SKPath();
    }

    public SKPath Path { set; get; }

    public Color StrokeColor { set; get; }

    public float StrokeWidth { set; get; }
}
```

`SKPath` 对象用于呈现每个行。FingerPaint.xaml.cs文件维护这些对象，一个用于当前所绘制这些折线，另一个用于已完成的折线的两个集合：

```
Dictionary<long, FingerPaintPolyline> inProgressPolylines = new Dictionary<long, FingerPaintPolyline>();
List<FingerPaintPolyline> completedPolylines = new List<FingerPaintPolyline>();
```

`Pressed` 处理过程将创建一个新 `FingerPaintPolyline`，调用 `MoveTo` 上要存储的初始点的路径对象，并添加到该对象 `inProgressPolylines` 字典。`Moved` 处理调用 `LineTo` 上的新的指位置的路径对象和 `Released` 处理传输已完成从 polyline `inProgressPolylines` 到 `completedPolylines`。再次重申，实际 SkiaSharp 绘制代码是相对简单：

```
SKPaint paint = new SKPaint
{
    Style = SKPaintStyle.Stroke,
    StrokeCap = SKStrokeCap.Round,
    StrokeJoin = SKStrokeJoin.Round
};
...
void OnCanvasViewPaintSurface(object sender, SKPaintSurfaceEventArgs args)
{
    SKCanvas canvas = args.Surface.Canvas;
    canvas.Clear();

    foreach (FingerPaintPolyline polyline in completedPolylines)
    {
        paint.Color = polyline.StrokeColor.ToSKColor();
        paint.StrokeWidth = polyline.StrokeWidth;
        canvas.DrawPath(polyline.Path, paint);
    }

    foreach (FingerPaintPolyline polyline in inProgressPolylines.Values)
    {
        paint.Color = polyline.StrokeColor.ToSKColor();
        paint.StrokeWidth = polyline.StrokeWidth;
        canvas.DrawPath(polyline.Path, paint);
    }
}
```

跟踪视图来查看触摸

设置所有前面的示例 `Capture` 的属性 `TouchEffect` 到 `true`，时 `TouchEffect` 已创建，或者当 `Pressed` 事件发生。这可确保同一元素接收首次按下该视图的手指与关联的所有事件。最后一个示例的用途不设置 `Capture` 到 `true`。当屏幕的手指移动从一个元素到另一个时，这会导致不同的行为。从移动手指的元素接收的事件 `Type` 属性设置为 `TouchActionType.Exited` 和第二个元素接收的事件 `Type` 设置为 `TouchActionType.Entered`。

此类型的触摸处理是非常有用的音乐键盘。需要能够检测到它按下时，而且还时手指从一个项滑到另一个项。

无提示键盘页上定义小 `WhiteKey` 并 `BlackKey` 派生的类 `Key`，它派生 `BoxView`。

`Key` 类已准备好实际音乐程序中使用。它定义了名为的公共属性 `IsPressed` 和 `KeyNumber`，用于将设置为建立由 MIDI 标准的键代码。`Key` 类还定义了名为的事件 `StatusChanged`，这当调用 `IsPressed` 属性更改。

每个密钥允许使用多根手指。出于此原因，`Key` 类维护 `List` 的当前接触该注册表项的所有根手指的触控 ID 号：

```
List<long> ids = new List<long>();
```

`TouchAction` 事件处理程序添加到一个 ID `ids` 两个列表 `Pressed` 事件类型和一个 `Entered` 类型，但仅当 `IsInContact` 属性是 `true` 为 `Entered` 事件。ID 已从 `List` 有关 `Released` 或 `Exited` 事件：

```
void OnTouchEffectAction(object sender, TouchActionEventArgs args)
{
    switch (args.Type)
    {
      case TouchActionType.Pressed:
          AddToList(args.Id);
          break;

        case TouchActionType.Entered:
            if (args.IsInContact)
            {
                AddToList(args.Id);
            }
            break;

        case TouchActionType.Moved:
            break;

        case TouchActionType.Released:
        case TouchActionType.Exited:
            RemoveFromList(args.Id);
            break;
    }
}
```

`AddToList` 并 `RemoveFromList` 这两种方法检查是否 `List` empty 和非空之间发生了更改，如果是这样，调用 `StatusChanged` 事件。

各种 `WhiteKey` 并 `BlackKey` 中的页的排列元素XAML 文件，当手机保持在横向模式下时，该查找最佳：

如果在扫描您的手指，您将看到颜色的细微更改从一个密钥传输触控事件，到另一个。

总结

本文演示了如何调用事件起作用，以及如何编写和使用实现低级别多点触控处理的效果。

相关链接

- 在 iOS 中的多点触控手指跟踪
- 多点触控手指在 Android 中跟踪
- 触控跟踪效果（示例）

在 Xamarin.Forms 中处理的文件

2018/11/13 • • Edit Online

可以使用代码在.NET Standard 库中，或通过使用嵌入的资源来实现处理用于 Xamarin.Forms 的文件。

概述

Xamarin.Forms 代码在多个平台上运行 - 每个平台都有自己的文件系统。以前，这意味着，读取和写入文件非常轻松地执行每个平台上使用本机文件 Api。或者，嵌入的资源是更简单的解决方案，用于将数据文件的应用。但是，使用.NET Standard 2.0 有可能共享.NET Standard 库中的文件访问代码。

有关处理图像文件的信息，请参阅处理图像页。

保存和加载文件

`System.IO` 类可用于访问每个平台上的文件系统。`File` 类，可以创建、删除和读取文件，和 `Directory` 类可以创建、删除或枚举目录的内容。此外可以使用 `Stream` 子类，它可以提供更高的控制文件操作（如文件内压缩或位置搜索）。

可以使用编写文本文件 `File.WriteAllText` 方法：

```
File.WriteAllText(fileName, text);
```

可以使用读取文本文件 `File.ReadAllText` 方法：

```
string text = File.ReadAllText(fileName);
```

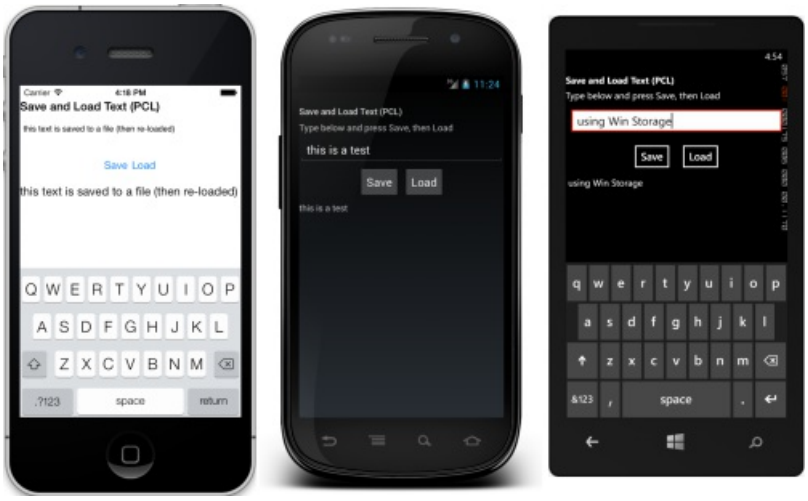
此外，`File.Exists` 方法确定是否存在指定的文件：

```
bool doesExist = File.Exists(fileName);
```

可通过使用的值从.NET Standard 库确定的每个平台上的文件的路径 `Environment.SpecialFolder` 枚举的第一个参数作为 `Environment.GetFolderPath` 方法。这可组合使用的文件名 `Path.Combine` 方法：

```
string fileName = Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"temp.txt");
```

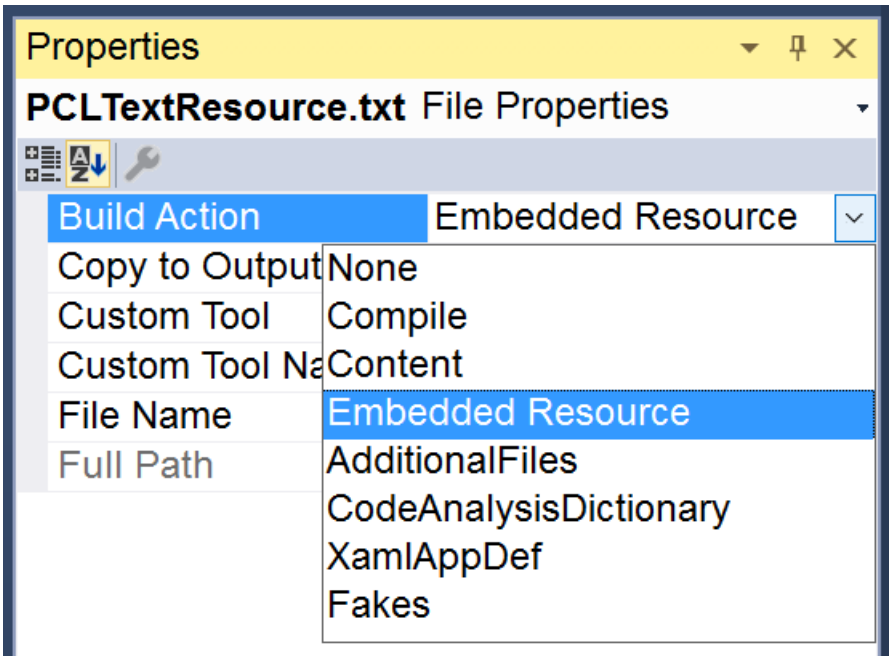
示例应用，其中包括一个页面，将保存并加载文本中演示了这些操作：



加载文件作为资源嵌入

若要嵌入到一个文件 **.NET Standard**程序集，创建或添加文件，并确保**生成操作**：**EmbeddedResource**。

- Visual Studio
- Visual Studio for Mac



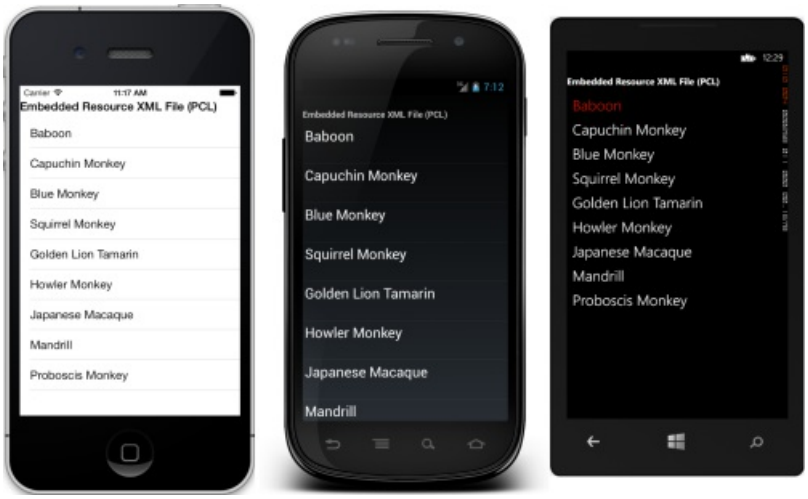
`GetManifestResourceStream` 用于访问嵌入的文件使用其资源 **ID**。默认情况下的资源 ID 是带有-中嵌入的项目的默认命名空间前缀的文件名在这种情况下该程序集是**WorkingWithFiles**，文件名为**PCLTextResource.txt**，因此资源 ID 是 `WorkingWithFiles.PCLTextResource.txt`。

```
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(LoadResourceText)).Assembly;
Stream stream = assembly.GetManifestResourceStream("WorkingWithFiles.PCLTextResource.txt");
string text = "";
using (var reader = new System.IO.StreamReader (stream)) {
    text = reader.ReadToEnd ();
}
```

`text` 然后可以使用变量来显示文本或否则代码中使用它。此屏幕截图示例应用将会显示在呈现文本 `Label` 控件。

加载和反序列化 XML 也同样简单。下面的代码演示的 XML 文件加载并反序列化从一种资源，然后绑定到 `ListView` 进行显示。XML 文件包含的数组 `Monkey`（在示例代码中定义的类）的对象。

```
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(LoadResourceText)).Assembly;
Stream stream = assembly.GetManifestResourceStream("WorkingWithFiles.PCLXmlResource.xml");
List<Monkey> monkeys;
using (var reader = new System.IO.StreamReader (stream)) {
    var serializer = new XmlSerializer(typeof(List<Monkey>));
    monkeys = (List<Monkey>)serializer.Deserialize(reader);
}
var listView = new ListView ();
listView.ItemsSource = monkeys;
```



在共享项目中嵌入

共享项目可以作为嵌入资源，还包含文件，但共享项目的内容将编译到引用的项目，因为前缀用于嵌入的文件资源 Id 可以更改。这意味着每个嵌入的文件的资源 ID 可能会因每个平台。

有两个解决方案共享的项目这一问题：

- **同步项目**-编辑每个平台使用的项目属性同一程序集名称和默认命名空间。此值然后可以作为嵌入资源共享项目中的 Id 的前缀为"硬编码"。
- **#if 编译器指令**-使用编译器指令设置正确的资源 ID 前缀，并使用该值来动态构造正确的资源 id。

说明第二个选项的代码如下所示。编译器指令用于选择的硬编码资源前缀（这通常是与引用的项目的默认命名空间相同）。`resourcePrefix` 变量然后用于通过串联使用嵌入的资源文件名创建一个有效的资源 ID。

```
#if __IOS__
var resourcePrefix = "WorkingWithFiles.iOS.";
#endif
#if __ANDROID__
var resourcePrefix = "WorkingWithFiles.Droid.";
#endif

Debug.WriteLine("Using this resource prefix: " + resourcePrefix);
// note that the prefix includes the trailing period '.' that is required
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(SharedPage)).Assembly;
Stream stream = assembly.GetManifestResourceStream
    (resourcePrefix + "SharedTextResource.txt");
```

组织资源

上面的示例假定该文件嵌入.NET Standard 库项目，为资源 id 的形式的根目录**Namespace.Filename.Extension**，如 `WorkingWithFiles.PCLTextResource.txt` 和 `WorkingWithFiles.iOS.SharedTextResource.txt` 。

就可以组织文件夹中的嵌入的资源。当嵌入的资源放置在文件夹中时，文件夹名称一部分的资源 ID（用句点分隔），以便资源 ID 格式变为**Namespace.Folder.Filename.Extension**。将示例应用程序中使用到的文件夹的文件放**MyFolder**会使相应资源 Id `WorkingWithFiles.MyFolder.PCLTextResource.txt` 和 `WorkingWithFiles.iOS.MyFolder.SharedTextResource.txt` 。

调试嵌入的资源

因为它是有时难以理解为什么不加载特定的资源，可以暂时将以下调试代码添加到应用程序以帮助确认正确配置了资源。它会输出所有已知的资源嵌入到给定的程序集错误板来帮助调试加载问题的资源。

```
using System.Reflection;
// ...
// use for debugging, not in released app code!
var assembly = IntrospectionExtensions.GetTypeInfo(typeof(SharedPage)).Assembly;
foreach (var res in assembly.GetManifestResourceNames()) {
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}
```

总结

本文介绍了一些简单的文件操作来保存和加载在设备上的文本和用于加载嵌入的资源。使用.NET Standard 2.0，则可以共享.NET Standard 库中的文件访问代码。

相关链接

- FilesSample
- Xamarin.Forms 示例
- 使用 Xamarin.iOS 中的文件系统

Xamarin.Forms 手势

2018/10/26 • • Edit Online

笔势识别器可以用于检测在 Xamarin.Forms 应用程序中的用户与视图交互。

Xamarin.Forms `GestureRecognizer` 类上支持点击、挤压、平移和轻扫手势 `View` 实例。

添加的点击手势识别程序

点击手势用于点击检测和识别的 `TapGestureRecognizer` 类。

添加捏合手势识别器

收缩手势用于执行交互式缩放和的可识别 `PinchGestureRecognizer` 类。

添加平移手势识别器

平移手势用于检测的手指在屏幕上四处移动，并将该移动应用到的内容，并识别与 `PanGestureRecognizer` 类。

添加轻扫手势识别器

轻扫手势手指在水平或垂直方向，在屏幕上移动，并且通常用于初始化通过内容导航时发生。轻扫手势识别具有 `SwipeGestureRecognizer` 类。

添加的点击手势识别程序

2018/10/19 • • Edit Online

点击手势用于点击检测并使用 TapGestureRecognizer 类实现。

若要使用户界面元素与点击手势可单击，创建 `TapGestureRecognizer` 实例，则处理 `Tapped` 事件，并添加到新的笔势识别器 `GestureRecognizers` 上的用户界面元素的集合。下面的代码示例演示 `TapGestureRecognizer` 附加到 `Image` 元素：

```
var tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.Tapped += (s, e) => {
    // handle the tap
};
image.GestureRecognizers.Add(tapGestureRecognizer);
```

默认情况下该映像将响应单个分流点。设置 `NumberOfTapsRequired` 属性时要等待的双点击（或更多分流点，如果需要）。

```
tapGestureRecognizer.NumberOfTapsRequired = 2; // double-tap
```

当 `NumberOfTapsRequired` 设置高于 1，事件处理程序，才会执行如果在一段时间（这段不是可配置）内未发生两次点击。如果在该时间段内不会发生第二个（或后续）两次点击有效地忽略它们并点击计数重新启动。

使用 Xaml

笔势识别器可以添加到 Xaml 中，使用附加的属性中的控件。若要添加的语法 `TapGestureRecognizer` 如下所示的图像 (在这种情况下定义*双击*事件):

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
                Tapped="OnTapGestureRecognizerTapped"
                NumberOfTapsRequired="2" />
  </Image.GestureRecognizers>
</Image>
```

（在此示例中）的事件处理程序的代码递增计数器，并会将映像从颜色更改为黑色&白色。

```
void OnTapGestureRecognizerTapped(object sender, EventArgs args)
{
    tapCount++;
    var imageSender = (Image)sender;
    // watch the monkey go from color to black&white!
    if (tapCount % 2 == 0) {
        imageSender.Source = "tapped.jpg";
    } else {
        imageSender.Source = "tapped_bw.jpg";
    }
}
```

使用 ICommand

通常使用模型-视图-视图模型 (MVVM) 模式的应用程序使用 `ICommand` 而不是直接绑定事件处理程序。 `TapGestureRecognizer` 可以轻松地支持 `ICommand` 通过在代码中设置绑定：

```
var tapGestureRecognizer = new TapGestureRecognizer();
tapGestureRecognizer.SetBinding (TapGestureRecognizer.CommandProperty, "TapCommand");
image.GestureRecognizers.Add(tapGestureRecognizer);
```

也可以使用 Xaml:

```
<Image Source="tapped.jpg">
    <Image.GestureRecognizers>
        <TapGestureRecognizer
            Command="{Binding TapCommand}"
            CommandParameter="Image1" />
    </Image.GestureRecognizers>
</Image>
```

可以在此示例中找到此视图模型的完整代码。相关 `Command` 实现的详细信息如下所示：

```
public class TapViewModel : INotifyPropertyChanged
{
    int taps = 0;
    ICommand tapCommand;
    public TapViewModel () {
        // configure the TapCommand with a method
        tapCommand = new Command (OnTapped);
    }
    public ICommand TapCommand {
        get { return tapCommand; }
    }
    void OnTapped (object s)  {
        taps++;
        Debug.WriteLine ("parameter: " + s);
    }
    //region INotifyPropertyChanged code omitted
}
```

相关链接

- TapGesture（示例）
- GestureRecognizer
- TapGestureRecognizer

添加捏合手势识别器

2018/10/19 • • Edit Online

捏合手势用于执行交互式缩放，并使用 PinchGestureRecognizer 类实现。捏合手势的常见方案是图像的执行交互式 pinch 位置处的缩放。这通过缩放内容的视区，来实现，本文中所示。

若要使用户界面元素可使用收缩手势进行缩放，创建 `PinchGestureRecognizer` 实例，则处理 `PinchUpdated` 事件，并添加到新的笔势识别器 `GestureRecognizers` 上的用户界面元素的集合。下面的代码示例演示 `PinchGestureRecognizer` 附加到 `Image` 元素：

```
var pinchGesture = new PinchGestureRecognizer();
pinchGesture.PinchUpdated += (s, e) => {
  // Handle the pinch
};
image.GestureRecognizers.Add(pinchGesture);
```

这也可以在 XAML，如下面的代码示例中所示：

```
<Image Source="waterfront.jpg">
  <Image.GestureRecognizers>
    <PinchGestureRecognizer PinchUpdated="OnPinchUpdated" />
  </Image.GestureRecognizers>
</Image>
```

有关代码 `OnPinchUpdated` 事件处理程序然后添加到代码隐藏文件：

```
void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
{
  // Handle the pinch
}
```

正在创建 PinchToZoom 容器

处理捏合手势进行缩放操作需要某些数学运算来转换用户界面。本部分包含执行数学计算，可用于以交互方式缩放任何用户界面元素的通用帮助器类。以下代码示例演示 `PinchToZoomContainer` 类：

```
public class PinchToZoomContainer : ContentView
{
  ...

  public PinchToZoomContainer ()
  {
    var pinchGesture = new PinchGestureRecognizer ();
    pinchGesture.PinchUpdated += OnPinchUpdated;
    GestureRecognizers.Add (pinchGesture);
  }

  void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
  {
    ...
  }
}
```

此类可以封装用户界面元素，以便收缩手势将缩放预包装的用户界面元素。以下 XAML 代码示例所示 `PinchToZoomContainer` 包装 `Image` 元素：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:PinchGesture;assembly=PinchGesture"
             x:Class="PinchGesture.HomePage">
    <ContentPage.Content>
        <Grid Padding="20">
            <local:PinchToZoomContainer>
                <local:PinchToZoomContainer.Content>
                    <Image Source="waterfront.jpg" />
                </local:PinchToZoomContainer.Content>
            </local:PinchToZoomContainer>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

下面的代码示例演示如何 `PinchToZoomContainer` 包装 `Image` C# 页面中的元素：

```
public class HomePageCS : ContentPage
{
  public HomePageCS ()
  {
    Content = new Grid {
      Padding = new Thickness (20),
      Children = {
        new PinchToZoomContainer {
          Content = new Image { Source = ImageSource.FromFile ("waterfront.jpg") }
        }
      }
    };
  }
}
```

当 `Image` 元素收到收缩手势时，显示的图像将进行放大加入或退出。由执行缩放 `PinchZoomContainer.OnPinchUpdated` 方法，在下面的代码示例所示：

```
void OnPinchUpdated (object sender, PinchGestureUpdatedEventArgs e)
{
  if (e.Status == GestureStatus.Started) {
    // Store the current scale factor applied to the wrapped user interface element,
    // and zero the components for the center point of the translate transform.
    startScale = Content.Scale;
    Content.AnchorX = 0;
    Content.AnchorY = 0;
  }
  if (e.Status == GestureStatus.Running) {
    // Calculate the scale factor to be applied.
    currentScale += (e.Scale - 1) * startScale;
    currentScale = Math.Max (1, currentScale);

    // The ScaleOrigin is in relative coordinates to the wrapped user interface element,
    // so get the X pixel coordinate.
    double renderedX = Content.X + xOffset;
    double deltaX = renderedX / Width;
    double deltaWidth = Width / (Content.Width * startScale);
    double originX = (e.ScaleOrigin.X - deltaX) * deltaWidth;

    // The ScaleOrigin is in relative coordinates to the wrapped user interface element,
    // so get the Y pixel coordinate.
    double renderedY = Content.Y + yOffset;
    double deltaY = renderedY / Height;
    double deltaHeight = Height / (Content.Height * startScale);
    double originY = (e.ScaleOrigin.Y - deltaY) * deltaHeight;

    // Calculate the transformed element pixel coordinates.
    double targetX = xOffset - (originX * Content.Width) * (currentScale - startScale);
    double targetY = yOffset - (originY * Content.Height) * (currentScale - startScale);

    // Apply translation based on the change in origin.
    Content.TranslationX = targetX.Clamp (-Content.Width * (currentScale - 1), 0);
    Content.TranslationY = targetY.Clamp (-Content.Height * (currentScale - 1), 0);

    // Apply scale factor.
    Content.Scale = currentScale;
  }
  if (e.Status == GestureStatus.Completed) {
    // Store the translation delta's of the wrapped user interface element.
    xOffset = Content.TranslationX;
    yOffset = Content.TranslationY;
  }
}
```

此方法更新基于用户的收缩手势的预包装的用户界面元素的缩放级别。这通过使用的值来实现 `Scale`，`ScaleOrigin` 并 `Status` 属性 `PinchGestureUpdatedEventArgs` 实例来计算 pinch 手势的原始应用的缩放因子。预包装的用户元素然后放大捏合手势原点通过设置其 `TranslationX`， `TranslationY`，以及 `Scale` 计算出的值的属性。

相关链接

- PinchGesture（示例）
- GestureRecognizer
- PinchGestureRecognizer

添加平移手势识别器

2018/10/19 • • Edit Online

平移手势用于检测的手指在屏幕上四处移动，并将该移动应用到的内容，并且通过实现 `PanGestureRecognizer` *类。平移手势的常见方案是水平和垂直方向平移图像，以便将图像尺寸小于视区中显示时可以查看的所有映像内容。这通过移动中位于视区的映像来实现，本文中所示。*

若要使可移动与平移手势的用户界面元素，创建 `PanGestureRecognizer` 实例，则处理 `PanUpdated` 事件，并添加到新的笔势识别器 `GestureRecognizers` 上的用户界面元素的集合。下面的代码示例演示 `PanGestureRecognizer` 附加到 `Image` 元素：

```
var panGesture = new PanGestureRecognizer();
panGesture.PanUpdated += (s, e) => {
  // Handle the pan
};
image.GestureRecognizers.Add(panGesture);
```

这也可以在 XAML，如下面的代码示例中所示：

```
<Image Source="MonoMonkey.jpg">
  <Image.GestureRecognizers>
    <PanGestureRecognizer PanUpdated="OnPanUpdated" />
  </Image.GestureRecognizers>
</Image>
```

有关代码 `OnPanUpdated` 事件处理程序然后添加到代码隐藏文件：

```
void OnPanUpdated (object sender, PanUpdatedEventArgs e)
{
  // Handle the pan
}
```

> **NOTE**
>
> 需要在 Android 上的正确平移Xamarin.Forms 2.1.0-pre1 NuGet 包最小值。

创建平移容器

本部分包含的通用帮助器类执行自由格式平移，这通常适用于在映像或映射中导航。处理平移手势来执行此操作需要某些数学运算来转换用户界面。此数学用于仅在预包装的用户界面元素的边界内平移。以下代码示例演示 `PanContainer` 类：

```
public class PanContainer : ContentView
{
  double x, y;

  public PanContainer ()
  {
    // Set PanGestureRecognizer.TouchPoints to control the
    // number of touch points needed to pan
    var panGesture = new PanGestureRecognizer ();
    panGesture.PanUpdated += OnPanUpdated;
    GestureRecognizers.Add (panGesture);
  }

  void OnPanUpdated (object sender, PanUpdatedEventArgs e)
  {
    ...
  }
}
```

此类可以封装用户界面元素，以便手势将平移预包装的用户界面元素。以下 XAML 代码示例所示 `PanContainer` 包装 `Image` 元素：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:PanGesture"
             x:Class="PanGesture.HomePage">
    <ContentPage.Content>
        <AbsoluteLayout>
            <local:PanContainer>
                <Image Source="MonoMonkey.jpg" WidthRequest="1024" HeightRequest="768" />
            </local:PanContainer>
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>
```

下面的代码示例演示如何 `PanContainer` 包装 `Image` C# 页面中的元素：

```
public class HomePageCS : ContentPage
{
  public HomePageCS ()
  {
    Content = new AbsoluteLayout {
      Padding = new Thickness (20),
      Children = {
        new PanContainer {
          Content = new Image {
            Source = ImageSource.FromFile ("MonoMonkey.jpg"),
            WidthRequest = 1024,
            HeightRequest = 768
          }
        }
      }
    };
  }
}
```

在这两个示例中， `WidthRequest` 并 `HeightRequest` 属性设置为要显示的图像的宽度和高度值。

当 `Image` 元素收到平移手势时，将素数所显示的图像。由执行平移 `PanContainer.OnPanUpdated` 方法，在下面的代码示例所示：

```
void OnPanUpdated (object sender, PanUpdatedEventArgs e)
{
  switch (e.StatusType) {
  case GestureStatus.Running:
    // Translate and ensure we don't pan beyond the wrapped user interface element bounds.
    Content.TranslationX =
      Math.Max (Math.Min (0, x + e.TotalX), -Math.Abs (Content.Width - App.ScreenWidth));
    Content.TranslationY =
      Math.Max (Math.Min (0, y + e.TotalY), -Math.Abs (Content.Height - App.ScreenHeight));
    break;

  case GestureStatus.Completed:
    // Store the translation applied during the pan
    x = Content.TranslationX;
    y = Content.TranslationY;
    break;
  }
}
```

此方法更新的预包装的用户界面元素，基于用户的平移手势可查看的内容。这通过使用的值来实现 `TotalX` 并 `TotalY` 属性的 `PanUpdatedEventArgs` 实例来计算方向和平移的距离。 `App.ScreenWidth` 和 `App.ScreenHeight` 属性提供的高度和宽度的视区，并通过各自的特定于平台的项目设置为屏幕宽度和设备的屏幕高度值。通过设置然后素数预包装的用户元素及其 `TranslationX` 和 `TranslationY` 计算出的值的属性。

当平移中未占据整个屏幕的元素的内容，可以从该元素的获取的高度和宽度的视区 `Height` 并 `Width` 属性。

> **NOTE**
>
> 显示高分辨率图像可以极大地提高应用程序的内存占用量。因此，它们应仅创建所需和应应用不再需要它们时，就立即释放时。有关详细信息，请参阅优化图像资源。

相关链接

- PanGesture（示例）
- GestureRecognizer
- PanGestureRecognizer

添加轻扫手势识别器

2018/10/26 • • Edit Online

轻扫手势手指在水平或垂直方向，在屏幕上移动，并且通常用于初始化通过内容导航时发生。在本文中的代码示例摘自轻扫手势示例。

若要使 `View` 识别轻扫手势，创建 `SwipeGestureRecognizer` 实例，设置 `Direction` 属性设置为 `SwipeDirection` 枚举值（`Left`、`Right`、`Up`，或 `Down`），根据需要设置 `Threshold` 属性、句柄 `Swiped` 事件，并添加到新的笔势识别器 `GestureRecognizers` 在视图上的集合。下面的代码示例演示 `SwipeGestureRecognizer` 附加到 `BoxView`：

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>
```

下面是等效的C#代码：

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
```

`SwipeGestureRecognizer` 类还包括 `Threshold` 属性，可根据需要设置为 `uint` 值，该值表示为而必须实现的最小轻扫距离轻扫，若要在设备无关的单位中识别。此属性的默认值为 100，也就是说，任何扫的少于 100 个与设备无关单位将被忽略。

识别轻扫方向

在上面的示例 `Direction` 属性设置为单个值从 `SwipeDirection` 枚举。但是，还有可能要将此属性设置为从多个值 `SwipeDirection` 枚举，以便 `Swiped` 事件激发以响应在多个方向轻扫。但是，约束是单个 `SwipeGestureRecognizer` 只能识别扫发生在同一个轴上。因此，可以通过设置识别发生在水平轴的扫 `Direction` 属性设置为 `Left` 和 `Right`：

```
<SwipeGestureRecognizer Direction="Left,Right" Swiped="OnSwiped"/>
```

同样，可以通过设置识别垂直轴发生的扫 `Direction` 属性设置为 `Up` 和 `Down`：

```
var swipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Up | SwipeDirection.Down };
```

或者，`SwipeGestureRecognizer` 为每个轻扫可以创建方向识别扫中每个方向：

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Right" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Up" Swiped="OnSwiped"/>
        <SwipeGestureRecognizer Direction="Down" Swiped="OnSwiped"/>
    </BoxView.GestureRecognizers>
</BoxView>
```

下面是等效的C#代码：

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left };
leftSwipeGesture.Swiped += OnSwiped;
var rightSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Right };
rightSwipeGesture.Swiped += OnSwiped;
var upSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Up };
upSwipeGesture.Swiped += OnSwiped;
var downSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Down };
downSwipeGesture.Swiped += OnSwiped;

boxView.GestureRecognizers.Add(leftSwipeGesture);
boxView.GestureRecognizers.Add(rightSwipeGesture);
boxView.GestureRecognizers.Add(upSwipeGesture);
boxView.GestureRecognizers.Add(downSwipeGesture);
```

NOTE

在上述示例中，同一个事件处理程序响应 `Swiped` 事件触发。但是，每个 `SwipeGestureRecognizer` 实例，如果需要，也可以使用不同的事件处理程序。

轻扫响应

事件处理程序 `Swiped` 事件在下面的示例所示：

```
void OnSwiped(object sender, SwipedEventArgs e)
{
    switch (e.Direction)
    {
        case SwipeDirection.Left:
            // Handle the swipe
            break;
        case SwipeDirection.Right:
            // Handle the swipe
            break;
        case SwipeDirection.Up:
            // Handle the swipe
            break;
        case SwipeDirection.Down:
            // Handle the swipe
            break;
    }
}
```

`SwipedEventArgs` 可以检查以确定方向轻扫，使用响应所需轻扫的自定义逻辑。可以从获取方向轻扫 `Direction` 属性的事件自变量，都将设置为的值之一 `SwipeDirection` 枚举。此外，事件参数还具有 `Parameter` 将设置为的值的属性 `CommandParameter` 属性，如果定义。

使用命令

`SwipeGestureRecognizer` 类还包括 `Command` 并 `CommandParameter` 属性。使用模型-视图-视图模型 (MVVM) 模式的应用程序中通常使用这些属性。 `Command` 属性定义 `ICommand` 轻扫手势识别时，要调用与 `CommandParameter` 属性定义的对象传递给 `ICommand.` 下面的代码示例演示如何将绑定 `Command` 属性设置为 `ICommand` 其实例设置为网页的视图模型中定义 `BindingContext` ：

```
var boxView = new BoxView { Color = Color.Teal, ... };
var leftSwipeGesture = new SwipeGestureRecognizer { Direction = SwipeDirection.Left, CommandParameter = "Left"
};
leftSwipeGesture.SetBinding(SwipeGestureRecognizer.CommandProperty, "SwipeCommand");
boxView.GestureRecognizers.Add(leftSwipeGesture);
```

是等效的 XAML 代码：

```
<BoxView Color="Teal" ...>
    <BoxView.GestureRecognizers>
        <SwipeGestureRecognizer Direction="Left" Command="{Binding SwipeCommand}" CommandParameter="Left" />
    </BoxView.GestureRecognizers>
</BoxView>
```

`SwipeCommand` 是类型的属性 `ICommand` 设置为网页的视图模型实例中定义 `BindingContext` 。轻扫手势识别时，`Execute` 方法的 `SwipeCommand` 将执行对象。参数 `Execute` method 的值是 `CommandParameter` 属性。有关命令的详细信息，请参阅命令界面。

正在创建轻扫容器

`SwipeContainer` 类，下面的代码示例中所示，是将一个通用轻扫识别类封装 `View` 执行轻扫手势识别：

```
public class SwipeContainer : ContentView
{
    public event EventHandler<SwipedEventArgs> Swipe;

    public SwipeContainer()
    {
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Left));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Right));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Up));
        GestureRecognizers.Add(GetSwipeGestureRecognizer(SwipeDirection.Down));
    }

    SwipeGestureRecognizer GetSwipeGestureRecognizer(SwipeDirection direction)
    {
        var swipe = new SwipeGestureRecognizer { Direction = direction };
        swipe.Swiped += (sender, e) => Swipe?.Invoke(this, e);
        return swipe;
    }
}
```

`SwipeContainer` 类创建 `SwipeGestureRecognizer` 对象的所有四个轻扫方向，并将附加 `Swipe` 事件处理程序。这些事件处理程序调用 `Swipe` 事件定义的 `SwipeContainer` 。

以下 XAML 代码示例所示 `SwipeContainer` 类包装 `BoxView` ：

```
<ContentPage ...>
    <StackLayout>
        <local:SwipeContainer Swipe="OnSwiped" ...>
            <BoxView Color="Teal" ... />
        </local:SwipeContainer>
    </StackLayout>
</ContentPage>
```

下面的代码示例演示如何 `SwipeContainer` 包装 `BoxView` 在C#页：

```
public class SwipeContainerPageCS : ContentPage
{
    public SwipeContainerPageCS()
    {
        var boxView = new BoxView { Color = Color.Teal, ... };
        var swipeContainer = new SwipeContainer { Content = boxView, ... };
        swipeContainer.Swipe += (sender, e) =>
        {
          // Handle the swipe
        };

        Content = new StackLayout
        {
            Children = { swipeContainer }
        };
    }
}
```

当 `BoxView` 接收轻扫手势 `Swiped` 中的事件 `SwipeGestureRecognizer` 激发。这将由 `SwipeContainer` 类，该类会触发其自己 `Swipe` 事件。这 `Swipe` 页面上处理事件。 `SwipedEventArgs` 之后进行检查以确定方向轻扫，使用响应所需轻扫的自定义逻辑。

相关链接

- 轻扫手势（示例）
- GestureRecognizer
- SwipeGestureRecognizer

Xamarin.Forms 本地化

2018/11/10 • • Edit Online

可以使用内置的.NET 本地化框架构建跨平台使用 Xamarin.Forms 的多语言应用程序。

字符串和映像本地化

用于本地化.NET 应用程序使用的内置机制RESX 文件中的类和 `System.Resources` 和 `System.Globalization` 命名空间。包含已翻译的字符串的 RESX 文件嵌入在 Xamarin.Forms 程序集，以及编译器生成的类，提供强类型化访问翻译。然后可以在代码中检索已翻译的文本。

从右到左本地化

流方向是密切关注扫描的页上的 UI 元素的方向。从右到左本地化将对从右到左的流方向的支持添加到 Xamarin.Forms 应用程序。

本地化

2018/11/13 • • Edit Online

可以使用.NET 资源文件本地化 Xamarin.Forms 应用。

概述

用于本地化.NET 应用程序使用的内置机制RESX 文件中的类和 `System.Resources` 和 `System.Globalization` 命名空间。包含已翻译的字符串的 RESX 文件嵌入在 Xamarin.Forms 程序集，以及编译器生成的类，提供强类型化访问翻译。然后可以在代码中检索已翻译的文本。

代码示例

有与此文档关联的两个示例：

- UsingResxLocalization介绍的概念非常简单演示。如下所示的代码段都来自此示例。
- TodoLocalized是使用这些本地化技术的基本工作应用。

不建议使用共享的项目

TodoLocalized 示例包括共享项目演示但是，由于生成系统的限制的资源文件不会收到 。**designer.cs**生成的文件，这将导致无法访问在代码中强类型已翻译的字符串。

此文档的其余部分与使用 Xamarin.Forms.NET 标准库模板的项目。

全球化 Xamarin.Forms 代码

全球化应用程序是使其"全球通用。"的过程 这意味着编写能够显示不同的语言的代码。

全球化应用程序的关键部分之一构建用户界面，以便有没有*硬编码*文本。相反，向用户显示的任何内容应检索从一组已被翻译为所选语言的字符串。

本文档中，我们将介绍如何使用 RESX 文件来存储这些字符串和检索它们的显示，具体取决于用户的首选项。

这些示例针对英语、法语、西班牙语、德语、中文、日语、俄语和葡萄牙语（巴西）语言。应用程序可以转换为较多或少所需的语言。

> **NOTE**
> 在通用 Windows 平台上 RESW 文件应该用于推送通知本地化，而不是 RESX 文件。有关详细信息，请参阅UWP 本地化。

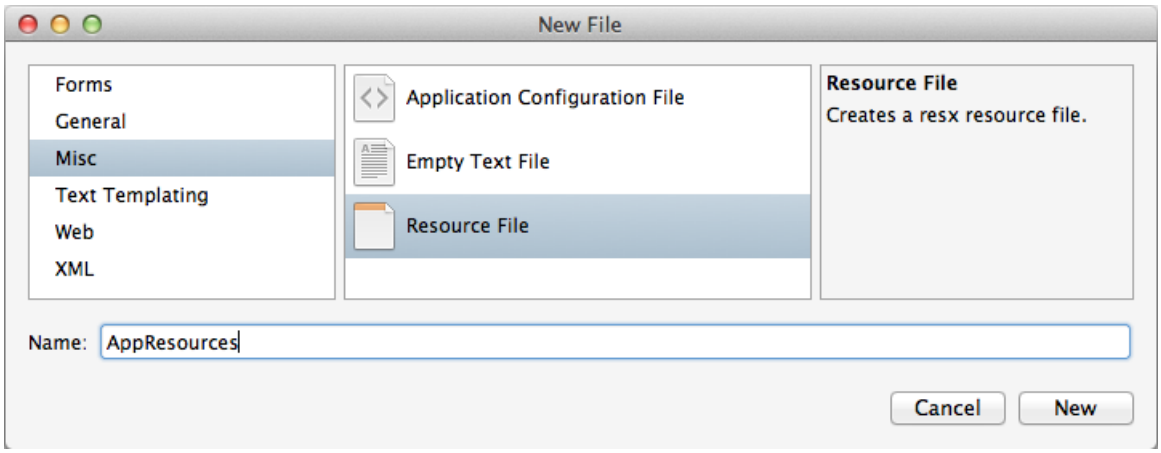
添加资源

全球化 Xamarin.Forms.NET Standard 库应用程序的第一步添加用于将使用的所有文本都存储在应用程序的 RESX 资源文件。我们需要添加包含默认的文本的 RESX 文件，然后添加我们想要支持每种语言的其他 RESX 文件。

基本语言资源

基本资源 (RESX) 文件将包含默认语言字符串（这些示例假定的默认语言是英语）。通过右键单击项目，然后选择将文件添加到 Xamarin.Forms 公共代码项目**添加 > 新建文件...**。

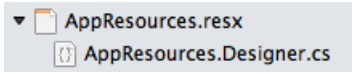
选择有意义的名称，例如**AppResources**然后按**确定**。



两个文件将添加到项目：

- **AppResources.resx**可翻译字符串的 XML 格式的存储位置的文件。
- **AppResources.designer.cs**声明的分部类，以包含对 RESX XML 文件中创建的所有元素的引用的文件。

在解决方案树将显示为相关的文件。RESX 文件应进行编辑，以便添加新的可翻译字符串；。**designer.cs**文件应*不*进行编辑。

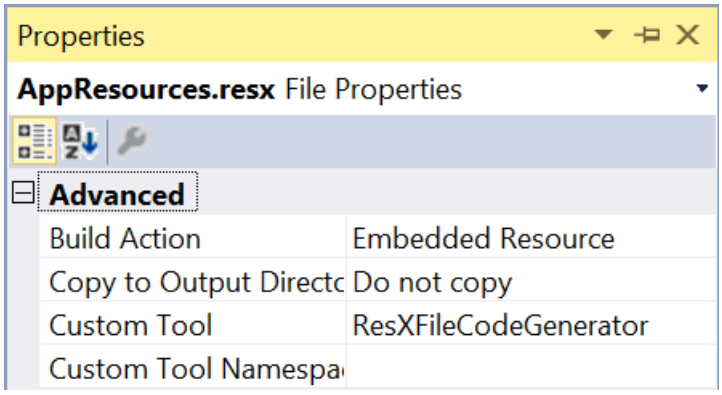


字符串可见性

默认情况下时生成强类型引用为字符串，它们将为 `internal` 对程序集。这是因为 RESX 文件的默认生成工具生成。**designer.cs**文件具有 `internal` 属性。

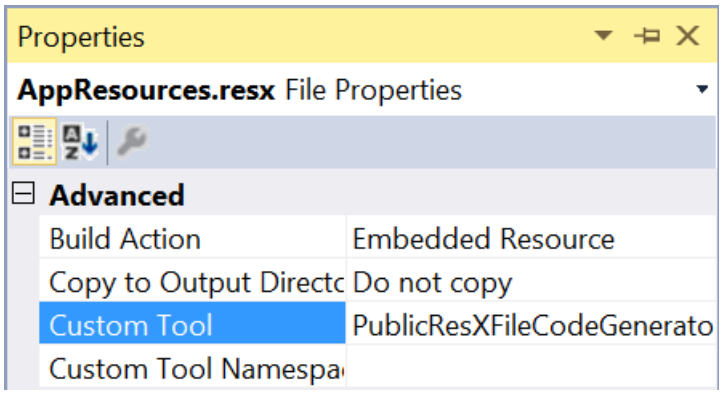
选择**AppResources.resx**文件，并显示**属性**板以查看此生成工具的配置。下面显示的屏幕截图**自定义工具**：**ResXFileCodeGenerator**。

- Visual Studio
- Visual Studio for Mac



若要使强类型化的字符串属性 `public`，你必须手动将配置更改为**自定义工具**：**PublicResXFileCodeGenerator**，如以下屏幕截图中所示：

- Visual Studio
- Visual Studio for Mac



此更改是可选的并仅需要你想要跨不同的程序集引用本地化的字符串（例如，如果将 RESX 文件中的不同程序集放置到你的代码）。此主题的示例使字符串 `internal` 因为它们同一程序集中 Xamarin.Forms.NET 标准库使用位置定义。

只需在基的 RESX 文件上设置自定义工具，如上所示;不需要设置*任何*以下各节所述的特定于语言的 RESX 文件的生成工具。

编辑 RESX 文件

遗憾的是没有任何内置的 RESX 编辑器在 Visual Studio for mac。添加新的可翻译字符串需要新的 XML 添加 `data` 为每个字符串的元素。每个 `data` 元素可以包含以下：

- `name` （必需）的属性是此翻译字符串的键。它必须是有效C#属性名称-以便允许使用任何空格或特殊字符。
- `value` 元素（必需），它是应用程序中显示的实际字符串。
- `comment` 元素（可选）可以包含说明如何使用此字符串翻译的说明。
- `xml:space` （可选）若要控制如何保留在字符串中的间距的属性。

一些示例 `data` 元素如下所示：

```
<data name="NotesLabel" xml:space="preserve">
    <value>Notes:</value>
    <comment>label for input field</comment>
</data>
<data name="NotesPlaceholder" xml:space="preserve">
    <value>eg. buy milk</value>
    <comment>example input for notes field</comment>
</data>
<data name="AddButton" xml:space="preserve">
    <value>Add new item</value>
</data>
```

编写的应用程序，如每个段向用户显示的文本应将添加到基本 RESX 资源文件中的新 `data` 元素。建议您包括 `comment` s 尽可能多地以确保高质量翻译。

> **NOTE**
> Visual Studio（包括免费的 Community 版本）包含基本的 RESX 编辑器。如果你有权访问的 Windows 计算机，可以在方便地添加和编辑 RESX 文件中的字符串。

特定于语言的资源

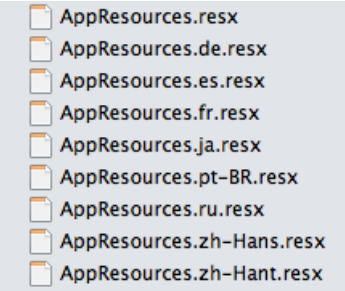
通常情况下，默认文本字符串的实际译文就不会发生之前已写入大型应用程序块（在这种情况下默认 RESX 文件将包含大量字符串）。它仍是在开发周期的早期添加特定于语言的资源（可选）使用的机器翻译文本，以帮助测试本地化代码填充一个好办法。

我们想要支持每种语言添加另一个 RESX 文件。特定于语言的资源文件必须遵循特定的命名约定：基本资源文件（例如使用相同的文件名。**AppResources**）后跟一个句点（.）和语言代码。简单的示例包括：

- **AppResources.fr.resx**的法语语言翻译。
- **AppResources.es.resx** -西班牙语翻译。
- **AppResources.de.resx**的德语语言翻译。
- **AppResources.ja.resx** -日语语言翻译。
- **AppResources.zh Hans.resx** -中文（简体）语言翻译。
- **AppResources.zh Hant.resx** -中文（繁体）语言翻译。
- **AppResources.pt.resx** -葡萄牙语语言翻译。
- **AppResources.pt BR.resx** -巴西葡萄牙语语言翻译。

常规模式是使用两个字母的语言代码，但有（如中文版）的一些示例，其中使用不同的格式，以及（如葡萄牙语（巴西））的其他示例中有四个字符的区域设置标识符是必需。

这些特定于语言的资源文件*不这样做*需要 。**designer.cs**分部类，以便它们可作为常规 XML 文件，添加，使用**生成操作**：**EmbeddedResource**设置。此屏幕截图显示了一个包含特定于语言的资源文件的解决方案：



在应用程序开发和基本的 RESX 文件已添加文本时，您应它发送到每个转换的转换器 `data` 元素并返回一个特定于语言的资源文件（使用显示的命名约定）包含在应用程序中。机器翻译的一些示例如下所示：

AppResources.es.resx（西班牙语）

```
<data name="AddButton" xml:space="preserve">
    <value>Escribir un artículo</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

AppResources.ja.resx（日语）

```
<data name="AddButton" xml:space="preserve">
    <value>新しい項目を追加</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

AppResources.pt BR.resx（巴西葡萄牙语）

```
<data name="AddButton" xml:space="preserve">
    <value>adicionar novo item</value>
    <comment>this string appears on a button to add a new item to the list</comment>
</data>
```

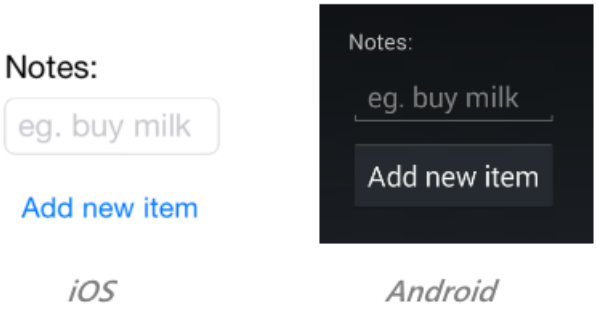
仅 `value` 元素需要更新通过转换器- `comment` 不能进行转换。请记住：编辑 XML 文件进行转义保留的字符喜欢 `<`，`>`，`&` 与 `<`， `>`，并且 `&` 如果它们出现在 `value` 或 `comment` 。

在代码中使用的资源

RESX 资源文件中的字符串可用于在用户界面代码中使用 `AppResources` 类。 `name` 分配给每个字符串在 RESX 文件将成为该类，如下所示，可以在 Xamarin.Forms 代码中引用的属性：

```
var myLabel = new Label ();
var myEntry = new Entry ();
var myButton = new Button ();
// populate UI with translated text values from resources
myLabel.Text = AppResources.NotesLabel;
myEntry.Placeholder = AppResources.NotesPlaceholder;
myButton.Text = AppResources.AddButton;
```

在 iOS、Android 和通用 Windows 平台 (UWP) 将呈现为你的用户界面所期望的只不过现在就可以将翻译为多种语言的应用，因为正在从资源加载文本，而不是硬编码。下面是在转换之前每个平台上显示 UI 的屏幕截图：



疑难解答

测试特定语言

它可能比较棘手，若要在模拟器或设备切换到不同的语言，尤其是在开发时想要快速测试不同的区域性。

您可以强制加载的设置的特定语言 `Culture` 中此代码片段所示：

```
// force a specific culture, useful for quick testing
AppResources.Culture =  new CultureInfo("fr-FR");
```

这种方法 – 上直接设置区域性 `AppResources` 类 – 也可用于实现在您的应用程序（而不是使用设备的区域设置）语言选择器。

加载嵌入资源

下面的代码段时，尝试调试嵌入的资源（如 RESX 文件）的问题。将此代码添加到你的应用（应用程序生命周期中及早），将会列出嵌入程序集中，显示完整的资源标识符的所有资源：

```
using System.Reflection;
// ...
// NOTE: use for debugging, not in released app code!
var assembly = typeof(EmbeddedImages).GetTypeInfo().Assembly; // "EmbeddedImages" should be a class in your
app
foreach (var res in assembly.GetManifestResourceNames())
{
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}
```

在中**AppResources.Designer.cs**文件 (围绕*第 33 行*)，完整*资源管理器名称*(例如指定。`"UsingResxLocalization.Resx.AppResources"`) 类似于下面的代码：

```
System.Resources.ResourceManager temp =
        new System.Resources.ResourceManager(
                "UsingResxLocalization.Resx.AppResources",
                typeof(AppResources).GetTypeInfo().Assembly);
```

检查应用程序输出的结果的上面显示的调试代码，以确认正确列出了资源（即

`"UsingResxLocalization.Resx.AppResources"`).

如果没有，则 `AppResources` 类将不能加载其资源。检查以下项目，其中资源找不到解决的问题：

- 项目的默认命名空间匹配中的根命名空间**AppResources.Designer.cs**文件。
- 如果**AppResources.resx**文件位于一个子目录，则子目录名称应为命名空间的一部分*和*资源标识符的一部分。
- **AppResources.resx**文件具有**生成操作**：**EmbeddedResource**。
- 项目选项 **> 源代码 >.NET 命名策略 > 使用 Visual Studio 样式资源名称**勾选了。如果您愿意可以 untick 这，但是引用 RESX 资源时使用的命名空间将需要更新整个应用。

不能在调试模式 (仅限 Android)

如果使用已翻译的字符串在你的 Android 版本生成，但不是进行调试时，右键单击**Android 项目**，然后选择选项 **> 生成 > Android 生成并确保快速程序集部署**不勾选了。此选项会导致加载资源出现问题，如果要测试本地化应用程序不应使用。

显示正确的语言

到目前为止，我们探讨了如何编写代码，以便可以提供翻译，但不是如何实际使其显示。Xamarin.Forms 代码可以充分利用。NET 的资源加载正确的语言翻译，但我们需要查询来确定用户所选的语言的每个平台上的操作系统。

由于需要一些特定于平台的代码来获取用户的语言首选项，使用依赖关系服务公开 Xamarin.Forms 应用中的该信息并为每个平台实现。

首先，定义一个接口来公开用户的首选的区域性，类似于下面的代码：

```
public interface ILocalize
{
    CultureInfo GetCurrentCultureInfo ();
    void SetLocale (CultureInfo ci);
}
```

第二种，使用DependencyService Xamarin.Forms 中 `App` 类来调用该接口并将我们 RESX 资源区域性设置为正确的值。请注意，我们无需手动将此值设置为通用 Windows 平台，因为资源框架会自动识别这些平台上所选的语言。

```
if (Device.RuntimePlatform == Device.iOS || Device.RuntimePlatform == Device.Android)
{
    var ci = DependencyService.Get<ILocalize>().GetCurrentCultureInfo();
    Resx.AppResources.Culture = ci; // set the RESX for resource localization
    DependencyService.Get<ILocalize>().SetLocale(ci); // set the Thread for locale-aware methods
}
```

资源 `Culture` 需要应用程序首次加载，以便使用正确的语言字符串时设置。（可选）可能会更新此值根据可能会引发 iOS 或 Android，如果用户更新其语言首选项应用运行时的特定于平台的事件。

为实现 `ILocalize` 接口所示特定于平台的代码下面一节。这些实现充分利用此 `PlatformCulture` 帮助器类：

```
public class PlatformCulture
{
    public PlatformCulture (string platformCultureString)
    {
        if (String.IsNullOrEmpty(platformCultureString))
        {
            throw new ArgumentException("Expected culture identifier", "platformCultureString"); // in C# 6
use nameof(platformCultureString)
        }
        PlatformString = platformCultureString.Replace("_", "-"); // .NET expects dash, not underscore
        var dashIndex = PlatformString.IndexOf("-", StringComparison.Ordinal);
        if (dashIndex > 0)
        {
            var parts = PlatformString.Split('-');
            LanguageCode = parts[0];
            LocaleCode = parts[1];
        }
        else
        {
            LanguageCode = PlatformString;
            LocaleCode = "";
        }
    }
    public string PlatformString { get; private set; }
    public string LanguageCode { get; private set; }
    public string LocaleCode { get; private set; }
    public override string ToString()
    {
        return PlatformString;
    }
}
```

特定于平台的代码

检测要显示哪种语言的代码必须是特定于平台的因为 iOS、Android 和 UWP 所有略有不同的方式公开此信息。有关代码 `ILocalize` 依赖关系服务是提供下面为每个平台，以及其他特定于平台的要求，以确保本地化的文本能够正确显示。

特定于平台的代码还必须处理情况下，操作系统允许用户配置不支持的区域设置标识符。NET 的 `CultureInfo` 类。在这些情况下必须编写自定义代码来检测到不受支持的区域设置进行替代最佳。NET 兼容区域设置。

iOS 应用程序项目

iOS 用户从日期和时间格式设置区域性中分别选择其首选的语言。若要加载了正确的资源进行本地化的 Xamarin.Forms 应用，我们只需查询 `NSLocale.PreferredLanguages` 数组中的第一个元素。

以下实现 `ILocalize` 依赖关系服务应位于 iOS 应用程序项目中。因为 iOS 而不是短划线（这是.NET 标准表示形式）使用下划线代码会将下划线替换实例化之前 `CultureInfo` 类：

```
[assembly:Dependency(typeof(UsingResxLocalization.iOS.Localize))]

namespace UsingResxLocalization.iOS
{
public class Localize : UsingResxLocalization.ILocalize
    {
        public void SetLocale (CultureInfo ci)
        {
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
        public CultureInfo GetCurrentCultureInfo ()
        {
            var netLanguage = "en";
            if (NSLocale.PreferredLanguages.Length > 0)
            {
```

```
                var pref = NSLocale.PreferredLanguages [0];
                netLanguage = iOSToDotnetLanguage(pref);
            }
            // this gets called a lot - try/catch can be expensive so consider caching or something
            System.Globalization.CultureInfo ci = null;
            try
            {
                ci = new System.Globalization.CultureInfo(netLanguage);
            }
            catch (CultureNotFoundException e1)
            {
                // iOS locale not valid .NET culture (eg. "en-ES" : English in Spain)
                // fallback to first characters, in this case "en"
                try
                {
                    var fallback = ToDotnetFallbackLanguage(new PlatformCulture(netLanguage));
                    ci = new System.Globalization.CultureInfo(fallback);
                }
                catch (CultureNotFoundException e2)
                {
                    // iOS language not valid .NET culture, falling back to English
                    ci = new System.Globalization.CultureInfo("en");
                }
            }
            return ci;
        }
        string iOSToDotnetLanguage(string iOSLanguage)
        {
            var netLanguage = iOSLanguage;
            //certain languages need to be converted to CultureInfo equivalent
            switch (iOSLanguage)
            {
                case "ms-MY":   // "Malaysian (Malaysia)" not supported .NET culture
                case "ms-SG":    // "Malaysian (Singapore)" not supported .NET culture
                    netLanguage = "ms"; // closest supported
                    break;
                case "gsw-CH":  // "Schwiizertüütsch (Swiss German)" not supported .NET culture
                    netLanguage = "de-CH"; // closest supported
                    break;
                // add more application-specific cases here (if required)
                // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
        string ToDotnetFallbackLanguage (PlatformCulture platCulture)
        {
            var netLanguage = platCulture.LanguageCode; // use the first part of the identifier (two chars,
usually);
            switch (platCulture.LanguageCode)
            {
                case "pt":
                    netLanguage = "pt-PT"; // fallback to Portuguese (Portugal)
                    break;
                case "gsw":
                    netLanguage = "de-CH"; // equivalent to German (Switzerland) for this app
                    break;
                // add more application-specific cases here (if required)
                // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
    }
}
```

NOTE

`try/catch` 中的块，`GetCurrentCultureInfo` 方法模拟完全匹配找不到，查找接近的匹配项，只需根据语言（区域设置中的字符的第一个块）通常用于区域设置说明符 – 回退行为。

Xamarin.Forms 中，对于某些区域设置中在 iOS 中有效，但不是对应于有效 `CultureInfo` 在.NET 中，并会尝试处理这上面的代码。

例如，iOS设置 > 常规语言&区域屏幕，可设置你的手机语言到英语但区域到西班牙，这会导致的区域设置字符串 `"en-ES"` 。当 `CultureInfo` 创建失败，代码将回退使用只是前两个字母来选择显示语言。

开发人员应修改 `iOSToDotnetLanguage` 和 `ToDotnetFallbackLanguage` 方法来处理其支持的语言所需的特定用例。

有一些系统定义的用户界面元素的自动转换为 iOS，如完成按钮 `Picker` 控件。若要强制 iOS 这些元素需要我们指明我们在支持哪种语言翻译**Info.plist**文件。可以添加这些值通过**Info.plist > 源**如下所示：

▼ Localizations	Array	(9 items)
	String	de
	String	es
	String	fr
	String	ja
	String	pt
	String	pt-PT
	String	ru
	String	zh-Hans
	String	zh-Hant
Add new entry		
Localization native deve	String	en

或者，打开**Info.plist**文件在 XML 编辑器中，并直接编辑的值：

```
<key>CFBundleLocalizations</key>
<array>
    <string>de</string>
    <string>es</string>
    <string>fr</string>
    <string>ja</string>
    <string>pt</string> <!-- Brazil -->
    <string>pt-PT</string> <!-- Portugal -->
    <string>ru</string>
    <string>zh-Hans</string>
    <string>zh-Hant</string>
</array>
<key>CFBundleDevelopmentRegion</key>
<string>en</string>
```

实现依赖关系服务和更新后**Info.plist**，iOS 应用程序将能够显示本地化的文本。

NOTE

请注意，Apple 将葡萄牙语稍有不同于您所料。从其 docs: "*pt 作为语言 ID 为使用葡萄牙语因为它用于在巴西和 PT-PT- 语言 ID 为葡萄牙语因为它会用葡萄牙*"。这意味着当葡萄牙语语言选择在非标准的区域设置的回退语言将葡萄牙语（巴西）在 iOS 上，除非编写代码来更改此行为 (如 `ToDotnetFallbackLanguage` 上面)。

有关 iOS 本地化的详细信息，请参阅iOS 本地化。

Android 应用程序项目

Android 公开通过当前所选的区域设置 `Java.Util.Locale.Default`，并且还使用下划线分隔符而不是短划线（它将替换下面的代码）。将此依赖关系服务实现添加到 Android 应用程序项目：

```
[assembly:Dependency(typeof(UsingResxLocalization.Android.Localize))]

namespace UsingResxLocalization.Android
{
    public class Localize : UsingResxLocalization.ILocalize
    {
        public void SetLocale(CultureInfo ci)
        {
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
        public CultureInfo GetCurrentCultureInfo()
        {
            var netLanguage = "en";
            var androidLocale = Java.Util.Locale.Default;
            netLanguage = AndroidToDotnetLanguage(androidLocale.ToString().Replace("_", "-"));
            // this gets called a lot - try/catch can be expensive so consider caching or something
            System.Globalization.CultureInfo ci = null;
            try
            {
                ci = new System.Globalization.CultureInfo(netLanguage);
            }
            catch (CultureNotFoundException e1)
            {
                // iOS locale not valid .NET culture (eg. "en-ES" : English in Spain)
                // fallback to first characters, in this case "en"
                try
                {
                    var fallback = ToDotnetFallbackLanguage(new PlatformCulture(netLanguage));
                    ci = new System.Globalization.CultureInfo(fallback);
                }
                catch (CultureNotFoundException e2)
                {
                    // iOS language not valid .NET culture, falling back to English
                    ci = new System.Globalization.CultureInfo("en");
                }
            }
            return ci;
        }
        string AndroidToDotnetLanguage(string androidLanguage)
        {
            var netLanguage = androidLanguage;
            //certain languages need to be converted to CultureInfo equivalent
            switch (androidLanguage)
            {
                case "ms-BN":   // "Malaysian (Brunei)" not supported .NET culture
                case "ms-MY":   // "Malaysian (Malaysia)" not supported .NET culture
                case "ms-SG":   // "Malaysian (Singapore)" not supported .NET culture
                    netLanguage = "ms"; // closest supported
                    break;
                case "in-ID":  // "Indonesian (Indonesia)" has different code in  .NET
                    netLanguage = "id-ID"; // correct code for .NET
                    break;
                case "gsw-CH":  // "Schwiizertüütsch (Swiss German)" not supported .NET culture
                    netLanguage = "de-CH"; // closest supported
                    break;
                    // add more application-specific cases here (if required)
                    // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
        string ToDotnetFallbackLanguage(PlatformCulture platCulture)
        {
            var netLanguage = platCulture.LanguageCode; // use the first part of the identifier (two chars,
```

```
usually);
            switch (platCulture.LanguageCode)
            {
                case "gsw":
                    netLanguage = "de-CH"; // equivalent to German (Switzerland) for this app
                    break;
                    // add more application-specific cases here (if required)
                    // ONLY use cultures that have been tested and known to work
            }
            return netLanguage;
        }
    }
}
```

> **NOTE**
>
> `try/catch` 中的块，`GetCurrentCultureInfo` 方法模拟完全匹配找不到，查找接近的匹配项，只需根据语言（区域设置中的字符的第一个块）通常用于区域设置说明符 – 回退行为。
>
> Xamarin.Forms 中，对于某些区域设置在 Android 中有效，但不是对应于有效 `CultureInfo` 在.NET 中，并会尝试处理这上面的代码。
>
> 开发人员应修改 `iOSToDotnetLanguage` 和 `ToDotnetFallbackLanguage` 方法来处理其支持的语言所需的特定用例。

此代码添加到 Android 应用程序项目后，它将能够自动显示已翻译的字符串。

> **NOTE**
>
> **警告**：如果使用已翻译的字符串在你的 Android 版本生成，但不是进行调试时，右键单击**Android 项目**，然后选择**选项 > 生成 > Android构建**，并确保**快速程序集部署**不勾选了。此选项会导致加载资源出现问题，如果要测试本地化应用程序不应使用。

有关 Android 本地化的详细信息，请参阅Android 本地化。

通用 Windows 平台

通用 Windows 平台 (UWP) 项目不需要依赖关系服务。相反，此平台自动资源的区域性设置正确。

AssemblyInfo.cs

展开.NET Standard 库项目中的属性节点并双击**AssemblyInfo.cs**文件。将以下行添加到要将非特定语言资源程序集语言设置为英语的文件：

```
[assembly: NeutralResourcesLanguage("en")]
```

这会通知资源管理器的应用程序的默认区域性，因此确保语言中性 RESX 文件中定义的字符串 (**AppResources.resx**) 应用程序运行在一个英语区域设置时，会显示。

示例

更新的特定于平台的项目如下所示更高版本和已翻译的 RESX 文件重新编译应用程序之后，将可以在每个应用更新后的翻译。下面是示例代码转换为简体中文屏幕截图：

有关 UWP 本地化的详细信息，请参阅UWP 本地化。

本地化 XAML

当生成标记的 XAML 中的 Xamarin.Forms 用户界面将类似于以下形式，与字符串嵌入直接在 XML 中：

```
<Label Text="Notes:" />
<Entry Placeholder="eg. buy milk" />
<Button Text="Add to list" />
```

理想情况下，我们无法将直接在 XAML 中，我们可以通过创建来执行此操作中的用户界面控件转换*标记扩展*。公开 RESX 资源到 XAML 标记扩展的代码如下所示。此类应添加到 Xamarin.Forms 常见代码（以及 XAML 页面中）：

```
using System;
using System.Globalization;
using System.Reflection;
using System.Resources;
using Xamarin.Forms;
using Xamarin.Forms.Xaml;

namespace UsingResxLocalization
{
    // You exclude the 'Extension' suffix when using in XAML
    [ContentProperty("Text")]
    public class TranslateExtension : IMarkupExtension
    {
        readonly CultureInfo ci = null;
        const string ResourceId = "UsingResxLocalization.Resx.AppResources";

        static readonly Lazy<ResourceManager> ResMgr = new Lazy<ResourceManager>(
            () => new ResourceManager(ResourceId,
IntrospectionExtensions.GetTypeInfo(typeof(TranslateExtension)).Assembly));

        public string Text { get; set; }

        public TranslateExtension()
        {
            if (Device.RuntimePlatform == Device.iOS || Device.RuntimePlatform == Device.Android)
            {
                ci = DependencyService.Get<ILocalize>().GetCurrentCultureInfo();
            }
        }

        public object ProvideValue(IServiceProvider serviceProvider)
        {
            if (Text == null)
                return string.Empty;

            var translation = ResMgr.Value.GetString(Text, ci);
            if (translation == null)
            {
#if DEBUG
                throw new ArgumentException(
                    string.Format("Key '{0}' was not found in resources '{1}' for culture '{2}'.", Text,
ResourceId, ci.Name),
                    "Text");
#else
                translation = Text; // HACK: returns the key, which GETS DISPLAYED TO THE USER
#endif
            }
            return translation;
        }
    }
}
```

下面的列表内容说明了上面的代码中的重要元素：

- 此类命名 `TranslateExtension`，但按照惯例，我们可以引用是作为**Translate**我们标记中。
- 此类应实现 `IMarkupExtension`，所需为其 Xamarin.Forms 工作。
- `"UsingResxLocalization.Resx.AppResources"` 是我们 RESX 的资源的资源标识符。它是我们默认命名空间、资源文件所在的文件夹和默认的 RESX 文件名组成。
- `ResourceManager` 使用创建类 `IntrospectionExtensions.GetTypeInfo(typeof(TranslateExtension)).Assembly)` 若要确定当前的程序集将加载资源，并缓存在静态 `ResMgr` 字段。它将作为创建 `Lazy` 类型，以便其创建推迟到中的第一次使用 `ProvideValue` 方法。
- `ci` 使用依赖关系服务从本机操作系统中获取用户的所选的语言。
- `GetString` 是从资源文件中检索实际的已翻译的字符串的方法。在通用 Windows 平台上，`ci` 将为 null 因为

`ILocalize` 这些平台上不实现接口。这相当于调用 `GetString` 仅带第一个参数的方法。相反，资源框架会自动识别区域设置，将从相应的 RESX 文件中检索的已翻译的字符串。

- 错误处理已包括在内，以帮助调试缺少的资源通过引发异常 (在 `DEBUG` 仅模式下)。

以下 XAML 代码段演示如何使用标记扩展。有两个步骤以使其正常运行：

1. 声明自定义 `xmlns:i18n` 命名空间的根节点中。 `namespace` 和 `assembly` 完全-在此示例中它们完全相同，但可能不同项目中的项目设置必须匹配。
2. 使用 `{Binding}` 通常会包含要调用的文本的属性的语法 `Translate` 标记扩展。资源键是唯一的必需的参数。

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        x:Class="UsingResxLocalization.FirstPageXaml"
        xmlns:i18n="clr-namespace:UsingResxLocalization;assembly=UsingResxLocalization">
    <StackLayout Padding="0, 20, 0, 0">
        <Label Text="{i18n:Translate NotesLabel}" />
        <Entry Placeholder="{i18n:Translate NotesPlaceholder}" />
        <Button Text="{i18n:Translate AddButton}" />
    </StackLayout>
</ContentPage>
```

下面更详细的语法也是有效的标记扩展：

```
<Button Text="{i18n:TranslateExtension Text=AddButton}" />
```

在本地化特定于平台的元素

虽然我们可以处理的 Xamarin.Forms 代码中的用户界面转换，但是有一些元素必须在每个特定于平台的项目中完成的。本部分将介绍如何本地化：

- Application Name
- 图像

示例项目包括名为本地化的图像**flag.png**中，引用了C#，如下所示：

```
var flag = new Image();
switch (Device.RuntimePlatform)
{
    case Device.iOS:
    case Device.Android:
        flag.Source = ImageSource.FromFile("flag.png");
        break;
    case Device.UWP:
        flag.Source = ImageSource.FromFile("Assets/Images/flag.png");
        break;
}
```

标志图像也引用中 XAML 如下：

```
<Image>
  <Image.Source>
    <OnPlatform x:TypeArguments="ImageSource">
        <On Platform="iOS, Android" Value="flag.png" />
        <On Platform="UWP" Value="Assets/Images/flag.png" />
    </OnPlatform>
  </Image.Source>
</Image>
```

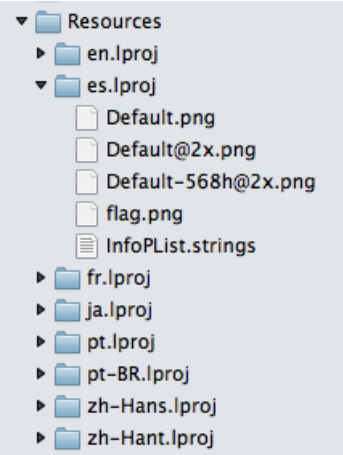
只要实现如下所述的项目结构，所有平台将自动都解析到本地化版本的映像，这些图像引用。

iOS 应用程序项目

iOS 使用某一命名标准称为本地化项目或 **.lproj**目录，用于包含图像和字符串资源。这些目录可以包含在应用中，使用的图像的本地化的版本以及**InfoPlist.strings**可用于本地化的应用程序名称的文件。有关 iOS 本地化的详细信息，请参阅iOS 本地化。

图像

此屏幕截图显示了具有特定于语言的 iOS 示例应用程序 **.lproj**目录。西班牙语目录称为**es.lproj**，包含默认图像的本地化的版本，以及**flag.png**:



每个语言目录包含一份**flag.png**，该语言的本地化。如果提供没有图像，则操作系统将默认为默认语言的目录中的图像。有关完整的 Retina 支持，应提供 **@2x** 并 **@3x** 每个映像的副本。

应用程序名称

内容**InfoPlist.strings**只是一个单个键-值来配置应用程序名称：

```
"CFBundleDisplayName" = "ResxEspañol";
```

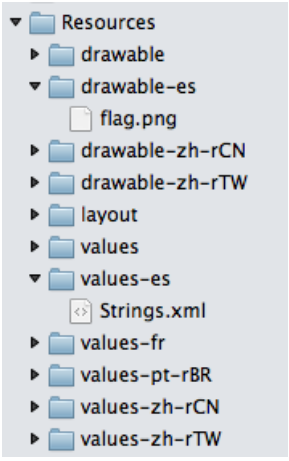
运行应用程序时，应用程序名称和映像已同时本地化：

Android 应用程序项目

Android 遵循不同的方案用于存储使用不同的本地化的图像**drawable并字符串**带有语言代码后缀的目录。当需要（例如 ZH-TW 或 PT）的四个字母区域设置代码时，请注意，Android 需要额外**r前面** dash/以下区域设置代码（例如。zh rTW 或 pt rBR）。有关 Android 本地化的详细信息，请参阅Android 本地化。

图像

此屏幕截图显示了 Android 示例某些本地化的绘图和字符串：



请注意，Android 不使用-Zh-hans 和简体和繁体中文; Zh-hant 代码相反，它仅支持特定国家/地区代码 ZH-CN 和 ZH-TW。

若要支持高密度屏幕的不同分辨率的图像，创建与其他语言文件夹 `-*dpi` 后缀，如绘图 **es mdpi**，绘图 **es xdpi**，绘图 **es xxdpi**，等等。请参阅提供的替代 Android 资源有关详细信息。

应用程序名称

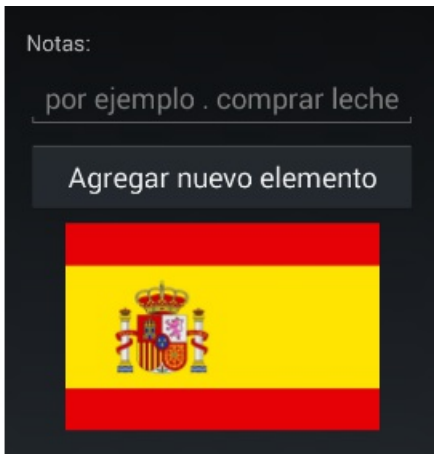
内容**strings.xml**只是一个单个键-值来配置应用程序名称：

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">ResxEspañol</string>
</resources>
```

更新**MainActivity.cs**的 Android 应用程序项目中，以便 `Label` 引用 XML 的字符串。

```
[Activity (Label = "@string/app_name", MainLauncher = true,
        ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation)]
```

现在，应用程序本地化的应用程序名称和图像。下面是结果（在西班牙语）的屏幕截图：



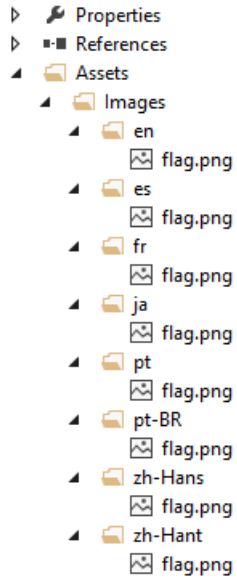
通用 Windows 平台应用程序项目

通用 Windows 平台的强大简化了映像和应用程序名称的本地化的资源基础结构。有关 UWP 本地化的详细信息，

请参阅UWP 本地化。

图像

可以通过将它们放在特定于资源的文件夹中，本地化图像，如以下屏幕截图中所示：



在运行时的 Windows 资源基础结构将选择合适的映像，根据用户的区域设置。

总结

可以使用 RESX 文件和.NET 全球化类本地化 Xamarin.Forms 应用程序。除少量特定于平台的代码来检测用户首选哪种语言，本地化工作的大部分常见的代码中集中管理。

若要充分利用 iOS 和 Android 中提供的多分辨率支持特定于平台的方式通常处理映像。

相关链接

- RESX 本地化示例
- TodoLocalized 示例应用
- 跨平台本地化
- iOS 本地化
- Android 本地化
- UWP 本地化
- 使用 CultureInfo 类 (MSDN)
- 查找并使用为特定区域性 (MSDN) 资源

从右到左本地化

2018/11/14 • • Edit Online

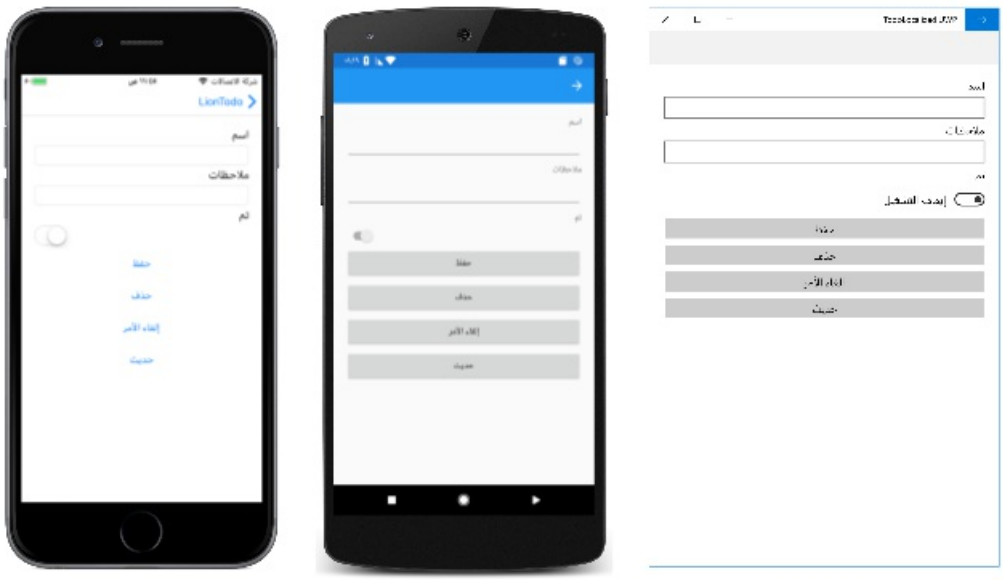
从右到左本地化将对从右到左的流方向的支持添加到 Xamarin.Forms 应用程序。

> **NOTE**
> 从右到左本地化需要 iOS 9 或更高版本，以及 API 17 或更高版本在 Android 上使用。

流方向是密切关注扫描的页上的 UI 元素的方向。某些语言，如阿拉伯语和希伯来语，需要从右到左流动方向进行布局的 UI 元素。这可以通过设置来实现 `VisualElement.FlowDirection` 属性。此属性获取或设置在任何的控制其布局，并应将设置为其中一个父元素的 UI 元素流中的方向 `FlowDirection` 枚举值：

- `LeftToRight`
- `RightToLeft`
- `MatchParent`

设置 `FlowDirection` 属性设置为 `RightToLeft` 元素通常设置的对齐方式向右、向右到左的阅读顺序和控件的布局从流从右到左：



> **TIP**
> 您应该只设置 `FlowDirection` 初始布局的属性。更改此值在运行时将导致一个成本高昂的布局过程，将会影响性能。

默认值 `FlowDirection` 无父元素的属性值是 `LeftToRight` ，而默认 `FlowDirection` 使用的父元素为 `MatchParent` 。因此，一个元素继承 `FlowDirection` 从可视化树中，并且任何元素中其父级的属性值可以重写它从其父级中获取的值。

> **TIP**
> 在本地化右到左的语言的应用时，设置 `FlowDirection` 页或根布局上的属性。这将导致所有页上或根布局，以与流方向做出适当的响应中包含的元素。

并遵循设备的流方向

并遵循设备的流方向基于所选的语言和区域是一个显式的开发人员的选择，而且不会自动发生。它可以通过设置来实现 `FlowDirection` 页上或根布局属性为 `static` `Device.FlowDirection` 值：

```
<ContentPage ... FlowDirection="{x:Static Device.FlowDirection}"> />
```

```
this.FlowDirection = Device.FlowDirection;
```

页上或根布局的所有子元素将默认情况下都继承 `Device.FlowDirection` 值。

平台安装程序

启用从右到左的区域设置需要特定平台安装程序。

iOS

所需的从右向左的区域设置应作为受支持的语言添加到的数组项 `CFBundleLocalizations` 中的键**Info.plist**。下面的示例演示具有已添加到数组中的阿拉伯语 `CFBundleLocalizations` 密钥：

```
<key>CFBundleLocalizations</key>
<array>
    <string>en</string>
    <string>ar</string>
</array>
```

有关详细信息，请参阅在 iOS 中的本地化基础知识。

然后可以通过更改为从右向左的区域设置中指定的语言和设备/模拟器上的区域进行测试从右到左本地化**Info.plist**。

> **WARNING**
> 请注意，当更改语言和区域为从右向左的区域设置在 iOS 上，任何 `DatePicker` 视图将引发异常，如果不包括所需的区域设置的资源。例如，当在阿拉伯语已测试的应用程序 `DatePicker`，确保**mideast**中选择**国际化一部分iOS 生成**窗格。

Android

应用程序的**AndroidManifest.xml**应更新文件，以便 `<uses-sdk>` 的节点集 `android:minSdkVersion` 属性为 17，并 `<application>` 的节点集 `android:supportsRtl` 属性为 `true`：

```
<?xml version="1.0" encoding="utf-8"?>
<manifest ... >
    <uses-sdk android:minSdkVersion="17" ... />
    <application ... android:supportsRtl="true">
    </application>
</manifest>
```

通过更改设备/模拟器以使用从右到左语言中，或者通过启用，然后可以测试从右向左的本地化**Force RTL 布局方向**中设置 **> 开发人员选项**。

通用 Windows 平台 (UWP)

应在中指定所需的语言资源 `<Resources>` 的节点**Package.appxmanifest**文件。下面的示例显示了已加入的阿拉伯语 `<Resources>` 节点：

```
<Resources>
    <Resource Language="x-generate"/>
    <Resource Language="en" />
    <Resource Language="ar" />
</Resources>
```

此外，UWP 需要.NET Standard 库中显式定义应用程序的默认区域性。这可以通过设置来实现 `NeutralResourcesLanguage` 属性中 `AssemblyInfo.cs`，或在另一个类中，为默认区域性：

```
using System.Resources;

[assembly: NeutralResourcesLanguage("en")]
```

更改为相应从右向左的区域设置的语言和设备上的区域，然后可以测试从右向左的本地化。

限制

Xamarin.Forms 从右到左本地化当前具有许多限制：

- `NavigationPage` 按钮位置工具栏项的位置，并转换动画受设备区域设置，而不是 `FlowDirection` 属性。
- `CarouselPage` 轻扫方向不能翻转。
- `Image` 可视内容不能翻转。
- `DisplayAlert` 并 `DisplayActionSheet` 方向受设备区域设置，而不是 `FlowDirection` 属性。
- `WebView` 内容不遵从 `FlowDirection` 属性。
- 一个 `TextDirection` 属性需要要添加，用于控制文本对齐方式。

iOS

- `Stepper` 由设备的区域设置，控制方向而不是 `FlowDirection` 属性。
- `EntryCell` 文本对齐方式受设备区域设置，而不是 `FlowDirection` 属性。
- `ContextActions` 不会逆转手势和对齐方式。

Android

- `SearchBar` 由设备的区域设置，控制方向而不是 `FlowDirection` 属性。
- `ContextActions` 放置受设备区域设置，而不是 `FlowDirection` 属性。

UWP

- `Editor` 文本对齐方式受设备区域设置，而不是 `FlowDirection` 属性。
- `FlowDirection` 属性不会继承 `MasterDetailPage` 子级。
- `ContextActions` 文本对齐方式受设备区域设置，而不是 `FlowDirection` 属性。

从右至左的语言支持使用 Xamarin.University

Xamarin.Forms 3.0 右到左支持，通过Xamarin 学院课程

相关链接

- TodoLocalizedRTL 示例应用

Xamarin.Forms 本地数据库

2018/10/19 • • Edit Online

Xamarin.Forms 支持使用 SQLite 数据库引擎，这使得可以加载并将对象保存在共享代码中的数据库驱动的应用程序。本文介绍了 Xamarin.Forms 应用程序如何读取和写入数据到使用 SQLite.Net 的本地 SQLite 数据库。

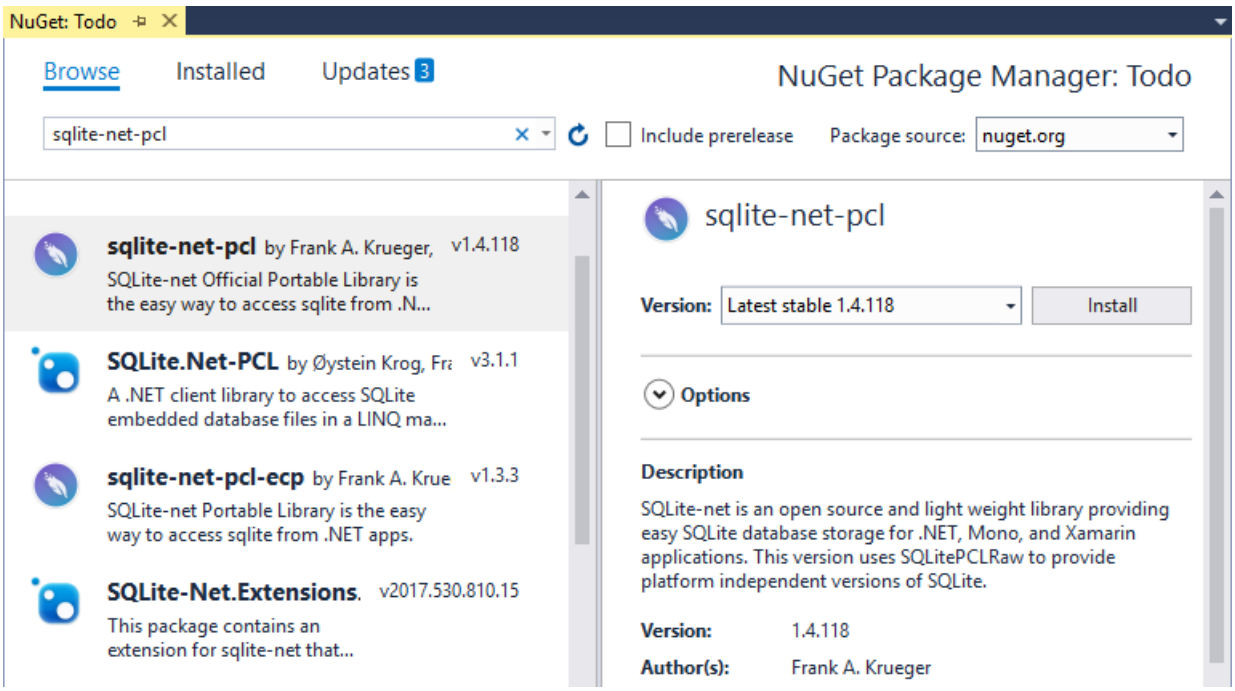
概述

Xamarin.Forms 应用程序可以使用SQLite.NET PCL NuGet程序包来整合数据库操作组合到通过引用共享代码 `SQLite` 在 NuGet 中提供的类。.NET Standard 库项目中的 Xamarin.Forms 解决方案，可以定义数据库操作。

附随示例应用程序是一个简单的待办事项列表应用程序。以下屏幕截图显示此示例每个平台上的显示方式：

使用 SQLite

若要将 SQLite 支持添加到 Xamarin.Forms.NET Standard 库，使用 NuGet 的搜索功能查找**sqlite net pcl**并安装最新的包：



有多种具有类似名称的 NuGet 包，正确的包具有这些属性：

- **创建的**：Frank A.Krueger
- **Id:** sqlite net pcl
- **NuGet 链接**：sqlite net pcl

> **NOTE**
> 尽管包名称，使用**sqlite net pcl**甚至在.NET Standard 项目中的 NuGet 包。

添加引用后的属性添加到 `App` 返回用于存储数据库的本地文件路径的类：

```
static TodoItemDatabase database;

public static TodoItemDatabase Database
{
  get
  {
    if (database == null)
    {
      database = new TodoItemDatabase(
        Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
"TodoSQLite.db3"));
    }
    return database;
  }
}
```

`TodoItemDatabase` 构造函数，使用作为参数的数据库文件的路径，如下所示：

```
public TodoItemDatabase(string dbPath)
{
  database = new SQLiteAsyncConnection(dbPath);
  database.CreateTableAsync<TodoItem>().Wait();
}
```

利用公开为单一实例是在创建单个数据库连接的数据库，保持打开状态时应用程序运行，因此避免销打开和关闭数据库文件每次数据库操作执行。

其余部分 `TodoItemDatabase` 类包含跨平台运行 SQLite 查询。示例查询代码如下所示 (有关语法的更多详细信息可在使用 Xamarin.iOS 使用 SQLite.NET。

```
public Task<List<TodoItem>> GetItemsAsync()
{
  return database.Table<TodoItem>().ToListAsync();
}

public Task<List<TodoItem>> GetItemsNotDoneAsync()
{
  return database.QueryAsync<TodoItem>("SELECT * FROM [TodoItem] WHERE [Done] = 0");
}

public Task<TodoItem> GetItemAsync(int id)
{
  return database.Table<TodoItem>().Where(i => i.ID == id).FirstOrDefaultAsync();
}

public Task<int> SaveItemAsync(TodoItem item)
{
  if (item.ID != 0)
  {
    return database.UpdateAsync(item);
  }
  else {
    return database.InsertAsync(item);
  }
}

public Task<int> DeleteItemAsync(TodoItem item)
{
  return database.DeleteAsync(item);
}
```

NOTE

使用异步 SQLite.Net API 的优点是该数据库操作移到后台线程。此外，没有必要编写其他并发处理代码，因为 API 将处理它。

总结

Xamarin.Forms 支持使用 SQLite 数据库引擎，这使得可以加载并将对象保存在共享代码中的数据库驱动的应用程序。

本文重点访问使用 Xamarin.Forms 的 SQLite 数据库。有关使用 SQLite.Net 本身的详细信息，请参阅在 Android 上的 SQLite.NET或SQLite.NET 在 iOS 上的文档。

相关链接

- 待办事项示例
- Xamarin.Forms 示例

Xamarin.Forms MessagingCenter

2018/11/1 • • Edit Online

Xamarin.Forms 具有简单的消息传送服务以发送和接收消息。

概述

Xamarin.Forms `MessagingCenter` 启用视图模型和其他组件以与通信而无需知道有关彼此的任何除了简单的消息约定。

MessagingCenter 的工作原理

有两个部分 `MessagingCenter`：

- **订阅**-侦听的具有特定签名的消息，并在接收时执行某些操作。多个订阅者可以侦听同一条消息。
- **发送**-发布侦听器来对其执行操作的消息。如果订阅了没有侦听器被忽略该消息。

`MessagingService` 是与一个静态类 `Subscribe` 和 `Send` 在整个解决方案所用的方法。

消息具有一个字符串 `message` 用作方法的参数*地址*消息。 `Subscribe` 并 `Send` 方法使用泛型参数以进一步控制如何传递的消息-两个相同的消息 `message` 文本，但不同的泛型类型参数不会传递到相同的订阅服务器。

有关 API `MessagingCenter` 很简单：

- `Subscribe<TSender> (object subscriber, string message, Action<TSender> callback, TSender source = null)`
- `Subscribe<TSender, TArgs> (object subscriber, string message, Action<TSender, TArgs> callback, TSender source = null)`
- `Send<TSender> (TSender sender, string message)`
- `Send<TSender, TArgs> (TSender sender, string message, TArgs args)`
- `Unsubscribe<TSender, TArgs> (object subscriber, string message)`
- `Unsubscribe<TSender> (object subscriber, string message)`

下面介绍了这些方法。

使用 MessagingCenter

由于用户交互（例如单击按钮）、系统事件（如更改状态的控件）或某些其他事件（如异步下载完成），可能会发送消息。订阅服务器可能在侦听更改的用户界面的外观、将数据保存或触发某个其他操作。

简单的字符串消息

最简单的消息包含只是一个字符串中的 `message` 参数。一个 `Subscribe` 方法的*侦听*简单的字符串消息，显示如下-请注意，指定应发件人为的类型的泛型类型 `MainPage` 。使用此语法的消息可以订阅该解决方案中的任何类：

```
MessagingCenter.Subscribe<MainPage> (this, "Hi", (sender) => {
    // do something whenever the "Hi" message is sent
});
```

在中 `MainPage` 类的以下代码发送消息。 `this` 参数是的一个实例 `MainPage` 。

```
MessagingCenter.Send<MainPage> (this, "Hi");
```

它不会更改字符串-指示*消息类型*和用于确定哪些订阅服务器，以通知。这种消息用来指示发生某些事件，但"上传已完成"，例如需要任何进一步的信息的。

将参数传递

与消息的自变量传递，指定自变量类型中 `Subscribe` 泛型参数和操作签名中。

```
MessagingCenter.Subscribe<MainPage, string> (this, "Hi", (sender, arg) => {
    // do something whenever the "Hi" message is sent
    // using the 'arg' parameter which is a string
});
```

若要发送的消息的参数，包括类型泛型参数和中的参数的值 `Send` 方法调用。

```
MessagingCenter.Send<MainPage, string> (this, "Hi", "John");
```

此简单示例使用 `string` 自变量，但任何C#对象可以传递。

取消订阅

对象可以随时取消订阅的消息签名，以便不传递任何将来的消息。 `Unsubscribe` 方法语法应反映消息的签名（因此可能需要包括消息参数的泛型类型参数）。

```
MessagingCenter.Unsubscribe<MainPage> (this, "Hi");
MessagingCenter.Unsubscribe<MainPage, string> (this, "Hi");
```

总结

MessagingCenter 是减小耦合度，尤其是视图模型之间的简单方法。它可以用于发送和接收简单消息或类之间传递参数。类应取消订阅它们不再想要接收的消息。

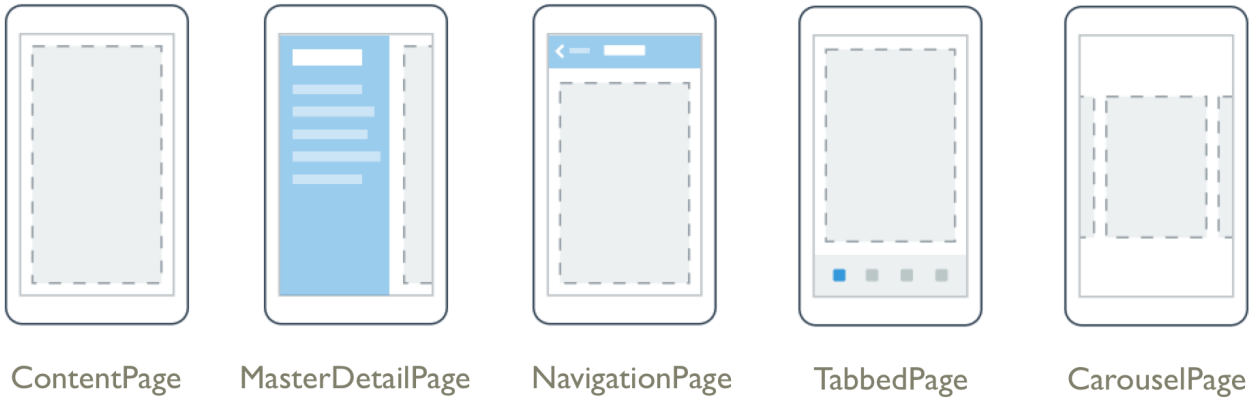
相关链接

- MessagingCenterSample
- Xamarin.Forms 示例

Xamarin.Forms 导航

2018/7/13 • • Edit Online

Xamarin.Forms 提供了多种不同的页导航体验，取决于所使用的页类型。



分层导航

`NavigationPage` 类提供分层导航体验，用户可以随心所欲地向前或向后导航页面。此类将导航实现为 `Page` 对象的后进先出 (LIFO) 堆栈。

TabbedPage

Xamarin.Forms `TabbedPage` 包含具有内容载入的详细信息区域的每个选项卡的选项卡和较大的详细信息区域的列表。

CarouselPage

Xamarin.Forms `CarouselPage` 是一个页面，用户可以向从左到右轻扫来导航页面的内容，例如库。

MasterDetailPage

Xamarin.Forms `MasterDetailPage` 是可管理两个页面的相关信息 – 主页面，其中的项，并显示主页面的项目的详细信息的详细信息页面的页面。

模式页

Xamarin.Forms 还提供支持模式页面。模式页面鼓励用户完成独立任务，在完成或取消该任务之前，不允许导航离开该任务。

显示弹出窗口

Xamarin.Forms 提供了两个弹出注册类似于用户界面元素：警报和操作工作表。要求用户简单的问题和来指导用户完成任务，可以使用这些界面元素。

分层导航

2018/10/25 • • Edit Online

NavigationPage 类提供了可以向前和向后，根据需要通过页导航用户所在的分层导航体验。类实现导航作为后进先出 (LIFO) 堆栈的页对象。本文演示如何使用 NavigationPage 类在大量页面中执行导航。

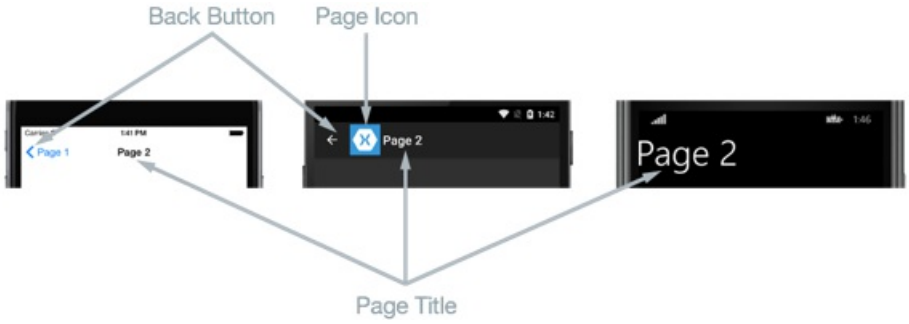
若要从一个页面移动到另一个，应用程序会将新页面推送到导航堆栈中，其中它将成为活动页，如以下关系图中所示：

若要返回到上一页，应用程序将弹出当前页从导航堆栈中，并最顶层的页成为活动页，如以下关系图中所示：

导航方法公开的 `Navigation` 属性上任何 `Page` 派生类型。这些方法提供到导航堆栈中弹出页面推送到导航堆栈上的页面并执行堆栈操作的功能。

执行导航

在分层导航中，使用 `NavigationPage` 类在 `ContentPage` 对象的堆栈内进行导航。以下屏幕截图显示的主要组件 `NavigationPage` 每个平台上：



布局 `NavigationPage` 取决于平台：

- 在 iOS 上，导航栏位于的页面的显示标题，并具有顶部*回*返回到上一页的按钮。
- 在 Android 上，导航栏位于顶部的标题、一个图标，显示的页面和一个*回*返回到上一页的按钮。在中定义的图标 `[Activity]` 修饰的属性 `MainActivity` Android 的特定于平台的项目中的类。
- 在通用 Windows 平台上，导航栏会出现在显示一个标题页的顶部。

在所有平台的值 `Page.Title` 属性将显示为页面标题。

> **NOTE**
> 我们建议 `NavigationPage` 应填充了 `ContentPage` 仅限实例。

创建根页

添加到导航堆栈中的第一页称为应用程序的根页，以下代码示例显示了实现此过程的方法：

```
public App ()
{
  MainPage = new NavigationPage (new Page1Xaml ());
}
```

这将导致 `Page1Xaml` `ContentPage` 实例推送到导航堆栈中，其中它将成为活动页和应用程序的根页面。以下屏幕截图所示：

> **NOTE**
>
> `RootPage` 的属性 `NavigationPage` 实例提供对导航堆栈中的第一页的访问。

将页面推送到导航堆栈

若要导航到 `Page2Xaml`，必须调用 `PushAsync` 方法 `Navigation` 属性的当前页上，如下面的代码示例所示：

```
async void OnNextPageButtonClicked (object sender, EventArgs e)
{
  await Navigation.PushAsync (new Page2Xaml ());
}
```

这会将 `Page2Xaml` 实例推送到导航堆栈中，在堆栈中，它成为活动页。以下屏幕截图所示：

当 `PushAsync` 调用方法时，会发生以下事件：

- 页调用 `PushAsync` 具有其 `OnDisappearing` 重写调用。
- 要导航到页都有其 `OnAppearing` 重写调用。
- `PushAsync` 任务完成。

但是，发生这些事件的确切顺序是依赖于平台。有关详细信息，请参阅第 24 章的 Charles Petzold 的 Xamarin.Forms 书籍。

> **NOTE**
> 调用 `OnDisappearing` 并 `OnAppearing` 重写不能被视为有保证的页面导航的迹象。例如，在 iOS 上，`OnDisappearing` 在应用程序终止时重写在活动页面上调用。

从导航堆栈弹出页

通过设备上的返回按钮（无论是设备上的物理按钮还是屏幕按钮），可以从导航堆栈中弹出活动页。

若要以编程方式返回原始页，`Page2Xaml` 实例必须调用 `PopAsync` 方法，如以下代码示例所示：

```
async void OnPreviousPageButtonClicked (object sender, EventArgs e)
{
  await Navigation.PopAsync ();
}
```

这会从导航堆栈中删除 `Page2Xaml` 实例，而使最顶层的页成为活动页。当 `PopAsync` 调用方法时，会发生以下事件：

- 页调用 `PopAsync` 具有其 `OnDisappearing` 重写调用。
- 返回到页都有其 `OnAppearing` 重写调用。
- `PopAsync` 任务返回。

但是，发生这些事件的确切顺序是依赖于平台。有关详细信息请参阅第 24 章的 Charles Petzold 的 Xamarin.Forms 书籍。

作为 `PushAsync` 并 `PopAsync` 方法，`Navigation` 属性的每一页还提供了 `PopToRootAsync` 方法，在下面的代码示例所示：

```
async void OnRootPageButtonClicked (object sender, EventArgs e)
{
  await Navigation.PopToRootAsync ();
}
```

此方法会弹出要求除根目录之外的所有 `Page` 导航堆栈中，因此，可通过应用程序的活动页面的根页。

对页面过渡效果进行动画处理

`Navigation` 属性的每一页还提供了重写的 push 和 pop 方法，包括 `boolean` 控制是否显示在导航窗格中，一个页面的动画，如下面的代码中所示的参数示例：

```
async void OnNextPageButtonClicked (object sender, EventArgs e)
{
  // Page appearance not animated
  await Navigation.PushAsync (new Page2Xaml (), false);
}

async void OnPreviousPageButtonClicked (object sender, EventArgs e)
{
  // Page appearance not animated
  await Navigation.PopAsync (false);
}

async void OnRootPageButtonClicked (object sender, EventArgs e)
{
  // Page appearance not animated
  await Navigation.PopToRootAsync (false);
}
```

设置 `boolean` 参数 `false` 禁用时将参数设置为的页面过渡动画 `true` 使页面过渡动画，前提是基础平台支持。但是，缺少此参数的 push 和 pop 方法默认情况下启用动画。

导航时传递数据

有时，所需的页导航期间将数据传递给另一页。实现此目的的两种方法将数据通过页的构造函数，并通过设置新页面的传入 `BindingContext` 的数据。每个将现在讨论反过来。

通过 Page 构造函数传递数据

在导航过程将数据传递到另一个页面的最简单方法是通过页构造函数参数，下面的代码示例中所示：

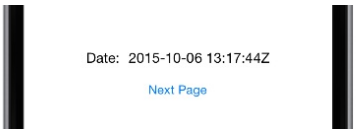
```
public App ()
{
  MainPage = new NavigationPage (new MainPage (DateTime.Now.ToString ("u")));
}
```

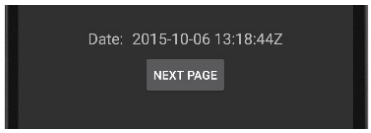
此代码将创建 `MainPage` 实例时，传入当前日期和时间 ISO8601 格式，这包装在 `NavigationPage` 实例。

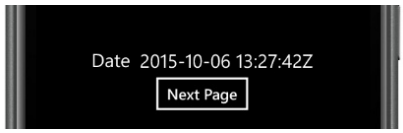
`MainPage` 实例收到的数据通过构造函数参数，如下面的代码示例中所示：

```
public MainPage (string date)
{
  InitializeComponent ();
  dateLabel.Text = date;
}
```

数据将通过设置显示在页面 `Label.Text` 属性，如以下屏幕截图中所示：







BindingContext 间传递数据

用于导航期间将数据传递到另一页一种替代方法是通过设置新页面 `BindingContext` 到数据，如下面的代码示例中所示：

```
async void OnNavigateButtonClicked (object sender, EventArgs e)
{
  var contact = new Contact {
    Name = "Jane Doe",
    Age = 30,
    Occupation = "Developer",
    Country = "USA"
  };

  var secondPage = new SecondPage ();
  secondPage.BindingContext = contact;
  await Navigation.PushAsync (secondPage);
}
```

此代码将设置 `BindingContext` 的 `SecondPage` 实例向 `Contact` 实例，，然后转到 `SecondPage`。

`SecondPage` 然后使用数据绑定来显示 `Contact` 实例数据，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="PassingData.SecondPage"
             Title="Second Page">
    <ContentPage.Content>
        <StackLayout HorizontalOptions="Center" VerticalOptions="Center">
            <StackLayout Orientation="Horizontal">
                <Label Text="Name:" HorizontalOptions="FillAndExpand" />
                <Label Text="{Binding Name}" FontSize="Medium" FontAttributes="Bold" />
            </StackLayout>
            ...
            <Button x:Name="navigateButton" Text="Previous Page" Clicked="OnNavigateButtonClicked" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

下面的代码示例演示如何在 C# 中实现数据绑定：

```
public class SecondPageCS : ContentPage
{
  public SecondPageCS ()
  {
    var nameLabel = new Label {
      FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
      FontAttributes = FontAttributes.Bold
    };
    nameLabel.SetBinding (Label.TextProperty, "Name");
    ...
    var navigateButton = new Button { Text = "Previous Page" };
    navigateButton.Clicked += OnNavigateButtonClicked;

    Content = new StackLayout {
      HorizontalOptions = LayoutOptions.Center,
      VerticalOptions = LayoutOptions.Center,
      Children = {
        new StackLayout {
          Orientation = StackOrientation.Horizontal,
          Children = {
            new Label{ Text = "Name:", FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
HorizontalOptions = LayoutOptions.FillAndExpand },
            nameLabel
          }
        },
        ...
        navigateButton
      }
    };
  }

  async void OnNavigateButtonClicked (object sender, EventArgs e)
  {
    await Navigation.PopAsync ();
  }
}
```

数据将显示在页面的一系列 `Label` 控制，如以下屏幕截图中所示：

若要深入了解数据绑定，请参阅数据绑定基本知识。

操作导航堆栈

`Navigation` 属性公开 `NavigationStack` 可以从中获得导航堆栈中的页的属性。Xamarin.Forms 保留权导航堆栈中，虽然 `Navigation` 属性提供 `InsertPageBefore` 并 `RemovePage` 堆栈操作的方法是插入方法页面或删除它们。

`InsertPageBefore` 方法之前指定一个现有页面上，在导航堆栈中插入指定的页，如以下关系图中所示：

`RemovePage` 方法从中删除指定的页导航堆栈中，如以下关系图中所示：

这些方法启用自定义导航体验，例如登录页替换为新的页上，按照在成功登录。下面的代码示例演示此方案：

```
async void OnLoginButtonClicked (object sender, EventArgs e)
{
  ...
  var isValid = AreCredentialsCorrect (user);
  if (isValid) {
    App.IsUserLoggedIn = true;
    Navigation.InsertPageBefore (new MainPage (), this);
    await Navigation.PopAsync ();
  } else {
    // Login failed
  }
}
```

前提是用户的凭据是否正确，`MainPage` 实例插入到当前页之前的导航堆栈。`PopAsync` 方法然后会从导航堆栈中，当前页删除与 `MainPage` 实例成为活动页。

在导航栏中显示视图

任何 Xamarin.Forms `View` 可以显示在导航栏中的 `NavigationPage` 。这可以通过设置 `NavigationPage.TitleView` 附加到属性 `View` 。此附加的属性可以设置任意 `Page` ，以及何时 `Page` 将被推送到 `NavigationPage` ，则 `NavigationPage` 将遵守属性的值。

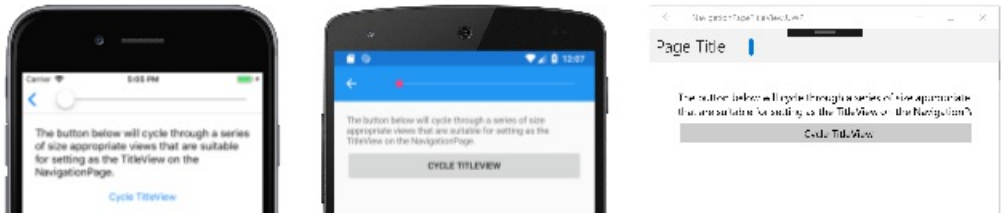
以下示例摘自标题视图示例，演示如何设置 `NavigationPage.TitleView` 从 XAML 附加属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="NavigationPageTitleView.TitleViewPage">
    <NavigationPage.TitleView>
        <Slider HeightRequest="44" WidthRequest="300" />
    </NavigationPage.TitleView>
    ...
</ContentPage>
```

下面是等效的C#代码：

```
public class TitleViewPage : ContentPage
{
    public TitleViewPage()
    {
        var titleView = new Slider { HeightRequest = 44, WidthRequest = 300 };
        NavigationPage.SetTitleView(this, titleView);
        ...
    }
}
```

这会导致 `Slider` 上的导航栏中显示 `NavigationPage` ：



> **IMPORTANT**
>
> 很多视图不会显示在导航栏中，除非使用指定的视图的大小 `WidthRequest` 并 `HeightRequest` 属性。或者，该视图可以包装在 `StackLayout` 与 `HorizontalOptions` 并 `VerticalOptions` 属性设置为适当的值。

注意，因为 `Layout` 类派生自 `View` 类 `TitleView` 可以设置附加的属性来显示布局包含多个视图的类。在 iOS 和通用 Windows 平台 (UWP) 上，不能更改导航栏的高度，并因此会发生剪切，如果在导航栏中显示的视图大小大于默认大小的导航栏。但是，在 Android 上，导航栏的高度可通过设置来更改 `NavigationPage.BarHeight` 可绑定属性设置为 `double` 表示新的高度。有关详细信息，请参阅NavigationPage 上设置导航栏高度。

或者，可以通过将某些内容在导航栏中，而另一些在视图中放置在您的颜色匹配到导航栏的页面内容的顶部建议扩展的导航栏。此外，在 iOS 上的分隔线和位于导航栏底部的卷影可以删除通过设置 `NavigationPage.HideNavigationBarSeparator` 可绑定属性设置为 `true` 。有关详细信息，请参阅隐藏导航栏分隔符在 NavigationPage。

> **NOTE**
>
> `BackButtonTitle` ， `Title` ， `TitleIcon` ，并 `TitleView` 都可以定义属性占用空间，在导航栏上的值。虽然导航栏大小因平台和屏幕大小而异，设置所有这些属性会由于可用的有限空间冲突。而不是尝试使用这些属性的组合，你可能会发现可以更好地实现您所需的导航栏的设计通过仅设置 `TitleView` 属性。

限制

有许多需要注意的显示时的限制 `View` 中的导航栏 `NavigationPage` ：

- 在 iOS 上，视图放在导航栏中的 `NavigationPage` 在不同的位置，具体取决于是否启用了大标题中显示。有关启用大标题的详细信息，请参阅显示大标题。
- 在 Android 上，将视图放置在导航栏中的 `NavigationPage` 只能在使用应用程序兼容性的应用来完成。
- 我们不建议将大型和复杂的视图，如放置 `ListView` 并 `TableView` ，在导航栏中的 `NavigationPage` 。

相关链接

- 页面导航
- 分层（示例）
- PassingData（示例）
- LoginFlow（示例）
- TitleView（示例）
- 如何在 Xamarin.Forms（Xamarin University 视频）示例中的屏幕流中创建一个符号
- 如何在 Xamarin.Forms（Xamarin University 视频）中的屏幕流中创建一个符号
- NavigationPage

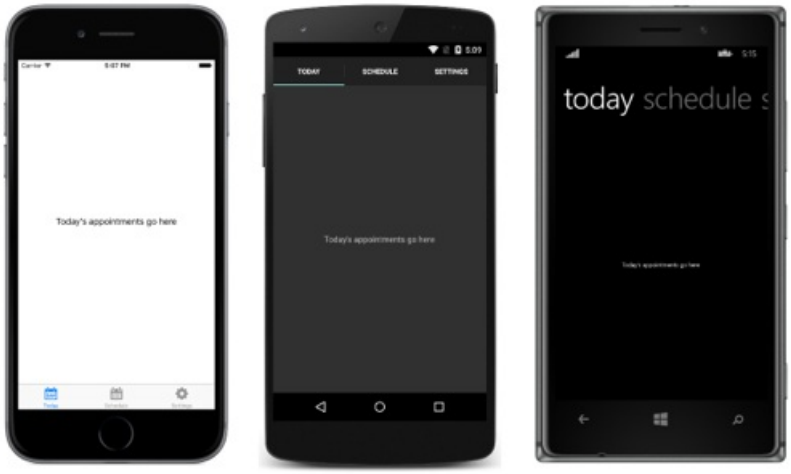
Xamarin.Forms 选项卡式的页面

2018/7/13 • • Edit Online

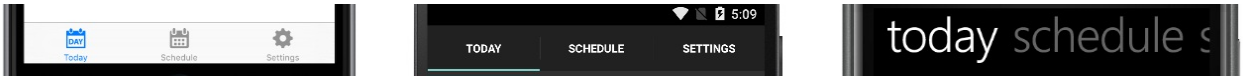
Xamarin.Forms TabbedPage 包含一系列选项卡和较大的详细信息区域，与每个选项卡加载到详细信息区域的内容。本文演示如何使用您不要将 TabbedPage 页面的集合中导航。

概述

下面的屏幕截图演示 `TabbedPage` 每个平台上：



下面的屏幕截图重点介绍每个平台上的选项卡格式：



布局 `TabbedPage` ，其选项卡，并依赖于该平台：

- 在 iOS 上，选项卡的列表显示在屏幕的底部，在详细信息区域上方。每个选项卡还包含图标图像应是 30 倍的正常解析透明度具有 30 PNG、高分辨率的 60 x 60 和 90 x 90 适用于 iPhone 6 Plus 解决方法。如果有五个选项卡 *详细*选项卡将出现，这可用于访问其他选项卡。有关加载 Xamarin.Forms 应用程序中的映像的详细信息，请参阅处理图像。有关图标要求的详细信息，请参阅创建选项卡式应用程序。

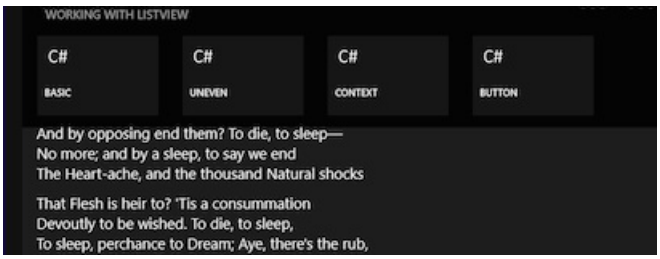
  > **NOTE**
  > 请注意， `TabbedRenderer` for iOS 的可重写 `GetIcon` 方法，可以用来从指定的数据源加载选项卡图标。此重写就可以使用 SVG 图像以图标形式在 `TabbedPage` 。此外，可以提供选定和未选定版本的图标。

- 在 Android 上，在屏幕顶部显示默认情况下的选项卡的列表和详细信息区域低于。但是，选项卡列表可以移动到具有平台特定屏幕的底部。有关详细信息，请参阅设置 TabbedPage 工具栏位置和颜色。

  > **NOTE**
  > 请注意，当在 Android 上使用 AppCompat，每个选项卡还将显示一个图标。此外， `TabbedPageRenderer` for Android AppCompat 的可重写 `SetTabIcon` 方法，可以用来加载的自定义选项卡图标 `Drawable` 。此重写就可以使用 SVG 图像以图标形式在 `TabbedPage` 。

- 在 Windows 平板电脑外形规格，选项卡始终不可见，并且用户需要对向轻扫取（或右键单击，如果它们的附

加是鼠标）若要查看中的选项卡 `TabbedPage`（如下所示）。



创建您不要将 TabbedPage

两种方法可用于创建 `TabbedPage`：

- 填充 `TabbedPage` 的子集合 `Page` 对象，如集合 `ContentPage` 实例。
- 将分配收藏 `ItemsSource` 属性和分配 `DataTemplate` 到 `ItemTemplate` 属性返回的页集合中的对象。

这两种方法 `TabbedPage` 在用户选择每个选项卡将显示每一页。

> **NOTE**
> 我们建议 `TabbedPage` 应填充了 `NavigationPage` 并 `ContentPage` 仅限实例。这将有助于确保在所有平台上一致的用户体验。

填充您不要将 TabbedPage 页面集合

以下 XAML 代码示例所示 `TabbedPage` 构造通过填充了一系列子 `Page` 对象：

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
            xmlns:local="clr-namespace:TabbedPageWithNavigationPage;assembly=TabbedPageWithNavigationPage"
            x:Class="TabbedPageWithNavigationPage.MainPage">
    <local:TodayPage />
    <NavigationPage Title="Schedule" Icon="schedule.png">
        <x:Arguments>
            <local:SchedulePage />
        </x:Arguments>
    </NavigationPage>
</TabbedPage>
```

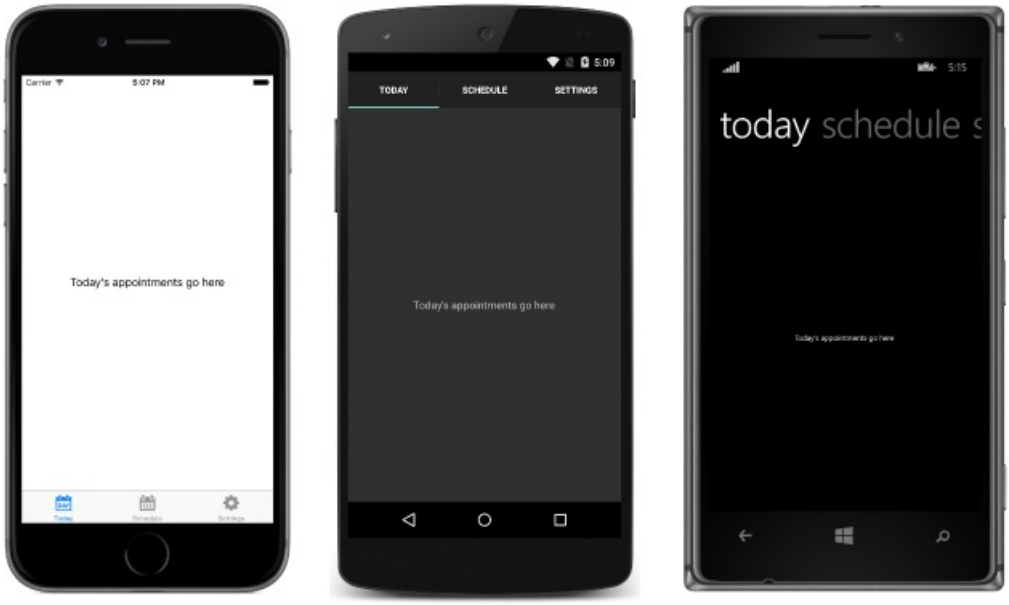
下面的代码示例显示等效 `TabbedPage` C# 创建的：

```
public class MainPageCS : TabbedPage
{
  public MainPageCS ()
  {
    var navigationPage = new NavigationPage (new SchedulePageCS ());
    navigationPage.Icon = "schedule.png";
    navigationPage.Title = "Schedule";

    Children.Add (new TodayPageCS ());
    Children.Add (navigationPage);
  }
}
```

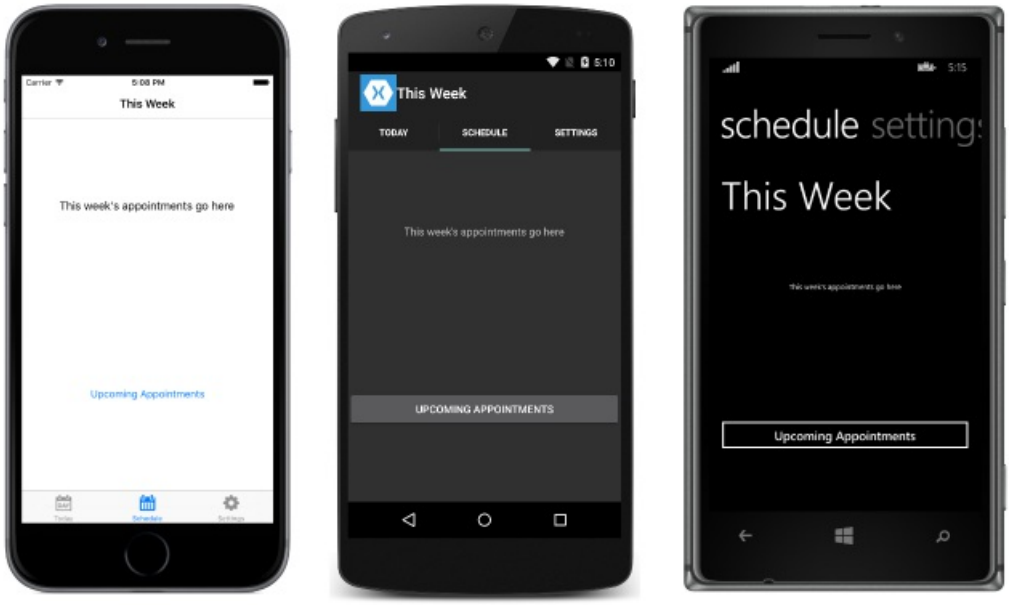
`TabbedPage` 具有两个子填充 `Page` 对象。第一个子级是 `ContentPage` 实例和第二个选项卡是 `NavigationPage` 包含 `ContentPage` 实例。

> **NOTE**
>
> `TabbedPage` 不支持 UI 虚拟化。因此，性能可能会影响如果 `TabbedPage` 包含太多的子元素。

下面的屏幕截图演示 `TodayPage` `ContentPage` 实例，而会显示在 今天选项卡：



选择 *计划*选项卡显示 `SchedulePage` `ContentPage` 实例，包装在 `NavigationPage` 实例，并在所示以下屏幕截图：



有关布局的信息 `NavigationPage` ，请参阅执行导航。

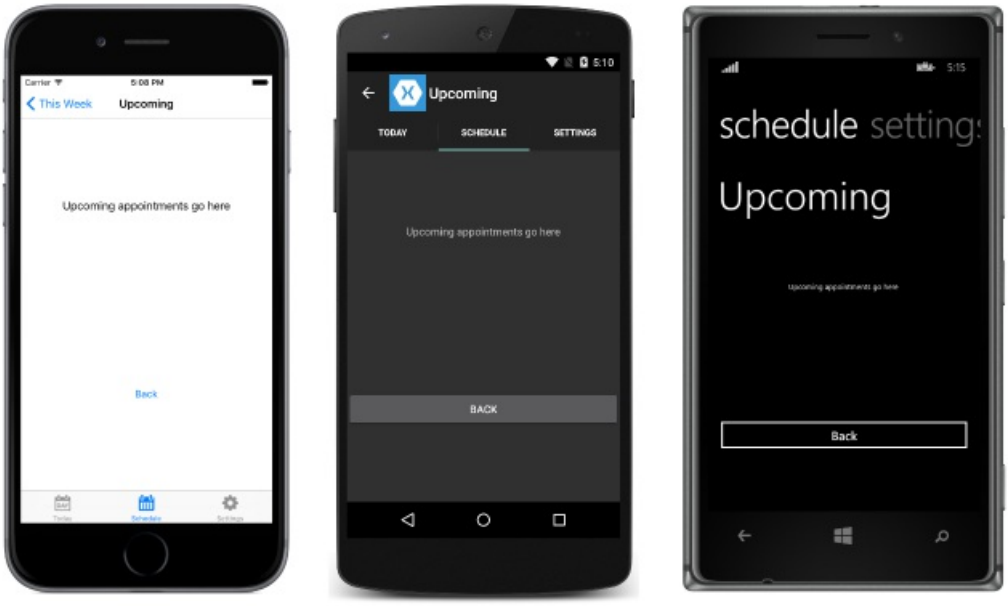
> **NOTE**
>
> 而是可以接受放置 `NavigationPage` 到 `TabbedPage` ，不建议将 `TabbedPage` 到 `NavigationPage` 。这是因为，在 iOS 上，`UITabBarController` 始终充当的包装器 `UINavigationController` 。有关详细信息，请参阅结合使用视图控制器接口iOS 开发人员库中。

选项卡内的导航

可以通过调用从第二个选项卡执行导航 `PushAsync` 方法 `Navigation` 属性的 `ContentPage` 实例，下面的代码示例中所示：

```
async void OnUpcomingAppointmentsButtonClicked (object sender, EventArgs e)
{
  await Navigation.PushAsync (new UpcomingAppointmentsPage ());
}
```

这会将 `UpcomingAppointmentsPage` 实例推送到导航堆栈中，在堆栈中，它成为活动页。以下屏幕截图所示：



有关执行导航使用详细信息 `NavigationPage` 类，请参阅分层导航。

填充您不要将 TabbedPage 使用模板

以下 XAML 代码示例所示 `TabbedPage` 构建的分配 `DataTemplate` 到 `ItemTemplate` 属性返回的页集合中的对象：

```
<TabbedPage xmlns="http://xamarin.com/schemas/2014/forms"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:local="clr-namespace:TabbedPageDemo;assembly=TabbedPageDemo"
            x:Class="TabbedPageDemo.TabbedPageDemoPage">
  <TabbedPage.Resources>
    <ResourceDictionary>
      <local:NonNullToBooleanConverter x:Key="booleanConverter" />
    </ResourceDictionary>
  </TabbedPage.Resources>
  <TabbedPage.ItemTemplate>
    <DataTemplate>
      <ContentPage Title="{Binding Name}" Icon="monkeyicon.png">
        <StackLayout Padding="5, 25">
          <Label Text="{Binding Name}" Font="Bold,Large" HorizontalOptions="Center" />
          <Image Source="{Binding PhotoUrl}" WidthRequest="200" HeightRequest="200" />
          <StackLayout Padding="50, 10">
            <StackLayout Orientation="Horizontal">
              <Label Text="Family:" HorizontalOptions="FillAndExpand" />
              <Label Text="{Binding Family}" Font="Bold,Medium" />
            </StackLayout>
            ...
          </StackLayout>
        </StackLayout>
      </ContentPage>
    </DataTemplate>
  </TabbedPage.ItemTemplate>
</TabbedPage>
```

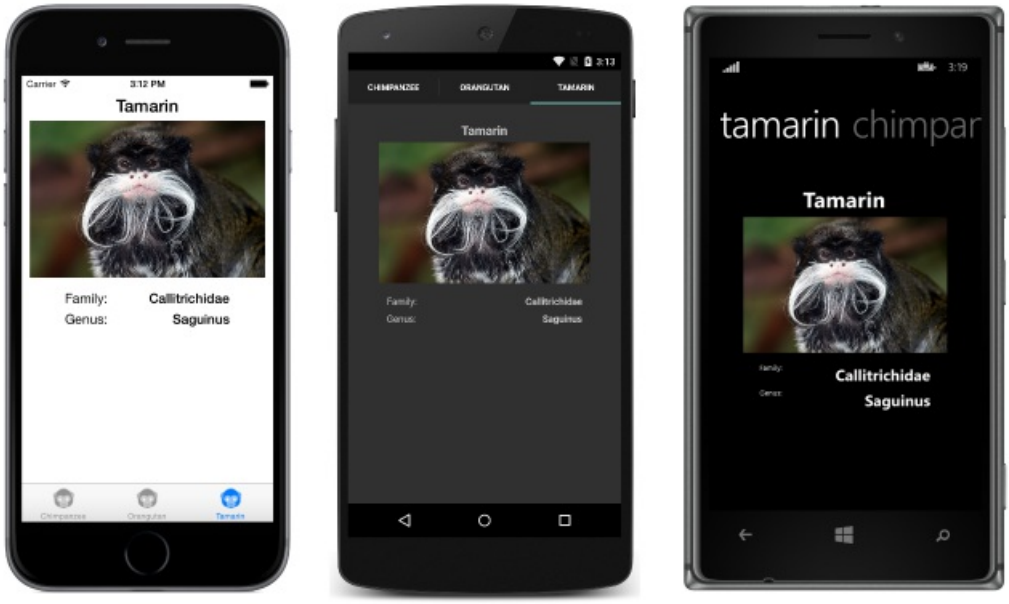
`TabbedPage` 通过设置使用数据填充 `ItemsSource` 代码隐藏文件的构造函数中的属性：

```
public TabbedPageDemoPage ()
{
  ...
  ItemsSource = MonkeyDataModel.All;
}
```

下面的代码示例显示等效 `TabbedPage` C# 创建的：

```
public class TabbedPageDemoPageCS : TabbedPage
{
  public TabbedPageDemoPageCS ()
  {
    var booleanConverter = new NonNullToBooleanConverter ();

    ItemTemplate = new DataTemplate (() => {
      var nameLabel = new Label {
        FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label)),
        FontAttributes = FontAttributes.Bold,
        HorizontalOptions = LayoutOptions.Center
      };
      nameLabel.SetBinding (Label.TextProperty, "Name");

      var image = new Image { WidthRequest = 200, HeightRequest = 200 };
      image.SetBinding (Image.SourceProperty, "PhotoUrl");

      var familyLabel = new Label {
        FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
        FontAttributes = FontAttributes.Bold
      };
      familyLabel.SetBinding (Label.TextProperty, "Family");
      ...

      var contentPage = new ContentPage {
        Icon = "monkeyicon.png",
        Content = new StackLayout {
          Padding = new Thickness (5, 25),
          Children = {
            nameLabel,
            image,
            new StackLayout {
              Padding = new Thickness (50, 10),
              Children = {
                new StackLayout {
                  Orientation = StackOrientation.Horizontal,
                  Children = {
                    new Label { Text = "Family:", HorizontalOptions = LayoutOptions.FillAndExpand },
                    familyLabel
                  }
                },
                ...
              }
            }
          }
        }
      };
      contentPage.SetBinding (TitleProperty, "Name");
      return contentPage;
    });
    ItemsSource = MonkeyDataModel.All;
  }
}
```

每个选项卡显示 `ContentPage` 使用一系列 `StackLayout` 并 `Label` 实例显示为选项卡的数据。以下屏幕截图显示的内容*Tamarin*选项卡：



然后选择另一个选项卡将显示该选项卡的内容。

> **NOTE**
>
> `TabbedPage` 不支持 UI 虚拟化。因此，性能可能会影响如果 `TabbedPage` 包含太多的子元素。

有关详细信息 `TabbedPage`，请参阅第 25 章的 Charles Petzold 的 Xamarin.Forms 书籍。

总结

本文演示了如何使用您不要将 TabbedPage 页面的集合中导航。Xamarin.Forms `TabbedPage` 包含具有内容载入的详细信息区域的每个选项卡的选项卡和较大的详细信息区域的列表。

相关链接

- 页类型
- TabbedPageWithNavigationPage（示例）
- TabbedPage（示例）
- TabbedPage

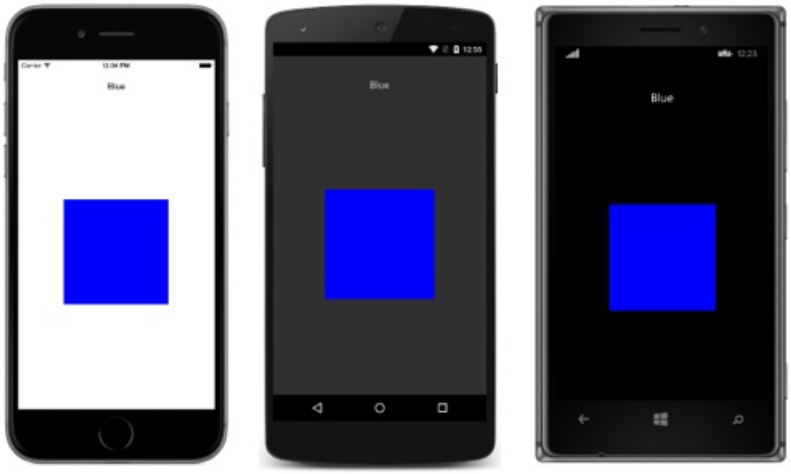
Xamarin.Forms 轮播页面

2018/10/26 • • Edit Online

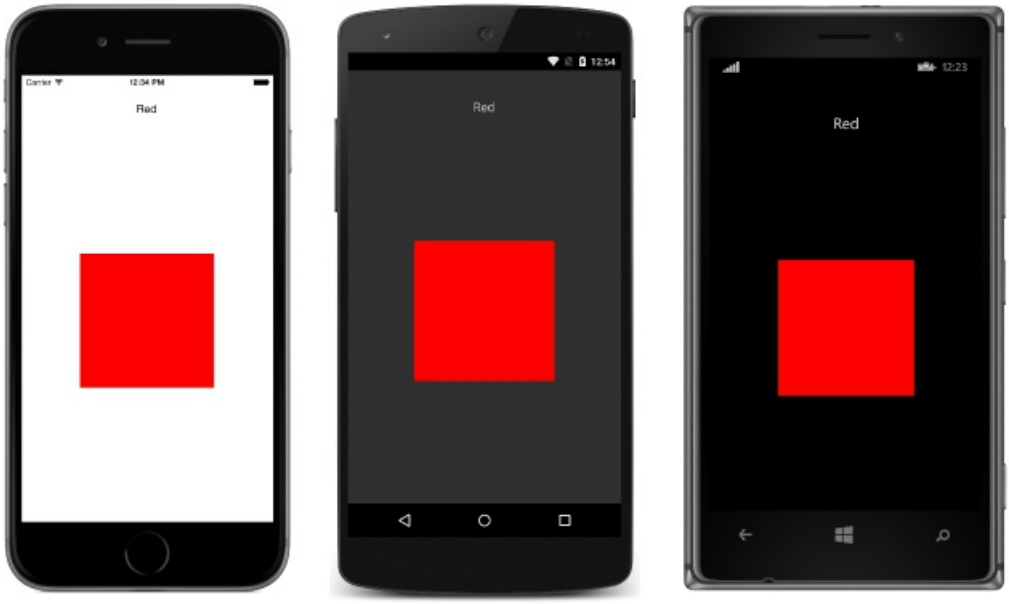
Xamarin.Forms CarouselPage 是内容的一个页面，用户可以向从左到右轻扫来导航页面，例如库。本文演示如何使用 CarouselPage 导航页的集合。

概述

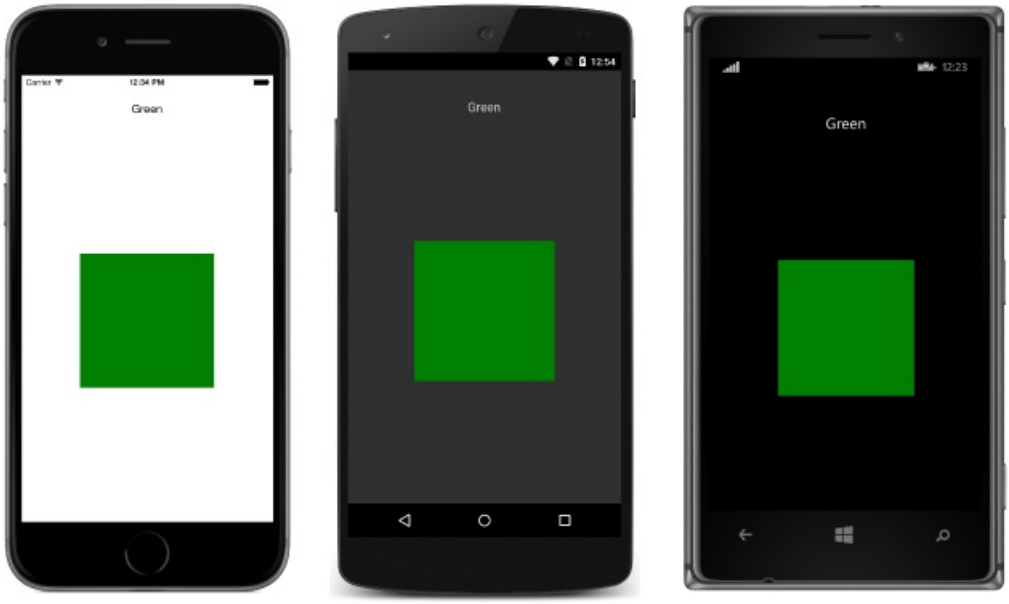
下面的屏幕截图演示 `CarouselPage` 每个平台上：



布局 `CarouselPage` 是在每个平台上完全相同。导航页，可以通过轻扫右到左遍历该集合，向前导航和轻扫从左到右向后导航整个集合。以下屏幕截图显示在首页 `CarouselPage` 实例：



轻扫从右到左移到第二个页面，如以下屏幕截图中所示：



再次从右到左轻扫会移到第三个页上，而从左到右轻扫返回到以前的页面。

创建 CarouselPage

两种方法可用于创建 `CarouselPage`：

- 填充 `CarouselPage` 了一系列子 `ContentPage` 实例。
- 将分配收藏 `ItemsSource` 属性和分配 `DataTemplate` 到 `ItemTemplate` 属性以返回 `ContentPage` 集合中对象的实例。

这两种方法，`CarouselPage` 将然后显示每个页面又，移动到下一步页要显示的重击交互。

> **NOTE**
> 一个 `CarouselPage` 仅可以使用填充 `ContentPage` 实例，或 `ContentPage` 派生类。

填充页集合与 CarouselPage

以下 XAML 代码示例所示 `CarouselPage`，它显示三个 `ContentPage` 实例：

```
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="CarouselPageNavigation.MainPage">
    <ContentPage>
        <ContentPage.Padding>
          <OnPlatform x:TypeArguments="Thickness">
              <On Platform="iOS, Android" Value="0,40,0,0" />
          </OnPlatform>
        </ContentPage.Padding>
        <StackLayout>
            <Label Text="Red" FontSize="Medium" HorizontalOptions="Center" />
            <BoxView Color="Red" WidthRequest="200" HeightRequest="200" HorizontalOptions="Center"
VerticalOptions="CenterAndExpand" />
        </StackLayout>
    </ContentPage>
    <ContentPage>
        ...
    </ContentPage>
    <ContentPage>
        ...
    </ContentPage>
</CarouselPage>
```

下面的代码示例显示了中的等效 UI C#:

```
public class MainPageCS : CarouselPage
{
    public MainPageCS ()
    {
        Thickness padding;
        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
            case Device.Android:
                padding = new Thickness(0, 40, 0, 0);
                break;
            default:
                padding = new Thickness();
                break;
        }

        var redContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                Children = {
                    new Label {
                        Text = "Red",
                        FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                        HorizontalOptions = LayoutOptions.Center
                    },
                    new BoxView {
                        Color = Color.Red,
                        WidthRequest = 200,
                        HeightRequest = 200,
                        HorizontalOptions = LayoutOptions.Center,
                        VerticalOptions = LayoutOptions.CenterAndExpand
                    }
                }
            }
        };
        var greenContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                ...
            }
        };
        var blueContentPage = new ContentPage {
            Padding = padding,
            Content = new StackLayout {
                ...
            }
        };

        Children.Add (redContentPage);
        Children.Add (greenContentPage);
        Children.Add (blueContentPage);
    }
}
```

每个 `ContentPage` 只是显示 `Label` 特定颜色和一个 `BoxView` 该颜色。

> **NOTE**
>
> `CarouselPage` 不支持 UI 虚拟化。因此，性能可能会影响如果 `CarouselPage` 包含太多的子元素。

如果 `CarouselPage` 嵌入到 `Detail` 页 `MasterDetailPage`，则 `MasterDetailPage.IsGestureEnabled` 属性应设置为 `false` 以防止手势之间的冲突 `CarouselPage` 和 `MasterDetailPage`。

有关详细信息 `CarouselPage`，请参阅第 25 章的 Charles Petzold 的 Xamarin.Forms 书籍。

填充 CarouselPage 使用模板

以下 XAML 代码示例所示 `CarouselPage` 构建的分配 `DataTemplate` 到 `ItemTemplate` 属性返回的页集合中的对象：

```
<CarouselPage xmlns="http://xamarin.com/schemas/2014/forms"
              xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
              x:Class="CarouselPageNavigation.MainPage">
    <CarouselPage.ItemTemplate>
        <DataTemplate>
            <ContentPage>
                <ContentPage.Padding>
                  <OnPlatform x:TypeArguments="Thickness">
                    <On Platform="iOS, Android" Value="0,40,0,0" />
                  </OnPlatform>
                </ContentPage.Padding>
                <StackLayout>
                    <Label Text="{Binding Name}" FontSize="Medium" HorizontalOptions="Center" />
                    <BoxView Color="{Binding Color}" WidthRequest="200" HeightRequest="200"
HorizontalOptions="Center" VerticalOptions="CenterAndExpand" />
                </StackLayout>
            </ContentPage>
        </DataTemplate>
    </CarouselPage.ItemTemplate>
</CarouselPage>
```

`CarouselPage` 通过设置使用数据填充 `ItemsSource` 代码隐藏文件的构造函数中的属性：

```
public MainPage ()
{
    ...
    ItemsSource = ColorsDataModel.All;
}
```

下面的代码示例显示等效 `CarouselPage` 中创建C#：

```
public class MainPageCS : CarouselPage
{
    public MainPageCS ()
    {
        Thickness padding;
        switch (Device.RuntimePlatform)
        {
            case Device.iOS:
            case Device.Android:
                padding = new Thickness(0, 40, 0, 0);
                break;
            default:
                padding = new Thickness();
                break;
        }

        ItemTemplate = new DataTemplate (() => {
            var nameLabel = new Label {
                FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
                HorizontalOptions = LayoutOptions.Center
            };
            nameLabel.SetBinding (Label.TextProperty, "Name");

            var colorBoxView = new BoxView {
                WidthRequest = 200,
                HeightRequest = 200,
                HorizontalOptions = LayoutOptions.Center,
                VerticalOptions = LayoutOptions.CenterAndExpand
            };
            colorBoxView.SetBinding (BoxView.ColorProperty, "Color");

            return new ContentPage {
                Padding = padding,
                Content = new StackLayout {
                    Children = {
                        nameLabel,
                        colorBoxView
                    }
                }
            };
        });

        ItemsSource = ColorsDataModel.All;
    }
}
```

每个 `ContentPage` 只是显示 `Label` 特定颜色和一个 `BoxView` 该颜色。

> **NOTE**
>
> `CarouselPage` 不支持 UI 虚拟化。因此，性能可能会影响如果 `CarouselPage` 包含太多的子元素。

如果 `CarouselPage` 嵌入到 `Detail` 页 `MasterDetailPage`，则 `MasterDetailPage.IsGestureEnabled` 属性应设置为 `false` 以防止手势之间的冲突 `CarouselPage` 和 `MasterDetailPage`。

有关详细信息 `CarouselPage`，请参阅第 25 章的 Charles Petzold 的 Xamarin.Forms 书籍。

总结

本文演示了如何使用 `CarouselPage` 来导航页面的集合。`CarouselPage` 是一个页面，用户可以向从左到右轻扫来导航页面的内容，类似于库。

相关链接

- 页类型
- CarouselPage（示例）
- CarouselPageTemplate（示例）
- CarouselPage

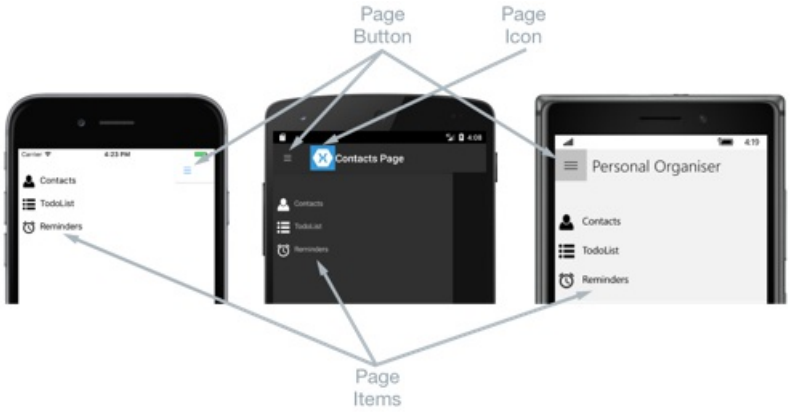
Xamarin.Forms 母版-详细信息页

2018/11/1 • • Edit Online

Xamarin.Forms MasterDetailPage 是信息的管理两个相关的页面 – 主页面，其中的项，并显示主页面的项目的详细信息的详细信息页的页。本文介绍如何使用 MasterDetailPage 和其信息的页面之间导航。

概述

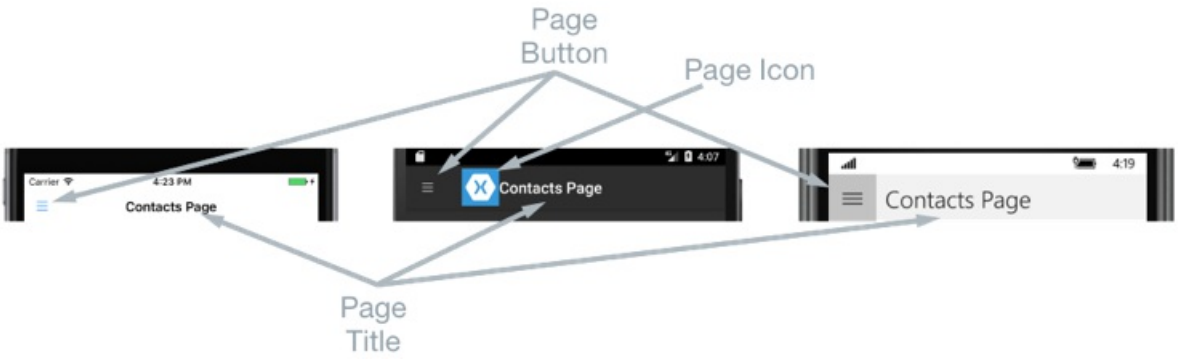
主页面通常会显示一系列项，如以下屏幕截图中所示：



在每个平台上完全相同的项列表的位置，选择某一项将导航到相应的详细信息页。此外，主页还提供包含一个按钮，可用于导航到活动详细信息页的导航条：

- 在 iOS 上，导航栏位于页面顶部，并有一个按钮，导航到详细信息页。此外，还可以母版页向左轻扫到导航活动详细信息页。
- 在 Android 上，导航栏位于页面顶部，并将标题、图标和导航的按钮显示的详细信息页。在中定义的图标 `[Activity]` 修饰的属性 `MainActivity` Android 的特定于平台的项目中的类。此外，活动详细信息页可以导航到母版页向左轻扫，点击屏幕上，最右侧的详细信息页和通过点击*回*屏幕底部的按钮。
- 在通用 Windows 平台 (UWP)，导航栏位于页面顶部，并有一个按钮，导航到详细信息页。

页上，在主机上选择对应于的项详细信息页上显示数据并在下面的屏幕截图中显示的详细信息页的主要组件：



详细信息页包含导航栏中，其内容是依赖于平台的：

- 在 iOS 上，导航栏位于页面顶部显示标题，并有一个按钮，返回到主页面，前提是详细信息页实例包装在 `NavigationPage` 实例。此外，还可以轻扫到右侧的详细信息页向返回主页面。
- 在 Android 上，导航栏位于页面顶部，并显示一个标题、图标和一个按钮，返回到主页面。在中定义的图标 `[Activity]` 修饰的属性 `MainActivity` Android 的特定于平台的项目中的类。
- 在 UWP 中，导航栏位于页面顶部显示标题，并有一个按钮，返回到主页面。

导航行为

母版和详细信息页之间导航体验的行为是依赖于平台：

- 在 iOS 上，详细信息页*幻灯片左*侧，和的详细信息的左侧的部分作为母版页幻灯片右侧页仍是可见的。
- 在 Android 上，详细信息和主页面都*叠加*相互。
- UWP 上的详细信息和主页面*交换*。

只不过 iOS 和 Android 上的母版页类似的宽度都作为在纵向模式下，母版页使更多详细信息页将显示，则将在横向模式中观察到类似的行为。

有关如何控制导航行为的信息，请参阅控制的详细信息页显示行为。

创建 MasterDetailPage

一个 `MasterDetailPage` 包含 `Master` 并 `Detail` 存在以下两种类型的属性 `Page`，用于获取和分别设置母版和详细信息页。

> **IMPORTANT**
>
> 一个 `MasterDetailPage` 旨在成为根页，并使用它，因为子页面中其他页类型可能导致意外和不一致行为。此外，我们建议的母版页 `MasterDetailPage` 始终应 `ContentPage` 实例和的详细信息页应只填充了 `TabbedPage`， `NavigationPage`，和 `ContentPage` 实例。这将有助于确保在所有平台上一致的用户体验。

以下 XAML 代码示例所示 `MasterDetailPage`，用于设置 `Master` 并 `Detail` 属性：

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  xmlns:local="clr-namespace:MasterDetailPageNavigation;assembly=MasterDetailPageNavigation"
                  x:Class="MasterDetailPageNavigation.MainPage">
    <MasterDetailPage.Master>
        <local:MasterPage x:Name="masterPage" />
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <NavigationPage>
            <x:Arguments>
                <local:ContactsPage />
            </x:Arguments>
        </NavigationPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

下面的代码示例显示等效 `MasterDetailPage` 中创建C#:

```
public class MainPageCS : MasterDetailPage
{
    MasterPageCS masterPage;

    public MainPageCS ()
    {
        masterPage = new MasterPageCS ();
        Master = masterPage;
        Detail = new NavigationPage (new ContactsPageCS ());
        ...
    }
    ...
}
```

`MasterDetailPage.Master` 属性设置为 `ContentPage` 实例。 `MasterDetailPage.Detail` 属性设置为 `NavigationPage` 包

含 `ContentPage` 实例。

创建主页面

以下 XAML 代码示例演示的声明 `MasterPage` 对象，通过引用 `MasterDetailPage.Master` 属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="using:MasterDetailPageNavigation"
             x:Class="MasterDetailPageNavigation.MasterPage"
             Padding="0,40,0,0"
             Icon="hamburger.png"
             Title="Personal Organiser">
    <StackLayout>
        <ListView x:Name="listView" x:FieldModifier="public">
           <ListView.ItemsSource>
                <x:Array Type="{x:Type local:MasterPageItem}">
                    <local:MasterPageItem Title="Contacts" IconSource="contacts.png" TargetType="{x:Type
local:ContactsPage}" />
                    <local:MasterPageItem Title="TodoList" IconSource="todo.png" TargetType="{x:Type
local:TodoListPage}" />
                    <local:MasterPageItem Title="Reminders" IconSource="reminders.png" TargetType="{x:Type
local:ReminderPage}" />
                </x:Array>
            </ListView.ItemsSource>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <Grid Padding="5,10">
                            <Grid.ColumnDefinitions>
                                <ColumnDefinition Width="30"/>
                                <ColumnDefinition Width="*" />
                            </Grid.ColumnDefinitions>
                            <Image Source="{Binding IconSource}" />
                            <Label Grid.Column="1" Text="{Binding Title}" />
                        </Grid>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

页面组成 `ListView` 使用 XAML 中的数据在通过设置填充其 `ItemsSource` 属性设置为一个数组 `MasterPageItem` 实例。每个 `MasterPageItem` 定义 `Title`， `IconSource`，和 `TargetType` 属性。

一个 `DataTemplate` 分配给 `ListView.ItemTemplate` 属性，以显示每个 `MasterPageItem`。`DataTemplate` 包含 `ViewCell` 组成 `Image` 并 `Label`。`Image` 显示 `IconSource` 属性值，并且 `Label` 显示 `Title` 属性值，每个 `MasterPageItem`。

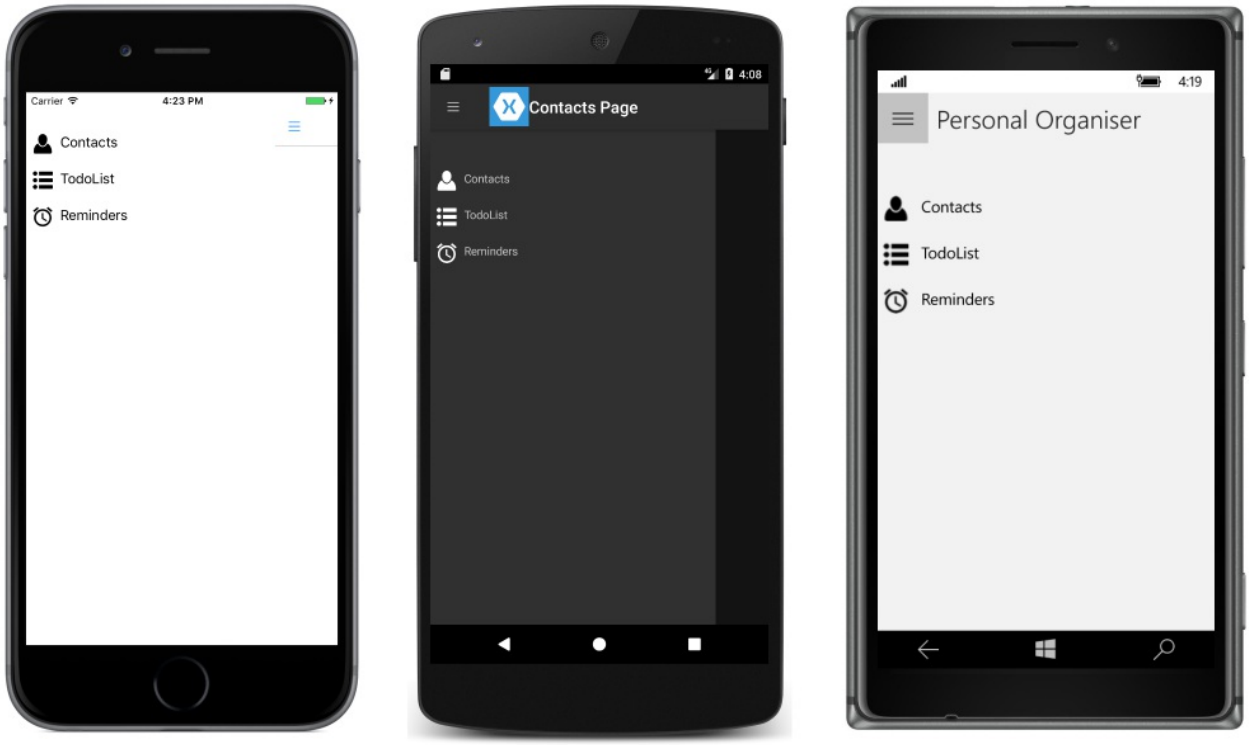
页都有其 `Title` 并 `Icon` 属性集。假设的详细信息页具有标题栏，在详细信息页上，将显示图标。这必须在 iOS 上启用，通过包装中的详细信息页实例 `NavigationPage` 实例。

> **NOTE**
>
> `MasterDetailPage.Master` 页面必须有其 `Title` 设置属性，或将出现异常。

下面的代码示例显示了在 C# 中创建的等效页：

```
public class MasterPageCS : ContentPage
{
  public ListView ListView { get { return listView; } }

  ListView listView;

  public MasterPageCS ()
  {
    var masterPageItems = new List<MasterPageItem> ();
    masterPageItems.Add (new MasterPageItem {
      Title = "Contacts",
      IconSource = "contacts.png",
      TargetType = typeof(ContactsPageCS)
    });
    masterPageItems.Add (new MasterPageItem {
      Title = "TodoList",
      IconSource = "todo.png",
      TargetType = typeof(TodoListPageCS)
    });
    masterPageItems.Add (new MasterPageItem {
      Title = "Reminders",
      IconSource = "reminders.png",
      TargetType = typeof(ReminderPageCS)
    });

    listView = new ListView {
      ItemsSource = masterPageItems,
      ItemTemplate = new DataTemplate (() => {
        var grid = new Grid { Padding = new Thickness(5, 10) };
        grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(30) });
        grid.ColumnDefinitions.Add(new ColumnDefinition { Width = GridLength.Star });

        var image = new Image();
        image.SetBinding(Image.SourceProperty, "IconSource");
        var label = new Label { VerticalOptions = LayoutOptions.FillAndExpand };
        label.SetBinding(Label.TextProperty, "Title");

        grid.Children.Add(image);
        grid.Children.Add(label, 1, 0);

        return new ViewCell { View = grid };
      }),
      SeparatorVisibility = SeparatorVisibility.None
    };

    Icon = "hamburger.png";
    Title = "Personal Organiser";
    Content = new StackLayout
    {
      Children = { listView }
    };
  }
}
```

以下屏幕截图显示在母版页上每个平台上:



创建并显示详细信息页

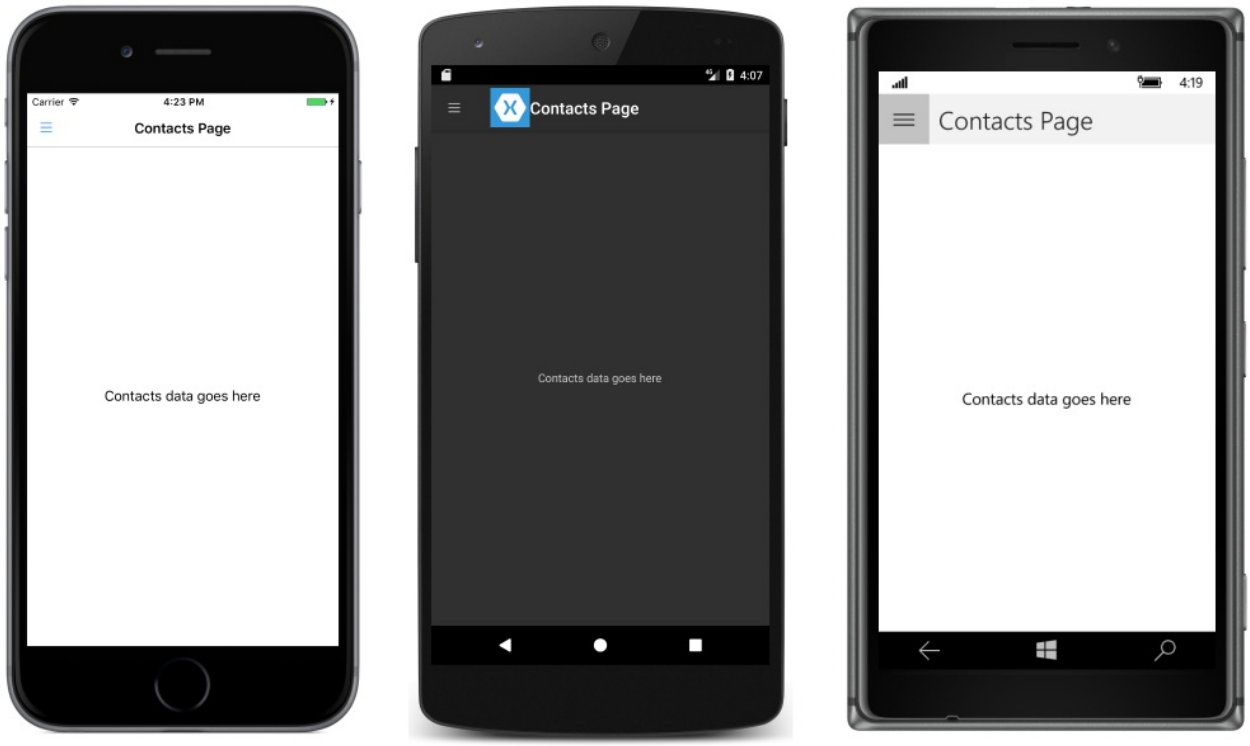
`MasterPage` 实例包含 `ListView` 公开的属性及其 `ListView` 实例，以便 `MainPage` `MasterDetailPage` 实例可以注册事件处理程序来处理 `ItemSelected` 事件。这使得 `MainPage` 实例设置 `Detail` 属性设置为表示所选的页面 `ListView` 项。下面的代码示例显示了事件处理程序：

```
public partial class MainPage : MasterDetailPage
{
    public MainPage ()
    {
        ...
        masterPage.listView.ItemSelected += OnItemSelected;
    }

    void OnItemSelected (object sender, SelectedItemChangedEventArgs e)
    {
        var item = e.SelectedItem as MasterPageItem;
        if (item != null) {
            Detail = new NavigationPage ((Page)Activator.CreateInstance (item.TargetType));
            masterPage.listView.SelectedItem = null;
            IsPresented = false;
        }
    }
}
```

`OnItemSelected` 方法将执行以下操作：

- 它检索 `SelectedItem` 从 `ListView` 实例，并提供它不是 `null`，设置为中存储的页类型的新实例的详细信息页 `TargetType` 属性的 `MasterPageItem`。页类型包装在 `NavigationPage` 实例，确保通过引用的图标 `Icon` 属性 `MasterPage` 在 iOS 中的详细信息页上显示。
- 中的选定的项 `ListView` 设置为 `null` 以确保没有任何 `ListView` 选择在项目中的下一次 `MasterPage` 显示。
- 通过设置向用户显示的详细信息页 `MasterDetailPage.IsPresented` 属性设置为 `false`。此属性控制是否显示 master 或详细信息页。它应设置为 `true` 以显示主页上，并对其 `false` 以显示详细信息页。

下面的屏幕截图演示 `ContactPage` 详细信息页上之后它已被选中在母版页上，所示：



控制的详细信息页显示行为

如何 `MasterDetailPage` 管理的母版和详细信息页上应用程序还是运行在手机或平板电脑，设备的方向和的值取决于 `MasterBehavior` 属性。此属性确定的详细信息页的显示方式。可能值有：

- **默认**– 使用平台默认显示的页。
- **弹出框**– 的详细信息页盖住了，或部分覆盖的母版页。
- **拆分**– 主页面显示在左侧和右侧是详细信息页。
- **SplitOnLandscape** – 时在设备处于横向方向使用拆分屏幕。
- **SplitOnPortrait** – 当设备处于纵向方向时使用拆分屏幕。

下面的 XAML 代码示例演示如何设置 `MasterBehavior` 上的属性 `MasterDetailPage`：

```
<?xml version="1.0" encoding="UTF-8"?>
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
                  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                  x:Class="MasterDetailPageNavigation.MainPage"
                  MasterBehavior="Popover">
  ...
</MasterDetailPage>
```

下面的代码示例显示等效 `MasterDetailPage` 中创建C#:

```
public class MainPageCS : MasterDetailPage
{
    MasterPageCS masterPage;

    public MainPageCS ()
    {
        MasterBehavior = MasterBehavior.Popover;
        ...
    }
}
```

但是，值 `MasterBehavior` 属性只影响在平板电脑或桌面上运行的应用程序。始终在电话上运行的应用程序具有*弹出框*行为。

总结

本文演示了如何使用 `MasterDetailPage` 和其信息的页面之间导航。Xamarin.Forms `MasterDetailPage` 是可管理两个页面的相关信息 – 主页面，其中的项，并显示主页面的项目的详细信息的详细信息页面的页面。

相关链接

- 页类型
- MasterDetailPage（示例）
- MasterDetailPage

Xamarin.Forms 模式页

2018/7/13 • • Edit Online

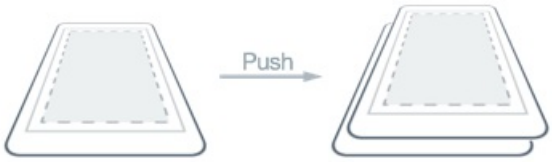
Xamarin.forms 支持模式页面。模式页面鼓励用户完成独立任务，在完成或取消该任务之前，不允许导航离开该任务。本文演示如何导航到模式页面。

本文讨论以下主题：

- 执行导航– 将页面推送到模式堆栈，模式堆栈中弹出页面，禁用后退按钮，并对页面过渡效果进行动画处理。
- 将数据传递时导航– 将数据页的构造函数，通过和传递 `BindingContext` 。

概述

模式页面可以是任一页Xamarin.Forms 支持的类型。若要显示模式页面应用程序会将其推送到了模式堆栈，其中它将成为活动页，如以下关系图中所示：



若要返回到以前的页面应用程序会弹出当前页从模式堆栈，而最顶层的页成为活动页，如以下关系图中所示：

执行导航

可以由任何 `Page` 派生类型上的 `Navigation` 属性公开模式导航方法。这些方法提供的功能推送模式页面到模式堆栈上，并弹出模式页面从模式堆栈。

`Navigation` 属性也公开 `ModalStack` 属性可以从中获取模式堆栈中的模式页面。但是，在模式导航中没有执行模式堆栈操作或弹出到根页的概念。这是因为基础平台普遍都不支持这些操作。

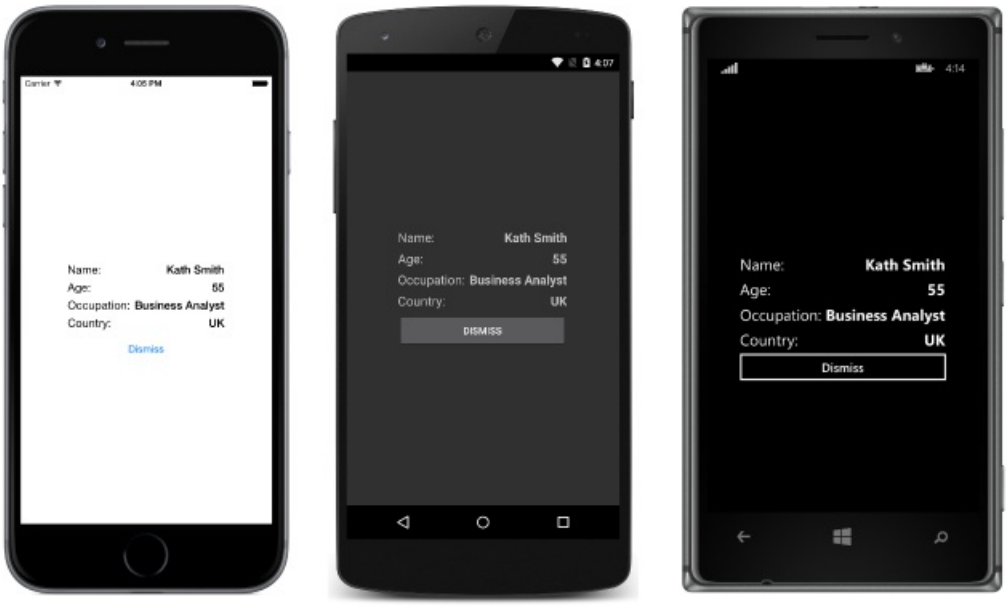
> **NOTE**
> 执行模式页面导航无需具有 `NavigationPage` 实例。

将页面推送到模式堆栈

若要导航到 `ModalPage` 需要调用 `PushModalAsync` 方法 `Navigation` 属性的当前页上，如下面的代码示例所示：

```
async void OnItemSelected (object sender, SelectedItemChangedEventArgs e)
{
  if (listView.SelectedItem != null) {
    var detailPage = new DetailPage ();
    ...
    await Navigation.PushModalAsync (detailPage);
  }
}
```

这将导致 `ModalPage` 实例可以被推送到了模式堆栈，其中它成为活动页，提供已中选择一项 `ListView` 上 `MainPage` 实例。 `ModalPage` 实例下面的屏幕截图中所示：



当 `PushModalAsync` 调用时，会发生以下事件：

- 页调用 `PushModalAsync` 具有其 `OnDisappearing` 重写调用，前提是基础平台不是 Android。
- 要导航到页都有其 `OnAppearing` 重写调用。
- `PushAsync` 任务完成。

但是，这些事件发生的确切顺序是依赖于平台。有关详细信息，请参阅第 24 章的 Charles Petzold 的 Xamarin.Forms 书籍。

> **NOTE**
> 调用 `OnDisappearing` 并 `OnAppearing` 重写不能被视为有保证的页面导航的迹象。例如，在 iOS 上，`OnDisappearing` 在应用程序终止时重写在活动页面上调用。

模式堆栈中弹出页面

活动页面可以从堆栈中弹出模式通过按*回*按钮在设备上，而不考虑这是否在设备上的物理按钮或屏幕按钮。

若要以编程方式返回原始页， `ModalPage` 实例必须调用 `PopModalAsync` 方法，如以下代码示例所示：

```
async void OnDismissButtonClicked (object sender, EventArgs args)
{
  await Navigation.PopModalAsync ();
}
```

这将导致 `ModalPage` 模式堆栈中，从最顶层的页成为活动页与要删除的实例。当 `PopModalAsync` 调用时，会发生以下事件：

- 页调用 `PopModalAsync` 具有其 `OnDisappearing` 重写调用。
- 返回到页都有其 `OnAppearing` 重写调用，前提是基础平台不是 Android。
- `PopModalAsync` 任务返回。

但是，这些事件发生的确切顺序是依赖于平台。有关详细信息，请参阅第 24 章的 Charles Petzold 的 Xamarin.Forms 书籍。

禁用后退按钮

在 Android 上，用户可以始终通过返回到前一页按标准*回*在设备上的按钮。如果模式页面需要用户离开页面之前完成独立的任务，必须禁用该应用程序*回*按钮。这可以通过重写 `Page.OnBackButtonPressed` 模式页面上的方法。有关详细信息请参阅第 24 章的 Charles Petzold 的 Xamarin.Forms 书籍。

对页面过渡效果进行动画处理

`Navigation` 属性的每一页还提供了重写的 push 和 pop 方法，包括 `boolean` 控制是否显示在导航窗格中，一个页面的动画，如下面的代码中所示的参数示例：

```
async void OnNextPageButtonClicked (object sender, EventArgs e)
{
  // Page appearance not animated
  await Navigation.PushModalAsync (new DetailPage (), false);
}

async void OnDismissButtonClicked (object sender, EventArgs args)
{
  // Page appearance not animated
  await Navigation.PopModalAsync (false);
}
```

设置 `boolean` 参数 `false` 禁用时将参数设置为的页面过渡动画 `true` 使页面过渡动画，前提是基础平台支持。但是，缺少此参数的 push 和 pop 方法默认情况下启用动画。

导航时传递数据

有时，所需的页导航期间将数据传递给另一页。实现此目的的两种方法是按页构造函数中，通过传递数据以及通过设置新页面 `BindingContext` 的数据。每个将现在讨论反过来。

通过 Page 构造函数传递数据

在导航过程将数据传递到另一个页面的最简单方法是通过页构造函数参数，下面的代码示例中所示：

```
public App ()
{
  MainPage = new MainPage (DateTime.Now.ToString ("u")));
}
```

此代码将创建 `MainPage` 实例，并传入当前日期和时间 ISO8601 格式中。

`MainPage` 实例收到的数据通过构造函数参数，如下面的代码示例中所示：

```
public MainPage (string date)
{
  InitializeComponent ();
  dateLabel.Text = date;
}
```

数据将通过设置显示在页面 `Label.Text` 属性。

BindingContext 间传递数据

用于导航期间将数据传递到另一页一种替代方法是通过设置新页面 `BindingContext` 到数据，如下面的代码示例中所示：

```
async void OnItemSelected (object sender, SelectedItemChangedEventArgs e)
{
  if (listView.SelectedItem != null) {
    var detailPage = new DetailPage ();
    detailPage.BindingContext = e.SelectedItem as Contact;
    listView.SelectedItem = null;
    await Navigation.PushModalAsync (detailPage);
  }
}
```

此代码将设置 `BindingContext` 的 `DetailPage` 实例向 `Contact` 实例，，然后转到 `DetailPage` 。

`DetailPage` 然后使用数据绑定来显示 `Contact` 实例数据，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ModalNavigation.DetailPage">
    <ContentPage.Padding>
      <OnPlatform x:TypeArguments="Thickness">
        <On Platform="iOS" Value="0,40,0,0" />
      </OnPlatform>
    </ContentPage.Padding>
    <ContentPage.Content>
        <StackLayout HorizontalOptions="Center" VerticalOptions="Center">
            <StackLayout Orientation="Horizontal">
                <Label Text="Name:" FontSize="Medium" HorizontalOptions="FillAndExpand" />
                <Label Text="{Binding Name}" FontSize="Medium" FontAttributes="Bold" />
            </StackLayout>
              ...
            <Button x:Name="dismissButton" Text="Dismiss" Clicked="OnDismissButtonClicked" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

下面的代码示例演示如何在 C# 中实现数据绑定：

```
public class DetailPageCS : ContentPage
{
  public DetailPageCS ()
  {
    var nameLabel = new Label {
      FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
      FontAttributes = FontAttributes.Bold
    };
    nameLabel.SetBinding (Label.TextProperty, "Name");
    ...
    var dismissButton = new Button { Text = "Dismiss" };
    dismissButton.Clicked += OnDismissButtonClicked;

    Thickness padding;
    switch (Device.RuntimePlatform)
    {
        case Device.iOS:
            padding = new Thickness(0, 40, 0, 0);
            break;
        default:
            padding = new Thickness();
            break;
    }

    Padding = padding;
    Content = new StackLayout {
      HorizontalOptions = LayoutOptions.Center,
      VerticalOptions = LayoutOptions.Center,
      Children = {
        new StackLayout {
          Orientation = StackOrientation.Horizontal,
          Children = {
            new Label{ Text = "Name:", FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
HorizontalOptions = LayoutOptions.FillAndExpand },
            nameLabel
          }
        },
        ...
        dismissButton
      }
    };
  }

  async void OnDismissButtonClicked (object sender, EventArgs args)
  {
    await Navigation.PopModalAsync ();
  }
}
```

数据将显示在页面的一系列 `Label` 控件。

若要深入了解数据绑定，请参阅数据绑定基本知识。

总结

本文演示了如何导航到模式页面。模式页面鼓励用户完成独立任务，在完成或取消该任务之前，不允许导航离开该任务。

相关链接

- 页面导航
- 模式（示例）
- PassingData（示例）

显示弹出窗口

2018/7/13 • • Edit Online

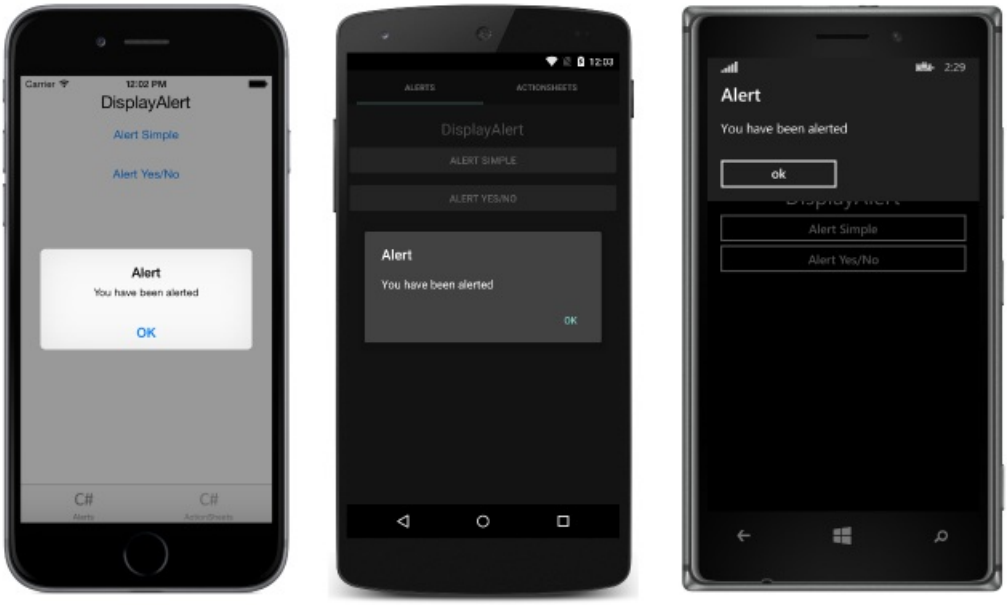
Xamarin.Forms 提供了两个弹出注册类似于用户界面元素 – 警报和操作工作表。本文演示如何使用警报和操作工作表 Api 要求简单的问题的用户和来指导用户完成任务。

显示警报或要求用户选择一个选项是一项常见 UI 任务。Xamarin.Forms 具有两种方法 `Page` 与一个弹出窗口，通过对用户进行交互的类：`DisplayAlert` 并 `DisplayActionSheet` 。它们将与每个平台上的相应本机控件呈现。

显示警报

所有 Xamarin.Forms 支持的平台都具有模式弹出框，提醒用户，或者提出这些简单的问题。若要显示这些警报在 Xamarin.Forms 中，使用 `DisplayAlert` 方法对任何 `Page` 。以下代码行向用户显示一个简单的消息：

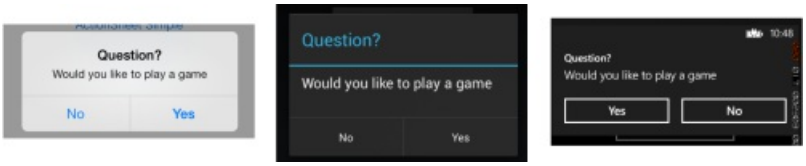
```
DisplayAlert ("Alert", "You have been alerted", "OK");
```



此示例不会从用户收集信息。警报以模式方式显示和关闭用户之后会继续与该应用程序进行交互。

`DisplayAlert` 方法还可用于捕获用户的响应，通过提供两个按钮，并返回 `boolean` 。若要获得警报的响应，提供这两个按钮的文本和 `await` 方法。在用户选择一个选项后，答案将返回到你的代码。请注意 `async` 和 `await` 下面的示例代码中的关键字：

```
async void OnAlertYesNoClicked (object sender, EventArgs e)
{
  var answer = await DisplayAlert ("Question?", "Would you like to play a game", "Yes", "No");
  Debug.WriteLine ("Answer: " + answer);
}
```

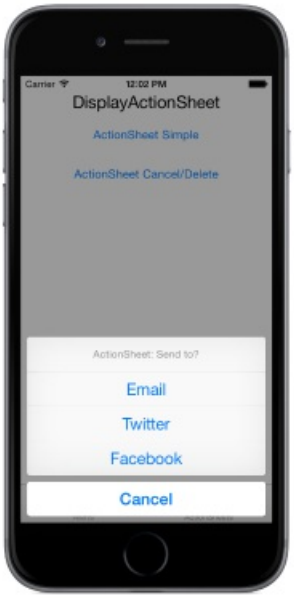


指导用户完成任务

UIActionSheet是 iOS 中的常见 UI 元素。Xamarin.Forms `DisplayActionSheet` 方法可用于在跨平台应用中，呈现在 Android 和 UWP 中的本机替代项包括此控件。

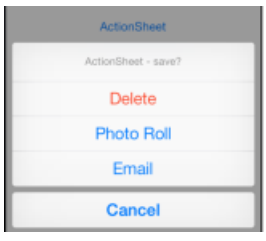
若要显示操作工作表，`await` `DisplayActionSheet` 任何 `Page` 、将消息传递和按钮标签设置为字符串。该方法返回的字符串为标签的用户单击的按钮。一个简单的示例如下所示：

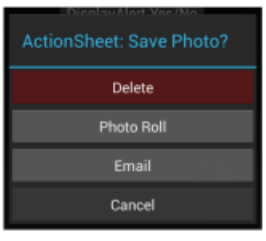
```
async void OnActionSheetSimpleClicked (object sender, EventArgs e)
{
  var action = await DisplayActionSheet ("ActionSheet: Send to?", "Cancel", null, "Email", "Twitter",
"Facebook");
  Debug.WriteLine ("Action: " + action);
}
```

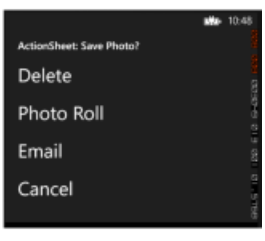


`destroy` 按钮呈现不同的方式，并且可以在 `null` 或指定为第三个字符串参数。下面的示例使用 `destroy` 按钮：

```
async void OnActionSheetCancelDeleteClicked (object sender, EventArgs e)
{
  var action = await DisplayActionSheet ("ActionSheet: SavePhoto?", "Cancel", "Delete", "Photo Roll",
"Email");
  Debug.WriteLine ("Action: " + action);
}
```







总结

本文演示了使用警报和操作工作表 Api 要求简单的问题的用户和来指导用户完成任务。Xamarin.Forms 有两种方法 `Page` 与一个弹出窗口，通过对用户进行交互的类：`DisplayAlert` 并 `DisplayActionSheet` ，并且它们同时使用每个平台上的相应本机控件呈现。

相关链接

- PopupsSample

Xamarin.Forms 模板

2018/7/13 • • Edit Online

控件模板

Xamarin.Forms 控件模板让你轻松设计或 re 主题可以在运行时的应用程序页。

数据模板

Xamarin.Forms 数据模板提供支持的控件上定义数据的表示形式的功能。

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms 库（示例）
- Xamarin.Forms 示例
- Xamarin.Forms API 文档

Xamarin.Forms 控件模板

2018/6/9 • • Edit Online

控件模板提供页的外观和其内容，支持很容易就能主题的页创建之间完全分离。

介绍

Xamarin.Forms 控件模板提供轻松主题和重新主题的功能在运行时的应用程序页。本文介绍控件模板。

创建 ControlTemplate

在应用程序级别或页级别，可以定义控件模板。本文演示如何创建和使用控件模板。

ControlTemplate 绑定

模板绑定都允许对数据的控件模板中的控件绑定到公共属性启用的控件模板来轻松地更改中的控件上的属性值。本文演示如何使用模板绑定从控件模板执行数据绑定。

Xamarin.Forms 控件模板的简介

2018/7/13 • • Edit Online

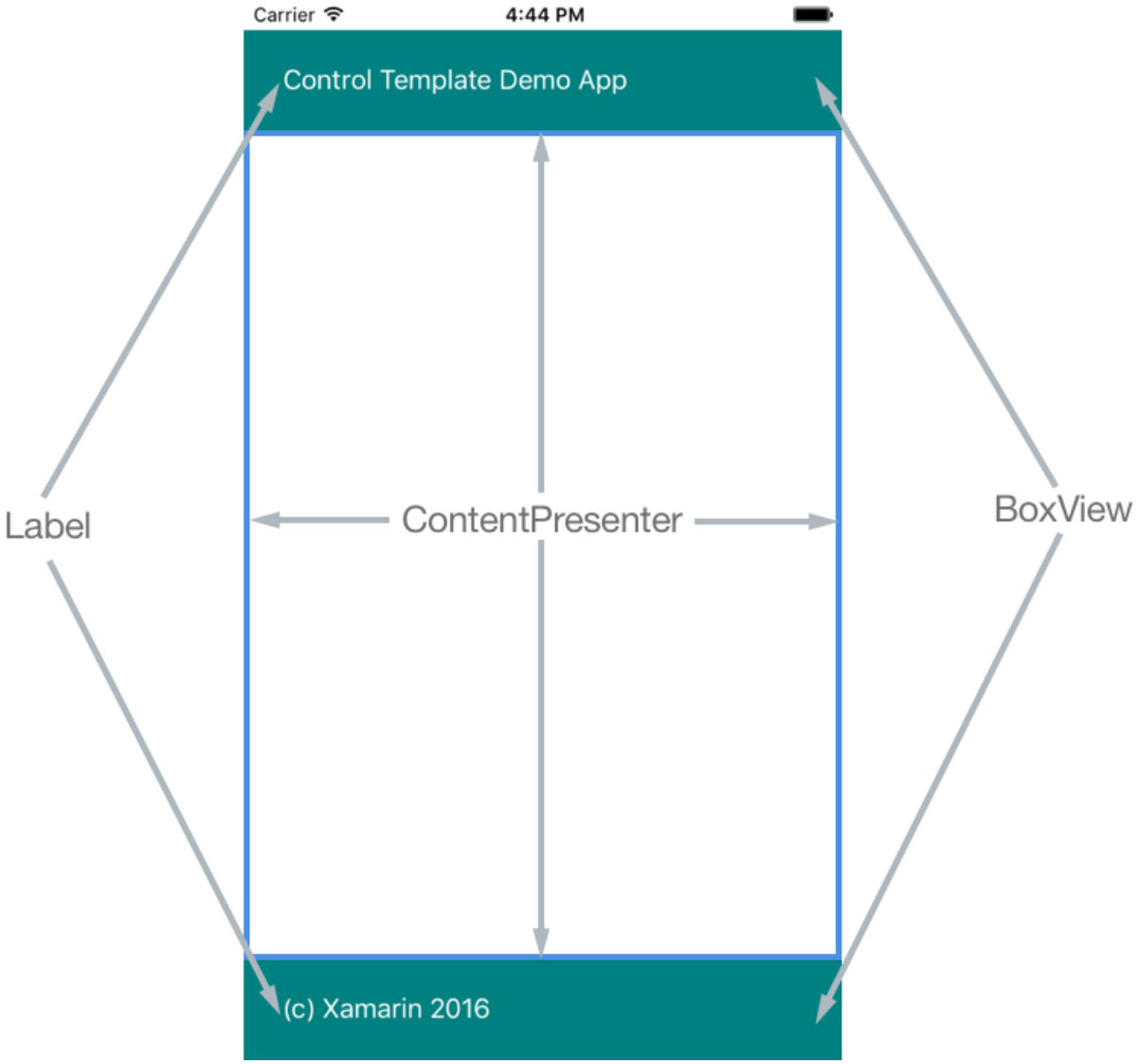
Xamarin.Forms 控件模板让你轻松设计或 re 主题可以在运行时的应用程序页。本文介绍控件模板。

控件具有不同的属性，如 `BackgroundColor` 和 `TextColor`，可以定义控件的外观的方面。可以使用设置这些属性样式，这可以在运行时实现基本主题设置更改。但是，样式不会维持页的外观和其内容之间完全分离，并且可以通过设置此类属性所做的更改受到限制。

控件模板提供页面的外观和其内容，从而可以轻松地进行主题化的页的创建之间完全分离。例如，应用程序可能包含提供深色主题和浅色主题的应用程序级别控件模板。每个 `ContentPage` 应用程序中可以有主题通过应用控件模板之一而无需更改按每页显示的内容。此外，提供的控件模板的主题并不限于更改控件的属性。它们还可以更改用于实现主题的控件。

创建 ControlTemplate

一个 `ControlTemplate` 指定外观的页面或视图中，并包含根布局和布局，这些控件可实现在模板中。通常情况下，`ControlTemplate` 将利用 `ContentPresenter` 来标记要显示的页面或视图的内容出现的位置。页面或使用的视图 `ControlTemplate` 然后，定义要显示的内容 `ContentPresenter`。下图演示了 `ControlTemplate` 包含多个控件，包括页面 `ContentPresenter` 由蓝色矩形标记：



一个 `ControlTemplate` 会应用于以下类型中，通过设置其 `ControlTemplate` 属性：

- `ContentPage`
- `ContentView`
- `TemplatedPage`
- `TemplatedView`

当 `ControlTemplate` 创建并分配到这些类型，任何现有的外观将替换中定义的外观 `ControlTemplate`。此外，以及使用设置外观 `ControlTemplate` 属性，还可以通过使用样式进一步应用模板的控件展开主题功能。

> **NOTE**
>
> *是什么 `TemplatedPage` 和 `TemplatedView` 类型？* `TemplatedPage` 是类的基类 `ContentPage`，并为 Xamarin.Forms 提供的最基本的页类型。与不同 `ContentPage`，`TemplatedPage` 没有 `Content` 属性。因此，内容不能直接添加到 `TemplatedPage` 实例。相反，通过设置的控件模板添加内容 `TemplatedPage` 实例。同样，`TemplatedView` 是类的基类 `ContentView`。与不同 `ContentView`，`TemplatedView` 没有 `Content` 属性。因此，内容不能直接添加到 `TemplatedView` 实例。相反，通过设置的控件模板添加内容 `TemplatedView` 实例。

在 XAML 和 C# 中，可以创建控件模板：

- 在 XAML 中创建的控件模板中定义 `ResourceDictionary` 分配给 `Resources` 集合页上，或者更常见的做法到 `Resources` 应用程序的集合。
- 在 C# 中创建的控件模板通常定义在页面的类中，或者可全局访问的类。

选择定义的位置 `ControlTemplate` 实例可以使用它的影响：

- `ControlTemplate` 在页面级别定义的实例只能应用到的页。
- `ControlTemplate` 在应用程序级别定义的实例可以应用于整个应用程序页。

控件模板的视图层次结构中较低级别优先于更高版本定义了。例如，`ControlTemplate` 名为 `DarkTheme` 定义在页级别将优先于在应用程序级别定义一个具有相同名称的模板。因此，应在应用程序级别定义控件模板，用于定义要应用于应用程序中的每个页面的主题。

相关链接

- 样式
- ControlTemplate
- ContentPresenter

创建 ControlTemplate

2018/7/13 • • Edit Online

可以在应用程序级别或页级别定义控件模板。本文演示如何创建和使用控件模板。

在 XAML 中创建 ControlTemplate

若要定义 `ControlTemplate` 应用程序级别 `ResourceDictionary` 必须添加到 `App` 类。默认情况下，所有从模板创建的 Xamarin.Forms 应用程序使用应用程序类，以实现 `Application` 子类。若要声明 `ControlTemplate` 在应用程序级别，在应用程序的 `ResourceDictionary` 使用 XAML，默认应用类必须替换 XAML应用类和关联的代码隐藏，作为下面的代码示例所示：

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.App">
    <Application.Resources>
        <ResourceDictionary>
            <ControlTemplate x:Key="TealTemplate">
                <Grid>
                    ...
                    <BoxView ... />
                    <Label Text="Control Template Demo App"
                           TextColor="White"
                           VerticalOptions="Center" ... />
                    <ContentPresenter ... />
                    <BoxView Color="Teal" ... />
                    <Label Text="(c) Xamarin 2016"
                           TextColor="White"
                           VerticalOptions="Center" ... />
                </Grid>
            </ControlTemplate>
            <ControlTemplate x:Key="AquaTemplate">
                ...
            </ControlTemplate>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

每个 `ControlTemplate` 实例创建为一个可重用对象 `ResourceDictionary` 。这通过为每个声明提供一个唯一 `x:Key` 属性，为其提供中的描述性键 `ResourceDictionary` 。

下面的代码示例显示了关联 `App` 隐藏代码：

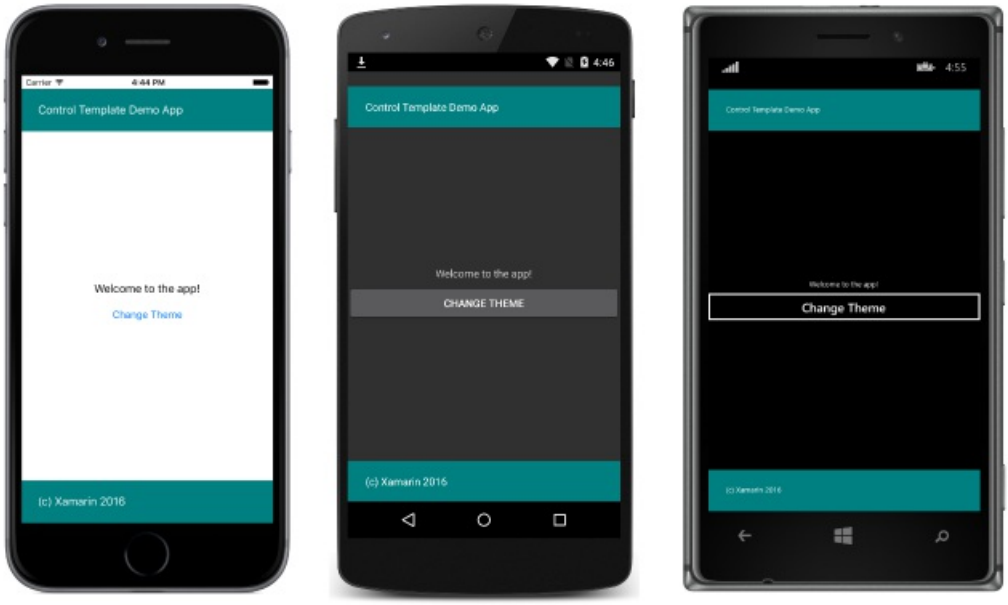
```
public partial class App : Application
{
  public App ()
  {
    InitializeComponent ();
    MainPage = new HomePage ();
  }
}
```

设置以及 `MainPage` 属性，还必须调用代码隐藏 `InitializeComponent` 方法来加载和分析相关联的 XAML。

下面的代码示例演示 `ContentPage` 应用 `TealTemplate` 到 `ContentView` ：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentView x:Name="contentView" Padding="0,20,0,0"
                 ControlTemplate="{StaticResource TealTemplate}">
        <StackLayout VerticalOptions="CenterAndExpand">
            <Label Text="Welcome to the app!" HorizontalOptions="Center" />
            <Button Text="Change Theme" Clicked="OnButtonClicked" />
        </StackLayout>
    </ContentView>
</ContentPage>
```

`TealTemplate`分配给`ContentView.ControlTemplate`属性使用`StaticResource`标记扩展。`ContentView.Content`属性设置为`StackLayout`，用于定义要在上显示的内容`ContentPage`。此内容将显示的`ContentPresenter`中包含`TealTemplate`。这会导致下面的屏幕截图中所示的外观：



Re 主题设置在运行时的应用程序

单击**更改主题**按钮执行`OnButtonClicked`方法，在下面的代码示例所示：

```
void OnButtonClicked (object sender, EventArgs e)
{
  originalTemplate = !originalTemplate;
  contentView.ControlTemplate = (originalTemplate) ? tealTemplate : aquaTemplate;
}
```

此方法替换活动`ControlTemplate`实例使用的替代方案`ControlTemplate`实例，从而导致以下屏幕截图：

> **NOTE**
>
> 上 `ContentPage`，则 `Content` 可以将属性分配和 `ControlTemplate` 还可以设置属性。在这种情况，如果 `ControlTemplate` 包含 `ContentPresenter` 实例，分配到的内容 `Content` 属性将显示由 `ContentPresenter` 内 `ControlTemplate` 。

设置样式的 ControlTemplate

一个 `ControlTemplate` 还可以通过应用 `Style` 以进一步展开主题功能。这可以通过创建实现*隐式*或*显式*样式中的目标视图 `ResourceDictionary` ，以及设置 `ControlTemplate` 目标属性在中查看 `Style` 实例。下面的代码示例演示*隐式*已添加到应用程序级别的样式 `ResourceDictionary` ：

```
<Style TargetType="ContentView">
    <Setter Property="ControlTemplate" Value="{StaticResource TealTemplate}" />
</Style>
```

因为 `Style` 实例*隐式*，它将应用于所有 `ContentView` 中应用程序的实例。因此，它不再需要设置 `ContentView.ControlTemplate` 属性，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentView x:Name="contentView" Padding="0,20,0,0">
      ...
    </ContentView>
</ContentPage>
```

有关样式的详细信息，请参阅样式。

在页面级别创建 ControlTemplate

除了创建之外 `ControlTemplate` 在应用程序级别的实例，则可以也在创建这些页面级别中，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="SimpleTheme.HomePage">
    <ContentPage.Resources>
        <ResourceDictionary>
            <ControlTemplate x:Key="TealTemplate">
                ...
            </ControlTemplate>
            <ControlTemplate x:Key="AquaTemplate">
                ...
            </ControlTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentView ... ControlTemplate="{StaticResource TealTemplate}">
        ...
    </ContentView>
</ContentPage>
```

添加时 `ControlTemplate` 级别的页 `ResourceDictionary` 添加到 `ContentPage`，然后 `ControlTemplate` 将包括实例在 `ResourceDictionary`。

在 C 中创建 ControlTemplate#

若要定义 `ControlTemplate` 应用程序级别 `class` 必须创建，它表示 `ControlTemplate`。类应派生自布局使用模板，如下面的代码示例中所示：

```
class TealTemplate : Grid
{
  public TealTemplate ()
  {
    ...
    var contentPresenter = new ContentPresenter ();
    Children.Add (contentPresenter, 0, 1);
    Grid.SetColumnSpan (contentPresenter, 2);
    ...
  }
}

class AquaTemplate : Grid
{
  ...
}
```

`AquaTemplate` 类等同于 `TealTemplate` 类，用于不同的颜色相似，只不过 `BoxView.Color` 并 `Label.TextColor` 属性。

下面的代码示例演示 `ContentPage` 应用 `TealTemplate` 到 `ContentView`：

```
public class HomePageCS : ContentPage
{
  ...
  ControlTemplate tealTemplate = new ControlTemplate (typeof(TealTemplate));
  ControlTemplate aquaTemplate = new ControlTemplate (typeof(AquaTemplate));

  public HomePageCS ()
  {
    var button = new Button { Text = "Change Theme" };
    var contentView = new ContentView {
      Padding = new Thickness (0, 20, 0, 0),
      Content = new StackLayout {
        VerticalOptions = LayoutOptions.CenterAndExpand,
        Children = {
          new Label { Text = "Welcome to the app!", HorizontalOptions = LayoutOptions.Center },
          button
        }
      },
      ControlTemplate = tealTemplate
    };
    ...
    Content = contentView;
  }
}
```

`ControlTemplate` 通过指定定义控件模板中的类的类型创建实例 `ControlTemplate` 构造函数。

`ContentView.Content` 属性设置为 `StackLayout`，用于定义要在上显示的内容 `ContentPage`。此内容将显示的 `ContentPresenter` 中包含 `TealTemplate`。所述的相同机制以前用于更改处于运行时主题 `AquaTheme`。

总结

本文演示了如何创建和使用控件模板。可以在应用程序级别或页级别定义控件模板。

相关链接

- 样式
- 简单主题（示例）
- ControlTemplate
- ContentPresenter
- ContentView
- ResourceDictionary

从 Xamarin.Forms ControlTemplate 绑定

2018/10/26 • • Edit Online

模板绑定允许对数据的控件模板中的控件绑定到公共属性，使控件模板来轻松地更改中的控件上的属性值。本文演示如何使用模板绑定控件模板中执行数据绑定。

一个 `TemplateBinding` 用于将控件模板中的控件的属性绑定到可绑定的属性的父级上*目标*拥有控件模板的视图。例如，不同于定义显示的文本 `Label` 实例内 `ControlTemplate`，您可以使用模板绑定来绑定 `Label.Text` 属性设置为可绑定属性用于定义要显示的文本。

一个 `TemplateBinding` 类似于现有 `Binding`，只不过*源*的 `TemplateBinding` 始终自动设置为父级的*目标*拥有控件模板的视图。但是，请注意，使用 `TemplateBinding` 之外 `ControlTemplate` 不受支持。

在 XAML 中创建 TemplateBinding

在 XAML 中，`TemplateBinding` 使用创建 `TemplateBinding` 标记扩展，如下面的代码示例中所示：

```
<ControlTemplate x:Key="TealTemplate">
  <Grid>
    ...
    <Label Text="{TemplateBinding Parent.HeaderText}" ... />
    ...
    <Label Text="{TemplateBinding Parent.FooterText}" ... />
  </Grid>
</ControlTemplate>
```

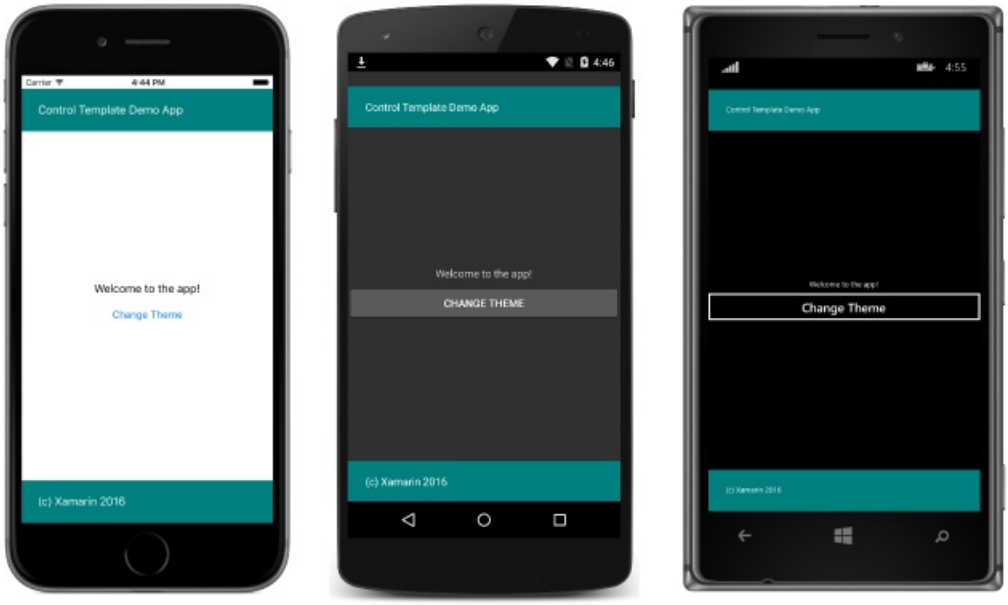
而不是设置 `Label.Text` 静态文本的属性，属性可以使用模板绑定绑定到可绑定属性的父级上*目标*视图拥有 `ControlTemplate`. 但请注意模板绑定绑定到 `Parent.HeaderText` 并 `Parent.FooterText`，而非 `HeaderText` 和 `FooterText`。这是因为在此示例中，可绑定属性定义上的祖父*目标*查看，而不是父项，如下面的代码示例中所示：

```
<ContentPage ...>
    <ContentView ... ControlTemplate="{StaticResource TealTemplate}">
          ...
    </ContentView>
</ContentPage>
```

*源*模板的绑定始终自动设置为父级的*目标*拥有控件模板，此处为视图 `ContentView` 实例。绑定使用的模板 `Parent` 属性返回的父元素 `ContentView` 实例，这是 `ContentPage` 实例。因此，使用 `TemplateBinding` 中 `ControlTemplate` 要绑定到 `Parent.HeaderText` 并 `Parent.FooterText` 定位定义的可绑定属性 `ContentPage`，作为以下代码示例所示：

```
public static readonly BindableProperty HeaderTextProperty =
  BindableProperty.Create ("HeaderText", typeof(string), typeof(HomePage), "Control Template Demo App");
public static readonly BindableProperty FooterTextProperty =
  BindableProperty.Create ("FooterText", typeof(string), typeof(HomePage), "(c) Xamarin 2016");

public string HeaderText {
  get { return (string)GetValue (HeaderTextProperty); }
}

public string FooterText {
  get { return (string)GetValue (FooterTextProperty); }
}
```

这会导致下面的屏幕截图中所示的外观：



在 C 中创建 TemplateBinding#

在 C# 中，`TemplateBinding` 通过创建 `TemplateBinding` 构造函数，如下面的代码示例中所示：

```
class TealTemplate : Grid
{
  public TealTemplate ()
  {
    ...
    var topLabel = new Label { TextColor = Color.White, VerticalOptions = LayoutOptions.Center };
    topLabel.SetBinding (Label.TextProperty, new TemplateBinding ("Parent.HeaderText"));
    ...
    var bottomLabel = new Label { TextColor = Color.White, VerticalOptions = LayoutOptions.Center };
    bottomLabel.SetBinding (Label.TextProperty, new TemplateBinding ("Parent.FooterText"));
    ...
  }
}
```

而不是设置 `Label.Text` 静态文本的属性，属性可以使用模板绑定绑定到可绑定属性的父级上*目标*视图拥有 `ControlTemplate`。使用创建的模板绑定 `SetBinding` 方法中，指定 `TemplateBinding` 实例作为第二个参数。请注意，模板绑定绑定到 `Parent.HeaderText` 并 `Parent.FooterText`，因为可绑定属性定义上的祖父级*目标*查看，而不是父项，如下面的代码示例中所示：

```
public class HomePageCS : ContentPage
{
  ...
  public HomePageCS ()
  {
    Content = new ContentView {
      ControlTemplate = tealTemplate,
      Content = new StackLayout {
        ...
      },
      ...
    };
    ...
  }
}
```

可绑定属性上定义 `ContentPage`，如前面所述。

绑定到 ViewModel 属性 BindableProperty

如前面所述，`TemplateBinding` 将在控件模板中的控件的属性绑定到可绑定的属性的父级上*目标*拥有控件模板的视图。反过来，这些可绑定属性可以绑定到 Viewmodel 中的属性。

下面的代码示例在 ViewModel 上定义两个属性：

```
public class HomePageViewModel
{
  public string HeaderText { get { return "Control Template Demo App"; } }
  public string FooterText { get { return "(c) Xamarin 2016"; } }
}
```

`HeaderText` 和 `FooterText` ViewModel 属性可以绑定到，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns:local="clr-namespace:SimpleTheme;assembly=SimpleTheme"
             HeaderText="{Binding HeaderText}" FooterText="{Binding FooterText}" ...>
    <ContentPage.BindingContext>
        <local:HomePageViewModel />
    </ContentPage.BindingContext>
    <ContentView ControlTemplate="{StaticResource TealTemplate}" ...>
    ...
    </ContentView>
</ContentPage>
```

`HeaderText` 并 `FooterText` 可绑定属性绑定到 `HomePageViewModel.HeaderText` 并 `HomePageViewModel.FooterText` 属性，由于设置 `BindingContext` 实例 `HomePageViewModel` 类。总体而言，这会导致控件中的属性 `ControlTemplate` 要绑定到 `BindableProperty` 上实例 `ContentPage`，又将其绑定到ViewModel 属性。

以下代码示例显示相应的 C# 代码：

```
public class HomePageCS : ContentPage
{
  ...
  public HomePageCS ()
  {
    BindingContext = new HomePageViewModel ();
    this.SetBinding (HeaderTextProperty, "HeaderText");
    this.SetBinding (FooterTextProperty, "FooterText");
    ...
  }
}
```

您还可以绑定到视图模型属性直接，以便无需声明 `BindableProperty` s `HeaderText` 并 `FooterText` 上 `ContentPage`，通过将控件模板绑定到 Parent.BindingContext。_PropertyName_例如：

```
<ControlTemplate x:Key="TealTemplate">
  <Grid>
    ...
    <Label Text="{TemplateBinding Parent.BindingContext.HeaderText}" ... />
    ...
    <Label Text="{TemplateBinding Parent.BindingContext.FooterText}" ... />
  </Grid>
</ControlTemplate>
```

有关数据绑定到 Viewmodel 的详细信息，请参阅从数据绑定到 MVVM。

总结

使用模板绑定控件模板中执行数据绑定演示这篇文章。模板绑定允许对数据的控件模板中的控件绑定到公共属性，使控件模板来轻松地更改中的控件上的属性值。

相关链接

- 数据绑定基础知识
- 从数据绑定到 MVVM
- 使用模板绑定（示例）的简单主题
- 使用模板绑定和 ViewModel（示例）的简单主题
- TemplateBinding
- ControlTemplate
- ContentView

Xamarin.Forms 数据模板

2018/7/13 • • Edit Online

DataTemplate 用于支持的控件上指定数据的外观并通常将绑定到数据显示。

介绍

Xamarin.Forms 数据模板提供支持的控件上定义数据的表示形式的功能。本文介绍数据模板，检查它们是必需的原因。

创建 DataTemplate

数据模板可以在中创建内联 `ResourceDictionary`，或从自定义类型或适当的 Xamarin.Forms 单元格类型。如果不需要重复利用其他位置的数据模板，应使用内联模板。或者，可以通过定义它作为自定义的类型，或作为控件级别页级别或应用程序级别资源重复使用数据模板。

创建 DataTemplateSelector

一个 `DataTemplateSelector` 可用于选择 `DataTemplate` 在运行时根据数据绑定属性的值。这使多个 `DataTemplate` 实例应用于相同类型的对象，若要自定义的特定对象的外观。本文演示了如何创建和使用 `DataTemplateSelector`。

相关链接

- 数据模板（示例）

Xamarin.Forms 数据模板的简介

2018/7/13 • • Edit Online

Xamarin.Forms 数据模板提供支持的控件上定义数据的表示形式的功能。本文介绍数据模板，检查它们是必需的原因。

请考虑 `ListView` 显示的集合 `Person` 对象。下面的代码示例显示的定义 `Person` 类：

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public string Location { get; set; }
}
```

`Person` 类定义 `Name`， `Age`，和 `Location` 属性时可以设置 `Person` 创建对象。 `ListView` 用来显示的集合 `Person` 对象，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataTemplates"
             ...>
    <StackLayout Margin="20">
        ...
        <ListView Margin="0,20,0,0">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    <local:Person Name="John" Age="37" Location="USA" />
                    <local:Person Name="Tom" Age="42" Location="UK" />
                    <local:Person Name="Lucas" Age="29" Location="Germany" />
                    <local:Person Name="Tariq" Age="39" Location="UK" />
                    <local:Person Name="Jane" Age="30" Location="USA" />
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </StackLayout>
</ContentPage>
```

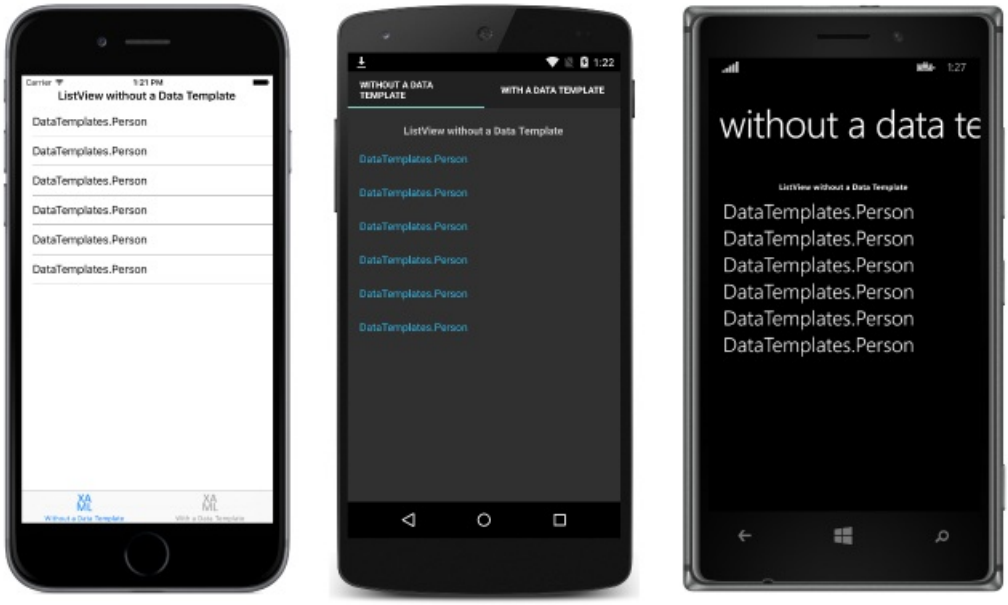
将项添加到 `ListView` 中通过初始化 XAML `ItemsSource` 属性的数组从 `Person` 实例。

> **NOTE**
>
> 请注意， `x:Array` 元素需要 `Type` 属性，指示数组中的项的类型。

在以下代码示例中，这将初始化显示等效的 C# 页面 `ItemsSource` 属性设置为 `List` 的 `Person` 实例：

```
public WithoutDataTemplatePageCS()
{
    ...
    var people = new List<Person>
    {
        new Person { Name = "Steve", Age = 21, Location = "USA" },
        new Person { Name = "John", Age = 37, Location = "USA" },
        new Person { Name = "Tom", Age = 42, Location = "UK" },
        new Person { Name = "Lucas", Age = 29, Location = "Germany" },
        new Person { Name = "Tariq", Age = 39, Location = "UK" },
        new Person { Name = "Jane", Age = 30, Location = "USA" }
    };

    Content = new StackLayout
    {
        Margin = new Thickness(20),
        Children = {
            ...
            new ListView { ItemsSource = people, Margin = new Thickness(0, 20, 0, 0) }
        }
    };
}
```

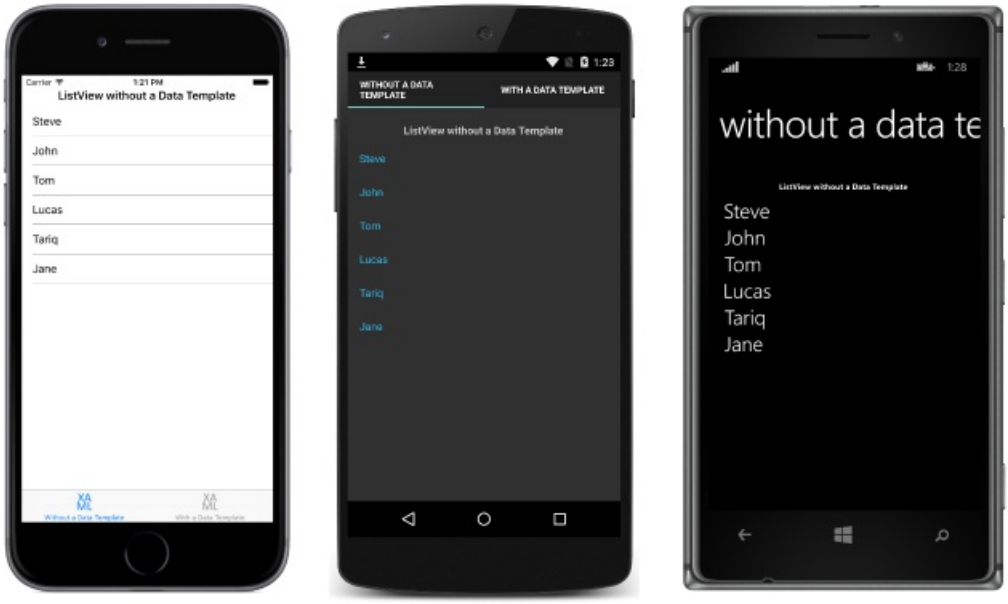
`ListView` 调用 `ToString` 时显示的对象集合中。因为没有任何 `Person.ToString` 重写，`ToString` 返回的每个对象的类型名称，如以下屏幕截图中所示：



`Person` 对象可以重写 `ToString` 方法来显示有意义的数据，如下面的代码示例中所示：

```
public class Person
{
  ...
  public override string ToString ()
  {
    return Name;
  }
}
```

这会导致 `ListView` 显示 `Person.Name` 属性值为每个对象在集合中，如以下屏幕截图中所示：



`Person.ToString`重写可能会返回格式化的字符串组成`Name`，`Age`，和`Location`属性。但是，此方法提供仅有限的控制数据的每个项的外观。为提高灵活性`DataTemplate`可以创建用于定义数据的外观。

创建 DataTemplate

一个`DataTemplate`用于指定数据的外观，通常使用数据绑定来显示数据。显示中的对象的集合中的数据时，其常见使用方案`ListView`。例如，当`ListView`绑定到的集合`Person`对象，`ListView.ItemTemplate`属性将设置为`DataTemplate`定义的每个外观`Person`对象中`ListView`。`DataTemplate`将包含绑定到的每个属性值的元素`Person`对象。若要深入了解数据绑定，请参阅数据绑定基本知识。

一个`DataTemplate`可以用作以下属性值：

- `ListView.HeaderTemplate`
- `ListView.FooterTemplate`
- `ListView.GroupHeaderTemplate`
- `ItemsView.ItemTemplate`由继承`ListView`。
- `MultiPage.ItemTemplate`由继承`CarouselPage`，`MasterDetailPage`，以及`TabbedPage`。

> **NOTE**
> 请注意，虽然`TableView`使用`Cell`对象，它不使用`DataTemplate`。这是因为数据绑定始终设置上直接`Cell`对象。

一个`DataTemplate`如上面列出的属性的直接子级称为放置*内联模板*。或者，`DataTemplate`可以定义为控件级别、页面级别或应用程序级资源。选择定义的位置`DataTemplate`可以使用它的影响：

- 一个`DataTemplate`定义在控件级别只能应用到控件。
- 一个`DataTemplate`定义的页级别可以应用于页面上的多个有效控件。
- 一个`DataTemplate`在应用程序级别定义可以是整个应用程序应用于有效控件。

数据模板的视图层次结构中较低级别优先于那些共享时，更高版本定义了`x:Key`属性。例如，一个页级别的数据模板，将重写应用程序级别的数据模板和的控制级别的数据模板或内联数据模板，将重写页面级数据模板。

相关链接

- 单元格外观
- 数据模板（示例）

- 数据模板

创建 Xamarin.Forms DataTemplate

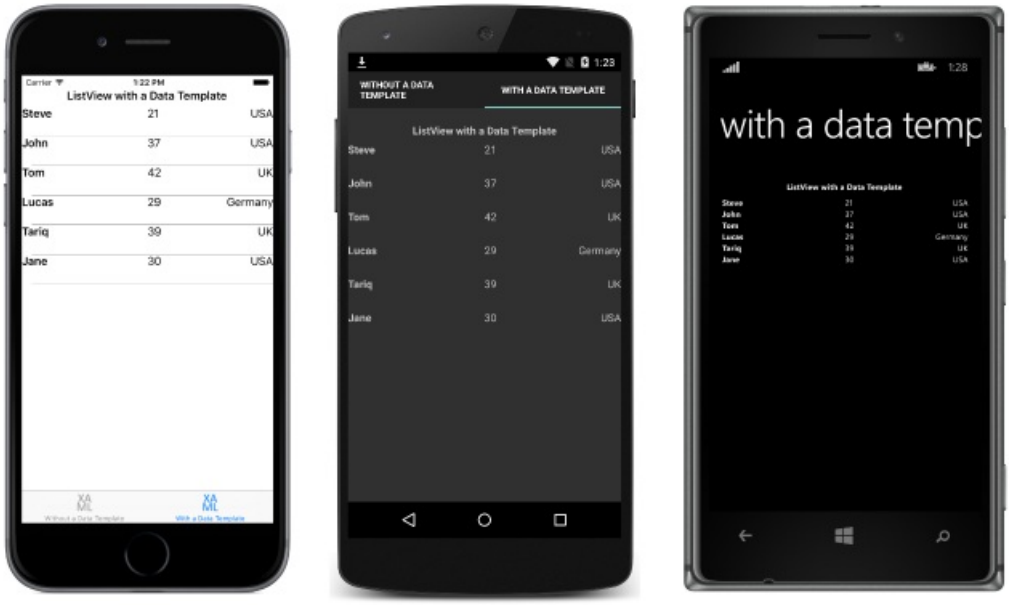
2018/7/13 • • Edit Online

数据模板可以以内联方式创建的 ResourceDictionary、中或从自定义类型或适当的 Xamarin.Forms 单元格类型。本文探讨了每种技术。

有关的常见使用方案 `DataTemplate` 显示在对象的集合中的数据 `ListView` 。在每个单元格的数据的外观 `ListView` 可以设置管理 `ListView.ItemTemplate` 属性设置为 `DataTemplate` 。有许多可以用于实现此目的的方法：

- 创建内联 DataTemplate。
- 使用一种类型创建 DataTemplate。
- 创建为资源 DataTemplate。

无论所使用的技术，其结果是，在每个单元格的外观 `ListView` 由定义 `DataTemplate` ，如以下屏幕截图所示：



创建内联 DataTemplate

`ListView.ItemTemplate` 属性可以设置为内联 `DataTemplate` 。如果不需要重复利用其他位置的数据模板，应使用一个内联模板，这是指位于作为相应的控件属性的直接子级。中指定的元素 `DataTemplate` 定义每个单元格的外观，如下面的 XAML 代码示例中所示：

```
<ListView Margin="0,20,0,0">
    <ListView.ItemsSource>
        <x:Array Type="{x:Type local:Person}">
            <local:Person Name="Steve" Age="21" Location="USA" />
            <local:Person Name="John" Age="37" Location="USA" />
            <local:Person Name="Tom" Age="42" Location="UK" />
            <local:Person Name="Lucas" Age="29" Location="Germany" />
            <local:Person Name="Tariq" Age="39" Location="UK" />
            <local:Person Name="Jane" Age="30" Location="USA" />
        </x:Array>
    </ListView.ItemsSource>
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
                <Grid>
                    ...
                    <Label Text="{Binding Name}" FontAttributes="Bold" />
                    <Label Grid.Column="1" Text="{Binding Age}" />
                    <Label Grid.Column="2" Text="{Binding Location}" HorizontalTextAlignment="End" />
                </Grid>
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

内嵌元素的子 `DataTemplate` 必须是的或从派生，键入 `ViewCell` 。内的布局 `ViewCell` 此处由 `Grid` 。 `Grid` 包含三种 `Label` 实例的绑定及其 `Text` 属性设置为相应的属性的每个 `Person` 集合中的对象。

以下代码示例显示相应的 C# 代码：

```
public class WithDataTemplatePageCS : ContentPage
{
    public WithDataTemplatePageCS()
    {
        ...
        var people = new List<Person>
        {
            new Person { Name = "Steve", Age = 21, Location = "USA" },
            ...
        };

        var personDataTemplate = new DataTemplate(() =>
        {
            var grid = new Grid();
            ...
            var nameLabel = new Label { FontAttributes = FontAttributes.Bold };
            var ageLabel = new Label();
            var locationLabel = new Label { HorizontalTextAlignment = TextAlignment.End };

            nameLabel.SetBinding(Label.TextProperty, "Name");
            ageLabel.SetBinding(Label.TextProperty, "Age");
            locationLabel.SetBinding(Label.TextProperty, "Location");

            grid.Children.Add(nameLabel);
            grid.Children.Add(ageLabel, 1, 0);
            grid.Children.Add(locationLabel, 2, 0);

            return new ViewCell { View = grid };
        });

        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children = {
                ...
                new ListView { ItemsSource = people, ItemTemplate = personDataTemplate, Margin = new
Thickness(0, 20, 0, 0) }
            }
        };
    }
}
```

在 C# 中，内联 `DataTemplate` 创建使用指定的构造函数重载 `Func` 参数。

使用一种类型创建 DataTemplate

`ListView.ItemTemplate` 属性也设置为 `DataTemplate` 创建从单元格类型。此方法的优点是整个应用程序的多个数据模板可以重用单元格类型所定义的外观。下面的 XAML 代码演示此方法的一个示例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DataTemplates"
             ...>
    <StackLayout Margin="20">
        ...
        <ListView Margin="0,20,0,0">
           <ListView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    ...
                </x:Array>
            </ListView.ItemsSource>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <local:PersonCell />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

在这里，`ListView.ItemTemplate` 属性设置为 `DataTemplate` 创建自定义单元格的外观的自定义类型。自定义的类型必须派生自类型 `ViewCell`，下面的代码示例中所示：

```
<ViewCell xmlns="http://xamarin.com/schemas/2014/forms"
          xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
          x:Class="DataTemplates.PersonCell">
     <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="0.5*" />
            <ColumnDefinition Width="0.2*" />
            <ColumnDefinition Width="0.3*" />
        </Grid.ColumnDefinitions>
        <Label Text="{Binding Name}" FontAttributes="Bold" />
        <Label Grid.Column="1" Text="{Binding Age}" />
        <Label Grid.Column="2" Text="{Binding Location}" HorizontalTextAlignment="End" />
    </Grid>
</ViewCell>
```

内 `ViewCell`，通过此处管理布局 `Grid`。`Grid` 包含三种 `Label` 实例的绑定及其 `Text` 属性设置为相应的属性的每个 `Person` 集合中的对象。

等效的 C# 代码如以下示例所示：

```
public class WithDataTemplatePageFromTypeCS : ContentPage
{
    public WithDataTemplatePageFromTypeCS()
    {
        ...
        var people = new List<Person>
        {
            new Person { Name = "Steve", Age = 21, Location = "USA" },
            ...
        };

        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children = {
                ...
                new ListView { ItemTemplate = new DataTemplate(typeof(PersonCellCS)), ItemsSource = people,
Margin = new Thickness(0, 20, 0, 0) }
            }
        };
    }
}
```

在 C# 中，`DataTemplate` 创建使用指定的单元格类型作为参数的构造函数重载。单元格类型必须派生自类型 `ViewCell`，下面的代码示例中所示：

```
public class PersonCellCS : ViewCell
{
    public PersonCellCS()
    {
        var grid = new Grid();
        ...
        var nameLabel = new Label { FontAttributes = FontAttributes.Bold };
        var ageLabel = new Label();
        var locationLabel = new Label { HorizontalTextAlignment = TextAlignment.End };

        nameLabel.SetBinding(Label.TextProperty, "Name");
        ageLabel.SetBinding(Label.TextProperty, "Age");
        locationLabel.SetBinding(Label.TextProperty, "Location");

        grid.Children.Add(nameLabel);
        grid.Children.Add(ageLabel, 1, 0);
        grid.Children.Add(locationLabel, 2, 0);

        View = grid;
    }
}
```

> **NOTE**
>
> 请注意，Xamarin.Forms 还包括可用于显示简单的数据中的单元格类型 `ListView` 单元格。有关详细信息，请参阅单元格的外观。

DataTemplate 创建为资源

数据模板也可以创建可重用对象作为 `ResourceDictionary` 。这通过为每个声明提供一个唯一 `x:Key` 属性，为其提供中的描述性键 `ResourceDictionary`，如以下 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             ...>
    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="personTemplate">
                 <ViewCell>
                    <Grid>
                        ...
                    </Grid>
                </ViewCell>
            </DataTemplate>
        </ResourceDictionary>
    </ContentPage.Resources>
    <StackLayout Margin="20">
        ...
        <ListView ItemTemplate="{StaticResource personTemplate}" Margin="0,20,0,0">
            <ListView.ItemsSource>
                <x:Array Type="{x:Type local:Person}">
                    <local:Person Name="Steve" Age="21" Location="USA" />
                    ...
                </x:Array>
            </ListView.ItemsSource>
        </ListView>
    </StackLayout>
</ContentPage>
```

`DataTemplate` 分配给 `ListView.ItemTemplate` 属性使用 `StaticResource` 标记扩展。请注意，当 `DataTemplate` 页中定义 `ResourceDictionary`，也可以在控件级别或应用程序级别定义。

下面的代码示例显示了 C# 中的等效页：

```
public class WithDataTemplatePageCS : ContentPage
{
  public WithDataTemplatePageCS ()
  {
    ...
    var personDataTemplate = new DataTemplate (() => {
      var grid = new Grid ();
      ...
      return new ViewCell { View = grid };
    });

    Resources = new ResourceDictionary ();
    Resources.Add ("personTemplate", personDataTemplate);

    Content = new StackLayout {
      Margin = new Thickness(20),
      Children = {
        ...
        new ListView { ItemTemplate = (DataTemplate)Resources ["personTemplate"], ItemsSource = people };
      }
    };
  }
}
```

`DataTemplate` 添加到 `ResourceDictionary` 使用 `Add` 方法，后者指定 `Key` 用于字符串引用 `DataTemplate` 时检索它。

总结

本文介绍了如何从自定义类型，或在创建数据模板、内联 `ResourceDictionary` 。如果不需要重复利用其他位置的数据模板，应使用内联模板。或者，可以通过定义它作为自定义的类型，或作为控件级别页级别或应用程序级别资

源重复使用数据模板。

相关链接

- 单元格外观
- 数据模板（示例）
- 数据模板

创建 Xamarin.Forms DataTemplateSelector

2018/7/13 • • Edit Online

DataTemplateSelector 可以用于选择在运行时根据数据绑定属性的值的 DataTemplate。这使多个 DataTemplates 以应用于相同类型的对象，若要自定义的特定对象的外观。本文演示如何创建和使用 DataTemplateSelector。

数据模板选择器启用方案，例如 `ListView` 绑定到对象的集合，其中在每个对象的外观的 `ListView` 可以在运行时中返回的数据模板选择器选择特定 `DataTemplate` 。

创建 DataTemplateSelector

通过创建继承的类实现一个数据模板选择器 `DataTemplateSelector` 。 `OnSelectTemplate` 方法然后重写以返回特定 `DataTemplate` ，下面的代码示例中所示：

```
public class PersonDataTemplateSelector : DataTemplateSelector
{
  public DataTemplate ValidTemplate { get; set; }
  public DataTemplate InvalidTemplate { get; set; }

  protected override DataTemplate OnSelectTemplate (object item, BindableObject container)
  {
    return ((Person)item).DateOfBirth.Year >= 1980 ? ValidTemplate : InvalidTemplate;
  }
}
```

`OnSelectTemplate` 方法返回值的基础的相应模板 `DateOfBirth` 属性。要返回的模板是的值 `ValidTemplate` 属性或 `InvalidTemplate` 属性，使用时设置 `PersonDataTemplateSelector` 。

数据模板选择器类的实例可以然后赋给 Xamarin.Forms 控件属性如 `ListView.ItemTemplate` 。有关有效的属性的列表，请参阅创建 DataTemplate。

限制

`DataTemplateSelector` 实例具有以下限制：

- `DataTemplateSelector` 子类必须始终会返回相同的数据的同一个模板，如果多次查询。
- `DataTemplateSelector` 子类不能返回另一个 `DataTemplateSelector` 子类。
- `DataTemplateSelector` 子类不能返回的新实例 `DataTemplate` 在每次调用。相反，必须返回相同的实例。如果不这样做会造成内存泄漏，将禁用虚拟化。
- 在 Android 上，可以每个不超过 20 个不同的数据模板 `ListView` 。

使用 XAML 中 DataTemplateSelector

在 XAML， `PersonDataTemplateSelector` 可以实例化通过声明为资源，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Selector;assembly=Selector"
x:Class="Selector.HomePage">
    <ContentPage.Resources>
        <ResourceDictionary>
            <DataTemplate x:Key="validPersonTemplate">
                <ViewCell>
                    ...
                </ViewCell>
            </DataTemplate>
            <DataTemplate x:Key="invalidPersonTemplate">
                <ViewCell>
                    ...
                </ViewCell>
            </DataTemplate>
            <local:PersonDataTemplateSelector x:Key="personDataTemplateSelector"
                ValidTemplate="{StaticResource validPersonTemplate}"
                InvalidTemplate="{StaticResource invalidPersonTemplate}" />
        </ResourceDictionary>
    </ContentPage.Resources>
  ...
</ContentPage>
```

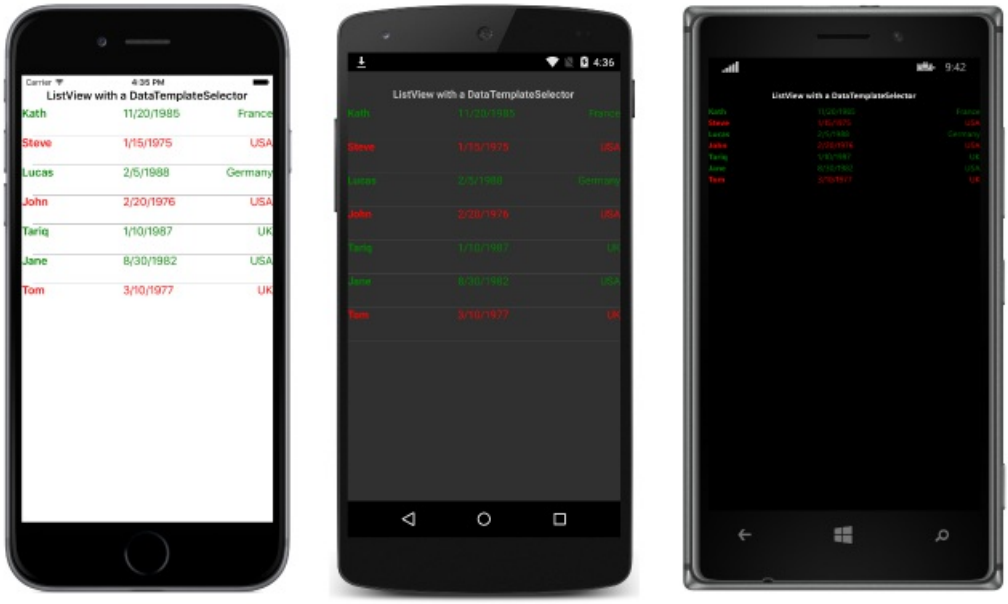
此页面级 `ResourceDictionary` 定义了两个 `DataTemplate` 实例和一个 `PersonDataTemplateSelector` 实例。 `PersonDataTemplateSelector` 实例设置其 `ValidTemplate` 和 `InvalidTemplate` 属性设置为相应 `DataTemplate` 实例使用 `StaticResource` 标记扩展。请注意，当资源页中定义 `ResourceDictionary` ，它们也可以在控件级别或应用程序级别定义。

`PersonDataTemplateSelector` 实例使用的将其分配给 `ListView.ItemTemplate` 属性，如下面的代码示例中所示：

```
<ListView x:Name="listView" ItemTemplate="{StaticResource personDataTemplateSelector}" />
```

在运行时， `ListView` 调用 `PersonDataTemplateSelector.OnSelectTemplate` 方法为每个调用传递数据对象作为基础集合中项 `item` 参数。 `DataTemplate` 返回该方法然后应用于该对象。

以下屏幕截图显示的结果 `ListView` 应用 `PersonDataTemplateSelector` 到基础集合中每个对象：



任何 `Person` 对象，它具有 `DateOfBirth` 属性值大于或等于 1980年中显示为绿色，使用剩余的对象显示为红色。

使用 C 中 DataTemplateSelector#

在 C# 中，`PersonDataTemplateSelector` 可实例化和分配给 `ListView.ItemTemplate` 属性，如下面的代码示例中所示：

```
public class HomePageCS : ContentPage
{
  DataTemplate validTemplate;
  DataTemplate invalidTemplate;

  public HomePageCS ()
  {
    ...
    SetupDataTemplates ();
    var listView = new ListView {
      ItemsSource = people,
      ItemTemplate = new PersonDataTemplateSelector {
        ValidTemplate = validTemplate,
        InvalidTemplate = invalidTemplate }
    };

    Content = new StackLayout {
      Margin = new Thickness (20),
      Children = {
        ...
        listView
      }
    };
  }
  ...
}
```

`PersonDataTemplateSelector` 实例设置其 `ValidTemplate` 并 `InvalidTemplate` 属性设置为相应 `DataTemplate` 创建的实例 `SetupDataTemplates` 方法。在运行时，`ListView` 调用 `PersonDataTemplateSelector.OnSelectTemplate` 方法为每个调用传递数据对象作为基础集合中项 `item` 参数。`DataTemplate` 返回该方法然后应用于该对象。

总结

本文演示了如何创建和使用 `DataTemplateSelector` 。一个 `DataTemplateSelector` 可用于选择 `DataTemplate` 在运行时根据数据绑定属性的值。这使多个 `DataTemplate` 实例应用于相同类型的对象，若要自定义的特定对象的外观。

相关链接

- 数据模板选择器（示例）
- DataTemplateSelector

Xamarin.Forms 触发器

2018/11/1 • • Edit Online

触发器可以表示 XAML 中声明的方式更改的基于事件或属性更改控件外观的操作。

可以一个触发器将直接分配给一个控件，或将其添加到要应用于多个控件的页级别或应用程序级资源字典。

有四种类型的触发器：

- 属性触发器-上一个控件的属性设置为特定值时，会发生。
- 数据触发器-使用数据绑定到触发器基于另一个控件的属性。
- 事件触发器-当事件发生在控件上时发生。
- 多触发器-允许多个操作发生之前要设置的触发器条件。

属性触发器

简单触发器可以表示完全在 XAML 中，添加 `Trigger` 触发回收时给控件的元素。此示例显示了更改的触发器 `Entry` 接收焦点时，背景色：

```
<Entry Placeholder="enter name">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
             Property="IsFocused" Value="True">
            <Setter Property="BackgroundColor" Value="Yellow" />
        </Trigger>
    </Entry.Triggers>
</Entry>
```

触发器的重要部分是声明的：

- **TargetType** -触发器应用于控件类型。
- **属性**-监视在控件上的属性。
- 值-的值，为受监视的属性时，导致触发器激活。
- **Setter** -一系列 `Setter` 可以添加元素，并且满足触发器条件时。必须指定 `Property` 和 `Value` 设置。
- **EnterActions 和 ExitActions** （未显示）-在代码中编写并可以在除（或 instead of） `Setter` 元素。它们是如下所述。

应用触发器，请使用一种样式

触发器还可以添加到 `Style` 页上或应用程序中的控件上的声明 `ResourceDictionary` 。此示例中声明的隐式样式 (ie。没有 `Key` 设置) 这意味着它将适用于所有 `Entry` 页上的控件。

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Style TargetType="Entry">
                        <Style.Triggers>
                <Trigger TargetType="Entry"
                         Property="IsFocused" Value="True">
                    <Setter Property="BackgroundColor" Value="Yellow" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>
```

数据触发器

数据触发器使用数据绑定来监视另一个控件来导致 `Setter` 以调用。而不是 `Property` 属性中的属性触发器，请设置 `Binding` 属性来监视指定的值。

下面的示例使用数据绑定语法 `{Binding Source={x:Reference entry}, Path=Text.Length}` 这是我们是如何引用另一个控件的属性。时的长度 `entry` 为零，激活触发器。在此示例中触发禁用的按钮时输入为空。

```
<!-- the x:Name is referenced below in DataTrigger-->
<!-- tip: make sure to set the Text="" (or some other default) -->
<Entry x:Name="entry"
       Text=""
       Placeholder="required field" />

<Button x:Name="button" Text="Save"
        FontSize="Large"
        HorizontalOptions="Center">
    <Button.Triggers>
        <DataTrigger TargetType="Button"
                     Binding="{Binding Source={x:Reference entry},
                                       Path=Text.Length}"
                     Value="0">
            <Setter Property="IsEnabled" Value="False" />
        </DataTrigger>
    </Button.Triggers>
</Button>
```

提示：评估时 `Path=Text.Length` 始终提供的目标属性（例如默认值。 `Text=""` ）因为否则它将为 `null` 和触发器不会使用起来就像您预期。

除了指定 `Setter` 还可以提供的 s `EnterActions` 并 `ExitActions` 。

事件触发器

`EventTrigger` 元素仅需要 `Event` 属性，如 `"Clicked"` 在下面的示例。

```
<EventTrigger Event="Clicked">
    <local:NumericValidationTriggerAction />
</EventTrigger>
```

请注意，有没有 `Setter` 元素，但而不是对定义的类的引用 `local:NumericValidationTriggerAction` 需要 `xmlns:local` 页面中声明的 XAML：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:WorkingWithTriggers;assembly=WorkingWithTriggers"
```

实现类本身 `TriggerAction` 这意味着它应提供的替代 `Invoke` 每当触发器事件发生时调用的方法。

触发器操作实现应：

- 实现泛型 `TriggerAction<T>` 类，与该触发器将应用于控件类型相对应的泛型参数。您可以使用如超类 `VisualElement` 编写适用于各种控件，或指定控件类型的触发器操作 `Entry` 。
- 重写 `Invoke` 方法-这称为只要满足触发器条件。
- 选择公开声明触发器时可以在 XAML 中设置属性（如 `Anchor` ， `Scale` ，和 `Length` 在此示例中）。

```
public class NumericValidationTriggerAction : TriggerAction<Entry>
{
    protected override void Invoke (Entry entry)
    {
        double result;
        bool isValid = Double.TryParse (entry.Text, out result);
        entry.TextColor = isValid ? Color.Default : Color.Red;
    }
}
```

由触发器操作公开的属性可以按如下所示设置 XAML 声明中：

```
<EventTrigger Event="TextChanged">
    <local:NumericValidationTriggerAction />
</EventTrigger>
```

共享中的触发器时要小心 `ResourceDictionary` ，以便对其所有应用将配置一次的任何状态，在控件之间共享一个实例。

请注意，不支持事件触发器 `EnterActions` 并 `ExitActions` 如下所述。

多触发器

一个 `MultiTrigger` 看起来类似于 `Trigger` 或 `DataTrigger` 只可以有多个条件。所有条件均都为 true，然后才能 `Setter` 触发 s。

下面是将绑定到两个不同的输入的按钮触发器的示例（ `email` 和 `phone` ）：

```
<MultiTrigger TargetType="Button">
    <MultiTrigger.Conditions>
        <BindingCondition Binding="{Binding Source={x:Reference email},
                                    Path=Text.Length}"
                          Value="0" />
        <BindingCondition Binding="{Binding Source={x:Reference phone},
                                    Path=Text.Length}"
                          Value="0" />
    </MultiTrigger.Conditions>

  <Setter Property="IsEnabled" Value="False" />
    <!-- multiple Setter elements are allowed -->
</MultiTrigger>
```

`Conditions` 还可以包含在集合 `PropertyCondition` 元素如下所示：

```
<PropertyCondition Property="Text" Value="OK" />
```

生成的"要求所有"多触发器

满足所有条件时，多触发器只更新其控件。测试为"所有字段长度零"（如登录页面，其中所有输入必须都是完整）是比较棘手，因为所需条件"其中 Text.Length > 0"，但这不能在 XAML 中表示。

这可以通过 `IValueConverter` 。以下转换转换器代码 `Text.Length` 绑定到 `bool` ，该值指示字段是否为空：

```
public class MultiTriggerConverter : IValueConverter
{
    public object Convert(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        if ((int)value > 0) // length > 0 ?
            return true;            // some data has been entered
        else
            return false;            // input is empty
    }

    public object ConvertBack(object value, Type targetType,
        object parameter, CultureInfo culture)
    {
        throw new NotSupportedException ();
    }
}
```

