

# Windows Presentation Foundation 文档

了解如何在 .NET 7 上使用 Windows Presentation Foundation (WPF)，这是一种适用于 Windows 的开放源代码图形用户界面。

## 了解 WPF

### 概述

[WPF 概述](#)

### 新变化

[与 .NET Framework 的差异](#)

### 教程

[创建新应用](#)

[从 .NET Framework 迁移](#)

### 下载

[Visual Studio 2022 版本 17.4](#)

## WPF 窗口

### 概述

[关于 WPF 窗口](#)

[关于对话框](#)

### 操作指南

[打开消息框](#)

[打开通用对话框](#)

[获取或设置主应用窗口](#)

## 控件样式和模板

### 回 概述

[关于样式和模板](#)

---

### 回 操作指南

[创建样式](#)

[创建模板](#)

## 数据绑定

### 回 概述

[关于数据绑定](#)

[声明绑定](#)

## WPF 中的 XAML

### 回 概述

[关于 XAML](#)

# 桌面指南 (WPF .NET)

项目 • 2023/10/13

欢迎使用 Windows Presentation Foundation (WPF) 桌面指南，这是一个与分辨率无关的 UI 框架，使用基于矢量的呈现引擎，构建用于利用现代图形硬件。 WPF 提供一套完善的应用程序开发功能，这些功能包括 Extensible Application Markup Language (XAML)、控件、数据绑定、布局、二维和三维图形、动画、样式、模板、文档、媒体、文本和版式。 WPF 属于 .NET，因此可以生成整合 .NET API 其他元素的应用程序。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

WPF 有两种实现：

### 1. .Net 版本 (本指南) :

[GitHub](#) 上托管的 WPF 开源实现，可在 .Net 5 上运行。适用于 XAML 设计器最低要求 [Visual Studio 2019 版本 16.8](#)。但根据 .NET 的版本，可能需要使用较新版本的 Visual Studio。

尽管 .NET 是一种跨平台技术，但 WPF 仅在 Windows 上运行。

### 2. .NET Framework 4 版本:

受 Visual Studio 2019 和 Visual Studio 2017 支持的 WPF 的 .NET Framework 实现。

.NET Framework 4 是仅限 Windows 的 .NET 版本，被视为一个 Windows 操作系统组件。此版本的 WPF 随 .NET Framework 一起分发。有关 WPF 的 .NET Framework 版本的详细信息，请参阅[适用于 .NET Framework 的 WPF 简介](#)。

本概述适用于新用户，介绍了 WPF 的主要功能和概念。 若要了解如何创建 WPF 应用程序，请参阅教程：[创建新的 WPF 应用](#)。

## 为何从 .NET Framework 升级

将应用程序从 .NET Framework 升级到 .NET 时，你将受益于：

- 性能更好
- 新的 .NET API
- 最新语言改进
- 改进的辅助功能和可靠性

- 更新的工具及其他

若要了解如何升级应用程序，请参阅[如何将 WPF 桌面应用升级到 .NET 7](#)。

## 使用 WPF 进行编程

WPF 作为 .NET 类型的一个子集存在，大部分位于 `System.Windows` 命名空间中。如果你曾经使用 ASP.NET 和 Windows 窗体等框架通过 .NET 构建应用程序，应该会熟悉基本的 WPF 编程体验：

- 实例化类
- 设置属性
- 调用方法
- 处理事件

WPF 还包括可增强属性和事件的其他编程构造：[依赖项属性](#)和[路由事件](#)。

## 标记和代码隐藏

通过 WPF，可以使用标记和代码隐藏开发应用程序，这是 ASP.NET 开发人员已经熟悉的体验。通常使用 XAML 标记实现应用程序的外观，同时使用托管编程语言（代码隐藏）来实现其行为。这种外观和行为的分离具有以下优点：

- 降低了开发和维护成本，因为特定于外观的标记与特定于行为的代码不紧密耦合。
- 开发效率更高，因为设计人员在实现应用程序外观的同时，开发人员可以实现应用程序的行为。
- WPF 应用程序的[全球化和本地化](#) 得以简化。

## 标记

XAML 是一种基于 XML 的标记语言，以声明形式实现应用程序的外观。通常用它定义窗口、对话框、页面和用户控件，并填充控件、形状和图形。

下面的示例使用 XAML 来实现包含一个按钮的窗口的外观：

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">
```

```
<!-- Add button to window -->
<Button Name="button">Click Me!</Button>

</Window>
```

具体而言，此 XAML 使用 `Window` 元素定义窗口，使用 `Button` 元素定义按钮。每个元素均配置了特性（如 `Window` 元素的 `Title` 特性）来指定窗口的标题栏文本。在运行时，WPF 会将标记中定义的元素和特性转换为 WPF 类的实例。例如，`Window` 元素被转换为 `Window` 类的实例，该类的 `Title` 属性是 `Title` 特性的值。

下图显示上一个示例中的 XAML 定义的用户界面 (UI)：



由于 XAML 是基于 XML 的，因此使用它编写的 UI 汇集在嵌套元素的层次结构中，称为 **元素树**。元素树提供了一种直观的逻辑方式来创建和管理 UI。

## 代码隐藏

应用程序的主要行为是实现响应用户交互的功能。例如，单击菜单或按钮，以及在响应中调用业务逻辑和数据访问逻辑。在 WPF 中，在与标记相关联的代码中实现此行为。此类代码称为代码隐藏。下面的示例演示上一个示例的更新标记和代码隐藏：

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

    <!-- Add button to window -->
    <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

更新的标记定义 `xmlns:x` 命名空间，并将其映射到为代码隐藏类型添加支持的架构。

`x:Class` 特性用于将代码隐藏类与此特定 XAML 标记相关联。考虑此特性在 `<Window>` 元素上声明，代码隐藏类必须从 `Window` 类继承。

C#

```
using System.Windows;

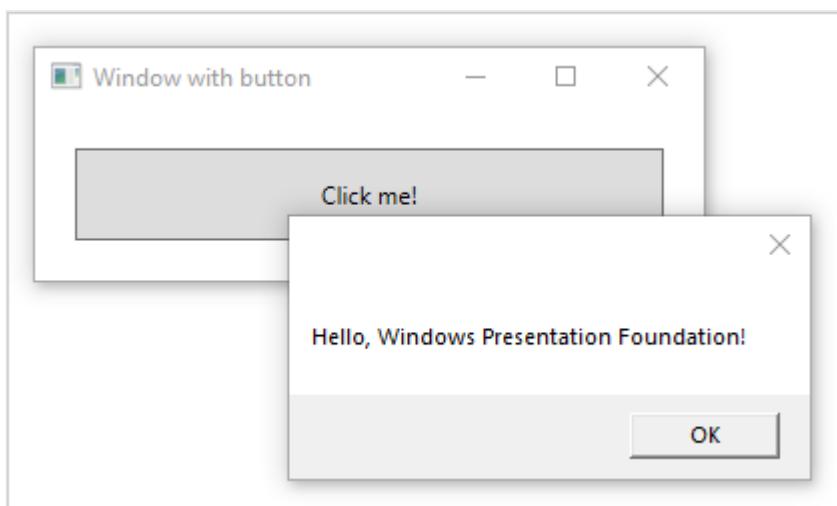
namespace SDKSample
{
    public partial class AWindow : Window
    {
        public AWindow()
        {
            // InitializeComponent call is required to merge the UI
            // that is defined in markup with this class, including
            // setting properties and registering event handlers
            InitializeComponent();
        }

        void button_Click(object sender, RoutedEventArgs e)
        {
            // Show message box when button is clicked.
            MessageBox.Show("Hello, Windows Presentation Foundation!");
        }
    }
}
```

从代码隐藏类的构造函数调用 `InitializeComponent`，以将标记中定义的 UI 与代码隐藏类合并在一起。（生成应用程序时即会生成 `InitializeComponent`，因此不需要手动实现它。）`x:Class` 和 `InitializeComponent` 的组合可确保在创建实现时正确地对其进行初始化。

请注意，在标记中，`<Button>` 元素定义了 `click` 属性的值 `button_click`。将标记和代码隐藏初始化并使其一起工作后，按钮的 `Click` 事件会自动映射到 `button_click` 方法。单击该按钮时，将调用事件处理程序，并通过调用 `System.Windows.MessageBox.Show` 方法显示一个消息框。

下图显示单击该按钮后的结果：



# 输入和命令

最常检测和响应用户输入的控件。 WPF 输入系统 使用直接事件和路由事件来支持文本输入、 焦点管理和鼠标定位。

应用程序通常具有复杂的输入要求。 WPF 提供了命令系统，用于将用户输入操作与对这些操作做出响应的代码分隔开来。 命令系统允许多个源调用相同的命令逻辑。 例如， 进行由不同应用程序使用的常见编辑操作： 复制、 剪切和粘贴。 如果使用命令实现了这些操作，则它们可以由不同的用户操作调用。

## 控件

应用程序模型带来的用户体验是构造的控件。 在 WPF 中，“控件”是一个概括性术语，适用于具有以下特征的 WPF 类类别：

- 托管在窗口或页面中。
- 拥有用户界面。
- 实现某些行为。

有关详细信息，请参阅 [控件](#)。

## 按功能分类的 WPF 控件

下面列出了内置的 WPF 控件：

- **按钮**: [Button](#) 和 [RepeatButton](#)。
- **数据显示**: [DataGrid](#)、 [ListView](#) 和 [TreeView](#)。
- **日期显示和选项**: [Calendar](#) 和 [DatePicker](#)。
- **对话框**: [OpenFileDialog](#)、 [PrintDialog](#)和 [SaveFileDialog](#)。
- **数字墨迹**: [InkCanvas](#) 和 [InkPresenter](#)。
- **文档**: [DocumentViewer](#)、 [FlowDocumentPageViewer](#)、 [FlowDocumentReader](#)、 [FlowDocumentScrollView](#)和 [StickyNoteControl](#)。
- **输入**: [TextBox](#)、 [RichTextBox](#)和 [PasswordBox](#)。
- **布局**: [Border](#)、 [BulletDecorator](#)、 [Canvas](#)、 [DockPanel](#)、 [Expander](#)、 [Grid](#)、 [GridView](#)、 [GridSplitter](#)、 [GroupBox](#)、 [Panel](#)、 [ResizeGrip](#)、 [Separator](#)、 [ScrollBar](#)、 [ScrollViewer](#)、 [StackPanel](#)、 [Thumb](#)、 [Viewbox](#)、 [VirtualizingStackPanel](#)、 [Window](#)和 [WrapPanel](#)。

- **媒体**: Image、 MediaElement和 SoundPlayerAction。
- **菜单**: ContextMenu、 Menu和 ToolBar。
- **导航**: Frame、 Hyperlink、 Page、 NavigationWindow和 TabControl。
- **选项**: CheckBox、 ComboBox、 ListBox、 RadioButton和 Slider。
- **用户信息**: AccessText、 Label、 Popup、 ProgressBar、 StatusBar、 TextBlock和 ToolTip。

## 布局

创建用户界面时，按照位置和大小排列控件以形成布局。任何布局的一项关键要求都是适应窗口大小和显示设置的变化。WPF 为你提供一流的可扩展布局系统，而不强制你编写代码以适应这些情况下的布局。

布局系统的基础是相对定位，这提高了适应不断变化的窗口和显示条件的能力。该布局系统还可管理控件之间的协商以确定布局。协商是一个两步过程：首先，控件将需要的位置和大小告知父级。其次，父级将控件可以有的空间告知控件。

该布局系统通过基 WPF 类公开给子控件。对于通用的布局（如网格、堆叠和停靠），WPF 包括若干布局控件：

- Canvas: 子控件提供其自己的布局。
- DockPanel: 子控件与面板的边缘对齐。
- Grid: 子控件由行和列定位。
- StackPanel: 子控件垂直或水平堆叠。
- VirtualizingStackPanel: 子控件在水平或垂直的行上虚拟化并排列。
- WrapPanel: 子控件按从左到右的顺序放置，在当前行上的空间不足时换行到下一行。

下面的示例使用 DockPanel 布置几个 TextBox 控件：

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.LayoutWindow"
    Title="Layout with the DockPanel" Height="143" Width="319">
```

```

<!--DockPanel to layout four text boxes-->
<DockPanel>
    <TextBox DockPanel.Dock="Top">Dock = "Top"</TextBox>
    <TextBox DockPanel.Dock="Bottom">Dock = "Bottom"</TextBox>
    <TextBox DockPanel.Dock="Left">Dock = "Left"</TextBox>
    <TextBox Background="White">This TextBox "fills" the remaining space.
</TextBox>
</DockPanel>

</Window>

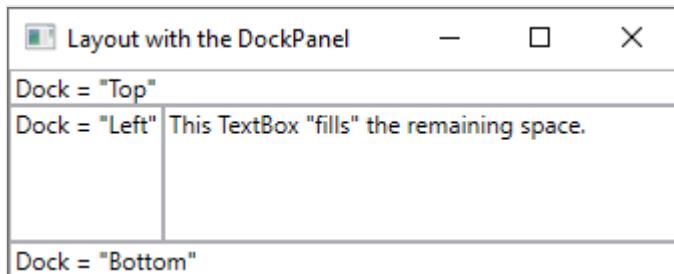
```

`DockPanel` 允许子 `TextBox` 控件，以告诉它如何排列这些控件。为了完成此操作，`DockPanel` 实现 `Dock` 附加了属性，该属性公开给子控件，以允许每个子控件指定停靠样式。

### ① 备注

由父控件实现以便子控件使用的属性是 WPF 构造，称为**附加属性**。

下图显示上一个示例中的 XAML 标记的结果：：

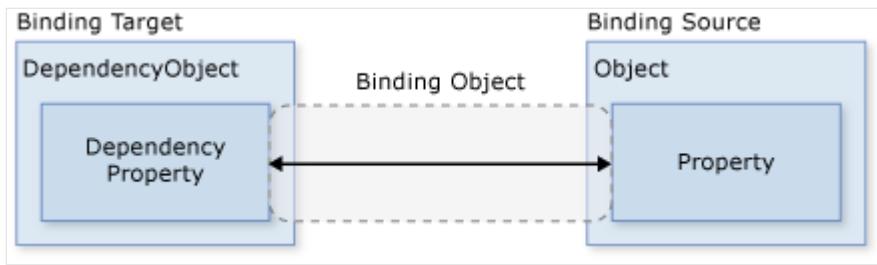


## 数据绑定

大多数应用程序旨在为用户提供查看和编辑数据的方法。对于 WPF 应用程序，存储和访问数据的工作已由许多不同的 .NET 数据访问库（例如 SQL 和 Entity Framework Core）提供。访问数据并将数据加载到应用程序的托管对象后，WPF 应用程序的复杂工作开始。从根本上来说，这涉及到两件事：

1. 将数据从托管对象复制到控件，在控件中可以显示和编辑数据。
2. 确保使用控件对数据所做的更改将复制回托管对象。

为了简化应用程序开发，WPF 提供了一个强大的数据绑定引擎来自动处理这些步骤。数据绑定引擎的核心单元是 `Binding` 类，其工作是将控件（绑定目标）绑定到数据对象（绑定源）。下图阐释了这种关系：



WPF 支持直接在 XAML 标记中声明绑定。例如，下面的 XAML 代码使用“{Binding ...}”XAML 语法将 `TextBox` 的 `Text` 属性绑定到对象的 `Name` 属性。这假设有一个数据对象设置为具有 `Name` 属性 `Window` 的 `DataContext` 属性。

```
XAML

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.DataBindWindow">

    <!-- Bind the TextBox to the data source (TextBox.Text to Person.Name) --
    >
    <TextBox Name="personNameTextBox" Text="{Binding Path=Name}" />

</Window>
```

WPF 数据绑定引擎不仅提供绑定，还提供验证、排序、筛选和分组。此外，数据绑定支持使用数据模板来为数据绑定创建自定义的用户界面。

有关详细信息，请参阅[数据绑定概述](#)。

## 图形和动画

WPF 提供一组广泛且灵活的图形功能，具有以下优点：

- 图形与分辨率和设备均无关。** WPF 图形系统中的基本度量单位是与设备无关的像素（即 1/96 英寸），为实现与分辨率和设备无关的呈现提供了基础。每个与设备无关的像素都会自动缩放，以匹配呈现它的系统的每英寸点数 (dpi) 设置。
- 精度更高。** WPF 坐标系统使用双精度浮点数字度量，而不是单精度数字。转换和不透明度值也表示为双精度数字。WPF 还支持广泛的颜色域 (scRGB)，并集成了对管理来自不同颜色空间的输入的支持。
- 高级图形和动画支持。** WPF 通过为你管理动画场景简化了图形编程，你无需担心场景处理、呈现循环和双线性内插。此外，WPF 还提供了点击测试支持和全面的 alpha 合成支持。
- 硬件加速。** WPF 图形系统充分利用图形硬件来尽量降低 CPU 使用率。

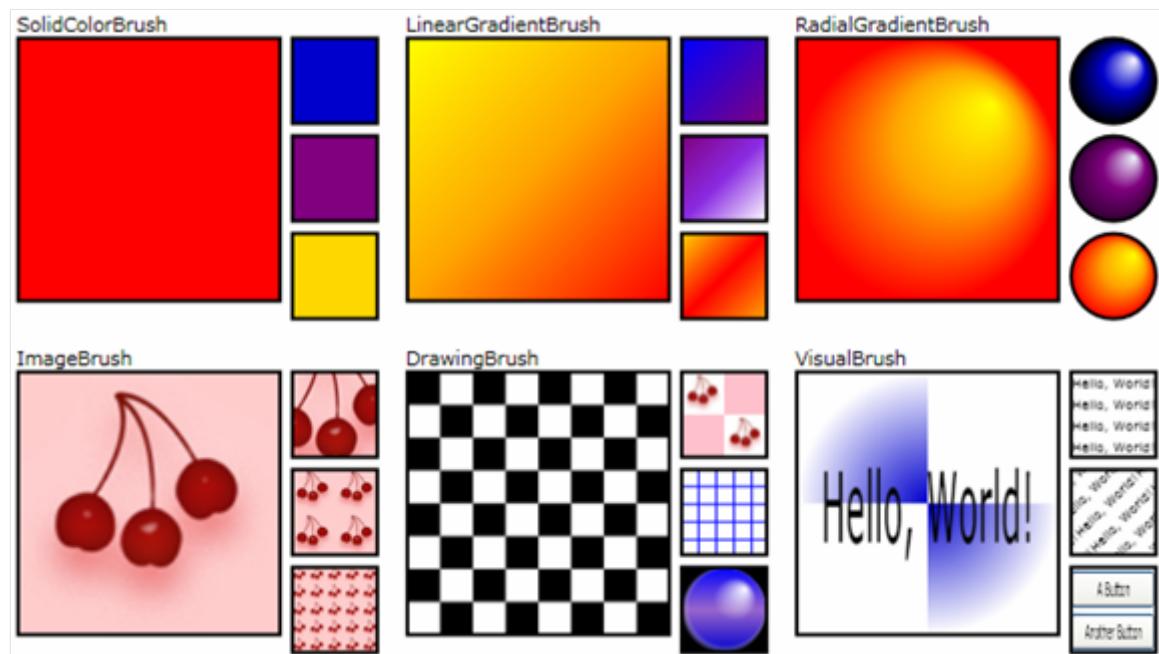
## 2D 图形

WPF 提供一个常用矢量绘制的二维形状库，例如矩形和椭圆。形状不只是用于显示；还会实现许多你期望的控件功能，包括键盘和鼠标输入。

WPF 提供的二维形状包含基本形状的标准集。但是，你可能需要创建自定义形状以辅助改进自定义用户界面的设计。WPF 提供几何图形来创建可直接绘制、用作画笔或用于剪辑其他形状和控件的自定义形状。

有关详细信息，请参阅[几何图形概述](#)。

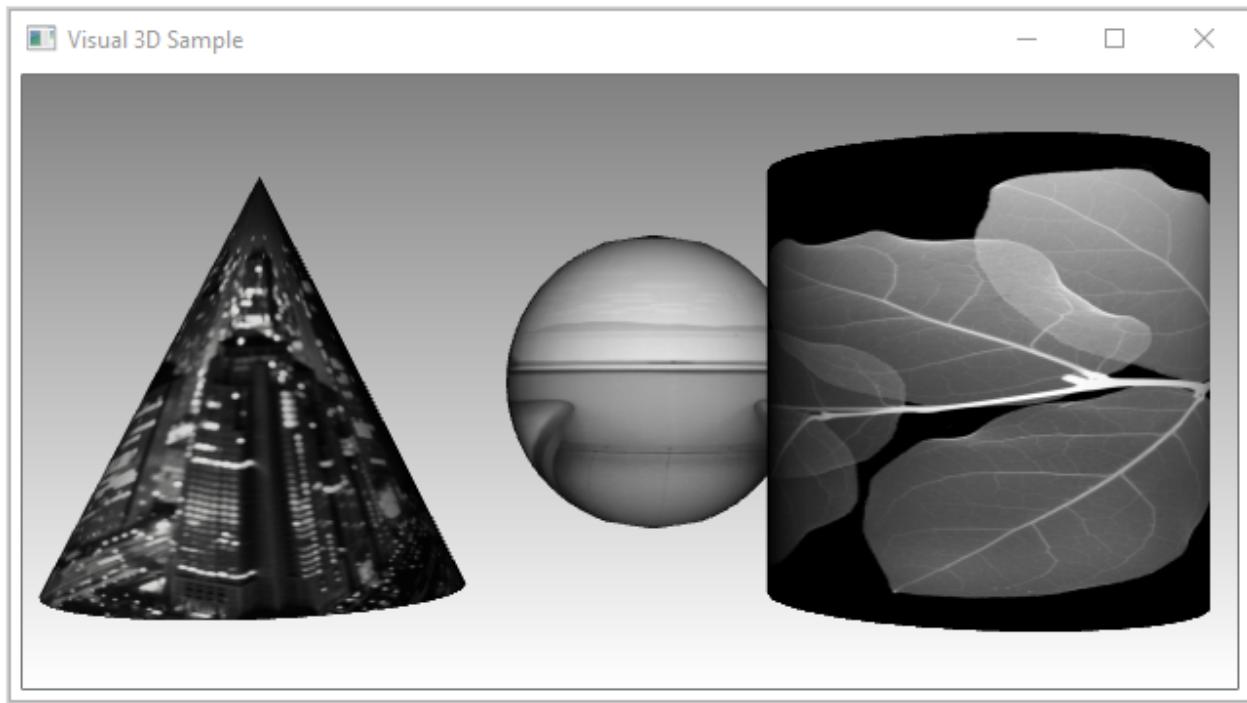
WPF 二维功能的子集包括视觉效果，如渐变、位图、绘图、用视频绘画、旋转、缩放和倾斜。这些效果都是通过画笔实现的。下图显示了一些示例：



有关详细信息，请参阅[WPF 画笔概述](#)。

## 三维呈现

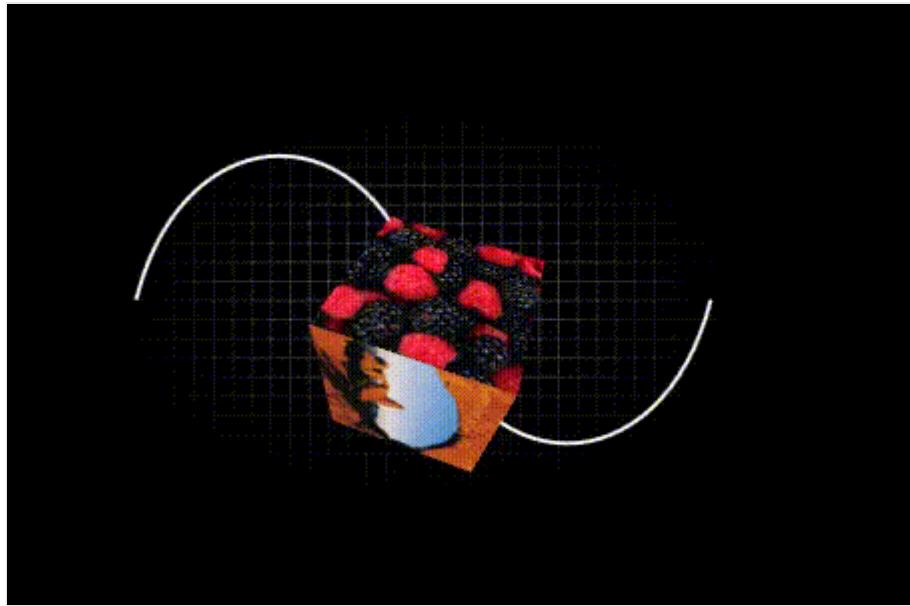
WPF 还包括三维呈现功能，这些功能与二维图形集成，以创建更精彩、更有趣的用户界面。例如，下图显示呈现在三维形状上的二维图像：



有关详细信息，请参阅[三维图形概述](#)。

## 动画

WPF 动画支持可以使控件变大、抖动、旋转和淡出，以形成有趣的页面过渡等。你可以对大多数 WPF 类，甚至自定义类进行动画处理。下图显示了运行中的一个简单动画：



有关详细信息，请参阅[动画概述](#)。

## 文本和版式

WPF 提供以下功能以实现高质量的文本呈现：

- OpenType 字体支持。

- ClearType 增强功能。
- 利用硬件加速的高性能。
- 文本与媒体、图形和动画的集成。
- 国际字体支持和回退机制。

作为文本与图形集成的演示，下图显示了文本修饰的应用程序：



有关详细信息，请参阅 [Windows Presentation Foundation 中的版式](#)。

## 自定义 WPF 应用

到目前为止，你已经了解用于开发应用程序的核心 WPF 构建块：

- 你可以使用该应用程序模型来托管和交付应用程序内容，它主要由控件组成。
- 为简化用户界面中控件的安排，可使用 WPF 布局系统。
- 可以使用数据绑定来减少将用户界面与数据集成的工作。
- 若要增强你应用程序的可视化外观，可以使用 WPF 提供的综合图形、动画和媒体支持。

不过，在创建和管理真正独特且视觉效果非凡的用户体验时，基础知识通常是不够的。标准的 WPF 控件可能无法与你所需的应用程序外观集成。数据可能不会以最有效的方式显示。你应用程序的整体用户体验可能不适合 Windows 主题的默认外观和感觉。

出于此原因，WPF 提供了各种机制来打造独特的用户体验。

## 内容模型

大多数 WPF 控件的主要用途是显示内容。在 WPF 中，可以构成控件内容的项的类型和数目称为控件的 **内容模型**。某些控件可以包含一种内容类型的一个项。例如，[TextBox](#) 的内容是分配给 [Text](#) 属性的一个字符串值。

但是，其他控件可以包含不同内容类型的多个项；[Button](#) 的内容（由 [Content](#) 属性指定）可以包含各种项，包括布局控件、文本、图像和形状。

有关各种控件支持的内容类型的详细信息，请参阅 [WPF 内容模型](#)。

## 触发器

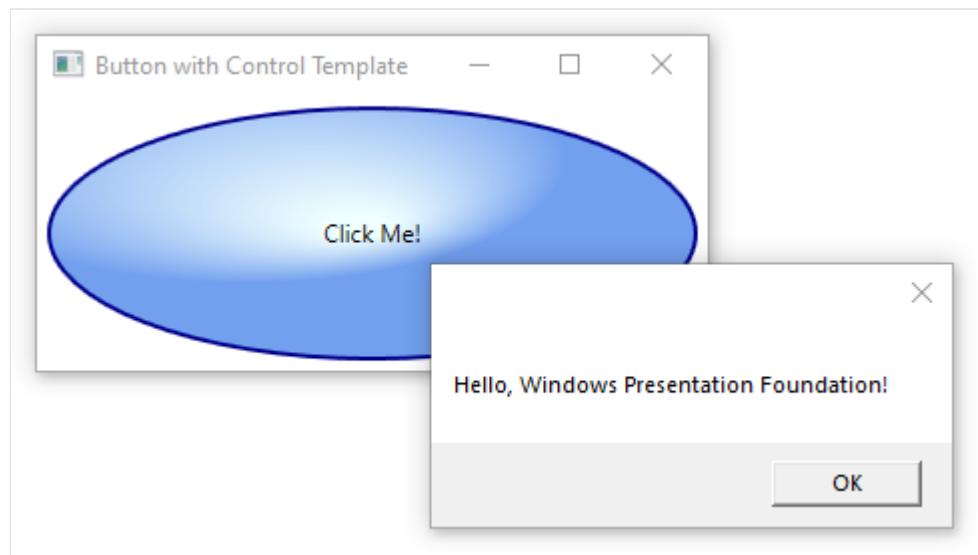
尽管 XAML 标记的主要用途是实现应用程序的外观，你也可以使用 XAML 来实现应用程序行为的某些方面。其中一个示例是使用触发器来基于用户交互更改应用程序的外观。有关详细信息，请参阅[样式和模板](#)。

## 模板

WPF 控件的默认用户界面通常是从其他控件和形状构造的。例如，[Button](#) 由 [ButtonChrome](#) 和 [ContentPresenter](#) 控件组成。[ButtonChrome](#) 提供了标准按钮外观，而 [ContentPresenter](#) 显示按钮的内容，正如 [Content](#) 属性所指定。

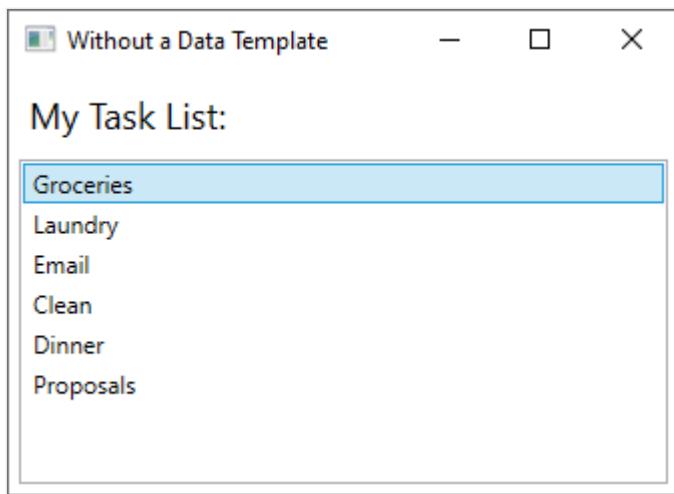
有时，某个控件的默认外观可能与应用程序的整体外观冲突。在这种情况下，可以使用 [ControlTemplate](#) 更改控件的用户界面的外观，而不更改其内容和行为。

例如，单击 [Button](#) 时会引发 [Click](#) 事件。通过更改按钮的模板来显示 [Ellipse](#) 形状，控件的可视方位发生了变化，但功能却没有。你仍可以单击该控件的可视方位，将按预期引发 [Click](#) 事件。

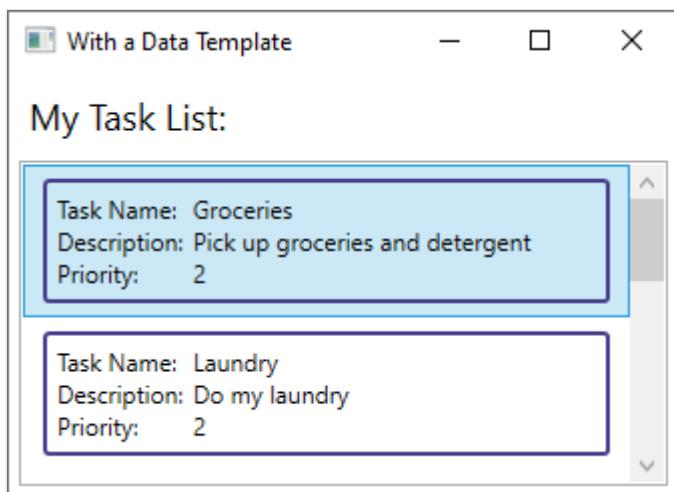


## 数据模板

使用控件模板可以指定控件的外观，而使用数据模板则可以指定控件内容的外观。数据模板经常用于改进绑定数据的显示方式。下图显示 [ListBox](#) 的默认外观，它绑定到 [Task](#) 对象的集合，其中每个任务都具有名称、描述和优先级：



默认外观是你对 [ListBox](#) 的期望。但是，每个任务的默认外观仅包含任务名称。若要显示任务名称、描述和优先级，必须使用 [ListBox](#) 更改 [DataTemplate](#) 控件绑定列表项的默认外观。下面是一个示例，说明如何应用为 [Task](#) 对象创建的数据模板。



[ListBox](#) 会保留其行为和整体外观；只有列表框所显示内容的外观发生变化。

有关详细信息，请参阅[数据模板化概述](#)。

## 样式

通过样式功能，开发人员和设计人员能够对其产品的特定外观进行标准化。WPF 提供了一个强样式模型，其基础是 [Style](#) 元素。样式可以将属性值应用于类型。引用样式时，可以根据类型将其自动应用于所有对象，或应用于单个对象。下面的示例创建一个样式，该样式将窗口上的每个 [Button](#) 的背景色设置为 [Orange](#)：

### XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.StyleWindow"
    Title="Styles">
```

```

<Window.Resources>
    <!-- Style that will be applied to all buttons for this window -->
    <Style TargetType="{x:Type Button}">
        <Setter Property="Background" Value="Orange" />
        <Setter Property="BorderBrush" Value="Crimson" />
        <Setter Property="FontSize" Value="20" />
        <Setter Property="FontWeight" Value="Bold" />
        <Setter Property="Margin" Value="5" />
    </Style>
</Window.Resources>
<StackPanel>

    <!-- This button will have the style applied to it -->
    <Button>Click Me!</Button>

    <!-- This label will not have the style applied to it -->
    <Label>Don't Click Me!</Label>

    <!-- This button will have the style applied to it -->
    <Button>Click Me!</Button>

</StackPanel>
</Window>

```

由于此样式针对所有 [Button](#) 控件，因此将自动应用于窗口中的所有按钮，如下图所示：



有关详细信息，请参阅[样式和模板](#)。

## 资源

应用程序中的控件应共享相同的外观，它可以包括从字体和背景色到控件模板、数据模板和样式的所有内容。你可以对用户界面资源使用 WPF 支持，以将这些资源封装在一个位置以便重复使用。

下面的示例定义 [Button](#) 和 [Label](#) 共享的通用背景色：

### XAML

```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

```

```
x:Class="SDKSample.ResourcesWindow"
Title="Resources Window">

<!-- Define window-scoped background color resource -->
<Window.Resources>
    <SolidColorBrush x:Key="defaultBackground" Color="Red" />
</Window.Resources>

<!-- Button background is defined by window-scoped resource -->
<Button Background="{StaticResource defaultBackground}">One
Button</Button>

<!-- Label background is defined by window-scoped resource -->
<Label Background="{StaticResource defaultBackground}">One Label</Label>
</Window>
```

有关详细信息，请参阅[如何定义和引用 WPF 资源](#)。

## 自定义控件

尽管 WPF 提供了大量自定义支持，但你仍可能会遇到现有 WPF 控件不满足你的应用程序或其用户的需求的情况。出现这种情况的原因有：

- 不能通过自定义现有 WPF 实现的外观和感觉创建所需的用户界面。
- 现有 WPF 实现不支持（或很难支持）所需的行为。

但是，此时，你可以充分利用三个 WPF 模型中的一个来创建新的控件。每个模型都针对一个特定的方案并要求你的自定义控件派生自特定 WPF 基类。下面列出了这三个模型：

- **用户控件模型**

自定义控件派生自 [UserControl](#) 并由一个或多个其他控件组成。

- **控件模型** 自定义控件派生自 [Control](#)，并用于生成使用模板将其行为与其外观分隔开来的实现，非常类似大多数 WPF 控件。派生自 [Control](#) 使得你可以更自由地创建自定义用户界面（相较用户控件），但它可能需要花费更多精力。

- **框架元素模型。**

当其外观由自定义呈现逻辑（而不是模板）定义时，自定义控件派生自 [FrameworkElement](#)。

有关自定义控件的详细信息，请参阅[控件创作概述](#)。

## 另请参阅

- 教程：创建新的 WPF 应用
- 将 WPF 应用迁移到 .NET

- WPF 窗口概述
- 数据绑定概述
- XAML概述

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 教程：使用 .NET 创建新的 WPF 应用

项目 • 2023/10/13

本简短教程将介绍如何使用 Visual Studio 创建新的 Windows Presentation Foundation (WPF) 应用。生成初始应用后，你将了解如何添加控件以及如何处理事件。学完本教程后，你将拥有一个可将名称添加到列表框的简单应用。

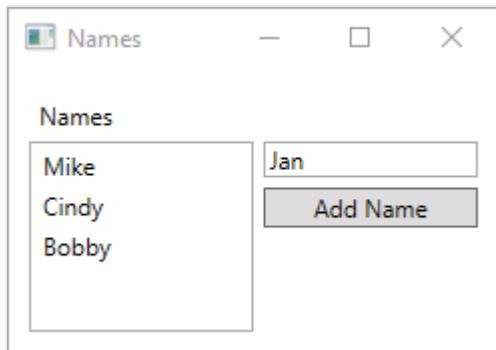
## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

本教程介绍如何执行下列操作：

- ✓ 创建新的 WPF 应用
- ✓ 将控件添加到窗体
- ✓ 处理控制事件以提供应用功能
- ✓ 运行应用

下面是按照本教程生成的应用的预览：



## 先决条件

### 💡 提示

使用 Visual Studio 2022 版本 17.4 或更高版本并安装 .NET 7 和 .NET 6 的各个组件。  
在 Visual Studio 2022 版本 17.4 中增加了对 .NET 7 的支持。

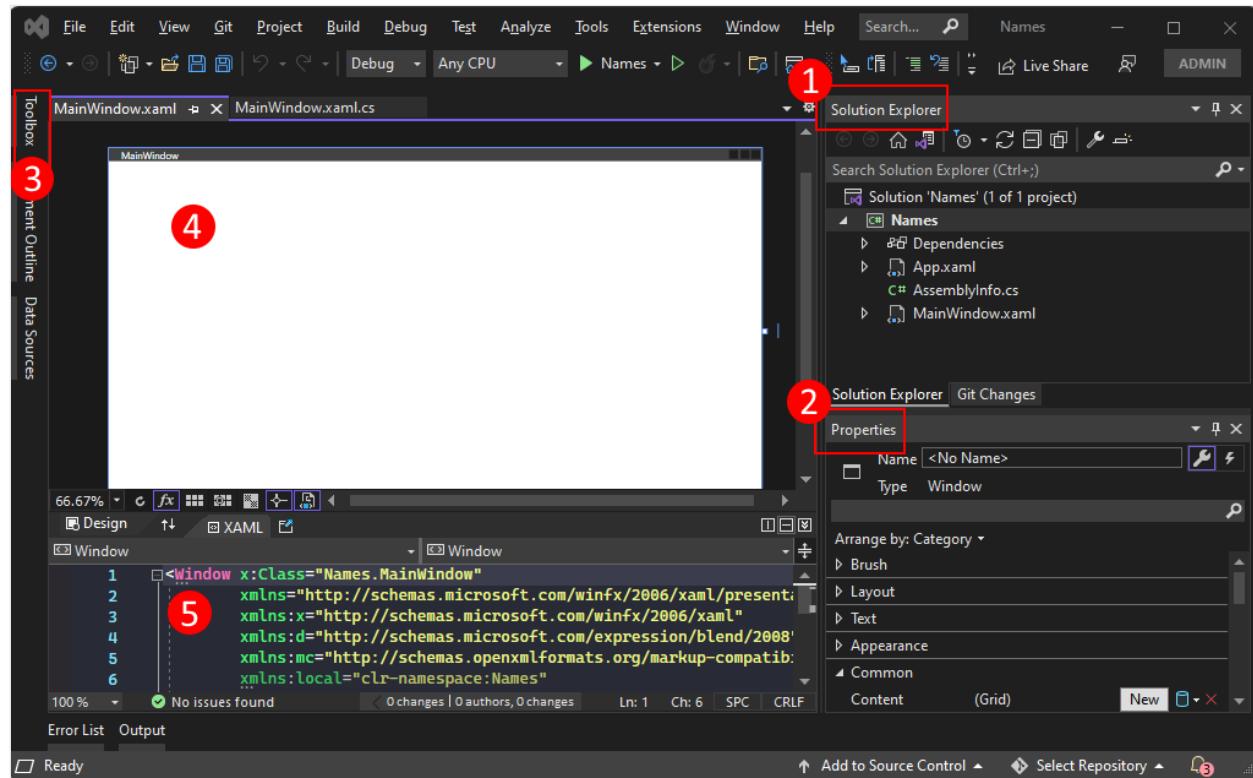
## 创建 WPF 应用

创建新应用的第一步是打开 Visual Studio 并通过模板生成应用。

生成应用后，Visual Studio 应会打开默认窗口 MainWindow 的 XAML 设计器窗格。如果看不到设计器，请在“解决方案资源管理器”窗格中双击 MainWindow.xaml 文件以打开设计器。

## Visual Studio 的重要部分

Visual Studio 中对 WPF 的支持包含五个在创建应用时你将与之交互的重要组件：



### 1. 解决方案资源管理器

所有项目文件、代码、窗口、资源都将显示在此窗格中。

### 2. 属性

此窗格显示可以基于所选项进行配置的属性设置。例如，如果从“解决方案资源管理器”中选择一个项，你会看到与该文件相关的属性设置。如果在“设计器”中选择一个对象，你会看到该项目的设置。

### 3. 工具箱

工具箱包含可添加到窗体的所有控件。若要将控件添加到当前窗体，请双击控件或拖放控件。

### 4. XAML 设计器

这是 XAML 文档的设计器。它是交互式的，可以从“工具箱”拖放对象。通过在设计器中选择和移动项，可以直观地为应用构建用户界面 (UI)。

当设计器和编辑器都可见时，对设计器的更改会反映在编辑器中，反之亦然。在设计器中选择项目时，“属性”窗格会显示有关该对象的属性和特性。

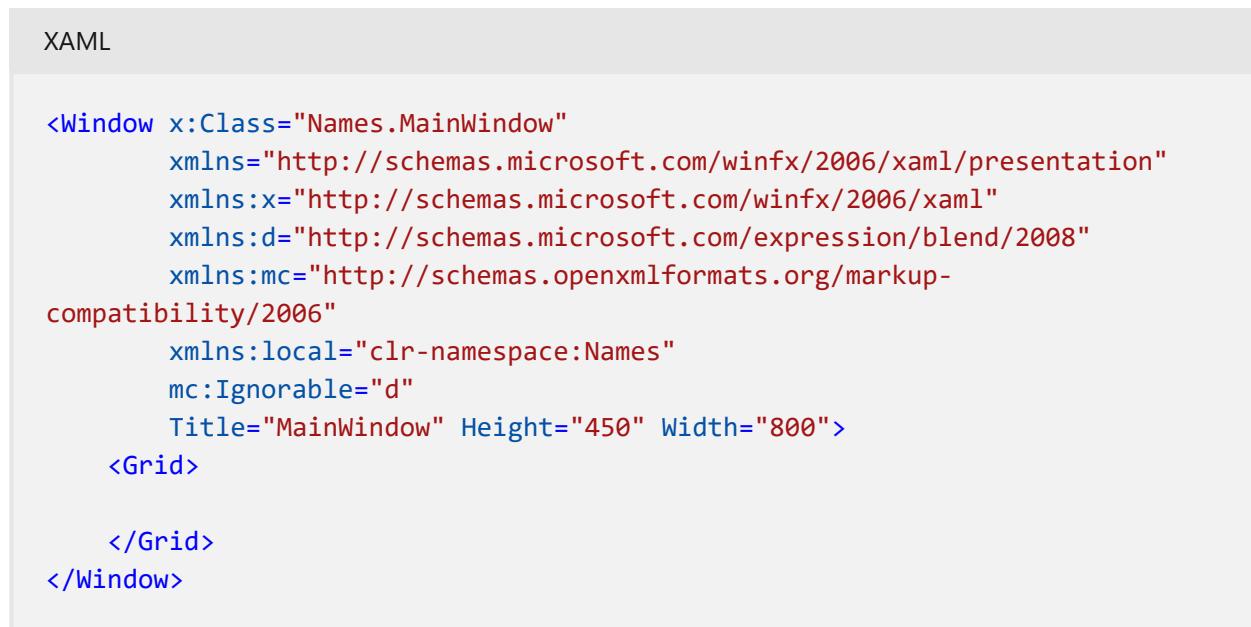
## 5. XAML 代码编辑器

这是 XAML 文档的 XAML 代码编辑器。XAML 代码编辑器是一种无需设计器即可手动创建 UI 的方法。将控件添加到设计器中时，设计器可以推断控件上的属性值。XAML 代码编辑器则为你提供更多控制权。

当设计器和编辑器都可见时，对设计器的更改会反映在编辑器中，反之亦然。在代码编辑器中导航文本插入点时，“属性”窗格会显示有关该对象的属性和特性。

# 检查 XAML

创建项目后，将显示 XAML 代码编辑器，并以最少的 XAML 代码显示窗口。如果编辑器未打开，请在“解决方案资源管理器”中双击“MainWindow.xaml”项目。你应该会看到类似于以下示例的 XAML：



The screenshot shows a code editor window titled "XAML". The code displayed is:

```
<Window x:Class="Names.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:local="clr-namespace:Names"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        </Grid>
</Window>
```

让我们分解此 XAML 代码，以便更好地了解它。XAML 就是可由 WPF 使用的编译器进行处理的 XML。它描述 WPF UI 并与 .NET 代码进行交互。若要了解 XAML，至少应熟悉 XML 的基础知识。

文档根 `<Window>` 表示 XAML 文件描述的对象的类型。它声明八个特性，这些特性通常分为三类：

- 命名空间

XML 命名空间为 XML 提供结构，确定可在文件中声明的 XML 内容。

主要 `xmlns` 特性将导入整个文件的 XML 命名空间，在本例中，将映射到 WPF 声明的类型。其他 XML 命名空间声明一个前缀，并导入 XAML 文件的其他类型和对象。例如，`xmlns:local` 命名空间声明 `local` 前缀，并映射到项目声明的对象，即，在 `Names` 代码命名空间中声明的对象。

- `x:Class` 属性

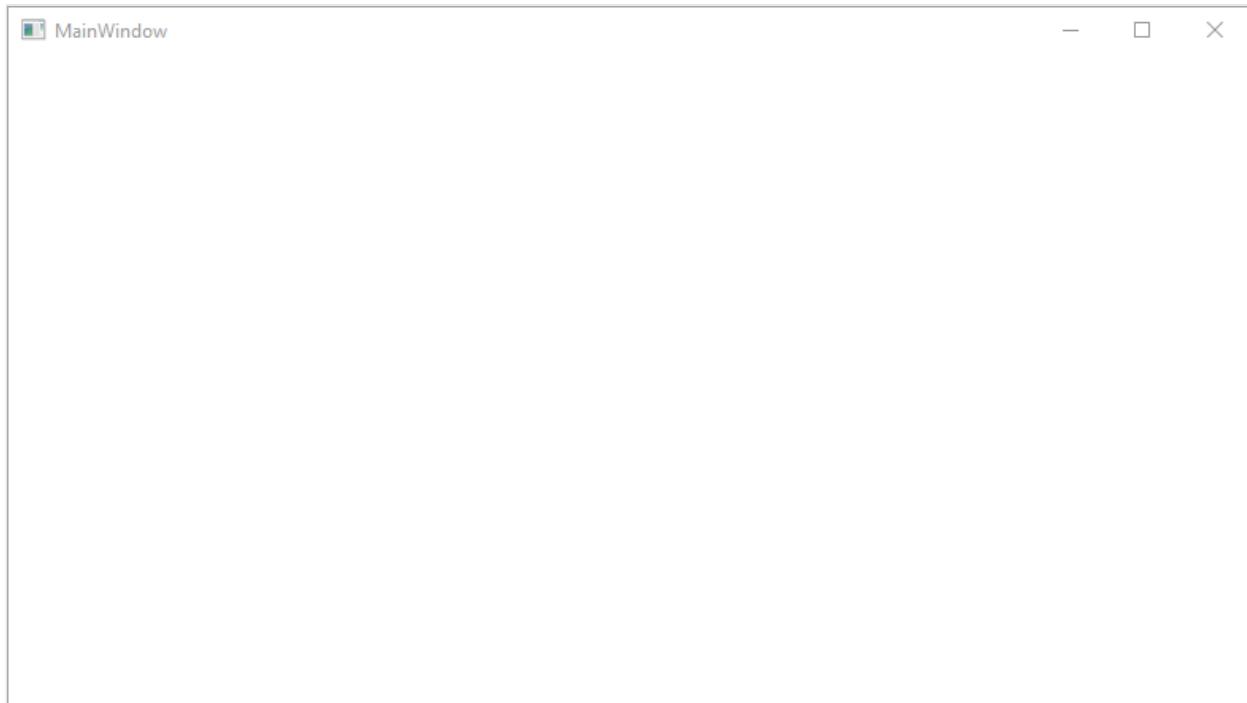
此特性将 `<Window>` 映射到代码定义的类型：`MainWindow.xaml.cs` 或 `MainWindow.xaml.vb` 文件，即 `Names.MainWindow` 类。

- `Title` 属性

在 XAML 对象上声明的任何常规特性都会设置该对象的属性。在本例中，`Title` 特性将设置 `Window.Title` 属性。

## 更改窗口

首先，运行项目并查看默认输出。你会看到一个没有任何控件且标题为 `MainWindow` 的弹出窗口：



对于我们的示例应用，此窗口太大，并且标题栏不是描述性的。通过将 XAML 中的相应特性更改为以下值，来更改窗口的标题和大小：

### XAML

```
<Window x:Class="Names.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:local="clr-namespace:Names"
    mc:Ignorable="d"
    Title="Names" Height="180" Width="260">
<Grid>

    </Grid>
</Window>
```

## 准备布局

WPF 提供一个功能强大的布局系统，其中包含许多不同的布局控件。 布局控件可帮助放置子控件和调整其大小，甚至可以自动执行这些操作。 此 XAML 中提供给你的默认布局控件是 `<Grid>` 控件。

通过使用 `Grid` 控件，可以像定义表一样定义行和列，并将控件置于特定行和列组合的边界内。 你可以将任意数量的子控件或其他布局控件添加到 `Grid` 中。 例如，你可以在特定的行和列组合中放置另一个 `Grid` 控件，然后新的 `Grid` 可以定义更多的行和列，并拥有自己的子级。

`<Grid>` 控件定义控件所在的行和列。 网格始终只声明单行和单列，这意味着默认情况下，网格就是一个单元格。 这并不能让你真正灵活地放置控件。

在添加新的行和列之前，请向 `<Grid>` 元素添加一个新特性：`Margin="10"`。 这样就可以从窗口中插入网格，使它看起来更漂亮一些。

接下来，定义两行两列，将网格划分为四个单元格：

XAML

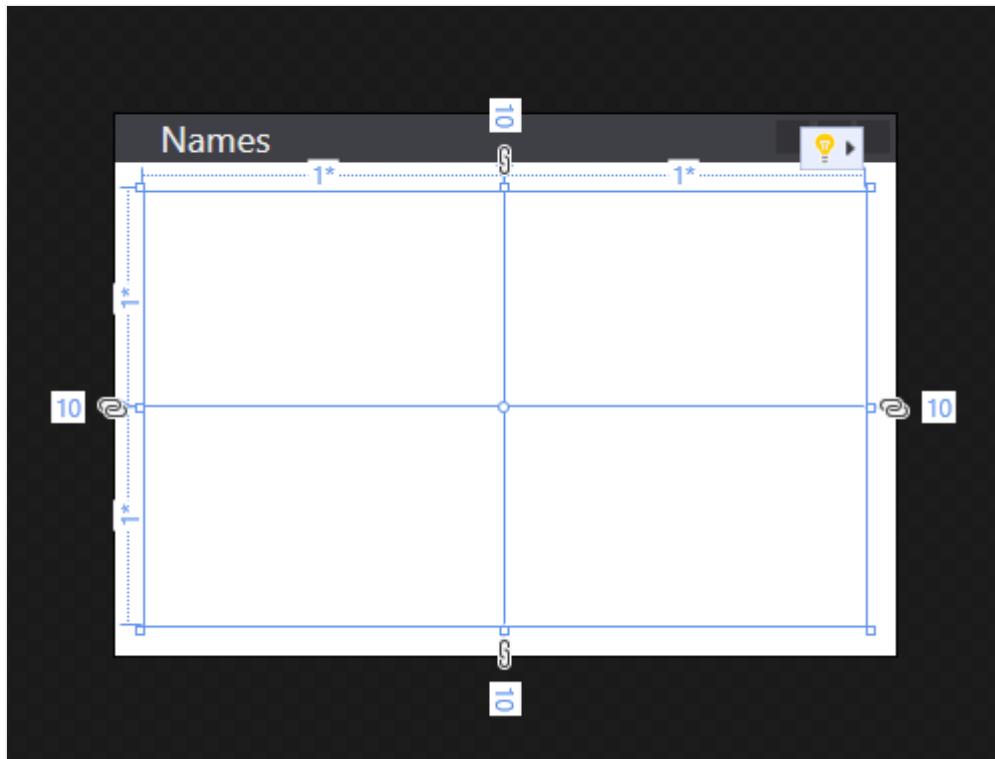
```
<Grid Margin="10">

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

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

</Grid>
```

在 XAML 代码编辑器或 XAML 设计器中选择网格，你将看到 XAML 设计器显示每一行和每一列：



## 添加第一个控件

现在已经创建了网格，接下来可以开始向其添加控件。首先，从标签控件开始。在 `<Grid>` 元素内的行定义和列定义后面创建一个新的 `<Label>` 元素，并为其指定字符串值 Names：

XAML

```
<Grid Margin="10">

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

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

    <Label>Names</Label>

</Grid>
```

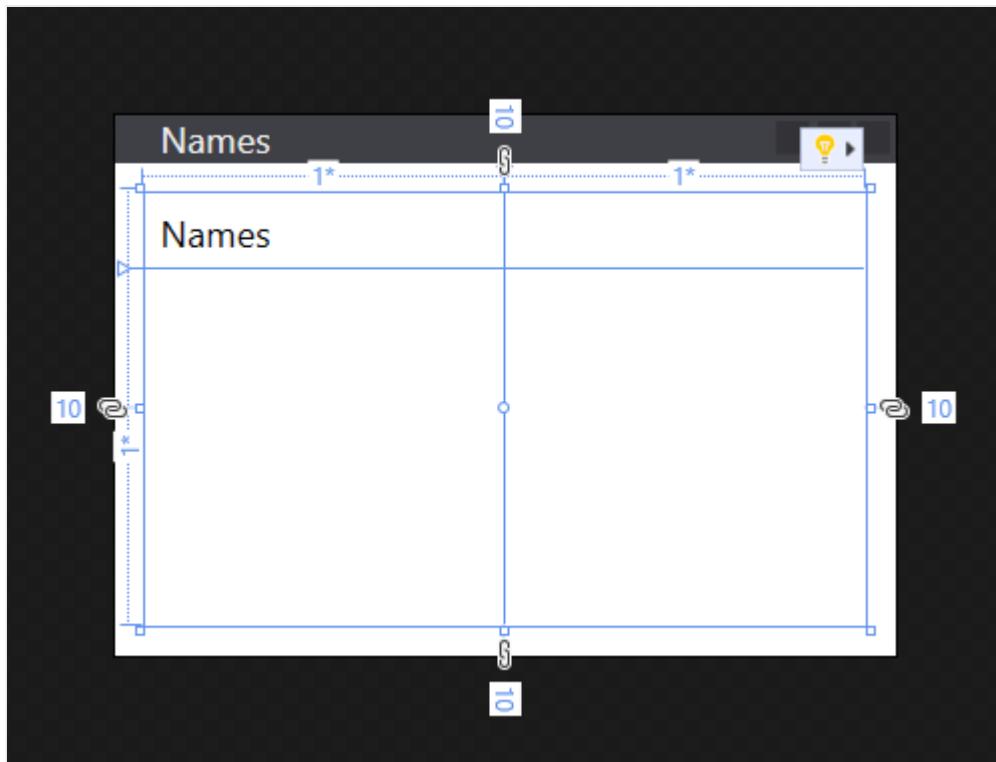
`<Label>Names</Label>` 定义内容 `Names`。有些控件知道如何处理内容，有些则不知道。控件的内容映射到 `Content` 属性。通过 XAML 特性语法设置内容时，将使用以下格式：`<Label Content="Names" />`。这两种方法可以实现相同的目的，即，将标签内容设置为显示文本 `Names`。

不过，我们有一个问题，由于标签是自动分配给网格的第一行和第一列的，因此它占据了半个窗口。对于第一行，我们不需要那么多空间，因为我们只需要在这一行放置标签。将第一个 `<RowDefinition>` 的 `Height` 特性从 `*` 更改为 `Auto`。`Auto` 值会自动将网格行的大小调整为其内容（在本例中为标签控件）的大小。

XAML

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

请注意，设计器现在显示标签占据的可用高度较少。现在，下一行有更多空间可用。大多数控件会定义应该占用的最适合自己的某种高度和宽度值。例如，标签控件有一个高度值，确保你可以读取它。



## 控件放置

让我们谈谈控件的放置。上一部分中创建的标签已自动放置在网格的第 0 行和第 0 列。行和列的编号从 0 开始，每新增一行或一列，编号就递增 1。控件无法识别网格，并且

不会定义任何属性来控制其在网格中的位置。控件甚至可能放置在其他布局控件中，后者有自己的一组规则来定义如何放置控件。

当控件无法识别网格时，如何告诉控件使用其他行或列？附加属性！网格采用 WPF 提供的强大属性系统。网格定义了子控件可以声明和使用的新属性。控件本身其实并不存在这些属性，它们是在将控件添加到网格时由网格附加的。

网格定义两个属性来确定子控件的行和列位置：`Grid.Row` 和 `Grid.Column`。如果控件中省略了这些属性，则意味着它们的默认值为 0，因此，控件将放置在网格的第 0 行和第 0 列。尝试通过将 `Grid.Column` 属性设置为 1 来更改 `<Label>` 控件的位置：

XAML

```
<Label Grid.Column="1">Names</Label>
```

注意标签现在如何移动到第二列。你可以使用 `Grid.Row` 和 `Grid.Column` 附加属性来放置我们接下来要创建的控件。不过现在，请将标签还原到第 0 行。

## 创建名称列表框

现在已经正确调整了网格的大小并创建了标签，接下来，在标签下方的行中添加一个列表框控件。列表框将位于第 1 行和第 0 列。我们还将此控件命名为 `lstNames`。为控件命名后，即可在代码隐藏中对其进行引用。该名称通过 `x:Name` 特性分配给控件。

XAML

```
<Grid Margin="10">

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

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

    <Label>Names</Label>
    <ListBox Grid.Row="1" x:Name="lstNames" />

</Grid>
```

## 添加其余控件

我们将添加的最后两个控件是一个文本框和一个按钮，用户将使用它们来输入要添加到列表框中的名称。但是，我们没有尝试为网格创建更多行和列，而是将这些控件放入 `<StackPanel>` 布局控件中。

堆叠面板与网格的不同之处在于控件的放置方式。你告诉网格你希望使用 `Grid.Row` 和 `Grid.Column` 附加属性将控件放置在哪个位置，堆叠面板则自动按以下方式运行：先放置第一个控件，然后将下一个控件置于其后，一直到所有控件都放置完毕。它将每个控件“堆叠”在另一个控件之下。

在列表框后创建 `<StackPanel>` 控件，并将其放在网格的第 1 行、第 1 列。另外添加一个名为 `Margin` 且值为 `5,0,0,0` 的特性：

XAML

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

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

<Label>Names</Label>
<ListBox Grid.Row="1" x:Name="lstNames" />

<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
</StackPanel>
```

之前在网格上使用了 `Margin` 特性，但我们只输入了一个值 (10)。现在，我们在堆叠面板上使用了值 `5,0,0,0`。边距是 `Thickness` 类型，可以解释这两个值。粗细定义矩形框每条边（分别为左、顶、右、底）周围的空间。如果边距的值是单一值，则四条边均使用该值。

接下来，在 `<StackPanel>` 中创建 `<TextBox>` 和 `<Button>` 控件。

XAML

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0">Add Name</Button>
</StackPanel>
```

窗口的布局已完成。但是，我们的应用不包含任何逻辑，无法真正发挥作用。接下来，我们需要将控件事件挂钩到代码，让应用能够实际派上用场。

## 为 Click 事件添加代码

我们创建的 `<Button>` 具有一个 `Click` 事件，该事件在用户按下按钮时引发。你可以订阅此事件并添加代码，以便向列表框添加名称。就像通过添加 XAML 特性在控件上设置属性一样，你可以使用 XAML 特性来订阅事件。将 `Click` 特性设置为

`ButtonAddName_Click`

XAML

```
<StackPanel Grid.Row="1" Grid.Column="1" Margin="5,0,0,0">
    <TextBox x:Name="txtName" />
    <Button x:Name="btnAdd" Margin="0,5,0,0" Click="ButtonAddName_Click">Add
        Name</Button>
</StackPanel>
```

现在，你需要生成处理程序代码。右键单击 `ButtonAddName_Click`，然后选择“转到定义”。此操作将在代码隐藏中为你生成一个与你输入的处理程序名称匹配的方法。

C#

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{}
```

接下来，添加以下代码以执行这三个步骤：

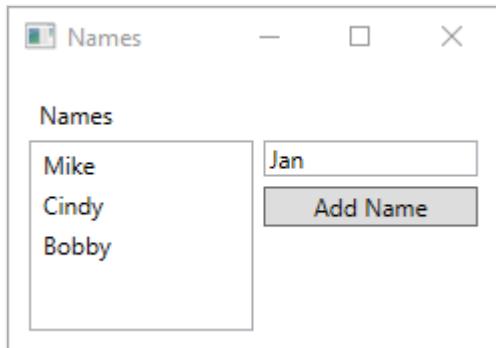
1. 确保文本框包含名称。
2. 验证文本框中输入的名称是否已经存在。
3. 将名称添加到列表框。

C#

```
private void ButtonAddName_Click(object sender, RoutedEventArgs e)
{
    if (!string.IsNullOrWhiteSpace(txtName.Text) &&
        !lstNames.Items.Contains(txtName.Text))
    {
        lstNames.Items.Add(txtName.Text);
        txtName.Clear();
    }
}
```

# 运行应用

现在已对事件进行编码，可以通过按 F5 键或从菜单中选择“调试”>“开始调试”来运行应用。随即显示窗口，可以在文本框中输入名称，然后通过单击按钮添加该名称。



## 后续步骤

[详细了解 Windows Presentation Foundation](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

### .NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# .NET 8 (WPF .NET) 的新增功能

项目 • 2024/02/14

WPF 添加了硬件加速以及用于浏览和选择 .NET 8 中的文件夹的新控件。

## 硬件加速

以前，远程访问的所有 WPF 应用程序都必须使用软件呈现，即使系统具有硬件呈现功能也是如此。.NET 8 添加了一个选项，它让你能够选择使用远程桌面协议 (RDP) 的硬件加速。

硬件加速是指使用计算机的图形处理单元 (GPU) 加快应用程序中图形和视觉效果的呈现。这可以提高性能并实现更无缝的响应式图形。相比之下，软件呈现只依赖于计算机的中央处理单元 (CPU) 来呈现图形，这可能会拖慢速度和效果。

若要选择使用，请在 runtimeconfig.json 文件中将

`Switch.System.Windows.Media.EnableHardwareAccelerationInRdp` 配置属性设置为 `true`。

有关详细信息，请参阅 [RDP 中的硬件加速](#)。

## OpenFileDialog

WPF 包含名为  [OpenFileDialog](#) 的新对话框控件。此控件允许用户浏览和选择文件夹。以前，应用开发人员依赖于第三方软件来实现此功能。

C#

```
var openFileDialog = new OpenFileDialog()
{
    Title = "Select folder to open ...",
    InitialDirectory = Environment.GetFolderPath(
        Environment.SpecialFolder.ProgramFiles)
};

string folderName = "";
if (openFileDialog.ShowDialog())
{
    folderName = openFileDialog.FolderName;
}
```

有关详细信息，请参阅 [.NET 8 中的 WPF 文件对话框改进 \(.NET 博客\)](#) ↗。

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback 反馈

.NET Desktop feedback 是一个开放源代码项目。选择一个链接以提供反馈：

 提出文档问题

 提供产品反馈

# .NET 7 (WPF .NET) 的新增功能

项目 • 2023/10/19

本文介绍 .NET 7 中一些新的 Windows Presentation Foundation 功能和增强功能。

## 性能改进

适用于 .NET 7 的 WPF 中的许多改进都集中在性能上，例如：

- 尽可能避免装箱和取消装箱。
- 避免对堆进行不必要的对象分配。
- 重用 `StringBuilder` 的实例，而不是创建新实例。
- 在不需要时停止使用 `StringBuilder`。

有关显著更改的列表，请参阅 [.NET 博客 - .NET 7 中 WPF 的新增功能](#)。

## 辅助功能改进和修复

添加了控件的其他键盘交互：

- `DataGridView` 列宽可以使用 `ALT` + 左/右箭头进行调整。
- 为 `DataGrid` 启用排序后，可以使用 `F3` 对列进行排序。
- 现在，可以使用屏幕上的讲述人正确宣读可选中的菜单项。

## Bug 修复

虽然 WPF 在 .NET Framework 上仍然完全受支持和享受服务，但大多数修补程序和所有新功能只适用于 .NET，在那里我们有机会做出更大的更改。WPF 社区帮助解决了此版本中一些长期存在的 bug：

- 无法全局覆盖 `FocusVisualStyle`
- `CommandParameter` 使 `CanExecute` 无效
- .NET 6 工具提示行为从 .NET 5 更改 (bug?)
- `Comboboxitem` 工具提示 bug
- 如果从可视化树中删除所有者，`ContextMenu` 将停止工作
- 修复了 `glyphrun` 序列化时的舍入错误

社区提供了更多 bug 修复，其中许多修复已在 [.NET 博客](#) 上记录。

## 另请参阅

- .NET 博客 - .NET 7 中 WPF 的新增功能 ↗
- .NET 社区工具包

## ⌚ 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

⌚ 提出文档问题

↗ 提供产品反馈

# 如何将 WPF 桌面应用升级到 .NET 8

项目 • 2024/02/07

本文介绍如何将 Windows Presentation Foundation (WPF) 桌面应用升级到 .NET 8。尽管 WPF 在 .NET 这一跨平台技术上运行，但它仍然是一个仅限 Windows 的框架。可以使用 .NET 升级助手升级以下与 WPF 相关的项目类型：

- WPF 项目
- 控件库
- .NET 库

如果要从 .NET Framework 升级到 .NET，不妨查看[与 WPF .NET 的差异](#)文章和[从 .NET Framework 移植到 .NET](#)指南。

## 先决条件

- Windows 操作系统
- 面向 .NET 8 的 Visual Studio 2022 17.7 或更高版本 ↗
- 面向 .NET 7 的 Visual Studio 2022 17.1 或更高版本 ↗
- 适用于 Visual Studio 的 .NET 升级助手扩展

## 演示应用

本文是在升级 Web 收藏夹示例项目的背景下编写的，可以从[.NET 示例 GitHub 存储库](#) ↗ 下载该项目。

## 启动升级

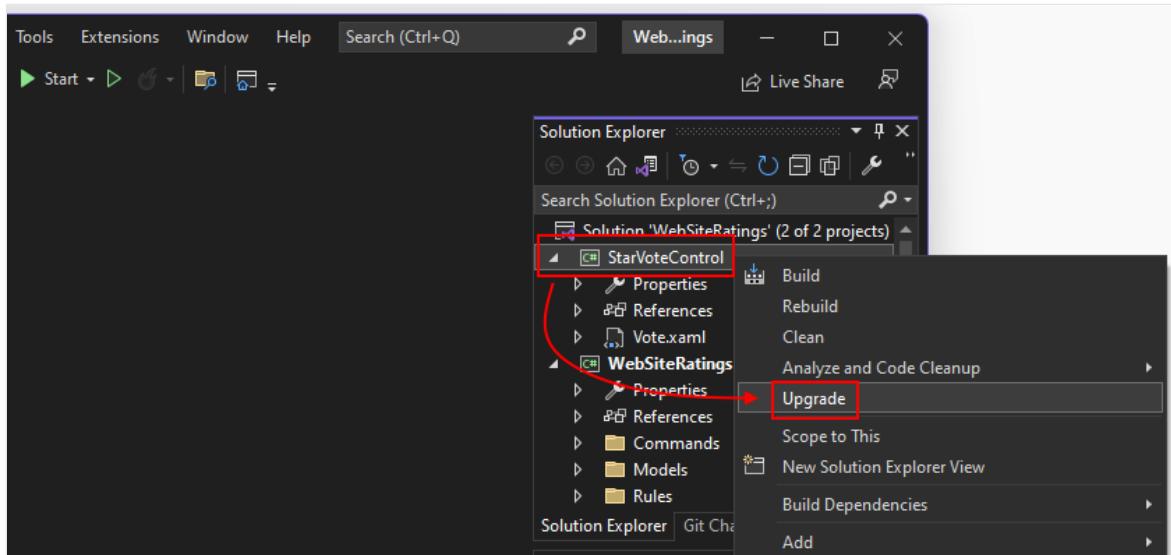
如果要升级多个项目，请从没有依赖项的项目开始。在 Web 收藏夹示例中，**WebSiteRatings** 项目依赖于 **StarVoteControl** 库，因此应首先升级 **StarVoteControl**。

### 💡 提示

请务必对代码进行备份，例如在源代码管理中或副本中。

使用以下步骤在 Visual Studio 中升级项目：

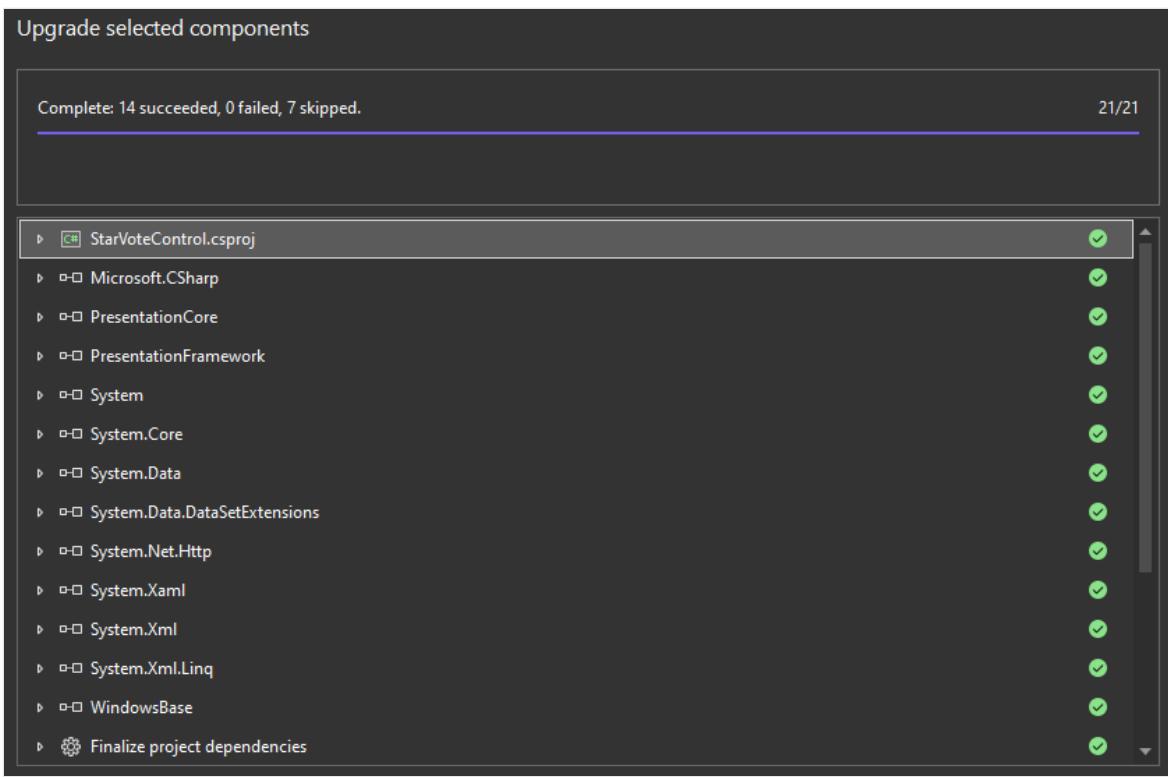
1. 右键单击“解决方案资源管理器”窗口中的 **StarVoteControl** 项目，然后选择“**升级**”：



将打开一个新选项卡，提示你选择执行升级的方式。

2. 选择“**就地项目升级**”。
3. 接下来，选择目标框架。根据要升级的项目类型，会显示不同的选项。如果该库不依赖于 WPF 等桌面技术并且可以由 .NET Framework 项目和 .NET 项目使用，那么 **.NET Standard 2.0** 是一个不错的选择。但是，最新的 .NET 版本相对于 .NET Standard 提供了许多语言和编译器改进。  
选择“.NET 8.0”，然后选择“下一步”。
4. 此时将显示一个树状结构，其中包含与项目相关的所有工件，例如代码文件和库。可以升级单个生成生成工件或整个项目（这是默认设置）。选择“**升级选择**”以开始升级。

升级完成后，将显示结果：



具有实心绿色圆圈的生成工件已升级，而具有空绿色圆圈的生成工件则被跳过。跳过的生成工件意味着升级助手没有找到任何可以升级的内容。

现在应用程序的支持库已升级，请升级主应用程序。

## 升级应用

升级所有支持库后，即可升级主应用项目。执行以下步骤：

- 右键单击“**解决方案资源管理器**”窗口中的 **WebSiteRatings** 项目，然后选择“**升级**”。
- 选择“**就地项目升级**”作为升级模式。
- 选择“.NET 8.0”作为目标框架，然后选择“**下一步**”。
- 保留选中所有项目，然后选择“**升级选择**”。

升级完成后，将显示结果。如果某个项目有警告符号，则意味着有一条注释可供阅读，可以通过展开该项目来阅读。

## 生成干净的内部版本

项目升级后，清理并编译它。

- 右键单击“**解决方案资源管理器**”窗口中的 **WebSiteRatings** 项目，然后选择“**清理**”。
- 右键单击“**解决方案资源管理器**”窗口中的 **WebSiteRatings** 项目，然后选择“**生成**”。

如果应用程序遇到任何错误，可以在“**错误列表**”窗口中找到这些错误，并提供修复这些错误的建议。

# 升级后的步骤

如果你的项目正在从 .NET Framework 升级到 .NET，请查看[从 .NET Framework 升级到 .NET 后的现代化](#)一文中的信息。

升级后，需要：

- 检查 NuGet 包。

.NET 升级助手将一些包升级到了新版本。借助本文中提供的示例应用程序，`Microsoft.Data.Sqlite` NuGet 包已从 1.0.0 升级到 8.0.x。不过，1.0.0 依赖于 `SQLite` NuGet 包，但 8.0.x 移除了该依赖关系。项目仍会引用 `SQLite` NuGet 包，尽管已经不再需要它。可以从项目中移除 `SQLite` 和 `SQLite.Native` NuGet 包。

- 清理旧的 NuGet 包。

不再需要 `packages.config` 文件，可以将其从项目中删除，因为 NuGet 包引用现已在项目文件中声明。此外，名为 `Packages` 的本地 NuGet 包缓存文件夹位于项目的文件夹或父文件夹中。可以删除此本地缓存文件夹。新的 NuGet 包引用使用包的全局缓存文件夹，该文件夹位于用户的配置文件目录中，名为 `.nuget\packages`。

- 移除 `System.Configuration` 库。

大多数 .NET Framework 应用都会引用 `System.Configuration` 库。升级后，有可能还是直接引用这个库。

`System.Configuration` 库使用 `app.config` 文件为应用程序提供运行时配置选项。对于 .NET，此库已替换为 `System.Configuration.ConfigurationManager` NuGet 包。移除对库的引用，并将 NuGet 包添加到项目。

- 检查对应用进行现代化改造的位置。

自 .NET 发布以来，API 和库发生了很大变化。在大多数情况下，.NET Framework 无法访问这些改进。通过升级到 .NET，现在可以访问更现代的库。

后续部分将介绍对本文使用的示例应用程序进行现代化改造的方面。

## 现代化：Web 浏览器控件

WPF 示例应用所引用的 `WebBrowser` 控件基于已过时的 Internet Explorer。WPF for .NET 可以使用基于 Microsoft Edge 的 `WebView2` 控件。完成以下步骤以升级到新的 [WebView2 Web 浏览器控件](#)：

1. 添加 `Microsoft.Web.WebView2` NuGet 包。

2. 在 MainWindow.xaml 文件中：

a. 将控件导入根元素中的 wpfControls 命名空间：

```
XAML

<mah:MetroWindow x:Class="WebSiteRatings.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/expressionblend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:mah="clr-
namespace:MahApps.Metro.Controls;assembly=MahApps.Metro"
    xmlns:local="clr-namespace:WebSiteRatings"
    xmlns:vm="clr-namespace:WebSiteRatings.ViewModels"
    xmlns:VoteControl="clr-
namespace:StarVoteControl;assembly=StarVoteControl"
    xmlns:wpfControls="clr-
namespace:Microsoft.Web.WebView2.Wpf;assembly=Microsoft.Web.WebView2
.Wpf"
        Loaded="MetroWindow_Loaded"
        mc:Ignorable="d"
        Title="My Sites" Height="650" Width="1000">
```

b. 在声明 `<Border>` 元素的位置下，删除 `WebBrowser` 控件并将其替换为 `wpfControls:WebView2` 控件：

```
XAML

<Border Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"
BorderThickness="1" BorderBrush="Black" Margin="5">
    <wpfControls:WebView2 x:Name="browser"
    ScrollViewer.CanContentScroll="True" />
</Border>
```

3. 编辑 MainWindow.xaml.cs 代码隐藏文件。更新 `ListBox_SelectionChanged` 方法以将 `browser.Source` 属性设置为有效的 `Uri`。此代码以前作为字符串传入网站 URL，但 `WebView2` 控件需要使用 `Uri`。

C#

```
private void ListBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    var siteCollection = (ViewModels.SiteCollection)DataContext;

    if (siteCollection.SelectedSite != null)
```

```
        browser.Source = new Uri(siteCollection.SelectedSite.Url);
    else
        browser.NavigateToString("<body></body>");
}
```

根据你的应用的用户运行的 Windows 版本，他们可能需要安装 WebView2 运行时。有关详细信息，请参阅 [WPF 应用中的 WebView2 入门](#)。

## 现代化：appsettings.json

.NET Framework 使用 App.config 文件为你的应用加载设置，例如连接字符串和日志记录提供程序。.NET 现在使用 appsettings.json 文件进行应用设置。通过 `System.Configuration ConfigurationManager` NuGet 包在 .NET 中支持 App.config 文件，并且通过 `Microsoft.Extensions.Configuration` NuGet 包提供对 appsettings.json 的支持。

随着其他库升级到 .NET，它们将通过支持 *appsettings.json* 而不是 *App.config* 实现新式化。例如，已针对 .NET 6+ 升级的 .NET Framework 中的日志记录提供程序将不再使用 *App.config* 进行设置。最好遵循它们的指示，并尽可能不要使用 *App.config*。

## 将 appsettings.json 与 WPF 示例应用配合使用

例如，升级 WPF 示例应用程序后，使用 *appsettings.json* 作为本地数据库的连接字符串。

1. 删除 `System.Configuration ConfigurationManager` NuGet 包。
2. 添加 `Microsoft.Extensions.Configuration.Json` NuGet 包。
3. 将文件添加到名为 *appsettings.json* 的项目。
4. 将 *appsettings.json* 文件设置为复制到输出目录。

在解决方案资源管理器中选择文件后，使用“属性”窗口通过 Visual Studio 设置“**复制到输出**”设置。或者，可以直接编辑项目并添加以下 `ItemGroup`：

XML

```
<ItemGroup>
    <Content Include="appsettings.json">
        <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
</ItemGroup>
```

5. 将 *App.config* 文件中的设置迁移到新的 *appsettings.json* 文件。

在 WPF 示例应用程序中，`app.config` 仅包含一个连接字符串。 编辑 `appsettings.json` 文件以定义连接字符串：

```
JSON

{
  "ConnectionStrings": {
    "database": "DataSource=sqlite.db;"
  }
}
```

