

# 高级 (Windows Presentation Foundation)

项目 • 2022/11/09

本节介绍 WPF 中的部分高级区域。

## 本节内容

[WPF 体系结构](#)

[WPF 中的 XAML](#)

[基元素类](#)

[元素树和序列化](#)

[WPF 属性系统](#)

[WPF 中的事件](#)

[输入](#)

[拖放](#)

[资源](#)

[文档](#)

[全球化和本地化](#)

[布局](#)

[从 WPF 迁移到 System.Xaml 的类型](#)

[迁移和互操作性](#)

[“性能”](#)

[线程模型](#)

[非托管 WPF API 参考](#)

# WPF 体系结构

项目 • 2023/02/06

本主题提供 Windows Presentation Foundation (WPF) 类层次结构的指导教程。它介绍了 WPF 的大部分主要子系统，并说明了它们的交互方式。它还详细介绍了 WPF 架构师所做的一些选择。

## System.Object

WPF 主要编程模型通过托管代码公开。在 WPF 的早期设计阶段，曾有过大量关于如何界定系统的托管组件和非托管组件的争论。CLR 提供一系列的功能，可以提高开发效率和可靠性（包括内存管理、错误处理和通用类型系统等），但这是需要付出代价的。

下图说明了 WPF 的主要组件。关系图的红色部分（PresentationFramework、PresentationCore 和 milcore）是 WPF 的主要代码部分。在这些组件中，只有一个是非托管组件 - milcore。milcore 是以非托管代码编写的，目的是实现与 DirectX 的紧密集成。WPF 中的所有显示均通过 DirectX 引擎完成，因此硬件和软件呈现都很高效。WPF 还要求对内存和执行进行精细控制。milcore 中的组合引擎受性能影响极大，需要放弃 CLR 的许多优点来提高性能。

PresentationFramework

PresentationCore

Common Language Runtime

milcore

User32

DirectX

Kernel

本主题的后面部分将讨论 WPF 的托管和非托管部分之间的通信。下面介绍托管编程模型的其余部分。

## System.Threading.DispatcherObject

WPF 中的大多数对象派生自 [DispatcherObject](#)，这提供了用于处理并发和线程的基本构造。WPF 基于调度程序实现的消息传递系统。其工作方式与常见的 Win32 消息泵非常类似；事实上，WPF 调度程序使用 User32 消息执行跨线程调用。

要讨论 WPF 中的并发，首先必须真正理解两个核心概念 - 调度程序和线程关联。

在 WPF 的设计阶段，目标是移动到单一线程的执行，但这不是一种与线程“关联”的“模型”。当一个组件使用执行线程的标识来存储某种类型的状态时，将发生线程关联。最常见的形式是使用线程本地存储 (TLS) 来存储状态。线程关联要求执行的每个逻辑线程仅由操作系统中的一个物理线程所拥有，这将占用大量内存。最后，WPF 的线程处理模型通过线程关联与单一线程执行的现有 User32 线程处理模型保持同步。主要原因是互操作性 - 类似于 OLE 2.0 的系统、剪贴板和 Internet Explorer 均需要单一线程关联 (STA) 执行。

假设你具有带有 STA 线程的对象，则需要在线程之间通信并验证你是否位于正确的线程上的一种方法。调度程序的作用就在于此。调度程序是一个基本的消息调度系统，具有多个按优先顺序排列的队列。消息的示例包括原始输入通知（鼠标移动）、框架函数（布局）或用户命令（执行此方法）。通过从 [DispatcherObject](#) 派生，可以创建一个具有 STA 行为的 CLR 对象，并在创建时获得一个指向调度程序的指针。

## System.Windows.DependencyObject

生成 WPF 时使用的主要体系结构原理之一是首选属性而不是方法或事件。属性具有声明性，可更方便地指定用途而不是操作。它还支持模型驱动或数据驱动的系统，以显示用户界面内容。这种理念的预期效果是创建更多可以绑定到的属性，从而更好地控制应用程序的行为。

为了更加充分地利用由属性驱动的系统，需要一个比 CLR 提供的功能更丰富的属性系统。这种丰富性的一个简单示例是更改通知。若要实现双向绑定，需要绑定的双方支持更改通知。若要使行为与属性值相关联，需要在属性值更改时收到通知。Microsoft .NET Framework 具有一个 [INotifyPropertyChanged](#) 接口，对象通过该接口可以发布更改通知，但该接口是可选的。

WPF 提供一个更丰富的属性系统，该属性系统从 [DependencyObject](#) 类型派生。该属性系统实际是一个“依赖”属性系统，因为它会跟踪属性表达式之间的依赖关系，并在依赖关系更改时自动重新验证属性值。例如，如果具有一个可继承的属性（如 [FontSize](#)），当继承该值的元素的父级发生属性更改时，系统会自动更新。

WPF 属性系统的基础是属性表达式的概念。在 WPF 的第一个版本中，属性表达式系统是关闭的，表达式均作为框架的一部分提供。表达式致使属性系统不具有数据绑定、样式调整或继承硬编码，而是由框架内后面的层来提供这些功能。

属性系统还提供属性值的稀疏存储。因为对象可能有数十个（如果达不到上百个）属性，并且大部分值处于其默认状态（被继承、由样式设置等），所以并非对象的每个实例都需要具有在其上定义的每个属性的完全权重。

属性系统的最后一个新功能是附加属性的概念。WPF 元素是基于组合和组件重用的原则生成的。某些包含元素（如 Grid 布局元素）通常需要子元素上的其他数据才能控制其行为（如行/列信息）。任何对象都可以为任何其他对象提供属性定义，而不是将所有这些属性与每个元素相关联。这与 JavaScript 中的“expando”功能相似。

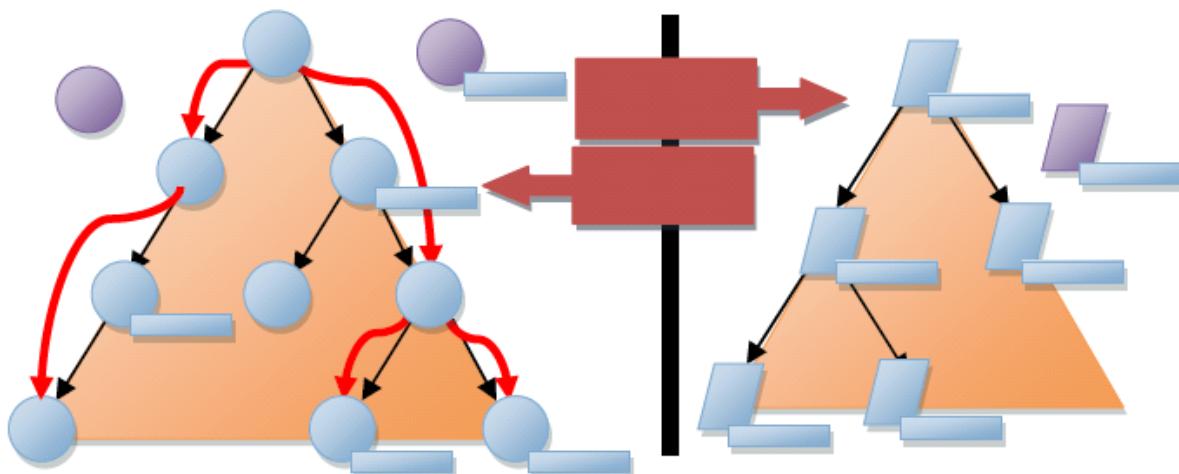
## System.Windows.Media.Visual

定义一个系统后，下一步是将像素绘制到屏幕上。Visual 类用于生成视觉对象的树，每个树可以选择性地包含绘制指令以及有关如何呈现这些指令（剪裁、转换等）的元数据。Visual 极其轻量且灵活，因此大部分功能没有公共 API 公开，并且非常依赖受保护的回调函数。

Visual 实际上是 WPF 复合系统的入口点。Visual 是托管 API 和非托管 milcore 这两个子系统之间的连接点。

WPF 通过遍历由 milcore 管理的非托管数据结构来显示数据。这些结构（称为组合节点）代表层次结构显示树，其中每个节点都有呈现指令。只能通过消息传递协议来访问此树（如下图右侧所示）。

对 WPF 编程时，将创建 Visual 元素及派生的类型，它们通过此消息传递协议在内部与此组合树进行通信。WPF 中的每个 Visual 可以不创建组合节点，也可以创建一个或多个组合节点。



请注意一个非常重要的体系结构细节 - 会缓存整个可视化树和绘制指令。在图形方面，WPF 使用保留呈现系统。这可以实现以高刷新率重绘系统，并且组合系统不会阻止对用户代码的回调。这有助于防止出现应用程序无响应的情况。

关系图中容易忽略的另一个重要细节是系统实际执行组合的方式。

在 User32 和 GDI 中，系统在一个即时模式剪裁系统上工作。当需要绘制一个组件时，系统会建立一个剪裁边界，在此边界外，不允许组件接触像素，然后会要求组件在该框中绘制像素。此系统在内存受限的系统上工作良好，因为当某些内容更改时，只需处理受影响的组件即可 - 不会由两个组件同时处理一个像素的颜色。

WPF 使用“绘画器的算法”绘制模型。要求每个组件从显示内容的背面绘制到正面，而不是剪裁每个组件。这允许每个组件在先前组件的显示内容上绘制。此模型的优点是可以生成部分透明的复杂形状。通过使用当今的新式图形硬件，此模型的速度相对较快（创建 User32/ GDI 时则不然）。

如上所述，WPF 的一个核心原理是转移到一个更具声明性且“以属性为中心”的编程模型。在可视化系统中，这体现在有意思的几个方面。

首先，对于保留的模式图形系统，这实际上是从命令性 DrawLine/DrawLine 类型模型移动到面向数据的模型 new Line()/new Line()。通过这种向数据驱动的绘制的移动，可以使用属性表达绘制指令上的复杂操作。从 [Drawing](#) 派生的类型实际上是用于呈现的对象模型。

第二，如果评估动画系统，你会发现它几乎是完全声明性的。可以将动画表示为动画对象上的一组属性，无需求求开发人员计算下一个位置或下一个颜色。这些动画可以表达开发人员或设计人员的意图（在 5 秒内将此按钮从一个位置移动到另一个位置），系统可以确定完成此任务的最高效方式。

## System.Windows.UIElement

[UIElement](#) 定义核心子系统，包括布局、输入和事件。

布局是 WPF 中的核心概念。在许多系统中，可能有一组固定的布局模型（HTML 支持三种布局模型：流、绝对和表），也可能没有布局模型（User32 实际仅支持绝对定位）。WPF 先假设开发人员和设计人员需要灵活的可扩展布局模型，该模型可能是由属性值而不是命令性逻辑驱动的。在 [UIElement](#) 级别，会引入布局的基本协定 - 具有 [Measure](#) 和 [Arrange](#) 阶段的两阶段模型。

[Measure](#) 允许组件确定其要采用的大小。此阶段独立于 [Arrange](#)，因为在许多情形下，父元素会要求子元素测量若干次，以确定其最佳位置和大小。父元素要求子元素测量这一事实体现了 WPF 的另一关键原则 - 调整内容大小。WPF 中的所有控件都支持调整到

内容自然大小的功能。这使本地化更加容易，并可实现调整内容大小时进行动态元素布局。[Arrange](#) 阶段允许父元素定位并确定每个子元素的最终大小。

通常会花费大量的时间来讨论 WPF 的输出端 ([Visual](#)) 及其相关对象。然而，在输入端也有许多创新。WPF 输入模型中的最基本更改也许是一致模型，借助此模型可通过系统对输入事件进行路由。

输入是作为内核模式设备驱动程序上的信号发出的，并通过涉及 Windows 内核和 User32 的复杂过程路由到正确的进程和线程。与输入相对应的 User32 消息路由到 WPF 后，转换为 WPF 原始输入消息并发送到调度程序。WPF 允许将原始输入事件转换为多个实际事件，在保证传递到位的情况下在低系统级别实现类似“MouseEnter”的功能。

每个输入事件至少会转换为两个事件 - “预览”事件和实际事件。WPF 中的所有事件都具有通过元素树路由的概念。如果事件从目标向上遍历树直到根，称为“浮升”，如果从根开始向下遍历到目标，称为“隧道”。输入预览事件隧道，使树中的任何元素都有机会筛选事件或对事件采取操作。然后，常规（非预览）事件将从目标向上浮升到根。

隧道和浮升阶段之间的划分使键盘快捷键等功能的实现在复合环境中采用一致的方式。在 User32 中，可以通过使用一个全局表来实现键盘快捷键，该表中包含你希望支持的所有快捷键（[Ctrl+N](#) 映射到“新建”）。在应用程序的调度程序中，可以调用 [TranslateAccelerator](#)，它会探查 User32 中的输入消息，并确定是否有任何消息与已注册的快捷键匹配。在 WPF 中，上述内容无效，因为系统是完全“可组合”的 - 任何元素都可以处理和使用任何键盘快捷键。将此两阶段模型用于输入，可允许组件实现其自己的“[TranslateAccelerator](#)”。

为了进一步深化此功能，[UIElement](#) 还引入了 [CommandBindings](#) 的概念。WPF 命令系统允许开发人员以命令终结点（一种用于实现  [ICommand](#) 的功能）的方式定义功能。命令绑定使元素可以定义输入笔势（[Ctrl+N](#)）和命令（“新建”）之间的映射。输入笔势和命令定义都是可扩展的，并且可以在使用时联系到一起。这使得一些操作（例如，允许最终用户自定义其要在应用程序内使用的键绑定）显得无关紧要。

至此，本主题已重点讨论了 WPF 的“核心”功能 - PresentationCore 程序集中实现的功能。生成 WPF 时，明确划分基础部分（例如带有 Measure 和 Arrange 的布局的协定）和框架部分（例如 [Grid](#) 等特定布局的实现）是所需的结果。目标是提供在堆栈中处于较低位置的可扩展性点，这将允许外部开发人员在需要时创建自己的框架。

## System.Windows.FrameworkElement

可以按两种不同的方式来看待 [FrameworkElement](#)。它在 WPF 的较低层中引入的子系统上引入一组策略和自定义项。它还引入了一组新的子系统。

[FrameworkElement](#) 引入的主要策略与应用程序布局有关。[FrameworkElement](#) 基于 [UIElement](#) 引入的基本布局协定构建而成，并增加了布局“插槽”的概念，布局“插槽”可使

布局作者更方便地拥有一组一致的属性驱动的布局语义。`HorizontalAlignment`、`VerticalAlignment`、`MinWidth` 和 `Margin` 等属性使得从 `FrameworkElement` 派生的所有组件在布局容器内具有一致的行为。

利用 `FrameworkElement`，WPF 的核心层中的许多功能可以更方便地进行 API 公开。例如，`FrameworkElement` 通过 `BeginStoryboard` 方法提供对动画的直接访问。`Storyboard` 提供针对一组属性为多个动画编写脚本的一种方式。

`FrameworkElement` 引入的两个最关键的内容是数据绑定和样式。

曾经使用 Windows 窗体或 ASP.NET 创建应用程序用户界面 (UI) 的用户应当对 WPF 中的数据绑定子系统较为熟悉。在上述每个系统中，可通过一种简单的方式来表达希望将给定元素中的一个或多个属性绑定到一个数据片段。WPF 完全支持属性绑定、转换和列表绑定。

WPF 中数据绑定的最值得关注的功能之一是引入了数据模板。利用数据模板，可以通过声明方式指定某个数据片断的可视化方式。无需创建可绑定到数据的自定义用户界面，而是转而让数据来确定要创建的显示内容。

样式实际上是轻量型的数据绑定。使用样式，可以将共享定义的一组属性绑定到元素的一个或多个实例。通过显式引用（通过设置 `Style` 属性）或通过将样式与元素的 CLR 类型隐式关联，可以将样式应用到元素。

## System.Windows.Controls.Control

控件的最重要功能是模板化。如果将 WPF 的组合系统视为一个保留模式绘制系统，则控件可通过模板化以一种参数化的声明性方式描述其绘制。`ControlTemplate` 实际上只是用于创建一组子元素的脚本，绑定到由控件提供的属性。

`Control` 提供一组常用属性，例如 `Foreground`、`Background`、`Padding`，模板作者可以使用这些常用属性来自定义控件的显示。控件的实现提供了数据模型和交互模型。交互模型定义了一组命令（如窗口的“关闭”），以及到输入笔势的绑定（如单击窗口右上角的红叉）。数据模型提供了一组属性，用于自定义交互模型或自定义显示内容（由模板确定）。

数据模型（属性）、交互模型（命令和事件）及显示模型（模板）之间的划分，可实现对控件的外观和行为的完全自定义。

最常见的控件数据模型是内容模型。如果查看 `Button` 之类的控件，会看到它具有一个 `Object` 类型的名为“Content”的属性。在 Windows 窗体和 ASP.NET 中，此属性通常是一个字符串 - 不过，这会限制可以在按钮中添加的内容类型。按钮的内容可以是简单的字符串、复杂的数据对象或整个元素树。如果是数据对象，可以使用数据模板构造显示内容。

# 总结

WPF 可创建动态的数据驱动的呈现系统。 系统的每一部分均可通过驱动行为的属性集来创建对象。 数据绑定是系统的基础部分，在每一层中均进行了集成。

传统的应用程序创建一个显示内容，然后绑定到某些数据。在 WPF 中，控件的所有内容、显示内容的所有方面都是由某种类型的数据绑定生成的。通过在按钮内部创建复合控件并将其显示内容绑定到按钮的内容属性，会显示按钮中的文本。

开始开发基于 WPF 的应用程序时，你应感到非常熟悉。设置属性、使用对象和数据绑定的方式与使用 Windows 窗体或 ASP.NET 的方式极为相似。如果对 WPF 体系结构有更深的了解，将能够创建更丰富的应用程序，这些应用程序从根本上会将数据视为应用程序的核心驱动要素。

## 另请参阅

- [Visual](#)
- [UIElement](#)
- [ICommand](#)
- [FrameworkElement](#)
- [DispatcherObject](#)
- [CommandBinding](#)
- [Control](#)
- [数据绑定概述](#)
- [布局](#)
- [动画概述](#)

# WPF 中的 XAML

项目 • 2022/09/27

WPF 类型的实现使它们能为 XAML 表示提供所需的类型支持。通常，你可以在 XAML 标记中创建大多数 WPF 应用程序 UI。

## 本节内容

[WPF 中的 XAML](#)  
[XAML 语法详述](#)  
[WPF 中的代码隐藏和 XAML](#)  
[XAML 及 WPF 的自定义类](#)  
[标记扩展和 WPF XAML](#)  
[WPF XAML 的 XAML 命名空间和命名空间映射](#)  
[WPF XAML 名称范围](#)  
[内联样式和模板](#)  
[XAML 中的空白处理](#)  
[TypeConverters 和 XAML](#)  
[XML 字符实体和 XAML](#)  
[XAML 命名空间 \(x:\) 语言功能](#)  
[WPF XAML 扩展](#)  
[标记兼容 \(mc:\) 语言功能](#)

## 相关章节

[WPF 体系结构](#)  
[基元素](#)  
[元素树和序列化](#)  
[属性](#)  
[事件](#)  
[输入](#)  
[资源](#)  
[样式设置和模板化](#)  
[线程模型](#)

# WPF (.NET Framework) 中的 XAML 概述

项目 • 2023/02/18

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。

## 什么是 XAML

XAML 是一种声明性标记语言。应用于.NET Framework 编程模型时，XAML 简化了为.NET Framework 应用的 UI 创建过程。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，还将创建指定为 `Content` 的显示文本。

#### 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--&gt;
&lt;Border&gt;
    &lt;Border.Child&gt;
        &lt;TextBox Width="300"/&gt;
    &lt;/Border.Child&gt;
&lt;/Border&gt;</pre>
```

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` 属性采用类型 `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`：

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ExampleNamespace.ExamplePage">
    <Button Click="Button_Click" >Click Me!</Button>
</Page>
```

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。

XAML

```
<Page.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
  <Style TargetType="Border" x:Key="PageBackground">
```

```
<Setter Property="Background" Value="Blue"/>
</Style>
```

XAML

```
</Page.Resources>
<StackPanel>
    <Border Style="{StaticResource PageBackground}">
```

XAML

```
    </Border>
</StackPanel>
```

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 Framework 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](#)。

## XAML 根元素和 XAML 命名空间

一个 XAML 文件只能有一个根元素，这样才能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
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
```

XAML

```
</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，为了引用来自 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](#)。

# XAML 中的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，实现该类型的目的就是在支持常规 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
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ExampleNamespace.ExamplePage">
    <Button Click="Button_Click" >Click Me!</Button>
</Page>
```

C#

```
namespace ExampleNamespace
{
    public partial class ExamplePage
    {
        void Button_Click(object sender, RoutedEventArgs e)
        {
            Button b = e.Source as Button;
            b.Foreground = Brushes.Red;
        }
    }
}
```

请注意，代码隐藏文件使用 CLR 命名空间 `ExampleNamespace` 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。

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](#)。

## XAML 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">
```

XAML

```
<Button Click="RemoveThis">Click to remove this button</Button>
</StackPanel>
```

C#

```
void RemoveThis(object sender, RoutedEventArgs e)
{
    FrameworkElement fe = e.Source as FrameworkElement;
    if (buttonContainer.Children.Contains(fe))
    {
        buttonContainer.Children.Remove(fe);
    }
}
```

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 指定了一个语言功能，该功能允许对任何元素指定某些属性或事件，而不管要设置属性或事件的元素的类型定义中是否存在该属性或事件。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，因此拥有附加属性。`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](#)。

## 基类型和 XAML

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 对象以及 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。

## XAML 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等效的生成代码那样与系统资源（如网络访问、文件系统 IO）进行交互。加载到完全受信任的应用中的 XAML 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)

适用于 .NET Framework 的 WPF 支持代码访问安全性 (CAS)。这意味着在 Internet 区域中运行的 WPF 内容具有缩减的执行权限。“宽松型 XAML”（由 XAML 查看器在加载时解释的非编译 XAML 的页面）和 XBAP 通常在此 Internet 区域中运行，并且使用相同的权限集。但是，加载到完全受信任的应用程序中的 XAML 与承载应用程序具有相同的系统资源访问权限。有关详细信息，请参阅[WPF 部分信任安全性](#)。

## 从代码中加载 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。此功能可用于实现部分自定义、在本地存储信息，使用 XAML 提供业务对象或者各种可能的方案。这些方案的关键是 [XmlReader](#) 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是一个合适的属性，而该内容模型具有最终

显示功能并且将通知执行引擎已向应用添加新内容，就可以通过在 XAML 中进行加载轻松地修改正在运行的应用的内容。对于 .NET Framework，通常只在完全受信任的应用程序中使用此功能，因为将文件加载到正在运行的应用程序中会带来明显的安全隐患。

## 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](#)

# XAML 语法详述

项目 • 2023/02/06

本主题定义用于描述 XAML 语法元素的术语。在本文档的其余部分，这些术语经常专门用于 WPF 文档，以及用于使用 XAML 或是 XAML 语言支持在 System.Xaml 级别启用的基本 XAML 概念的其他框架。本主题扩展了在 [WPF 中的 XAML](#) 主题中介绍的基本术语。

## XAML 语言规范

此处定义的 XAML 语法术语也在 XAML 语言规范中进行定义或引用。XAML 是基于 XML 的语言，遵循或扩展 XML 结构规则。某些术语来自或基于描述 XML 语言或 XML 文档对象模型时常用的术语。

有关 XAML 语言规范的详细信息，请从 Microsoft 下载中心下载 [\[MS-XAML\]](#)。

## XAML 和 CLR

XAML 是一种标记语言。顾名思义，公共语言运行时 (CLR) 可实现运行时执行。XAML 本身不是 CLR 运行时直接使用的公共语言之一。相反，可以将 XAML 视为支持其自己的类型系统。WPF 使用的特定 XAML 分析系统基于 CLR 和 CLR 类型系统构建。XAML 类型映射到 CLR 类型，以便在分析适用于 WPF 的 XAML 时实例化运行时表示形式。因此，本文档中语法讨论的其余部分会包含对 CLR 类型系统的引用，尽管 XAML 语言规范中的等效语法讨论不包含这些引用。（按照 XAML 语言规范级别，XAML 类型可以映射到任何其他类型系统，这不必是 CLR，但这需要创建和使用其他 XAML 分析器。）

## 类型和类继承的成员

显示为 WPF 类型的 XAML 成员的属性和事件通常是从基类型继承的。例如，请考虑以下示例：`<Button Background="Blue" .../>`。如果要查看类定义、反射结果或文档，`Background` 属性不是 `Button` 类中的立即声明属性。相反，`Background` 从基类 `Control` 进行继承。

WPF XAML 元素的类继承行为与 XML 标记的架构强制解释有显著区别。类继承可能变得十分复杂，特别是在中间基类是抽象类，或者涉及接口时。这是 XAML 元素集及其允许属性难以使用通常用于 XML 编程的架构类型（如 DTD 或 XSD 格式）准确且完整地表示的原因之一。另一个原因是 XAML 语言本身的扩展性和类型映射功能排除了允许类型和成员的任何固定表示形式的完整性。

## 对象元素语法

对象元素语法是 XAML 标记语法，通过声明 XML 元素来实例化 CLR 类或结构。此语法类似于其他标记语言（如 HTML）的元素语法。对象元素语法以左尖括号 (<) 开头，后面紧跟进行实例化的类或结构的类型名称。零个或多个空格可以跟在类型名称后面，零个或多个属性也可以在对象元素上声明，使用一个或多个空格分隔每个属性名称=“值”对。最后，必须满足以下条件之一：

- 元素和标记必须以正斜杠 (/) 结尾，后面紧跟右尖括号 (>)。
- 开始标记必须以右尖括号 (>) 结尾。其他对象元素、属性元素或内部文本可以跟在开始标记后面。此处可能包含的确切内容通常受元素的对象模型约束。对象元素的等效结束标记也必须存在，与其他开始和结束标记对进行正确的嵌套和平衡。

.NET 实现的 XAML 有一组规则，这些规则将对象元素映射到类型、将特性映射到属性或事件，以及将 XAML 命名空间映射到 CLR 命名空间和程序集。对于 WPF 和 .NET，XAML 对象元素会映射到引用程序集中定义的 .NET 类型，特性会映射到这些类型的成员。在 XAML 中引用 CLR 类型时，还可以访问该类型的继承成员。

例如，以下示例是对象元素语法，它实例化 [Button](#) 类的新实例的，还指定 [Name](#) 特性以及该特性的值：

XAML

```
<Button Name="CheckoutButton"/>
```

以下示例是还包括 XAML 内容属性语法的对象元素语法。其中包含的内部文本会用于设置 [TextBox](#) XAML 内容属性 [Text](#)。

XAML

```
<TextBox>This is a Text Box</TextBox>
```

## 内容模型

类在语法方面可能支持用作 XAML 对象元素，但只有将该元素置于整体内容模型或元素树的预期位置时，它才会在应用程序或页面中正常运行。例如，[MenuItem](#) 通常应仅作为 [MenuBase](#) 派生类（例如 [Menu](#)）的子级进行放置。特定元素的内容模型在可用作 XAML 元素的控件和其他 WPF 类的类页面上记录为注解的一部分。

## 对象元素的属性

XAML 中的属性通过各种可能的语法进行设置。根据所设置的属性的基础类型系统特征，可用于特定属性的语法会有所不同。

通过设置属性的值，可在对象存在于运行时对象图中时，向它们添加功能或特征。从对象元素创建的对象的初始状态基于无参数构造函数行为。通常，应用程序会使用任何给定对象的完全默认实例以外的其他内容。

## 特性语法（属性）

特性语法是 XAML 标记语法，它通过对现有对象元素声明特性来设置属性的值。特性名称必须与支持相关对象元素的类的属性的 CLR 成员名称匹配。特性名称后跟赋值运算符 (=)。特性值必须是括在引号中的字符串。

### ① 备注

可以使用交替引号在特性中放置文本引用。例如，可以使用单引号来声明其中包含双引号字符的字符串。无论是使用单引号还是双引号，都应使用匹配对来开始和结束特性值字符串。还有一些转义序列或其他方法可用于解决任何特定 XAML 语法施加的字符限制。请参阅 [XML 字符实体和 XAML](#)。

若要通过特性语法进行设置，属性必须是公共属性，并且必须可写。后备类型系统中属性的值必须是值类型，或者必须是在访问相关后备类型时可由 XAML 处理器实例化或引用的引用类型。

对于 WPF XAML 事件，作为特性名称引用的事件必须是公共的，并且具有公共委托。

属性或事件必须是由包含对象元素实例化的类或结构的成员。

## 特性值的处理

左和右引号中包含的字符串值由 XAML 处理器进行处理。对于属性，默认处理行为由基础 CLR 属性的类型确定。

特性值会按此处理顺序，使用以下内容之一进行填充：

1. 如果 XAML 处理器遇到大括号或派生自 [MarkupExtension](#) 的对象元素，则首先计算引用的标记扩展，而不是将值作为字符串进行处理，标记扩展返回的对象会用作值。在许多情况下，标记扩展返回的对象会是对现有对象的引用，或者是将计算延迟到运行时的表达式，不是新实例化的对象。
2. 如果使用特性化 [TypeConverter](#) 声明属性，或者使用特性化 [TypeConverter](#) 声明该属性的值类型，则特性的字符串值会作为转换输入提交到类型转换器，转换器会返回新的对象实例。

3. 如果没有 [TypeConverter](#)，则尝试直接转换为属性类型。此最终级别是在 XAML 语言基元类型之间对分析器本机值进行直接转换，或者检查枚举中命名常量的名称（分析器随后会访问匹配值）。

## 枚举特性值

XAML 中的枚举由 XAML 分析器以内部方式进行处理，枚举的成员应通过指定枚举命名常量之一的字符串名称来指定。

对于非标志枚举值，本机行为是处理特性值的字符串并将它解析为枚举值之一。无需如同在代码中一样，以“枚举.值”格式指定枚举。而是仅指定值，枚举通过所设置的属性类型进行推断。如果以“枚举.值”形式指定特性，则不会正确进行解析。

对于按标志枚举，行为基于 [Enum.Parse](#) 方法。可以通过用逗号分隔每个值，为按标志枚举指定多个值。但是，不能合并不按标志的枚举值。例如，不能使用逗号语法尝试创建对非标志枚举的多个条件执行操作的 [Trigger](#)：

XAML

```
<!--This will not compile, because Visibility is not a flagwise
enumeration.-->
...
<Trigger Property="Visibility" Value="Collapsed,Hidden">
    <Setter ... />
</Trigger>
...
```

支持在 XAML 中可设置的特性的按标志枚举在 WPF 中很少见。不过，其中这样一个枚举是 [StyleSimulations](#)。例如，可以使用逗号分隔的按标志特性语法修改 [Glyphs](#) 类的注解中提供的示例；`StyleSimulations = "BoldSimulation"` 可能变为 `StyleSimulations = "BoldSimulation,ItalicSimulation"`。[KeyBinding.Modifiers](#) 是可以在其中指定多个枚举值的另一个属性。但是，此属性恰好是特殊情况，因为 [ModifierKeys](#) 枚举支持其自己的类型转换器。修饰符的类型转换器使用加号 (+) 作为分隔符，而不是逗号 (,)。此转换支持更传统的语法，以便在 Microsoft Windows 编程中表示组合键，例如“Ctrl+Alt”。

## 属性和事件成员名称引用

指定属性时，可以引用以针对包含对象元素实例化的 CLR 类型的成员形式而存在的任何属性或事件。

或者，可以引用独立于包含对象元素的附加属性或附加事件。（附加属性会在后面的部分中进行讨论。）

还可以使用 `typeName.event` 部分限定名称对可通过默认命名空间访问的任何对象中的任何事件进行命名；此语法支持为路由事件附加处理程序，其中处理程序旨在处理从子元素路由的事件，但父元素在其成员表中也不包含该事件。此语法类似于附加事件语法，但此处的事件不是真正的附加事件。而是使用限定名称引用事件。有关详细信息，请参阅[路由事件概述](#)。

对于某些情形，属性名称有时作为特性的值提供，而不是特性名称。该属性名称也可以包含限定符，如采用 `ownerType.dependencyPropertyName` 形式指定的属性。在 XAML 中编写样式或模板时，这种情形很常见。作为特性值提供的属性名称的处理规则有所不同，受进行设置的属性的类型或特定 WPF 子系统的行为所控制。有关详细信息，请参阅[样式设置和模板化](#)。

属性名称的另一种用法是当特性值描述属性-属性关系时。此功能用于数据绑定和情节提要目标，通过 `PropertyPath` 类及其类型转换器来实现。有关查找语义的更完整说明，请参阅[PropertyPath XAML 语法](#)。

## 属性元素语法

属性元素语法是一种与元素的基本 XML 语法规则有一些差异的语法。在 XML 中，特性的值是事实上的字符串，唯一可能的变化是所使用的字符串编码格式。在 XAML 中，可以分配其他对象元素作为属性的值。属性元素语法在默认情况下会启用此功能。属性不是在元素标记中指定为特性，而是采用 `elementTypeName.propertyName` 形式，使用开始元素标记进行指定，在其中指定属性的值，然后结束属性元素。

具体而言，语法以左尖括号 (<) 开头，后面紧跟属性元素语法包含在其中的类或结构的类型名称。紧接着是单个点 (.)，然后是属性的名称，再然后是右尖括号 (>)。与特性语法一样，该属性必须存在于指定类型的已声明公共成员中。要分配给属性的值包含在属性元素中。通常，值作为一个或多个对象元素提供，因为将对象指定为值是属性元素语法旨在解决的情形。最后，必须提供指定同一个 `elementTypeName.propertyName` 组合的等效结束标记，与其他元素标记进行正确的嵌套和平衡。

例如，下面是 `Button` 的 `ContextMenu` 属性的属性元素语法。

XAML

```
<Button>
  <Button.ContextMenu>
    <ContextMenu>
      <MenuItem Header="1">First item</MenuItem>
      <MenuItem Header="2">Second item</MenuItem>
    </ContextMenu>
  </Button.ContextMenu>
  Right-click me!</Button>
```

如果指定的属性类型是基元值类型（如 [String](#)），或是在其中指定了名称的枚举，属性元素中的值也可以作为内部文本提供。这两种用法有些不常见，因为每种情况也可以使用更简单的特性语法。使用字符串填充属性元素的一种情形是用于不是 XAML 内容属性，但仍用于 UI 文本表示形式的属性，特定空白元素（如换行符）需要出现在该 UI 文本中。特性语法不能保留换行符，但属性元素语法可以，只要有效空白保留处于活动状态（有关详细信息，请参阅 [XAML 中的空白处理](#)）。另一种情形是可以将 [x:Uid 指令](#) 应用于属性元素，从而将其中的值标记为应在 WPF 输出 BAML 中或通过其他技术本地化的值。

WPF 逻辑树中未表示属性元素。属性元素只是用于设置属性的特定语法，不是具有支持它的实例或对象的元素。（有关逻辑树概念的详细信息，请参阅 [WPF 中的树](#)。）

对于支持特性和属性元素语法的属性，这两种语法通常具有相同的结果，不过细微之处（如空白处理）可能在语法之间略有不同。

## 集合语法

XAML 规范要求 XAML 处理器实现标识值类型为集合的属性。[.NET 中的常规 XAML 处理器](#) 实现基于托管代码和 CLR，它通过下列方法之一标识集合类型：

- 类型实现  [IList](#)。
- 类型实现  [IDictionary](#)。
- 类型派生自  [Array](#)（有关 XAML 中数组的详细信息，请参阅  [x:Array 标记扩展](#)。）

如果属性的类型是集合，则推断的集合类型不需要在标记中指定为对象元素。相反，旨在成为集合中的项的元素被指定为属性元素的一个或多个子元素。每个此类项会在加载期间计算为对象，并通过调用隐式集合的 `Add` 方法来添加到集合中。例如，[Style](#) 的 [Triggers](#) 属性采用实现  [IList](#) 的专用集合类型  [TriggerCollection](#)。不需要实例化标记中的  [TriggerCollection](#) 对象元素。而是将一个或多个  [Trigger](#) 项指定为 `style.Triggers` 属性元素中的元素，其中  [Trigger](#)（或派生类）是应作为强类型化的隐式  [TriggerCollection](#) 的项类型的类型。