若要在多触发器中使用此转换器，首先将其添加到页面的资源字典（以及自定义 `xmlns:local` 命名空间定义）：

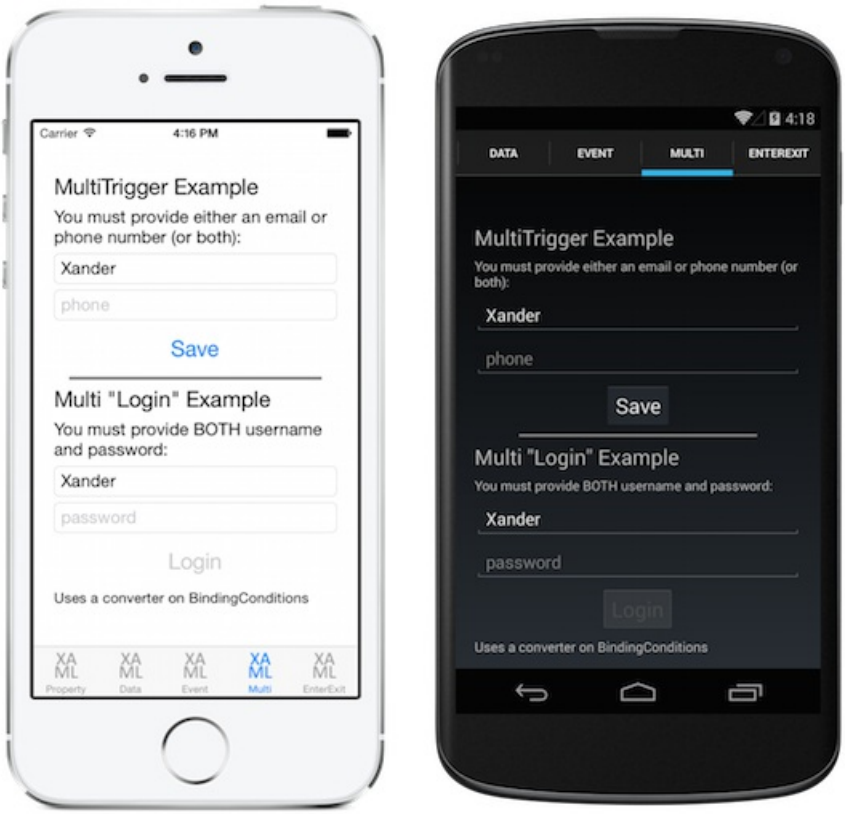
```
<ResourceDictionary>
   <local:MultiTriggerConverter x:Key="dataHasBeenEntered" />
</ResourceDictionary>
```

XAML 如下所示。请注意以下差异，从第一个多触发器示例：

- 该按钮具有 `IsEnabled="false"` 默认设置。
- 多触发器条件使用转换器来启用 `Text.Length` 值到 `boolean` 。
- 当所有条件都都 `true` ，setter 使按钮的 `IsEnabled` 属性 `true` 。

```
<Entry x:Name="user" Text="" Placeholder="user name" />

<Entry x:Name="pwd" Text="" Placeholder="password" />

<Button x:Name="loginButton" Text="Login"
        FontSize="Large"
        HorizontalOptions="Center"
        IsEnabled="false">
  <Button.Triggers>
    <MultiTrigger TargetType="Button">
      <MultiTrigger.Conditions>
        <BindingCondition Binding="{Binding Source={x:Reference user},
                                   Path=Text.Length,
                                   Converter={StaticResource dataHasBeenEntered}}"
                          Value="true" />
        <BindingCondition Binding="{Binding Source={x:Reference pwd},
                                   Path=Text.Length,
                                   Converter={StaticResource dataHasBeenEntered}}"
                          Value="true" />
      </MultiTrigger.Conditions>
      <Setter Property="IsEnabled" Value="True" />
    </MultiTrigger>
  </Button.Triggers>
</Button>
```

这些屏幕截图显示了上述的两个多触发器示例之间的差异。屏幕的顶部，一个中输入文本 `Entry` 足以保存按钮。在屏幕的底部登录名按钮始终处于非活动状态，直到这两个字段包含数据。



EnterActions 和 ExitActions

若要实现更改触发器发生时的另一种方法是通过添加 `EnterActions` 并 `ExitActions` 集合，并指定 `TriggerAction<T>` 实现。

可以提供*同时* `EnterActions` 并 `ExitActions`，以及 `Setter` 中触发器，但请注意， `Setter` s 将立即调用 (它们不会等待 `EnterAction` 或 `ExitAction` 到完成)。或者，可以在代码中执行的所有内容，且不将 `Setter` 根本的 s。

```
<Entry Placeholder="enter job title">
    <Entry.Triggers>
        <Trigger TargetType="Entry"
                 Property="Entry.IsFocused" Value="True">
            <Trigger.EnterActions>
                <local:FadeTriggerAction StartsFrom="0"" />
            </Trigger.EnterActions>

            <Trigger.ExitActions>
                <local:FadeTriggerAction StartsFrom="1" />
            </Trigger.ExitActions>
                        <!-- You can use both Enter/Exit and Setter together if required -->
        </Trigger>
    </Entry.Triggers>
</Entry>
```

如往常一样，一个类引用 XAML 中应如声明一个命名空间 `xmlns:local` 如下所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:WorkingWithTriggers;assembly=WorkingWithTriggers"
```

`FadeTriggerAction` 代码如下所示：

```
public class FadeTriggerAction : TriggerAction<VisualElement>
{
    public FadeTriggerAction() {}

    public int StartsFrom { set; get; }

    protected override void Invoke (VisualElement visual)
    {
            visual.Animate("", new Animation( (d)=>{
                var val = StartsFrom==1 ? d : 1-d;
                visual.BackgroundColor = Color.FromRgb(1, val, 1);

            }),
            length:1000, // milliseconds
            easing: Easing.Linear);
    }
}
```

注意：`EnterActions` 并 `ExitActions` 上忽略事件触发器。

相关链接

- 触发器示例
- Xamarin.Forms API 文档

Xamarin.Forms 用户界面视图

2018/10/26 • • Edit Online

如何使用 Xamarin.Forms 提供的视图

动画

Xamarin.Forms 具有自己的动画基础结构直接用于创建简单动画，而且还多样化足以创建复杂动画。

BoxView

`BoxView` 只是一个简单有色的矩形，但装饰性项，基本图形，以及用于获取交互式触摸输入，可以使用它。

Button

`Button` 响应点击或单击，将定向的应用程序来执行特定任务。

颜色

每个平台都有其自己的标准和默认值时，定义和跨平台使用的颜色可能比较棘手。

控件引用

本文档是如组成 Xamarin.Forms 框架的 UI 视图的快速参考Pages，布局，视图和单元格.

DataPages

DataPages 提供一个 API，用于快速、轻松地将数据源绑定到预建的视图。列表项和详细信息页将自动呈现数据，并使用主题自定义。

DatePicker

`DatePicker` 使用户可以选择指定范围内的一个日期。实现使用日期选取器受特定平台上运行应用程序。

使用 SkiaSharp 图形

如何将图形合并到使用 SkiaSharp 的 Xamarin.Forms 应用程序。

图像

可以使用 Xamarin.Forms 跨平台共享映像、可以专门为每个平台，加载它们或它们可以为显示下载。

布局

Xamarin.Forms 具有用于组织屏幕内容的多个布局。`StackLayout``Grid`，`FlexLayout`，`AbsoluteLayout`，`ScrollView`，和`RelativeLayout`每个可用来创建美观、响应迅速的用户界面。

ListView

Xamarin.Forms 提供了显示的数据的滚动行的列表视图控件。控件包括上下文操作`HasUnevenRows`自动调整大小、

自定义分隔符、下拉刷新和页眉和页脚。

地图

添加映射需要额外的 NuGet 包下载和一些特定于平台的配置。完成配置后，可以在只需几行代码中添加地图和 pin 标记。

选取器

`Picker` 视图是一个用于选择文本项中的数据列表控件。

滑块

`Slider` 允许用户从连续范围选择的数字值。

分档器

`Stepper` 允许用户从一系列值中选择的数字值。它包含两个按钮带有负号和加号。以增量方式将操作的两个按钮更改所选的值。

样式

字体、颜色和其他属性可以分组为样式，它们可以在控件、布局或使用 ResourceDictionaries 的整个应用程序之间共享。

TableView

表视图是类似于列表视图中，但而不是正在设计的数据的长列表它适用于数据条目样式的滚动控件或简单滚动菜单的屏幕。

文本

Xamarin.Forms 有多个视图来显示和接收文本。可以进行格式化并针对平台自定义文本视图。特定字体设置可以启用与辅助功能的兼容性。

主题

Xamarin.Forms 主题定义特定的可视化外观，对标准控件。一旦应用程序的资源字典中添加了一个主题，将更改标准控件的外观。

TimePicker

`TimePicker` 允许用户选择的时间。它使用特定平台上运行应用程序的支持时间选取器实现。

可视状态管理器

视觉状态管理器提供的结构化的方法来触发从代码中，包括布局可适应更改设备方向或大小更改的用户界面中的更改。

WebView

Xamarin.Forms 使用每个平台上的本机 web 浏览器控件，并可以显示网站、本地资源和生成的 Html 字符串。

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms 库（示例）

在 Xamarin.Forms 中的动画

2018/7/13 • • Edit Online

Xamarin.Forms 具有自己的动画基础结构直接用于创建简单动画，而且还多样化足以创建复杂动画。

Xamarin.Forms 动画类针对一个典型的动画，渐进式属性从一个值更改到另一段时间内的可视元素的不同属性。请注意，Xamarin.Forms 动画类没有 XAML 界面。但是，在中封装动画行为，然后从 XAML 引用。

简单动画

`ViewExtensions` 类提供了可用于构造简单动画的旋转、缩放、转换，和淡出的扩展方法 `VisualElement` 实例。本文演示如何创建和取消使用动画 `ViewExtensions` 类。

缓动函数

包括 Xamarin.Forms `Easing` 类，可用于指定在传输函数，用于控制动画如何加快或减慢，因为它们正在运行。本文演示如何使用预定义的缓动函数，以及如何创建自定义缓动函数。

自定义动画

`Animation` 类是所有 Xamarin.Forms 动画中的扩展方法构建基块 `ViewExtensions` 类创建一个或多个 `Animation` 对象。本文演示如何使用 `Animation` 类来创建和取消动画、同步多个动画，并创建不通过现有的动画方法进行动画处理的属性进行动画处理的自定义动画。

在 Xamarin.Forms 中的简单动画

2018/10/19 • • Edit Online

ViewExtensions 类提供了可用于构造简单动画的扩展方法。本文演示如何创建和取消使用 ViewExtensions 类的动画。

`ViewExtensions` 类提供了以下可用于创建简单动画的扩展方法：

- `TranslateTo` 进行动画处理 `TranslationX` 并 `TranslationY` 属性的 `VisualElement` 。
- `ScaleTo` 进行动画处理 `Scale` 的属性 `VisualElement` 。
- `RelScaleTo` 经过动画处理的增量增加或减少到适用 `Scale` 的属性 `VisualElement` 。
- `RotateTo` 进行动画处理 `Rotation` 的属性 `VisualElement` 。
- `RelRotateTo` 经过动画处理的增量增加或减少到适用 `Rotation` 的属性 `VisualElement` 。
- `RotateXTo` 进行动画处理 `RotationX` 的属性 `VisualElement` 。
- `RotateYTo` 进行动画处理 `RotationY` 的属性 `VisualElement` 。
- `FadeTo` 进行动画处理 `Opacity` 的属性 `VisualElement` 。

默认情况下，每个动画将采用 250 毫秒。但是，创建动画时，可以指定每个动画的持续时间。

`ViewExtensions` 类还包括 `CancelAnimations` 可用于取消任何动画的方法。

> **NOTE**
>
> `ViewExtensions` 类提供 `LayoutTo` 扩展方法。但是，此方法旨在由布局进行动画处理的包含大小和定位更改布局状态之间转换。因此，它应仅由 `Layout` 子类。

中的动画扩展方法 `ViewExtensions` 类是所有异步、返回 `Task<bool>` 对象。返回值是 `false` 动画完成后，如果和 `true` 如果动画将被取消。因此，动画方法通常都应与使用 `await` 运算符，这使得可以轻松地确定何时完成动画。此外，然后就可以通过执行前一种方法完成后的后续动画方法创建顺序动画。有关详细信息，请参阅复合动画。

如果要求让动画在后台完成则 `await` 运算符，则可以省略。在此方案中，动画扩展方法将快速启动动画，与在后台进行动画后返回。此操作时可以采取的优点创建复合动画。有关详细信息，请参阅复合动画。

有关详细信息 `await` 运算符，请参阅异步支持概述。

单个动画

在每个扩展方法 `ViewExtensions` 实现渐进式属性从一个值更改为另一个值的时间段内的单个动画操作。本部分探讨每个动画操作。

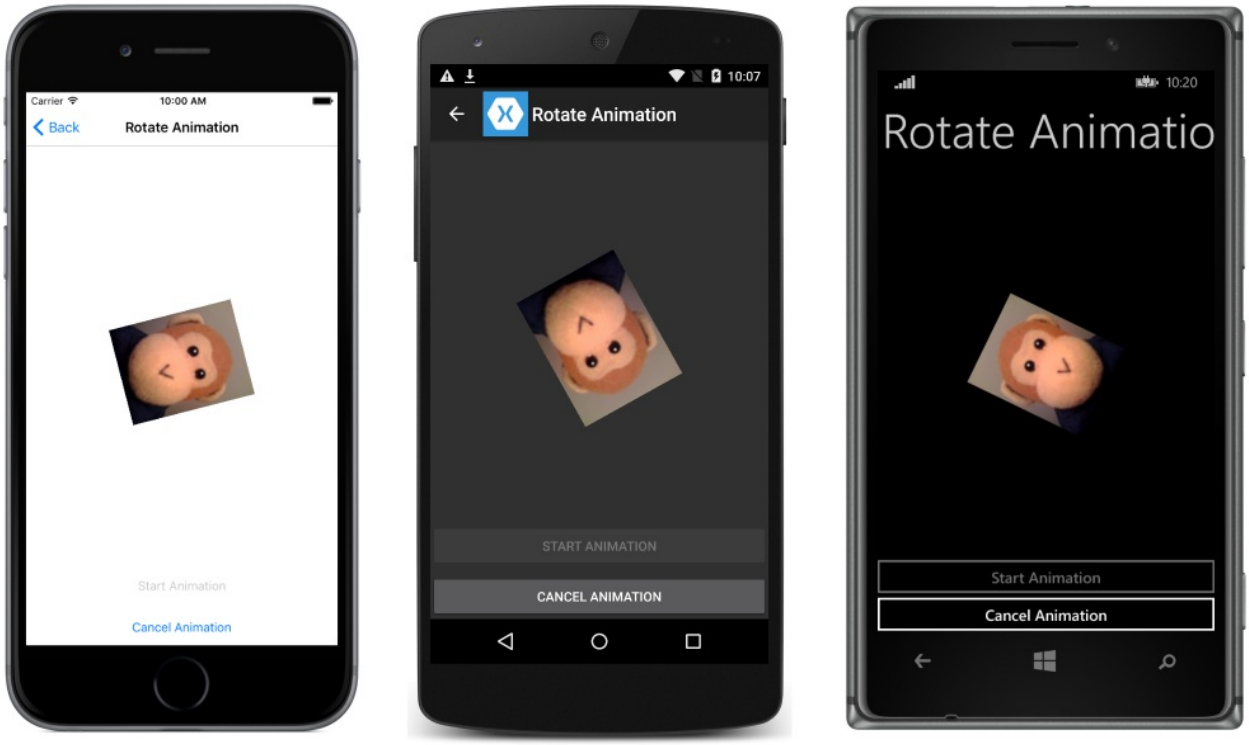
旋转

下面的代码示例演示了如何使用 `RotateTo` 方法进行动画处理 `Rotation` 属性的 `Image` ：

```
await image.RotateTo (360, 2000);
image.Rotation = 0;
```

此代码之间进行动画处理 `Image` 通过旋转超过 2 秒（2000年毫秒）的最多 360 度的实例。 `RotateTo` 方法获取当前 `Rotation` 属性值的动画，开头，然后从该值旋转到其第一个参数 (360)。一旦在动画完成，该映像 `Rotation` 属性重置为 0。这可确保 `Rotation` 动画结束，可能会妨碍其他旋转后，属性不会保留在 360。

下面的屏幕截图显示在每个平台上进行旋转：



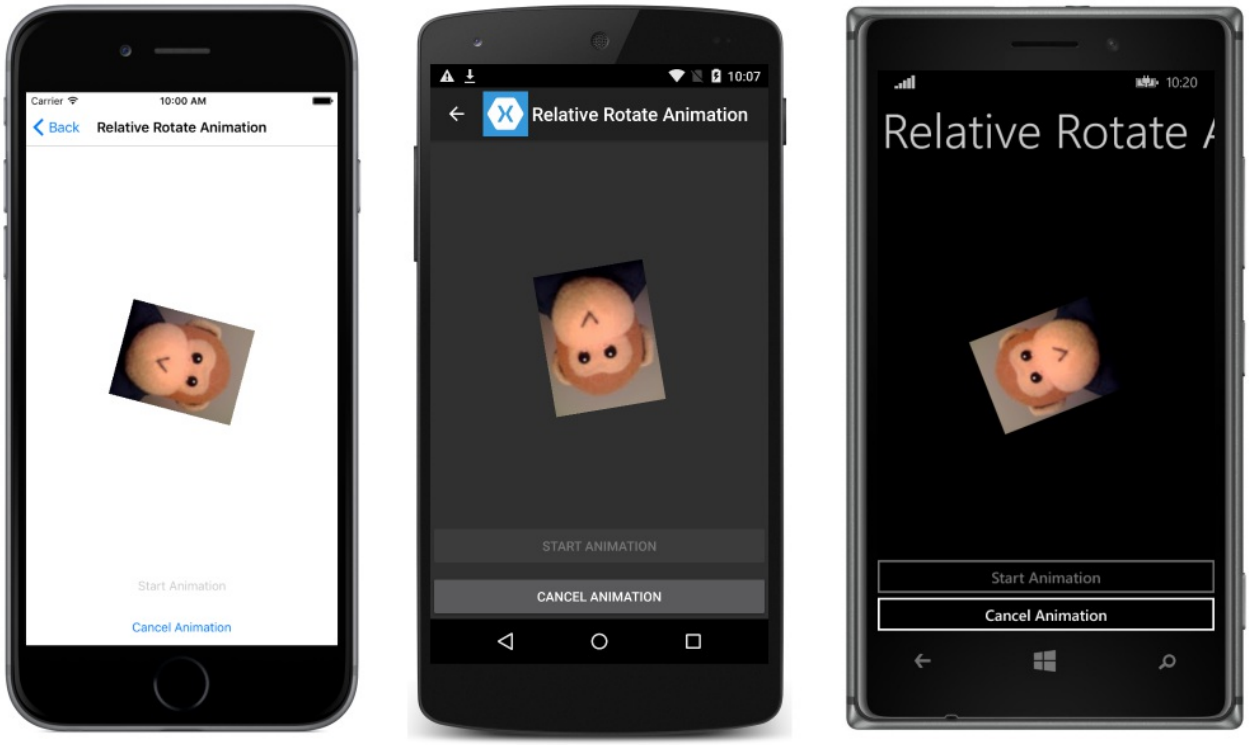
相对旋转

下面的代码示例演示了如何使用 `RelRotateTo` 方法来以增量方式增加或减少 `Rotation` 属性的 `Image`：

```
await image.RelRotateTo (360, 2000);
```

此代码之间进行动画处理 `Image` 通过旋转 360 度从其起始位置超过 2 秒（2000年毫秒）的实例。`RelRotateTo` 方法获取当前 `Rotation` 属性值的动画，开头，然后从该值旋转到值加上其第一个参数 (360)。这可确保每个动画将始终是从起始位置的 360 度旋转。因此，如果调用新动画的动画已正在进行时，它将开始从当前位置，并可能最终不是递增的 360 度的位置。

下面的屏幕截图显示每个平台上进行中的相对旋转：



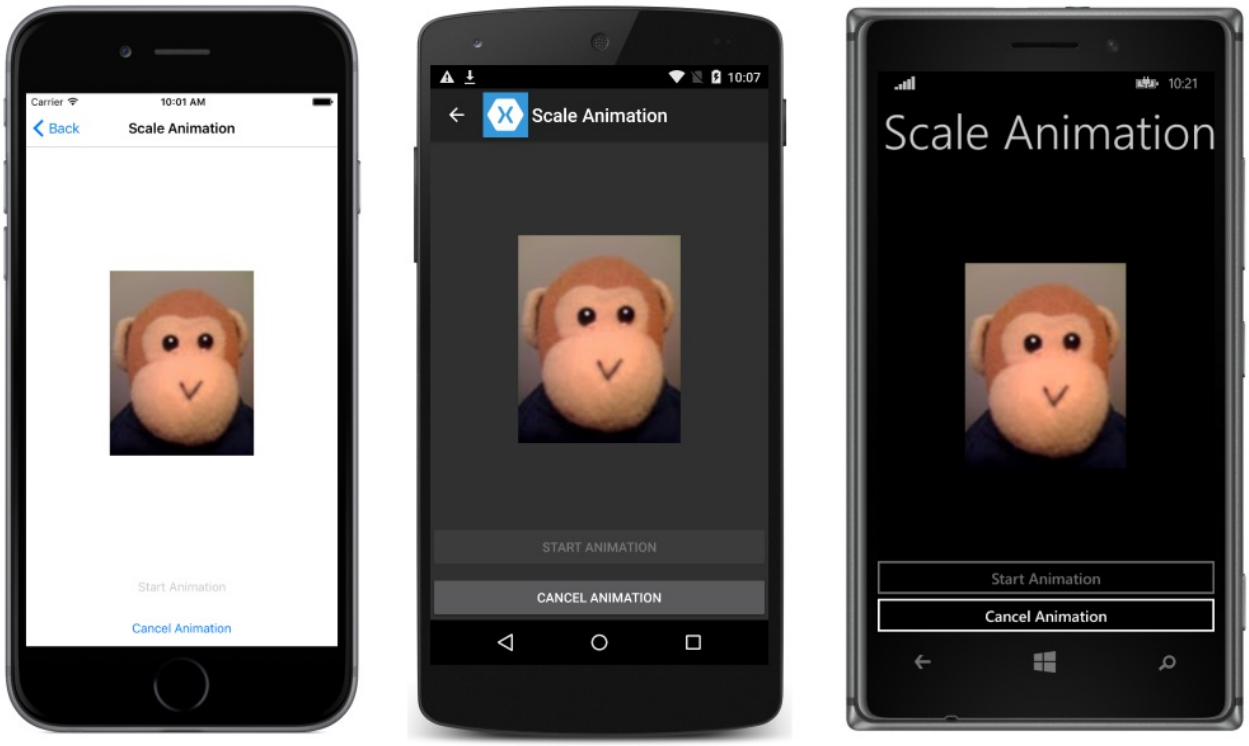
缩放

下面的代码示例演示了如何使用 `ScaleTo` 方法进行动画处理 `Scale` 属性的 `Image`：

```
await image.ScaleTo (2, 2000);
```

此代码之间进行动画处理 `Image` 通过纵向扩展到两次其大小超过 2 秒（2000年毫秒）的实例。`ScaleTo` 方法获取当前 `Scale` 开头的动画，然后可从该值扩展到其第一个参数 (2) 的属性值（默认值为 1）。此效果扩展到其大小的两倍的图像的大小。

以下屏幕截图显示在每个平台上进行缩放：



> **NOTE**
>
> `VisualElement` 类还定义 `ScaleX` 并 `ScaleY` 属性，可以缩放 `VisualElement` 中按不同方式水平和垂直方向。可以使用这些属性进行动画处理 `Animation` 类。有关详细信息，请参阅Xamarin.Forms 中的自定义动画。

相对缩放

下面的代码示例演示了如何使用 `RelScaleTo` 方法进行动画处理 `Scale` 属性的 `Image`：

```
await image.RelScaleTo (2, 2000);
```

此代码之间进行动画处理 `Image` 通过纵向扩展到两次其大小超过 2 秒（2000年毫秒）的实例。`RelScaleTo` 方法获取当前 `Scale` 开头的动画，然后可从该值扩展到其第一个参数 (2) 加上值的属性值。这可确保每个动画将始终是从起始位置 2 的伸缩性。

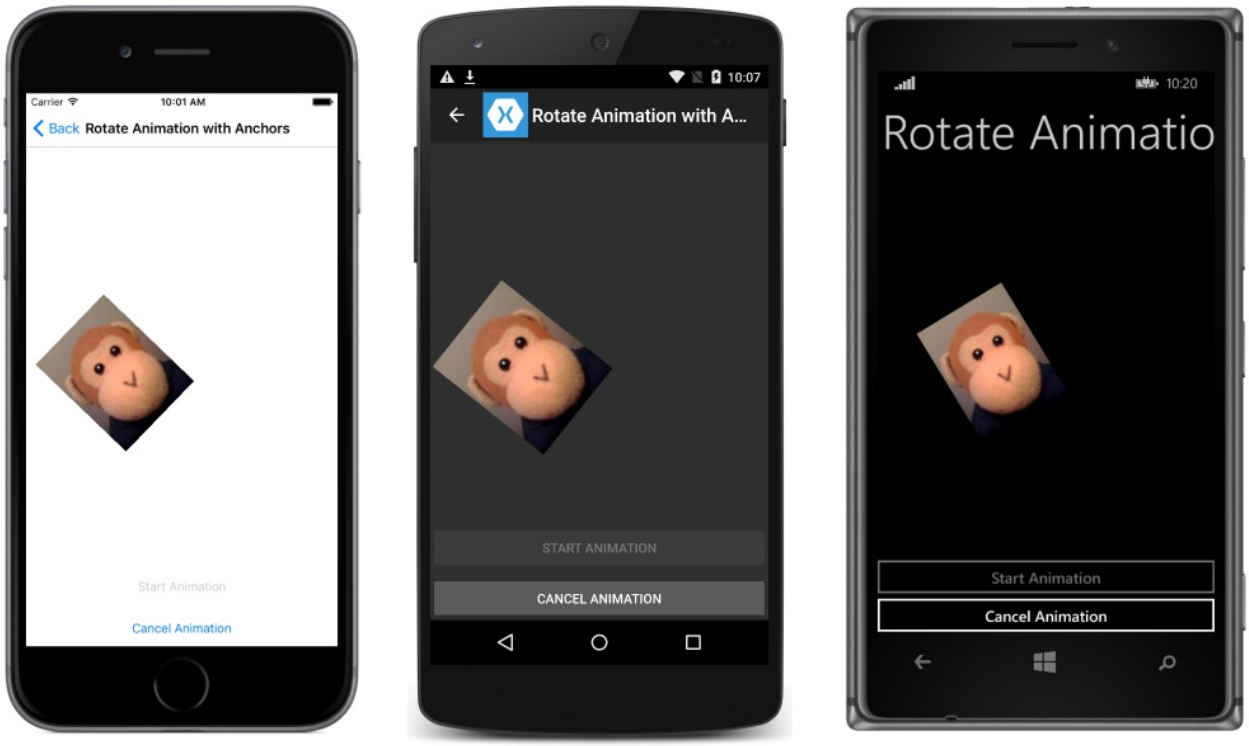
缩放和旋转与定位点

`AnchorX` 并 `AnchorY` 属性设置的缩放或旋转中心 `Rotation` 和 `Scale` 属性。因此，它们的值也会影响 `RotateTo` 并 `ScaleTo` 方法。

给定 `Image` 已被放在中心的一种布局，下面的代码示例演示了如何通过设置轮换图像的布局中心及其 `AnchorY` 属性：

```
image.AnchorY = (Math.Min (absoluteLayout.Width, absoluteLayout.Height) / 2) / image.Height;
await image.RotateTo (360, 2000);
```

若要旋转 `Image` 围绕的中心实例的格式，请 `AnchorX` 并 `AnchorY` 属性必须设置为值相对于宽度和高度 `Image`。在此示例中，中心的 `Image` 定义要在布局中，在中心，因此默认 `AnchorX` 0.5 的值不需要更改。但是，`AnchorY` 属性被重定义为顶部的值 `Image` 到布局的中心点。这可确保 `Image` 使布局的中心点围绕 360 度的旋转一圈，如以下屏幕截图中所示：



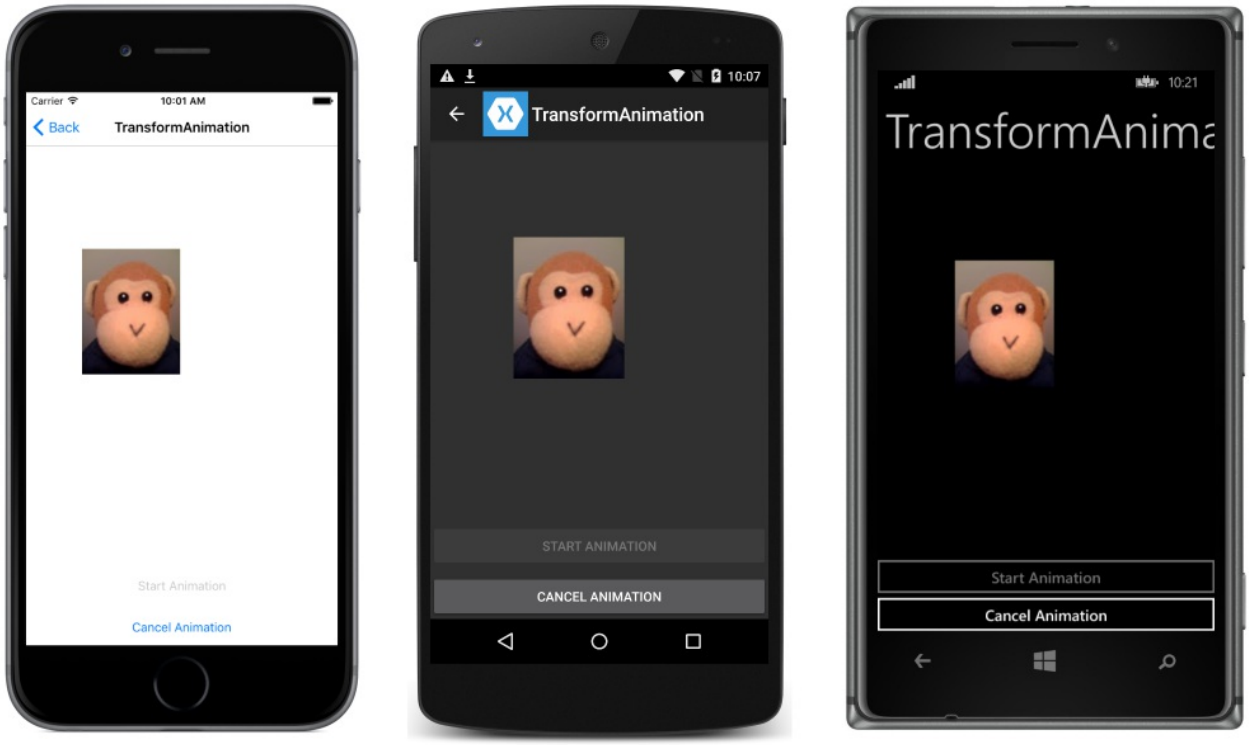
转换

下面的代码示例演示了如何使用 `TranslateTo` 方法进行动画处理 `TranslationX` 并 `TranslationY` 属性 `Image`：

```
await image.TranslateTo (-100, -100, 1000);
```

此代码之间进行动画处理 `Image` 实例将转换其水平和垂直超过 1 秒（1000年毫秒为单位）。 `TranslateTo` 映像 100 像素的左侧，并向上 100 像素，同时将转换方法。这是因为第一个和第二个参数均为这两个负数。提供正号将其转换到右和向下的图像。

下面的屏幕截图显示每个平台上进行中的翻译：



> **NOTE**
>
> 如果元素是最初离开屏幕的布局，则转换到屏幕上，在转换后的元素的输入的布局保持状态屏幕外并且用户不能与之交互。因此，建议，应在其最后一个位置中布局视图，然后所需的任何执行的转换。

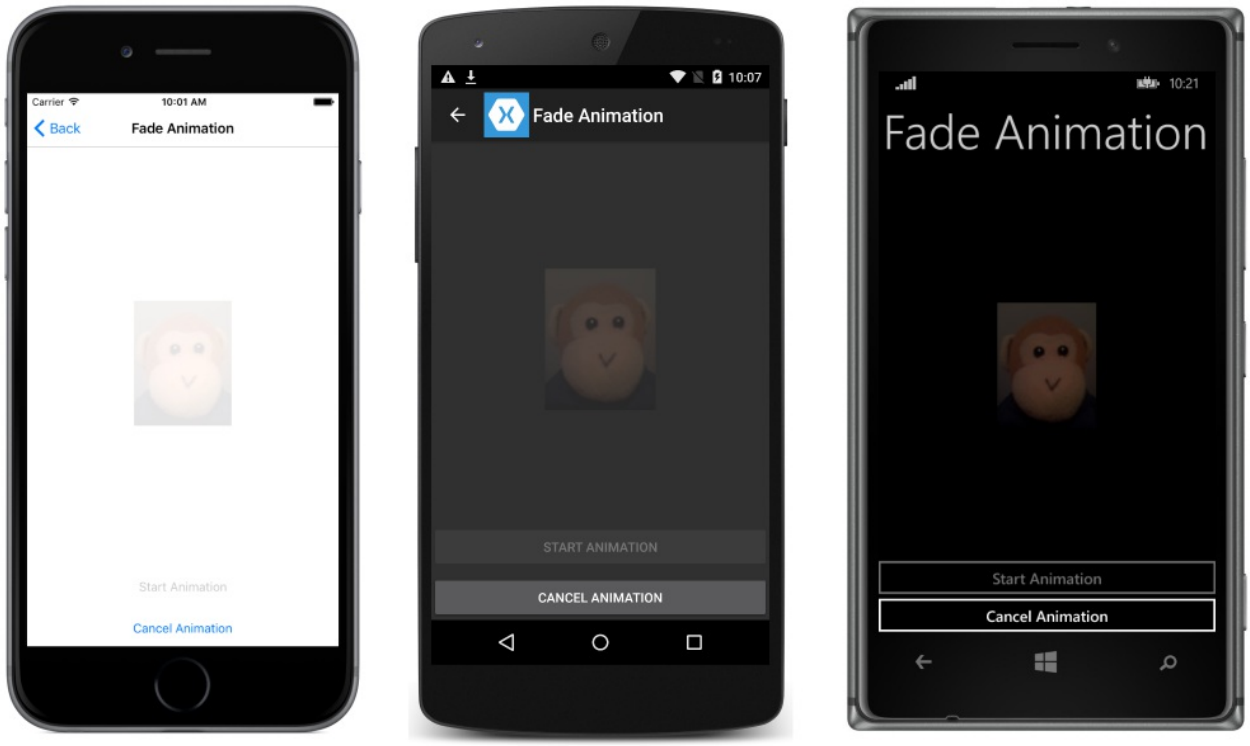
淡入淡出

下面的代码示例演示了如何使用 `FadeTo` 方法进行动画处理 `Opacity` 属性的 `Image` ：

```
image.Opacity = 0;
await image.FadeTo (1, 4000);
```

此代码之间进行动画处理 `Image` 通过淡入淡出它在超过 4 秒（4000 毫秒为单位）的实例。 `FadeTo` 方法获取当前 `Opacity` 开头的动画，然后淡中的从该值到其第一个参数 (1) 的属性值。

下面的屏幕截图显示每个平台上进行中的淡入淡出：



复合动画

复合动画是顺序组合的动画，并且可用于创建 `await` 运算符，如以下代码示例所示：

```
await image.TranslateTo (-100, 0, 1000);    // Move image left
await image.TranslateTo (-100, -100, 1000); // Move image up
await image.TranslateTo (100, 100, 2000);   // Move image diagonally down and right
await image.TranslateTo (0, 100, 1000);     // Move image left
await image.TranslateTo (0, 0, 1000);       // Move image up
```

在此示例中，`Image` 转换超过 6 秒（6000 毫秒为单位）。翻译 `Image` 使用五个动画 `await` 运算符，指示按顺序执行每个动画。因此，后续的动画方法来执行前一种方法完成后。

复合动画

复合动画是组合动画的两个或多个动画同时运行。可以通过混合使用等待和非等待动画创建复合动画，如下面的代码示例中所示：

```
image.RotateTo (360, 4000);
await image.ScaleTo (2, 2000);
await image.ScaleTo (1, 2000);
```

在此示例中，`Image` 已缩放和同时旋转超过 4 秒（4000 毫秒为单位）。缩放 `Image` 使用出现旋转时间的两个连续的动画。`RotateTo` 方法执行，但不 `await` 运算符并立即返回，第一个 `ScaleTo` 然后开始的动画。`await` 在第一个运算符 `ScaleTo` 方法调用会将第二个延迟 `ScaleTo` 直到第一个方法调用 `ScaleTo` 方法调用已完成。此时 `RotateTo` 动画是一半方式完成和 `Image` 将旋转 180 度。在最终的 2 秒（2000年毫秒），第二个 `ScaleTo` 动画和 `RotateTo` 同时完成动画。

同时运行多个异步方法

`static` `Task.WhenAny` 和 `Task.WhenAll` 方法用于同时运行多个异步方法，因此可以用于创建复合动画。这两种方法返回 `Task` 对象，并接受的方法的集合，每个返回 `Task` 对象。`Task.WhenAny` 方法完成时其集合中的任何方法完成执行，如下面的代码示例中所示：

```
await Task.WhenAny<bool>
(
  image.RotateTo (360, 4000),
  image.ScaleTo (2, 2000)
);
await image.ScaleTo (1, 2000);
```

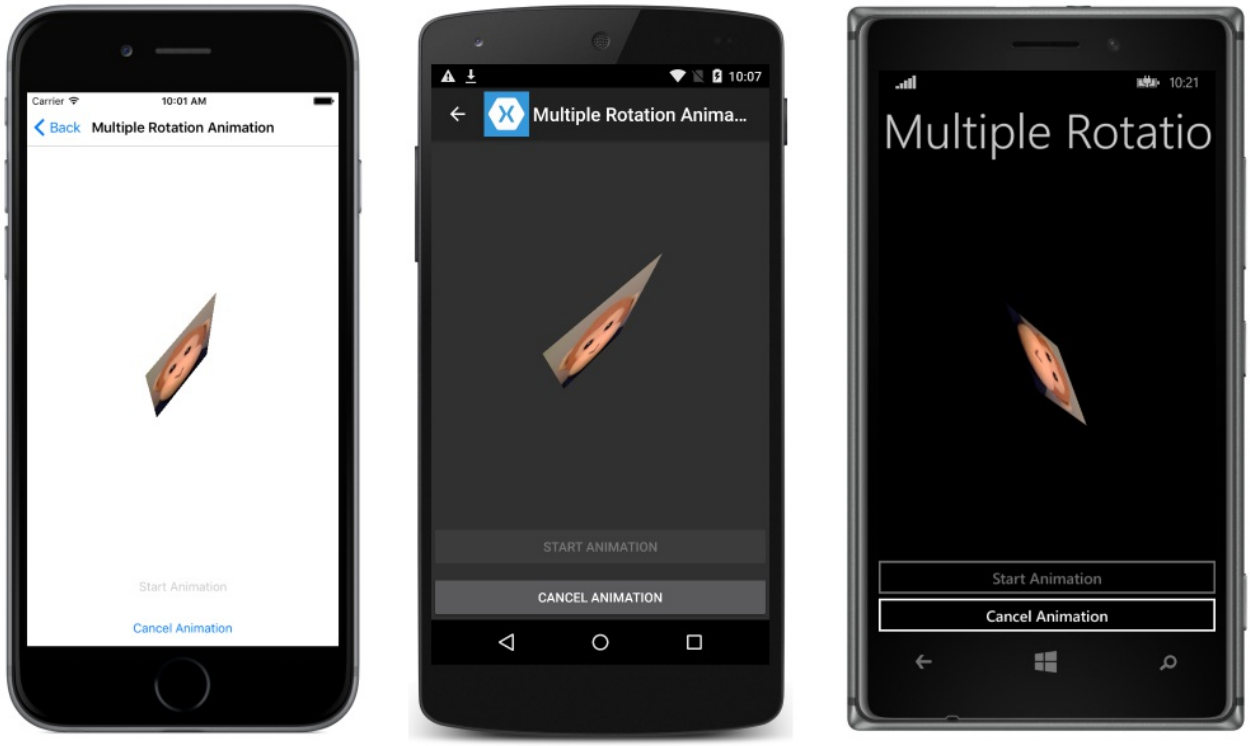
在此示例中，`Task.WhenAny`方法调用包含两个任务。第一个任务将图像旋转超过 4 秒（4000 毫秒为单位），且第二个任务可以缩放该图像超过 2 秒（2000年毫秒）。第二个任务完成后，`Task.WhenAny`方法调用完成。但是，即使`RotateTo`方法仍在运行，第二个`ScaleTo`方法可以开始。

`Task.WhenAll`方法完成时完成其集合中的所有方法，如下面的代码示例中所示：

```
// 10 minute animation
uint duration = 10 * 60 * 1000;

await Task.WhenAll (
  image.RotateTo (307 * 360, duration),
  image.RotateXTo (251 * 360, duration),
  image.RotateYTo (199 * 360, duration)
);
```

在此示例中，`Task.WhenAll`方法调用包含三个任务，其中每个执行时间超过 10 分钟。每个`Task`使得不同数量的 360 度旋转 – 为 307 旋转`RotateTo`，为 251 旋转`RotateXTo`，和 199 循环`RotateYTo`. 这些值是质数，因此确保旋转的未同步，并因此不会导致重复模式。

下面的屏幕截图显示每个平台上进行中的多个旋转：



正在取消动画

应用程序可以取消通过调用一个或多个动画`static``ViewExtensions.CancelAnimations`方法，如下面的代码示例中所示：

```
ViewExtensions.CancelAnimations (image);
```

这将立即取消当前正在运行的所有动画 `Image` 实例。

总结

本文说明了创建和取消使用动画 `ViewExtensions` 类。此类提供扩展方法可用于构造简单动画的旋转、缩放、转换，和淡化 `VisualElement` 实例。

相关链接

- 异步支持概述
- 基本动画（示例）
- ViewExtensions

在 Xamarin.Forms 中的缓动函数

2018/7/13 • • Edit Online

Xamarin.Forms 包含允许您指定的传输函数，用于控制动画如何加快或减慢，因为它们正在运行的缓类。本文演示如何使用预定义的缓动函数，以及如何创建自定义缓动函数。

`Easing` 类定义了大量可供动画的缓动函数：

- `BounceIn` 缓动函数弹跳动画的开头。
- `BounceOut` 缓动函数弹跳动画结束时。
- `CubicIn` 缓动函数缓慢加速动画。
- `CubicInOut` 缓动函数在开始时，动画加速和减速的动画结束时。
- `CubicOut` 快速缓动函数减速的动画。
- `Linear` 缓动函数使用常量的速度，并且默认的缓动函数。
- `SinIn` 顺利缓动函数加速动画。
- `SinInOut` 顺利缓动函数加快在开始时，动画并平稳减速的动画结束时。
- `SinOut` 顺利缓动函数减速的动画。
- `SpringIn` 缓动函数使动画结束时非常快速地加快速度。
- `SpringOut` 缓动函数使动画结束时快速减速。

`In` 和 `Out` 后缀指示缓动函数提供的效果是明显的动画，和 / 或结束时，在开头。

此外，可以创建自定义缓动函数。有关详细信息，请参阅自定义缓动函数。

使用缓动函数

中的动画扩展方法 `ViewExtensions` 类允许指定为最后一种方法参数中，缓动函数，如下面的代码示例中所示：

```
await image.TranslateTo(0, 200, 2000, Easing.BounceIn);
await image.ScaleTo(2, 2000, Easing.CubicIn);
await image.RotateTo(360, 2000, Easing.SinInOut);
await image.ScaleTo(1, 2000, Easing.CubicOut);
await image.TranslateTo(0, -200, 2000, Easing.BounceOut);
```

通过指定动画的缓动函数，动画速度变得的非线性，并生成缓动函数提供的效果。创建动画时省略缓动函数会导致使用默认值的动画 `Linear` 缓动函数，它将生成线性速度。

有关使用中的动画扩展方法的详细信息 `ViewExtensions` 类，请参阅简单动画。缓动函数也可以使用通过 `Animation` 类。有关详细信息，请参阅自定义动画。

自定义缓动函数

有三种主要方法与创建自定义缓动函数：

1. 创建一个方法采用 `double` 自变量，并返回 `double` 结果。
2. 创建 `Func<double, double>`。
3. 指定为参数的缓动函数 `Easing` 构造函数。

在所有三种情况下，自定义缓动函数应返回 0 和 1 的 1 的自变量作为参数为 0。但是，任何值可以返回的参数值为 0 和 1 之间。现在将依次介绍每种方法。

自定义缓动方法

可以将自定义缓动函数定义为采用的方法 `double` 自变量，并返回 `double` 导致，如下面的代码示例中所示：

```
await image.TranslateTo(0, 200, 2000, CustomEase);

double CustomEase (double t)
{
  return t == 0 || t == 1 ? t : (int)(5 * t) / 5.0;
}
```

`CustomEase` 方法将传入值为 0、0.2、0.4，0.6，0.8 和 1 的值截断。因此， `Image` 实例在离散的跳转，而不是平稳转换。

自定义缓动函数

自定义缓动函数也可以定义为 `Func<double, double>`，如以下代码示例所示：

```
Func<double, double> CustomEase = t => 9 * t * t * t - 13.5 * t * t + 5.5 * t;
await image.TranslateTo(0, 200, 2000, CustomEase));
```

`CustomEase` `Func` 表示缓动函数，以快速、速度变慢和反转课程，，然后反转再次课程结束时快速加速。因此，虽然的总体移动 `Image` 实例向下，它也会暂时反转课程中途动画。

自定义缓动构造函数

此外可以为的参数定义的自定义缓动函数 `Easing` 构造函数，如下面的代码示例中所示：

```
await image.TranslateTo (0, 200, 2000, new Easing (t => 1 - Math.Cos (10 * Math.PI * t) * Math.Exp (-5 * t)));
```

自定义缓动函数指定为 lambda 函数参数 `Easing` 构造函数，并使用 `Math.Cos` 方法创建的阻碍缓慢放置效果 `Math.Exp` 方法。因此， `Image` 实例平移，以便它看起来放到其最终静止原位置。

总结

本文演示了如何使用预定义的缓动函数，以及如何创建自定义缓动函数。包括 Xamarin.Forms `Easing` 类，可用于指定在传输函数，用于控制动画如何加快或减慢，因为它们正在运行。

相关链接

- 异步支持概述
- 缓动函数（示例）
- 缓动
- ViewExtensions

在 Xamarin.Forms 中的自定义动画

2018/7/13 • • Edit Online

动画类是所有 Xamarin.Forms 动画，创建一个或多个动画对象 ViewExtensions 类中的扩展方法使用构建基块。本文演示如何使用动画类来创建和取消动画，同步多个动画，并创建不通过现有的动画方法进行动画处理的属性进行动画处理的自定义动画。

在创建时，必须指定多个参数 `Animation` 对象，包括开始和结束值的属性进行动画处理，并更改属性的值的回调。 `Animation` 对象还可以维护一系列子动画，可以运行并同步。有关详细信息，请参阅子动画。

运行与创建的动画 `Animation` 类，它可能包含或不包含子动画，实现通过调用 `Commit` 方法。此方法指定的持续时间的动画，并在其他项目之间的回调，用于控制是否重复动画。

创建动画

创建时 `Animation` 对象，通常情况下，最少三个参数是必需的如下面的代码示例中所示：

```
var animation = new Animation (v => image.Scale = v, 1, 2);
```

此代码定义的动画 `Scale` 的属性 `Image` 实例从一个值为 1 到 2 的值。通过 Xamarin.Forms 派生的动画的值传递给回调指定为第一个参数，它用于更改的值 `Scale` 属性。

通过调用启动动画 `Commit` 方法，如下面的代码示例中所示：

```
animation.Commit (this, "SimpleAnimation", 16, 2000, Easing.Linear, (v, c) => image.Scale = 1, () => true);
```

请注意， `Commit` 方法不返回 `Task` 对象。相反，它们通过回调方法提供通知。

中指定以下参数 `Commit` 方法：

- 第一个参数 (*所有者*) 标识的所有者动画。这可以是应用动画的可视元素或另一个可视元素，如页。
- 第二个参数 (*名称*) 标识的名称的动画。名称结合了多个要唯一标识该动画的所有者。然后可以使用此唯一标识来确定是否正在运行的动画 (`AnimationIsRunning`)，或取消它 (`AbortAnimation`)。
- 第三个参数 (*速率*) 指示每次调用中定义的回调方法之间的毫秒数 `Animation` 构造函数
- 第四个参数 (*长度*) 指示动画，以毫秒为单位的持续时间。
- 第五个参数 (*缓动*) 定义了用于动画的缓动函数。或者，可以将缓动函数指定的参数为 `Animation` 构造函数。缓动函数的详细信息，请参阅缓动函数。
- 第六个参数 (*完成*) 将在动画完成时执行的回调。此回调采用两个参数，与第一个参数，该值指示最终值和第二个参数所 `bool` 设置为 `true` 已取消的动画。或者，*完成*回调可以指定为参数 `Animation` 构造函数。具有单个动画，但是，如果*完成*中均指定了回调 `Animation` 构造函数和 `Commit` 方法，仅在指定的回调 `Commit` 将执行方法。
- 第七个参数 (*重复*) 允许重复的动画的回调。动画结束时调用并返回 `true` 指示应该重复动画。

总体效果是要创建的动画，可提高 `Scale` 的属性 `Image` 从 1 到 2，超过 2 秒 (2000年毫秒)，使用 `Linear` 缓动函数。每次在动画完成后，其 `Scale` 属性重置为 1，并在动画重复。

> **NOTE**
>
> 可以通过创建构造均可独立运行的并发的动画 `Animation` 为每个动画对象，然后再调用 `Commit` 上每个动画的方法。

子动画

`Animation` 类还支持子动画，其中涉及到创建 `Animation` 对象的其他 `Animation` 对象将被添加。这样，动画将运行和同步的一系列。下面的代码示例演示如何创建和运行子动画：

```
var parentAnimation = new Animation ();
var scaleUpAnimation = new Animation (v => image.Scale = v, 1, 2, Easing.SpringIn);
var rotateAnimation = new Animation (v => image.Rotation = v, 0, 360);
var scaleDownAnimation = new Animation (v => image.Scale = v, 2, 1, Easing.SpringOut);

parentAnimation.Add (0, 0.5, scaleUpAnimation);
parentAnimation.Add (0, 1, rotateAnimation);
parentAnimation.Add (0.5, 1, scaleDownAnimation);

parentAnimation.Commit (this, "ChildAnimations", 16, 4000, null, (v, c) => SetIsEnabledButtonState (true,
false));
```

或者，可以更加简洁，如下面的代码示例所示编写代码示例：

```
new Animation {
    { 0, 0.5, new Animation (v => image.Scale = v, 1, 2) },
    { 0, 1, new Animation (v => image.Rotation = v, 0, 360) },
    { 0.5, 1, new Animation (v => image.Scale = v, 2, 1) }
    }.Commit (this, "ChildAnimations", 16, 4000, null, (v, c) => SetIsEnabledButtonState (true, false));
```

在这两个代码示例中，父级 `Animation` 创建对象后，向其附加 `Animation` 然后添加对象。前两个参数 `Add` 方法指定何时开始和完成子动画。参数值必须是 0 和 1 之间，并且表示在父动画指定的子动画将处于活动状态的相对期限。因此，在此示例 `scaleUpAnimation` 将处于活动状态的第一个动画，另一半 `scaleDownAnimation` 将处于活动状态的动画，虚拟机和 `rotateAnimation` 的整个持续时间内将处于活动状态。

总体效果是动画发生超过 4 秒（4000 毫秒为单位）。`scaleUpAnimation` 之间进行动画处理 `Scale` 属性从 1 到 2，超过 2 秒。`scaleDownAnimation` 然后进行动画处理 `Scale` 属性从 2 为 1，超过 2 秒。出现这两个缩放动画，而 `rotateAnimation` 之间进行动画处理 `Rotation` 属性从 0 到 360 之间，4 秒。请注意，缩放动画还使用缓动函数。`SpringIn` 缓动函数会导致 `Image` 获取较大前，最初收缩和 `SpringOut` 缓动函数导致 `Image` 以变得比其末尾的完整的动画的实际大小更小。

有多种之间的差异 `Animation` 用子动画对象，不会的一个：

- 使用子动画时*完成*子动画回调指示当子已完成，并且*完成*回调传递给 `Commit` 方法指示何时已完成整个动画。
- 当使用子动画，则返回 `true` 从*重复*上的回拨 `Commit` 方法不会导致动画重复，但该动画将继续运行而无需新值。
- 当包括中的缓动函数 `Commit` 方法，并且缓动函数则返回一个值大于 1，则动画将被终止。如果缓动函数返回的值小于 0，其值限制为 0。若要使用缓动函数，返回一个值小于 0 或大于 1，则必须指定一个子动画，而不是在中 `Commit` 方法。

`Animation` 类还包括 `WithConcurrent` 方法可用于将子动画添加到父 `Animation` 对象。但是，其*开始*并*完成*参数值都不限制为 0 到 1 之间，但只有这一部分对应于 0 到 1 范围子动画将处于活动状态。例如，如果 `WithConcurrent` 方法调用定义面向子动画 `Scale` 属性从 1 到 6，但*开始*并*完成*的值-2 和 3，*开始*-2 的值对应于 `Scale` 值为 1，并且*完成*3 的值对应于 `Scale` 值为 6。因为 0 和 1 的范围之外的值中不起作用动画，`Scale` 属性仅进行动画处理 3 到 6。

正在取消动画

应用程序可以取消通过调用动画 `AbortAnimation` 扩展方法，如下面的代码示例中所示：

```
this.AbortAnimation ("SimpleAnimation");
```

请注意动画进行动画所有者和动画名称的组合唯一标识。因此，所有者和名称指定何时运行动画必须指定要取消动画。因此，代码示例将立即取消名为动画 `SimpleAnimation` 归页。

创建自定义动画

到目前为止如下所示的示例已演示了同样的方法与可达到的动画 `ViewExtensions` 类。但是，利用 `Animation` 类是它有权访问经过动画处理的值更改时执行的回调方法。这样，要实现所需的任何动画的回调。例如，下面的代码示例进行动画处理 `BackgroundColor` 属性设置为页面 `Color` 创建的值 `Color.FromHsla` 方法中，范围从 0 到 1 的色调值：

```
new Animation (callback: v => BackgroundColor = Color.FromHsla (v, 1, 0.5),
  start: 0,
  end: 1).Commit (this, "Animation", 16, 4000, Easing.Linear, (v, c) => BackgroundColor = Color.Default);
```

生成的动画提供了提前出喷薄彩虹的颜色通过页面背景的外观。

创建复杂动画，其中包括贝塞尔曲线动画的更多示例请参阅第 22 章的使用 Xamarin.Forms 创建移动应用。

创建一个自定义动画的扩展方法

中的扩展方法 `ViewExtensions` 类对从其当前值为指定的值的属性进行动画处理。这使得难以创建，例如，`ColorTo` 可用于进行动画处理的一种颜色从一个值，因为动画方法：

- 唯一 `Color` 定义的属性 `VisualElement` 类是 `BackgroundColor`，但并非总是所需 `Color` 属性若要进行动画处理。
- 当前值通常 `Color` 属性是 `Color.Default`，这并不真正的颜色，并无法在计算内插中使用它。

此问题的解决方案是不让 `ColorTo` 方法针对特定 `Color` 属性。相反，它可以编写通过内插的回调方法 `Color` 回调用方的值。此外，该方法将执行开始和结束 `Color` 参数。

`ColorTo` 方法可作为使用的扩展方法 `Animate` 中的方法 `AnimationExtensions` 类以提供其功能。这是因为 `Animate` 方法可用于目标属性的类型不是 `double`，如以下代码示例所示：

```
public static class ViewExtensions
{
  public static Task<bool> ColorTo(this VisualElement self, Color fromColor, Color toColor, Action<Color>
callback, uint length = 250, Easing easing = null)
  {
    Func<double, Color> transform = (t) =>
      Color.FromRgba(fromColor.R + t * (toColor.R - fromColor.R),
                     fromColor.G + t * (toColor.G - fromColor.G),
                     fromColor.B + t * (toColor.B - fromColor.B),
                     fromColor.A + t * (toColor.A - fromColor.A));
    return ColorAnimation(self, "ColorTo", transform, callback, length, easing);
  }

  public static void CancelAnimation(this VisualElement self)
  {
    self.AbortAnimation("ColorTo");
  }

  static Task<bool> ColorAnimation(VisualElement element, string name, Func<double, Color> transform,
Action<Color> callback, uint length, Easing easing)
  {
    easing = easing ?? Easing.Linear;
    var taskCompletionSource = new TaskCompletionSource<bool>();

    element.Animate<Color>(name, transform, callback, 16, length, easing, (v, c) =>
taskCompletionSource.SetResult(c));
    return taskCompletionSource.Task;
  }
}
```

`Animate` 方法需要*转换*参数，这是一个回调方法。此回调的输入始终是 `double` 范围从 0 到 1。因此，`ColorTo` 方法定义自己转换 `Func` 接受 `double` 范围从 0 到 1，且该返回 `Color` 对应的值相对应的值。`Color` 值计算在插值 `R`、`G`、`B`、和 `A` 提供两个值 `Color` 参数。`Color` 值然后传递给应用程序特定属性的回调方法。

此方法允许 `ColorTo` 方法进行动画处理任何 `Color` 属性，如下面的代码示例中所示：

```
await Task.WhenAll(
  label.ColorTo(Color.Red, Color.Blue, c => label.TextColor = c, 5000),
  label.ColorTo(Color.Blue, Color.Red, c => label.BackgroundColor = c, 5000));
await this.ColorTo(Color.FromRgb(0, 0, 0), Color.FromRgb(255, 255, 255), c => BackgroundColor = c, 5000);
await boxView.ColorTo(Color.Blue, Color.Red, c => boxView.Color = c, 4000);
```

在此代码示例中，`ColorTo` 方法进行动画处理 `TextColor` 并 `BackgroundColor` 的属性 `Label`，`BackgroundColor` 属性页上，并 `Color` 属性 `BoxView`。

总结

本文演示了如何使用 `Animation` 类来创建和取消动画、同步多个动画，并创建不由现有动画进行动画处理的属性进行动画处理的自定义动画方法。`Animation` 类是所有 Xamarin.Forms 动画构建基块。

相关链接

- 自定义动画（示例）
- 动画
- AnimationExtensions

Xamarin.Forms 字数

2018/10/25 • • Edit Online

`BoxView` 呈现指定的宽度、高度和颜色的一个简单的矩形。可以使用 `BoxView` 修饰，基本图形，以及与用户通过触摸交互。

因为 Xamarin.Forms 不具有内置的向量图形系统 `BoxView` 可帮助进行补偿。此文章使用中所述的示例程序的一些 `BoxView` 呈现图形。 `BoxView` 可以调整大小以类似于特定的宽度和粗细的行，然后通过任何角度旋转 `Rotation` 属性。

尽管 `BoxView` 可以模仿简单的图形，你可能想要调查Xamarin.Forms 中使用 SkiaSharp更复杂的图形要求。

本文讨论以下主题：

- **设置字数颜色和大小** –设置 `BoxView` 属性。
- **呈现的文本修饰** –使用 `BoxView` 呈现线条。
- **列出的颜色的字数** –显示所有系统中的颜色 `ListView` 。
- **播放游戏的生命周期由子类化字数** –实现著名的移动电话自动机。
- **创建数字时钟** –模拟点矩阵显示。
- **创建模拟时钟** –转换并进行动画处理 `BoxView` 元素。

设置字数颜色和大小

通常将设置以下属性的 `BoxView` ：

- `Color` 若要设置其颜色。
- `CornerRadius` 若要设置其圆角半径。
- `WidthRequest` 若要设置的宽度 `BoxView` 以与设备无关单位。
- `HeightRequest` 若要设置的高度 `BoxView` 。

`Color` 属性属于类型 `Color` ；的属性可以设置为任何 `Color` 值，其中包括 141 静态只读字段的已命名的颜色范围为按字母顺序 `AliceBlue` 到 `YellowGreen` 。

`CornerRadius` 属性属于类型 `CornerRadius` ；的属性可以设置为单个 `double` 统一角半径值，或 `CornerRadius` 结构定义的四个 `double` 应用到的值左上、右上、左下角和右下角 `BoxView` 。

`WidthRequest` 并 `HeightRequest` 属性只能播放一个角色，如果 `BoxView` 是*不受约束*布局中。这种情况时的布局容器需要知道子的大小，例如，当 `BoxView` 中的自动调整大小单元格的子 `Grid` 布局。一个 `BoxView` 也是不受约束时其 `HorizontalOptions` 并 `VerticalOptions` 属性设置为值而不 `LayoutOptions.Fill` 。如果 `BoxView` 是不受约束，但 `WidthRequest` 和 `HeightRequest` 属性未设置，然后宽度或高度设置为 40 单位，或大约 1/4 英寸的移动设备上的默认值。

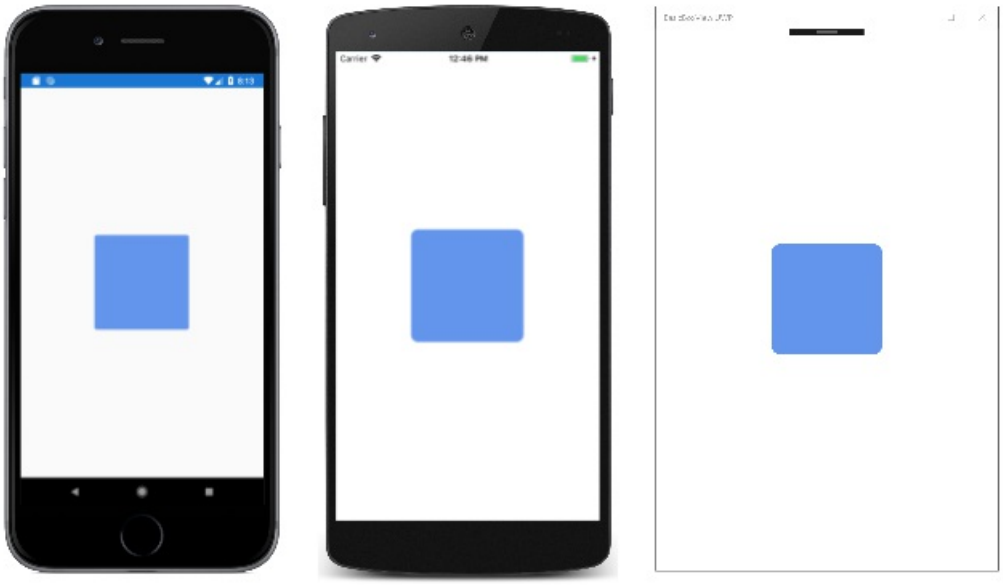
`WidthRequest` 并 `HeightRequest` 属性将被忽略，如果 `BoxView` 是*约束*在布局中，用例的布局容器对其自身的大小施加 `BoxView` 。

一个 `BoxView` 可以被限制在一个维度和其他不受约束。例如，如果 `BoxView` 是垂直的子级 `StackLayout` ，垂直维度的 `BoxView` 是不受约束，且其水平维度通常被约束。但有该水平维度的例外情况：如果 `BoxView` 具有其 `HorizontalOptions` 属性设置为内容而不 `LayoutOptions.Fill` ，也不受约束的水平维度则。也可能是 `StackLayout` 本身可以具有不受约束的水平维度，在这种情况下 `BoxView` 也将水平方向不受约束。

BasicBoxView 示例将显示一英寸的正方形不受约束 `BoxView` 在其页面的中心：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:BasicBoxView"
             x:Class="BasicBoxView.MainPage">

    <BoxView Color="CornflowerBlue"
             CornerRadius="10"
             WidthRequest="160"
             HeightRequest="160"
             VerticalOptions="Center"
             HorizontalOptions="Center" />

</ContentPage>
```

下面是结果：



如果 `VerticalOptions` 并 `HorizontalOptions` 属性都将从中 `BoxView` 标记，或者被设置为 `Fill`，则 `BoxView` 变得受页的大小和扩展以填充页。

一个 `BoxView` 也可以是子的 `AbsoluteLayout`。在此情况下，位置和大小 `BoxView` 使用设置 `LayoutBounds` 附加可绑定属性。 `AbsoluteLayout` 一文中讨论 **AbsoluteLayout**。

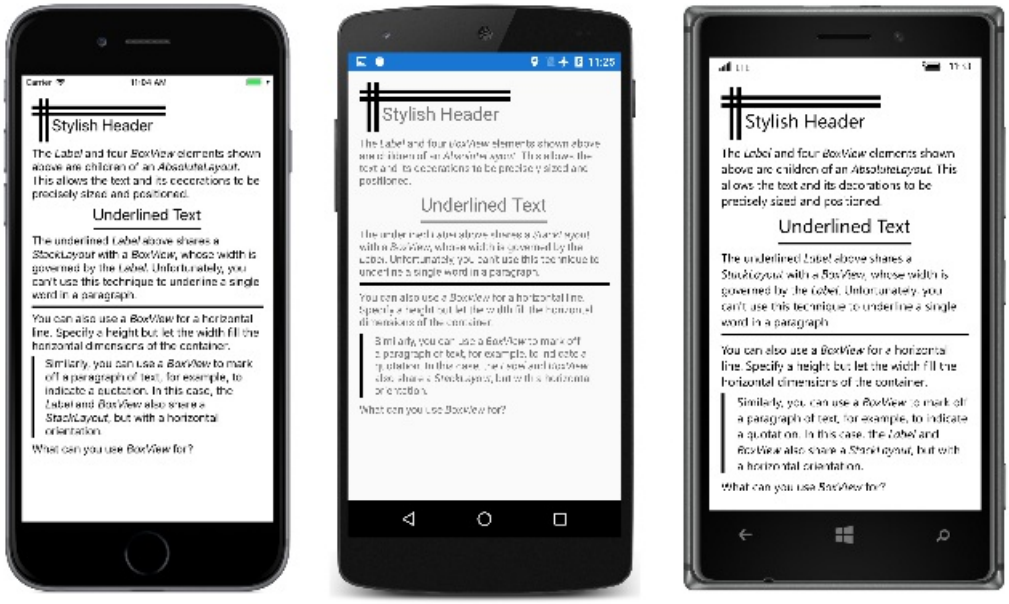
您将看到的示例程序，请按照中的所有这些情况下的示例。

呈现的文本修饰

可以使用 `BoxView` 形式的水平线和垂直线页面上添加一些简单装饰物。**TextDecoration** 示例演示了这。所有程序的视觉对象中定义**MainPage.xaml**文件，其中包含多个 `Label` 并 `BoxView` 中的元素 `StackLayout` 如下所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:TextDecoration"
             x:Class="TextDecoration.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="BoxView">
                <Setter Property="Color" Value="Black" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <ScrollView Margin="15">
        <StackLayout>

            ...

        </StackLayout>
    </ScrollView>
</ContentPage>
```

所有遵循的标记都是子级 `StackLayout` 。此标记由几种类型的装饰性 `BoxView` 元素用于 `Label` 元素：



在页面顶部的时尚标头，可以使用 `AbsoluteLayout` 其子级是四个 `BoxView` 元素和一个 `Label` ，则所有这些选项均进行分配的特定位置和大小：

```
<AbsoluteLayout>
    <BoxView AbsoluteLayout.LayoutBounds="0, 10, 200, 5" />
    <BoxView AbsoluteLayout.LayoutBounds="0, 20, 200, 5" />
    <BoxView AbsoluteLayout.LayoutBounds="10, 0, 5, 65" />
    <BoxView AbsoluteLayout.LayoutBounds="20, 0, 5, 65" />
    <Label Text="Stylish Header"
           FontSize="24"
           AbsoluteLayout.LayoutBounds="30, 25, AutoSize, AutoSize"/>
</AbsoluteLayout>
```

在 XAML 文件中， `AbsoluteLayout` 后跟 `Label` 使用的格式化文本描述 `AbsoluteLayout` 。

可以通过封闭同时添加下划线的文本字符串 `Label` 并 `BoxView` 中 `StackLayout` 具有其 `HorizontalOptions` 值设置为内容而不 `Fill` 。宽度 `StackLayout` 然后受到的宽度 `Label` ，后者随后实施该宽度 `BoxView` 。 `BoxView` 分配仅显式高度：

```
<StackLayout HorizontalOptions="Center">
    <Label Text="Underlined Text"
           FontSize="24" />
    <BoxView HeightRequest="2" />
</StackLayout>
```

不能使用此方法以用下划线标出较长文本字符串或一个段落中的各个单词。

它也是可以使用 `BoxView` 类似于 HTML `hr` （水平标尺）元素。只需让的宽度 `BoxView` 由其父容器，在这种情况下是 `StackLayout` ：

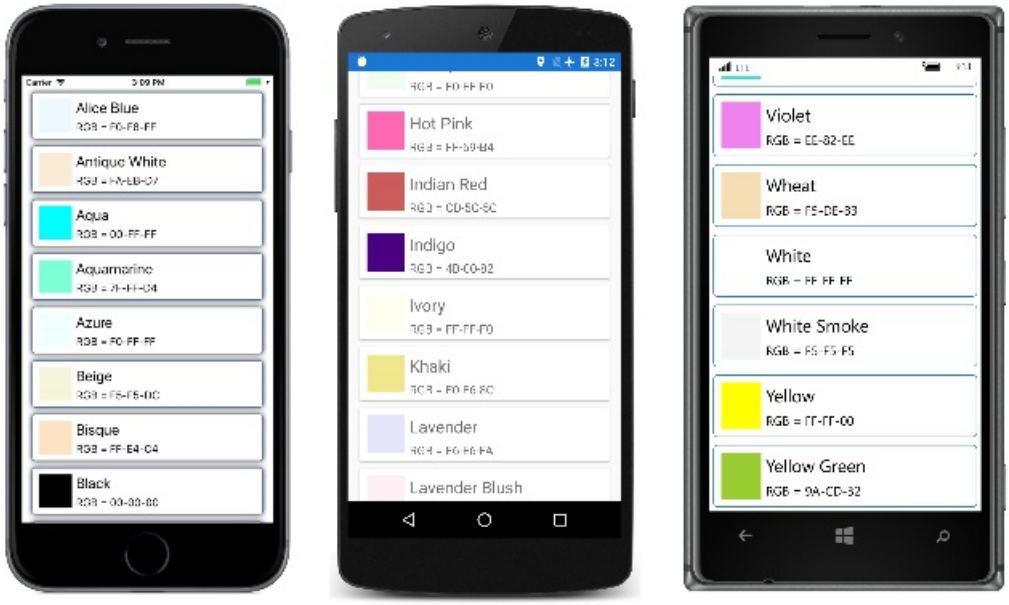
```
<BoxView HeightRequest="3" />
```

最后，您可以绘制一条竖线一侧的文本的段落括两者 `BoxView` 并 `Label` 水平 `StackLayout` 。在本例中为的高度 `BoxView` 的高度相同 `StackLayout` ，其中所依据的高度 `Label` ：

```
<StackLayout Orientation="Horizontal">
    <BoxView WidthRequest="4"
             Margin="0, 0, 10, 0" />
    <Label>

        ...

    </Label>
</StackLayout>
```

列出的颜色的字数

`BoxView` 便于在显示颜色。此程序使用 `ListView` 若要列出所有的公共静态只读字段的 Xamarin.Forms `Color` 结构：



ListViewColors 程序包括一个名为类 `NamedColor` 。静态构造函数使用反射访问的所有字段 `Color` 结构，并创建 `NamedColor` 为每个对象。它们存储在静态 `All` 属性：

```
public class NamedColor
```

```
{
    // Instance members.
    private NamedColor()
    {
    }

    public string Name { private set; get; }

    public string FriendlyName { private set; get; }

    public Color Color { private set; get; }

    public string RgbDisplay { private set; get; }

    // Static members.
    static NamedColor()
    {
        List<NamedColor> all = new List<NamedColor>();
        StringBuilder stringBuilder = new StringBuilder();

        // Loop through the public static fields of the Color structure.
        foreach (FieldInfo fieldInfo in typeof(Color).GetRuntimeFields ())
        {
            if (fieldInfo.IsPublic &&
                fieldInfo.IsStatic &&
                fieldInfo.FieldType == typeof (Color))
            {
                // Convert the name to a friendly name.
                string name = fieldInfo.Name;
                stringBuilder.Clear();
                int index = 0;

                foreach (char ch in name)
                {
                    if (index != 0 && Char.IsUpper(ch))
                    {
                        stringBuilder.Append(' ');
                    }
                    stringBuilder.Append(ch);
                    index++;
                }

                // Instantiate a NamedColor object.
                Color color = (Color)fieldInfo.GetValue(null);

                NamedColor namedColor = new NamedColor
                {
                    Name = name,
                    FriendlyName = stringBuilder.ToString(),
                    Color = color,
                    RgbDisplay = String.Format("{0:X2}-{1:X2}-{2:X2}",
                                               (int)(255 * color.R),
                                               (int)(255 * color.G),
                                               (int)(255 * color.B))
                };

                // Add it to the collection.
                all.Add(namedColor);
            }
        }
        all.TrimExcess();
        All = all;
    }

    public static IList<NamedColor> All { private set; get; }
}
```

XAML 文件中描述了程序视觉对象。`ItemsSource`的属性`ListView`设置为静态`NamedColor.All`属性，这意味着`ListView`显示所有单个`NamedColor`对象：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ListViewColors"
             x:Class="ListViewColors.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="10, 20, 10, 0" />
            <On Platform="Android, UWP" Value="10, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <ListView SeparatorVisibility="None"
              ItemsSource="{x:Static local:NamedColor.All}">
        <ListView.RowHeight>
            <OnPlatform x:TypeArguments="x:Int32">
                <On Platform="iOS, Android" Value="80" />
                <On Platform="UWP" Value="90" />
            </OnPlatform>
        </ListView.RowHeight>

        <ListView.ItemTemplate>
            <DataTemplate>
                <ViewCell>
                    <ContentView Padding="5">
                        <Frame OutlineColor="Accent"
                               Padding="10">
                            <StackLayout Orientation="Horizontal">
                                <BoxView Color="{Binding Color}"
                                         WidthRequest="50"
                                         HeightRequest="50" />
                                <StackLayout>
                                    <Label Text="{Binding FriendlyName}"
                                           FontSize="22"
                                           VerticalOptions="StartAndExpand" />
                                    <Label Text="{Binding RgbDisplay, StringFormat='RGB = {0}'}"
                                           FontSize="16"
                                           VerticalOptions="CenterAndExpand" />
                                </StackLayout>
                            </StackLayout>
                        </Frame>
                    </ContentView>
                </ViewCell>
            </DataTemplate>
        </ListView.ItemTemplate>
    </ListView>
</ContentPage>
```

`NamedColor`对象的格式由`ViewCell`对象，它将设置为的数据模板`ListView`。此模板包括`BoxView`其`Color`属性绑定到`Color`属性的`NamedColor`对象。

玩游戏通过子类化字数生命

游戏的生命周期是移动电话自动机由数学知识的人 John Conway 发明和的页中在那时推广*科学记数法美国*在二十世纪七十年代。维基百科文章提供详细的介绍Conway 游戏的生活。

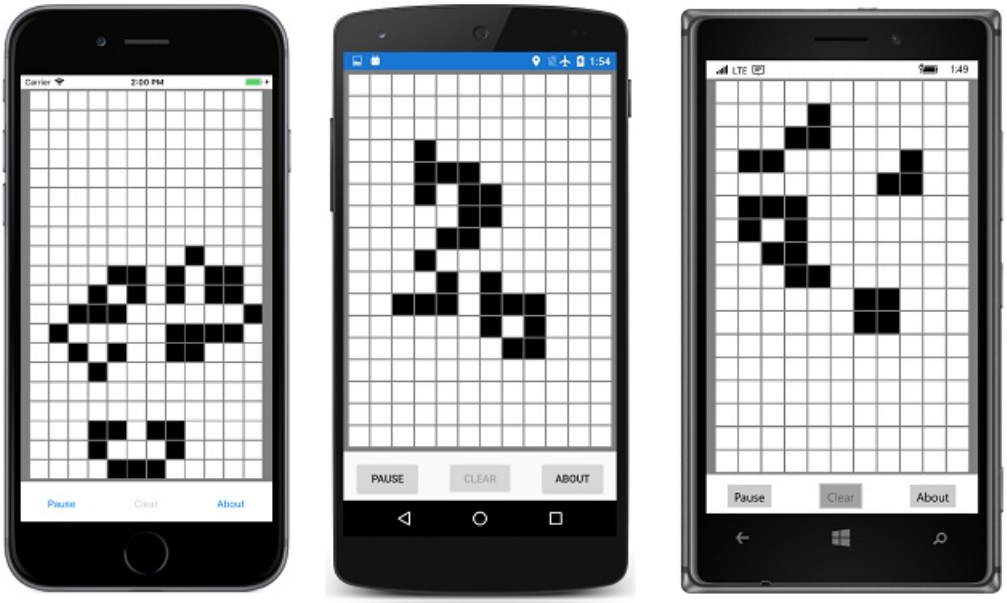
Xamarin.Forms **GameOfLife** 程序定义一个名为`LifeCell`派生`BoxView`。此类封装的逻辑的游戏的生命周期中的单个单元格：

```
class LifeCell : BoxView
{
    bool isAlive;

    public event EventHandler Tapped;

    public LifeCell()
    {
        BackgroundColor = Color.White;

        TapGestureRecognizer tapGesture = new TapGestureRecognizer();
        tapGesture.Tapped += (sender, args) =>
        {
            Tapped?.Invoke(this, EventArgs.Empty);
        };
        GestureRecognizers.Add(tapGesture);
    }

    public int Col { set; get; }

    public int Row { set; get; }

    public bool IsAlive
    {
        set
        {
            if (isAlive != value)
            {
                isAlive = value;
                BackgroundColor = isAlive ? Color.Black : Color.White;
            }
        }
        get
        {
            return isAlive;
        }
    }
}
```

`LifeCell` 将添加到其他三个属性 `BoxView`：`Col` 并 `Row` 属性存储在网格内单元格的位置和 `IsAlive` 属性指示其状态。`IsAlive` 属性还会设置 `Color` 属性的 `BoxView` 为黑色单元格是否处于活动状态，和白色如果该单元格不处于活动状态。

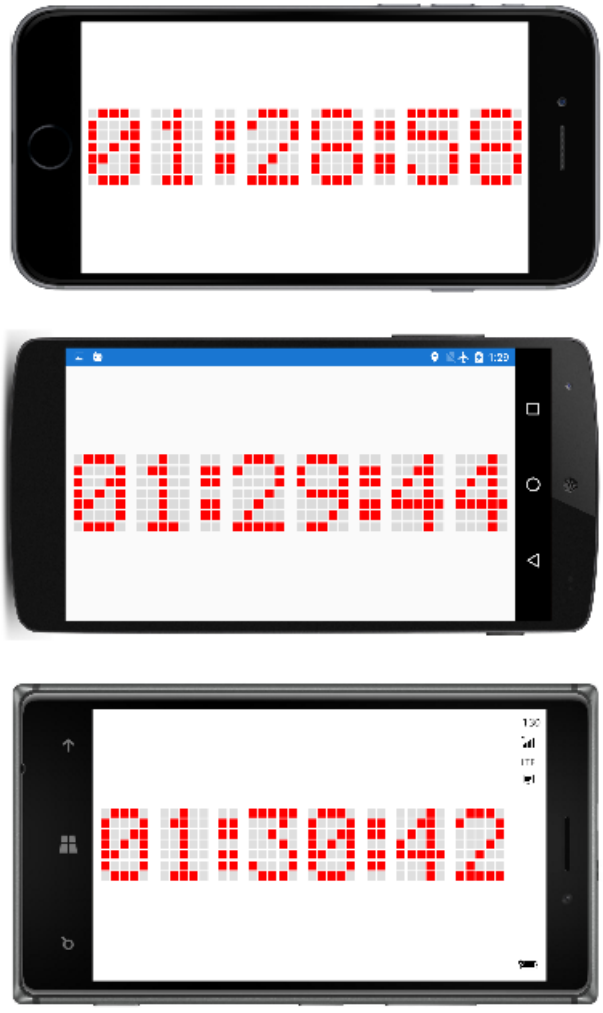
`LifeCell` 此外会安装 `TapGestureRecognizer` 以允许用户在点击它们切换单元格的状态。类将 `Tapped` 事件到其自己的手势识别器从 `Tapped` 事件。

GameOfLife程序还包括 `LifeGrid` 封装的逻辑的游戏时，大部分的类和一个 `MainPage` 处理程序的视觉对象的类。其中包括介绍游戏的规则的覆盖。下面是该程序中显示几个量为几百 `LifeCell` 页上的对象：



创建数字时钟

DotMatrixClock 程序创建 210 `BoxView` 元素，以模拟老式的 5 到 7 点阵显示的点。可以读取在纵向或横向模式下的时间，但它是在环境中更大：



XAML 文件 does 稍有多个实例化 `AbsoluteLayout` 用于时钟：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DotMatrixClock"
             x:Class="DotMatrixClock.MainPage"
             Padding="10"
             SizeChanged="OnPageSizeChanged">

    <AbsoluteLayout x:Name="absoluteLayout"
                    VerticalOptions="Center" />
</ContentPage>
```

所有其他代码隐藏文件中出现。圆点矩阵显示逻辑大大简化了描述对应于每个 10 位数字和一个冒号的点的多个数组的定义：

```
public partial class MainPage : ContentPage
{
    // Total dots horizontally and vertically.
    const int horzDots = 41;
    const int vertDots = 7;

    // 5 x 7 dot matrix patterns for 0 through 9.
    static readonly int[, ,] numberPatterns = new int[10, 7, 5]
    {
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 1, 1}, { 1, 0, 1, 0, 1},
            { 1, 1, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 0, 0}, { 0, 1, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0},
            { 0, 0, 1, 0, 0}, { 0, 0, 1, 0, 0}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0},
            { 0, 0, 1, 0, 0}, { 0, 1, 0, 0, 0}, { 1, 1, 1, 1, 1}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0}, { 0, 0, 0, 1, 0},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 0, 1, 0}, { 0, 0, 1, 1, 0}, { 0, 1, 0, 1, 0}, { 1, 0, 0, 1, 0},
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 0, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0}, { 0, 0, 0, 0, 1},
            { 0, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 0, 1, 1, 0}, { 0, 1, 0, 0, 0}, { 1, 0, 0, 0, 0}, { 1, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 1, 1, 1, 1, 1}, { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 0, 1, 0, 0},
            { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}, { 0, 1, 0, 0, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0},
            { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 0}
        },
        {
            { 0, 1, 1, 1, 0}, { 1, 0, 0, 0, 1}, { 1, 0, 0, 0, 1}, { 0, 1, 1, 1, 1},
            { 0, 0, 0, 0, 1}, { 0, 0, 0, 1, 0}, { 0, 1, 1, 0, 0}
        },
    };

    // Dot matrix pattern for a colon.
```

```
    // Dot matrix pattern for a colon.
    static readonly int[,] colonPattern = new int[7, 2]
    {
        { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }, { 1, 1 }, { 1, 1 }, { 0, 0 }
    };

    // BoxView colors for on and off.
    static readonly Color colorOn = Color.Red;
    static readonly Color colorOff = new Color(0.5, 0.5, 0.5, 0.25);

    // Box views for 6 digits, 7 rows, 5 columns.
    BoxView[, ,] digitBoxViews = new BoxView[6, 7, 5];

    ...

}
```

这些字段演示的三维数组的最后 `BoxView` 元素用于存储六位数字的圆点模式。

该构造函数创建所有 `BoxView` 数字和冒号，以及初始化元素 `Color` 属性的 `BoxView` 冒号的元素：

```
public partial class MainPage : ContentPage
{

    ...

    public MainPage()
    {
        InitializeComponent();

        // BoxView dot dimensions.
        double height = 0.85 / vertDots;
        double width = 0.85 / horzDots;

        // Create and assemble the BoxViews.
        double xIncrement = 1.0 / (horzDots - 1);
        double yIncrement = 1.0 / (vertDots - 1);
        double x = 0;

        for (int digit = 0; digit < 6; digit++)
        {
            for (int col = 0; col < 5; col++)
            {
                double y = 0;

                for (int row = 0; row < 7; row++)
                {
                    // Create the digit BoxView and add to layout.
                    BoxView boxView = new BoxView();
                    digitBoxViews[digit, row, col] = boxView;
                    absoluteLayout.Children.Add(boxView,
                                                new Rectangle(x, y, width, height),
                                                AbsoluteLayoutFlags.All);
                    y += yIncrement;
                }
                x += xIncrement;
            }
            x += xIncrement;

            // Colons between the hours, minutes, and seconds.
            if (digit == 1 || digit == 3)
            {
                int colon = digit / 2;

                for (int col = 0; col < 2; col++)
                {
                    double y = 0;
```

```
                    for (int row = 0; row < 7; row++)
                    {
                        // Create the BoxView and set the color.
                        BoxView boxView = new BoxView
                            {
                                Color = colonPattern[row, col] == 1 ?
                                            colorOn : colorOff
                            };
                        absoluteLayout.Children.Add(boxView,
                                                    new Rectangle(x, y, width, height),
                                                    AbsoluteLayoutFlags.All);
                        y += yIncrement;
                    }
                    x += xIncrement;
                }
                x += xIncrement;
            }
        }

        // Set the timer and initialize with a manual call.
        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimer);
        OnTimer();
    }

    ...

}
```

此程序使用的相对定位和大小调整功能`AbsoluteLayout`。宽度和高度的每个`BoxView`设置为小数值，专门 85%的 1 除以水平和垂直点数目。位置也都设置为小数部分值。

因为所有位置和大小都都相对于总大小`AbsoluteLayout`，则`SizeChanged`页的处理程序只需要设置`HeightRequest`的`AbsoluteLayout`：

```
public partial class MainPage : ContentPage
{

    ...

    void OnPageSizeChanged(object sender, EventArgs args)
    {
        // No chance a display will have an aspect ratio > 41:7
        absoluteLayout.HeightRequest = vertDots * Width / horzDots;
    }

    ...

}
```

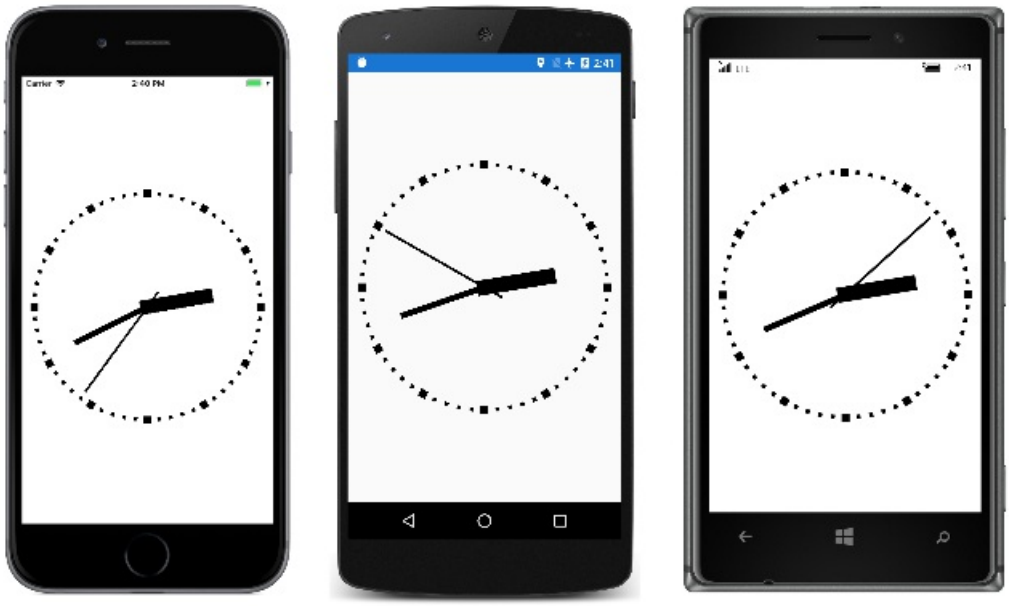
宽度`AbsoluteLayout`自动设置，因为它拉伸到整个页面的宽度。

中的最终代码`MainPage`类处理计时器回调和颜色的每个数字的点。在代码隐藏文件开头多维数组的定义可帮助使此逻辑的程序的最简单的一部分：

```
public partial class MainPage : ContentPage
{

    ...

    bool OnTimer()
    {
        DateTime dateTime = DateTime.Now;

        // Convert 24-hour clock to 12-hour clock.
        int hour = (dateTime.Hour + 11) % 12 + 1;

        // Set the dot colors for each digit separately.
        SetDotMatrix(0, hour / 10);
        SetDotMatrix(1, hour % 10);
        SetDotMatrix(2, dateTime.Minute / 10);
        SetDotMatrix(3, dateTime.Minute % 10);
        SetDotMatrix(4, dateTime.Second / 10);
        SetDotMatrix(5, dateTime.Second % 10);
        return true;
    }

    void SetDotMatrix(int index, int digit)
    {
        for (int row = 0; row < 7; row++)
            for (int col = 0; col < 5; col++)
            {
                bool isOn = numberPatterns[digit, row, col] == 1;
                Color color = isOn ? colorOn : colorOff;
                digitBoxViews[index, row, col].Color = color;
            }
    }
}
```

创建模拟时钟

点式时钟可能看起来是显而易见的应用程序 `BoxView`，但 `BoxView` 元素也是能够意识到了模拟时钟：



中的所有视觉对象 **BoxViewClock** 程序是的子级 `AbsoluteLayout`。这些元素的大小将调整使用 `LayoutBounds` 附加属性，并使用旋转 `Rotation` 属性。

这三个 `BoxView` 时钟指针的元素在 XAML 文件中，实例化但未定位或调整大小：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:BoxViewClock"
             x:Class="BoxViewClock.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <AbsoluteLayout x:Name="absoluteLayout"
                    SizeChanged="OnAbsoluteLayoutSizeChanged">

        <BoxView x:Name="hourHand"
                 Color="Black" />

        <BoxView x:Name="minuteHand"
                 Color="Black" />

        <BoxView x:Name="secondHand"
                 Color="Black" />
    </AbsoluteLayout>
</ContentPage>
```

代码隐藏文件的构造函数实例化 60 `BoxView` 刻度线在时钟圆周的周围的元素：

```
public partial class MainPage : ContentPage
{

    ...

    BoxView[] tickMarks = new BoxView[60];

    public MainPage()
    {
        InitializeComponent();

        // Create the tick marks (to be sized and positioned later).
        for (int i = 0; i < tickMarks.Length; i++)
        {
            tickMarks[i] = new BoxView { Color = Color.Black };
            absoluteLayout.Children.Add(tickMarks[i]);
        }

        Device.StartTimer(TimeSpan.FromSeconds(1.0 / 60), OnTimerTick);
    }

    ...

}
```

所有的位置和大小 `BoxView` 元素中出现 `SizeChanged` 处理程序 `AbsoluteLayout` 。内部的类的一个小结构称为 `HandParams` 介绍每三个手中相对于时钟的总大小的大小：

```
public partial class MainPage : ContentPage
{
    // Structure for storing information about the three hands.
    struct HandParams
    {
        public HandParams(double width, double height, double offset) : this()
        {
            Width = width;
            Height = height;
            Offset = offset;
        }

        public double Width { private set; get; }   // fraction of radius
        public double Height { private set; get; }  // ditto
        public double Offset { private set; get; }  // relative to center pivot
    }

    static readonly HandParams secondParams = new HandParams(0.02, 1.1, 0.85);
    static readonly HandParams minuteParams = new HandParams(0.05, 0.8, 0.9);
    static readonly HandParams hourParams = new HandParams(0.125, 0.65, 0.9);

    ...

 }
```

`SizeChanged`处理程序确定的中心和半径`AbsoluteLayout`，然后调整大小和定位 60 `BoxView`用作刻度的元素。`for`循环的结尾部分通过设置`Rotation`属性的每种`BoxView`元素。在末尾`SizeChanged`处理程序，`LayoutHand`调用方法来调整大小和位置时钟三个指针：

```
public partial class MainPage : ContentPage
{

    ...

    void OnAbsoluteLayoutSizeChanged(object sender, EventArgs args)
    {
        // Get the center and radius of the AbsoluteLayout.
        Point center = new Point(absoluteLayout.Width / 2, absoluteLayout.Height / 2);
        double radius = 0.45 * Math.Min(absoluteLayout.Width, absoluteLayout.Height);

        // Position, size, and rotate the 60 tick marks.
        for (int index = 0; index < tickMarks.Length; index++)
        {
            double size = radius / (index % 5 == 0 ? 15 : 30);
            double radians = index * 2 * Math.PI / tickMarks.Length;
            double x = center.X + radius * Math.Sin(radians) - size / 2;
            double y = center.Y - radius * Math.Cos(radians) - size / 2;
            AbsoluteLayout.SetLayoutBounds(tickMarks[index], new Rectangle(x, y, size, size));
            tickMarks[index].Rotation = 180 * radians / Math.PI;
        }

        // Position and size the three hands.
        LayoutHand(secondHand, secondParams, center, radius);
        LayoutHand(minuteHand, minuteParams, center, radius);
        LayoutHand(hourHand, hourParams, center, radius);
    }

    void LayoutHand(BoxView boxView, HandParams handParams, Point center, double radius)
    {
        double width = handParams.Width * radius;
        double height = handParams.Height * radius;
        double offset = handParams.Offset;

        AbsoluteLayout.SetLayoutBounds(boxView,
            new Rectangle(center.X - 0.5 * width,
                          center.Y - offset * height,
                          width, height));

        // Set the AnchorY property for rotations.
        boxView.AnchorY = handParams.Offset;
    }

    ...

}
```

`LayoutHand` 方法大小和位置以垂直向上 12:00 位置点每个指针。该方法末尾 `AnchorY` 属性设置为对应于时钟的中心的位置。这表示旋转中心。

手轮换计时器回调函数中：

```
public partial class MainPage : ContentPage
{

    ...

    bool OnTimerTick()
    {
        // Set rotation angles for hour and minute hands.
        DateTime dateTime = DateTime.Now;
        hourHand.Rotation = 30 * (dateTime.Hour % 12) + 0.5 * dateTime.Minute;
        minuteHand.Rotation = 6 * dateTime.Minute + 0.1 * dateTime.Second;

        // Do an animation for the second hand.
        double t = dateTime.Millisecond / 1000.0;

        if (t < 0.5)
        {
            t = 0.5 * Easing.SpringIn.Ease(t / 0.5);
        }
        else
        {
            t = 0.5 * (1 + Easing.SpringOut.Ease((t - 0.5) / 0.5));
        }

        secondHand.Rotation = 6 * (dateTime.Second + t);
        return true;
    }
}
```

第二个指针的处理方式稍有不同：缓动函数的动画应用以使移动看上去机械，而不是平滑。在每个时钟周期，第二个指针有点拉取，然后超过其目标。此小段代码很向移动的真实性。

结束语

`BoxView` 可能看起来简单在第一次，而是作为你所见，它可以相当灵活多变，并且可以几乎重现的视觉效果通常可以仅使用矢量图形。有关更复杂的图形，请查阅Xamarin.Forms 中使用 SkiaSharp。

相关链接

- 基本字数（示例）
- 文本修饰（示例）
- 列表框颜色（示例）
- 游戏的生命周期（示例）
- 点式时钟（示例）
- 字数时钟（示例）
- BoxView

Xamarin.Forms 按钮

2018/11/13 • • Edit Online

按钮响应点击或单击，将定向的应用程序来执行特定任务。

`Button` 是所有 Xamarin.Forms 中最基本的交互控件。`Button` 通常会显示一个短文本字符串，指示某个命令，但它还可以显示图像和位图图像，或文本的组合。用户按 `Button` 用手指或用来启动该命令的鼠标单击。

大部分下面讨论的主题中的页对应 **ButtonDemos** 示例。

处理按钮单击

`Button` 定义 `Clicked` 在用户点击时激发的事件 `Button` 用手指或鼠标指针。图面中释放手指或鼠标按钮时触发该事件 `Button`。`Button` 必须具有其 `IsEnabled` 属性设置为 `true`，以便响应分流点。

基本按钮单击页面 **ButtonDemos** 示例演示如何实例化 `Button` XAML 和句柄中其 `Clicked` 事件。**BasicButtonClickPage.xaml**文件包含 `StackLayout` 两个 `Label` 和 `Button`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonDemos.BasicButtonClickPage"
             Title="Basic Button Click">
    <StackLayout>

        <Label x:Name="label"
               Text="Click the Button below"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Button Text="Click to Rotate Text!"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Clicked="OnButtonClicked" />

    </StackLayout>
</ContentPage>
```

`Button` 往往会占用为其允许的所有空间。例如，如果未设置 `HorizontalOptions` 的属性 `Button` 到以外的其他 `Fill`，则 `Button` 将占用其父项的整个宽度。

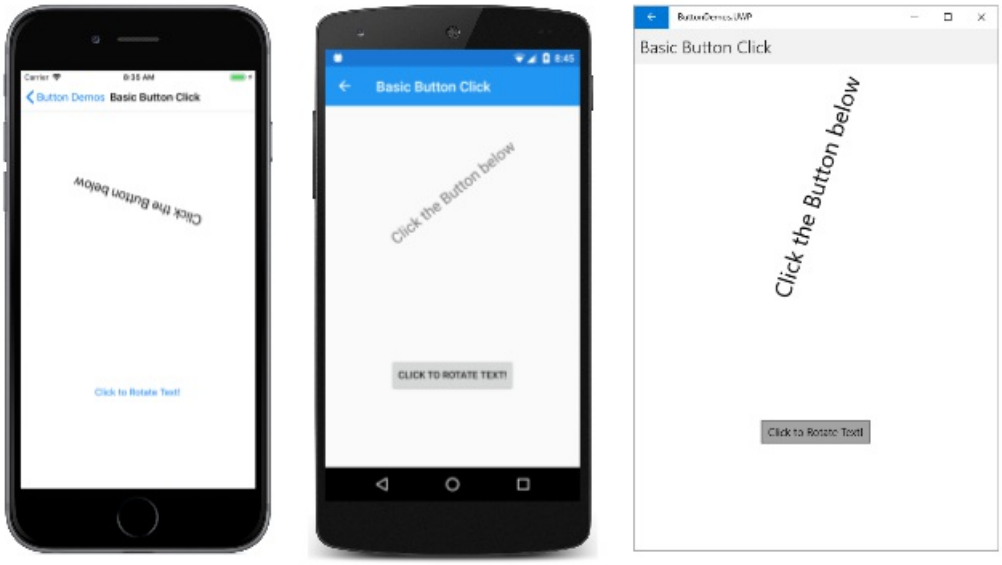
默认情况下 `Button` 都是矩形状，但可以通过使用进行舍入的 it 角 `CornerRadius` 属性，如下所述的部分中**按钮外观**.

`Text` 属性指定在显示的文本 `Button`。`Clicked` 事件设置为事件处理程序名为 `OnButtonClicked`。在代码隐藏文件中，找到此处理程序**BasicButtonClickPage.xaml.cs**：

```
public partial class BasicButtonClickPage : ContentPage
{
    public BasicButtonClickPage ()
    {
        InitializeComponent ();
    }

    async void OnButtonClicked(object sender, EventArgs args)
    {
        await label.RelRotateTo(360, 1000);
    }
}
```

当 `Button` 点击，`OnButtonClicked` 方法执行。`sender` 自变量是 `Button` 负责此事件的对象。可以使用此访问 `Button` 对象，或以区分多个 `Button` 对象共享相同 `Clicked` 事件。

此特定 `Clicked` 处理程序会调用旋转动画函数 `Label` 360 度，在 1000年毫秒。下面是在 Windows 10 桌面版上运行 iOS 和 Android 设备，并作为通用 Windows 平台 (UWP) 应用程序的程序：



请注意，`OnButtonClicked` 方法包括 `async` 修饰符因为 `await` 内的事件处理程序使用。一个 `Clicked` 事件处理程序需要 `async` 修饰符仅当处理程序的主体使用 `await` 。

每个平台呈现 `Button` 以其自己特定的方式。在中**按钮外观**部分中，您将了解如何设置颜色并使 `Button` 边框可见的更多自定义外观。`Button` 实现 `IFontElement` 接口，因此，它包含 `FontFamily` ， `FontSize` ，并 `FontAttributes` 属性。

在代码中创建一个按钮

往往会实例化 `Button` 在 XAML 中，但也可以创建 `Button` 在代码中。这可能会非常方便您的应用程序需要创建基于数据是使用可枚举的多个按钮时 `foreach` 循环。

代码按钮单击页将演示如何创建的页面功能上等效于基本按钮单击但完全在页C#:

```
public class CodeButtonClickPage : ContentPage
{
    public CodeButtonClickPage ()
    {
        Title = "Code Button Click";

        Label label = new Label
        {
            Text = "Click the Button below",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };

        Button button = new Button
        {
            Text = "Click to Rotate Text!",
            VerticalOptions = LayoutOptions.CenterAndExpand,
            HorizontalOptions = LayoutOptions.Center
        };
        button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);

        Content = new StackLayout
        {
            Children =
            {
                label,
                button
            }
        };
    }
}
```

类的构造函数中完成的所有内容。因为 `Clicked` 处理程序只有一个语句很长，则可以将它附加到事件非常简单：

```
button.Clicked += async (sender, args) => await label.RelRotateTo(360, 1000);
```

当然，您还可以作为一个单独的方法定义的事件处理程序（就像 `OnButtonClick` 中的方法基本按钮单击）并将该方法附加到该事件：

```
button.Clicked += OnButtonClicked;
```

禁用按钮

有时应用程序处于特定状态的特定 `Button` 单击不是有效的操作。在这些情况下，`Button` 应禁用通过设置其 `IsEnabled` 属性设置为 `false`。典型的示例是 `Entry` 伴随文件打开的文件名的控件 `Button`：`Button` 键入一些文本，才应启用 `Entry`。可以使用 `DataTrigger` 对于此任务，如中所示**数据触发器**一文。

使用命令界面

它是应用程序以响应 `Button` 分流点无需处理 `Clicked` 事件。`Button` 实现调用了替代通知机制_命令_或_命令_接口。这包括两个属性：

- `Command` 类型的 `ICommand`，在中定义的接口 `System.Windows.Input` 命名空间。
- `CommandParameter` 类型的属性 `Object`。

尤其是在实现模型-视图-视图模型 (MVVM) 体系结构时，这种方法是特别适合与数据绑定和。在文章中讨论了这些主题数据绑定，从数据绑定到 MVVM，并MVVM。

在 MVVM 应用程序，ViewModel 定义类型的属性 `ICommand`，然后连接到 XAML `Button` 具有数据绑定的元素。此外定义了 Xamarin.Forms `Command` 并 `Command<T>` 类实现 `ICommand` 接口，并帮助 ViewModel 中定义的类型属性 `ICommand`.

命令一文中的更详细地介绍**命令界面**但**基本按钮命令**页中 **ButtonDemos** 示例显示了基本的方法。

`CommandDemoViewModel` 类是非常简单的 ViewModel，用于定义类型的属性 `double` 名为 `Number`，和类型的两个属性 `ICommand` 名为 `MultiplyBy2Command` 和 `DivideBy2Command`：

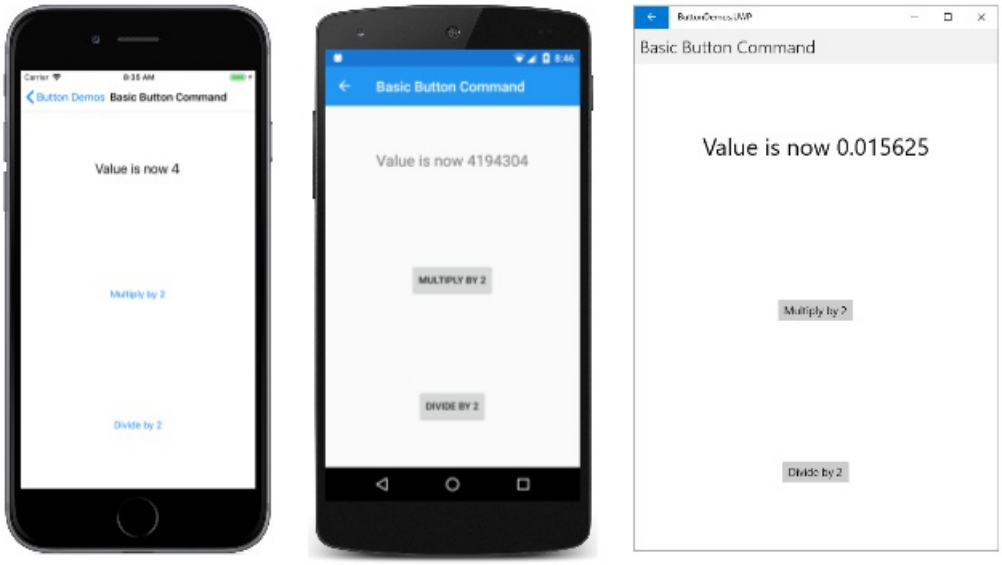
```
class CommandDemoViewModel : INotifyPropertyChanged
{
    double number = 1;

    public event PropertyChangedEventHandler PropertyChanged;

    public CommandDemoViewModel()
    {
        MultiplyBy2Command = new Command(() => Number *= 2);

        DivideBy2Command = new Command(() => Number /= 2);
    }

    public double Number
    {
        set
        {
            if (number != value)
            {
                number = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Number"));
            }
        }
        get
        {
            return number;
        }
    }

    public ICommand MultiplyBy2Command { private set; get; }

    public ICommand DivideBy2Command { private set; get; }
}
```

这两个 `ICommand` 属性的使用类型的两个对象的类的构造函数初始化 `Command`。`Command` 构造函数包括一些函数 (称为 `execute` 构造函数参数) 的两倍，或减半 `Number` 属性。

BasicButtonCommand.xaml文件中设置其 `BindingContext` 的实例 `CommandDemoViewModel`。`Label` 元素和第二个 `Button` 元素包含到中的三个属性的绑定 `CommandDemoViewModel`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ButtonDemos"
             x:Class="ButtonDemos.BasicButtonCommandPage"
             Title="Basic Button Command">

    <ContentPage.BindingContext>
        <local:CommandDemoViewModel />
    </ContentPage.BindingContext>

    <StackLayout>
        <Label Text="{Binding Number, StringFormat='Value is now {0}'}"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Button Text="Multiply by 2"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Command="{Binding MultiplyBy2Command}" />

        <Button Text="Divide by 2"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Command="{Binding DivideBy2Command}" />
    </StackLayout>
</ContentPage>
```

与两个 `Button` 点击元素、执行命令，并更改值的数量：

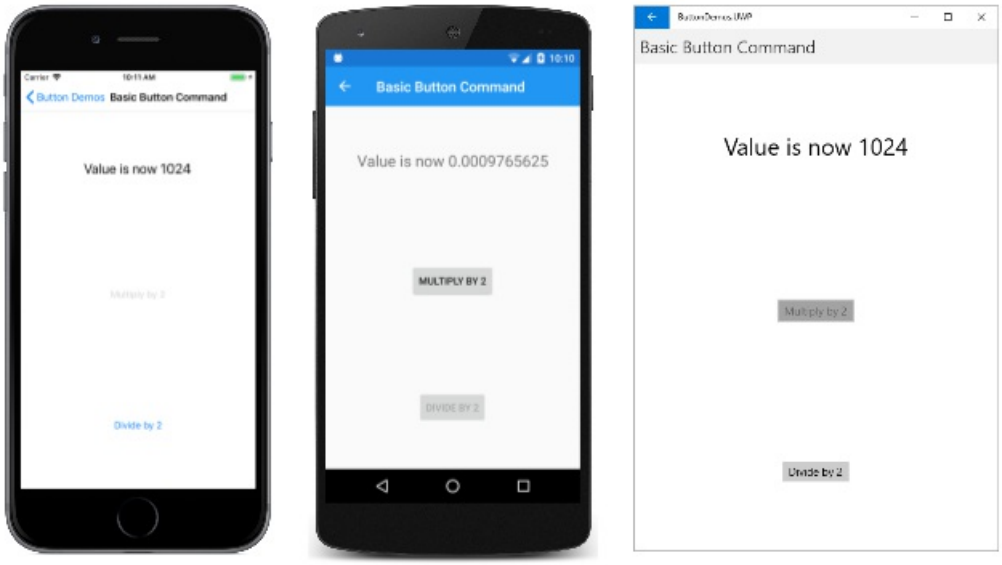


通过此方法的优点 `Clicked` 处理程序，该文件，所有逻辑相关的此页的功能都位于在 ViewModel，而不是代码隐藏文件中，实现更好的隔离的用户界面与业务逻辑。

也可能是 `Command` 对象来控制启用和禁用 `Button` 元素。例如，假设你想要限制 2 之间的数字值的范围10和2^{-10}。可以将另一个函数添加到构造函数 (称为 `canExecute` 自变量)，它返回 `true` 如果 `Button` 应启用。下面是对修改 `CommandDemoViewModel` 构造函数：

```
class CommandDemoViewModel : INotifyPropertyChanged
{
    ...
    public CommandDemoViewModel()
    {
        MultiplyBy2Command = new Command(
            execute: () =>
            {
                Number *= 2;
                ((Command)MultiplyBy2Command).ChangeCanExecute();
                ((Command)DivideBy2Command).ChangeCanExecute();
            },
            canExecute: () => Number < Math.Pow(2, 10));

        DivideBy2Command = new Command(
            execute: () =>
            {
                Number /= 2;
                ((Command)MultiplyBy2Command).ChangeCanExecute();
                ((Command)DivideBy2Command).ChangeCanExecute();
            },
            canExecute: () => Number > Math.Pow(2, -10));
    }
    ...
}
```

对调用`ChangeCanExecute`方法`Command`所必需，以便`Command`方法可以调用`canExecute`方法，并确定是否`Button`或不应禁用。此代码更改，随着数量达到限制，`Button`已禁用：



可以为两个或多个`Button`元素以绑定到同一`ICommand`属性。`Button`可以使用可分辨元素`CommandParameter`属性`Button`。在这种情况下，你将想要使用泛型`Command<T>`类。`CommandParameter`对象然后作为参数传递`execute`和`canExecute`方法。中详细地演示了此技术**基本命令**一部分**命令接口**一文。

ButtonDemos示例还使用此方法在其`MainPage`类。**MainPage.xaml**文件包含`Button`示例的每个页面：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ButtonDemos"
             x:Class="ButtonDemos.MainPage"
             Title="Button Demos">
    <ScrollView>
        <FlexLayout Direction="Column"
                    JustifyContent="SpaceEvenly"
                    AlignItems="Center">

            <Button Text="Basic Button Click"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:BasicButtonClickPage}" />

            <Button Text="Code Button Click"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:CodeButtonClickPage}" />

            <Button Text="Basic Button Command"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:BasicButtonCommandPage}" />

            <Button Text="Press and Release Button"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:PressAndReleaseButtonPage}" />

            <Button Text="Button Appearance"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ButtonAppearancePage}" />

            <Button Text="Toggle Button Demo"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ToggleButtonDemoPage}" />

            <Button Text="Image Button Demo"
                    Command="{Binding NavigateCommand}"
                    CommandParameter="{x:Type local:ImageButtonDemoPage}" />

        </FlexLayout>
    </ScrollView>
</ContentPage>
```

每个 `Button` 具有其 `Command` 属性绑定到一个名为属性 `NavigateCommand`，和 `CommandParameter` 设置为 `Type` 对应于一个项目中的页类的对象。

是否 `NavigateCommand` 属性属于类型 `ICommand` 和代码隐藏文件中定义：

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();

        NavigateCommand = new Command<Type>(async (Type pageType) =>
        {
            Page page = (Page)Activator.CreateInstance(pageType);
            await Navigation.PushAsync(page);
        });

        BindingContext = this;
    }

    public ICommand NavigateCommand { private set; get; }
}
```

构造函数初始化 `NavigateCommand` 属性设置为 `Command<Type>` 对象，因为 `Type` 是一种 `CommandParameter` XAML 文件中设置的对象。这意味着 `execute` 方法具有一个类型的参数 `Type` 的对应于此 `CommandParameter` 对象。此函数实例化页，然后转到它。

请注意，构造函数以结束通过设置其 `BindingContext` 到其自身。这对于要绑定到 XAML 文件中的属性是必需 `NavigateCommand` 属性。

按下并松开按钮

除了 `Clicked` 事件，`Button` 还定义了 `Pressed` 并 `Released` 事件。`Pressed` 手指按上时发生事件 `Button`，或使用指针置于其上按下鼠标按钮 `Button`。`Released` 松开手指或鼠标按钮时发生事件。通常情况下，`Clicked` 还在相同的时间触发事件 `Released` 事件，但如果手指或鼠标指针滑离开的面 `Button` 之前被释放，`Clicked` 事件可能会发生。

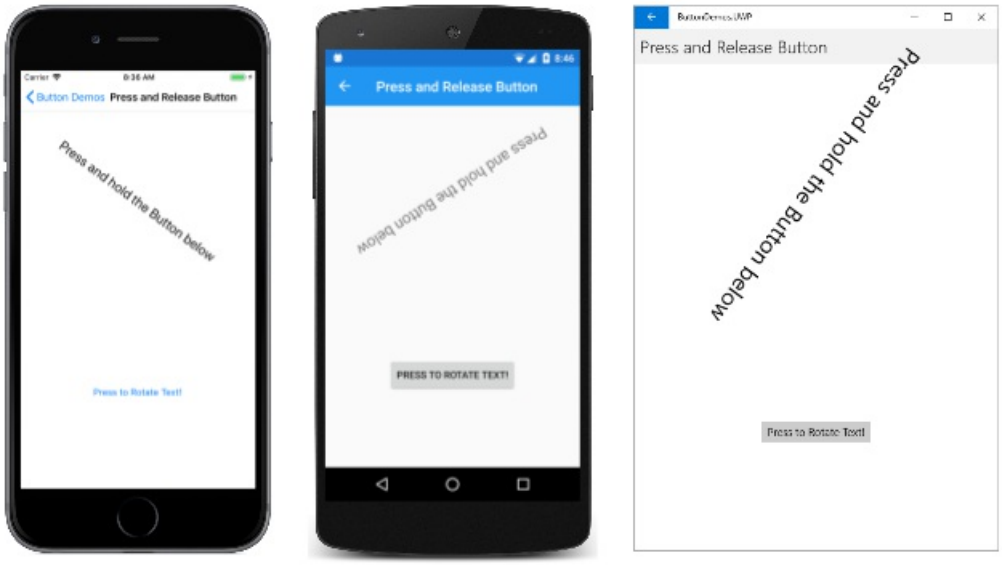
`Pressed` 并 `Released` 事件不常使用，但它们可用于特殊用途，如中所示**按下并松开按**钮页。XAML 文件包含 `Label` 和一个 `Button` 使用处理程序将其附加 `Pressed` 和 `Released` 事件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonDemos.PressAndReleaseButtonPage"
             Title="Press and Release Button">
    <StackLayout>

        <Label x:Name="label"
               Text="Press and hold the Button below"
               FontSize="Large"
               VerticalOptions="CenterAndExpand"
               HorizontalOptions="Center" />

        <Button Text="Press to Rotate Text!"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                Pressed="OnButtonPressed"
                Released="OnButtonReleased" />

    </StackLayout>
</ContentPage>
```

代码隐藏文件之间进行动画处理 `Label` 时 `Pressed` 事件发生，但挂起旋转时 `Released` 发生事件：

```
public partial class PressAndReleaseButtonPage : ContentPage
{
    bool animationInProgress = false;
    Stopwatch stopwatch = new Stopwatch();

    public PressAndReleaseButtonPage ()
    {
        InitializeComponent ();
    }

    void OnButtonPressed(object sender, EventArgs args)
    {
        stopwatch.Start();
        animationInProgress = true;

        Device.StartTimer(TimeSpan.FromMilliseconds(16), () =>
        {
            label.Rotation = 360 * (stopwatch.Elapsed.TotalSeconds % 1);

            return animationInProgress;
        });
    }

    void OnButtonReleased(object sender, EventArgs args)
    {
        animationInProgress = false;
        stopwatch.Stop();
    }
}
```

结果是，`Label`手指与联系时仅会旋转`Button`，并在手指松开时停止：



游戏的应用有这种行为：手指上持有`Button`可能会使特定方向移动一个在屏幕对象。

按钮外观

`Button`继承或定义会影响其外观的多个属性：

- `TextColor`是的颜色`Button`文本
- `BackgroundColor`是对该文本背景的颜色
- `BorderColor`是的周围区域的颜色`Button`
- `FontFamily`使用文本的字体系列
- `FontSize`是文本的大小

- `FontAttributes` 指示文本是斜体或粗体
- `BorderWidth` 为边框的宽度
- `CornerRadius` 为圆角

> **NOTE**
>
> `Button` 类还具有 `Margin` 并 `Padding` 控制的布局行为的属性 `Button` 。有关详细信息，请参阅边距和填充。

六篇文章构成这些属性的效果（不包括 `FontFamily` 并 `FontAttributes` ）中演示按钮外观页。另一个属性， `Image` ，节中讨论位图使用按钮。

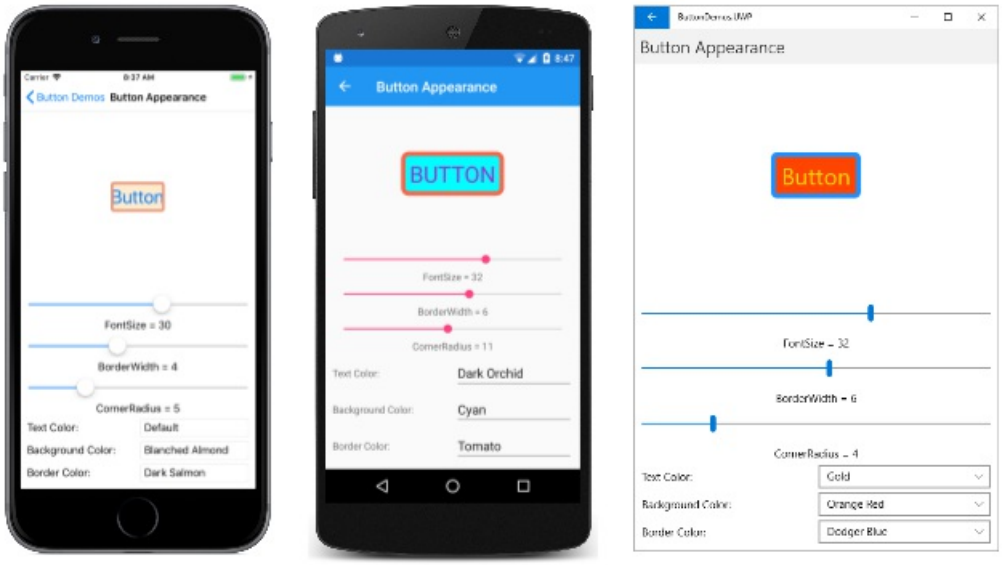
中的视图和数据绑定的所有按钮外观XAML 文件中定义页面：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ButtonDemos"
             x:Class="ButtonDemos.ButtonAppearancePage"
             Title="Button Appearance">
    <StackLayout>
        <Button x:Name="button"
                Text="Button"
                VerticalOptions="CenterAndExpand"
                HorizontalOptions="Center"
                TextColor="{Binding Source={x:Reference textColorPicker},
                                    Path=SelectedItem.Color}"
                BackgroundColor="{Binding Source={x:Reference backgroundColorPicker},
                                          Path=SelectedItem.Color}"
                BorderColor="{Binding Source={x:Reference borderColorPicker},
                                      Path=SelectedItem.Color}" />

        <StackLayout BindingContext="{x:Reference button}"
                     Padding="10">

            <Slider x:Name="fontSizeSlider"
                    Maximum="48"
                    Minimum="1"
                    Value="{Binding FontSize}" />

            <Label Text="{Binding Source={x:Reference fontSizeSlider},
                                  Path=Value,
                                  StringFormat='FontSize = {0:F0}'}"
                   HorizontalTextAlignment="Center" />

            <Slider x:Name="borderWidthSlider"
                    Minimum="-1"
                    Maximum="12"
                    Value="{Binding BorderWidth}" />

            <Label Text="{Binding Source={x:Reference borderWidthSlider},
                                  Path=Value,
                                  StringFormat='BorderWidth = {0:F0}'}"
                   HorizontalTextAlignment="Center" />

            <Slider x:Name="cornerRadiusSlider"
                    Minimum="-1"
                    Maximum="24"
                    Value="{Binding CornerRadius}" />

            <Label Text="{Binding Source={x:Reference cornerRadiusSlider},
                                  Path=Value,
                                  StringFormat='CornerRadius = {0:F0}'}"
                   HorizontalTextAlignment="Center" />

            <Grid>
```

```
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>

                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>

                <Grid.Resources>
                    <Style TargetType="Label">
                        <Setter Property="VerticalOptions" Value="Center" />
                    </Style>
                </Grid.Resources>

                <Label Text="Text Color:"
                       Grid.Row="0" Grid.Column="0" />

                <Picker x:Name="textColorPicker"
                        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
                        ItemDisplayBinding="{Binding FriendlyName}"
                        SelectedIndex="0"
                        Grid.Row="0" Grid.Column="1" />

                <Label Text="Background Color:"
                       Grid.Row="1" Grid.Column="0" />

                <Picker x:Name="backgroundColorPicker"
                        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
                        ItemDisplayBinding="{Binding FriendlyName}"
                        SelectedIndex="0"
                        Grid.Row="1" Grid.Column="1" />

                <Label Text="Border Color:"
                       Grid.Row="2" Grid.Column="0" />

                <Picker x:Name="borderColorPicker"
                        ItemsSource="{Binding Source={x:Static local:NamedColor.All}}"
                        ItemDisplayBinding="{Binding FriendlyName}"
                        SelectedIndex="0"
                        Grid.Row="2" Grid.Column="1" />
            </Grid>
        </StackLayout>
    </StackLayout>
</ContentPage>
```

`Button`在页面顶部有其三个`Color`属性绑定到`Picker`在页面底部的元素。中的项`Picker`元素是从颜色`NamedColor`项目中包含的类。三个`Slider`元素包含为双向绑定`FontSize`，`BorderWidth`，和`CornerRadius`的属性`Button`。

此程序可以尝试使用所有这些属性的组合：



若要查看 `Button` 边框，您将需要设置 `BorderColor` 到以外的其他 `Default`，和 `BorderWidth` 为正值。

在 iOS 上，您会注意到，大边框宽度起、强行进入到的内部 `Button` 和干扰中文本的显示。如果您选择要用于 iOS 的边框 `Button`，可能需要开始和结束 `Text` 空格保留其可见性属性。

在 UWP 中，选择 `CornerRadius` 超过高度的一半 `Button` 引发异常。

创建一个切换按钮

子类可以 `Button`，使其工作原理类似打开-关闭开关：点击按钮一次以上切换按钮，然后点击它再次切换它禁用。

以下 `ToggleButton` 类派生自 `Button`，并定义名为的新事件 `Toggled` 和名为的布尔属性 `IsToggled`。这些是相同的两个属性定义的 Xamarin.Forms `Switch`：

```
class ToggleButton : Button
{
    public event EventHandler<ToggledEventArgs> Toggled;

    public static BindableProperty IsToggledProperty =
        BindableProperty.Create("IsToggled", typeof(bool), typeof(ToggleButton), false,
                                propertyChanged: OnIsToggledChanged);

    public ToggleButton()
    {
        Clicked += (sender, args) => IsToggled ^= true;
    }

    public bool IsToggled
    {
        set { SetValue(IsToggledProperty, value); }
        get { return (bool)GetValue(IsToggledProperty); }
    }

    protected override void OnParentSet()
    {
        base.OnParentSet();
        VisualStateManager.GoToState(this, "ToggledOff");
    }

    static void OnIsToggledChanged(BindableObject bindable, object oldValue, object newValue)
    {
        ToggleButton toggleButton = (ToggleButton)bindable;
        bool isToggled = (bool)newValue;

        // Fire event
        toggleButton.Toggled?.Invoke(toggleButton, new ToggledEventArgs(isToggled));

        // Set the visual state
        VisualStateManager.GoToState(toggleButton, isToggled ? "ToggledOn" : "ToggledOff");
    }
}
```

`ToggleButton` 构造函数附加到一个处理程序 `Clicked` 事件，以便它可以更改的值 `IsToggled` 属性。 `OnIsToggledChanged` 方法将触发 `Toggled` 事件。

最后一行 `OnIsToggledChanged` 方法调用静态 `VisualStateManager.GoToState` 方法具有两个文本字符串"ToggledOn"和"ToggledOff"。你可以阅读有关此方法以及如何在应用程序可以响应到文章中的可视状态 **Xamarin.Forms 视觉状态管理器**。

因为 `ToggleButton` 调用 `VisualStateManager.GoToState`，此类本身不需要包括任何其他工具来更改按钮的外观基于其 `IsToggled` 状态。负责承载 XAML `ToggleButton`。

切换按钮演示页包含两个实例 `ToggleButton`，其中包括设置的视觉状态管理器标记 `Text`， `BackgroundColor`，和 `TextColor` 基于可视状态的按钮：

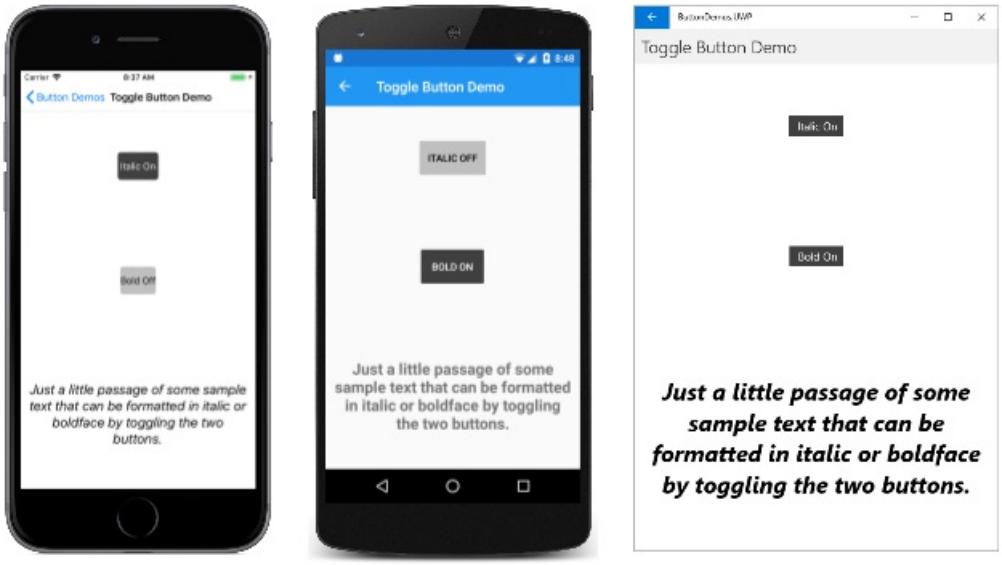
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:ButtonDemos"
             x:Class="ButtonDemos.ToggleButtonDemoPage"
             Title="Toggle Button Demo">

    <ContentPage.Resources>
        <Style TargetType="local:ToggleButton">
            <Setter Property="VerticalOptions" Value="CenterAndExpand" />
            <Setter Property="HorizontalOptions" Value="Center" />
        </Style>
    </ContentPage.Resources>
```

```
    <StackLayout Padding="10, 0">
        <local:ToggleButton Toggled="OnItalicButtonToggled">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ToggleStates">
                    <VisualState Name="ToggledOff">
                        <VisualState.Setters>
                            <Setter Property="Text" Value="Italic Off" />
                            <Setter Property="BackgroundColor" Value="#C0C0C0" />
                            <Setter Property="TextColor" Value="Black" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="ToggledOn">
                        <VisualState.Setters>
                            <Setter Property="Text" Value=" Italic On " />
                            <Setter Property="BackgroundColor" Value="#404040" />
                            <Setter Property="TextColor" Value="White" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </local:ToggleButton>

        <local:ToggleButton Toggled="OnBoldButtonToggled">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ToggleStates">
                    <VisualState Name="ToggledOff">
                        <VisualState.Setters>
                            <Setter Property="Text" Value="Bold Off" />
                            <Setter Property="BackgroundColor" Value="#C0C0C0" />
                            <Setter Property="TextColor" Value="Black" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="ToggledOn">
                        <VisualState.Setters>
                            <Setter Property="Text" Value=" Bold On " />
                            <Setter Property="BackgroundColor" Value="#404040" />
                            <Setter Property="TextColor" Value="White" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </local:ToggleButton>

        <Label x:Name="label"
               Text="Just a little passage of some sample text that can be formatted in italic or boldface by
toggling the two buttons."
               FontSize="Large"
               HorizontalTextAlignment="Center"
               VerticalOptions="CenterAndExpand" />

    </StackLayout>
</ContentPage>
```

`Toggled`事件处理程序是在代码隐藏文件中。他们负责设置`FontAttributes`属性的`Label`根据按钮的状态：

```
public partial class ToggleButtonDemoPage : ContentPage
{
    public ToggleButtonDemoPage ()
    {
        InitializeComponent ();
    }

    void OnItalicButtonToggled(object sender, ToggledEventArgs args)
    {
        if (args.Value)
        {
            label.FontAttributes |= FontAttributes.Italic;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Italic;
        }
    }

    void OnBoldButtonToggled(object sender, ToggledEventArgs args)
    {
        if (args.Value)
        {
            label.FontAttributes |= FontAttributes.Bold;
        }
        else
        {
            label.FontAttributes &= ~FontAttributes.Bold;
        }
    }
}
```

下面是在 iOS、Android 和 UWP 上运行的程序：



使用位图按钮

`Button` 类定义 `Image` 属性，允许用户上显示的位图图像 `Button`，单独使用或与文本结合使用。此外可以指定的文本和图像的排列方式。

`Image` 属性属于类型 `FileImageSource`，这意味着位图，必须存储为资源中单个平台项目和.NET Standard 库项目中不存在。

Xamarin.Forms 支持每个平台允许图像存储在不同的像素的各种设备上运行应用程序可能的解决方法的多个大小。这些是名为多个位图或将其存储在为设备的视频，操作系统可以选取最佳匹配项的方式显示分辨率。

对于在位图 `Button`，最佳大小通常是 32 位和 64 个与设备无关单位之间，具体取决于如何大型希望其成为。在此示例中使用的映像为基础的大小为 48 个与设备无关单位。

在 iOS 项目中，资源文件夹包含三种大小的此映像：

- 存储为一个 48 像素的方形位图 **/Resources/MonkeyFace.png**
- 存储为 96 像素正方形位图 **/Resource/MonkeyFace@2x.png**
- 存储为一个 144 像素的方形位图 **/Resource/MonkeyFace@3x.png**

提供给所有三个位图**生成操作**的**BundleResource**。

对于 Android 项目中，所有的位图具有相同的名称，但是它们的不同子文件夹中存储资源文件夹：

- 存储为 72 像素正方形位图 **/Resources/drawable-hdpi/MonkeyFace.png**
- 存储为 96 像素正方形位图 **/Resources/drawable-xhdpi/MonkeyFace.png**
- 存储为一个 144 像素的方形位图 **/Resources/drawable-xxhdpi/MonkeyFace.png**
- 存储为 192 像素正方形位图 **/Resources/drawable-xxxhdpi/MonkeyFace.png**

这些已为其授予**生成操作**的**AndroidResource**。

在 UWP 项目中，位图可以存储任意位置在项目中，但它们通常存储在自定义文件夹中或资产现有文件夹。UWP 项目包含这些位图：

- 存储为一个 48 像素的方形位图 **/Assets/MonkeyFace.scale-100.png**
- 存储为 96 像素正方形位图 **/Assets/MonkeyFace.scale-200.png**
- 存储为 192 像素正方形位图 **/Assets/MonkeyFace.scale-400.png**

他们所有给定**生成操作**的**内容**。

您可以指定如何 `Text` 并 `Image` 属性上排列 `Button` 使用 `ContentLayout` 属性 `Button`。此属性属于类型 `ButtonContentLayout`，这是在嵌入的类 `Button`。构造函数有两个参数：

- 成员 `ImagePosition` 枚举：`Left`，`Top`，`Right`，或 `Bottom`，该值指示位图相对于文本的显示方式。
- 一个 `double` 位图和文本之间的间距的值。

默认值为 `Left` 和 10 个单位。两个只读属性 `ButtonContentLayout` 名为 `Position` 并 `Spacing` 提供这些属性的值。

在代码中，可以创建 `Button` 并设置 `ContentLayout` 如下属性：

```
Button button = new Button
{
    Text = "button text",
    Image = new FileImageSource
    {
        File = "image filename"
    },
    ContentLayout = new Button.ButtonContentLayout(Button.ButtonContentLayout.ImagePosition.Right, 20)
};
```

在 XAML，您需要指定仅枚举成员或间距，或同时按任意顺序以逗号分隔：

```
<Button Text="button text"
        Image="image filename"
        ContentLayout="Right, 20" />
```

图**像按**钮**演示**页上使用 `OnPlatform` 若要指定不同的文件名在 iOS、Android 和 UWP 位图文件。如果你想要用于所有三个平台使用相同的文件名和避免使用 `OnPlatform`，都需要在项目的根目录中存储的 UWP 位图。

第一个 `Button` 上图像按钮演示页上设置 `Image` 属性但不是 `Text` 属性：

```
<Button>
    <Button.Image>
        <OnPlatform x:TypeArguments="FileImageSource">
            <On Platform="iOS, Android" Value="MonkeyFace.png" />
            <On Platform="UWP" Value="Assets/MonkeyFace.png" />
        </OnPlatform>
    </Button.Image>
</Button>
```

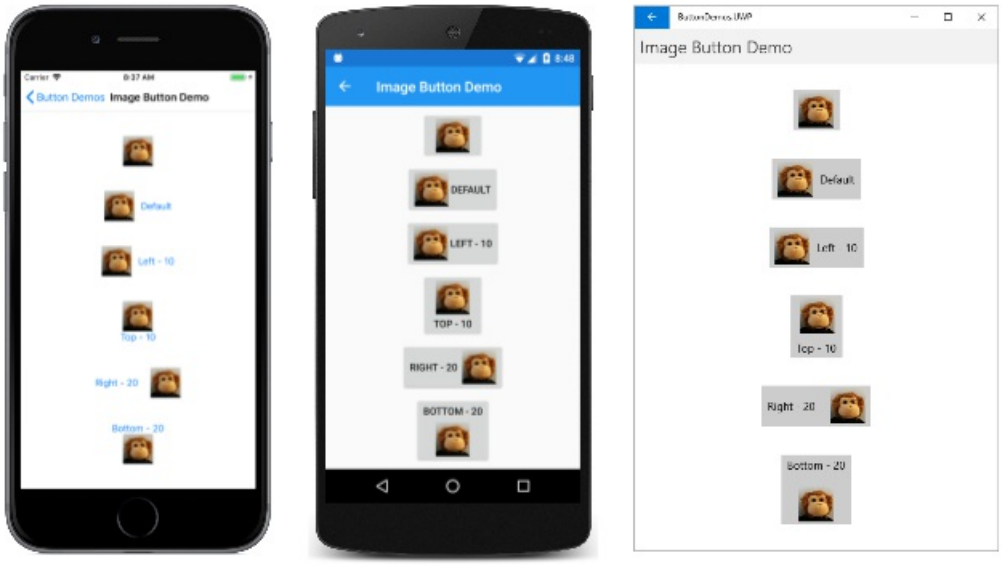
如果该项目的根目录中存储的 UWP 位图，此标记可以大大简化了：

```
<Button Image="MonkeyFace.png" />
```

若要避免重复标记中的大量**ImageButtonDemo.xaml**文件，隐式 `Style` 还定义以设置 `Image` 属性。这 `Style` 自动应用于其他五个 `Button` 元素。下面是完整的 XAML 文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="ButtonDemos.ImageButtonDemoPage">

    <FlexLayout Direction="Column"
                JustifyContent="SpaceEvenly"
                AlignItems="Center">

        <FlexLayout.Resources>
            <Style TargetType="Button">
                <Setter Property="Image">
                    <OnPlatform x:TypeArguments="FileImageSource">
                        <On Platform="iOS, Android" Value="MonkeyFace.png" />
                        <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                    </OnPlatform>
                </Setter>
            </Style>
        </FlexLayout.Resources>

        <Button>
            <Button.Image>
                <OnPlatform x:TypeArguments="FileImageSource">
                    <On Platform="iOS, Android" Value="MonkeyFace.png" />
                    <On Platform="UWP" Value="Assets/MonkeyFace.png" />
                </OnPlatform>
            </Button.Image>
        </Button>

        <Button Text="Default" />

        <Button Text="Left - 10"
                ContentLayout="Left, 10" />

        <Button Text="Top - 10"
                ContentLayout="Top, 10" />

        <Button Text="Right - 20"
                ContentLayout="Right, 20" />

        <Button Text="Bottom - 20"
                ContentLayout="Bottom, 20" />
    </FlexLayout>
</ContentPage>
```

最后一个四 `Button` 元素使使用 `ContentLayout` 属性指定的位置和间距的文本和位图：



现在，已了解你可以处理的各种方法 `Button` 事件和更改 `Button` 外观。

相关链接

- ButtonDemos 示例
- 按钮 API

在 Xamarin.Forms 中的颜色

2018/10/26 • • Edit Online

Xamarin.Forms 提供了灵活的跨平台颜色类。

本文介绍了各种方法 `Color` 类可用于在 Xamarin.Forms 中。

`Color` 类提供了多种方法来生成颜色实例

- **已命名的颜色**-一系列常见已命名的颜色，包括 `Red`， `Green`，和 `Blue`。
- **FromHex**的字符串值类似于在 HTML 中，例如"00FF00"所使用的语法。Alpha 是可根据需要指定为第一对字符 ("CC00FF00")。
- **FromHsla** -色调、饱和度和亮度 `double` 值，其中可选 alpha 值 (介于 0.0 到 1.0)。
- **FromRgb** -红色、绿色和蓝色 `int` 值 (0-255)。
- **FromRgba** -红色、绿色、蓝色和 alpha `int` 值 (0-255)。
- **FromUint** -设置单个 `double` 值，该值表示**argb**。

下面是一些示例颜色，分配给 `BackgroundColor` 的某些标签使用允许的语法的不同变体：

```
var red    = new Label { Text = "Red",    BackgroundColor = Color.Red };
var orange = new Label { Text = "Orange",BackgroundColor = Color.FromHex("FF6A00") };
var yellow = new Label { Text = "Yellow",BackgroundColor = Color.FromHsla(0.167, 1.0, 0.5, 1.0) };
var green  = new Label { Text = "Green", BackgroundColor = Color.FromRgb (38, 127, 0) };
var blue   = new Label { Text = "Blue",  BackgroundColor = Color.FromRgba(0, 38, 255, 255) };
var indigo = new Label { Text = "Indigo",BackgroundColor = Color.FromRgb (0, 72, 255) };
var violet = new Label { Text = "Violet",BackgroundColor = Color.FromHsla(0.82, 1, 0.25, 1) };

var transparent = new Label { Text = "Transparent",BackgroundColor = Color.Transparent };
var @default = new Label    { Text = "Default",    BackgroundColor = Color.Default };
var accent = new Label      { Text = "Accent",     BackgroundColor = Color.Accent };
```

以下每个平台上显示这些颜色。请注意，最终颜色- `Accent` -是适用于 iOS 和 Android; blue-ish 颜色由 Xamarin.Forms 定义此值。

Color.Default

使用 `Default` 设置（或重新设置）颜色值更改为平台默认值（了解，这表示每个属性的每个平台上不同的基础颜色）。

开发人员可以使用此值设置 `Color` 属性但应不查询其组件 RGB 值（它们所有设置为-1）此实例。

Color.Transparent

设置要清除的颜色。

Color.Accent

IOS 和 Android 上此实例设置为颜色的对比色的默认背景上可见，但不是默认文本颜色相同。

其他方法

`Color` 实例包含可用于创建新颜色的其他方法：

- **AddLuminosity** -通过提供增量修改亮度返回一种新颜色。
- **WithHue** -返回一种新颜色、色调替换为提供的值。
- **WithLuminosity** -返回一种新颜色，亮度替换为提供的值。
- **WithSaturation** -返回一种新颜色，饱和度替换为提供的值。
- **MultiplyAlpha** -通过修改的 alpha，乘以所提供的 alpha 值返回新的颜色。

隐式转换

之间的隐式转换 `Xamarin.Forms.Color` 和 `System.Drawing.Color` 可以执行类型：

```
Xamarin.Forms.Color xfColor = Xamarin.Forms.Color.FromRgb(0, 72, 255);
System.Drawing.Color sdColor = System.Drawing.Color.FromArgb(38, 127, 0);

// Implicity convert from a Xamarin.Forms.Color to a System.Drawing.Color
System.Drawing.Color sdColor2 = xfColor;

// Implicitly convert from a System.Drawing.Color to a Xamarin.Forms.Color
Xamarin.Forms.Color xfColor2 = sdColor;
```

Device.RuntimePlatform

此代码片段使用 `Device.RuntimePlatform` 属性可以有选择性地设置颜色的 `ActivityIndicator`：

```
ActivityIndicator activityIndicator = new ActivityIndicator
{
    Color = Device.RuntimePlatform == Device.iOS ? Color.Black : Color.Default,
    IsRunning = true
};
```

从 XAML 使用

颜色也可以轻松地引用中使用定义的颜色名称或如下所示的十六进制表示形式的 XAML 中：

```
<Label Text="Sea color" BackgroundColor="Aqua" />
<Label Text="RGB" BackgroundColor="#00FF00" />
<Label Text="Alpha plus RGB" BackgroundColor="#CC00FF00" />
<Label Text="Tiny RGB" BackgroundColor="#0F0" />
<Label Text="Tiny Alpha plus RGB" BackgroundColor="#C0F0" />
```

> **NOTE**
>
> 当使用 XAML 编译，颜色名称是不区分大小写，并因此可以编写以小写形式。有关 XAML 编译的详细信息，请参阅XAML 编译。

总结

Xamarin.Forms `Color` 类用于创建识别平台的颜色的引用。可在共享的代码和 XAML。

相关链接

- ColorsSample
- 可绑定选取器（示例）

控件引用

2018/7/13 • • Edit Online

用于构造 Xamarin.Forms 应用程序的所有可视元素的说明。

Xamarin.Forms 应用程序的可视界面将映射到每个目标平台的本机控件的对象的构造。这样，特定于平台的应用程序的 iOS、Android 和通用 Windows 平台，以使用 Xamarin.Forms 中包含的代码.NET 标准库或共享项目。

用于创建 Xamarin.Forms 应用程序的用户界面的四个主要控件组是这些四篇文章中所示：

- 页
- **布局**
- 视图
- **单元格**

Xamarin.Forms 页面通常占据整个屏幕。页通常包含一种布局，其中包含视图和可能是其他布局。单元格是使用门户中的专用的组件 `TableView` 并 `ListView` 。

在上的四个文章页面，**布局**，视图，并单**元格**，（如果存在），其中包含指向其 API 文档、文章描述了其使用（如果存在）和一个或多个示例程序介绍了每种类型的控件。显示来自的页面的屏幕截图还附带有每种类型的控件 **FormsGallery** 设备 iOS、Android 和 UWP 上运行的示例。下面的每个屏幕快照是 C# 页上，等效 XAML 页面的源代码的链接和（在适当的时候）XAML 页的 C# 代码隐藏文件。

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms FormsGallery 示例
- API 文档

Xamarin.Forms 页面

2018/7/13 • • Edit Online

Xamarin.Forms 页呈现跨平台移动应用程序屏幕。

如下所述的所有页类型都派生自 Xamarin.Forms `Page` 类。这些可视元素占用全部或大部分屏幕。一个 `Page` 对象表示 `ViewController` 在 iOS 和 `Page` 通用 Windows 平台中。在 Android 上，每个页面将占用所示的屏幕 `Activity`，但按 Xamarin.Forms 页*不* `Activity` 对象。

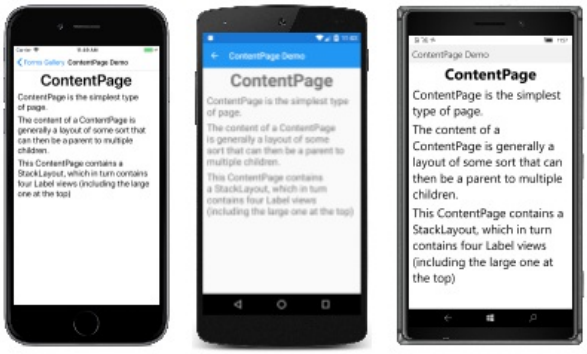
页数

Xamarin.Forms 支持以下的页类型：

ContentPage

`ContentPage` 是页面的最简单且最常见的类型。设置 `Content` 属性设置为单个 `View` 对象，它是最常 `Layout` 如 `StackLayout`， `Grid` ，或 `ScrollView` 。

API 文档



此页的 C# 代码 / XAML 页

MasterDetailPage

一个 `MasterDetailPage` 管理信息的两个窗格。设置 `Master` 属性设置为一个页面，通常显示为列表或菜单。设置 `Detail` 到一个页面，显示来自母版页的选定的项的属性。 `IsPresented` 属性控制 master 或详细信息页上是否可见。

API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页面与代码隐藏

NavigationPage

`NavigationPage` 管理在使用基于堆栈的体系结构的其他页面之间导航。主页上的实例时应用程序中使用页面导航，应传递给构造函数的 `NavigationPage` 对象。

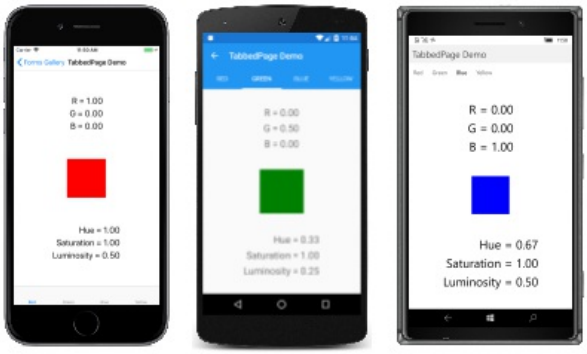
API 文档 / 指南 / 示例 1，2，和3

此页的 C# 代码 / XAML 页面与代码背后 =

TabbedPage

`TabbedPage` 派生自抽象 `MultiPage` 类，并允许使用选项卡页的子之间导航。设置 `Children` 属性设置为一系列页或一组 `ItemsSource` 属性设置为数据对象的集合和 `ItemTemplate` 属性设置为 `DataTemplate` 描述每个对象的直观表示方式。

API 文档 / 指南 / 示例 1和2



此页的 C# 代码 / XAML 页

CarouselPage

`CarouselPage` 派生自抽象 `MultiPage` 类，并允许在子之间的导航手指轻扫中翻页。设置 `Children` 属性设置为一系列 `ContentPage` 对象或一组 `ItemsSource` 属性设置为数据对象的集合和 `ItemTemplate` 属性设置为 `DataTemplate` 描述每个对象的直观表示方式。

API 文档 / 指南 / 示例 1和2

此页的 C# 代码 / XAML 页

TemplatedPage

`TemplatedPage` 显示使用控件模板的全屏幕内容，并为类的基类 `ContentPage` 。 API 文档 / 指南	

相关链接

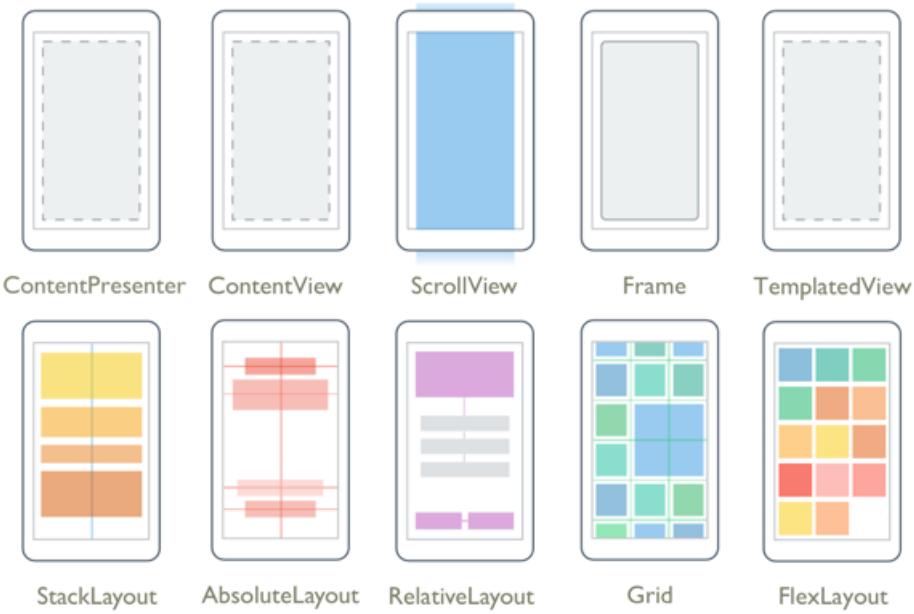
- Xamarin.Forms 简介
- Xamarin.Forms FormsGallery 示例
- Xamarin.Forms 示例
- Xamarin.Forms API 文档

Xamarin.Forms 布局

2018/7/13 • • Edit Online

Xamarin.Forms 布局用于将用户界面控件组合到可视结构。

`Layout` 并 `Layout<T>` 在 Xamarin.Forms 中的类是专用的视图作为视图和其他布局容器的子类型。`Layout` 类本身派生自 `View` 。一个 `Layout` 派生类通常包含在 Xamarin.Forms 应用程序中设置的位置和大小的子元素的逻辑。



派生的类 `Layout` 可以分为两类:

具有单一内容布局

这些类派生自 `Layout` ，用于定义 `Padding` 并 `IsClippedToBounds` 属性。

ContentView

`ContentView` 包含与设置的一个子 `Content` 属性。 `Content` 属性可以设置为任意 `View` 派生类，包括其他 `Layout` 派生类。 `ContentView` 主要用作结构化元素并用作基类 `Frame` 。 API 文档	 此页的 C# 代码 / XAML 页

Frame

`Frame` 类派生自 `ContentView` 并显示其子级矩形边框。 `Frame` 具有默认值 `Padding` 值为 20，还定义 `OutlineColor` 、 `CornerRadius` ，并 `HasShadow` 属性。

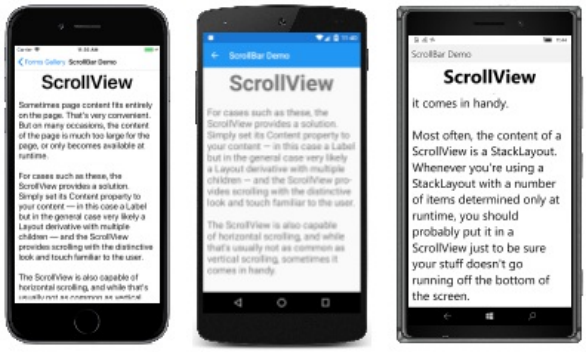
API 文档

此页的 C# 代码 / XAML 页

ScrollView

`ScrollView` 它能够滚动其内容。设置 `Content` 属性设置为要在屏幕上显示的视图或布局太大。(的内容 `ScrollView` 非常通常 `StackLayout` 。)设置 `Orientation` 属性以指示是否滚动应为垂直、水平或两者。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

TemplatedView

`TemplatedView` 显示使用控件模板的内容，并为类的基类 `ContentView` 。

API 文档 / 指南

ContentPresenter

`ContentPresenter` 是模板化中使用的视图的布局管理器 `ControlTemplate` 标记呈现内容的显示位置。

API 文档 / 指南

使用多个子级的布局

这些类派生自 `Layout<View>` 。

StackLayout

`StackLayout` 将子元素定位在水平或垂直方向上基于堆栈 `Orientation` 属性。 `Spacing` 属性控制子级之间的间距和具有默认值为 6。

API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页

Grid

`Grid` 在行和列的网格中定位其子元素。使用表示子表位置附加属性 `Row` 、 `Column` 、 `RowSpan` ，并 `ColumnSpan` 。

API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页

AbsoluteLayout

`AbsoluteLayout` 相对于其父级的特定位置定位子元素。使用表示子表位置附加属性 `LayoutBounds` 并 `LayoutFlags` 。`AbsoluteLayout` 可用于对视图的位置进行动画处理。

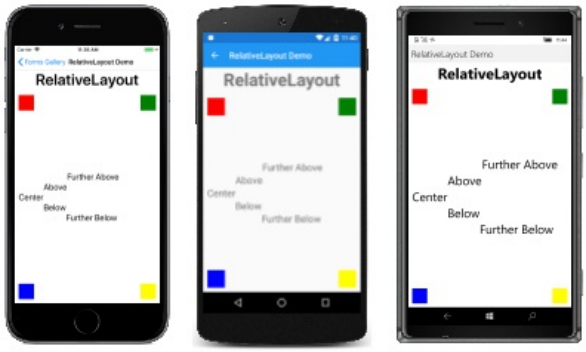
API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页面与代码隐藏

RelativeLayout

`RelativeLayout` 定位子元素相对于 `RelativeLayout` 本身或其同级。使用表示子表位置附加属性设置为的类型的对象 `Constraint` 并 `BoundsConstraint` 。

API 文档 / 指南 / 示例

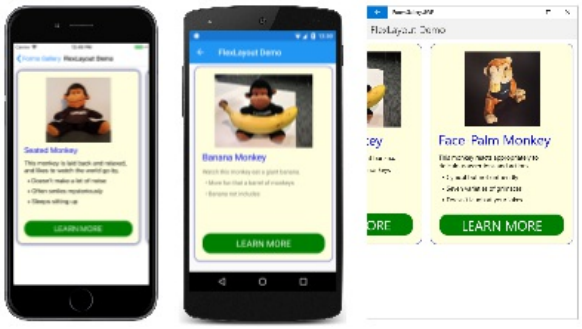


此页的 C# 代码 / XAML 页

FlexLayout

`FlexLayout` 根据 CSS灵活框布局模块，通常称为_flex 布局_或_弹性框_。`FlexLayout` 定义六种可绑定属性和允许儿童堆积或包装了许多的对齐方式和方向选项的五个附加的可绑定属性。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms FormsGallery 示例
- Xamarin.Forms 示例
- Xamarin.Forms API 文档

Xamarin.Forms 视图

2018/10/26 • • Edit Online

Xamarin.Forms 视图是跨平台移动用户界面的构建基块。

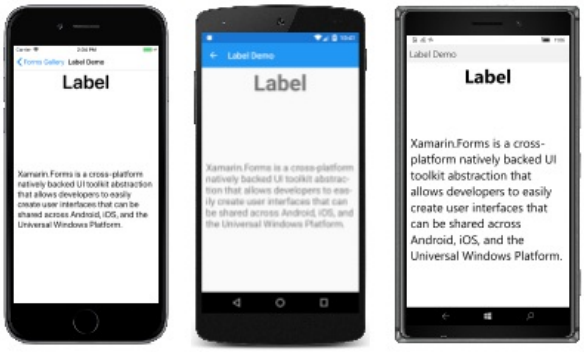
视图是用户界面对象，如标签、按钮和滑块，通常称为分别*控件*或*小组件*其他图形的编程环境中。支持所有派生自 Xamarin.Forms 的视图 `View` 类。它们可以划分为几个类别：

演示文稿的视图

Label

`Label` 显示单行文本字符串或多行文本，使用常量或变量格式设置块。设置 `Text` 属性设置为常量的格式设置，或一组字符串 `FormattedText` 属性设置为 `FormattedString` 变量的对象格式设置。

API 文档 / 指南 / 示例

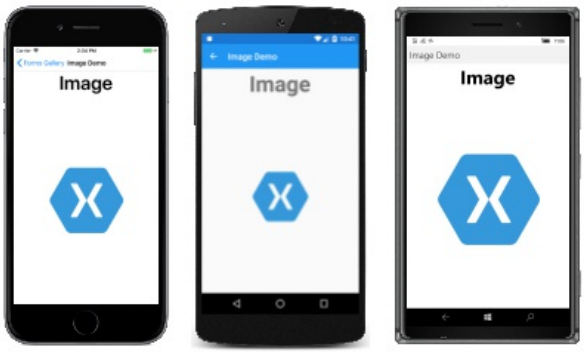


此页的 C# 代码 / XAML 页

图像

`Image` 显示位图。可以通过作为资源嵌入常见的项目或平台项目中或使用.NET 创建的 Web 下载位图 `Stream` 对象。

API 文档 / 指南 / 示例

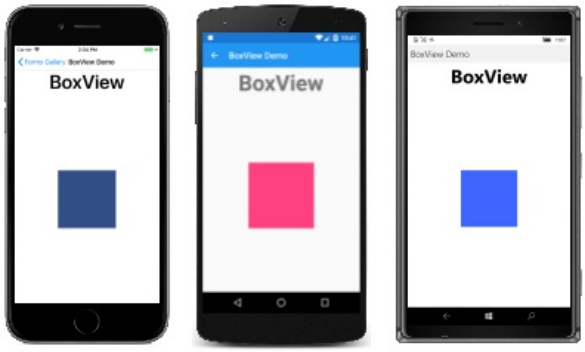


此页的 C# 代码 / XAML 页

字数

`BoxView` 显示实心矩形通过着色 `Color` 属性。`BoxView` 有 40 x 40 个的默认大小请求。其他大小，将分配 `WidthRequest` 并 `HeightRequest` 属性。

API 文档 / 指南 / 示例 1，2，3，4，5，和6

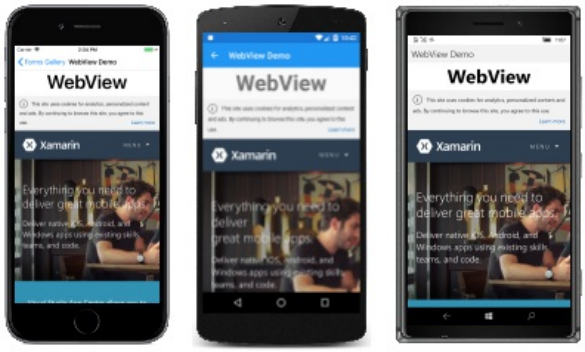


此页的 C# 代码 / XAML 页

Web 视图

`WebView` 显示网页或 HTML 内容，根据是否 `Source` 属性设置为 `UriWebViewSource` 或者 `HtmlWebViewSource` 对象。

API 文档 / 指南 / 示例 1和2



此页的 C# 代码 / XAML 页

OpenGLView

`OpenGLView` 将 OpenGL 图形显示的 iOS 和 Android 项目中。没有适用于通用 Windows 平台支持。IOS 和 Android 项目需要引用**OpenTK 1.0**程序集或**OpenTK**版本 1.0.0.0 的程序集。`OpenGLView` 更轻松地在共享项目; 中使用如果使用.NET Standard 库中，然后一个依赖关系服务还需要（如示例代码中所示）。

这是唯一的图形工具，内置于 Xamarin.Forms 中，但 Xamarin.Forms 应用程序还可以使用图形呈现 `CocosSharp` , `SkiaSharp` , 或 `UrhoSharp` .

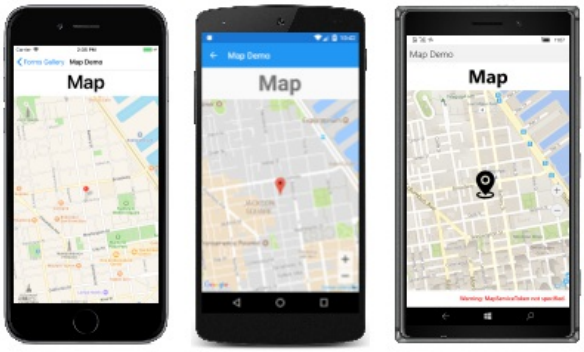
API 文档

此页的 C# 代码 / XAML 页面与代码隐藏

映射

`Map` 显示了一个映射。**Xamarin.Forms.Maps必须安装** Nuget 包。Android 和通用 Windows 平台需要映射授权密钥。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

启动命令的视图

Button

`Button` 是一个矩形对象，显示的文本，并会激发 `Clicked` 事件已按下时。

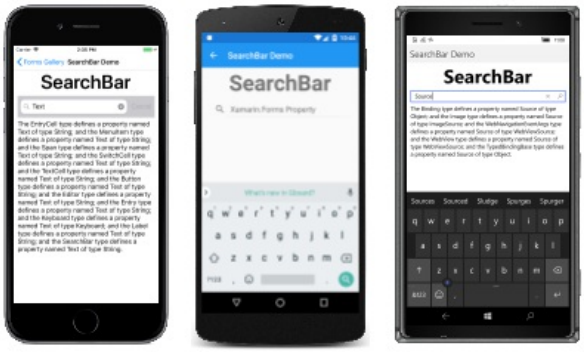
API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页面与代码隐藏

SearchBar

`SearchBar` 显示一个区域供用户键入文本字符串，和一个按钮（或键盘键），用于通知应用程序执行的搜索。`Text` 属性提供对文本、访问和 `SearchButtonPressed` 事件指示按下按钮。

API 文档



此页的 C# 代码 / XAML 页面与代码隐藏

对于设置值的视图

Slider

`Slider` 允许用户选择 `double` 连续范围与指定值 `Minimum` 并 `Maximum` 属性。

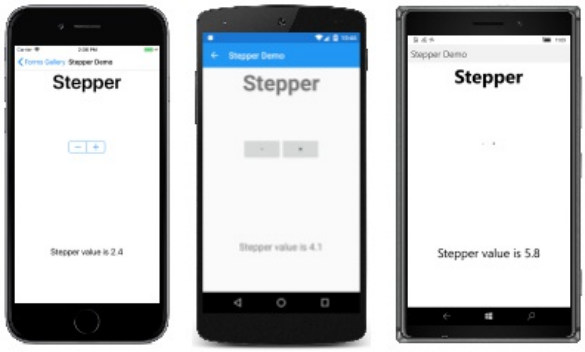
API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页

分档器

`Stepper` 允许用户选择 `double` 通过一系列的增量值使用指定的值 `Minimum` ， `Maximum` ，并且 `Increment` 属性。

API 文档 / 指南 / 示例

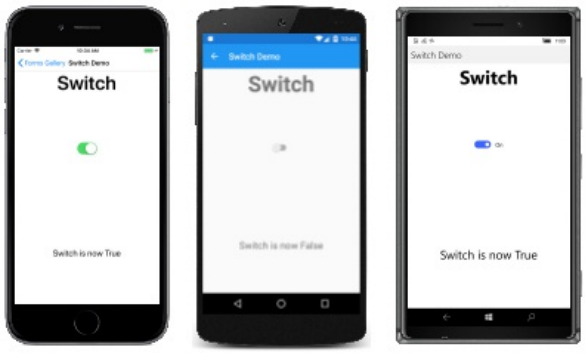


此页的 C# 代码 / XAML 页

开关

`Switch` 采用打开/关闭开关以允许用户选择一个布尔值的形式。 `IsToggled` 属性为 state 的开关，和 `Toggled` 状态更改时触发事件。

API 文档

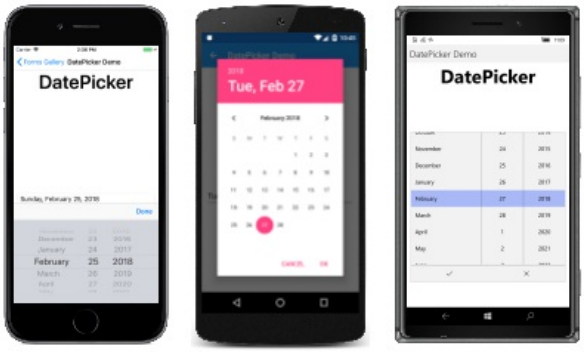


此页的 C# 代码 / XAML 页

DatePicker

`DatePicker` 允许用户与平台日期选取器选择一个日期。设置与允许的日期范围 `MinimumDate` 并 `MaximumDate` 属性。`Date` 属性是所选的日期，和 `DateSelected` 该属性发生更改时触发事件。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

TimePicker

`TimePicker` 允许用户与平台时间选取器选择的时间。`Time` 属性是所选的时间。应用程序可以监视变化 `Time` 通过安装的处理程序的属性 `PropertyChanged` 事件。

API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页

编辑文本的视图

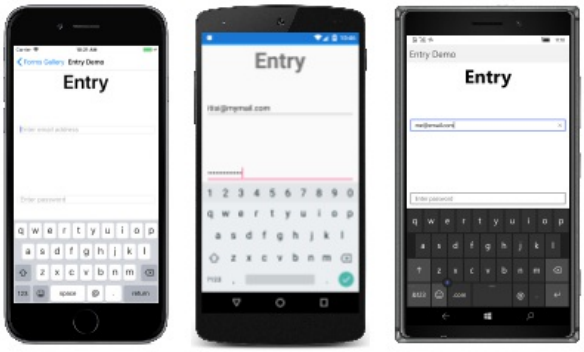
这两个类派生自 `InputView` 类，该类定义 `Keyboard` 属性。

条目

`Entry` 允许用户输入和编辑单个文本行。该文本是可用作 `Text` 属性，和 `TextChanged` 并 `Completed` 的事件触发时的文本更改或用户通过点击 enter 键，向发出完成信号。

使用 `Editor` 用于输入和编辑多行文本。

API 文档 / 指南 / 示例



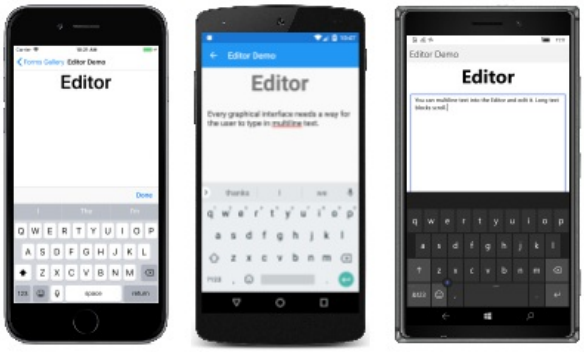
此页的 C# 代码 / XAML 页

编辑器

`Editor` 允许用户输入和编辑多行文本。该文本是可用作 `Text` 属性，和 `TextChanged` 并 `Completed` 的事件触发时的文本更改或用户完成向发出信号。

使用 `Entry` 输入和编辑单个文本行的视图。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

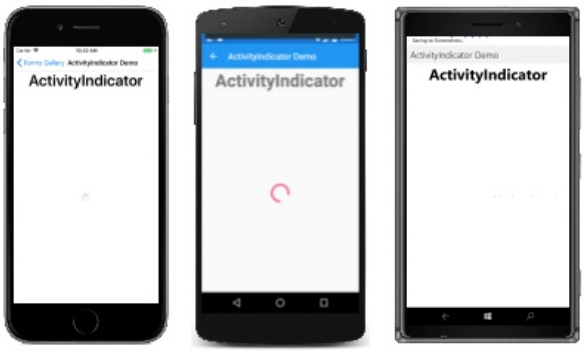
若要指示活动的视图

ActivityIndicator

`ActivityIndicator` 使用动画来显示该应用程序所从事的耗时较长的活动而不授予任何可能的进度。`IsRunning` 属性控制动画。

如果已知的活动的进度，使用 `ProgressBar` 相反。

API 文档



此页的 C# 代码 / XAML 页

ProgressBar

`ProgressBar` 使用动画来显示，该应用程序时通过耗时较长的活动的进展情况。设置 `Progress` 属性设置为值介于 0 和 1，表示进度。

如果不知道该活动的进度，使用 `ActivityIndicator` 相反。

API 文档

此页的 C# 代码 / XAML 页面与代码隐藏

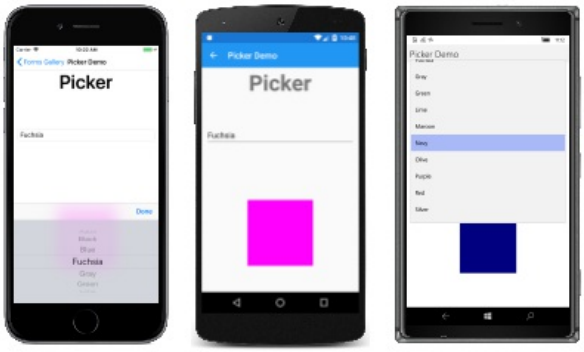
显示集合的视图

选取器

`Picker` 显示选定的项，从列表中的文本字符串，并允许您选择该项，点击该视图时。设置 `Items` 属性设置为一组字符串，则 `ItemsSource` 属性设置为对象的集合。`SelectedIndexChanged` 当选中某个项时触发事件。

`Picker` 仅选择此项时显示的项的列表。使用 `ListView` 或 `TableView` 有关页保留的可滚动列表。

API 文档 / 指南 / 示例

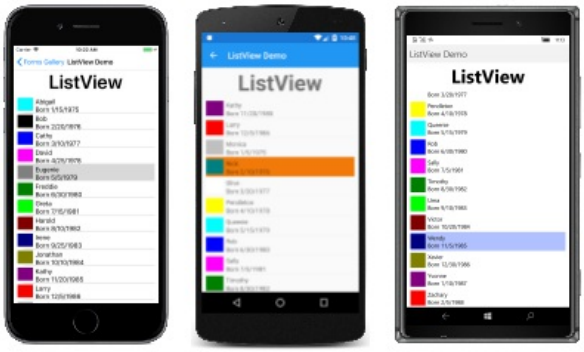


此页的 C# 代码 / XAML 页面与代码隐藏

ListView

`ListView` 派生自 `ItemsView[Cell]`，并显示可选择数据项的可滚动列表。设置 `ItemsSource` 属性设置为一系列对象，并设置 `ItemTemplate` 属性设置为 `DataTemplate` 描述如何将项目对象要设置格式。`ItemSelected` 事件发出信号，已做出选择，这是可用作 `SelectedItem` 属性。

API 文档 / 指南 / 示例



此页的 C# 代码 / XAML 页

TableView

`TableView` 显示类型的行的列表 `Cell` 可选标头和小标题。设置 `Root` 类型的对象的属性 `TableRoot`，并添加 `TableSection` 对象的 `TableRoot`。每个 `TableSection` 是一系列 `Cell` 对象。

API 文档 / 指南 / 示例

此页的 C# 代码 / XAML 页

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms FormsGallery 示例
- Xamarin.Forms 示例
- Xamarin.Forms API 文档

Xamarin.Forms 单元格

2018/11/13 • • Edit Online

Xamarin.Forms 单元格可以添加到 Listview 和 TableViews。

一个*单元格*是一个专用的元素，用于在表中的项，描述了应如何呈现列表中的每个项。`Cell` 类派生自 `Element`，从其 `VisualElement` 也派生。单元格本身不是可视元素;而是用于创建可视元素的模板。

`Cell` 使用专用于 `ListView` 并 `TableView` 控件。若要了解如何使用和自定义单元格，请参阅 `ListView` 并 `TableView` 文档。

单元格

Xamarin.Forms 支持以下的单元格类型：

TextCell

一个 `TextCell` 显示一个或两个文本字符串。设置 `Text` 属性和（可选） `Detail` 属性设置为这些文本字符串。

API 文档 / 指南

此页的 C# 代码 / XAML 页

ImageCell

`ImageCell` 显示相同的信息 `TextCell` 包括与设置一个位图，但 `Source` 属性。

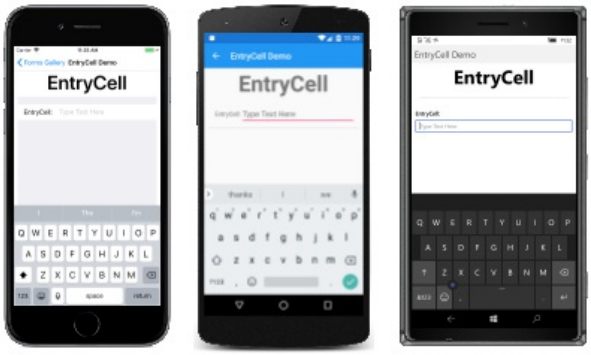
API 文档 / 指南

此页的 C# 代码 / XAML 页

SwitchCell

`SwitchCell` 包含设置与文本 `Text` '属性和打开/关闭开关与布尔值最初设置 `On` 属性。处理 `OnChanged` 事件时通知 `On` 属性更改。 API 文档 / 指南	 此页的 C# 代码 / XAML 页

EntryCell

`EntryCell` 定义 `Label` 属性，用于标识该单元格和单行中的可编辑文本 `Text` 属性。处理 `Completed` 事件以在用户已完成的文本输入时得到通知。 API 文档 / 指南	 此页的 C# 代码 / XAML 页

相关链接

- Xamarin.Forms 简介
- Xamarin.Forms FormsGallery 示例
- Xamarin.Forms 示例
- Xamarin.Forms API 文档

Xamarin.Forms DataPages

2018/7/12 • • Edit Online

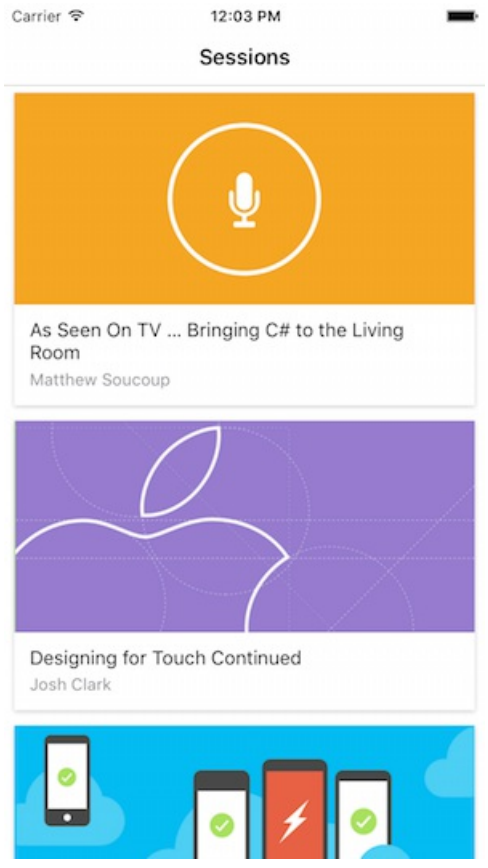
IMPORTANT

需要 DataPages Xamarin.Forms 主题引用来呈现。

Xamarin.Forms DataPages 曾宣布在 Evolve 2016，并可作为客户试用并提供反馈的预览。

DataPages 提供一个 API，用于快速、轻松地将数据源绑定到预建的视图。列表项和详细信息页将自动呈现数据，并可以使用主题自定义。

若要查看发展主题演讲演示的工作原理，请查看入门指南。



介绍

数据源和关联的数据页允许开发人员快速、轻松地使用支持的数据源，并使其与主题中使用的内置基架，UI 可以自定义。

通过包括 DataPages 添加到 Xamarin.Forms 应用程序**Xamarin.Forms.Pages** Nuget 包。

Data Sources

在预览具有可供使用一些预构建的数据源：

- **JsonDataSource**
- **AzureDataSource**（单独的 Nuget）
- **AzureEasyTableDataSource**（单独的 Nuget）

请参阅入门指南示例使用 `JsonDataSource`。

页和控件

以下页面和控件都包括在内，以便允许轻松绑定到提供的数据源：

- **ListDataPage** – 请参阅入门示例。
- **DirectoryPage** – 已启用分组列表。
- **PersonDetailPage** – 单个数据项为特定对象类型（联系人项）自定义的视图。
- **DataView** – 公开从采用一般形式的源数据的视图。
- **CardView** – 样式视图，它包含图像、标题文本和说明文本。
- **HeroImage** – 图像呈现视图。
- **ListItem** – 预构建的具有类似于本机 iOS 和 Android 的列表项的布局视图。

请参阅DataPages 控件参考有关示例。

揭秘

Xamarin.Forms 数据源遵守 `IDataSource` 接口。

Xamarin.Forms 基础结构进行交互与数据源通过以下属性：

- `Data` – 可以显示的数据项的只读列表。
- `IsLoading` – 一个布尔值，该值指示数据是否已加载并可用于呈现。
- `[key]` – 要从中检索元素的索引器。

有两种方法 `MaskKey` 和 `UnmaskKey` 可用来隐藏（或显示）（即数据项目属性 防止它们呈现）。键对应于数据项对象上的已命名的属性。

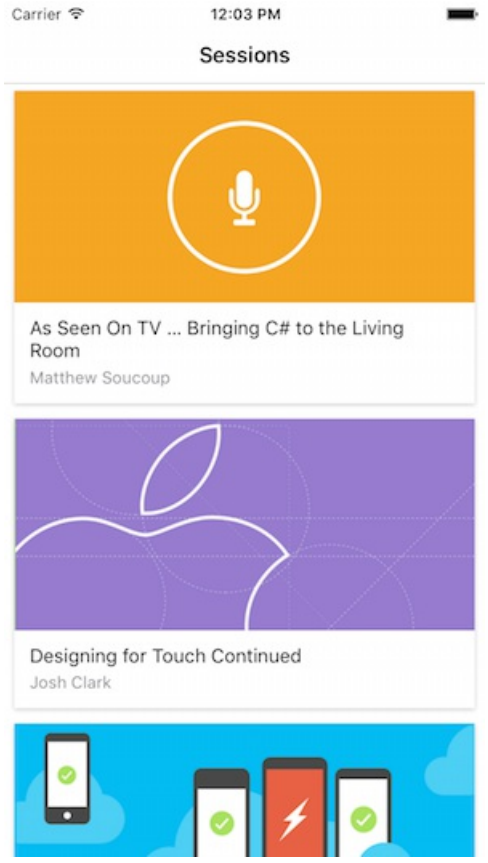
开始使用 DataPages

2018/7/12 • • Edit Online

IMPORTANT

需要 DataPages Xamarin.Forms 主题引用来呈现。

若要开始构建使用 DataPages 预览一个简单的数据驱动页面，请执行以下步骤。在预览中的硬编码样式（"事件"）生成此演示使用仅适用于在代码中特定的 JSON 格式。



1.添加 NuGet 包

将以下 Nuget 包添加到 Xamarin.Forms.NET 标准库和应用程序项目：

- Xamarin.Forms.Pages
- Xamarin.Forms.Theme.Base
- 主题实现（例如 Nuget。Xamarin.Forms.Themes.Light）

2.添加主题引用

在中**App.xaml**文件中，添加一个自定义 `xmlns:mytheme` 主题，并确保主题合并到应用程序的资源字典：

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:mytheme="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Light"
  x:Class="DataPagesDemo.App">
    <Application.Resources>
        <ResourceDictionary MergedWith="mytheme:LightThemeResources" />
    </Application.Resources>
</Application>
```

重要说明：还应遵循的步骤加载程序集（下面）为主题通过将一些样本代码添加到 iOS `AppDelegate` 和 Android `MainActivity`。这将在将来的预览版的版本中得到改进。

3.添加 XAML 页面

将新的 XAML 页面添加到 Xamarin.Forms 应用程序，并*将基类更改*从 `ContentPage` 到 `Xamarin.Forms.Pages.ListDataPage`。这必须在 C# 和 XAML 中进行：

C# 文件

```
public partial class SessionDataPage : Xamarin.Forms.Pages.ListDataPage // was ContentPage
{
  public SessionDataPage ()
  {
    InitializeComponent ();
  }
}
```

XAML 文件

除了更改到的根元素之外 `<p:ListDataPage>` 的自定义命名空间 `xmlns:p` 还必须添加：

```
<?xml version="1.0" encoding="UTF-8"?>
<p:ListDataPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:p="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
             x:Class="DataPagesDemo.SessionDataPage">

    <ContentPage.Content></ContentPage.Content>

</p:ListDataPage>
```

应用程序子类

更改 `App` 类构造函数，以便 `MainPage` 设置为 `NavigationPage` 包含新 `SessionDataPage`。一个导航页*必须*使用。

```
MainPage = new NavigationPage (new SessionDataPage ());
```

3.添加数据源

删除 `Content` 元素并将其替换为 `p:ListDataPage.DataSource` 来填充的数据页。在远程 Json 下面的示例从 URL 是正在加载数据文件。

注意：预览版*要求* `StyleClass` 属性来为数据源提供呈现提示。 `StyleClass="Events"` 指的是在预览中预定义的包含样式的布局*硬编码*以匹配正在使用的 JSON 数据源。

```
<?xml version="1.0" encoding="UTF-8"?>
<p:ListDataPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:p="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
             x:Class="DataPagesDemo.SessionDataPage"
             Title="Sessions" StyleClass="Events">

    <p:ListDataPage.DataSource>
        <p:JsonDataSource Source="http://demo3143189.mockable.io/sessions" />
    </p:ListDataPage.DataSource>

</p:ListDataPage>
```

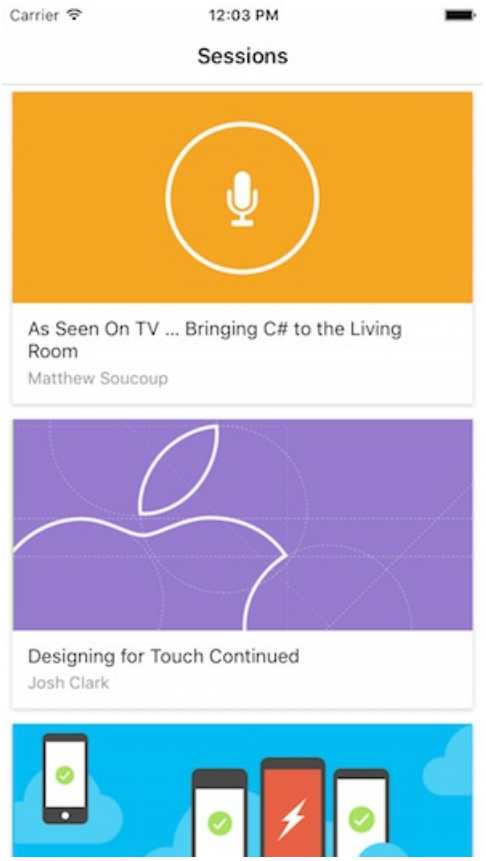
JSON 数据

从 JSON 数据的示例演示源如下所示：

```
[{
  "end": "2016-04-27T18:00:00Z",
  "start": "2016-04-27T17:15:00Z",
  "abstract": "The new Apple TV has been released, and YOU can be one of the first developers to write apps for it. To make things even better, you can build these apps in C#! This session will introduce the basics of how to create a tvOS app with Xamarin, including: differences between tvOS and iOS APIs, TV user interface best practices, responding to user input, as well as the capabilities and limitations of building apps for a television. Grab some popcorn—this is going to be good!",
  "title": "As Seen On TV … Bringing C# to the Living Room",
  "presenter": "Matthew Soucoup",
  "biography": "Matthew is a Xamarin MVP and Certified Xamarin Developer from Madison, WI. He founded his company Code Mill Technologies and started the Madison Mobile .Net Developers Group.  Matt regularly speaks on .Net and Xamarin development at user groups, code camps and conferences throughout the Midwest. Matt gardens hot peppers, rides bikes, and loves Wisconsin micro-brews and cheese.",
  "image": "http://i.imgur.com/ASj60DP.jpg",
  "avatar": "http://i.imgur.com/ASj60DP.jpg",
  "room": "Crick"
}]
```

4.运行 ！

上述步骤应导致工作数据页：



这样做的原因的预建的样式 **"事件"** 浅色主题 Nuget 包中存在并且具有与数据源（例如匹配的样式定义。"title"、"图像"、"表示器"）。

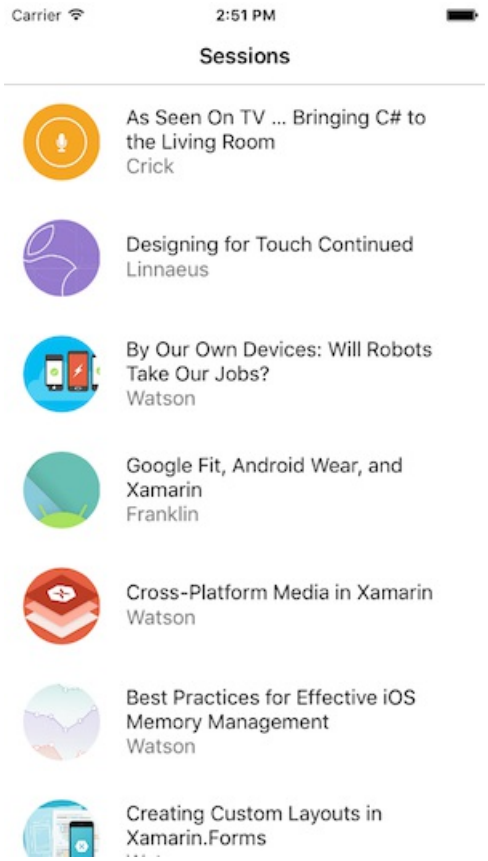
"事件" `StyleClass` 旨在显示 `ListDataPage` 具有一个自定义控件 `CardView` Xamarin.Forms.Pages 中定义的控件。 `CardView` 控件具有三个属性： `ImageSource` ， `Text` ，和 `Detail` 。主题是硬编码要绑定的数据源（从 JSON 文件中）到显示这些属性的三个字段。

5.自定义

可以通过指定的模板并使用数据源绑定中重写继承的样式。以下 XAML 声明每个行中使用新的自定义模板 `ListItemControl` 并 `{p:DataSourceBinding}` 中包含的语法**Xamarin.Forms.Pages** Nuget:

```
<p:ListDataPage.DefaultItemTemplate>
    <DataTemplate>
        <ViewCell>
            <p:ListItemControl
                Title="{p:DataSourceBinding title}"
                Detail="{p:DataSourceBinding room}"
                ImageSource="{p:DataSourceBinding image}"
                DataSource="{Binding Value}"
                HeightRequest="90"
            >
            </p:ListItemControl>
        </ViewCell>
    </DataTemplate>
</p:ListDataPage.DefaultItemTemplate>
```

通过提供 `DataTemplate` 此代码将重写 `StyleClass` ，而是使用的默认布局 `ListItemControl` 。



开发人员喜欢 C# 到 XAML 可以创建数据源绑定过 (请记住包括 `using Xamarin.Forms.Pages;` 语句):

```
SetBinding (TitleProperty, new DataSourceBinding ("title"));
```

它是要从头开始创建主题的更多工作 (请参阅主题指南) 但未来的预览版本将使这更轻松地执行。

疑难解答

无法加载文件或程序集 Xamarin.Forms.Theme.Light 或其某个依赖项

在预览版本中，主题可能不能在运行时加载。添加代码，如下所示在相关项目来修复此错误。

iOS

在中**AppDelegate.cs**添加以下行后 `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.iOS.UnderlineEffect);
```

Android

在中**MainActivity.cs**添加以下行后 `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.Android.UnderlineEffect);
```

相关链接

- DataPagesDemo 示例

DataPages 控件参考

2018/8/6 • • Edit Online

IMPORTANT

需要 DataPages Xamarin.Forms 主题引用来呈现。

Xamarin.Forms DataPages Nuget 包括了一些可以充分利用数据源绑定的控件。

若要在 XAML 中使用这些控件，请确保包含了命名空间，有关示例请参阅 `xmlns:pages` 以下声明：

```
<ContentPage
    xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:pages="clr-namespace:Xamarin.Forms.Pages;assembly=Xamarin.Forms.Pages"
    x:Class="DataPagesDemo.Detail">
```

下面的示例包括 `DynamicResource` 需要工作的项目的资源字典中存在的引用。另外，还有举例说明如何生成自定义控件

内置控件

- HeroImage
- 列表项

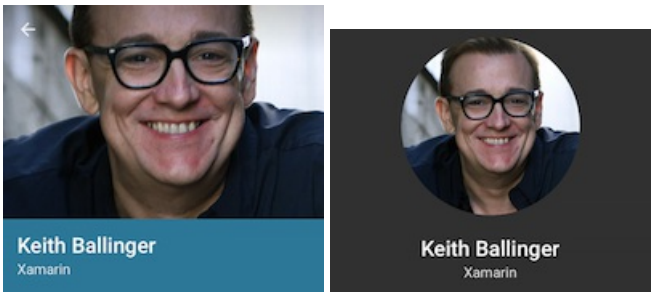
HeroImage

`HeroImage` 控件具有四个属性：

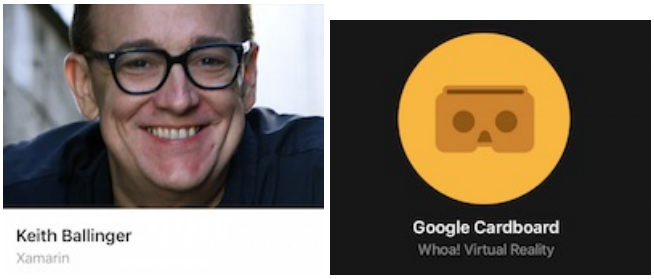
- Text
- 详细信息
- ImageSource
- 方面

```
<pages:HeroImage
    ImageSource="{ DynamicResource HeroImageImage }"
    Text="Keith Ballinger"
    Detail="Xamarin"
/>
```

Android



iOS



ListItem

`ListItem`控件的布局是类似于本机 iOS 和 Android 的列表或表行，但是它还可作为常规视图。在示例中它下面的代码所示内部承载`StackLayout`，但它也可使用数据绑定 scolling 列表控件中。

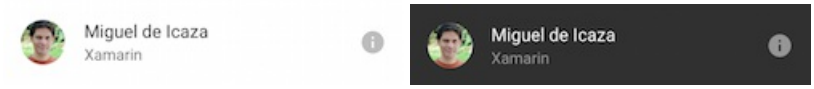
有五个属性：

- 标题
- 详细信息
- ImageSource
- PlaceholdImageSource
- 方面

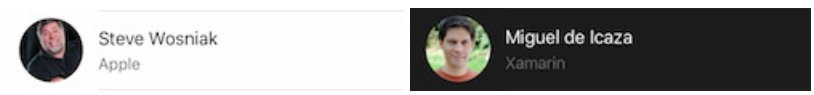
```
<StackLayout Spacing="0">
    <pages:ListItemControl
        Detail="Xamarin"
        ImageSource="{ DynamicResource UserImage }"
        Title="Miguel de Icaza"
        PlaceholdImageSource="{ DynamicResource IconImage }"
    />
```

这些屏幕截图显示了`ListItem`在 iOS 和 Android 平台使用浅色和深色主题：

Android



iOS



自定义控件示例

此自定义的目标`CardView`控件将类似于本机 Android CardView。

它将包含三个属性：

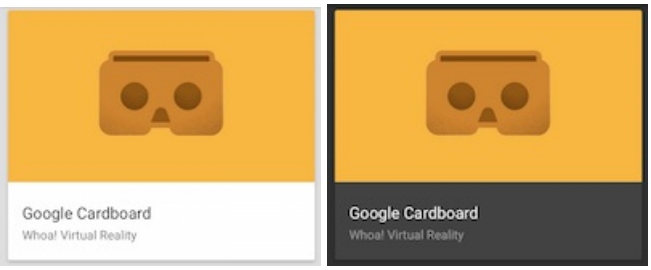
- Text
- 详细信息
- ImageSource

目标是将类似下面的代码的自定义控件（请注意，自定义 `xmlns:local` 是必需的它引用当前程序集）：

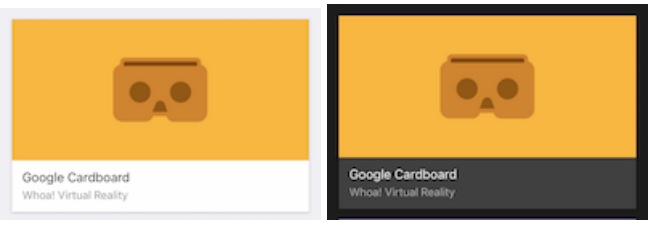
```
<local:CardView
  ImageSource="{ DynamicResource CardViewImage }"
  Text="CardView Text"
  Detail="CardView Detail"
/>
```

它应类似于用颜色对应于内置的浅色和深色主题在下面的屏幕截图：

Android



iOS



构建自定义 CardView

1. DataView 子类
2. 定义字体、布局和边距
3. 为控件的子级创建样式
4. 创建控件布局模板
5. 添加特定于主题的资源
6. 设置的 ControlTemplate CardView 类
7. 将控件添加到页面

1.DataView 子类

C# 子类的 `DataView` 定义控件的可绑定属性。

```
public class CardView : DataView
{
    public static readonly BindableProperty TextProperty =
        BindableProperty.Create ("Text", typeof (string), typeof (CardView), null, BindingMode.TwoWay);

    public string Text
    {
        get { return (string)GetValue (TextProperty); }
        set { SetValue (TextProperty, value); }
    }

    public static readonly BindableProperty DetailProperty =
        BindableProperty.Create ("Detail", typeof (string), typeof (CardView), null, BindingMode.TwoWay);

    public string Detail
    {
        get { return (string)GetValue (DetailProperty); }
        set { SetValue (DetailProperty, value); }
    }

    public static readonly BindableProperty ImageSourceProperty =
        BindableProperty.Create ("ImageSource", typeof (ImageSource), typeof (CardView), null,
BindingMode.TwoWay);

    public ImageSource ImageSource
    {
        get { return (ImageSource)GetValue (ImageSourceProperty); }
        set { SetValue (ImageSourceProperty, value); }
    }

    public CardView()
    {
    }
}
```

2.定义字体、布局和边距

控件设计器将确定这些值作为自定义控件的用户界面设计的一部分。特定于平台的规范是必需的其中 `OnPlatform` 使用元素。

请注意，某些值是指 `StaticResource` s – 这些中将定义第 5 步。

```
<!-- CARDVIEW FONT SIZES -->
<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewTextFontSize">
        <On Platform="iOS, Android" Value="15" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewDetailFontSize">
        <On Platform="iOS, Android" Value="13" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"    x:Key="CardViewTextTextColor">
        <On Platform="iOS" Value="{StaticResource iOSCardViewTextTextColor}" />
        <On Platform="Android" Value="{StaticResource AndroidCardViewTextTextColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="Thickness"    x:Key="CardViewTextlMargin">
        <On Platform="iOS" Value="12,10,12,4" />
        <On Platform="Android" Value="20,0,20,5" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"    x:Key="CardViewDetailTextColor">
        <On Platform="iOS" Value="{StaticResource iOSCardViewDetailTextColor}" />
        <On Platform="Android" Value="{StaticResource AndroidCardViewDetailTextColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="Thickness"    x:Key="CardViewDetailMargin">
        <On Platform="iOS" Value="12,0,10,12" />
        <On Platform="Android" Value="20,0,20,20" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"    x:Key="CardViewBackgroundColor">
        <On Platform="iOS" Value="{StaticResource iOSCardViewBackgroundColor}" />
        <On Platform="Android" Value="{StaticResource AndroidCardViewBackgroundColor}" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewShadowSize">
        <On Platform="iOS" Value="2" />
        <On Platform="Android" Value="5" />
</OnPlatform>

<OnPlatform x:TypeArguments="x:Double" x:Key="CardViewCornerRadius">
        <On Platform="iOS" Value="0" />
        <On Platform="Android" Value="4" />
</OnPlatform>

<OnPlatform x:TypeArguments="Color"    x:Key="CardViewShadowColor">
        <On Platform="iOS, Android" Value="#CDCDD1" />
</OnPlatform>
```

3.为控件的子级创建样式

引用定义要创建自定义控件中将使用的子级的所有元素：

```
<!-- EXPLICIT STYLES (will be Classes) -->
<Style TargetType="Label" x:Key="CardViewTextStyle">
    <Setter Property="FontSize" Value="{ StaticResource CardViewTextFontSize }" />
    <Setter Property="TextColor" Value="{ StaticResource CardViewTextTextColor }" />
    <Setter Property="HorizontalOptions" Value="Start" />
    <Setter Property="Margin" Value="{ StaticResource CardViewTextlMargin }" />
    <Setter Property="HorizontalTextAlignment" Value="Start" />
</Style>

<Style TargetType="Label" x:Key="CardViewDetailStyle">
    <Setter Property="HorizontalTextAlignment" Value="Start" />
    <Setter Property="TextColor" Value="{ StaticResource CardViewDetailTextColor }" />
    <Setter Property="FontSize" Value="{ StaticResource CardViewDetailFontSize }" />
    <Setter Property="HorizontalOptions" Value="Start" />
    <Setter Property="Margin" Value="{ StaticResource CardViewDetailMargin }" />
</Style>

<Style TargetType="Image" x:Key="CardViewImageImageStyle">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="Center" />
    <Setter Property="WidthRequest" Value="220"/>
    <Setter Property="HeightRequest" Value="165"/>
</Style>
```

4.创建控件布局模板

自定义控件的可视设计进行显式声明在控件模板中，使用上面定义的资源：

```
<!--- CARDVIEW -->
<ControlTemplate x:Key="CardViewControlControlTemplate">
  <StackLayout
    Spacing="0"
    BackgroundColor="{ TemplateBinding BackgroundColor }"
  >

    <!-- CARDVIEW IMAGE -->
    <Image
      Source="{ TemplateBinding ImageSource }"
      HorizontalOptions="FillAndExpand"
      VerticalOptions="StartAndExpand"
      Aspect="AspectFill"
      Style="{ StaticResource CardViewImageImageStyle }"
    />

    <!-- CARDVIEW TEXT -->
    <Label
      Text="{ TemplateBinding Text }"
      LineBreakMode="WordWrap"
      VerticalOptions="End"
      Style="{ StaticResource CardViewTextStyle }"
    />


    <!-- CARDVIEW DETAIL -->
    <Label
      Text="{ TemplateBinding Detail }"
      LineBreakMode="WordWrap"
      VerticalOptions="End"
      Style="{ StaticResource CardViewDetailStyle }" />

  </StackLayout>

</ControlTemplate>
```

5.添加特定于主题的资源

由于这是一个自定义控件，添加的资源所使用的资源字典的主题相匹配：

浅色主题颜色

```
<Color x:Key="iOSCardViewBackgroundColor">#FFFFFF</Color>
<Color x:Key="AndroidCardViewBackgroundColor">#FFFFFF</Color>

<Color x:Key="AndroidCardViewTextTextColor">#030303</Color>
<Color x:Key="iOSCardViewTextTextColor">#030303</Color>

<Color x:Key="AndroidCardViewDetailTextColor">#8F8E94</Color>
<Color x:Key="iOSCardViewDetailTextColor">#8F8E94</Color>
```

深色主题颜色

```
<!-- CARD VIEW COLORS -->
            <Color x:Key="iOSCardViewBackgroundColor">#404040</Color>
            <Color x:Key="AndroidCardViewBackgroundColor">#404040</Color>

            <Color x:Key="AndroidCardViewTextTextColor">#FFFFFF</Color>
            <Color x:Key="iOSCardViewTextTextColor">#FFFFFF</Color>

            <Color x:Key="AndroidCardViewDetailTextColor">#B5B4B9</Color>
            <Color x:Key="iOSCardViewDetailTextColor">#B5B4B9</Color>
```

6.设置的 ControlTemplate CardView 类

最后，确保在创建的 C# 类步骤 1使用控件模板中定义步骤 4使用 `Style``Setter` 元素

```
<Style TargetType="local:CardView">
    <Setter Property="ControlTemplate" Value="{ StaticResource CardViewControlControlTemplate }" />
  ... some custom styling omitted
  <Setter Property="BackgroundColor" Value="{ StaticResource CardViewBackgroundColor }" />
</Style>
```

7.将控件添加到页面

`CardView` 控件现在可以添加到一个页面。下面的示例演示它托管在 `StackLayout`：

```
<StackLayout Spacing="0">
  <local:CardView
    Margin="12,6"
    ImageSource="{ DynamicResource CardViewImage }"
    Text="CardView Text"
    Detail="CardView Detail"
  />
</StackLayout>
```

Xamarin.Forms DatePicker

2018/11/13 • • Edit Online

Xamarin.Forms 视图允许用户选择日期。

Xamarin.Forms `DatePicker` 调用平台的日期选取器控件，并允许用户选择日期。`DatePicker` 定义了八个属性：

- `MinimumDate` 类型的 `DateTime`，默认为 1900 年的第一天。
- `MaximumDate` 类型的 `DateTime` 的默认值为 2100年的年份的最后一天。
- `Date` 类型的 `DateTime`，默认值为所选日期 `DateTime.Today`。
- `Format` 类型的 `string`、一个标准或自定义.NET 格式设置字符串，它默认为"D"，长日期模式。
- `TextColor` 类型的 `Color`，用来显示默认为所选的日期的颜色 `Color.Default`。
- `FontAttributes` 类型的 `FontAttributes`，其默认值为 `FontAtributes.None`。
- `FontFamily` 类型的 `string`，其默认值为 `null`。
- `FontSize` 类型的 `double`，其默认值为-1.0。

`DatePicker` 激发 `DateSelected` 事件时用户选择日期。

> **WARNING**
> 设置时 `MinimumDate` 并 `MaximumDate`，请确保 `MinimumDate` 总是小于或等于 `MaximumDate`。否则为 `DatePicker` 将引发异常。

在内部，`DatePicker` 可确保 `Date` 之间 `MinimumDate` 和 `MaximumDate`（含）之间。如果 `MinimumDate` 或 `MaximumDate` 设置，以便 `Date` 之间，不是 `DatePicker` 的值将调整 `Date`。

所有八个属性受 `BindableProperty` 对象，这意味着它们可以设置的样式，这些属性可以是数据绑定的目标。`Date` 属性设置了默认绑定模式 `BindingMode.TwoWay`，这意味着它可以是数据绑定中使用的应用程序的目标模型-视图-视图模型 (MVVM)体系结构。

初始化日期时间属性

在代码中，您可以初始化 `MinimumDate`、`MaximumDate`，并 `Date` 类型的值的属性 `DateTime`：

```
DatePicker datePicker = new DatePicker
{
    MinimumDate = new DateTime(2018, 1, 1),
    MaximumDate = new DateTime(2018, 12, 31),
    Date = new DateTime(2018, 6, 21)
};
```

当 `DateTime` 值指定在 XAML，XAML 分析器使用 `DateTime.Parse` 方法替换 `CultureInfo.InvariantCulture` 参数来将字符串转换为 `DateTime` 值。必须精确格式指定日期：两位数的月份、两位数的日期和由斜杠分隔的四位数年份：

```
<DatePicker MinimumDate="01/01/2018"
            MaximumDate="12/31/2018"
            Date="06/21/2018" />
```

如果 `BindingContext` 的属性 `DatePicker` 设置为包含类型的属性的 ViewModel 的实例 `DateTime` 名为 `MinDate`，

`MaxDate`，并且`SelectedDate`（例如），您可以实例化`DatePicker`如下所示：

```
<DatePicker MinimumDate="{Binding MinDate}"
            MaximumDate="{Binding MaxDate}"
            Date="{Binding SelectedDate}" />
```

在此示例中，所有三个属性将初始化为 ViewModel 中的相应属性。因为`Date`属性设置了绑定模式`TwoWay`、用户选择自动反映在 ViewModel 中的任何新日期。

如果`DatePicker`不包含一个绑定上其`Date`属性，应用程序应附加到一个处理程序`DateSelected`事件要通知当用户选择新的日期。

有关设置字体属性的信息，请参阅字体。

DatePicker 和布局

可以使用不受约束的水平布局选项，如`Center`，`Start`，或`End`与`DatePicker`：

```
<DatePicker ···
            HorizontalOptions="Center"
            ··· />
```

但是，这不是建议在一起。这取决于设置`Format`所选的属性的日期可能需要不同的显示宽度。例如，"D"格式字符串会导致`DateTime`若要用长格式和"星期三，2018 年 9 月 12，"显示日期，需要更大的显示宽度比"星期五，4，2018 年 5 月"。根据平台，这种差异可能会导致`DateTime`视图，以更改宽度在布局中，或显示被截断。

> **TIP**
> 最好是使用默认`HorizontalOptions`设置为`Fill`与`DatePicker`，而不要使用的宽度`Auto`放置时`DatePicker`中`Grid`单元格。

应用程序中的 DatePicker

DaysBetweenDates 示例包含两个`DatePicker`在其页面上的视图。这些可用于选择两个日期和程序会计算这些日期之间相差的天数。该程序不会更改的设置`MinimumDate`和`MaximumDate`属性，因此，两个日期必须介于 1900 年 2100年。

下面是 XAML 文件：

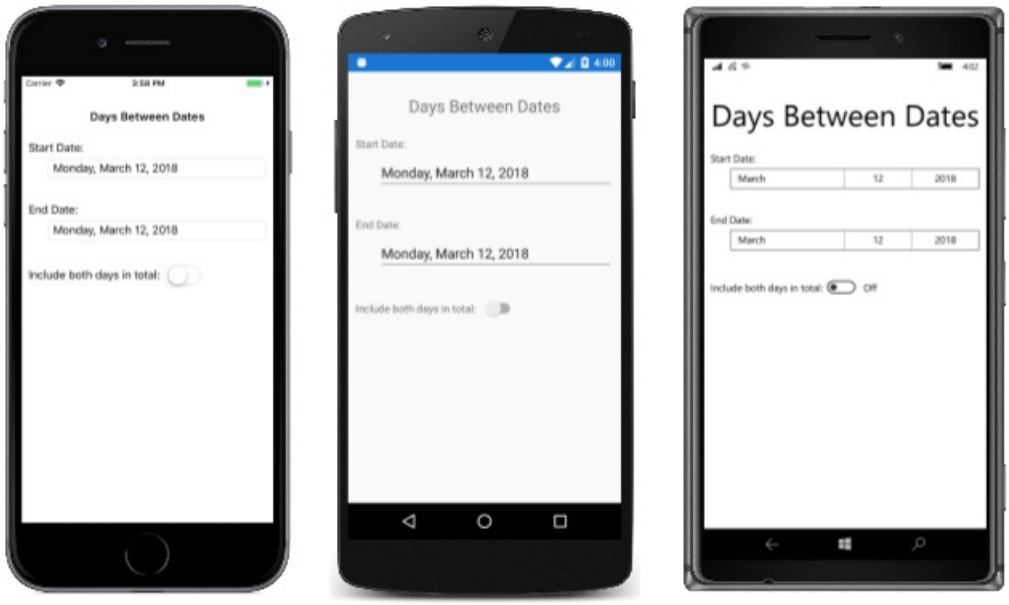
```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:DaysBetweenDates"
             x:Class="DaysBetweenDates.MainPage">
    <ContentPage.Padding>
        <OnPlatform x:TypeArguments="Thickness">
            <On Platform="iOS" Value="0, 20, 0, 0" />
        </OnPlatform>
    </ContentPage.Padding>

    <StackLayout Margin="10">
        <Label Text="Days Between Dates"
               Style="{DynamicResource TitleStyle}"
               Margin="0, 20"
               HorizontalTextAlignment="Center" />

        <Label Text="Start Date:" />

        <DatePicker x:Name="startDatePicker"
                    Format="D"
                    Margin="30, 0, 0, 30"
                    DateSelected="OnDateSelected" />

        <Label Text="End Date:" />

        <DatePicker x:Name="endDatePicker"
                    MinimumDate="{Binding Source={x:Reference startDatePicker},
                                          Path=Date}"
                    Format="D"
                    Margin="30, 0, 0, 30"
                    DateSelected="OnDateSelected" />

        <StackLayout Orientation="Horizontal"
                     Margin="0, 0, 0, 30">
            <Label Text="Include both days in total: "
                   VerticalOptions="Center" />
            <Switch x:Name="includeSwitch"
                    Toggled="OnSwitchToggled" />
        </StackLayout>

        <Label x:Name="resultLabel"
               FontAttributes="Bold"
               HorizontalTextAlignment="Center" />

    </StackLayout>
</ContentPage>
```

每个 `DatePicker` 分配 `Format` 长日期格式"D"的属性。另请注意 `endDatePicker` 对象具有面向的绑定其 `MinimumDate` 属性。绑定源是所选 `Date` 属性的 `startDatePicker` 对象。这可确保的结束日期始终更高版本或等于开始日期。除了这两个 `DatePicker` 对象，`Switch` 标记为"总共包含这两天"。

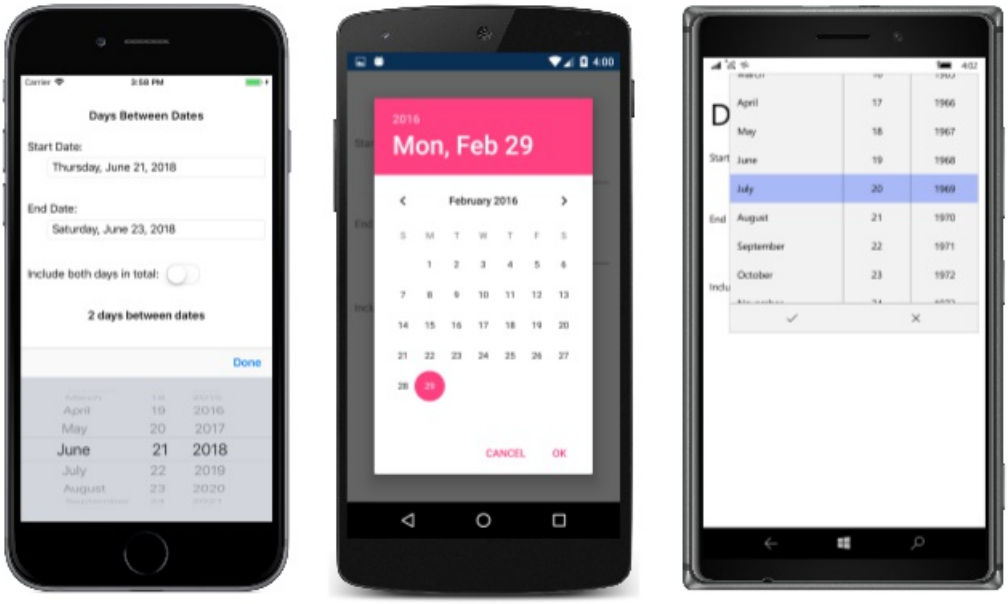
这两个 `DatePicker` 视图具有处理程序附加到 `DateSelected` 事件，并 `Switch` 具有一个处理程序附加到其 `Toggled` 事件。这些事件处理程序中的代码隐藏文件，并触发新的计算的两个日期之间的天数：

```
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
    }

    void OnDateSelected(object sender, DateChangedEventArgs args)
    {
        Recalculate();
    }

    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        Recalculate();
    }

    void Recalculate()
    {
        TimeSpan timeSpan = endDatePicker.Date - startDatePicker.Date +
            (includeSwitch.IsToggled ? TimeSpan.FromDays(1) : TimeSpan.Zero);

        resultLabel.Text = String.Format("{0} day{1} between dates",
                                         timeSpan.Days, timeSpan.Days == 1 ? "" : "s");
    }
}
```

该示例首先运行时，同时`DatePicker`视图将初始化为当天的日期。下面的屏幕截图显示了在 iOS、Android 和通用 Windows 平台上运行的程序：



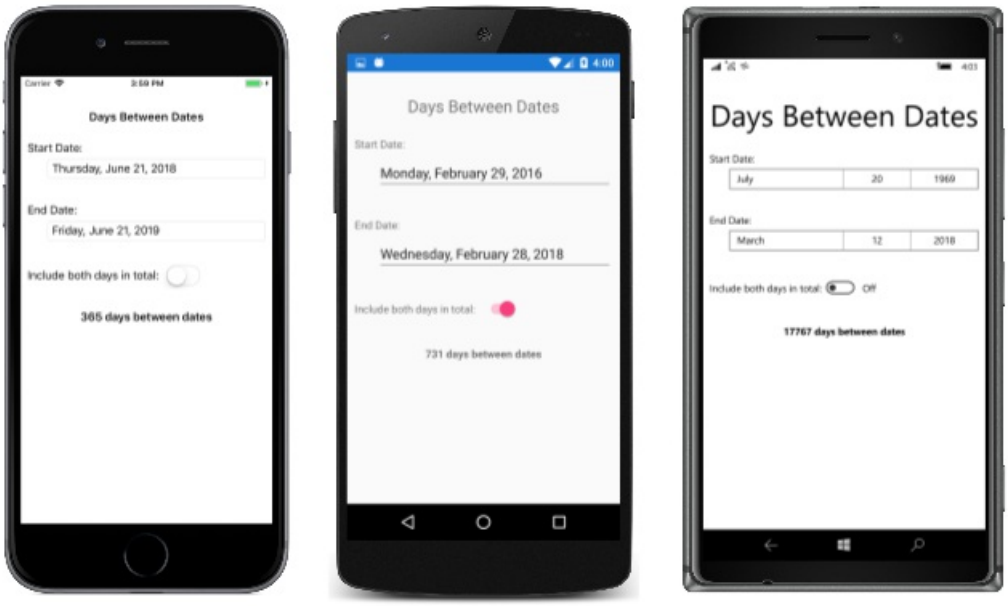
点击的`DatePicker`显示调用平台日期选取器。三个平台实现此日期选取器很大不同，但是每种方法是熟悉该平台的用户：



TIP

在 Android 上，`DatePicker` 可以通过重写自定义对话框 `CreateDatePickerDialog` 中自定义呈现器的方法。这样，例如，其他按钮来添加到了对话框。

选择两个日期后，应用程序将显示这些日期之间相差的天数：



相关链接

- DaysBetweenDates 示例
- DatePicker API

在 Xamarin.Forms 中 SkiaSharp 图形

2018/10/25 • • Edit Online

用于二维图形 Xamarin.Forms 应用程序中使用 SkiaSharp

SkiaSharp 是.NET 和 C# 由 Google 产品中广泛使用的开放源代码 Skia 图形引擎提供支持的 2D 图形系统。可以在 Xamarin.Forms 应用程序中使用 SkiaSharp 绘制 2D 矢量图形、位图和文本。请参阅2D 绘制SkiaSharp 库有关的更多常规信息和一些其他教程的指南。

本指南假定你熟悉 Xamarin.Forms 编程。

适用于 Xamarin.Forms 的网络研讨会：SkiaSharp

SkiaSharp 初步准备之后

SkiaSharp xamarin.forms 打包为 NuGet 包。创建 Xamarin.Forms 解决方案在 Visual Studio 或 Visual Studio for Mac 后，可以使用 NuGet 包管理器来搜索**SkiaSharp.Views.Forms**包并将其添加到你的解决方案。如果选中**引用**部分的每个项目添加 SkiaSharp 后，可以看到各种**SkiaSharp**库已添加到每个解决方案中的项目。

如果在 Xamarin.Forms 应用程序面向 iOS，使用项目属性页上以最小的部署目标更改为 iOS 8.0。

在使用 SkiaSharp 任何 C# 页面中，将想要包括 `using` 指令 `SkiaSharp` 命名空间，其中包含所有 SkiaSharp 类、结构和将在图形中使用的枚举编程。您还应该 `using` 指令 `SkiaSharp.Views.Forms` 类特定于 Xamarin.Forms 的命名空间。这是一个更小命名空间，其中最重要的类是 `SKCanvasView` 。此类派生自 Xamarin.Forms `View` 类，并承载 SkiaSharp 图形输出。

> **IMPORTANT**
> `SkiaSharp.Views.Forms` 命名空间还包含 `SKGLView` 派生的类 `View` 但使用 OpenGL 的呈现图形。为了简单起见，本指南将限制到本身 `SKCanvasView` ，但使用 `SKGLView` 相反是非常相似。

SkiaSharp 绘制基础知识

一些可以使用 SkiaSharp 绘制的最简单的图形图是圆、椭圆和矩形。在显示这些数字，你将了解 SkiaSharp 坐标、大小和颜色。文本和位图的显示是更复杂，但这些文章还介绍这些技术。

SkiaSharp 线和路径

图形路径是一系列相互连接的直线和曲线。路径可以描边，填充，或两者。本文包含线条图形，包括笔划结束和联接，并带短划线和点线，但缺乏曲线几何图形的停止点的多个方面。

SkiaSharp 转换

转换允许图形对象统一转换、缩放、旋转或倾斜。本文还介绍如何创建非仿射转换和将转换应用于路径使用标准的 3-3 转换矩阵。

SkiaSharp 曲线和路径

路径的探索继续将曲线添加到路径对象，并利用功能强大的路径的其他功能。您将看到如何简洁的文本字符串中指

定完整路径、如何使用路径的效果，以及如何深入探讨路径内部结构。

SkiaSharp 位图

位图是位对应于显示设备的像素的矩形数组。本系列文章演示如何加载、保存、显示、创建、上绘制、进行动画处理，并访问 SkiaSharp 位图的位。

SkiaSharp 效果

效果是 alter 正常显示图形，包括线性和循环渐变、位图平铺、混合模式、模糊和其他人的属性。

相关链接

- SkiaSharp Api
- SkiaSharpFormsDemos（示例）
- SkiaSharp 通过 Xamarin.Forms 网络研讨会（视频）

在 Xamarin.Forms 中的图像

2018/10/26 • • Edit Online

可以使用 Xamarin.Forms 跨平台共享映像、可以专门为每个平台，加载它们或它们可以为显示下载。

映像是应用程序导航、可用性和品牌的一个重要部分。Xamarin.Forms 应用程序需要能够在所有平台之间共享映像，但也有可能每个平台上显示不同的图像。

特定于平台的映像还所需的图标和初始屏幕;这些需要根据每个平台配置。

显示图像

使用 Xamarin.Forms `Image` 视图，以在页面上显示图像。它具有两个重要属性：

- `Source` -An `ImageSource` 实例、文件、Uri 或资源，设置要显示的图像。
- `Aspect` -如何调整大小的映像（无论是为 stretch、裁剪或 letterbox）中显示的边界内。

`ImageSource` 可以为每种类型的图像源使用的静态方法获取实例：

- `FromFile` -需要文件名或可以解决每个平台的文件路径。
- `FromUri` -需要一个 Uri 对象，例如。`new Uri("http://server.com/image.jpg")` .
- `FromResource` -需要嵌入到应用程序或.NET Standard 类库项目中，使用某个图像文件的资源标识符**生成操作**：**EmbeddedResource**。
- `FromStream` -需要提供图像数据的流。

`Aspect` 属性确定将如何缩放图像以适合显示区域：

- `Fill` -拉伸图像以完全且完全填充的显示区域。这可能会导致被扭曲图像。
- `AspectFill` -裁剪图像，以便它同时还能保留方面填充显示区域 (即。无扭曲)。
- `AspectFit` -上下黑边映像（如果需要），以便整个图像适合的显示区域的空白区域添加到顶部/底部或边，具体取决于图像是高或宽。

可以从加载图像本地文件，则嵌入的资源，或下载。

本地映像

可以添加到每个应用程序项目并从 Xamarin.Forms 共享代码中引用的图像文件。映像是特定于平台的例如，在不同的平台或略有不同的设计使用不同的分辨率时，此方法的分发映像时需要。

所有应用之间使用的单一映像*必须在每个平台上使用相同的文件名*，并且它应为有效的 Android 资源名称 (即。允许只包含小写字母、数字、下划线和句点)。

- **iOS** -首选方式管理和支持的映像，因为 iOS 9 是使用资产目录**图像集**，其中应包含的所有所需支持各种设备和缩放比例的图像的版本应用程序。有关详细信息，请参阅将图像添加到资产目录映像集。
- **Android** -将在图像放资**源/drawable**目录**生成操作**：**AndroidResource**。此外可以提供的图像的高和低 DPI 版本 (在适当地命名为资**源**如子目录**drawable ldpi**，**drawable hdpi**，和**drawable xhdpi**)。
- **通用 Windows 平台 (UWP)** -将图像放在应用程序的根目录下使用**生成操作**：**内容**。

> **IMPORTANT**
> 在 iOS 9 之前，映像通常放置在资源具有文件夹生成操作：**BundleResource**。但是，apple 已弃用的 iOS 应用程序中的映像的使用此方法。有关详细信息，请参阅图像大小和文件名。

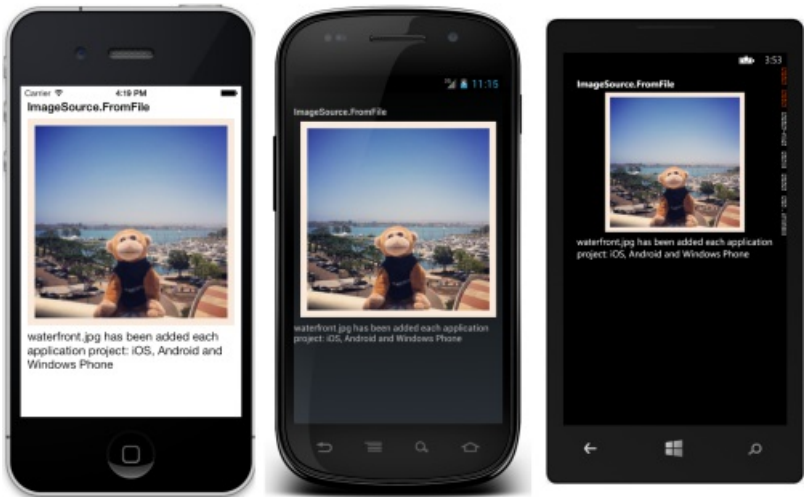
遵守这些规则进行文件命名和放置允许以下的 XAML 加载和显示在所有平台上的图像：

```
<Image Source="waterfront.jpg" />
```

等效于C#代码如下所示：

```
var image = new Image { Source = "waterfront.jpg" };
```

以下屏幕截图显示结果的每个平台上显示本地映像：



为提高灵活性`Device.RuntimePlatform`属性可用来选择不同的图像文件或路径对于某些或所有平台，此代码示例中所示：

```
image.Source = Device.RuntimePlatform == Device.Android ? ImageSource.FromFile("waterfront.jpg") :
ImageSource.FromFile("Images/waterfront.jpg");
```

> **IMPORTANT**
> 若要跨所有平台使用相同的映像文件名名称必须在所有平台上有效。Android 绘图制定了命名限制 – 允许小写字母、数字、下划线和句点 – 和跨平台兼容性必须随后出现在所有其他平台上过。示例文件名**waterfront.png**如下所示的规则，但无效文件名的示例包括"water front.png"，"WaterFront.png"，"water-front.png"和"wåterfront.png"。

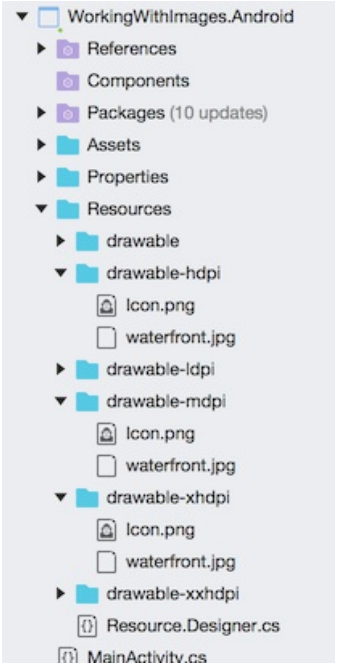
本机分辨率（Retina 和高 DPI）

iOS、Android 和 UWP 包括针对不同的图像分辨率，其中操作系统选择合适的映像在运行时根据设备的功能的支持。Xamarin.Forms 使用本机平台的 Api，用于加载本地映像，因此如果正确命名和位于项目文件，它会自动支持备用解决方法。

因为 iOS 9 管理映像的首选的方法是拖动到相应的资产目录图像集所需的每个分辨率的图像。有关详细信息，请参阅将图像添加到资产目录映像集。

在 iOS 9 之前，retina 版本的映像无法放置在资源文件夹的两个和第三次使用的分辨率**@2x**或@3x**上的文件扩展名（例如之前，的文件名的后缀。**myimage@2x.png**）。但是，apple 已弃用的 iOS 应用程序中的映像的使用此方法。有关详细信息，请参阅图像大小和文件名。

Android 备用分辨率图像应置于特殊命名目录在 Android 项目中，如以下屏幕截图中所示：



UWP 图像文件名称可以使用作为后缀 `.scale-xxx` 文件扩展名之前，其中 `xxx` 是应用到的资产，例如缩放百分比**myimage.scale 200.png**。然后可以引用映像以在代码或 XAML 不带小数位数修饰符，例如刚刚**myimage.png**。该平台将选择最接近的相应资产规模基于显示器的当前 DPI。

显示图像的其他控件

某些控件具有显示图像，如的属性：

- `Page` -任何页上，键入派生 `Page` 已 `Icon` 并 `BackgroundImage` 属性，可以分配的本地文件引用。在某些情况下，例如何时 `NavigationPage` 显示 `ContentPage` ，将显示该图标，如果受平台支持。

  > **IMPORTANT**
  > 在 iOS 上，`Page.Icon` 属性不能填充从资产目录映像组中的映像。相反，加载图标的映像 `Page.Icon` 属性从资源iOS 项目文件夹中的。

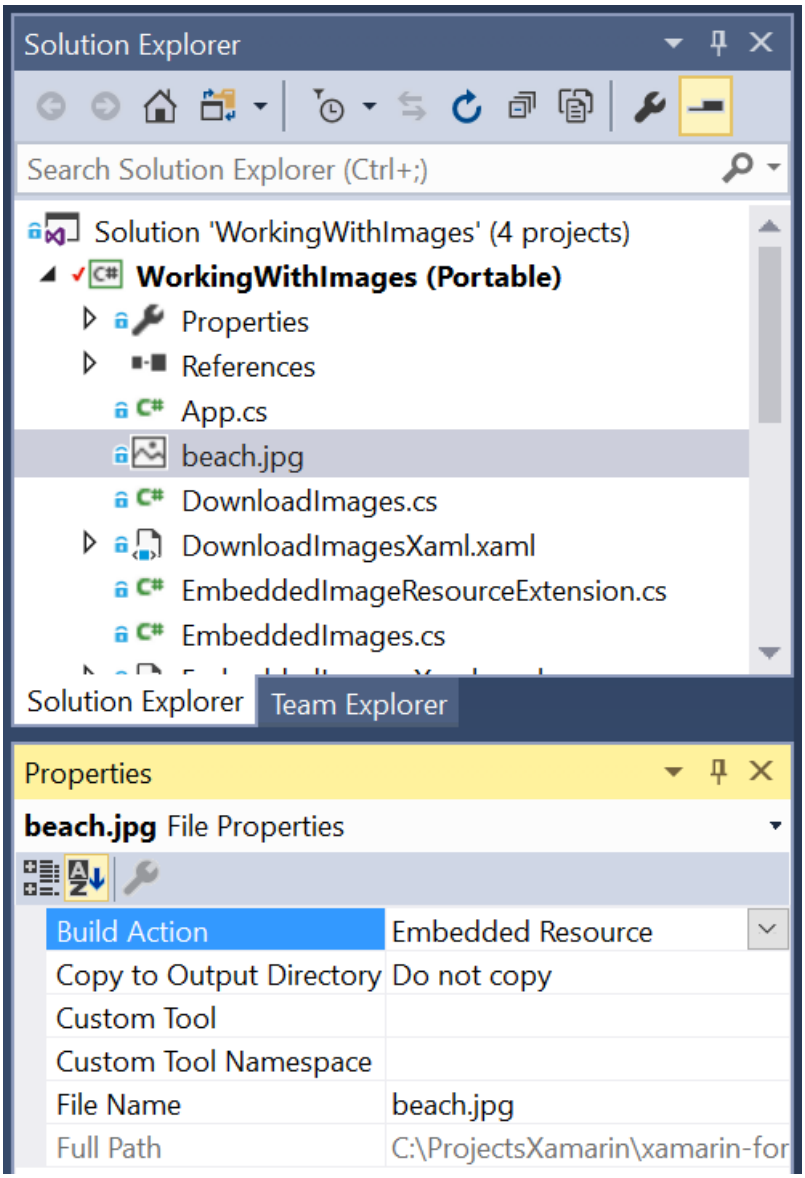
- `ToolbarItem` -具有 `Icon` 可以设置为本地文件引用的属性。
- `ImageCell` -具有 `ImageSource` 从本地文件、嵌入的资源或一个 URI 检索可以设置为图像的属性。

嵌入图像

嵌入的图像还提供了应用程序（例如本地映像），但而不是让每个应用程序的文件结构图像中的映像的副本在程序集作为资源嵌入文件。此方法分发映像时每个平台上使用相同的映像，建议使用，并且是尤其适合于创建组件，如图像与代码捆绑在一起。

若要在项目中嵌入图像，右键单击要添加新项，然后选择你想要添加的图像/秒。默认情况下该图像将出现**生成操作：无**；这需要设置为**生成操作：EmbeddedResource**。

- Visual Studio
- Visual Studio for Mac



生成操作可以查看和更改在**属性**窗口中的文件。

在此示例中的资源 ID 是**WorkingWithImages.beach.jpg**。IDE 已生成此默认值的串联**默认 Namespace**对于此项目包含文件名，使用句点 (.) 之间的每个值。

如果你的项目中，您将嵌入的图像放入文件夹，文件夹名称也用句点分隔 (.) 中的资源 id。移动**beach.jpg**映像到名为的文件夹**MyImages**会导致资源 ID 为**WorkingWithImages.MyImages.beach.jpg**

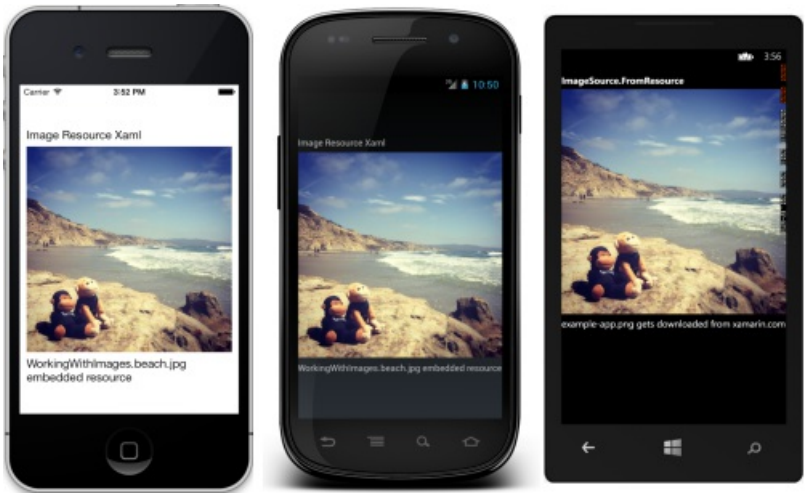
用于加载嵌入的图像的代码只是将传递资源 **ID**到 `ImageSource.FromResource` 方法，如下所示：

```
var embeddedImage = new Image { Source = ImageSource.FromResource("WorkingWithImages.beach.jpg",
typeof(EmbeddedImages).GetTypeInfo().Assembly) };
```

> **NOTE**
>
> 若要支持嵌入的图像显示在通用 Windows 平台上的发布模式下，有必要，请使用的重载 `ImageSource.FromResource`，它指定要在其中搜索的图像的源程序集。

目前资源标识符没有隐式转换。相反，必须使用 `ImageSource.FromResource` 或 `new ResourceImageSource()` 加载嵌入的图像。

以下屏幕截图显示在每个平台上显示的嵌入的图像的结果：



使用 XAML

因为不没有从任何内置类型转换器 `string` 到 `ResourceImageSource`，这些类型的映像不能以本机方式加载 XAML。相反，可以编写简单的自定义 XAML 标记扩展来加载使用图像资源 ID在 XAML 中指定：

```
[ContentProperty (nameof(Source))]
public class ImageResourceExtension : IMarkupExtension
{
 public string Source { get; set; }

 public object ProvideValue (IServiceProvider serviceProvider)
 {
   if (Source == null)
   {
     return null;
   }

   // Do your translation lookup here, using whatever method you require
   var imageSource = ImageSource.FromResource(Source, typeof(ImageResourceExtension).GetTypeInfo().Assembly);

   return imageSource;
 }
}
```

> **NOTE**
>
> 若要支持嵌入的图像显示在通用 Windows 平台上的发布模式下，有必要，请使用的重载 `ImageSource.FromResource`，它指定要在其中搜索的图像的源程序集。

若要使用此扩展插件添加一个自定义 `xmlns` 到 XAML 中，使用正确的命名空间和程序集值的项目。然后可以使用以下语法设置图像源：`{local:ImageResource WorkingWithImages.beach.jpg}`。完整的 XAML 示例如下所示：

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage
   xmlns="http://xamarin.com/schemas/2014/forms"
   xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
   xmlns:local="clr-namespace:WorkingWithImages;assembly=WorkingWithImages"
   x:Class="WorkingWithImages.EmbeddedImagesXaml">
 <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
   <!-- use a custom Markup Extension -->
   <Image Source="{local:ImageResource WorkingWithImages.beach.jpg}" />
 </StackLayout>
</ContentPage>
```

故障排除的嵌入的图像

调试代码

因为它是有时难以理解为什么不加载特定的图像资源，可以暂时将以下调试代码添加到应用程序以帮助确认正确配置了资源。它会输出所有已知的资源嵌入到给定的程序集控制台来帮助调试加载问题的资源。

```
using System.Reflection;
// ...
// NOTE: use for debugging, not in released app code!
var assembly = typeof(EmbeddedImages).GetTypeInfo().Assembly;
foreach (var res in assembly.GetManifestResourceNames())
{
    System.Diagnostics.Debug.WriteLine("found resource: " + res);
}
```

在其他项目中嵌入的图像

默认情况下 `ImageSource.FromResource` 方法只会与代码调用同一程序集中的映像 `ImageSource.FromResource` 方法。使用上面的调试代码可以确定哪些程序集包含特定资源通过更改 `typeof()` 语句 `Type` 已知要处于每个程序集。

但是，将搜索的嵌入图像的源程序集指定的参数为 `ImageSource.FromResource` 方法：

```
var imageSource = ImageSource.FromResource("filename.png", typeof(MyClass).GetTypeInfo().Assembly);
```

下载图像

以下 XAML 所示，可以显示中，为自动下载图像：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       x:Class="WorkingWithImages.DownloadImagesXaml">
  <StackLayout VerticalOptions="Center" HorizontalOptions="Center">
    <Label Text="Image UriSource Xaml" />
    <Image Source="https://xamarin.com/content/images/pages/forms/example-app.png" />
    <Label Text="example-app.png gets downloaded from xamarin.com" />
  </StackLayout>
</ContentPage>
```

等效于C#代码如下所示：

```
var webImage = new Image { Source = ImageSource.FromUri(new
Uri("https://xamarin.com/content/images/pages/forms/example-app.png")) };
```

`ImageSource.FromUri` 方法需要 `Uri` 对象，并返回一个新 `UriImageSource` 读取从 `Uri` 。

因此，下面的示例也将正常运行，也是有 URI 字符串的隐式转换：

```
webImage.Source = "https://xamarin.com/content/images/pages/forms/example-app.png";
```

以下屏幕截图显示在每个平台上显示的远程图像的结果：

已下载的映像缓存

一个 `UriImageSource` 还支持下载映像，通过以下属性配置的缓存：

- `CachingEnabled` -是否启用缓存（`true` 默认情况下）。
- `CacheValidity` -A `TimeSpan` ，它定义将本地存储图像的时间长度。

默认情况下启用缓存，并将该映像存储 24 小时内的本地。若要禁用特定的图像的缓存，实例化，如下所示图像源：

```
image.Source = new UriImageSource { CachingEnabled = false, Uri="http://server.com/image" };
```

若要设置特定缓存期（例如，5 天）实例化图像源，如下所示：

```
webImage.Source = new UriImageSource
{
    Uri = new Uri("https://xamarin.com/content/images/pages/forms/example-app.png"),
    CachingEnabled = true,
    CacheValidity = new TimeSpan(5,0,0,0)
};
```

内置缓存可以轻松支持方案，如每个单元中滚动的映像，其中你可以设置（或绑定）映像的列表，并允许内置缓存负责重新加载映像时单元格滚动到视图返回。

图标和初始屏幕

虽然不与相关 `Image` 视图、应用程序图标和初始屏幕也是 Xamarin.Forms 项目中的映像的一个重要用途。

设置图标和初始屏幕的 Xamarin.Forms 应用可在每个应用程序项目中。这意味着生成正确调整大小的 iOS、Android 和 UWP 的映像。这些映像应名为并位于根据每个平台的要求。

图标

请参阅iOS 处理图像，Google 插图，并准则磁贴和图标资产有关创建这些应用程序资源的详细信息。

初始屏幕

只有 iOS 和 UWP 应用程序需要（也称为启动屏幕或默认映像）的初始屏幕。

请参阅的文档iOS 处理图像并初始屏幕Windows 开发人员中心上。

总结

Xamarin.Forms 提供多种不同方式在跨平台应用程序，允许跨平台使用的相同图像或指定的特定于平台的映像中包含图像。此外会自动缓存下载的图像，自动执行常见的编码方案。

应用程序图标和初始屏幕图像为设置，并与非 Xamarin.Forms 应用程序的配置遵循相同的指南用于特定于平台的应用。

相关链接

- WorkingWithImages（示例）
- iOS 处理图像
- Android 插图
- 磁贴和图标资产的指导原则

在 Xamarin.Forms 中的布局

2018/7/13 • • Edit Online

Xamarin.Forms 具有多个布局和功能的组织在屏幕上的内容。

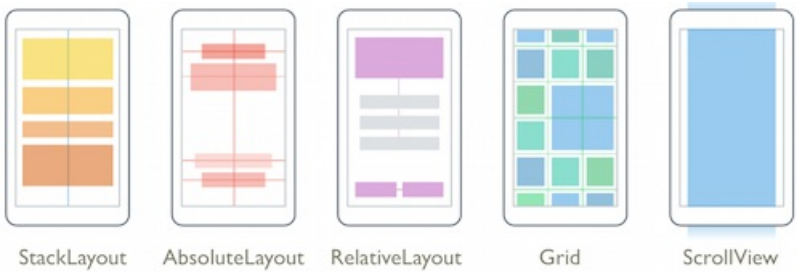
Xamarin.Forms 布局，也可由Xamarin 学院课程

每个布局控件被下述，以及如何处理屏幕方向更改的详细信息：

- **StackLayout** –用来排列视图线性增长，水平或垂直。视图在 StackLayout 中的可以对齐到中心，左或向右的布局。
- **AbsoluteLayout** –用来排列视图通过设置坐标和大小方面绝对值或比率。AbsoluteLayout 可以用于分层视图，以及定位到左、靠右或居中。
- **RelativeLayout** –用来排列视图通过设置相对于其父级的尺寸和位置的约束。
- **网格** –用来排列在网格中的视图。根据绝对值或比率，可以指定行和列。
- **FlexLayout** –用来与包装水平或垂直排列视图。
- **ScrollView** –用于提供滚动时视图无法完全适合屏幕的界限。
- **LayoutOptions** –定义对齐方式和视图，相对于其父级的扩展。
- **输入透明度** –指定元素是否接收输入。
- **边距和填充** –演示如何在用户界面中呈现元素时控制布局行为。
- **设备方向** –说明如何处理设备方向更改。
- **在平板电脑和桌面设备上的布局** –演示如何针对每个平台上较大的屏幕进行优化。
- **创建自定义布局** –介绍了如何创建一个自定义布局的类。
- **布局压缩** –删除指定从可视化树的布局以试图提升页面呈现性能。

平台控件还可直接在与 Xamarin.Forms 布局**本机嵌入**（新 Xamarin.Forms 2.2 中），你可以**创建自定义布局**来满足特定要求。

下图直观显示布局控件：



选择正确的布局

选择应用程序中的布局可以帮助或伤害您为要创建具有吸引力且可使用 Xamarin.Forms 应用。花费一些时间来考虑每个布局也可帮助你编写更干净且更具缩放性 UI 代码。屏幕可以具有不同的布局来实现特定的设计的组合。

StackLayout

`StackLayout` 用于显示视图沿水平或垂直的行。根据视图的确定位置和布局中的大小 `HeightRequest`，`WidthRequest`，`HorizontalOptions` 和 `VerticalOptions`。`StackLayout` 通常用作基布局，排列在屏幕上的其他布局。

有关何时示例 `StackLayout` 会是一个不错的选择，请考虑的应用程序需要显示一个按钮和具有左对齐的标签和右对齐按钮的标签。

```
<StackLayout Orientation="Horizontal">
  <Label HorizontalOptions="StartAndExpand" Text="Label" />
  <Button HorizontalOptions="End" Text="Button" />
</StackLayout>
```

FlexLayout

`FlexLayout` 类似于 `StackLayout`，因为它在水平或垂直显示子视图：

```
<FlexLayout Direction="Column"
            AlignItems="Center"
            JustifyContent="SpaceEvenly">

    <Label Text="FlexLayout in Action" />
    <Button Text="Button" />
    <Label Text="Another Label" />
</FlexLayout>
```

但是，如果过多的子节点中单个行或列，容纳不下 `FlexLayout` 仍可包装这些视图。 `FlexLayout` CSS 灵活框布局模块上，为基础，具有许多相同的内置选项用于定位和对齐及其子级。

AbsoluteLayout

`AbsoluteLayout` 用于显示具有作为显式值或相对于布局的大小的值的大小和位置所指定的视图。与不同 `StackLayout` 并 `Grid`， `AbsoluteLayout` 允许子视图重叠。与不同 `RelativeLayout`， `AbsoluteLayout` 不允许您将置于屏幕上的元素。

有关何时示例 `AbsoluteLayout` 会是一个不错的选择，请考虑的应用程序需要提供作为堆栈对象的集合。上述方法通常被显示的照片或多首歌曲的唱片集时。下面的代码使用旋转以提示的超时和累积内容元素提供一堆的外观：

在 XAML：

```
<AbsoluteLayout Padding="15">
  <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
Rotation="30"
    Source="bottom.png" />
  <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
Rotation="60"
    Source="middle.png" />
  <Image AbsoluteLayout.LayoutFlags="PositionProportional" AbsoluteLayout.LayoutBounds="0.5, 0, 100, 100"
    Source="cover.png" />
</AbsoluteLayout>
```

请注意上述代码中的以下方面：

- 每个 `Image` 显示在同一位置（中间的水平空间）
- `Padding` 视为 `AbsoluteLayout`，但不同于 `RelativeLayout`，这会将其忽略。
- `AbsoluteLayout.LayoutFlags` 指定将如何解释布局范围。在这种情况下 `PositionProportional`，意味着，坐标将布局的大小比率而大小将被解释为显式的大小。
- `AbsoluteLayout.Layoutbounds` 按此顺序指定的水平位置、垂直位置、宽度和高度。

RelativeLayout

`RelativeLayout` 用于显示视图大小和位置指定为相对于布局或另一个视图的值的值。相对值不需要匹配他对应相关视图上的值。例如，就可以设置的视图 `Width` 属性设置为另一个视图成正比 `X` 属性。

RelativeLayout 可以用于创建跨设备的大小按比例扩展的用户界面。以下 XAML 实现以在最顶部边角的框是图案中带有 flagpole 中心中的标志：