6. 编辑 `App.xaml.cs` 文件，实例化加载 `appsettings.json` 文件的配置对象，添加的行会突出显示：

```
C#

using System.Windows;
using Microsoft.Extensions.Configuration;

namespace WebSiteRatings
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    public partial class App : Application
    {
        public static IConfiguration Config { get; private set; }

        public App()
        {
            Config = new ConfigurationBuilder()
                .AddJsonFile("appsettings.json")
                .Build();
        }
    }
}
```

7. 在 `.\Models\Database.cs` 文件中，更改 `OpenConnection` 方法以使用新的 `App.Config` 属性。 这需要导入 `Microsoft.Extensions.Configuration` 命名空间：

```
C#

using Microsoft.Data.Sqlite;
using System.Collections.Generic;
using Microsoft.Extensions.Configuration;

namespace WebSiteRatings.Models
{
    internal class Database
    {
```

```
public static SqliteConnection OpenConnection() =>
    new
    SqliteConnection(App.Config.GetConnectionString("database"));

    public static IEnumerable<Site> ReadSites()
```

`GetConnectionString` 是 `Microsoft.Extensions.Configuration` 命名空间提供的扩展方法。

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback 反馈

.NET Desktop feedback 是一个开放源代码项目。选择一个链接以提供反馈：

 提出文档问题

 提供产品反馈

# 与 WPF .NET 的差异

项目 • 2023/10/20

本文介绍了 .NET 与 .NET Framework 上的 Windows Presentation Foundation (WPF) 之间的差异。WPF for .NET 是一个[开放源代码框架](#)，它从原始 WPF for .NET Framework 源代码派生而来。

.NET Framework 有一些 .NET 不支持的功能。关于不支持的技术的详细信息，请参阅[在 .NET 上不可用的 .NET Framework 技术](#)。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

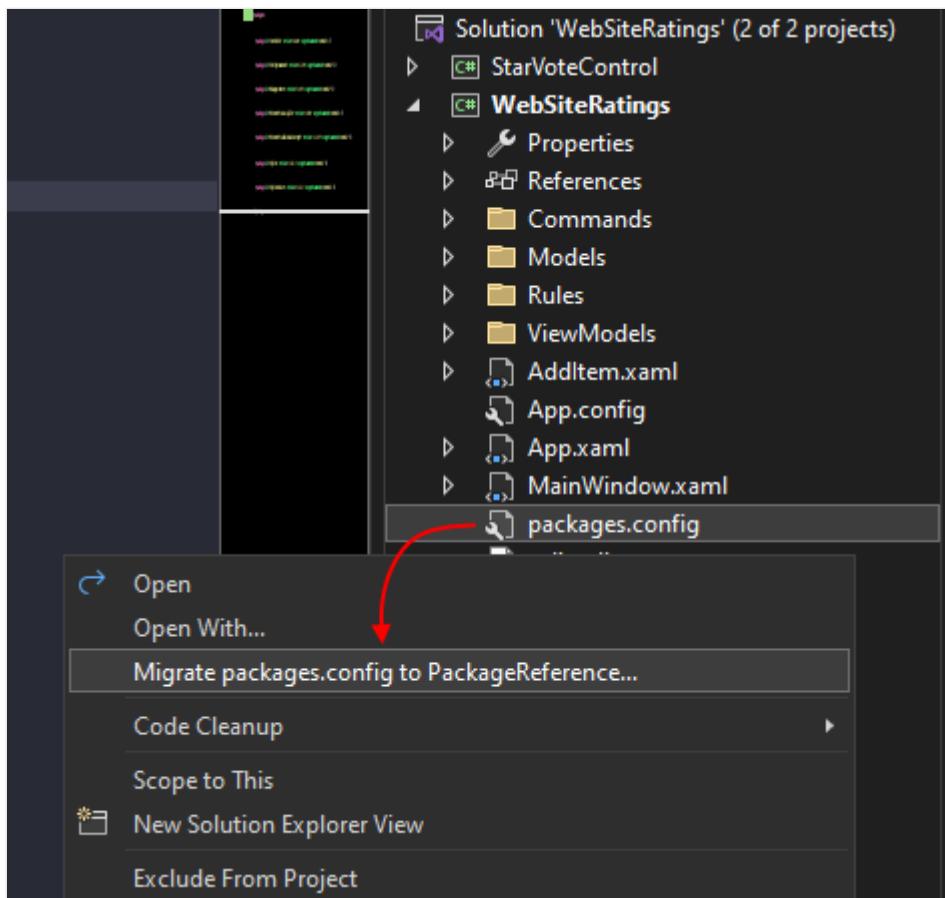
## SDK 样式项目

.NET 使用 SDK 样式的项目文件。这些项目文件与 Visual Studio 管理的传统 .NET Framework 项目文件不同。要将 .NET Framework WPF 应用迁移到 .NET，必须转换项目。有关详细信息，请参阅[如何将 WPF 桌面应用升级到 .NET 7](#)。

## NuGet 包引用

如果 .NET Framework 应用在 packages.config 文件中列出其 NuGet 依赖项，请迁移为 [`<PackageReference>`](#) 格式：

1. 在 Visual Studio 中，打开“解决方案资源管理器”窗格。
2. 在 WPF 项目中，右键单击“packages.config”>“将 packages.config 迁移到 PackageReference”。



将出现一个对话框，显示计算出的顶级 NuGet 依赖关系，并询问应将其他哪些 NuGet 包提升到顶级。选择“确定”，然后将“packages.config”文件从项目中删除，并将 `<PackageReference>` 元素添加到项目文件中。

当项目使用 `<PackageReference>` 时，包不会存储在本地“Packages”文件夹中，而是全局存储。打开项目文件，删除任何引用到“Packages”文件夹的 `<Analyzer>` 元素。这些分析器会自动包含在 NuGet 包引用中。

## 代码访问安全性

.NET 不支持代码访问安全性 (CAS)。在完全信任的前提下，处理所有与 CAS 相关的功能。WPF for .NET 会删除与 CAS 相关的代码。这些类型的公共 API 表面仍然存在，以确保对这些类型的调用成功。

公开定义的 CAS 相关类型被移出 WPF 程序集，并移入 Core .NET 库程序集中。WPF 程序集将类型转发设置为移动类型的新位置。

源程序集	目标程序集	类型
<code>WindowsBase.dll</code>	<code>System.Security.Permissions.dll</code>	<code>MediaPermission</code>
		<code>MediaPermissionAttribute</code>
		<code>MediaPermissionAudio</code>
		<code>MediaPermissionImage</code>
		<code>MediaPermissionVideo</code>

源程序集	目标程序集	类型
		WebBrowserPermission WebBrowserPermissionAttribute WebBrowserPermissionLevel
System.Xaml.dll	System.Security.Permissions.dll	XamlLoadPermission
System.Xaml.dll	System.Windows.Extension.dll	XamlAccessLevel

## ① 备注

为了最大限度地减少移植摩擦，`XamlAccessLevel` 类型中保留了用于存储和检索与以下属性有关的信息的功能。

- `PrivateAccessToTypeName`
- `AssemblyNameString`

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# WPF 窗口概述 (WPF .NET)

项目 • 2023/10/13

用户通过窗口与 Windows Presentation Foundation (WPF) 应用程序交互。 窗口的主要用途是托管使数据可视化并使用户能够与数据交互的内容。 WPF 应用程序使用 [Window](#) 类提供自己的窗口。 本文先介绍 [Window](#)，然后讲述在应用程序中创建和管理窗口的基础知识。

## ① 重要

本文使用从 C# 项目生成的 XAML。 如果使用 Visual Basic，则 XAML 看上去可能有所不同。 这些差异通常出现在 `x:Class` 属性值上。 C# 包括项目的根命名空间，而 Visual Basic 不包括。

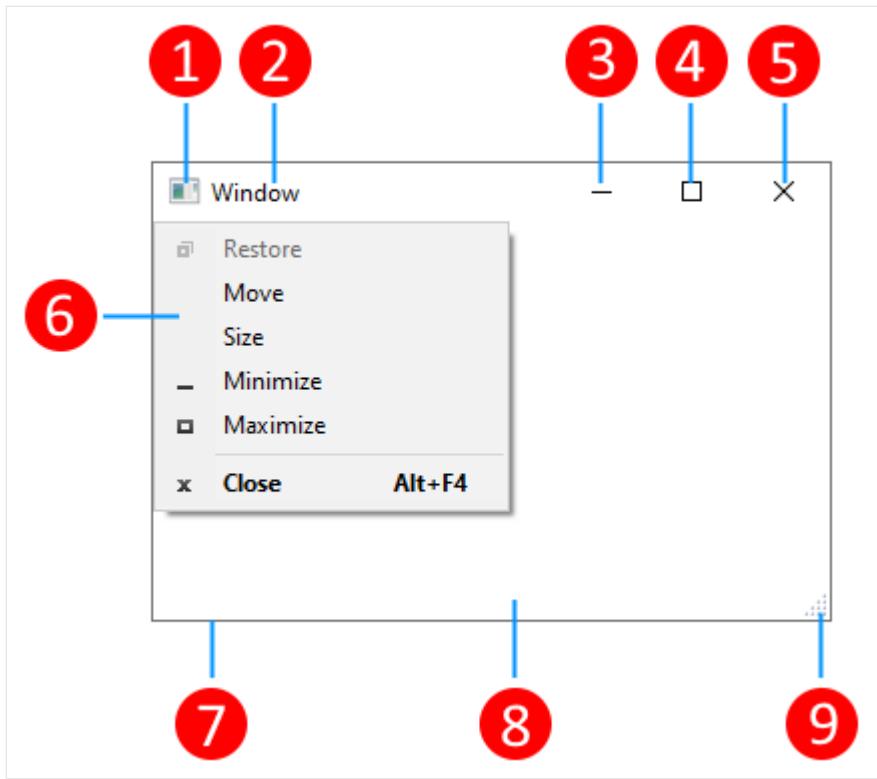
C# 的项目模板创建了一个 `App` 类型，包含在 `app.xaml` 文件中。 在 Visual Basic 中，该类型名为 `Application`，该文件名为 `Application.xaml`。

## 窗口类

在 WPF 中，窗口由用于执行以下操作的 [Window](#) 类封装：

- 显示窗口。
- 配置窗口的大小、位置和外观。
- 托管特定于应用程序的内容。
- 管理窗口的生存期。

下图展示了窗口的构成部分：



窗口分为两个区域：非工作区和工作区。

窗口的非工作区由 WPF 实现，它包括大多数窗口所共有的窗口部分，包括：

- 标题栏 (1-5)。
- 图标 (1)。
- 标题 (2)。
- 最小化 (3)、最大化 (4) 和关闭 (5) 按钮。
- 包含菜单项的系统菜单 (6)。单击图标 (1) 时出现。
- 边框 (7)。

窗口的工作区是窗口的非工作区内部的区域，由开发人员用于添加特定于应用程序的内容，例如菜单栏、工具栏和控件。

- 工作区 (8)。
- 大小调整手柄 (9)。这是添加到工作区 (8) 的控件。

## 实现窗口

典型窗口的实现既包括外观又包括行为，外观定义用户看到的窗口的样子，行为定义用户与之交互时窗口的运行方式。在 WPF 中，可以使用代码或 XAML 标记实现窗口的外观和行为。

但在一般情况下，窗口的外观使用 XAML 标记实现，行为使用代码隐藏实现，如以下示例所示。

## XAML

```
<Window x:Class="WindowsOverview.Window1"
    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:local="clr-namespace:WindowsOverview"
    >

    <!-- Client area containing the content of the window -->

</Window>
```

下面的代码是 XAML 的代码隐藏。

## C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```

若要使 XAML 标记文件和代码隐藏文件配合工作，需要满足以下要求：

- 在标记中，`Window` 元素必须包含 `x:Class` 属性。生成应用程序时，标记文件中存在 `x:Class` 会使 Microsoft 生成引擎 (MSBuild) 生成派生自 `Window` 的 `partial` 类，其名称由 `x:Class` 属性指定。这要求为 XAML 架构 (`xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"`) 添加 XAML 命名空间声明。生成的 `partial` 类实现 `InitializeComponent` 方法，注册事件和设置在标记中实现的属性时将调用此方法。
- 在代码隐藏中，类必须是 `partial` 类、名称必须是标记中 `x:Class` 属性指定的相同名称，并且它必须派生自 `Window`。这样，代码隐藏文件就与应用程序生成时为标记文件生成的 `partial` 类相关联（请参阅[编译 WPF 应用程序](#)以了解详细信息）。
- 在代码隐藏中，`Window` 类必须实现调用 `InitializeComponent` 方法的构造函数。`InitializeComponent` 由标记文件已生成的 `partial` 类实现，用以注册事件并设置

标记中定义的属性。

## ① 备注

使用 Visual Studio 将新的 Window 添加到项目时，Window 通过同时使用标记和代码隐藏实现，并且包括必要的配置来创建此处所述的标记文件和代码隐藏文件之间的关联。

进行了此配置后，可以专注于在 XAML 标记中定义窗口的外观，并可在代码隐藏中实现窗口的行为。以下示例显示了一个窗口，该窗口中的一个按钮定义了 Click 事件的事件处理程序。这是在 XAML 中实现的，处理程序是在代码隐藏中实现的。

### XAML

```
<Window x:Class="WindowsOverview.Window1"
        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:local="clr-namespace:WindowsOverview"
        >

    <!-- Client area containing the content of the window -->

    <Button Click="Button_Click">Click This Button</Button>

</Window>
```

下面的代码是 XAML 的代码隐藏。

### C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Button was clicked.");
        }
}
```

```
    }  
}
```

## 为 MSBuild 配置窗口

实现窗口的方式决定为 MSBuild 配置窗口的方式。对于使用 XAML 标记和代码隐藏定义的窗口：

- XAML 标记文件配置为 MSBuild `Page` 项。
- 代码隐藏文件配置为 MSBuild `Compile` 项。

.NET SDK 项目会自动导入正确的 `Page` 和 `Compile` 项，无需对其进行声明。如果为 WPF 配置了项目，则会将 XAML 标记文件自动导入为 `Page`，并将相应的代码隐藏文件导入为 `Compile`。

MSBuild 项目不会自动导入类型，你必须自行声明它们：

XML

```
<Project>  
  ...  
  <Page Include="MarkupAndCodeBehindWindow.xaml" />  
  <Compile Include=" MarkupAndCodeBehindWindow.xaml.cs" />  
  ...  
</Project>
```

有关生成 WPF 应用程序的信息，请参阅[编译 WPF 应用程序](#)。

## 窗口生存期

与所有类一样，窗口也有生存期，开始于首次实例化窗口，在这之后将打开、激活/停用直至最终关闭窗口。

## 打开窗口

若要打开窗口，首先要创建窗口实例，下面的示例演示此操作：

C#

```
using System.Windows;  
  
namespace WindowsOverview  
{  
    public partial class App : Application
```

```
{  
    private void Application_Startup(object sender, StartupEventArgs e)  
    {  
        // Create the window  
        Window1 window = new Window1();  
  
        // Open the window  
        window.Show();  
    }  
}
```

在本示例中，`Window1` 在应用程序启动时实例化，此过程在引发 `Startup` 事件时发生。有关“启动”窗口的详细信息，请参阅[如何获取或设置主应用程序窗口](#)。

实例化窗口后，对其的引用将自动添加到由 `Application` 对象管理的[窗口列表](#)。  
`Application` 会将要实例化的第一个窗口自动设置为[主应用程序窗口](#)。

最后通过调用 `Show` 方法打开窗口，如以下图像所示：



通过调用 `Show` 打开的窗口是无模式窗口，应用程序不会阻止用户与该应用程序中的其他窗口交互。通过 `ShowDialog` 打开窗口时，会将窗口打开为模式，并限制用户与该窗口交互。有关详细信息，请参阅[对话框概述](#)。

调用 `Show` 时，窗口先执行初始化工作，然后显示窗口以建立让窗口可以接收用户输入的基础结构。初始化窗口时，将引发 `SourceInitialized` 事件并显示窗口。

有关详细信息，请参阅[如何打开窗口或对话框](#)。

## 启动窗口

上一个示例使用了 `Startup` 事件来运行显示初始应用程序窗口的代码。作为快捷方式，请改用 `StartupUri` 来指定应用程序中 XAML 文件的路径。应用程序将自动创建并显示由该属性指定的窗口。

XAML

```
<Application x:Class="WindowsOverview.App"  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:local="clr-namespace:WindowsOverview"
    StartupUri="ClippedWindow.xaml">
<Application.Resources>

</Application.Resources>
</Application>
```

## 窗口所有权

使用 [Show](#) 方法打开的窗口与创建它的窗口不具有隐式关系。 用户可以与其中任意一个窗口独立交互，这意味着这两个窗口都可以执行以下操作：

- 覆盖另一个窗口（除非其中一个窗口的 [Topmost](#) 属性设置为 `true`）。
- 在不影响另一个窗口的情况下最小化、最大化和还原。

某些窗口要求与打开它们的窗口保持某种关系。 例如，集成开发环境 (IDE) 应用程序可能打开属性窗口和工具窗口，这些窗口的典型行为是覆盖创建它们的窗口。 此外，此类窗口应始终与创建它们的窗口一起关闭、最小化、最大化和还原。 可以通过让一个窗口拥有另一个窗口来建立这种关系，通过使用对所有者窗口的引用设置被拥有窗口的 [Owner](#) 属性来实现。 这在下面的示例中显示。

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create a window and make the current window its owner
    var ownedWindow = new ChildWindow1();
    ownedWindow.Owner = this;
    ownedWindow.Show();
}
```

建立所有权后：

- 被拥有的窗口可以通过检查其 [Owner](#) 属性的值来引用它的所有者窗口。
- 所有者窗口可以通过检查其 [OwnedWindows](#) 属性的值来发现它拥有的所有窗口。

## 窗口激活

第一次打开窗口时，它即成为活动窗口。 活动窗口是当前捕获用户输入（例如击键和鼠标单击）的窗口。 当窗口处于活动状态时，它会引发 [Activated](#) 事件。

① 备注

第一次打开窗口时，只有当引发 `Activated` 后才会引发 `Loaded` 和 `ContentRendered` 事件。记住这一点，在引发 `ContentRendered` 时，实际上就可以认为窗口已打开。

某个窗口成为活动窗口后，用户可以在同一应用程序内激活其他窗口，或者激活其他应用程序。发生这种情况时，将停用当前的活动窗口，并引发 `Deactivated` 事件。同样，如果用户选择当前停用的窗口，该窗口将再次成为活动窗口，并引发 `Activated` 事件。

处理 `Activated` 和 `Deactivated` 的一个常见原因是启用和禁用仅在窗口处于活动状态时才能够运行的功能。例如，一些窗口显示需要用户持续输入或关注的交互式内容，这些内容包括游戏和视频播放器。以下示例是简化的视频播放器，展示如何处理 `Activated` 和 `Deactivated` 以实现此行为。

XAML

```
<Window x:Class="WindowsOverview.CustomMediaPlayerWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Activated="Window_Activated"
        Deactivated="Window_Deactivated"
        Title="CustomMediaPlayerWindow" Height="450" Width="800">
    <Grid>
        <MediaElement x:Name="mediaElement" Stretch="Fill"
                      LoadedBehavior="Manual" Source="numbers.mp4" />
    </Grid>
</Window>
```

下面的代码是 XAML 的代码隐藏。

C#

```
using System;
using System.Windows;

namespace WindowsOverview
{
    public partial class CustomMediaPlayerWindow : Window
    {
        public CustomMediaPlayerWindow() =>
            InitializeComponent();

        private void Window_Activated(object sender, EventArgs e)
        {
            // Continue playing media if window is activated
            mediaElement.Play();
        }

        private void Window_Deactivated(object sender, EventArgs e)
        {
```

```
// Pause playing if media is being played and window is
deactivated
    mediaElement.Pause();
}
}
}
```

停用某个窗口后，其他类型的应用程序可能仍会在后台运行代码。例如，在用户使用其他应用程序时，邮件客户端可能会继续轮询邮件服务器。类似的应用程序在主窗口停用时，通常将提供不同的或额外的行为。对于邮件程序，这可能意味着将新邮件项添加到收件箱和将通知图标添加到系统任务栏。仅当邮件窗口不处于活动状态时才显示通知图标，活动状态是通过检查 [IsActive](#) 属性来确定的。

完成某个后台任务后，窗口可能需要通过调用 [Activate](#) 方法来更迫切地通知用户。如果在调用 [Activate](#) 时，用户正与其他激活的应用程序进行交互，窗口的任务栏按钮会闪烁。但是，如果用户正在与当前应用程序交互，则调用 [Activate](#) 会将窗口置于前景。

### ① 备注

可以使用 [Application.Activated](#) 和 [Application.Deactivated](#) 事件处理应用程序范围的激活。

## 防止窗口激活

在一些情况下，不应在显示窗口时将其激活，例如聊天应用程序的对话窗口或电子邮件应用程序的通知窗口。

如果应用程序的窗口在显示时不激活，可以在首次调用 [Show](#) 方法之前，先将其 [ShowActivated](#) 属性设置为 `false`。结果是：

- 此窗口未激活。
- 未引发窗口的 [Activated](#) 事件。
- 当前激活的窗口保持激活状态。

但是，只要用户通过单击工作区或非工作区激活了窗口，窗口就会变为激活状态。在这种情况下：

- 已激活窗口。
- 已引发窗口的 [Activated](#) 事件。
- 停用之前激活的窗口。
- 然后按照预期，响应用户操作引发窗口的 [Deactivated](#) 和 [Activated](#) 事件。

# 关闭窗口

窗口的生存期在用户关闭它时终止。 窗口关闭后，就不能再重新打开它。 可以使用非工作区中的元素关闭窗口，这些元素包括：

- “系统”菜单的“关闭”项。
- 按 **ALT+F4**。
- 按“关闭”按钮。
- 在模式窗口上，当按钮的 `IsCancel` 属性设置为 `true` 时，按 **ESC**。

可以向工作区提供关闭窗口的更多机制，较为常见的机制包括：

- “文件”菜单中的“退出”项，通常用于主应用程序窗口。
- “文件”菜单中的“关闭”项，通常位于辅助应用程序窗口中。
- “取消”按钮，通常位于模式对话框中。
- “关闭”按钮，通常位于非模式对话框中。

若要为响应其中一种自定义机制而关闭窗口，需要调用 `Close` 方法。 以下示例通过选择“文件”菜单上的“退出”来实现关闭窗口的功能。

XAML

```
<Window x:Class="WindowsOverview.ClosingWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ClosingWindow" Height="450" Width="800">
    <StackPanel>
        <Menu>
            <MenuItem Header="_File">
                <MenuItem Header="E_exit" Click="fileExitMenuItem_Click" />
            </MenuItem>
        </Menu>
    </StackPanel>
</Window>
```

下面的代码是 XAML 的代码隐藏。

C#

```
using System.Windows;

namespace WindowsOverview
{
    public partial class ClosingWindow : Window
    {
        public ClosingWindow() =>
            InitializeComponent();
```

```
        private void fileExitMenuItem_Click(object sender, RoutedEventArgs e)
    {
        // Close the current window
        this.Close();
    }
}
```

## ① 备注

可将应用程序配置为在出现以下情况时自动关闭：主应用程序窗口关闭（请参阅 [MainWindow](#)）或最后一个窗口关闭。有关详细信息，请参阅 [ShutdownMode](#)。

虽然窗口可通过非工作区和工作区中提供的机制显式关闭，但它也可能因为应用程序或 Windows 的其他部分中的行为而隐式关闭，行为包括：

- 用户注销或关闭 Windows。
- 窗口的 [Owner](#) 关闭。
- 主应用程序窗口关闭且 [ShutdownMode](#) 为 [OnMainWindowClose](#)。
- 调用 [Shutdown](#)。

## ① 重要

窗口在关闭后无法重新打开。

## 取消关闭窗口

窗口关闭时，会引发两个事件：[Closing](#) 和 [Closed](#)。

[Closing](#) 在窗口关闭前引发，并提供一种可以阻止窗口关闭的机制。阻止窗口关闭的一个常见原因是窗口内容包含修改的数据。在这种情况下，处理 [Closing](#) 事件可以确定数据是否为已更新，如果已更新，询问用户是在不保存数据的情况下继续关闭窗口，还是取消关闭窗口。以下示例演示了处理 [Closing](#) 的关键方面。

### XAML

```
<Window x:Class="WindowsOverview.DataWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="DataWindow" Height="450" Width="800"
    Closing="Window_Closing">
    <Grid>
        <TextBox x:Name="documentTextBox"
```

```
TextChanged="documentTextBox_TextChanged" />
    </Grid>
</Window>
```

下面的代码是 XAML 的代码隐藏。

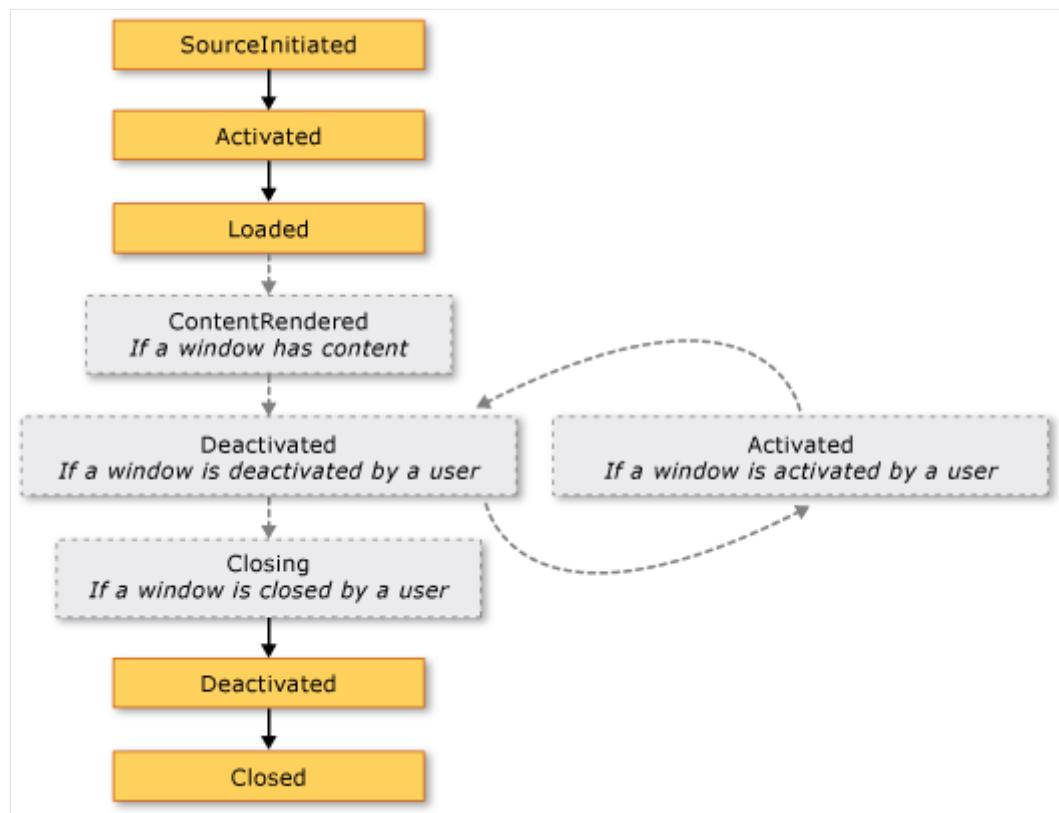
```
C#  
  
using System.Windows;
using System.Windows.Controls;  
  
namespace WindowsOverview
{
    public partial class DataWindow : Window
    {
        private bool _isDataDirty;  
  
        public DataWindow() =>
            InitializeComponent();  
  
        private void documentTextBox_TextChanged(object sender,
TextChangedEventArgs e) =>
            _isDataDirty = true;  
  
        private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
        {
            // If data is dirty, prompt user and ask for a response
            if (_isDataDirty)
            {
                var result = MessageBox.Show("Document has changed. Close
without saving?",
                    "Question",
                    MessageBoxButton.YesNo);
  
                // User doesn't want to close, cancel closure
                if (result == MessageBoxResult.No)
                    e.Cancel = true;
            }
        }
    }
}
```

向 [Closing](#) 事件处理程序传递 [CancelEventArgs](#)，它会实现 [Cancel](#) 属性，将该属性设置为 `true` 以防止窗口关闭。

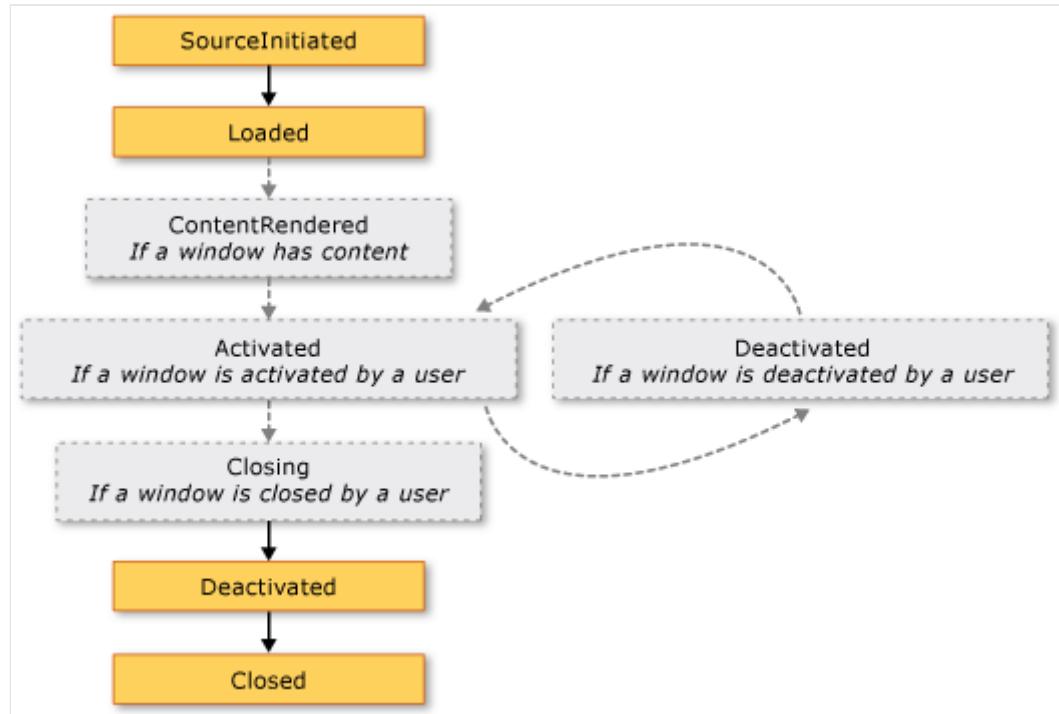
如果未处理 [Closing](#)，或者已处理但未取消，窗口将关闭。在窗口真正关闭前，将引发 [Closed](#)。此时，无法阻止窗口关闭。

## 窗口生存期事件

下图显示窗口生存期中的主体事件的顺序：



下图显示窗口（显示时没有激活）生存期中的主体事件的顺序（显示窗口之前将 `ShowActivated` 设置为 `false`）：



## 窗口位置

当窗口打开时，它在相对于桌面的 x 和 y 维度中有一个位置。可以通过检查 `Left` 和 `Top` 来确定此位置。设置这些属性以更改窗口的位置。

还可通过使用以下任一 `WindowStartupLocation` 枚举值来设置 `WindowStartupLocation` 属性，从而指定 `Window` 首次出现时的初始位置：

- `CenterOwner` (默认值)
- `CenterScreen`
- `Manual`

如果将启动位置指定为 `Manual`，并且未设置 `Left` 和 `Top` 属性，`Window` 将要求操作系统指定其显示位置。

## 最顶层窗口和 z 顺序

除了有 x 和 y 位置外，窗口还在 z 维度中有一个位置，该位置确定窗口相对于其他窗口的垂直位置。它称为窗口的 z 顺序，并且有两种类型：正常 z 顺序和最顶层 z 顺序。正常 z 顺序中的窗口位置取决于窗口当前是否处于活动状态。默认情况下，窗口位于正常 z 顺序中。最顶层 z 顺序中的窗口位置也取决于它当前是否处于活动状态。此外，最顶层 z 顺序中的窗口始终位于正常 z 顺序中的窗口之上。窗口通过将其 `Topmost` 属性设置为 `true` 来采用最顶层 z 顺序。

在每种 z 顺序类型中，当前的活动窗口显示在同一 z 顺序中所有其他窗口之上。

## 窗口大小

除了拥有桌面位置外，窗口还有大小，大小由多个属性确定，包括各种宽度和高度属性以及 `SizeToContent`。

`MinWidth`、`Width` 和 `MaxWidth` 用于管理窗口在其生存期内可以具有的宽度范围。

### XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    MinWidth="300" Width="400" MaxWidth="500">
</Window>
```

窗口高度由 `MinHeight`、`Height` 和 `MaxHeight` 管理。

### XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    MinHeight="300" Height="400" MaxHeight="500">
</Window>
```

由于各种宽度值和高度值各自指定了一个范围，所以大小可调整大小的窗口的宽度和高度可以是相应维度中指定范围内的任何值。 若要检测其当前的宽度和高度，请分别检查 [ActualWidth](#) 和 [ActualHeight](#)。

如果希望窗口的宽度和高度适应窗口内容的大小，可以使用 [SizeToContent](#) 属性，它具有以下值：

- [SizeToContent.Manual](#)  
不起作用（默认值）。
- [SizeToContent.Width](#)  
适应内容宽度，与将 [MinWidth](#) 和 [MaxWidth](#) 都设置为内容的宽度效果相同。
- [SizeToContent.Height](#)  
适应内容高度，与将 [MinHeight](#) 和 [MaxHeight](#) 都设置为内容的高度效果相同。
- [SizeToContent.WidthAndHeight](#)  
适应内容宽度和高度，与将 [MinHeight](#) 和 [MaxHeight](#) 都设置为内容的高度，并将 [MinWidth](#) 和 [MaxWidth](#) 都设置为内容的宽度效果相同。

以下示例显示了一个窗口，它在第一次显示时即自动调整垂直方向和水平方向上的大小以适应内容。

#### XAML

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    SizeToContent="WidthAndHeight">  
</Window>
```

以下示例显示如何在代码中设置 [SizeToContent](#) 属性以指定重设窗口大小使其适应内容的方式。

#### C#

```
// Manually alter window height and width  
this.SizeToContent = SizeToContent.Manual;  
  
// Automatically resize width relative to content  
this.SizeToContent = SizeToContent.Width;  
  
// Automatically resize height relative to content  
this.SizeToContent = SizeToContent.Height;  
  
// Automatically resize height and width relative to content  
this.SizeToContent = SizeToContent.WidthAndHeight;
```

## 大小调整属性的优先级顺序

从根本上说，窗口的各种大小属性可以结合使用，以定义可调整大小的窗口的宽度和高度范围。为了确保保持有效的范围，[Window](#) 将使用以下优先级顺序计算大小属性的值。

### 对于高度属性：

1. [FrameworkElement.MinHeight](#)
2. [FrameworkElement.MaxHeight](#)
3. [SizeToContent.Height / SizeToContent.WidthAndHeight](#)
4. [FrameworkElement.Height](#)

### 对于宽度属性：

1. [FrameworkElement.MinWidth](#)
2. [FrameworkElement.MaxWidth](#)
3. [SizeToContent.Width / SizeToContent.WidthAndHeight](#)
4. [FrameworkElement.Width](#)

优先级顺序还可以确定窗口在最大化时的大小，此时的窗口大小由 [WindowState](#) 属性管理。

## Window state

可调整大小的窗口在生存期中拥有三种状态：正常、最小化和最大化。正常是窗口的默认状态。这种状态下的窗口允许用户使用重设大小手柄或边框移动窗口和重设其大小（前提是大小可以重设）。

如果 [ShowInTaskbar](#) 设置为 `true`，则最小化状态下的窗口将折叠到任务栏按钮；否则，它将尽可能折叠到最小大小，并将自己移动到桌面的左下角。虽然不在任务栏显示的最小化窗口可以在桌面上四处拖动，但这两种类型的最小化窗口都不可以使用边框或重设大小手柄重设窗口大小。

具有最大化状态的窗口会扩展到它能具有的最大大小，这不能超过 [MaxWidth](#)、[MaxHeight](#) 和 [SizeToContent](#) 属性指定的大小。与最小化窗口一样，最大化窗口无法使用重设大小手柄或通过拖动边框来重设大小。

### ① 备注

即使窗口当前已最大化或最小化，窗口的 [Top](#)、[Left](#)、[Width](#) 和 [Height](#) 属性的值也始终表示正常状态的值。

可以通过设置 [WindowState](#) 属性来配置窗口的状态，该属性可以具有以下 [WindowState](#) 枚举值之一：

- Normal (默认值)
- Maximized
- Minimized

以下示例显示如何创建在打开时最大化显示的窗口。

#### XAML

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    WindowState="Maximized">  
</Window>
```

通常，应设置 [WindowState](#) 以配置窗口的初始状态。 显示可调整大小的窗口后，用户可以按窗口标题栏上的“最小化”、“最大化”和“还原”按钮来更改窗口状态。

## 窗口外观

通过将特定于窗口的内容（例如按钮、标签和文本框）添加到窗口的工作区可以更改它的外观。 为配置非工作区，[Window](#) 提供几个属性，包括用于设置窗口图标的 [Icon](#) 和用于设置其标题的 [Title](#)。

还可以通过配置窗口的重设大小模式、窗口样式，以及窗口是否显示为桌面任务栏中的按钮，更改非工作区边框的外观和行为。

## 重设大小模式

根据 [WindowStyle](#) 属性，你可以控制用户是否可以重设窗口的大小，以及如何重设。 窗口样式影响以下各项：

- 允许或禁止使用鼠标拖动窗口边框来调整大小。
- 非工作区是否显示“最小化”、“最大化”和“关闭”按钮。
- 是否启用了“最小化”、“最大化”和“关闭”按钮。

可以通过设置窗口的 [ResizeMode](#) 属性来配置重设窗口大小的方式，该属性可以是下列 [ResizeMode](#) 枚举值之一：

- [NoResize](#)
- [CanMinimize](#)
- [CanResize](#) (默认值)
- [CanResizeWithGrip](#)

与 [WindowStyle](#) 一样，窗口的重设大小模式在生存期中不太可能更改，这意味着它最有可能在 XAML 标记中进行设置。

#### XAML

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    ResizeMode="CanResizeWithGrip">  
</Window>
```

请注意，可以通过检查 [WindowState](#) 属性来检测是否已最大化、最小化或还原窗口。

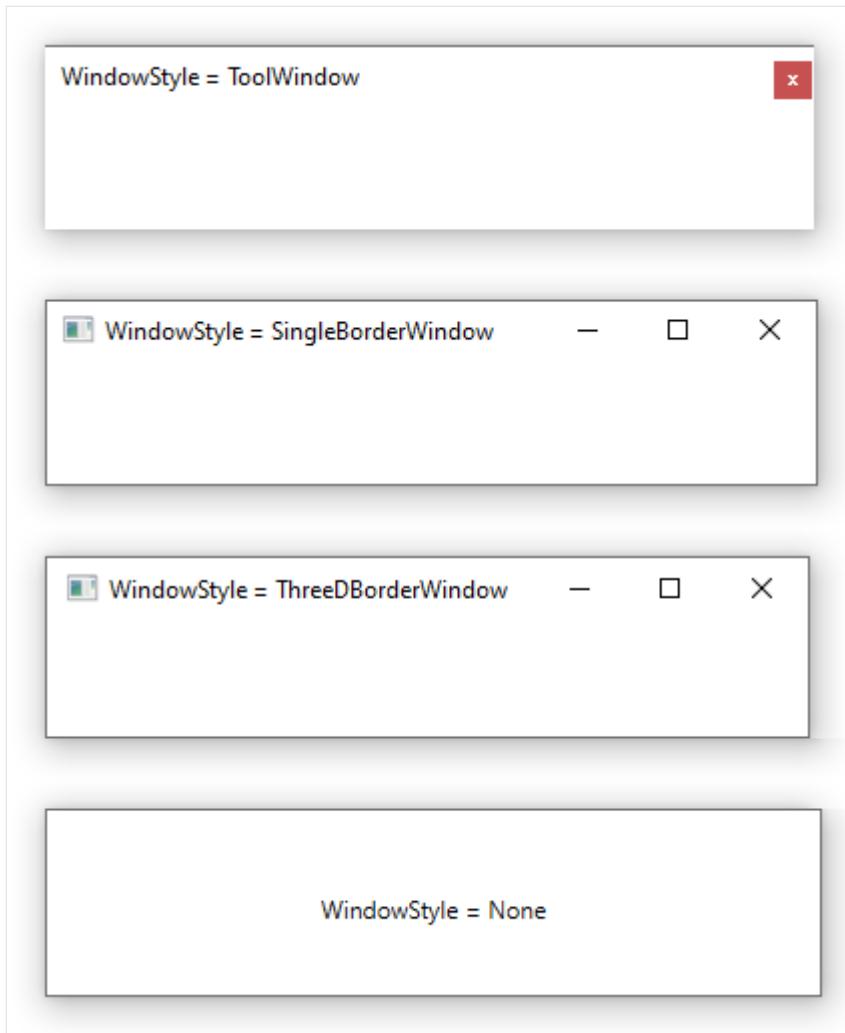
## Window style

从窗口非工作区公开的边框适用于大多数应用程序。但是，有时候会需要不同类型的边框，或者根本不需要边框，具体取决于窗口类型。

若要控制窗口的边框类型，请将其 [WindowStyle](#) 属性设置为以下 [WindowStyle](#) 枚举值之一：

- [None](#)
- [SingleBorderWindow](#) (默认值)
- [ThreeDBorderWindow](#)
- [ToolWindow](#)

应用窗口样式的效果如下图所示：



请注意，上图未显示 `SingleBorderWindow` 与 `ThreeDBorderWindow` 之间的任何明显差异。回到 Windows XP 中，`ThreeDBorderWindow` 确实会影响窗口的绘制方式，将三维边框添加到工作区。从 Windows 7 开始，这两种样式之间的差异很小。

可以使用 XAML 标记或代码设置 `WindowStyle`。由于在窗口生存期内不太可能发生更改，因此你最有可能使用 XAML 标记对其进行配置。

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    WindowStyle="ToolWindow">
</Window>
```

## 非矩形窗口样式

在另外一些情况下，`WindowStyle` 提供的边框样式不能满足需要。例如，可能希望创建一个带有非矩形边框（如 Windows Media Player 所使用的边框）的应用程序。

下图中显示的对话气泡框就是一个例子：



可以通过将 `WindowStyle` 属性设置为 `None`，并使用 `Window` 为透明度提供的特殊支持，来创建这种类型的窗口。

#### XAML

```
<Window x:Class="WindowsOverview.ClippedWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="ClippedWindow" SizeToContent="WidthAndHeight"
    WindowStyle="None" AllowsTransparency="True"
Background="Transparent">
    <Grid Margin="20">
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="20"/>
        </Grid.RowDefinitions>

        <Rectangle Stroke="#FF000000" RadiusX="10" RadiusY="10"/>
        <Path Fill="White" Stretch="Fill" Stroke="#FF000000"
HorizontalAlignment="Left" Margin="15,-5.597,0,-0.003" Width="30"
Grid.Row="1" Data="M22.166642,154.45381 L29.999666,187.66699
40.791059,154.54395" />
        <Rectangle Fill="White" RadiusX="10" RadiusY="10" Margin="1"/>

        <TextBlock HorizontalAlignment="Left" VerticalAlignment="Center"
FontSize="25" Text="Greetings!" TextWrapping="Wrap" Margin="5,5,50,5"/>
        <Button HorizontalAlignment="Right" VerticalAlignment="Top"
Background="Transparent" BorderBrush="{x:Null}" Foreground="Red"
Content="✖" FontSize="15" />

        <Grid.Effect>
            <DropShadowEffect BlurRadius="10" ShadowDepth="3"
Color="LightBlue"/>
        </Grid.Effect>
    </Grid>
</Window>
```

多个值组合起来可以指示窗口呈现透明的效果。在此状态下，不能使用窗口的非工作区修饰按钮，而需要提供你自己的修饰按钮。

## 任务栏显示

窗口的默认外观包含一个任务栏按钮。某些类型的窗口没有任务栏按钮，如消息框、对话框或 `WindowStyle` 属性设置为 `ToolWindow` 的窗口。设置 `ShowInTaskbar` 属性（默认

为 `true`) 可以控制是否显示窗口的任务栏按钮。

#### XAML

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    ShowInTaskbar="False">  
</Window>
```

## 其他类型的窗口

[NavigationWindow](#) 是设计用于托管可导航内容的窗口。

对话框是通常用来收集用户信息以完成某项功能的窗口。例如，用户需要打开某个文件时，应用程序会显示“打开文件”对话框，以从用户那里获取文件名。有关详细信息，请参阅[对话框概述](#)。

## 另请参阅

- [对话框概述](#)
- [如何打开窗口或对话框](#)
- [如何打开通用对话框](#)
- [如何打开消息框](#)
- [如何关闭窗口或对话框](#)
- [System.Windows.Window](#)
- [System.Windows.MessageBox](#)
- [System.Windows.Navigation.NavigationWindow](#)
- [System.Windows.Application](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

### .NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# 对话框概述 (WPF .NET)

项目 · 2023/10/13

Windows Presentation Foundation (WPF) 为你提供了自行设计对话框的方法。对话框是窗口，但具有特定的意图和用户体验。本文讨论对话框的工作原理以及可以创建和使用的对话框类型。对话框用于：

- 向用户显示特定信息。
- 从用户处收集信息。
- 同时显示并收集信息。
- 显示操作系统提示，例如打印窗口。
- 选择文件或文件夹。

这些类型的窗口称为对话框。对话框可以通过两种方式显示：模式和非模式。

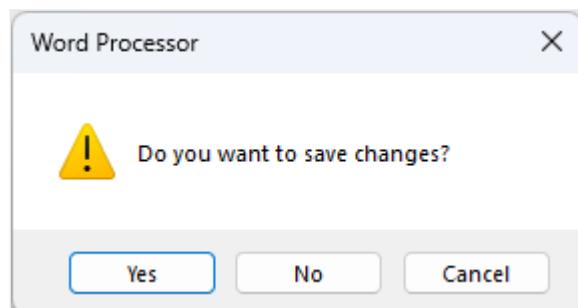
向用户显示模式对话框是一种技术，应用程序使用该技术中断其正在执行的操作，直到用户关闭对话框。这通常以提示或警报的形式出现。在关闭对话框之前，无法与应用程序中的其他窗口进行交互。模式对话框关闭后，应用程序将继续运行。最常见的对话框用于显示打开文件或保存文件提示、显示打印机对话框或向用户发送一些状态消息。

非模式对话框打开时不会阻止用户激活其他窗口。例如，如果用户想要在文档中查找特定单词的匹配项，主窗口通常会打开一个对话框，询问用户要查找什么单词。由于应用程序不想阻止用户编辑文档，因此该对话框不必为模式对话框。非模式对话框至少提供“关闭”按钮来关闭对话框。可能还会提供其他按钮来运行特定功能，例如提供“查找下一个”按钮以在单词搜索中查找下一个单词。

你可以使用 WPF 创建多种类型的对话框，例如消息框、通用对话框和自定义对话框。本文将讨论每种对话框，[对话框示例](#) 提供了匹配示例。

## 消息框

消息框是可以用来显示文本信息并使用户可以使用按钮做出决定的对话框。下图显示了一个消息框，框中询问问题并为用户提供三个按钮来回答问题。



要创建消息框，可以使用 [MessageBox](#) 类。 [MessageBox](#) 允许你配置消息框文本、标题、图标和按钮。

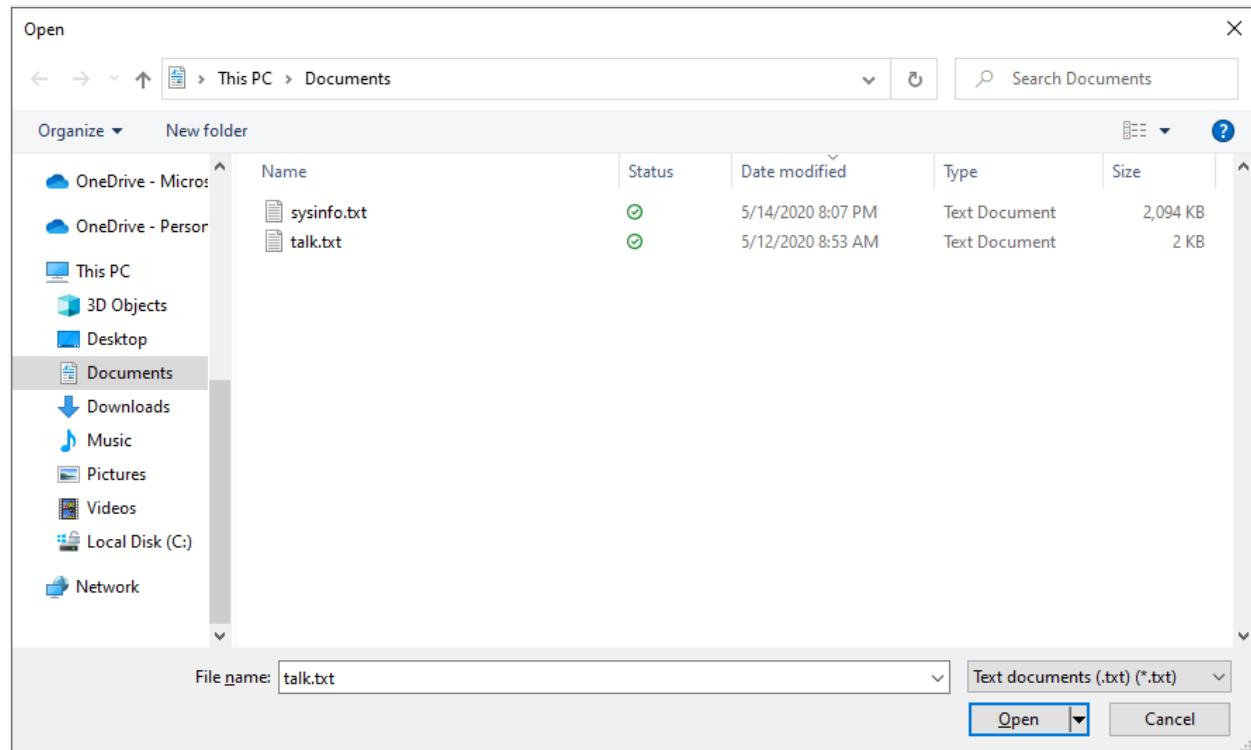
有关详细信息，请参阅[如何打开消息框](#)。

## 通用对话框

Windows 实现了所有应用程序通用的不同类型的可重用对话框，其中包括用于选择文件和打印的对话框。

由于这些对话框是由操作系统提供的，因此它们在操作系统上运行的所有应用程序之间共享。 这些对话框提供一致的用户体验，被称为通用对话框。 当用户在一个应用程序中使用通用对话框时，他们不需要学习如何在其他应用程序中使用该对话框。

WPF 封装了“打开文件”、“保存文件”和“打印”通用对话框，并将它们公开为托管类，供你在独立应用程序中使用。



若要详细了解通用对话框，请参阅以下文章：

- [如何显示通用对话框](#)
- [显示“打开文件”对话框](#)
- [显示“保存文件”对话框](#)
- [显示“打印”对话框](#)

## 自定义对话框

虽然通用对话框很有用，并且应尽可能使用，但它们不支持域特定对话框的要求。在这些情况下，就需要创建自己的对话框。如我们所见，对话框是具有特殊行为的窗口。[Window](#) 实现了这些行为，你可以使用窗口来创建自定义模式和非模式对话框。

自行创建对话框时，需要考虑许多设计注意事项。尽管应用程序窗口和对话框有相似之处，例如共享相同的基类，但对话框用于特定目的。当你需要提示用户提供某种信息或响应时，通常需要对话框。通常，应用程序会在显示对话框（模式）时暂停，从而限制对应用程序其余部分的访问。对话框关闭后，应用程序将继续运行。但是，将交互仅限制于对话框并非必要。

当 WPF 窗口关闭时，它无法重新打开。自定义对话框是 WPF 窗口，适用相同的规则。若要了解如何关闭窗口，请参阅[如何关闭窗口或对话框](#)。

## 实现对话框

设计对话框时，请遵循以下建议来创造良好的用户体验：

- ✖ 不要让对话框窗口变得杂乱无章。对话框体验是让用户输入一些数据或做出选择。
- ✓ 务必提供“确定”按钮来关闭窗口。
- ✓ 务必将“确定”按钮的 `IsDefault` 属性设置为 `true`，以允许用户按 `ENTER` 键接受并关闭窗口。
- ✓ 考虑添加“取消”按钮，以便用户可以关闭窗口并表明他们不想继续操作。
- ✓ 务必将“取消”按钮的 `IsCancel` 属性设置为 `true`，以允许用户按 `ESC` 键关闭窗口。
- ✓ 务必设置窗口标题，以准确描述对话框所代表的内容，或者用户应对对话框执行的操作。
- ✓ 务必为窗口设置最小宽度和高度值，以防止用户将窗口调整得太小。
- ✓ 如果 `ShowInTaskbar` 设置为 `false`，请考虑禁用调整窗口大小的功能。可以通过将 `ResizeMode` 设置为 `NoResize` 来禁用调整大小

以下代码演示了这种配置。

XAML

```
<Window x:Class="Dialogs.Margins"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Change Margins"
```

```

        Closing="Window_Closing"
        MinHeight="200"
        MinWidth="300"
        SizeToContent="WidthAndHeight"
        ResizeMode="NoResize"
        ShowInTaskbar="False"
        WindowStartupLocation="CenterOwner"
        FocusManager.FocusedElement="{Binding
ElementName=leftMarginTextBox}">
    <Grid Margin="10">
        <Grid.Resources>
            <!-- Default settings for controls -->
            <Style TargetType="{x:Type Label}">
                <Setter Property="Margin" Value="0,3,5,5" />
                <Setter Property="Padding" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type TextBox}">
                <Setter Property="Margin" Value="0,0,0,5" />
            </Style>
            <Style TargetType="{x:Type Button}">
                <Setter Property="Width" Value="70" />
                <Setter Property="Height" Value="25" />
                <Setter Property="Margin" Value="5,0,0,0" />
            </Style>
        </Grid.Resources>

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

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

        <!-- Left,Top,Right,Bottom margins-->
        <Label Grid.Column="0" Grid.Row="0">Left Margin:</Label>
        <TextBox Name="leftMarginTextBox" Grid.Column="1" Grid.Row="0" />

        <Label Grid.Column="0" Grid.Row="1">Top Margin:</Label>
        <TextBox Name="topMarginTextBox" Grid.Column="1" Grid.Row="1" />

        <Label Grid.Column="0" Grid.Row="2">Right Margin:</Label>
        <TextBox Name="rightMarginTextBox" Grid.Column="1" Grid.Row="2" />

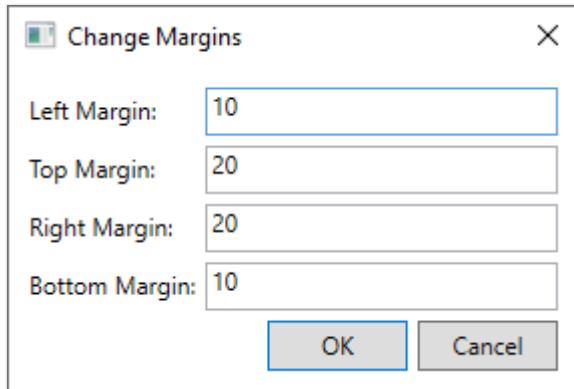
        <Label Grid.Column="0" Grid.Row="3">Bottom Margin:</Label>
        <TextBox Name="bottomMarginTextBox" Grid.Column="1" Grid.Row="3" />

        <!-- Accept or Cancel -->

```

```
<StackPanel Grid.Column="0" Grid.ColumnSpan="2" Grid.Row="4"
Orientation="Horizontal" HorizontalAlignment="Right">
    <Button Name="okButton" Click="okButton_Click"
IsDefault="True">OK</Button>
    <Button Name="cancelButton" IsCancel="True">Cancel</Button>
</StackPanel>
</Grid >
</Window>
```

上述 XAML 创建了一个类似于下图的窗口：



## 打开对话框的 UI 元素

对话框用户体验还扩展到打开对话框的窗口菜单栏或按钮。当菜单项或按钮运行需要用户通过对话框交互才能继续运行的函数时，控件应在其标题文本的末尾使用省略号：

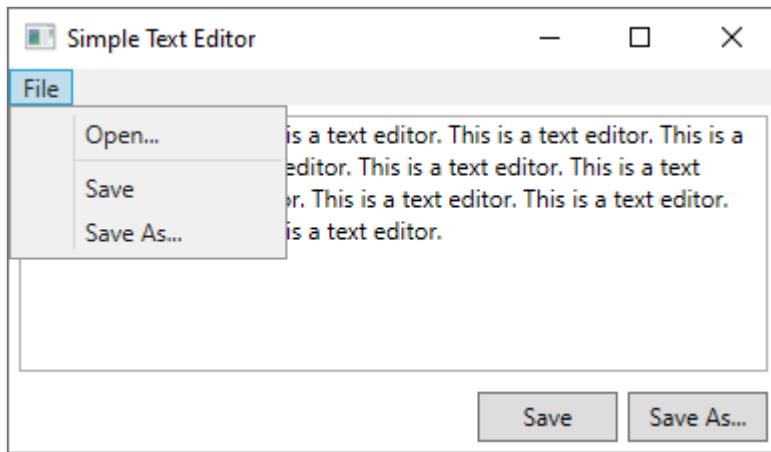
### XAML

```
<MenuItem Header="_Margins..." Click="formatMarginsMenuItem_Click" />
<!-- or -->
<Button Content="_Margins..." Click="formatMarginsButton_Click" />
```

当菜单项或按钮运行的函数显示无需用户交互的对话框（如“关于”对话框）时，则不需要省略号。

## 菜单项

菜单项是向用户提供按相关主题分组的应用程序操作的常用方式。你可能在许多不同的应用程序上看到过“文件”菜单。在典型应用程序中，“文件”菜单项提供保存文件、加载文件和打印文件的方法。如果操作要显示模式窗口，则标题通常包含省略号，如下图所示：

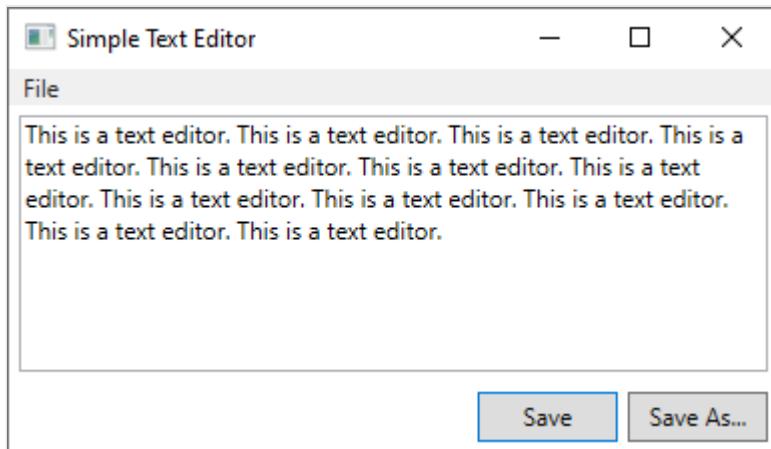


其中两个菜单项带有省略号：...。这有助于用户确定，当他们选择这些菜单项时，系统会显示一个模式窗口，并暂停应用程序直到用户关闭该窗口。

这种设计技术是向用户传达预期内容的一种简单方法。

## 按钮

你可以遵循[菜单项](#)部分中所述的相同原则。在按钮文本上使用省略号表示当用户按下按钮时，会显示一个模式对话框。下图中有两个按钮，哪个按钮会显示对话框一目了然：



## 返回结果

打开另一个窗口（尤其是模式对话框）是将状态和信息返回给调用代码的好方法。

## 模式对话框

通过调用[ShowDialog\(\)](#) 显示对话框时，打开对话框的代码会等待 `ShowDialog` 方法返回结果。该方法返回结果时，调用它的代码需要决定是继续处理还是停止处理。用户通常通过按对话框上的“确定”或“取消”按钮做出指示。

按下“确定”按钮时，`ShowDialog` 应设计为返回 `true`，而按下“取消”按钮时，应返回 `false`。这是通过在按下按钮时设置 `DialogResult` 属性来实现的。

C#

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

VB

```
Private Sub okButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = True
End Sub

Private Sub cancelButton_Click(sender As Object, e As RoutedEventArgs)
    DialogResult = False
End Sub
```

只有使用 `ShowDialog()` 显示对话框时，才能设置 `DialogResult` 属性。设置 `DialogResult` 属性后，对话框关闭。

如果按钮的 `IsCancel` 属性设置为 `true`，并且使用 `ShowDialog()` 打开窗口，则 `ESC` 键将关闭窗口并将 `DialogResult` 设置为 `false`。

有关关闭对话框的详细信息，请参阅[如何关闭窗口或对话框](#)。

## 处理响应

`ShowDialog()` 返回一个布尔值，指示用户是接受还是取消了对话框。如果你要提醒用户注意某事，但不要求他们做出决定或提供数据，则可以忽略响应。也可以通过检查 `DialogResult` 属性来检查响应。以下代码演示如何处理响应：

C#

```
var dialog = new Margins();

// Display the dialog box and read the response
bool? result = dialog.ShowDialog();

if (result == true)
{
    // User accepted the dialog box
    MessageBox.Show("Your request will be processed.");
}
```

```
else
{
    // User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.");
}
```

VB

```
Dim marginsWindow As New Margins

Dim result As Boolean? = marginsWindow.ShowDialog()

If result = True Then
    ' User accepted the dialog box
    MessageBox.Show("Your request will be processed.")
Else
    ' User cancelled the dialog box
    MessageBox.Show("Sorry it didn't work out, we'll try again later.")
End If

marginsWindow.Show()
```

## 非模式对话框

若要显示非模式对话框，请调用 [Show\(\)](#)。该对话框至少应提供“关闭”按钮。可以提供其他按钮和交互元素来运行特定功能，例如提供“查找下一个”按钮以在单词搜索中查找下一个单词。

由于非模式对话框不会阻止调用代码继续执行，因此你必须提供其他返回结果的方式。可以执行以下操作之一：

- 在窗口上公开数据对象属性。
- 处理调用代码中的 [Window.Closed](#) 事件。
- 在窗口上创建事件，这些事件在用户选择对象或按下特定按钮时引发。

以下示例使用 [Window.Closed](#) 事件在对话框关闭时向用户显示消息框。显示的消息引用已关闭对话框的属性。有关关闭对话框的详细信息，请参阅[如何关闭窗口或对话框](#)。

C#

```
var marginsWindow = new Margins();

marginsWindow.Closed += (sender, eventArgs) =>
{
    MessageBox.Show($"You closed the margins window! It had the title of
{marginsWindow.Title}");
};
```

```
marginsWindow.Show();
```

VB

```
Dim marginsWindow As New Margins

AddHandler marginsWindow.Closed, Sub(sender As Object, e As EventArgs)
    MessageBox.Show($"You closed the
margins window! It had the title of {marginsWindow.Title}")
End Sub

marginsWindow.Show()
```

## 另请参阅

- [WPF 窗口概述](#)
- [如何打开窗口或对话框](#)
- [如何打开通用对话框](#)
- [如何打开消息框](#)
- [如何关闭窗口或对话框](#)
- [对话框示例](#)
- [System.Windows.Window](#)
- [System.Windows.MessageBox](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

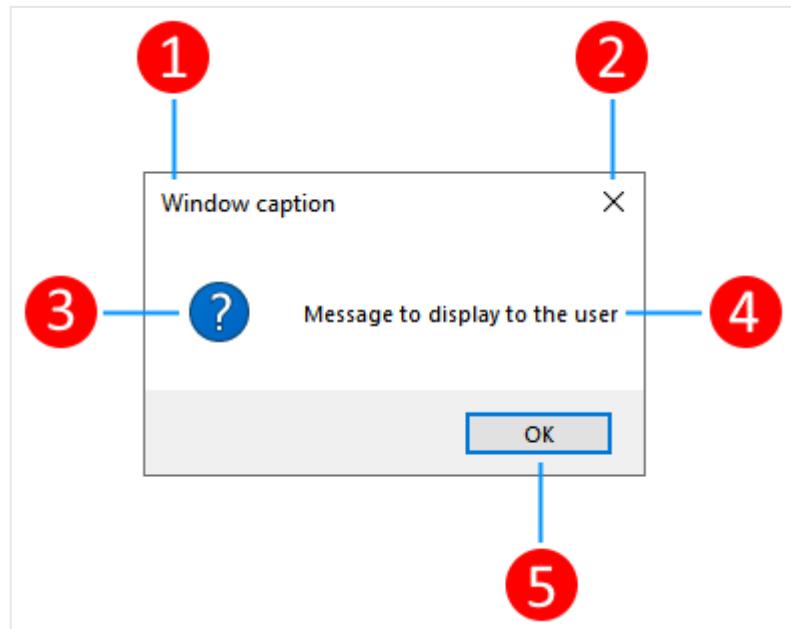
 提供产品反馈

# 如何打开消息框 (WPF .NET)

项目 • 2023/10/13

“消息框”是一个用于快速显示信息并允许用户有选择地做出决策的对话框。对消息框的访问由 [MessageBox](#) 类提供。以模式方式显示消息框。显示消息框的代码将暂停，直到用户使用关闭按钮或响应按钮关闭消息框。

下图演示了消息框的各个部分：



- 带有标题的标题栏 (1)。
- 关闭按钮 (2)。
- 图标 (3)。
- 向用户显示的消息 (4)。
- 响应按钮 (5)。

对于显示或收集复杂数据，对话框可能比消息框更适合。有关详细信息，请参阅[对话框概述](#)。

## 显示消息框

要创建消息框，可以使用 [MessageBox](#) 类。使用 [MessageBox.Show](#) 方法可以配置消息框文本、标题、图标和按钮，如下面的代码所示：

C#

```
string messageBoxText = "Do you want to save changes?";
string caption = "Word Processor";
MessageBoxButton button = MessageBoxButton.YesNoCancel;
MessageBoxImage icon = MessageBoxIcon.Warning;
```

```
MessageBoxResult result;

result = MessageBox.Show(messageBoxText, caption, button, icon,
MessageBoxResult.Yes);
```

MessageBox.Show 方法重载提供了配置消息框的方法。这些选项包括：

- 标题栏标题
- 消息图标
- 消息文本
- 响应按钮

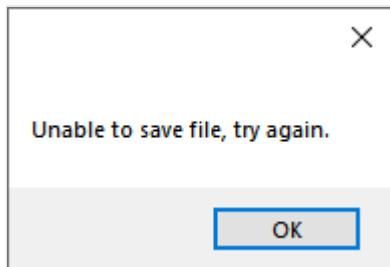
下面是使用消息框的更多示例。

- 显示警报。

C#

```
MessageBox.Show("Unable to save file, try again.");
```

前面的代码显示如下所示的消息框：

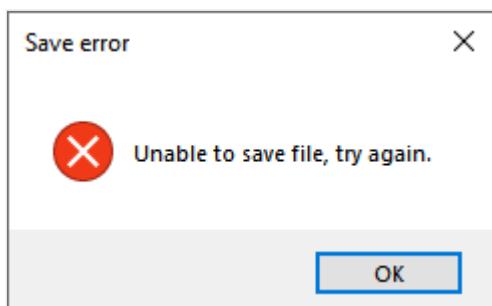


最好使用消息框类提供的选项。使用与之前相同的警报，设置更多选项，使其更具视觉吸引力：

C#

```
MessageBox.Show("Unable to save file, try again.", "Save error",
MessageBoxButton.OK, MessageBoxIcon.Error);
```

前面的代码显示如下所示的消息框：

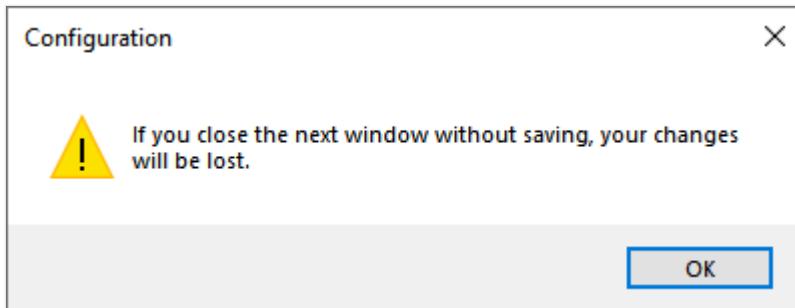


- 显示警告。

```
C#
```

```
MessageBox.Show("If you close the next window without saving, your  
changes will be lost.", "Configuration", MessageBoxButtons.OK,  
MessageBoxImage.Warning);
```

前面的代码显示如下所示的消息框：

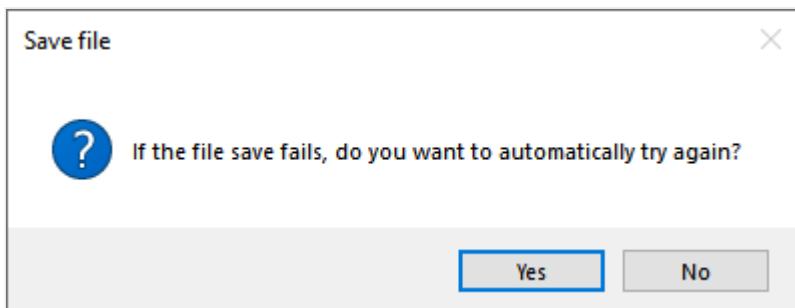


- 询问用户一个问题。

```
C#
```

```
if (MessageBox.Show("If the file save fails, do you want to  
automatically try again?",  
    "Save file",  
    MessageBoxButtons.YesNo,  
    MessageBoxIcon.Question) == DialogResult.Yes)  
{  
    // Do something here  
}
```

前面的代码显示如下所示的消息框：



## 处理消息框响应

[MessageBox.Show](#) 方法显示消息框并返回结果。 结果指示用户如何关闭消息框：

```
C#
```

```
result = MessageBox.Show(messageBoxText, caption, button, icon,
MessageBoxResult.Yes);

switch (result)
{
    case MessageBoxResult.Cancel:
        // User pressed Cancel
        break;
    case MessageBoxResult.Yes:
        // User pressed Yes
        break;
    case MessageBoxResult.No:
        // User pressed No
        break;
}
```

当用户按下消息框底部的按钮时，将返回相应的 [MessageBoxResult](#)。但是，如果用户按“ESC”键或按“关闭”按钮（[消息框图示](#)中的 #2），则消息框的结果因按钮选项而异○：

按钮选项	“ESC”或“关闭”按钮结果○
OK	OK
OKCancel	Cancel
YesNo	“ESC”键盘快捷方式和“关闭”按钮已禁用○。 用户必须按“是”或“否”。
YesNoCancel	Cancel

有关使用消息框的详细信息，请参阅 [MessageBox](#)[MessageBox 示例](#)○。

## 另请参阅

- [WPF 窗口概述](#)
- [对话框概述](#)
- [如何显示通用对话框](#)
- [MessageBox](#) [示例](#) ↴
- [System.Windows.MessageBox](#)
- [System.Windows.MessageBox.Show](#)
- [System.Windows.MessageBoxResult](#)

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何打开窗口或对话框 (WPF .NET)

项目 • 2023/10/13

你可以创建自己的窗口并在 Windows Presentation Foundation (WPF) 中显示这些窗口。本文介绍如何显示模式窗口和无模式窗口以及对话框。

从概念上讲，窗口和对话框相同：都显示给用户以提供信息或进行交互。二者都属于“窗口”对象。窗口的设计及其使用方式构成了对话框。对话框通常较小，需要用户对其进行响应。有关详细信息，请参阅 [WPF 窗口概述](#) 和 [对话框概述](#)。

如果对打开操作系统对话框感兴趣，请参阅[如何打开通用对话框](#)。

## 以模式方式打开

打开的模式窗口通常表示一个对话框。WPF 限制与模式窗口的交互，打开窗口的代码会在窗口关闭前暂停。此机制提供了一种简单的方法，用于提示用户输入数据并等待其响应。

使用 [ShowDialog](#) 方法打开一个窗口。以下代码将实例化窗口，并以模式方式打开窗口。打开窗口的代码会暂停，等待窗口关闭：

```
C#  
  
var window = new Margins();  
  
window.Owner = this;  
window.ShowDialog();
```

### ① 重要

关闭某个窗口后，不能使用同一对象实例重新打开该窗口。

有关如何处理用户对对话框的响应的详细信息，请参阅[对话框概述：处理响应](#)。

## 以无模式方式打开

以无模式方式打开窗口意味着窗口将显示为普通窗口。打开窗口的代码会在窗口成为可见状态时继续运行。可以聚焦应用程序显示的所有无模式窗口并与之交互，而不受任何限制。

使用 [Show](#) 方法打开一个窗口。以下代码将实例化窗口，并以无模式方式打开窗口。打开窗口的代码继续运行：

C#

```
var window = new Windows.Window();
window.Owner = this;
window.Show();
```

### ① 重要

关闭某个窗口后，不能使用同一对象实例重新打开该窗口。

## 另请参阅

- [WPF 窗口概述](#)
- [对话框概述](#)
- [如何关闭窗口或对话框](#)
- [如何打开通用对话框](#)
- [如何打开消息框](#)
- [System.Windows.Window](#)
- [System.Windows.Window.DialogResult](#)
- [System.Windows.Window.Show\(\)](#)
- [System.Windows.Window.ShowDialog\(\)](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何关闭窗口或对话框 (WPF .NET)

项目 · 2023/10/13

在本文中，你将了解关闭窗口或对话框的不同方法。 用户可使用非工作区中的元素关闭窗口，这些元素包括：

- “系统”菜单的“关闭”项。
- 按 `ALT+F4`。
- 按“关闭”按钮。
- 在模式窗口上，当按钮的 `IsCancel` 属性设置为 `true` 时，按 `ESC`。

设计窗口时，为工作区提供了更多用于关闭窗口的机制。 窗口中用于关闭该窗口的一些常见设计元素包括：

- “文件”菜单中的“退出”项，通常用于主应用程序窗口。
- “文件”菜单中的“关闭”项，通常位于辅助应用程序窗口中。
- “取消”按钮，通常位于模式对话框中。
- “关闭”按钮，通常位于非模式对话框中。

## ① 重要

关闭某个窗口后，不能使用同一对象实例重新打开该窗口。

有关窗口生命周期的详细信息，请参阅 [WPF 窗口概述：窗口生命周期](#)。

## 关闭模式窗口

当关闭使用 `ShowDialog` 方法打开的窗口时，需将 `DialogResult` 属性设置为 `true` 或 `false`，以分别指示“已接受”或“已取消”状态。 如果将 `DialogResult` 属性设置为某个值，窗口将关闭。 下面的代码演示如何设置 `DialogResult` 属性：

C#

```
private void okButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = true;

private void cancelButton_Click(object sender, RoutedEventArgs e) =>
    DialogResult = false;
```

也可以调用 `Close` 方法。 如果使用了 `Close` 方法，则将 `DialogResult` 属性设置为 `false`。

关闭某个窗口后，就不能用同一对象实例将其重新打开。如果尝试显示同一窗口，则会引发 [InvalidOperationException](#)。请改为创建新的窗口实例并将其打开。

## 关闭非模式窗口

当关闭使用 [Show](#) 方法打开的窗口时，请使用 [Close](#) 方法。以下代码演示如何关闭非模式窗口：

C#

```
private void closeButton_Click(object sender, RoutedEventArgs e) =>
    Close();
```

## 用 [IsCancel](#) 关闭

若要启用 [\[Esc\]](#) 键以自动关闭窗口，可以将 [Button.IsCancel](#) 属性设置为 [true](#)。这仅在使用 [ShowDialog](#) 方法打开窗口时有效。

XAML

```
<Button Name="cancelButton" IsCancel="True">Cancel</Button>
```

## 隐藏窗口

可使用 [Hide](#) 方法隐藏窗口，而不是关闭窗口。与已关闭的窗口不同，可以重新打开隐藏的窗口。如果要重用窗口对象实例，请隐藏窗口，而不是将其关闭。以下代码演示如何隐藏窗口：

C#

```
private void saveButton_Click(object sender, RoutedEventArgs e) =>
    Hide();
```

## 取消关闭并隐藏

如果将按钮设计为隐藏而不是关闭窗口，用户仍可绕过该按钮并关闭窗口。系统菜单的“关闭”项和窗口非工作区的“关闭”按钮将关闭窗口，而不是隐藏窗口。假设以下场景：你的意图是隐藏窗口而不是将其关闭。

## ⊗ 注意

如果使用 `ShowDialog` 以模态方式显示窗口，则在隐藏窗口时 `DialogResult` 属性将设置为 `null`。需要通过将自己的属性添加到窗口来将状态返回给调用代码。

当窗口关闭时，将引发 `Closing` 事件。将处理程序传递给实现 `Cancel` 属性的 `CancelEventArgs`。若要防止窗口关闭，请将该属性设置为 `true`。以下代码演示如何取消关闭并隐藏窗口：

C#

```
private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
{
    // Cancel the closure
    e.Cancel = true;

    // Hide the window
    Hide();
}
```

有时，你可能不想隐藏某个窗口，但实际上会阻止用户关闭该窗口。有关详细信息，请参阅 [WPF 窗口概述：取消关闭窗口](#)。

## 另请参阅

- [WPF 窗口概述](#)
- [对话框概述](#)
- [如何打开窗口或对话框](#)
- [System.Windows.Window.Close\(\)](#)
- [System.Windows.Window.Closing](#)
- [System.Windows.Window.DialogResult](#)
- [System.Windows.Window.Hide\(\)](#)
- [System.Windows.Window.Show\(\)](#)
- [System.Windows.Window.ShowDialog\(\)](#)

 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问

.NET

.NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

题和拉取请求。有关详细信息，  
请参阅[参与者指南](#)。

 提出文档问题

 提供产品反馈

# 如何打开通用对话框 (WPF .NET)

项目 • 2023/11/21

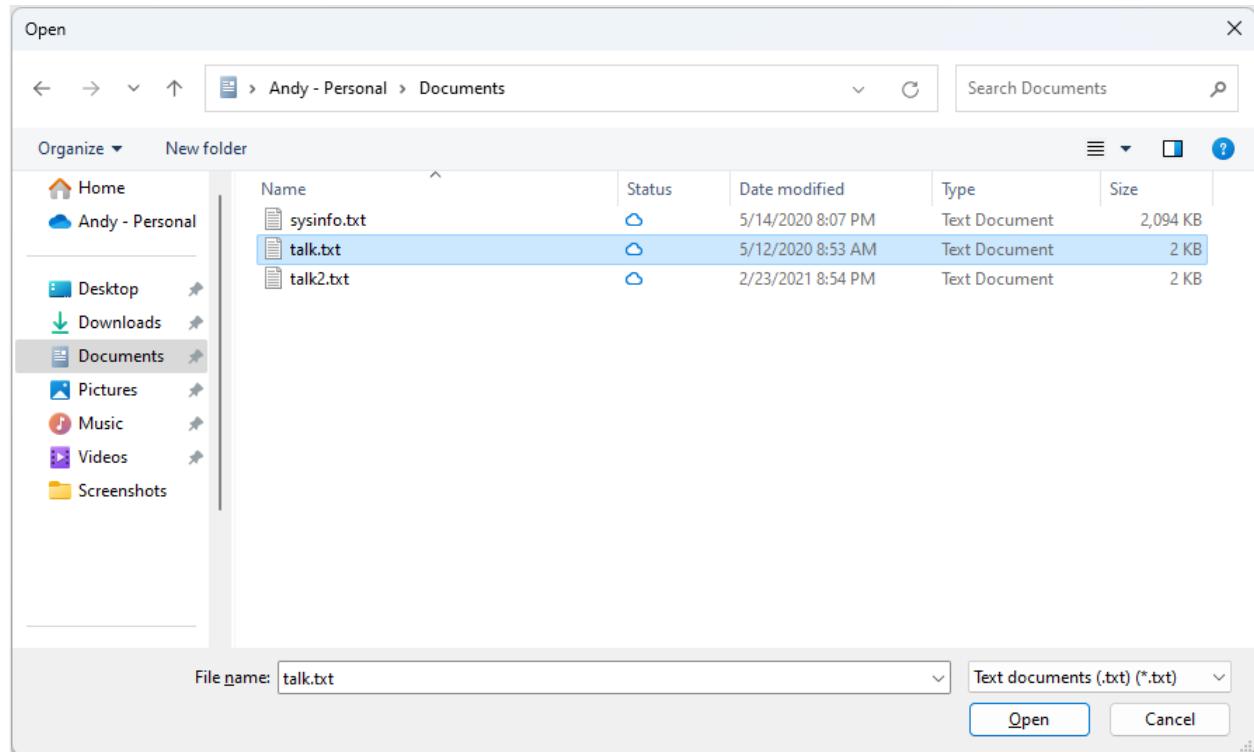
本文演示如何在 Windows Presentation Foundation (WPF) 中显示通用系统对话框。Windows 实现了所有应用程序通用的不同类型的可重用对话框，其中包括用于选择文件和打印的对话框。

由于这些对话框是由操作系统提供的，因此它们在操作系统上运行的所有应用程序之间共享。这些对话框提供一致的用户体验，被称为通用对话框。当用户在一个应用程序中使用通用对话框时，他们不需要学习如何在其他应用程序中使用该对话框。

消息框是另一种通用对话框。有关详细信息，请参阅[如何打开消息框](#)。

## “打开文件”对话框

“打开文件”对话框由文件打开功能用于检索要打开文件的名称。



常见的打开文件对话框作为 `OpenFileDialog` 类实现，并位于 `Microsoft.Win32` 命名空间中。以下代码显示了如何创建、配置和显示对话框。

C#