XAML

```
<Style x:Key="SpecialButton" TargetType="{x:Type Button}">
  <Style.Triggers>
    <Trigger Property="Button.IsChecked" Value="true">
      <Setter Property = "Background" Value="Red"/>
    </Trigger>
    <Trigger Property="Button.IsChecked" Value="false">
      <Setter Property = "Background" Value="Green"/>
    </Trigger>
```

```
</Style.Triggers>  
</Style>
```

属性可以是集合类型以及该类型和派生类型和 XAML 内容属性，本主题下一节对此进行了讨论。

隐式集合元素会在逻辑树表示形式中创建成员，即使它不会作为元素出现在标记中。通常，父类型的构造函数对作为其属性之一的集合执行实例化，而最初为空的集合会成为对象树的一部分。

### ① 备注

集合检测不支持泛型列表和字典接口（`IList<T>` 和 `IDictionary< TKey, TValue >`）。但是，可将 `List<T>` 类用作基类，因为它直接实现 `IList`；或将 `Dictionary< TKey, TValue >` 用作基类，因为它直接实现 `IDictionary`。

在集合类型的 .NET 参考页中，这种故意对集合省略对象元素的语法有时会在 XAML 语法部分中注明为隐式集合语法。

除了根元素之外，XAML 文件中作为另一个元素的子元素嵌套的每个对象元素实际上是以下一种或两种情况的元素：其父元素的隐式集合属性的成员，或者为父元素指定 XAML 内容属性值的元素（XAML 内容属性会在后面的部分中进行讨论）。换句话说，标记页中的父元素和子元素的关系实际上是根处的单个对象，并且根下的每个对象元素都是提供父级的属性值的单个实例，或者是集合中也是父级的集合类型属性值的一个项。此单根概念与 XML 相同，在加载 XAML 的 API（如 `Load`）行为中经常得到加强。

以下示例是一种语法，其中具有显式指定的集合 (`GradientStopCollection`) 的对象元素。

#### XAML

```
<LinearGradientBrush>  
  <LinearGradientBrush.GradientStops>  
    <GradientStopCollection>  
      <GradientStop Offset="0.0" Color="Red" />  
      <GradientStop Offset="1.0" Color="Blue" />  
    </GradientStopCollection>  
  </LinearGradientBrush.GradientStops>  
</LinearGradientBrush>
```

请注意，并不总是可以显式声明集合。例如，尝试在前面所示的 `Triggers` 示例中显式声明 `TriggerCollection` 会失败。显式声明集合要求集合类必须支持无参数构造函数，而 `TriggerCollection` 没有无参数构造函数。

# XAML 内容属性

XAML 内容语法是一种仅在指定 ContentPropertyAttribute 作为类声明一部分的类上启用的语法。 ContentPropertyAttribute 会引用作为该类型元素（包括派生类）的内容属性的属性名称。由 XAML 处理器进行处理时，在对象元素的开始标记与结束标记之间找到的任何子元素或内部文本都会分配给该对象的 XAML 内容属性值。允许为内容属性指定显式属性元素，但此用法通常不会出现在 .NET 参考的 XAML 语法部分中。显式/详细技术偶尔对使标记清晰或是在标记样式方面有价值，但内容属性的意图通常是简化标记，以便可以直接嵌套直观上以父子形式相关的元素。按照严格的 XAML 语言定义，元素上其他属性的属性元素标记不会分配为“内容”；它们以前按 XAML 分析器的处理顺序进行处理，不被视为“内容”。

## XAML 内容属性值必须是连续的

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。无论 XAML 内容属性的值是指定为字符串还是一个或多个对象，情况都是如此。例如，以下标记不会进行分析：

XAML

```
<Button>I am a  
<Button.Background>Blue</Button.Background>  
blue button</Button>
```

这实质上是非法的，因为如果是使用内容属性的属性元素语法将此语法设为显式，则内容属性会设置两次：

XAML

```
<Button>  
  <Button.Content>I am a </Button.Content>  
  <Button.Background>Blue</Button.Background>  
  <Button.Content> blue button</Button.Content>  
</Button>
```

一个类似的非法示例是，如果内容属性是集合，子元素与属性元素交织在一起：

XAML

```
<StackPanel>  
  <Button>This example</Button>  
  <StackPanel.Resources>  
    <SolidColorBrush x:Key="BlueBrush" Color="Blue"/>  
  </StackPanel.Resources>
```

```
<Button>... is illegal XAML</Button>
</StackPanel>
```

## 内容属性和集合语法组合

若要接受多个对象元素作为内容，内容属性的类型必须明确为集合类型。与集合类型的属性元素语法类似，XAML 处理器必须标识属于集合类型的类型。如果元素具有 XAML 内容属性，并且 XAML 内容属性的类型是集合，则隐式集合类型不需要在标记中指定为对象元素，并且 XAML 内容属性不需要指定为属性元素。因此，标记中的明显内容模型现在可以将多个子元素分配为内容。下面是 [Panel](#) 派生类的内容语法。所有 [Panel](#) 派生类都会建立 XAML 内容属性以作为 [Children](#)，这需要 [UIElementCollection](#) 类型的值。

XAML

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  >
  <StackPanel>
    <Button>Button 1</Button>
    <Button>Button 2</Button>
    <Button>Button 3</Button>
  </StackPanel>
</Page>
```

请注意，标记中既不需要 [Children](#) 的属性元素，也不需要 [UIElementCollection](#) 的元素。这是 XAML 的设计功能，以便递归包含的定义 UI 的元素可更直观地表示为具有直接父子元素关系的嵌套元素树，而无需插入属性元素标记或集合对象。事实上根据设计，[UIElementCollection](#) 不能在标记中显式指定为对象元素。因为它的唯一预期用途是作为隐式集合，[UIElementCollection](#) 不会公开公共无参数构造函数，因而无法实例化为对象元素。

## 将对象中的属性元素和对象元素与内容属性混合

XAML 规范声明，XAML 处理器可以强制用于填充对象元素内 XAML 内容属性的对象元素必须是连续的，不得混合。WPF XAML 处理器会强制实施这一针对混合属性元素和内容的限制。

可以将子对象元素作为对象元素中的第一个直接标记。随后可以引入属性元素。或者，可以指定一个或多个属性元素，随后指定内容，再指定更多属性元素。但是，一旦属性元素跟在内容后面，便不能引入任何进一步的内容，只能添加属性元素。

此内容/属性元素顺序要求不适用于用作内容的内部文本。但是，使内部文本保持连续仍然是一种很好的标记样式，因为如果属性元素与内部文本交织在一起，则难以在标记中直观地检测到有效空白。

## XAML 命名空间

前面的语法示例都未指定默认 XAML 命名空间以外的 XAML 命名空间。在典型 WPF 应用程序中，默认 XAML 命名空间指定为 WPF 命名空间。可以指定默认 XAML 命名空间以外的 XAML 命名空间，不过仍使用类似的语法。但是，只要命名了在默认 XAML 命名空间中不可访问的类，则该类名之前必须有映射到对应 CLR 命名空间的 XAML 命名空间的前缀。例如，`<custom:Example/>` 是用于实例化 `Example` 类的实例的对象元素语法，其中包含该类（并且可能有包含后备类型的外部程序集信息）的 CLR 命名空间以前映射到 `custom` 前缀。

有关 XAML 命名空间的详细信息，请参阅 [WPF XAML 的 XAML 命名空间和命名空间映射](#)。

## 标记扩展

XAML 定义了一个标记扩展编程实体，该实体可实现跳过字符串特性值或对象元素的正常 XAML 处理器处理，将处理延迟到后备类。在使用特性语法时向 XAML 处理器标识标记扩展的字符是左大括号 (`{`)，后跟右大括号 (`}`) 以外的任何字符。左大括号后面的第一个字符串必须引用提供特定扩展行为的类，其中的引用可能会在子字符串“Extension”是真实类名的一部分时省略该子字符串。此后可能会出现单个空格，随后扩展实现会将每个后续字符串用作输入，直到遇到右大括号为止。

.NET XAML 实现使用 `MarkupExtension` 抽象类作为 WPF 以及其他框架或技术支持的所有标记扩展的基础。WPF 专门实现的标记扩展通常旨在提供一种方法来引用其他现有对象，或是对将在运行时计算的对象进行延迟引用。例如，通过指定 `{Binding}` 标记扩展来代替特定属性通常采用的值，可完成简单的 WPF 数据绑定。许多 WPF 标记扩展为本来无法使用特性语法的属性实现了特性语法。例如，`Style` 对象是一种相对复杂的类型，其中包含一系列嵌套的对象和属性。WPF 中的样式通常定义为 `ResourceDictionary` 中的资源，然后通过请求资源的两个 WPF 标记扩展之一进行引用。标记扩展将属性值的计算延迟到资源查找，并且可以在特性语法中提供 `Style` 属性的值（采用类型 `Style`），如以下示例所示：

```
<Button Style="{StaticResource MyStyle}">My button</Button>
```

此处，`StaticResource` 标识提供标记扩展实现的 `StaticResourceExtension` 类。下一个字符串 `MyStyle` 用作非默认 `StaticResourceExtension` 构造函数的输入，其中从扩展字符串获取的参数会声明所请求的 `ResourceKey`。`MyStyle` 应为定义为资源的 `Style` 的 `x:Key`

值。 [StaticResource 标记扩展](#) 用法请求在加载时通过静态资源查找逻辑使用资源提供 [Style](#) 属性值。

有关标记扩展的详细信息，请参阅[标记扩展和 WPF XAML](#)。有关在常规 .NET XAML 实现中启用的标记扩展和其他 XAML 编程功能的参考，请参阅[XAML 命名空间 \(x:\) 语言功能](#)。有关特定于 WPF 的标记扩展，请参阅[WPF XAML 扩展](#)。

## 附加属性

附加属性是在 XAML 中引入的一个编程概念，通过该概念，属性可以由特定类型拥有和定义，但设置为任何元素上的特性或属性元素。附加属性的主要用途是使标记结构中的子元素可以向父元素报告信息，而不需要跨所有元素进行广泛共享的对象模型。反之，父元素可以使用附加属性向子元素报告信息。有关附加属性的用途以及如何创建自己的附加属性的详细信息，请参阅[附加属性概述](#)。

附加属性使用的语法表面上类似于属性元素语法，因为你也会指定 `typeName.propertyName` 组合。有两个重要的差异：

- 即使通过特性语法设置附加属性时，也可以使用 `typeName.propertyName` 组合。附加属性是特性语法中唯一要求限定属性名称的情况。
- 还可以对附加属性使用属性元素语法。但是，对于典型属性元素语法，指定的 `typeName` 是包含属性元素的对象元素。如果引用附加属性，则 `typeName` 是定义附加属性的类，而不是包含对象元素。

## 附加事件

附加事件是 XAML 中引入的另一个编程概念，其中事件可由特定类型定义，但处理程序可以附加到任何对象元素上。在 WOF 实现中，定义附加事件的类型通常是定义服务的静态类型，有时这些附加事件在公开服务的类型中通过路由事件别名公开。附加事件的处理程序通过特性语法进行指定。与附加事件一样，特性语法针对附加事件进行了扩展以允许使用 `typeName.eventName`，其中 `typeName` 是为附加事件基础结构提供 `Add` 和 `Remove` 事件处理程序访问器的类，而 `eventName` 是事件名称。

## XAML 根元素剖析

下表显示一个经过细分的典型 XAML 根元素，其中显示了根元素的特定特性：

Attribute	描述
<code>&lt;Page</code>	根元素的开始对象元素

Attribute	描述
<code>xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"</code>	默认 (WPF) XAML 命名空间
<code>xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"</code>	XAML 语言 XAML 命名空间
<code>x:Class="ExampleNamespace.ExampleCode"</code>	将标记连接到为分部类定义的任何代码隐藏的分部类声明
<code>&gt;</code>	根的对象元素结尾。 对象未结束，因为元素包含子元素

## 可选的非推荐 XAML 用法

以下各节介绍 XAML 处理器在技术上支持的 XAML 用法，但这些用法会产生冗长或其他美学问题，在开发包含 XAML 源的应用程序时会影响 XAML 文件保持可读性。

### 可选属性元素用法

可选属性元素用法包括显式写出 XAML 处理器视为隐式的元素内容属性。例如，当你声明 [Menu](#) 的内容时，可以选择将 [Menu](#) 的 [Items](#) 集合显式声明为 `<Menu.Items>` 属性元素标记，并将每个 [MenuItem](#) 放置在 `<Menu.Items>` 中，而不是使用隐式 XAML 处理器行为（[Menu](#) 中的所有子元素都必须是 [MenuItem](#) 并且放置在 [Items](#) 集合中）。有时，可选用法可帮助直观地阐明标记中表示的对象结构。或者，有时显式属性元素用法可以避免在技术上可正常运行但视觉上令人困惑的标记，例如特性值中的嵌套标记扩展。

### 完整 `typeName.memberName` 限定属性

特性的 `typeName.memberName` 形式实际上比路由事件情况更普遍。但在其他情况下，该形式是多余的，你应避免使用它（如果只是出于标记样式和可读性的原因）。在以下示例中，对 [Background](#) 特性的所有三个引用都完全等效：

#### XAML

```
<Button Background="Blue">Background</Button>
<Button Button.Background="Blue">Button.Background</Button>
<Button Control.Background="Blue">Control.Background</Button>
```

`Button.Background` 可正常工作，因为 `Button` 上对该属性的限定查找成功（`Background` 继承自 `Control`），并且 `Button` 是对象元素的类或基类。 `Control.Background` 可正常工作，因为 `Control` 类实际定义 `Background` 并且 `Control` 是 `Button` 基类。

但是，以下 `typeName.memberName` 形式示例不起作用，因而显示为已注释掉：

#### XAML

```
<!--<Button Label.Background="Blue">Does not work</Button> -->
```

`Label` 是 `Control` 的另一个派生类，如果在 `Label` 对象元素中指定了 `Label.Background`，则此用法将正常工作。但是，由于 `Label` 不是 `Button` 的类或基类，因此指定 XAML 处理器行为是随后将 `Label.Background` 作为附加属性进行处理。`Label.Background` 不是可用的附加属性，此用法会失败。

## baseTypeName.memberName 属性元素

与 `typeName.memberName` 形式适用于特性语法的方式类似，`baseTypeName.memberName` 语法适用于属性元素语法。例如，以下语法可正常工作：

#### XAML

```
<Button>Control.Background PE
  <Control.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="1,1">
      <GradientStop Color="Yellow" Offset="0.0" />
      <GradientStop Color="LimeGreen" Offset="1.0" />
    </LinearGradientBrush>
  </Control.Background>
</Button>
```

此处，属性元素作为 `Control.Background` 提供，即使属性元素包含在 `Button` 中。

但是正如 `typeName.memberName` 形式对于特性一样，`baseTypeName.memberName` 是标记中的糟糕样式，应避免使用。

## 请参阅

- [WPF 中的 XAML](#)
- [XAML 命名空间 \(x:\) 语言功能](#)
- [WPF XAML 扩展](#)
- [依赖项属性概述](#)
- [TypeConverters 和 XAML](#)

- XAML 及 WPF 的自定义类

# WPF 中的代码隐藏和 XAML

项目 • 2023/02/06

代码隐藏是一个术语，用于描述在对 XAML 页进行标记编译时与采用标记定义的对象联接的代码。本主题介绍代码隐藏的要求，以及 XAML 中代码的替代内联代码机制。

本主题包含以下各节：

- 必备条件
- 代码隐藏和 XAML 语言
- WPF 中的代码隐藏、事件处理程序和分部类要求
- `x:Code`
- 内联代码限制

## 先决条件

本主题假设你已阅读 [WPF 中的 XAML](#)，并且你对 CLR 和面向对象的编程方面有一些基础知识。

## 代码隐藏和 XAML 语言

XAML 语言包括语言级别功能，使代码文件能够与标记文件从标记文件端进行关联。具体而言，XAML 语言定义语言功能 [x:Class 指令](#)、[x:Subclass 指令](#) 和 [x:ClassModifier 指令](#)。XAML 语言不指定应如何生成代码以及如何集成标记和代码。而是由 WPF 等框架决定如何集成代码、如何在应用程序和编程模型中使用 XAML，以及生成操作或其他所有这一切所需的支持。

## WPF 中的代码隐藏、事件处理程序和分部类要求

- 分部类必须派生自支持根元素的类型。
- 请注意，在标记编译生成操作的默认行为下，可以在代码隐藏端的分部类定义中将派生留空。编译后的结果将假定页面根的后备类型是分部类的基础，即使未指定分部类也是如此。但是，依赖此行为并不是最佳做法。
- 在代码隐藏中编写的事件处理程序必须是实例方法，不能是静态方法。这些方法必须由 `x:Class` 标识的 CLR 命名空间内的分部类定义。不能通过限定事件处理程序

的名称来指示 XAML 处理器在不同的类范围内查找事件处理器以进行事件连接。

- 处理程序必须与后备类型系统中相应事件的委托匹配。
- 特别是对于 Microsoft Visual Basic 语言，你可以使用特定于语言的 `Handles` 关键字将处理程序与处理程序声明中的实例和事件相关联，而不是将处理程序与 WPF 事件系统中的属性附加在一起，例如某些路由事件场景或附加事件。有关详细信息，请参阅 [Visual Basic 和 WPF 事件处理](#)。

## x:Code

`x:Code` 是 WPF 架构中定义的指令元素，不会尝试将内容逐字解释为 XML。

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="MyNamespace.MyCanvasCodeInline"
>
    <Button Name="button1" Click="Clicked">Click Me!</Button>
    <x:Code><![CDATA[
        void Clicked(object sender, RoutedEventArgs e)
        {
            button1.Content = "Hello World";
        }
    ]]></x:Code>
</Page>
```

## 内联代码限制

应考虑避免或限制内联代码的使用。在体系结构和编码理念方面，保持标记和代码隐藏之间的分离可使设计器和开发人员角色更加鲜明。在更技术层面，可能很难写入为内联代码编写的代码，因为要始终写入 WPF 映射，包括 WPF 程序集中存在的大多数但并非所有 CLR 命名空间；必须完全限定对其他 CLR 命名空间中包含的类型和成员的调用。也不能在内联代码中定义分部类以外的任何内容，所引用的所有用户代码实体必须作为生成的分部类中的成员或变量存在。其他特定于语言的编程功能（例如宏或针对全局变量或生成变量的 `#ifdef`）也不可用。有关详细信息，请参阅 [x:Code 内部 XAML 类型](#)。

## 请参阅

- [WPF 中的 XAML](#)
- [x:Code 内部 XAML 类型](#)

- 生成 WPF 应用程序
- XAML 语法详述

# XAML 及 WPF 的自定义类

项目 • 2023/02/06

公共语言运行时 (CLR) 框架中实现的 XAML 支持定义任何公共语言运行时 (CLR) 语言的自定义类或结构，然后使用 XAML 标记访问类。通常通过将自定义类型映射到 XAML 命名空间前缀，可在同一标记文件中混合使用 Windows Presentation Foundation (WPF) 定义类型和自定义类型。本主题讨论将自定义类用作 XAML 元素必须满足的要求。

## 应用程序或程序集中的自定义类

XAML 中使用的自定义类可通过两种不同的方式进行定义：在代码隐藏或其他生成主 Windows Presentation Foundation (WPF) 应用程序的代码内，或者在单独程序集中作为类（例如用作类库的可执行文件或 DLL）。这些方法各有特定的优点和缺点。

- 创建类库的优点在于可在多个不同的应用程序间共享任意此类自定义类。通过使用单独的库，更易于控制应用程序的版本控制问题，并可简化类创建过程，在此过程中，所需的类用法是作为 XAML 页面上的根元素。
- 在应用程序中定义自定义类的优点在于此方法相对轻量，可减少在主应用程序可执行文件外引入单独程序集时遇到的部署和测试问题。
- 无论定义在相同还是不同的程序集中，自定义类若要在 XAML 中用作元素，都需要在 CLR 命名空间和 XML 命名空间之间进行映射。请参阅 [WPF XAML 的 XAML 命名空间和命名空间映射](#)。

## 将自定义类用作 XAML 元素的要求

为能够实例化为对象元素，类必须满足以下要求：

- 自定义类必须是公共的且支持默认（无参数）公共构造函数。（有关结构注释，请参阅下节内容。）
- 自定义类不得为嵌套类。嵌套类及其常规 CLR 使用语法中的“点”会干扰其他 WPF 和/或 XAML 功能（例如附加属性）。

除启用对象元素语法外，对象定义还会对任何其他将该对象作为值类型的公共属性启用属性元素语法。这是因为对象现在可被实例化为对象元素，且可填充此类属性的属性元素值。

## 结构

可始终在 WPF 的 XAML 中构造定义为自定义类型的结构。这是因为 CLR 编译器会对将所有属性值初始化为默认值的结构隐式创建无参数构造函数。某些情况下，结构并不需要默认构造行为和/或对象元素用法。这可能是因为结构需要通过概念方式将值和函数作为联合来填充，其中包含的值可能具有互斥的解释，因而其不存在任何可设置属性。此类结构的 WPF 示例为 [GridLength](#)。通常情况下，此类结构应实现类型转换器，以便可通过属性形式表达值，方法是使用创建结构值的不同解释或模式的字符串约定。结构还应通过非无参数构造函数对代码构造公开类似的行为。

## 将自定义类属性用作 XAML 特性的要求

属性必须引用按值类型（例如基元），或者使用特定类型的一个类（此特定类型需具有无参数构造函数或 XAML 处理器可访问的专用类型转换器）。在 CLR XAML 实现中，XAML 处理器通过语言基元的本机支持或通过将 [TypeConverterAttribute](#) 应用于后备类型定义中的类型或成员，查找此类转换器

或者，属性可引用抽象类类型或接口。对于抽象类或接口，XAML 分析的所需条件是必须用实现接口的实际类实例或派生自抽象类的类型实例填充属性值。

属性可在抽象类上声明，但仅可在派生自抽象类的实际类上设置。这是因为创建类的对象元素需要类上的公共无参数构造函数。

## 启用 TypeConverter 的特性语法

如果提供类级别的专用特性化类型转换器，则应用的类型转换会对需实例化该类型的任何属性启用特性语法。类型转换器不会启用类型的对象元素用法；仅在该类型存在无参数构造函数时启用对象元素用法。因此，启用类型转换器的属性通常不适用于属性语法，除非类型本身也支持对象元素语法。此规则存在一个例外，即可指定属性元素语法，但使属性元素包含一个字符串。此用法基本等效于特性语法用法，且此类用法并不常见，除非需要更可靠的特性值空白处理。例如，以下是一个采用字符串的属性元素用法以及一个特性用法等效项：

XAML

```
<Button>Hallo!
  <Button.Language>
    de-DE
  </Button.Language>
</Button>
```

XAML

```
<Button Language="de-DE">Hallo!</Button>
```

XAML 中允许特性语法但不允许包含对象元素的属性元素语法的属性示例是具有 [Cursor](#) 类型的各种属性。 [Cursor](#) 类具有专用类型转换器 [CursorConverter](#)，但不公开无参数构造函数，因此，即使实际 [Cursor](#) 类型为引用类型，[Cursor](#) 属性也仅可通过特性语法进行设置。

## 按属性类型转换器

或者，属性本身可能声明属性级别的类型转换器。通过基于相应类型将特性的传入字符串值处理为 [ConvertFrom](#) 操作的输出，这会启用可实例化具有内联属性类型的对象的“最小语言”。此操作的目的通常是提供方便的访问器，且这不是在 XAML 中启用属性设置的唯一方式。但是，如果要使用不提供无参数构造函数或特性化类型转换器的现有 CLR 类型，也可使用特性的类型转换器。WPF API 中的示例是具有 [CultureInfo](#) 类型的某些属性。这种情况下，WPF 使用现有 Microsoft .NET Framework [CultureInfo](#) 类型来更好地处理先前框架版本中使用的兼容性和迁移方案，但 [CultureInfo](#) 类型不支持将必需的构造函数或类型级别的类型转换器直接用作 XAML 属性值。

每当公开具有 XAML 用法的属性时，特别是对于控件作者，应特别考虑使用依赖属性支持此属性。如果使用 XAML 处理器的现有 Windows Presentation Foundation (WPF) 实现，更应该这样做，因为可通过 [DependencyProperty](#) 支持提高性能。依赖属性将对用户针对 XAML 可访问属性所需的属性公开属性系统功能。这包括动画、数据绑定和样式支持等功能。有关详细信息，请参阅[自定义依赖属性](#)和[XAML 加载和依赖属性](#)。

## 编写和特性化类型转换器

有时需要编写自定义 [TypeConverter](#) 派生类来对属性类型提供类型转换。有关如何派生和支持 XAML 用法的类型转换器以及如何应用 [TypeConverterAttribute](#) 的说明，请参阅 [TypeConverter 和 XAML](#)。

## 自定义类事件上 XAML 事件处理程序特性语法的要求

若要用作 CLR 事件，事件必须在支持无参数构造函数的类上或在派生类可访问事件的抽象类上公开为公共事件。为可方便地用作路由事件，CLR 事件应实现显式的 `add` 和 `remove` 方法，这些方法可对 CLR 事件签名添加和删除处理程序并将这些处理程序转发至 [AddHandler](#) 和 [RemoveHandler](#) 方法。这些方法添加或删除事件所附加到的实例上的路由事件处理程序存储的处理程序。

### ① 备注

可使用 `AddHandler` 直接对路由事件注册处理程序和特意不定义公开路由事件的 CLR 事件。通常不建议采用此操作，因为事件不会启用 XAML 特性语法用于附加处理程序，并且生成类提供的类型功能的 XAML 视图透明度较低。

## 编写集合属性

采用集合类型的属性所具有的 XAML 语法使你可指定添加到集合的对象。此语法具有两个重要功能。

- 无需在对象元素语法中指定作为集合对象的对象。无论何时在采用集合类型的 XAML 中指定属性，该集合类型的状态总是隐式。
- 标记中集合属性的子元素经处理后变成集合的成员。对集合成员的代码访问通常通过列表/字典方法（例如 `Add`）或通过索引器执行。但是，XAML 语法不支持方法和索引器（例外：XAML 2009 可支持这些方法，但使用 XAML 2009 会限制可能的 WPF 用法；请参阅 [XAML 2009 语言功能](#)）。对生成元素树而言，集合显然是非常常见的要求，并且你需要某种方法来填充声明 XAML 中的这些集合。因此，通过将集合属性的子元素添加到作为集合属性类型值的集合中来对其进行处理。

.NET Framework XAML 服务实现和 WPF XAML 处理器将以下定义用于组成集合属性的项。属性的属性类型必须实现以下项之一：

- 实现  [IList](#)。
- 实现  [IDictionary](#)。
- 派生自  [Array](#)（有关 XAML 中数组的详细信息，请参阅  [x:Array 标记扩展](#)。）
- 实现  [IAddChild](#)（由 WPF 定义的接口）。

CLR 中这些类型每个都具有 `Add` 方法，创建对象图时，XAML 处理器使用该方法将项添加到基础集合。

### ① 备注

WPF XAML 处理器的集合检测不支持通用 `List` 和 `Dictionary` 接口（`IList<T>` 和 `IDictionary< TKey, TValue >`）。但是，可将 `List<T>` 类用作基类，因为它直接实现 `IList`；或将 `Dictionary< TKey, TValue >` 用作基类，因为它直接实现 `IDictionary`。

声明采用集合的属性时，请注意类型的新实例中如何实例化此属性值。如果不将此属性实现为依赖属性，则使属性使用调用此集合类型构造函数的支持字段已可满足使用需求。如果属性为依赖属性，则可能需要将集合属性初始化为默认类型构造函数的一部分。这

是因为依赖属性从元数据获取其默认值，而通常不希望集合属性的初始值为静态共享集合。每个包含类型实例应具有一个集合实例。有关详细信息，请参阅[自定义依赖属性](#)。

可为集合属性实现自定义集合类型。由于隐式集合属性处理的原因，自定义集合类型无需提供无参数构造函数即可隐式用于 XAML 中。但是，可选择为集合类型提供无参数构造函数。此做法是有用的。除非提供无参数构造函数，否则无法显式将集合声明为对象元素。一些标记作者可能希望看到作为标记样式的显式集合。此外，创建将集合类型用作属性值的新对象时，无参数构造函数可简化初始化要求。

## 声明 XAML 内容属性

XAML 语言定义 XAML 内容属性的概念。对象语法中可用的每个类仅可具有一个 XAML 内容属性。若要对类将属性声明为 XAML 内容属性，请将 [ContentPropertyAttribute](#) 应用于类定义中。将所需 XAML 内容属性的名称指定为特性中的 [Name](#)。此属性由名称指定为一个字符串，而不是  [PropertyInfo](#) 等反射构造。

可将集合属性指定为 XAML 内容属性。这产生一种属性的用法，通过此用法，对象元素可具有一个或多个子元素，不干扰集合对象元素或属性元素标记。这些元素被视为 XAML 内容属性的值，并添加到支持集合实例中。

一些现有 XAML 内容属性使用 [Object](#) 的属性类型。这会启用采用 [String](#) 等基元值和单一引用对象值的 XAML 内容属性。如果按照此模型，类型负责类型确定以及处理可能的类型。[Object](#) 内容类型的典型目的是支持将对象内容添加为字符串（接收默认演示文稿处理）的简单方法和添加指定非默认演示文稿或其他数据的对象内容的高级方法。

## 序列化 XAML

某些情况下（例如对于控件作者），可能还需要确保任何可在 XAML 中实例化的对象演示文稿也可被序列化到等效的 XAML 标记。本主题中未介绍序列化要求。请参阅[控件创作概述](#)和[元素树和序列化](#)。

## 请参阅

- [WPF 中的 XAML](#)
- [自定义依赖属性](#)
- [控件创作概述](#)
- [基元素概述](#)
- [XAML 加载和依赖项属性](#)

# 标记扩展和 WPF XAML

项目 • 2023/02/06

本主题介绍 XAML 的标记扩展概念，包括其语法规则、用途以及作为其基础的类对象模型。标记扩展是 XAML 语言以及 XAML 服务的 .NET 实现的常规功能。本主题专门详细讨论用于 WPF XAML 的标记扩展。

## XAML 处理器和标记扩展

通常，XAML 分析器可将特性值解释为可转换成基元的文本字符串，或可通过某种方法将特性值转换为对象。其中一种方法是引用类型转换器；详情请参阅主题 [TypeConverters 和 XAML](#)。不过，也存在要求其他行为的情况。例如，可以指示 XAML 处理器，特性的值不应在对象图中生成新对象。相反，特性应生成引用对象图另一部分中的已构造对象或引用静态对象的对象图。另一种情况是，可以指示 XAML 处理器使用向对象的构造函数提供非默认参数的语法。在这些类型的情况下，标记扩展可以提供解决方案。

## 基本标记扩展语法

可以实现标记扩展来为特性用法中的属性和/或属性元素用法中的属性提供值。

当用于提供特性值时，将标记扩展序列与 XAML 处理器区分开的语法是左右大括号 ({ 和 })。然后，由紧跟在左大括号后面的字符串标记来标识标记扩展的类型。

当用在属性元素语法中时，标记扩展在外观上与任何其他用于提供属性元素值的元素相同，即：一个将标记扩展类作为元素引用并以尖括号 (<>) 括起的 XAML 元素声明。

## XAML 定义的标记扩展

存在这么几种标记扩展，它们并非特定于 XAML 的 WPF 实现，而是语言形式的 XAML 的内部函数或功能的实现。这些标记扩展在 System.Xaml 程序集中作为常规 .NET Framework XAML 服务的一部分而实现，并且位于 XAML 语言 XAML 命名空间内。就常见标记用法而言，这些标记扩展通常可由用法中的 `x:` 前缀标识。[MarkupExtension 基类](#)（也在 System.Xaml 中定义）提供了所有标记扩展均应使用的模式，以便在 XAML 读取器和 XAML 编写器中（包括在 WPF XAML 中）受到支持。

- `x:Type` 为命名类型提供 [Type](#) 对象。此扩展最常用于样式和模板。有关详细信息，请参阅 [x:Type 标记扩展](#)。
- `x:Static` 生成静态值。这些值来自于值类型代码实体，它们不直接是目标属性值的类型，但可以计算为该类型。有关详细信息，请参阅 [x:Static 标记扩展](#)。

- `x:Null` 将 `null` 指定为属性的值，可用于特性或属性元素值。有关详细信息，请参阅 [x:Null 标记扩展](#)。
- 在特意不使用 WPF 基元素和控件模型提供的集合支持的情况下，`x:Array` 为 XAML 语法中常规数组的创建提供支持。有关详细信息，请参阅 [x:Array 标记扩展](#)。

## ① 备注

`x:` 前缀在 XAML 文件或生成的根元素中用于 XAML 语言内部函数的典型 XAML 命名空间映射。例如，WPF 应用程序的 Visual Studio 模板使用此 `x:` 映射启动 XAML 文件。可以在自己的 XAML 命名空间映射中选择不同的前缀标记，但本文档将采用默认的 `x:` 映射，并通过它来标识属于 XAML 语言的 XAML 命名空间已定义部分的那些实体，这与 WPF 默认命名空间或与特定框架不相关的其他 XAML 命名空间相反。

# 特定于 WPF 的标记扩展

WPF 编程中最常用的标记扩展是支持资源引用的标记扩展（`StaticResource` 和 `DynamicResource`），和支持数据绑定的标记扩展（`Binding`）。

- `StaticResource` 通过替换已定义资源的值来为属性提供值。`StaticResource` 计算最终在 XAML 加载时进行，并且在运行时没有访问对象图的权限。有关详细信息，请访问 [StaticResource 标记扩展](#)。
- `DynamicResource` 通过将值推迟为对资源的运行时引用来为属性提供值。动态资源引用强制在每次访问此类资源时都进行新查找，且在运行时有权访问对象图。为了获取此访问权限，WPF 属性系统中的依赖项属性和计算出的表达式支持 `DynamicResource` 概念。因此，只能对依赖项属性目标使用 `DynamicResource`。有关详细信息，请参阅 [DynamicResource 标记扩展](#)。
- `Binding` 使用在运行时应用于父对象的数据上下文来为属性提供数据绑定值。此标记扩展相对复杂，因为它会启用大量内联语法来指定数据绑定。有关详细信息，请参阅 [Binding 标记扩展](#)。
- `RelativeSource` 为可以在运行时对象树中定位若干可能关系的 `Binding` 提供源信息。对于在多用途模板中创建的绑定，或在未充分了解周围的对象树的情况下以代码创建的绑定，此标记扩展为其提供专用源。有关详细信息，请参阅 [RelativeSource 标记扩展](#)。
- `TemplateBinding` 使控件模板能够使用模板化属性的值，这些属性来自于将使用该模板的类的对象模型定义属性。换言之，模板定义中的属性可访问仅在应用了模板之

后才存在的上下文。有关详细信息，请参阅 [TemplateBinding 标记扩展](#)。有关 [TemplateBinding](#) 的实际使用的详细信息，请参阅[使用 ControlTemplates 设置样式示例](#)。

- [ColorConvertedBitmap](#) 支持相对高级的映像方案。有关详细信息，请参阅 [ColorConvertedBitmap 标记扩展](#)。
- [ComponentResourceKey](#) 和 [ThemeDictionary](#) 支持资源查找的各个方面，特别是支持查找与自定义控件一起打包的资源和主题。有关详细信息，请参阅 [ComponentResourceKey 标记扩展、ThemeDictionary 标记扩展或控件创作概述](#)。

## \*Extension 类

对于常规 XAML 语言和特定于 WPF 的标记扩展，每个标记扩展的行为都会通过派生自 [MarkupExtension](#) 的 [\\*Extension](#) 类标识到 XAML 处理器，并提供 [ProvideValue](#) 方法的实现。每个扩展上的此方法都会提供在计算标记扩展时返回的对象。通常会基于传递给标记扩展的各个字符串标记来计算返回的对象。

例如，[StaticResourceExtension](#) 类提供实际资源查找的图面实现，以便其 [ProvideValue](#) 实现返回请求的对象，该特定实现的输入是用于按其 [x:Key](#) 查找资源的字符串。如果使用的是现有标记扩展，则其大部分实现详细信息都无关紧要。

有些标记扩展不使用字符串标记参数。这是因为它们返回静态值或一致的值，或者因为应返回何值的上下文可通过经  [serviceProvider](#) 参数传递的服务之一提供。