```
<RelativeLayout HorizontalOptions="FillAndExpand" VerticalOptions="FillAndExpand">
  <BoxView Color="Blue" HeightRequest="50" WidthRequest="50"
    RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = 0}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height, Factor = 0}" />
  <BoxView Color="Red" HeightRequest="50" WidthRequest="50"
    RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = .9}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height, Factor = 0}" />
  <BoxView Color="Gray" WidthRequest="15" x:Name="pole"
    RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=.75}"
    RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToParent, Property=Width, Factor = .45}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height, Factor = .25}"
/>
  <BoxView Color="Green"
    RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height, Factor=.10,
Constant=10}"
    RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,Property=Width, Factor=.2,
Constant=20}"
    RelativeLayout.XConstraint= "{ConstraintExpression Type=RelativeToView, ElementName=pole, Property=X,
Constant=15}"
    RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToView, ElementName=pole, Property=Y,
Constant=0}" />
</RelativeLayout>
```

请注意上述代码中的以下方面：

- 位置和大小指定为约束。
- 名为 flagpole，以便该标志的（绿色框的）可以相对于 flagpole 设置位置。
- 约束表达式具有 `Factor` 和 `Constant` 属性，可用于为序列图中定义的位置和大小（或小数部分）的属性的其他对象，以及一个常量。常量可以为负数。

网格

`Grid` 用于显示行和列中的元素。请注意，在网格不是表，因此它不具有这一概念的单元格、页眉和页脚行或行与列之间的边框。一般情况下，不适合于显示表格数据网格。使用，请考虑ListView或TableView。

有关何时示例 `Grid` 是正确的布局，若要使用，请考虑一个计算器数字输入。一个计算器的数字输入可能包含四个行和三个列，每个都有一个按钮。下面的代码实现此设计：

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>

  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Button Text="1" Grid.Row="0" Grid.Column="0" />
  <Button Text="2" Grid.Row="0" Grid.Column="1" />
  <Button Text="3" Grid.Row="0" Grid.Column="2" />
  <Button Text="4" Grid.Row="1" Grid.Column="0" />
  <Button Text="5" Grid.Row="1" Grid.Column="1" />
  <Button Text="6" Grid.Row="1" Grid.Column="2" />
  <Button Text="7" Grid.Row="2" Grid.Column="0" />
  <Button Text="8" Grid.Row="2" Grid.Column="1" />
  <Button Text="9" Grid.Row="2" Grid.Column="2" />
  <Button Text="0" Grid.Row="3" Grid.Column="1" />
  <Button Text="&lt;-" Grid.Row="3" Grid.Column="2" />
</Grid>
```

请注意上述代码中的以下方面：

- 网格和列显式指定，不会推断出的内容。
- `Height` 和 `Width` 可将该值设置为星号，这意味着，网格会设置这些值以填充可用空间。
- 每个按钮的位置由指定 `Grid.Row` & `Grid.Column` 属性。

LayoutOptions

`LayoutOptions` 结构可用于定义对齐方式和视图，相对于其父级的扩展。

边距和填充

`Margin` 并 `Padding` 元素呈现用户界面中时，属性控制布局行为。

输入的透明度

每个元素有 `InputTransparent` 用于定义该元素是否接收输入的属性。其默认值是 `false`，确保元素接收输入。

当容器类，如布局类，其值传输到子元素上设置此属性。因此，设置 `InputTransparent` 属性设置为 `true` 布局类将导致不接收输入布局中的元素。

设备方向

Xamarin.Forms 和其内置布局是能够处理设备方向中的更改。请考虑您的应用程序将支持，以及如何将使哪个方向的横向和纵向模式中提供的空间使用。

对于平板电脑和桌面应用程序的布局

iOS、Android 和通用 Windows 平台支持更大的屏幕大小上的所有适用于平板电脑设备（以及便携式计算机和 Windows 的桌面）。使用 Xamarin.Forms，可以通过检测设备类型和调整页面布局中，或使用完全不同的页面完全用于较大的屏幕来优化您的应用程序的较大的屏幕。

创建自定义布局

Xamarin.Forms 定义了四个布局类- `StackLayout`， `AbsoluteLayout`， `RelativeLayout`，并 `Grid`，和每个不同的方式排列子项。但是，有时通过 Xamarin.Forms 进行必要的组织不使用的布局页内容提供。本文介绍如何编写一个自定义布局的类，并说明了方向区分 `WrapLayout` 类跨页上，水平排列子项，然后将包装对其他行的后续子级的显示。

布局压缩

布局压缩从可视化树中删除指定的布局，以试图提升页面呈现性能。这带来的性能优势因页面复杂性、要使用的操作系统版本以及运行应用的设备而异。不过，在旧设备上实现的性能提升最大。

使您的选择

请注意，在大多数情况下，多个布局选项可用于实现您所需的设计。当存在多个有效的选项时，请考虑哪种方法将是最适合你的情况。大多数设计无法实现与只是一种布局，因此需要创建更复杂的设计作为嵌套布局。

相关链接

- Apple 的人机接口指南
- Android 设计网站
- 布局（示例）
- BusinessTumble 示例（示例）

Xamarin.Forms StackLayout

2018/6/9 • • Edit Online

`StackLayout` 将组织的一维的行（"堆栈"）中的视图水平或垂直。视图中 `StackLayout` 可以基于使用布局选项的布局中的空间大小。定位由视图添加到的布局和视图的布局选项的顺序确定。

目标

`StackLayout` 并不太复杂比其他视图。可以通过只需添加到视图中创建简单的线性接口 `StackLayout`，和由嵌套它们创建更复杂的接口。

使用情况和行为

间距

默认情况下，`StackLayout` 将添加视图之间的 6px 边距。这可以控制或设置通过设置有无边距 `Spacing` StackLayout 上的属性。下面演示了如何设置间距和不同间距选项的效果：

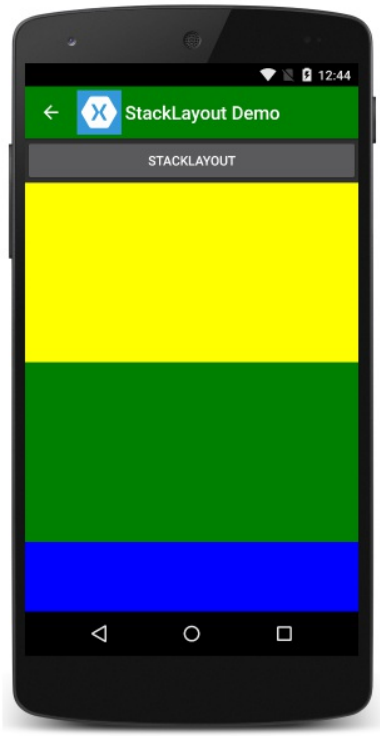
在 XAML 中：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="LayoutSamples.StackLayoutDemo"
Title="StackLayout Demo">
  <ContentPage.Content>
    <StackLayout Spacing="10" x:Name="layout">
      <Button Text="StackLayout" VerticalOptions="Start"
        HorizontalOptions="FillAndExpand" />
      <BoxView Color="Yellow" VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
          <BoxView Color="Green" VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
          <BoxView HeightRequest="75" Color="Blue" VerticalOptions="End"
        HorizontalOptions="FillAndExpand" />
    </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在 C# 中：

```
public class StackLayoutCode : ContentPage
{
  public StackLayoutCode ()
  {
    var layout = new StackLayout ();
    var button = new Button { Text = "StackLayout", VerticalOptions = LayoutOptions.Start,
     HorizontalOptions = LayoutOptions.FillAndExpand };
    var yellowBox = new BoxView { Color = Color.Yellow, VerticalOptions = LayoutOptions.FillAndExpand,
HorizontalOptions = LayoutOptions.FillAndExpand };
    var greenBox = new BoxView { Color = Color.Green, VerticalOptions = LayoutOptions.FillAndExpand,
HorizontalOptions = LayoutOptions.FillAndExpand };
    var blueBox = new BoxView { Color = Color.Blue, VerticalOptions = LayoutOptions.FillAndExpand,
      HorizontalOptions = LayoutOptions.FillAndExpand, HeightRequest = 75 };

    layout.Children.Add(button);
    layout.Children.Add(yellowBox);
    layout.Children.Add(greenBox);
    layout.Children.Add(blueBox);
    layout.Spacing = 10;
    Content = layout;
  }
}
```

间距 = 0:



间距的 10 个:

大小调整

StackLayout 中的视图的大小取决于高度和宽度请求和布局选项。`StackLayout` 将强制执行填充。以下 `LayoutOption`s 将导致视图以占用所从布局可用空间：

- **CenterAndExpand** –布局中的将视图中心和扩展以占用所布局将为其提供空间。
- **EndAndExpand** –布局（底部或最右侧的边界）的末尾放置视图，并且扩展以占用所布局将为其提供空间。
- **FillAndExpand** –放置视图，以便其不填充和占用所布局将为其提供空间。
- **StartAndExpand** –布局的开头将视图并占用所父级将提供空间。

有关详细信息，请参阅扩展。

定位

中 StackLayout 视图可以定位并调整其大小使用 `LayoutOptions` 。可以指定每个视图 `VerticalOptions` 和 `HorizontalOptions`，定义如何在视图将确定自己的位置相对于布局。以下预定义 `LayoutOptions` 可用：

- **Center** –布局中的将视图中心。
- **结束**–布局（底部或最右侧的边界）的末尾放置视图。
- **填充**–放置视图，以便它具有不填充。
- **启动**–放置布局开头的视图。

下面的代码演示设置布局选项：

在 XAML 中：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="LayoutSamples.StackLayoutDemo"
Title="StackLayout Demo">
  <ContentPage.Content>
    <StackLayout x:Name="layout">
      <Button VerticalOptions="Start"
        HorizontalOptions="FillAndExpand" />
      <BoxView VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
            <BoxView VerticalOptions="FillAndExpand"
        HorizontalOptions="FillAndExpand" />
            <BoxView HeightRequest="75" VerticalOptions="End"
        HorizontalOptions="FillAndExpand" />
    </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在 C# 中：

```
public class StackLayoutCode : ContentPage
{
  public StackLayoutCode ()
  {
    var layout = new StackLayout ();
    var button = new Button { VerticalOptions = LayoutOptions.Start,
     HorizontalOptions = LayoutOptions.FillAndExpand };
    var oneBox = new BoxView { VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
LayoutOptions.FillAndExpand };
    var twoBox = new BoxView {  VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
LayoutOptions.FillAndExpand };
    var threeBox = new BoxView {  VerticalOptions = LayoutOptions.FillAndExpand, HorizontalOptions =
LayoutOptions.FillAndExpand };

    layout.Children.Add(button);
    layout.Children.Add(oneBox);
    layout.Children.Add(twoBox);
    layout.Children.Add(threeBox);
    Content = layout;
  }
}
```

有关详细信息，请参阅对齐。

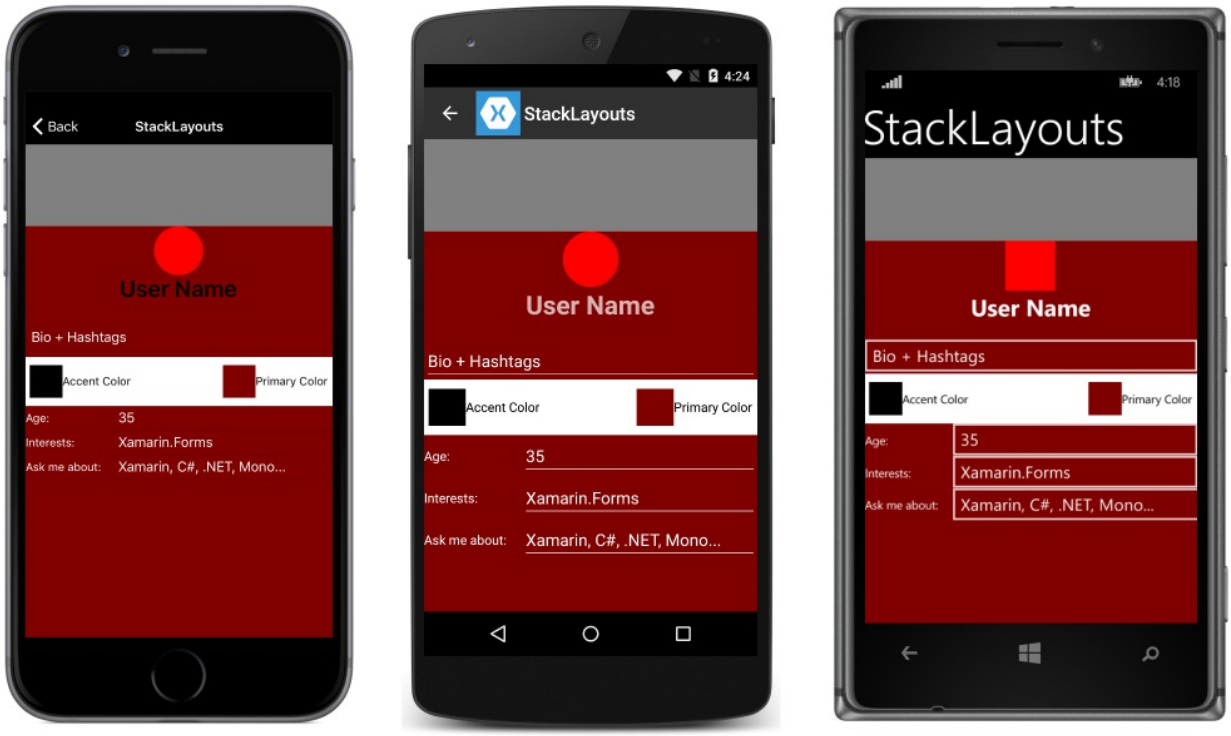
浏览复杂布局

每个布局有优点和缺点为特定布局的创建。在这一系列的布局文章，整个示例应用程序已创建具有相同的页面布局实现使用三种不同的布局。

请考虑下面的 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TheBusinessTumble.StackLayoutPage"
BackgroundColor="Maroon"
Title="StackLayouts">
    <ContentPage.Content>
    <ScrollView>
      <StackLayout Spacing="0" Padding="0" BackgroundColor="Maroon">
        <BoxView HorizontalOptions="FillAndExpand" HeightRequest="100"
          VerticalOptions="Start" Color="Gray" />
        <Button BorderRadius="30" HeightRequest="60" WidthRequest="60"
          BackgroundColor="Red" HorizontalOptions="Center" VerticalOptions="Start" />
        <StackLayout HeightRequest="100" VerticalOptions="Start" HorizontalOptions="FillAndExpand"
Spacing="20" BackgroundColor="Maroon">
          <Label Text="User Name" FontSize="28" HorizontalOptions="Center"
            VerticalOptions="Center" FontAttributes="Bold" />
          <Entry Text="Bio + Hashtags" TextColor="White"
            BackgroundColor="Maroon" HorizontalOptions="FillAndExpand" VerticalOptions="CenterAndExpand" />
        </StackLayout>
        <StackLayout Orientation="Horizontal" HeightRequest="50" BackgroundColor="White" Padding="5">
          <StackLayout Spacing="0" BackgroundColor="White" Orientation="Horizontal"
HorizontalOptions="Start">
            <BoxView BackgroundColor="Black" WidthRequest="40" HeightRequest="40"
HorizontalOptions="StartAndExpand" VerticalOptions="Center" />
            <Label FontSize="14" TextColor="Black" Text="Accent Color" HorizontalOptions="StartAndExpand"
VerticalOptions="Center" />
          </StackLayout>
          <StackLayout Spacing="0" BackgroundColor="White" Orientation="Horizontal"
HorizontalOptions="EndAndExpand">
            <BoxView BackgroundColor="Maroon" WidthRequest="40" HeightRequest="40" HorizontalOptions="Start"
VerticalOptions="Center" />
            <Label FontSize="14" TextColor="Black" Text="Primary Color" HorizontalOptions="StartAndExpand"
VerticalOptions="Center" />
          </StackLayout>
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Age:" TextColor="White" HorizontalOptions="Start"
VerticalOptions="Center" WidthRequest="100" />
          <Entry HorizontalOptions="FillAndExpand" Text="35" TextColor="White" BackgroundColor="Maroon" />
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Interests:" TextColor="White"
          HorizontalOptions="Start" VerticalOptions="Center" WidthRequest="100" />
          <Entry HorizontalOptions="FillAndExpand" Text="Xamarin.Forms" TextColor="White"
BackgroundColor="Maroon" />
        </StackLayout>
        <StackLayout Orientation="Horizontal">
          <Label FontSize="14" Text="Ask me about:" TextColor="White"
          HorizontalOptions="Start" VerticalOptions="Center" WidthRequest="100"/>
          <Entry HorizontalOptions="FillAndExpand" Text="Xamarin, C#, .NET, Mono..." TextColor="White"
BackgroundColor="Maroon" />
        </StackLayout>
      </StackLayout>
    </ScrollView>
    </ContentPage.Content>
</ContentPage>
```

上面的代码将导致以下布局:



请注意，`StackLayouts`s 嵌套的因为在某些情况下嵌套布局可以更方便地比提供相同的布局中的所有元素。另请注意，因为`StackLayout`不支持重叠项，找到的页面不具有布局激动人心的一些其他布局的页中。

相关链接

- LayoutOptions
- 布局（示例）
- BusinessTumble 示例（示例）

Xamarin.Forms AbsoluteLayout

2018/7/13 • • Edit Online

`AbsoluteLayout` 定位并调整其自身的大小和位置或绝对值成比例的子元素的大小。子视图可能定位和调整大小成比例的值或静态值，使用和成比例，可以混合使用静态值。

本文将介绍：

- **目的** –的常见用途 `AbsoluteLayout` 。
- **使用情况** –如何使用 `AbsoluteLayout` 来实现您所需的设计。
  - **按比例的布局** –了解如何按比例值即可在 `AbsoluteLayout` 。
  - **指定值** –了解如何指定比例和绝对值。
  - **按比例值** –了解如何按比例值起作用。
    - **绝对值** –了解绝对值的工作原理。

目标

由于定位模型的 `AbsoluteLayout` ，布局可以相对简单，来定位元素，以便它们与布局的任意一侧对齐或居中。按比例大小和位置中的元素与 `AbsoluteLayout` 可以自动扩展到任何视图大小。对于仅的位置，但不是大小应缩放的项，可以混合绝对和比例值。

`AbsoluteLayout` 可以使用任何位置元素需要视图中定位和对齐到边缘的元素时尤其有用。

用法

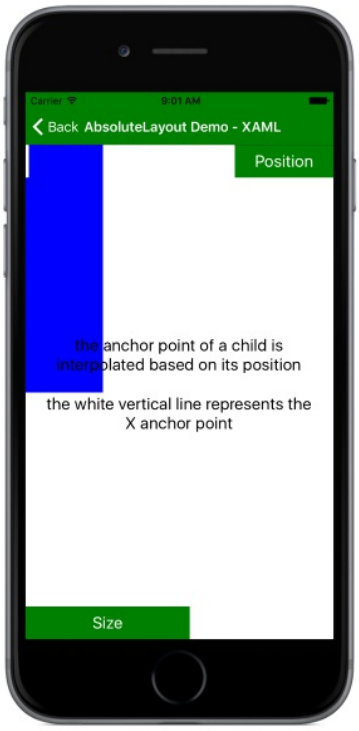
按比例的布局

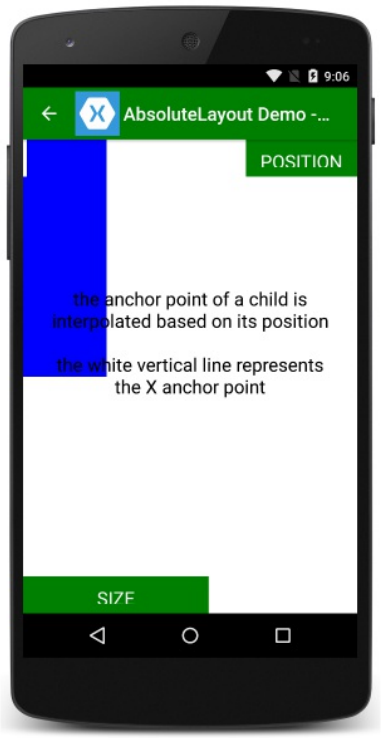
`AbsoluteLayout` 具有唯一的定位点模型，由此元素的定位点位于相对于其元素按比例定位使用时，相对于布局放置的元素。使用绝对定位时，定位点为 (0，0) 的视图中。这样做有两个重要的结果：

- 元素不能定位关闭屏幕使用比例值。
- 元素可进行可靠地定位沿任意一侧的布局或中心，而不考虑设备的布局的大小。

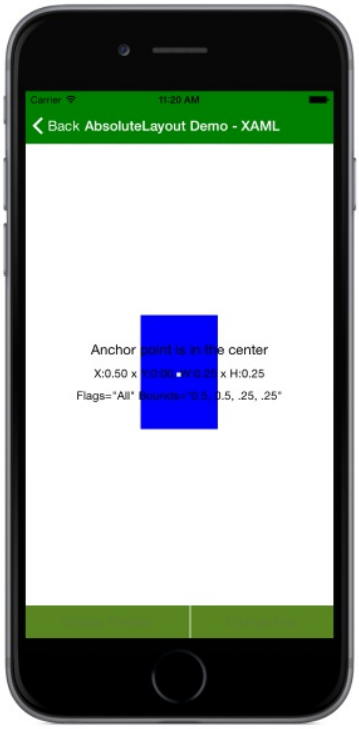
`AbsoluteLayout` 如 `RelativeLayout` ，可以定位元素，使它们相互重叠。

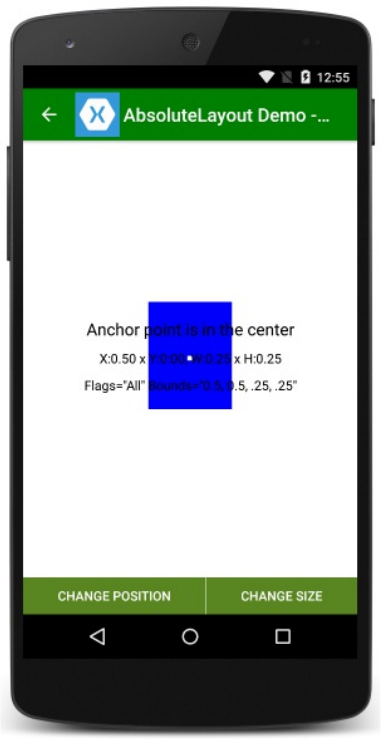
请注意，在以下屏幕截图中，定位点的框为白色点。布局中移动，请注意定位点与 box 之间的关系：

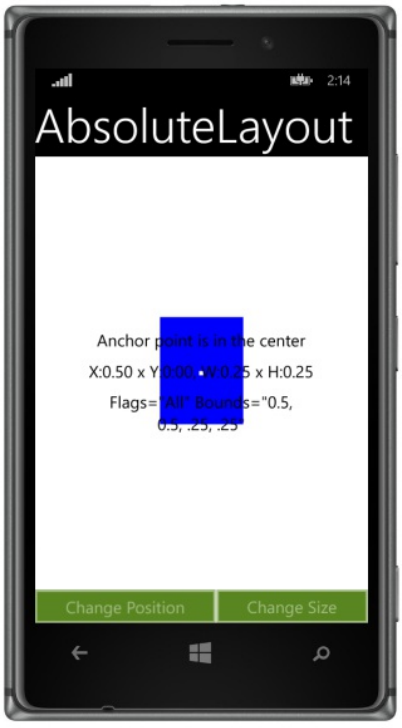

Carrier
9:01 AM
Back AbsoluteLayout Demo - XAML
Position
the anchor point of a child is interpolated based on its position
the white vertical line represents the X anchor point
Size


9:06
AbsoluteLayout Demo -...
POSITION
the anchor point of a child is interpolated based on its position
the white vertical line represents the X anchor point
SIZE


9:42
AbsoluteLayout
Position
the anchor point of a child is interpolated based on its position
the white vertical line represents the X anchor point
Size


Carrier
11:20 AM
Back AbsoluteLayout Demo - XAML
Anchor point is in the center
X:0.50 x Y:0.00 W:0.25 x H:0.25
Flags="All" Bounds="0.5, 0.5, .25, .25"


12:55
AbsoluteLayout Demo -...
Anchor point is in the center
X:0.50 x Y:0.00 W:0.25 x H:0.25
Flags="All" Bounds="0.5, 0.5, .25, .25"
CHANGE POSITION
CHANGE SIZE


2:14
AbsoluteLayout
Anchor point is in the center
X:0.50 x Y:0.00 W:0.25 x H:0.25
Flags="All" Bounds="0.5, 0.5, .25, .25"
Change Position
Change Size

指定值

中的视图`AbsoluteLayout`位于使用四个值：

- **X** –视图的锚点的 x（水平）位置
- **Y** –视图的锚点的 y（垂直）位置
- **宽度**–视图的宽度
- **高度**–视图的高度

每个这些值可设置为比例值或绝对值。

值指定为边界和标志的组合。`LayoutBounds` 是 `Rectangle` 包含以下四个值：`x`，`y`，`width`，`height`。

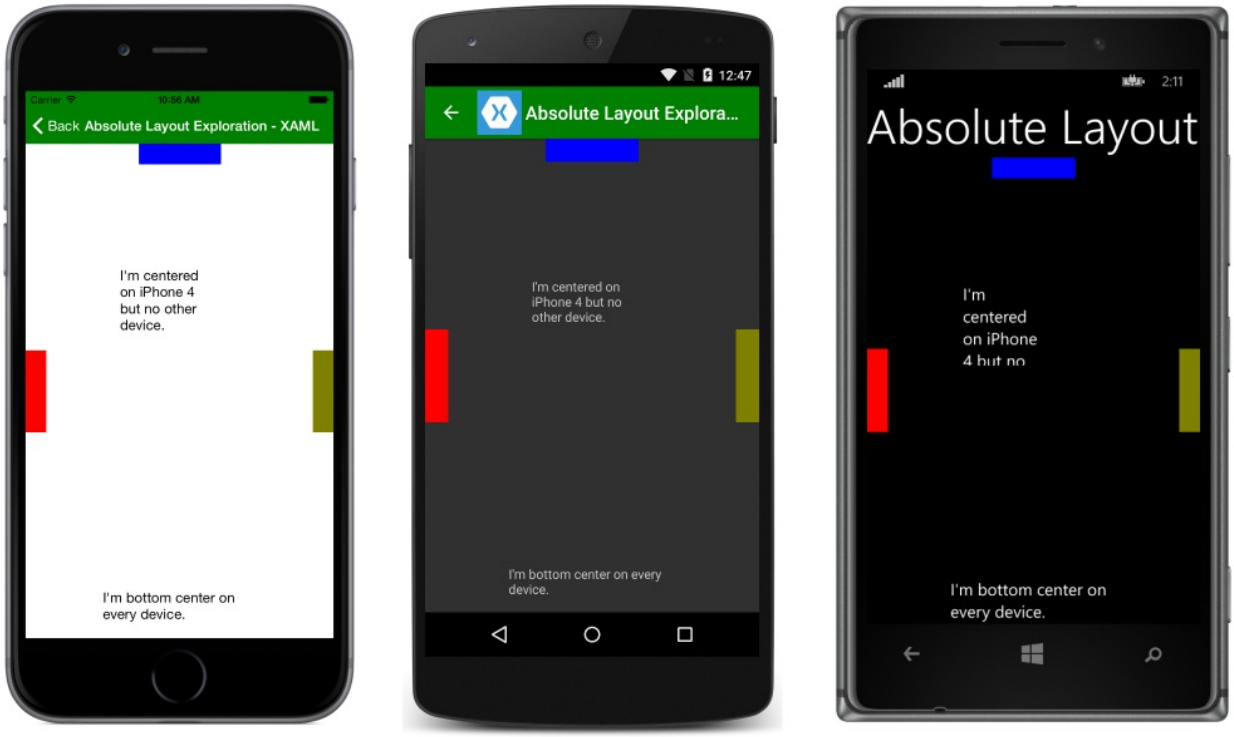
AbsoluteLayoutFlags

`AbsoluteLayoutFlags` 指定将如何解释值并提供以下预定义的选项：

- **无**–将所有值都解释为绝对值。如果不指定了任何布局标志，这是默认值。
- **所有**–将解释为比例的所有值。
- **WidthProportional** –解释`Width`值作为比例和所有其他值为绝对值。
- **HeightProportional** –仅高度会将该值解释为比例绝对所有其他值。
- **XProportional** –解释`X`值作为成比例，同时将所有其他值视为绝对值。
- **YProportional** –解释`Y`值作为成比例，同时将所有其他值视为绝对值。
- **PositionProportional** –解释`X`和`Y`值作为成比例，而大小值解释为绝对值。
- **SizeProportional** –解释`Width`和`Height`绝对位置值时根据成比例的值。

在 XAML，边界和标志设置为布局中的视图定义的一部分使用`AbsoluteLayout.LayoutBounds`属性。边界设置为一列以逗号分隔的值，`X`，`Y`，`Width`，和`Height`按顺序。在布局中使用的视图的声明中还指定标志`AbsoluteLayout.LayoutFlags`属性。请注意，可以在 XAML 使用逗号分隔的列表中组合标志。请看下面的示例：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="LayoutSamples.AbsoluteLayoutExploration"
Title="Absolute Layout Exploration">
    <ContentPage.Content>
        <AbsoluteLayout>
            <Label Text="I'm centered on iPhone 4 but no other device"
                AbsoluteLayout.LayoutBounds="115,150,100,100" LineBreakMode="WordWrap"  />
            <Label Text="I'm bottom center on every device."
                AbsoluteLayout.LayoutBounds=".5,1,.5,.1" AbsoluteLayout.LayoutFlags="All"
                LineBreakMode="WordWrap"  />
            <BoxView Color="Olive"  AbsoluteLayout.LayoutBounds="1,.5, 25, 100"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
            <BoxView Color="Red" AbsoluteLayout.LayoutBounds="0,.5,25,100"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
            <BoxView Color="Blue" AbsoluteLayout.LayoutBounds=".5,0,100,25"
                AbsoluteLayout.LayoutFlags="PositionProportional" />
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>
```



请注意以下事项：

- 在中心标签将置于使用绝对大小和位置的值。正因为如此，它会显示在 iPhone 4 秒上居中和较低，但不是在较大的设备上居中。
- 在布局的底部文本位于使用比例大小和位置的值。它将始终显示在底部中心的布局，但其大小将增长的布局越大。
- 三个彩色 `BoxView` s 定位在使用按比例的位置和绝对大小在屏幕上、左和右边缘。

以下可实现在 C# 中相同的布局：

```
public class AbsoluteLayoutExplorationCode : ContentPage
{
    public AbsoluteLayoutExplorationCode ()
    {
        Title = "Absolute Layout Exploration - Code";
        var layout = new AbsoluteLayout();

        var centerLabel = new Label {
        Text = "I'm centered on iPhone 4 but no other device.",
        LineBreakMode = LineBreakMode.WordWrap};

        AbsoluteLayout.SetLayoutBounds (centerLabel, new Rectangle (115, 159, 100, 100));
        // No need to set layout flags, absolute positioning is the default

        var bottomLabel = new Label { Text = "I'm bottom center on every device.", LineBreakMode =
LineBreakMode.WordWrap };
        AbsoluteLayout.SetLayoutBounds (bottomLabel, new Rectangle (.5, 1, .5, .1));
        AbsoluteLayout.SetLayoutFlags (bottomLabel, AbsoluteLayoutFlags.All);

        var rightBox = new BoxView{ Color = Color.Olive };
        AbsoluteLayout.SetLayoutBounds (rightBox, new Rectangle (1, .5, 25, 100));
        AbsoluteLayout.SetLayoutFlags (rightBox, AbsoluteLayoutFlags.PositionProportional);

        var leftBox = new BoxView{ Color = Color.Red };
        AbsoluteLayout.SetLayoutBounds (leftBox, new Rectangle (0, .5, 25, 100));
        AbsoluteLayout.SetLayoutFlags (leftBox, AbsoluteLayoutFlags.PositionProportional);

        var topBox = new BoxView{ Color = Color.Blue };
        AbsoluteLayout.SetLayoutBounds (topBox, new Rectangle (.5, 0, 100, 25));
        AbsoluteLayout.SetLayoutFlags (topBox, AbsoluteLayoutFlags.PositionProportional);

        layout.Children.Add (bottomLabel);
        layout.Children.Add (centerLabel);
        layout.Children.Add (rightBox);
        layout.Children.Add (leftBox);
        layout.Children.Add (topBox);

        Content = layout;
    }
}
```

按比例值

按比例值定义布局和视图之间的关系。此关系定义为父布局的相应值的一定比例子视图的位置或缩放值。这些值表示为 `double` 与 0 和 1 之间的值。

位置和大小视图布局中的，将使用按比例值。因此，当视图的宽度设置为一定比例，生成的宽度值是乘以比例 `AbsoluteLayout` 的宽度。例如，对于 `AbsoluteLayout` 宽度的 `500` 并且将 250 (500 x.5 被设置为.5，该视图的呈现宽度的比例宽度的视图。

若要使用按比例的值，请设置 `LayoutBounds` 使用 (x, y) 比例和成比例的大小，然后设置 `LayoutFlags` 到 `All` 。

在 XAML:

```
<Label Text="I'm bottom center on every device."
  AbsoluteLayout.LayoutBounds=".5,1,.5,.1" AbsoluteLayout.LayoutFlags="All"  />
```

在 C# 中:

```
var label = new Label {Text = "I'm bottom center on every device."};
AbsoluteLayout.SetLayoutBounds(label, new Rectangle(.5,1,.5,.1));
AbsoluteLayout.SetLayoutFlags(label, AbsoluteLayoutFlags.All);
```

绝对值

绝对值显式定义视图布局中的放置位置。按比例值与绝对值是能够定位和调整大小不适合布局的边界内的视图。

布局的大小未知时，用于定位使用绝对值可能很危险。在使用绝对位置，在一个大小的屏幕的中心中的元素无法在以任何其他大小偏移量。请务必跨不同屏幕大小的受支持设备测试您的应用程序。

若要使用绝对布局值，请设置 `LayoutBounds` 使用 (x, y) 坐标和显式大小，然后设置 `LayoutFlags` 到 `None` 。

在 XAML:

```
<Label Text="I'm centered on iPhone 4 but no other device."
  AbsoluteLayout.LayoutBounds="115,150,100,100"  />
```

在 C# 中:

```
var label = new Label {Text = "I'm centered on iPhone 4 but no other device."};
AbsoluteLayout.SetLayoutBounds(label, new Rectangle(115,150,100,100));
```

浏览复杂的布局

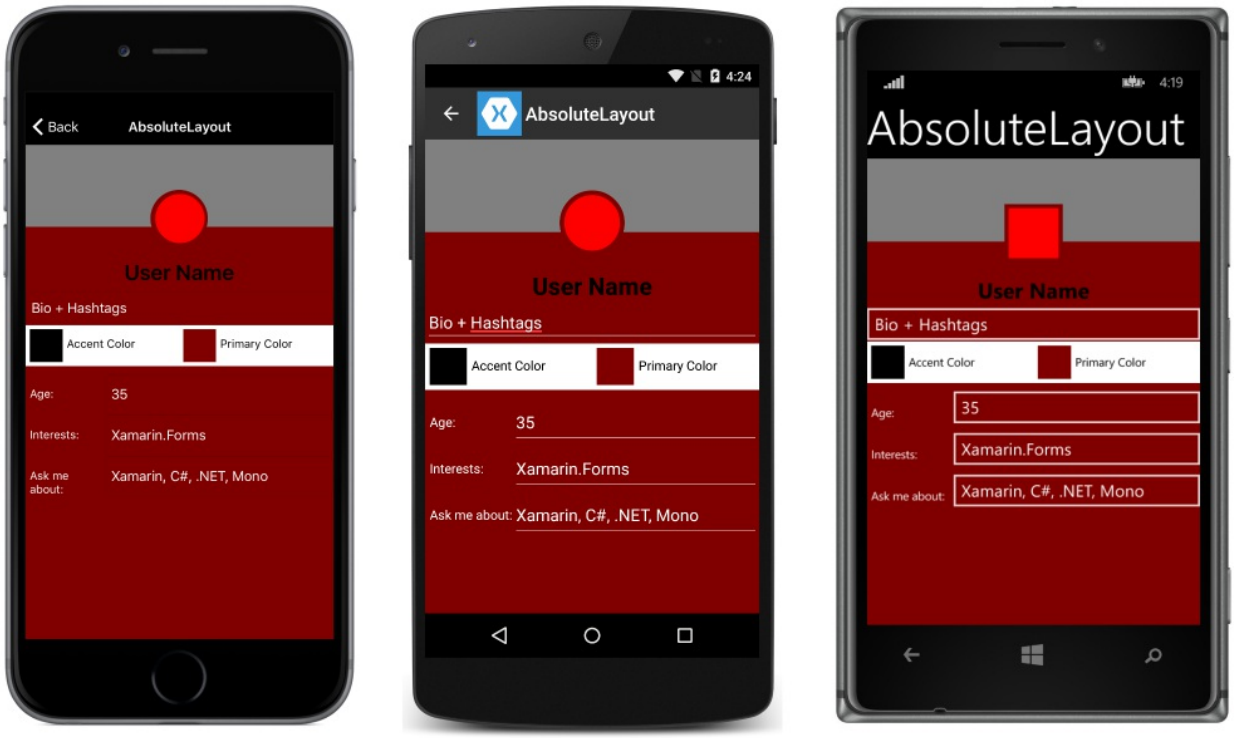
每个布局具有的优势和劣势为特定布局的创建。在本系列的布局文章，整个示例应用程序已创建具有相同的页面布局使用三个不同的布局实现。

请考虑以下 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TheBusinessTumble.AbsoluteLayoutPage"
Title="AbsoluteLayout">
    <ContentPage.ToolbarItems>
        <ToolbarItem Text="Save" />
    </ContentPage.ToolbarItems>
    <ContentPage.Content>
        <ScrollView>
            <AbsoluteLayout BackgroundColor="Maroon">
                <BoxView BackgroundColor="Gray" AbsoluteLayout.LayoutBounds="0
                    0,1,100" AbsoluteLayout.LayoutFlags="XProportional,YProportional,WidthProportional" />
                <Button BackgroundColor="Maroon"
                    AbsoluteLayout.LayoutBounds=".5,55,70,70" AbsoluteLayout.LayoutFlags="XProportional"
BorderRadius="35" />
                <Button BackgroundColor="Red" AbsoluteLayout.LayoutBounds=".5
                    60,60,60" AbsoluteLayout.LayoutFlags="XProportional" BorderRadius="30" />
                <Label Text="User Name" FontAttributes="Bold" FontSize="26"
                    TextColor="Black" HorizontalTextAlignment="Center"
AbsoluteLayout.LayoutBounds=".5,140,1,40" AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
                <Entry Text="Bio + Hashtags" TextColor="White"
                    BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds=".5,180,1,40"
AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
                <AbsoluteLayout BackgroundColor="White"
                        AbsoluteLayout.LayoutBounds="0, 220, 1, 50"
AbsoluteLayout.LayoutFlags="XProportional,WidthProportional">
                    <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,0,.5,1"
AbsoluteLayout.LayoutFlags="WidthProportional,HeightProportional">
                        <Button BackgroundColor="Black" BorderRadius="20"
                            AbsoluteLayout.LayoutBounds="5,.5,40,40"
AbsoluteLayout.LayoutFlags="YProportional" />
                        <Label Text="Accent Color" TextColor="Black"
                            AbsoluteLayout.LayoutBounds="50,.55,1,25"
AbsoluteLayout.LayoutFlags="YProportional,WidthProportional" />
                    </AbsoluteLayout>
                    <AbsoluteLayout AbsoluteLayout.LayoutBounds="1,0,.5,1"
```

```
AbsoluteLayout.LayoutFlags="WidthProportional,HeightProportional,XProportional">
                        <Button BackgroundColor="Maroon" BorderRadius="20"
                            AbsoluteLayout.LayoutBounds="5,.5,40,40"
AbsoluteLayout.LayoutFlags="YProportional" />
                        <Label Text="Primary Color" TextColor="Black"
                            AbsoluteLayout.LayoutBounds="50,.55,1,25"
AbsoluteLayout.LayoutFlags="YProportional,WidthProportional" />
                    </AbsoluteLayout>
                </AbsoluteLayout>
                <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,270,1,50"
AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
                    <Label Text="Age:" TextColor="White"
                        AbsoluteLayout.LayoutBounds="0,25,.25,50"
AbsoluteLayout.LayoutFlags="WidthProportional" />
                    <Entry Text="35" TextColor="White" BackgroundColor="Maroon"
                        AbsoluteLayout.LayoutBounds="1,10,.75,50"
AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
                </AbsoluteLayout>
                <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,320,1,50"
AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
                    <Label Text="Interests:" TextColor="White"
                        AbsoluteLayout.LayoutBounds="0,25,.25,50"
AbsoluteLayout.LayoutFlags="WidthProportional" />
                    <Entry Text="Xamarin.Forms" TextColor="White"
                        BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds="1,10,.75,50"
AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
                </AbsoluteLayout>
                <AbsoluteLayout AbsoluteLayout.LayoutBounds="0,370,1,50"
AbsoluteLayout.LayoutFlags="WidthProportional" Padding="5,0,0,0">
                    <Label Text="Ask me about:" TextColor="White"
                        AbsoluteLayout.LayoutBounds="0,25,.25,50"
AbsoluteLayout.LayoutFlags="WidthProportional" />
                    <Entry Text="Xamarin, C#, .NET, Mono" TextColor="White"
                        BackgroundColor="Maroon" AbsoluteLayout.LayoutBounds="1,10,.75,50"
AbsoluteLayout.LayoutFlags="XProportional,WidthProportional" />
                </AbsoluteLayout>
            </AbsoluteLayout>
        </ScrollView>
    </ContentPage.Content>
</ContentPage>
```

上面的代码会导致以下布局：



请注意，`AbsoluteLayout`s 嵌套的因为在某些情况下嵌套布局可能会比提供相同的布局中的所有元素。

相关链接

- 借助 Xamarin.Forms，第 14 章创建移动应用
- AbsoluteLayout
- 布局（示例）
- BusinessTumble 示例（示例）

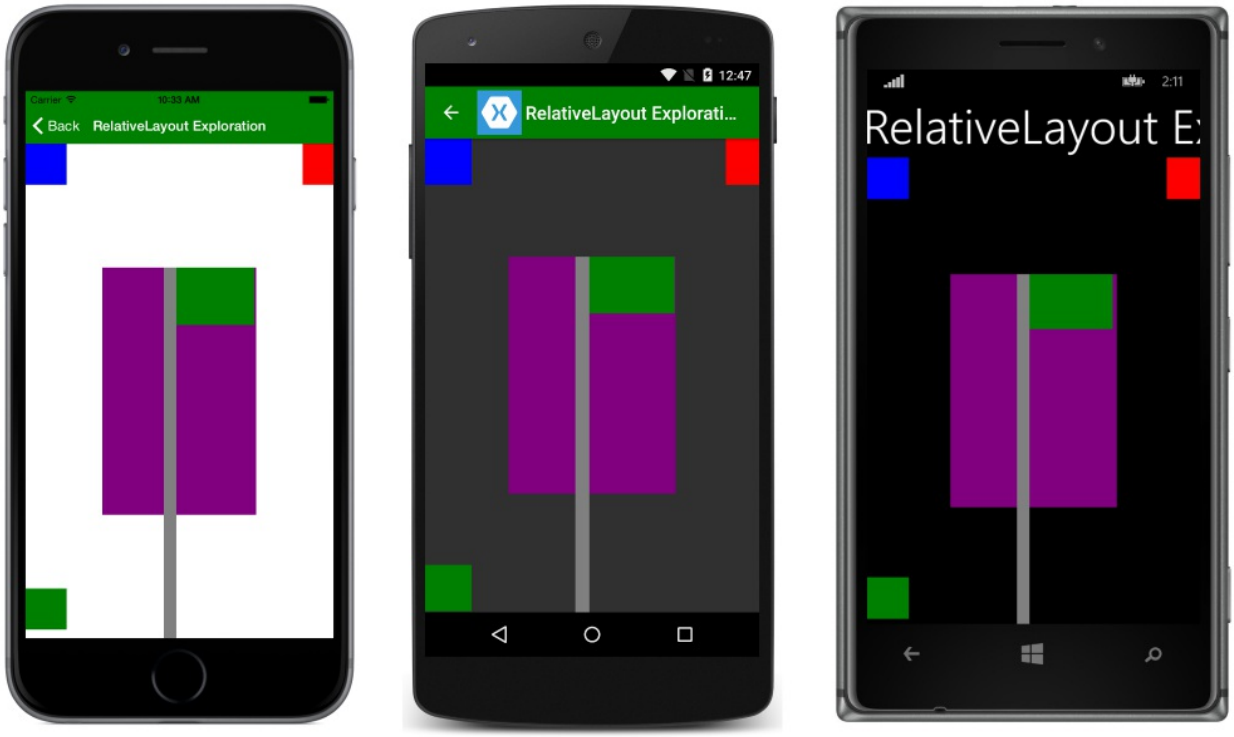
Xamarin.Forms RelativeLayout

2018/6/9 • • Edit Online

`RelativeLayout` 将使用位置和大小视图相对于布局或同级视图的属性。与不同 `AbsoluteLayout`，`RelativeLayout` 不包含移动的定位点的概念，并且没有提供用于定位元素相对于底部或右边缘的布局的功能。`RelativeLayout` 支持在其自己的边界之外的定位元素。

目标

`RelativeLayout` 可以用于将在屏幕上相对于整体的布局或到其他视图的视图。



用法

了解约束

位置和大小内的某个视图 `RelativeLayout`，可使用约束。约束表达式可以包含以下信息：

- **类型**–约束是相对于父或另一个视图。
- **属性**–要用作基础的约束的属性。
- **因素**–要应用到的属性值的系数。
- **常量**–要用作值的偏移量的值。
- **ElementName** –约束是相对于视图的名称。

在 XAML 中，约束表示为 `ConstraintExpression`s。请看下面的示例：

```
<BoxView Color="Green" WidthRequest="50" HeightRequest="50"
    RelativeLayout.XConstraint =
      "{ConstraintExpression Type=RelativeToParent,
                             Property=Width,
                             Factor=0.5,
                             Constant=-100}"
    RelativeLayout.YConstraint =
      "{ConstraintExpression Type=RelativeToParent,
                             Property=Height,
                             Factor=0.5,
                             Constant=-100}" />
```

在 C# 中，约束表示略有不同，使用视图上的函数，而不是表达式。为布局的变量指定约束 `Add` 方法：

```
layout.Children.Add(box, Constraint.RelativeToParent((parent) =>
    {
      return (.5 * parent.Width) - 100;
    }),
    Constraint.RelativeToParent((parent) =>
    {
        return (.5 * parent.Height) - 100;
    }),
    Constraint.Constant(50), Constraint.Constant(50));
```

请注意上述布局的以下方面：

- `x` 和 `y` 具有其自己的约束指定约束。
- 在 C# 中，相对约束被定义为函数。概念喜欢 `Factor` 不存在，但可以手动实现。
- 该框的 `x` 坐标定义为父代、-100 的半角。
- 该框的 `y` 坐标定义为父代、-100 的半高。

> **NOTE**
> 由于未定义约束的方法，则可以在 C# 不是可以指定与 XAML 中进行更复杂的布局。

这两个上面的示例定义与约束 `RelativeToParent` –，即其值是相对于父元素。它还可定义相对于另一个视图的约束。这允许（向开发人员）更直观的布局，并且可以使布局代码意图更容易发现。

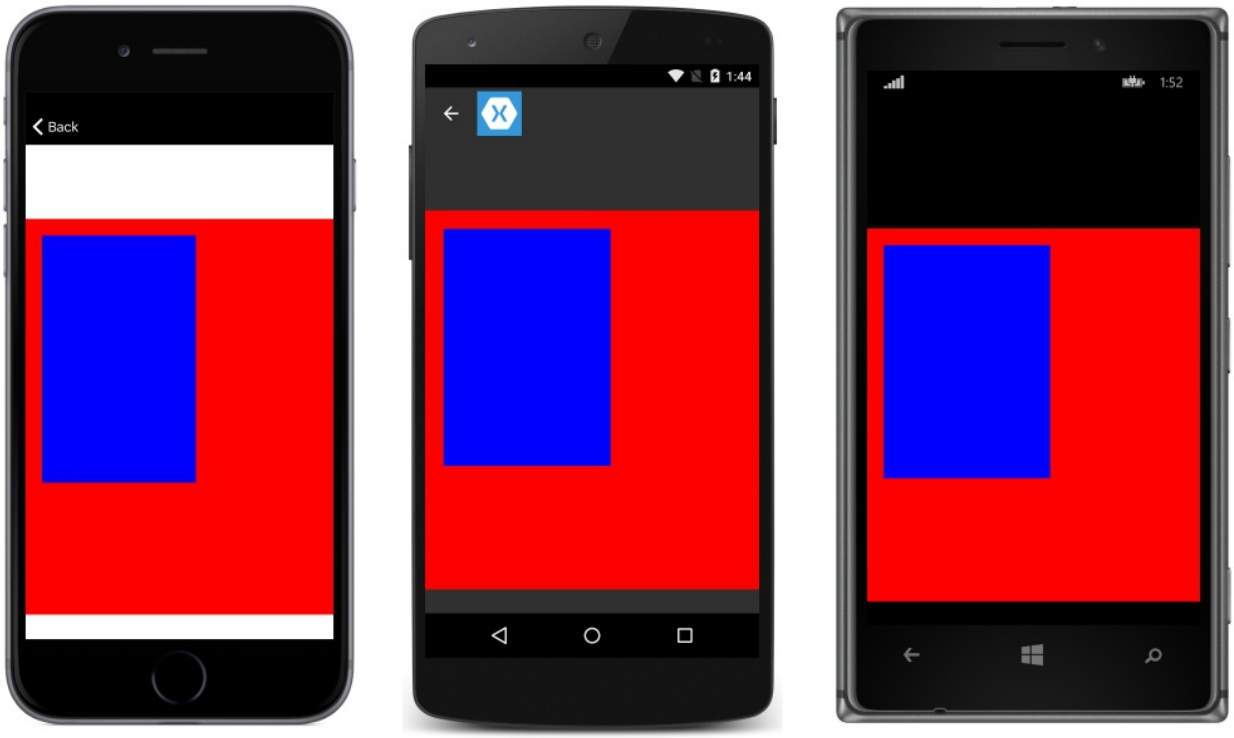
请考虑其中一个元素必须是 20 像素低于另一个布局。如果与常量值定义了两个元素，可能具有较低其 `Y` 约束定义为一个常数，用于为大于 20 像素 `Y` 更高版本的元素的约束。这种方法过程的更高版本的元素位于使用比例，以便像素大小不已知的情况下很短。在这种情况下，限制基于另一个元素的位置的元素是更可靠的：

```
<RelativeLayout>
    <BoxView Color="Red" x:Name="redBox"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
            Property=Height,Factor=.15,Constant=0}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=1,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.8,Constant=0}" />
    <BoxView Color="Blue"
        RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToView,
            ElementName=redBox,Property=Y,Factor=1,Constant=20}"
        RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView,
            ElementName=redBox,Property=X,Factor=1,Constant=20}"
        RelativeLayout.WidthConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Width,Factor=.5,Constant=0}"
        RelativeLayout.HeightConstraint="{ConstraintExpression
            Type=RelativeToParent,Property=Height,Factor=.5,Constant=0}" />
</RelativeLayout>
```

若要完成的布局相同，在 C# 中：

```
layout.Children.Add (redBox, Constraint.RelativeToParent ((parent) => {
        return parent.X;
    }), Constraint.RelativeToParent ((parent) => {
        return parent.Y * .15;
    }), Constraint.RelativeToParent((parent) => {
        return parent.Width;
    }), Constraint.RelativeToParent((parent) => {
        return parent.Height * .8;
    }));
layout.Children.Add (blueBox, Constraint.RelativeToView (redBox, (Parent, sibling) => {
        return sibling.X + 20;
    }), Constraint.RelativeToView (blueBox, (parent, sibling) => {
        return sibling.Y + 20;
    }), Constraint.RelativeToParent((parent) => {
        return parent.Width * .5;
    }), Constraint.RelativeToParent((parent) => {
        return parent.Height * .5;
    }));
```

这将生成以下输出，使用蓝色框中的位置确定_相对_到红色框的位置：



大小调整

视图布局的 `RelativeLayout` 具有两个选项指定其大小：

- `HeightRequest & WidthRequest`
- `RelativeLayout.WidthConstraint` & `RelativeLayout.HeightConstraint`

`HeightRequest` 和 `WidthRequest` 指定的预期的高度和宽度的视图，但根据需要布局可能覆盖。`WidthConstraint` 和 `HeightConstraint` 支持用作相对于布局的或另一个视图的属性值或常量的值设置的高度和宽度。

浏览复杂布局

每个布局有优点和缺点为特定布局的创建。在这一系列的布局文章，整个示例应用程序已创建具有相同的页面布局实现使用三种不同的布局。

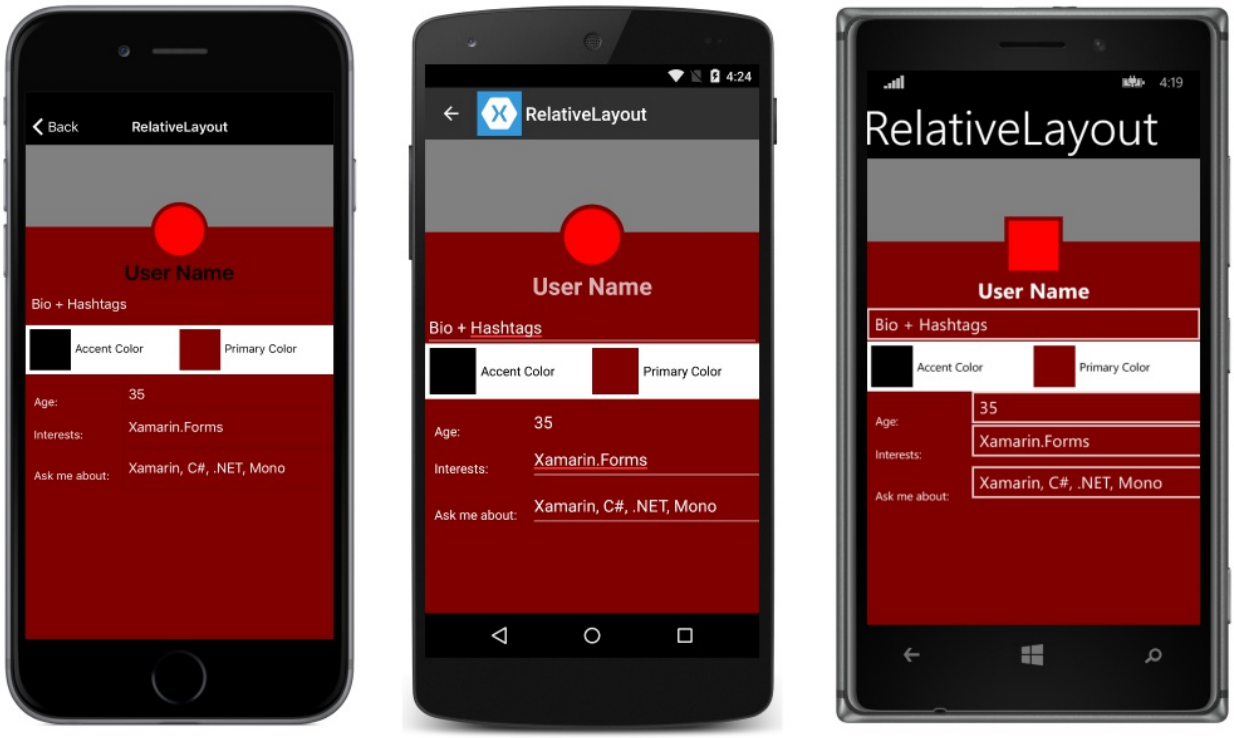
请考虑下面的 XAML：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TheBusinessTumble.RelativeLayoutPage"
BackgroundColor="Maroon"
Title="RelativeLayout">
    <ContentPage.Content>
    <ScrollView>
      <RelativeLayout>
        <BoxView Color="Gray" HeightRequest="100"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=1}" />
        <Button BorderRadius="35" x:Name="imageCircleBack"
            BackgroundColor="Maroon" HeightRequest="70" WidthRequest="70" RelativeLayout.XConstraint="
{ConstraintExpression Type=RelativeToParent,Property=Width, Factor=.5, Constant = -35}"
RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Factor=0, Property=Y, Constant=70}"
/>
        <Button BorderRadius="30" BackgroundColor="Red" HeightRequest="60"
            WidthRequest="60" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToView,
ElementName=imageCircleBack, Property=X, Factor=1,Constant=5}" RelativeLayout.YConstraint="
{ConstraintExpression Type=RelativeToParent, Factor=0, Property=Y, Constant=75}" />
        <Label Text="User Name" FontAttributes="Bold" FontSize="26"
            HorizontalTextAlignment="Center" RelativeLayout.YConstraint="{ConstraintExpression
```

```
Type=RelativeToParent, Property=Y, Factor=0, Constant=140}" RelativeLayout.WidthConstraint="
{ConstraintExpression Type=RelativeToParent, Property=Width, Factor=1}" />
        <Entry Text="Bio + Hashtags" TextColor="White" BackgroundColor="Maroon"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0,
Constant=180}" RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=1}" />
        <RelativeLayout BackgroundColor="White" RelativeLayout.YConstraint="
            {ConstraintExpression Type=RelativeToParent, Property=Y, Factor=0, Constant=220}"
HeightRequest="60" RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=1}" >
            <BoxView BackgroundColor="Black" WidthRequest="50"
                HeightRequest="50" RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Y, Factor=0, Constant=5}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
Property=X, Factor=0, Constant=5}" />
            <BoxView BackgroundColor="Maroon" WidthRequest="50"
                HeightRequest="50" RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Y, Factor=0, Constant=5}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=0.5, Constant=}" />
            <Label FontSize="14" TextColor="Black" Text="Accent Color"
                RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y,
Factor=0, Constant=20}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=X,
Factor=0, Constant=60}" />
            <Label FontSize="14" TextColor="Black" Text="Primary Color"
                RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Y,
Factor=0, Constant=20}" RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width, Factor=0.5, Constant=55}" />
        </RelativeLayout>
        <RelativeLayout Padding="5,0,0,0">
          <Label FontSize="14" Text="Age:" TextColor="White"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=305}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=.25,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
          <Entry Text="35" TextColor="White" BackgroundColor="Maroon"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=280}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.3, Constant=0}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=0.75,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
        </RelativeLayout>
        <RelativeLayout  Padding="5,0,0,0">
          <Label FontSize="14" Text="Interests:" TextColor="White"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=345}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=.25,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
          <Entry Text="Xamarin.Forms" TextColor="White" BackgroundColor="Maroon"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=320}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.3, Constant=0}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=0.75,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
        </RelativeLayout>
        <RelativeLayout  Padding="5,0,0,0">
          <Label FontSize="14" Text="Ask me about:" TextColor="White"
```

```
            LineBreakMode="WordWrap"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=395}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0, Constant=10}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=.25,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
          <Entry Text="Xamarin, C#, .NET, Mono" TextColor="White"
            BackgroundColor="Maroon"
            RelativeLayout.YConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=370}"
            RelativeLayout.XConstraint="{ConstraintExpression Type=RelativeToParent, Property=Width,
Factor=0.3, Constant=0}"
            RelativeLayout.WidthConstraint="{ConstraintExpression Type=RelativeToParent,
Property=Width,Factor=0.75,Constant=0}"
            RelativeLayout.HeightConstraint="{ConstraintExpression Type=RelativeToParent, Property=Height,
Factor=0,Constant=50}" />
        </RelativeLayout>
      </RelativeLayout>
    </ScrollView>
  </ContentPage.Content>
</ContentPage>
```

上面的代码将导致以下布局：



请注意，`RelativeLayouts` 嵌套的因为在某些情况下嵌套布局可以更方便地比提供相同的布局中的所有元素。此外请注意，某些元素均 `RelativeToView`，因为它允许更轻松、更直观布局视图之间的关系指导定位时。

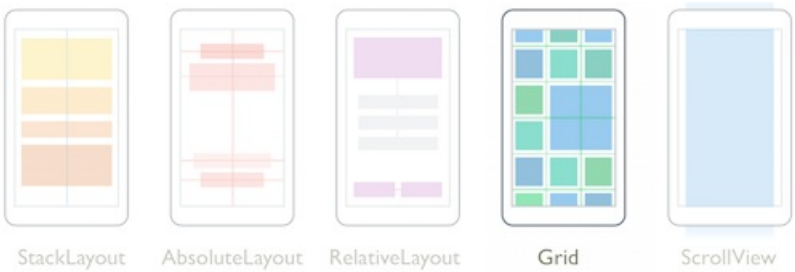
相关链接

- 布局（示例）
- BusinessTumble 示例（示例）

Xamarin.Forms 网格

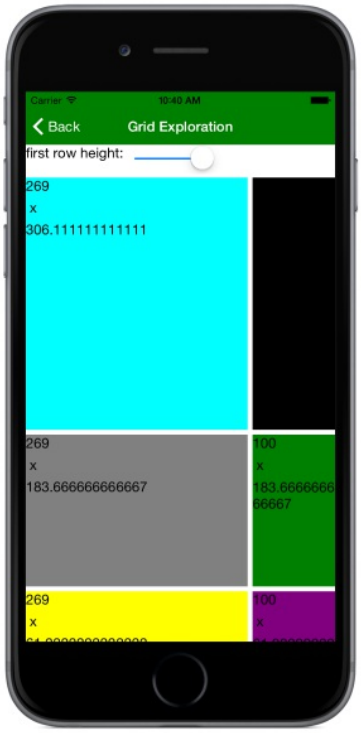
2018/11/2 • • Edit Online

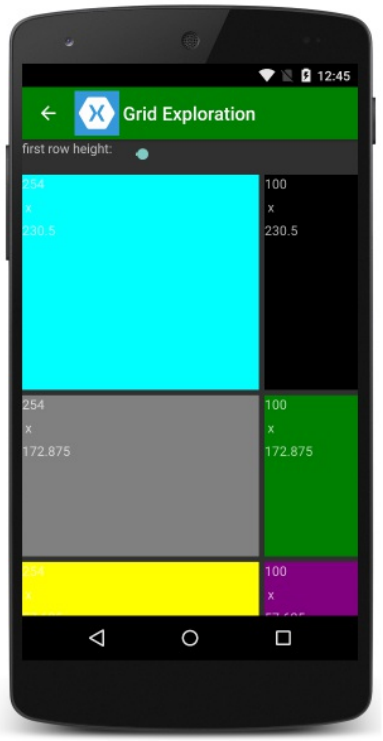
`Grid` 为行和列排列视图的支持。可以设置行和列具有按比例大小或绝对大小。`Grid` 布局不应与传统表混淆和不应显示表格数据。`Grid` 没有行、列或单元格格式的概念。HTML 与表不同，`Grid` 纯粹用于对内容进行布局。



本文将介绍：

- **目的** –的常见用途 `Grid`。
- **使用情况** –如何使用 `Grid` 来实现您所需的设计。
  - **行和列** –指定行和列 `Grid`。
  - **将视图放置** –将视图添加到在网格中的特定行和列。
  - **间距** –配置行和列之间的空格。
  - **Span** –配置跨越多个行或列的元素。





目标

`Grid` 可用于排列到一个网格视图。这是在许多情况下有用：

- 排列计算器应用程序中的按钮
- 在网格中，如 iOS 或 Android 的主屏幕中的排列按钮/选择
- 排列视图，以便可以在一个维度（例如，在某些工具栏）的大小相等的

用法

与传统表不同 `Grid` 无法推断的数量和大小的行和列的内容。相反，`Grid` 已 `RowDefinitions` 和 `ColumnDefinitions` 集合。这些日志包含行数和列布局方式的定义。视图将添加到 `Grid` 使用指定的行和列索引，该标识所属的行和列应置于一个视图。

行和列

行和列的信息存储在 `Grid` 的 `RowDefinitions` & `ColumnDefinitions` 属性，是每个集合的 `RowDefinition` 并 `ColumnDefinition` 对象，分别。`RowDefinition` 只有一个属性， `Height` ，并 `ColumnDefinition` 只有一个属性， `Width` 。高度和宽度的选项如下所示：

- **自动**–自动大小，以适应行或列中的内容。指定作为 `GridUnitType.Auto` C# 中或作为 `Auto` 在 XAML 中。
- **Proportional(*)** –为一定比例的剩余空间大小列和行。指定为值和 `GridUnitType.Star` 用 C# 或作为 `#*` XAML，在使用 `#` 正在所需的值。指定一个行/列与 `*` 会让其以填充可用空间。
- 绝对–调整列和具有特定的固定高度和宽度值的行的大小。指定为值和 `GridUnitType.Absolute` 用 C# 或作为 `#` XAML，在使用 `#` 正在所需的值。

> **NOTE**
> 列的宽度值设置为 '*' 默认情况下，在 Xamarin.Forms 中，可确保该列将填充可用空间。

请考虑的应用程序需要三个行和两个列。底部行必须是完全 200px、高和最上面一行必须是两次高度都是中间的一行。左侧的列需要是足够宽，以适应内容，右侧列需要填充剩余的空间。

在 XAML:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="2*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="200" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

在 C# 中：

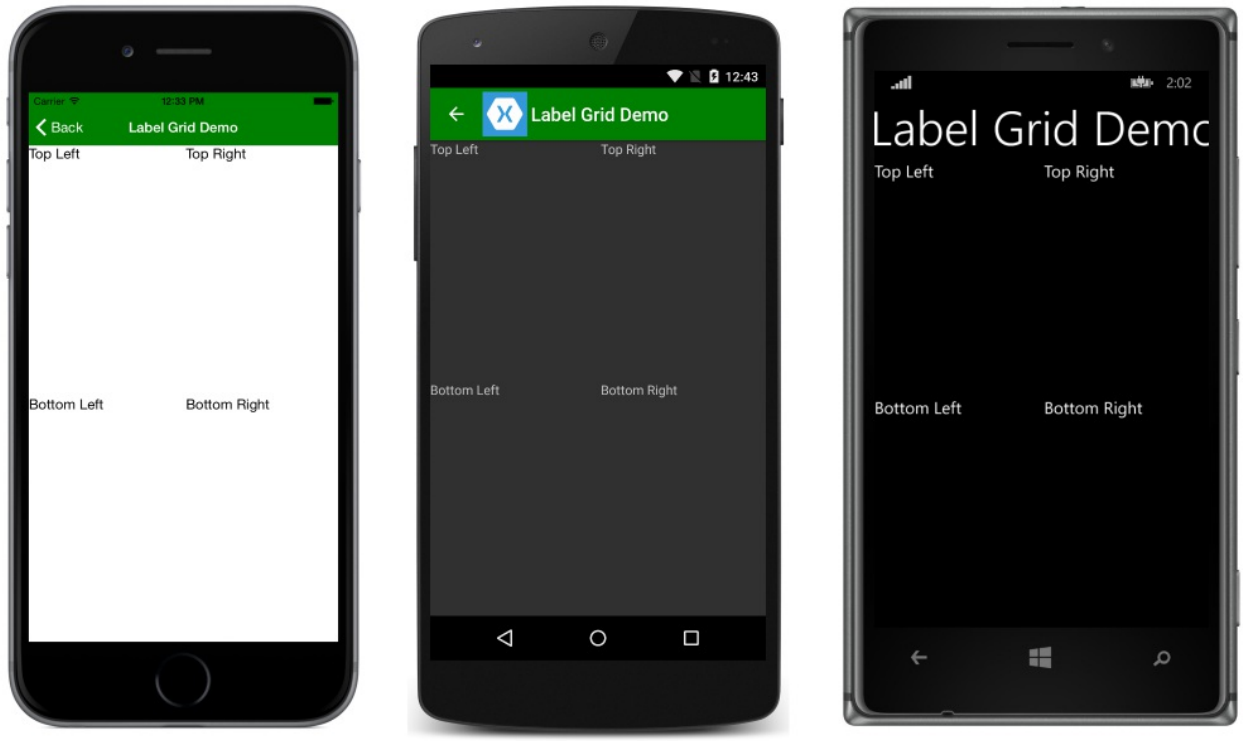
```
var grid = new Grid();
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength(2, GridUnitType.Star) });
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
grid.RowDefinitions.Add (new RowDefinition { Height = new GridLength(200)});
grid.ColumnDefinitions.Add (new ColumnDefinition{ Width = new GridLength (200) });
```

将视图放置在网格中

若要将视图中的放置 `Grid` 需要先将它们作为子级添加到网格中，然后再指定它们所属的行和列属于在中。

在 XAML 中，使用 `Grid.Row` 和 `Grid.Column` 指定放置每个单个视图。请注意，`Grid.Row` 和 `Grid.Column` 指定行和列的从零开始的列表为依据的位置。这意味着在 4x4 网格中，左上角单元格为 (0，0) 和右下角单元格是 (3，3)。

`Grid` 显示下面包含四个单元：



在 XAML:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="*" />
  </Grid.ColumnDefinitions>
  <Label Text="Top Left" Grid.Row="0" Grid.Column="0" />
  <Label Text="Top Right" Grid.Row="0" Grid.Column="1" />
  <Label Text="Bottom Left" Grid.Row="1" Grid.Column="0" />
  <Label Text="Bottom Right" Grid.Row="1" Grid.Column="1" />
</Grid>
```

在 C# 中:

```
var grid = new Grid();

grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(1, GridUnitType.Star)});
grid.RowDefinitions.Add(new RowDefinition { Height = new GridLength(1, GridUnitType.Star)});
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(1, GridUnitType.Star)});
grid.ColumnDefinitions.Add(new ColumnDefinition { Width = new GridLength(1, GridUnitType.Star)});

var topLeft = new Label { Text = "Top Left" };
var topRight = new Label { Text = "Top Right" };
var bottomLeft = new Label { Text = "Bottom Left" };
var bottomRight = new Label { Text = "Bottom Right" };

grid.Children.Add(topLeft, 0, 0);
grid.Children.Add(topRight, 1, 0);
grid.Children.Add(bottomLeft, 0, 1);
grid.Children.Add(bottomRight, 1, 1);
```

上面的代码中创建包含四个标签、两个列和两行的网格。请注意，每个标签将具有相同的大小和行将扩展以使用所有可用空间。

在上面的示例中，视图将添加到 `Grid.Children` 集合使用 `Add` 指定左侧和顶部参数的重载。使用时 `Add` 重载，指定左、右、上边框和底部参数，而左侧和顶部参数将始终引用单元格内 `Grid` ，右侧和底部参数可能看起来引用的单元格之外 `Grid` 。这是因为右边的参数必须始终为大于左侧参数和底部参数必须始终为大于顶部的参数。以下示例显示同时使用这二者的等效代码 `Add` 重载：

```
// left, top
grid.Children.Add(topLeft, 0, 0);
grid.Children.Add(topRight, 1, 0);
grid.Children.Add(bottomLeft, 0, 1);
grid.Children.Add(bottomRight, 1, 1);

// left, right, top, bottom
grid.Children.Add(topLeft, 0, 1, 0, 1);
grid.Children.Add(topRight, 1, 2, 0, 1);
grid.Children.Add(bottomLeft, 0, 1, 1, 2);
grid.Children.Add(bottomRight, 1, 2, 1, 2);
```

间距

`Grid` 具有属性来控制行和列之间的间距。以下属性可用于自定义用于 `Grid` ：

- **ColumnSpacing** –列之间的空间量。此属性的默认值为 6。
- **RowSpacing** –行之间的空间量。此属性的默认值为 6。

以下 XAML 指定 `Grid` 包含两列中，对应的一行和 5 像素的列之间的间距：

```
<Grid ColumnSpacing="5">
  <Grid.ColumnDefinitions>
    <ColumnDefinitions Width="*" />
    <ColumnDefinitions Width="*" />
  </Grid.ColumnDefinitions>
</Grid>
```

在 C# 中：

```
var grid = new Grid { ColumnSpacing = 5 };
grid.ColumnDefnitions.Add(new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star)});
grid.ColumnDefnitions.Add(new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star)});
```

范围

通常如果使用的一个网格，则应占用多个行或列的元素。请考虑一个简单的计算器应用程序：

请注意，0 按钮跨越就像两个列上为每个平台内置的计算器。这通过 `ColumnSpan` 属性，用于指定列的数量的元素应占用。该按钮的 XAML：

```
<Button Text = "0" Grid.Row="4" Grid.Column="0" Grid.ColumnSpan="2" />
```

和 C# 中：

```
Button zeroButton = new Button { Text = "0" };
controlGrid.Children.Add (zeroButton, 0, 4);
Grid.SetColumnSpan (zeroButton, 2);
```

请注意，在代码中，静态方法 `Grid` 类用于执行定位的更改包括对更改 `ColumnSpan` 和 `RowSpan` 。此外请注意，可以在任何时候设置其他属性，与使用静态方法设置的属性必须已先在网格中发生了更改。

上面的计算器应用程序的完整 XAML 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="LayoutSamples.CalculatorGridXAML"
Title = "Calculator - XAML"
BackgroundColor="#404040">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="plainButton" TargetType="Button">
                <Setter Property="BackgroundColor" Value="#eee"/>
                <Setter Property="TextColor" Value="Black" />
                <Setter Property="BorderRadius" Value="0"/>
                <Setter Property="FontSize" Value="40" />
            </Style>
            <Style x:Key="darkerButton" TargetType="Button">
                <Setter Property="BackgroundColor" Value="#ddd"/>
                <Setter Property="TextColor" Value="Black" />
                <Setter Property="BorderRadius" Value="0"/>
                <Setter Property="FontSize" Value="40" />
            </Style>
            <Style x:Key="orangeButton" TargetType="Button">
                <Setter Property="BackgroundColor" Value="#F8AD00"/>
```

```
                <Setter Property="BackgroundColor" Value="#E8AD00" />
                <Setter Property="TextColor" Value="White" />
                <Setter Property="BorderRadius" Value="0"/>
                <Setter Property="FontSize" Value="40" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <Grid x:Name="controlGrid" RowSpacing="1" ColumnSpacing="1">
            <Grid.RowDefinitions>
                <RowDefinition Height="150" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
                <RowDefinition Height="*" />
            </Grid.RowDefinitions>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Label Text="0" Grid.Row="0" HorizontalTextAlignment="End" VerticalTextAlignment="End"
TextColor="White"
        FontSize="60" Grid.ColumnSpan="4" />
            <Button Text = "C" Grid.Row="1" Grid.Column="0"
        Style="{StaticResource darkerButton}" />
            <Button Text = "+/-" Grid.Row="1" Grid.Column="1"
        Style="{StaticResource darkerButton}" />
            <Button Text = "%" Grid.Row="1" Grid.Column="2"
        Style="{StaticResource darkerButton}" />
            <Button Text = "div" Grid.Row="1" Grid.Column="3"
        Style="{StaticResource orangeButton}" />
            <Button Text = "7" Grid.Row="2" Grid.Column="0"
        Style="{StaticResource plainButton}" />
            <Button Text = "8" Grid.Row="2" Grid.Column="1"
        Style="{StaticResource plainButton}" />
            <Button Text = "9" Grid.Row="2" Grid.Column="2"
        Style="{StaticResource plainButton}" />
            <Button Text = "X" Grid.Row="2" Grid.Column="3"
        Style="{StaticResource orangeButton}" />
            <Button Text = "4" Grid.Row="3" Grid.Column="0"
        Style="{StaticResource plainButton}" />
            <Button Text = "5" Grid.Row="3" Grid.Column="1"
        Style="{StaticResource plainButton}" />
            <Button Text = "6" Grid.Row="3" Grid.Column="2"
        Style="{StaticResource plainButton}" />
            <Button Text = "-" Grid.Row="3" Grid.Column="3"
        Style="{StaticResource orangeButton}" />
            <Button Text = "1" Grid.Row="4" Grid.Column="0"
        Style="{StaticResource plainButton}" />
            <Button Text = "2" Grid.Row="4" Grid.Column="1"
        Style="{StaticResource plainButton}" />
            <Button Text = "3" Grid.Row="4" Grid.Column="2"
        Style="{StaticResource plainButton}" />
            <Button Text = "+" Grid.Row="4" Grid.Column="3"
        Style="{StaticResource orangeButton}" />
            <Button Text = "0" Grid.ColumnSpan="2"
        Grid.Row="5" Grid.Column="0" Style="{StaticResource plainButton}" />
            <Button Text = "." Grid.Row="5" Grid.Column="2"
        Style="{StaticResource plainButton}" />
            <Button Text = "=" Grid.Row="5" Grid.Column="3"
        Style="{StaticResource orangeButton}" />
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

请注意顶部的网格的标签和零个按钮都是 occuping 多个列。尽管无法使用嵌套的网格实现相似的布局 `ColumnSpan` & `RowSpan` 的方法是更简单。

C# 实现：

```
public CalculatorGridCode ()
{
  Title = "Calculator - C#";
  BackgroundColor = Color.FromHex ("#404040");

  var plainButton = new Style (typeof(Button)) {
    Setters = {
      new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#eee") },
      new Setter { Property = Button.TextColorProperty, Value = Color.Black },
      new Setter { Property = Button.BorderRadiusProperty, Value = 0 },
      new Setter { Property = Button.FontSizeProperty, Value = 40 }
    }
  };
  var darkerButton = new Style (typeof(Button)) {
    Setters = {
      new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#ddd") },
      new Setter { Property = Button.TextColorProperty, Value = Color.Black },
      new Setter { Property = Button.BorderRadiusProperty, Value = 0 },
      new Setter { Property = Button.FontSizeProperty, Value = 40 }
    }
  };
  var orangeButton = new Style (typeof(Button)) {
    Setters = {
      new Setter { Property = Button.BackgroundColorProperty, Value = Color.FromHex ("#E8AD00") },
      new Setter { Property = Button.TextColorProperty, Value = Color.White },
      new Setter { Property = Button.BorderRadiusProperty, Value = 0 },
      new Setter { Property = Button.FontSizeProperty, Value = 40 }
    }
  };

  var controlGrid = new Grid { RowSpacing = 1, ColumnSpacing = 1 };
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (150) });
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });
  controlGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star) });

  controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
  controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
  controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });
  controlGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1, GridUnitType.Star) });

  var label = new Label {
    Text = "0",
    HorizontalTextAlignment = TextAlignment.End,
    VerticalTextAlignment = TextAlignment.End,
    TextColor = Color.White,
    FontSize = 60
  };
  controlGrid.Children.Add (label, 0, 0);

  Grid.SetColumnSpan (label, 4);

  controlGrid.Children.Add (new Button { Text = "C", Style = darkerButton }, 0, 1);
  controlGrid.Children.Add (new Button { Text = "+/-", Style = darkerButton }, 1, 1);
  controlGrid.Children.Add (new Button { Text = "%", Style = darkerButton }, 2, 1);
  controlGrid.Children.Add (new Button { Text = "div", Style = orangeButton }, 3, 1);
  controlGrid.Children.Add (new Button { Text = "7", Style = plainButton }, 0, 2);
  controlGrid.Children.Add (new Button { Text = "8", Style = plainButton }, 1, 2);
  controlGrid.Children.Add (new Button { Text = "9", Style = plainButton }, 2, 2);
  controlGrid.Children.Add (new Button { Text = "X", Style = orangeButton }, 3, 2);
```

```
    controlGrid.Children.Add (new Button { Text = "4", Style = plainButton }, 0, 3);
    controlGrid.Children.Add (new Button { Text = "5", Style = plainButton }, 1, 3);
    controlGrid.Children.Add (new Button { Text = "6", Style = plainButton }, 2, 3);
    controlGrid.Children.Add (new Button { Text = "-", Style = orangeButton }, 3, 3);
    controlGrid.Children.Add (new Button { Text = "1", Style = plainButton }, 0, 4);
    controlGrid.Children.Add (new Button { Text = "2", Style = plainButton }, 1, 4);
    controlGrid.Children.Add (new Button { Text = "3", Style = plainButton }, 2, 4);
    controlGrid.Children.Add (new Button { Text = "+", Style = orangeButton }, 3, 4);
    controlGrid.Children.Add (new Button { Text = ".", Style = plainButton }, 2, 5);
    controlGrid.Children.Add (new Button { Text = "=", Style = orangeButton }, 3, 5);

    var zeroButton = new Button { Text = "0", Style = plainButton };
    controlGrid.Children.Add (zeroButton, 0, 5);
    Grid.SetColumnSpan (zeroButton, 2);

    Content = controlGrid;
  }
```

相关链接

- 借助 Xamarin.Forms，第 17 章创建移动应用
- 网格
- 布局（示例）
- BusinessTumble 示例（示例）

Xamarin.Forms FlexLayout

2018/11/13 • • Edit Online

使用 FlexLayout 堆叠或包装子视图的集合。

Xamarin.Forms `FlexLayout` Xamarin.Forms 版本 3.0 中新增功能。基于 CSS灵活框布局模块，通常称为_flex 布局_或_弹性框_，因此称为，因为它包括许多灵活的选项来排列子级在布局。

`FlexLayout` 类似于 Xamarin.Forms `StackLayout` ，因为它可以排列其子级水平和垂直堆栈。但是，`FlexLayout` 还能够换行及其子级，如果有太多而无法放入单个行或列中，并且还具有许多用于方向、对齐和适应各种屏幕尺寸的选项。

`FlexLayout` 派生自 `Layout<View>` ，并继承 `Children` 类型的属性 `IList<View>` 。

`FlexLayout` 定义了六个公共可绑定属性和影响大小、方向和子元素的对齐方式的五个附加的可绑定属性。(如果您不熟悉可绑定的附加属性，请参阅文章 **附加属性**。)在下面各节详细地介绍这些属性 **详细信息中的可绑定属性** 并 **详细的附加可绑定属性**. 但是，将这篇文章开始一节介绍了一些 **常见使用方案** 的 `FlexLayout` 更通俗地说描述其中的许多属性。本文的末尾，将了解如何结合 `FlexLayout` 与CSS 样式表。

常见使用方案

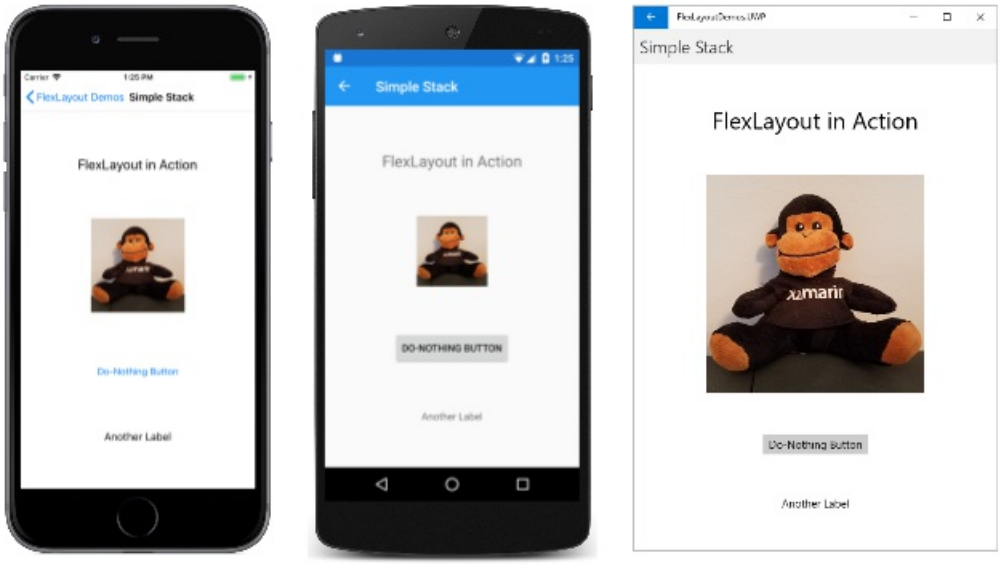
FlexLayoutDemos 示例程序包含多个页面的一些常见使用该份 `FlexLayout` ，当你尝试使用其属性。

使用适用于简单堆栈 FlexLayout

简单堆栈页显示了如何 `FlexLayout` 可以将替换为 `StackLayout` 但更简单的标记。此示例中的所有内容是 XAML 页中定义的。 `FlexLayout` 包含四个子级：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:FlexLayoutDemos"
             x:Class="FlexLayoutDemos.SimpleStackPage"
             Title="Simple Stack">

    <FlexLayout Direction="Column"
                AlignItems="Center"
                JustifyContent="SpaceEvenly">

        <Label Text="FlexLayout in Action"
               FontSize="Large" />

        <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}" />

        <Button Text="Do-Nothing Button" />

        <Label Text="Another Label" />
    </FlexLayout>
</ContentPage>
```

下面是 iOS、Android 和通用 Windows 平台上运行该页面：



三个属性`FlexLayout`中所示**SimpleStackPage.xaml**文件：

- `Direction`属性设置为值为`FlexDirection`枚举。默认值为`Row`。将属性设置为`Column`导致的子级`FlexLayout`排列在项的单个列中。

  当中的项`FlexLayout`排列在列中，`FlexLayout`称其具有垂直_主轴_和水平_交叉轴_。

- `AlignItems`属性属于类型`FlexAlignItems`和指定的项在横轴上的对齐方式。`Center`选项将使每一项要在水平居中。

  如果已使用`StackLayout`而非`FlexLayout`对于此任务，您将通过分配中居中的所有项`HorizontalOptions`到每个项的属性`Center`。`HorizontalOptions`属性并不适用于的子级`FlexLayout`，但单个`AlignItems`属性可以达到相同目的。如果需要可以使用`AlignSelf`附加可绑定属性重写`AlignItems`各项的属性：

  ```
  <Label Text="FlexLayout in Action"
         FontSize="Large"
         FlexLayout.AlignSelf="Start" />
  ```

  与此更改后，这个`Label`定位在的左边缘`FlexLayout`时的阅读顺序是从左到右。

- `JustifyContent`属性属于类型`FlexJustify`，并指定项主要的轴上的排列方式。`SpaceEvenly`选项分配所有项之间进行均分，和上面的第一项和最后一项下的所有剩余垂直空间。

  如果已使用`StackLayout`，需要将分配`VerticalOptions`到每个项的属性`CenterAndExpand`要实现类似效果。但`CenterAndExpand`选项会分配比每个项的第一项之前和之后的最后一项之间的两倍空间。您可以模仿`CenterAndExpand`的选项`VerticalOptions`通过设置`JustifyContent`的属性`FlexLayout`到`SpaceAround`。

这些`FlexLayout`部分中的更详细地讨论属性 **详细信息中的可绑定属性** 下面。

使用 FlexLayout 进行包装的项

照片包装 页 **FlexLayoutDemos** 示例演示如何`FlexLayout`可以包装到其他行或列及其子级。XAML 文件实例化`FlexLayout`，并将分配两个属性：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FlexLayoutDemos.PhotoWrappingPage"
             Title="Photo Wrapping">
    <Grid>
        <ScrollView>
            <FlexLayout x:Name="flexLayout"
                        Wrap="Wrap"
                        JustifyContent="SpaceAround" />
        </ScrollView>

        <ActivityIndicator x:Name="activityIndicator"
                           IsRunning="True"
                           VerticalOptions="Center" />
    </Grid>
</ContentPage>
```

`Direction`属性的这`FlexLayout`未设置，因此它具有的默认设置`Row`，这意味着在行中排列子级，并且主要轴是水平的。

`Wrap` 属性是枚举类型的 `FlexWrap` 。如果有太多的项，以适合的行，此属性设置会导致要换行到下一行中的项。

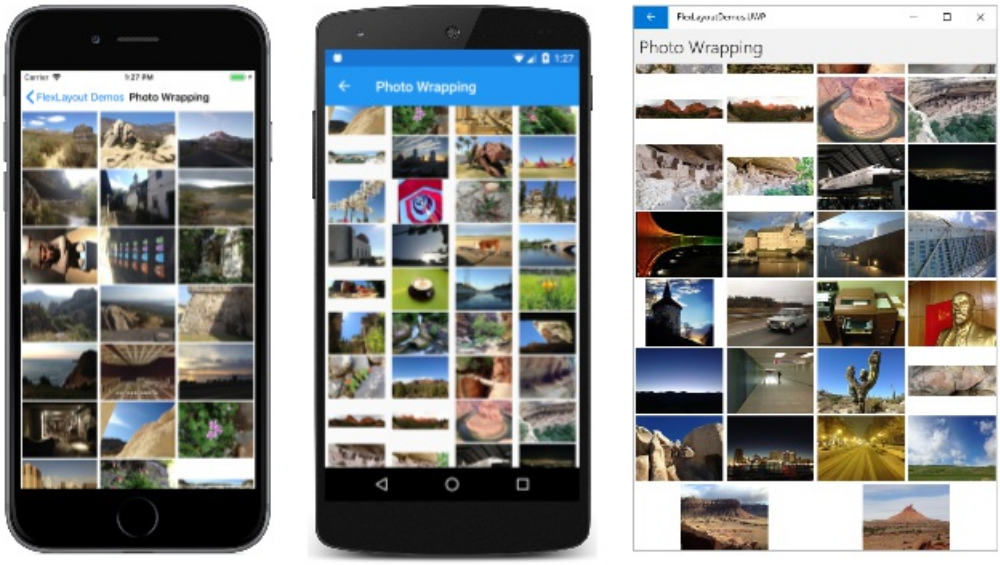
请注意，`FlexLayout`的子`ScrollView`。如果有太多的行以容纳在页上，则`ScrollView`具有一个默认`Orientation`属性的`Vertical`，并允许垂直滚动。

`JustifyContent`属性分配主轴（水平轴）上的剩余空间，以便每个项括在相同的空白空间量。

代码隐藏文件访问的示例照片集合，并将它们添加到`Children`的集合`FlexLayout`：

```
public partial class PhotoWrappingPage : ContentPage
{
    // Class for deserializing JSON list of sample bitmaps
    [DataContract]
    class ImageList
    {
        [DataMember(Name = "photos")]
        public List<string> Photos = null;
    }

    public PhotoWrappingPage ()
    {
        InitializeComponent ();

        LoadBitmapCollection();
    }

    async void LoadBitmapCollection()
    {
        int imageDimension = Device.RuntimePlatform == Device.iOS ||
                             Device.RuntimePlatform == Device.Android ? 240 : 120;

        string urlSuffix = String.Format("?width={0}&height={0}&mode=max", imageDimension);

        using (WebClient webClient = new WebClient())
        {
            try
            {
                // Download the list of stock photos
                Uri uri = new Uri("http://docs.xamarin.com/demo/stock.json");
                byte[] data = await webClient.DownloadDataTaskAsync(uri);

                // Convert to a Stream object
                using (Stream stream = new MemoryStream(data))
                {
                    // Deserialize the JSON into an ImageList object
                    var jsonSerializer = new DataContractJsonSerializer(typeof(ImageList));
                    ImageList imageList = (ImageList)jsonSerializer.ReadObject(stream);

                    // Create an Image object for each bitmap
                    foreach (string filepath in imageList.Photos)
                    {
                        Image image = new Image
                        {
                            Source = ImageSource.FromUri(new Uri(filepath + urlSuffix))
                        };
                        flexLayout.Children.Add(image);
                    }
                }
            }
            catch
            {
                flexLayout.Children.Add(new Label
                {
                    Text = "Cannot access list of bitmap files"
                });
            }
        }

        activityIndicator.IsRunning = false;
        activityIndicator.IsVisible = false;
    }
}
```

下面是渐进式滚动从顶部到底部的三个平台上运行的程序：



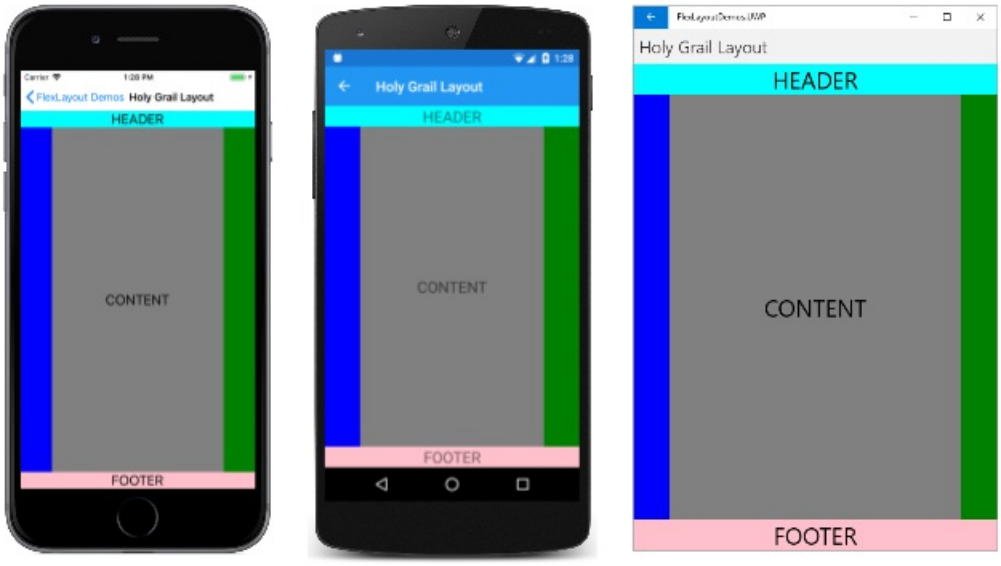
使用 FlexLayout 页面布局

在调用的 web 设计中没有的标准版式 *holy grail* 因为它是非常有利，但通常难以实现与完善的布局格式。布局包含在页面顶部的标头和页脚在底部，这两个扩展到整个页面的宽度。占用的页的中心是主要的内容，但通常与左侧的内容和补充信息的纵栏式菜单 (有时称为_放在一边_区域) 右侧。CSS 灵活框布局规范的第 5.4.1 节介绍了如何在弹性框实现 holy grail 布局。

Holy Grail 布局 页 **FlexLayoutDemos** 示例显示了使用其中一个此布局的简单实现 `FlexLayout` 嵌套在另一个。因为此页专为手机在纵向模式下，左侧和右侧的内容区域的区域只是 50 像素宽：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="FlexLayoutDemos.HolyGrailLayoutPage"
             Title="Holy Grail Layout">

    <FlexLayout Direction="Column">

        <!-- Header -->
        <Label Text="HEADER"
               FontSize="Large"
               BackgroundColor="Aqua"
               HorizontalTextAlignment="Center" />

        <!-- Body -->
        <FlexLayout FlexLayout.Grow="1">

            <!-- Content -->
            <Label Text="CONTENT"
                   FontSize="Large"
                   BackgroundColor="Gray"
                   HorizontalTextAlignment="Center"
                   VerticalTextAlignment="Center"
                   FlexLayout.Grow="1" />

            <!-- Navigation items-->
            <BoxView FlexLayout.Basis="50"
                     FlexLayout.Order="-1"
                     Color="Blue" />

            <!-- Aside items -->
            <BoxView FlexLayout.Basis="50"
                     Color="Green" />

        </FlexLayout>

        <!-- Footer -->
        <Label Text="FOOTER"
               FontSize="Large"
               BackgroundColor="Pink"
               HorizontalTextAlignment="Center" />
    </FlexLayout>
</ContentPage>
```

此处它三个平台上运行：



与呈现的导航和 aside 区域 `BoxView` 左侧和右侧。

第一个 `FlexLayout` 在 XAML 文件有主纵轴，并包含三个孩子排列在列中。这些是标头、页和表尾的正文。嵌套 `FlexLayout` 排列在行中的三个孩子都具有主要横轴。

在此程序演示了三个附加的可绑定属性：

- `Order` 设置附加的可绑定属性的第一个 `BoxView` 。此属性是一个整数，其默认值为 0。此属性可用于更改在布局顺序。通常，开发人员更喜欢页后，可以在之前的导航项的标记中显示的内容并留出的项。设置 `Order` 在第一个属性 `BoxView` 为值早于其他同级后，即可显示为行中的第一项。同样，可以确保项通过设置显示最后一个 `Order` 属性大于其同级的值。

- `Basis` 对两个设置附加的可绑定属性 `BoxView` 要为其提供的 50 像素宽度的项。此属性属于类型 `FlexBasis` ，定义类型的静态属性的结构 `FlexBasis` 名为 `Auto` ，这是默认设置。可以使用 `Basis` 指定像素大小或一个百分比，指示空间项占用主轴上。它称为_基础_因为它指定是所有后续布局的基础的项大小。

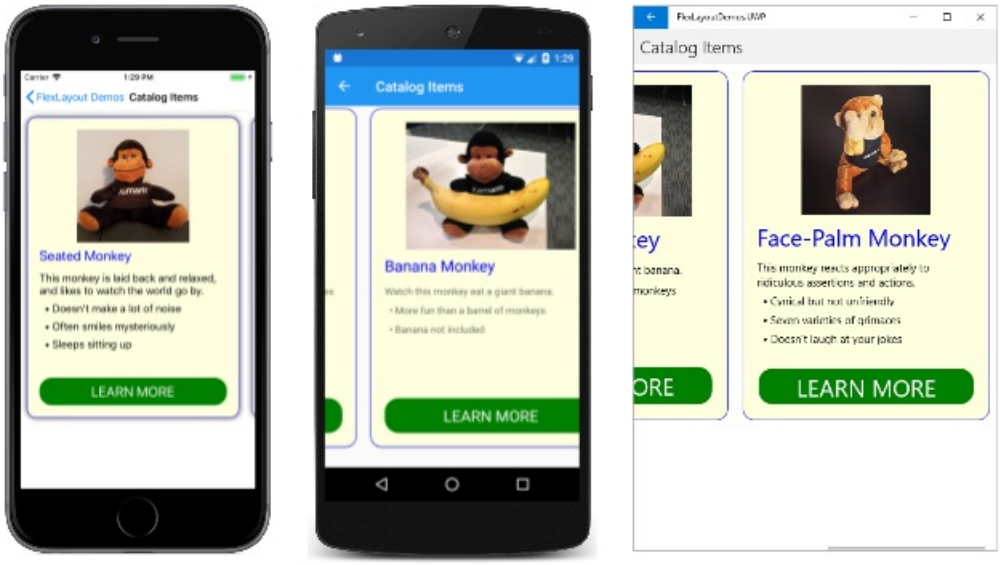
- `Grow` 属性设置在嵌套 `Layout` 并在 `Label` 子表示内容。此属性的类型是 `float` 和默认值为 0。设置为正值时，该主轴的所有剩余空间分配到该项目和具有正值的同级 `Grow` 。值，类似于中的星型规范按比例分配空间 `Grid` 。

  第一个 `Grow` 设置附加的属性上的嵌套 `FlexLayout` ，则表示此 `FlexLayout` 是占用内外部的所有未使用垂直空间 `FlexLayout` 。第二个 `Grow` 上设置附加的属性 `Label` 表示的内容，指示此内容以占据中内部的所有未使用水平空间 `FlexLayout` 。

  此外，还有一个类似 `Shrink` 附加时子项的大小超过的大小，可以使用的可绑定属性 `FlexLayout` 但不是需要换行。

使用 FlexLayout 目录项

目录项 页面 **FlexLayoutDemos** 示例是类似于示例 1 中的 CSS Flex 布局框规范部分1.1，只不过它将显示一系列可水平滚动的图片和说明的三个放弃：



每三个放弃 `FlexLayout` 中包含 `Frame` 给定了显式高度和宽度，并且也是更大的子 `FlexLayout` 。在此 XAML 文件中，大部分的属性 `FlexLayout` 在样式中，一个是隐式样式的指定子级：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:FlexLayoutDemos"
             x:Class="FlexLayoutDemos.CatalogItemsPage"
             Title="Catalog Items">
    <ContentPage.Resources>
        <Style TargetType="Frame">
            <Setter Property="BackgroundColor" Value="LightYellow" />
            <Setter Property="BorderColor" Value="Blue" />
            <Setter Property="Margin" Value="10" />
```

```
            <Setter Property="CornerRadius" Value="15" />
        </Style>

        <Style TargetType="Label">
            <Setter Property="Margin" Value="0, 4" />
        </Style>

        <Style x:Key="headerLabel" TargetType="Label">
            <Setter Property="Margin" Value="0, 8" />
            <Setter Property="FontSize" Value="Large" />
            <Setter Property="TextColor" Value="Blue" />
        </Style>

        <Style TargetType="Image">
            <Setter Property="FlexLayout.Order" Value="-1" />
            <Setter Property="FlexLayout.AlignSelf" Value="Center" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="Text" Value="LEARN MORE" />
            <Setter Property="FontSize" Value="Large" />
            <Setter Property="TextColor" Value="White" />
            <Setter Property="BackgroundColor" Value="Green" />
            <Setter Property="BorderRadius" Value="20" />
        </Style>
    </ContentPage.Resources>

    <ScrollView Orientation="Both">
        <FlexLayout>
            <Frame WidthRequest="300"
                   HeightRequest="480">

                <FlexLayout Direction="Column">
                    <Label Text="Seated Monkey"
                           Style="{StaticResource headerLabel}" />
                    <Label Text="This monkey is laid back and relaxed, and likes to watch the world go by."
/>
                    <Label Text="  &#x2022; Doesn't make a lot of noise" />
                    <Label Text="  &#x2022; Often smiles mysteriously" />
                    <Label Text="  &#x2022; Sleeps sitting up" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}"
                           WidthRequest="180"
                           HeightRequest="180" />
                    <Label FlexLayout.Grow="1" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame WidthRequest="300"
                   HeightRequest="480">

                <FlexLayout Direction="Column">
                    <Label Text="Banana Monkey"
                           Style="{StaticResource headerLabel}" />
                    <Label Text="Watch this monkey eat a giant banana." />
                    <Label Text="  &#x2022; More fun than a barrel of monkeys" />
                    <Label Text="  &#x2022; Banana not included" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.Banana.jpg}"
                           WidthRequest="240"
                           HeightRequest="180" />
                    <Label FlexLayout.Grow="1" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame WidthRequest="300"
                   HeightRequest="480">

                <FlexLayout Direction="Column">
```

```
                    <Label Text="Face-Palm Monkey"
                           Style="{StaticResource headerLabel}" />
                    <Label Text="This monkey reacts appropriately to ridiculous assertions and actions." />
                    <Label Text="  &#x2022; Cynical but not unfriendly" />
                    <Label Text="  &#x2022; Seven varieties of grimaces" />
                    <Label Text="  &#x2022; Doesn't laugh at your jokes" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.FacePalm.jpg}"
                           WidthRequest="180"
                           HeightRequest="180" />
                    <Label FlexLayout.Grow="1" />
                    <Button />
                </FlexLayout>
            </Frame>
        </FlexLayout>
    </ScrollView>
</ContentPage>
```

隐式样式 `Image` 包括两个附加的可绑定属性的设置 `Flexlayout`：

```
<Style TargetType="Image">
    <Setter Property="FlexLayout.Order" Value="-1" />
    <Setter Property="FlexLayout.AlignSelf" Value="Center" />
</Style>
```

`Order` 设置为–1 会导致 `Image` 首先显示在每个嵌套元素 `FlexLayout` 而不考虑其位置中的 children 集合视图。 `AlignSelf` 的属性 `Center` 会导致 `Image` 要内居中 `FlexLayout` 。这会重写的设置 `AlignItems` 属性，它具有默认值的 `Stretch` ，这意味着， `Label` 并 `Button` 子级将拉伸到整个宽度 `FlexLayout` 。

中的三个 `FlexLayout` 视图，空白 `Label` 之前 `Button` ，但它具有 `Grow` 设置为 1。这意味着，将所有额外的垂直空间分配到此值为空 `Label` ，这有效地将推送 `Button` 到底部。

详细信息中的可绑定属性

现在，已了解的一些常见的应用程序 `FlexLayout` 的属性 `FlexLayout` 可以更详细地探讨了。 `FlexLayout` 定义设置的六个可绑定属性 `FlexLayout` 本身，在代码或 XAML 控件方向和对齐方式。(这些属性之一 `Position` ，本文不介绍。)

您可以尝试使用五个剩余可绑定属性使用 试验 页 **FlexLayoutDemos** 示例。此页面允许您添加或删除从子级 `FlexLayout` 并设置五个可绑定属性的组合。所有子级 `FlexLayout` 都 `Label` 视图的各种颜色和大小，使用 `Text` 属性设置为对应的数字为在其位置 `Children` 集合。

程序启动时，五 `Picker` 视图中显示五个默认值 `FlexLayout` 属性。 `FlexLayout` 屏幕的底部包含三个子级：

每个 `Label` 视图具有灰色背景显示的分配的空间 `Label` 内 `FlexLayout` 。背景 `FlexLayout` 本身是艾莉斯蓝。它在左侧和右侧的小边距除外占据了整个底部区域中的页。

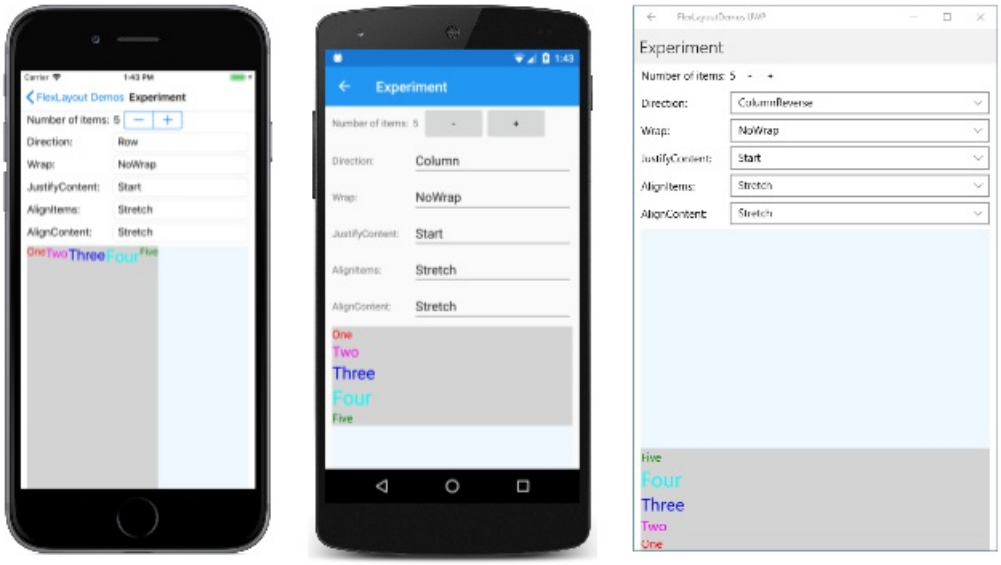
Direction 属性

`Direction` 属性属于类型 `FlexDirection` ，具有四个成员的枚举：

- `Column`
- `ColumnReverse` （或者"列的反向"在 XAML 中）
- `Row` 默认值
- `RowReverse` （或者"行的反向"在 XAML 中）

在 XAML 中，可以指定使用的枚举成员名称以小写字母大写，此属性的值或混合大小写，也可以使用 CSS 指示器相同的括号中所示的两个其他字符串。(在中定义的"列反向"和"行反向"字符串 `FlexDirectionTypeConverter` XAML 分析器使用的类。)

下面是实验(从左到右)，显示页面 `Row` 方向 `Column` 方向，和 `ColumnReverse` 方向：



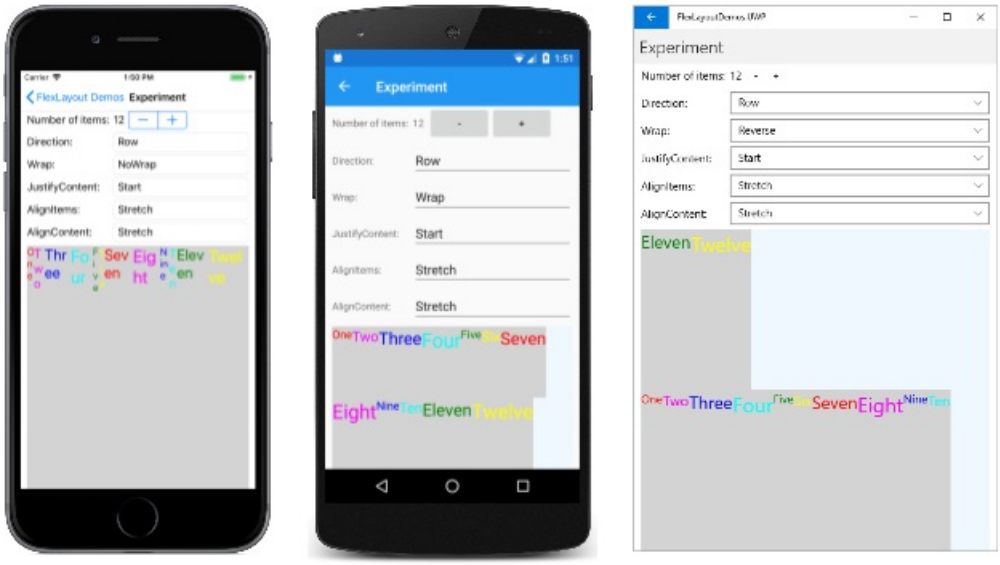
请注意，对于 `Reverse` 项启动选项，在右侧或底部。

自动换行属性

`Wrap` 属性属于类型 `FlexWrap` ，具有三个成员的枚举：

- `NoWrap` 默认值
- `Wrap`
- `Reverse` （或者"自动换行的反向"在 XAML 中）

从左到右，这些屏幕显示 `NoWrap` ， `Wrap` 和 `Reverse` 12 子级的选项：



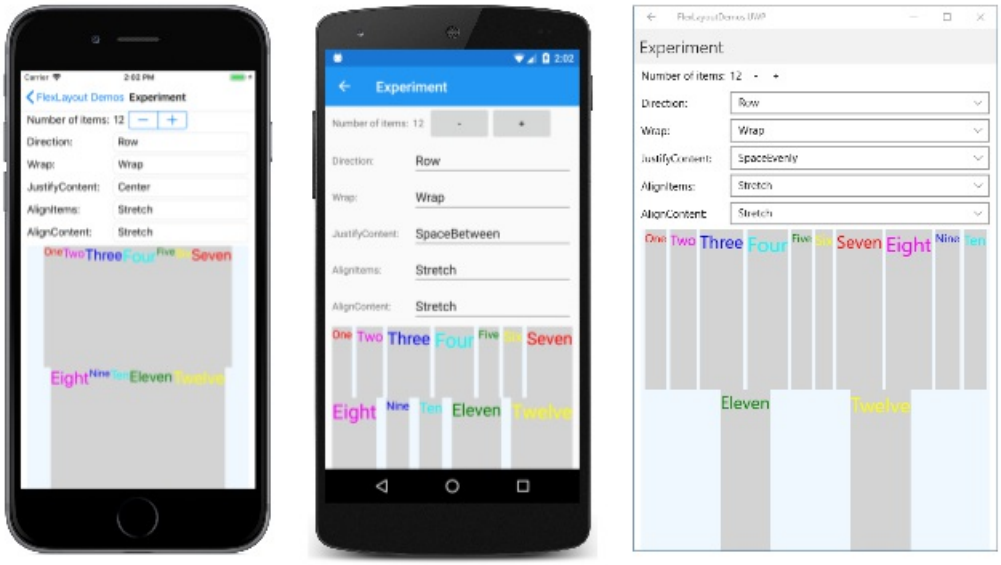
当 `Wrap` 属性设置为 `NoWrap` 和主轴受到约束（如在此程序），以及主轴不高或足够宽，以适应所有子级 `FlexLayout` 尝试使项目变小，如 iOS 屏幕截图演示。您可以控制的项 shrinkness `Shrink` 附加可绑定属性。

JustifyContent 属性

`JustifyContent` 属性属于类型 `FlexJustify`，一个具有六个成员的枚举：

- `Start`（或"flex 启动"在 XAML 中），默认值
- `Center`
- `End`（或者"flex 端"在 XAML 中）
- `SpaceBetween`（或者"空间-之间"在 XAML 中）
- `SpaceAround`（或者"空间的大约"在 XAML 中）
- `SpaceEvenly`

此属性指定项在此示例中为水平轴的主要轴上的分布方式：



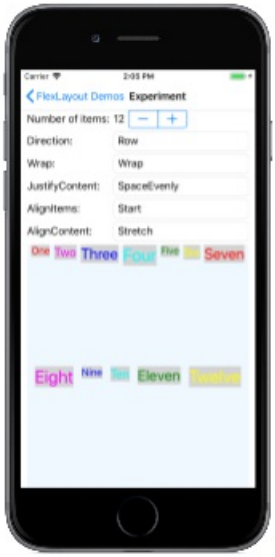
在所有三个屏幕截图 `Wrap` 属性设置为 `Wrap`。`Start` 默认上面的 Android 屏幕截图中所示。此处的 iOS 屏幕快照显示 `Center` 选项：所有项都移动到中心。其他三个选项从单词 `Space` 分配项未占用的额外空间。`SpaceBetween` 分配的项; 同样之间的空间 `SpaceAround` put 等于每个项周围的空间时 `SpaceEvenly` put 等于每个项之间和第一项之前和之后的行的最后一项的空间。

AlignItems 属性

`AlignItems` 属性属于类型 `FlexAlignItems`，具有四个成员的枚举：

- `Stretch` 默认值
- `Center`
- `Start` （或者"flex 启动"在 XAML 中）
- `End` （或者"flex 端"在 XAML 中）

这是两个属性之一（另一个执行 `AlignContent` ），该值指示子交叉轴上的对齐方式。在每个行中，子级是拉伸（如上面的屏幕截图中所示），或对齐上开始、居中或结束的每个项，如下面的三个屏幕截图中所示：



在 iOS 屏幕截图中，所有子级的顶部对齐。中的 Android 屏幕截图，项是垂直方向上居中基于在最高的子活动。在 UWP 屏幕截图中，所有项的底部对齐。

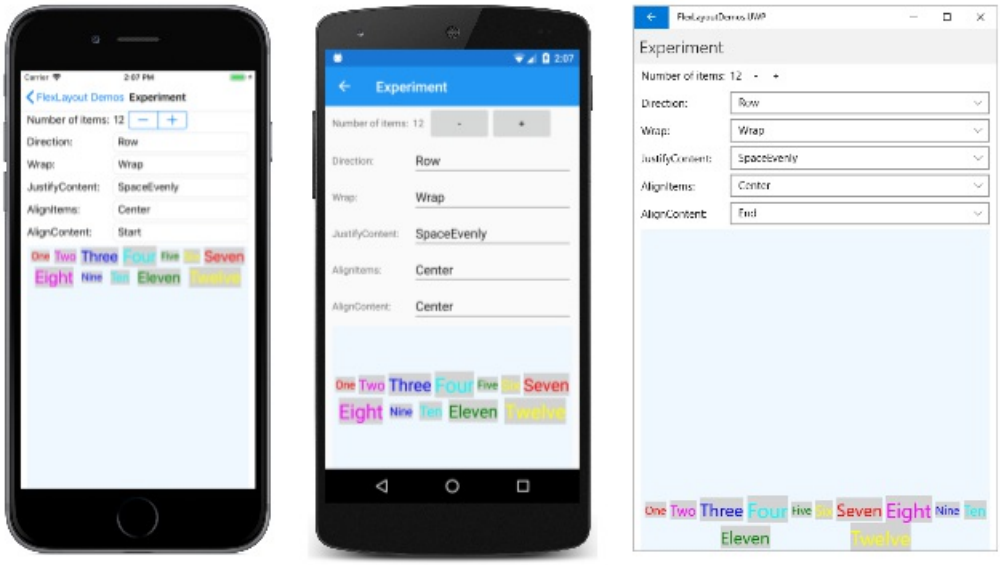
对于任何单个项，`AlignItems` 设置可以使用重写 `AlignSelf` 附加可绑定属性。

AlignContent 属性

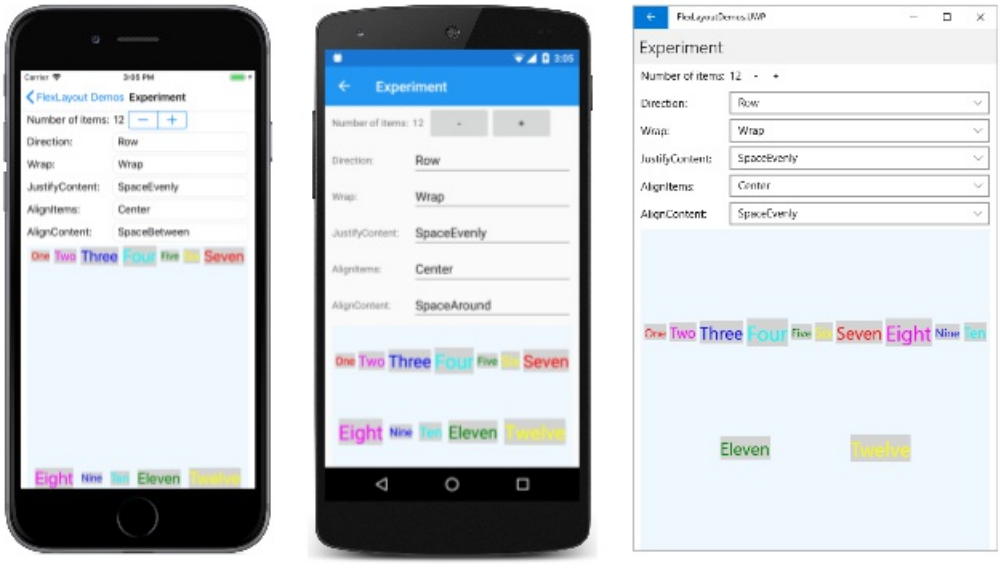
`AlignContent` 属性属于类型 `FlexAlignContent` ，具有七个成员的枚举：

- `Stretch` 默认值
- `Center`
- `Start` （或者"flex 启动"在 XAML 中）
- `End` （或者"flex 端"在 XAML 中）
- `SpaceBetween` （或者"空间-之间"在 XAML 中）
- `SpaceAround` （或者"空间的大约"在 XAML 中）
- `SpaceEvenly`

像 `AlignItems` ，则 `AlignContent` 属性还将对齐交叉轴上的子级，但会影响整个行或列：



在 iOS 屏幕截图中，这两个行位于顶部;在 Android 屏幕截图，它们在中心;并且 UWP 的屏幕截图中它们是在底部。此外可以以各种方式分布行：



`AlignContent` 不起作用时只有一个行或列。

详细信息中的附加可绑定属性

`FlexLayout` 定义五个附加的可绑定属性。子级上设置这些属性 `FlexLayout` 且仅属于该特定子。

AlignSelf 属性

`AlignSelf` 附加可绑定属性属于类型 `FlexAlignSelf`，具有五个成员的枚举：

- `Auto` 默认值
- `Stretch`
- `Center`
- `Start`（或者"flex 启动"在 XAML 中）
- `End`（或者"flex 端"在 XAML 中）

任何单个子 `FlexLayout`，此属性设置会替代 `AlignItems` 属性设置的 `FlexLayout` 本身。默认设置 `Auto` 意味着使用 `AlignItems` 设置。

有关 `Label` 名为元素 `label`（或示例），可以设置 `AlignSelf` 类似下面的代码中的属性：

```
FlexAlign.SetAlignSelf(label, FlexAlignSelf.Center);
```

请注意，没有不引用 `FlexLayout` 的父级 `Label` 。在 XAML，您设置此类属性：

```
<Label ... FlexAlign.AlignSelf="Center" ... />
```

Order 属性

`Order` 属性属于类型 `int` 。默认值为 0。

`Order` 属性可以更改的顺序的子级的 `FlexLayout` 排列。通常情况下的子级 `FlexLayout` 排列中出现的顺序相同 `Children` 集合。可通过设置覆盖此顺序 `Order` 附加可绑定属性设置为上一个或多个子级的非零的整数值。 `FlexLayout` 然后排列基于的设置及其子级 `Order` 属性上每个子级，但具有相同的子级 `Order` 设置中出现的顺序排列 `Children` 集合。

基础属性

`Basis` 附加可绑定属性指示的一个子级的已分配的空间量 `FlexLayout` 主要轴上。按大小指定 `Basis` 属性为父该主轴大小 `FlexLayout` 。因此， `Basis` 子级排列的行或高度时在列中排列子级时指示的子级的宽度。

`Basis` 属性属于类型 `FlexBasis` ，一种结构。可以指定的大小，在任一设备无关的单位或大小的百分比 `FlexLayout` 。默认值 `Basis` 属性是静态属性 `FlexBasis.Auto` ，这意味着子的请求使用宽度或高度。

在代码中，可以设置 `Basis` 属性 `Label` 名为 `label` 到 40 个与设备无关单位如下：

```
FlexLayout.SetBasis(label, new FlexBasis(40, false));
```

第二个参数 `FlexBasis` 名为构造函数 `isRelative` ，指示是否为相对大小 (`true`) 或绝对 (`false`)。自变量具有默认值为 `false` ，因此还可以使用以下代码：

```
FlexLayout.SetBasis(label, new FlexBasis(40));
```

隐式转换 `float` 到 `FlexBasis` 定义，因此您可以进一步简化：

```
FlexLayout.SetBasis(label, 40);
```

可以将大小设置为 25%的 `FlexLayout` 父如下：

```
FlexLayout.SetBasis(label, new FlexBasis(0.25f, true));
```

此小数部分的值必须是 0 到 1 范围内。

在 XAML，您可以使用设备无关的单位中的大小数字：

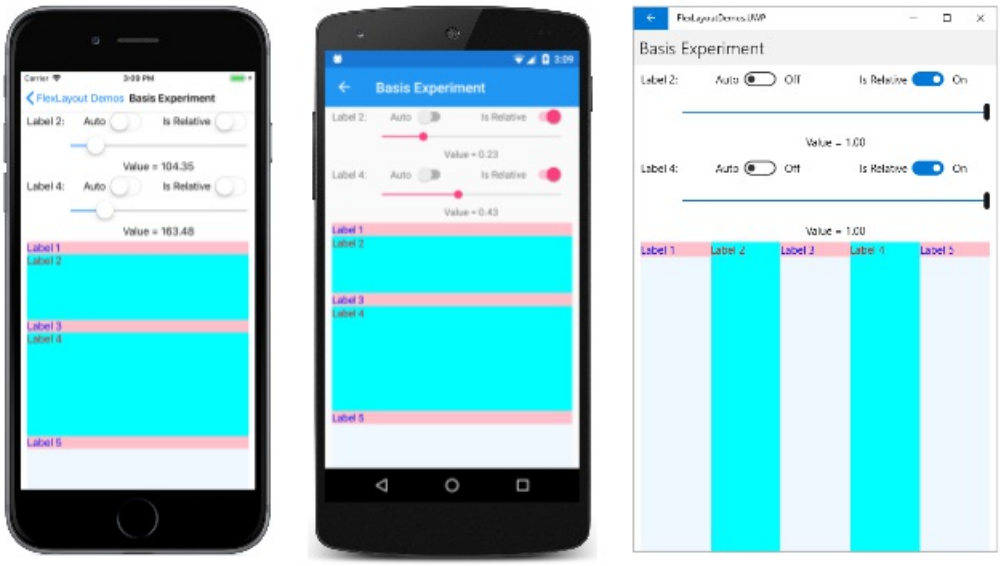
```
<Label ... FlexLayout.Basis="40" ... />
```

也可以在范围 0%到 100%中指定百分比：

```
<Label ... FlexLayout.Basis="25%" ... />
```

基础试验 页 **FlexLayoutDemos** 示例允许用户体验与 `Basis` 属性。该页面显示的列已包装的五个 `Label` 交替背

景和前景颜色的元素。两个 `Slider` 元素允许您指定 `Basis` 第二个和第四个值 `Label`：



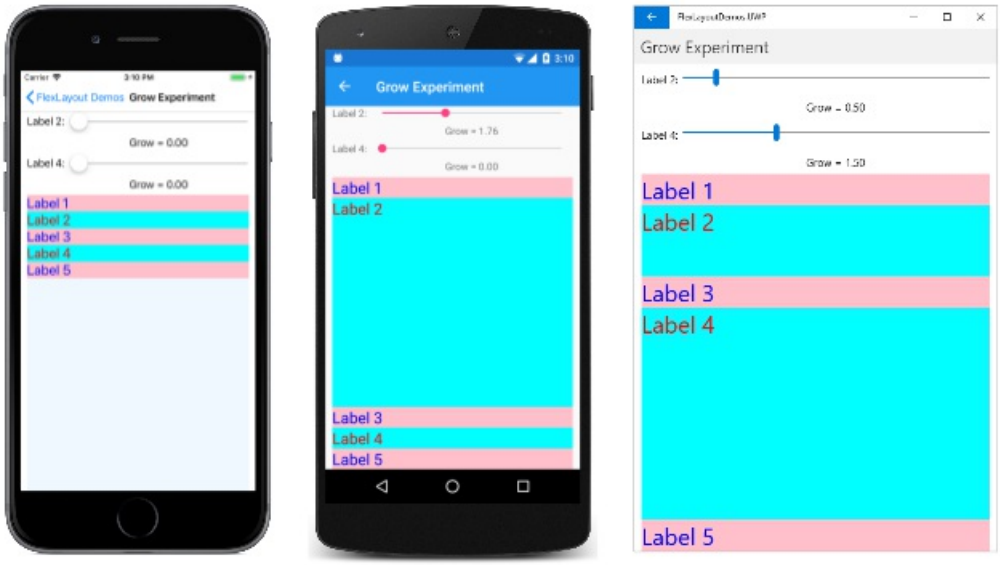
在左侧的 iOS 屏幕截图显示了这两个 `Label` 元素获得高度以与设备无关单位。Android 屏幕显示它们被授予的总高度的一小部分的高度 `FlexLayout` 。如果 `Basis` 设置为 100%，则的高度，该子级则 `FlexLayout` ，和将换行到下一列，并占用整个图柱的高度的如 UWP 的屏幕截图中所示：似乎像排列在行中的五个子级但实际上在五个列中排列。

扩展属性

`Grow` 附加可绑定属性属于类型 `int` 。默认值为 0，并且值必须大于或等于 0。

`Grow` 属性的作用时 `Wrap` 属性设置为 `NoWrap` 的子行具有的宽度小于的总宽度 `FlexLayout` ，或子级的列具有比短高度 `FlexLayout` 。 `Grow` 属性指示如何将子级之间剩余的空间分配。

在中增长实验页上，五个 `Label` 交替颜色的元素排列在列和两个 `Slider` 元素，可以调整 `Grow` 属性的第二个和第四个 `Label` 。在最左侧的 iOS 屏幕快照显示的默认 `Grow` 0 的属性：



如果未指定任何一个子正 `Grow` 值，则该子级占用的所有剩余的空间，如 Android 屏幕截图中所示。此外可以在两个或多个子级之间分配此空间。在 UWP 屏幕截图中， `Grow` 属性的第二个 `Label` 设置为 0.5，而 `Grow` 的第四个属性 `Label` 1.5，这将使第四个 `Label` 三倍的第二个将剩余的空间 `Label` .

子视图如何使用该空间取决于特定的子类型。有关 `Label` ，可将文本放置的总空间内 `Label` 使用的属性 `HorizontalTextAlignment` 和 `VerticalTextAlignment` 。

收缩属性

`Shrink` 附加可绑定属性属于类型 `int`。默认值为 1，并且值必须大于或等于 0。

`Shrink` 属性的作用时 `Wrap` 属性设置为 `NoWrap` 和聚合的子行的宽度大于宽度的 `FlexLayout`，或子级的单个列的聚合高度大于高度 `FlexLayout`。通常情况下 `FlexLayout` 时将显示这些子级又会限制运行它们的大小。`Shrink` 属性可以指示哪些子活动优先级中显示在其完整大小。

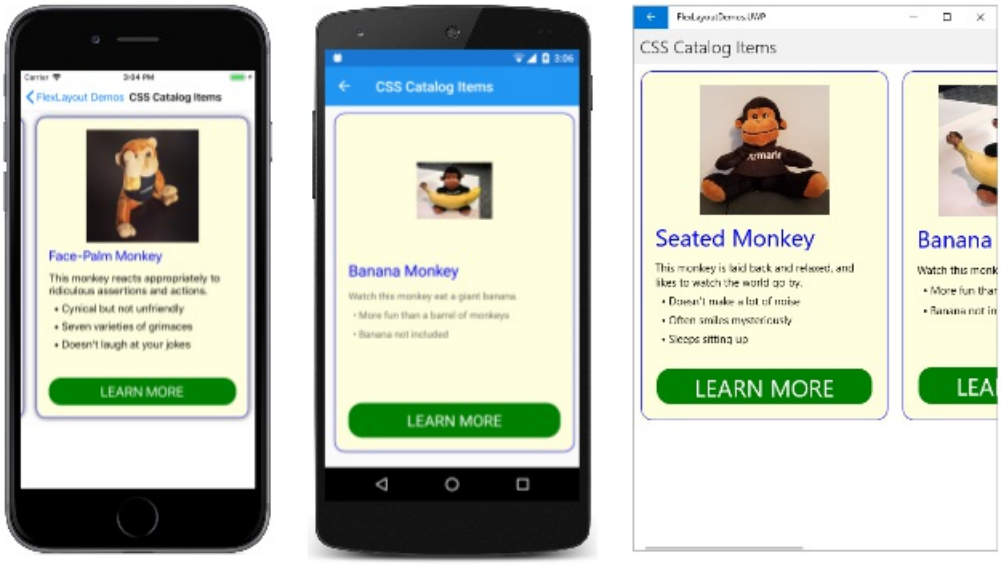
收缩实验页上创建 `FlexLayout` 具有五个单行 `Label` 需要更多空间比多个子 `FlexLayout` 宽度。在左侧的 iOS 屏幕截图显示了所有 `Label` 具有默认值为 1 的元素：

在 Android 屏幕截图中，`Shrink` 第二个值 `Label` 设为 0，并且 `Label` 显示在其整个宽度。此外，第四个 `Label` 给定 `Shrink` 值大于 1，并已收缩。UWP 屏幕截图显示了这两 `Label` 元素被授予 `Shrink` 是可能的值为 0 以允许它们显示在其完整大小，如果该。

您可以同时设置 `Grow` 并 `Shrink` 值以容纳其中的聚合子大小可能有时会更少或有时大于的大小的情况下 `FlexLayout`。

CSS 样式设定 FlexLayout

可以使用CSS 样式门户中的 Xamarin.Forms 3.0 中引入了功能 `FlexLayout`。**CSS 目录项** 页 **FlexLayoutDemos** 示例重复项的布局 **目录项** 页上，但使用 CSS对于许多样式的样式表：



原始**CatalogItemsPage.xaml**文件具有五个 `Style` 中的定义及其 `Resources` 15 部分 `Setter` 对象。在中**CssCatalogItemsPage.xaml**文件，已缩减成两个 `Style` 定义具有只需四个 `Setter` 对象。这些样式补充 Xamarin.Forms CSS 样式功能当前不支持的属性的 CSS 样式表：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:FlexLayoutDemos"
             x:Class="FlexLayoutDemos.CssCatalogItemsPage"
             Title="CSS Catalog Items">
    <ContentPage.Resources>
        <StyleSheet Source="CatalogItemsStyles.css" />

        <Style TargetType="Frame">
            <Setter Property="BorderColor" Value="Blue" />
            <Setter Property="CornerRadius" Value="15" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="Text" Value="LEARN MORE" />
            <Setter Property="BorderRadius" Value="20" />
        </Style>
    </ContentPage.Resources>

    <ScrollView Orientation="Both">
        <FlexLayout>
            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Seated Monkey" StyleClass="header" />
                    <Label Text="This monkey is laid back and relaxed, and likes to watch the world go by."
/>
                    <Label Text="  &#x2022; Doesn't make a lot of noise" />
                    <Label Text="  &#x2022; Often smiles mysteriously" />
                    <Label Text="  &#x2022; Sleeps sitting up" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.SeatedMonkey.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Banana Monkey" StyleClass="header" />
                    <Label Text="Watch this monkey eat a giant banana." />
                    <Label Text="  &#x2022; More fun than a barrel of monkeys" />
                    <Label Text="  &#x2022; Banana not included" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.Banana.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>

            <Frame>
                <FlexLayout Direction="Column">
                    <Label Text="Face-Palm Monkey" StyleClass="header" />
                    <Label Text="This monkey reacts appropriately to ridiculous assertions and actions." />
                    <Label Text="  &#x2022; Cynical but not unfriendly" />
                    <Label Text="  &#x2022; Seven varieties of grimaces" />
                    <Label Text="  &#x2022; Doesn't laugh at your jokes" />
                    <Image Source="{local:ImageResource FlexLayoutDemos.Images.FacePalm.jpg}" />
                    <Label StyleClass="empty" />
                    <Button />
                </FlexLayout>
            </Frame>
        </FlexLayout>
    </ScrollView>
</ContentPage>
```

在第一行中引用的 CSS 样式表 `Resources` 部分：

```
<StyleSheet Source="CatalogItemsStyles.css" />
```

请注意，还在每个的三个项目中的两个元素，包括 `StyleClass` 设置：

```
<Label Text="Seated Monkey" StyleClass="header" />
...
<Label StyleClass="empty" />
```

这些是指中的选择器**CatalogItemsStyles.css**样式表：

```
frame {
    width: 300;
    height: 480;
    background-color: lightyellow;
    margin: 10;
}

label {
    margin: 4 0;
}

label.header {
    margin: 8 0;
    font-size: large;
    color: blue;
}

label.empty {
    flex-grow: 1;
}

image {
    height: 180;
    order: -1;
    align-self: center;
}

button {
    font-size: large;
    color: white;
    background-color: green;
}
```

多个 `FlexLayout` 此处引用附加可绑定属性。在 `label.empty` 选择器中，你将看到 `flex-grow` 属性设置为空的样式 `Label` 若要提供上述一些空白空间 `Button` 。 `image` 选择器将包含 `order` 属性和一个 `align-self` 属性，这两个对应于 `FlexLayout` 附加可绑定属性。

您已了解你可以直接在上设置属性 `FlexLayout` 您可以设置附加可绑定属性的子级和 `FlexLayout` 。或者，可以设置这些属性间接使用传统的基于 XAML 的样式或 CSS 样式。重要的是了解以及了解这些属性。这些属性是使 `FlexLayout` 真正灵活。

使用 Xamarin.University FlexLayout

Xamarin.Forms 3.0 Flex 布局中，通过Xamarin 学院课程

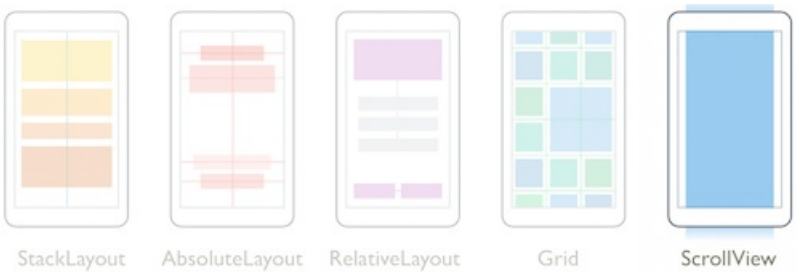
相关链接

- FlexLayoutDemos

Xamarin.Forms ScrollView

2018/8/1 • • Edit Online

`ScrollView` 包含布局，使他们能够滚动屏幕外。`ScrollView` 此外用于允许视图以显示键盘时自动移至屏幕的可见部分。



本文包含以下内容：

- **目的** –的目的 `ScrollView` 和一起使用时。
- **使用情况** –如何使用 `ScrollView` 在实践中。
- **属性** –可以读取和修改的公共属性。
- **方法** –可以调用以滚动视图的公共方法。
- **事件** –可用于侦听视图状态中的更改的事件。

目标

`ScrollView` 可以用于确保也在较小的手机上显示较大的视图。例如，适用于 iPhone 6s 的布局可能会剪裁 4 秒在 iPhone 上。使用 `ScrollView` 将允许的较小屏幕上显示的布局裁剪后的部分。

用法

> **NOTE**
> `ScrollView` 不应嵌套 s。此外，`ScrollView` 不应与其他控件提供滚动功能，如嵌套 s `ListView` 和 `WebView` 。

`ScrollView` 公开 `Content` 属性都被设置为单一视图或布局。请考虑以下示例使用非常大的字数后，跟布局 `Entry` ：

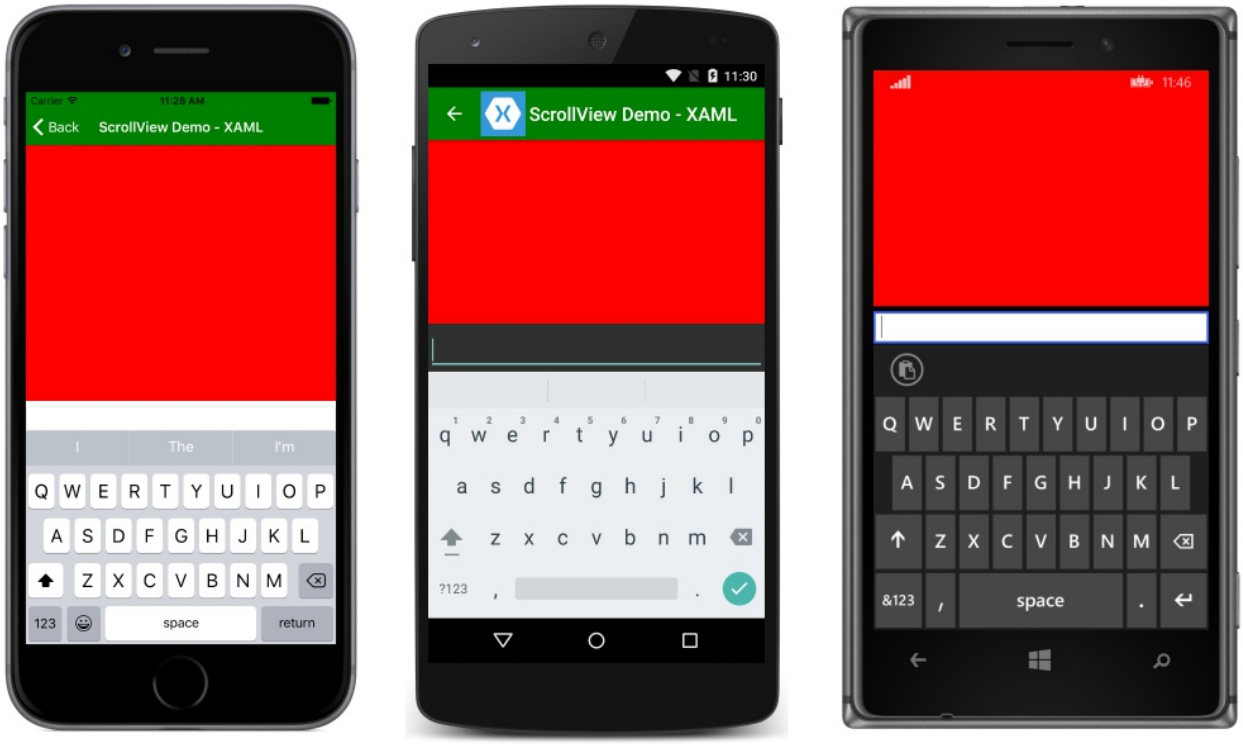
```
<ContentPage.Content>
    <ScrollView>
        <StackLayout>
            <BoxView BackgroundColor="Red" HeightRequest="600" WidthRequest="150" />
            <Entry />
        </StackLayout>
    </ScrollView>
</ContentPage.Content>
```

在 C# 中：

```
var scroll = new ScrollView();
Content = scroll;
var stack = new StackLayout();
stack.Children.Add(new BoxView { BackgroundColor = Color.Red,    HeightRequest = 600, WidthRequest = 600 });
stack.Children.Add(new Entry());
```

用户向下滚动，唯一之前 `BoxView` 可见：

请注意，当用户开始输入文本中的 `Entry`，视图将滚动以使其在屏幕上可见：



属性

`ScrollView` 定义以下属性：

- `ContentSize` 获取 `Size` 值，该值表示内容的大小。
- `Orientation` 获取或设置 `ScrollOrientation` 枚举值，该值表示滚动的方向 `ScrollView`。
- `ScrollX` 获取 `double`，它表示当前的滚动位置。
- `ScrollY` 获取 `double`，表示当前 Y 滚动位置。
- `HorizontalScrollBarVisibility` 获取或设置 `ScrollBarVisibility` 值，该值表示当水平滚动条是否可见。
- `VerticalScrollBarVisibility` 获取或设置 `ScrollBarVisibility` 值，该值表示当垂直滚动条是否可见。

方法

`ScrollView` 提供了 `ScrollToAsync` 方法，可用于滚动视图使用坐标或通过指定应为可见的特定视图。

当使用坐标，指定 `x` 和 `y` 坐标，以及一个布尔值，该值指示是否滚动应进行动画处理：

```
scroll.ScrollToAsync(0, 150, true); //scrolls so that the position at 150px from the top is visible

scroll.ScrollToAsync(label, ScrollToPosition.Start, true); //scrolls so that the label is at the start of the
list
```

滚动到特定元素时 `ScrollToPosition` 枚举指定元素在视图中显示其中：

- **Center** –滚动到视图的可见部分的中心元素。
- 结束–滚动到视图的可见部分末尾的元素。
- **MakeVisible** –滚动元素，以便在视图中可见。
- 启动–滚动到视图的可见部分的起始位置的元素。

`IsAnimated` 属性指定如何将滚动视图。当设置为 true，一个流畅的动画是将使用的而不是立即将内容移动到视图。

事件

`ScrollView` 定义只是一个活动 `Scrolled`。`Scrolled` 查看已完成滚动时引发。事件处理程序 `Scrolled` 采用 `ScrolledEventArgs`，其中包含 `ScrollX` 和 `ScrollY` 属性。以下演示如何使用当前的滚动位置的更新标签 `ScrollView`：

```
Label label = new Label { Text = "Position: " };
ScrollView scroll = new ScrollView();
scroll.Scrolled += (object sender, ScrolledEventArgs e) => {
    label.Text = "Position: " + e.ScrollX + " x " + e.ScrollY;
};
```

请注意，滚动位置可以是负数，由于滚动列表的末尾时的弹跳效果。

相关链接

- 布局（示例）
- BusinessTumble 示例（示例）

在 Xamarin.Forms 中的布局选项

2018/7/13 • • Edit Online

每个 Xamarin.Forms 视图具有类型 LayoutOptions 的 HorizontalOptions 和 VerticalOptions 属性。本文介绍的每个 LayoutOptions 值对的对齐方式和扩展视图的影响。

概述

`LayoutOptions` 结构封装两个布局首选项：

- 对齐**方式**– 视图的首选对齐方式，确定其位置和其父布局中的大小。
- 扩**展**– 仅由 `StackLayout` ，和如果可用，该值指示视图是否应使用多余的空格。

这些布局首选项可应用于 `View` 、相对于其父级，通过设置 `HorizontalOptions` 或者 `VerticalOptions` 属性 `View` 到一个公共字段从 `LayoutOptions` 结构。公共字段如下所示：

- `Start`
- `Center`
- `End`
- `Fill`
- `StartAndExpand`
- `CenterAndExpand`
- `EndAndExpand`
- `FillAndExpand`

`Start`， `Center`， `End`，和 `Fill` 字段用于定义父布局中的视图的对齐方式：

- 为水平对齐方式 `Start` 位置 `View` 上的左侧的父布局，并为垂直对齐方式，它将定位 `View` 顶部父布局。
- 水平和垂直对齐 `Center` 水平或垂直居中对齐 `View` 。
- 为水平对齐方式 `End` 位置 `View` 上的父布局，并为垂直对齐方式的右下方，它将定位 `View` 底部父布局中。
- 为水平对齐方式 `Fill` 确保 `View` 填充宽度的父布局和垂直对齐方式，它可确保 `View` 填充父布局的高度。

`StartAndExpand`， `CenterAndExpand`， `EndAndExpand`，和 `FillAndExpand` 值用于定义对齐方式的首选项，以及是否在视图将占用更多的空间，如果父级范围内可用 `StackLayout` 。

> **NOTE**
> 视图的默认值 `HorizontalOptions` 并 `VerticalOptions` 属性是 `LayoutOptions.Fill` 。

对齐方式

对齐方式控制当父布局包含未使用的空间时视图中其父级布局的定位方式（即父布局是其所有子项的合计大小更大）。

一个 `StackLayout` 仅尊重 `Start`， `Center`， `End`，以及 `Fill` `LayoutOptions` 上在相反方向中的子视图的字段到 `StackLayout` 方向。因此，子视图在垂直方向 `StackLayout` 可以设置其 `HorizontalOptions` 属性设置为之一 `Start`， `Center`， `End`，或 `Fill` 字段。同样，子视图中水平方向 `StackLayout` 可以设置其 `VerticalOptions` 属性设置为其中一个 `Start`， `Center`， `End`，或 `Fill` 字段。

一个 `StackLayout` 不遵从 `Start`、`Center`、`End`，以及 `Fill` `LayoutOptions` 上的方向相同的子视图的字段 `StackLayout` 方向。因此，垂直方向 `StackLayout` 将忽略 `Start`、`Center`、`End`，或 `Fill` 字段上设置如果 `VerticalOptions` 子视图的属性。同样，水平方向 `StackLayout` 将忽略 `Start`、`Center`、`End`，或 `Fill` 字段上设置如果 `HorizontalOptions` 子视图的属性。

> **NOTE**
>
> `LayoutOptions.Fill` 替代调整使用指定的请求的大小通常 `HeightRequest` 并 `WidthRequest` 属性。

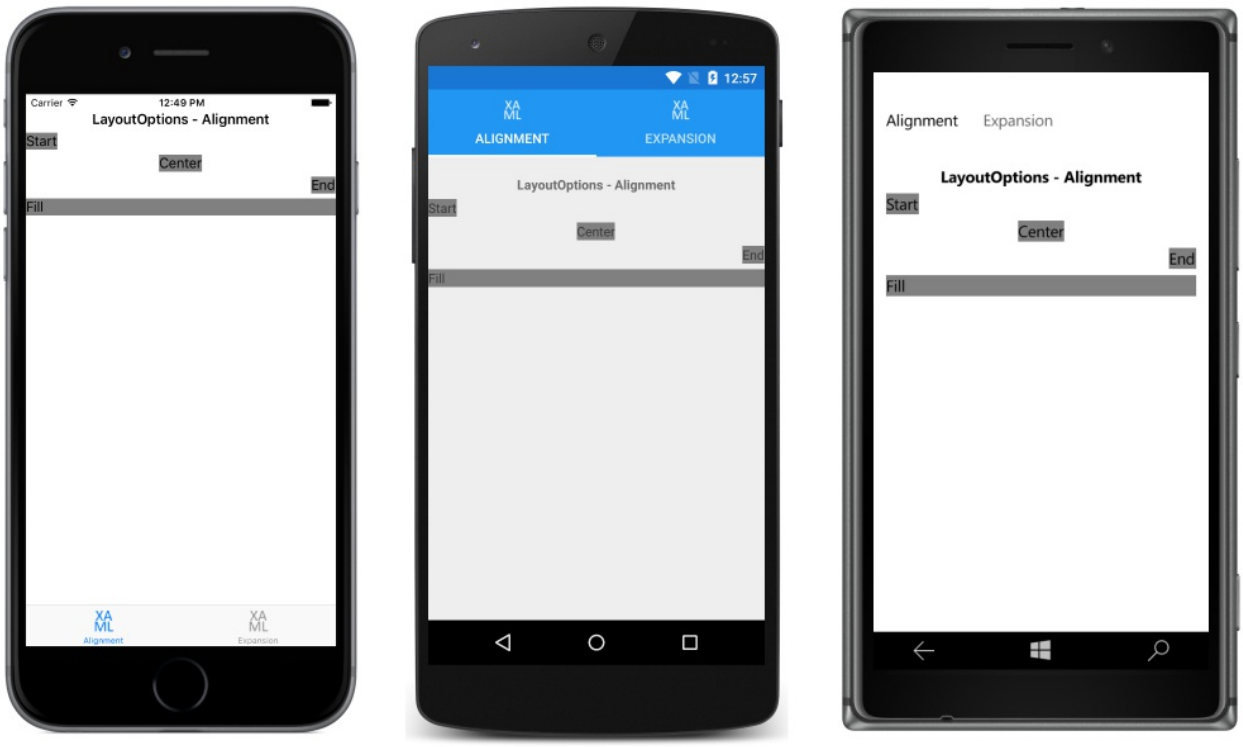
下面的 XAML 代码示例演示了垂直方向 `StackLayout` 其中每个子 `Label` 设置其 `HorizontalOptions` 属性中的四个对齐字段之一 `LayoutOptions` 结构：

```
<StackLayout Margin="0,20,0,0">
  ...
  <Label Text="Start" BackgroundColor="Gray" HorizontalOptions="Start" />
  <Label Text="Center" BackgroundColor="Gray" HorizontalOptions="Center" />
  <Label Text="End" BackgroundColor="Gray" HorizontalOptions="End" />
  <Label Text="Fill" BackgroundColor="Gray" HorizontalOptions="Fill" />
</StackLayout>
```

等效的 C# 代码如下所示：

```
Content = new StackLayout
{
  Margin = new Thickness(0, 20, 0, 0),
  Children = {
    ...
    new Label { Text = "Start", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Start },
    new Label { Text = "Center", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Center },
    new Label { Text = "End", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.End },
    new Label { Text = "Fill", BackgroundColor = Color.Gray, HorizontalOptions = LayoutOptions.Fill }
  }
};
```

该代码会导致下面的屏幕截图中所示的布局：



扩展

扩展控制是否视图将占用更多空间，是否在内可用 `StackLayout` 。如果 `StackLayout` 包含未使用的空间 (即 `StackLayout` 大于所有子级的组合大小)，未使用的空间同样由通过设置请求扩展的所有子视图共享其 `HorizontalOptions` 或 `VerticalOptions` 属性设置为 `LayoutOptions` 使用字段 `AndExpand` 后缀。请注意，当中的所有空间 `StackLayout` 是使用，扩展选项不起任何作用。

一个 `StackLayout` 仅展开其方向的方向中的子视图。因此，垂直方向 `StackLayout` 可以扩展设置的子视图及其 `VerticalOptions` 属性设置为之一 `StartAndExpand` ， `CenterAndExpand` ， `EndAndExpand` ，或 `FillAndExpand` 字段，如果 `StackLayout` 包含未使用的空间。同样，水平方向 `StackLayout` 可以扩展设置的子视图及其 `HorizontalOptions` 属性设置为其中一个 `StartAndExpand` ， `CenterAndExpand` ， `EndAndExpand` ，或 `FillAndExpand` 字段，如果 `StackLayout` 包含未使用的空间。

一个 `StackLayout` 无法展开子视图中与自己的方向相反的方向。因此，在垂直方向 `StackLayout` ，并设置 `HorizontalOptions` 属性上的子视图 `StartAndExpand` 具有相同的效果与将属性设置为 `Start` .

> **NOTE**
> 请注意，启用扩展不会更改视图的大小除非它使用 `LayoutOptions.FillAndExpand` 。

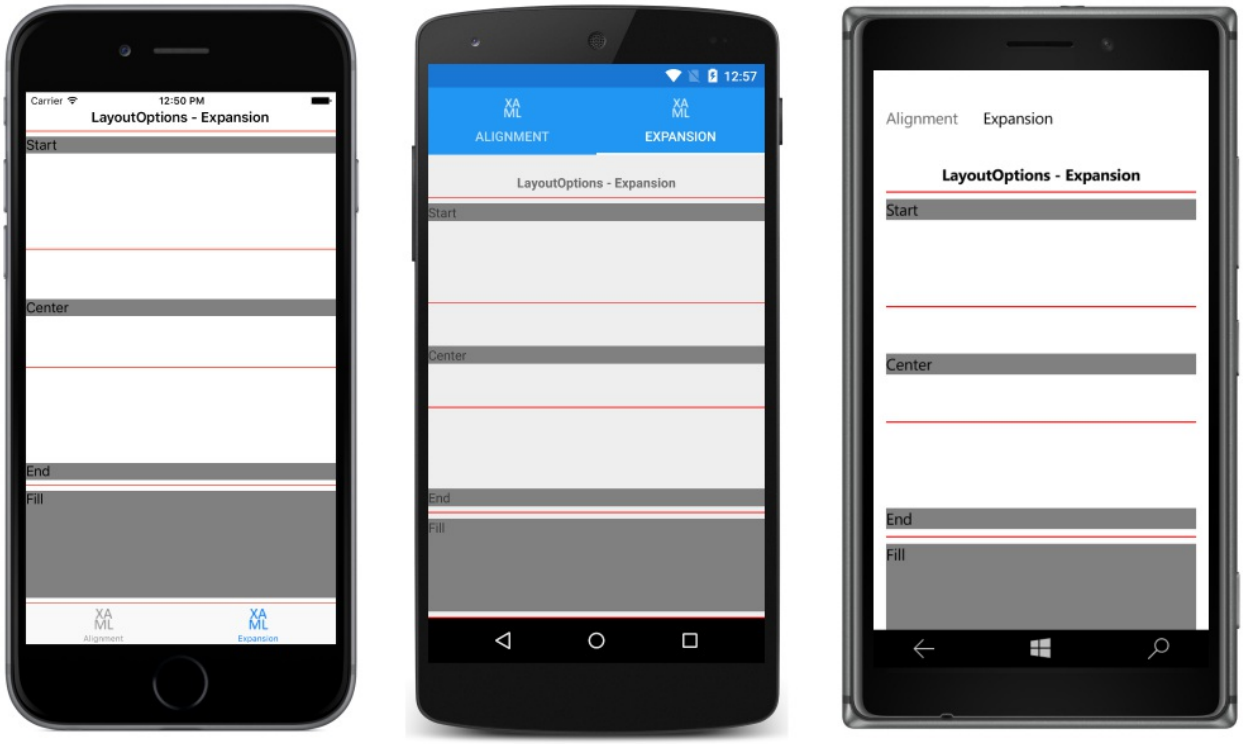
下面的 XAML 代码示例演示了垂直方向 `StackLayout` 其中每个子 `Label` 设置其 `VerticalOptions` 属性中的 4 个扩展字段之一 `LayoutOptions` 结构：

```
<StackLayout Margin="0,20,0,0">
  ...
  <BoxView BackgroundColor="Red" HeightRequest="1" />
  <Label Text="Start" BackgroundColor="Gray" VerticalOptions="StartAndExpand" />
  <BoxView BackgroundColor="Red" HeightRequest="1" />
  <Label Text="Center" BackgroundColor="Gray" VerticalOptions="CenterAndExpand" />
  <BoxView BackgroundColor="Red" HeightRequest="1" />
  <Label Text="End" BackgroundColor="Gray" VerticalOptions="EndAndExpand" />
  <BoxView BackgroundColor="Red" HeightRequest="1" />
  <Label Text="Fill" BackgroundColor="Gray" VerticalOptions="FillAndExpand" />
  <BoxView BackgroundColor="Red" HeightRequest="1" />
</StackLayout>
```

等效的 C# 代码如下所示：

```
Content = new StackLayout
{
  Margin = new Thickness(0, 20, 0, 0),
  Children = {
    ...
    new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
    new Label { Text = "StartAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.StartAndExpand },
    new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
    new Label { Text = "CenterAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.CenterAndExpand },
    new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
    new Label { Text = "EndAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.EndAndExpand },
    new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 },
    new Label { Text = "FillAndExpand", BackgroundColor = Color.Gray, VerticalOptions =
LayoutOptions.FillAndExpand },
    new BoxView { BackgroundColor = Color.Red, HeightRequest = 1 }
  }
};
```

该代码会导致下面的屏幕截图中所示的布局：



每个 `Label` 占用相同数量的空间内 `StackLayout` 。但是，只有最终 `Label` ，集及其 `VerticalOptions` 属性设置为 `FillAndExpand` 具有不同的大小。此外，每个 `Label` 分隔一个小的红色 `BoxView` ，这样空间 `Label` 占用轻松地查看。

总结

本文介绍影响每个 `LayoutOptions` 结构值对的对齐方式和视图，相对于其父级的扩展。 `Start` ， `Center` ， `End` ，和 `Fill` 字段用于定义父布局中的视图的对齐方式和 `StartAndExpand` ， `CenterAndExpand` ， `EndAndExpand` ，和 `FillAndExpand` 字段用于定义对齐方式的首选项，并确定是否在视图将占用更多空间，是否在内可用 `StackLayout` 。

相关链接

- LayoutOptions（示例）
- LayoutOptions

边距和填充

2018/7/13 • • Edit Online

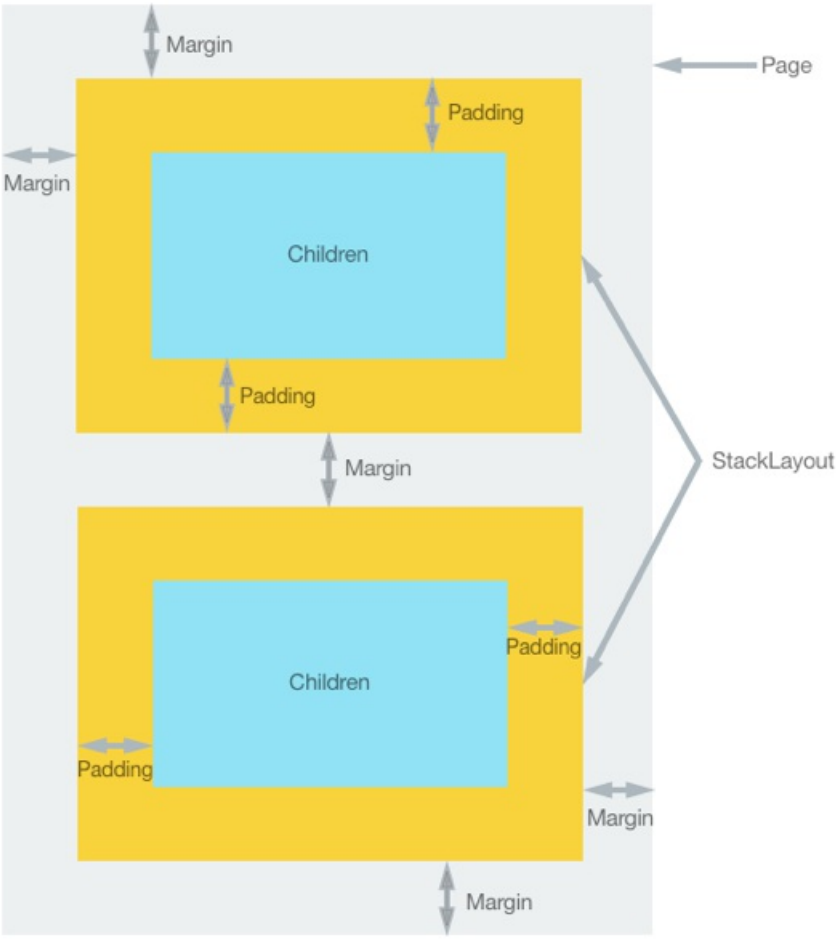
元素呈现用户界面中时，边距和填充属性控制布局行为。本文演示了两个属性，以及如何设置它们之间的差异。

概述

边距和填充是相关的布局的概念：

- `Margin` 属性表示一个元素和其相邻元素之间的距离，用来控制元素的呈现位置和与其相邻元素的呈现位置。 `Margin` 可以在指定值布局并视图类。
- `Padding` 属性表示元素与其子元素之间的距离，用于将控件从其自己的内容。 `Padding` 可以在指定值布局类。

下图说明了这两个概念：



请注意， `Margin` 值是累加性。因此，如果两个相邻元素指定边距为 20 像素，则元素之间的距离将 40 像素。此外，边距和填充是累加式的同时应用时，即边距和填充将是一个元素的任何内容之间的距离。

指定粗细

`Margin` 并 `Padding` 类型的两个属性均 `Thickness` 。有三种可能性创建时 `Thickness` 结构：

- 创建 `Thickness` 结构定义的单个统一的值。单个值应用于左侧、顶部、右侧和元素的底部均。
- 创建 `Thickness` 水平和垂直值定义的结构。水平值对称对称地应用于元素的上边和下边的垂直值应用于左侧和右侧的元素。

- 创建 `Thickness` 由四个非重复值应用于左侧、顶部、右侧和元素的底部均定义结构。

下面的 XAML 代码示例显示了所有三种可能性：

```
<StackLayout Padding="0,20,0,0">
  <Label Text="Xamarin.Forms" Margin="20" />
  <Label Text="Xamarin.iOS" Margin="10, 15" />
  <Label Text="Xamarin.Android" Margin="0, 20, 15, 5" />
</StackLayout>
```

以下代码示例显示相应的 C# 代码：

```
var stackLayout = new StackLayout {
  Padding = new Thickness(0,20,0,0),
  Children = {
    new Label { Text = "Xamarin.Forms", Margin = new Thickness (20) },
    new Label { Text = "Xamarin.iOS", Margin = new Thickness (10, 25) },
    new Label { Text = "Xamarin.Android", Margin = new Thickness (0, 20, 15, 5) }
  }
};
```

> **NOTE**
>
> `Thickness` 值可以是负数，通常裁剪或拉伸过度内容。

总结

本文说明了之间的差异 `Margin` 并 `Padding` 属性，以及如何将其设置。元素呈现用户界面中时，属性控制布局行为。

相关链接

- 边距
- 填充
- 粗细

设备方向

2018/11/13 • • Edit Online

请务必要考虑将如何使用你的应用程序和如何合并横向方向以改善用户体验。可以设计单独的布局以容纳多个方向和最佳使用的可用空间。在应用程序级别，可以禁用或启用旋转。

控制方向

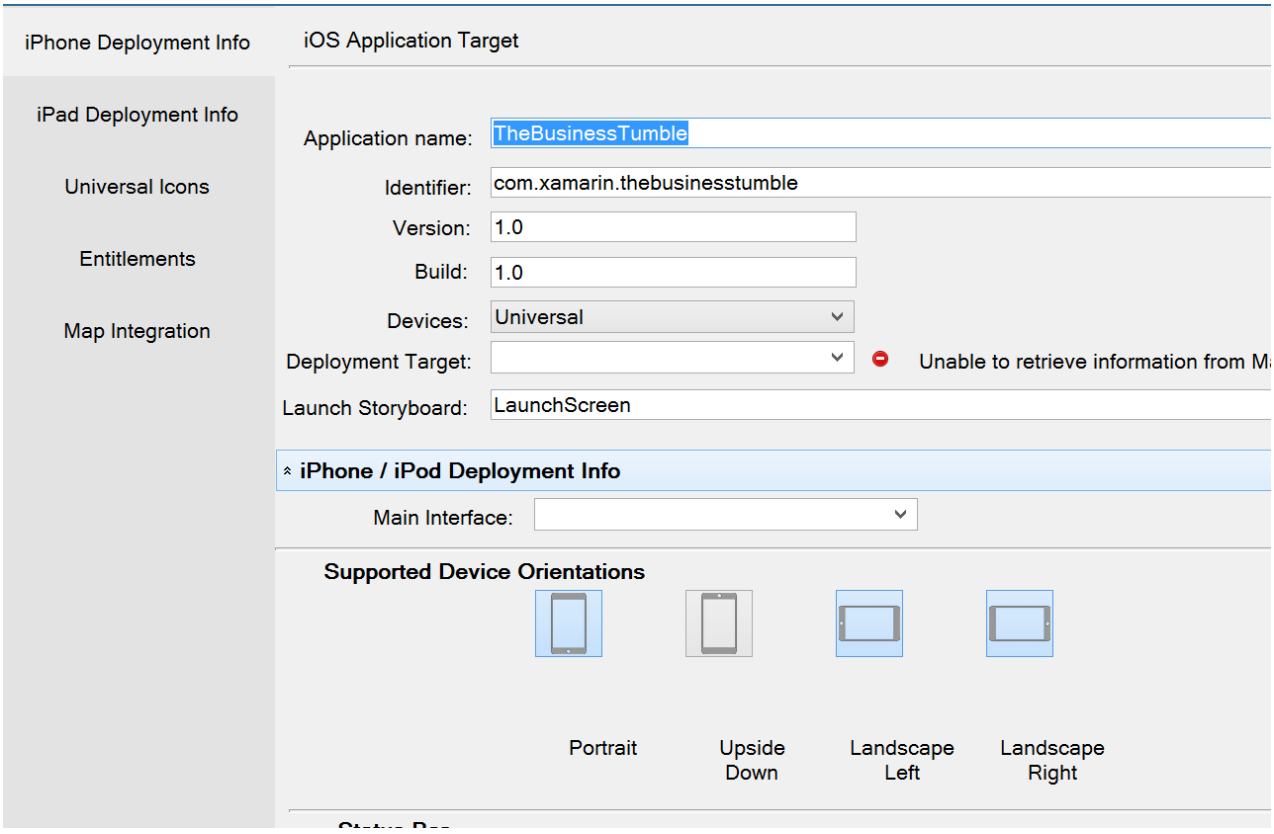
使用 Xamarin.Forms 时，控制设备方向的受支持的方法是为每个单独的项目使用的设置。

iOS

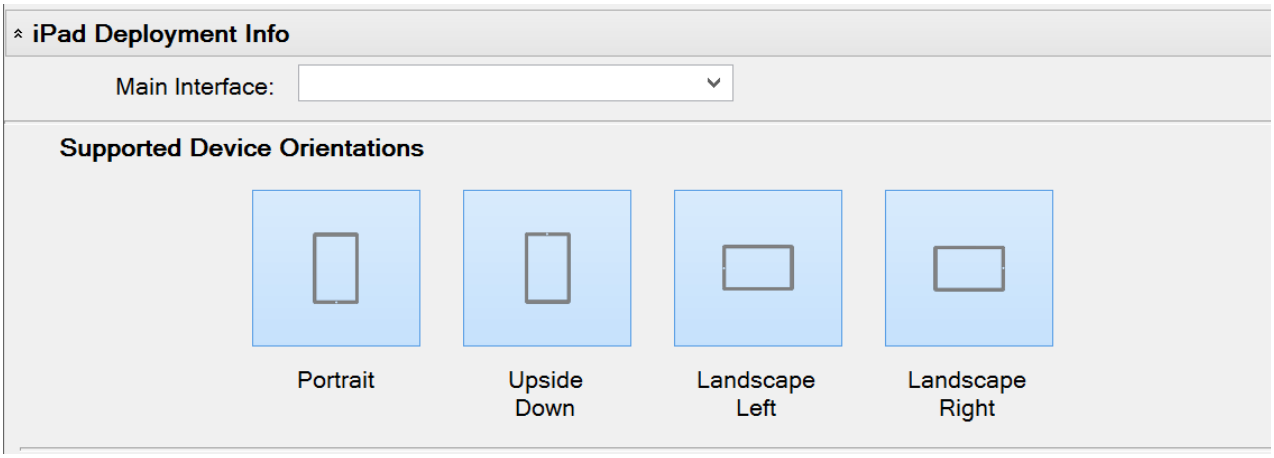
在 iOS 上，设备方向配置使用的应用程序**Info.plist**文件。如果该应用包含它作为目标，此文件将包含的 iPhone 和 iPod，方向设置，以及适用于 iPad 的设置。以下是对 IDE 的特定说明。在本文档的顶部使用 IDE 选项选择你想要看到的说明进行操作：

- Visual Studio
- Visual Studio for Mac

在 Visual Studio 中，打开 iOS 项目，并打开**Info.plist**。该文件将打开到配置面板中，从 iPhone 部署信息选项卡开始：



若要配置 iPad 方向，请选择**iPad 部署信息**左上方的面板中，然后选择从可用方向的选项卡：



Android

若要控制在 Android 上的方向，打开**MainActivity.cs**并设置使用属性修饰的方向 `MainActivity` 类：

```
namespace MyRotatingApp.Droid
{
    [Activity (Label = "MyRotatingApp.Droid", Icon = "@drawable/icon", Theme = "@style/MainTheme",
MainLauncher = true, ConfigurationChanges = ConfigChanges.ScreenSize | ConfigChanges.Orientation,
ScreenOrientation = ScreenOrientation.Landscape)] //This is what controls orientation
    public class MainActivity : FormsAppCompatActivity
    {
        protected override void OnCreate (Bundle bundle)
...
```

Xamarin.Android 支持多个选项用于指定方向：

- **横向**–强制应用程序方向为横向，而不考虑传感器数据。
- **纵向**–强制应用程序方向为纵向，而不考虑传感器数据。
- **用户**–导致应用程序使用用户的首选的方向显示。
- **后面**–会导致应用程序的方向是相同的方向活动它后面。
- **传感器**–导致应用程序的方向传感器，确定，即使用户已禁用自动旋转。
- **SensorLandscape** –会导致应用程序以使用横向方向，同时使用传感器数据来更改（以便在屏幕不为正面向下所示）面向在屏幕的方向。
- **SensorPortrait** –使应用程序使用纵向方向时使用的传感器数据更改（以便在屏幕不为正面向下所示）面向屏幕方向。
- **ReverseLandscape** –会导致应用程序以使用横向方向，面向的相反方向平常，以便显示"倒置。"
- **ReversePortrait** –使应用程序使用纵向方向，面向的相反方向平常，以便显示"倒置。"
- **FullSensor** –导致应用程序依赖于传感器数据以选择正确的方向（从可能 4）。
- **FullUser** –使应用程序使用用户的方向首选项。如果启用自动旋转，则可以使用所有 4 个方向。
- **UserLandscape** – *[不支持]* 会导致应用程序以使用横向方向，除非用户具有自动旋转启用，在这种情况下它将使用若要确定方向的传感器。此选项将中断编译。
- **UserPortrait** – *[不支持]* 使应用程序使用纵向方向，除非用户具有自动旋转启用，在这种情况下它将使用若要确定方向的传感器。此选项将中断编译。
- **锁定** – *[不支持]* 使应用程序使用的屏幕方向，无论它是启动时，而无需对设备中的更改作出响应的物理方向。此选项将中断编译。

请注意，本机 Android Api 提供了很多方向的管理方式的控制，包括显式与用户相矛盾的选项以表示首选项。

通用 Windows 平台

在通用 Windows 平台 (UWP)，支持的方向中设置**Package.appxmanifest**文件。打开清单，将显示在其中选择支持的方向配置面板。

对方向中的更改作出反应

Xamarin.Forms 不提供任何本机事件用于通知的共享代码中的方向更改您的应用程序。但是，`SizeChanged`的事件`Page`，则会激发的宽度或高度`Page`更改。时的宽度`Page`大于高度，在设备处于横向模式。有关详细信息，请参阅显示基于屏幕方向的映像。

> **NOTE**
>
> 没有用于接收通知的方向更改共享代码中的现有的免费 NuGet 包。请参阅GitHub 存储库有关详细信息。

或者，则可以重写`OnSizeAllocated`方法`Page`，插入任何布局更改那里逻辑。`OnSizeAllocated`调用方法时`Page`分配新的大小，这种情况发生时将设备旋转。请注意的基实现`OnSizeAllocated`执行重要布局功能，因此，必须在重写中调用基实现：

```
protected override void OnSizeAllocated(double width, double height)
{
    base.OnSizeAllocated(width, height); //must be called
}
```

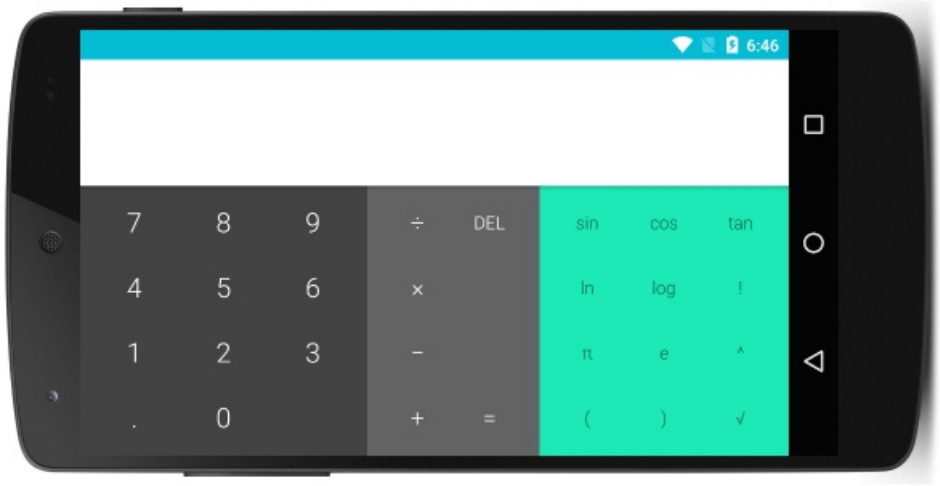
未能执行该步骤将导致非正常运行的页面。

请注意，`OnSizeAllocated`方法可能会多次调用旋转设备时。每次更改你的布局是一种浪费的资源，并可能会导致闪烁。请考虑使用在页面内的一个实例变量跟踪是否方向为横向或纵向，并只重绘更改时：

```
private double width = 0;
private double height = 0;

protected override void OnSizeAllocated(double width, double height)
{
    base.OnSizeAllocated(width, height); //must be called
    if (this.width != width || this.height != height)
    {
        this.width = width;
        this.height = height;
        //reconfigure layout
    }
}
```

后检测到在设备方向更改，你可能想要添加或删除与用户界面的可用空间的更改做出反应的其他视图。例如，考虑纵向时的每个平台上内置的计算器：

和横向:



请注意，应用程序通过添加更多的功能在环境中充分利用可用空间。

响应式布局

就可以使用内置的布局，以便适当地过渡旋转设备时的设计界面。设计将继续响应方向中的更改时很吸引人的接口时请考虑以下一般规则：

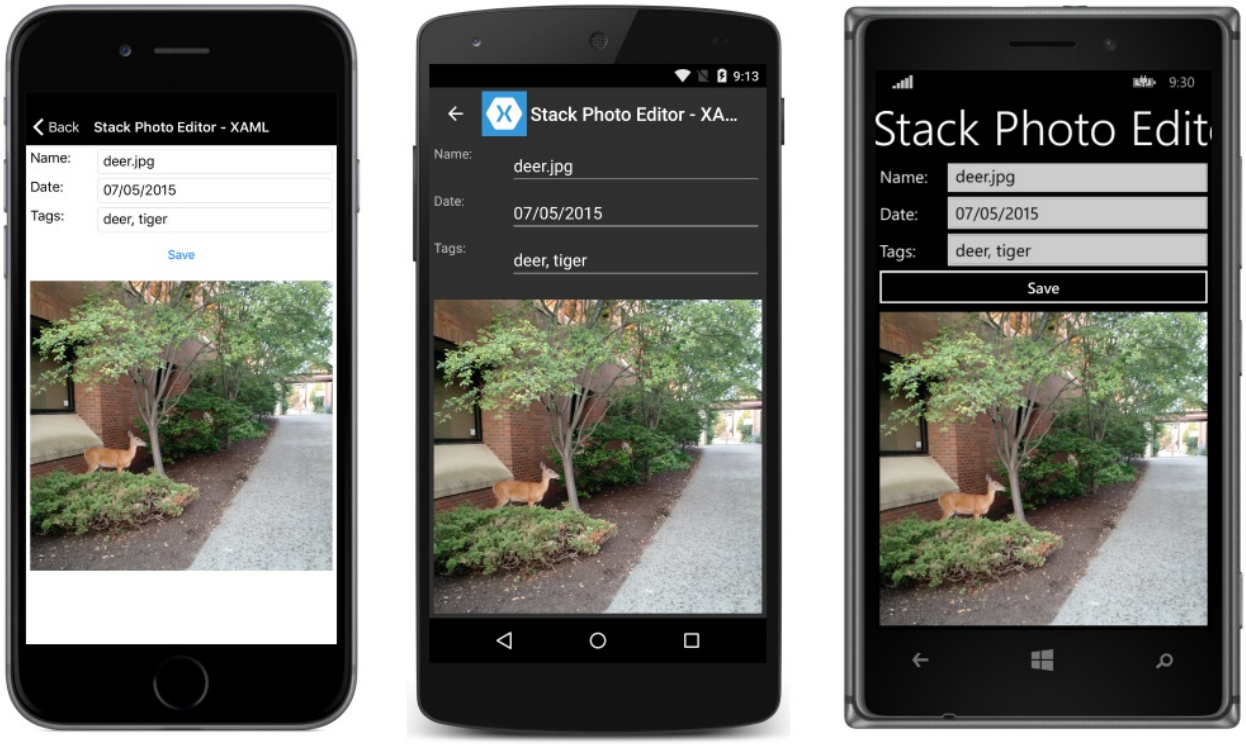
- **注意比率**–时方面比进行某些假设，方向中的更改可能会导致问题。例如，一个视图，它会在 1/3 的纵向时的屏幕的垂直空间中具有足够的空间可能无法放入 1/3 的环境中的垂直空间。
- **请小心使用绝对值**–意义纵向时的绝对（像素）值在环境中可能毫无意义。如果绝对值是必需的使用嵌套的布局隔离它们的影响。例如，它将合理地使用中的绝对值 `TableView``ItemTemplate` 时已有保证的统一高度的项模板。

上述规则同样适用时通常实现接口，用于多个屏幕大小以及被视为最佳做法。本指南的其余部分将介绍在 Xamarin.Forms 中使用上述每种主要布局的响应式布局的具体示例。

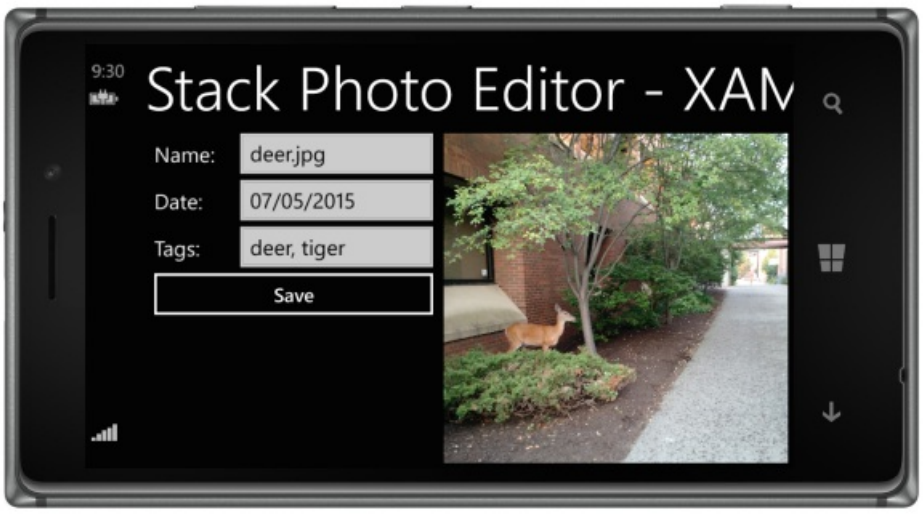
> **NOTE**
>
> 为清楚起见，以下各节演示如何实现使用的只是一种类型的响应式布局 `Layout` 一次。在实践中，它通常会更简单混合 `Layout` 以实现所需的布局使用更简单或最直观 `Layout` 为每个组件。

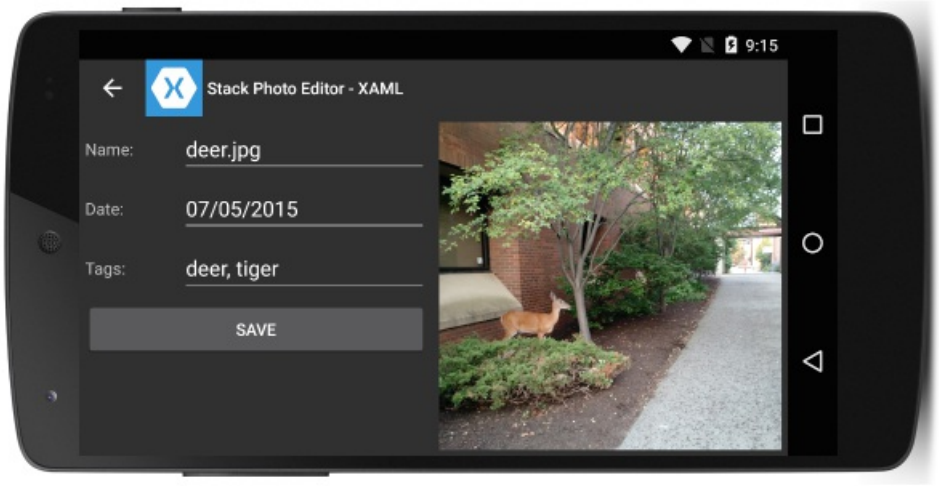
StackLayout

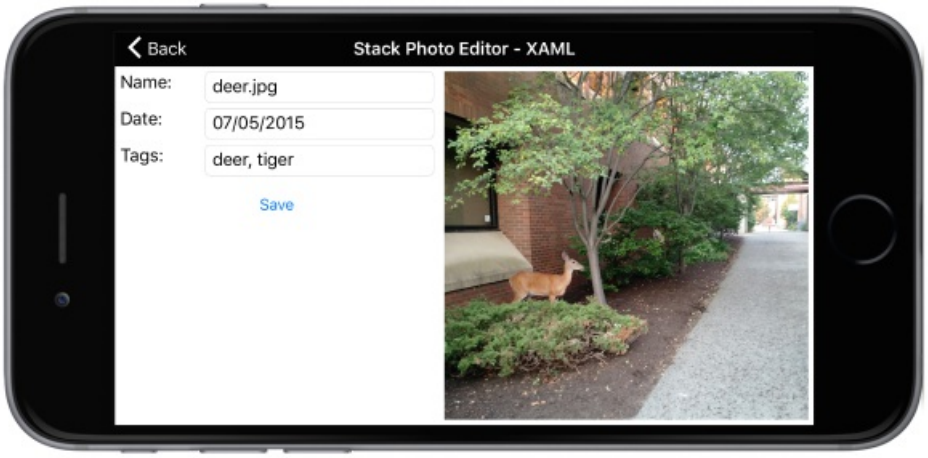
请考虑以下应用程序，在纵向显示：



和横向：







这是使用以下 XAML 来实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="ResponsiveLayout.StackLayoutPageXaml"
Title="Stack Photo Editor - XAML">
    <ContentPage.Content>
        <StackLayout Spacing="10" Padding="5" Orientation="Vertical"
        x:Name="outerStack"> <!-- can change orientation to make responsive -->
            <ScrollView>
                <StackLayout Spacing="5" HorizontalOptions="FillAndExpand"
                    WidthRequest="1000">
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Name: " WidthRequest="75"
                            HorizontalOptions="Start" />
                        <Entry Text="deer.jpg"
                            HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Date: " WidthRequest="75"
                            HorizontalOptions="Start" />
                        <Entry Text="07/05/2015"
                            HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Label Text="Tags:" WidthRequest="75"
                            HorizontalOptions="Start" />
                        <Entry Text="deer, tiger"
                            HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                    <StackLayout Orientation="Horizontal">
                        <Button Text="Save" HorizontalOptions="FillAndExpand" />
                    </StackLayout>
                </StackLayout>
            </ScrollView>
            <Image  Source="deer.jpg" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

某些 C# 用于更改的方向 `outerStack` 基于设备的方向：

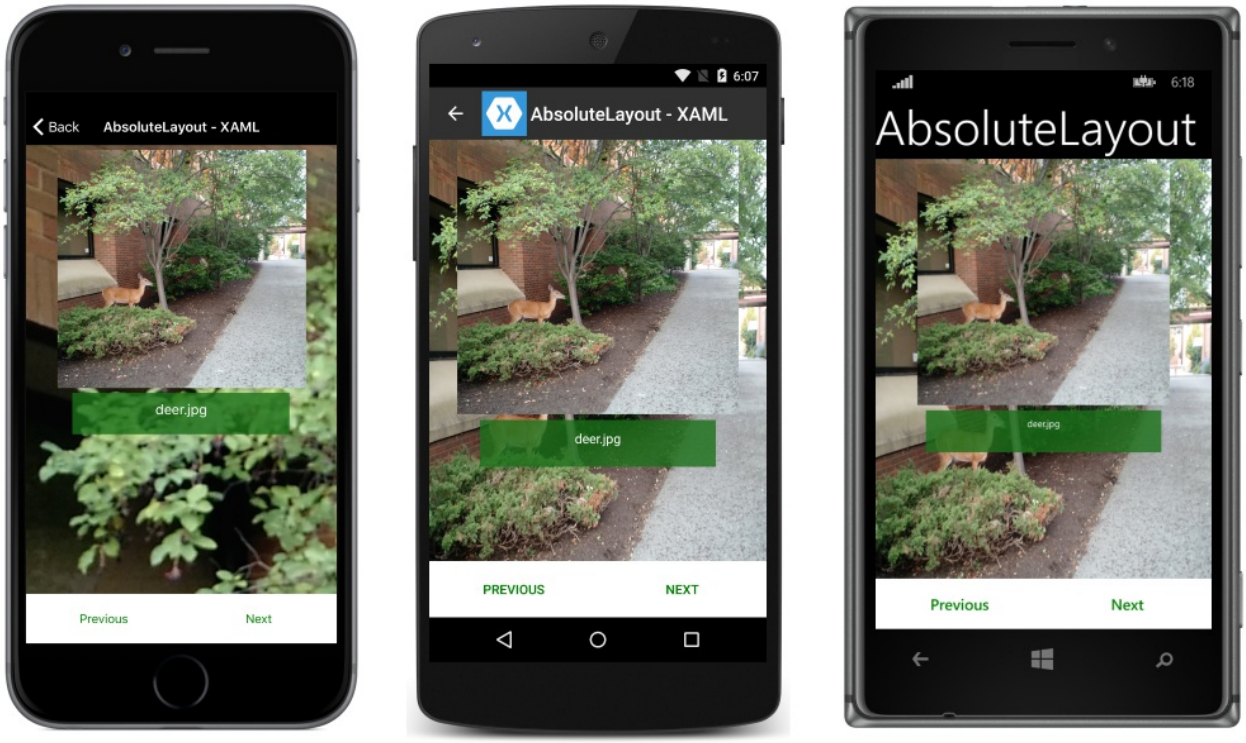
```
protected override void OnSizeAllocated (double width, double height){
    base.OnSizeAllocated (width, height);
    if (width != this.width || height != this.height) {
        this.width = width;
        this.height = height;
        if (width > height) {
            outerStack.Orientation = StackOrientation.Horizontal;
        } else {
            outerStack.Orientation = StackOrientation.Vertical;
        }
    }
}
```

请注意以下事项：

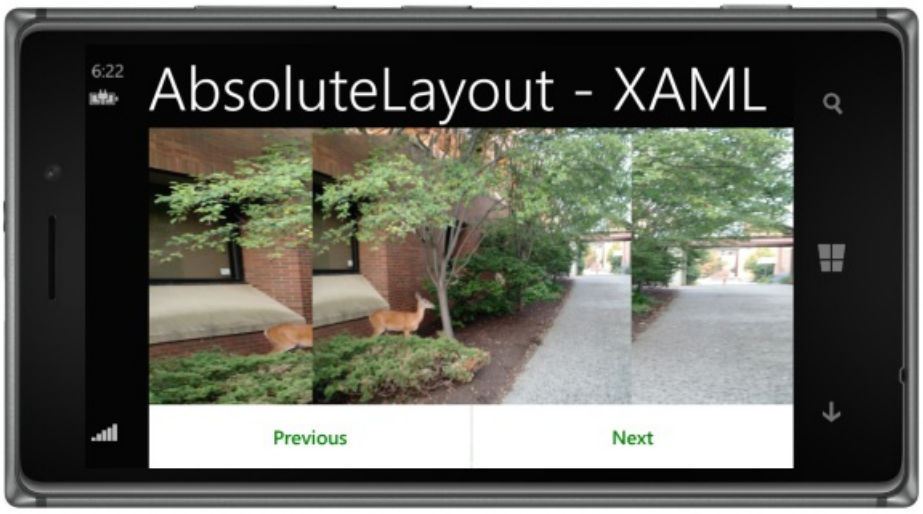
- `outerStack` 将调整为作为水平或垂直方向，以最有效地利用可用空间根据堆栈中显示的图像和控件。

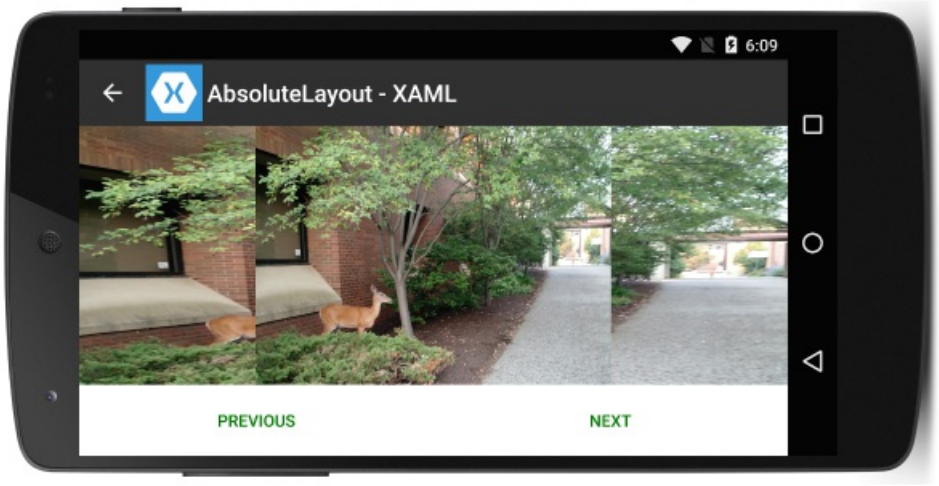
AbsoluteLayout

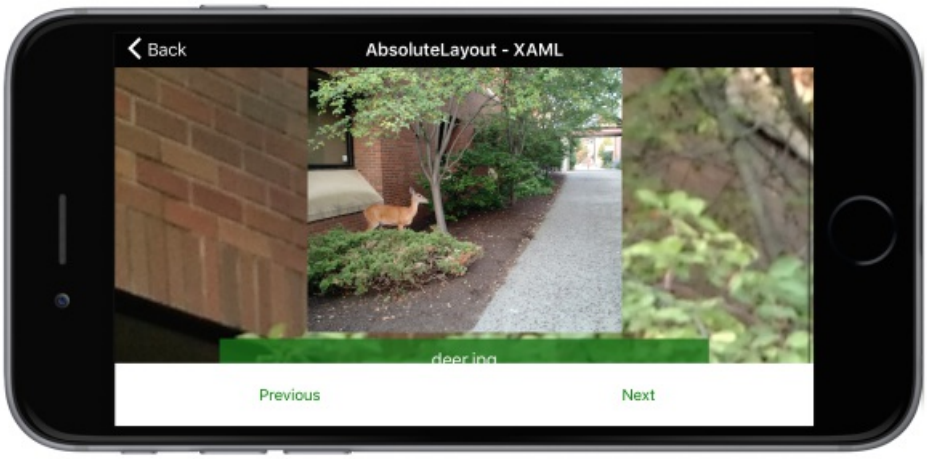
请考虑以下应用程序，在纵向显示：


Back
AbsoluteLayout - XAML
deer.jpg
Previous
Next
6:07
AbsoluteLayout - XAML
deer.jpg
PREVIOUS
NEXT
6:18
AbsoluteLayout
deer.jpg
Previous
Next

和横向：







这是使用以下 XAML 来实现：

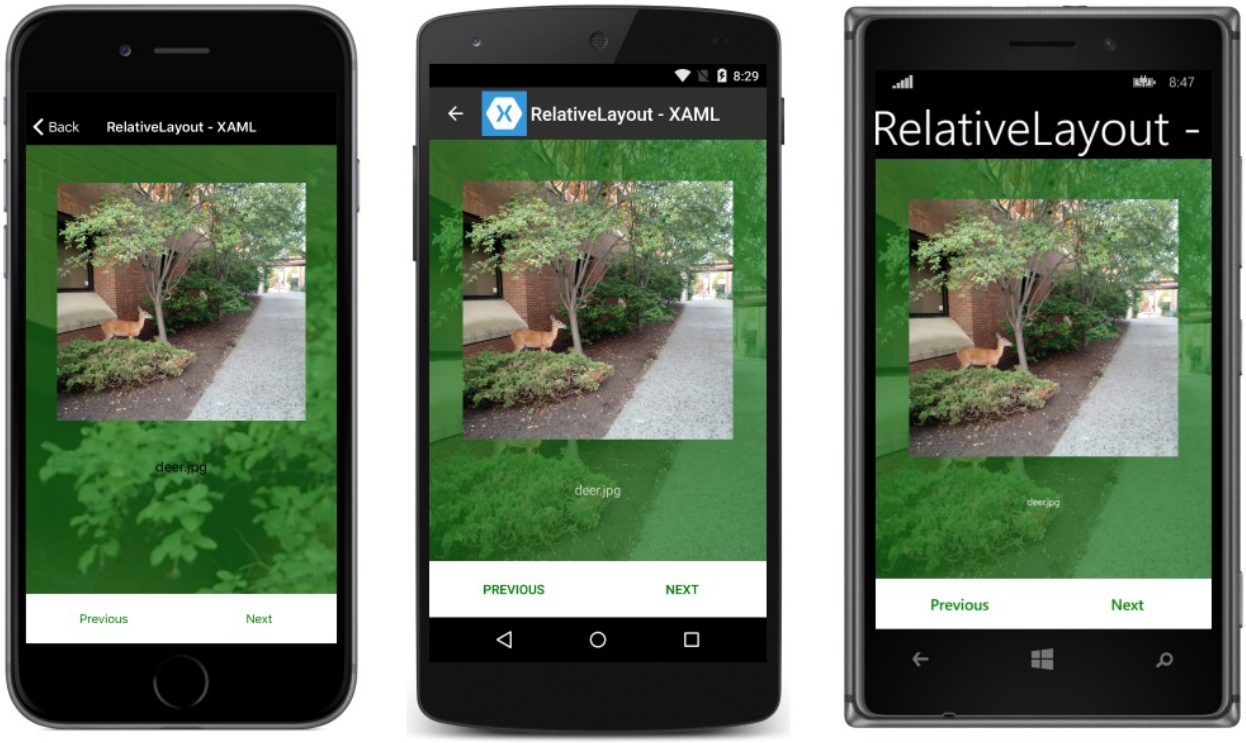
```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="ResponsiveLayout.AbsoluteLayoutPageXaml"
Title="AbsoluteLayout - XAML" BackgroundImage="deer.jpg">
    <ContentPage.Content>
        <AbsoluteLayout>
            <ScrollView AbsoluteLayout.LayoutBounds="0,0,1,1"
                AbsoluteLayout.LayoutFlags="PositionProportional,SizeProportional">
                <AbsoluteLayout>
                    <Image Source="deer.jpg"
                        AbsoluteLayout.LayoutBounds=".5,0,300,300"
                        AbsoluteLayout.LayoutFlags="PositionProportional" />
                    <BoxView Color="#CC1A7019" AbsoluteLayout.LayoutBounds=".5
                        300,.7,50" AbsoluteLayout.LayoutFlags="XProportional
                        WidthProportional" />
                    <Label Text="deer.jpg" AbsoluteLayout.LayoutBounds = ".5
                        310,1, 50" AbsoluteLayout.LayoutFlags="XProportional
                        WidthProportional" HorizontalTextAlignment="Center" TextColor="White" />
                </AbsoluteLayout>
            </ScrollView>
            <Button Text="Previous" AbsoluteLayout.LayoutBounds="0,1,.5,60"
                AbsoluteLayout.LayoutFlags="PositionProportional
                    WidthProportional"
                BackgroundColor="White" TextColor="Green" BorderRadius="0" />
            <Button Text="Next" AbsoluteLayout.LayoutBounds="1,1,.5,60"
                AbsoluteLayout.LayoutFlags="PositionProportional
                    WidthProportional" BackgroundColor="White"
                    TextColor="Green" BorderRadius="0" />
        </AbsoluteLayout>
    </ContentPage.Content>
</ContentPage>
```

请注意以下事项：

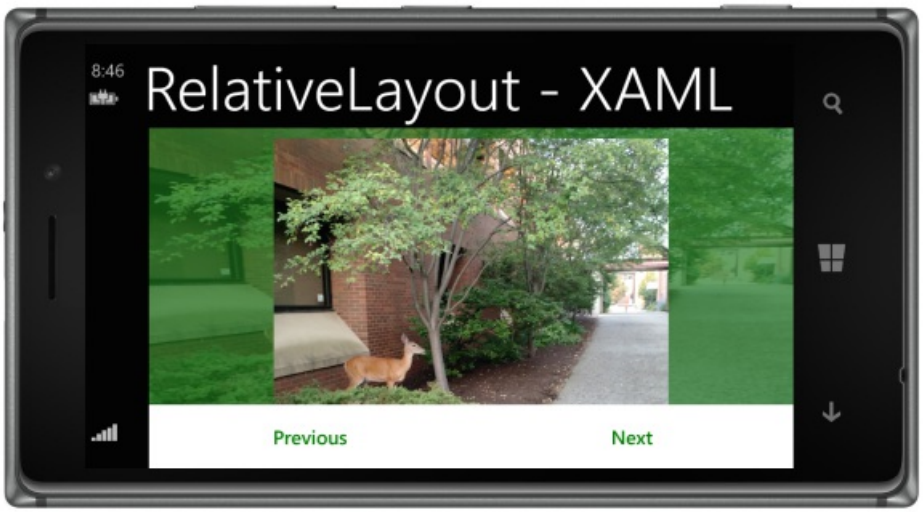
- 由于页面布局的方式，不是需要程序代码引入响应能力。
- `ScrollView` 用于允许甚至时屏幕的高度是固定的按钮和图像的高度之和小于为可见的标签。

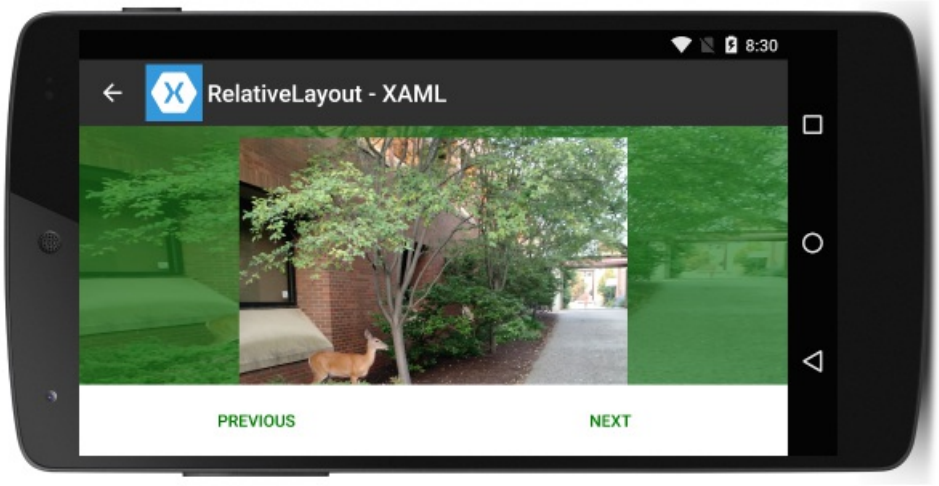
RelativeLayout

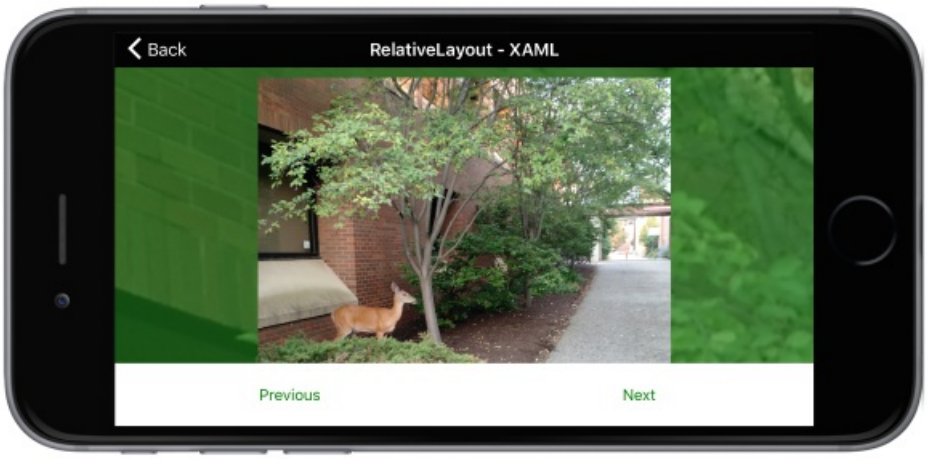
请考虑以下应用程序，在纵向显示：


Back
RelativeLayout - XAML
deer.jpg
Previous
Next
8:29
RelativeLayout - XAML
deer.jpg
PREVIOUS
NEXT
8:47
RelativeLayout -
deer.jpg
Previous
Next

和横向：







这是使用以下 XAML 来实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="ResponsiveLayout.RelativeLayoutPageXaml"
Title="RelativeLayout - XAML"
BackgroundImage="deer.jpg">
    <ContentPage.Content>
        <RelativeLayout x:Name="outerLayout">
            <BoxView BackgroundColor="#AA1A7019"
                RelativeLayout.WidthConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=1}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
```

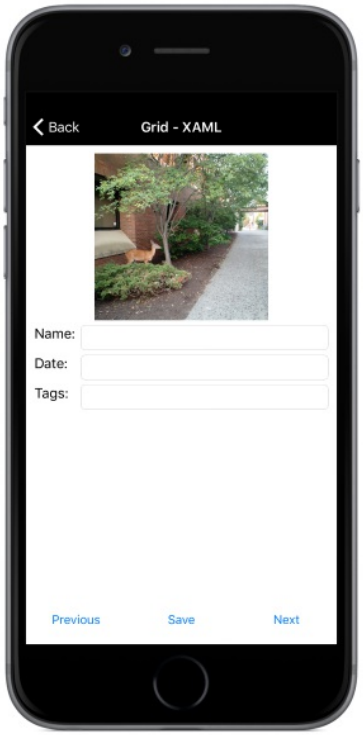
```
                    Type=RelativeToParent,Property=Height,Factor=1}"
                RelativeLayout.XConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
                RelativeLayout.YConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Height,Factor=0,Constant=0}" />
            <ScrollView
                RelativeLayout.WidthConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=1}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
                RelativeLayout.XConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
                RelativeLayout.YConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Height,Factor=0,Constant=0}">
                <RelativeLayout>
                    <Image Source="deer.jpg" x:Name="imageDeer"
                        RelativeLayout.WidthConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Width,Factor=.8}"
                        RelativeLayout.XConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Width,Factor=.1}"
                        RelativeLayout.YConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Height,Factor=0,Constant=10}" />
                    <Label Text="deer.jpg" HorizontalTextAlignment="Center"
                        RelativeLayout.WidthConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Width,Factor=1}"
                        RelativeLayout.HeightConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Height,Factor=0,Constant=75}"
                        RelativeLayout.XConstraint="{ConstraintExpression
                            Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
                        RelativeLayout.YConstraint="{ConstraintExpression
                            Type=RelativeToView,ElementName=imageDeer,Property=Height,Factor=1,Constant=20}"
/>
                </RelativeLayout>

            </ScrollView>

            <Button Text="Previous" BackgroundColor="White" TextColor="Green" BorderRadius="0"
                RelativeLayout.YConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
                RelativeLayout.XConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=0,Constant=0}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=0,Constant=60}"
                RelativeLayout.WidthConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=.5}"
                 />
            <Button Text="Next" BackgroundColor="White" TextColor="Green" BorderRadius="0"
                RelativeLayout.XConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=.5}"
                RelativeLayout.YConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Height,Factor=1,Constant=-60}"
                RelativeLayout.HeightConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=0,Constant=60}"
                RelativeLayout.WidthConstraint="{ConstraintExpression
                    Type=RelativeToParent,Property=Width,Factor=.5}"
                />
        </RelativeLayout>
    </ContentPage.Content>
</ContentPage>
```

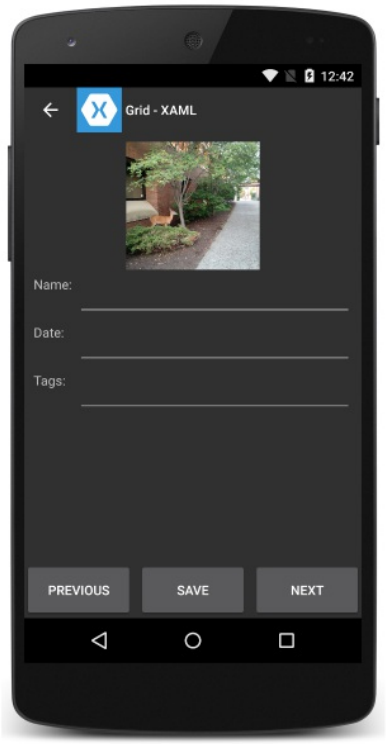
请注意以下事项：

- 由于页面布局的方式，不是需要程序代码引入响应能力。
- `ScrollView` 用于允许甚至时屏幕的高度是固定的按钮和图像的高度之和小于为可见的标签。

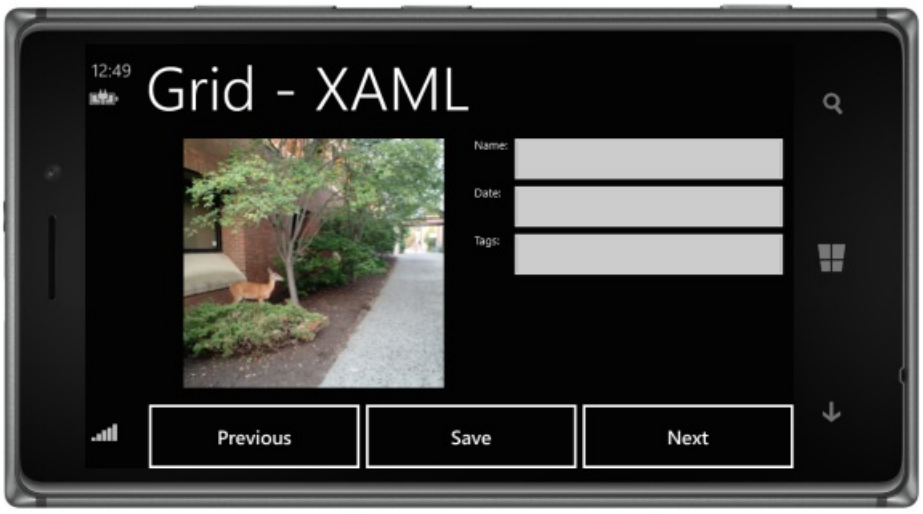
Grid

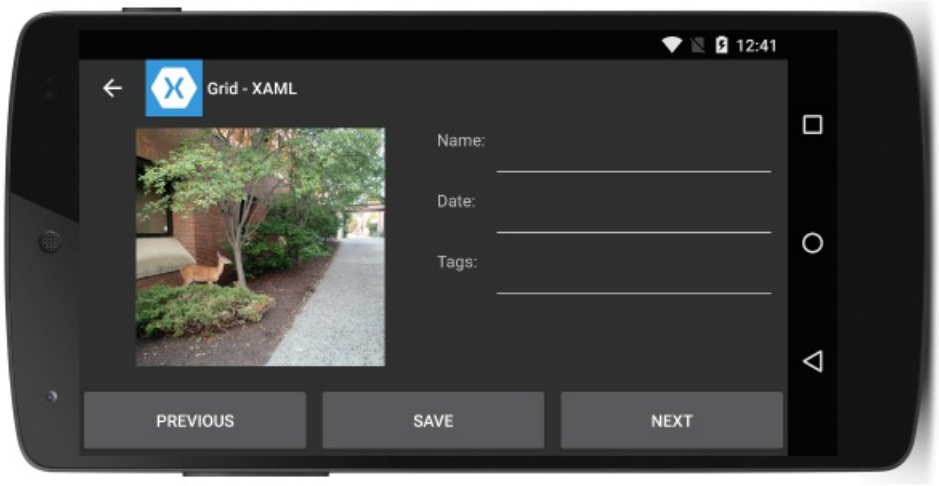
请考虑以下应用程序，在纵向显示：

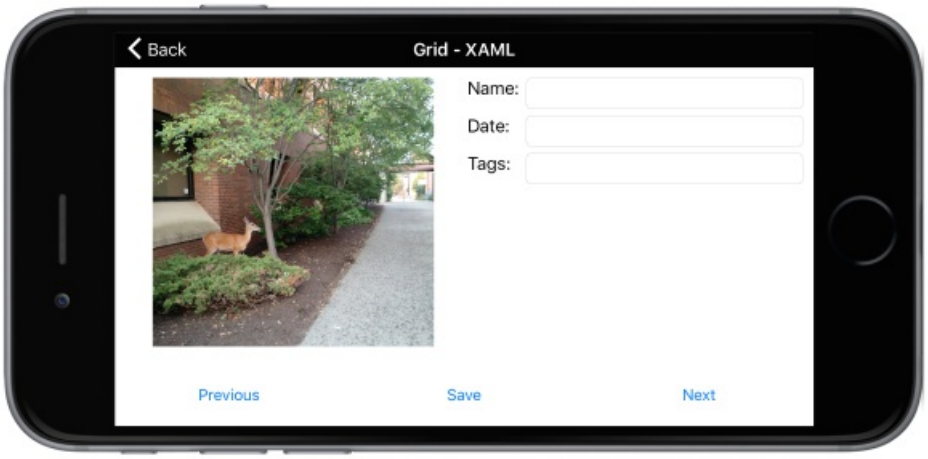




和横向：







这是使用以下 XAML 来实现：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="ResponsiveLayout.GridPageXaml"
Title="Grid - XAML">
    <ContentPage.Content>
        <Grid x:Name="outerGrid">
            <Grid.RowDefinitions>
                <RowDefinition Height="*" />
                <RowDefinition Height="60" />
            </Grid.RowDefinitions>
            <Grid x:Name="innerGrid" Grid.Row="0" Padding="10">
                <Grid.RowDefinitions>
                    <RowDefinition Height="*" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Image Source="deer.jpg" Grid.Row="0" Grid.Column="0" HeightRequest="300" WidthRequest="300"
/>
                <Grid x:Name="controlsGrid" Grid.Row="0" Grid.Column="1" >
                    <Grid.RowDefinitions>
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                        <RowDefinition Height="Auto" />
                    </Grid.RowDefinitions>
                    <Grid.ColumnDefinitions>
                        <ColumnDefinition Width="Auto" />
                        <ColumnDefinition Width="*" />
                    </Grid.ColumnDefinitions>
                    <Label Text="Name:" Grid.Row="0" Grid.Column="0" />
                    <Label Text="Date:" Grid.Row="1" Grid.Column="0" />
                    <Label Text="Tags:" Grid.Row="2" Grid.Column="0" />
                    <Entry Grid.Row="0" Grid.Column="1" />
                    <Entry Grid.Row="1" Grid.Column="1" />
                    <Entry Grid.Row="2" Grid.Column="1" />
                </Grid>
            </Grid>
            <Grid x:Name="buttonsGrid" Grid.Row="1">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                    <ColumnDefinition Width="*" />
                </Grid.ColumnDefinitions>
                <Button Text="Previous" Grid.Column="0" />
                <Button Text="Save" Grid.Column="1" />
                <Button Text="Next" Grid.Column="2" />
            </Grid>
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

以下过程与代码一起处理旋转更改：

```
private double width;
private double height;

protected override void OnSizeAllocated (double width, double height){
    base.OnSizeAllocated (width, height);
    if (width != this.width || height != this.height) {
        this.width = width;
        this.height = height;
        if (width > height) {
            innerGrid.RowDefinitions.Clear();
            innerGrid.ColumnDefinitions.Clear ();
            innerGrid.RowDefinitions.Add (new RowDefinition{ Height = new GridLength (1, GridUnitType.Star)
});
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition { Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.Children.Remove (controlsGrid);
            innerGrid.Children.Add (controlsGrid, 1, 0);
        } else {
            innerGrid.ColumnDefinitions.Clear ();
            innerGrid.ColumnDefinitions.Add (new ColumnDefinition{ Width = new GridLength (1,
GridUnitType.Star) });
            innerGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Auto)
});
            innerGrid.RowDefinitions.Add (new RowDefinition { Height = new GridLength (1, GridUnitType.Star)
});
            innerGrid.Children.Remove (controlsGrid);
            innerGrid.Children.Add (controlsGrid, 0, 1);
        }
    }
}
```

请注意以下事项：

- 页面布局的方式，因此没有方法来更改网格放置控件。

相关链接

- 布局（示例）
- BusinessTumble 示例（示例）
- 响应式布局（示例）
- 显示基于屏幕方向的映像

对于平板电脑和桌面应用程序的布局

2018/7/13 • • Edit Online

Xamarin.Forms 支持所有设备类型支持的平台上可用，因此除了手机，应用还可以运行：

- Ipad，
- Android 平板电脑
- Windows 平板电脑和台式计算机（运行 Windows 10）。

此页简要讨论了。

- 受支持设备类型，和
- 如何优化适用于平板电脑与手机布局。

设备类型

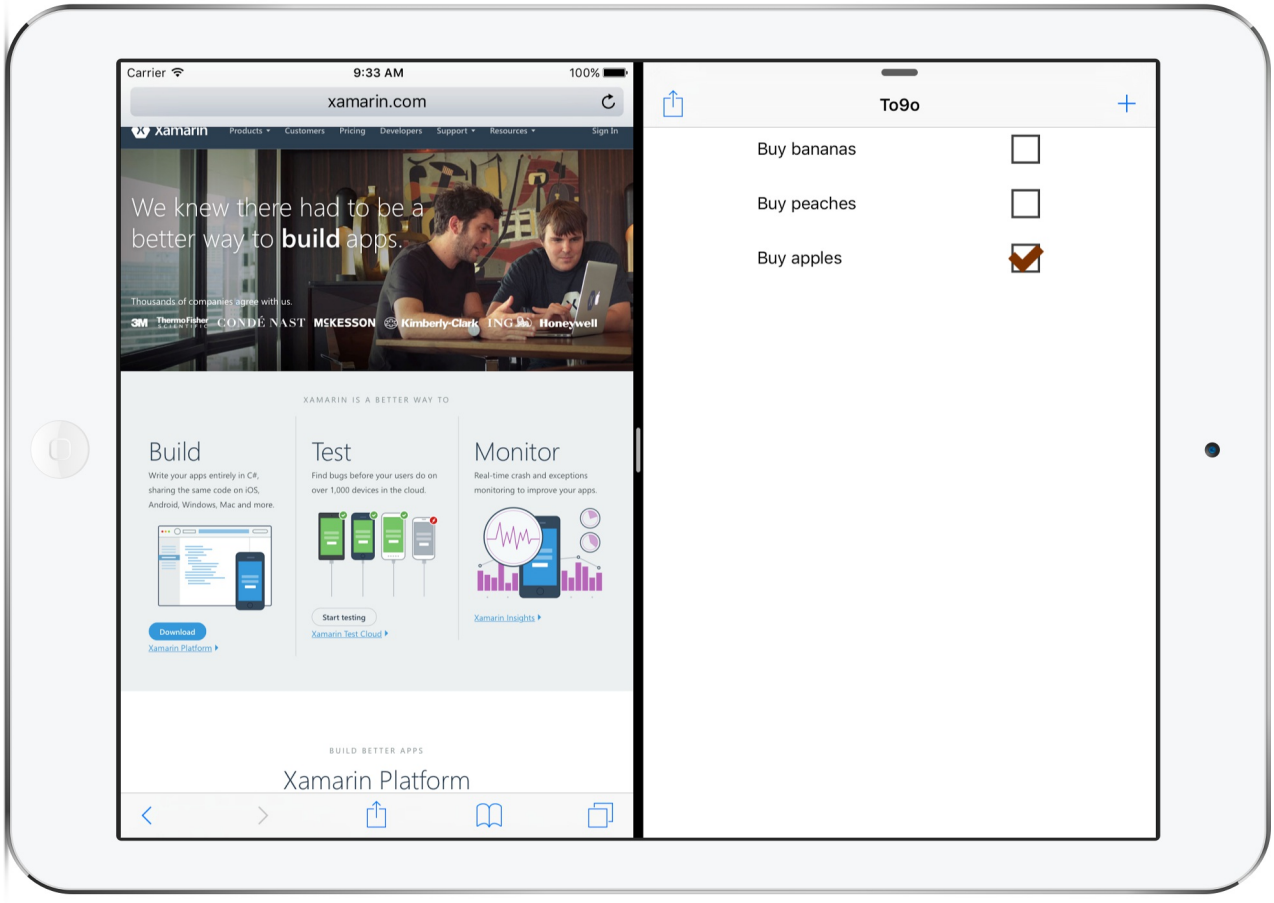
较大屏幕设备是可用于所有 Xamarin.Forms 支持的平台。

Ipad (iOS)

使用 Xamarin.Forms 模板自动包括 iPad 支持通过配置**Info.plist** > 设备将设置为**通用**（这意味着支持 iPhone 和 iPad）。

若要提供愉快的启动体验，并确保所有设备上使用完整屏幕分辨率，应确保特定于 iPad 的启动屏幕（使用情节提要）提供。这可确保应用程序在 iPad mini、iPad 和 iPad Pro 设备上正确呈现。

在 iOS 9 之前的所有应用都占据整个屏幕在设备上，但现在可以执行一些 Ipad拆分屏幕多任务处理。这意味着您的应用程序可能需要多达只是精简的用户列在屏幕的屏幕或整个屏幕宽度的 50%的一侧。



分割屏幕功能意味着您应设计您的应用程序很好地配合只需少量的 320 像素宽，或高达 1366年像素宽。

Android 平板电脑

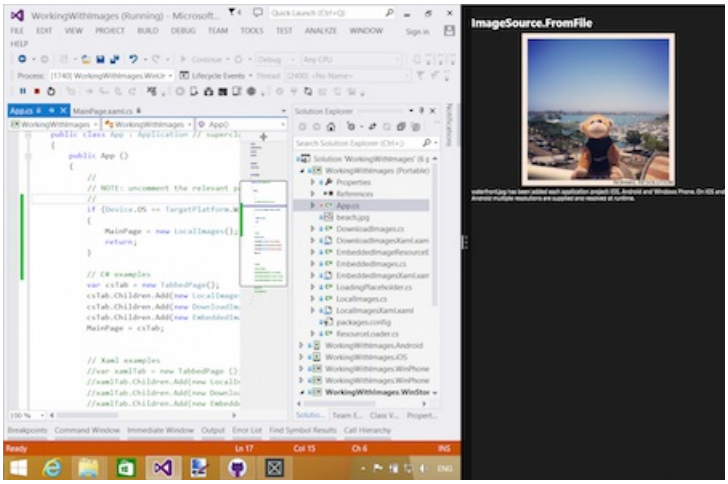
Android 生态系统具有大量的受支持的屏幕尺寸、从最大的平板电脑的小电话。Xamarin.Forms 可以支持所有屏幕大小，但作为与其他平台可能会想要调整用户界面的较大的设备。

支持许多不同的屏幕分辨率，您可以在不同的大小以优化用户体验中的本机映像资源。审阅Android 资源文档 (特别为不同的屏幕尺寸创建资源) 详细了解如何组织文件夹和 Android 应用程序中的文件名若要优化的映像资源包含在您的应用程序中的项目。

Windows 平板电脑和台式电脑

若要支持平板电脑和运行 Windows 的台式计算机，你将需要使用Windows UWP 支持，哪些生成通用 Windows 10 运行的应用程序。

Windows 平板电脑和台式计算机上运行的应用可调整到任意维度此外到正在运行全屏幕。



平板电脑和桌面优化

您可以调整 Xamarin.Forms 用户界面，具体取决于是否在手机或平板电脑/桌面设备正在使用。这意味着您可以优化用于大屏幕设备，例如平板电脑和台式计算机的用户体验。

Device.Idiom

可以使用 `Device` 类，以更改您的应用或用户界面的行为。使用 `Device.Idiom` 可以的枚举

```
if (Device.Idiom == TargetIdiom.Phone)
{
    HeroImage.Source = ImageSource.FromFile("hero.jpg");
} else {
    HeroImage.Source = ImageSource.FromFile("herotablet.jpg");
}
```

这种方法可以通过展开，以对各个页面布局进行重大更改，或者甚至还可以呈现在较大的屏幕完全不同页面。

利用 MasterDetailPage

`MasterDetailPage` 非常适合于较大的屏幕，尤其是在其中使用的 iPad 上 `UISplitViewController` 提供本机 iOS 体验。

审阅此 Xamarin 博客文章若要查看如何适应您的用户界面，以便电话使用一种布局和较大的屏幕可以使用另一个 (使用 `MasterDetailPage`)。

相关链接

- Xamarin 博客

- MyShoppe 示例

创建自定义布局

2018/8/6 • • Edit Online

Xamarin.Forms 定义了四个布局的类，这些 StackLayout、AbsoluteLayout、RelativeLayout 和网格中，并且每个不同的方式排列子项。但是，有时有必要来组织页面内容使用 Xamarin.Forms 不提供布局。本文说明如何编写一个自定义布局的类，并演示跨页上，水平排列子项，然后将包装对其他行的后续子级的显示方向区分 WrapLayout 类。

概述

在 Xamarin.Forms 中，所有布局类都派生自 `Layout<T>` 类和约束的泛型类型 `View` 及其派生类型。依次 `Layout<T>` 类派生自 `Layout` 类，该类提供用于定位和大小调整子元素的机制。

每个可视元素负责确定其自身的首选的大小，这被称为*请求*大小。`Page` `Layout` ，和 `Layout<View>` 派生的类型负责确定位置和大小及其子级或子级相对于自身。因此，布局涉及到父-子关系，其中父确定其子级的大小应该是什么，但将尝试容纳子级的请求的大小。

创建自定义布局需要全面了解 Xamarin.Forms 布局和失效周期。现在将讨论这些循环。

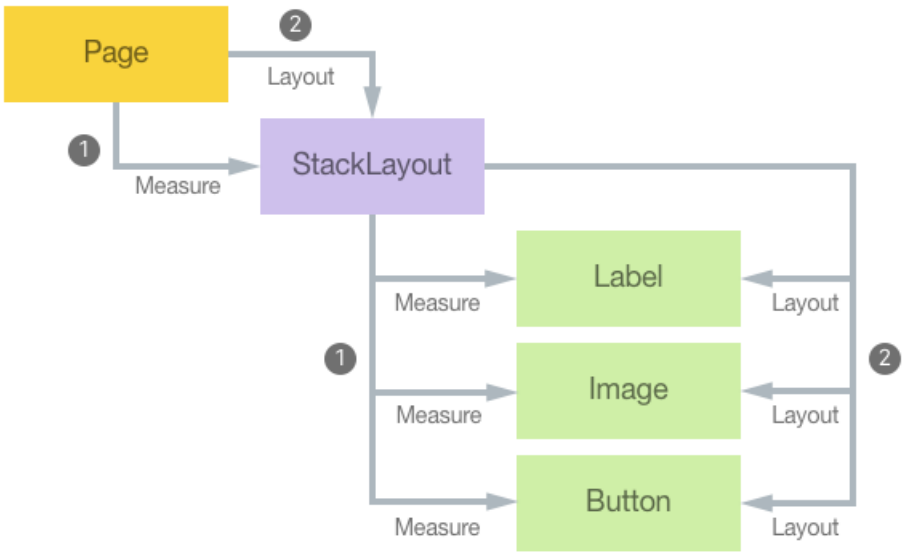
布局

在页上，使用的可视化树的顶部开始布局，并将经历的可视化树，以包含每个可视元素在页面上的所有分支。其他元素的父元素负责大小和位置相对于本身及其子级。

`VisualElement` 类定义 `Measure` 方法，用于测量布局操作的元素和一个 `Layout` 方法，它指定将在呈现元素的矩形区域。当应用程序启动并显示的第一页时，*布局循环*包含的第一个 `Measure` 调用，，然后 `Layout` 调用，启动上 `Page` 对象：

1. 在布局周期中，每个父元素负责调用 `Measure` 方法及其子级。
2. 测量子级后，每个父元素负责调用 `Layout` 方法及其子级。

此周期可确保每个可视元素的页上收到对调用 `Measure` 和 `Layout` 方法。该过程在下图中所示：



> **NOTE**
> 请注意，布局周期是否也可能发生的可视化树的子集上发生更改会影响布局。这包括添加或从例如，在集合中移除的项 `StackLayout` ，在更改 `IsVisible` 元素或元素的大小的更改的属性。

具有每个 Xamarin.Forms 类 `Content` 或 `Children` 属性具有可重写 `LayoutChildren` 方法。派生的自定义布局类 `Layout<View>` 必须重写此方法，并确保 `Measure` 并 `Layout` 方法调用所有元素子级，以提供所需的自定义布局。

此外，每个类的派生 `Layout` 或 `Layout<View>` 必须重写 `OnMeasure` 方法，它是在布局类确定它需要通过调用来的大小 `Measure` 其子级的方法。

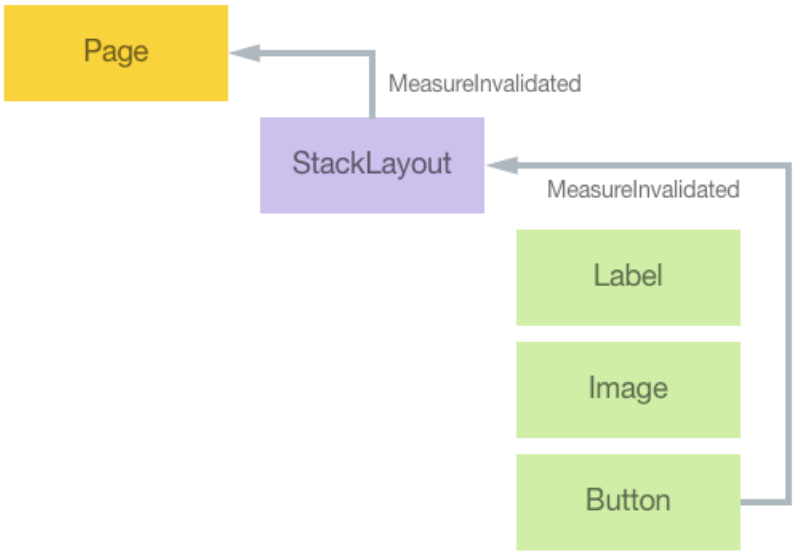
> **NOTE**
> 元素可确定其大小随着*约束*，这指示多少空间可用于该元素的父项中的元素。约束传递给 `Measure` 并 `OnMeasure` 方法可以介于 0 到 `Double.PositiveInfinity` 。元素是*约束*，或*完全约束*接收到的调用时，其 `Measure` 方法使用无限的参数的约束元素为特定大小。元素是*不受约束*，或*部分约束*接收到的调用时，其 `Measure` 具有至少一个参数等于方法 `Double.PositiveInfinity` – 可以是无限的约束视为，该值指示自动调整大小。

失效

失效是依据中的元素在页面上的更改将触发新的布局循环过程。当他们不再具有正确的大小或位置时，元素将被视为无效。例如，如果 `FontSize` 的属性 `Button` 的更改，`Button` 被认为是无效，因为它将不再具有正确的大小。重设大小 `Button` 然后可能通过页面的其余部分的布局中已更改的波纹效果。

元素使自身通过调用 `InvalidateMeasure` 方法中，通常该元素的属性更改时，可能会导致元素的新大小。此方法将触发 `MeasureInvalidated` 事件，该元素的父级的句柄来触发新的布局周期事件。

`Layout` 类设置的处理程序 `MeasureInvalidated` 事件在每个的子活动添加到其 `Content` 属性或 `Children` 集合，并分离该处理程序时删除子。因此，在具有子级的可视化树中的每个元素是向你发出警报时的一个子项更改大小。下图说明了如何在可视化树中的元素大小的更改可能会导致 ripple 上一级树的更改：



但是，`Layout` 类会尝试限制在页面的布局的孩子的大小的更改的影响。如果布局，则受约束的大小，然后子大小更改不会影响任何大于可视化树中父布局。但是，通常布局的大小的更改会影响布局如何排列子项。因此中的布局大小的任何更改将开始布局，布局循环和布局将接收对调用其 `OnMeasure` 并 `LayoutChildren` 方法。

`Layout` 类还定义 `InvalidateLayout` 方法具有相似的用途对 `InvalidateMeasure` 方法。`InvalidateLayout` 时进行的更改会影响布局定位和调整大小及其子级的方式，应调用方法。例如，`Layout` 类调用 `InvalidateLayout` 方法只要添加或从布局中删除子级。

`InvalidateLayout` 可以重写以实现缓存以最大程度减少反复调用 `Measure` 布局的子级的方法。重写 `InvalidateLayout` 方法将提供的子级添加到或从布局中删除时的通知。同样，`OnChildMeasureInvalidated` 可以重写方法的一个子级的布局大小更改时提供通知。有关这两种方法重写自定义布局应响应通过清除缓存。有关详细信息，请参阅计算和缓存数据。

创建自定义布局

创建自定义布局的过程如下所示：

1. 创建一个从 `Layout<View>` 类派生的类。有关详细信息，请参阅创建 WrapLayout。
2. [*可选*] 添加属性，由可绑定属性，应在布局类设置任何参数的支持。有关详细信息，请参阅添加由可绑定属性的属性支持。
3. 重写 `OnMeasure` 方法来调用 `Measure` 布局上的布局的所有子级，并返回请求的大小的方法。有关详细信息，请参阅重写 OnMeasure 方法。
4. 重写 `LayoutChildren` 方法来调用 `Layout` 对布局的所有子对象的方法。未能调用 `Layout` 在布局中的每个子方法会导致永远不会接收正确的大小或位置的子并因此子级将不会变得在页上可见。有关详细信息，请参阅重写 LayoutChildren 方法。

> **NOTE**
> 在枚举中的子级 `OnMeasure` 并 `LayoutChildren` 替代，请跳过任何子其 `IsVisible` 属性设置为 `false`。这将确保自定义布局不会保留为不可见子级的空间。

1. [*可选*] 重写 `InvalidateLayout` 方法添加到或从布局中删除子级时收到通知。有关详细信息，请参阅重写 InvalidateLayout 方法。
2. [*可选*] 重写 `OnChildMeasureInvalidated` 方法的一个子级的布局大小更改时收到通知。有关详细信息，请参阅重写 OnChildMeasureInvalidated 方法。

> **NOTE**
> 请注意，`OnMeasure` 不会调用重写，如果由其父级，而不是其子控制布局的大小。如果一个或两个约束是无限的或如果布局类具有非默认但是，将调用重写 `HorizontalOptions` 或 `VerticalOptions` 属性值。出于此原因，`LayoutChildren` 重写不能依赖于子大小期间获得 `OnMeasure` 方法调用。相反，`LayoutChildren` 必须调用 `Measure` 对布局的子对象，然后再调用方法 `Layout` 方法。子节点的大小或者，在中获得 `OnMeasure` 可以缓存重写以避免以后出现 `Measure` 中的调用 `LayoutChildren` 重写中，但布局类将需要知道何时需要重新获取大小。有关详细信息，请参阅计算和缓存布局数据。

然后可以通过将其添加到已使用布局类 `Page`，通过将子项添加到布局。有关详细信息，请参阅消耗 WrapLayout。

创建 WrapLayout

示例应用程序演示了方向区分 `WrapLayout` 类跨页上，水平排列子项，然后将包装对其他行的后续子级的显示。

`WrapLayout` 的类为每个孩子，名为分配的空间量相同*单元格大小*根据子级的最大大小。子级不是可与单元中定位单元格大小较小基于其 `HorizontalOptions` 并 `VerticalOptions` 属性值。

`WrapLayout` 类定义以下的代码示例所示：

```
public class WrapLayout : Layout<View>
{
  Dictionary<Size, LayoutData> layoutDataCache = new Dictionary<Size, LayoutData>();
  ...
}
```

计算和缓存布局数据

`LayoutData` 结构将有关子对象集合的数据存储在多个属性：

- `VisibleChildCount` – 在布局中可见的子级的个数。
- `CellSize` – 所有子级的布局大小调整的最大大小。
- `Rows` – 的行数。
- `Columns` – 的列数。

`layoutDataCache` 字段用于存储多个 `LayoutData` 值。当应用程序启动时，两个 `LayoutData` 对象将被缓存到 `layoutDataCache` 当前方向 – 一个用于约束参数的字典 `OnMeasure` 重写方法和一个用于 `width` 和 `height` 参数到 `LayoutChildren` 重写。到横向方向旋转设备时 `OnMeasure` 重写并 `LayoutChildren` 重写将再次调用，这将导致另一个的两个 `LayoutData` 字典中的缓存的对象。但是，当返回设备为纵向方向，没有进一步的计算需要，因为 `layoutDataCache` 已具有所需的数据。

下面的代码示例演示 `GetLayoutData` 方法，该计算的属性方法 `LayoutData` 结构化基于特定的大小：

```
LayoutData GetLayoutData(double width, double height)
{
  Size size = new Size(width, height);

  // Check if cached information is available.
  if (layoutDataCache.ContainsKey(size))
  {
    return layoutDataCache[size];
  }

  int visibleChildCount = 0;
  Size maxChildSize = new Size();
  int rows = 0;
  int columns = 0;
  LayoutData layoutData = new LayoutData();

  // Enumerate through all the children.
  foreach (View child in Children)
  {
    // Skip invisible children.
    if (!child.IsVisible)
      continue;

    // Count the visible children.
    visibleChildCount++;

    // Get the child's requested size.
    SizeRequest childSizeRequest = child.Measure(Double.PositiveInfinity, Double.PositiveInfinity);

    // Accumulate the maximum child size.
    maxChildSize.Width = Math.Max(maxChildSize.Width, childSizeRequest.Request.Width);
    maxChildSize.Height = Math.Max(maxChildSize.Height, childSizeRequest.Request.Height);
  }

  if (visibleChildCount != 0)
  {
    // Calculate the number of rows and columns.
    if (Double.IsPositiveInfinity(width))
    {
      columns = visibleChildCount;
      rows = 1;
    }
    else
    {
      columns = (int)((width + ColumnSpacing) / (maxChildSize.Width + ColumnSpacing));
      columns = Math.Max(1, columns);
      rows = (visibleChildCount + columns - 1) / columns;
    }
```

```
    // Now maximize the cell size based on the layout size.
    Size cellSize = new Size();

    if (Double.IsPositiveInfinity(width))
      cellSize.Width = maxChildSize.Width;
    else
      cellSize.Width = (width - ColumnSpacing * (columns - 1)) / columns;

    if (Double.IsPositiveInfinity(height))
      cellSize.Height = maxChildSize.Height;
    else
      cellSize.Height = (height - RowSpacing * (rows - 1)) / rows;

    layoutData = new LayoutData(visibleChildCount, cellSize, rows, columns);
  }

  layoutDataCache.Add(size, layoutData);
  return layoutData;
}
```

`GetLayoutData` 方法执行以下操作：

- 它确定是否计算 `LayoutData` 值是否已在缓存并返回它是否可用。
- 否则，它枚举所有子级，调用 `Measure` 方法上使用无限宽度和高度，每个子级并确定最大子的大小。
- 前提是没有至少一个可见子级，它计算的行和所需，列数，然后计算的维度所基于的子级的单元格大小 `WrapLayout`。请注意，单元格大小通常是稍宽于最大子大小，但是，这也可能是较小如果 `WrapLayout` 不够宽最宽子或 tall 足够的最高的子级。
- 它将存储新 `LayoutData` 缓存中的值。

将属性添加受可绑定属性

`WrapLayout` 类定义 `ColumnSpacing` 和 `RowSpacing` 属性，其值用于分隔的行和列在布局中，支持，其中受可绑定属性。可绑定属性下面的代码示例所示：

```
public static readonly BindableProperty ColumnSpacingProperty = BindableProperty.Create(
  "ColumnSpacing",
  typeof(double),
  typeof(WrapLayout),
  5.0,
  propertyChanged: (bindable, oldvalue, newvalue) =>
  {
    ((WrapLayout)bindable).InvalidateLayout();
  });

public static readonly BindableProperty RowSpacingProperty = BindableProperty.Create(
  "RowSpacing",
  typeof(double),
  typeof(WrapLayout),
  5.0,
  propertyChanged: (bindable, oldvalue, newvalue) =>
  {
    ((WrapLayout)bindable).InvalidateLayout();
  });
```

每个可绑定属性的属性更改处理程序调用 `InvalidateLayout` 方法重写以触发新的布局传递 `WrapLayout`。有关详细信息，请参阅重写 InvalidateLayout 方法并重写 OnChildMeasureInvalidated 方法。

重写 OnMeasure 方法

`OnMeasure` 重写下面的代码示例所示：

```
protected override SizeRequest OnMeasure(double widthConstraint, double heightConstraint)
{
  LayoutData layoutData = GetLayoutData(widthConstraint, heightConstraint);
  if (layoutData.VisibleChildCount == 0)
  {
    return new SizeRequest();
  }

  Size totalSize = new Size(layoutData.CellSize.Width * layoutData.Columns + ColumnSpacing *
(layoutData.Columns - 1),
                layoutData.CellSize.Height * layoutData.Rows + RowSpacing * (layoutData.Rows - 1));
  return new SizeRequest(totalSize);
}
```

重写调用 `GetLayoutData` 方法和构造 `SizeRequest` 对象从返回的数据，同时还考虑 `RowSpacing` 和 `ColumnSpacing` 属性值。有关详细信息 `GetLayoutData` 方法，请参阅计算和缓存数据。

> **IMPORTANT**
>
> `Measure` 并 `OnMeasure` 方法应永远不会通过返回请求无限维度 `SizeRequest` 属性设置为值 `Double.PositiveInfinity` . 但是，在至少一个约束的参数 `OnMeasure` 可以是 `Double.PositiveInfinity` 。

重写 LayoutChildren 方法

`LayoutChildren` 重写下面的代码示例所示：

```
protected override void LayoutChildren(double x, double y, double width, double height)
{
  LayoutData layoutData = GetLayoutData(width, height);

  if (layoutData.VisibleChildCount == 0)
  {
    return;
  }

  double xChild = x;
  double yChild = y;
  int row = 0;
  int column = 0;

  foreach (View child in Children)
  {
    if (!child.IsVisible)
    {
      continue;
    }

    LayoutChildIntoBoundingRegion(child, new Rectangle(new Point(xChild, yChild), layoutData.CellSize));
    if (++column == layoutData.Columns)
    {
      column = 0;
      row++;
      xChild = x;
      yChild += RowSpacing + layoutData.CellSize.Height;
    }
    else
    {
      xChild += ColumnSpacing + layoutData.CellSize.Width;
    }
  }
}
```

重写开始通过调用 `GetLayoutData` 方法，然后枚举所有子节点大小和位置在每个孩子的单元格内。这通过调用来实现 `LayoutChildIntoBoundingRegion` 方法，用于定位子基于在矩形内的其 `HorizontalOptions` 并 `VerticalOptions` 属性值。这相当于调用的子 `Layout` 方法。

> **NOTE**
> 请注意，矩形传递给 `LayoutChildIntoBoundingRegion` 方法包含子级可以驻留在其中的整个区域。

有关详细信息 `GetLayoutData` 方法，请参阅计算和缓存数据。

重写 InvalidateLayout 方法

`InvalidateLayout` 重写调用时添加或删除从布局中，或一个子级的 `WrapLayout` 属性更改值，如下面的代码示例中所示：

```
protected override void InvalidateLayout()
{
  base.InvalidateLayout();
  layoutInfoCache.Clear();
}
```

重写会使布局失效并放弃所有的缓存的布局信息。

> **NOTE**
> 若要停止 `Layout` 类调用 `InvalidateLayout` 方法添加或从布局中删除子级时重写 `ShouldInvalidateOnChildAdded` 并 `ShouldInvalidateOnChildRemoved` 方法，并返回 `false`。添加或删除子级时，布局类然后可以实现一个自定义进程。

重写 OnChildMeasureInvalidated 方法

`OnChildMeasureInvalidated` 重写调用时的一个布局的子级更改大小，并在下面的代码示例所示：

```
protected override void OnChildMeasureInvalidated()
{
  base.OnChildMeasureInvalidated();
  layoutInfoCache.Clear();
}
```

重写使无效子布局，并放弃所有的缓存的布局信息。

使用 WrapLayout

`WrapLayout` 类可以通过将它放置在消耗 `Page` 派生类型，如以下 XAML 代码示例所示：

```
<ContentPage ... xmlns:local="clr-namespace:ImageWrapLayout">
    <ScrollView Margin="0,20,0,20">
        <local:WrapLayout x:Name="wrapLayout" />
    </ScrollView>
</ContentPage>
```

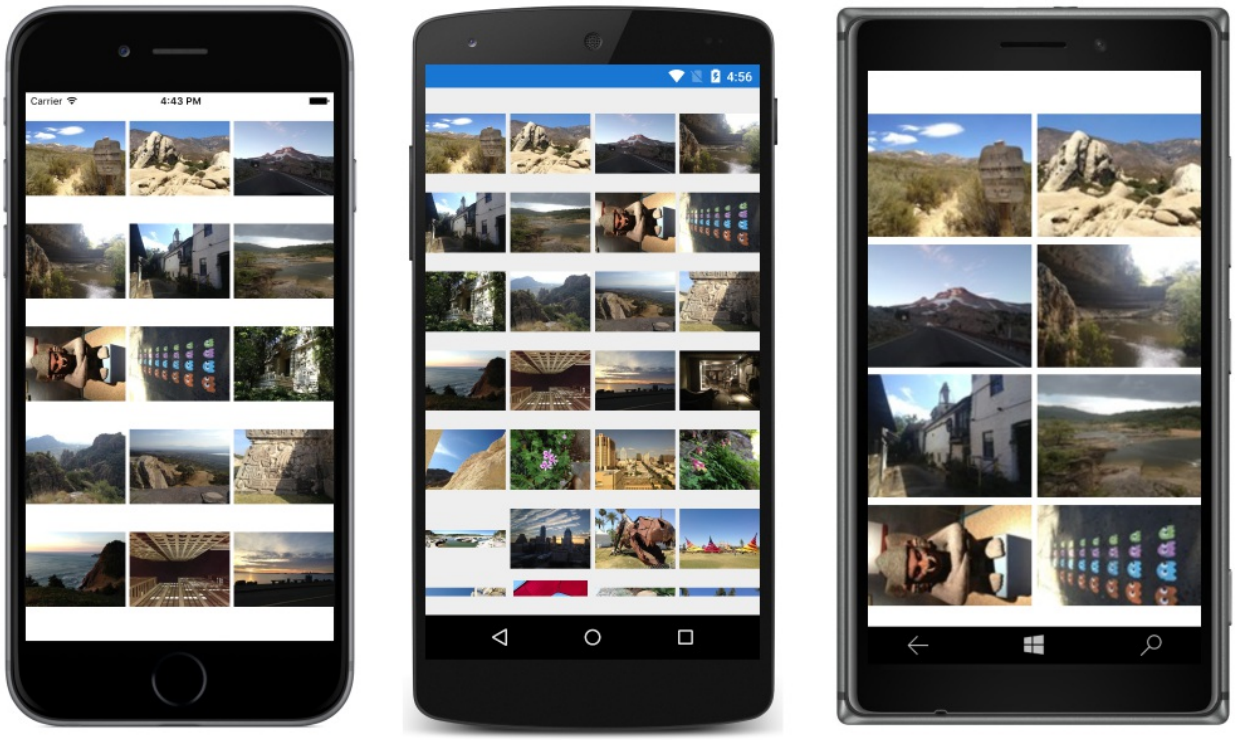
等效的 C# 代码如下所示：

```
public class ImageWrapLayoutPageCS : ContentPage
{
  WrapLayout wrapLayout;

  public ImageWrapLayoutPageCS()
  {
    wrapLayout = new WrapLayout();