```
// Configure open file dialog box
var dialog = new Microsoft.Win32.OpenFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
```

```

dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show open file dialog box
bool? result = dialog.ShowDialog();

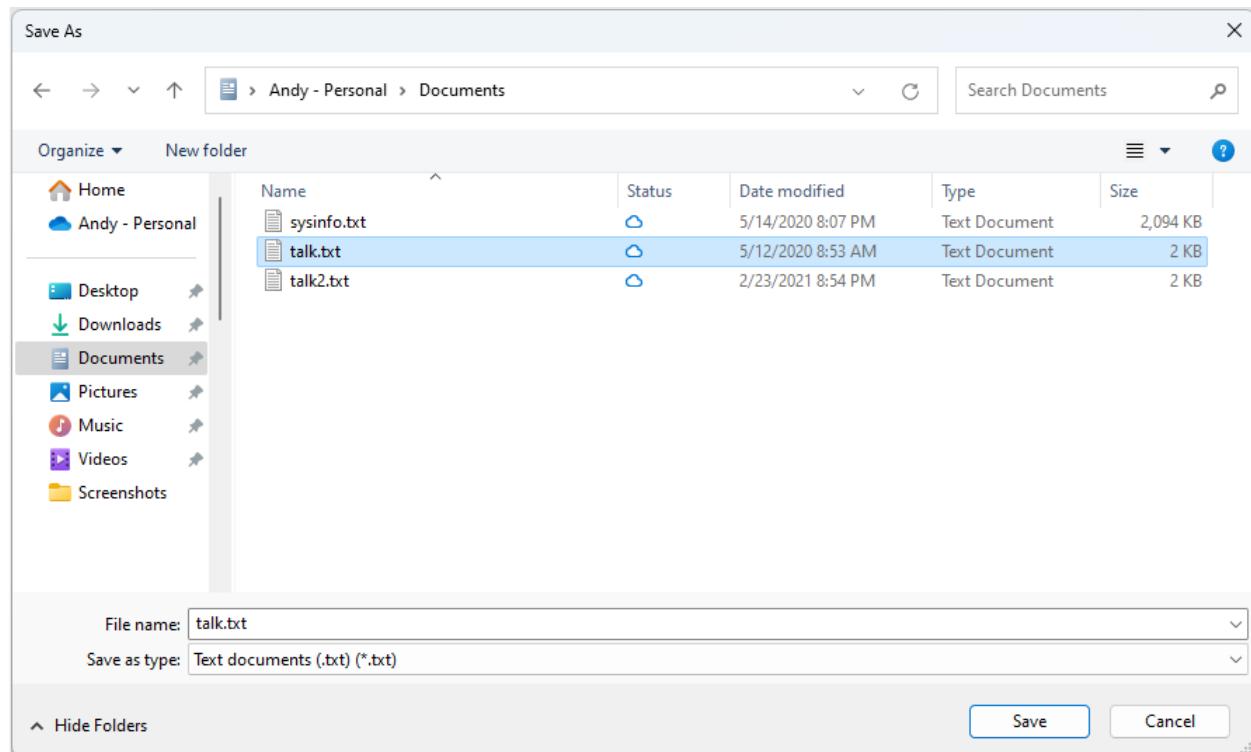
// Process open file dialog box results
if (result == true)
{
    // Open document
    string filename = dialog.FileName;
}

```

有关打开文件对话框的详细信息，请参阅 [Microsoft.Win32.OpenFileDialog](#)。

## 保存文件对话框

“保存文件”对话框由文件保存功能用于检索要保存文件的名称。



常见的保存文件对话框作为 `SaveFileDialog` 类实现，并位于 `Microsoft.Win32` 命名空间中。以下代码显示了如何创建、配置和显示对话框。

C#

```

// Configure save file dialog box
var dialog = new Microsoft.Win32.SaveFileDialog();
dialog.FileName = "Document"; // Default file name
dialog.DefaultExt = ".txt"; // Default file extension
dialog.Filter = "Text documents (.txt)|*.txt"; // Filter files by extension

// Show save file dialog box

```

```
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Save document
    string filename = dialog.FileName;
}
```

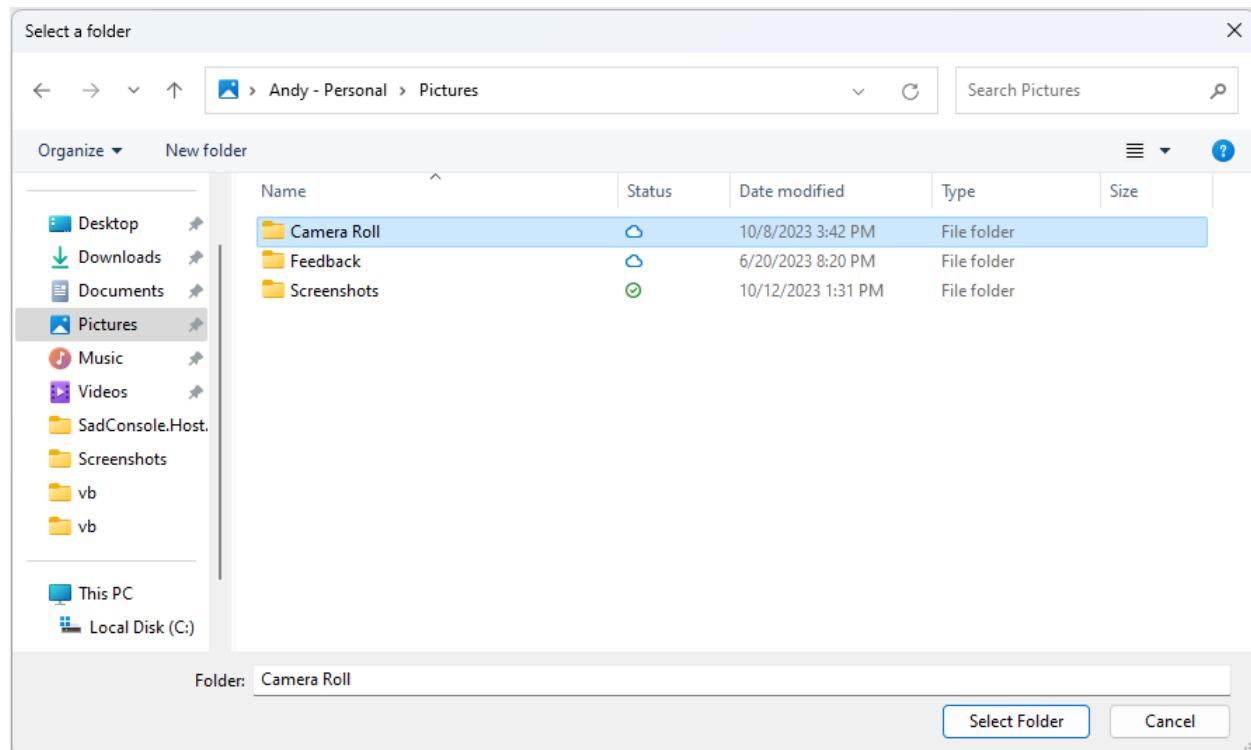
有关保存文件对话框的详细信息，请参阅 [Microsoft.Win32.SaveFileDialog](#)。

## “打开文件夹”对话框

### ① 重要

.NET 8.0 及更高版本中提供了“打开文件夹”对话框。

用户可使用“打开文件夹”对话框选择一个或多个文件夹，然后将其返回到程序。例如，如果程序显示有关文件夹的信息，例如文件夹中的文件量和文件名，则可以使用“打开文件夹”对话框来支持用户选择文件夹。



常见的打开文件夹对话框是作为 [OpenFileDialog](#) 类实现的，位于 [Microsoft.Win32](#) 命名空间中。以下代码显示了如何创建、配置和显示对话框。

C#

```
// Configure open folder dialog box
Microsoft.Win32.OpenFileDialog dialog = new();

dialog.Multiselect = false;
dialog.Title = "Select a folder";

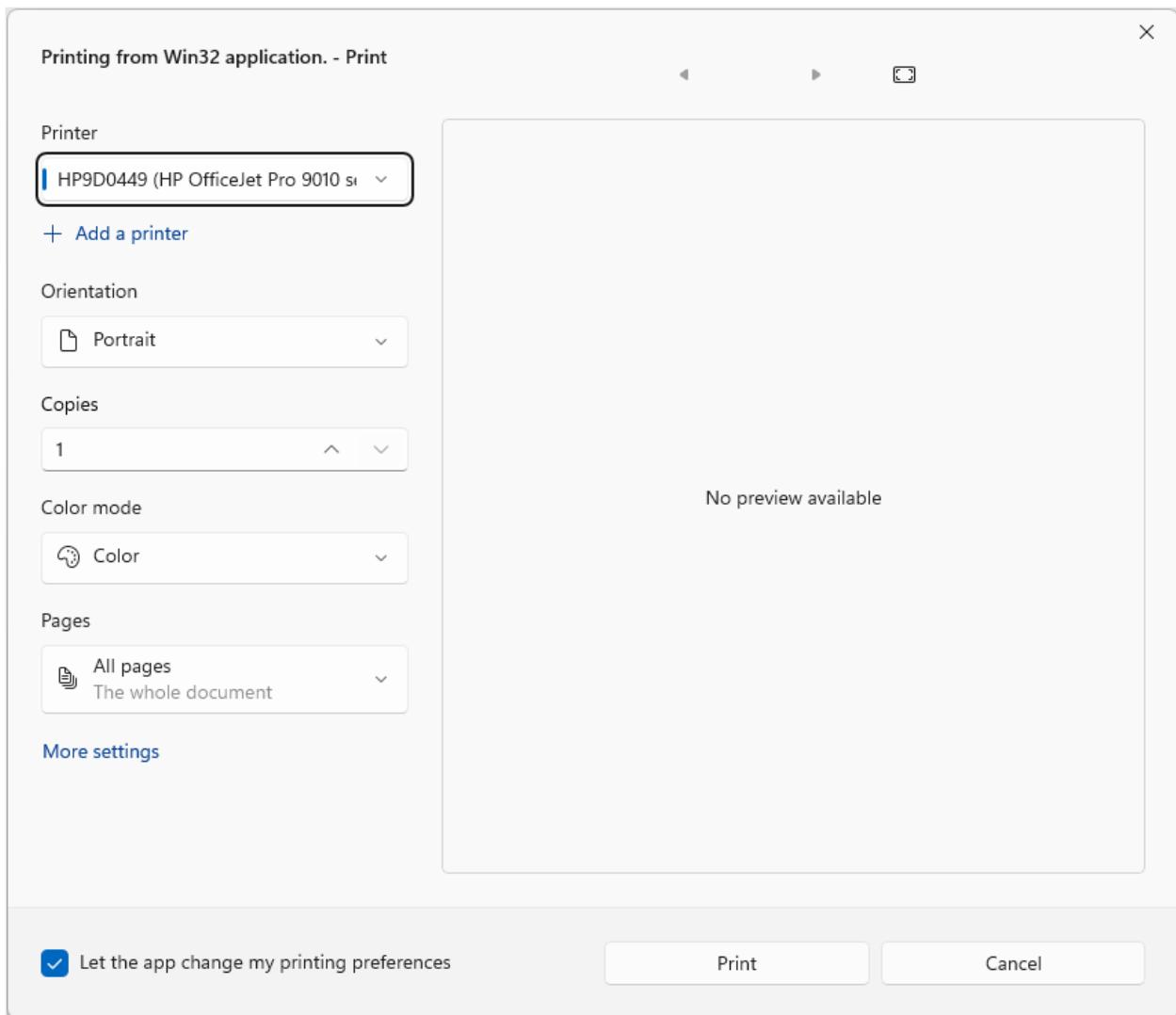
// Show open folder dialog box
bool? result = dialog.ShowDialog();

// Process open folder dialog box results
if (result == true)
{
    // Get the selected folder
    string fullPathToFolder = dialog.FolderName;
    string folderNameOnly = dialog.SafeFolderName;
}
```

有关打开文件夹对话框的详细信息，请参阅 [Microsoft.Win32.OpenFileDialog](#)。

## “打印”对话框

打印对话框由打印功能用以选择和配置用户想要将数据打印到的打印机。



常见的打印对话框作为 `PrintDialog` 类实现，并位于 `System.Windows.Controls` 命名空间中。以下代码显示了如何创建、配置和显示打印对话框。

C#

```
// Configure printer dialog box
var dialog = new System.Windows.Controls.PrintDialog();
dialog.PageRangeSelection =
    System.Windows.Controls.PageRangeSelection.AllPages;
dialog.UserPageRangeEnabled = true;

// Show save file dialog box
bool? result = dialog.ShowDialog();

// Process save file dialog box results
if (result == true)
{
    // Document was printed
}
```

有关打印对话框的详细信息，请参阅 [System.Windows.Controls.PrintDialog](#)。有关在 WPF 中打印的详细讨论，请参阅[打印概述](#)。

# 另请参阅

- [如何打开消息框](#)
- [对话框概述](#)
- [WPF 窗口概述](#)
- [Microsoft.Win32.OpenFileDialog](#)
- [Microsoft.Win32.SaveFileDialog](#)
- [System.Windows.Controls.PrintDialog](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何获取或设置主应用程序窗口 (WPF .NET)

项目 • 2023/10/13

本文将教授如何获取或设置 Windows Presentation Foundation (WPF) 的主应用程序窗口。在 WPF 应用程序中实例化的第一个 [Window](#) 由 [Application](#) 自动设置为主应用程序窗口。使用 [Application.MainWindow](#) 属性引用主窗口。

大多数情况下，项目模板会将 [Application.StartupUri](#) 设置为应用程序中的 XAML 文件，例如 `_Window1.xaml_`。这是应用程序实例化和显示的第一个窗口，它将作为主窗口。

## 💡 提示

关闭最后一个窗口后会默认关闭应用程序。此行为由 [Application.ShutdownMode](#) 属性控制。不过，也可以将应用程序配置为当 [MainWindow](#) 关闭时关闭应用程序。将 [Application.ShutdownMode](#) 设置为 [OnMainWindowClose](#) 可以启用此行为。

## 在 XAML 中设置主窗口

生成 WPF 应用程序的模板通常将 [Application.StartupUri](#) 属性设置为 XAML 文件。此属性非常有用，因为：

1. 它可以轻松更改为项目中的其他 XAML 文件。
2. 可自动实例化并显示指定的窗口。
3. 指定的窗口变为 [Application.MainWindow](#)。

### XAML

```
<Application x:Class="MainApp.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:MainApp"
    StartupUri="Window1.xaml">
</Application>
```

可以将 [Application.MainWindow](#) 设置为 XAML 声明的窗口，而不必使用 [Application.StartupUri](#)。但此处指定的窗口不会显示，必须对其可见性进行设置。

## XAML

```
<Application x:Class="MainApp.App"

    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:MainApp">

    <Application.MainWindow>
        <local:Window2 Visibility="Visible" />
    </Application.MainWindow>
</Application>
```

### ④ 注意

如果同时设置 `Application.StartupUri` 属性和 `Application.MainWindow` 属性，则应用程序启动时将显示这两个窗口。

此外，还可以使用 `Application.Startup` 事件打开窗口。有关详细信息，请参阅[使用启动事件打开窗口](#)。

## 通过代码设置主窗口

应用程序实例化的第一个窗口将自动成为主窗口并设置为 `Application.MainWindow` 属性。若要设置其他主窗口，将此属性更改为窗口：

### C#

```
Application.Current.MainWindow = new Window2();

Application.Current.MainWindow.Show();
```

如果应用程序从未创建过窗口实例，则以下代码在功能上等效于前面的代码：

### C#

```
var appWindow = new Window2();

appWindow.Show();
```

一旦创建窗口对象实例，会将其分配给 `Application.MainWindow`。

## 获取主窗口

通过检查 `Application.MainWindow` 属性，可以访问选择作为主窗口的窗口。 使用以下代码，当单击按钮时，会显示带有主窗口标题的消息框：

C#

```
private void Button_Click(object sender, RoutedEventArgs e) =>
    MessageBox.Show($"The main window's title is:
{Application.Current.MainWindow.Title}");
```

## 另请参阅

- [WPF 窗口概述](#)
- [使用启动事件打开窗口](#)
- [如何打开窗口或对话框](#)
- [System.Windows.Application](#)
- [System.Windows.Application.MainWindow](#)
- [System.Windows.Application.StartupUri](#)
- [System.Windows.Application.ShutdownMode](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 样式和模板 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 样式设置和模板化是指一套功能，这套功能使开发者和设计者能够为其产品创建极具视觉表现力的效果和一致的外观。自定义应用的外观时，需要一个强大的样式设置和模板化模型，以便维护和共享应用内部和应用之间的外观。WPF 就提供了这样的模型。

WPF 样式设置模型的另一项功能是将呈现与逻辑分离。设计者可以仅使用 XAML 处理应用外观，与此同时开发者使用 C# 或 Visual Basic 处理编程逻辑。

本概述侧重于应用的样式设置和模板化两方面，不讨论任何数据绑定概念。有关数据绑定的信息，请参阅[数据绑定概述](#)。

了解资源很重要，正是这些资源使样式和模板能够重复使用。有关资源的详细信息，请参阅[XAML 资源概述](#)。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 示例

本概述中提供的示例代码基于下图所示的[简单照片浏览应用程序](#)。



此简单照片示例使用样式设置和模板化创建极具视觉表现力的用户体验。该示例具有两个 [TextBlock](#) 元素和一个绑定到图像列表的 [ListBox](#) 控件。

有关完整示例，请参阅[样式设置和模板化示例简介](#)。

## 样式

可以将 [Style](#) 视为一种将一组属性值应用到多个元素的便捷方法。可以对从 [FrameworkElement](#) 或 [FrameworkContentElement](#) (如 [Window](#) 或 [Button](#)) 派生的任何元素使用样式。

声明样式的最常见方法是在 XAML 文件的 [Resources](#) 部分中声明为资源。由于样式是一种资源，因此它们同样遵从适用于所有资源的范围规则。简而言之，声明样式的位置会影响样式的应用范围。例如，如果在应用定义 XAML 文件的根元素中声明样式，则该样式可以在应用中的任何位置使用。

例如，以下 XAML 代码为 [TextBlock](#) 声明了两个样式，一个自动应用于所有 [TextBlock](#) 元素，另一个则必须显式引用。

XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>

    <!--A Style that extends the previous TextBlock Style with an x:Key of
    TitleText-->
    <Style BasedOn="{StaticResource {x:Type TextBlock}}"
        TargetType="TextBlock"
        x:Key="TitleText">
        <Setter Property="FontSize" Value="26"/>
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                    <LinearGradientBrush.GradientStops>
                        <GradientStop Offset="0.0" Color="#90DDDD" />
                        <GradientStop Offset="1.0" Color="#5BFFFF" />
                    </LinearGradientBrush.GradientStops>
                </Setter.Value>
            </Setter>
        </Style>
    </Window.Resources>
```

下面举例说明了如何使用上面声明的样式。

XAML

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My
    Pictures</TextBlock>
```

```
<TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```



有关详细信息，请参阅[为控件创建样式](#)。

## ControlTemplate

在 WPF 中，控件的 [ControlTemplate](#) 用于定义控件的外观。可以通过定义新的 [ControlTemplate](#) 并将其分配给控件来更改控件的结构和外观。在许多情况下，模板提供了足够的灵活性，从而无需自行编写自定义控件。

每个控件都有一个分配给 [Control.Template](#) 属性的默认模板。该模板将控件的视觉呈现与控件的功能关联起来。因为在 XAML 中定义了模板，所以无需编写任何代码即可更改控件的外观。每个模板都是为特定控件（例如 [Button](#)）设计的。

通常在 XAML 文件的 [Resources](#) 部分中将模板声明为资源。与其他所有资源一样，范围规则在此也适用。

控件模板比样式复杂得多。这是因为控件模板重写了整个控件的视觉外观，而样式只是将属性更改应用于现有控件。但是，[控件模板是通过设置 Control.Template 属性来应用的，因此可以使用样式来定义或设置模板](#)。

设计器通常允许创建现有模板的副本并进行修改。例如，在 Visual Studio WPF 设计器中，选择一个 [CheckBox](#) 控件，然后右键单击并选择“编辑模板”>“创建副本”。此命令会生成一个[用于定义模板的样式](#)。

XAML

```
<Style x:Key="CheckBoxStyle1" TargetType="{x:Type CheckBox}">
    <Setter Property="FocusVisualStyle" Value="{StaticResource
FocusVisual1}" />
    <Setter Property="Background" Value="{StaticResource
OptionMark.Static.Background1}" />
    <Setter Property="BorderBrush" Value="{StaticResource
OptionMark.Static.Border1}" />
    <Setter Property="Foreground" Value="{DynamicResource {x:Static
SystemColors.ControlTextBrushKey}}" />
    <Setter Property="BorderThickness" Value="1" />
    <Setter Property="Template">
        <Setter.Value>
```

```

<ControlTemplate TargetType="{x:Type CheckBox}">
    <Grid x:Name="templateRoot" Background="Transparent"
SapsToDevicePixels="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Border x:Name="checkBoxBorder" Background="
{TemplateBinding Background}" BorderThickness="{TemplateBinding
BorderThickness}" BorderBrush="{TemplateBinding BorderBrush}"
HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
Margin="1" VerticalAlignment="{TemplateBinding VerticalContentAlignment}">
            <Grid x:Name="markGrid">
                <Path x:Name="optionMark" Data="F1 M
9.97498,1.22334L 4.6983,9.09834L 4.52164,9.09834L 0,5.19331L
1.27664,3.52165L 4.255,6.08833L 8.33331,1.52588e-005L 9.97498,1.22334 Z "
Fill="{StaticResource OptionMark.Static.Glyph1}" Margin="1" Opacity="0"
Stretch="None"/>
                <Rectangle x:Name="indeterminateMark" Fill="
{StaticResource OptionMark.Static.Glyph1}" Margin="2" Opacity="0"/>
            </Grid>
        </Border>
        <ContentPresenter x:Name="contentPresenter"
Grid.Column="1" Focusable="False" HorizontalAlignment="{TemplateBinding
HorizontalContentAlignment}" Margin="{TemplateBinding Padding}"
RecognizesAccessKey="True" SapsToDevicePixels="{TemplateBinding
SapsToDevicePixels}" VerticalAlignment="{TemplateBinding
VerticalContentAlignment}"/>
    </Grid>
    <ControlTemplate.Triggers>
        <Trigger Property="HasContent" Value="true">
            <Setter Property="FocusVisualStyle" Value="
{StaticResource OptionMarkFocusVisualStyle}"/>
            <Setter Property="Padding" Value="4,-1,0,0"/>
        </Trigger>
    </ControlTemplate.Triggers>

```

... content removed to save space ...

通过编辑模板副本可以很好地了解模板的工作原理。与其新建一个空白模板，不如编辑副本并更改视觉呈现的某些方面来得简单。

有关示例，请参阅[为控件创建模板](#)。

## TemplateBinding

你可能已经注意到，上一部分中定义的模板资源使用了 [TemplateBinding 标记扩展](#)。对于模板方案来说，`TemplateBinding` 是绑定的优化形式，类似于使用 `{Binding RelativeSource={RelativeSource TemplatedParent}}` 构造的绑定。`TemplateBinding` 可用于将模板的各个部分绑定到控件的各个属性。例如，每个控件都有一个 `BorderThickness` 属性。可使用 `TemplateBinding` 管理此控件设置影响模板中的哪个元素。

# ContentControl 和 ItemsControl

如果在 ContentControl 的 ControlTemplate 中声明了 ContentPresenter，ContentPresenter 将自动绑定到 ContentTemplate 和 Content 属性。同样，ItemsControl 的 ControlTemplate 中的 ItemsPresenter 将自动绑定到 ItemTemplate 和 Items 属性。

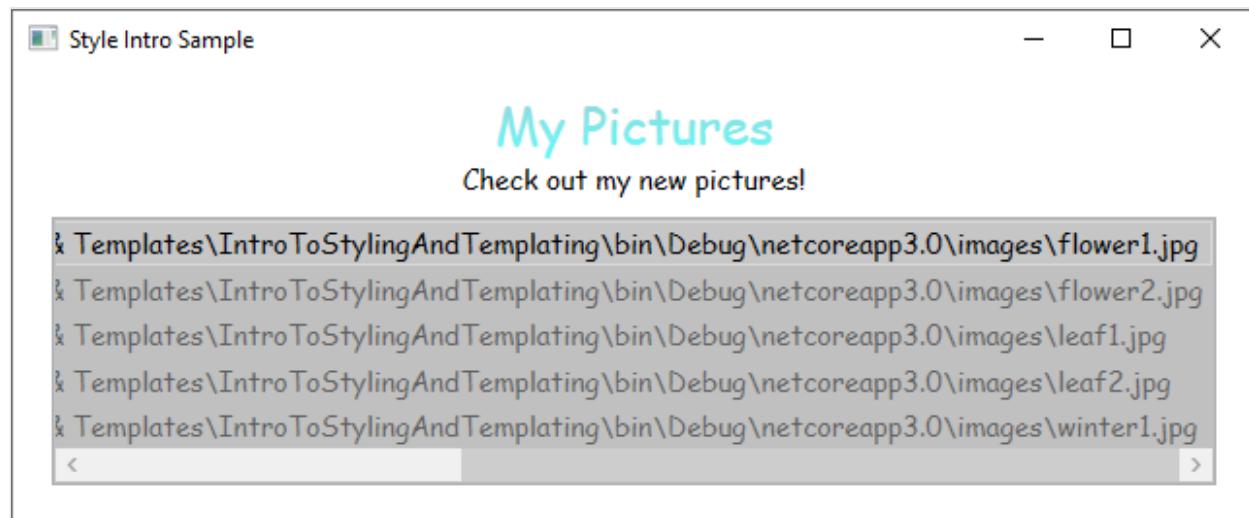
## DataTemplate

在此示例应用中，有一个绑定到照片列表的 ListBox 控件。

XAML

```
<ListBox ItemsSource="{Binding Source={StaticResource MyPhotos}}"
         Background="Silver" Width="600" Margin="10" SelectedIndex="0"/>
```

此 ListBox 当前如下所示。



大多数控件都具有某些类型的内容，并且该内容通常来自要绑定到的数据。在此示例中，数据是照片列表。在 WPF 中，使用 DataTemplate 定义数据的视觉表示形式。基本上，放入 DataTemplate 的内容决定了数据在呈现的应用中的外观。

在示例应用中，每个自定义 Photo 对象都有一个字符串类型的 Source 属性，用于指定图像的文件路径。当前，照片对象显示为文件路径。

C#

```
public class Photo
{
    public Photo(string path)
    {
        Source = path;
```

```
}

public string Source { get; }

public override string ToString() => Source;
}
```

若要使照片显示为图像，请将 [DataTemplate](#) 创建为资源。

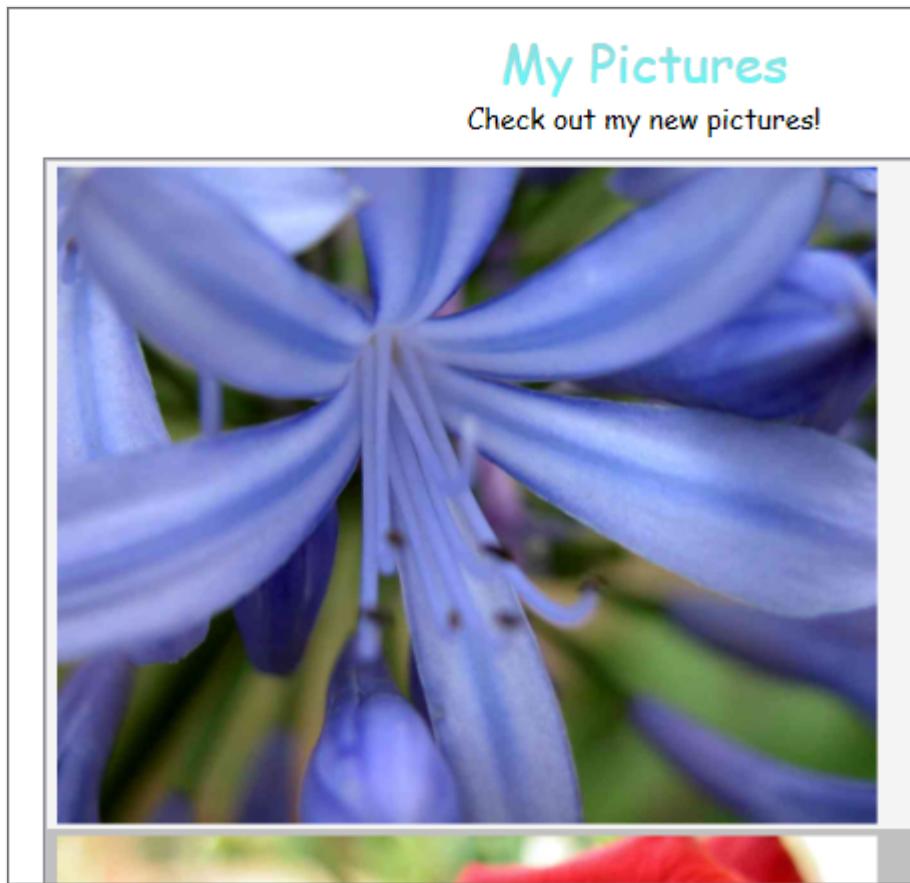
#### XAML

```
<Window.Resources>
    <!-- .... other resources .... -->

    <!--DataTemplate to display Photos as images
    instead of text strings of Paths-->
    <DataTemplate DataType="{x:Type local:Photo}">
        <Border Margin="3">
            <Image Source="{Binding Source}" />
        </Border>
    </DataTemplate>
</Window.Resources>
```

请注意，[DataType](#) 属性与 [Style](#) 的 [TargetType](#) 属性相似。如果 [DataTemplate](#) 在 [resources](#) 部分中，则在将 [DataType](#) 属性指定为某个类型并且省略 [x:Key](#) 时，只要该类型出现，就会应用 [DataTemplate](#)。始终可以选择为 [DataTemplate](#) 分配 [x:Key](#)，然后将其设置为采用 [DataTemplate](#) 类型的属性（例如 [ItemTemplate](#) 属性或 [ContentTemplate](#) 属性）的 [StaticResource](#)。

实质上，上述示例中的 [DataTemplate](#) 定义每当存在 [Photo](#) 对象时，它都应显示为 [Border](#) 内的 [Image](#)。有了这个 [DataTemplate](#)，我们的应用现在如下所示。



数据模板化模型提供其他功能。例如，如果要使用 [HeaderedItemsControl](#) 类型（例如 [Menu](#) 或 [TreeView](#)）显示包含其他集合的集合数据，可以使用 [HierarchicalDataTemplate](#)。另一个数据模板化功能是 [DataTemplateSelector](#)，该功能允许基于自定义逻辑选择要使用的 [DataTemplate](#)。有关详细信息，请参阅[数据模板化概述](#)，该概述提供不同数据模板化功能的更多深入讨论。

## 触发器

触发器在属性值发生更改或引发事件时设置属性或启动操作，例如动画。[Style](#)、[ControlTemplate](#) 和 [DataTemplate](#) 都具有可包含一组触发器的 [Triggers](#) 属性。触发器分为几种类型。

### PropertyTrigger

根据属性的值设置属性值或启动操作的 [Trigger](#) 称为属性触发器。

若要演示如何使用属性触发器，可以使每个 [ListBoxItem](#) 在未选中时部分透明。以下样式将 [ListBoxItem](#) 的 [Opacity](#) 值设置为 `0.5`。但是，当 [IsSelected](#) 属性为 `true` 时，[Opacity](#) 设置为 `1.0`。

XAML

```

<Window.Resources>
    <!-- .... other resources .... -->

    <Style TargetType="ListBoxItem">
        <Setter Property="Opacity" Value="0.5" />
        <Setter Property="MaxHeight" Value="75" />
        <Style.Triggers>
            <Trigger Property="IsSelected" Value="True">
                <Trigger.Setters>
                    <Setter Property="Opacity" Value="1.0" />
                </Trigger.Setters>
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>

```

此示例使用 Trigger 设置属性值，但请注意，Trigger 类还有 EnterActions 和 ExitActions 属性，这些属性可使触发器执行操作。

请注意，ListBoxItem 的 MaxHeight 属性设置为 75。在下图中，第三项是选中的项。



## EventTrigger 和情节提要

另一个触发器类型是 EventTrigger，用于根据某个事件的发生启动一组操作。例如，以下 EventTrigger 对象指定当鼠标指针进入 ListBoxItem 时，MaxHeight 属性在 0.2 秒的时间内动画化为值 90。当鼠标离开该项时，属性在 1 秒的时间内返回到原始值。请注意，不需要为 MouseLeave 动画指定 To 值。这是因为动画能够跟踪原始值。

XAML

```

<Style.Triggers>
    <Trigger Property="IsSelected" Value="True">
        <Trigger.Setters>
            <Setter Property="Opacity" Value="1.0" />
        </Trigger.Setters>
    </Trigger>
    <EventTrigger RoutedEvent="Mouse.MouseEnter">

```

```

<EventTrigger.Actions>
  <BeginStoryboard>
    <Storyboard>
      <DoubleAnimation
        Duration="0:0:0.2"
        Storyboard.TargetProperty="MaxHeight"
        To="90"  />
    </Storyboard>
  </BeginStoryboard>
</EventTrigger.Actions>
</EventTrigger>
<EventTrigger RoutedEvent="Mouse.MouseLeave">
  <EventTrigger.Actions>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Duration="0:0:1"
          Storyboard.TargetProperty="MaxHeight"  />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger.Actions>
</EventTrigger>
</Style.Triggers>

```

有关详细信息，请参阅[情节提要概述](#)。

在下图中，鼠标指向第三项。



## MultiTrigger、DataTrigger 和 MultiDataTrigger

除了 [Trigger](#) 和 [EventTrigger](#)，还有其他类型的触发器。 [MultiTrigger](#) 允许基于多个条件设置属性值。当条件的属性为数据绑定时，使用 [DataTrigger](#) 和 [MultiDataTrigger](#)。

## 视觉状态

控件始终处于特定的状态。例如，当鼠标在控件的表面上移动时，该控件被视为处于公用状态 [MouseOver](#)。没有特定状态的控件被视为处于公用 [Normal](#) 状态。状态分为多个

组，前面提到的状态属于 `CommonStates` 状态组。大多数控件都有两个状态组：`CommonStates` 和 `FocusStates`。在应用于控件的每个状态组中，控件始终处于每个组的一种状态，例如 `CommonStates.MouseOver` 和 `FocusStates.Unfocused`。但是，控件不能处于同一组中的两种不同状态，例如 `CommonStates.Normal` 和 `CommonStates.Disabled`。下面是大多数控件可以识别和使用的状态表。

VisualState 名称	VisualStateGroup 名称	说明
普通	CommonStates	默认状态。
MouseOver	CommonStates	鼠标指针悬停在控件上。
Pressed	CommonStates	已按下控件。
已禁用	CommonStates	已禁用控件。
Focused	FocusStates	控件有焦点。
失去焦点	FocusStates	控件没有焦点。

通过在控件模板的根元素上定义 `System.Windows.VisualStateManager`，可以在控件进入特定状态时触发动画。`VisualStateManager` 声明要监视的 `VisualStateGroup` 和 `VisualState` 的组合。当控件进入受监视状态时，将启动 `VisualStateManager` 定义的动画。

例如，以下 XAML 代码监视 `CommonStates.MouseOver` 状态，以对名为 `backgroundElement` 的元素的填充颜色进行动画处理。当控件恢复为 `CommonStates.Normal` 状态时，将还原名为 `backgroundElement` 的元素的填充颜色。

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
                    <ColorAnimation
                        Storyboard.TargetName="backgroundElement"
                        Storyboard.TargetProperty="(Shape.Fill).(
                            SolidColorBrush.Color)"
                        To="{TemplateBinding Background}"
                        Duration="0:0:0.3"/>
                </VisualState>
                <VisualState Name="MouseOver">
                    <ColorAnimation
                        Storyboard.TargetName="backgroundElement"
                        Storyboard.TargetProperty="(Shape.Fill).(
                            SolidColorBrush.Color)"
                        To="Yellow"/>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager>
    </Grid>
</ControlTemplate>
```

```
Duration="0:0:0.3" />
    </VisualState>
</VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

...

有关情节提要的详细信息，请参阅[情节提要概述](#)。

## 共享资源和主题

典型的 WPF 应用可能具有多个 UI 资源，它们可应用于整个应用。这组资源统称为应用的主题。WPF 支持使用封装为 [ResourceDictionary](#) 类的资源字典将 UI 资源打包为主题。

WPF 主题通过使用 WPF 公开的样式设置和模板化机制来定义，该机制用于自定义任何元素的视觉对象。

WPF 主题资源存储在嵌入的资源字典中。这些资源必须嵌入到已签名的程序集内，并且可以嵌入到与代码本身相同的程序集内或并行程序集内。对于包含 WPF 控件的程序集 `PresentationFramework.dll`，主题资源位于一系列并行程序集内。

该主题成为在搜索元素样式时最后查看的位置。通常，搜索先沿着元素树搜索适当的资源，然后在应用资源集合中查找，最后查询系统。这样一来，应用开发者便有机会在到达主题之前在树或应用级别上为任何对象重新定义样式。

可以将资源字典定义为单独的文件，这些文件支持跨多个应用重复使用主题。还可以通过定义多个资源字典来创建可交换的主题，这些资源字典以不同的值提供相同类型的资源。在应用级别上重新定义这些样式或其他资源是设计应用外观的推荐方法。

若要跨应用共享一组资源（包括样式和模板），可创建 XAML 文件，并定义包含对 `shared.xaml` 文件的引用的 [ResourceDictionary](#)。

XAML

```
<ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Shared.xaml" />
</ResourceDictionary.MergedDictionaries>
```

它是 `shared.xaml` 的共享，用于定义包含一组样式和画笔资源的 [ResourceDictionary](#)，从而使应用中的控件具有一致的外观。

有关详细信息，请参阅[合并资源字典](#)。

如果要为自定义控件创建主题，请参阅[控件创作概述](#)的[在主题级别定义资源](#)部分。

# 另请参阅

- [WPF 中的 Pack URI](#)
- [如何：查找由 ControlTemplate 生成的元素](#)
- [查找由 DataTemplate 生成的元素](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何为控件创建样式 (WPF .NET)

项目 • 2023/10/13

使用 Windows Presentation Foundation (WPF)，可以使用自己的可重用样式自定义现有控件的外观。可以对应用、窗口和页面全局应用样式，也可以将样式直接应用于控件。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 创建样式

可以将 [Style](#) 视为一种将一组属性值应用到一个或多个元素的便利方法。可以对从 [FrameworkElement](#) 或 [FrameworkContentElement](#) (如 [Window](#) 或 [Button](#)) 派生的任何元素使用样式。

声明样式的最常见方法是在 XAML 文件的 `Resources` 部分中声明为资源。由于样式是一种资源，因此它们同样遵从适用于所有资源的范围规则。简而言之，声明样式的位置会影响样式的应用范围。例如，如果在应用定义 XAML 文件的根元素中声明样式，则该样式可以在应用中的任何位置使用。

XAML

```
<Application x:Class="IntroToStylingAndTemplating.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:IntroToStylingAndTemplating"
    StartupUri="WindowExplicitStyle.xaml">
    <Application.Resources>
        <ResourceDictionary>
            <Style x:Key="Header1" TargetType="TextBlock">
                <Setter Property="FontSize" Value="15" />
                <Setter Property="FontWeight" Value="ExtraBold" />
            </Style>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

如果在应用的一个 XAML 文件中声明样式，则该样式只能在该 XAML 文件中使用。有关资源的范围规则的详细信息，请参阅 [XAML 资源概述](#)。

## XAML

```
<Window x:Class="IntroToStylingAndTemplating.WindowSingleResource"
    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:local="clr-namespace:IntroToStylingAndTemplating"
    mc:Ignorable="d"
    Title="WindowSingleResource" Height="450" Width="800">
<Window.Resources>

    <Style x:Key="Header1" TargetType="TextBlock">
        <Setter Property="FontSize" Value="15" />
        <Setter Property="FontWeight" Value="ExtraBold" />
    </Style>

</Window.Resources>
<Grid />
</Window>
```

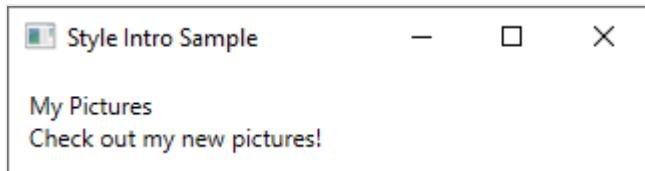
样式由 `<Setter>` 子元素组成，这些元素在应用了样式的元素上设置属性。在上面的示例中，请注意，样式已设置为通过 `TargetType` 属性应用于 `TextBlock` 类型。样式会将 `FontSize` 设置为 `15`，并将 `FontWeight` 设置为 `ExtraBold`。为样式更改的每个属性添加一个 `<Setter>`。

## 隐式应用样式

`Style` 是一种将一组属性值应用到多个元素的便利方法。例如，请考虑以下 `TextBlock` 元素及其在窗口中的默认外观。

## XAML

```
<StackPanel>
    <TextBlock>My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```



可以通过直接对每个 `TextBlock` 元素设置属性（如 `FontSize` 和 `FontFamily`）来更改默认外观。但是，如果需要让 `TextBlock` 元素共享某些属性，可以在 XAML 文件的 `Resources`

部分中创建 **Style**，如下所示。

XAML

```
<Window.Resources>
    <!--A Style that affects all TextBlocks-->
    <Style TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

将样式的 **TargetType** 设置为 **TextBlock** 类型并省略 **x:Key** 属性时，该样式将应用到样式的所有 **TextBlock** 元素，通常是 XAML 文件本身。

现在 **TextBlock** 元素如下所示。



## 显式应用样式

如果向样式添加具有值的 **x:Key** 属性，则样式将不再隐式应用于 **TargetType** 的所有元素。只有显式引用样式的元素才会应用样式。

下面是上一节中的样式，但使用 **x:Key** 属性进行了声明。

XAML

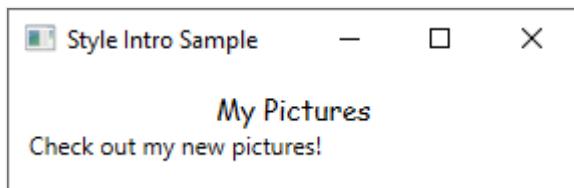
```
<Window.Resources>
    <Style x:Key="TitleText" TargetType="TextBlock">
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="FontFamily" Value="Comic Sans MS"/>
        <Setter Property="FontSize" Value="14"/>
    </Style>
</Window.Resources>
```

若要应用样式，请使用 **StaticResource** 标记扩展将元素上的 **Style** 属性设置为 **x:Key** 值，如下所示。

XAML

```
<StackPanel>
    <TextBlock Style="{StaticResource TitleText}">My Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>
```

请注意，第一个 `TextBlock` 元素已应用样式，而第二个 `TextBlock` 元素保持不变。上一节中的隐式样式已更改为声明了 `x:Key` 属性的样式，也就是说，该样式影响的唯一元素是直接引用该样式的那个元素。



在显式或隐式应用样式后，它将变为密封状态，不能更改。如果要更改已应用的样式，请创建新的样式来替换现有样式。有关更多信息，请参见 [IsSealed 属性](#)。

可以创建一个根据自定义逻辑选择要应用的样式的对象。有关示例，请参阅为 [StyleSelector](#) 类提供的示例。

## 以编程方式应用样式

若要以编程方式将已命名的样式分配到元素，请从资源集合中获取样式，并将其分配到元素的 `Style` 属性。资源集合中的项属于 `Object` 类型。因此，在将检索到的样式分配给 `Style` 属性之前，必须将它强制转换为 `System.Windows.Style`。例如，下面的代码将名为 `textblock1` 的 `TextBlock` 的样式设置为定义的样式 `TitleText`。

C#

```
textblock1.Style = (Style)Resources["TitleText"];
```

## 扩展样式

也许你希望两个 `TextBlock` 元素共享某些属性值，如 `FontFamily` 和居中的 `HorizontalAlignment`。你可能还希望文本"My Pictures"具有一些其他属性。可以通过创建基于第一个样式的新样式来实现此目的，如下所示。

XAML

```
<Window.Resources>
    <!-- .... other resources .... -->
```

```

<!--A Style that affects all TextBlocks-->
<Style TargetType="TextBlock">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="FontFamily" Value="Comic Sans MS"/>
    <Setter Property="FontSize" Value="14"/>
</Style>

<!--A Style that extends the previous TextBlock Style with an x:Key of
TitleText-->
<Style BasedOn="{StaticResource {x:Type TextBlock}}"
    TargetType="TextBlock"
    x:Key="TitleText">
    <Setter Property="FontSize" Value="26"/>
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">
                <LinearGradientBrush.GradientStops>
                    <GradientStop Offset="0.0" Color="#90DDDD" />
                    <GradientStop Offset="1.0" Color="#5BFFFF" />
                </LinearGradientBrush.GradientStops>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
</Window.Resources>

```

### XAML

```

<StackPanel>
    <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My
Pictures</TextBlock>
    <TextBlock>Check out my new pictures!</TextBlock>
</StackPanel>

```

此 `TextBlock` 样式现在居中，使用大小为 26 的 `Comic Sans MS` 字体，并将前景色设置为如示例中显示的 `LinearGradientBrush`。请注意，它会重写基本样式的 `FontSize` 值。如果有多个 `Setter` 指向 `Style` 中的同一属性，则最后声明的 `Setter` 优先。

`TextBlock` 元素现在如下所示：



此 `TitleText` 样式扩展为 `TextBlock` 类型创建的样式，该样式由 `BasedOn="`  
`{StaticResource {x:Type TextBlock}}"` 引用。还可以使用样式的 `x:Key` 扩展具有 `x:Key`

的样式。例如，如果有一个名为 `Header1` 的样式，并且你需要扩展该样式，可以使用 `BasedOn="{StaticResource Header1}"`。

## TargetType 属性与 x:Key 属性之间的关系

如前所述，将 `TargetType` 属性设置为 `TextBlock` 时，如果不为样式分配 `x:Key`，会导致将样式应用于所有 `TextBlock` 元素。在此情况下，`x:Key` 隐式设置为 `{x:Type TextBlock}`。这意味着，如果将 `x:Key` 值显式设置为除 `{x:Type TextBlock}` 以外的任何值，则 `Style` 不会自动应用于所有 `TextBlock` 元素。相反，必须将该样式（通过使用 `x:Key` 值）显式应用到 `TextBlock` 元素。如果你的样式位于资源部分中，并且你未对样式设置 `TargetType` 属性，则必须设置 `x:Key` 属性。

除了为 `x:Key` 提供默认值以外，`TargetType` 属性还指定 `setter` 属性应用到的类型。如果不指定 `TargetType`，则必须使用语法 `Property="ClassName.Property"`，通过类名称限定 `Setter` 对象中的属性。例如，必须将 `Property` 设置为 `"TextBlock.FontSize"` 或 `"Control.FontSize"`，而不是设置 `Property="FontSize"`。

另请注意，许多 WPF 控件由其他 WPF 控件的组合构成。如果创建应用于某个类型的的所有控件的样式，可能会产生意外结果。例如，如果创建一个样式，该样式以 `Window` 中的 `TextBlock` 类型为目标，那么，即使 `TextBlock` 是另一个控件（如 `ListBox`）的一部分，该样式也将应用于窗口中的所有 `TextBlock` 控件。

## 另请参阅

- [如何创建控件模板](#)
- [XAML 资源概述](#)
- [XAML 概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何为控件创建模板 (WPF.NET)

项目 • 2023/10/13

使用 Windows Presentation Foundation (WPF)，可以使用自己的可重用模板自定义现有控件的可视结构和行为。可以对应用程序、窗口和页面全局应用模板，也可以将模板直接应用于控件。需要新建控件的大多数场景均可改为为现有控件创建新模板。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

本文将介绍如何为 [Button 控件](#) 创建新的 [ControlTemplate](#)。

## 何时创建 ControlTemplate

控件有许多属性，例如 [Background](#)、[Foreground](#) 和 [FontFamily](#)。这些属性控制控件外观的不同方面，但可通过设置这些属性进行的更改有限。例如，可以从 [CheckBox](#) 中将 [Foreground](#) 属性设置为蓝色，并将 [FontStyle](#) 设置为斜体。要自定义设置控件中其他属性无法实现的控件外观时，则创建 [ControlTemplate](#)。

在多数用户界面中，按钮的总体外观相同：即一个包含某些文本的矩形。若想要创建一个圆形的按钮，可以创建一个继承自该按钮或重新创建该按钮功能的新控件。此外，新用户控件还会提供圆形视觉对象。

通过自定义现有控件的可视布局，可以避免创建新控件。借助圆形按钮，可创建具有所需可视布局的 [ControlTemplate](#)。

另一方面，如果你需要具有新功能、其他属性和新设置的控件，可创建新的 [UserControl](#)。

## 先决条件

创建新的 WPF 应用程序，在 `MainWindow.xaml`（或选择的其他窗口）的 `<Window>` 元素中设置以下属性：

属性	Value
Title	Template Intro Sample
SizeToContent	WidthAndHeight

属性	Value
MinWidth	250

将 `<Window>` 元素的内容设置为以下 XAML:

XAML

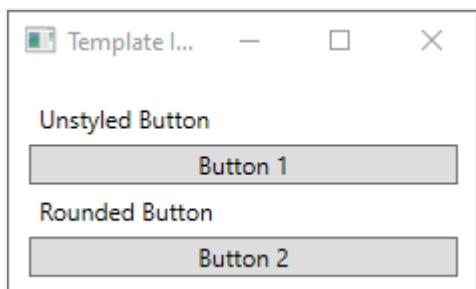
```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

最后, `MainWindow.xaml` 文件应如下所示:

XAML

```
<Window x:Class="IntroToStylingAndTemplating.Window1"
        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:local="clr-namespace:IntroToStylingAndTemplating"
        mc:Ignorable="d"
        Title="Template Intro Sample" SizeToContent="WidthAndHeight"
        MinWidth="250">
    <StackPanel Margin="10">
        <Label>Unstyled Button</Label>
        <Button>Button 1</Button>
        <Label>Rounded Button</Label>
        <Button>Button 2</Button>
    </StackPanel>
</Window>
```

如果你运行应用程序, 它将如下所示:



## 创建 ControlTemplate

声明 [ControlTemplate](#) 的最常见方法是在 XAML 文件的 `Resources` 部分中声明为资源。模板是资源，因此它们遵从适用于所有资源的相同范围规则。简言之，声明模板的位置会影响模板的应用范围。例如，如果在应用程序定义 XAML 文件的根元素中声明模板，则该模板可以在应用程序中的任何位置使用。如果在窗口中定义模板，则仅该窗口中的控件可以使用该模板。

首先，将 `Window.Resources` 元素添加到 `MainWindow.xaml` 文件：

XAML

```
<Window x:Class="IntroToStylingAndTemplating.Window2"
    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:local="clr-namespace:IntroToStylingAndTemplating"
    mc:Ignorable="d"
    Title="Template Intro Sample" SizeToContent="WidthAndHeight"
    MinWidth="250">
    <Window.Resources>

        </Window.Resources>
        <StackPanel Margin="10">
            <Label>Unstyled Button</Label>
            <Button>Button 1</Button>
            <Label>Rounded Button</Label>
            <Button>Button 2</Button>
        </StackPanel>
    </Window>
```

使用以下属性集创建新的 `<ControlTemplate>`：

属性	Value
<code>x:Key</code>	<code>roundbutton</code>
<code>TargetType</code>	<code>Button</code>

此控制模板很简单：

- 控件的根元素 `Grid`
- 用于绘制按钮圆形外观的 `Ellipse`
- 用于显示用户指定的按钮内容的 `ContentPresenter`

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
```

```
<Ellipse Fill="{TemplateBinding Background}" Stroke=" {TemplateBinding Foreground}" />
<ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
</Grid>
</ControlTemplate>
```

## TemplateBinding

创建新的 [ControlTemplate](#) 时，可能仍然想要使用公共属性更改控件外观。

[TemplateBinding](#) 标记扩展将 [ControlTemplate](#) 中元素的属性绑定到由控件定义的公共属性。 使用 [TemplateBinding](#) 时，可让控件属性用作模板参数。 换言之，设置控件属性后，该值将传递到包含 [TemplateBinding](#) 的元素。

### 椭圆形

请注意，[<Ellipse>](#) 元素的 [Fill](#) 和 [Stroke](#) 属性绑定到了控件的 [Foreground](#) 和 [Background](#) 属性。

## ContentPresenter

此外，还将 [<ContentPresenter>](#) 元素添加到了模板。此模板专为按钮设计，因此请注意该按钮继承自 [ContentControl](#)。此按钮会显示该元素的内容。可以在该按钮中设置任何内容，例如纯文本，甚至其他控件。以下两个按钮均有效：

### XAML

```
<Button>My Text</Button>

<!-- and -->

<Button>
    <CheckBox>Checkbox in a button</CheckBox>
</Button>
```

在前面的两个示例中，将文本和复选框设置为 [Button.Content](#) 属性。设置为内容的任何内容都可通过 [<ContentPresenter>](#) 显示，这是模板的功能。

若将 [ControlTemplate](#) 应用到 [ContentControl](#) 类型（例如 [Button](#)），将在元素树中搜索 [ContentPresenter](#)。若找到了 [ContentPresenter](#)，模板会自动将控件的 [Content](#) 属性绑定到 [ContentPresenter](#)。

# 使用模板

找到本文开头声明的按钮。

XAML

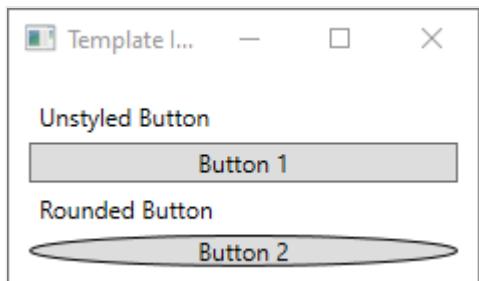
```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button>Button 2</Button>
</StackPanel>
```

将第二个按钮的 `Template` 属性设置为 `roundbutton` 资源：

XAML

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}">Button 2</Button>
</StackPanel>
```

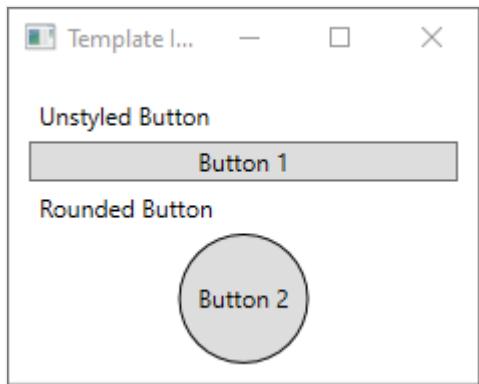
若运行项目并查看结果，将看到此按钮具有圆形背景。



你可能已注意到，此按钮不是一个圆形，而是倾斜的。由于 `<Ellipse>` 元素的工作方式，它始终会扩展并填充可用空间。将此按钮的 `width` 和 `height` 属性更改为同一个值，以使圆形均衡：

XAML

```
<StackPanel Margin="10">
    <Label>Unstyled Button</Label>
    <Button>Button 1</Button>
    <Label>Rounded Button</Label>
    <Button Template="{StaticResource roundbutton}" Width="65"
          Height="65">Button 2</Button>
</StackPanel>
```



## 添加触发器

即使已应用模板的按钮看上去与众不同，但它的行为与任何其他按钮相同。若按下此按钮，将触发 [Click](#) 事件。不过，你可能已注意到，当你将鼠标移到此按钮上方时，此按钮的视觉对象不会改变。这些视觉对象交互均由模板定义。

通过 WPF 提供的动态事件和属性系统，你可以监视特定属性是否是某个值，必要时还可重新设置模板样式。在此示例中，你将监视按钮的 [IsMouseOver](#) 属性。当鼠标位于控件上方时，使用新颜色设置 [<Ellipse>](#) 的样式。此触发器类型称为 [PropertyTrigger](#)。

必须为 [<Ellipse>](#) 添加一个可引用的名称，以便于触发器起作用。将其命名为“`backgroundElement`”。

### XAML

```
<Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}"  
Stroke="{TemplateBinding Foreground}" />
```

接下来，将新的 [Trigger](#) 添加到 [ControlTemplate.Triggers](#) 集合。此触发器将监视 [IsMouseOver](#) 事件是否为值 `true`。

### XAML

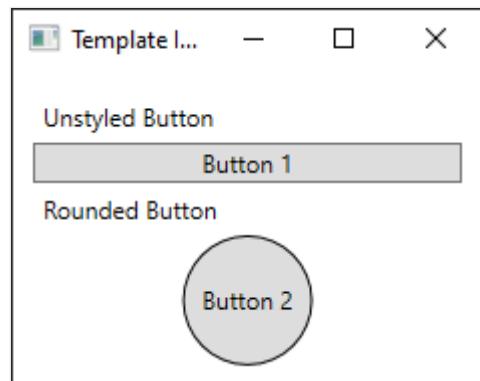
```
<ControlTemplate x:Key="roundbutton" TargetType="Button">  
    <Grid>  
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}"  
        Stroke="{TemplateBinding Foreground}" />  
        <ContentPresenter HorizontalAlignment="Center"  
        VerticalAlignment="Center" />  
    </Grid>  
    <ControlTemplate.Triggers>  
        <Trigger Property="IsMouseOver" Value="true">  
            </Trigger>  
    </ControlTemplate.Triggers>  
</ControlTemplate>
```

接下来，将 `<Setter>` 添加到 `<Trigger>`，后者会将 `<Ellipse>` 的 `Fill` 属性更改为一种新颜色。

XAML

```
<Trigger Property="IsMouseOver" Value="true">
    <Setter Property="Fill" TargetName="backgroundElement"
Value="AliceBlue"/>
</Trigger>
```

运行该项目。请注意，当你将鼠标移到按钮上方时，`<Ellipse>` 的颜色会改变。



## 使用 VisualState

视觉状态由控件定义和触发。例如，当鼠标移到控件上方时，将触发 `CommonStates.MouseOver` 状态。可以基于控件的当前状态对属性更改进行动画处理。在上一部分中，当 `IsMouseOver` 属性为 `true` 时，使用 `<PropertyTrigger>` 将按钮的背景更改为 `AliceBlue`。可改为创建一个视觉状态，来对此颜色的更改进行动画处理，以实现平稳过渡。有关 `VisualStates` 的详细信息，请参阅 [WPF 中的样式和模板](#)。

若要将 `<PropertyTrigger>` 转换为动画效果的可视状态，首先要从模板删除 `<ControlTemplate.Triggers>` 元素。

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding
Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

接下来，在控件模板的 `<Grid>` 根中，添加 `<VisualStateManager.VisualStateGroups>`，其中包含 `CommonStates` 的 `<VisualStateGroup>`。定义两种状态：`Normal` 和 `MouseOver`。

#### XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">
    <Grid>
        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup Name="CommonStates">
                <VisualState Name="Normal">
                </VisualState>
                <VisualState Name="MouseOver">
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
        <Ellipse x:Name="backgroundElement" Fill="{TemplateBinding Background}" Stroke="{TemplateBinding Foreground}" />
        <ContentPresenter HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</ControlTemplate>
```

触发 `<VisualState>` 时，将应用该状态中定义的任何动画。为每种状态创建动画。动画位于 `<Storyboard>` 元素中。有关情节提要的详细信息，请参阅[情节提要概述](#)。

- 正文

此状态对椭圆填充进行动画处理，将其还原为控件的 `Background` 颜色。

#### XAML

```
<Storyboard>
    <ColorAnimation Storyboard.TargetName="backgroundElement"
        Storyboard.TargetProperty="(Shape.Fill).(
SolidColorBrush.Color)"
        To="{TemplateBinding Background}"
        Duration="0:0:0.3"/>
</Storyboard>
```

- MouseOver

此状态对椭圆 `Background` 颜色进行动画处理，将其更改为新颜色 `Yellow`。

#### XAML

```
<Storyboard>
    <ColorAnimation Storyboard.TargetName="backgroundElement"
        Storyboard.TargetProperty="(Shape.Fill)."
```

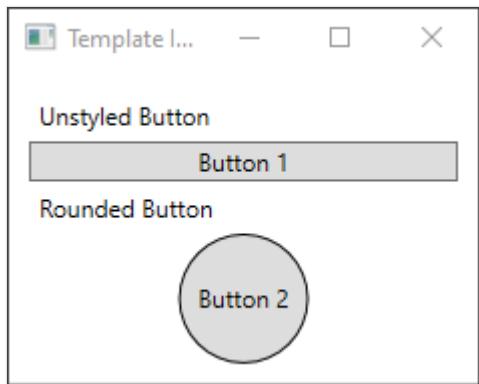
```
(SolidColorBrush.Color)"  
    To="Yellow"  
    Duration="0:0:0.3"/>  
</Storyboard>
```

现在，`<ControlTemplate>` 应如下所示。

XAML

```
<ControlTemplate x:Key="roundbutton" TargetType="Button">  
    <Grid>  
        <VisualStateManager.VisualStateGroups>  
            <VisualStateGroup Name="CommonStates">  
                <VisualState Name="Normal">  
                    <Storyboard>  
                        <ColorAnimation  
                            Storyboard.TargetName="backgroundElement"  
                            Storyboard.TargetProperty="(Shape.Fill).  
(SolidColorBrush.Color)"  
                            To="{TemplateBinding Background}"  
                            Duration="0:0:0.3"/>  
                    </Storyboard>  
                </VisualState>  
                <VisualState Name="MouseOver">  
                    <Storyboard>  
                        <ColorAnimation  
                            Storyboard.TargetName="backgroundElement"  
                            Storyboard.TargetProperty="(Shape.Fill).  
(SolidColorBrush.Color)"  
                            To="Yellow"  
                            Duration="0:0:0.3"/>  
                    </Storyboard>  
                </VisualState>  
            </VisualStateGroup>  
        </VisualStateManager.VisualStateGroups>  
        <Ellipse Name="backgroundElement" Fill="{TemplateBinding  
Background}" Stroke="{TemplateBinding Foreground}" />  
        <ContentPresenter x:Name="contentPresenter"  
            HorizontalAlignment="Center" VerticalAlignment="Center" />  
    </Grid>  
</ControlTemplate>
```

运行该项目。请注意，当你将鼠标移到按钮上方时，`<Ellipse>` 的颜色会进行动画处理。



## 后续步骤

- [为控件创建样式](#)
- [样式和模板](#)
- [XAML 资源概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

提出文档问题

提供产品反馈

# 打印文档概述 (WPF .NET)

项目 • 2023/10/13

借助 Microsoft .NET，使用 Windows Presentation Foundation (WPF) 的应用程序开发人员可以获得一组丰富的打印和打印系统管理 API。此功能的核心是 XML 纸张规范 (XPS) 文件格式和 XPS 打印路径。

## 关于 XPS

XPS 是一种电子文档格式、后台打印文件格式和页面描述语言。它是一种开放文档格式，使用 XML、开放式打包约定和其他行业标准来创建跨平台文档。XPS 简化了创建、共享、打印、查看和存档数字文档的过程。有关 XPS 的详细信息，请参阅 [XPS 文档](#)。

## XPS 打印路径

[XPS 打印路径](#)是一项 Windows 功能，它重新定义了在 Windows 应用程序中处理打印的方式。XPS 打印路径可以替换：

- 文档呈现语言，例如 RTF 格式或可移植文档格式。
- 打印后台处理程序格式，例如 Windows 图元文件或增强型图元文件 (EMF)。
- 页面描述语言，例如打印机命令语言或 PostScript。

因此，从应用程序发布到打印机驱动程序或设备中的最终处理，XPS 打印路径一直保持 XPS 格式。

XPS 文档的打印后台处理程序支持 XPS 打印路径和 GDI 打印路径。XPS 打印路径本身使用 XPS 后台打印文件并需要 XPS 打印机驱动程序。XPS 打印路径基于 [XPS 打印机驱动程序](#) (XPSSDrv) 模型构建而成。

XPS 打印路径的优势包括：

- 所见即所得的打印支持。
- 对高级颜色配置文件的本机支持，例如每通道 32 位、CMYK 颜色模型、已命名的颜色、n 墨迹以及透明度和渐变。
- 改进的打印性能 - XPS 功能和增强功能仅适用于以 XPS 打印路径为目标的应用程序。
- 行业标准 XPS 格式。

对于基本的打印场景，可以使用简单直观的 API 以及用于打印配置和作业提交的标准 UI。对于高级场景，API 支持 UI 自定义或完全没有 UI、同步或异步打印以及批量打印功能。简单和高级选项都以完全或部分信任模式提供打印支持。

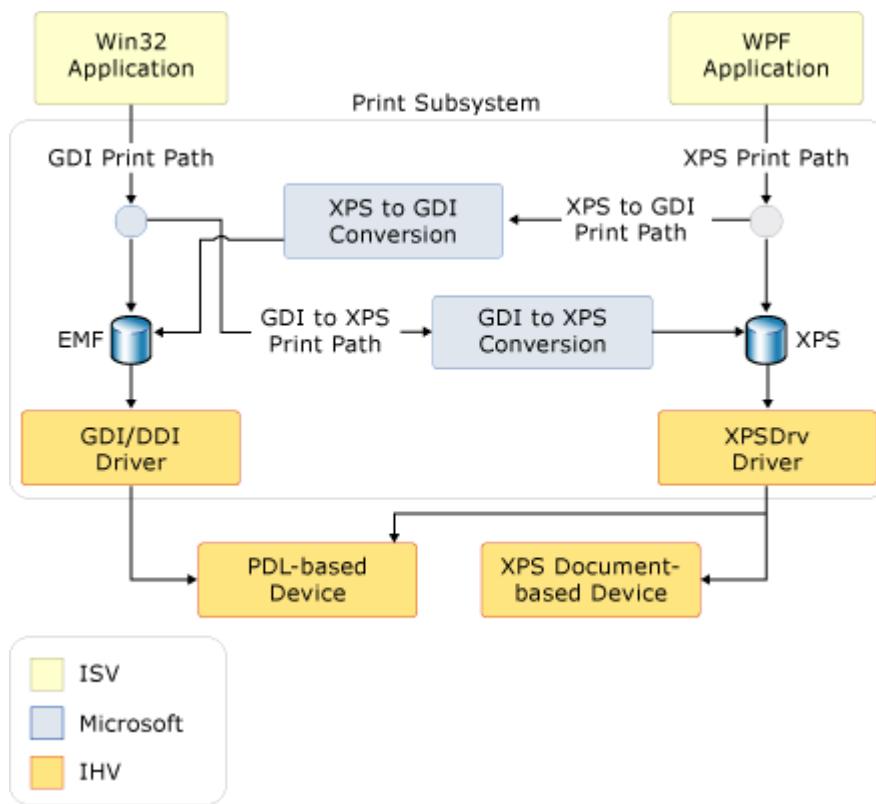
XPS 在设计时考虑了扩展性，因此能够以模块化方式将特性和功能添加到 XPS。扩展性功能包括：

- 支持快速扩展设备功能的打印架构。架构的公共部分会定期更新，以添加所需设备功能。有关详细信息，请参阅[可扩展体系结构](#)。
- XPSDrv 驱动程序使用的可扩展筛选器管道，支持 XPS 文档的直接打印和可缩放打印。有关详细信息，请参阅[XPSDrv 打印机驱动程序](#)。

## 打印路径体系结构

WPF 应用程序本机支持 XPS 打印路径，并且可以使用 XPS 打印 API 直接打印到 XPSDrv 驱动程序。如果写入操作的目标打印队列没有 XPSDrv 驱动程序，[XpsDocumentWriter](#) 类的 [Write](#) 和 [WriteAsync](#) 方法会自动将内容从 XPS 转换为 GDI 格式（适用于 GDI 打印路径）。

下图显示了打印子系统，并定义了 Microsoft 以及独立软件和硬件供应商提供的部分。



## 基本 XPS 打印

WPF 具有支持基本和高级打印功能的打印 API。对于那些不需要大量打印自定义或访问完整 XPS 功能集的应用程序，基本打印支持可能就足够了。基本打印支持通过 [PrintDialog](#) 控件提供，该控件只需最少的配置、具有熟悉的 UI 并支持许多 XPS 功能。

### PrintDialog

[System.Windows.Controls.PrintDialog](#) 控件为 UI、配置和 XPS 作业提交提供单一入口点。 若要了解如何实例化和使用该控件，请参阅[如何显示打印对话框](#)。

## 高级 XPS 打印

若要访问完整的 XPS 功能集，请使用高级打印 API。本部分介绍了几个相关的 API，包括 [PrintTicket](#)、[PrintCapabilities](#)、[PrintServer](#)、[PrintQueue](#) 和 [XpsDocumentWriter](#)。有关 XPS 打印路径 API 的完整列表，请参阅 [System.Windows.Xps](#) 和 [System.Printing](#) 命名空间。

### PrintTicket 和 PrintCapabilities

[PrintTicket](#) 和 [PrintCapabilities](#) 类是高级 XPS 功能的基础。这两个对象都包含由打印架构定义的面向打印的功能的 XML 格式结构。这些功能包括双面打印、自动分页和装订。[PrintTicket](#) 指示打印机如何处理打印作业。[PrintCapabilities](#) 类定义打印机的各种功能。通过查询打印机的功能，可以创建充分利用打印机的受支持功能的 [PrintTicket](#)。同样，可以避免不受支持的功能。

以下示例查询打印机的 [PrintCapabilities](#) 并使用代码创建 [PrintTicket](#)。

C#

```
/// <summary>
/// Returns a print ticket, which is a set of instructions telling a printer
/// how
/// to set its various features, such as duplexing, collating, and stapling.
/// </summary>
/// <param name="printQueue">The print queue to print to.</param>
/// <returns>A print ticket.</returns>
public static PrintTicket GetPrintTicket(PrintQueue printQueue)
{
    PrintCapabilities printCapabilites = printQueue.GetPrintCapabilities();

    // Get a default print ticket from printer.
    PrintTicket printTicket = printQueue.DefaultPrintTicket;

    // Modify the print ticket.
    if (printCapabilites.CollationCapability.Contains(Collation.Collated))
        printTicket.Collation = Collation.Collated;
    if
    (printCapabilites.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge))
        printTicket.Duplexing = Duplexing.TwoSidedLongEdge;
    if
    (printCapabilites.StaplingCapability.Contains(Stapling.StapleDualLeft))
        printTicket.Stapling = Stapling.StapleDualLeft;

    // Returns a print ticket, which is a set of instructions telling a
    printer how
```

```
// to set its various features, such as duplexing, collating, and
// stapling.
    return printTicket;
}
```

## PrintServer 和 PrintQueue

`PrintServer` 类表示网络打印服务器, `PrintQueue` 类表示打印机以及与其关联的输出作业队列。这些 API 一起支持对服务器的打印作业进行高级管理。`PrintServer` 或其派生类之一用于管理 `PrintQueue`。

以下示例创建 `LocalPrintServer` 并使用代码访问本地计算机的 `PrintQueueCollection`。

C#

```
/// <summary>
/// Return a collection of print queues, which individually hold the
features or states
/// of a printer as well as common properties for all print queues.
/// </summary>
/// <returns>A collection of print queues.</returns>
public static PrintQueueCollection GetPrintQueues()
{
    // Create a LocalPrintServer instance, which represents
    // the print server for the local computer.
    LocalPrintServer localPrintServer = new();

    // Get the default print queue on the local computer.
    //PrintQueue printQueue = localPrintServer.DefaultPrintQueue;

    // Get all print queues on the local computer.
    PrintQueueCollection printQueueCollection =
localPrintServer.GetPrintQueues();

    // Return a collection of print queues, which individually hold the
features or states
    // of a printer as well as common properties for all print queues.
    return printQueueCollection;
}
```

## XpsDocumentWriter

`XpsDocumentWriter` 及其许多 `Write` 和 `WriteAsync` 方法用于将 XPS 文档添加到 `PrintQueue`。例如, `Write(FixedDocumentSequence, PrintTicket)` 方法用于将具有打印票证的 XPS 文档同步添加到队列中。`WriteAsync(FixedDocumentSequence, PrintTicket)` 方法用于将具有打印票证的 XPS 文档异步添加到队列中。

以下示例使用代码创建 `XpsDocumentWriter` 并以同步和异步方式将 XPS 文档添加到 `PrintQueue`。

C#

```
/// <summary>
/// Asynchronously, add the XPS document together with a print ticket to the
/// print queue.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file.</param>
/// <param name="printQueue">The print queue to print to.</param>
/// <param name="printTicket">The print ticket for the selected print queue.
/// </param>
public static void PrintXpsDocumentAsync(string xpsFilePath, PrintQueue
printQueue, PrintTicket printTicket)
{
    // Create an XpsDocumentWriter object for the print queue.
    XpsDocumentWriter xpsDocumentWriter =
PrintQueue.CreateXpsDocumentWriter(printQueue);

    // Open the selected document.
    XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

    // Get a fixed document sequence for the selected document.
    FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

    // Asynchronously, add the XPS document together with a print ticket to
    // the print queue.
    xpsDocumentWriter.WriteAsync(fixedDocSeq, printTicket);
}

/// <summary>
/// Synchronously, add the XPS document together with a print ticket to the
/// print queue.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file.</param>
/// <param name="printQueue">The print queue to print to.</param>
/// <param name="printTicket">The print ticket for the selected print queue.
/// </param>
public static void PrintXpsDocument(string xpsFilePath, PrintQueue
printQueue, PrintTicket printTicket)
{
    // Create an XpsDocumentWriter object for the print queue.
    XpsDocumentWriter xpsDocumentWriter =
PrintQueue.CreateXpsDocumentWriter(printQueue);

    // Open the selected document.
    XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

    // Get a fixed document sequence for the selected document.
    FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();
```

```
// Synchronously, add the XPS document together with a print ticket to  
// the print queue.  
xpsDocumentWriter.Write(fixedDocSeq, printTicket);  
}
```

## GDI 打印路径

尽管 WPF 应用程序本机支持 XPS 打印路径，但它们也可以通过以下方式输出到 GDI 打印路径：调用 [XpsDocumentWriter](#) 类的 [Write](#) 或 [WriteAsync](#) 方法之一，并为非 XpsDrv 打印机选择打印队列。

对于不需要 XPS 功能或支持的应用程序，当前的 GDI 打印路径保持不变。有关 GDI 打印路径和各种 XPS 转换选项的详细信息，请参阅 [Microsoft XPS 文档转换器 \(MXDC\)](#) 和 [XPSDrv 打印机驱动程序](#)。

## XPSDrv 驱动程序模型

在打印到支持 XPS 的打印机或驱动程序时，XPS 打印路径将 XPS 用作本机后台打印格式，从而提高后台处理程序的效率。与 EMF（将应用程序输出表示为对呈现服务的 GDI 进行的一系列调用）不同，XPS 后台打印格式表示文档。因此，当 XPS 后台打印文件输出到基于 XPS 的打印机驱动程序时，它们不需要进一步解释，因为驱动程序会直接对采用该格式的数据进行操作。此功能消除了 EMF 文件和基于 GDI 的打印驱动程序所需的数据和颜色空间转换。

简化的后台打印过程不需要在后台打印文档之前生成中间后台打印文件（例如 EMF 数据文件）。通过减小后台打印文件的大小，XPS 打印路径可以减少网络流量并提高打印性能。与其 EMF 等效项相比，使用 XPS 打印路径时，XPS 后台打印文件通常会变小。可通过几种机制来缩小后台打印文件：

- **字体子集划分**，仅将文档中使用的字符存储在 XPS 文件中。
- **高级图形支持**，本机支持透明度和渐变基元以避免 XPS 内容光栅化。
- **公共资源的识别**，例如在文档中多次使用的公司徽标图像。公共资源被视为共享资源，只加载一次。
- **ZIP 压缩**，用于所有 XPS 文档。

如果矢量图形高度复杂、分为多层或编写效率低下，则 XPS 后台打印文件可能不会变小。与 GDI 后台打印文件不同，XPS 文件嵌入设备字体和基于计算机的字体以用于屏幕显示，不过这两种字体都划分了子集，并且打印机驱动程序可以在将文件传输到打印机之前删除设备字体。



提示

还可以使用 `PrintQueue.AddJob` 方法打印 XPS 文件。有关详细信息，请参阅[如何打印 XPS 文件](#)。

## 另请参阅

- [PrintDialog](#)
- [XpsDocumentWriter](#)
- [XpsDocument](#)
- [PrintTicket](#)
- [PrintCapabilities](#)
- [PrintServer](#)
- [PrintQueue](#)
- [操作说明主题](#)
- [WPF 中的文档](#)
- [XPS 文档](#)
- [文档序列化和存储](#)
- [Microsoft XPS 文档转换器 \(MXDC\)](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何显示打印对话框 (WPF .NET)

项目 • 2023/10/13

希望从应用程序打印？可使用 [PrintDialog](#) 类打开标准 Microsoft Windows 打印对话框。操作方法如下。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## ② 备注

此处讨论的用于 WPF 的 `System.Windows.Controls.PrintDialog` 控件不应与 Windows 窗体的 `System.Windows.Forms.PrintDialog` 组件混淆。

`PrintDialog` 类为打印配置和打印作业提交提供单一控件。该控件易于使用，可使用 XAML 标记或代码进行实例化。下面的示例使用代码创建和显示 `PrintDialog` 实例。

可使用打印对话框配置打印选项，例如：

- 仅打印特定页面范围。
- 从计算机上安装的打印机中选择。可使用“Microsoft XPS 文档编写器”选项创建以下文档类型：
  - XML 纸张规范 (XPS)
  - Open XML 纸张规范 (OpenXPS)

## 打印整个文档

本示例打印 XPS 文档的所有页面。默认情况下，代码将执行以下操作：

1. 打开一个打印对话框窗口，提示用户选择打印机并启动打印作业。
2. 使用 XPS 文档的内容实例化 `XpsDocument` 对象。
3. 使用 `XpsDocument` 对象生成一个包含 XPS 文档所有页面的 `DocumentPaginator` 对象。
4. 调用 `PrintDocument` 方法，传入 `DocumentPaginator` 对象，以将所有页面发送到指定的打印机。

C#

```

/// <summary>
/// Print all pages of an XPS document.
/// Optionally, hide the print dialog window.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file</param>
/// <param name="hidePrintDialog">Whether to hide the print dialog window
/// (shown by default)</param>
/// <returns>Whether the document printed</returns>
public static bool PrintWholeDocument(string xpsFilePath, bool
hidePrintDialog = false)
{
    // Create the print dialog object and set options.
    PrintDialog printDialog = new();

    if (!hidePrintDialog)
    {
        // Display the dialog. This returns true if the user presses the
        Print button.
        bool? isPrinted = printDialog.ShowDialog();
        if (isPrinted != true)
            return false;
    }

    // Print the whole document.
    try
    {
        // Open the selected document.
        XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

        // Get a fixed document sequence for the selected document.
        FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();

        // Create a paginator for all pages in the selected document.
        DocumentPaginator docPaginator = fixedDocSeq.DocumentPaginator;

        // Print to a new file.
        printDialog.PrintDocument(docPaginator, $"Printing
{Path.GetFileName(xpsFilePath)}");

        return true;
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);

        return false;
    }
}

```

## 打印某个页面范围

有时，你只希望打印 XPS 文档中的特定页面范围。为此，我们扩展了抽象 [DocumentPaginator](#) 类以添加对页面范围的支持。默认情况下，代码将执行以下操作：

1. 打开一个打印对话框窗口，提示用户选择打印机、指定页面范围并启动打印作业。
2. 使用 XPS 文档的内容实例化 [XpsDocument](#) 对象。
3. 使用 [XpsDocument](#) 对象生成一个包含 XPS 文档所有页面的默认 [DocumentPaginator](#) 对象。
4. 创建支持页范围的扩展 [DocumentPaginator](#) 类的实例，传入默认 [DocumentPaginator](#) 对象以及 [PrintDialog](#) 返回的 [PageRange](#) 结构。
5. 调用 [PrintDocument](#) 方法，传入扩展 [DocumentPaginator](#) 类的实例，以将指定的页面范围发送到指定的打印机。

C#

```
/// <summary>
/// Print a specific range of pages within an XPS document.
/// </summary>
/// <param name="xpsFilePath">Path to source XPS file</param>
/// <returns>Whether the document printed</returns>
public static bool PrintDocumentPageRange(string xpsFilePath)
{
    // Create the print dialog object and set options.
    PrintDialog printDialog = new()
    {
        UserPageRangeEnabled = true
    };

    // Display the dialog. This returns true if the user presses the Print
    // button.
    bool? isPrinted = printDialog.ShowDialog();
    if (isPrinted != true)
        return false;

    // Print a specific page range within the document.
    try
    {
        // Open the selected document.
        XpsDocument xpsDocument = new(xpsFilePath, FileAccess.Read);

        // Get a fixed document sequence for the selected document.
        FixedDocumentSequence fixedDocSeq =
            xpsDocument.GetFixedDocumentSequence();

        // Create a paginator for all pages in the selected document.
        DocumentPaginator docPaginator = fixedDocSeq.DocumentPaginator;

        // Check whether a page range was specified in the print dialog.
        if (printDialog.PageRangeSelection == PageRangeSelection.UserPages)
        {
            // Create a document paginator for the specified range of pages.
            docPaginator = new DocPaginator(fixedDocSeq.DocumentPaginator,
```

```
printDialog.PageRange);
}

    // Print to a new file.
    printDialog.PrintDocument(docPaginator, $"Printing
{Path.GetFileName(xpsFilePath)}");

        return true;
}
catch (Exception e)
{
    MessageBox.Show(e.Message);

        return false;
}
}

/// <summary>
/// Extend the abstract DocumentPaginator class to support page range
printing. This class is based on the following online resources:
///
/// https://www.thomasclaudiushuber.com/2009/11/24/wpf-printing-how-to-print-a-pagerange-with-wpf-printdialog-that-means-the-user-can-select-specific-pages-and-only-these-pages-are-printed/
///
/// https://social.msdn.microsoft.com/Forums/vstudio/en-US/9180e260-0791-4f2d-962d-abcb22ba8d09/how-to-print-multiple-page-ranges-with-wpf-printdialog
///
/// https://social.msdn.microsoft.com/Forums/en-US/841e804b-9130-4476-8709-0d2854c11582/exception-quotfixedpage-cannot-contain-another-fixedpagequot-when-printing-to-the-xps-document?forum=wpf
/// </summary>
public class DocPaginator : DocumentPaginator
{
    private readonly DocumentPaginator _documentPaginator;
    private readonly int _startPageIndex;
    private readonly int _endPageIndex;
    private readonly int _pageCount;

    public DocPaginator(DocumentPaginator documentPaginator, PageRange
pageRange)
    {
        // Set document paginator.
        _documentPaginator = documentPaginator;

        // Set page indices.
        _startPageIndex = pageRange.PageFrom - 1;
        _endPageIndex = pageRange.PageTo - 1;

        // Validate and set page count.
        if (_startPageIndex >= 0 &&
            _endPageIndex >= 0 &&
            _startPageIndex <= _documentPaginator.PageCount - 1 &&
            _endPageIndex <= _documentPaginator.PageCount - 1 &&
```

```
        _startPageIndex <= _endPageIndex)
        _pageCount = _endPageIndex - _startPageIndex + 1;
    }

    public override bool IsPageCountValid => true;

    public override int PageCount => _pageCount;

    public override IDocumentPaginatorSource Source =>
_documentPaginator.Source;

    public override Size PageSize { get => _documentPaginator.PageSize; set
=> _documentPaginator.PageSize = value; }

    public override DocumentPage GetPage(int pageNumber)
    {
        DocumentPage documentPage =
_documentPaginator.GetPage(_startPageIndex + pageNumber);

        // Workaround for "FixedPageInPage" exception.
        if (documentPage.Visual is FixedPage fixedPage)
        {
            var containerVisual = new ContainerVisual();
            foreach (object child in fixedPage.Children)
            {
                var childClone =
(UIElement)child.GetType().GetMethod("MemberwiseClone",
BindingFlags.Instance | BindingFlags.NonPublic).Invoke(child, null);

                FieldInfo parentField =
childClone.GetType().GetField("_parent", BindingFlags.Instance |
BindingFlags.NonPublic);
                if (parentField != null)
                {
                    parentField.SetValue(childClone, null);
                    containerVisual.Children.Add(childClone);
                }
            }
        }

        return new DocumentPage(containerVisual, documentPage.Size,
documentPage.BleedBox, documentPage.ContentBox);
    }

    return documentPage;
}
}
```

💡 提示

尽管可使用 `PrintDocument` 方法打印而无需打开打印对话框，但出于性能原因，最好使用 `AddJob` 方法，或使用 `XpsDocumentWriter` 的众多 `Write` 和 `WriteAsync` 方法之一。有关此内容的详细信息，请参阅[如何打印 XPS 文件和打印文档概述](#)。

## 另请参阅

- [PrintDialog](#)
- [WPF 中的文档](#)
- [如何打印 XPS 文件](#)
- [打印文档概述](#)
- [Microsoft XPS 文档编写器](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何打印 XPS 文件 (WPF .NET)

项目 • 2023/10/13

有时你希望无需打开打印对话框即可向打印队列添加新的打印作业。可以使用其中一种 `PrintQueue.AddJob` 方法执行此操作。操作方法如下。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

在以下示例中，我们使用 `AddJob` 的几种重载之一的 `AddJob(String, String, Boolean)` 方法：

- 将 XML 纸张规范 (XPS) 文档的新打印作业添加到默认打印队列中。
- 命名新作业。
- 指定是否应验证 XPS 文档（通过使用 `fastCopy` 参数）。

使用 `AddJob(String, String, Boolean)` 方法时，参数 `fastCopy` 的值是个关键考虑因素：

- 如果将 `fastCopy` 参数设置为 `true`，则跳过 XPS 验证，打印作业将快速在后台处理，而无需逐页进度反馈。
- 如果将 `fastCopy` 参数设置为 `false`，则调用 `AddJob` 方法的线程必须具有单线程单元状态，否则将引发异常。有关详细信息，请参阅“备注”部分了解 `AddJob(String, String, Boolean)`。

## 向队列添加新打印作业

本示例将一个或多个 XPS 文档添加到默认队列。代码将执行以下操作：

1. 使用 `Task.Run` 来避免阻碍 UI 线程，因为没有异步版本的 `AddJob`。
2. 如果 `fastCopy` 参数值为 `false`，则以单线程单元状态在线程上运行 `AddJob(String, String, Boolean)`。
3. 获取对 `LocalPrintServer` 的默认 `PrintQueue` 的引用。
4. 对打印队列引用调用 `AddJob(String, String, Boolean)`，传入作业名称、XPS 文档路径和 `fastCopy` 参数。

如果队列未暂停且打印机正在运行，则在打印作业到达打印队列的顶部时，打印作业将自动开始打印。

## 💡 提示

若要避免在将打印作业添加到默认队列时出现“将输出文件另存为”对话框，请确保默认打印机不是“Microsoft XPS 文档编写器”、“Microsoft 打印到 PDF”或其他“打印到文件”选项。

C#

```
/// <summary>
/// Asynchronously, add a batch of XPS documents to the print queue using a
PrintQueue.AddJob method.
/// Handle the thread apartment state required by the PrintQueue.AddJob
method.
/// </summary>
/// <param name="xpsFilePaths">A collection of XPS documents.</param>
/// <param name="fastCopy">Whether to validate the XPS documents.</param>
/// <returns>Whether all documents were added to the print queue.</returns>
public static async Task<bool> BatchAddToPrintQueueAsync(IEnumerable<string>
xpsFilePaths, bool fastCopy = false)
{
    bool allAdded = true;

    // Queue some work to run on the ThreadPool.
    // Wait for completion without blocking the calling thread.
    await Task.Run(() =>
    {
        if (fastCopy)
            allAdded = BatchAddToPrintQueue(xpsFilePaths, fastCopy);
        else
        {
            // Create a thread to call the PrintQueue.AddJob method.
            Thread newThread = new(() =>
            {
                allAdded = BatchAddToPrintQueue(xpsFilePaths, fastCopy);
            });

            // Set the thread to single-threaded apartment state.
            newThread.SetApartmentState(ApartmentState.STA);

            // Start the thread.
            newThread.Start();

            // Wait for thread completion. Blocks the calling thread,
            // which is a ThreadPool thread.
            newThread.Join();
        }
    });
    return allAdded;
}
```

```
/// <summary>
/// Add a batch of XPS documents to the print queue using a
PrintQueue.AddJob method.
/// </summary>
/// <param name="xpsFilePaths">A collection of XPS documents.</param>
/// <param name="fastCopy">Whether to validate the XPS documents.</param>
/// <returns>Whether all documents were added to the print queue.</returns>
public static bool BatchAddToPrintQueue(IEnumerable<string> xpsFilePaths,
bool fastCopy)
{
    bool allAdded = true;

    // To print without getting the "Save Output File As" dialog, ensure
    // that your default printer is not the Microsoft XPS Document Writer,
    // Microsoft Print to PDF, or other print-to-file option.