[\\*Extension](#) 命名模式的目的是为了实现方便和一致。XAML 处理器不必将该类标识为支持标记扩展。只要基本代码包含 [System.Xaml](#) 并使用 .NET Framework XAML 服务实现，所有需要识别为 XAML 标记扩展的内容就都会从 [MarkupExtension](#) 派生并将支持构造语法。WPF 可定义不遵循 [\\*Extension](#) 命名模式的标记扩展启用类，例如 [Binding](#)。原因通常是该类支持纯标记扩展支持以外的方案。对于 [Binding](#)，该类支持在与 XAML 无关的方案中对方法和属性的运行时访问权限。

## 初始化文本的扩展类解释

跟在标记扩展名称后且仍在括号内的字符串标记由 XAML 处理器通过以下方式之一进行解释：

- 逗号始终表示各个标记的分隔符。
- 如果各个分隔的标记不包含任何等号，则每个标记都将被视为构造函数参数。必须按该签名所期望的类型和该签名所期望的适当顺序给出每个构造函数参数。

## ① 备注

XAML 处理器必须调用与对的数量这一参数计数匹配的构造函数。出于此原因，如果要实现自定义标记扩展，请不要提供具有相同参数计数的多个构造函数。如果多个标记扩展构造函数路径具有相同的参数计数，则不会定义 XAML 处理器的行为方式，但应预计到如果这种情况存在于标记扩展类型定义中，则会允许 XAML 处理器引发有关使用情况的异常。

- 如果各个分隔的标记包含等号，则 XAML 处理器会首先为标记扩展调用无参数构造函数。之后，每个“名称=值”对会解释为标记扩展上存在的属性名称以及赋给该属性的值。
- 如果标记扩展中的构造函数行为与属性设置行为之间存在并行结果，则使用哪个行为都无关紧要。较常见的用法是将“属性=值”对用于具有多个可设置属性的标记扩展，因为这可使标记意图性更强，并减少意外转置构造函数参数的可能性。（指定 `property=value` 对时，这些属性可能按任意顺序排列。）也不能保证标记扩展提供一个构造函数参数来设置其每一个可设置的属性。例如，`Binding` 是一个标记扩展，具有多个可通过扩展以“属性=值”形式设置的属性，但 `Binding` 仅支持两个构造函数：无参数构造函数和一个设置初始路径的构造函数。
- 文本逗号在未转义的情况下无法传递给标记扩展。

## 转义序列和标记扩展

XAML 处理器中的特性处理使用大括号作为标记扩展序列的指示符。必要时，也可以使用后跟文本大括号的空大括号对输入转义序列，来生成文本大括号字符特性值。请参阅 [转义序列 - 标记扩展](#)。

## XAML 中的嵌套标记扩展用法

支持多个标记扩展的嵌套，并且将首先计算每个标记扩展的最里层。例如，考虑下面的用法：

XAML

```
<Setter Property="Background"
Value="{DynamicResource {x:Static SystemColors.ControlBrushKey}}" />
```

在此用法中，将首先计算 `x:Static` 语句并返回字符串。该字符串随后用作 `DynamicResource` 的参数。

# 标记扩展和属性元素语法

当用作填写属性元素值的对象元素时，标记扩展类在外观上与可用在 XAML 中的基于典型类型的对象元素没有区别。典型对象元素与标记扩展之间的实际差异是，标记扩展要么计算为类型化值，要么延迟为表达式。因此，标记扩展的属性值的任何可能类型错误的机制都将是不同的，这与在其他编程模型中处理后期绑定属性的方式类似。普通对象元素将针对分析 XAML 时它设置的目标属性进行类型匹配计算。

当用在对象元素语法中以填充属性元素时，大多数标记扩展都不会包含任何内容或深层属性元素语法。这样你便可以关闭对象元素标记，而不提供任何子元素。不论何时 XAML 处理器遇到任何对象元素，都会调用该类的构造函数来实例化从已分析元素创建的对象。标记扩展类也一样：如果希望标记扩展可在对象元素语法中使用，就必须提供无参数构造函数。有些现有标记扩展具有至少一个必需的属性值，必须指定该属性值才能使实例化生效。如果是这样，该属性值通常会作为对象元素上的属性特性而给出。在 [XAML 命名空间 \(x:\) 语言功能](#) 和 [WPF XAML 扩展](#) 参考页中，会指出具有必需属性的标记扩展（以及必需属性的名称）。参考页还将指出特定标记扩展是否禁止使用对象元素语法或特性语法。需要注意 [x:Array 标记扩展](#)，它无法支持特性语法，因为该数组的内容必须在标记内作为内容指定。数组内容的处理方式与常规对象一样，因此特性可以没有默认的类型转换器。此外，[x:Array 标记扩展](#) 需要 `type` 参数。

## 请参阅

- [WPF 中的 XAML](#)
- [XAML 命名空间 \(x:\) 语言功能](#)
- [WPF XAML 扩展](#)
- [StaticResource 标记扩展](#)
- [绑定标记扩展](#)
- [DynamicResource 标记扩展](#)
- [x>Type 标记扩展](#)

# WPF XAML 的 XAML 命名空间和命名空间映射

项目 · 2022/09/27

本主题进一步解释通常在 WPF XAML 文件的根标记中出现的两个 XAML 命名空间映射的存在性和用途。此外，还介绍如何生成相似映射以使用代码中和/或单独程序集中定义的元素。

## 什么是 XAML 命名空间

XAML 命名空间实际上是 XML 命名空间概念的扩展。指定 XAML 命名空间的方法依赖于 XML 命名空间语法、将 URI 用作命名空间标识符以及使用前缀提供从相同标记源引用多个命名空间等约定。XML 命名空间的 XAML 定义增添的主要概念是，XAML 命名空间表示标记用法唯一性范围，还影响标记实体可如何受特定 CLR 命名空间和引用程序集支持。后者也会受 XAML 架构上下文概念影响。但是出于 WPF 如何处理 XAML 命名空间的目的，对于默认 XAML 命名空间、XAML 语言命名空间以及任何其他由 XAML 标记直接映射到特定支持 CLR 命名空间和引用程序集的 XAML 命名空间，可通常考虑为 XAML 命名空间。

## WPF 和 XAML 命名空间声明

在许多 XAML 文件的根标记中的命名空间声明内，通常可看到两个 XML 命名空间声明。第一个声明默认映射整个 WPF 客户端/框架 XAML 命名空间：

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

第二个声明映射单独的 XAML 命名空间，（通常）将其映射到 `x:` 前缀。

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

这些声明之间的关系是 `x:` 前缀映射支持 XAML 语言定义中的内部函数，并且 WPF 是将 XAML 用作语言并为 XAML 定义对象词汇的一种实现。因为 WPF 词汇用法远比 XAML 内部函数用法常见，因此默认映射 WPF 词汇。

此 SDK 内，映射 XAML 语言内部函数支持的 `x:` 前缀约定后跟项目模板、示例代码和语言功能文档。XAML 命名空间定义许多常用功能，即使对于基本 WPF 应用程序而言，这些功能也是必需的。例如，若要通过分部类将任何代码隐藏加入到 XAML 文件，必须将该类命名为相关 XAML 文件根元素中的 `x:Class` 属性。或者，XAML 页面中定义的任何

要作为键控资源访问的元素都应在当前元素上设置 `x:Key` 属性。有关 XAML 的这些和其他方面的详细信息，请参阅 [WPF 中的 XAML](#) 或 [XAML 语法详细信息](#)。

## 映射到自定义类和程序集

在 `xmlns` 前缀声明内使用一系列标记可将 XML 命名空间映射到程序集，方法类似于将标准 WPF 和 XAML 内部函数 XAML 命名空间映射到前缀。

此语法采用以下可能的已命名标记和以下值：

`clr-namespace`: 在程序集中声明的 CLR 命名空间，此程序集包含要作为元素公开的公共类型。

`assembly=` 包含部分或全部引用 CLR 命名空间的程序集。此值通常为程序集的名称而不是路径，且不包含扩展名（例如 .dll 或 .exe）。程序集路径必须创建为包含要映射的 XAML 的项目文件中的项目引用。为添加版本控制和强名称签名，`assembly` 值可以是由 [AssemblyName](#) 定义的字符串，而不是简单的字符串名称。

请注意，分隔 `clr-namespace` 标记和其值的字符是冒号 (:)，而分隔 `assembly` 标记和其值的字符为等号 (=)。这两个标记之间应使用的字符是分号。此外，声明中任何位置不应含有空白。

## 基本自定义映射示例

如下代码定义一个示例自定义类：

```
C#  
  
namespace SDKSample {  
    public class ExampleClass : ContentControl {  
        public ExampleClass() {  
            ...  
        }  
    }  
}
```

此自定义类随后编译到库，库按项目设置（未显示）命名为 `SDKSampleLibrary`。

为引用此自定义类，还需将其添加为当前项目的引用（通常可使用 Visual Studio 中的解决方案资源管理器 UI 完成此操作）。

具有包含类的库并在项目设置中对其进行引用后，可将如下前缀映射到 XAML 中的根元素中：

```
xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"
```

综合而言，以下为根标记中包含自定义映射、典型默认值和 x: 映射的 XAML，然后使用前缀引用实例化此 UI 中的 ExampleClass：

XAML

```
<Page x:Class="WPFApplication1.MainPage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary">
  ...
  <custom:ExampleClass/>
  ...
</Page>
```

## 映射到当前程序集

如果要在与引用自定义类的应用程序代码相同的程序集中内定义引用的 `clr-namespace`，则可省略 `assembly`。或者，这种情况的等效语法是指定 `assembly=` 且等号后不含任何字符串标记。

如果在同一程序集中定义，则自定义类无法用作页面的根元素。分部类无需映射；仅需映射应用程序中不是页面分部类的类（若要将其引用为 XAML 中的元素）。

## 将 CLR 命名空间映射到程序集中的 XML 命名空间

WPF 定义 XAML 处理器使用的 CLR 属性，以便将多个 CLR 命名空间映射到单个 XAML 命名空间。属性 `XmlNsDefinitionAttribute` 放置于生成程序集的源代码中的程序集级别。WPF 程序集源代码使用此属性将多个常见命名空间（例如 `System.Windows` 和 `System.Windows.Controls`）映射到

`http://schemas.microsoft.com/winfx/2006/xaml/presentation` 命名空间。

`XmlNsDefinitionAttribute` 具有两个参数：XML/XAML 命名空间名称和 CLR 命名空间名称。可存在多个 `XmlNsDefinitionAttribute` 以将多个 CLR 命名空间映射到同一个 XML 命名空间。映射后，通过在分部类代码隐藏页中提供相应 `using` 语句，可在无完全限定的情况下引用这些命名空间的成员（如果需要）。如需了解详情，请访问 [XmlNsDefinitionAttribute](#)。

## 设计器命名空间和 XAML 模板中的其他前缀

如果使用 WPF XAML 的开发环境和/或设计工具，你可能会注意到 XAML 标记内存在其他定义的 XAML 命名空间/前缀。

适用于 Visual Studio 的 WPF 设计器使用通常映射到前缀 `d:` 的设计器命名空间。WPF 的较新项目模板可能会预映射此 XAML 命名空间，以支持适用于 Visual Studio 的 WPF 设计器和其他设计环境之间的交换。此设计 XAML 命名空间用于在设计器中往返基于 XAML 的 UI 时保持设计状态。它也用于 `d:IsDataSource`（在设计器中启用运行时数据源）等功能。

可能看到的另一个映射前缀是 `mc:`。`mc:` 用于标记兼容，使用一种并不一定特定于 XAML 的标记兼容模式。某种程度上，标记兼容功能可用于在框架之间或跨后备实现的其他边界交换 XAML、在 XAML 架构上下文之间运行、为设计器中限制模式提供兼容性等。有关标记兼容概念及其与 WPF 的关系的详细信息，请参阅[标记兼容 \(mc:\) 语言功能](#)。

## WPF 和程序集加载

WPF 的 XAML 架构上下文与 WPF 应用程序模型集成，此模型进而使用 [AppDomain](#) 的 CLR 定义概念。以下序列介绍根据 WPF 使用的 [AppDomain](#) 和其他因素，XAML 架构上下文如何解释如何在运行时或设计时加载程序集或查找类型。

1. 循环访问 [AppDomain](#)，从最近加载的程序集开始，查找匹配名称所有方面的已加载程序集。
2. 如果名称已限定，则对限定名称调用 [Assembly.Load\(String\)](#)。
3. 如果限定名称的短名称和公钥令牌匹配从中加载标记的程序集，则返回此程序集。
4. 使用短名称 + 公钥令牌调用 [Assembly.Load\(String\)](#)。
5. 如果名称未限定，请调用 [Assembly.LoadWithPartialName](#)。

宽松型 XAML 不使用步骤 3；不存在从中加载标记的程序集。

WPF 的已编译 XAML（通过 `XamlBuildTask` 生成）不使用从 [AppDomain](#) 中已加载的程序集（步骤 1）。此外，名称应不会从 `XamlBuildTask` 输出进行限定，因此步骤 5 不适用。

虽然 BAML 也不应包含非限定程序集名称，但是已编译 BAML（通过 `PresentationBuildTask` 生成）会使用所有步骤。

## 另请参阅

- [了解 XML 命名空间](#)

- WPF 中的 XAML

# WPF XAML 名称范围

项目 • 2023/02/06

XAML 名称范围是关于标识 XAML 中定义的对象的一个概念。XAML 名称范围中的名称可用于在对象树的对象 XAML 定义名称和其实例等效项之间建立关系。通常，在加载 XAML 应用程序的各个 XAML 页面根时，会在 WPF 托管代码中创建 XAML 名称范围。作为编程对象的 XAML 名称范围由 [INamespaceScope](#) 接口定义，且由实际类 [NameScope](#) 实现。

## 加载的 XAML 应用程序中的名称范围

从更广泛的编程或计算机科学来说，编程概念通常包括可用于访问对象的唯一标识符或名称的原则。对于使用标识符或名称的系统，名称范围会定义边界，在该边界中，进程或技术会搜索是否请求了具有该名称的对象或者是否执行了标识名称唯一性。这些一般原则适用于 XAML 名称范围。在 WPF 中，当 XAML 页面加载时，会在该页面的根元素上创建 XAML 名称范围。在 XAML 页面的页面根位置处指定的每个名称会添加到相关的 XAML 名称范围中。

在 WPF XAML 中，作为公共根元素的元素（例如 [Page](#) 和 [Window](#)）总是控制一个 XAML 名称范围。如果元素（如 [FrameworkElement](#) 或 [FrameworkContentElement](#)）是标记中页面的根元素，则 XAML 处理器会隐式添加 [Page](#) 根，以便 [Page](#) 可以提供一个有效的 XAML 名称范围。

### ① 备注

即使在 XAML 标记中的任何元素上未定义 `Name` 或 `x:Name` 属性，WPF 生成操作也会为 XAML 产品创建 XAML 名称范围。

如果尝试在任何 XAML 名称范围内两次使用相同的名称，将引发异常。对于具有代码隐藏且作为已编译应用程序的一部分的 WPF XAML，在初始标记编译期间为页面创建生成类时，WPF 生成操作在生成时会引发异常。对于不由任何生成操作进行标记编译的 XAML，加载 XAML 时可能会引发与 XAML 名称范围问题相关的异常。XAML 设计器在设计时也可能会出现 XAML 名称范围问题。

## 将对象添加到运行时对象树

分析 XAML 时即意味着创建并定义 WPF XAML 名称范围的时刻。如果在分析生成对象树的 XAML 之后的时间点将对象添加到对象树，新对象上的 `Name` 或 `x:Name` 值不会在 XAML 名称范围内自动更新信息。若要在加载 XAML 后将对象的名称添加到 WPF XAML

命名范围中，则必须在定义 XAML 名称范围的对象（通常为 XAML 页面根）上调用 [RegisterName](#) 的适当实现。如果未注册该名称，则不能通过 [FindName](#) 等方法使用名称引用已添加的对象，且无法将该名称用于动画定位。

对于应用程序开发人员来说最常见的方案是使用 [RegisterName](#) 将名称注册到页面当前根的 XAML 名称范围中。[RegisterName](#) 是情节提要（针对动画目标对象）的重要场景的一部分。有关详细信息，请参阅[情节提要概述](#)。

如果对并非定义 XAML 名称范围的对象调用 [RegisterName](#)，则该名称仍会注册到调用对象所在的 XAML 名称范围，就像在 XAML 名称范围内调用 [RegisterName](#) 定义对象一样。

## 代码中的 XAML 名称范围

可以通过代码创建然后使用 XAML 名称范围。创建 XAML 名称范围所涉及的 API 和概念都是相同的，甚至是使用纯代码，因为 WPF 的 XAML 处理器在处理 XAML 本身就会使用这些 API 和概念。这些概念和 API 存在的主要目的是为了能够在对象树中按名称查找对象（此对象树通常在 XAML 中完全或部分定义）。

对于以编程方式而不是通过加载的 XAML 创建的应用程序，定义 XAML 名称范围的对象必须实现 [INamescope](#)，或者为 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生类，以便支持在其实例上创建 XAML 名称范围。

此外，对于任何不由 XAML 处理器加载和处理的元素，默认情况下不会创建或初始化对象的 XAML 名称范围。必须为随后要将名称注册到其中的任何对象显式创建新的 XAML 名称范围。要创建 XAML 名称范围，调用 [SetNameScope](#) 静态方法。指定对象将其作为 `dependencyObject` 参数，并将新的 [NameScope](#) 构造函数作为 `value` 参数调用。

如果提供的对象作为 [SetNameScope](#) 的 `dependencyObject` 不是 [INamescope](#) 实现，[FrameworkElement](#) 或 [FrameworkContentElement](#)，在任何子元素上调用 [RegisterName](#) 都将不起作用。如果未能成功显式创建新的 XAML 名称范围，则对 [RegisterName](#) 的调用将引发异常。

有关以代码方式使用 XAML 名称范围 API 的示例，请参阅[定义名称范围](#)。

## 样式和模板中的 XAML 名称范围

通过 WPF 中的样式和模板，可以直接重复使用和重复应用内容。但是，样式和模板可能还包含具有模板级别定义的 XAML 名称的元素。此相同模板可在一页中多次使用。出于此原因，样式和模板皆定义其自身的 XAML 名称范围，与在应用样式或模板的对象树中的位置无关。

请考虑以下示例：

## XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
<Page.Resources>
    <ControlTemplate x:Key="MyButtonTemplate" TargetType="{x:Type Button}">
        <Border BorderBrush="Red" Name="TheBorder" BorderThickness="2">
            <ContentPresenter/>
        </Border>
    </ControlTemplate>
</Page.Resources>
<StackPanel>
    <Button Template="{StaticResource MyButtonTemplate}">My first
button</Button>
    <Button Template="{StaticResource MyButtonTemplate}">My second
button</Button>
</StackPanel>
</Page>
```

此处，同一模板应用到两个不同的按钮。如果模板不具有离散的 XAML 名称范围，则模板中使用的 `TheBorder` 名称会导致 XAML 名称范围中的名称冲突。模板的每个实例都具有其自己的 XAML 名称范围，因此在本例中，每个实例化模板的 XAML 名称范围仅包含一个名称。

样式也定义其自身的 XAML 名称范围，因此情节提要的各部分均可分配有特定的名称。即使在控件自定义过程中重新定义模板，这些名称也可实现控件特定行为，定位具有该名称的元素。

由于这些分开的 XAML 名称范围，在模板中查找命名元素比在页面中查找非模板命名元素难度更大。首先需通过获取控件（应用了模板）的 `Template` 属性值来确定已应用的模板。然后，调用 `FindName` 的模板版本，将应用了模板的控件作为第二个参数传递。

如果你是一位控件作者并且正在生成一种约定，其中已应用的模板中一个特定的命名元素是控件本身所定义的一种行为的目标，你可以使用你的控件实现代码中的 `GetTemplateChild` 方法。`GetTemplateChild` 方法是受保护的，所以只有控件作者能够访问它。

如果在模板内使用，并且需要进入应用了模板的 XAML 名称范围，则会获取 `TemplatedParent` 的值，随后在那里调用 `FindName`。从模板内入手的一个示例是编写事件将从已应用模板中的元素引发的事件处理程序实现。

## XAML 名称范围和与名称相关的 API

[FrameworkElement](#) 具有 [FindName](#)[RegisterName](#) 和 [UnregisterName](#) 方法。如果在其上调用这些方法的对象拥有 XAML 名称范围，则这些方法会调入相关 XAML 名称范围的方法。否则，将检查父元素以查看其是否拥有 XAML 名称范围，此过程以递归方式持续发生，直到找到 XAML 名称范围（由于 XAML 处理器行为，根处必定存在一个 XAML 名称范围）。[FrameworkContentElement](#) 具有类似的行为，例外是没有 [FrameworkContentElement](#) 曾拥有一个 XAML 名称范围。该方法存在于 [FrameworkContentElement](#)，因此最终可将调用转嫁给 [FrameworkElement](#) 父元素。

[SetNameScope](#) 用于将新的 XAML 名称范围映射到现有对象。可多次调用 [SetNameScope](#) 来重置或清除 XAML 名称范围，但这不是常见用法。此外，[GetNameScope](#) 通常不从代码中使用。

## XAML 名称范围实现

下面的类直接实现 [INamescope](#)：

- [NameScope](#)
- [Style](#)
- [ResourceDictionary](#)
- [FrameworkTemplate](#)

[ResourceDictionary](#) 不使用 XAML 名称或名称范围；相反，它使用键，因为它是一个字典实现。[ResourceDictionary](#) 实现 [INamescope](#) 的唯一原因是它可以引发用户代码异常，帮助区分真正的 XAML 名称范围与如何 [ResourceDictionary](#) 处理键之间的区别，并确保 XAML 名称范围不是由父元素应用到 [ResourceDictionary](#)。

[FrameworkTemplate](#) 和 [Style](#) 通过显式接口定义实现 [INamescope](#)。通过 [INamescope](#) 接口访问 XAML 名称范围时，显式实现可使这些 XAML 名称范围按惯例行为，这也是 XAML 名称范围通过 WPF 内部进程进行通信的方式。但显式接口定义不是 [FrameworkTemplate](#) 和 [Style](#) 常规 API 表面的一部分，因为很少需要在 [FrameworkTemplate](#) 和 [Style](#) 上直接调用 [INamescope](#) 方法，而改为使用其他 API，如 [GetTemplateChild](#)。

通过使用 [System.Windows.NameScope](#) 帮助程序类并通过 [NameScope.NameScope](#) 附加属性连接到其 XAML 名称范围实现，以下类定义了它们自己的 XAML 名称范围：

- [FrameworkElement](#)
- [FrameworkContentElement](#)

## 另请参阅

- [WPF XAML 的 XAML 命名空间和命名空间映射](#)
- [x:Name 指令](#)

# 内联样式和模板

项目 · 2023/02/06

采用 [Style](#) 和 [FrameworkTemplate](#) 类型的 XAML 几乎总是对现有样式和模板进行资源引用，而不是定义新的嵌入式样式和模板。

## 嵌入式样式和模板的限制

在 Extensible Application Markup Language (XAML) 中，样式和模板属性在技术上可通过两种方式之一进行设置。可以使用属性语法来引用已在资源中定义的样式，例如

`<object Style="{StaticResource myResourceKey}" .../>`。也可以使用属性元素语法来定义嵌入式样式，例如：

```
<object>  
  <object.Style>  
    <Style .../>  
  </object.Style>  
</object>
```

属性用法更为常见。定义为嵌入式且未在资源中定义的样式必须仅限于包含元素，由于没有资源键，因此不能轻易重用它。通常，资源定义的样式更为通用和有用，并且更符合一般的 Windows Presentation Foundation (WPF) 编程模型原则（即将代码中的程序逻辑与标记中的设计分离）。

通常没有理由设置嵌入式样式或模板，即使你只打算在该位置使用该样式或模板。大多数可以采用样式或模板的元素也支持内容属性和内容模型。如果仅使用通过样式或模板创建的任何逻辑树，那么仅用直接标记中的等效子元素填充该内容属性将会更容易。这样做将完全绕过样式和模板机制。

由返回对象的标记扩展启用的其他语法也可用于样式和模板。有两种可能的情况的扩展包括 [TemplateBinding](#) 和 [Binding](#)。

## 请参阅

- [样式设置和模板化](#)

# TypeConverters 和 XAML

项目 · 2023/02/06

本主题介绍将从字符串进行的类型转换作为常规 XAML 语言功能的用途。在.NET Framework 中，类 [TypeConverter](#) 作为托管自定义类实现的一部分，该类可用作 XAML 属性用法中的属性值的一部分。如果编写自定义类，并且希望类的实例可作为 XAML 可设置属性值使用，则可能需要向类应用、[TypeConverterAttribute](#) 编写自定义 [TypeConverter](#) 类或两者。

## 类型转换概念

### XAML 和字符串值

在 XAML 文件中设置特性值时，该值的初始类型是纯文本形式的字符串。即使是其他基元，如 [Double](#) 最初是 XAML 处理器的文本字符串。

XAML 处理器需要两条信息来处理特性值。第一条信息是所设置的属性的值类型。定义特性值以及在 XAML 中进行处理的任何字符串都必须最终转换或解析为该类型的值。如果值是 XAML 分析器可理解的基元（如数值），则会尝试直接转换字符串。如果值是枚举，则字符串用于检查是否存在与该枚举中的命名常量匹配的名称。如果值既不是分析器可理解的基元，也不是枚举，则所讨论的类型必须能够基于转换后的字符串提供类型的实例或值。可通过指示类型转换器类达到此目的。类型转换器实际上是提供其他类的值的帮助器类，可用于的 XAML 方案和 .NET 代码中的代码调用。

### 在 XAML 中使用现有的类型转换行为

你可能已经在基础应用程序 XAML 中使用了类型转换行为，只是你还不知道，具体取决于你对基础 XAML 概念的熟悉程度。例如，WPF 定义了数百个采用类型值 [Point](#) 的属性。A [Point](#) 是一个值，用于描述二维坐标空间中的坐标，并且它实际上只有两个重要属性：[X](#) 和 [Y](#)。在 XAML 中指定点时，将其指定为带分隔符的字符串，（通常以逗号）提供的值。[XY](#) 例如：`<LinearGradientBrush StartPoint="0,0" EndPoint="1,1"/>`。

即使在 XAML 中，这种简单类型 [Point](#) 及其简单用法也涉及类型转换器。在本例中，即类 [PointConverter](#)。

类级别定义的类型转换器 [Point](#) 简化了所有属性 [Point](#) 的标记用法。如果没有类型转换器，那么对于前面显示的同一示例，将需要更冗长的标记，如下所示：

XAML

```
<LinearGradientBrush>
  <LinearGradientBrush.StartPoint>
    <Point X="0" Y="0"/>
  </LinearGradientBrush.StartPoint>
  <LinearGradientBrush.EndPoint>
    <Point X="1" Y="1"/>
  </LinearGradientBrush.EndPoint>
</LinearGradientBrush>
```

使用类型转换字符串或使用更复杂的等效语法通常是编码风格的选择。 XAML 工具工作流也可能会影响值的设置方式。 某些 XAML 工具可能会生成最复杂的标记窗体，以便更容易往返于设计器视图或其自身的序列化机制。

现有类型转换器通常可以通过检查类 (或属性) 来发现 WPF 和.NET Framework 类型是否存在。 [TypeConverterAttribute](#) 此属性将对支持类型转换器转换类型的值的类进行命名，用于 XAML 或其他可能的用途。

## 类型转换器和标记扩展

标记扩展和类型转换器根据 XAML 处理器行为及其应用场景来实现正交角色。 尽管上下文可用于标记扩展用途，但通常不会在标记扩展实现中检查属性的类型转换行为（其中标记扩展提供了一个值）。 换言之，即使标记扩展返回一个文本字符串作为其 [ProvideValue](#) 输出，该字符串上应用于特定属性或属性值类型的类型转换行为也不会被调用。 通常，标记扩展的目的是在不调用任何类型转换器的情况下，处理字符串和返回对象。

需要标记扩展而不是类型转换器的一种常见情况是使对已存在的对象进行引用。 无状态类型转换器最多只能生成新实例，这可能并不理想。 若要深入了解标记扩展，请参阅[标记扩展和 WPF XAML](#)。

## 本机类型转换器

在 XAML 分析器的 WPF 和 .NET XAML 实现中，某些特定类型具有本机类型转换处理，但在传统上可能不会将这些类型视为基元。 这种类型的一个示例是 [DateTime](#)。 原因取决于.NET Framework 体系结构的工作原理：类型 [DateTime](#) 在 mscorelib 中定义，这是 .NET 中最基本的库。 [DateTime](#) 不允许使用来自另一个程序集（引入依赖项 (的程序集) 属性进行特性属性，该属性来自系统) [TypeConverterAttribute](#)，因此不支持通常的类型转换器发现机制。 相反，XAML 分析器具有需要此类本机处理的类型的列表，它可通过与真正基元的处理方式类似的方式来处理这些类型。（在这种情况下 [DateTime](#)，涉及到对 . 的 [Parse](#) 调用）

# 实现类型转换器

## TypeConverter

在前面给出的示例 [Point](#) 中，提到了该类 [PointConverter](#)。对于 XAML 的 .NET 实现，用于 XAML 目的所有类型转换器都是派生自基类的类 [TypeConverter](#)。类 [TypeConverter](#) 存在于 XAML 存在之前的 .NET Framework 版本中；其原始用法之一是为视觉设计器中的属性对话提供字符串转换。对于 XAML，将角色 [TypeConverter](#) 扩展为包含用于字符串和字符串转换的基类，用于分析字符串属性值，并可能将特定对象属性的运行时值处理回字符串中，以序列化为属性。

[TypeConverter](#) 定义四个成员，这些成员与用于 XAML 处理目的的字符串转换和从字符串相关：

- [CanConvertTo](#)
- [CanConvertFrom](#)
- [ConvertTo](#)
- [ConvertFrom](#)

其中，最重要的方法是 [ConvertFrom](#)。此方法将输入字符串转换为所需的对象类型。严格地说，[ConvertFrom](#) 可以实现此方法，将更广泛的类型转换为转换器的预期目标类型，从而用于超越 XAML 的目的，例如支持运行时转换，但出于 XAML 目的，它只是可以处理 [String](#) 重要输入的代码路径。

下一个最重要的方法是 [ConvertTo](#)。例如，如果应用程序转换为标记表示形式（，如果应用程序作为文件）保存到 XAML，[ConvertTo](#) 则负责生成标记表示形式。在这种情况下，对于 XAML 而言很重要的代码路径是传递一个 `destinationType String` 代码路径。

[CanConvertTo](#) 和 [CanConvertFrom](#) 是在服务查询 [TypeConverter](#) 实现的功能时使用的支持方法。必须实现这些方法以便为转换器的等效转换方法支持的特定于类型的情况返回 `true`。对于 XAML 用途，这通常意味着 [String](#) 类型。

## XAML 的区域性信息和类型转换器

每个 [TypeConverter](#) 实现都可以自行解释构成转换的有效字符串，还可以使用或忽略作为参数传递的类型说明。对于区域性和 XAML 类型转换，有一个重要的注意事项。XAML 完全支持使用可本地化的字符串作为特性值。但不支持将该可本地化字符串用作具有特定区域性要求的类型转换器输入，因为 XAML 特性值的类型转换器包含一个必要的固定语言分析行为，该行为使用 `en-US` 区域性。有关此限制的设计原因的详细信息，应参阅 XAML 语言规范 ([\[MS-XAML\]](#))。

区域性可能会产生问题的示例之一是，某些区域性使用逗号作为数字的小数点分隔符。这将与许多 WPF XAML 类型转换器所具有的使用逗号作为分隔符的行为发生冲突（根据常用的 X,Y 形式等过去的先例，或逗号分隔的列表）。即使在周围的 XAML 中传递区域性（将 `Language` 或 `xml:lang` 设置为 `s1-SI` 区域性，以此方式使用逗号作为小数点的区域性的一个示例）也不能解决此问题。

## 实现 ConvertFrom

若要能够用作支持 XAML 的 `TypeConverter` 实现，该转换器的 `ConvertFrom` 方法必须接受字符串作为 `value` 参数。如果字符串的格式有效，并且可由实现转换 `TypeConverter`，则返回的对象必须支持强制转换为属性所需的类型。否则，`ConvertFrom` 实现必须返回 `null`。

每个 `TypeConverter` 实现都可以自行解释构成转换的有效字符串，还可以使用或忽略作为参数传递的类型说明或区域性上下文。但是，WPF XAML 处理可能不会在所有情况下都将值传递给类型说明上下文，还可能不会基于 `xml:lang` 传递区域性。

### ① 备注

请勿使用大括号字符（尤其是 {}）作为字符串格式的可能元素。这些字符保留作为标记扩展序列的入口和出口。

## 实现 ConvertTo

`ConvertTo` 可能用于序列化支持。通过 `ConvertTo` 为自定义类型及其类型转换器实现的序列化支持不是绝对要求。但是，如果要实现控件，或使用序列化作为类的功能或设计的一部分，则应实现 `ConvertTo`。

若要用作 `TypeConverter` 支持 XAML 的实现，该 `ConvertTo` 转换器的方法必须接受类型（或作为参数支持的值）实例 `value`。`destinationType` 当参数是类型 `String` 时，返回的对象必须能够强制转换为 `String`。返回字符串必须表示 `value` 的序列化值。理想情况下，如果选择的序列化格式应该能够生成相同的值（如果该字符串传递到 `ConvertFrom` 同一转换器的实现），而不会丢失大量信息。

如果无法序列化该值，或者转换器不支持序列化，`ConvertTo` 则实现必须返回 `null`，并且在这种情况下允许引发异常。但是，如果确实引发异常，则应报告无法将该转换用作实现的 `CanConvertTo` 一部分，以便首先检查 `CanConvertTo` 并避免异常的最佳做法。

如果 `destinationType` 参数不是类型 `String`，则可以选择自己的转换器处理。通常，你将还原到基本实现处理，这在基本实现处理中引发 `ConvertTo` 特定异常。

## 实现 CanConvertTo

对于 [CanConvertTo](#) 类型的 `true` , `destinationType` 实现应返回 [String](#) , 否则遵从基实现。

## 实现 CanConvertFrom

对于 [CanConvertFrom](#) 类型的 `true` , `sourceType` 实现应返回 [String](#) , 否则遵从基实现。

## 应用 TypeConverterAttribute

为了使自定义类型转换器能够用作 XAML 处理器自定义类的代理类型转换器，必须应用于 [TypeConverterAttribute](#) 类定义。通过特性指定的 [ConverterTypeName](#) 必须是自定义类型转换器的类型名。应用此特性后，当 XAML 处理器处理属性类型使用自定义类类型的值时，它可以输入字符串并返回对象实例。

还可以基于每个属性提供类型转换器。(主定义，而不是对类定义应用一个 [TypeConverterAttribute](#) 属性定义，而不是 `get`/`set`) 它的实现。属性的类型必须与自定义类型转换器处理的类型匹配。应用此特性时，当 XAML 处理器处理该属性的值时，它可以处理输入字符串并返回对象实例。如果选择从 Microsoft .NET Framework 或使用某些其他库中的属性类型，则每个属性类型转换器技术特别有用，在该库中无法控制类定义，并且无法应用[TypeConverterAttribute](#)此类定义。

## 请参阅

- [TypeConverter](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)
- [XAML 语法详述](#)

# WPF XAML 扩展

项目 • 2023/02/06

## 本节内容

[绑定标记扩展](#)

[ColorConvertedBitmap 标记扩展](#)

[ComponentResourceKey 标记扩展](#)

[DynamicResource 标记扩展](#)

[RelativeSource MarkupExtension](#)

[StaticResource 标记扩展](#)

[TemplateBinding 标记扩展](#)

[ThemeDictionary 标记扩展](#)

[PropertyPath XAML 语法](#)

[PresentationOptions:Freeze 特性](#)

# 绑定标记扩展

项目 · 2023/02/06

将属性值延迟为数据绑定值，创建中间表达式对象并在运行时解释应用于元素及其绑定的数据上下文。

## 绑定表达式用法

XAML