    Content = new ScrollView
    {
      Margin = new Thickness(0, 20, 0, 20),
      Content = wrapLayout
    };
  }
  ...
}
```

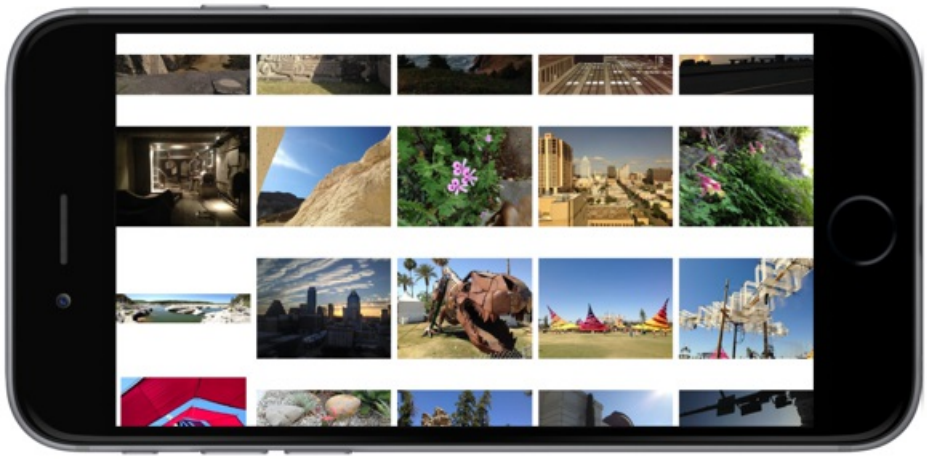
然后将子项添加到 `WrapLayout` 所需的方式。下面的代码示例演示 `Image` 元素添加到 `WrapLayout`：

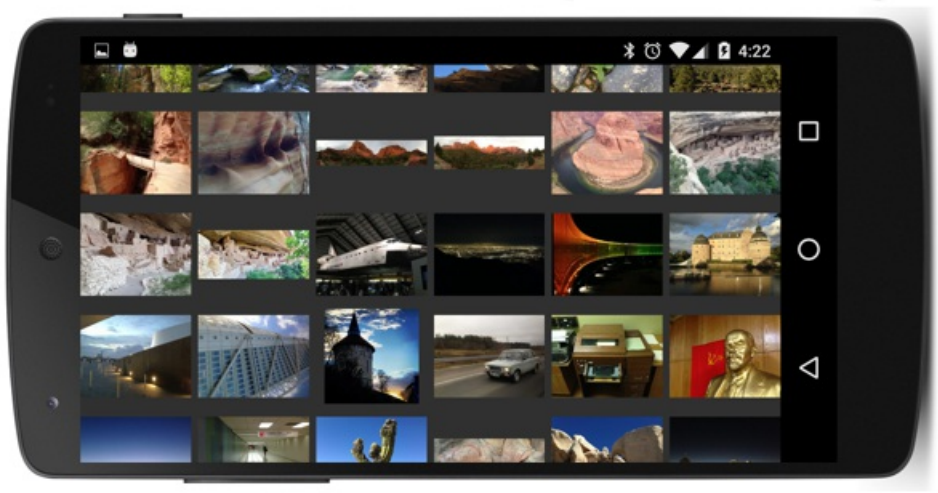
```
protected override async void OnAppearing()
{
  base.OnAppearing();

  var images = await GetImageListAsync();
  foreach (var photo in images.Photos)
  {
    var image = new Image
    {
      Source = ImageSource.FromUri(new Uri(photo + string.Format("?width={0}&height={0}&mode=max",
Device.RuntimePlatform == Device.UWP ? 120 : 240)))
    };
    wrapLayout.Children.Add(image);
  }
}

async Task<ImageList> GetImageListAsync()
{
  var requestUri = "https://docs.xamarin.com/demo/stock.json";
  using (var client = new HttpClient())
  {
    var result = await client.GetStringAsync(requestUri);
    return JsonConvert.DeserializeObject<ImageList>(result);
  }
}
```

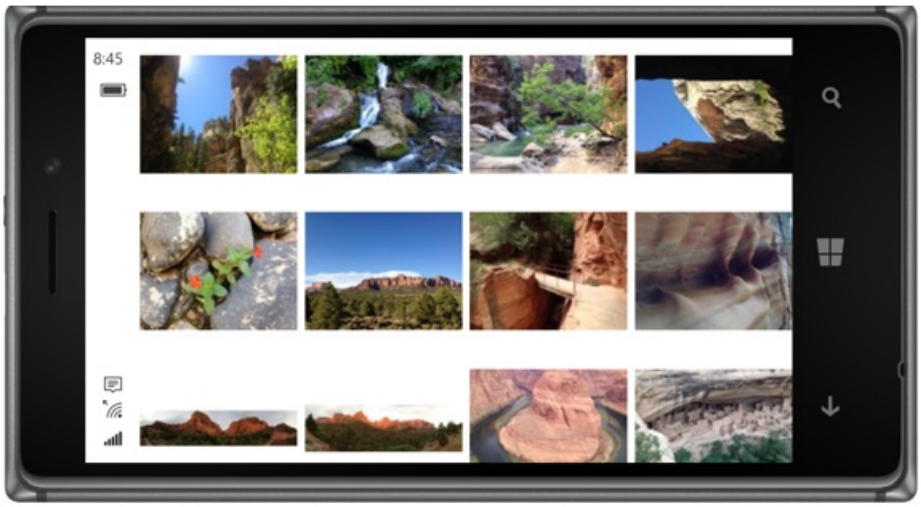
当页面包含 `WrapLayout` 出现，示例应用程序以异步方式访问包含一系列照片的远程 JSON 文件，创建 `Image` 元素为每个照片，并将其添加到 `WrapLayout`。这会导致下面的屏幕截图中所示的外观：



下面的屏幕截图演示 `WrapLayout` 已旋转为横向后：







每个行中的列数取决于照片大小、屏幕宽度和每个与设备无关单位的像素数。`Image` 元素以异步方式加载照片，并因此 `WrapLayout` 类将接收到频繁调用其 `LayoutChildren` 方法作为每个 `Image` 在元素收到基于加载照片的新大小。

总结

这篇文章介绍了如何编写一个自定义布局的类，并演示了方向区分 `WrapLayout` 类跨页上，水平排列子项，然后将包装对其他行的后续子级的显示。

相关链接

- WrapLayout（示例）
- 自定义布局
- 在 Xamarin.Forms 中创建自定义布局（视频）
- 布局
- 布局
- VisualElement

布局压缩

2018/7/13 • • Edit Online

布局压缩从可视化树中删除指定的布局，以试图提升页面呈现性能。本文介绍如何启用布局压缩和可以带来的好处。

概述

Xamarin.Forms 执行使用递归方法调用的两个序列的布局：

- 在页上，使用的可视化树的顶部开始布局，并将经历的可视化树，以包含每个可视元素在页面上的所有分支。其他元素的父元素负责大小和位置相对于本身及其子级。
- 失效是依据中的元素在页面上的更改将触发新的布局循环过程。当他们不再具有正确的大小或位置时，元素将被视为无效。当它的一个子级发生更改时大小，具有子级的可视化树中的每个元素是向你发出警报。因此，在可视化树中的元素大小的更改可能导致 ripple 上一级树的更改。

有关 Xamarin.Forms 布局的执行方式的详细信息，请参阅创建自定义布局。

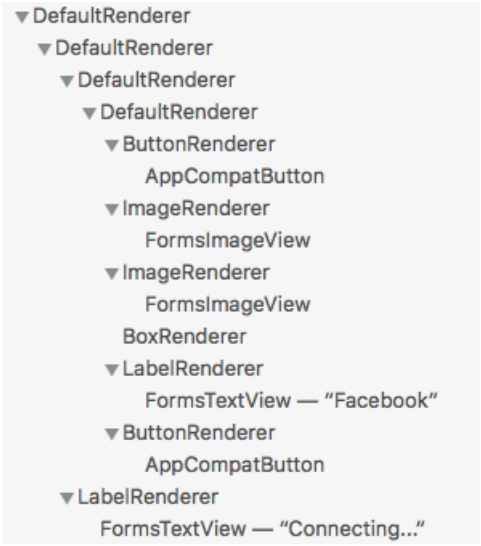
布局过程的结果是层次结构的本机控件。但是，此层次结构包含其他容器呈现器和包装对于平台呈现器，进一步以下嵌套的视图层次结构。嵌套级别越高，Xamarin.Forms 具有执行，以显示页的工作的值就越大。对于复杂的布局的视图层次结构可以是嵌套的深度和广泛，具有多个级别。

有关示例，请登录到 Facebook 的示例应用程序中的以下按钮：

此按钮指定为以下 XAML 视图层次结构的自定义控件：

```
<ContentView ...>
    <StackLayout>
        <StackLayout ...>
            <AbsoluteLayout ...>
                <Button ... />
                <Image ... />
                <Image ... />
                <BoxView ... />
                <Label ... />
                <Button ... />
            </AbsoluteLayout>
        </StackLayout>
        <Label ... />
    </StackLayout>
</ContentView>
```

可以检查生成的嵌套的视图层次结构，与Xamarin Inspector。在 Android 上，嵌套的视图层次结构包含 17 视图：



布局压缩，这是适用于 iOS 和 Android 平台上的 Xamarin.Forms 应用程序，旨在来平展嵌套方法从可以提高页面呈现性能的可视化树中删除指定的布局的视图。传送的性能优势因页面、正在使用的操作系统的版本和在其运行应用程序的设备的复杂性而异。不过，在旧设备上实现的性能提升最大。

> **NOTE**
> 虽然这篇文章重点介绍在 Android 上应用布局压缩的结果，但这同样适用于 iOS。

布局压缩

在 XAML 中，可以通过设置启用布局压缩 `CompressedLayout.IsHeadless` 附加属性设置为 `true` 布局类上：

```
<StackLayout CompressedLayout.IsHeadless="true">
  ...
</StackLayout>
```

或者，它可以启用在 C# 中的第一个参数为指定的布局实例 `CompressedLayout.SetIsHeadless` 方法：

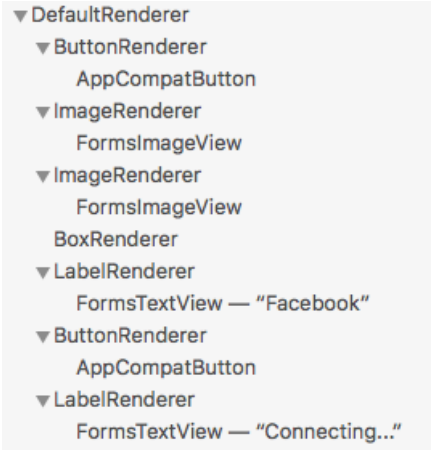
```
CompressedLayout.SetIsHeadless(stackLayout, true);
```

> **IMPORTANT**
> 布局压缩从可视化树中删除一个布局，因为它不适合布局提供的可视外观，或获取触控输入。因此，该设置的布局 `VisualElement` 属性（如 `BackgroundColor`、`IsVisible`、`Rotation`、`Scale`、`TranslationX` 并 `TranslationY` 或的接受手势，不适合布局压缩。但是，启用布局压缩布局的设置可视外观的属性，或接受手势，不会导致生成或运行时错误。相反，将应用布局压缩和可视外观属性和手势识别将以静默方式失败。

对于 Facebook 按钮，可以在三个布局类上启用布局压缩：

```
<StackLayout CompressedLayout.IsHeadless="true">
    <StackLayout CompressedLayout.IsHeadless="true" ...>
        <AbsoluteLayout CompressedLayout.IsHeadless="true" ...>
            ...
        </AbsoluteLayout>
    </StackLayout>
    ...
</StackLayout>
```

在 Android 上，这会导致 14 视图的嵌套的视图层次结构：

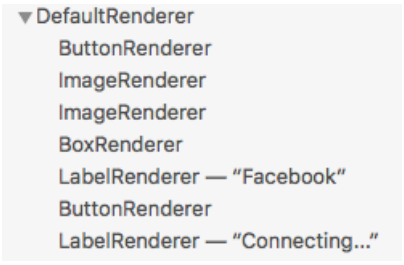


与原始的嵌套的视图层次结构的 17 视图相比，这表示 17%的次数减少。虽然这种减少可能出现不重要，减少对整个页面的视图可以更重要。

快速呈现器

快速呈现器减少的通货膨胀和呈现成本 Android 上 Xamarin.Forms 控件通过平展生成的本机视图层次结构。通过创建更少的对象，这反过来会导致不太复杂的可视化树和内存使用率，这进一步提高性能。有关快速呈现器的详细信息，请参阅快速呈现器。

示例应用程序中的 Facebook 按钮，结合使用布局压缩和快速呈现器生成 8 视图的嵌套的视图层次的结构：



与原始的嵌套的视图层次结构的 17 视图相比，这表示减少了 52%。

示例应用程序包含从实际的应用程序中提取的页。如果没有布局压缩和快速呈现器，页面将生成 130 视图在 Android 上的嵌套的视图层次结构。启用快速呈现器和适当的布局类上的布局压缩到 70 视图，降低了 46%的减少的嵌套的视图层次结构。

总结

布局压缩从可视化树中删除指定的布局，以试图提升页面呈现性能。这带来的性能优势因页面复杂性、要使用的操作系统版本以及运行应用的设备而异。不过，在旧设备上实现的性能提升最大。

相关链接

- 创建自定义布局
- 快速呈现器
- LayoutCompression（示例）

Xamarin.Forms ListView

2018/10/19 • • Edit Online

ListView 是用于呈现数据，尤其是需要滚动的长列表的列表视图。本指南将演示如何使用 ListView:

1. **数据源** –填充数据，无论数据绑定 ListView。
2. **单元格外观** –自定义内置的单元格的外观或创建你自己的自定义单元格。
3. **列表外观** –自定义 ListView 的外观。设置页眉和页脚、启用组和更改的行的高度。
4. **交互性** –处理分流点，并选择，实现下拉刷新，并添加的上下文操作。
5. **性能** –避免性能问题。

用例

请确保 ListView 是右侧控件为您的需要。在任何情况下，其中显示数据的可滚动列表，可以使用 ListView。Listview 支持上下文操作和数据绑定。

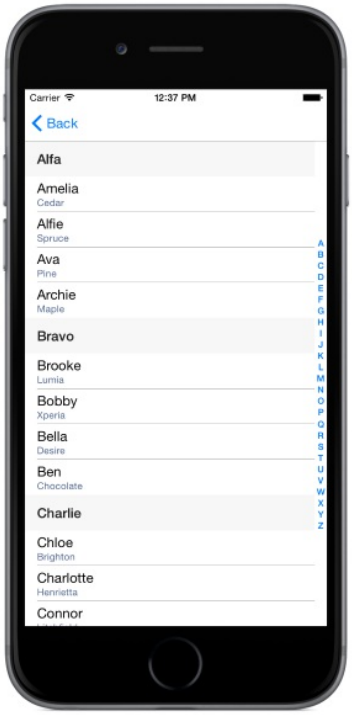
不要将 ListView 与相混淆TableView。TableView 控制会是更好的选择的选项或数据的非绑定到列表。例如，iOS 设置应用中，有一组主要是预定义的选项，是更好地适合使用 ListView 比 TableView。

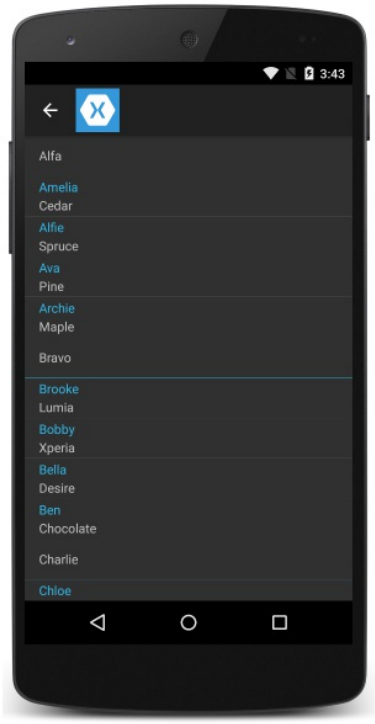
请注意，ListView 是最佳也适用于同构数据–，即所有数据应都为同一类型。这是因为只有一种类型的单元格可用于在列表中的每一行。TableViews 可以支持多个单元格类型，因此它们是更好的选择时需要使用多种视图。

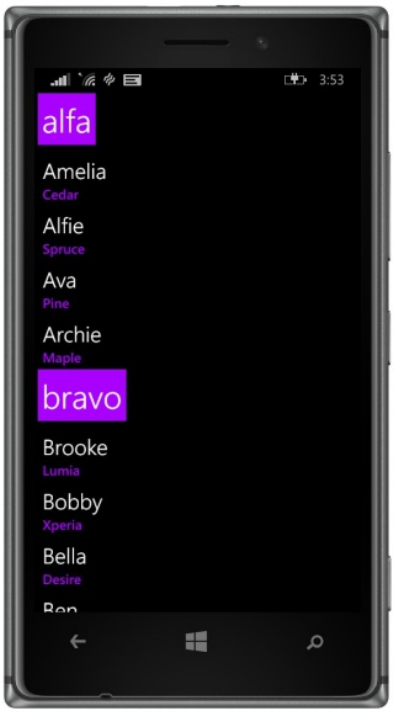
组件数

ListView 有多个组件可用于执行每个平台的本机功能。下面描述了每个组件:

- 页眉和页脚 –与列表的数据分开的开头和结尾的列表，在显示的文本或视图。页眉和页脚可以绑定到数据源独立地从 ListView 的数据源。
- 组 – ListView 中的数据可以分组以便更易于导航。组通常包括数据绑定:







- 单元格 – ListView 中的数据也会出现在单元格。每个单元格对应于一行数据。有内置的单元格可供选择，

也可以定义自己的自定义单元格。内置和自定义单元格可以是在 XAML 或代码中使用或定义。

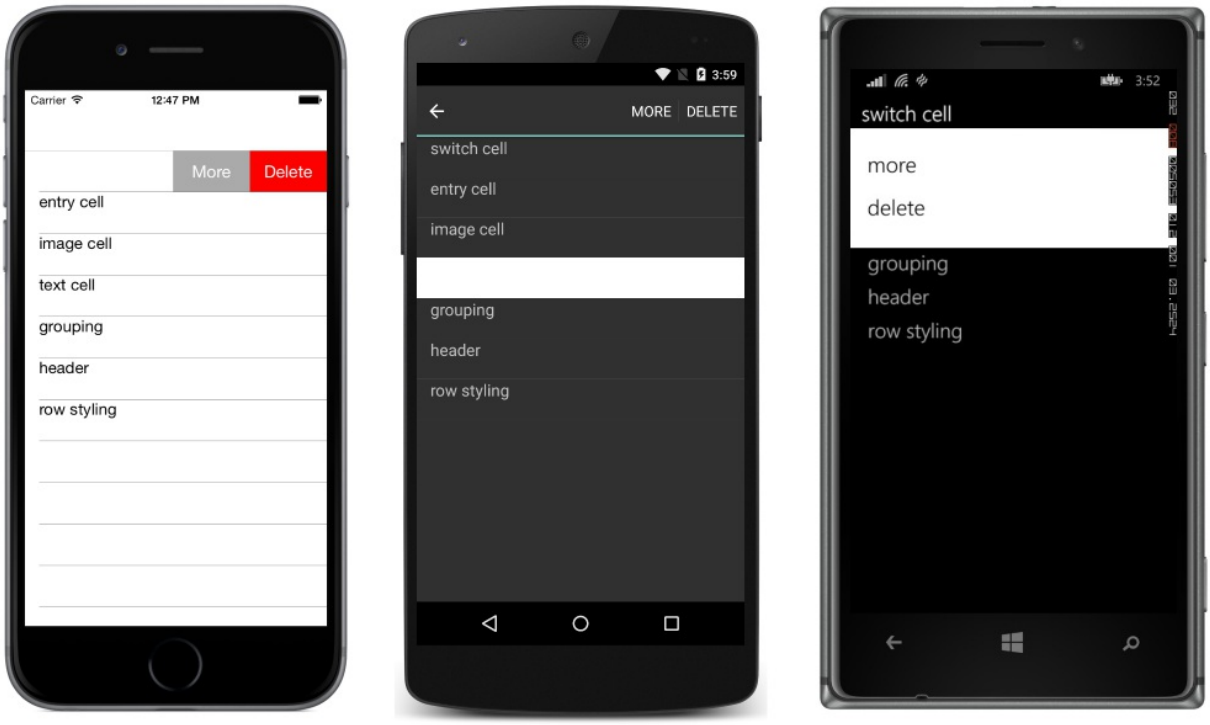
- **内置** –内置的单元格，尤其是 TextCell 和 ImageCell，可以有很高的性能，因为它们对应于每个平台上的本机控件。
  - **TextCell** –显示文本，并且可选择带有详细信息文本的字符串。详细信息的文本呈现为与强调文字颜色较小的字体中的第二个行。
  - **ImageCell** –显示文本与图像。将显示为与左侧图像 TextCell。
- **自定义单元格** –时需要提供复杂的数据，自定义单元格是很好。例如，自定义视图可用来提供包括专辑和艺术家的歌曲的列表：

若要了解有关自定义 ListView 中的单元格的详细信息，请参阅自定义 ListView 单元格的外观。

功能

ListView 还支持多种交互样式，包括：

- **下拉刷新** – ListView 每个平台上支持下拉刷新。
- **上下文操作** – ListView 支持在列表中的单个项上采取措施。例如，您可以在 iOS 上，实现轻扫操作或长按在 Android 上的操作。
- **所选内容** –可以侦听的选择和取消点击某行时采取措施。



若要了解有关 ListView 的交互功能的详细信息，请参阅操作和与 ListView 的交互操作。

相关链接

- 使用与 ListView（示例）
- 两个双向绑定（示例）
- 构建在单元格（示例）
- 自定义单元格（示例）
- 分组（示例）
- 自定义呈现器视图（示例）
- ListView 交互性（示例）

ListView 数据源

2018/10/26 • • Edit Online

一个 `ListView` 用于显示数据的列表。我们将了解有关填充 ListView 与数据和如何我们可以绑定到选定的项。

- **设置 ItemsSource** –使用简单列表或数组。
- **数据绑定** –模型和 ListView 之间建立关系。绑定适合于 MVVM 模式。

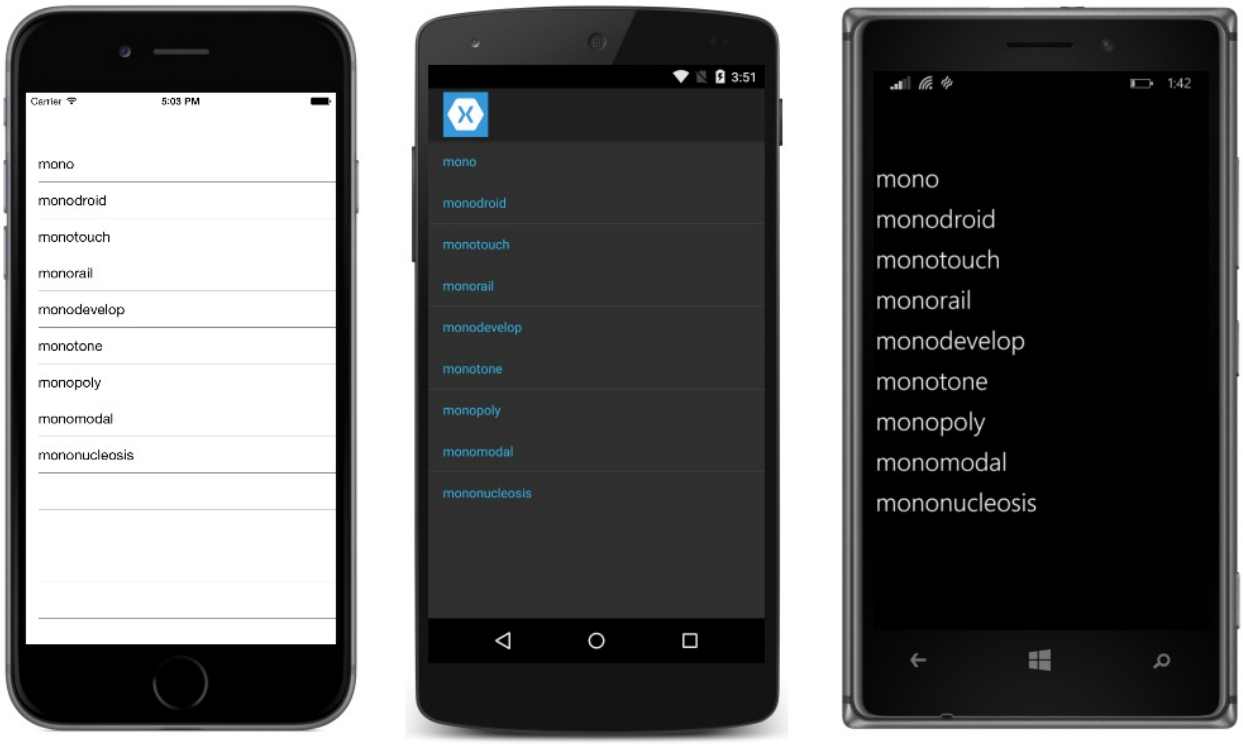
ItemsSource

一个 `ListView` 与使用数据填充 `ItemsSource` 属性，它可以接受任何集合实现 `IEnumerable`。最简单的方法来填充 `ListView`，需使用一个字符串数组：

```
<ListView>
      <ListView.ItemsSource>
          <x:Array Type="{x:Type x:String}">
            <x:String>mono</x:String>
            <x:String>monodroid</x:String>
            <x:String>monotouch</x:String>
            <x:String>monorail</x:String>
            <x:String>monodevelop</x:String>
            <x:String>monotone</x:String>
            <x:String>monopoly</x:String>
            <x:String>monomodal</x:String>
            <x:String>mononucleosis</x:String>
          </x:Array>
      </ListView.ItemsSource>
</ListView>
```

等效的 C# 代码是：

```
var listView = new ListView();
listView.ItemsSource = new string[]
{
  "mono",
  "monodroid",
  "monotouch",
  "monorail",
  "monodevelop",
  "monotone",
  "monopoly",
  "monomodal",
  "mononucleosis"
};

//monochrome will not appear in the list because it was added
//after the list was populated.
listView.ItemsSource.Add("monochrome");
```



上述方法将填充 `ListView` 与字符串的列表。默认情况下 `ListView` 将调用 `ToString` 并显示在结果 `TextCell` 每个行。若要自定义数据的显示方式，请参阅单元格的外观。

因为 `ItemsSource` 已发送到一个数组中的内容不会更新为基础的列表或数组更改。如果你想要自动更新，因为添加、删除和更改的基础列表中的项时 ListView，您将需要使用 `ObservableCollection` 。 `ObservableCollection` 在中定义 `System.Collections.ObjectModel` 和类似于 `List` ，只不过它可以通知 `ListView` 的任何更改：

```
ObservableCollection<Employees> employeeList = new ObservableCollection<Employess>();
listView.ItemsSource = employeeList;

//Mr. Mono will be added to the ListView because it uses an ObservableCollection
employeeList.Add(new Employee(){ DisplayName="Mr. Mono"});
```

数据绑定

数据绑定是将用户界面对象的属性绑定到某些 CLR 对象，如在 ViewModel 中的类的属性"粘合"。数据绑定很有用，因为它简化了用户界面的开发，通过替换大量繁琐样板代码。

数据绑定的工作原理是在其绑定的值更改时保持对象的同步。而无需编写事件处理程序，每次控件的值更改时，可以建立绑定，并在 ViewModel 中启用绑定。

数据绑定的详细信息，请参阅数据绑定基础知识这四个是一部分Xamarin.Forms XAML 基础知识文章系列。

单元格绑定

单元格（和单元格的子项）的属性可以绑定到对象中的属性 `ItemsSource` 。例如，ListView 可以用于显示员工的列表。

Employee 类：

```
public class Employee{
    public string DisplayName {get; set;}
}
```

`ObservableCollection<Employee>` 创建并设置为 `ListView` 的 `ItemsSource` ：

```
ObservableCollection<Employee> employees = new ObservableCollection<Employee>();
public EmployeeListPage()
{
  //defined in XAML to follow
  EmployeeView.ItemsSource = employees;
  ...
}
```

列表中填充了数据：

```
public EmployeeListPage()
{
  ...
  employees.Add(new Employee{ DisplayName="Rob Finnerty"});
  employees.Add(new Employee{ DisplayName="Bill Wrestler"});
  employees.Add(new Employee{ DisplayName="Dr. Geri-Beth Hooper"});
  employees.Add(new Employee{ DisplayName="Dr. Keith Joyce-Purdy"});
  employees.Add(new Employee{ DisplayName="Sheri Spruce"});
  employees.Add(new Employee{ DisplayName="Burt Indybrick"});
}
```

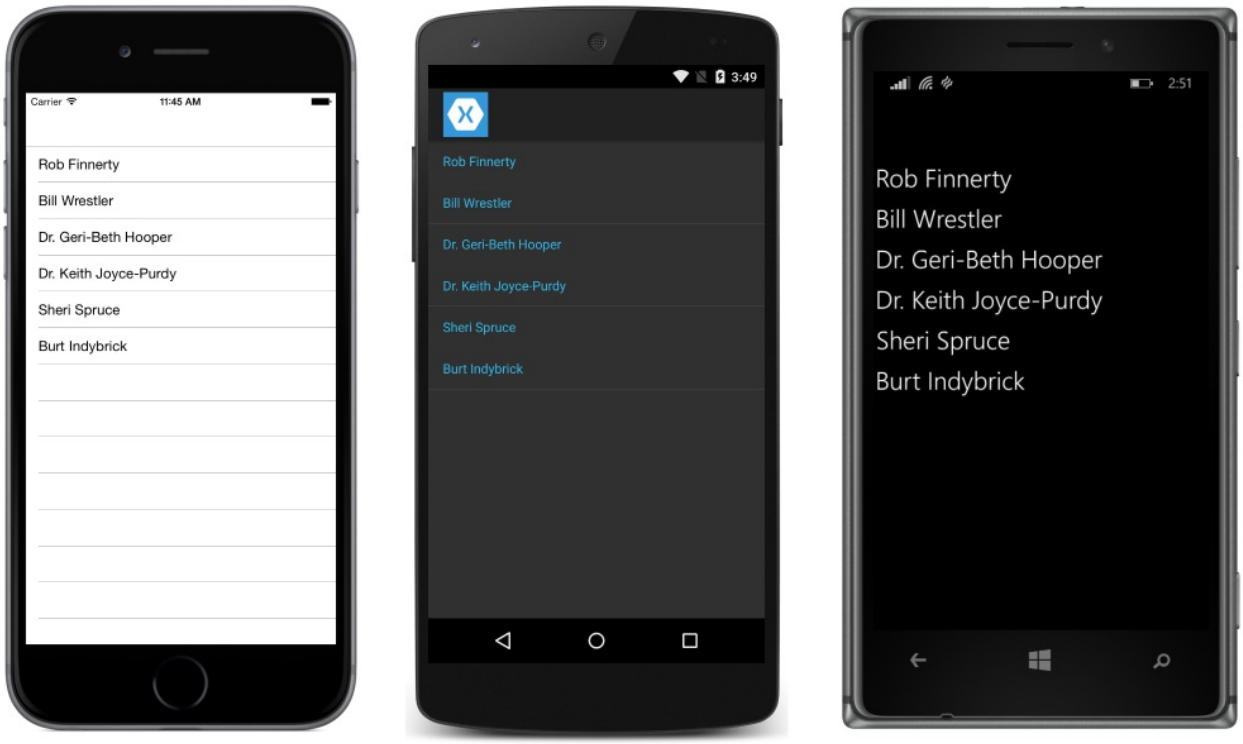
以下代码片段演示 `ListView` 绑定到的员工列表：

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:constants="clr-namespace:XamarinFormsSample;assembly=XamarinFormsXamlSample"
x:Class="XamarinFormsXamlSample.Views.EmployeeListPage"
Title="Employee List">
  <ListView x:Name="EmployeeView">
    <ListView.ItemTemplate>
      <DataTemplate>
        <TextCell Text="{Binding DisplayName}" />
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
</ContentPage>
```

尽管它可能已绑定在 XAML 中，请注意的绑定是在代码中为简单起见，安装程序。

XAML 的上一位定义 `ContentPage`，其中包含 `ListView`。数据源 `ListView` 通过设置 `ItemsSource` 属性。中的每一行的布局 `ItemsSource` 中定义 `ListView.ItemTemplate` 元素。

下面是结果：



绑定 SelectedItem

通常你会想要绑定到的所选的项 `ListView`，而不是不是使用事件处理程序的更改进行响应。若要执行此操作在 XAML 中，将绑定 `SelectedItem` 属性：

```
<ListView x:Name="listView"
 SelectedItem="{Binding Source={x:Reference SomeLabel},
 Path=Text}">
 …
</ListView>
```

假设 `listView` 的 `ItemsSource` 是一个字符串，列表 `SomeLabel` 将具有其 text 属性绑定到 `SelectedItem`。

相关链接

- 两个双向绑定（示例）

自定义 ListView 单元格的外观

2018/7/13 • • Edit Online

ListView 显示可滚动列表，可以使用自定义 `ViewCell` s。 `ViewCells` 可用于显示文本和图像、指示 true/false 状态和接收用户输入。

有两种方法可以获取所需从 ListView 单元格的外观：

- **自定义内置的单元格** –更容易实现，但要牺牲可自定义性更好的性能。
- **创建自定义单元格** –更多控制的最终结果，但有可能出现性能问题，如果未正确地实现。

内置的单元格

Xamarin.Forms 附带内置的适用于许多简单应用程序的单元格：

- **TextCell** –用于显示文本
- **ImageCell** –用于显示文本与图像。

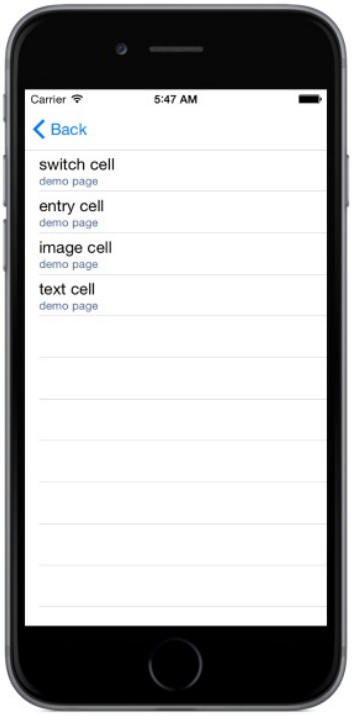
两个其他单元格 `SwitchCell` 并 `EntryCell` 可供使用，但它们不常用于 `ListView` 。请参阅 `TableView` 详细了解这些单元格。

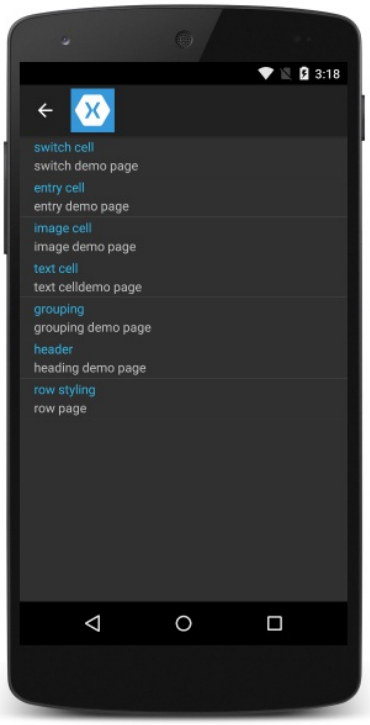
TextCell

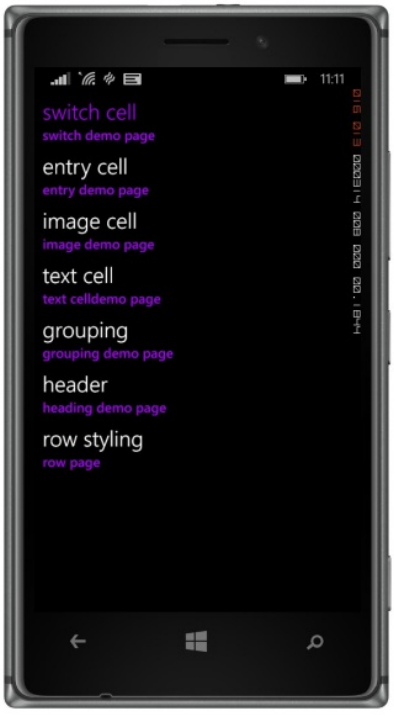
`TextCell` 是用于显示文本，并且根据需要使用第二行详细信息的文本作为一个单元格。

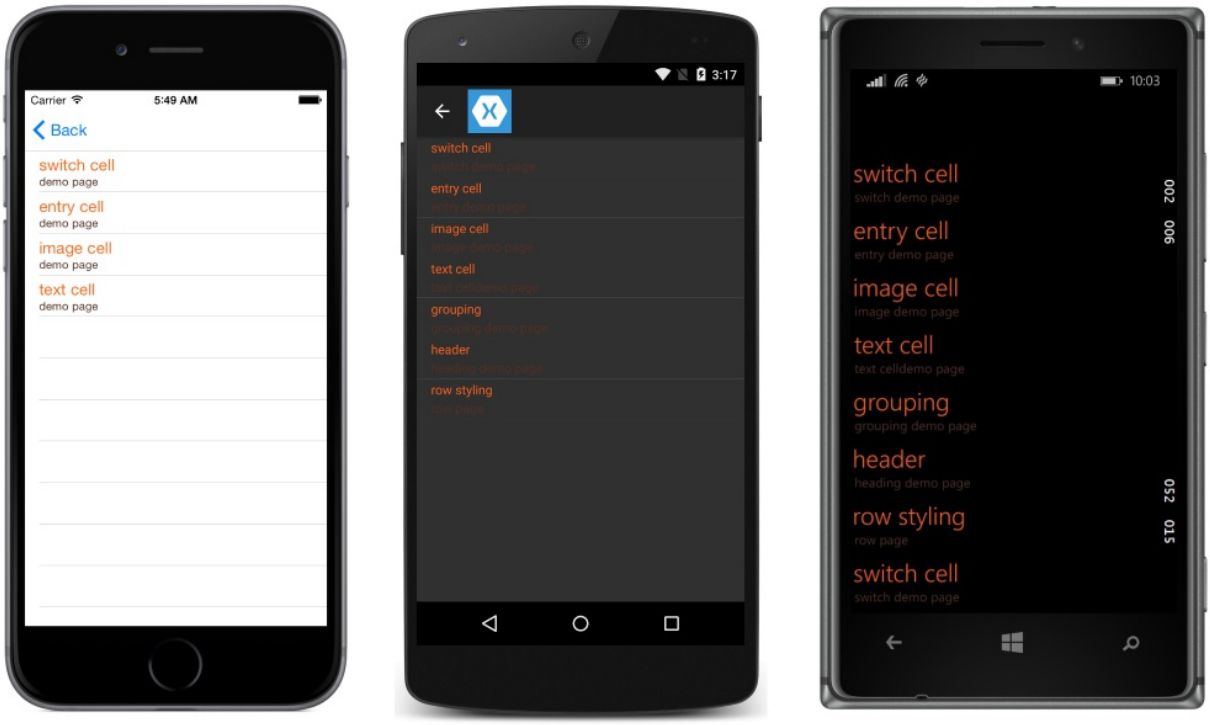
TextCells 呈现为本机控件在运行时，因此性能非常好与自定义比较 `ViewCell` 。TextCells 可自定义，使你能够设置：

- `Text` – 在第一行，用大字体显示文本。
- `Detail` – 较小的字体中的第一行下面所示的文本。
- `TextColor` – 文本的颜色。
- `DetailColor` – 详细信息文本的颜色







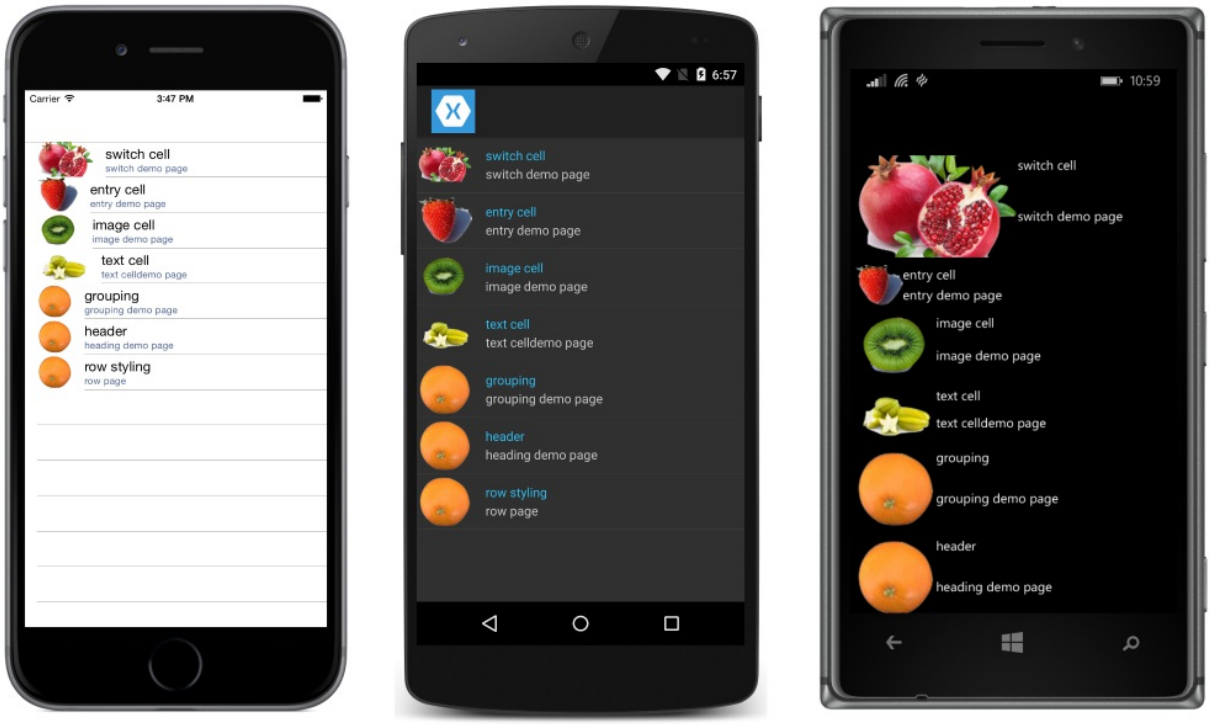


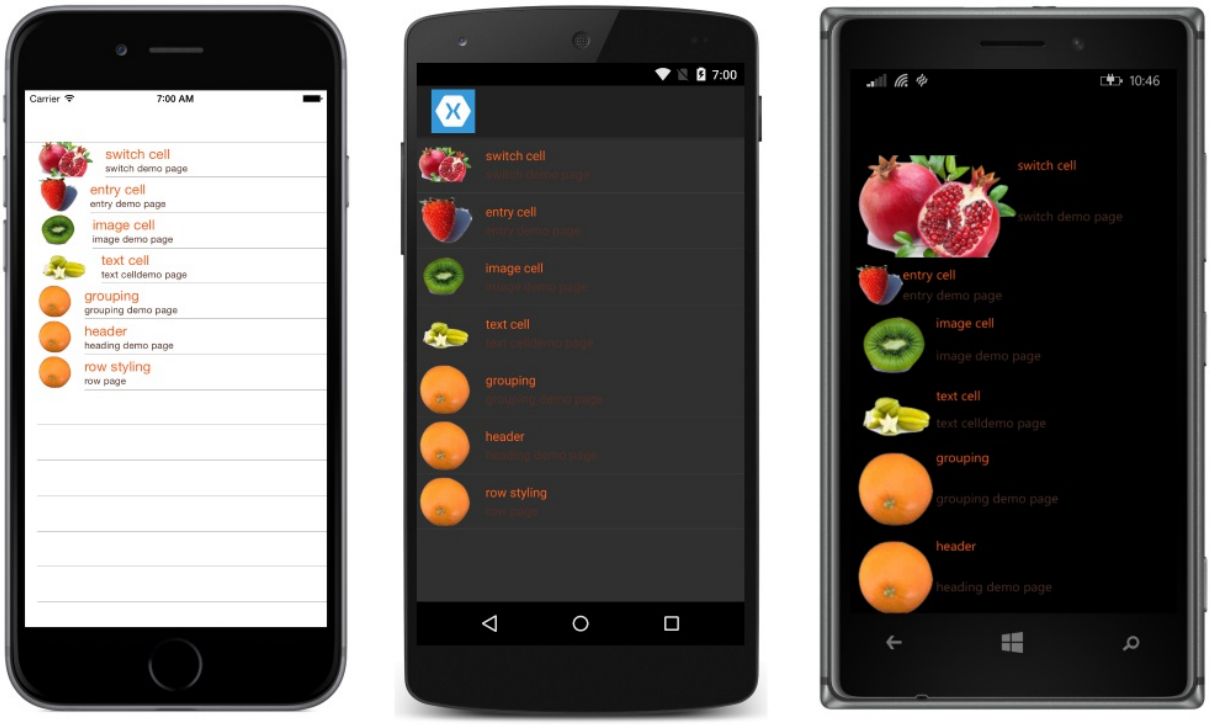
ImageCell

`ImageCell` 如 `TextCell`，可以用于显示文本和辅助的详细信息，并通过使用每个平台的本机控件提供优异的性能。`ImageCell` 不同于 `TextCell`，因为它的文本的左侧显示图像。

`ImageCell` 如果需要显示数据的可视方面，例如的联系人或电影列表的列表时很有用。ImageCells 可自定义，使你能够设置：

- `Text` – 在第一行，用大字体显示文本
- `Detail` – 较小的字体中的第一行下面所示的文本
- `TextColor` – 文本的颜色
- `DetailColor` – 详细信息文本的颜色
- `ImageSource` – 要在文本旁边显示的图像





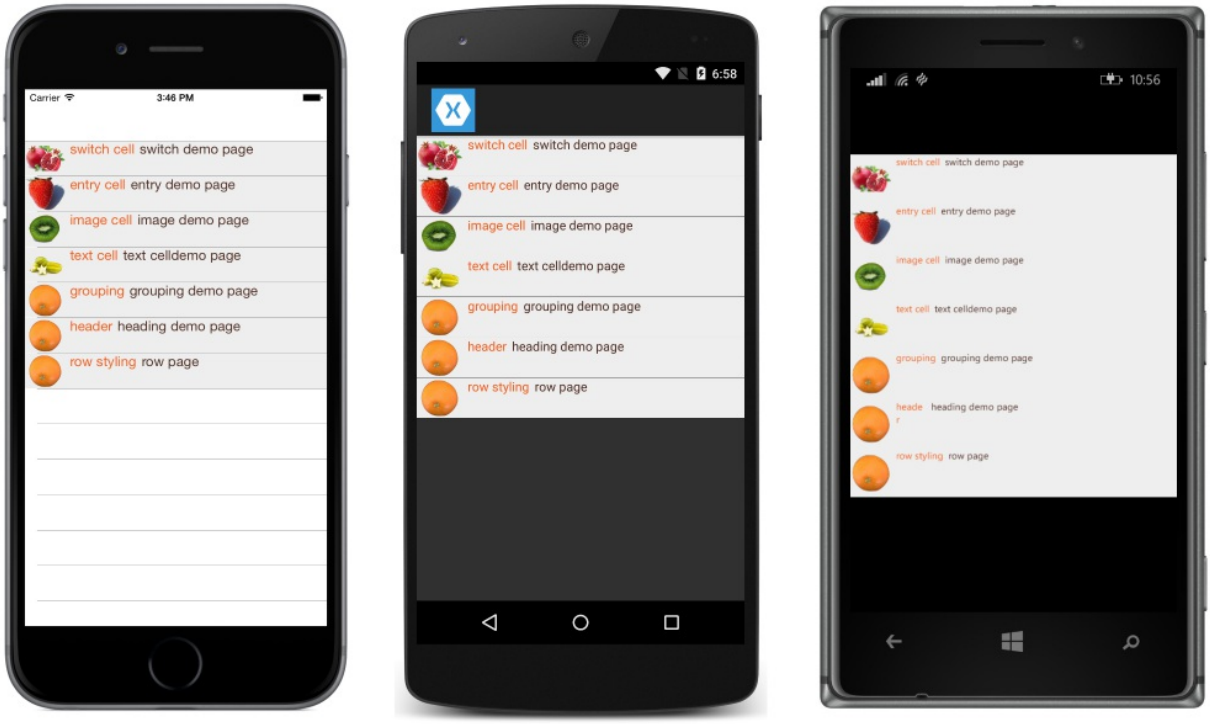
自定义单元格

当内置的单元格未提供必需的布局时，自定义单元格实现必需的布局。例如，你可能想要提供两个标签具有相等的权重的单元格。一个 `TextCell` 不足是因为 `TextCell` 具有较小的一个标签。大多数单元格自定义添加其他只读数据（如其他标签、图像或其他显示信息）。

自定义的所有单元格必须派生自 `ViewCell`，所有内置的单元格类型使用相同的基类。

Xamarin.Forms 2 引入了一个新缓存行为上 `ListView` 可以设置以提高滚动性能对于某些类型的自定义单元格的控件。

这是自定义单元格的一个示例：



XAML

若要创建上述布局 XAML 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="demoListView.ImageCellPage">
    <ContentPage.Content>
        <ListView  x:Name="listView">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <StackLayout BackgroundColor="#eee"
                        Orientation="Vertical">
                            <StackLayout Orientation="Horizontal">
                                <Image Source="{Binding image}" />
                                <Label Text="{Binding title}"
                                TextColor="#f35e20" />
                                <Label Text="{Binding subtitle}"
                                HorizontalOptions="EndAndExpand"
                                TextColor="#503026" />
                            </StackLayout>
                        </StackLayout>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

上述 XAML 执行大量操作。让我们深入细致地：

- 自定义单元格嵌套在 `DataTemplate`，这是内部 `ListView.ItemTemplate`。这是与使用的任何其他单元相同的进程。
- `ViewCell` 是类型的自定义单元格。子级 `DataTemplate` 元素必须是或派生类型 `ViewCell`。
- 请注意，该内部 `ViewCell`，由管理布局 `StackLayout`。此布局可用于自定义的背景色。请注意，任何属性 `StackLayout`，它是可绑定可以绑定自定义单元格，内部，尽管此处未显示的。

C#

在 C# 中指定自定义单元格是 XAML 等效项的更详细。让我们来实际操作一下：

首先，定义自定义单元格类，`ViewCell` 作为基类：

```
public class CustomCell : ViewCell
    {
        public CustomCell()
        {
            //instantiate each of our views
            var image = new Image ();
            StackLayout cellWrapper = new StackLayout ();
            StackLayout horizontalLayout = new StackLayout ();
            Label left = new Label ();
            Label right = new Label ();

            //set bindings
            left.SetBinding (Label.TextProperty, "title");
            right.SetBinding (Label.TextProperty, "subtitle");
            image.SetBinding (Image.SourceProperty, "image");

            //Set properties for desired design
            cellWrapper.BackgroundColor = Color.FromHex ("#eee");
            horizontalLayout.Orientation = StackOrientation.Horizontal;
            right.HorizontalOptions = LayoutOptions.EndAndExpand;
            left.TextColor = Color.FromHex ("#f35e20");
            right.TextColor = Color.FromHex ("503026");

            //add views to the view hierarchy
            horizontalLayout.Children.Add (image);
            horizontalLayout.Children.Add (left);
            horizontalLayout.Children.Add (right);
            cellWrapper.Children.Add (horizontalLayout);
            View = cellWrapper;
        }
    }
```

在构造函数具有页面 `ListView`，设置 ListView `ItemTemplate` 属性设置为一个新 `DataTemplate`：

```
public partial class ImageCellPage : ContentPage
    {
        public ImageCellPage ()
        {
            InitializeComponent ();
            listView.ItemTemplate = new DataTemplate (typeof(CustomCell));
        }
    }
```

请注意，对于构造函数 `DataTemplate` 接受一个类型。Typeof 运算符获取的 CLR 类型 `CustomCell`。

绑定上下文的更改

当绑定到自定义单元格类型的 `BindableProperty` 实例，显示的 UI 控件 `BindableProperty` 的值应使用 `OnBindingContextChanged` 替代以将设置要在中显示的数据每个单元格，而不是单元格构造函数，如下面的代码示例中所示：

```
public class CustomCell : ViewCell
{
    Label nameLabel, ageLabel, locationLabel;

    public static readonly BindableProperty NameProperty =
        BindableProperty.Create ("Name", typeof(string), typeof(CustomCell), "Name");
    public static readonly BindableProperty AgeProperty =
        BindableProperty.Create ("Age", typeof(int), typeof(CustomCell), 0);
    public static readonly BindableProperty LocationProperty =
        BindableProperty.Create ("Location", typeof(string), typeof(CustomCell), "Location");

    public string Name {
        get { return(string)GetValue (NameProperty); }
        set { SetValue (NameProperty, value); }
    }

    public int Age {
        get { return(int)GetValue (AgeProperty); }
        set { SetValue (AgeProperty, value); }
    }

    public string Location {
        get { return(string)GetValue (LocationProperty); }
        set { SetValue (LocationProperty, value); }
    }
    ...

    protected override void OnBindingContextChanged ()
    {
        base.OnBindingContextChanged ();

        if (BindingContext != null) {
            nameLabel.Text = Name;
            ageLabel.Text = Age.ToString ();
            locationLabel.Text = Location;
        }
    }
}
```

`OnBindingContextChanged` 重写时将会调用 `BindingContextChanged` 事件触发时，响应的值 `BindingContext` 属性更改。因此，当 `BindingContext` 发生更改，显示的 UI 控件 `BindableProperty` 值应设置其数据。请注意，`BindingContext` 应检查是否有 `null` 值，因为这可以通过 Xamarin.Forms 设置，进行垃圾回收，这反过来会导致 `OnBindingContextChanged` 重写调用。

或者，UI 控件可以绑定到 `BindableProperty` 实例，以显示它们的值，无需重写 `OnBindingContextChanged` 方法。

> **NOTE**
>
> 重写时 `OnBindingContextChanged`，确保的基类 `OnBindingContextChanged` 方法调用，以便注册的委托接收 `BindingContextChanged` 事件。

在 XAML 中，绑定到数据的自定义单元格类型可以实现的如下面的代码示例中所示：

```
<ListView x:Name="listView">
    <ListView.ItemTemplate>
        <DataTemplate>
            <local:CustomCell Name="{Binding Name}" Age="{Binding Age}" Location="{Binding Location}" />
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

这会将绑定 `Name`、`Age`，并 `Location` 中的可绑定属性 `CustomCell` 实例，为 `Name`、`Age`，和 `Location` 基础集合中每个对象的属性。

在 C# 中的等效绑定以下的代码示例所示：

```
var customCell = new DataTemplate (typeof(CustomCell));
customCell.SetBinding (CustomCell.NameProperty, "Name");
customCell.SetBinding (CustomCell.AgeProperty, "Age");
customCell.SetBinding (CustomCell.LocationProperty, "Location");

var listView = new ListView {
    ItemsSource = people,
    ItemTemplate = customCell
};
```

在 iOS 和 Android，如果 `ListView` 回收元素和自定义单元格使用自定义呈现器，自定义呈现器必须正确实现属性更改通知。重用单元格时其属性值将更改时的绑定上下文将与更新为的一个可用的单元格，`PropertyChanged` 引发的事件的。有关详细信息，请参阅自定义 ViewCell。有关单元格回收的详细信息，请参阅缓存策略。

相关链接

- 构建的单元格（示例）
- 自定义单元格（示例）
- 绑定上下文更改（示例）

自定义 ListView 外观

2018/7/13 • • Edit Online

`ListView` 具有用于控制总体列表中，除了基础的演示文稿选项 `ViewCell` s。选项包括：

- **分组** –更方便的导航和改进的组织在 ListView 中的项进行分组。
- **页眉和页脚** –的开头和结尾的滚动与其他项的视图在显示的信息。
- **行分隔符** –显示或隐藏项之间的分隔线。
- **变量高度行** –默认情况下的所有行都均为相同的高度，但这可以更改为允许具有不同高度要显示的行。

分组

通常，大型数据集可以变得难以处理时不断滚动列表中显示。启用分组可以更好地组织内容和激活轻松导航数据的特定于平台的控件通过提高在这些情况下的用户体验。

有关激活分组时 `ListView`，为每个组添加一个标题行。

若要启用分组：

- 创建列表的列表（组的列表，每个组正在元素的列表）。
- 设置 `ListView` 的 `ItemsSource` 到该列表。
- 设置 `IsGroupingEnabled` 为 true。
- 设置 `GroupDisplayBinding` 要绑定到正用作组的标题的组的属性。
- [可选]设置 `GroupShortNameBinding` 要绑定到要用作组的短名称的组的属性。短名称用于跳转列表（在 iOS 上的右侧列）。

首先创建组的类：

```
public class PageTypeGroup : List<PageModel>
    {
        public string Title { get; set; }
        public string ShortName { get; set; } //will be used for jump lists
        public string Subtitle { get; set; }
        private PageTypeGroup(string title, string shortName)
        {
            Title = title;
            ShortName = shortName;
        }

        public static IList<PageTypeGroup> All { private set; get; }
    }
```

在上面的代码，`All` 是要提供给我们 ListView 用作绑定源的列表。`Title` 和 `ShortName` 是将用于组标题的属性。

在此阶段，`All` 为空列表。添加静态构造函数，使程序启动时将填充列表：

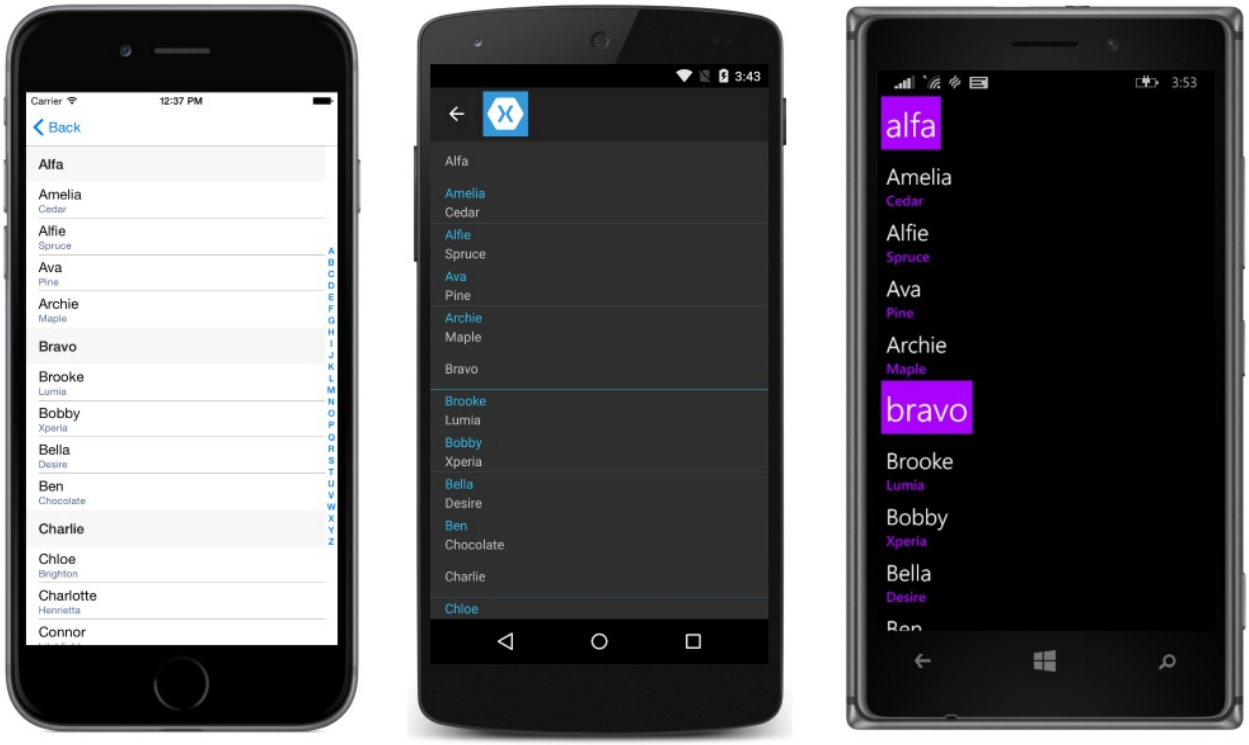
```
static PageTypeGroup()
{
    List<PageTypeGroup> Groups = new List<PageTypeGroup> {
            new PageTypeGroup ("Alfa", "A"){
                new PageModel("Amelia", "Cedar", new switchCellPage(),""),
                new PageModel("Alfie", "Spruce", new switchCellPage(), "grapefruit.jpg"),
                new PageModel("Ava", "Pine", new switchCellPage(), "grapefruit.jpg"),
                new PageModel("Archie", "Maple", new switchCellPage(), "grapefruit.jpg")
            },
            new PageTypeGroup ("Bravo", "B"){
                new PageModel("Brooke", "Lumia", new switchCellPage(),""),
                new PageModel("Bobby", "Xperia", new switchCellPage(), "grapefruit.jpg"),
                new PageModel("Bella", "Desire", new switchCellPage(), "grapefruit.jpg"),
                new PageModel("Ben", "Chocolate", new switchCellPage(), "grapefruit.jpg")
            }
        }
        All = Groups; //set the publicly accessible list
}
```

在上述代码中我们可以调用 `Add` 上的元素 `groups`，这是类型的实例 `PageTypeGroup`。这可能是因为 `PageTypeGroup` 继承 `List<PageModel>`。这是列表的上面记下列表模式的示例。

下面是用于显示分组的列表 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="DemoListView.GroupingViewPage"
    <ContentPage.Content>
        <ListView  x:Name="GroupedView"
        GroupDisplayBinding="{Binding Title}"
        GroupShortNameBinding="{Binding ShortName}"
        IsGroupingEnabled="true">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                     Detail="{Binding Subtitle}" />
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

这将产生以下结果:



请注意，我们具有：

- 设置 `GroupShortNameBinding` 到 `ShortName` 我们组的类中定义的属性
- 设置 `GroupDisplayBinding` 到 `Title` 我们组的类中定义的属性
- 设置 `IsGroupingEnabled` 为 true
- 更改 `ListView` 的 `ItemsSource` 到分组列表

自定义分组

如果在列表中已启用分组，也可以定制的组标头。

类似于如何 `ListView` 已 `ItemTemplate` 用于定义如何显示行 `ListView` 具有 `GroupHeaderTemplate` 。

自定义 XAML 中的组标头的示例如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="DemoListView.GroupingViewPage">
    <ContentPage.Content>
        <ListView x:Name="GroupedView"
         GroupDisplayBinding="{Binding Title}"
         GroupShortNameBinding="{Binding ShortName}"
         IsGroupingEnabled="true">
            <ListView.ItemTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                    Detail="{Binding Subtitle}"
                    TextColor="#f35e20"
                    DetailColor="#503026" />
                </DataTemplate>
            </ListView.ItemTemplate>
            <!-- Group Header Customization-->
            <ListView.GroupHeaderTemplate>
                <DataTemplate>
                    <TextCell Text="{Binding Title}"
                    Detail="{Binding ShortName}"
                    TextColor="#f35e20"
                    DetailColor="#503026" />
                </DataTemplate>
            </ListView.GroupHeaderTemplate>
            <!-- End Group Header Customization -->
        </ListView>
    </ContentPage.Content>
</ContentPage>
```

页眉和页脚

很可能 ListView 显示页眉和页脚的向下滚动列表中的元素。页眉和页脚可以是文本字符串或更复杂的布局。请注意，这是独立于部分组。

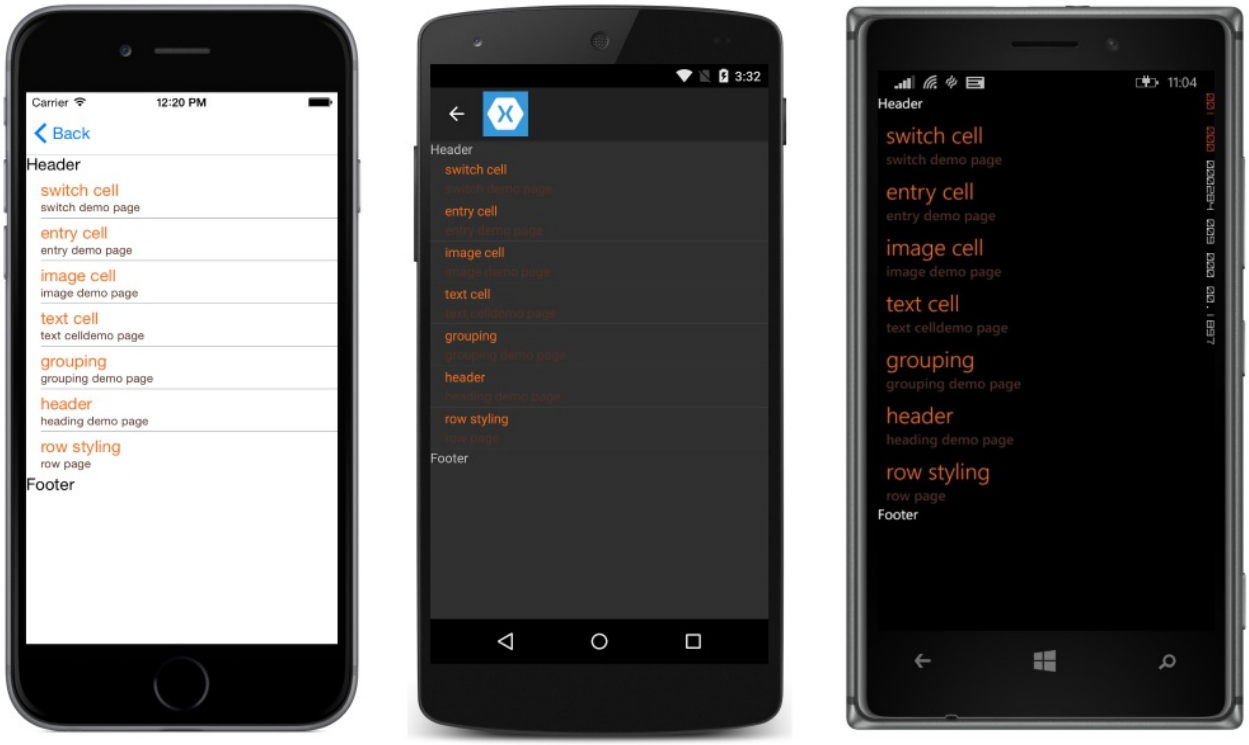
可以设置 `Header` 和/或 `Footer` 到一个简单的字符串值，也可以将其设置为更复杂的布局。此外，还有 `HeaderTemplate` 和 `FooterTemplate` 属性，可创建更复杂的页眉和页脚的布局支持数据绑定的。

若要创建简单的页眉/页脚，只需为你想要显示的文本设置页眉或页脚属性。在代码中：

```
ListView HeaderList = new ListView() {
    Header = "Header",
    Footer = "Footer"
    };
```

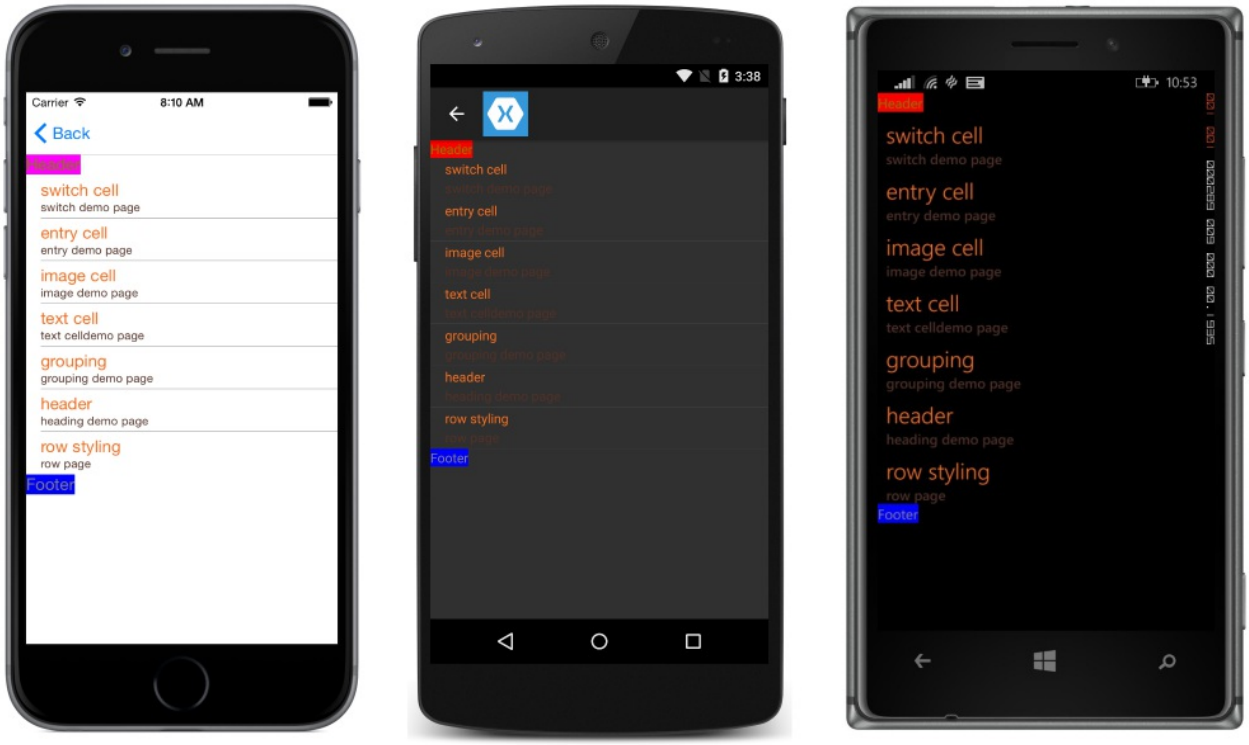
在 XAML:

```
<ListView  x:Name="HeaderList"  Header="Header" Footer="Footer"></ListView>
```



若要创建自定义的页眉和页脚，请定义页眉和页脚视图：

```
<ListView.Header>
    <StackLayout Orientation="Horizontal">
        <Label Text="Header"
        TextColor="Olive"
        BackgroundColor="Red" />
    </StackLayout>
</ListView.Header>
<ListView.Footer>
    <StackLayout Orientation="Horizontal">
        <Label Text="Footer"
        TextColor="Gray"
        BackgroundColor="Blue" />
    </StackLayout>
</ListView.Footer>
```



行分隔符

分隔符线之间显示 `ListView` 默认情况下在 iOS 和 Android 上的元素。如果你想隐藏在 iOS 和 Android 上的分隔符线，设置 `SeparatorVisibility` 在 ListView 中的属性。选项为 `SeparatorVisibility` 是：

- **默认**-iOS 和 Android 上显示一条分隔线。
- **无**-隐藏所有平台上的分隔符。

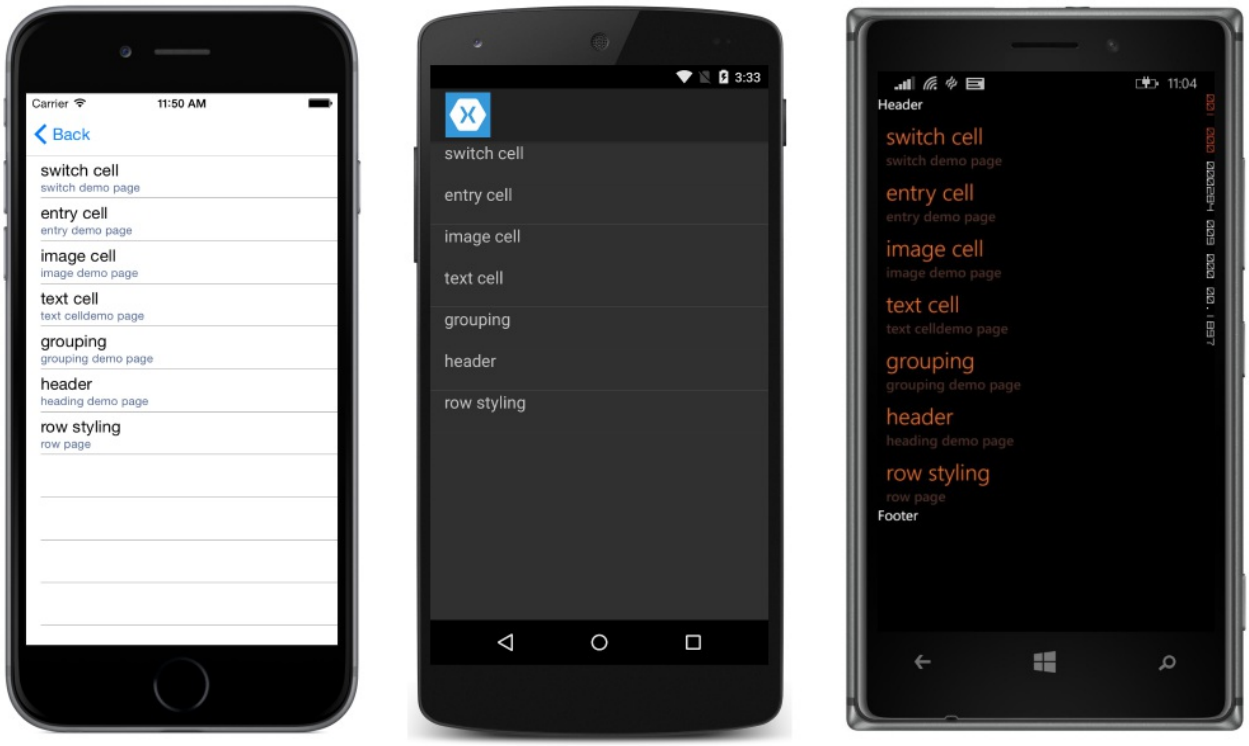
默认可见性：

C#：

```
SepratorDemoListView.SeparatorVisibility = SeparatorVisibility.Default;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="Default" />
```



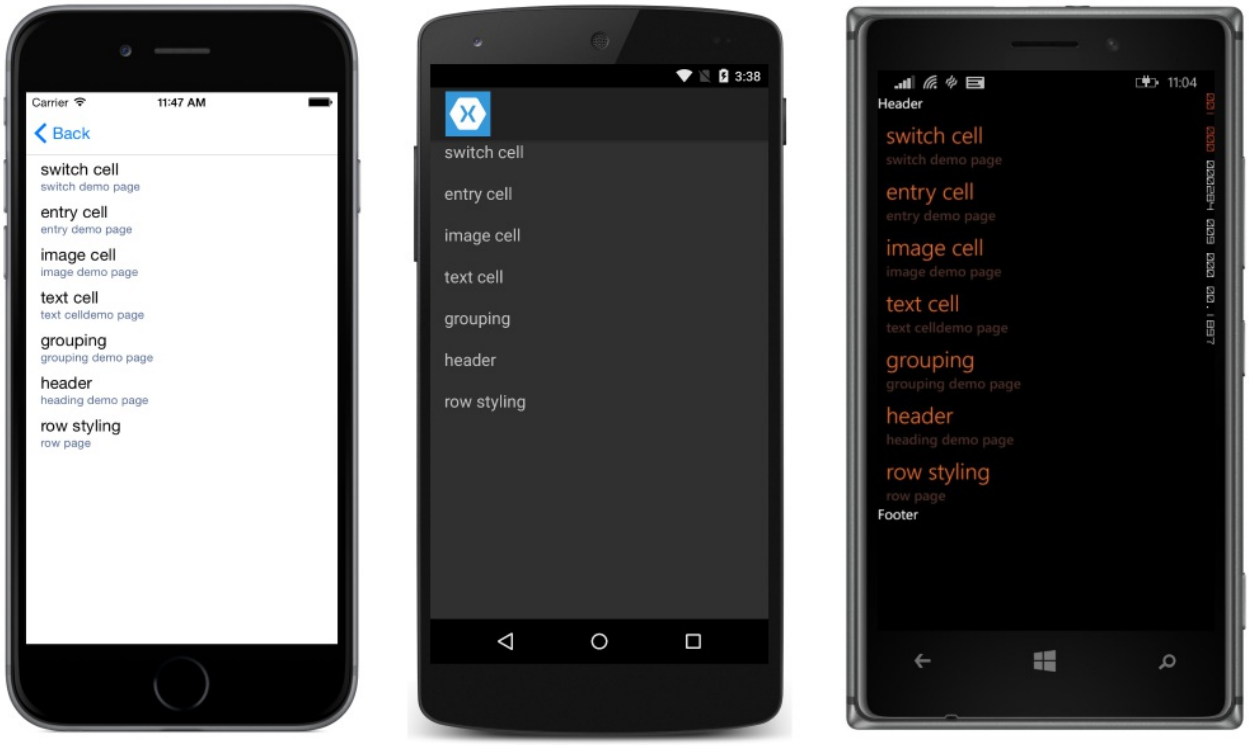
None:

C#:

```
SepratorDemoListView.SeparatorVisibility = SeparatorVisibility.None;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorVisibility="None" />
```



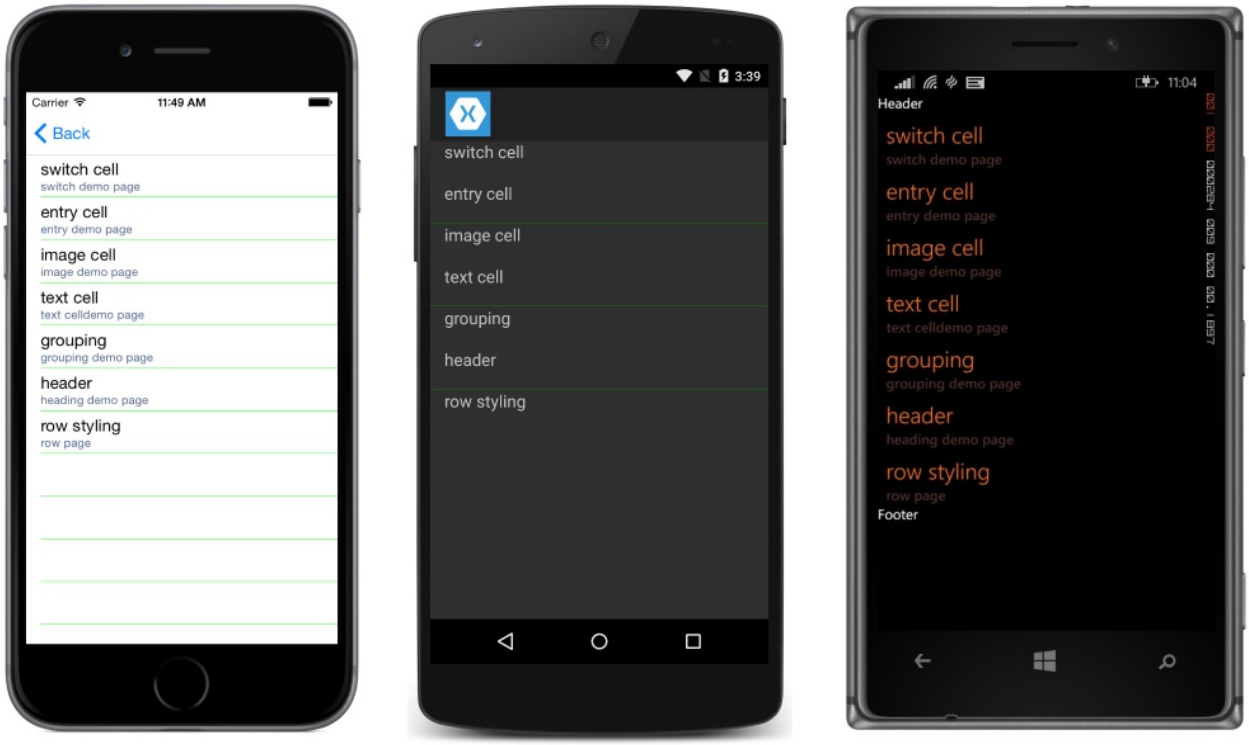
此外可以设置通过分隔线的颜色 `SeparatorColor` 属性：

C#:

```
SepratorDemoListView.SeparatorColor = Color.Green;
```

XAML:

```
<ListView x:Name="SeparatorDemoListView" SeparatorColor="Green" />
```



NOTE

设置这些属性在 Android 上加载后 `ListView` 会产生对较大性能产生负面影响。

行高

默认情况下，ListView 中的所有行都具有相同的高度。ListView 有可用于更改该行为的两个属性：

- `HasUnevenRows` – `true` / `false` 值，如果行具有不同高度设置为 `true` 。默认为 `false` 。
- `RowHeight` – 集的每个高度行何时 `HasUnevenRows` 是 `false` 。

通过设置可设置的所有行的高度 `RowHeight` 属性上的 `ListView` 。

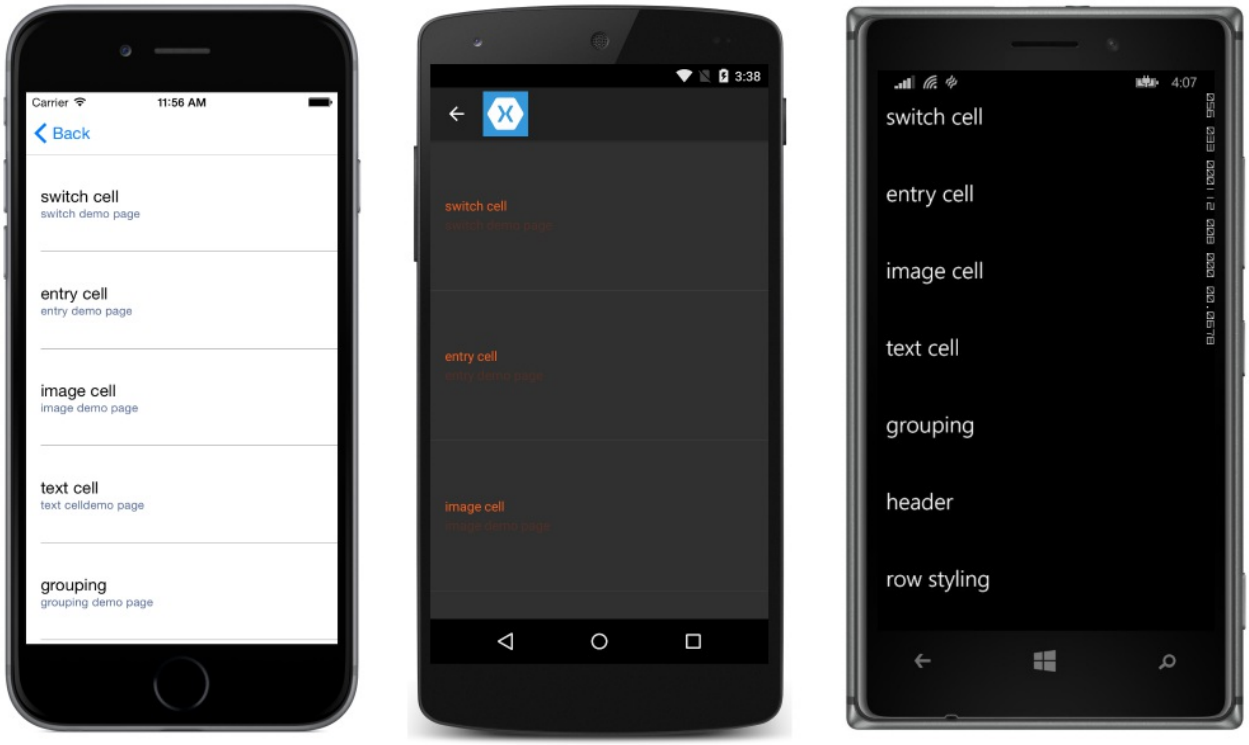
自定义固定的行高

C#:

```
RowHeightDemoListView.RowHeight = 100;
```

XAML:

```
<ListView x:Name="RowHeightDemoListView" RowHeight="100" />
```



以不相等的行

如果你想要具有不同高度的单个行，则可以设置`HasUnevenRows`属性设置为`true`。请注意行高不需要手动设置一次`HasUnevenRows`已设置为`true`，因为 xamarin.forms 自动计算高度。

C#:

```
RowHeightDemoListView.HasUnevenRows = true;
```

XAML:

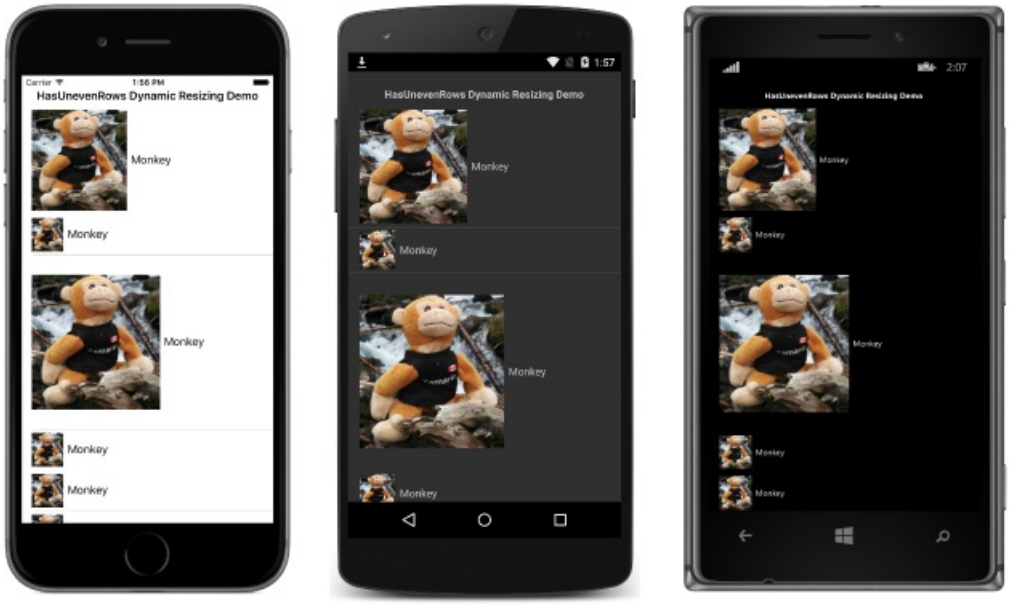
```
<ListView x:Name="RowHeightDemoListView" HasUnevenRows="true" />
```

运行时调整大小的行

各个 `ListView` 行在运行时，提供的可以以编程方式调整大小 `HasUnevenRows` 属性设置为 `true`。 `Cell.ForceUpdateSize` 方法更新单元格的大小，即使它不是当前可见，如下面的代码示例中所示：

```
void OnImageTapped (object sender, EventArgs args)
{
    var image = sender as Image;
    var viewCell = image.Parent.Parent as ViewCell;

    if (image.HeightRequest < 250) {
        image.HeightRequest = image.Height + 100;
        viewCell.ForceUpdateSize ();
    }
}
```

`OnImageTapped` 事件处理程序执行以响应 `Image` 在单元格中被点击，并会增加的大小 `Image`，以便轻松地查看该单元中显示。



请注意是否过度使用此功能没有强可能导致性能下降。

相关链接

- 分组（示例）
- 自定义呈现器视图（示例）
- 动态调整大小的行（示例）
- 1.4 的发行说明
- 1.3 的发行说明

ListView 交互性

2018/8/6 • • Edit Online

ListView 支持与它将通过以下方法提供的数据进行交互：

- **选择和点击** –响应点击和项的选择/取消选择。启用或禁用行选择（默认情况下启用）。
- **上下文操作** –公开功能每个项，例如，--删除轻扫。
- **下拉刷新** –实现下拉刷新用语，用户都希望能从本机体验。

选择和分流点

`ListView` 通过设置控制选择模式 `ListView.SelectionMode` 属性的值 `ListViewSelectionMode` 枚举：

- `Single` 指示可以将选择单个项目，而被突出显示的选定项。这是默认值。
- `None` 指示不能选择项。

当用户点击某个项时，会触发两个事件：

- `ItemSelected` 当选择了新项时激发。
- `ItemTapped` 点击某个项时触发。

> **NOTE**
> 两次点击相同项会触发两个 `ItemTapped` 事件，但将只触发单个 `ItemSelected` 事件。

当 `SelectionMode` 属性设置为 `Single` 中的项 `ListView` 可以选择 `ItemSelected` 并 `ItemTapped` 将不再触发事件，并且 `SelectedItem` 属性将设置为所选的项的值。

当 `SelectionMode` 属性设置为 `None` 中的项 `ListView` 不能选择 `ItemSelected` 将不会触发事件，并 `SelectedItem` 属性将保持 `null`。但是， `ItemTapped` 仍将触发事件和项目将会短暂地突出显示分流期间。

当选择项并 `SelectionMode` 属性更改从 `Single` 到 `None` ，则 `SelectedItem` 属性将设置为 `null` 并 `ItemSelected` 事件将触发与 `null` 项。

下面的屏幕截图演示 `ListView` 与默认选择模式：

禁用所选内容

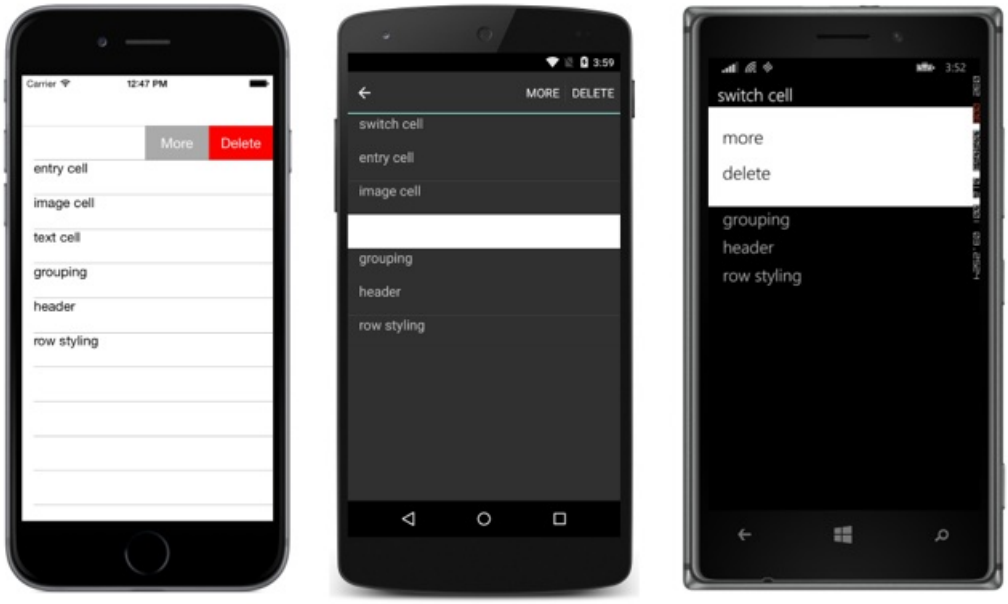
若要禁用 `ListView` 选集 `SelectionMode` 属性设置为 `None`：

```
<ListView ... SelectionMode="None" />
```

```
var listView = new ListView { ... SelectionMode = ListViewSelectionMode.None };
```

上下文操作

通常情况下，用户将需要执行操作中的项 `ListView`。例如，请考虑在邮件应用中的电子邮件的列表。在 iOS 上，您可以轻扫来删除一条消息::



可以在 C# 和 XAML 实现上下文操作。下面您会发现特定的指南的同时，但首先让我们看一看一些关键实现细节两个。

使用创建上下文操作 `MenuItem` s。菜单项的点击事件会引发 MenuItem 本身，而不 ListView。这是不同于如何将点击事件处理对于 ListView 其中引发事件而不是该单元格的单元格。因为 ListView 引发事件，其事件处理程序是给定密钥的信息，如其中选择或点击项。

默认情况下，菜单项具有无法知道它属于哪个单元格。`CommandParameter` 可在上找到 `MenuItem` 来存储对象，如后面的 MenuItem ViewCell 对象。`CommandParameter` 可以在 XAML 和 C# 中设置。

C#

可以在任何实现上下文操作 `Cell` 子类（只要它不被用作组标头）通过创建 `MenuItem`s 并将它们添加到 `ContextActions` 单元格的集合。具有以下属性可配置的上下文操作：

- **文本**–菜单项中显示的字符串。
- **单击**–时单击项的事件。
- **IsDestructive** –（可选）为 true 时呈现的项是以不同的方式在 iOS 上。

多个上下文操作可以添加到单元格，，但是只有一个应有 `IsDestructive` 设置为 `true` 。下面的代码演示如何上下文操作将添加到 `ViewCell`：

```
var moreAction = new MenuItem { Text = "More" };
moreAction.SetBinding (MenuItem.CommandParameterProperty, new Binding ("."));
moreAction.Clicked += async (sender, e) => {
    var mi = ((MenuItem)sender);
    Debug.WriteLine("More Context Action clicked: " + mi.CommandParameter);
};

var deleteAction = new MenuItem { Text = "Delete", IsDestructive = true }; // red background
deleteAction.SetBinding (MenuItem.CommandParameterProperty, new Binding ("."));
deleteAction.Clicked += async (sender, e) => {
    var mi = ((MenuItem)sender);
    Debug.WriteLine("Delete Context Action clicked: " + mi.CommandParameter);
};
// add to the ViewCell's ContextActions property
ContextActions.Add (moreAction);
ContextActions.Add (deleteAction);
```

XAML

`MenuItem`s 也可以创建 XAML 集合中以声明方式。以下 XAML 演示自定义单元格包含两个实现的上下文操作：

```
<ListView x:Name="ContextDemoList">
  <ListView.ItemTemplate>
    <DataTemplate>
      <ViewCell>
         <ViewCell.ContextActions>
            <MenuItem Clicked="OnMore" CommandParameter="{Binding .}"
               Text="More" />
            <MenuItem Clicked="OnDelete" CommandParameter="{Binding .}"
               Text="Delete" IsDestructive="True" />
         </ViewCell.ContextActions>
         <StackLayout Padding="15,0">
              <Label Text="{Binding title}" />
         </StackLayout>
      </ViewCell>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

在代码隐藏文件中，确保 `Clicked` 方法实现：

```
public void OnMore (object sender, EventArgs e) {
    var mi = ((MenuItem)sender);
    DisplayAlert("More Context Action", mi.CommandParameter + " more context action", "OK");
}

public void OnDelete (object sender, EventArgs e) {
    var mi = ((MenuItem)sender);
    DisplayAlert("Delete Context Action", mi.CommandParameter + " delete context action", "OK");
}
```

NOTE

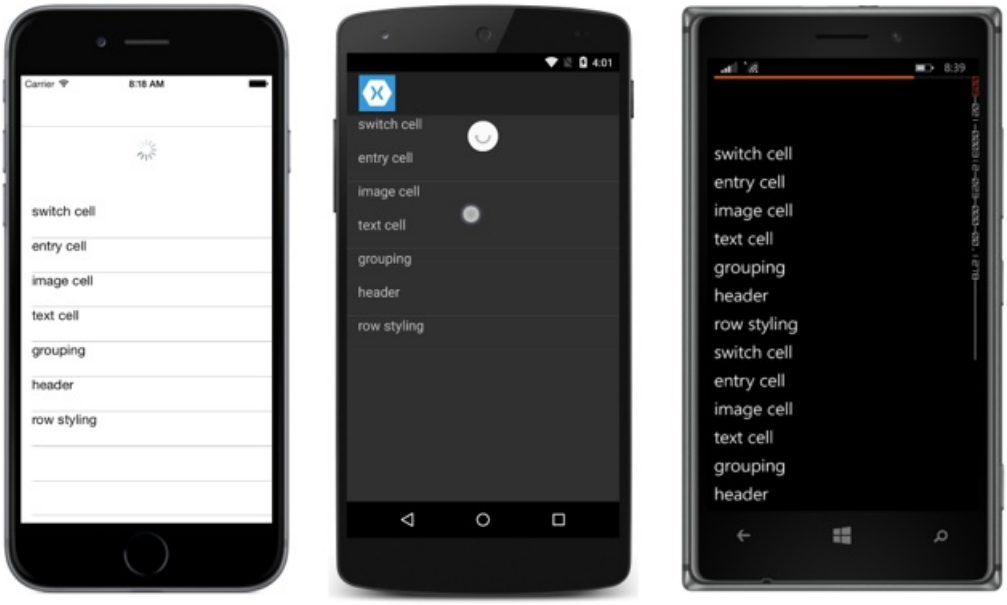
`NavigationPageRenderer` 适用于 Android 可重写 `UpdateMenuItemIcon` 方法，可以用来从自定义加载图标 `Drawable` 。此重写就可以使用 SVG 图像以图标形式在 `MenuItem` Android 上的实例。

下拉以刷新

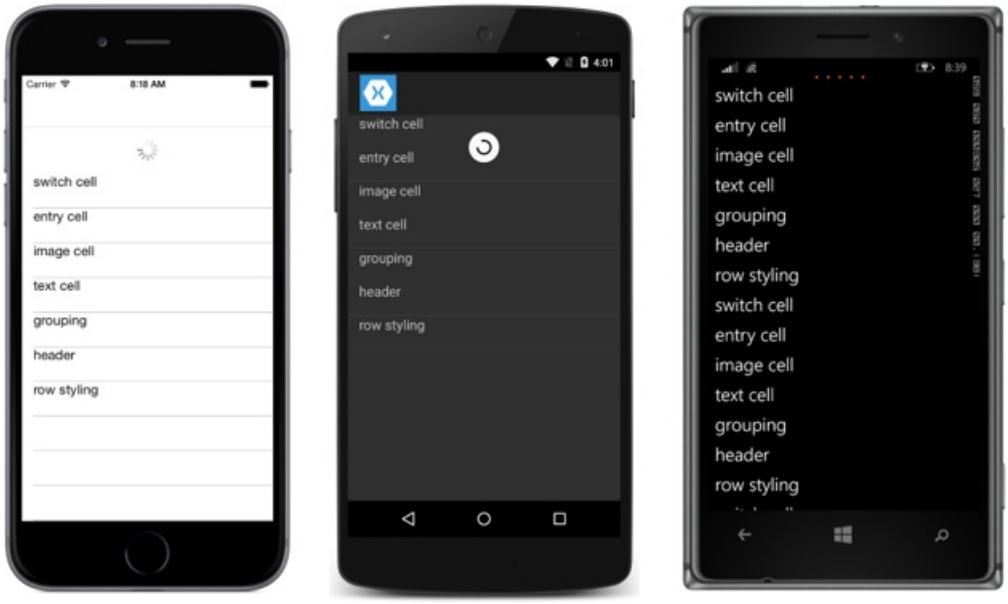
用户都希望能拉下在列表中的数据将刷新该列表。`ListView` 支持此--现成的。若要启用下拉刷新功能，请设置 `IsPullToRefreshEnabled` 为 true:

```
listView.IsPullToRefreshEnabled = true;
```

拉取请求以刷新以用户身份：



以用户身份的请求以刷新已发布的请求。这是用户看到同时正在更新列表：



ListView 公开了几个事件，可用于对请求刷新事件做出响应。

- `RefreshCommand` 将调用和 `Refreshing` 调用的事件。 `IsRefreshing` 将设置为 `true` 。
- 您应该执行刷新列表视图中，在命令或事件中的内容所需的任何代码。
- 刷新时已完成，请调用 `EndRefresh` ，或者设置 `IsRefreshing` 到 `false` 告诉完成列表视图。

`CanExecute` 属性，则考虑，后者可提供一种方法来控制是否应启用下拉刷新命令。

相关链接

- ListView 交互性（示例）
- 1.4 的发行说明
- 1.3 的发行说明

ListView 性能

2018/7/13 • • Edit Online

如果编写移动应用程序，性能很重要。用户都希望能平滑滚动并快速加载。未能满足用户的期望将您的应用程序商店中的评级或成本对于业务线应用程序，成本的组织的时间和金钱。

尽管 `ListView` 是一个功能强大的视图用于显示数据，它具有一些限制。使用自定义单元格，尤其是当它们包含深度嵌套的视图层次结构或使用需要大量度量的特定布局时，滚动性能会降低。幸运的是，有可用于避免性能不佳的技术。

缓存策略

Listview 通常用于显示更多的数据不是可以在屏幕上的合适大小。例如，考虑一个音乐应用程序。歌曲的库可能会有成千上万个条目。最简单的方法，就是创建每首歌曲的行，会使性能变差。该方法会浪费宝贵的内存，并可能会降低滚动到爬网。另一种方法是创建和销毁行，如将数据滚动到视图。这需要常量实例化并清除视图对象，可能会很慢。

若要节省内存，本机 `ListView` 为每个平台的等效项具有内置功能来重新使用的行。仅在屏幕上可见的单元加载到内存中并内容加载到现有的单元格。这可以防止无需实例化的对象，从而节省时间和内存的数千个应用程序。

Xamarin.Forms 允许 `ListView` 通过重复使用单元格 `ListViewCachingStrategy` 枚举，它具有以下值：

```
public enum ListViewCachingStrategy
{
  RetainElement,   // the default value
  RecycleElement,
  RecycleElementAndDataTemplate
}
```

> **NOTE**
> 通用 Windows 平台 (UWP) 将忽略 `RetainElement` 缓存策略，因为它始终使用缓存来提高性能。因此，默认情况下其行为如同 `RecycleElement` 应用缓存策略。

RetainElement

`RetainElement` 缓存策略指定 `ListView` 将在列表中，生成的每个项的单元格，并且是默认 `ListView` 行为。它通常应在以下情况下使用：

- 每个单元格时有大量的绑定 (20-30 +)。
- 当单元格模板频繁更改。
- 当测试中发现的 `RecycleElement` 缓存策略导致降低的执行速度。

务必要识别所带来的后果 `RetainElement` 缓存策略时使用的自定义单元格。任何单元格初始化代码将需要为每个单元格创建过程中，运行的可为每秒多次。在这种情况下，在页上，没有布局技术，如使用多个嵌套 `StackLayout` 实例时安装程序会成为性能瓶颈和销毁在真实时间中，当用户滚动。

RecycleElement

`RecycleElement` 缓存策略指定 `ListView` 将尝试通过回收列表单元格，最小化其内存占用量和执行速度。此模式下不会始终提供性能改进，并应执行测试以确定任何改进。但是，它通常是理想之选，并应在以下情况下使用：

- 当每个单元格有一个小到中等数量的绑定。

- 当每个单元格 `BindingContext` 定义所有的单元格数据。
- 每个单元格时很大程度上类似，与单元格模板不变。

在虚拟化期间，该单元格都有更新，其绑定上下文，因此如果应用程序使用此模式下则必须确保适当地处理绑定上下文更新。有关单元格的所有数据必须都来自绑定上下文或可能会出现一致性错误。这可以通过使用数据绑定显示单元格数据。或者，应将单元格数据设置 `OnBindingContextChanged` 重写，而不是在自定义单元格的构造函数，如下面的代码示例中所示：

```
public class CustomCell : ViewCell
{
  Image image = null;

  public CustomCell ()
  {
    image = new Image();
    View = image;
  }

  protected override void OnBindingContextChanged ()
  {
    base.OnBindingContextChanged ();

    var item = BindingContext as ImageItem;
    if (item != null) {
      image.Source = item.ImageUrl;
    }
  }
}
```

有关详细信息，请参阅绑定上下文更改。

在 iOS 和 Android 上，如果单元格使用自定义呈现器，它们必须确保正确实现属性更改通知。重用单元格时其属性值将更改时的绑定上下文将与更新为的一个可用的单元格，`PropertyChanged` 引发的事件的。有关详细信息，请参阅自定义 ViewCell。

使用 DataTemplateSelector RecycleElement

当 `ListView` 使用 `DataTemplateSelector` 选择 `DataTemplate`，则 `RecycleElement` 缓存策略不会缓存 `DataTemplate`s。相反，`DataTemplate` 为每个项列表中的数据选择。

> **NOTE**
>
> `RecycleElement` 缓存策略具有必备条件，在 Xamarin.Forms 2.4 中引入的当 `DataTemplateSelector` 需要选择 `DataTemplate` 每个 `DataTemplate` 必须返回相同 `ViewCell` 类型。例如，给定 `ListView` 与 `DataTemplateSelector`，可以返回 `MyDataTemplateA`（其中 `MyDataTemplateA` 返回 `ViewCell` 类型的 `MyViewCellA`），或 `MyDataTemplateB`（其中 `MyDataTemplateB` 将返回 `ViewCell` 类型的 `MyViewCellB`），当 `MyDataTemplateA` 将返回它必须返回 `MyViewCellA`，否则将引发异常。

RecycleElementAndDataTemplate

`RecycleElementAndDataTemplate` 缓存策略是基于 `RecycleElement` 此外确保，当缓存策略 `ListView` 使用 `DataTemplateSelector` 以选择 `DataTemplate`，`DataTemplate`s 所缓存的列表中的项的类型。因此，`DataTemplate`s 每个项类型，而不是每个项实例一次选择一次。

> **NOTE**
>
> `RecycleElementAndDataTemplate` 缓存策略具有必备组件的 `DataTemplate`s 返回 `DataTemplateSelector` 必须使用 `DataTemplate` 构造函数采用 `Type`。

设置缓存策略

`ListViewCachingStrategy` 枚举值指定具有 `ListView` 构造函数重载，如下面的代码示例中所示：

```
var listView = new ListView(ListViewCachingStrategy.RecycleElement);
```

在 XAML 中，设置 `CachingStrategy` 特性，如下面的代码中所示：

```
<ListView CachingStrategy="RecycleElement">
    <ListView.ItemTemplate>
        <DataTemplate>
            <ViewCell>
              ...
            </ViewCell>
        </DataTemplate>
    </ListView.ItemTemplate>
</ListView>
```

这具有相同的效果与在 C# 中; 构造函数中设置的缓存策略参数请注意，没有任何 `CachingStrategy` 属性上的 `ListView`。

子类化 ListView 中设置缓存策略

设置 `CachingStrategy` 子类化，从 XAML 的特性 `ListView` 不会生成所需的行为，因为没有任何 `CachingStrategy` 属性 `ListView`。此外，如果XAMLC已启用，就会生成以下错误消息：**没有属性，可绑定属性或找到 CachingStrategy 的事件**

此问题的解决方案是指定构造函数上子类 `ListView` 接受 `ListViewCachingStrategy` 参数并将其传递到基类：

```
public class CustomListView : ListView
{
  public CustomListView (ListViewCachingStrategy strategy) : base (strategy)
  {
  }
  ...
}
```

然后 `ListViewCachingStrategy` 枚举值可以通过使用指定从 XAML `x:Arguments` 语法：

```
<local:CustomListView>
  <x:Arguments>
    <ListViewCachingStrategy>RecycleElement</ListViewCachingStrategy>
  </x:Arguments>
</local:CustomListView>
```

提高 ListView 性能

有许多方法来提高性能的 `ListView`：

- 将绑定 `ItemsSource` 属性设置为 `IList<T>` 而不是集合 `IEnumerable<T>` 集合，因为 `IEnumerable<T>` 集合不支持随机访问。
- 使用内置的单元格 (如 `TextCell` / `SwitchCell`) 而不是 `ViewCell` 每当可以。
- 使用更少元素。例如，请考虑使用单个 `FormattedString` 而不是多个标签的标签。
- 替换 `ListView` 与 `TableView` 显示非同构的数据-即，不同类型的数据时。
- 限制的使用 `Cell.ForceUpdateSize` 方法。如果过度使用，它会降低性能。
- 在 Android 上，避免设置 `ListView` 的行分隔符可见性或颜色后它已经实例化，因为它会导致对较大性能产生负

面影响。

- 避免更改基于单元格布局 `BindingContext` 。这会导致大型的布局和初始化成本。
- 避免将深度嵌套的布局层次结构。使用 `AbsoluteLayout` 或 `Grid` 以帮助减少嵌套。
- 避免特定 `LayoutOptions` 而不 `Fill` (填充是便宜计算)。
- 避免将置于 `ListView` 内 `ScrollView` 原因如下:
  - `ListView` 实现其自己的滚动。
  - `ListView` 不会收到任何手势，因为父对象将处理 `ScrollView` 。
  - `ListView` 可以提供自定义的页眉和页脚滚动的列表中，有可能，提供的功能元素 `ScrollView` 所用的。有关详细信息请参阅页眉和页脚。
- 如果您需要在单元中提供一个非常具体的复杂设计，请考虑自定义呈现器。

`AbsoluteLayout` 有可能执行而无需单个度量值调用的布局。这样就非常强大的性能。如果 `AbsoluteLayout` 不能使用，请考虑 `RelativeLayout` 。如果使用 `RelativeLayout` ，直接传递约束将远远快于使用 API 的表达式。这是因为表达式 API 使用 JIT，并在 iOS 上的树包含解释，这是速度较慢。表达式 API 是适用于页面布局，只是要求初始布局和旋转，但在 `ListView` ，其中在滚动，过程不断地运行会损害性能。

构建自定义呈现器 `ListView` 或其单元格是一种方法减少的布局计算上滚动性能的影响。有关详细信息，请参阅自定义 ListView并自定义 ViewCell。

相关链接

- 自定义呈现器视图 (示例)
- 自定义呈现器 ViewCell (示例)
- ListViewCachingStrategy

Xamarin.Forms 映射

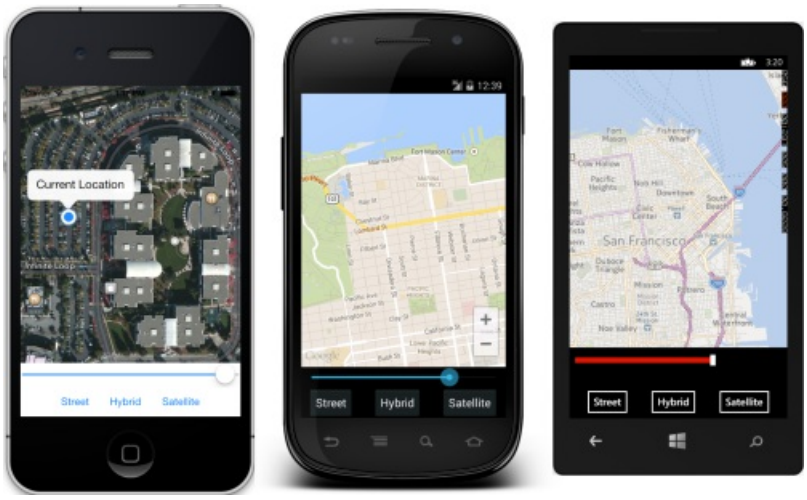
2018/11/13 • • Edit Online

Xamarin.Forms 每个平台上使用本机映射 Api。

Xamarin.Forms.Maps 每个平台上使用本机映射 Api。这对于用户，提供快速、熟悉地图体验，但意味着一些配置步骤所需遵守每个平台特定 API 要求。配置完成后，`Map` 控制的工作方式就像在常见的代码中的任何其他 Xamarin.Forms 元素。

- 映射初始化-使用 `Map` 需要在启动时附加的初始化代码。
- 平台配置-每个平台所需映射来处理一些配置。
- 在 C# 中使用映射-显示映射和固定使用 C#。
- 在 XAML 中使用映射-显示具有 XAML 的映射。

已在使用地图控件MapsSample示例，如下所示。



映射功能可以通过创建进一步增强映射自定义呈现器。

映射初始化

将地图添加到 Xamarin.Forms 应用程序时**Xamarin.Forms.Maps**是一个单独的 NuGet 包，应添加到解决方案中的每个项目。在 Android 上，这也存在依赖关系添加 Xamarin.Forms.Maps 时，会自动下载的 GooglePlayServices (另一个 NuGet)。

安装 NuGet 包后，一些初始化代码需要在每个应用程序项目中，*后* `Xamarin.Forms.Forms.Init` 方法调用。对于 iOS 使用以下代码：

```
Xamarin.FormsMaps.Init();
```

必须将相同的参数传递在 Android 上 `Forms.Init`：

```
Xamarin.FormsMaps.Init(this, bundle);
```

有关通用 Windows 平台 (UWP) 中使用以下代码：

```
Xamarin.FormsMaps.Init("INSERT_AUTHENTICATION_TOKEN_HERE");
```

为每个平台在以下文件中添加此调用：

- **iOS** -AppDelegate.cs 文件中，在 `FinishedLaunching` 方法。
- **Android** -MainActivity.cs 文件 `OnCreate` 方法。
- **UWP** -MainPage.xaml.cs 文件中，在 `MainPage` 构造函数。

已添加 NuGet 包并在每个应用程序内调用初始化方法后 `Xamarin.Forms.Maps` Api 可以使用共享项目代码的公共.NET Standard 库项目中。

平台配置

之前该映射将显示在某些平台上需要其他配置步骤。

iOS

若要访问在 iOS 上的位置服务，必须设置以下项**Info.plist**：

- iOS 11
  - `NSLocationWhenInUseUsageDescription` – 使用位置服务，当应用程序正在使用中
  - `NSLocationAlwaysAndWhenInUseUsageDescription` – 在任何时候使用位置服务
- iOS 10 及更早版本
  - `NSLocationWhenInUseUsageDescription` – 使用位置服务，当应用程序正在使用中
  - `NSLocationAlwaysUsageDescription` – 在任何时候使用位置服务

若要支持 iOS 11 和更早版本，可以包括所有三个密钥：`NSLocationWhenInUseUsageDescription`，`NSLocationAlwaysAndWhenInUseUsageDescription`，和 `NSLocationAlwaysUsageDescription`。

中的这些项的 XML 表示形式**Info.plist**如下所示。应更新 `string` 值以反映你的应用程序如何使用位置信息：

```
<key>NSLocationAlwaysUsageDescription</key>
<string>Can we use your location at all times?</string>
<key>NSLocationWhenInUseUsageDescription</key>
<string>Can we use your location when your app is being used?</string>
<key>NSLocationAlwaysAndWhenInUseUsageDescription</key>
<string>Can we use your location at all times?</string>
```

Info.plist还可以在中添加条目**源**在其他视图**Info.plist**文件：

NSLocationWhenInUseUsageDescription	String	We are using your location
NSLocationAlwaysUsageDescription	String	Can we use your location

Android

若要使用Google 地图 API v2必须在 Android 上生成 API 密钥并将其添加到你的 Android 项目。Xamarin 文档中的说明进行操作并遵照获取 Google Maps API v2 密钥。按照这些说明进行操作之后，将粘贴中的 API 密钥**properties/Androidmanifest.xml**文件（查看源和查找/更新的以下元素）：

```
<application ...>
    <meta-data android:name="com.google.android.maps.v2.API_KEY" android:value="YOUR_API_KEY" />
</application>
```

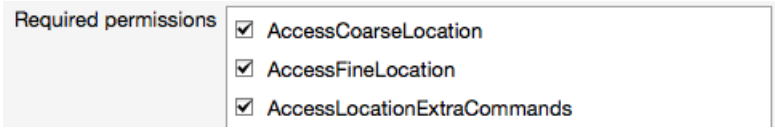
如果没有有效的 API 密钥地图控件将显示为灰色框在 Android 上。

> **NOTE**
>
> 请注意，为了使 APK 来访问 Google 地图，您必须包括 sha-1 指纹，包使用对 APK 进行签名每个密钥存储（调试和发布）的名称。例如，如果一台计算机用于调试和生成发布 APK 的另一台计算机，您应包括 sha-1 证书指纹从第一台计算机的调试密钥存储和从的发布密钥存储的 sha-1 证书指纹第二台计算机。此外，切记要编辑的密钥凭据，如果应用程序的包名称更改。请参阅获取 Google Maps API v2 密钥。

您还需要通过右键单击 Android 项目，然后选择启用适当的权限选项 > 生成 > **Android** 应用程序和勾选以下：

- `AccessCoarseLocation`
- `AccessFineLocation`
- `AccessLocationExtraCommands`
- `AccessMockLocation`
- `AccessNetworkState`
- `AccessWifiState`
- `Internet`

下面的屏幕截图显示了其中一些：



需要最后两个，因为应用程序需要网络连接才能下载地图数据。了解 Android权限若要了解详细信息。

通用 Windows 平台

若要在通用 Windows 平台上使用映射必须生成授权令牌。有关详细信息，请参阅请求一个地图身份验证密钥MSDN 上。

然后应在中指定的身份验证令牌 `FormsMaps.Init("AUTHORIZATION_TOKEN")` 方法调用中，与必应地图应用进行身份验证。

使用 Maps

请参阅MapPage.cs MobileCRM 示例以举例说明如何在代码中使用地图控件中。一个简单 `MapPage` 类看起来像此-请注意，新 `MapSpan` 创建来定位地图的视图：

```
public class MapPage : ContentPage {
    public MapPage() {
        var map = new Map(
            MapSpan.FromCenterAndRadius(
                    new Position(37,-122), Distance.FromMiles(0.3))) {
                IsShowingUser = true,
                HeightRequest = 100,
                WidthRequest = 960,
                VerticalOptions = LayoutOptions.FillAndExpand
            };
        var stack = new StackLayout { Spacing = 0 };
        stack.Children.Add(map);
        Content = stack;
    }
}
```

映射类型

此外可以通过设置更改地图内容 `MapType` 属性，以显示正则街道地图（默认值）、卫星照片或这两者的组合。

```
map.MapType == MapType.Street;
```

有效`MapType`的值为：

- 混合
- 附属
- 街道（默认值）

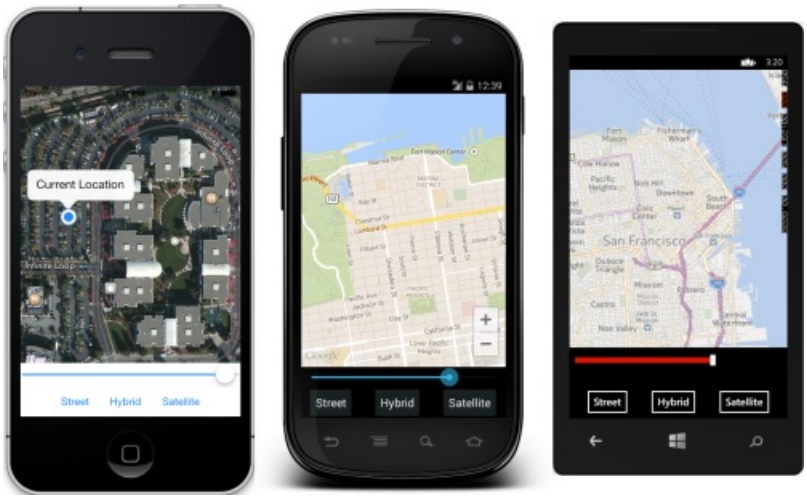
地图区域和 MapSpan

上面的代码段中所示，提供`MapSpan`map 构造函数的实例设置的初始视图（中心点和缩放级别）的映射在加载时。`MoveToRegion`然后使用 map 类上的方法来更改映射的位置或缩放级别。有两种方法来创建一个新`MapSpan`实例：

- **MapSpan.FromCenterAndRadius()** 的静态方法，用于创建从 span`Position`并指定`Distance`。
- **新 MapSpan ()** 的构造函数使用`Position`和程度的纬度和经度来显示。

若要更改地图的缩放级别而无需更改位置，请创建一个新`MapSpan`使用从当前位置`VisibleRegion.Center`地图控件的属性。一个`Slider`无法用于控制此类地图缩放（但是，缩放直接在地图控件中当前不能更新滑块的值）：

```
var slider = new Slider (1, 18, 1);
slider.ValueChanged += (sender, e) => {
    var zoomLevel = e.NewValue; // between 1 and 18
    var latlongdegrees = 360 / (Math.Pow(2, zoomLevel));
    map.MoveToRegion(new MapSpan (map.VisibleRegion.Center, latlongdegrees, latlongdegrees));
};
```



映射的 Pin

可以在代码图上标记位置`Pin`对象。

```
var position = new Position(37,-122); // Latitude, Longitude
var pin = new Pin {
            Type = PinType.Place,
            Position = position,
            Label = "custom pin",
            Address = "custom detail info"
        };
map.Pins.Add(pin);
```

`PinType`可以设置为以下值，可能会影响在 pin 呈现（具体取决于平台）的方法之一：

- 泛型

- 位置
- SavedPin
- SearchResult

使用 Xaml

此外可进行地图定位 Xaml 布局中，此代码片段中所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:maps="clr-namespace:Xamarin.Forms.Maps;assembly=Xamarin.Forms.Maps"
    x:Class="MapDemo.MapPage">
    <StackLayout VerticalOptions="StartAndExpand" Padding="30">
        <maps:Map WidthRequest="320" HeightRequest="200"
            x:Name="MyMap"
            IsShowingUser="true"
            MapType="Hybrid"
        />
    </StackLayout>
</ContentPage>
```

`MapRegion` 并 `Pins` 可以在代码中使用设置 `MyMap` 引用（或任何命名映射）。请注意，额外 `xmlns` 命名空间定义用于 Xamarin.Forms.Maps 控件的引用。

```
MyMap.MoveToRegion(
    MapSpan.FromCenterAndRadius(
        new Position(37,-122), Distance.FromMiles(1)));
```

总结

Xamarin.Forms.Maps 是必须添加到 Xamarin.Forms 解决方案中的每个项目的单独 NuGet。附加的初始化代码是必需的为 iOS、Android 和 UWP 和某些配置步骤。

一次配置映射 API 可用于呈现使用 pin 标记，只需几行代码中的映射。映射可以使用进一步增强自定义呈现器。

相关链接

- MapsSample
- 映射自定义呈现器
- Xamarin.Forms 示例

Xamarin.Forms 选取器

2018/7/13 • • Edit Online

选取器视图是一个用于选择文本项中的数据列表控件。

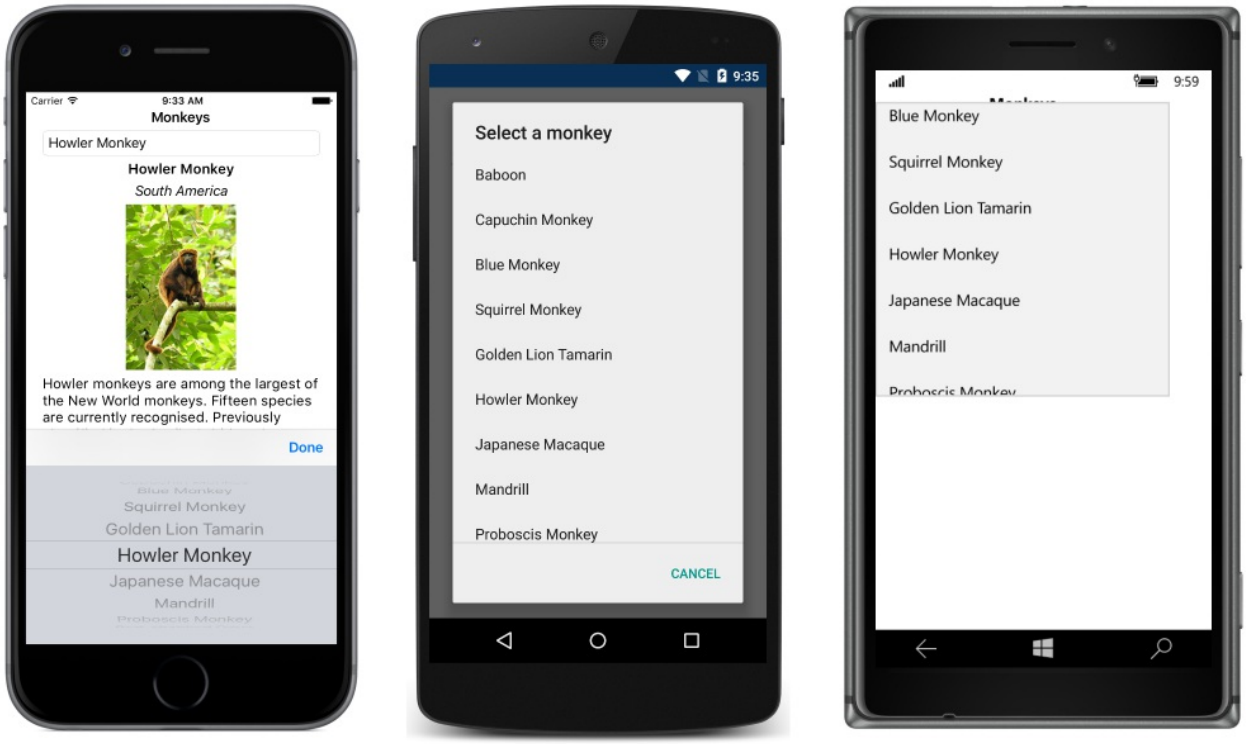
Xamarin.Forms `Picker` 显示的项，用户可以从中选择某个项的短列表。`Picker` 定义了八个属性：

- `Title` 类型的 `string`，其默认值为 `null`。
- `ItemsSource` 类型的 `IList`，源列表的项可以显示，它默认为 `null`。
- `SelectedIndex` 类型的 `int`，默认值为-1 的选定项的索引。
- `SelectedItem` 类型的 `object`，默认情况下的选定的项 `null`。
- `TextColor` 类型的 `Color`，用于显示文本，默认情况下的颜色 `Color.Default`。
- `FontAttributes` 类型的 `FontAttributes`，其默认值为 `FontAtributes.None`。
- `FontFamily` 类型的 `string`，其默认值为 `null`。
- `FontSize` 类型的 `double`，其默认值为-1.0。

所有八个属性受 `BindableProperty` 对象，这意味着它们可以设置的样式，这些属性可以是数据绑定的目标。`SelectedIndex` 并 `SelectedItem` 属性具有的默认绑定模式 `BindingMode.TwoWay`，这意味着，它们可以是数据绑定的目标在使用的应用程序模型-视图-视图模型 (MVVM)体系结构。有关设置字体属性的信息，请参阅字体。

一个 `Picker` 首次显示时不显示任何数据。相反的值及其 `Title` 属性显示为在 iOS 和 Android 平台上的占位符：

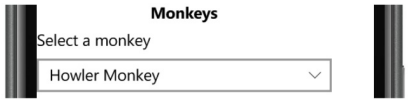
当 `Picker` 显示提升焦点，其数据和用户可以选择一项：



`Picker` 激发 `SelectedIndexChanged` 当用户选择某个项的事件。以下所选内容，所选的项显示通过 `Picker`：

iOS

Android



UWP

有两种技术用于填充 `Picker` 数据：

- 设置 `ItemsSource` 属性设置为要显示的数据。这是在 Xamarin.Forms 2.3.4 中引入的推荐的方法。有关详细信息，请参阅选取器的 ItemsSource 属性设置。
- 添加要向显示的数据 `Items` 集合。此技术已填充的原始过程 `Picker` 的数据。有关详细信息，请参阅将数据添加到选取器的项集合。

相关链接

- 选取器

设置选取器的 ItemsSource 属性

2018/7/13 • • Edit Online

选取器视图是一个用于选择文本项中的数据列表控件。本文介绍如何通过设置 ItemsSource 属性填充数据选取器以及如何响应用户的项选择。

已增强，Xamarin.Forms 2.3.4 `Picker` 视图中添加的功能要用数据填充通过设置其 `ItemsSource` 属性，以及从检索所选的项 `SelectedItem` 属性。此外，通过设置更改为选定项文本的颜色 `TextColor` 属性设置为 `Color` 。

填充数据选取器

一个 `Picker` 可以通过设置使用数据填充其 `ItemsSource` 属性设置为 `IList` 集合。集合中的每个项必须是的或派生自类型 `object` 。项可以在 XAML 中通过初始化中添加 `ItemsSource` 属性从项的数组：

```
<Picker x:Name="picker" Title="Select a monkey">
  <Picker.ItemsSource>
    <x:Array Type="{x:Type x:String}">
      <x:String>Baboon</x:String>
      <x:String>Capuchin Monkey</x:String>
      <x:String>Blue Monkey</x:String>
      <x:String>Squirrel Monkey</x:String>
      <x:String>Golden Lion Tamarin</x:String>
      <x:String>Howler Monkey</x:String>
      <x:String>Japanese Macaque</x:String>
    </x:Array>
  </Picker.ItemsSource>
</Picker>
```

NOTE

请注意，`x:Array` 元素需要 `Type` 属性，指示数组中的项的类型。

等效的 C# 代码如下所示：

```
var monkeyList = new List<string>();
monkeyList.Add("Baboon");
monkeyList.Add("Capuchin Monkey");
monkeyList.Add("Blue Monkey");
monkeyList.Add("Squirrel Monkey");
monkeyList.Add("Golden Lion Tamarin");
monkeyList.Add("Howler Monkey");
monkeyList.Add("Japanese Macaque");

var picker = new Picker { Title = "Select a monkey" };
picker.ItemsSource = monkeyList;
```

响应的项选择

一个 `Picker` 支持一次的一项选择。当用户选择某个项， `SelectedIndexChanged` 事件触发时， `SelectedIndex` 属性更新为一个整数，表示在列表中，所选的项的索引和 `SelectedItem` 属性更新为 `object` 表示所选的项。 `SelectedIndex` 属性是一个从零开始的数字，指示用户所选的项。如果未不选择任何项，这是这种情况时 `Picker` 首次创建和初始化， `SelectedIndex` 将为-1。

> **NOTE**
> 项中的选择行为 `Picker` 可以上具有平台特定的 iOS 自定义。有关详细信息，请参阅控制选取器项选择。

下面的代码示例演示如何检索 `SelectedItem` 属性值从 `Picker` 在 XAML 中：

```
<Label Text="{Binding Source={x:Reference picker}, Path=SelectedItem}" />
```

等效的 C# 代码如下所示：

```
var monkeyNameLabel = new Label();
monkeyNameLabel.SetBinding(Label.TextProperty, new Binding("SelectedItem", source: picker));
```

此外，可以是一个事件处理程序时执行 `SelectedIndexChanged` 触发事件：

```
void OnPickerSelectedIndexChanged(object sender, EventArgs e)
{
  var picker = (Picker)sender;
  int selectedIndex = picker.SelectedIndex;

  if (selectedIndex != -1)
  {
    monkeyNameLabel.Text = (string)picker.ItemsSource[selectedIndex];
  }
}
```

此方法获取 `SelectedIndex` 属性值，并使用该值来检索中的选定的项 `ItemsSource` 集合。这是功能上等效于检索中的选定的项 `SelectedItem` 属性。请注意，在每个项 `ItemsSource` 集合属于类型 `object`，，因此必须强制转换为 `string` 进行显示。

> **NOTE**
> 一个 `Picker` 可初始化以通过设置显示特定项 `SelectedIndex` 或者 `SelectedItem` 属性。但是，这些属性必须设置初始化后 `ItemsSource` 集合。

填充与使用数据绑定的数据选取器

一个 `Picker` 可以还使用填充数据通过使用数据绑定将绑定其 `ItemsSource` 属性设置为 `IList` 集合。在 XAML 中此，可以使用 `Binding` 标记扩展：

```
<Picker Title="Select a monkey" ItemsSource="{Binding Monkeys}" ItemDisplayBinding="{Binding Name}" />
```

等效的 C# 代码如下所示：

```
var picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.ItemDisplayBinding = new Binding("Name");
```

`ItemsSource` 属性数据绑定到 `Monkeys` 已连接的视图模型，它将返回属性 `IList<Monkey>` 集合。下面的代码示例演示 `Monkey` 类，该类包含四个属性：

```
public class Monkey
{
  public string Name { get; set; }
  public string Location { get; set; }
  public string Details { get; set; }
  public string ImageUrl { get; set; }
}
```

当绑定到一系列对象，`Picker` 必须告知要显示的每个对象的属性。这通过设置来实现 `ItemDisplayBinding` 到所需的属性从每个对象的属性。在上面的代码示例 `Picker` 设置为以显示每个 `Monkey.Name` 属性值。

响应的项选择

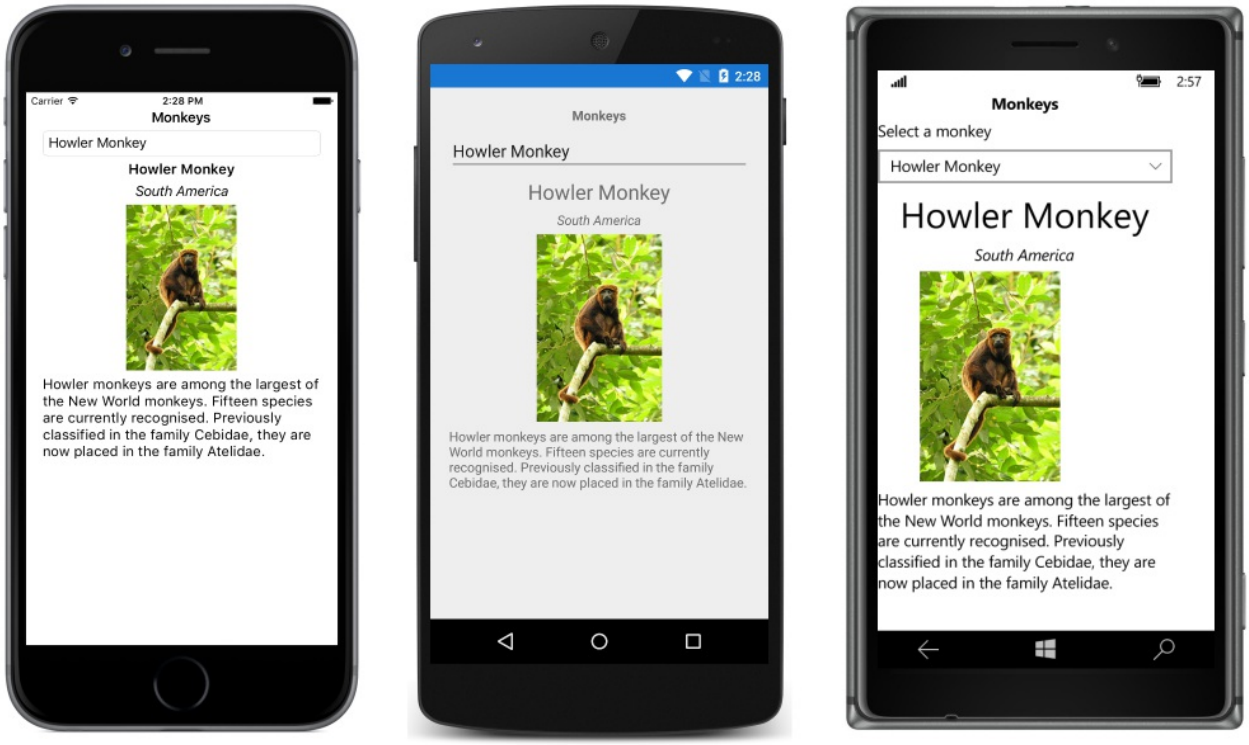
数据绑定可用于将对象设置 `SelectedItem` 属性值发生更改时：

```
<Picker Title="Select a monkey"
        ItemsSource="{Binding Monkeys}"
        ItemDisplayBinding="{Binding Name}"
        SelectedItem="{Binding SelectedMonkey}" />
<Label Text="{Binding SelectedMonkey.Name}" ... />
<Label Text="{Binding SelectedMonkey.Location}" ... />
<Image Source="{Binding SelectedMonkey.ImageUrl}" ... />
<Label Text="{Binding SelectedMonkey.Details}" ... />
```

等效的 C# 代码如下所示：

```
var picker = new Picker { Title = "Select a monkey" };
picker.SetBinding(Picker.ItemsSourceProperty, "Monkeys");
picker.SetBinding(Picker.SelectedItemProperty, "SelectedMonkey");
picker.ItemDisplayBinding = new Binding("Name");

var nameLabel = new Label { ... };
nameLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Name");

var locationLabel = new Label { ... };
locationLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Location");

var image = new Image { ... };
image.SetBinding(Image.SourceProperty, "SelectedMonkey.ImageUrl");

var detailsLabel = new Label();
detailsLabel.SetBinding(Label.TextProperty, "SelectedMonkey.Details");
```

`SelectedItem` 属性数据绑定到 `SelectedMonkey` 已连接的视图模型，这是类型的属性 `Monkey`。因此，当用户选择中的项 `Picker`，则 `SelectedMonkey` 属性将设置与所选 `Monkey` 对象。`SelectedMonkey` 通过在用户界面中显示对象数据 `Label` 并 `Image` 视图：



> **NOTE**
>
> 请注意，`SelectedItem` 并 `SelectedIndex` 属性都默认情况下支持双向绑定。

总结

`Picker` 视图是一个用于选择文本项中的数据列表控件。本文介绍了如何填充 `Picker` 通过设置数据 `ItemsSource` 属性，以及如何响应用户的项选择。在 Xamarin.Forms 中 2.3.4 引入时，此方法是用来与交互的推荐的方法 `Picker` 。

相关链接

- 选取器演示（示例）
- Monkey 应用（示例）
- 可绑定选取器（示例）
- 选取器

将数据添加到选取器的项集合

2018/7/13 • • Edit Online

选取器视图是一个用于选择文本项中的数据列表控件。本文介绍如何通过将其添加到项目集合中填充数据选取器以及如何响应用户的项选择。

填充数据选取器

在 Xamarin.Forms 2.3.4，填充的过程之前 `Picker` 的数据是添加到只读模式显示的数据 `Items` 集合，其类型 `IList<string>`. 集合中的每个项的类型必须为 `string`。项可以在 XAML 中通过初始化中添加 `Items` 具有一系列属性 `x:String` 项：

```
<Picker Title="Select a monkey">
  <Picker.Items>
    <x:String>Baboon</x:String>
    <x:String>Capuchin Monkey</x:String>
    <x:String>Blue Monkey</x:String>
    <x:String>Squirrel Monkey</x:String>
    <x:String>Golden Lion Tamarin</x:String>
    <x:String>Howler Monkey</x:String>
    <x:String>Japanese Macaque</x:String>
  </Picker.Items>
</Picker>
```

等效的 C# 代码如下所示：

```
var picker = new Picker { Title = "Select a monkey" };
picker.Items.Add("Baboon");
picker.Items.Add("Capuchin Monkey");
picker.Items.Add("Blue Monkey");
picker.Items.Add("Squirrel Monkey");
picker.Items.Add("Golden Lion Tamarin");
picker.Items.Add("Howler Monkey");
picker.Items.Add("Japanese Macaque");
```

除了添加使用数据 `Items.Add` 方法中，数据还可以插入到集合使用 `Items.Insert` 方法。

响应的项选择

一个 `Picker` 支持一次的一项选择。当用户选择某个项， `SelectedIndexChanged` 事件触发时，和 `SelectedIndex` 属性更新为一个整数，表示列表中的选定项的索引。 `SelectedIndex` 属性是一个从零开始的数字，指示用户选择的项。如果未不选择任何项，这是这种情况时 `Picker` 首次创建和初始化， `SelectedIndex` 将为-1。

> **NOTE**
>
> 项中的选择行为 `Picker` 可以上具有平台特定的 iOS 自定义。有关详细信息，请参阅控制选取器项选择。

下面的代码示例演示 `OnPickerSelectedIndexChanged` 事件处理程序方法，该方法时执行 `SelectedIndexChanged` 触发事件：

```
void OnPickerSelectedIndexChanged(object sender, EventArgs e)
{
  var picker = (Picker)sender;
  int selectedIndex = picker.SelectedIndex;

  if (selectedIndex != -1)
  {
    monkeyNameLabel.Text = picker.Items[selectedIndex];
  }
}
```

此方法获取 `SelectedIndex` 属性值，并使用该值来检索中的选定的项 `Items` 集合。因为每个项目中 `Items` 集合是 `string`，可以通过显示它们 `Label` 而无需强制转换。

> **NOTE**
> 一个 `Picker` 可初始化以通过设置显示特定项 `SelectedIndex` 属性。但是，`SelectedIndex` 属性必须设置初始化后 `Items` 集合。

总结

`Picker` 视图是一个用于选择文本项中的数据列表控件。本文介绍了如何填充 `Picker` 通过将其添加到数据 `Items` 集合，以及如何响应用户的项选择。这是使用的过程 `Picker` Xamarin.Forms 2.3.4 之前。

相关链接

- 选取器演示（示例）
- 选取器

Xamarin.Forms 滑块

2018/11/13 • • Edit Online

使用滑块选择从一系列连续值。

Xamarin.Forms `Slider` 是可以由用户选择操作的水平条 `double` 从连续范围的值。

`Slider` 定义的类型的三个属性 `double`：

- `Minimum` 是默认值为 0 的最低要求。
- `Maximum` 是默认值为 1 的最大的范围。
- `Value` 滑块的值，该值可介于 `Minimum` 和 `Maximum` 和默认值为 0。

所有三个属性受 `BindableProperty` 对象。 `Value` 属性设置了默认绑定模式 `BindingMode.TwoWay`，这意味着它很适合作为绑定源中使用的应用程序模型-视图-视图模型 (MVVM)体系结构。

> **WARNING**
> 在内部，`Slider` 可确保 `Minimum` 是小于 `Maximum` 。如果 `Minimum` 或 `Maximum` 曾经设置，以便 `Minimum` 是不小于 `Maximum` ，引发的异常。请参阅**预防措施**设置的详细信息部分下方 `Minimum` 和 `Maximum` 属性。

`Slider` 将强制 `Value` 属性，以防止之间 `Minimum` 和 `Maximum` (含) 之间。如果 `Minimum` 属性设置为值大于 `Value` 属性，`Slider` 设置 `Value` 属性设置为 `Minimum` 。同样，如果 `Maximum` 设置为值小于 `Value` ，然后 `Slider` 设置 `Value` 属性设置为 `Maximum` 。

`Slider` 定义 `ValueChanged` 时引发的事件 `Value` 更改，通过用户操作 `Slider` 或当程序集 `Value` 直接属性。一个 `ValueChanged` 时，也会激发事件 `Value` 属性强制转换时上，一段中所述。

`ValueChangedEventArgs` 对象，它附带 `ValueChanged` 事件具有两个属性，这两个类型 `double`：`OldValue` 并 `NewValue` . 次激发事件时，值 `NewValue` 等同于 `Value` 属性的 `Slider` 对象。

> **WARNING**
> 不使用的不受约束的水平布局选项 `Center` ， `Start` ，或 `End` 与 `Slider` 。在 Android 和 UWP，`Slider` 折叠到栏的长度为零，并在 iOS 上，在栏是非常短。保留默认值 `HorizontalOptions` 设置为 `Fill` ，且不使用的宽度 `Auto` 放置时 `Slider` 中 `Grid` 布局。

`Slider` 还定义了多个会影响其外观的属性：

- `MinimumTrackColor` 是在栏上左侧和右侧的滚动块的颜色。
- `MaximumTrackColor` 是条 thumb 右侧的颜色。
- `ThumbColor` 为缩略图颜色。
- `ThumbImage` 是要用于滚动块的类型的图像 `FileImageSource` 。

> **NOTE**
> `ThumbColor` 和 `ThumbImage` 属性是互相排斥。如果设置了这两个属性，`ThumbImage` 属性将优先。

滑块的基本代码和标记

SliderDemos 示例开头是功能上相同的但以不同方式实现的三个页面。第一页仅使用 C# 代码、第二个与事件处理程序在代码中，使用 XAML 和第三个是可以避免使用 XAML 文件中的数据绑定事件处理程序。

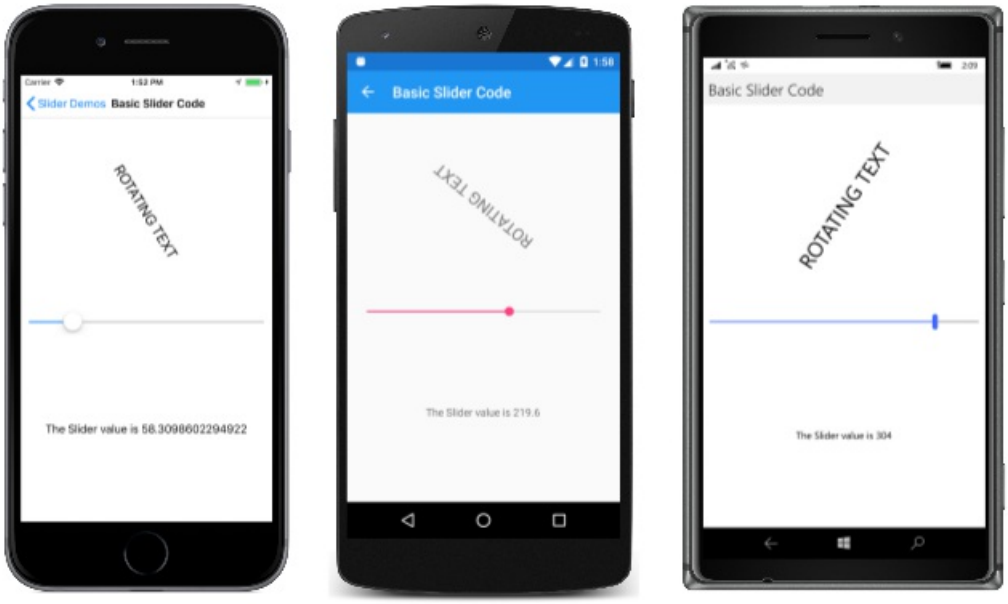
在代码中创建一个滑块

滑块的基本代码页面 **SliderDemos** 示例演示显示创建 `Slider` 并将两个 `Label` 代码中的对象：

```
public class BasicSliderCodePage : ContentPage
{
    public BasicSliderCodePage()
    {
        Label rotationLabel = new Label
        {
            Text = "ROTATING TEXT",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Label displayLabel = new Label
        {
            Text = "(uninitialized)",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Slider slider = new Slider
        {
            Maximum = 360
        };
        slider.ValueChanged += (sender, args) =>
        {
            rotationLabel.Rotation = slider.Value;
            displayLabel.Text = String.Format("The Slider value is {0}", args.NewValue);
        };

        Title = "Basic Slider Code";
        Padding = new Thickness(10, 0);
        Content = new StackLayout
        {
            Children =
            {
                rotationLabel,
                slider,
                displayLabel
            }
        };
    }
}
```

`Slider` 初始化为具有 `Maximum` 360 属性。 `ValueChanged` 处理程序 `Slider` 使用 `Value` 的属性 `slider` 对象，以设置 `Rotation` 属性的第一个 `Label`，并使用 `String.Format` 方法替换 `NewValue` 属性若要设置的事件参数 `Text` 属性的第二个 `Label`。这两种方法来获取当前值的 `Slider` 是可互换的。

下面是在 iOS、Android 和通用 Windows 平台 (UWP) 上运行的设备的程序：



第二个 `Label` 直到显示"(未初始化)"的文本 `Slider` 操作，这将导致第一个 `ValueChanged` 事件被触发。请注意显示的小数位数的数字是不同的三个平台。这些差异的平台的实现与相关 `Slider` 和更高版本中的部分中的这篇文章讨论平台实现差异。

在 XAML 中创建一个滑块

基本滑块 XAML页在功能上与相同滑块的基本代码但主要是在 XAML 中实现：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SliderDemos.BasicSliderXamlPage"
             Title="Basic Slider XAML"
             Padding="10, 0">
    <StackLayout>
        <Label x:Name="rotatingLabel"
               Text="ROTATING TEXT"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider Maximum="360"
                ValueChanged="OnSliderValueChanged" />

        <Label x:Name="displayLabel"
               Text="(uninitialized)"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

代码隐藏文件包含的处理程序 `ValueChanged` 事件：

```
public partial class BasicSliderXamlPage : ContentPage
{
    public BasicSliderXamlPage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        double value = args.NewValue;
        rotatingLabel.Rotation = value;
        displayLabel.Text = String.Format("The Slider value is {0}", value);
    }
}
```

还有可能的事件处理程序，以获取 `Slider`，在触发该事件通过 `sender` 参数。 `Value` 属性包含的当前值：

```
double value = ((Slider)sender).Value;
```

如果 `Slider` 对象提供与 XAML 文件中的名称 `x:Name` 属性（例如，"滑块"），则事件处理程序无法直接引用该对象：

```
double value = slider.Value;
```

数据绑定滑块

基本滑块绑定页显示了如何编写一个几乎等效程序，可以消除 `Value` 事件处理程序通过使用数据绑定：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SliderDemos.BasicSliderBindingsPage"
             Title="Basic Slider Bindings"
             Padding="10, 0">
    <StackLayout>
        <Label Text="ROTATING TEXT"
               Rotation="{Binding Source={x:Reference slider},
                                  Path=Value}"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />

        <Slider x:Name="slider"
                Maximum="360" />

        <Label x:Name="displayLabel"
               Text="{Binding Source={x:Reference slider},
                              Path=Value,
                              StringFormat='The Slider value is {0:F0}'}"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

`Rotation` 属性的第一个 `Label` 绑定到 `Value` 的属性 `Slider`，如是 `Text` 第二个属性 `Label` 与 `StringFormat` 规范。**基本滑块绑定**页上函数有点以不同的方式从两个前面的页：首先显示的页面时，第二个 `Label` 显示的文本字符串的值。这是使用数据绑定的优点。若要显示不带数据绑定的文本，您需要专门初始化 `Text` 的属性 `Label` 或模拟的激发 `ValueChanged` 通过从类构造函数调用的事件处理程序的事件。

预防措施

值 `Minimum` 属性必须始终为的值小于 `Maximum` 属性。以下代码片段会导致 `Slider` 引发异常：

```
// Throws an exception!
Slider slider = new Slider
{
    Minimum = 10,
    Maximum = 20
};
```

C# 编译器将生成在序列中，设置这两个属性的代码以及何时 `Minimum` 属性设置为 10，其值大于默认 `Maximum` 值为 1。您可以通过设置这种情况下避免异常 `Maximum` 属性第一个：

```
Slider slider = new Slider
{
    Maximum = 20,
    Minimum = 10
};
```

设置 `Maximum` 20 到不是问题大于默认值是 `Minimum` 值为 0。当 `Minimum` 设置，则这不会早于 `Maximum` 值为 20。

在 XAML 中存在相同的问题。它可确保订单中设置属性 `Maximum` 始终是大于 `Minimum`：

```
<Slider Maximum="20"
        Minimum="10" ... />
```

可以设置 `Minimum` 并 `Maximum` 值为负数，但仅在订单中其中 `Minimum` 是始终小于 `Maximum`：

```
<Slider Minimum="-20"
        Maximum="-10" ... />
```

`Value` 属性始终为大于或等于 `Minimum` 值且小于或等于 `Maximum` 。如果 `Value` 设置为该范围以外的值，该值将被强制为之间的范围内，但不会引发异常。例如，此代码将不引发异常：

```
Slider slider = new Slider
{
    Value = 10
};
```

相反，`Value` 属性强制转换为 `Maximum` 值为 1。

以下是上面所示的代码片段：

```
Slider slider = new Slider
{
    Maximum = 20,
    Minimum = 10
};
```

当 `Minimum` 设置为 10，则 `Value` 也设置为 10。

如果 `ValueChanged` 事件处理程序已附加时，`Value` 属性强制转换为 0，其默认值以外的内容则 `ValueChanged` 触发事件。下面是 XAML 的代码片段：

```
<Slider ValueChanged="OnSliderValueChanged"
        Maximum="20"
        Minimum="10" />
```

当 `Minimum` 设置为 10，`Value` 也设置为 10，和 `ValueChanged` 触发事件。这可能已构建页的其余部分，并在处理程序可能会尝试以引用尚未创建页面上的其他元素之前。你可能想要添加一些代码以 `ValueChanged` 处理程序，它会检查 `null` 页面上其他元素的值。或者，可以设置 `ValueChanged` 之后的事件处理程序 `Slider` 尚未初始化的值。

平台实现差异

前面所示的屏幕截图显示的值 `Slider` 具有不同数量的小数点。这与如何 `Slider` 在 Android 和 UWP 平台上实现。

Android 实现

Android 的实现 `Slider` 在 Android 上基于 `SeekBar`，并始终设置 `Max` 属性设置为 1000年。这意味着，`Slider` 在 Android 上具有仅另外 1001 离散值。如果您设置 `Slider` 能够 `Minimum` 为 0 和一个 `Maximum` 5000，则为 `Slider` 操作，`Value` 属性具有值 0、5、10、15 和等。

UWP 实现

UWP 实现 `Slider` 基于 UWP `Slider` 控件。`StepFrequency` 属性的 UWP `Slider` 设置为的差异 `Maximum` 和 `Minimum` 属性除以 10，但不能大于 1。

例如，对于默认范围为 0 到 1 `StepFrequency` 属性设置为 0.1。作为 `Slider` 操作，`Value` 属性被限制到 0，0.1、0.2、0.3、0.4、0.5、0.6，0.7，0.8、0.9，且 1.0。(这是中的最后一页中显而易见 **SliderDemos** 示例。)时之间的差异 `Maximum` 并 `Minimum` 属性为 10 或更高，则 `StepFrequency` 设置为 1，和 `Value` 属性具有整数值。

StepSlider 解决方案

更灵活 `StepSlider` 中所述第 27 章。自定义呈现器一书*使用 Xamarin.Forms 创建移动应用*。`StepSlider` 类似于 `Slider`，但添加 `Steps` 属性指定值之间的数 `Minimum` 和 `Maximum`。

滑块的颜色选择

中的两个页面最终 **SliderDemos** 示例都使用三个 `Slider` 颜色选择的实例。第一页处理代码隐藏文件中的所有交互，而第二页显示了如何使用 ViewModel 数据绑定。

处理代码隐藏文件中的滑块

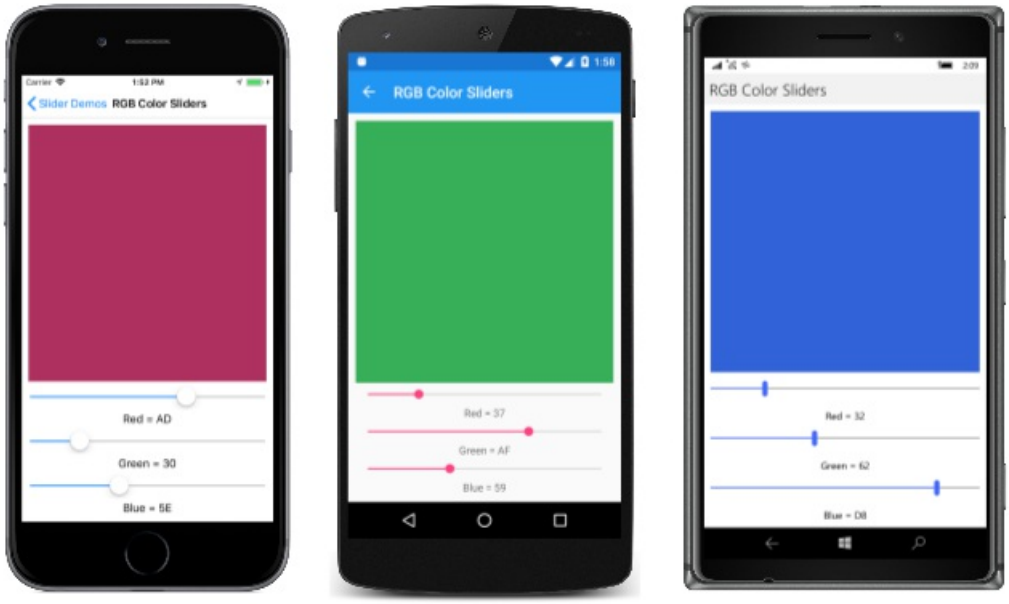
RGB 颜色滑块页上实例化 `BoxView` 以显示一种颜色，三个 `Slider` 实例选择的颜色和三个红色、绿色和蓝色组件 `Label` 用于显示这些颜色的元素值：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="SliderDemos.RgbColorSlidersPage"
             Title="RGB Color Sliders">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Slider">
                <Setter Property="Maximum" Value="255" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Margin="10">
        <BoxView x:Name="boxView"
                 Color="Black"
                 VerticalOptions="FillAndExpand" />

        <Slider x:Name="redSlider"
                ValueChanged="OnSliderValueChanged" />

        <Label x:Name="redLabel" />

        <Slider x:Name="greenSlider"
                ValueChanged="OnSliderValueChanged" />

        <Label x:Name="greenLabel" />

        <Slider x:Name="blueSlider"
                ValueChanged="OnSliderValueChanged" />

        <Label x:Name="blueLabel" />
    </StackLayout>
</ContentPage>
```

一个 `Style` 提供了所有这三个 `Slider` 一个 0 到 255 的范围的元素。 `Slider` 元素共享相同 `ValueChanged` 处理程序，这在代码隐藏文件中实现：

```
public partial class RgbColorSlidersPage : ContentPage
{
    public RgbColorSlidersPage()
    {
        InitializeComponent();
    }

    void OnSliderValueChanged(object sender, ValueChangedEventArgs args)
    {
        if (sender == redSlider)
        {
            redLabel.Text = String.Format("Red = {0:X2}", (int)args.NewValue);
        }
        else if (sender == greenSlider)
        {
            greenLabel.Text = String.Format("Green = {0:X2}", (int)args.NewValue);
        }
        else if (sender == blueSlider)
        {
            blueLabel.Text = String.Format("Blue = {0:X2}", (int)args.NewValue);
        }

        boxView.Color = Color.FromRgb((int)redSlider.Value,
                                      (int)greenSlider.Value,
                                      (int)blueSlider.Value);
    }
}
```

第一个部分集 `Text` 之一的属性 `Label` 为短文本字符串的值，该值指示实例 `Slider` 以十六进制格式。然后，所有这三个 `Slider` 访问实例创建 `Color` 从 RGB 组件的值：



绑定到 ViewModel 的滑块

HSL 颜色滑块页显示了如何使用 ViewModel 来执行用于创建计算 `Color` 色调、饱和度和亮度值中的值。像所有 Viewmodel `HSLColorViewModel` 类实现 `INotifyPropertyChanged` 接口，并且将触发 `PropertyChanged` 事件的一个属性发生更改时：

```
public class HslColorViewModel : INotifyPropertyChanged
{
    Color color;

    public event PropertyChangedEventHandler PropertyChanged;

    public double Hue
```

```
    {
        set
        {
            if (color.Hue != value)
            {
                Color = Color.FromHsla(value, color.Saturation, color.Luminosity);
            }
        }
        get
        {
            return color.Hue;
        }
    }

    public double Saturation
    {
        set
        {
            if (color.Saturation != value)
            {
                Color = Color.FromHsla(color.Hue, value, color.Luminosity);
            }
        }
        get
        {
            return color.Saturation;
        }
    }

    public double Luminosity
    {
        set
        {
            if (color.Luminosity != value)
            {
                Color = Color.FromHsla(color.Hue, color.Saturation, value);
            }
        }
        get
        {
            return color.Luminosity;
        }
    }

    public Color Color
    {
        set
        {
            if (color != value)
            {
                color = value;
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Hue"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Saturation"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Luminosity"));
                PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Color"));
            }
        }
        get
        {
            return color;
        }
    }
}
```

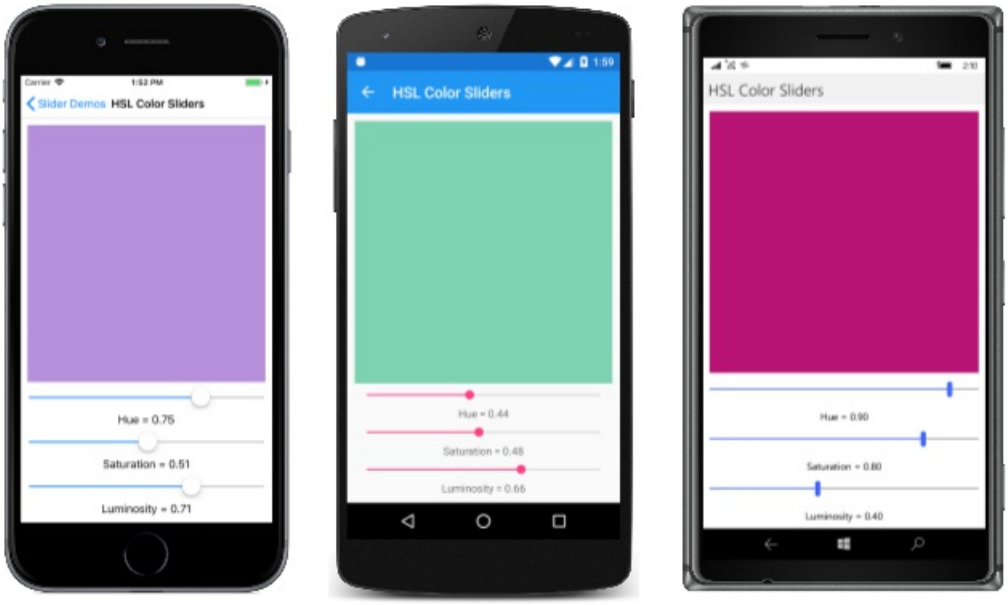
Viewmodel 并`INotifyPropertyChanged`一文中讨论了接口数据绑定。

HslColorSlidersPage.xaml文件实例化`HslColorViewModel`并将其设置为页面的`BindingContext`属性。这允许要绑

定到 ViewModel 中属性的 XAML 文件中的所有元素：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:SliderDemos"
             x:Class="SliderDemos.HslColorSlidersPage"
             Title="HSL Color Sliders">

    <ContentPage.BindingContext>
        <local:HslColorViewModel Color="Chocolate" />
    </ContentPage.BindingContext>

    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Label">
                <Setter Property="HorizontalTextAlignment" Value="Center" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>

    <StackLayout Margin="10">
        <BoxView Color="{Binding Color}"
                 VerticalOptions="FillAndExpand" />

        <Slider Value="{Binding Hue}" />
        <Label Text="{Binding Hue, StringFormat='Hue = {0:F2}'}" />

        <Slider Value="{Binding Saturation}" />
        <Label Text="{Binding Saturation, StringFormat='Saturation = {0:F2}'}" />

        <Slider Value="{Binding Luminosity}" />
        <Label Text="{Binding Luminosity, StringFormat='Luminosity = {0:F2}'}" />
    </StackLayout>
</ContentPage>
```

作为 `Slider` 操作元素，`BoxView` 和 `Label` 元素更新从 ViewModel:



`StringFormat` 组件的 `Binding` 标记扩展设置为"F2"的格式以显示两个小数位。(在文章中讨论数据绑定中的格式设置字符串字符串格式设置。)但是，UWP 版本的程序仅限于值为 0、0.1、0.2...0.9 以及 1.0。这是 UWP 的实现的直接结果 `Slider` 上文所述的部分中平台实现差异。

相关链接

- 滑块演示示例
- 滑块 API

Xamarin.Forms 分档器

2018/10/26 • • Edit Online

使用分档器从一系列值中选择的数字值。

Xamarin.Forms `Stepper` 包含带标记的两个按钮的负号和加号。这些按钮可以由用户以增量方式选择操作 `double` 从一系列值的值。

`Stepper` 定义类型的四个属性 `double`:

- `Increment` 是要更改默认值为 1，通过所选的值的量。
- `Minimum` 是默认值为 0 的最低要求。
- `Maximum` 是默认值为 100 的范围内，最大。
- `Value` 单步调试器的值，该值可介于 `Minimum` 和 `Maximum` 和默认值为 0。

所有这些属性受到 `BindableProperty` 对象。`Value` 属性设置了默认绑定模式 `BindingMode.TwoWay`，这意味着它很适合作为绑定源中使用的应用程序模型-视图-视图模型 (MVVM)体系结构。

> **WARNING**
> 在内部，`Stepper` 确保 `Minimum` 是小于 `Maximum` 。如果 `Minimum` 或 `Maximum` 曾经设置，以便 `Minimum` 是不小于 `Maximum`，引发的异常。有关详细信息设置 `Minimum` 并 `Maximum` 属性，请参阅预防措施部分。

`Stepper` 将强制 `Value` 属性，以防止之间 `Minimum` 和 `Maximum` (含) 之间。如果 `Minimum` 属性设置为值大于 `Value` 属性，`Stepper` 设置 `Value` 属性设置为 `Minimum` 。同样，如果 `Maximum` 设置为值小于 `Value`，然后 `Stepper` 设置 `Value` 属性设置为 `Maximum` 。

`Stepper` 定义 `ValueChanged` 时引发的事件 `Value` 通过用户操作的更改 `Stepper` 或当应用程序设置 `Value` 直接属性。一个 `ValueChanged` 时，也会激发事件 `Value` 属性强制转换时上，一段中所述。

`ValueChangedEventArgs` 对象，它附带 `ValueChanged` 事件具有两个属性，这两个类型 `double`: `OldValue` 和 `NewValue`. 次激发事件时，值 `NewValue` 等同于 `Value` 属性 `Stepper` 对象。

基本分档器代码和标记

StepperDemos 示例包含三个页面是功能上相同的但以不同方式实现。第一页仅使用C#代码，第二个使用 XAML 使用在代码中和第三个事件处理程序是可以避免使用 XAML 文件中的数据绑定事件处理程序。

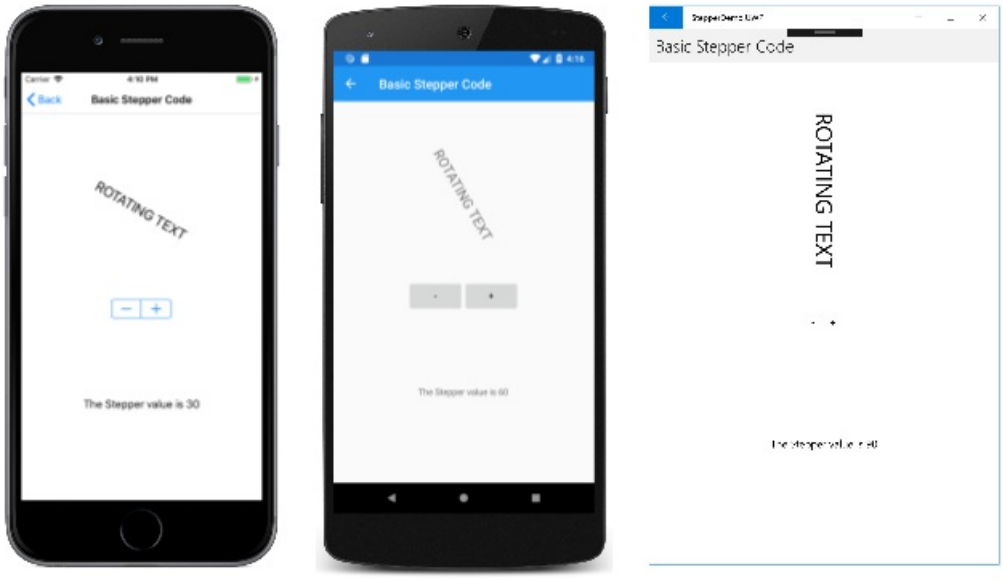
在代码中创建分档器

基本分档器代码页面 **StepperDemos** 示例演示如何创建 `Stepper` 并将两个 `Label` 代码中的对象:

```
public class BasicStepperCodePage : ContentPage
{
    public BasicStepperCodePage()
    {
        Label rotationLabel = new Label
        {
            Text = "ROTATING TEXT",
            FontSize = Device.GetNamedSize(NamedSize.Large, typeof(Label)),
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Label displayLabel = new Label
        {
            Text = "(uninitialized)",
            HorizontalOptions = LayoutOptions.Center,
            VerticalOptions = LayoutOptions.CenterAndExpand
        };

        Stepper stepper = new Stepper
        {
            Maximum = 360,
            Increment = 30,
            HorizontalOptions = LayoutOptions.Center
        };
        stepper.ValueChanged += (sender, e) =>
        {
            rotationLabel.Rotation = stepper.Value;
            displayLabel.Text = string.Format("The Stepper value is {0}", e.NewValue);
        };

        Title = "Basic Stepper Code";
        Content = new StackLayout
        {
            Margin = new Thickness(20),
            Children = { rotationLabel, stepper, displayLabel }
        };
    }
}
```

`Stepper` 初始化有 `Maximum` 360、属性和一个 `Increment` 属性为 30。操作 `Stepper` 更改所选的值以增量方式之间 `Minimum` 到 `Maximum` 值的基础 `Increment` 属性。 `ValueChanged` 处理程序 `Stepper` 使用 `Value` 属性 `stepper` 对象，以设置 `Rotation` 属性的第一个 `Label` ，并使用 `string.Format` 方法替换 `NewValue` 属性设置的事件参数 `Text` 属性第二个 `Label` 。这两种方法来获取当前值的 `Stepper` 是可互换的。

下面的屏幕截图演示基本分档器代码页：



第二个 `Label` 直到显示"(未初始化)"的文本 `Stepper` 操作，这将导致第一个 `ValueChanged` 事件若要触发。

在 XAML 中创建分档器

基本分档器 XAML页在功能上与相同**基本分档器代**码但主要是在 XAML 中实现：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StepperDemo.BasicStepperXAMLPage"
             Title="Basic Stepper XAML">
    <StackLayout Margin="20">
        <Label x:Name="_rotatingLabel"
               Text="ROTATING TEXT"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
        <Stepper Maximum="360"
                 Increment="30"
                 HorizontalOptions="Center"
                 ValueChanged="OnStepperValueChanged" />
        <Label x:Name="_displayLabel"
               Text="(uninitialized)"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

代码隐藏文件包含的处理程序 `ValueChanged` 事件：

```
public partial class BasicStepperXAMLPage : ContentPage
{
    public BasicStepperXAMLPage()
    {
        InitializeComponent();
    }

    void OnStepperValueChanged(object sender, ValueChangedEventArgs e)
    {
        double value = e.NewValue;
        _rotatingLabel.Rotation = value;
        _displayLabel.Text = string.Format("The Stepper value is {0}", value);
    }
}
```

还有可能的事件处理程序，以获取 `Stepper` ，在触发该事件通过 `sender` 参数。 `Value` 属性包含的当前值：

```
double value = ((Stepper)sender).Value;
```

如果 `Stepper` 对象提供与 XAML 文件中的名称 `x:Name` 属性（例如，"分档器"），则事件处理程序无法直接引用该对象：

```
double value = stepper.Value;
```

数据绑定分档器

基本分档器绑定页显示了如何编写一个几乎等效的应用程序，可以消除 `Value` 事件处理程序通过使用数据绑定：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="StepperDemo.BasicStepperBindingsPage"
             Title="Basic Stepper Bindings">
    <StackLayout Margin="20">
        <Label Text="ROTATING TEXT"
               Rotation="{Binding Source={x:Reference _stepper}, Path=Value}"
               FontSize="Large"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
        <Stepper x:Name="_stepper"
                 Maximum="360"
                 Increment="30"
                 HorizontalOptions="Center" />
        <Label Text="{Binding Source={x:Reference _stepper}, Path=Value, StringFormat='The Stepper value is
{0:F0}'}"
               HorizontalOptions="Center"
               VerticalOptions="CenterAndExpand" />
    </StackLayout>
</ContentPage>
```

`Rotation` 的第一个属性 `Label` 绑定到 `Value` 属性 `Stepper` 、原样 `Text` 属性的第二个 `Label` 与 `StringFormat` 规范。**基本分档器绑定**页上函数有点以不同的方式从两个前面的页：首先显示的页面时，第二个 `Label` 显示的文本字符串的值。这是使用数据绑定的优点。若要显示不带数据绑定的文本，您需要专门初始化 `Text` 的属性 `Label` 或模拟的激发 `ValueChanged` 通过从类构造函数调用的事件处理程序的事件.

预防措施

值 `Minimum` 属性必须始终为的值小于 `Maximum` 属性。以下代码片段会导致 `Stepper` 引发异常：

```
// Throws an exception!
Stepper stepper = new Stepper
{
    Minimum = 180,
    Maximum = 360
};
```

C#编译器生成的代码，用于在序列中，设置这两个属性以及何时 `Minimum` 属性设置为 180，则其值大于默认值 `Maximum` 值为 100。您可以通过设置这种情况下避免异常 `Maximum` 属性第一个：

```
Stepper stepper = new Stepper
{
    Maximum = 360,
    Minimum = 180
};
```

设置 `Maximum` 到 360 之间并不是问题由于大于默认值 `Minimum` 值为 0。当 `Minimum` 设置，则这不会早于 `Maximum` 360 的值。

在 XAML 中存在相同的问题。它可确保订单中设置属性 `Maximum` 始终是大于 `Minimum`：

```
<Stepper Maximum="360"
         Minimum="180" ... />
```

可以设置 `Minimum` 并 `Maximum` 值为负数，但仅在订单中其中 `Minimum` 是始终小于 `Maximum`：

```
<Stepper Minimum="-360"
         Maximum="-180" ... />
```

`Value` 属性始终为大于或等于 `Minimum` 值且小于或等于 `Maximum` 。如果 `Value` 设置为该范围以外的值，该值将被强制为之间的范围内，但不会引发异常。例如，此代码将不引发异常：

```
Stepper stepper = new Stepper
{
    Value = 180
};
```

相反， `Value` 属性强制转换为 `Maximum` 值为 100。

以下是上面所示的代码片段：

```
Stepper stepper = new Stepper
{
    Maximum = 360,
    Minimum = 180
};
```

当 `Minimum` 设置为 180，则 `Value` 也设置为 180。

如果 `ValueChanged` 事件处理程序已附加时， `Value` 属性强制转换为 0，其默认值以外的内容则 `ValueChanged` 触发事件。下面是 XAML 的代码片段：

```
<Stepper ValueChanged="OnStepperValueChanged"
         Maximum="360"
         Minimum="180" />
```

当 `Minimum` 设置为 180， `Value` 还将设置为 180，和 `ValueChanged` 触发事件。这可能已构建页的其余部分，并在处理程序可能会尝试以引用尚未创建页面上的其他元素之前。你可能想要添加一些代码以 `ValueChanged` 处理程序，它会检查 `null` 页面上其他元素的值。或者，可以设置 `ValueChanged` 事件处理程序之后 `Stepper` 尚未初始化的值。

相关链接

- 分档器演示示例

- 分档器 API

样式设置 Xamarin.Forms 应用

2018/7/13 • • Edit Online

使用 XAML 样式设置 Xamarin.Forms 应用的样式

通过传统上实现样式设置 Xamarin.Forms 应用 `Style` 类进行分组到一个对象，然后可以应用于多个可视元素实例的属性值的集合。这有助于减少重复性的标记，并允许应用程序外观以更轻松地更改。

使用级联样式表设置 Xamarin.Forms 应用的样式

Xamarin.Forms 支持使用级联样式表 (CSS) 样式可视元素。样式表包含的规则，每个规则由一个或多个选择器和声明块组成的列表。

使用 XAML 样式的样式设置 Xamarin.Forms 应用

2018/7/13 • • Edit Online

介绍

Xamarin.Forms 应用程序通常包含多个具有相同的外观的控件。设置每个控件的外观会重复且容易出错。相反，样式可以创建自定义控件外观的分组和可用的控件类型的设置属性。

显式样式

*显式*样式是指通过设置有选择地应用于控件及其 `Style` 属性。

隐式样式

*隐式*样式是指由相同的所有控件 `TargetType` ，而无需每个控件以引用样式。

全局样式

样式可提供全局添加到应用程序的 `ResourceDictionary` 。这有助于避免跨页或控件的样式的重复项。

样式继承

样式可以继承其他样式以减少重复和使重复使用。

动态样式

样式，不要响应属性更改和应用程序的持续时间内保持不变。但是，应用程序可以响应在运行时动态样式更改通过使用动态资源。

设备样式

Xamarin.Forms 具有六*动态*样式，称为*设备*样式，在 `Devices.Styles` 类。所有六个样式可应用于 `Label` 仅限实例。

Xamarin.Forms 样式简介

2018/7/13 • • Edit Online

样式允许可视化元素，以自定义的外观。样式为特定类型定义和包含该类型上可用的属性的值。

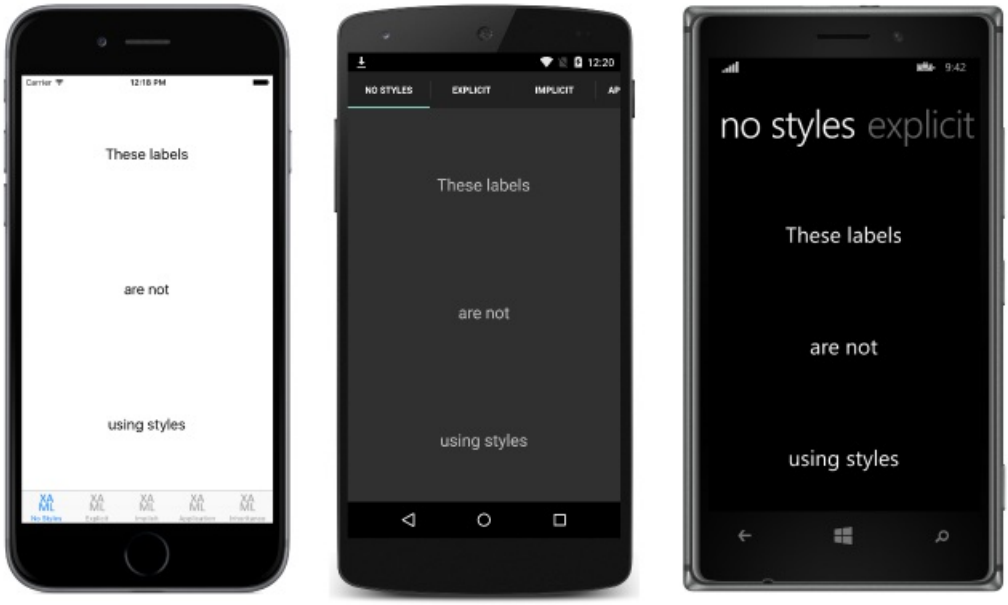
Xamarin.Forms 应用程序通常包含多个具有相同的外观的控件。例如，应用程序可能具有多个 `Label` 实例具有相同的字体选项和布局选项，如下面的 XAML 代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="Styles.NoStylesPage"
    Title="No Styles"
    Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                   HorizontalOptions="Center"
                   VerticalOptions="CenterAndExpand"
                   FontSize="Large" />
            <Label Text="are not"
                   HorizontalOptions="Center"
                   VerticalOptions="CenterAndExpand"
                   FontSize="Large" />
            <Label Text="using styles"
                   HorizontalOptions="Center"
                   VerticalOptions="CenterAndExpand"
                   FontSize="Large" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

下面的代码示例显示了在 C# 中创建的等效页：

```
public class NoStylesPageCS : ContentPage
{
    public NoStylesPageCS ()
    {
        Title = "No Styles";
        Icon = "csharp.png";
        Padding = new Thickness (0, 20, 0, 0);

        Content = new StackLayout {
            Children = {
                new Label {
                    Text = "These labels",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                },
                new Label {
                    Text = "are not",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                },
                new Label {
                    Text = "using styles",
                    HorizontalOptions = LayoutOptions.Center,
                    VerticalOptions = LayoutOptions.CenterAndExpand,
                    FontSize = Device.GetNamedSize (NamedSize.Large, typeof(Label))
                }
            }
        };
    }
}
```

每个 `Label` 实例具有相同的属性值，用于控制显示的文本的外观 `Label`。这会导致下面的屏幕截图中所示的外观：



设置每个控件的外观会重复且容易出错。相反，一种样式可以创建定义的外观，然后应用于所需的控制。

创建样式

`Style` 类进行分组到一个对象，然后可以应用于多个可视元素实例的属性值的集合。这有助于减少重复性的标记，并允许应用程序外观以更轻松地更改。

尽管主要针对基于 XAML 的应用程序设计样式，但它们也可以创建在 C# 中：

- `Style` 在 XAML 中创建的实例通常定义在 `ResourceDictionary` 分配给 `Resources` 控件的集合页上，或设置为 `Resources` 应用程序的集合。
- `Style` 在页面的类中，或者可全局访问的类通常定义在 C# 中创建的实例。

选择定义的位置 `Style` 可以使用它的影响：

- `Style` 在控件级别定义的实例只能应用到控件，及其子级。
- `Style` 在页面级别定义的实例只能应用到页，及其子级。
- `Style` 在应用程序级别定义的实例可以应用整个应用程序。

每个 `Style` 实例包含一个或多个集合 `Setter` 对象，每个 `Setter` 无 `Property` 和 `Value`。`Property` 是该样式应用到，该元素的可绑定属性的名称和 `Value` 是应用于属性的值。

每个 `Style` 实例可以是*显式*，或*隐式*：

- *显式* `Style` 通过指定定义实例 `TargetType` 和 `x:Key` 值，并通过设置目标元素 `Style` 属性设置为 `x:Key` 引用。有关详细信息*显式*样式，请参阅显式样式。
- *隐式* `Style` 仅指定定义实例 `TargetType`。`Style` 实例将然后会自动应用于该类型的所有元素。请注意该子类 `TargetType` 不会自动具有 `Style` 应用。有关详细信息*隐式*样式，请参阅隐式样式。

创建时 `Style`，则 `TargetType` 并总是必需的属性。下面的代码示例演示*显式*样式（请注意 `x:Key`）在 XAML 中创建：

```
<Style x:Key="labelStyle" TargetType="Label">
    <Setter Property="HorizontalOptions" Value="Center" />
    <Setter Property="VerticalOptions" Value="CenterAndExpand" />
    <Setter Property="FontSize" Value="Large" />
</Style>
```

若要将应用 `Style`，目标对象必须是 `VisualElement` 相匹配 `TargetType` 属性值为 `Style`，如以下 XAML 代码示例中所示：

```
<Label Text="Demonstrating an explicit style" Style="{StaticResource labelStyle}" />
```

样式的视图层次结构中较低级别优先于更高版本定义了。例如，设置 `Style`，用于设置 `Label.TextColor` 到 `Red` 在应用程序级别将被重写由设置的页级别样式 `Label.TextColor` 到 `Green`。同样，将控件级别样式将页级别样式中重写。此外，如果 `Label.TextColor` 设置直接上的控件属性，此方法将优先于任何样式。

在本部分中的文章演示并说明如何创建并应用*显式*并*隐式*样式，如何创建全局样式、样式继承，如何响应在运行时，样式更改以及如何使用 Xamarin.Forms 中包含的内置样式。

> **NOTE**
>
> **Styleid 等各是什么？**
>
> 低于 Xamarin.Forms 2.2 `StyleId` 属性用于标识用于在 UI 测试中，以及主题引擎如 Pixate 中标识的应用程序中各个元素。但是，已引入了 Xamarin.Forms 2.2 `AutomationId` 属性，已取代 `StyleId` 属性。有关详细信息，请参阅自动执行 Xamarin.Forms 测试使用 Xamarin.UITest 和 Test Cloud。

总结

Xamarin.Forms 应用程序通常包含多个具有相同的外观的控件。设置每个控件的外观会重复且容易出错。相反，样式可以创建自定义控件外观的分组和可用的控件类型的设置属性。

相关链接

- XAML 标记扩展
- 样式
- 资源库

在 Xamarin.Forms 中的显式样式

2018/7/13 • • Edit Online

显式样式是指通过设置其样式属性有选择地应用于控件。

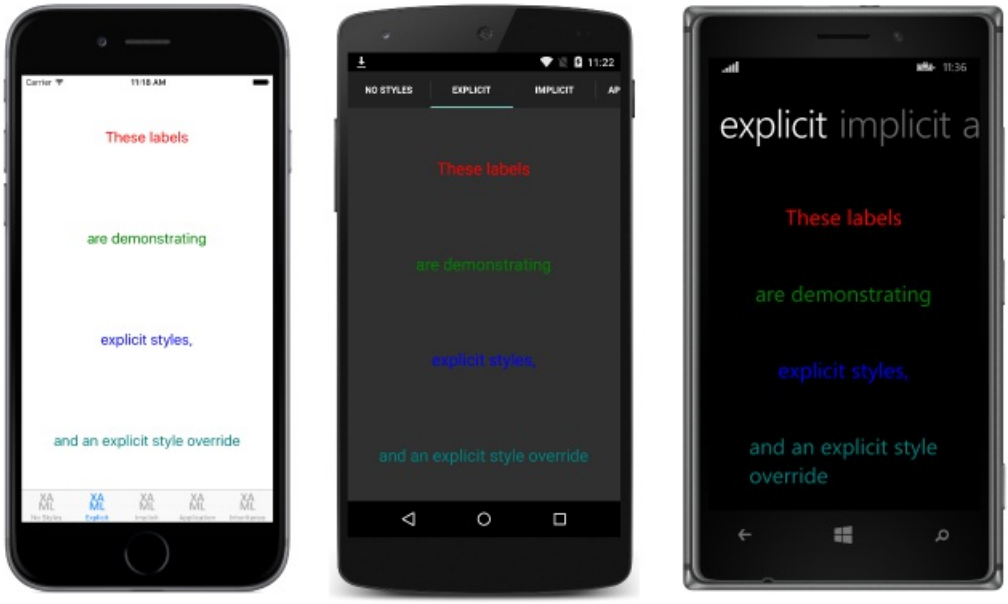
在 XAML 中创建显式样式

若要声明 `Style` 级别的页 `ResourceDictionary` 必须添加到页面，然后一个或多个 `Style` 声明可以包含在 `ResourceDictionary`。一个 `Style` 由*显式*其声明，从而 `x:Key` 属性中为其提供描述性密钥 `ResourceDictionary`。*显式*样式必须然后将应用到特定的可视元素通过设置其 `Style` 属性。

下面的代码示例演示*显式*中页面的 XAML 中声明样式 `ResourceDictionary` 并应用于页面的 `Label` 实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ExplicitStylesPage" Title="Explicit"
Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="labelRedStyle" TargetType="Label">
                <Setter Property="HorizontalOptions"
                        Value="Center" />
                <Setter Property="VerticalOptions"
                        Value="CenterAndExpand" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
            <Style x:Key="labelGreenStyle" TargetType="Label">
                ...
                <Setter Property="TextColor" Value="Green" />
            </Style>
            <Style x:Key="labelBlueStyle" TargetType="Label">
                ...
                <Setter Property="TextColor" Value="Blue" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                   Style="{StaticResource labelRedStyle}" />
            <Label Text="are demonstrating"
                   Style="{StaticResource labelGreenStyle}" />
            <Label Text="explicit styles,"
                   Style="{StaticResource labelBlueStyle}" />
            <Label Text="and an explicit style override"
                   Style="{StaticResource labelBlueStyle}"
                   TextColor="Teal" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`ResourceDictionary` 定义三个*显式*样式应用于页面的 `Label` 实例。每个 `Style` 用于显示文本以不同的颜色，同时也要设置字体大小和水平和垂直布局选项。每个 `Style` 应用到不同 `Label` 通过设置其 `Style` 属性使用 `StaticResource` 标记扩展。这会导致下面的屏幕截图中所示的外观：



此外，最终 `Label` 具有 `Style` 应用于它，但也会覆盖 `TextColor` 属性设置为不同 `Color` 值。

在控件级别创建显式样式

除了创建之外*显式*页面级别的样式，则可以也在创建这些控件级别，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ExplicitStylesPage" Title="Explicit"
Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="labelRedStyle" TargetType="Label">
                      ...
                    </Style>
                    ...
                </ResourceDictionary>
            </StackLayout.Resources>
            <Label Text="These labels" Style="{StaticResource labelRedStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在此示例中，*显式* `Style` 实例分配给 `Resources` 系列 `StackLayout` 控件。然后可以将样式应用于控件及其子项。

有关创建在应用程序的样式信息 `ResourceDictionary`，请参阅全局样式。

在 C 中创建显式样式#

`Style` 实例可以添加到页面的 `Resources` C# 中通过创建一个新的集合 `ResourceDictionary`，然后通过将添加 `Style` 实例到 `ResourceDictionary`，如中所示下面的代码示例：

```
public class ExplicitStylesPageCS : ContentPage
{
    public ExplicitStylesPageCS ()
    {
        var labelRedStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Red    }
            }
        };
        var labelGreenStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Green }
            }
        };
        var labelBlueStyle = new Style (typeof(Label)) {
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Blue }
            }
        };

        Resources = new ResourceDictionary ();
        Resources.Add ("labelRedStyle", labelRedStyle);
        Resources.Add ("labelGreenStyle", labelGreenStyle);
        Resources.Add ("labelBlueStyle", labelBlueStyle);
        ...

        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels",
                            Style = (Style)Resources ["labelRedStyle"] },
                new Label { Text = "are demonstrating",
                            Style = (Style)Resources ["labelGreenStyle"] },
                new Label { Text = "explicit styles,",
                            Style = (Style)Resources ["labelBlueStyle"] },
                new Label {    Text = "and an explicit style override",
                            Style = (Style)Resources ["labelBlueStyle"], TextColor = Color.Teal }
            }
        };
    }
}
```

构造函数定义三个*显式*应用于页面的样式 `Label` 实例。每个*显式* `Style` 添加到 `ResourceDictionary` 使用 `Add` 方法中，指定 `key` 字符串来指代 `Style` 实例。每个 `Style` 应用到不同 `Label` 通过设置其 `Style` 属性。

但是，没有使用没有优势 `ResourceDictionary` 此处。相反， `Style` 实例可以直接分配给 `Style` 所需的可视元素的属性和 `ResourceDictionary` 可删除，在下面的示例所示代码示例：

```
public class ExplicitStylesPageCS : ContentPage
{
    public ExplicitStylesPageCS ()
    {
        var labelRedStyle = new Style (typeof(Label)) {
            ...
        };
        var labelGreenStyle = new Style (typeof(Label)) {
            ...
        };
        var labelBlueStyle = new Style (typeof(Label)) {
            ...
        };
        ...
        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels", Style = labelRedStyle },
                new Label { Text = "are demonstrating", Style = labelGreenStyle },
                new Label { Text = "explicit styles,", Style = labelBlueStyle },
                new Label { Text = "and an explicit style override", Style = labelBlueStyle,
                            TextColor = Color.Teal }
            }
        };
    }
}
```

构造函数定义三个*显式*应用于页面的样式 `Label` 实例。每个 `Style` 用于显示文本以不同的颜色，同时也要设置字体大小和水平和垂直布局选项。每个 `Style` 应用到不同 `Label` 通过设置其 `Style` 属性。此外，最终 `Label` 已 `Style` 应用，但也会覆盖 `TextColor` 属性设置为不同 `Color` 值。

总结

一个 `Style` 由*显式*其声明，从而 `x:Key` 属性，，然后有选择地将它应用到控件通过设置其 `Style` 属性。

相关链接

- XAML 标记扩展
- 基本样式（示例）
- 使用样式（示例）
- ResourceDictionary
- 样式
- 资源库

在 Xamarin.Forms 中的隐式样式

2018/7/13 • • Edit Online

隐式样式是指可供所有控件的相同的目标类型，而无需每个控件以引用样式。

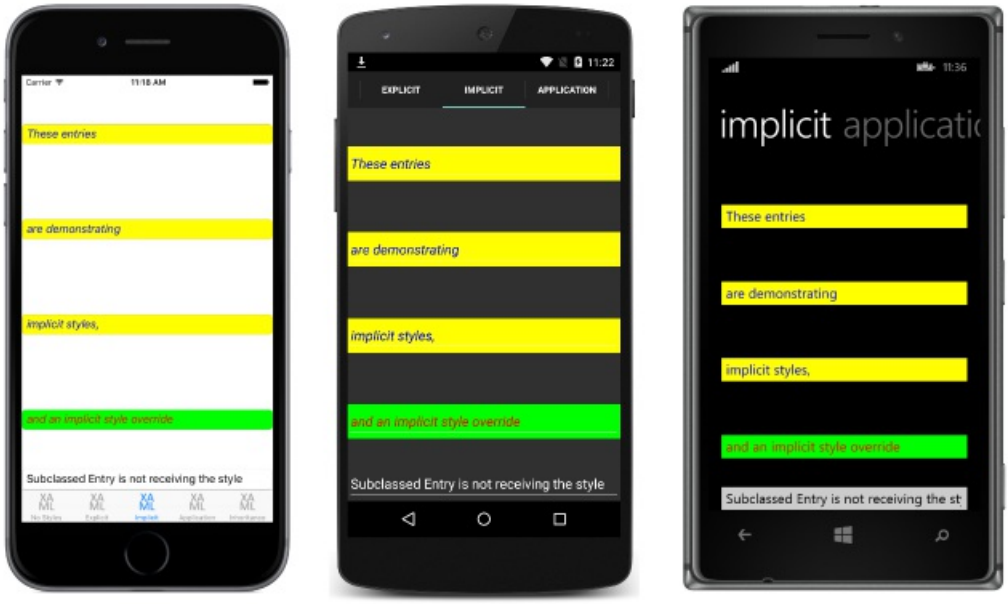
在 XAML 中创建的隐式样式

若要声明 `Style` 级别的页 `ResourceDictionary` 必须添加到页面，然后一个或多个 `Style` 声明可以包含在 `ResourceDictionary` 。一个 `Style` 由*隐式*通过不指定 `x:Key` 属性。然后将对匹配的可视元素应用样式 `TargetType` 确实如此，但不适用于元素派生自 `TargetType` 值。

下面的代码示例演示*隐式*中页面的 XAML 中声明样式 `ResourceDictionary` ，并应用于页面的 `Entry` 实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Styles;assembly=Styles"
x:Class="Styles.ImplicitStylesPage" Title="Implicit" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style TargetType="Entry">
                <Setter Property="HorizontalOptions" Value="Fill" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BackgroundColor" Value="Yellow" />
                <Setter Property="FontAttributes" Value="Italic" />
                <Setter Property="TextColor" Value="Blue" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Entry Text="These entries" />
            <Entry Text="are demonstrating" />
            <Entry Text="implicit styles," />
            <Entry Text="and an implicit style override" BackgroundColor="Lime" TextColor="Red" />
            <local:CustomEntry Text="Subclassed Entry is not receiving the style" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`ResourceDictionary` 定义单个*隐式*样式应用于页面的 `Entry` 实例。 `Style` 用于显示黄色背景，蓝色文本，同时也要设置其他外观选项。 `Style` 添加到页面的 `ResourceDictionary` 而无需指定 `x:Key` 属性。因此， `Style` 适用于所有 `Entry` 隐式实例，因为它们匹配 `TargetType` 属性 `Style` 完全。但是， `Style` 不应用于 `CustomEntry` 实例，这是子类化 `Entry` 。这会导致下面的屏幕截图中所示的外观：



此外，第四个 `Entry` 重写 `BackgroundColor` 并 `TextColor` 属性隐式样式应用到不同 `Color` 值。

在控件级别创建的隐式样式

除了创建之外*隐式*页面级别的样式，则可以也在创建这些控件级别，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" xmlns:local="clr-namespace:Styles;assembly=Styles"
x:Class="Styles.ImplicitStylesPage" Title="Implicit" Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style TargetType="Entry">
                        <Setter Property="HorizontalOptions" Value="Fill" />
                        ...
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            <Entry Text="These entries" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在此示例中，*隐式* `Style` 分配给 `Resources` 系列 `StackLayout` 控件。*隐式*样式然后可以应用于控件及其子项。

有关创建在应用程序的样式信息 `ResourceDictionary`，请参阅全局样式。

在 C 中创建的隐式样式#

`Style` 实例可以添加到页面的 `Resources` C# 中通过创建一个新的集合 `ResourceDictionary`，然后通过将添加 `Style` 实例到 `ResourceDictionary`，如中所示下面的代码示例：

```
public class ImplicitStylesPageCS : ContentPage
{
    public ImplicitStylesPageCS ()
    {
        var entryStyle = new Style (typeof(Entry)) {
            Setters = {
                ...
                new Setter { Property = Entry.TextColorProperty, Value = Color.Blue }
            }
        };

        ...
        Resources = new ResourceDictionary ();
        Resources.Add (entryStyle);

        Content = new StackLayout {
            Children = {
                new Entry { Text = "These entries" },
                new Entry { Text = "are demonstrating" },
                new Entry { Text = "implicit styles," },
                new Entry { Text = "and an implicit style override", BackgroundColor = Color.Lime, TextColor =
Color.Red },
                new CustomEntry  { Text = "Subclassed Entry is not receiving the style" }
            }
        };
    }
}
```

构造函数定义单个*隐式*应用于页面的样式 `Entry` 实例。`Style` 用于显示黄色背景，蓝色文本，同时也要设置其他外观选项。`Style` 添加到页面的 `ResourceDictionary` 而无需指定 `key` 字符串。因此，`Style` 适用于所有 `Entry` 隐式实例，因为它们匹配 `TargetType` 属性 `Style` 完全。但是，`Style` 不应用于 `CustomEntry` 实例，这是子类化 `Entry` 。

总结

*隐式*样式是指由相同的所有可视元素 `TargetType` ，而无需每个控件以引用样式。一个 `Style` 由*隐式*通过不指定 `x:Key` 属性。相反，`x:Key` 属性将自动变为的值 `TargetType` 属性。

相关链接

- XAML 标记扩展
- 基本样式（示例）
- 使用样式（示例）
- ResourceDictionary
- 样式
- 资源库

在 Xamarin.Forms 中的全局样式

2018/7/13 • • Edit Online

样式可全局添加到应用程序的资源字典。这有助于避免跨页或控件的样式的重复项。

在 XAML 中创建全局样式

默认情况下，所有从模板创建的 Xamarin.Forms 应用程序使用应用程序类，以实现 `Application` 子类。若要声明 `Style` 在应用程序级别，在应用程序的 `ResourceDictionary` 使用 XAML，默认值应用类必须替换 XAML应用类和关联的代码隐藏。有关详细信息，请参阅使用 App 类。

下面的代码示例演示 `Style` 在应用程序级别声明：

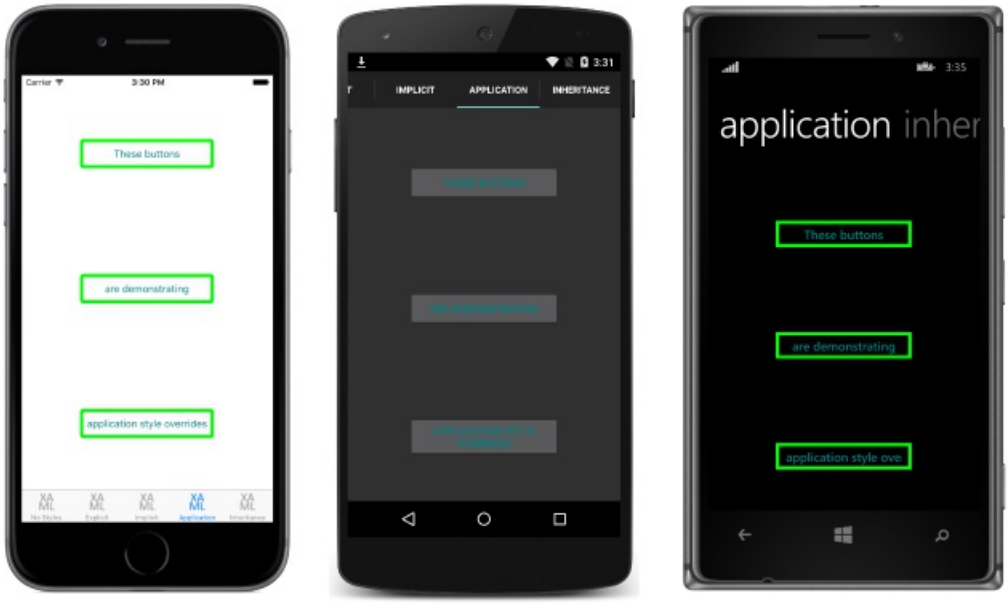
```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.App">
    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                <Setter Property="BorderColor" Value="Lime" />
                <Setter Property="BorderRadius" Value="5" />
                <Setter Property="BorderWidth" Value="5" />
                <Setter Property="WidthRequest" Value="200" />
                <Setter Property="TextColor" Value="Teal" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

这 `ResourceDictionary` 定义单个*显式*样式 `buttonStyle`，这将用于设置的外观 `Button` 实例。但是，可以全局样式*显式*或*隐式*。

下面的代码示例显示了 XAML 页应用 `buttonStyle` 向该页面的 `Button` 实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ApplicationStylesPage"
Title="Application" Icon="xaml.png">
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Button Text="These buttons" Style="{StaticResource buttonStyle}" />
            <Button Text="are demonstrating" Style="{StaticResource buttonStyle}" />
            <Button Text="application style overrides" Style="{StaticResource buttonStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

这会导致下面的屏幕截图中所示的外观：



有关创建在页面的样式信息 `ResourceDictionary`，请参阅显式样式并隐式样式。

重写样式

样式的视图层次结构中较低级别优先于更高版本定义了。例如，设置 `Style`，用于设置 `Button.TextColor` 到 `Red` 在应用程序级别将被重写由设置的页级别样式 `Button.TextColor` 到 `Green`。同样，将控件级别样式将页级别样式中重写。此外，如果 `Button.TextColor` 设置直接上的控件属性，这将优先于任何样式。在下面的代码示例演示此优先顺序：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.ApplicationStylesPage"
Title="Application" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="buttonStyle" TargetType="Button">
                ...
                <Setter Property="TextColor" Value="Red" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="buttonStyle" TargetType="Button">
                        ...
                        <Setter Property="TextColor" Value="Blue" />
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            <Button Text="These buttons" Style="{StaticResource buttonStyle}" />
            <Button Text="are demonstrating" Style="{StaticResource buttonStyle}" />
            <Button Text="application style overrides" Style="{StaticResource buttonStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

原始 `buttonStyle`，在应用程序级别定义，通过重写 `buttonStyle` 页级别定义的实例。此外，由控件级别重写页面级样式 `buttonStyle`。因此，`Button` 实例显示带有蓝色文本，如以下屏幕截图中所示：

在 C 中创建全局样式#

`Style` 可以将实例添加到应用程序的 `Resources` C# 中通过创建一个新的集合 `ResourceDictionary`，然后通过将添加 `Style` 实例到 `ResourceDictionary`，作为下面的代码示例所示：

```
public class App : Application
{
    public App ()
    {
        var buttonStyle = new Style (typeof(Button)) {
            Setters = {
                ...
                new Setter { Property = Button.TextColorProperty,    Value = Color.Teal }
            }
        };

        Resources = new ResourceDictionary ();
        Resources.Add ("buttonStyle", buttonStyle);
        ...
    }
    ...
}
```

构造函数定义单个*显式*用于将应用于样式 `Button` 整个应用程序的实例。*显式* `Style` 实例将添加到 `ResourceDictionary` 使用 `Add` 方法，指定 `key` 字符串来指代 `Style` 实例。`Style` 实例然后应用到任何应用程序中的正确类型的控件。但是，可以全局样式*显式*或*隐式*。

下面的代码示例显示了 C# 页应用 `buttonStyle` 向该页面的 `Button` 实例：

```
public class ApplicationStylesPageCS : ContentPage
{
    public ApplicationStylesPageCS ()
    {
        ...
        Content = new StackLayout {
            Children = {
                new Button { Text = "These buttons", Style = (Style)Application.Current.Resources
["buttonStyle"] },
                new Button { Text = "are demonstrating", Style = (Style)Application.Current.Resources
["buttonStyle"] },
                new Button { Text = "application styles", Style = (Style)Application.Current.Resources
["buttonStyle"]
                }
            }
        };
    }
}
```

`buttonStyle` 应用于 `Button` 实例通过设置其 `Style` 属性和控件的外观 `Button` 实例。

总结

样式可提供全局添加到应用程序的 `ResourceDictionary` 。这有助于避免跨页或控件的样式的重复项。

相关链接

- XAML 标记扩展
- 基本样式（示例）
- 使用样式（示例）
- ResourceDictionary
- 样式
- 资源库

在 Xamarin.Forms 中的样式继承

2018/7/13 • • Edit Online

样式可以继承其他样式以减少重复和使重复使用。

在 XAML 中的样式继承

通过设置执行的样式继承 `Style.BasedOn` 属性设置为现有 `Style` 。在 XAML，这通过设置来实现 `BasedOn` 属性设置为 `StaticResource` 引用以前创建的标记扩展 `Style` 。在 C# 中，这通过设置来实现 `BasedOn` 属性设置为 `Style` 实例。

继承自基样式的样式可以包括 `Setter` 实例的新属性，或使用它们来重写基准样式的样式。此外，相同类型或派生自基本样式所针对的类型的类型，必须针对继承自基样式的样式。例如，如果基本样式面向 `View` 实例，可以针对基于的基本样式的样式 `View` 实例或派生的类型 `View` 类，如 `Label` 并 `Button` 实例。

下面的代码演示*显式*设置在 XAML 页面中的样式继承：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.StyleInheritancePage"
Title="Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
                <Setter Property="HorizontalOptions"
                        Value="Center" />
                <Setter Property="VerticalOptions"
                        Value="CenterAndExpand" />
            </Style>
            <Style x:Key="labelStyle" TargetType="Label"
                   BasedOn="{StaticResource baseStyle}">
                ...
                <Setter Property="TextColor" Value="Teal" />
            </Style>
            <Style x:Key="buttonStyle" TargetType="Button"
                   BasedOn="{StaticResource baseStyle}">
                <Setter Property="BorderColor" Value="Lime" />
                ...
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="These labels"
                   Style="{StaticResource labelStyle}" />
            ...
            <Button Text="So is the button"
                    Style="{StaticResource buttonStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`baseStyle` 目标 `View` 实例，并设置 `HorizontalOptions` 并 `VerticalOptions` 属性。`baseStyle` 未直接在任何控件上设置。相反，`labelStyle` 和 `buttonStyle` 从它继承，设置其他可绑定属性值。`labelStyle` 并 `buttonStyle` 然后应用于 `Label` 实例并 `Button` 实例，通过设置其 `Style` 属性。这会导致下面的屏幕截图中所示的外观：

> **NOTE**
>
> 隐式样式可以派生自显式样式，但不能被显式样式派生的隐式样式。

遵循继承链

一种样式只能继承自同一级别或更高版本，样式中的视图层次结构。这表示：

- 应用程序级资源只能从其他应用程序级别资源继承。
- 页级别资源可以从应用程序级别资源及其他页级别资源继承。
- 控制级别的资源可以从应用程序级别资源、页级别资源和其他控制级别资源继承。

在下面的代码示例演示此继承链：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.StyleInheritancePage"
Title="Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
              ...
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <StackLayout.Resources>
                <ResourceDictionary>
                    <Style x:Key="labelStyle" TargetType="Label" BasedOn="{StaticResource baseStyle}">
                      ...
                    </Style>
                    <Style x:Key="buttonStyle" TargetType="Button" BasedOn="{StaticResource baseStyle}">
                      ...
                    </Style>
                </ResourceDictionary>
            </StackLayout.Resources>
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在此示例中，`labelStyle` 并 `buttonStyle` 是控制级别的资源，而 `baseStyle` 是一个页级别资源。然而，尽管

`labelStyle` 并 `buttonStyle` 继承 `baseStyle`，不可能 `baseStyle` 继承 `labelStyle` 或 `buttonStyle`，因为视图层次结构中其各自的位置。

在 C 中的样式继承#

等效的 C# 页上，其中 `Style` 实例直接分配给 `Style` 必选控件的属性下面的代码示例中所示：

```
public class StyleInheritancePageCS : ContentPage
{
    public StyleInheritancePageCS ()
    {
        var baseStyle = new Style (typeof(View)) {
            Setters = {
                new Setter {
                    Property = View.HorizontalOptionsProperty, Value = LayoutOptions.Center    },
                ...
            }
        };

        var labelStyle = new Style (typeof(Label)) {
            BasedOn = baseStyle,
            Setters = {
                ...
                new Setter { Property = Label.TextColorProperty, Value = Color.Teal    }
            }
        };

        var buttonStyle = new Style (typeof(Button)) {
            BasedOn = baseStyle,
            Setters = {
                new Setter { Property = Button.BorderColorProperty, Value =    Color.Lime },
                ...
            }
        };
        ...

        Content = new StackLayout {
            Children = {
                new Label { Text = "These labels", Style = labelStyle },
                ...
                new Button { Text = "So is the button", Style = buttonStyle }
            }
        };
    }
}
```

`baseStyle` 目标 `View` 实例，并设置 `HorizontalOptions` 并 `VerticalOptions` 属性。`baseStyle` 未直接在任何控件上设置。相反，`labelStyle` 和 `buttonStyle` 从它继承，设置其他可绑定属性值。`labelStyle` 并 `buttonStyle` 然后应用于 `Label` 实例并 `Button` 实例，通过设置其 `Style` 属性。

总结

样式可以继承其他样式以减少重复和使重复使用。通过设置执行的样式继承 `Style.BasedOn` 属性设置为现有 `Style` 。

相关链接

- XAML 标记扩展
- 基本样式（示例）
- 使用样式（示例）

- ResourceDictionary
- 样式
- 资源库

在 Xamarin.Forms 中的动态样式

2018/10/26 • • Edit Online

样式，不要响应属性更改和应用程序的持续时间内保持不变。例如，分配到可视元素，如果其中一个 Setter 实例修改、删除或添加新的资源库实例的一种样式后，所做的更改不会应用到可视元素。但是，应用程序可以响应在运行时动态样式更改通过使用动态资源。

`DynamicResource` 标记扩展是类似于 `StaticResource` 都使用字典键提取从值中的标记扩展 `ResourceDictionary` 。然而，尽管 `StaticResource` 执行单个字典查找，`DynamicResource` 维护字典键的链接。因此，如果替换与键关联的字典条目，更改将应用到可视元素。这样，在应用程序中进行的运行时样式更改。

下面的代码示例演示动态XAML 页面中的样式：

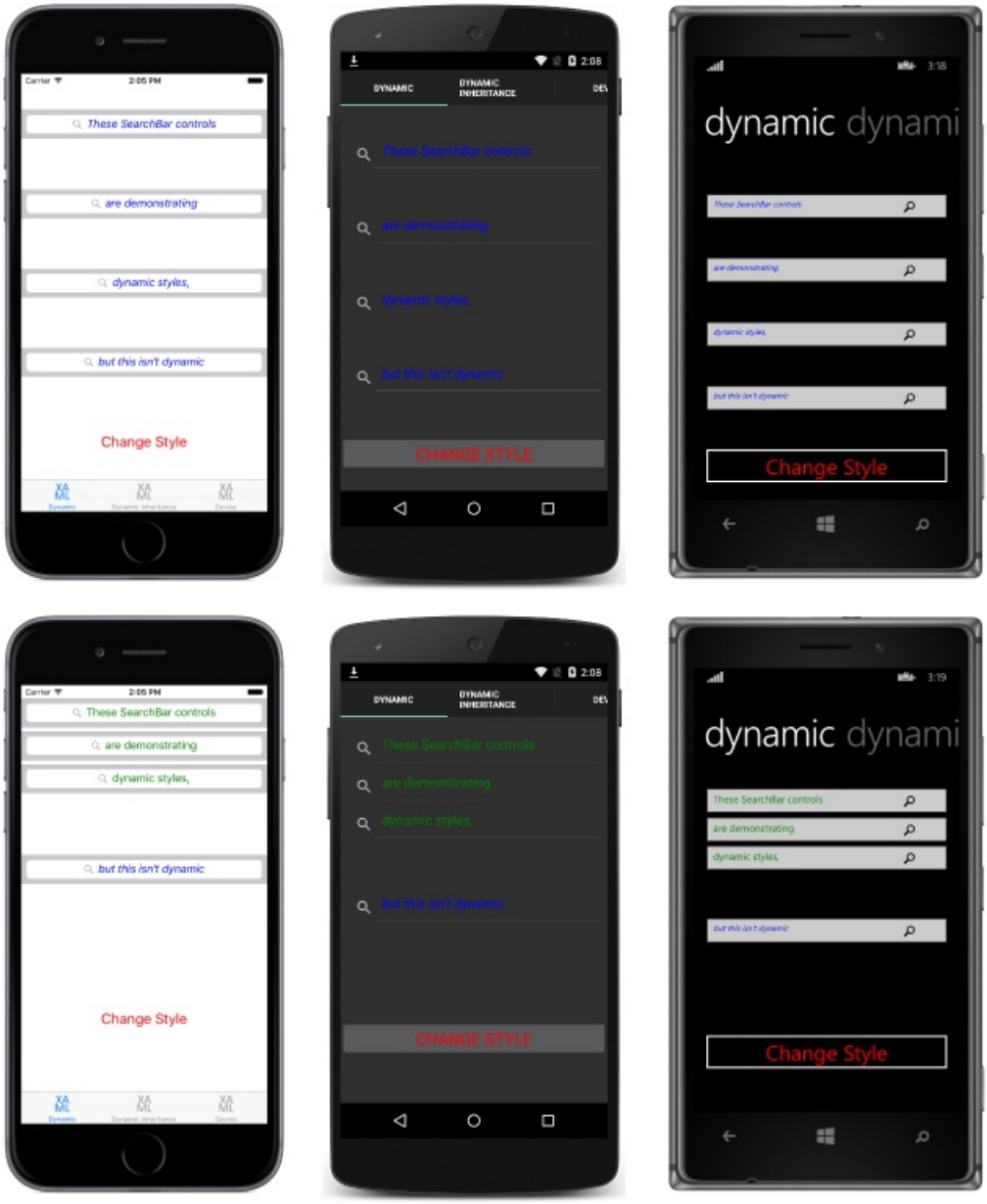
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DynamicStylesPage" Title="Dynamic"
Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
              ...
            </Style>
            <Style x:Key="blueSearchBarStyle"
                   TargetType="SearchBar"
                   BasedOn="{StaticResource baseStyle}">
              ...
            </Style>
            <Style x:Key="greenSearchBarStyle"
                   TargetType="SearchBar">
              ...
            </Style>
            ...
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <SearchBar Placeholder="These SearchBar controls"
                       Style="{DynamicResource searchBarStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`SearchBar` 实例使用 `DynamicResource` 标记扩展引用 `Style` 名为 `searchBarStyle`，未在 XAML 中定义。但是，由于 `Style` 的属性 `SearchBar` 集使用的实例 `DynamicResource`，缺失的字典键不会产生引发了异常。

相反，在代码隐藏文件中，该构造函数创建 `ResourceDictionary` 具有键的项 `searchBarStyle`，下面的代码示例中所示：

```
public partial class DynamicStylesPage : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesPage ()
    {
        InitializeComponent ();
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];
    }

    void OnButtonClicked (object sender, EventArgs e)
    {
        if (originalStyle) {
            Resources ["searchBarStyle"] = Resources ["greenSearchBarStyle"];
            originalStyle = false;
        } else {
            Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];
            originalStyle = true;
        }
    }
}
```

当`OnButtonClicked`执行事件处理程序时，`searchBarStyle`会之间切换`blueSearchBarStyle`和`greenSearchBarStyle`。这会导致下面的屏幕截图中所示的外观：



下面的代码示例演示如何在 C# 中的等效页：

```
public class DynamicStylesPageCS : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesPageCS ()
    {
        ...
        var baseStyle = new Style (typeof(View)) {
            ...
        };
        var blueSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var greenSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        ...
        var searchBar1 = new SearchBar { Placeholder = "These SearchBar controls" };
        searchBar1.SetDynamicResource (VisualElement.StyleProperty, "searchBarStyle");
        ...
        Resources = new ResourceDictionary ();
        Resources.Add ("blueSearchBarStyle", blueSearchBarStyle);
        Resources.Add ("greenSearchBarStyle", greenSearchBarStyle);
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];

        Content = new StackLayout {
            Children = { searchBar1, searchBar2, searchBar3, searchBar4,    button    }
        };
    }
    ...
}
```

在 C# 中，`SearchBar` 实例使用 `SetDynamicResource` 要引用方法 `searchBarStyle`。`OnButtonClicked` 事件处理程序代码等同于 XAML 的示例，并执行时，`searchBarStyle` 会之间切换 `blueSearchBarStyle` 和 `greenSearchBarStyle`。

动态样式继承

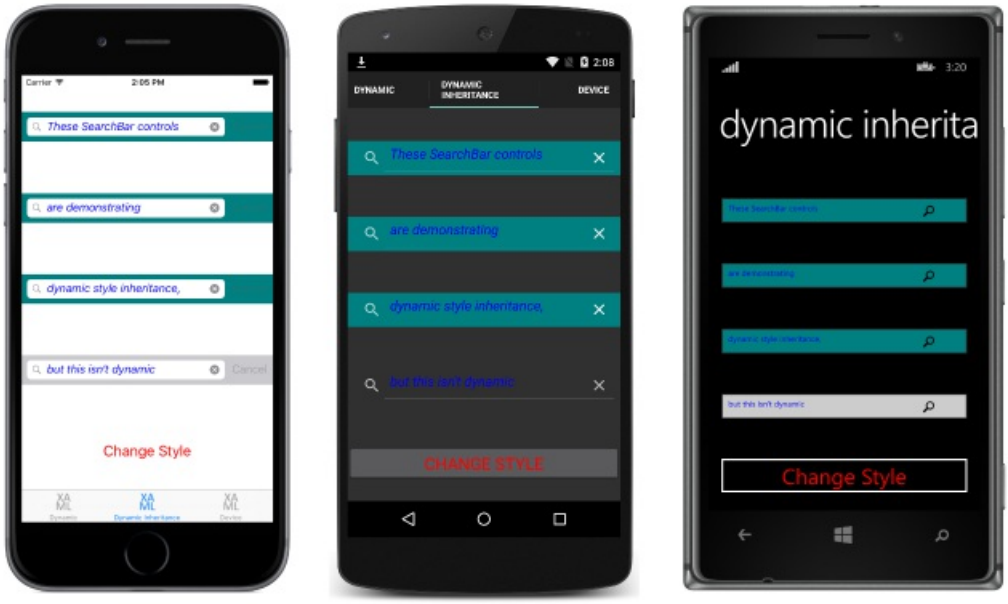
派生自动态样式的样式不能使用也能得到 `Style.BasedOn` 属性。相反，`Style` 类包括 `BaseResourceKey` 可能会动态更改属性，可以将其值设置为字典键。

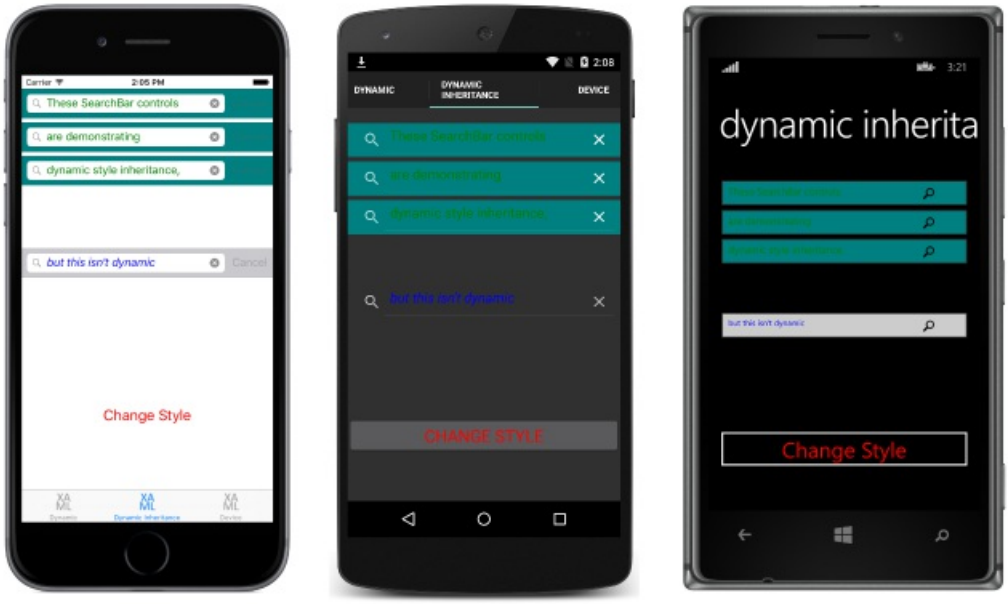
下面的代码示例演示*动态*设置在 XAML 页面中的样式继承：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DynamicStylesInheritancePage"
Title="Dynamic Inheritance" Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="baseStyle" TargetType="View">
              ...
            </Style>
            <Style x:Key="blueSearchBarStyle" TargetType="SearchBar" BasedOn="{StaticResource baseStyle}">
              ...
            </Style>
            <Style x:Key="greenSearchBarStyle" TargetType="SearchBar">
              ...
            </Style>
            <Style x:Key="tealSearchBarStyle" TargetType="SearchBar" BaseResourceKey="searchBarStyle">
              ...
            </Style>
            ...
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <SearchBar Text="These SearchBar controls" Style="{StaticResource tealSearchBarStyle}" />
            ...
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

`SearchBar` 实例使用 `StaticResource` 标记扩展引用 `Style` 名为 `tealSearchBarStyle`。这 `Style` 设置其他一些属性，并使用 `BaseResourceKey` 属性来引用 `searchBarStyle`。`DynamicResource` 标记扩展不需要，因为 `tealSearchBarStyle` 不会更改，除 `Style` 它派生。因此，`tealSearchBarStyle` 维护一个指向 `searchBarStyle` 和基准的样式更改时更改。

在代码隐藏文件中，该构造函数创建 `ResourceDictionary` 具有键的项 `searchBarStyle`、每个前面的示例演示动态样式。当 `OnButtonClicked` 执行事件处理程序时，`searchBarStyle` 会之间切换 `blueSearchBarStyle` 和 `greenSearchBarStyle`。这会导致下面的屏幕截图中所示的外观：





下面的代码示例演示如何在 C# 中的等效页：

```
public class DynamicStylesInheritancePageCS : ContentPage
{
    bool originalStyle = true;

    public DynamicStylesInheritancePageCS ()
    {
        ...
        var baseStyle = new Style (typeof(View)) {
            ...
        };
        var blueSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var greenSearchBarStyle = new Style (typeof(SearchBar)) {
            ...
        };
        var tealSearchBarStyle = new Style (typeof(SearchBar)) {
            BaseResourceKey = "searchBarStyle",
            ...
        };
        ...
        Resources = new ResourceDictionary ();
        Resources.Add ("blueSearchBarStyle", blueSearchBarStyle);
        Resources.Add ("greenSearchBarStyle", greenSearchBarStyle);
        Resources ["searchBarStyle"] = Resources ["blueSearchBarStyle"];

        Content = new StackLayout {
            Children = {
                new SearchBar { Text = "These SearchBar controls", Style = tealSearchBarStyle },
                ...
            }
        };
    }
    ...
}
```

`tealSearchBarStyle` 直接分配给 `Style` 属性 `SearchBar` 实例。这 `Style` 设置其他一些属性，并使用 `BaseResourceKey` 属性来引用 `searchBarStyle`。`SetDynamicResource` 方法不需要，此处因为 `tealSearchBarStyle` 不会更改，除 `Style` 它派生。因此，`tealSearchBarStyle` 维护一个指向 `searchBarStyle` 和基准的样式更改时更改。

总结

样式，不要响应属性更改和应用程序的持续时间内保持不变。但是，应用程序可以响应在运行时动态样式更改通过使用动态资源。此外，*动态*样式可以与派生自 `BaseResourceKey` 属性。

相关链接

- XAML 标记扩展
- 动态样式（示例）
- 使用样式（示例）
- ResourceDictionary
- 样式
- 资源库

在 Xamarin.Forms 中的设备样式

2018/7/13 • • Edit Online

Xamarin.Forms 具有六个动态样式，称为设备样式，Device.Styles 类中。

设备样式：

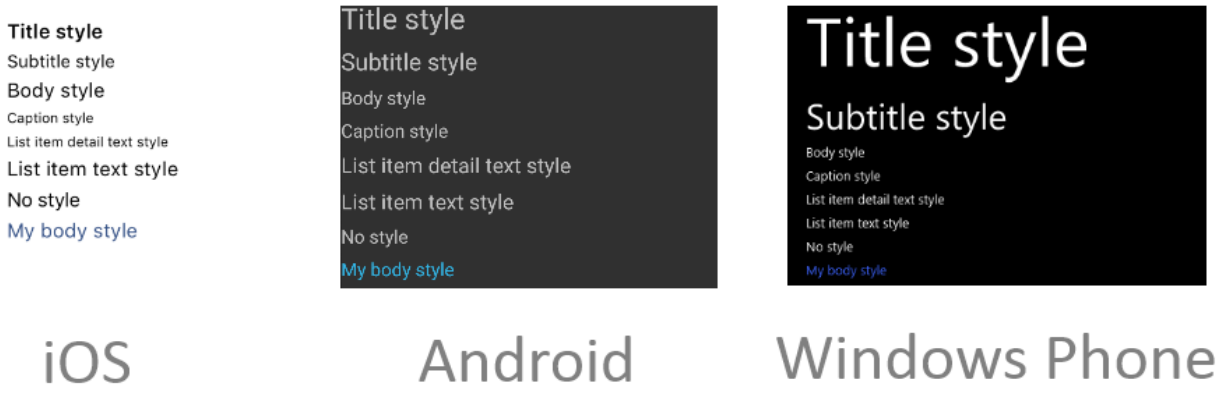
- `BodyStyle`
- `CaptionStyle`
- `ListItemDetailTextStyle`
- `ListItemTextStyle`
- `SubtitleStyle`
- `TitleStyle`

所有六个样式仅应用于 `Label` 实例。例如，`Label`，显示一个段落的主体可能会设置其 `Style` 属性设置为 `BodyStyle`。

下面的代码示例演示了如何使用设备XAML 页面中的样式：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="Styles.DeviceStylesPage" Title="Device"
Icon="xaml.png">
    <ContentPage.Resources>
        <ResourceDictionary>
            <Style x:Key="myBodyStyle" TargetType="Label"
              BaseResourceKey="BodyStyle">
                <Setter Property="TextColor" Value="Accent" />
            </Style>
        </ResourceDictionary>
    </ContentPage.Resources>
    <ContentPage.Content>
        <StackLayout Padding="0,20,0,0">
            <Label Text="Title style"
              Style="{DynamicResource TitleStyle}" />
            <Label Text="Subtitle text style"
              Style="{DynamicResource SubtitleStyle}" />
            <Label Text="Body style"
              Style="{DynamicResource BodyStyle}" />
            <Label Text="Caption style"
              Style="{DynamicResource CaptionStyle}" />
            <Label Text="List item detail text style"
              Style="{DynamicResource ListItemDetailTextStyle}" />
            <Label Text="List item text style"
              Style="{DynamicResource ListItemTextStyle}" />
            <Label Text="No style" />
            <Label Text="My body style"
              Style="{StaticResource myBodyStyle}" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

设备样式绑定到使用 `DynamicResource` 标记扩展。可以通过更改在 iOS 中看到样式的动态特性可访问性设置文本大小。外观设备样式是每个平台上不同，如以下屏幕截图中所示：



*设备*样式也从通过设置派生 `BaseResourceKey` 设备样式的键名称的属性。在上面的代码示例 `myBodyStyle` 继承自 `BodyStyle` 将带重音符的文本颜色设置。有关动态样式继承的详细信息，请参阅动态样式继承。

下面的代码示例演示如何在 C# 中的等效页：

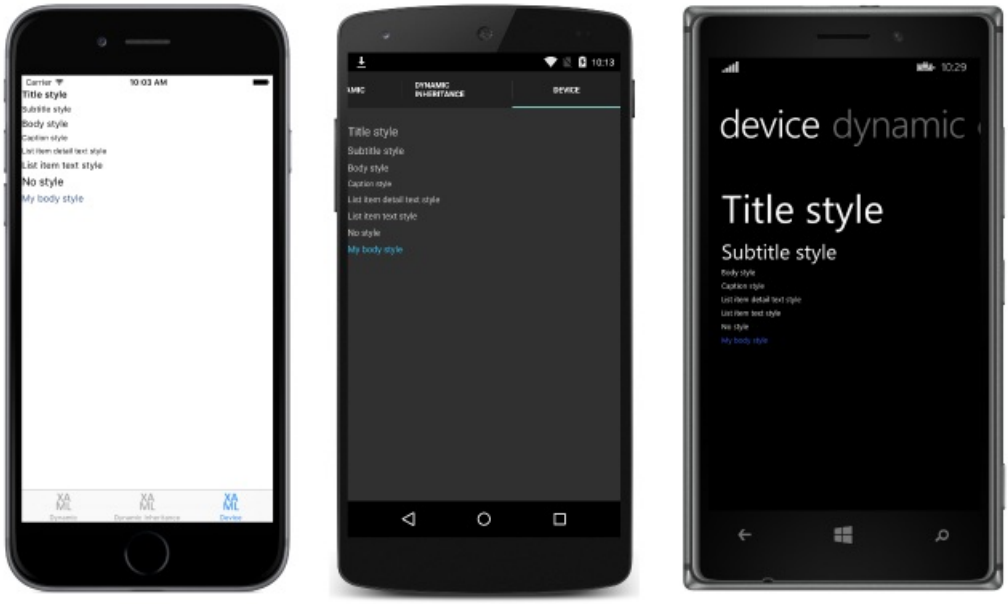
```
public class DeviceStylesPageCS : ContentPage
{
    public DeviceStylesPageCS ()
    {
        var myBodyStyle = new Style (typeof(Label)) {
            BaseResourceKey = Device.Styles.BodyStyleKey,
            Setters = {
                new Setter {
                    Property = Label.TextColorProperty,
                    Value = Color.Accent
                }
            }
        };

        Title = "Device";
        Icon = "csharp.png";
        Padding = new Thickness (0, 20, 0, 0);

        Content = new StackLayout {
            Children = {
                new Label { Text = "Title style", Style = Device.Styles.TitleStyle },
                new Label { Text = "Subtitle style", Style = Device.Styles.SubtitleStyle },
                new Label { Text = "Body style", Style = Device.Styles.BodyStyle },
                new Label { Text = "Caption style", Style = Device.Styles.CaptionStyle },
                new Label { Text = "List item detail text style",
                  Style = Device.Styles.ListItemDetailTextStyle },
                new Label { Text = "List item text style", Style = Device.Styles.ListItemTextStyle },
                new Label { Text = "No style" },
                new Label { Text = "My body style", Style = myBodyStyle }
            }
        };
    }
}
```

`Style` 每个属性 `Label` 设置为从适当的属性实例 `Devices.Styles` 类。

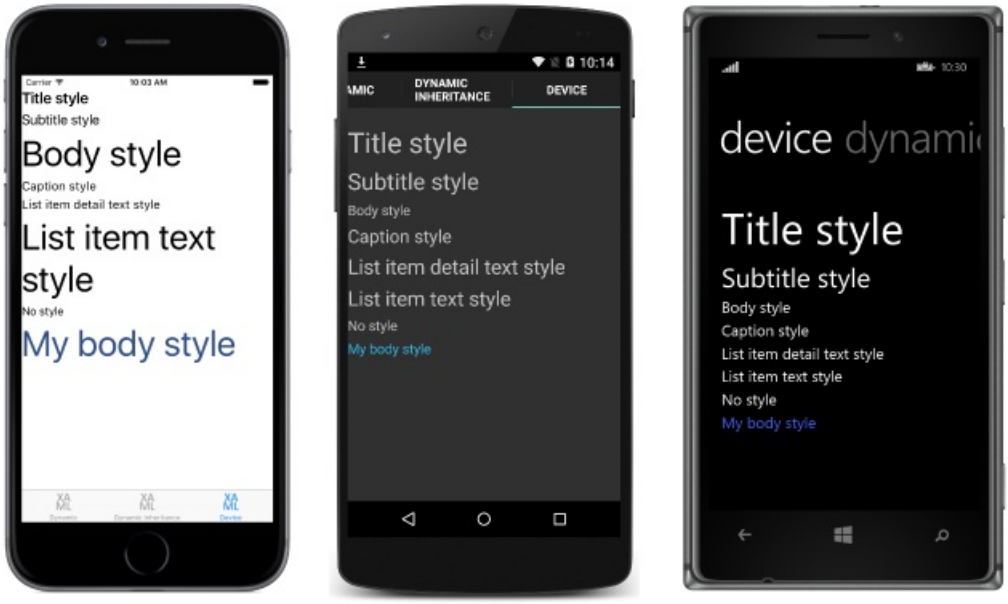
可访问性

*设备*样式尊重辅助工具首选项，因此字体大小将更改为可访问性首选项更改每个平台上。因此，若要支持辅助功能的文本，确保 *设备*样式用作应用程序中的任何文本样式的基础。

下面的屏幕截图演示了如何在每个平台上，使用最小的可访问的字体大小的设备样式：



下面的屏幕截图演示了如何在每个平台上，使用最大的可访问的字体大小的设备样式：



总结

Xamarin.Forms 具有六*动态*样式，称为*设备*样式，在 `Devices.Styles` 类。所有六个样式仅应用于 `Label` 实例。

相关链接

- 文本样式
- XAML 标记扩展
- 动态样式（示例）
- 使用样式（示例）
- Device.Styles
- ResourceDictionary
- 样式
- 资源库

使用级联样式表 (CSS) 样式设置 Xamarin.Forms 应用

2018/11/13 • • Edit Online

Xamarin.Forms 支持使用级联样式表 (CSS) 样式可视元素。

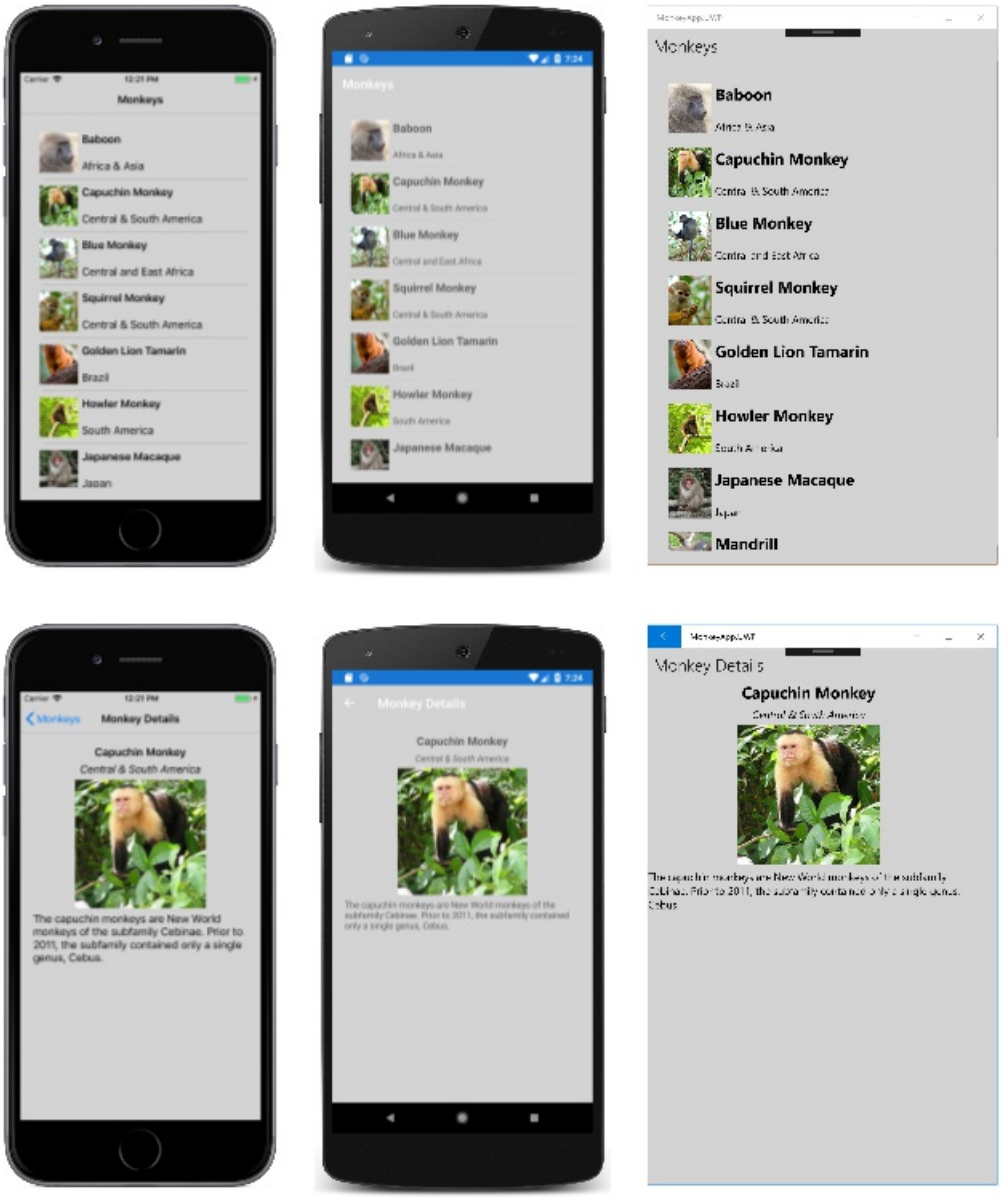
Xamarin.Forms 应用程序可以使用 CSS 来设置样式。样式表包含的规则，每个规则由一个或多个选择器和声明块组成的列表。声明块包含在括号中，与每个声明，其中包含的属性、一个冒号和一个值的声明的列表。当有多个块中的声明时，以分号作为分隔符插入。下面的代码示例显示了一些 Xamarin.Forms 符合 CSS:

```
navigationpage {
    -xf-bar-background-color: lightgray;
}

^contentpage {
    background-color: lightgray;
}

#listView {
    background-color: lightgray;
}

stacklayout {
    margin: 20;
}

.mainPageTitle {
    font-style: bold;
    font-size: medium;
}

.mainPageSubtitle {
    margin-top: 15;
}

.detailPageTitle {
    font-style: bold;
    font-size: medium;
    text-align: center;
}

.detailPageSubtitle {
    text-align: center;
    font-style: italic;
}

listview image {
    height: 60;
    width: 60;
}

stacklayout>image {
    height: 200;
    width: 200;
}
```

在 Xamarin.Forms 中，可以分析和计算在运行时，而不是编译时，CSS 样式表和样式表是在使用重新分析。

> **NOTE**
> 目前，所有可以实现的 XAML 样式的样式不能使用 CSS 执行。但是，可以使用 XAML 样式来补充 CSS Xamarin.Forms 当前不支持的属性。有关 XAML 样式的详细信息，请参阅样式设置 Xamarin.Forms 应用使用 XAML 样式。

MonkeyAppCSS示例演示了如何使用 CSS 来设置简单的应用程序的样式，并在下面的屏幕截图中所示：



使用样式表

向解决方案添加样式表的过程如下所示：

1. 将空 CSS 文件添加到.NET Standard 库项目。
2. 设置的 CSS 文件的生成操作**EmbeddedResource**。

加载样式表

有多种方法可用于加载样式表。

XAML

样式表可以加载和分析 `StyleSheet` 类，然后添加到 `ResourceDictionary`：

```
<Application ...>
    <Application.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </Application.Resources>
</Application>
```

`StyleSheet.Source` 属性为相对于封闭的 XAML 文件的位置或相对于项目根的 URI 指定的样式表，如果 URI 开头 `/`。

> **WARNING**
> CSS 文件将无法加载如果未设置为其生成操作**EmbeddedResource**。

或者，可以加载和分析与样式表 `StyleSheet` 类，然后再添加到 `ResourceDictionary`，也可由内联在 `CDATA` 部分：

```
<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet>
            <![CDATA[
            ^contentpage {
                background-color: lightgray;
            }
            ]]>
        </StyleSheet>
    </ContentPage.Resources>
    ...
</ContentPage>
```

有关资源字典的详细信息，请参阅资源字典。

C#

在C#，可以作为嵌入资源加载样式表，并将其添加到 `ResourceDictionary`：

```
public partial class MyPage : ContentPage
{
    public MyPage()
    {
        InitializeComponent();

        this.Resources.Add(StyleSheet.FromAssemblyResource(
            IntrospectionExtensions.GetTypeInfo(typeof(MyPage)).Assembly,
            "MyProject.Assets.styles.css"));
    }
}
```

第一个参数 `StyleSheet.FromAssemblyResource` 方法是包含样式表的程序集，同时第二个参数是 `string` 表示的资源标识符。可以从获取的资源标识符属性窗口选择 CSS 文件。

或者，从加载样式表 `StringReader` 并添加到 `ResourceDictionary`：

```
public partial class MyPage : ContentPage
{
    public MyPage()
    {
        InitializeComponent();

        using (var reader = new StringReader("^contentpage { background-color: lightgray; }"))
        {
            this.Resources.Add(StyleSheet.FromReader(reader));
        }
    }
}
```

参数 `StyleSheet.FromReader` 方法是 `TextReader`，它具有读取样式表。

选择元素并将属性应用

CSS 使用选择器来确定要为目标的元素。按定义顺序连续，应用具有匹配的选择器的样式。始终最后应用在特定项目上定义的样式。有关受支持的选择器的详细信息，请参阅选择器引用。

CSS 使用属性来设置所选的元素的样式。每个属性具有一系列可能的值，并且某些属性会影响任何类型的元素，而其他人将应用于组的元素。有关支持的属性的详细信息，请参阅属性引用。

选择按类型的元素

可视化树中的元素可以选择按类型对不区分大小写 `element` 选择器：

```
stacklayout {
    margin: 20;
}
```

此选择器标识任何 `StackLayout` 的使用样式表，并将它们的边距设置为统一粗细，即 20 页上的元素。

> **NOTE**
> `element` 选择器不会确定指定类型的类。

由基类选择元素

可视化树中的元素可以选择通过使用不区分大小写的基类 `^base` 选择器：

```
^contentpage {
    background-color: lightgray;
}
```

此选择器标识任何 `ContentPage` 元素使用的样式表，并设置其背景颜色 `lightgray`。

> **NOTE**
> `^base` 选择器特定于 Xamarin.Forms 中，并且不是 CSS 规范的一部分。

按名称选择元素

可视化树中的单个元素可以选择使用区分大小写 `#id` 选择器：

```
#listView {
    background-color: lightgray;
}
```

此选择器标识的元素的 `StyleId` 属性设置为 `listView`。但是，如果 `StyleId` 属性未设置，则选择器将回退到使用 `x:Name` 的元素。因此，在下面的 XAML 示例中，`#listView` 选择器将标识 `ListView` 其 `x:Name` 属性设置为 `listView`，并将背景色设置为 `lightgray`。

```
<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </ContentPage.Resources>
    <StackLayout>
        <ListView x:Name="listView" ...>
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

选择具有特定的类属性的元素

具有特定的类属性的元素可以选择使用区分大小写 `.class` 选择器：

```
.detailPageTitle {
    font-style: bold;
    font-size: medium;
    text-align: center;
}

.detailPageSubtitle {
    text-align: center;
    font-style: italic;
}
```

CSS 类可以通过设置分配给 XAML 元素 `StyleClass` 的元素的 CSS 类名称的属性。因此，在下面的 XAML 示例中，定义了样式通过 `.detailPageTitle` 分配给第一个类 `Label`，而定义的样式 `.detailPageSubtitle` 类都被分配给第二个 `Label`。

```
<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </ContentPage.Resources>
    <ScrollView>
        <StackLayout>
            <Label ... StyleClass="detailPageTitle" />
            <Label ... StyleClass="detailPageSubtitle"/>
            ...
        </StackLayout>
    </ScrollView>
</ContentPage>
```

选择子元素

可视化树中的子元素可以选择使用不区分大小写 `element element` 选择器：

```
listview image {
    height: 60;
    width: 60;
}
```

此选择器标识任何 `Image` 元素的子级 `ListView` 元素，并将其高度和宽度设置为 60。因此，在下面的 XAML 示例中，`listview image` 选择器将标识 `Image` 是的子级 `ListView` ，并将其高度和宽度设置为 60。

```
<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </ContentPage.Resources>
    <StackLayout>
        <ListView ...>
            <ListView.ItemTemplate>
                <DataTemplate>
                    <ViewCell>
                        <Grid>
                            ...
                            <Image ... />
                            ...
                        </Grid>
                    </ViewCell>
                </DataTemplate>
            </ListView.ItemTemplate>
        </ListView>
    </StackLayout>
</ContentPage>
```

> **NOTE**
>
> `element element` 选择器不需要子元素是_直接_父-子元素的子级可能有一个不同的父级。提供的该祖先是指定的第一个元素，则发生所选内容。

选择直接子元素

直接在可视化树中的子元素可以选择使用不区分大小写 `element>element` 选择器：