    // Get a reference to the default print queue.
    PrintQueue defaultPrintQueue = LocalPrintServer.GetDefaultPrintQueue();

    // Iterate through the document collection.
    foreach (string xpsFilePath in xpsFilePaths)
    {
        // Get document name.
        string xpsFileName = Path.GetFileName(xpsFilePath);

        try
        {
            // The AddJob method adds a new print job for an XPS
            // document into the print queue, and assigns a job name.
            // Use fastCopy to skip XPS validation and progress
            notifications.

            // If fastCopy is false, the thread that calls PrintQueue.AddJob
            // must have a single-threaded apartment state.
            PrintSystemJobInfo xpsPrintJob =
                defaultPrintQueue.AddJob(jobName: xpsFileName,
documentPath: xpsFilePath, fastCopy);

            // If the queue is not paused and the printer is working, then
            jobs will automatically begin printing.
            Debug.WriteLine($"Added {xpsFileName} to the print queue.");
        }
        catch (PrintJobException e)
        {
            allAdded = false;
            Debug.WriteLine($"Failed to add {xpsFileName} to the print
queue: {e.Message}\r\n{e.InnerException}");
        }
    }

    return allAdded;
}
```



还可以使用以下方法打印 XPS 文件：

- `PrintDialog.PrintDocument` 或 `PrintDialog.PrintVisual` 方法。
- `XpsDocumentWriter.Write` 和 `XpsDocumentWriter.WriteAsync` 方法。

有关详细信息，请参阅[如何显示打印对话框](#)和[打印文档概述](#)。

## 另请参阅

- [PrintQueue.AddJob](#)
- [WPF 中的文档](#)
- [如何显示打印对话框](#)
- [打印文档概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 数据绑定概述 (WPF .NET)

项目 · 2023/10/13

Windows Presentation Foundation (WPF) 中的数据绑定为应用呈现数据并与数据交互提供了一种简单而一致的方法。元素能够以 .NET 对象和 XML 的形式绑定到不同类型的数据源中的数据。所有 [ContentControl](#) (例如 [Button](#)) 以及所有 [ItemsControl](#) (例如 [ListBox](#) 和 [ListView](#)) 都具有内置功能，使单个数据项或数据项集合可以灵活地进行样式设置。可基于数据生成排序、筛选和分组视图。

WPF 中的数据绑定与传统模型相比具有几个优点，包括本质上支持数据绑定的大量属性、灵活的数据 UI 表示形式以及业务逻辑与 UI 的完全分离。

本文首先讨论 WPF 数据绑定的基本概念，然后介绍 [Binding](#) 类的用法和数据绑定的其他功能。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

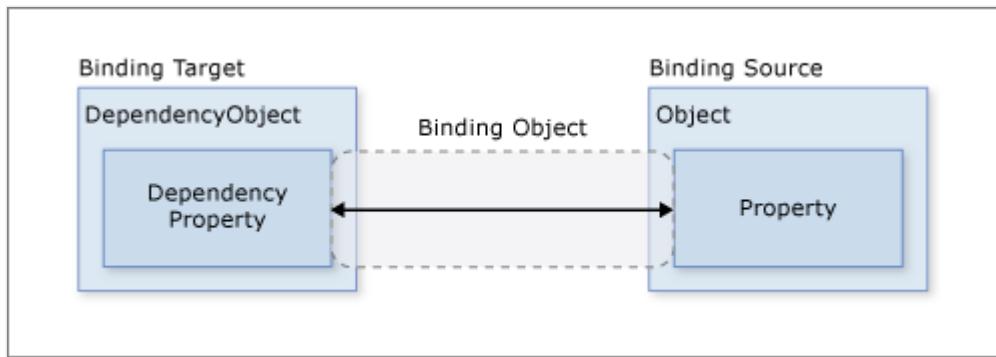
## 什么是数据绑定？

数据绑定是在应用 UI 与其显示的数据之间建立连接的过程。如果绑定具有正确的设置，并且数据提供适当的通知，则在数据更改其值时，绑定到该数据的元素会自动反映更改。数据绑定还意味着，如果元素中数据的外部表示形式发生更改，则基础数据可以自动进行更新以反映更改。例如，如果用户编辑 [TextBox](#) 元素中的值，则基础数据值会自动更新以反映该更改。

数据绑定的典型用法是将服务器或本地配置数据放置到窗体或其他 UI 控件中。此概念在 WPF 中得到扩展，包括将大量属性绑定到不同类型的数据源。在 WPF 中，元素的依赖属性可以绑定到 .NET 对象（包括 ADO.NET 对象或与 Web 服务和 Web 属性关联的对象）和 XML 数据。

## 数据绑定基本概念

不论要绑定什么元素，也不论数据源是什么性质，每个绑定都始终遵循下图所示的模型。



如图所示，数据绑定实质上是绑定目标与绑定源之间的桥梁。该图演示了以下基本的 WPF 数据绑定概念：

- 通常情况下，每个绑定具有四个组件：
  - 绑定目标对象。
  - 目标属性。
  - 绑定源。
  - 指向绑定源中要使用的值的路径。

例如，如果将 `TextBox` 的内容绑定到 `Employee.Name` 属性，则可以类似如下所示设置绑定：

设置	“值”
目标	<code>TextBox</code>
目标属性	<code>Text</code>
源对象	<code>Employee</code>
源对象值路径	<code>Name</code>

- 目标属性必须为依赖属性。

大多数 `UIElement` 属性都是依赖属性，而大多数依赖属性（只读属性除外）默认支持数据绑定。只有从 `DependencyObject` 派生的类型才能定义依赖项属性。所有 `UIElement` 类型从 `DependencyObject` 派生。

- 绑定源不限于自定义 .NET 对象。

尽管未在图中显示，但请注意，绑定源对象不限于自定义 .NET 对象。WPF 数据绑定支持 .NET 对象、XML 甚至是 XAML 元素对象形式的数据。例如，绑定源可以是 `UIElement`、任何列表对象、ADO.NET 或 Web 服务对象，或包含 XML 数据的 `XXmlNode`。有关详细信息，请参阅[绑定源概述](#)。

请务必记住，在建立绑定时，需要将绑定目标绑定到绑定源。例如，如果要使用数据绑定在 `ListBox` 中显示一些基础 XML 数据，则需要将 `ListBox` 绑定到 XML 数据。

若要建立绑定，请使用 [Binding](#) 对象。本文的其余部分讨论了与 [Binding](#) 对象相关的许多概念以及该对象的一些属性和用法。

## 数据上下文

当在 XAML 元素上声明数据绑定时，它们会通过查看其直接的 [DataContext](#) 属性来解析数据绑定。数据上下文通常是绑定源值路径评估的绑定源对象。可以在绑定中重写此行为，并设置特定的绑定源对象值。如果未设置承载绑定的对象的 [DataContext](#) 属性，则将检查父元素的 [DataContext](#) 属性，依此类推，直到 XAML 对象树的根。简而言之，除非在对象上显式设置，否则用于解析绑定的数据上下文将继承自父级。

绑定可以配置为使用特定的对象进行解析，而不是使用数据上下文进行绑定解析。例如，在将对象的前景色绑定到另一个对象的背景色时，将使用直接指定源对象。无需数据上下文，因为绑定在这两个对象之间解析。相反，未绑定到特定源对象的绑定使用数据上下文解析。

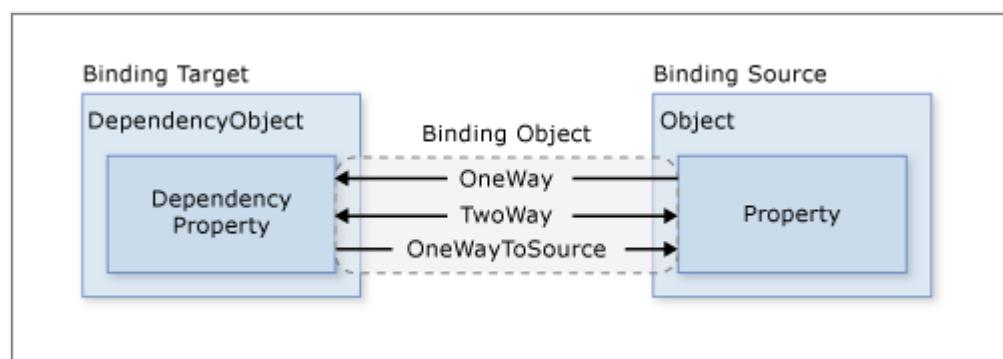
当 [DataContext](#) 属性发生更改时，重新评估可能会受数据上下文影响的所有绑定。

## 数据流的方向

正如上图中的箭头所示，绑定的数据流可以从绑定目标流向绑定源（例如，当用户编辑 [TextBox](#) 的值时，源值会发生更改）和/或（在绑定源提供正确通知的情况下）从绑定源流向绑定目标（例如，[TextBox](#) 内容会随绑定源中的更改而进行更新）。

你可能希望应用允许用户更改数据，然后将该数据传播回源对象。或者，可能不希望允许用户更新源数据。可以通过设置 [Binding.Mode](#) 来控制数据流。

此图演示了不同类型的数据流：



- 通过 [OneWay](#) 绑定，对源属性的更改会自动更新目标属性，但对目标属性的更改不会传播回源属性。如果绑定的控件为隐式只读，则此类类型的绑定适用。例如，可能会绑定到股票行情自动收录器这样的源，也可能目标属性没有用于进行更改的控件接口（例如表的数据绑定背景色）。如果无需监视目标属性的更改，则使用 [OneWay](#) 绑定模式可避免 [TwoWay](#) 绑定模式的系统开销。

- 通过 [TwoWay 绑定](#)，更改源属性或目标属性时会自动更新另一方。此类型的绑定适用于可编辑窗体或其他完全交互式 UI 方案。大多数属性默认为 [OneWay 绑定](#)，但某些依赖属性（通常为用户可编辑控件的属性，例如 [TextBox.Text](#) 和 [CheckBox.IsChecked](#)）默认为 [TwoWay 绑定](#)。

用于确定依赖项属性绑定在默认情况下是单向还是双向的编程方法是：使用 [DependencyProperty.GetMetadata](#) 获取属性元数据。此方法的返回类型为 [PropertyMetadata](#)，它不包含任何有关绑定的元数据。但是，如果可以将此类型强制转换为派生的 [FrameworkPropertyMetadata](#)，则可以检查 [FrameworkPropertyMetadata.BindsTwoWayByDefault](#) 属性的布尔值。以下代码示例演示了如何获取 [TextBox.Text](#) 属性的元数据：

C#

```

public static void PrintMetadata()
{
    // Get the metadata for the property
    PropertyMetadata metadata =
        TextBox.TextProperty.GetMetadata(typeof(TextBox));

    // Check if metadata type is FrameworkPropertyMetadata
    if (metadata is FrameworkPropertyMetadata frameworkMetadata)
    {
        System.Diagnostics.Debug.WriteLine($"TextBox.Text property
metadata:");
        System.Diagnostics.Debug.WriteLine($"  BindsTwoWayByDefault:
{frameworkMetadata.BindsTwoWayByDefault}");
        System.Diagnostics.Debug.WriteLine($"  IsDataBindingAllowed:
{frameworkMetadata.IsDataBindingAllowed}");
        System.Diagnostics.Debug.WriteLine($"      AffectsArrange:
{frameworkMetadata.AffectsArrange}");
        System.Diagnostics.Debug.WriteLine($"      AffectsMeasure:
{frameworkMetadata.AffectsMeasure}");
        System.Diagnostics.Debug.WriteLine($"      AffectsRender:
{frameworkMetadata.AffectsRender}");
        System.Diagnostics.Debug.WriteLine($"      Inherits:
{frameworkMetadata.Inherits}");
    }

    /* Displays:
     *
     * TextBox.Text property metadata:
     *   BindsTwoWayByDefault: True
     *   IsDataBindingAllowed: True
     *   AffectsArrange: False
     *   AffectsMeasure: False
     *   AffectsRender: False
     *   Inherits: False
     */
}

```

- [OneWayToSource](#) 绑定与 [OneWay](#) 绑定相反；当目标属性更改时，它会更新源属性。一个示例方案是只需要从 UI 重新计算源值的情况。
- [OneTime](#) 绑定未在图中显示，该绑定会使源属性初始化目标属性，但不传播后续更改。如果数据上下文发生更改，或者数据上下文中的对象发生更改，则更改不会在目标属性中反映。如果适合使用当前状态的快照或数据实际为静态数据，则此类型的绑定适合。如果你想使用源属性中的某个值来初始化目标属性，且提前不知道数据上下文，则此类型的绑定也有用。此模式实质上是 [OneWay](#) 绑定的一种简化形式，它在源值不更改的情况下提供更好的性能。

若要检测源更改（适用于 [OneWay](#) 和 [TwoWay](#) 绑定），则源必须实现合适的属性更改通知机制，例如 [INotifyPropertyChanged](#)。请参阅[如何：实现属性更改通知 \(.NET Framework\)](#)，获取 [INotifyPropertyChanged](#) 实现的示例。

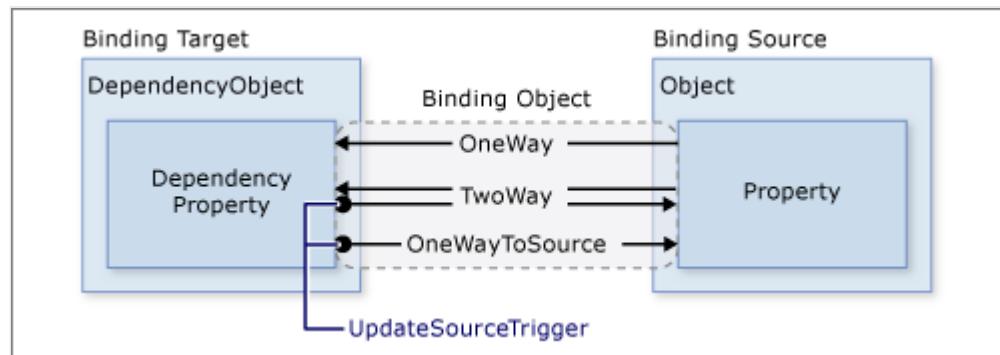
[Binding.Mode](#) 属性提供有关绑定模式的详细信息，以及如何指定绑定方向的示例。

## 触发源更新的因素

[TwoWay](#) 或 [OneWayToSource](#) 绑定侦听目标属性中的更改，并将更改传播回源（称为更新源）。例如，可以编辑文本框的文本以更改基础源值。

但是，在编辑文本时或完成文本编辑后控件失去焦点时，源值是否会更新？

[Binding.UpdateSourceTrigger](#) 属性确定触发源更新的因素。下图中右箭头的点说明了 [Binding.UpdateSourceTrigger](#) 属性的角色。



如果 [UpdateSourceTrigger](#) 值为 [UpdateSourceTrigger.PropertyChanged](#)，则目标属性更改后，[TwoWay](#) 或 [OneWayToSource](#) 绑定的右箭头指向的值会立即更新。但是，如果 [UpdateSourceTrigger](#) 值为 [LostFocus](#)，则仅当目标属性失去焦点时才会使用新值更新该值。

与 [Mode](#) 属性类似，不同的依赖属性具有不同的默认 [UpdateSourceTrigger](#) 值。大多数依赖属性的默认值为 [PropertyChanged](#)，这将导致源属性的值在目标属性值更改时立即更改。即时更改适用于 [CheckBox](#) 和其他简单控件。但对于文本字段，每次击键后都进行更新会降低性能，用户也没有机会在提交新值之前使用 Backspace 键修改键入错误。例如，[TextBox.Text](#) 属性默认为 [LostFocus](#) 的 [UpdateSourceTrigger](#) 值，这会导致源值仅

在控件元素失去焦点时（而不是在 `TextBox.Text` 属性更改时）更改。有关如何查找依赖属性的默认值的信息，请参阅 [UpdateSourceTrigger 属性页](#)。

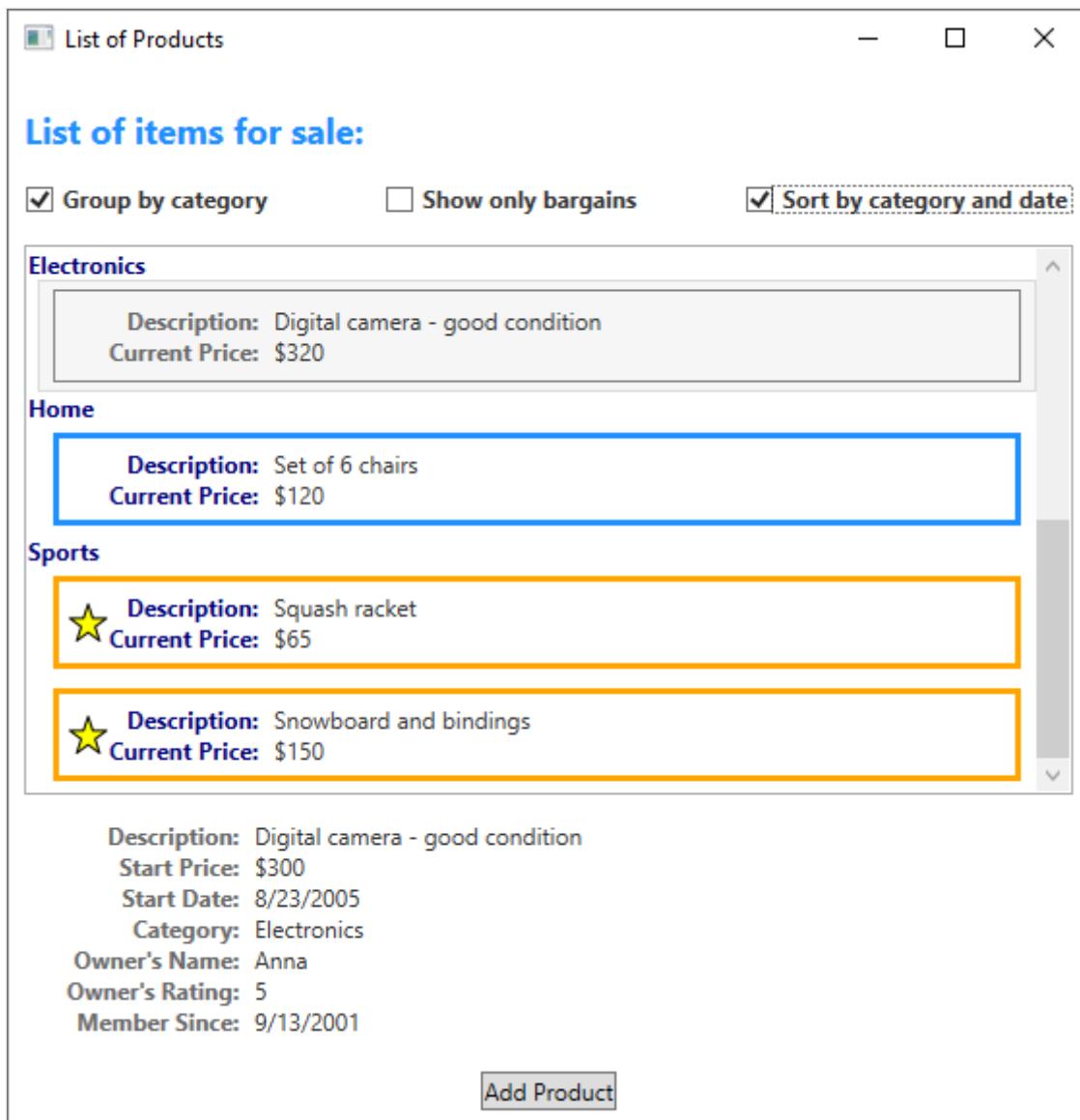
下表以 `TextBox` 为例，提供每个 `UpdateSourceTrigger` 值的示例方案。

<code>UpdateSourceTrigger</code> 值	源值更新时间	<code>TextBox</code> 的示例方案
<code>LostFocus</code> ( <code>TextBox.Text</code> 的默认值)	<code>TextBox</code> 控件失去焦点时。	与验证逻辑关联的 <code>TextBox</code> （请参阅下文的 <a href="#">数据验证</a> ）。
<code>PropertyChanged</code>	键入 <code>TextBox</code> 时。	聊天室窗口中的 <code>TextBox</code> 控件。
<code>Explicit</code>	应用调用 <code>UpdateSource</code> 时。	可编辑窗体中的 <code>TextBox</code> 控件（仅当用户按“提交”按钮时才更新源值）。

有关示例，请参阅[如何：控制 TextBox 文本更新源的时间 \(.NET Framework\)](#)。

## 数据绑定的示例

有关数据绑定的示例，请参阅[数据绑定演示](#)（显示拍卖项的列表）中的以下应用 UI。



应用演示了数据绑定的以下功能：

- ListBox 的内容已绑定到 *AuctionItem* 对象的集合。 *AuctionItem* 对象具有 *Description*、*StartPrice*、*StartDate*、*Category* 和 *SpecialFeatures* 等属性。
- ListBox 中显示的数据 (*AuctionItem* 对象) 已进行模板化，以便显示每个项的说明和当前价格。通过使用 [DataTemplate](#) 来创建模板。此外，每个项的外观取决于要显示的 *AuctionItem* 的 *SpecialFeatures* 值。如果 *AuctionItem* 的 *SpecialFeatures* 值为 *Color*，则该项具有蓝色边框。如果值为 *Highlight*，则该项具有橙色边框和一个星号。[数据模板化](#)部分提供了数据模板化的相关信息。
- 用户可以使用提供的 [Checkboxes](#) 对数据进行分组、筛选或排序。在上图中，选中了“按类别分组”和“按类别和日期排序”[Checkboxes](#)。你可能已注意到，数据按产品类别分组，类别名称按字母顺序排序。这些项还按每个类别中的开始日期排序，但难以从图中注意到这一点。排序使用[集合视图](#)实现。[绑定到集合](#)部分讨论了集合视图。

- 当用户选择某个项时，ContentControl 显示所选项的详细信息。此体验称为主-从方案。主-从方案部分提供有关此绑定类型的信息。
- StartDate* 属性的类型为 [DateTime](#)，该类型返回一个包括精确到毫秒的时间的日期。在此应用中，使用了一个自定义转换器，以便显示较短的日期字符串。[数据转换](#)部分提供有关转换器的信息。

当用户选择“添加产品”按钮时，会出现以下窗体。

**Item for sale:**

Item Description: Brand New Computer

Start Price: 650

Start Date: 9/28/2019

Category: Computers

Special Features: Highlight

Submit

**Description:** Brand New Computer  
**Current Price:** \$650

Description: Brand New Computer  
Start Price: \$650  
Start Date: 9/28/2019  
Category: Computers  
Owner's Name: John  
Owner's Rating: 12  
Member Since: 4/20/2003

用户可以编辑窗体中的字段，使用简略或详细预览窗格预览产品清单，然后选择 [Submit](#) 以添加新的产品清单。任何现有的分组、筛选和排序设置都将应用于新条目。在这种特殊情况下，上图中输入的项会作为 *Computer* 类别中的第二项显示。

“开始日期”[TextBox](#) 中提供的验证逻辑未在此图中显示。如果用户输入一个无效日期（格式无效或日期已过），则会通过 [ToolTip](#) 和 [TextBox](#) 旁边显示的红色感叹号来通知用户。[数据验证](#)一节讨论了如何创建验证逻辑。

在详细介绍数据绑定的上述不同功能之前，我们会先讨论对理解 WPF 数据绑定非常重要的基本概念。

## 创建绑定

前面部分中讨论的一些概念可以重申为：使用 [Binding](#) 对象建立绑定，且每个绑定通常具有四个组件：绑定目标、目标属性、绑定源以及指向要使用的源值的路径。本节讨论如何设置绑定。

绑定源绑定到元素的活动 [DataContext](#)。如果元素没有显式定义 [DataContext](#)，则会自动继承。

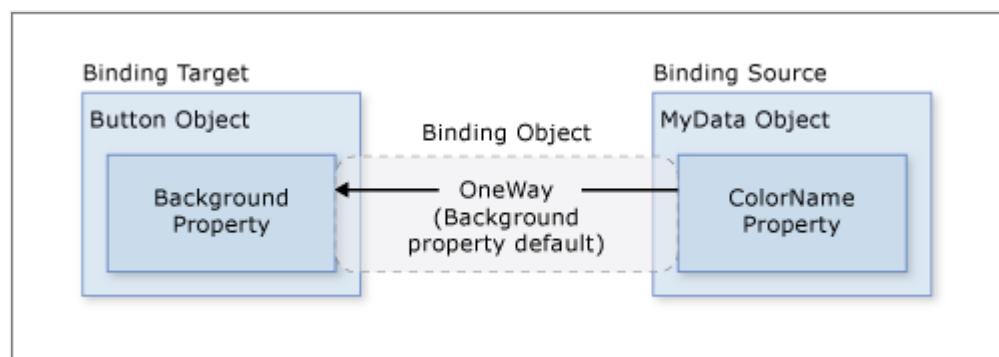
请考虑以下示例，其中的绑定源对象是一个名为 *MyData* 的类，该类在 *SDKSample* 命名空间中定义。出于演示目的，*MyData* 具有名为 *ColorName* 的字符串属性，其值设置为“Red”。因此，此示例生成一个具有红色背景的按钮。

XAML

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
    <DockPanel.DataContext>
        <Binding Source="{StaticResource myDataSource}"/>
    </DockPanel.DataContext>
    <Button Background="{Binding Path=ColorName}"
            Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>
```

有关绑定声明语法的详细信息以及如何在代码中设置绑定的示例，请参阅[绑定声明概述](#)。

如果将此示例应用于基本关系图，则生成的图如下所示。此图描述 [OneWay](#) 绑定，因为 *Background* 属性默认支持 [OneWay](#) 绑定。



你可能会想知道，此绑定为何在 *ColorName* 属性的类型为字符串而 *Background* 属性的类型为 [Brush](#) 的情况下也会起作用。此绑定使用默认类型转换，这会在[数据转换](#)部分中进行讨论。

## 指定绑定源

请注意，在前面的示例中，通过设置 `DockPanel.DataContext` 属性指定绑定源。然后，`Button` 从其父元素 `DockPanel` 继承 `DataContext` 值。重申一下，绑定源对象是绑定的四个必需组件之一。所以，如果未指定绑定源对象，则绑定将没有任何作用。

可通过多种方法指定绑定源对象。将多个属性绑定到同一个源时，可以使用父元素上的 `DataContext` 属性。不过，有时在个别绑定声明中指定绑定源可能更为合适。对于前面的示例，不使用 `DataContext` 属性，而是通过在按钮的绑定声明中直接设置 `Binding.Source` 属性来指定绑定源，如以下示例所示。

### XAML

```
<DockPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
            xmlns:c="clr-namespace:SDKSample">
    <DockPanel.Resources>
        <c:MyData x:Key="myDataSource"/>
    </DockPanel.Resources>
    <Button Background="{Binding Source={StaticResource myDataSource},
Path=ColorName}"
           Width="150" Height="30">
        I am bound to be RED!
    </Button>
</DockPanel>
```

除直接在元素中设置 `DataContext` 属性、从上级元素（例如第一个示例中的按钮）继承 `DataContext` 值以及通过在绑定上设置 `Binding.Source` 属性（例如最后一个示例中的按钮）来显式指定绑定源外，你还可以使用 `Binding.ElementName` 属性或 `Binding.RelativeSource` 属性指定绑定源。当绑定到应用中的其他元素时（例如，使用滑块调整按钮的宽度时），`ElementName` 属性非常有用。在 `ControlTemplate` 或 `Style` 中指定绑定时，可以使用 `RelativeSource` 属性。有关详细信息，请参阅[绑定源概述](#)。

## 指定指向值的路径

如果绑定源是一个对象，则使用 `Binding.Path` 属性指定要用于绑定的值。如果要绑定到 XML 数据，则使用 `Binding.XPath` 属性指定值。在某些情况下，使用 `Path` 属性（即使数据为 XML）可能更为合适。例如，如果要访问返回的 `XmlNode`（作为 XPath 查询的结果）的 `Name` 属性，则除 `XPath` 属性外，还应使用 `Path` 属性。

有关详细信息，请参阅 `Path` 和 `XPath` 属性。

虽然我们已强调要使用的值的 `Path` 是绑定的四个必需组件之一，但在要绑定到整个对象的方案中，要使用的值会与绑定源对象相同。在这些情况下，可以不指定 `Path`。请看下面的示例。

## XAML

```
<ListBox ItemsSource="{Binding}"  
        IsSynchronizedWithCurrentItem="true"/>
```

以上示例使用空绑定语法：{Binding}。在此示例中，`ListBox` 从父 `DockPanel` 元素继承 `DataContext`（此示例中未显示）。未指定路径时，默认为绑定到整个对象。换句话说，此示例中的路径已省略，因为要将 `ItemsSource` 属性绑定到整个对象。（有关深入讨论，请参阅[绑定到集合部分](#)。）

除了绑定到集合以外，在希望绑定到整个对象，而不是仅绑定到对象的单个属性时，也可以使用此方案。例如，如果源对象的类型为 `String`，则可能仅希望绑定到字符串本身。另一种常见情况是希望将一个元素绑定到一个具有多个属性的对象。

你可能需要应用自定义逻辑，以便数据对于绑定的目标属性有意义。如果不存在默认类型转换，则自定义逻辑可能采用自定义转换器的形式。有关转换器的信息，请参阅[数据转换](#)。

## Binding 和 BindingExpression

在介绍数据绑定的其他功能和用法前，先介绍一下 `BindingExpression` 类会很有用。如前面部分所述，`Binding` 类是用于绑定声明的高级类；该类提供许多供用户指定绑定特征的属性。相关类 `BindingExpression` 是维持源与目标之间连接的基础对象。一个绑定包含了可以在多个绑定表达式之间共享的所有信息。`BindingExpression` 是无法共享的实例表达式，并包含 `Binding` 的所有实例信息。

举例来说，假设 `myDataObject` 是 `MyData` 类的实例，`myBinding` 是源 `Binding` 对象，而 `MyData` 是包含名为 `ColorName` 的字符串属性的定义类。此示例将 `TextBlock` 的实例 `myText` 的文本内容绑定到 `ColorName`。

## C#

```
// Make a new source  
var myDataObject = new MyData();  
var myBinding = new Binding("ColorName")  
{  
    Source = myDataObject  
};  
  
// Bind the data source to the TextBox control's Text dependency property  
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

可以使用同一 `myBinding` 对象来创建其他绑定。例如，可使用 `myBinding` 对象将复选框的文本内容绑定到 `ColorName`。在该方案中，将有两个 `BindingExpression` 实例共享

`myBinding` 对象。

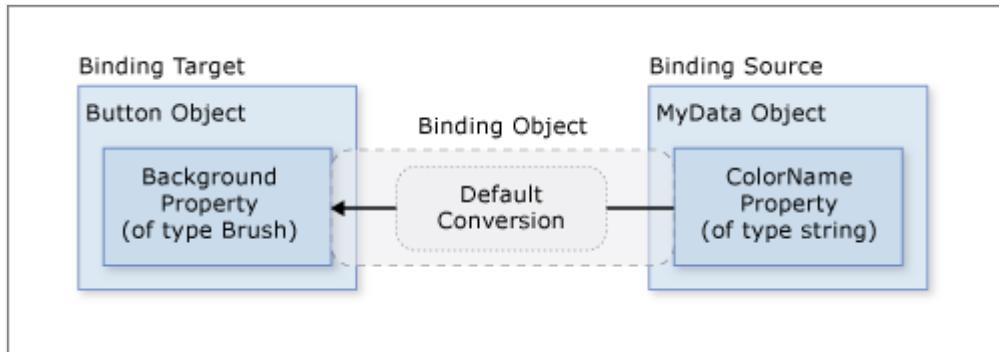
通过对数据绑定对象调用 `GetBindingExpression` 来返回 `BindingExpression` 对象。以下文章演示了 `BindingExpression` 类的一些用法：

- 从绑定目标属性获取绑定对象 (.NET Framework)
- 控制 `TextBox` 文本更新源的时间 (.NET Framework)

## 数据转换

在创建绑定部分，该按钮为红色，因为其 `Background` 属性绑定到值为“Red”的字符串属性。此字符串值有效是因为 `Brush` 类型中存在类型转换器，可用于将字符串值转换为 `Brush`。

将此信息添加到创建绑定部分的图中的情况如下所示。



但是，如果绑定源对象拥有的不是字符串类型的属性，而是 `Color` 类型的 `Color` 属性，该怎么办？在这种情况下，为了使绑定正常工作，首先需要将 `Color` 属性值转换为 `Background` 属性可接受的值。需要通过实现 `IValueConverter` 接口来创建一个自定义转换器，如以下示例所示。

C#

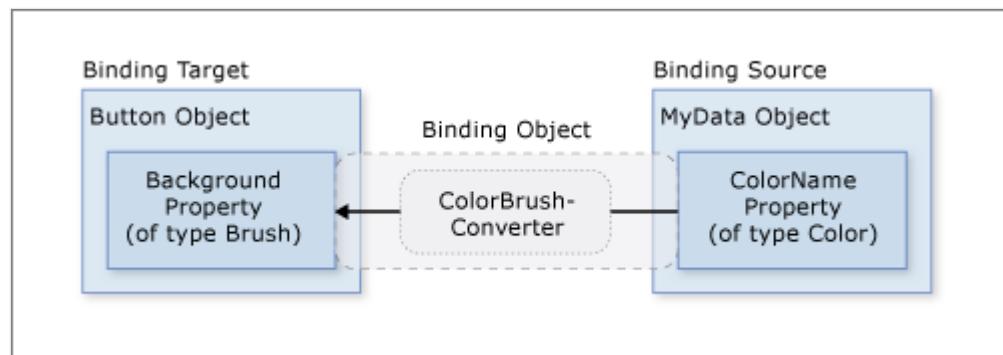
```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]
public class ColorBrushConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter,
System.Globalization.CultureInfo culture)
    {
        Color color = (Color)value;
        return new SolidColorBrush(color);
    }

    public object ConvertBack(object value, Type targetType, object
parameter, System.Globalization.CultureInfo culture)
    {
        return null;
    }
}
```

```
    }  
}
```

有关详细信息，请参阅[IValueConverter](#)。

现在，使用的是自定义转换器而不是默认转换，关系图如下所示。



重申一下，由于要绑定到的类型中提供了类型转换器，因此可以使用默认转换。此行为取决于目标中可用的类型转换器。如果无法确定，请创建自己的转换器。

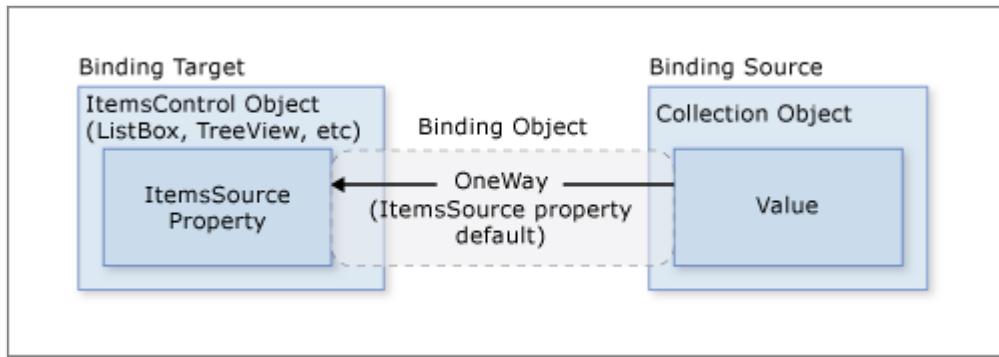
下面提供了一些典型方案，在这些方案中，实现数据转换器非常有意义：

- 数据应根据区域性以不同方式显示。例如，可能需要根据在特定区域性中使用的约定，实现货币转换器或日历日期/时间转换器。
- 使用的数据不一定会更改属性的文本值，但会更改其他某个值（如图像的源，或显示文本的颜色或样式）。在这种情况下，可以通过转换可能不合适的属性绑定（如将文本字段绑定到表单元格的 Background 属性）来使用转换器。
- 多个控件或控件的多个属性会绑定到相同数据。在这种情况下，主绑定可能仅显示文本，而其他绑定则处理特定的显示问题，但仍使用同一绑定作为源信息。
- 目标属性具有绑定集合，称为 [MultiBinding](#)。对于 [MultiBinding](#)，使用自定义 [IMultiValueConverter](#) 从绑定的值中生成最终值。例如，可以从红色、蓝色和绿色的值来计算颜色，这些值可能来自相同绑定源对象，也可能来自不同绑定源对象。有关示例和信息，请参阅 [MultiBinding](#)。

## 绑定到集合

绑定源对象可以被视为其属性包含数据的单个对象，也可以被视为通常组合在一起的多态对象的数据集合（例如数据库查询的结果）。目前为止，我们仅讨论了绑定到单个对象，但绑定到数据集合也是常见方案。例如，一种常见方案是使用 [ItemsControl](#)（例如 [ListBox](#)、[ListView](#) 或 [TreeView](#)）来显示数据集合，如在[什么是数据绑定](#)部分所示的应用中。

幸运的是，基本关系图仍然适用。如果将 [ItemsControl](#) 绑定到集合，则关系图如下所示。



如图所示，若要将 [ItemsControl](#) 绑定到集合对象，则需要使用 [ItemsControl.ItemsSource](#) 属性。你可以将 [ItemsSource](#) 视为 [ItemsControl](#) 的内容。绑定为 [OneWay](#)，因为 [ItemsSource](#) 属性默认支持 [OneWay](#) 绑定。

## 如何实现集合

你可以枚举实现 [IEnumerable](#) 接口的任何集合。但是，若要设置动态绑定，以便集合中的插入或删除操作可以自动更新 UI，则集合必须实现 [INotifyCollectionChanged](#) 接口。此接口公开一个事件，只要基础集合发生更改，就应该引发该事件。

WPF 提供 [ObservableCollection<T>](#) 类，该类是公开 [INotifyCollectionChanged](#) 接口的数据集合的内置实现。若要完全支持将数据值从源对象传输到目标，支持可绑定属性的集合中的每个对象还必须实现 [INotifyPropertyChanged](#) 接口。有关详细信息，请参阅[绑定源概述](#)。

在实现自己的集合前，请考虑使用 [ObservableCollection<T>](#) 或现有集合类之一，例如 [List<T>](#)、[Collection<T>](#) 和 [BindingList<T>](#) 等。如果有高级方案并且希望实现自己的集合，请考虑使用 [IList](#)，它提供可以按索引逐个访问的对象的非泛型集合，因而可提供最佳性能。

## 集合视图

在 [ItemsControl](#) 绑定到数据集合后，你可能希望对数据进行排序、筛选或分组。为此，应使用集合视图，这些视图是实现 [ICollectionView](#) 接口的类。

### 什么是集合视图？

集合视图这一层基于绑定源集合，它允许基于排序、筛选和分组查询来导航并显示源集合，而无需更改基础源集合本身。集合视图还维护一个指向集合中当前项的指针。如果

源集合实现 [INotifyCollectionChanged](#) 接口，则 [CollectionChanged](#) 事件引发的更改会传播到视图。

由于视图不会更改基础源集合，因此每个源集合都可以有多个关联的视图。例如，可以有 [Task](#) 对象的集合。使用视图，可以通过不同方式显示相同数据。例如，可能希望在页面左侧显示按优先级排序的任务，而在页面右侧显示按区域分组的任务。

## 视图创建方法

创建并使用视图的一种方式是直接实例化视图对象，然后将它用作绑定源。以[什么是数据绑定部分](#)中所示的[数据绑定演示](#)应用为例。该应用的实现方式是将 [ListBox](#) 绑定到基于数据集合的视图，而不是直接绑定到数据集合。下面的示例摘自[数据绑定演示](#)应用。[CollectionViewSource](#) 类是从 [CollectionView](#) 继承的类的 XAML 代理。在此特定示例中，视图的 [Source](#) 绑定到当前应用对象的 [AuctionItem](#) 集合（类型为 [ObservableCollection](#)<T>）。

XAML

```
<Window.Resources>
    <CollectionViewSource
        Source="{Binding Source={x:Static Application.Current},
        Path=AuctionItems}"
        x:Key="listing DataView" />
</Window.Resources>
```

资源 *listing DataView* 随后用作应用中元素（例如 [ListBox](#)）的绑定源。

XAML

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
    ItemsSource="{Binding Source={StaticResource listing DataView}}" />
```

若要为同一集合创建另一个视图，则可以创建另一个 [CollectionViewSource](#) 实例，并为其提供不同的 [x:Key](#) 名称。

下表显示作为默认集合视图创建或由 [CollectionViewSource](#) 根据源集合类型创建的视图数据类型。

源集合类型	集合视图类型	说明
<a href="#">IEnumerable</a>	基于 <a href="#">CollectionView</a> 的内部类型	无法对项进行分组。
<a href="#">IList</a>	<a href="#">ListCollectionView</a>	最快。
<a href="#">IBindingList</a>	<a href="#">BindingListCollectionView</a>	

## 使用默认视图

创建并使用集合视图的一种方式是指定集合视图作为绑定源。 WPF 还会为用作绑定源的每个集合创建一个默认集合视图。 如果直接绑定到集合， WPF 会绑定到该集合的默认视图。 此默认视图由同一集合的所有绑定共享，因此一个绑定控件或代码对默认视图所做的更改（例如排序或对当前项指针的更改，下文将对此进行讨论）会在同一集合的所有其他绑定中反映。

若要获取默认视图，请使用 [GetDefaultView](#) 方法。 有关示例，请参阅[获取数据集合的默认视图 \(.NET Framework\)](#)。

## 包含 ADO.NET DataTables 的集合视图

为了提高性能，ADO.NET [DataTable](#) 或 [DataView](#) 对象的集合视图将排序和筛选委托给 [DataView](#)，这导致排序和筛选在数据源的所有集合视图之间共享。 若要使每个集合视图都能独立进行排序和筛选，请使用每个集合视图自己的 [DataView](#) 对象进行初始化。

## 排序

如前所述，视图可以将排序顺序应用于集合。 如同在基础集合中一样，数据可能具有或不具有相关的固有顺序。 借助集合视图，可以根据自己提供的比较条件来强制确定顺序，或更改默认顺序。 由于这是基于客户端的数据视图，因此一种常见情况是用户可能希望根据列对应的值，对多列表格数据进行排序。 通过使用视图，可以应用这种用户实施的排序，而无需对基础集合进行任何更改，甚至不必再次查询集合内容。 有关示例，请参阅[在单击标题时对 GridView 列进行排序 \(.NET Framework\)](#)。

以下示例演示了[什么是数据绑定](#)部分中的应用 UI 的“按类别和日期排序”[CheckBox](#) 的排序逻辑。

C#

```
private void AddSortCheckBox_CheckedChanged(object sender, RoutedEventArgs e)
{
    // Sort the items first by Category and then by StartDate
    listingDataView.SortDescriptions.Add(new SortDescription("Category",
ListSortDirection.Ascending));
    listingDataView.SortDescriptions.Add(new SortDescription("StartDate",
ListSortDirection.Ascending));
}
```

## 筛选

视图还可以将筛选器应用于集合，以便视图仅显示完整集合的特定子集。可以根据条件在数据中进行筛选。例如，正如[什么是数据绑定](#)部分中的应用所做的那样，“仅显示成交商品”[CheckBox](#) 包含了筛选出成交价等于或大于 25 美元的项的逻辑。如果选择了 [CheckBox](#)，则会执行以下代码将 [ShowOnlyBargainsFilter](#) 设置为 [Filter](#) 事件处理程序。

C#

```
private void AddFilteringCheckBox_Cheked(object sender, RoutedEventArgs e)
{
    if (((CheckBox)sender).IsChecked == true)
        listingDataView.Filter += ListingDataView_Filter;
    else
        listingDataView.Filter -= ListingDataView_Filter;
}
```

[ShowOnlyBargainsFilter](#) 事件处理程序具有以下实现。

C#

```
private void ListingDataView_Filter(object sender, FilterEventArgs e)
{
    // Start with everything excluded
    e.Accepted = false;

    // Only include items with a price less than 25
    if (e.Item is AuctionItem product && product.CurrentPrice < 25)
        e.Accepted = true;
}
```

如果直接使用其中一个 [ICollectionView](#) 类而不是 [CollectionViewSource](#)，则可以使用 [Filter](#) 属性指定回叫。有关示例，请参阅[筛选视图中的数据 \(.NET Framework\)](#)。

## 分组

除了用来查看 [IEnumerable](#) 集合的内部类之外，所有集合视图都支持分组功能，用户可以利用此功能将集合视图中的集合划分成逻辑组。这些组可以是显式的，由用户提供组列表；也可以是隐式的，这些组依据数据动态生成。

以下示例演示了“按类别分组”[CheckBox](#) 的逻辑。

C#

```
// This groups the items in the view by the property "Category"
var groupDescription = new PropertyGroupDescription();
groupDescription.PropertyName = "Category";
listingDataView.GroupDescriptions.Add(groupDescription);
```

有关其他分组示例，请参阅对实现 GridView 的 ListView 中的项进行分组 (.NET Framework)。

## 当前项指针

视图还支持当前项的概念。可以在集合视图中的对象之间导航。在导航时，你是在移动项指针，该指针可用于检索存在于集合中特定位置的对象。有关示例，请参阅[在CollectionView 中的对象之间导航 \(.NET Framework\)](#)。

由于 WPF 只通过使用视图（你指定的视图或集合的默认视图）绑定到集合，因此集合的所有绑定都有一个当前项指针。绑定到视图时，`Path` 值中的斜杠（“/”）字符用于指定视图的当前项。在下面的示例中，数据上下文是一个集合视图。第一行绑定到集合。第二行绑定到集合中的当前项。第三行绑定到集合中当前项的 `Description` 属性。

XAML

```
<Button Content="{Binding }" />
<Button Content="{Binding Path=/}" />
<Button Content="{Binding Path=/Description}" />
```

还可以连着使用斜杠和属性语法以遍历集合的分层。以下示例绑定到一个名为 `Offices` 的集合的当前项，此集合是源集合的当前项的属性。

XAML

```
<Button Content="{Binding /Offices/}" />
```

当前项指针可能会受对集合应用的任何排序或筛选操作的影响。排序操作将当前项指针保留在所选的最后一项上，但集合视图现已围绕此指针重构。（或许所选项以前曾位于列表的开头，但现在所选项可能位于中间的某个位置。）如果所选内容在筛选之后保留在视图中，则筛选操作会保留所选项。否则，当前项指针会设置为经过筛选的集合视图的第一项。

## 主-从绑定方案

当前项的概念不仅适用于集合中各项的导航，也适用于主-从绑定方案。再考虑一下[什么是数据绑定部分中的应用 UI](#)。在该应用中，`ListBox` 中的选择确定 `ContentControl` 中显示的内容。换句话说，选择 `ListBox` 项目时，`ContentControl` 显示所选项的详细信息。

只需将两个或更多控件绑定到同一视图即可实现主-从方案。[数据绑定演示](#) 中的以下示例演示了在[什么是数据绑定部分中的应用 UI](#) 上看到的 `ListBox` 和 `ContentControl` 的标记。

## XAML

```
<ListBox Name="Master" Grid.Row="2" Grid.ColumnSpan="3" Margin="8"
         ItemsSource="{Binding Source={StaticResource listing DataView}}"/>
<ContentControl Name="Detail" Grid.Row="3" Grid.ColumnSpan="3"
                  Content="{Binding Source={StaticResource listing DataView}}"
                  ContentTemplate="{StaticResource
detailsProductListingTemplate}"
                  Margin="9,0,0,0"/>
```

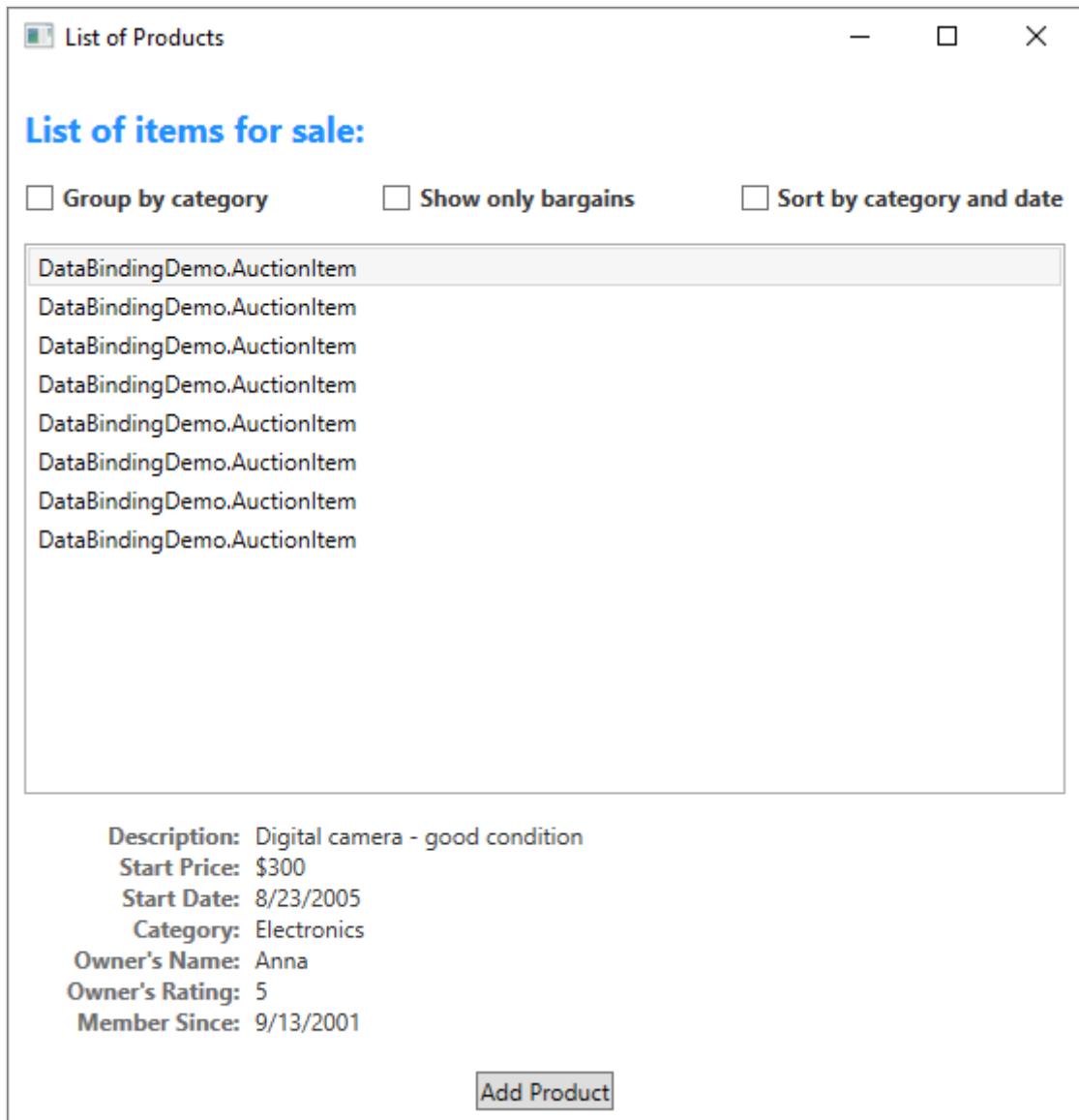
请注意，这两个控件都绑定到同一个源，即 *listing DataView* 静态资源（请参阅[如何创建视图部分](#)中关于此资源的定义）。此绑定有效是因为将单一实例对象（在本例中为 [ContentControl](#)）绑定到集合视图时，它会自动绑定到该视图的 [CurrentItem](#)。[CollectionViewSource](#) 对象会自动同步货币和选择。如果列表控件未像本示例中那样绑定到 [CollectionViewSource](#) 对象，则需要将其 [IsSynchronizedWithCurrentItem](#) 属性设置为 `true` 才能起作用。

有关其他示例，请参阅[绑定到集合并基于选择显示信息 \(.NET Framework\)](#)和[对层次结构数据使用主-从模式 \(.NET Framework\)](#)。

你可能已经注意到上述示例使用了一个模板。实际上，如果不使用模板（[ContentControl](#) 显式使用的模板以及 [ListBox](#) 隐式使用的模板），数据不会按照我们希望的方式显示。现在，我们开始介绍下一节中的数据模板化。

## 数据模板化

如果不使用数据模板，[数据绑定示例](#)部分中的应用 UI 将如下所示：



如前面部分中的示例所示，[ListBox](#) 控件和 [ContentControl](#) 都绑定到 [AuctionItem](#) 的整个集合对象（更具体地说，是绑定到集合对象视图）。如果未提供如何显示数据集合的特定说明，则 [ListBox](#) 会以字符串形式显示基础集合中的每个对象，[ContentControl](#) 会以字符串形式显示绑定到的对象。

为了解决该问题，应用定义了 [DataTemplates](#)。如前面部分中的示例所示，[ContentControl](#) 显式使用 *detailsProductListingTemplate* 数据模板。显示集合中的 [AuctionItem](#) 对象时，[ListBox](#) 控件隐式使用以下数据模板。

XAML

```
<DataTemplate DataType="{x:Type src:AuctionItem}">
    <Border BorderThickness="1" BorderBrush="Gray"
        Padding="7" Name="border" Margin="3" Width="500">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
                <RowDefinition/>
            </Grid>
```

```

        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="20"/>
            <ColumnDefinition Width="86"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
            Fill="Yellow" Stroke="Black" StrokeThickness="1"
            StrokeLineJoin="Round" Width="20" Height="20"
            Stretch="Fill"
            Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7
7,7"
            Visibility="Hidden" Name="star"/>

        <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
            Name="descriptionTitle"
            Style="{StaticResource smallTitleStyle}">Description:
    </TextBlock>

        <TextBlock Name="DescriptionDTDataType" Grid.Row="0"
Grid.Column="2"
            Text="{Binding Path=Description}"
            Style="{StaticResource textStyleTextBlock}"/>

        <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
            Name="currentPriceTitle"
            Style="{StaticResource smallTitleStyle}">Current
Price:</TextBlock>

        <StackPanel Grid.Row="1" Grid.Column="2"
Orientation="Horizontal">
            <TextBlock Text="$" Style="{StaticResource
textStyleTextBlock}"/>
            <TextBlock Name="CurrentPriceDTDataType"
                Text="{Binding Path=CurrentPrice}"
                Style="{StaticResource textStyleTextBlock}"/>
        </StackPanel>
    </Grid>
</Border>
<DataTemplate.Triggers>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Color</src:SpecialFeatures>
        </DataTrigger.Value>
        <DataTrigger.Setters>
            <Setter Property="BorderBrush" Value="DodgerBlue"
TargetName="border" />
            <Setter Property="Foreground" Value="Navy"
TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy"
TargetName="currentPriceTitle" />
            <Setter Property="BorderThickness" Value="3"
TargetName="border" />
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger.Setters>
    </DataTrigger>

```

```

        </DataTrigger.Setters>
    </DataTrigger>
    <DataTrigger Binding="{Binding Path=SpecialFeatures}">
        <DataTrigger.Value>
            <src:SpecialFeatures>Highlight</src:SpecialFeatures>
        </DataTrigger.Value>
        <Setter Property="BorderBrush" Value="Orange"
TargetName="border" />
            <Setter Property="Foreground" Value="Navy"
TargetName="descriptionTitle" />
            <Setter Property="Foreground" Value="Navy"
TargetName="currentPriceTitle" />
            <Setter Property="Visibility" Value="Visible" TargetName="star"
/>
            <Setter Property="BorderThickness" Value="3" TargetName="border"
/>
            <Setter Property="Padding" Value="5" TargetName="border" />
        </DataTrigger>
    </DataTemplate.Triggers>
</DataTemplate>

```

使用这两个 DataTemplate 时，生成的 UI 即为为什么是数据绑定部分中所示的 UI。如屏幕截图所示，除了可以在控件中放置数据以外，使用 DataTemplate 还可以为数据定义引人注目的视觉对象。例如，上述 DataTemplate 中使用了 DataTrigger，因而 SpecialFeatures 值为 *HighLight* 的 AuctionItem 会显示为带有橙色边框和一个星号。

有关数据模板的详细信息，请参阅[数据模板化概述 \(.NET Framework\)](#)。

## 数据验证

接受用户输入的大多数应用都需要具有验证逻辑，以确保用户输入了预期信息。可基于类型、范围、格式或特定于应用的其他要求执行验证检查。本部分讨论数据验证在 WPF 中的工作原理。

## 将验证规则与绑定关联

WPF 数据绑定模型允许将 ValidationRules 与 Binding 对象关联。例如，以下示例将 TextBox 绑定到名为 StartPrice 的属性，并将 ExceptionValidationRule 对象添加到 Binding.ValidationRules 属性。

XAML

```

<TextBox Name="StartPriceEntryForm" Grid.Row="2"
        Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5"
        Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">

```

```
<Binding.ValidationRules>
    <ExceptionValidationRule />
</Binding.ValidationRules>
</Binding>
</TextBox.Text>
</TextBox>
```

[ValidationRule](#) 对象检查属性的值是否有效。 WPF 有两种类型的内置 [ValidationRule](#) 对象：

- [ExceptionValidationRule](#) 检查在绑定源属性更新期间引发的异常。 在以上示例中，`StartPrice` 为整数类型。 当用户输入的值无法转换为整数时，将引发异常，这会导致将绑定标记为无效。 用于显式设置 [ExceptionValidationRule](#) 的替代语法是在 [Binding](#) 或 [MultiBinding](#) 对象上将 [ValidatesOnExceptions](#) 属性设置为 `true`。
- [DataErrorValidationRule](#) 对象检查实现 [IDataErrorInfo](#) 接口的对象所引发的错误。 有关使用此验证规则的详细信息，请参阅 [DataErrorValidationRule](#)。 用于显式设置 [DataErrorValidationRule](#) 的替代语法是在 [Binding](#) 或 [MultiBinding](#) 对象上将 [ValidatesOnDataErrors](#) 属性设置为 `true`。

还可以通过从 [ValidationRule](#) 类派生并实现 [Validate](#) 方法来创建自己的验证规则。 以下示例演示了什么是数据绑定部分中添加产品清单“起始日期”[TextBox](#) 所用的规则。

C#

```
public class FutureDateRule : ValidationRule
{
    public override ValidationResult Validate(object value, CultureInfo
cultureInfo)
    {
        // Test if date is valid
        if (DateTime.TryParse(value.ToString(), out DateTime date))
        {
            // Date is not in the future, fail
            if (DateTime.Now > date)
                return new ValidationResult(false, "Please enter a date in
the future.");
            }
            else
            {
                // Date is not a valid date, fail
                return new ValidationResult(false, "Value is not a valid
date.");
            }

            // Date is valid and in the future, pass
            return ValidationResult.ValidResult;
    }
}
```

`StartDateEntryForm` 使用此 `FutureDateRule`, 如以下示例所示。

#### XAML

```
<TextBox Name="StartDateEntryForm" Grid.Row="3"
    Validation.ErrorTemplate="{StaticResource validationTemplate}"
    Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5"
Grid.ColumnSpan="2">
    <TextBox.Text>
        <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged"
            Converter="{StaticResource dateConverter}" >
            <Binding.ValidationRules>
                <src:FutureDateRule />
            </Binding.ValidationRules>
        </Binding>
    </TextBox.Text>
</TextBox>
```

因为 `UpdateSourceTrigger` 值为 `PropertyChanged`, 所以绑定引擎会在每次击键时更新源值, 这意味着它还会在每次击键时检查 `ValidationRules` 集合中的每条规则。 我们会在“验证过程”一节中对此深入讨论。

## 提供视觉反馈

如果用户输入的值无效, 你可能希望在应用 UI 上提供一些有关错误的反馈。 提供此类反馈的一种方法是将 `Validation.ErrorTemplate` 附加属性设置为自定义 `ControlTemplate`。 如前面部分所示, `StartDateEntryForm` 使用名为 `validationTemplate` 的 `ErrorTemplate`。 以下示例显示了 `validationTemplate` 的定义。

#### XAML

```
<ControlTemplate x:Key="validationTemplate">
    <DockPanel>
        <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
        <AdornedElementPlaceholder/>
    </DockPanel>
</ControlTemplate>
```

`AdornedElementPlaceholder` 元素指定应放置待装饰控件的位置。

此外, 还可以使用 `ToolTip` 来显示错误消息。 `StartDateEntryForm` 和 `StartPriceEntryForm` 都使用样式 `textStyleTextBox`, 该样式创建显示错误消息的 `ToolTip`。 以下示例显示了 `textStyleTextBox` 的定义。 如果绑定元素属性上的一个或多个绑定出错, 则附加属性 `Validation.HasError` 为 `true`。

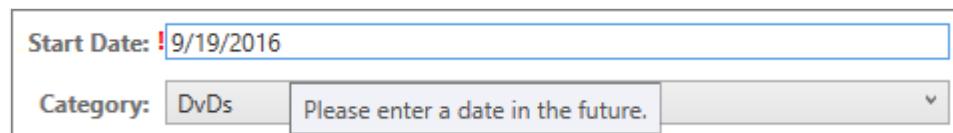
#### XAML

```

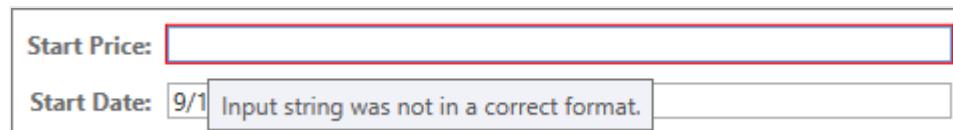
<Style x:Key="textStyleTextBox" TargetType="TextBox">
    <Setter Property="Foreground" Value="#333333" />
    <Setter Property="MaxLength" Value="40" />
    <Setter Property="Width" Value="392" />
    <Style.Triggers>
        <Trigger Property="Validation.HasError" Value="true">
            <Setter Property="ToolTip"
                Value="{Binding
                    (Validation.Errors).CurrentItem.ErrorContent, RelativeSource={RelativeSource
                    Self}}" />
        </Trigger>
    </Style.Triggers>
</Style>

```

使用自定义 ErrorTemplate 和 ToolTip 时，StartDateEntryForm TextBox 在发生验证错误时如下所示。



如果 Binding 具有关联的验证规则，但未在绑定控件上指定 ErrorTemplate，则发生验证错误时，将使用默认的 ErrorTemplate 通知用户。默认的 ErrorTemplate 是一个控件模板，它在装饰层中定义红色边框。使用默认的 ErrorTemplate 和 ToolTip 时，StartPriceEntryForm TextBox 的 UI 在发生验证错误时如下所示。



有关如何提供逻辑以验证对话框中所有控件的示例，请参阅对话框概述中的“自定义对话框”部分。

## 验证过程

通常，在目标的值传输到绑定源属性时会进行验证。此传输在 TwoWay 和 OneWayToSource 绑定上发生。重申一下，导致源更新的因素取决于 UpdateSourceTrigger 属性的值，如触发源更新的因素部分所述。

以下各项描述了验证过程。只要验证过程中发生验证错误或其他类型的错误，该过程就会中断：

1. 绑定引擎检查是否为该 Binding 定义了任何将 ValidationStep 设置为 RawProposedValue 的自定义 ValidationRule 对象，在这种情况下，绑定引擎将对每个 ValidationRule 调用 Validate 方法，直到其中一个出错或直到全部通过。

2. 绑定引擎随后会调用转换器（如果存在）。
3. 如果转换器成功，则绑定引擎会检查是否为该 Binding 定义了任何将 ValidationStep 设置为 ConvertedProposedValue 的自定义 ValidationRule 对象，在这种情况下，绑定引擎将对每个 ValidationRule（将 ValidationStep 设置为 ConvertedProposedValue）调用 Validate 方法，直到其中一个出错或直到全部通过。
4. 绑定引擎设置源属性。
5. 绑定引擎检查是否为该 Binding 定义了任何将 ValidationStep 设置为 UpdatedValue 的自定义 ValidationRule 对象，在这种情况下，绑定引擎将对每个 ValidationRule（将 ValidationStep 设置为 UpdatedValue）调用 Validate 方法，直到其中一个出错或直到全部通过。如果 DataErrorValidationRule 与绑定关联并且其 ValidationStep 设置为默认的 UpdatedValue，则此时将检查 DataErrorValidationRule。此时检查将 ValidatesOnDataErrors 设置为 true 的所有绑定。
6. 绑定引擎检查是否为该 Binding 定义了任何将 ValidationStep 设置为 CommittedValue 的自定义 ValidationRule 对象，在这种情况下，绑定引擎将对每个 ValidationRule（将 ValidationStep 设置为 CommittedValue）调用 Validate 方法，直到其中一个出错或直到全部通过。

如果 ValidationRule 在整个过程中的任何时间都没有通过，则绑定引擎会创建 ValidationError 对象并将其添加到绑定元素的 Validation.Errors 集合中。绑定引擎在任何给定步骤运行 ValidationRule 对象之前，它会删除在执行该步骤期间添加到绑定元素的 Validation.Errors 附加属性的所有 ValidationError。例如，如果将 ValidationStep 设置为 UpdatedValue 的 ValidationRule 失败，则下次执行验证过程时，绑定引擎会在调用将 ValidationStep 设置为 UpdatedValue 的任何 ValidationRule 之前删除 ValidationError。

如果 Validation.Errors 不为空，则元素的 Validation.HasError 附加属性设置为 true。此外，如果 Binding 的 NotifyOnValidationError 属性设置为 true，则绑定引擎将在元素上引发 Validation.Error 附加事件。

另请注意，任何方向（目标到源或源到目标）的有效值传输操作都会清除 Validation.Errors 附加属性。

如果绑定具有关联的 ExceptionValidationRule，或将 ValidatesOnExceptions 属性设置为 true，并且在绑定引擎设置源时引发异常，则绑定引擎将检查是否存在 UpdateSourceExceptionFilter。可以使用 UpdateSourceExceptionFilter 回叫来提供用于处理异常的自定义处理程序。如果未在 Binding 上指定 UpdateSourceExceptionFilter，则绑定引擎会创建具有异常的 ValidationError 并将其添加到绑定元素的 Validation.Errors 集合中。

# 调试机制

可以在与绑定相关的对象上设置附加属性 `PresentationTraceSources.TraceLevel`，以接收有关特定绑定状态的信息。

## 另请参阅

- [数据绑定演示 ↗](#)
- [绑定声明概述](#)
- [绑定源概述](#)
- [DataErrorValidationRule](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 绑定声明概述 (WPF .NET)

项目 • 2023/10/13

通常，开发人员直接在要将数据绑定到的 UI 元素的 XAML 标记中声明绑定。不过，也可以在代码中声明绑定。本文介绍如何在 XAML 和代码中声明绑定。

## 先决条件

在阅读本文之前，请务必熟悉标记扩展的概念和使用。有关标记扩展的详细信息，请参阅[标记扩展和 WPF XAML](#)。

本文不介绍数据绑定概念。有关数据绑定概念的讨论，请参阅[数据绑定概述](#)。

## 在 XAML 中声明绑定

`Binding` 是标记扩展。使用绑定扩展声明绑定时，声明包含一系列子句，这些子句跟在 `Binding` 关键字后面，并由逗号 (,) 分隔。绑定声明中的子句可以按任意顺序排列，有多种可能的组合。这些子句是 `Name=Value` 对，其中 `Name` 是 `Binding` 属性的名称，`Value` 是为该属性设置的值。

在标记中创建绑定声明字符串时，必须将这些字符串附加到目标对象的特定依赖属性。下面的示例演示如何使用绑定扩展并指定 `Source` 和 `Path` 属性来绑定 `TextBox.Text` 属性。

XAML

```
<TextBlock Text="{Binding Source={StaticResource myDataSource},  
Path=Name}"/>
```

可以通过这种方式指定 `Binding` 类的大多数属性。有关绑定扩展以及使用绑定扩展无法设置的 `Binding` 属性列表的详细信息，请参阅[绑定标记扩展 \(.NET Framework\)](#)概述。

## 对象元素语法

对象元素语法是创建绑定声明的替代方法。在大多数情况下，使用标记扩展或对象元素语法没有特定的优势。不过，如果标记扩展不支持你的方案，例如，当属性值是不存在任何类型转换的非字符串类型时，需要使用对象元素语法。

上一部分演示了如何使用 XAML 扩展进行绑定。以下示例演示如何使用对象元素语法执行相同的绑定：

XAML

```
<TextBlock>
    <TextBlock.Text>
        <Binding Source="{StaticResource myDataSource}" Path="Name"/>
    </TextBlock.Text>
</TextBlock>
```

有关其他术语的详细信息，请参阅 [XAML 语法详述 \(.NET Framework\)](#)。

## MultiBinding 和 PriorityBinding

MultiBinding 和 PriorityBinding 不支持 XAML 扩展语法。因此，在 XAML 中声明 MultiBinding 或 PriorityBinding 时必须使用对象元素语法。

## 在代码中创建绑定

指定绑定的另一种方法是直接在代码中的 [Binding](#) 对象上设置属性，然后将绑定分配给属性。下面的示例演示如何在代码中创建 [Binding](#) 对象。

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Make a new data source object
    var personDetails = new Person()
    {
        Name = "John",
        Birthdate = DateTime.Parse("2001-02-03")
    };

    // New binding object using the path of 'Name' for whatever source
    // object is used
    var nameBindingObject = new Binding("Name");

    // Configure the binding
    nameBindingObject.Mode = BindingMode.OneWay;
    nameBindingObject.Source = personDetails;
    nameBindingObject.Converter = NameConverter.Instance;
    nameBindingObject.ConverterCulture = new CultureInfo("en-US");

    // Set the binding to a target object. The TextBlock.Name property on
    // the NameBlock UI element
    BindingOperations.SetBinding(NameBlock, TextBlock.TextProperty,
        nameBindingObject);
}
```

前面的代码在绑定上设置了以下内容：

- 数据源对象上属性的路径。
- 绑定的模式。
- 在本例中，数据源是表示人的简单对象实例。
- 一个可选转换器，该转换器在将数据源对象传入的值分配给目标属性之前对其进行处理。

如果要绑定的对象是 [FrameworkElement](#) 或 [FrameworkContentElement](#)，可以直接在对象上调用 `SetBinding` 方法，而不是使用 `BindingOperations.SetBinding`。有关示例，请参阅[如何：在代码中创建绑定](#)。

上一个示例使用 `Person` 的简单数据对象类型。下面是该对象的代码：

C#