```
<object property="{Binding}" .../>
-or-
<object property="{Binding bindProp1=value1[, bindPropN=valueN]*}" ...
/>
-or-
<object property="{Binding path}" .../>
-or
<object property="{Binding path[, bindPropN=valueN]*}" .../>
```

## 语法注释

在这些语法中，`[]` 和 `*` 不是文本。它们是表示法的一部分，表示可以使用零个或多个 `bindProp=value` 对，它们与前面的 `bindProp=value` 对之间用 `,` 分隔符分隔。

“可以使用绑定扩展设置的绑定属性”部分中列出的任何属性都可以使用 `Binding` 对象元素的特性来设置。但是，这并不是 `Binding` 真正的标记扩展用法，而只是对设置 CLR `Binding` 类属性的特性的常规 XAML 处理。换句话说，`<Binding bindProp1="value1"
[bindPropN="valueN"]*/*>` 是 `Binding` 对象元素用法（而不是 `Binding` 表达式用法）的特性的等效语法。若要了解 `Binding` 的特定属性的 XAML 特性用法，请参阅 .NET Framework 类库中 `Binding` 的相关属性的“XAML 特性用法”部分。

## XAML 值

值	描述
<code>bindProp1,</code> <code>bindPropN</code>	要设置的 <code>Binding</code> 或 <code>BindingBase</code> 属性的名称。并非所有 <code>Binding</code> 属性都可以使用 <code>Binding</code> 扩展来设置，某些属性只能通过使用进一步嵌套的标记扩展在 <code>Binding</code> 表达式中设置。请参阅“可以使用绑定扩展设置的绑定属性”部分。
<code>value1,</code> <code>valueN</code>	要将属性设置为的值。特性值的处理最终特定于所设置的特定 <code>Binding</code> 属性的类型和逻辑。

值	描述
path	用于设置隐式 <code>Binding.Path</code> 属性的路径字符串。另请参阅 <a href="#">PropertyPath XAML 语法</a> 。

## 未限定的 {Binding}

“绑定表达式用法”中所示的 `{Binding}` 用法创建了一个具有默认值的 `Binding` 对象，其中包括 `Binding.Path` 的初始值 `null`。该用法在许多情况下仍然有用，因为创建的 `Binding` 可能依赖于在运行时数据上下文中设置的关键数据绑定属性（如 `Binding.Path` 和 `Binding.Source`）。有关数据上下文概念的详细信息，请参阅[数据绑定](#)。

## 隐式路径

`Binding` 标记扩展使用 `Binding.Path` 作为概念上的“默认属性”，其中 `Path=` 不需要出现在表达式中。如果指定具有隐式路径的 `Binding` 表达式，则隐式路径必须首先出现在表达式中，即，在 `Binding` 属性由名称指定的任何其他 `bindProp = value` 对之前。例如：

`{Binding PathString}`，其中 `PathString` 是一个字符串，其计算结果为标记扩展用法创建的 `Binding` 中的 `Binding.Path` 值。你可以在逗号分隔符后追加包含其他命名属性的隐式路径，例如 `{Binding LastName, Mode=TwoWay}`。

## 可以使用绑定扩展设置的绑定属性

本主题中显示的语法使用泛型 `bindProp = value` 近似值，因为 `BindingBase` 或 `Binding` 的许多读/写属性可以通过 `Binding` 标记扩展/表达式语法进行设置。它们可以按任意顺序设置，隐式 `Binding.Path` 除外。（你可以选择显式指定 `Path=`，在这种情况下，它可以按任意顺序设置）。基本上，你可以使用以逗号分隔的 `bindProp = value` 对来设置以下列表中的零个或多个属性。

其中一些属性值所需的对象类型不支持从 XAML 中的文本语法进行本机类型转换，因此需要标记扩展才能设置为特性值。有关详细信息，请查看每个属性的 .NET Framework 类库中的“XAML 特性用法”部分；用于 XAML 特性语法的字符串（无论是否有进一步的标记扩展用法）与在 `Binding` 表达式中指定的值基本相同，但不会将 `Binding` 表达式中的每个 `bindProp = value` 括在引号中。

- `BindingGroupName`：标识可能的绑定组的字符串。这是一个比较高级的绑定概念；请参阅 [BindingGroupName](#) 的参考页。
- `BindsDirectlyToSource`：布尔值，可以是 `true` 或 `false`。默认为 `false`。

- [Converter](#)：可以在表达式中设置为 `bindProp = value` 字符串，但这样做需要值的对象引用，例如 [StaticResource 标记扩展](#)。本例中的值是自定义转换器类的实例。
- [ConverterCulture](#)：可在表达式中设置为基于标准的标识符；请参阅 [ConverterCulture 的参考主题](#)。
- [ConverterParameter](#)：可以在表达式中设置为 `bindProp = value` 字符串，但这取决于所传递的参数的类型。如果传递值的引用类型，则此用法需要对象引用，例如嵌套的 [StaticResource 标记扩展](#)。
- [ElementName](#)：与 [RelativeSource](#) 和 [Source](#) 互斥；这些绑定属性中的每一个均表示一种特定的绑定方法。请参阅[数据绑定概述](#)。
- [FallbackValue](#)：可以在表达式中设置为 `bindProp = value` 字符串，但这取决于所传递的值的类型。如果传递引用类型，则需要对象引用，例如嵌套的 [StaticResource 标记扩展](#)。
- [IsAsync](#)：布尔值，可以是 `true` 或 `false`。默认为 `false`。
- [Mode](#)：值为 [BindingMode](#) 枚举中的常量名称。例如 `{Binding Mode=OneWay}`。
- [NotifyOnSourceUpdated](#)：布尔值，可以是 `true` 或 `false`。默认为 `false`。
- [NotifyOnTargetUpdated](#)：布尔值，可以是 `true` 或 `false`。默认为 `false`。
- [NotifyOnValidationError](#)：布尔值，可以是 `true` 或 `false`。默认为 `false`。
- [Path](#)：描述数据对象或通用对象模型的路径的字符串。该格式提供几种不同的约定来遍历对象模型，这些约定在本主题中无法充分描述。请参阅 [PropertyPath XAML 语法](#)。
- [RelativeSource](#)：与 [ElementName](#) 和 [Source](#) 互斥；这些绑定属性中的每一个均表示一种特定的绑定方法。请参阅[数据绑定概述](#)。需要嵌套的 [RelativeSource MarkupExtension](#) 用法来指定值。
- [Source](#)：与 [RelativeSource](#) 和 [ElementName](#) 互斥；这些绑定属性中的每一个均表示一种特定的绑定方法。请参阅[数据绑定概述](#)。需要嵌套的扩展用法，通常是 [StaticResource 标记扩展](#)，该扩展引用来自键控资源字典的对象数据源。
- [StringFormat](#)：描述绑定数据的字符串格式约定的字符串。这是一个比较高级的绑定概念；请参阅 [StringFormat 的参考页](#)。
- [TargetNullValue](#)：可以在表达式中设置为 `bindProp = value` 字符串，但这取决于所传递的参数的类型。如果传递值的引用类型，则需要对象引用，例如嵌套的 [StaticResource 标记扩展](#)。

- `UpdateSourceTrigger`：值为 `UpdateSourceTrigger` 枚举中的常量名称。例如 `{Binding UpdateSourceTrigger=LostFocus}`。特定控件可能对此绑定属性使用不同的默认值。请参阅 [UpdateSourceTrigger](#)。
- `ValidatesOnDataErrors`：布尔值，可以是 `true` 或 `false`。默认为 `false`。请参阅“备注”。
- `ValidatesOnExceptions`：布尔值，可以是 `true` 或 `false`。默认为 `false`。请参阅“备注”。
- `XPath`：描述 XML 数据源的 XMLDOM 路径的字符串。请参阅[使用 XMLDataProvider 和 XPath 查询绑定到 XML 数据](#)。

以下是无法使用 `Binding` 标记扩展/`{Binding}` 表达式格式设置的 `Binding` 属性。

- `UpdateSourceExceptionFilter`：此属性需要引用回调实现。不能在 XAML 语法中引用事件处理程序以外的回调/方法。
- `ValidationRules`：此属性使用 `ValidationRule` 对象的泛型集合。这可以表示为 `Binding` 对象元素中的属性元素，但在 `Binding` 表达式中没有现成的特性分析方法可以使用。请参阅 [ValidationRules](#) 的参考主题。
- `XmlNamespaceManager`

## 注解

### ① 重要

在依赖属性优先级方面，`Binding` 表达式等效于本地设置的值。如果为先前使用 `Binding` 表达式的属性设置本地值，则会完全删除 `Binding`。有关详细信息，请参阅[依赖属性值优先级](#)。

本主题不包括从基本层面描述数据绑定。请参阅[数据绑定概述](#)。

### ① 备注

`MultiBinding` 和 `PriorityBinding` 不支持 XAML 扩展语法。请改用属性元素。请参阅 [MultiBinding 和 PriorityBinding](#) 的参考主题。

XAML 的布尔值不区分大小写。例如，可指定 `{Binding NotifyOnValidationError=true}` 或 `{Binding NotifyOnValidationError=True}`。

涉及数据验证的绑定通常由显式 `Binding` 元素指定，而不是指定为 `{Binding ...}` 表达式，而且在表达式中设置 `ValidatesOnDataErrors` 或 `ValidatesOnExceptions` 的做法并不常见。这是因为无法使用表达式格式轻松设置配套属性 `ValidationRules`。有关详细信息，请参阅[实现绑定验证](#)。

`Binding` 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于在某些类型或属性上特性化的类型转换器而言，此要求更具有全局性。XAML 中的所有标记扩展在其特性语法中均使用 `{` 和 `}` 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理字符串内容。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

`Binding` 是一种非典型标记扩展，因为为 WPF 的 XAML 实现实现扩展功能的 `Binding` 类还实现了与 XAML 无关的其他几个方法和属性。其他成员旨在使 `Binding` 成为一个更通用且独立的类，除了用作 XAML 标记扩展外，还能解决许多数据绑定的情况。

## 另请参阅

- [Binding](#)
- [数据绑定概述](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)

# ColorConvertedBitmap 标记扩展

项目 • 2023/02/06

提供方法来指定没有嵌入配置文件的位图源。 颜色上下文/配置文件由 URI 指定，与图像源 URI 一样。

## XAML 属性用法

XML

```
<object property="{ColorConvertedBitmap imageSource sourceIIC  
destinationIIC}" ... />
```

## XAML 值

值	描述
imageSource	无配置文件位图的 URI。
sourceIIC	源配置文件配置的 URI。
destinationIIC	目标配置文件配置的 URI

## 注解

此标记扩展旨在填充一组相关的图像源属性值，如 [UriSource](#)。

特性语法是最常用于该标记扩展的语法。 `ColorConvertedBitmap` 或 `ColorConvertedBitmapExtension` 不能用于属性元素语法，因为值只能设置为初始构造函数上的值，初始构造函数是扩展标识符后面的字符串。

`ColorConvertedBitmap` 是标记扩展。 当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。 XAML 中的所有标记扩展在其特性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理特性。 有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 另请参阅

- [UriSource](#)

- 标记扩展和 WPF XAML
- 图像处理概述

# ComponentResourceKey 标记扩展

项目 • 2023/02/06

定义和引用从外部程序集加载的资源的键。这使资源查找能够在程序集中指定目标类型，而不是在程序集中或类上指定显式资源字典。

## XAML 属性用法（设置键，精简）

XML

```
<object x:Key="{ComponentResourceKey {x>Type targetTypeName}, targetID}" ... />
```

## XAML 属性用法（设置键，详细）

XML

```
<object x:Key="{ComponentResourceKey TypeInTargetAssembly={x>Type targetTypeName}, ResourceID=targetID}" ... />
```

## XAML 属性用法（请求资源，精简）

XML

```
<object property="{DynamicResource {ComponentResourceKey {x>Type targetTypeName}, targetID}}" ... />
```

## XAML 属性用法（请求资源，详细）

XML

```
<object property="{DynamicResource {ComponentResourceKey TypeInTargetAssembly={x>Type targetTypeName}, ResourceID=targetID}}" ... />
```

## XAML 值

值	描述
---	----

值	描述
<code>targetTypeName</code>	在资源程序集中定义的公共语言运行时 (CLR) 类型的名称。
<code>targetID</code>	资源的密钥。 查找资源时， <code>targetID</code> 将类似于资源的 <code>x:Key</code> 指令。

## 注解

如上面的用法所示，在两个位置可以找到 `{ComponentResourceKey}` 标记扩展用法：

- 由控件作者提供的主题资源字典中的键的定义。
- 重新模板化控件但想要使用来自控件主题提供的资源的属性值时，可从程序集中访问主题资源。

对于来自主题的引用组件资源，通常建议使用 `{DynamicResource}` 而不是 `{StaticResource}`。用法中显示了这一点。推荐使用 `{DynamicResource}`，因为用户可以更改主题本身。如果希望组件资源与控件作者支持主题的意图最为匹配，则应使组件资源引用也成为动态的。

`TypeInTargetAssembly` 标识存在于实际定义资源的目标程序集中的类型。可以独立地定义和使用 `ComponentResourceKey`，而无需知道 `TypeInTargetAssembly` 的确切定义位置，但最终必须通过引用的程序集解析类型。

`ComponentResourceKey` 的一个常见用法是定义键，然后将它们作为类的成员公开。对于这种用法，请使用 `ComponentResourceKey` 类构造函数，而不是标记扩展。有关详细信息，请参阅 `ComponentResourceKey` 或主题控件创作概述的“定义和引用主题资源的键”部分。

对于建立键和引用键控资源，属性语法通常用于 `ComponentResourceKey` 标记扩展。

显示的简明语法依赖于标记扩展的 `ComponentResourceKey` 构造函数签名和位置参数用法。`targetTypeName` 和 `targetID` 的给出顺序很重要。详细语法依赖于 `ComponentResourceKey` 无参数构造函数，然后以类似于对象元素上的真正属性语法的方式设置 `TypeInTargetAssembly` 和 `ResourceId`。在详细语法中，设置属性的顺序并不重要。标记扩展和 WPF XAML 主题中更详细地描述了这两种替代方案（简明和详细）的关系和机制。

从技术上讲，`targetID` 的值可以是任何对象，不用非得是字符串。但是，WPF 中最常见的用法是将 `targetID` 值与字符串形式对齐，并且此类字符串在 `XamlName` 语法中有有效。

`ComponentResourceKey` 可以在对象元素语法中使用。在这种情况下，需要同时指定 `TypeInTargetAssembly` 和 `ResourceId` 属性的值才能正确初始化扩展。

在 WPF XAML 读取器实现中，对此标记扩展的处理由 `ComponentResourceKey` 类定义。

`ComponentResourceKey` 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。XAML 中的所有标记扩展在其特性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理特性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 另请参阅

- [ComponentResourceKey](#)
- [ControlTemplate](#)
- [控件创作概述](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)

# DateTime XAML 语法

项目 · 2023/02/06

某些控件（例如 [Calendar](#) 和 [DatePicker](#)）具有使用 [DateTime](#) 类型的属性。虽然通常会在运行时在代码隐藏中指定这些控件的初始日期或时间，但可以在 XAML 中指定初始日期或时间。WPF XAML 分析器使用内置的 XAML 文本语法对 [DateTime](#) 值进行分析。本主题介绍 [DateTime](#) XAML 文本语法的具体内容。

## DateTime XAML 语法的使用场合

无需经常对 XAML 中的日期进行设置，而且设置也可能并不合适。例如，可以使用 [DateTime.Now](#) 属性在运行时初始化一个日期，或者可以基于用户输入在代码隐藏中为日历的所有日期进行调整。但是，有些场景下，可能要硬编码日期到控件模板的 [Calendar](#) 和 [DatePicker](#) 中。这些场景必须使用 [DateTime](#) XAML 语法。

## DateTime XAML 语法是本机行为

[DateTime](#) 是在 CLR 基类库中定义的类。由于基类库与 CLR 其余部分的关联方式的原因，不可能将 [TypeConverterAttribute](#) 应用于该类，也不可能在运行时对象模型中使用类型转换器处理 XAML 中的字符串并将它们转换为 [DateTime](#)。没有提供转换行为的 [DateTimeConverter](#) 类；本主题中描述的转换行为源于 WPF XAML 分析器。

## DateTime XAML 语法的格式字符串

可通过格式字符串指定 [DateTime](#) 的格式。格式字符串会规范可用于创建值的文本语法。现有 WPF 控件的 [DateTime](#) 值通常只使用 [DateTime](#) 的日期部分，而不使用时间部分。

当指定 XAML 中的 [DateTime](#) 时，可以交替使用任何格式字符串。

还可以使用未在本主题中专门显示的格式和格式字符串。从技术上讲，指定并由 WPF XAML 分析器分析的任何 [DateTime](#) 值的 XAML 使用对 [DateTime.Parse](#) 的内部调用，因此可以将 [DateTime.Parse](#) 接受的任何字符串用于 XAML 输入。有关详细信息，请参阅 [DateTime.Parse](#)。

### ① 重要

DateTime XAML 语法在其本机转换中始终将 `en-us` 用作 [CultureInfo](#)。这不会受 XAML 中 [Language](#) 值或 `xml:lang` 值的影响，因为 XAML 属性级别的类型转换可在

没有该上下文的情况下执行。出于区域性差异的原因，请不要尝试插入此处所示的格式字符串，例如，日期和月份的显示顺序。此处所示的格式字符串是在分析 XAML 时（不考虑其他区域性设置）使用的确切格式字符串。

以下部分介绍了一些常见的 [DateTime](#) 格式字符串。

## 短日期模式（“d”）

下面显示 XAML 中 [DateTime](#) 的短日期格式：

```
M/d/YYYY
```

这是最简单的形式，可指定 WPF 控件一般使用的所有必要信息，并且不受针对时间部分的偶然时区偏移的影响，因此比其他格式更适用。

例如，若要指定日期 2010 年 6 月 1 日，可使用以下字符串：

```
3/1/2010
```

有关详细信息，请参阅 [DateTimeFormatInfo.ShortDatePattern](#)。

## 可排序 DateTime 模式（“s”）

下面显示 XAML 中可排序的 [DateTime](#) 模式：

```
yyyy'-'MM'-'dd'T'HH':'mm':'ss
```

例如，若要指定日期 2010 年 6 月 1 日，可使用以下字符串（时间部分全部输入为 0）：

```
2010-06-01T00:00:00
```

## RFC1123 模式（“r”）

RFC1123 模式很有用，因为它可以是来自其他日期生成器的字符串输入，这些字符生成器出于区域性固定原因同样使用 RFC1123 模式。下面显示 XAML 中的 RFC1123 [DateTime](#) 模式：

```
ddd, dd MMM yyyy HH':'mm':'ss 'UTC'
```

例如，若要指定日期 2010 年 6 月 1 日，可使用以下字符串（时间部分全部输入为 0）：

```
Mon, 01 Jun 2010 00:00:00 UTC
```

## 其他格式和模式

如上所述，XAML 中的 [DateTime](#) 可以指定为任何可接受作为 [DateTime.Parse](#) 输入的字符串。这包括其他正式化格式（例如 [UniversalSortableDateTimePattern](#)）以及未正式化为特定 [DateTimeFormatInfo](#) 形式的格式。例如，窗体 `YYYY/mm/dd` 可以作为 [DateTime.Parse](#) 的输入。本主题并没有试图介绍所有可能有效的格式，而是推荐将短日期模式作为一种标准做法。

## 请参阅

- [WPF 中的 XAML](#)

# DynamicResource 标记扩展

项目 • 2023/02/06

通过推迟一个值作为对已定义资源的引用，为任何 XAML 属性提供该值。该资源的查找行为类似于运行时查找。

## XAML 属性用法

XML

```
<object property="{DynamicResource key}" ... />
```

## XAML 属性元素用法

XML

```
<object>
  <object.property>
    <DynamicResource ResourceKey="key" ... />
  </object.property>
</object>
```

## XAML 值

值	描述
key	所请求资源的密钥。如果资源在标记中创建，则此键最初由 <a href="#">x:Key 指令</a> 分配，如果资源在代码中创建，则此键在调用 <a href="#">ResourceDictionary.Add</a> 时作为 key 参数提供。

## 注解

DynamicResource 将在初始编译期间创建一个临时表达式，从而延迟查找资源，直到真正需要请求的资源值来构造对象。这可能是在加载 XAML 页面之后。将基于以从当前页面范围开始的所有活动资源字典为范围的键搜索查找资源值，并替换编译中的占位符表达式。

① 重要

就依赖属性优先级而言，`DynamicResource` 表达式等效于应用动态资源引用的位置。如果为先前使用 `DynamicResource` 表达式作为本地值的属性设置本地值，则会完全删除 `DynamicResource`。有关详细信息，请参阅[依赖属性值优先级](#)。

某些资源访问方案特别适合于 `DynamicResource`，这与 `StaticResource` 标记扩展相反。有关 `DynamicResource` 和 `StaticResource` 的相对优点和性能影响的讨论，请参阅[XAML 资源](#)。

指定的 `ResourceKey` 在某种程度上应对应于由 `x:Key` 指令在你的页面、应用程序、可用控件主题和外部资源或系统资源中确定的现有资源，并且资源查找将按该顺序进行。有关静态和动态资源查找的详细信息，请参阅[XAML 资源](#)。

资源键可以是以 `XamlName` 语法定义的任何字符串。资源键也可能是其他对象类型，例如 `Type`。`Type` 键是如何按主题设置控件样式的基础。有关详细信息，请参阅[控件创作概述](#)。

查找资源值的 API（例如 `FindResource`）遵循与 `DynamicResource` 所用相同的资源查找逻辑。

引用资源的替代声明性方法与 `StaticResource` 标记扩展一样。

特性语法是最常用于该标记扩展的语法。在 `DynamicResource` 标识符字符串之后提供的字符串标记被指定为基础 `ResourceKey` 扩展类的 `DynamicResourceExtension` 值。

`DynamicResource` 可以在对象元素语法中使用。在这种情况下，需要指定 `ResourceKey` 属性的值。

`DynamicResource` 还可以在详细特性用法中使用，以便将 `ResourceKey` 属性指定为一个 `property=value` 对：

XML

```
<object property="{DynamicResource ResourceKey=key}" ... />
```

如果扩展具有一个以上的可设置属性，或者某些属性是可选的，则详细用法通常会很有用。由于 `DynamicResource` 仅有一个可设置的属性，并且此属性是必需的，因此该详细用法不具有典型性。

在 WPF XAML 处理器实现中，对此标记扩展的处理由 `DynamicResourceExtension` 类来定义。

`DynamicResource` 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全

属性。 XAML 中的所有标记扩展在其特性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理特性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 另请参阅

- [XAML 资源](#)
- [资源和代码](#)
- [x:Key 指令](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)
- [StaticResource 标记扩展](#)
- [标记扩展和 WPF XAML](#)

# RelativeSource MarkupExtension

项目 • 2022/09/27

指定一个 `RelativeSource` 绑定源的属性，以便在 [绑定标记扩展](#) 中使用，或在设置 XAML 中创建的 `Binding` 元素的 `RelativeSource` 属性时使用。

## XAML 属性用法

XML

```
<Binding RelativeSource="{RelativeSource modeEnumValue}" ... />
```

## XAML 属性用法（嵌套在 Binding 扩展内）

XML

```
<object property="{Binding RelativeSource={RelativeSource modeEnumValue} ... }" ... />
```

## XAML 对象元素用法

XML

```
<Binding>
  <Binding.RelativeSource>
    <RelativeSource Mode="modeEnumValue"/>
  </Binding.RelativeSource>
</Binding>
```

- 或 -

XML

```
<Binding>
  <Binding.RelativeSource>
    <RelativeSource
      Mode="FindAncestor"
      AncestorType="{x:Type typeName}"
      AncestorLevel="intLevel"
    />
  </Binding.RelativeSource>
</Binding>
```

# XAML 值

值	描述
modeEnumValue	下列类型之一： <ul style="list-style-type: none"><li>- 字符串标记 <code>Self</code>；对应于通过将 <code>Mode</code> 属性设置为 <code>Self</code> 而创建的 <code>RelativeSource</code>。</li><li>- 字符串标记 <code>TemplatedParent</code>；对应于通过将 <code>Mode</code> 属性设置为 <code>TemplatedParent</code> 而创建的 <code>RelativeSource</code>。</li><li>- 字符串标记 <code>PreviousData</code>；对应于通过将 <code>Mode</code> 属性设置为 <code>PreviousData</code> 而创建的 <code>RelativeSource</code>。</li><li>- 有关 <code>FindAncestor</code> 模式的信息，请参阅下文。</li></ul>
FindAncestor	字符串标记 <code>FindAncestor</code> 。使用此标记可输入一个模式，以便 <code>RelativeSource</code> 指定上级类型和可选的上级级别。它对应于通过将 <code>RelativeSource</code> 属性设置为 <code>Mode</code> 而创建的 <code>FindAncestor</code> 。
typeName	对于 <code>FindAncestor</code> 模式是必需的。类型的名称，用于填充 <code>AncestorType</code> 属性。
intLevel	对于 <code>FindAncestor</code> 模式是可选的。上级级别（在逻辑树中向父级方向计算）。

## 注解

`{RelativeSource TemplatedParent}` 绑定用法是解析分离控件的用户界面和控件的逻辑这一大概念的关键方法。这可以实现从模板定义内绑定到模板化父级（在其中应用模板的运行时对象实例）。这种情况下，[TemplateBinding Markup Extension](#) 其实是以下绑定表达式的简写：`{Binding RelativeSource={RelativeSource TemplatedParent}}`。

`TemplateBinding` 或 `{RelativeSource TemplatedParent}` 用法仅在用于定义模板的 XAML 内部适用。有关详细信息，请参阅 [TemplateBinding 标记扩展](#)。

`{RelativeSource FindAncestor}` 主要用于控件模板或可预测的自包含 UI 组合，在此情况下，控件应该始终位于某个上级类型的可视化树中。例如，项控件的项可能使用 `FindAncestor` 用法绑定到其项控件父级/祖先级。或者，属于模板中控件组合一部分的元素可使用 `FindAncestor` 绑定到同一组合结构中的父元素。

在 XAML 语法章节显示的 `FindAncestor` 模式的对象元素语法中，第二个对象元素语法专门用于 `FindAncestor` 模式。`FindAncestor` 模式需要 `AncestorType` 值。必须使用要查找的上级类型的 [x>Type 标记扩展](#)引用，将 `AncestorType` 设置为一个特性。当在运行时处理绑定请求时，将会用到 `AncestorType` 值。

对于 `FindAncestor` 模式，当元素树中可能存在多个该类型的上级时，可以使用可选属性 `AncestorLevel` 帮助消除上级查找的歧义。

有关如何使用 `FindAncestor` 模式的详细信息，请参阅 [RelativeSource](#)。

`{RelativeSource Self}` 在以下情形下很有用：即一个实例的一个属性应依赖同一个实例的另一个属性的值，并且这两个属性之间不存在任何一般依赖属性关系（如强制）。虽然很少有两个属性位于同一对象上的情况（这种情况下，它们的值可以说完全相同，并且以相同方式键入），但你也可以将一个 `Converter` 参数应用于具有 `{RelativeSource Self}` 的绑定，并使用此转换器在源类型和目标类型之间进行转换。`{RelativeSource Self}` 的另一种情形是作为 `MultiDataTrigger` 的一部分。

例如，以下 XAML 定义了一个 `Rectangle` 元素，以便无论为 `Width` 输入什么值，都确保 `Rectangle` 始终是一个方形：`<Rectangle Width="200" Height="{Binding RelativeSource={RelativeSource Self}, Path=Width}" .../>`

`{RelativeSource PreviousData}` 在数据模板中很有用，在绑定使用集合作为数据源的情况下也很有用。你可使用 `{RelativeSource PreviousData}` 来突出集合中相邻数据项之间的关系。相关技术是在数据源中的当前项和前一个项之间建立 `MultiBinding`，并使用此绑定上的转换器来确定这两个项及其属性的差异。

在下面的示例中，项目模板中的第一个 `TextBlock` 可显示当前编号。第二个 `TextBlock` 绑定是 `MultiBinding`，名义上有两个 `Binding` 构成要素：当前记录，以及通过使用 `{RelativeSource PreviousData}` 刻意采用前一个数据记录的绑定。然后，`MultiBinding` 上的转换器将计算差异，并将其返回到绑定。

XML

```
<ListBox Name="fibolist">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding}" />
                <TextBlock>, difference = </TextBlock>
                <TextBlock>
                    <TextBlock.Text>
                        <MultiBinding Converter="{StaticResource DiffConverter}">
                            <Binding/>
                            <Binding RelativeSource="{RelativeSource PreviousData}" />
                        </MultiBinding>
                    </TextBlock.Text>
                </TextBlock>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

```
</ListBox.ItemTemplate>  
</ListBox>
```

本文未介绍数据绑定概念，请参阅[数据绑定概述](#)。

在 WPF XAML 处理器实现中，对此标记扩展的处理由 [RelativeSource](#) 类定义。

[RelativeSource](#) 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。XAML 中的所有标记扩展在其属性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理属性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 请参阅

- [Binding](#)
- [样式设置和模板化](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)
- [数据绑定概述](#)
- [绑定声明概述](#)
- [x:Type 标记扩展](#)

# StaticResource 标记扩展

项目 • 2023/02/06

通过查找对已定义资源的引用，为任何 XAML 属性提供值。查找该资源的行为类似于加载时查找，将查找先前从当前 XAML 页面的标记以及其他应用程序源中加载的资源，并将生成该资源值作为运行时对象中的属性值。

## XAML 属性用法

XML

```
<object property="{StaticResource key}" ... />
```

## XAML 对象元素用法

XML

```
<object>
  <object.property>
    <StaticResource ResourceKey="key" ... />
  </object.property>
</object>
```

## XAML 值

### 值 描述

`key` 所请求资源的密钥。如果资源在标记中创建，则此键最初由 [x:Key 指令](#) 分配，如果资源在代码中创建，则此键在调用 [ResourceDictionary.Add](#) 时作为 `key` 参数提供。

## 注解

### ① 重要

`StaticResource` 不应尝试正向引用一个在 XAML 文件中按词法进一步定义的资源。不支持此尝试，即使此类引用没有失败，当搜索表示 [ResourceDictionary](#) 的内部哈希表时，尝试正向引用也会导致加载时间性能损失。为实现最佳效果，请调整你的

资源字典的组成，以避免使用正向引用。如果无法避免正向引用，请改用 [DynamicResource 标记扩展](#)。

指定的 [ResourceKey](#) 在某种程度上应对应于使用 [x:Key](#) 指令在你的页面、应用程序、可用控件主题和外部资源或系统资源中标识的现有资源。资源查找将按该顺序进行。有关静态和动态资源查找的详细信息，请参阅 [XAML 资源](#)。

资源键可以是以 [XamlName 语法](#) 定义的任何字符串。资源键也可以是其他对象类型，例如 [Type](#)。[Type](#) 键是如何通过隐式样式键按主题设置控件样式的基础。有关详细信息，请参阅[控件创作概述](#)。

引用资源的替代声明性方法与 [DynamicResource 标记扩展](#)一样。

特性语法是最常用于该标记扩展的语法。在 [StaticResource](#) 标识符字符串之后提供的字符串标记被指定为基础 [ResourceKey](#) 扩展类的 [StaticResourceExtension](#) 值。

[StaticResource](#) 可以在对象元素语法中使用。在这种情况下，需要指定 [ResourceKey](#) 属性的值。

[StaticResource](#) 还可以在详细特性用法中使用，以便将 [ResourceKey](#) 属性指定为一个 `property=value` 对：

XML

```
<object property="{StaticResource ResourceKey=key}" ... />
```

如果扩展具有一个以上的可设置属性，或者某些属性是可选的，则详细用法通常会很有用。由于 [StaticResource](#) 仅有一个可设置的属性，并且此属性是必需的，因此该详细用法不具有典型性。

在 WPF XAML 处理器实现中，对此标记扩展的处理由 [StaticResourceExtension](#) 类来定义。

[StaticResource](#) 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。XAML 中的所有标记扩展在其特性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理特性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 请参阅

- [样式设置和模板化](#)
- [WPF 中的 XAML](#)

- 标记扩展和 WPF XAML
- XAML 资源
- 资源和代码

# TemplateBinding 标记扩展

项目 • 2023/02/06

连接某一控件模板中的属性值，使之成为模板化控件上另一个属性的值。

## XAML 属性用法

XML

```
<object property="{TemplateBinding sourceProperty}" ... />
```

## XAML 特性用法（适用于模板或样式的 Setter 属性）

XML

```
<Setter Property="propertyName" Value="{TemplateBinding sourceProperty}" ... />
```

## XAML 值

值	描述
propertyName	在资源库语法中设置的属性的 <a href="#">DependencyProperty.Name</a> 。
sourceProperty	另一个在要模板化的类型上存在的依赖项属性，由其 <a href="#">DependencyProperty.Name</a> 来指定。  - 或 -  由要模板化的目标类型之外的类型所定义的“dotted-down”属性名称。这实际上是 <a href="#">PropertyPath</a> 。请参阅 <a href="#">PropertyPath XAML 语法</a> 。

## 注解

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}, Mode=OneWay}` 构造的 `Binding`。  
`TemplateBinding` 始终为单向绑定，即使所涉及的属性默认为双向绑定。所涉及的两个属性都必须是依赖项属性。为了实现对模板化父级的双向绑定，请改用以下绑定语句

```
{Binding RelativeSource={RelativeSource TemplatedParent}, Mode=TwoWay,  
Path=MyDependencyProperty}。
```

[RelativeSource](#) 是另一个标记扩展，有时与 [TemplateBinding](#) 结合使用或者代替它使用，以便在模板中执行相对属性绑定。

此处未介绍控件模板概念；有关详细信息，请参阅[控件样式和模板](#)。

特性语法是最常用于该标记扩展的语法。在 [TemplateBinding](#) 标识符字符串之后提供的字符串标记被指定为基础 [Property](#) 扩展类的 [TemplateBindingExtension](#) 值。

对象元素语法也可行，但因为没有实际的应用，所以未进行演示。[TemplateBinding](#) 用于使用计算的表达式来填充资源库内的值，因此使用 [TemplateBinding](#) 的对象元素语法来填充 `<Setter.Property>` 属性元素语法就会变得繁冗而多余。

[TemplateBinding](#) 还可以在详细特性用法中使用，以便将 [Property](#) 属性指定为一个 `property=value` 对：

XML

```
<object property="{TemplateBinding Property=sourceProperty}" ... />
```

如果扩展具有一个以上的可设置属性，或者某些属性是可选的，则详细用法通常会很有用。由于 [TemplateBinding](#) 仅有一个可设置的属性，并且此属性是必需的，因此该详细用法不具有典型性。

在 WPF XAML 处理器实现中，对此标记扩展的处理由 [TemplateBindingExtension](#) 类定义。

[TemplateBinding](#) 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。XAML 中的所有标记扩展在其属性语法中均使用 `{` 和 `}` 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理属性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 请参阅

- [Style](#)
- [ControlTemplate](#)
- [样式设置和模板化](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)

- RelativeSource MarkupExtension
- 绑定标记扩展

# ThemeDictionary 标记扩展

项目 • 2023/02/06

为集成第三方控件的自定义控件创作者或应用程序提供一种方法，用于加载要在设置控件样式时使用的特定于主题的资源字典。

## XAML 属性用法

XML

```
<object property="{ThemeDictionary assemblyUri}" ... />
```

## XAML 对象元素用法

XML

```
<object>
  <object.property>
    <ThemeDictionary AssemblyName="assemblyUri"/>
  <object.property>
<object>
```

## XAML 值

值	描述
assemblyUri	包含主题信息的程序集的统一资源标识符 (URI)。通常，这是一个引用较大包中程序集的 Pack URI。程序集资源和 Pack URI 可以简化部署问题。有关详细信息，请参阅 <a href="#">WPF 中的 Pack URI</a> 。

## 注解

此扩展旨在仅填充一个特定的属性值：[ResourceDictionary.Source](#) 的值。

通过使用此扩展，可以指定一个纯资源程序集，该程序集包含一些仅在向用户系统应用 Windows Aero 主题时使用的样式、仅在当 Luna 主题处于活动状态时使用的其他样式等。通过使用此扩展，可以自动让特定于控件的资源字典的内容失效，并在需要时重新将其加载为特定于其他主题的内容。