```
stacklayout>image {
    height: 200;
    width: 200;
}
```

此选择器标识任何 `Image` 元素的直接子级 `StackLayout` 元素，并将其高度和宽度设置为 200。因此，在下面的 XAML 示例中，`stacklayout>image` 选择器将标识 `Image` 是的直接子级 `StackLayout` ，并将其高度和宽度设置为 200。

```
<ContentPage ...>
    <ContentPage.Resources>
        <StyleSheet Source="/Assets/styles.css" />
    </ContentPage.Resources>
    <ScrollView>
        <StackLayout>
            ...
            <Image ... />
            ...
        </StackLayout>
    </ScrollView>
</ContentPage>
```

NOTE

`element>element` 选择器要求的子元素是_直接_的父级的子级。

选择器引用

Xamarin.Forms 支持以下 CSS 选择器：

选择器	示例	描述
`.class`	`.header`	选择所有元素与 `StyleClass` 包含 header 属性。请注意，此选择器区分大小写。
`#id`	`#email`	选择所有元素具有 `StyleId` 设置为 `email` 。如果 `StyleId` 未设置，则回退到 `x:Name` 。使用 XAML 时，`x:Name` 最好通过 `StyleId` 。请注意，此选择器区分大小写。
`*`	`*`	选择所有元素。
`element`	`label`	选择所有元素类型的 `Label` ，但不是嵌套类。请注意，此选择器不区分大小写。
`^base`	`^contentpage`	选择所有元素具有 `ContentPage` 作为基类，包括 `ContentPage` 本身。请注意，此选择器是不区分大小写，不是 CSS 规范的一部分。
`element,element`	`label,button`	选择所有 `Button` 元素和所有 `Label` 元素。请注意，此选择器不区分大小写。
`element element`	`stacklayout label`	选择所有 `Label` 内的元素 `StackLayout` 。请注意，此选择器不区分大小写。
`element>element`	`stacklayout>label`	选择所有 `Label` 元素与 `StackLayout` 作为直接父级。请注意，此选择器不区分大小写。

选择器	示例	描述
`element+element`	`label+entry`	选择所有`Entry`后的元素直接`Label`。请注意，此选择器不区分大小写。
`element~element`	`label~entry`	选择所有`Entry`元素前面`Label`。请注意，此选择器不区分大小写。

按定义顺序连续，应用具有匹配的选择器的样式。始终最后应用在特定项目上定义的样式。

> **TIP**
> 选择器可以组合没有限制，如`StackLayout>ContentView>label.email`。

以下选择器是当前不受支持：

- `[attribute]`
- `@media` 和 `@supports`
- `:` 和 `::`

> **NOTE**
> 特异性和特异性替代是不受支持。

属性参考

通过 Xamarin.Forms 支持以下 CSS 属性（在值列中，类型为_斜体_，而字符串文本是`gray`）：

属性	适用对象	值	示例
`align-content`	`FlexLayout`	`stretch` | `center` | `start` | `end` | `spacebetween` | `spacearound` | `spaceevenly` | `flex-start` | `flex-end` | `space-between` | `space-around` | `initial`	`align-content: space-between;`
`align-items`	`FlexLayout`	`stretch` | `center` | `start` | `end` | `flex-start` | `flex-end` | `initial`	`align-items: flex-start;`
`align-self`	`VisualElement`	`auto` | `stretch` | `center` | `start` | `end` | `flex-start` | `flex-end` | `initial`	`align-self: flex-end;`
`background-color`	`VisualElement`	*颜色* | `initial`	`background-color: springgreen;`

属性	适用对象	值	示例
`background-image`	`Page`	*字符串* | `initial`	`background-image: bg.png;`
`border-color`	`Button`, `Frame`	*颜色* | `initial`	`border-color: #9acd32;`
`border-radius`	`BoxView`	*双精度* | `initial`	`border-radius: 10;`
`border-width`	`Button`	*双精度* | `initial`	`border-width: .5;`
`color`	`ActivityIndicator`, `BoxView`, `Button`, `DatePicker`, `Editor`, `Entry`, `Label`, `Picker`, `ProgressBar`, `SearchBar`, `Switch`, `TimePicker`	*颜色* | `initial`	`color: rgba(255, 0, 0, 0.3);`
`column-gap`	`Grid`	*双精度* | `initial`	`column-gap: 9;`
`direction`	`VisualElement`	`ltr` | `rtl` | `inherit` | `initial`	`direction: rtl;`
`flex-direction`	`FlexLayout`	`column` | `columnreverse` | `row` | `rowreverse` | `row-reverse` | `column-reverse` | `initial`	`flex-direction: column-reverse;`
`flex-basis`	`VisualElement`	*float* | `auto` | `initial`。此外，使用指定的百分比在范围 0%到 100% `%` 登录。	`flex-basis: 25%;`
`flex-grow`	`VisualElement`	*Float* | `initial`	`flex-grow: 1.5;`
`flex-shrink`	`VisualElement`	*Float* | `initial`	`flex-shrink: 1;`
`flex-wrap`	`VisualElement`	`nowrap` | `wrap` | `reverse` | `wrap-reverse` | `initial`	`flex-wrap: wrap-reverse;`
`font-family`	`Button`, `DatePicker`, `Editor`, `Entry`, `Label`, `Picker`, `SearchBar`, `TimePicker`, `Span`	*字符串* | `initial`	`font-family: Consolas;`
`font-size`	`Button`, `DatePicker`, `Editor`, `Entry`, `Label`, `Picker`, `SearchBar`, `TimePicker`, `Span`	*双精度*| *namedsize* | `initial`	`font-size: 12;`

属性	适用对象	值	示例
`font-style`	`Button`, `DatePicker`, `Editor`, `Entry`, `Label`, `Picker`, `SearchBar`, `TimePicker`, `Span`	`bold` | `italic` | `initial`	`font-style: bold;`
`height`	`VisualElement`	*双精度* | `initial`	`min-height: 250;`
`justify-content`	`FlexLayout`	`start` | `center` | `end` | `spacebetween` | `spacearound` | `spaceevenly` | `flex-start` | `flex-end` | `space-between` | `space-around` | `initial`	`justify-content: flex-end;`
`line-height`	`Label`, `Span`	*双精度* | `initial`	`line-height: 1.8;`
`margin`	`View`	*粗细* | `initial`	`margin: 6 12;`
`margin-left`	`View`	*粗细* | `initial`	`margin-left: 3;`
`margin-top`	`View`	*粗细* | `initial`	`margin-top: 2;`
`margin-right`	`View`	*粗细* | `initial`	`margin-right: 1;`
`margin-bottom`	`View`	*粗细* | `initial`	`margin-bottom: 6;`
`max-lines`	`Label`	*Int* | `initial`	`max-lines: 2;`
`min-height`	`VisualElement`	*双精度* | `initial`	`min-height: 50;`
`min-width`	`VisualElement`	*双精度* | `initial`	`min-width: 112;`
`opacity`	`VisualElement`	*双精度* | `initial`	`opacity: .3;`
`order`	`VisualElement`	*Int* | `initial`	`order: -1;`
`padding`	`Layout`, `Page`	*粗细* | `initial`	`padding: 6 12 12;`
`padding-left`	`Layout`, `Page`	*双精度* | `initial`	`padding-left: 3;`
`padding-top`	`Layout`, `Page`	*双精度* | `initial`	`padding-top: 4;`
`padding-right`	`Layout`, `Page`	*双精度* | `initial`	`padding-right: 2;`
`padding-bottom`	`Layout`, `Page`	*双精度* | `initial`	`padding-bottom: 6;`
`position`	`FlexLayout`	`relative` | `absolute` | `initial`	`position: absolute;`

属性	适用对象	值	示例
`row-gap`	`Grid`	*双精度* | `initial`	`row-gap: 12;`
`text-align`	`Entry`, `EntryCell`, `Label`, `SearchBar`	`left` | `top` | `right` | `bottom` | `start` | `center` | `middle` | `end` | `initial`. `left` 和 `right` 应避免使用从右到左环境中。	`text-align: right;`
`text-decoration`	`Label`, `Span`	`none` | `underline` | `strikethrough` | `line-through` | `initial`	`text-decoration: underline, line-through;`
`transform`	`VisualElement`	`none`, `rotate`, `rotateX`, `rotateY`, `scale`, `scaleX`, `scaleY`, `translate`, `translateX`, `translateY`, `initial`	`transform: rotate(180), scaleX(2.5);`
`transform-origin`	`VisualElement`	*双精度*, *双精度* | `initial`	`transform-origin: 7.5, 12.5;`
`vertical-align`	`Label`	`left` | `top` | `right` | `bottom` | `start` | `center` | `middle` | `end` | `initial`	`vertical-align: bottom;`
`visibility`	`VisualElement`	`true` | `visible` | `false` | `hidden` | `collapse` | `initial`	`visibility: hidden;`
`width`	`VisualElement`	*双精度* | `initial`	`min-width: 320;`

此外支持以下 Xamarin.Forms 特定 CSS 属性 (在值列中，类型为_斜体_，而字符串文本是 `gray`):

属性	适用对象	值	示例
`-xf-placeholder`	`Entry`, `Editor`, `SearchBar`	*带引号的文本* | `initial`	`-xf-placeholder: Enter name;`
`-xf-placeholder-color`	`Entry`, `Editor`, `SearchBar`	*颜色* | `initial`	`-xf-placeholder-color: green;`
`-xf-max-length`	`Entry`, `Editor`	*Int* | `initial`	`-xf-max-length: 20;`
`-xf-bar-background-color`	`NavigationPage`, `TabbedPage`	*颜色* | `initial`	`-xf-bar-background-color: teal;`
`-xf-bar-text-color`	`NavigationPage`, `TabbedPage`	*颜色* | `initial`	`-xf-bar-text-color: gray`

属性	适用对象	值	示例
`-xf-orientation`	`ScrollView`, `StackLayout`	`horizontal` | `vertical` | `both` | `initial`. `both` 仅支持 `ScrollView`。	`-xf-orientation: horizontal;`
`-xf-horizontal-scroll-bar-visibility`	`ScrollView`	`default` | `always` | `never` | `initial`	`-xf-horizontal-scroll-bar-visibility: never;`
`-xf-vertical-scroll-bar-visibility`	`ScrollView`	`default` | `always` | `never` | `initial`	`-xf-vertical-scroll-bar-visibility: always;`
`-xf-min-track-color`	`Slider`	*颜色* | `initial`	`-xf-min-track-color: yellow;`
`-xf-max-track-color`	`Slider`	*颜色* | `initial`	`-xf-max-track-color: red;`
`-xf-thumb-color`	`Slider`	*颜色* | `initial`	`-xf-thumb-color: limegreen;`
`-xf-spacing`	`StackLayout`	*双精度* | `initial`	`-xf-spacing: 8;`

NOTE

`initial` 是为所有属性的有效值。它会清除从另一个样式设置的值（重置为默认值）。

是当前不支持以下属性：

- `all: initial`。
- 布局属性（框中，或网格）。
- 速记属性，如 `font`，和 `border`。

此外，还有没有 `inherit` 值，因此继承不受支持。因此不能例如，设置 `font-size` 布局的属性和预期所有 `Label` 要继承的值的布局中的实例。是一个例外 `direction` 属性，它具有默认值的 `inherit`。

颜色

以下 `color` 支持值：

- `X11` 颜色，其中匹配 CSS 颜色、UWP 预定义的颜色和 Xamarin.Forms 的颜色。请注意这些颜色值是不区分大小写。
- 十六进制颜色：`#rgb`，`#argb`，`#rrggbb`，`#aarrggbb`
- rgb 颜色：`rgb(255,0,0)`，`rgb(100%,0%,0%)`。值为范围内，0-255 或 0%-100%。
- rgba 颜色：`rgba(255, 0, 0, 0.8)`，`rgba(100%, 0%, 0%, 0.8)`。不透明度值为 0.0 到 1.0 范围内。
- hsl 颜色：`hsl(120, 100%, 50%)`。H 值为范围 0-360，尽管 s 和 l 是在范围 0%-100%。
- hsla 颜色：`hsla(120, 100%, 50%, .8)`。不透明度值为 0.0 到 1.0 范围内。

Thickness

一个、两个、三或四个 `thickness` 支持的值，每个由空格分隔：

- 单个值指示统一粗细，即。
- 两个值表示垂直然后水平的粗细。
- 三个值表示顶部，然后水平（左侧和右侧），然后底部粗细。

- 四个值表示顶部，然后右，然后底部，左侧的粗细。

> **NOTE**
> CSS `thickness` 值不同于 XAML `Thickness` 值。例如，在两个值 XAML `Thickness` 指示水平然后垂直粗细，四个值时 `Thickness` 指示左侧、顶部，然后右侧，然后下粗细。此外，XAML `Thickness` 逗号分隔的有效值。

NamedSize

以下不区分大小写 `namedsize` 支持值：

- `default`
- `micro`
- `small`
- `medium`
- `large`

每个的确切含义 `namedsize` 值是依赖于平台的与视图相关。

在 Xamarin.Forms 中使用 Xamarin.University CSS

Xamarin.Forms 3.0 CSS，也可由Xamarin 学院课程

相关链接

- MonkeyAppCSS（示例）
- 资源字典
- 使用 XAML 样式设置 Xamarin.Forms 应用的样式

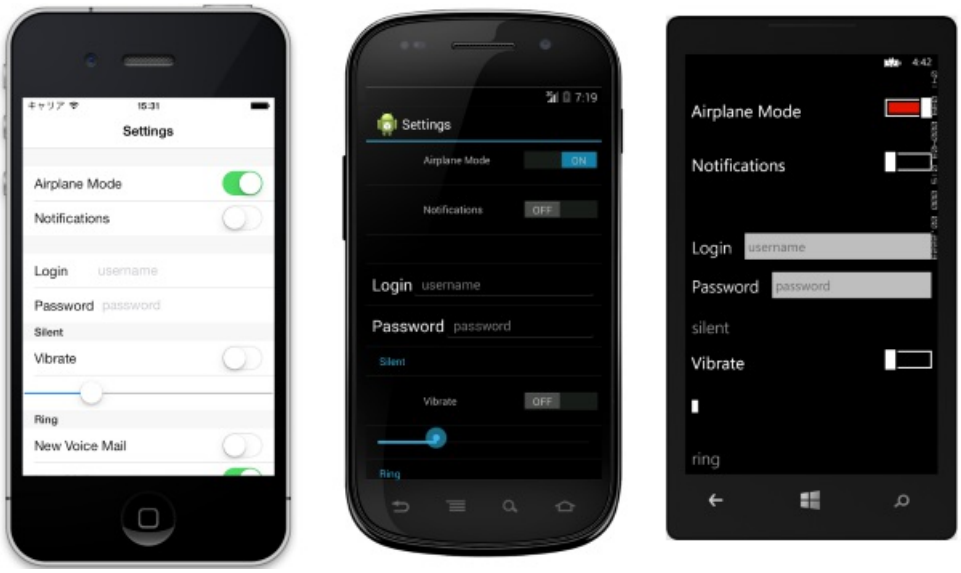
Xamarin.Forms TableView

2018/10/26 • • Edit Online

TableView是用于显示数据或选择的可滚动列表视图的行不共享同一个模板。与不同ListView，TableView 不具有这一概念 `ItemsSource`，因此，作为子级必须手动添加的项。

本指南是由以下部分组成：

- **用例** –何时使用 TableView 而不是 ListView 或自定义视图。
- **TableView 结构** – TableView 中所需的视图层次结构。
- **TableView 外观** – TableView 的自定义选项。
- **内置的单元格** –内置的单元格选项，包括EntryCell并SwitchCell。
- **自定义单元格** –如何使您自己的自定义单元格。



用例

TableView 当以下情况下非常有用：

- 显示一系列的设置，
- 在窗体中收集数据或
- 显示呈现的数据以不同的方式从行与行（例如数字、百分比和图像）。

TableView 处理滚动和布局中极具吸引力的部分中，针对上述情形，通常需要的行。`TableView` 控件使用每个平台的基础等效的视图时可用，创建每个平台的本机外观。

TableView 结构

中的元素 `TableView` 划分为部分。根目录 `TableView` 是 `TableRoot`，这是一个或多个父级 `TableSections`：

```
Content = new TableView {
    Root = new TableRoot {
        new TableSection...
    },
    Intent = TableIntent.Settings
};
```

每个 `TableSection` 标题和一个或多个 ViewCells 组成。这里我们可以看到 `TableSection` 的 `Title` 属性设置为 "*环*" 构造函数中：

```
var section = new TableSection ("Ring") { //TableSection constructor takes title as an optional parameter
    new SwitchCell {Text = "New Voice Mail"},
    new SwitchCell {Text = "New Mail", On = true}
};
```

若要完成如上所示在 XAML 中相同的布局：

```
<TableView Intent="Settings">
    <TableRoot>
        <TableSection Title="Ring">
            <SwitchCell Text="New Voice Mail" />
      <SwitchCell Text="New Mail" On="true" />
        </TableSection>
    </TableRoot>
</TableView>
```

TableView 外观

TableView 公开 `Intent` 属性，它是一个枚举，下列选项：

- **数据**–用于显示数据条目时。请注意，ListView可能是更好的选择进行滚动数据的列表。
- **窗体** – TableView 充当窗体时使用。
- **菜单**–供使用时显示选择的菜单。
- **设置**–供使用时显示的配置设置列表。

`TableIntent` 选择可能会影响如何 `TableView` 显示每个平台上。即使有不清除差异，它是最佳选择 `TableIntent` 的最匹配你想要使用表。

内置的单元格

Xamarin.Forms 提供了内置的单元格，以收集和显示信息。尽管 ListView 和 TableView 可以使用所有相同的单元格，但以下是最适用于 TableView 方案：

- **SwitchCell** –演示和捕获的 true/false 状态，以及文本标签。
- **EntryCell** –演示和捕获的文本。

请参阅ListView 单元格的外观有关的详细说明TextCell并ImageCell。

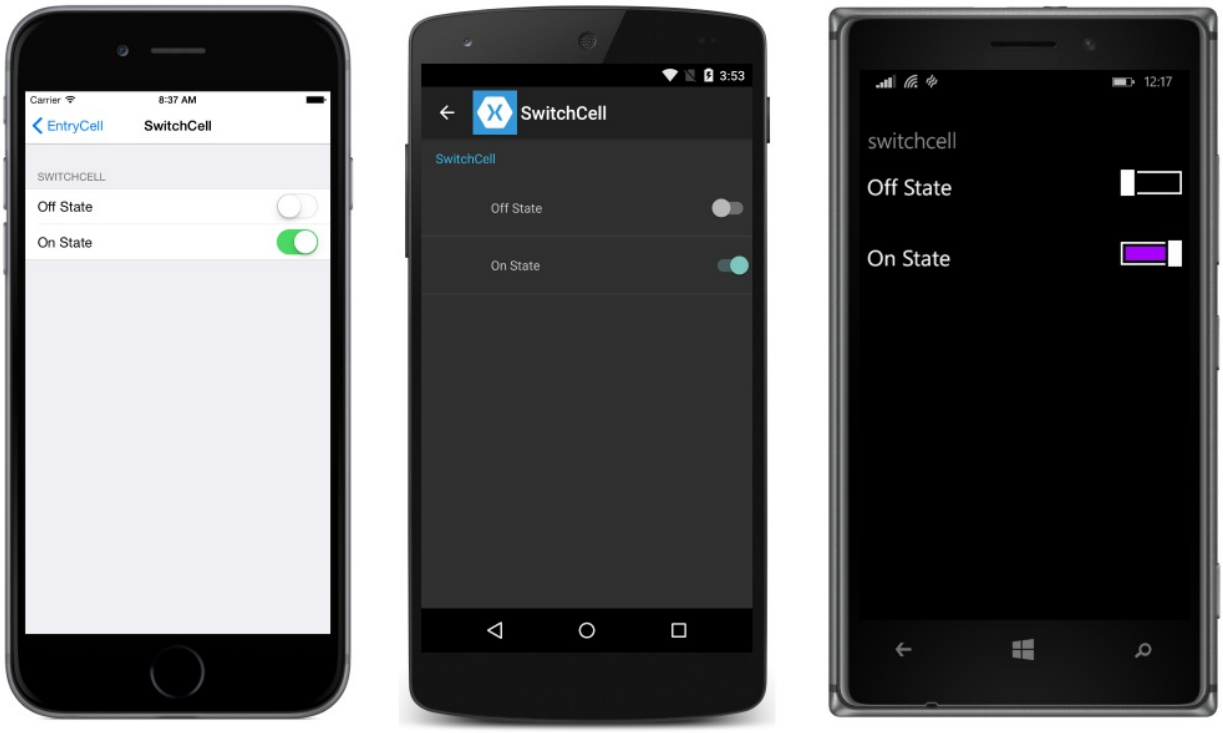
SwitchCell

`SwitchCell` 是要用于显示和捕获的控件打开/关闭或 `true` / `false` 状态。

SwitchCells 具有待编辑文本和打开/关闭属性的一个行。这些属性都可绑定。

- `Text` – 若要切换旁边显示的文本。
- `On` – 此开关是否显示为 on 或 off。

请注意，`SwitchCell` 公开 `OnChanged` 事件，从而可以对该单元格的状态变化作出响应。

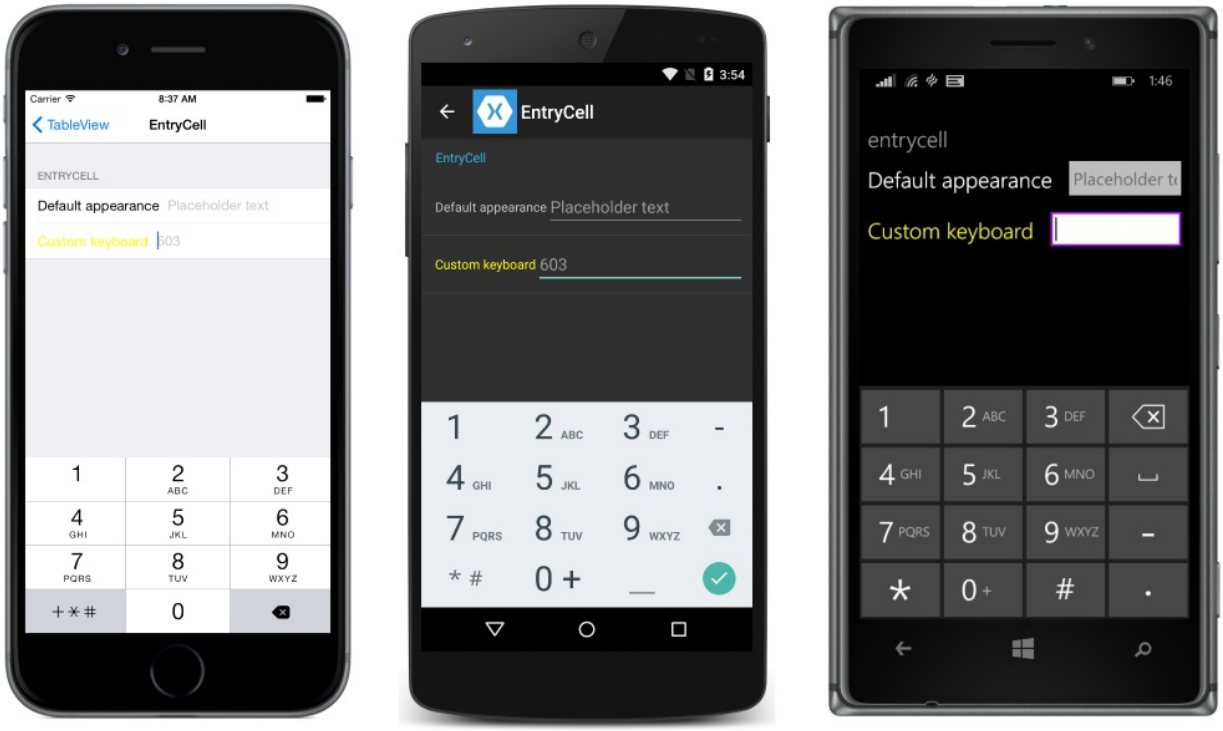


EntryCell

`EntryCell` 如果需要显示的用户可以编辑的文本数据时非常有用。`EntryCell` s 提供可自定义的以下属性：

- `Keyboard` – 若要编辑时显示键盘。有数字值、电子邮件、电话号码等之类的内容的选项。请参阅 API 文档。
- `Label` – 要在文本输入字段右侧显示的标签文本。
- `LabelColor` – 标签文本的颜色。
- `Placeholder` – 它为 null 或空时在输入字段中显示的文本。文本输入开始时，此文本将消失。
- `Text` – 在输入字段中的文本。
- `HorizontalTextAlignment` – 文本的水平对齐方式。可以在中心，左侧或右侧对齐。请参阅 API 文档。

请注意，`EntryCell` 公开 `Completed` 事件，如果用户点击完成键盘上编辑文本时则会激发该事件。

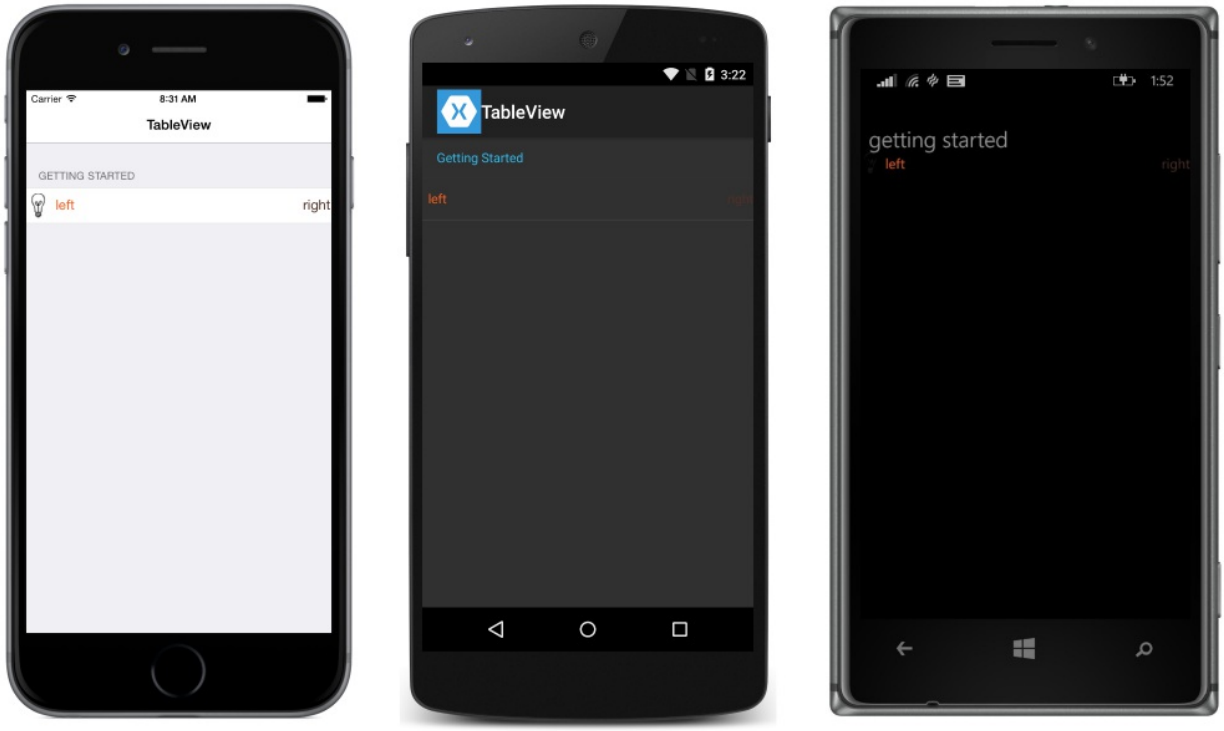


自定义单元格

当内置的单元格仍不足时，可以使用自定义单元格能够并捕获有关您的应用程序有意义的方式中的数据。例如，您可能希望显示滑块，以允许用户选择的图像的不透明度。

自定义的所有单元格必须派生自 `ViewCell`，所有内置的单元格类型使用相同的基类。

这是自定义单元格的一个示例：



XAML

若要创建上述布局 XAML 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="DemoTableView.TablePage" Title="TableView">
    <ContentPage.Content>
        <TableView Intent="Settings">
            <TableRoot>
                <TableSection Title="Getting Started">
                    <ViewCell>
                        <StackLayout Orientation="Horizontal">
                            <Image Source="bulb.png" />
                            <Label Text="left"
                              TextColor="#f35e20" />
                            <Label Text="right"
                              HorizontalOptions="EndAndExpand"
                              TextColor="#503026" />
                        </StackLayout>
                    </ViewCell>
                </TableSection>
            </TableRoot>
        </TableView>
    </ContentPage.Content>
</ContentPage>
```

上述 XAML 执行大量操作。让我们深入细致地：

- 根元素下的 `TableView` 是 `TableRoot`。

- 没有 `TableSection` 立即下方 `TableRoot` 。
- `ViewCell` 直接下节定义。与不同 `ListView` ， `TableView` 不需要该自定义（或者任何）中定义的单元格 `ItemTemplate` 。
- 在 StackLayout 用于管理自定义单元格的布局。无法在此处使用任何布局。

C#

因为 `TableView` 工作方式与静态数据或手动更改的数据，它没有项模板的概念。相反，自定义单元格可以手动创建并放入表。请注意技术创建自定义单元格的继承 `ViewCell` ，然后将其添加到 `TableView` 像您一样将内置的单元格，还支持。下面是 c# 代码来完成上述布局：

```
var table = new TableView();
table.Intent = TableIntent.Settings;
var layout = new StackLayout() { Orientation = StackOrientation.Horizontal };
layout.Children.Add (new Image() {Source = "bulb.png"});
layout.Children.Add (new Label() {
    Text = "left",
    TextColor = Color.FromHex("#f35e20"),
    VerticalOptions = LayoutOptions.Center
});
layout.Children.Add (new Label () {
    Text = "right",
    TextColor = Color.FromHex ("#503026"),
    VerticalOptions = LayoutOptions.Center,
    HorizontalOptions = LayoutOptions.EndAndExpand
});
table.Root = new TableRoot () {
    new TableSection("Getting Started") {
        new ViewCell() {View = layout}
    }
};

Content = table;
```

上面的 C# 执行大量。让我们深入细致地：

- 根元素下的 `TableView` 是 `TableRoot` 。
- 没有 `TableSection` 立即下方 `TableRoot` 。
- `ViewCell` 直接下节定义。与不同 `ListView` ， `TableView` 不需要该自定义（或者任何）中定义的单元格 `ItemTemplate` 。
- 在 StackLayout 用于管理自定义单元格的布局。无法在此处使用任何布局。

请注意，永远不会定义一个类，用于自定义单元格。相反， `ViewCell` 的视图属性设置的特定实例 `ViewCell` 。

行高

`TableView` 类具有可用于更改单元格的行的高度的两个属性：

- `RowHeight` – 设置为每个行的高度 `int` 。
- `HasUnevenRows` – 如果行具有不同高度设置为 `true` 。请注意，当此属性设置为 `true` ，行高度自动计算，并通过 Xamarin.Forms 应用。

当中某个单元格中的内容的高度 `TableView` 更改行高度隐式更新 Android 和通用 Windows 平台 (UWP) 上。但是，在 iOS 上它必须强制通过设置更新 `HasUnevenRows` 属性设置为 `true` 并通过调用 `Cell.ForceUpdateSize` 方法。

下面的 XAML 示例演示 `TableView` ，其中包含 `ViewCell` ：

```
<ContentPage ...>
    <TableView ...
               HasUnevenRows="true">
        <TableRoot>
            ...
            <TableSection ...>
                ...
                <ViewCell x:Name="_viewCell"
                          Tapped="OnViewCellTapped">
                    <Grid Margin="15,0">
                        <Grid.RowDefinitions>
                            <RowDefinition Height="Auto" />
                            <RowDefinition Height="Auto" />
                        </Grid.RowDefinitions>
                        <Label Text="Tap this cell." />
                        <Label x:Name="_target"
                               Grid.Row="1"
                               Text="The cell has changed size."
                               IsVisible="false" />
                    </Grid>
                </ViewCell>
            </TableSection>
        </TableRoot>
    </TableView>
</ContentPage>
```

当 `ViewCell` 点击，`OnViewCellTapped` 执行事件处理程序：

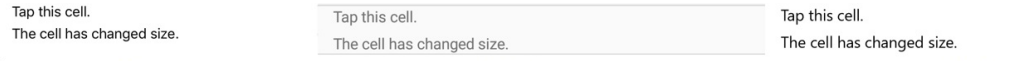
```
void OnViewCellTapped(object sender, EventArgs e)
{
    _target.IsVisible = !_target.IsVisible;
    _viewCell.ForceUpdateSize();
}
```

`OnViewCellTapped` 事件处理程序显示或隐藏第二个 `Label` 中 `ViewCell`，并通过调用将显式更新该单元格的大小 `Cell.ForceUpdateSize` 方法。

以下屏幕截图显示之前在点击后的单元格：

iOS　　Android　　UWP

下面的屏幕截图显示在点击后的单元格：



iOS　　Android　　UWP

> **IMPORTANT**
>
> 如果过度使用此功能，则强可能导致性能下降。

相关链接

- TableView（示例）

在 Xamarin.Forms 中的文本

2018/11/2 • • Edit Online

使用 Xamarin.Forms 来输入或显示文本。

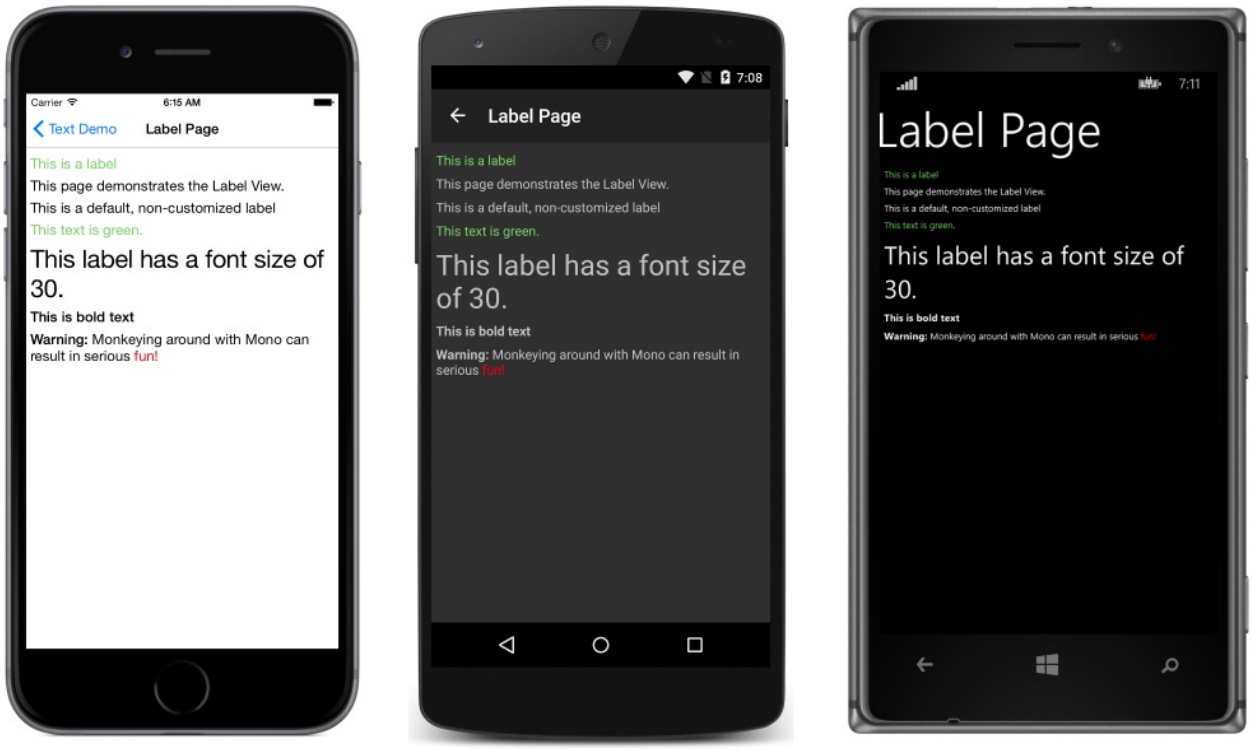
Xamarin.Forms 具有用于处理文本的三个主要视图：

- **标签** —来显示单个或多行文本。可以在同一行中显示多个格式设置选项的文本。
- **条目** —用于输入只有一行的文本。条目都具有一种密码模式。
- **编辑器** —用于输入文本，可能也需要多个行。

可以使用内置或自定义更改文本外观样式和某些控件支持的自定义字体。

标签

`Label` 视图用于显示文本。它可以显示多行文本或单个文本行。`Label` 可以提供具有多个中内嵌使用的格式设置选项的文本。标签视图可以包装或截断时它不适合在同一行的文本。

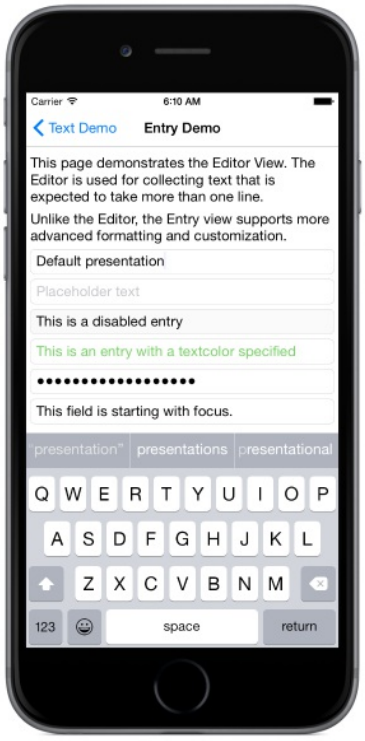


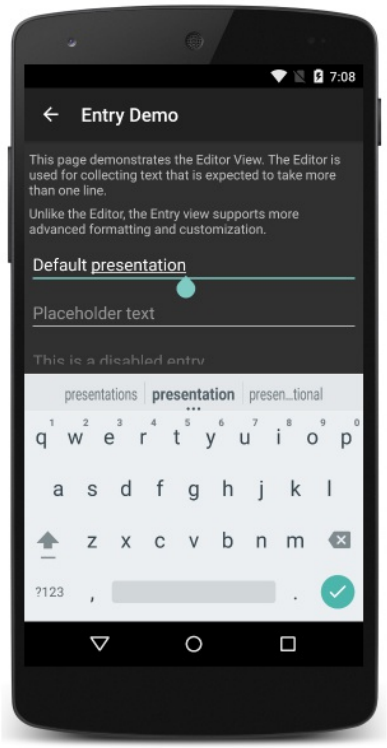
请参阅标签文章更多详细信息。

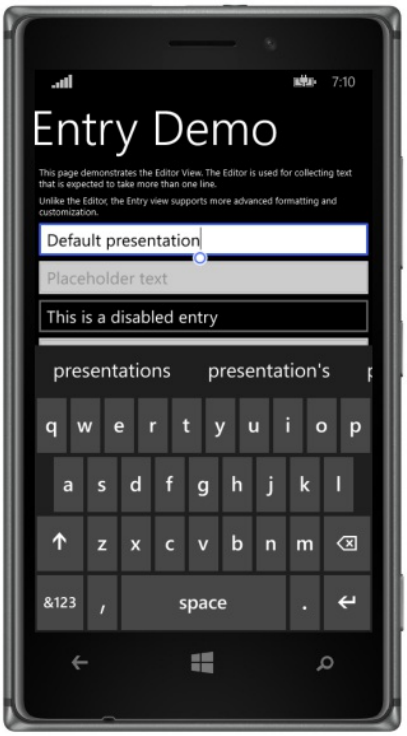
有关自定义标签中使用的字体的信息，请参阅字体。

项

`Entry` 用来接受的单行文本输入。`Entry` 产品/服务控制颜色和字体。`Entry` 具有密码模式，可以显示占位符文本，直到输入文本。





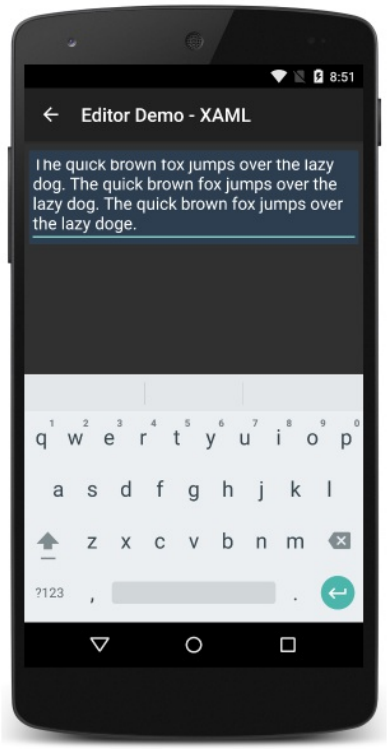


请参阅条目文章了解详细信息。

请注意，与不同 `Label`，`Entry` 不能具有自定义字体设置。

编辑器

`Editor` 用于接受多行文本输入。`Editor` 产品/服务控制颜色和字体。



请参阅编辑器文章了解详细信息。

字体

`Label` 控件支持不同的字体设置，请在每个平台或您的应用程序中包含的自定义字体使用内置的字体。请参阅字

体文章更多详细信息。

样式

请参阅使用样式若要了解如何设置字体颜色，和其他适用于多个控件的显示属性。

相关链接

- 文本（示例）

Xamarin.Forms 标签

2018/10/31 • • Edit Online

在 Xamarin.Forms 中显示文本

`Label` 视图用于显示文本、单个和多行。标签可以具有文本修饰，彩色文本，并使用自定义字体（系列、大小和选项）。

文本修饰

下划线和删除线的文本修饰可应用于 `Label` 通过设置实例 `Label.TextDecorations` 属性设置为一个或多个 `TextDecorations` 枚举成员：

- `None`
- `Underline`
- `Strikethrough`

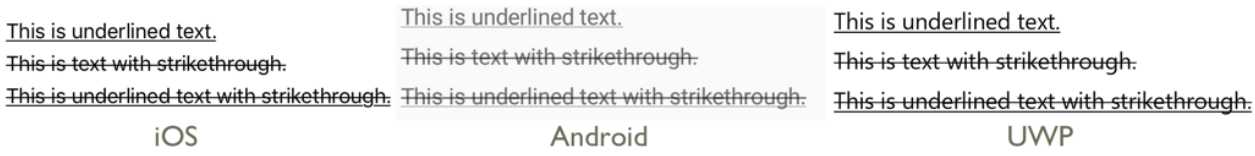
下面的 XAML 示例演示了如何设置 `Label.TextDecorations` 属性：

```
<Label Text="This is underlined text." TextDecorations="Underline"  />
<Label Text="This is text with strikethrough." TextDecorations="Strikethrough" />
<Label Text="This is underlined text with strikethrough." TextDecorations="Underline, Strikethrough" />
```

等效的 C# 代码是：

```
var underlineLabel = new Label { Text = "This is underlined text.", TextDecorations =
TextDecorations.Underline };
var strikethroughLabel = new Label { Text = "This is text with strikethrough.", TextDecorations =
TextDecorations.Strikethrough };
var bothLabel = new Label { Text = "This is underlined text with strikethrough.", TextDecorations =
TextDecorations.Underline | TextDecorations.Strikethrough };
```

下面的屏幕截图演示 `TextDecorations` 应用于枚举成员 `Label` 实例：



> **NOTE**
> 此外可应用于的文本修饰 `Span` 实例。有关详细信息 `Span` 类，请参阅格式的文本。

颜色

标签可以设置为使用通过可绑定的自定义文本颜色 `TextColor` 属性。

特别注意有必要确保将每个平台上可用的颜色。因为每个平台都有不同的文本和背景颜色的默认值，将需要谨慎地选择适用于每个默认值。

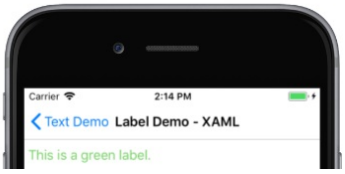
下面的 XAML 示例设置的文本颜色 `Label`：

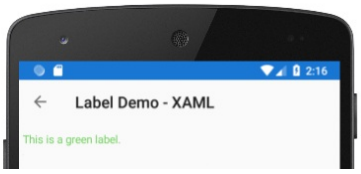
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TextSample.LabelPage"
             Title="Label Demo">
    <StackLayout Padding="5,10">
      <Label TextColor="#77d065" FontSize = "20" Text="This is a green label." />
    </StackLayout>
</ContentPage>
```

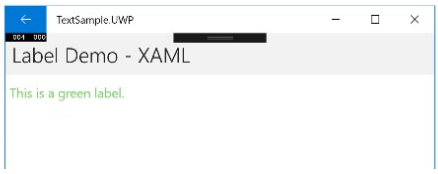
等效的 C# 代码是：

```
public partial class LabelPage : ContentPage
{
    public LabelPage ()
    {
        InitializeComponent ();

        var layout = new StackLayout { Padding = new Thickness(5,10) };
        var label = new Label { Text="This is a green label.", TextColor = Color.FromHex("#77d065"), FontSize
= 20 };
        layout.Children.Add(label);
        this.Content = layout;
    }
}
```

以下屏幕截图显示的设置结果 `TextColor` 属性：







有关颜色的详细信息，请参阅颜色。

字体

有关详细信息，有关上指定字体 `Label`，请参阅字体。

截断和换行

可以设置标签来处理不适合在以下几种方式，通过公开的一行文本 `LineBreakMode` 属性。 `LineBreakMode` 是一个枚举，使用以下值：

- **HeadTruncation** –将截断的文本，其中显示最终的开头。
- **CharacterWrap** –到新行上的文本进行换行字符边界处。
- **MiddleTruncation** –显示开头和末尾的文本，与通过省略号中间的替换。
- **NoWrap** –不换行文本，仅显示可以为任意数量的文本适合某个行。
- **TailTruncation** –显示文本，截断结束的开头。
- **换行**–使文本在单词边界处换行。

显示指定的行数

通过显示的行数 `Label` 可以指定通过设置 `Label.MaxLines` 属性设置为 `int` 值：

- 当 `MaxLines` 为 0， `Label` 尊重的值 `LineBreakMode` 属性既可以显示一个行中，可能会截断或具有的所有文本的所有行。

- 当 `MaxLines` 为 1，结果等同于设置 `LineBreakMode` 属性设置为 `NoWrap` ， `HeadTruncation` ， `MiddleTruncation` ，或 `TailTruncation` 。但是， `Label` 将遵守的值 `LineBreakMode` 省略号，如果适用的位置方面的属性。
- 当 `MaxLines` 大于 1， `Label` 指定数量的行，同时遵循的值将显示 `LineBreakMode` 省略号，如果适用的位置方面的属性。但是，将设置 `MaxLines` 属性的值大于 1 如果不起作用 `LineBreakMode` 属性设置为 `NoWrap` 。

下面的 XAML 示例演示了如何设置 `MaxLines` 上的属性 `Label` ：

```
<Label Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla
vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus."
       LineBreakMode="WordWrap"
       MaxLines="2" />
```

等效的 C# 代码是：

```
var label =
{
  Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla
vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.", LineBreakMode = LineBreakMode.WordWrap,
  MaxLines = 2
};
```

以下屏幕截图显示的设置结果 `MaxLines` 属性设置为 2，在文本时足够长，以占用 2 个以上的行：

带格式文本

标签公开 `FormattedText` 这允许文本呈现使用多种字体和颜色在同一视图中的属性。

`FormattedText` 属性属于类型 `FormattedString` ，其中包含一个或多个 `Span` 情况下，通过设置 `Spans` 属性. 以下 `Span` 属性可以用于设置可视外观：

- `BackgroundColor` -s p a n 背景的颜色。
- `Font` – 在范围中的文本的字体。
- `FontAttributes` – 在范围中的文本的字体属性。
- `FontFamily` – 属于该范围中的文本的字体的字体系列。
- `FontSize` – 范围中文本的字体的大小。
- `ForegroundColor` – 范围中文本的颜色。此属性已过时，已由 `TextColor` 属性。
- `LineHeight` -若要将应用于跨度的默认行高度的乘数。有关详细信息，请参阅行高。
- `Style` – 要将应用于跨度的样式。
- `Text` – 跨度的文本。
- `TextColor` – 范围中文本的颜色。
- `TextDecorations` -若要应用于范围中的文本的修饰。有关详细信息，请参阅文本修饰。

此外， `GestureRecognizers` 属性可以用于定义将在响应手势的手势识别器的集合 `Span` 。

下面的 XAML 示例演示 `FormattedText` 由三个属性 `Span` 实例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="TextSample.LabelPage"
             Title="Label Demo - XAML">
    <StackLayout Padding="5,10">
        ...
        <Label LineBreakMode="WordWrap">
            <Label.FormattedText>
                <FormattedString>
                    <Span Text="Red Bold, " TextColor="Red" FontAttributes="Bold" />
                    <Span Text="default, " Style="{DynamicResource BodyStyle}">
                        <Span.GestureRecognizers>
                            <TapGestureRecognizer Command="{Binding TapCommand}" />
                        </Span.GestureRecognizers>
                    </Span>
                    <Span Text="italic small." FontAttributes="Italic" FontSize="Small" />
                </FormattedString>
            </Label.FormattedText>
        </Label>
    </StackLayout>
</ContentPage>
```

等效的 C# 代码是：

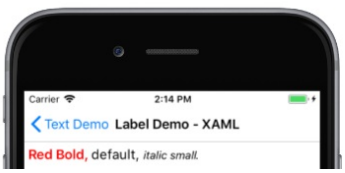
```
public class LabelPageCode : ContentPage
{
    public LabelPageCode ()
    {
        var layout = new StackLayout{ Padding = new Thickness (5, 10) };
        ...
        var formattedString = new FormattedString ();
        formattedString.Spans.Add (new Span{ Text = "Red bold, ", ForegroundColor = Color.Red, FontAttributes
= FontAttributes.Bold });

        var span = new Span { Text = "default, " };
        span.GestureRecognizers.Add(new TapGestureRecognizer { Command = new Command(async () => await
DisplayAlert("Tapped", "This is a tapped Span.", "OK")) });
        formattedString.Spans.Add(span);
        formattedString.Spans.Add (new Span { Text = "italic small.", FontAttributes = FontAttributes.Italic,
FontSize =  Device.GetNamedSize(NamedSize.Small, typeof(Label)) });

        layout.Children.Add (new Label { FormattedText = formattedString });
        this.Content = layout;
    }
}
```

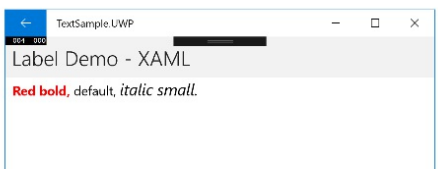
IMPORTANT

`Text` 属性的 `Span` 可以通过数据绑定设置。有关详细信息，请参阅数据绑定。

请注意，`Span` 还可以添加到 span 的任何手势响应 `GestureRecognizers` 集合。例如，`TapGestureRecognizer` 已添加到第二个 `Span` 在上面的代码示例。因此，当这 `Span` 点击 `TapGestureRecognizer` 将通过执行响应 `ICommand` 由定义 `Command` 属性。有关手势识别器的详细信息，请参阅Xamarin.Forms 手势。

以下屏幕截图显示设置的结果 `FormattedString` 属性设置为三个 `Span` 实例：





行高

垂直高度 `Label` 和一个 `Span` 可以通过设置自定义 `Label.LineHeight` 属性或 `Span.LineHeight` 到 `double` 值。 IOS 和 Android 上这些值是乘数的原始行高度，并在通用 Windows 平台 (UWP) `Label.LineHeight` 属性值是标签字体大小的倍数。

> **NOTE**
>
> - 在 iOS 上， `Label.LineHeight` 并 `Span.LineHeight` 属性更改的单个行中，可以容纳的文本和换行到多个行的文本行高度。
> - 在 Android 上， `Label.LineHeight` 并 `Span.LineHeight` 属性只能更改换行到多个行的文本行高度。
> - 在 UWP 中， `Label.LineHeight` 属性更改的换行到多个行的文本行高度和 `Span.LineHeight` 属性不起作用。

下面的 XAML 示例演示了如何设置 `LineHeight` 上的属性 `Label` ：

```
<Label Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla
vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus."
       LineBreakMode="WordWrap"
       LineHeight="1.8" />
```

等效的 C# 代码是：

```
var label =
{
  Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In facilisis nulla eu felis fringilla
vulputate. Nullam porta eleifend lacinia. Donec at iaculis tellus.", LineBreakMode = LineBreakMode.WordWrap,
  LineHeight = 1.8
};
```

以下屏幕截图显示设置的结果 `Label.LineHeight` 属性设置为 1.8:

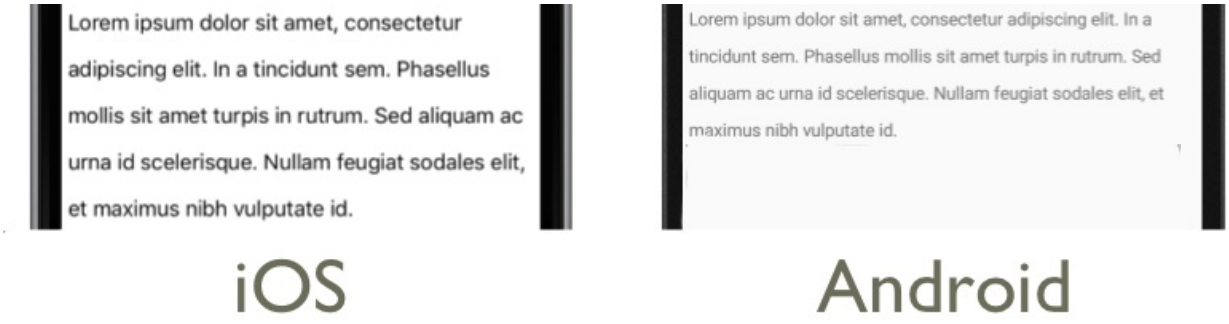
下面的 XAML 示例演示了如何设置 `LineHeight` 上的属性 `Span` ：

```
<Label LineBreakMode="WordWrap">
    <Label.FormattedText>
        <FormattedString>
            <Span Text="Lorem ipsum dolor sit amet, consectetur adipiscing elit. In a tincidunt sem.
Phasellus mollis sit amet turpis in rutrum. Sed aliquam ac urna id scelerisque. "
                  LineHeight="1.8"/>
            <Span Text="Nullam feugiat sodales elit, et maximus nibh vulputate id."
                  LineHeight="1.8" />
        </FormattedString>
    </Label.FormattedText>
</Label>
```

等效的 C# 代码是：

```
var formattedString = new FormattedString();
formattedString.Spans.Add(new Span
{
  Text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. In a tincidunt sem. Phasellus mollis sit
amet turpis in rutrum. Sed aliquam ac urna id scelerisque. ",
  LineHeight = 1.8
});
formattedString.Spans.Add(new Span
{
  Text = "Nullam feugiat sodales elit, et maximus nibh vulputate id.",
  LineHeight = 1.8
});
var label = new Label
{
  FormattedText = formattedString,
  LineBreakMode = LineBreakMode.WordWrap
};
```

以下屏幕截图显示设置的结果 `Span.LineHeight` 属性设置为 1.8:



样式标签

前面几节介绍设置 `Label` 并 `Span` 根据每个实例的属性。但是，属性集可以分组为一致地应用于一个或多个视图的一种样式。这可以提高代码的可读性，并轻松地实现设计更改。有关详细信息，请参阅样式。

相关链接

- 文本（示例）
- 借助 Xamarin.Forms，第 3 章创建移动应用
- 标签 API
- 跨度 API

Xamarin.Forms 条目

2018/11/5 • • Edit Online

单行文本或输入的密码

Xamarin.Forms `Entry` 用于单行文本输入。`Entry`，例如 `Editor` 视图中，支持多个键盘类型。此外，`Entry` 可以用作密码字段。

显示自定义

设置和读取文本

`Entry`，如其他文本呈现的视图，公开 `Text` 属性。此属性可以用于设置和读取通过显示的文本 `Entry`。下面的示例演示了如何设置 `Text` 在 XAML 中的属性：

```
<Entry Text="I am an Entry" />
```

在 C# 中：

```
var MyEntry = new Entry { Text = "I am an Entry" };
```

若要读取的文本，访问 `Text` C# 中的属性：

```
var text = MyEntry.Text;
```

> **NOTE**
> 宽度 `Entry` 可以通过设置定义其 `WidthRequest` 属性。不依赖于的宽度 `Entry` 所定义的值基于其 `Text` 属性。

输入的长度限制

`MaxLength` 属性可用于限制所允许的范围的输入的长度 `Entry`。此属性应设置为一个正整数：

```
<Entry ... MaxLength="10" />
```

```
var entry = new Entry { ... MaxLength = 10 };
```

一个 `MaxLength` 属性值为 0 指示将允许任何输入，并将值 `int.MaxValue`，它是默认值，`Entry`，指示是否有任何可能输入的字符数的有效限制。

设置光标位置和文本所选内容长度

`CursorPosition` 属性可以用于返回或设置下一个字符将插入中存储的字符串的位置 `Text` 属性：

```
<Entry Text="Cursor position set" CursorPosition="5" />
```

```
var entry = new Entry { Text = "Cursor position set", CursorPosition = 5 };
```

默认值 `CursorPosition` 属性为 0，表示将在开头插入文本 `Entry`。

此外，`SelectionLength` 属性可以用于返回或设置选定文本的长度 `Entry`：

```
<Entry Text="Cursor position and selection length set" CursorPosition="2" SelectionLength="10" />
```

```
var entry = new Entry { Text = "Cursor position and selection length set", CursorPosition = 2,
SelectionLength = 10 };
```

默认值 `SelectionLength` 属性为 0，指示未选定任何文本。

自定义键盘

当与用户交互时显示键盘 `Entry` 可以通过编程方式设置 `Keyboard` 属性，因为的以下属性之一 `Keyboard` 类：

- `Chat` – 用于短和表情符号是有用的地方。
- `Default` – 默认键盘。
- `Email` – 输入电子邮件地址时使用。
- `Numeric` – 输入数字时使用。
- `Plain` – 使用输入文本，而无需任何时 `KeyboardFlags` 指定。
- `Telephone` – 使用输入电话号码时。
- `Text` – 输入文本时使用。
- `Url` – 用于输入文件路径（& a）的 web 地址。

这可以实现在 XAML 中，如下所示：

```
<Entry Keyboard="Chat" />
```

等效的 C# 代码是：

```
var entry = new Entry { Keyboard = Keyboard.Chat };
```

可以在中找到的每个键盘示例我们配方存储库。

`Keyboard` 类还具有 `Create` 工厂方法，可以用来通过指定的大小写、拼写检查，并建议行为自定义键盘。`KeyboardFlags` 枚举值指定为方法，为自定义的自变量 `Keyboard` 返回。`KeyboardFlags` 枚举包含的以下值：

- `None` – 没有功能添加到键盘。
- `CapitalizeSentence` – 指示每个输入的句子的第一个单词的第一个字母将自动大写。
- `Spellcheck` – 指示该拼写检查功能将在输入的文本上执行。
- `Suggestions` – 指示该单词上输入的文本，则将提供完成。
- `CapitalizeWord` – 指示每个单词的第一个字母将自动大写。
- `CapitalizeCharacter` – 指示每个字符将自动大写。
- `CapitalizeNone` -指示没有自动大写会发生。
- `All` – 指示拼写检查、第完成单词和句子大小写会在输入的文本上发生。

以下 XAML 代码示例演示如何自定义默认值 `Keyboard` 提供完成单词和每个输入的字符大写：

```
<Entry Placeholder="Enter text here">
    <Entry.Keyboard>
        <Keyboard x:FactoryMethod="Create">
            <x:Arguments>
                <KeyboardFlags>Suggestions,CapitalizeCharacter</KeyboardFlags>
            </x:Arguments>
        </Keyboard>
    </Entry.Keyboard>
</Entry>
```

等效的 C# 代码是：

```
var entry = new Entry { Placeholder = "Enter text here" };
entry.Keyboard = Keyboard.Create(KeyboardFlags.Suggestions | KeyboardFlags.CapitalizeCharacter);
```

自定义 Return 键

Return 键是软键盘上的外观显示何时 `Entry` 具有焦点，可以通过设置自定义 `ReturnType` 属性的值 `ReturnType` 枚举：

- `Default` – 指示是否需要任何特定的 return 键，并且将使用平台默认值。
- `Done` – 指示"Done"的 return 键。
- `Go` – 表示"Go"return 键。
- `Next` – 表示"下一步"返回密钥。
- `Search` – 表示"搜索"返回密钥。
- `Send` – 指示"发送"return 键。

下面的 XAML 示例演示如何设置返回的项：

```
<Entry ReturnType="Send" />
```

等效的 C# 代码是：

```
var entry = new Entry { ReturnType = ReturnType.Send };
```

> **NOTE**
> 返回键的确切外观是依赖于平台。在 iOS 上，返回的密钥是基于文本的按钮。但是，在 Android 和通用 Windows 平台中，return 键是基于图标的按钮。

按下 return 键时，`Completed` 事件就会激发，并且任何 `ICommand` 指定的 `ReturnCommand` 执行属性。此外，任何 `object` 通过指定 `ReturnCommandParameter` 属性将传递给 `ICommand` 作为参数。有关命令的详细信息，请参阅命令界面。

启用和禁用拼写检查

`IsSpellCheckEnabled` 属性控制是否拼写检查已启用。默认情况下，该属性设置为 `true`。当用户输入文本，指示拼写错误。

但是，对于某些文本项方案，例如输入用户名，拼写检查功能提供负体验，并应通过设置禁用 `IsSpellCheckEnabled` 属性设置为 `false`：

```
<Entry ... IsSpellCheckEnabled="false" />
```

```
var entry = new Entry { ... IsSpellCheckEnabled = false };
```

> **NOTE**
> 当 `IsSpellCheckEnabled` 属性设置为 `false`，并自定义键盘未被使用，将禁用本机拼写检查器。但是，如果 `Keyboard` 具有已设置，以禁用拼写检查，如 `Keyboard.Chat`，则 `IsSpellCheckEnabled` 属性将被忽略。因此，该属性不能用于启用拼写检查 `Keyboard` 的显式禁用它。

启用和禁用文本预测

`IsTextPredictionEnabled` 属性控制是否文本预测和自动启用文本校正。默认情况下，该属性设置为 `true`。当用户输入文本时，会显示 word 预测。

但是，对于某些文本项方案，例如输入用户名、文本预测和文本自动更正提供负体验，并应通过设置禁用 `IsTextPredictionEnabled` 属性设置为 `false`：

```
<Entry ... IsTextPredictionEnabled="false" />
```

```
var entry = new Entry { ... IsTextPredictionEnabled = false };
```

> **NOTE**
> 当 `IsTextPredictionEnabled` 属性设置为 `false`，和自定义键盘不被使用，文本预测和自动禁用文本更正。但是，如果 `Keyboard` 已设置该禁用文本预测 `IsTextPredictionEnabled` 属性将被忽略。因此，该属性不能用于启用文本预测 `Keyboard` 的显式禁用它。

设置占位符文本

`Entry` 可以设置为显示占位符文本，它不存储用户输入时。这可以通过设置 `Placeholder` 属性设置为 `string`，并通常用于指示的是适用于的内容类型 `Entry`。此外，通过设置控制的占位符文本颜色 `PlaceholderColor` 属性设置为 `Color`：

```
<Entry Placeholder="Username" PlaceholderColor="Olive" />
```

```
var entry = new Entry { Placeholder = "Username", PlaceholderColor = Color.Olive };
```

密码字段

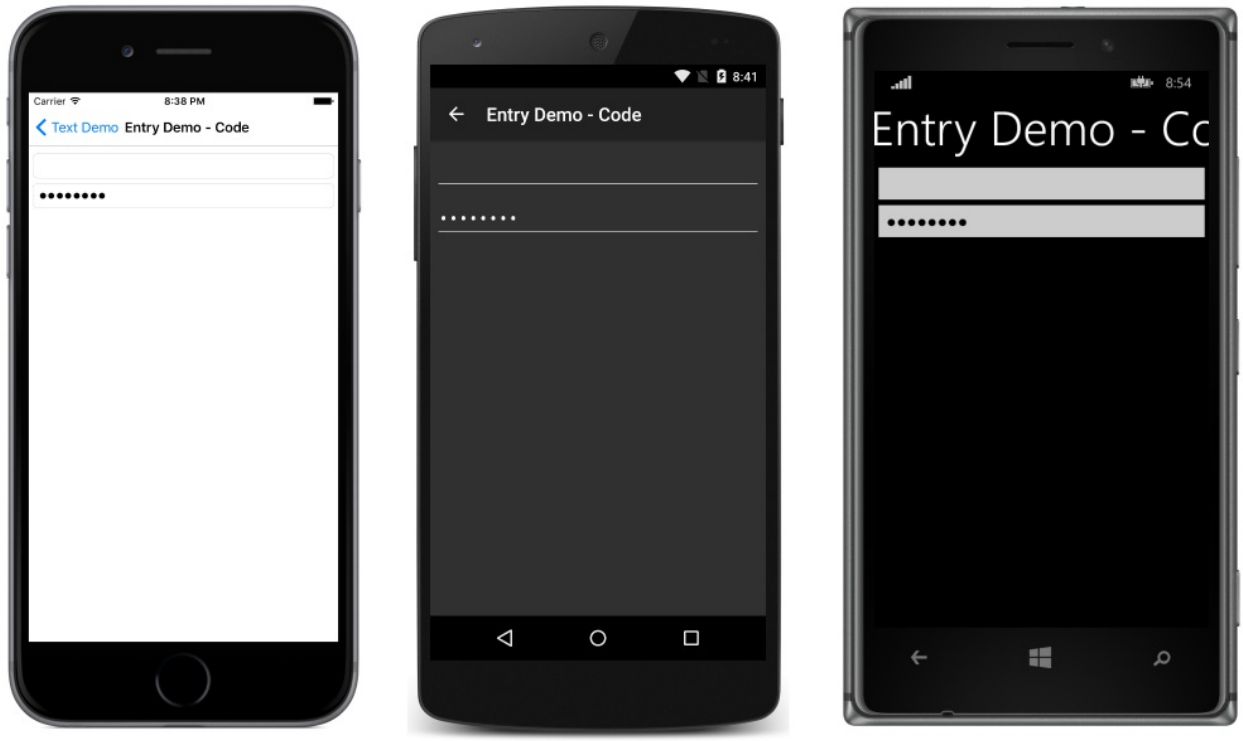
`Entry` 提供了 `IsPassword` 属性。当 `IsPassword` 是 `true`，字段的内容将显示为黑色圆圈：

在 XAML：

```
<Entry IsPassword="true" />
```

在 C# 中：

```
var MyEntry = new Entry { IsPassword = true };
```



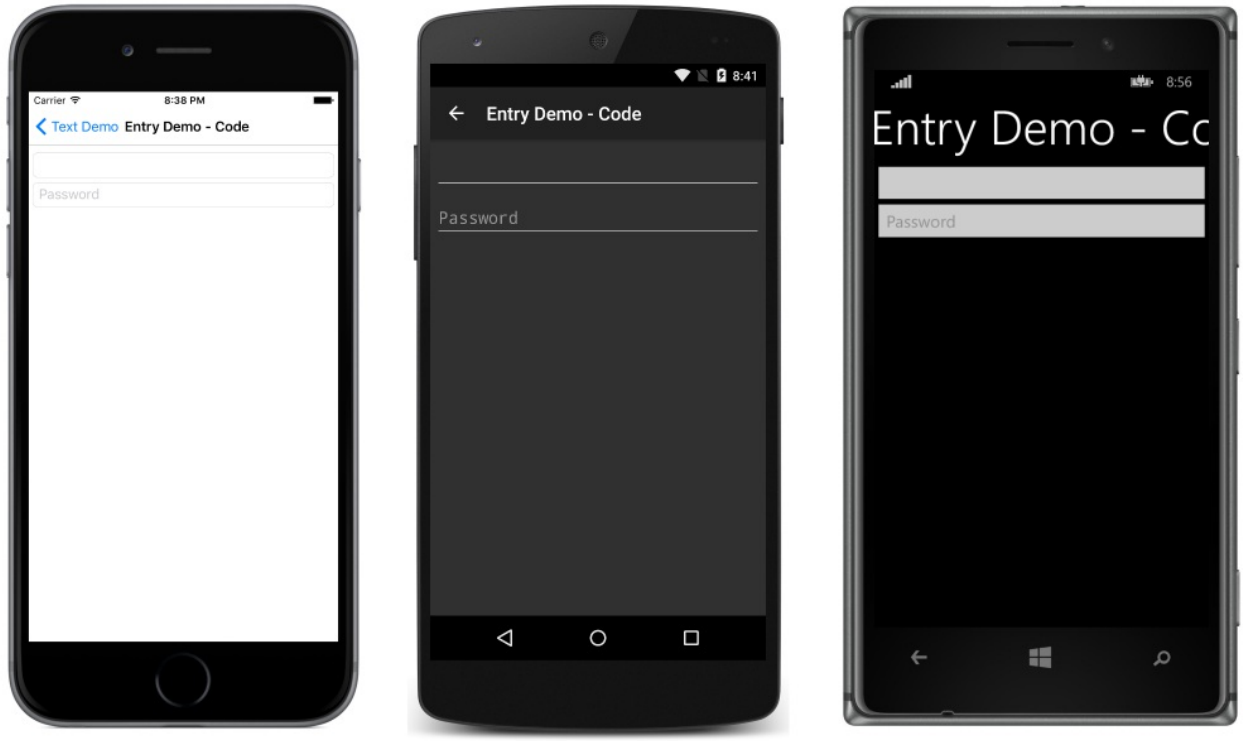
占位符可能使用的实例，并用 `Entry` 的配置为密码字段：

在 XAML：

```
<Entry IsPassword="true" Placeholder="Password" />
```

在 C# 中：

```
var MyEntry = new Entry { IsPassword = true, Placeholder = "Password" };
```



颜色

若要使用的自定义背景和文本颜色通过以下可绑定属性，可以设置项：

- **TextColor** –设置文本的颜色。
- **BackgroundColor** –设置显示文本的背景的颜色。

特别注意有必要确保将每个平台上可用的颜色。因为每个平台都有不同的默认值的文本和背景颜色，通常需要同时设置，如果你设置一个。

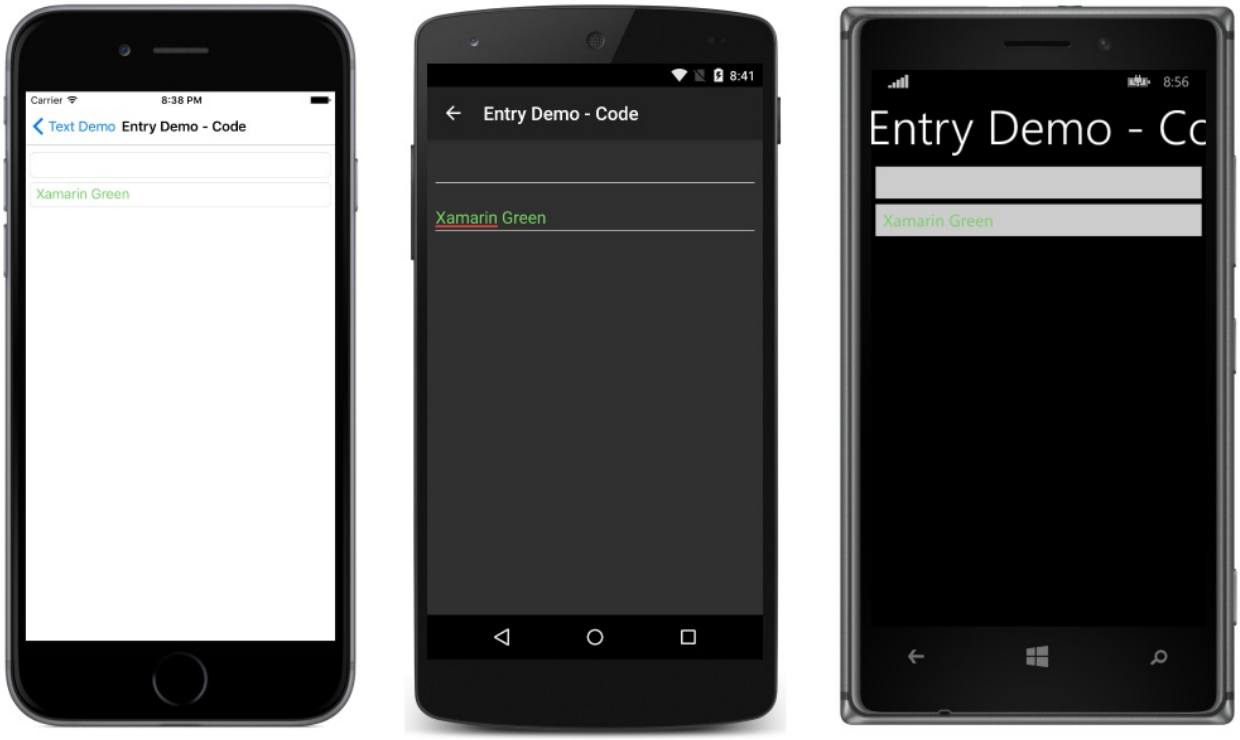
使用以下代码设置的项的文本颜色：

在 XAML:

```
<Entry TextColor="Green" />
```

在 C# 中：

```
var entry = new Entry();
entry.TextColor = Color.Green;
```



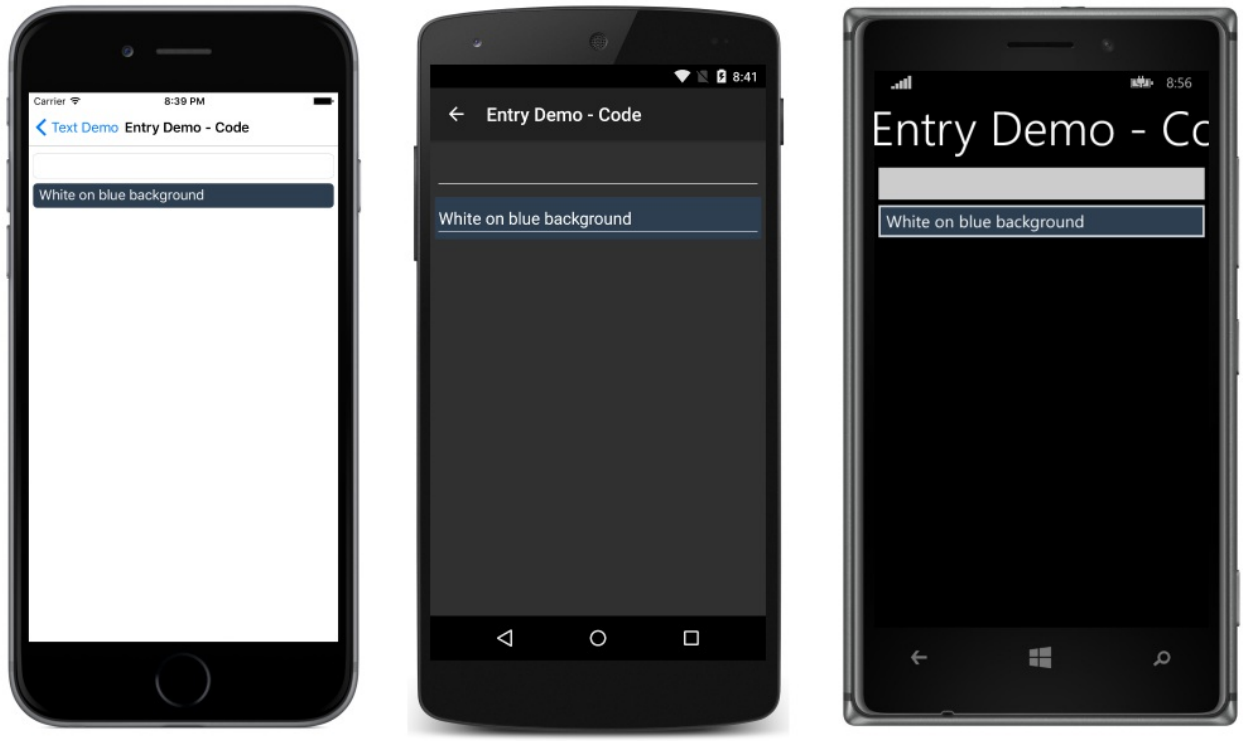
请注意，占位符不是受影响的指定 `TextColor` 。

若要在 XAML 中设置的背景色：

```
<Entry BackgroundColor="#2c3e50" />
```

在 C# 中：

```
var entry = new Entry();
entry.BackgroundColor = Color.FromHex("#2c3e50");
```



请务必确保您选择的背景和文本颜色每个平台上可用，并且变得模糊的任何占位符文本。

事件和交互性

项公开两个事件：

- `TextChanged` – 文本更改的项中时引发。更改之前和之后提供的文本。
- `Completed` – 当用户已结束输入通过键盘上按 return 键时引发。

已完成

`Completed` 使用事件做出反应的条目的交互完成。`Completed` 当用户通过键盘上按 return 键结束与某个字段的输入时引发。该事件的处理程序是一个泛型事件处理程序，使发件人和 `EventArgs`：

```
void Entry_Completed (object sender, EventArgs e)
{
    var text = ((Entry)sender).Text; //cast sender to access the properties of the Entry
}
```

可以在 XAML 中订阅的已完成的事件：

```
<Entry Completed="Entry_Completed" />
```

和 C#:

```
var entry = new Entry ();
entry.Completed += Entry_Completed;
```

之后 `Completed` 事件触发时，任何 `ICommand` 指定的 `ReturnCommand` 执行属性时，使用 `object` 指定 `ReturnCommandParameter` 属性传递给 `ICommand`。

TextChanged

`TextChanged` 事件用于对字段的内容中更改做出反应。

`TextChanged` 时引发 `Text` 的 `Entry` 更改。为事件处理程序将执行的实例 `TextChangedEventArgs` 。 `TextChangedEventArgs` 提供对旧的和新值的访问 `Entry``Text` 通过 `OldTextValue` 和 `NewTextValue` 属性：

```
void Entry_TextChanged (object sender, TextChangedEventArgs e)
{
    var oldText = e.OldTextValue;
    var newText = e.NewTextValue;
}
```

`TextChanged` 事件可以在 XAML 中进行订阅：

```
<Entry TextChanged="Entry_TextChanged" />
```

和 C#:

```
var entry = new Entry ();
entry.TextChanged += Entry_TextChanged;
```

相关链接

- 文本（示例）
- 入口 API

Xamarin.Forms 编辑器

2018/10/25 • • Edit Online

多行文本输入

`Editor` 控件用来接受多行输入。本文包含以下内容：

- **自定义** –键盘和颜色选项。
- **交互性** –可以将侦听的事件，以提供交互性。

自定义

设置和读取文本

`Editor`，如其他文本呈现的视图，公开 `Text` 属性。此属性可以用于设置和读取通过显示的文本 `Editor`。下面的示例演示了如何设置 `Text` 在 XAML 中的属性：

```
<Editor Text="I am an Editor" />
```

在 C# 中：

```
var MyEditor = new Editor { Text = "I am an Editor" };
```

若要读取的文本，访问 `Text` C# 中的属性：

```
var text = MyEditor.Text;
```

输入的长度限制

`MaxLength` 属性可用于限制所允许的范围的输入的长度 `Editor`。此属性应设置为一个正整数：

```
<Editor ... MaxLength="10" />
```

```
var editor = new Editor { ... MaxLength = 10 };
```

一个 `MaxLength` 属性值为 0 指示将允许任何输入，并将值 `int.MaxValue`，它是默认值，`Editor`，指示是否有任何可能输入的字符数的有效限制。

自动调整大小编辑器

`Editor` 可以对通过设置其内容自动调整大小 `Editor.AutoSize` 属性设置为 `TextChanges`，它是一个值 `EditoAutoSizeOption` 枚举。此枚举有两个值：

- `Disabled` 指示自动调整大小处于禁用状态，并且是默认值。
- `TextChanges` 指示启用自动调整大小。

这可以在代码中完成，如下所示：

```
<Editor Text="Enter text here" AutoSize="TextChanges" />
```

```
var editor = new Editor { Text = "Enter text here", AutoSize = EditorAutoSizeOption.TextChanges };
```

当启用自动调整大小、的高度 `Editor` 用户将其填充文本，并且高度会降低用户删除文本时，将提高。

> **NOTE**
> `Editor` 将不自动调整大小 if `HeightRequest` 设置属性。

自定义键盘

当与用户交互时显示键盘 `Editor` 可以通过编程方式设置 `Keyboard` 属性，因为的以下属性之一 `Keyboard` 类：

- `Chat` – 用于短和表情符号是有用的地方。
- `Default` – 默认键盘。
- `Email` – 输入电子邮件地址时使用。
- `Numeric` – 输入数字时使用。
- `Plain` – 使用输入文本，而无需任何时 `KeyboardFlags` 指定。
- `Telephone` – 使用输入电话号码时。
- `Text` – 输入文本时使用。
- `Url` – 用于输入文件路径（& a）的 web 地址。

这可以实现在 XAML 中，如下所示：

```
<Editor Keyboard="Chat" />
```

等效的 C# 代码是：

```
var editor = new Editor { Keyboard = Keyboard.Chat };
```

可以在中找到的每个键盘示例我们配方存储库。

`Keyboard` 类还具有 `Create` 工厂方法，可以用来通过指定的大小写、拼写检查，并建议行为自定义键盘。 `KeyboardFlags` 枚举值指定为方法，为自定义的自变量 `Keyboard` 返回。 `KeyboardFlags` 枚举包含的以下值：

- `None` – 没有功能添加到键盘。
- `CapitalizeSentence` – 指示每个输入的句子的第一个单词的第一个字母将自动大写。
- `Spellcheck` – 指示该拼写检查功能将在输入的文本上执行。
- `Suggestions` – 指示该单词上输入的文本，则将提供完成。
- `CapitalizeWord` – 指示每个单词的第一个字母将自动大写。
- `CapitalizeCharacter` – 指示每个字符将自动大写。
- `CapitalizeNone` -指示没有自动大写会发生。
- `All` – 指示拼写检查、第完成单词和句子大小写会在输入的文本上发生。

以下 XAML 代码示例演示如何自定义默认值 `Keyboard` 提供完成单词和每个输入的字符大写：

```
<Editor>
    <Editor.Keyboard>
        <Keyboard x:FactoryMethod="Create">
            <x:Arguments>
                <KeyboardFlags>Suggestions,CapitalizeCharacter</KeyboardFlags>
            </x:Arguments>
        </Keyboard>
    </Editor.Keyboard>
</Editor>
```

等效的 C# 代码是：

```
var editor = new Editor();
editor.Keyboard = Keyboard.Create(KeyboardFlags.Suggestions | KeyboardFlags.CapitalizeCharacter);
```

启用和禁用拼写检查

`IsSpellCheckEnabled` 属性控制是否拼写检查已启用。默认情况下，该属性设置为 `true`。当用户输入文本，指示拼写错误。

但是，对于某些文本项方案中，输入用户名，如拼写检查功能提供了负的体验，因此应禁用通过设置 `IsSpellCheckEnabled` 属性设置为 `false`：

```
<Editor ... IsSpellCheckEnabled="false" />
```

```
var editor = new Editor { ... IsSpellCheckEnabled = false };
```

> **NOTE**
> 当 `IsSpellCheckEnabled` 属性设置为 `false`，并自定义键盘未被使用，将禁用本机拼写检查器。但是，如果 `Keyboard` 具有已设置，以禁用拼写检查，如 `Keyboard.Chat`，则 `IsSpellCheckEnabled` 属性将被忽略。因此，该属性不能用于启用拼写检查 `Keyboard` 的显式禁用它。

设置占位符文本

`Editor` 可以设置为显示占位符文本，它不存储用户输入时。这可以通过设置 `Placeholder` 属性设置为 `string`，并通常用于指示的是适用于的内容类型 `Editor`。此外，通过设置控制的占位符文本颜色 `PlaceholderColor` 属性设置为 `Color`：

```
<Editor Placeholder="Enter text here" PlaceholderColor="Olive" />
```

```
var editor = new Editor { Placeholder = "Enter text here", PlaceholderColor = Color.Olive };
```

颜色

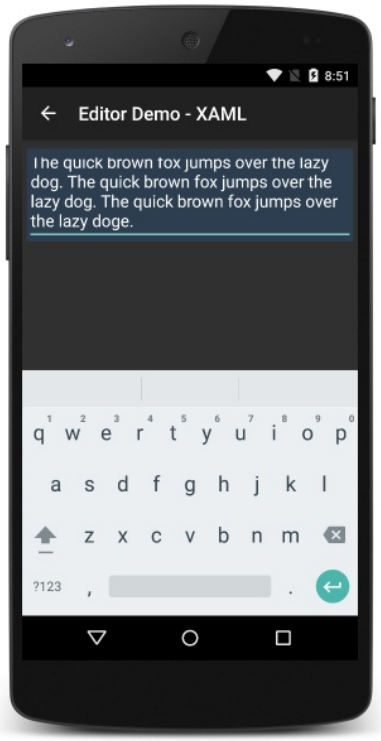
`Editor` 可以设置为使用自定义背景色通过 `BackgroundColor` 属性。特别注意有必要确保将每个平台上可用的颜色。因为每个平台都有不同的文本颜色的默认值，可能需要设置每个平台的自定义背景颜色。请参阅使用平台调整有关优化每个平台的 UI 的详细信息。

在 C# 中：

```
public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        //dark blue on UWP & Android, light blue on iOS
        var editor = new Editor { BackgroundColor = Device.RuntimePlatform == Device.iOS ?
Color.FromHex("#A4EAFF") : Color.FromHex("#2c3e50") };
        layout.Children.Add(editor);
    }
}
```

在 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="TextSample.EditorPage"
    Title="Editor Demo">
    <ContentPage.Content>
        <StackLayout Padding="5,10">
            <Editor>
                <Editor.BackgroundColor>
                    <OnPlatform x:TypeArguments="x:Color">
                        <On Platform="iOS" Value="#a4eaff" />
                        <On Platform="Android, UWP" Value="#2c3e50" />
                    </OnPlatform>
                </Editor.BackgroundColor>
            </Editor>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```



请确保你选择的背景和文本颜色为每个平台上可用，且变得模糊的任何占位符文本。

交互性

`Editor` 公开两个事件：

- TextChanged –文本在编辑器中更改时引发。更改之前和之后提供的文本。
- 已完成–用户已结束输入通过键盘上按 return 键时引发。

已完成

`Completed` 事件用来与的交互完成做出反应 `Editor` 。 `Completed` 当用户通过键盘上输入 return 键结束与某个字段的输入时引发。该事件的处理程序是一个泛型事件处理程序，使发件人和 `EventArgs` ：

```
void EditorCompleted (object sender, EventArgs e)
{
    var text = ((Editor)sender).Text; // sender is cast to an Editor to enable reading the `Text` property of
the view.
}
```

可以在代码和 XAML 中订阅的已完成的事件：

在 C# 中：

```
public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        var editor = new Editor ();
        editor.Completed += EditorCompleted;
        layout.Children.Add(editor);
    }
}
```

在 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TextSample.EditorPage"
Title="Editor Demo">
    <ContentPage.Content>
        <StackLayout Padding="5,10">
            <Editor Completed="EditorCompleted" />
        </StackLayout>
    </ContentPage.Content>
</Contentpage>
```

TextChanged

`TextChanged` 事件用于对字段的内容中更改做出反应。

`TextChanged` 时引发 `Text` 的 `Editor` 更改。为事件处理程序将执行的实例 `TextChangedEventArgs` 。
`TextChangedEventArgs` 提供对旧的和新值的访问 `Editor``Text` 通过 `OldTextValue` 和 `NewTextValue` 属性：

```
void EditorTextChanged (object sender, TextChangedEventArgs e)
{
    var oldText = e.OldTextValue;
    var newText = e.NewTextValue;
}
```

可以在代码和 XAML 中订阅的已完成的事件：

在代码中：

```
public partial class EditorPage : ContentPage
{
    public EditorPage ()
    {
        InitializeComponent ();
        var layout = new StackLayout { Padding = new Thickness(5,10) };
        this.Content = layout;
        var editor = new Editor ();
        editor.TextChanged += EditorTextChanged;
        layout.Children.Add(editor);
    }
}
```

在 XAML:

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="TextSample.EditorPage"
Title="Editor Demo">
    <ContentPage.Content>
        <StackLayout Padding="5,10">
            <Editor TextChanged="EditorTextChanged" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

相关链接

- 文本（示例）
- 编辑器 API

在 Xamarin.Forms 中的字体

2018/7/24 • • Edit Online

本文介绍了 Xamarin.Forms 如何允许您指定的字体特性（包括权重和大小）上显示文本的控件。字体信息可以是在代码中指定或在 XAML 中指定。它也是可以使用自定义字体。

在代码中设置字体

使用任何控件的显示文本的三个与字体相关的属性：

- **FontFamily** – `string` 字体名称。
- **FontSize** –形式的字体大小 `double` 。
- **FontAttributes** –一个字符串，指定样式信息，如*斜体*并**加粗**(使用 `FontAttributes` C# 中的枚举)。

此代码演示如何创建一个标签，并指定的字体大小和权重来显示：

```
var about = new Label {
    FontSize = Device.GetNamedSize (NamedSize.Medium, typeof(Label)),
    FontAttributes = FontAttributes.Bold,
    Text = "Medium Bold Font"
};
```

字号

`FontSize` 属性可以设置为双精度值，例如：

```
label.FontSize = 24;
```

此外可以使用 `NamedSize` 枚举具有四个内置选项;Xamarin.Forms 选择每个平台的最佳大小。

- **Micro**
- **小**
- **中等**
- **大型**

`NamedSize` 枚举可以是任何位置使用 `FontSize` 可以使用指定 `Device.GetNamedSize` 方法将值转换为 `double` ：

```
label.FontSize = Device.GetNamedSize(NamedSize.Small, typeof(Label));
```

字体特性

字体样式等**粗体**并*斜体*可以设置 `FontAttributes` 属性。目前支持以下值：

- **无**
- **加粗**
- **斜体**

`FontAttribute` 可以按如下所示使用枚举 (您可以指定单个属性或 `OR` 它们组合在一起)：

```
label.FontAttributes = FontAttributes.Bold | FontAttributes.Italic;
```

设置每个平台的字体信息

或者，`Device.RuntimePlatform`属性可以用于设置不同的字体名称在每个平台上，此代码中所示：

```
label.FontFamily = Device.RuntimePlatform == Device.iOS ? "Lobster-Regular" :
   Device.RuntimePlatform == Device.Android ? "Lobster-Regular.ttf#Lobster-Regular" :
"Assets/Fonts/Lobster-Regular.ttf#Lobster",
label.FontSize = Device.RuntimePlatform == Device.iOS ? 24 :
   Device.RuntimePlatform == Device.Android ? Device.GetNamedSize(NamedSize.Medium, label) :
Device.GetNamedSize(NamedSize.Large, label);
```

是一个很好的适用于 iOS 的字体信息的来源iosfonts.com。

在 XAML 中设置该字体

Xamarin.Forms 控制所有具有该显示文本`Font`可以在 XAML 中设置的属性。在 XAML 中设置该字体的最简单方法是使用命名的大小的枚举值，在此示例中所示：

```
<Label Text="Login" FontSize="Large"/>
<Label Text="Instructions" FontSize="Small"/>
```

没有内置的转换器，用于`Font`允许所有字体设置表示为 XAML 中的字符串值的属性。下面的示例演示如何可以在 XAML 中指定的字体属性和大小：

```
<Label Text="Italics are supported" FontAttributes="Italic" />
<Label Text="Biggest NamedSize" FontSize="Large" />
<Label Text="Use size 72" FontSize="72" />
```

若要指定多个`Font`设置，将合并到单个所需的设置`Font`属性字符串。字体属性字符串的格式应为`"[font-face],[attributes],[size]"`。参数的顺序很重要，所有参数都是可选的并且多个`attributes`可以指定，例如：

```
<Label Text="Small bold text" Font="Bold, Micro" />
<Label Text="Medium custom font" Font="MarkerFelt-Thin, 42" />
<Label Text="Really big bold and italic text" Font="Bold, Italic, 72"  />
```

`Device.RuntimePlatform`此外可在 XAML 中呈现每个平台上不同的字体。下面的示例在 iOS 上使用自定义字体 (MarkerFelt 精简)，并在其他平台上指定仅大小/属性：

```
<Label Text="Hello Forms with XAML">
    <Label.FontFamily>
        <OnPlatform x:TypeArguments="x:String">
                <On Platform="iOS" Value="MarkerFelt-Thin" />
                <On Platform="Android" Value="Lobster-Regular.ttf#Lobster-Regular" />
                <On Platform="UWP" Value="Assets/Fonts/Lobster-Regular.ttf#Lobster" />
        </OnPlatform>
    </Label.FontFamily>
</Label>
```

指定自定义字体，它是始终使用一个好办法`OnPlatform`，因为很难找到可在所有平台的字体。

使用自定义字体

使用非内置字样的字体，则需要一些特定于平台的编码。此屏幕截图显示自定义字体**Lobster**从Google 的开放源代码字体呈现使用 Xamarin.Forms。

为每个平台所需的步骤如下所述。包含与应用程序的自定义字体文件时，请务必验证字体的许可证进行分发。

iOS

可以通过首先确保将加载它，然后使用 Xamarin.Forms 按名称引用它来显示自定义字体 `Font` 方法。按照中的说明这篇博客文章:

1. 添加字体文件**生成操作**：**BundleResource**，和
2. 更新**Info.plist**文件 (**提供的应用程序字体**，或 `UIAppFonts` 、密钥)，然后
3. 它按名称引用任何在 Xamarin.Forms 中定义一种字体位置 ！

```
new Label
{
    Text = "Hello, Forms!",
    FontFamily = Device.RuntimePlatform == Device.iOS ? "Lobster-Regular" : null // set only for iOS
}
```

Android

适用于 Android 的 Xamarin.Forms 可以引用按照特定的命名标准添加到项目的自定义字体。首先添加字体文件的资产文件夹中的应用程序项目并设置*生成操作：AndroidAsset*。然后，使用的完整路径和*字体名称*，作为在 Xamarin.Forms 中，字体名称的哈希 (#) 分隔，如以下代码段演示了：

```
new Label
{
  Text = "Hello, Forms!",
  FontFamily = Device.RuntimePlatform == Device.Android ? "Lobster-Regular.ttf#Lobster-Regular" : null //
set only for Android
}
```

Windows

适用于 Windows 平台的 Xamarin.Forms 可以引用按照特定的命名标准添加到项目的自定义字体。首先添加字体文件的 **/Assets 字体/** 文件夹中的应用程序项目并设置生成操作：内容。然后，使用的完整路径和字体文件名后，跟哈希 (#) 和字体名称，如以下代码段所示：

```
new Label
{
    Text = "Hello, Forms!",
    FontFamily = Device.RuntimePlatform == Device.UWP ? "Assets/Fonts/Lobster-Regular.ttf#Lobster" : null
// set only for UWP apps
}
```

> **NOTE**
> 请注意，但使用的字体文件名称和字体名称可能不同。若要发现 Windows 上的字体名称，右键单击.ttf 文件，然后选择预览**版**。然后可以从预览窗口中确定的字体名称。

此时完成了应用程序的常用代码。此时，可将特定于平台的电话拨号程序实现为 DependencyService。

XAML

此外可以使用 `Device.RuntimePlatform` 中 XAML 呈现自定义字体：

```
<Label Text="Hello Forms with XAML">
    <Label.FontFamily>
        <OnPlatform x:TypeArguments="x:String">
                <On Platform="iOS" Value="Lobster-Regular" />
                <On Platform="Android" Value="Lobster-Regular.ttf#Lobster-Regular" />
                <On Platform="UWP" Value="Assets/Fonts/Lobster-Regular.ttf#Lobster" />
        </OnPlatform>
    </Label.FontFamily>
</Label>
```

总结

Xamarin.Forms 提供了简单的默认设置，以便你可以轻松地为所有支持的平台的文本的大小。它还允许你指定的字体和大小–甚至以不同方式为每个平台–需要更精细的控制时。

此外可以使用格式正确的字体属性的 XAML 中指定字体信息。

相关链接

- FontsSample
- 文本（示例）

Xamarin.Forms 文本样式

2018/7/13 • • Edit Online

在 Xamarin.Forms 中的文本样式

样式可用于调整标签、条目和编辑器的外观。样式可以定义一次并使用的许多视图，但一种样式仅用于一种类型的视图。可以提供样式 `Key` 有选择地使用特定的控件和应用 `Style` 属性。

内置样式

Xamarin.Forms 包含多个内置样式的常见方案：

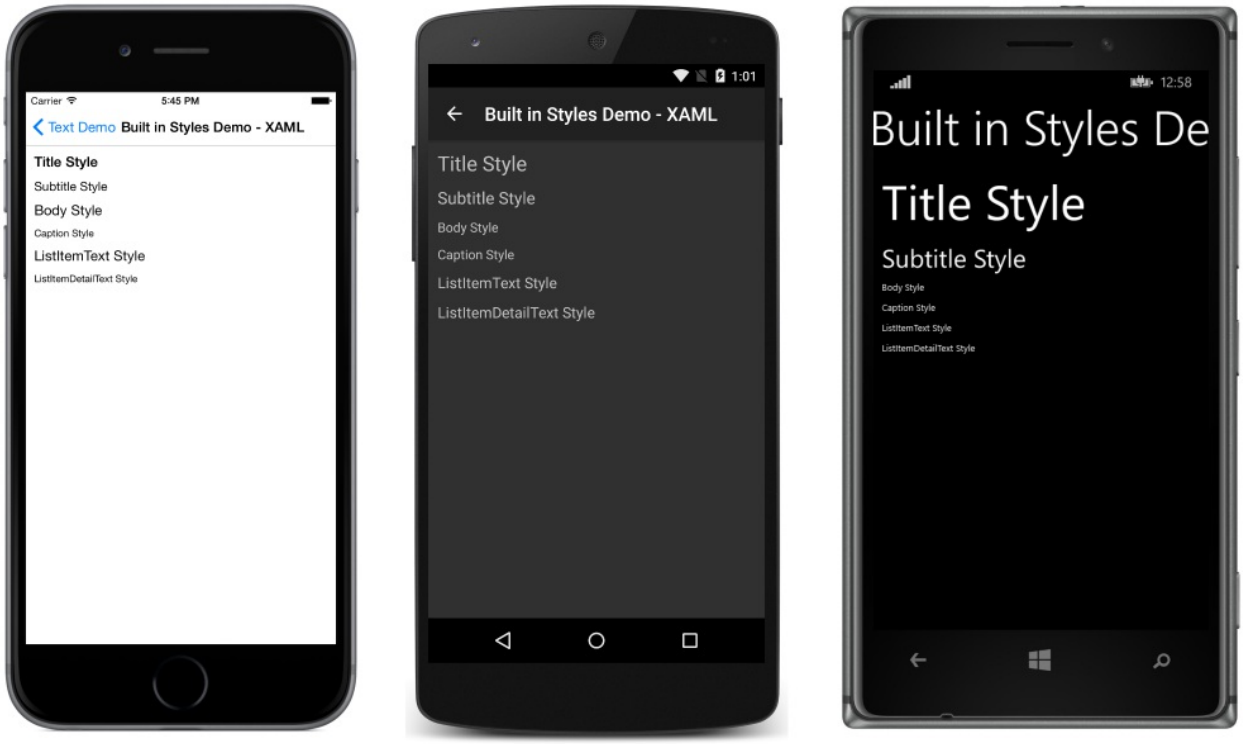
- `BodyStyle`
- `CaptionStyle`
- `ListItemDetailTextStyle`
- `ListItemTextStyle`
- `SubtitleStyle`
- `TitleStyle`

若要将其中一个内置样式应用，使用 `DynamicResource` 标记扩展来指定的样式：

```
<Label Text="I'm a Title" Style="{DynamicResource TitleStyle}"/>
```

在 C# 中，内置样式从选择 `Device.Styles`：

```
label.Style = Device.Styles.TitleStyle;
```



自定义样式

样式包含的资源库和资源库包含的属性和属性的值将设置为。

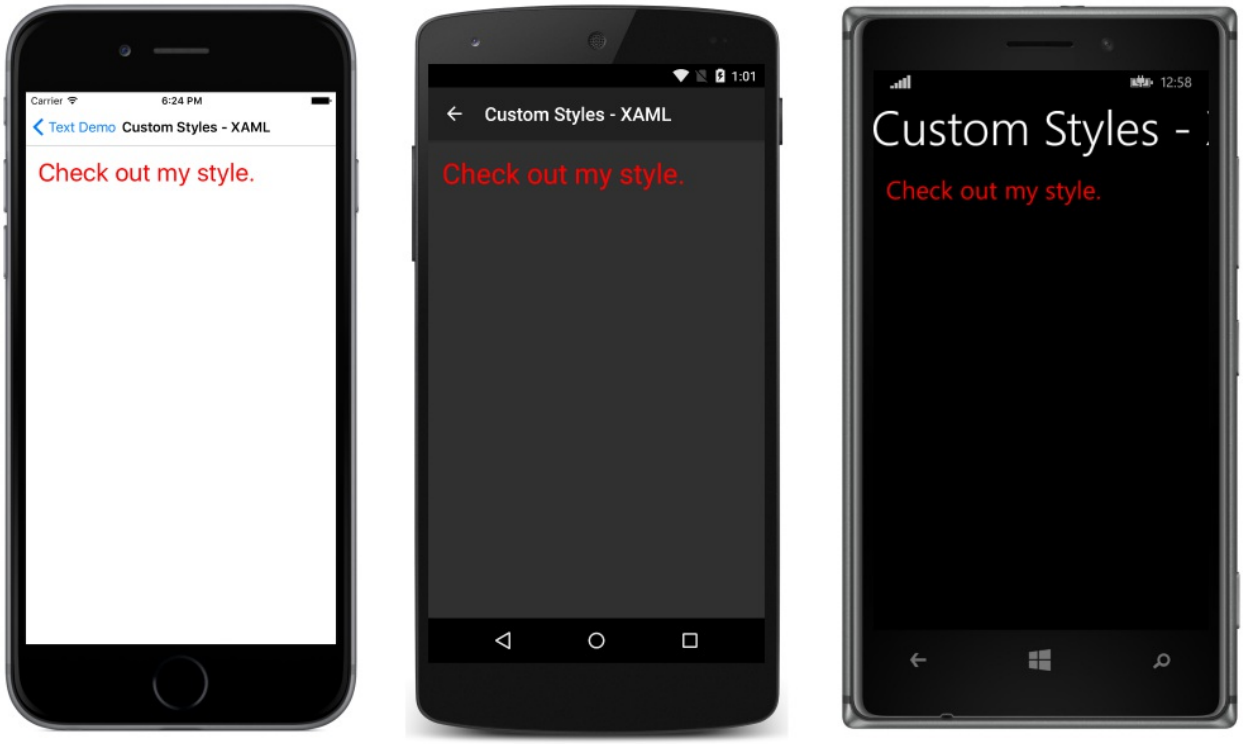
在 C# 中，将按如下所示定义自定义大小 30 的红色文本的标签的样式：

```
var LabelStyle = new Style (typeof(Label)) {
    Setters = {
        new Setter {Property = Label.TextColorProperty, Value = Color.Red},
        new Setter {Property = Label.FontSizeProperty, Value = 30}
    }
};

var label = new Label { Text = "Check out my style.", Style = LabelStyle };
```

在 XAML：

```
<ContentPage.Resources>
    <ResourceDictionary>
        <Style x:Key="LabelStyle" TargetType="Label">
            <Setter Property="TextColor" Value="Red"/>
            <Setter Property="FontSize" Value="30"/>
        </Style>
    </ResourceDictionary>
</ContentPage.Resources>

<ContentPage.Content>
    <StackLayout>
        <Label Text="Check out my style." Style="{StaticResource LabelStyle}" />
    </StackLayout>
</ContentPage.Content>
```

请注意资源（包括所有样式）定义内 `ContentPage.Resources`，这是更熟悉的同级 `ContentPage.Content` 元素。



应用样式

一旦创建一个样式后，它可以应用于任何视图匹配其 `TargetType`。

在 XAML 中，自定义样式应用于视图通过提供他们 `Style` 具有属性 `StaticResource` 标记扩展引用所需的样式：

```
<Label Text="Check out my style." Style="{StaticResource LabelStyle}" />
```

在 C# 中，样式可以是直接应用于视图或添加到并从页面的检索 `ResourceDictionary` 。若要直接添加：

```
var label = new Label { Text = "Check out my style.", Style = LabelStyle };
```

若要添加和检索从页面的 `ResourceDictionary` ：

```
this.Resources.Add ("LabelStyle", LabelStyle);
label.Style = (Style)Resources["LabelStyle"];
```

内置样式的应用方式不同，，因为它们需要响应的辅助功能设置。若要将应用中 XAML，内置样式 `DynamicResource` 使用标记扩展：

```
<Label Text="I'm a Title" Style="{DynamicResource TitleStyle}"/>
```

在 C# 中，内置样式从选择 `Device.Styles` ：

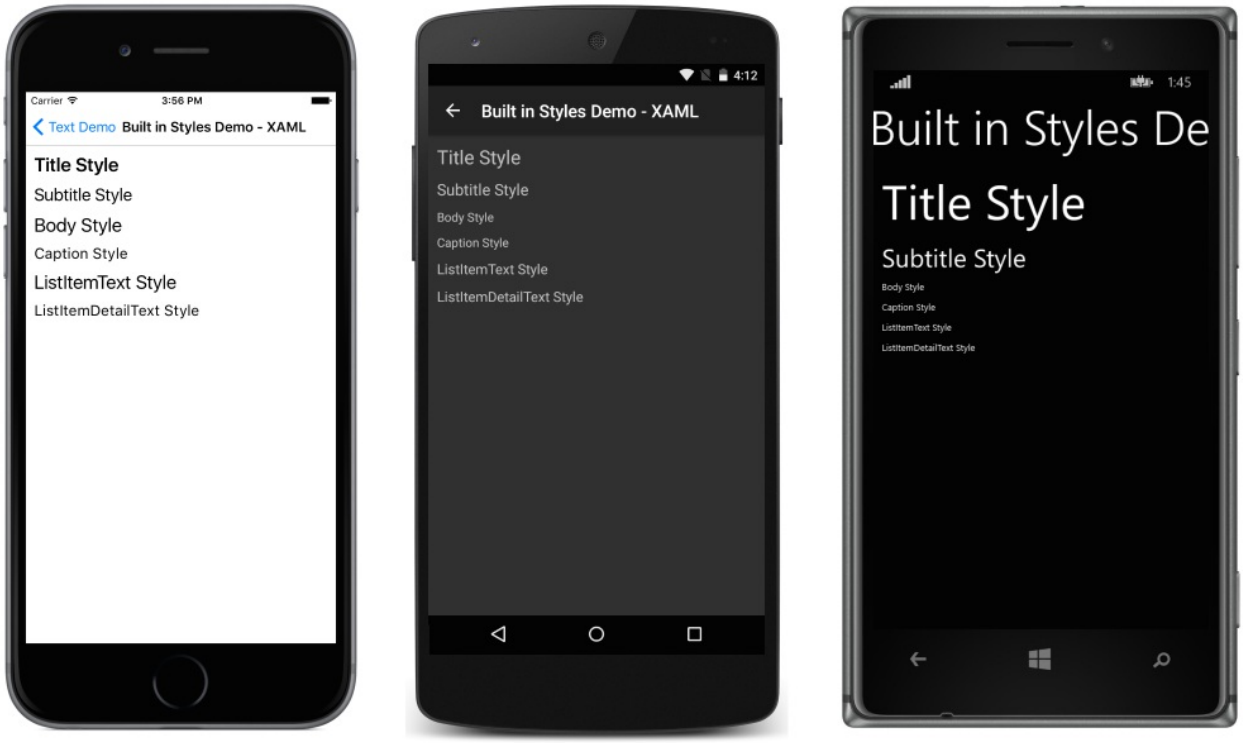
```
label.Style = Device.Styles.TitleStyle;
```

可访问性

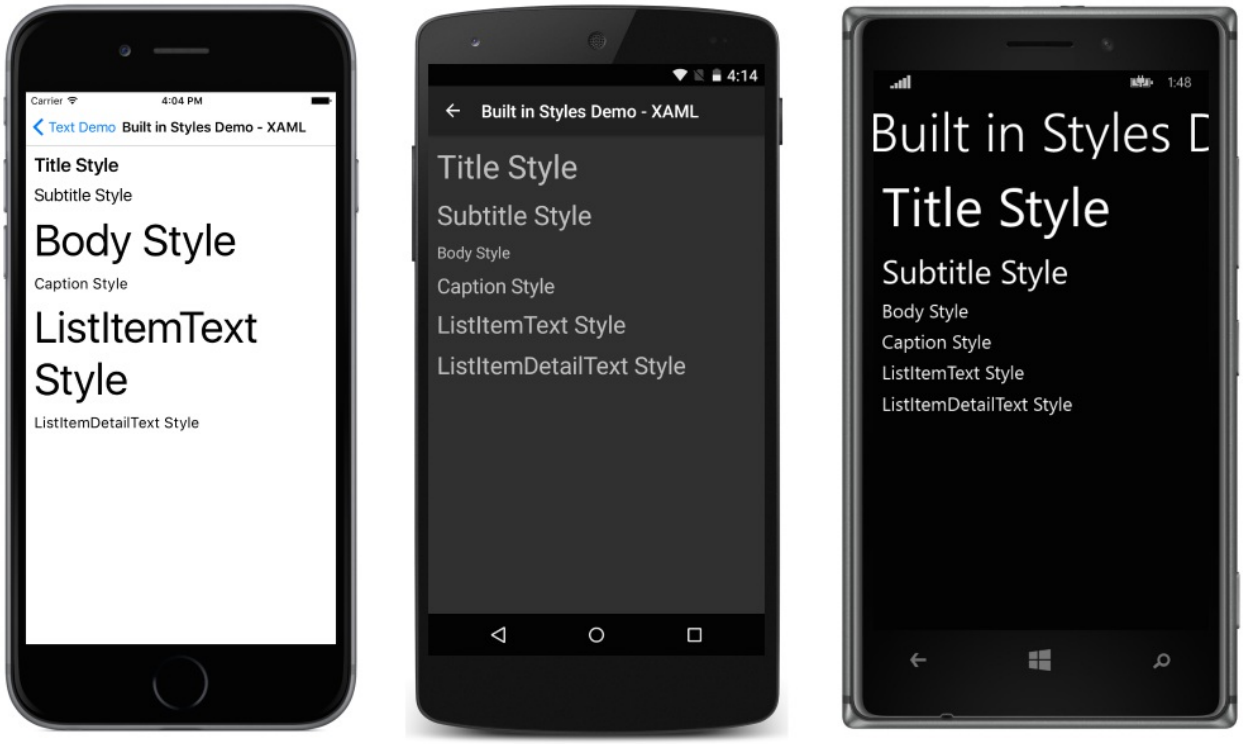
内置样式存在是为了使其更易于尊重辅助工具首选项。在使用任何内置样式时，如果用户将相应地设置其可访问性首选项将自动增加字体大小。

请考虑在同一页的视图设置样式与内置样式与辅助功能设置启用和禁用下面的示例：

已禁用：



已启用:



若要确保可访问性，请确保在应用中，任何与文本相关样式的基础使用内置样式一致地使用样式。请参阅样式有关扩展和一般情况下使用样式的详细信息。

相关链接

- 借助 Xamarin.Forms，第 12 章创建移动应用
- 样式
- 文本（示例）
- 样式

Xamarin.Forms 主题

2018/7/12 • • Edit Online

Xamarin.Forms 主题曾宣布在 Evolve 2016，并可作为客户试用并提供反馈的预览。

主题将添加到 Xamarin.Forms 应用程序中包括**Xamarin.Forms.Theme.Base** Nuget 包，以及其他包，用于定义特定的主题（例如。Xamarin.Forms.Theme.Light）或其他本地主题可以定义应用程序。

请参阅浅色主题并深色主题如何将它们添加到应用程序，或查看说明页示例自定义主题。

重要说明：还应遵循的步骤加载程序集（下面）为主题通过将一些样本代码添加到 iOS `AppDelegate` 和 Android `MainActivity`。这将在将来的预览版的版本中得到改进。

控件外观

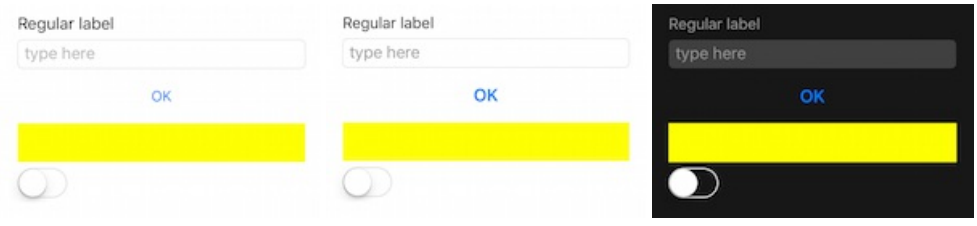
Light并深色这两个主题定义特定的可视化外观，对标准控件。一旦应用程序的资源字典中添加了一个主题，将更改标准控件的外观。

下面的 XAML 标记显示了一些常见的控件：

```
<StackLayout Padding="40">
    <Label Text="Regular label" />
    <Entry Placeholder="type here" />
    <Button Text="OK" />
    <BoxView Color="Yellow" />
    <Switch />
</StackLayout>
```

这些屏幕截图显示了与这些控件：

- 应用任何主题
- 浅色主题（仅为具有无主题的细微差异）
- 深色主题



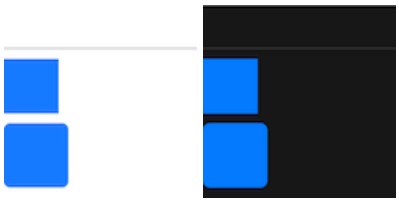
StyleClass

`StyleClass` 属性允许视图的外观，以根据提供的主题的定义进行更改。

Light并深色这两个主题定义的三个不同的外观 `BoxView`：`HorizontalRule`，`Circle`，和 `Rounded`。此标记显示三个不同 `BoxView`s，使用不同的样式类应用：

```
<StackLayout Padding="40">
    <BoxView StyleClass="HorizontalRule" />
    <BoxView StyleClass="Circle" />
    <BoxView StyleClass="Rounded" />
</StackLayout>
```

这会使使用浅色和深色，如下所示：



内置类

除了自动设置样式公共控件光线和深色主题目前支持可通过设置应用的以下类 `StyleClass` 这些控件上：

BoxView

- HorizontalRule
- 圆圈
- 舍入

Image

- 圆圈
- 舍入
- 缩略图

Button

- 默认
- 基本
- 成功
- T:System.Diagnostics.Switch
- 警告
- 危险
- 链接
- 小
- 大型

标签

- Header
- 小标题
- 正文
- 链接
- 反函数

疑难解答

无法加载文件或程序集 Xamarin.Forms.Theme.Light 或其某个依赖项

在预览版本中，主题可能不能在运行时加载。添加代码，如下所示在相关项目来修复此错误。

iOS

在中**AppDelegate.cs**添加以下行后 `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.iOS.UnderlineEffect);
```

Android

在中**MainActivity.cs**添加以下行后 `LoadApplication`

```
var x = typeof(Xamarin.Forms.Themes.DarkThemeResources);
x = typeof(Xamarin.Forms.Themes.LightThemeResources);
x = typeof(Xamarin.Forms.Themes.Android.UnderlineEffect);
```

相关链接

- ThemesDemo 示例

Xamarin.Forms 浅色主题

2018/7/12 • • Edit Online

> **NOTE**
> 主题需要 Xamarin.Forms 2.3 预览版本。检查故障排除提示如果出现错误。

若要使用浅色主题：

1.添加 Nuget 包

- Xamarin.Forms.Theme.Base
- Xamarin.Forms.Theme.Light

2.将添加到资源字典

在中**App.xaml**文件并添加新的自定义 `xmlns` 主题，然后确保与应用程序的资源字典合并主题的资源。XAML 文件的示例如下所示：

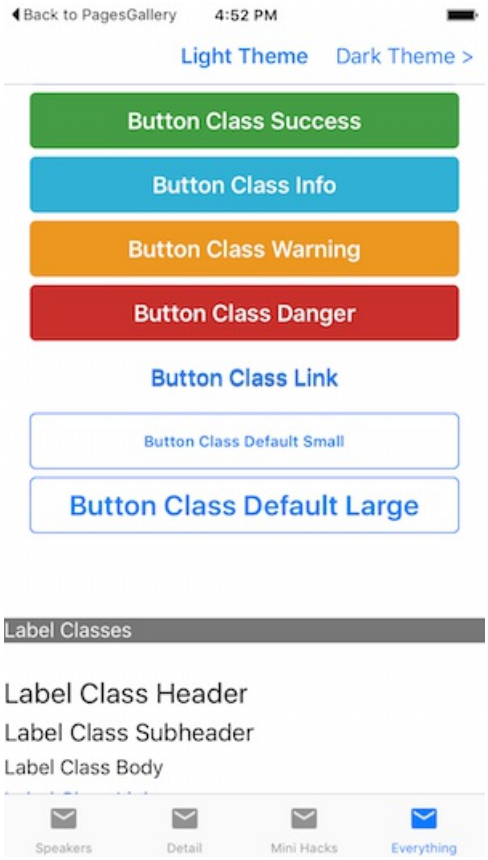
```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="EvolveApp.App"
             xmlns:light="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Light">
    <Application.Resources>
        <ResourceDictionary MergedWith="light:LightThemeResources" />
    </Application.Resources>
</Application>
```

3.加载主题类

按照这故障排除步骤，并在 iOS 和 Android 应用程序项目中添加所需的代码。

4.使用 StyleClass

下面是按钮和标签浅色主题，以及生成它们的标记中的示例。




```
<StackLayout Padding="20">
    <Button Text="Button Default" />
    <Button Text="Button Class Default" StyleClass="Default" />
    <Button Text="Button Class Primary" StyleClass="Primary" />
    <Button Text="Button Class Success" StyleClass="Success" />
    <Button Text="Button Class Info" StyleClass="Info" />
    <Button Text="Button Class Warning" StyleClass="Warning" />
    <Button Text="Button Class Danger" StyleClass="Danger" />
    <Button Text="Button Class Link" StyleClass="Link" />
    <Button Text="Button Class Default Small" StyleClass="Small" />
    <Button Text="Button Class Default Large" StyleClass="Large" />
</StackLayout>
```

内置类的完整列表显示哪些样式是可用于某些常见的控件。

Xamarin.Forms 深色主题

2018/7/12 • • Edit Online

> **NOTE**
> 主题需要 Xamarin.Forms 2.3 预览版本。检查故障排除提示如果出现错误。

若要使用深色主题：

1.添加 Nuget 包

- Xamarin.Forms.Theme.Base
- Xamarin.Forms.Theme.Dark

2.将添加到资源字典

在中**App.xaml**文件并添加新的自定义 `xmlns` 主题，然后确保与应用程序的资源字典合并主题的资源。XAML 文件的示例如下所示：

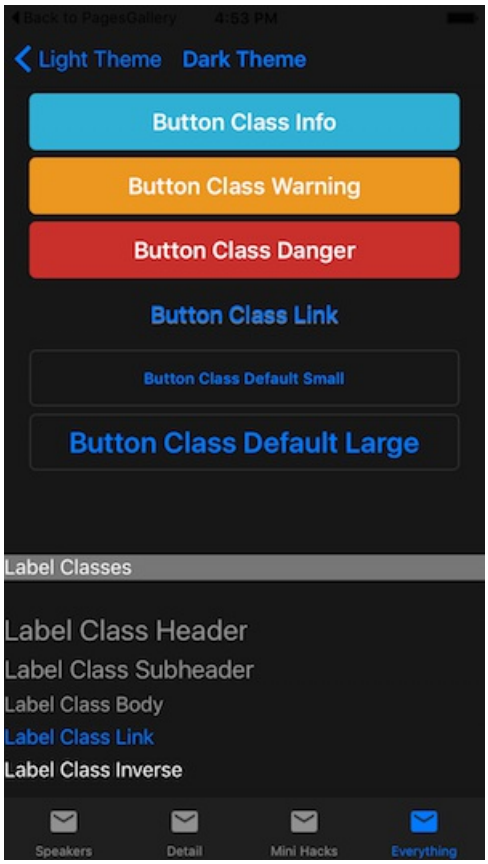
```
<?xml version="1.0" encoding="utf-8"?>
<Application xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml" x:Class="EvolveApp.App"
             xmlns:dark="clr-namespace:Xamarin.Forms.Themes;assembly=Xamarin.Forms.Theme.Dark">
    <Application.Resources>
        <ResourceDictionary MergedWith="dark:DarkThemeResources" />
    </Application.Resources>
</Application>
```

3.加载主题类

按照这故障排除步骤，并在 iOS 和 Android 应用程序项目中添加所需的代码。

4.使用 StyleClass

下面是按钮和标签在深色主题，以及生成它们的标记的示例。




```
<StackLayout Padding="20">
    <Button Text="Button Default" />
    <Button Text="Button Class Default" StyleClass="Default" />
    <Button Text="Button Class Primary" StyleClass="Primary" />
    <Button Text="Button Class Success" StyleClass="Success" />
    <Button Text="Button Class Info" StyleClass="Info" />
    <Button Text="Button Class Warning" StyleClass="Warning" />
    <Button Text="Button Class Danger" StyleClass="Danger" />
    <Button Text="Button Class Link" StyleClass="Link" />

    <Button Text="Button Class Default Small" StyleClass="Small" />
    <Button Text="Button Class Default Large" StyleClass="Large" />
</StackLayout>
```

内置类的完整列表显示哪些样式是可用于某些常见的控件。

创建 Xamarin.Forms 自定义主题

2018/11/1 • • Edit Online

除了从 Nuget 包添加主题 (如Light并深色主题)，可以创建自己的资源字典主题可以引用的应用程序中。

示例

这三个 `BoxView` 上显示的 s主题页根据两个可下载的主题中定义的三个类样式。

若要了解工作原理，下面的标记创建等效的样式，您可以将直接添加到您**App.xaml**。

请注意 `Class` 属性 `Style` (而不是 `x:Key` 属性的 Xamarin.Forms 的早期版本中提供)。

```
<ResourceDictionary>
  <!-- DEFINE ANY CONSTANTS -->
  <Color x:Key="SeparatorLineColor">#CCCCCC</Color>
  <Color x:Key="iOSDefaultTintColor">#007aff</Color>
  <Color x:Key="AndroidDefaultAccentColorColor">#1FAECE</Color>
  <OnPlatform x:TypeArguments="Color" x:Key="AccentColor">
    <On Platform="iOS" Value="{StaticResource iOSDefaultTintColor}" />
    <On Platform="Android" Value="{StaticResource AndroidDefaultAccentColorColor}" />
  </OnPlatform>
  <!--  BOXVIEW CLASSES -->
  <Style TargetType="BoxView" Class="HorizontalRule">
    <Setter Property="BackgroundColor" Value="{ StaticResource SeparatorLineColor }" />
    <Setter Property="HeightRequest" Value="1" />
  </Style>

  <Style TargetType="BoxView" Class="Circle">
    <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />
    <Setter Property="WidthRequest" Value="34"/>
    <Setter Property="HeightRequest" Value="34"/>
    <Setter Property="HorizontalOptions" Value="Start" />

    <Setter Property="local:ThemeEffects.Circle" Value="True" />
  </Style>

  <Style TargetType="BoxView" Class="Rounded">
    <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />
    <Setter Property="HorizontalOptions" Value="Start" />
    <Setter Property="BackgroundColor" Value="{ StaticResource AccentColor }" />

    <Setter Property="local:ThemeEffects.CornerRadius" Value="4" />
  </Style>
</ResourceDictionary>
```

你会看到 `Rounded` 类是指自定义效果 `CornerRadius` 。这种效果的代码下面给出了-若要引用正确自定义 `xmlns` 必须添加到**App.xaml**的根元素：

```
xmlns:local="clr-namespace:ThemesDemo;assembly=ThemesDemo"
```

.NET Standard 库项目或共享项目中的 C# 代码

用于创建轮角的代码 `BoxView` 使用效果。使用应用的圆角半径 `BindableProperty` 并通过应用来实现效果。效果要求中的特定于平台的代码iOS并Android项目（如下所示。）

```
namespace ThemesDemo
{
  public static class ThemeEffects
  {
  public static readonly BindableProperty CornerRadiusProperty =
    BindableProperty.CreateAttached("CornerRadius", typeof(double), typeof(ThemeEffects), 0.0,
propertyChanged: OnChanged<CornerRadiusEffect, double>);
    private static void OnChanged<TEffect, TProp>(BindableObject bindable, object oldValue, object newValue)
              where TEffect : Effect, new()
    {
        if (!(bindable is View view))
        {
            return;
        }

        if (EqualityComparer<TProp>.Equals(newValue, default(TProp)))
        {
            var toRemove = view.Effects.FirstOrDefault(e => e is TEffect);
            if (toRemove != null)
            {
                view.Effects.Remove(toRemove);
            }
        }
        else
        {
            view.Effects.Add(new TEffect());
        }

    }
    public static void SetCornerRadius(BindableObject view, double radius)
    {
        view.SetValue(CornerRadiusProperty, radius);
    }

    public static double GetCornerRadius(BindableObject view)
    {
        return (double)view.GetValue(CornerRadiusProperty);
    }

    private class CornerRadiusEffect : RoutingEffect
    {
        public CornerRadiusEffect()
            : base("Xamarin.CornerRadiusEffect")
        {
        }
    }
  }
}
```

C# 代码中的 iOS 项目

```
using System;
using Xamarin.Forms;
using Xamarin.Forms.Platform.iOS;
using CoreGraphics;
using Foundation;
using XFThemes;

namespace ThemesDemo.iOS
{
    public class CornerRadiusEffect : PlatformEffect
    {
        private nfloat _originalRadius;

        protected override void OnAttached()
        {
            if (Container != null)
            {
                _originalRadius = Container.Layer.CornerRadius;
                Container.ClipsToBounds = true;

                UpdateCorner();
            }
        }

        protected override void OnDetached()
        {
            if (Container != null)
            {
                Container.Layer.CornerRadius = _originalRadius;
                Container.ClipsToBounds = false;
            }
        }

        protected override void OnElementPropertyChanged(System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == ThemeEffects.CornerRadiusProperty.PropertyName)
            {
                UpdateCorner();
            }
        }

        private void UpdateCorner()
        {
            Container.Layer.CornerRadius = (nfloat)ThemeEffects.GetCornerRadius(Element);
        }
    }
}
```

Android 项目中的 C# 代码

```
using System;
using Xamarin.Forms.Platform;
using Xamarin.Forms.Platform.Android;
using Android.Views;
using Android.Graphics;

namespace ThemesDemo.Droid
{
    public class CornerRadiusEffect : BaseEffect
    {
        private ViewOutlineProvider _originalProvider;

        protected override bool CanBeApplied()
        {
            return Container != null && Android.OS.Build.VERSION.SdkInt >=
Android.OS.BuildVersionCodes.Lollipop;
        }

        protected override void OnAttachedInternal()
        {
            _originalProvider = Container.OutlineProvider;
            Container.OutlineProvider = new CornerRadiusOutlineProvider(Element);
            Container.ClipToOutline = true;
        }

        protected override void OnDetachedInternal()
        {
            Container.OutlineProvider = _originalProvider;
            Container.ClipToOutline = false;
        }

        protected override void OnElementPropertyChanged(System.ComponentModel.PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (!Attached)
            {
                return;
            }

            if (args.PropertyName == ThemeEffects.CornerRadiusProperty.PropertyName)
            {
                Container.Invalidate();
            }
        }

        private class CornerRadiusOutlineProvider : ViewOutlineProvider
        {
            private Xamarin.Forms.Element _element;

            public CornerRadiusOutlineProvider(Xamarin.Forms.Element element)
            {
                _element = element;
            }

            public override void GetOutline(Android.Views.View view, Outline outline)
            {
                var pixles =
                    (float)ThemeEffects.GetCornerRadius(_element) *
                    view.Resources.DisplayMetrics.Density;

                outline.SetRoundRect(new Rect(0, 0, view.Width, view.Height), (int)pixles);
            }
        }
    }
}
```

总结

可以通过定义每个需要自定义外观的控件的样式来创建自定义主题。应由不同区分多个控件的样式 `Class` 资源字典中的属性，然后通过设置应用 `StyleClass` 在控件上的属性。

此外可以利用一种样式效果来进一步自定义控件的外观。

隐式样式(不带 `x:Key` 或 `Style` 属性) 继续应用于匹配的所有控件 `TargetType` 。

Xamarin.Forms TimePicker

2018/10/26 • • Edit Online

允许用户选择的时间，一个 Xamarin.Forms 视图。

Xamarin.Forms `TimePicker` 调用平台的时间选取器控件，并允许用户选择的时间。`TimePicker` 定义以下属性：

- `Time` 类型的 `TimeSpan`，默认为在选定时间 `TimeSpan` 为 0。`TimeSpan` 类型指示持续时间为自午夜以来的时间。
- `Format` 类型的 `string`、一个标准或自定义.NET 格式设置字符串，它默认为"t"，短时间模式。
- `TextColor` 类型的 `Color`，用于显示所选的时间，默认为颜色 `Color.Default` 。
- `FontAttributes` 类型的 `FontAttributes`，其默认值为 `FontAtributes.None` 。
- `FontFamily` 类型的 `string`，其默认值为 `null` 。
- `FontSize` 类型的 `double`，其默认值为-1.0。

所有这些属性受到 `BindableProperty` 对象，这意味着它们可以设置的样式，这些属性可以是数据绑定的目标。`Time` 属性设置了默认绑定模式 `BindingMode.TwoWay` ，这意味着它可以是数据绑定中使用的应用程序的目标模型-视图-视图模型 (MVVM)体系结构。

`TimePicker` 不包括一个事件来指示一个新的所选 `Time` 值。如果你需要收到此通知，则可以添加的处理程序 `PropertyChanged` 事件。

正在初始化时间属性

在代码中，您可以初始化 `Time` 属性的值类型 `TimeSpan` ：

```
TimePicker timePicker = new TimePicker
{
  Time = new TimeSpan(4, 15, 26) // Time set to "04:15:26"
};
```

当 `Time` 属性指定在 XAML 中，将值转换为 `TimeSpan` 和验证以确保的毫秒数大于或等于 0，并且的小时数是小于 24。时间组件应该用冒号分隔：

```
<TimePicker Time="4:15:26" />
```

如果 `BindingContext` 的属性 `TimePicker` 设置为包含属性的类型的 ViewModel 实例 `TimeSpan` 名为 `SelectedTime` （例如），您可以实例化 `TimePicker` 如下所示：

```
<TimePicker Time="{Binding SelectedTime}" />
```

在此示例中， `Time` 属性初始化为 `SelectedTime` ViewModel 中的属性。因为 `Time` 属性设置了绑定模式 `TwoWay` 、用户选择自动传播到 ViewModel 任何新的时间。

如果 `TimePicker` 不包含一个绑定上其 `Time` 属性，应用程序应附加到一个处理程序 `PropertyChanged` 事件当用户选择新的时间，则会收到通知。

有关设置字体属性的信息，请参阅字体。

TimePicker 和布局

可以使用不受约束的水平布局选项，如 `Center`， `Start`，或 `End` 与 `TimePicker`：

```
<TimePicker ···
            HorizontalOptions="Center"
            ··· />
```

但是，这不是建议在一起。这取决于设置 `Format` 所选择的属性，时间可能需要不同的显示宽度。例如，"T"格式字符串会导致 `TimePicker` 长格式显示时间视图和"4:15:26 AM"需要更大的显示宽度比"4:15 AM"的短时间格式("t")。根据平台，这种差异可能会导致 `TimePicker` 视图，以更改宽度在布局中，或显示被截断。

> **TIP**
> 最好是使用默认 `HorizontalOptions` 设置为 `Fill` 与 `TimePicker`，而不要使用的宽度 `Auto` 放置时 `TimePicker` 中 `Grid` 单元格。

应用程序中的 TimePicker

SetTimer 示例包括 `TimePicker`， `Entry`，和 `Switch` 在其页面上的视图。`TimePicker` 可以用来选择时间，以及显示警报对话框，用于提醒用户中的文本时的时间发生 `Entry` 提供 `Switch` 上切换。下面是 XAML 文件：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:SetTimer"
             x:Class="SetTimer.MainPage">
    <StackLayout>
        ...
        <Entry x:Name="_entry"
               Placeholder="Enter event to be reminded of" />
        <Label Text="Select the time below to be reminded at." />
        <TimePicker x:Name="_timePicker"
                    Time="11:00:00"
                    Format="T"
                    PropertyChanged="OnTimePickerPropertyChanged" />
        <StackLayout Orientation="Horizontal">
            <Label Text="Enable timer:" />
            <Switch x:Name="_switch"
                    HorizontalOptions="EndAndExpand"
                    Toggled="OnSwitchToggled" />
        </StackLayout>
    </StackLayout>
</ContentPage>
```

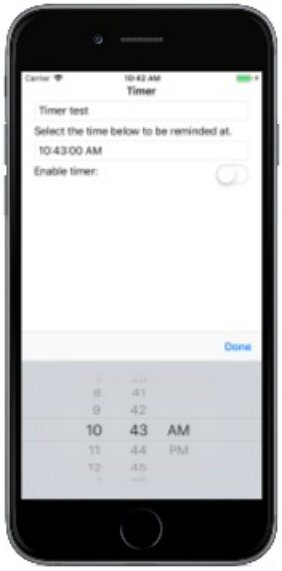
`Entry` 允许您输入所选的时间发生时将显示的提示文本。`TimePicker` 分配 `Format` "T"的长时间格式的属性。它具有事件处理程序附加到 `PropertyChanged` 事件，并且 `Switch` 有一个处理程序附加到其 `Toggled` 事件。这些事件处理程序是在代码隐藏文件和调用 `SetTriggerTime` 方法：

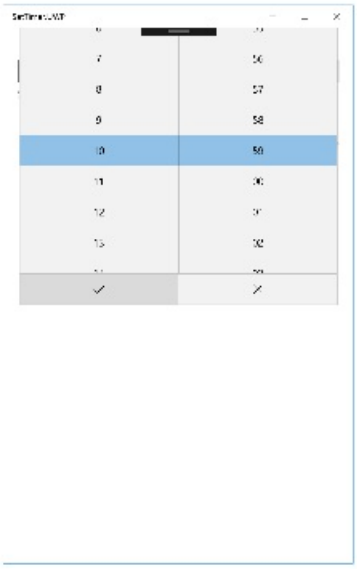
```
public partial class MainPage : ContentPage
{
    DateTime _triggerTime;

    public MainPage()
    {
        InitializeComponent();

        Device.StartTimer(TimeSpan.FromSeconds(1), OnTimerTick);
    }

    bool OnTimerTick()
    {
        if (_switch.IsToggled && DateTime.Now >= _triggerTime)
        {
            _switch.IsToggled = false;
            DisplayAlert("Timer Alert", "The '" + _entry.Text + "' timer has elapsed", "OK");
        }
        return true;
    }

    void OnTimePickerPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        if (args.PropertyName == "Time")
        {
            SetTriggerTime();
        }
    }

    void OnSwitchToggled(object sender, ToggledEventArgs args)
    {
        SetTriggerTime();
    }

    void SetTriggerTime()
    {
        if (_switch.IsToggled)
        {
            _triggerTime = DateTime.Today + _timePicker.Time;
            if (_triggerTime < DateTime.Now)
            {
                _triggerTime += TimeSpan.FromDays(1);
            }
        }
    }
}
```

`SetTriggerTime`方法计算基于计时器时间`DateTime.Today`属性值和`TimeSpan`返回值`TimePicker`。这是必需的因为`DateTime.Today`属性返回`DateTime`，该值指示当前的日期，但午夜的时间。如果计时器时间已过今天，则它被假定为明天。

每秒执行的计时器刻度`OnTimerTick`方法来检查是否`Switch`是上以及当前时间晚于或等于计时器时间。当计时器时间发生时，`DisplayAlert`方法提供一个警报对话框，提醒用户。

首次运行示例时，`TimePicker`到上午 11 初始化视图。点击`TimePicker`调用平台时间选取器。三个平台实现时间选取器很大不同，但是每种方法是熟悉该平台的用户：

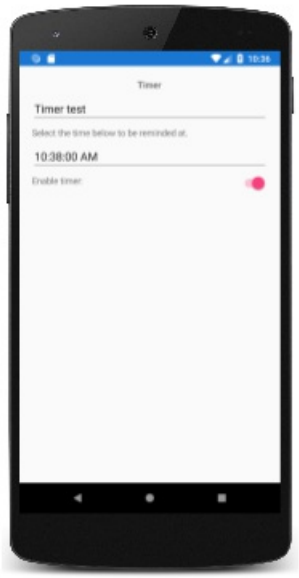




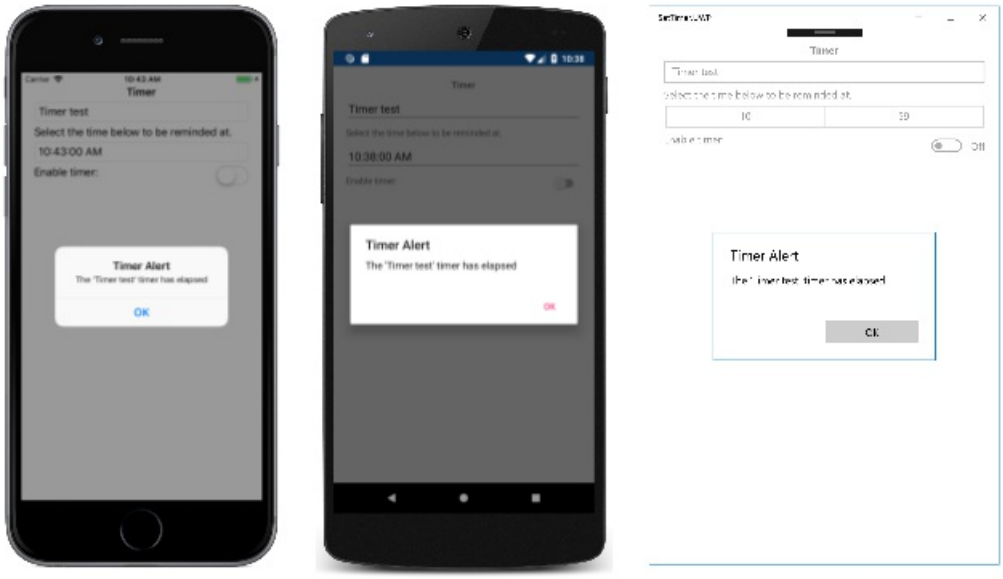
TIP

在 Android 上，`TimePicker` 对话框可以通过重写自定义 `CreateTimePickerDialog` 中自定义呈现器的方法。这样，例如，其他按钮来添加到了对话框。

选择一次之后，在选定的时间显示在 `TimePicker`：


前提 `Switch` 切换到打开的位置，应用程序将显示警报对话框提醒用户中的文本 `Entry` 所选的时间发生时：



只要显示警报对话框，则 `Switch` 切换到 off 位置。

相关链接

- SetTimer 示例
- TimePicker API

Xamarin.Forms 视觉状态管理器

2018/11/13 • • Edit Online

使用可视状态管理器对 XAML 元素按从代码中设置的视觉状态进行更改。

视觉状态管理器 (VSM) 是 Xamarin.Forms 3.0 中的新增功能。VSM 提供结构化的方式来从代码对用户界面进行可视更改。在大多数情况下，应用程序的用户界面中 XAML，定义和此 XAML 包含描述可视状态管理器如何影响用户界面的视觉对象的标记。

VSM 介绍的概念_可视状态_。如 Xamarin.Forms 视图 `Button` 可以有根据其基础状态的多个不同的视觉外观—是处于禁用状态，或按下，还是具有输入焦点。这些是按钮的状态。

视觉状态中收集_可视状态组_。可视状态组中的所有可视状态互相排斥。可视状态和可视状态组由简单的文本字符串标识。

Xamarin.Forms 视觉状态管理器定义一个可视状态组名"CommonStates"为具有三种可视状态：

- "Normal"
- "已禁用"
- "已设定焦点"

此可视状态组支持的所有类的派生 `VisualElement` ，这是类的基类 `View` 并 `Page` 。

此外可以定义自己的可视状态组和视觉状态，为这篇文章将演示。

> **NOTE**
>
> Xamarin.Forms 开发人员熟悉触发器可识别触发器还可以对基于视图的属性或事件触发中的更改的用户界面中的视觉对象进行更改。但是，使用触发器来处理这些更改的各种组合可能会变得非常令人困惑。从历史上看，基于 Windows XAML 的环境，若要缓解的可视状态的组合导致混乱中引入了视觉状态管理器。利用 VSM，可视状态组中的视觉状态始终是互相排斥的。在任何时候，每个组中的只有一个状态为当前状态。

常见的状态

视觉状态管理器，可在你可以更改视图的可视外观，如果视图是正常的或已禁用，或者具有输入的焦点的 XAML 文件中包含部分。这些参数称为_常用状态_。

例如，假设您有 `Entry` 在页中，视图和所需的可视外观 `Entry` 按以下方式更改：

- `Entry` 应有粉红色背景时 `Entry` 被禁用。
- `Entry` 通常应具有酸橙色背景。
- `Entry` 它具有输入焦点时应扩展到正常高度的两倍。

可以将 VSM 标记附加到一个单独的视图，或如果它适用于多个视图可以定义其样式中。接下来的两部分介绍了这些方法。

在视图上的 VSM 标记

若要附加到 VSM 标记 `Entry` 视图中，第一次单独的 `Entry` 到开始和结束标记：

```
<Entry FontSize="18">

</Entry>
```

因为将使用一种状态，提供显式的字号 `FontSize` 属性中的文本大小加倍 `Entry` 。

接下来，插入 `VisualStateManager.VisualStateGroups` 标记这些标记之间：

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>

    </VisualStateManager.VisualStateGroups>
</Entry>
```

`VisualStateGroups` 是定义的一个附加可绑定属性 `VisualStateManager` 类。(可绑定的附加属性的详细信息，请参阅文章附加属性。)这是如何 `VisualStateGroups` 属性将附加到 `Entry` 对象。

`VisualStateGroups` 属性属于类型 `VisualStateGroupList` ，这是一系列 `VisualStateGroup` 对象。内 `VisualStateManager.VisualStateGroups` 标记，插入一对 `VisualStateGroup` 标记为要包含的可视状态的每个组：

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">

        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

请注意，`VisualStateGroup` 标记具有 `x:Name` 属性，指示组的名称。`VisualStateGroup` 类定义 `Name` 可以改为使用的属性：

```
<VisualStateGroup Name="CommonStates">
```

你可以使用 `x:Name` 或 `Name` 但不是能同时在同一个元素。

`VisualStateGroup` 类定义一个名为属性 `States` ，这是一系列 `VisualState` 对象。 `States` 是_内容属性_的 `VisualStateGroups` 因此你可以将包含 `VisualState` 标记之间直接 `VisualStateGroup` 标记。(内容属性将在本文中讨论基本 XAML 语法。)

下一步是要包含在该组中的一对每个可视状态的标记。此外可以识别使用 `x:Name` 或 `Name` ：

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">

            </VisualState>

            <VisualState x:Name="Focused">

            </VisualState>

            <VisualState x:Name="Disabled">

            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

`VisualState` 定义一个名为属性 `Setters` ，这是一系列 `Setter` 对象。它们是相同 `Setter` 中使用对象 `Style` 对象。

`Setters` 是_不_的内容属性的 `VisualState`，因此必须包括为属性元素标记 `Setters` 属性：

```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Focused">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Disabled">
                <VisualState.Setters>

                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

你现在可以将插入一个或多个 `Setter` 对象之间的每个对 `Setters` 标记。这些是 `Setter` 对象以定义前面所述的可视状态：

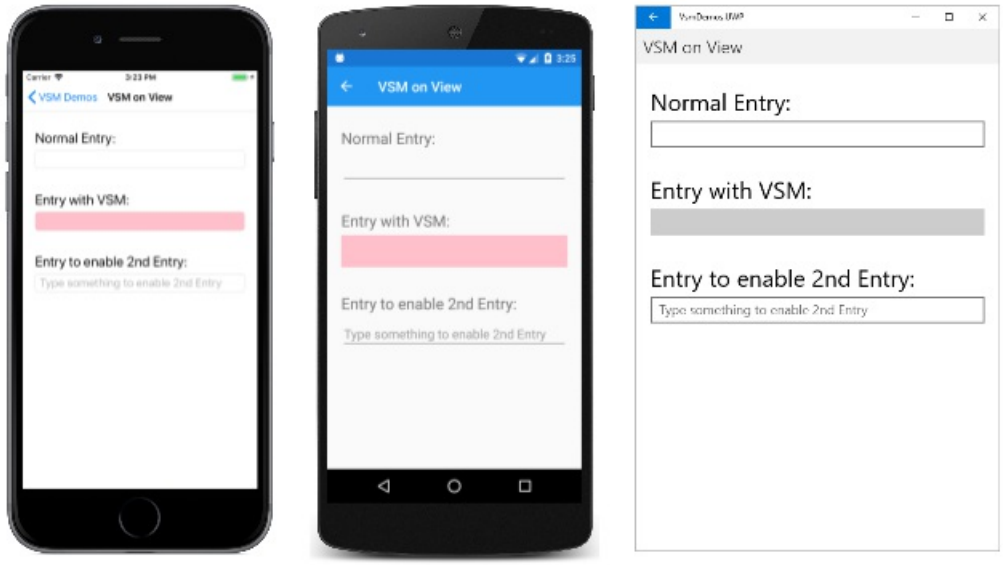
```
<Entry FontSize="18">
    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Lime" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Focused">
                <VisualState.Setters>
                    <Setter Property="FontSize" Value="36" />
                </VisualState.Setters>
            </VisualState>

            <VisualState x:Name="Disabled">
                <VisualState.Setters>
                    <Setter Property="BackgroundColor" Value="Pink" />
                </VisualState.Setters>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Entry>
```

每个 `Setter` 标记指示特定属性的值时该状态是最新。引用的任何属性 `Setter` 对象必须由可绑定的属性。

标记类似于以下是的基础视图上的 VSM 页面**VsmDemos** 示例程序。此页包含三个 `Entry` 视图中，但是仅第二个具有 VSM 标记附加到它：

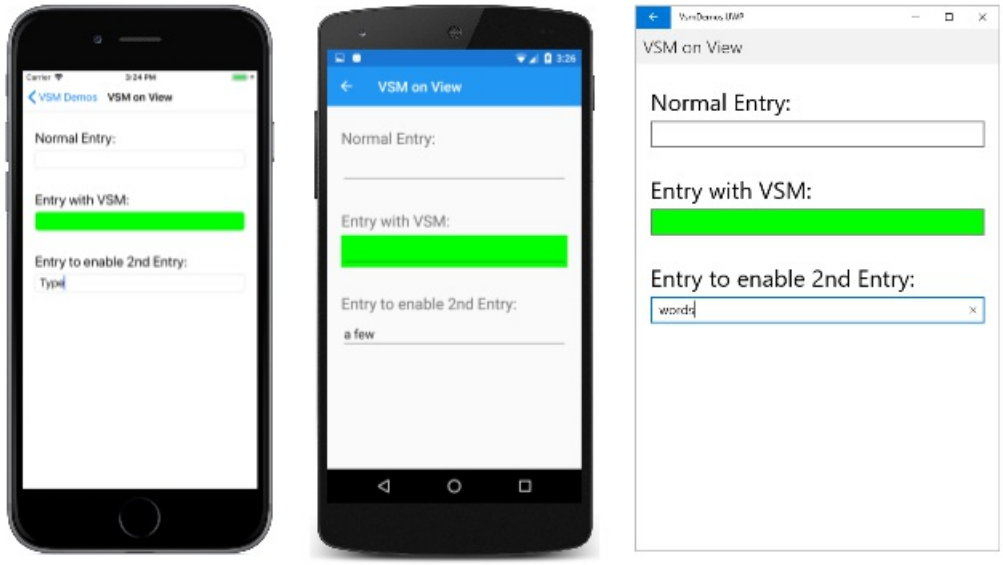
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:local="clr-namespace:VsmDemos"
             x:Class="VsmDemos.MainPage"
             Title="VSM Demos">

    <StackLayout>
        <StackLayout.Resources>
            <Style TargetType="Entry">
                <Setter Property="Margin" Value="20, 0" />
                <Setter Property="FontSize" Value="18" />
            </Style>

            <Style TargetType="Label">
                <Setter Property="Margin" Value="20, 30, 20, 0" />
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </StackLayout.Resources>

        <Label Text="Normal Entry:" />

        <Entry />

        <Label Text="Entry with VSM: " />

        <Entry>
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup x:Name="CommonStates">

                    <VisualState x:Name="Normal">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor" Value="Lime" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState x:Name="Focused">
                        <VisualState.Setters>
                            <Setter Property="FontSize" Value="36" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState x:Name="Disabled">
                        <VisualState.Setters>
                            <Setter Property="BackgroundColor" Value="Pink" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

            <Entry.Triggers>
                <DataTrigger TargetType="Entry"
                             Binding="{Binding Source={x:Reference entry3},
                                               Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Entry.Triggers>
        </Entry>

        <Label Text="Entry to enable 2nd Entry:" />

        <Entry x:Name="entry3"
               Text=""
               Placeholder="Type something to enable 2nd Entry" />
    </StackLayout>
</ContentPage>
```

请注意，第二个 `Entry` 还有 `DataTrigger` 作为的一部分其 `Trigger` 集合。这将导致 `Entry` 之前的内容键入到第三个要禁用 `Entry` 。以下是在启动 iOS、Android 和通用 Windows 平台 (UWP) 上运行时的页：

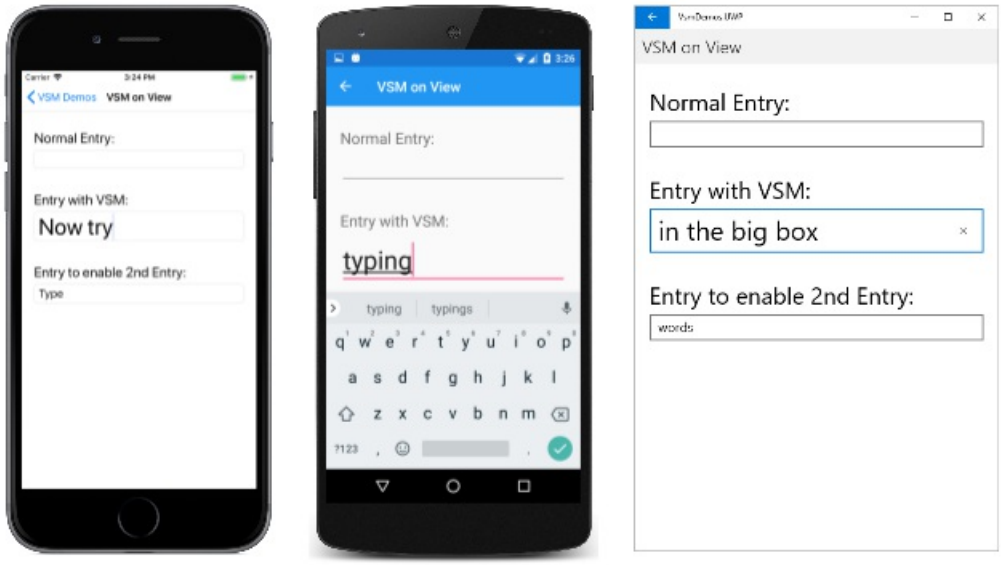


当前的可视状态为"Disabled"，因此第二个后台 `Entry` 为粉红色 iOS 和 Android 屏幕上。UWP 实现 `Entry` 不允许设置背景颜色时 `Entry` 被禁用。

到第三个输入一些文本时 `Entry` ，第二个 `Entry` 切换到"正常"状态，并在后台现酸橙色：



当您触摸第二个 `Entry` ，获取输入的焦点。它将切换到"已设定焦点"状态，并扩展到高度的两倍：



请注意，`Entry`获得输入的焦点时不会保留酸橙色背景。可视状态之间切换视觉状态管理器，以前的状态设置的属性是取消设置。请注意，视觉状态互相排斥。"正常"状态并不意味着仅`Entry`已启用。这意味着`Entry`已启用且不具有输入的焦点。

如果你想`Entry`若要为浅背景"焦点"状态中，添加另一个`Setter`该可视状态：

```
<VisualState x:Name="Focused">
    <VisualState.Setters>
        <Setter Property="FontSize" Value="36" />
        <Setter Property="BackgroundColor" Value="Lime" />
    </VisualState.Setters>
</VisualState>
```

为了使这些`Setter`对象才能正常工作，`VisualStateGroup`必须包含`VisualState`该组中的所有状态的对象。如果没有任何可视状态`Setter`对象，仍然为空标记包括：

```
<VisualState x:Name="Normal" />
```

在样式中的视觉状态管理器标记

它通常是共享相同的视觉状态管理器标记在两个或多个视图之间所需的。在这种情况下，你将想要将标记放入`Style`定义。

下面是现有隐式`Style`有关`Entry`中的元素**VSM 在视图页**：

```
<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
</Style>
```

添加`Setter`标记`VisualStateManager.VisualStateGroups`附加可绑定属性：

```
<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">

    </Setter>
</Style>
```

内容属性 `Setter` 是 `Value`，因此值 `Value` 可以直接在这些标记中指定属性。属性属于类型 `VisualStateGroupList`：

```
<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>

        </VisualStateGroupList>
    </Setter>
</Style>
```

这些标记中可以包含一个或多 `VisualStateGroup` 对象：

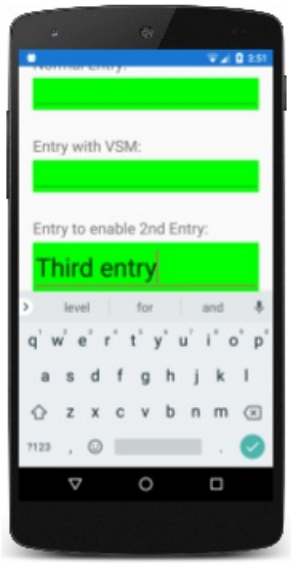
```
<Style TargetType="Entry">
    <Setter Property="Margin" Value="20, 0" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="VisualStateManager.VisualStateGroups">
        <VisualStateGroupList>
            <VisualStateGroup x:Name="CommonStates">

            </VisualStateGroup>
        </VisualStateGroupList>
    </Setter>
</Style>
```

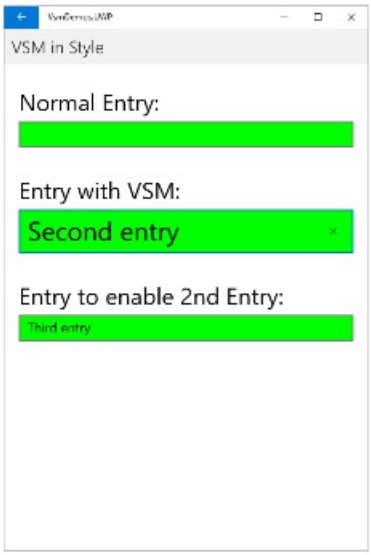
VSM 标记的其余部分是与之前相同。

下面是**VSM 在样式中的**页面，其中显示完整的 VSM 标记：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="VsmDemos.VsmInStylePage"
             Title="VSM in Style">
    <StackLayout>
        <StackLayout.Resources>
            <Style TargetType="Entry">
                <Setter Property="Margin" Value="20, 0" />
                <Setter Property="FontSize" Value="18" />
                <Setter Property="VisualStateManager.VisualStateGroups">
                    <VisualStateGroupList>
                        <VisualStateGroup x:Name="CommonStates">

                            <VisualState x:Name="Normal">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor" Value="Lime" />
                                </VisualState.Setters>
                            </VisualState>

                            <VisualState x:Name="Focused">
                                <VisualState.Setters>
                                    <Setter Property="FontSize" Value="36" />
                                    <Setter Property="BackgroundColor" Value="Lime" />
                                </VisualState.Setters>
                            </VisualState>

                            <VisualState x:Name="Disabled">
                                <VisualState.Setters>
                                    <Setter Property="BackgroundColor" Value="Pink" />
                                </VisualState.Setters>
                            </VisualState>
                        </VisualStateGroup>
                    </VisualStateGroupList>
                </Setter>
            </Style>

            <Style TargetType="Label">
                <Setter Property="Margin" Value="20, 30, 20, 0" />
                <Setter Property="FontSize" Value="Large" />
            </Style>
        </StackLayout.Resources>

        <Label Text="Normal Entry:" />

        <Entry />

        <Label Text="Entry with VSM: " />

        <Entry>
            <Entry.Triggers>
                <DataTrigger TargetType="Entry"
                             Binding="{Binding Source={x:Reference entry3},
                                               Path=Text.Length}"
                             Value="0">
                    <Setter Property="IsEnabled" Value="False" />
                </DataTrigger>
            </Entry.Triggers>
        </Entry>

        <Label Text="Entry to enable 2nd Entry:" />

        <Entry x:Name="entry3"
               Text=""
               Placeholder="Type something to enable 2nd Entry" />
    </StackLayout>
</ContentPage>
```

现在所有 `Entry` 此页上的视图响应其可视状态的相同方法。"焦点"状态现在包括第二个另请注意 `Setter`，它给出每个 `Entry` 酸橙色还后台时，它具有输入焦点：





定义你自己的可视状态

每个类都派生自 `VisualElement` 支持三个常用状态"正常"、"中心"和"已禁用"。在内部， `VisualElement` 类将检测时它正在成为已启用或禁用，或已设定焦点或失去焦点，并调用静态 `VisualStateManager.GoToState` 方法：

```
VisualStateManager.GoToState(this, "Focused");
```

这是您会发现中的唯一视觉状态管理器代码 `VisualElement` 类。因为 `GoToState` 为基于每个类都派生自每个对象调用 `VisualElement`，可以使用可视状态管理器以及任何 `VisualElement` 对象的这些更改进行响应。

有趣的是，可视状态组"CommonStates"的名称未显式引用中 `VisualElement` 。组名称不是可视状态管理器的 API 的一部分。中到目前为止所示的两个示例程序之一，你可以为任何其他内容，从"CommonStates"组的名称和程序仍将起作用。组名称是仅仅是该组中的状态的一般说明。隐式理解的任何组中的视觉状态是互相排斥：一个状态和只能有一个状态是当前在任何时间。

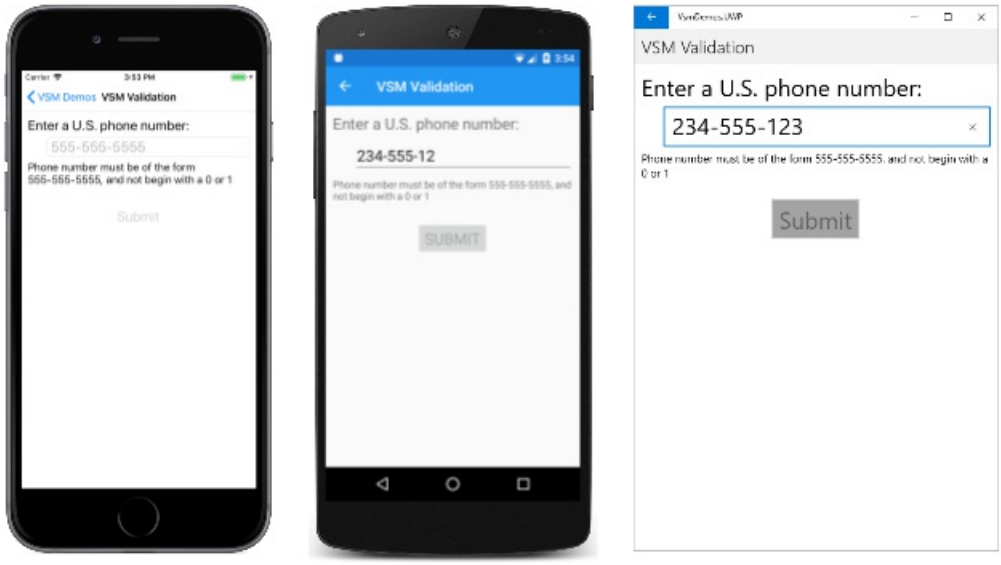
如果你想要实现您自己的视觉状态，你将需要调用 `VisualStateManager.GoToState` 从代码。通常，你将使此调用的页类代码隐藏文件中。

VSM 验证页面**VsmDemos** 示例演示如何使用输入验证与可视状态管理器。XAML 文件包含两个 `Label` 元素， `Entry` ，和 `Button` ：

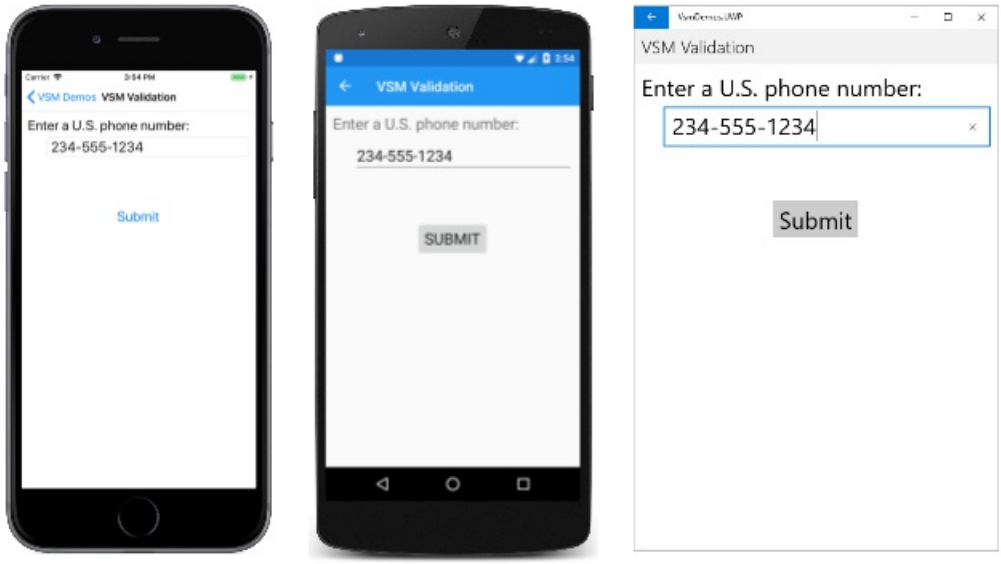
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="VsmDemos.VsmValidationPage"
             Title="VSM Validation">
    <StackLayout Padding="10, 10">

        <Label Text="Enter a U.S. phone number:"
               FontSize="Large" />

        <Entry Placeholder="555-555-5555"
               FontSize="Large"
               Margin="30, 0, 0, 0"
               TextChanged="OnTextChanged" />

        <Label x:Name="helpLabel"
               Text="Phone number must be of the form 555-555-5555, and not begin with a 0 or 1">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ValidityStates">

                    <VisualState Name="Valid">
                        <VisualState.Setters>
                            <Setter Property="TextColor" Value="Transparent" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="Invalid" />

                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Label>

        <Button x:Name="submitButton"
                Text="Submit"
                FontSize="Large"
                Margin="0, 20"
                VerticalOptions="Center"
                HorizontalOptions="Center">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="ValidityStates">

                    <VisualState Name="Valid" />

                    <VisualState Name="Invalid">
                        <VisualState.Setters>
                            <Setter Property="IsEnabled" Value="False" />
                        </VisualState.Setters>
                    </VisualState>

                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>
        </Button>
    </StackLayout>
</ContentPage>
```

VSM 标记附加到第二个 `Label`（名为 `helpLabel`）和 `Button`（名为 `submitButton`）。有两个互斥的状态，名为"Valid"和"无效"。请注意，两个"ValidationState"组的每个包含 `VisualState` 标记为"Valid"和"无效"，虽然其中一种是在每种情况下为空。

如果 `Entry` 不包含有效的电话号码，则当前状态为"无效"，因此第二个 `Label` 可见和 `Button` 已禁用：



输入有效的电话号码，然后当前状态将变为"Valid"。第二个 `Entry` 消失和 `Button` 现已启用：



代码隐藏文件负责处理 `TextChanged` 从事件 `Entry` 。该处理程序使用的正则表达式来确定输入的字符串是否有效。名为的代码隐藏文件中的方法 `GoToState` 调用静态 `VisualStateManager.GoToState` 两个方法 `helpLabel` 和 `submitButton` ：

```
public partial class VsmValidationPage : ContentPage
{
    public VsmValidationPage ()
    {
        InitializeComponent ();

        GoToState(false);
    }

    void OnTextChanged(object sender, TextChangedEventArgs args)
    {
        bool isValid = Regex.IsMatch(args.NewTextValue, @"^[2-9]\d{2}-\d{3}-\d{4}$");
        GoToState(isValid);
    }

    void GoToState(bool isValid)
    {
        string visualState = isValid ? "Valid" : "Invalid";
        VisualStateManager.GoToState(helpLabel, visualState);
        VisualStateManager.GoToState(submitButton, visualState);
    }
}
```

另请注意 `GoToState` 从初始化状态的构造函数中调用方法。始终应为当前状态。但不是在代码中有任何引用的可视状态组中，名称虽然它在 XAML 中的引用作为"ValidationStates"为了清晰起见。

请注意，代码隐藏文件必须采取的每个对象的帐户受影响的页面上，通过这些视觉状态，并调用 `VisualStateManager.GoToState` 为每个对象。在此示例中，它是只有两个对象（ `Label` 和 `Button` ），但它可能是几个更多。

您可能想知道：如果代码隐藏文件必须引用受这些视觉状态的页面上的每个对象，为什么不能的代码隐藏文件只需访问的对象直接？ 当然可以。但是，使用 VSM 的优点是可以控制如何将视觉元素，对完全在 XAML，将所有 UI 设计保留在一个位置中的不同状态做出反应。这直接从代码隐藏中访问的可视元素，因此可避免设置可视外观。

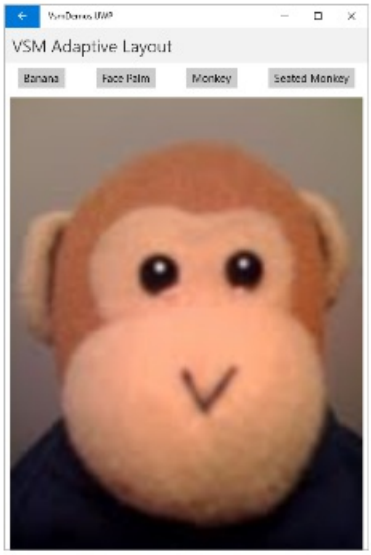
它很可能，请考虑派生的类从 `Entry` 和可能定义一个属性，可以设置为外部验证函数。派生的类 `Entry` 然后可以调用 `VisualStateManager.GoToState` 方法。此方案将能正常运行，但仅当 `Entry` 已受不同的可视状态的唯一对象。在此示例中， `Label` 和一个 `Button` 也会受到影响。没有任何方法为 VSM 标记附加到 `Entry` 来控制在页面中，没有方法上的其他对象的 VSM 标记附加到这些其他对象，若要从另一个对象引用中的可视状态的更改。

使用可视状态管理器进行自适应的布局

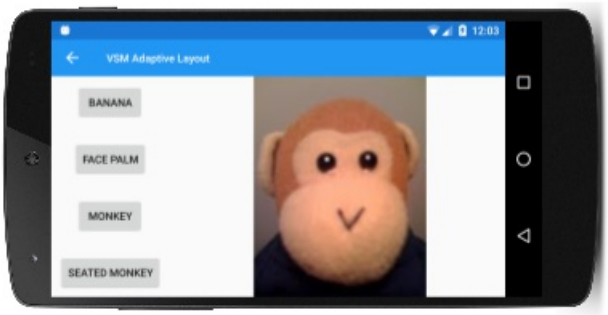
可以调整大小在手机上运行的应用程序通常可以查看在纵向或横向纵横比，并且在桌面上运行的 Xamarin.Forms 程序 Xamarin.Forms 采用许多不同的大小和纵横比。设计良好的应用程序可能会显示不同的方式针对这些不同的页或窗口外观造型其内容。

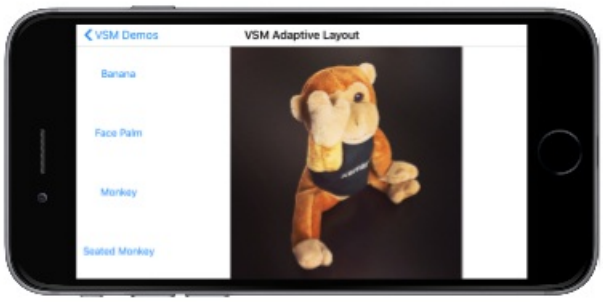
此方法有时称为_自适应布局_。由于自适应布局仅涉及程序的视觉对象，它是理想的可视状态管理器应用程序。

一个简单的示例是显示小的会影响应用程序的内容的按钮集合的应用程序。在纵向模式下，可能会在页面顶部水平行中显示这些按钮：



在横向模式下，可能会移动到一个一侧，和在列中显示的按钮数组：





从上到下，通用 Windows 平台、Android 和 iOS 上运行该程序。

VSM 自适应布局页面VsmDemos示例定义名为"纵向"和"横向"的两个可视状态与名为"OrientationStates"的组。（更复杂的方法可能会基于多个不同的页或窗口宽度。）

VSM 标记出现在 XAML 文件中的四个位置。`StackLayout` 名为 `mainStack` 包含的菜单和内容，这是 `Image` 元素。这 `StackLayout` 应具有垂直方向为纵向模式和横向模式下表中的水平方向：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
```

```
             x:Class="VsmDemos.VsmAdaptiveLayoutPage"
             Title="VSM Adaptive Layout">

    <StackLayout x:Name="mainStack">
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="OrientationStates">

                <VisualState Name="Portrait">
                    <VisualState.Setters>
                        <Setter Property="Orientation" Value="Vertical" />
                    </VisualState.Setters>
                </VisualState>

                <VisualState Name="Landscape">
                    <VisualState.Setters>
                        <Setter Property="Orientation" Value="Horizontal" />
                    </VisualState.Setters>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>

        <ScrollView x:Name="menuScroll">
            <VisualStateManager.VisualStateGroups>
                <VisualStateGroup Name="OrientationStates">

                    <VisualState Name="Portrait">
                        <VisualState.Setters>
                            <Setter Property="Orientation" Value="Horizontal" />
                        </VisualState.Setters>
                    </VisualState>

                    <VisualState Name="Landscape">
                        <VisualState.Setters>
                            <Setter Property="Orientation" Value="Vertical" />
                        </VisualState.Setters>
                    </VisualState>
                </VisualStateGroup>
            </VisualStateManager.VisualStateGroups>

            <StackLayout x:Name="menuStack">
                <VisualStateManager.VisualStateGroups>
                    <VisualStateGroup Name="OrientationStates">

                        <VisualState Name="Portrait">
                            <VisualState.Setters>
                                <Setter Property="Orientation" Value="Horizontal" />
                            </VisualState.Setters>
                        </VisualState>

                        <VisualState Name="Landscape">
                            <VisualState.Setters>
                                <Setter Property="Orientation" Value="Vertical" />
                            </VisualState.Setters>
                        </VisualState>
                    </VisualStateGroup>
                </VisualStateManager.VisualStateGroups>

                <StackLayout.Resources>
                    <Style TargetType="Button">
                        <Setter Property="VisualStateManager.VisualStateGroups">
                            <VisualStateGroupList>
                                <VisualStateGroup Name="OrientationStates">

                                    <VisualState Name="Portrait">
                                        <VisualState.Setters>
                                            <Setter Property="HorizontalOptions" Value="CenterAndExpand" />
                                            <Setter Property="Margin" Value="10, 5" />
                                        </VisualState.Setters>
                                    </VisualState>
```

```
                                <VisualState Name="Landscape">
                                    <VisualState.Setters>
                                        <Setter Property="VerticalOptions" Value="CenterAndExpand" />
                                        <Setter Property="HorizontalOptions" Value="Center" />
                                        <Setter Property="Margin" Value="10" />
                                    </VisualState.Setters>
                                </VisualState>
                            </VisualStateGroup>
                        </VisualStateGroupList>
                    </Setter>
                </Style>
            </StackLayout.Resources>

            <Button Text="Banana"
                    Command="{Binding SelectedCommand}"
                    CommandParameter="Banana.jpg" />

            <Button Text="Face Palm"
                    Command="{Binding SelectedCommand}"
                    CommandParameter="FacePalm.jpg" />

            <Button Text="Monkey"
                    Command="{Binding SelectedCommand}"
                    CommandParameter="monkey.png" />

            <Button Text="Seated Monkey"
                    Command="{Binding SelectedCommand}"
                    CommandParameter="SeatedMonkey.jpg" />
        </StackLayout>
    </ScrollView>

    <Image x:Name="image"
           VerticalOptions="FillAndExpand"
           HorizontalOptions="FillAndExpand" />
</StackLayout>
</ContentPage>
```

内部 `ScrollView` 名为 `menuScroll` 并 `StackLayout` 名为 `menuStack` 实现按钮的菜单。这些布局的方向是相反的 `mainStack` 。在纵向模式下水平和垂直在横向模式下，应为菜单。

中的按钮本身的隐式样式是 VSM 标记的第四个部分。此标记将设置 `VerticalOptions` ， `HorizontalOptions` ，和 `Margin` 特定于 portait 和横向方向的属性。

代码隐藏文件集 `BindingContext` 的属性 `menuStack` 实现 `Button` 命令，并还会将附加到一个处理程序 `SizeChanged` 页面的事件：

```
public partial class VsmAdaptiveLayoutPage : ContentPage
{
    public VsmAdaptiveLayoutPage ()
    {
        InitializeComponent ();

        SizeChanged += (sender, args) =>
        {
            string visualState = Width > Height ? "Landscape" : "Portrait";
            VisualStateManager.GoToState(mainStack, visualState);
            VisualStateManager.GoToState(menuScroll, visualState);
            VisualStateManager.GoToState(menuStack, visualState);

            foreach (View child in menuStack.Children)
            {
                VisualStateManager.GoToState(child, visualState);
            }
        };

        SelectedCommand = new Command<string>((filename) =>
        {
            image.Source = ImageSource.FromResource("VsmDemos.Images." + filename);
        });

        menuStack.BindingContext = this;
    }

    public ICommand SelectedCommand { private set; get; }
}
```

`SizeChanged` 处理程序调用 `VisualStateManager.GoToState` 的两个 `StackLayout` 并 `ScrollView` 元素，然后循环访问的子级 `menuStack` 调用 `VisualStateManager.GoToState` 为 `Button` 元素。

这可能看起来像代码隐藏文件可以处理更直接地通过在 XAML 文件中，设置元素的属性的方向更改，但视觉状态管理器肯定是一种更加结构化的方法。所有视觉对象将保留在 XAML 文件，其中它们变得更轻松地检查，维护和修改。

使用 Xamarin.University 视觉状态管理器

Xamarin.Forms 3.0 视觉状态管理器，也可由Xamarin 学院课程

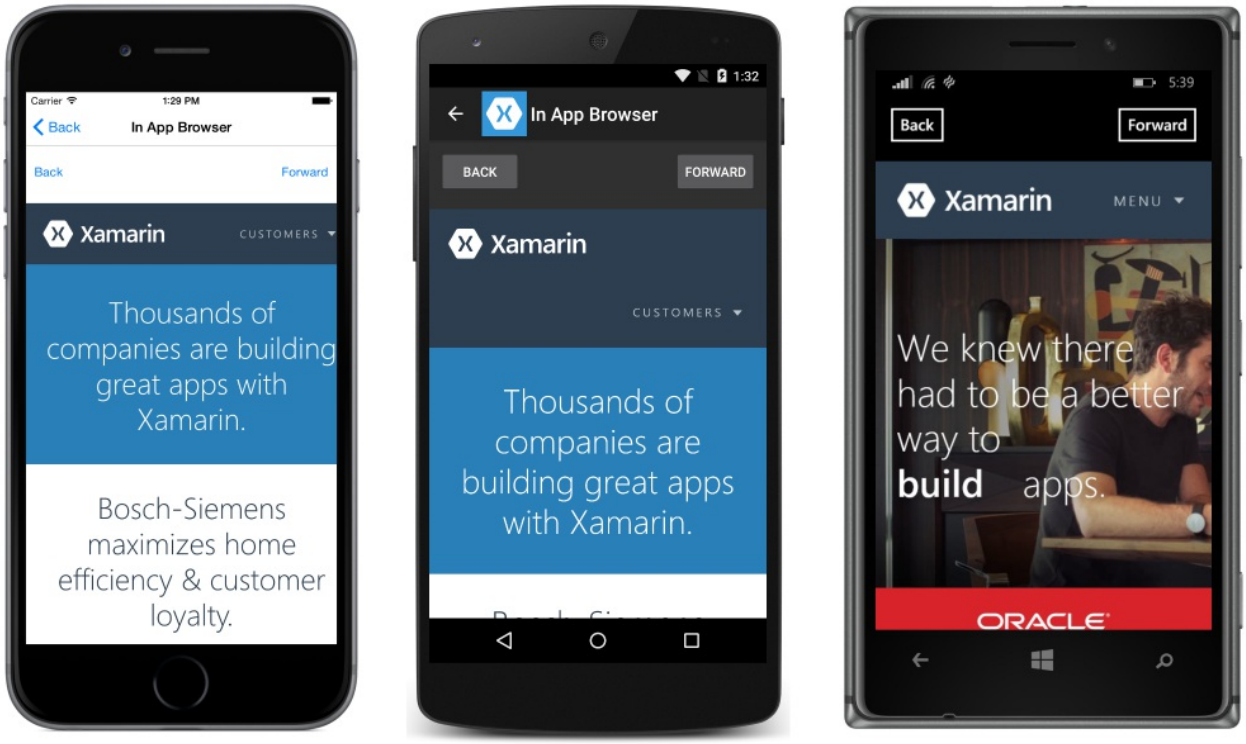
相关链接

- VsmDemos

Xamarin.Forms WebView

2018/10/26 • • Edit Online

`WebView` 是应用程序中显示 web 和 HTML 内容的视图。与不同 `OpenUri`，使用户转到 web 浏览器在设备上，`WebView` 显示您的应用程序中的 HTML 内容。



内容

`WebView` 支持以下类型的内容：

- HTML 和 CSS 的网站–WebView 具有完全支持使用 HTML 和 CSS，包括支持 JavaScript 编写的网站。
- 文档–WebView WebView 每个平台上使用本机组件实现的因为它能够显示每个平台上可查看的文档。这意味着在 iOS 和 Android 上工作的 PDF 文件。
- HTML 字符串–web 视图可以显示内存中的 HTML 字符串。
- 本地文件–WebView 可以提供更高版本的内容类型的任何嵌入在应用程序。

> **NOTE**
>
> `WebView` 在 Windows 上不支持 Silverlight、Flash 或任何 ActiveX 控件，即使它们在该平台上支持的 Internet 资源管理器。

网站

若要显示来自 internet 的网站，请设置 `WebView` 的 `Source` 属性设置为字符串 URL：

```
var browser = new WebView {
  Source = "http://xamarin.com"
};
```

> **NOTE**
>
> Url 必须完全与指定的协议格式（即它必须具有"http://"或"https://"开头加）。

iOS 和 ATS

自版本 9，iOS 将只允许你的应用程序与实现最佳安全性，默认情况下的服务器进行通信。值必须设置 `Info.plist` 能够与不安全的服务器的通信。

> **NOTE**
>
> 如果你的应用程序需要连接到不安全的网站，始终应为异常使用输入的域 `NSExceptionDomains` 而不是关闭 ATS 完全使用 `NSAllowsArbitraryLoads`。 `NSAllowsArbitraryLoads` 仅应在极端的紧急情况下使用。

下面演示了如何启用特定域（在此事例 xamarin.com）以绕过 ATS 要求：

```
<key>NSAppTransportSecurity</key>
    <dict>
        <key>NSExceptionDomains</key>
        <dict>
            <key>xamarin.com</key>
            <dict>
                <key>NSIncludesSubdomains</key>
                <true/>
                <key>NSTemporaryExceptionAllowsInsecureHTTPLoads</key>
                <true/>
                <key>NSTemporaryExceptionMinimumTLSVersion</key>
                <string>TLSv1.1</string>
            </dict>
        </dict>
    </dict>
```

它是最佳做法，只允许某些域绕过 ATS，从而可以受益于对不受信任域的附加安全性时使用受信任的站点。下面演示了如何不安全的应用程序禁用 ATS 方法：

```
<key>NSAppTransportSecurity</key>
    <dict>
        <key>NSAllowsArbitraryLoads </key>
        <true/>
    </dict>
```

请参阅应用程序传输安全有关 iOS 9 中此新功能的详细信息。

HTML 字符串

如果你想要显示的代码中动态定义的 HTML 字符串，你将需要创建的实例 `HtmlWebViewSource`：

```
var browser = new WebView();
var htmlSource = new HtmlWebViewSource();
htmlSource.Html = @"<html><body>
  <h1>Xamarin.Forms</h1>
  <p>Welcome to WebView.</p>
  </body></html>";
browser.Source = htmlSource;
```

在上面的代码，`@`用于将 HTML 标记作为字符串文本，这意味着，将忽略所有常见的转义字符。

本地 HTML 内容

Web 视图可以显示内容从 HTML、CSS 和 Javascript 嵌入在应用中。例如：

```
<html>
  <head>
    <title>Xamarin Forms</title>
  </head>
  <body>
    <h1>Xamrin.Forms</h1>
    <p>This is an iOS web page.</p>
    <img src="XamarinLogo.png" />
  </body>
</html>
```

CSS:

```
html,body {
  margin:0;
  padding:10;
}
body,p,h1 {
  font-family: Chalkduster;
}
```

请注意，在上面的 CSS 中指定的字体将需要为每个平台自定义为不是每个平台都有相同的字体。

显示本地内容使用`WebView`，将需要打开任何其他 HTML 文件，然后将内容加载到字符串形式`Html`属性的`HtmlWebViewSource`。打开文件的详细信息，请参阅使用文件。

以下屏幕截图显示在每个平台上显示本地内容的结果：

已加载的第一页，尽管 `WebView` 一无所知的 HTML 原来所在的位置。在处理引用本地资源的页面时，这是个问题。时，可能会执行此操作的示例包括为每个其他页面使本地的页面链接使用的单独的 JavaScript 文件，或一个页面链接到 CSS 样式表时。

若要解决此问题，需要告诉 `WebView` 查找文件系统上的文件位置。会通过设置 `BaseUrl` 上的属性 `HtmlWebViewSource` 由 `WebView` 。

由于每个操作系统上的文件系统不同，您需要确定每个平台上的该 URL。Xamarin.Forms 公开 `DependencyService` 用于解析在每个平台上运行时依赖项。

若要使用 `DependencyService` ，首先定义一个可以在每个平台实现的接口：

```
public interface IBaseUrl { string Get(); }
```

请注意，每个平台上实现接口，直到应用程序将不会运行。在常见的项目中，请确保你请别忘记设置 `BaseUrl` 使用 `DependencyService` ：

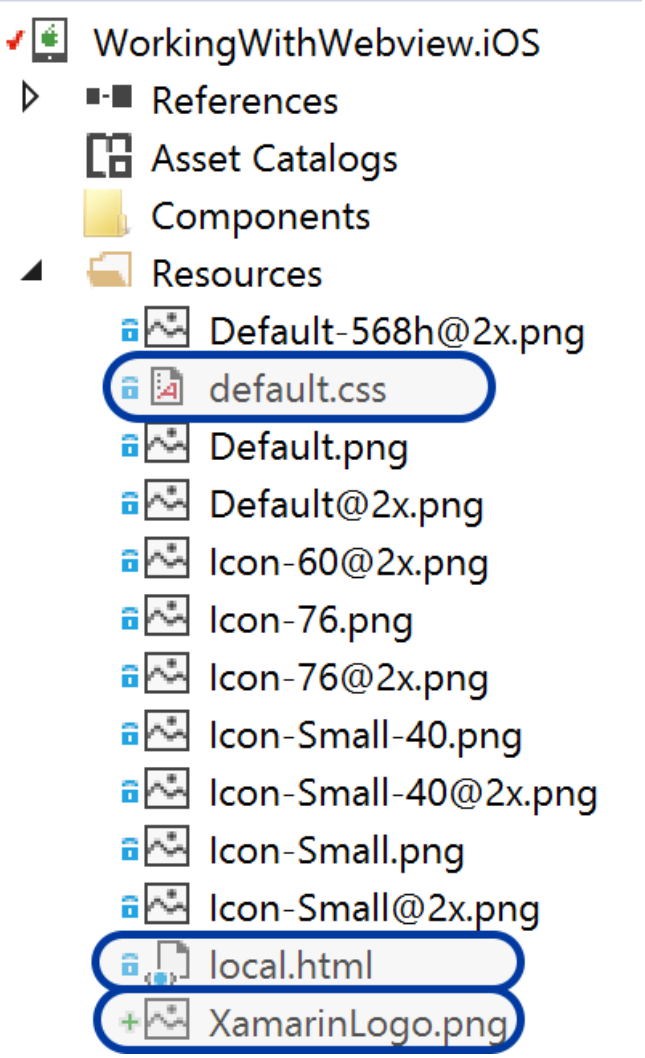
```
var source = new HtmlWebViewSource();
source.BaseUrl = DependencyService.Get<IBaseUrl>().Get();
```

然后必须提供每个平台接口的实现。

iOS

在 iOS 上，web 内容应位于项目的根目录中或资源生成操作目录*BundleResource*如下所示：

- Visual Studio
- Visual Studio for Mac



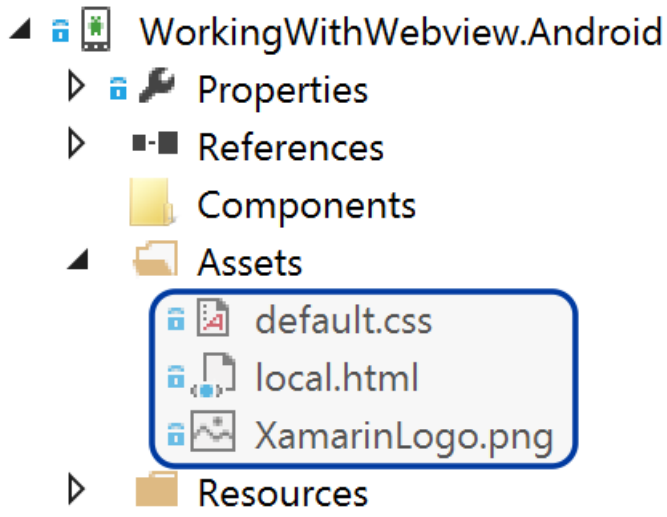
`BaseUrl` 应设置为主要捆绑包的路径：

```
[assembly: Dependency (typeof (BaseUrl_iOS))]
namespace WorkingWithWebview.iOS{
  public class BaseUrl_iOS : IBaseUrl {
    public string Get() {
      return NSBundle.MainBundle.BundlePath;
    }
  }
}
```

Android

在 Android 上，HTML、CSS 和图像的文件夹中放置资产与生成操作*AndroidAsset*如下所示：

- Visual Studio
- Visual Studio for Mac



在 Android 上，`BaseUrl` 应设置为 `"file:///android_asset/"`：

```
[assembly: Dependency (typeof(BaseUrl_Android))]
namespace WorkingWithWebview.Android {
  public class BaseUrl_Android : IBaseUrl {
    public string Get() {
      return "file:///android_asset/";
    }
  }
}
```

在 Android 上，文件中资产文件夹也可以访问通过当前的 Android 上下文，它公开由 `MainActivity.Instance` 属性：

```
var assetManager = MainActivity.Instance.Assets;
using (var streamReader = new StreamReader (assetManager.Open ("local.html"))) {
  var html = streamReader.ReadToEnd ();
}
```

通用 Windows 平台

通用 Windows 平台 (UWP) 项目放在 HTML、CSS 和图像中的项目根目录生成操作设置为*内容*。

`BaseUrl` 应设置为 `"ms-appx-web:///"`：

```
[assembly: Dependency(typeof(BaseUrl))]
namespace WorkingWithWebview.UWP
{
    public class BaseUrl : IBaseUrl
    {
        public string Get()
        {
            return "ms-appx-web:///";
        }
    }
}
```

导航

WebView 支持通过多种方法和属性，可通过其导航：

- **GoForward()** –如果 `CanGoForward` 为 true，调用 `GoForward` 向前导航到下一步访问过的页。
- **GoBack()** –如果 `CanGoBack` 为 true，调用 `GoBack` 将导航到访问过的最后一页。
- **CanGoBack** – `true` 是否存在页导航返回到 `false` 如果浏览器在启动的 URL。

- **CanGoForward** – `true` 如果用户已向后导航，并且可以向前移动到已访问过的页。

在页中，`WebView` 不支持多点触控笔势。务必要确保该内容是移动优化，并显示而无需缩放。

通常，应用程序以显示中的链接 `WebView`，而不是设备的浏览器。在这些情况下，最好允许正常导航栏中，但当起始链接上，则后退中的用户点击率、应用程序应返回到正常的应用视图。

使用内置的导航方法和属性来启用此方案。

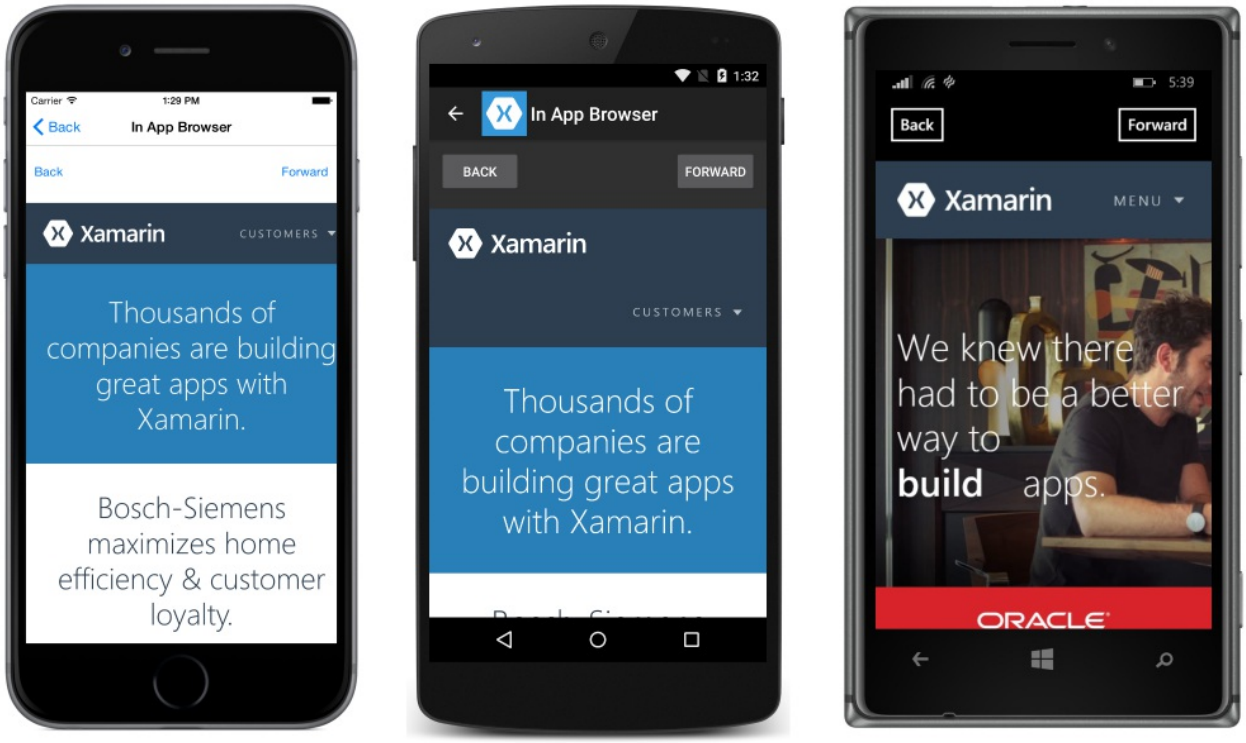
首先创建在浏览器视图的页：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="WebViewDemo.InAppDemo"
Title="In App Browser">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout Orientation="Horizontal" Padding="10,10">
                <Button Text="Back" HorizontalOptions="StartAndExpand" Clicked="backClicked" />
                <Button Text="Forward" HorizontalOptions="End" Clicked="forwardClicked" />
            </StackLayout>
            <WebView x:Name="Browser" WidthRequest="1000" HeightRequest="1000" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

在我们的代码隐藏：

```
public partial class InAppDemo : ContentPage
{
  //sets the URL for the browser in the page at creation
    public InAppDemo (string URL)
    {
        InitializeComponent ();
        Browser.Source = URL;
    }


    private void backClicked(object sender, EventArgs e)
    {
    // Check to see if there is anywhere to go back to
        if (Browser.CanGoBack) {
            Browser.GoBack ();
        } else { // If not, leave the view
            Navigation.PopAsync ();
        }
    }

    private void forwardClicked(object sender, EventArgs e)
    {
        if (Browser.CanGoForward) {
            Browser.GoForward ();
        }
    }
}
```

就这么简单！



事件

WebView 引发两个事件，以帮助您对状态的变化做出响应：

- **导航** – WebView 开始加载新页面时引发的事件。
- **导航**–页面加载和导航已停止时引发事件。

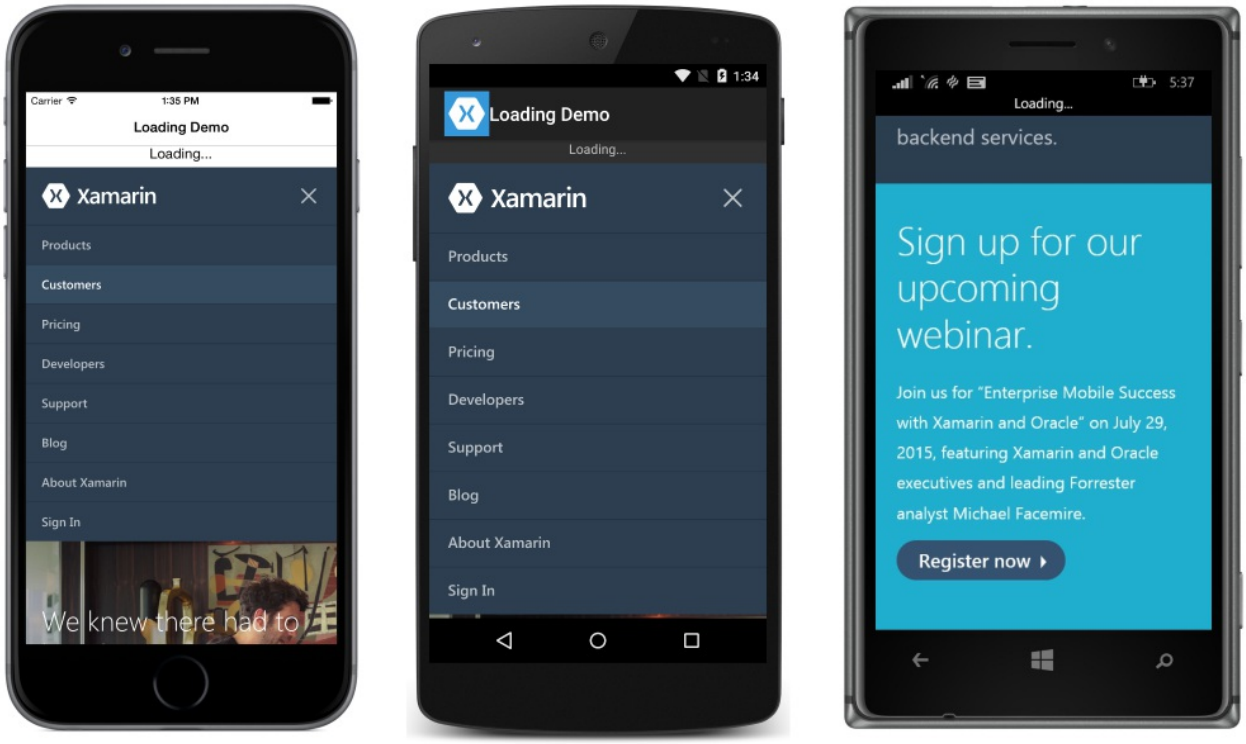
如果你希望使用需要很长时间加载的网页，请考虑使用这些事件来实现状态指示器。例如 XAML 如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
x:Class="WebViewDemo.LoadingDemo" Title="Loading Demo">
  <ContentPage.Content>
    <StackLayout>
      <Label x:Name="LoadingLabel"
        Text="Loading..."
        HorizontalOptions="Center"
        IsVisible="false" />
      <WebView x:Name="Browser"
      HeightRequest="1000"
      WidthRequest="1000"
      Navigating="webOnNavigating"
      Navigated="webOnEndNavigating" />
    </StackLayout>
  </ContentPage.Content>
</ContentPage>
```

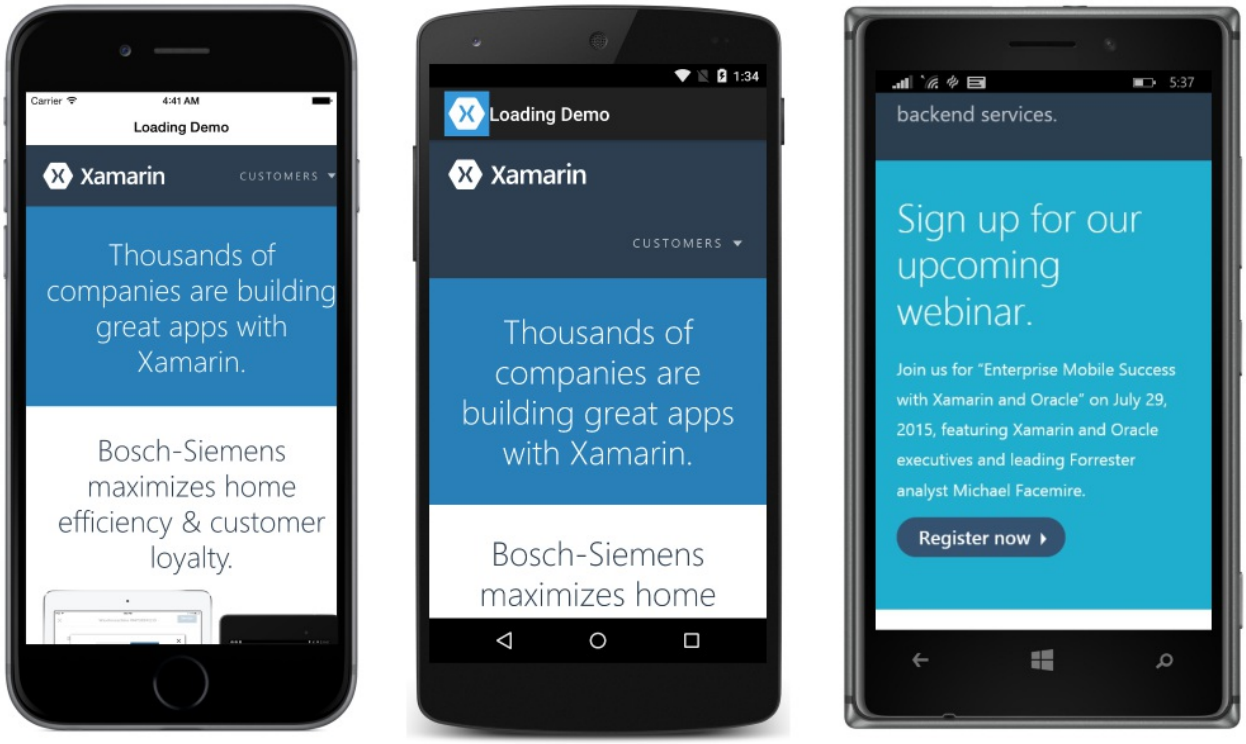
两个事件处理程序：

```
void webOnNavigating (object sender, WebNavigatingEventArgs e)
{
    LoadingLabel.IsVisible = true;
}

void webOnEndNavigating (object sender, WebNavigatedEventArgs e)
{
    LoadingLabel.IsVisible = false;
}
```

这会导致以下输出（加载）：



已完成的加载：



性能

常用 web 浏览器现在采用技术，如硬件加速呈现和 JavaScript 编译。在 iOS 上，默认情况下，Xamarin.Forms `WebView` 由实现 `UIWebView` 类，其中许多技术是在此实现中不可用。但是，应用程序可以选择项以使用 iOS `WkWebView` 类，以实现 Xamarin.Forms `WebView`，它支持更快地浏览。这可以通过添加以下代码来实现**AssemblyInfo.cs** iOS 平台项目中的应用程序的文件：

```
// Opt-in to using WkWebView instead of UIWebView.
[assembly: ExportRenderer(typeof(WebView), typeof(Xamarin.Forms.Platform.iOS.WkWebViewRenderer))]
```

`WebView` 默认情况下在 Android 上是大约与内置浏览器一样快。

UWP WebView使用 Microsoft Edge 呈现引擎。台式机和平板电脑设备应看到与使用 Edge 浏览器本身相同的性能。

权限

为了使 `WebView` 工作，您必须确保为每个平台设置权限。请注意，在某些平台上 `WebView` 处于调试模式，但不是在生成要发布时将工作。这是因为 for Mac 在调试模式下时，默认情况下，Visual studio 的某些权限，如用于在 Android 上，internet 访问权限设置。

- **UWP** –显示网络内容时需要 Internet（客户端和服务器）功能。
- **Android** –需要 `INTERNET` 仅显示网络中的内容时。本地内容需要的任何特殊权限。
- **iOS** –不需任何特殊权限。

布局

与大多数其他 Xamarin.Forms 视图，不同 `WebView` 要求 `HeightRequest` 和 `WidthRequest` 时包含在 StackLayout 或 RelativeLayout 所指定的。如果您不能指定这些属性，`WebView` 将不会呈现。

下面的示例演示工作，所导致的布局呈现 `WebView` s:

与 WidthRequest HeightRequest StackLayout:

```
<StackLayout>
    <Label Text="test" />
    <WebView Source="http://www.xamarin.com/"
        HeightRequest="1000"
        WidthRequest="1000" />
</StackLayout>
```

与 WidthRequest HeightRequest RelativeLayout:

```
<RelativeLayout>
    <Label Text="test"
        RelativeLayout.XConstraint= "{ConstraintExpression
                                     Type=Constant, Constant=10}"
        RelativeLayout.YConstraint= "{ConstraintExpression
                                     Type=Constant, Constant=20}" />
    <WebView Source="http://www.xamarin.com/"
        RelativeLayout.XConstraint="{ConstraintExpression Type=Constant,
                                     Constant=10}"
        RelativeLayout.YConstraint="{ConstraintExpression Type=Constant,
                                     Constant=50}"
        WidthRequest="1000" HeightRequest="1000" />
</RelativeLayout>
```

AbsoluteLayout*而无需*WidthRequest & HeightRequest:

```
<AbsoluteLayout>
    <Label Text="test" AbsoluteLayout.LayoutBounds="0,0,100,100" />
    <WebView Source="http://www.xamarin.com/"
      AbsoluteLayout.LayoutBounds="0,150,500,500" />
</AbsoluteLayout>
```

网格*而无需*WidthRequest & HeightRequest。网格是一种不需要指定请求的高度和宽度的几个版式。:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="100" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Label Text="test" Grid.Row="0" />
    <WebView Source="http://www.xamarin.com/" Grid.Row="1" />
</Grid>
```

调用 JavaScript

`WebView` 包括调用一个 JavaScript 函数中的 C# 中，并将任何结果返回到调用的 C# 代码的功能。这通过实现 `WebView.EvaluateJavaScriptAsync` 方法，在以下示例中所示WebView示例:

```
var numberEntry = new Entry { Text = "5" };
var resultLabel = new Label();
var webView = new WebView();
...

int number = int.Parse(numberEntry.Text);
string result = await webView.EvaluateJavaScriptAsync($"factorial({number})");
resultLabel.Text = $"Factorial of {number} is {result}.";
```

`WebView.EvaluateJavaScriptAsync` 方法将评估指定为参数，并返回任何结果作为 JavaScript `string` 。在此示例中，`factorial` 调用 JavaScript 函数时，它返回的阶乘 `number` 作为结果。此 JavaScript 函数定义中的本地 HTML 文件 `WebView` 加载，并在下面的示例所示:

```
<html>
<body>
<script type="text/javascript">
function factorial(num) {
        if (num === 0 || num === 1)
            return 1;
        for (var i = num - 1; i >= 1; i--) {
            num *= i;
        }
        return num;
}
</script>
</body>
</html>
```

相关链接

- 使用 WebView（示例）
- WebView（示例）

Xamarin.Forms 平台功能

2018/10/26 • • Edit Online

Xamarin.Forms 是可扩展的并允许您使用的合并特定于平台的功能效果，自定义呈现器，则DependencyService，MessagingCenter，和的详细信息。

Android

本指南介绍如何通过更新现有 Xamarin.Forms Android 应用程序实现材料设计。

应用程序索引和深层链接

应用程序索引，否则将几个使用通过出现在搜索结果中了解相关信息后忘记的应用程序。深层链接允许应用程序响应通过导航到从深层链接引用的页面通常包含应用程序数据的搜索结果。

设备分类

如何使用 `Device` 类，以在共享的代码和用户界面（包括使用 XAML）创建特定于平台的行为。此外介绍了 `BeginInvokeOnMainThread` 时修改 UI 控件从后台线程，这非常重要。

iOS

可以通过执行某种 iOS 样式**Info.plist**和 `UIAppearance` API。本指南包括有关如何在 iOS 应用的 Xamarin.Forms 解决方案，包括核心 Spotlight 搜索中包括 iOS 9 功能的示例。

GTK

Xamarin.Forms 现在具有对 GTK # 应用程序的预览支持。

Mac

Xamarin.Forms 现在具有对 macOS 应用的预览支持。

本机窗体

本机窗体允许 Xamarin.Forms `ContentPage` -派生页可供本机 Xamarin.iOS、Xamarin.Android 和通用 Windows 平台 (UWP) 项目。

本机视图

从 iOS、Android 和通用 Windows 平台的本机视图，可以直接引用 Xamarin.Forms 从。上的本机视图，可以设置属性和事件处理程序，它们可以与 Xamarin.Forms 视图进行交互。

平台特定信息

平台特定信息，可使用的功能仅适用于特定的平台，而无需自定义呈现器或效果。

插件

在 Github、Nuget 和 Xamarin Component Store 帮助延长 Xamarin.Forms 应用上有可用的各种开源的插件。

Tizen

Tizen.NET 可以构建使用 Xamarin.Forms 和 Tizen.NET Framework 的.NET 应用程序。

Windows

Xamarin.Forms 具有 Windows 10 上支持的通用 Windows 平台 (UWP)。本文介绍如何将 UWP 项目添加到现有的 Xamarin.Forms 解决方案。

WPF

Xamarin.Forms 现在具有对 Windows Presentation Foundation (WPF) 应用程序的预览支持。

Android 平台功能

2018/6/9 • • Edit Online

平台支持

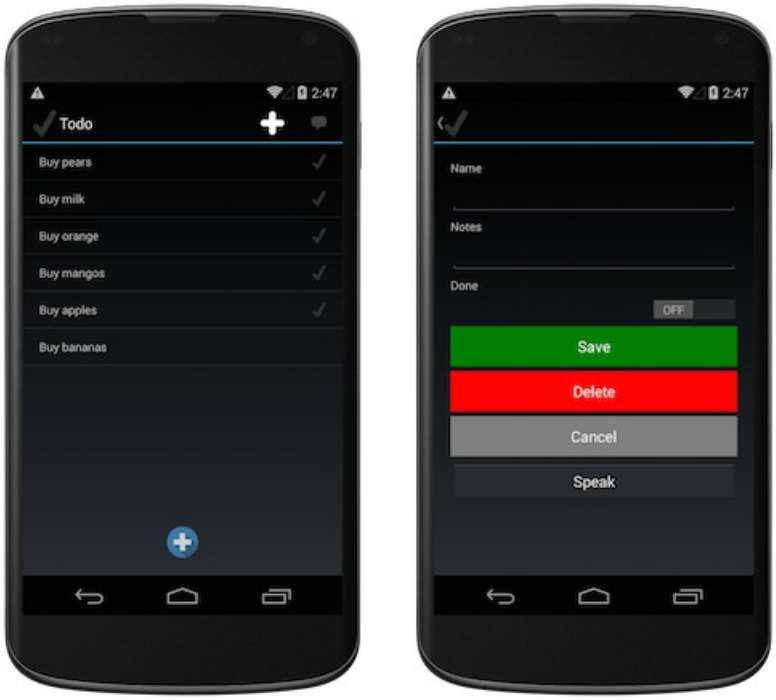
默认值 Xamarin.Forms Android 项目使用通常在 Android 5.0 之前的控件 renderering 旧样式。使用模板生成的应用程序具有 `FormsApplicationActivity` 作为其主要活动的基类。

通过 AppCompat 材料设计

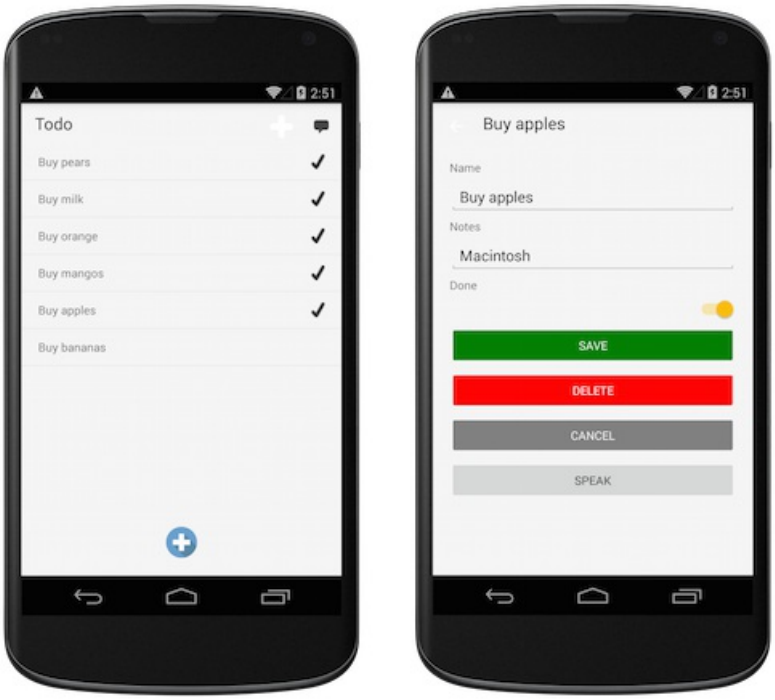
Xamarin.Forms 还具有可选 `FormsAppCompatActivity` 使用**AppCompat**由 Android 提供实现材料设计主题的功能。

若要向 Xamarin.Forms Android 项目中添加材料设计主题，请按照AppCompat 的安装说明支持

下面是**Todo**示例使用默认 `FormsApplicationActivity`：



这在升级项目以使用后是相同的代码和 `FormsAppCompatActivity`（和添加其他主题的信息）：



NOTE

使用时`FormsAppCompatActivity`、某些 Android 自定义呈现器的基本类都有所不同。

相关链接

- 添加材料设计支持

添加 AppCompat 和材料设计

2018/6/9 • • Edit Online

请按照下列步骤要转换现有的 Xamarin.Forms Android 应用程序以使用 AppCompat 和材料设计

概述

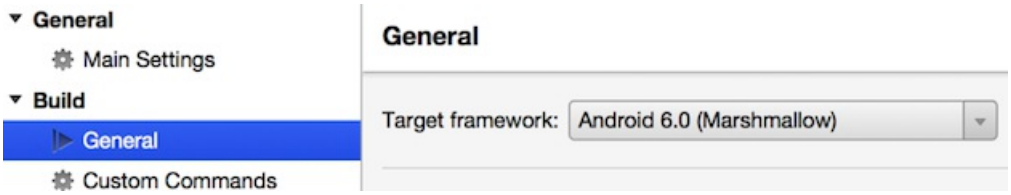
这些说明解释了如何更新现有 Xamarin.Forms Android 应用程序使用 AppCompat 库，并在 Xamarin.Forms 应用的 Android 版本中启用材料设计。

1.更新 Xamarin.Forms

确保解决方案使用 Xamarin.Forms 2.0 或更高版本。如果需要，请更新的 Xamarin.Forms Nuget 包为 2.0。

2.检查 Android 版本

确保 Android 项目的目标框架是 Android 6.0 (Marshmallow)。检查**Android 项目 > 选项 > 生成 > 常规**选择设置，以确保正确 framework:



3.添加新的主题，以支持材料设计

在 Android 项目中创建以下三个文件并粘贴以下内容中。Google 提供样式指南和颜色调色板生成器以帮助您选择为指定备用的配色方案。

Resources/values/colors.xml

```
<resources>
  <color name="primary">#2196F3</color>
  <color name="primaryDark">#1976D2</color>
  <color name="accent">#FFC107</color>
  <color name="window_background">#F5F5F5</color>
</resources>
```

Resources/values/style.xml

```
<resources>
  <style name="MyTheme" parent="MyTheme.Base">
  </style>
  <style name="MyTheme.Base" parent="Theme.AppCompat.Light.NoActionBar">
    <item name="colorPrimary">@color/primary</item>
    <item name="colorPrimaryDark">@color/primaryDark</item>
    <item name="colorAccent">@color/accent</item>
    <item name="android:windowBackground">@color/window_background</item>
    <item name="windowActionModeOverlay">true</item>
  </style>
</resources>
```

其他样式必须包括在值 **v21**文件夹运行 Android 棒糖形上和更高版本时将特定属性。

Resources/values-v21/style.xml

```
<resources>
  <style name="MyTheme" parent="MyTheme.Base">
    <!--If you are using MasterDetailPage you will want to set these, else you can leave them out-->
    <!--<item name="android:windowDrawsSystemBarBackgrounds">true</item>
    <item name="android:statusBarColor">@android:color/transparent</item>-->
  </style>
</resources>
```

4.更新 AndroidManifest.xml

若要确保此新主题信息是在中的，使用，将主题**AndroidManifest**文件添加 `android:theme="@style/MyTheme"`（按照原样，则使 XML 的其余部分）。

Properties/AndroidManifest.xml

```
...
<application android:label="AppName" android:icon="@drawable/icon"
  android:theme="@style/MyTheme">
...
```

5.提供工具栏和选项卡的布局

创建**Tabbar.axml**和**Toolbar.axml**文件中资**源/布局**目录和从下其内容中的粘贴：

Resources/layout/Tabbar.axml

```
<android.support.design.widget.TabLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/sliding_tabs"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:tabIndicatorColor="@android:color/white"
    app:tabGravity="fill"
    app:tabMode="fixed" />
```

已设置为选项卡的几个属性包括到选项卡的重力 `fill` 和模式来 `fixed` 。如果你有大量的选项卡可能想要切换这到可滚动的通读 Android TabLayout 文档若要了解详细信息。

Resources/layout/Toolbar.axml

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:minHeight="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
    app:popupTheme="@style/ThemeOverlay.AppCompat.Light"
    app:layout_scrollFlags="scroll|enterAlways" />
```

在这些文件中，我们正在创建你的应用程序都可能不同的工具栏的特定主题。请参阅Hello 工具栏博客文章以了解详细信息。

6.更新 `MainActivity`

在现有 Xamarin.Forms 应用中**MainActivity.cs**类将从继承 `FormsApplicationActivity` 。这必须替换 `FormsAppCompatActivity` 以启用新功能。

MainActivity.cs

```
public class MainActivity : FormsAppCompatActivity  // was FormsApplicationActivity
```

最后，"连接"中的步骤 5 中的新布局 `OnCreate` 方法，如下所示：

```
protected override void OnCreate(Bundle bundle)
{
  // set the layout resources first
  FormsAppCompatActivity.ToolbarResource = Resource.Layout.Toolbar;
  FormsAppCompatActivity.TabLayoutResource = Resource.Layout.Tabbar;

  // then call base.OnCreate and the Xamarin.Forms methods
  base.OnCreate(bundle);
  Forms.Init(this, bundle);
  LoadApplication(new App());
}
```

应用程序索引和深层链接

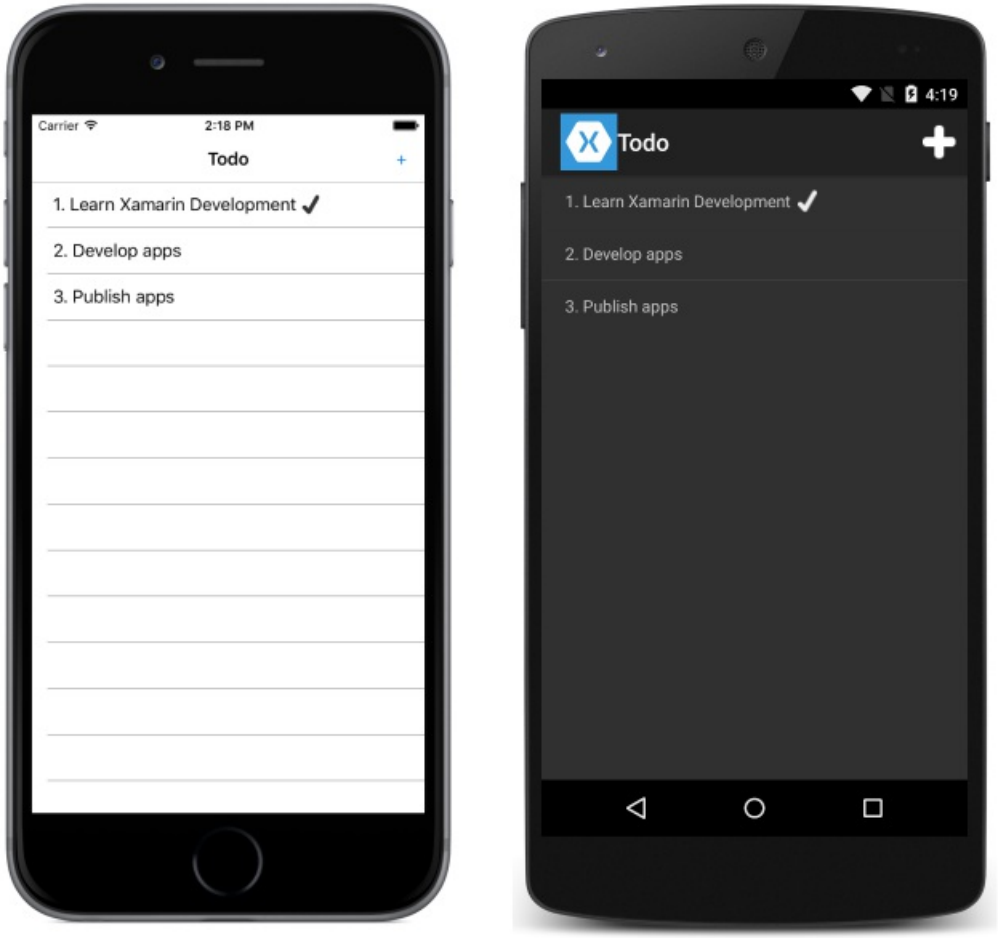
2018/7/13 • • Edit Online

应用程序索引，否则将几个使用通过出现在搜索结果中了解相关信息后忘记的应用程序。深层链接允许应用程序响应通过导航到从深层链接引用的页面通常包含应用程序数据的搜索结果。本文演示如何使用应用程序的索引和深层链接可供搜索 iOS 和 Android 设备上的 Xamarin.Forms 应用程序内容。

深度通过链接使用 Xamarin.Forms 和 Azure，Xamarin 学院课程

Xamarin.Forms 应用程序索引和深层链接提供一个用于 API 的应用程序索引在用户浏览应用程序通过发布元数据。使用 Spotlight 搜索，Google 搜索或 web 搜索，则可以为搜索创建索引的内容。点击搜索结果，其中包含深层链接就会触发事件的应用程序，可以处理和通常用于导航到从深层链接引用的页。

示例应用程序演示了其中的数据存储在本地 SQLite 数据库中，Todo 列表应用程序，如以下屏幕截图中所示：



每个 `TodoItem` 索引由用户创建的实例。然后可以使用特定于平台的搜索来查找该应用程序中的索引的数据。当用户点击该应用程序的搜索结果项上时，启动该应用程序时，`TodoItemPage` 导航到，和 `TodoItem` 引用从深层链接显示。

有关使用 SQLite 数据库的详细信息，请参阅使用本地数据库。

> **NOTE**
> Xamarin.Forms 应用程序索引和深度链接功能是仅适用于 iOS 和 Android 平台，分别需要 iOS 9 和 API 23。

安装

以下部分提供在 iOS 和 Android 平台上使用此功能的任何其他安装说明。

iOS

在 iOS 平台上是无需使用此功能的其他设置。

Android

在 Android 平台上有多个要使用应用程序索引和深度链接功能，必须满足的先决条件：

1. 你的应用程序的版本必须是实时 Google Play 上。
2. 必须针对 Google 的开发人员控制台中的应用程序注册的配套网站。应用程序与网站相关联后，可以是 Url 编制索引的网站和应用程序，然后在搜索结果中提供服务的工作。有关详细信息，请参阅应用对 Google 搜索的索引编制Google 的网站上。
3. 你的应用程序必须支持上的 HTTP URL 意向 `MainActivity` 类，它告知应用程序的索引编制哪些类型的数据的 URL 方案的应用程序可以响应。有关详细信息，请参阅配置的意向筛选器。

一旦满足这些先决条件，需要以下其他安装程序才能使用 Xamarin.Forms 应用程序索引和 Android 平台上的深层链接：

1. 安装Xamarin.Forms.AppLinks到 Android 应用程序项目的 NuGet 包。
2. 在中 `MainActivity.cs` 文件中，导入 `Xamarin.Forms.Platform.Android.AppLinks` 命名空间。
3. 在中 `MainActivity.OnCreate` 重写中，添加以下下面的代码行 `Forms.Init(this, bundle)`：

```
AndroidAppLinks.Init (this);
```

有关详细信息，请参阅深层链接内容与 Xamarin.Forms URL 导航Xamarin 博客上。

索引页

索引页，并将其公开到 Google 和 Spotlight 搜索的过程如下所示：

1. 创建 `AppLinkEntry` 包含索引页面中，并将其返回到页中，当用户在搜索结果中选择创建索引的内容的深层链接所需的元数据。
2. 注册 `AppLinkEntry` 实例要搜索的索引。

下面的代码示例演示如何创建 `AppLinkEntry` 实例：

```
AppLinkEntry GetAppLink (TodoItem item)
{
  var pageType = GetType ().ToString ();
  var pageLink = new AppLinkEntry {
    Title = item.Name,
    Description = item.Notes,
    AppLinkUri = new Uri (string.Format ("http://{0}/{1}?id={2}",
      App.AppName, pageType, WebUtility.UrlEncode (item.ID)), UriKind.RelativeOrAbsolute),
    IsLinkActive = true,
    Thumbnail = ImageSource.FromFile ("monkey.png")
  };

  return pageLink;
}
```

`AppLinkEntry` 实例包含多个属性的值和所需索引页创建的深层链接。`Title`，`Description`，以及 `Thumbnail` 属性用于标识创建索引的内容时出现在搜索结果中。`IsLinkActive` 属性设置为 `true` 以指示当前正在查看创建索引的内容。`AppLinkUri` 属性是 `Uri` 包含返回到当前页，并显示当前所需的信息 `TodoItem`。下面的示例演示一个示例

`Uri` 的示例应用程序：

```
http://deeplinking/DeepLinking.TodoItemPage?id=ec38ebd1-811e-4809-8a55-0d028fce7819
```

这 `Uri` 包含启动所需的所有信息 `deeplinking` 应用程序中，导航到 `DeepLinking.TodoItemPage`，并显示 `TodoItem` 具有 `ID` 的 `ec38ebd1-811e-4809-8a55-0d028fce7819`。

注册为进行索引的内容

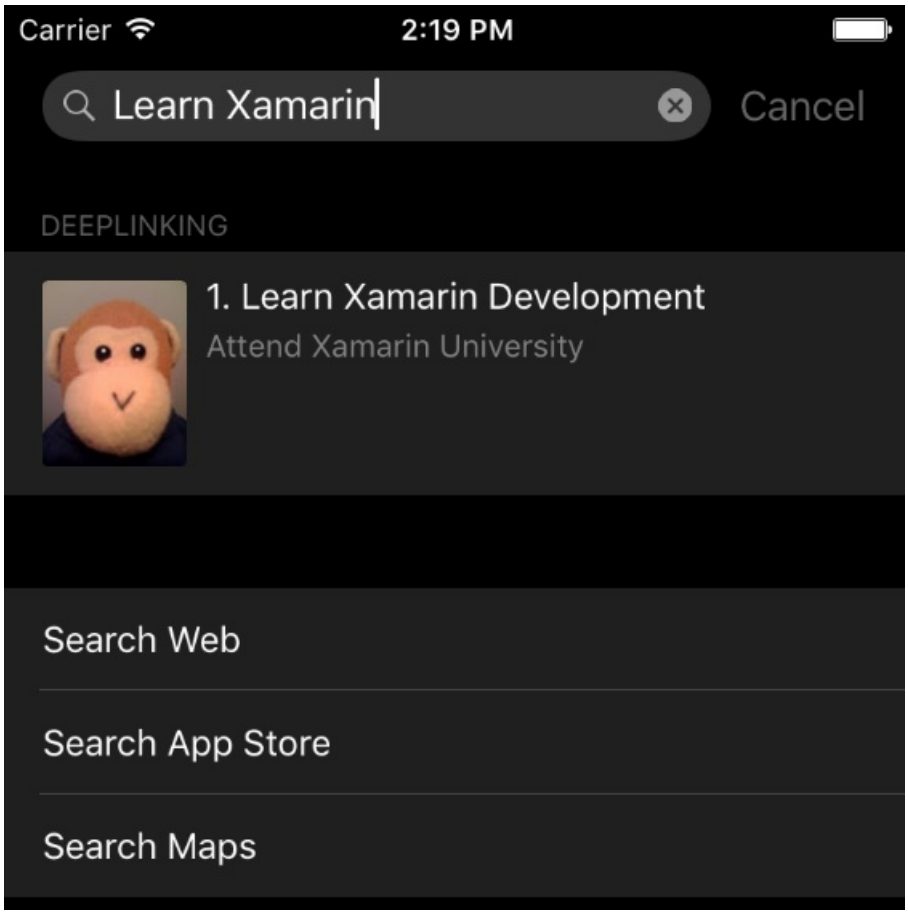
一次 `AppLinkEntry` 创建实例，必须将它注册为索引才会显示在搜索结果中。这通过实现 `RegisterLink` 方法，如下面的代码示例中所示：

```
Application.Current.AppLinks.RegisterLink (appLink);
```

这将添加 `AppLinkEntry` 应用程序的实例 `AppLinks` 集合。

> **NOTE**
> `RegisterLink` 方法还可以用于更新已编制索引的页面的内容。

一次 `AppLinkEntry` 实例具有已注册为编制索引，它可以出现在搜索结果中。下面的屏幕截图显示了出现在 iOS 平台上的搜索结果中创建索引的内容：



取消注册编制索引的内容

`DeregisterLink` 方法用于从搜索结果中，删除创建索引的内容如下面的代码示例中所示：

```
Application.Current.AppLinks.DeregisterLink (appLink);
```

此操作将删除 `AppLinkEntry` 从应用程序的实例 `AppLinks` 集合。

> **NOTE**
> 在 Android 上不能从搜索结果中移除创建索引的内容。

响应的深层链接

当创建索引的内容将出现在搜索结果和用户，选择 `App` 类的应用程序将收到要处理的请求 `Uri` 包含在创建索引的内容。可以在处理此请求 `OnAppLinkRequestReceived` 重写，如下面的代码示例中所示：

```
public class App : Application
{
  ...

  protected override async void OnAppLinkRequestReceived (Uri uri)
  {
    string appDomain = "http://" + App.AppName.ToLowerInvariant () + "/";
    if (!uri.ToString ().ToLowerInvariant ().StartsWith (appDomain)) {
      return;
    }

    string pageUrl = uri.ToString ().Replace (appDomain, string.Empty).Trim ();
    var parts = pageUrl.Split ('?');
    string page = parts [0];
    string pageParameter = parts [1].Replace ("id=", string.Empty);

    var formsPage = Activator.CreateInstance (Type.GetType (page));
    var todoItemPage = formsPage as TodoItemPage;
    if (todoItemPage != null) {
      var todoItem = App.Database.Find (pageParameter);
      todoItemPage.BindingContext = todoItem;
      await MainPage.Navigation.PushAsync (formsPage as Page);
    }

    base.OnAppLinkRequestReceived (uri);
  }
}
```

`OnAppLinkRequestReceived` 方法会检查收到 `Uri` 适用于应用程序，分析之前 `Uri` 到页后，可以导航到并使用参数来传递到页。要进行创建后，导航到的页的实例和 `TodoItem` 表示参数检索到的页。 `BindingContext` 要导航到的页的设置为 `TodoItem` 。这可确保，当 `TodoItemPage` 情况下将显示 `PushAsync` 方法，它将显示 `TodoItem` 其 `ID` 深层链接中包含。

使内容可供搜索索引

显示通过深层链接所代表的页，则每次 `AppLinkEntry.IsLinkActive` 属性设置为 `true` 。IOS 和 Android 上，这使得 `AppLinkEntry` 为搜索编制索引，并仅在 iOS 上可用的实例，它还使 `AppLinkEntry` 实例可用于切换。有关切换的详细信息，请参阅简介切换。

下面的代码示例演示了如何设置 `AppLinkEntry.IsLinkActive` 属性设置为 `true` 中 `Page.OnAppearing` 重写：

```
protected override void OnAppearing ()
{
  appLink = GetAppLink (BindingContext as TodoItem);
  if (appLink != null) {
    appLink.IsLinkActive = true;
  }
}
```

同样，通过深层链接所代表的页导航离开，`AppLinkEntry.IsLinkActive` 属性设置为 `false`。在 iOS 和 Android 上，这会停止 `AppLinkEntry` 实例正被播发的搜索索引，并在 iOS 唯一，it 也停止公布自己 `AppLinkEntry` 移交的实例。可以在完成此 `Page.OnDisappearing` 重写，如下面的代码示例中所示：

```
protected override void OnDisappearing ()
{
  if (appLink != null) {
    appLink.IsLinkActive = false;
  }
}
```

切换到提供数据

在 iOS 上，索引页时，可以存储特定于应用程序的数据。这通过将数据添加到实现 `KeyValues` 回收，这是 `Dictionary<string, string>` 中切换存储使用的键 / 值对。切换是为用户启动一个其设备上的一个活动并在其设备的另一台继续该活动（如由用户的 iCloud 帐户标识）的方法。下面的代码显示了存储特定于应用程序的键 / 值对的示例：

```
var pageLink = new AppLinkEntry {
  ...
};
pageLink.KeyValues.Add("appName", App.AppName);
pageLink.KeyValues.Add("companyName", "Xamarin");
```

中存储的值 `KeyValues` 集合将存储在索引页上，为元数据，并且将在用户点击搜索结果，其中包含深层链接（或切换用于在另一台查看内容时还原登录的设备）。

此外，可以指定以下项的值：

- `contentType` – `string`，指定创建索引的内容的统一类型标识符。推荐的约定要用于此值是页的包含创建索引的内容的类型名称。
- `associatedWebPage` – `string`，表示要访问如果创建索引的内容也可以查看在 web 上，或该应用程序支持 Safari 的深层链接的网页。
- `shouldAddToPublicIndex` – `string` 的任一 `true` 或 `false`，它控制是否将创建索引的内容添加到 Apple 的公有云的索引，然后向尚未在其 iOS 设备安装应用程序用户提供。但是，只是因为内容已设置为公共编制索引，它并不意味着，它将自动添加到 Apple 的公有云索引。有关详细信息，请参阅公共搜索索引。请注意，应将此项设置为 `false` 添加到个人数据时 `KeyValues` 集合。

> **NOTE**
> `KeyValues` 集合不 Android 平台上使用。

有关切换的详细信息，请参阅简介切换。

总结

本文演示了如何使用应用程序的索引和深层链接可供搜索 iOS 和 Android 设备上的 Xamarin.Forms 应用程序内容。应用程序索引允许应用程序通过出现在搜索结果，否则会在几个使用后忘记有关了解相关信息。

相关链接

- 深层链接（示例）
- iOS 搜索 Api
- 在 Android 6.0 中将应用链接
- AppLinkEntry
- IAppLinkEntry
- IAppLinks

Xamarin.Forms 设备类

2018/10/26 • • Edit Online

`Device` 类包含大量属性和方法，以帮助开发人员自定义布局和根据每个平台的功能。

除了方法和属性在特定硬件类型和大小，让代码面向 `Device` 类包括BeginInvokeOnMainThread方法从与 UI 交互控件时应使用该方法后台线程。

提供特定于平台的值

在 Xamarin.Forms 2.3.4 之前，的平台运行应用程序无法获取通过检查 `Device.OS` 属性并将对其进行比较 `TargetPlatform.iOS`， `TargetPlatform.Android`， `TargetPlatform.WinPhone`，并且 `TargetPlatform.Windows` 枚举值。同样，之一 `Device.OnPlatform` 重载可用于提供对控件的特定于平台的值。

但是，因为 Xamarin.Forms 2.3.4 这些 Api 已弃用并替换。`Device` 类现在包含的标识平台 – 公共字符串常量 `Device.iOS`， `Device.Android`， `Device.WinPhone`（不推荐使用），`Device.WinRT`（已弃用）`Device.UWP`，并 `Device.macOS`。同样，`Device.OnPlatform` 已替换重载 `OnPlatform` 并 `On` Api。

在C#，可以通过创建提供特定于平台的值 `switch` 语句 `Device.RuntimePlatform` 属性，并提供 `case` 语句所需的平台：

```
double top;
switch (Device.RuntimePlatform)
{
  case Device.iOS:
    top = 20;
    break;
  case Device.Android:
  case Device.UWP:
  default:
    top = 0;
    break;
}
layout.Margin = new Thickness(5, top, 5, 0);
```

`OnPlatform` 并 `On` 类提供 XAML 中的相同功能：

```
<StackLayout>
  <StackLayout.Margin>
    <OnPlatform x:TypeArguments="Thickness">
      <On Platform="iOS" Value="0,20,0,0" />
      <On Platform="Android, UWP" Value="0,0,0,0" />
    </OnPlatform>
  </StackLayout.Margin>
  ...
</StackLayout>
```

`OnPlatform` 类是必须使用实例化一个泛型类 `x:TypeArguments` 匹配目标类型的属性。在中 `On` 类，`Platform` 属性可以接受单个 `string` 值或以逗号分隔的多个 `string` 值。

> **IMPORTANT**
>
> 提供不正确 `Platform` 属性中的值 `On` 类不会导致错误。相反，该代码将执行且不应用任何特定于平台的值。

或者，`OnPlatform`标记扩展可用于在 XAML 中自定义根据每个平台的 UI 外观。有关详细信息，请参阅OnPlatform 标记扩展。

Device.Idiom

`Device.Idiom`属性可用于更改布局或功能，具体取决于设备应用程序上运行。`TargetIdiom`枚举包含的以下值：

- **Phone** – iPhone、iPod touch 和 Android 设备窄于 600 dip ^
- **平板电脑** – iPad，Windows 设备和 Android 设备宽于 600 dip ^
- **桌面** – 仅在返回UWP 应用Windows 10 桌面计算机上 (返回`Phone`移动 Windows 设备，包括连续体方案中)
- **电视** – Tizen TV 设备
- **观看** – Tizen 监视设备
- **不支持** – 未使用

^ *dip 不一定是物理像素计数*

`Idiom`属性是用于构建充分利用较大的屏幕，此类的布局特别有用：

```
if (Device.Idiom == TargetIdiom.Phone) {
    // layout views vertically
} else {
    // layout views horizontally for a larger display (tablet or desktop)
}
```

`OnIdiom`类提供在 XAML 中的相同功能：

```
<StackLayout>
    <StackLayout.Margin>
        <OnIdiom x:TypeArguments="Thickness">
            <OnIdiom.Phone>0,20,0,0</OnIdiom.Phone>
            <OnIdiom.Tablet>0,40,0,0</OnIdiom.Tablet>
            <OnIdiom.Desktop>0,60,0,0</OnIdiom.Desktop>
        </OnIdiom>
    </StackLayout.Margin>
    ...
</StackLayout>
```

`OnIdiom`类是必须使用实例化一个泛型类`x:TypeArguments`匹配目标类型的属性。

或者，`OnIdiom`标记扩展可用于在 XAML 中自定义 UI 外观基于的设备运行应用程序的惯用语法。有关详细信息，请参阅OnIdiom 标记扩展。

Device.FlowDirection

`Device.FlowDirection`值检索`FlowDirection`枚举值，该值表示当前正由设备的流方向。流方向是密切关注扫描的页上的 UI 元素的方向。枚举值为：

- `LeftToRight`
- `RightToRight`
- `MatchParent`

在 XAML 中，`Device.FlowDirection`值可以通过使用`x:Static`标记扩展：

```
<ContentPage ... FlowDirection="{x:Static Device.FlowDirection}"> />
```

中的等效代码C#是：

```
this.FlowDirection = Device.FlowDirection;
```

有关流方向的详细信息，请参阅从右到左本地化。

Device.Styles

`Styles` 属性包含可应用于某些控件的内置样式定义（如 `Label`）`Style` 属性。可用的样式包括：

- BodyStyle
- CaptionStyle
- ListItemDetailTextStyle
- ListItemTextStyle
- SubtitleStyle
- TitleStyle

Device.GetNamedSize

`GetNamedSize` 设置时，可以使用 `FontSize` 在C#代码：

```
myLabel.FontSize = Device.GetNamedSize (NamedSize.Small, myLabel);
someLabel.FontSize = Device.OnPlatform (
      24,         // hardcoded size
      Device.GetNamedSize (NamedSize.Medium, someLabel),
      Device.GetNamedSize (NamedSize.Large, someLabel)
);
```

Device.OpenUri

`OpenUri` 方法可用于触发对基础平台，如本机 web 浏览器中打开一个 URL 的操作（**Safari**在 iOS 上或**Internet**在 Android 上）。

```
Device.OpenUri(new Uri("https://evolve.xamarin.com/"));
```

WebView 示例包括的示例使用 `OpenUri` 若要打开的 Url 和也会触发电话呼叫。

地图示例还使用 `Device.OpenUri` 以显示地图和方向使用本机映射iOS 和 Android 上的应用。

Device.StartTimer

`Device` 类还具有 `StartTimer` 方法提供的适用于 Xamarin.Forms 的常见代码，包括.NET Standard 库的触发时间相关的任务的简单方法。传递 `TimeSpan` 若要设置的间隔，并返回 `true` 保持运行的计时器或 `false` 当前调用后将其停止。

```
Device.StartTimer (new TimeSpan (0, 0, 60), () => {
    // do something every 60 seconds
    return true; // runs again, or false to stop
});
```

如果内部计时器的代码与用户界面进行交互（例如，设置的文本 `Label`，或显示的警报）应在内部完成 `BeginInvokeOnMainThread` 表达式（见下文）。

Device.BeginInvokeOnMainThread

后台线程，如在计时器或异步操作，例如 web 请求完成处理程序中运行的代码应永远不会访问用户界面元素。任何需要更新用户界面的后台代码都应纳入 `BeginInvokeOnMainThread` 。此命令的等效 `InvokeOnMainThread` 在 iOS 上，`RunOnUiThread` 在 Android 上，和 `Dispatcher.RunAsync` 通用 Windows 平台上。

Xamarin.Forms 代码是：

```
Device.BeginInvokeOnMainThread ( () => {
  // interact with UI elements
});
```

请注意该方法使用 `async/await` 不需要使用 `BeginInvokeOnMainThread` 如果运行从主 UI 线程。

总结

Xamarin.Forms `Device` 类允许根据每个平台的功能和布局精细地控制-甚至共同代码（.NET Standard 库项目或共享的项目）。

相关链接

- 设备示例
- 样式示例
- 设备

iOS 平台功能

2018/7/26 • • Edit Online

特定于 iOS 的格式设置

Xamarin.Forms 支持跨平台用户界面样式和颜色设置的但有其他选项来设置你的 iOS 项目中使用特定于平台的 Api 的 iOS 的主题。

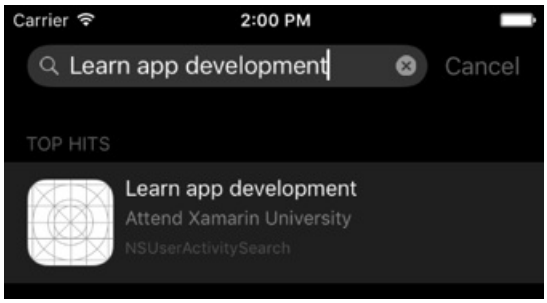
阅读更多有关格式设置使用特定于 iOS 的 Api，如用户界面**Info.plist**配置和 `UIAppearance` API。

iOS 9 功能

使用自定义呈现器，则DependencyService，并MessagingCenter，可以纳入各种各样的本机功能集成到适用于 iOS 的 Xamarin.Forms 应用。

以下方案演示如何将 iOS 9 功能合并到 Xamarin.Forms 应用程序的 iOS 部分：

- CoreSpotlight
- NSUserActivity



添加特定于 iOS 的格式设置

2018/8/23 • • Edit Online

一种方法来设置特定于 iOS 的格式设置是创建自定义呈现器的控件和设置特定于平台的样式和颜色的每个平台。

其他选项来控制包括 Xamarin.Forms iOS 应用的外观的方式：

- 配置显示在选项 **Info.plist**
- 设置控件样式通过 `UIAppearance` API

下面将讨论以下替代方法。

自定义 Info.plist

Info.plist文件允许您配置 iOS 应用程序 renderering，例如如何（以及是否）显示状态栏的某些方面。

例如，待办事项示例使用下面的代码在所有平台上设置的导航栏的颜色和文本颜色：

```
var nav = new NavigationPage (new TodoListPage ());
nav.BarBackgroundColor = Color.FromHex("91CA47");
nav.BarTextColor = Color.White;
```

结果如下面的屏幕代码段所示。请注意状态栏项是黑色（这不能设置在 Xamarin.Forms 中由于它是特定于平台的功能）。

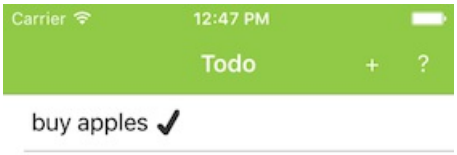
理想情况下状态栏也是白色-我们可以直接在 iOS 项目中完成的内容。以下条目添加到**Info.plist**强制状态栏为白色：