```
class Person
{
    public string Name { get; set; }
    public DateTime Birthdate { get; set; }
}
```

## 绑定路径语法

使用 `Path` 属性指定要绑定到的源值：

- 在最简单的情况下，`Path` 属性值是要用于绑定的源对象的属性名称，如 `Path=PropertyName`。
- 在 C# 中，可以通过类似语法指定属性的子属性。例如，子句 `Path=ShoppingCart.Order` 设置与对象或属性 `ShoppingCart` 的子属性 `Order` 的绑定。
- 若要绑定到附加属性，请将附加属性置于括号中。例如，若要绑定到附加属性 `DockPanel.Dock`，则语法为 `Path=(DockPanel.Dock)`。
- 可以在已应用索引器的属性名后面的方括号内指定属性的索引器。例如，子句 `Path=ShoppingCart[0]` 将绑定设置为与属性的内部索引处理文本字符串“0”的方式对应的索引。还支持嵌套索引器。
- 可以在 `Path` 子句中混用索引器和子属性，例如 `Path=ShoppingCart.ShippingInfo[MailingAddress,Street]`。
- 索引器内部。可以使用多个由逗号 (,) 分隔的索引器参数。可以使用括号指定每个参数的类型。例如，可以使用 `Path="[(sys:Int32)42,(sys:Int32)24]"`，其中 `sys`

将映射到 `System` 命名空间。

- 如果源是集合视图，则可以使用斜杠 (/) 指定当前项。例如，子句 `Path=/` 设置与视图中当前项的绑定。如果源是集合，则此语法指定默认集合视图的当前项。
- 可以组合使用属性名和斜杠，以遍历作为集合的属性。例如，`Path=/Offices/ManagerName` 指定源集合的当前项，源集合中包含的 `Offices` 属性也是一个集合。它的当前项是一个包含 `ManagerName` 属性的对象。
- 句点 (.) 路径也可以用于绑定到当前源。例如，`Text="{Binding}"` 等效于 `Text="{Binding Path=.}"`。

## 转义机制

- 在索引器 ([ ]) 内部，脱字符号 (^) 用于对下一个字符进行转义。
- 如果在 XAML 中设置 `Path`，还需要（使用 XML 实体）对 XML 语言定义专用的某些字符进行转义：
  - 使用 `&amp;` 对字符“&”进行转义。
  - 使用 `&gt;` 对结束标记“>”进行转义。
- 此外，如果在属性中使用标记扩展语法描述整个绑定，需要（使用反斜杠 (\)）对 WPF 标记扩展分析器专用的字符进行转义：
  - 反斜杠 (\) 本身是转义字符。
  - 等号 (=) 将属性名与属性值分隔开。
  - 逗号 (,) 用于分隔属性。
  - 右大括号 (}) 是标记扩展的结尾。

## 绑定方向

使用 `Binding.Mode` 属性指定绑定的方向。以下模式是可供绑定更新的选项：

绑定模式	说明
<code>BindingMode.TwoWay</code>	无论是目标属性还是源属性，只要发生了更改，就会更新目标属性或属性。
<code>BindingMode.OneWay</code>	仅当源属性发生更改时更新目标属性。

绑定模式	说明
BindingMode.OneTime	仅当应用程序启动或 <a href="#">DataContext</a> 进行更改时更新目标属性。
BindingMode.OneWayToSource	在目标属性更改时，更新源属性。
BindingMode.Default	导致使用目标属性的默认值 <a href="#">Mode</a> 。

有关详细信息，请参见 [BindingMode 枚举](#)。

下面的示例演示如何设置 [Mode 属性](#)：

#### XAML

```
<TextBlock Name="IncomeText" Text="{Binding Path=TotalIncome, Mode=OneTime}" />
```

若要检测源更改（适用于 [OneWay](#) 和 [TwoWay](#) 绑定），则源必须实现合适的属性更改通知机制，例如 [INotifyPropertyChanged](#)。有关详细信息，请参阅[提供更改通知](#)。

对于 [TwoWay](#) 或 [OneWayToSource](#) 绑定，可以通过设置 [UpdateSourceTrigger](#) 属性来控制源更新执行时间。有关详细信息，请参阅 [UpdateSourceTrigger](#)。

## 默认行为

如果未在声明中指定默认行为，则默认行为如下所示：

- 创建一个默认转换器，尝试在绑定源值和绑定目标值之间执行类型转换。如果不能进行转换，默认转换器会返回 `null`。
- 如果未设置 [ConverterCulture](#)，绑定引擎会使用绑定目标对象的 [Language](#) 属性。在 XAML 中，此属性默认为“`en-US`”，如果已显式设置了一个值，则从页面的根元素（或任何元素）继承该值。
- 只要绑定已具有数据上下文（例如，来自父元素的继承数据上下文），并且该上下文返回的任何项或集合无需进一步修改路径即适用于绑定，则绑定声明可以不必使用任何子句：`{Binding}`。这通常是为数据样式指定绑定所用的方法，其中该绑定作用于某个集合。有关详细信息，请参阅[将整个对象用作绑定源](#)。
- 默认 [Mode](#) 可能为单向，也可能为双向，具体取决于要绑定的依赖属性。始终可以显式声明绑定模式，以确保绑定具有所需行为。一般情况下，用户可编辑的控件属性（例如 [TextBox.Text](#) 和 [RangeBase.Value](#)）默认为双向绑定，但其他大多数属性默认为单向绑定。

- 默认 `UpdateSourceTrigger` 值可能为 `PropertyChanged`，也可能为 `LostFocus`，具体同样取决于绑定的依赖属性。大多数依赖属性的默认值为 `PropertyChanged`，而 `TextBox.Text` 属性的默认值为 `LostFocus`。

## 另请参阅

- [数据绑定概述](#)
- [绑定源概述](#)
- [PropertyPath XAML 语法 \(.NET Framework\)](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 绑定源概述 (WPF .NET)

项目 · 2023/10/13

在数据绑定中，绑定源对象是指用户从其获取数据的对象。本文讨论可以用作绑定源的对象类型，如 .NET CLR 对象、XML 和 [DependencyObject](#) 对象。

## 绑定源类型

Windows Presentation Foundation (WPF) 数据绑定支持以下绑定源类型：

- **.NET 公共语言运行时 (CLR) 对象**

可以绑定到任何公共语言运行时 (CLR) 对象的公共属性、子属性和索引器。绑定引擎使用 CLR 反射来获取属性值。实现了 [ICustomTypeDescriptor](#) 或具有已注册 [TypeDescriptionProvider](#) 的对象也可以使用绑定引擎。

有关如何实现可用作绑定源的类的详细信息，请参阅本文后面的[在对象上实现绑定源](#)。

- **动态对象**

可以绑定到对象的可用属性和索引器，该对象实现 [IDynamicMetaObjectProvider](#) 接口。如果可以访问代码中的成员，则可以绑定到该成员。例如，如果动态对象使用用户可以通过 `SomeObject.AProperty` 访问代码中的成员，则可以通过将绑定路径设置为 `AProperty` 来绑定到该成员。

- **ADO.NET 对象**

可以绑定到 ADO.NET 对象，例如 [DataTable](#)。ADO.NET [DataView](#) 实现 [IBindingList](#) 接口，该接口提供绑定引擎侦听的更改通知。

- **XML 对象**

可以绑定到 [XmlNode](#)、 [XmlDocument](#) 或 [XmlElement](#)，并对其运行 `xPath` 查询。访问 XML 数据（标记中的绑定源）的便捷方法是使用 [XmlDataProvider](#) 对象。有关详细信息，请参阅[使用 XmlDataProvider 和 XPath 查询绑定到 XML 数据 \(.NET Framework\)](#)。

使用 LINQ to XML，还可以绑定到  [XElement](#) 或  [XDocument](#)，或者绑定到对这些类型的对象运行查询而得到的结果。使用 LINQ to XML 访问 XML 数据（标记中的绑定源）的便捷方法是使用 [ObjectDataProvider](#) 对象。有关详细信息，请参阅[绑定到 XDocument、 XElement 或 LINQ for XML 查询结果 \(.NET Framework\)](#)。

- [DependencyObject 对象](#)

可以绑定到任何 [DependencyObject](#) 的依赖属性。有关示例，请参阅[绑定两个控件的属性 \(.NET Framework\)](#)。

## 在对象上实现绑定源

CLR 对象可能成为绑定源。实现用作绑定源的类时，需要注意一些事项。

### 提供更改通知

如果要使用 [OneWay](#) 或 [TwoWay](#) 绑定，请实现适当的“属性更改”通知机制。对于 CLR 或动态类，建议的机制是实现 [INotifyPropertyChanged](#) 接口。有关详细信息，请参阅[如何：实现属性更改通知 \(.NET Framework\)](#)。

有两种方法通知订阅者属性更改：

1. 实现 [INotifyPropertyChanged](#) 接口。

这是建议的通知机制。[INotifyPropertyChanged](#) 提供绑定系统所遵守的 [PropertyChanged](#) 事件。通过引发此事件并提供已更改属性的名称，你将通知绑定目标更改。

2. 实现 [PropertyChanged](#) 模式。

每个需要通知绑定目标其更改的属性都有相应的 [PropertyNameChanged](#) 事件，其中 [PropertyName](#) 是属性的名称。每次更改属性时都会引发该事件。

如果绑定源实现了其中一个通知机制，将会自动进行目标更新。如果绑定源由于某种原因未提供正确的属性更改通知，则可以使用 [UpdateTarget](#) 方法来显式更新目标属性。

## 其他特征

下表提供了需要注意的其他要点：

- 作为绑定源的数据对象可以在 XAML 中声明为资源，前提是它们有一个“无参数构造函数”。否则，必须在代码中创建数据对象，并直接将其分配给 XAML 对象树的数据上下文，或者作为绑定的绑定源。
- 用作绑定源属性的属性必须是类的公共属性。不能出于绑定目的来访问显式定义的接口属性，也不能访问没有基实现的受保护、私有、内部或虚拟属性。
- 不能绑定到公共字段。

- 类中声明的属性类型是传递给绑定的类型。不过，绑定最终所用的类型取决于绑定目标属性的类型，而不是绑定源属性的类型。如果类型不同，可能需要编写一个转换器来处理自定义属性最初传递给绑定的方式。有关详细信息，请参阅 [IValueConverter](#)。

## 整个对象作为绑定源

可以将整个对象用作绑定源。通过使用 [Source](#) 或 [DataContext](#) 属性指定绑定源，然后提供一个空白绑定声明：`{Binding}`。适用的场景包括绑定到属于类型字符串的对象、绑定到具有感兴趣的多个属性的对象或绑定到集合对象。有关绑定到整个集合对象的示例，请参阅[如何对分层数据使用主-从模式 \(.NET Framework\)](#)。

你可能需要应用自定义逻辑，以便数据对于绑定的目标属性有意义。自定义逻辑可以采用自定义转换器或 [DataTemplate](#) 的形式。有关转换器的详细信息，请参阅[数据转换](#)。有关数据模板的详细信息，请参阅[数据模板化概述 \(.NET Framework\)](#)。

## 集合对象作为绑定源

通常，要用作绑定源的对象是自定义对象的集合。每个对象都用作重复绑定的一个实例的源。例如，可能存在由 `CustomerOrder` 对象组成的 `CustomerOrders` 集合，其中应用程序会循环访问该集合以确定存在的订单数量，以及每个订单中的数据。

你可以枚举实现 [IEnumerable](#) 接口的任何集合。但是，若要设置动态绑定，以便集合中的插入或删除操作可以自动更新 UI，则集合必须实现 [INotifyCollectionChanged](#) 接口。此接口公开一个事件，只要基础集合发生更改，就必须引发该事件。

`ObservableCollection<T>` 类是公开 [INotifyCollectionChanged](#) 接口的数据集合的内置实现。集合中的个别数据对象必须满足前面章节中所述的要求。有关示例，请参阅[如何创建和绑定到 ObservableCollection \(.NET Framework\)](#)。在实现自己的集合前，请考虑使用 `ObservableCollection<T>` 或现有集合类之一，例如 `List<T>`、`Collection<T>` 和 `BindingList<T>` 等。

将指定集合作为绑定源时，WPF 不会直接绑定到该集合。相反，WPF 实际上绑定到集合的默认视图。有关默认视图的信息，请参阅[使用默认视图](#)。

如果有高级方案，并且想要实现自己的集合，请考虑使用  [IList](#) 接口。此接口提供可通过索引逐个访问的对象的非泛型集合，这可以提高性能。

## 数据绑定中的权限要求

与 .NET Framework 不同，.NET 运行时具有完全信任的安全性。所有数据绑定都以与运行应用程序的用户相同的访问权限运行。

## 另请参阅

- [ObjectDataProvider](#)
- [XmlDataProvider](#)
- [数据绑定概述](#)
- [绑定源概述](#)
- [使用 LINQ to XML 进行 WPF 数据绑定概述 \(.NET Framework\)](#)
- [优化性能：数据绑定 \(.NET Framework\)](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# 如何绑定到枚举 (WPF .NET)

项目 • 2023/10/19

此示例演示如何绑定到枚举。遗憾的是，没有直接方法可以将枚举用作数据绑定源。但是，[Enum.GetValues\(Type\)](#) 方法可返回值的集合。这些值可以包装在 [ObjectDataProvider](#) 中并用作数据源。

[ObjectDataProvider](#) 类型提供了一种在 XAML 中创建对象并将其用作数据源的便捷方式。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 引用枚举

使用 [ObjectDataProvider](#) 类型包装枚举类型本身提供的枚举值数组。

1. 在应用程序 XAML 或正在使用的对象的 XAML 中创建一个新的 [ObjectDataProvider](#) 作为 XAML 资源。此示例使用一个窗口并使用资源键 [EnumDataSource](#) 创建 [ObjectDataProvider](#)。

XAML

```
<Window.Resources>
    <ObjectDataProvider x:Key="EnumDataSource"
        ObjectType="{x:Type sys:Enum}"
        MethodName="GetValues">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="HorizontalAlignment" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
```

在此示例中，[ObjectDataProvider](#) 使用三个属性来检索枚举：

properties	说明
<a href="#">ObjectType</a>	数据提供程序要返回的对象类型。在本示例中为 <a href="#">System.Enum</a> 。 <a href="#">sys:</a> XAML 命名空间映射到 <a href="#">System</a> 。
<a href="#">MethodName</a>	要在 <a href="#">System.Enum</a> 类型上运行的方法的名称。在本示例中为 <a href="#">Enum.GetValues</a> 。

properties	说明
MethodParameters	要提供给 <code>MethodName</code> 方法的值的集合。在此示例中，该方法采用枚举的 <code>System.Type</code> 。

实际上，XAML 正在分解方法调用、方法名称、参数和返回类型。上一示例中配置的 `ObjectDataProvider` 等效于以下代码：

C#

```
var enumDataSource =
    System.Enum.GetValues(typeof(System.Windows.HorizontalAlignment));
```

2. 引用 `ObjectDataProvider` 资源。以下 XAML 列出了 `ListBox` 控件中的枚举值：

XAML

```
<ListBox Name="myComboBox" SelectedIndex="0"
         ItemsSource="{Binding Source={StaticResource
EnumDataSource}}"/>
```

## 完整 XAML

以下 XAML 代码表示执行以下操作的简单窗口：

1. 将 `ObjectDataProvider` 数据源中的 `HorizontalAlignment` 枚举包装为资源。
2. 提供一个 `ListBox` 控件以列出所有枚举值。
3. 将 `Button` 控件的 `HorizontalAlignment` 属性绑定到 `ListBox` 中的选定项。

XAML

```
<Window x:Class="ArticleExample.BindEnumFull"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"
        SizeToContent="WidthAndHeight"
        Title="Enum binding">
<Window.Resources>
    <ObjectDataProvider x:Key="EnumDataSource"
                       ObjectType="{x:Type sys:Enum}"
                       MethodName="GetValues">
        <ObjectDataProvider.MethodParameters>
            <x:Type TypeName="HorizontalAlignment" />
        </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
```

```
<StackPanel Width="300" Margin="10">
    <TextBlock>Choose the HorizontalAlignment value of the Button:
</TextBlock>

    <ListBox Name="myComboBox" SelectedIndex="0"
        ItemsSource="{Binding Source={StaticResource
EnumDataSource}}"/>

    <Button Content="I'm a button"
        HorizontalAlignment="{Binding ElementName=myComboBox,
Path=SelectedItem}" />
</StackPanel>
</Window>
```

## 另请参阅

- [数据绑定概述](#)
- [绑定源概述](#)
- [StaticResource 标记扩展](#)
- [绑定到枚举的替代方法](#)

### 在 GitHub 上与我们协作

可以在 GitHub [上找到此内容的源](#)，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback [here](#).

 提出文档问题

 提供产品反馈

# 路由事件概述 (WPF .NET)

项目 • 2023/10/19

Windows Presentation Foundation (WPF) 应用程序开发人员和组件创建者可以使用路由事件，通过元素树来传播事件，并在树中的多个侦听器上调用事件处理程序。公共语言运行时 (CLR) 事件中没有这些功能。有一些 WPF 事件是路由事件，例如 [ButtonBase.Click](#)。本文介绍路由事件的基本概念，并对何时以及如何响应路由事件提供了指导。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定读者已基本了解公共语言运行时 (CLR)、面向对象的编程以及如何将 [WPF 元素布局](#) 概念化为树。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 [WPF 应用程序](#)。

## 什么是路由事件？

可以从功能或实现的角度来理解路由事件：

- 从功能角度来看，路由事件是一种可以针对元素树中的多个侦听器（而不是仅针对事件源）调用处理程序的事件。事件侦听器是附加和调用了事件处理程序的元素。事件源是最初引发事件的元素或对象。
- 从实现的角度来看，路由事件是向 WPF 事件系统注册的事件，由 [RoutedEventArgs](#) 类的实例提供支持，并由 WPF 事件系统处理。通常，路由事件是使用 CLR 事件“包装器”实现的，可以在 XAML 和代码隐藏中启用附加处理程序，就像你使用 CLR 事件一样。

WPF 应用程序通常包含许多元素，它们要么在 XAML 中声明，要么在代码中实例化。应用程序的元素存在于其元素树中。根据路由事件的定义方式，当事件在源元素上被引发时，它会：

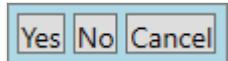
- 通过元素树从源元素浮升到根元素（通常是页面或窗口）。
- 通过元素树从根元素到源元素向下进行隧道操作。
- 不会遍历元素树，只发生在源元素上。

来看看下面的一部分元素树：

#### XAML

```
<Border Height="30" Width="200" BorderBrush="Gray" BorderThickness="1">
    <StackPanel Background="LightBlue" Orientation="Horizontal"
Button.Click="YesNoCancelButton_Click">
        <Button Name="YesButton">Yes</Button>
        <Button Name="NoButton">No</Button>
        <Button Name="CancelButton">Cancel</Button>
    </StackPanel>
</Border>
```

元素树的呈现的效果如下所示：



这三个按钮中的每一个都是潜在的 `Click` 事件源。单击其中一个按钮时，它会引发 `Click` 事件，从按钮浮升到根元素。`Button` 和 `Border` 元素没有附加事件处理程序，但 `StackPanel` 有。树中较高、未显示的其他元素可能也附加了 `Click` 事件处理程序。当 `Click` 事件到达 `StackPanel` 元素时，WPF 事件系统将调用附加到它的 `YesNoCancelButton_Click` 处理程序。示例中 `Click` 事件的事件路由为：  
`Button` > `StackPanel` > `Border` > 连续的父元素。

#### ① 备注

最初引发路由事件的元素在事件处理程序参数中被标识为 `RoutedEventArgs.Source`。事件侦听器是附加和调用了事件处理程序的元素，它在事件处理程序参数中标识为 `sender`。

## 路由事件的顶层方案

下面介绍一些需运用路由事件概念的方案，并将其与典型的 CLR 事件进行区分：

- 控件的撰写和封装：**WPF 中的各个控件都有一个丰富的内容模型。例如，可以将图像放在 `Button` 的内部，这会有效地扩展按钮的可视化树。但是，添加的图像不能破坏按钮的命中测试行为，它需要在用户单击图像像素时做出响应。
- 单一处理程序附加点：**可以为每个按钮的 `Click` 事件注册一个处理程序，但通过路由事件，可以附加单个处理程序，如前面的 XAML 示例所示。这样，你就可以更改单一处理程序下的元素树，例如添加或删除更多按钮，而无需注册每个按钮的

`Click` 事件。引发 `click` 事件时，处理程序逻辑可以确定事件来自何处。在前面演示的 XAML 元素树中指定的以下处理程序包含该逻辑：

C#

```
private void YesNoCancelButton_Click(object sender, RoutedEventArgs e)
{
    FrameworkElement sourceFrameworkElement = e.Source as
FrameworkElement;
    switch (sourceFrameworkElement.Name)
    {
        case "YesButton":
            // YesButton logic.
            break;
        case "NoButton":
            // NoButton logic.
            break;
        case "CancelButton":
            // CancelButton logic.
            break;
    }
    e.Handled = true;
}
```

- 类处理：路由事件支持在类中定义的类事件处理程序。类处理程序会在该类的任何实例上处理同一事件的任何实例处理程序之前处理事件。
- 在没有反射的情况下引用事件：每个路由事件都会创建一个 `RoutedEventArgs` 字段标识符，从而提供一种不需要静态反射或运行时反射的可靠的事件标识技术来标识事件。

## 路由事件的实现方式

路由事件是向 WPF 事件系统注册的事件，由 `RoutedEventArgs` 类的实例提供支持，并由 WPF 事件系统处理。从注册获取的 `RoutedEventArgs` 实例通常存储为已注册的类的 `public static readonly` 成员。该类称为事件“拥有者”类。通常，路由事件会实现一个名称相同的 CLR 事件“包装器”。CLR 事件包装器包含 `add` 和 `remove` 访问器，可以通过特定于语言的事件语法在 XAML 和代码隐藏中附加处理程序。`add` 和 `remove` 访问器重写其 CLR 实现，并调用路由事件 `AddHandler` 和 `RemoveHandler` 方法。路由事件的支持和连接机制在概念上与以下机制相似：依赖属性是一个 CLR 属性，该属性由 `DependencyProperty` 类提供支持并向 WPF 属性系统注册。

以下示例注册 `Tap` 路由事件，存储返回的 `RoutedEventArgs` 实例，并实现 CLR 事件包装器。

C#

```
// Register a custom routed event using the Bubble routing strategy.
public static readonly RoutedEvent TapEvent =
EventManager.RegisterRoutedEvent(
    name: "Tap",
    routingStrategy: RoutingStrategy.Bubble,
    handlerType: typeof(RoutedEventHandler),
    ownerType: typeof(CustomButton));

// Provide CLR accessors for adding and removing an event handler.
public event RoutedEventHandler Tap
{
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

## 路由策略

路由事件使用以下三种路由策略之一：

- **浮升**：最初，调用事件源上的事件处理程序。 路由事件随后会路由到后续的父级元素，依次调用其事件处理程序，直到到达元素树的根。 大多数路由事件都使用浮升路由策略。 浮升路由事件通常用于报告来自复合控件或其他 UI 元素的输入或状态变化。
- **隧道**：最初将调用元素树的根处的事件处理程序。 路由事件随后会路由到后续的子级元素，依次调用其事件处理程序，直到到达事件源。 遵循隧道路由的事件也称为**预览**事件。 WPF 输入事件通常是作为**预览和浮升对**实现的。
- **直接**：仅调用事件源上的事件处理程序。 这种非路由策略类似于 Windows 窗体 UI 框架事件，它是标准 CLR 事件。 与 CLR 事件不同，直接路由事件支持类处理，可由 EventSetters 和 EventTriggers 使用。

## 为什么使用路由事件？

作为应用程序开发人员，你不需要始终了解或关注要处理的事件是否作为路由事件实现。 路由事件具有特殊的行为，但是，如果在引发该行为的元素上处理事件，则该行为通常会不可见。 但是，如果要将事件处理程序附加到父元素以处理子元素引发的事件（例如在复合控件中），路由事件就是有意义的。

路由事件侦听器不需要让它们处理的路由事件成为其类的成员。 任何 **UIElement** 或 **ContentElement** 可以是任一路由事件的事件侦听器。 由于可视元素派生自 **UIElement** 或 **ContentElement**，你可以将路由事件作为一个概念性的“接口”，支持在应用程序中的不同元素之间交换事件信息。 路由事件的这个“接口”概念特别适用于**输入事件**。

路由事件支持在事件路由路线上的元素之间交换事件信息，因为每个侦听器都可以访问事件数据的同一实例。如果事件数据中的某个元素更改了某些内容，则该更改对事件路由中的后续元素可见。

除了路由方面的原因，还有以下两个原因让你可能会选择实现路由事件而不是标准 CLR 事件：

- 一些 WPF 样式和模板功能（如 [EventSetters](#) 和 [EventTriggers](#)）要求被引用的事件是路由事件。
- 路由事件支持[类事件处理程序](#)，在侦听器类的任何实例上处理同一事件的任何实例处理程序之前处理事件。此功能在控件设计中非常有用，因为类处理程序可以强制执行事件驱动的类行为，这些行为不能被实例处理程序意外地禁止。

## 附加并实现路由事件处理程序

在 XAML 中，通过将事件名称声明为事件侦听器元素上的属性，将事件处理程序附加到元素。属性值是处理方法名称。处理方法必须在 XAML 页的代码隐藏分部类中实现。事件侦听器是附加和调用了事件处理程序的元素。

对于（通过继承或其他方式）成为侦听器类的成员的事件，可以按如下所示附加处理程序：

XAML

```
<Button Name="Button1" Click="Button_Click">Click me</Button>
```

如果事件不是侦听器类的成员，则必须以 `<owner type>.<event name>` 的形式使用限定的事件名称。例如，由于 [StackPanel](#) 类不实现 [Click](#) 事件，若要将处理程序附加到浮升到该元素的 `Click` 事件的 [StackPanel](#)，需要使用限定的事件名称语法：

XAML

```
<StackPanel Name="StackPanel1" Button.Click="Button_Click">
    <Button>Click me</Button>
</StackPanel>
```

代码隐藏中事件处理方法的签名必须与路由事件的委托类型匹配。[Click](#) 事件的 [RoutedEventHandler](#) 委托的 `sender` 参数指定事件处理程序附加到的元素。

[RoutedEventHandler](#) 委托的 `args` 参数包含事件数据。[Button\\_Click](#) 事件处理程序的兼容代码隐藏实现可能是：

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Click event logic.
}
```

尽管 `RoutedEventHandler` 是基本的路由事件处理程序委托，但某些控件或实现方案需要不同的委托来支持更专用的事件数据。例如，对于 `DragEnter` 路由事件，处理程序应实现 `DragEventHandler` 委托。通过执行此操作，处理程序代码可以访问事件数据中的 `DragEventArgs.Data` 属性，其中包含来自拖动操作的剪贴板有效负载。

用于添加路由事件处理程序的 XAML 语法与标准 CLR 事件处理程序的语法相同。有关在 XAML 中添加事件处理程序的详细信息，请参阅 [WPF 中的 XAML](#)。有关如何使用 XAML 向元素中添加事件处理程序的完整示例，请参阅[如何处理路由事件](#)。

若要使用代码将路由事件的事件处理程序附加到元素，通常有两个选择：

- 直接调用 `AddHandler` 方法。始终可以通过这种方式附加路由事件处理程序。下面的示例使用 `AddHandler` 方法将 `Click` 事件处理程序附加到按钮：

C#

```
Button1.AddHandler(ButtonBase.ClickEvent, new
RoutedEventHandler(Button_Click));
```

将按钮的 `Click` 事件的处理程序附加到事件路由中的不同元素，例如名为 `StackPanel1` 的 `StackPanel`：

C#

```
StackPanel1.AddHandler(ButtonBase.ClickEvent, new
RoutedEventHandler(Button_Click));
```

- 如果路由事件实现 CLR 事件包装器，请使用特定于语言的事件语法添加事件处理程序，就像对标准 CLR 事件的操作一样。大多数现有的 WPF 路由事件实现 CLR 包装器，从而实现了特定于语言的事件语法。此示例使用特定于语言的语法将 `Click` 事件处理程序附加到按钮：

C#

```
Button1.Click += Button_Click;
```

有关如何在代码中附加事件处理程序的示例，请参阅[如何使用代码添加事件处理程序](#)。如果是使用 Visual Basic 编码，则还可以使用 `Handles` 关键字，将处理程序添加到处理程

序声明中。有关详细信息，请参阅 [Visual Basic 和 WPF 事件处理](#)。

## “已处理”概念

所有路由事件都有一个共同的事件数据基类，即 [RoutedEventArgs](#) 类。[RoutedEventArgs](#) 类定义布尔 [Handled](#) 属性。[Handled](#) 属性的目的在于，允许事件路由线路中的任何事件处理程序将路由事件标记为“已处理”。若要将事件标记为已处理，请在事件处理程序代码中将 [Handled](#) 的值设置为 `true`。

[Handled](#) 的值会影响路由事件在沿事件路由传播过程中的处理方式。如果在路由事件的共享事件数据中 [Handled](#) 为 `true`，则通常不会为该特定事件实例调用附加到事件路由中其他元素的处理程序。在最常见的处理程序方案中，将事件标记为已处理，会有效地阻止事件路由中的后续处理程序（无论是实例处理程序还是类处理程序）响应该特定事件实例。但是，在极少数情况下，需要事件处理程序来响应已标记为已处理的路由事件，可以执行以下操作：

- 使用 [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 重载在代码隐藏中附加处理程序，将 [handledEventsToo](#) 参数设置为 `true`。
- 将 [EventSetter](#) 中的 [HandledEventsToo](#) 属性设置为 `true`。

[Handled](#) 的概念可能会影响你对应用程序和的设计和对事件处理程序的编码。可以将 [Handled](#) 概念化为用于处理路由事件的一个简单协议。你如何使用此协议由你决定，但 [Handled](#) 参数的预期用法是：

- 如果路由事件标记为“已处理”，则它不必由该路由中的其他元素再次处理。
- 如果路由事件未标记为已处理，事件路由中较早的侦听器就没有该事件的处理程序，或者没有已注册的处理程序对该事件的响应可以证明将该事件标记为已处理。当前侦听器上的处理程序有三个可能的操作过程：
  - 根本不执行任何操作。该事件保持未处理状态，并路由到树中的下一个侦听器。
  - 运行代码以响应事件，但不能达到可证明将事件标记为已处理的程度。该事件保持未处理状态，并路由到树中的下一个侦听器。
  - 运行代码以响应事件，并且达到可证明将事件标记为已处理的程度。在事件数据中将事件标记为已处理。事件仍路由到树中的下一个侦听器，但大多数侦听器将不会调用更多处理程序。例外情况是具有专门注册的处理程序的侦听器，它们的 [handledEventsToo](#) 设置为 `true`。

若要详细了解如何处理路由事件，请参阅[将路由事件标记为“已处理”和“类处理”](#)。

尽管只针对引发浮升路由事件的对象来处理该事件的开发人员可能不关心其他侦听器，但最好还是将事件标记为已处理。这样做是为了防止事件路由路线上更远的元素在具有相同路由事件的处理程序时产生意想不到的副作用。

## 类处理程序

路由事件处理程序可以是实例处理程序或类处理程序。给定类的类处理程序会在任何实例处理程序对该类的任何实例响应相同事件之前进行调用。由于此行为，当路由事件标记为已处理时，它们通常会在类处理程序中标记为这样。有两种类型的类处理程序：

- **静态类事件处理程序**，通过在静态类构造函数中调用 `RegisterClassHandler` 方法进行注册。
- **重写类事件处理程序**，通过重写基类虚拟事件方法进行注册。基类虚拟事件方法的存在主要是用于输入事件，名称以 `On<事件名称>` 和 `OnPreview<事件名称>` 开头。

有些 WPF 控件对某些路由事件具有固有的类处理。类处理可能看起来从未引发过路由事件，但实际上它被标记为由类处理程序处理。如果需要事件处理程序来响应已处理的事件，可以注册处理程序，并将 `handledEventsToo` 设置为 `true`。若要详细了解如何实现自己的类处理程序或解决不需要的类处理，请参阅[将路由事件标记为“已处理”和“类处理”](#)。

## WPF 中的附加事件

XAML 语言还定义了一个名为附加事件的特殊类型的事件。附加事件可用于在非元素类中定义新的路由事件，并在树中的任何元素上引发该事件。为此，必须将附加事件注册为路由事件，并提供支持附加事件功能的特定[支持代码](#)。由于附加事件注册为路由事件，因此在元素上引发时，它们会通过元素树传播。

在 XAML 语法中，附加事件由其事件名称及其所有者类型指定，格式为 `<owner type>. <event name>`。因为事件名称是使用具有其所有者类型的名称**限定的**，所以语法允许将该事件附加到可以实例化的任何元素。此语法也适用于附加到沿事件路由的任意元素的常规路由事件的处理程序。还可以通过在处理程序应附加到的对象上调用 `AddHandler` 方法，在代码隐藏中为附加事件附加处理程序。

WPF 输入系统广泛使用附加事件。但是，几乎所有附加事件都通过基本元素显示为等效的非附加路由事件。你很少会直接使用或处理附加事件。例如，与在 XAML 或代码隐藏中使用附加事件语法相比，通过等效 `UIElement.MouseDown` 路由事件处理 `UIElement` 上的基础附加 `Mouse.MouseDown` 事件更为容易。

有关 WPF 中附加事件的详细信息，请参阅[附加事件概述](#)。

# XAML 中的限定事件名称

`<owner type>.<event name>` 语法使用其所有者类型的名称限定事件名称。此语法允许将事件附加到任何元素，而不仅仅是将事件作为其类的成员实现的元素。在 XAML 中为[附加事件](#)或沿事件路由的任意元素上的路由事件附加处理程序时，该语法适用。考虑一下这样的场景：你想要将处理程序附加到父元素，以便处理子元素上引发的路由事件。如果父元素没有作为成员的路由事件，你将需要使用限定的事件名称语法。例如：

XAML

```
<StackPanel Name="StackPanel1" Button.Click="Button_Click">
    <Button>Click me</Button>
</StackPanel>
```

在此示例中，向其添加事件处理程序的父元素侦听器是 `StackPanel`。但是，`Click` 路由事件是在 `ButtonBase` 类上实现和引发的，并通过继承提供给 `Button` 类。尽管 `Button` 类“拥有”`Click` 事件，但是路由事件系统允许将任何路由事件的处理程序附加到任何 `UIElement` 或 `ContentElement` 实例侦听器，否则就会有 CLR 事件的处理程序。对于这些限定的事件属性名称来说，默认的 `xmlns` 命名空间通常是默认的 WPF `xmlns` 命名空间，但是还可以为自定义路由事件指定带有前缀的命名空间。有关 `xmlns` 的详细信息，请参阅 [WPF XAML 的 XAML 命名空间和命名空间映射](#)。

## WPF 输入事件

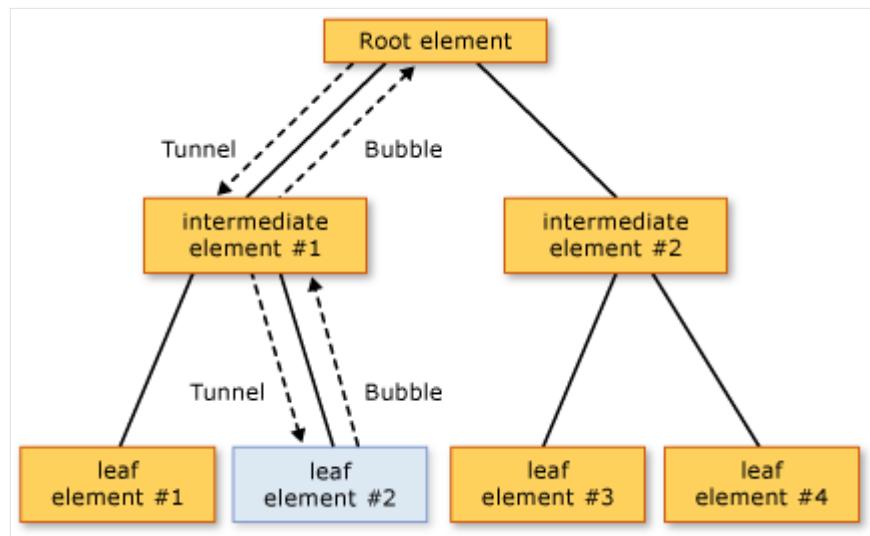
路由事件在 WPF 平台中的常见应用之一是用于[输入事件](#)。按照约定，遵循隧道路由的 WPF 路由事件的名称以“Preview”为前缀。“Preview”前缀表示预览事件在配对浮升事件开始之前完成。输入事件通常成对出现，一个是预览事件，另一个是浮升路由事件。例如，`PreviewKeyDown` 和 `KeyDown`。事件对共享相同的事件数据实例，对于 `PreviewKeyDown` 和 `KeyDown`，类型为 `KeyEventArg`s。有时，输入事件只有浮升版本，或者只有直接路由版本。在 API 文档中，路由事件主题交叉引用路由事件对，并阐明每个路由事件的路由策略。

实现成对出现的 WPF 输入事件，使来自输入设备的单个用户操作（如按鼠标按钮）按顺序引发预览和浮升路由事件。首先引发预览事件并完成其路由。预览事件完成后，将引发浮升事件并完成其路由。引发浮升事件的实现类中的 `RaiseEvent` 方法调用将重复使用来自浮升事件的预览事件中的事件数据。

标记为已处理的预览输入事件不会为预览路由的其余部分调用任何正常注册的事件处理程序，并且不会引发配对的浮升事件。对于希望在其控件的顶层报告基于命中测试的输入事件或基于焦点的输入事件的复合控件设计人员而言，此处理行为非常有用。控件的顶

级元素有机会对控件子组件中的预览事件进行类处理，以便用特定于控件的顶层事件“替换”它们。

为了说明输入事件处理的工作方式，请思考下面的输入事件示例。在下面的树插图中，`leaf element #2` 是 `PreviewMouseDown` 和 `MouseDown` 配对事件的源：



对叶元素 #2 执行鼠标按下操作后的事件处理顺序如下：

1. 根元素上的 `PreviewMouseDown` 隧道事件。
2. 中间元素 #1 上的 `PreviewMouseDown` 隧道事件。
3. 叶元素 #2（源元素）上的 `PreviewMouseDown` 隧道事件。
4. 叶元素 #2（源元素）上的 `MouseDown` 浮升事件。
5. 中间元素 #1 上的 `MouseDown` 浮升事件。
6. 根元素上的 `MouseDown` 浮升事件。

路由事件处理程序委托提供对以下两个对象的引用：引发该事件的对象以及在其中调用处理程序的对象。最初引发了事件的对象由事件数据中的 `Source` 属性报告。在其中调用处理程序的对象是由 `sender` 参数报告的对象。对于任何给定的路由事件实例，引发事件的对象在事件通过元素树时不会更改，但 `sender` 会更改。在上图的步骤 3 和 4 中，`Source` 和 `sender` 是同一对象。

如果输入事件处理程序完成了处理事件所需的特定于应用程序的逻辑，应将输入事件标记为已处理。通常，一旦输入事件被标记为 `Handled`，就不会调用事件路由路线上的后续处理程序。但是，即使事件标记为已处理，也会调用已注册且 `handledEventsToo` 参数设置为 `true` 的输入事件处理程序。有关详细信息，请参阅[预览事件](#)和[将路由事件标记为“已处理”和“类处理”](#)。

预览和浮升事件对的概念（共享事件数据以及依次引发预览事件和浮升事件）仅适用于某些 WPF 输入事件，并不能适用于所有路由事件。如果你要实现自己的输入事件来解决高级方案，请考虑遵循 WPF 输入事件对方法。

如果你要实现自己的可以响应输入事件的复合控件，请考虑使用预览事件来抑制子组件上引发的输入事件，并将其替换为表示完整控件的顶层事件。有关详细信息，请参阅[将路由事件标记为“已处理”和类处理](#)。

有关 WPF 输入系统以及在典型的应用程序方案中输入和事件如何交互的详细信息，请参阅[输入概述](#)。

## EventSetter 和 EventTrigger

在标记样式中，可以使用 [EventSetter](#) 添加预声明的 XAML 事件处理语法。在处理 XAML 时，所引用的处理程序会添加到带样式的实例中。只能针对路由事件声明 [EventSetter](#)。在下面的示例中，引用的 [ApplyButtonStyle](#) 事件处理程序方法在代码隐藏中实现。

XAML

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="{x:Type Button}">
            <EventSetter Event="Click" Handler="ApplyButtonStyle"/>
        </Style>
    </StackPanel.Resources>
    <Button>Click me</Button>
    <Button Click="Button_Click">Click me</Button>
</StackPanel>
```

[Style](#) 节点可能已包含与指定类型的控件相关的其他样式信息，并且使 [EventSetter](#) 成为这些样式的一部分可以提高代码的重用率，即使在标记级别也是如此。此外，与常用的应用程序和页面标记相比，[EventSetter](#) 还提取处理程序的方法名称。

另一个将 WPF 的路由事件和动画功能结合在一起的专用语法是 [EventTrigger](#)。与 [EventSetter](#) 一样，只能针对路由事件声明 [EventTrigger](#)。通常将 [EventTrigger](#) 声明为样式的一部分，但是可以在页面级元素上将 [EventTrigger](#) 声明为 [Triggers](#) 集合的一部分，或者在 [ControlTemplate](#) 中对其进行声明。使用 [EventTrigger](#)，可以指定当路由事件到达其路由中的某个元素（这个元素针对该事件声明了 [EventTrigger](#)）时将运行的 [Storyboard](#)。与只是处理事件并且使其启动现有情节提要相比，[EventTrigger](#) 的优势在于，[EventTrigger](#) 可对情节提要及其运行时行为提供更好的控制。有关详细信息，请参阅[在情节提要启动之后使用事件触发器来控制情节提要](#)。

## 有关路由事件的更多信息

在你自己的类中创建自定义路由事件时，可以从本文中的概念和指南开始入门。还可以使用专用的事件数据类和委托来支持自定义事件。路由事件的所有者可以是任何类，但是路由事件只有由 [UIElement](#) 或 [ContentElement](#) 派生类引发和处理才有用。有关自定义事件的详细信息，请参阅[创建自定义路由事件](#)。

## 另请参阅

- [EventManager](#)
- [RoutedEventArgs](#)
- [RoutedEventArgs](#)
- [将路由事件标记为“已处理”和“类处理”](#)
- [输入概述](#)
- [命令概述](#)
- [自定义依赖属性](#)
- [WPF 中的树](#)
- [弱事件模式](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 附加事件概述 (WPF .NET)

项目 • 2023/10/19

Extensible Application Markup Language (XAML) 定义了一种语言组件和称为附加事件的事件类型。附加事件可用于在非元素类中定义新的[路由事件](#)，并在树中的任何元素上引发该事件。为此，必须将附加事件注册为路由事件，并提供支持附加事件功能的特定[支持代码](#)。由于附加事件注册为路由事件，因此在元素上引发时，它们会通过元素树传播。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你已基本了解 Windows Presentation Foundation (WPF) 路由事件，并已阅读[路由事件概述](#)和[WPF 中的 XAML](#)。若要理解本文中的示例，还应当熟悉 XAML 并知道如何编写 WPF 应用程序。

## 附加事件语法

在 XAML 语法中，附加事件由其事件名称及其所有者类型指定，格式为 `<owner type>. <event name>`。因为事件名称是使用具有其所有者类型的名称限定的，所以语法允许将该事件附加到可以实例化的任何元素。此语法也适用于附加到沿事件路由的任意元素的常规路由事件的处理程序。

以下 XAML 属性语法将 `AquariumFilter.Clean` 附加事件的 `AquariumFilter_Clean` 处理程序附加到 `aquarium1` 元素：

XAML

```
<aqua: Aquarium x:Name="aquarium1" Height="300" Width="400"
aqua: AquariumFilter.Clean="AquariumFilter_Clean"/>
```

在此示例中，`aqua:` 前缀是必需的，因为 `AquariumFilter` 和 `Aquarium` 类存在于不同的公共语言运行时 (CLR) 命名空间和程序集中。

还可以在代码隐藏中附加已附加事件的处理程序。为此，请在处理程序应附加到的对象上调用 `AddHandler` 方法，并将事件标识符和处理程序作为参数传递给此方法。

# WPF 如何实现附加事件

WPF 附加事件作为由 [RoutedEventArgs](#) 字段支持的路由事件实现。因此，附加事件在引发后会通过元素树传播。通常，引发附加事件的对象（称为事件源）是系统或服务源。系统或服务源不是元素树的直接部分。对于其他附加事件，事件源可能是树中的元素，例如复合控件中的组件。

## 附加事件方案

在 WPF 中，附加事件用于具有服务级别抽象的某些功能区域。例如，WPF 使用由静态 [Mouse](#) 或 [Validation](#) 类启用的附加事件。与服务交互或使用服务的类可以使用附加事件语法与事件交互，或者将附加事件显示为路由事件。后一个选项是类如何集成服务功能的一部分。

WPF 输入系统广泛使用附加事件。但是，几乎所有附加事件都通过基本元素显示为等效的非附加路由事件。每个路由输入事件都是基本元素类的一个成员，并使用 CLR 事件“包装器”提供支持。你很少会直接使用或处理附加事件。例如，与在 XAML 或代码隐藏中使用附加事件语法相比，通过等效 [UIElement.MouseDown](#) 路由事件处理 [UIElement](#) 上的基础附加 [Mouse.MouseDown](#) 事件更为容易。

附加事件通过启用输入设备的未来扩展来服务于体系结构目的。例如，新的输入设备只需引发 [Mouse.MouseDown](#) 即可模拟鼠标输入，并且无需从 [Mouse](#) 派生即可执行此操作。此方案会涉及事件的代码处理，而附加事件的 XAML 处理则与此方案无关。

## 处理附加事件

编码和处理附加事件的过程与非附加路由事件的基本相同。

如[前文](#)所述，现有的 WPF 附加事件通常不是专门用于在 WPF 中进行直接处理。通常，附加事件的用途是使复合控件中的元素能够向控件中的父元素报告其状态。在这种情况下，事件在代码中引发，并依赖于相关父类中的类处理。例如，[Selector](#) 中的项应引发 [Selected](#) 附加事件，该事件随后由 [Selector](#) 类进行类处理。[Selector](#) 类可能将 [Selected](#) 事件转换为 [SelectionChanged](#) 路由事件。有关路由事件和类处理的详细信息，请参阅[将路由事件标记为“已处理”和“类处理”](#)。

## 定义自定义附加事件

如果从常见的 WPF 基类派生，可以通过在类中包含两个访问器方法来实现自定义附加事件。这些方法包括：

- Add<事件名称>Handler 方法，其中第一个参数是附加事件处理程序的元素，第二个参数是要添加的事件处理程序。方法必须是 `public` 和 `static`，没有返回值。该方法调用 `AddHandler` 基类方法，将路由事件和处理程序作为参数传入。此方法支持 XAML 属性语法，用于将事件处理程序附加到元素。此方法还可实现对附加事件的事件处理程序存储的代码访问。
- Remove<事件名称>Handler 方法，其中第一个参数是附加事件处理程序的元素，第二个参数是要移除的事件处理程序。方法必须是 `public` 和 `static`，没有返回值。该方法调用 `RemoveHandler` 基类方法，将路由事件和处理程序作为参数传入。此方法允许代码访问附加事件的事件处理程序存储。

WPF 将附加事件作为路由事件实现，因为 `RoutedEventArgs` 的标识符是由 WPF 事件系统定义的。另外，路由一个事件也是对附加事件的 XAML 语言级概念的自然扩展。此实现策略将附加事件的处理限制为 `UIElement` 派生类或 `ContentElement` 派生类，因为只有这些类才具有 `AddHandler` 实现。

例如，以下代码定义了 `AquariumFilter` 所有者类（不是元素类）上的 `Clean` 附加事件。代码将附加事件定义为路由事件，并实现所需的访问器方法。

C#

```
public class AquariumFilter
{
    // Register a custom routed event using the bubble routing strategy.
    public static readonly RoutedEvent CleanEvent =
EventManager.RegisterRoutedEvent(
    "Clean", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
    typeof(AquariumFilter));

    // Provide an add handler accessor method for the Clean event.
    public static void AddCleanHandler(DependencyObject dependencyObject,
RoutedEventHandler handler)
    {
        if (dependencyObject is not UIElement uiElement)
            return;

        uiElement.AddHandler(CleanEvent, handler);
    }

    // Provide a remove handler accessor method for the Clean event.
    public static void RemoveCleanHandler(DependencyObject dependencyObject,
RoutedEventHandler handler)
    {
        if (dependencyObject is not UIElement uiElement)
            return;

        uiElement.RemoveHandler(CleanEvent, handler);
    }
}
```

```
    }  
}
```

返回附加事件标识符的 [RegisterRoutedEvent](#) 方法与用于注册非附加路由事件的方法相同。附加和非附加路由事件均已注册到集中式内部存储。此事件存储实现启用了[路由事件概述](#)中介绍的“事件即界面”概念。

与用于支持非附加路由事件的 CLR 事件“包装器”不同，附加事件访问器方法可以在并非派生自 [UIElement](#) 或 [ContentElement](#) 的类中实现。这很可能是因为附加事件支持代码调用被传递到 [UIElement](#) 实例上的 [UIElement.AddHandler](#) 和 [UIElement.RemoveHandler](#) 方法。相比之下，非附加路由事件的 CLR 包装器直接在所属类上调用这些方法，因此该类必须派生自 [UIElement](#)。

## 引发 WPF 附加事件

引发附加事件的过程实质上与引发非附加路由事件的过程相同。

通常，代码不需要引发任何现有的 WPF 定义的附加事件，因为这些事件遵循常规的“服务”概念模型。在该模型中，服务类（如 [InputManager](#)）负责引发 WPF 定义的附加事件。

当使用 WPF 基于[路由事件](#)的附加事件的 WPF 模型定义自定义附加事件时，使用 [UIElement.RaiseEvent](#) 方法即可在任何 [UIElement](#) 或 [ContentElement](#) 上引发附加事件。引发路由事件时，无论它是否附加，都需要将元素树中的元素指定为事件源。然后，该源将报告为 [RaiseEvent](#) 调用方。例如，要在 [aquarium1](#) 上引发 [AquariumFilter.Clean](#) 附加路由事件：

```
C#
```

```
aquarium1.RaiseEvent(new RoutedEventArgs(AquariumFilter.CleanEvent));
```

在上述示例中，[aquarium1](#) 是事件源。

## 另请参阅

- [路由事件概述](#)
- [XAML 语法详述](#)
- [XAML 及 WPF 的自定义类](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 对象生存期事件 (WPF .NET)

项目 • 2023/10/19

在所有对象的生存期内，Microsoft .NET 托管代码中的所有对象都会经历“创建”、“使用”和“销毁”的阶段。当关于这些阶段的通知出现在对象上时，Windows Presentation Foundation (WPF) 会通过引发生存期事件来进行提供。对于 WPF 框架级元素（视觉对象），WPF 会实现 `Initialized`、`Loaded` 和 `Unloaded` 生存期事件。开发人员可以将这些生存期事件用作涉及元素的代码隐藏操作的挂钩。本文先介绍视觉对象的生存期事件，然后介绍专门应用于窗口元素、导航宿主或应用程序对象的其他生存期事件。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你已基本了解如何将 WPF 元素布局概念化为树，并且你已经阅读过[路由事件概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 视觉对象的生存期事件

WPF 框架级元素派生自 `FrameworkElement` 或 `FrameworkContentElement`。`Initialized`、`Loaded` 和 `Unloaded` 生存期事件是所有 WPF 框架级别元素所通用的。以下示例演示主要在 XAML 中实现的元素树。XAML 定义一个父 `Canvas` 元素，其中包含嵌套元素，每个元素都使用 XAML 属性语法附加 `Initialized`、`Loaded` 和 `Unloaded` 生存期事件处理程序。

XAML

```
<Canvas x:Name="canvas">
    <StackPanel x:Name="outerStackPanel" Initialized="InitHandler"
    Loaded="LoadHandler" Unloaded="UnloadHandler">
        <custom:ComponentWrapper x:Name="componentWrapper"
    Initialized="InitHandler" Loaded="LoadHandler" Unloaded="UnloadHandler">
            <TextBox Name="textBox1" Initialized="InitHandler"
    Loaded="LoadHandler" Unloaded="UnloadHandler" />
            <TextBox Name="textBox2" Initialized="InitHandler"
    Loaded="LoadHandler" Unloaded="UnloadHandler" />
        </custom:ComponentWrapper>
    </StackPanel>
```

```
<Button Content="Remove canvas child elements" Click="Button_Click"/>
</Canvas>
```

其中一个 XAML 元素是自定义控件，它派生自在代码隐藏中分配生存期事件处理程序的基类。

C#

```
public partial class MainWindow : Window
{
    public MainWindow() => InitializeComponent();

    // Handler for the Initialized lifetime event (attached in XAML).
    private void InitHandler(object sender, System.EventArgs e) =>
        Debug.WriteLine($"Initialized event on
{((FrameworkElement)sender).Name}.");

    // Handler for the Loaded lifetime event (attached in XAML).
    private void LoadHandler(object sender, RoutedEventArgs e) =>
        Debug.WriteLine($"Loaded event on
{((FrameworkElement)sender).Name}.");

    // Handler for the Unloaded lifetime event (attached in XAML).
    private void UnloadHandler(object sender, RoutedEventArgs e) =>
        Debug.WriteLine($"Unloaded event on
{((FrameworkElement)sender).Name}.");

    // Remove nested controls.
    private void Button_Click(object sender, RoutedEventArgs e) =>
        canvas.Children.Clear();
}

// Custom control.
public class ComponentWrapper : ComponentWrapperBase { }

// Custom base control.
public class ComponentWrapperBase : StackPanel
{
    public ComponentWrapperBase()
    {
        // Assign handler for the Initialized lifetime event (attached in
        // code-behind).
        Initialized += (object sender, System.EventArgs e) =>
            Debug.WriteLine($"Initialized event on componentWrapperBase.");

        // Assign handler for the Loaded lifetime event (attached in code-
        // behind).
        Loaded += (object sender, RoutedEventArgs e) =>
            Debug.WriteLine($"Loaded event on componentWrapperBase.");

        // Assign handler for the Unloaded lifetime event (attached in code-
        // behind).
        Unloaded += (object sender, RoutedEventArgs e) =>
```

```
        Debug.WriteLine($"Unloaded event on componentWrapperBase.");
    }

}

/* Output:
Initialized event on textBox1.
Initialized event on textBox2.
Initialized event on componentWrapperBase.
Initialized event on componentWrapper.
Initialized event on outerStackPanel.

Loaded event on outerStackPanel.
Loaded event on componentWrapperBase.
Loaded event on componentWrapper.
Loaded event on textBox1.
Loaded event on textBox2.

Unloaded event on outerStackPanel.
Unloaded event on componentWrapperBase.
Unloaded event on componentWrapper.
Unloaded event on textBox1.
Unloaded event on textBox2.
*/
```

程序输出显示在每个树对象上调用 `Initialized`、`Loaded` 和 `Unloaded` 生存期事件的顺序。以下各节按在每个树对象上引发这些事件的顺序对其进行介绍。

## 初始化的生存期事件

在以下情况下，WPF 事件系统会在元素上引发 `Initialized` 事件：

- 设置元素的属性时。
- 大约在同一时间通过调用对象构造函数对其进行初始化。

某些元素属性（如 `Panel.Children`）可以包含子元素。父元素在初始化其子元素之前无法报告初始化。因此，从元素树中嵌套最深的元素开始设置属性值，后跟连续父元素，一直到应用程序根。由于在设置元素的属性时发生 `Initialized` 事件，因此首先在标记中定义的嵌套最深的元素上调用该事件，后跟连续父元素，一直到应用程序根。在代码隐藏中动态创建对象时，其初始化可能不按顺序进行。

WPF 事件系统不会等待元素树中的所有元素都完成初始化，然后再对元素引发 `Initialized` 事件。因此，在为任何元素编写 `Initialized` 事件处理程序时，请记住，逻辑树或可视化树中的周围元素（尤其是父元素）可能尚未创建。或者，其成员变量和数据绑定可能未初始化。

### ① 备注

在元素上引发 `Initialized` 事件时，将取消计算元素的表达式用法，例如动态资源或绑定。

## 加载的生存期事件

在以下情况下，WPF 事件系统会在元素上引发 `Loaded` 事件：

- 当包含该元素的逻辑树完成并连接到演示文稿源时。 演示源提供窗口句柄 (HWND) 和呈现图面。
- 当数据绑定到本地源（例如其他属性或直接定义的数据源）完成时。
- 在布局系统已计算呈现所需的所有值后。
- 在最终呈现之前。

在加载逻辑树中的所有元素之前，不会在元素树中的任何元素上引发 `Loaded` 事件。 WPF 事件系统首先在元素树的根元素上引发 `Loaded` 事件，然后在每个连续的子元素上向下引发嵌套最深的元素。 尽管此事件可能类似于隧道路由事件，但 `Loaded` 事件不会将事件数据从一个元素传输到另一个元素，因此将事件标记为已处理没有效果。

### ① 备注

WPF 事件系统无法保证异步数据绑定在 `Loaded` 事件之前已经完成。 异步数据绑定会绑定到外部或动态源。

## 卸载的生存期事件

在以下情况下，WPF 事件系统会在元素上引发 `Unloaded` 事件：

- 删除其演示文稿源时，或
- 删除其视觉对象父级时。

WPF 事件系统首先在元素树的根元素上引发 `Unloaded` 事件，然后在每个连续的子元素上向下引发嵌套最深的元素。 尽管此事件可能类似于隧道路由事件，但 `Unloaded` 事件不会将事件数据在元素间传播，因此将事件标记为已处理没有效果。

在 `Unloaded` 元素上引发事件时，它的父元素或逻辑树或可视化树中更高级的元素可能已取消设置。 取消设置意味着元素的数据绑定、资源引用和样式不再设置为其正常或上次已知运行时值。

## 其他生存期事件

从生存期事件角度来看，主要有四种 WPF 对象类型：常规元素、窗口元素、导航宿主和应用程序对象。[Initialized](#)、[Loaded](#) 和 [Unloaded](#) 生存期事件适用于所有框架级元素。其他生存期事件专门应用于窗口元素、导航宿主或应用程序对象。有关这些其他生存期事件的信息，请参阅：

- [Application](#) 对象的[应用程序管理概述](#)。
- [Window](#) 元素的[WPF 窗口概述](#)。
- [Page](#)、[NavigationWindow](#) 和 [Frame](#) 元素的[导航概述](#)。

## 另请参阅

- [Initialized](#)
- [Loaded](#)
- [Unloaded](#)
- [处理 Loaded 事件](#)
- [加载的事件和初始化的事件](#)
- [WPF 中的树](#)
- [路由事件概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 将路由事件标记为已处理和类处理 (WPF .NET)

项目 • 2023/10/13

尽管对于何时将路由事件标记为已处理没有绝对规则，但如果代码以重要方式响应事件，请考虑将事件标记为已处理。标记为已处理的路由事件会继续进行其路由，但只会调用配置为响应已处理事件的处理程序。基本上，将路由事件标记为已处理会限制其在事件路上对侦听器的可见性。

路由事件处理程序可以是实例处理程序或类处理程序。实例处理程序处理对象或 XAML 元素上的路由事件。类处理程序在类级别处理路由事件，会在任何实例处理程序对类的任何实例响应相同事件之前进行调用。当路由事件标记为已处理时，它们通常会在类处理程序中标记为这样。本文讨论了将路由事件标记为已处理的好处和潜在缺陷、不同类型的路由事件和路由事件处理程序以及复合控件中的事件禁止。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 何时将路由事件标记为已处理

通常，只应有一个处理程序为每个路由事件提供重要响应。避免使用路由事件系统跨多个处理程序提供重要响应。构成重要响应的定义是主观的，取决于应用程序。一般准则是：

- 重要响应包括设置焦点、修改公共状态、设置影响视觉表示形式的属性、引发新事件以及完全处理事件。
- 不重要响应包括修改私有状态（而没有视觉或编程影响）、事件日志记录以及检查事件数据而不响应事件。

某些 WPF 控件通过将不需要进一步处理的组件级别事件标记为已处理来禁止这些事件。如果要处理已由控件标记为已处理的事件，请参阅[通过控件解决事件禁止问题](#)。

若要将事件标记为已处理，请在其事件数据中将 `Handled` 属性值设置为 `true`。 尽管可以将该值还原到 `false`，但很少需要这样做。

## 预览和浮升路由事件对

预览和浮升路由事件对特定于输入事件。 多个输入事件实现隧道 和 浮升 路由事件对，例如 `PreviewKeyDown` 和 `KeyDown`。`Preview` 前缀表示一旦预览事件完成，浮升事件便会启动。 每个预览和浮升事件对会共享事件数据的相同实例。

路由事件处理程序按对应于事件路由策略的顺序进行调用：

1. 预览事件从应用程序根元素向下传递到引发路由事件的元素。 附加到应用程序根元素的预览事件处理程序会首先进行调用，接下来是附加到后续嵌套元素的处理程序。
2. 预览事件完成后，配对的浮升事件会从将路由事件引发的元素传递到应用程序根元素。 附加到引发路由事件的相同元素的浮升事件处理程序会首先进行调用，接下来是附加到后续父元素的处理程序。

配对的预览和浮升事件是声明并引发自身路由事件的多个 WPF 类的内部实现的一部分。 如果没有该类级别内部实现，预览和浮升路由事件会完全独立，不会共享事件数据（无论事件命名如何）。 有关如何在自定义类中实现浮升或隧道输入路由事件的信息，请参阅 [创建自定义路由事件](#)。

由于每个预览和浮升事件对共享事件数据的相同实例，因此如果预览路由事件标记为已处理，则其配对的浮升事件也会进行处理。 如果浮升路由事件标记为已处理，则不会影响配对的预览事件，因为预览事件已完成。 将预览和浮升输入事件对标记为已处理时要小心。 已处理的预览输入事件不会为隧道路由的其余部分调用任何正常注册的事件处理程序，并且不会引发配对的浮升事件。 已处理的浮升输入事件不会为浮升路由的其余部分调用任何正常注册的事件处理程序。

## 实例和类路由事件处理程序

路由事件处理程序可以是实例处理程序或类处理程序。 给定类的类处理程序会在任何实例处理程序对该类的任何实例响应相同事件之前进行调用。 由于此行为，当路由事件标记为已处理时，它们通常会在类处理程序中标记为这样。 有两种类型的类处理程序：

- **静态类事件处理程序**，通过在静态类构造函数中调用 `RegisterClassHandler` 方法进行注册。
- **重写类事件处理程序**，通过重写基类虚拟事件方法进行注册。 基类虚拟事件方法的存在主要是用于输入事件，名称以 `On<事件名称>` 和 `OnPreview<事件名称>` 开头。

# 实例事件处理程序

可以通过直接调用 `AddHandler` 方法，将实例处理程序附加到对象或 XAML 元素。 WPF 路由事件实现使用 `AddHandler` 方法附加事件处理程序的公共语言运行时 (CLR) 事件包装器。由于用于附加事件处理程序的 XAML 特性语法会导致调用 CLR 事件包装器，因此即时是在 XAML 中附加处理程序也会解析为 `AddHandler` 调用。对于已处理的事件：

- 不会调用使用 XAML 特性语法或 `AddHandler` 的公共签名附加的处理程序。
- 会调用在 `handledEventsToo` 参数设置为 `true` 的情况下使用 `AddHandler(RoutedEventArgs, Delegate, Boolean)` 重载附加的处理程序。此重载适用于需要响应已处理的事件的极少数情况。例如，元素树中的某个元素已将事件标记为已处理，但事件路由中的其他元素需要响应已处理的事件。

下面的 XAML 示例将名为 `componentWrapper` 的自定义控件（包装名为 `componentTextBox` 的 `TextBox`）添加到名为 `outerStackPanel` 的 `StackPanel`。`PreviewKeyDown` 事件的实例事件处理程序使用 XAML 特性语法附加到 `componentWrapper`。因此，实例处理程序只会响应由 `componentTextBox` 引发的未经处理的 `PreviewKeyDown` 隧道事件。

XAML

```
<StackPanel Name="outerStackPanel" VerticalAlignment="Center">
    <custom:ComponentWrapper
        x:Name="componentWrapper"
        TextBox.PreviewKeyDown="HandlerInstanceEventInfo"
        HorizontalAlignment="Center">
        <TextBox Name="componentTextBox" Width="200" />
    </custom:ComponentWrapper>
</StackPanel>
```

`MainWindow` 构造函数使用 `UIElement.AddHandler(RoutedEventArgs, Delegate, Boolean)` 重载 (`handledEventsToo` 参数设置为 `true`) 将 `KeyDown` 浮升事件的实例处理程序附加到 `componentWrapper`。因此，实例事件处理程序会响应未经处理和已处理的事件。

C#

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        // Attach an instance handler on componentWrapper that will be
        // invoked by handled KeyDown events.
        componentWrapper.AddHandler(Keyboard.KeyDownEvent, new
        RoutedEventHandler(Handler.InstanceEventInfo),
        handledEventsToo: true);
```

```
}

// The handler attached to componentWrapper in XAML.
public void HandlerInstanceEventArgs(object sender, KeyEventArgs e) =>
    Handler.InstanceEventArgs(sender, e);
}
```

下一部分显示了 `ComponentWrapper` 的代码隐藏实现。

## 静态类事件处理程序

可以通过在类的静态构造函数中调用 `RegisterClassHandler` 方法来附加静态类事件处理程序。类层次结构中的每个类都可以为每个路由事件注册其自己的静态类处理程序。因此，可以在事件路由中的任何给定节点上为相同事件调用多个静态类处理程序。构造事件的事件路由时，每个节点的所有静态类处理程序都会添加到事件路由中。节点上静态类处理程序的调用顺序从派生程度最高的静态类处理程序开始，接下来是来自每个后续基类的静态类处理程序。

使用 `RegisterClassHandler(Type, RoutedEvent, Delegate, Boolean)` 重载

(`handledEventsToo` 参数设置为 `true`) 注册的静态类事件处理程序会响应未经处理和已处理的路由事件。

静态类处理程序通常注册为仅响应未经处理的事件。在这种情况下，如果节点上的派生类处理程序将事件标记为已处理，则不会调用该事件的基类处理程序。在这种情况下，基类处理程序实际上会被派生类处理程序所替换。基类处理程序通常在视觉外观、状态逻辑、输入处理和命令处理等领域中帮助控制设计，因此在替换它们时要谨慎。不将事件标记为已处理的派生类处理程序最终会补充基类处理程序，而不是替换它们。

下面的代码示例演示在前面 XAML 中引用的 `ComponentWrapper` 自定义控件的类层次结构。`ComponentWrapper` 类从 `ComponentWrapperBase` 派生类，而后者又从 `StackPanel` 类派生。在 `ComponentWrapper` 和 `ComponentWrapperBase` 类的静态构造函数中使用的 `RegisterClassHandler` 方法会为其中每个类注册静态类事件处理程序。WPF 事件系统在 `ComponentWrapperBase` 静态类处理程序之前调用 `ComponentWrapper` 静态类处理程序。

C#

```
public class ComponentWrapper : ComponentWrapperBase
{
    static ComponentWrapper()
    {
        // Class event handler implemented in the static constructor.
        EventManager.RegisterClassHandler(typeof(ComponentWrapper),
KeyEventArgs,
        new RoutedEventHandler(Handler.KeyEventArgs_Static));
    }
}
```

```

// Class event handler that overrides a base class virtual method.
protected override void OnKeyDown(KeyEventEventArgs e)
{
    Handler.ClassEventInfo_Override(this, e);

    // Call the base OnKeyDown implementation on ComponentWrapperBase.
    base.OnKeyDown(e);
}

public class ComponentWrapperBase : StackPanel
{
    // Class event handler implemented in the static constructor.
    static ComponentWrapperBase()
    {
        EventManager.RegisterClassHandler(typeof(ComponentWrapperBase),
KeyEventArgs,
        new RoutedEventHandler(Handler.ClassEventInfoBase_Static));
    }

    // Class event handler that overrides a base class virtual method.
    protected override void OnKeyDown(KeyEventEventArgs e)
    {
        Handler.ClassEventInfoBase_Override(this, e);

        e.Handled = true;
        Debug.WriteLine("The KeyDown routed event is marked as handled.");

        // Call the base OnKeyDown implementation on StackPanel.
        base.OnKeyDown(e);
    }
}

```

下一部分会讨论此代码示例中的重写类事件处理程序的代码隐藏实现。

## 重写类事件处理程序

某些视觉元素基类会为其每个公共路由输入事件公开空的 <On> 事件名称 和 <OnPreview> 事件名称 虚拟方法。例如，[UIElement](#) 会实现 [OnKeyDown](#) 和 [OnPreviewKeyDown](#) 虚拟事件处理程序以及许多其他事件处理程序。可以重写基类虚拟事件处理程序，以便为派生类实现重写类事件处理程序。例如，可以通过重写 [OnDragEnter](#) 虚拟方法，为任何 [UIElement](#) 派生类中的 [DragEnter](#) 事件添加重写类处理程序。重写基类虚拟方法是比在静态构造函数中注册类处理程序更简单的一种实现类处 理程序的方法。在重写中，可以引发事件、启动特定于类的逻辑以更改实例中的元素属性、将事件标记为已处理或执行其他事件处理逻辑。

与静态类事件处理程序不同，WPF 事件系统仅为类层次结构中派生程度最高的类调用重写类事件处理程序。类层次结构中派生程度最高的类随后可以使用 [base](#) 关键字调用虚拟

方法的基实现。在大多数情况下，无论是否将事件标记为已处理，都应调用基本实现。如果类要求替换基实现（如果有），则应仅省略调用基实现。在重写代码之前还是之后调用基实现取决于实现的性质。

在前面的代码示例中，基类 `OnKeyDown` 虚拟方法在 `ComponentWrapper` 和 `ComponentWrapperBase` 类中进行重写。由于 WPF 事件系统仅调用 `ComponentWrapper.OnKeyDown` 重写类事件处理程序，因此该处理程序使用 `base.OnKeyDown(e)` 调用 `ComponentWrapperBase.OnKeyDown` 重写类事件处理程序，后者进而使用 `base.OnKeyDown(e)` 调用 `StackPanel.OnKeyDown` 虚拟方法。前面的代码示例中的事件顺序为：

1. 附加到 `componentWrapper` 的实例处理程序由 `PreviewKeyDown` 路由事件触发。
2. 附加到 `componentWrapper` 的静态类处理程序由 `KeyDown` 路由事件触发。
3. 附加到 `componentWrapperBase` 的静态类处理程序由 `KeyDown` 路由事件触发。
4. 附加到 `componentWrapper` 的重写类处理程序由 `KeyDown` 路由事件触发。
5. 附加到 `componentWrapperBase` 的重写类处理程序由 `KeyDown` 路由事件触发。
6. `KeyDown` 路由事件标记为已处理。
7. 附加到 `componentWrapper` 的实例处理程序由 `KeyDown` 路由事件触发。处理程序已注册（`handledEventsToo` 参数设置为 `true`）。

## 复合控件中的输入事件禁止

某些复合控件会在组件级别禁止输入事件，以便将它们替换为包含更多信息或暗示更特定行为的自定义高级事件。按照定义，复合控件是由多个实际控件或控件基类组成的。经典示例是将各种鼠标事件转换为 `Click` 路由事件的 `Button` 控件。`Button` 基类是间接派生自 `UIElement` 的 `ButtonBase` 类。控制输入处理所需的大部分事件基础结构在 `UIElement` 级别提供。`UIElement` 会公开多个 `Mouse` 事件，例如 `MouseLeftButtonDown` 和 `MouseRightButtonDown`。`UIElement` 还实现空虚拟方法 `OnMouseLeftButtonDown` 和 `OnMouseRightButtonDown` 作为预注册类处理程序。`ButtonBase` 会重写这些类处理程序，在重写处理程序中将 `Handled` 属性设置 `true` 为并引发 `Click` 事件。大多数侦听器的最终结果是 `MouseLeftButtonDown` 和 `MouseRightButtonDown` 事件会隐藏，而高级 `Click` 事件可见。

## 解决输入事件禁止问题

有时，各个控件内的事件禁止可能会干扰应用程序中的事件处理逻辑。例如，如果应用程序使用 XAML 特性语法在 XAML 根元素上附加 `MouseLeftButtonDown` 事件的处理程序，则不会调用该处理程序，因为 `Button` 控件将 `MouseLeftButtonDown` 事件标记为已处理。如果希望对已处理的路由事件调用应用程序的根的元素，则可以：

- 通过调用 `UIElement.AddHandler(RoutedEvent, Delegate, Boolean)` 方法 (`handledEventsToo` 参数设置为 `true`) 来附加处理程序。此方法需要在获取要附加到的元素的对象引用后，在代码隐藏中附加事件处理程序。
- 如果标记为已处理的事件是浮升输入事件，则附加配对的预览事件的处理程序（如果可用）。例如，如果控件禁止了 `MouseLeftButtonDown` 事件，则可以改为附加 `PreviewMouseLeftButtonDown` 事件的处理程序。此方法仅适用于共享事件数据的预览和浮升输入事件对。请注意不要将 `PreviewMouseLeftButtonDown` 标记为已处理，因为这会完全禁止 `Click` 事件。

有关如何解决输入事件禁止的示例，请参阅[通过控件解决事件禁止问题](#)。

## 另请参阅

- [EventManager](#)
- [预览事件](#)
- [创建自定义路由事件](#)
- [路由事件概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 预览事件 (WPF .NET)

项目 • 2023/10/13

预览事件，也称为隧道事件，是从应用程序根元素向下遍历元素树到引发事件的元素的路由事件。引发事件的元素在事件数据中报告为 [Source](#)。并非所有事件场景都支持或需要预览事件。本文介绍了预览事件存在的位置以及应用程序或组件如何与其交互。有关如何创建预览事件的信息，请参阅[如何创建自定义路由事件](#)。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 预览标记为“已处理”的事件

在事件数据中将预览事件标记为“已处理”时要谨慎。将预览事件标记为已在引发该事件的元素之外的其他元素上处理，可阻止引发该事件的元素处理该事件。有时，将预览事件标记为“已处理”是有意的。例如，复合控件可能会禁止由单个组件引发的事件，并将它们替换为由完整控件引发的事件。控件的自定义事件可以根据组件状态关系提供自定义的事件数据和触发器。

对于[输入事件](#)，事件数据由每个事件的预览和非预览（浮升）等效项共享。如果使用预览事件类处理器将输入事件标记为“已处理”，则通常不会调用浮升输入事件的类处理器。或者，如果使用预览事件实例处理器将事件标记为“已处理”，则通常不会调用浮升输入事件的实例处理器。尽管即使将事件标记为“已处理”，你也可以配置要调用的类和实例处理器，但该处理器配置并不常见。有关类处理及其与预览事件的关系的详细信息，请参阅[将路由事件标记为“已处理”和类处理](#)。

## ① 备注

并非所有预览事件都是**隧道**事件。例如，`PreviewMouseLeftButtonDown` 输入事件通过元素树跟踪向下路由，但它是由路径中的每个 `UIElement` 引发和重新引发的**直接**路由事件。

# 通过控件解决事件禁止问题

一些复合控件在组件级别禁止输入事件，以便将其替换为自定义的高级事件。例如，WPF [ButtonBase](#) 将 [MouseLeftButtonDown](#) 浮升输入事件标记为在其 [OnMouseLeftButtonDown](#) 方法中处理并引发 [Click](#) 事件。[MouseLeftButtonDown](#) 事件及其事件数据仍沿元素树路径继续，但由于该事件在事件数据中标记为 [Handled](#)，因此将仅调用配置为响应“已处理”事件的处理程序。

如果希望由应用程序根目录的其他元素处理标记为“已处理”的路由事件，则可以执行以下操作：

- 通过调用 [UIElement.AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 方法并将参数 [handledEventsToo](#) 设置为 [true](#) 来附加处理程序。此方法需要在获取要附加到的元素的对象引用后，在代码隐藏中附加事件处理程序。
- 如果标记为“已处理”的事件是浮升事件，则附加等效预览事件的处理程序（如果可用）。例如，如果控件禁止了 [MouseLeftButtonDown](#) 事件，则可以改为附加 [PreviewMouseLeftButtonDown](#) 事件的处理程序。此方法仅适用于实现[隧道](#)和浮升路由策略并共享事件数据的基本元素输入事件。

以下示例实现了一个名为 [componentWrapper](#) 的基本自定义控件，其中包含一个 [TextBox](#)。控件被添加到名为 [outerStackPanel](#) 的 [StackPanel](#)。

XAML

```
<StackPanel Name="outerStackPanel"
    VerticalAlignment="Center"
    custom:ComponentWrapper.CustomKey="Handler_PrintEventInfo"
    TextBox.KeyDown="Handler_PrintEventInfo"
    TextBox.PreviewKeyDown="Handler_PrintEventInfo" >
<custom:ComponentWrapper
    x:Name="componentWrapper"
    TextBox.KeyDown="ComponentWrapper_KeyDown"
    custom:ComponentWrapper.CustomKey="Handler_PrintEventInfo"
    HorizontalAlignment="Center">
    <TextBox Name="componentTextBox" Width="200"
    KeyDown="Handler_PrintEventInfo" />
</custom:ComponentWrapper>
</StackPanel>
```

每当发生击键操作时，[componentWrapper](#) 控件都会侦听由其 [TextBox](#) 组件引发的 [KeyDown](#) 浮升事件。在这种情况下，[componentWrapper](#) 控件：

1. 将 [KeyDown](#) 浮升路由事件标记为“已处理”以禁止该事件。因此，只会触发在代码隐藏中配置为响应“已处理”[KeyDown](#) 事件的 [outerStackPanel](#) 处理程序。不会调用在

XAML 中为 `KeyDown` 事件附加的 `outerStackPanel` 处理程序。

2. 引发名为 `CustomKey` 的自定义浮升路由事件，该事件触发 `CustomKey` 事件的 `outerStackPanel` 处理程序。

C#

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();

        // Attach a handler on outerStackPanel that will be invoked by
        // handled KeyDown events.
        outerStackPanel.AddHandler(KeyDownEvent, new
RoutedEventHandler(Handler_PrintEventInfo),
            handledEventsToo: true);
    }

    private void ComponentWrapper_KeyDown(object sender,
System.Windows.Input.KeyEventArgs e)
    {
        Handler_PrintEventInfo(sender, e);

        Debug.WriteLine("KeyDown event marked as handled on
componentWrapper.\r\n" +
            "CustomKey event raised on componentWrapper.");

        // Mark the event as handled.
        e.Handled = true;

        // Raise the custom click event.
        componentWrapper.RaiseCustomRoutedEventArgs();
    }

    private void Handler_PrintEventInfo(object sender,
System.Windows.Input.KeyEventArgs e)
    {
        string senderName = ((FrameworkElement)sender).Name;
        string sourceName = ((FrameworkElement)e.Source).Name;
        string eventName = e.RoutedEvent.Name;
        string handledEventsToo = e.Handled ? " Parameter handledEventsToo
set to true." : "";

        Debug.WriteLine($"Handler attached to {senderName} "
            $"triggered by {eventName} event raised on {sourceName}.
{handledEventsToo}");
    }

    private void Handler_PrintEventInfo(object sender, RoutedEventArgs e)
    {
        string senderName = ((FrameworkElement)sender).Name;
```

```

        string sourceName = ((FrameworkElement)e.Source).Name;
        string eventName = e.RoutedEvent.Name;
        string handledEventsToo = e.Handled ? " Parameter handledEventsToo
set to true." : "";
    }

    Debug.WriteLine($"Handler attached to {senderName} +
        $"triggered by {eventName} event raised on {sourceName}.
{handledEventsToo}");
}

// Debug output:
//
// Handler attached to outerStackPanel triggered by PreviewKeyDown event
raised on componentTextBox.
// Handler attached to componentTextBox triggered by KeyDown event
raised on componentTextBox.
// Handler attached to componentWrapper triggered by KeyDown event
raised on componentTextBox.
// KeyDown event marked as handled on componentWrapper.
// CustomKey event raised on componentWrapper.
// Handler attached to componentWrapper triggered by CustomKey event
raised on componentWrapper.
// Handler attached to outerStackPanel triggered by CustomKey event
raised on componentWrapper.
// Handler attached to outerStackPanel triggered by KeyDown event raised
on componentTextBox. Parameter handledEventsToo set to true.
}

public class ComponentWrapper : StackPanel
{
    // Register a custom routed event using the Bubble routing strategy.
    public static readonly RoutedEvent CustomKeyEvent =
        EventManager.RegisterRoutedEvent(
            name: "CustomKey",
            routingStrategy: RoutingStrategy.Bubble,
            handlerType: typeof(RoutedEventHandler),
            ownerType: typeof(ComponentWrapper));

    // Provide CLR accessors for assigning an event handler.
    public event RoutedEventHandler CustomKey
    {
        add { AddHandler(CustomKeyEvent, value); }
        remove { RemoveHandler(CustomKeyEvent, value); }
    }

    public void RaiseCustomRoutedEventArgs()
    {
        // Create a RoutedEventArgs instance.
        RoutedEventArgs routedEventArgs = new(routedEvent: CustomKeyEvent);

        // Raise the event, which will bubble up through the element tree.
        RaiseEvent(routedEventArgs);
    }
}