`assemblyUri` 字符串 (`AssemblyName` 属性值) 构成命名约定的基础，命名约定用于标识适用于特定主题的字典。`ThemeDictionary` 的 `ProvideValue` 逻辑通过生成一个指向某个特殊主题字典变体 (包含在预编译的资源程序集中) 的统一资源标识符 (URI) 来完成约定。本文并未将该约定 (即主题与一般控件样式设置和页/应用程序级样式设置之间的交互) 作为一个概念进行详细介绍。使用 `ThemeDictionary` 的基本场景是指定在应用程序级声明的 `ResourceDictionary` 的 `Source` 属性。将 URI 通过 `ThemeDictionary` 扩展而不是作为一个直接的 URI 提供给程序集时，扩展逻辑将提供系统主题更改时适用的失效逻辑。

特性语法是最常用于该标记扩展的语法。在 `ThemeDictionary` 标识符字符串之后提供的字符串标记被指定为基础 `AssemblyName` 扩展类的 `ThemeDictionaryExtension` 值。

`ThemeDictionary` 还可以在对象元素语法中使用。在这种情况下，需要指定 `AssemblyName` 属性的值。

`ThemeDictionary` 还可以在详细特性用法中使用，以便将 `Member` 属性指定为一个 `property=value` 对：

XML

```
<object property="{ThemeDictionary AssemblyName=assemblyUri}" ... />
```

如果扩展具有一个以上的可设置属性，或者某些属性是可选的，则详细用法通常会很有用。由于 `ThemeDictionary` 仅有一个可设置的属性，并且此属性是必需的，因此该详细用法不具有典型性。

在 WPF XAML 处理器实现中，对此标记扩展的处理由 `ThemeDictionaryExtension` 类定义。

`ThemeDictionary` 是标记扩展。当要求转义特性值应为非文本值或非处理程序名称时，通常会实现标记扩展，相对于只在某些类型或属性上放置类型转换器而言，此需求更具有全局性。XAML 中的所有标记扩展在其属性语法中均使用 { 和 } 字符，正是依据这一约定，XAML 处理器认定标记扩展必须处理属性。有关详细信息，请参阅[标记扩展和 WPF XAML](#)。

## 请参阅

- [样式设置和模板化](#)
- [WPF 中的 XAML](#)
- [标记扩展和 WPF XAML](#)
- [WPF 应用程序资源、内容和数据文件](#)

# PropertyPath XAML 语法

项目 · 2022/10/19

PropertyPath 对象支持复杂的内联 XAML 语法，用于设置将 PropertyPath 类型作为其值的各种属性。本主题讨论应用于绑定的 PropertyPath 语法和动画语法。

## PropertyPath 的使用情景

PropertyPath 是在多个 Windows Presentation Foundation (WPF) 功能中使用的通用对象。尽管使用通用 PropertyPath 传输属性路径信息，但是将 PropertyPath 用作类型的每个功能区域的用法却各不相同。因此，基于每个功能来讨论语法更为可行。

WPF 主要使用 PropertyPath 来描述用于遍历对象数据源属性的对象模型路径，以及描述目标动画的目标路径。

某些样式和模板属性（例如 Setter.Property）采用表面上类似于 PropertyPath 的限定属性名称。但是，这不是实际的 PropertyPath，而是由 WPF XAML 处理器与 DependencyProperty 的类型转换器联合启用的限定的 owner.property 字符串格式用法。

## 数据绑定中对象的 PropertyPath

数据绑定是一项 WPF 功能，借此可以绑定到任意依赖属性的目标值。但是，此类数据绑定的源不需要是依赖属性；可以是适用数据提供程序能识别的任意属性类型。属性路径特别适用于 ObjectDataProvider，后者用于从公共语言运行时 (CLR) 对象及其属性获取绑定源。

请注意，由于与 XML 的数据绑定不使用 Binding 中的 Path，因此它不使用 PropertyPath，而是使用 XPath 并在数据的 XML 文档对象模型 (DOM) 中指定有效的 XPath 语法。XPath 也被指定为字符串，但不会在此讨论；请参阅[使用 XMLDataProvider 和 XPath 查询绑定到 XML 数据](#)。

理解数据绑定中的属性路径的关键是将绑定到单个属性值设置为目标，或者改为绑定到采用列表或集合的目标属性。如果要绑定集合（例如绑定将根据集合中的数据项数量而展开的 [ListBox](#)），则属性路径应引用集合对象，而不是单个集合项。数据绑定引擎会自动将用作数据源的集合与绑定目标的类型匹配，从而导致使用项数组填充 [ListBox](#) 等行为。

## 直接对象上作为数据上下文的单个属性

XML

```
<Binding Path="propertyName" ... />
```

`propertyName` 必须解析为当前 `DataContext` 中用于 `Path` 的属性名称。如果绑定更新源，则属性必须是可读取/写入的，并且源对象必须可变。

## 直接对象上作为数据上下文的单个索引器

XML

```
<Binding Path="[key]" ... />
```

`key` 必须是字典或哈希表的类型化索引，或者是数组的整数索引。此外，键值必须是可直接绑定到所应用属性的类型。例如，可以通过这种方式使用包含字符串键和字符串值的哈希表以绑定到 `TextBox` 的文本。或者，如果键指向集合或子索引，则可使用此语法绑定到目标集合属性。否则，需要通过 `<Binding Path="[key].propertyName" .../>` 等语法来引用特定属性。

如有必要，可以指定索引的类型。有关索引属性路径这方面的详细信息，请参阅 [Binding.Path](#)。

## 多个属性（间接属性目标设置）

XML

```
<Binding Path="propertyName.propertyName2" ... />
```

`propertyName` 必须解析为作为当前 `DataContext` 的属性名称。路径属性 `propertyName` 和 `propertyName2` 可以是关系中的任意属性，其中 `propertyName2` 是存在于类型中的值为 `propertyName` 的属性。

## 附加的或类型限定的单个属性

XML

```
<object property="(ownerType.propertyName)" ... />
```

圆括号表示 `PropertyPath` 中的此属性应使用部分限定来构建。它可以使用 XML 命名空间来查找具有适当映射的类型。`ownerType` 通过每个程序集中的 `XmlNsDefinitionAttribute` 声明搜索 XAML 处理器有权访问的类型。大部分应用程序都具有映射到 `http://schemas.microsoft.com/winfx/2006/xaml/presentation` 命名空间的默认 XML 命名空间，因此通常仅有自定义类型或该命名空间之外的类型才需要前缀。

`propertyName` 必须解析为 `ownerType` 中存在的属性名称。此语法一般用于以下任一情况：

- 路径是在 XAML 中的样式或模板（该样式或模板没有指定的目标类型）中指定的。除此之外，限定用法一般无效，因为在非样式、非模板情况下，属性存在于实例中，而不是类型中。
- 属性为附加属性。
- 要绑定到静态属性中。

用作情节提要目标时，指定为 `propertyName` 的属性必须为 [DependencyProperty](#)。

## 源遍历（绑定到集合的层次结构）

XML

```
<object Path="propertyName/propertyNameX" ... />
```

此语法中的 / 用于在分层数据源对象中导航，并且支持使用连续的 / 字符分多个步骤导航层次结构。源遍历说明了当前记录指针位置，该位置是通过将数据与其视图的 UI 同步而确定的。有关与分层数据源对象绑定的详细信息，以及数据绑定中当前记录指针的概念，请参阅对分层数据使用主-从模式或数据绑定概述。

### ① 备注

从表面上看，此语法类似于 XPath。用于绑定到 XML 数据源的真正 XPath 表达式不用作 `Path` 值，而应用于互斥的 `XPath` 属性。

## 集合视图

若要引用一个已命名的集合视图，请使用哈希字符 (#) 为集合视图名称添加前缀。

## 当前记录指针

若要引用集合视图的当前记录指针或引用主从数据绑定方案，请启用带正斜杠 (/) 的路径字符串。从当前记录指针开始遍历任何超出正斜杠的路径。

## 多个索引器

XAML

```
<object Path="[index1,index2...]" ... />
```

或

XAML

```
<object Path="propertyName[index,index2...]" ... />
```

如果给定的对象支持多个索引器，则可以按顺序指定这些索引器，类似于数组引用语法。上述对象可以是当前上下文，也可以是包含多个索引对象的属性的值。

默认情况下，通过使用基础对象的特性来类型化索引器值。如有必要，可以指定索引的类型。有关键入索引器的详细信息，请参阅 [Binding.Path](#)。

## 混合语法

上述每条语法都可以独立使用。例如，下述示例在包含 `SolidColorBrush` 对象的像素网格数组的 `ColorGrid` 属性的特定 `x`、`y` 处创建了颜色的属性路径：

XML

```
<Rectangle Fill="{Binding ColorGrid[20,30].SolidColorBrushResult}" ... />
```

## 属性路径字符串的转义

对于某些业务对象，你可能会遇到这样的情况：属性路径字符串需要转义序列以进行正确分析。因为大多数此类字符在常用于定义业务对象的语言方面具有类似的命名交互问题，因此转义需要非常少见。

- 在索引器 ([ ]) 内部，脱字符号 (^) 用于对下一个字符进行转义。
- 必须（使用 XML 实体）对 XML 语言定义专用的某些字符进行转义。使用 & 对字符“&”进行转义。使用 > 对结束标记“>”进行转义。
- 必须（使用反斜杠 \ ）对特定于 WPF XAML 分析程序行为的字符进行转义，以处理标记扩展。
  - 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。

### ① 备注

从技术上讲，这些转义符还适用于情节提要属性路径，但通常会遍历适用于现有 WPF 对象的对象模型，转义应该是不必要的。

## 动画目标的 PropertyPath

动画的目标属性必须是采用 [Freezable](#) 或基元类型的依赖属性。但是，类型中的目标属性和最终动画属性可以存在于不同的对象中。对于动画，属性路径用于通过遍历属性值中的对象-属性关系，定义命名动画目标对象的属性和预期目标动画属性之间的连接。

### 动画的一般对象-属性注意事项

有关一般动画概念的详细信息，请参阅[情节提要概述](#)和[动画概述](#)。

要进行动画处理的值类型或属性必须是 [Freezable](#) 类型或基元。启动路径的属性必须解析为存在于指定的 [TargetName](#) 类型中的依赖属性的名称。

为了支持对已冻结的 [Freezable](#) 进行动画处理的克隆，由 [TargetName](#) 指定的对象必须是 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生类。

### 目标对象上的单个属性

#### XML

```
<animation Storyboard.TargetProperty="propertyName" ... />
```

`propertyName` 必须解析为存在于指定的 [TargetName](#) 类型中的依赖项属性的名称。

### 间接属性目标设定

#### XML

```
<animation Storyboard.TargetProperty="propertyName.propertyName2" ... />
```

`propertyName` 必须是存在于指定 [TargetName](#) 类型中的 [Freezable](#) 值类型或基元的属性。

`propertyName2` 必须为依赖属性的名称，该属性存在于作为 `propertyName` 值的对象中。

也就是说，`propertyName2` 必须作为 `propertyName` `PropertyType` 类型上的依赖属性存在。

因为应用了样式和模板，所以间接设定动画的目标是必要的。若要定位动画，需在目标对象上添加一个 `TargetName`，并且该名称由 `x:Name` 或 `Name` 创建。虽然模板和样式元素也可以有名称，但这些名称仅在样式和模板的命名范围内有效。（如果模板和样式确实与应用程序标记共享名称范围，则名称不能是唯一的。样式和模板在实例之间按字面形式共享，将永久延续重复名称。）因此，如果想要对元素进行动画处理的元素的各个属性来自样式或模板，则需要从不是样式模板的命名元素实例开始，然后定位到样式或模板可视化树中，以到达要进行动画处理的属性。

例如，`Panel` 的 `Background` 属性是来自主题模板的完整 `Brush`（实际上是 `SolidColorBrush`）。若要对 `Brush` 进行完全动画处理，需要 `BrushAnimation`（可能每个 `Brush` 类型都需要一个），然而没有这样的类型。若要对 `Brush` 进行动画处理，请改为对特定 `Brush` 类型的属性进行动画处理。需要获取从 `SolidColorBrush` 到其 `Color` 才能在何处应用 `ColorAnimation`。本示例的属性路径是 `Background.Color`。

## 附加属性

XML

```
<animation Storyboard.TargetProperty="(ownerType.propertyName)" ... />
```

圆括号表示 `PropertyPath` 中的此属性应使用部分限定来构建。可以使用 XML 命名空间来查找类型。`ownerType` 搜索将 WPF 附加属性实现为依赖属性的类型，因此该问题仅与自定义附加属性有关。）

## 索引器

XML

```
<animation
Storyboard.TargetProperty="propertyName.propertyName2[index].propertyName3"
... />
```

大部分依赖属性或 `Freezable` 类型不支持索引器。因此，动画路径中唯一使用索引器的地方是在用于启动命名目标上的链的属性与最终动画属性之间的中间位置。在提供的语法中，为 `propertyName2`。例如，如果中间属性是属性路径（例如 `RenderTransform.Children[1].Angle`）中的集合（例如 `TransformGroup`），则可能需要使用索引器。

# 代码中的 PropertyPath

[PropertyPath](#) 的参考主题中介绍了 [PropertyPath](#) 的代码用法，包括如何构造 [PropertyPath](#)。

一般而言，[PropertyPath](#) 使用两个不同的构造函数，一个用于绑定用法和最简单的动画用法，另一个用于复杂动画用法。将 [PropertyPath\(Object\)](#) 签名用于绑定用法，其中对象为字符串。将 [PropertyPath\(Object\)](#) 签名用于单步动画路径，其中对象为 [DependencyProperty](#)。将 [PropertyPath\(String, Object\[\]\)](#) 签名用于复杂动画。后一种构造函数使用第一个参数的令牌字符串，以及在该令牌字符串中填充位置的对象的数组，以定义属性路径关系。

## 另请参阅

- [PropertyPath](#)
- [数据绑定概述](#)
- [演示图板概述](#)

# PresentationOptions:Freeze 特性

项目 • 2023/02/06

在包含内容的 `Freezable` 元素上，将 `IsFrozen` 状态设置为 `true`。未指定 `PresentationOptions:Freeze` 属性的 `Freezable` 的默认行为是，`IsFrozen` 在加载时为 `false`，并在运行时取决于一般的 `Freezable` 行为。

## XAML 属性用法

### XAML

```
<object  
  
    xmlns:PresentationOptions="http://schemas.microsoft.com/winfx/2006/xaml/pres  
entation/options"  
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  
    mc:Ignorable="PresentationOptions">  
        <freezableElement PresentationOptions:Freeze="true"/>  
    </object>
```

## XAML 值

值	描述
<code>PresentationOptions</code>	XML 命名空间前缀，根据 XML 1.0 规范，它可以是任何有效的前缀字符串。前缀 <code>PresentationOptions</code> 在本文档中用于标识。
<code>freezableElement</code>	用于实例化 <code>Freezable</code> 的任何派生类的元素。

## 注解

`Freeze` 属性是 `http://schemas.microsoft.com/winfx/2006/xaml/presentation/options` XML 命名空间中定义的唯一属性或其他编程元素。`Freeze` 属性专门存在于此特殊命名空间中，以便在根元素声明中可以使用 `mc:Ignorable Attribute` 将其指定为可忽略。`Freeze` 必须能够可忽略的原因是，并非所有 XAML 处理器实现都能够在加载时冻结 `Freezable`；此功能不是 XAML 规范的一部分。

若在加载时处理 `Freeze` 元素上的 `Freezable` 属性，处理 `Freeze` 属性的功能会专门内置于 WPF XAML 处理器中。

`Freeze` 属性除 `true` 以外的任何值（不区分大小写）会产生一个加载时错误。（将 `Freeze` 属性指定为 `false` 不是错误，但它已是默认值，因此设置为 `false` 将不执行任何操作）。

## 另请参阅

- [Freezable](#)
- [Freezable 对象概述](#)
- [mc:Ignorable 特性](#)

# 标记兼容 (mc:)语言功能

项目 • 2023/02/06

## 本节内容

[mc:Ignorable 特性](#)

[mc:ProcessContent 特性](#)

# mc:Ignorable 特性

项目 • 2023/02/06

指定 XAML 处理器可以忽略标记文件中的哪些 XML 命名空间前缀。 mc:Ignorable 特性支持自定义命名空间映射和 XAML 版本控制的标记兼容。

## XAML 特性用法 ( 单一前缀 )

XAML

```
<object
    xmlns:ignorablePrefix="ignorableUri"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="ignorablePrefix"...
        <ignorablePrefix1:ThisElementCanBeIgnored/>
</object>
```

## XAML 特性用法 ( 两个前缀 )

XAML

```
<object
    xmlns:ignorablePrefix1="ignorableUri"
    xmlns:ignorablePrefix2="ignorableUri2"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="ignorablePrefix1 ignorablePrefix2"...
        <ignorablePrefix1:ThisElementCanBeIgnored/>
</object>
```

## XAML 值

值	描述
<i>ignorablePrefix</i> 、 <i>ignorablePrefix1</i> 等	根据 XML 1.0 规范，任何有效前缀字符串。
<i>ignorableUri</i>	根据 XML 1.0 规范，用于指定命名空间的任何有效 URI。
<i>ThisElementCanBeIgnored</i>	Extensible Application Markup Language (XAML) 处理器实现（如果无法解析基础类型）可以忽略的元素。

## 注解

`mc` XML 命名空间前缀是映射 XAML 兼容性命名空间

<http://schemas.openxmlformats.org/markup-compatibility/2006> 时推荐使用的前缀约定。

XAML 处理器处理元素名称的前缀部分标识为 `mc:Ignorable` 的元素或特性时不会引发错误。如果该特性无法解析为基础类型或编程构造，则忽略该元素。但是请注意，被忽略的元素可能仍会因其他元素要求而生成其他解析错误，这些错误是未处理该元素所导致的结果。例如，特定元素内容模型可能只需要一个子元素，但如果指定的子元素位于 `mc:Ignorable` 前缀，且指定的子元素无法解析为类型，则 XAML 处理器可能会引发错误。

`mc:Ignorable` 仅适用于将命名空间映射到标识符字符串。`mc:Ignorable` 不适用于将名称空间映射到程序集，它指定 CLR 命名空间和程序集（或将当前可执行文件默认为程序集）。

如果实现的是 XAML 处理器，则对于标识为 `mc:Ignorable` 的前缀所限定的任何元素或特性，处理器实现不得引发关于类型解析的解析错误或处理错误。但处理器实现仍然可能引发异常，这些异常是元素加载失败或处理失败的次要结果，例如上述的单一子元素示例。

默认情况下，XAML 处理器将忽略被忽略元素中的内容。但你可以指定一个附加特性 [mc:ProcessContent 特性](#)，以要求下一个可用父元素继续处理被忽略元素中的内容。

可以在特性中指定多个前缀，并使用一个或多个空格字符作为分隔符，例如：

`mc:Ignorable="ignore1 ignore2"。`

<http://schemas.openxmlformats.org/markup-compatibility/2006> 命名空间定义 SDK 的此区域中未记录的其他元素和特性。有关详细信息，请参阅 [XML 标记兼容规范](#)。

## 另请参阅

- [XamlReader](#)
- [PresentationOptions:Freeze 特性](#)
- [WPF 中的 XAML](#)
- [WPF 中的文档](#)

# mc:ProcessContent 特性

项目 · 2022/09/27

指定哪些 XAML 元素仍然具有相关父元素处理过的内容，即使由于指定 [mc:Ignorable 特性](#)，XAML 处理器可能忽略即时父元素。`mc:ProcessContent` 属性支持自定义命名空间映射和 XAML 版本控制的标记兼容性。

## XAML 属性用法

### XAML

```
<object
    xmlns:ignorablePrefix="ignorableUri"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="ignorablePrefix"...
    mc:ProcessContent="ignorablePrefix:ThisElementCanBeIgnored"
>
    <ignorablePrefix:ThisElementCanBeIgnored>
        [content]
    </ignorablePrefix:ThisElementCanBeIgnored>
</object>
```

## XAML 值

值	描述
ignorablePrefix	根据 XML 1.0 规范，任何有效前缀字符串。
ignorableUri	根据 XML 1.0 规范，用于指定命名空间的任何有效 URI。
ThisElementCanBeIgnored	Extensible Application Markup Language (XAML) 处理器实现（如果无法解析基础类型）可以忽略的元素。
[content]	ThisElementCanBeIgnored 标记为可忽略。如果处理器忽略该元素，则 [content] 由对象处理。

## 注解

默认情况下，XAML 处理器将忽略被忽略元素中的内容。可以通过 `mc:ProcessContent` 指定一个特定元素，XAML 处理器将继续处理被忽略元素中的内容。如果内容嵌套在多个标记中，其中至少有一个是可忽略的，至少有一个是不可忽略的，则通常会使用这种方法。

可以使用空格分隔符在属性中指定多个前缀，例如：

```
mc:ProcessContent="ignore:Element1 ignore:Element2"。
```

<http://schemas.openxmlformats.org/markup-compatibility/2006> 命名空间定义 SDK 的此区域中未记录的其他元素和属性。有关详细信息，请参阅 [XML 标记兼容规范](#)。

## 另请参阅

- [mc:Ignorable 特性](#)
- [WPF 中的 XAML](#)

# 基元素

项目 • 2023/02/06

四个关键类 ([UIElement](#)、[ContentElement](#)、[FrameworkElement](#) 和 [FrameworkContentElement](#)) 实现了 WPF 编程中可用的大部分常见元素功能。 这四个类在此 SDK 中称为基元素类。

## 本节内容

[基元素概述](#)

[Freezable 对象概述](#)

[Alignment、Margin 和 Padding 概述](#)

[操作指南主题](#)

## 参考

[UIElement](#)

[ContentElement](#)

[FrameworkElement](#)

[FrameworkContentElement](#)

## 相关章节

[WPF 体系结构](#)

[WPF 中的 XAML](#)

[元素树和序列化](#)

[属性](#)

[事件](#)

[输入](#)

[资源](#)

[样式设置和模板化](#)

[线程模型](#)

# 基元素概述

项目 · 2023/02/06

Windows Presentation Foundation (WPF) 中较高比重的类都派生自四类，它们通常在 SDK 文档中称为基元素。这些类分别是：[UIElement](#)、[FrameworkElement](#)、[ContentElement](#) 和 [FrameworkContentElement](#)。[DependencyObject](#) 类也相关，因为它属于 [UIElement](#) 和 [ContentElement](#) 的常见基类

## WPF 类中的基元素 API

[UIElement](#) 和 [ContentElement](#) 都派生自 [DependencyObject](#)，但通过的路径略有不同。以此级别进行拆分解决 [UIElement](#) 或 [ContentElement](#) 在用户界面中如何使用以及它们在应用程序中的用途问题。[UIElement](#) 在其类层次结构中还有 [Visual](#)，该类以 Windows Presentation Foundation (WPF) 为基础，公开低级别图形支持。[Visual](#) 通过定义独立矩形屏幕区域提供呈现框架。在实践中，[UIElement](#) 适用于支持较大对象模型的元素，专用于呈现和布局到可描述为矩形屏幕区域的区域，在该区域中，有意将内容模型设置得更加开放，以允许不同的元素组合。[ContentElement](#) 未派生自 [Visual](#)；其模型为 [ContentElement](#) 将由其他对象使用，如读取器或查看器随后可解释元素并生成完整的 [Visual](#) 供 Windows Presentation Foundation (WPF) 使用。某些 [UIElement](#) 类旨在成为内容宿主：它们为一个或多个 [ContentElement](#) 类提供承载和呈现服务（[DocumentViewer](#) 即属于此类）。[ContentElement](#) 用作具有较小对象模型的元素的基类，这些元素更多地处理可能在 [UIElement](#) 中承载的文本、信息或文档内容。

## 框架级别和核心级别

[UIElement](#) 作为 [FrameworkElement](#) 的基类，[ContentElement](#) 作为 [FrameworkContentElement](#) 的基类。此下一级别类的目的是支持独立于 WPF 框架级别的 WPF 核心级别，这种划分也存在于 API 在 [PresentationCore](#) 与 [PresentationFramework](#) 程序集之间的划分方式。WPF 框架级别表示一种更完整的解决方案，以满足基本应用程序需求，其中包括实现演示文稿的布局管理器。WPF 核心级别可提供一种方法，使你能够充分利用它，而无需承担其他程序集的开销。这些级别间的区别对大多数典型的应用程序开发方案几乎没影响，但一般情况下，你应整体考虑 WPF API，而不是关注 WPF 框架级别和 WPF 核心级别间的差异。如果应用程序设计选择替换大量 WPF 框架级别功能，建议了解级别差异，例如，整体解决方案是否已有其自己的用户界面 (UI) 组合和布局的实现。

## 选择要从其中派生的元素

若要创建扩展 WPF 的自定义类，最实用的方法是派生自 WPF 类之一，你可在其中通过现有类层次结构获得尽可能多的所需功能。本节列出了以下三个最重要的元素类附带的功能，以帮助决定要从其中继承的类。

若要实现控件（这实际上是从 WPF 类派生的更常见的一种理由），建议从属于实用控件的类、控件系列基类派生或至少从 [Control](#) 基类派生。有关指导和实际示例，请参阅[控件创作概述](#)。

如果没有创建控件且需要派生自位于层次结构中较高层次的类，以下各节可用于指导在每个基元素类中定义哪些特征。

如果创建派生自 [DependencyObject](#) 的类，将继承以下功能：

- [GetValue](#) 和 [SetValue](#) 支持，以及常规属性系统支持。
- 能够使用依赖属性和实现为依赖属性的附加属性。

如果创建派生自 [UIElement](#) 的类，除 [DependencyObject](#) 提供的功能外，还将继承以下功能：

- 对已动画处理的属性值的基本支持。有关详细信息，请参阅[动画概述](#)。
- 基本输入事件支持以及命令支持。有关详细信息，请参阅[输入概述](#)和[命令概述](#)。
- 可进行替代以便介绍布局系统的虚拟方法。

如果创建派生自 [FrameworkElement](#) 的类，除 [UIElement](#) 提供的功能外，还将继承以下功能：

- 样式和情节提要支持。有关详细信息，请参阅[Style](#) 和[情节提要概述](#)。
- 数据绑定支持。有关详细信息，请参阅[数据绑定概述](#)。
- 动态资源引用支持。有关详细信息，请参阅[XAML 资源](#)。
- 属性值继承支持，以及元数据中的其他标记，这些标记有助于报告关于框架服务属性的情况，例如数据绑定、样式或布局的框架实现。有关详细信息，请参阅[框架属性元数据](#)。
- 逻辑树概念。有关详细信息，请参见[WPF 中的树](#)。
- 支持布局系统的实用 WPF 框架级别的实现，包括可检测对影响布局的属性的更改的 [OnPropertyChanged](#) 替代。

如果创建派生自 [ContentElement](#) 的类，除 [DependencyObject](#) 提供的功能外，还将继承以下功能：

- 支持动画。有关详细信息，请参阅[动画概述](#)。
- 基本输入事件支持以及命令支持。有关详细信息，请参阅[输入概述](#)和[命令概述](#)。

如果创建派生自 [FrameworkContentElement](#) 的类，除 [ContentElement](#) 提供的功能外，还将获取以下功能：

- 样式和情节提要支持。有关详细信息，请参阅[Style](#) 和[动画概述](#)。
- 数据绑定支持。有关详细信息，请参阅[数据绑定概述](#)。
- 动态资源引用支持。有关详细信息，请参阅[XAML 资源](#)。
- 属性值继承支持，以及元数据中的其他标记，这些标记有助于报告关于框架服务属性的情况，例如数据绑定、样式或布局的框架实现。有关详细信息，请参阅[框架属性元数据](#)。
- 你不会继承布局系统修改权限（例如 [ArrangeOverride](#)）。布局系统实现仅对 [FrameworkElement](#) 适用。但是，你将继承 [OnPropertyChanged](#) 替代，该替代可检测影响布局的属性的更改并将这些更改报告给任何内容宿主。

针对各种类记录内容模型。如果想要查找相应的类进行派生，类的内容模型是应该考虑的一个可能因素。有关详细信息，请参阅[WPF 内容模型](#)。

## 其他基类

### DispatcherObject

[DispatcherObject](#) 提供对 WPF 线程模型的支持，且使所有为 WPF 应用程序创建的对象都能够与 [Dispatcher](#) 关联。即使不从 [UIElement](#)、[DependencyObject](#) 或 [Visual](#) 派生，也应考虑从 [DispatcherObject](#) 派生，以便获得该线程模型支持。有关详细信息，请参阅[线程模型](#)。

## 视觉对象

[Visual](#) 实现了 2D 对象的概念，这种概念通常要求在大致矩形区域中进行视觉呈现。[Visual](#) 的实际呈现发生在其他类（非自包含）中，但 [Visual](#) 类提供了一种已知类型，供各个级别的呈现进程使用。[Visual](#) 实现命中测试，但它不公开报告命中测试阳性的事件（位于 [UIElement](#) 中）。有关详细信息，请参阅[可视化层编程](#)。

## Freezable

当出于性能原因要求或需要不可变对象时，[Freezable](#) 通过提供生成对象副本的方式来模拟可变对象的不可变性。[Freezable](#) 类型为某些图形元素（如几何图形、画笔以及动画）提供公共基础。值得注意的是，[Freezable](#) 不是 [Visual](#)；当应用 [Freezable](#) 填充其他对象的属性值时，它可保留成为子属性的属性，而这些子属性可能会影响呈现。有关详细信息，请参阅 [Freezable 对象概述](#)。

## Animatable

[Animatable](#) 是 [Freezable](#) 派生类，它特别添加了动画控件层和一些实用程序成员，以便可以区分当前已动画处理属性和未动画处理属性。

## 控制

[Control](#) 属于称为控件或组件的对象类型的目标基类，具体视技术而定。通常，WPF 控件类是直接表示 UI 控件或积极参与控件组合的类。[Control](#) 实现的主要功能是控件模板化。

## 另请参阅

- [Control](#)
- [依赖项属性概述](#)
- [控件创作概述](#)
- [WPF 体系结构](#)

# Freezable 对象概述

项目 • 2022/09/27

本主题介绍如何高效使用和创建 [Freezable](#) 对象，这些对象提供有助于提高应用程序性能的特殊功能。 [Freezable](#) 对象的示例包括画笔、笔、转换、几何图形和动画。

## 什么是 Freezable？

[Freezable](#) 是一种特殊类型的对象，具有两种状态：解冻和冻结。 解冻后，[Freezable](#) 的行为看起来与任何其他对象一样。 冻结后，不能再修改 [Freezable](#)。

[Freezable](#) 提供 [Changed](#) 事件来通知观察者对象发生的任何修改。 冻结 [Freezable](#) 可以提高其性能，因为它不再需要在更改通知上消耗资源。 冻结的 [Freezable](#) 还可以跨线程共享，解冻的 [Freezable](#) 则不能。

尽管 [Freezable](#) 类有许多应用程序，但 Windows Presentation Foundation (WPF) 中的大多数 [Freezable](#) 对象都与图形子系统相关。

借助 [Freezable](#) 类，可以更轻松地使用某些图形系统对象，并有助于提高应用程序性能。 从 [Freezable](#) 继承的类型示例包括 [Brush](#)、[Transform](#) 和 [Geometry](#) 类。 由于它们包含非托管资源，因此系统必须监视这些对象发生的修改，然后在原始对象发生更改时更新对应的非托管资源。 即使实际上并未修改图形系统对象，系统仍必须消耗一些资源来监视该对象，以防你更改它。

例如，假设创建一个 [SolidColorBrush](#) 画笔并用它来绘制按钮的背景。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
myButton.Background = myBrush;
```

呈现按钮时，WPF 图形子系统使用你提供的信息来绘制一组像素，以创建按钮的外观。 尽管使用纯色画笔来描述按钮的绘制方式，但纯色画笔实际上并没有进行绘制。 图形系统为按钮和画笔生成快速、低级别的对象，实际显示在屏幕上的就是这些对象。

如果要修改画笔，则必须重新生成这些低级别对象。 [Freezable](#) 类使画笔能够找到生成的相应低级别对象并在更改时更新它们。 启用此功能后，画笔即“解冻”。

使用 [Freezable](#) 的 [Freeze](#) 方法可以禁用此自更新功能。 你可以使用此方法使画笔“冻结”或无法修改。

## ① 备注

并非每个 [Freezable](#) 对象都可以冻结。为避免引发 [InvalidOperationException](#)，请在尝试冻结 [Freezable](#) 对象之前检查该对象的 [CanFreeze](#) 属性值，以确定是否可以将其冻结。

C#

```
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}
```

当你不再需要修改 [Freezable](#) 对象时，冻结它会带来性能优势。如果冻结此示例中的画笔，图形系统将不再需要监视其更改。图形系统还可以进行其他优化，因为它知道画笔不会更改。

## ① 备注

为方便起见，除非显式冻结 [Freezable](#) 对象，否则它们将保持解冻状态。

# 使用 [Freezable](#)

使用解冻的 [Freezable](#) 就像使用任何其他类型的对象一样。在以下示例中，在使用 [SolidColorBrush](#) 绘制按钮背景后，其颜色从黄色变为红色。图形系统在后台工作，以在下次刷新屏幕时自动将按钮从黄色更改为红色。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
myButton.Background = myBrush;

// Changes the button's background to red.
myBrush.Color = Colors.Red;
```

## 冻结 [Freezable](#)

若要使 [Freezable](#) 不可修改，请调用其 [Freeze](#) 方法。冻结包含 [Freezable](#) 对象的对象时，这些对象也会被冻结。例如，如果冻结 [PathGeometry](#)，它所包含的图形和段也会被

冻结。

如果满足以下任一条件，则无法冻结 **Freezable**：

- 它具有动画属性或数据绑定属性。
- 它具有由动态资源设置的属性。 (有关动态资源的详细信息，请参阅 [XAML 资源](#)。)
- 它包含无法冻结的 **Freezable** 子对象。

如果这些条件不成立，并且你不打算修改 **Freezable**，则应冻结它，以获得前面所述的性能优势。

一旦调用了 **Freezable** 的 **Freeze** 方法，就无法再对其进行修改。尝试修改冻结对象会导致引发 **InvalidOperationException**。以下代码会引发异常，因为我们试图在冻结画笔后对其进行修改。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

try {
    // Throws an InvalidOperationException, because the brush is frozen.
    myBrush.Color = Colors.Red;
} catch(InvalidOperationException ex)
{
    MessageBox.Show("Invalid operation: " + ex.ToString());
}
```

为避免引发此异常，可以使用 **IsFrozen** 方法来确定 **Freezable** 是否已冻结。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);
```

```
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

if (myBrush.IsFrozen) // Evaluates to true.
{
    // If the brush is frozen, create a clone and
    // modify the clone.
    SolidColorBrush myBrushClone = myBrush.Clone();
    myBrushClone.Color = Colors.Red;
    myButton.Background = myBrushClone;
}
else
{
    // If the brush is not frozen,
    // it can be modified directly.
    myBrush.Color = Colors.Red;
}
```

前面的代码示例使用 [Clone](#) 方法创建了冻结对象的可修改副本。下一部分将更详细地讨论克隆。

### ① 备注

由于无法对冻结的 [Freezable](#) 进行动画处理，因此当你尝试使用 [Storyboard](#) 对其进行动画处理时，动画系统会自动创建冻结的 [Freezable](#) 对象的可修改克隆。为了消除克隆导致的性能开销，如果你打算对对象进行动画处理，请让其保持解冻状态。有关使用情节提要进行动画处理的详细信息，请参阅[情节提要概述](#)。

## 从标记冻结

若要冻结标记中声明的 [Freezable](#) 对象，请使用 [PresentationOptions:Freeze](#) 属性。以下示例将 [SolidColorBrush](#) 声明为页面资源并将其冻结。然后，使用它来设置按钮背景。

XAML

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

    xmlns:PresentationOptions="http://schemas.microsoft.com/winfx/2006/xaml/pres
    entation/options"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
```

```
mc:Ignorable="PresentationOptions">

<Page.Resources>

    <!-- This resource is frozen. -->
    <SolidColorBrush
        x:Key="MyBrush"
        PresentationOptions:Freeze="True"
        Color="Red" />
</Page.Resources>

<StackPanel>

    <Button Content="A Button"
        Background="{StaticResource MyBrush}">
    </Button>

</StackPanel>
</Page>
```

若要使用 `Freeze` 属性，必须映射到呈现选项命名空间：

<http://schemas.microsoft.com/winfx/2006/xaml/presentation/options>。

`PresentationOptions` 是推荐用于映射此命名空间的前缀：

XAML