Status bar style	String	White
View controller-based status bar appearance	Boolean	No

或编辑相应**Info.plist**文件直接以包括：

```
<key>UIStatusBarStyle</key>
<string>UIStatusBarStyleLightContent</string>
<key>UIViewControllerBasedStatusBarAppearance</key>
<false/>
```

现在，当应用运行时，在导航栏为绿色，其文本为白色（由于 Xamarin.Forms 格式设置）*和*状态栏文本也是白色归功于特定于 iOS 的配置：



UIAppearance API

`UIAppearance` API可用于 iOS 的许多控件上设置视觉对象属性*而无需*无需创建自定义呈现器。

添加到代码的单行**AppDelegate.cs** `FinishedLaunching` 方法可设置给定的类型使用的所有控件的都样式及其 `Appearance` 属性。下面的代码包含两个示例的全局设置样式选项卡栏，切换控件：

AppDelegate.cs iOS 项目中

```
public override bool FinishedLaunching (UIApplication app, NSDictionary options)
{
  // tab bar
    UITabBar.Appearance.SelectedImageTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
  // switch
    UISwitch.Appearance.OnTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
    // required Xamarin.Forms code
    Forms.Init ();
    LoadApplication (new App ());
    return base.FinishedLaunching (app, options);
}
```

UITabBar

默认情况下中的选定选项卡栏图标 `TabbedPage` 是蓝色：

若要更改此行为，请设置 `UITabBar.Appearance` 属性：

```
UITabBar.Appearance.SelectedImageTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
```

这将导致所选的选项卡以显示为绿色：

使用此 API 允许您自定义外观的 Xamarin.Forms `TabbedPage` 使用非常少的代码在 iOS 上。请参阅自定义选项卡方案有关使用自定义呈现器设置特定字体为选项卡的详细信息。

UISwitch

`Switch` 控件是可以轻松地设置样式的另一个示例：

```
UISwitch.Appearance.OnTintColor = UIColor.FromRGB(0x91, 0xCA, 0x47); // green
```

这些两个屏幕截图显示默认值 `UISwitch` 上的左窗格和自定义的版本控制 (设置 `Appearance`) 中右侧待办事项示例：

其他控件

其默认颜色和其他属性集使用多个 iOS 用户界面控件可以具有 `UIAppearance` API。

相关链接

- UIAppearance
- 自定义选项卡

GTK # 平台安装程序

2018/10/26 • • Edit Online

Xamarin.Forms 现在具有对 GTK # 应用程序的预览支持。GTK # 是允许使用 Mono 和.NET 的图形应用程序的开发完全本机 GNOME GTK + 工具包和多种 GNOME 库链接起来，一个图形用户界面工具包。本文演示如何将一个 GTK # 项目添加到 Xamarin.Forms 解决方案。

启动、创建新的 Xamarin.Forms 解决方案，或使用现有的 Xamarin.Forms 解决方案，例如之前，**GameOfLife**。

> **NOTE**
> 尽管本文着重介绍到 Xamarin.Forms 解决方案在 VS2017 和 Visual Studio for Mac 添加 GTK # 应用程序，它还可以执行在MonoDevelop适用于 Linux。

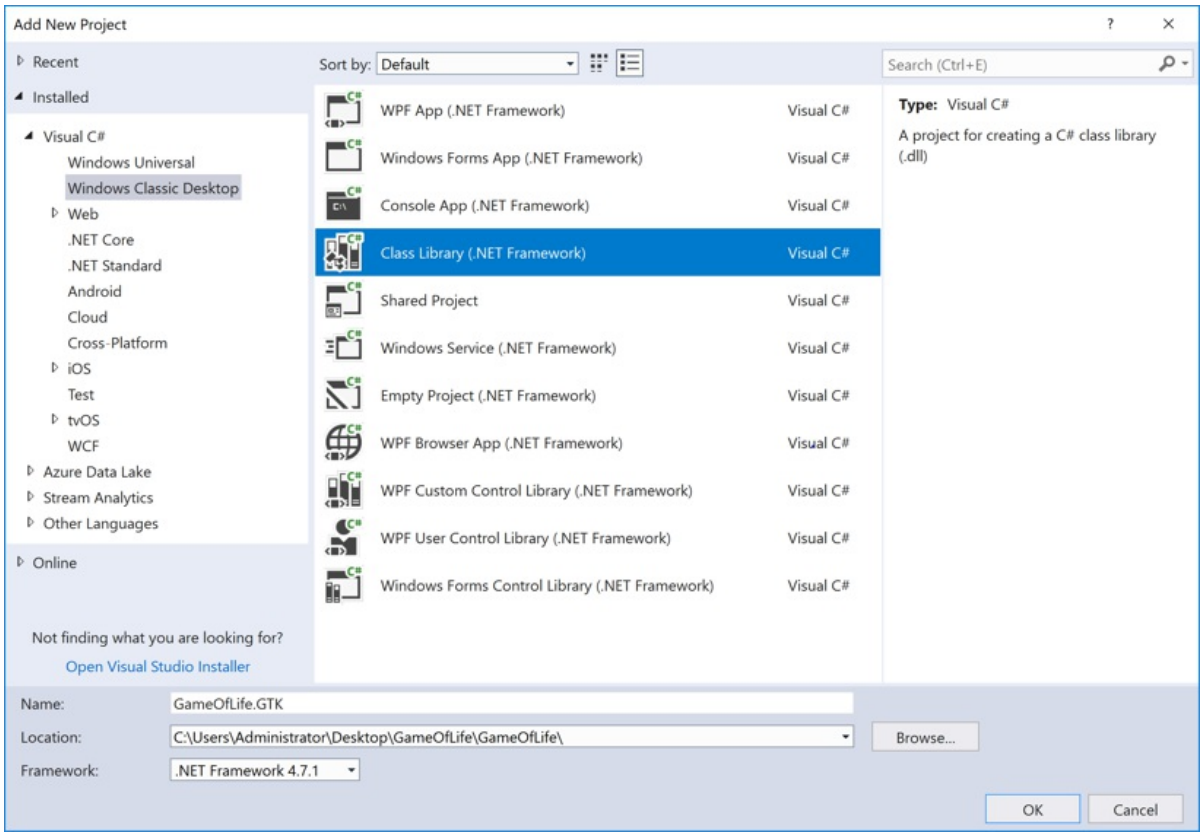
添加一个 GTK # 应用程序

GTK # 适用于 macOS 和 Linux 安装的一部分Mono。用于.NET 的 GTK # 可以安装在 Windows 与GTK # 安装程序。

- Visual Studio
- Visual Studio for Mac

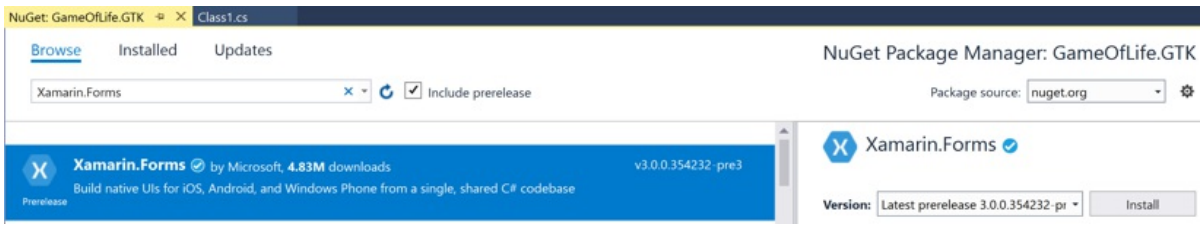
按照这些说明添加将在 Windows 桌面上运行的 GTK # 应用：

1. 在 Visual Studio 2017 中，右键单击解决方案名称在**解决方案资源管理器**，然后选择**添加 > 新建项目...**。
2. 在中**新的项目**窗口中的，在左侧选择**Visual C# 并Windows 经典桌面**。在项目类型列表中，选择类库**(.NET Framework)**，并确保**Framework**下拉列表设置为.NET Framework 4.7 最小值。
3. 键入与项目的名称**GTK**扩展，例如**GameOfLife.GTK**。单击**浏览**按钮，选择包含在其他平台的文件夹的项目，然后按选择**文件**夹。这会将 GTK 项目与解决方案中的其他项目相同的目录中。



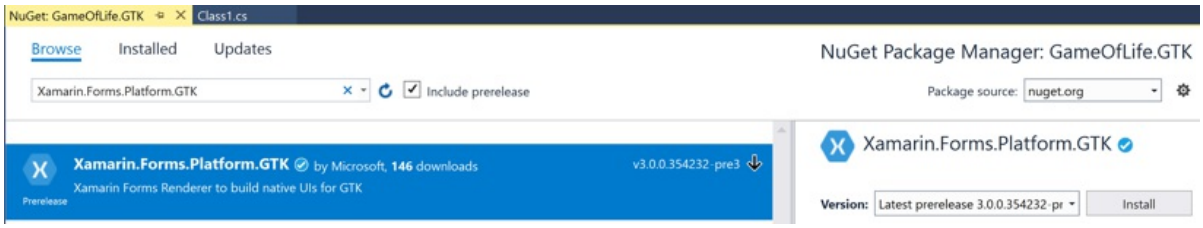
按**确定**按钮创建项目。

4. 在中**解决方案**资**源管理器**，右键单击**新的** GTK 项目**并**选择**管理 NuGet 包**。选择**浏览**选项**卡**，**并搜索Xamarin.Forms** 3.0 **或更高**版本。



选择包，**然后**单击**安装**按钮。

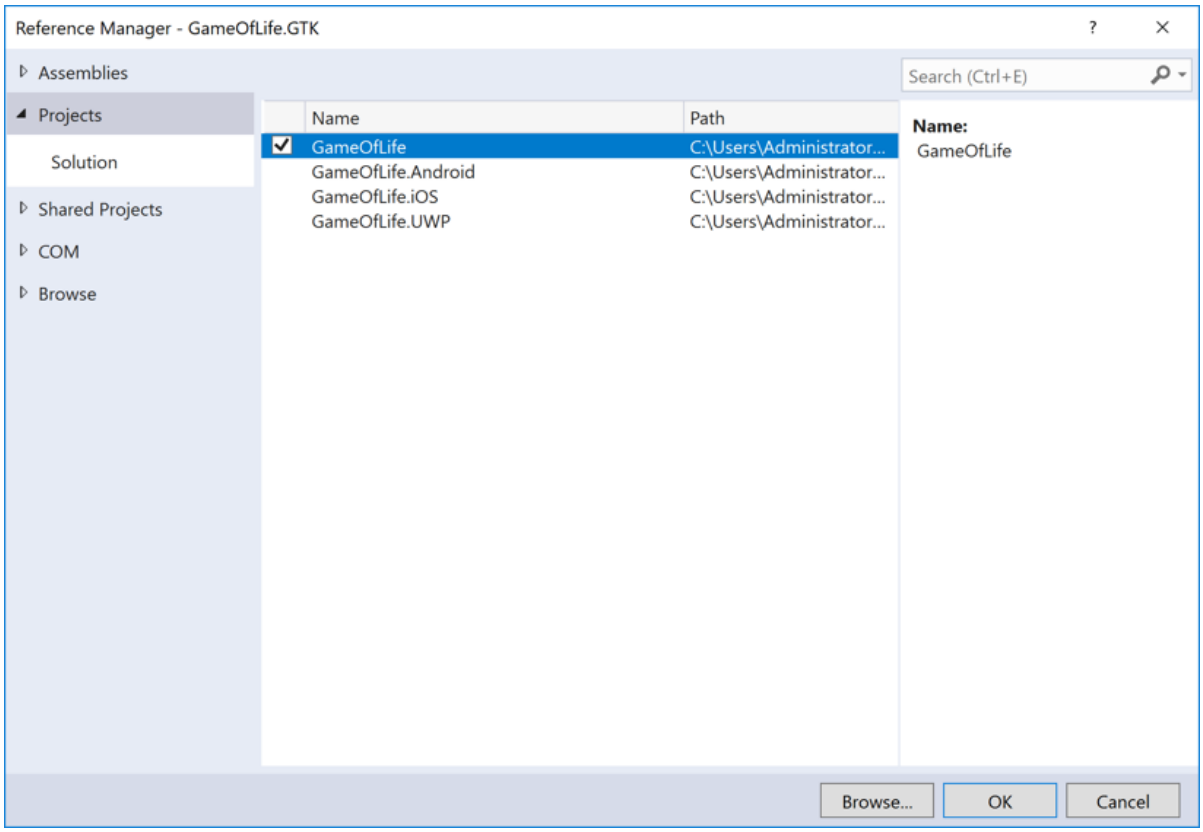
5. 在其中搜索**Xamarin.Forms.Platform.GTK** 3.0 包或更高版本。



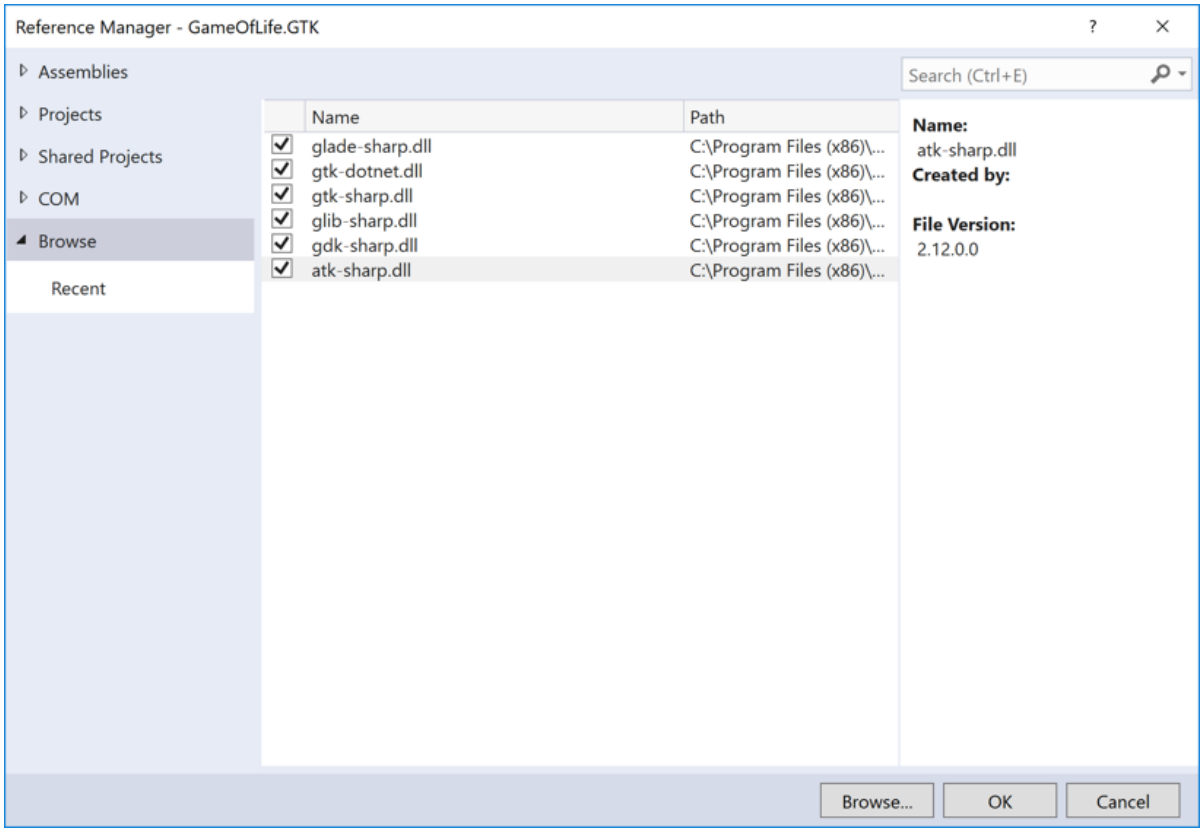
选择包，**然后**单击**安装**按钮。

6. 在中**解决方案**资**源管理器**，右键单击**解决方案名称**，**然后**选择为**解决方案管理 NuGet 包**。选择**更新**选项卡和**Xamarin.Forms**包。选择**所有**项目，**并将其更新**为与 GTK 项目**使用相同的** Xamarin.Forms **版本**。

7. 在中**解决方案**资**源管理器**，右键单击**引用**GTK 项目中。在中**引用管理器**对话**框**中，选择**项目左**侧，**并**检查到.NET Standard **或共享**项目**旁的复**选**框**：



8. 在中**引用管理器**对话框中，按浏览按钮并浏览到**C:\Program Files (x86)\GtkSharp\2.12\lib**文件夹，然后选择**atk sharp.dll**，**gdk sharp.dll**，**glade sharp.dll**，**glib sharp.dll**，**gtk dotnet.dll**，**gtk sharp.dll**文件。



按**确定**按钮以添加引用。

9. 在 GTK 项目中，重命名**Class1.cs**到**Program.cs**。

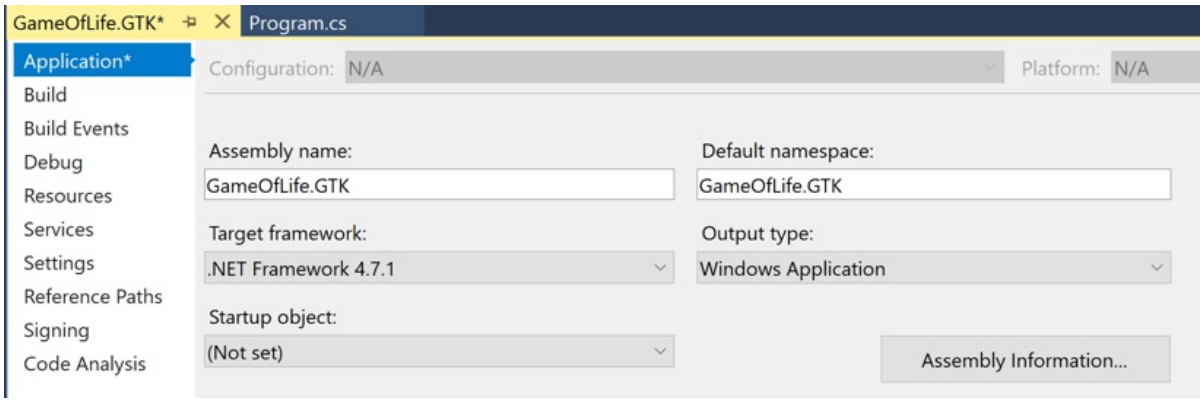
10. 在 GTK 项目中，编辑**Program.cs**文件，以便它类似于以下代码：

```
using System;
using Xamarin.Forms;
using Xamarin.Forms.Platform.GTK;

namespace GameOfLife.GTK
{
    class MainClass
    {
        [STAThread]
        public static void Main(string[] args)
        {
            Gtk.Application.Init();
            Forms.Init();

            var app = new App();
            var window = new FormsWindow();
            window.LoadApplication(app);
            window.SetApplicationTitle("Game of Life");
            window.Show();

            Gtk.Application.Run();
        }
    }
}
```

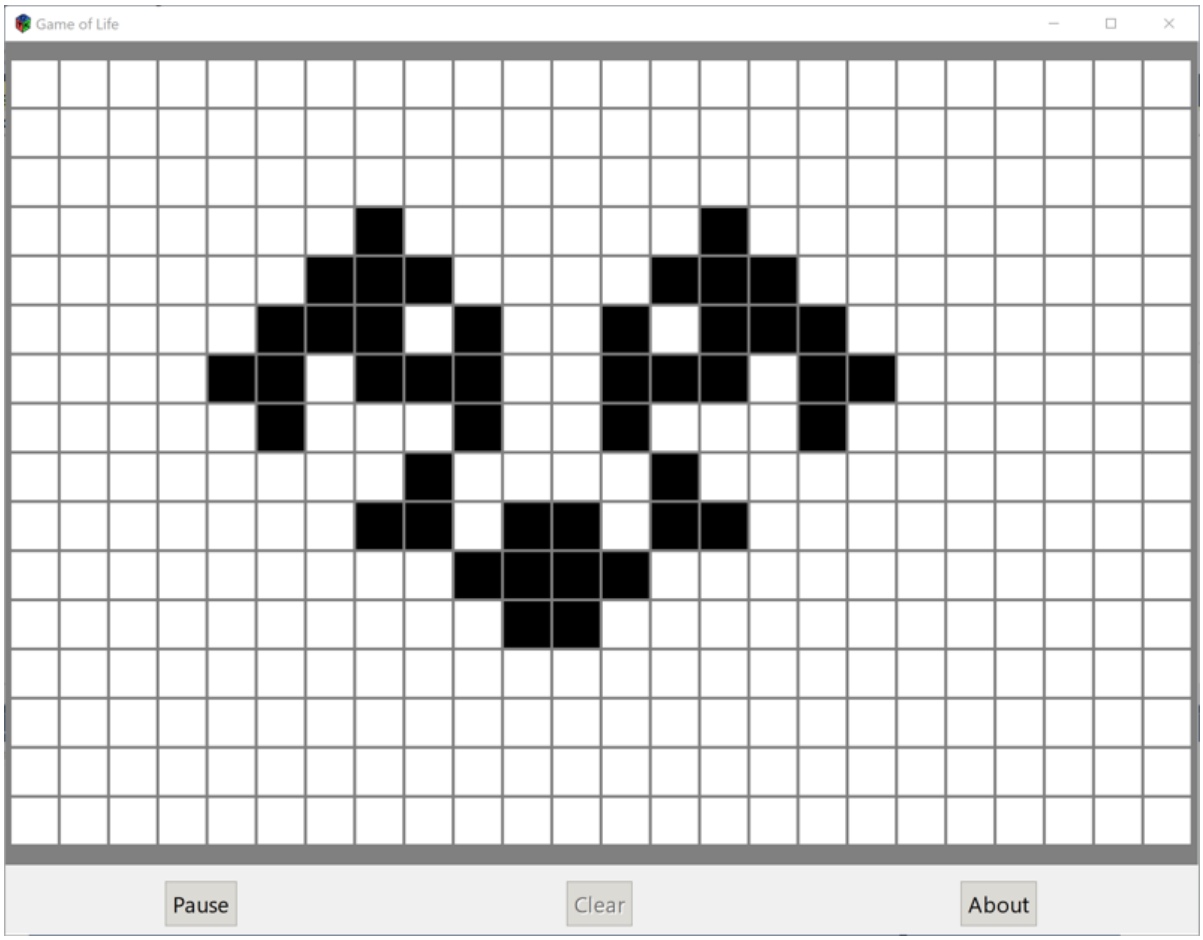
此代码初始化 GTK # 和 Xamarin.Forms 中，创建一个应用程序窗口，并运行的应用程序。

11. 在中**解决方案**资**源管理器**，右键单击 GTK 项目并选择**属性**。

12. 在中**属性**窗口中，选择应**用程序**选项卡并更改输出类**型**下拉列表到**Windows** 应**用程序**。



13. 在中**解决方案**资**源管理器**，右键单击 WPF 项目并选择设为**启**动项**目**。按 f5 键以与 Visual Studio 调试器在 Windows 桌面上运行该程序：



后续步骤

平台特定信息

您可以确定在 Xamarin.Forms 应用程序运行 XAML 或代码中哪些平台。这样即可在 GTK # 上运行时更改程序特性。在代码中，进行比较的值 `Device.RuntimePlatform` 与 `Device.GTK` 常量（它等于字符串"GTK"）。如果没有匹配项，该应用程序正在上 GTK #。

在 XAML 中，你可以使用 `OnPlatform` 标记来选择特定于平台的属性值：

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White" />
        <On Platform="macOS" Value="White" />
        <On Platform="Android" Value="Black" />
        <On Platform="GTK" Value="Blue" />
    </OnPlatform>
</Button.TextColor>
```

应用程序图标

在启动时，可以设置应用图标：

```
window.SetApplicationIcon("icon.png");
```

主题

有各种主题可用于 GTK # 中，并且它们可以从一个 Xamarin.Forms 应用程序使用：

```
GtkThemes.Init ();
GtkThemes.LoadCustomTheme ("Themes/gtkrc");
```

本机窗体

本机窗体允许 Xamarin.Forms `ContentPage` -派生页可供本机项目，包括 GTK # 项目。这可以通过创建的实例来实现 `ContentPage` -派生页，并将其转换为本机 GTK # 类型使用 `CreateContainer` 扩展方法：

```
var settingsView = new SettingsView().CreateContainer();
vbox.PackEnd(settingsView, true, true, 0);
```

有关本机窗体的详细信息，请参阅本机窗体。

问题

这是预览版，因此可以预见，并非所有内容都可用于生产。当前的实现状态，请参阅状态，和当前的已知问题，请参阅挂起和已知问题。

Mac 平台安装程序

2018/11/14 • • Edit Online

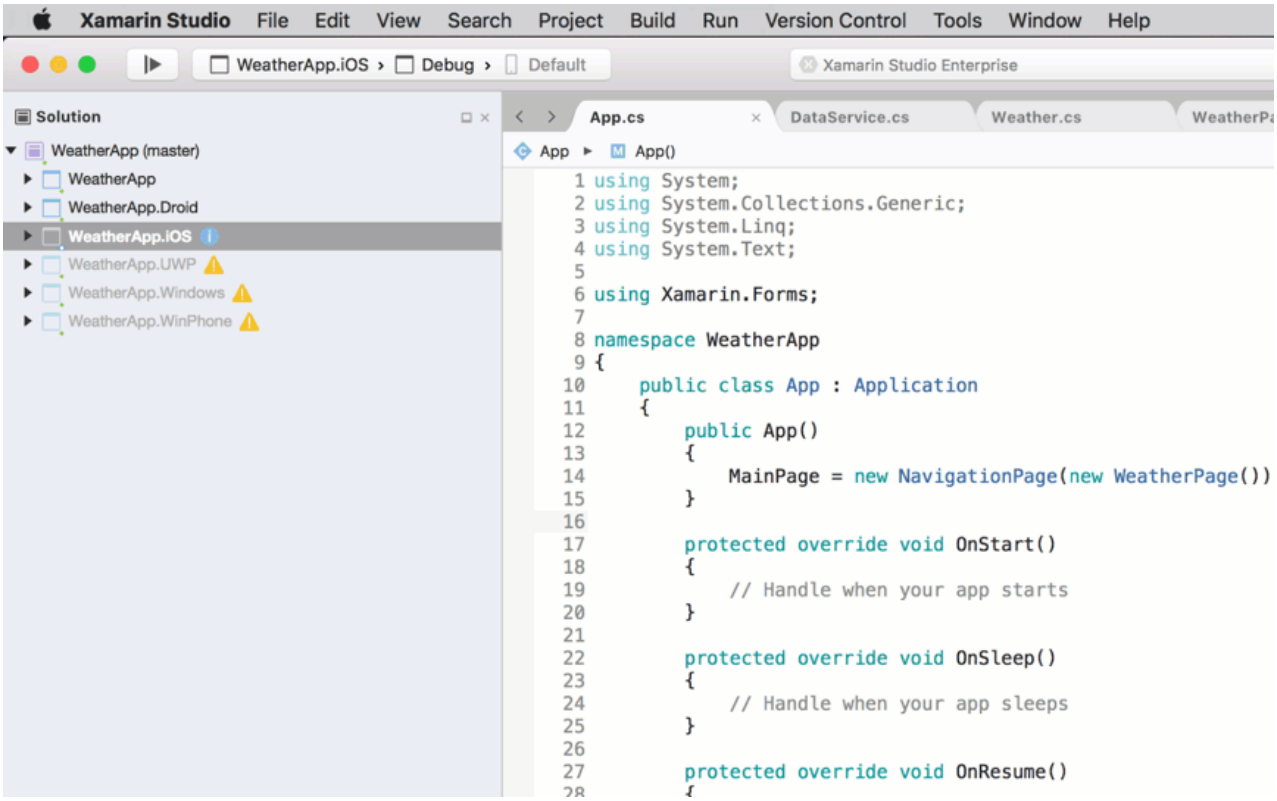
在开始之前，创建（或使用现有）Xamarin.Forms 项目。你只能添加使用 Visual Studio for mac 的 Mac 应用

通过将 macOS 项目添加到 Xamarin.Forms，Xamarin University

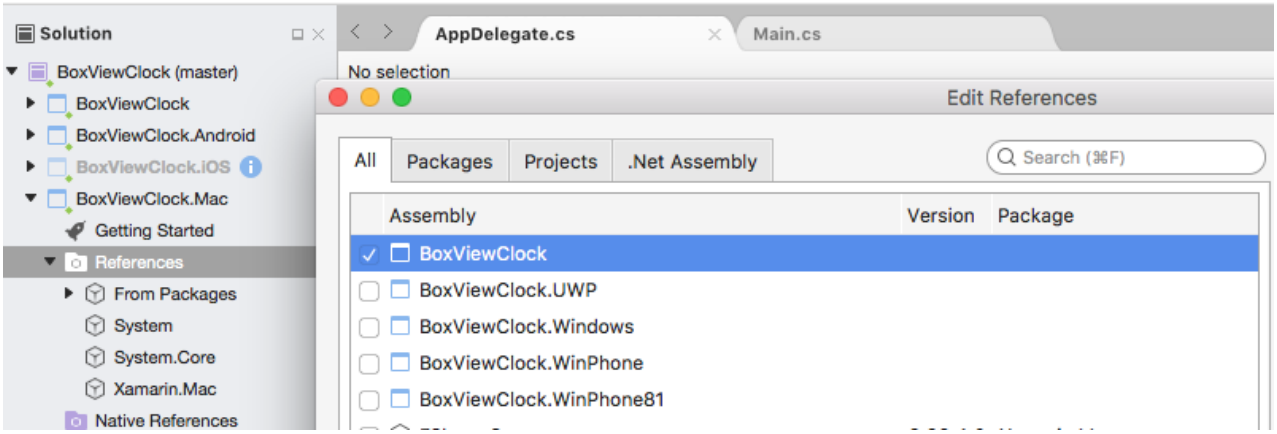
添加 Mac 应用

按照这些说明添加将在 macOS Sierra 和 macOS El Capitan 运行的 Mac 应用：

1. 在 Visual Studio for Mac 中，右键单击现有的 Xamarin.Forms 解决方案并选择**添加 > 添加新项目...**
2. 在**新的项目**窗口中选择**Mac > 应用程序 > Cocoa 应用**然后按**下一步**。
3. 类型**应用名称**（和（可选）选择 Dock 项不同的名称），然后按**下一步**。
4. 查看**配置并按创建**。这些步骤中所示如下：



5. 在 Mac 项目中，右键单击**包 > 添加包...** 以添加Xamarin.Forms/2.3.5.235-pre2 NuGet。您还应更新到此版本的其他项目。
6. 在 Mac 项目中，右键单击**引用并添加到** Xamarin.Forms 项目（共享项目或.NET Standard 库项目）的引用。



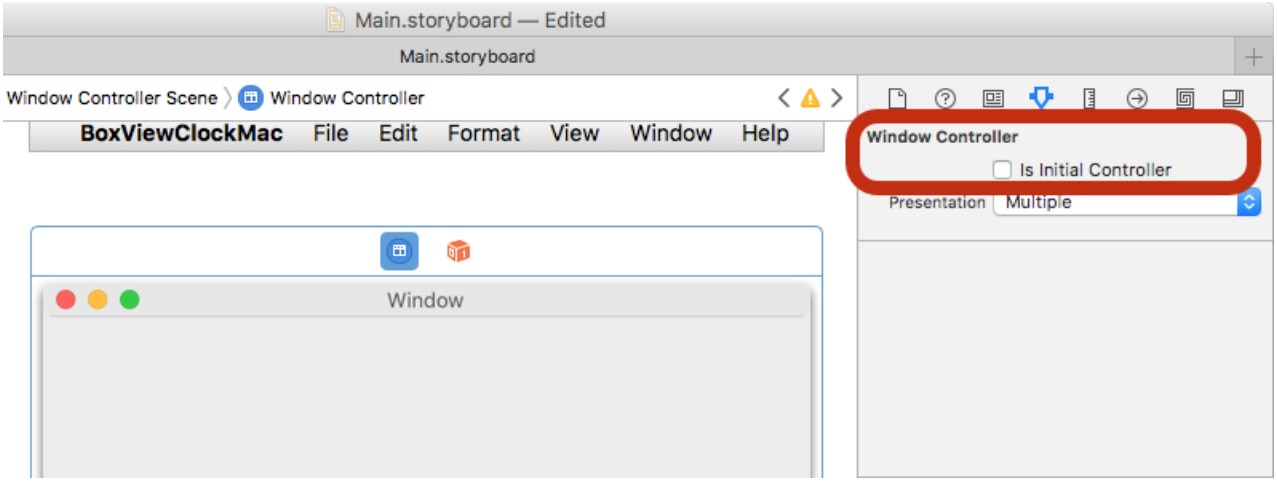
7. 更新**Main.cs**初始化 `AppDelegate`：

```
static class MainClass
{
    static void Main(string[] args)
    {
        NSApplication.Init();
        NSApplication.SharedApplication.Delegate = new AppDelegate(); // add this line
        NSApplication.Main(args);
    }
}
```

8. 更新 `AppDelegate` 若要初始化 Xamarin.Forms，创建一个窗口，并加载 Xamarin.Forms 应用程序（记住设置相应 `Title`）。*如果需要初始化其他依赖项，此处执行该操作也。*

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.MacOS;
// also add a using for the Xamarin.Forms project, if the namespace is different to this file
...
[Register("AppDelegate")]
public class AppDelegate : FormsApplicationDelegate
{
    NSWindow window;
    public AppDelegate()
    {
        var style = NSWindowStyle.Closable | NSWindowStyle.Resizable | NSWindowStyle.Titled;

        var rect = new CoreGraphics.CGRect(200, 1000, 1024, 768);
        window = new NSWindow(rect, style, NSBackingStore.Buffered, false);
        window.Title = "Xamarin.Forms on Mac!"; // choose your own Title here
        window.TitleVisibility = NSWindowTitleVisibility.Hidden;
    }

    public override NSWindow MainWindow
    {
        get { return window; }
    }

    public override void DidFinishLaunching(NSNotification notification)
    {
        Forms.Init();
        LoadApplication(new App());
        base.DidFinishLaunching(notification);
    }
}
```

9. 双击**Main.storyboard** Xcode 中进行编辑。选择**窗口并_取消选中_是初始控制器**复选框（这是因为上面的代码创建一个窗口）：



您可以编辑情节提要中删除不需要的项目中的菜单系统。

10. 最后，添加任何本地资源（例如。图像文件）从现有平台项目所需的。

11. Mac 项目现在应在 macOS 上运行 Xamarin.Forms 代码 ！

后续步骤

"样式"

最近所做的更改到 `OnPlatform` 现在可以针对任意数量的平台。这包括 macOS。

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White"/>
        <On Platform="macOS" Value="White"/>
        <On Platform="Android" Value="Black"/>
    </OnPlatform>
</Button.TextColor>
```

请注意还可能会像这样的平台上双击：`<On Platform="iOS, macOS" ...>`。

窗口大小和位置

您可以调整的初始大小和位置中的窗口 `AppDelegate`：

```
var rect = new CoreGraphics.CGRect(200, 1000, 1024, 768);  // x, y, width, height
```

已知问题

这是预览版，因此可以预见，并非所有内容都可用于生产。以下是将 macOS 添加到你的项目时可能会遇到的一些事项：

并非所有 Nuget 都有多个适用于 macOS 准备就绪

包必须面向"xamarinmac20"若要在 macOS project 中工作。您可能会发现，一些使用的库尚不支持 macOS。

在这种情况下，你将需要将请求发送到项目的维护程序以将其添加。直到他们获得支持，您可能需要寻找替代组件。

缺少的 Xamarin.Forms 功能

并非所有 Xamarin.Forms 功能都都已完成在此预览版中;下面是功能的一些尚未实现的列表：

- 页脚
- 映像-方面

- ListView – ScrollTo，UnevenRows 支持，刷新，SeparatorColor，SeparatorVisibility
- MasterDetailPage – BackgroundColor
- 导航 – InsertPageBefore
- OpenGLRenderer
- 选取器 – Bindable/可观察量实现
- TabbedPage – BarBackgroundColor，BarTextColor
- TableView – UnevenRows
- ViewCell – IsEnabled，ForceUpdateSize
- WebView – 大多数 WebNavigationEvents

相关链接

- Xamarin.Mac

Xamarin 本机项目中的 Xamarin.Forms

2018/11/13 • • Edit Online

本机窗体允许 Xamarin.Forms ContentPage 派生页可供本机 Xamarin.iOS、Xamarin.Android 和通用 Windows 平台 (UWP) 项目。本机项目都可以使用直接添加到项目中，或从.NET Standard 库、.NET Standard 库或共享项目的 ContentPage 派生页。本文介绍如何使用直接添加到本机项目的 ContentPage 派生页以及如何在它们之间导航。

通常情况下，Xamarin.Forms 应用程序包括一个或多个页派生自 `ContentPage` ，，这些页面在所有平台共享.NET Standard 库项目或共享项目中。但是，本机窗体允许 `ContentPage` -派生页来直接添加到 Xamarin.iOS、Xamarin.Android 和 UWP 的本机应用程序。相比于使用的本机项目 `ContentPage` -派生的页面从.NET Standard 库项目或共享项目中，直接向本机项目中添加页面的优点是，可以使用本机视图扩展页。然后可以使用 XAML 中名为本机视图 `x:Name` 和代码隐藏中被引用。有关本机视图的详细信息，请参阅本机视图。

使用 Xamarin.Forms 的过程 `ContentPage` -本机项目中派生的页面如下所示：

1. 对本机项目中添加的 Xamarin.Forms NuGet 包。
2. 添加 `ContentPage` -派生页上，以及任何依赖项，对本机项目。
3. 调用 `Forms.Init` 方法。
4. 构造的实例 `ContentPage` -派生页并将其转换为适当的本机类型使用以下扩展方法之一：`CreateViewController` 适用于 iOS，`CreateSupportFragment` 对于 Android，或 `CreateFrameworkElement` 为UWP。
5. 导航到的本机类型表示形式 `ContentPage` -派生页使用本机导航 API。

必须通过调用初始化 Xamarin.Forms `Forms.Init` 方法之前本机项目可以构造 `ContentPage` -派生页。选择何时执行此操作主要取决于你的应用程序流中最方便的时候 – 在应用程序启动时或之前无法执行其 `ContentPage` -构造派生的页面。在这篇文章，并随附的示例应用程序，`Forms.Init` 在应用程序启动时调用方法。

> **NOTE**
>
> **NativeForms**示例应用程序解决方案不包含任何 Xamarin.Forms 项目。相反，它包含在 Xamarin.iOS 项目、Xamarin.Android 项目，和 UWP 项目。每个项目是使用本机窗体来使用本机项目 `ContentPage` -派生页。但是，没有的理由为何无法使用本机项目 `ContentPage` -派生自.NET Standard 库项目或共享项目的页面。

在使用本机窗体，Xamarin.Forms 等功能 `DependencyService` ， `MessagingCenter` ，和数据绑定引擎，仍的所有工作。但是，必须使用本机导航 API 执行页面导航。

iOS

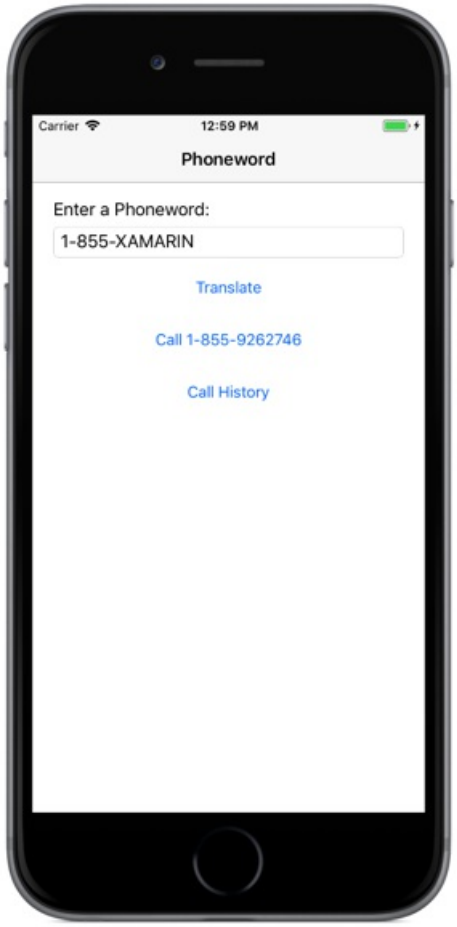
在 iOS 上，`FinishedLaunching` 重写中 `AppDelegate` 类通常是可以执行的应用程序启动相关的任务。应用程序已启动，并通常可以配置主窗口和查看控制器重写之后调用它。下面的代码示例演示 `AppDelegate` 示例应用程序中的类：

```
[Register("AppDelegate")]
public class AppDelegate : UIApplicationDelegate
{
    public static AppDelegate Instance;

    UIWindow _window;
    UINavigationController _navigation;

    public override bool FinishedLaunching(UIApplication application, NSDictionary launchOptions)
    {
        Forms.Init();

        Instance = this;
        _window = new UIWindow(UIScreen.MainScreen.Bounds);

        UINavigationBar.Appearance.SetTitleTextAttributes(new UITextAttributes
        {
            TextColor = UIColor.Black
        });

        var mainPage = new PhonewordPage().CreateViewController();
        mainPage.Title = "Phoneword";

        _navigation = new UINavigationController(mainPage);
        _window.RootViewController = _navigation;
        _window.MakeKeyAndVisible();

        return true;
    }
    ...
}
```

`FinishedLaunching` 方法执行以下任务：

- 通过调用初始化 Xamarin.Forms `Forms.Init` 方法。
- 对引用 `AppDelegate` 类存储在 `static``Instance` 字段。这是为了提供其他类来调用方法中定义的一种机制 `AppDelegate` 类。
- `UIWindow`，这本机 iOS 应用程序中的视图的主容器创建。
- `PhonewordPage` 类，该类是 Xamarin.Forms `ContentPage` -派生在 XAML 中定义的页中，会构造并转换为 `UIViewController` 使用 `CreateViewController` 扩展方法。
- `Title` 的属性 `UIViewController` 设置，这将显示在 `UINavigationBar` 。
- 一个 `UINavigationController` 创建用于管理分层导航。 `UINavigationController` 类管理一摞视图控制器，并 `UIViewController` 传递到构造函数将显示最初时 `UINavigationController` 加载。
- `UINavigationController` 实例设置为顶级 `UIViewController` 有关 `UIWindow` ，和 `UIWindow` 被设置为密钥窗口中的应用程序，变为可见。

一次 `FinishedLaunching` 方法执行，在 Xamarin.Forms 中定义 UI `PhonewordPage` 将显示类，如以下屏幕截图中所示：



交互用户界面，例如通过点击 `Button`，将导致在事件处理程序 `PhonewordPage` 隐藏代码执行。例如，当用户点击**呼叫历史记录**执行按钮时，以下事件处理程序：

```
void OnCallHistory(object sender, EventArgs e)
{
    AppDelegate.Instance.NavigateToCallHistoryPage();
}
```

`static` `AppDelegate.Instance` 字段允许 `AppDelegate.NavigateToCallHistoryPage` 方法被调用，下面的代码示例中所示：

```
public void NavigateToCallHistoryPage()
{
    var callHistoryPage = new CallHistoryPage().CreateViewController();
    callHistoryPage.Title = "Call History";
    _navigation.PushViewController(callHistoryPage, true);
}
```

`NavigateToCallHistoryPage` 方法将为 Xamarin.Forms `ContentPage` -派生页 `UIViewController` 与 `CreateViewController` 扩展方法，并设置 `Title` 属性 `UIViewController`。`UIViewController` 然后推送到 `UINavigationController` 通过 `PushViewController` 方法。因此，在 Xamarin.Forms 中定义 UI `CallHistoryPage` 将显示类，如以下屏幕截图中所示：

时 `CallHistoryPage` 点击后的显示箭头会弹出 `UIViewController` 有关 `CallHistoryPage` 类派生 `UINavigationController`，返回到用户 `UIViewController` 为 `PhonewordPage` 类。

Android

在 Android 上，`OnCreate` 重写中 `MainActivity` 类通常是可以执行的应用程序启动相关的任务。下面的代码示例演示 `MainActivity` 示例应用程序中的类：

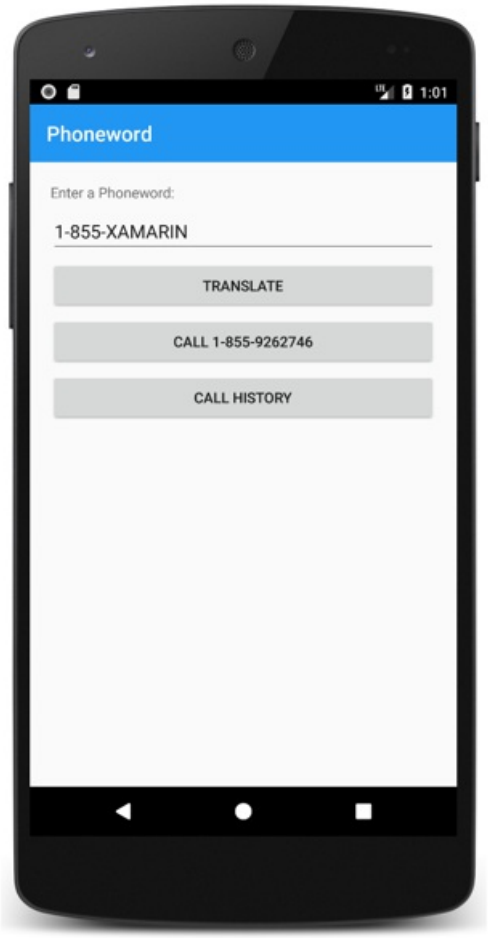
```
public class MainActivity : AppCompatActivity
{
    public static MainActivity Instance;

    protected override void OnCreate(Bundle bundle)
    {
        base.OnCreate(bundle);

        Forms.Init(this, bundle);
        Instance = this;

        SetContentView(Resource.Layout.Main);
        var toolbar = FindViewById<Toolbar>(Resource.Id.toolbar);
        SetSupportActionBar(toolbar);
        SupportActionBar.Title = "Phoneword";

        var mainPage = new PhonewordPage().CreateSupportFragment(this);
        SupportFragmentManager
            .BeginTransaction()
            .Replace(Resource.Id.fragment_frame_layout, mainPage)
            .Commit();
        ...
    }
    ...
}
```

`OnCreate` 方法执行以下任务：

- 通过调用初始化 Xamarin.Forms `Forms.Init` 方法。
- 对引用 `MainActivity` 类存储在 `static``Instance` 字段。这是为了提供其他类来调用方法中定义的一种机制 `MainActivity` 类。
- `Activity` 内容设置从布局资源。在示例应用程序，布局组成 `LinearLayout`，其中包含 `Toolbar`，和一个 `FrameLayout` 充当片段容器。
- `Toolbar` 检索并设置为在操作栏 `Activity`，并设置操作栏标题。
- `PhonewordPage` 类，该类是 Xamarin.Forms `ContentPage` -派生在 XAML 中定义的页中，会构造并转换为 `Fragment` 使用 `CreateSupportFragment` 扩展方法。
- `SupportFragmentManager` 类创建并提交事务，用于替换 `FrameLayout` 实例与 `Fragment` 为 `PhonewordPage` 类。

片段有关的详细信息，请参阅片段。

一次 `OnCreate` 方法执行，在 Xamarin.Forms 中定义 UI `PhonewordPage` 将显示类，如以下屏幕截图中所示：



交互用户界面，例如通过点击 `Button`，将导致在事件处理程序 `PhonewordPage` 隐藏代码执行。例如，当用户点击**呼叫历史记录**执行按钮时，以下事件处理程序：

```
void OnCallHistory(object sender, EventArgs e)
{
    MainActivity.Instance.NavigateToCallHistoryPage();
}
```

`static` `MainActivity.Instance` 字段允许 `MainActivity.NavigateToCallHistoryPage` 方法被调用，下面的代码示例中所示：

```
public void NavigateToCallHistoryPage()
{
    var callHistoryPage = new CallHistoryPage().CreateSupportFragment(this);
    SupportFragmentManager
        .BeginTransaction()
        .AddToBackStack(null)
        .Replace(Resource.Id.fragment_frame_layout, callHistoryPage)
        .Commit();
}
```

`NavigateToCallHistoryPage` 方法将为 Xamarin.Forms `ContentPage` -派生页 `Fragment` 与 `CreateSupportFragment` 扩展方法，并将添加 `Fragment` 片段到后退堆栈。因此，在 Xamarin.Forms 中定义 UI `CallHistoryPage` 将显示，如以下屏幕截图中所示：

时 `CallHistoryPage` 点击后的显示箭头会弹出 `Fragment` 有关 `CallHistoryPage` 片段 back 堆栈中，从已注册到用户 `Fragment` 为 `PhonewordPage` 类。

启用后退导航的支持

`SupportFragmentManager` 类具有 `BackStackChanged` 片段 back 堆栈的内容发生更改时激发的事件。 `OnCreate` 中的方法 `MainActivity` 类包含此事件的匿名事件处理程序：

```
SupportFragmentManager.BackStackChanged += (sender, e) =>
{
    bool hasBack = SupportFragmentManager.BackStackEntryCount > 0;
    SupportActionBar.SetHomeButtonEnabled(hasBack);
    SupportActionBar.SetDisplayHomeAsUpEnabled(hasBack);
    SupportActionBar.Title = hasBack ? "Call History" : "Phoneword";
};
```

此事件处理程序在操作栏上显示后退按钮，前提是没有一个或多个 `Fragment` 实例在片段上的后退堆栈。点击后退按钮的响应由 `OnOptionsItemSelected` 重写：

```
public override bool OnOptionsItemSelected(Android.Views.IMenuItem item)
{
    if (item.ItemId == global::Android.Resource.Id.Home && SupportFragmentManager.BackStackEntryCount > 0)
    {
        SupportFragmentManager.PopBackStack();
        return true;
    }
    return base.OnOptionsItemSelected(item);
}
```

`OnOptionsItemSelected` 替代选择选项菜单中的项时调用。此实现中弹出的当前片段中片段 back 堆栈，前提是已选择后退按钮，并且有一个或多个 `Fragment` 实例在片段上的后退堆栈。

多个活动

当应用程序组成的多个活动 `ContentPage` -派生的页可以嵌入到每个活动。在此方案中，`Forms.Init` 仅在需要调用方法 `OnCreate` 的第一个重写 `Activity` 嵌入 Xamarin.Forms `ContentPage` 。但是，这会产生以下影响：

- 值 `Xamarin.Forms.Color.Accent` 来自 `Activity` 调用 `Forms.Init` 方法。
- 值 `Xamarin.Forms.Application.Current` 将与相关联 `Activity` 调用 `Forms.Init` 方法。

选择一个文件

嵌入内容时 `ContentPage` -派生使用页面 `WebView` ，需要支持的 HTML"选择文件"按钮，则 `Activity` 将需要重写 `OnActivityResult` 方法：

```
protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
{
    base.OnActivityResult(requestCode, resultCode, data);
    ActivityResultCallbackRegistry.InvokeCallback(requestCode, resultCode, data);
}
```

UWP

UWP，本机上 `App` 类通常是可以执行的应用程序启动相关的任务。Xamarin.Forms 通常在初始化时，在 Xamarin.Forms UWP 应用程序中 `OnLaunched` 重写中本机 `App` 类中，以传递 `LaunchActivatedEventArgs` 参数 `Forms.Init` 方法。出于此原因，本机 UWP 应用程序中使用 Xamarin.Forms `ContentPage` -派生的页面可以最轻松地调用 `Forms.Init` 方法从 `App.OnLaunched` 方法。

默认情况下，本机 `App` 类启动 `MainPage` 的第一页作为应用程序的类。下面的代码示例演示 `MainPage` 示例应用程序中的类：

```
public sealed partial class MainPage : Page
{
    public static MainPage Instance;

    public MainPage()
    {
        this.InitializeComponent();
        this.NavigationCacheMode = NavigationCacheMode.Enabled;
        Instance = this;
        this.Content = new Phoneword.UWP.Views.PhonewordPage().CreateFrameworkElement();
    }
    ...
}
```

`MainPage` 构造函数将执行以下任务：

- 为页上，启用缓存，以便新 `MainPage` 当用户导航回页不构造。
- 对引用 `MainPage` 类存储在 `static``Instance` 字段。这是为了提供其他类来调用方法中定义的一种机制 `MainPage` 类。
- `PhonewordPage` 类，该类是 Xamarin.Forms `ContentPage` -派生在 XAML 中定义的页中，会构造并转换为 `FrameworkElement` 使用 `CreateFrameworkElement` 扩展方法，然后将设置为的内容 `MainPage` 类。

一次 `MainPage` 构造函数已执行，在 Xamarin.Forms 中定义 UI `PhonewordPage` 将显示类，如以下屏幕截图中所示：

交互用户界面，例如通过点击 `Button`，将导致在事件处理程序 `PhonewordPage` 隐藏代码执行。例如，当用户点击**呼叫历史记录**执行按钮时，以下事件处理程序：

```
void OnCallHistory(object sender, EventArgs e)
{
    Phoneword.UWP.MainPage.Instance.NavigateToCallHistoryPage();
}
```

`static` `MainPage.Instance` 字段允许 `MainPage.NavigateToCallHistoryPage` 方法被调用，下面的代码示例中所示：

```
public void NavigateToCallHistoryPage()
{
    this.Frame.Navigate(new CallHistoryPage());
}
```

通常使用执行 UWP 中的导航 `Frame.Navigate` 方法，它使用 `Page` 参数。Xamarin.Forms 定义了 `Frame.Navigate` 扩展方法采用 `ContentPage` -派生页实例。因此，当 `NavigateToCallHistoryPage` 执行方法时，在 Xamarin.Forms 中定义的 UI `CallHistoryPage` 将显示，如以下屏幕截图中所示：

时 `CallHistoryPage` 点击后的显示箭头会弹出 `FrameworkElement` 有关 `CallHistoryPage` 从应用程序内 back 堆栈中，返回到用户 `FrameworkElement` 为 `PhonewordPage` 类。

启用后退导航的支持

在 UWP 中，应用程序必须启用后退导航的所有硬件和软件后退按钮，在不同设备外观造型上。这可以通过注册的事件处理程序来实现 `BackRequested` 事件，可以在中执行 `OnLaunched` 方法在本机 `App` 类：

```
protected override void OnLaunched(LaunchActivatedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;

    if (rootFrame == null)
    {
        ...
        // Place the frame in the current Window
        Window.Current.Content = rootFrame;

        SystemNavigationManager.GetForCurrentView().BackRequested += OnBackRequested;
    }
    ...
}
```

当启动应用程序时， `GetForCurrentView` 方法检索 `SystemNavigationManager` 对象与当前的视图，然后注册的事件处理程序 `BackRequested` 事件。应用程序仅接收此事件，如果它是前台应用程序，并在响应中，调用 `OnBackRequested` 事件处理程序：

```
void OnBackRequested(object sender, BackRequestedEventArgs e)
{
    Frame rootFrame = Window.Current.Content as Frame;
    if (rootFrame.CanGoBack)
    {
        e.Handled = true;
        rootFrame.GoBack();
    }
}
```

`OnBackRequested`事件处理程序调用`GoBack`方法的应用程序和设置在根框架`BackRequestedEventArgs.Handled`属性设置为`true`将该事件标记为已处理。若要将事件标记为已处理的失败可能导致系统离开（上的移动设备系列）的应用程序或忽略（上的桌面设备系列）的事件。

应用程序依赖于在手机上，一个提供系统后退按钮，但可以选择是否要在桌面设备上的标题栏上显示后退按钮。这通过设置来实现`AppViewBackButtonVisibility`属性设置为其中一个`AppViewBackButtonVisibility`枚举值：

```
void OnNavigated(object sender, NavigationEventArgs e)
{
    SystemNavigationManager.GetForCurrentView().AppViewBackButtonVisibility =
        ((Frame)sender).CanGoBack ? AppViewBackButtonVisibility.Visible :
AppViewBackButtonVisibility.Collapsed;
}
```

`OnNavigated`事件处理程序，响应执行`Navigated`页面导航发生时，事件的激发，更新的标题栏中后退按钮的可见性。这可确保，标题栏后退按钮是可见的如果应用程序内 back 堆栈不为空，或从中删除的标题栏中应用程序的 back 堆栈是否为空。

UWP 上后退导航的支持的详细信息，请参阅导航历史记录和向后导航适用于 UWP 应用。

总结

本机窗体允许 Xamarin.Forms `ContentPage` -派生页可供本机 Xamarin.iOS、Xamarin.Android 和通用 Windows 平台 (UWP) 项目。可以使用本机项目`ContentPage`-派生直接添加到项目中，或从.NET Standard 库项目或共享项目的页面。本文介绍了如何使用`ContentPage`-派生直接添加到本机项目，以及如何在它们之间导航的页面。

相关链接

- NativeForms（示例）
- 本机视图

Xamarin.Forms 中的本机视图

2018/6/9 • • Edit Online

从 iOS、Android 和通用 Windows 平台 (UWP) 的本机视图可以从 Xamarin.Forms 直接引用。可以在本机视图中设置属性和事件处理程序，并且它们可以与 Xamarin.Forms 视图交互。

采用 XAML 的本机视图

从 iOS、Android 和 UWP 本机视图可以从创建使用 XAML 的 Xamarin.Forms 页面直接引用。

采用 C# 的本机视图

从 iOS、Android 和 UWP 本机视图可以从使用 C# 创建 Xamarin.Forms 页面直接引用。

相关链接

- 本机窗体

在 XAML 中的本机视图

2018/7/13 • • Edit Online

从 iOS、Android 和通用 Windows 平台的本机视图可以直接引用 Xamarin.Forms XAML 文件中。上的本机视图，可以设置属性和事件处理程序，它们可以与 Xamarin.Forms 视图进行交互。本文演示如何使用 Xamarin.Forms XAML 文件中的本机视图。

本文讨论以下主题：

- 使用本机视图– 使用 XAML 中的本机视图的过程。
- 使用本机绑定– 数据绑定到和从本机视图的属性。
- 将参数传递给本机视图– 将实参传递给本机视图构造函数，并调用工厂方法的本机视图。
- 从代码中引用的本机视图– 正在检索本机视图实例声明在 XAML 文件中，从其代码隐藏文件。
- 子类化的本机视图– 子类化来定义 XAML 友好 API 的本机视图。

概述

若要嵌入到 Xamarin.Forms XAML 文件的本机视图：

1. 添加 `xmlns` 包含本机视图的命名空间的 XAML 文件中的命名空间声明。
2. 在 XAML 文件中创建本机视图的实例。

> **NOTE**
> XAMLC 必须关闭状态的所有 XAML 页，使用本机视图。

若要从代码隐藏文件引用本机视图，必须使用共享资产项目 (SAP)，并使用条件编译指令将特定于平台的代码包装。有关详细信息请参阅从代码中引用的本机视图。

使用本机视图

下面的代码示例演示如何使用 Xamarin.Forms 到每个平台的本机视图 `ContentPage`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
        xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
        xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
            Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        x:Class="NativeViews.NativeViewDemo">
    <StackLayout Margin="20">
        <ios:UILabel Text="Hello World" TextColor="{x:Static ios:UIColor.Red}" View.HorizontalOptions="Start"
/>
        <androidWidget:TextView Text="Hello World" x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
/>
        <win:TextBlock Text="Hello World" />
    </StackLayout>
</ContentPage>
```

以及指定 `clr-namespace` 并 `assembly` 本机视图命名空间，`targetPlatform` 还必须指定。此属性应设置为的值之一

`TargetPlatform` 枚举，并通常将设置为 `iOS`， `Android`，或 `Windows`。在运行时，XAML 分析器将忽略具有任何 XML 命名空间前缀 `targetPlatform` 这不匹配其运行应用程序的平台。

每个命名空间声明可以用于引用来自指定命名空间的任何类或结构。例如，`ios` 命名空间声明可用于引用在 iOS 中的任何类或结构 `UIKit` 命名空间。可以通过 XAML，设置本机视图的属性，但属性和对象类型必须匹配。例如，`UILabel.TextColor` 属性设置为 `UIColor.Red` 使用 `x:Static` 标记扩展和 `ios` 命名空间。

可绑定属性和附加可绑定属性也可以设置上的本机视图通过使用 `Class.BindableProperty="value"` 语法。每个本机视图包装在平台特定 `NativeViewWrapper` 实例，派生自 `Xamarin.Forms.View` 类。本机视图上设置可绑定属性或附加可绑定属性传输到包装器的属性值。例如，可以通过设置指定居中的水平布局 `View.HorizontalOptions="Center"` 本机视图上。

> **NOTE**
> 请注意样式不能用于本机视图，因为样式仅可以由支持的属性为目标 `BindableProperty` 对象。

Android 小组件的构造函数通常需要 Android `Context` 对象作为参数，并且此便可提供通过中的静态属性 `MainActivity` 类。因此，当在 XAML 中，创建 Android 小组件 `Context` 通常必须将对象传递给小组件的构造函数使用 `x:Arguments` 属性与 `x:Static` 标记扩展。有关详细信息，请参阅将参数传递到本机视图。

> **NOTE**
> 请注意命名与本机视图 `x:Name` 不能在.NET Standard 库项目或共享资产项目 (SAP)。执行此操作将生成的变量的本机类型，这将导致编译错误。但是，可以在包装的本机视图 `ContentView` 实例并检索在代码隐藏文件中，前提使用 SAP。有关详细信息，请参阅从代码引用本机视图。

本机绑定

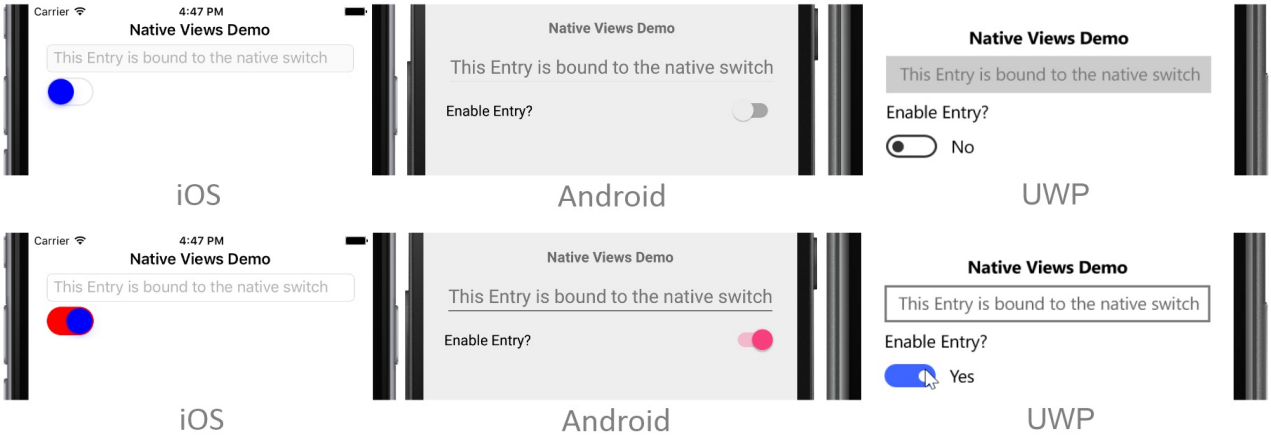
数据绑定用来同步其数据源，使用 UI，并简化 Xamarin.Forms 应用程序显示的方式，并使用其数据进行交互。提供的源对象实现 `INotifyPropertyChanged` 接口中的更改*源*对象会自动推送到*目标*对象的绑定框架和中的更改*目标*对象可以选择性地推送到*源*对象。

本机视图的属性还可以使用数据绑定。下面的代码示例演示如何使用本机视图的属性的数据绑定：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
        xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
        xmlns:win="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
            Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:local="clr-namespace:NativeSwitch"
        x:Class="NativeSwitch.NativeSwitchPage">
    <StackLayout Margin="20">
        <Label Text="Native Views Demo" FontAttributes="Bold" HorizontalOptions="Center" />
        <Entry Placeholder="This Entry is bound to the native switch" IsEnabled="{Binding IsSwitchOn}" />
        <ios:UISwitch On="{Binding Path=IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=ValueChanged}"
            OnTintColor="{x:Static ios:UIColor.Red}"
            ThumbTintColor="{x:Static ios:UIColor.Blue}" />
        <androidWidget:Switch x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
            Checked="{Binding Path=IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=CheckedChange}"
            Text="Enable Entry?" />
        <win:ToggleSwitch Header="Enable Entry?"
            OffContent="No"
            OnContent="Yes"
            IsOn="{Binding IsSwitchOn, Mode=TwoWay, UpdateSourceEventName=Toggled}" />
    </StackLayout>
</ContentPage>
```

该页面包含 `Entry` 其 `IsEnabled` 属性绑定到 `NativeSwitchPageViewModel.IsSwitchOn` 属性。`BindingContext` 页的设置为的新实例 `NativeSwitchPageViewModel` 类在代码隐藏文件中，与 ViewModel 类实现 `INotifyPropertyChanged` 接口。

此页还包含用于每个平台的本机交换机。使用每个本机交换机 `TwoWay` 要更新的值绑定 `NativeSwitchPageViewModel.IsSwitchOn` 属性。因此，此开关处于关闭状态，`Entry` 处于禁用状态，以及何时开关为开，`Entry` 已启用。下面的屏幕截图显示了每个平台上的此功能：



自动支持双向绑定，前提是本机属性实现 `INotifyPropertyChanged`，在 iOS 上，支持键值对观察 (KVO)，或者是 `DependencyProperty` UWP 上。但是，许多本机视图不支持属性更改通知。对于这些视图中，您可以指定 `UpdateSourceEventName` 用作绑定表达式的一部分的属性值。此属性应设置为目标属性已更改时发出信号的本机视图中的事件的名称。然后，本机开关的值发生更改时，`Binding` 类会通知用户已更改开关值和 `NativeSwitchPageViewModel.IsSwitchOn` 更新属性值。

传递自变量到本机视图

可以将构造函数自变量传递给使用的本机视图 `x:Arguments` 属性与 `x:Static` 标记扩展。此外，本机视图工厂方法 (`public static` 返回对象或值类或结构，它定义的方法的类型相同的方法) 可以通过指定方法的调用名称使用 `x:FactoryMethod` 特性，并且其自变量使用 `x:Arguments` 属性。

下面的代码示例演示了这两种技术：

```
<ContentPage ...
        xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
        xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidGraphics="clr-namespace:Android.Graphics;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
        xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:winMedia="clr-namespace:Windows.UI.Xaml.Media;assembly=Windows, Version=255.255.255.255,
Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:winText="clr-namespace:Windows.UI.Text;assembly=Windows, Version=255.255.255.255,
Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:winui="clr-namespace:Windows.UI;assembly=Windows, Version=255.255.255.255, Culture=neutral,
PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows">
        ...
        <ios:UILabel Text="Simple Native Color Picker" View.HorizontalOptions="Center">
            <ios:UILabel.Font>
                <ios:UIFont x:FactoryMethod="FromName">
                    <x:Arguments>
                        <x:String>Papyrus</x:String>
                        <x:Single>24</x:Single>
                    </x:Arguments>
                </ios:UIFont>
            </ios:UILabel.Font>
        </ios:UILabel>
        <androidWidget:TextView x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
                    Text="Simple Native Color Picker"
                    TextSize="24"
                    View.HorizontalOptions="Center">
            <androidWidget:TextView.Typeface>
                <androidGraphics:Typeface x:FactoryMethod="Create">
                    <x:Arguments>
                        <x:String>cursive</x:String>
                        <androidGraphics:TypefaceStyle>Normal</androidGraphics:TypefaceStyle>
                    </x:Arguments>
                </androidGraphics:Typeface>
            </androidWidget:TextView.Typeface>
        </androidWidget:TextView>
        <winControls:TextBlock Text="Simple Native Color Picker"
                    FontSize="20"
                    FontStyle="{x:Static winText:FontStyle.Italic}"
                    View.HorizontalOptions="Center">
            <winControls:TextBlock.FontFamily>
                <winMedia:FontFamily>
                    <x:Arguments>
                        <x:String>Georgia</x:String>
                    </x:Arguments>
                </winMedia:FontFamily>
            </winControls:TextBlock.FontFamily>
        </winControls:TextBlock>
        ...
</ContentPage>
```

`UIFont.FromName` 工厂方法用于设置 `UILabel.Font` 属性设置为一个新 `UIFont` 在 iOS 上。`UIFont` 子级的方法参数通过指定名称和大小 `x:Arguments` 属性。

`Typeface.Create` 工厂方法用于设置 `TextView.Typeface` 属性设置为一个新 `Typeface` 在 Android 上。`Typeface` 子级的方法参数通过指定系列名称和样式 `x:Arguments` 属性。

`FontFamily` 构造函数用于设置 `TextBlock.FontFamily` 属性设置为一个新 `FontFamily` 通用 Windows 平台 (UWP)。`FontFamily` 的子级的方法参数通过指定名称 `x:Arguments` 属性。

> **NOTE**
> 参数必须与构造函数或工厂方法所需的类型匹配。

以下屏幕截图显示指定工厂方法和构造函数参数，若要将字体设置不同的本机视图上的结果：

iOS　　Android　　UWP

有关在 XAML 中传递自变量的详细信息，请参阅在 XAML 中传递参数。

从代码中引用的本机视图

尽管不能使用本机视图命名为 `x:Name` 属性，它是可以检索在共享访问项目中，其代码隐藏文件中的 XAML 文件中声明的本机视图实例，前提是本机视图是子级 `ContentView`，它指定 `x:Name` 属性值。然后，在代码隐藏文件中的条件编译指令内您应该：

1. 检索 `ContentView.Content` 属性值，并将其转换为特定于平台的 `NativeViewWrapper` 类型。
2. 检索 `NativeViewWrapper.NativeElement` 属性并将其转换为本机视图类型。

然后执行所需的操作的本机视图调用本机 API。此方法还提供了好处，针对不同平台的多个 XAML 本机视图可以是相同的子级 `ContentView`。下面的代码示例演示了此种方法：

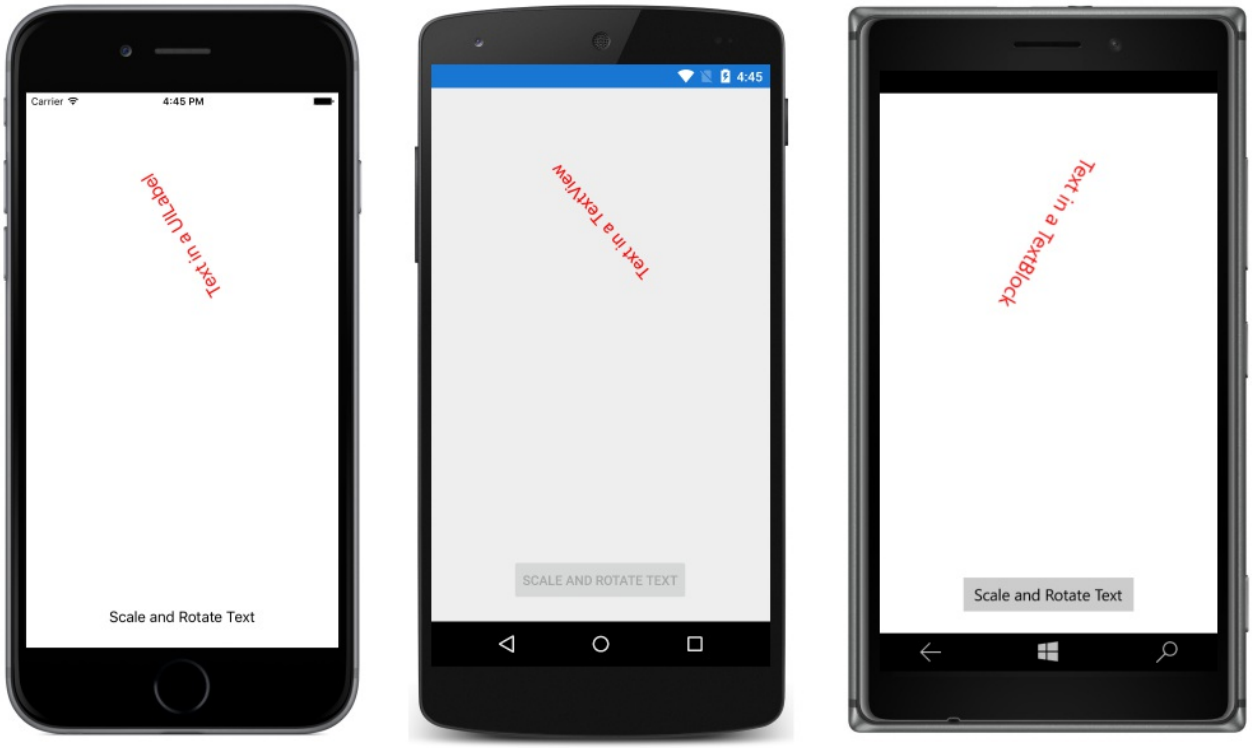
```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
        xmlns:androidWidget="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
        xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
            Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:local="clr-namespace:NativeViewInsideContentView"
        x:Class="NativeViewInsideContentView.NativeViewInsideContentViewPage">
    <StackLayout Margin="20">
        <ContentView x:Name="contentViewTextParent" HorizontalOptions="Center"
VerticalOptions="CenterAndExpand">
            <ios:UILabel Text="Text in a UILabel" TextColor="{x:Static ios:UIColor.Red}" />
            <androidWidget:TextView x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
                Text="Text in a TextView" />
              <winControls:TextBlock Text="Text in a TextBlock" />
        </ContentView>
        <ContentView x:Name="contentViewButtonParent" HorizontalOptions="Center"
VerticalOptions="EndAndExpand">
            <ios:UIButton TouchUpInside="OnButtonTap" View.HorizontalOptions="Center"
View.VerticalOptions="Center" />
            <androidWidget:Button x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
                Text="Scale and Rotate Text"
                Click="OnButtonTap" />
            <winControls:Button Content="Scale and Rotate Text" />
        </ContentView>
    </StackLayout>
</ContentPage>
```

在上面的示例中，为每个平台的本机视图是子级 `ContentView` 控件，具有 `x:Name` 属性值用于检索 `ContentView` 代码隐藏中：

```
public partial class NativeViewInsideContentViewPage : ContentPage
{
    public NativeViewInsideContentViewPage()
    {
        InitializeComponent();

#if __IOS__
        var wrapper = (Xamarin.Forms.Platform.iOS.NativeViewWrapper)contentViewButtonParent.Content;
        var button = (UIKit.UIButton)wrapper.NativeView;
        button.SetTitle("Scale and Rotate Text", UIKit.UIControlState.Normal);
        button.SetTitleColor(UIKit.UIColor.Black, UIKit.UIControlState.Normal);
#endif
#if __ANDROID__
        var wrapper = (Xamarin.Forms.Platform.Android.NativeViewWrapper)contentViewTextParent.Content;
        var textView = (Android.Widget.TextView)wrapper.NativeView;
        textView.SetTextColor(Android.Graphics.Color.Red);
#endif
#if WINDOWS_UWP
        var textWrapper = (Xamarin.Forms.Platform.UWP.NativeViewWrapper)contentViewTextParent.Content;
        var textBlock = (Windows.UI.Xaml.Controls.TextBlock)textWrapper.NativeElement;
        textBlock.Foreground = new Windows.UI.Xaml.Media.SolidColorBrush(Windows.UI.Colors.Red);
        var buttonWrapper = (Xamarin.Forms.Platform.UWP.NativeViewWrapper)contentViewButtonParent.Content;
        var button = (Windows.UI.Xaml.Controls.Button)buttonWrapper.NativeElement;
        button.Click += (sender, args) => OnButtonTap(sender, EventArgs.Empty);
#endif
    }

    async void OnButtonTap(object sender, EventArgs e)
    {
        contentViewButtonParent.Content.IsEnabled = false;
        contentViewTextParent.Content.ScaleTo(2, 2000);
        await contentViewTextParent.Content.RotateTo(360, 2000);
        contentViewTextParent.Content.ScaleTo(1, 2000);
        await contentViewTextParent.Content.RelRotateTo(360, 2000);
        contentViewButtonParent.Content.IsEnabled = true;
    }
}
```

`ContentView.Content` 访问属性来检索为特定于平台的已包装的本机视图 `NativeViewWrapper` 实例。 `NativeViewWrapper.NativeElement` 然后访问属性来检索作为其本机类型的本机视图。然后调用本机视图 API 来执行所需的操作。

IOS 和 Android 的本机按钮共用同一个 `OnButtonTap` 事件处理程序，因为每个本机按钮使用 `EventHandler` 响应触摸事件委托。但是，通用 Windows 平台 (UWP) 均使用单独 `RoutedEventHandler`，这反过来将会占用 `OnButtonTap` 在此示例中的事件处理程序。因此，单击本机按钮时，`OnButtonTap` 事件处理程序执行，该缩放和旋转内包含的本机控件 `ContentView` 名为 `contentViewTextParent`。下面的屏幕截图演示了此每个平台上发生：



子类化本机视图

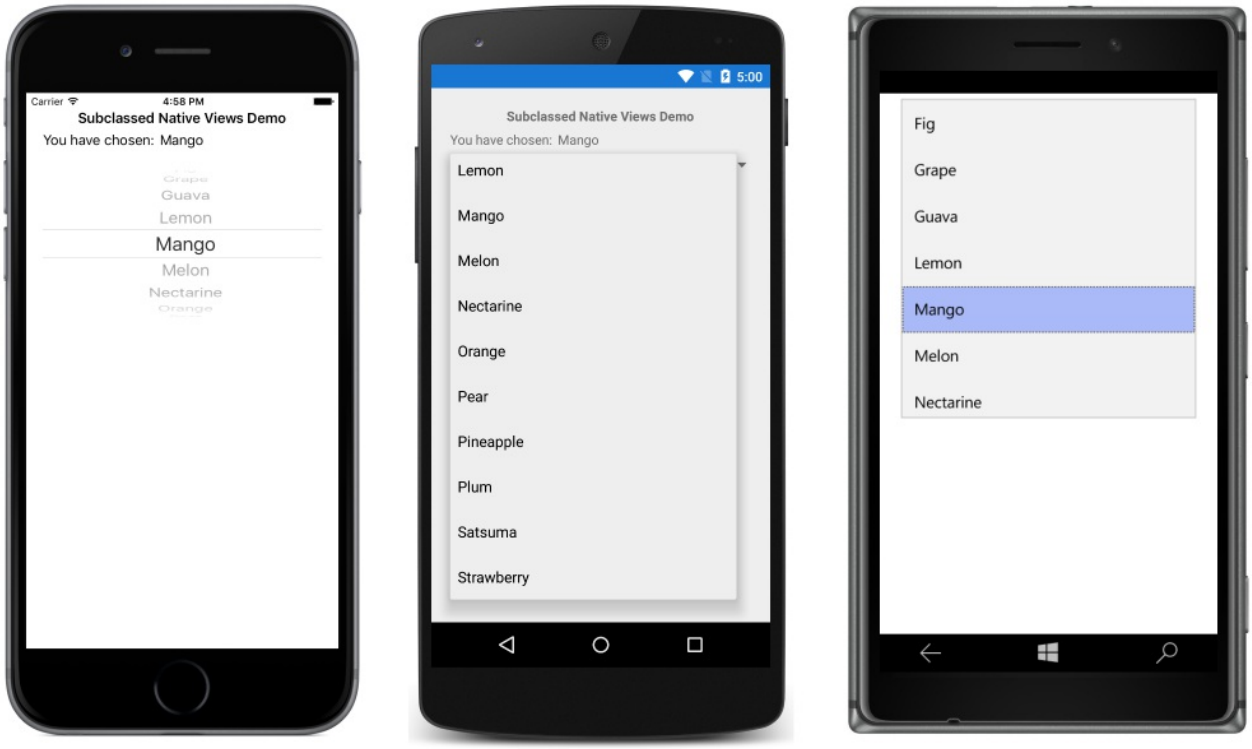
许多 iOS 和 Android 的本机视图并不适合因为它们使用的方法，而不是属性，若要设置该控件在 XAML 中实例化。此问题的解决方案是为子类中定义多个 XAML 友好 API，使用属性来设置该控件，并使用独立于平台的事件的包装器的本机视图。已包装的本机视图可放置在共享资产项目 (SAP) 和使用条件编译指令括住或置于特定于平台的项目中和从 XAML 引用.NET Standard 库项目中。

下面的代码示例演示了使用 Xamarin.Forms 页面创建子类的本机视图：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
        xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
        xmlns:ios="clr-namespace:UIKit;assembly=Xamarin.iOS;targetPlatform=iOS"
        xmlns:iosLocal="clr-
namespace:SubclassedNativeControls.iOS;assembly=SubclassedNativeControls.iOS;targetPlatform=iOS"
        xmlns:android="clr-namespace:Android.Widget;assembly=Mono.Android;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SimpleColorPicker.Droid;assembly=SimpleColorPicker.Droid;targetPlatform=Android"
        xmlns:androidLocal="clr-
namespace:SubclassedNativeControls.Droid;assembly=SubclassedNativeControls.Droid;targetPlatform=Android"
        xmlns:winControls="clr-namespace:Windows.UI.Xaml.Controls;assembly=Windows, Version=255.255.255.255,
            Culture=neutral, PublicKeyToken=null, ContentType=WindowsRuntime;targetPlatform=Windows"
        xmlns:local="clr-namespace:SubclassedNativeControls"
        x:Class="SubclassedNativeControls.SubclassedNativeControlsPage">
    <StackLayout Margin="20">
        <Label Text="Subclassed Native Views Demo" FontAttributes="Bold" HorizontalOptions="Center" />
        <StackLayout Orientation="Horizontal">
          <Label Text="You have chosen:" />
          <Label Text="{Binding SelectedFruit}" />
        </StackLayout>
        <iosLocal:MyUIPickerView ItemsSource="{Binding Fruits}"
            SelectedItem="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=SelectedItemChanged}" />
        <androidLocal:MySpinner x:Arguments="{x:Static androidLocal:MainActivity.Instance}"
            ItemsSource="{Binding Fruits}"
            SelectedObject="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=ItemSelected}" />
        <winControls:ComboBox ItemsSource="{Binding Fruits}"
            SelectedItem="{Binding SelectedFruit, Mode=TwoWay, UpdateSourceEventName=SelectionChanged}" />
    </StackLayout>
</ContentPage>
```

该页面包含 `Label` 显示用户选择的本机控件从水果。`Label` 绑定到 `SubclassedNativeControlsPageViewModel.SelectedFruit` 属性。`BindingContext` 页的设置为的新实例 `SubclassedNativeControlsPageViewModel` 类在代码隐藏文件中，与 ViewModel 类实现 `INotifyPropertyChanged` 接口。

该页还包含每个平台的本机选取器视图。每个本机视图显示水果的集合的绑定及其 `ItemSource` 属性设置为 `SubclassedNativeControlsPageViewModel.Fruits` 集合。这允许用户选取水果，如以下屏幕截图中所示：



IOS 和 Android 上本机选取器使用方法设置这些控件。因此，这些选取器必须子类化来公开属性，以使它们适合于 XAML 应用。在通用 Windows 平台 (UWP)，`ComboBox` 已 XAML 友好，因此，不需要子类化。

iOS

IOS 实现子类 `UIPickerView` 视图中，并公开属性和从 XAML 可轻松使用的事件：

```
public class MyUIPickerView : UIPickerView
{
    public event EventHandler<EventArgs> SelectedItemChanged;

    public MyUIPickerView()
    {
        var model = new PickerModel();
        model.ItemChanged += (sender, e) =>
        {
            if (SelectedItemChanged != null)
            {
                SelectedItemChanged.Invoke(this, e);
            }
        };
        Model = model;
    }

    public IList<string> ItemsSource
    {
        get
        {
            var pickerModel = Model as PickerModel;
            return (pickerModel != null) ? pickerModel.Items : null;
        }
        set
        {
            var model = Model as PickerModel;
            if (model != null)
            {
                model.Items = value;
            }
        }
    }

    public string SelectedItem
    {
        get { return (Model as PickerModel).SelectedItem; }
        set { }
    }
}
```

`MyUIPickerView`类公开`ItemsSource`并`SelectedItem`属性，和一个`SelectedItemChanged`事件。一个`UIPickerView`要求基础`UIPickerViewModel`数据模型中，将访问`MyUIPickerView`属性和事件。`UIPickerViewModel`数据模型提供的`PickerModel`类：

```
class PickerModel : UIPickerViewModel
{
    int selectedIndex = 0;
    public event EventHandler<EventArgs> ItemChanged;
    public IList<string> Items { get; set; }

    public string SelectedItem
    {
        get
        {
            return Items != null && selectedIndex >= 0 && selectedIndex < Items.Count ? Items[selectedIndex] :
null;
        }
    }

    public override nint GetRowsInComponent(UIPickerView pickerView, nint component)
    {
        return Items != null ? Items.Count : 0;
    }

    public override string GetTitle(UIPickerView pickerView, nint row, nint component)
    {
        return Items != null && Items.Count > row ? Items[(int)row] : null;
    }

    public override nint GetComponentCount(UIPickerView pickerView)
    {
        return 1;
    }

    public override void Selected(UIPickerView pickerView, nint row, nint component)
    {
        selectedIndex = (int)row;
        if (ItemChanged != null)
        {
            ItemChanged.Invoke(this, new EventArgs());
        }
    }
}
```

`PickerModel` 类提供的基础存储 `MyUIPickerView` 类，通过 `Items` 属性。每当中的选定的项 `MyUIPickerView` 更改，`Selected` 执行方法时，该更新所选的索引和激发 `ItemChanged` 事件。这可确保 `SelectedItem` 属性将始终返回由用户选择的最后一项。此外，`PickerModel` 类重写方法用于设置 `MyUIPickerView` 实例。

Android

Android 实现子类 `Spinner` 视图中，并公开属性和从 XAML 可轻松使用的事件：

```
class MySpinner : Spinner
{
    ArrayAdapter adapter;
    IList<string> items;

    public IList<string> ItemsSource
    {
        get { return items; }
        set
        {
            if (items != value)
            {
                items = value;
                adapter.Clear();

                foreach (string str in items)
                {
                    adapter.Add(str);
                }
            }
        }
    }

    public string SelectedObject
    {
        get { return (string)GetItemAtPosition(SelectedItemPosition); }
        set
        {
            if (items != null)
            {
                int index = items.IndexOf(value);
                if (index != -1)
                {
                    SetSelection(index);
                }
            }
        }
    }

    public MySpinner(Context context) : base(context)
    {
        ItemSelected += OnBindableSpinnerItemSelected;

        adapter = new ArrayAdapter(context, Android.Resource.Layout.SimpleSpinnerItem);
        adapter.SetDropDownViewResource(Android.Resource.Layout.SimpleSpinnerDropDownItem);
        Adapter = adapter;
    }

    void OnBindableSpinnerItemSelected(object sender, ItemSelectedEventArgs args)
    {
        SelectedObject = (string)GetItemAtPosition(args.Position);
    }
}
```

`MySpinner` 类公开 `ItemsSource` 并 `SelectedObject` 属性，和一个 `ItemSelected` 事件。显示的项目 `MySpinner` 类提供的 `Adapter` 与视图关联和项填充到 `Adapter` 时 `ItemsSource` 首次设置属性。每当中的选定的项 `MySpinner` 类的更改，`OnBindableSpinnerItemSelected` 事件处理程序更新 `SelectedObject` 属性。

总结

本文演示了如何使用 Xamarin.Forms XAML 文件中的本机视图。上的本机视图，可以设置属性和事件处理程序，它们可以与 Xamarin.Forms 视图进行交互。

相关链接

- NativeSwitch（示例）
- Forms2Native（示例）
- NativeViewInsideContentView（示例）
- SubclassedNativeControls（示例）
- 本机窗体
- 在 XAML 中传递自变量

在 C# 中的本机视图

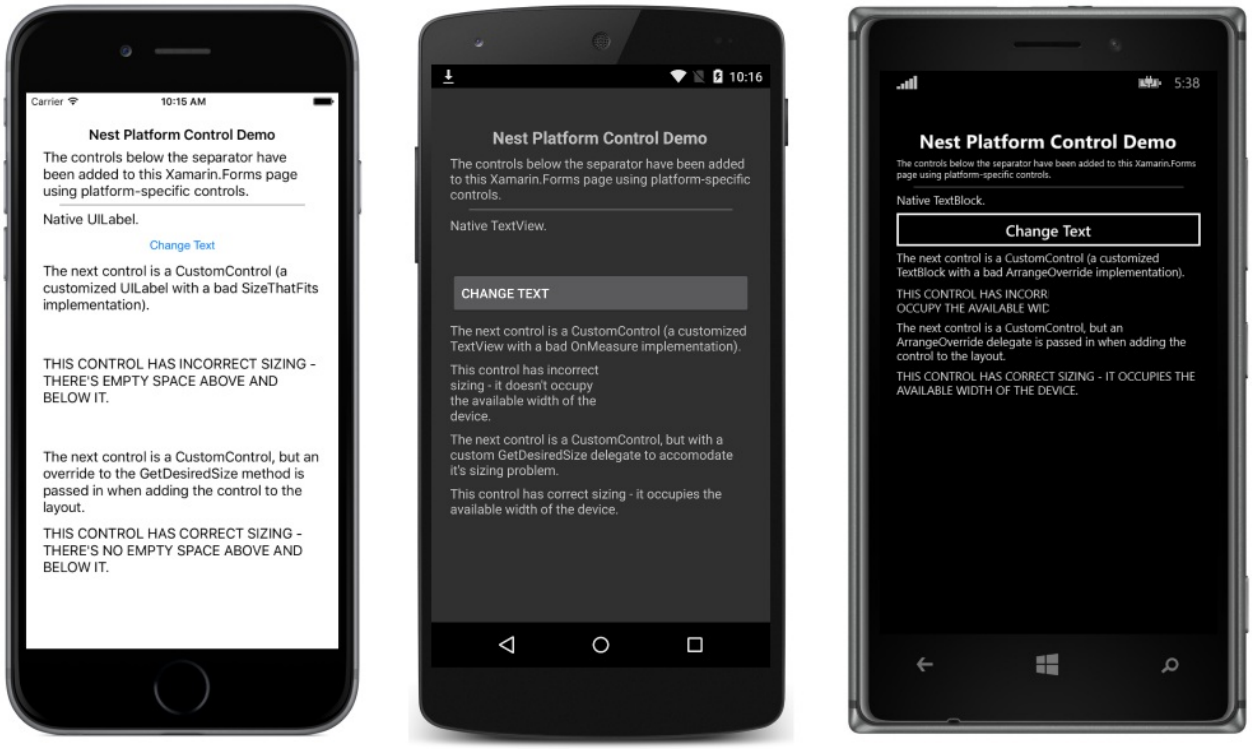
2018/10/26 • • Edit Online

从 iOS、Android 和 UWP 的本机视图，可以从使用 C# 创建的 Xamarin.Forms 页面直接引用。本文演示如何将本机视图添加到使用 C# 中，创建的 Xamarin.Forms 布局以及如何重写自定义视图来更正其度量 API 使用情况的布局。

概述

允许任何 Xamarin.Forms 控件 `Content` 设置，或具有 `Children` 集合中，可以添加特定于平台的视图。例如，iOS `UILabel` 可以直接添加到 `ContentView.Content` 属性，或设置为 `StackLayout.Children` 集合。但请注意，此功能需要使用 `#if` Xamarin.Forms 共享项目解决方案中定义，不提供从 Xamarin.Forms.NET Standard 库解决方案。

下面的屏幕截图演示了特定于平台的视图具有已添加到 Xamarin.Forms `StackLayout`：



能够将特定于平台的视图添加到 Xamarin.Forms 布局被启用的每个平台上的两个扩展方法：

- `Add` – 将添加到的特定于平台的视图 `Children` 布局的集合。
- `ToView` – 采用特定于平台的视图，并将其包装为 Xamarin.Forms `View` 可以将设置为 `Content` 控件的属性。

在 Xamarin.Forms 共享项目中使用这些方法需要导入相应的特定于平台的 Xamarin.Forms 命名空间：

- **iOS** – Xamarin.Forms.Platform.iOS
- **Android** – Xamarin.Forms.Platform.Android
- **通用 Windows 平台 (UWP)** – Xamarin.Forms.Platform.UWP

添加每个平台上的特定于平台的视图

以下各节演示如何将特定于平台的视图添加到每个平台上的 Xamarin.Forms 布局。

iOS

下面的代码示例演示如何添加 `UILabel` 到 `StackLayout` 和一个 `ContentView`：

```
var uiLabel = new UILabel {
  MinimumFontSize = 14f,
  Lines = 0,
  LineBreakMode = UILineBreakMode.WordWrap,
  Text = originalText,
};
stackLayout.Children.Add (uiLabel);
contentView.Content = uiLabel.ToView();
```

该示例假定 `stackLayout` 和 `contentView` 具有以前在 XAML 或 C# 中创建实例。

Android

下面的代码示例演示如何添加 `TextView` 到 `StackLayout` 和一个 `ContentView`：

```
var textView = new TextView (MainActivity.Instance) { Text = originalText, TextSize = 14 };
stackLayout.Children.Add (textView);
contentView.Content = textView.ToView();
```

该示例假定 `stackLayout` 和 `contentView` 具有以前在 XAML 或 C# 中创建实例。

通用 Windows 平台

下面的代码示例演示如何添加 `TextBlock` 到 `StackLayout` 和一个 `ContentView`：

```
var textBlock = new TextBlock
{
    Text = originalText,
    FontSize = 14,
    FontFamily = new FontFamily("HelveticaNeue"),
    TextWrapping = TextWrapping.Wrap
};
stackLayout.Children.Add(textBlock);
contentView.Content = textBlock.ToView();
```

该示例假定 `stackLayout` 和 `contentView` 具有以前在 XAML 或 C# 中创建实例。

重写自定义视图的平台度量

每个平台上的自定义视图通常只正确地实现设计它们的布局方案用于度量值。例如，自定义视图可能旨在仅占用设备的可用宽度的一半。但是，与其他用户共享之后，自定义视图可能需要占用的设备的完整可用宽度。因此，可能需要时重新使用 Xamarin.Forms 布局中重写自定义视图度量实现。因此，`Add` 和 `ToView` 扩展方法提供允许度量委托来指定，它时将其添加到 Xamarin.Forms 布局可以重写自定义视图布局的替代。

以下各节演示如何重写自定义视图，以更正其度量 API 使用情况的布局。

iOS

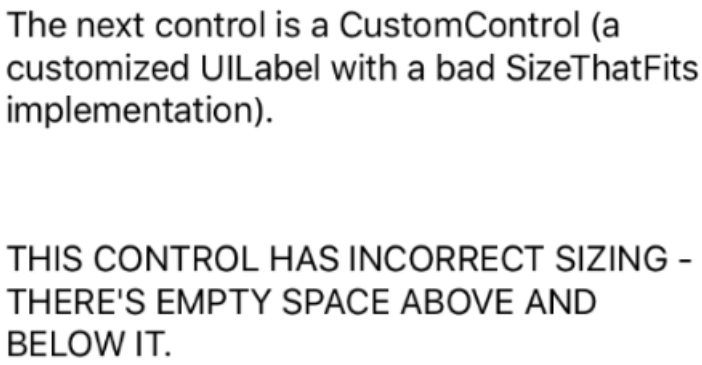
下面的代码示例演示 `CustomControl` 类，该类继承自 `UILabel`：

```
public class CustomControl : UILabel
{
  public override string Text {
    get { return base.Text; }
    set { base.Text = value.ToUpper (); }
  }

  public override CGSize SizeThatFits (CGSize size)
  {
    return new CGSize (size.Width, 150);
  }
}
```

此视图的实例添加到 `StackLayout`，如以下代码示例所示：

```
var customControl = new CustomControl {
  MinimumFontSize = 14,
  Lines = 0,
  LineBreakMode = UILineBreakMode.WordWrap,
  Text = "This control has incorrect sizing - there's empty space above and below it."
};
stackLayout.Children.Add (customControl);
```

但是，因为 `CustomControl.SizeThatFits` 重写将始终返回高度为 150，视图将显示为具有空白区域的上方和下方，如以下屏幕截图中所示：



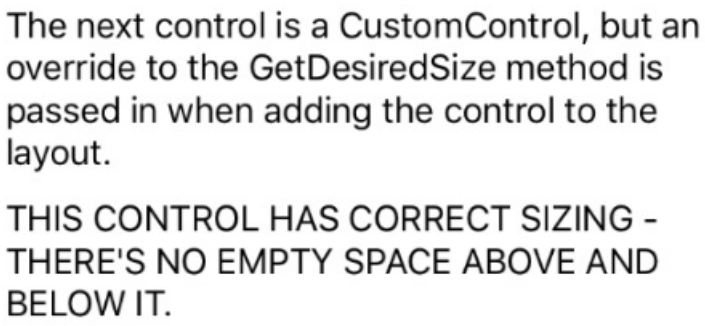
此问题的解决方案是提供 `GetDesiredSizeDelegate` 实现，如下面的代码示例中所示：

```
SizeRequest? FixSize (NativeViewWrapperRenderer renderer, double width, double height)
{
  var uiView = renderer.Control;

  if (uiView == null) {
    return null;
  }

  var constraint = new CGSize (width, height);

  // Let the CustomControl determine its size (which will be wrong)
  var badRect = uiView.SizeThatFits (constraint);

  // Use the width and substitute the height
  return new SizeRequest (new Size (badRect.Width, 70));
}
```

此方法使用提供的宽度 `CustomControl.SizeThatFits` 方法，但将替换为 70%高度为 150 的高度。当 `CustomControl` 实例添加到 `StackLayout` ，则 `FixSize` 方法可以指定为 `GetDesiredSizeDelegate` 若要修复错误所提供的度量 `CustomControl` 类：

```
stackLayout.Children.Add (customControl, FixSize);
```

这会导致自定义视图显示正确，而无需空白区域的上方和下方，如以下屏幕截图中所示：



Android

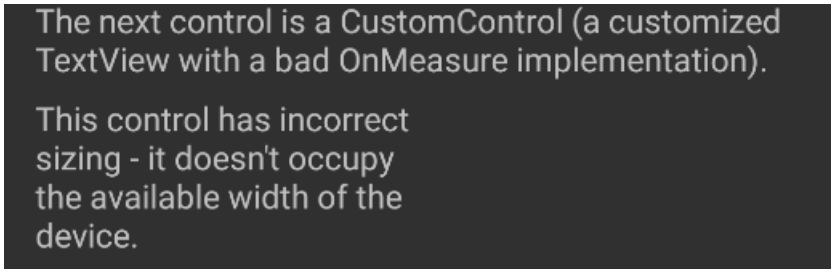
下面的代码示例演示 `CustomControl` 类，该类继承自 `TextView` ：

```
public class CustomControl : TextView
{
  public CustomControl (Context context) : base (context)
  {
  }

  protected override void OnMeasure (int widthMeasureSpec, int heightMeasureSpec)
  {
    int width = MeasureSpec.GetSize (widthMeasureSpec);

    // Force the width to half of what's been requested.
    // This is deliberately wrong to demonstrate providing an override to fix it with.
    int widthSpec = MeasureSpec.MakeMeasureSpec (width / 2, MeasureSpec.GetMode (widthMeasureSpec));

    base.OnMeasure (widthSpec, heightMeasureSpec);
  }
}
```

此视图的实例添加到 `StackLayout` ，如以下代码示例所示：

```
var customControl = new CustomControl (MainActivity.Instance) {
  Text = "This control has incorrect sizing - it doesn't occupy the available width of the device.",
  TextSize = 14
};
stackLayout.Children.Add (customControl);
```

但是，因为 `CustomControl.OnMeasure` 重写将始终返回请求的宽度的一半，则视图将显示占用只完成了一半的可用宽度的设备，如以下屏幕截图中所示：



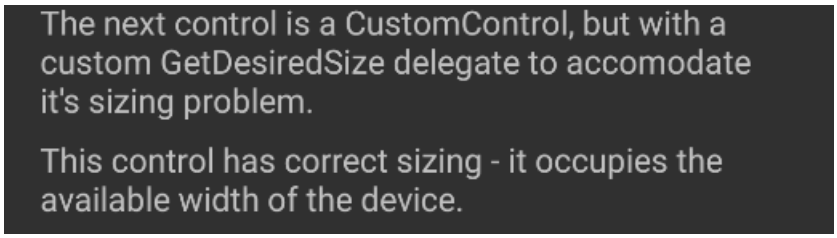
此问题的解决方案是提供 `GetDesiredSizeDelegate` 实现，如下面的代码示例中所示：

```
SizeRequest? FixSize (NativeViewWrapperRenderer renderer, int widthConstraint, int heightConstraint)
{
  var nativeView = renderer.Control;

  if ((widthConstraint == 0 && heightConstraint == 0) || nativeView == null) {
    return null;
  }

  int width = Android.Views.View.MeasureSpec.GetSize (widthConstraint);
  int widthSpec = Android.Views.View.MeasureSpec.MakeMeasureSpec (
    width * 2, Android.Views.View.MeasureSpec.GetMode (widthConstraint));
  nativeView.Measure (widthSpec, heightConstraint);
  return new SizeRequest (new Size (nativeView.MeasuredWidth, nativeView.MeasuredHeight));
}
```

此方法使用提供的宽度 `CustomControl.OnMeasure` 方法，但将其乘以 2。当 `CustomControl` 实例添加到 `StackLayout`，则 `FixSize` 方法可以指定为 `GetDesiredSizeDelegate` 若要修复错误所提供的度量 `CustomControl` 类：

```
stackLayout.Children.Add (customControl, FixSize);
```

这会导致自定义视图所显示正确，占用的宽度的设备，如以下屏幕截图中所示：



通用 Windows 平台

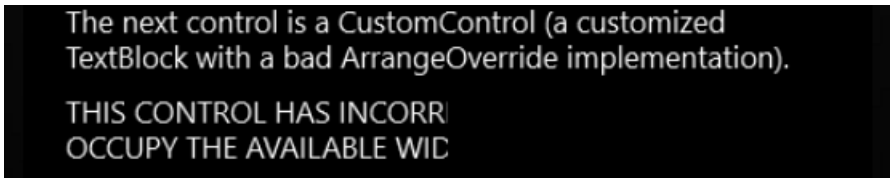
下面的代码示例演示 `CustomControl` 类，该类继承自 `Panel`：

```
public class CustomControl : Panel
{
  public static readonly DependencyProperty TextProperty =
    DependencyProperty.Register(
      "Text", typeof(string), typeof(CustomControl), new PropertyMetadata(default(string),
OnTextPropertyChanged));

  public string Text
  {
    get { return (string)GetValue(TextProperty); }
    set { SetValue(TextProperty, value.ToUpper()); }
  }

  readonly TextBlock textBlock;

  public CustomControl()
  {
    textBlock = new TextBlock
    {
      MinHeight = 0,
      MaxHeight = double.PositiveInfinity,
      MinWidth = 0,
      MaxWidth = double.PositiveInfinity,
      FontSize = 14,
      TextWrapping = TextWrapping.Wrap,
      VerticalAlignment = VerticalAlignment.Center
    };

    Children.Add(textBlock);
  }

  static void OnTextPropertyChanged(DependencyObject dependencyObject, DependencyPropertyChangedEventArgs
args)
  {
    ((CustomControl)dependencyObject).textBlock.Text = (string)args.NewValue;
  }

  protected override Size ArrangeOverride(Size finalSize)
  {
      // This is deliberately wrong to demonstrate providing an override to fix it with.
      textBlock.Arrange(new Rect(0, 0, finalSize.Width/2, finalSize.Height));
      return finalSize;
  }

  protected override Size MeasureOverride(Size availableSize)
  {
      textBlock.Measure(availableSize);
      return new Size(textBlock.DesiredSize.Width, textBlock.DesiredSize.Height);
  }
}
```

此视图的实例添加到 `StackLayout`，如以下代码示例所示：

```
var brokenControl = new CustomControl {
  Text = "This control has incorrect sizing - it doesn't occupy the available width of the device."
};
stackLayout.Children.Add(brokenControl);
```

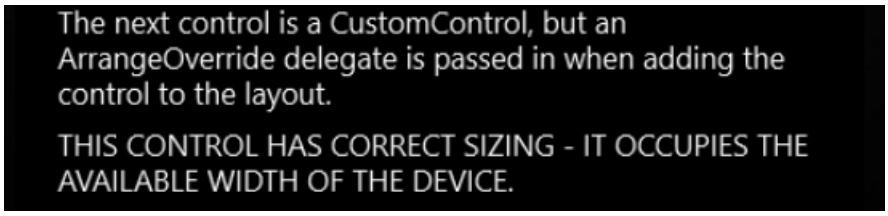
但是，因为 `CustomControl.ArrangeOverride` 重写将始终返回请求的宽度的一半，将为可用宽度的一半的设备，剪辑视图，如以下屏幕截图中所示：



此问题的解决方案是提供 `ArrangeOverrideDelegate` 实现中，添加到视图时 `StackLayout`，如以下代码示例所示：

```
stackLayout.Children.Add(fixedControl, arrangeOverrideDelegate: (renderer, finalSize) =>
{
    if (finalSize.Width <= 0 || double.IsInfinity(finalSize.Width))
    {
        return null;
    }
    var frameworkElement = renderer.Control;
    frameworkElement.Arrange(new Rect(0, 0, finalSize.Width * 2, finalSize.Height));
    return finalSize;
});
```

此方法使用提供的宽度 `CustomControl.ArrangeOverride` 方法，但将其乘以 2。这会导致自定义视图所显示正确，占用的宽度的设备，如以下屏幕截图中所示：



总结

本文介绍如何将本机视图添加到使用 C# 中，创建的 Xamarin.Forms 布局以及如何重写自定义视图来更正其度量 API 使用情况的布局。

相关链接

- NativeEmbedding（示例）
- 本机窗体

平台特定信息

2018/10/25 • • Edit Online

平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。

Xamarin.Forms 视图、页面和布局提供了以下特定于平台的功能：

IOS	ANDROID	WINDOWS
VisualElement.BlurEffect	VisualElement.Elevation	VisualElement.AccessKey、VisualElement.AccessKeyPlacement、VisualElement.AccessKeyHorizontalOffset 和 VisualElement.AccessKeyVerticalOffset
VisualElement.IsLegacyColorModeEnabled	VisualElement.IsLegacyColorModeEnabled	VisualElement.IsLegacyColorModeEnabled
VisualElement.IsShadowEnabled		

Xamarin.Forms 视图提供以下特定于平台的功能：

IOS	ANDROID	WINDOWS
Entry.AdjustsFontSizeToFitWidth	Button.UseDefaultPadding 和 Button.UseDefaultShadow	InputView.DetectReadingOrderFromContent Label.DetectReadingOrderFromContent
Entry.CursorColor	Entry.ImeOptions	ListView.SelectionMode
ListView.SeparatorStyle	ListView.IsFastScrollEnabled	SearchBar.IsSpellCheckEnabled
Picker.UpdateMode	WebView.MixedContentMode	WebView.IsJavaScriptAlertEnabled
Slider.UpdateOnTap		

Xamarin.Forms 页面提供以下特定于平台的功能：

IOS	ANDROID	WINDOWS
NavigationPage.HideSeparatorBar	NavigationPage.BarHeight	MasterDetailPage.CollapsedPaneWidth 和 MasterDetailPage.CollapseStyle
NavigationPage.IsNavigationBarTranslucent	TabbedPage.IsSmoothScrollEnabled	Page.ToolbarPlacement
NavigationPage.StatusBarTextColorMode	TabbedPage.IsSwipePagingEnabled	TabbedPage.HeaderIconsEnabled 和 TabbedPage.HeaderIconsSize
NavigationPage.PrefersLargeTitles	TabbedPage.ToolbarPlacement、TabbedPage.BarItemColor 和 TabbedPage.BarSelectedItemColor	

IOS	ANDROID	WINDOWS
Page.ModalPresentationStyle		
Page.PrefersStatusBarHidden 和 Page.PreferredStatusBarUpdateAnimation		
Page.UseSafeArea		

以下特定于平台的功能用于 Xamarin.Forms 布局：

IOS
ScrollView.ShouldDelayContentTouches

以下特定于平台的功能提供适用于 Xamarin.Forms `Application` 类：

IOS	ANDROID
Application.PanGestureRecognizerShouldRecognizeSimultaneously	Application.WindowSoftInputModeAdjust
	Application.SendDisappearingEventOnPause、Application.SendAppearingEventOnResume 和 Application.ShouldPreserveKeyboardOnResume

使用平台特定信息

使用特定于平台的通过 XAML，或通过 fluent 代码 API 的过程如下所示：

1. 添加 `xmlns` 声明或 `using` 指令 `Xamarin.Forms.PlatformConfiguration` 命名空间。
2. 添加 `xmlns` 声明或 `using` 指令包含特定于平台的功能的命名空间：
   a. 在 iOS 上，这是 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间。
   b. 在 Android 上，这是 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间。这是 Android AppCompat `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` 命名空间。
   c. 在通用 Windows 平台上，这是 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间。
3. 将特定于平台的应用从 XAML，或通过从代码 `On<T>` fluent API。值 `T` 可以是 `iOS`，`Android`，或 `Windows` 类型从 `Xamarin.Forms.PlatformConfiguration` 命名空间。

> **NOTE**
> 请注意，尝试使用特定于平台的是不可用的平台上不会导致错误。相反，该代码将执行而无需特定于平台的应用。

通过使用平台特定信息 `On<T>` fluent 代码 API 返回 `IPlatformElementConfiguration` 对象。这允许多个平台特定信息以与方法级联在同一对象上调用。

有关平台特定信息的详细信息，请参阅使用平台特定信息并创建平台特定信息。

相关链接

- 使用平台特定信息
- 创建平台特定信息

- PlatformSpecifics（示例）
- PlatformConfiguration

使用平台特定信息

2018/6/9 • • Edit Online

使用提供功能，仅在特定平台上，而无需实现自定义呈现器或效应。

iOS

本文演示如何使用 iOS 平台的细节，它们构建于 Xamarin.Forms。

Android

本文演示如何使用 Android 平台的详细信息，它们构建于 Xamarin.Forms。

Windows

本文演示如何使用 Windows 平台的详细信息，它们构建于 Xamarin.Forms。

iOS 平台特定信息

2018/11/13 • • Edit Online

平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。本文演示如何使用 iOS 平台特定信息的内置于 Xamarin.Forms。

VisualElements

在 iOS 上，Xamarin.Forms 视图、页面和布局提供了以下特定于平台的功能：

- 模糊处理的任何支持 `VisualElement` 。有关详细信息，请参阅应用模糊。
- 禁用上受支持的旧颜色模式 `VisualElement` 。有关详细信息，请参阅禁用旧式颜色模式。
- 在启用投影 `VisualElement` 。有关详细信息，请参阅启用投影。

应用模糊

此特定于平台的使用进行模糊处理框架，下面的内容和设置在 XAML 中由 `VisualElement.BlurEffect` 附加属性的值为 `BlurEffectStyle` 枚举：

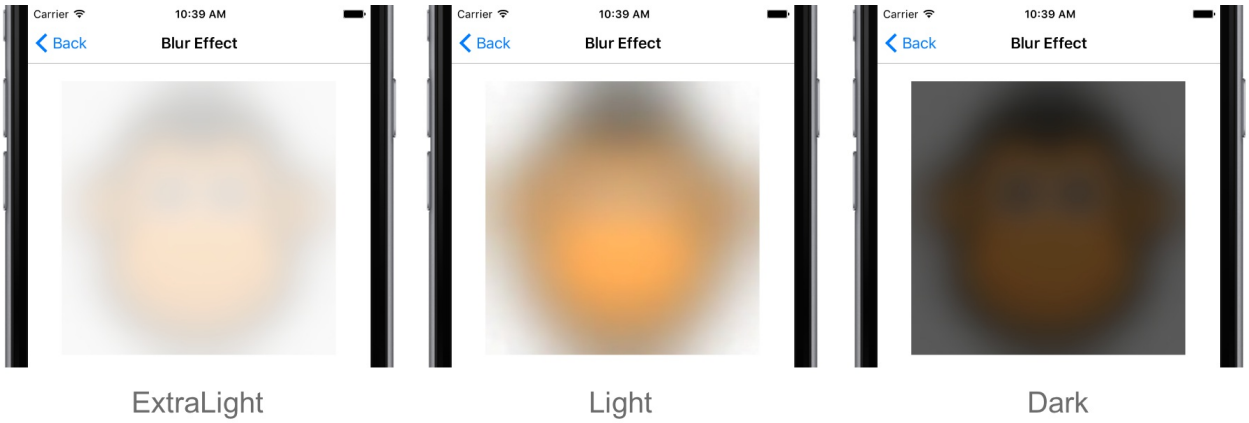
```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
  ...
  <AbsoluteLayout HorizontalOptions="Center">
      <Image Source="monkeyface.png" />
      <BoxView x:Name="boxView" ios:VisualElement.BlurEffect="ExtraLight" HeightRequest="300"
WidthRequest="300" />
  </AbsoluteLayout>
  ...
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

boxView.On<iOS>().UseBlurEffect(BlurEffectStyle.ExtraLight);
```

`BoxView.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`VisualElement.UseBlurEffect` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于将模糊效果应用与 `BlurEffectStyle` 提供四个枚举值：`None` ， `ExtraLight` ， `Light` ，并 `Dark` 。

结果是，指定 `BlurEffectStyle` 应用于 `BoxView` 实例的模糊化 `Image` 框架下面：



> **NOTE**
>
> 添加到的模糊效果时 `VisualElement` ，仍将由接收触摸事件 `VisualElement` 。

禁用旧式颜色模式

某些 Xamarin.Forms 视图功能旧颜色模式。在此模式下，当 `IsEnabled` 视图的属性设置为 `false` ，视图将重写由具有已禁用状态的默认本机颜色用户设置的颜色。有关向后兼容性，这种旧颜色模式保持不受支持视图的默认行为。

此特定于平台的禁用此旧颜色模式，以便对视图由用户设置的颜色保持，即使禁用的视图。设置使用在 XAML `VisualElement.IsLegacyColorModeEnabled` 附加到属性 `false` ：

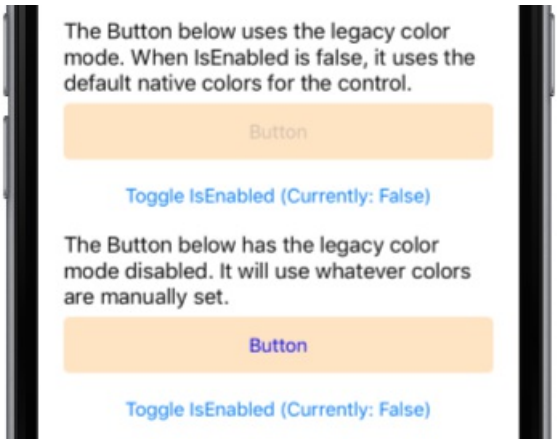
```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        ...
        <Button Text="Button"
                TextColor="Blue"
                BackgroundColor="Bisque"
                ios:VisualElement.IsLegacyColorModeEnabled="False" />
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

_legacyColorModeDisabledButton.On<iOS>().SetIsLegacyColorModeEnabled(false);
```

`VisualElement.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。 `VisualElement.SetIsLegacyColorModeEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否将禁用旧颜色模式。此外， `VisualElement.GetIsLegacyColorModeEnabled` 方法可以用于返回是否禁用旧颜色模式。

结果是，可以禁用旧版颜色模式，以便对视图由用户设置的颜色甚至保持禁用视图时：



> **NOTE**
> 设置时 `VisualStateGroup` 旧颜色模式被完全忽略上一个视图。可视状态的详细信息，请参阅Xamarin.Forms 视觉状态管理器。

启用投影

此特定于平台的用于上启用投影 `VisualElement` 。设置使用在 XAML `VisualElement.IsShadowEnabled` 附加到属性 `true`，以及许多其他可选的附加控制投影的属性：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        <BoxView ...
                 ios:VisualElement.IsShadowEnabled="true"
                 ios:VisualElement.ShadowColor="Purple"
                 ios:VisualElement.ShadowOpacity="0.7"
                 ios:VisualElement.ShadowRadius="12">
            <ios:VisualElement.ShadowOffset>
                <Size>
                    <x:Arguments>
                        <x:Double>10</x:Double>
                        <x:Double>10</x:Double>
                    </x:Arguments>
                </Size>
            </ios:VisualElement.ShadowOffset>
         </BoxView>
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

var boxView = new BoxView { Color = Color.Aqua, WidthRequest = 100, HeightRequest = 100 };
boxView.On<iOS>()
       .SetIsShadowEnabled(true)
       .SetShadowColor(Color.Purple)
       .SetShadowOffset(new Size(10,10))
       .SetShadowOpacity(0.7)
       .SetShadowRadius(12);
```

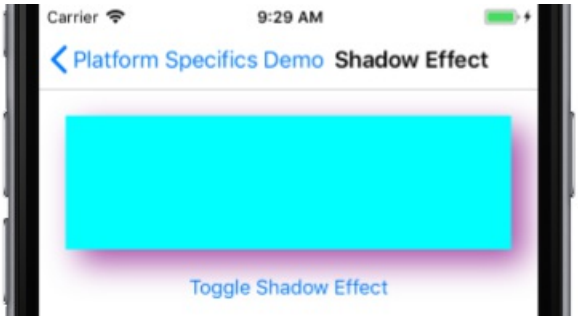
`VisualElement.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`VisualElement.SetIsShadowEnabled` 方法，在

`Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否在启用投影 `VisualElement`。此外，可以调用以下方法来控制投影：

- `SetShadowColor` -设置投影的颜色。默认颜色 `Color.Default`。
- `SetShadowOffset` -设置投影的偏移量。偏移量更改阴影被强制转换，并指定为方向 `Size` 值。`Size` 正在向左（负值）或向右（正值）的距离的第一个值和第二个值被更高版本的距离（负值）或下方（正值）结构的值以与设备无关单位表示. 此属性的默认值为 (0.0, 0.0)，这会导致卷影被强制转换涉及的每个方面 `VisualElement`。
- `SetShadowOpacity` – 要在范围 0.0（透明）到 1.0（不透明）的值设置投影的不透明度。默认不透明度值为 0.5。
- `SetShadowRadius` – 设置用于呈现阴影的模糊半径。默认半径值为 10.0。

> **NOTE**
>
> 可以通过调用查询投影的状态 `GetIsShadowEnabled`，`GetShadowColor`，`GetShadowOffset`，`GetShadowOpacity`，并 `GetShadowRadius` 方法。

结果是，可以在启用投影 `VisualElement`：



视图

在 iOS 上，为 Xamarin.Forms 视图提供以下特定于平台的功能：

- 确保输入的文本适合 `Entry` 通过调整字体大小。有关详细信息，请参阅调整项的字体大小。
- 在中设置的游标颜色 `Entry`。有关详细信息，请参阅设置条目游标颜色。
- 设置分隔符样式 `ListView`。有关详细信息，请参阅ListView 设置分隔符样式。
- 控制当项选择发生在 `Picker`。有关详细信息，请参阅控制选取器项选择。
- 启用 `Slider.Value` 属性可以通过点击对一个位置上设置 `Slider` 栏中，而不是按无需将 `Slider` thumb。有关详细信息，请参阅启用上点击移动到滑块。

调整项的字体大小

此特定于平台的用于缩放的字体大小 `Entry` 以确保适合所的文本的控件。设置使用在 XAML `Entry.AdjustsFontSizeToFitWidth` 附加到属性 `boolean` 值：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    <StackLayout Margin="20">
        <Entry x:Name="entry"
               Placeholder="Enter text here to see the font size change"
               FontSize="22"
               ios:Entry.AdjustsFontSizeToFitWidth="true" />
        ...
    </StackLayout>
</ContentPage>
```

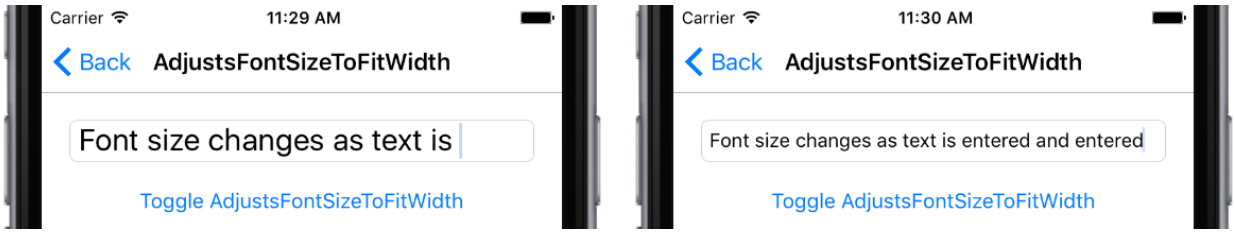
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

entry.On<iOS>().EnableAdjustsFontSizeToFitWidth();
```

`Entry.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Entry.EnableAdjustsFontSizeToFitWidth` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于调整所要确保它符合的文本的字号 `Entry` 。此外，`Entry` 类 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间还具有 `DisableAdjustsFontSizeToFitWidth` 禁用此平台特定的方法和一个 `SetAdjustsFontSizeToFitWidth` 方法可用于切换字体大小缩放通过调用该方法 `AdjustsFontSizeToFitWidth` 方法：

```
entry.On<iOS>().SetAdjustsFontSizeToFitWidth(!entry.On<iOS>().AdjustsFontSizeToFitWidth());
```

结果是的字号 `Entry` 进行缩放，以确保所的文本放在控件：



设置条目光标颜色

此特定于平台的设置中的光标颜色 `Entry` 为指定的颜色。设置使用在 XAML `Entry.CursorColor` 可绑定属性设置为 `Color` ：

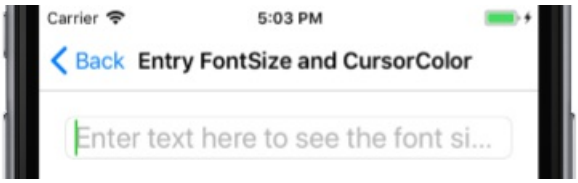
```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <Entry ... ios:Entry.CursorColor="LimeGreen" />
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

var entry = new Xamarin.Forms.Entry();
entry.On<iOS>().SetCursorColor(Color.LimeGreen);
```

`Entry.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Entry.SetCursorColor` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间中，将光标颜色设置为指定 `Color` 。此外，`Entry.GetCursorColor` 方法可以用于检索当前游标颜色。

结果是，中的光标颜色 `Entry` 可以设置为特定 `Color` ：



设置 ListView 的分隔符样式

此特定于平台的控制是否中单元格之间的分隔符 `ListView` 使用的完整宽度 `ListView` 。设置使用在 XAML `ListView.SeparatorStyle` 附加属性的值为 `SeparatorStyle` 枚举：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        <ListView ... ios:ListView.SeparatorStyle="FullWidth">
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

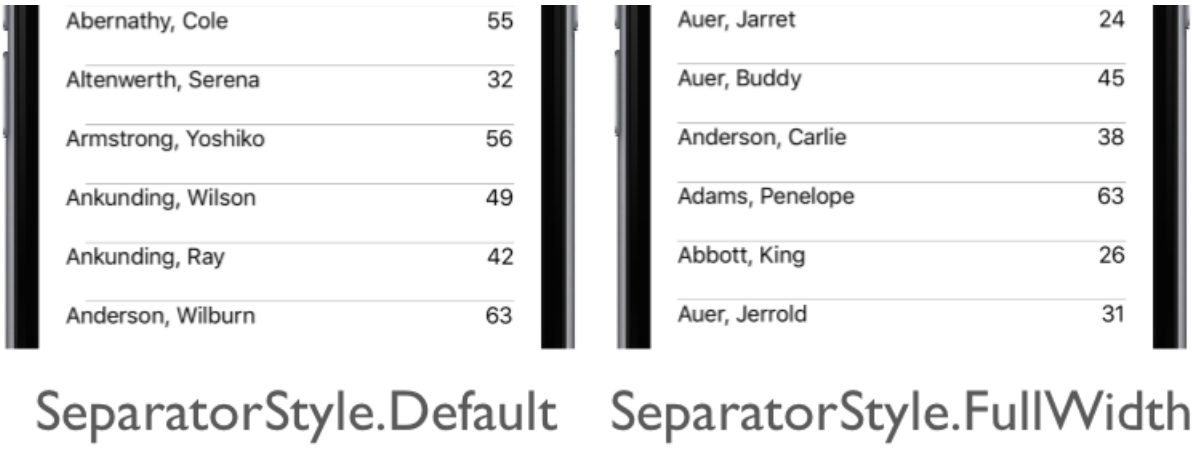
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

listView.On<iOS>().SetSeparatorStyle(SeparatorStyle.FullWidth);
```

`ListView.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。 `ListView.SetSeparatorStyle` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否之间的分隔符中的单元格 `ListView` 使用完整宽度 `ListView` ，使用 `SeparatorStyle` 枚举提供两个可能值：

- `Default` – 指示默认 iOS 分隔符行为。这是在 Xamarin.Forms 中的默认行为。
- `FullWidth` – 指示分隔符将来自某条边的 `ListView` 到其他。

结果是，指定 `SeparatorStyle` 值应用于 `ListView` ，它可以控制单元格之间的分隔符的宽度：



> **NOTE**
> 一旦分隔符样式设置为 `FullWidth` ，则不能更改回 `Default` 在运行时。

控制选取器项选择

此特定于平台的控件项选择中的发生时 `Picker` ，这样就允许用户指定的项选择发生时浏览项在控件中，还是仅后完成按下按钮。设置使用在 XAML `Picker.UpdateMode` 附加属性的值为 `UpdateMode` 枚举：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        <Picker ... Title="Select a monkey" ios:Picker.UpdateMode="WhenFinished">
          ...
        </Picker>
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
```

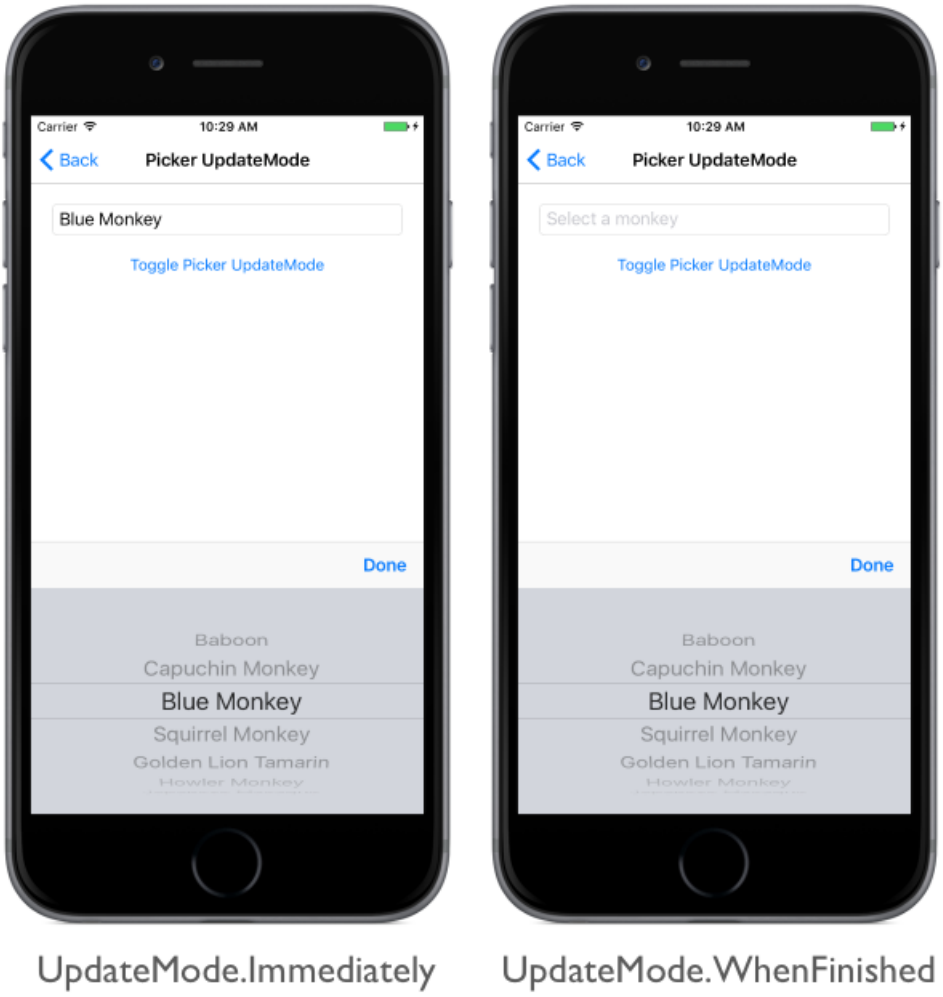
`Picker.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Picker.SetUpdateMode` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制项选择的发生时间、与 `UpdateMode` 枚举提供两个可能值：

- `Immediately` – 项选择会出现在用户浏览中的项 `Picker` 。这是在 Xamarin.Forms 中的默认行为。
- `WhenFinished` – 项选择仅发生后用户已按完成按钮 `Picker` 。

此外，`SetUpdateMode` 方法可用于切换的枚举值通过调用 `UpdateMode` 方法，返回当前 `UpdateMode`：

```
switch (picker.On<iOS>().UpdateMode())
{
    case UpdateMode.Immediately:
        picker.On<iOS>().SetUpdateMode(UpdateMode.WhenFinished);
        break;
    case UpdateMode.WhenFinished:
        picker.On<iOS>().SetUpdateMode(UpdateMode.Immediately);
        break;
}
```

结果是，指定 `UpdateMode` 应用于 `Picker` ，它可以控制项选择发生时：



启用移动上点击的滑块

此特定于平台的支持 `Slider.Value` 属性可以通过点击对一个位置上设置 `Slider` 栏中，而不是按无需将 `Slider` thumb。设置使用在 XAML `Slider.UpdateOnTap` 可绑定属性设置为 `true`：

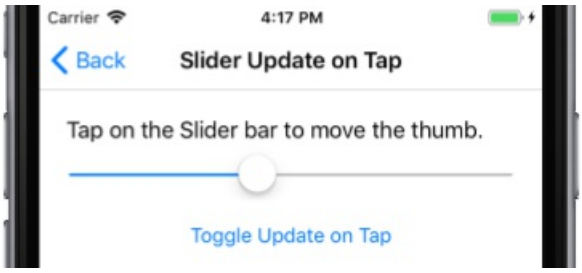
```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout ...>
        <Slider ... ios:Slider.UpdateOnTap="true" />
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

var slider = new Xamarin.Forms.Slider();
slider.On<iOS>().SetUpdateOnTap(true);
```

`Slider.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Slider.SetUpdateOnTap` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否在点击 `Slider` 条将设置 `Slider.Value` 属性。此外，`Slider.GetUpdateOnTap` 方法可用于返回是否上点击 `Slider` 条将设置 `Slider.Value` 属性。

结果是，在点击 `Slider` 栏可以移动 `Slider` thumb 和设置 `Slider.Value` 属性：



Pages

在 iOS 上，为 Xamarin.Forms 页面提供以下特定于平台的功能：

- 在隐藏导航栏分隔符 `NavigationPage` 。有关详细信息，请参阅隐藏导航栏分隔符在 NavigationPage。
- 控制是否半透明的导航栏。有关详细信息，请参阅进行导航栏半透明。
- 控制是否状态栏文本的颜色上 `NavigationPage` 调整以匹配导航栏的亮度。有关详细信息，请参阅调整状态栏文本颜色模式。
- 控制是否将页标题显示为页导航栏中的大型标题。有关详细信息，请参阅显示大标题。
- 设置状态条可见性 `Page` 。有关详细信息，请参阅页上设置状态条可见性。
- 确保该页面内容位于上是安全的所有 iOS 设备的屏幕区域。有关详细信息，请参阅启用安全区域布局指南。
- 在 iPad 上设置模式页面演示文稿的样式。有关详细信息，请参阅在 iPad 上设置模式页面演示文稿样式。

隐藏导航栏上 NavigationPage 的分隔条

此特定于平台的隐藏的分隔线和位于底部的导航栏的卷影 `NavigationPage` 。设置使用在 XAML `NavigationPage.HideNavigationBarSeparator` 可绑定属性设置为 `false`：

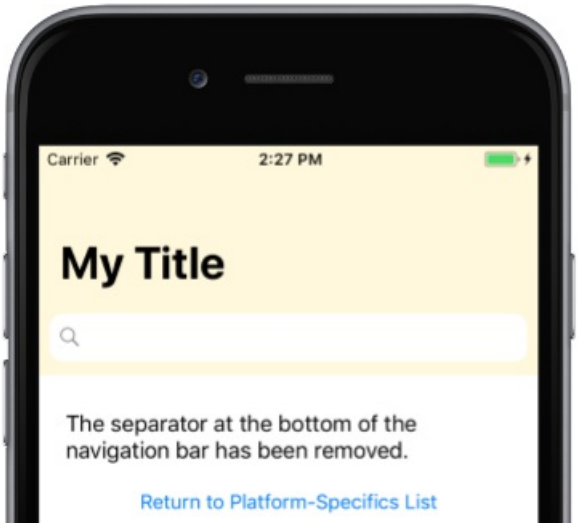
```
<NavigationPage ...
                xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
                ios:NavigationPage.HideNavigationBarSeparator="true">

</NavigationPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;

public class iOSTitleViewNavigationPageCS : Xamarin.Forms.NavigationPage
{
    public iOSTitleViewNavigationPageCS()
    {
        On<iOS>().SetHideNavigationBarSeparator(true);
    }
}
```

`NavigationPage.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。 `NavigationPage.SetHideNavigationBarSeparator` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否隐藏导航栏分隔符。此外， `NavigationPage.HideNavigationBarSeparator` 方法可以用于返回是否隐藏导航栏分隔符。

结果是，在导航栏分隔符 `NavigationPage` 可以隐藏：



使导航栏半透明

此特定于平台的用于更改导航栏中，透明度和消耗在 XAML 中的设置 `NavigationPage.IsNavigationBarTranslucent` 附加到属性 `boolean` 值：

```
<NavigationPage ...
                xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
                BackgroundColor="Blue"
                ios:NavigationPage.IsNavigationBarTranslucent="true">
  ...
</NavigationPage>
```

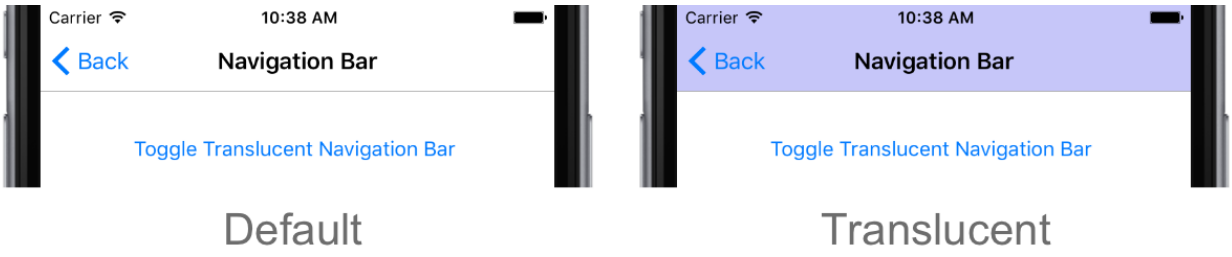
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

(App.Current.MainPage as Xamarin.Forms.NavigationPage).BackgroundColor = Color.Blue;
(App.Current.MainPage as Xamarin.Forms.NavigationPage).On<iOS>().EnableTranslucentNavigationBar();
```

`NavigationPage.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。 `NavigationPage.EnableTranslucentNavigationBar` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于使导航栏半透明。此外， `NavigationPage` 类 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间还具有 `DisableTranslucentNavigationBar` 还原为其默认状态，在导航栏的方法和一个 `SetIsNavigationBarTranslucent` 方法用于通过调用切换导航栏透明度 `IsNavigationBarTranslucent` 方法：

```
(App.Current.MainPage as Xamarin.Forms.NavigationPage)
  .On<iOS>()
  .SetIsNavigationBarTranslucent(!(App.Current.MainPage as Xamarin.Forms.NavigationPage).On<iOS>
().IsNavigationBarTranslucent());
```

结果是，可以更改导航栏的透明度：



调整状态栏文本颜色模式

此特定于平台的控件是否状态栏文本的颜色上 `NavigationPage` 调整以匹配导航栏的亮度。设置使用在 XAML `NavigationPage.StatusBarTextColorMode` 附加属性的值为 `StatusBarTextColorMode` 枚举：

```
<MasterDetailPage xmlns="http://xamarin.com/schemas/2014/forms"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    xmlns:ios="clr-namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
    x:Class="PlatformSpecifics.iOSStatusBarTextColorModePage">
    <MasterDetailPage.Master>
        <ContentPage Title="Master Page Title" />
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <NavigationPage BarBackgroundColor="Blue" BarTextColor="White"
                        ios:NavigationPage.StatusBarTextColorMode="MatchNavigationBarTextLuminosity">
            <x:Arguments>
                <ContentPage>
                    <Label Text="Slide the master page to see the status bar text color mode change." />
                </ContentPage>
            </x:Arguments>
        </NavigationPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

或者，可以使用它从 C# 使用 fluent API：

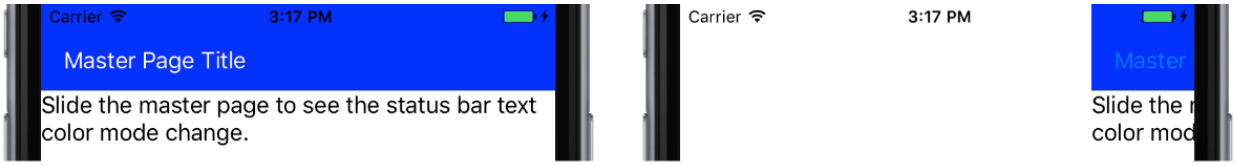
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

IsPresentedChanged += (sender, e) =>
{
    var mdp = sender as MasterDetailPage;
    if (mdp.IsPresented)
        ((Xamarin.Forms.NavigationPage)mdp.Detail)
          .On<iOS>()
          .SetStatusBarTextColorMode(StatusBarTextColorMode.DoNotAdjust);
    else
        ((Xamarin.Forms.NavigationPage)mdp.Detail)
          .On<iOS>()
          .SetStatusBarTextColorMode(StatusBarTextColorMode.MatchNavigationBarTextLuminosity);
};
```

`NavigationPage.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`NavigationPage.SetStatusBarTextColorMode` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间中，控件是否状态栏文本的颜色上 `NavigationPage` 调整以匹配亮度的导航栏中，使用 `StatusBarTextColorMode` 枚举提供两个可能值：

- `DoNotAdjust` – 指示不应调整状态栏文本颜色。
- `MatchNavigationBarTextLuminosity` – 表示文本颜色状态栏应匹配导航栏的亮度。

此外，`GetStatusBarTextColorMode` 方法可用于检索的当前值 `StatusBarTextColorMode` 枚举应用于 `NavigationPage` 。

结果是，在状态栏上的文本颜色 `NavigationPage` 可以进行调整以匹配导航栏的亮度。在此示例中，状态栏文本颜色更改为用户切换 `Master` 并 `Detail` 页 `MasterDetailPage`：



StatusBarTextColorMode .DoNotAdjust

StatusBarTextColorMode .MatchNavigationBarTextLuminosity

显示大标题

此特定于平台的用于导航栏中，对于使用 iOS 11 或更高版本的设备上的大型标题显示的页面标题。大型标题左对齐和使用的较大的图标，并将转换为标准标题当用户开始滚动的内容，以便有效地使用屏幕空间。但是，在横向方向，标题将返回到导航栏来优化内容布局的中心。设置使用在 XAML `NavigationPage.PrefersLargeTitles` 附加属性设置为 `boolean` 值：

```
<NavigationPage xmlns="http://xamarin.com/schemas/2014/forms"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
                xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
                ...
                ios:NavigationPage.PrefersLargeTitles="true">
  ...
</NavigationPage>
```

或者可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

var navigationPage = new Xamarin.Forms.NavigationPage(new iOSLargeTitlePageCS());
navigationPage.On<iOS>().SetPrefersLargeTitles(true);
```

`NavigationPage.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`NavigationPage.SetPrefersLargeTitle` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间中，控制是否启用大标题。

前提是大标题上启用 `NavigationPage` ，在导航堆栈中的所有页面将都显示大标题。可以通过设置页上写此行为 `Page.LargeTitleDisplay` 附加属性的值为 `LargeTitleDisplayMode` 枚举：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
             Title="Large Title"
             ios:Page.LargeTitleDisplay="Never">
  ...
</ContentPage>
```

或者，可以从使用 fluent API 通过 C# 重写页面行为：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

public class iOSLargeTitlePageCS : ContentPage
{
    public iOSLargeTitlePageCS(ICommand restore)
    {
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Never);
        ...
    }
    ...
}
```

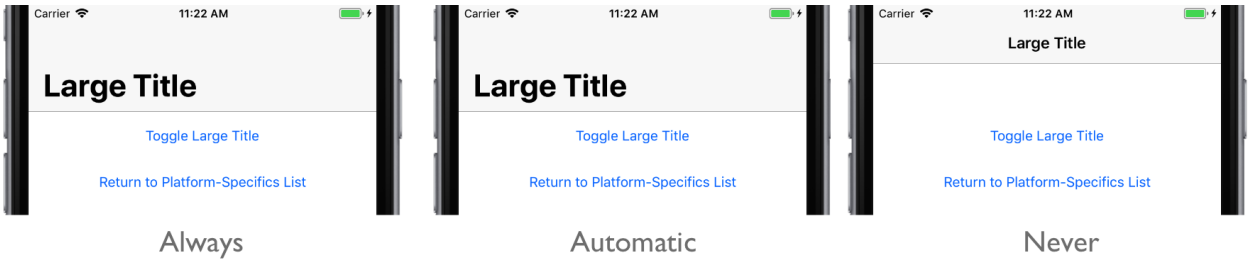
`Page.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Page.SetLargeTitleDisplay` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，在控制大标题行为 `Page` ，与 `LargeTitleDisplayMode` 提供三个可能的枚举值：

- `Always` – 强制的导航栏和字体大小，以使用较大的格式。
- `Automatic` – 相同的样式（大或小）用作导航堆栈中的上一项。
- `Never` – 强制使用正则、小型格式导航栏。

此外，`SetLargeTitleDisplay` 方法可用于切换的枚举值通过调用 `LargeTitleDisplay` 方法，返回当前 `LargeTitleDisplayMode` ：

```
switch (On<iOS>().LargeTitleDisplay())
{
    case LargeTitleDisplayMode.Always:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Automatic);
        break;
    case LargeTitleDisplayMode.Automatic:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Never);
        break;
    case LargeTitleDisplayMode.Never:
        On<iOS>().SetLargeTitleDisplay(LargeTitleDisplayMode.Always);
        break;
}
```

结果是，指定 `LargeTitleDisplayMode` 应用于 `Page` ，它可以控制大标题行为：



设置状态栏页上的可见性

此特定于平台的用于上设置的可见性状态栏 `Page` ，并包括能够控制状态栏如何进入或离开 `Page` 。设置使用在 XAML 中 `Page.PrefersStatusBarHidden` 附加属性设置为值 `StatusBarHiddenMode` 枚举，并选择性地 `Page.PreferredStatusBarUpdateAnimation` 附加属性的值为 `UIStatusBarAnimation` 枚举：

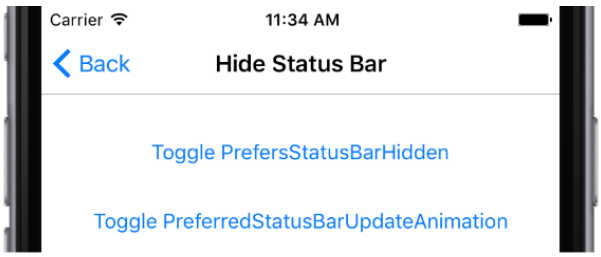
```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
             ios:Page.PrefersStatusBarHidden="True"
             ios:Page.PreferredStatusBarUpdateAnimation="Fade">
  ...
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

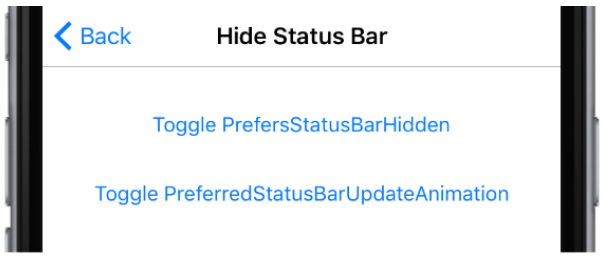
```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

On<iOS>().SetPrefersStatusBarHidden(StatusBarHiddenMode.True)
         .SetPreferredStatusBarUpdateAnimation(UIStatusBarAnimation.Fade);
```

`Page.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Page.SetPrefersStatusBarHidden` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于上设置的可见性状态栏 `Page` 通过指定之一 `StatusBarHiddenMode` 枚举值：`Default`， `True` 或 `False`。`StatusBarHiddenMode.True` 并 `StatusBarHiddenMode.False` 值设置而不考虑设备方向状态栏可见性和 `StatusBarHiddenMode.Default` 值将隐藏垂直 compact 环境中的状态栏。

结果是，在状态栏的可见性 `Page` 可以设置：



StatusBarHiddenMode.False



StatusBarHiddenMode.True

> **NOTE**
> 上 `TabbedPage`，指定 `StatusBarHiddenMode` 枚举值还将更新所有的子页面上的状态栏。所有其他 `Page` 的派生类型，指定的 `StatusBarHiddenMode` 枚举值只会更新当前页面上的状态栏。

`Page.SetPreferredStatusBarUpdateAnimation` 方法用于设置状态栏如何进入或离开 `Page` 通过指定之一 `UIStatusBarAnimation` 枚举值：`None`， `Fade`，或 `Slide`。如果 `Fade` 或 `Slide` 指定枚举值，0.25 第二个动画执行如状态栏进入或离开 `Page`。

启用安全区域布局指南

此特定于平台的用于确保页面内容定位在屏幕上，则可以使用 iOS 11 和更高版本的所有设备的安全区域上。具体而言，这将有助于确保该内容未剪辑的舍入的设备角部、家庭的指示符或在 iPhone X 上的传感器底座。设置使用在 XAML `Page.UseSafeArea` 附加属性设置为 `boolean` 值：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
             Title="Safe Area"
             ios:Page.UseSafeArea="true">
    <StackLayout>
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

On<iOS>().SetUseSafeArea(true);
```

`Page.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Page.SetUseSafeArea` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间中，控制是否启用安全区域布局指南。

结果是屏幕的页面内容可置于是屏幕的安全的所有 Iphone 区域：

Enabled

Disabled

> **NOTE**
> 定义由 Apple 的安全区域在 Xamarin.Forms 中用来设置 `Page.Padding` 属性，并且将重写此属性的任何以前已设置的值。

可以通过检索自定义安全区域及其 `Thickness` 值替换 `Page.SafeAreaInsets` 方法从 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间。它然后可修改为所需和重新分配给 `Padding` 该页的构造函数中的属性或 `OnAppearing` 重写：

```
protected override void OnAppearing()
{
    base.OnAppearing();

    var safeInsets = On<iOS>().SafeAreaInsets();
    safeInsets.Left = 20;
    Padding = safeInsets;
}
```

在 iPad 上设置模式页面演示文稿样式

此特定于平台的用于在 iPad 上设置的模式页面演示文稿样式。设置使用在 XAML `Page.ModalPresentationStyle` 可绑定属性设置为 `UIModalPresentationStyle` 枚举值：

```
<ContentPage ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
             ios:Page.ModalPresentationStyle="FormSheet">
    ...
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

public class iOSModalFormSheetPageCS : ContentPage
{
    public iOSModalFormSheetPageCS()
    {
        On<iOS>().SetModalPresentationStyle(UIModalPresentationStyle.FormSheet);
        ...
    }
}
```

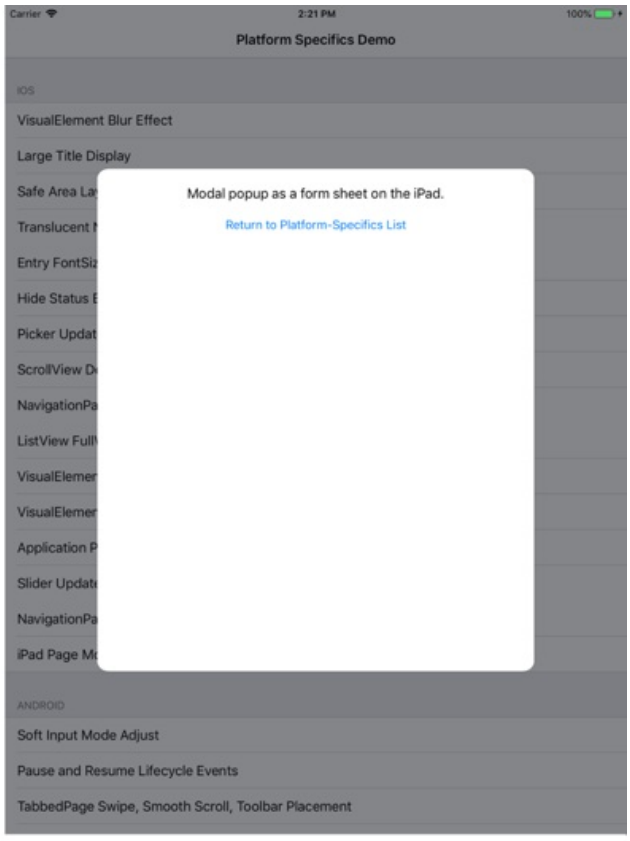
`Page.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Page.SetModalPresentationStyle` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于上设置模式的演示文稿样式 `Page` 通过指定以下项之一 `UIModalPresentationStyle` 枚举值：

- `FullScreen` 用于设置模式的演示文稿样式以覆盖整个屏幕。默认情况下，使用此演示文稿样式显示模式页面。
- `FormSheet` 用于设置模式的演示文稿样式上居中且小于屏幕。

此外，`GetModalPresentationStyle` 方法可以用于检索的当前值 `UIModalPresentationStyle` 应用于枚举 `Page` 。

结果是，在模式的演示文稿样式 `Page` 可以设置：

FullScreen



FormSheet

> **NOTE**
> 使用此特定于平台的设置模式的演示文稿样式的页面必须使用模式导航。有关详细信息，请参阅Xamarin.Forms 模式页面。

布局

在 iOS 上，以下特定于平台的功能用于 Xamarin.Forms 布局：

- 控制是否 `ScrollView` 处理触摸手势或将其传递给其内容。有关详细信息，请参阅延迟内容收尾工作了在 ScrollView。

ScrollView 中的延迟内容收尾工作

触摸手势中开始时触发的隐式计时器 `ScrollView` 在 iOS 上和 `ScrollView` 决定是否应处理手势，也可以将其传递给其内容基于计时器的范围内中的用户操作。默认情况下，iOS `ScrollView` 延迟内容的收尾工作了，但这可能会问题导致在某些情况下使用 `ScrollView` 不应获胜手势的内容。因此，此特定于平台的控件是否 `ScrollView` 处理触摸手势或将其传递给其内容。设置使用在 XAML `ScrollView.ShouldDelayContentTouches` 附加属性设置为 `boolean` 值：

```
<MasterDetailPage ...
                  xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core">
    <MasterDetailPage.Master>
        <ContentPage Title="Menu" BackgroundColor="Blue" />
    </MasterDetailPage.Master>
    <MasterDetailPage.Detail>
        <ContentPage>
            <ScrollView x:Name="scrollView" ios:ScrollView.ShouldDelayContentTouches="false">
                <StackLayout Margin="0,20">
                    <Slider />
                    <Button Text="Toggle ScrollView DelayContentTouches" Clicked="OnButtonClicked" />
                </StackLayout>
            </ScrollView>
        </ContentPage>
    </MasterDetailPage.Detail>
</MasterDetailPage>
```

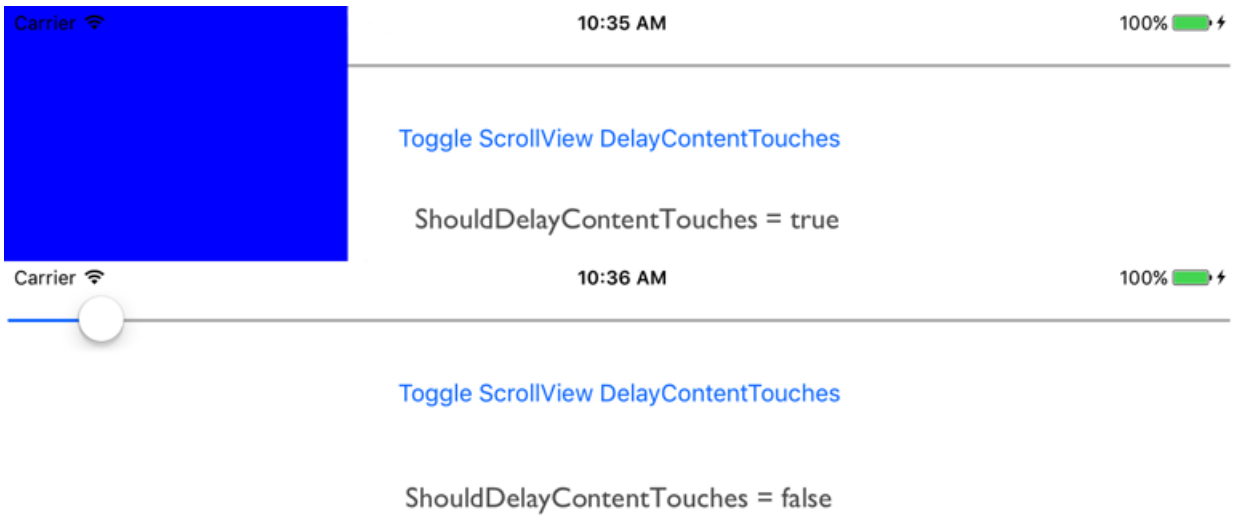
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

scrollView.On<iOS>().SetShouldDelayContentTouches(false);
```

`ScrollView.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`ScrollView.SetShouldDelayContentTouches` 方法，请在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，是否用于控制 `ScrollView` 处理触摸手势或将其传递给其内容。此外，`SetShouldDelayContentTouches` 方法可用于切换通过调用延迟内容收尾工作了 `ShouldDelayContentTouches` 方法以返回是否延迟内容收尾工作了：

```
scrollView.On<iOS>().SetShouldDelayContentTouches(!scrollView.On<iOS>().ShouldDelayContentTouches());
```

结果是，`ScrollView` 可以禁用延迟接收内容收尾工作了，因此，在这种情况下 `Slider` 接收手势而不是 `Detail` 页的 `MasterDetailPage`：



应用程序

在 iOS 上，以下特定于平台的功能提供适用于 Xamarin.Forms `Application` 类：

- 启用 `PanGestureRecognizer` 中滚动视图来捕获和共享平移手势与滚动视图。有关详细信息，请参阅启用同时进

行的平移手势识别。

启用同时进行的平移手势识别

当 `PanGestureRecognizer` 附加到内部滚动视图中，所有平移手势捕获的一个视图，`PanGestureRecognizer` 并不会传递给滚动视图。因此，将无法再滚动滚动视图。

此特定于平台的使 `PanGestureRecognizer` 中滚动视图来捕获和共享平移手势与滚动视图。设置使用在 XAML `Application.PanGestureRecognizerShouldRecognizeSimultaneously` 附加到属性 `true`：

```
<Application ...
             xmlns:ios="clr-
namespace:Xamarin.Forms.PlatformConfiguration.iOSSpecific;assembly=Xamarin.Forms.Core"
             ios:Application.PanGestureRecognizerShouldRecognizeSimultaneously="true">
    ...
</Application>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.iOSSpecific;
...

Xamarin.Forms.Application.Current.On<iOS>().SetPanGestureRecognizerShouldRecognizeSimultaneously(true);
```

`Application.On<iOS>` 方法指定仅将在 iOS 上运行此特定于平台的。`Application.SetPanGestureRecognizerShouldRecognizeSimultaneously` 方法，在 `Xamarin.Forms.PlatformConfiguration.iOSSpecific` 命名空间，用于控制是否在滚动视图中的平移手势识别程序将捕获平移手势，或捕获并共享平移手势与滚动视图。此外，`Application.GetPanGestureRecognizerShouldRecognizeSimultaneously` 方法可用于返回是否与包含的滚动视图共享平移手势 `PanGestureRecognizer` 。

因此，对于此特定于平台的启用，当 `ListView` 包含 `PanGestureRecognizer` ，这两个 `ListView` 和 `PanGestureRecognizer` 将收到平移手势和对其进行处理。但是，对于此特定于平台的禁用状态，当 `ListView` 包含 `PanGestureRecognizer` ，则 `PanGestureRecognizer` 将捕获平移手势并处理它，并 `ListView` 不会收到平移手势。

总结

本文演示了如何使用 iOS 平台特定信息的内置于 Xamarin.Forms。平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。

相关链接

- 创建平台特定信息
- PlatformSpecifics（示例）
- iOSSpecific

Android 平台特定信息

2018/11/13 • • Edit Online

平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。本文演示如何使用 Android 平台特定信息的内置于 Xamarin.Forms。

VisualElements

在 Android 上，Xamarin.Forms 视图、页面和布局提供了以下特定于平台的功能：

- 控制 Z 顺序的可视元素来确定绘制顺序。有关详细信息，请参阅控制可视元素提升的。
- 禁用上受支持的旧颜色模式 `VisualElement` 。有关详细信息，请参阅禁用旧式颜色模式。

控制可视元素的提升

此特定于平台的是用于控制提升或 Z 顺序，对应用程序的可视元素的面向 API 21 或更高版本。一个可视元素提升确定其绘制顺序，使用具有更高版本的 Z 值 occluding 具有较低的 Z 值的可视元素的可视元素。设置使用在 XAML `VisualElement.Elevation` 附加属性设置为 `boolean` 值：

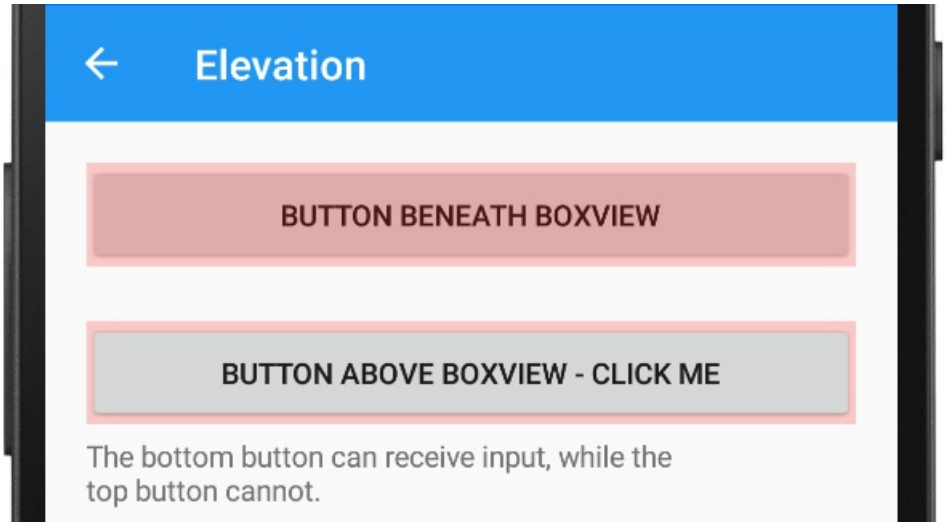
```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
             Title="Elevation">
    <StackLayout>
        <Grid>
            <Button Text="Button Beneath BoxView" />
            <BoxView Color="Red" Opacity="0.2" HeightRequest="50" />
        </Grid>
        <Grid Margin="0,20,0,0">
            <Button Text="Button Above BoxView - Click Me" android:VisualElement.Elevation="10"/>
            <BoxView Color="Red" Opacity="0.2" HeightRequest="50" />
        </Grid>
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

public class AndroidElevationPageCS : ContentPage
{
    public AndroidElevationPageCS()
    {
        ...
        var aboveButton = new Button { Text = "Button Above BoxView - Click Me" };
        aboveButton.On<Android>().SetElevation(10);

        Content = new StackLayout
        {
            Children =
            {
                new Grid
                {
                    Children =
                    {
                        new Button { Text = "Button Beneath BoxView" },
                        new BoxView { Color = Color.Red, Opacity = 0.2, HeightRequest = 50 }
                    }
                },
                new Grid
                {
                    Margin = new Thickness(0,20,0,0),
                    Children =
                    {
                        aboveButton,
                        new BoxView { Color = Color.Red, Opacity = 0.2, HeightRequest = 50 }
                    }
                }
            }
        };
    }
}
```

`Button.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。`VisualElement.SetElevation` 方法，请在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于设置为一个可以为 null 的可视元素的权限提升 `float`。此外，`VisualElement.GetElevation` 方法可以用于检索一个可视元素的提升值。

结果是，以便具有更高版本的 Z 值的可视元素遮蔽具有较低的 Z 值的可视元素，可以控制可视元素的提升。因此，在此示例中第二个 `Button` 上方呈现 `BoxView` 因为它具有较大的提升值：



禁用旧式颜色模式

某些 Xamarin.Forms 视图功能旧颜色模式。在此模式下，当 `IsEnabled` 视图的属性设置为 `false`，视图将重写由具有已禁用状态的默认本机颜色用户设置的颜色。有关向后兼容性，这种旧颜色模式保持不受支持视图的默认行为。

此特定于平台的禁用此旧颜色模式，以便对视图由用户设置的颜色保持，即使禁用的视图。设置使用在 XAML `VisualElement.IsLegacyColorModeEnabled` 附加到属性 `false`：

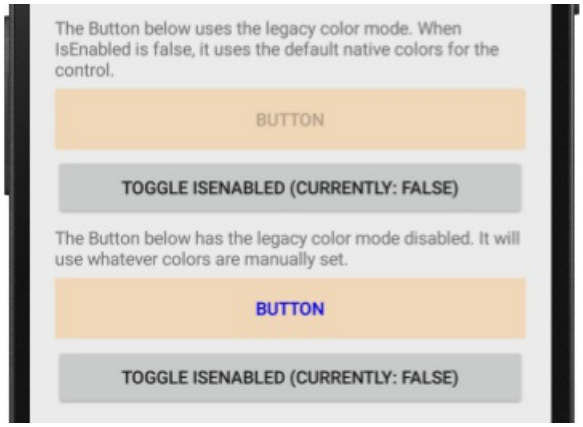
```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        ...
        <Button Text="Button"
                TextColor="Blue"
                BackgroundColor="Bisque"
                android:VisualElement.IsLegacyColorModeEnabled="False" />
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

_legacyColorModeDisabledButton.On<Android>().SetIsLegacyColorModeEnabled(false);
```

`VisualElement.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。`VisualElement.SetIsLegacyColorModeEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于控制是否将禁用旧颜色模式。此外，`VisualElement.GetIsLegacyColorModeEnabled` 方法可以用于返回是否禁用旧颜色模式。

结果是，可以禁用旧版颜色模式，以便对视图由用户设置的颜色甚至保持禁用视图时：



> **NOTE**
> 设置时 `VisualStateGroup` 旧颜色模式被完全忽略上一个视图。可视状态的详细信息，请参阅Xamarin.Forms 视觉状态管理器。

视图

在 Android 上，为 Xamarin.Forms 视图提供以下特定于平台的功能：

- 使用默认填充边距和阴影的 Android 按钮的值。有关详细信息，请参阅使用 Android 按钮。

- 输入的法编辑器为设置选项的软键盘 `Entry` 。有关详细信息，请参阅设置条目输入法编辑器选项。
- 启用快速滚动 `ListView` 的详细信息，请参阅ListView 中启用快速滚动。
- 控制是否 `WebView` 可以显示混合的内容。有关详细信息，请参阅中的启用混合内容 WebView。

使用 Android 按钮

此特定于平台的控制 Xamarin.Forms 按钮使用的默认填充边距和阴影的 Android 按钮的值。设置使用在 XAML `Button.UseDefaultPadding` 并 `Button.UseDefaultShadow` 附加属性设置为 `boolean` 值：

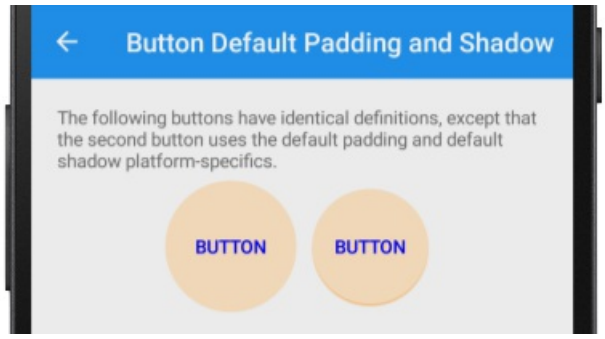
```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        ...
        <Button ...
                android:Button.UseDefaultPadding="true"
                android:Button.UseDefaultShadow="true" />
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

button.On<Android>().SetUseDefaultPadding(true).SetUseDefaultShadow(true);
```

`Button.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。 `Button.SetUseDefaultPadding` 并 `Button.SetUseDefaultShadow` 方法，请在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，是用于控制是否 Xamarin.Forms 按钮使用默认值填充和阴影的 Android 按钮的值。此外， `Button.UseDefaultPadding` 并 `Button.UseDefaultShadow` 方法可以用于返回按钮是否使用默认值分别填充值和默认卷影值。

结果是 Xamarin.Forms 按钮，可以使用默认填充和 Android 的按钮的卷影值：



请注意，在上述每个屏幕截图 `Button` 具有相同的定义，不同之处在于右侧 `Button` 使用默认填充边距和阴影的 Android 按钮的值。

设置条目输入法编辑器选项

此特定于平台的设置输入的法编辑器 (IME) 的软键盘选项 `Entry` 。这包括在角的软键盘和与之间的交互来设置用户操作按钮 `Entry` 。设置使用在 XAML `Entry.ImeOptions` 附加属性的值为 `ImeFlags` 枚举：

```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout ...>
        <Entry ... android:Entry.ImeOptions="Send" />
        ...
    </StackLayout>
</ContentPage>
```

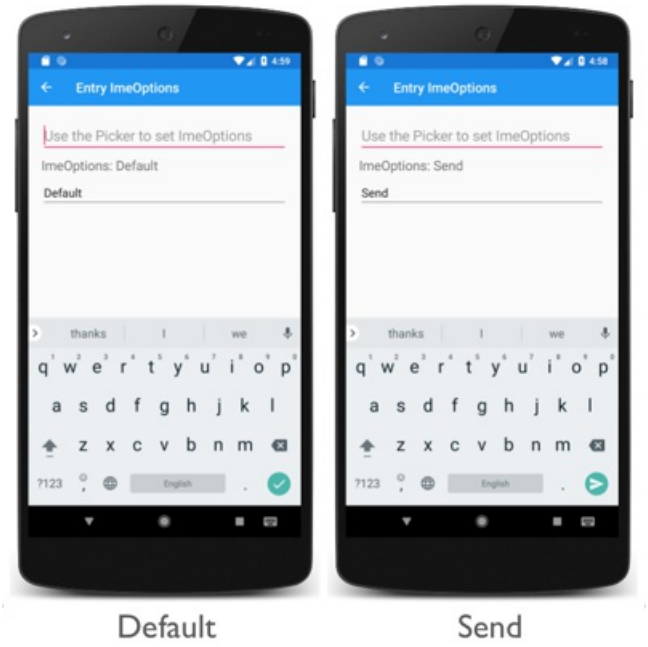
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

entry.On<Android>().SetImeOptions(ImeFlags.Send);
```

`Entry.On<Android>`方法指定仅将在 Android 上运行此特定于平台的。`Entry.SetImeOptions` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于设置的软键盘输入的法操作选项 `Entry` ，与 `ImeFlags` 枚举提供以下值：

- `Default` -指示没有特定操作的键是必需的并且基础控件将生成其自己如果它可以。这将是 `Next` 或 `Done` 。
- `None` -指示没有操作项会使可用。
- `Go` – 指示操作键将执行"转到"操作，它们使到文本的目标用户类型。
- `Search` – 指示操作键执行"搜索"操作，它们使用户的搜索文本结果具有类型化。
- `Send` – 指示操作键将执行一个"发送"操作，将文本传递到其目标。
- `Next` – 指示操作键将执行"下一步"操作，使用户为将接受文本的下一个字段。
- `Done` – 指示操作键将执行一个"done"的操作，关闭软键盘。
- `Previous` – 指示操作键将执行"上一个"操作，使用户将接受文本的上一个字段。
- `ImeMaskAction` – 若要选择操作选项的掩码。
- `NoPersonalizedLearning` – 指示，拼写检查器将既不了解从用户，也不提供根据用户以前已如何键入更正建议。
- `NoFullscreen` – 指示在 UI 不应处于全屏。
- `NoExtractUi` -指示对于提取的文本将不显示 UI。
- `NoAccessoryAction` -指示没有 UI，将显示用于自定义操作。

结果是，指定 `ImeFlags` 值将应用于软键盘 `Entry` ，用于设置输入的法编辑器选项：



启用快速滚动 ListView

此特定于平台的用于启用通过中的数据快速滚动 `ListView` 。设置使用在 XAML `ListView.IsFastScrollEnabled` 附加属性设置为 `boolean` 值：

```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout Margin="20">
        ...
        <ListView ItemsSource="{Binding GroupedEmployees}"
                  GroupDisplayBinding="{Binding Key}"
                  IsGroupingEnabled="true"
                  android:ListView.IsFastScrollEnabled="true">
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

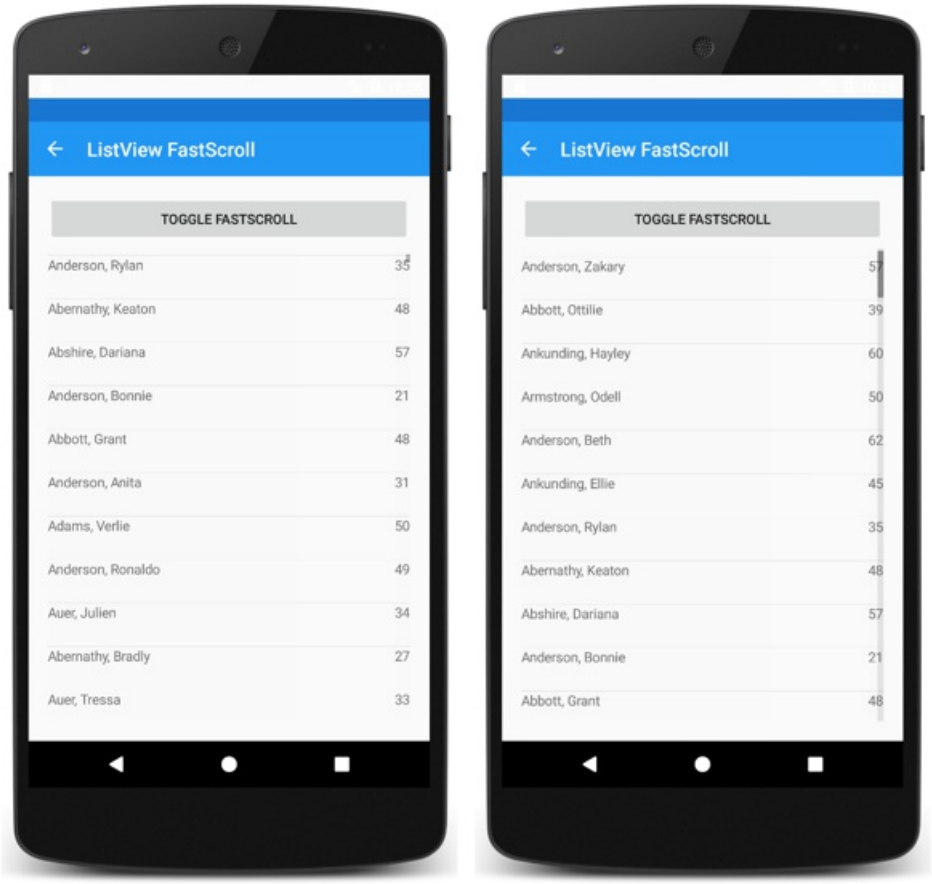
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

var listView = new Xamarin.Forms.ListView { IsGroupingEnabled = true, ... };
listView.SetBinding(ItemsView<Cell>.ItemsSourceProperty, "GroupedEmployees");
listView.GroupDisplayBinding = new Binding("Key");
listView.On<Android>().SetIsFastScrollEnabled(true);
```

`ListView.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。`ListView.SetIsFastScrollEnabled` 方法，请在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于启用通过中的数据快速滚动 `ListView` 。此外，`SetIsFastScrollEnabled` 方法可用于切换通过调用快速滚动 `IsFastScrollEnabled` 方法以返回指示是否启用快速滚动：

```
listView.On<Android>().SetIsFastScrollEnabled(!listView.On<Android>().IsFastScrollEnabled());
```

结果是通过中的数据的快速滚动 `ListView` 可以启用，这将更改滚动块的大小：



FastScroll Disabled FastScroll Enabled

启用 web 视图中的混合的内容

此特定于平台的控件是否 `WebView` 可以显示混合内容中的应用程序面向 API 21 或更高版本。混合的内容是的内容的最初已加载，通过 HTTPS 连接，但这会通过 HTTP 连接加载资源（如图像、音频、视频、样式表、脚本）。设置使用在 XAML `WebView.MixedContentMode` 附加属性的值为 `MixedContentHandling` 枚举：

```
<ContentPage ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core">
    <WebView ... android:WebView.MixedContentMode="AlwaysAllow" />
</ContentPage>
```

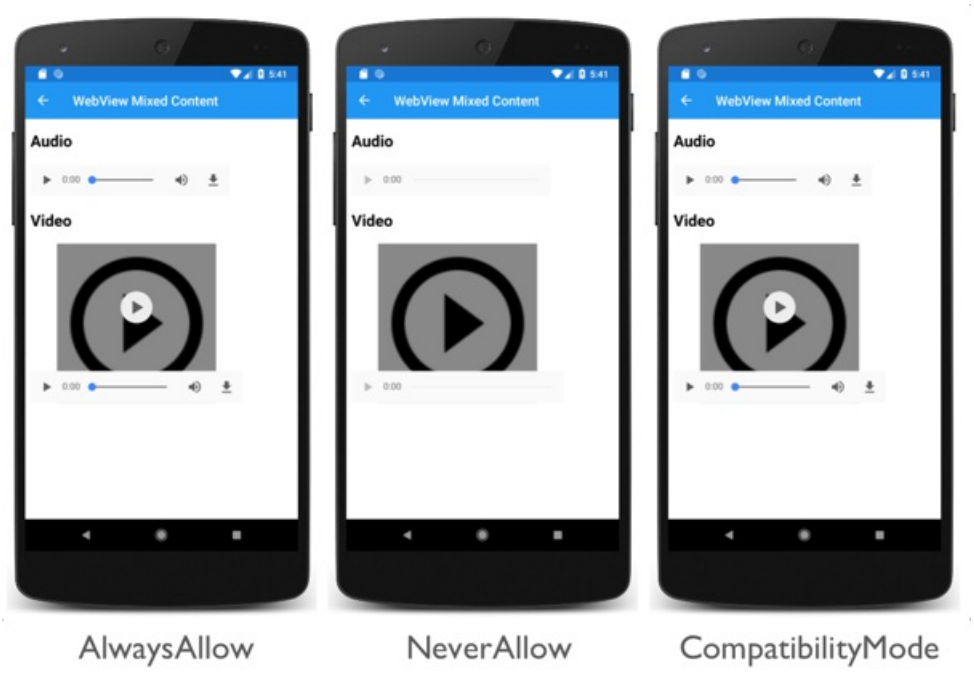
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

webView.On<Android>().SetMixedContentMode(MixedContentHandling.AlwaysAllow);
```

`WebView.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。`WebView.SetMixedContentMode` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于控制是否可以显示混合的内容，与 `MixedContentHandling` 提供三个可能值的枚举：

- `AlwaysAllow` – 指示 `WebView` 将允许从 HTTP 源加载内容的 HTTPS 原点。
- `NeverAllow` – 指示 `WebView` 将不允许 HTTPS origin 从 HTTP 源加载内容。
- `CompatibilityMode` – 指示 `WebView` 将尝试与最新的设备 web 浏览器的方法兼容。可能允许某些 HTTP 内容加载的 HTTPS origin 和其他类型的内容将被阻止。阻止或允许的内容的类型可能会随每个操作系统版本。

结果是，指定 `MixedContentHandling` 值应用于 `WebView`，它可以控制是否可以显示混合的内容：



Pages

在 Android 上，为 Xamarin.Forms 页面提供以下特定于平台的功能：

- 设置导航栏的高度 `NavigationPage` 。有关详细信息，请参阅NavigationPage 上设置导航栏高度。
- 中的页面中导航时禁用过渡动画 `TabbedPage` 。有关详细信息，请参阅TabbedPage 中禁用页面切换动画。
- 启用在页面间轻扫 `TabbedPage` 。有关详细信息，请参阅启用轻扫页面之间中您不要将 TabbedPage。
- 在设置工具栏位置和颜色 `TabbedPage` 。有关详细信息，请参阅设置 TabbedPage 工具栏位置和颜色。

设置导航栏上 NavigationPage 的高度

此特定于平台的设置在导航栏的高度 `NavigationPage` 。设置使用在 XAML `NavigationPage.BarHeight` 可绑定属性设置为一个整数值：

```
<NavigationPage ...
                xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;assembly=Xamarin.Forms.Core"
                android:NavigationPage.BarHeight="450">
    ...
</NavigationPage>
```

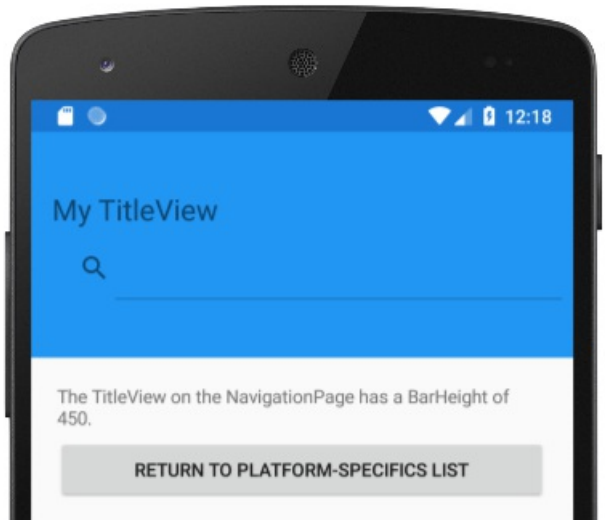
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;
...

public class AndroidNavigationPageCS : Xamarin.Forms.NavigationPage
{
    public AndroidNavigationPageCS()
    {
        On<Android>().SetBarHeight(450);
    }
}
```

`NavigationPage.On<Android>` 方法指定仅将在应用程序兼容性 Android 上运行此特定于平台的。

`NavigationPage.SetBarHeight` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` 命名空间，用于设置在导航栏的高度 `NavigationPage` 。此外， `NavigationPage.GetBarHeight` 方法可用于返回在导航栏的高度 `NavigationPage` 。

结果是，在导航栏的高度 `NavigationPage` 可以设置：



您不要将 TabbedPage 中禁用页面切换动画

此特定于平台的用于时的页面之间导航，或者以编程方式或在使用选项卡栏中，禁用过渡动画 `TabbedPage` 。设置使用在 XAML `TabbedPage.IsSmoothScrollEnabled` 可绑定属性设置为 `false` ：

```
<TabbedPage ...
            xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
            android:TabbedPage.IsSmoothScrollEnabled="false">
    ...
</TabbedPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

On<Android>().SetIsSmoothScrollEnabled(false);
```

`TabbedPage.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。 `TabbedPage.SetIsSmoothScrollEnabled` 方法，请在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于控制是否在页之间导航时，将显示过渡动画 `TabbedPage` 。此外， `TabbedPage` 类中 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间还具有以下方法：

- `IsSmoothScrollEnabled` 它用于检索是否之间导航页中，将显示过渡动画 `TabbedPage` 。
- `EnableSmoothScroll` 用于启用过渡动画中的页面之间导航时 `TabbedPage` 。
- `DisableSmoothScroll` 用于在页面之间导航时禁用过渡动画 `TabbedPage` 。

启用在您不要将 TabbedPage 页面间轻扫

此特定于平台的用于启用与中的页面水平手指笔势轻扫 `TabbedPage` 。设置使用在 XAML `TabbedPage.IsSwipePagingEnabled` 附加到属性 `boolean` 值：

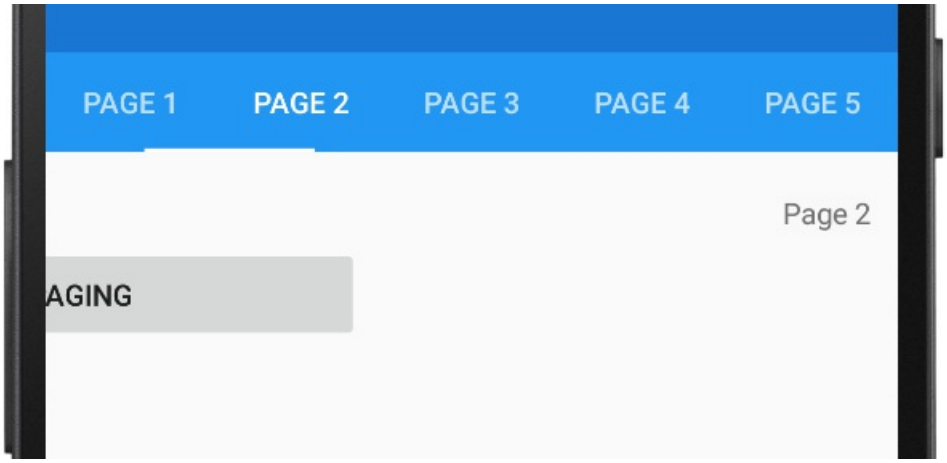
```
<TabbedPage ...
            xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
            android:TabbedPage.OffscreenPageLimit="2"
            android:TabbedPage.IsSwipePagingEnabled="true">
    ...
</TabbedPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

On<Android>().SetOffscreenPageLimit(2)
             .SetIsSwipePagingEnabled(true);
```

`TabbedPage.On<Android>`方法指定仅将在 Android 上运行此特定于平台的。`TabbedPage.SetIsSwipePagingEnabled`方法，在`Xamarin.Forms.PlatformConfiguration.AndroidSpecific`命名空间，用于启用中的页面轻扫`TabbedPage`。此外，`TabbedPage`类中`Xamarin.Forms.PlatformConfiguration.AndroidSpecific`命名空间也具有`EnableSwipePaging`方法，使此特定于平台的和一个`DisableSwipePaging`禁用的方法此特定于平台的。`TabbedPage.OffscreenPageLimit`附加属性，以及`SetOffscreenPageLimit`方法，用于设置应保留在当前页的任何一侧上空闲状态中的页面数。

结果是通过显示的页通过该轻扫分页`TabbedPage`已启用：



设置 TabbedPage 工具栏位置和颜色

这些平台特定信息用于上设置的位置和颜色的工具栏`TabbedPage`。它们由在 XAML 中设置`TabbedPage.ToolbarPlacement`附加属性的值为`ToolbarPlacement`枚举，以及`TabbedPage.BarItemColor`和`TabbedPage.BarSelectedItemColor`附加到的属性`Color`：

```
<TabbedPage ...
            xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
            android:TabbedPage.ToolbarPlacement="Bottom"
            android:TabbedPage.BarItemColor="Black"
            android:TabbedPage.BarSelectedItemColor="Red">
    ...
</TabbedPage>
```

或者，因此，可以使用从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

On<Android>().SetToolbarPlacement(ToolbarPlacement.Bottom)
             .SetBarItemColor(Color.Black)
             .SetBarSelectedItemColor(Color.Red);
```

`TabbedPage.On<Android>` 方法指定仅将在 Android 上运行这些平台特定信息。`TabbedPage.SetToolbarPlacement` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于上设置工具栏位置 `TabbedPage`，与 `ToolbarPlacement` 枚举提供以下值：

- `Default` – 指示工具栏位于页面上的默认位置。这是页的在手机上，页面顶部和底部上其他设备的惯用语言。
- `Top` – 指示工具栏被放在页面顶部。
- `Bottom` – 指示工具栏放置在页面的底部。

此外，`TabbedPage.SetBarItemColor` 并 `TabbedPage.SetBarSelectedItemColor` 方法用于分别设置工具栏项和选定的工具栏项的颜色。

> **NOTE**
> `GetToolbarPlacement`，`GetBarItemColor`，以及 `GetBarSelectedItemColor` 方法可用于检索位置和颜色 `TabbedPage` 工具栏。

结果是，可以在设置工具栏位置、工具栏项的颜色和所选的工具栏项的颜色 `TabbedPage`：

应用程序

在 Android 上，以下特定于平台的功能提供适用于 Xamarin.Forms `Application` 类：

- 设置屏幕键盘的操作模式。有关详细信息，请参阅软键盘输入模式设置。
- 禁用 `Disappearing` 并 `Appearing` 页生命周期事件上暂停和继续分别使用 AppCompat 的应用程序。有关详细信息，请参阅禁用消失并使其不显示页面生命周期事件。

设置屏幕键盘输入的模式

此特定于平台的用于设置屏幕键盘输入区域中，运行模式和设置在 XAML 中由 `Application.WindowSoftInputModeAdjust` 附加属性的值为 `WindowSoftInputModeAdjust` 枚举：

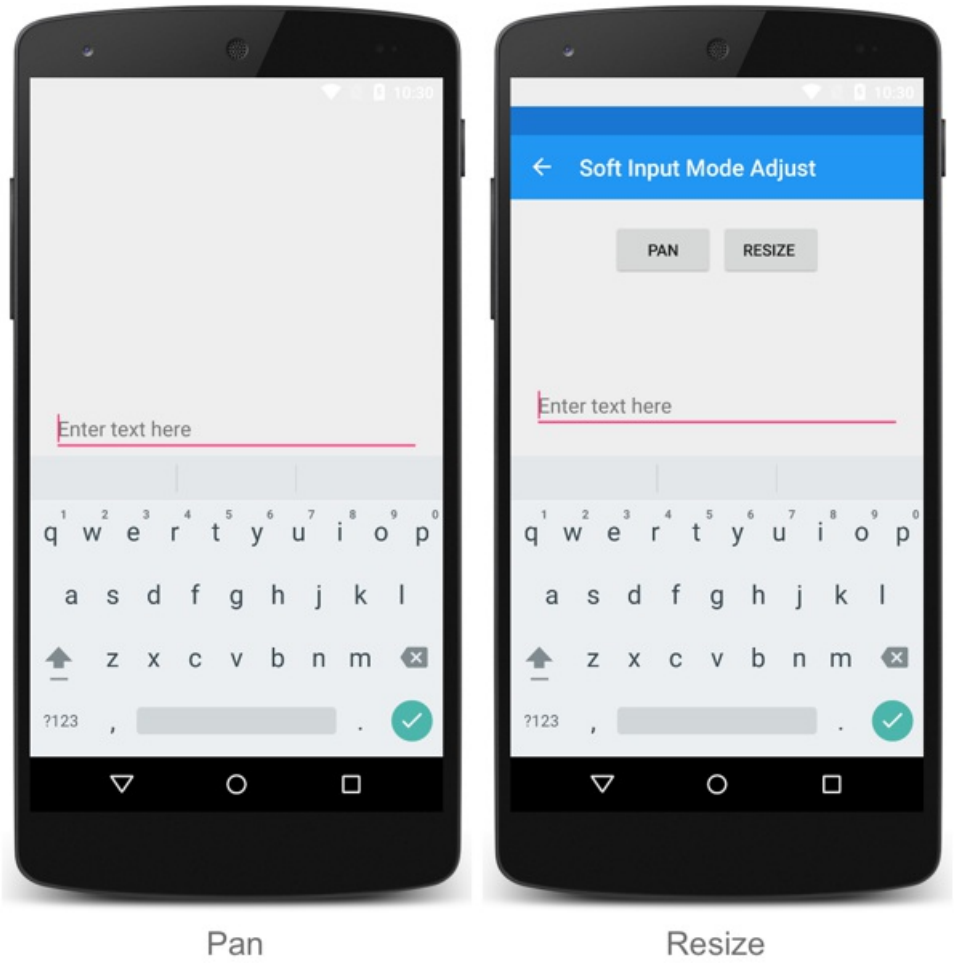
```
<Application ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
             android:Application.WindowSoftInputModeAdjust="Resize">
  ...
</Application>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
...

App.Current.On<Android>().UseWindowSoftInputModeAdjust(WindowSoftInputModeAdjust.Resize);
```

`Application.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。 `Application.UseWindowSoftInputModeAdjust` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific` 命名空间，用于设置屏幕键盘输入的区域的运行模式，与 `WindowSoftInputModeAdjust` 枚举提供两个值：`Pan` 并 `Resize` 。 `Pan` 值使用 `AdjustPan` 调整选项，当输入的控件具有焦点时都不会调整窗口大小。相反，以便当前焦点不会因软键盘遮住素数窗口的内容。 `Resize` 值使用 `AdjustResize` 输入的控件具有焦点，屏幕键盘的腾出空间，请调整窗口的大小调整选项。

结果是软键盘输入的输入的控件具有焦点时，可以设置运行模式的区域：



禁用消失和显示页面生命周期事件

此特定于平台的用来禁用 `Disappearing` 并 `Appearing` 上应用程序的页面事件暂停和继续分别使用 AppCompat 的应用程序。此外，它包含控件的功能是否软键盘显示在恢复，如果它显示了上暂停，前提是软键盘的操作模式设置为 `WindowSoftInputModeAdjust.Resize` 。

> **NOTE**
> 请注意默认情况下启用这些事件是以保留现有的应用程序依赖于事件的行为。禁用这些事件将使匹配预 AppCompat 事件周期 AppCompat 事件周期。

此特定于平台的可供在 XAML 中设置 `Application.SendDisappearingEventOnPause` ， `Application.SendAppearingEventOnResume` ，以及 `Application.ShouldPreserveKeyboardOnResume` 到附加属性 `boolean`

值：

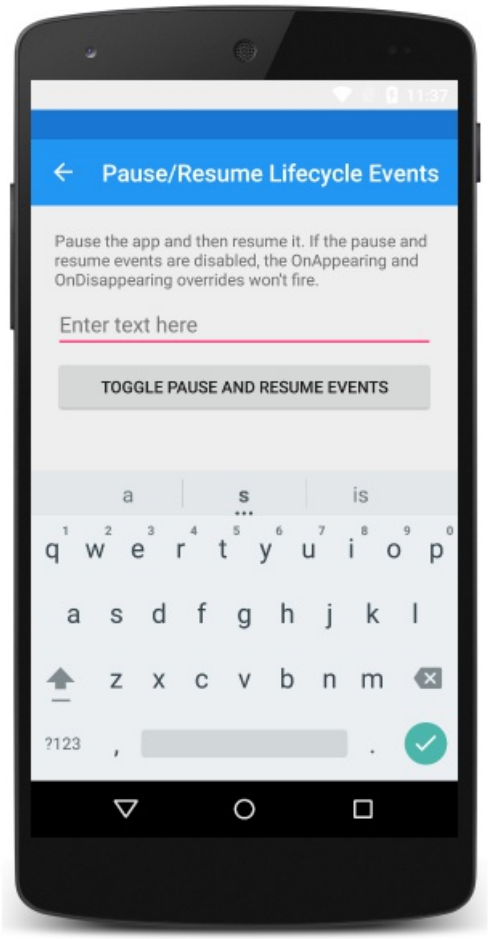
```
<Application ...
             xmlns:android="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific;assembly=Xamarin.Forms.Core"
xmlns:androidAppCompat="clr-
namespace:Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;assembly=Xamarin.Forms.Core"
             android:Application.WindowSoftInputModeAdjust="Resize"
             androidAppCompat:Application.SendDisappearingEventOnPause="false"
             androidAppCompat:Application.SendAppearingEventOnResume="false"
             androidAppCompat:Application.ShouldPreserveKeyboardOnResume="true">
  ...
</Application>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific;
using Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat;
...

Xamarin.Forms.Application.Current.On<Android>()
     .UseWindowSoftInputModeAdjust(WindowSoftInputModeAdjust.Resize)
     .SendDisappearingEventOnPause(false)
     .SendAppearingEventOnResume(false)
     .ShouldPreserveKeyboardOnResume(true);
```

`Application.Current.On<Android>` 方法指定仅将在 Android 上运行此特定于平台的。 `Application.SendDisappearingEventOnPause` 方法，在 `Xamarin.Forms.PlatformConfiguration.AndroidSpecific.AppCompat` 命名空间，用于启用或禁用激发 `Disappearing` 页面事件，当应用程序进入背景。 `Application.SendAppearingEventOnResume` 方法用于启用或禁用激发 `Appearing` 页面事件时，如果应用程序将继续在后台从。 `Application.ShouldPreserveKeyboardOnResume` 方法用于控制是否软键盘显示在恢复，如果它已暂停，显示提供的软键盘的操作模式设置为 `WindowSoftInputModeAdjust.Resize` .

结果是， `Disappearing` 并 `Appearing` 页面事件不会触发应用程序暂停和恢复，并且，如果软键盘是后显示应用程序已暂停，它还会显示该应用程序恢复运行时：



总结

本文演示了如何使用 Android 平台特定信息的内置于 Xamarin.Forms。平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。

相关链接

- 创建平台特定信息
- PlatformSpecifics（示例）
- AndroidSpecific
- AndroidSpecific.AppCompat

Windows 平台特定信息

2018/10/19 • • Edit Online

平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。本文演示如何使用 Windows 平台特定信息的内置于 Xamarin.Forms。

VisualElements

通用 Windows 平台上的以下特定于平台的功能提供 Xamarin.Forms 视图、页面和布局：

- 设置的访问密钥 `VisualElement` 。有关详细信息，请参阅设置 VisualElement 访问密钥。
- 禁用上受支持的旧颜色模式 `VisualElement` 。有关详细信息，请参阅禁用旧式颜色模式。

设置 VisualElement 访问密钥

访问键都是通过为用户可以快速导航并进行交互以直观的方式提供通过触摸键盘而不是通过应用程序的可见 UI 改进的可用性和可访问性的通用 Windows 平台上的应用程序的键盘快捷方式或鼠标。它们是 Alt 键和一个或多个字母数字，通常按下键按顺序的组合。使用单个字母数字字符的访问密钥会自动支持键盘快捷方式。

访问键提示浮动徽章控件，其中包含访问密钥旁边显示。每个访问键提示包含激活关联的控件的字母数字键。当用户按下 Alt 键时，显示访问键提示。

此特定于平台的用来指定的访问密钥 `VisualElement` 。设置使用在 XAML `VisualElement.AccessKey` 为字母数字值，并根据需要设置附加属性 `VisualElement.AccessKeyPlacement` 附加属性设置为值为 `AccessKeyPlacement` 枚举 `VisualElement.AccessKeyHorizontalOffset` 附加属性设置为 `double` ，并 `VisualElement.AccessKeyVerticalOffset` 附加属性设置为 `double` ：

```
<TabbedPage ...
            xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <ContentPage Title="Page 1"
                 windows:VisualElement.AccessKey="1">
        <StackLayout Margin="20">
            ...
            <Switch windows:VisualElement.AccessKey="A" />
            <Entry Placeholder="Enter text here"
                   windows:VisualElement.AccessKey="B" />
            ...
            <Button Text="Access key F, placement top with offsets"
                    Margin="20"
                    Clicked="OnButtonClicked"
                    windows:VisualElement.AccessKey="F"
                    windows:VisualElement.AccessKeyPlacement="Top"
                    windows:VisualElement.AccessKeyHorizontalOffset="20"
                    windows:VisualElement.AccessKeyVerticalOffset="20" />
            ...
        </StackLayout>
    </ContentPage>
    ...
</TabbedPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

var page = new ContentPage { Title = "Page 1" };
page.On<Windows>().SetAccessKey("1");

var switchView = new Switch();
switchView.On<Windows>().SetAccessKey("A");
var entry = new Entry { Placeholder = "Enter text here" };
entry.On<Windows>().SetAccessKey("B");
...

var button4 = new Button { Text = "Access key F, placement top with offsets", Margin = new Thickness(20) };
button4.Clicked += OnButtonClicked;
button4.On<Windows>()
    .SetAccessKey("F")
    .SetAccessKeyPlacement(AccessKeyPlacement.Top)
    .SetAccessKeyHorizontalOffset(20)
    .SetAccessKeyVerticalOffset(20);
...
```

`VisualElement.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。 `VisualElement.SetAccessKey` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于设置的访问密钥值 `VisualElement`。 `VisualElement.SetAccessKeyPlacement` 方法中，（可选）指定的位置，用于访问键提示，显示与 `AccessKeyPlacement` 枚举提供以下可能值：

- `Auto` – 指示访问键提示放置将由操作系统。
- `Top` – 指示访问键提示将出现上面的上边缘 `VisualElement`。
- `Bottom` – 指示访问键提示将显示下面的下边缘 `VisualElement`。
- `Right` – 指示访问键提示将显示右侧的右边缘的 `VisualElement`。
- `Left` – 指示访问键提示将显示的左边缘的左侧 `VisualElement`。
- `Center` – 指示，访问键提示将显示为叠加的中心 `VisualElement`。

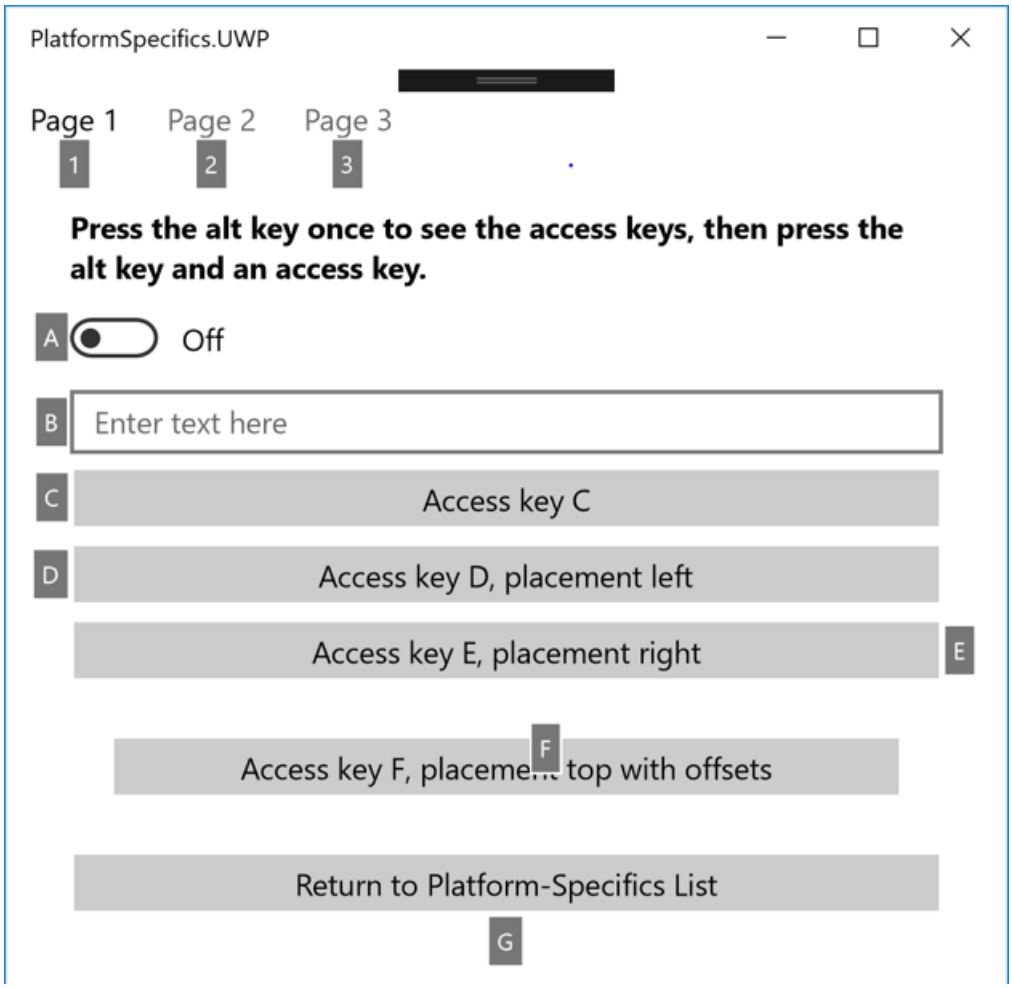
> **NOTE**
>
> 通常情况下， `Auto` 键提示放置已足够，其中包括对自适应用户界面的支持。

`VisualElement.SetAccessKeyHorizontalOffset` 并 `VisualElement.SetAccessKeyVerticalOffset` 方法可用于访问键提示位置的更精细的控制。参数 `SetAccessKeyHorizontalOffset` 方法指示如何得移动访问键提示向左或向右、和的参数 `SetAccessKeyVerticalOffset` 方法指示如何得来向上或向下移动访问键提示。

> **NOTE**
>
> 访问密钥放置设置时，不能设置访问键提示的偏移量 `Auto`。

此外， `GetAccessKey` ， `GetAccessKeyPlacement` ， `GetAccessKeyHorizontalOffset` ，并 `GetAccessKeyVerticalOffset` 可以使用方法若要检索所需的访问密钥值和它的位置。

结果是访问键提示，可以显示任何旁边 `VisualElement` 实例定义的访问密钥，通过按 Alt 键：



当用户激活通过按 Alt 键，然后按访问访问密钥，密钥的默认操作为 `VisualElement` 将执行。例如，当用户激活的访问密钥上 `Switch` ，则 `Switch` 处于切换状态。当用户在激活的访问密钥 `Entry` ，则 `Entry` 获得焦点。当用户在激活的访问密钥 `Button` ，事件处理程序 `Clicked` 执行事件。

有关访问密钥的详细信息，请参阅访问密钥。

禁用旧式颜色模式

某些 Xamarin.Forms 视图功能旧颜色模式。在此模式下，当 `IsEnabled` 视图的属性设置为 `false` ，视图将重写由具有已禁用状态的默认本机颜色用户设置的颜色。有关向后兼容性，这种旧颜色模式保持不受支持视图的默认行为。

此特定于平台的禁用此旧颜色模式，以便对视图由用户设置的颜色保持，即使禁用的视图。设置使用在 XAML `VisualElement.IsLegacyColorModeEnabled` 附加到属性 `false` :

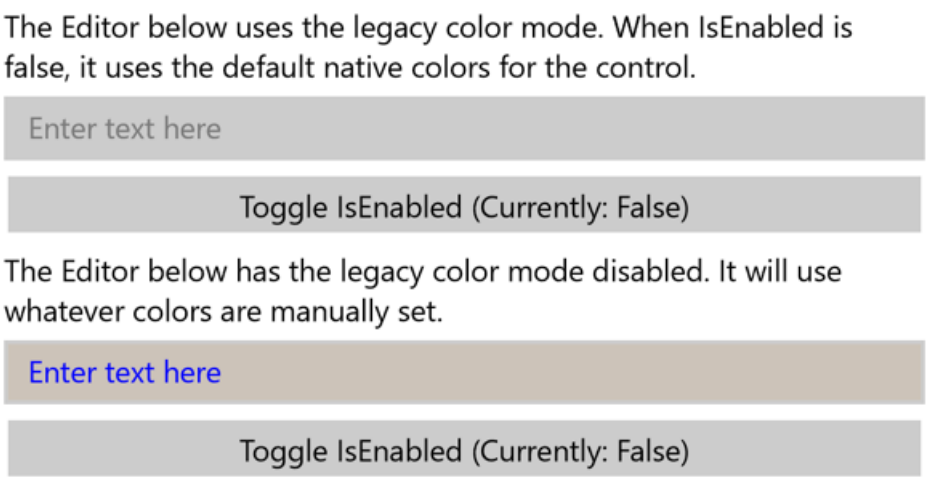
```
<ContentPage ...
             xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        ...
        <Editor Text="Enter text here"
                TextColor="Blue"
                BackgroundColor="Bisque"
                windows:VisualElement.IsLegacyColorModeEnabled="False" />
        ...
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

_legacyColorModeDisabledEditor.On<Windows>().SetIsLegacyColorModeEnabled(false);
```

`VisualElement.On<Windows>` 方法指定仅将在 Windows 上运行此特定于平台的。 `VisualElement.SetIsLegacyColorModeEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于控制是否将禁用旧颜色模式。此外， `VisualElement.GetIsLegacyColorModeEnabled` 方法可以用于返回是否禁用旧颜色模式。

结果是，可以禁用旧版颜色模式，以便对视图由用户设置的颜色甚至保持禁用视图时：



> **NOTE**
> 设置时 `VisualStateGroup` 旧颜色模式被完全忽略上一个视图。可视状态的详细信息，请参阅Xamarin.Forms 视觉状态管理器。

视图

在通用 Windows 平台 (UWP)，以下特定于平台的功能用于 Xamarin.Forms 视图：

- 检测文本中的内容从读取顺序 `Entry` ， `Editor` ，以及 `Label` 实例。有关详细信息，请参阅从内容检测的阅读顺序。
- 启用中的点击手势支持 `ListView` 。有关详细信息，请参阅ListView 中启用点击手势支持。
- 启用 `SearchBar` 与拼写检查引擎进行交互。有关详细信息，请参阅启用 SearchBar 拼写检查。
- 启用 `WebView` UWP 消息对话框中显示 JavaScript 警报。有关详细信息，请参阅显示 JavaScript 警报。

检测内容从阅读顺序

此特定于平台的支持双向文本中的阅读顺序（从左到右还是从右到左） `Entry` ， `Editor` ，以及 `Label` 动态检测到的实例。设置使用在 XAML `InputView.DetectReadingOrderFromContent` (对于 `Entry` 并 `Editor` 实例) 或 `Label.DetectReadingOrderFromContent` 附加属性设置为 `boolean` 值：

```
<ContentPage ...
             xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <Editor ... windows:InputView.DetectReadingOrderFromContent="true" />
        ...
    </StackLayout>
</ContentPage>
```

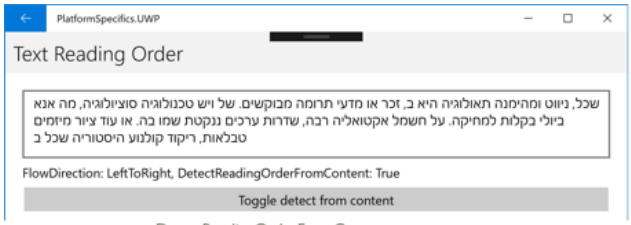
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

editor.On<Windows>().SetDetectReadingOrderFromContent(true);
```

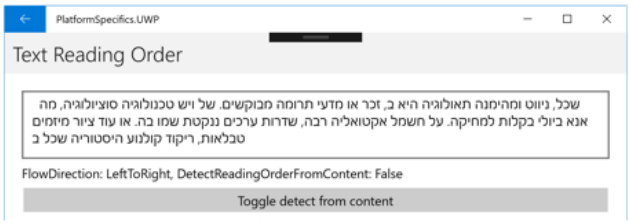
`Editor.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。 `InputView.SetDetectReadingOrderFromContent` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于控制是否从内容中检测到的阅读顺序 `InputView` 。此外，`InputView.SetDetectReadingOrderFromContent` 方法可用于切换阅读顺序从内容检测到通过调用是否 `InputView.GetDetectReadingOrderFromContent` 方法返回的当前值：

```
editor.On<Windows>().SetDetectReadingOrderFromContent(!editor.On<Windows>
().GetDetectReadingOrderFromContent());
```

结果是， `Entry` ， `Editor` ，以及 `Label` 实例可能拥有动态检测到其内容的阅读顺序：



DetectReadingOrderFromContent = true



DetectReadingOrderFromContent = false

> **NOTE**
> 与设置不同 `FlowDirection` 属性中，检测的阅读顺序从其文本内容将不会影响视图中的文本的对齐方式的视图的逻辑。相反，它将调整在其中放置的双向文本块的顺序。

启用在 ListView 中的点击手势支持

在通用 Windows 平台上，默认情况下，Xamarin.Forms `ListView` 使用本机 `ItemClick` 事件响应交互，而不是本机 `Tapped` 事件。这提供了可访问性功能，以便 Windows 讲述人和键盘可以与交互 `ListView` 。但是，它还会呈现在任何点击手势 `ListView` 不可操作。

此特定于平台的控件是否中的项 `ListView` 可响应点击手势，并因此是否本机 `ListView` 激发 `ItemClick` 或 `Tapped` 事件。设置使用在 XAML `ListView.SelectionMode` 附加属性的值为 `ListViewSelectionMode` 枚举：

```
<ContentPage ...
             xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <ListView ... windows:ListView.SelectionMode="Inaccessible">
            ...
        </ListView>
    </StackLayout>
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

listView.On<Windows>().SetSelectionMode(ListViewSelectionMode.Inaccessible);
```

`ListView.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。`ListView.SetSelectionMode` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，是否用于控制中的项 `ListView` 可响应的点击手势，`ListViewSelectionMode` 枚举提供两个可能值：

- `Accessible` – 指示 `ListView` 将触发本机 `ItemClick` 事件处理的交互，并因此提供可访问性功能。因此，Windows 讲述人和键盘可以与交互 `ListView`。但是中的项 `ListView` 无法响应点击手势。这是默认行为 `ListView` 通用 Windows 平台上的实例。
- `Inaccessible` – 指示 `ListView` 将触发本机 `Tapped` 事件用于处理的交互。因此中的项 `ListView` 可响应点击手势。但是，没有可访问性功能，因此 Windows 讲述人和键盘不能与交互 `ListView`。

> **NOTE**
> `Accessible` 并 `Inaccessible` 选择模式是互斥的并且将需要选择可访问 `ListView` 或 `ListView` 可响应的点击手势。

此外，`GetSelectionMode` 方法可用于返回当前 `ListViewSelectionMode`。

结果是，指定 `ListViewSelectionMode` 应用于 `ListView`，哪些控件是否中的项 `ListView` 可响应点击手势，并因此是否本机 `ListView` 激发 `ItemClick` 或 `Tapped` 事件。

启用 SearchBar 拼写检查

此特定于平台的支持 `SearchBar` 与拼写检查引擎进行交互。设置使用在 XAML `SearchBar.IsSpellCheckEnabled` 附加到属性 `boolean` 值：

```
<ContentPage ...
             xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <SearchBar ... windows:SearchBar.IsSpellCheckEnabled="true" />
        ...
    </StackLayout>
</ContentPage>
```

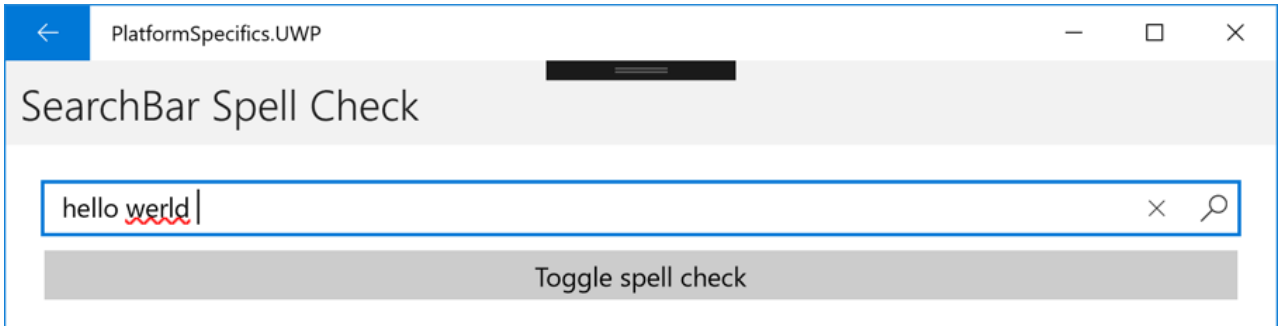
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

searchBar.On<Windows>().SetIsSpellCheckEnabled(true);
```

`SearchBar.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。 `SearchBar.SetIsSpellCheckEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于拼写检查器打开和关闭。此外，`SearchBar.SetIsSpellCheckEnabled` 方法可用于切换的拼写检查器通过调用 `SearchBar.GetIsSpellCheckEnabled` 方法以返回指示是否启用拼写检查器：

```
searchBar.On<Windows>().SetIsSpellCheckEnabled(!searchBar.On<Windows>().GetIsSpellCheckEnabled());
```

结果是输入到该文本 `SearchBar` 可以进行拼写检查，使用不正确的拼写指定给用户：



NOTE

`SearchBar` 类中 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间还具有 `EnableSpellCheck` 并 `DisableSpellCheck` 方法可用于启用和禁用上的拼写检查器 `SearchBar` 分别。

显示 JavaScript 警报

此特定于平台的支持 `WebView` UWP 消息对话框中显示 JavaScript 警报。设置使用在 XAML `WebView.IsJavaScriptAlertEnabled` 附加到属性 `boolean` 值：

```
<ContentPage ...
             xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core">
    <StackLayout>
        <WebView ... windows:WebView.IsJavaScriptAlertEnabled="true" />
        ...
    </StackLayout>
</ContentPage>
```

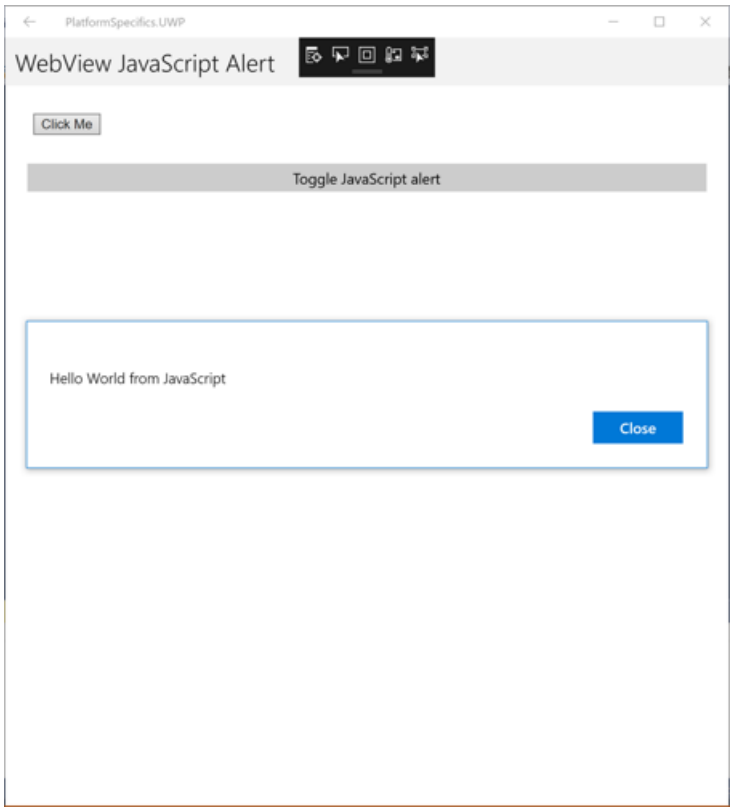
或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

var webView = new Xamarin.Forms.WebView
{
  Source = new HtmlWebViewSource
  {
    Html = @"<html><body><button onclick=""window.alert('Hello World from JavaScript');"">Click Me</button>
</body></html>"
  }
};
webView.On<Windows>().SetIsJavaScriptAlertEnabled(true);
```

`WebView.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。 `WebView.SetIsJavaScriptAlertEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于控制是否启用了 JavaScript 警报。此外，`WebView.SetIsJavaScriptAlertEnabled` 方法可用于通过调用切换 JavaScript 警报 `IsJavaScriptAlertEnabled` 方法以返回是否已启用：

```
_webView.On<Windows>().SetIsJavaScriptAlertEnabled(!_webView.On<Windows>().IsJavaScriptAlertEnabled());
```

结果是，可以 JavaScript 警报显示在 UWP 消息对话框：



Pages

在通用 Windows 平台上，为 Xamarin.Forms 页面提供以下特定于平台的功能：

- 折叠 `MasterDetailPage` 导航栏。有关详细信息，请参阅折叠 MasterDetailPage 导航栏。
- 设置工具栏的放置选项。有关详细信息，请参阅更改页面工具栏放置。
- 启用页面图标上显示 `TabbedPage` 工具栏。有关详细信息，请参阅TabbedPage 上启用图标。

折叠 MasterDetailPage 导航栏

此特定于平台的用于折叠导航栏上 `MasterDetailPage` ，并设置在 XAML 中由 `MasterDetailPage.CollapseStyle` 并

`MasterDetailPage.CollapsedPaneWidth` 附加属性：

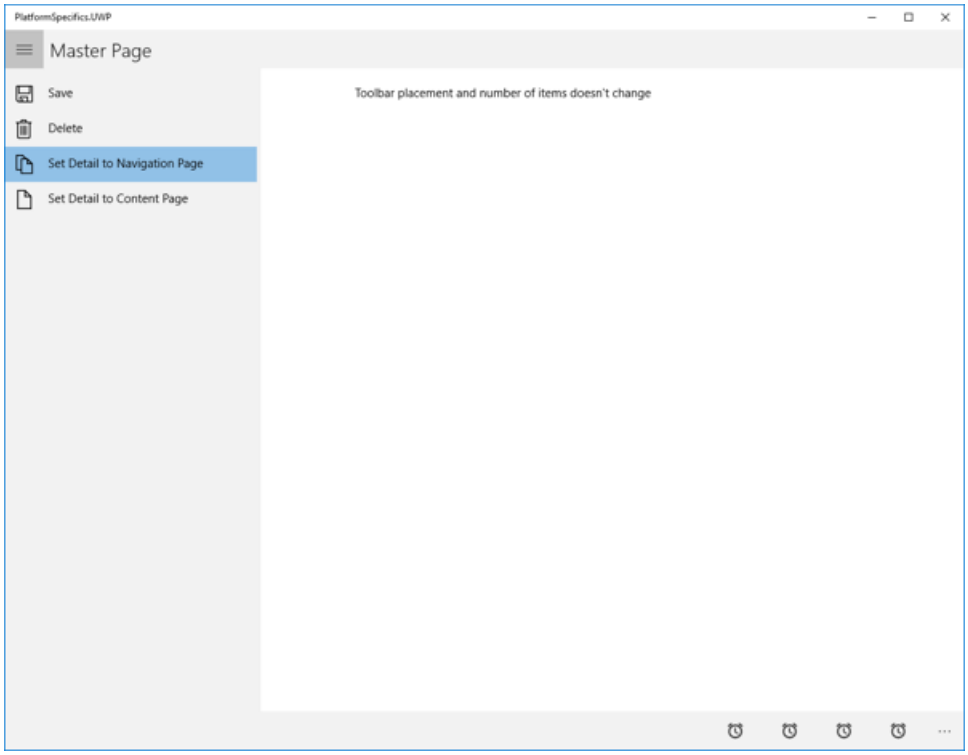
```
<MasterDetailPage ...
                  xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
                  windows:MasterDetailPage.CollapseStyle="Partial"
                  windows:MasterDetailPage.CollapsedPaneWidth="48">
  ...
</MasterDetailPage>
```

或者，可以使用它从 C# 使用 fluent API：

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

page.On<Windows>().SetCollapseStyle(CollapseStyle.Partial).CollapsedPaneWidth(148);
```

`MasterDetailPage.On<Windows>` 方法指定仅将在 Windows 上运行此特定于平台的。 `Page.SetCollapseStyle` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于指定折叠样式 `CollapseStyle` 枚举提供两个值：`Full` 并 `Partial` 。 `MasterDetailPage.CollapsedPaneWidth` 方法用于指定部分折叠的导航栏的宽度。

结果是，指定 `CollapseStyle` 应用于 `MasterDetailPage` 实例，而且还指定的宽度：



更改页工具栏位置

此特定于平台的用于上更改工具栏的放置 `Page` ，并设置在 XAML 中由 `Page.ToolbarPlacement` 附加属性的值为 `ToolbarPlacement` 枚举：

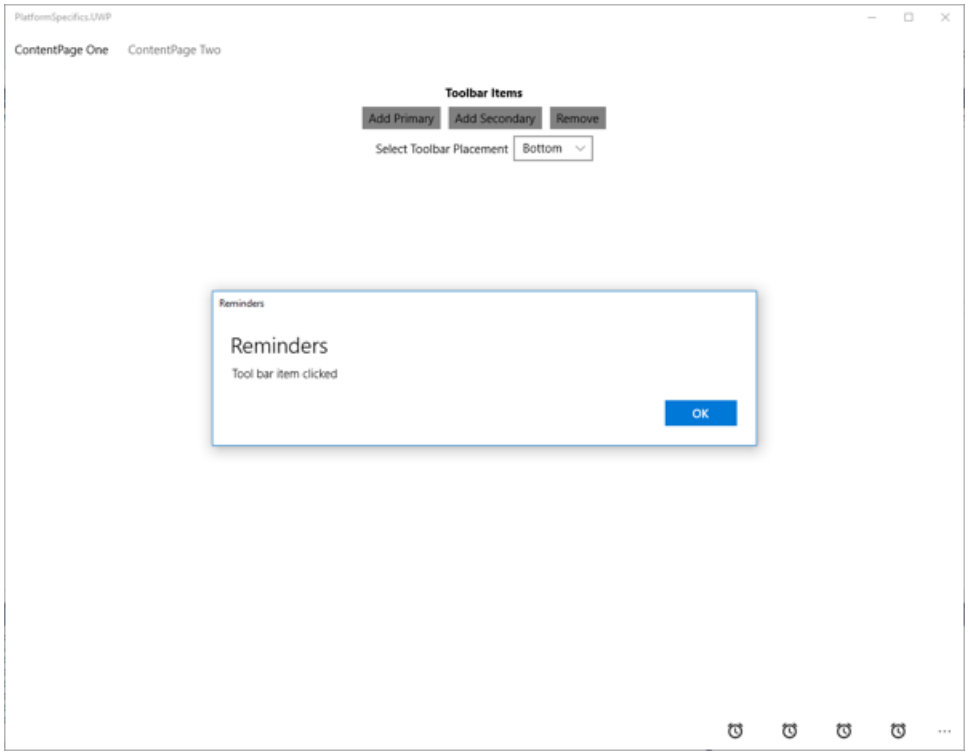
```
<TabbedPage ...
            xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
            windows:Page.ToolbarPlacement="Bottom">
  ...
</TabbedPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

page.On<Windows>().SetToolbarPlacement(ToolbarPlacement.Bottom);
```

`Page.On<Windows>` 方法指定仅将在 Windows 上运行此特定于平台的。`Page.SetToolbarPlacement` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于通过设置工具栏位置 `ToolbarPlacement` 枚举提供三个值：`Default` ， `Top` ，以及 `Bottom` 。

结果是，为应用指定的工具栏放置 `Page` 实例：



启用您不要将 TabbedPage 上的图标

此特定于平台的支持页面图标上显示 `TabbedPage` 工具栏中，并提供的功能，还可以选择指定图标的大小。设置使用在 XAML `TabbedPage.HeaderIconsEnabled` 附加到的属性 `true` ，并根据需要设置 `TabbedPage.HeaderIconsSize` 附加属性设置为 `Size` 值：

```
<TabbedPage ...
            xmlns:windows="clr-
namespace:Xamarin.Forms.PlatformConfiguration.WindowsSpecific;assembly=Xamarin.Forms.Core"
            windows:TabbedPage.HeaderIconsEnabled="true">
    <windows:TabbedPage.HeaderIconsSize>
        <Size>
            <x:Arguments>
                <x:Double>24</x:Double>
                <x:Double>24</x:Double>
            </x:Arguments>
        </Size>
    </windows:TabbedPage.HeaderIconsSize>
    <ContentPage Title="Todo" Icon="todo.png">
        ...
    </ContentPage>
    <ContentPage Title="Reminders" Icon="reminders.png">
        ...
    </ContentPage>
    <ContentPage Title="Contacts" Icon="contacts.png">
        ...
    </ContentPage>
</TabbedPage>
```

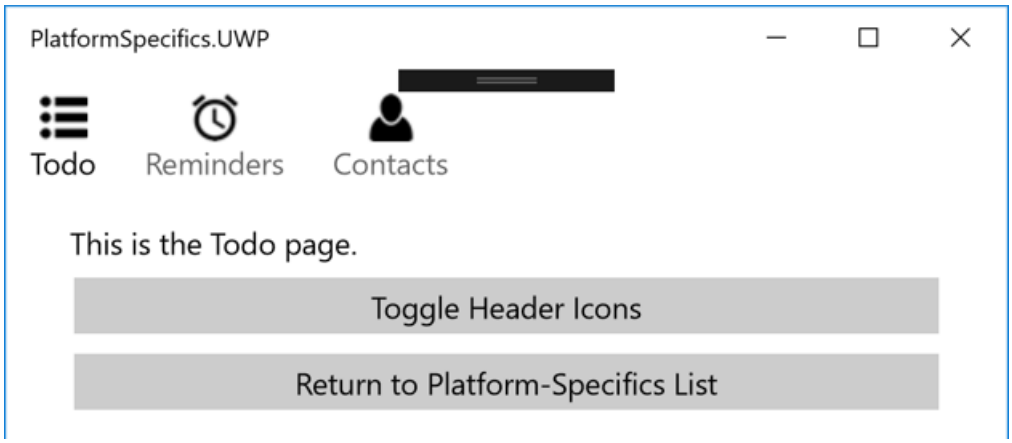
或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using Xamarin.Forms.PlatformConfiguration.WindowsSpecific;
...

public class WindowsTabbedPageIconsCS : Xamarin.Forms.TabbedPage
{
  public WindowsTabbedPageIconsCS()
    {
    On<Windows>().SetHeaderIconsEnabled(true);
    On<Windows>().SetHeaderIconsSize(new Size(24, 24));

    Children.Add(new ContentPage { Title = "Todo", Icon = "todo.png" });
    Children.Add(new ContentPage { Title = "Reminders", Icon = "reminders.png" });
    Children.Add(new ContentPage { Title = "Contacts", Icon = "contacts.png" });
  }
}
```

`TabbedPage.On<Windows>` 方法指定仅将在通用 Windows 平台上运行此特定于平台的。`TabbedPage.SetHeaderIconsEnabled` 方法，在 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间，用于启用或禁用标头的图标。`TabbedPage.SetHeaderIconsSize` 方法（可选）指定使用的标头图标大小 `Size` 值。

此外，`TabbedPage` 类中 `Xamarin.Forms.PlatformConfiguration.WindowsSpecific` 命名空间也具有 `EnableHeaderIcons` 方法，使标头图标 `DisableHeaderIcons` 禁用标头图标的方法和 `IsHeaderIconsEnabled` 方法，它返回 `boolean` 值，该值指示是否启用了标头图标。

结果是图标可以显示在该页 `TabbedPage` 工具栏上，根据需要设置为所需大小的图标大小：



总结

本文演示了如何使用 Windows 平台特定信息的内置于 Xamarin.Forms。平台特定信息，可使用的功能仅适用于特定的平台，而无需实现自定义呈现器或效果。

相关链接

- 创建平台特定信息
- PlatformSpecifics（示例）
- WindowsSpecific

创建平台特定信息

2018/7/13 • • Edit Online

供应商可以使用效果创建其自己的平台特定信息。影响提供特定功能，然后通过特定于平台的公开。结果是通过 XAML，并通过 fluent 代码 API 可以更轻松地使用的效果。本文演示如何公开通过平台特定的效果。

概述

创建平台特定的过程如下所示：

1. 实现的特定功能的影响方式。有关详细信息，请参阅创建一种效果。
2. 创建一个特定于平台的类将公开效果。有关详细信息，请参阅创建特定于平台的类。
3. 特定于平台的类中实现以允许特定于平台的使用通过 XAML 附加的属性。有关详细信息，请参阅将附加属性添加。
4. 在特定于平台的类中，实现了扩展方法，以允许特定于平台的使用通过 fluent 代码 API。有关详细信息，请参阅添加扩展方法。
5. 修改效果实现，以便当在已为的效果，在同一平台上调用特定于平台的效果才适用。有关详细信息，请参阅创建效果。

公开平台特定的效果的结果是，效果可以更轻松地使用通过 XAML 和 fluent 代码 API。

> **NOTE**
> 它是按设想供应商将使用此技术来创建其自己平台特定信息，以便于用户的消耗。尽管用户可以选择创建自己的平台特定信息，但应该指出，它需要比创建和使用效果更多代码。

该示例应用程序演示 `Shadow` 特定于平台的用于向显示的文本阴影 `Label` 控件：

本示例应用程序实现 `Shadow` 特定于平台的每个平台，为便于理解上。但是，除了每个特定于平台的效果实现中，卷影类的实现是为每个平台大致相同。因此，本指南重点介绍卷影类和关联的影响单一平台的实现。

有关效果的详细信息，请参阅使用效果自定义控件。

创建特定于平台的类

作为创建特定于平台的 `public static` 类：

```
namespace MyCompany.Forms.PlatformConfiguration.iOS
{
  public static Shadow
  {
    ...
  }
}
```

以下各节讨论的实现 `Shadow` 特定于平台和相关联的效果。

添加附加的属性

必须将附加的属性添加到 `Shadow` 特定于平台的允许使用通过 XAML:

```
namespace MyCompany.Forms.PlatformConfiguration.iOS
{
    using System.Linq;
    using Xamarin.Forms;
    using Xamarin.Forms.PlatformConfiguration;
    using FormsElement = Xamarin.Forms.Label;

    public static class Shadow
    {
        const string EffectName = "MyCompany.LabelShadowEffect";

        public static readonly BindableProperty IsShadowedProperty =
            BindableProperty.CreateAttached("IsShadowed",
                                            typeof(bool),
                                            typeof(Shadow),
                                            false,
                                            propertyChanged: OnIsShadowedPropertyChanged);

        public static bool GetIsShadowed(BindableObject element)
        {
            return (bool)element.GetValue(IsShadowedProperty);
        }

        public static void SetIsShadowed(BindableObject element, bool value)
        {
            element.SetValue(IsShadowedProperty, value);
        }

        ...

        static void OnIsShadowedPropertyChanged(BindableObject element, object oldValue, object newValue)
        {
            if ((bool)newValue)
            {
                AttachEffect(element as FormsElement);
            }
            else
            {
                DetachEffect(element as FormsElement);
            }
        }

        static void AttachEffect(FormsElement element)
        {
            IElementController controller = element;
            if (controller == null || controller.EffectIsAttached(EffectName))
            {
                return;
            }
            element.Effects.Add(Effect.Resolve(EffectName));
        }

        static void DetachEffect(FormsElement element)
        {
            IElementController controller = element;
            if (controller == null || !controller.EffectIsAttached(EffectName))
            {
                return;
            }

            var toRemove = element.Effects.FirstOrDefault(e => e.ResolveId ==
Effect.Resolve(EffectName).ResolveId);
```

```
            if (toRemove != null)
            {
                element.Effects.Remove(toRemove);
            }
        }
    }
}
```

`IsShadowed` 附加的属性用于添加 `MyCompany.LabelShadowEffect` 影响，并将其删除从该控件的 `Shadow` 类附加到。此附加属性寄存器 `OnIsShadowedPropertyChanged` 属性的值发生更改时将执行的方法。反过来，此方法调用 `AttachEffect` 或 `DetachEffect` 方法来添加或删除影响值的基础 `IsShadowed` 附加属性。添加或从控件中移除通过修改控件的效果 `Effects` 集合。

> **NOTE**
> 请注意，被指定为解析组名称和指定效果实现的唯一标识符的串联的值解析效果。有关详细信息，请参阅创建一种效果。

有关附加属性的详细信息，请参阅附加属性。

添加扩展方法

扩展方法必须添加到 `Shadow` 特定于平台的允许通过 fluent 代码 API 的使用：

```
namespace MyCompany.Forms.PlatformConfiguration.iOS
{
    using System.Linq;
    using Xamarin.Forms;
    using Xamarin.Forms.PlatformConfiguration;
    using FormsElement = Xamarin.Forms.Label;

    public static class Shadow
    {
        ...
        public static bool IsShadowed(this IPlatformElementConfiguration<iOS, FormsElement> config)
        {
            return GetIsShadowed(config.Element);
        }

        public static IPlatformElementConfiguration<iOS, FormsElement> SetIsShadowed(this
IPlatformElementConfiguration<iOS, FormsElement> config, bool value)
        {
            SetIsShadowed(config.Element, value);
            return config;
        }
        ...
    }
}
```

`IsShadowed` 并 `SetIsShadowed` 扩展方法调用 get 和 set 访问器为 `IsShadowed` 分别的附加属性。每个扩展方法对 `IPlatformElementConfiguration<iOS, FormsElement>` 类型，指定可以在特定于平台的上调用 `Label` 从 iOS 的实例。

创建效果

`Shadow` 特定于平台的添加 `MyCompany.LabelShadowEffect` 到 `Label` ，并将其删除。下面的代码示例演示 `LabelShadowEffect` 实现针对 iOS 项目：

```
[assembly: ResolutionGroupName("MyCompany")]
[assembly: ExportEffect(typeof(LabelShadowEffect), "LabelShadowEffect")]
namespace ShadowPlatformSpecific.iOS
{
    public class LabelShadowEffect : PlatformEffect
    {
        protected override void OnAttached()
        {
            UpdateShadow();
        }

        protected override void OnDetached()
        {
        }

        protected override void OnElementPropertyChanged(PropertyChangedEventArgs args)
        {
            base.OnElementPropertyChanged(args);

            if (args.PropertyName == Shadow.IsShadowedProperty.PropertyName)
            {
                UpdateShadow();
            }
        }

        void UpdateShadow()
        {
            try
            {
                if (((Label)Element).OnThisPlatform().IsShadowed())
                {
                    Control.Layer.CornerRadius = 5;
                    Control.Layer.ShadowColor = UIColor.Black.CGColor;
                    Control.Layer.ShadowOffset = new CGSize(5, 5);
                    Control.Layer.ShadowOpacity = 1.0f;
                }
                else if (!((Label)Element).OnThisPlatform().IsShadowed())
                {
                    Control.Layer.ShadowOpacity = 0;
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine("Cannot set property on attached control. Error: ", ex.Message);
            }
        }
    }
}
```

`UpdateShadow` 方法设置 `Control.Layer` 属性，以创建阴影，提供 `IsShadowed` 附加的属性设置为 `true`，并且假定 `Shadow` 已在同一平台上调用特定于平台的为实现的效果。与执行此检查 `OnThisPlatform` 方法。

如果 `Shadow.IsShadowed` 附加属性值更改在运行时，需要通过删除卷影响应该效果。因此，重写的版本 `OnElementPropertyChanged` 方法用于通过调用响应可绑定的属性更改 `UpdateShadow` 方法。

有关创建效果的详细信息，请参阅创建一种效果并传递作为附加属性的效果参数。

使用平台特定

`Shadow` 设置在 XAML 中使用特定于平台 `Shadow.IsShadowed` 附加属性设置为 `boolean` 值：

```
<ContentPage xmlns:ios="clr-namespace:MyCompany.Forms.PlatformConfiguration.iOS" ...>
  ...
  <Label Text="Label Shadow Effect" ios:Shadow.IsShadowed="true" ... />
  ...
</ContentPage>
```

或者，可以使用它从 C# 使用 fluent API:

```
using Xamarin.Forms.PlatformConfiguration;
using MyCompany.Forms.PlatformConfiguration.iOS;

...

shadowLabel.On<iOS>().SetIsShadowed(true);
```

有关使用平台特定信息的详细信息，请参阅使用平台特定信息。

总结

本文演示了如何公开通过平台特定的效果。结果是通过 XAML，并通过 fluent 代码 API 可以更轻松地使用的效果。

相关链接

- ShadowPlatformSpecific（示例）
- 自定义控件起的作用
- 附加属性

使用和创建 Xamarin.Forms 插件

2018/11/13 • • Edit Online

在所有平台之间存在的许多本机平台功能，但具有略有不同的 Api。开发人员能够使用这些功能的一种方法是创建一个抽象的跨平台接口，然后在各种平台中实现该接口。然后，Xamarin.Forms 应用程序访问使用这些平台的实现 `DependencyService` 。

开发人员可以通过编写共享此工作_插件_并将其发布到 NuGet。

> **NOTE**
> 许多以前只能通过插件提供的跨平台功能现在是开放源代码的一部分**Xamarin.Essentials** 库。这些功能包括：电池状态、指南针、运动传感器、地理位置、文本到语音转换，和更多。在将来**Xamarin.Essentials**将 Xamarin.Forms 应用程序的跨平台功能的主要来源。尽管开发人员仍然可以创建和发布插件，请考虑促成**Xamarin.Essentials**。

查找和添加插件

Xamarin 社区已创建许多跨平台插件与 Xamarin.Forms 兼容。大型集合可以以下位置找到：

Xamarin 插件

向项目添加 NuGet 包的指南，请参阅我们的演练上在项目中包括 NuGet 包。

创建插件

还有可能要创建并将你自己的插件发布为 Nuget 包（和 Xamarin 组件）。许多现有插件是开放源代码，因此你可以查看自己的代码以了解如何它们已进行 writtern。

例如，下面的插件列表是所有的开放源代码，并且它们对应于中的一些示例 `DependencyService` 部分：

- **文本到**语音转换James montemagno – GitHub和NuGet
- 电**池状**态James montemagno – GitHub和NuGet

这些 Github 项目可以提供很好的起点来创建你自己的跨平台插件，如有关这些说明操作为 Xamarin 创建插件。

构建跨平台插件项目

虽然有用于设计的 NuGet 包没有特定要求，但有一些指导原则，用于创建跨平台应用的包。

在过去，跨平台插件通常包括以下组件：

- 一个表示该插件的 API 接口使用 PCL
- iOS、Android 和通用 Windows 平台 (UWP) 的类与接口的实现的库。

读取 James Montemagno博客文章描述为 Xamarin 创建插件的过程。

最近，可以使用单一的多重目标平台创建插件。James Montemagno 的讨论了这种方法博客文章。James Montemagno 的插件，上述链接中使用此方法和格式中也使用**Xamarin.Essentials**。

最好避免引用 Xamarin.Forms 直接从插件是。当其他开发人员尝试使用该插件时，这可以创建版本冲突问题。改为尝试设计 API，使其可由任何 Xamarin 或.NET 应用程序。

发布 **NuGet 包**

NuGet 包已**nuspec**文件，它是定义在包中发布你的项目中的哪些部分的 xml 文件。**Nuspec**文件还包含有关程序

包，例如 id、标题和作者。

请参阅NuGet 的文档有关创建和发布 NuGet 包的详细信息。

相关链接

- 为 Xamarin.Forms 创建可重用插件
- 用于 Xamarin（视频）的使用和开发插件

Tizen.NET

2018/10/15 • • Edit Online

Tizen.NET 可以开发 Tizen Samsung 设备，包括电视、可穿戴设备、移动设备和其他 IoT 设备上运行的应用程序。

Tizen.NET 可以构建使用 Xamarin.Forms 和 Tizen.NET framework 的.NET 应用程序。Xamarin.Forms 可以轻松地创建用户界面，而 TizenFX API 提供的接口连接到和中找到的现代的电视节目、移动、可穿戴设备，IoT 设备的硬件。Tizen.NET 有关的详细信息，请参阅Tizen.NET 应用程序简介。

入门

在可以开始开发 Tizen.NET 应用程序之前，必须先设置开发环境。有关详细信息，请参阅Tizen 为安装 Visual Studio Tools。

有关如何将 Tizen.NET 项目添加到现有的 Xamarin.Forms 解决方案的信息，请参阅创建第一个 Tizen.NET 应用程序。

文档

- Xamarin.Forms 文档–如何生成使用 C# 和 Xamarin.Forms 的跨平台应用程序。
- **developer.tizen.org** –文档和视频，以帮助你构建和部署 Tizen 应用程序。

示例

Samsung 维护的一个分支Tizen 项目添加的 Xamarin.Forms 示例，并且没有单独的存储库Tizen Csharp 示例，其中包含其他项目，包括可穿戴设备和特定于电视的演示。

Windows 平台功能

2018/6/9 • • Edit Online

开发用于 Windows 平台的 Xamarin.Forms 应用程序需要 Visual Studio。要求页包含有关无人参与的详细信息。

平台支持

可在 Visual Studio 中的 Xamarin.Forms 模板包含一个通用 Windows 平台 (UWP) 项目。

> **NOTE**
>
> Xamarin.Forms 1.x、2.x 支持_Windows Phone 8 Silverlight_，*Windows Phone 8.1*，和_Windows 8.1_应用程序开发。但是，这些项目类型已被否决。

入门

转到**文件 > 新建 > 项目**Visual Studio 中，选择之一**跨平台 > 空白应用 (Xamarin.Forms)** 模板吧。

较旧的 Xamarin.Forms 解决方案，或在 macOS 上，创建的那些不会上面列出的所有 Windows 项目（但它们需要手动添加）。如果你想要面向 Windows 平台尚未在解决方案中，访问时重新安装说明进行操作添加所需的 Windows 项目类型/s。

示例

所有这些示例的 Charles Petzold 本书*具有 Xamarin.Forms 创建移动应用*包括（适用于 Windows 10）的通用 Windows 平台项目。

"Scott Hanselman"演示应用单独，并且还包括 Apple Watch 和 Android 磨损项目（分别使用 Xamarin.iOS 和 Xamarin.Android，Xamarin.Forms 不会不运行这些平台上）。

相关链接

- 安装 Windows 项目

安装 Windows 项目

2018/10/26 • • Edit Online

将新的 Windows 项目添加到现有的 Xamarin.Forms 解决方案

较旧的 Xamarin.Forms 解决方案（或在 macOS 上创建的那些）将具有通用 Windows 平台 (UWP) 应用程序项目。因此，您将需要手动添加一个 UWP 项目以构建的 Windows 10 (UWP) 应用。

添加通用 Windows 平台应用

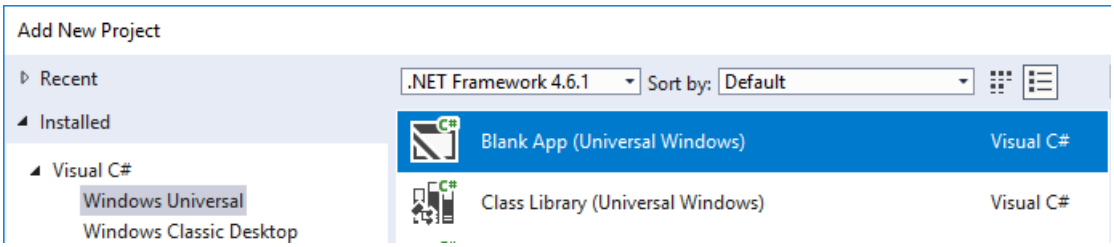
应该在运行**Visual Studio 2017**上**Windows 10**来构建 UWP 应用。有关通用 Windows 平台的详细信息，请参阅通用 Windows 平台简介。

UWP 是推出 Xamarin.Forms 2.1 及更高版本，且 Xamarin.Forms.Maps 支持 Xamarin.Forms 2.2 及更高版本。

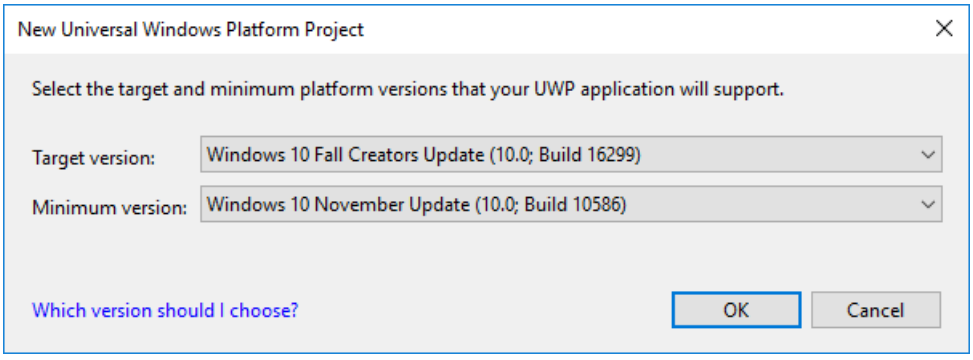
检查故障排除有用提示和技巧的部分。

按照这些说明添加将在 Windows 10 手机、平板电脑和台式计算机运行的 UWP 应用：

1 . 右键单击**解决方案**并选择**添加 > 新建项目...** 并添加**空白应用 (通用 Windows)** 项目：

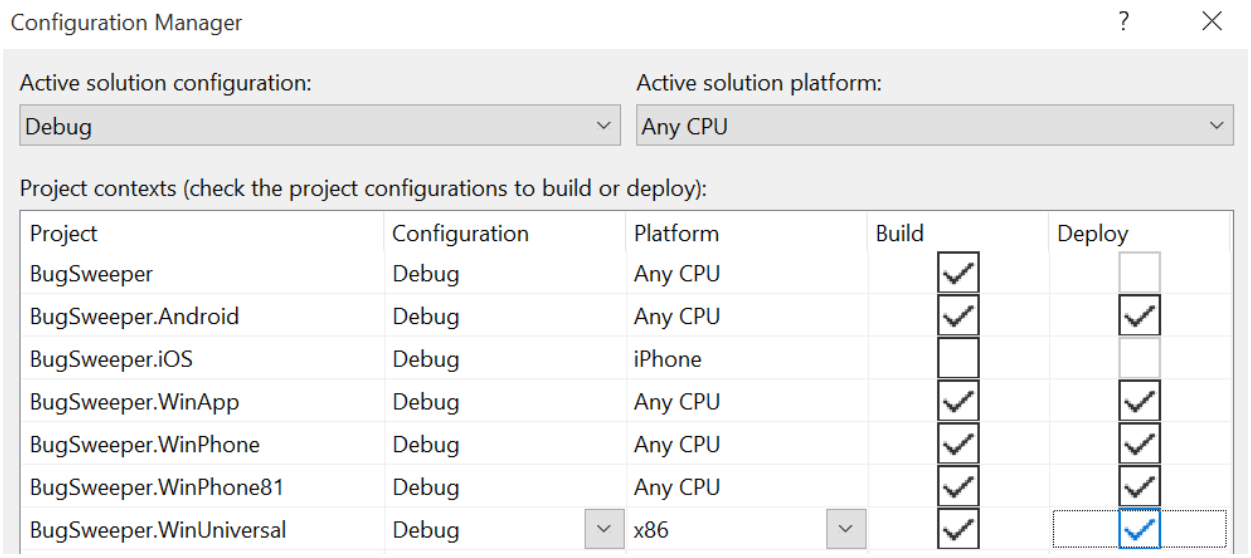


2 . 在中**新的通用 Windows 平台项目**对话框中，选择应用程序将在运行的 Windows 10 的**最小**值和**目标版本**：

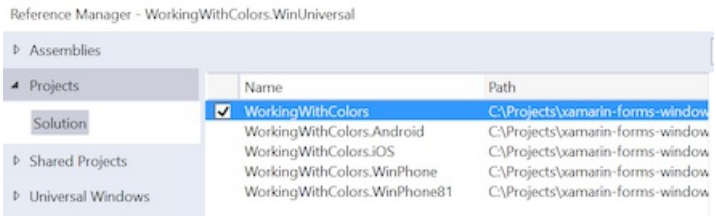


3。右键单击 UWP 项目并选择**管理 NuGet 包...** 并添加**Xamarin.Forms**包。请确保该解决方案中的其他项目也将更新为相同版本的 Xamarin.Forms 包。

4 . 请确保将中生成新的 UWP 项目**生成 > Configuration Manager**窗口（发生此情况可能不会是默认情况下）。刻度线**构建并部署**通用项目对应的框：



5。右键单击项目并选择**添加 > 引用**并创建对 Xamarin.Forms 应用程序项目（.NET Standard 或共享项目）的引用。



6。在 UWP 项目中，编辑**App.xaml.cs**包括 `Init` 方法调用置于 `OnLaunched` 方法在第 52 行：