```

该示例演示了两种解决方法，用于获取禁止的 `KeyDown` 路由事件以调用附加到 `outerStackPanel` 的事件处理程序：

- 将 `PreviewKeyDown` 事件处理程序附加到 `outerStackPanel`。由于预览输入路由事件先于等效的浮升路由事件，因此示例中的 `PreviewKeyDown` 处理程序在 `KeyDown` 处理程序之前运行，后者通过共享事件数据禁止预览和浮升事件。
- 使用代码隐藏中的 `UIElement.AddHandler(RoutedEvent, Delegate, Boolean)` 方法将 `KeyDown` 事件处理程序附加到 `outerStackPanel`，并将 `handledEventsToo` 参数设置为 `true`。

### ① 备注

将输入事件的预览或非预览等效项标记为“已处理”都是禁止控件组件引发的事件的策略。使用的方法取决于应用程序要求。

## 另请参阅

- [将路由事件标记为“已处理”和“类处理”](#)
- [路由事件概述](#)
- [如何创建自定义路由事件](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 属性更改事件 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 定义几个为响应属性值的更改而引发的事件。该属性通常是依赖项属性。事件本身可以是路由事件，也可以是标准公共语言运行时 (CLR) 事件，具体取决于事件是应通过元素树路由，还是仅在属性发生更改的对象上发生。当属性更改仅与属性值发生更改的对象相关时，后一种方案适用。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖有基本的了解，并且已阅读[路由事件概述](#)。

## 标识属性更改事件

并非所有报告属性更改的事件都通过签名或命名模式显式地标识为属性更改事件。SDK 文档交叉引用事件和属性，并指示事件是否直接与属性值更改相关。

某些事件使用特定于属性更改事件的事件数据类型和委托。例如，[RoutedPropertyChanged](#) 和 [DependencyPropertyChanged](#) 事件都有特定的签名。以下几节中讨论了这些事件类型。

## RoutedPropertyChanged 事件

[RoutedPropertyChanged](#) 事件具有 [RoutedEventArgs<T>](#) 事件数据和 [RoutedEventHandler<T>](#) 委托。事件数据和委托都有泛型类型参数 [T](#)。定义处理程序时，指定更改的属性的实际类型。事件数据包含 [OldValue](#) 和 [NewValue](#) 属性，其运行时类型与更改的属性相同。

名称中的“Routed”部分表示属性更改事件注册为路由事件。属性更改路由事件的优势在于，每当子元素属性发生更改时，父元素都会收到通知。这意味着当控件的任何复合部件的值发生更改时，控件的顶层元素会接收属性更改事件。例如，假设创建一个合并 [RangeBase](#) 控件的控件，例如 [Slider](#)。如果 [Value](#) 属性的值在滑块部分发生更改，你可在父控件（而非该部分）上处理此更改。

避免使用属性更改事件处理程序来验证属性值，因为这不是大多数属性更改事件的设计意图。通常，提供属性更改事件是为了你能够在代码的其他逻辑区域响应值更改。在属性更改事件处理程序内再次更改属性值并不明智，并且可能导致意外的递归，具体取决于处理程序的实现方式。

如果属性是自定义依赖属性，或者处理的是定义了实例化代码的派生类，则 WPF 属性系统有更好的方式来跟踪属性更改。这种方式是使用内置 [CoerceValueCallback](#) 和 [PropertyChangedCallback](#) 属性系统回调。若要深入了解如何使用 WPF 属性系统进行验证和强制转换，请参阅[依赖属性回调和验证](#)和[自定义依赖属性](#)。

## DependencyPropertyChanged 事件

DependencyPropertyChanged 事件具有 [DependencyPropertyChangedEventArgs](#) 事件数据和 [DependencyPropertyChangedEventHandler](#) 委托。这些事件是标准 CLR 事件，而不是路由事件。[DependencyPropertyChangedEventArgs](#) 不是通常的事件数据报告类型，因为它不派生自 [EventArgs](#)，而且它是一个结构，并非一个类。

[DependencyPropertyChanged](#) 事件的一个示例是 [IsMouseCapturedChanged](#)。

[DependencyPropertyChanged](#) 事件比 [RoutedPropertyChanged](#) 事件稍微常见一些。

与 [RoutedPropertyChanged](#) 事件数据类似，[DependencyPropertyChanged](#) 事件数据包含 [OldValue](#) 和 [NewValue](#) 属性。出于[前面提到的原因](#)，请避免使用属性更改事件处理程序再次更改属性值。

## 属性触发器

与属性更改事件密切相关的一个概念是属性触发器。属性触发器是在样式或模板内创建的。通过属性触发器，可以创建基于分配了触发器的属性的值的条件行为。

属性触发器操作的属性必须是依赖属性。它可以是（且通常是）只读依赖属性。如果控件公开的依赖属性的名称以“Is”开头，则表明该属性至少部分设计为属性触发器。采用此命名规则的属性通常是只读的 [Boolean](#) 依赖属性，其属性的主要作用是报告控件状态。如果控件状态影响实时 UI，则该依赖属性是一个属性触发器候选项。

有些属性具有专用属性更改事件。例如，[IsMouseCaptured](#) 具有 [IsMouseCapturedChanged](#) 属性更改事件。[IsMouseCaptured](#) 属性是只读的，其值由输入系统修改。输入系统在每次实时更改时都将引发 [IsMouseCapturedChanged](#) 事件。

## 属性触发器限制

与真正的属性更改事件相比，属性触发器具有一些限制。

属性触发器通过完全匹配逻辑来工作，在该逻辑中指定将激活触发器的属性名称和特定值。示例为 `<Setter Property="IsMouseCaptured" Value="true"> ... </Setter>`。属性触发器语法限制大多数属性触发器用于 Boolean 属性或采用专用枚举值的属性。可能值的范围必须可管理，这样你才能为每种情况定义一个触发器。有时，属性触发器仅针对特殊值存在，例如当项计数达到零时。单个触发器不能设置为在属性值偏离特定值（如零）时激活。请考虑实现代码事件处理程序，或实现在值不为零时从触发器状态切换回来的默认行为，而不是针对所有非零情况使用多个触发器。

属性触发器语法与编程中的“if”语句类似。如果触发器条件为 true，将“运行”属性触发器的“主体”。属性触发器的“主体”是标记，而不是代码。该标记被限制为只能使用一个或多个 `Setter` 元素来设置应用了样式或模板的对象的其他属性。

当属性触发器的“if”条件具有各种可能值时，建议使用触发器外的 `Setter` 将此相同的属性值设置为默认值。这样，当触发器条件为 `true` 时，触发器内的 `setter` 将优先，否则触发器外的 `Setter` 将优先。

对于一个或多个外观属性应基于同一元素的其他属性的状态而更改的情况，属性触发器非常有用。

若要深入了解属性触发器，请参阅[样式设置和模板化](#)。

## 另请参阅

- [路由事件概述](#)
- [依赖属性概述](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# Visual Basic 和 WPF 事件处理 (WPF .NET)

项目 • 2023/10/13

如果使用 Visual Basic .NET 进行编码，则可以使用特定于语言的 `Handles` 关键字将事件处理程序附加到对象。该对象可以是代码隐藏中的实例，也可以是 Extensible Application Markup Language (XAML) 中的元素。`Handles` 可用于为公共语言运行时 (CLR) 事件或 Windows Presentation Foundation (WPF) 路由事件分配事件处理程序。但是，当用于为路由事件附加事件处理程序时，`Handles` 存在一些使用限制。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 语法

使用 `Handles` 关键字的 `Sub` 声明的语法为：`Sub <procedure name> Handles <object name>.<event name>`。该语法将一个过程指定为事件处理程序，当 `<event name>` 指定的事件在 `<object name>` 指定的对象上引发时，该过程将运行。该事件必须是对象类或基类的成员。以下示例演示如何使用 `Handles` 将事件处理程序附加到 XAML 元素。

VB

```
' Click event handler attached to XamlButton using Handles.
Private Sub XamlButton_Click(sender As Object, e As RoutedEventArgs) Handles
    XamlButton.Click

    ' Handler logic.
    Debug.WriteLine($"Click event handler attached to XamlButton using
Handles.")

End Sub
```

若要将 `Handles` 与在代码隐藏中定义的对象一起使用，通常使用 `WithEvents` 关键字声明该对象。有关 `WithEvents` 用法的详细信息，请参阅这些[示例](#)。WPF 使用 `Friend WithEvents` 自动声明所有 XAML 元素。以下示例演示如何使用 `WithEvents` 声明在代码隐藏中定义的对象。

VB

```
' Declare a new button using WithEvents.  
Dim WithEvents CodeButton As New Button With {  
    .Content = "New button",  
    .Background = Brushes.Yellow  
}  
  
' Click event handler attached to CodeButton using Handles.  
Private Sub CodeButton_Click(sender As Object, e As RoutedEventArgs) Handles  
CodeButton.Click  
  
    ' Handler logic.  
    Debug.WriteLine($"Click event handler attached to CodeButton using  
Handles.")  
  
End Sub
```

若要对多个事件使用相同的处理程序，请用逗号分隔 `<object name>.<event name>` 事件。例如 `Sub Button_Click(sender As Object, e As RoutedEventArgs) Handles  
Button1.Click, Button2.Click`。逗号分隔事件的顺序不重要。

可以使用多个 `Handles` 语句为同一事件分配不同的处理程序。`Handles` 语句的顺序不会决定事件发生时调用处理程序的顺序。

### 💡 提示

若要删除使用 `Handles` 添加的处理程序，请调用 `RemoveHandler`。例如  
`RemoveHandler Button1.Click, AddressOf Button1_Click`。

## 在 WPF 应用程序中使用“Handles”

对于在 XAML 中定义的对象，`Handles` 事件语法 `<object name>.<event name>` 要求表示该对象的 XAML 元素具有 `Name` 或 `x:Name` 属性。但是，XAML 页面根元素不需要名称属性，因此可以使用名称 `Me`。以下示例演示如何使用 `Handles` 将事件处理程序附加到 XAML 页面根元素。

VB

```
' Loaded event handler attached to the XAML page root using Handles.  
Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs) Handles  
Me.Loaded  
  
    ' Handler logic.  
    Debug.WriteLine($"Loaded event handler attached to Window using  
Handles.")  
  
End Sub
```

编译 XAML 页面时，每个带有 `Name` 或 `x:Name` 参数的 XAML 元素都将声明为 `Friend WithEvents`。因此，可以将任何 XAML 元素与 `Handles` 一起使用。

### 💡 提示

Visual Studio IntelliSense 显示可与 `Handles` 一起使用的对象。

无论是使用 `Handles`、XAML 特性语法、`AddHandler` 语句还是 `AddHandler` 方法附加事件处理程序，事件系统行为都是相同的。

### ⚠ 备注

请勿同时使用 XAML 特性和 `Handles` 将同一事件处理程序附加到同一事件，否则每个事件都会调用该事件处理程序两次。

## 限制

`Handles` 关键字具有以下使用限制：

- 如果事件是对象的类或基类的成员，则只能使用 `Handles` 将事件处理程序附加到对象。例如，可以使用 `Handles` 将 `Click` 事件处理程序附加到基类 `ButtonBase` 引发 `Click` 路由事件的按钮。但是，[路由事件](#)的功能之一是遍历元素树，这使得它可以在比引发它的元素更高的级别上侦听和处理 `click` 事件。父元素侦听和处理的路由事件称为“附加事件”。`Handles` 不能用于附加事件，因为其语法不支持在 XAML 元素树中指定不同于引发事件的元素的侦听器。若要为附加事件分配事件处理程序，需要使用 XAML 特性语法或 `AddHandler` 方法。有关附加事件的详细信息，请参阅[附加事件概述](#)和[WPF 中的附加事件](#)。
- `Handles` 语法不支持为 `Handled` 事件调用事件处理程序。若要为 `Handled` 事件调用事件处理程序，请使用 `AddHandler(RoutedEventArgs, Delegate, Boolean)` 方法附加事件

处理程序并将其 `handledEventsToo` 参数设置为 `true`。

## 另请参阅

- [AddHandler](#)
- [将路由事件标记为“已处理”和“类处理”](#)
- [路由事件概述](#)
- [附加事件概述](#)
- [WPF 中的 XAML](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 弱事件模式 (WPF .NET)

项目 • 2023/10/19

在应用程序中，附加到事件源的处理程序可能不会与将处理程序附加到源的侦听器对象一同销毁。这种情况下会导致内存泄漏。Windows Presentation Foundation (WPF) 引入了可用于解决此问题的设计模式。设计模式为特定事件提供专用的管理器类，并在该事件的侦听器上实现接口。此设计模式称为弱事件模式。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 为什么要实现弱事件模式？

对事件的侦听可能会导致内存泄漏。侦听事件的常用技术是使用特定于语言的语法，将处理程序附加到源上的事件。例如，C# 语句 `source.SomeEvent += new SomeEventHandler(MyEventHandler)` 或 VB 语句 `AddHandler source.SomeEvent, AddressOf MyEventHandler`。然而，此技术可创建从事件源到事件侦听器的强引用。除非显式注销事件处理程序，否则侦听器的对象生存期将受到源的对象生存期的影响。在某些情况下，你可能希望通过其他因素（例如，当前是否属于应用程序的可视化树）控制侦听器的对象生存期。每当源的对象生存期超出侦听器的有用对象生存期时，侦听器的存活时间比必要时间要长。在这种情况下，未分配的内存相当于内存泄漏。

弱事件模式旨在解决内存泄漏问题。当侦听器需要注册事件时，都可以使用弱事件模式，但侦听器并不明确知晓事件会在何时注销。当源的对象生存期超过侦听器的有用对象生存期时，也可以使用弱事件模式。在这种情况下，有用与否将由你来决定。弱事件模式允许侦听器注册事件和接收事件，而不会以任何方式影响侦听器的对象生存期特征。实际上，对源的隐式引用并不能确定侦听器是否有资格执行垃圾回收。由于是弱引用，因而引用是对弱事件模式和相关 API 的命名。侦听器可以被垃圾回收或以其他方式销毁，而源可以继续运行，无需保留针对现已销毁的对象的不可回收的处理程序引用。

## 应该由谁实现弱事件模式？

弱事件模式主要与控件作者相关。 控件作者主要负责控件行为和控件包含，以及控件对其所插入的应用程序的影响。 这包括控件对象生存期行为，特别是处理所述的内存泄漏问题。

某些方案本身就适合应用弱事件模式。 此类方案之一是数据绑定。 在数据绑定中，源对象通常独立于作为绑定目标的侦听器对象。 WPF 数据绑定的许多方面已经在事件的实现方式上应用了弱事件模式。

## 如何实现弱事件模式

有四种方法可以实现弱事件模式，每种方法都使用不同的事件管理器。 选择最适合你的方案的事件管理器。

- [现有弱事件管理器](#)：

当要订阅的事件具有对应的 `WeakEventManager`，请使用现有的弱事件管理器。 有关 WPF 附带的弱事件管理器列表，请参阅 `WeakEventManager` 类中的继承层次结构。 由于包含的弱事件管理器有限，可能需要选择其他方法中的一个。

- [通用弱事件管理器](#)：

如果现有的 `WeakEventManager<TEventArgs>` 事件不可用，并且你正在寻找实现弱事件的最简单方法，请使用泛型 `WeakEventManager`。 但是，泛型 `WeakEventManager<TEventArgs>` 比现有或自定义弱事件管理器更低效，因为它使用反射从其名称中发现事件。 此外，使用泛型 `WeakEventManager<TEventArgs>` 注册事件所需的代码比使用现有或自定义 `WeakEventManager` 注册事件所需的代码更详细。

- [自定义弱事件管理器](#)：

在现有 `WeakEventManager` 不可用且效率至关重要时，创建自定义的 `WeakEventManager`。 尽管比泛型 `WeakEventManager` 更有效，但自定义 `WeakEventManager` 要求编写更多前期代码。

- [第三方弱事件管理器](#)：

当需要其他方法未提供的功能时，请使用第三方弱事件管理器。 NuGet 具有一些[较弱的事件管理器](#)。 许多 WPF 框架也支持此模式。 例如，请参阅 [Prism 关于松散耦合事件订阅的文档](#)。

以下部分介绍如何通过使用不同的事件管理器类型来实现弱事件模式。 对于泛型和自定义弱事件管理器示例，要订阅的事件具有以下特征。

- 事件名称为 `SomeEvent`。

- 事件由 `SomeEventSource` 类引发。
- 事件处理程序的类型为 `EventHandler<SomeEventArgs>`。
- 事件将 `SomeEventArgs` 类型的参数传递给事件处理程序。

## 使用现有弱事件管理器类

1. 查找现有弱事件管理器。有关 WPF 附带的弱事件管理器列表，请参阅 [WeakEventManager](#) 类的继承层次结构。
2. 使用新的弱事件管理器，而不是普通事件挂钩。

例如，如果代码使用以下模式订阅事件：

C#

```
source.LostFocus += new RoutedEventHandler(Source_LostFocus);
```

将其更改为以下模式：

C#

```
LostFocusEventManager.AddHandler(source, Source_LostFocus);
```

同样，如果代码使用以下模式取消订阅事件：

C#

```
source.LostFocus -= new RoutedEventHandler(Source_LostFocus);
```

将其更改为以下模式：

C#

```
LostFocusEventManager.RemoveHandler(source, Source_LostFocus);
```

## 使用泛型弱事件管理器类

使用泛型 [WeakEventManager<TEventArgs>](#) 类，而不是普通事件挂钩。

使用 `WeakEventManager<TEventArgs>` 注册事件侦听器时，需要将事件源和 `EventArgs` 类型作为类型参数提供给类。调用 `AddHandler`，如以下代码所示：

C#

```
WeakEventManager<SomeEventSource, SomeEventArgs>.AddHandler(source,
    "SomeEvent", Source_SomeEvent);
```

## 创建自定义弱事件管理器类

- 将以下类模板复制到项目。以下类继承自 [WeakEventManager](#) 类：

C#

```
class SomeEventWeakEventManager : WeakEventManager
{
    private SomeEventWeakEventManager()
    {

        /// <summary>
        /// Add a handler for the given source's event.
        /// </summary>
        public static void AddHandler(SomeEventSource source,
                                      EventHandler<SomeEventArgs> handler)
        {
            if (source == null)
                throw new ArgumentNullException(nameof(source));
            if (handler == null)
                throw new ArgumentNullException(nameof(handler));

            CurrentManager.ProtectedAddHandler(source, handler);
        }

        /// <summary>
        /// Remove a handler for the given source's event.
        /// </summary>
        public static void RemoveHandler(SomeEventSource source,
                                         EventHandler<SomeEventArgs>
handler)
        {
            if (source == null)
                throw new ArgumentNullException(nameof(source));
            if (handler == null)
                throw new ArgumentNullException(nameof(handler));

            CurrentManager.ProtectedRemoveHandler(source, handler);
        }

        /// <summary>
        /// Get the event manager for the current thread.
        /// </summary>
        private static SomeEventWeakEventManager CurrentManager
        {
            get
            {
```

```
Type managerType = typeof(SomeEventWeakEventManager);
SomeEventWeakEventManager manager =
(SomeEventWeakEventManager)GetCurrentManager(managerType);

    // at first use, create and register a new manager
    if (manager == null)
    {
        manager = new SomeEventWeakEventManager();
        SetCurrentManager(managerType, manager);
    }

    return manager;
}

/// <summary>
/// Return a new list to hold listeners to the event.
/// </summary>
protected override ListenerList NewListenerList()
{
    return new ListenerList<SomeEventArgs>();
}

/// <summary>
/// Listen to the given source for the event.
/// </summary>
protected override void StartListening(object source)
{
    SomeEventSource typedSource = (SomeEventSource)source;
    typedSource.SomeEvent += new EventHandler<SomeEventArgs>
(OnSomeEvent);
}

/// <summary>
/// Stop listening to the given source for the event.
/// </summary>
protected override void StopListening(object source)
{
    SomeEventSource typedSource = (SomeEventSource)source;
    typedSource.SomeEvent -= new EventHandler<SomeEventArgs>
(OnSomeEvent);
}

/// <summary>
/// Event handler for the SomeEvent event.
/// </summary>
void OnSomeEvent(object sender, SomeEventArgs e)
{
    DeliverEvent(sender, e);
}
```

2. 重命名 `SomeEventWeakEventManager`、`SomeEvent`、`SomeEventSource` 和 `SomeEventArgs` 以匹配事件名称。

3. 设置弱事件管理器类的访问修饰符，用于匹配其管理的事件的可访问性。

4. 使用新的弱事件管理器，而不是普通事件挂钩。

例如，如果代码使用以下模式订阅事件：

C#

```
source.SomeEvent += new EventHandler<SomeEventArgs>(Source_SomeEvent);
```

将其更改为以下模式：

C#

```
SomeEventWeakEventManager.AddHandler(source, Source_SomeEvent);
```

同样，如果代码使用以下模式取消订阅事件：

C#

```
source.SomeEvent -= new EventHandler<SomeEventArgs>(Source_SomeEvent);
```

将其更改为以下模式：

C#

```
SomeEventWeakEventManager.RemoveHandler(source, Source_SomeEvent);
```

## 另请参阅

- [WeakEventManager](#)
- [IWeakEventListener](#)
- [路由事件概述](#)
- [数据绑定概述](#)



在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问



.NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

题和拉取请求。有关详细信息，  
请参阅[参与者指南](#)。

 提出文档问题

 提供产品反馈

# 如何使用代码添加事件处理程序 (WPF .NET)

项目 • 2023/10/13

可以使用标记或代码隐藏将事件处理程序分配给 Windows Presentation Foundation (WPF) 中的元素。尽管通常在 Extensible Application Markup Language (XAML) 中分配事件处理程序，但有时可能需要在代码隐藏中分配事件处理程序。例如，在以下情况使用代码：

- 在加载包含元素的标记页之后，为元素分配事件处理程序。
- 在加载将包含元素的标记页之后，添加一个元素并为其分配事件处理程序。
- 完全在代码中定义应用程序的元素树。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 事件处理程序分配的语法

C# 支持使用以下方法分配事件处理程序：

- `+=` 运算符，公共语言运行时 (CLR) 事件处理模型中也使用此运算符。
- [UIElement.AddHandler](#) 方法。

VB 支持使用以下方法分配事件处理程序：

- 带 [AddressOf](#) 运算符的 [AddHandler](#) 语句，CLR 事件处理模型中也使用此语句。
- 事件处理程序定义中的 [Handles](#) 关键字。有关详细信息，请参阅 [Visual Basic 和 WPF 事件处理](#)。
- [UIElement.AddHandler](#) 方法，与 [AddressOf](#) 运算符一起引用事件处理程序。

## 示例

以下示例使用 XAML 定义名为 `ButtonCreatedByXaml` 的 `Button` 并分配 `ButtonCreatedByXaml_Click` 方法作为其 `Click` 事件处理程序。`Click` 是派生自 `ButtonBase` 的按钮的内置路由事件。

XAML

```
<StackPanel Name="StackPanel1">
    <Button
        Name="ButtonCreatedByXaml"
        Click="ButtonCreatedByXaml_Click"
        Content="Create a new button with an event handler"
        Background="LightGray">
    </Button>
</StackPanel>
```

此示例使用代码隐藏来实现 `ButtonCreatedByXaml_Click` 和 `ButtonCreatedByCode_Click` 处理程序，并将 `ButtonCreatedByCode_Click` 处理程序分配给 `ButtonCreatedByCode` 和 `StackPanel1` 元素。事件处理程序方法只能在代码隐藏中实现。

C#

```
// The click event handler for the existing button 'ButtonCreatedByXaml'.
private void ButtonCreatedByXaml_Click(object sender, RoutedEventArgs e)
{
    // Create a new button.
    Button ButtonCreatedByCode = new();

    // Specify button properties.
    ButtonCreatedByCode.Name = "ButtonCreatedByCode";
    ButtonCreatedByCode.Content = "New button and event handler created in
code";
    ButtonCreatedByCode.Background = Brushes.Yellow;

    // Add the new button to the StackPanel.
    StackPanel1.Children.Add(ButtonCreatedByCode);

    // Assign an event handler to the new button using the '+=' operator.
    ButtonCreatedByCode.Click += new
    RoutedEventHandler(ButtonCreatedByCode_Click);

    // Assign an event handler to the new button using the AddHandler
    // method.
    // AddHandler(ButtonBase.ClickEvent, new
    RoutedEventHandler(ButtonCreatedByCode_Click));

    // Assign an event handler to the StackPanel using the AddHandler
    // method.
    StackPanel1.AddHandler(ButtonBase.ClickEvent, new
    RoutedEventHandler(ButtonCreatedByCode_Click));
}
```

```
// The Click event handler for the new button 'ButtonCreatedByCode'.
private void ButtonCreatedByCode_Click(object sender, RoutedEventArgs e)
{
    string sourceName = ((FrameworkElement)e.Source).Name;
    string senderName = ((FrameworkElement)sender).Name;

    Debug.WriteLine($"Routed event handler attached to {senderName}, " +
        $"triggered by the Click routed event raised by {sourceName}.");
}
```

单击 `ButtonCreatedByXaml` 并且其事件处理程序 `ButtonCreatedByXaml_Click` 以编程方式运行时：

1. 向已构造的 XAML 元素树添加名为 `ButtonCreatedByCode` 的新按钮。
2. 指定新按钮的属性，例如名称、内容和背景色。
3. 将 `ButtonCreatedByCode_Click` 事件处理程序分配给 `ButtonCreatedByCode`。
4. 将相同的 `ButtonCreatedByCode_Click` 事件处理程序分配给 `StackPanel1`。

单击 `ButtonCreatedByCode` 时：

1. `Click` 路由事件在 `ButtonCreatedByCode` 上引发。
2. 触发分配给 `ButtonCreatedByCode` 的 `ButtonCreatedByCode_Click` 事件处理程序。
3. `Click` 路由事件在元素树中向上遍历到 `StackPanel1`。
4. 触发分配给 `StackPanel1` 的 `ButtonCreatedByCode_Click` 事件处理程序。
5. `Click` 路由事件继续向上遍历元素树，可能会触发分配给其他已遍历元素的其他 `Click` 事件处理程序。

`ButtonCreatedByCode_Click` 事件处理程序获取有关触发它的事件的以下信息：

- `sender` 对象，它是分配了事件处理程序的元素。 处理程序首次运行时，`sender` 为 `ButtonCreatedByCode`，第二次运行时则为 `StackPanel1`。
- `RoutedEventArgs.Source` 对象，它是最初引发事件的元素。 在本示例中，`Source` 始终为 `ButtonCreatedByCode`。

## ① 备注

路由事件和 CLR 事件之间的一个主要区别是，路由事件遍历元素树来查找处理程序，而 CLR 事件不遍历元素树，处理程序只能附加到引发事件的源对象。因此，路由事件 `sender` 可以是元素树中的任何已遍历的元素。

有关如何创建和处理路由事件的详细信息，请参阅[如何创建自定义路由事件](#)和[处理路由事件](#)。

# 另请参阅

- [路由事件概述](#)
- [WPF 中的 XAML](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何创建自定义路由事件 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 应用程序开发人员和组件作者可以创建自定义路由事件，用于扩展公共语言运行时 (CLR) 事件的功能。有关路由事件功能的信息，请参阅[为什么使用路由事件](#)。本文介绍创建自定义路由事件的基本知识。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对路由事件有基本的了解，并且已阅读[路由事件概述](#)。若要遵循本文中的示例，如果熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序，将会很有帮助。

## 路由事件步骤

创建路由事件的基本步骤如下：

1. 使用 [RegisterRoutedEvent](#) 方法注册 [RoutedEventArgs](#)。
2. 注册调用返回一个 [RoutedEventArgs](#) 实例，称为路由事件标识符，该标识符包含已注册的事件名、[路由策略](#)和其他事件详细信息。将该标识符分配给静态只读字段。按照惯例：
  - 具有[浮升策略](#)的路由事件的标识符命名为 `<event name>Event`。例如，如果事件名为 `Tap`，则标识符应命名为 `TapEvent`。
  - 具有[隧道策略](#)的路由事件的标识符命名为 `Preview<event name>Event`。例如，如果事件名为 `Tap`，则标识符应命名为 `PreviewTapEvent`。
3. 定义 CLR `add` 和 `remove` 事件访问器。如果没有 CLR 事件访问器，你就只能通过直接调用 [UIElement.AddHandler](#) 和 [UIElement.RemoveHandler](#) 方法来添加或删除事件处理程序。使用 CLR 事件访问器时，你会获得以下事件处理程序分配机制：
  - 对于 Extensible Application Markup Language (XAML)，可以使用属性语法来添加事件处理程序。
  - 对于 C#，可以使用 `+=` 和 `-=` 运算符来添加或删除事件处理程序。

- 对于 VB，可以使用 `AddHandler` 和 `RemoveHandler` 语句来添加或删除事件处理程序。

4. 添加用于触发路由事件的自定义逻辑。例如，你的逻辑可能会基于用户输入和应用程序状态触发事件。

## 示例

以下示例在自定义控件库中实现 `CustomButton` 类。派生自 `Button` 的 `CustomButton` 类：

- 使用 `RegisterRoutedEvent` 方法注册一个名为 `ConditionalClick` 的 `RoutedEventArgs`，并在注册期间指定浮升策略。
- 将从注册调用返回的 `RoutedEventArgs` 实例分配给名为 `ConditionalClickEvent` 的静态只读字段。
- 定义 CLR `add` 和 `remove` 事件访问器。
- 添加自定义逻辑，以在单击 `CustomButton` 并应用外部条件时引发自定义路由事件。虽然示例代码从重写的 `OnClick` 虚拟方法内引发 `ConditionalClick` 路由事件，但你可选用任何方式来引发事件。

C#

```
public class CustomButton : Button
{
    // Register a custom routed event using the Bubble routing strategy.
    public static readonly RoutedEventArgs ConditionalClickEvent =
        EventManager.RegisterRoutedEvent(
            name: "ConditionalClick",
            routingStrategy: RoutingStrategy.Bubble,
            handlerType: typeof(RoutedEventHandler),
            ownerType: typeof(CustomButton));

    // Provide CLR accessors for assigning an event handler.
    public event RoutedEventHandler ConditionalClick
    {
        add { AddHandler(ConditionalClickEvent, value); }
        remove { RemoveHandler(ConditionalClickEvent, value); }
    }

    void RaiseCustomRoutedEvent()
    {
        // Create a RoutedEventArgs instance.
        RoutedEventArgs routedEventArgs = new(routedEvent:
        ConditionalClickEvent);

        // Raise the event, which will bubble up through the element tree.
        RaiseEvent(routedEventArgs);
    }
}
```

```

// For demo purposes, we use the Click event as a trigger.
protected override void OnClick()
{
    // Some condition combined with the Click event will trigger the
    // ConditionalClick event.
    if (DateTime.Now > new DateTime())
        RaiseCustomRoutedEventArgs();

    // Call the base class OnClick() method so Click event subscribers
    // are notified.
    base.OnClick();
}

```

该示例包含一个单独的 WPF 应用程序，该应用程序使用 XAML 标记将 `CustomButton` 实例添加到 `StackPanel`，并将 `Handler_ConditionalClick` 方法分配为 `CustomButton` 和 `StackPanel1` 元素的 `ConditionalClick` 事件处理程序。

#### XAML

```

<Window x:Class="CodeSample.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:WpfControl;assembly=WpfControlLibrary"
    Title="How to create a custom routed event" Height="100"
    Width="300">

    <StackPanel Name="StackPanel1"
    custom:CustomButton.CConditionalClick="Handler_ConditionalClick">
        <custom:CustomButton
            Name="customButton"
            ConditionalClick="Handler_ConditionalClick"
            Content="Click to trigger a custom routed event"
            Background="LightGray">
        </custom:CustomButton>
    </StackPanel>
</Window>

```

WPF 应用程序在代码隐藏中定义 `Handler_ConditionalClick` 事件处理程序方法。事件处理程序方法只能在代码隐藏中实现。

#### C#

```

// The ConditionalClick event handler.
private void Handler_ConditionalClick(object sender, RoutedEventArgs e)
{
    string senderName = ((FrameworkElement)sender).Name;
    string sourceName = ((FrameworkElement)e.Source).Name;

    Debug.WriteLine($"Routed event handler attached to {senderName}, " +

```

```
$"triggered by the ConditionalClick routed event raised on
{sourceName}.");
}

// Debug output when CustomButton is clicked:
// Routed event handler attached to CustomButton,
//     triggered by the ConditionalClick routed event raised on
CustomButton.
// Routed event handler attached to StackPanel1,
//     triggered by the ConditionalClick routed event raised on
CustomButton.
```

单击 `CustomButton` 时：

1. `ConditionalClick` 路由事件在 `CustomButton` 上引发。
2. 触发了附加到 `CustomButton` 的 `Handler_ConditionalClick` 事件处理程序。
3. `ConditionalClick` 路由事件在元素树中向上遍历到 `StackPanel1`。
4. 触发了附加到 `StackPanel1` 的 `Handler_ConditionalClick` 事件处理程序。
5. `ConditionalClick` 路由事件继续向上遍历元素树，可能会触发附加到其他已遍历元素的其他 `ConditionalClick` 事件处理程序。

`Handler_ConditionalClick` 事件处理程序获取有关触发它的事件的以下信息：

- `sender` 对象，它是事件处理程序附加到的元素。处理程序首次运行时，`sender` 为 `CustomButton`，第二次运行时则为 `StackPanel1`。
- `RoutedEventArgs.Source` 对象，它最初引发事件的元素。在本示例中，`Source` 始终为 `CustomButton`。

### ① 备注

路由事件和 CLR 事件之间的一个主要区别是，路由事件遍历元素树来查找处理程序，而 CLR 事件不遍历元素树，处理程序只能附加到引发事件的源对象。因此，路由事件 `sender` 可以是元素树中的任何已遍历的元素。

你可以像创建浮升事件一样创建隧道事件，但将在 `Tunnel` 事件注册调用中设置路由策略。有关隧道事件的详细信息，请参阅 [WPF 输入事件](#)。

## 另请参阅

- [路由事件概述](#)
- [输入概述](#)
- [控件创作概述](#)
- [处理路由事件](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 依赖属性概述 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 提供一组服务，这些服务可用于扩展类型的属性的功能。这些服务统称为 WPF 属性系统。由 WPF 属性系统提供支持的属性称为依赖属性。本概述文章介绍 WPF 属性系统和依赖属性的功能，包括如何在 XAML 和代码中使用现有的依赖属性。本概述还介绍依赖属性所特有的方面（如依赖属性元数据），并说明如何在自定义类中创建自己的依赖属性。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对 .NET 类型系统和面向对象的编程有基本的了解。若要理解本文中的示例，了解 XAML 并知道如何编写 WPF 应用程序很有帮助。有关详细信息，请参阅[教程：使用 .NET 创建新的 WPF 应用](#)。

## 依赖属性和 CLR 属性

WPF 属性通常公开为标准 .NET 属性。你可能在基本级别与这些属性进行交互，而不必了解它们是以依赖属性的形式实现的。但是，熟悉 WPF 属性系统的部分或全部功能有助于充分利用这些功能。

依赖属性的用途在于提供一种方法来基于其他输入的值计算属性值，例如：

- 系统属性，例如主题和用户首选项。
- 即时属性确定机制，例如数据绑定和动画/情节提要。
- 多用途模板，例如资源和样式。
- 通过与元素树中其他元素的父子关系知道的值。

此外，依赖属性还可以提供：

- 独立验证。
- 默认值。
- 回调，用于监视对其他属性的更改。
- 可以根据运行时信息强制转换属性值的系统。

派生类可以通过替代依赖属性的元数据（而不是替代现有属性的实际实现或创建新属性）来更改现有属性的某些特征。

在 SDK 参考中，可以根据某个属性的托管引用页上是否有“依赖属性信息”部分来确定该属性是否为依赖属性。“依赖属性信息”部分包含指向该依赖属性的 [DependencyProperty](#) 标识符字段的链接。它还包含该属性的元数据选项列表、每个类的替代信息和其他详细信息。

## 依赖属性支持 CLR 属性

依赖属性和 WPF 属性系统通过提供一个支持属性的类型来扩展属性功能，这是使用专用字段支持属性的标准模式的替代方法。此类型的名称为 [DependencyProperty](#)。定义 WPF 属性系统的另一个重要类型是 [DependencyObject](#)，它定义了可以注册和拥有依赖属性的基类。

下面是一些常用的术语：

- **依赖属性**：由 [DependencyProperty](#) 提供支持的属性。
- **依赖属性标识符**：一个 [DependencyProperty](#) 实例，在注册依赖属性时以返回值的形式获取它，之后将其存储为类的静态成员。许多与 WPF 属性系统交互的 API 使用依赖属性标识符作为参数。
- **CLR“包装器”**：属性的 `get` 和 `set` 实现。这些实现通过在 [GetValue](#) 和 [SetValue](#) 调用中使用依赖属性标识符来并入依赖属性标识符。这样，WPF 属性系统就可以为属性提供支持。

以下示例定义 `IsSpinning` 依赖属性，以说明 [DependencyProperty](#) 标识符与它所支持的属性之间的关系。

C#

```
public static readonly DependencyProperty IsSpinningProperty =
DependencyProperty.Register(
    "IsSpinning", typeof(bool),
    typeof(MainWindow)
);

public bool IsSpinning
{
    get => (bool)GetValue(IsSpinningProperty);
    set => SetValue(IsSpinningProperty, value);
}
```

属性及其支持性 [DependencyProperty](#) 字段的命名约定非常重要。字段总是与属性同名，但其后面追加了 `Property` 后缀。有关此约定及其原因的详细信息，请参阅[自定义依赖属性](#)。

## 设置属性值

可以在代码或 XAML 中设置属性。

### 在 XAML 中设置属性值

以下 XAML 示例将按钮的背景色设置为红色。XAML 属性的字符串值类型由 WPF XAML 分析程序转换为 WPF 类型。在生成的代码中，WPF 类型为 `Color` (以 `SolidColorBrush` 的形式)。

XAML

```
<Button Content="I am red" Background="Red"/>
```

XAML 支持多种用于设置属性的语法形式。要对特定的属性使用哪种语法取决于该属性所使用的值类型以及其他因素（例如，是否存在类型转换器）。有关用于设置属性的 XAML 语法的详细信息，请参阅[WPF 中的 XAML](#) 和 [XAML 语法详述](#)。

以下 XAML 示例显示了另一个使用属性元素语法而不是特性语法的按钮背景。XAML 不设置简单的纯色，而是将按钮 `Background` 属性设置为图像。通过元素表示该图像，通过嵌套元素的属性指定图像来源。

XAML

```
<Button Content="I have an image background">
    <Button.Background>
        <ImageBrush ImageSource="stripes.jpg"/>
    </Button.Background>
</Button>
```

### 在代码中设置属性

在代码中设置依赖属性值通常只是调用由 CLR“包装器”公开的 `set` 实现：

C#

```
Button myButton = new();
myButton.Width = 200.0;
```

获取属性值实质上是在调用 `get` “包装器”实现：

C#

```
double whatWidth = myButton.Width;
```

还可以直接调用属性系统 API `GetValue` 和 `SetValue`。直接调用 API 适用于某些方案，但通常不适用于使用现有属性的情况。通常，包装器更方便，并为开发人员工具提供更好的属性公开。

还可以在 XAML 中设置属性，然后通过代码隐藏在代码中访问这些属性。有关详细信息，请参阅 [WPF 中的代码隐藏和 XAML](#)。

## 由依赖属性提供的属性功能

与字段支持的属性不同，依赖属性扩展了属性的功能。通常，添加的功能表示或支持以下功能之一：

- [资源](#)
- [数据绑定](#)
- [样式](#)
- [动画](#)
- [元数据重写](#)
- [属性值继承](#)
- [WPF 设计器集成](#)

## 资源

可以通过引用资源来设置依赖属性值。资源通常指定为页面根元素或应用程序的 `Resources` 属性值，因为通过这些位置可以非常方便地访问资源。在此示例中，我们定义一个 `SolidColorBrush` 资源：

XAML

```
<StackPanel.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</StackPanel.Resources>
```

现在资源已定义，我们可以引用资源来为 `Background` 属性提供值：

#### XAML

```
<Button Background="{DynamicResource MyBrush}" Content="I am gold" />
```

在 WPF XAML 中，可以使用静态或动态资源引用。此特定资源作为 `DynamicResource` 引用。动态资源引用只能用于设置依赖属性，因此它是由 WPF 属性系统明确启用的动态资源引用用法。有关详细信息，请参阅 [XAML 资源](#)。

#### ① 备注

资源被视为本地值，这意味着，如果设置另一个本地值，该资源引用将被消除。有关详细信息，请参阅 [依赖属性值优先级](#)。

## 数据绑定

依赖属性可以通过数据绑定来引用值。数据绑定通过特定标记扩展语法（在 XAML 中）或 `Binding` 对象（在代码中）起作用。使用数据绑定，最终属性值的确定将延迟到运行时，在运行时，将从数据源获取属性值。

以下示例使用在 XAML 中声明的绑定来设置 `Button` 的 `Content` 属性。该绑定使用继承的数据上下文和 `XmlDataProvider` 数据源（未显示）。绑定本身通过 `XPath` 指定数据源中的源属性。

#### XAML

```
<Button Content="{Binding Source={StaticResource TestData},  
XPath=test[1]/@text}"/>
```

#### ① 备注

绑定被视为本地值，这意味着，如果设置另一个本地值，该绑定将被消除。有关详细信息，请参阅 [依赖属性值优先级](#)。

依赖属性或 `DependencyObject` 类本身不支持通过 `INotifyPropertyChanged` 来通知数据绑定操作的 `DependencyObject` 源属性值的更改。有关如何创建要在数据绑定中并且可以向数据绑定目标报告更改的属性的详细信息，请参阅 [数据绑定概述](#)。

## 样式

之所以使用依赖属性，令人心动的原因在于样式和模板。设置定义应用程序 UI 的属性时，样式尤其有用。在 XAML 中，通常将样式定义为资源。样式与属性系统交互，因为它们通常包含特定属性的“资源库”，以及基于另一个属性的运行时值更改属性值的“触发器”。

以下示例创建一个简单样式，该样式在 [Resources](#) 字典（未显示）内定义。然后将该样式直接应用于 [Button](#) 的 [Style](#) 属性。样式中的资源库将带样式 [Button](#) 的 [Background](#) 属性设置为绿色。

XAML

```
<Style x:Key="GreenButtonStyle">
    <Setter Property="Control.Background" Value="Green"/>
</Style>
```

XAML

```
<Button Style="{StaticResource GreenButtonStyle}" Content="I am green"/>
```

有关详细信息，请参阅[样式设置和模板化](#)。

## 动画

可以对依赖属性进行动画处理。当应用的动画运行时，动画值的优先级高于任何其他属性值，包括本地值。

以下示例对 [Button](#) 的 [Background](#) 属性进行动画处理。从技术上说，属性元素语法将空白 [SolidColorBrush](#) 设置为 [Background](#)，并对 [SolidColorBrush](#) 的 [Color](#) 属性进行动画处理。

XAML

```
<Button Content="I am animated">
    <Button.Background>
        <SolidColorBrush x:Name="AnimBrush"/>
    </Button.Background>
    <Button.Triggers>
        <EventTrigger RoutedEvent="FrameworkElement.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <ColorAnimation
                        Storyboard.TargetName="AnimBrush"
                        Storyboard.TargetProperty="(SolidColorBrush.Color)"
                        From="Blue" To="White" Duration="0:0:1"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
</Button>
```

```
</BeginStoryboard>
</EventTrigger>
</Button.Triggers>
</Button>
```

有关对属性进行动画处理的详细信息，请参阅[动画概述](#)和[情节提要概述](#)。

## 元数据重写

在从最初注册依赖属性的类派生时，可以通过替代依赖属性的元数据来更改该属性的特定行为。 替代元数据依赖于 [DependencyProperty](#) 标识符，不需要重新实现该属性。 元数据更改由属性系统在本机处理。 对于所有从基类继承的属性，每个类都有可能基于每个类型保留各自的元数据。

以下示例将替代 [DefaultStyleKey](#) 依赖属性的元数据。 替代此特定依赖属性的元数据是某个实现模式的一部分，该模式创建可以使用主题中的默认样式的控件。

C#

```
public class SpinnerControl : ItemsControl
{
    static SpinnerControl() => DefaultStyleKeyProperty.OverrideMetadata(
        typeof(SpinnerControl),
        new FrameworkPropertyMetadata(typeof(SpinnerControl))
    );
}
```

有关替代或访问依赖属性的元数据的详细信息，请参阅[替代依赖属性的元数据](#)。

## 属性值继承

元素可以从其在对象树中的父级继承依赖属性的值。

### ① 备注

属性值继承行为并未针对所有依赖属性在全局启用，因为继承的计算时间会影响性能。 属性值继承通常仅在指出适合使用属性值继承时启用。 可以通过在 SDK 参考中查看某个依赖属性的“依赖属性信息”部分，来检查该依赖属性是否继承属性值。

以下示例显示了一个绑定，它包含用于指定绑定源的 [DataContext](#) 属性。 因此，子对象中的绑定无需指定源，它们可以使用父对象 [StackPanel](#) 中 [DataContext](#) 的继承值。 或者，子对象可以直接在 [Binding](#) 中指定自己的 [DataContext](#) 或 [Source](#)，而不使用继承值。

## XAML

```
<StackPanel Canvas.Top="50" DataContext="{Binding Source={StaticResource TestData}}">
    <Button Content="{Binding XPath=test[2]/@text}" />
</StackPanel>
```

有关详细信息，请参阅[属性值继承](#)。

## WPF 设计器集成

具有作为依赖属性实现的属性的自定义控件可以与适用于 Visual Studio 的 WPF 设计器很好地集成。一个示例就是能够在“属性”窗口中编辑直接依赖属性和附加依赖属性。有关详细信息，请参阅[控件创作概述](#)。

## 依赖项属性值优先级

WPF 属性系统中任何基于属性的输入都可以设置依赖属性的值。由于存在[依赖属性值优先级](#)，使得属性获取值的方式的各种方案得以按可预测的方式交互。

### ① 备注

SDK 文档在讨论依赖属性时有时会使用“本地值”或“本地设置的值”等术语。本地设置的值是指在代码中直接为对象实例设置的属性值，或者在 XAML 中设置为元素特性的属性值。

下一个示例包含适用于任何按钮的 `Background` 属性的样式，但指定了一个具有本地设置的 `Background` 属性的按钮。从技术上说，该按钮的 `Background` 属性设置了两次，但是仅应用一个值，即具有最高优先级的值。本地设置的值具有最高优先级，对于正在运行的动画除外，但是在本示例中没有应用动画。因此，第二个按钮使用 `Background` 属性的本地设置值，而不使用样式资源库值。第一个按钮没有本地值或其他优先级高于样式资源库的值，因此使用 `Background` 属性的样式资源库值。

## XAML

```
<StackPanel>
    <StackPanel.Resources>
        <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Orange"/>
        </Style>
    </StackPanel.Resources>
    <Button>I am styled orange</Button>
    <Button Background="Pink">I am locally set to pink (not styled orange)</Button>
```

```
</Button>  
</StackPanel>
```

## 为什么存在依赖属性优先级？

本地设置的值优先于样式资源库值，后者支持元素属性的本地控制。有关详细信息，请参阅[依赖属性值优先级](#)。

### ① 备注

为 WPF 元素定义的许多属性并不是依赖属性，因为依赖属性通常仅在需要 WPF 属性系统的某个功能时实现。这些功能包括数据绑定、样式设置、动画、默认值支持、继承、附加属性和失效。

## 了解有关依赖属性的详细信息

- 组件开发人员或应用程序开发人员可能希望创建自己的依赖属性来添加功能，例如数据绑定或样式支持，或失效和值强制转换支持。有关详细信息，请参阅[自定义依赖属性](#)。
- 将依赖属性视为公共属性，可以由任何具有实例访问权限的调用方访问或发现。有关详细信息，请参阅[依赖属性安全性](#)。
- 附加属性是一种支持 XAML 中的专用语法的属性。附加属性通常与公共语言运行时属性没有 1:1 的对应关系，而且不一定是依赖属性。附加属性的主要用途是允许子元素向其父元素报告属性值，即使父元素和子元素的类成员列表中没有该属性也是如此。一个主要方案是使子元素能够告知父元素如何在 UI 中呈现它们。有关示例，请参阅[Dock](#) 和 [Left](#)。有关详细信息，请参阅[附加属性概述](#)。

## 另请参阅

- [自定义依赖属性](#)
- [只读依赖属性](#)
- [WPF 中的 XAML](#)
- [WPF 体系结构](#)

 在 GitHub 上与我们协作

.NET

.NET Desktop feedback

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 依赖属性值优先级 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 属性系统的工作会影响依赖属性的值。本文说明 WPF 属性系统中基于属性的不同输入的优先级如何确定依赖属性的有效值。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## WPF 属性系统

WPF 属性系统使用各种因素来确定依赖属性的值，例如实时属性验证、后期绑定和相关属性的属性更改通知。尽管用于确定依赖属性值的顺序和逻辑比较复杂，但了解它有助于避免不必要的属性设置，还可查明设置依赖属性的尝试未生成预期值的原因。

## 在多个位置设置的依赖属性

以下 XAML 示例演示对按钮 `Background` 属性进行的三种不同的“设置”操作如何影响其值。

XAML

```
<StackPanel>
    <StackPanel.Resources>
        <ControlTemplate x:Key="ButtonTemplate" TargetType="{x:Type Button}">
            <Border Background="{TemplateBinding Background}"
BorderThickness="{TemplateBinding BorderThickness}"
BorderBrush="{TemplateBinding BorderBrush}">
                <ContentPresenter HorizontalAlignment="Center"
VerticalAlignment="Center" />
            </Border>
        </ControlTemplate>
    </StackPanel.Resources>
```

```

<Button Template="{StaticResource ButtonTemplate}" Background="Red">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Blue"/>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="Yellow" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
    Which color do you expect?
</Button>
</StackPanel>

```

在该示例中，`Background` 属性在本地设置为 `Red`。但是，在按钮范围内声明的隐式样式会尝试将 `Background` 属性设置为 `Blue`。并且，当鼠标悬停在按钮上方时，隐式样式中的触发器会尝试将 `Background` 属性设置为 `Yellow`。除强制转换和动画外，本地设置的属性值具有最高优先级，因此按钮会使红色（即使在鼠标悬停时也是如此）。但是，如果从按钮中移除本地设置的值，它会从样式中获取其 `Background` 值。在样式中，触发器优先，因此按钮会在鼠标悬停时为黄色，否则为蓝色。该示例替换了按钮的默认 `ControlTemplate`，因为默认模板具有硬编码的鼠标悬停 `Background` 值。

## 依赖属性优先级列表

下面列出了在将运行时值分配给依赖属性时，属性系统所使用的最终优先级顺序。最高优先级最先列出。

1. 属性系统强制。有关强制转换的详细信息，请参阅[强制转换和动画](#)。
2. 活动动画或具有 Hold 行为的动画。若要拥有实用效果，动画必须拥有比基值（无动画）更高的优先级，即使基值进行了本地设置也是如此。有关详细信息，请参阅[强制转换和动画](#)。
3. 本地值。可以通过“包装器”属性设置本地值，这相当于在 XAML 中设置特性或属性元素，或者使用特定实例的属性调用 `SetValue` API。通过绑定或资源设置的本地值会具有与直接设置的值相同的优先级。
4. `TemplatedParent` 模板属性值。如果元素是通过模板创建（`ControlTemplate` 或 `DataTemplate`）创建的，则具有 `TemplatedParent`。有关详细信息，请参阅[TemplatedParent](#)。在通过 `TemplatedParent` 指定的模板中，优先级顺序为：
  - a. 触发器。
  - b. 属性集（通常通过 XAML 特性进行设置）。

5. 隐式样式。仅应用于 [Style](#) 属性。`Style` 值是具有与元素类型匹配的 [TargetType](#) 值的任何样式资源。样式资源必须存在于页面或应用程序中。对隐式样式资源的查找不会扩展到主题中的样式资源。
6. 样式触发器。样式触发器是显式或隐式样式中的触发器。样式必须存在于页面或应用程序中。默认样式中的触发器的优先级较低。
7. 模板触发器。模板触发器是直接应用的模板或样式中的模板中的触发器。样式必须存在于页面或应用程序中。
8. 样式资源库值。样式资源库值是样式中通过 [Setter](#) 应用的值。样式必须存在于页面或应用程序中。
9. 默认样式，也称为主题样式。有关详细信息，请参阅[默认（主题）样式](#)。在默认样式中，优先级顺序如下：
  - a. 活动触发器。
  - b. 资源库。
10. 继承。子元素的某些依赖属性从父元素继承其值。因此，可能不需要在整个应用程序中对每个元素设置属性值。有关详细信息，请参阅[属性值继承](#)。
11. 依赖属性元数据中的默认值。依赖属性可以在该属性的属性系统注册期间设置默认值。继承依赖属性的派生类可以选择按照类型重写依赖属性元数据（包括默认值）。有关详细信息，请参阅[依赖属性元数据](#)。对于继承的属性，父元素的默认值优先于子元素的默认值。因此，如果未设置可继承属性，则会使用根或父级的默认值，而不是子元素的默认值。

## TemplatedParent

[TemplatedParent](#) 优先级不应用于在标准应用程序标记中直接声明的元素的属性。只有对于通过应用模板而产生的可视化树中的子项而言，才存在 [TemplatedParent](#) 概念。当属性系统搜索由 [TemplatedParent](#) 为元素的属性值指定的模板时，它会搜索创建该元素的模板。来自 [TemplatedParent](#) 模板的属性值通常就像在元素上本地设置的值一样来应用，但是这些属性值的优先级低于实际本地值，因为模板可能进行共享。有关详细信息，请参阅 [TemplatedParent](#)。

## 样式属性

相同的优先级顺序应用于所有依赖属性（[Style](#) 属性除外）。`Style` 属性的独特性在于它本身无法样式化。不建议对 `Style` 属性进行强制转换或动画处理（对 `Style` 属性进行动

画处理需要自定义动画类）。因此，并非所有优先级项都适用。有三种设置 `Style` 属性的方法：

- **显式样式。** 直接设置元素的 `style` 属性。`Style` 属性值的作用如同本地值一样，具有与[优先级列表](#)中的项 3 相同的优先级。在大多数情形下，显式样式不会内联定义，而是作为资源显式引用，例如 `Style="{StaticResource myResourceKey}"`。
- **隐式样式。** 不直接设置元素的 `Style` 属性。而是当样式存在于页面或应用程序中的某个级别时应用样式，并且样式具有与应用样式的元素类型（例如 `<Style TargetType="x:Type Button">`）匹配的资源键。类型必须完全匹配，例如 `<Style TargetType="x:Type Button">` 不会应用于 `MyButton` 类型，即使 `MyButton` 派生自 `Button`。`Style` 属性值具有与[优先级列表](#)中的项 5 相同的优先级。可以通过调用 `DependencyPropertyHelper.GetValueSource` 方法、传入 `Style` 属性以及在结果中检查 `ImplicitStyleReference` 来检测隐式样式值。
- 默认样式，也称为主题样式。不直接设置元素的 `Style` 属性。相反，它来自 WPF 表示引擎进行的运行时主题评估。在运行时之前，`Style` 属性值为 `null`。`Style` 属性值具有与[优先级列表](#)中的项 9 相同的优先级。

## 默认（主题）样式

WPF 附带的每个控件都具有默认样式，该样式可能因主题而异，这便是默认样式有时称为主题样式的原因。

`ControlTemplate` 是控件的默认样式中的一个重要项。`ControlTemplate` 是样式的 `Template` 属性的资源库值。如果默认样式不包含模板，自定义样式中没有自定义模板的控件没有可视化外观。模板不仅定义控件的可视化外观，还定义在模板的可视化树中的属性与对应的控件类之间的联系。每个控件都公开一组属性，这些属性可以影响控件的可视化外观，而无需替换模板。例如，请考虑 `Thumb` 控件（它是 `ScrollBar` 组件）的默认可视化外观。

`Thumb` 控件具有某些可自定义属性。`Thumb` 控件的默认模板可创建一个基本结构或可视化树（具有几个嵌套的 `Border` 组件），以创建棱台外观。在模板中，旨在可由 `Thumb` 类自定义的属性通过 `TemplateBinding` 进行公开。`Thumb` 控件的默认模板具有各种边界属性，这些属性与诸如 `Background` 或 `BorderThickness` 这类属性共享模板绑定。但是，在属性或可视化排列的值在模板中进行硬编码，或者绑定到直接来自主题的值的情况下，只能通过替换整个模板来更改这些值。一般而言，如果属性来自模板化父级，并且不是通过 `TemplateBinding` 进行公开，则不能通过样式更改属性值，因为没有方便的方法可以将其设置为目标。但是，该属性仍然可能受所应用的模板中的属性值继承影响，或受默认值影响。

默认样式在其定义中指定一个 `TargetType`。运行时主题评估会将默认样式的 `TargetType` 与控件的 `DefaultStyleKey` 属性进行匹配。相反，隐式样式的查找行为使用控件的实际类型。`DefaultStyleKey` 的值由派生类继承，因此可能没有关联样式的派生元素会获取默认可视化外观。例如，如果从 `Button` 派生 `MyButton`，则 `MyButton` 会继承 `Button` 的默认模板。派生类可以重写依赖属性元数据中的 `DefaultStyleKey` 的默认值。因此，如果对 `MyButton` 需要其他可视化表示形式，则可以对 `MyButton` 上的 `DefaultStyleKey` 重写依赖属性元数据，然后定义你将随 `MyButton` 控件一起打包的相关默认样式（包括模板）。有关详细信息，请参阅[控件创作概述](#)。

## 动态资源

动态资源引用和绑定操作具有其设置位置的优先级。例如，应用于本地值的动态资源具有与[优先级列表](#)中的项 3 相同的优先级。作为另一个示例，应用于默认样式中属性资源库的动态资源绑定具有与[优先级列表](#)中的项 9 相同的优先级。由于动态资源引用和绑定必须从应用程序的运行时状态中获取值，因此确定任何给定属性的属性值优先级的过程会扩展到运行时。

动态资源引用在技术上不是属性系统的一部分，具有其自己的与[优先级列表](#)交互的查找顺序。本质上，动态资源引用的优先级是：元素到页面根、应用程序、主题，然后是系统。有关详细信息，请参阅[XAML 资源](#)。

尽管动态资源引用和绑定具有其设置位置的优先级，但值会延迟。这样的一个后果是，如果将动态资源或绑定设置为某个本地值，则对该本地值的任何更改都会完全替换该动态资源或绑定。即使调用 `ClearValue` 方法清除本地设置的值，动态资源或绑定也不会还原。实际上，如果对具有动态资源或绑定的属性（没“文字”本地值）调用 `ClearValue`，则会清除动态资源或绑定。

## SetCurrentValue

`SetCurrentValue` 方法是设置属性的另一种方式，但它不在[优先级列表](#)中。通过 `SetCurrentValue` 可以更改属性值而不会覆盖以前值的源。例如，如果某个属性由触发器设置，然后你使用 `SetCurrentValue` 分配另一个值，则下一个触发器操作会将该属性设置回触发器值。每当要设置属性值但是不向该值提供某个本地值的优先级时，便可以使用 `SetCurrentValue`。同样，可以使用 `SetCurrentValue` 更改属性值而不覆盖绑定。

## 强制转换和动画

强制转换和动画都对基值执行操作。基值是优先级最高的依赖属性值，通过在[优先级列表](#)中向上评估来确定（直到达到项 2）。

如果动画没有为某些行为指定 `From` 和 `To` 属性值，或者动画在完成时有意还原为基值，则基值可能会影响动画值。 若要了解实际效果，请运行[目标值](#)示例应用程序。 在示例中，对于矩形高度，尝试设置与任何 `From` 值不同的初始本地值。 示例动画立即使用 `From` 值（而不是基值）开始。 通过将 `Stop` 指定为 `FillBehavior`，完成时动画会将属性值重置为其基值。 正常优先级会用于动画结束后的基值确定。

可以将多个动画应用于单个属性，每个动画具有不同的优先级。 WPF 表示引擎可以组合动画值，而不是应用优先级最高的动画（具体取决于动画的定义方式和进行动画处理的值类型）。 有关详细信息，请参阅[动画概述](#)。

强制转换处于[优先级列表](#)的顶部。 即使正在运行的动画也会受到值强制转换的制约。 WPF 中的某些现有依赖属性具有内置强制转换。 对于自定义依赖属性，可以通过编写在创建属性时作为元数据的一部分进行传递的 `CoerceValueCallback` 来定义强制转换行为。 还可以通过在派生类中重写现有属性的元数据来重写该属性的强制转换行为。 强制转换与基值的交互使强制转换上的约束就像当时存在这些约束一样进行应用，但基值仍将保留。 因此，如果强制转换中的约束后来被解除，强制转换将返回与基值最接近的可能值，并且一旦所有约束都解除，强制转换对属性的影响可能会立即停止。 有关强制转换行为的详细信息，请参阅[依赖属性回调和验证](#)。

## 触发器行为

控件常常将触发器行为定义为其[默认样式](#)的一部分。 在控件上设置本地属性可能会与这些触发器发生冲突，从而防止触发器响应（在视觉上或行为上）用户驱动的事件。 属性触发器常用于控件状态属性，如 `IsSelected` 或 `.IsEnabled`。 例如，默认情况下，当禁用 `Button` 时，主题样式触发器（`Enabled` 为 `false`）会设置 `Foreground` 值，以使 `Button` 灰显。 如果设置了本地 `Foreground` 值，则优先级较高的本地属性值会覆盖主题样式 `Foreground` 值，即使 `Button` 已禁用。 设置为控件重写主题级别触发器行为的属性值时，请注意不要过度干扰该控件的预期用户体验。

## ClearValue

`ClearValue` 方法为元素清除依赖属性的任何本地应用值。但是，调用 `ClearValue` 并不能保证注册属性时在元数据中指定的默认值就是新的有效值。[优先级列表](#)中的所有其他参与者仍然处于活动状态，仅移除本地设置的值。例如，如果对具有主题样式的属性调用 `ClearValue`，主题样式值将作为新值而不是基于元数据的默认值进行应用。如果要将属性值设置为已注册的元数据默认值，则通过查询依赖属性元数据来获取默认元数据值，并通过对 `SetValue` 的调用在本地设置属性值。

## 另请参阅

- [DependencyObject](#)
- [DependencyProperty](#)
- [依赖属性概述](#)
- [自定义依赖属性](#)
- [依赖属性回调和验证](#)

## ⌚ 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

⌚ 提出文档问题

↗ 提供产品反馈

# 附加属性概述 (WPF .NET)

项目 • 2023/10/13

附加属性是一个 Extensible Application Markup Language (XAML) 概念。附加属性允许为派生自 [DependencyObject](#) 的任何 XAML 元素设置额外的属性/值对，即使该元素未在其对象模型中定义这些额外的属性。额外的属性可进行全局访问。附加属性通常定义为没有常规属性包装器的依赖属性的专用形式。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 XAML 并知道如何编写 Windows Presentation Foundation (WPF) 应用程序。

## 使用附加属性的原因

附加属性允许子元素为父元素中定义的属性指定唯一值。一个常见方案是，一个子元素指定它应如何被其父元素呈现在 UI 中。例如，[DockPanel.Dock](#) 是一个附加属性，因为它在 [DockPanel](#) 的子元素上设置，而不是在 [DockPanel](#) 本身设置。[DockPanel](#) 类定义名为 [DockProperty](#) 的静态 [DependencyProperty](#) 字段，然后提供 [GetDock](#) 和 [SetDock](#) 方法作为附加属性的公共访问器。

## XAML 中的附加属性

在 XAML 中，可以使用语法 `<attached property provider type>.<property name>` 设置附加属性，其中附加属性提供程序是定义附加属性的类。以下示例演示 [DockPanel](#) 的子元素如何设置 [DockPanel.Dock](#) 属性值。

XAML

```
<DockPanel>
    <TextBox DockPanel.Dock="Top">Enter text</TextBox>
</DockPanel>
```

其用法类似于静态属性，因为你引用的是拥有和注册附加属性的类型（例如 `DockPanel`），而不是实例名称。

使用 XAML 特性指定附加属性时，只适合执行 `set` 操作。尽管存在一些用于比较值的间接机制（如样式中的触发器），但无法通过 XAML 直接获取属性值。

## WPF 中的附加属性

附加属性是 XAML 概念，依赖属性是 WPF 概念。在 WPF 中，WPF 类型上大多数与 UI 相关的附加属性都作为依赖属性实现。作为依赖属性实现的 WPF 附加属性支持依赖属性概念，例如包含元数据中的默认值的属性元数据。

## 附加属性用法模型

尽管任何对象都可以设置附加属性值，但这并不意味着设置值会生成有形的结果，或者该值将被另一个对象使用。附加属性的主要用途是为来自各种类层次结构和逻辑关系的对象提供一种方法，将公共信息报告给定义附加属性的类型。附加属性用法通常遵循以下模型之一：

- 定义附加属性的类型是为附加属性设置值的元素的父级。父类型通过作用于对象树结构的内部逻辑循环访问其子对象，获取值，并以某种方式作用于这些值。
- 定义附加属性的类型将用作各种可能的父元素和内容模型的子元素。
- 定义附加属性的类型表示一项服务。其他类型为该附加属性设置值。然后，当在服务的上下文中计算设置该属性的元素时，将通过服务类的内部逻辑获取附加属性的值。

## 父级定义的附加属性示例

WPF 定义附加属性的典型场景如下：父元素支持子元素集合，并且父元素根据其每个子元素报告的数据实现行为。

`DockPanel` 定义 `DockPanel.Dock` 附加属性。`DockPanel` 具有类级代码，特别是 `MeasureOverride` 和 `ArrangeOverride`，这是其呈现逻辑的一部分。`DockPanel` 实例检查其任何直接子元素是否为 `DockPanel.Dock` 设置了值。如果已设置，这些值将变为应用于每个子元素的呈现逻辑的输入。尽管理论上附加属性可能会影响直接父元素之外的元素，但嵌套 `DockPanel` 实例的定义行为是仅与其直接子元素集合交互。因此，如果有没有 `DockPanel` 父级的元素设置 `DockPanel.Dock`，则不会引发错误或异常，并且会创建一个不由任何 `DockPanel` 使用的全局属性值。

## 代码中的附加属性

WPF 中的附加属性没有典型的 CLR `get` 和 `set` 包装器方法，因为这些属性可能从 CLR 命名空间外部设置。若要允许 XAML 处理器在分析 XAML 时设置这些值，定义附加属性的类必须实现 `Get<property name>` 和 `Set<property name>` 形式的专用访问器方法。

你还可以使用专用访问器方法在代码中获取和设置附加属性，如以下示例所示。在此示例中，`myTextBox` 是 `TextBox` 类的实例。

C#

```
DockPanel myDockPanel = new();
TextBox myTextBox = new();
myTextBox.Text = "Enter text";

// Add child element to the DockPanel.
myDockPanel.Children.Add(myTextBox);

// Set the attached property value.
DockPanel.SetDock(myTextBox, Dock.Top);
```

如果不将 `myTextBox` 添加为 `myDockPanel` 的子元素，则调用 `SetDock` 不会引发异常或产生任何影响。只有对 `DockPanel` 的子元素设置的 `DockPanel.Dock` 值才会影响呈现效果，并且无论在将子元素添加到 `DockPanel` 之前还是之后设置值，呈现效果都相同。

从代码的角度来看，附加属性类似于具有方法访问器而不是属性访问器的支持字段，可以在任何对象上设置，并且无需先在这些对象上定义。

## 附加属性元数据

附加属性的元数据通常与依赖属性的元数据没有什么不同。注册附加属性时，可使用 `FrameworkPropertyMetadata` 指定属性的特征，例如属性是否影响呈现或测量。通过替代附加属性元数据指定默认值时，该值将成为替代类实例上显式附加属性的默认值。如果未另外设置附加属性值，则在使用 `Get<property name>` 访问器和指定元数据的类实例查询属性时，将报告默认值。

若要对属性启用属性值继承，应使用附加属性，而不是非附加依赖属性。有关详细信息，请参阅[属性值继承](#)。

## 自定义附加属性

### 何时创建附加属性

在以下情况下创建附加属性很有用：

- 需要一种可用于类（定义类除外）的属性设置机制。一个常见方案是用于 UI 布局，例如，`DockPanel.Dock`、`Panel.ZIndex` 和 `Canvas.Top` 都是现有布局属性的示例。在布局方案中，布局控制元素的子元素能够向其布局父级表达布局要求，并为父级定义的附加属性设置值。
- 你的某个类表示服务，你希望其他类更透明地集成该服务。
- 需要 Visual Studio WPF 设计器支持，例如能够通过“属性”窗口编辑属性。有关详细信息，请参阅[控件创作概述](#)。
- 你想使用属性值继承。

## 如何创建附加属性

如果类定义了一个仅供其他类型使用的附加属性，那么该类不需要从 `DependencyObject` 派生。否则，应遵循 WPF 模型，通过从 `DependencyObject` 派生类，将附加属性也设置成依赖属性。

通过声明 `DependencyProperty` 类型的 `public static readonly` 字段，将附加属性定义为定义类中的依赖项。然后，将 `RegisterAttached` 方法的返回值分配给该字段，该字段也称为依赖属性标识符。遵循 WPF 属性命名约定，通过命名标识符字段 `<property name>Property` 将字段与它们所表示的属性区分开来。此外，提供静态 `Get<property name>` 和 `Set<property name>` 访问器方法，让属性系统访问附加属性。

以下示例演示如何使用 `RegisterAttached` 方法注册依赖属性，以及如何定义访问器方法。在此示例中，附加属性的名称为 `HasFish`，因此标识符字段命名为 `HasFishProperty`，访问器方法命名为 `GetHasFish` 和 `SetHasFish`。

C#

```
public class Aquarium : UIElement
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata.
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.RegisterAttached(
            "HasFish",
            typeof(bool),
            typeof(Aquarium),
            new FrameworkPropertyMetadata(defaultValue: false,
                flags: FrameworkPropertyMetadataOptions.AffectsRender)
        );

    // Declare a get accessor method.
    public static bool GetHasFish(UIElement target) =>
        (bool)target.GetValue(HasFishProperty);
}
```

```
// Declare a set accessor method.  
public static void SetHasFish(UIElement target, bool value) =>  
    target.SetValue(HasFishProperty, value);  
}
```

## Get 访问器

`get` 访问器方法签名名为 `public static object Get<property name>(DependencyObject target)`，其中：

- `target` 是从中读取附加属性的 `DependencyObject`。`target` 类型可以比 `DependencyObject` 更具体。例如，`DockPanel.GetDock` 访问器方法将 `UIElement` 类型化为 `target`，因为附加属性要在 `UIElement` 实例上设置。`UIElement` 间接派生自 `DependencyObject`。
- 返回类型可以比 `object` 更具体。例如，`GetDock` 方法将返回值类型化为 `Dock`，因为返回值应为 `Dock` 枚举。

### ① 备注

附加属性的 `get` 访问器是设计工具（如 Visual Studio 或 Blend for Visual Studio）中的数据绑定支持所必需的。

## Set 访问器

`set` 访问器方法签名名为 `public static void Set<property name>(DependencyObject target, object value)`，其中：

- `target` 是写入附加属性的 `DependencyObject`。`target` 类型可以比 `DependencyObject` 更具体。例如，`SetDock` 方法将 `UIElement` 类型化为 `target`，因为附加属性要在 `UIElement` 实例上设置。`UIElement` 间接派生自 `DependencyObject`。
- `value` 类型可以比 `object` 更具体。例如，`SetDock` 方法需要一个 `Dock` 值。XAML 加载程序必须能够根据表示附加属性值的标记字符串生成 `value` 类型。因此，你使用的类型必须有类型转换、值序列化程序或标记扩展支持。

## 附加属性特性

WPF 定义了几个 .NET 特性，这些特性向反射进程以及反射和属性信息的使用者（例如设计器）提供有关附加属性的信息。设计器使用 WPF 定义的 .NET 特性来限制属性窗口中

显示的属性，以避免用户被包含所有附加属性的全局列表淹没。你可以考虑对自己的自定义附加属性应用这些特性。以下参考页面介绍了 .NET 特性的用途和语法：

- [AttachedPropertyBrowsableAttribute](#)
- [AttachedPropertyBrowsableForChildrenAttribute](#)
- [AttachedPropertyBrowsableForTypeAttribute](#)
- [AttachedPropertyBrowsableWhenAttributePresentAttribute](#)

## 了解详细信息

- 有关如何创建附加属性的详细信息，请参阅[注册附加属性](#)。
- 有关依赖属性和附加属性的更多高级使用方案，请参阅[自定义依赖属性](#)。
- 你可以将一个属性同时注册为附加属性和依赖属性，并包含常规属性包装器。这样一来，就可以使用属性包装器对某个元素设置属性，同时使用 XAML 附加属性语法对任何其他元素设置属性。有关示例，请参见 [FrameworkElement.FlowDirection](#)。

## 另请参阅

- [DependencyProperty](#)
- [依赖属性概述](#)
- [自定义依赖属性](#)
- [WPF 中的 XAML](#)
- [如何：注册附加属性](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 集合类型依赖属性 (WPF .NET)

项目 • 2023/10/13

本文提供了用于实现集合类型依赖属性的指南和建议模式。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 实现集合类型依赖属性

通常，依赖属性的实现模式由 `DependencyProperty` 标识符（而非字段或其他构造）提供支持。实现集合类型依赖属性时，可以按照此相同的模式。如果集合元素类型是 `DependencyObject` 或 `Freezable` 派生类，则模式会更复杂。

## 初始化集合

创建依赖属性时，通常通过依赖属性元数据指定默认值，而不是指定初始属性值。但是，如果属性值是引用类型，则应在注册依赖属性的类的构造函数中设置默认值。依赖属性元数据不应包含默认引用类型值，因为该值将分配给该类的所有实例，从而创建单一实例类。

以下示例声明了一个 `Aquarium` 类，该类包含泛型 `List<T>` 中的 `FrameworkElement` 元素的集合。传递给 `RegisterReadOnly(String, Type, Type, PropertyMetadata)` 方法的 `PropertyMetadata` 中不包含默认集合值，而是使用类构造函数将默认集合值设置为新的泛型 `List`。

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, and property metadata.
    private static readonly DependencyPropertyKey
```

```

s_aquariumContentsPropertyKey =
    DependencyProperty.RegisterReadOnly(
        name: "AquariumContents",
        propertyType: typeof(List<FrameworkElement>),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata()
        //typeMetadata: new FrameworkPropertyMetadata(new
List<FrameworkElement>())
    );

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(s_aquariumContentsPropertyKey, new
List<FrameworkElement>());

    // Declare a public get accessor.
    public List<FrameworkElement> AquariumContents =>

    (List<FrameworkElement>)GetValue(s_aquariumContentsPropertyKey.DependencyPro
perty);
}

public class Fish : FrameworkElement { }

```

以下测试代码实例化两个单独的 `Aquarium` 实例，并向每个集合添加不同的 `Fish` 项。如果运行此代码，你将看到每个 `Aquarium` 实例都有一个集合项，正如预期的那样。

C#

```

private void InitializeAquariums(object sender, RoutedEventArgs e)
{
    Aquarium aquarium1 = new();
    Aquarium aquarium2 = new();
    aquarium1.AquariumContents.Add(new Fish());
    aquarium2.AquariumContents.Add(new Fish());
    MessageBox.Show(
        $"aquarium1 contains {aquarium1.AquariumContents.Count} fish\r\n" +
        $"aquarium2 contains {aquarium2.AquariumContents.Count} fish");
}

```

但是，如果注释掉类构造函数并将默认集合值作为 `PropertyMetadata` 传递给 `RegisterReadOnly(String, Type, Type, PropertyMetadata)` 方法，你将看到每个 `Aquarium` 实例都会获取两个集合项！这是因为两个 `Fish` 实例都添加到了同一列表中，该列表由 `Aquarium` 类的所有实例共享。因此，如果旨在让每个对象实例都有其自己的列表，则应在类构造函数中设置默认值。

## 初始化读写集合

以下示例使用非键签名方法 `Register(String, Type, Type)` 和 `SetValue(DependencyProperty, Object)` 在 `Aquarium` 类中声明了一个读写集合类型依赖属性。

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, and owner type. Store the dependency property
    // identifier as a public static readonly member of the class.
    public static readonly DependencyProperty AquariumContentsProperty =
        DependencyProperty.Register(
            name: "AquariumContents",
            propertyType: typeof(List<FrameworkElement>),
            ownerType: typeof(Aquarium)
        );

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(AquariumContentsProperty, new
List<FrameworkElement>());

    // Declare public get and set accessors.
    public List<FrameworkElement> AquariumContents
    {
        get => (List<FrameworkElement>)GetValue(AquariumContentsProperty);
        set => SetValue(AquariumContentsProperty, value);
    }
}
```

## FreezableCollection 依赖属性

集合类型依赖属性不会自动报告其子属性的更改。因此，如果要绑定到集合，则绑定可能不会报告更改，从而导致某些数据绑定方案无效。但是，如果将 `FreezableCollection<T>` 用于依赖属性类型，则会正确报告对集合元素的属性的更改，并且绑定会按预期工作。

若要在依赖对象集合中启用子属性绑定，请使用集合类型 `FreezableCollection`，并具有任何 `DependencyObject` 派生类的类型约束。

以下示例声明了一个 `Aquarium` 类，该类包含类型约束为 `FrameworkElement` 的 `FreezableCollection`。传递给 `RegisterReadOnly(String, Type, Type, PropertyMetadata)` 方法的 `PropertyMetadata` 中不包含默认集合值，而是使用类构造函数将默认集合值设置为新的 `FreezableCollection`。

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, and owner type.
    private static readonly DependencyPropertyKey
s_aquariumContentsPropertyKey =
    DependencyProperty.RegisterReadOnly(
        name: "AquariumContents",
        propertyType: typeof(FreezableCollection<FrameworkElement>),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata()
    );

    // Store the dependency property identifier as a static member of the
    // class.
    public static readonly DependencyProperty AquariumContentsProperty =
        s_aquariumContentsPropertyKey.DependencyProperty;

    // Set the default collection value in a class constructor.
    public Aquarium() => SetValue(s_aquariumContentsPropertyKey, new
FreezableCollection<FrameworkElement>());