```
xmlns:PresentationOptions="http://schemas.microsoft.com/winfx/2006/xaml/pres
entation/options"
```

由于并非所有 XAML 读取器都能识别此属性，因此建议使用 `mc:Ignorable` 属性将 `Presentation:Freeze` 属性标记为可忽略：

XAML

```
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
mc:Ignorable="PresentationOptions"
```

有关详细信息，请参阅 [mc:Ignorable 属性](#) 页面。

## “解冻”Freezable

一旦冻结，`Freezable` 就永远无法修改或解冻；但是，可以使用 `Clone` 或 `CloneCurrentValue` 方法创建解冻克隆。

以下示例使用画笔设置按钮背景，然后冻结该画笔。 使用 `Clone` 方法创建画笔的解冻副本。 修改克隆并使用它将按钮的背景从黄色更改为红色。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

// Freezing a Freezable before it provides
// performance improvements if you don't
// intend on modifying it.
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

// If you need to modify a frozen brush,
// the Clone method can be used to
// create a modifiable copy.
SolidColorBrush myBrushClone = myBrush.Clone();

// Changing myBrushClone does not change
// the color of myButton, because its
// background is still set by myBrush.
myBrushClone.Color = Colors.Red;

// Replacing myBrush with myBrushClone
// makes the button change to red.
myButton.Background = myBrushClone;
```

### ① 备注

无论使用哪种克隆方法，动画都不会复制到新的 **Freezable**。

[Clone](#) 和 [CloneCurrentValue](#) 方法可生成 **Freezable** 的深层副本。如果 **Freezable** 包含其他已冻结的 **Freezable** 对象，它们也会被克隆并变为可修改。例如，如果克隆冻结的 [PathGeometry](#) 以使其可修改，则它包含的图形和段也会被复制并变为可修改。

## 创建自己的 **Freezable** 类

派生自 [Freezable](#) 的类具有以下特性。

- 特殊状态：只读（冻结）和可写状态。
- 线程安全：冻结的 **Freezable** 可以跨线程共享。
- 详细的更改通知：与其他 [DependencyObject](#) 不同，**Freezable** 对象在子属性值更改时提供更改通知。

- 轻松克隆：Freezable 类已经实现了几种生成深层克隆的方法。

Freezable 是一种 [DependencyObject](#)，因此使用依赖属性系统。类属性不必是依赖属性，但使用依赖属性可减少必须编写的代码量，因为 Freezable 类的设计考虑了依赖属性。有关依赖属性系统的详细信息，请参阅[依赖属性概述](#)。

每个 Freezable 子类都必须替代 [CreateInstanceCore](#) 方法。如果类对其所有数据使用依赖属性，则表示操作已完成。

如果类包含非依赖属性数据成员，则还必须替代以下方法：

- [CloneCore](#)
- [CloneCurrentValueCore](#)
- [GetAsFrozenCore](#)
- [GetCurrentValueAsFrozenCore](#)
- [FreezeCore](#)

访问和写入不是依赖属性的数据成员时，还必须遵守以下规则：

- 在读取非依赖属性数据成员的任何 API 的开头，调用 [ReadPreamble](#) 方法。
- 在写入非依赖属性数据成员的任何 API 的开头，调用 [WritePreamble](#) 方法。（在 API 中调用 [WritePreamble](#) 后，如果还读取了非依赖属性数据成员，则无需额外调用 [ReadPreamble](#)。）
- 在退出写入非依赖属性数据成员的方法之前调用 [WritePostscript](#) 方法。

如果类包含属于 [DependencyObject](#) 对象的非依赖属性数据成员，那么每次更改其中一个值时还必须调用 [OnFreezablePropertyChanged](#) 方法，即使将该成员设置为 `null` 也是如此。

### ① 备注

在替代的每个 Freezable 方法开始时都要调用基本实现，这一点很重要。

有关自定义 Freezable 类的示例，请参阅[自定义动画示例](#)。

## 另请参阅

- [Freezable](#)
- [自定义动画示例](#)

- 依赖项属性概述
- 自定义依赖属性

# Alignment、Margin 和 Padding 概述

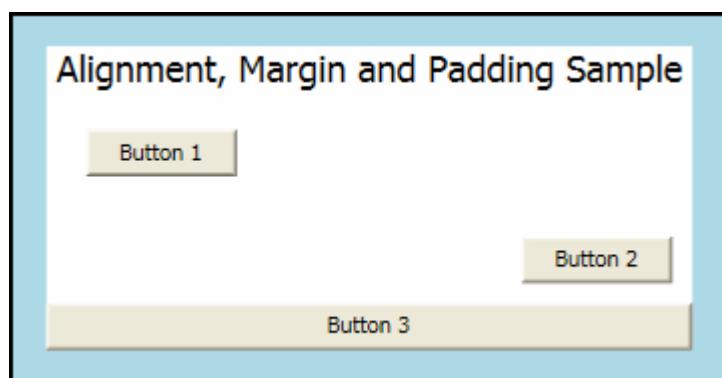
项目 · 2022/09/27

FrameworkElement 类公开一些用于精确定位子元素的属性。本主题讨论四个最重要的属性：[HorizontalAlignment](#)、[Margin](#)、[Padding](#) 和 [VerticalAlignment](#)。了解这些属性的作用非常重要，因为这些属性是控制元素在 Windows Presentation Foundation (WPF) 应用程序中的位置的基础。

## 元素定位简介

可使用 WPF 通过多种方式来定位元素。但是，获得理想的布局远不止仅选择正确的 Panel 元素。对定位的精细控制需要了解 [HorizontalAlignment](#)、[Margin](#)、[Padding](#) 和 [VerticalAlignment](#) 属性。

下图显示了一个采用若干定位属性的布局方案。



乍看上去，此图中的 Button 元素似乎是随意放置的。但是，其位置实际上是通过使用边距、对齐和填充加以精确控制的。

下面的示例描述如何创建上图中的布局。Border 元素封装父级 StackPanel，并且 Padding 值为 15 个与设备无关的像素。这解释了围绕子级 StackPanel 的窄 LightBlue 带。StackPanel 的子元素用于说明本主题中详述的各个定位属性。三 Button 个元素用于演示 Margin 和 HorizontalAlignment 属性。

C#

```
// Create the application's main Window.
mainWindow = new Window();
mainWindow.Title = "Margins, Padding and Alignment Sample";

// Add a Border
myBorder = new Border();
myBorder.Background = Brushes.LightBlue;
myBorder.BorderBrush = Brushes.Black;
```

```

myBorder.Padding = new Thickness(15);
myBorder.BorderThickness = new Thickness(2);

myStackPanel = new StackPanel();
myStackPanel.Background = Brushes.White;
myStackPanel.HorizontalAlignment = HorizontalAlignment.Center;
myStackPanel.VerticalAlignment = VerticalAlignment.Top;

TextBlock myTextBlock = new TextBlock();
myTextBlock.Margin = new Thickness(5, 0, 5, 0);
myTextBlock.FontSize = 18;
myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock.Text = "Alignment, Margin and Padding Sample";
Button myButton1 = new Button();
myButton1.HorizontalAlignment = HorizontalAlignment.Left;
myButton1.Margin = new Thickness(20);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.HorizontalAlignment = HorizontalAlignment.Right;
myButton2.Margin = new Thickness(10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.HorizontalAlignment = HorizontalAlignment.Stretch;
myButton3.Margin = new Thickness(0);
myButton3.Content = "Button 3";

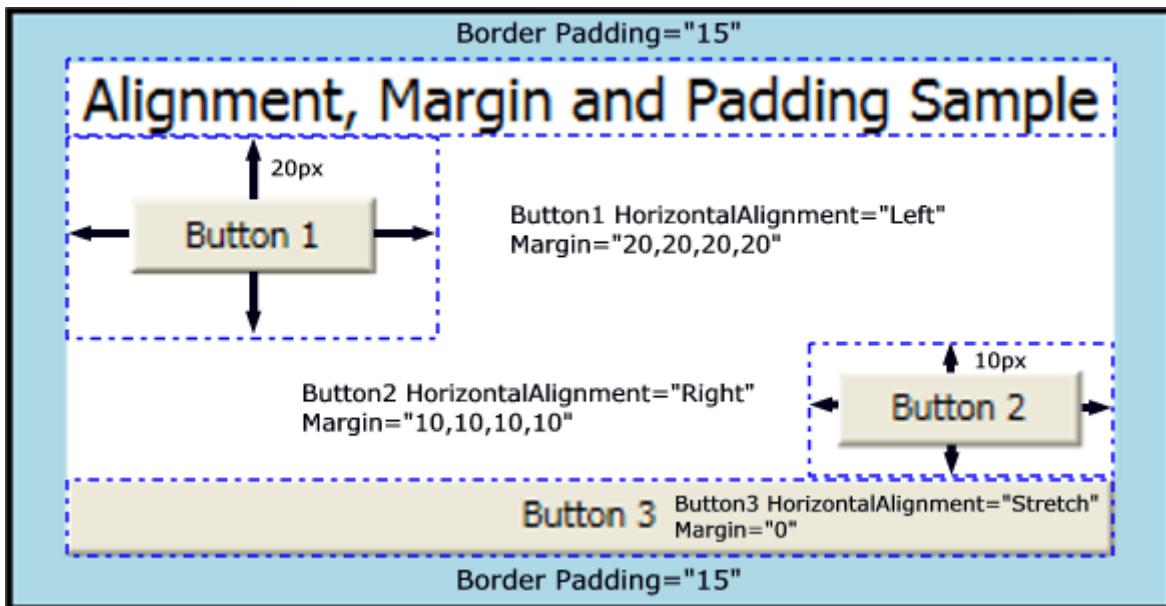
// Add child elements to the parent StackPanel.
myStackPanel.Children.Add(myTextBlock);
myStackPanel.Children.Add(myButton1);
myStackPanel.Children.Add(myButton2);
myStackPanel.Children.Add(myButton3);

// Add the StackPanel as the lone Child of the Border.
myBorder.Child = myStackPanel;

// Add the Border as the Content of the Parent Window Object.
mainWindow.Content = myBorder;
mainWindow.Show ();

```

通过下图可以仔细查看上例中使用的各个定位属性。本主题的后面各节更详细地介绍了如何使用各个定位属性。



## 了解 Alignment 属性

`HorizontalAlignment` 和 `VerticalAlignment` 属性描述子元素应当如何在父元素的已分配布局空间中定位。结合使用这些属性可精确定位子元素。例如，`DockPanel` 的子元素可以指定四种不同的水平对齐方式 (`Left`、`Right`、`Center` 或 `Stretch`) 来填补空闲空间。类似的值可用于垂直定位。

### ① 备注

元素上显式设置的 `Height` 和 `Width` 属性优先于 `Stretch` 属性值。尝试设置 `Height`、`Width` 和被忽略的 `Stretch` 请求中的 `Stretch` 结果的 `HorizontalAlignment` 值。

## HorizontalAlignment 属性

`HorizontalAlignment` 属性声明适用于子元素的水平对齐特征。下表列出了 `HorizontalAlignment` 属性的每个可能值。

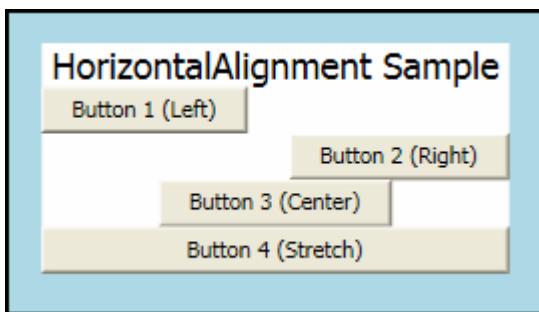
成员	说明
<code>Left</code>	子元素与父元素的已分配布局空间的左端对齐。
<code>Center</code>	子元素与父元素的已分配布局空间的中心对齐。
<code>Right</code>	子元素与父元素的已分配布局空间的右端对齐。
<code>Stretch</code> (默认值)	拉伸子元素以填充父元素的已分配布局空间。显式 <code>Width</code> 值和 <code>Height</code> 值优先。

下面的示例演示如何将 [HorizontalAlignment](#) 属性应用于 [Button](#) 元素。 显示每个特性值，以便更好地阐释各种呈现行为。

C#

```
Button myButton1 = new Button();
myButton1.HorizontalAlignment = HorizontalAlignment.Left;
myButton1.Content = "Button 1 (Left)";
Button myButton2 = new Button();
myButton2.HorizontalAlignment = HorizontalAlignment.Right;
myButton2.Content = "Button 2 (Right)";
Button myButton3 = new Button();
myButton3.HorizontalAlignment = HorizontalAlignment.Center;
myButton3.Content = "Button 3 (Center)";
Button myButton4 = new Button();
myButton4.HorizontalAlignment = HorizontalAlignment.Stretch;
myButton4.Content = "Button 4 (Stretch);
```

以上代码将产生与下图类似的布局。 每个 [HorizontalAlignment](#) 值的定位效果均在图中可见。



## VerticalAlignment 属性

[VerticalAlignment](#) 属性描述适用于子元素的垂直对齐特征。 下表列出了 [VerticalAlignment](#) 属性的各可能值。

成员	说明
Top	子元素与父元素的已分配布局空间的顶端对齐。
Center	子元素与父元素的已分配布局空间的中心对齐。
Bottom	子元素与父元素的已分配布局空间的底端对齐。
Stretch (默认值)	拉伸子元素以填充父元素的已分配布局空间。 显式 <a href="#">Width</a> 值和 <a href="#">Height</a> 值优先。

下面的示例演示如何将 [VerticalAlignment](#) 属性应用于 [Button](#) 元素。 显示每个特性值，以便更好地阐释各种呈现行为。 在此示例中，使用带有可见网格线的 [Grid](#) 元素作为父

项，以便更好地阐释各个属性值的布局行为。

C#

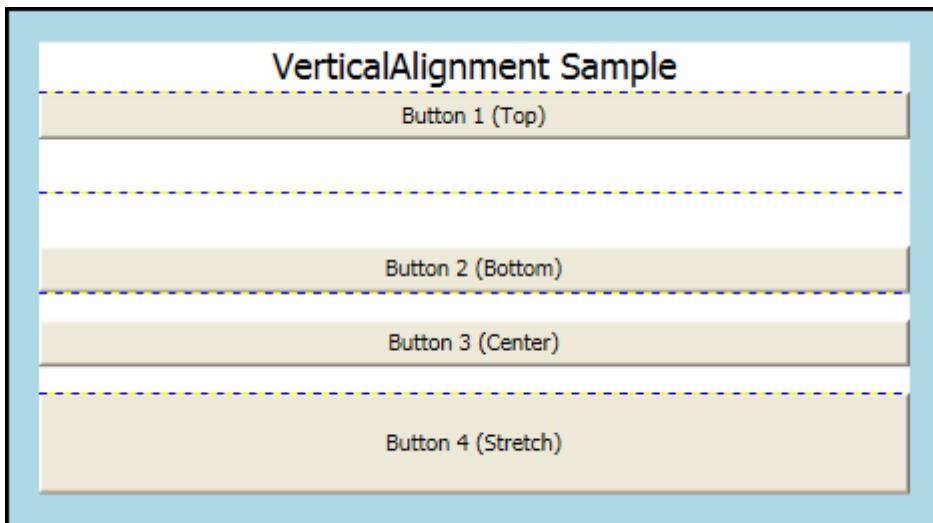
```
TextBlock myTextBlock = new TextBlock();
myTextBlock.FontSize = 18;
myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock.Text = "VerticalAlignment Sample";
Grid.SetRow(myTextBlock, 0);
Button myButton1 = new Button();
myButton1.VerticalAlignment = VerticalAlignment.Top;
myButton1.Content = "Button 1 (Top)";
Grid.SetRow(myButton1, 1);
Button myButton2 = new Button();
myButton2.VerticalAlignment = VerticalAlignment.Bottom;
myButton2.Content = "Button 2 (Bottom)";
Grid.SetRow(myButton2, 2);
Button myButton3 = new Button();
myButton3.VerticalAlignment = VerticalAlignment.Center;
myButton3.Content = "Button 3 (Center)";
Grid.SetRow(myButton3, 3);
Button myButton4 = new Button();
myButton4.VerticalAlignment = VerticalAlignment.Stretch;
myButton4.Content = "Button 4 (Stretch)";
Grid.SetRow(myButton4, 4);
```

XAML

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      WindowTitle="VerticalAlignment Sample">
    <Border Background="LightBlue" BorderBrush="Black" BorderThickness="2"
            Padding="15">
        <Grid Background="White" ShowGridLines="True">
            <Grid.RowDefinitions>
                <RowDefinition Height="25"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="50"/>
            </Grid.RowDefinitions>
            <TextBlock Grid.Row="0" Grid.Column="0" FontSize="18"
                      HorizontalAlignment="Center">VerticalAlignment Sample</TextBlock>
            <Button Grid.Row="1" Grid.Column="0"
                   VerticalAlignment="Top">Button 1 (Top)</Button>
            <Button Grid.Row="2" Grid.Column="0"
                   VerticalAlignment="Bottom">Button 2 (Bottom)</Button>
            <Button Grid.Row="3" Grid.Column="0"
                   VerticalAlignment="Center">Button 3 (Center)</Button>
            <Button Grid.Row="4" Grid.Column="0"
                   VerticalAlignment="Stretch">Button 4 (Stretch)</Button>
        </Grid>
    </Border>
</Page>
```

```
</Border>
</Page>
```

以上代码将产生与下图类似的布局。每个 `VerticalAlignment` 值的定位效果均在图中可见。



## 了解 Margin 属性

`Margin` 属性描述一个元素与其子元素或同级元素之间的距离。通过使用语法（例如 `Margin="20"`），`Margin` 值可以是统一的。利用此语法，向元素应用 20 个与设备无关的像素的统一 `Margin`。`Margin` 值还可以采用四个不同值的形式，每个值描述一个要应用到左侧、顶部、右侧和底部（采用该顺序）的不同边距，例如 `Margin="0,10,5,25"`。`Margin` 属性的正确使用可很好地控制元素及其相邻元素和子元素的呈现位置。

### ① 备注

非零边距在元素的 `ActualWidth` 和 `ActualHeight` 外应用空白。

下面的示例演示如何在一组 `Button` 元素的周围应用统一边距。`Button` 元素各个方向的间距都相同，并带有 10 像素的边距缓冲区。

C#

```
Button myButton7 = new Button();
myButton7.Margin = new Thickness(10);
myButton7.Content = "Button 7";
Button myButton8 = new Button();
myButton8.Margin = new Thickness(10);
myButton8.Content = "Button 8";
Button myButton9 = new Button();
myButton9.Margin = new Thickness(10);
myButton9.Content = "Button 9";
```

## XAML

```
<Button Margin="10">Button 7</Button>
<Button Margin="10">Button 8</Button>
<Button Margin="10">Button 9</Button>
```

在许多情况下，不适合使用统一边距。对于这些情况，可应用非统一间距。下面的示例演示如何将非统一边距间距应用于子元素。按此顺序描述边距：左端、顶端、右端和底端。

## C#

```
Button myButton1 = new Button();
myButton1.Margin = new Thickness(0, 10, 0, 10);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.Margin = new Thickness(0, 10, 0, 10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.Margin = new Thickness(0, 10, 0, 10);
```

## XAML

```
<Button Margin="0,10,0,10">Button 1</Button>
<Button Margin="0,10,0,10">Button 2</Button>
<Button Margin="0,10,0,10">Button 3</Button>
```

# 了解 Padding 属性

Padding 在大多数方面与 Margin 相似。Padding 属性只在少数类上公开，主要是为了方便：例如，[Block](#)、[Border](#)、[Control](#) 和 [TextBlock](#) 类公开 Padding 属性。Padding 属性将一个子元素的有效尺寸放大指定的 Thickness 值。

下面的例子展示了如何将 Padding 应用于父级 Border 元素。

## C#

```
myBorder = new Border();
myBorder.Background = Brushes.LightBlue;
myBorder.BorderBrush = Brushes.Black;
myBorder.BorderThickness = new Thickness(2);
myBorder.CornerRadius = new CornerRadius(45);
myBorder.Padding = new Thickness(25);
```

## XAML

```
<Border Background="LightBlue"
        BorderBrush="Black"
        BorderThickness="2"
        CornerRadius="45"
        Padding="25">
```

# 在应用程序中使用 Alignment、Margins 和 Padding

[HorizontalAlignment](#)、[Margin](#)、[Padding](#) 和 [VerticalAlignment](#) 提供创建复杂用户界面 (UI) 所需的定位控件。 可利用每个属性的作用来更改子元素定位，以便能够灵活地创建动态应用程序和用户体验。

下面的示例演示本主题中详述的各个概念。 此示例以本主题第一个示例中的基础结构为基础，添加了一个 [Grid](#) 元素作为第一个示例中 [Border](#) 的子项。 [Padding](#) 应用于父级 [Border](#) 元素。 [Grid](#) 用于在三个子 [StackPanel](#) 元素之间划分空间。 [Button](#) 元素再次用于显示 [Margin](#) 和 [HorizontalAlignment](#) 的各种效果。 [TextBlock](#) 元素添加到每个 [ColumnDefinition](#) 中，以更好地定义应用于每列中的 [Button](#) 元素的各种属性。

## C#

```
mainWindow = new Window();

myBorder = new Border();
myBorder.Background = Brushes.LightBlue;
myBorder.BorderBrush = Brushes.Black;
myBorder.BorderThickness = new Thickness(2);
myBorder.CornerRadius = new CornerRadius(45);
myBorder.Padding = new Thickness(25);

// Define the Grid.
myGrid = new Grid();
myGrid.Background = Brushes.White;
myGrid.ShowGridLines = true;

// Define the Columns.
ColumnDefinition myColDef1 = new ColumnDefinition();
myColDef1.Width = new GridLength(1, GridUnitType.Auto);
ColumnDefinition myColDef2 = new ColumnDefinition();
myColDef2.Width = new GridLength(1, GridUnitType.Star);
ColumnDefinition myColDef3 = new ColumnDefinition();
myColDef3.Width = new GridLength(1, GridUnitType.Auto);

// Add the ColumnDefinitions to the Grid.
myGrid.ColumnDefinitions.Add(myColDef1);
myGrid.ColumnDefinitions.Add(myColDef2);
```

```

myGrid.ColumnDefinitions.Add(myColDef3);

// Add the first child StackPanel.
StackPanel myStackPanel = new StackPanel();
myStackPanel.HorizontalAlignment = HorizontalAlignment.Left;
myStackPanel.VerticalAlignment = VerticalAlignment.Top;
Grid.SetColumn(myStackPanel, 0);
Grid.SetRow(myStackPanel, 0);
TextBlock myTextBlock1 = new TextBlock();
myTextBlock1.FontSize = 18;
myTextBlock1.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock1.Margin = new Thickness(0, 0, 0, 15);
myTextBlock1.Text = "StackPanel 1";
Button myButton1 = new Button();
myButton1.Margin = new Thickness(0, 10, 0, 10);
myButton1.Content = "Button 1";
Button myButton2 = new Button();
myButton2.Margin = new Thickness(0, 10, 0, 10);
myButton2.Content = "Button 2";
Button myButton3 = new Button();
myButton3.Margin = new Thickness(0, 10, 0, 10);
TextBlock myTextBlock2 = new TextBlock();
myTextBlock2.Text = @"ColumnDefinition.Width = ""Auto""";
TextBlock myTextBlock3 = new TextBlock();
myTextBlock3.Text = @"StackPanel.HorizontalAlignment = ""Left""";
TextBlock myTextBlock4 = new TextBlock();
myTextBlock4.Text = @"StackPanel.VerticalAlignment = ""Top""";
TextBlock myTextBlock5 = new TextBlock();
myTextBlock5.Text = @"StackPanel.Orientation = ""Vertical""";
TextBlock myTextBlock6 = new TextBlock();
myTextBlock6.Text = @"Button.Margin = "1,10,0,10""";
myStackPanel.Children.Add(myTextBlock1);
myStackPanel.Children.Add(myButton1);
myStackPanel.Children.Add(myButton2);
myStackPanel.Children.Add(myButton3);
myStackPanel.Children.Add(myTextBlock2);
myStackPanel.Children.Add(myTextBlock3);
myStackPanel.Children.Add(myTextBlock4);
myStackPanel.Children.Add(myTextBlock5);
myStackPanel.Children.Add(myTextBlock6);

// Add the second child StackPanel.
StackPanel myStackPanel2 = new StackPanel();
myStackPanel2.HorizontalAlignment = HorizontalAlignment.Stretch;
myStackPanel2.VerticalAlignment = VerticalAlignment.Top;
myStackPanel2.Orientation = Orientation.Vertical;
Grid.SetColumn(myStackPanel2, 1);
Grid.SetRow(myStackPanel2, 0);
TextBlock myTextBlock7 = new TextBlock();
myTextBlock7.FontSize = 18;
myTextBlock7.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock7.Margin = new Thickness(0, 0, 0, 15);
myTextBlock7.Text = "StackPanel 2";
Button myButton4 = new Button();
myButton4.Margin = new Thickness(10, 0, 10, 0);

```

```

myButton4.Content = "Button 4";
Button myButton5 = new Button();
myButton5.Margin = new Thickness(10, 0, 10, 0);
myButton5.Content = "Button 5";
Button myButton6 = new Button();
myButton6.Margin = new Thickness(10, 0, 10, 0);
myButton6.Content = "Button 6";
TextBlock myTextBlock8 = new TextBlock();
myTextBlock8.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock8.Text = @"ColumnDefinition.Width = ""*""";
TextBlock myTextBlock9 = new TextBlock();
myTextBlock9.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock9.Text = @"StackPanel.HorizontalAlignment = "Stretch""";
TextBlock myTextBlock10 = new TextBlock();
myTextBlock10.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock10.Text = @"StackPanel.VerticalAlignment = "Top""";
TextBlock myTextBlock11 = new TextBlock();
myTextBlock11.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock11.Text = @"StackPanel.Orientation = "Horizontal""";
TextBlock myTextBlock12 = new TextBlock();
myTextBlock12.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock12.Text = @"Button.Margin = "10,0,10,0""";
myStackPanel2.Children.Add(myTextBlock7);
myStackPanel2.Children.Add(myButton4);
myStackPanel2.Children.Add(myButton5);
myStackPanel2.Children.Add(myButton6);
myStackPanel2.Children.Add(myTextBlock8);
myStackPanel2.Children.Add(myTextBlock9);
myStackPanel2.Children.Add(myTextBlock10);
myStackPanel2.Children.Add(myTextBlock11);
myStackPanel2.Children.Add(myTextBlock12);

// Add the final child StackPanel.
StackPanel myStackPanel3 = new StackPanel();
myStackPanel3.HorizontalAlignment = HorizontalAlignment.Left;
myStackPanel3.VerticalAlignment = VerticalAlignment.Top;
Grid.SetColumn(myStackPanel3, 2);
Grid.SetRow(myStackPanel3, 0);
TextBlock myTextBlock13 = new TextBlock();
myTextBlock13.FontSize = 18;
myTextBlock13.HorizontalAlignment = HorizontalAlignment.Center;
myTextBlock13.Margin = new Thickness(0, 0, 0, 15);
myTextBlock13.Text = "StackPanel 3";
Button myButton7 = new Button();
myButton7.Margin = new Thickness(10);
myButton7.Content = "Button 7";
Button myButton8 = new Button();
myButton8.Margin = new Thickness(10);
myButton8.Content = "Button 8";
Button myButton9 = new Button();
myButton9.Margin = new Thickness(10);
myButton9.Content = "Button 9";
TextBlock myTextBlock14 = new TextBlock();
myTextBlock14.Text = @"ColumnDefinition.Width = "Auto""";
TextBlock myTextBlock15 = new TextBlock();

```

```

myTextBlock15.Text = @"StackPanel.HorizontalAlignment = ""Left""";
TextBlock myTextBlock16 = new TextBlock();
myTextBlock16.Text = @"StackPanel.VerticalAlignment = ""Top""";
TextBlock myTextBlock17 = new TextBlock();
myTextBlock17.Text = @"StackPanel.Orientation = ""Vertical""";
TextBlock myTextBlock18 = new TextBlock();
myTextBlock18.Text = @"Button.Margin = ""10""";
myStackPanel3.Children.Add(myTextBlock13);
myStackPanel3.Children.Add(myButton7);
myStackPanel3.Children.Add(myButton8);
myStackPanel3.Children.Add(myButton9);
myStackPanel3.Children.Add(myTextBlock14);
myStackPanel3.Children.Add(myTextBlock15);
myStackPanel3.Children.Add(myTextBlock16);
myStackPanel3.Children.Add(myTextBlock17);
myStackPanel3.Children.Add(myTextBlock18);

// Add child content to the parent Grid.
myGrid.Children.Add(myStackPanel);
myGrid.Children.Add(myStackPanel2);
myGrid.Children.Add(myStackPanel3);

// Add the Grid as the lone child of the Border.
myBorder.Child = myGrid;

// Add the Border to the Window as Content and show the Window.
mainWindow.Content = myBorder;
mainWindow.Title = "Margin, Padding, and Alignment Sample";
mainWindow.Show();

```

### XAML

```

<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
WindowTitle="Margins, Padding and Alignment Sample">
    <Border Background="LightBlue"
        BorderBrush="Black"
        BorderThickness="2"
        CornerRadius="45"
        Padding="25">
        <Grid Background="White" ShowGridLines="True">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto"/>
                <ColumnDefinition Width="*"/>
                <ColumnDefinition Width="Auto"/>
            </Grid.ColumnDefinitions>

            <StackPanel Grid.Column="0" Grid.Row="0" HorizontalAlignment="Left"
Name="StackPanel1" VerticalAlignment="Top">
                <TextBlock FontSize="18" HorizontalAlignment="Center"
Margin="0,0,0,15">StackPanel1</TextBlock>
                <Button Margin="0,10,0,10">Button 1</Button>
                <Button Margin="0,10,0,10">Button 2</Button>
                <Button Margin="0,10,0,10">Button 3</Button>
            </StackPanel1>
        </Grid>
    </Border>
</Page>

```

```

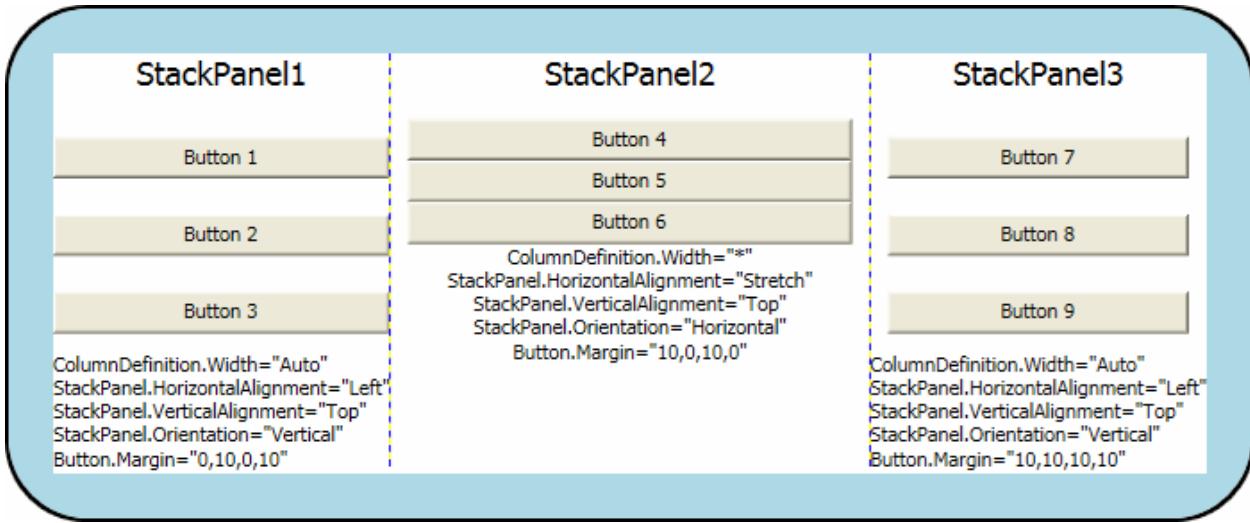
        <TextBlock>ColumnDefinition.Width="Auto"</TextBlock>
        <TextBlock>StackPanel.HorizontalAlignment="Left"</TextBlock>
        <TextBlock>StackPanel.VerticalAlignment="Top"</TextBlock>
        <TextBlock>StackPanel.Orientation="Vertical"</TextBlock>
        <TextBlock>Button.Margin="0,10,0,10"</TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="1" Grid.Row="0" HorizontalAlignment="Stretch"
Name="StackPanel2" VerticalAlignment="Top" Orientation="Vertical">
        <TextBlock FontSize="18" HorizontalAlignment="Center"
Margin="0,0,0,15">StackPanel2</TextBlock>
        <Button Margin="10,0,10,0">Button 4</Button>
        <Button Margin="10,0,10,0">Button 5</Button>
        <Button Margin="10,0,10,0">Button 6</Button>
        <TextBlock HorizontalAlignment="Center">ColumnDefinition.Width="*"
</TextBlock>
        <TextBlock
HorizontalAlignment="Center">StackPanel.HorizontalAlignment="Stretch"
</TextBlock>
        <TextBlock
HorizontalAlignment="Center">StackPanel.VerticalAlignment="Top"</TextBlock>
        <TextBlock
HorizontalAlignment="Center">StackPanel.Orientation="Horizontal"</TextBlock>
        <TextBlock HorizontalAlignment="Center">Button.Margin="10,0,10,0"
</TextBlock>
    </StackPanel>

    <StackPanel Grid.Column="2" Grid.Row="0" HorizontalAlignment="Left"
Name="StackPanel3" VerticalAlignment="Top">
        <TextBlock FontSize="18" HorizontalAlignment="Center"
Margin="0,0,0,15">StackPanel3</TextBlock>
        <Button Margin="10">Button 7</Button>
        <Button Margin="10">Button 8</Button>
        <Button Margin="10">Button 9</Button>
        <TextBlock>ColumnDefinition.Width="Auto"</TextBlock>
        <TextBlock>StackPanel.HorizontalAlignment="Left"</TextBlock>
        <TextBlock>StackPanel.VerticalAlignment="Top"</TextBlock>
        <TextBlock>StackPanel.Orientation="Vertical"</TextBlock>
        <TextBlock>Button.Margin="10"</TextBlock>
    </StackPanel>
</Grid>
</Border>
</Page>

```

编译后，前面的应用程序生成类似于下图的 UI。可在元素之间的间距中一目了然地看到各个属性值的效果，并且各列中元素的重要属性值显示在 `TextBlock` 元素内。



## 后续步骤

利用 [FrameworkElement](#) 类定义的定位属性，可精确控制元素在 WPF 应用程序内的位置。至此你已了解多种方法，可使用这些方法更好地通过 WPF 来定位元素。

我们还提供了一些附加资源，这些资源更详细地介绍 WPF 布局。[面板概述](#)主题包含有关各个 [Panel](#) 元素的更多详细信息。[主题演练：我的第一个 WPF 桌面应用程序](#)介绍使用布局元素来定位组件并将其操作绑定到数据源的高级方法。

## 另请参阅

- [FrameworkElement](#)
- [HorizontalAlignment](#)
- [VerticalAlignment](#)
- [Margin](#)
- [面板概述](#)
- [布局](#)
- [WPF 布局库示例](#)

# 基元素帮助主题

项目 • 2023/02/06

本部分中的主题介绍如何使用四个 WPF 基元素：[UIElement](#)、[ContentElement](#)、[FrameworkElement](#) 和 [FrameworkContentElement](#)。

## 本节内容

- [将 UIElement 设为透明或半透明](#)
- [对 FrameworkElement 的大小进行动画处理](#)
- [确定 Freezable 是否处于冻结状态](#)
- [处理 Loaded 事件](#)
- [设置元素和控件的边距](#)
- [将 Freezable 设为只读](#)
- [获取只读 Freezable 的可写副本](#)
- [水平或垂直翻转 UIElement](#)
- [使用 ThicknessConverter 对象](#)
- [处理 ContextMenuOpening 事件](#)

## 参考

- [UIElement](#)
- [ContentElement](#)
- [FrameworkElement](#)
- [FrameworkContentElement](#)

## 相关章节

- [基元素](#)

# 如何：使 UIElement 呈现为透明或半透明

项目 • 2023/02/06

此示例演示如何使 [UIElement](#) 呈现为透明或半透明。若要使元素呈现为透明或半透明，请设置其 [Opacity](#) 属性。值为 `0.0`，元素完全透明；值为 `1.0`，元素完全不透明。值为 `0.5`，元素 50% 不透明，依此类推。默认情况下，元素的 [Opacity](#) 设置为 `1.0`。

## 示例

以下示例将按钮的 [Opacity](#) 设置为 `0.25`，使其及其内容（在本例中为按钮的文本）25% 不透明。

XAML