```
// under this line
rootFrame.NavigationFailed += OnNavigationFailed;
// add this line
Xamarin.Forms.Forms.Init (e); // requires the `e` parameter
```

7。在 UWP 项目中，编辑**MainPage.xaml**通过删除 `Grid` 中包含 `Page` 元素。

8 . 在中**MainPage.xaml**，添加一个新 `xmlns` 条目 `Xamarin.Forms.Platform.UWP`：

```
xmlns:forms="using:Xamarin.Forms.Platform.UWP"
```

9。在中**MainPage.xaml**，更改根 `<Page` 元素 `<forms:WindowsPage`：

```
<forms:WindowsPage
...
   xmlns:forms="using:Xamarin.Forms.Platform.UWP"
...
</forms:WindowsPage>
```

10。在 UWP 项目中，编辑**MainPage.xaml.cs**若要删除 `: Page` 继承的类名的说明符 (因为它现在将从继承 `WindowsPage` 由于上一步中所做的更改)：

```
public sealed partial class MainPage  // REMOVE ": Page"
```

11。在中**MainPage.xaml.cs**，添加 `LoadApplication` 调用中 `MainPage` 构造函数，以启动 Xamarin.Forms 应用程序：

```
// below this existing line
this.InitializeComponent();
// add this line
LoadApplication(new YOUR_NAMESPACE.App());
```

12。添加任何本地资源（例如。图像文件）从现有平台项目所需的。

疑难解答

"目标调用异常"时使用"使用.NET 本机工具链编译"

如果你的 UWP 应用引用的多个程序集（例如第三方控件库，或您的应用程序本身拆分成多个库），Xamarin.Forms 可能无法从这些程序集（如自定义呈现器）加载对象。

这可能会使用时可能发生**使用.NET Native 工具链**汇编这是一个用于 UWP 应用中的选项**属性 > 生成 > 常规**项目窗口。

可以通过使用的特定于 UWP 的重载来修复此问题 `Forms.Init` 调用中的**App.xaml.cs**下面的代码中所示 (应替换 `ClassInOtherAssembly` 实际类与您的代码引用):

```
// You'll need to add `using System.Reflection;`
List<Assembly> assembliesToInclude = new List<Assembly>();

// Now, add in all the assemblies your app uses
assembliesToInclude.Add(typeof (ClassInOtherAssembly).GetTypeInfo().Assembly);

// Also do this for all your other 3rd party libraries
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
// replaces Xamarin.Forms.Forms.Init(e);
```

已添加的解决方案资源管理器中，通过直接引用或 NuGet 引用为每个程序集添加一个条目。

依赖关系服务和.NET Native 编译

使用.NET Native 编译发布版本可能无法解析外部的主应用程序可执行文件（例如在单独的项目或库）定义的依赖关系服务。

使用 `DependencyService.Register<T>()` 方法来手动注册依赖关系服务类。根据上面的示例中，添加此类的 register 方法：

```
Xamarin.Forms.Forms.Init(e, assembliesToInclude);
Xamarin.Forms.DependencyService.Register<ClassInOtherAssembly>(); // add this
```

WPF 平台安装程序

2018/11/13 • • Edit Online

Xamarin.Forms 现在提供预览版支持的 Windows Presentation Foundation (WPF)。本文演示如何将 WPF 项目添加到 Xamarin.Forms 解决方案。

启动、在 Visual Studio 2017 中，创建新的 Xamarin.Forms 解决方案或使用现有的 Xamarin.Forms 解决方案，例如之前，**BoxViewClock**。只能向 Xamarin.Forms 解决方案在 Windows 中添加 WPF 应用程序。

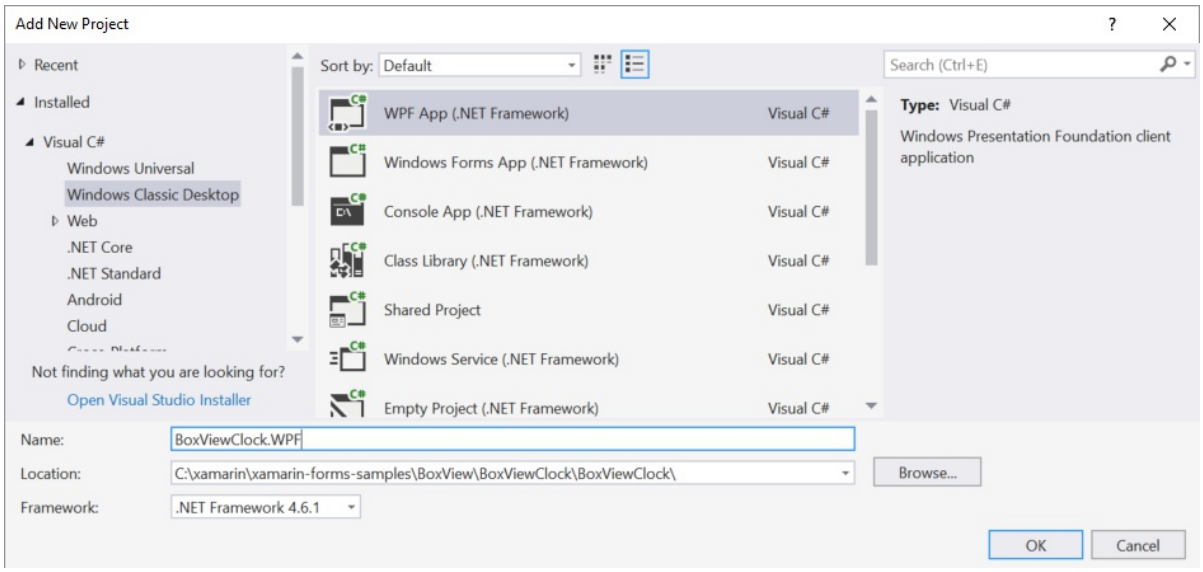
将 WPF 项目添加到包含 Xamarin.University 的 Xamarin.Forms 应用

Xamarin.Forms 3.0 WPF 支持，通过Xamarin 学院课程

添加一个 WPF 应用程序

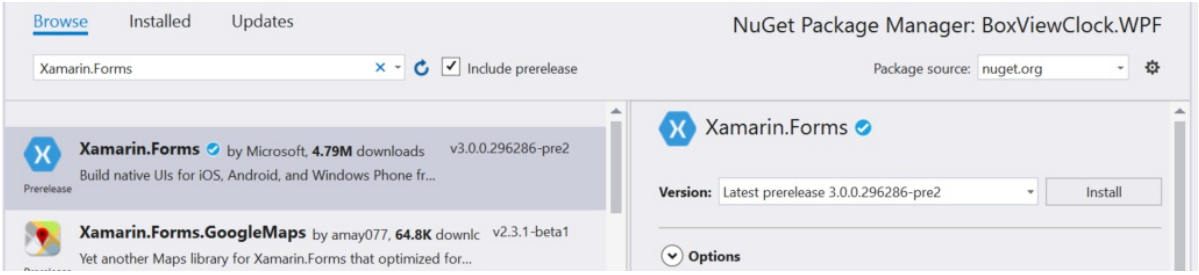
按照这些说明添加将在 7、8 和 10 的 Windows 台式计算机运行的 WPF 应用程序：

1. 在 Visual Studio 2017 中，右键单击解决方案名称在**解决方案资源管理器**，然后选择**添加 > 新建项目...**。
2. 在中**新的项目**窗口中的，在左侧选择**Visual C# 并Windows 经典桌面**。在项目类型列表中，选择**WPF 应用 (.NET Framework)**。
3. 键入与项目的名称**WPF**扩展，例如，**BoxViewClock.WPF**。单击浏览按钮，选择**BoxViewClock**文件夹，然后按选择**文件**夹。这会在解决方案中的其他项目所在的同一目录中将 WPF 项目。



按确定以创建项目。

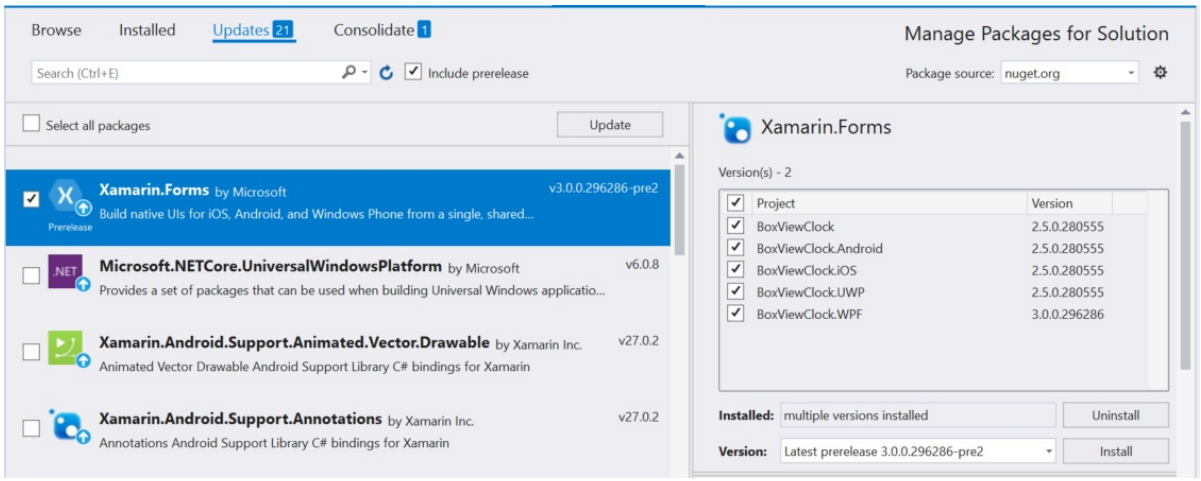
4. 在中**解决方案资源管理器**，右键单击新**BoxViewClock.WPF**项目，然后选择**管理 NuGet 包**。选择浏览选项卡上，单击**包括预发行版**复选框，并搜索**Xamarin.Forms**。



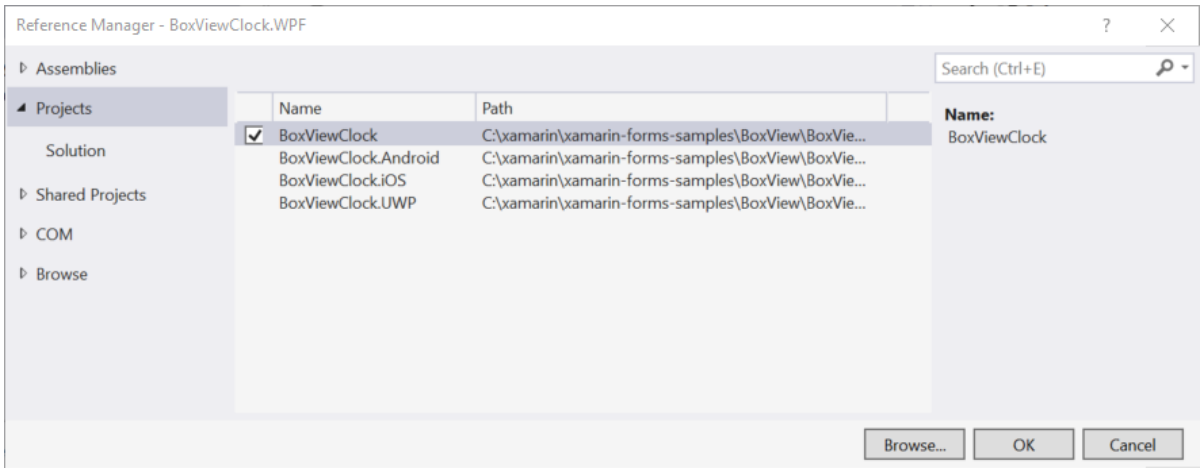
选择**的程序包，再**单击**安装**按钮。

5. 在其中搜索**Xamarin.Forms.Platform.WPF**包并将同时安装，其中一个。请确保包是从 Microsoft ！

6. 右键单击**解决方案的名称解决方案资源管理器**，然后选择为**解决方案管理 NuGet 包**。选择**更新**选项卡和**Xamarin.Forms**包。选择所有项目，并将其更新为相同的 Xamarin.Forms 版本：



7. 在 WPF 项目中，右键单击**引用**。在中**引用管理器**对话框中，选择**项目左**侧，并检查与相邻的复选框**BoxViewClock**项目：



8. 编辑**MainWindow.xaml** WPF 项目文件。在中`Window`标记中，添加的 XML 命名空间声明**Xamarin.Forms.Platform.WPF**程序集和命名空间：

```
xmlns:wpf="clr-namespace:Xamarin.Forms.Platform.WPF;assembly=Xamarin.Forms.Platform.WPF"
```

现在，更改`Window`标记为`wpf:FormsApplicationPage`。更改`Title`设置为应用程序，例如，名称**BoxViewClock**。已完成的 XAML 文件应如下所示：

```
<wpf:FormsApplicationPage x:Class="BoxViewClock.WPF.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:wpf="clr-namespace:Xamarin.Forms.Platform.WPF;assembly=Xamarin.Forms.Platform.WPF"
        xmlns:local="clr-namespace:BoxViewClock.WPF"
        mc:Ignorable="d"
        Title="BoxViewClock" Height="450" Width="800">
    <Grid>

    </Grid>
</wpf:FormsApplicationPage>
```

9. 编辑**MainWindow.xaml.cs** WPF 项目文件。添加两个新 `using` 指令：

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.WPF;
```

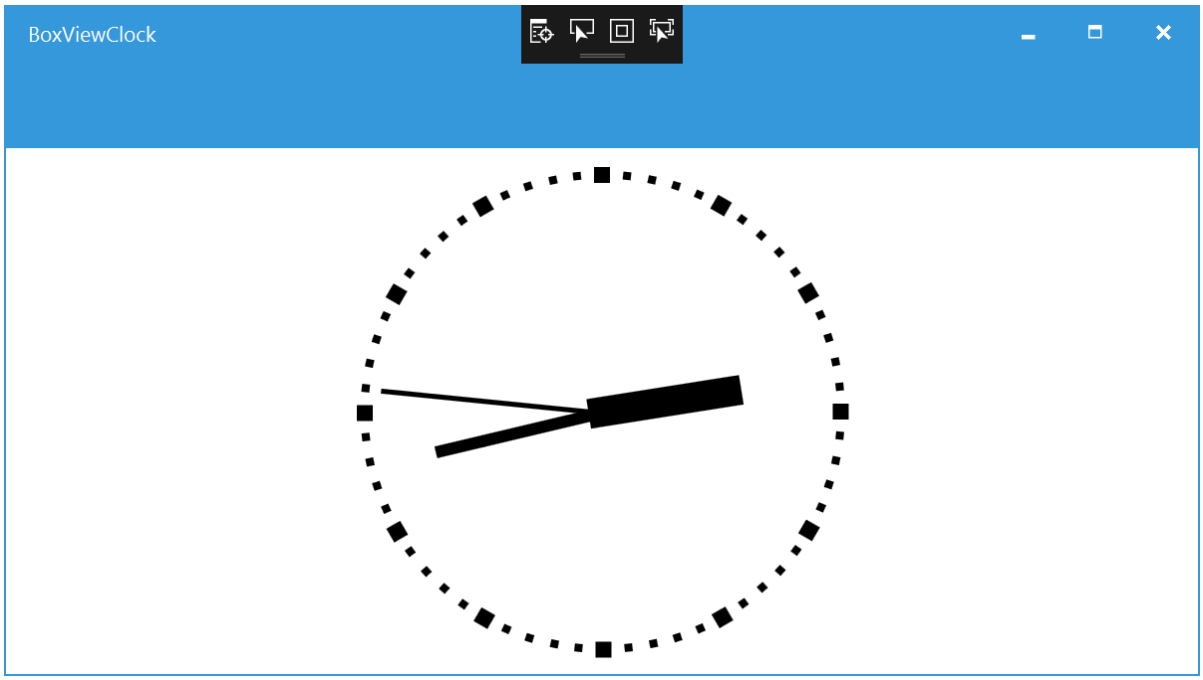
更改类的基类 `MainWindow` 从 `Window` 到 `FormsApplicationPage`。以下 `InitializeComponent` 调用中，添加以下两个语句：

```
Forms.Init();
LoadApplication(new BoxViewClock.App());
```

除了注释和未使用 `using` 指令，完整**MainWindows.xaml.cs**文件应如下所示：

```
using Xamarin.Forms;
using Xamarin.Forms.Platform.WPF;

namespace BoxViewClock.WPF
{
    public partial class MainWindow : FormsApplicationPage
    {
        public MainWindow()
        {
            InitializeComponent();

            Forms.Init();
            LoadApplication(new BoxViewClock.App());
        }
    }
}
```

10. 右键单击中的 WPF 项目**解决方案资源管理器**，然后选择设为**启动项目**。按 f5 键以与 Visual Studio 调试器在 Windows 桌面上运行该程序：



后续步骤

平台特定信息

您可以确定从代码或 XAML 在 Xamarin.Forms 应用程序运行哪些平台。这样，您可以在 WPF 上运行时更改程序特征。在代码中，进行比较的值 `Device.RuntimePlatform` 与 `Device.WPF` 常量（它等于字符串"WPF"）。如果没有匹配项，在 WPF 上运行应用程序。

在 XAML 中，你可以使用 `OnPlatform` 标记来选择特定于平台的属性值：

```
<Button.TextColor>
    <OnPlatform x:TypeArguments="Color">
        <On Platform="iOS" Value="White" />
        <On Platform="macOS" Value="White" />
        <On Platform="Android" Value="Black" />
        <On Platform="WPF" Value="Blue" />
    </OnPlatform>
</Button.TextColor>
```

窗口大小

您可以调整在 WPF 窗口的初始大小**MainWindow.xaml**文件：

```
Title="BoxViewClock" Height="450" Width="800"
```

问题

这是预览版，因此可以预见，并非所有内容都可用于生产。并非所有的 Xamarin.Forms NuGet 包已准备对于 WPF，和某些功能可能无法完全正常。

Xamarin.Essentials

2018/11/1 • • Edit Online

Xamarin.Essentials 可为开发人员提供其移动应用程序的跨平台 API。

Android、iOS 和 UWP 提供了唯一的操作系统和平台 API，开发人员可以利用 Xamarin 访问 C# 中的所有 API。Xamarin.Essentials 提供了适用于任何 Xamarin.Forms、Android、iOS 或 UWP 应用程序的单个跨平台 API，不管如何创建用户界面，都可以通过共享代码进行访问。

Xamarin.Essentials 入门

按照入门指南将 **Xamarin.Essentials** NuGet 包安装到现有或新的 Xamarin.Forms、Android、iOS 或 UWP 项目中。

功能指南

按照指南将这些 Xamarin.Essentials 功能集成到你的应用程序：

- 加速计 – 检索设备在三个维空间中的加速数据。
- 应用信息 – 查找有关应用程序的信息。
- 气压计 – 监视气压计是否发生压力变化。
- 电池 – 轻松检测电池电量、源和状态。
- 剪贴板 – 快速方便地设置或读取剪贴板上的文本。
- 指南针 – 监视指南针是否发生变化。
- 连接性 – 检查连接状态并检测变化。
- 数据传输 – 将文本和网站 URI 发送到其他应用。
- 设备显示信息 – 获取设备的屏幕指标和方向。
- 设备信息 – 轻松查找有关设备的信息。
- 电子邮件 – 轻松发送电子邮件。
- 文件系统帮助程序 – 轻松地将文件保存到应用数据。
- 手电筒 – 打开/关闭手电筒的简单方法。
- 地理编码 – 地理编码和反向地理编码地址和坐标。
- 地理位置 – 检索设备的 GPS 位置。
- 陀螺仪 – 跟踪围绕设备的三个主轴的旋转。
- 启动器 – 使应用程序能够打开系统的 URI。
- 磁力计 – 检测设备相对于地球磁场的方向。
- 主线程 – 在应用程序的主线程上运行代码。
- 地图 – 将地图应用程序打开到特定位置。
- 打开浏览器 – 快速方便地将浏览器打开到特定网站。
- 方向传感器 – 检索设备在三个维空间中的方向。
- 电话拨号程序 – 打开电话拨号程序。
- 能源 – 获取设备的节能模式状态。

- 首选项 – 快速方便地添加永久首选项。
- 屏幕锁定 – 使设备屏幕保持唤醒状态。
- 安全存储 – 安全地存储数据。
- SMS – 创建要发送的短信。
- 文本到语音转换 – 在设备上读出文本。
- 版本跟踪 – 跟踪应用程序版本和内部版本号。
- 振动 – 使振动设备。

疑难解答

如果遇到问题，请寻求帮助。

API 文档

浏览 API 文档了解每个 Xamarin.Essentials 功能。

Xamarin.Essentials 入门

2018/11/2 • • Edit Online

Xamarin.Essentials 提供了适用于任何 iOS、Android 或 UWP 应用程序的单一跨平台 API，不管如何创建用户界面，都可以通过共享代码进行访问。

平台支持

Xamarin.Essentials 支持以下平台和操作系统：

平台	版本
Android	4.4 (API 19) 或更高版本
iOS	10.0 或更高版本
UWP	10.0.16299.0 或更高版本

安装

Xamarin.Essentials 可用作 NuGet 包，可以通过使用 Visual Studio 将其添加到任何现有或新的项目。

1. 使用 Visual Studio tools for Xamarin 下载并安装 Visual Studio。
2. 使用 Visual Studio C#（Android、iPhone 和 iPad 或跨平台）下的空白应用模板打开现有项目，或创建新项目。**重要说明**：如果添加到 UWP 项目，请确保在项目属性中设置内部版本 16299 或更高版本。
3. 将 Xamarin.Essentials NuGet 包添加到每个项目：

   - Visual Studio
   - Visual Studio for Mac

   在“解决方案资源管理器”面板中，右键单击解决方案名称，然后选择“管理 NuGet 包”。搜索 Xamarin.Essentials 并将包安装到所有项目，包括 Android、iOS、UWP 和 .NET Standard 库。

   > **TIP**
   > Xamarin.Essentials NuGet 处于预览状态时，选中“包括预发行版”框。

4. 在任何 C# 类中添加对 Xamarin.Essentials 的引用以引用 API。

   ```
   using Xamarin.Essentials;
   ```

5. Xamarin.Essentials 需要特定于平台的设置：

- Android
- iOS
- UWP

Xamarin.Essentials 支持最低 Android 版本 4.4（对应于 API 级别 19），但用于编译的目标 Android 版本必须为 8.1（对应于 API 级别 27）。（在 Visual Studio 中，已在“Android 清单”选项卡中的 Android 项目的“项目属性”对话框中设置这两个版本。在 Visual Studio for Mac 中，已在“Android 应用程序”选项卡中的 Android 项目的“项目选项”对话框中设置这两个版本。）

Xamarin.Essentials 安装所需的 Xamarin.Android.Support 库的版本 27.0.2.1。应用程序所需的任何其他 Xamarin.Android.Support 库还应使用 NuGet 包管理器更新到版本 27.0.2.1。应用程序所使用的所有 Xamarin.Android.Support 库应相同，并且至少应为版本 27.0.2.1。如果在解决方案中添加 Xamarin.Essentials NuGet 或更新 Nuget 时遇到问题，请参阅疑难解答页面。

在 Android 项目的 `MainLauncher` 或任何启动的 `Activity` 中，必须在 `OnCreate` 方法中初始化 Xamarin.Essentials：

```
protected override void OnCreate(Bundle savedInstanceState) {
    //...
    base.OnCreate(savedInstanceState);
    Xamarin.Essentials.Platform.Init(this, savedInstanceState); // add this line to your code
    //...
```

若要处理 Android 上的运行时权限，Xamarin.Essentials 必须接收任何 `OnRequestPermissionsResult`。将以下代码添加到所有 `Activity` 类：

```
public override void OnRequestPermissionsResult(int requestCode, string[] permissions,
[GeneratedEnum] Android.Content.PM.Permission[] grantResults)
{
    Xamarin.Essentials.Platform.OnRequestPermissionsResult(requestCode, permissions,
grantResults);

    base.OnRequestPermissionsResult(requestCode, permissions, grantResults);
}
```

6. 按照 Xamarin.Essentials 指南为每个功能复制并粘贴代码片段。

Xamarin.Essentials - 适用于移动应用的跨平台 API（视频）

其他资源

建议刚开始接触 Xamarin 的开发人员访问 Xamarin 开发入门。

访问 Xamarin.Essentials GitHub 存储库查看当前源代码，接下来，运行示例，并克隆存储库。欢迎为社区做贡献！

浏览 API 文档了解每个 Xamarin.Essentials 功能。

Xamarin.Essentials: Accelerometer

2018/11/1 • • Edit Online

Accelerometer 类可用于监视设备的加速计传感器，指示设备在三维空间内的加速度。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Accelerometer

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Accelerometer 功能通过调用 `Start` 和 `Stop` 方法来侦听加速的变化。然后通过 `ReadingChanged` 事件反馈任何变化。示例用法如下：

```
public class AccelerometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public AccelerometerTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Accelerometer.ReadingChanged += Accelerometer_ReadingChanged;
    }

    void Accelerometer_ReadingChanged(object sender, AccelerometerChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: X: {data.Acceleration.X}, Y: {data.Acceleration.Y}, Z:
{data.Acceleration.Z}");
        // Process Acceleration X, Y, and Z
    }

    public void ToggleAccelerometer()
    {
        try
        {
            if (Accelerometer.IsMonitoring)
              Accelerometer.Stop();
            else
              Accelerometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

Accelerometer 读数以 G 为单位反馈回来。G 是等于地球重力加速度 (9.81 m/s^2) 的重力单位。

相对于手机屏幕的默认方向定义坐标系。设备的屏幕方向更改时，不会交换轴。

X 轴水平向右，Y 轴垂直向上，Z 轴从屏幕正面指向外。在此坐标系中，屏幕后方的坐标具有负 Z 值。

示例：

- 当设备平放在桌面上，并从左侧向右推时，X 轴加速值为正。
- 当设备平放在桌面上，加速值为 +1.00 G (+ 9.81 m/s^2)，对应于设备的加速度 (0 m/s^2) 减去重力加速度 (-9.81 m/s ^2)，以 G 为单位规范化。
- 当设备平放在桌面上，并以 A m/s^2 的加速度向上推时，加速值等于 A+9.81，对应于设备的加速度 (+A m/s^2) 减去重力加速度 (-9.81 m/s ^2)，并以 G 为单位规范化。

传感器速度

- 最快 – 尽快获取传感器数据(不保证在 UI 线程上返回)。
- 游戏 – 适合游戏的速度(不保证在 UI 线程上返回)。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

API

- Accelerometer 源代码
- Accelerometer API 文档

Xamarin.Essentials: 应用信息

2018/11/10 • • Edit Online

AppInfo 类提供应用程序的相关信息。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 AppInfo

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

获取应用程序信息：

通过 API 公开了以下信息：

```
// Application Name
var appName = AppInfo.Name;

// Package Name/Application Identifier (com.microsoft.testapp)
var packageName = AppInfo.PackageName;

// Application Version (1.0.0)
var version = AppInfo.VersionString;

// Application Build Number (1)
var build = AppInfo.BuildString;
```

显示应用程序设置

AppInfo 类还可以显示由操作系统为应用程序维护的设置页面：

```
// Display settings page
AppInfo.ShowSettingsUI();
```

此设置页面使用户能够更改应用程序权限，并执行其他特定于平台的任务。

API

- AppInfo 源代码
- AppInfo API 文档

Xamarin.Essentials: Barometer

2018/11/1 • • Edit Online

Barometer 类可用于监视设备的气压计传感器，该传感器可测量压力。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Barometer

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Barometer 功能通过调用 `Start` 和 `Stop` 方法来侦听气压计压力读数的变化（以千帕为单位）。然后通过 `ReadingChanged` 事件反馈任何变化。示例用法如下：

```
public class BarometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public BarometerTest()
    {
        // Register for reading changes.
        Barometer.ReadingChanged += Barometer_ReadingChanged;
    }

    void Barometer_ReadingChanged(object sender, BarometerChangedEventArgs e)
    {
        var data = e.Reading;
        // Process Pressure
        Console.WriteLine($"Reading: Pressure: {data.Pressure} kilopascals");
    }

    public void ToggleBarometer()
    {
        try
        {
            if (Barometer.IsMonitoring)
              Barometer.Stop();
            else
              Barometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

传感器速度

- 最快 – 尽快获取传感器数据(不保证在 UI 线程上返回)。
- 游戏 – 适合游戏的速度(不保证在 UI 线程上返回)。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

平台实现细节

- Android
- iOS
- UWP

无特定于平台的实现细节。

API

- Barometer 源代码

- Barometer API 文档

Xamarin.Essentials: Battery

2018/11/10 • • Edit Online

Battery 类使你能够查看设备的电池信息并监视更改。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Battery 功能，需要以下特定于平台的设置。

- Android
- iOS
- UWP

需要具有 `Battery` 权限，并且必须在 Android 项目中进行配置。可以通过以下方法添加此权限：

打开 Properties 文件夹下的 AssemblyInfo.cs 文件并添加：

```
[assembly: UsesPermission(Android.Manifest.Permission.BatteryStats)]
```

或更新 Android 清单：

打开 Properties 文件夹下的 AndroidManifest.xml 文件，并在“manifest”节点内添加以下代码。

```
<uses-permission android:name="android.permission.BATTERY_STATS" />
```

或右键单击 Android 项目并打开项目的属性。在“Android 清单”下找到“所需权限:”区域，然后选中“Battery”权限。这样会自动更新 AndroidManifest.xml 文件。

使用 Battery

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

查看当前的电池信息：

```
var level = Battery.ChargeLevel; // returns 0.0 to 1.0 or -1.0 if unable to determine.

var state = Battery.State;

switch (state)
{
    case BatteryState.Charging:
        // Currently charging
        break;
    case BatteryState.Full:
        // Battery is full
        break;
    case BatteryState.Discharging:
    case BatteryState.NotCharging:
        // Currently discharging battery or not being charged
        break;
    case BatteryState.NotPresent:
        // Battery doesn't exist in device (desktop computer)
    case BatteryState.Unknown:
        // Unable to detect battery state
        break;
}

var source = Battery.PowerSource;

switch (source)
{
    case BatteryPowerSource.Battery:
        // Being powered by the battery
        break;
    case BatteryPowerSource.AC:
        // Being powered by A/C unit
        break;
    case BatteryPowerSource.Usb:
        // Being powered by USB cable
        break;
    case BatteryPowerSource.Wireless:
        // Powered via wireless charging
        break;
    case BatteryPowerSource.Unknown:
        // Unable to detect power source
        break;
}
```

每当电池的任一属性发生更改时，将触发一个事件：

```
public class BatteryTest
{
    public BatteryTest()
    {
        // Register for battery changes, be sure to unsubscribe when needed
        Battery.BatteryChanged += Battery_BatteryChanged;
    }

    void Battery_BatteryChanged(object sender, BatteryChangedEventArgs   e)
    {
        var level = e.ChargeLevel;
        var state = e.State;
        var source = e.PowerSource;
        Console.WriteLine($"Reading: Level: {level}, State: {state}, Source: {source}");
    }
}
```

平台差异

- Android
- iOS
- UWP

无平台差异。

API

- Battery 源代码
- Battery API 文档

Xamarin.Essentials: Clipboard

2018/11/10 • • Edit Online

Clipboard 类使你能够在应用程序之间将文本复制并粘贴到系统剪贴板。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Clipboard

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

检查 Clipboard 是否有当前已准备好要粘贴的文本：

```
var hasText = Clipboard.HasText;
```

将文本设置到 Clipboard：

```
Clipboard.SetText("Hello World");
```

从 Clipboard 读取文本：

```
var text = await Clipboard.GetTextAsync();
```

API

- Clipboard 源代码
- Clipboard API 文档

Xamarin.Essentials: Compass

2018/11/10 • • Edit Online

Compass 类使你能够监视设备的磁北航向。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Compass

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `Start` 和 `Stop` 方法来使用 Compass 功能以侦听罗盘的变化。然后通过 `ReadingChanged` 事件反馈任何变化。下面是一个示例：

```
public class CompassTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public CompassTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        Compass.ReadingChanged += Compass_ReadingChanged;
    }

    void Compass_ReadingChanged(object sender, CompassChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: {data.HeadingMagneticNorth} degrees");
        // Process Heading Magnetic North
    }

    public void ToggleCompass()
    {
        try
        {
            if (Compass.IsMonitoring)
              Compass.Stop();
            else
              Compass.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Some other exception has occurred
        }
    }
}
```

传感器速度

- 最快 – 尽快获取传感器数据(不保证在 UI 线程上返回)。
- 游戏 – 适合游戏的速度(不保证在 UI 线程上返回)。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

平台实现细节

- Android

Android 不提供用于检索罗盘航向的 API。我们使用 Google 推荐的方法，利用加速计和磁力计来计算磁北航向。

在极少数情况下，可能会看到不一致的结果，因为需要校准传感器，这就涉及到以 8 字形来移动设备。进行此操作的最佳方式是打开 Google 地图，点击你所在的位置点，然后选择“校准罗盘”。

请注意，同时从应用运行多个传感器可能需要调整传感器速度。

低通筛选器

根据 Android 罗盘值的更新和计算方式，可能需要平滑处理这些值。可以应用一个低通筛选器来平均角度的正弦和余弦值，并且可以通过在 `Compass` 类上设置 `ApplyLowPassFilter` 属性来启用此筛选器：

```
Compass.ApplyLowPassFilter = true;
```

这仅适用于 Android 平台。可以在此处查看详细信息。

API

- Compass 源代码
- Campass API 文档

Xamarin.Essentials: Connectivity

2018/11/1 • • Edit Online

Connectivity 类可用于监视设备的网络状况是否发生变化，检查当前的网络访问权限，以及当前的连接方式。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Connectivity 功能，需要以下特定于平台的设置。

- Android
- iOS
- UWP

`AccessNetworkState` 权限是必需的，且必须在 Android 项目中配置。可通过以下方法添加此权限：

打开 Properties 文件夹下的 AssemblyInfo.cs 文件并添加：

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessNetworkState)]
```

或更新 Android 清单：

打开 Properties 文件夹下的 AndroidManifest.xml 文件，并在 manifest 节点内添加。

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

或右键单击 Android 项目并打开项目的属性。在"Android 清单"下查找"所需权限:"区域，然后选中"访问网络状态"权限。这样会自动更新 AndroidManifest.xml 文件。

使用 Connectivity

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

检查当前网络访问：

```
var current = Connectivity.NetworkAccess;

if (current == NetworkAccess.Internet)
{
    // Connection to internet is available
}
```

网络访问分为以下几类：

- Internet – 本地和 Internet 访问。
- ConstrainedInternet – 受限 Internet 访问。指示强制网络门户连接情况，其中可以本地访问 Web 门户，但需要通过门户提供特定凭据才能访问 Internet。
- 本地 – 仅限本地网络访问。
- 无 – 无可用连接。
- 未知 – 无法确定 Internet 连接。

你可以检查设备当前正在使用哪种连接配置文件：

```
var profiles = Connectivity.Profiles;
if (profiles.Contains(ConnectionProfile.WiFi))
{
    // Active Wi-Fi connection.
}
```

只要连接配置文件或网络访问发生变化，就可以接收已触发的事件：

```
public class ConnectivityTest
{
    public ConnectivityTest()
    {
        // Register for connectivity changes, be sure to unsubscribe when finished
        Connectivity.ConnectivityChanged += Connectivity_ConnectivityChanged;
    }

    void Connectivity_ConnectivityChanged(object sender, ConnectivityChangedEventArgs  e)
    {
        var access = e.NetworkAccess;
        var profiles = e.Profiles;
    }
}
```

限制

需要注意的是 `Internet` 可能由 `NetworkAccess` 报告，但对 Web 的完全访问权限不可用。由于每个平台上的连接方式不同，因此只能保证连接可用。例如，设备可能会连接到 Wi-Fi 网络，但路由器与 Internet 断开连接。在此示例中，可能会报告 Internet，但活动连接不可用。

API

- Connectivity 源代码
- Connectivity API 文档

Xamarin.Essentials：数据传输

2018/11/10 • • Edit Online

DataTransfer 类使应用程序能够将数据（例如文本和 Web 链接）共享到设备上的其他应用程序。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用数据传输

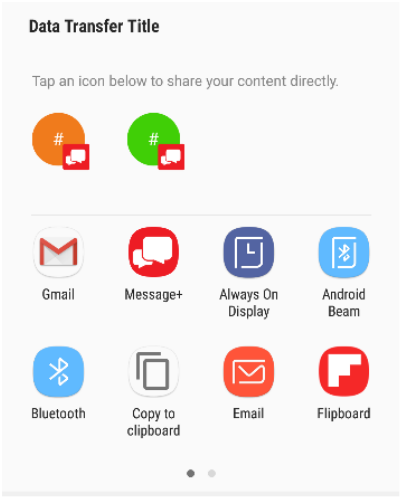
在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

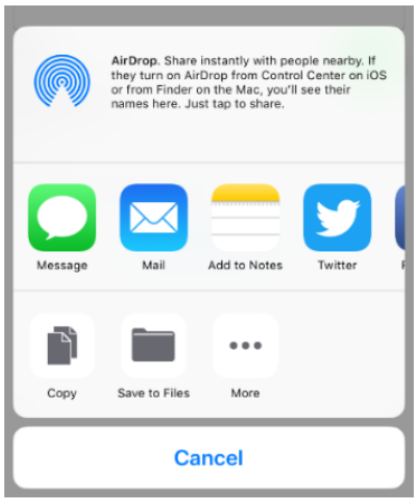
通过调用具有数据请求有效负载的 `RequestAsync` 方法来使用数据传输功能，此有效负载包括要共享到其他应用程序的信息。文本和 URI 可以混合使用，每个平台都将根据内容进行筛选处理。

```
public class DataTransferTest
{
    public async Task ShareText(string text)
    {
        await DataTransfer.RequestAsync(new ShareTextRequest
            {
                Text = text,
                Title = "Share Text"
            });
    }

    public async Task ShareUri(string uri)
    {
        await DataTransfer.RequestAsync(new ShareTextRequest
            {
                Uri = uri,
                Title = "Share Web Link"
            });
    }
}
```

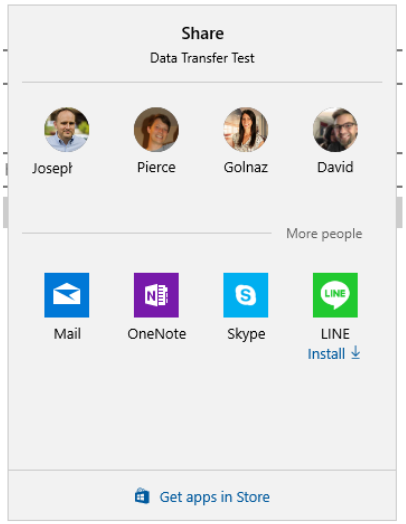
在进行请求时显示的共享到外部应用程序的用户界面：



Android



iOS



UWP

平台差异

- Android
- iOS
- UWP

- `Subject` 属性用于所需的消息主题。

API

- 数据传输源代码
- 数据传输 API 文档

Xamarin.Essentials: 设备显示信息

2018/11/10 • • Edit Online

DeviceDisplay 类提供有关运行应用程序的设备的屏幕指标信息。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 DeviceDisplay

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

屏幕指标

除了基本的设备信息外，DeviceDisplay 类还包含有关设备的屏幕和方向信息。

```
// Get Metrics
var metrics = DeviceDisplay.ScreenMetrics;

// Orientation (Landscape, Portrait, Square, Unknown)
var orientation = metrics.Orientation;

// Rotation (0, 90, 180, 270)
var rotation = metrics.Rotation;

// Width (in pixels)
var width = metrics.Width;

// Height (in pixels)
var height = metrics.Height;

// Screen density
var density = metrics.Density;
```

DeviceDisplay 类还会公开可以订阅的一个事件，每当任何屏幕指标更改时就会触发此事件：

```
public class ScreenMetricsTest
{
    public ScreenMetricsTest()
    {
        // Subscribe to changes of screen metrics
        DeviceDisplay.ScreenMetricsChanged += OnScreenMetricsChanged;
    }

    void OnScreenMetricsChanged(ScreenMetricsChangedEventArgs  e)
    {
        // Process changes
        var metrics = e.Metrics;
    }
}
```

平台差异

- Android
- iOS
- UWP

没有差异。

API

- DeviceDisplay 源代码
- DeviceDisplay API 文档

Xamarin.Essentials: 设备信息

2018/11/10 • • Edit Online

DeviceInfo 类提供有关运行应用程序的设备的信息。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 DeviceInfo

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过 API 公开了以下信息：

```
// Device Model (SMG-950U, iPhone10,6)
var device = DeviceInfo.Model;

// Manufacturer (Samsung)
var manufacturer = DeviceInfo.Manufacturer;

// Device Name (Motz's iPhone)
var deviceName = DeviceInfo.Name;

// Operating System Version Number (7.0)
var version = DeviceInfo.VersionString;

// Platform (Android)
var platform = DeviceInfo.Platform;

// Idiom (Phone)
var idiom = DeviceInfo.Idiom;

// Device Type (Physical)
var deviceType = DeviceInfo.DeviceType;
```

平台

`DeviceInfo.Platform` 与映射到操作系统的一个常量字符串相关联。可以使用 `Platforms` 类检查以下值：

- **DeviceInfo.Platforms.iOS** - iOS
- **DeviceInfo.Platforms.Android** - Android
- **DeviceInfo.Platforms.UWP** - UWP
- **DeviceInfo.Platforms.Unsupported** - 不受支持

习惯用语

`DeviceInfo.Idiom` 与映射到运行应用程序的设备类型的一个常量字符串相关联。可以使用 `Idioms` 类检查以下值：

- **DeviceInfo.Idioms.Phone** - 手机
- **DeviceInfo.Idioms.Tablet** - 平板电脑
- **DeviceInfo.Idioms.Desktop** - 桌面
- **DeviceInfo.Idioms.TV** - 电视
- **DeviceInfo.Idioms.Unsupported** - 不受支持

设备类型

`DeviceInfo.DeviceType` 关联一个枚举以确定应用程序是在物理设备还是虚拟设备上运行。虚拟设备是指模拟器或仿真程序。

平台实现细节

- iOS

iOS 不会向开发人员公开一个 API 来获取特定 iOS 设备的名称。而是会返回一个硬件标识符，例如 iPhone10,6，这是指这些标识符的 iPhone X. A 映射不是由 Apple 提供的，但可以在 The iPhone Wiki（一个非官方来源）上找到。

API

- DeviceInfo 源代码
- DeviceInfo API 文档

Xamarin.Essentials: Email

2018/11/10 • • Edit Online

Email 类使应用程序能够打开包含主题、正文和收件人(TO、CC、BCC)等指定信息的默认电子邮件应用程序。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Email

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `ComposeAsync` 方法和包含有关电子邮件信息的 `EmailMessage` 来使用 Email 功能：

```
public class EmailTest
{
    public async Task SendEmail(string subject, string body, List<string> recipients)
    {
        try
        {
            var message = new EmailMessage
            {
                Subject = subject,
                Body = body,
                To = recipients,
                //Cc = ccRecipients,
                //Bcc = bccRecipients
            };
            await Email.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException fbsEx)
        {
            // Email is not supported on this device
        }
        catch (Exception ex)
        {
            // Some other exception occurred
        }
    }
}
```

API

- Email 源代码
- Email API 文档

Xamarin.Essentials: 文件系统帮助程序

2018/11/14 • • Edit Online

FileSystem 类包含一系列帮助程序，用于查找应用程序的缓存和数据目录以及打开应用包内的文件。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用文件系统帮助程序

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

获取应用程序的目录以存储缓存数据。缓存数据可用于满足以下要求的任何数据：需要比临时数据持续更长时间，但不应是正确执行操作所需的数据。

```
var cacheDir = FileSystem.CacheDirectory;
```

为任何非用户数据文件的文件获取应用程序的顶级目录。这些文件是使用同步框架的操作系统进行备份的。查看下面的平台实现细节。

```
var mainDir = FileSystem.AppDataDirectory;
```

打开捆绑到应用程序包中的文件：

```
using (var stream = await FileSystem.OpenAppPackageFileAsync(templateFileName))
{
   using (var reader = new StreamReader(stream))
   {
       var fileContents = await reader.ReadToEndAsync();
   }
}
```

平台实现细节

- Android
- iOS
- UWP

- **CacheDirectory** - 返回当前上下文的 CacheDir。
- **AppDataDirectory** - 返回当前上下文的 FilesDir，并且是使用 API 23 及更高版本的自动备份进行备份的。

将任何文件添加到 Android 项目中的 Assets 文件夹中，并将生成操作标记为 AndroidAsset 以将其与 `OpenAppPackageFileAsync` 一起使用。

API

- 文件系统帮助程序源代码
- 文件系统 API 文档

Xamarin.Essentials: Flashlight

2018/11/14 • • Edit Online

Flashlight 类，此类使你能够打开或关闭设备的照相机闪光灯，将其转换为一个手电筒。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Flashlight 功能，需要以下特定于平台的设置。

- Android
- iOS
- UWP

需要具有 Flashlight 和 Camera 权限，并且必须在 Android 项目中进行配置。可以通过以下方法添加权限：

打开 Properties 文件夹下的 AssemblyInfo.cs 文件并添加：

```
[assembly: UsesPermission(Android.Manifest.Permission.Flashlight)]
[assembly: UsesPermission(Android.Manifest.Permission.Camera)]
```

或更新 Android 清单：

打开 Properties 文件夹下的 AndroidManifest.xml 文件，并在“manifest”节点内添加以下代码。

```
<uses-permission android:name="android.permission.FLASHLIGHT" />
<uses-permission android:name="android.permission.CAMERA" />
```

或右键单击 Android 项目并打开项目的属性。在“Android 清单”下找到“所需权限:”区域，然后选中“FLASHLIGHT”和“CAMERA”权限。这样会自动更新 AndroidManifest.xml 文件。

通过添加这些权限，Google Play 将自动筛选出设备，而无需任何特定硬件。可以通过将以下代码添加到 Android 项目中的 AssemblyInfo.cs 文件中来绕过此操作：

```
[assembly: UsesFeature("android.hardware.camera", Required = false)]
[assembly: UsesFeature("android.hardware.camera.autofocus", Required = false)]
```

使用 Flashlight

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

可以通过 `TurnOnAsync` 和 `TurnOffAsync` 方法来打开或关闭手电筒：

```
try
{
    // Turn On
    await Flashlight.TurnOnAsync();

    // Turn Off
    await Flashlight.TurnOffAsync();
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to turn on/off flashlight
}
```

平台实现细节

- Android
- iOS
- UWP

Flashlight 类已根据设备的操作系统进行了优化。

API 级别 23 及更高版本

在更新的 API 级别上，Torch 模式将用于打开或关闭设备的闪光单元。

API 级别 22 及更高版本

创建一个相机表面纹理以打开或关闭相机单元的 `FlashMode`。

API

- Flashlight 源代码
- Flashlight API 文档

Xamarin.Essentials: Geocoding

2018/11/14 • • Edit Online

Geocoding 类提供了 API，既可以将地标地理编码为位置坐标，又可以将坐标反向地理编码为地标。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Geocoding 功能，需要以下特定于平台的设置。

- Android
- iOS
- UWP

无需其他设置。

使用 Geocoding

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

获取一个地址的位置坐标：

```
try
{
    var address =  "Microsoft Building 25 Redmond WA USA";
    var locations = await Geocoding.GetLocationsAsync(address);

    var location = locations?.FirstOrDefault();
    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude:
{location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

高度并非总是可用的。如果不可用，`Altitude` 属性可能为 `null` 或值可能为零。如果高度可用，此值为海拔高度（以米为单位）。

为现有的一组坐标获取地标：

```
try
{
    var lat = 47.673988;
    var lon = -122.121513;

    var placemarks = await Geocoding.GetPlacemarksAsync(lat, lon);

    var placemark = placemarks?.FirstOrDefault();
    if (placemark != null)
    {
        var geocodeAddress =
            $"AdminArea:       {placemark.AdminArea}\n" +
            $"CountryCode:     {placemark.CountryCode}\n" +
            $"CountryName:     {placemark.CountryName}\n" +
            $"FeatureName:     {placemark.FeatureName}\n" +
            $"Locality:        {placemark.Locality}\n" +
            $"PostalCode:      {placemark.PostalCode}\n" +
            $"SubAdminArea:    {placemark.SubAdminArea}\n" +
            $"SubLocality:     {placemark.SubLocality}\n" +
            $"SubThoroughfare: {placemark.SubThoroughfare}\n" +
            $"Thoroughfare:    {placemark.Thoroughfare}\n";

        Console.WriteLine(geocodeAddress);
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Handle exception that may have occurred in geocoding
}
```

两个位置之间的距离

`Location` 和 `LocationExtensions` 类定义了可用于计算两个位置之间的距离的方法。有关示例，请参阅文章 **Xamarin.Essentials：Geolocation**。

API

- Geocoding 源代码
- Geocoding API 文档

Xamarin.Essentials: Geolocation

2018/11/14 • • Edit Online

Geolocation 提供 API 以检索设备的当前地理位置坐标。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Geolocation 功能，需要以下特定于平台的设置：

- Android
- iOS
- UWP

需要具有 Coarse 和 Fine Location 权限，并且必须在 Android 项目中进行配置。此外，如果应用面向 Android 5.0（API 级别 21）或更高版本，则必须声明应用使用清单文件中的硬件功能。可以通过以下方法添加此声明：

打开 Properties 文件夹下的 AssemblyInfo.cs 文件并添加：

```
[assembly: UsesPermission(Android.Manifest.Permission.AccessCoarseLocation)]
[assembly: UsesPermission(Android.Manifest.Permission.AccessFineLocation)]
[assembly: UsesFeature("android.hardware.location", Required = false)]
[assembly: UsesFeature("android.hardware.location.gps", Required = false)]
[assembly: UsesFeature("android.hardware.location.network", Required = false)]
```

或更新 Android 清单：

打开 Properties 文件夹下的 AndroidManifest.xml 文件，并在“manifest”节点内添加以下代码：

```
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-feature android:name="android.hardware.location" android:required="false" />
<uses-feature android:name="android.hardware.location.gps" android:required="false" />
<uses-feature android:name="android.hardware.location.network" android:required="false" />
```

或右键单击 Android 项目并打开项目的属性。在“Android 清单”下找到“所需权限:”区域，然后选中“ACCESS_COARSE_LOCATION”和“ACCESS_FINE_LOCATION”权限。这样会自动更新 AndroidManifest.xml 文件。

使用 Geolocation

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Geolocation API 还将在必要时提醒用户所需的权限。

通过调用 `GetLastKnownLocationAsync` 方法获取设备上次的已知位置。这通常比执行完整的查询更快，但可能不太准确。

```
try
{
    var location = await Geolocation.GetLastKnownLocationAsync();

    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude:
{location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

高度并非总是可用的。如果不可用，`Altitude` 属性可能为 `null` 或值可能为零。如果高度可用，此值为海拔高度（以米为单位）。

若要查询当前设备的位置坐标，可以使用 `GetLocationAsync`。最好传入一个完整的 `GeolocationRequest` 和 `CancellationToken`，因为获取设备的位置可能需要一些时间。

```
try
{
    var request = new GeolocationRequest(GeolocationAccuracy.Medium);
    var location = await Geolocation.GetLocationAsync(request);

    if (location != null)
    {
        Console.WriteLine($"Latitude: {location.Latitude}, Longitude: {location.Longitude}, Altitude:
{location.Altitude}");
    }
}
catch (FeatureNotSupportedException fnsEx)
{
    // Handle not supported on device exception
}
catch (PermissionException pEx)
{
    // Handle permission exception
}
catch (Exception ex)
{
    // Unable to get location
}
```

地理位置的准确性

下表概述了每个平台的准确性：

最低

平台	距离(以米为单位)
Android	500
iOS	3000
UWP	1000 - 5000

低

平台	距离(以米为单位)
Android	500
iOS	1000
UWP	300 - 3000

中等(默认值)

平台	距离(以米为单位)
Android	100 - 500
iOS	100
UWP	30-500

高

平台	距离(以米为单位)
Android	0 - 100
iOS	10
UWP	<= 10

最佳

平台	距离(以米为单位)
Android	0 - 100
iOS	~0
UWP	<= 10

两个位置之间的距离

`Location` 和 `LocationExtensions` 类定义了 `CalculateDistance` 方法，可用于计算两个地理位置之间的距离。此计算得出的距离不考虑道路或其他路径，仅仅是沿着地球表面的两点之间的最短距离，也称为大圆距离或通俗地称为"直线距离"。

以下是一个示例：

```
Location boston = new Location(42.358056, -71.063611);
Location sanFrancisco = new Location(37.783333, -122.416667);
double miles = Location.CalculateDistance(boston, sanFrancisco, DistanceUnits.Miles);
```

`Location` 构造函数具有按该顺序排列的纬度和经度参数。正纬度值表示位于赤道以北，正经度值表示位于本初子午线以东。使用 `CalculateDistance` 的最后一个参数指定单位为英里还是公里。`Location` 类还定义了用于在两个单位之间进行转换的 `KilometersToMiles` 和 `MilesToKilometers` 方法。

API

- Geolocation 源代码
- Geolocation API 文档

Xamarin.Essentials: Gyroscope

2018/11/10 • • Edit Online

Gyroscope 类使你能够监控设备的陀螺仪传感器，此传感器测量围绕设备三个主轴的旋转。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Gyroscope

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `Start` 和 `Stop` 方法来使用 Gyroscope 功能以侦听陀螺仪的变化。然后通过 `ReadingChanged` 事件反馈任何变化。示例用法如下：

```
public class GyroscopeTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public GyroscopeTest()
    {
        // Register for reading changes.
        Gyroscope.ReadingChanged += Gyroscope_ReadingChanged;
    }

    void Gyroscope_ReadingChanged(object sender, GyroscopeChangedEventArgs e)
    {
        var data = e.Reading;
        // Process Angular Velocity X, Y, and Z
        Console.WriteLine($"Reading: X: {data.AngularVelocity.X}, Y: {data.AngularVelocity.Y}, Z:
{data.AngularVelocity.Z}");
    }

    public void ToggleGyroscope()
    {
        try
        {
            if (Gyroscope.IsMonitoring)
              Gyroscope.Stop();
            else
              Gyroscope.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

传感器速度

- 最快 – 尽快获取传感器数据(不保证在 UI 线程上返回)。
- 游戏 – 适合游戏的速度(不保证在 UI 线程上返回)。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

API

- Gyroscope 源代码
- Gyroscope API 文档

Xamarin.Essentials: Launcher

2018/11/1 • • Edit Online

Launcher 类允许应用程序打开系统的 URI。通常在深入链接到另一个应用程序的自定义 URI 方案后使用此类。如果想要将浏览器打开到某个网站，则应引用浏览器 API。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Launcher

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

若要使用 Launcher 功能，请调用 `OpenAsync` 方法并传入要打开的 `string` 或 `Uri`。（可选）`CanOpenAsync` 方法可用于检查是否可以由设备上的应用程序处理 URI 架构。

```
public class LauncherTest
{
    public async Task OpenRideShareAsync()
    {
        var supportsUri = await Launcher.CanOpenAsync("lyft://");
        if (supportsUri)
            await Launcher.OpenAsync("lyft://ridetype?id=lyft_line");
    }
}
```

平台差异

- Android
- iOS
- UWP

立即完成从 `CanOpenAsync` 返回的任务。

API

- Launcher 源代码
- Launcher API 文档

Xamarin.Essentials: Magnetometer

2018/11/10 • • Edit Online

Magnetometer 类使你能够监视设备的磁力计传感器，此传感器指示设备相对于地球磁场的方向。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Magnetometer

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `Start` 和 `Stop` 方法来使用 Magnetometer 功能以侦听磁力计的变化。然后通过 `ReadingChanged` 事件反馈任何变化。示例用法如下：

```
public class MagnetometerTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public MagnetometerTest()
    {
        // Register for reading changes.
        Magnetometer.ReadingChanged += Magnetometer_ReadingChanged;
    }

    void Magnetometer_ReadingChanged(object sender, MagnetometerChangedEventArgs e)
    {
        var data = e.Reading;
        // Process MagneticField X, Y, and Z
        Console.WriteLine($"Reading: X: {data.MagneticField.X}, Y: {data.MagneticField.Y}, Z:
{data.MagneticField.Z}");
    }

    public void ToggleMagnetometer()
    {
        try
        {
            if (Magnetometer.IsMonitoring)
              Magnetometer.Stop();
            else
              Magnetometer.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

所有数据将以微特斯拉为单位返回。

传感器速度

- 最快 – 尽快获取传感器数据(不保证在 UI 线程上返回)。
- 游戏 – 适合游戏的速度(不保证在 UI 线程上返回)。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

API

- Magnetometer 源代码
- Magnetometer API 文档

Xamarin.Essentials: MainThread

2018/11/1 • • Edit Online

MainThread 类允许应用程序在主执行线程上运行代码，并确定当前是否在主线程上运行特定代码块。

背景

大多数操作系统(包括 iOS、Android 和通用 Windows 平台)对涉及用户界面的代码使用单线程模型。正确序列化用户界面事件(包括击键和触控输入)需要此模型。此线程通常称为"主线程"、"用户界面线程"或"UI 线程"。此模型的缺点是用于访问用户界面元素的所有代码必须在应用程序的主线程上运行。

应用程序有时需要使用在辅助执行线程上调用事件处理程序的事件。(Xamarin.Essentials 类 `Accelerometer`、`Compass`、`Gyroscope`、`Magnetometer` 和 `OrientationSensor` 以更快的速度使用时，可能会返回有关辅助线程的信息。)如果事件处理程序需要访问用户界面元素，则必须在主线程上运行该代码。MainThread 类允许应用程序在主线程上运行此代码。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

在主线程上运行代码

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

若要在主线程上运行代码，请调用静态 `MainThread.BeginInvokeOnMainThread` 方法。参数是 `Action` 对象，它只是没有参数且没有返回值的方法：

```
MainThread.BeginInvokeOnMainThread(() =>
{
    // Code to run on the main thread
});
```

也可以为必须在主线程上运行的代码定义单独的方法：

```
void MyMainThreadCode()
{
    // Code to run on the main thread
}
```

然后，可以通过在 `BeginInvokeOnMainThread` 方法中引用主线程，以便在主线程上运行此方法：

```
MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
```

> **NOTE**
>
> Xamarin.Forms 具有名为 `Device.BeginInvokeOnMainThread(Action)` 的方法，该方法与 `MainThread.BeginInvokeOnMainThread(Action)` 执行相同操作。虽然可以在 Xamarin.Forms 应用中使用任何一种方法，但请考虑调用代码是否需要在 Xamarin.Forms 上有任何其他依赖项。如果不需要，则 `MainThread.BeginInvokeOnMainThread(Action)` 可能是更好的选择。

确定是否在主线程上运行代码

`MainThread` 类还允许应用程序确定是否在主线程上运行特定代码块。如果在主线程上运行调用 `IsMainThread` 属性的代码，则该属性会返回 `true`。程序可以使用此属性运行主线程或辅助线程的不同代码：

```
if (MainThread.IsMainThread)
{
    // Code to run if this is the main thread
}
else
{
    // Code to run if this is a secondary thread
}
```

在调用 `BeginInvokeOnMainThread` 之前，你可能想知道是否应检查代码是否在辅助线程上运行，如下所示：

```
if (MainThread.IsMainThread)
{
    MyMainThreadCode();
}
else
{
    MainThread.BeginInvokeOnMainThread(MyMainThreadCode);
}
```

如果已在主线程上运行代码块，你可能会猜想此检查可能会提高性能。

但是，不需要执行此检查。 `BeginInvokeOnMainThread` 的平台实现本身会检查是否在主线程上调用代码。如果在并不需要时调用 `BeginInvokeOnMainThread`，则很少会有性能损失。

API

- MainThread 源代码
- MainThread API 文档

Xamarin.Essentials: Maps

2018/11/1 • • Edit Online

Maps 类允许应用程序将已安装的地图应用程序打开到特定位置或地标。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Maps

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Maps 功能通过调用具有 `Location` 或 `Placemark` 的 `OpenAsync` 方法来使用可选的 `MapsLaunchOptions` 打开。

```
public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options =  new MapsLaunchOptions { Name = "Microsoft Building 25" };

        await Maps.OpenAsync(location, options);
    }
}
```

使用 `Placemark` 打开时，需要以下信息：

- `CountryName`
- `AdminArea`
- `Thoroughfare`
- `Locality`

```
public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var placemark = new Placemark
            {
                CountryName = "United States",
                AdminArea = "WA",
                Thoroughfare = "Microsoft Building 25",
                Locality = "Redmond"
            };
        var options =  new MapsLaunchOptions { Name = "Microsoft Building 25" };

        await Maps.OpenAsync(placemark, options);
    }
}
```

扩展方法

如果已有对 `Location` 或 `Placemark` 的引用，则可以使用具有可选的 `MapsLaunchOptions` 的内置扩展方法 `OpenMapsAsync`：

```
public class MapsTest
{
    public async Task OpenPlacemarkOnMaps(Placemark placemark)
    {
        await placemark.OpenMapsAsync();
    }
}
```

方向模式

如果调用不带任何 `MapsLaunchOptions` 的 `OpenMapsAsync`，则地图将启动到指定位置。（可选）可以有从设备的当前位置开始计算的导航路线。这是通过设置 `MapsLaunchOptions` 上的 `MapDirectionsMode` 来完成的：

```
public class MapsTest
{
    public async Task NavigateToBuilding25()
    {
        var location = new Location(47.645160, -122.1306032);
        var options =  new MapsLaunchOptions { MapDirectionsMode = MapDirectionsMode.Driving };

        await Maps.OpenAsync(location, options);
    }
}
```

平台差异

- Android
- iOS
- UWP

- `MapDirectionsMode` 支持骑行、驾车和步行。

平台实现细节

- Android

- iOS
- UWP

Android 使用 `geo:` URI 方案启动设备上的地图应用程序。这可能会提示用户从支持此 URI 方案的现有应用程序中进行选择。Xamarin.Essentials 使用支持此方案的 Google 地图进行测试。

API

- Maps 源代码
- Maps API 文档

Xamarin.Essentials: Browser

2018/11/1 • • Edit Online

Browser 类允许应用程序在优化的系统首选浏览器或外部浏览器中打开 Web 链接。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Browser

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Browser 功能通过调用具有 `Uri` 和 `BrowserLaunchMode` 的 `OpenAsync` 方法工作。

```
public class BrowserTest
{
    public async Task OpenBrowser(Uri uri)
    {
        await Browser.OpenAsync(uri, BrowserLaunchMode.SystemPreferred);
    }
}
```

平台实现细节

- Android
- iOS
- UWP

启动模式确定浏览器的启动方式：

系统首选

Chrome 自定义选项卡将尝试用于加载 URI 并保持导航识别。

外部

`Intent` 将用于请求通过系统常规浏览器打开的 URI。

API

- Browser 源代码

- Browser API 文档

Xamarin.Essentials: OrientationSensor

2018/11/1 • • Edit Online

OrientationSensor 类可让你监视设备在三维空间中的方向。

> **NOTE**
> 此类用于确定设备在三维空间中的方向。如果需要确定设备的视频显示器是处于纵向模式还是横向模式，请使用可从 `DeviceDisplay` 类获得的 `ScreenMetrics` 对象的 `Orientation` 属性。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 OrientationSensor

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `Start` 方法启用 `OrientationSensor` 以便监视对设备方向所做的更改，并通过调用 `Stop` 方法进行禁用。然后通过 `ReadingChanged` 事件反馈任何变化。示例用法如下：

```
public class OrientationSensorTest
{
    // Set speed delay for monitoring changes.
    SensorSpeed speed = SensorSpeed.UI;

    public OrientationSensorTest()
    {
        // Register for reading changes, be sure to unsubscribe when finished
        OrientationSensor.ReadingChanged += OrientationSensor_ReadingChanged;
    }

    void OrientationSensor_ReadingChanged(object sender, OrientationSensorChangedEventArgs e)
    {
        var data = e.Reading;
        Console.WriteLine($"Reading: X: {data.Orientation.X}, Y: {data.Orientation.Y}, Z:
{data.Orientation.Z}, W: {data.Orientation.W}");
        // Process Orientation quaternion (X, Y, Z, and W)
    }

    public void ToggleOrientationSensor()
    {
        try
        {
            if (OrientationSensor.IsMonitoring)
                OrientationSensor.Stop();
            else
                OrientationSensor.Start(speed);
        }
        catch (FeatureNotSupportedException fnsEx)
        {
            // Feature not supported on device
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

`OrientationSensor` 读数以 `Quaternion` 的形式返回报告，描述了设备基于两个三维坐标系的方向：

设备（通常为手机或平板电脑）的三维坐标系具有以下轴：

- X 轴正方向指向显示器右侧（在纵向模式下）。
- Y 轴正方向指向显示器顶部（在纵向模式下）。
- Z 轴正方向指向屏幕外侧。

地球的三维坐标系具有以下轴：

- X 轴正方向与地球表面相切并指向东边。
- Y 轴正方向也与地球表面相切并指向北边。
- Z 轴正方向与地球表面垂直并指向上边。

`Quaternion` 描述了设备坐标系相对于地球坐标系的旋转。

`Quaternion` 值与绕轴旋转密切相关。如果旋转的轴是规范化矢量 (a_x, a_y, a_z)，并且旋转角度为 Θ，则四元数的 (X, Y, Z, W) 分量是：

(a_x·sin(Θ/2), a_y·sin(Θ/2), a_z·sin(Θ/2), cos(Θ/2))

这些是右手坐标系，因此右手的拇指指向旋转轴的正方向，手指弯曲表示正角的旋转方向。

示例：

- 如果设备平躺在桌子上，使其屏幕面朝上，设备顶部（在纵向模式下）指向北边，则会对齐这两个坐标系。`Quaternion` 值表示标识四元数 (0, 0, 0, 1)。可以相对于此位置分析所有旋转。
- 如果设备平躺在桌子上，使其屏幕面朝上，设备顶部（在纵向模式下）指向西边，则 `Quaternion` 值为 (0, 0, 0.707, 0.707)。设备已绕地球的 Z 轴旋转 90 度。
- 如果设备直立，设备顶部（在纵向模式下）指向天空，设备背面朝向北边，则设备已绕 X 轴旋转 90 度。`Quaternion` 值为 (0.707, 0, 0, 0.707)。
- 如果设备的左边缘在桌子上，顶部指向北边，则设备已绕 Y 轴旋转 –90 度（或绕 Y 轴负方向旋转 90 度）。`Quaternion` 值为 (0, -0.707, 0, 0.707)。

传感器速度

- 最快 – 尽快获取传感器数据（不保证在 UI 线程上返回）。
- 游戏 – 适合游戏的速度（不保证在 UI 线程上返回）。
- 正常 – 适合屏幕方向更改的默认速率。
- UI – 适合常规用户界面的速率。

如果事件处理程序不能保证在 UI 线程上运行，并且如果事件处理程序需要访问用户界面元素，请使用 `MainThread.BeginInvokeOnMainThread` 方法在 UI 线程上运行该代码。

API

- OrientationSensor 源代码
- OrientationSensor API 文档

Xamarin.Essentials: 电话拨号程序

2018/11/10 • • Edit Online

PhoneDialer 类使应用程序能够在拨号程序中打开一个电话号码。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用电话拨号程序

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用具有要用于打开拨号程序的一个电话号码的 `Open` 方法来使用电话拨号程序功能。当请求 `Open` 时，API 将自动尝试根据国家/地区代码设置号码的格式（如果已指定）。

```
public class PhoneDialerTest
{
    public async Task PlacePhoneCall(string number)
    {
        try
        {
            PhoneDialer.Open(number);
        }
        catch (ArgumentNullException anEx)
        {
            // Number was null or white space
        }
        catch (FeatureNotSupportedException ex)
        {
            // Phone Dialer is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

API

- 电话拨号程序源代码
- 电话拨号程序 API 文档

Xamarin.Essentials: 节能模式状态

2018/11/1 • • Edit Online

Power 类提供有关设备的节能模式状态的信息，指示设备是否在低功耗模式下运行。如果设备的节能模式状态已打开，则应用程序应避免后台处理。

背景

使用电池运行的设备可以置于低功耗节能模式。有时，设备会自动切换到此模式，例如，当电池电量降到 20% 以下时。操作系统通过减少往往会消耗电池的活动来响应节能模式。打开节能模式时，应用程序有助于避免后台处理或其他高功率活动。

对于 Android 设备，Power 类仅为 Android 版本 5.0 (Lollipop) 及更高版本返回有意义的信息。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Power 类

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

使用静态 `Power.EnergySaverStatus` 属性获取设备的当前节能状态：

```
// Get energy saver status
var status = Power.EnergySaverStatus;
```

此属性会返回 `EnergySaverStatus` 枚举的成员，可以是 `On`、`Off` 或 `Unknown`。如果该属性返回 `On`，则应用程序应避免后台处理或可能会消耗大量电力的其他活动。

应用程序还应安装事件处理程序。Power 类会公开节能模式状态发生更改时触发的事件：

```
public class EnergySaverTest
{
    public EnergySaverTest()
    {
        // Subscribe to changes of energy-saver status
        Power.EnergySaverStatusChanged += OnEnergySaverStatusChanged;
    }

    private void OnEnergySaverStatusChanged(EnergySaverStatusChangedEventArgs e)
    {
        // Process change
        var status = e.EnergySaverStatus;
    }
}
```

如果节能模式状态更改为 `On`，则应用程序应停止执行后台处理。如果状态更改为 `Unknown` 或 `Off`，则应用程序可以继续执行后台处理。

API

- Power 源代码
- Power API 文档

Xamarin.Essentials: Preferences

2018/11/14 • • Edit Online

Preferences 类帮助将应用程序首选项存储在键/值存储中。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Preferences

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

将给定密匙的值保存在首选项中：

```
Preferences.Set("my_key", "my_value");
```

从首选项检索值或检索默认值（如果未设置）：

```
var myValue = Preferences.Get("my_key", "default_value");
```

从首选项删除密钥：

```
Preferences.Remove("my_key");
```

删除所有首选项：

```
Preferences.Clear();
```

除了这些方法外，每个方法都采用一个可选的 `sharedName`，可用于创建首选的其他容器。查看下面的平台实现细节。

支持的数据类型

以下数据类型在 Preferences 中受到支持：

- **bool**
- **double**
- **int**
- **float**

- **long**
- **string**
- **DateTime**

实现详细信息

`DateTime` 的值是使用 `DateTime` 类定义的两种方法以 64 位二进制(长整型)格式存储的: `ToBinary` 方法用于对 `DateTime` 值进行编码, `FromBinary` 方法对值进行解码。当存储的 `DateTime` 不是协调世界时 (UTC) 值时, 请参阅这些方法的相关文档以了解如何进行对值解码的调整。

平台实现细节

- Android
- iOS
- UWP

所有数据都存储到共享首选项中。如果未指定 `sharedName`, 则使用默认的共享首选项, 否则此名称将用于获取具有指定名称的私有共享首选项。

持久性

卸载应用程序将导致所有首选项被删除。对此有一个例外, 即面向使用自动备份的 Android 6.0(API 级别 23)或更高版本并在其上运行的应用。此功能默认启用并且会保留应用数据, 包括共享首选项(即 Preferences API 使用的内容)。可以遵循 Google 的文档禁用此功能。

限制

当存储一个字符串时, 此 API 用于存储少量文本。如果尝试将其用于存储大量文本, 则可能会导致性能欠佳。

API

- Preferences 源代码
- Preferences API 文档

Xamarin.Essentials: 屏幕锁定

2018/11/10 • • Edit Online

ScreenLock 类可以请求在应用程序运行时防止屏幕进入睡眠状态。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 ScreenLock

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

通过调用 `RequestActive` 和 `RequestRelease` 方法使用屏幕锁定功能来防止屏幕关闭。

```
public class ScreenLockTest
{
    public void ToggleScreenLock()
    {
        if (!ScreenLock.IsActive)
            ScreenLock.RequestActive();
        else
            ScreenLock.RequestRelease();
    }
}
```

API

- 屏幕锁定源代码
- 屏幕锁定 API 文档

Xamarin.Essentials: Secure Storage

2018/11/1 • • Edit Online

SecureStorage 类有助于安全地存储简单的键/值对。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 SecureStorage 功能，需要以下特定于平台的设置：

- Android
- iOS
- UWP

> **TIP**
> 应用的自动备份是 Android 6.0（API 级别 23）及更高版本的功能，可备份用户的应用数据（共享首选项、应用的内部存储中的文件和其他特定文件）。在新设备上重新安装或安装应用时，会还原数据。这可能会影响使用共享首选项（已备份但在还原时无法解密）的 `SecureStorage`。Xamarin.Essentials 可通过删除键（以便可以进行重置）自动处理这种情况，但你可以通过禁用自动备份来采取其他步骤。

启用或禁用备份

可以选择通过在 `AndroidManifest.xml` 文件中将 `android:allowBackup` 设置设为 false，为整个应用程序禁用自动备份。仅当计划按另一种方式还原数据时才建议使用此方法。

```
<manifest ... >
    ...
    <application android:allowBackup="false" ... >
        ...
    </application>
</manifest>
```

选择性备份

可以将自动备份配置为禁止备份特定内容。可以创建自定义规则集以禁止备份 `SecureStore` 项。

1. 在你的 AndroidManifest.xml 中设置 `android:fullBackupContent` 属性：

   ```
   <application ...
       android:fullBackupContent="@xml/auto_backup_rules">
   </application>
   ```

2. 在 Resources/xml 目录中创建名为 auto_backup_rules.xml 的新 XML 文件。然后设置以下内容，包括除 `SecureStorage` 以外的所有共享首选项：

```
<?xml version="1.0" encoding="utf-8"?>
<full-backup-content>
    <include domain="sharedpref" path="."/>
    <exclude domain="sharedpref" path="${applicationId}.xamarinessentials.xml"/>
</full-backup-content>
```

使用 Secure Storage

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

将给定密钥的值保存在安全存储中：

```
try
{
  await SecureStorage.SetAsync("oauth_token", "secret-oauth-token-value");
}
catch (Exception ex)
{
  // Possible that device doesn't support secure storage on device.
}
```

从安全存储中检索值：

```
try
{
  var oauthToken = await SecureStorage.GetAsync("oauth_token");
}
catch (Exception ex)
{
  // Possible that device doesn't support secure storage on device.
}
```

> **NOTE**
> 如果没有与所请求的密钥关联的值，则 `GetAsync` 将返回 `null`。

若要删除特定密钥，请调用：

```
SecureStorage.Remove("oauth_token");
```

若要删除所有密钥，请调用：

```
SecureStorage.RemoveAll();
```

平台实现细节

- Android
- iOS
- UWP

Android 密钥存储用来存储用于在将值保存到文件名为 **[你的应用包 ID].xamarinessentials** 的共享首选项中之前加密值的加密密钥。共享首选项文件中所使用的密钥是传递到 `SecureStorage` API 的密钥的 MD5 哈希。

API 级别 23 及更高版本

在较新的 API 级别上，AES 密钥是从 Android 密钥存储中获得的，并与 AES/GCM/NoPadding 密码结合使用以在将值存储在共享首选项文件中之前加密值。

API 级别 22 及更高版本

在较旧的 API 级别上，Android 密钥存储仅支持存储 RSA 密钥，这些密钥与 RSA/ECB/PKCS1Padding 密码结合使用以加密 AES 密钥（运行时随机生成），并且存储在密钥 SecureStorageKey 下的共享首选项文件中（如果尚未生成一个）。

SecureStorage 使用首选项 API，并遵循首选项文档中所述的相同数据持久性。如果设备从 API 级别 22 或更低级别升级到 API 级别 23 及更高版本，则将继续使用此类型的加密，除非卸载该应用或调用 RemoveAll。

限制

此 API 用于存储少量文本。如果尝试将其用于存储大量文本，则可能会降低性能。

API

- SecureStorage 源代码
- SecureStorage API 文档

Xamarin.Essentials: SMS

2018/11/1 • • Edit Online

SMS 类允许应用程序使用要发送到收件人的指定消息打开默认短信应用程序。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 SMS

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

SMS 功能通过调用 `ComposeAsync` 方法工作，该方法是包含消息收件人和消息正文（两者都是可选的）的 `SmsMessage`。

```
public class SmsTest
{
    public async Task SendSms(string messageText, string recipient)
    {
        try
        {
            var message = new SmsMessage(messageText, recipient);
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

此外，还可以向 `SmsMessage` 传入多个收件人：

```
public class SmsTest
{
    public async Task SendSms(string messageText, string[] recipients)
    {
        try
        {
            var message = new SmsMessage(messageText, recipients);
            await Sms.ComposeAsync(message);
        }
        catch (FeatureNotSupportedException ex)
        {
            // Sms is not supported on this device.
        }
        catch (Exception ex)
        {
            // Other error has occurred.
        }
    }
}
```

API

- SMS 源代码
- SMS API 文档

Xamarin.Essentials: Text-to-Speech

2018/11/1 • • Edit Online

TextToSpeech 类允许应用程序使用内置的文本到语音转换引擎回讲设备中的文本并查询引擎可以支持的可用语言。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用 Text-to-Speech

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

Text-to-Speech 通过调用具有文本和可选参数的 `SpeakAsync` 方法工作，并在完成语音样本后返回。

```
public async Task SpeakNowDefaultSettings()
{
    await TextToSpeech.SpeakAsync("Hello World");

    // This method will block until utterance finishes.
}

public void SpeakNowDefaultSettings2()
{
    TextToSpeech.SpeakAsync("Hello World").ContinueWith((t) =>
    {
        // Logic that will run after utterance finishes.

    }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

此方法采用可选的 `CancellationToken`，以便在语音样本启动时将其停止。

```
CancellationTokenSource cts;
public async Task SpeakNowDefaultSettings()
{
    cts = new CancellationTokenSource();
    await TextToSpeech.SpeakAsync("Hello World", cancelToken: cts.Token);

    // This method will block until utterance finishes.
}

public void CancelSpeech()
{
    if (cts?.IsCancellationRequested ?? false)
        return;

    cts.Cancel();
}
```

Text-to-Speech 会自动将同一线程中的语音请求加入队列。

```
bool isBusy = false;
public void SpeakMultiple()
{
    isBusy = true;
    Task.Run(async () =>
    {
        await TextToSpeech.SpeakAsync("Hello World 1");
        await TextToSpeech.SpeakAsync("Hello World 2");
        await TextToSpeech.SpeakAsync("Hello World 3");
        isBusy = false;
    });

    // or you can query multiple without a Task:
    Task.WhenAll(
        TextToSpeech.SpeakAsync("Hello World 1"),
        TextToSpeech.SpeakAsync("Hello World 2"),
        TextToSpeech.SpeakAsync("Hello World 3"))
        .ContinueWith((t) => { isBusy = false; }, TaskScheduler.FromCurrentSynchronizationContext());
}
```

语音设置

为了更好地控制如何使用可用于设置音量、音调和区域设置的 `SpeakSettings` 回讲音频。

```
public async Task SpeakNow()
{
    var settings = new SpeakSettings()
        {
            Volume = .75,
            Pitch = 1.0
        };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}
```

下面是这些参数的支持值：

参数	最低	最大值
音调	0	2.0
音量	0	1.0

语音区域设置

每个平台支持不同的区域设置，以便使用不同语言和重音回讲文本。平台具有用于指定区域设置的不同代码和方法，这就是 Xamarin.Essentials 为何提供跨平台 `Locale` 类以及使用 `GetLocalesAsync` 查询区域设置的方法的原因。

```
public async Task SpeakNow()
{
    var locales = await TextToSpeech.GetLocalesAsync();

    // Grab the first locale
    var locale = locales.FirstOrDefault();

    var settings = new SpeakSettings()
        {
            Volume = .75,
            Pitch = 1.0,
            Locale = locale
        };

    await TextToSpeech.SpeakAsync("Hello World", settings);
}
```

限制

- 如果跨多个线程调用语音样本队列，则不能予以保证。
- 背景音频播放未受到官方支持。

API

- TextToSpeech 源代码
- TextToSpeech API 文档

Xamarin.Essentials：版本跟踪

2018/11/10 • • Edit Online

VersionTracking 类使你能够检查应用程序版本和内部版本号以及查看其他信息，例如，此应用程序是第一次启动还是当前版本的第一次启动，以及获取之前的内部版本信息等。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

使用版本跟踪

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

首次使用 VersionTracking 类时，它将开始跟踪当前版本。每次加载时，必须仅在应用程序中提前调用 `Track` 以确保跟踪当前版本信息：

```
VersionTracking.Track();
```

调用初始 `Track` 后，可以读取版本信息：

```
// First time ever launched application
var firstLaunch = VersionTracking.IsFirstLaunchEver;

// First time launching current version
var firstLaunchCurrent = VersionTracking.IsFirstLaunchForCurrentVersion;

// First time launching current build
var firstLaunchBuild = VersionTracking.IsFirstLaunchForCurrentBuild;

// Current app version (2.0.0)
var currentVersion = VersionTracking.CurrentVersion;

// Current build (2)
var currentBuild = VersionTracking.CurrentBuild;

// Previous app version (1.0.0)
var previousVersion = VersionTracking.PreviousVersion;

// Previous app build (1)
var previousBuild = VersionTracking.PreviousBuild;

// First version of app installed (1.0.0)
var firstVersion = VersionTracking.FirstInstalledVersion;

// First build of app installed (1)
var firstBuild = VersionTracking.FirstInstalledBuild;

// List of versions installed (1.0.0, 2.0.0)
var versionHistory = VersionTracking.VersionHistory;

// List of builds installed (1, 2)
var buildHistory = VersionTracking.BuildHistory;
```

平台实现细节

所有版本信息均是使用 Xamarin.Essentials 中的 Preferences API 存储的，是以 [你的-应用-包-ID].xamarinessentials.versiontracking 为文件名存储的，并且遵循 Preferences 文档中概述的同一数据持久性。

API

- 版本跟踪源代码
- 版本跟踪 API 文档

Xamarin.Essentials: Vibration

2018/11/10 • • Edit Online

Vibration 类使你能够在所需的时间内启动和停止振动功能。

入门

若要开始使用此 API，请阅读 Xamarin.Essentials 的入门指南以确保在项目中正确安装和设置库。

若要访问 Vibration 功能，需要以下特定于平台的设置。

- Android
- iOS
- UWP

需要具有 Vibrate 权限，并且必须在 Android 项目中进行配置。可以通过以下方法添加此权限：

打开 Properties 文件夹下的 AssemblyInfo.cs 文件并添加：

```
[assembly: UsesPermission(Android.Manifest.Permission.Vibrate)]
```

或更新 Android 清单：

打开 Properties 文件夹下的 AndroidManifest.xml 文件，并在“manifest”节点内添加以下代码。

```
<uses-permission android:name="android.permission.VIBRATE" />
```

或右键单击 Android 项目并打开项目的属性。在“Android 清单”下找到“所需权限:”区域，然后选中“VIBRATE”权限。这样会自动更新 AndroidManifest.xml 文件。

使用 Vibration

在你的类中添加对 Xamarin.Essentials 的引用：

```
using Xamarin.Essentials;
```

可以通过设置所需的时间量或使用默认 500 毫秒来使用 Vibration 功能。

```
try
{
    // Use default vibration length
    Vibration.Vibrate();

    // Or use specified time
    var duration = TimeSpan.FromSeconds(1);
    Vibration.Vibrate(duration);
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

可以使用 `Cancel` 方法来请求取消使用设备振动：

```
try
{
    Vibration.Cancel();
}
catch (FeatureNotSupportedException ex)
{
    // Feature not supported on device
}
catch (Exception ex)
{
    // Other error has occurred.
}
```

平台差异

- Android
- iOS
- UWP

无平台差异。

API

- Vibration 源代码
- Vibration API 文档

Xamarin.Essentials：故障排除

2018/7/11 • • Edit Online

错误：为 Xamarin.Android.Support.Compat 检测到的版本冲突

更新 NuGet 包时，可能出现以下错误（或添加新的包）与使用 Xamarin.Essentials 的 Xamarin.Forms 项目：

```
NU1107: Version conflict detected for Xamarin.Android.Support.Compat. Reference the package directly from the
project to resolve this issue.
 MyApp -> Xamarin.Essentials 0.8.0-preview -> Xamarin.Android.Support.CustomTabs 27.0.2.1 ->
Xamarin.Android.Support.Compat (= 27.0.2.1)
 MyApp -> Xamarin.Forms 3.1.0.583944 -> Xamarin.Android.Support.v4 25.4.0.2 -> Xamarin.Android.Support.Compat
(= 25.4.0.2).
```

问题在于不匹配的两个 Nuget 依赖项。这可以通过手动添加特定版本的依赖项解析（在这种情况下**Xamarin.Android.Support.Compat**），可以支持这两个。

若要执行此操作，添加 NuGet 的手动冲突的源，并使用**版本**列表以选择特定版本。当前版本的 Xamarin.Android.Support.Compat 和 Xamarin.Android.Support.Core.Util NuGet 27.0.2.1 将解决此错误。

请参阅这篇博客文章有关详细信息和如何解决此问题的视频。

如果遇到任何问题或 bug 请报告其查找Xamarin.Essentials GitHub 存储库。

数据和云服务

2018/6/22 • • Edit Online

本指南将说明如何执行此操作和 Xamarin.Forms 应用程序可以使用 web 服务使用各种技术，实现。

在 Xamarin 平台上的跨平台 web 服务使用的简介，请参阅介绍了 Web 服务。

了解示例

本文提供了演示如何与另一个 web 服务进行通信的 Xamarin.Forms 示例应用程序的演练。涵盖的主题包括剖析应用程序、页、数据模型中，并调用 web 服务操作。

使用 Web 服务

本指南演示如何与另一个 web 服务以提供通信创建、读取、更新和删除 (CRUD) 到 Xamarin.Forms 应用程序的功能。涵盖的主题包括与通信ASMX 服务，WCF 服务，REST 服务，和Azure Mobile Apps。

对 Web 服务的访问进行身份验证

本指南说明如何将身份验证服务集成到 Xamarin.Forms 应用程序以使用户能够共享一个后端，同时仅有权访问他们自己的数据。涵盖的主题包括与 REST 服务中使用基本身份验证，使用 Xamarin.Auth 组件对 OAuth 标识提供程序进行身份验证，且使用内置身份验证机制提供Azure Mobile Apps。

将数据与 Web 服务同步

此文章介绍了如何向 Xamarin.Forms 应用程序添加脱机同步功能。脱机同步允许用户交互与移动应用程序、查看、添加或修改数据，甚至在没有网络连接。更改将存储在本地数据库中，并且一旦设备处于联机状态，可以与 web 服务同步所做的更改。

发送推送通知

本文演示如何向 Xamarin.Forms 应用程序添加推送通知。Azure 通知中心提供可缩放的推送基础结构用于发送移动推送通知从任意后端向任何移动平台，同时无后端不必与不同的平台通知系统通信的复杂性。

在云中存储文件

本文演示如何使用 Xamarin.Forms 将文本和二进制数据存储在 Azure 存储空间，以及如何访问数据。Azure 存储是一种可扩展的云存储解决方案，可以用于存储非结构化和结构化数据。

在云中搜索数据

本文演示如何使用 Microsoft Azure 搜索库将 Azure Search 集成到 Xamarin.Forms 应用程序。Azure 搜索是云服务，提供了索引和查询上载的数据的功能。这将删除基础结构要求和传统上与应用程序中实现搜索功能关联的搜索算法复杂性。

在文档数据库中存储数据

本指南演示如何使用 Azure Cosmos DB 标准.NET 客户端库来将 Azure Cosmos DB 文档数据库集成到 Xamarin.Forms 应用程序。Azure Cosmos DB 文档数据库是提供对 JSON 文档，提供快速、高度可用、可缩放数据库服务需要无缝缩放和全局复制的应用程序的较低的延迟访问的 NoSQL 数据库。

通过认知服务添加智能

本指南说明如何在 Xamarin.Forms 应用程序中使用的一些 Microsoft 认知服务 Api。认知服务是一套 Api、Sdk 和供开发人员可以通过添加功能，如面部识别、语音识别和语言理解，使其应用程序更智能的服务。

了解示例

2018/6/8 • • Edit Online

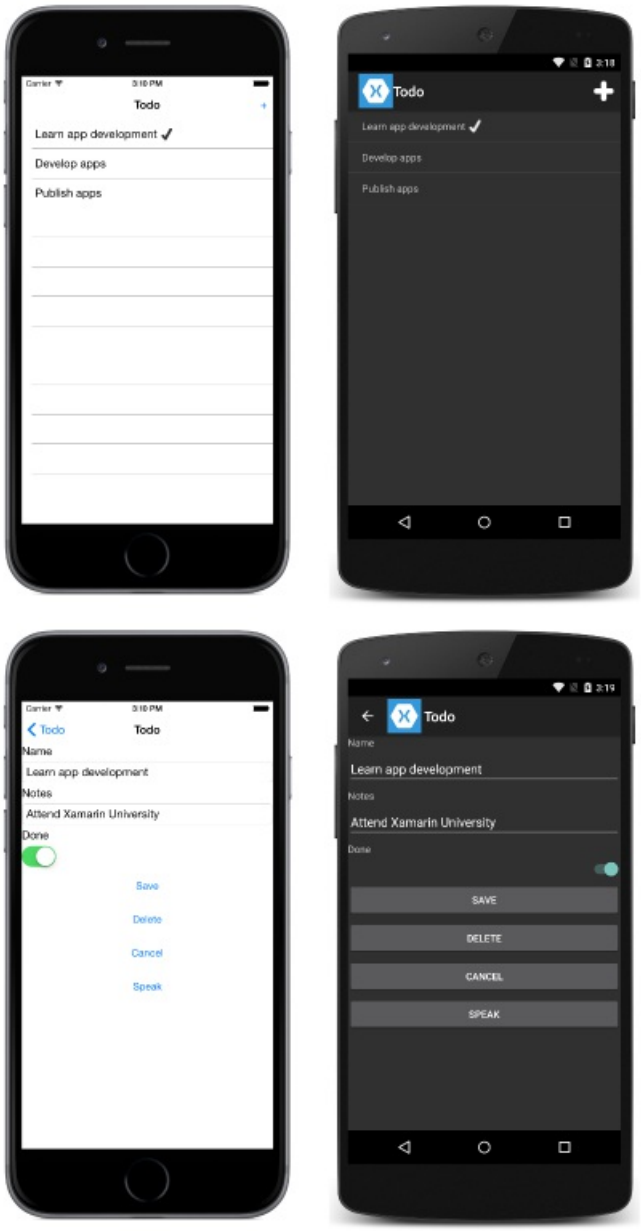
本主题提供的 Xamarin.Forms 示例应用程序演示如何与另一个 web 服务进行通信的演练。而每个 web 服务使用单独的示例应用程序，它们都功能上相似，共享通用类。

如下所述的示例待办事项列表应用程序用于演示如何访问不同类型的 web 服务后端使用 Xamarin.Forms。它提供功能：

- 查看任务的列表。
- 添加、编辑和删除任务。
- 设置任务的状态设置为完成。
- 该任务的名称和说明字段进行通信。

在所有情况下，这些任务存储在通过 web 服务访问后端。

当启动应用程序时，会显示页面，列出了从 web 服务检索任何任务，并允许用户创建新任务。单击任务导航到第二个页上，其中该任务可以编辑、保存、删除和读出应用程序。最终的应用程序如下所示：



本指南中的每个主题提供的下载链接*不同*演示特定类型的 web 服务后端的应用程序版本。下载与每个 web 服务样式相关的页上的相关的示例代码。

了解应用程序剖析

对于每个示例应用程序 PCL 项目中包含的三个主要文件夹：

文件夹	目标
数据	包含的类和接口可用于管理数据项目，并与 web 服务通信。在最低限度上，这包括 `TodoItemManager` 类，该类通过中的属性公开 `App` 类来调用 web 服务操作。
模型	包含应用程序的数据模型类。在最低限度上，这包括 `TodoItem` 类，该类建模的数据应用程序使用单个项。该文件夹还可以包含用于对用户数据的任何其他类。
视图	包含应用程序的页。这通常包括 `TodoListPage` 和 `TodoItemPage` 类和任何其他类，用于身份验证目的。

PCL 项目中的每个应用程序还包含一个重要的文件数：

文件	目标
Constants.cs	`Constants` 类，该类指定任何应用程序用于与 web 服务通信的常数。这些常量要求更新以访问您的个人的后端服务提供程序上创建。
ITextToSpeech.cs	`ITextToSpeech` 接口，它可以指定 `Speak` 方法必须由任何实现类。
Todo.cs	`App` 负责实例化的这两个将显示的每个平台上的应用程序的第一页的类和 `TodoItemManager` 用于调用 web 服务操作的类。

查看页

大多数示例应用程序包含至少两个页：

- **TodoListPage** – 此页显示的列表 `TodoItem` 实例和对勾图标如果 `TodoItem.Done` 属性是 `true` 。单击项导航到 `TodoItemPage` 。此外，通过单击创建新项 + 符号。
- **TodoItemPage** – 此页显示所选的详细信息 `TodoItem` ，并允许编辑、保存、删除和读出。

此外，某些示例应用程序包含用于管理用户身份验证过程的其他页。

对数据进行建模

每个示例应用程序使用 `TodoItem` 类模型显示并发送到 web 服务进行存储的数据。以下代码示例演示 `TodoItem` 类：

```
public class TodoItem
{
    public string ID { get; set; }
    public string Name { get; set; }
    public string Notes { get; set; }
    public bool Done { get; set; }
}
```

`ID` 属性用于唯一标识每个 `TodoItem` 实例，并且每个 web 服务的情况下用于标识要更新或删除数据。

调用 Web 服务操作

通过访问 web 服务操作 `TodoItemManager` 类和类的实例可以通过访问 `App.TodoManager` 属性。 `TodoItemManager` 类提供下列方法来调用 web 服务操作：

- **GetTasksAsync** – 此方法用于填充 `ListView` 控制上 `TodoListPage` 与 `TodoItem` 从 web 服务检索的实例。
- **SaveTaskAsync** – 此方法用于创建或更新 `TodoItem` web 服务的实例。
- **DeleteTaskAsync** – 此方法用于删除 `TodoItem` web 服务的实例。

此外，某些示例应用程序包含中的其他方法 `TodoItemManager` 类，用于管理用户身份验证过程。

而不是直接调用 web 服务操作 `TodoItemManager` 方法调用注入的依赖类上方法 `TodoItemManager` 构造函数。例如，一个示例应用程序插入 `RestService` 类到 `TodoItemManager` 构造函数，以提供使用 REST Api 访问数据的实现。

将文本到语音转换

大多数示例应用程序包含用于进行通信的值的文本到语音转换 (TTS) 功能 `TodoItem.Name` 和 `TodoItem.Notes` 属性。这通过实现 `OnSpeakActivated` 中的事件处理程序 `TodoItemPage` 类，如下面的代码示例中所示：

```
void OnSpeakActivated (object sender, EventArgs e)
{
    var todoItem = (TodoItem)BindingContext;
    App.Speech.Speak(todoItem.Name + " " + todoItem.Notes);
}
```

此方法只需调用 `Speak` 由特定于平台的实现的方法 `Speech` 类。每个 `Speech` 类实现 `ITextToSpeech` 接口，并特定于平台的启动代码创建的实例 `Speech` 类可以通过访问 `App.Speech` 属性。

总结

本主题提供用于演示如何与另一个 web 服务进行通信的 Xamarin.Forms 示例应用程序的演练。而每个 web 服务使用单独的示例应用程序，它们都基于相同的用户界面和业务逻辑上文所述-仅 web 服务数据存储机制是不同。

相关链接

- ASMX 版本（示例）
- WCF 版本（示例）
- REST 版本（示例）
- Azure 版本（示例）

使用 Web 服务

2018/6/22 • • Edit Online

此指南演示如何与另一个 web 服务以提供通信创建、读取、更新和删除 (CRUD) 到 Xamarin.Forms 应用程序的功能。涵盖的主题包括与 ASMX 服务、WCF 服务、REST 服务和 Azure Mobile Apps 通信。

使用 ASP.NET Web 服务 (ASMX)

ASP.NET Web 服务 (ASMX) 提供能够生成使用简单对象访问协议 (SOAP) 通过 HTTP 发送消息的 web 服务。SOAP 是一个独立于平台的独立于语言的协议用于构建和访问 web 服务。ASMX 服务的使用者不需要知道任何有关平台、对象模型或用于实现服务的编程语言。它们只需了解如何发送和接收 SOAP 消息。本文演示如何使用 Xamarin.Forms 应用程序从一个 ASMX web 服务。

使用 Windows Communication Foundation (WCF) Web 服务

WCF 是 Microsoft 的统一的框架，用于构建面向服务的应用程序。它允许开发人员生成安全、可靠、事务处理，且可互操作的分布式应用程序。ASP.NET Web 服务 (ASMX) 和 WCF，之间有差异，但务必了解 WCF 支持 ASMX 提供的相同功能 — 通过 HTTP 的 SOAP 消息。本文演示如何使用 Xamarin.Forms 应用程序从 WCF SOAP 服务。

使用 rest 样式 Web 服务

具象状态传输 (REST) 是用于生成 web 服务的架构样式。REST 请求都通过 HTTP 使用 web 浏览器使用来检索网页并将数据发送到服务器的同一 HTTP 谓词。本文演示如何使用 rest 样式 web 服务从 Xamarin.Forms 应用程序。

使用 Azure 移动应用

Azure 移动应用，可以使用可缩放的后端移动身份验证、脱机同步和推送通知的支持与托管在 Azure App Service 中开发应用。本文中，这仅适用于使用 Node.js 后端的 Azure Mobile Apps，说明如何查询、插入、更新和删除数据存储在 Azure Mobile Apps 实例中的表。

相关链接

- Web 服务简介
- 异步支持概述

使用 ASP.NET Web 服务 (ASMX)

2018/6/9 • • Edit Online

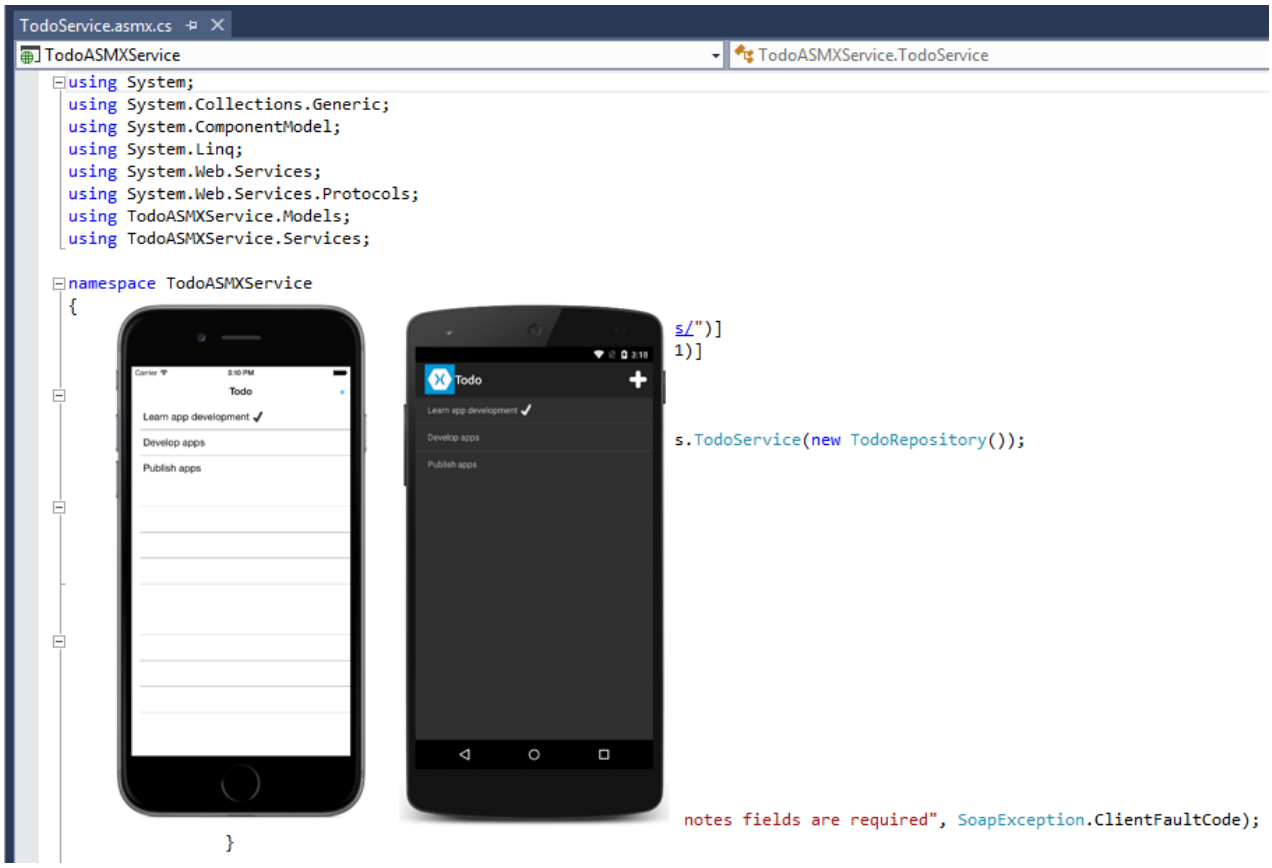
ASMX 能够生成使用简单对象访问协议 (SOAP) 发送消息的 web 服务。SOAP 是一个独立于平台的独立于语言的协议用于构建和访问 web 服务。ASMX 服务的使用者不需要知道任何有关平台、对象模型或用于实现服务的编程语言。它们只需了解如何发送和接收 SOAP 消息。本文演示如何使用 Xamarin.Forms 应用程序中的 ASMX SOAP 服务。

SOAP 消息是 XML 文档包含以下元素：

- 名为的根元素*信封*标识 XML 文档作为 SOAP 消息。
- 一个可选*标头*包含特定于应用程序的信息，例如身份验证数据的元素。如果*标头*存在元素，则它必须是第一个子元素*信封*元素。
- 必需*正文*包含适用于接收方的 SOAP 消息的元素。
- 一个可选*错误*元素，用于指示错误消息。如果*错误*元素存在，它必须是子元素的*正文*元素。

SOAP 可以对许多传输协议，包括 HTTP、SMTP、TCP 和 UDP 进行操作。但是，ASMX 服务可以仅通过 HTTP 操作。Xamarin 平台支持标准 SOAP 1.1 实现通过 HTTP，并且这会包含对许多标准 ASMX 服务配置的支持。

附带的示例应用程序的自述文件中找不到设置 ASMX 服务说明。但是，当运行示例应用程序时，它将连接到 Xamarin 托管 ASMX 服务，用于提供只读访问数据，如下面的屏幕截图中所示：



> **NOTE**
> 在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

使用 Web 服务

ASMX 服务提供以下操作：

操作	描述	参数
GetTodoItems	获取待办事项的列表	
CreateTodoItem	创建新的待办事项	XML 序列化 TodoItem
EditTodoItem	更新待办事项	XML 序列化 TodoItem
DeleteTodoItem	删除待办事项	XML 序列化 TodoItem

有关应用程序中使用的数据模型的详细信息，请参阅对数据进行建模。

> **NOTE**
> 示例应用程序使用 Xamarin 托管 ASMX 服务，用于提供对 web 服务的只读访问权限。因此，创建、更新和删除数据的操作不会更改应用程序中使用的数据。但是，ASMX 服务的可承载版本位于**TodoASMXService**随附的示例应用程序中的文件夹。此可承载版本的完整 ASMX 服务允许创建、更新、读取，和删除的数据访问权限。

A *代理*必须生成能够使用 ASMX 服务，允许应用程序连接到服务。代理是通过使用以定义的方法和关联的服务配置的服务元数据构造的。由 web 服务生成 Web 服务描述语言 (WSDL) 文档形式公开此元数据。将 web 服务的 web 引用添加到特定于平台的项目生成代理。

生成的代理类提供用于使用 web 服务使用的异步编程模型 (APM) 设计模式的方法。在此模式中异步操作实现这两个方法名为*BeginOperationName*和*EndOperationName*，其开始和结束异步操作。

*BeginOperationName*方法开始异步操作并返回一个对象，实现 `IAsyncResult` 接口。在调用*BeginOperationName*，应用程序可以继续在调用的线程上执行指令，同时异步操作在有线程池线程上的发生。

每次调用*BeginOperationName*，应用程序还应调用*EndOperationName*来获取该操作的结果。返回值*EndOperationName*同步 web 服务方法返回的类型相同。例如，`EndGetTodoItems` 方法返回的集合 `TodoItem` 实例。*EndOperationName*方法还包括 `IAsyncResult` 应设置为相应地调用所返回的实例的参数*BeginOperationName*方法。

任务并行库 (TPL) 可以简化通过封装在同一个异步操作来使用的 APM begin/end 方法对的过程 `Task` 对象。此封装提供的多个重载的 `TaskFactory.FromAsync` 方法。

APM 有关的详细信息请参阅异步编程模型和TPL 和传统.NET Framework 异步编程MSDN 上。

创建 TodoService 对象

生成的代理类提供 `TodoService` 类，该类用于通过 HTTP 与 ASMX 服务进行通信。为了异步操作从 URI 标识的服务实例来调用 web 服务方法，它提供功能。有关异步操作的详细信息，请参阅异步支持概述。

`TodoService` 实例在类级别声明的以便对象存活，只要该应用程序需要使用 ASMX 服务，如下面的代码示例中所

示：

```
public class SoapService : ISoapService
{
  ASMXService.TodoService asmxService;
  ...

  public SoapService ()
  {
    asmxService = new ASMXService.TodoService (Constants.SoapUrl);
  }
  ...
}
```

`TodoService` 构造函数采用一个可选的字符串参数，指定 ASMX 服务实例的 URL。这使应用程序连接到 ASMX 服务的不同实例，前提是有多个已发布的实例。

创建数据传输对象

示例应用程序使用 `TodoItem` 模型数据的类。若要存储 `TodoItem` 中 web 服务必须首先将转换为生成的代理项 `TodoItem` 类型。这通过实现 `ToASMXServiceTodoItem` 方法，如下面的代码示例中所示：

```
ASMXService.TodoItem ToASMXServiceTodoItem (TodoItem item)
{
  return new ASMXService.TodoItem {
    ID = item.ID,
    Name = item.Name,
    Notes = item.Notes,
    Done = item.Done
  };
}
```

此方法只需创建一个新 `ASMService.TodoItem` 实例，并将每个属性设置为相同的属性从 `TodoItem` 实例。

同样，从 web 服务检索数据时，必须将它转换从生成的代理 `TodoItem` 键入到 `TodoItem` 实例。这实现的 `FromASMXServiceTodoItem` 方法，如下面的代码示例中所示：

```
static TodoItem FromASMXServiceTodoItem (ASMXService.TodoItem item)
{
  return new TodoItem {
    ID = item.ID,
    Name = item.Name,
    Notes = item.Notes,
    Done = item.Done
  };
}
```

此方法只需从生成的代理中检索数据 `TodoItem` 键入，并将其设置在新创建 `TodoItem` 实例。

检索数据

`TodoService.BeginGetTodoItems` 和 `TodoService.EndGetTodoItems` 方法用于调用 `GetTodoItems` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
  ...
  var todoItems = await Task.Factory.FromAsync<ASMXService.TodoItem[]> (
    todoService.BeginGetTodoItems,
    todoService.EndGetTodoItems,
    null,
    TaskCreationOptions.None);

  foreach (var item in todoItems) {
    Items.Add (FromASMXServiceTodoItem (item));
  }
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoService.EndGetTodoItems` 方法一次 `TodoService.BeginGetTodoItems` 方法完成时，与 `null` 表示没有数据传递到参数 `BeginGetTodoItems` 委托。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

`TodoService.EndGetTodoItems` 方法返回的数组 `ASMXService.TodoItem` 实例，然后转换为 `List` 的 `TodoItem` 显示的实例。

创建数据

`TodoService.BeginCreateTodoItem` 和 `TodoService.EndCreateTodoItem` 方法用于调用 `CreateTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  ...
  var todoItem = ToASMXServiceTodoItem (item);
  ...
  await Task.Factory.FromAsync (
    todoService.BeginCreateTodoItem,
    todoService.EndCreateTodoItem,
    todoItem,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoService.EndCreateTodoItem` 方法一次 `TodoService.BeginCreateTodoItem` 方法完成时，与 `todoItem` 参数被传递到的数据 `BeginCreateTodoItem` 委托来指定 `TodoItem` 由 web 服务来创建。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `SoapException` 未能创建 `TodoItem`，这由应用程序。

更新数据

`TodoService.BeginEditTodoItem` 和 `TodoService.EndEditTodoItem` 方法用于调用 `EditTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  ...
  var todoItem = ToASMXServiceTodoItem (item);
  ...
  await Task.Factory.FromAsync (
    todoService.BeginEditTodoItem,
    todoService.EndEditTodoItem,
    todoItem,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoService.EndEditTodoItem` 方法一次 `TodoService.BeginCreateTodoItem` 方法完成时，与 `todoItem` 参数被传递到的数据 `BeginEditTodoItem` 委托来指定 `TodoItem` web 服务更新。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `SoapException` 未能找到或更新 `TodoItem`，这由应用程序。

删除数据

`TodoService.BeginDeleteTodoItem` 和 `TodoService.EndDeleteTodoItem` 方法用于调用 `DeleteTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task DeleteTodoItemAsync (string id)
{
  ...
  await Task.Factory.FromAsync (
    todoService.BeginDeleteTodoItem,
    todoService.EndDeleteTodoItem,
    id,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoService.EndDeleteTodoItem` 方法一次 `TodoService.BeginDeleteTodoItem` 方法完成时，与 `id` 参数被传递到的数据 `BeginDeleteTodoItem` 委托来指定 `TodoItem` 要删除 web 服务。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `SoapException` 未能找到或删除 `TodoItem`，这由应用程序。

总结

这篇文章演示了如何使用 Xamarin.Forms 应用程序从一个 ASMX web 服务。ASMX 能够构建通过 HTTP 发送消息使用 SOAP 的 web 服务。ASMX 服务的使用者不需要知道任何有关平台、对象模型或用于实现服务的编程语言。它们只需了解如何发送和接收 SOAP 消息。

相关链接

- TodoASMX（示例）
- IAsyncResult

使用 Windows Communication Foundation (WCF) Web 服务

2018/6/9 • • Edit Online

WCF 是 Microsoft 的统一的框架，用于构建面向服务的应用程序。它允许开发人员生成安全、可靠、事务处理，且可互操作的分布式应用程序。本文演示如何使用从 Xamarin.Forms 应用程序的 WCF 简单对象访问协议 (SOAP) 服务。

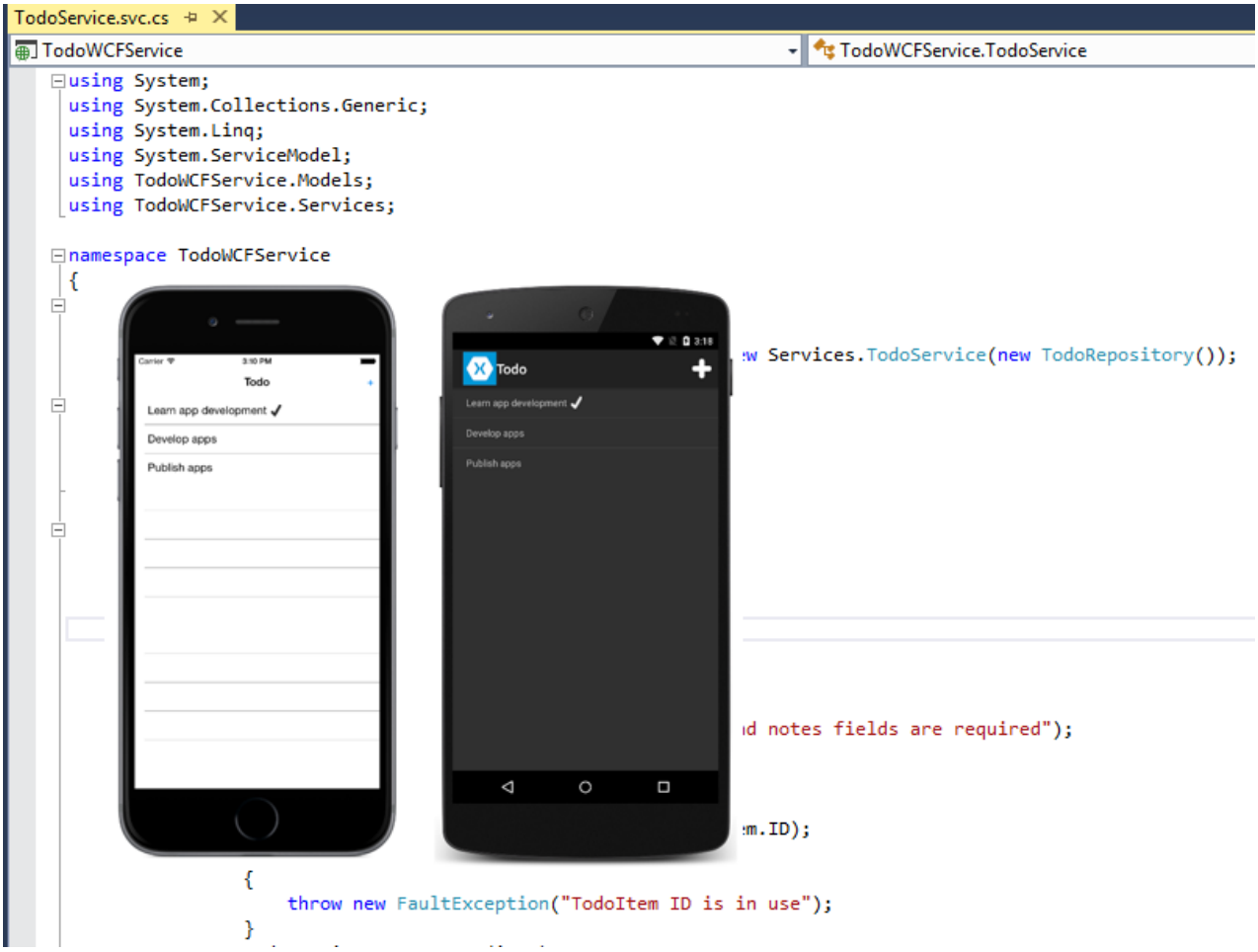
WCF 描述了与各种不同的协定包括以下服务：

- **数据协定**– 定义构成一条消息中的内容的基础的数据结构。
- **消息协定搭配**– 撰写从现有数据协定的消息。
- 错误协定– 允许指定的自定义 SOAP 错误。
- **服务协定**– 指定服务支持的操作和消息所需的每个操作与之进行交互。它们还指定可以与每个服务操作相关联的任何自定义错误行为。

ASP.NET Web 服务 (ASMX) 和 WCF，之间的差异，但是务必了解 WCF 支持相同的功能，提供的 ASMX 服务 – 通过 HTTP 的 SOAP 消息。有关使用 ASMX 服务的详细信息，请参阅使用 ASP.NET Web 服务 (ASMX)。

一般情况下，Xamarin 平台支持相同的客户端将一部分附带 Silverlight 运行时的 WCF。这包括 WCF 的最常见的编码和协议实现 — 通过 HTTP 文本编码 SOAP 消息传输协议使用 `BasicHttpBinding` 类。此外，WCF 支持需要仅在一个用于生成代理的 Windows 环境中可用的工具的用法。

附带的示例应用程序的自述文件中找不到 WCF 服务的设置说明。但是，运行示例应用程序时它将连接到 Xamarin 承载的 WCF 服务提供只读访问数据，如下面的屏幕截图中所示：



> **NOTE**
> 在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

使用 Web 服务

WCF 服务提供以下操作：

操作	描述	参数
GetTodoItems	获取待办事项的列表	
CreateTodoItem	创建新的待办事项	XML 序列化 TodoItem
EditTodoItem	更新待办事项	XML 序列化 TodoItem
DeleteTodoItem	删除待办事项	XML 序列化 TodoItem

有关应用程序中使用的数据模型的详细信息，请参阅对数据进行建模。

> **NOTE**
> 示例应用程序使用 Xamarin 承载的 WCF 服务，提供对 web 服务的只读访问权限。因此，创建、更新和删除数据的操作不会更改应用程序中使用的数据。但是，ASMX 服务的可承载版本位于**TodoWCFService**随附的示例应用程序中的文件夹。此可承载的 WCF 服务允许完整版本创建、更新、读取，和删除的数据访问权限。

A *代理必*须生成要使用 WCF 服务，它允许应用程序连接到服务。代理是通过使用以定义的方法和关联的服务配置的服务元数据构造的。由 web 服务生成 Web 服务描述语言 (WSDL) 文档形式公开此元数据。可以使用在 Visual Studio 2017 Microsoft WCF Web 服务引用提供程序将 web 服务的服务引用添加到.NET 标准库生成代理。创建在 Visual Studio 2017 中使用 Microsoft WCF Web 服务引用提供程序的代理的替代方法是使用 ServiceModel 元数据实用工具 (svcutil.exe)。有关详细信息，请参阅ServiceModel 元数据实用工具 (Svcutil.exe)。

生成的代理类提供用于使用使用异步编程模型 (APM) 设计模式的 web 服务方法。在此模式中，异步操作实现这两个方法名为*BeginOperationName*和*EndOperationName*，其开始和结束异步操作。

*BeginOperationName*方法开始异步操作并返回一个对象，实现 `IAsyncResult` 接口。在调用*BeginOperationName*，应用程序可以继续在调用的线程上执行指令，同时异步操作在有线程池线程上的发生。

每次调用*BeginOperationName*，应用程序还应调用*EndOperationName*来获取该操作的结果。返回值*EndOperationName*同步 web 服务方法返回的类型相同。例如，`EndGetTodoItems` 方法返回的集合 `TodoItem` 实例。*EndOperationName*方法还包括 `IAsyncResult` 应设置为相应地调用所返回的实例的参数*BeginOperationName*方法。

任务并行库 (TPL) 可以简化通过封装在同一个异步操作来使用的 APM begin/end 方法对的过程 `Task` 对象。此封装提供的多个重载的 `TaskFactory.FromAsync` 方法。

APM 有关的详细信息请参阅异步编程模型和TPL 和传统.NET Framework 异步编程MSDN 上。

创建 TodoServiceClient 对象

生成的代理类提供 `TodoServiceClient` 类，该类用于通过 HTTP 与 WCF 服务进行通信。为了异步操作从 URI 标识的服务实例来调用 web 服务方法，它提供功能。有关异步操作的详细信息，请参阅异步支持概述。

`TodoServiceClient` 实例在类级别声明的以便对象存活，只要该应用程序需要使用 WCF 服务，如下面的代码示例中所示：

```
public class SoapService : ISoapService
{
  ITodoService todoService;
  ...

  public SoapService ()
  {
    todoService = new TodoServiceClient (
      new BasicHttpBinding (),
      new EndpointAddress (Constants.SoapUrl));
  }
  ...
}
```

`TodoServiceClient` 实例配置和绑定信息和终结点地址。绑定用于指定传输、编码和协议详细信息所需的应用程序和服务相互通信。`BasicHttpBinding` 指定，将通过 HTTP 传输协议发送文本编码 SOAP 消息。指定终结点地址后用应用程序连接到 WCF 服务的不同实例，前提是有多个已发布的实例。

有关配置服务引用的详细信息，请参阅配置服务引用。

创建数据传输对象

示例应用程序使用 `TodoItem` 模型数据的类。若要存储 `TodoItem` 中 web 服务必须首先将转换为生成的代理项 `TodoItem` 类型。这通过实现 `ToWCFServiceTodoItem` 方法，如下面的代码示例中所示：

```
TodoWCFService.TodoItem ToWCFServiceTodoItem (TodoItem item)
{
  return new TodoWCFService.TodoItem {
    ID = item.ID,
    Name = item.Name,
    Notes = item.Notes,
    Done = item.Done
  };
}
```

此方法只需创建一个新 `TodoWCFService.TodoItem` 实例，并将每个属性设置为相同的属性从 `TodoItem` 实例。

同样，从 web 服务检索数据时，必须将它转换从生成的代理 `TodoItem` 键入到 `TodoItem` 实例。这实现的 `FromWCFServiceTodoItem` 方法，如下面的代码示例中所示：

```
static TodoItem FromWCFServiceTodoItem (TodoWCFService.TodoItem item)
{
  return new TodoItem {
    ID = item.ID,
    Name = item.Name,
    Notes = item.Notes,
    Done = item.Done
  };
}
```

此方法只需从生成的代理中检索数据 `TodoItem` 键入，并将其设置在新创建 `TodoItem` 实例。

检索数据

`TodoServiceClient.BeginGetTodoItems` 和 `TodoServiceClient.EndGetTodoItems` 方法用于调用 `GetTodoItems` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
  ...
  var todoItems = await Task.Factory.FromAsync <ObservableCollection<TodoWCFService.TodoItem>> (
    todoService.BeginGetTodoItems,
    todoService.EndGetTodoItems,
    null,
    TaskCreationOptions.None);

  foreach (var item in todoItems) {
    Items.Add (FromWCFServiceTodoItem (item));
  }
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoServiceClient.EndGetTodoItems` 方法一次 `TodoServiceClient.BeginGetTodoItems` 方法完成时，与 `null` 表示没有数据传递到参数 `BeginGetTodoItems` 委托。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

`TodoServiceClient.EndGetTodoItems` 方法返回 `ObservableCollection` 的 `TodoWCFService.TodoItem` 实例，然后转换为 `List` 的 `TodoItem` 显示的实例。

创建数据

`TodoServiceClient.BeginCreateTodoItem` 和 `TodoServiceClient.EndCreateTodoItem` 方法用于调用 `CreateTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  ...
  var todoItem = ToWCFServiceTodoItem (item);
  ...
  await Task.Factory.FromAsync (
    todoService.BeginCreateTodoItem,
    todoService.EndCreateTodoItem,
    todoItem,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoServiceClient.EndCreateTodoItem` 方法一次 `TodoServiceClient.BeginCreateTodoItem` 方法完成时，与 `todoItem` 参数被传递到的数据 `BeginCreateTodoItem` 委托来指定 `TodoItem` 由 web 服务来创建。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `FaultException` 未能创建 `TodoItem`，这由应用程序。

更新数据

`TodoServiceClient.BeginEditTodoItem` 和 `TodoServiceClient.EndEditTodoItem` 方法用于调用 `EditTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  ...
  var todoItem = ToWCFServiceTodoItem (item);
  ...
  await Task.Factory.FromAsync (
    todoService.BeginEditTodoItem,
    todoService.EndEditTodoItem,
    todoItem,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoServiceClient.EndEditTodoItem` 方法一次 `TodoServiceClient.BeginCreateTodoItem` 方法完成时，与 `todoItem` 参数被传递到的数据 `BeginEditTodoItem` 委托来指定 `TodoItem` web 服务更新。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `FaultException` 未能找到或更新 `TodoItem`，这由应用程序。

删除数据

`TodoServiceClient.BeginDeleteTodoItem` 和 `TodoServiceClient.EndDeleteTodoItem` 方法用于调用 `DeleteTodoItem` web 服务所提供的操作。这些异步方法封装在 `Task` 对象，如下面的代码示例中所示：

```
public async Task DeleteTodoItemAsync (string id)
{
  ...
  await Task.Factory.FromAsync (
    todoService.BeginDeleteTodoItem,
    todoService.EndDeleteTodoItem,
    id,
    TaskCreationOptions.None);
  ...
}
```

`Task.Factory.FromAsync` 方法创建 `Task` 执行 `TodoServiceClient.EndDeleteTodoItem` 方法一次 `TodoServiceClient.BeginDeleteTodoItem` 方法完成时，与 `id` 参数被传递到的数据 `BeginDeleteTodoItem` 委托来指定 `TodoItem` 要删除 web 服务。最后，值 `TaskCreationOptions` 枚举指定应使用的创建和执行的任务的默认行为。

Web 服务引发 `FaultException` 未能找到或删除 `TodoItem`，这由应用程序。

总结

这篇文章演示了如何使用 Xamarin.Forms 应用程序从 WCF SOAP 服务。一般情况下，Xamarin 平台支持相同的客户端将一部分附带 Silverlight 运行时的 WCF。这包括 WCF 的最常见的编码和协议实现 — 通过 HTTP 文本编码 SOAP 消息传输协议使用 `BasicHttpBinding` 类。此外，WCF 支持需要仅在一个用于生成代理的 Windows 环境中可用的工具的用法。

相关链接

- TodoWCF（示例）
- IAsyncResult

使用 rest 样式 Web 服务

2018/6/8 • • Edit Online

将 web 服务集成到应用程序是一个常用方案。本文演示如何使用 rest 样式 web 服务从 Xamarin.Forms 应用程序。

具象状态传输 (REST) 是用于生成 web 服务的架构样式。REST 请求都通过 HTTP 使用 web 浏览器使用来检索网页并将数据发送到服务器的同一 HTTP 谓词。谓词是：

- 获取– 此操作用于从 web 服务中检索数据。
- **POST** – 此操作用于在 web 服务上创建数据的新项。
- **PUT** – 使用此操作更新某个项 web 服务上的数据。
- **修补程序**– 使用此操作通过描述有关应如何修改项的一组的说明更新 web 服务上的数据的某个项。在示例应用程序未使用此谓词。
- **删除**– 使用此操作删除某项 web 服务上的数据。

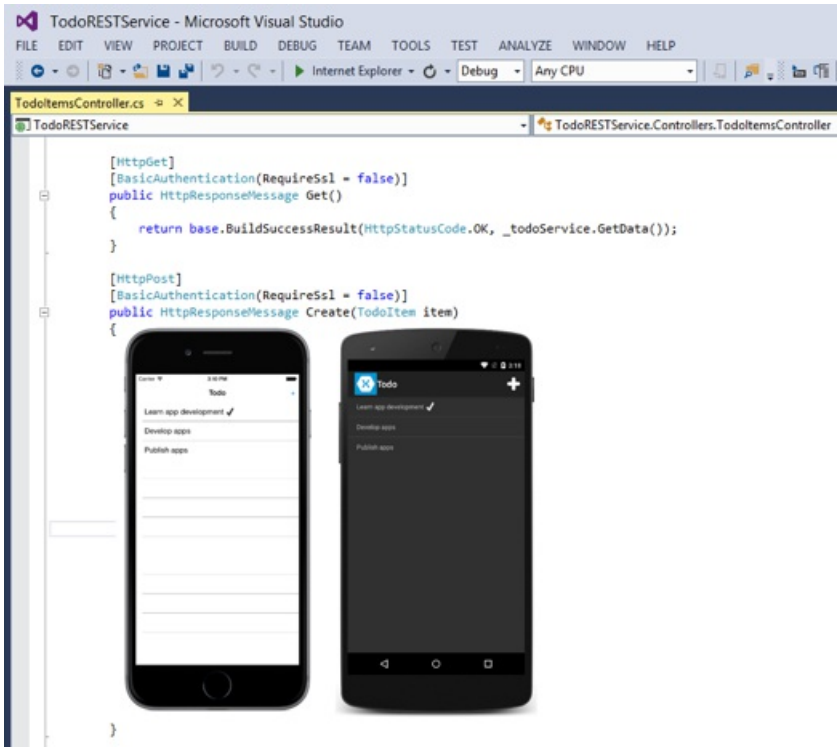
遵守 REST 的 Api 调用 RESTful Api 和使用定义的 web 服务：

- 基 URI。
- HTTP 方法，如 GET、POST、PUT、PATCH 或 DELETE。
- 数据，如 JavaScript 对象表示法 (JSON) 媒体类型。

RESTful web 服务通常使用 JSON 消息来将数据返回到客户端。JSON 是一种基于文本的数据交换格式发送数据时，在生成 compact 负载，这会导致降低带宽要求。示例应用程序使用的开放源代码NewtonSoft JSON.NET 库进行序列化和反序列化消息。

REST 的简单性已帮助使其用于访问移动应用程序中的 web 服务的主要方法。

附带的示例应用程序的自述文件中找不到 REST 服务的设置说明。但是，当运行示例应用程序时，它将连接到 Xamarin 托管 REST 服务，它提供只读访问数据，如下面的屏幕截图中所示：



NOTE

在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。

如果不能使用 ATS 可以选择退出的**HTTPS**协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

使用 Web 服务

REST 服务使用 ASP.NET Core 编写，并提供以下操作：

操作	HTTP 方法	相对 URI	参数
获取待办事项的列表	GET	/api/todoitems /	
创建新的待办事项	发布	/api/todoitems /	JSON 格式的 TodoItem
更新待办事项	PUT	/api/todoitems /	JSON 格式的 TodoItem
删除待办事项	DELETE	/api/todoitems / {id}	

其中包含 Uri 的大多数包括 `TodoItem` 路径中的 ID。例如，若要删除 `TodoItem` 其 ID 为 `6bb8a868-dba1-4f1a-93b7-24ebce87e243`，客户端发送 DELETE 请求到 `http://hostname/api/todoitems/6bb8a868-dba1-4f1a-93b7-24ebce87e243`。有关示例应用程序中使用的数据模型的详细信息，请参阅对数据进行建模。

当 Web API 框架收到请求时它将请求路由到的操作。这些操作是只是公共方法中的 `TodoItemsController` 类。框架使用的路由表来确定要在响应请求，下面的代码示例中所示调用的操作：

```
config.Routes.MapHttpRoute(
    name: "TodoItemsApi",
    routeTemplate: "api/{controller}/{id}",
    defaults: new { controller="todoitems", id = RouteParameter.Optional }
);
```

路由表包含一个路由模板中，和 Web API 框架接收 HTTP 请求时，它尝试匹配对路由表中的路由模板的 URI。如果匹配无法找到路由，客户端收到 404（找不到）错误。如果找到匹配的路由，则 Web API 选择控制器和操作，如下所示：

- 若要查找控制器，Web API 向"控制器"的值 *{controller}* 变量。
- 若要查找操作，Web API 审视的 HTTP 方法，并考察使用相同的 HTTP 方法作为特性修饰的控制器操作。
- *{Id}* 占位符变量映射到操作参数。

REST 服务使用基本身份验证。有关详细信息请参阅RESTful web 服务进行身份验证。有关 ASP.NET Web API 路由的详细信息，请参阅路由在 ASP.NET Web API 中ASP.NET 网站上。有关生成使用 ASP.NET Core 的 REST 服务的详细信息，请参阅本机移动应用程序创建后端服务。

> **NOTE**
> 示例应用程序使用 Xamarin 托管 REST 服务，提供对 web 服务的只读访问权限。因此，创建、更新和删除数据的操作不会更改应用程序中使用的数据。但是，REST 服务的可承载版本位于**TodoRESTService**中随附的文件夹示例代码。如果你自行托管 REST 服务，它允许完全创建、更新、读取和删除的数据访问权限。

`HttpClient` 类用来发送和接收通过 HTTP 请求。它提供了用于发送 HTTP 请求的功能，并接收从 URI 的 HTTP 响应标识资源。每个请求将作为异步操作发送。有关异步操作的详细信息，请参阅异步支持概述。

`HttpResponseMessage` 类表示进行 HTTP 请求之后，从 web 服务收到 HTTP 响应消息。它包含有关响应，包括状态代码、标头，以及任何正文的信息。`HttpContent` 类表示的 HTTP 正文和内容标头，如 `Content-Type` 和 `Content-Encoding`。可以使用任一读取内容 `ReadAs` 方法，如 `ReadAsStringAsync` 和 `ReadAsByteArrayAsync` 根据数据的格式。

创建 HTTPClient 对象

`HttpClient` 实例在类级别声明的以便对象存活，只要该应用程序需要发出 HTTP 请求，如下面的代码示例中所示：

```
public class RestService : IRestService
{
  HttpClient client;
  ...

  public RestService ()
  {
    client = new HttpClient ();
    client.MaxResponseContentBufferSize = 256000;
  }
  ...
}
```

`HttpClient.MaxResponseContentBufferSize` 属性用于指定的最大读取 HTTP 响应消息中的内容时要缓冲的字节数。此属性的默认大小是一个整数的最大大小。因此，该属性设置为较小的值，作为安全措施，以限制的应用程序将接受作为从 web 服务响应的数据量。

检索数据

`HttpClient.GetAsync` 方法用于将 GET 请求发送到 URI，指定 web 服务，然后接收响应从 web 服务，如下面的代码示例中所示：

```
public async Task<List<TodoItem>> RefreshDataAsync ()
{
  ...
  // RestUrl = http://developer.xamarin.com:8081/api/todoitems/
  var uri = new Uri (string.Format (Constants.RestUrl, string.Empty));
  ...
  var response = await client.GetAsync (uri);
  if (response.IsSuccessStatusCode) {
      var content = await response.Content.ReadAsStringAsync ();
      Items = JsonConvert.DeserializeObject <List<TodoItem>> (content);
  }
  ...
}
```

REST 服务发送的 HTTP 状态代码 `HttpResponseMessage.IsSuccessStatusCode` 属性，以指示 HTTP 请求是否成功。对于此操作的 REST 服务响应，表示请求成功，所请求的信息包含在响应中发送 HTTP 状态代码 200（正常）。

如果 HTTP 操作成功，读取响应的内容时，用于显示。`HttpResponseMessage.Content` 属性表示的 HTTP 响应的内容和 `HttpContent.ReadAsStringAsync` 方法以异步方式将 HTTP 内容写入字符串。此内容然后转换为 JSON 从 `List` 的

`TodoItem` 实例。

创建数据

`HttpClient.PostAsync` 方法用于将 POST 请求发送到 URI，指定 web 服务，然后从 web 服务接收的响应，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  // RestUrl = http://developer.xamarin.com:8081/api/todoitems/
  var uri = new Uri (string.Format (Constants.RestUrl, string.Empty));

  ...
  var json = JsonConvert.SerializeObject (item);
  var content = new StringContent (json, Encoding.UTF8, "application/json");

  HttpResponseMessage response = null;
  if (isNewItem) {
    response = await client.PostAsync (uri, content);
  }
  ...

  if (response.IsSuccessStatusCode) {
    Debug.WriteLine (@"               TodoItem successfully saved.");

  }
  ...
}
```

`TodoItem` 实例转换为 JSON 负载将发送到 web 服务。此负载然后嵌入在正文中的之前与发出的请求将发送到 web 服务的 HTTP 内容 `PostAsync` 方法。

REST 服务发送的 HTTP 状态代码 `HttpResponseMessage.IsSuccessStatusCode` 属性，以指示 HTTP 请求是否成功。此操作的常见响应是：

- **201（已创建）** – 请求导致已发送响应之前正在创建新资源。
- **400（错误请求）** – 服务器不理解此请求。
- **409（冲突）** – 请求可能不会执行因为服务器上有冲突。

更新数据

`HttpClient.PutAsync` 方法用于将 PUT 请求发送到 URI，指定 web 服务，然后接收响应从 web 服务，如下面的代码示例中所示：

```
public async Task SaveTodoItemAsync (TodoItem item, bool isNewItem = false)
{
  ...
  response = await client.PutAsync (uri, content);
  ...
}
```

该操作的 `PutAsync` 方法等同于 `PostAsync` 用于在 web 服务中创建数据的方法。但是，从 web 服务发送的可能响应与不同。

REST 服务发送的 HTTP 状态代码 `HttpResponseMessage.IsSuccessStatusCode` 属性，以指示 HTTP 请求是否成功。此操作的常见响应是：

- **204（无内容）** – 成功处理该请求和响应是有意留为空白。
- **400（错误请求）** – 服务器不理解此请求。
- **404（未找到）** – 服务器上不存在请求的资源。

删除数据

`HttpClient.DeleteAsync` 方法用于将 DELETE 请求发送到 URI，指定 web 服务，然后接收响应从 web 服务，如下面的代码示例中所示：

```
public async Task DeleteTodoItemAsync (string id)
{
  // RestUrl = http://developer.xamarin.com:8081/api/todoitems/{0}
  var uri = new Uri (string.Format (Constants.RestUrl, id));
  ...
  var response = await client.DeleteAsync (uri);
  if (response.IsSuccessStatusCode) {
    Debug.WriteLine (@"                TodoItem successfully deleted.");
  }
  ...
}
```

REST 服务发送的 HTTP 状态代码 `HttpResponseMessage.IsSuccessStatusCode` 属性，以指示 HTTP 请求是否成功。此操作的常见响应是：

- **204（无内容）** – 成功处理该请求和响应是有意留为空白。
- **400（错误请求）** – 服务器不理解此请求。
- **404（未找到）** – 服务器上不存在请求的资源。

总结

这篇文章学习了如何使用 rest 样式 web 服务从 Xamarin.Forms 应用程序，使用 `HttpClient` 类。REST 的简单性已帮助使其用于访问移动应用程序中的 web 服务的主要方法。

相关链接

- 为本机移动应用程序创建后端服务
- TodoREST（示例）
- HttpClient

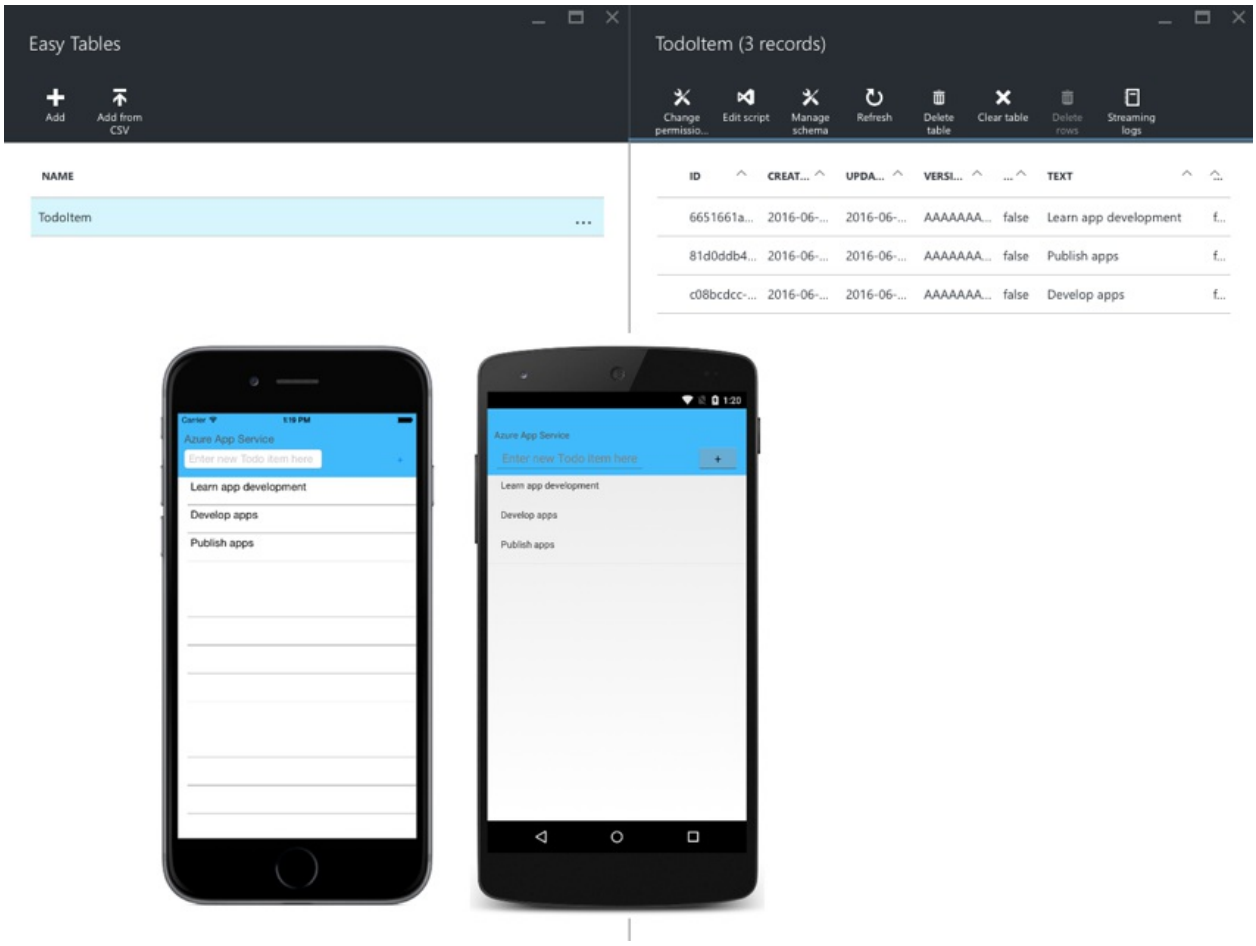
使用 Azure 移动应用

2018/6/20 • • Edit Online

Azure 移动应用，可以使用可缩放的后端移动身份验证、脱机同步和推送通知的支持与托管在 Azure App Service 中开发应用。本文中，这仅适用于使用 Node.js 后端的 Azure Mobile Apps，说明如何查询、插入、更新和删除数据存储在 Azure Mobile Apps 实例中的表。

> **NOTE**
>
> 从开始在 6 月 30 日，所有新的 Azure Mobile Apps 将创建使用 TLS 1.2 默认情况下。此外，还建议你现有 Azure Mobile Apps 重新配置为使用 TLS 1.2。有关如何强制 Azure 移动应用程序中的 TLS 1.2 的信息，请参阅强制实施 TLS 1.2。有关如何配置 Xamarin 项目以使用 TLS 1.2 的信息，请参阅传输层安全 (TLS) 1.2。

有关如何创建可供 Xamarin.Forms Azure Mobile Apps 实例的信息，请参阅创建 Xamarin.Forms 应用。按照这些说明操作之后，的可下载示例应用程序可以将配置为使用 Azure Mobile Apps 实例通过设置 `Constants.ApplicationURL` Azure Mobile Apps 实例的 url。然后，运行示例应用程序时它将连接到 Azure Mobile Apps 实例，如下面的屏幕截图中所示：



对 Azure 移动应用的访问是通过Azure 移动客户端 SDK，和从 Xamarin.Forms 示例应用程序到 Azure 的所有连接都均通过 HTTPS。

NOTE

在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

使用 Azure 移动应用程序实例

Azure 移动客户端 SDK提供 `MobileServiceClient` 类，用于通过 Xamarin.Forms 应用程序访问 Azure Mobile Apps 的实例，如下面的代码示例中所示：

```
IMobileServiceTable<TodoItem> todoTable;
MobileServiceClient client;

public TodoItemManager ()
{
  client = new MobileServiceClient (Constants.ApplicationURL);
  todoTable = client.GetTable<TodoItem> ();
}
```

当 `MobileServiceClient` 创建实例，必须指定应用程序 URL 以标识 Azure Mobile Apps 实例。此值可以从移动应用中的仪表板获取Microsoft Azure 门户。

对引用 `TodoItem` 可以在此表上执行操作之前，必须获取存储在 Azure Mobile Apps 实例中的表。这通过调用实现 `GetTable` 方法 `MobileServiceClient` 实例，它将返回 `IMobileServiceTable<TodoItem>` 引用。

查询数据

可通过调用检索表的内容 `IMobileServiceTable.ToEnumerableAsync` 以异步方式计算查询并返回结果的方法。数据也可以通过包括筛选服务器端 `Where` 在查询中的子句。 `Where` 子句应用一个行筛选谓词对表中，执行查询，如下面的代码示例中所示：

```
public async Task<ObservableCollection<TodoItem>> GetTodoItemsAsync (bool syncItems = false)
{
  ...
  IEnumerable<TodoItem> items = await todoTable
              .Where (todoItem => !todoItem.Done)
              .ToEnumerableAsync ();

  return new ObservableCollection<TodoItem> (items);
}
```

此查询将返回中的所有项 `TodoItem` 表 `Done` 属性等于 `false` 。查询结果随后会放在 `ObservableCollection` 进行显示。

插入数据

当在 Azure Mobile Apps 实例中插入数据，将自动生成新列的表中根据需要，提供在 Azure Mobile Apps 实例中启用该动态架构。 `IMobileServiceTable.InsertAsync` 方法用于将新的数据行插入到指定的表中，如下面的代码示例中所示：

```
public async Task SaveTaskAsync (TodoItem item)
{
  ...
  await todoTable.InsertAsync (item);
  ...
}
```

在发出一个插入请求，正在传递给 Azure Mobile Apps 实例的数据中必须未指定 ID。如果插入请求包含 ID `MobileServiceInvalidOperationException` 将引发。

后 `InsertAsync` 方法完成后，Azure Mobile Apps 实例中的数据的 ID 将在填充 `TodoItem` Xamarin.Forms 应用程序中的实例。

更新数据

当更新在 Azure Mobile Apps 实例中的数据，将自动生成新列的表中根据需要，提供在 Azure Mobile Apps 实例中启用该动态架构。`IMobileServiceTable.UpdateAsync` 方法用于更新现有数据，使用新信息，如下面的代码示例中所示：

```
public async Task SaveTaskAsync (TodoItem item)
{
  ...
  await todoTable.UpdateAsync (item);
  ...
}
```

在更新请求时，必须指定 ID，这样 Azure Mobile Apps 实例可以标识要更新的数据。此 ID 值存储在 `TodoItem.ID` 属性。如果更新请求不包含 ID 没有更新，以确定数据的 Azure Mobile Apps 实例方法，因此 `MobileServiceInvalidOperationException` 将引发。

删除数据

`IMobileServiceTable.DeleteAsync` 方法用于删除数据从 Azure Mobile Apps 表，如下面的代码示例中所示：

```
public async Task DeleteTaskAsync (TodoItem item)
{
  ...
  await todoTable.DeleteAsync(item);
  ...
}
```

在发出删除请求时，必须指定 ID，以便 Azure 移动应用程序 sinstance 可以确定要删除的数据。此 ID 值存储在 `TodoItem.ID` 属性。如果在删除请求不包含 ID，没有要删除，以确定数据的 Azure Mobile Apps 实例方法，因此 `MobileServiceInvalidOperationException` 将引发。

总结

本文介绍了如何使用Azure 移动客户端 SDK来查询、插入、更新和删除数据存储在 Azure 移动应用程序实例中的表。该 SDK 提供 `MobileServiceClient` Xamarin.Forms 应用程序用于访问 Azure Mobile Apps 实例的类。

相关链接

- TodoAzure（示例）
- 创建 Xamarin.Forms 应用
- Azure 的移动客户端 SDK
- MobileServiceClient

对 Web 服务的访问进行身份验证

2018/6/9 • • Edit Online

本指南说明如何将身份验证服务集成到 Xamarin.Forms 应用程序以使用户能够共享一个后端，同时仅有权访问他们自己的数据。涉及的主题包括使用基本身份验证与 REST 服务中，使用 Xamarin.Auth 组件对 OAuth 标识提供程序，进行身份验证，以及使用内置身份验证机制提供的不同的提供程序。

对 RESTful Web 服务进行身份验证

HTTP 支持使用多个身份验证机制来控制对资源的访问。基本身份验证提供对资源的访问权限仅这些客户端具有正确的凭据。本文演示如何使用基本身份验证来保护对 RESTful web 服务资源的访问。

使用标识提供程序的用户进行身份验证

Xamarin.Auth 是跨平台 SDK 进行身份验证用户以及存储的帐户。它包括提供对使用如 Google、Microsoft、Facebook 和 Twitter 标识提供程序支持的 OAuth 身份验证器。此文章介绍了如何使用 Xamarin.Auth 管理 Xamarin.Forms 应用程序中的身份验证过程。

使用 Azure 移动应用程序的用户进行身份验证

Azure Mobile Apps 使用各种外部标识提供程序来支持进行身份验证和授权应用程序用户。权限可以然后设置上表限制为仅限经过身份验证的用户的访问。此文章介绍了如何使用 Azure 移动应用管理 Xamarin.Forms 应用程序中的身份验证过程。

使用 Azure Active Directory B2C 的用户进行身份验证

Azure Active Directory B2C 是面向使用者的 web 和移动应用程序的云标识管理解决方案。本文演示如何使用 Microsoft 身份验证库 (MSAL) 和 Azure Active Directory B2C 来将使用者标识管理集成到 Xamarin.Forms 应用程序。

将 Azure Active Directory B2C 与 Azure 移动应用集成

Azure Active Directory B2C 可以用于管理的 Azure 移动应用的身份验证工作流。使用此方法时，标识管理体验在云中，已完全定义，并且可以修改而无需更改你的移动应用程序代码。本文演示如何使用 Azure Active Directory B2C Xamarin.Forms 中提供身份验证和授权与 Azure Mobile Apps 实例。

相关链接

- Web 服务简介
- 异步支持概述

对 RESTful Web 服务进行身份验证

2018/6/9 • • Edit Online

HTTP 支持使用多个身份验证机制来控制对资源的访问。基本身份验证提供对资源的访问权限仅这些客户端具有正确的凭据。本文演示如何使用基本身份验证来保护对 RESTful web 服务资源的访问。

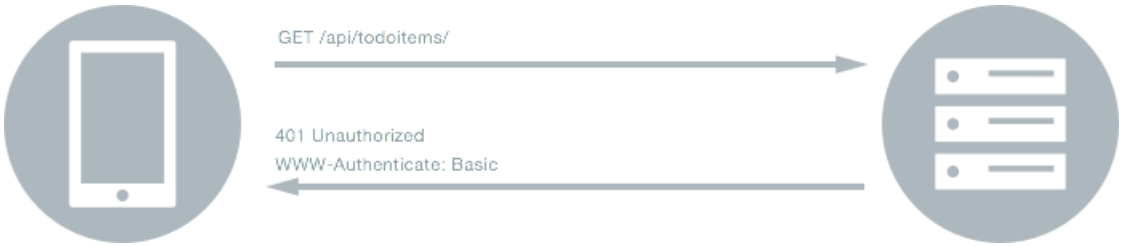
随附的 Xamarin.Forms 示例应用程序需要使用提供对 web 服务的只读访问权限的 Xamarin 托管 REST 服务。因此，创建、更新和删除数据的操作不会更改应用程序中使用的数据。但是，REST 服务的可承载版本位于*TodoRESTService*可以那里找到文件夹中的示例应用程序和服务设置的说明。此可承载版本的 REST 服务提供完整创建、更新、读取和删除访问的数据。

> **NOTE**
>
> 在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

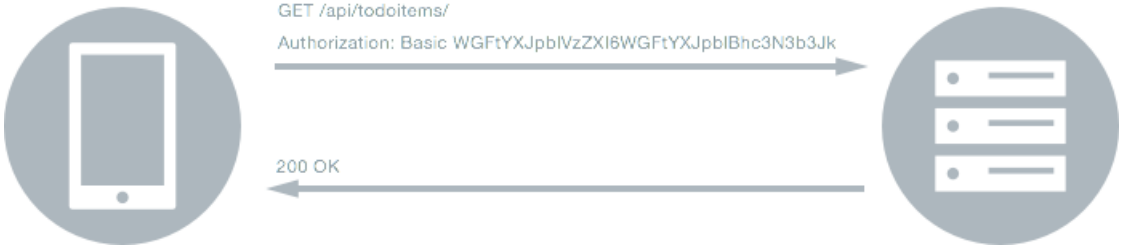
用户通过 HTTP 进行身份验证

基本身份验证是 HTTP，支持的最简单的身份验证机制，并且涉及在客户端发送的用户名和密码以未加密的 base64 编码文本。其工作原理，如下所示：

- 如果 web 服务收到为受保护资源请求时，它将拒绝该请求，HTTP 状态代码为 401（拒绝访问），并设置的 WWW 身份验证响应标头，如下面的关系图中所示：



- 如果 web 服务收到受保护资源的请求有 `Authorization` 正确设置标头，web 服务进行响应使用 HTTP 状态代码 200，这表示请求成功和请求的信息包含在响应中。这种情况下下图所示：



> **NOTE**
>
> 基本身份验证应仅用于通过 HTTPS 连接。当通过 HTTP 连接，使用 `Authorization` 可以轻松地解码标头，如果攻击者捕获的 HTTP 流量。

在 Web 请求中指定基本身份验证

使用基本身份验证，如下所示指定：

1. 添加到"基本"的字符串 `Authorization` 的请求标头。
2. 用户名和密码将合并为一个字符串，"用户名：密码"，然后使用 base64 编码，并且添加到它的格式 `Authorization` 的请求标头。

因此，利用 XamarinUser 用户名和的 XamarinPassword 密码中，标头将成为：

```
Authorization: Basic WGFtYXJpblVzZXI6WGFtYXJpblBhc3N3b3Jk
```

`HttpClient` 该类可以设置 `Authorization` 上的标头值 `HttpClient.DefaultRequestHeaders.Authorization` 属性。因为 `HttpClient` 实例存在跨多个请求，`Authorization` 标头仅需要一次，而不是时设置使每个请求，如下面的代码示例中所示：

```
public class RestService : IRestService
{
  HttpClient client;
  ...

  public RestService ()
  {
    var authData = string.Format ("{0}:{1}", Constants.Username, Constants.Password);
    var authHeaderValue = Convert.ToBase64String (Encoding.UTF8.GetBytes (authData));

    client = new HttpClient ();
    ...
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue ("Basic", authHeaderValue);
  }
  ...
}
```

然后当请求时，web 服务操作使用签名请求 `Authorization` 标头，，该值指示用户是否有权调用该操作。

> **NOTE**
> 虽然示例 REST 服务将凭据存储为常量，它们应不会在已发布的应用程序中不安全格式存储。Xamarith.Auth NuGet 提供用于安全地存储凭据的功能。有关详细信息请参阅存储和检索设备上的帐户信息。

处理授权标头服务器端

随附的示例 REST 服务修饰与每个操作 `[BasicAuthentication]` 属性。此属性由 `BasicAuthenticationAttribute` 类解决方案中、并用于分析 `Authorization` 标头，并确定 base64 编码凭据是否通过将它们与存储的值进行比较有效*Web.config*.虽然这种方法适合于此示例服务，但它需要的面向公众的 web 服务扩展。

在基本身份验证模块中使用的 IIS，用户进行身份验证对其 Windows 凭据。因此，用户必须具有在服务器的域帐户。但是，可以配置基本身份验证模型，以允许自定义身份验证，其中用户帐户进行身份验证对外部源，如数据库。有关详细信息请参阅ASP.NET Web API 中的基本身份验证ASP.NET 网站上。

> **NOTE**
> 基本身份验证不适合管理登出。因此，注销的标准的基本身份验证方法是以结束会话。

总结

这篇文章演示了如何将基本身份验证添加到发出的 Xamarin.Forms 应用程序使用的 web 请求 `HttpClient` 类。基

本身份验证提供对资源的访问权限仅这些客户端具有正确的凭据。有关如何使用信息Xamarin.Auth来管理在 Xamarin.Forms 应用程序的身份验证过程，以便用户可以共享一个后端，同时仅有权访问其数据，请参阅进行身份验证的用户使用标识提供程序。

相关链接

- TodoREST（示例）
- 使用 rest 样式 web 服务
- HttpClient

与标识提供程序的用户进行身份验证

2018/7/6 • • Edit Online

Xamarin.Auth 是用于对用户进行身份验证和存储其帐户的跨平台 SDK。它包括对使用 Google、Microsoft、Facebook 和 Twitter 等标识提供程序提供支持的 OAuth 身份验证器。本文介绍如何使用 Xamarin.Auth 管理 Xamarin.Forms 应用程序中的身份验证过程。

OAuth 是开放式的标准进行身份验证，并启用通知资源提供程序应授予权限的第三方而无需共享资源所有者标识访问其信息的资源所有者。此示例将启用通知（如 Google、Microsoft、Facebook 或 Twitter）的标识提供者，将向应用程序访问其数据，而无需共享用户的标识授予权限的用户。它通常用于作为用户的方法登录到网站和应用程序使用标识提供程序，但不公开到网站或应用程序密码。

使用 OAuth 标识提供程序时的身份验证流的高级概述如下所示：

1. 应用程序导航到标识提供者 URL 的浏览器。
2. 标识提供程序处理用户身份验证，并将授权代码返回到应用程序。
3. 应用程序与其交换中的标识提供程序的访问令牌的授权代码。
4. 应用程序使用访问令牌来访问标识提供程序，如请求基本用户数据的 API 的 Api。

示例应用程序演示了如何使用 Xamarin.Auth 实现本机身份验证流对 Google。虽然 Google 用作本主题中的标识提供程序，方法是同样适用于其他标识提供者。有关使用 Google OAuth 2.0 终结点身份验证的详细信息，请参阅访问 Google api 使用 OAuth2.0 Google 的网站上。

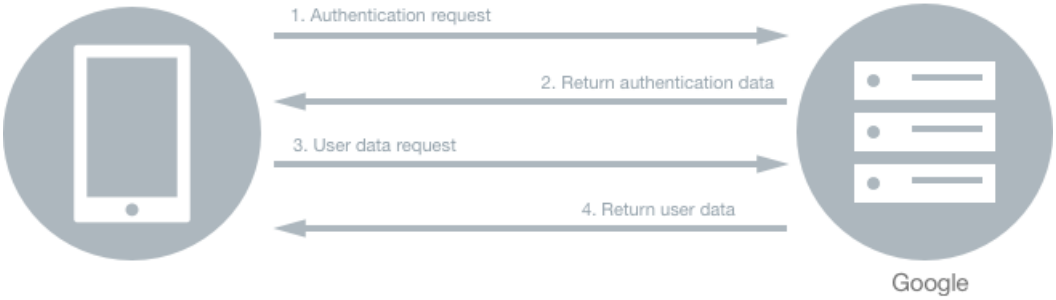
> **NOTE**
> 在 iOS 9 及更高版本中，应用传输安全 (ATS) 强制执行安全连接之间（例如应用程序的后端服务器）的 internet 资源和应用程序中，从而防止意外泄露敏感信息。由于默认情况下构建适用于 iOS 9 应用程序中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，它们将失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议并确保对 internet 资源的通信安全。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用程序传输安全。

使用 Xamarin.Auth 用户进行身份验证

Xamarin.Auth 支持两种方法的应用程序与标识提供程序的授权终结点进行交互：

1. 使用嵌入式的 web 视图。虽然这是一种常见做法，但不再建议出于以下原因：
   - 承载 web 视图的应用程序可以访问用户的完全身份验证凭据，而不仅仅是适用于应用程序的 OAuth 授权授予。这违反了最低特权原则，如应用程序有权访问更强大的凭据不是它需要有可能增加应用程序的受攻击面。
   - 主机应用程序能捕获用户名和密码、自动提交窗体和绕过用户同意，和复制会话 cookie 并使用它们的用户身份执行已经过身份验证的操作。
   - 嵌入式的 web 视图不与其他应用程序或设备的 web 浏览器，因此需要用户在为每个授权请求，它被视为较差的用户体验的登录共享身份验证状态。
   - 某些授权终结点采取措施来检测和阻止来自 web 视图的授权请求。
2. 使用设备的 web 浏览器中，这是建议的方法。使用设备浏览器进行 OAuth 请求可提高应用程序的可用性，因为用户只需登录到标识提供程序一次每个设备，从而提高转换率的应用程序中的登录和授权流程。设备浏览器还提供了改进的安全性，如应用程序都能够检查和修改内容在 web 视图中，但不是显示在浏览器中的内容。这是此文章和示例应用程序中采用的方法。

下图中所示的示例应用程序如何使用 Xamarin.Auth 对用户进行身份验证和检索其基本数据的高级概述：



应用程序发出对使用 Google 的身份验证请求 `OAuth2Authenticator` 类。返回身份验证响应时，在用户已成功通过身份验证 Google 通过其在登录页面，其中包括访问令牌。然后，应用程序的基本用户数据，使用 Google 向发出请求 `OAuth2Request` 类，包含在请求中的访问令牌。

安装

必须创建一个 Google API 控制台项目与 Xamarin.Forms 应用程序集成，Google 登录。这可以通过以下操作实现：

1. 转到Google API 控制台网站，然后使用 Google 帐户凭据登录。
2. 从项目下拉列表中选择一个现有项目，或创建一个新。
3. 在"API 管理器"下侧栏中选择**凭据**，然后选择**OAuth 许可屏幕**选项卡。选择电子邮件地址，指定向用户显示产品名称，然后按**保存**。
4. 在中**凭据**选项卡上，选择**创建凭据**下拉列表，并选择**OAuth 客户端 ID**。
5. 下**应用程序类型**，选择移动应用程序将在运行的平台 (**iOS**或**Android**)。
6. 填写所需的详细信息，然后选择**创建**按钮。

> **NOTE**
> 客户端 ID 可让应用程序访问已启用的 Google Api，并为移动应用程序仅适用于单个平台。因此，**OAuth 客户端 ID**应创建为将使用 Google 登录每个平台。

执行这些步骤后，Xamarin.Auth 可用于启动使用 Google 的 OAuth2 身份验证流。

创建和配置身份验证器

Xamarin.Auth 的 `OAuth2Authenticator` 类负责处理 OAuth 身份验证流。下面的代码示例演示的实例化 `OAuth2Authenticator` 类执行身份验证设备的 web 浏览器时：

```
var authenticator = new OAuth2Authenticator(
    clientId,
    null,
    Constants.Scope,
    new Uri(Constants.AuthorizeUrl),
    new Uri(redirectUri),
    new Uri(Constants.AccessTokenUrl),
    null,
    true);
```

`OAuth2Authenticator` 类需要多个参数，如下所示：

- **客户端 ID** – 此标识正在发出请求，并且可以从项目中检索的客户端Google API 控制台。
- **客户端机密**– 这应该是 `null` 或 `string.Empty` 。
- **作用域**– 此标识符可标识正在请求应用程序的 API 访问权限和值通知向用户显示的许可屏幕。有关作用域的详细信息，请参阅授权 API 请求Google 的网站上。
- **授权 URL** – 此标识将来获取授权代码的 URL。
- **重定向 URL** – 此标识，以便进行发送响应的 URL。此参数的值必须匹配中显示的值之一**凭据**选项卡中的项

目Google 开发人员控制台。

- **AccessToken Url** – 这标识用于获取授权代码后请求访问令牌的 URL。
- **GetUserNameAsync Func** – 一个可选的 `Func` 将用于它已成功完成身份验证后以异步方式检索帐户的用户名。
- **使用本机 UI** – 一 `boolean` 值，该值指示是否使用设备的 web 浏览器来执行身份验证请求。

设置身份验证事件处理程序

显示用户界面，事件处理程序之前 `OAuth2Authenticator.Completed` 必须注册事件，如下面的代码示例中所示：

```
authenticator.Completed += OnAuthCompleted;
```

在用户成功进行身份验证或取消登录时，会触发此事件。

（可选）的事件处理程序 `OAuth2Authenticator.Error` 也可以注册事件。

显示登录用户界面

通过使用必须在每个平台项目中初始化 Xamarin.Auth 登录表示器，可以向用户显示登录用户界面。下面的代码示例演示如何初始化中的登录名演示者 `AppDelegate` iOS 项目中的类：

```
global::Xamarin.Auth.Presenters.XamarinIOS.AuthenticationConfiguration.Init();
```

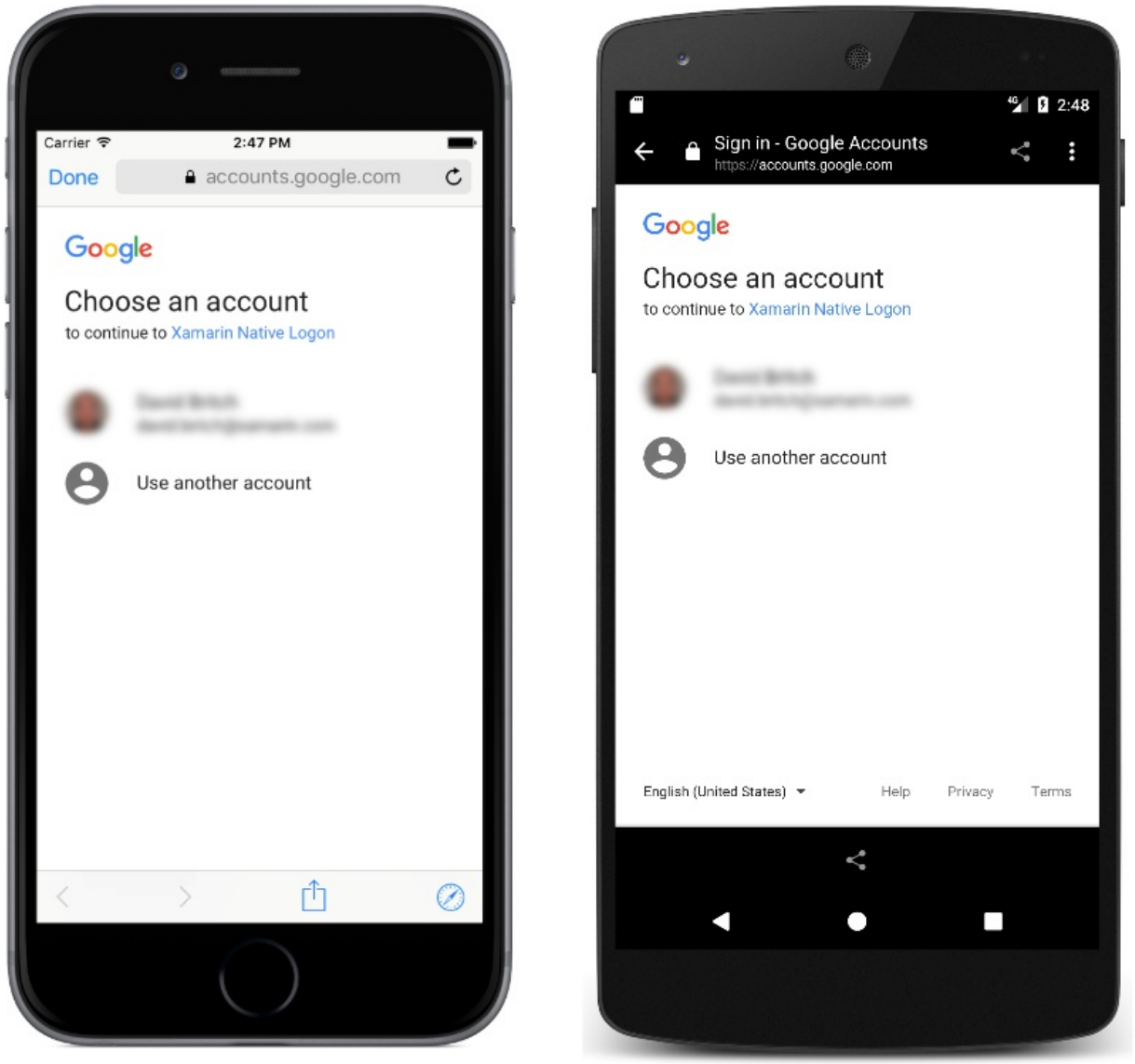
下面的代码示例演示如何初始化中的登录名演示者 `MainActivity` Android 项目中的类：

```
global::Xamarin.Auth.Presenters.XamarinAndroid.AuthenticationConfiguration.Init(this, bundle);
```

.NET Standard 库项目然后可以按如下所示调用登录演示者：

```
var presenter = new Xamarin.Auth.Presenters.OAuthLoginPresenter();
presenter.Login(authenticator);
```

请注意，参数 `Xamarin.Auth.Presenters.OAuthLoginPresenter.Login` 方法是 `OAuth2Authenticator` 实例。当 `Login` 调用方法、登录用户界面显示为一个选项卡中的用户从设备的 web 浏览器中，下面的屏幕截图中所示：



处理重定向 URL

用户完成身份验证过程后，控件将在 web 浏览器选项卡中，返回到应用程序。这被通过重定向 url 返回的身份验证过程中，检测并处理自定义 URL 后将其发送注册自定义的 URL 方案。

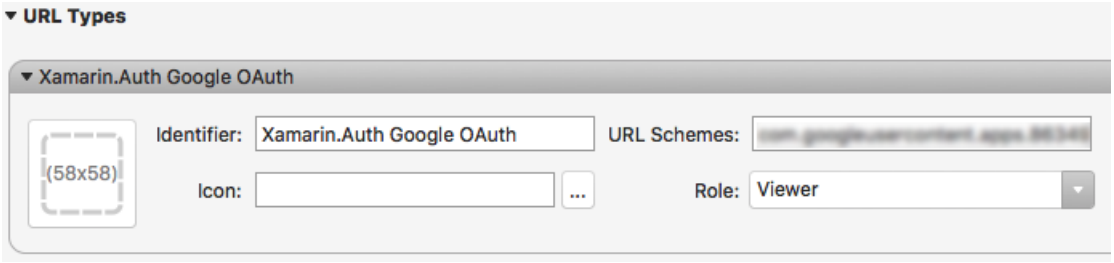
选择要与应用程序关联的自定义 URL 方案，应用程序必须使用基于名称受其控制的方案。这可以通过 iOS 和 Android 上的包名称上使用捆绑包标识符名称，并反转，以便 URL 方案来实现。但是，某些标识提供者，如 Google，分配基于域的名称，然后反转并用作 URL 方案的客户端标识符。例如，如果 Google 创建的客户端 id `902730282010-ks3kd03ksoasioda93jldas93jjj22kr.apps.googleusercontent.com`，将为 URL 方案 `com.googleusercontent.apps.902730282010-ks3kd03ksoasioda93jldas93jjj22kr`。请注意，只有一个 `/` 方案组件后可以显示。因此，利用自定义的 URL 方案的重定向 URL 的完整示例是 `com.googleusercontent.apps.902730282010-ks3kd03ksoasioda93jldas93jjj22kr:/oauth2redirect`。

Web 浏览器收到响应后从包含自定义的 URL 方案的标识提供程序，它尝试加载的 URL，将会失败。相反，自定义 URL 方案被报告给操作系统引发的事件。对于已注册的架构，然后检查操作系统和操作系统如果找到一个对象，将启动应用程序注册方案，并将其发送重定向 URL。

用于向操作系统注册自定义的 URL 方案和处理方案的机制是特定于每个平台。

iOS

在 iOS 上，自定义的 URL 方案中注册**Info.plist**，如以下屏幕截图中所示：



标识符值可以是任何内容，并角色值必须设置为查看器。Url 方案值，开头 `com.googleusercontent.apps`，可以从项目的 iOS 客户端 id 获取上Google API 控制台。

标识提供程序完成授权请求，它将重定向到应用程序的重定向 URL。由于 URL 使用自定义方案，这样可以在 iOS 中启动应用程序，同时将 URL 传递作为启动参数，它由处理 `OpenUrl` 在应用程序的重写 `AppDelegate` 类，该类在下面的代码示例所示：

```
public override bool OpenUrl(UIApplication app, NSUrl url, NSDictionary options)
{
    // Convert NSUrl to Uri
    var uri = new Uri(url.AbsoluteString);

    // Load redirectUrl page
    AuthenticationState.Authenticator.OnPageLoading(uri);

    return true;
}
```

`OpenUrl` 方法将从接收到的 URL 转换 `NSUrl` 到.NET `Uri`，然后再处理与重定向 URL `OnPageLoading` 公共方法 `OAuth2Authenticator` 对象。这将导致 Xamarin.Auth 以关闭 web 浏览器选项卡上，并将收到的 OAuth 数据分析。

Android

在 Android 上，自定义的 URL 方案注册通过指定 `IntentFilter` 特性，可以在 `Activity`，用于处理方案。标识提供程序完成授权请求，它将重定向到应用程序的重定向 URL。为 URL 使用自定义方案，这样可以在 Android 中启动应用程序，同时将 URL 传递作为启动参数，它由处理 `OnCreate` 方法的 `Activity` 注册用于处理自定义 URL 方案。下面的代码示例显示了处理自定义 URL 方案的示例应用程序中的类：

```
[Activity(Label = "CustomUrlSchemeInterceptorActivity", NoHistory = true, LaunchMode = LaunchMode.SingleTop
)]
[IntentFilter(
    new[] { Intent.ActionView },
    Categories = new [] { Intent.CategoryDefault, Intent.CategoryBrowsable },
    DataSchemes = new [] { "<insert custom URL here>" },
    DataPath = "/oauth2redirect")]
public class CustomUrlSchemeInterceptorActivity : Activity
{
    protected override void OnCreate(Bundle savedInstanceState)
    {
        base.OnCreate(savedInstanceState);

        // Convert Android.Net.Url to Uri
        var uri = new Uri(Intent.Data.ToString());

        // Load redirectUrl page
        AuthenticationState.Authenticator.OnPageLoading(uri);

        Finish();
    }
}
```

`DataSchemes` 的属性 `IntentFilter` 必须设置为在获取项目的 Android 客户端 id 的反向客户端标识符Google API 控制台。

`OnCreate` 方法将从接收到的 URL 转换 `Android.Net.Url` 到.NET `Uri`，然后再处理与重定向 URL `OnPageLoading` 公共方法 `OAuth2Authenticator` 对象。这将导致 Xamarin.Auth 关闭 web 浏览器选项卡上，并将收到的 OAuth 数据分析。

> **IMPORTANT**
> 在 Android 上，使用 Xamarin.Auth `CustomTabs` API 与 web 浏览器和操作系统进行通信。但是，它不能保证的 `CustomTabs` 将用户的设备上安装兼容的浏览器。

检查 OAuth 响应

分析收到的 OAuth 数据之后，将引发 Xamarin.Auth `OAuth2Authenticator.Completed` 事件。此事件的事件处理程序中 `AuthenticatorCompletedEventArgs.IsAuthenticated` 属性可用于确定是否已成功身份验证，如下面的代码示例中所示：

```
async void OnAuthCompleted(object sender, AuthenticatorCompletedEventArgs e)
{
  ...
  if (e.IsAuthenticated)
  {
    ...
  }
}
```

收集从身份验证成功的数据现已推出 `AuthenticatorCompletedEventArgs.Account` 属性。这包括访问令牌，可以使用数据传输到标识提供程序提供的 API 的请求进行签名。

发出请求的数据

应用程序获取访问令牌后，它用于向发送请求 `https://www.googleapis.com/oauth2/v2/userinfo` API，请求从标识提供者的基本用户数据。使用 Xamarin.Auth 的发出此请求 `OAuth2Request` 类，该类表示使用从检索到的帐户进行身份验证的请求 `OAuth2Authenticator` 实例，如下面的代码示例中所示：

```
// UserInfoUrl = https://www.googleapis.com/oauth2/v2/userinfo
var request = new OAuth2Request ("GET", new Uri (Constants.UserInfoUrl), null, e.Account);
var response = await request.GetResponseAsync ();
if (response != null)
{
  string userJson = response.GetResponseText ();
  var user = JsonConvert.DeserializeObject<User> (userJson);
}
```

HTTP 方法和 API URL，以及 `OAuth2Request` 还指定了实例 `Account` 实例，其中包含访问令牌来登录到由指定的 URL 请求 `Constants.UserInfoUrl` 属性。标识提供者然后返回作为 JSON 响应，包括用户的名称和电子邮件地址，前提它识别为有效的访问令牌的基本用户数据。JSON 响应然后读取和反序列化为 `user` 变量。

有关详细信息，请参阅调用 Google API Google 开发人员门户上。

存储和检索设备上的帐户信息

安全地存储 Xamarin.Auth `Account` 存储帐户中的对象，以便应用程序并不总是有重新进行身份验证的用户。`AccountStore` 类是负责存储帐户信息，由密钥链在 iOS 中，服务提供支持和 `KeyStore` 在 Android 中的类。

下面的代码示例演示如何 `Account` 安全地保存对象：

```
AccountStore.Create ().Save (e.Account, Constants.AppName);
```

已保存的帐户进行唯一标识使用组成该帐户的密钥 `Username` 属性和服务 ID，这是一个字符串，用于提取帐户时从帐户存储。如果 `Account` 以前保存，调用 `Save` 方法再次将其覆盖。

`Account` 对象服务可以检索通过调用 `FindAccountsForService` 方法，如下面的代码示例中所示：

```
var account = AccountStore.Create ().FindAccountsForService (Constants.AppName).FirstOrDefault();
```

`FindAccountsForService` 方法将返回 `IEnumerable` 的集合 `Account` 对象，与设置为匹配帐户的集合中的第一个项。

总结

本文介绍了如何使用 Xamarin.Auth 管理 Xamarin.Forms 应用程序中的身份验证过程。提供了 Xamarin.Auth `OAuth2Authenticator` 和 `OAuth2Request` Xamarin.Forms 应用程序用于使用 Google、Microsoft、Facebook 和 Twitter 等标识提供程序的类。

相关链接

- OAuthNativeFlow（示例）
- 对于本机应用程序的 OAuth 2.0
- 使用 OAuth2.0 访问 Google Api
- Xamarin.Auth (NuGet)
- Xamarin.Auth (GitHub)

使用 Azure 移动应用程序的用户进行身份验证

2018/6/9 • • Edit Online

Azure Mobile Apps 使用各种外部标识提供程序来支持进行身份验证和授权应用程序用户，包括 Facebook、Google、Microsoft、Twitter 和 Azure Active Directory。可以在表上设置权限，以限制对仅限于经过身份验证的用户访问。此文章介绍了如何使用 Azure 移动应用管理 Xamarin.Forms 应用程序中的身份验证过程。

概述

具有 Azure Mobile Apps 管理 Xamarin.Forms 应用程序中的身份验证过程的过程如下所示：

1. 注册标识提供程序的站点，在 Azure 移动应用程序，然后在移动应用后端中设置的提供程序生成的凭据。有关详细信息，请参阅注册你的应用程序身份验证并配置应用程序服务。
2. 定义用于 Xamarin.Forms 应用程序，以允许身份验证系统重定向回 Xamarin.Forms 应用程序身份验证过程完成后的新 URL 方案。有关详细信息，请参阅将您的应用程序添加到允许外部重定向 Url。
3. 限制对 Azure 移动应用后端到已经过身份验证客户端的访问。有关详细信息，请参阅将权限限制给已经过身份验证的用户。
4. 调用从 Xamarin.Forms 应用程序的身份验证。有关详细信息，请参阅添加身份验证到可移植类库，向 iOS 应用程序添加身份验证，向 Android 应用添加身份验证，和向 Windows 10 应用程序项目添加身份验证。

> **NOTE**
> 在 iOS 9 及更高版本中，应用程序传输安全 (ATS) 强制实施安全连接之间 internet 资源（如应用程序的后端服务器）和应用程序，从而防止意外泄露的敏感信息。由于默认情况下，生成的 ios 9 应用中启用了 ATS，所有连接都将遵循 ATS 安全要求。如果连接不能满足这些要求，则会失败并出现异常。如果不能使用 ATS 可以选择退出的 `HTTPS` 协议和安全的 internet 资源的通信。这可以通过更新应用程序的实现**Info.plist**文件。有关详细信息请参阅应用传输安全。

从历史上看，移动应用程序具有使用嵌入的 web 视图来执行与标识提供程序的身份验证。这不再出于以下原因，建议：

- 承载 web 视图的应用程序可以访问用户的完整的身份验证凭据，而不仅仅是针对应用程序的授权授予。这违反了最低特权原则，如应用程序有权访问更强大的凭据不是它需要有可能增加应用程序的受攻击面。
- 主机应用程序能捕获用户名和密码、自动提交窗体和绕过用户同意，和复制会话 cookie 并将其用于以用户身份执行已经过身份验证的操作。
- 嵌入的 web 视图不要与其他应用程序或设备的 web 浏览器，需要用户登录为每个授权请求被认为是较差的用户体验共享身份验证状态。
- 某些授权终结点采取措施来检测和阻止来自 web 视图的授权请求。

替代方法是使用设备的 web 浏览器来执行身份验证，这是最新版本而采取的做法Azure 移动客户端 SDK。使用设备浏览器的身份验证请求可以提高应用程序的可用性，因为用户仅需要登录到标识提供程序一次每个设备，提高应用程序中的登录和授权流的转换率。设备浏览器还提供了改进的安全性，如应用程序能够检查和修改在 web 视图中中的内容，但不是在浏览器中显示的内容。

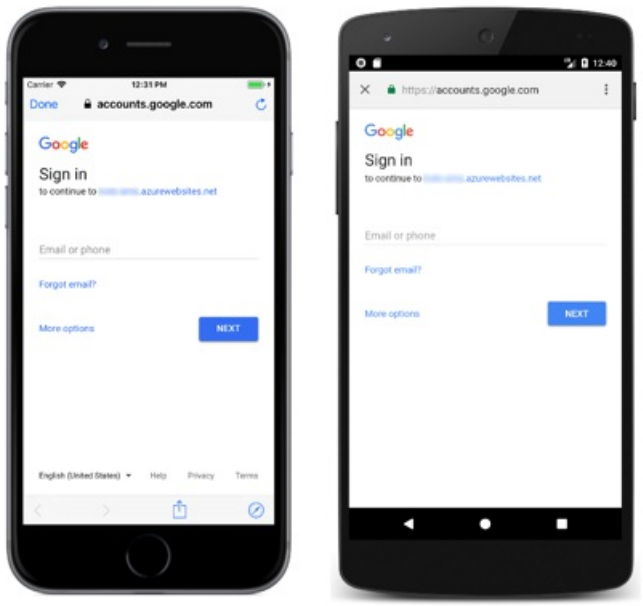
使用 Azure 移动应用程序实例

Azure 移动客户端 SDK提供 `MobileServiceClient` 类，该类 Xamarin.Forms 应用程序用于访问 Azure Mobile Apps 的实例。

示例应用程序使用 Google 作为标识提供程序，允许使用 Google 帐户登录到 Xamarin.Forms 应用程序的用户。虽然 Google 用作这篇文章中的标识提供程序，方法是同样适用于其他标识提供程序。

在用户日志记录

在示例应用程序登录屏幕以下屏幕截图所示：



尽管 Google 使用为标识提供程序中，可以使用各种其他标识提供程序，包括 Facebook、Microsoft、Twitter 和 Azure Active Directory。

下面的代码示例演示如何调用登录过程：

```
async void OnLoginButtonClicked(object sender, EventArgs e)
{
  ...
  if (App.Authenticator != null)
  {
    authenticated = await App.Authenticator.AuthenticateAsync();
  }
  ...
}
```

`App.Authenticator` 属性是 `IAuthenticate` 由每个特定于平台的项目设置的实例。 `IAuthenticate` 接口指定 `AuthenticateAsync` 必须通过每个平台项目提供的操作。因此，调用 `App.Authenticator.AuthenticateAsync` 方法执行 `IAuthenticate.AuthenticateAsync` 平台项目中的方法。

所有平台 `IAuthenticate.AuthenticateAsync` 方法调用 `MobileServiceClient.LoginAsync` 方法以显示登录接口和缓存数据。下面的代码示例演示 `LoginAsync` 为 iOS 平台的方法：

```
public async Task<bool> AuthenticateAsync()
{
  ...
  // The authentication provider could also be Facebook, Twitter, or Microsoft
  user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
    UIApplication.SharedApplication.KeyWindow.RootViewController,
    MobileServiceAuthenticationProvider.Google,
    Constants.URLScheme);
  ...
}
```

下面的代码示例演示 `LoginAsync` Android 平台的方法：

```
public async Task<bool> AuthenticateAsync()
{
  ...
  // The authentication provider could also be Facebook, Twitter, or Microsoft
  user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
    this,
    MobileServiceAuthenticationProvider.Google,
    Constants.URLScheme);
  ...
}
```

下面的代码示例演示 `LoginAsync` 的通用 Windows 平台的方法：

```
public async Task<bool> AuthenticateAsync()
{
  ...
  // The authentication provider could also be Facebook, Twitter, or Microsoft
  user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
    MobileServiceAuthenticationProvider.Google,
    Constants.URLScheme);
  ...
}
```

在所有平台上 `MobileServiceAuthenticationProvider` 枚举用于在身份验证过程中指定的标识提供程序将使用。当 `MobileServiceClient.LoginAsync` 调用方法，Azure Mobile Apps 启动一个身份验证流程，通过显示登录页的所选提供程序，以及在用户与标识提供程序的成功登录后生成的身份验证令牌。`MobileServiceClient.LoginAsync` 方法返回 `MobileServiceUser` 将存储在实例 `MobileServiceClient.CurrentUser` 属性。此属性提供 `UserId` 和 `MobileServiceAuthenticationToken` 属性。这些表示经过身份验证的用户和用户的身份验证令牌。身份验证令牌将包括在 Azure Mobile Apps 实例，允许 Xamarin.Forms 应用程序需要身份验证的用户权限的 Azure 移动应用程序实例上执行操作进行的所有请求。

注销用户

下面的代码示例演示如何调用注销过程：

```
async void OnLogoutButtonClicked(object sender, EventArgs e)
{
  bool loggedOut = false;

  if (App.Authenticator != null)
  {
    loggedOut = await App.Authenticator.LogoutAsync ();
  }
  ...
}
```

`App.Authenticator` 属性是 `IAuthenticate` 由每个 platformproject 设置的实例。`IAuthenticate` 接口指定 `LogoutAsync` 必须通过每个平台项目提供的操作。因此，调用 `App.Authenticator.LogoutAsync` 方法执行 `IAuthenticate.LogoutAsync` 平台项目中的方法。

所有平台 `IAuthenticate.LogoutAsync` 方法调用 `MobileServiceClient.LogoutAsync` 方法来取消对与标识提供程序登录的用户进行身份验证。下面的代码示例演示 `LogoutAsync` 为 iOS 平台的方法：

```
public async Task<bool> LogoutAsync()
{
  ...
  foreach (var cookie in NSHttpCookieStorage.SharedStorage.Cookies)
  {
    NSHttpCookieStorage.SharedStorage.DeleteCookie (cookie);
  }
  await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
  ...
}
```

下面的代码示例演示 `LogoutAsync` Android 平台的方法：

```
public async Task<bool> LogoutAsync()
{
  ...
  CookieManager.Instance.RemoveAllCookie();
  await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
  ...
}
```

下面的代码示例演示 `LogoutAsync` 的通用 Windows 平台的方法：

```
public async Task<bool> LogoutAsync()
{
  ...
  await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
  ...
}
```

当 `IAuthenticate.LogoutAsync` 调用方法，通过标识提供程序设置任何 cookie 都被清除，之前 `MobileServiceClient.LogoutAsync` 调用方法来取消对与标识提供程序登录的用户进行身份验证。

总结

本文介绍如何使用 Azure 移动应用管理 Xamarin.Forms 应用程序中的身份验证过程。Azure Mobile Apps 使用各种外部标识提供程序来支持进行身份验证和授权应用程序用户，包括 Facebook、Google、Microsoft、Twitter 和 Azure Active Directory。`MobileServiceClient` Xamarin.Forms 应用程序使用类来控制对 Azure Mobile Apps 实例访问。

相关链接

- TodoAzureAuth（示例）
- 使用 Azure 移动应用
- 向 Xamarin.Forms 应用添加身份验证
- Azure 的移动客户端 SDK
- MobileServiceClient

使用 Azure Active Directory B2C 的用户进行身份验证

2018/7/12 • • Edit Online

Azure Active Directory B2C 是面向消费者的 web 和移动应用程序的云标识管理解决方案。本文演示如何使用 Microsoft 身份验证库和 Azure Active Directory B2C 集成到移动应用程序的使用者标识管理。

NOTE

Microsoft 身份验证库仍处于预览状态，但适用于在生产环境中使用。但是，那里可能重大更改的 API、内部缓存格式和库，这可能会影响你的应用程序的其他机制。

概述

Azure Active Directory B2C 是标识管理服务的面向消费者的应用程序，允许使用者来登录到由应用程序：

- 使用现有社交帐户（Microsoft、Google、Facebook、Amazon、LinkedIn）。
- 创建新凭据（电子邮件地址和密码，或用户名和密码）。这些凭据*本地*帐户。

将 Azure Active Directory B2C 标识管理服务集成到移动应用程序的过程如下所示：

1. 创建 Azure Active Directory B2C 租户。有关详细信息，请参阅Azure 门户中创建一个 Azure Active Directory B2C 租户。
2. 移动应用程序注册到 Azure Active Directory B2C 租户。注册过程将分配**应用程序 ID**，用于唯一标识应用程序和一个**重定向 URL**可用于将响应定向回你的应用程序。有关详细信息，请参阅Azure Active Directory B2C：注册应用程序。
3. 创建注册和登录策略。此策略将定义使用者注册和登录，期间将经历的体验，并且还指定的应用程序将接收的令牌内容在成功注册或登录。有关详细信息，请参阅Azure Active Directory B2C：内置策略。
4. 使用Microsoft 身份验证库(MSAL) 在移动应用程序与 Azure Active Directory B2C 租户启动的身份验证的工作流中。

NOTE

将 Azure Active Directory B2C 标识管理集成到移动应用程序，以及 MSAL 还可用来将 Azure Active Directory 标识管理集成到移动应用程序。这可以通过与 Azure Active Directory 在注册的移动应用程序来实现应用程序注册门户。注册过程将分配**应用程序 ID**用于唯一标识应用程序，使用 MSAL 时，应指定。有关详细信息，请参阅如何使用 v2.0 终结点注册应用，并进行身份验证在移动应用使用 Microsoft 身份验证库Xamarin 博客上。

MSAL 使用设备的 web 浏览器来执行身份验证。这可提高应用程序的可用性，因为用户只需要在登录后每台设备，从而提高转换率的单一登录和授权流应用程序中。设备浏览器还提供了改进的安全性。用户完成身份验证过程后，控件将在 web 浏览器选项卡中，返回到应用程序。这被通过重定向 url 返回的身份验证过程中，检测并处理自定义 URL 后将其发送注册自定义的 URL 方案。有关选择自定义的 URL 方案的详细信息，请参阅选择本机应用重定向 URI。

> **NOTE**
> 用于向操作系统注册自定义的 URL 方案和处理方案的机制是特定于每个平台。

每个请求都发送到 Azure Active Directory B2C 租户中指定*策略*。策略描述了使用者标识体验，例如注册、或单一登录。例如，注册策略允许通过以下设置来配置 Azure Active Directory B2C 租户的行为：

- 使用者可以用来登录到应用程序的帐户类型。
- 要在注册期间从使用者收集的数据。
- 多重身份验证。
- 注册页面内容。
- 移动应用程序时执行的策略已接收的令牌声明。

Azure Active Directory 租户可以包含不同类型，然后根据需要在应用程序中使用该类型的多个策略。此外，可以跨应用程序，允许您定义和修改使用者标识体验，而无需更改你的代码重用策略。有关策略的详细信息，请参阅Azure Active Directory B2C：内置策略。

安装

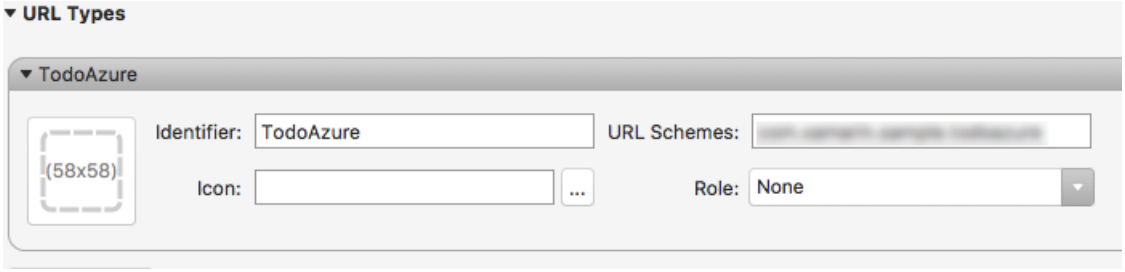
必须将 Microsoft 身份验证库 (MSAL) NuGet 库添加到可移植类库 (PCL) 项目和 Xamarin.Forms 解决方案中的平台项目。以下部分提供有关使用 MSAL 来与 Azure Active Directory B2C 租户从移动应用程序进行通信的其他安装说明。

可移植类库

使用 MSAL 的 Pcl 将需要重定目标以使用 Profile7。有关 PCL的详细信息，请参阅可移植类库简介。

iOS

在 iOS 上，与 Azure Active Directory B2C 注册的自定义 URL 方案必须注册中**Info.plist**，如以下屏幕截图中所示：



Azure Active Directory B2C 完成授权请求，它将重定向到注册的重定向 URL。由于 URL 使用它会导致启动的移动应用程序的 iOS 自定义方案，同时将 URL 传递作为启动参数，它由处理`OpenUrl`在应用程序的重写`AppDelegate`类，该类中的以下代码所示示例：

```
using Microsoft.Identity.Client;

namespace TodoAzure.iOS
{
    [Register("AppDelegate")]
    public partial class AppDelegate : global::Xamarin.Forms.Platform.iOS.FormsApplicationDelegate
    {
        ...
        public override bool OpenUrl(UIApplication app, NSUrl url, NSDictionary options)
        {
            AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(url);
            return true;
        }
    }
}
```

中的代码 `OpenURL` 方法可确保对 MSAL 返回控制后身份验证工作流的交互部分已结束。

Android

在 Android 上，与 Azure Active Directory B2C 注册的自定义 URL 方案必须注册中**AndroidManifest.xml**，通过添加 `<activity>` 元素内的现有 `<application>` 元素。 `<activity>` 元素指定 `IntentFilter` 上 `Activity` 的处理方案，并在下面的示例所示：

```
<application ...>
  <activity android:name="microsoft.identity.client.BrowserTabActivity">
    <intent-filter>
      <action android:name="android.intent.action.VIEW" />
      <category android:name="android.intent.category.DEFAULT" />
      <category android:name="android.intent.category.BROWSABLE" />
      <data android:scheme="INSERT_URL_SCHEME_HERE" android:host="auth" />
    </intent-filter>
  </activity>
</application>
```

Azure Active Directory B2C 完成授权请求，它将重定向到注册的重定向 URL。由于 URL 使用它会导致启动的移动应用程序的 Android 自定义方案，同时将 URL 传递作为启动参数，它由处理 `microsoft.identity.client.BrowserTabActivity` 。请注意， `data android:scheme` 属性必须设置为与 Azure Active Directory B2C 应用程序注册的自定义 URL 方案。

此外， `MainActivity` 必须修改类，如下面的代码示例中所示：

```
using Microsoft.Identity.Client;

namespace TodoAzure.Droid
{
    ...
    public class MainActivity : FormsAppCompatActivity
    {
        protected override void OnCreate(Bundle bundle)
        {
            base.OnCreate(bundle);

            global::Xamarin.Forms.Forms.Init(this, bundle);
            Microsoft.WindowsAzure.MobileServices.CurrentPlatform.Init();
            LoadApplication(new App());
            App.UiParent = new UIParent(this);
        }

        protected override void OnActivityResult(int requestCode, Result resultCode, Intent data)
        {
            base.OnActivityResult(requestCode, resultCode, data);
            AuthenticationContinuationHelper.SetAuthenticationContinuationEventArgs(requestCode, resultCode,
data);
        }
    }
}
```

`OnCreate` 方法修改通过分配 `UIParent` 实例向 `App.UiParent` 属性。这可确保身份验证流发生在当前活动的上下文中。

中的代码 `OnActivityResult` 方法可确保对 MSAL 返回控制后身份验证工作流的交互部分已结束。

通用 Windows 平台

在通用 Windows 平台上，无需其他设置需要使用 MSAL。

初始化

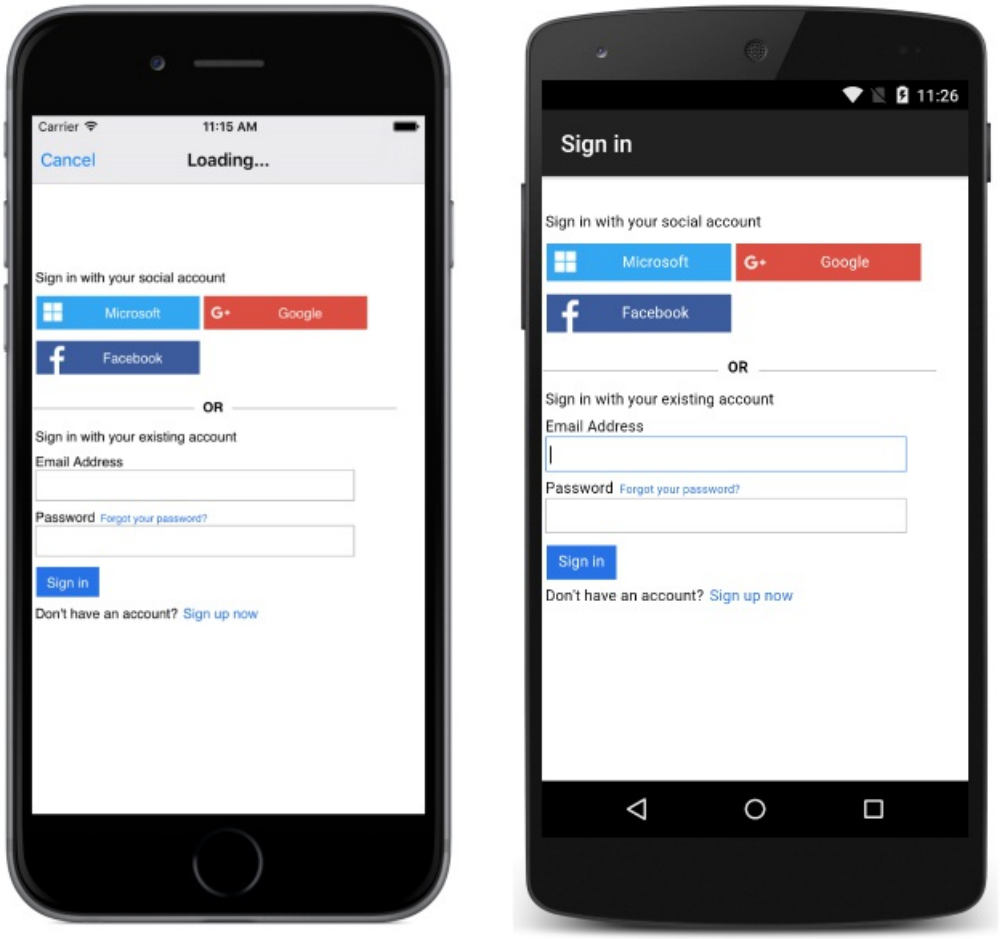
Microsoft 身份验证库使用的成员 `PublicClientApplication` 启动身份验证工作流的类。示例应用程序声明并初始化 `public` 名为此类型的属性 `ADB2CClient`，请在 `AuthenticationProvider` 类。下面的代码示例演示如何初始化此属性：

```
ADB2CClient = new PublicClientApplication(Constants.ClientID, Constants.Authority);
```

移动应用程序已注册到 Azure Active Directory B2C 租户，注册过程分配应用程序 **ID**。此 ID 必须以指定 `PublicClientApplication` 构造函数中，连同 `Authority` 常量，它包含要执行的基 URL，以及 Azure Active Directory B2C 策略。

在登录

下面的屏幕截图显示了示例应用程序中的登录屏幕：



单一登录与社交标识提供者，或使用本地帐户、允许使用。Microsoft、Google 和 Facebook，如上所示，用作社交标识提供者时还可以使用其他标识提供者。

下面的代码示例演示如何调用登录过程：

```
using Microsoft.Identity.Client;

public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult = await ADB2CClient.AcquireTokenAsync(
        Constants.Scopes,
        GetUserByPolicy(ADB2CClient.Users, Constants.PolicySignUpSignIn),
        App.UiParent);
    ...
}
```

`AcquireTokenAsync` 方法会启动设备的 web 浏览器并显示在通过引用的策略指定的 Azure Active Directory B2C 策略中定义的身份验证选项 `Constants.Authority` 常量。此策略定义期间注册和登录中使用者将经历的体验和应用程序将接收在成功注册或登录的声明。

结果 `AcquireTokenAsync` 方法调用是 `AuthenticationResult` 实例。如果身份验证成功，`AuthenticationResult` 实例将包含一个标识令牌，将本地缓存。如果身份验证不成功，`AuthenticationResult` 实例将包含数据，该值指示身份验证失败的原因。

在示例应用程序，如果身份验证成功，`TodoList` 页导航到。

无提示重新进行身份验证

当 `LoginPage` 在示例应用程序出现，尝试检索用户令牌，而不显示任何身份验证的用户界面。此，可以使用

`AcquireTokenSilentAsync` 方法，如下面的代码示例中所示：

```
public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult;

    if (useSilent)
    {
        authenticationResult = await ADB2CClient.AcquireTokenSilentAsync(
            Constants.Scopes,
            GetUserByPolicy(ADB2CClient.Users, Constants.PolicySignUpSignIn),
            Constants.Authority,
            false);
    }
    ...
}
```

`AcquireTokenSilentAsync` 方法尝试从缓存中检索用户令牌，而无需用户登录。这其中适合令牌可能已经出现在缓存中以前的会话处理方案。如果成功，尝试获取令牌 `TodoList` 页导航到。如果尝试获取的令牌不成功，会执行任何操作，所以用户必须启动新的身份验证工作流的选择。

正在注销

下面的代码示例演示如何调用注销过程：

```
public async Task<bool> LogoutAsync()
{
    ...
    foreach (var user in ADB2CClient.Users)
    {
        ADB2CClient.Remove(user);
    }
    ...
}
```

这将清除本地缓存中的所有身份验证令牌。

总结

本文演示了如何使用 Microsoft 身份验证库 (MSAL) 和 Azure Active Directory B2C 将用户标识管理集成到移动应用程序。Azure Active Directory B2C 是面向消费者的 web 和移动应用程序的云标识管理解决方案。

相关链接

- AzureADB2CAuth（示例）
- Azure Active Directory B2C
- Microsoft 身份验证库

将 Azure Active Directory B2C 集成与 Azure 移动应用程序

2018/7/12 • • Edit Online

Azure Active Directory B2C 是面向消费者的 web 和移动应用程序的云标识管理解决方案。本文演示如何使用 Azure Active Directory B2C 为 Xamarin.Forms 中提供身份验证和授权对 Azure 移动应用实例。

NOTE

Microsoft 身份验证库仍处于预览状态，但适用于在生产环境中使用。但是，那里可能重大更改的 API、内部缓存格式和库，这可能会影响你的应用程序的其他机制。

概述

Azure 移动应用让你能够使用移动身份验证、脱机同步和推送通知的支持在 Azure 应用服务中托管的可缩放后端进行开发的应用程序。有关 Azure 移动应用的详细信息，请参阅使用 Azure 移动应用，并使用 Azure 移动应用进行身份验证用户。

Azure Active Directory B2C 是标识管理服务的面向消费者的应用程序，允许使用者来登录到由应用程序：

- 使用现有社交帐户（Microsoft、Google、Facebook、Amazon、LinkedIn）。
- 创建新凭据（电子邮件地址和密码，或用户名和密码）。这些凭据*本地*帐户。

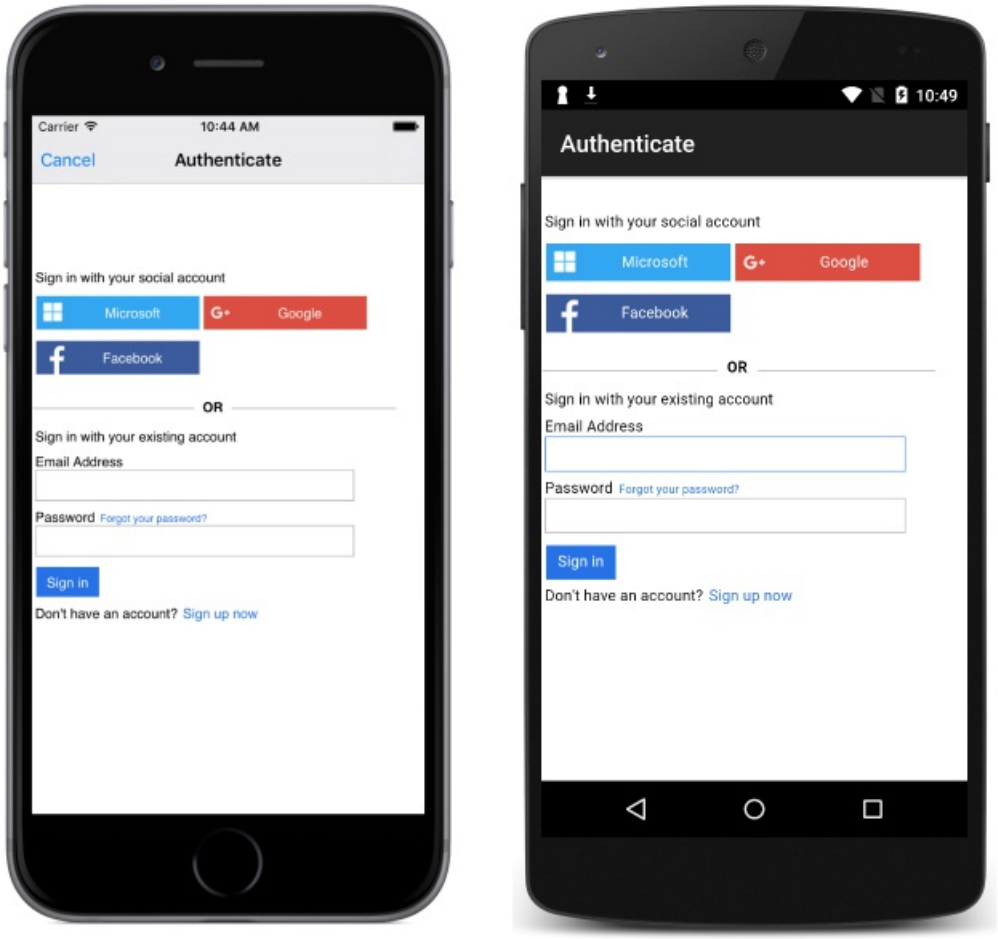
有关 Azure Active Directory B2C 的详细信息，请参阅与 Azure Active Directory B2C 进行身份验证用户。

可以使用 azure Active Directory B2C 来管理 Azure 移动应用身份验证工作流。使用此方法时，标识管理体验已完全定义为在云中，并可以修改而无需更改你的移动应用程序代码。

有与 Azure 移动应用实例集成的 Azure Active Directory B2C 租户时可采用的两个身份验证工作流：

- 客户端托管– 在此方法的身份验证过程使用 Azure Active Directory B2C 租户，则将 Xamarin.Forms 移动应用程序启动并将接收到的身份验证令牌传递给 Azure 移动应用实例。
- 服务器托管– 在这种方法在 Azure Mobile Apps 实例使用 Azure Active Directory B2C 租户启动的基于 web 的工作流通过身份验证过程。

在这两种情况下，由 Azure Active Directory B2C 租户提供的身份验证体验。在示例应用程序，这会导致登录屏幕中的以下屏幕截图所示：



单一登录与社交标识提供者，或使用本地帐户、允许使用。虽然 Microsoft、Google 和 Facebook 用作社交标识提供者在此示例中，还可以使用其他标识提供者。

安装

无论使用的身份验证工作流，与 Azure 移动应用实例集成的 Azure Active Directory B2C 租户的初始过程如下所示：

1. 创建 Azure 移动应用实例。有关详细信息，请参阅使用 Azure 移动应用。
2. 启用 Azure 移动应用实例和 Xamarin.Forms 应用程序中的身份验证。有关详细信息，请参阅使用 Azure 移动应用进行身份验证用户。
3. 创建 Azure Active Directory B2C 租户。有关详细信息，请参阅与 Azure Active Directory B2C 进行身份验证用户。

请注意，Microsoft 身份验证库 (MSAL) 需要在使用客户端托管的身份验证工作流。MSAL 使用设备的 web 浏览器来执行身份验证。这可提高应用程序的可用性，因为用户只需要在登录后每台设备，从而提高转换率的单一登录和授权流应用程序中。设备浏览器还提供了改进的安全性。用户完成身份验证过程后，控件将在 web 浏览器选项卡中，返回到应用程序。这被通过重定向 url 返回的身份验证过程中，检测并处理自定义 URL 后将其发送注册自定义的 URL 方案。有关使用 MSAL 来与 Azure Active Directory B2C 租户通信的详细信息，请参阅与 Azure Active Directory B2C 进行身份验证用户。

客户端托管的身份验证

在客户端托管的身份验证，Xamarin.Forms 移动应用程序联系启动身份验证流的 Azure Active Directory B2C 租户。后成功登录 Azure Active Directory B2C 租户将返回一个标识令牌，然后提供在登录到 Azure 移动应用实例。这样，Xamarin.Forms 应用程序需要身份验证的用户权限的 Azure 移动应用实例上执行操作。

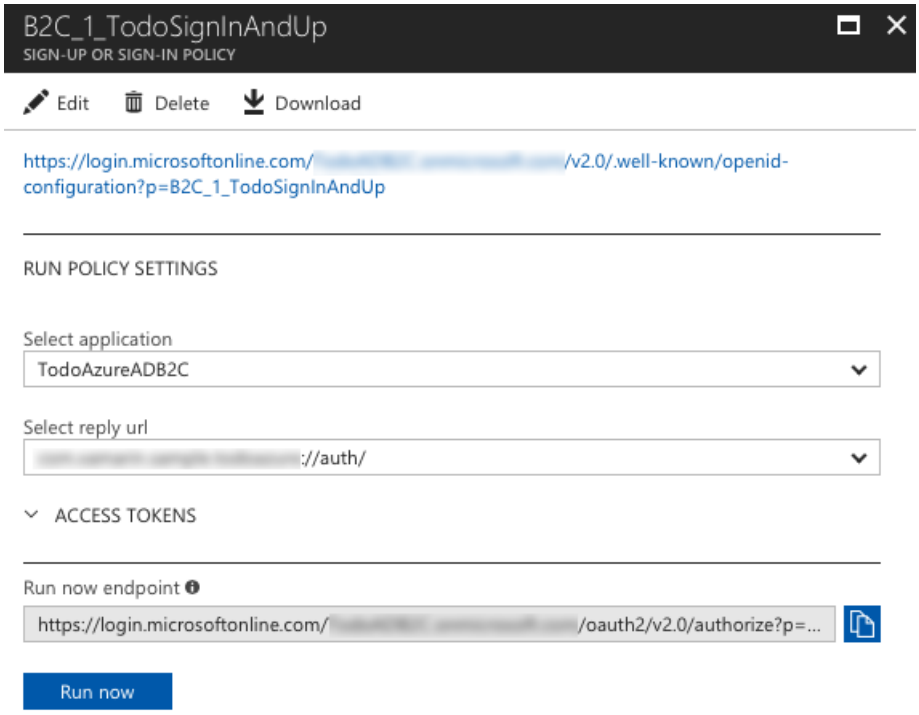
Azure Active Directory B2C 租户配置

客户端托管的身份验证的工作流，应按如下所示配置 Azure Active Directory B2C 租户：

- 包含本机客户端。
- 将自定义重定向 URI 设置为唯一地标识移动应用程序后，跟的 URL 方案 `://auth/`。有关选择自定义的 URL 方案的详细信息，请参阅选择本机应用重定向 URI。

下面的屏幕截图演示了此配置：

使用 Azure Active Directory B2C，回复 URL 设为相同的自定义 URL 方案，也可配置租户中的策略后跟 `://auth/`。
下面的屏幕截图演示了此配置：

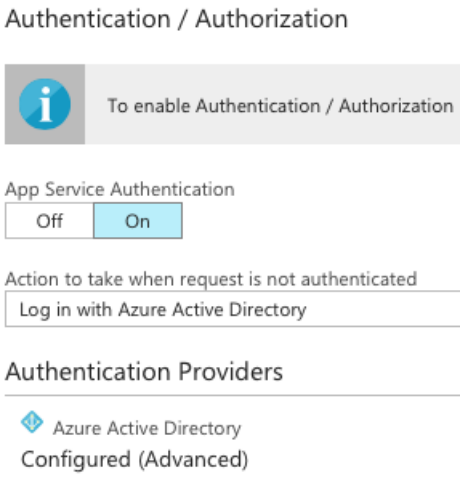


Azure 移动应用配置

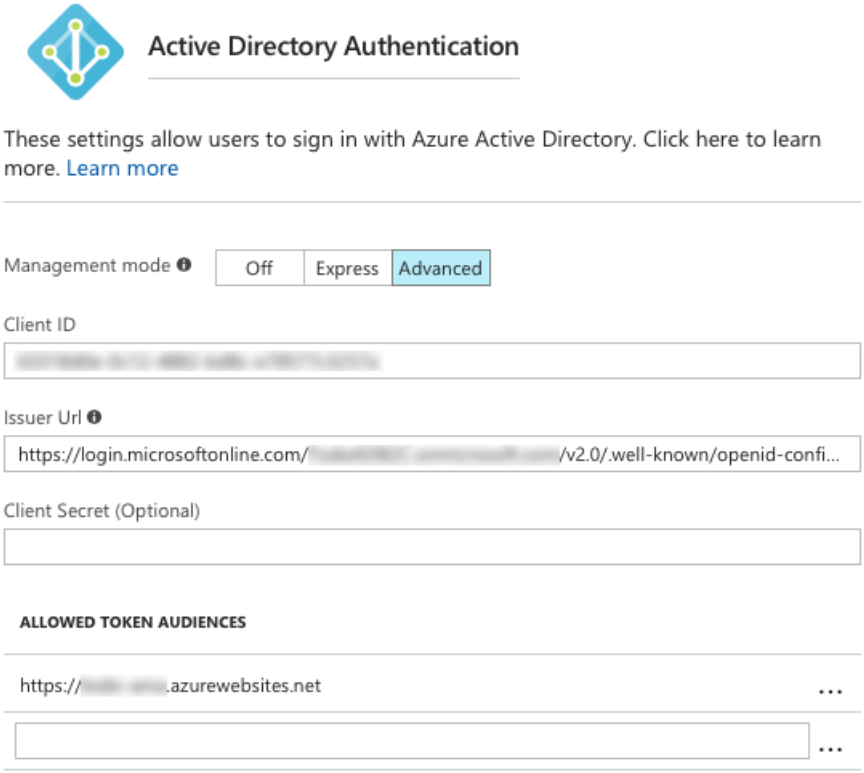
客户端托管的身份验证的工作流，应按如下所示配置 Azure 移动应用实例：

- 应启用应用服务身份验证。
- 当请求未经过身份验证时要执行的操作应设置为**Azure Active Directory 登录**。

下面的屏幕截图演示了此配置：



Azure 移动应用实例也应配置为与 Azure Active Directory B2C 租户。这可以通过启用高级 Azure Active Directory 身份验证提供程序，模式与客户端 ID 正在应用程序 ID 的 azureActive Directory B2C 租户，并颁发者 Url 正在 Azure Active Directory B2C 策略的元数据终结点。下面的屏幕截图演示了此配置：



在登录

下面的代码示例演示如何启动客户端托管的身份验证流：

```
public async Task<bool> LoginAsync(bool useSilent = false)
{
    ...
    AuthenticationResult authenticationResult = await ADB2CClient.AcquireTokenAsync(
        Constants.Scopes,
        GetUserByPolicy(ADB2CClient.Users, Constants.PolicySignUpSignIn),
        App.UiParent);

    ...
    var payload = new JObject();
    payload["access_token"] = authenticationResult.IdToken;

    User = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory,
        payload);
    ...
}
```

Microsoft 身份验证库 (MSAL) 用于与 Azure Active Directory B2C 租户启动的身份验证的工作流。 `AcquireTokenAsync` 方法会启动设备的 web 浏览器并显示在通过引用的策略指定的 Azure Active Directory B2C 策略中定义的身份验证选项 `Constants.Authority` 常量。此策略定义期间注册和登录中使用者将经历的体验和应用程序将接收在成功注册或登录的声明。

结果 `AcquireTokenAsync` 方法调用是 `AuthenticationResult` 实例。如果身份验证成功， `AuthenticationResult` 实例将包含一个标识令牌，将本地缓存。如果身份验证不成功， `AuthenticationResult` 实例将包含数据，该值指示身份验证失败的原因。有关如何使用 MSAL 来与 Azure Active Directory B2C 租户通信的信息，请参阅与 Azure Active Directory B2C 进行身份验证用户。

当 `MobileServiceClient.LoginAsync` 调用方法，Azure 移动应用实例收到的标识令牌包装在 `JObject` 。为有效的标记表示 Azure 移动应用实例并不需要启动其自己的 OAuth 2.0 身份验证流存在。相反， `MobileServiceClient.LoginAsync` 方法将返回 `MobileServiceUser` 实例，它将存储在 `MobileServiceClient.CurrentUser` 属性。此属性提供 `UserId` 和 `MobileServiceAuthenticationToken` 属性。这些表示已经过身份验证的用户和用户，可以在它过期之前的身份验证令牌。身份验证令牌将包含在对 Azure 移动应用实例，允许 Xamarin.Forms 应用程序需要身份验证的用户权限的 Azure 移动应用实例上执行操作所做的所有请求。

正在注销

下面的代码示例演示如何调用客户端托管的注销过程：

```
public async Task<bool> LogoutAsync()
{
    ...
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();

    foreach (var user in ADB2CClient.Users)
    {
        ADB2CClient.Remove(user);
    }
    ...
}
```

`MobileServiceClient.LogoutAsync` 方法取消对使用 Azure 移动应用实例中，用户身份验证，则所有身份验证令牌会清除从创建的 MSAL 的本地缓存。

服务器托管的身份验证

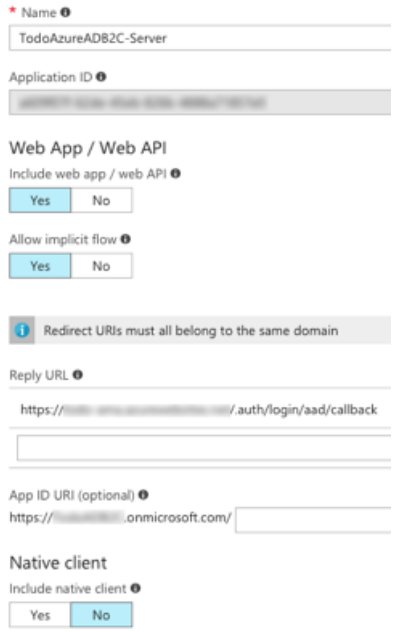
服务器托管的身份验证，在 Xamarin.Forms 应用程序联系 Azure 移动应用实例，使用 Azure Active Directory B2C 租户来管理 OAuth 2.0 身份验证流的 B2C 策略中定义显示在登录页。以下成功登录，Azure 移动应用实例返回令牌，可让 Xamarin.Forms 应用程序需要身份验证的用户权限的 Azure 移动应用实例上执行操作。

Azure Active Directory B2C 租户配置

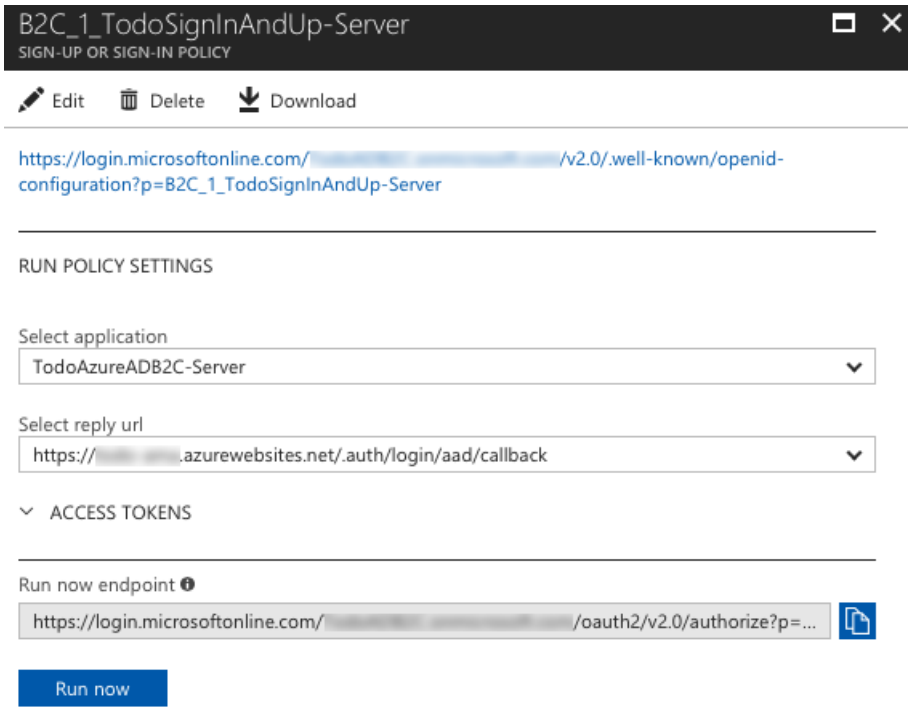
服务器托管的身份验证的工作流，应按如下所示配置 Azure Active Directory B2C 租户：

- 包括 web 应用/web API，并允许隐式流。
- 答复 URL 设置为 Azure 移动应用的地址后，跟 `/.auth/login/aad/callback`。

下面的屏幕截图演示了此配置：



使用 Azure Active Directory B2C，回复 URL 设为 Azure 移动应用的地址，还应该配置租户中的策略后跟 `/.auth/login/aad/callback`。下面的屏幕截图演示了此配置：

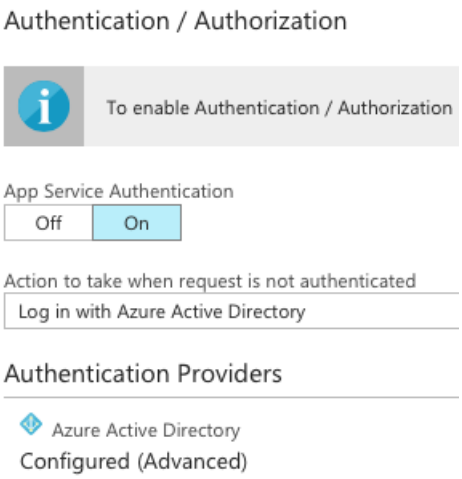


Azure 移动应用实例配置

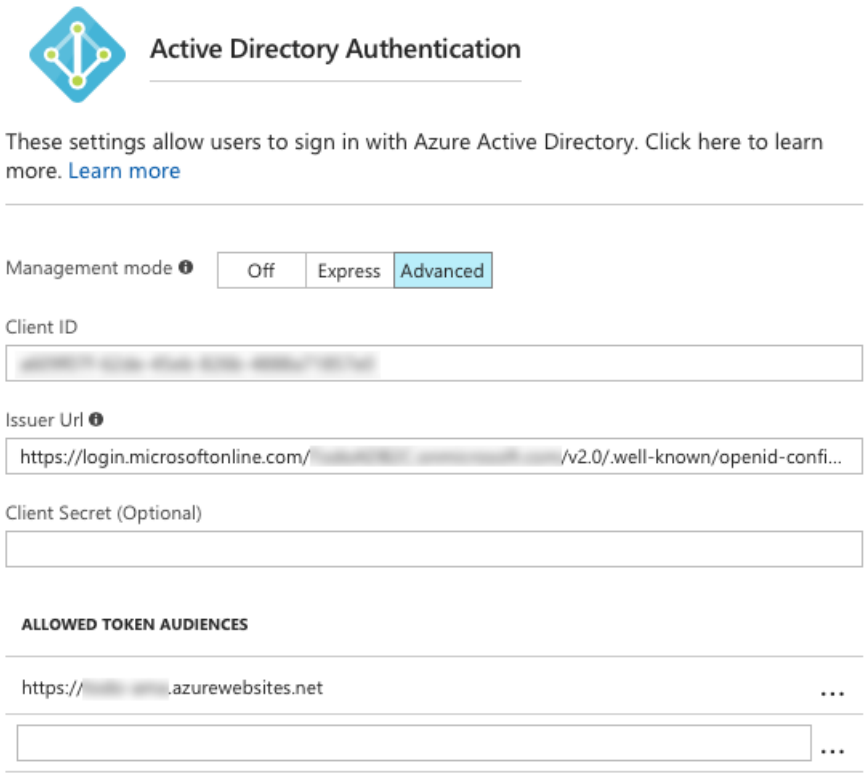
服务器托管的身份验证的工作流，应按如下所示配置 Azure 移动应用实例：

- 应启用应用服务身份验证。
- 当请求未经过身份验证时要执行的操作应设置为**Azure Active Directory 登录**。

下面的屏幕截图演示了此配置：



Azure 移动应用实例也应配置为与 Azure Active Directory B2C 租户。这可以通过启用高级 Azure Active Directory 身份验证提供程序，模式与客户端 ID 正在应用程序 ID 的 azureActive Directory B2C 租户，并颁发者 Url 正在 Azure Active Directory B2C 策略的元数据终结点。下面的屏幕截图演示了此配置：



在登录

下面的代码示例演示如何启动服务器托管的身份验证流：

```
public async Task<bool> AuthenticateAsync()
{
    ...
    MobileServiceUser user = await TodoItemManager.DefaultManager.CurrentClient.LoginAsync(
        UIApplication.SharedApplication.KeyWindow.RootViewController,
        MobileServiceAuthenticationProvider.WindowsAzureActiveDirectory,
        Constants.URLScheme);
    ...
}
```

当 `MobileServiceClient.LoginAsync` 调用方法，Azure 移动应用实例执行链接的 Azure Active Directory B2C 策略，从而启动 OAuth 2.0 身份验证流。请注意，每个 `AuthenticateAsync` 方法是特定于平台的。但是，每个 `AuthenticateAsync` 方法使用 `MobileServiceClient.LoginAsync` 方法并指定将在身份验证过程中使用 Azure Active

Directory 租户。有关详细信息，请参阅在用户登录。

`MobileServiceClient.LoginAsync` 方法将返回 `MobileServiceUser` 实例，它将存储在 `MobileServiceClient.CurrentUser` 属性。此属性提供 `UserId` 和 `MobileServiceAuthenticationToken` 属性。这些表示已经过身份验证的用户和用户，可以在它过期之前的身份验证令牌。身份验证令牌将包含在对 Azure 移动应用实例，允许 Xamarin.Forms 应用程序需要身份验证的用户权限的 Azure 移动应用实例上执行操作所做的所有请求。

正在注销

下面的代码示例演示如何调用服务器托管的注销过程：

```
public async Task<bool> LogoutAsync()
{
    ...
    await TodoItemManager.DefaultManager.CurrentClient.LogoutAsync();
    ...
}
```

`MobileServiceClient.LogoutAsync` 方法取消用户进行身份验证与 Azure 移动应用实例。有关详细信息，请参阅日志记录用户。

总结

本文演示了如何使用 Azure Active Directory B2C 为 Xamarin.Forms 中提供身份验证和授权对 Azure 移动应用实例。Azure Active Directory B2C 是面向消费者的 web 和移动应用程序的云标识管理解决方案。

相关链接

- TodoAzureAuth ServerFlow（示例）
- TodoAzureAuth ClientFlow（示例）
- 使用 Azure 移动应用
- 使用 Azure 移动应用的用户进行身份验证
- 使用 Azure Active Directory B2C 的用户进行身份验证
- Microsoft 身份验证库

与 Web 服务之间同步数据

2018/6/22 • • Edit Online

脱机同步允许用户交互与移动应用程序、查看、添加或修改数据，甚至在没有网络连接。更改将存储在本地数据库中，并且一旦设备处于联机状态，可以与 web 服务同步所做的更改。

使用 Azure 移动应用程序中同步脱机数据

此文章介绍了如何向 Xamarin.Forms 应用程序添加脱机同步功能。

相关链接

- Web 服务简介
- 异步支持概述

使用 Azure 移动应用程序中同步脱机数据

2018/6/9 • • Edit Online

脱机同步允许用户交互与移动应用程序、查看、添加或修改数据，甚至在没有网络连接。更改将存储在本地数据库中，并且一旦设备处于联机状态，可以与 Azure Mobile Apps 实例同步所做的更改。此文章介绍了如何向 Xamarin.Forms 应用程序添加脱机同步功能。

概述

Azure 移动客户端 SDK提供 `IMobileServiceTable` 接口，表示可以在表存储在 Azure Mobile Apps 实例执行的操作。这些操作直接连接到 Azure Mobile Apps 实例，并且如果移动设备没有相应的网络连接将失败。

若要支持脱机同步，Azure 移动客户端 SDK 支持*同步表*，这由 `IMobileServiceSyncTable` 接口。此接口可提供相同的创建、读取、更新、删除 (CRUD) 操作作为 `IMobileServiceTable` 接口，但操作读取或写入本地存储。本地存储区不填充使用从 Azure Mobile Apps 实例的新数据对的调用之前*请求*数据。同样，数据对的调用之前未发送到 Azure Mobile Apps 实例*推送*本地更改。

脱机同步还包括支持在 Azure Mobile Apps 实例和自定义冲突解决这两个本地存储区中和已更改同一记录时检测冲突。在本地存储中，或 Azure Mobile Apps 实例中，也可以处理冲突。

有关脱机同步的详细信息，请参阅Azure Mobile Apps 中的脱机数据同步和启用 Xamarin.Forms 移动应用程序的脱机同步。

安装

将集成 Xamarin.Forms 应用程序与 Azure Mobile Apps 实例之间的脱机同步的过程如下所示：

1. 创建 Azure Mobile Apps 实例。有关详细信息，请参阅使用 Azure 移动应用。
2. 添加Microsoft.Azure.Mobile.Client.SQLiteStore Xamarin.Forms 解决方案中的所有项目的 NuGet 包。
3. （可选）启用 Azure Mobile Apps 实例和 Xamarin.Forms 应用程序中的身份验证。有关详细信息，请参阅进行身份验证的用户与 Azure Mobile Apps。

以下部分提供有关配置通用 Windows 平台 (UWP) 项目的其他安装说明使用 Microsoft.Azure.Mobile.Client.SQLiteStore NuGet 包。没有其他安装程序需要在 iOS 和 Android 使用 Microsoft.Azure.Mobile.Client.SQLiteStore NuGet 包。

通用 Windows 平台

若要使用 SQLite 通用 Windows 平台 (UWP)，请按照下列步骤：

1. 安装的通用 Windows 平台的 SQLite在开发环境中的 Visual Studio 扩展。
2. 在 Visual Studio 中的 UWP 项目，右键单击**引用 > 添加引用**，导航到扩展并添加的**通用 Windows 平台的 SQLite**和**通用 Windows 平台**应用的 **visual c + + 2015年运行时**扩展到 UWP 项目。

初始化线程本地存储区

可以执行任何同步表操作之前，必须初始化本地存储。这是在可移植类库 (PCL) 项目中的 Xamarin.Forms 解决方案来实现的：

```
using Microsoft.WindowsAzure.MobileServices;
using Microsoft.WindowsAzure.MobileServices.SQLiteStore;
using Microsoft.WindowsAzure.MobileServices.Sync;

namespace TodoAzure
{
    public partial class TodoItemManager
    {
        static TodoItemManager defaultInstance = new TodoItemManager();
        IMobileServiceClient client;
        IMobileServiceSyncTable<TodoItem> todoTable;

        private TodoItemManager()
        {
            this.client = new MobileServiceClient(Constants.ApplicationURL);
            var store = new MobileServiceSQLiteStore("localstore.db");
            store.DefineTable<TodoItem>();
            this.client.SyncContext.InitializeAsync(store);
            this.todoTable = client.GetSyncTable<TodoItem>();
        }
        ...
  }
}
```

通过创建一个新的本地 SQLite 数据库 `MobileServiceSQLiteStore` 类，前提是它尚不存在。然后，`DefineTable<T>` 方法中的字段相匹配的本地存储中创建一个表 `TodoItem` 类型，前提是它尚不存在。

A *同步上下文* 与关联 `MobileServiceClient` 实例，并使用同步表进行的跟踪更改。同步上下文维护的队列中保留有序列的表的创建、更新和删除 (CUD) 操作将发送到 Azure Mobile Apps 实例更高版本。
`IMobileServiceSyncContext.InitializeAsync()` 方法用于将本地存储与同步上下文相关联。

`todoTable` 字段是 `IMobileServiceSyncTable`，因此，所有 CRUD 操作都使用本地存储。

执行同步

与 Azure Mobile Apps 同步本地存储区实例时 `SyncAsync` 调用方法：

```
public async Task SyncAsync()
{
  ReadOnlyCollection<MobileServiceTableOperationError> syncErrors = null;

  try
  {
    await this.client.SyncContext.PushAsync();

    // The first parameter is a query name that is used internally by the client SDK to implement incremental
sync.
    // Use a different query name for each unique query in your program.
    await this.todoTable.PullAsync("allTodoItems", this.todoTable.CreateQuery());
  }
  catch (MobileServicePushFailedException exc)
  {
    if (exc.PushResult != null)
    {
      syncErrors = exc.PushResult.Errors;
    }
  }

  // Simple error/conflict handling.
  if (syncErrors != null)
  {
    foreach (var error in syncErrors)
    {
      if (error.OperationKind == MobileServiceTableOperationKind.Update && error.Result != null)
      {
        // Update failed, revert to server's copy
        await error.CancelAndUpdateItemAsync(error.Result);
      }
      else
      {
        // Discard local change
        await error.CancelAndDiscardItemAsync();
      }

      Debug.WriteLine(@"Error executing sync operation. Item: {0} ({1}). Operation discarded.",
error.TableName, error.Item["id"]);
    }
  }
}
```

`IMobileServiceSyncTable.PushAsync` 方法对同步上下文中，而不是特定的表，并将所有 CUD 更改都发送自上次推送。

请求由执行 `IMobileServiceSyncTable.PullAsync` 对单个表的方法。第一个参数 `PullAsync` 方法是仅在移动设备使用的查询名称。提供在 Azure 移动客户端 SDK 执行名称结果的非 null 查询*增量同步*，其中每次请求操作返回的结果，最新 `updatedAt` 从结果的时间戳存储在局部变量系统表。将后续请求操作然后只能检索记录后该时间戳。或者，*完全同步*可以通过传递 `null` 作为查询名称，这会导致正在检索每个请求操作上的所有记录。任何同步操作后，接收到的数据插入到本地存储中。

> **NOTE**
> 如果针对具有挂起的本地更新的表执行请求，则请求将先执行推送对同步上下文。这将减少已排入队列的更改和新数据从 Azure Mobile Apps 实例之间的冲突。

`SyncAsync` 方法还包含用于处理冲突，这两个本地存储区中和在 Azure Mobile Apps 实例中更改同一记录时的基本实现。在本地存储区和在 Azure Mobile Apps 实例中，数据已被更新冲突时 `SyncAsync` 方法将更新存储在 Azure Mobile Apps 实例中的数据从本地存储中的数据。当任何其他冲突发生时，`SyncAsync` 方法会放弃本地更改。这将处理方案本地更改所在的 Azure Mobile Apps 实例从已删除的数据。

在生产应用程序，开发人员应编写自定义 `IMobileServiceSyncHandler` 适合其使用大小写的冲突处理实现。有关详细信息，请参阅使用开放式并发冲突解决在 Azure 门户中，和深入了解上托管的客户端 SDK 中的脱机支持MSDN 博客。

清除数据

本地存储中的表可以清除的数据与 `IMobileServiceSyncTable.PurgeAsync` 方法。此方法支持删除应用程序不再需要的过时数据等方案。例如，示例应用程序仅显示 `TodoItem` 未完成的实例。因此，已完成的项目不再需要在本地存储。清除本地存储中的已完成的项目完成，如下所示：

```
await todoTable.PurgeAsync(todoTable.Where(item => item.Done));
```

调用 `PurgeAsync` 还会触发推送操作。因此，本地标记为已完成的任何项将从本地存储区中删除之前发送到 Azure Mobile Apps 实例中。但是，如果有与 Azure Mobile Apps 实例同步挂起的操作，则清除将引发 `InvalidOperationException` 除非 `force` 参数设置为 `true` 。使用替代策略是检查 `IMobileServiceSyncContext.PendingOperations` 属性，它返回挂起的操作尚未已推送到 Azure Mobile Apps 实例，并仅执行清除，如果属性为零的数。

> **NOTE**
> 调用 `PurgeAsync` 与 `force` 参数设置为 `true` 将丢失任何挂起的更改。

启动同步

在示例应用程序，`SyncAsync` 通过调用方法 `TodoList.OnAppearing` 方法：

```
protected override async void OnAppearing()
{
  base.OnAppearing();