    // Declare a public get accessor.
    public FreezableCollection<FrameworkElement> AquariumContents =>
        (FreezableCollection<FrameworkElement>)GetValue(AquariumContentsProperty);
}
```

## 另请参阅

- [FreezableCollection<T>](#)
- [XAML 及 WPF 的自定义类](#)
- [数据绑定概述](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [依赖属性元数据](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# 自定义依赖属性 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 应用程序开发人员和组件创建者可以创建自定义依赖属性来扩展其属性的功能。与公共语言运行时 (CLR) 属性不同，依赖属性添加了对样式、数据绑定、继承、动画和默认值的支持。`Background`、`Width` 和 `Text` 是 WPF 类中现有依赖属性的示例。本文介绍如何实现自定义依赖属性，并提供用于提高性能、可用性和多样性的选项。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 依赖属性标识符

依赖属性是通过 `Register` 或 `RegisterReadOnly` 调用向 WPF 属性系统注册的属性。

`Register` 方法返回 `DependencyProperty` 实例，该实例保存依赖属性的已注册名称和特征。你会将 `DependencyProperty` 实例分配到一个静态只读字段（称为依赖属性标识符），该字段按约定命名为 `<property name>Property`。例如，`Background` 属性的标识符字段始终为 `BackgroundProperty`。

依赖属性标识符用作用于获取或设置属性值的支持字段，而不是通过私有字段支持属性的标准模式。属性系统不仅使用标识符，XAML 处理器也可以使用它，代码（并且可能是外部代码）可以通过标识符访问依赖属性。

依赖属性只能应用于派生自 `DependencyObject` 类型的类。大多数 WPF 类都支持依赖属性，因为 `DependencyObject` 靠近 WPF 类层次结构的根。有关依赖属性以及用于描述它们的术语和约定，请参阅[依赖属性概述](#)。

## 依赖属性包装器

不是附加属性的 WPF 依赖属性通过实现 `get` 和 `set` 访问器的 CLR 包装器进行公开。通过使用属性包装器，依赖属性的使用者可以获取或设置依赖属性值，就像获取或设置任何其他 CLR 属性一样。`get` 和 `set` 访问器通过 `DependencyObject.GetValue` 和 `DependencyObject.SetValue` 调用与底层属性系统交互，并以参数的形式传入依赖属性标识符。依赖属性的使用者通常不会直接调用 `GetValue` 或 `SetValue`，但如果要实现自定义依赖属性，则会在包装器中使用这些方法。

## 何时实现依赖属性

对派生自 `DependencyObject` 的类实现属性时，可以通过使用 `DependencyProperty` 标识符来支持属性使其成为依赖属性。创建依赖属性是否有益取决于你的方案。尽管通过私有字段支持属性足够用于某些方案，但如果希望属性支持以下一个或多个 WPF 功能，请考虑实现依赖属性：

- 可在样式中设置的属性。有关详细信息，请参阅[样式和模板](#)。
- 支持数据绑定的属性。有关数据绑定依赖属性的详细信息，请参阅[绑定两个控件的属性](#)。
- 可通过动态资源引用设置的属性。有关详细信息，请参阅[XAML 资源](#)。
- 自动从元素树中的父元素继承其值的属性。为此，即使还会创建属性包装器以进行 CLR 访问，也需要使用 `RegisterAttached` 进行注册。有关详细信息，请参阅[属性值继承](#)。
- 可进行动画处理的属性。有关详细信息，请参阅[动画概述](#)。
- 当属性值发生更改时，WPF 属性系统进行的通知。更改可能是由属性系统、环境、用户或样式执行的操作造成的。属性可以在属性元数据中指定回调方法，每次属性系统确定属性值已更改时会调用此回调方法。与此相关的一个概念是属性值强制转换。有关详细信息，请参阅[依赖属性回调和验证](#)。
- 对 WPF 进程读取的依赖属性元数据的访问。例如，可以使用属性元数据：
  - 指定更改的依赖属性值是否应使布局系统重新安排元素的视觉对象。
  - 通过替代派生类的元数据来设置依赖属性的默认值。
- Visual Studio WPF 设计器支持，例如在“属性”窗口中编辑自定义控件的属性。有关详细信息，请参阅[控件创作概述](#)。

对于某些方案，替代现有依赖属性的元数据是比实现新依赖属性更好的选项。元素据替代是否可行取决于方案以及方案与现有 WPF 依赖属性和类的实现的相似度。有关替代现有依赖属性上的元素据的详细信息，请参阅[依赖属性元素据](#)。

# 创建依赖属性的检查清单

按照以下步骤创建依赖属性。某些步骤可以合并，在单行代码中并实现。

1. (可选) 创建依赖属性元数据。
2. 将依赖属性注册到属性系统，指定属性名称、所有者类型、属性值类型以及 (可选) 属性元数据。
3. 在所有者类型上将 `DependencyProperty` 标识符定义为 `public static readonly` 字段。标识符字段名称是追加了 `Property` 后缀的属性名称。
4. 定义名称与依赖属性名称相同的 CLR 包装器属性。在 CLR 包装器中，实现 `get` 和 `set` 访问器，它们与支持包装器的依赖属性连接。

## 注册属性

若要使属性成为依赖属性，必须将它注册到属性系统。若要注册属性，请在类的主体中（但在任何成员定义之外）调用 `Register` 方法。`Register` 方法会返回调用属性系统 API 时将使用的唯一依赖属性标识符。在成员定义之外进行 `Register` 调用的原因是会将返回值分配给 `DependencyProperty` 类型的 `public static readonly` 字段。将在类中创建的此字段是依赖属性的标识符。在以下示例中，`Register` 的第一个参数将依赖属性命名为 `AquariumGraphic`。

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
        name: "AquariumGraphic",
        propertyType: typeof(Uri),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata(
            defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
            flags: FrameworkPropertyMetadataOptions.AffectsRender,
            propertyChangedCallback: new
            PropertyChangedCallback(OnUriChanged))
    );
```

### ① 备注

在类的主体中定义依赖属性是典型的实现，但也可以在类静态构造函数中定义依赖属性。需要多行代码来初始化依赖属性时，此方法会很有用。

## 依赖属性命名

对于属性系统的正常行为，为依赖属性建立的命名约定是必需的。创建的标识符字段的名称必须是属性的已注册名称并且具有后缀 `Property`。

依赖属性名称在注册类中必须是唯一的。通过基类型继承的依赖属性已注册，无法由派生类型注册。但是，可以通过将你的类添加为依赖属性的所有者，使用通过不同类型（甚至是你的类不继承的类型）注册的依赖属性。有关将类添加为所有者的详细信息，请参阅[依赖属性元数据](#)。

## 实现属性包装器

按照约定，包装器属性的名称必须与 `Register` 调用的第一个参数（这是依赖属性名称）相同。包装器实现会在 `get` 访问器中调用 `GetValue`，并在 `set` 访问器中调用 `SetValue`（用于读写属性）。以下示例演示一个包装器（在注册调用和标识符字段声明后面）。WPF 类上的所有公共依赖属性都使用类似的包装器模型。

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
        name: "AquariumGraphic",
        propertyType: typeof(Uri),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata(
            defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
            flags: FrameworkPropertyMetadataOptions.AffectsRender,
            propertyChangedCallback: new
            PropertyChangedCallback(OnUriChanged))
    );

// Declare a read-write property wrapper.
public Uri AquariumGraphic
{
    get => (Uri)GetValue(AquariumGraphicProperty);
    set => SetValue(AquariumGraphicProperty, value);
}
```

除了在极少数情况下，包装器实现应仅包含 `GetValue` 和 `SetValue` 代码。有关这背后的原因，请参阅[自定义依赖属性的影响](#)。

如果属性未遵循建立的命名约定，则可能会遇到以下问题：

- 样式和模板的某些方面不起作用。
- 大多数工具和设计器依赖命名约定来正确序列化 XAML 并在每个属性级别提供设计器环境帮助。
- WPF XAML 加载程序的当前实现会完全跳过包装器，并依赖于命名约定来处理特性值。有关详细信息，请参阅[XAML 加载和依赖属性](#)。

## 依赖属性元数据

注册依赖属性时，属性系统会创建一个存储属性特征的元数据对象。通过 `Register` 方法的重载可以在注册期间指定属性元数据，例如 `Register(String, Type, Type, PropertyMetadata)`。属性元数据的常见用途是为使用依赖属性的新实例应用自定义默认值。如果未提供属性元数据，则属性系统会将默认值分配给许多依赖属性特征。

如果要对派生自 `FrameworkElement` 的类创建依赖属性，则可以使用更专业的元数据类 `FrameworkPropertyMetadata`，而不是其基类 `PropertyMetadata`。通过多个 `FrameworkPropertyMetadata` 构造函数签名可指定元数据特征的不同组合。如果只想指定默认值，请使用 `FrameworkPropertyMetadata(Object)` 并将默认值传递给 `Object` 参数。确保值类型与 `Register` 调用中指定的 `propertyType` 匹配。

通过某些 `FrameworkPropertyMetadata` 重载可以为属性指定元数据选项标志。属性系统会将这些标志转换为离散属性，标志值会由 WPF 进程（如布局引擎）使用。

## 设置元数据标志

设置元数据标志时，请考虑以下事项：

- 如果属性值（或对其进行的更改）会影响布局系统呈现 UI 元素的方式，则设置以下一个或多个标志：
  - `AffectsMeasure`，指示属性值的更改需要更改 UI 呈现，尤其是对象在其父级内占用的空间。例如，为 `Width` 属性设置此元数据标志。
  - `AffectsArrange`，指示属性值的更改需要更改 UI 呈现，尤其是对象在其父级内的位置。通常，对象不会也更改大小。例如，为 `Alignment` 属性设置此元数据标志。

- [AffectsRender](#), 指出发生了不影响布局和度量值的更改，但仍需要其他呈现方式。例如，为 `Background` 属性或影响元素颜色的任何其他属性设置此标志。

还可以将这些标志用作属性系统（或布局）回调的替代实现的输入。例如，当实例的属性报告值更改并在元数据中设置了 [AffectsArrange](#) 时，可以使用 [OnPropertyChanged](#) 回调调用 [InvalidateArrange](#)。

- 某些属性会以其他方式影响其父元素的呈现特征。例如，对 [MinOrphanLines](#) 属性的更改可以更改流文档的整体呈现。可在自己的属性中使用 [AffectsParentArrange](#) 或 [AffectsParentMeasure](#) 表示父操作。
- 默认情况下，依赖属性支持数据绑定。但是，当数据绑定没有现实方案时，或者数据绑定性能有问题（例如针对大型对象）时，可以使用 [IsDataBindingAllowed](#) 禁用数据绑定。
- 尽管依赖属性的默认数据绑定模式是 [OneWay](#)，但可以将特定绑定的绑定模式更改为 [TwoWay](#)。有关详细信息，请参阅[绑定方向](#)。作为依赖属性作者，你甚至可以选择将双向绑定设置为默认模式。使用双向数据绑定的现有依赖属性的示例是 [MenuItem.IsSubmenuOpen](#)，它具有基于其他属性和方法调用的状态。  
`IIsSubmenuOpen` 的应用场景为：其设置逻辑和 [MenuItem](#) 合成与默认主题样式交互。[TextBox.Text](#) 是默认情况下使用双向绑定的另一个 WPF 依赖属性。
- 还可以通过设置 [Inherits](#) 标志为依赖属性启用属性继承。属性继承对于父元素和子元素具有共同属性的情况非常有用，它使子元素可以继承共同属性的父值。可继承属性的示例是 [DataContext](#)，它支持使用[主-从方案](#)进行数据呈现的绑定操作。通过属性值继承可以在页面或应用程序根处指定数据上下文，从而不必为子元素绑定指定数据上下文。尽管继承的属性值会替代默认值，但可以在任何子元素上本地设置属性值。请谨慎使用属性值继承，因为它具有性能成本。有关详细信息，请参阅[属性值继承](#)。
- 设置 [Journal](#) 标志，以指示导航日志服务是否应该检测或使用依赖属性。例如，[SelectedIndex](#) 属性设置 `Journal` 标志，以建议应用程序保留所选项的日志历史记录。

## 只读依赖项属性

可以定义只读的依赖属性。典型方案是存储内部状态的依赖属性。例如，[IsMouseOver](#) 是只读的，因为其状态只应由鼠标输入确定。有关详细信息，请参阅[只读依赖属性](#)。

## 集合类型依赖属性

集合类型依赖属性具有需要考虑的额外实现问题，例如为集合元素设置引用类型和数据绑定支持的默认值。有关详细信息，请参阅[集合类型依赖属性](#)。

## 依赖项属性的安全性

通常会将依赖属性声明为公共属性，并将 `DependencyProperty` 标识符字段声明为 `public static readonly` 字段。如果指定限制性更高的访问级别（例如 `protected`），仍可通过将其标识符与属性系统 API 结合使用来访问依赖属性。甚至可以通过 WPF 元数据报告或值确定 API（例如 `LocalValueEnumerator`）访问受保护的标识符字段。有关详细信息，请参阅[依赖属性安全性](#)。

对于只读依赖属性，从 `RegisterReadOnly` 返回的值是 `DependencyPropertyKey`，通常不会使 `DependencyPropertyKey` 成为类的 `public` 成员。由于 WPF 属性系统不会在代码外部传播 `DependencyPropertyKey`，因此只读依赖属性具有比读写依赖属性更好的 `set` 安全性。

## 依赖属性和类构造函数

托管代码编程（通常通过代码分析工具强制执行）的一般原则是：类构造函数不应调用虚拟方法。这是因为基构造函数可以在派生类构造函数的初始化期间进行调用，并且基构造函数调用的虚拟方法可能会在完成派生类的初始化之前运行。从已派生自 `DependencyObject` 的类进行派生时，属性系统本身会在内部调用和公开虚拟方法。这些虚拟方法属于 WPF 属性系统服务。替代方法会使派生类参与值确定。为避免运行时初始化出现潜在问题，不应该在类的构造函数中设置依赖属性值，除非遵循特定的构造函数模式进行操作。有关详细信息，请参阅[DependencyObject 的安全构造函数模式](#)。

## 另请参阅

- [依赖属性概述](#)
- [依赖属性元数据](#)
- [控件创作概述](#)
- [集合类型依赖属性](#)
- [依赖属性的安全性](#)
- [XAML 加载和依赖属性](#)
- [DependencyObject 的安全构造函数模式](#)

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 依赖属性元数据 (WPF .NET)

项目 · 2023/10/13

Windows Presentation Foundation (WPF) 属性系统包括一个依赖属性元数据报告系统。通过元数据报告系统提供的信息超过了通过反射或通用公共语言运行时 (CLR) 特征提供的信息。注册依赖属性时，可以选择创建元数据并将元数据分配给它。如果从定义依赖属性的类派生，则可以替代继承依赖属性的元数据。如果将类添加为依赖属性的所有者，也可以替代继承依赖属性的元数据。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 如何使用元数据

可以通过查询依赖属性元数据来检查依赖属性的特征。当属性系统处理依赖属性时，它会访问该属性的元数据。依赖属性的元数据对象包含以下类型的信息：

- 依赖属性的默认值，当没有其他值（例如本地值、样式值或继承值）适用时由属性系统设置。有关依赖属性值的运行时分配期间值优先级的详细信息，请参阅[依赖属性值优先级](#)。
- 对所有者类型上的强制转换值回调和属性更改回调的引用。只能获取对具有 `public` 访问修饰符或在允许访问范围内的回调的引用。有关依赖属性回调的详细信息，请参阅[依赖属性回调和验证](#)。
- WPF 框架级依赖属性特征（如果依赖属性是 WPF 框架属性）。WPF 进程（如框架布局引擎和属性继承逻辑）将查询 WPF 框架级元数据。有关详细信息，请参阅[框架属性元数据](#)。

## 元数据 API

`PropertyMetadata` 类存储属性系统使用的大部分元数据。可以通过以下类型创建和分配元数据实例：

- 向属性系统注册依赖属性的类型。
- 从定义依赖属性的类继承的类型。
- 将自身添加为依赖属性所有者的类型。

如果某个类型在没有指定元数据的情况下注册了依赖属性，属性系统会将具有该类型的默认值的 `PropertyMetadata` 对象分配给依赖属性。

若要检索依赖属性的元数据，请对 `DependencyProperty` 标识符调用 `GetMetadata` 重载之一。元数据作为 `PropertyMetadata` 对象返回。

不同的体系结构领域存在派生自 `PropertyMetadata` 的更具体的元数据类。例如，`UIPropertyMetadata` 支持动画报告，`FrameworkPropertyMetadata` 支持 WPF 框架属性。依赖属性也可以注册到 `PropertyMetadata` 派生类中。尽管 `GetMetadata` 返回 `PropertyMetadata` 对象，但如果适用，你可以强制转换为派生类型，以检查特定于类型的属性。

`FrameworkPropertyMetadata` 公开的属性特征有时称为标志。创建 `FrameworkPropertyMetadata` 实例时，可以选择将枚举类型 `FrameworkPropertyMetadataOptions` 的实例传递给 `FrameworkPropertyMetadata` 构造函数。`FrameworkPropertyMetadataOptions` 允许按位组合指定元数据标记。

`FrameworkPropertyMetadata` 通过 `FrameworkPropertyMetadataOptions` 使构造函数签名保持合理的长度。注册依赖属性时，在 `FrameworkPropertyMetadataOptions` 上设置的元数据标记在 `FrameworkPropertyMetadata` 中作为 `Boolean` 属性公开，而不是作为标记的位组合公开，从而使查询元数据特征更加直观。

## 替代或创建新的元数据？

继承依赖属性时，可以选择通过替代元数据来更改依赖属性的特征。但是，可能无法始终通过替代元数据来完成依赖属性方案，有时需要在类中使用新的元数据定义自定义依赖属性。自定义依赖属性与 WPF 类型定义的依赖属性具有相同的功能。有关详细信息，请参阅[自定义依赖属性](#)。

不能替代的依赖属性的一个特征就是它的值类型。如果继承依赖属性具有所需的近似行为，但你的方案需要不同的值类型，请考虑实现自定义依赖属性。你可以通过派生类中的类型转换或其他实现来链接属性值。

## 替代元数据的方案

替代现有依赖属性元数据的示例方案包括：

- 更改默认值，这是一种常见方案。
- 更改或添加属性更改回调，如果继承依赖属性与其他依赖属性的交互方式不同于其基本实现，则可能需要这样做。支持代码和标记的编程模型的特征之一是可以按任意顺序设置属性值。此因素会影响你实现属性更改回调的方式。有关详细信息，请参阅[依赖属性回调和验证](#)。
- 更改 WPF 框架属性元数据选项。通常，元数据选项是在注册新依赖属性期间设置的，但你可以在 `OverrideMetadata` 或 `AddOwner` 调用中重新指定。有关替代框架属性元数据的详细信息，请参阅[指定 FrameworkPropertyMetadata](#)。有关如何在注册依赖属性时设置框架属性元数据选项的信息，请参阅[自定义依赖属性](#)。

#### ① 备注

由于验证回调不是元数据的一部分，因此无法通过替代元数据来更改它们。有关详细信息，请参阅[验证值回调](#)。

## 替代元数据

实现新的依赖属性时，可以使用 `Register` 方法的重载来设置其元数据。如果类继承了依赖属性，你可以使用 `OverrideMetadata` 方法替代继承的元数据值。例如，可以使用 `OverrideMetadata` 设置特定于类型的值。有关详细信息和代码示例，请参阅[替代依赖属性的元数据](#)。

`Focusable` 就是一个 WPF 依赖属性示例。`FrameworkElement` 类注册 `Focusable`。`Control` 类派生自 `FrameworkElement`，继承 `Focusable` 依赖属性，并替代继承属性元数据。替代操作会将默认属性值从 `false` 更改为 `true`，但保留其他继承元数据值。

由于大多数现有依赖属性不是虚拟属性，因此其继承实现会隐藏现有成员。替代元数据特征时，新的元数据值要么替换原始值，要么与之合并：

- 对于 `DefaultValue`，新值将替换现有的默认值。如果未在替代元数据中指定 `DefaultValue`，则该值来自在元数据中指定 `DefaultValue` 的最近上级。
- 对于 `PropertyChangedCallback`，默认合并逻辑将所有 `PropertyChangedCallback` 值存储在一个表中，并在属性更改时调用所有值。回调顺序取决于类深度，其中由层次结构中的基类注册的回调将首先运行。
- 对于 `CoerceValueCallback`，新值将替换现有的 `CoerceValueCallback` 值。如果未在替代元数据中指定 `CoerceValueCallback`，则该值来自在元数据中指定

`CoerceValueCallback` 的最近上级。

### ① 备注

默认合并逻辑由 `Merge` 方法实现。 你可以在继承依赖属性的派生类中指定自定义合并逻辑，方法是替代该类中的 `Merge`。

## 将类添加为所有者

若要“继承”在不同类层次结构中注册的依赖属性，请使用 `AddOwner` 方法。 当添加类不是从注册依赖属性的类型派生时，通常使用此方法。 在 `AddOwner` 调用中，添加类可以为继承依赖属性创建和分配特定于类型的元数据。 若要通过代码和标记成为属性系统中的完整参与者，添加类应实现以下公共成员：

- 依赖属性标识符字段。 依赖属性标识符的值是 `AddOwner` 调用的返回值。 此字段应该是 `DependencyProperty` 类型的 `public static readonly` 字段。
- 实现 `get` 和 `set` 访问器的 CLR 包装器。 通过使用属性包装器，依赖属性的使用者可以获取或设置依赖属性值，就像获取或设置任何其他 CLR 属性一样。 `get` 和 `set` 访问器通过 `DependencyObject.GetValue` 和 `DependencyObject.SetValue` 调用与底层属性系统交互，并以参数的形式传入依赖属性标识符。 以与注册自定义依赖属性时相同的方式实现包装器。 有关详细信息，请参阅[自定义依赖属性](#)

公开继承依赖属性的对象模型时，调用 `AddOwner` 的类与定义新自定义依赖属性的类具有相同的要求。 有关详细信息，请参阅[添加依赖属性的所有者类型](#)。

## 附加属性元数据

在 WPF 中，WPF 类型上大多数与 UI 相关的附加属性都作为依赖属性实现。 作为依赖属性实现的附加属性支持依赖属性概念，例如派生类可以替代的元数据。 附加属性的元数据通常与依赖属性的元数据没有什么不同。 你可以在替代类的实例上替代继承附加属性的默认值、属性更改回调和 WPF 框架属性。 有关详细信息，请参阅[附加属性元数据](#)

### ① 备注

始终使用 `RegisterAttached` 来注册在元数据中指定 `Inherits` 的属性。 尽管属性值继承看起来对非附加依赖项属性有效，但通过运行时树中特定对象-对象划分的非附加属性的值继承行为并未定义。`Inherits` 属性与非附加属性无关。 有关详细信息，

请参阅 `RegisterAttached(String, Type, Type, PropertyMetadata)` 和 `Inherits` 的备注部分。

## 添加类作为附加属性的所有者

若要从另一个类继承附加属性，但将其公开为你的类的非附加依赖属性：

- 调用 `AddOwner` 以将你的类添加为附加依赖属性的所有者。
- 将 `AddOwner` 调用的返回值分配给 `public static readonly` 字段，以用作依赖属性标识符。
- 定义 CLR 包装器，它将属性添加为类成员并支持非附加属性用法。

## 另请参阅

- [PropertyMetadata](#)
- [DependencyObject](#)
- [DependencyProperty](#)
- [GetMetadata](#)
- [AddOwner](#)
- [依赖属性概述](#)
- [框架属性元数据](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

#### 提出文档问题

#### 提供产品反馈

# 依赖属性回叫和验证 (WPF .NET)

项目 • 2023/10/13

本文介绍如何定义依赖属性并实现依赖属性回叫。 回叫支持值验证、值强制转换和其他属性值更改时所需的逻辑。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。 若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 验证值回叫

验证值回叫提供了一种方法，可以在属性系统应用新依赖属性值之前检查其是否有效。 如果值不符合验证条件，此回叫将引发异常。

在属性注册期间，验证值回叫只能分配给依赖属性一次。 注册依赖属性时，可以选择传递对 `Register(String, Type, Type, PropertyMetadata, ValidateValueCallback)` 方法的 `ValidateValueCallback` 引用。 验证值回叫不属于属性元数据，不可替代。

依赖属性的有效值是其应用的值。 当存在多个基于属性的输入时，有效值根据[属性值优先级](#)确定。 如果为依赖属性注册了验证值回叫，则属性系统将在值更改时调用其验证值回叫，并将新值作为对象传递。 回叫时，可以将值对象强制转换回属性系统中注册的类型，然后对其进行验证逻辑。 如果值对属性有效，则回叫返回 `true`，否则返回 `false`。

如果验证值回叫返回 `false`，则会引发异常，且不会应用新值。 应用程序编写器必须准备处理这些异常。 验证值回叫的一个常见用法是验证枚举值，或者在数值表示具有限制的度量值时对其进行约束。 验证值回叫在不同方案中由属性系统调用，包括：

- 对象初始化，在创建时应用默认值。
- 通过编程调用 `SetValue`。
- 指定新默认值的元数据替代。

验证值回叫没有指定设置新值的 `DependencyObject` 实例的参数。`DependencyObject` 的所有实例共享相同的验证值回叫，因此其无法用于验证特定于实例的方案。 有关详细信

息，请参阅 [ValidateValueCallback](#)。

以下示例展示了如何防止将类型为 `Double` 的属性设置为 `PositiveInfinity` 或 `NegativeInfinity`。

C#

```
public class Gauge1 : Control
{
    public Gauge1() : base() { }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty CurrentReadingProperty =
        DependencyProperty.Register(
            name: "CurrentReading",
            propertyType: typeof(double),
            ownerType: typeof(Gauge1),
            typeMetadata: new FrameworkPropertyMetadata(
                defaultValue: double.NaN,
                flags: FrameworkPropertyMetadataOptions.AffectsMeasure),
            validateValueCallback: new
            ValidateValueCallback(IsValidReading));

    // CLR wrapper with get/set accessors.
    public double CurrentReading
    {
        get => (double)GetValue(CurrentReadingProperty);
        set => SetValue(CurrentReadingProperty, value);
    }

    // Validate-value callback.
    public static bool IsValidReading(object value)
    {
        double val = (double)value;
        return !val.Equals(double.NegativeInfinity) &&
            !val.Equals(double.PositiveInfinity);
    }
}
```

C#

```
public static void TestValidationBehavior()
{
    Gauge1 gauge = new();

    Debug.WriteLine($"Test value validation scenario:");

    // Set allowed value.
    gauge.CurrentReading = 5;
    Debug.WriteLine($"Current reading: {gauge.CurrentReading}");
```

```
try
{
    // Set disallowed value.
    gauge.CurrentReading = double.PositiveInfinity;
}
catch (ArgumentException e)
{
    Debug.WriteLine($"Exception thrown by ValidateValueCallback:
{e.Message}");
}

Debug.WriteLine($"Current reading: {gauge.CurrentReading}");

// Current reading: 5
// Exception thrown by ValidateValueCallback: '∞' is not a valid value
for property 'CurrentReading'.
// Current reading: 5
}
```

## 属性更改回叫

属性更改回叫会在依赖属性的有效值发生更改时通知你。

属性更改回叫属于依赖属性元数据。如果从定义依赖属性的类派生，或者将类添加为依赖属性的所有者，则可以替代元数据。替代元数据时，可以选择提供新的 [PropertyChangedCallback](#) 引用。使用属性更改回叫来运行属性值更改时所需的逻辑。

与验证值回叫不同，属性更改回叫具有一个参数，用于指定设置新值的 [DependencyObject](#) 实例。下一个示例展示了属性更改回叫如何使用 [DependencyObject](#) 实例引用来触发强制值回叫。

## 强制值回叫

强制值回叫提供了一种方法，可在依赖属性的有效值即将更改时收到通知，以便在应用新值之前对其进行调整。除了由属性系统触发以外，还可以从代码中调用强制值回叫。

强制值回叫属于依赖属性元数据。如果从定义依赖属性的类派生，或者将类添加为依赖属性的所有者，则可以替代元数据。替代元数据时，可以选择提供对新 [CoerceValueCallback](#) 的引用。使用强制值回叫来评估新值，并在必要时强制转换这些值。如果发生强制转换，回叫将返回强制值，否则返回未更改的新值。

与属性更改回叫类似，强制值回叫有一个参数，用于指定设置新值的 [DependencyObject](#) 实例。下一个示例演示强制值回叫如何使用 [DependencyObject](#) 实例引用来强制转换属性值。

## ① 备注

无法强制转换默认属性值。 依赖属性在对象初始化时设置其默认值，或者在使用 `ClearValue` 清除其他值时设置默认值。

## 组合使用强制值和属性更改回叫

通过结合使用强制值回叫和属性更改回叫，可以在元素上创建属性之间的依赖关系。 例如，一个属性的更改会强制实施强制转换或重新评估另一个依赖属性。 下一个示例演示了一个常见方案：三个依赖属性，分别存储 UI 元素的当前值、最小值和最大值。 如果最大值发生更改，使其小于当前值，则当前值将设置为新的最大值。 而如果最小值发生变化，使其大于当前值，则将当前值将设置为新的最小值。 在示例中，当前值的 `PropertyChangedCallback` 显式调用 `CoerceValueCallback` 的最小值和最大值。

C#

```
public class Gauge2 : Control
{
    public Gauge2() : base() { }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty CurrentReadingProperty =
        DependencyProperty.Register(
            name: "CurrentReading",
            propertyType: typeof(double),
            ownerType: typeof(Gauge2),
            typeMetadata: new FrameworkPropertyMetadata(
                defaultValue: double.NaN,
                flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
                propertyChangedCallback: new
                    PropertyChangedCallback(OnCurrentReadingChanged),
                coerceValueCallback: new
                    CoerceValueCallback(CoerceCurrentReading)
            ),
            validateValueCallback: new ValidateValueCallback(IsValidReading)
        );

    // CLR wrapper with get/set accessors.
    public double CurrentReading
    {
        get => (double)GetValue(CurrentReadingProperty);
        set => SetValue(CurrentReadingProperty, value);
    }

    // Validate-value callback.
    public static bool IsValidReading(object value)
    {
```

```

        double val = (double)value;
        return !val.Equals(double.NegativeInfinity) &&
!val.Equals(double.PositiveInfinity);
    }

    // Property-changed callback.
    private static void OnCurrentReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
{
    depObj.CoerceValue(MinReadingProperty);
    depObj.CoerceValue(MaxReadingProperty);
}

    // Coerce-value callback.
    private static object CoerceCurrentReading(DependencyObject depObj,
object value)
{
    Gauge2 gauge = (Gauge2)depObj;
    double currentVal = (double)value;
    currentVal = currentVal < gauge.MinReading ? gauge.MinReading :
currentVal;
    currentVal = currentVal > gauge.MaxReading ? gauge.MaxReading :
currentVal;
    return currentVal;
}

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty MaxReadingProperty =
DependencyProperty.Register(
    name: "MaxReading",
    propertyType: typeof(double),
    ownerType: typeof(Gauge2),
    typeMetadata: new FrameworkPropertyMetadata(
        defaultValue: double.NaN,
        flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
        propertyChangedCallback: new
PropertyChangedCallback(OnMaxReadingChanged),
        coerceValueCallback: new CoerceValueCallback(CoerceMaxReading)
    ),
    validateValueCallback: new ValidateValueCallback(IsValidReading)
);

    // CLR wrapper with get/set accessors.
    public double MaxReading
{
    get => (double)GetValue(MaxReadingProperty);
    set => SetValue(MaxReadingProperty, value);
}

    // Property-changed callback.
    private static void OnMaxReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
{
    depObj.CoerceValue(MinReadingProperty);
}

```

```

        depObj.CoerceValue(CurrentReadingProperty);
    }

    // Coerce-value callback.
    private static object CoerceMaxReading(DependencyObject depObj, object
value)
    {
        Gauge2 gauge = (Gauge2)depObj;
        double maxVal = (double)value;
        return maxVal < gauge.MinReading ? gauge.MinReading : maxVal;
    }

    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata, and callbacks.
    public static readonly DependencyProperty MinReadingProperty =
DependencyProperty.Register(
    name: "MinReading",
    propertyType: typeof(double),
    ownerType: typeof(Gauge2),
    typeMetadata: new FrameworkPropertyMetadata(
        defaultValue: double.NaN,
        flags: FrameworkPropertyMetadataOptions.AffectsMeasure,
        propertyChangedCallback: new
PropertyChangedCallback(OnMinReadingChanged),
        coerceValueCallback: new CoerceValueCallback(CoerceMinReading)
    ),
    validateValueCallback: new ValidateValueCallback(IsValidReading));
}

// CLR wrapper with get/set accessors.
public double MinReading
{
    get => (double)GetValue(MinReadingProperty);
    set => SetValue(MinReadingProperty, value);
}

// Property-changed callback.
private static void OnMinReadingChanged(DependencyObject depObj,
DependencyPropertyChangedEventArgs e)
{
    depObj.CoerceValue(MaxReadingProperty);
    depObj.CoerceValue(CurrentReadingProperty);
}

// Coerce-value callback.
private static object CoerceMinReading(DependencyObject depObj, object
value)
{
    Gauge2 gauge = (Gauge2)depObj;
    double minVal = (double)value;
    return minVal > gauge.MaxReading ? gauge.MaxReading : minVal;
}
}

```

# 高级回叫方案

## 约束和所需值

如果强制转换依赖属性的本地设置值，则将未更改的本地设置值保留为所需值。如果根据其他属性值进行强制转换，则属性系统将在这些其他值更改时动态重新评估强制转换。在强制转换的约束下，属性系统将应用最接近所需值的值。如果强制转换条件不再适用，属性系统将还原所需值，前提是没有任何更高的优先级值处于活动状态。以下示例测试当前值、最小值和最大值方案中的强制转换。

C#

```
public static void TestCoercionBehavior()
{
    Gauge2 gauge = new()
    {
        // Set initial values.
        MinReading = 0,
        MaxReading = 10,
        CurrentReading = 5
    };

    Debug.WriteLine($"Test current/min/max values scenario:");

    // Current reading is not coerced.
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading is coerced to max value.
    gauge.MaxReading = 3;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading is coerced, but tracking back to the desired value.
    gauge.MaxReading = 4;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading reverts to the desired value.
    gauge.MaxReading = 10;
    Debug.WriteLine($"Current reading: " +
        $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

    // Current reading remains at the desired value.
    gauge.MinReading = 5;
    gauge.MaxReading = 5;
```

```

        Debug.WriteLine($"Current reading: " +
            $"{gauge.CurrentReading} (min: {gauge.MinReading}, max:
{gauge.MaxReading})");

        // Current reading: 5 (min=0, max=10)
        // Current reading: 3 (min=0, max=3)
        // Current reading: 4 (min=0, max=4)
        // Current reading: 5 (min=0, max=10)
        // Current reading: 5 (min=5, max=5)
    }
}

```

如果有多个属性以循环方式相互依赖，则可能会出现相当复杂的依赖场景。从技术上讲，复杂的依赖关系没有任何问题，只是大量的重新评估会降低性能。此外，UI 中公开的复杂依赖关系可能会使用户感到困惑。尽可能简单明了地处理 [PropertyChangedCallback](#) 和 [CoerceValueCallback](#)，不要过度约束。

## 取消值更改

通过从 [CoerceValueCallback](#) 返回 [UnsetValue](#)，可以拒绝属性值更改。此机制在异步启动属性值更改时很有用，但应用此机制后将对当前对象状态不再有效。另一种方案可能是根据值来源选择性地抑制值更改。在以下示例中，[CoerceValueCallback](#) 调用 [GetValueSource](#) 方法，该方法会返回一个带有 [BaseValueSource](#) 枚举的 [ValueSource](#) 结构，该枚举会标识新值的来源。

C#

```

// Coerce-value callback.
private static object CoerceCurrentReading(DependencyObject depObj, object
value)
{
    // Get value source.
    ValueSource valueSource =
        DependencyPropertyHelper.GetValueSource(depObj,
CurrentReadingProperty);

    // Reject any property value change that's a locally set value.
    return valueSource.BaseValueSource == BaseValueSource.Local ?
        DependencyProperty.UnsetValue : value;
}

```

## 另请参阅

- [ValidateValueCallback](#)
- [PropertyChangedCallback](#)
- [CoerceValueCallback](#)

- 依赖属性概述
- 依赖属性元数据
- 自定义依赖属性

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 只读依赖属性 (WPF .NET)

项目 • 2023/10/13

可以使用只读依赖属性来防止从代码外部设置属性值。本文讨论现有的只读依赖属性，以及创建自定义只读依赖属性的方案和技术。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 现有只读依赖属性

只读依赖属性通常报告状态，不应通过 `public` 访问器进行修改。例如，Windows Presentation Foundation (WPF) 框架将 `IsMouseOver` 属性实现为只读，因为它的值应仅由鼠标输入确定。如果 `IsMouseOver` 允许其他输入，则其值可能与鼠标输入不一致。虽然无法通过 `public` 访问器设置，但许多现有的只读依赖属性具有由多个输入确定的值。

## 只读依赖属性的使用

对于依赖属性通常提供解决方案的几种场景，只读依赖属性不适用。不适用的场景包括数据绑定、将样式应用于值、验证、动画和继承。但是，只读依赖属性可以用作样式中的属性触发器。例如，`IsMouseOver` 通常用于在鼠标悬停在控件上时触发对控件的背景、前景或其他可见属性的更改。WPF 属性系统检测并报告只读依赖属性的更改，从而支持属性触发功能。只读依赖属性在实现集合类型依赖属性时也很有用，其中只有集合元素需要是可写的，而不是集合对象本身。有关详细信息，请参阅[集合类型依赖属性](#)。

## ① 备注

只有依赖属性，而不是常规的公共语言运行时属性，可以用作样式中的属性触发器。

# 创建自定义只读依赖属性

在创建只读依赖属性之前，请检查[不适用的场景](#)。

创建只读依赖属性的过程在许多方面类似于创建读写依赖属性，但有以下区别：

- 注册只读属性时，需要调用 [RegisterReadOnly](#) 而不是 Register。
- 实现 CLR 属性包装器时，请确保它没有公共 `set` 访问器。
- `RegisterReadOnly` 返回 [DependencyPropertyKey](#) 而不是 [DependencyProperty](#)。将 [DependencyPropertyKey](#) 存储在非公共类成员中。

可以使用选择的任何逻辑来确定只读依赖属性的值。设置属性值的建议方法（最初或作为运行时逻辑的一部分）是使用接受 [DependencyPropertyKey](#) 类型参数的 [SetValue](#) 的重载。使用 `SetValue` 比绕过属性系统和直接设置支持字段更可取。

在应用程序中设置只读依赖属性值的方式和位置将影响分配给存储 [DependencyPropertyKey](#) 的类成员的访问级别。如果仅从注册依赖属性的类中设置属性值，可以使用 `private` 访问修饰符。对于依赖属性值相互影响的场景，可以使用配对 [PropertyChangedCallback](#) 和 [CoerceValueCallback](#) 回叫来触发值更改。有关详细信息，请参阅[依赖属性元数据](#)。

如果需要从注册只读依赖属性的类外部更改其值，可以为 [DependencyPropertyKey](#) 使用 `internal` 访问修饰符。例如，你可以从同一程序集中的事件处理程序调用 [SetValue](#)。以下示例定义了一个 [Aquarium](#) 类，该类调用 [RegisterReadOnly](#) 来创建只读依赖属性 `FishCount`。将 [DependencyPropertyKey](#) 分配给 `internal static readonly` 字段，因此同一程序集中的代码可以更改只读依赖属性值。

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, and property metadata.
    // Assign DependencyPropertyKey to a nonpublic field.
    internal static readonly DependencyPropertyKey FishCountPropertyKey =
        DependencyProperty.RegisterReadOnly(
            name: "FishCount",
            propertyType: typeof(int),
            ownerType: typeof(Aquarium),
            typeMetadata: new FrameworkPropertyMetadata());
}

// Declare a public get accessor.
public int FishCount =>
```

```
        (int)GetValue(FishCountPropertyKey.DependencyProperty);  
    }
```

由于 WPF 属性系统不会在代码外部传播 `DependencyPropertyKey`，因此只读依赖属性具有比读写依赖属性更好的写入安全性。若要将写入访问权限限制为具有对 `DependencyPropertyKey` 的引用，请使用只读依赖属性。

相反，无论为其分配什么访问修饰符，读写依赖属性的依赖属性标识符都可以通过属性系统访问。有关详细信息，请参阅[依赖属性安全性](#)。

## 另请参阅

- [依赖属性概述](#)
- [实现依赖属性](#)
- [自定义依赖属性](#)
- [集合类型依赖属性](#)
- [依赖属性的安全性](#)
- [WPF 中的样式和模板](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 框架属性元数据 (WPF .NET)

项目 • 2023/10/13

可以在 Windows Presentation Foundation (WPF) 框架级别为依赖属性设置框架属性元数据选项。当 WPF 演示 API 和可执行文件处理呈现和数据绑定时，WPF 框架级别规定适用。演示 API 和可执行文件查询依赖属性的 [FrameworkPropertyMetadata](#)。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 框架属性元数据类别

[FrameworkPropertyMetadata](#) 分为以下类别：

- 影响元素布局（特别是 [AffectsArrange](#)、[AffectsMeasure](#) 和 [AffectsRender](#) 元数据标志）的元数据。如果你的依赖属性实现影响视觉对象方面，并且要在类中实现 [MeasureOverride](#) 或 [ArrangeOverride](#)，则可设置这些标志。`MeasureOverride` 和 `ArrangeOverride` 方法向布局系统提供特定于实现的行为和呈现信息。当在依赖属性的元数据中将 `AffectsArrange`、`AffectsMeasure` 或 `AffectsRender` 设置为 `true` 并且该依赖属性的有效值发生更改时，WPF 属性系统将发起一个请求，使元素的视觉对象失效以触发重绘。
- 影响某个元素的父元素的布局（特别是 [AffectsParentArrange](#) 和 [AffectsParentMeasure](#) 元数据标志）的元数据。设置这些标志的 WPF 依赖属性的示例有 [FixedPage.Left](#) 和 [Paragraph.KeepWithNext](#)。
- 属性值继承元数据，特别是 [Inherits](#) 和 [OverridesInheritanceBehavior](#) 元数据标志。默认情况下，依赖属性不会继承值。[OverridesInheritanceBehavior](#) 允许将继承的路径也纳入可视化树，这对于某些控件复合方案来说是必需的。有关详细信息，请参阅[属性值继承](#)。

## ① 备注

属性值上下文中的术语“继承”特定于依赖属性，与托管代码类型和通过派生类类型的成员继承没有直接关系。在依赖属性的上下文中，这意味着子元素可以从父元素继承依赖属性值。

- 数据绑定元数据，特别是 `BindsTwoWayByDefault` 和  `IsNotDataBindable` 元数据标志。默认情况下，WPF 框架中的依赖属性支持单向绑定。考虑将双向绑定设置为报告状态且可通过用户操作来修改的属性的默认设置，例如  `IsSelected`。此外，当控件的用户希望某个属性来实现双向绑定时，请考虑将双向绑定设置为默认设置，例如  `TextBox.Text`。 `BindsTwoWayByDefault` 仅影响默认绑定模式。若要编辑绑定的数据流方向，请设置  `Binding.Mode`。当数据绑定没有用例时，可使用  `IsNotDataBindable` 禁用数据绑定。有关数据绑定的详细信息，请参阅[数据绑定概述](#)。
- 日志元数据，特别是  `Journal` 元数据标志。仅对于某些依赖属性（如  `SelectedIndex`）， `Journal` 标志的默认值才为  `true`。用户输入控件应为其值包含需要存储的用户选择的属性设置  `Journal` 标志。 `Journal` 标志由支持日志的应用程序或服务读取，包括 WPF 日志服务。有关存储导航步骤的信息，请参阅[导航概述](#)。

`FrameworkPropertyMetadata` 直接派生自 `UIPropertyMetadata`，并实现此处讨论的标志。除非专门设置，否则  `FrameworkPropertyMetadata` 标志的默认值为  `false`。

## 读取 FrameworkPropertyMetadata

若要检索依赖属性的元数据，请对  `DependencyProperty` 标识符调用  `GetMetadata`。 `GetMetadata` 调用返回  `PropertyMetadata` 对象。如果需要查询框架元数据值，请将  `PropertyMetadata` 强制转换为  `FrameworkPropertyMetadata`。

## 指定 FrameworkPropertyMetadata

注册依赖属性时，可以选择创建元数据并将元数据分配给它。所分配的元数据对象可以是  `PropertyMetadata` 或其派生类之一，例如  `FrameworkPropertyMetadata`。对于依赖于 WPF 演示文稿 API 和可执行文件进行呈现和数据绑定的依赖属性，请选择  `FrameworkPropertyMetadata`。更高级的选项是从  `FrameworkPropertyMetadata` 派生以创建具有更多标志的自定义元数据报告类。或者，可将  `UIPropertyMetadata` 用于影响 UI 呈现的非框架属性。

尽管元数据选项通常是在注册新依赖属性期间设置的，但你可以在  `OverrideMetadata` 或  `AddOwner` 调用中重新指定。替代元数据时，始终使用属性注册期间使用的相同元数据类型替代。

`FrameworkPropertyMetadata` 公开的属性特征有时称为标志。如果要创建 `FrameworkPropertyMetadata` 实例，可通过两种方式填充标志值：

1. 在 `FrameworkPropertyMetadataOptions` 枚举类型的实例上设置标志。

`FrameworkPropertyMetadataOptions` 允许按位“或”组合指定元数据标志。然后，使用具有 `FrameworkPropertyMetadata` 参数的构造函数实例化

`FrameworkPropertyMetadataOptions`，并传入 `FrameworkPropertyMetadataOptions` 实例。若要在将 `FrameworkPropertyMetadataOptions` 传入 `FrameworkPropertyMetadata` 构造函数后更改元数据标志，请更改新 `FrameworkPropertyMetadata` 实例上的相应属性。例如，如果设置了 `FrameworkPropertyMetadataOptions.NotBindable` 标志，则可通过将 `FrameworkPropertyMetadata.IsNotBindable` 设置为 `false` 来撤消该标志。

2. 使用不具有 `FrameworkPropertyMetadata` 参数的构造函数实例化

`FrameworkPropertyMetadataOptions`，然后对 `FrameworkPropertyMetadata` 设置适用的 `Boolean` 标志。在将 `FrameworkPropertyMetadata` 实例与依赖属性关联之前设置标志值，否则将出现 `InvalidOperationException`。

## 元数据替代行为

替代框架属性元数据时，更改的元数据值将替换原始值或与原始值合并：

- 对于 `PropertyChangedCallback`，默认合并逻辑将在一个表中保留以前的 `PropertyChangedCallback` 值，并在属性更改时调用所有值。回调顺序取决于类深度，其中由层次结构中的基类注册的回调将首先运行。继承的回调仅运行一次，并且由将它们添加到元数据中的类拥有。
- 对于 `DefaultValue`，新值将替换现有的默认值。如果未在替代元数据中指定 `DefaultValue`，并且现有的 `FrameworkPropertyMetadata` 已设置 `Inherits` 标志，则该默认值来自在元数据中指定 `DefaultValue` 的最近上级。
- 对于 `CoerceValueCallback`，新值将替换现有的 `CoerceValueCallback` 值。如果未在替代元数据中指定 `CoerceValueCallback`，则该值来自指定 `CoerceValueCallback` 的继承链中最近的上级。
- 对于 `FrameworkPropertyMetadata` 非继承标志，可使用 `true` 值替代默认 `false` 值。但对于 `Inherits`、`Journal`、`OverridesInheritanceBehavior` 和 `SubPropertiesDoNotAffectRender`，只能使用 `false` 值替代 `true` 值。

### ① 备注

默认合并逻辑由 `Merge` 方法实现。 你可以在继承依赖属性的派生类中指定自定义合并逻辑，方法是替代该类中的 `Merge`。

## 另请参阅

- [PropertyMetadata](#)
- [GetMetadata](#)
- [OverrideMetadata](#)
- [AddOwner](#)
- [依赖属性元数据](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# 依赖项属性的安全性 (WPF .NET)

项目 • 2023/10/13

由于可通过 Windows Presentation Foundation (WPF) 属性系统访问读写依赖项属性，因此该属性可有效地成为公共属性。因此，无法对读写依赖项属性值进行安全保证。WPF 属性系统可为只读依赖项属性提供更多的安全性，以便限制写入访问。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 属性包装器的访问和安全性

公共语言运行时 (CLR) 属性包装器通常包含在读写依赖项属性实现中，以简化属性值的获取或设置。如果包含，CLR 属性包装器是一种便捷方法，可实现 `GetValue` 和 `SetValue` 静态调用以与基础依赖项属性交互。从本质上讲，CLR 属性包装器将依赖项属性公开为由依赖项属性而不是私有字段支持的 CLR 属性。

应用安全机制并限制对 CLR 属性包装器的访问可能会阻止使用便利方法，但这些技术不会阻止直接调用 `GetValue` 或 `SetValue`。换句话说，始终可通过 WPF 属性系统访问读写依赖项属性。如果要实现读写依赖项属性，请避免限制对 CLR 属性包装器的访问。相反，将 CLR 属性包装声明为公共成员，以便调用方知道依赖项属性的真正访问级别。

## 依赖项属性的属性系统公开

WPF 属性系统通过其 `DependencyProperty` 标识符提供对读写依赖项属性的访问权限。标识符可用于 `GetValue` 和 `SetValue` 调用。即使静态标识符字段是非公共的，属性系统的几个方面也会返回 `DependencyProperty`，因为它存在于类或派生类的实例上。例如，`GetLocalValueEnumerator` 方法可返回具有本地设定值的依赖项属性实例的标识符。此外，还可以替代 `OnPropertyChanged` 虚拟方法来接收事件数据，该数据将报告已更改值的依赖项属性的 `DependencyProperty` 标识符。若要让调用方知道读写依赖项属性的真正访问级别，请将其标识符字段声明为公共成员。

## ① 备注

虽然将依赖项属性标识符字段声明为 `private` 可减少可读写依赖项属性变得可访问的方式，但根据 CLR 语言定义，该属性不会是私有的。

## 验证安全性

对 `ValidateValueCallback` 应用 `Demand` 和预期 `Demand` 失败时验证会失败并不是限制属性值更改的足够安全机制。另外，如果恶意调用方在应用程序域中操作，则这些调用方还可能会禁止通过 `ValidateValueCallback` 强制执行的新值验证。

## 对只读依赖项属性的访问

若要限制访问，请通过调用 `RegisterReadOnly` 方法将属性注册为只读依赖项属性。

`RegisterReadOnly` 方法会返回 `DependencyPropertyKey`，你可以将其分配给非公共类字段。对于只读依赖项属性，WPF 属性系统将仅提供对可引用 `DependencyPropertyKey` 的属性的写入访问权限。为了说明此行为，以下测试代码：

- 实例化实现读写和只读依赖项属性的类。
- 为每个标识符分配 `private` 访问修饰符。
- 仅实现 `get` 访问器。
- 使用 `GetLocalValueEnumerator` 方法通过 WPF 属性系统访问基础依赖项属性。
- 调用 `GetValue` 和 `SetValue` 以测试对每个依赖项属性值的访问。

C#

```
/// <summary>
/// Test get/set access to dependency properties exposed through the
WPF property system.
/// </summary>
public static void DependencyPropertyAccessTests()
{
    // Instantiate a class that implements read-write and read-only
    dependency properties.
    Aquarium _aquarium = new();
    // Access each dependency property using the LocalValueEnumerator
    method.
    LocalValueEnumerator localValueEnumerator =
    _aquarium.GetLocalValueEnumerator();
    while (localValueEnumerator.MoveNext())
    {
        DependencyProperty dp = localValueEnumerator.Current.Property;
        string dpType = dp.ReadOnly ? "read-only" : "read-write";
        // Test read access.
        Debug.WriteLine($"Attempting to get a {dpType} dependency
property value...");  

        Debug.WriteLine($"Value ({dpType}):  

{((int)_aquarium.GetValue(dp)})");
        // Test write access.
        try
        {
            Debug.WriteLine($"Attempting to set a {dpType} dependency
property value to 2...");
```

```

        _aquarium.SetValue(dp, 2);
    }
    catch (InvalidOperationException e)
    {
        Debug.WriteLine(e.Message);
    }
    finally
    {
        Debug.WriteLine($"Value ({dpType}): {(_aquarium.GetValue(dp)})");
    }
}

// Test output:

// Attempting to get a read-write dependency property value...
// Value (read-write): 1
// Attempting to set a read-write dependency property value to 2...
// Value (read-write): 2

// Attempting to get a read-only dependency property value...
// Value (read-only): 1
// Attempting to set a read-only dependency property value to 2...
// 'FishCountReadOnly' property was registered as read-only
// and cannot be modified without an authorization key.
// Value (read-only): 1
}
}

public class Aquarium : DependencyObject
{
    public Aquarium()
    {
        // Assign locally-set values.
        SetValue(FishCountProperty, 1);
        SetValue(FishCountReadOnlyPropertyKey, 1);
    }

    // Failed attempt to restrict write-access by assigning the
    // DependencyProperty identifier to a non-public field.
    private static readonly DependencyProperty FishCountProperty =
        DependencyProperty.Register(
            name: "FishCount",
            propertyType: typeof(int),
            ownerType: typeof(Aquarium),
            typeMetadata: new PropertyMetadata());
}

// Successful attempt to restrict write-access by assigning the
// DependencyPropertyKey to a non-public field.
private static readonly DependencyPropertyKey
FishCountReadOnlyPropertyKey =
    DependencyProperty.RegisterReadOnly(
        name: "FishCountReadOnly",
        propertyType: typeof(int),
        ownerType: typeof(Aquarium),

```

```
        typeMetadata: new PropertyMetadata()));

    // Declare public get accessors.
    public int FishCount => (int)GetValue(FishCountProperty);
    public int FishCountReadOnly =>
        (int)GetValue(FishCountReadOnlyPropertyKey.DependencyProperty);
}
```

## 另请参阅

- [DependencyProperty](#)
- [GetLocalValueEnumerator](#)
- [OnPropertyChanged](#)
- [自定义依赖属性](#)
- [实现依赖属性](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# DependencyObject 的安全构造函数模式 (WPF .NET)

项目 • 2023/10/13

托管代码编程（通常通过代码分析工具强制执行）的一般原则是：类构造函数不应调用可替代方法。如果基类构造函数调用了可替代方法，并且派生类替代了该方法，则派生类中的替代方法可以在派生类构造函数之前运行。如果派生类构造函数执行类初始化，则派生类方法可能会访问未初始化的类成员。为避免运行时初始化问题，依赖属性类应避免在类构造函数中设置依赖属性值。本文介绍如何以避免这些问题的方式实现 [DependencyObject 构造函数](#)。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 属性系统虚拟方法和回叫

依赖属性虚拟方法和回叫是 Windows Presentation Foundation (WPF) 属性系统的一部分，扩展了依赖属性的多功能性。

使用 [SetValue](#) 设置依赖项属性值等基本操作将调用 [OnPropertyChanged](#) 事件和可能的几个 WPF 属性系统回叫。

[OnPropertyChanged](#) 是 WPF 属性系统虚拟方法的一个示例，它可以被继承层次结构中具有 [DependencyObject](#) 的类替代。如果在自定义依赖属性类的实例化期间调用的构造函数中设置依赖属性值，并且从它派生的类替代 [OnPropertyChanged](#) 虚拟方法，则派生类 [OnPropertyChanged](#) 方法将在任何派生类构造函数之前运行。

[PropertyChangedCallback](#) 和 [CoerceValueCallback](#) 是 WPF 属性系统回叫的示例，它们可以由依赖属性类注册，并由派生自它们的类替代。如果在自定义依赖属性类的构造函数中设置依赖属性值，并且从它派生的类替代属性元数据中的这些回叫之一，则派生类回叫将在任何派生类构造函数之前运行。此问题与 [ValidateValueCallback](#) 不相关，因为它不是属性元数据的一部分，只能由注册类指定。

有关依赖属性回叫的详细信息，请参阅[依赖属性回叫和验证](#)。

## .NET 分析器

.NET Compiler Platform 分析器会检查 C# 或 Visual Basic 代码的代码质量和样式问题。如果在分析器规则 [CA2214](#) 处于活动状态时调用构造函数中的可替代方法，你将收到警告 `CA2214: Don't call overridable methods in constructors`。但是，当在构造函数中设置依赖项属性值时，该规则不会标记由基础 WPF 属性系统调用的虚拟方法和回叫。

## 由派生类引起的问题

如果**密封**自定义依赖属性类，或者知道你的类不会派生自该类，则派生类运行时初始化问题不是由该类引起的。但如果要创建可继承的依赖属性类，例如，要创建模板或可扩展控件库集，请避免调用可替代方法或从构造函数设置依赖属性值。

以下测试代码演示了不安全的构造函数模式，其中基类构造函数设置依赖属性值，从而触发对虚拟方法和回叫的调用。

C#

```
private static void TestUnsafeConstructorPattern()
{
    //Aquarium aquarium = new();
    //Debug.WriteLine($"Aquarium temperature (C): {aquarium.TempCelcius}");

    // Instantiate and set tropical aquarium temperature.
    TropicalAquarium tropicalAquarium = new(tempCelcius: 25);
    Debug.WriteLine($"Tropical aquarium temperature (C): " +
        $"{tropicalAquarium.TempCelcius}");

    /* Test output:
    Derived class static constructor running.
    Base class ValidateValueCallback running.
    Base class ValidateValueCallback running.
    Base class ValidateValueCallback running.
    Base class parameterless constructor running.
    Base class ValidateValueCallback running.
    Derived class CoerceValueCallback running.
    Derived class CoerceValueCallback: null reference exception.
    Derived class OnPropertyChanged event running.
    Derived class OnPropertyChanged event: null reference exception.
    Derived class PropertyChangedCallback running.
    Derived class PropertyChangedCallback: null reference exception.
    Aquarium temperature (C): 20
    Derived class parameterless constructor running.
    Derived class parameter constructor running.
    Base class ValidateValueCallback running.
    Derived class CoerceValueCallback running.
    Derived class OnPropertyChanged event running.
    Derived class PropertyChangedCallback running.
    Tropical aquarium temperature (C): 25
    */
}
```

```
}

public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
    // property type, owner type, property metadata with default value,
    // and validate-value callback.
    public static readonly DependencyProperty TempCelciusProperty =
        DependencyProperty.Register(
            name: "TempCelcius",
            propertyType: typeof(int),
            ownerType: typeof(Aquarium),
            typeMetadata: new PropertyMetadata(defaultValue: 0),
            validateValueCallback:
                new ValidateValueCallback(ValidateValueCallback));

    // Parameterless constructor.
    public Aquarium()
    {
        Debug.WriteLine("Base class parameterless constructor running.");

        // Set typical aquarium temperature.
        TempCelcius = 20;

        Debug.WriteLine($"Aquarium temperature (C): {TempCelcius}");
    }

    // Declare public read-write accessors.
    public int TempCelcius
    {
        get => (int)GetValue(TempCelciusProperty);
        set => SetValue(TempCelciusProperty, value);
    }

    // Validate-value callback.
    public static bool ValidateValueCallback(object value)
    {
        Debug.WriteLine("Base class ValidateValueCallback running.");
        double val = (int)value;
        return val >= 0;
    }
}

public class TropicalAquarium : Aquarium
{
    // Class field.
    private static List<int> _temperatureLog;

    // Static constructor.
    static TropicalAquarium()
    {
        Debug.WriteLine("Derived class static constructor running.");

        // Create a new metadata instance with callbacks specified.
        PropertyMetadata newPropertyMetadata = new(
```

```
        defaultValue: 0,
        propertyChangedCallback: new
PropertyChangedCallback(PropertyChangedCallback),
        coerceValueCallback: new
CoerceValueCallback(CoerceValueCallback));

    // Call OverrideMetadata on the dependency property identifier.
    TempCelciusProperty.OverrideMetadata(
        forType: typeof(TropicalAquarium),
        typeMetadata: newPropertyMetadata);
}

// Parameterless constructor.
public TropicalAquarium()
{
    Debug.WriteLine("Derived class parameterless constructor running.");
    s_temperatureLog = new List<int>();
}

// Parameter constructor.
public TropicalAquarium(int tempCelcius) : this()
{
    Debug.WriteLine("Derived class parameter constructor running.");
    TempCelcius = tempCelcius;
    s_temperatureLog.Add(tempCelcius);
}

// Property-changed callback.
private static void PropertyChangedCallback(DependencyObject depObj,
    DependencyPropertyChangedEventArgs e)
{
    Debug.WriteLine("Derived class PropertyChangedCallback running.");
    try
    {
        s_temperatureLog.Add((int)e.NewValue);
    }
    catch (NullReferenceException)
    {
        Debug.WriteLine("Derived class PropertyChangedCallback: null
reference exception.");
    }
}

// Coerce-value callback.
private static object CoerceValueCallback(DependencyObject depObj,
object value)
{
    Debug.WriteLine("Derived class CoerceValueCallback running.");
    try
    {
        s_temperatureLog.Add((int)value);
    }
    catch (NullReferenceException)
    {
        Debug.WriteLine("Derived class CoerceValueCallback: null
```

```

        reference exception.");
    }
    return value;
}

// OnPropertyChanged event.
protected override void
OnPropertyChanged(DependencyPropertyChangedEventArgs e)
{
    Debug.WriteLine("Derived class OnPropertyChanged event running.");
    try
    {
        s_temperatureLog.Add((int)e.NewValue);
    }
    catch (NullReferenceException)
    {
        Debug.WriteLine("Derived class OnPropertyChanged event: null
reference exception.");
    }

    // Mandatory call to base implementation.
    base.OnPropertyChanged(e);
}
}

```

在不安全构造函数模式测试中调用方法的顺序为：

1. 派生类静态构造函数，该构造函数替代 `Aquarium` 的依赖属性元数据以注册 `PropertyChangedCallback` 和 `CoerceValueCallback`。
2. 基类构造函数，用于设置新的依赖属性值，从而调用 `SetValue` 方法。`SetValue` 调用按以下顺序触发回叫和事件：
  - a. `ValidateValueCallback`，它是在基类中实现的。此回叫不是依赖属性元数据的一部分，不能通过替代元数据在派生类中实现。
  - b. `PropertyChangedCallback`，它通过替代依赖属性元数据在派生类中实现。此回叫在调用未初始化的类字段 `s_temperatureLog` 上的方法时会导致空引用异常。
  - c. `CoerceValueCallback`，它通过替代依赖属性元数据在派生类中实现。此回叫在调用未初始化的类字段 `s_temperatureLog` 上的方法时会导致空引用异常。
  - d. `OnPropertyChanged` 事件，它通过替代虚拟方法在派生类中实现。此事件在调用未初始化的类字段 `s_temperatureLog` 上的方法时会导致空引用异常。
3. 派生类无参数构造函数，该构造函数初始化 `s_temperatureLog`。
4. 派生类参数构造函数，用于设置新的依赖属性值，从而再次调用 `SetValue` 方法。由于 `s_temperatureLog` 现在已初始化，因此回叫和事件将运行，而不会导致空引用

异常。

使用安全构造函数模式可以避免这些初始化问题。

## 安全构造函数模式

可通过不同的方式解决测试代码中演示的派生类初始化问题，包括：

- 如果类可能用作基类，请避免在自定义依赖属性类的构造函数中设置依赖属性值。如果需要初始化依赖属性值，请考虑在依赖属性注册期间或替代元数据时将所需值设置为属性元数据中的默认值。
- 在使用派生类字段之前先对其进行初始化。例如，使用以下任一方法：
  - 在单个语句中实例化和分配实例字段。在前面的示例中，`List<int> s_temperatureLog = new();` 语句将避免延迟赋值。
  - 在派生类静态构造函数中执行赋值，该构造函数在任意基类构造函数之前运行。在前面的示例中，将赋值语句 `s_temperatureLog = new List<int>();` 放入派生类静态构造函数将避免延迟赋值。
  - 使用迟缓初始化和实例化，在需要时初始化对象。在前面的示例中，使用迟缓初始化和实例化来实例化和分配 `s_temperatureLog` 将避免延迟分配。若要了解详细信息，请参阅[迟缓初始化](#)。
- 避免在 WPF 属性系统回叫和事件中使用未初始化的类变量。

## 另请参阅

- [OnPropertyChanged](#)
- [PropertyChangedCallback](#)
- [CoerceValueCallback](#)
- [ValidateValueCallback](#)
- [依赖属性回调和验证](#)
- [自定义依赖属性](#)
- [依赖项属性概述](#)
- [依赖属性的安全性](#)

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 属性值继承 (WPF .NET)

项目 • 2023/10/13

属性值继承是 Windows Presentation Foundation (WPF) 属性系统的一项功能，适用于依赖属性。属性值继承允许元素树中的子元素从最近的父元素获取特定属性的值。由于父元素也可能通过属性值继承获得其属性值，因此系统可能递归回页面根。

WPF 属性系统默认不启用属性值继承，除非在依赖属性[元数据](#)中专门启用，否则值继承处于非活动状态。即使启用了属性值继承，子元素也仅会在缺少更高[优先级](#)值的情况下继承属性值。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## 通过元素树继承

属性值继承与面向对象的编程中的类继承不同，在面向对象编程中，派生类继承基类成员。这种继承在 WPF 中也处于活动状态，但在 XAML 中，继承的基类属性作为表示派生类的 XAML 元素的属性公开。

属性值继承是一种机制，通过这种机制，依赖属性值可以在包含该属性的元素树中从父元素传播到子元素。在 XAML 标记中，元素树作为嵌套元素可见。

以下示例演示 XAML 中的嵌套元素。WPF 通过属性[元数据](#)注册 `AllowDrop` 依赖属性到 `UIElement` 类，该属性启用属性值[继承](#)，并将默认值设置为 `false`。`AllowDrop` 依赖属性存在于 `Canvas`、`StackPanel` 和 `Label` 元素上，因为它们都派生自 `UIElement`。由于 `canvas1` 上的 `AllowDrop` 依赖属性被设置为 `true`，因此后代元素 `stackPanel1` 和 `label1` 将继承 `true` 作为它们的 `AllowDrop` 值。

### XAML

```
<Canvas x:Name="canvas1" Grid.Column="0" Margin="20" Background="Orange"
AllowDrop="True">
```

```
<StackPanel Name="stackPanel1" Margin="20" Background="Green">
    <Label Name="label1" Margin="20" Height="40" Width="40"
        Background="Blue"/>
</StackPanel>
</Canvas>
```

还可以通过将元素对象添加到另一个元素对象的子元素集合，以编程方式创建元素树。在运行时，属性值继承操作生成的对象树。在以下示例中，`stackPanel1` 将添加到 `canvas2` 的子集合中。同样，将 `label2` 添加到 `stackPanel1` 的子集合。由于 `canvas2` 上的 `AllowDrop` 依赖属性被设置为 `true`，因此后代元素 `stackPanel1` 和 `label2` 将继承 `true` 作为它们的 `AllowDrop` 值。

C#

```
Canvas canvas2 = new()
{
    AllowDrop = true
};
StackPanel stackPanel1 = new();
Label label1 = new();
canvas2.Children.Add(stackPanel1);
stackPanel1.Children.Add(label1);
```

## 属性值继承的实际应用

默认情况下，特定的 WPF 依赖属性启用了值继承，例如 `AllowDrop` 和 `FlowDirection`。通常，默认情况下启用值继承的属性在基 UI 元素类上实现，因此它们存在于派生类上。例如，由于 `AllowDrop` 在 `UIElement` 基类上实现，因此每个派生于 `UIElement` 的控件上也存在该依赖属性。WPF 允许对依赖属性进行值继承，用户可以方便地在父元素上设置属性值，并将该属性值传播到元素树中的后代元素。

属性值继承模型根据 [依赖属性值优先级](#) 分配继承和未继承的属性值。因此，如果子元素属性没有更高优先级的值（例如本地设置值或通过样式、模板或数据绑定获取的值），则父元素属性值将仅应用于子元素。

`FlowDirection` 依赖属性设置父元素中的文本和子 UI 元素的布局方向。通常，预期页面内文本和 UI 元素的流方向保持一致。由于在 `FlowDirection` 的属性[元数据](#)中启用了值继承，因此只需要在页面的元素树顶部设置一次值。在极少数情况下，混合流方向适用于页面，可以通过分配一个本地设置的值，在树中的元素上设置不同的流方向。然后，新的流方向将传播到该级别以下的后代元素。

## 使自定义属性可继承

通过在 `FrameworkPropertyMetadata` 实例中启用 `Inherits` 属性，然后向该元数据实例注册自定义依赖属性，可以使自定义依赖属性可继承。默认情况下，

`FrameworkPropertyMetadata` 中的 `Inherits` 设置为 `false`。使属性值可继承会影响性能，因此仅在需要该功能时才将 `Inherits` 设置为 `true`。

注册一个在元数据中启用 `Inherits` 的依赖属性时，请使用[注册附加属性](#)中所述的 `RegisterAttached` 方法。此外，为属性分配默认值，以便存在可继承的值。你可能还想在所有者类型上创建具有 `get` 和 `set` 访问器的属性包装器，就像对非附加依赖属性所做的那样。这样，就可以使用所有者或派生类型上的属性包装器来设置属性值。以下示例创建一个名为 `IsTransparent` 的依赖属性，启用了 `Inherits`，默认值为 `false`。该示例还包含一个带有 `get` 和 `set` 访问器的属性包装器。

C#

```
public class Canvas_IsTransparentInheritEnabled : Canvas
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata
    // (default value is 'false' and property value inheritance is enabled).
    public static readonly DependencyProperty IsTransparentProperty =
        DependencyProperty.RegisterAttached(
            name: "IsTransparent",
            propertyType: typeof(bool),
            ownerType: typeof(Canvas_IsTransparentInheritEnabled),
            defaultMetadata: new FrameworkPropertyMetadata(
                defaultValue: false,
                flags: FrameworkPropertyMetadataOptions.Inherits));

    // Declare a get accessor method.
    public static bool GetIsTransparent(Canvas element)
    {
        return (bool)element.GetValue(IsTransparentProperty);
    }

    // Declare a set accessor method.
    public static void SetIsTransparent(Canvas element, bool value)
    {
        element.SetValue(IsTransparentProperty, value);
    }

    // For convenience, declare a property wrapper with get/set accessors.
    public bool IsTransparent
    {
        get => (bool)GetValue(IsTransparentProperty);
        set => SetValue(IsTransparentProperty, value);
    }
}
```

附加属性在概念上类似于全局属性。在任何 [DependencyObject](#) 上检查其值并获得有效的结果。附加属性的典型方案是针对子元素设置属性值，如果相关属性隐式地作为一个附加属性出现在树中的每个 [DependencyObject](#) 元素上，则该方案会更为有效。

## 跨树边界继承属性值

属性继承通过遍历元素树来工作。此树通常与逻辑树并行。但是，每当在定义元素树的标记中包括 WPF 核心级别对象（如 [Brush](#)）时，都会随即创建一个不连续的逻辑树。真正的逻辑树在概念上不会通过 [Brush](#) 进行扩展，因为逻辑树是 WPF 框架级别概念。可以使用 [LogicalTreeHelper](#) 的帮助程序方法来分析和查看逻辑树的盘区。只要可继承的属性注册为附加属性，且没有遇到有意的继承阻止边界（如 [Frame](#)），那么属性值继承就能够通过不连续的逻辑树传递继承的值。

### ① 备注

尽管属性值继承看起来对非附加依赖项属性有效，但通过运行时树中特定元素边界的非附加属性的继承行为并未定义。每当在属性元数据中指定 [Inherits](#) 时，请使用 [RegisterAttached](#) 注册属性。

## 另请参阅

- [依赖属性元数据](#)
- [附加属性概述](#)
- [注册附加属性](#)
- [自定义依赖属性](#)
- [依赖项属性值优先级](#)
- [框架属性元数据](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

#### 提出文档问题

#### 提供产品反馈

# XAML 加载和依赖属性 (WPF .NET)

项目 • 2023/10/13

其 Extensible Application Markup Language (XAML) 处理器的 Windows Presentation Foundation (WPF) 实现本质上是依赖属性感知的。因此，XAML 处理器使用 WPF 属性系统方法来加载 XAML 和处理依赖属性特性，并通过使用 WPF 属性系统方法（如 [GetValue](#) 和 [SetValue](#)）完全绕过依赖属性包装器。因此，如果将自定义逻辑添加到自定义依赖属性的属性包装器，则在 XAML 中设置属性值时，XAML 处理器不会调用该逻辑。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 先决条件

本文假定你对依赖属性有基本的了解，并且已阅读[依赖属性概述](#)。若要理解本文中的示例，还应当熟悉 Extensible Application Markup Language (XAML) 并知道如何编写 WPF 应用程序。

## WPF XAML 加载程序性能

对于 WPF XAML 处理器来说，如果直接调用 [SetValue](#) 来设置依赖属性的值而不是使用依赖属性的属性包装器，其计算成本更低。

如果 XAML 处理器确实使用了属性包装器，则需要仅根据标记中指示的类型和成员关系来推断支持代码的整个对象模型。虽然可以通过使用 `xmlns` 和程序集特性的组合从标记中识别类型，但若要识别成员、确定哪些成员可以设置为特性以及解析支持的属性值类型，都需要使用  [PropertyInfo](#) 进行广泛的反射。

WPF 属性系统维护在给定  [DependencyObject](#) 派生类型上实现的依赖属性的存储表。XAML 处理器使用该表来推断依赖属性的依赖属性标识符。例如，按照约定，名为 `ABC` 的依赖属性的依赖属性标识符为 `ABCProperty`。XAML 处理器可以通过使用依赖属性标识符对依赖属性包含的类型调用  `SetValue` 方法，来有效地设置任何依赖属性的值。

有关依赖属性包装器的详细信息，请参阅[自定义依赖属性](#)。

## 自定义依赖属性的影响

WPF XAML 处理器绕过属性包装器并直接调用 `SetValue` 来设置依赖属性值。因此，请避免在自定义依赖属性的 `set` 访问器中放置任何额外的逻辑，因为在 XAML 中设置属性值时，该逻辑不会运行。`set` 访问器应仅包含一个 `SetValue` 调用。

同样，获取属性值的 WPF XAML 处理器的各个方面都会绕过属性包装器并直接调用 `GetValue`。因此，还应避免在自定义依赖属性的 `get` 访问器中放置任何额外的逻辑，因为在 XAML 中读取属性值时，该逻辑不会运行。`get` 访问器应仅包含一个 `GetValue` 调用。

## 带包装器的依赖属性示例

以下示例显示了带属性包装器的推荐依赖属性定义。依赖属性标识符存储为 `public static readonly` 字段，并且除了支持依赖属性值的必要 WPF 属性系统方法之外，`get` 和 `set` 访问器不包含其他任何代码。如果需要在依赖属性的值发生更改时运行某段代码，请考虑将该代码置于依赖属性的 `PropertyChangedCallback` 中。有关详细信息，请参阅[属性更改回叫](#)。

C#

```
// Register a dependency property with the specified property name,
// property type, owner type, and property metadata. Store the dependency
// property identifier as a public static readonly member of the class.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
        name: "AquariumGraphic",
        propertyType: typeof(Uri),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata(
            defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
            flags: FrameworkPropertyMetadataOptions.AffectsRender,
            propertyChangedCallback: new
            PropertyChangedCallback(OnUriChanged))
    );

// Property wrapper with get & set accessors.
public Uri AquariumGraphic
{
    get => (Uri)GetValue(AquariumGraphicProperty);
    set => SetValue(AquariumGraphicProperty, value);
}

// Property-changed callback.
private static void OnUriChanged(DependencyObject dependencyObject,
    DependencyPropertyChangedEventArgs e)
{
    // Some custom logic that runs on effective property value change.
    Uri newValue = (Uri)dependencyObject.GetValue(AquariumGraphicProperty);
```

```
        Debug.WriteLine($"OnUriChanged: {newValue}");  
    }
```

## 另请参阅

- [依赖属性概述](#)
- [WPF 中的 XAML](#)
- [自定义依赖属性](#)
- [依赖属性元数据](#)
- [集合类型依赖属性](#)
- [依赖属性的安全性](#)
- [DependencyObject 的安全构造函数模式](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何实现依赖属性 (WPF .NET)

项目 • 2023/10/13

本文介绍如何通过使用 `DependencyProperty` 字段来支持公共语言运行时 (CLR) 属性，从而实现依赖属性。依赖属性支持多个高级 Windows Presentation Foundation (WPF) 属性系统功能。这些功能包括样式、数据绑定、继承、动画和默认值。如果希望定义的属性支持这些功能，请将属性实现为依赖属性。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 示例

以下示例演示如何通过调用 `Register` 方法注册依赖属性。`Register` 方法返回一个称为依赖属性标识符的 `DependencyProperty` 实例。标识符存储在 `static readonly` 字段中，并保存依赖属性的名称和特征。

标识符字段必须遵循命名约定 `<property name>Property`。例如，如果使用名称 `Location` 注册依赖属性，则应将标识符字段命名为 `LocationProperty`。如果不遵循此命名模式，则 WPF 设计器可能无法正确地报告属性，而且属性系统样式应用程序的某些方面可能不会以预期的方式工作。

在以下示例中，依赖属性及其 CLR 访问器的 `name` 为 `HasFish`，因此标识符字段命名为 `HasFishProperty`。依赖属性类型为 `Boolean`，注册依赖属性的所有者类型为 `Aquarium`。

可为依赖属性指定默认元数据。此示例为 `HasFish` 依赖属性设置 `false` 的默认值。

C#

```
public class Aquarium : DependencyObject
{
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.Register(
            name: "HasFish",
            propertyType: typeof(bool),
            ownerType: typeof(Aquarium),
            typeMetadata: new FrameworkPropertyMetadata(defaultValue:
false));

    public bool HasFish
    {
        get => (bool)GetValue(HasFishProperty);
```

```
    set => SetValue(HasFishProperty, value);
}
}
```

若要深入了解实现依赖属性而非仅使用私有字段支持 CLR 属性的原因及其实现方式，请参阅[依赖属性概述](#)。

## 另请参阅

- [依赖属性概述](#)
- [操作说明主题](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

### 提出文档问题

### 提供产品反馈

# 如何注册附加属性 (WPF .NET)

项目 • 2023/10/13

本文介绍如何注册附加属性并提供公共访问器，以便通过 Extensible Application Markup Language (XAML) 和代码访问附加属性。附加属性支持对任何 XAML 元素设置额外的属性/值对，即使该元素没有在其对象模型中定义这些额外的属性。额外的属性可进行全局访问。附加属性通常定义为没有常规属性包装器的依赖属性的专用形式。Windows Presentation Foundation (WPF) 类型的大多数附加属性也作为依赖属性实现。可以在任何 [DependencyObject](#) 派生类型上创建依赖属性。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 示例

以下示例显示如何使用 [RegisterAttached](#) 方法将附加属性注册为依赖属性。提供程序类可以选择在属性元数据中指定默认值。有关属性元数据的详细信息，请参阅[依赖属性元数据](#)。在此示例中，`HasFish` 属性具有 `Boolean` 值类型，其默认值设置为 `false`。

附加属性的提供程序类必须提供遵循命名约定 `Get<property name>` 和 `Set<property name>` 的静态 `get/set` 访问器方法。XAML 读取器使用访问器识别附加属性的 XAML 特性，并将其值解析为适当的类型。即使附加属性未注册为依赖属性，这些访问器也是必需的。

C#

```
public class Aquarium : UIElement
{
    // Register an attached dependency property with the specified
    // property name, property type, owner type, and property metadata.
    public static readonly DependencyProperty HasFishProperty =
        DependencyProperty.RegisterAttached(
            "HasFish",
            typeof(bool),
            typeof(Aquarium),
            new FrameworkPropertyMetadata(defaultValue: false,
                flags: FrameworkPropertyMetadataOptions.AffectsRender)
        );

    // Declare a get accessor method.
    public static bool GetHasFish(UIElement target) =>
        (bool)target.GetValue(HasFishProperty);
```

```
// Declare a set accessor method.  
public static void SetHasFish(UIElement target, bool value) =>  
    target.SetValue(HasFishProperty, value);  
}
```

## 另请参阅

- [DependencyProperty](#)
- [附加属性概述](#)
- [依赖属性概述](#)
- [自定义依赖属性](#)
- [操作说明主题](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何覆盖依赖属性的元数据 (WPF .NET)

项目 • 2023/10/13

从定义依赖属性的类派生时，就继承了依赖属性及其元数据。本文介绍如何通过调用 [OverrideMetadata](#) 方法来重写继承的依赖属性的元数据。重写元数据可以修改继承的依赖属性的特征，以匹配特定于子类的要求。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 背景

定义依赖属性的类可以在 [PropertyMetadata](#) 或其派生类型之一（例如 [FrameworkPropertyMetadata](#)）中指定其特征。其中一个特征是依赖属性的默认值。许多定义依赖属性的类都在依赖属性注册期间指定属性元数据。如果在注册期间未指定元数据，则 WPF 属性系统会分配一个具有默认值的 [PropertyMetadata](#) 对象。通过类继承继承依赖属性的派生类可以选择重写任何依赖属性的原始元数据。通过这种方式，派生类可以选择性地修改依赖属性特性以满足类的要求。调用 [OverrideMetadata\(Type, PropertyMetadata\)](#) 时，派生类将其自己的类型指定为第一个参数，并将元数据实例指定为第二个参数。

重写依赖属性上的元数据的派生类必须在属性系统使用该属性之前执行此操作。当注册属性的类的任何实例被实例化时，就会使用依赖属性。为帮助满足此要求，派生类应在其静态构造函数中调用 [OverrideMetadata](#)。在实例化其所有者类型后重写依赖属性的元数据不会引发异常，但会导致属性系统中的行为不一致。此外，派生类型不能多次重写依赖属性的元数据，并且尝试这样做将引发异常。

## 示例

在以下示例中，派生类 [TropicalAquarium](#) 重写从基类 [Aquarium](#) 继承的依赖属性的元数据。元数据类型为 [FrameworkPropertyMetadata](#)，支持与 UI 相关的 WPF 框架特征，例如 [AffectsRender](#)。派生类不会重写继承的 [AffectsRender](#) 标志，但会更新派生类实例上 [AquariumGraphic](#) 的默认值。