```
<!-- Both the button and its text are made 25% opaque. -->
<Button Opacity="0.25">A Button</Button>
```

C#

```
//
// Both the button and its text are made 25% opaque.
//
Button myTwentyFivePercentOpaqueButton = new Button();
myTwentyFivePercentOpaqueButton.Opacity = new Double();
myTwentyFivePercentOpaqueButton.Opacity = 0.25;
myTwentyFivePercentOpaqueButton.Content = "A Button";
```

如果元素的内容有自己的 [Opacity](#) 设置，则这些值将与包含的元素 [Opacity](#) 相乘。

以下示例将按钮的 [Opacity](#) 设置为 `0.25`，并将按钮中包含的 [Image](#) 控件的 [Opacity](#) 设置为 `0.5`。结果，图像显示为 12.5% 不透明： $0.25 * 0.5 = 0.125$ 。

XAML

```
<!-- The image contained within this button has an effective
     opacity of 0.125 (0.25 * 0.5 = 0.125). -->
<Button Opacity="0.25">
    <StackPanel Orientation="Horizontal">
        <TextBlock VerticalAlignment="Center" Margin="10">A Button</TextBlock>
        <Image Source="sampleImages\berries.jpg" Width="50" Height="50"
              Opacity="0.5"/>
    </StackPanel>
</Button>
```

C#

```
//  
// The image contained within this button has an  
// effective opacity of 0.125 (0.25*0.5 = 0.125);  
  
Button myImageButton = new Button();  
myImageButton.Opacity = new Double();  
myImageButton.Opacity = 0.25;  
  
StackPanel myImageStackPanel = new StackPanel();  
myImageStackPanel.Orientation = Orientation.Horizontal;  
  
TextBlock myTextBlock = new TextBlock();  
myTextBlock.VerticalAlignment = VerticalAlignment.Center;  
myTextBlock.Margin = new Thickness(10);  
myTextBlock.Text = "A Button";  
myImageStackPanel.Children.Add(myTextBlock);  
  
Image myImage = new Image();  
BitmapImage myBitmapImage = new BitmapImage();  
myBitmapImage.BeginInit();  
myBitmapImage.UriSource = new  
Uri("sampleImages/berries.jpg", UriKind.Relative);  
myBitmapImage.EndInit();  
myImage.Source = myBitmapImage;  
ImageBrush myImageBrush = new ImageBrush(myBitmapImage);  
myImage.Width = 50;  
myImage.Height = 50;  
myImage.Opacity = 0.5;  
myImageStackPanel.Children.Add(myImage);  
myImageButton.Content = myImageStackPanel;
```

控制元素不透明度的另一种方法是设置绘制元素的 **Brush** 的不透明度。此方法使你能够有选择地更改元素部分的不透明度，并提供较使用元素的 **Opacity** 属性更为有利的性能优势。以下示例将用于绘制按钮的 **Background** 的 **SolidColorBrush** 的 **Opacity** 设置为 0.25。结果，画笔的背景 25% 不透明，但其内容（按钮的文本）仍为 100% 不透明。

XAML

```
<!-- This button's background is made 25% opaque, but its  
text remains 100% opaque. -->  
<Button>  
    <Button.Background>  
        <SolidColorBrush Color="Gray" Opacity="0.25" />  
    </Button.Background>  
    A Button  
</Button>
```

C#

```
//  
// This button's background is made 25% opaque,  
// but its text remains 100% opaque.  
//  
Button myOpaqueTextButton = new Button();  
SolidColorBrush mySolidColorBrush = new SolidColorBrush(Colors.Gray);  
mySolidColorBrush.Opacity = 0.25;  
myOpaqueTextButton.Background = mySolidColorBrush;  
myOpaqueTextButton.Content = "A Button";
```

还可以控制画笔中单个颜色的不透明度。有关颜色和画笔的详细信息，请参阅[使用纯色和渐变进行绘制概述](#)。有关如何对元素的不透明度进行动画处理的示例，请参阅[对元素或画笔的不透明度进行动画处理](#)。

# 如何：对 FrameworkElement 的大小进行动画处理

项目 • 2023/02/06

若要对 FrameworkElement 的大小进行动画处理，可以对其 Width 和 Height 属性进行动画处理或使用动画 ScaleTransform。

在下面的示例中，使用这两种方法对两个按钮的大小进行动画处理。对一个按钮的 Width 属性进行动画处理来重设该按钮大小，对应用于另一个按钮的 RenderTransform 属性的 ScaleTransform 进行动画处理来重设另一个按钮的大小。每个按钮都包含一些文本。一开始，这两个按钮中显示的文本相同，但随着按钮大小的调整，第二个按钮中的文本变得扭曲。

## 示例

XAML

```
<!-- AnimatingSizeExample.xaml
    This example shows two ways of animating the size
    of a framework element. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Microsoft.Samples.Animation.AnimatingSizeExample"
    WindowTitle="Animating Size Example">
<Canvas Width="650" Height="400">

    <Button Name="AnimatedWidthButton"
        Canvas.Left="20" Canvas.Top="20"
        Width="200" Height="150"
        BorderBrush="Red" BorderThickness="5">
        Click Me
    <Button.Triggers>

        <!-- Animate the button's Width property. -->
        <EventTrigger RoutedEvent="Button.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="AnimatedWidthButton"
                        Storyboard.TargetProperty="(Button.Width)"
                        To="500" Duration="0:0:10" AutoReverse="True"
                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
</Canvas>
</Page>
```

```
</Button>

<Button
    Canvas.Left="20" Canvas.Top="200"
    Width="200" Height="150"
    BorderBrush="Black" BorderThickness="3">
    Click Me
    <Button.RenderTransform>
        <ScaleTransform x:Name="MyAnimatedScaleTransform"
            ScaleX="1" ScaleY="1" />
    </Button.RenderTransform>
    <Button.Triggers>

        <!-- Animate the ScaleX property of a ScaleTransform
            applied to the button. -->
        <EventTrigger RoutedEvent="Button.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="MyAnimatedScaleTransform"
                        Storyboard.TargetProperty="(ScaleTransform.ScaleX)"
                        To="3.0" Duration="0:0:10" AutoReverse="True"
                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Button.Triggers>
</Button>
</Canvas>
</Page>
```

转换元素时，整个元素及其内容都会转换。直接更改元素的大小时（如第一个按钮的情况），元素的内容大小不会调整，除非该元素的大小和位置取决于其父元素的大小。

与直接对元素的 [Width](#) 和 [Height](#) 进行动画处理相比，通过将动画转换应用于元素的 [RenderTransform](#) 属性来对元素的大小进行动画处理可提供更好的性能，因为 [RenderTransform](#) 属性不会触发布局传递。

有关对属性进行动画处理的详细信息，请参阅[动画概述](#)。有关转换的详细信息，请参阅[转换概述](#)。

# 如何：确定 Freezable 是否处于冻结状态

项目 • 2023/02/06

此示例演示如何确定 [Freezable](#) 对象是否被冻结。如果尝试修改已冻结的 [Freezable](#) 对象，则会引发 [InvalidOperationException](#)。为了避免引发此异常，请使用 [Freezable](#) 对象的 [IsFrozen](#) 属性来确定其是否已冻结。

## 示例

以下示例冻结一个 [SolidColorBrush](#)，然后使用 [IsFrozen](#) 属性测试它，以确定它是否已冻结。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

if (myBrush.IsFrozen) // Evaluates to true.
{
    // If the brush is frozen, create a clone and
    // modify the clone.
    SolidColorBrush myBrushClone = myBrush.Clone();
    myBrushClone.Color = Colors.Red;
    myButton.Background = myBrushClone;
}
else
{
    // If the brush is not frozen,
    // it can be modified directly.
    myBrush.Color = Colors.Red;
}
```

有关 [Freezable](#) 对象的详细信息，请参阅 [Freezable 对象概述](#)。

## 另请参阅

- [Freezable](#)
- [IsFrozen](#)
- [Freezable 对象概述](#)
- [操作指南主题](#)

# 如何：处理 Loaded 事件

项目 • 2023/02/06

此示例演示如何处理 [FrameworkElement.Loaded](#) 事件，以及处理该事件的相应方案。 处理程序在页面加载时创建一个 [Button](#)。

## 示例

以下示例使用 Extensible Application Markup Language (XAML) 和代码隐藏文件。

XAML

```
<StackPanel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.FELoaded"
    Loaded="OnLoad"
    Name="root"
>
</StackPanel>
```

C#

```
void OnLoad(object sender, RoutedEventArgs e)
{
    Button b1 = new Button();
    b1.Content = "New Button";
    root.Children.Add(b1);
    b1.Height = 25;
    b1.Width = 200;
    b1.HorizontalAlignment = HorizontalAlignment.Left;
}
```

## 另请参阅

- [FrameworkElement](#)
- [对象生存期事件](#)
- [路由事件概述](#)
- [操作指南主题](#)

# 如何：设置元素和控件的边距

项目 • 2023/02/06

此示例介绍如何通过更改代码隐藏中边距的任何现有属性值来设置 Margin 属性。Margin 属性是 FrameworkElement 基元素的属性，因此由各种控件和其他元素继承。

此示例用 XAML 编写。代码隐藏同时以 C# 和 Microsoft Visual Basic 版本显示。

## 示例

XAML

```
<Button Click="OnClick" Margin="10" Name="btn1">  
Click To See Change!!</Button>
```

C#

```
void OnClick(object sender, RoutedEventArgs e)  
{  
    // Get the current value of the property.  
    Thickness marginThickness = btn1.Margin;  
    // If the current leftlength value of margin is set to 10 then change it  
    // to a new value.  
    // Otherwise change it back to 10.  
    if(marginThickness.Left == 10)  
    {  
        btn1.Margin = new Thickness(60);  
    } else {  
        btn1.Margin = new Thickness(10);  
    }  
}
```

# 如何：使 Freezable 成为只读

项目 • 2023/02/06

本例演示如何通过调用其 [Freeze](#) 方法使 [Freezable](#) 成为只读。

如果以下任一与对象有关的条件为 `true`，则无法冻结 [Freezable](#) 对象：

- 它具有动画属性或数据绑定属性。
- 它具有由动态资源设置的属性。 有关动态资源的详细信息，请参阅 [XAML 资源](#)。
- 它包含无法冻结的 [Freezable](#) 子对象。

如果 [Freezable](#) 对象的这些条件为 `false`，并且你不打算修改它，请考虑冻结它以获得性能优势。

## 示例

以下示例冻结一个 [SolidColorBrush](#)，它是 [Freezable](#) 对象的一种类型。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;
```

有关 [Freezable](#) 对象的详细信息，请参阅 [Freezable 对象概述](#)。

## 另请参阅

- [Freezable](#)
- [CanFreeze](#)
- [Freeze](#)
- [Freezable 对象概述](#)
- [操作指南主题](#)

# 如何：获取只读 Freezable 的可写副本

项目 • 2023/02/06

此示例演示如何使用 [Clone](#) 方法创建一个只读 [Freezable](#) 的可写副本。

将 [Freezable](#) 对象标记为只读（“冻结”）后，无法对其进行修改。但是，可以使用 [Clone](#) 方法创建冻结对象的可修改克隆。

## 示例

下面的示例创建冻结的 [SolidColorBrush](#) 对象的可修改克隆。

C#

```
Button myButton = new Button();
SolidColorBrush myBrush = new SolidColorBrush(Colors.Yellow);

// Freezing a Freezable before it provides
// performance improvements if you don't
// intend on modifying it.
if (myBrush.CanFreeze)
{
    // Makes the brush unmodifiable.
    myBrush.Freeze();
}

myButton.Background = myBrush;

// If you need to modify a frozen brush,
// the Clone method can be used to
// create a modifiable copy.
SolidColorBrush myBrushClone = myBrush.Clone();

// Changing myBrushClone does not change
// the color of myButton, because its
// background is still set by myBrush.
myBrushClone.Color = Colors.Red;

// Replacing myBrush with myBrushClone
// makes the button change to red.
myButton.Background = myBrushClone;
```

有关 [Freezable](#) 对象的详细信息，请参阅[可冻结对象概述](#)。

## 另请参阅

- [Freezable](#)
- [CloneCurrentValue](#)
- [Freezable 对象概述](#)
- [操作指南主题](#)

# 如何：水平或垂直翻转 UIElement

项目 • 2023/02/06

此示例演示如何使用 [ScaleTransform](#) 水平或垂直翻转 [UIElement](#)。在此示例中，通过将 [ScaleTransform](#) 应用于其 [RenderTransform](#) 属性来翻转 [Button](#) 控件（一种 [UIElement](#)）。

## 翻转按钮的插图

下图显示了要翻转的按钮。



按钮

要翻转的 [UIElement](#)

下面显示了创建按钮的代码。

XAML

```
<Button Content="Flip me!" Padding="5">
</Button>
```

## 水平翻转按钮的插图

若要水平翻转按钮，请创建一个 [ScaleTransform](#) 并将其 [ScaleX](#) 属性设置为 -1。将 [ScaleTransform](#) 应用到按钮的 [RenderTransform](#) 属性。

XAML

```
<Button Content="Flip me!" Padding="5">
<Button.RenderTransform>
  <ScaleTransform ScaleX="-1" />
</Button.RenderTransform>
</Button>
```



应用 ScaleTransform 后的按钮

## 在原位翻转按钮的插图

从上图中可以看出，按钮已翻转，但同时进行了移动。那是因为按钮是绕左上角翻转的。若要将按钮就地翻转，需要将 [ScaleTransform](#) 应用到其中心，而不是其角。将 [ScaleTransform](#) 应用于按钮中心的一种简单方法是将按钮的 [RenderTransformOrigin](#) 属性设置为“0.5, 0.5”。

XAML

```
<Button Content="Flip me!" Padding="5"
    RenderTransformOrigin="0.5,0.5">
    <Button.RenderTransform>
        <ScaleTransform ScaleX="-1" />
    </Button.RenderTransform>
</Button>
```



RenderTransformOrigin 为“0.5, 0.5”的按钮

## 垂直翻转按钮的插图

若要垂直翻转按钮，请设置 [ScaleTransform](#) 对象的 [ScaleY](#) 属性而不是其 [ScaleX](#) 属性。

XAML

```
<Button Content="Flip me!" Padding="5"
    RenderTransformOrigin="0.5,0.5">
    <Button.RenderTransform>
        <ScaleTransform ScaleY="-1" />
    </Button.RenderTransform>
</Button>
```

```
</Button.RenderTransform>  
</Button>
```



垂直翻转按钮

## 另请参阅

- [转换概述](#)

# 如何：使用 ThicknessConverter 对象

项目 • 2022/10/19

## 示例

此示例演示如何创建 [ThicknessConverter](#) 的实例并用它更改边框粗细。

该示例定义一个名为 `changeThickness` 的自定义方法；此方法首先按照单独的 Extensible Application Markup Language (XAML) 文件的定义将 [ListBoxItem](#) 的内容转换为 [Thickness](#) 的实例，然后将内容转换为 [String](#)。此方法将 [ListBoxItem](#) 传递给 [ThicknessConverter](#) 对象，该对象将 [ListBoxItem](#) 的 [Content](#) 转换为 [Thickness](#) 的实例。然后，此值作为 [Border](#) 的 [BorderThickness](#) 属性的值传回。

此示例不运行。

C#

```
private void changeThickness(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li = ((sender as ListBox).SelectedItem as ListBoxItem);
    ThicknessConverter myThicknessConverter = new ThicknessConverter();
    Thickness th1 =
(Thickness)myThicknessConverter.ConvertFromString(li.Content.ToString());
    border1.BorderThickness = th1;
    bThickness.Text = "Border.BorderThickness =" + li.Content.ToString();
}
```

## 另请参阅

- [Thickness](#)
- [ThicknessConverter](#)
- [Border](#)
- [如何：更改 Margin 属性](#)
- [如何：将 ListBoxItem 转换为新数据类型](#)
- [面板概述](#)

# 如何：处理 ContextMenuOpening 事件

项目 • 2023/02/06

可以在应用程序中处理 [ContextMenuOpening](#) 事件，以在显示之前调整现有的上下文菜单，或者通过在事件数据中将 [Handled](#) 属性设置为 `true` 来禁止以其他方式显示的菜单。在事件数据中将 [Handled](#) 设置为 `true` 的原因通常是为了用新的 [ContextMenu](#) 对象完全替换菜单，因此有时需要取消操作并启动新的打开。如果针对 [ContextMenuOpening](#) 事件编写处理程序，则应该注意 [ContextMenu](#) 控件与通常负责打开和定位控件上下文菜单的服务之间的计时问题。本主题说明了各种上下文菜单打开方案的一些代码技术，并说明了会出现计时问题的情况。

有多种方案可用于处理 [ContextMenuOpening](#) 事件：

- 在显示前调整菜单项。
- 在显示前替换整个菜单。
- 完全取消显示任何现有上下文菜单，且不再显示任何上下文菜单。

## 示例

### 在显示前调整菜单项

调整现有菜单项相当简单，而且可能也是最常见的方案。这样做可能是为了添加或减少上下文菜单选项，以响应应用程序中的当前状态信息或响应特定状态信息（可作为请求上下文菜单的对象上的属性）。

一般做法是获取事件源（即右键单击的特定控件），并从中获取 [ContextMenu](#) 属性。通常需要检查 [Items](#) 集合以查看菜单中已存在哪些上下文菜单项，然后在集合中添加或删除适当的新 [MenuItem](#) 项。

C#

```
void AddItemToCM(object sender, ContextMenuEventArgs e)
{
    //check if Item4 is already there, this will probably run more than once
    FrameworkElement fe = e.Source as FrameworkElement;
    ContextMenu cm = fe.ContextMenu;
    foreach (MenuItem mi in cm.Items)
    {
        if ((String)mi.Header == "Item4") return;
    }
    MenuItem mi4 = new MenuItem();
    mi4.Header = "Item4";
```

```
    fe.ContextMenu.Items.Add(mi4);
}
```

## 在显示前替换整个菜单

另一种方案则是替换整个上下文菜单。当然，也可以调整上述代码，删除现有上下文菜单的每个项，然后从 0 开始添加新项。但是，要替换上下文菜单中的所有项，一个更直观的方法是创建新的 [ContextMenu](#)，用项对其进行填充，然后将控件的 [FrameworkElement.ContextMenu](#) 属性设置为新的 [ContextMenu](#)。

以下是替换 [ContextMenu](#) 的简单处理程序代码。代码引用了自定义 [BuildMenu](#) 方法，将该方法分离出来是因为有多个示例处理程序对其进行调用。

C#

```
void HandlerForCM0(object sender, ContextMenuEventArgs e)
{
    FrameworkElement fe = e.Source as FrameworkElement;
    fe.ContextMenu = BuildMenu();
}
```

C#

```
ContextMenu BuildMenu()
{
    ContextMenu theMenu = new ContextMenu();
    MenuItem mia = new MenuItem();
    mia.Header = "Item1";
    MenuItem mib = new MenuItem();
    mib.Header = "Item2";
    MenuItem mic = new MenuItem();
    mic.Header = "Item3";
    theMenu.Items.Add(mia);
    theMenu.Items.Add(mib);
    theMenu.Items.Add(mic);
    return theMenu;
}
```

但是，如果对 [ContextMenuOpening](#) 使用此类型处理程序，那么当设置 [ContextMenu](#) 的对象没有预先具备上下文菜单时，可能会出现计时问题。用户右键单击控件时，即使现有的 [ContextMenu](#) 为空或 NULL，也会引发 [ContextMenuOpening](#)。但在这种情况下，在源元素上设置的任何新的 [ContextMenu](#) 都来不及显示。此外，如果用户碰巧第二次右键单击，那么这一次将显示新的 [ContextMenu](#)，该值不为 NULL，且处理程序将在处理程序第二次运行时适时替换并显示菜单。这表明有两种可能的解决方法：

- 确保 `ContextMenuOpening` 处理程序始终针对至少有一个占位符 `ContextMenu` 可用的控件运行，你计划将该占位符替换为处理程序代码。在这种情况下，仍然可以使用前面示例中显示的处理程序，但通常需要在初始标记中分配一个占位符 `ContextMenu`：

XAML

```
<StackPanel>
    <Rectangle Fill="Yellow" Width="200" Height="100"
    ContextMenuOpening="HandlerForCMO">
        <Rectangle.ContextMenu>
            <ContextMenu>
                <MenuItem>Initial menu; this will be replaced ...</MenuItem>
            </ContextMenu>
        </Rectangle.ContextMenu>
    </Rectangle>
    <TextBlock>Right-click the rectangle above, context menu gets
replaced</TextBlock>
</StackPanel>
```

- 根据一些初步逻辑，假设初始 `ContextMenu` 值可能为 `NULL`。可以检查 `ContextMenu` 是否为 `NULL`，或者在代码中使用标志来检查处理程序是否已至少运行一次。由于假设要显示 `ContextMenu`，因此处理程序随后会在事件数据中将 `Handled` 设置为 `true`。对于负责显示上下文菜单的 `ContextMenuService`，事件数据中的 `Handled` 值为 `true`，表示请求取消显示引发事件的上下文菜单/控件组合。

现在，你已取消显示可能出现的可疑上下文菜单，下一步是提供一个新的菜单，然后对其进行显示。设置新的上下文菜单与之前的处理程序基本相同：需要构建一个新的 `ContextMenu`，然后使用它设置控件源的 `FrameworkElement.ContextMenu` 属性。额外一步是，现在必须强制显示上下文菜单，因为你取消了第一次尝试。要强制显示，请在处理程序中将 `Popup.IsOpen` 属性设置为 `true`。执行此操作时要小心，因为在处理程序中打开上下文菜单会再次引发 `ContextMenuOpening` 事件。如果重新进入处理程序，则处理程序会无限递归。因此，在从 `ContextMenuOpening` 事件处理程序中打开上下文菜单时，始终需要检查是否为 `null` 或使用标志。

## 取消显示任何现有上下文菜单并不再显示上下文菜单

最后一种方案（即编写一个完全取消显示菜单的处理程序）并不常见。如果给定的控件不应该显示上下文菜单，则比起仅在用户请求时取消显示菜单，可能有更合适的方法来确保不显示。但是，如果想使用处理程序来取消显示上下文菜单并且不显示任何内容，那么处理程序只需在事件数据中将 `Handled` 设置为 `true`。负责显示上下文菜单的

[ContextMenuService](#) 将检查其在控件上引发的事件的事件数据。如果事件在路由中的任意位置标记为 [Handled](#)，则启动事件的上下文菜单打开操作将被取消。

C#

```
void HandlerForCM02(object sender, ContextMenuEventArgs e)
{
    if (!FlagForCustomContextMenu)
    {
        e.Handled = true; //need to suppress empty menu
        FrameworkElement fe = e.Source as FrameworkElement;
        fe.ContextMenu = BuildMenu();
        FlagForCustomContextMenu = true;
        fe.ContextMenu.IsOpen = true;
    }
}
```

## 另请参阅

- [ContextMenu](#)
- [FrameworkElement.ContextMenu](#)
- [基元素概述](#)
- [ContextMenu 概述](#)

# 元素树和序列化

项目 · 2023/02/06

WPF 编程元素彼此之间通常以某种形式的树关系存在。例如，XAML 中创建的应用程序 UI 可概念化为一个对象树。可进一步将元素树分为两个离散但有时会并行的树：逻辑树和可视化树。WPF 中的序列化涉及保存这两个树和应用程序的状态并将状态写入文件（可能以 XAML 形式）。

## 本节内容

[WPF 中的树](#)

[XamlWriter.Save 的序列化限制](#)

[不在对象树中的对象元素的初始化](#)

[操作指南主题](#)

## 参考

[System.Windows.Markup](#)

[LogicalTreeHelper](#)

[VisualTreeHelper](#)

## 相关章节

[WPF 体系结构](#)

[WPF 中的 XAML](#)

[基元素](#)

[属性](#)

[事件](#)

[输入](#)

[资源](#)

[样式设置和模板化](#)

[线程模型](#)

# WPF 中的树

项目 • 2022/09/27

在许多技术中，元素和组件都按树结构的形式组织。在这种结构中，开发人员可以直接操作树中的对象节点来影响应用程序的绘制或行为。Windows Presentation Foundation (WPF) 也使用了若干树结构形式来定义程序元素之间的关系。多数情况下，在概念层面考虑对象树形式时，WPF 开发人员会用代码创建应用程序，或用 XAML 定义应用程序的组成部分，但他们调用具体的 API 或使用特定的标记来执行此操作，而不是像在 XML DOM 中那样，使用某些常规对象树操作 API。WPF 公开提供树形式视图的两个帮助程序类：[LogicalTreeHelper](#) 和 [VisualTreeHelper](#)。WPF 文档中还使用了“可视化树”和“逻辑树”两个术语，它们有助于理解某些关键 WPF 功能的行为。本主题定义可视化树和逻辑树的含义，讨论这些树与总体对象树概念之间的关系，并介绍 [LogicalTreeHelper](#) 和 [VisualTreeHelper](#)。

## WPF 中的树

WPF 中，最完整的树结构是对象树。如果在 WPF 子系统中定义应用程序页，并影响在标记或代码中做出的选择。

尽管你并不会总是直接操作逻辑树或可视化树，但理解它们之间的关系有助于你从技术角度了解 WPF。若要理解 WPF 中属性继承和事件路由的工作原理，将 WPF 视为某种树形式也相当重要。

### ① 备注

因为对象树更像是概念，而不像是实际 API，所以还可以将此概念视为对象图。实际上，在运行时，对象之间的某些关系不能由树形式表示。尽管如此，树形式的相关性还是很强，尤其是对于 XAML 定义的 UI。因此，大多数 WPF 文档在引用这个常见概念时，仍使用术语“对象树”。

## 逻辑树

在 WPF 中，通过为支持 UI 元素的对象设置属性，可以向这些 UI 元素添加内容。例如，通过操作 [ListBox](#) 控件的 [Items](#) 属性，可以将项添加到该控件。通过这种方法，可以将项放入用作 [Items](#) 属性值的 [ItemCollection](#) 中。同样，通过操作 [DockPanel](#) 的 [Children](#) 属性值，可以将对象添加到该控件中。这里，你将对象添加到 [UIElementCollection](#) 中。有关代码示例，请参阅[如何：动态添加元素](#)。

在 Extensible Application Markup Language (XAML) 中，当在 [ListBox](#) 中放置列表项或在 [DockPanel](#) 中放置控件或其他 UI 元素时，还会显式或隐式使用 [Items](#) 和 [Children](#) 属性，如下例所示。

XAML

```
<DockPanel
    Name="ParentElement"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <!--implicit: <DockPanel.Children>-->
    <ListBox DockPanel.Dock="Top">
        <!--implicit: <ListBox.Items>-->
        <ListBoxItem>
            <TextBlock>Dog</TextBlock>
        </ListBoxItem>
        <ListBoxItem>
            <TextBlock>Cat</TextBlock>
        </ListBoxItem>
        <ListBoxItem>
            <TextBlock>Fish</TextBlock>
        </ListBoxItem>
        <!--implicit: </ListBox.Items>-->
    </ListBox>
    <Button Height="20" Width="100" DockPanel.Dock="Top">Buy a Pet</Button>
    <!--implicit: </DockPanel.Children>-->
</DockPanel>
```

如果此 XAML 是作为文档对象模型下的 XML 进行处理，且已包含作为隐式项禁止注释的标记（可能是合法的），生成的 XML DOM 树已包含 [ListBox.Items](#) 的元素以及其他隐式项。但是，读取标记和写入对象时，XAML 不会这样处理，生成的对象图不包含 [ListBox.Items](#)。不过，它确实有一个名为 [Items](#) 的 [ListBox](#) 属性，其中包含一个 [ItemCollection](#)，并且在处理 [ListBox](#) XAML 时，[ItemCollection](#) 会进行初始化，但是为空。然后，作为 [ListBox](#) 的内容存在的每个子对象元素，都将通过对 [ItemCollection.Add](#) 的分析程序调用，添加到 [ItemCollection](#) 中。此示例将 XAML 处理成对象树，目前这似乎表明所创建的对象树基本上是逻辑树。

不过，即使不考虑 XAML 隐式语法项，该逻辑树也不是应用程序 UI 在运行时存在的整个对象图。主要原因是视觉对象和模板。例如，考虑 [Button](#)。逻辑树报告 [Button](#) 对象及其字符串 [Content](#)。但在运行时对象树中，此按钮还有更多内容。具体而言，该按钮在屏幕上仅显示为现在这样，是因为应用了特定的 [Button](#) 控件模板。逻辑树不会报告来自所应用模板的视觉对象，例如可视化按钮周围由模板定义的深灰色的 [Border](#)。即使在运行时查看逻辑树（例如，处理来自可见 UI 的输入事件，然后读取逻辑树），也是如此。若要查找模板视觉对象，需要改为检查可视化树。

有关 XAML 语法如何映射到所创建的对象图，以及 XAML 中隐式语法的详细信息，请参阅 [XAML 语法详述](#) 或 [WPF 中的 XAML](#)。

## 逻辑树用途

借助逻辑树，内容模型可以方便地循环访问其可能的子对象，从而实现扩展。此外，逻辑树还为某些通知提供框架，例如在加载逻辑树中的所有对象时。基本上，逻辑树是框架级别的近似运行时对象图（排除了视觉对象），但其足以用于对你自己的运行时应用程序组合执行多种查询操作。

此外，静态和动态资源引用具有相同的解析过程：针对最初发出请求的对象，沿逻辑树向上查找 [Resources](#) 集合，然后沿逻辑树继续向上，检查每一个 [FrameworkElement](#) 或 [FrameworkContentElement](#)，以查找另一个包含 [ResourceDictionary](#)（可能包含该键）的 [Resources](#) 值。当同时存在逻辑树和可视化树时，将使用逻辑树进行资源查找。有关资源字典和查找的详细信息，请参见 [XAML 资源](#)。

## 逻辑树的构成

逻辑树在 WPF 框架级别定义。这意味着，与逻辑树操作关系最密切的 WPF 基元素是 [FrameworkElement](#) 或 [FrameworkContentElement](#)。但是你会发现，如果实际使用 [LogicalTreeHelper API](#)，则逻辑树有时会包含既不是 [FrameworkElement](#)，也不是 [FrameworkContentElement](#) 的节点。例如，逻辑树会报告 [TextBlock](#) 的 [Text](#) 值，该值是一个字符串。

## 替代逻辑树

经验丰富的控件作者会通过替代若干 API（用于定义常规对象或内容模型如何在逻辑树中添加或删除对象）来替代逻辑树。有关如何替代逻辑树的示例，请参阅[替代逻辑树](#)。

## 属性值继承

属性值继承通过混合树操作。包含用于启用属性继承的 [Inherits](#) 属性的实际元数据是 WPF 框架级别 [FrameworkPropertyMetadata](#) 类。因此，保留原始值的父对象和继承该值的子对象都必须是 [FrameworkElement](#) 或 [FrameworkContentElement](#)，且都必须属于某个逻辑树。但是，对于支持属性继承的现有 WPF 属性，属性值的继承可通过逻辑树中没有的中介对象永久存在。这主要适用于以下情况：让模板元素使用在应用了模板的实例上设置的任何继承属性值，或者使用在更高级别的页级构成（因此在逻辑树中也位于更高位置）中设置的任何继承属性值。为了使属性值的继承在这两种情况下保持一致，继承属性必须注册为附加属性。如果要定义具有属性继承行为的自定义依赖属性，则应采用这

种模式。无法通过帮助器类实用工具方法完全预测属性继承确切使用的树，即使在运行时也是如此。有关详细信息，请参阅[属性值继承](#)。

## 可视化树

WPF 中除了逻辑树的概念，还存在可视化树的概念。可视化树描述由 [Visual](#) 基类表示的可视化对象的结构。为控件编写模板时，将定义或重新定义适用于该控件的可视化树。对于出于性能和优化考虑需要对绘图进行较低级别控制的开发人员来说，他们也会对可视化树感兴趣。在传统 WPF 应用程序编程中，可视化树的一个应用是：路由事件的事件路由大多遍历可视化树而非逻辑树。路由事件行为的这种微妙之处可能不会很明显，除非你是控件作者。通过可视化树对事件进行路由可使控件在可视化级别实现组合以处理事件或创建事件资源库。

## 树、内容元素和内容宿主

内容元素（从 [ContentElement](#) 派生的类）不属于可视化树；内容元素不从 [Visual](#) 继承并且没有可视化表示形式。若要完全显示在 UI 中，[ContentElement](#) 必须承载在既是 [Visual](#) 又是逻辑树参与者的[内容宿主](#)中。这样的对象通常是 [FrameworkElement](#)。从概念上讲，内容宿主有些类似于内容的“浏览器”，它选择在该宿主控制的屏幕区域中显示内容的方式。承载内容时，可以使内容成为通常与可视化树关联的某些树进程的参与者。通常，[FrameworkElement](#) 宿主类包括实现代码，该代码用于通过内容逻辑树的子节点将任何已承载的 [ContentElement](#) 添加到事件路由，即使承载内容不属于实际可视化树也是如此。这样做是必要的，以便 [ContentElement](#) 可以获取路由到非本身的任何元素的路由事件。

## 树遍历

[LogicalTreeHelper](#) 类提供用于逻辑树遍历的 [GetChildren](#)、[GetParent](#) 和 [FindLogicalNode](#) 方法。在大多数情况下，不需要遍历现有控件的逻辑树，因为这些控件几乎总是将其逻辑子元素公开为一个专用集合属性，这种属性支持集合访问，如 [Add](#)、索引器等等。如果控件作者选择不从预期控件模式（例如已定义了集合属性的 [ItemsControl](#) 或 [Panel](#)）派生或希望提供其自己的集合属性支持，则树遍历是他们使用的一种主要方案。

可视化树还支持用于可视化树遍历的帮助器类 [VisualTreeHelper](#)。无法通过特定于控件的属性方便地公开可视化树，因此，如果你的编程方案必须遍历可视化树，建议使用 [VisualTreeHelper](#) 类。有关详细信息，请参阅 [WPF 图形呈现概述](#)。

### ① 备注

有时有必要检查所应用模板的可视化树。执行此操作时应谨慎。即便是遍历定义有模板的控件的可视化树，该控件的使用者仍可以通过设置实例的 `Template` 属性随时更改模板，甚至最终用户也可以通过更改系统主题来影响所应用的模板。

## “树”形式路由事件的路由

如前所述，对于任何给定的路由事件，其路由都沿着一条预定的树路径进行，这棵树是可视化树和逻辑树表示形式的混合体。事件路由可在树中向上或向下进行，具体取决于该事件是隧道路由事件还是浮升路由事件。事件路由概念没有直接支持的帮助器类（此类可用于独立于引发实际路由的事件，遍历事件）。存在表示路由的类 `EventRoute`，但该类的方法通常仅供内部使用。

## 资源字典和树

对页中定义的所有 `Resources` 进行资源字典查找时，基本上遍历逻辑树。逻辑树之外的对象可以引用键控资源，但资源查找顺序将从该对象与逻辑树的连接点开始。在 WPF 中，只有逻辑树节点可以有包含 `ResourceDictionary` 的 `Resources` 属性，因此通过遍历可视化树从 `ResourceDictionary` 中查找键控资源并无益处。

但是，资源查找也可以超出直接逻辑树。对于应用程序标记，资源查找可向前继续进行到应用程序级资源字典，然后再作为静态属性或键进行引用的主题支持和系统值。如果资源引用是动态的，则主题本身也可以引用主题逻辑树之外的系统值。有关资源字典和查找逻辑的详细信息，请参阅 [XAML 资源](#)。

## 请参阅

- [输入概述](#)
- [WPF 图形呈现疑难解答](#)
- [路由事件概述](#)
- [不在对象树中的对象元素的初始化](#)
- [WPF 体系结构](#)

# XamlWriter.Save 的序列化限制

项目 • 2023/02/06

API [Save](#) 可用于将 Windows Presentation Foundation (WPF) 应用程序的内容序列化为 Extensible Application Markup Language (XAML) 文件。但是，对于所序列化的内容有一些显著限制。本主题对这些限制和某些一般注意事项进行了介绍。

## 运行时、非设计时表示形式

对于通过调用 [Save](#) 进行序列化的内容，基本原则是：其结果为在运行时生成序列化对象的表示形式。在将 XAML 加载为内存中对象时，原始 XAML 文件的许多设计时属性可能已经优化或丢失，而且在调用 [Save](#) 进行序列化时未保留这些属性。序列化的结果是应用程序的结构化逻辑树的有效表示形式，但并不一定是生成该树的原始 XAML 的有效表示形式。这些问题导致将 [Save](#) 序列化用作大型 XAML 设计图面的一部分变得极为困难。

## 序列化是自包含的

[Save](#) 的序列化输出是自包含的；序列化所有内容都包含在单个 XAML 页面中，具有单个根元素，而且不存在 URI 以外的外部引用。例如，如果页面从应用程序资源引用了资源，则这些资源看上去如同正在进行序列化的页面的一个组件。

## 取消引用扩展引用

由各种标记扩展格式（如 `StaticResource` 或 `Binding`）对对象进行的公共引用将会由序列化进程取消引用。当应用程序运行时创建内存中对象时，已对这些公共引用取消引用，且 [Save](#) 逻辑不会重新访问原始的 XAML 来将这些引用还原到序列化的输出。这样可能会将任何数据绑定的或资源获得的值冻结为运行时表示形式最后使用的值，并且只能有限地或间接地区别这样的值与任何其他在本地设置的值。由于图像存在于项目中，因此图像也会序列化为图像的对象引用（而不是原始的源引用），从而会丢失最初引用的文件名或 URI。即使是在同一页面内声明的资源，也会序列化到引用点内，而不是保留为资源集合的键。

## 不保留事件处理

当对通过 XAML 添加的事件处理程序进行序列化后，不会保留这些事件处理程序。不具有代码隐藏功能（并且也不具有相关的 `x:Code` 机制）的 XAML 无法对运行时过程逻辑进行序列化。因为序列化是自包含的且限于逻辑树，所以不存在用于存储事件处理程序的

设施。因此，会从输出 XAML 中删除事件处理程序特性（特性本身和用于命名处理程序的字符串值）。

## XAMLWriter.Save 实用方案

虽然此处列出了较多限制，但仍然存在几个适合使用 [Save](#) 进行序列化的方案。

- 向量或图形输出：所呈现的区域的输出可用于在重新加载时重新生成相同的向量或图形。
- 格式文本和流文档：输出中会保留文本以及文本内的所有元素格式和元素所含内容。这对类似于剪贴板功能的机制可能非常有用。
- 保留业务对象数据：如果已在自定义元素中存储数据（如 XML 数据），只要业务对象遵循基本 XAML 规则（如为按引用属性值提供自定义构造函数和转换），这些业务对象就可以通过序列化永久保留。

# 不在对象树中的对象元素的初始化

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 初始化时某些方面会被推迟，在通常依赖连接到逻辑树或可视化树的元素的进程中执行。本主题介绍了针对未连接到两种树之一的元素，将其初始化可能需要的步骤。

## 元素和逻辑树

在代码中创建 Windows Presentation Foundation (WPF) 类的实例时，应该注意在调用类构造函数时所执行的代码中，需要有意地不包含 Windows Presentation Foundation (WPF) 类的对象初始化的几个方面。特别是对于控件类，该控件的大部分可视化表示形式都不是由构造函数定义的。而是由控件的模板定义。模板可能来自各种源，但最常见的情况是来自于主题样式。模板实际上是晚期绑定的；在控件已准备好布局之后，才会将所需模板附加到相关控件。并且控件只有在已附加到连接到根级别的呈现图面的逻辑树时，才能准备好应用布局。正是该根级别元素启动逻辑树中定义的所有子元素的呈现。

可视化树也参与此过程。通过模板成为可视化树一部分的元素也是在连接后才完全实例化的。

此行为的结果是依赖某个元素已完成的可视化特征的某些操作需要额外的步骤。例如，如果你试图获取一个已构造但尚未附加到树中的类的可视化特征，就需要额外的步骤。例如，如果想要在 [RenderTargetBitmap](#) 上调用 [Render](#)，并且要传递的视觉对象是一个未连接到树的元素，则直到完成了额外的初始化步骤，该元素在可视化方面才完整。

## 使用 BeginInit 和 EndInit 初始化元素

WPF 中的各种类实现 [ISupportInitialize](#) 接口。使用接口的 [BeginInit](#) 和 [EndInit](#) 方法来表示代码中包含初始化步骤（如设置影响呈现的属性值）的区域。按顺序调用 [EndInit](#) 之后，布局系统就可以处理元素并开始查找隐式样式。

如果要在其上设置属性的元素是 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生类，则可以调用 [BeginInit](#) 和 [EndInit](#) 的类版本，而不是强制转换为 [ISupportInitialize](#)。

## 代码示例

下面的示例是一个控制台应用程序的示例代码，该应用程序使用呈现 API 和宽松 XAML 文件的 [XamlReader.Load\(Stream\)](#) 来说明如何将 [BeginInit](#) 和 [EndInit](#) 正确放置在调整影响呈现的属性的其他 API 调用周围。

该示例仅演示主要函数。 函数 `Rasterize` 和 `Save` ( 未显示 ) 是负责图像处理和 IO 的实用工具函数。

C#