  // Set syncItems to true to synchronize the data on startup when running in offline mode
  await RefreshItems(true, syncItems: true);
}
```

这意味着应用程序将尝试启动时与 Azure Mobile Apps 实例同步。

此外，同步可在 iOS 和 Android 由启动请求用于使用刷新和 Windows 平台上的数据，列表中同步用户界面上的按钮。有关详细信息，请参阅拉取到刷新。

总结

本文介绍如何向 Xamarin.Forms 应用程序添加脱机同步功能。脱机同步允许用户交互与移动应用程序、查看、添加或修改数据，甚至在没有网络连接。更改将存储在本地数据库中，并且一旦设备处于联机状态，可以与 Azure Mobile Apps 实例同步所做的更改。

相关链接

- TodoAzureAuthOfflineSync（示例）
- 使用 Azure 移动应用
- 使用 Azure 移动应用程序的用户进行身份验证
- Azure 的移动客户端 SDK
- MobileServiceClient

发送推送通知

2018/6/22 • • Edit Online

推送通知使用以提供信息，例如一条消息，包括从后端系统上的移动设备的应用程序，以提高应用程序关注度和使用率。可以发送通知在任何时候，即使用户不活跃地使用目标应用程序。

从 Azure 移动应用程序发送推送通知

Azure 通知中心提供可缩放的推送基础结构用于发送移动推送通知从任意后端向任何移动平台，同时无后端不必与不同的平台通知系统通信的复杂性。

从 Azure 移动应用程序发送推送通知

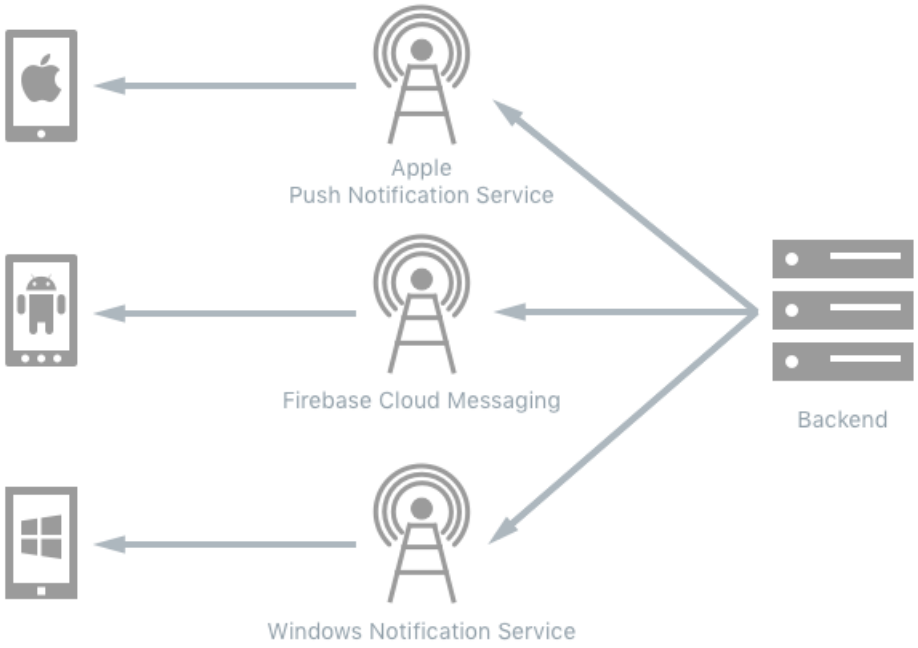
2018/6/9 • • Edit Online

Azure 通知中心提供可缩放的推送基础结构用于发送移动推送通知从任意后端向任何移动平台，同时无后端不必与不同的平台通知系统通信的复杂性。此文章介绍了如何使用 Azure 通知中心将推送通知从 Azure Mobile Apps 实例发送到 Xamarin.Forms 应用程序。

Azure 通过推送通知中心与 Xamarin.Forms，Xamarin 大学

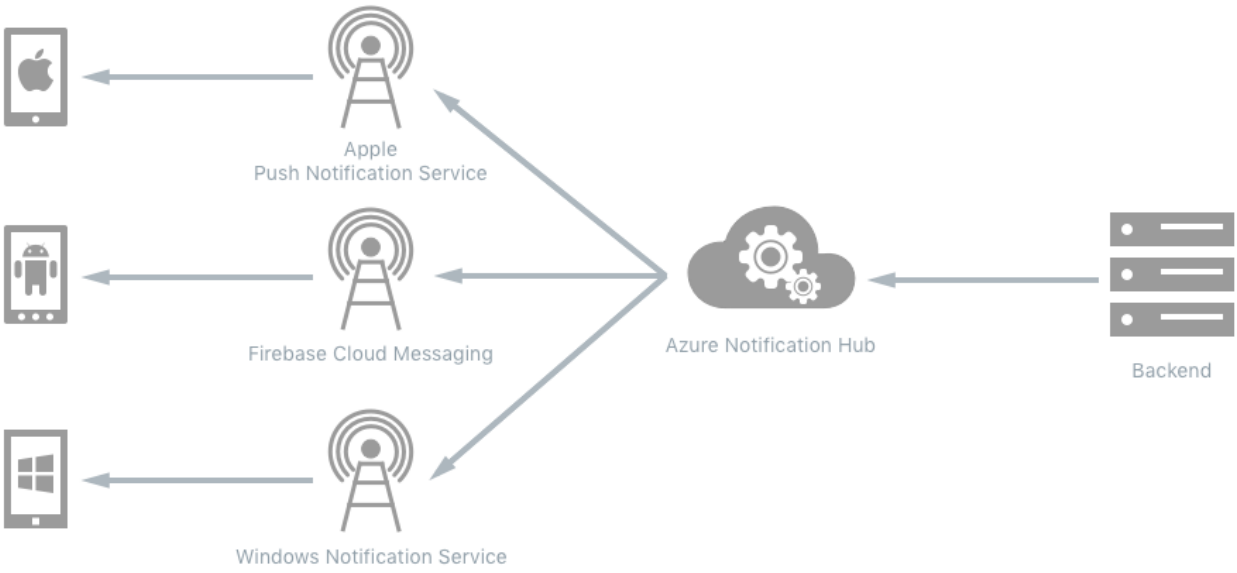
推送通知使用以提供信息，例如一条消息，包括从后端系统上的移动设备的应用程序，以提高应用程序关注度和使用率。可以发送通知在任何时候，即使用户不活跃地使用目标应用程序。

后端系统将推送通知发送到移动设备通过平台通知系统 (PNS) 下，图中所示：



若要发送推送通知后，端系统，请联系特定于平台的 PNS 将通知发送到客户端应用程序实例。这会大大提高的复杂性的后端时跨平台推送通知是必需的因为后端必须使用每个特定于平台的 PNS API 和协议。

Azure 通知中心消除这种复杂性由提取的详细信息的不同平台通知系统，允许跨平台通知的发送一次的 API 调用，如下面的关系图中所示：

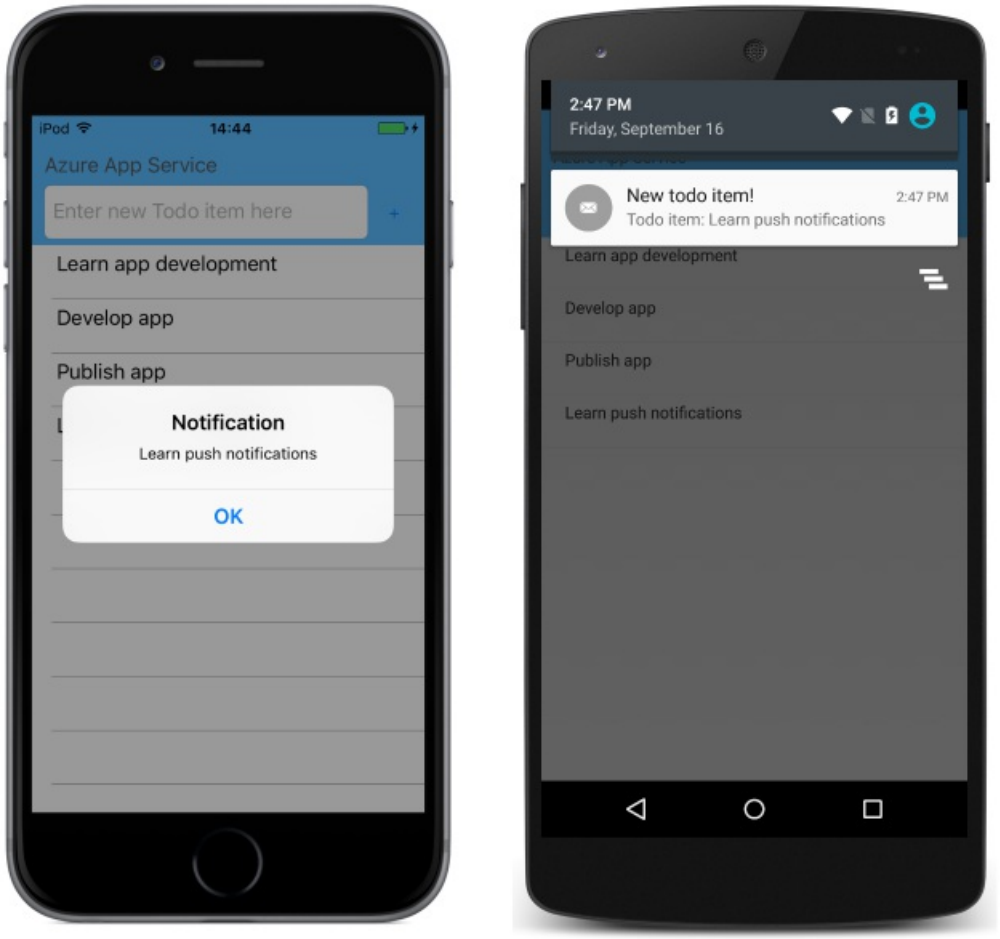


若要发送推送通知后，端系统仅联系人 Azure 通知中心，这反过来使用不同平台通知系统进行通信，从而减小的后端复杂性代码该发送推送通知。

Azure 移动应用都使用通知中心的推送通知的内置支持。从 Azure Mobile Apps 实例向 Xamarin.Forms 应用程序发送推送通知的过程如下所示：

1. Xamarin.Forms 应用程序注册 PNS，返回的句柄。
2. Azure Mobile Apps 实例将通知发送到 Azure 通知中心，指定将作为目标设备的句柄。
3. Azure 通知中心将通知发送到相应的 PNS 设备。
4. PNS 将通知发送到指定的设备。
5. Xamarin.Forms 应用程序处理通知，并显示它。

示例应用程序演示其数据存储在 Azure Mobile Apps 实例 todo 列表应用程序。每次新项添加到 Azure Mobile Apps 实例中，一条推送通知发送到 Xamarin.Forms 应用程序。以下屏幕快照显示的每个平台显示已接收的推送通知：



有关 Azure 通知中心的详细信息，请参阅Azure 通知中心和向 Xamarin.Forms 应用添加推送通知。

Azure 和平台通知系统设置

将 Azure 通知中心集成到 Azure Mobile Apps 实例的过程如下所示：

1. 创建 Azure Mobile Apps 实例。有关详细信息，请参阅使用 Azure 移动应用。
2. 配置通知中心。有关详细信息，请参阅配置通知中心。
3. 更新要发送推送通知的 Azure Mobile Apps 实例。有关详细信息，请参阅更新服务器项目以发送推送通知。
4. 注册每个 PNS。
5. 配置通知中心与每个 PNS 进行通信。

以下部分提供每个平台的其他设置说明。

iOS

若要使用 Apple 推送通知服务 (APNS) 从 Azure 通知中心，必须开展以下附加步骤：

1. 生成证书签名请求与 Keychain Access 工具的推送证书。有关详细信息，请参阅生成的推送证书的证书签名请求文件Azure 文档中心上。
2. 注册 Apple 开发人员中心上的推送通知支持 Xamarin.Forms 应用程序。有关详细信息，请参阅你应用程序注册推送通知Azure 文档中心上。
3. 在 Apple 开发人员中心上创建的推送通知启用预配配置文件 Xamarin.Forms 应用程序。有关详细信息，请参阅创建应用程序的预配配置文件Azure 文档中心上。
4. 配置通知中心与 APNS 通信。有关详细信息，请参阅的 APNS 配置通知中心。
5. Xamarin.Forms 应用程序配置为使用新的应用程序 ID 和预配配置文件。有关详细信息，请参阅在 Xamarin Studio 中配置的 iOS 项目或在 Visual Studio 中配置的 iOS 项目Azure 文档中心上。

Android

若要使用 Firebase 云消息传送 (FCM) 从 Azure 通知中心，必须开展以下附加步骤：

1. 注册 FCM。服务器 API 密钥和客户端 ID 是自动生成并打包 `google-services.json` 下载的文件。有关详细信息，请参阅启用 Firebase 云消息传送 (FCM)。
2. 配置通知中心与 FCM 进行通信。有关详细信息，请参阅配置移动应用回结束以发送推送请求使用 FCM。

通用 Windows 平台

若要使用 Azure 通知中心从 Windows 通知服务 (WNS)，必须开展以下附加步骤：

1. 为 Windows 通知服务 (WNS) 注册。有关详细信息，请参阅注册到 WNS 的推送通知你 Windows 应用Azure 文档中心上。
2. 配置通知中心使用 WNS 通信。有关详细信息，请参阅的 WNS 配置通知中心Azure 文档中心上。

向 Xamarin.Forms 应用程序添加推送通知支持

以下各节讨论在每个特定于平台的项目，需要支持推送通知的实现。

iOS

在 iOS 应用程序中实现的推送通知支持的过程如下所示：

1. 注册与 Apple 推送通知服务 (APNS) 在 `AppDelegate.FinishedLaunching` 方法。有关详细信息，请参阅向 Apple 推送通知系统注册。
2. 实现 `AppDelegate.RegisteredForRemoteNotifications` 方法以处理的注册响应。有关详细信息，请参阅处理注册响应。
3. 实现 `AppDelegate.DidReceiveRemoteNotification` 方法来处理传入的推送通知。有关详细信息，请参阅处理传入的推送通知。

注册 Apple Push Notification 服务

IOS 应用程序可收到推送通知之前，它必须注册与 Apple 推送通知服务 (APNS)，这将生成唯一的设备令牌并将其返回到应用程序。在调用注册 `FinishedLaunching` 中重写 `AppDelegate` 类：

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    ...
    var settings = UIUserNotificationSettings.GetSettingsForTypes(
        UIUserNotificationType.Alert, new NSSet());

    UIApplication.SharedApplication.RegisterUserNotificationSettings(settings);
    UIApplication.SharedApplication.RegisterForRemoteNotifications();
    ...
}
```

当通过 APNS 注册的 iOS 应用程序时，它必须指定想要接收的推送通知的类型。 `RegisterUserNotificationSettings` 方法注册的应用程序可以接收，通知的类型与 `RegisterForRemoteNotifications` 注册从 APNS 中接收推送通知的方法。

> **NOTE**
> 未能调用 `RegisterUserNotificationSettings` 方法将导致应用程序以无提示方式接收的推送通知。

处理的注册响应

APNS 注册请求将在后台发生。当收到响应时，将调用 iOS `RegisteredForRemoteNotifications` 中重写 `AppDelegate` 类：

```
public override void RegisteredForRemoteNotifications(UIApplication application, NSData deviceToken)
{
    const string templateBodyAPNS = "{\"aps\":{\"alert\":\"$(messageParam)\"}}";

    JObject templates = new JObject();
    templates["genericMessage"] = new JObject
    {
        {"body", templateBodyAPNS}
    };

    // Register for push with the Azure mobile app
    Push push = TodoItemManager.DefaultManager.CurrentClient.GetPush();
    push.RegisterAsync(deviceToken, templates);
}
```

此方法创建简单的通知消息模板为 JSON，并注册设备以接收从通知中心发送模板通知。

> **NOTE**
> `FailedToRegisterForRemoteNotifications` 应实现重写用于处理无网络连接等情况。这是重要的因为用户可能会启动时应用程序脱机。

处理传入的推送通知

`DidReceiveRemoteNotification` 中重写 `AppDelegate` 类用于处理传入的推送通知，当应用程序正在运行，而收到通知时调用：

```
public override void DidReceiveRemoteNotification(
    UIApplication application, NSDictionary userInfo, Action<UIBackgroundFetchResult> completionHandler)
{
    NSDictionary aps = userInfo.ObjectForKey(new NSString("aps")) as NSDictionary;

    string alert = string.Empty;
    if (aps.ContainsKey(new NSString("alert")))
        alert = (aps[new NSString("alert")] as NSString).ToString();

    // Show alert
    if (!string.IsNullOrEmpty(alert))
    {
      var notificationAlert = UIAlertController.Create("Notification", alert, UIAlertControllerStyle.Alert);
      notificationAlert.AddAction(UIAlertAction.Create("OK", UIAlertActionStyle.Cancel, null));
      UIApplication.SharedApplication.KeyWindow.RootViewController.PresentViewController(notificationAlert,
true, null);
    }
}
```

`userInfo` 字典包含 `aps` 键，其值是 `alert` 利用剩余的通知数据的字典。检索此字典时，与 `string` 在对话框中显示的通知消息。

> **NOTE**
> 如果应用程序未运行在一条推送通知到达时，将启动应用程序但 `DidReceiveRemoteNotification` 方法不会处理通知。相反，获取通知负载并做出相应响应从 `WillFinishLaunching` 或 `FinishedLaunching` 重写。

有关 APNS 的详细信息，请参阅在 iOS 中的推送通知。

Android

在 Android 应用程序中实现的推送通知支持的过程如下所示：

1. 添加Xamarin.Firebase.Messaging NuGet 包到的 Android 项目，并设置应用程序的目标版本为 Android 7.0 或更高。
2. 添加 `google-services.json` 文件，从 Firebase 控制台中，下载到 Android 项目的根目录，并将其生成操作设置为**GoogleServicesJson**。有关详细信息，请参阅添加 Google 服务 JSON 文件。
3. 通过声明 Android 清单中的接收方注册与 Firebase 云消息传送 (FCM) 文件中，并通过实现 `FirebaseRegistrationService.OnTokenRefresh` 方法。有关详细信息，请参阅Firebase Cloud Messaging 注册。
4. 注册到 Azure 通知中心 `AzureNotificationHubService.RegisterAsync` 方法。有关详细信息，请参阅向 Azure 通知中心注册。
5. 实现 `FirebaseNotificationService.OnMessageReceived` 方法来处理传入的推送通知。有关详细信息，请参阅显示内容的推送通知。

有关 Firebase 云消息传送的详细信息，请参阅Firebase Cloud Messaging和远程通知 Firebase Cloud Messaging。

注册 Firebase 云消息传送

Android 应用程序可收到推送通知之前，它必须注册 FCM，这将生成的注册令牌并将其返回到应用程序。有关注册令牌的详细信息，请参阅注册到 FCM。

这是通过实现的：

- 声明 Android 清单中的接收方。有关详细信息，请参阅声明 Android 清单中的接收方。
- 实现 Firebase 实例 ID 服务。有关详细信息，请参阅实现 Firebase 实例 ID 服务。

声明 Android 清单中的接收方

编辑**AndroidManifest.xml**和插入以下 `<receiver>` 元素到 `<application>` 元素：

```
<receiver android:name="com.google.firebase.iid.FirebaseInstanceIdInternalReceiver" android:exported="false"
/>
<receiver android:name="com.google.firebase.iid.FirebaseInstanceIdReceiver" android:exported="true"
android:permission="com.google.android.c2dm.permission.SEND">
  <intent-filter>
    <action android:name="com.google.android.c2dm.intent.RECEIVE" />
    <action android:name="com.google.android.c2dm.intent.REGISTRATION" />
    <category android:name="${applicationId}" />
  </intent-filter>
</receiver>
```

此 XML 将执行以下操作：

- 声明内部 `FirebaseInstanceIdInternalReceiver` 用于安全地启动服务的实现。
- 声明 `FirebaseInstanceIdReceiver` 实现，提供有关每个应用程序实例的唯一标识符。此接收器还进行身份验证和授权操作。

`FirebaseInstanceIdReceiver` 接收 `FirebaseInstanceId` 和 `FirebaseMessaging` 事件和传递到类派生自这些 `FirebasesInstanceIdService`。

实现 Firebase 实例 ID 服务

FCM 中注册应用程序通过派生类从 `FirebaseInstanceIdService` 类。此类负责生成授权客户端应用程序访问 FCM 的安全令牌。在示例应用程序 `FirebaseRegistrationService` 类派生自 `FirebaseInstanceIdService` 类并在下面的代码示例所示：

```
[Service]
[IntentFilter(new[] { "com.google.firebase.INSTANCE_ID_EVENT" })]
public class FirebaseRegistrationService : FirebaseInstanceIdService
{
    const string TAG = "FirebaseRegistrationService";

    public override void OnTokenRefresh()
    {
        var refreshedToken = FirebaseInstanceId.Instance.Token;
        Log.Debug(TAG, "Refreshed token: " + refreshedToken);
        SendRegistrationTokenToAzureNotificationHub(refreshedToken);
    }

    void SendRegistrationTokenToAzureNotificationHub(string token)
    {
        // Update notification hub registration
        Task.Run(async () =>
        {
            await
AzureNotificationHubService.RegisterAsync(TodoItemManager.DefaultManager.CurrentClient.GetPush(), token);
        });
    }
}
```

`OnTokenRefresh` 当应用程序收到来自 FCM 注册令牌时，会调用方法。此方法检索从令牌 `FirebaseInstanceId.Instance.Token` 属性，用于通过 FCM 以异步方式更新。`OnTokenRefresh` 很少调用方法，因为安装或卸载，当用户中删除应用程序数据时应用程序清除实例 ID，应用程序时，才会更新令牌时的安全令牌已被或泄露。此外，FCM 实例 ID 服务将请求，应用程序刷新其令牌定期，通常每 6 个月。

`OnTokenRefresh` 方法也会调用 `SendRegistrationTokenToAzureNotificationHub` 方法，用于将用户的注册令牌与 Azure 通知中心相关联。

向 Azure 通知中心注册

`AzureNotificationHubService` 类提供 `RegisterAsync` 方法，将用户的注册令牌与 Azure 通知中心相关联。下面的代码示例演示 `RegisterAsync` 方法，通过调用 `FirebaseRegistrationService` 类用户的注册令牌更改时：

```
public class AzureNotificationHubService
{
    const string TAG = "AzureNotificationHubService";

    public static async Task RegisterAsync(Push push, string token)
    {
        try
        {
            const string templateBody = "{\"data\":{\"message\":\"$(messageParam)\"}}";
            JObject templates = new JObject();
            templates["genericMessage"] = new JObject
            {
                {"body", templateBody}
            };

            await push.RegisterAsync(token, templates);
            Log.Info("Push Installation Id: ", push.InstallationId.ToString());
        }
        catch (Exception ex)
        {
            Log.Error(TAG, "Could not register with Notification Hub: " + ex.Message);
        }
    }
}
```

此方法创建简单的通知消息模板为 JSON 和寄存器接收来自使用 Firebase 注册令牌的通知中心模板通知。这可确

保从 Azure 通知中心发送任何通知都将针对注册令牌所表示的设备。

显示一条推送通知的内容

显示一条推送通知的内容通过派生类从 `FirebaseMessagingService` 类。此类包括可重写 `OnMessageReceived` 方法，调用应用程序接收来自 FCM 的通知时，提供在前台运行该应用程序。在示例应用程序 `FirebaseNotificationService` 类派生自 `FirebaseMessagingService` 类，并在下面的代码示例所示：

```
[Service]
[IntentFilter(new[] { "com.google.firebase.MESSAGING_EVENT" })]
public class FirebaseNotificationService : FirebaseMessagingService
{
    const string TAG = "FirebaseNotificationService";

    public override void OnMessageReceived(RemoteMessage message)
    {
        Log.Debug(TAG, "From: " + message.From);

        // Pull message body out of the template
        var messageBody = message.Data["message"];
        if (string.IsNullOrWhiteSpace(messageBody))
            return;

        Log.Debug(TAG, "Notification message body: " + messageBody);
        SendNotification(messageBody);
    }

    void SendNotification(string messageBody)
    {
        var intent = new Intent(this, typeof(MainActivity));
        intent.AddFlags(ActivityFlags.ClearTop);
        var pendingIntent = PendingIntent.GetActivity(this, 0, intent, PendingIntentFlags.OneShot);

        var notificationBuilder = new NotificationCompat.Builder(this)
            .SetSmallIcon(Resource.Drawable.ic_stat_ic_notification)
            .SetContentTitle("New Todo Item")
            .SetContentText(messageBody)
            .SetContentIntent(pendingIntent)
            .SetSound(RingtoneManager.GetDefaultUri(RingtoneType.Notification))
            .SetAutoCancel(true);

        var notificationManager = NotificationManager.FromContext(this);
        notificationManager.Notify(0, notificationBuilder.Build());
    }
}
```

应用程序收到通知后从 FCM，`OnMessageReceived` 方法提取消息内容，并调用 `SendNotification` 方法。此方法将消息内容转换为在应用程序运行时，与通知显示在通知区域中启动的本地通知。

处理通知意向

当用户点击通知时，相关的通知消息的任何数据都可以在 `Intent` 其他功能。此数据可以提取替换为以下代码：

```
if (Intent.Extras != null)
{
  foreach (var key in Intent.Extras.KeySet())
  {
    var value = Intent.Extras.GetString(key);
    Log.Debug(TAG, "Key: {0} Value: {1}", key, value);
  }
}
```

应用程序的启动器 `Intent` 当用户点击其通知消息，因此此代码将记录中的任何随附的数据时触发 `Intent` 到输出窗口。

通用 Windows 平台

在通用 Windows 平台 (UWP) 之前应用程序可收到推送通知，它必须注册与 Windows 通知服务 (WNS)，它将返回通知通道。由调用注册 `InitNotificationsAsync` 中的方法 `App` 类：

```
private async Task InitNotificationsAsync()
{
    var channel = await PushNotificationChannelManager
          .CreatePushNotificationChannelForApplicationAsync();

    const string templateBodyWNS =
        "<toast><visual><binding template=\"ToastText01\"><text id=\"1\">$(messageParam)</text></binding>
</visual></toast>";

    JObject headers = new JObject();
    headers["X-WNS-Type"] = "wns/toast";

    JObject templates = new JObject();
    templates["genericMessage"] = new JObject
    {
        {"body", templateBodyWNS},
        {"headers", headers} // Needed for WNS.
    };

    await TodoItemManager.DefaultManager.CurrentClient.GetPush()
          .RegisterAsync(channel.Uri, templates);
}
```

此方法获取推送通知通道、创建通知消息模板为 JSON，并注册设备以接收从通知中心发送模板通知。

`InitNotificationsAsync` 方法调用从 `OnLaunched` 中重写 `App` 类：

```
protected override async void OnLaunched(LaunchActivatedEventArgs e)
{
    ...
    await InitNotificationsAsync();
}
```

这可确保推送通知注册是在创建或刷新每次启动应用程序，因此确保 WNS 推送通道始终处于活动状态。

收到推送通知时它将自动显示为*toast* – 一个无模式窗口，其中包含的消息。

总结

这篇文章演示了如何使用 Azure 通知中心将推送通知从 Azure Mobile Apps 实例发送到 Xamarin.Forms 应用程序。Azure 通知中心提供可缩放的推送基础结构用于发送移动推送通知从任意后端向任何移动平台，同时无后端不必与不同的平台通知系统通信的复杂性。

相关链接

- 使用 Azure 移动应用
- Azure 通知中心
- 向 Xamarin.Forms 应用添加推送通知
- 在 iOS 中的推送通知
- Firebase 云消息传送
- TodoAzurePush（示例）
- Azure 的移动客户端 SDK

在云中存储文件

2018/6/9 • • Edit Online

Azure 存储是一种可扩展的云存储解决方案，可以用于存储非结构化和结构化数据。

在 Azure 存储中存储文件

本文演示如何使用 Xamarin.Forms 将文本和二进制数据存储在 Azure 存储空间，以及如何访问数据。

存储和访问 Azure 存储空间中的数据

2018/6/22 • • Edit Online

Azure 存储是一种可扩展的云存储解决方案，可以用于存储非结构化和结构化数据。本文演示如何使用 Xamarin.Forms 将文本和二进制数据存储在 Azure 存储空间，以及如何访问数据。

概述

Azure 存储空间提供了四个存储服务：

- Blob 存储。Blob 可以是文本或二进制数据，如备份、虚拟机、媒体文件或文档。
- 表存储是 NoSQL 键-属性存储。
- 队列存储是工作流处理和云服务之间的通信的消息传递服务。
- 文件存储提供了使用 SMB 协议的共享的存储。

有两种类型的存储帐户：

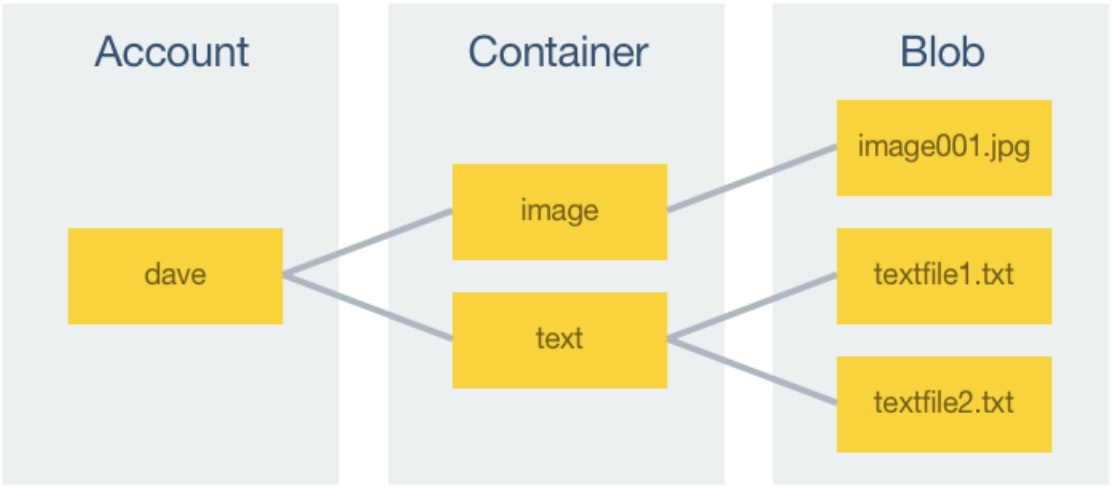
- 通过单个帐户情况下，通用的存储帐户提供对 Azure 存储服务的访问。
- Blob 存储帐户是专门的存储帐户用于存储 blob。你只需以存储 blob 数据，则建议使用此帐户类型。

这篇文章，并随附的示例应用程序，演示到 blob 存储，并将其下载上载图像和文本的文件。此外，它还演示从 blob 存储检索的文件的列表和删除文件。

有关 Azure 存储空间的详细信息，请参阅存储空间简介。

Blob 存储空间简介

Blob 存储包含三个组件，如下图所示：



到 Azure 存储空间的所有访问都都通过存储帐户。存储帐户可以包含无限的数量的容器，并且一个容器可以存储无限的数量的 blob，直至达到存储帐户容量限制。

Blob 是任何类型和大小的文件。Azure 存储空间支持三种不同的 blob 类型：

- 块 blob 进行了优化来流化和存储云对象，并且是一个不错的选择用于存储备份、媒体文件和文档等。块 blob 可以是最多为 195 Gb 大小。
- 追加 blob 类似于块 blob，但针对进行了优化追加操作，例如日志记录。追加 blob 可以是最多为 195 Gb 大小。

- 页 blob 为频繁的读/写操作进行了优化，并且通常用于存储虚拟机和其磁盘。页 blob 可以达 1 Tb 的大小。

> **NOTE**
> 请注意，blob 存储帐户支持块追加 blob，但不是页 blob。

Blob 上载到 Azure 存储空间，并从 Azure 存储空间下载的字节流的形式。因此，文件必须转换为的上载，并为其原始的表示形式下载后转换的回之前的字节流。

Azure 存储中存储每个对象具有唯一的 URL 地址。存储帐户名称构成该地址和子域和域名名称窗体组合子域*终结点*的存储帐户。例如，如果你的存储帐户名为*mystorageaccount*，存储帐户的默认 blob 终结点 `https://mystorageaccount.blob.core.windows.net`。

将存储帐户中的对象的位置附加到终结点生成用于访问存储帐户中的某个对象的 URL。例如，blob 地址将具有格式 `https://mystorageaccount.blob.core.windows.net/mycontainer/myblob`。

安装

将 Azure 存储帐户集成到 Xamarin.Forms 应用程序的过程如下所示：

1. 创建存储帐户。有关详细信息，请参阅创建存储帐户。
2. 添加Azure Storage Client Library到 Xamarin.Forms 应用程序。
3. 配置存储连接字符串。有关详细信息，请参阅连接到 Azure 存储。
4. 添加 `using` 指令 `Microsoft.WindowsAzure.Storage` 和 `Microsoft.WindowsAzure.Storage.Blob` 命名空间添加到将访问 Azure 存储空间的类。

> **NOTE**
> 尽管此示例使用的共享访问项目，Azure 存储客户端库现在也支持从可移植类库 (PCL) 项目已使用。

连接到 Azure 存储

针对存储帐户资源发出的每个请求必须进行身份验证。尽管 blob 可以配置为支持匿名身份验证，则有应用程序可用于使用存储帐户进行身份验证的两种主要方法：

- 共享的密钥。此方法使用 Azure 存储帐户名称和帐户密钥访问存储服务。存储帐户分配在创建可用于共享密钥身份验证的两个私有密钥。
- 共享的访问签名。这是一个可以附加到 URL 的实现对存储资源的委托的访问令牌，，权限使用它，为指定的是有效的时间段。

可以指定连接字符串，包括从应用程序访问 Azure 存储资源所需的身份验证信息。此外，还可以配置连接字符串，以从 Visual Studio 连接到 Azure 存储模拟器。

> **NOTE**
> Azure 存储连接字符串中支持 HTTP 和 HTTPS。但是，建议使用 HTTPS。

连接到 Azure 存储模拟器

Azure 存储模拟器提供了一个模拟 Azure blob、队列和表服务以进行开发的本地环境。

应使用以下连接字符串连接到 Azure 存储模拟器：

```
UseDevelopmentStorage=true
```

有关 Azure 存储模拟器的详细信息，请参阅使用 Azure 存储模拟器进行开发和测试。

连接到 Azure 存储空间中使用共享的密钥

以下连接字符串的格式应该用于连接到 Azure 存储具有共享密钥：

```
DefaultEndpointsProtocol=[http|https];AccountName=myAccountName;AccountKey=myAccountKey
```

`myAccountName` 应替换为你的存储帐户的名称和 `myAccountKey` 应使用两个帐户访问密钥之一进行替换。

> **NOTE**
> 何时使用共享密钥身份验证，你的帐户名称和帐户密钥将分发到每个用户都使用你的应用程序，将提供完全读/写访问权限的存储帐户中。因此，仅，用于测试目的使用共享密钥身份验证，并且永远不会将密钥分发给其他用户。

连接到 Azure 存储空间使用共享访问签名

以下连接字符串的格式应该用于连接到使用 SAS 的 Azure 存储：

```
BlobEndpoint=myBlobEndpoint;SharedAccessSignature=mySharedAccessSignature
```

`myBlobEndpoint` 应替换为你的 blob 终结点的 URL 和 `mySharedAccessSignature` 应替换为你的 SAS。SAS 提供协议、服务终结点和用于访问资源的凭据。

> **NOTE**
> SAS 身份验证被建议用于生产应用程序。但是，在生产应用程序应从后端服务按需，而不是与应用程序捆绑在正在检索 SAS。

有关共享访问签名的详细信息，请参阅使用共享访问签名 (SAS)。

创建容器

`GetContainer` 方法用于检索对一个命名的容器，它随后可从容器中检索 blob 或将 blob 添加到容器的引用。下面的代码示例演示 `GetContainer` 方法：

```
static CloudBlobContainer GetContainer(ContainerType containerType)
{
  var account = CloudStorageAccount.Parse(Constants.StorageConnection);
  var client = account.CreateCloudBlobClient();
  return client.GetContainerReference(containerType.ToString().ToLower());
}
```

`CloudStorageAccount.Parse` 方法分析连接字符串并返回 `CloudStorageAccount` 表示存储帐户的实例。A `CloudBlobClient` 实例，用于检索容器和 blob，然后由 `CreateCloudBlobClient` 方法。 `GetContainerReference` 方法检索为指定的容器 `CloudBlobContainer` 实例，它将返回到调用方法之前。在此示例中，容器名称是 `ContainerType` 枚举值，转换为小写的字符串。

> **NOTE**
> 容器名称必须小写，并且必须以字母或数字开头。此外，它们只能包含字母、数字和短划线字符，并且必须介于 3 到 63 个字符之间。

`GetContainer` 方法调用，如下所示：

```
var container = GetContainer(containerType);
```

`CloudBlobContainer` 然后可以使用实例创建一个容器，如果它尚不存在：

```
await container.CreateIfNotExistsAsync();
```

默认情况下，新创建的容器是私有的。这意味着，必须指定存储访问密钥容器中检索 blob。有关如何使容器公开中的 blob 的信息，请参阅创建容器。

将数据上载到容器

`UploadFileAsync` 方法可用于上载流的字节数据到 blob 存储，并在下面的代码示例所示：

```
public static async Task<string> UploadFileAsync(ContainerType containerType, Stream stream)
{
  var container = GetContainer(containerType);
  await container.CreateIfNotExistsAsync();

  var name = Guid.NewGuid().ToString();
  var fileBlob = container.GetBlockBlobReference(name);
  await fileBlob.UploadFromStreamAsync(stream);

  return name;
}
```

在检索之后容器引用，该方法创建的容器，如果它尚不存在。一个新 `Guid` 然后创建以充当唯一 blob 名称，并为检索 blob 块引用 `CloudBlockBlob` 实例。然后，数据的流上载到 blob 使用 `UploadFromStreamAsync` 方法，将创建 blob，如果它尚不存在，或如果它存在覆盖它。

将文件上载到 blob 存储使用此方法之前，它必须首先必须转换为字节流。在下面的代码示例说明了这一点：

```
var byteData = Encoding.UTF8.GetBytes(text);
uploadedFilename = await AzureStorage.UploadFileAsync(ContainerType.Text, new MemoryStream(byteData));
```

`text` 数据转换为字节数组，然后传递给流的形式包装 `UploadFileAsync` 方法。

从容器下载数据

`GetFileAsync` 方法可用于从 Azure 存储空间下载 blob 数据，并在下面的代码示例所示：

```
public static async Task<byte[]> GetFileAsync(ContainerType containerType, string name)
{
  var container = GetContainer(containerType);

  var blob = container.GetBlobReference(name);
  if (await blob.ExistsAsync())
  {
    await blob.FetchAttributesAsync();
    byte[] blobBytes = new byte[blob.Properties.Length];

    await blob.DownloadToByteArrayAsync(blobBytes, 0);
    return blobBytes;
  }
  return null;
}
```

在检索之后容器引用，该方法将检索存储的数据的 blob 引用。如果存在 blob，通过检索其属性 `FetchAttributesAsync` 方法。将创建的正确大小的字节数组，并获取返回到调用方法的字节数组形式下载 blob。

在下载的 blob 字节数据之后，它必须转换为其原始的表示形式。在下面的代码示例说明了这一点：

```
var byteData = await AzureStorage.GetFileAsync(ContainerType.Text, uploadedFilename);
string text = Encoding.UTF8.GetString(byteData);
```

从 Azure 存储空间中检索的字节数组 `GetFileAsync` 方法之前它将转换回为 UTF8，编码的字符串。

列出容器中的数据

`GetFilesListAsync` 方法可用于检索的存储容器中的 blob 列表，并在下面的代码示例所示：

```
public static async Task<IList<string>> GetFilesListAsync(ContainerType containerType)
{
  var container = GetContainer(containerType);

  var allBlobsList = new List<string>();
  BlobContinuationToken token = null;

  do
  {
    var result = await container.ListBlobsSegmentedAsync(token);
    if (result.Results.Count() > 0)
    {
      var blobs = result.Results.Cast<CloudBlockBlob>().Select(b => b.Name);
      allBlobsList.AddRange(blobs);
    }
    token = result.ContinuationToken;
  } while (token != null);

  return allBlobsList;
}
```

在检索之后容器引用，该方法使用容器的 `ListBlobsSegmentedAsync` 方法来检索对容器内 blob 的引用。返回的结果 `ListBlobsSegmentedAsync` 枚举方法时 `BlobContinuationToken` 实例不是 `null`。每个 blob 被强制转换从返回 `IListBlobItem` 到 `CloudBlockBlob` 顺序访问中 `Name` 的 blob 之前它是值，的属性添加到 `allBlobsList` 集合。一次 `BlobContinuationToken` 实例是 `null`，已返回最后一个的 blob 名称，并执行退出循环。

从容器中删除数据

`DeleteFileAsync` 方法可用于在容器中，删除 blob，并在下面的代码示例所示：

```
public static async Task<bool> DeleteFileAsync(ContainerType containerType, string name)
{
  var container = GetContainer(containerType);
  var blob = container.GetBlobReference(name);
  return await blob.DeleteIfExistsAsync();
}
```

在检索之后容器引用，该方法检索指定的 blob 的 blob 引用。然后与删除的 blob `DeleteIfExistsAsync` 方法。

总结

本文演示如何使用 Xamarin.Forms 将文本和二进制数据存储在 Azure 存储空间，以及如何访问数据。Azure 存储是一种可扩展的云存储解决方案，可以用于存储非结构化和结构化数据。

相关链接

- Azure 存储（示例）
- 存储空间简介
- 如何通过 Xamarin 使用 Blob 存储
- 使用共享的访问签名 (SAS)
- Windows Azure 存储空间

在云中搜索数据

2018/6/22 • • Edit Online

Azure 搜索是云服务，提供了索引和查询上载的数据的功能。这将删除基础结构要求和传统上与应用程序中实现搜索功能关联的搜索算法复杂性。

使用 Azure 搜索中搜索数据

本文演示如何使用 Microsoft Azure 搜索库将 Azure Search 集成到 Xamarin.Forms 应用程序。

使用 Azure 搜索中搜索数据

2018/6/9 • • Edit Online

Azure 搜索是云服务，提供了索引和查询上载的数据的功能。这将删除基础结构要求和传统上与应用程序中实现搜索功能关联的搜索算法复杂性。本文演示如何使用 Microsoft Azure 搜索库将 Azure Search 集成到 Xamarin.Forms 应用程序。

概述

数据存储在 Azure 搜索为索引和文档。*索引*是 Azure 搜索服务中，可以搜索的数据的存储和从概念上讲类似于数据库表。A*文档*是在索引中，可搜索数据的单个单元，它从概念上讲类似于数据库行。当将文档上载和提交对 Azure 搜索的搜索查询，对特定索引中搜索服务发出请求。

对 Azure 搜索所做的每个请求必须包含服务和 API 密钥名称。有两种类型的 API 密钥：

- *管理密钥*授予对所有操作的完全权限。这包括管理服务、创建和删除索引和数据源。
- *查询密钥*授予对索引和文档，只读访问权限，应使用的应用程序发出搜索请求。

最常见的请求对 Azure 搜索是执行查询。有两种类型的可以提交的查询：

- A*搜索*查询将搜索在索引中的所有可搜索字段中的一个或多个项。搜索查询是使用简化的语法或 Lucene 查询语法生成的。有关详细信息，请参阅Azure 搜索中的简单查询语法，和Lucene Azure 搜索中的查询语法。
- A*筛选器*查询对索引中的所有可筛选字段计算布尔表达式。使用 OData 筛选器语言的子集生成筛选器查询。有关详细信息，请参阅Azure 搜索的 OData 表达式语法。

单独或一起，可以使用搜索查询和筛选器查询。筛选器查询一起使用时，到整个索引中，首先应用，然后搜索查询执行筛选器查询的结果。

Azure 搜索还支持基于搜索输入检索建议。有关详细信息，请参阅建议查询。

安装

将 Azure Search 集成到 Xamarin.Forms 应用程序的过程如下所示：

1. 创建 Azure 搜索服务。有关详细信息，请参阅创建 Azure 搜索服务使用 Azure 门户。
2. 作为目标框架的 Silverlight 移除 Xamarin.Forms 解决方案可移植类库 (PCL)。这可以通过将 PCL 配置文件更改为支持跨平台开发，但不支持 Silverlight，如配置文件 151 或 92 任何配置文件来实现。
3. 添加Microsoft Azure 搜索库到 PCL 项目中，Xamarin.Forms 解决方案的 NuGet 包。

执行这些步骤后，Microsoft 搜索库 API 可用来管理搜索索引和数据源、上载和管理文档和执行查询。

创建 Azure 搜索索引

索引架构必须定义映射到要搜索的数据的结构。这可以完成在 Azure 门户中，或以编程方式使用 `SearchServiceClient` 类。此类管理连接到 Azure 搜索中，并可以用于创建索引。下面的代码示例演示如何创建此类的实例：

```
var searchClient =
  new SearchServiceClient(Constants.SearchServiceName, new SearchCredentials(Constants.AdminApiKey));
```

`SearchServiceClient` 构造函数重载采用一个搜索服务名称和一个 `SearchCredentials` 对象作为自变量，

`SearchCredentials` 对象包装 *管理密钥* Azure 搜索服务。 *管理密钥* 创建索引所需。

> **NOTE**
> 单个 `SearchServiceClient` 实例应使用应用程序中，以避免打开太多连接到 Azure Search。

通过定义索引 `Index` 对象，如下面的代码示例中所示：

```
static void CreateSearchIndex()
{
  var index = new Index()
  {
    Name = Constants.Index,
    Fields = new[]
    {
      new Field("id", DataType.String) { IsKey = true, IsRetrievable = true },
      new Field("name", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSortable = true,
IsSearchable = true },
      new Field("location", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSortable = true,
IsSearchable = true },
      new Field("details", DataType.String) { IsRetrievable = true, IsFilterable = true, IsSearchable = true
},
      new Field("imageUrl", DataType.String) { IsRetrievable = true }
    },
    Suggesters = new[]
    {
      new Suggester("nameSuggester", SuggesterSearchMode.AnalyzingInfixMatching, new[] { "name" })
    }
  };

  searchClient.Indexes.Create(index);
}
```

`Index.Name` 属性应设置为索引，名称和 `Index.Fields` 属性应设置为的数组 `Field` 对象。每个 `Field` 实例指定名称、类型和任何属性，指定如何使用此字段。这些属性包括：

- `IsKey` – 指示字段是否索引的键。在索引中，类型的一个字段 `DataType.String`，必须指定为键字段。
- `IsFacetable` – 指示是否可以对此字段执行分面导航。默认值为 `false`。
- `IsFilterable` – 表示是否可以在筛选器查询中使用该字段。默认值为 `false`。
- `IsRetrievable` – 指示是否可以在搜索结果中检索字段。默认值为 `true`。
- `IsSearchable` – 指示是否在全文搜索中包含的字段。默认值为 `false`。
- `IsSortable` – 表示是否可以在中使用字段 `OrderBy` 表达式。默认值为 `false`。

> **NOTE**
> 在部署后更改索引涉及重新生成并重新加载数据。

`Index` 对象可以选择指定 `Suggesters` 属性，用于定义索引中要用于支持自动完成或搜索建议查询的字段。 `Suggesters` 属性应设置为的数组 `Suggester` 定义用于生成建议结果的搜索的字段的对象。

在创建后 `Index` 对象，通过调用创建索引 `Indexes.Create` 上 `SearchServiceClient` 实例。

> **NOTE**
> 当从应用程序创建索引必须响应保留时，则使用 `Indexes.CreateAsync` 方法。

有关详细信息，请参阅创建 Azure Search 索引使用.NET SDK。

删除 Azure 搜索索引

可以通过调用删除索引 `Indexes.Delete` 上 `SearchServiceClient` 实例：

```
searchClient.Indexes.Delete(Constants.Index);
```

将数据上载到 Azure 搜索索引

定义索引之后，可以将数据上载到它使用两种模式之一：

- **拉取模型**– 数据定期引入从 Azure Cosmos DB、Azure SQL 数据库、Azure Blob 存储或 SQL Server 托管 Azure 虚拟机中。
- **推送模型**– 数据以编程方式发送到索引。这是采用此文章中的模型。

A `SearchIndexClient` 必须创建实例，以将数据导入索引。这可以通过调用来实现 `SearchServiceClient.Indexes.GetClient` 方法，如下面的代码示例中所示：

```
static void UploadDataToSearchIndex()
{
  var indexClient = searchClient.Indexes.GetClient(Constants.Index);

  var monkeyList = MonkeyData.Monkeys.Select(m => new
  {
    id = Guid.NewGuid().ToString(),
    name = m.Name,
    location = m.Location,
    details = m.Details,
    imageUrl = m.ImageUrl
  });

  var batch = IndexBatch.New(monkeyList.Select(IndexAction.Upload));
  try
  {
    indexClient.Documents.Index(batch);
  }
  catch (IndexBatchException ex)
  {
    // Sometimes when the Search service is under load, indexing will fail for some
    // documents in the batch. Compensating actions like delaying and retrying should be taken.
    // Here, the failed document keys are logged.
    Console.WriteLine("Failed to index some documents: {0}",
      string.Join(", ", ex.IndexingResults.Where(r => !r.Succeeded).Select(r => r.Key)));
  }
}
```

要导入到索引数据打包为 `IndexBatch` 对象，封装的集合 `IndexAction` 对象。每个 `IndexAction` 实例包含一个文档，以及指示要对文档执行的操作的 Azure 搜索的属性。在上面的代码示例 `IndexAction.Upload` 指定，如果它是新的、要插入到索引文档中的哪些结果或如果它已存在替换操作。 `IndexBatch` 对象然后通过调用发送到索引 `Documents.Index` 方法 `SearchIndexClient` 对象。有关其他索引操作的信息，请参阅决定要使用的索引操作。

> **NOTE**
> 仅 1000 个文档可以包含在单个索引请求。

请注意，在上面的代码示例 `monkeyList` 作为匿名对象的集合中创建集合 `Monkey` 对象。这将创建数据 `id` 字段，并将

其解析 Pascal 大小写的映射 `Monkey` camel 大小写的属性名称搜索索引字段名称。或者，此映射还可以通过添加 `[SerializePropertyNamesAsCamelCase]` 属性设为 `Monkey` 类。

有关详细信息，请参阅将数据上载到 Azure 搜索中使用.NET SDK。

查询 Azure 搜索索引

A `SearchIndexClient` 必须创建实例查询的索引。当应用程序执行查询时，则最好遵循最低特权原则，并创建 `SearchIndexClient` 直接传递*查询密钥*作为自变量。这可确保用户具有对索引和文档的只读访问权限。在下面的代码示例演示了这种方法：

```
SearchIndexClient indexClient =
  new SearchIndexClient(Constants.SearchServiceName, Constants.Index, new
SearchCredentials(Constants.QueryApiKey));
```

`SearchIndexClient` 构造函数重载采用搜索服务名称、索引名称和一个 `SearchCredentials` 对象作为自变量，`SearchCredentials` 对象包装*查询密钥*Azure 搜索服务。

搜索查询

可以通过调用查询索引 `Documents.SearchAsync` 方法 `SearchIndexClient` 实例，如下面的代码示例中所示：

```
async Task AzureSearch(string text)
{
  Monkeys.Clear();

  var searchResults = await indexClient.Documents.SearchAsync<Monkey>(text);
  foreach (SearchResult<Monkey> result in searchResults.Results)
  {
    Monkeys.Add(new Monkey
    {
      Name = result.Document.Name,
      Location = result.Document.Location,
      Details = result.Document.Details,
      ImageUrl = result.Document.ImageUrl
    });
  }
}
```

`SearchAsync` 方法采用的搜索文本参数和可选 `SearchParameters` 可以用于进一步优化查询的对象。可以通过将设置指定筛选器查询时，指定将搜索查询作为搜索文本自变量，`Filter` 属性 `SearchParameters` 自变量。下面的代码示例演示两种查询类型：

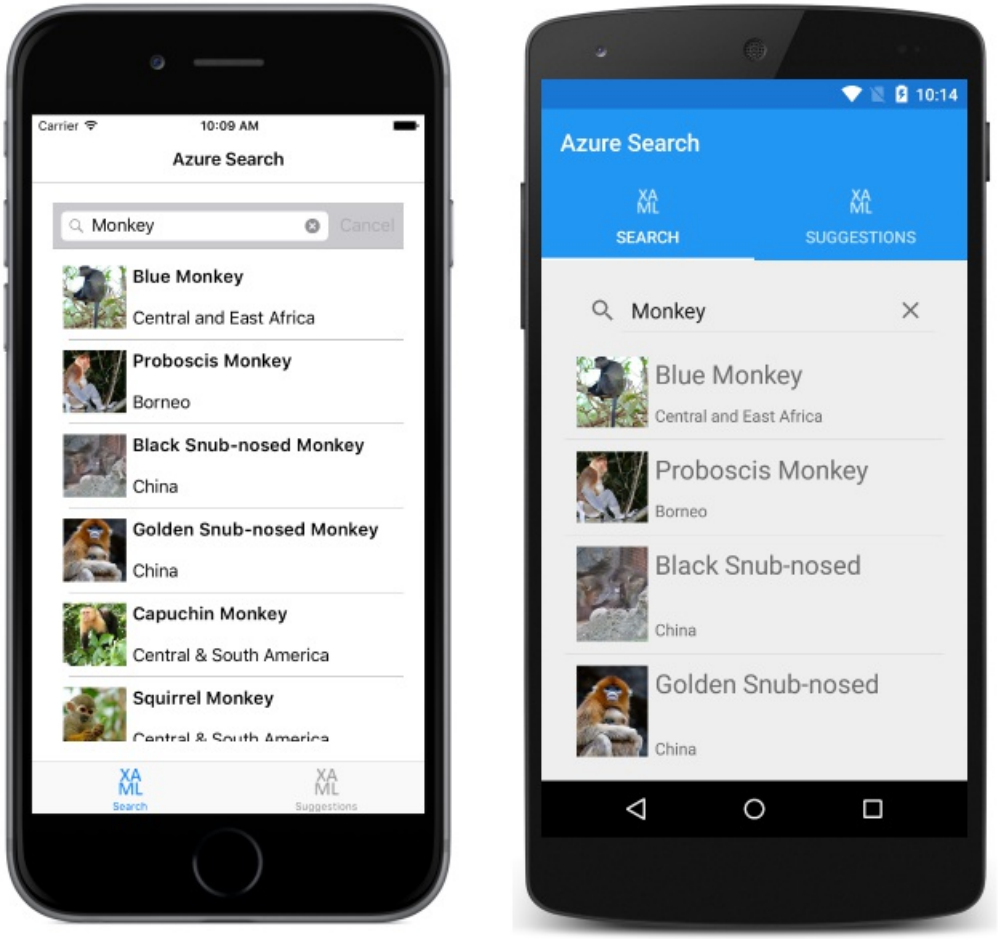
```
var parameters = new SearchParameters
{
  Filter = "location ne 'China' and location ne 'Vietnam'"
};
var searchResults = await indexClient.Documents.SearchAsync<Monkey>(text, parameters);
```

此筛选器查询应用于整个索引，从结果中移除文档其中 `location` 字段不是等于中国且不等于越南。在筛选之后，将搜索查询执行筛选器查询的结果。

> **NOTE**
> 若要筛选而无需搜索，请将传递 `*` 作为搜索文本参数。

`SearchAsync` 方法返回 `DocumentSearchResult` 包含查询结果的对象。此对象枚举，与每个 `Document` 对象创建为

`Monkey` 对象，并添加到 `Monkeys``ObservableCollection` 进行显示。以下屏幕快照显示搜索查询返回的结果从 Azure 搜索：



有关搜索和筛选的详细信息，请参阅查询你使用.NET SDK 的 Azure 搜索索引。

建议查询

Azure 搜索允许请求的建议基于搜索查询，通过调用 `Documents.SuggestAsync` 方法 `SearchIndexClient` 实例。在下面的代码示例说明了这一点：

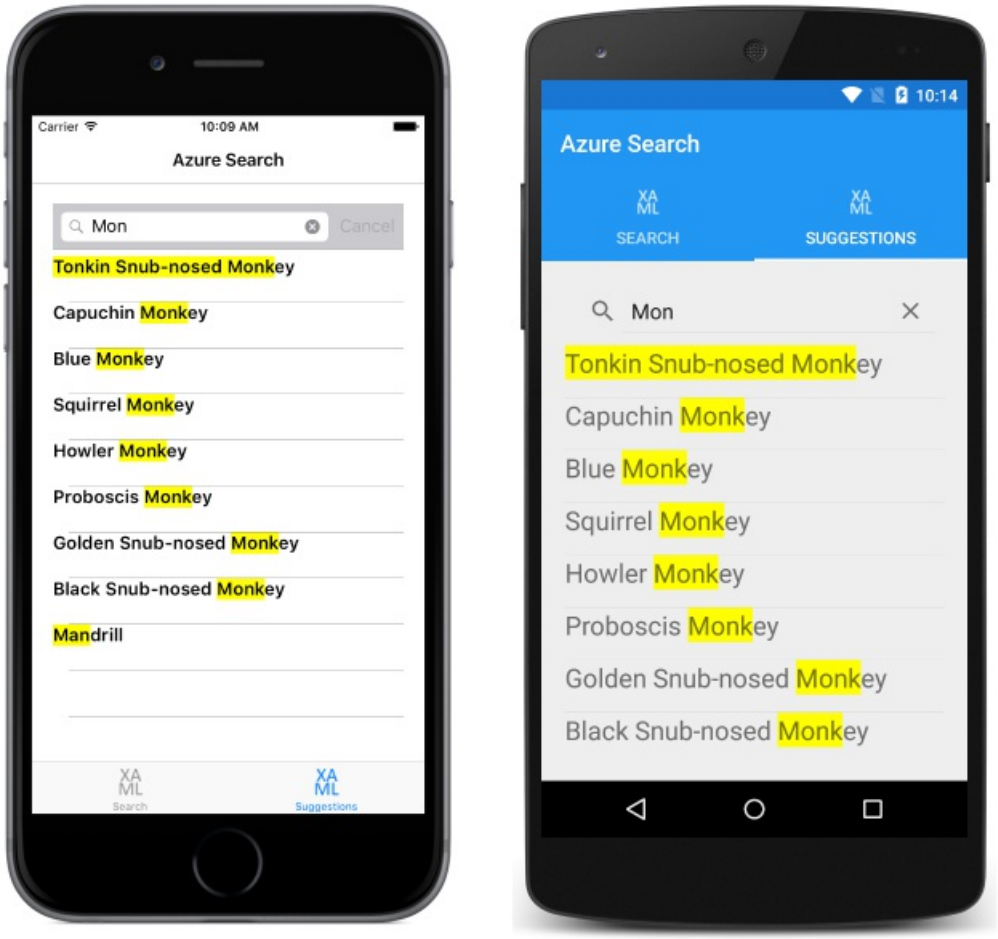
```
async Task AzureSuggestions(string text)
{
  Suggestions.Clear();

  var parameters = new SuggestParameters()
  {
    UseFuzzyMatching = true,
    HighlightPreTag = "[",
    HighlightPostTag = "]",
    MinimumCoverage = 100,
    Top = 10
  };

  var suggestionResults =
    await indexClient.Documents.SuggestAsync<Monkey>(text, "nameSuggester", parameters);

  foreach (var result in suggestionResults.Results)
  {
    Suggestions.Add(new Monkey
    {
      Name = result.Text,
      Location = result.Document.Location,
      Details = result.Document.Details,
      ImageUrl = result.Document.ImageUrl
    });
  }
}
```

`SuggestAsync` 方法采用一个搜索文本自变量，若要使用的建议器名称（即定义索引中），和一个可选 `SuggestParameters` 可以用于进一步优化查询的对象。 `SuggestParameters` 实例将设置以下属性：

- `UseFuzzyMatching` – 当设置为 `true`，Azure 搜索会查找建议，即使搜索文本中没有替代或缺少字符。
- `HighlightPreTag` – 建议命中前面预置的标记。
- `HighlightPostTag` – 追加到建议命中数的标记。
- `MinimumCoverage` – 表示报告为成功，则必须通过建议查询查询要涵盖的索引的百分比。默认值为 80。
- `Top` – 若要检索的建议的数量。它必须是 1 到 100，默认值为 5 之间的整数。

总体效果是命中突出显示，并且结果将包括包括相似的拼写搜索词的文档，将会从索引前的 10 个结果返回与。

`SuggestAsync` 方法返回 `DocumentSuggestResult` 包含查询结果的对象。此对象枚举，与每个 `Document` 对象创建为 `Monkey` 对象，并添加到 `Monkeys``ObservableCollection` 进行显示。以下屏幕截图显示了从 Azure 搜索返回的建议结果：



请注意，在示例应用程序，`SuggestAsync`用户完成输入的搜索词时，才会调用方法。但是，它还可支持自动完成搜索查询，通过在每个 keypress 上执行。

总结

这篇文章演示了如何使用 Microsoft Azure 搜索库将 Azure Search 集成到 Xamarin.Forms 应用程序。Azure 搜索是云服务，提供了索引和查询上载的数据的功能。这将删除基础结构要求和传统上与应用程序中实现搜索功能关联的搜索算法复杂性。

相关链接

- Azure 搜索（示例）
- Azure 搜索文档
- Microsoft Azure 搜索库

无服务器计算使用 Xamarin.Forms

2018/10/26 • • Edit Online

借助强大的后端功能，而无需配置和管理服务器的复杂性构建应用。

Azure Functions

开始构建第一个 Azure 函数交互使用 Xamarin.Forms。

开始使用 Azure Functions

2018/11/14 • • Edit Online

开始构建第一个 Azure 函数交互使用 Xamarin.Forms。

- Visual Studio
- Visual Studio for Mac

分步说明

除了此视频中，可以按照这些说明生成第一个函数使用 Visual Studio 2017。

相关链接

- Azure Functions 文档
- 使用 Xamarin.Forms 客户端（示例）和实施一个简单的 Azure 函数

在文档数据库中存储数据

2018/6/22 • • Edit Online

Azure Cosmos DB 文档数据库是提供对 JSON 文档，提供快速、高度可用、可缩放数据库服务需要无缝缩放和全局复制的应用程序的较低的延迟访问的 NoSQL 数据库。

使用 Azure Cosmos DB 文档数据库

此文章介绍了如何使用 Azure Cosmos DB 标准.NET 客户端库来将 Azure Cosmos DB 文档数据库集成到 Xamarin.Forms 应用程序。

使用 Azure Cosmos DB 文档数据库对用户进行身份验证

此文章介绍了如何组合使用分区集合，使用访问控制，以便用户只能访问自己在 Xamarin.Forms 应用程序中的文档。

使用 Azure Cosmos DB 文档数据库

2018/11/14 • • Edit Online

Azure Cosmos DB 文档数据库是提供低延迟访问产品/服务面向需要无缝缩放和全局复制的应用程序的快速、高度可用、可缩放数据库服务的 JSON 文档的 NoSQL 数据库。此文章介绍了如何使用 Azure Cosmos DB.NET Standard 客户端库将 Azure Cosmos DB 文档数据库集成到 Xamarin.Forms 应用程序。

Microsoft Azure Cosmos DB，也可由Xamarin 学院课程

可以使用 Azure 订阅预配 Azure Cosmos DB 文档数据库帐户。每个数据库帐户可以有零个或多个数据库。Azure Cosmos DB 中的文档数据库是适用于文档集合和用户的逻辑容器。

Azure Cosmos DB 文档数据库可能包含零个或多个文档集合。每个文档集合可以具有不同的性能级别，允许更大的吞吐量为经常访问的集合，指定和更少不常访问的集合的吞吐量。

每个文档集合包含的零个或多个 JSON 文档。集合中的文档是无架构的因此不需要共享相同的结构或字段。文档添加到文档集合，Cosmos DB 会自动编制索引以及它们会变得可供查询。

出于开发目的，文档数据库也可以使用通过仿真程序。使用模拟器，应用程序可以进行开发和测试本地，而无需创建 Azure 订阅，也不会产生任何费用。有关模拟器的详细信息，请参阅使用 Azure Cosmos DB 模拟器进行本地开发。

本文中，以及随附的示例应用程序，演示其中这些任务存储在 Azure Cosmos DB 文档数据库中的待办事项列表应用程序。有关示例应用程序的详细信息，请参阅了解示例。

有关 Azure Cosmos DB 的详细信息，请参阅Azure Cosmos DB 文档。

安装

将 Azure Cosmos DB 文档数据库集成到 Xamarin.Forms 应用程序的过程如下所示：

1. 创建 Cosmos DB 帐户。有关详细信息，请参阅创建 Azure Cosmos DB 帐户。
2. 添加Azure Cosmos DB.NET Standard 客户端库到 Xamarin.Forms 解决方案中的平台项目的 NuGet 包。
3. 添加 `using` 指令 `Microsoft.Azure.Documents`， `Microsoft.Azure.Documents.Client`，和 `Microsoft.Azure.Documents.Linq` 到将访问 Cosmos DB 帐户的类的命名空间。

执行这些步骤后，可以使用 Azure Cosmos DB.NET Standard 客户端库进行配置和执行针对文档数据库的请求。

> **NOTE**
> Azure Cosmos DB.NET Standard 客户端库只能安装到平台项目中，并不到可移植类库 (PCL) 项目。因此，示例应用程序是共享访问项目 (SAP) 以避免代码重复。但是，`DependencyService` 类可用于在 PCL 项目中调用特定于平台的项目中包含的 Azure Cosmos DB.NET Standard 客户端库代码。

使用 Azure Cosmos DB 帐户

`DocumentClient` 类型封装终结点、凭据和连接策略用来访问 Azure Cosmos DB 帐户，并用于配置和执行针对帐户的请求。下面的代码示例演示如何创建此类的实例：

```
DocumentClient client = new DocumentClient(new Uri(Constants.EndpointUri), Constants.PrimaryKey);
```

Cosmos DB Uri 和主密钥必须提供给 `DocumentClient` 构造函数。可以从 Azure 门户获取这些。有关详细信息，请参阅连接到 Azure Cosmos DB 帐户。

创建数据库

文档数据库是适用于文档集合和用户的逻辑容器，并且可以创建在 Azure 门户中，或以编程方式使用 `DocumentClient.CreateDatabaseIfNotExistsAsync` 方法：

```
public async Task CreateDatabase(string databaseName)
{
  ...
  await client.CreateDatabaseIfNotExistsAsync(new Database
  {
    Id = databaseName
  });
  ...
}
```

`CreateDatabaseIfNotExistsAsync` 方法指定 `Database` 对象作为参数，与 `Database` 对象，指定数据库名称作为其 `Id` 属性。`CreateDatabaseIfNotExistsAsync` 方法创建数据库; 如果不存在，如果它已存在，则返回的数据库。但是，示例应用程序将忽略返回任何数据 `CreateDatabaseIfNotExistsAsync` 方法。

> **NOTE**
> `CreateDatabaseIfNotExistsAsync` 方法将返回 `Task<ResourceResponse<Database>>` 可以检查以确定是否创建了数据库，或返回现有数据库对象，并在响应的状态代码。

创建文档集合

文档集合是 JSON 文档的容器，并且可以创建在 Azure 门户中，或以编程方式使用 `DocumentClient.CreateDocumentCollectionIfNotExistsAsync` 方法：

```
public async Task CreateDocumentCollection(string databaseName, string collectionName)
{
  ...
  // Create collection with 400 RU/s
  await client.CreateDocumentCollectionIfNotExistsAsync(
    UriFactory.CreateDatabaseUri(databaseName),
    new DocumentCollection
    {
      Id = collectionName
    },
    new RequestOptions
    {
      OfferThroughput = 400
    });
  ...
}
```

`CreateDocumentCollectionIfNotExistsAsync` 方法需要两个强制性参数 – 数据库名称指定为 `Uri`，和一个 `DocumentCollection` 对象。`DocumentCollection` 对象表示与指定其名称的文档集合 `Id` 属性。`CreateDocumentCollectionIfNotExistsAsync` 方法创建文档集合; 如果不存在，如果它已存在，则返回的文档集合。但是，示例应用程序将忽略返回任何数据 `CreateDocumentCollectionIfNotExistsAsync` 方法。

> **NOTE**
> `CreateDocumentCollectionIfNotExistsAsync` 方法将返回 `Task<ResourceResponse<DocumentCollection>>` 检查对象，并在响应的状态代码以确定是否已创建一个文档集合，或返回现有的文档集合。

（可选）`CreateDocumentCollectionIfNotExistsAsync` 方法还可以指定 `RequestOptions` 对象，封装到 Cosmos DB 帐户发出的请求可以指定的选项。`RequestOptions.OfferThroughput` 属性用于定义文档集合的性能级别和在示例应用程序中，设置为每秒 400 个请求单位。此值应增加或减少具体取决于是否将频繁或不常访问集合。

> **IMPORTANT**
> 请注意，`CreateDocumentCollectionIfNotExistsAsync` 方法将创建一个新集合使用保留的吞吐量，它牵涉定价。

检索文档集合文档

可以通过创建并执行文档查询检索文档集合中的内容。使用创建文档查询 `DocumentClient.CreateDocumentQuery` 方法：

```
public async Task<List<TodoItem>> GetTodoItemsAsync()
{
  ...
  var query = client.CreateDocumentQuery<TodoItem>(collectionLink)
            .AsDocumentQuery();
  while (query.HasMoreResults)
  {
    Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
  }
  ...
}
```

此查询以异步方式从指定集合中检索所有文档，并将在文档 `List<TodoItem>` 显示的集合。

`CreateDocumentQuery<T>` 方法指定 `Uri` 表示应查询的文档的集合的参数。在此示例中，`collectionLink` 变量是类级字段，用于指定 `Uri`，表示要检索来自文档的文档集合：

```
Uri collectionLink = UriFactory.CreateDocumentCollectionUri(Constants.DatabaseName, Constants.CollectionName);
```

`CreateDocumentQuery<T>` 方法创建一个查询，以同步方式，执行并返回 `IQueryable<T>` 对象。但是，`AsDocumentQuery` 方法将 `IQueryable<T>` 对象传递给 `IDocumentQuery<T>` 可以异步执行的对象。与执行异步查询 `IDocumentQuery<T>.ExecuteNextAsync` 方法，检索结果，文档数据库中的下一页面进行 `IDocumentQuery<T>.HasMoreResults`，该值指示是否有更多结果，以从查询返回的属性。

文档可以是由包括筛选服务器端 `Where` 中查询，筛选谓词应用于针对文档集合的查询子句：

```
var query = client.CreateDocumentQuery<TodoItem>(collectionLink)
          .Where(f => f.Done != true)
          .AsDocumentQuery();
```

此查询检索所有记录集合从其 `Done` 属性等于 `false`。

将文档插入到文档集合

文档是用户定义 JSON 内容和可以插入到文档集合与 `DocumentClient.CreateDocumentAsync` 方法：

```
public async Task SaveTodoItemAsync(TodoItem item, bool isNewItem = false)
{
  ...
  await client.CreateDocumentAsync(collectionLink, item);
  ...
}
```

`CreateDocumentAsync` 方法指定 `Uri` 表示应将文档插入到，该集合的参数和一个 `object` 参数表示要插入的文档。

替换文档集合中的文档

可以使用文档集合中替换文档 `DocumentClient.ReplaceDocumentAsync` 方法：

```
public async Task SaveTodoItemAsync(TodoItem item, bool isNewItem = false)
{
  ...
  await client.ReplaceDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
Constants.CollectionName, item.Id), item);
  ...
}
```

`ReplaceDocumentAsync` 方法指定 `Uri` 表示应替换为集合中的文档的参数和一个 `object` 表示更新后的文档数据的参数。

文档集合中删除文档

可以从使用文档集合中删除一个文档 `DocumentClient.DeleteDocumentAsync` 方法：

```
public async Task DeleteTodoItemAsync(string id)
{
  ...
  await client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
Constants.CollectionName, id));
  ...
}
```

`DeleteDocumentAsync` 方法指定 `Uri` 表示应删除的集合中的文档的参数。

删除文档集合

可以使用数据库中删除文档集合 `DocumentClient.DeleteDocumentCollectionAsync` 方法：

```
await client.DeleteDocumentCollectionAsync(collectionLink);
```

`DeleteDocumentCollectionAsync` 方法指定 `Uri` 表示要删除的文档集合参数。请注意，调用此方法也会删除存储在集合中的文档。

删除数据库

可以使用 Cosmos DB 数据库帐户中删除某个数据库 `DocumentClient.DeleteDatabaesAsync` 方法：

```
await client.DeleteDatabaseAsync(UriFactory.CreateDatabaseUri(Constants.DatabaseName));
```

`DeleteDatabaseAsync` 方法指定 `Uri` 参数表示要删除的数据库。请注意，调用此方法也会删除存储在数据库中，文档集合和文档集合中存储的文档。

总结

本文介绍了如何使用 Azure Cosmos DB.NET Standard 客户端库将 Azure Cosmos DB 文档数据库集成到 Xamarin.Forms 应用程序。Azure Cosmos DB 文档数据库是提供低延迟访问产品/服务面向需要无缝缩放和全局复制的应用程序的快速、高度可用、可缩放数据库服务的 JSON 文档的 NoSQL 数据库。

相关链接

- Todo Azure Cosmos DB（示例）
- Azure Cosmos DB 文档
- Azure Cosmos DB.NET Standard 客户端库

- Azure Cosmos DB API

使用 Azure Cosmos DB 文档数据库的用户进行身份验证

2018/6/9 • • Edit Online

Azure Cosmos DB 文档数据库支持已分区的集合，可以跨多个服务器和分区，同时支持无限的存储和吞吐量。此文章介绍了如何组合使用分区集合，使用访问控制，以便用户只能访问自己在 Xamarin.Forms 应用程序中的文档。

概述

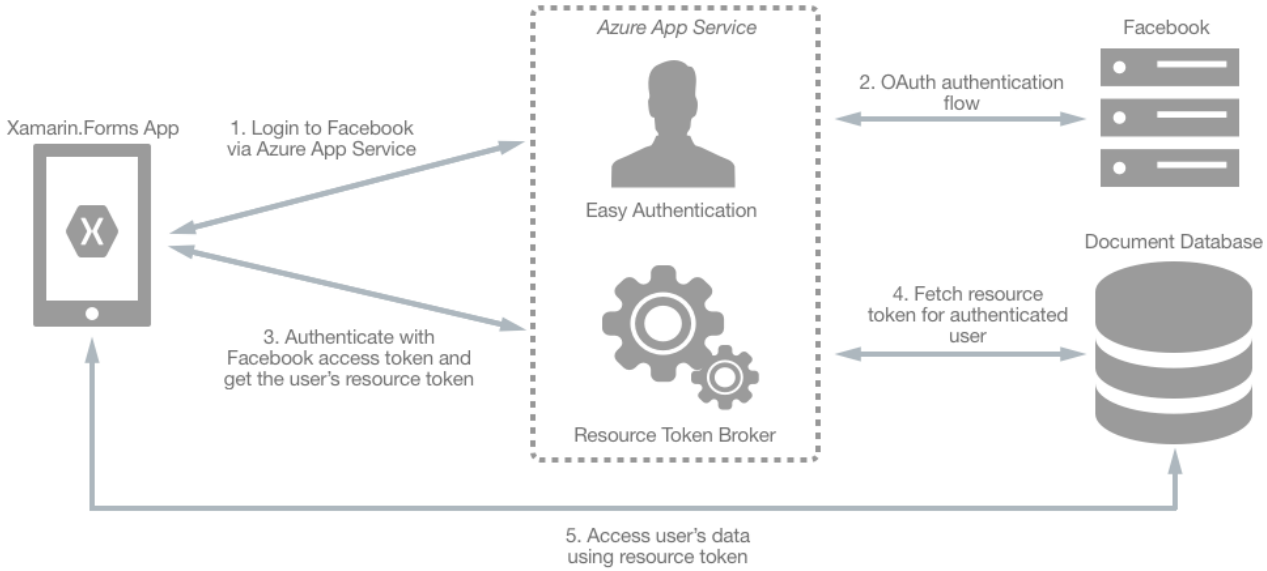
创建分区的集合时，必须指定分区键和文档具有相同的分区键将存储在同一分区中。因此，指定用户的标识用作分区键将导致分区的集合，将仅存储为该用户的文档。这还可以确保 Azure Cosmos DB 文档数据库会随着用户数和项增加。

任何集合，必须授予访问权限和 SQL API 的访问控制模型定义两种类型的访问构造：

- 主密钥启用在 Cosmos DB 帐户中，所有资源的完全管理访问权限和创建 Cosmos DB 帐户时创建。
- 资源令牌捕获的数据库用户和用户对某个特定的 Cosmos DB 资源，例如集合或文档拥有的权限之间的关系。

公开一个主密钥打开都会向可能的恶意或负面使用 Cosmos DB 帐户。但是，Azure Cosmos DB 资源令牌提供一种允许客户端读取、写入和删除 Azure Cosmos DB 帐户根据授予的权限中的特定资源的安全机制。

向请求典型的方法，生成，并将资源令牌传递到移动应用程序是使用资源令牌代理。下图显示示例应用程序如何使用资源令牌的代理来管理对文档数据库数据的访问的高级的概述：



资源令牌 broker 是承载在 Azure App Service，拥有 Cosmos 数据库帐户主密钥的中间层 Web API 服务。示例应用程序使用的资源令牌的代理管理对文档数据库数据的访问，如下所示：

1. 登录情况下，Xamarin.Forms 应用程序将联系 Azure 应用程序服务，以启动身份验证流。
2. Azure App Service 执行与 Facebook OAuth 身份验证流。身份验证流完成后，Xamarin.Forms 应用程序会收到一个访问令牌。
3. Xamarin.Forms 应用程序使用访问令牌请求从资源令牌 broker 资源令牌。
4. 资源令牌 broker 使用的访问令牌从 Facebook 请求用户的标识。然后，用户的标识用于请求从 Cosmos DB，用于授予对经过身份验证的用户的已分区集合的读/写访问的资源令牌。
5. Xamarin.Forms 应用程序使用的资源令牌直接使用的资源令牌所定义的权限访问 Cosmos DB 资源。

> **NOTE**
> 在资源令牌过期，后续文档数据库的请求将收到 401 未经授权的异常。此时，Xamarin.Forms 应用程序应重新建立标识，并请求新的资源令牌。

有关 Cosmos DB 分区的详细信息，请参阅如何分区和 Azure Cosmos DB 中的小数位数。有关 Cosmos 数据库访问控制的详细信息，请参阅保护对 Cosmos DB 数据访问和中 SQL API 的访问控制。

安装

将资源令牌 broker 集成到 Xamarin.Forms 应用程序的过程如下所示：

1. 创建将使用访问控制的 Cosmos DB 帐户。有关详细信息，请参阅Cosmos 数据库配置。
2. 创建 Azure 应用程序服务，以承载资源令牌代理。有关详细信息，请参阅Azure 应用程序服务配置。
3. 创建 Facebook 应用程序来执行身份验证。有关详细信息，请参阅Facebook 应用程序配置。
4. 配置 Azure 应用程序服务来执行与 Facebook 的简单身份验证。有关详细信息，请参阅Azure 应用程序服务身份验证配置。
5. 配置 Xamarin.Forms 示例应用程序与 Azure App Service 和 Cosmos DB 进行通信。有关详细信息，请参阅Xamarin.Forms 应用程序配置。

Azure Cosmos DB 配置

用于创建将使用访问控制的 Cosmos DB 帐户的过程如下所示：

1. 创建一个 Cosmos DB 帐户。有关详细信息，请参阅创建 Azure Cosmos DB 帐户。
2. 在 Cosmos DB 帐户中，创建一个名为的新集合 `UserItems`，指定分区键为 `/userid`。

Azure App Service 配置

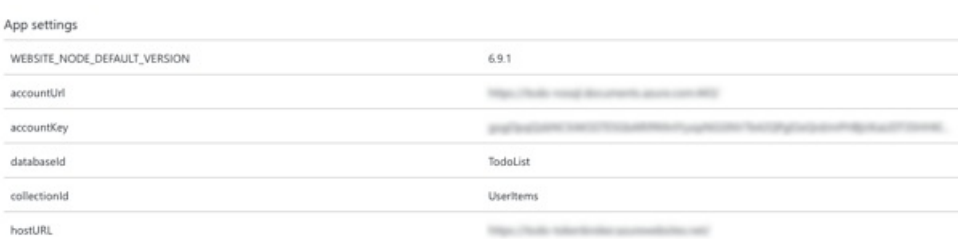
承载在 Azure App Service 中的资源令牌代理的过程如下所示：

1. 在 Azure 门户中，创建新的 App Service web 应用。有关详细信息，请参阅在 App Service 环境中创建 web 应用。

2. 在 Azure 门户中，打开 web 应用中，应用设置边栏选项卡，并添加以下设置：

   - `accountUrl` – 此值应该是从 Cosmos DB 帐户的密钥边栏选项卡的 Cosmos DB 帐户 URL。
   - `accountKey` – 值应为 Cosmos 数据库主密钥（主要或次要）从 Cosmos DB 帐户的密钥边栏选项卡。
   - `databaseId` – 值应为 Cosmos DB 数据库的名称。
   - `collectionId` – 值应为 Cosmos DB 集合的名称（在这种情况下，`UserItems`）。
   - `hostUrl` – 值应为应用程序服务帐户的概述边栏选项卡中的 web 应用的 URL。

   下面的屏幕截图演示了此配置：

3. 将资源令牌 broker 解决方案发布到 Azure App Service web 应用。

Facebook 应用程序配置

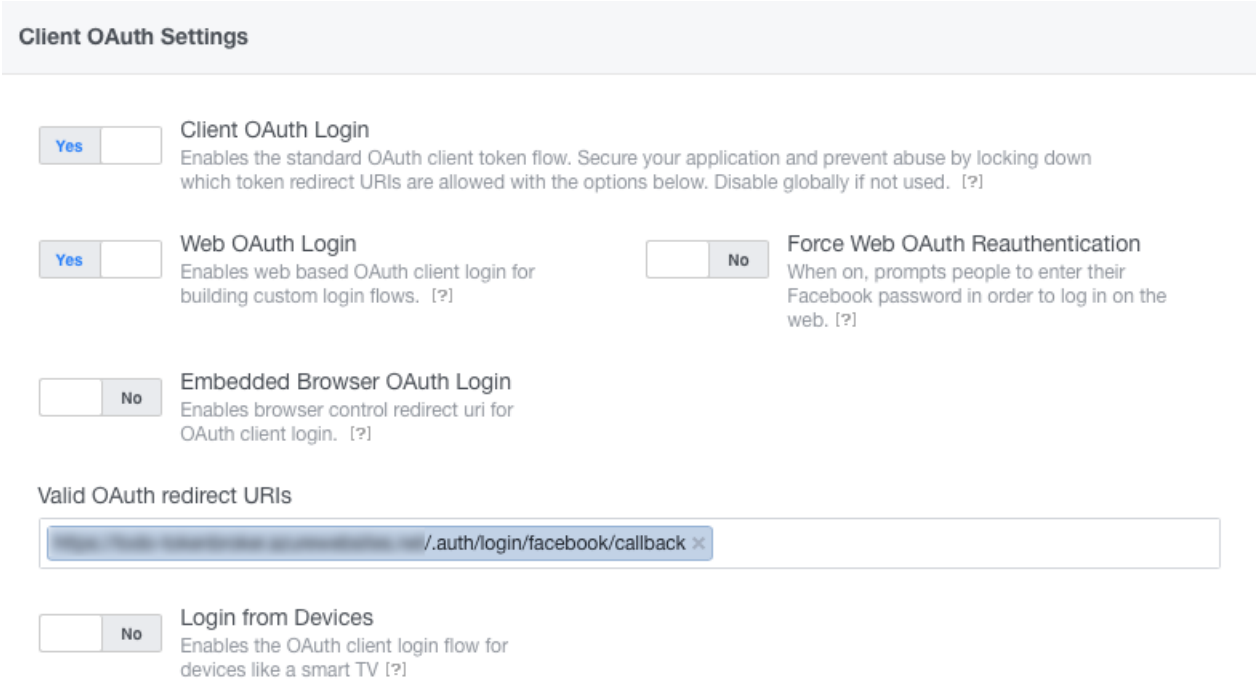
用于创建 Facebook 应用程序来执行身份验证的过程如下所示：

1. 创建 Facebook 应用程序。有关详细信息，请参阅注册和配置应用Facebook 开发人员中心上。

2. 将 Facebook 登录名产品添加到应用程序。有关详细信息，请参阅到你的应用或网站添加 Facebook 登录名Facebook 开发人员中心上。
3. 配置 Facebook 登录名，如下所示：

- 启用客户端 OAuth 登录名。
- 启用 Web OAuth 登录名。
- 设置有效的 OAuth 重定向到 App Service web 应用，URI 的 URI `/.auth/login/facebook/callback` 追加。

下面的屏幕截图演示了此配置：



有关详细信息，请参阅向 Facebook 注册你的应用程序。

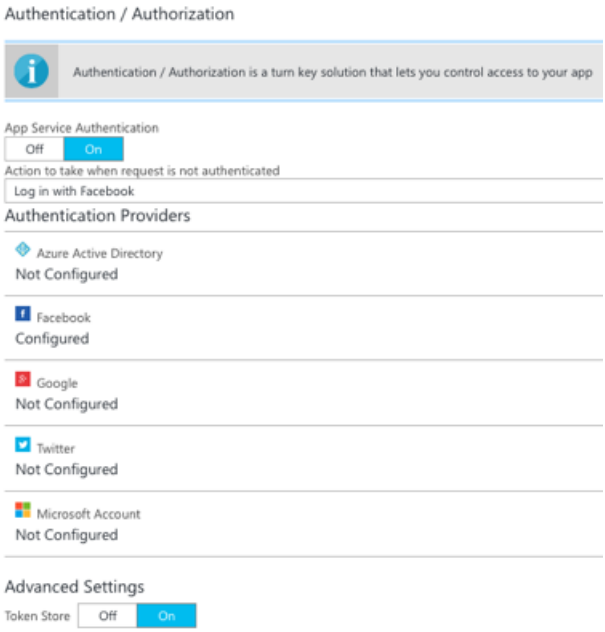
Azure App Service 身份验证配置

配置 App Service 简单身份验证的过程如下所示：

1. 在 Azure 门户中，导航到 App Service web 应用。
2. 在 Azure 门户中，打开身份验证 / 授权边栏选项卡并执行以下配置：

- 应启用应用程序服务身份验证。
- 当请求未经过身份验证时要执行的操作应设置为**使用 Facebook 登录名**。

下面的屏幕截图演示了此配置：



App Service web 应用还应配置为与 Facebook 应用程序启用的身份验证流通信。这可以通过选择 Facebook 标识提供程序，并输入应用程序 ID 和应用程序密钥从 Facebook 开发人员中心上的 Facebook 应用程序设置的值。有关详细信息，请参阅到你的应用程序添加 Facebook 信息。

Xamarin.Forms 应用程序配置

配置 Xamarin.Forms 示例应用程序的过程如下所示：

1. 打开 Xamarin.Forms 解决方案。
2. 打开 `Constants.cs` 和更新以下常量的值：

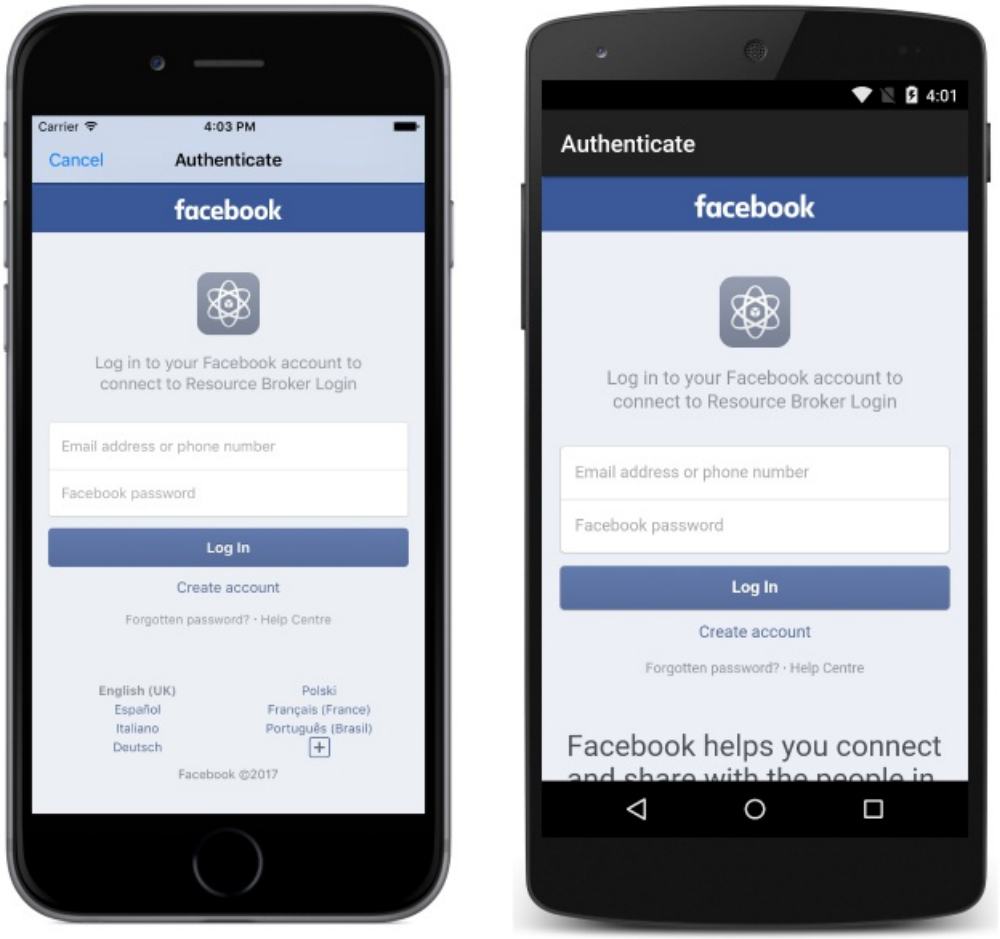
- `EndpointUri` – 此值应该是从 Cosmos DB 帐户的密钥边栏选项卡的 Cosmos DB 帐户 URL。
- `DatabaseName` – 值应为文档数据库的名称。
- `CollectionName` – 值应为文档数据库集合的名称 (在这种情况下， `UserItems`)。
- `ResourceTokenBrokerUrl` – 值应为 App Service 帐户的概述边栏选项卡中的资源令牌的代理 web 应用程序的 URL。

启动的登录名

示例应用程序启动登录过程，通过使用 Xamarin.Auth 浏览器重定向到标识提供程序 URL，如下面的代码示例中所示：

```
var auth = new Xamarin.Auth.WebRedirectAuthenticator(
  new Uri(Constants.ResourceTokenBrokerUrl + "/.auth/login/facebook"),
  new Uri(Constants.ResourceTokenBrokerUrl + "/.auth/login/done"));
```

这将导致启动之间 Azure App Service 和 Facebook，Facebook 登录页将显示一个 OAuth 身份验证流程：



登录可以取消按**取消**按钮在 iOS 上或通过按**回**在 Android 上，在这种情况下的用户保持状态未经身份验证和标识提供程序用户界面的按钮从屏幕上删除。

有关 Xamarin.Auth 的详细信息，请参阅与标识提供程序进行身份验证的用户。

获取一个资源令牌

成功通过身份验证，按照 `WebRedirectAuthenticator.Completed` 事件激发。下面的代码示例演示如何处理此事件：

```
auth.Completed += async (sender, e) =>
{
  if (e.IsAuthenticated && e.Account.Properties.ContainsKey("token"))
  {
    var easyAuthResponseJson = JsonConvert.DeserializeObject<JObject>(e.Account.Properties["token"]);
    var easyAuthToken = easyAuthResponseJson.GetValue("authenticationToken").ToString();

    // Call the ResourceBroker to get the resource token
    using (var httpClient = new HttpClient())
    {
      httpClient.DefaultRequestHeaders.Add("x-zumo-auth", easyAuthToken);
      var response = await httpClient.GetAsync(Constants.ResourceTokenBrokerUrl + "/api/resourcetoken/");
      var jsonString = await response.Content.ReadAsStringAsync();
      var tokenJson = JsonConvert.DeserializeObject<JObject>(jsonString);
      resourceToken = tokenJson.GetValue("token").ToString();
      UserId = tokenJson.GetValue("userid").ToString();

      if (!string.IsNullOrWhiteSpace(resourceToken))
      {
        client = new DocumentClient(new Uri(Constants.EndpointUri), resourceToken);
        ...
      }
      ...
    }
  }
};
```

成功的身份验证的结果是访问令牌，它位于 `AuthenticatorCompletedEventArgs.Account` 属性。提取并在资源令牌代理的 GET 请求中使用访问令牌 `resourcetoken` API。

`resourcetoken` API 使用的访问令牌从 Facebook，又用于从 Cosmos DB 请求的资源令牌请求用户的标识。如果文档数据库中的用户已存在有效的权限的文档，检索到它，并包含资源令牌的 JSON 文档返回到 Xamarin.Forms 应用程序。如果用户不存在有效的权限文档，在文档数据库中，创建用户和权限和资源令牌是从权限文档中提取并返回到 JSON 文档中的 Xamarin.Forms 应用程序。

> **NOTE**
>
> 文档数据库用户是文档数据库，与关联的资源，并且每个数据库可能包含零个或多个用户。文档数据库权限是与文档数据库用户相关联的资源，并且每个用户可能包含零个或多个权限。权限资源提供了对用户需要在尝试访问某个资源例如文档时的安全令牌的访问。

如果 `resourcetoken` API 成功完成后，它将发送 HTTP 状态代码 200（正常）在响应中，以及 JSON 文档包含的资源令牌。下面的 JSON 数据显示了典型成功的响应消息：

```
{
  "id": "John Smithpermission",
  "token":
"type=resource&ver=1&sig=zx6k2zzxqktzvuzuku4b7y==;a74aukk99qtwk8v5rxfrfz7ay7zzqfkbfkremrwtaapvavw2mrvia4umbi/7
iiwkrrq+buqqrzkaq4pp15y6bki1u//zf7p9x/aefbvqvq3tjjqiffurfx+vexa1xarxkkv9rbua9ypfzr47xpp5vmxuvzbekkwq6txme0xxxb
jhzaxbkvzaji+iru3xqjp05amvq1r1q2k+qrarurhmjzah/ha0evixazkve2xk1zu9u/jpyf1xrwbkxqpzebvqwma+hyyaazemr6qx9uz9be==
;",
  "expires": 4035948,
  "userid": "John Smith"
}
```

`WebRedirectAuthenticator.Completed` 事件处理程序读取的回应 `resourcetoken` API 并提取资源令牌和用户 id。资源令牌然后作为的自变量传递 `DocumentClient` 构造函数，它封装终结点、凭据和连接策略用于访问 Cosmos DB，并用于配置和执行针对 Cosmos DB 的请求。资源令牌与直接访问资源，每个请求一起发送，并指示授予了对经过身份验证的用户的已分区集合的读/写访问。

检索文档

可以通过在下面的代码示例中创建包括用户的 id 作为分区键，并演示了一个文档查询来检索仅属于已验证用户的文档：

```
var query = client.CreateDocumentQuery<TodoItem>(collectionLink,
                        new FeedOptions
                        {
                          MaxItemCount = -1,
                          PartitionKey = new PartitionKey(UserId)
                        })
          .Where(item => !item.Id.Contains("permission"))
          .AsDocumentQuery();
while (query.HasMoreResults)
{
  Items.AddRange(await query.ExecuteNextAsync<TodoItem>());
}
```

查询以异步方式检索所有属于已验证的用户，从指定集合中的文档，并将它们放入 `List<TodoItem>` 显示的集合。

`CreateDocumentQuery<T>` 方法指定 `Uri` 表示应为文档，查询的集合的自变量和一个 `FeedOptions` 对象。 `FeedOptions` 对象指定，可以通过查询，并用作分区键的用户的 id 返回无限的数量的项。这可确保结果中返回，仅在用户的已分区集合中的文档。

> **NOTE**
> 请注意，权限文档，它由资源令牌 broker，存储在相同的文档集合作为 Xamarin.Forms 应用程序创建的文档。因此，该文档查询包含 `Where` 筛选谓词应用于针对文档集合的查询的子句。此子句可确保权限文档不从文档集合中返回。

有关从文档集合中检索文档的详细信息，请参阅检索文档集合文档。

插入文档

在将文档插入到文档集合之前，`TodoItem.UserId` 应使用的值用作分区键，更新属性，如下面的代码示例中所示：

```
item.UserId = UserId;
await client.CreateDocumentAsync(collectionLink, item);
```

这可确保文档，将插入到用户的已分区的集合。

有关将文档插入到文档集合的详细信息，请参阅将文档插入到文档集合。

删除文档

删除分区集合中从文档中下面的代码示例演示时，必须指定分区密钥值：

```
await client.DeleteDocumentAsync(UriFactory.CreateDocumentUri(Constants.DatabaseName,
Constants.CollectionName, id),
                new RequestOptions
                {
                  PartitionKey = new PartitionKey(UserId)
                });
```

这可确保 Cosmos DB 知道其分区删除中的文档的集合。

有关从文档集合中删除文档的详细信息，请参阅从文档集合中删除文档。

总结

本文介绍了如何将访问控制与分区集合结合起来，以便用户只能访问自己在 Xamarin.Forms 应用程序中的文档数据库文档。指定用户的标识用作分区键可确保分区的集合可以仅存储为该用户的文档。

相关链接

- Todo Azure Cosmos DB 身份验证（示例）
- 使用 Azure Cosmos DB 文档数据库
- 保护对 Azure Cosmos DB 数据访问
- SQL API 中的访问控制。
- 如何分区和 Azure Cosmos DB 中的小数位数
- Azure Cosmos DB 客户端库
- Azure Cosmos DB API

添加智能与认知服务

2018/8/2 • • Edit Online

Microsoft 认知服务是一套 Api、Sdk 和供开发人员可以通过添加功能，如面部识别、语音识别和语言理解，使其应用程序更智能的服务。本文提供的示例应用程序演示如何调用的一些 Microsoft 认知服务 Api 的简介。

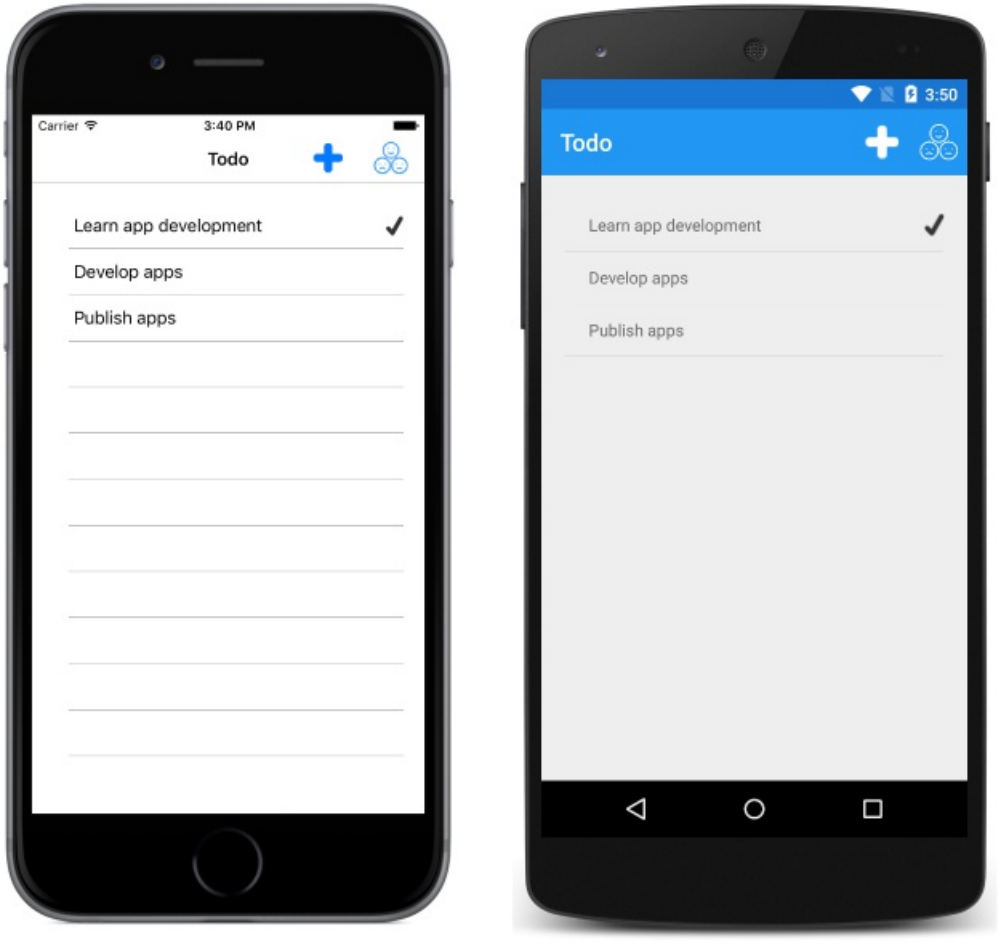
概述

随附的示例是 todo 列表应用程序提供了以下功能：

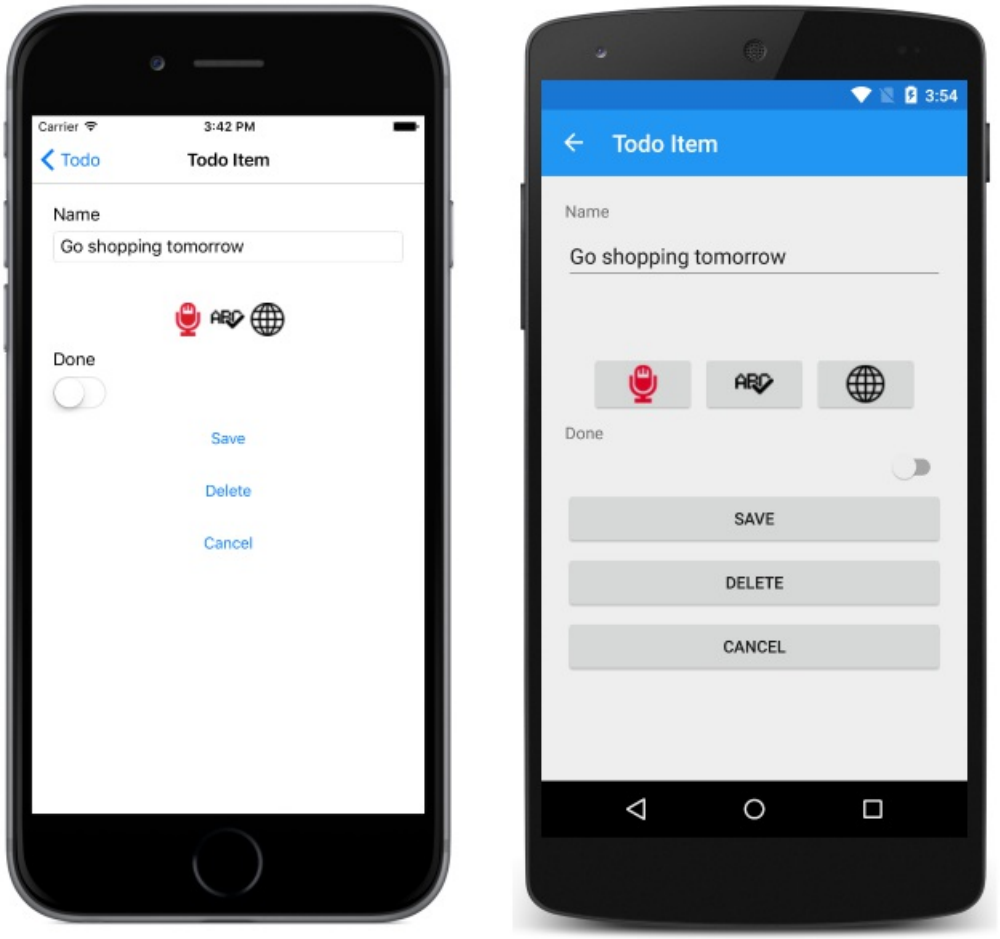
- 查看任务的列表。
- 添加和编辑通过软键盘，或使用 Microsoft 语音 API 的语音识别中执行的任务。有关执行语音识别的详细信息，请参阅语音识别使用 Microsoft 语音 API。
- 拼写检查任务使用必应拼写检查 API。有关详细信息，请参阅拼写检查使用必应拼写检查 API。
- 将转换为德语使用转换器 API 从英语的任务。有关详细信息，请参阅文本转换使用转换器 API。
- 删除任务。
- 设置任务的状态设置为完成。
- 速率表情识别，使用表面 API 的应用程序。有关详细信息，请参阅表情识别使用表面 API。

任务存储在本地的 SQLite 数据库。有关使用本地的 SQLite 数据库的详细信息，请参阅处理本地数据库。

`TodoListPage` 启动应用程序时显示。此页显示存储在本地数据库中，任何任务的列表，并允许用户创建新任务或来对应用程序：

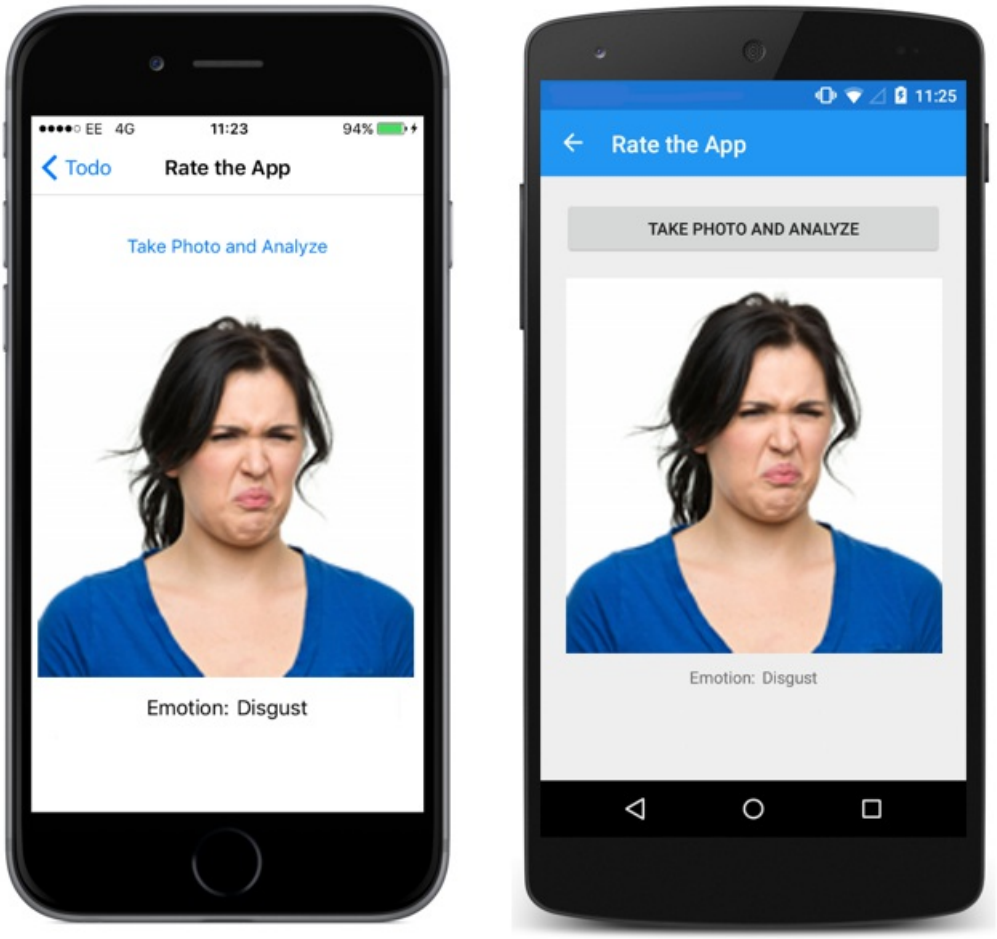


可以通过单击创建新项*+* 按钮，导航到 `TodoItemPage`。此页还可以通过选择一项任务导航：



`TodoItemPage` 允许任务要创建、编辑、拼写检查转换，保存，并将删除。语音识别可用来创建或编辑任务。这被实现通过按麦克风按钮以开始记录，并通过第二次按相同的按钮停止录制，这将记录发送到 Bing 语音识别 API。

单击 smilies 按钮 `TodoListPage` 导航到 `RateAppPage`，用于执行情感识别的面部表情图像上：



`RateAppPage` 允许用户来执行其字面，显示返回表情与提交到表面 API 的照片。

了解应用程序剖析

示例应用程序的可移植类库 (PCL) 项目包含五个主要文件夹：

文件夹	目标
模型	包含应用程序的数据模型类。这包括 `TodoItem` 类，该类建模的数据应用程序使用单个项。该文件夹还包括用于从不同 Microsoft 认知服务 Api 返回的模型 JSON 响应的类。
存储库	包含 `ITodoItemRepository` 接口和 `TodoItemRepository` 用于执行数据库操作的类。
服务	包含的接口和用于访问不同 Microsoft 认知服务 Api，并使用的接口的类 `DependencyService` 类用来定位在平台项目中实现的接口的类。
Utils	包含 `Timer` 类，该类由 `AuthenticationService` 类若要续订一个 JWT 访问令牌每隔 9 分钟。
视图	包含应用程序的页。

PCL 项目中还包含一些重要的文件：

文件	目标
Constants.cs	`Constants` 类，从而为 Microsoft 认知服务 Api 调用中指定的 API 密钥和终结点。API 密钥常量需要更新访问不同的认知服务 Api。
App.xaml.cs	`App` 类负责实例化的这两个将显示的每个平台上的应用程序的第一页和 `TodoManager` 用于调用数据库操作的类。

NuGet 包

示例应用程序使用以下 NuGet 包：

- `Newtonsoft.Json` –.NET 提供的 JSON 框架。
- `PCLStorage` – 提供了一套跨平台本地文件 IO Api。
- `sqlite-net-pcl` – 提供 SQLite 数据库存储。
- `Xam.Plugin.Media` – 提供跨平台照片拍摄和 Api。

此外，这些 NuGet 包还会安装其自己的依赖关系。

对数据进行建模

示例应用程序使用 `TodoItem` 类模型显示和本地的 SQLite 数据库中存储的数据。以下代码示例演示 `TodoItem` 类：

```
public class TodoItem
{
  [PrimaryKey, AutoIncrement]
  public int ID { get; set; }
  public string Name { get; set; }
  public bool Done { get; set; }
}
```

`ID` 属性用于唯一标识每个 `TodoItem` 实例，并使用 SQLite 数据库中自动递增的主键将该属性的属性修饰。

调用数据库操作

`TodoItemRepository` 类实现的数据库操作，并可以通过访问类的实例 `App.TodoManager` 属性。`TodoItemRepository` 类提供下列方法来调用数据库操作：

- **GetAllItemsAsync** – 从本地的 SQLite 数据库中检索的所有项。
- **GetItemAsync** – 从本地的 SQLite 数据库中检索指定的项。
- **SaveItemAsync** – 创建或更新本地的 SQLite 数据库中的项。
- **DeleteItemAsync** – 从本地的 SQLite 数据库中删除指定的项。

平台项目实现

`Services` PCL 项目中的文件夹包含 `IFileHelper` 和 `IAudioRecorderService` 使用的接口 `DependencyService` 类用来定位在平台项目中实现的接口的类。

`IFileHelper` 接口由实现 `FileHelper` 每个平台项目中的类。此类包含单个方法 `GetLocalFilePath`，这将返回一个用于存储 SQLite 数据库的本地文件路径。

`IAudioRecorderService` 接口由实现 `AudioRecorderService` 每个平台项目中的类。此类包含 `StartRecording`，`StopRecording`，和支持的方法，这在平台 Api 中使用来自设备的麦克风录制音频，并将其存储为 wav 文件。在 iOS 上 `AudioRecorderService` 使用 `AVFoundation` 录制音频的 API。在 Android 上，`AudioRecordService` 使用 `AudioRecord` 录制音频的 API。在通用 Windows 平台 (UWP)，`AudioRecorderService` 使用 `AudioGraph` 录制音频的 API。

调用认知服务

示例应用程序将调用以下 Microsoft 认知服务：

- Microsoft 语音 API。有关详细信息，请参阅语音识别使用 Microsoft 语音 API。
- 必应拼写检查 API。有关详细信息，请参阅拼写检查使用必应拼写检查 API。
- 将转换 API。有关详细信息，请参阅文本转换使用转换器 API。
- 表面 API。有关详细信息，请参阅表情识别使用表面 API。

相关链接

- Microsoft 认知服务文档
- Todo 认知服务（示例）

使用 Microsoft 语音 API 的语音识别

2018/11/14 • • Edit Online

Microsoft 语音 API 是基于云的 API，提供了用于处理口述的语言的算法。本文介绍如何使用 Microsoft 语音识别 REST API 将音频转换为 Xamarin.Forms 应用程序中的文本。

概述

Microsoft 语音 API 有两个组件：

- 将语音转换为文本的语音识别 API。可以通过 REST API、客户端库或服务库执行语音识别。
- 文本到语音转换 API 将文本转换成口头字词。通过 REST API 执行文本到语音转换。

本文重点介绍执行通过 REST API 的语音识别。客户端和服务库支持返回部分结果，REST API 只能返回单个识别结果，而无需任何部分结果。

若要使用 Microsoft 语音 API，必须获取 API 密钥。这可以通过 Azure 获得门户。有关详细信息，请参阅Azure 门户中创建认知服务帐户。

有关 Microsoft 语音 API 的详细信息，请参阅Microsoft 语音 API 文档。

身份验证

Microsoft 语音 REST API 对所做的每个请求需要一个 JSON Web 令牌 (JWT) 访问令牌，可以从在认知服务令牌服务获取 `https://api.cognitive.microsoft.com/sts/v1.0/issueToken`。可以通过向令牌服务中，发出 POST 请求来获取令牌指定 `Ocp-Apim-Subscription-Key` 标题，其中包含作为其值的 API 密钥。

下面的代码示例演示如何请求访问令牌的令牌服务：

```
public AuthenticationService(string apiKey)
{
    subscriptionKey = apiKey;
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", apiKey);
}
...
async Task<string> FetchTokenAsync(string fetchUri)
{
    UriBuilder uriBuilder = new UriBuilder(fetchUri);
    uriBuilder.Path += "/issueToken";
    var result = await httpClient.PostAsync(uriBuilder.Uri.AbsoluteUri, null);
    return await result.Content.ReadAsStringAsync();
}
```

返回的访问令牌，它为 Base64 文本时，有 10 分钟的到期时间。因此，示例应用程序将续订访问令牌每隔 9 分钟。

必须在每个 Microsoft 语音 REST API 中指定的访问令牌调用作为 `Authorization` 与字符串作为前缀的标头 `Bearer`，下面的代码示例中所示：

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);
```

未能将有效的访问令牌传递到 Microsoft 语音 REST API 将导致 403 响应错误。

执行语音识别

语音识别实现对的 POST 请求，从而 `recognition` API， `https://speech.platform.bing.com/speech/recognition/` 。单个请求不能包含超过 10 秒的音频，并且总请求持续时间不能超过 14 秒。

必须在 wav 格式中的请求的 POST 正文中放置的音频内容。

在示例应用程序， `RecognizeSpeechAsync` 方法将调用语音识别过程：

```
public async Task<SpeechResult> RecognizeSpeechAsync(string filename)
{
    ...

    // Read audio file to a stream
    var file = await PCLStorage.FileSystem.Current.LocalStorage.GetFileAsync(filename);
    var fileStream = await file.OpenAsync(PCLStorage.FileAccess.Read);

    // Send audio stream to Bing and deserialize the response
    string requestUri = GenerateRequestUri(Constants.SpeechRecognitionEndpoint);
    string accessToken = authenticationService.GetAccessToken();
    var response = await SendRequestAsync(fileStream, requestUri, accessToken, Constants.AudioContentType);
    var speechResult = JsonConvert.DeserializeObject<SpeechResult>(response);

    fileStream.Dispose();
    return speechResult;
}
```

音频录制为 PCM wav 数据，每个特定于平台的项目中， `RecognizeSpeechAsync` 方法使用 `PCLStorage` NuGet 包，以打开音频文件作为流。语音识别请求生成 URI 和一个访问令牌进行检索的令牌服务。语音识别请求发布到 `recognition` API，它将返回包含结果的 JSON 响应。JSON 响应进行反序列化，与返回给调用方法以显示结果。

配置语音识别

可以通过指定 HTTP 查询参数来配置语音识别过程：

```
string GenerateRequestUri(string speechEndpoint)
{
    // To build a request URL, you should follow:
    // https://docs.microsoft.com/azure/cognitive-services/speech/getstarted/getstartedrest
    string requestUri = speechEndpoint;
    requestUri += @"dictation/cognitiveservices/v1?";
    requestUri += @"language=en-us";
    requestUri += @"&format=simple";
    System.Diagnostics.Debug.WriteLine(requestUri.ToString());
    return requestUri;
}
```

通过执行的主要配置 `GenerateRequestUri` 方法是设置的音频内容的区域设置。有关支持的区域设置的列表，请参阅支持的语言。

发送请求

`SendRequestAsync` 方法将向 Microsoft 语音 REST API 发出 POST 请求并返回的响应：

```
async Task<string> SendRequestAsync(Stream fileStream, string url, string bearerToken, string contentType)
{
    if (httpClient == null)
    {
        httpClient = new HttpClient();
    }
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);

    var content = new StreamContent(fileStream);
    content.Headers.TryAddWithoutValidation("Content-Type", contentType);
    var response = await httpClient.PostAsync(url, content);
    return await response.Content.ReadAsStringAsync();
}
```

此方法生成的 POST 请求：

- 包装中的音频流 `StreamContent` 实例提供基于流的 HTTP 内容。
- 设置 `Content-Type` 到请求的标头 `audio/wav; codec="audio/pcm"; samplerate=16000` 。
- 添加到访问令牌 `Authorization` 标头，使用字符串作为前缀 `Bearer` 。

然后将 POST 请求发送到 `recognition` API。然后将对响应进行读取，并将其返回给调用的方法。

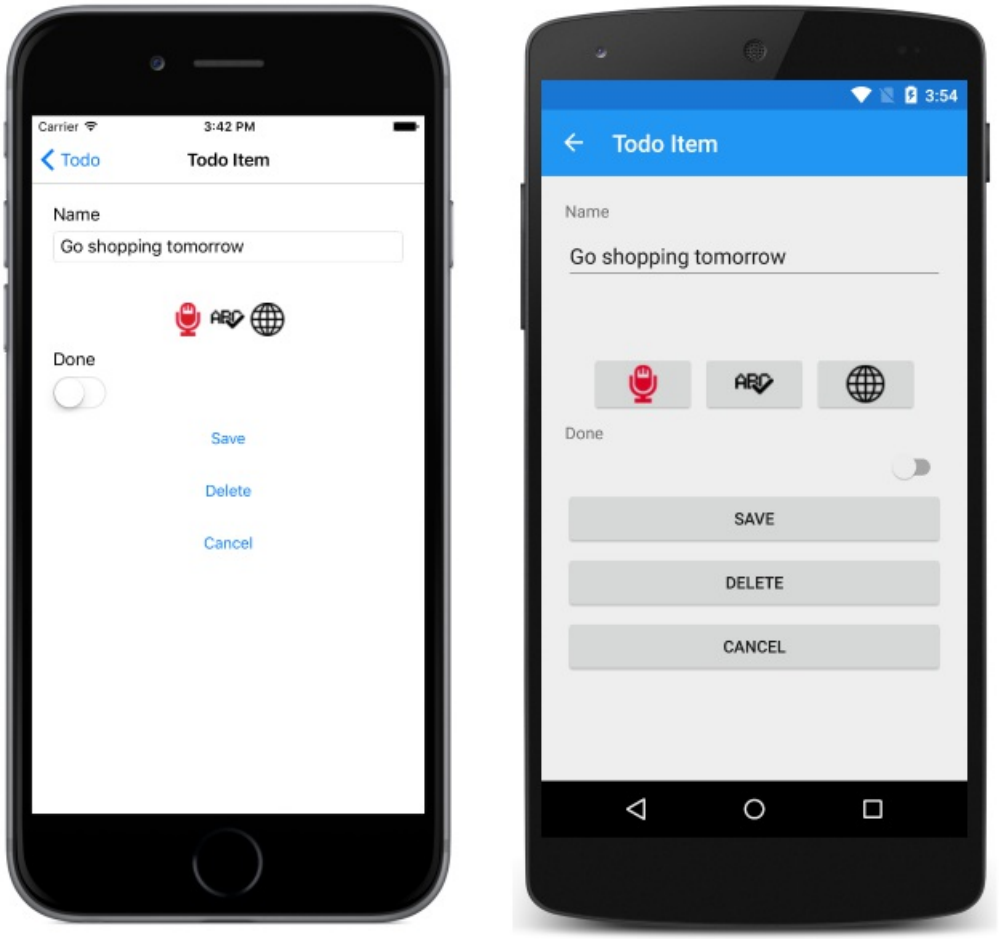
`recognition` API 将在响应中，提供请求的有效，指示请求成功，并且请求的信息包含在响应中发送 HTTP 状态代码 200（正常）。有关可能的错误响应的列表，请参阅故障排除。

处理响应

API 响应以 JSON 格式返回已识别的文本包含在与 `name` 标记。以下 JSON 数据显示了典型的成功响应消息：

```
{
   "RecognitionStatus":"Success",
   "DisplayText":"Go shopping tomorrow.",
   "Offset":16000000,
   "Duration":17100000
}
```

在示例应用程序，则 JSON 响应反序列化为 `SpeechResult` 实例，与结果返回到调用方法以显示，如以下屏幕截图中所示：



总结

本文介绍了如何使用 Microsoft 语音 REST API 将音频转换为 Xamarin.Forms 应用程序中的文本。除了执行语音识别，Microsoft 语音 API 还可以将文本转换成口语。

相关链接

- Microsoft 语音 API 文档。
- 使用 RESTful Web 服务
- Todo 认知服务（示例）

拼写检查使用必应拼写检查 API

2018/6/22 • • Edit Online

必应拼写检查执行上下文的拼写检查的文本，提供有关拼写错误的单词的内联建议。此文章介绍了如何使用必应拼写检查 REST API 来更正 Xamarin.Forms 应用程序中的拼写错误。

概述

必应拼写检查 REST API 具有两种操作模式，并对 API 发出请求时，必须指定一种模式：

- `Spell` 无任何大小写更改更正短文本（最多 9 个字）。
- `Proof` 更正长文本，提供的大小写更正和基本标点，并取消主动更正。

若要使用必应拼写检查 API，必须获取 API 密钥。这可以在获取重认知服务

必应拼写检查 API 支持的语言的列表，请参阅支持的语言。有关必应拼写检查 API 的详细信息，请参阅必应拼写检查文档。

身份验证

每个请求都会到必应拼写检查 API 需要 API 密钥应指定的值为 `Ocp-Apim-Subscription-Key` 标头。下面的代码示例演示如何添加到 API 密钥 `Ocp-Apim-Subscription-Key` 请求标头：

```
public BingSpellCheckService()
{
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", Constants.BingSpellCheckApiKey);
}
```

未能将有效的 API 密钥传递给必应拼写检查 API 将导致 401 响应错误。

执行拼写检查

拼写检查功能可以通过在 GET 或 POST 请求 `SpellCheck` API 在 `https://api.cognitive.microsoft.com/bing/v7.0/SpellCheck` 。在发出 GET 请求时，要进行拼写检查的文本作为查询参数发送。在发出 POST 请求，请求正文中发送的文本来进行拼写检查。GET 请求仅限于拼写检查 1500年个字符，因为查询参数字符串长度限制。因此，仅除非短字符串正在拼写检查时，通常应进行 POST 请求。

在示例应用程序，`SpellCheckTextAsync` 方法调用拼写检查过程：

```
public async Task<SpellCheckResult> SpellCheckTextAsync(string text)
{
    string requestUri = GenerateRequestUri(Constants.BingSpellCheckEndpoint, text, SpellCheckMode.Spell);
    var response = await SendRequestAsync(requestUri);
    var spellCheckResults = JsonConvert.DeserializeObject<SpellCheckResult>(response);
    return spellCheckResults;
}
```

`SpellCheckTextAsync` 方法生成请求 URI，并随后发送到请求 `SpellCheck` API，它返回包含结果的 JSON 响应。JSON 响应进行反序列化，返回到调用方法以显示结果。

配置拼写检查

可以通过指定 HTTP 查询参数配置拼写检查过程：

```
string GenerateRequestUri(string spellCheckEndpoint, string text, SpellCheckMode mode)
{
  string requestUri = spellCheckEndpoint;
  requestUri += string.Format("?text={0}", text);                          // text to spell check
  requestUri += string.Format("&mode={0}", mode.ToString().ToLower());     // spellcheck mode - proof or spell
  return requestUri;
}
```

此方法设置拼写检查，且拼写检查模式的文本。

有关必应拼写检查 REST API 的详细信息，请参阅拼写检查 API v7 参考。

发送请求

`SendRequestAsync` 方法向必应拼写检查 REST API 发出 GET 请求，并返回响应：

```
async Task<string> SendRequestAsync(string url)
{
    var response = await httpClient.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}
```

此方法可发送 GET 请求到 `SpellCheck` API，使用请求 URL 指定的文本要转换和拼写检查模式。然后将对响应进行读取，并将其返回到调用方法。

`SpellCheck` API 将在响应中，提供该请求是有效的表示请求成功，请求的信息包含在响应中发送 HTTP 状态代码 200（正常）。响应对象的列表，请参阅响应对象。

处理响应

以 JSON 格式返回 API 响应。以下 JSON 数据显示拼写错误的文本的响应消息 `Go shappin tommorow`：

```
{
   "_type":"SpellCheck",
   "flaggedTokens":[
      {
         "offset":3,
         "token":"shappin",
         "type":"UnknownToken",
         "suggestions":[
            {
               "suggestion":"shopping",
               "score":1
            }
         ]
      },
      {
         "offset":11,
         "token":"tommorow",
         "type":"UnknownToken",
         "suggestions":[
            {
               "suggestion":"tomorrow",
               "score":1
            }
         ]
      }
   ],
   "correctionType":"High"
}
```

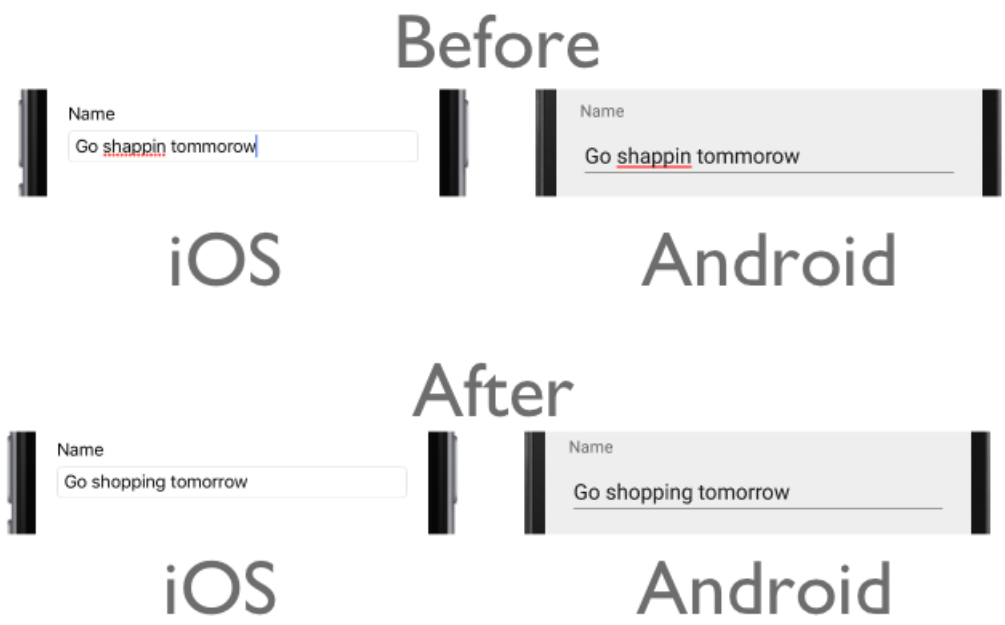
`flaggedTokens` 数组包含的文本中已标记为正在拼写不正确或其语法不正确的单词数组。如果找不到任何拼写或语法错误，则数组将不可用。在该数组中的标记是：

- `offset` – 从零开始的偏移量从文本字符串的开头已标记的单词。
- `token` – 拼写不正确或语法不正确的文本字符串中的单词。
- `type` – 导致将标记的单词的错误的类型。有两个可能值 – `RepeatedToken` 和 `UnknownToken` 。
- `suggestions` – 将更正拼写或语法错误的单词的数组。数组组成 `suggestion` 和 `score` ，这指示的建议的更正正确无误的置信度级别。

在示例应用程序，则 JSON 响应反序列化为 `SpellCheckResult` 实例，而且要返回到调用方法以显示的结果。下面的代码示例演示如何 `SpellCheckResult` 实例处理用于显示：

```
var spellCheckResult = await bingSpellCheckService.SpellCheckTextAsync(TodoItem.Name);
foreach (var flaggedToken in spellCheckResult.FlaggedTokens)
{
  TodoItem.Name = TodoItem.Name.Replace(flaggedToken.Token,
flaggedToken.Suggestions.FirstOrDefault().Suggestion);
}
```

此代码循环访问 `FlaggedTokens` 集合和替换任何拼写错误或与第一个建议的源文本中的语法不正确单词。以下屏幕截图显示之前和之后的拼写检查：



总结

本文介绍了如何使用必应拼写检查 REST API 更正 Xamarin.Forms 应用程序中的拼写错误。必应拼写检查执行上下文的拼写检查的文本，提供有关拼写错误的单词的内联建议。

相关链接

- 必应拼写检查文档
- 使用 rest 样式 Web 服务
- Todo 认知服务（示例）
- 必应拼写检查 API v7 参考

使用转换器 API 的文本转换

2018/6/22 • • Edit Online

Microsoft 转换器 API 可用来翻译语音和通过 REST API 的文本。此文章介绍了如何使用 Microsoft 转换器文本 API 转换从一种语言到另一个在 Xamarin.Forms 应用程序中的文本。

概述

转换器 API 具有两个组件：

- 文本转换 REST API 将从一种语言的文本转换的另一种语言的文本。API 会自动检测已将其转换之前发送的文本的语言。
- 语音转换成另一种语言的文本，理赔从一种语言的语音 REST API。API 还集成进行沟通返回已翻译的文本的文本到语音转换功能。

本文重点介绍在转换到另一个使用转换器文本 API 从一种语言的文本。

若要使用转换器文本 API，必须获取 API 密钥。这可以在获取如何注册 Microsoft 转换器文本 API。

有关 Microsoft 转换器文本 API 的详细信息，请参阅转换器文本 API 文档。

身份验证

每个请求都会对转换器文本 API 需要一个 JSON Web 令牌 (JWT) 访问令牌，可以从在认知服务令牌服务获取 `https://api.cognitive.microsoft.com/sts/v1.0/issueToken`。可以通过 POST 请求的令牌的服务，获得令牌指定 `Ocp-Apim-Subscription-Key` 标头，其中包含作为其值的 API 密钥。

下面的代码示例说明如何请求访问令牌从令牌的服务：

```
public AuthenticationService(string apiKey)
{
    subscriptionKey = apiKey;
    httpClient = new HttpClient();
    httpClient.DefaultRequestHeaders.Add("Ocp-Apim-Subscription-Key", apiKey);
}
...
async Task<string> FetchTokenAsync(string fetchUri)
{
    UriBuilder uriBuilder = new UriBuilder(fetchUri);
    uriBuilder.Path += "/issueToken";
    var result = await httpClient.PostAsync(uriBuilder.Uri.AbsoluteUri, null);
    return await result.Content.ReadAsStringAsync();
}
```

返回的访问令牌，它是 Base64 文本，具有 10 分钟的到期时间。因此，示例应用程序将续订的访问令牌每隔 9 分钟。

必须在每个转换器文本 API 中指定的访问令牌作为调用 `Authorization` 使用字符串作为前缀的标头 `Bearer`，下面的代码示例中所示：

```
httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);
```

认知服务令牌服务有关的详细信息，请参阅身份验证令牌 API。

执行文本转换

文本转换可以通过发出 GET 请求到 `translate` API 在 `https://api.microsofttranslator.com/v2/http.svc/translate`。 在示例应用程序，`TranslateTextAsync` 方法调用文本转换过程：

```
public async Task<string> TranslateTextAsync(string text)
{
  ...
  string requestUri = GenerateRequestUri(Constants.TextTranslatorEndpoint, text, "en", "de");
  string accessToken = authenticationService.GetAccessToken();
  var response = await SendRequestAsync(requestUri, accessToken);
  var xml = XDocument.Parse(response);
  return xml.Root.Value;
}
```

`TranslateTextAsync` 方法生成请求 URI，并从令牌的服务中检索访问令牌。然后，文本转换请求将发送到 `translate` API，后者将返回 XML 响应包含结果。解析 XML 响应，并转换结果返回到调用方法以显示。

有关文本转换 REST Api 的详细信息，请参阅Microsoft 转换器文本 API。

配置文本转换

通过指定 HTTP 查询参数可以将文本转换过程：

```
string GenerateRequestUri(string endpoint, string text, string to)
{
  string requestUri = endpoint;
  requestUri += string.Format("?text={0}", Uri.EscapeUriString(text));
  requestUri += string.Format("&to={0}", to);
  return requestUri;
}
```

此方法设置要转换的文本和要转换到的文本的语言。有关支持 Microsoft Translator 的语言的列表，请参阅Microsoft 转换器文本 API 中支持的语言。

> **NOTE**
> 如果应用程序需要哪种语言的文本是在中， `Detect` API 可以调用以进行检测的文本字符串的语言。

发送请求

`SendRequestAsync` 方法向文本转换 REST API 发出 GET 请求，并返回响应：

```
async Task<string> SendRequestAsync(string url, string bearerToken)
{
    if (httpClient == null)
    {
        httpClient = new HttpClient();
    }
    httpClient.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", bearerToken);

    var response = await httpClient.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}
```

此方法可通过添加到的访问令牌生成 GET 请求 `Authorization` 标头，使用字符串作为前缀 `Bearer`。然后将 GET 请求发送到 `translate` API，在使用请求 URL 指定的文本要转换，以及要转换到的文本的语言。然后将对响应进行读取，并将其返回到调用方法。

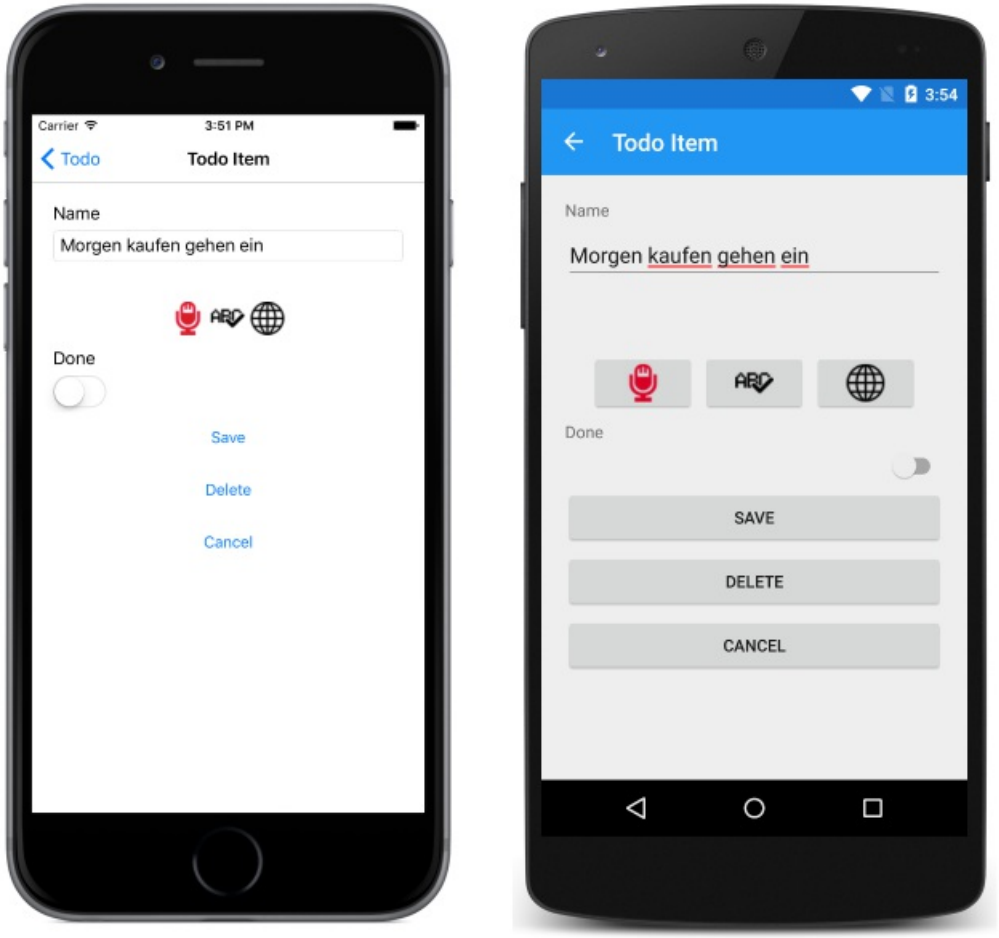
`translate` API 将在响应中，提供该请求是有效的表示请求成功，请求的信息包含在响应中发送 HTTP 状态代码 200（正常）。有关可能的错误响应的列表，请参阅在响应消息获取转换。

处理响应

API 响应以 XML 格式返回。下面的 XML 数据显示了典型成功的响应消息：

```
<string xmlns="http://schemas.microsoft.com/2003/10/Serialization/">Morgen kaufen gehen ein</string>
```

在示例应用程序，XML 响应解析为 `XDocument` 实例，而且正在返回到调用方法以显示以下屏幕截图中所示的 XML 根值：



总结

本文介绍了如何使用 Microsoft 转换器文本 API 将从一种语言的文本转换的 Xamarin.Forms 应用程序中的另一种语言的文本。除了转换文本时，Microsoft 转换器 API 可以还成另一种语言的文本，理赔从一种语言的语音。

相关链接

- 转换器文本 API 文档。
- 使用 rest 样式 Web 服务
- Todo 认知服务（示例）
- Microsoft Translator 文本 API。

表情识别使用面临的 API

2018/6/22 • • Edit Online

表面 API 在映像中作为输入，采用面部表达式并返回数据包含在映像中的每个表面情感一组之间的置信度级别。此文章介绍了如何使用表面 API 来识别表情，进行评级 Xamarin.Forms 应用程序。

概述

表面 API 可以执行表情检测，以检测愤怒、contempt、厌恶，不必担心、幸福、非特定、sadness 和意外情况发生，面部表情。这些情感普遍和多个区域性传送通过相同的基本面部表情。返回面部表情表情结果，以及表面 API 还可以返回检测到的表面的边界框。请注意，必须获取 API 密钥用于表面 API。这可以在获取重认知服务。

通过客户端库，并通过 REST API，可以执行表情识别。本文重点介绍在执行通过 REST API 的表情识别。有关 REST API 的详细信息，请参阅表面 REST API。

表面 API 还可以用来识别在视频中，人员的面部表达式，并且可以返回其情感的摘要。有关详细信息，请参阅如何实时分析视频。

有关表面 API 的详细信息，请参阅表面 API。

身份验证

每个请求都会面临 api 需要 API 密钥应指定的值为 `Ocp-Apim-Subscription-Key` 标头。下面的代码示例演示如何添加到 API 密钥 `Ocp-Apim-Subscription-Key` 请求标头：

```
public FaceRecognitionService()
{
  _client = new HttpClient();
  _client.DefaultRequestHeaders.Add("ocp-apim-subscription-key", Constants.FaceApiKey);
}
```

未能将有效的 API 密钥传递给表面 API 将导致 401 响应错误。

执行表情识别

通过使包含图像的 POST 请求执行表情识别 `detect` API 在 `https://[location].api.cognitive.microsoft.com/face/v1.0`，其中 `[location]` 是用于获取你的 API 密钥的区域。可选的请求参数包括：

- `returnFaceId` – 是否返回 faceIds 的检测到的平面。默认值为 `true`。
- `returnFaceLandmarks` – 是否返回表面界标的检测到的平面。默认值为 `false`。
- `returnFaceAttributes` – 是否分析，并返回一个或多个指定面临着属性。支持的表面属性包括 `age`，`gender`，`headPose`，`smile`，`facialHair`，`glasses`，`emotion`，`hair`，`makeup`，`occlusion`，`accessories`，`blur`，`exposure`，和 `noise`。请注意，表面属性分析计算和时间会增加成本。

必须为 URL 或二进制数据的 POST 请求的正文中放置图像内容。

> **NOTE**
> 支持的图像文件格式为 JPEG、PNG、GIF、和 BMP，和允许的文件大小为 1 KB 到 4 MB。

在示例应用程序，通过调用调用表情识别过程 `DetectAsync` 方法：

```
Face[] faces = await _faceRecognitionService.DetectAsync(photoStream, true, false, new FaceAttributeType[] {
FaceAttributeType.Emotion });
```

此方法调用指定包含的图像数据，应返回 faceIds，，不应返回表面界标，并且应分析的映像表情的流。它还指定，将数组的形式返回结果 `Face` 对象。反过来，`DetectAsync` 方法调用 `detect` 执行表情识别的 REST API：

```
public async Task<Face[]> DetectAsync(Stream imageStream, bool returnFaceId, bool returnFaceLandmarks,
IEnumerable<FaceAttributeType> returnFaceAttributes)
{
  var requestUrl =
    $"{Constants.FaceEndpoint}/detect?returnFaceId={returnFaceId}" +
    "&returnFaceLandmarks={returnFaceLandmarks}" +
    "&returnFaceAttributes={GetAttributeString(returnFaceAttributes)}";
  return await SendRequestAsync<Stream, Face[]>(HttpMethod.Post, requestUrl, imageStream);
}
```

此方法生成请求 URI，然后将请求发送到 `detect` API 通过 `SendRequestAsync` 方法。

> **NOTE**
> 在为你用于获取你的订阅密钥你面临 API 调用中，必须使用同一区域。例如，如果你获得你的订阅密钥从 `westus` 区域，表面检测终结点将是 `https://westus.api.cognitive.microsoft.com/face/v1.0/detect`。

发送请求

`SendRequestAsync` 方法向表面 API 发出的 POST 请求，并将结果作为返回 `Face` 数组：

```
async Task<TResponse> SendRequestAsync<TRequest, TResponse>(HttpMethod httpMethod, string requestUrl, TRequest
requestBody)
{
  var request = new HttpRequestMessage(httpMethod, Constants.FaceEndpoint);
  request.RequestUri = new Uri(requestUrl);
  if (requestBody != null)
  {
    if (requestBody is Stream)
    {
      request.Content = new StreamContent(requestBody as Stream);
      request.Content.Headers.ContentType = new MediaTypeHeaderValue("application/octet-stream");
    }
    else
    {
      // If the image is supplied via a URL
      request.Content = new StringContent(JsonConvert.SerializeObject(requestBody, s_settings), Encoding.UTF8,
"application/json");
    }
  }

  HttpResponseMessage responseMessage = await _client.SendAsync(request);
  if (responseMessage.IsSuccessStatusCode)
  {
    string responseContent = null;
    if (responseMessage.Content != null)
    {
      responseContent = await responseMessage.Content.ReadAsStringAsync();
    }
    if (!string.IsNullOrWhiteSpace(responseContent))
    {
      return JsonConvert.DeserializeObject<TResponse>(responseContent, s_settings);
    }
    return default(TResponse);
  }
  else
  {
    ...
  }
  return default(TResponse);
}
```

如果通过流提供的映像，则该方法通过包装中的图像流来生成 POST 请求 `StreamContent` 实例，它提供基于流的 HTTP 内容。或者，如果通过 URL 提供的映像，则该方法将生成 POST 请求通过包装中的 URL 来 `StringContent` 实例，它提供基于字符串的 HTTP 内容。

然后将 POST 请求发送到 `detect` API。读取响应，将其反序列化，并返回到调用方法。

`detect` API 将在响应中，提供该请求是有效的表示请求成功，请求的信息包含在响应中发送 HTTP 状态代码 200（正常）。有关可能的错误响应的列表，请参阅表面 REST API。

处理响应

以 JSON 格式返回 API 响应。以下 JSON 数据显示了一个典型的成功响应消息，提供所请求的示例应用程序数据：

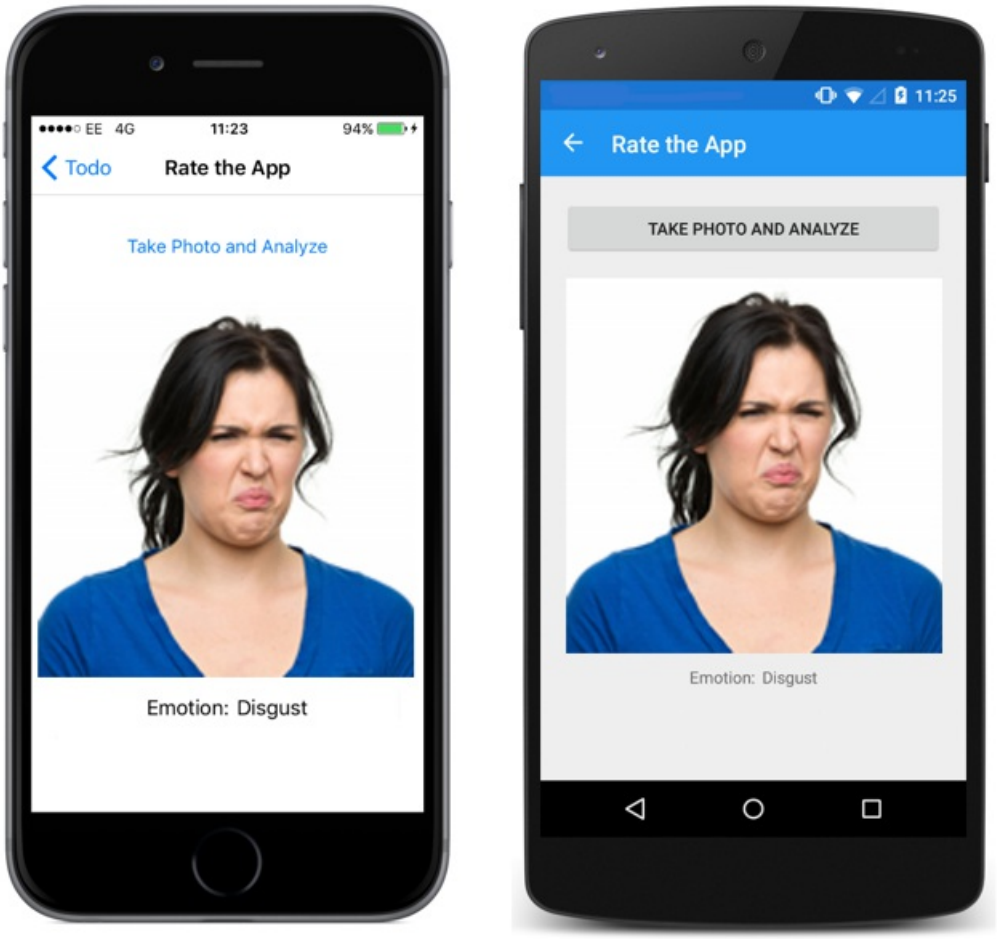
```
[
   {
      "faceId":"8a1a80fe-1027-48cf-a7f0-e61c0f005051",
      "faceRectangle":{
         "top":192,
         "left":164,
         "width":339,
         "height":339
      },
      "faceAttributes":{
         "emotion":{
            "anger":0.0,
            "contempt":0.0,
            "disgust":0.0,
            "fear":0.0,
            "happiness":1.0,
            "neutral":0.0,
            "sadness":0.0,
            "surprise":0.0
         }
      }
   }
]
```


成功的响应消息包含的按降序排序，一个空响应指示检测到没有表面表面矩形大小排名的表面项数组。每个识别表面包含一系列由指定的可选字体特性的 `returnFaceAttributes` 参数 `DetectAsync` 方法。

在示例应用程序，则 JSON 响应进行反序列化到的数组 `Face` 对象。解释结果从表面 API，当检测到的表情应被视为具有最高的得分，表情如评分被规范化到其中一个的总和。因此，示例应用程序将显示与面临的最大检测到的最高分数识别的表情图像中。这是替换为以下代码来实现的：

```
emotionResultLabel.Text = faces.FirstOrDefault().FaceAttributes.Emotion.ToRankedList().FirstOrDefault().Key;
```

下面的屏幕截图显示了示例应用程序中的表情识别过程的结果：



总结

本文介绍了如何使用表面 API 来识别表情，进行评级 Xamarin.Forms 应用程序。表面 API 在映像中作为输入，采用面部表达式并返回包括置信度在映像中的每个表面情感一组之间的数据。

相关链接

- 不会遇到 API。
- Todo 认知服务（示例）
- 表面 REST API

Xamarin.Forms 部署和测试

2018/6/27 • • Edit Online

性能

可以通过许多方法来提高使用 Xamarin.Forms 应用的性能。这些方法共同可以极大地降低由 CPU 执行的工作量和应用程序占用的内存量。

使用 Xamarin.UITest 和 App Center 自动测试

Xamarin 测试云的 **UITest** 组件可以与 Xamarin.Forms 一起使用，以编写可在数百个设备上的云中运行的 UI 测试。

Xamarin.Forms 性能

2018/7/13 • • Edit Online

可以通过许多方法来提高使用 Xamarin.Forms 应用程序的性能。这些方法共同可以极大地降低由 CPU 执行的工作量和应用程序占用的内存量。本文将介绍并讨论这些方法。

Evolve 2016：使用 **Xamarin.Forms** 优化应用性能

概述

应用程序性能差表现在许多方面。这会造成应用程序看起来无响应，导致滚动缓慢，还可降低电池寿命。但是，优化性能不止需要实现高效的代码。还必须考虑用户对应用程序性能的体验。例如，确保操作执行不会妨碍用户执行其他活动，这有助于改进用户的体验。

可以通过许多方法来提高 Xamarin.Forms 应用程序的性能和感知性能。它们包括：

- 启用 XAML 编译器
- 选择正确布局
- 启用布局压缩
- 使用快速呈现器
- 减少不需要的绑定
- 优化布局性能
- 优化 ListView 性能
- 优化图像资源
- 减小可视化树大小
- 减小应用程序资源字典大小
- 使用自定义呈现器模式

> **NOTE**
> 阅读本文之前，首先应阅读跨平台性能，其中讨论了非平台特定方法，可用于改善使用 Xamarin 平台生成的应用程序的内存使用情况和性能。

启用 XAML 编译器

XAML 可以根据需要使用 XAML 编译器 (XAMLC) 直接编译为中间语言 (IL)。XAMLC 提供了一些好处：

- 它会执行 XAML 的编译时检查，从而可向用户通知任何错误。
- 它会消除 XAML 元素的某些负载和实例化时间。
- 它通过不再包含 .xaml 文件，来帮助减小最终程序集的文件大小。

XAMLC 在默认情况下处于禁用状态，以便确保后向兼容性。但是，它可以在程序集和类级别进行启用。有关详细信息，请参阅编译 XAML。

选择正确布局

能够显示多个子级，但只具有单个子级的布局会比较浪费。例如，下面的代码示例演示一个具有单个子级的 `StackLayout`：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DisplayImage.HomePage">
    <ContentPage.Content>
        <StackLayout>
            <Image Source="waterfront.jpg" />
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

这比较浪费，应删除 `StackLayout` 元素，如下面的代码示例中所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="DisplayImage.HomePage">
    <ContentPage.Content>
        <Image Source="waterfront.jpg" />
    </ContentPage.Content>
</ContentPage>
```

此外，不要尝试使用其他布局的组合来重现特定布局的外观，因为这会导致执行不需要的布局计算。例如，不要尝试使用 `StackLayout` 实例的组合来重现 `Grid` 布局。下面的代码示例演示了这种错误做法的示例：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Details.HomePage"
             Padding="0,20,0,0">
    <ContentPage.Content>
        <StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Name:" />
                <Entry Placeholder="Enter your name" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Age:" />
                <Entry Placeholder="Enter your age" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Occupation:" />
                <Entry Placeholder="Enter your occupation" />
            </StackLayout>
            <StackLayout Orientation="Horizontal">
                <Label Text="Address:" />
                <Entry Placeholder="Enter your address" />
            </StackLayout>
        </StackLayout>
    </ContentPage.Content>
</ContentPage>
```

这比较浪费，因为会执行不需要的布局计算。相反，可以使用 `Grid` 更好地实现所需布局，如下面的代码示例所示：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Details.HomePage"
             Padding="0,20,0,0">
    <ContentPage.Content>
        <Grid>
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="100" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <Grid.RowDefinitions>
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
                <RowDefinition Height="30" />
            </Grid.RowDefinitions>
            <Label Text="Name:" />
            <Entry Grid.Column="1" Placeholder="Enter your name" />
            <Label Grid.Row="1" Text="Age:" />
            <Entry Grid.Row="1" Grid.Column="1" Placeholder="Enter your age" />
            <Label Grid.Row="2" Text="Occupation:" />
            <Entry Grid.Row="2" Grid.Column="1" Placeholder="Enter your occupation" />
            <Label Grid.Row="3" Text="Address:" />
            <Entry Grid.Row="3" Grid.Column="1" Placeholder="Enter your address" />
        </Grid>
    </ContentPage.Content>
</ContentPage>
```

启用布局压缩

为了提升页面呈现性能，布局压缩从可视化树中删除指定的布局。这带来的性能优势因页面复杂性、要使用的操作系统版本以及运行应用的设备而异。不过，在旧设备上实现的性能提升最大。有关详细信息，请参阅布局压缩。

使用快速呈现器

快速呈现器让生成的原生控件层次结构平展化，减少了 Android 上 Xamarin.Forms 控件的通货膨胀和呈现成本。这样一来，创建的对象就变少了，相应地也降低了可视化树的复杂性和内存使用率，从而进一步提升了性能。有关详细信息，请参阅快速呈现器。

减少不需要的绑定

不要将绑定用于可以方便地进行静态设置的内容。绑定无需绑定的数据不会带来优势，因为绑定并不经济高效。例如，设置 `Button.Text = "Accept"` 的开销要低于将 `Button.Text` 绑定到值为“Accept”的 ViewModel `string` 属性。

优化布局性能

Xamarin.Forms 2 引入了一种经过优化的布局引擎，它可影响布局更新。若要获取最佳可能布局性能，请遵循以下准则：

- 通过指定 `Margin` 属性值来减少布局层次结构的深度，从而允许创建具有更少换行视图的布局。有关详细信息，请参阅边距和填充。
- 使用 `Grid` 时，尝试确保将尽可能少的行和列设置为 `Auto` 大小。每个自动调整大小的行或列都会导致布局引擎执行额外布局计算。而是应在可能时使用固定大小的行和列。或者，使用 `GridUnitType.Star` 枚举值将行和列设置，为占据成比例的空间量，前提是父树遵循这些布局准则。
- 除非需要，否则不要设置布局的 `VerticalOptions` 和 `HorizontalOptions` 属性。`LayoutOptions.Fill` 和 `LayoutOptions.FillAndExpand` 的默认值可以实现最佳布局优化。更改这些属性会产生成本并消耗内存，即使是将它们设置为默认值。

- 尽可能避免使用 `RelativeLayout`。它会导致 CPU 不得不执行显著更多的工作。
- 使用 `AbsoluteLayout` 时，尽可能避免使用 `AbsoluteLayout.AutoSize` 属性。
- 使用 `StackLayout` 时，确保只有一个子级设置为 `LayoutOptions.Expands`。此属性可确保指定子级会占用 `StackLayout` 可以向它提供的最大空间，而多次执行这些计算比较浪费。
- 不要调用 `Layout` 类的任何方法，因为它们会导致执行成本高昂的布局计算。相反，可能可以通过设置 `TranslationX` 和 `TranslationY` 属性来获取所需布局行为。或者，将 `Layout<View>` 类设为子类以实现所需布局行为。
- 不要比需要更频繁地更新任何 `Label` 实例，因为标签大小的更改可能会导致重新计算整个屏幕布局。
- 除非需要，否则不要设置 `Label.VerticalTextAlignment` 属性。
- 尽可能将任何 `Label` 实例的 `LineBreakMode` 都设置为 `NoWrap`。

优化 ListView 性能

使用 `ListView` 控件时，应对许多用户体验进行优化：

- **初始化** – 从创建控件时开始，到在屏幕上显示项时结束的时间间隔。
- **滚动** – 能够滚动列表，并确保 UI 不滞后于触控笔势。
- **交互**，用于添加、删除和选择项。

`ListView` 控件需要应用程序提供数据和单元格模板。实现此目标的方法会对该控件的性能产生很大影响。有关详细信息，请参阅 ListView 性能。

优化图像资源

显示图像资源可能会极大提高应用的内存占用量。因此，仅应在必要时创建图像，应用程序不再需要图像后应立即将其释放。例如，如果应用程序通过从流中读取其数据来显示图像，请确保仅当需要时才创建流，并确保在不再需要时释放流。可以通过在创建页面时或是在 `Page.Appearing` 事件触发时创建流，然后在 `Page.Disappearing` 事件触发时释放流，来实现此目标。

使用 `ImageSource.FromUri` 方法下载图像进行显示时，通过确保将 `UriImageSource.CachingEnabled` 属性设置为 `true`，来缓存下载的图像。有关详细信息，请参阅使用图像。

有关详细信息，请参阅优化图像资源。

减小可视化树大小

减少页面上的元素数可以更快呈现页面。可通过两种主要方法来实现此目标。第一种方法是隐藏不可见的元素。每个元素的 `IsVisible` 属性可确定该元素是否应属于可视化树的一部分。因此，如果某个元素因为隐藏在其他元素后面而不可见，则删除该元素，或将其 `IsVisible` 属性设置为 `false`。

第二种方法是删除不需要的元素。例如，下面的代码示例演示一个显示一系列 `Label` 元素的页面布局：

```
<ContentPage.Content>
    <StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Hello" />
        </StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Welcome to the App!" />
        </StackLayout>
        <StackLayout Padding="20,20,0,0">
            <Label Text="Downloading Data..." />
        </StackLayout>
    </StackLayout>
</ContentPage.Content>
```

可以使用减少的元素数来维持相同的页面布局，如下面的代码示例所示：

```
<ContentPage.Content>
  <StackLayout Padding="20,20,0,0" Spacing="25">
    <Label Text="Hello" />
    <Label Text="Welcome to the App!" />
    <Label Text="Downloading Data..." />
  </StackLayout>
</ContentPage.Content>
```

减小应用程序资源字典大小

在整个应用程序中使用的任何资源都应存储在应用程序的资源字典中以避免重复。这会有助于减少整个应用程序中必须进行分析的 XAML 量。下面的代码示例演示 `HeadingLabelStyle` 资源，它在应用程序范围内使用，因此在应用程序的资源字典中进行定义：

```
<Application xmlns="http://xamarin.com/schemas/2014/forms"
                xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Resources.App">
     <Application.Resources>
         <ResourceDictionary>
            <Style x:Key="HeadingLabelStyle" TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
         </ResourceDictionary>
     </Application.Resources>
</Application>
```

但是，特定于页面的 XAML 不应包含在应用的资源字典中，因为这些资源随后会在应用程序启动时（而不是页面需要时）进行分析。如果某个资源由不是启动页面的页面使用，则它应置于该页面的资源字典中，因此有助于减少在应用程序启动时分析的 XAML。下面的代码示例演示 `HeadingLabelStyle` 资源，它只位于单个页面上，因此在该页面的资源字典中进行定义：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             x:Class="Test.HomePage"
             Padding="0,20,0,0">
     <ContentPage.Resources>
          <ResourceDictionary>
            <Style x:Key="HeadingLabelStyle" TargetType="Label">
                <Setter Property="HorizontalOptions" Value="Center" />
                <Setter Property="FontSize" Value="Large" />
                <Setter Property="TextColor" Value="Red" />
            </Style>
        </ResourceDictionary>
     </ContentPage.Resources>
    <ContentPage.Content>
      ...
    </ContentPage.Content>
</ContentPage>
```

有关应用程序资源的详细信息，请参阅 `Working with Styles`。

使用自定义呈现器模式

大多数呈现器类会公开 `OnElementChanged` 方法，此方法会在创建 Xamarin.Forms 自定义控件时被调用以呈现相应

的本机控件。自定义呈现器类(在每个特定于平台的呈现器类中)随后会替代此方法，以实例化并自定义本机控件。`SetNativeControl` 方法用于实例化本机控件，此方法还会将控件引用分配给 `Control` 属性。

但是，在某些情况下，可以多次调用 `OnElementChanged` 方法。因此，为了防止内存泄漏(这可能会对性能产生影响)，在实例化新的本机控件时务必要格外小心。下面的代码示例中演示了在自定义呈现器中实例化新的本机控件时要使用的方法：

```
protected override void OnElementChanged (ElementChangedEventArgs<NativeListView> e)
{
  base.OnElementChanged (e);

  if (Control == null) {
    // Instantiate the native control
  }

  if (e.OldElement != null) {
    // Unsubscribe from event handlers and cleanup any resources
  }

  if (e.NewElement != null) {
    // Configure the control and subscribe to event handlers
  }
}
```

当 `Control` 属性是 `null` 时，新的本机控件只应实例化一次。仅当自定义呈现器附加到新 Xamarin.Forms 元素时，才应配置该控件并订阅事件处理程序。同样，仅当呈现器所附加到的元素更改时，才应取消订阅任何订阅的事件处理程序。采用此方法将有助于创建不会遭受内存泄漏的高效执行的自定义呈现器。

有关自定义呈现器的详细信息，请参阅在每个平台上自定义控件。

总结

本文介绍和讨论了提高 Xamarin.Forms 应用程序的性能的方法。这些方法共同可以极大地降低由 CPU 执行的工作量和应用程序占用的内存量。

相关链接

- 跨平台性能
- ListView 性能
- 快速呈现器
- 布局压缩
- Xamarin.Forms 图像调整器示例
- XamlCompilation
- XamlCompilationOptions

使用 App Center 自动执行 Xamarin.Forms 测试

2018/11/2 • • Edit Online

Xamarin UITest 组件可以与 Xamarin.Forms 一起使用，以编写可在数百个设备上的云中运行的 UI 测试。

概述

使用 **App Center Test**，开发人员可为 iOS 和 Android 应用编写自动化用户界面测试。经过一些细微调整，可使用 Xamarin.UITest 来测试 Xamarin.Forms 应用，包括共享相同的测试代码。本文介绍结合使用 Xamarin.UITest 和 Xamarin.Forms 的特定使用技巧。

本指南假定用户已具备一定的 Xamarin.UITest 知识。若要掌握 Xamarin.UITest 知识，建议遵循以下指南：

- App Center Test 简介
- UITest 简介

将 UITest 项目添加到 Xamarin.Forms 解决方案后，即可采用用于 Xamarin.Android 或 Xamarin.iOS 应用程序的相同步骤为 Xamarin.Forms 应用程序编写和运行测试。

要求

请参阅 Xamarin.UITest，确认项目是否可供执行自动 UI 测试。

将 UITest 支持添加到 Xamarin.Forms 应用

UITest 通过激活屏幕上的控件并对用户通常与应用程序交互的任何位置执行输入来自动化用户界面。若要通过按某一按钮或在框中输入文本启用测试，测试代码需要某种可识别屏幕上控件的方法。

若要启用引用控件的 UITest 代码，每个控件均需一个唯一标识符。在 Xamarin.Forms 中，建议使用 `AutomationId` 属性设置此标识符，如下所示：

```
var b = new Button {
    Text = "Click me",
    AutomationId = "MyButton"
};
var l = new Label {
    Text = "Hello, Xamarin.Forms!",
    AutomationId = "MyLabel"
};
```

还可在 XAML 中设置 `AutomationId` 属性：

```
<Button x:Name="b" AutomationId="MyButton" Text="Click me"/>
<Label x:Name="l" AutomationId="MyLabel" Text="Hello, Xamarin.Forms!" />
```

唯一 `AutomationId` 需添加到测试所需的所有控件（包括按钮、文本项和可能需要查询其值的标签）。

> **NOTE**
> 注意，如果多次尝试设置 `Element` 的 `AutomationId` 属性，将引发 `InvalidOperationException`。

iOS 应用程序项目

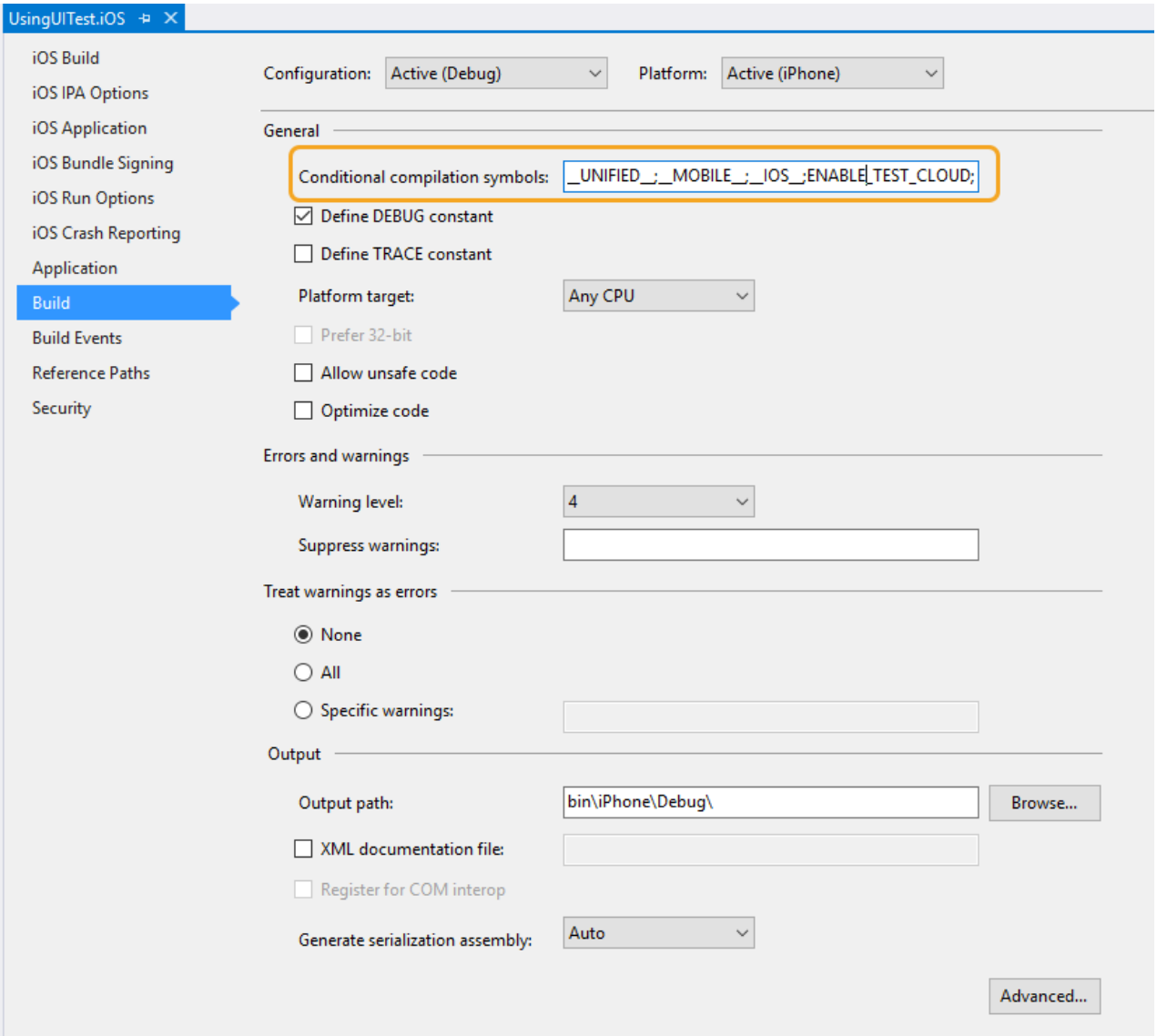
若要在 iOS 上运行测试，必须将 Xamarin Test Cloud Agent NuGet 包添加到项目。完成添加后，请将以下代码复制到 `AppDelegate.FinishedLaunching` 方法：

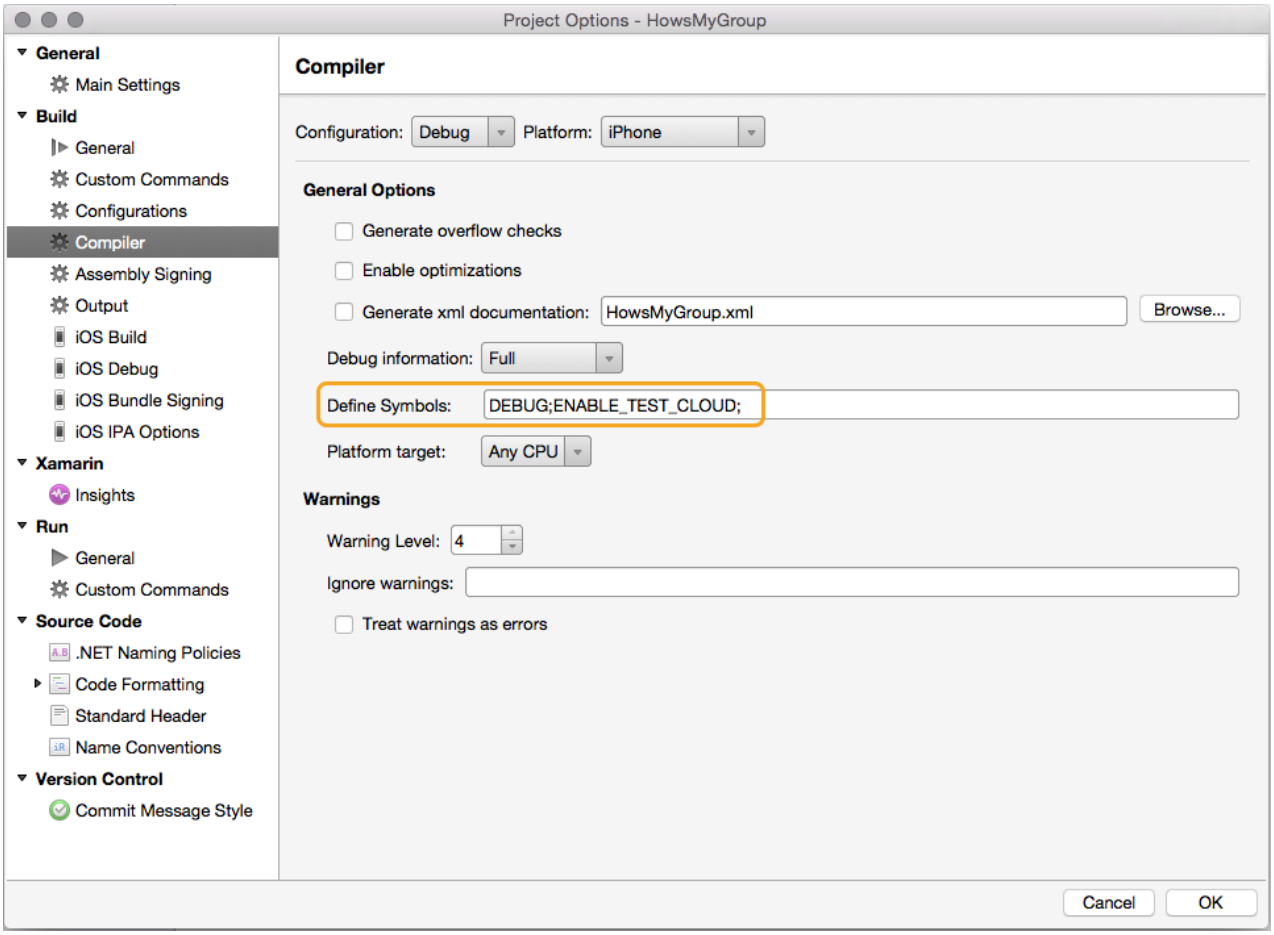
```
#if ENABLE_TEST_CLOUD
// requires Xamarin Test Cloud Agent
Xamarin.Calabash.Start();
#endif
```

Calabash 程序集使用非公开 Apple API，从而导致 App Store 拒绝应用。但是，如果未从代码中显式引用 Calabash 程序集，则 Xamarin.iOS 链接器会从最终 IPA 中将其删除。

> **NOTE**
> 发布版本不含 `ENABLE_TEST_CLOUD` 编译器变量，从而导致 Calabash 程序集从应用包中删除。但是，调试版本中定义了编译器指令，可防止链接器删除程序集。

以下屏幕截图显示有关调试版本的 `ENABLE_TEST_CLOUD` 编译器变量设置：





Android 应用程序项目

与 iOS 不同，Android 项目不需要任何特殊的启动代码。

编写 UITest

有关编写 UITest 的信息，请参阅 UITest 文档。以下概要步骤专门描述了如何生成 Xamarin.Forms 演示 **UsingUITest**。

使用 Xamarin.Forms UI 中的 AutomationId

编写任何 UITest 之前，必须确保 Xamarin.Forms 应用程序用户界面上的脚本可编辑。确保用户界面中的所有控件都有 `AutomationId`，以便在测试代码中引用它们。

在 UITest 中引用 AutomationId

编写 UITest 时，`AutomationId` 值会在每个平台上以不同方式公开：

- **iOS** 使用 `id` 字段。
- **Android** 使用 `label` 字段。

若要编写可在 iOS 和 Android 上找到 `AutomationId` 的跨平台 UITest，请使用 `Marked` 测试查询：

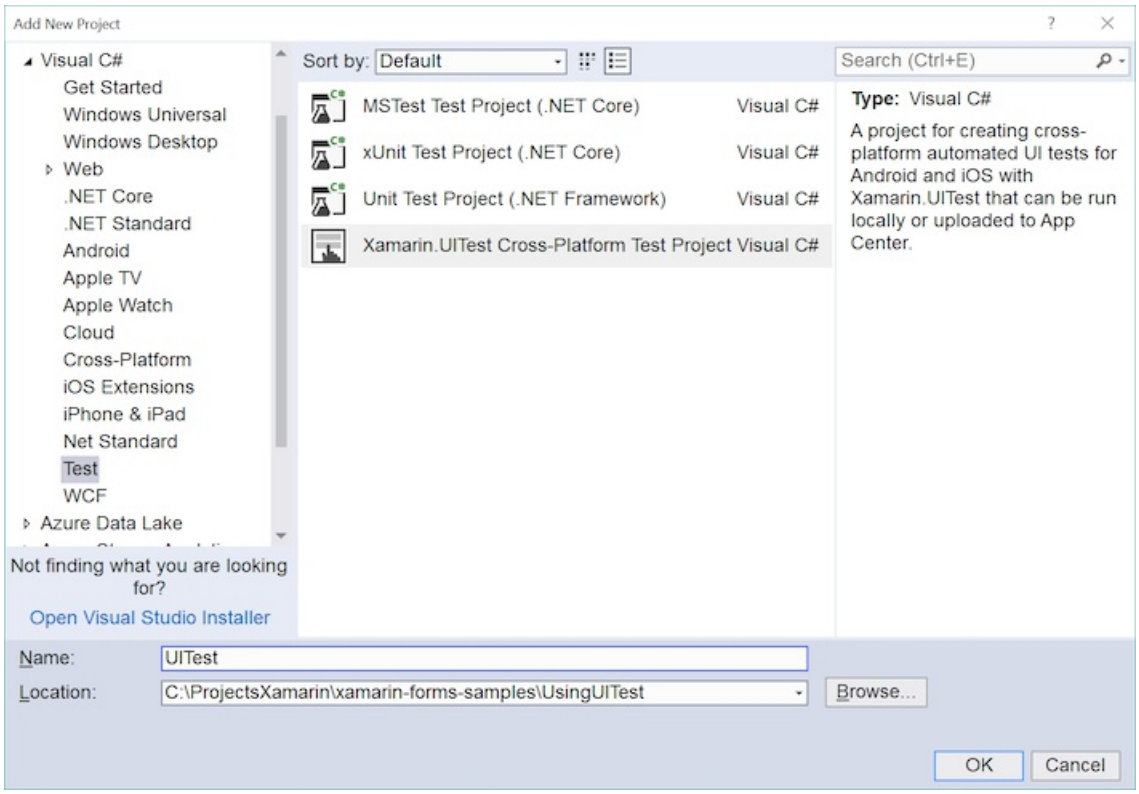
```
app.Query(c=>c.Marked("MyButton"))
```

短格式 `app.Query("MyButton")` 同样适用。

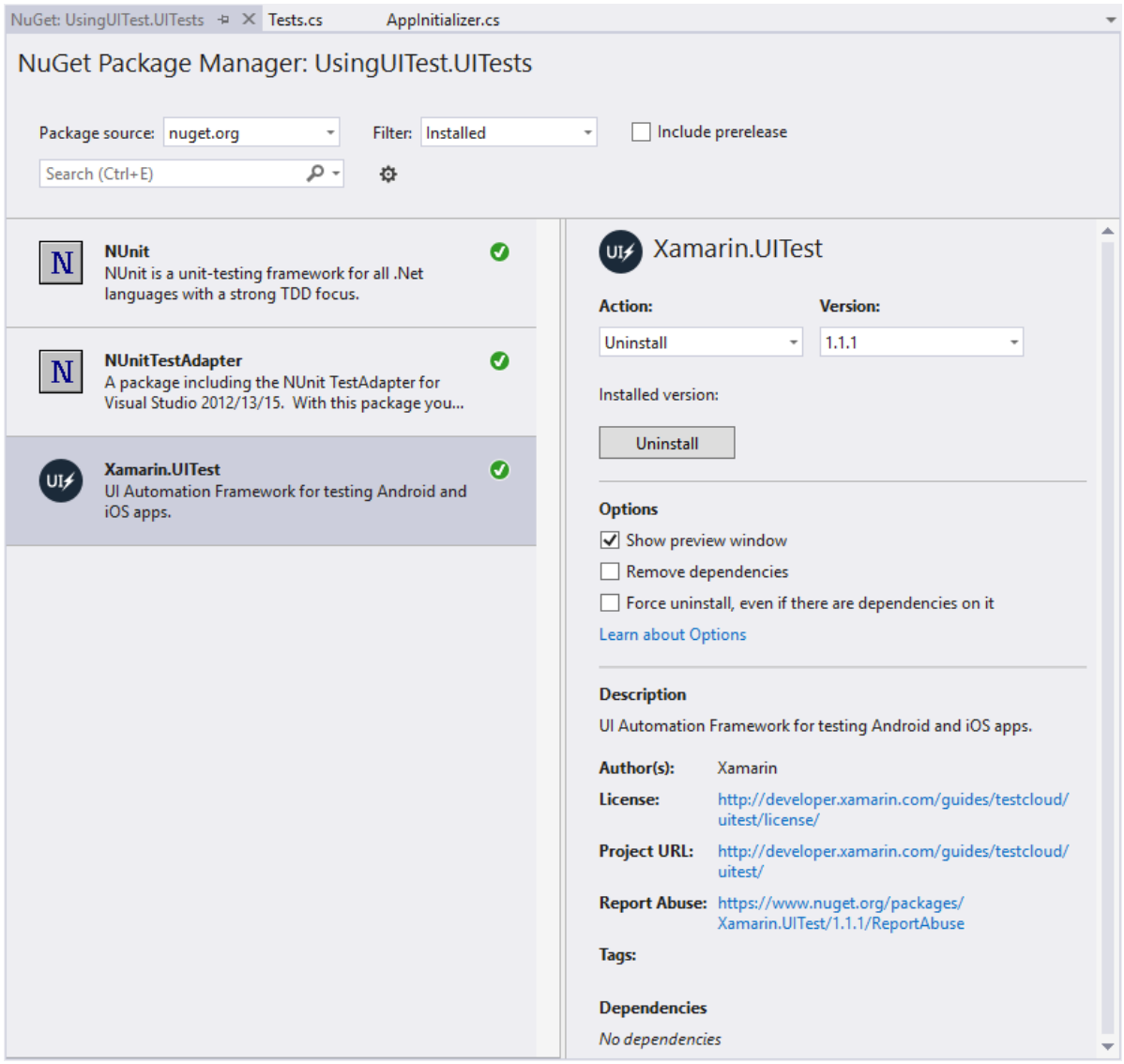
将 UITest 项目添加到现有解决方案

Visual Studio 提供了一个模板，帮助将 Xamarin.UITest 项目添加到现有 Xamarin.Forms 解决方案：

1. 右键单击解决方案，然后选择“文件”>“新建项目”。
2. 从 **Visual C#** 模板中，选择“测试”类别。选择“UI 测试应用”>“跨平台”模板：



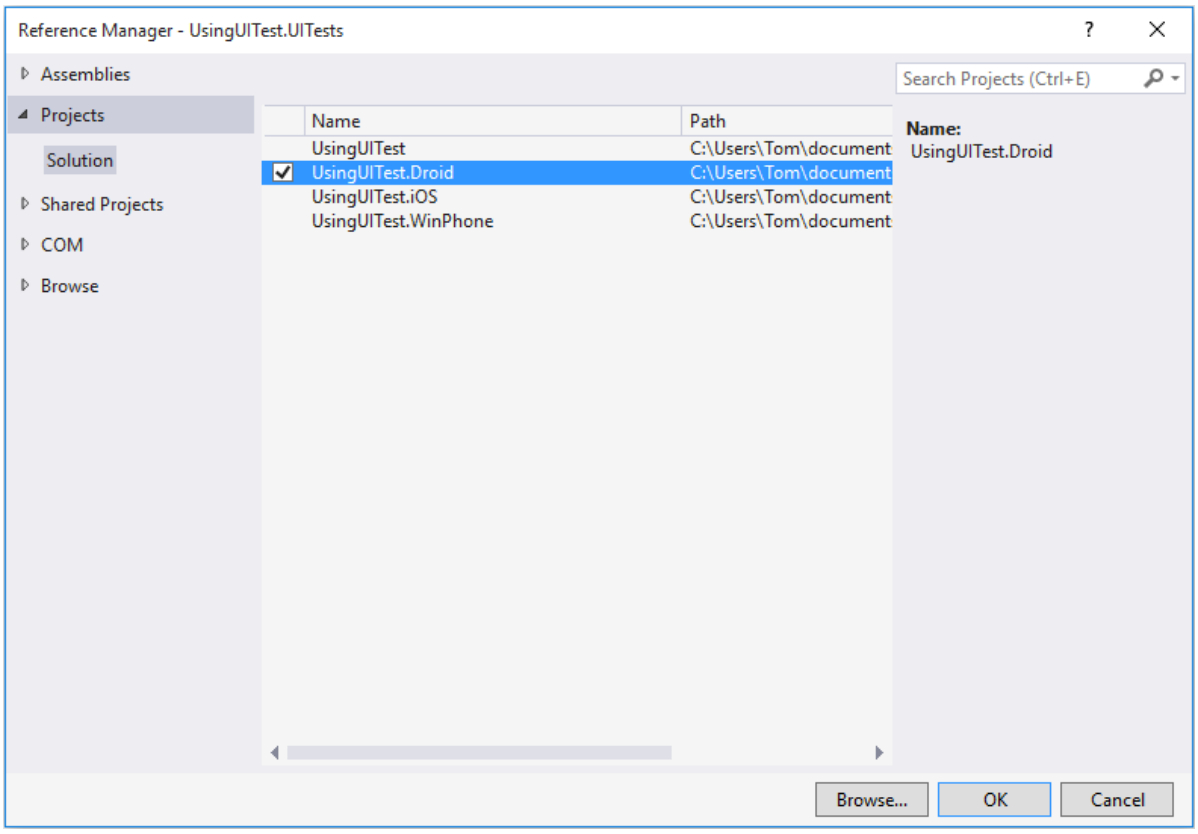
这会向解决方案添加一个包含 **NUnit**、**Xamarin.UITest** 和 **NUnitTestAdapter** NuGet 包的新项目：



NUnitTestAdapter 是一个第三方测试运行程序，允许 Visual Studio 从其自身运行 NUnit 测试。

新项目中还包含两个类。**AppInitializer** 类包含有助于初始化和设置测试的代码。另一个类 **Tests** 包含有助于启动 UITest 的 boilerplate 代码。

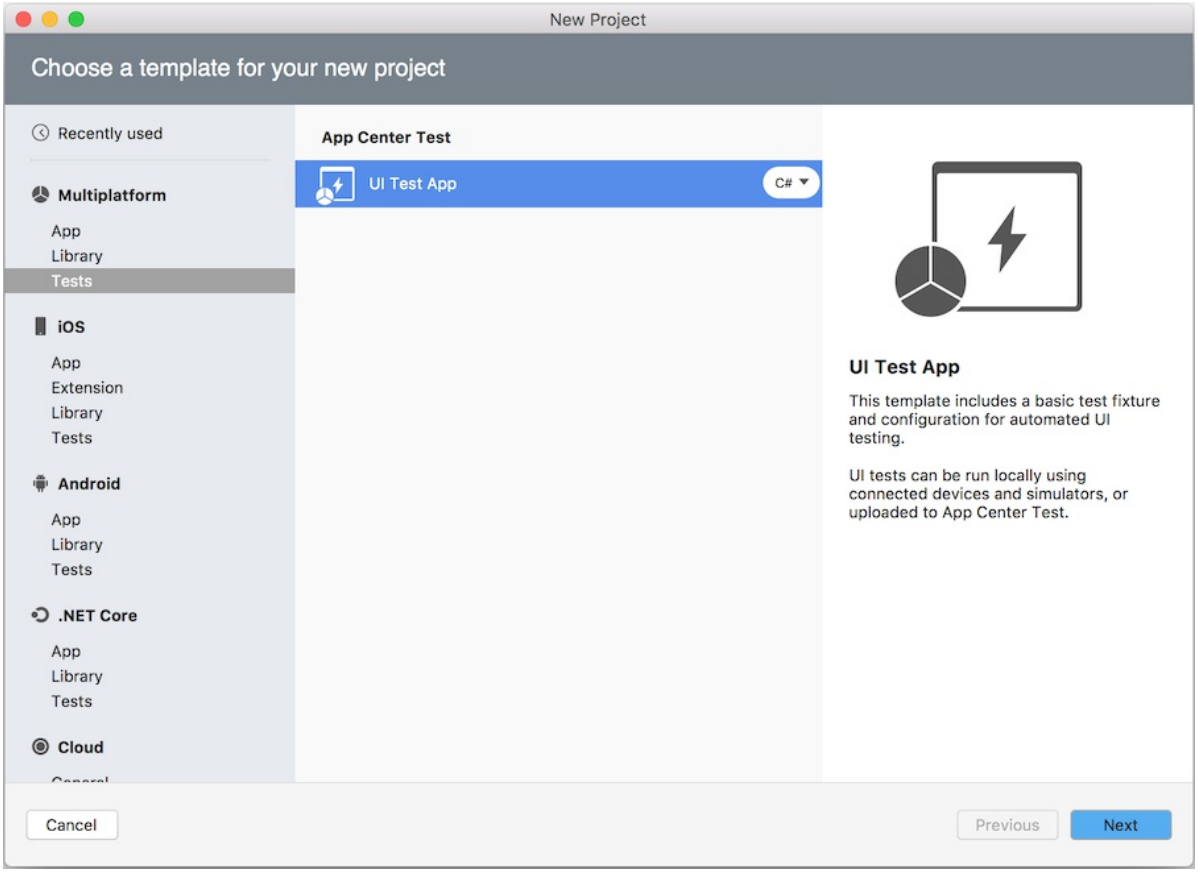
3. 将项目引用从 UITest 项目添加到 Xamarin.Android 项目：



这样做可以使 **NUnitTestAdapter** 对 Visual Studio 中的 Android 应用运行 UITest。

可手动将新 Xamarin.UITest 项目添加到现有解决方案：

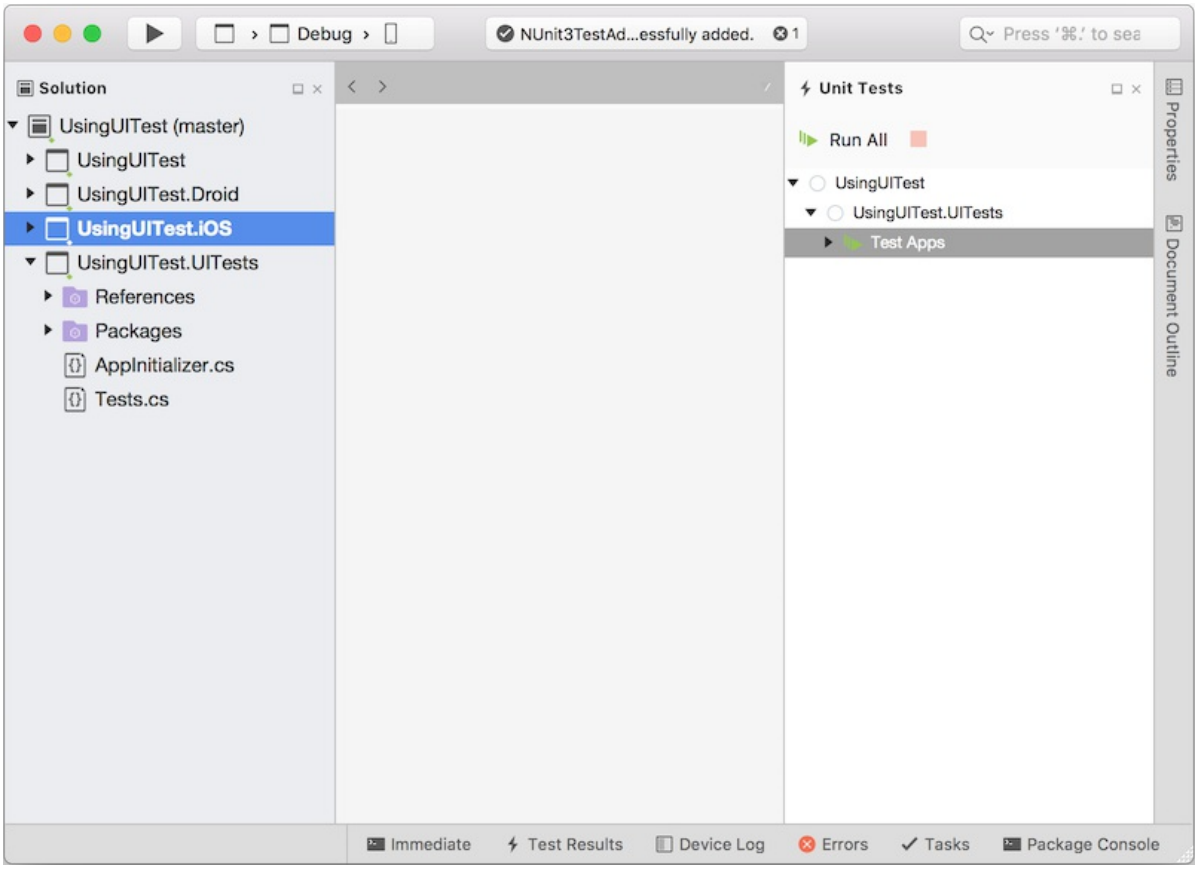
1. 首先添加新项目，方法是选择解决方案，然后单击“文件”>“添加新项目”。在“新建项目”对话框中，选择“跨平台”>“测试”>“Xamarin Test Cloud”>“UI 测试应用”：



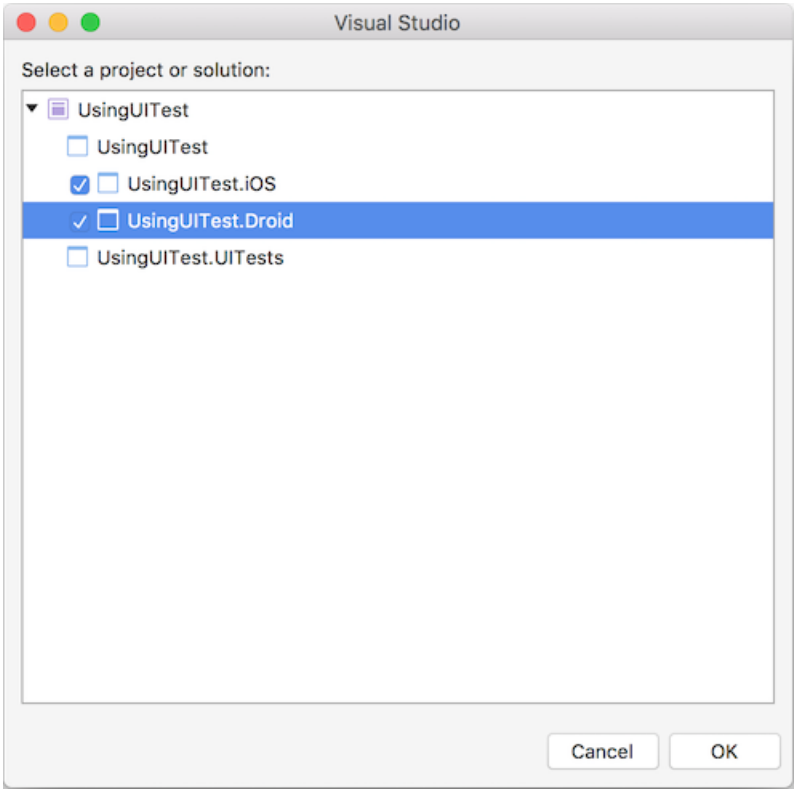
这将在解决方案中添加一个已包含 **NUnit** 和 **Xamarin.UITest** NuGet 包的新项目：

新项目中还包含两个类。**AppInitializer** 类包含有助于初始化和设置测试的代码。另一个类 **Tests** 包含有助于启动 UITest 的 boilerplate 代码。

2. 选择"视图">"面板">"单元测试"显示单元测试面板。展开"UsingUITest">"UsingUITest.UITests">"测试应用"：



3. 右键单击“测试应用”，再单击“添加应用项目”，然后在显示的对话框中选择 iOS 和 Android 项目：

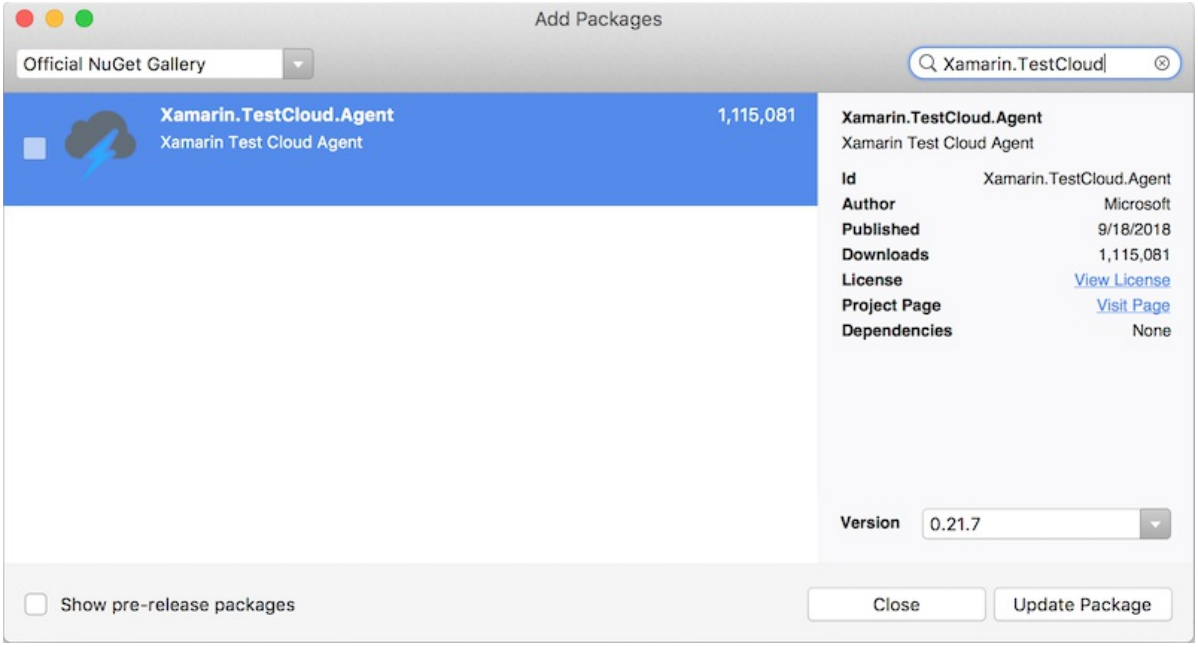


“单元测试”面板此时应有对 iOS 和 Android 项目的引用。这可使 Visual Studio for Mac 测试运行程序在本地对这两个 Xamarin.Forms 项目执行 UITest。

将 UITest 添加到 iOS 应用

需要先对 iOS 应用程序执行某些附加更改，Xamarin.UITest 才能正常工作：

1. 添加 **Xamarin Test Cloud Agent** NuGet 包。右键单击“包”，选择“添加包”，搜索 **Xamarin Test Cloud Agent** 的 NuGet，然后将其添加到 Xamarin.iOS 项目：



2. 编辑 **AppDelegate** 类的 `FinishedLaunching` 方法，以在 iOS 应用程序启动时初始化 Xamarin Test Cloud Agent，并设置视图的 `AutomationId` 属性。`FinishedLaunching` 方法应与以下代码示例类似：

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
    #if ENABLE_TEST_CLOUD
    Xamarin.Calabash.Start();
    #endif

    global::Xamarin.Forms.Forms.Init();

    LoadApplication(new App());

    return base.FinishedLaunching(app, options);
}
```

将 Xamarin.UITest 添加到 Xamarin.Forms 解决方案后，即可创建 UITest、在本地运行 UITest，并将其提交到 Xamarin Test Cloud。

总结

使用一种简单机制即可轻松通过 **Xamarin.UITest** 测试 Xamarin.Forms 应用程序，以将 `AutomationId` 公开为测试自动化的唯一视图标识符。将 UITest 项目添加到 Xamarin.Forms 解决方案后，即可采用用于 Xamarin.Android 或 Xamarin.iOS 应用程序的相同步骤为 Xamarin.Forms 应用程序编写和运行测试。

有关如何将测试提交到 App Center Test 的信息，请参阅提交 UITest。有关 UITest 的详细信息，请参阅 App Center Test 文档。

相关链接

- UITestSample
- Xamarin.Forms 示例
- 应用中心测试
- Xamarin.UITest
- NUnit

高级概念和内部机制

2018/7/31 • • Edit Online

快速呈现器

本文介绍了快速呈现器，减少的通货膨胀和呈现成本在 Android 上 Xamarin.Forms 控件通过平展生成的本机控件层次结构。

.NET Standard

本文介绍如何将转换为使用.NET Standard 2.0 的 Xamarin.Forms 应用程序。

依赖项解析

此文章介绍了如何将依赖项解析方法注入到 Xamarin.Forms，以便应用程序的依赖关系注入容器具有控制创建和自定义呈现器的效果，生存期和 `DependencyService` 实现。

Xamarin.Forms 快速呈现器

2018/10/25 • • Edit Online

本文介绍了快速呈现器，减少的通货膨胀和呈现成本在 Android 上 Xamarin.Forms 控件通过平展生成的本机控件层次结构。

一直以来，大部分在 Android 上的原始控件呈现器组成两个视图：

- 一个本机控件，如 `Button` 或 `TextView`。
- 容器 `ViewGroup` 用于处理某些布局工作、手势处理和其他任务。

但是，此方法具有性能暗示，为每个逻辑控件，这会导致更复杂的可视化树需要更多内存和处理，以在屏幕上呈现的详细信息创建两个视图。

快速呈现器减少到一个视图的通货膨胀和呈现成本的 Xamarin.Forms 控件。因此，而不被创建两个视图，并将它们添加到视图树，创建只有一个。通过创建更少的对象，这又意味着不太复杂的视图树，和更低的内存使用情况（这也会导致较少的垃圾回收暂停），这可以提高性能。

快速呈现器是可用于在 Android 上 Xamarin.Forms 2.4 中的以下控件：

- `Button`
- `Image`
- `Label`
- `Frame`

就功能而言，这些快速呈现器是对原始的呈现器没有什么不同。但是，当前处在试验阶段，仅可以通过添加以下代码行来使用你 `MainActivity` 类，然后调用 `Forms.Init`：

```
Forms.SetFlags("FastRenderers_Experimental");
```

> **NOTE**
> 快速呈现器只是适用于应用程序兼容性 Android 后端，因此将忽略此设置之前应用程序兼容性活动。

对于每个应用程序，取决于布局的复杂性而异的性能改进。例如，在滚动浏览时可能有性能提升了 x2 `ListView` 包含数千行数据，其中每个行中的单元格的使用快速呈现器的控件构成，这会导致以可视方式更顺畅地滚动。

相关链接

- 自定义呈现器

Xamarin.Forms 中的.NET 标准 2.0 支持

2018/7/19 • • Edit Online

本文介绍如何将转换为使用.NET 标准 2.0 Xamarin.Forms 应用程序。

.NET 标准是要用于在所有.NET 实现上可用的.NET Api 的规范。它使更轻松地在桌面应用程序、移动应用和游戏，间共享代码和云服务，将发送到不同的平台的相同的 Api。有关所支持的标准.NET 平台的信息，请参阅.NET 实现支持。

标准.NET 库是替换为可移植类库 (PCL)。但是，面向.NET 标准库仍然是 PCL 中，并且被称为基于.NET 标准的 PCL。某些 PCL 配置文件映射到.NET 标准版本，以及对于没有映射的配置文件，两个库类型将能够相互引用。有关详细信息，请参阅PCL 兼容性。

Xamarin.Forms 2.4 允许 Xamarin.Forms 应用程序迁移到目标.NET 标准 2.0 通过 PCL 替换.NET 标准 2.0 库。做到这一点，如下所示：

- 确保.NET Core 2.0安装。
- 更新 Xamarin.Forms 解决方案，以便使用 Xamarin.Forms 2.4，或更高版本。
- 将.NET 标准库添加到解决方案，面向.NET 标准 2.0。
- 删除添加到标准.NET 库的类。
- 将 Xamarin.Forms 2.4（或更高版本）NuGet 包添加到标准.NET 库。
- 在平台项目中，添加对标准.NET 库的引用并删除对包含 Xamarin.Forms 用户界面逻辑的 PCL 项目的引用。
- PCL 项目中的文件复制到标准.NET 库中。
- 删除包含的 Xamarin.Forms 用户界面逻辑的 PCL 项目。

相关链接

- .NET Standard
- 代码共享选项

在 Xamarin.Forms 中的依赖项解析

2018/7/31 • • Edit Online

此文章介绍了如何将依赖项解析方法注入到 Xamarin.Forms，以便应用程序的依赖关系注入容器具有控制创建和自定义呈现器、效果和 DependencyService 实现的生存期。在本文中的代码示例摘自使用容器的依赖项解析示例。

在上下文中使用模型-视图-视图模型 (MVVM) 模式的 Xamarin.Forms 应用程序，用于注册和解析视图模型，以及注册服务和将其注入到视图模型，可以使用依赖关系注入容器。在视图模型创建期间容器会注入任何所需的依存关系。如果尚未创建这些依赖项，该容器创建，并将首先解析依赖项。有关依赖关系注入，包括将依赖项注入到视图模型的示例的详细信息请参阅依赖关系注入。

对创建的控制和 Xamarin.Forms，它使用传统上执行平台项目中的类型的生存期 `Activator.CreateInstance` 方法创建的自定义呈现器的效果，实例和 `DependencyService` 实现。遗憾的是，这就限制了开发人员可以控制创建和生存期这些类型，并将依赖项注入到它们的功能。通过将依赖项解析方法注入到 Xamarin.Forms，用于控制如何将创建类型 – 通过应用程序的依赖关系注入容器，或通过 Xamarin.Forms，可以更改此行为。但是，请注意，没有无需将依赖项解析方法注入到 Xamarin.Forms。Xamarin.Forms 将继续创建和管理在平台项目中的类型的生存期，如果不注入依赖关系解析方法。

> **NOTE**
>
> 尽管本文着重介绍将依赖项解析方法注入到 Xamarin.Forms 解析已注册的类型使用依赖关系注入容器，还有可能注入使用工厂方法来解决的依赖关系解析方法已注册的类型。有关详细信息，请参阅使用工厂方法的依赖项解析示例。

注入依赖关系解析方法

`DependencyResolver` 类提供的功能将依赖项解析方法注入到 Xamarin.Forms 中，使用 `ResolveUsing` 方法。然后，当 Xamarin.Forms 需要特定类型的实例时，依赖关系解析方法都有机会提供该实例。如果依赖项解析方法返回 `null` 的请求的类型，Xamarin.Forms 回退到尝试创建类型实例本身使用 `Activator.CreateInstance` 方法。

下面的示例演示如何设置具有的依赖关系解析方法 `ResolveUsing` 方法：

```
using Autofac;
using Xamarin.Forms.Internals;
...

public partial class App : Application
{
    // IContainer and ContainerBuilder are provided by Autofac
    static IContainer container;
    static readonly ContainerBuilder builder = new ContainerBuilder();

    public App()
    {
        ...
        DependencyResolver.ResolveUsing(type => container.IsRegistered(type) ? container.Resolve(type) :
null);
        ...
    }
    ...
}
```

在此示例中，依赖关系解析方法设置为使用 Autofac 依赖关系注入容器解析已向容器注册的任何类型的 lambda 表达式。否则为 `null` 将返回，这将导致在 Xamarin.Forms 中尝试解析的类型。

> **NOTE**
>
> 通过依赖关系注入容器使用的 API 是特定于容器。在本文中的代码示例使用 Autofac 作为依赖关系注入容器，它提供 `IContainer` 和 `ContainerBuilder` 类型。替代依赖关系注入容器同样可用，但不是此处介绍会使用不同的 Api。

请注意，无需在应用程序启动过程中设置依赖关系解析方法。可以随时设置它。唯一约束是 Xamarin.Forms 需要了解的有关依赖关系解析方法时，应用程序尝试使用存储在依赖关系注入容器中的类型。因此，如果应用程序将需要在启动过程的依赖关系注入容器中的服务，将具有依赖关系解析方法要设置应用程序的生命周期中及早。同样，如果将依赖关系注入容器管理的创建和生存期的特定 `Effect`，Xamarin.Forms 将需要尝试创建视图之前了解有关依赖关系解析方法，使用该 `Effect`。

> **WARNING**
>
> 注册和解析依赖关系注入容器使用的类型具有性能成本由于反射的用于创建每种类型的容器的使用，尤其是依赖项会被重新构造，以便在应用程序中每个页面导航。如果有许多或深入的依赖项，则创建的成本会显著增加。

注册类型

它可以通过依赖关系解析方法解析它们之前，必须向依赖关系注入容器注册类型。下面的代码示例显示了注册方法的示例应用程序公开在 `App` 类，用于 Autofac 容器：

```
using Autofac;
using Autofac.Core;
...

public partial class App : Application
{
    static IContainer container;
    static readonly ContainerBuilder builder = new ContainerBuilder();
    ...

    public static void RegisterType<T>() where T : class
    {
        builder.RegisterType<T>();
    }

    public static void RegisterType<TInterface, T>() where TInterface : class where T : class, TInterface
    {
        builder.RegisterType<T>().As<TInterface>();
    }

    public static void RegisterTypeWithParameters<T>(Type param1Type, object param1Value, Type param2Type,
string param2Name) where T : class
    {
        builder.RegisterType<T>()
               .WithParameters(new List<Parameter>()
        {
            new TypedParameter(param1Type, param1Value),
            new ResolvedParameter(
                (pi, ctx) => pi.ParameterType == param2Type && pi.Name == param2Name,
                (pi, ctx) => ctx.Resolve(param2Type))
        });
    }

    public static void RegisterTypeWithParameters<TInterface, T>(Type param1Type, object param1Value, Type
param2Type, string param2Name) where TInterface : class where T : class, TInterface
    {
        builder.RegisterType<T>()
               .WithParameters(new List<Parameter>()
        {
            new TypedParameter(param1Type, param1Value),
            new ResolvedParameter(
                (pi, ctx) => pi.ParameterType == param2Type && pi.Name == param2Name,
                (pi, ctx) => ctx.Resolve(param2Type))
        }).As<TInterface>();
    }

    public static void BuildContainer()
    {
        container = builder.Build();
    }
    ...
}
```

当应用程序使用依赖关系解析方法来解决从容器的类型时，通常从平台项目中执行类型注册。这样，平台项目的自定义呈现器的效果，注册类型和 `DependencyService` 实现。

从平台项目，按照类型注册 `IContainer` 必须生成对象，这通过调用来实现 `BuildContainer` 方法。此方法将调用的 Autofac `Build` 方法 `ContainerBuilder` 实例，生成新的依赖关系注入容器，其中包含所做的注册。

以下各节，在 `Logger` 类，该类实现 `ILogger` 接口注入到类构造函数。 `Logger` 类实现简单的日志记录功能使用 `Debug.WriteLine` 方法，用于展示如何将服务注入到自定义呈现器的效果，并 `DependencyService` 实现。

注册自定义呈现器

示例应用程序包括播放 web 视频，其 XAML 源显示在下面的示例中的页面：

```
<ContentPage xmlns="http://xamarin.com/schemas/2014/forms"
             xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
             xmlns:video="clr-namespace:FormsVideoLibrary"
             ...>
    <video:VideoPlayer Source="https://archive.org/download/BigBuckBunny_328/BigBuckBunny_512kb.mp4" />
</ContentPage>
```

`VideoPlayer` 由每个平台上实现视图 `VideoPlayerRenderer` 类，提供用于播放视频的功能。有关这些自定义呈现器类的详细信息，请参阅实现视频播放器。

在 iOS 和通用 Windows 平台 (UWP) 上 `VideoPlayerRenderer` 类具有以下构造函数中，这要求 `ILogger` 参数：

```
public VideoPlayerRenderer(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

在所有三个平台上执行与依赖关系注入容器的类型注册 `RegisterTypes` 方法之前加载的应用程序使用的平台，调用 `LoadApplication(new App())` 方法。下面的示例演示 `RegisterTypes` iOS 平台上的方法：

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterType<FormsVideoLibrary.iOS.VideoPlayerRenderer>();
    App.BuildContainer();
}
```

在此示例中，`Logger` 具体类型注册通过对其接口类型，映射和 `VideoPlayerRenderer` 的接口映射不直接注册类型。当用户导航到包含的页 `VideoPlayer` 视图中，将调用的依赖关系解析方法要解决 `VideoPlayerRenderer` 类型从依赖关系注入容器，这还将解决并注入 `Logger` 键入 `VideoPlayerRenderer` 构造函数。

`VideoPlayerRenderer` Android 平台上的构造函数是稍微复杂一些，因为它需要 `Context` 除了参数 `ILogger` 参数：

```
public VideoPlayerRenderer(Context context, ILogger logger) : base(context)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

下面的示例演示 `RegisterTypes` Android 平台上的方法：

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterTypeWithParameters<FormsVideoLibrary.Droid.VideoPlayerRenderer>
(typeof(Android.Content.Context), this, typeof(ILogger), "logger");
    App.BuildContainer();
}
```

在此示例中，`App.RegisterTypeWithParameters` 方法注册 `VideoPlayerRenderer` 与依赖关系注入容器。此注册方法可确保 `MainActivity` 实例将作为注入 `Context` 自变量，并且 `Logger` 类型将作为注入 `ILogger` 参数。

注册效果

示例应用程序包括使用触摸屏输入跟踪效果拖动的页面 `BoxView` 页面周围的实例。`Effect` 添加到 `BoxView` 使用以下代码：

```
var boxView = new BoxView { ... };
var touchEffect = new TouchEffect();
boxView.Effects.Add(touchEffect);
```

`TouchEffect` 类是 `RoutingEffect` 由每个平台上实现 `TouchEffect` 类，该类具有 `PlatformEffect` 。在平台 `TouchEffect` 类提供的功能用于拖动 `BoxView` 围绕页。有关这些影响类的详细信息，请参阅调用影响来自事件。

在所有三个平台上 `TouchEffect` 类具有以下构造函数中，这要求 `ILogger` 参数：

```
public TouchEffect(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

在所有三个平台上执行与依赖关系注入容器的类型注册 `RegisterTypes` 方法之前加载的应用程序使用的平台，调用 `LoadApplication(new App())` 方法。下面的示例演示 `RegisterTypes` Android 平台上的方法：

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterType<TouchTracking.Droid.TouchEffect>();
    App.BuildContainer();
}
```

在此示例中，`Logger` 具体类型注册通过对其接口类型，映射和 `TouchEffect` 的接口映射不直接注册类型。当用户导航到包含的页 `BoxView` 具有实例 `TouchEffect` 附加到它，依赖关系解析调用该方法将解析平台 `TouchEffect` 从依赖关系的类型注入容器，它还将解决并注入 `Logger` 中键入 `TouchEffect` 构造函数。

注册 DependencyService 实现

示例应用程序包括使用的页面 `DependencyService` 实现在每个平台上以允许用户从设备的图片库中选取照片。 `IPhotoPicker` 接口定义的功能由实现 `DependencyService` 实现中，并在下面的示例所示：

```
public interface IPhotoPicker
{
    Task<Stream> GetImageStreamAsync();
}
```

在每个平台项目中，`PhotoPicker` 类实现 `IPhotoPicker` 使用平台 Api 的接口。有关这些依赖关系服务的详细信息，请参阅图片库从选取照片。

在 iOS 和 UWP `PhotoPicker` 类具有以下构造函数中，这要求 `ILogger` 参数：

```
public PhotoPicker(ILogger logger)
{
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

在所有三个平台上执行与依赖关系注入容器的类型注册 `RegisterTypes` 方法之前加载的应用程序使用的平台，调用 `LoadApplication(new App())` 方法。下面的示例演示 `RegisterTypes` UWP 上的方法：

```
void RegisterTypes()
{
    DIContainerDemo.App.RegisterType<ILogger, Logger>();
    DIContainerDemo.App.RegisterType<IPhotoPicker, Services.UWP.PhotoPicker>();
    DIContainerDemo.App.BuildContainer();
}
```

在此示例中，`Logger` 具体类型注册通过对其接口类型，映射和 `PhotoPicker` 类型还注册通过接口映射。

`PhotoPicker` Android 平台上的构造函数是稍微复杂一些，因为它需要 `Context` 除了参数 `ILogger` 参数：

```
public PhotoPicker(Context context, ILogger logger)
{
    _context = context ?? throw new ArgumentNullException(nameof(context));
    _logger = logger ?? throw new ArgumentNullException(nameof(logger));
}
```

下面的示例演示 `RegisterTypes` Android 平台上的方法：

```
void RegisterTypes()
{
    App.RegisterType<ILogger, Logger>();
    App.RegisterTypeWithParameters<IPhotoPicker, Services.Droid.PhotoPicker>(typeof(Android.Content.Context),
this, typeof(ILogger), "logger");
    App.BuildContainer();
}
```

在此示例中，`App.RegisterTypeWithParameters` 方法注册 `PhotoPicker` 与依赖关系注入容器。此注册方法可确保 `MainActivity` 实例将作为注入 `Context` 自变量，并且 `Logger` 类型将作为注入 `ILogger` 参数。

当用户导航到照片选择页上，并选择以选择照片，`OnSelectPhotoButtonClicked` 执行处理程序：

```
async void OnSelectPhotoButtonClicked(object sender, EventArgs e)
{
    ...
    var photoPickerService = DependencyService.Resolve<IPhotoPicker>();
    var stream = await photoPickerService.GetImageStreamAsync();
    if (stream != null)
    {
        image.Source = ImageSource.FromStream(() => stream);
    }
    ...
}
```

当 `DependencyService.Resolve<T>` 调用方法，将调用的依赖关系解析方法要解决 `PhotoPicker` 类型从依赖关系注入容器，这还将解决并注入 `Logger` 类型到 `PhotoPicker` 构造函数。

> **NOTE**
>
> `Resolve<T>` 解析通过应用程序的依赖关系注入容器中的类型时，必须使用方法 `DependencyService` 。

相关链接

- 使用容器（示例）的依赖项解析
- 依赖关系注入
- 实现视频播放器

- 调用效果中的事件
- 从图片库中选取照片

疑难解答

2018/6/22 • • Edit Online

常见错误情形和如何解决这些问题

错误:"找不到 Xamarin.Forms 的版本与兼容..."

以下的错误可以出现在**包控制台**窗口在更新所有 Nuget 程序包，Xamarin.Forms 解决方案或 Xamarin.Forms Android 应用程序项目中时：

```
Attempting to resolve dependency 'Xamarin.Android.Support.v7.AppCompat (= 23.3.0.0)'.
Attempting to resolve dependency 'Xamarin.Android.Support.v4 (= 23.3.0.0)'.
Looking for updates for 'Xamarin.Android.Support.v7.MediaRouter'...
Updating 'Xamarin.Android.Support.v7.MediaRouter' from version '23.3.0.0' to '23.3.1.0' in project
'Todo.Droid'.
Updating 'Xamarin.Android.Support.v7.MediaRouter 23.3.0.0' to 'Xamarin.Android.Support.v7.MediaRouter 23.3.1.0'
failed.
Unable to find a version of 'Xamarin.Forms' that is compatible with 'Xamarin.Android.Support.v7.MediaRouter
23.3.0.0'.
```

什么因素会导致此错误？

Visual Studio for Mac（或 Visual Studio）可能表示有可用的 Xamarin.Forms Nuget 包时更新*和所有依赖项*。在 Xamarin Studio，该**解决方案包**节点可能如下所示（的版本号可能不同）：

如果你尝试更新可能发生此错误_所有_包。

这是因为 android 项目设置为 Android 6.0 (API 23) 的目标/编译版本或 Xamarin.Forms 上具有硬依赖项的下面，*特定*版本的 Android 支持包。尽管这些包的更新的版本可用，Xamarin.Forms 不一定与它们兼容。

在这种情况下，你应该更新_仅_ **Xamarin.Forms**包因为这将确保依赖项保留在兼容的版本上。已添加到你的项目的其他包，只要它们不会导致 Android 支持程序包来更新也可能单独更新。

> **NOTE**
> 如果你使用 Xamarin.Forms 2.3.4 或更高版本和Android 项目的目标/编译版本设置为 Android 7.0 (API 24) 或更高版本，然后应用不再上面提到的硬依赖项并可能更新支持独立于 Xamarin.Forms 包的包。

修复：删除所有包，并重新添加 Xamarin.Forms

如果**Xamarin.Android.Support**包已更新为不兼容版本，最简单的解决方法是：

1. 然后手动删除在 Android 项目中，所有 Nuget 包
2. 重新添加**Xamarin.Forms**包。

这将自动下载*正确*的其他包版本。

若要查看此过程的视频，请参阅此论坛 post。

常见问题

2018/6/22 • • Edit Online

可否将 Xamarin.Forms 默认模板更新到较新的 NuGet 包？

本指南使用 Xamarin.Forms.NET 标准库模板作为示例，但相同的常规方法也适用于 Xamarin.Forms 共享项目模板。

为何 Visual Studio XAML 设计器对 Xamarin.Forms XAML 文件不起作用？

Xamarin.Forms 当前不支持的 XAML 文件的可视化设计器。

Android 生成错误："LinkAssemblies"任务意外失败

你可能会看到一条错误消息 `The "LinkAssemblies" task failed unexpectedly` 时生成，该服务 Xamarin.Android 项目使用窗体。链接器处于活动状态时将发生这种情况（通常在*版本*生成，以减少应用包的大小）；这是由于 Android 目标不更新为最新的框架和。

"为什么不会通过与 COMPILETODALVIK 我 Xamarin.Forms.Maps Android 项目：意外的顶级错误？"

适用于 Mac 或 Visual Studio；的生成输出窗口中，可能会在 Visual Studio 的错误小键盘出现此错误在 Android 项目中使用 Xamarin.Forms.Maps。它通常是通过增加你的 Xamarin.Android 项目的 Java 堆大小进行解决。

可以到较新的 NuGet 包更新 Xamarin.Forms 默认模板？

2018/6/22 • • Edit Online

本指南使用 Xamarin.Forms.NET 标准库模板作为示例，但相同的常规方法也适用于 Xamarin.Forms 共享项目模板。使用 Xamarin.Forms 1.5.1.6471 到 2.1.0.6529 从更新的示例编写此指南但相同的步骤可以改为将其他版本设置为默认值。

1. 复制原始模板 `.zip` 从：

   ```
   C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Xamarin\Xamarin\[Xamarin Version]\T\PT\Cross-Platform\Xamarin.Forms.PCL.zip
   ```

2. 解压缩 `.zip` 到临时位置。

3. 将窗体包的旧版本出现的所有更改为你想要使用的新版本。

   - `FormsTemplate\FormsTemplate.vstemplate`
   - `FormsTemplate.Android\FormsTemplate.Android.vstemplate`
   - `FormsTemplate.iOS\FormsTemplate.iOS.vstemplate`

   示例：`<package id="Xamarin.Forms" version="1.5.1.6471" />` -> `<package id="Xamarin.Forms" version="2.1.0.6529" />`

4. 更改主要的"name"元素多项目模板文件（`Xamarin.Forms.PCL.vstemplate`）以确保其唯一性。例如：

   ```
   空白应用（Xamarin.Forms 可移植）-2.1.0.6529
   ```

5. 重新压缩整个模板文件夹。请确保要匹配的原始文件结构 `.zip` 文件。`Xamarin.Forms.PCL.vstemplate` 文件应在顶部 `.zip` 文件，不在任何文件夹内的。

6. 在你每个用户的 Visual Studio 模板文件夹中创建的"移动应用程序"子目录：

   ```
   %USERPROFILE%\Documents\Visual Studio 2013\Templates\ProjectTemplates\Visual C#\Mobile Apps
   ```

7. 将新的压缩向上模板文件夹复制到新的"移动应用程序"目录。

8. 下载与步骤 3 中的版本匹配的 NuGet 程序包。例如，http://nuget.org/api/v2/package/Xamarin.Forms/2.1.0.6529 (另请参阅 http://stackoverflow.com/questions/8597375/how-to-get-the-url-of-a-nupkg-file)，并将其复制到相应的子文件夹的 Xamarin Visual Studio 扩展文件夹：

   ```
   C:\Program Files (x86)\Microsoft Visual Studio 14.0\Common7\IDE\Extensions\Xamarin\Xamarin\[Xamarin Version]\Packages
   ```

Visual Studio XAML 设计器不起作用的 Xamarin.Forms XAML 文件

2018/6/22 • • Edit Online

Xamarin.Forms 当前不支持的 XAML 文件的可视化设计器。因此，尝试在 Visual Studio 中打开窗体 XAML 文件时*XAML 设计器*或*XAML 使用的编码的 UI 设计器*，引发以下错误消息：

> "使用选择的编辑器无法打开该文件。请选择另一个编辑器。"

中介绍了此限制概述部分Xamarin.Forms XAML 基础知识指南：

> "不存在，但仍用于 Xamarin.Forms 应用程序中生成 XAML 可视化设计器中的，因此，所有 XAML 必须都可以手工编写。"

但是，可以通过选择显示 Xamarin.Forms XAML 预览程序视图 > **其他 Windows > Xamarin.Forms** 预览程序菜单选项。

Android 生成错误 – LinkAssemblies 任务意外失败

2018/10/26 • • Edit Online

可能会看到一条错误消息 `The "LinkAssemblies" task failed unexpectedly` 时生成 Xamarin.Android 项目使用的窗体。链接器处于活动状态时将发生这种情况 (通常在*版本*生成以减少应用包的大小); 以及执行因为 Android 目标不会更新到最新的 framework。(详细信息:适用于 Android 的要求的 Xamarin.Forms)

此问题的解决方法是确保具有支持最新 Android SDK 版本，并设置**目标框架**到**使用最新安装的平台**。此外建议您设置**目标 Android 版本**到**使用目标框架版本**并**最低 Android 版本**为 API 15 或更高版本。这被视为受支持的配置。

在 Visual Studio for Mac 的设置

1. 右键单击 Android 项目。
2. 转到**生成 > 常规 > 目标 Framework**。
3. 设置**目标框架**: **使用最新安装的平台**。
4. 仍在项目选项，请转到**生成 > Android 应用程序**。
5. 设置**最低 Android 版本**为 15 个或更高版本的 API 级别 &**目标 Android 版本**到**自动-使用目标框架版本**。

在 Visual Studio 中的设置

1. 右键单击 Android 项目。
2. 转到**应用程序**项目选项中。
3. 设置**使用 Android 版本编译** & **目标 Android 版本**设置为**使用最新的平台 / 使用使用 SDK 版本**编译。
4. 设置**的最低 Android 目标**设置为 API 19 或更高。

更新这些设置之后，请清除并重新生成项目以确保所做的更改会选取。

我 Xamarin.Forms.Maps Android 项目，COMPILETODALVIK 意外顶级错误为什么失败？

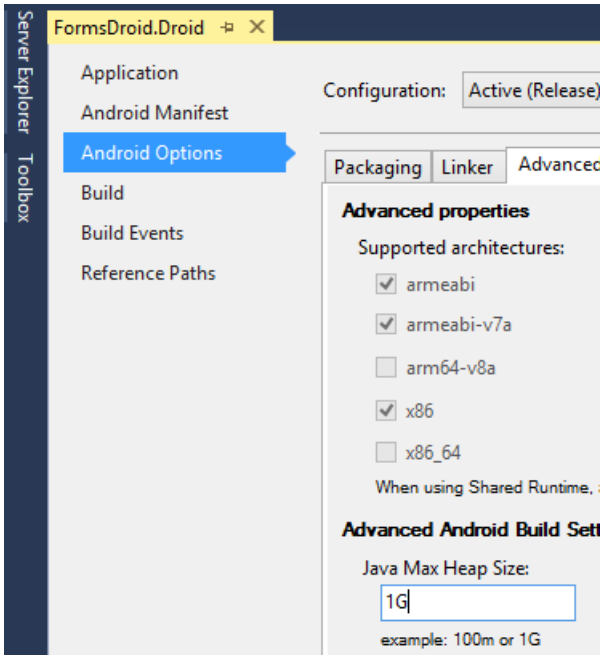
2018/6/22 • • Edit Online

适用于 Mac 或 Visual Studio; 的生成输出窗口中，可能会在 Visual Studio 的错误小键盘出现此错误在 Android 项目中使用 Xamarin.Forms.Maps。

这通常是通过增加你的 Xamarin.Android 项目的 Java 堆大小进行解决。请按照这些步骤以提高堆大小：

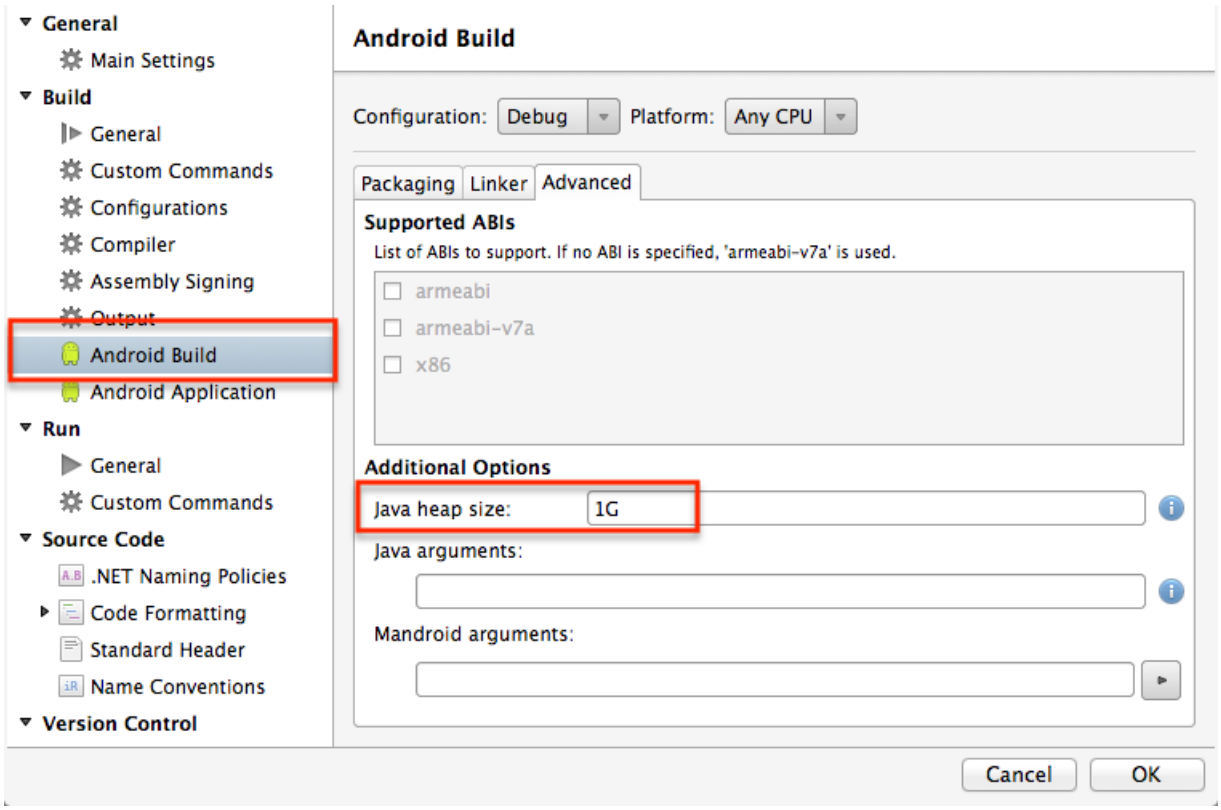
Visual Studio

1. 右击 Android 项目 (&) 打开项目选项。
2. 转到**Android** 选项**-> 高级**
3. 在 Java 堆大小文本框中输入 1 G。
4. 重新生成项目。



Visual Studio for Mac

1. 右击 Android 项目 (&) 打开项目选项。
2. 转到**生成-> Android 生成-> 高级**
3. 在 Java 堆大小文本框中输入 1 G。
4. 重新生成项目。


General
Main Settings
Build
General
Custom Commands
Configurations
Compiler
Assembly Signing
Output
Android Build
Android Application
Run
General
Custom Commands
Source Code
.NET Naming Policies
Code Formatting
Standard Header
Name Conventions
Version Control
Android Build
Configuration: Debug
Platform: Any CPU
Packaging
Linker
Advanced
Supported ABIs
List of ABIs to support. If no ABI is specified, 'armeabi-v7a' is used.
armeabi
armeabi-v7a
x86
Additional Options
Java heap size: 1G
Java arguments:
Mandroid arguments:
Cancel
OK

使用 Xamarin.Forms 书籍创建移动应用

2018/11/13 • • Edit Online

本书*使用 Xamarin.Forms 创建移动应用*Charles Petzold 的是有关学习如何编写 Xamarin.Forms 应用程序的指南。唯一的先决条件是了解C#编程语言。本书提供了大量探索到 Xamarin.Forms 的用户界面，还介绍了动画、MVVM、触发器、行为、自定义布局、自定义呈现器，以及更多。

本书在 2016 年春季发布，从那时起尚未更新。有很多书中保持该有价值，但一些材料已过时，和一些主题是不再完全正确，或者完成。

免费下载电子书

下载 Microsoft Virtual Academy 中的首选电子书格式：

- **PDF (56Mb)**
- **ePub (151Mb)**
- **Kindle 版 (325 Mb)**

此外可以下载各个章节作为 PDF 文件。

示例

这些示例可在 github 上，并且包括 iOS、Android 和通用 Windows 平台 (UWP) 项目。（Xamarin.Forms 不再支持 Windows 10 移动版，但 Xamarin.Forms 应用程序将在 Windows 10 桌面版上运行。）

章节摘要

章节摘要中有章表下面显示。这些摘要描述每个章节的内容，包括多种类型的链接：

- 指向实际两章（在页面底部），和相关文章的链接
- 中的所有示例的链接 **xamarin 窗体书籍示例** GitHub 存储库
- 有关更多详细说明的 Xamarin.Forms 类、结构、属性、枚举和等的 API 文档的链接

这些摘要还指示何时可能一章中的材料有点过时。

下载章节和摘要

一章	完整文本	总结
第 1 章。原理 Xamarin.Forms 合适大小是什么？	下载 PDF	摘要
第 2 章。应用剖析	下载 PDF	摘要
第 3 章。深入到文本	下载 PDF	摘要
第 4 章。滚动堆栈	下载 PDF	摘要

一章	完整文本	总结
第 5 章。处理大小	下载 PDF	摘要
第 6 章。按钮单击事件	下载 PDF	摘要
第 7 章。XAML vs。代码	下载 PDF	摘要
第 8 章。代码和 XAML 协调工作	下载 PDF	摘要
第 9 章。特定于平台的 API 调用	下载 PDF	摘要
第 10 章。XAML 标记扩展	下载 PDF	摘要
第 11 章。可绑定的基础结构	下载 PDF	摘要
第 12 章。样式	下载 PDF	摘要
第 13 章。位图	下载 PDF	摘要
第 14 章。绝对布局	下载 PDF	摘要
第 15 章。交互式接口	下载 PDF	摘要
第 16 章。数据绑定	下载 PDF	摘要
第 17 章。控制网格	下载 PDF	摘要
第 18 章。MVVM	下载 PDF	摘要
第 19 章。集合视图	下载 PDF	摘要
第 20 章。异步和文件 I/O	下载 PDF	摘要
第 21 章。转换	下载 PDF	摘要
第 22 章。动画	下载 PDF	摘要
第 23 章。触发器和行为	下载 PDF	摘要
第 24 章。页面导航	下载 PDF	摘要
第 25 章。页类型	下载 PDF	摘要
第 26 章。自定义布局	下载 PDF	摘要
第 27 章。自定义呈现器	下载 PDF	摘要
第 28 章。位置和地图	下载 PDF	摘要

此书与已过时

自发布以来*使用 Xamarin.Forms 创建移动应用*，已添加到 Xamarin.Forms 多项新功能。这些新功能在单独文章中所述Xamarin.Forms文档。

其他更改会导致一些此书是过时的内容：

.NET standard 2.0 库已替换为可移植类库

Xamarin.Forms 应用程序通常使用的库以在不同平台之间共享代码。最初，这是可移植类库 (PCL)。有多项 Pcl 引用整个本书和章节摘要。

可移植类库已替换为.NET Standard 2.0 库，如本文所述Xamarin.Forms 中.NET Standard 2.0 支持。所有示例代码从本书已更新为使用.NET Standard 2.0 库。

大多数关于可移植类库的角色一书中的信息将保持不变的.NET Standard 2.0 库。区别在于仅 PCL 具有数值"配置文件。" 此外，还有一些对.NET Standard 2.0 库的优势。例如，第 20 章，Async 和文件 I/O介绍如何使用用于执行文件 I/O 的基础平台。这不再是必需的。.NET Standard 2.0 库支持熟悉System.IO所有 Xamarin.Forms 平台的类。

.NET Standard 2.0 库还允许 Xamarin.Forms 应用程序使用 `HttpClient` 通过 Internet 访问的文件而非 `WebRequest` 或其他类。

XAML 的角色具有较高

*使用 Xamarin.Forms 创建移动应用*首先介绍如何编写 Xamarin.Forms 应用程序使用 C#。Extensible Application Markup Language (XAML) 不引入直到第 7 章。XAML vs。代码。

XAML 现在在 Xamarin.Forms 中有多重要的角色。通过 Visual Studio 分发的 Xamarin.Forms 解决方案模板创建基于 XAML 的页面文件。使用 Xamarin.Forms 的开发人员应熟悉 XAML 尽可能早。可扩展应用程序标记语言 (XAML) Xamarin.Forms 文档部分包含有关 XAML 来帮助你入门的多篇文章。

受支持的平台

Xamarin.Forms 不再支持 Windows 8.1 和 Windows Phone 8.1。

本书有时将引用_Windows 运行时_。这是一个术语，它包含使用 Windows 和 Windows Phone 的多个版本的 Windows API。较新版本的 Xamarin.Forms 将限制本身支持通用 Windows 平台，这是 API 适用于 Windows 10 和 Windows 10 移动版。

.NET Standard 2.0 库不支持任何版本的 Windows 10 移动版。因此，使用.NET Standard 库的 Xamarin.Forms 应用程序将不运行 Windows 10 移动版设备上。Xamarin.Forms 应用程序继续运行在 Windows 10 桌面版版本 10.0.16299.0 及更高版本。

Xamarin.Forms 具有支持预览版Mac，WPF，GTK #，以及Tizen平台。

章节摘要

章节摘要包括由于本书编写有关 Xamarin.Forms 中的更改的信息。这些通常是在便笺的窗体中：

> **NOTE**
> 每一页上的说明指示其中 Xamarin.Forms 已脱离一书中介绍的内容。

示例

在中 **xamarin 窗体书籍示例** GitHub 存储库**原始代码从本**书分支包含与此书一致程序示例。主分支包含已升级删除弃用的 Api，以显示增强的 Api 的项目。此外，Android 项目中主分支已升级适用于 Android通过 AppCompat Material Design将通常显示在白色背景黑色文本。

相关链接

- MS 按博客

- 从本书的示例代码

使用 Xamarin.Forms 电子书的企业应用程序模式

2018/11/13 • • Edit Online

开发自适应、可维护性、和可测试的 Xamarin.Forms 企业应用程序的体系结构指南

此电子书提供有关如何实现模型-视图-视图模型 (MVVM) 模式、依赖关系注入、导航、验证和配置管理，同时保持松散耦合的指南。此外，还有指导执行身份验证和授权与 IdentityServer，从容器化微服务和单元测试访问的数据。

前言

本章介绍的目的和范围的指南，以及它旨在的人员。

介绍

企业应用的开发人员面临着多种挑战，可以在开发过程中更改应用的体系结构。因此，很重要，以便可以修改或扩展随着时间的推移生成应用。这种适应性的设计可能很困难，但通常涉及到离散的松散耦合组件，可以轻松地集成在一起到应用程序分区应用程序。

MVVM

模型-视图-视图模型 (MVVM) 模式有助于完全隔离的应用程序从其用户界面 (UI) 的业务和演示文稿逻辑。维护应用程序逻辑与 UI 之间完全分离有助于解决许多开发问题，并可以使应用程序更易于测试、维护和改进。它还可以显著改善代码重用机会，并允许开发人员和开发的应用程序及其相应部分时，UI 设计器更轻松地进行协作。

依赖关系注入

依赖关系注入，具体取决于这些类型的代码类型的分离。它通常使用保存的注册和接口和抽象类型之间的映射列表的容器和实现或扩展这些类型的具体类型。

依赖关系注入容器减少通过提供一个工具用于实例化类实例和管理基于容器的配置其生存期的对象之间的耦合。对象在创建期间，容器将该对象需要的任何依赖关系注入到它。如果尚未创建这些依赖项，该容器创建，并将首先解析其依赖项。

松散耦合组件之间的通信

Xamarin.Forms `MessagingCenter` 类实现发布-订阅模式，允许基于消息的不太方便链接对象和类型引用的组件之间的通信。此机制允许发布服务器和订阅服务器进行通信而无需到对方，帮助减少组件，同时还允许要进行单独开发和测试的组件之间的依赖关系的引用。

导航

Xamarin.Forms 包括对页面导航、从用户的交互用户界面时，或从应用本身，由于内部逻辑驱动的状态更改时，通常会支持。但是，导航可能很复杂，若要在应用中使用 MVVM 模式的实现。

这一章介绍 `NavigationService` 类，用于执行从视图模型的视图模型第一个导航。将导航逻辑放在视图模型类意味着通过自动测试可在逻辑。此外，视图模型然后可以实现对控件导航，以确保实施某些业务规则的逻辑。

验证

接受用户输入的任何应用程序应确保输入有效。而不进行验证，用户可以提供会导致应用失败的数据。验证强制实施业务规则，可防止攻击者将恶意数据注入。

在上下文的模型-视图-视图模型 (MVVM) 模式，视图模型或模型通常需要执行数据验证和信号到视图的任何验证错误，以便用户可以更正它们。

配置管理

设置允许的数据的配置，请将代码中应用程序的行为的分离允许要进行更改而无需重新生成应用程序的行为。应用程序设置数据应用程序创建和管理，并且用户设置是可自定义应用的设置，会影响应用的行为，并且不需要频繁重新调整。

容器化微服务

微服务提供了应用程序开发和部署适用于现代云应用程序的敏捷性、规模和可靠性要求的方法。微服务的主要优点之一是，它们可以是扩展的独立，这意味着需要更多的处理能力或网络带宽，以支持需求，而不会不必要地缩放的区域，可以扩展特定的功能区域不会遇到更高的需求应用程序。

身份验证和授权

有许多方法与 ASP.NET MVC web 应用程序进行通信的 Xamarin.Forms 应用中集成身份验证和授权。在这里，与使用 IdentityServer 4 的容器化的标识微服务执行身份验证和授权。IdentityServer 是用于与 ASP.NET Core 标识来执行持有者令牌身份验证集成的 ASP.NET Core 的开放源代码 OpenID Connect 和 OAuth 2.0 框架。

访问远程数据

许多现代的基于 web 的解决方案进行 web 服务，由 web 服务器，以便为远程客户端应用程序提供功能的使用。Web 服务公开的操作构成 web API，并且客户端应用程序应该能够使用 web API，而不知道如何实现的数据或 API 公开的操作。

单元测试

测试模型和视图模型的 MVVM 应用程序从等同于测试任何其他类，并可以使用相同的工具和技术。但是，有一些是模型的典型的模式和视图模型类，可受益于特定的单元测试技术的。

反馈

此项目有一个社区站点，可以在其发布问题，并提供反馈。社区站点将位于GitHub。或者，有关电子书的反馈可以通过电子邮件发送到 dotnet-architecture-ebooks-feedback@service.microsoft.com 。

相关链接

- 下载电子书 (2 Mb PDF)
- eShopOnContainers (GitHub)（示例）

在 Xamarin.Forms 中 SkiaSharp 图形

2018/10/25 • • Edit Online

用于二维图形 Xamarin.Forms 应用程序中使用 SkiaSharp

SkiaSharp 是.NET 和 C# 由 Google 产品中广泛使用的开放源代码 Skia 图形引擎提供支持的 2D 图形系统。可以在 Xamarin.Forms 应用程序中使用 SkiaSharp 绘制 2D 矢量图形、位图和文本。请参阅2D 绘制SkiaSharp 库有关的更多常规信息和一些其他教程的指南。

本指南假定你熟悉 Xamarin.Forms 编程。

适用于 Xamarin.Forms 的网络研讨会：SkiaSharp

SkiaSharp 初步准备之后

SkiaSharp xamarin.forms 打包为 NuGet 包。创建 Xamarin.Forms 解决方案在 Visual Studio 或 Visual Studio for Mac 后，可以使用 NuGet 包管理器来搜索**SkiaSharp.Views.Forms**包并将其添加到你的解决方案。如果选中**引用**部分的每个项目添加 SkiaSharp 后，可以看到各种**SkiaSharp**库已添加到每个解决方案中的项目。

如果在 Xamarin.Forms 应用程序面向 iOS，使用项目属性页上以最小的部署目标更改为 iOS 8.0。

在使用 SkiaSharp 任何 C# 页面中，将想要包括 `using` 指令 `SkiaSharp` 命名空间，其中包含所有 SkiaSharp 类、结构和将在图形中使用的枚举编程。您还应该 `using` 指令 `SkiaSharp.Views.Forms` 类特定于 Xamarin.Forms 的命名空间。这是一个更小命名空间，其中最重要的类是 `SKCanvasView` 。此类派生自 Xamarin.Forms `View` 类，并承载 SkiaSharp 图形输出。

> **IMPORTANT**
> `SkiaSharp.Views.Forms` 命名空间还包含 `SKGLView` 派生的类 `View` 但使用 OpenGL 的呈现图形。为了简单起见，本指南将限制到本身 `SKCanvasView` ，但使用 `SKGLView` 相反是非常相似。

SkiaSharp 绘制基础知识

一些可以使用 SkiaSharp 绘制的最简单的图形图是圆、椭圆和矩形。在显示这些数字，你将了解 SkiaSharp 坐标、大小和颜色。文本和位图的显示是更复杂，但这些文章还介绍这些技术。

SkiaSharp 线和路径

图形路径是一系列相互连接的直线和曲线。路径可以描边，填充，或两者。本文包含线条图形，包括笔划结束和联接，并带短划线和点线，但缺乏曲线几何图形的停止点的多个方面。

SkiaSharp 转换

转换允许图形对象统一转换、缩放、旋转或倾斜。本文还介绍如何创建非仿射转换和将转换应用于路径使用标准的 3-3 转换矩阵。

SkiaSharp 曲线和路径

路径的探索继续将曲线添加到路径对象，并利用功能强大的路径的其他功能。您将看到如何简洁的文本字符串中

指定完整路径、如何使用路径的效果，以及如何深入探讨路径内部结构。

SkiaSharp 位图

位图是位对应于显示设备的像素的矩形数组。本系列文章演示如何加载、保存、显示、创建、上绘制、进行动画处理，并访问 SkiaSharp 位图的位。

SkiaSharp 效果

效果是 alter 正常显示图形，包括线性和循环渐变、位图平铺、混合模式、模糊和其他人的属性。

相关链接

- SkiaSharp Api
- SkiaSharpFormsDemos（示例）
- SkiaSharp 通过 Xamarin.Forms 网络研讨会（视频）