C#

```
public class Aquarium : DependencyObject
{
    // Register a dependency property with the specified property name,
```

```

// property type, owner type, and property metadata.
public static readonly DependencyProperty AquariumGraphicProperty =
    DependencyProperty.Register(
        name: "AquariumGraphic",
        propertyType: typeof(Uri),
        ownerType: typeof(Aquarium),
        typeMetadata: new FrameworkPropertyMetadata(
            defaultValue: new Uri("http://www.contoso.com/aquarium-
graphic.jpg"),
            flags: FrameworkPropertyMetadataOptions.AffectsRender)
    );

// Declare a read-write CLR wrapper with get/set accessors.
public Uri AquariumGraphic
{
    get => (Uri)GetValue(AquariumGraphicProperty);
    set => SetValue(AquariumGraphicProperty, value);
}

```

C#

```

public class TropicalAquarium : Aquarium
{
    // Static constructor.
    static TropicalAquarium()
    {
        // Create a new metadata instance with a modified default value.
        FrameworkPropertyMetadata newPropertyMetadata = new(
            defaultValue: new Uri("http://www.contoso.com/tropical-aquarium-
graphic.jpg"));

        // Call OverrideMetadata on the dependency property identifier.
        // Pass in the type for which the new metadata will be applied
        // and the new metadata instance.
        AquariumGraphicProperty.OverrideMetadata(
            forType: typeof(TropicalAquarium),
            typeMetadata: newPropertyMetadata);
    }
}

```

## 另请参阅

- [DependencyProperty](#)
- [依赖属性元数据](#)
- [依赖属性概述](#)
- [自定义依赖属性](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# XAML 资源概述 (WPF .NET)

项目 • 2023/10/13

资源是可以在应用中的不同位置重复使用的对象。 资源的示例包括画笔和样式。 本概述介绍如何使用 Extensible Application Markup Language (XAML) 中的资源。 你还可以使用代码创建和访问资源。

## ① 备注

本文所述的 XAML 资源与应用资源不同，后者通常指添加到应用中的文件，例如内容、数据或嵌入式文件。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 使用 XAML 中的资源

下面的示例将 `SolidColorBrush` 定义为页面根元素上的资源。 该示例随后引用资源，并使用它来设置多个子元素的属性，其中包括 `Ellipse`、`TextBlock` 和 `Button`。

XAML

```
<Window x:Class="resources.ResExample"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ResExample" Height="400" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
        <Style TargetType="Border">
            <Setter Property="Background" Value="#4E1A3D" />
            <Setter Property="BorderThickness" Value="5" />
            <Setter Property="BorderBrush">
                <Setter.Value>
                    <LinearGradientBrush>
                        <GradientStop Offset="0.0" Color="#4E1A3D"/>
                        <GradientStop Offset="1.0" Color="Salmon"/>
                    </LinearGradientBrush>
                </Setter.Value>
            </Setter>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
        </Style>
    </Window.Resources>
    <Grid>
        <Ellipse Fill="Red" HorizontalAlignment="Center" VerticalAlignment="Center" Width="100" Height="100" />
        <TextBlock Text="Hello, World!" Margin="10" Style="{StaticResource TitleText}"/>
        <Button Content="Click Me!" Margin="10" Click="Button_Click" />
    </Grid>
</Window>
```

```

        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Margin" Value="0,10,10,10"/>
    </Style>
    <Style TargetType="TextBlock" x:Key="Label">
        <Setter Property="HorizontalAlignment" Value="Right"/>
        <Setter Property="FontSize" Value="13"/>
        <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
        <Setter Property="FontFamily" Value="Arial"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Margin" Value="0,3,10,0"/>
    </Style>
</Window.Resources>

<Border>
    <StackPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock HorizontalAlignment="Right" FontSize="36"
Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />
        <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
        <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
    </StackPanel>
</Border>

</Window>

```

每个框架级元素（[FrameworkElement](#) 或 [FrameworkContentElement](#)）都具有 [Resources](#) 属性，该属性是包含已定义资源的 [ResourceDictionary](#) 类型。你可以在任何元素上定义资源，例如 [Button](#)。但是，最常在根元素上定义资源，本示例中的根元素为 [Window](#)。

资源字典中的每个资源都必须具有唯一键。在标记中定义资源时，可通过 [x:Key 指令](#) 来分配唯一键。通常情况下，这个键是一个字符串；但是，也可使用相应的标记扩展将其设置为其他对象类型。资源的非字符串键用于 WPF 中的某些功能区，尤其是样式、组件资源和数据样式。

你可以使用具有资源标记扩展语法（指定资源的键名）的已定义资源。例如，将资源用作另一个元素上的属性的值。

#### XAML

```

<Button Background="{StaticResource MyBrush}"/>
<Ellipse Fill="{StaticResource MyBrush}"/>

```

在前面的示例中，如果 XAML 加载程序处理 [Button](#) 上 [Background](#) 属性的值 `{StaticResource MyBrush}`，则资源查找逻辑会首先检查 [Button](#) 元素的资源字典。如果 [Button](#) 没有资源键 `MyBrush` 的定义（在该示例中没有；其资源集合为空），则查找逻辑

接下来会检查 `Button` 的父元素。如果未在父级上定义资源，会继续向上检查对象的逻辑树，直到找到它。

如果在根元素上定义资源，则逻辑树中的所有元素（如 `Window` 或 `Page`）都可以访问它。而且，你可以重复使用相同的资源来设置接受与该资源所表示类型相同的类型的所有属性的值。在前面的示例中，同一 `MyBrush` 资源设置两个不同的属性：`Button.Background` 和 `Rectangle.Fill`。

## 静态和动态资源

资源可引用为静态资源或动态资源。可通过使用 [StaticResource 标记扩展](#) 或 [DynamicResource 标记扩展](#) 创建引用。标记扩展是 XAML 的一项功能，可以通过使用标记扩展来处理属性字符串并将对象返回到 XAML 加载程序，从而指定对象引用。有关标记扩展行为的详细信息，请参阅 [标记扩展和 WPF XAML](#)。

使用标记扩展时，通常会以字符串的形式提供一个或多个由该特定标记扩展处理的参数。[StaticResource 标记扩展](#) 通过在所有可用的资源字典中查找键值来处理键。处理在加载期间进行，即加载过程需要分配属性值时。[DynamicResource 标记扩展](#) 则通过创建表达式来处理键，而且表达式会保持未计算状态，直至应用运行为止。当应用实际运行时，表达式会进行计算以提供一个值。

在引用某个资源时，下列注意事项可能会对于使用静态资源引用还是使用动态资源引用产生影响：

- 确定如何为应用创建资源的整体设计（在每页上、在应用程序中、在宽松的 XAML 中或在仅包含资源的程序集中）时，请考虑以下事项：
- 应用的功能。实时更新资源是否为应用要求的一部分？
- 该资源引用类型的相应查找行为。
- 特定的属性或资源类型，以及这些类型的本机行为。

## 静态资源

在以下情况下，最适合使用静态资源引用：

- 应用设计将其大多数资源集中到页面或应用程序级资源字典中。

静态资源引用不基于运行时行为（例如重载页面）重新计算。因此，根据资源和应用设计，如果避免不必要的使用大量动态资源引用，可能会一定程度地提高性能。

- 要设置不在 `DependencyObject` 或 `Freezable` 上的属性的值。

- 你要创建一个资源字典，该字典编译为在应用间共享的 DLL。
- 要为自定义控件创建主题，并要定义在主题中使用的资源。

在这种情况下，通常不需要动态资源引用查找行为。所以，请改用静态资源引用行为，使查找是可预测的并且独立于主题。使用动态资源引用时，即使主题中的引用也会在运行时前保持未计算状态。而且，主题可能会得到应用，但某个本地元素仍会重新定义主题正尝试引用的键，并且该本地元素在查找期间会排在主题之前。如果发生这种情况，主题行为将不会按预期进行。

- 要使用资源设置大量依赖属性。依赖属性会通过属性系统启用有效值缓存功能；因此，如果为可在加载时进行计算的依赖属性提供了值，则该依赖属性不必检查是否存在重新计算的表达式并可返回最后一个有效值。此项技术可以提高性能。
- 想为所有使用者更改基础资源，或想通过使用 [x:Shared 属性](#) 为每个使用者维护单独的可写实例。

## 静态资源查找行为

下面介绍属性或元素引用静态资源时自动发生的查找过程：

1. 查找进程在用于设置属性的元素所定义的资源字典中查找请求的键。
2. 查找过程随后会向上遍历逻辑树，以查找父元素及其资源字典。此过程到达根元素后才会停止。
3. 检查应用资源。应用资源就是 [Application](#) 对象为 WPF 应用定义的资源字典中的资源。

从资源字典中进行的静态资源引用必须引用已在资源引用前进行过词法定义的资源。静态资源引用无法解析前向引用。因此，请设计资源字典的结构，以便在每个相应资源字典的开头或邻近开头的位置定义资源。

静态资源查找可以扩展到主题或系统资源中，但此查找受支持只是因为 XAML 加载程序推迟了请求。为了让页面加载时的运行时主题正确地应用到应用，这种延迟是必需的。但是，不建议使用对已知仅在主题中存在或作为系统资源存在的键的静态资源引用，因为如果用户实时更改主题，不会重新计算此类引用。请求主题或系统资源时，动态资源引用更为可靠。例外情况是当主题元素自身请求另一个资源。出于上述原因，这些引用应该是静态资源引用。

因找不到静态资源引用而引发的异常行为各不相同。如果资源被延迟，则异常会在运行时发生。如果资源未延迟，则加载时会发生异常。

# 动态资源

在以下情况下，最适合使用动态资源：

- 资源（包括系统资源或用户可设置的资源）的值取决于直到运行时才知道的条件。例如，你可以创建 setter 值（引用由 `SystemColors`、`SystemFonts` 或 `SystemParameters` 公开的系统属性）。这些值是真正的动态值，因为它们最终来自用户和操作系统的运行时环境。或许还拥有可能会发生变化的应用程序级主题，而页面级资源访问也必须捕获其中的变化。
- 要为自定义控件创建或引用主题样式。
- 打算在应用生存期内调整 `ResourceDictionary` 的内容。
- 拥有存在相互依赖关系且可能需要进行前向引用的复杂资源结构。静态资源引用不支持前向引用，但动态资源引用支持，因为资源在运行时之前不需要计算，所以前向引用是一个不相关的概念。
- 要引用从编译或工作集的角度来看很大的资源，而且该资源在页面加载时可能不会立即使用。页面加载时，始终会从 XAML 加载静态资源引用。但是，动态资源引用在使用前不会加载。
- 要创建的样式的 setter 值可能来自受主题或其他用户设置影响的其他值。
- 要将资源应用于可能会在应用生存期内在逻辑树中重定父级的元素。父级更改后，资源查找范围也可能会随之更改；因此，如果希望重定父级的元素的资源基于新范围重新进行计算，请始终使用动态资源引用。

## 动态资源查找行为

如果调用 `FindResource` 或 `SetResourceReference`，则动态资源引用的资源查找行为会与代码中的查找行为并行执行：

1. 查找在用于设置属性的元素所定义的资源字典中查找请求的键：
  - 如果元素定义 `Style` 属性，则该元素的 `System.Windows.FrameworkElement.Style` 将检查其 `Resources` 字典。
  - 如果元素定义 `Template` 属性，则检查该元素的 `System.Windows.FrameworkTemplate.Resources` 字典。
2. 查找会向上遍历逻辑树，以查找父元素及其资源字典。此过程到达根元素后才会停止。

3. 检查应用资源。应用资源就是 [Application](#) 对象为 WPF 应用定义的资源字典中的资源。
4. 检查主题资源字典中当前处于活动状态的主题。如果主题在运行时发生更改，则会重新计算值。
5. 检查系统资源。

异常行为（如果有）各不相同：

- 如果 [FindResource](#) 调用请求了某个资源但未找到该资源，则会引发异常。
- 如果 [TryFindResource](#) 调用请求了某个资源但未找到该资源，不会引发任何异常，并且返回的值为 `null`。如果要设置的属性不接受 `null`，则仍有可能引发更深的异常（取决于要设置的单独属性）。
- 如果 XAML 中的动态资源引用请求了某个资源但未找到该资源，则行为取决于常规属性系统。常规行为即存在资源的级别上没有发生属性设置操作时执行的行为。例如，如果尝试使用无法计算的资源来设置个别按钮元素上的背景，则值设置操作不会产生任何结果，但有效值可能仍来自属性系统和值优先级中的其他参与者。例如，背景值可能仍来自在本地定义的某个按钮样式，或来自主题样式。对于并非由主题样式定义的属性，资源计算失败后的有效值可能来自属性元数据中的默认值。

## 限制

动态资源引用存在一些重要限制。必须至少满足以下条件之一：

- 要设置的属性必须是 [FrameworkElement](#) 或 [FrameworkContentElement](#) 上的属性。该属性必须由 [DependencyProperty](#) 支持。
- 该引用用于 `Style.Setter` 内的值。
- 要设置的属性必须是 [Freezable](#)（以 [FrameworkElement](#) 或 [FrameworkContentElement](#) 属性的值或 [Setter](#) 值的形式提供）上的属性。

由于要设置的属性必须是 [DependencyProperty](#) 或 [Freezable](#) 属性，大多数属性更改都可以传播到 UI，这是因为属性更改（更改的动态资源值）会经由属性系统确认。大多数控件都包含相应的逻辑；当 [DependencyProperty](#) 有所更改且该属性可能会影响布局时，该逻辑将强制使用控件的其他布局。但是，并不保证所有使用 [DynamicResource](#) 标记扩展作为其值的属性都能在 UI 中提供实时更新。此功能可能仍会因属性、属性所属的类型，甚至应用的逻辑结构而异。

## 样式、DataTemplate 和隐式键

尽管 `ResourceDictionary` 中的所有项都必须具有键，但这并不意味着所有资源都必须具有显式 `x:Key`。多种对象类型在定义为资源时都支持隐式键，其键值会与另一属性的值绑定。这类键被称为隐式键，而 `x:Key` 属性为显式键。任何隐式键都可通过指定显式键来覆盖。

关于资源，一个重要的方案就是用于定义 `Style`。事实上，`Style` 几乎总会作为资源字典中的条目进行定义，因为样式在本质上可供重复使用。有关样式的详细信息，请参阅[样式和模板 \(WPF .NET\)](#)。

控件样式可通过隐式键来创建和引用。用于定义控件默认外观的主题样式依赖于该隐式键。从请求的角度来看，隐式键是控件本身的 `Type`。从定义资源的角度来看，隐式键是样式的 `TargetType`。因此，如果要创建自定义控件的主题或要创建会与现有主题样式交互的样式，则无需为该 `Style` 指定 `x:Key` 指令。另外，如果想要使用主题样式，则根本无需指定任何样式。例如，即使 `Style` 资源似乎没有键，以下样式定义仍起作用：

XAML

```
<Style TargetType="Button">
    <Setter Property="Background" Value="#4E1A3D" />
    <Setter Property="Foreground" Value="White" />
    <Setter Property="BorderThickness" Value="5" />
    <Setter Property="BorderBrush">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0.0" Color="#4E1A3D"/>
                <GradientStop Offset="1.0" Color="Salmon"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

该样式确实具有一个键：隐式键：`System.Windows.Controls.Button` 类型。在标记中，可以直接将 `TargetType` 指定为类型名称（或者，可以选择使用 `{x:Type...}`，以返回 `Type`）。

通过 WPF 使用的默认主题样式机制，即使 `Button` 本身不尝试指定其 `Style` 属性或对样式的特定资源引用，该样式也将作为页面上 `Button` 的运行时样式应用。在页面中定义的样式位于查找序列中的靠前位置（在主题字典样式之前），其所用的键与主题字典样式的键相同。可以在页面上的任意位置指定 `<Button>Hello</Button>`，使用 `Button` 的 `TargetType` 定义的样式将应用于该按钮。如果需要，仍可为此样式显式指定与 `TargetType` 的类型值相同的键，以求在标记中清楚明示，但这是可选的。

如果 `OverridesDefaultStyle` 为 `true`，则样式的隐式键不会应用于控件。（另请注意，`OverridesDefaultStyle` 可能被设置为控件类的本机行为的一部分，而不是在控件的实例上显式设置。）此外，为了支持在派生类方案中使用隐式键，控件必须替代

[DefaultStyleKey](#) (作为 WPF 的一部分提供的所有现有控件都包括此替代)。有关样式、主题和控件设计的详细信息，请参阅[可样式化控件的设计指南](#)。

[DataTemplate](#) 也有一个隐式键。[DataTemplate](#) 的隐式键是 [DataType](#) 属性值。[DataType](#) 也可以作为类型的名称来指定，而不是使用 [{x:Type...}](#) 来显式指定。有关详细信息，请参阅[数据模板化概述](#)。

## 另请参阅

- [代码中的资源](#)
- [合并的资源字典](#)
- [如何定义和引用 WPF 资源](#)
- [如何使用系统资源](#)
- [如何使用应用程序资源](#)
- [x:Type 标记扩展](#)
- [ResourceDictionary](#)
- [应用程序资源](#)
- [定义和引用资源](#)
- [应用程序管理概述](#)
- [StaticResource 标记扩展](#)
- [DynamicResource 标记扩展](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

#### 提出文档问题

#### 提供产品反馈

# 合并资源字典 (WPF .NET)

项目 • 2023/10/13

Windows Presentation Foundation (WPF) 资源支持合并资源字典功能。此功能提供了一种方法，用于在已编译的 XAML 应用程序外部定义 WPF 应用程序的资源部分。然后可以在应用程序之间共享资源，还可更方便地将资源隔离以进行本地化。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 创建合并字典

在标记中，使用以下语法将合并资源字典引入页面：

XAML

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="myresourcedictionary.xaml"/>
      <ResourceDictionary Source="myresourcedictionary2.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Page.Resources>
```

`ResourceDictionary` 元素不具有 `x:Key` 指令，通常该指令对于资源集合中的所有项是必需的。但 `MergedDictionaries` 集合中的另一个 `ResourceDictionary` 引用是一个特殊情况，是为此合并资源字典方案预留的。进一步讲，引入合并资源字典的 `ResourceDictionary` 不能有 `x:Key` 指令。

通常情况下，`MergedDictionaries` 集合中的每个 `ResourceDictionary` 都指定一个 `Source` 属性。`Source` 的值应是解析到待合并资源文件的位置的统一资源标识符 (URI)。该 URI 的目标必须是另一个 XAML 文件，以 `ResourceDictionary` 作为其根元素。

## ① 备注

在指定为合并字典的 `ResourceDictionary` 内定义资源是合法的，可将此作为指定 `Source` 的替代方法，或作为指定源中包括的任何资源的补充。但是，这不是一种常见的方案。合并字典的主要方案是从外部文件位置合并资源。如果想要在标记内为

某一页指定资源，请在主要 `ResourceDictionary` 中（而不是合并字典中）定义这些资源。

## 合并字典行为

合并字典中的资源会占用资源查找范围中的一个位置，此范围恰好在它们合并到的主要资源字典中的范围之后。尽管任何字典中的资源键必须唯一，但一个资源键可在一组合并字典中多次存在。在这种情况下，返回的资源将来自在 `MergedDictionaries` 集合中按顺序找到的最后一个字典。如果在 XAML 中定义了 `MergedDictionaries` 集合，则合并字典在此集合中的顺序即是标记中提供的元素顺序。如果在主要字典和合并的字典中均定义了同一个键，则返回的资源将来自主要字典。这些范围规则同等地适用于静态资源引用和动态资源引用。

## 合并字典和代码

可通过代码将合并字典添加到 `Resources` 字典。对于任何 `Resources` 属性存在的初始为空的默认 `ResourceDictionary` 也有初始为空的默认 `MergedDictionaries` 集合属性。若要通过代码添加合并字典，需要获取对所需主要 `ResourceDictionary` 的引用、获取其 `MergedDictionaries` 属性值，并对 `MergedDictionaries` 中包含的泛型 `Collection` 调用 `Add`。添加的对象必须是新的 `ResourceDictionary`。

在代码中，不要设置 `Source` 属性。相反，必须通过创建或加载来获得 `ResourceDictionary` 对象。加载现有 `ResourceDictionary` 的一种方法是对具有 `ResourceDictionary` 根的现有 XAML 文件流调用 `XamlReader.Load`，然后将返回值强制转换为 `ResourceDictionary`。

## 合并字典 URI

可以通过几种技术来包括合并资源字典，具体视你所使用的统一资源标识符 (URI) 格式而定。宽泛地讲，这些技术可以分为两个类别：作为项目的一部分编译的资源，以及未作为项目的一部分编译的资源。

对于作为项目的一部分编译的资源，可使用引用资源位置的相对路径。相对路径会在编辑期间计算。必须将资源作为“资源”生成操作定义为项目的一部分。如果将资源 `.xaml` 文件作为“资源”包括在项目中，则无需将资源文件复制到输出目录，此资源已包括在已编译的应用程序中。也可以使用“内容”生成操作，但随后必须将文件复制到输出目录，还必须将相同路径关系中的资源文件部署到可执行文件。

① 备注

请勿使用“嵌入资源”生成操作。 WPF 应用程序支持该生成操作本身，但 `Source` 的解析不会纳入 `ResourceManager`，因此不能将单个资源从流中分离出来。只要你还使用了 `ResourceManager` 来访问资源，那么你仍然可以将“嵌入资源”用于其他目的。

一种相关技术是使用指向 XAML 文件的 Pack URI，并将它作为“源”进行引用。 Pack URI 支持对应用程序集的组件和其他技术的引用。有关 Pack URI 的详细信息，请参阅 [WPF 应用程序资源、内容和数据文件](#)。

对于未作为项目的一部分编译的资源，URI 是在运行时计算的。可以使用常见的 URI 传输（例如 `file:` 或 `http:`）来引用资源文件。使用非编译的资源方法的缺点在于 `file:` 访问需要额外的部署步骤，并且 `http:` 访问意味着需要访问 Internet 安全区域。

## 重用合并字典

可以重复使用或在应用程序之间共享合并资源字典，因为可以通过任何有效的统一资源标识符 (URI) 引用要合并的资源字典。执行此操作的确切方式取决于应用程序部署策略以及所遵循的应用程序模型。[前面提及的 Pack URI 策略](#)提供了一种方法，通常用于通过共享程序集引用在开发期间跨多个项目将合并资源作为源使用。在此方案中，资源仍由客户端分发，并且至少有一个应用程序必须部署引用的程序集。还可以通过使用 `http:` 协议的分布式 URI 引用合并资源。

另一种合并字典和应用程序部署的可能方案是将合并字典编写为本地应用程序文件或编写到本地共享存储。

## 本地化

如果需要本地化的资源独立于将合并为主要字典的字典，并且保留为宽松 XAML，则可以单独本地化这些文件。这是本地化附属资源程序集的轻量级替代方法。有关详细信息，请参阅 [WPF 全球化和本地化概述](#)。

## 另请参阅

- [ResourceDictionary](#)
- [XAML 资源概述](#)
- [代码中的资源](#)
- [WPF 应用程序资源、内容和数据文件](#)
- [WPF 全球化和本地化概述](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 代码中的资源 (WPF .NET)

项目 · 2023/10/13

本概述主要介绍如何使用代码（而非 XAML 语法）来访问或创建 Windows Presentation Foundation (WPF) 资源。若要从 XAML 语法角度来详细了解常规资源用法以及资源，请参阅 [XAML 资源概述](#)。

## 从代码访问资源

如果在代码中请求资源，标识 XAML 定义的资源的键也用于检索特定的资源。从代码检索资源最简单的方法是从应用程序中的框架级对象调用 [FindResource](#) 或 [TryFindResource](#) 方法。这两个方法之间的行为差异在于未找到所请求的键时所发生的情况。

[FindResource](#) 会引发异常。[TryFindResource](#) 不会引发异常，但会返回 `null`。每个方法都将资源键作为一个输入参数，并返回一个松散类型化对象。

通常，资源键是一个字符串，但也有[偶尔的非字符串用法](#)。代码资源解析的查找逻辑与动态资源引用 XAML 的情况相同。对资源的搜索从调用元素开始，然后继续通过逻辑树中的父元素进行搜索。如果必要，将继续查找应用程序资源、主题以及系统资源。针对资源的代码请求将正确考虑这些资源在运行时期间发生的更改。

下面的代码示例展示了一个 [Click](#) 事件处理程序，它按键来查找资源并使用返回的值来设置属性。

C#

```
private void myButton_Click(object sender, RoutedEventArgs e)
{
    Button button = (Button)sender;
    button.Background = (Brush)this.FindResource("RainbowBrush");
}
```

分配资源引用的另一种方法是 [SetResourceReference](#)。该方法采用两个参数：资源的键，以及特定依赖属性（应向其分配资源值的元素实例上存在的依赖属性）的标识符。就功能而言，此方法是相同的，且具有无需强制转换任何返回值的优点。

还有另一种以编程方式访问资源的方法，即：将 [Resources](#) 属性的内容作为字典来访问。资源字典用于将新资源添加到现有集合，检查给定的键名称是否已由集合使用以及其他操作。如果完全通过代码编写 WPF 应用程序，还可以通过代码创建整个集合，为它分配资源。随后，可以将此集合分配给元素的 [Resources](#) 属性。下一部分对此进行介绍。

通过将特定键用作索引，可以在任何给定的 [Resources](#) 集合内编制索引。以这种方式访问的资源不遵循资源解析的普通运行时规则。你访问的仅是该特定集合。如果在所请求

的键处未找到有效的对象，资源查找则不会将资源范围遍历到根或应用程序。但是，在某些情况下，正因为对键的搜索范围进行了更多的约束，才使得此方法在性能上具有优势。有关如何直接使用资源字典的详细信息，请参阅 [ResourceDictionary](#) 类。

## 使用代码创建资源

如果要通过代码方式创建整个 WPF 应用程序，可能也需要通过代码方式创建该应用程序中的任何资源。为此，应先新建一个 [ResourceDictionary](#) 实例，接着使用对 [ResourceDictionary.Add](#) 的连续调用将所有资源添加到字典中。然后，分配创建的 [ResourceDictionary](#) 来设置位于页面范围内的元素上的 [Resources](#) 属性，或设置 [Application.Resources](#) 属性。也可以将 [ResourceDictionary](#) 作为一个单独的对象来维护（而不将它添加到元素中）。但是，如果这样做，必须通过项键来访问其中的资源，就好像它是泛型字典一样。未附加到元素 [Resources](#) 属性的 [ResourceDictionary](#) 将不作为元素树的一部分存在，在查找序列中也不具有可供 [FindResource](#) 及相关方法使用的范围。

## 将对象用作键

大多数资源用法都会将资源的键设置为字符串。但是，各种 WPF 功能会有意地将对象类型用作键，而不是字符串。WPF 样式和主题支持使用按对象类型对资源进行键控的功能。成为非常样式控件的默认设置的样式和主题是按它们应该应用到的控件的 [Type](#) 来进行键控的。

按类型进行键控提供了一种可靠的查找机制，该机制作用于每个控件类型的默认实例。可以通过反射检测到类型，并且类型可用于设置派生类的样式，即使派生类型没有默认样式。可以使用 [x:Type 标记扩展](#) 为 XAML 中定义的资源指定 [Type](#) 键。对于支持 WPF 功能的其他非字符串键用法，也存在类似扩展，例如 [ComponentResourceKey 标记扩展](#)。

有关详细信息，请参阅[样式](#)、[DataTemplate](#) 和[隐式键](#)。

## 另请参阅

- [XAML 资源概述](#)
- [如何定义和引用 WPF 资源](#)
- [如何使用系统资源](#)
- [如何使用应用程序资源](#)

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何定义和引用 WPF 资源 (WPF .NET)

项目 • 2023/10/13

此示例演示如何定义资源并引用它。可以通过 XAML 或代码引用资源。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## XAML 示例

以下示例定义两种类型的资源：一种 `SolidColorBrush` 资源和多个 `Style` 资源。

XAML

```
<Window.Resources>
    <SolidColorBrush x:Key="MyBrush" Color="#05E0E9"/>
    <Style TargetType="Border">
        <Setter Property="Background" Value="#4E1A3D" />
        <Setter Property="BorderThickness" Value="5" />
        <Setter Property="BorderBrush">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0.0" Color="#4E1A3D"/>
                    <GradientStop Offset="1.0" Color="Salmon"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
    <Style TargetType="TextBlock" x:Key="TitleText">
        <Setter Property="FontSize" Value="18"/>
        <Setter Property="Foreground" Value="#4E87D4"/>
        <Setter Property="FontFamily" Value="Trebuchet MS"/>
        <Setter Property="Margin" Value="0,10,10,10"/>
    </Style>
    <Style TargetType="TextBlock" x:Key="Label">
        <Setter Property="HorizontalAlignment" Value="Right"/>
        <Setter Property="FontSize" Value="13"/>
        <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
        <Setter Property="FontFamily" Value="Arial"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Margin" Value="0,3,10,0"/>
    </Style>
</Window.Resources>
```

# 资源

[SolidColorBrush 资源](#) `MyBrush` 用于提供多个属性的值，每个属性都采用 [Brush](#) 类型的值。此资源通过 `x:Key` 值引用。

XAML

```
<Border>
    <StackPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="
{StaticResource MyBrush}" Text="Text" Margin="20" />
        <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
        <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
    </StackPanel>
</Border>
```

在前面的示例中，通过 [StaticResource 标记扩展](#) 访问 `MyBrush` 资源。资源分配给可接受所定义资源的类型的属性。在此示例中为 `Background`、`Foreground` 和 `Fill` 属性。

资源字典中的所有资源都必须提供键。但是，在定义样式时，它们可以省略键，如下一部分中所述。

如果使用 [StaticResource 标记扩展](#) 从其他资源中引用资源，则也会按照在字典中找到的顺序请求资源。请确保在集合中定义的任何资源都早于请求该资源的位置。有关详细信息，请参阅[静态资源](#)。

如有必要，可通过使用 [DynamicResource 标记扩展](#) 在运行时引用资源来解决资源引用的严格创建顺序，但你需要知道，这种 [DynamicResource](#) 技术会影响性能。有关详细信息，请参阅[动态资源](#)。

# 样式资源

以下示例隐式和显式引用样式：

XAML

```
<Border>
    <StackPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock HorizontalAlignment="Right" FontSize="36" Foreground="
{StaticResource MyBrush}" Text="Text" Margin="20" />
        <Button HorizontalAlignment="Left" Height="30" Background="
{StaticResource MyBrush}" Margin="40">Button</Button>
    </StackPanel>
</Border>
```

```
{StaticResource MyBrush}" Margin="40">Button</Button>
    <Ellipse HorizontalAlignment="Center" Width="100" Height="100"
Fill="{StaticResource MyBrush}" Margin="10" />
</StackPanel>
</Border>
```

在前面的代码示例中，`Style` 资源 `TitleText` 和 `Label` 都以特定控件类型为目标。在本例中，它们都以 `TextBlock` 为目标。当该样式资源由其 `Style` 属性的资源键引用时，这些样式将在目标控件上设置各种不同的属性。

样式虽然以 `Border` 控件为目标，但并不定义键。省略键时，`TargetType` 属性所针对的对象类型将隐式用作样式的键。当样式键控为某个类型时，只要这些控件在该样式的范围内，它就将成为该类型的所有控件的默认样式。有关详细信息，请参阅[样式、DataTemplate 和隐式键](#)。

## 代码示例

以下代码片段演示如何通过代码创建和设置资源

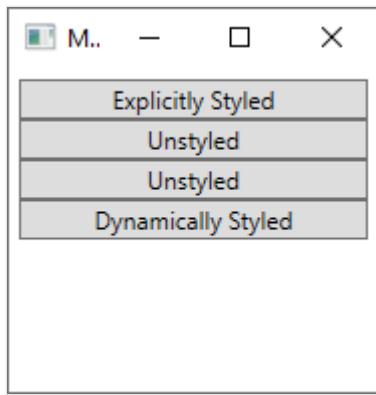
## 创建样式资源

创建资源并将其分配给资源字典可以随时发生。但是，只有使用 `DynamicResource` 语法的 XAML 元素才会在创建后使用资源自动更新。

例如，请看下面的窗口。它有四个按钮。第四个按钮使用 `DynamicResource` 来设置自己的样式。但是，此资源尚不存在，因此它看起来就像一个普通的按钮：

XAML

```
<StackPanel Margin="5">
    <Button Click="Button_Click">Explicitly Styled</Button>
    <Button>Unstyled</Button>
    <Button>Unstyled</Button>
    <Button Style="{DynamicResource ResourceKey=buttonStyle1}">Dynamically
Styled</Button>
</StackPanel>
```



单击第一个按钮并执行以下任务时，将调用以下代码：

- 创建一些颜色以便于参考。
- 创建新样式。
- 为样式分配资源库。
- 将样式作为名为 `buttonStyle1` 的资源添加到窗口的资源字典中。
- 将样式直接分配给引发 `Click` 事件的按钮。

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    // Create colors
    Color purple = (Color)ColorConverter.ConvertFromString("#4E1A3D");
    Color white = Colors.White;
    Color salmon = Colors.Salmon;

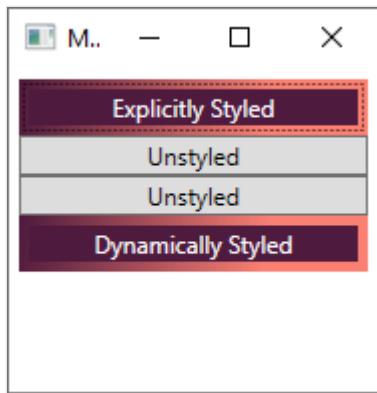
    // Create a new style for a button
    var buttonStyle = new Style(typeof(Button));

    // Set the properties of the style
    buttonStyle.Setters.Add(new Setter(Control.BackgroundProperty, new
SolidColorBrush(purple)));
    buttonStyle.Setters.Add(new Setter(Control.ForegroundProperty, new
SolidColorBrush(white)));
    buttonStyle.Setters.Add(new Setter(Control.BorderBrushProperty, new
LinearGradientBrush(purple, salmon, 45d)));
    buttonStyle.Setters.Add(new Setter(Control.BorderThicknessProperty, new
Thickness(5)));

    // Set this style as a resource. Any DynamicResource tied to this key
    // will be updated.
    this.Resources["buttonStyle1"] = buttonStyle;

    // Set this style directly to a button
    ((Button)sender).Style = buttonStyle;
}
```

运行代码后，窗口将更新：



请注意，第四个按钮的样式已更新。该样式已自动应用，因为按钮使用 [DynamicResource 标记扩展](#) 来引用尚不存在的样式。创建样式并将其添加到窗口的资源后，该样式将应用于按钮。有关详细信息，请参阅[动态资源](#)。

## 查找资源

下面的代码遍历运行其中的 XAML 对象的逻辑树，以查找指定的资源。资源可以在对象本身（资源的父级）上定义，直到到达根（即应用程序本身）。下面的代码从按钮本身开始搜索资源：

C#

```
myButton.Style = myButton.TryFindResource("buttonStyle1") as Style;
```

## 显式引用资源

引用资源时，通过搜索或创建资源，可以将其直接分配给属性：

C#

```
// Set this style as a resource. Any DynamicResource tied to this key will  
// be updated.  
this.Resources["buttonStyle1"] = buttonStyle;
```

## 另请参阅

- [XAML 资源概述](#)
- [样式和模板](#)
- [如何使用系统资源](#)
- [如何使用应用程序资源](#)

## 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何使用应用程序资源 (WPF .NET)

项目 • 2023/10/13

此示例演示如何使用应用程序定义的资源。可以在应用程序级别定义资源，通常通过 App.xaml 或 Application.xaml 文件进行，具体取决于你的项目使用哪个文件。应用程序定义的资源是全局范围的，可由应用程序的所有部分访问。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 示例

以下示例演示应用程序定义文件。应用程序定义文件定义资源部分（Resources 属性的值）。构成应用程序的所有其他页均可访问在应用程序级别定义的资源。这种情况下，资源是声明样式。由于包含控件模板的完整样式可能很长，因此此示例省略了在样式的 ContentTemplate 属性设置器中定义的控件模板。

XAML

```
<Application.Resources>
    <Style TargetType="Border" x:Key="FancyBorder">
        <Setter Property="Background" Value="#4E1A3D" />
        <Setter Property="BorderThickness" Value="5" />
        <Setter Property="BorderBrush">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0.0" Color="#4E1A3D"/>
                    <GradientStop Offset="1.0" Color="Salmon"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Application.Resources>
```

下面的示例显示了引用上一个示例中的应用程序级资源的 XAML 页面。资源使用 [StaticResource 标记扩展](#) 引用，该扩展指定资源的唯一资源键。在当前对象和窗口的范围内找不到资源“FancyBorder”，因此超出当前页面范围进入应用程序级资源范围进行资源查找。

XAML

```
<Border Style="{StaticResource FancyBorder}">
  <StackPanel Margin="5">
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
    <Button>Button 4</Button>
  </StackPanel>
</Border>
```

## 另请参阅

- [XAML 资源概述](#)
- [代码中的资源](#)
- [如何定义和引用 WPF 资源](#)
- [如何使用系统资源](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# 如何使用系统资源 (WPF .NET)

项目 • 2023/10/20

此示例演示如何使用系统定义的资源。 系统资源由 WPF 提供，并允许访问操作系统资源，例如字体、颜色和图标。 系统资源将系统确定的许多值以资源和属性的形式公开，以帮助创建与 Windows 一致的视觉效果。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 字体

使用 [SystemFonts](#) 类引用操作系统使用的字体。 此类既包含作为静态属性的系统字体值，又包含引用可用于在运行时动态访问这些值的资源键的属性。 例如，[CaptionFontFamily](#) 是一个 [SystemFonts](#) 值，而 [CaptionFontFamilyKey](#) 是相应的资源键。

以下示例演示如何访问并使用 [SystemFonts](#) 的属性作为静态值来设置按钮样式或自定义文本块：

XAML

```
<TextBlock FontSize="{x:Static SystemFonts.SmallCaptionFontSize}"  
          FontWeight="{x:Static SystemFonts.SmallCaptionFontWeight}"  
          FontFamily="{x:Static SystemFonts.SmallCaptionFontFamily}"  
          Text="Small Caption Font">  
</TextBlock>
```

若要在代码中使用 [SystemFonts](#) 的值，不必使用静态值或动态资源引用。 而是可以使用 [SystemFonts](#) 类的非键属性。 尽管非键属性已明确定义为静态属性，但是系统托管的 WPF 的运行时行为将会实时重新评估这些属性，并且会适当考虑对系统值进行用户驱动的更改。 以下示例演示如何指定按钮的字体设置：

C#

```
var myButton = new Button()  
{  
    Content = "SystemFonts",  
    Background = SystemColors.ControlDarkDarkBrush,  
    FontSize = SystemFonts.IconFontSize,  
    FontWeight = SystemFonts.MessageFontWeight,  
    FontFamily = SystemFonts.CaptionFontFamily  
};
```

```
mainStackPanel.Children.Add(myButton);
```

## XAML 中的动态字体

系统字体规格可用作静态或动态资源。如果希望字体规格在应用程序运行时自动更新，请使用动态资源；否则，请使用静态资源。

### ① 备注

动态资源的属性名称后面附有关键字 `Key`。

以下示例演示如何访问和使用系统字体动态资源来设计按钮样式或自定义文本块：

#### XAML

```
<TextBlock FontSize="{DynamicResource {x:Static  
SystemFonts.SmallCaptionFontSize}}"  
          FontWeight="{DynamicResource {x:Static  
SystemFonts.SmallCaptionFontWeight}}"  
          FontFamily="{DynamicResource {x:Static  
SystemFonts.SmallCaptionFontFamily}}"  
          Text="Small Caption Font">  
</TextBlock>
```

## 参数

使用 [SystemParameters](#) 类引用系统级属性，例如主显示器的大小。此类包含系统参数值属性和绑定到值的资源键。例如，[FullPrimaryScreenHeight](#) 是一个 [SystemParameters](#) 属性值，而 [FullPrimaryScreenHeightKey](#) 是相应的资源键。

以下示例演示如何访问并使用 [SystemParameters](#) 的静态值来设置按钮样式或自定义按钮。此标记示例通过将 [SystemParameters](#) 值应用于按钮来调整按钮大小：

#### XAML

```
<Button FontSize="8"  
        Height="{x:Static SystemParameters.CaptionHeight}"  
        Width="{x:Static SystemParameters.IconGridWidth}"  
        Content="System Parameters">  
</Button>
```

要在代码中使用 [SystemParameters](#) 的值，不必使用静态引用或动态资源引用。可改为使用 [SystemParameters](#) 类的值。尽管非键属性已明确定义为静态属性，但是系统托管的 WPF 的运行时行为将会实时重新评估这些属性，并且会适当考虑对系统值进行用户驱动的更改。以下示例演示如何使用 [SystemParameters](#) 值设置按钮的宽度和高度：

C#

```
var myButton = new Button()
{
    Content = "SystemParameters",
    FontSize = 8,
    Background = SystemColors.ControlDarkDarkBrush,
    Height = SystemParameters.CaptionHeight,
    Width = SystemParameters.CaptionWidth,
};

mainStackPanel.Children.Add(myButton);
```

## XAML 中的动态参数

系统参数规格可用作静态或动态资源。如果希望参数规格在应用程序运行时自动更新，请使用动态资源；否则，请使用静态资源。

### ① 备注

动态资源的属性名称后面附有关键字 `Key`。

以下示例演示如何访问和使用系统参数动态资源来设计按钮样式或自定义按钮。此 XAML 示例通过将 [SystemParameters](#) 值分配给按钮的宽度和高度来调整按钮大小。

XAML

```
<Button FontSize="8"
        Height="{DynamicResource {x:Static
SystemParameters.CaptionHeightKey}}"
        Width="{DynamicResource {x:Static
SystemParameters.IconGridWidthKey}}"
        Content="System Parameters">
</Button>
```

## 另请参阅

- [XAML 资源概述](#)
- [代码中的资源](#)

- 如何定义和引用 WPF 资源
- 如何使用应用程序资源

## ⌚ 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

## .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

⌚ 提出文档问题

↗ 提供产品反馈

# XAML 概述 (WPF .NET)

项目 • 2023/10/13

本本介绍 XAML 语言的功能，并演示如何使用 XAML 编写 Windows Presentation Foundation (WPF) 应用。本主题专门介绍 WPF 实现的 XAML。XAML 本身是一个比 WPF 更大的语言概念。

## ① 重要

面向 .NET 7 和 .NET 6 的桌面指南文档正在撰写中。

## 什么是 XAML

XAML 是一种声明性标记语言。应用于 .NET 编程模型时，XAML 可简化为 .NET 应用创建 UI 的过程。你可以在声明性 XAML 标记中创建可见的 UI 元素，然后使用代码隐藏文件（这些文件通过分部类定义与标记相联接）将 UI 定义与运行时逻辑相分离。XAML 直接以程序集中定义的一组特定后备类型表示对象的实例化。这与大多数其他标记语言不同，后者通常是与后备类型系统没有此类直接关系的解释语言。XAML 实现了一个工作流，通过此工作流，各方可以采用不同的工具来处理 UI 和应用的逻辑。

以文本表示时，XAML 文件是通常具有 `.xaml` 扩展名的 XML 文件。可通过任何 XML 编码对文件进行编码，但通常以 UTF-8 编码。

以下示例演示如何创建 UI 中的按钮。此示例用于初步了解 XAML 如何表示常用 UI 编程形式（它不是一个完整的示例）。

XAML

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

## XAML 语法概述

下面章节介绍 XAML 语法的基本形式，并提供一个简短的标记示例。这些章节并不提供每个语法形式的完整信息，例如这些语法形式如何在后备类型系统中表示。有关 XAML 语法细节的详细信息，请参阅 [XAML 语法详述](#)。

如果已熟悉 XML 语言，则下面几节中的很多材料对你而言都是基础知识。这得益于 XAML 的一个基本设计原则。XAML 语言定义它自己的概念，但这些概念也适用于 XML

语言和标记形式。

## XAML 对象元素

对象元素通常声明类型的实例。该类型在将 XAML 用作语言的技术所引用的程序集中定义。

对象元素语法始终以左尖括号 (<) 开头。后跟要创建实例的类型的名称。（该名称可能包含前缀，下文将解释前缀的概念。）此后可以选择声明该对象元素的特性。要完成对象元素标记，请以右尖括号 (>) 结尾。也可以使用不含任何内容的自结束形式，方法是用一个正斜杠后接一个右尖括号 (/>) 来完成标记。例如，请再次查看之前演示的标记片段。

XAML

```
<StackPanel>
    <Button Content="Click Me"/>
</StackPanel>
```

这指定了两个对象元素：`<StackPanel>`（含有内容，后面有一个结束标记）和 `<Button .../>`（自结束形式，包含几个特性）。对象元素 `StackPanel` 和 `Button` 各映射到一个类名，该类由 WPF 定义并且属于 WPF 程序集。指定对象元素标记时，会创建一条指令，指示 XAML 处理创建基础类型的新实例。每个实例都是在分析和加载 XAML 时通过调用基础类型的无参数构造函数来创建。

## 特性语法（属性）

对象的属性通常可表示为对象元素的特性。特性语法对设置的对象属性命名，后跟赋值运算符 (=)。特性的值始终指定为包含在引号中的字符串。

特性语法是最简化的属性设置语法，并且对曾使用过标记语言的开发人员而言是最直观的语法。例如，以下标记将创建一个具有红色文本和蓝色背景的按钮，以及 `Content` 显示文本。

XAML

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

## 属性元素语法

对于对象元素的某些属性，无法使用特性语法，因为无法在特性语法的引号和字符串限制内充分地表达提供属性值所必需的对象或信息。对于这些情况，可以使用另一个语法，即属性元素语法。

属性元素开始标记的语法为 `<TypeName.PropertyName>`。通常，该标记的内容是类型的对象元素，属性会将该元素作为其值。指定内容之后，必须用结束标记结束属性元素。结束标记的语法为 `</TypeName.PropertyName>`。

如果可以使用特性语法，那么使用特性语法通常更为方便，且能够实现更为精简的标记，但这通常只是样式问题，而不是技术限制。以下示例演示在前面的特性语法示例中设置的相同属性，但这次对 `Button` 的所有属性使用属性元素语法。

#### XAML

```
<Button>
    <Button.Background>
        <SolidColorBrush Color="Blue"/>
    </Button.Background>
    <Button.Foreground>
        <SolidColorBrush Color="Red"/>
    </Button.Foreground>
    <Button.Content>
        This is a button
    </Button.Content>
</Button>
```

## 集合语法

XAML 语言包含一些优化，可以生成更易于阅读的标记。其中一项优化是：如果某个特定属性采用集合类型，则在标记中声明为该属性的值内的子元素的项将成为集合的一部分。在这种情况下，子对象元素的集合是设置为集合属性的值。

下面的示例演示用于设置 `GradientStops` 属性的值的集合语法。

#### XAML

```
<LinearGradientBrush>
    <LinearGradientBrush.GradientStops>
        <!-- no explicit new GradientStopCollection, parser knows how to
find or create -->
        <GradientStop Offset="0.0" Color="Red" />
        <GradientStop Offset="1.0" Color="Blue" />
    </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

## XAML 内容属性

XAML 指定了一个语言功能，通过该功能，类可以指定它的一个且仅一个属性为 XAML 内容属性。该对象元素的子元素用于设置该内容属性的值。换言之，仅对内容属性而言，可以在 XAML 标记中设置该属性时省略属性元素，并在标记中生成更直观的父级/子级形式。

例如，`Border` 指定 `Child` 的内容属性。以下两个 `Border` 元素的处理方式相同。第一个元素利用内容属性语法并省略 `Border.Child` 属性元素。第二个元素显式显示 `Border.Child`。

```
XAML

<Border>
    <TextBox Width="300"/>
</Border>
<!--explicit equivalent-->
<Border>
    <Border.Child>
        <TextBox Width="300"/>
    </Border.Child>
</Border>
```

作为 XAML 语言的规则，XAML 内容属性的值必须完全在该对象元素的其他任何属性元素之前或之后指定。例如，以下标记不会进行编译。

```
XAML

<Button>I am a
<Button.Background>Blue</Button.Background>
blue button</Button>
```

有关 XAML 语法细节的详细信息，请参阅 [XAML 语法详述](#)。

## 文本内容

有少量 XAML 元素可直接将文本作为其内容来处理。若要实现此功能，必须满足以下条件之一：

- 类必须声明一个内容属性，并且该内容属性必须是可赋值给字符串的类型（该类型可以是 `Object`）。例如，任何 `ContentControl` 都使用 `Content` 作为其内容属性，并且它属于类型 `Object`，这对于 `ContentControl`（如 `Button`）支持以下用法：  
`<Button>Hello</Button>`。

- 类型必须声明一个类型转换器，该类型转换器将文本内容用作初始化文本。例如，`<Brush>Blue</Brush>` 将 `Blue` 的内容值转换为画笔。这种情况实际上并不常见。
- 类型必须为已知的 XAML 语言基元。

## 内容属性和集合语法组合

考虑以下示例。

XAML

```
<StackPanel>
    <Button>First Button</Button>
    <Button>Second Button</Button>
</StackPanel>
```

此处，每个 `Button` 都是 `StackPanel` 的子元素。这是一个简单直观的标记，此标记由于两个不同的原因省略了两个标记。

- 省略了 `StackPanel.Children` 属性元素：`StackPanel` 派生自 `Panel`。`Panel` 将 `Panel.Children` 定义为其 XAML 内容属性。
- 省略了 `UIElementCollection` 对象元素：`Panel.Children` 属性使用类型 `UIElementCollection`，该类型实现 `IList`。根据处理 `IList` 等集合的 XAML 规则，集合的元素标记可以省略。（在本例中，`UIElementCollection` 实际上无法实例化，因为它不公开无参数构造函数，这便是 `UIElementCollection` 对象元素显示为注释掉的原因）。

XAML

```
<StackPanel>
    <StackPanel.Children>
        <!--<UIElementCollection>-->
        <Button>First Button</Button>
        <Button>Second Button</Button>
        <!--</UIElementCollection>-->
    </StackPanel.Children>
</StackPanel>
```

## 特性语法（事件）

特性语法还可用于事件成员，而非属性成员。在这种情况下，特性的名称为事件的名称。在 XAML 事件的 WPF 实现中，特性的值是实现该事件的委托的处理程序的名称。例如，以下标记将 `Click` 事件的处理程序分配给在标记中创建的 `Button`：

```
<Button Click="Button_Click" >Click Me!</Button>
```

除此特性语法示例外，还有更多关于 WPF 中的事件和 XAML 的内容。例如，可了解此处引用的 `ClickHandler` 表示什么，以及它是如何定义的。这将在本文后面的[事件和 XAML 代码隐藏](#)一节中介绍。

## XAML 中的大小写和空白

一般而言，XAML 区分大小写。出于解析后备类型的目的，WPF XAML 按照 CLR 区分大小写的相同规则区分大小写。以下情况下，对象元素、属性元素和特性名称均必须使用区分大小写的形式指定：按名称与程序集中的基础类型进行比较或者与类型的成员进行比较。XAML 语言关键字和基元也区分大小写。值并不总是区分大小写。值是否区分大小写将取决于与采用该值的属性关联的类型转换器行为，或取决于属性值类型。例如，采用 `Boolean` 类型的属性可以采用 `true` 或 `True` 作为等效值，但只是因为将字符串转换为 `Boolean` 的本机 WPF XAML 分析程序类型转换已经允许将这些值作为等效值。

WPF XAML 处理器和序列化程序将忽略或删除所有无意义的空白，并标准化任何有意义的空白。这与 XAML 规范的默认空白行为建议一致。只有在 XAML 内容属性中指定字符串时，此行为的重要性才会体现出来。简言之，XAML 将空格、换行符和制表符转化为空格，如果它们出现在一个连续字符串的任一端，则保留一个空格。本文不包含有关 XAML 空白处理的完整说明。有关详细信息，请参阅 [XAML 中的空白处理](#)。

## 标记扩展

标记扩展是一个 XAML 语言概念。用于提供特性语法的值时，大括号 (`{` 和 `}`) 表示标记扩展用法。此用法指示 XAML 处理不要像通常那样将特性值视为文本字符串或者可转换为字符串的值。

WPF 应用编程中最常用的标记扩展是 `Binding` (用于数据绑定表达式) 以及资源引用 `StaticResource` 和 `DynamicResource`。通过使用标记扩展，即使属性通常不支持特性语法，也可以使用特性语法为属性提供值。标记扩展经常使用中间表达式类型实现一些功能，例如，推迟值或引用仅在运行时才存在的其他对象。

例如，以下标记使用特性语法设置 `Style` 属性的值。`Style` 属性采用 `Style` 类的实例，该类在默认情况下无法由特性语法字符串进行实例化。但在本例中，特性引用了特定的标记扩展 `StaticResource`。处理该标记扩展时，将返回对以前在资源字典中作为键控资源进行实例化的某个样式的引用。

```
<Window x:Class="index.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="100" Width="300">
    <Window.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="BorderBrush" Value="Blue"/>
            <Setter Property="BorderThickness" Value="5" />
        </Style>
    </Window.Resources>
    <Border Style="{StaticResource PageBackground}">
        <StackPanel>
            <TextBlock Text="Hello" />
        </StackPanel>
    </Border>
</Window>
```

有关特定在 WPF 中实现的所有 XAML 标记扩展的参考列表，请参阅 [WPF XAML 扩展](#)。有关由 System.Xaml 定义并且可更广泛用于 .NET XAML 实现的标记扩展的参考列表，请参阅 [XAML 命名空间 \(x:\) 语言功能](#)。有关标记扩展概念的详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 类型转换器

在 [XAML 语法概述](#)一节中，曾提到特性值必须能够通过字符串进行设置。对字符串如何转换为其他对象类型或基元值的基本本机处理取决于 [String](#) 类型本身，以及对某些类型（如 [DateTime](#) 或 [Uri](#)）的本机处理。但是很多 WPF 类型或这些类型的成员扩展了基本字符串特性处理行为，因此可以将更复杂的对象类型的实例指定为字符串和特性。

[Thickness](#) 结构是一个类型示例，该类型拥有可使用 XAML 的类型转换。[Thickness](#) 指示嵌套矩形中的度量，可用作属性（如 [Margin](#)）的值。通过对 [Thickness](#) 放置类型转换器，所有使用 [Thickness](#) 的属性都可以更容易地在 XAML 中指定，因为它们可指定为特性。以下示例使用类型转换和特性语法来为 [Margin](#) 提供值：

### XAML

```
<Button Margin="10,20,10,30" Content="Click me"/>
```

以上特性语法示例与下面更为详细的语法示例等效，但在下面的示例中，[Margin](#) 改为通过包含 [Thickness](#) 对象元素的属性元素语法进行设置。而且将 [Thickness](#) 的四个关键属性设置为新实例的特性：

### XAML

```
<Button Content="Click me">
    <Button.Margin>
        <Thickness Left="10" Top="20" Right="10" Bottom="30"/>
    </Button.Margin>
</Button>
```

### ① 备注

对于少数对象，类型转换是在不涉及子类的情况下将属性设置为此类型的唯一公开方式，因为类型自身没有无参数构造函数。示例为 `Cursor`。

有关类型转换的详细信息，请参阅 [TypeConverters](#) 和 [XAML](#)。

## 根元素和命名空间

一个 XAML 文件只能有一个根元素，以便同时作为格式正确的 XML 文件和有效的 XAML 文件。对于典型 WPF 方案，可使用在 WPF 应用模型中具有突出意义的根元素（例如，页面的 `Window` 或 `Page`、外部字典的 `ResourceDictionary` 或应用定义的 `Application`）。以下示例演示 WPF 页的典型 XAML 文件的根元素，此根元素为 `Page`。

XAML

```
<Page x:Class="index.Page1"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Page1">

</Page>
```

根元素还包含特性 `xmlns` 和 `xmlns:x`。这些特性向 XAML 处理器指示哪些 XAML 命名空间包含标记将其作为元素引用的后备类型的类型定义。`xmlns` 特性明确指示默认的 XAML 命名空间。在默认的 XAML 命名空间中，可以不使用前缀指定标记中的对象元素。对于大多数 WPF 应用程序方案以及 SDK 的 WPF 部分中给出的几乎所有示例，默认的 XAML 命名空间均映射到 WPF 命名空间

<http://schemas.microsoft.com/winfx/2006/xaml/presentation>。`xmlns:x` 特性指示另一个 XAML 命名空间，该命名空间映射 XAML 语言命名空间  
<http://schemas.microsoft.com/winfx/2006/xaml>。

使用 `xmlns` 定义用法范围和名称范围映射的做法符合 XML 1.0 规范。XAML 名称范围与 XML 名称范围的不同仅在于：XAML 名称范围还包含有关进行类型解析和分析 XAML 时名称范围的元素如何受类型支持的信息。

只有在每个 XAML 文件的根元素上，`xmlns` 特性才是绝对必需的。`xmlns` 定义将适用于根元素的所有子代元素（此行为也符合 `xmlns` 的 XML 1.0 规范）。同时允许根以下的其他元素上具有 `xmlns` 特性，这些特性将适用于定义元素的任何子代元素。但是，频繁定义或重新定义 XAML 命名空间会导致难以阅读 XAML 标记样式。

其 XAML 处理器的 WPF 实现包括可识别 WPF 核心程序集的基础结构。已知 WPF 核心程序集包含支持指向默认 XAML 命名空间的 WPF 映射的类型。这是通过项目生成文件中的配置以及 WPF 生成和项目系统实现的。因此，要引用来自 WPF 程序集的 XAML 元素，只需将默认 XAML 命名空间声明为默认 `xmlns`。

## x: 前缀

在之前的根元素示例中，前缀 `x:` 用于映射 XAML 命名空间

<http://schemas.microsoft.com/winfx/2006/xaml>，该命名空间是支持 XAML 语言构造的专用 XAML 命名空间。在这整个 SDK 的项目模板、示例以及文档中，此 `x:` 前缀用于映射该 XAML 命名空间。XAML 语言的 XAML 命名空间包含多个将在 XAML 中频繁使用的编程构造。下面列出了最常用的 `x:` 前缀编程构造：

- `x:Key`: 为 `ResourceDictionary` (或其他框架中的类似字典概念) 中的每个资源设置唯一的键。在典型的 WPF 应用标记中的所有 `x:` 用法中，`x:Key` 可能占到 90%。
- `x:Class`: 向为 XAML 页提供代码隐藏的类指定 CLR 命名空间和类名。必须具有这样一个类才能支持每个 WPF 编程模型的代码隐藏，因此即使没有资源，也几乎总是能看到映射的 `x:`。
- `x:Name`: 处理对象元素后，为运行时代码中存在的实例指定运行时对象名称。通常，经常为 `x:Name` 使用 WPF 定义的等效属性。此类属性特定映射到 CLR 后备属性，因此更便于进行应用编程，在应用编程中，经常使用运行时代码从初始化的 XAML 中查找命名元素。最常见的此类属性是 `FrameworkElement.Name`。在特定类型中不支持等效的 WPF 框架级 `Name` 属性时，仍然可以使用 `x:Name`。某些动画方案中会发生这种情况。
- `x:Static`: 启用一个返回静态值的引用，该静态值不是与 XAML 兼容的属性。
- `x>Type`: 根据类型名称构造 `Type` 引用。用于指定采用 `Type` (例如 `Style.TargetType`) 的特性，但属性经常具有本机的字符串到 `Type` 的转换功能，因此使用 `x>Type` 标记扩展用法是可选的。

`x:` 前缀/XAML 命名空间中还有其他一些不太常见的编程构造。有关详细信息，请参阅 [XAML 命名空间 \(x:\) 语言功能](#)。

# 自定义前缀和自定义类型

对于自身的自定义程序集或 PresentationCore、PresentationFramework 和 WindowsBase 的 WPF 核心以外的程序集，可以将该程序集指定为自定义 `xmlns` 映射的一部分。只要该类型能够正确地实现以支持正在尝试的 XAML 用法，就可以在 XAML 中引用该程序集中的类型。

下面是一个说明自定义前缀如何在 XAML 标记中工作的基本示例。前缀 `custom` 在根元素标记中定义，并映射为打包在应用中并随应用一起提供的特定程序集。此程序集包含 `NumericUpDown` 类型，实现该类型的目的的是在支持常规 XAML 用法之外，还可以使用允许在 WPF XAML 内容模型的此特定点执行插入的类继承。通过使用该前缀，此 `NumericUpDown` 控件的一个实例声明为对象元素，以便 XAML 分析程序可找到包含该类型的 XAML 命名空间，从而找到包含该类型定义的后备程序集的位置。

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-
    namespace:NumericUpDownCustomControl;assembly=CustomLibrary"
    >
    <StackPanel Name="LayoutRoot">
        <custom:NumericUpDown Name="numericCtrl1" Width="100" Height="60"/>
    ...
</StackPanel>
</Page>
```

有关 XAML 中自定义类型的详细信息，请参阅 [XAML 及 WPF 的自定义类](#)。

若要深入了解 XML 命名空间与程序集中代码命名空间之间的关系，请参阅 [WPF XAML 的 XAML 命名空间和命名空间映射](#)。

## 事件和 XAML 代码隐藏

大多数 WPF 应用既包括 XAML 标记，也包括代码隐藏。在一个项目中，XAML 编写为 `.xaml` 文件，而 CLR 语言（如 Microsoft Visual Basic 或 C#）用于编写代码隐藏文件。

在 WPF 编程和应用程序模型中对 XAML 文件进行标记编译时，XAML 文件的 XAML 代码隐藏文件的位置是通过如下方式来标识的：以 XAML 根元素的 `x:Class` 特性形式指定一个命名空间和类。

通过目前已介绍的示例，你已了解了几个按钮，但这其中没有一个按钮具有任何与其关联的逻辑行为。为对象元素添加行为的主要应用程序级机制是使用元素类的现有事件，并

为在运行时引发该事件时调用的该事件编写特定的处理程序。在标记中指定事件名称以及要使用的处理程序的名称，而在代码隐藏中定义实现处理程序的代码。

#### XAML

```
<Page x:Class="ExampleNamespace.ExamplePage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <StackPanel>
        <Button Click="Button_Click">Click me</Button>
    </StackPanel>
</Page>
```

#### C#

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;

namespace ExampleNamespace;

public partial class ExamplePage : Page
{
    public ExamplePage() =>
        InitializeComponent();

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        var buttonControl = (Button)e.Source;
        buttonControl.Foreground = Brushes.Red;
    }
}
```

请注意，代码隐藏文件使用 CLR 命名空间 `ExampleNamespace`（该命名空间在 Visual Basic 中不可见）并将 `ExamplePage` 声明为该命名空间内的一个分部类。这相当于 `ExampleNamespace` 的 `x:Class` 特性值。在标记根中提供的 `ExamplePage`。WPF 标记编译器将通过从根元素类型派生一个类，为编译的任何 XAML 文件创建一个分部类。在提供定义同一分部类的代码隐藏时，将在与编译的应用相同的命名空间和类中合并生成的代码。

#### ① 重要

在 Visual Basic 中，XAML 和代码隐藏都隐含根命名空间。只有嵌套命名空间可见。本文演示 C# 项目的 XAML。

若要深入了解 WPF 中代码隐藏编程的要求，请参阅 [WPF 中的代码隐藏、事件处理程序和分部类要求](#)。

如果不需要创建单独的代码隐藏文件，还可以将代码内联到 XAML 文件中。但是，内联代码是一种通用性较低的方法，具有很多的限制。有关详细信息，请参阅 [WPF 中的代码隐藏和 XAML](#)。

## 路由事件

路由事件是一个特殊的事件功能，该功能是 WPF 的基础。路由事件使一个元素可以处理另一个元素引发的事件，前提是这些元素通过树关系连接在一起。使用 XAML 特性指定事件处理时，可以对任何元素（包括未在类成员表中列出该特定事件的元素）侦听和处理该路由事件。这是通过以所属类名限定事件名特性来实现的。例如，在当前讨论的 `StackPanel / Button` 示例中，父 `StackPanel` 可以在 `StackPanel` 对象元素上指定特性 `Button.Click`，并将处理程序名用作特性值，从而为子元素按钮的 `Click` 事件注册一个处理程序。有关详细信息，请参阅[路由事件概述](#)。

## 命名元素

默认情况下，通过处理 XAML 对象元素在对象图中创建的对象实例没有唯一标识符或对象引用。相反，如果在代码中调用构造函数，则几乎总是使用构造函数结果为构造的实例设置变量，以便以后在代码中引用该实例。为了对通过标记定义创建的对象提供标准化访问，XAML 定义了 [x:Name 特性](#)。可以在任何对象元素上设置 `x:Name` 特性的值。在代码隐藏中，所选标识符等效于引用所构造的实例的实例变量。在所有方面，命名元素以类似于对象实例的方式工作（名称引用实例），并且代码隐藏可以通过引用命名元素来处理应用内的运行时交互。实例和变量之间的这种连接是由 WPF XAML 标记编译器实现的，并且更具体地涉及到功能和模式，例如本文中未详细讨论的 [InitializeComponent](#)。

WPF 框架级 XAML 元素继承 `Name` 属性，该属性等效于 XAML 定义的 `x:Name` 特性。其他某些类也为 `x:Name`（通常也定义为 `Name` 属性）提供属性级等效项。一般而言，如果在所选元素/类型的成员表中找不到 `Name` 属性，则可以改用 `x:Name`。`x:Name` 值将通过特定子系统或通过 `FindName` 等实用工具方法，为可在运行时使用的 XAML 元素提供标识符。

下面的示例对 `StackPanel` 元素设置 `Name`。然后，该 `StackPanel` 中 `Button` 上的一个处理程序通过其实例引用 `buttonContainer`（由 `Name` 设置）来引用 `StackPanel`。

XAML

```
<StackPanel Name="buttonContainer">
    <Button Click="RemoveThis_Click">Click to remove this button</Button>
```

```
</StackPanel>
```

C#

```
private void RemoveThis_Click(object sender, RoutedEventArgs e)
{
    var element = (FrameworkElement)e.Source;

    if (buttonContainer.Children.Contains(element))
        buttonContainer.Children.Remove(element);
}
```

和变量一样，实例的 XAML 名称受范围概念约束，因此可以在可预测的某个范围内强制名称唯一。定义页面的主标记表示一个唯一的 XAML 名称范围，而该 XAML 名称范围的边界是该页面的根元素。但是，其他标记源（如样式或样式中的模板）可以在运行时与页面交互，这种标记源常常具有自己的 XAML 名称范围，这些名称范围不一定与页面的 XAML 名称范围相关联。有关 `x:Name` 和 XAML 名称范围的详细信息，请参阅 [Name](#)、[x:Name 指令](#) 或 [WPF XAML 名称范围](#)。

## 附加属性和附加事件

XAML 指定了一种语言功能，该功能允许对任何元素指定某些属性或事件，即使要设置属性或事件的元素的类型定义中不存在该属性或事件也是如此。该功能的属性版本称为附加属性，事件版本称为附加事件。从概念上讲，可以将附加属性和附加事件视为可以在任何 XAML 元素/对象实例上设置的全局成员。但是，元素/类或更大的基础结构必须支持附加值的后备属性存储。

通常通过特性语法来使用 XAML 中的附加属性。在特性语法中，可以采用 `ownerType.propertyName` 的形式指定附加属性。

表面上，这与属性元素用法类似，但在这种情况下，所指定的 `ownerType` 始终是一种与从中要设置附加属性的对象元素不同的类型。`ownerType` 这种类型提供 XAML 处理器为获取或设置附加属性值所需要的访问器方法。

附加属性的最常见方案是使子元素向其父元素报告属性值。

下面的示例演示 `DockPanel.Dock` 附加属性。`DockPanel` 类为 `DockPanel.Dock` 定义访问器，并拥有附加属性。`DockPanel` 类还包括一个逻辑，该逻辑迭代其子元素并具体检查每个元素是否具有 `DockPanel.Dock` 设置值。如果找到一个值，将在布局过程中使用该值定位子元素。使用 `DockPanel.Dock` 附加属性和此定位功能实际上是 `DockPanel` 类激动人心的一面。

XAML

```
<DockPanel>
    <Button DockPanel.Dock="Left" Width="100" Height="20">I am on the
left</Button>
    <Button DockPanel.Dock="Right" Width="100" Height="20">I am on the
right</Button>
</DockPanel>
```

在 WPF 中，大部分或所有附加属性还作为依赖属性实现。有关详细信息，请参阅[附加属性概述](#)。

附加事件使用类似的 `ownerType.eventName` 特性语法形式。和非附加事件一样，XAML 中附加事件的特性值指定对元素处理事件时调用的处理程序方法的名称。在 WPF XAML 中使用附加事件并不常见。有关详细信息，请参阅[附加事件概述](#)。

## 基类型

基础 WPF XAML 及其 XAML 命名空间是类型的一个集合，这些类型对应于 CLR 对象和 XAML 的标记元素。但是，并不是所有的类都能映射到元素。抽象类（如 [ButtonBase](#)）和某些非抽象基类在 CLR 对象模型中用于继承。基类（包括抽象类）对于 XAML 开发仍然很重要，因为每个具体的 XAML 元素都从其层次结构中的某个基类继承成员。通常，这些成员包括可以设置为元素特性的属性或者可以处理的事件。[FrameworkElement](#) 是 WPF 在 WPF 框架级的具体 UI 基类。设计 UI 时，将使用各种形状、面板、装饰器或控件类，它们全部派生自 [FrameworkElement](#)。一个相关的基类

[FrameworkContentElement](#) 使用可在 [FrameworkElement](#) 中特意镜像 API 的 API，支持适合流布局表示形式的面向文档的元素。元素级的特性和 CLR 对象模型的组合提供一组通用的属性，这些属性可以在大多数具体的 XAML 元素上设置，而不管具体的 XAML 元素及其基础类型。

## 安全性

XAML 是一种直接表示对象实例化和执行的标记语言。因此，在 XAML 中创建的元素能够像你的应用代码那样与系统资源进行交互（如网络访问、文件系统 IO）。XAML 还与承载应用具有相同的系统资源访问权限。

## WPF 中的代码访问安全性 (CAS)

与 .NET Framework 不同，适用于 .NET 的 WPF 不支持 CAS。有关详细信息，请参阅[代码访问安全性差异](#)。

## 从代码加载 XAML

XAML 可用于定义整个 UI，但有时也适合在 XAML 中定义一部分 UI。此功能可用于：

- 启用部分自定义。
- 本地存储 UI 信息。
- 对业务对象进行建模。

这些方案的关键是 [XamlReader](#) 类及其 [Load](#) 方法。输入是 XAML 文件，而输出是对象，该对象表示从该标记创建的整个运行时对象树。然后可以插入该对象，作为应用中已存在的另一个对象的属性。只要该属性在内容模型中，并且具有显示功能以通知执行引擎已向应用添加新内容，就可以通过在 XAML 中动态加载来轻松修改正在运行的应用的内容。

## 另请参阅

- [XAML 语法详述](#)
- [XAML 及 WPF 的自定义类](#)
- [XAML 命名空间 \(x:\) 语言功能](#)
- [WPF XAML 扩展](#)
- [基元素概述](#)
- [WPF 中的树](#)

### 在 GitHub 上与我们协作

可以在 GitHub 上找到此内容的源，还可以在其中创建和查看问题和拉取请求。有关详细信息，请参阅[参与者指南](#)。

.NET

### .NET Desktop feedback

The .NET Desktop documentation is open source. Provide feedback here.

 提出文档问题

 提供产品反馈

# XAML 服务

项目 • 2023/05/04

本主题介绍 .NET XAML 服务技术集的功能。其中所述的大部分服务和 API 位于程序集 `System.Xaml` 中。这些服务包括读取器和编写器、架构类和架构支持、工厂、类的特性化、XAML 语言内部支持以及其他 XAML 语言功能。

## 关于本文档

.NET XAML 服务的概念文档假定你之前接触过 XAML 语言，并了解如何将它应用到特定框架（例如 Windows Presentation Foundation (WPF) 或 Windows Workflow Foundation），或者了解特定的技术功能领域，例如 [Microsoft.Build.Framework.XamlTypes](#) 中的生成自定义项功能。本文档不会尝试解释作为标记语言的 XAML 的基础知识、XAML 语法术语或其他介绍性材料。相反，本文档重点介绍如何使用 `System.Xaml` 程序集库中启用的 .NET XAML 服务。其中大多数 API 适用于 XAML 语言集成和扩展性场景。这可能包括以下任何场景：

- 扩展基础 XAML 读取器或 XAML 编写器的功能（直接处理 XAML 节点流；派生自己的 XAML 读取器或 XAML 编写器）。
- 定义不具有特定框架依赖项的 XAML 可用的自定义类型，并对这些类型进行特性化，以将其 XAML 类型系统特征传递给 .NET XAML 服务。
- 将 XAML 读取器或 XAML 编写器托管为应用程序的组件，例如适用于 XAML 标记源的可视化设计器或交互式编辑器。
- 编写 XAML 值转换器（标记扩展、自定义类型的类型转换器）。
- 定义自定义 XAML 架构上下文（对后备类型源使用替代程序集加载技术；使用已知类型查找技术而不是总是反映程序集；使用加载的程序集概念，这些概念不使用公共语言运行时 (CLR) `AppDomain` 及其关联的安全模型）。
- 扩展基础 XAML 类型系统。
- 使用 `Lookup` 或 `Invoker` 技术来影响 XAML 类型系统和类型后备的评估方式。

如果你需要 XAML 语言的介绍性材料，可以查看 [XAML 概述 \(WPF .NET\)](#)。该主题面向不熟悉 Windows Presentation Foundation (WPF) 和不知道如何使用 XAML 标记及 XAML 语言功能的受众介绍 XAML。另一个有用的文档是 [XAML 语言规范](#) 中的介绍性材料。

# .NET 体系结构中的 .NET XAML 服务和 System.Xaml

.NET XAML 服务和 `System.Xaml` 程序集定义了支持 XAML 语言功能所需的大部分内容。其中一项便是 XAML 读取器和 XAML 编写器的基类。添加到 .NET XAML 服务中的最重要功能（以前特定于该框架的任何 XAML 实现中均不存在）是 XAML 的类型系统表示形式。此类型系统表示形式以面向对象的方式表示 XAML，它以 XAML 功能为中心，而不依赖于框架的特定功能。

XAML 类型系统既不受 XAML 源的标记形式或运行时细节的限制，也不受任何特定后备类型系统的限制。XAML 类型系统包括类型、成员、XAML 架构上下文、XML 级别概念和其他 XAML 语言概念或 XAML 内部函数的对象表示形式。使用或扩展 XAML 类型系统可以从 XAML 读取器和 XAML 编写器等类实现派生，并可将 XAML 表示形式的功能扩展为受框架、技术或使用或发出 XAML 的应用程序支持的特定功能。XAML 架构上下文的概念支持从 XAML 对象编写器实现、通过上下文中程序集信息传达的技术后备类型系统和 XAML 节点源的组合来实现实际的对象图写入操作。若要详细了解 XAML 架构概念，请参阅[默认 XAML 架构上下文](#)和 [WPF XAML 架构上下文](#)。

## XAML 节点流、XAML 读取器和 XAML 编写器

要了解 .NET XAML 服务在 XAML 语言和使用 XAML 作为语言的特定技术之间的关系中扮演着何种角色，首先了解 XAML 节点流概念以及该概念如何影响 API 和术语会很有帮助。XAML 节点流是 XAML 语言表示形式与 XAML 表示或定义的对象图之间的概念中间产物。

- XAML 读取器是一个以某种形式处理 XAML 的实体，会生成 XAML 节点流。在 API 中，XAML 读取器由基类 [XamlReader](#) 表示。
- XAML 编写器是一个处理 XAML 节点流并生成其他内容的实体。在 API 中，XAML 编写器由基类 [XamlWriter](#) 表示。

涉及 XAML 的最常见情况有两种：加载 XAML 来实例化对象图，以及保存来自应用程序或工具的对象图并生成 XAML 表示形式（通常以标记形式保存为文本文件）。在本文档中，加载 XAML 和创建对象图通常称为加载路径。在本文档中，保存现有对象图或将其序列化为 XAML 通常称为保存路径。

最常见的加载路径类型如下所示：

- 以采用 UTF 编码的 XML 格式开始使用 XAML 表示形式，并保存为文本文件。
- 将该 XAML 加载到 [XamlXmlReader](#) 中。[XamlXmlReader](#) 是一个 [XamlReader](#) 子类。

- 结果是一个 XAML 节点流。 可以使用 [XamlXmlReader](#) / [XamlReader](#) API 访问此 XAML 节点流中的各个节点。 此时最典型的操作是前进至 XAML 节点流，使用“当前记录”工具处理各个节点。
- 将 XAML 节点流中生成的节点传递到 [XamlObjectWriter](#) API。 [XamlObjectWriter](#) 是一个 [XamlWriter](#) 子类。
- [XamlObjectWriter](#) 根据源 XAML 节点流中的进度编写对象图（一次一个对象）。 对象编写是通过 XAML 架构上下文和可访问后备类型系统和框架的程序集和类型的实现来完成的。
- 在 XAML 节点流的末尾调用 [Result](#) 来获取对象图的根对象。

保存路径的最常见类型如下所述：

- 以整个应用程序运行时的对象图、运行时的 UI 内容和状态，或整体应用程序在运行时的对象表示形式的一小部分开始。
- 从逻辑起始对象（例如应用程序或文档根目录）中，将对象加载到 [XamlObjectReader](#)。[XamlObjectReader](#) 是一个 [XamlReader](#) 子类。
- 结果是一个 XAML 节点流。 可以使用 [XamlObjectReader](#) 和 [XamlReader](#) API 访问此 XAML 节点流中的各个节点。 此时最典型的操作是前进至 XAML 节点流，使用“当前记录”工具处理各个节点。
- 将 XAML 节点流中生成的节点传递到 [XamlXmlWriter](#) API。[XamlXmlWriter](#) 是一个 [XamlWriter](#) 子类。
- [XamlXmlWriter](#) 采用 XML UTF 编码编写 XAML。 可以将其保存为文本文件、流或其他形式。
- 调用 [Flush](#) 以获取最终输出。

若要详细了解 XAML 节点流概念，请参阅[了解 XAML 节点流结构和概念](#)。

## XamlServices 类

并非始终需要处理 XAML 节点流。 如果需要基础的加载路径或基础的保存路径，可以使用 [XamlServices](#) 类中的 API。

- [Load](#) 的各种签名实现了一个加载路径。 你可以加载文件或流，或者可以加载 [XmlReader](#)、[TextReader](#) 或 [XamlReader](#)，它们通过使用读取器的 API 进行加载来包装你的 XAML 输入。

- [Save](#) 的各种签名保存对象图并生成流、文件或 [XmlWriter/TextWriter](#) 实例形式的输出。
- [Transform](#) 通过将一个加载路径和一个保存路径链接为单个操作来转换 XAML。可以将不同的架构上下文或不同的后备类型系统用于 [XamlReader](#) 和 [XamlWriter](#)，这会影响生成的 XAML 的转换方式。

有关如何使用 [XamlServices](#) 的详细信息，请参阅 [XAML Services 类和基本的 XAML 读取或写入](#)。

## XAML 类型系统

XAML 类型系统提供了处理 XAML 节点流的特定单个节点所需的 API。

[XamlType](#) 是对象的表示形式（开始对象节点和结束对象节点之间的处理内容）。

[XamlMember](#) 是对象成员的表示形式（开始成员节点和结束成员节点之间的处理内容）。

[GetAllMembers](#)、[GetMember](#) 和 [DeclaringType](#) 等 API 报告 [XamlType](#) 和 [XamlMember](#) 之间的关系。

由 .NET XAML 服务实现的 XAML 类型系统的默认行为基于公共语言运行时 (CLR) 以及使用反射对程序集中 CLR 类型进行的静态分析。因此，对于特定的 CLR 类型，XAML 类型系统的默认实现可以公开该类型及其成员的 XAML 架构，并根据 XAML 类型系统对其进行报告。在默认的 XAML 类型系统中，类型转让概念映射到 CLR 继承上，实例、值类型等概念也映射到 CLR 的支持行为和功能上。

## XAML 语言功能参考

为支持 XAML，.NET XAML 服务提供了为 XAML 语言 XAML 命名空间定义的 XAML 语言概念的具体实现。它们记录在特定的参考页中。介绍语言功能的角度是这些语言功能在由 .NET XAML 服务定义的 XAML 读取器或 XAML 编写器处理时产生的行为方式。有关详细信息，请参阅 [XAML 命名空间 \(x:\) 语言功能](#)。