```
[STAThread]
static void Main(string[] args)
{
    UIElement e;
    string file = Directory.GetCurrentDirectory() + "\\starting.xaml";
    using (Stream stream = File.Open(file, FileMode.Open))
    {
        // loading files from current directory, project settings take care
        // of copying the file
        ParserContext pc = new ParserContext();
        pc.BaseUri = new Uri(file, UriKind.Absolute);
        e = (UIElement)XamlReader.Load(stream, pc);
    }

    Size paperSize = new Size(8.5 * 96, 11 * 96);
    e.Measure(paperSize);
    e.Arrange(new Rect(paperSize));
    e.UpdateLayout();

    /*
     * Render effect at normal dpi, indicator is the original RED
     rectangle
    */
    RenderTargetBitmap image1 = Rasterize(e, paperSize.Width,
    paperSize.Height, 96, 96);
    Save(image1, "render1.png");

    Button b = new Button();
    b.BeginInit();
    b.Background = Brushes.Blue;
    b.Width = b.Height = 200;
    b.EndInit();
    b.Measure(paperSize);
    b.Arrange(new Rect(paperSize));
    b.UpdateLayout();

    // now render the altered version, with the element built up and
    // initialized

    RenderTargetBitmap image2 = Rasterize(b, paperSize.Width,
    paperSize.Height, 96, 96);
    Save(image2, "render2.png");
}
```

## 另请参阅

- WPF 中的树
- WPF 图形呈现疑难解答
- WPF 中的 XAML

# 元素树和序列化帮助主题

项目 • 2023/02/06

本部分中的主题介绍如何使用 WPF 元素树。

## 本节内容

[按名称查找元素](#)

[重写逻辑树](#)

## 参考

[LogicalTreeHelper](#)

[VisualTreeHelper](#)

[System.Windows.Markup](#)

## 相关章节

# 如何：按名称查找元素

项目 • 2023/02/06

此示例介绍如何使用 [FindName](#) 方法根据元素的 [Name](#) 值查找元素。

## 示例

在此示例中，按元素名称查找特定元素的方法将作为按钮的事件处理程序写入。

`stackPanel` 是正在搜索的根 [FrameworkElement](#) 的 [Name](#)，然后示例方法通过将其强制转换为 [TextBlock](#) 并更改 [TextBlock](#) 可见 UI 属性之一来直观地指示所找到的元素。

C#

```
void Find(object sender, RoutedEventArgs e)
{
    object wantedNode = stackPanel.FindName("dog");
    if (wantedNode is TextBlock)
    {
        // Following executed if Text element was found.
        TextBlock wantedChild = wantedNode as TextBlock;
        wantedChild.Foreground = Brushes.Blue;
    }
}
```

# 如何：重写逻辑树

项目 • 2023/02/06

尽管在大多数情况下不必重写逻辑树，但资深的控件创作者可选择执行此操作。

## 示例

本示例介绍如何创建 `StackPanel` 子类以重写逻辑树，在本示例中重写逻辑树是为了强制该面板只能具有并且仅仅呈现一个子元素的行为。这不一定是实际需要的行为，但在这里进行演示是为了举例说明重写元素的正常逻辑树的情况。

C#

```
public class SingletonPanel : StackPanel
{
    //private UIElementCollection _children;
    private FrameworkElement _child;

    public SingletonPanel()
    {
    }

    public FrameworkElement SingleChild
    {

        get { return _child; }
        set
        {
            if (value == null)
            {
                RemoveLogicalChild(_child);
            }
            else
            {
                if (_child == null)
                {
                    _child = value;
                }
                else
                {
                    // raise an exception?
                    MessageBox.Show("Needs to be a single element");
                }
            }
        }
    }

    public void SetSingleChild(object child)
    {
        this.AddLogicalChild(child);
    }
}
```

```
}

public new void AddLogicalChild(object child)
{
    _child = (FrameworkElement)child;
    if (this.Children.Count == 1)
    {
        this.RemoveLogicalChild(this.Children[0]);
        this.Children.Add((UIElement)child);
    }
    else
    {
        this.Children.Add((UIElement)child);
    }
}

public new void RemoveLogicalChild(object child)
{
    _child = null;
    this.Children.Clear();
}
protected override IEnumerator LogicalChildren
{
    get
    {
        // cheat, make a list with one member and return the enumerator
        ArrayList _list = new ArrayList();
        _list.Add(_child);
        return (IEnumerator)_list.GetEnumerator();
    }
}
}
```

有关逻辑树的详细信息，请参阅 [WPF 中的树](#)。

# 属性 (WPF)

项目 · 2023/02/06

Windows Presentation Foundation (WPF) 提供一组服务，这些服务可用于扩展公共语言运行时 (CLR) 属性的功能。这些服务通常统称为 WPF 属性系统。由 WPF 属性系统支持的属性称为依赖属性。

## 本节内容

- 依赖项属性概述
- 附加属性概述
- 自定义依赖属性
- 依赖属性元数据
- 依赖属性回调和验证
- 框架属性元数据
- 依赖项属性值优先级
- 只读依赖项属性
- 属性值继承
- 依赖属性的安全性
- DependencyObject 的安全构造函数模式
- 集合类型依赖属性
- XAML 加载和依赖项属性
- 操作指南主题

## 参考

[DependencyProperty](#)

[PropertyMetadata](#)

[FrameworkPropertyMetadata](#)

[DependencyObject](#)

## 相关章节

[WPF 体系结构](#)

[WPF 中的 XAML](#)

[基元素](#)

[元素树和序列化](#)

事件

输入

资源

WPF 内容模型

线程模型

# 依赖属性概述

项目 • 2023/02/06

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。这些服务通常统称为 WPF 属性系统。由 WPF 属性系统支持的属性称为依赖属性。本概述介绍 WPF 属性系统以及依赖属性的功能。这包括如何在 XAML 和在代码中使用现有依赖属性。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。

## 先决条件

本主题假设你在 .NET 类型系统和面向对象的编程方面有一些基础知识。为了能理解本主题中的示例，还应了解 XAML 并知道如何编写 WPF 应用程序。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。

## 依赖属性和 CLR 属性

在 WPF 中，属性通常公开为标准 .NET 属性。在基本级别，可以直接与这些属性交互，而不必了解它们是以依赖属性的形式实现的。但是，应当熟悉 WPF 属性系统的部分或全部功能，以便利用这些功能。

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。这些其他输入可能包括系统属性（如主题和用户首选项）、实时属性确定机制（如数据绑定和动画/情节提要）、重用模板（如资源和样式）或者通过与元素树中其他元素的父子关系来公开的值。另外，可以通过实现依赖属性来提供独立验证、默认值、监视其他属性的更改的回叫以及可以基于可能的运行时信息来强制指定属性值的系统。派生类还可以通过重写依赖属性元数据（而不是重写现有属性的实际实现或者创建新属性）来更改现有属性的某些具体特征。

在 SDK 参考中，可以根据某个属性的托管引用页上是否存在“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”部分包括一个指向该依赖属性的 [DependencyProperty](#) 标识符字段的链接，还包括一个为该属性设置的元数据选项列表、每个类的重写信息以及其他详细信息。

## 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该属性的标准模式的替代实现方法。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](#)。[DependencyObject](#) 定义可以注册和拥有依赖属性的基类。

下面列出了与依赖属性一起使用的术语：

- **依赖属性**：[DependencyProperty](#) 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形式获取它，之后将其存储为类的静态成员。对于与 WPF 属性系统交互的许多 API，此标识符用作一个参数。
- **CLR“包装器”**：属性的实际 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，从而使用 WPF 属性系统为属性提供支持。

下面的示例定义 `IsSpinning` 依赖属性，并说明 [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(Boolean),
        typeof(MyCode)
    );
public bool IsSpinning
{
    get { return (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

下面的 XAML 示例将按钮的背景色指定为红色。此示例演示这样一种情况：在所生成的代码中，WPF XAML 分析器将 XAML 属性的简单字符串值类型转换为 WPF 类型（一个 [Color](#)，通过 [SolidColorBrush](#) 实现）。

## XAML

```
<Button Background="Red" Content="Button!" />
```

XAML 支持多种设置属性的语法格式。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）。有关属性设置的 XAML 语法的详细信息，请参阅 [WPF 中的 XAML 和 XAML 语法详述](#)。

作为非属性语法的示例，下面的 XAML 示例显示了另一种按钮背景。这一次不是设置简单的纯色，而是将背景设置为图像，用一个元素表示该图像并将该图像的源指定为嵌套元素的属性。这是属性元素语法的示例。

## XAML

```
<Button Content="Button!">
  <Button.Background>
    <ImageBrush ImageSource="wavy.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 Button();
myButton.Width = 200.0;
```

获取属性值实质上也是在调用 get“包装器”实现：

### C#

```
double whatWidth;
whatWidth = myButton.Width;
```

还可以直接调用属性系统 API `GetValue` 和 `SetValue`。如果使用的是现有属性，则上述操作通常不是必需的（使用包装器会更方便，并能够更好地向开发人员工具公开属性），但是在某些情况下适合直接调用 API。

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

依赖属性提供用来扩展属性功能的功能，这与字段支持的属性相反。通常，此类功能代表或支持以下特定功能之一：

- 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

依赖属性值可以通过引用资源来设置。资源通常指定为页面根元素或应用程序的 `Resources` 属性值（通过这些位置可以非常方便地访问资源）。以下示例演示如何定义 `SolidColorBrush` 资源。

XAML

```
<DockPanel.Resources>
  <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
</DockPanel.Resources>
```

在定义该资源之后，可以引用该资源并使用它来提供属性值：

XAML

```
<Button Background="{DynamicResource MyBrush}" Content="I am gold" />
```

这个特定资源称为 `DynamicResource` 标记扩展（在 WPF XAML 中，可以使用静态或动态资源引用）。若要使用动态资源引用，必须设置为依赖属性，因此它专门是由 WPF 属性系统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）。绑定本身通过数据源中的 [XPath](#) 指定所需的源属性。

XAML

```
<Button Content="{Binding XPath=Team/@TeamName}" />
```

### 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](#)。

为了生成数据绑定操作的 [DependencyObject](#) 源属性值的更改通知，依赖属性或 [DependencyObject](#) 类本身不支持 [INotifyPropertyChanged](#)。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变化的属性的详细信息，请参阅[数据绑定概述](#)。

## 样式

样式和模板是使用依赖属性的两个主要激发方案。在设置定义应用程序用户界面 (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实时值更改属性值的“触发器”。

以下示例创建一个简单的样式（该样式在 [Resources](#) 字典中定义，未显示），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}">I am green!</Button>
```

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。在应用和运行动画时，经过动画处理的值的操作优先级高于该属性以其他方式具有的任何值（如本地值）。

以下示例在 `Button` 属性上对 `Background` 进行动画处理（从技术上讲，通过使用属性元素语法将空白 `SolidColorBrush` 指定为 `Background` 来对 `Background` 进行动画处理，然后该 `SolidColorBrush` 的 `Color` 属性就是直接动画处理过的属性）。

## XAML

```
<Button>I am animated
<Button.Background>
  <SolidColorBrush x:Name="AnimBrush"/>
</Button.Background>
<Button.Triggers>
  <EventTrigger RoutedEvent="Button.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <ColorAnimation
          Storyboard.TargetName="AnimBrush"
          Storyboard.TargetProperty="(SolidColorBrush.Color)"
          From="Red" To="Green" Duration="0:0:5"
          AutoReverse="True" RepeatBehavior="Forever" />
      </Storyboard>
    </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，可以通过重写依赖属性的元数据来更改该属性的某些行为。重写元数据依赖于 `DependencyProperty` 标识符。重写元数据不需要重新实现属性。元数据的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元数据。

以下示例重写依赖属性 `DefaultStyleKey` 的元数据。重写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))
        );
    }
}
```

有关替代或获取属性元数据的详细信息，请参阅[依赖属性元数据](#)。

## 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确实会对性能产生一定的影响。属性值继承通常只有在特定方案指出适合使用属性值继承时才对属性启用。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，来确定该依赖属性是否继承属性值。

下面的示例演示一个绑定，并设置指定绑定（在前面的绑定示例中未显示出来）的源的 `DataContext` 属性。子对象中的任何后续绑定都无需指定源，它们可以使用父对象 `StackPanel` 中 `DataContext` 的继承值。（或者，子对象可以选择直接在 `Binding` 中指定自己的 `DataContext` 或 `Source`，并且有意不将继承值用于其绑定的数据上下文。）

XAML

```
<StackPanel Canvas.Top="50" DataContext="{Binding Source={StaticResource XmlTeamsSource}}">
    <Button Content="{Binding XPath=Team/@TeamName}"/>
</StackPanel>
```

有关详细信息，请参阅[属性值继承](#)。

## WPF 设计器集成

如果自定义控件具有实现为依赖属性的属性，则它会收到相应的适用于 Visual Studio 的 WPF 设计器支持。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

获取依赖属性的值时，获得的值可能是通过参与 WPF 属性系统的其他任一基于属性的输入而在该属性上设置的。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。

请看下面的示例。该示例包含适用于所有按钮及其 `Background` 属性的样式，但也会指定一个具有本地设置的 `Background` 值的按钮。

### 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。

原则上，对于第一个按钮，该属性会设置两次，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，对于第一个按钮的背景将使用本地设置的值，而不使用样式资源库值。第二个按钮没有本地值（而且没有其他比样式资源库优先级更高的值），因此该按钮中的背景来自样式资源库。

#### XAML

```
<StackPanel>
  <StackPanel.Resources>
    <Style x:Key="{x:Type Button}" TargetType="{x:Type Button}">
      <Setter Property="Background" Value="Red"/>
    </Style>
  </StackPanel.Resources>
  <Button Background="Green">I am NOT red!</Button>
  <Button>I am styled red</Button>
</StackPanel>
```

## 为什么存在依赖属性优先级？

通常，你不会希望总是应用样式，而且不希望样式遮盖单个元素的哪怕一个本地设置值（否则，通常难以使用样式或元素）。因此，来自样式的值的操作优先级低于本地设置

的值。有关依赖属性以及它的有效值可能来自何处的更完整列表，请参阅[依赖属性值优先级](#)。

### ① 备注

在 WPF 元素定义了许多非依赖属性的属性。一般说来，只有在需要支持至少一个由属性系统启用的方案（数据绑定、样式、动画、默认值支持、继承、附加属性或失效）时，才将属性实现为依赖属性。

## 了解有关依赖属性的详细信息

- 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 (CLR) 没有 1:1 的对应关系，而且不一定是依赖属性。附加属性的典型用途是使子元素可以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均没有该属性也是如此。一种主要情况是使子元素能够告知父元素应如何在 UI 中呈现它们；有关示例，请参阅 [Dock](#) 或 [Left](#)。有关详细信息，请参阅[附加属性概述](#)。
- 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，以便实现数据绑定或样式支持之类的功能，或者实现对失效和值强制的支持。有关详细信息，请参阅[自定义依赖属性](#)。
- 依赖属性应当被视为公共属性，这些公共属性可以由任何具有实例访问权限的调用方访问，或至少可被这样的调用方发现。有关详细信息，请参阅[依赖属性的安全性](#)。

## 另请参阅

- [自定义依赖属性](#)
- [只读依赖项属性](#)
- [WPF 中的 XAML](#)
- [WPF 体系结构](#)

# 附加属性概述

项目 · 2022/09/27

附加属性是由 XAML 定义的概念。附加属性旨在用作可在任何依赖项对象上设置的一类全局属性。在 Windows Presentation Foundation (WPF) 中，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。

## 先决条件

本文假设你从 Windows Presentation Foundation (WPF) 类上现有依赖属性的使用者角度已对依赖属性有所了解，并已阅读[依赖属性概述](#)。为理解本文中的示例，还应了解 XAML 并知道如何编写 WPF 应用程序。

## 为何使用附加属性

附加属性的一个用途是允许不同的子元素在父元素中定义的属性指定唯一值。此方案的一个具体应用是，让子元素通知父元素它们在用户界面 (UI) 中的呈现方式。一个示例是 `DockPanel.Dock` 属性。将 `DockPanel.Dock` 属性创建为附加属性，因为它旨在对包含在 `DockPanel` 中的元素进行设置，而不是对 `DockPanel` 本身进行设置。`DockPanel` 类定义了名为 `DockProperty` 的静态 `DependencyProperty` 字段，然后提供了 `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 `AttachedPropertyProvider.PropertyName` 来设置附加属性

下面的示例说明如何在 XAML 中设置 `DockPanel.Dock`：

XAML

```
<DockPanel>
    <TextBox DockPanel.Dock="Top">Enter text</TextBox>
</DockPanel>
```

此用法与静态属性有些类似，即总是引用拥有并注册附加属性的类型 `DockPanel`，而不是引用通过名称指定的任何实例。

此外，由于 XAML 中的附加属性是在标记中设置的属性，因此，只有设置操作具有相关性。尽管存在一些用于比较值的间接机制（如在样式中触发），但无法直接在 XAML 中直接获取属性（有关详细信息，请参阅[样式设置和模板化](#)）。

# WPF 中的附加属性实现

在 Windows Presentation Foundation (WPF) 中，WPF 类型上的大多数与 UI 相关的附加属性都作为依赖属性实现。附加属性是个 XAML 概念，而依赖属性则是个 WPF 概念。因为 WPF 附加属性是依赖属性，所以它们支持依赖属性概念，如属性元数据和属性元数据默认值。

## 所属类型 如何使用附加属性

尽管可以在任何对象上设置附加属性，但这并不自动意味着设置该属性会产生实际的结果，或者该值会被其他对象使用。通常，附加属性是为了使来自各种可能的类层次结构或逻辑关系的对象都可以向用于定义附加属性的类型报告公用信息。定义附加属性的类型通常采用以下模型之一：

- 设计定义附加属性的类型，以便它可以是将为附加属性设置值的元素的父元素。随后，该类型将在内部逻辑中对照某些对象树结构循环访问其子对象，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 定义附加属性的最典型方案是：父元素支持子元素集合，并实现分别为每个子元素报告行为细节的行为。

[DockPanel](#) 定义 [DockPanel.Dock](#) 附加属性，且 [DockPanel](#) 具有类级代码作为其呈现逻辑的一部分（具体是 [MeasureOverride](#) 和 [ArrangeOverride](#)）。[DockPanel](#) 实例将始终检查它，以查看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该特定子元素的呈现逻辑的输入。嵌套的 [DockPanel](#) 实例处理它们各自的直接子元素集合，但是这种行为是特定于如何 [DockPanel](#) 处理 [DockPanel.Dock](#) 值来实现的。理论上，可以有影响直接父级之外的元素的附加属性。如果对某个元素设置了 [DockPanel.Dock](#) 附加属性，而该元素没有 [DockPanel](#) 父元素对其进行操作，则不会引发错误或异常。这仅仅意味着设置了全局属性值，但它没有可以使用这一信息的当前 [DockPanel](#) 父级。

## 代码 中的附加属性

WPF 中的附加属性没有用于简化 get/set 访问的典型 CLR“包装”方法。这是因为附加属性不是必须属于设置它的实例的 CLR 命名空间的一部分。但是，分析 XAML 时，XAML 处理器必须能够设置这些值。若要支持有效的附加属性用法，附加属性的所有者类型必须在“获取 PropertyName”和“设置 PropertyName”窗体中实现专用访问器方法。这些专用访问器方法对在代码中设置附加属性也很有帮助。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，且支持字段可在任何对象上存在，无需专门定义。

下面的示例演示如何在代码中设置附加属性。在此示例中，`myCheckBox` 是 `CheckBox` 类的实例。

C#

```
DockPanel myDockPanel = new DockPanel();
CheckBox myCheckBox = new CheckBox();
myCheckBox.Content = "Hello";
myDockPanel.Children.Add(myCheckBox);
DockPanel.SetDock(myCheckBox, Dock.Top);
```

与 XAML 用例类似，如果 `myCheckBox` 尚未由第四行代码将其添加为 `myDockPanel` 的子元素，则第五行代码不会引发异常，但该属性值不会与 `DockPanel` 父级交互，因此也就不会执行任何操作。只有当对子元素设置了 `DockPanel.Dock` 值且存在 `DockPanel` 父级元素时，才会在呈现的应用程序中产生有效行为。（在这种情况下，可以设置附加属性，然后附加到树。或者可以附加到树，然后设置附加属性。任一操作顺序均得出相同结果。）

## 附加属性元数据

注册该属性时，设置 `FrameworkPropertyMetadata` 以指定属性的特征，如属性是否会影响呈现、度量等。附加属性的元数据通常与依赖属性上的元数据基本上都相同。如果在附加属性元数据替代中指定默认值，该值将成为替代类实例上显式附加属性的默认值。具体而言，当某些进程通过该属性的 `Get` 方法访问器请求附加属性值，并指定在其中指定元数据的类的示例时，将报告默认值，而不会设置该附加属性的值。

如果希望对属性启用属性值继承，应使用附加属性，而不是非附加的依赖属性。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

当需要有一个可用于定义类之外的其他类的属性设置机制时，建议创建附加属性。对于这一情况，最常见的方案是布局。现有布局属性的示例包括 [DockPanel.Dock](#)、[Panel.ZIndex](#) 和 [Canvas.Top](#)。这里启用的方案是，作为布局控制元素的子元素存在的元素能够分别向其布局父级元素表达布局要求，其中每个元素都设置一个被父级定义为附加属性的属性值。

使用附加属性的另一种情况是，你的类表示一种服务，且你希望类能够以更透明的方式继承该服务。

但还有一种情况是接收 Visual Studio WPF 设计器支持，例如“属性”窗口编辑。有关详细信息，请参阅[控件创作概述](#)。

如前文所述，如果想要使用属性值继承，你应该注册为附加属性。

## 如何创建附加属性

如果你的类将附加属性严格定义为用于其他类型，则该类不必从 [DependencyObject](#) 派生。但如果遵循使附加属性也作为依赖属性的总体 WPF 模型，则确实需要从 [DependencyObject](#) 派生。

通过声明 [DependencyProperty](#) 类型的 `public static readonly` 字段将附加属性定义为依赖属性。可以使用 [RegisterAttached](#) 方法的返回值来定义此字段。字段名称必须匹配附加属性名称，且追加字符串 `Property`，以遵循命名标识字段及其所表示的属性的 WPF 模式。附加属性提供程序还必须提供静态的“获取 `PropertyName`”和“设置 `PropertyName`”方法，作为附加属性访问器，否则将导致属性系统无法使用附加属性。

### ① 备注

如果省略附加属性的 `get` 访问器，该属性的数据绑定在设计工具（如 Visual Studio 和 Blend for Visual Studio）中将无法工作。

## Get 访问器

“获取 `PropertyName`”访问器的签名必须如下所示：

```
public static object GetPropertyName(object target)
```

- `target` 对象在实现中可以指定为更具体的类型。例如，[DockPanel.GetDock](#) 方法将该参数的类型设置为 [UIElement](#)，因为附加属性仅用于设置 [UIElement](#) 实例。
- 返回值在实现中可以指定为更具体的类型。例如，[GetDock](#) 方法将其类型设置为 [Dock](#)，因为此值只能设置为该枚举。

## Set 访问器

“设置 PropertyName”访问器的签名必须如下所示：

```
public static void SetPropertyName(object target, object value)
```

- `target` 对象在实现中可以指定为更具体的类型。例如，`SetDock` 方法将其类型设置为 `UIElement`，因为附加属性仅用于设置 `UIElement` 实例。
- `value` 对象在实现中可以指定为更具体的类型。例如，`SetDock` 方法将其类型设置为 `Dock`，因为此值只能设置为该枚举。请记住，此方法的值是 XAML 加载器在标记中的附加属性用法中遇到附加属性时的输入。该输入是在标记中指定为 XAML 属性值的值。因此必须存在可用于你所使用的类型的类型转换、值序列化程序或标记扩展支持，以便可以从属性值（最终仅仅是一个字符串）创建相应的类型。

下面的示例介绍了依赖属性注册（使用 `RegisterAttached` 方法），以及“获取 `PropertyName`”和“设置 `PropertyName`”访问器。在此示例中，附加属性名称为 `IsBubbleSource`。因此，访问器必须名为 `GetIsBubbleSource` 和 `SetIsBubbleSource`。

C#

```
public static readonly DependencyProperty IsBubbleSourceProperty =
DependencyProperty.RegisterAttached(
    "IsBubbleSource",
    typeof(Boolean),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(false,
    FrameworkPropertyMetadataOptions.AffectsRender)
);
public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}
public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

## 附加属性特性

WPF 定义了多个 .NET 属性，后者用于向反射进程和反射的典型用户提供关于附加属性的信息以及如设计器等属性信息。由于附加属性的类型没有范围限制，因此设计者需要一种方法来避免用户查看全局列表时，看到使用 XAML 的特定技术实现中定义的所有附加属性。WPF 为附加属性定义的 .NET 属性可用于限定应在属性窗口中显示给定附加属性的情况的范围。你还可考虑对自己的自定义附加属性应用这些特性。有关 .NET 属性的用途和语法说明，请参阅相应参考页面：

- [AttachedPropertyBrowsableAttribute](#)
- [AttachedPropertyBrowsableForChildrenAttribute](#)
- [AttachedPropertyBrowsableForTypeAttribute](#)
- [AttachedPropertyBrowsableWhenAttributePresentAttribute](#)

## 深入了解附加属性

- 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](#)。
- 还可将属性注册为附加属性和依赖属性，但仍需公开“包装器”实现。在这种情况下，属性可在该元素上设置，也可通过 XAML 附加属性语法在任何元素上设置。[FrameworkElement.FlowDirection](#) 是一个为标准用法和附加用法提供适当场景的属性示例。

## 另请参阅

- [DependencyProperty](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [WPF 中的 XAML](#)
- [注册附加属性](#)

# 自定义依赖项属性

项目 · 2022/09/27

本主题介绍 Windows Presentation Foundation (WPF) 应用程序开发人员和组件作者想要创建自定义依赖属性的原因，属性的实现步骤以及一些可以提高属性性能、可用性或通用性的实现选项。

## 先决条件

本主题假定你作为 WPF 类的现有依赖属性的使用者已经对依赖属性有所了解，并且已经阅读了[依赖属性概述](#)主题。若要理解本主题中的示例，还应当了解 WPF 应用程序。

## 什么是依赖属性？

可以通过启用原本是公共语言运行时 (CLR) 属性的属性来支持样式设置、数据绑定、继承、动画和默认值，方法是将该属性作为依赖属性进行实现。依赖属性是通过调用 [Register](#) 方法（或 [RegisterReadOnly](#)）在 WPF 属性系统中注册，并且受 [DependencyProperty](#) 标识符字段支持的属性。只有 [DependencyObject](#) 类型可以使用依赖属性，但是 [DependencyObject](#) 在 WPF 类层次结构中级别很高，因此 WPF 中大部分可用的类都支持依赖属性。若要详细了解依赖属性和此 SDK 中对依赖属性进行描述所使用的一些术语和约定，请参阅[依赖属性概述](#)。

## 依赖属性示例

WPF 类上实现的依赖属性示例包括 [Background](#) 属性、[Width](#) 属性、[Text](#) 属性以及其他众多属性。类公开的每个依赖属性都有一个在相同类上公开且类型为 [DependencyProperty](#) 的相应公共静态字段。这是依赖属性的标识符。此标识符的命名约定为：依赖属性名称后面加上字符串 `Property`。例如，[Background](#) 属性对应的 [DependencyProperty](#) 标识符字段为 [BackgroundProperty](#)。标识符存储了注册依赖属性时的相关信息，之后此标识符用于涉及依赖属性的其他操作，例如调用 [SetValue](#)。

如[依赖属性概述](#)中所述，因为“包装器”实现，WPF 中的所有依赖属性（大多数附加属性除外）也是 CLR 属性。因此，在代码中通过调用定义包装器的 CLR 访问器（调用方法与使用其他 CLR 属性相同）可以获取或设置依赖属性。已建立依赖属性的使用者通常不会使用 [DependencyObject](#) 方法 [GetValue](#) 和 [SetValue](#)（这二者为基础属性系统的连接点）。相反，CLR 属性的现有实现已经使用相应的标识符字段在属性的 `get` 和 `set` 包装器实现中调用 [GetValue](#) 和 [SetValue](#)。若要自己实现自定义依赖属性，则需要使用类似的方法定义包装器。

# 应该何时实现依赖属性？

在类上实现属性时，只要类派生自 [DependencyObject](#)，就可以选择使用 [DependencyProperty](#) 标识符支持属性，从而使其成为依赖属性。不必总是将属性实现为依赖属性，这不一定合适，具体取决于方案需要。有时，使用私有字段支持属性的通常方法已足够满足需求。但是，如果要使属性支持以下一个或多个 WPF 功能，则应该将属性实现为依赖属性：

- 需要在样式中设置属性。有关详细信息，请参阅[样式设置和模板化](#)。
- 需要属性支持数据绑定。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](#)。
- 需要可以使用动态资源引用设置属性。有关详细信息，请参阅[XAML 资源](#)。
- 需要从元素树中的父元素自动继承属性值。这种情况下，即使为 CLR 访问也创建了属性包装器，也应该使用 [RegisterAttached](#) 方法注册。有关详细信息，请参阅[属性值继承](#)。
- 需要属性可以进行动画处理。有关详细信息，请参阅[动画概述](#)。
- 需要属性系统在先前的值因属性系统、环境或用户执行的操作而发生更改，或者因读取和使用样式而发生更改时进行报告。通过使用属性元素据，属性可以指定回调方法，每次属性系统确定属性值已明确改动时将调用此回调方法。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](#)。
- 需要使用同时也被 WPF 进程使用的已建立的元数据约定，例如报告更改属性值是否需要布局系统重新安排元素的视觉对象。或者需要能够使用元素据替代，以便派生类可以更改基于元数据的特性，例如默认值。
- 需要自定义控件的属性接收“属性”窗口编辑等 Visual Studio WPF 设计器支持。有关详细信息，请参阅[控件创作概述](#)。

检查这些方案时，还应考虑是否可以通过替代现有依赖属性的元素据而不是通过实现一个全新的属性来实现方案。元数据替代是否可行取决于方案以及方案与现有 WPF 依赖属性和类中的实现的相似度。有关替代现有属性上的元素据的详细信息，请参阅[依赖属性元素据](#)。

## 定义依赖属性的检查清单

定义依赖属性包含 4 个不同的概念。这些概念并不一定是严格的过程步骤，因为其中一些概念在实现中会被合并为一行代码：

- （可选）创建依赖属性的属性元素据。

- 在属性系统中注册属性名称，并指定所有者类型和属性值类型。此外，还应指定属性元素据（如果用到）。
- 在所有者类型上将 `DependencyProperty` 标识符定义为 `public static readonly` 字段。
- 定义一个 CLR“包装器”属性，并且其名称与依赖属性名称相匹配。实现 CLR“包装器”属性的 `get` 和 `set` 访问器，以连接支持此属性的依赖属性。

## 在属性系统中注册属性

为使属性成为依赖属性，必须在属性系统维护的表中注册该属性，并为属性指定一个唯一标识符。此唯一标识符会用作后续属性系统操作的限定符。这些操作可能是内部操作，也可能使用你自己的代码调用属性系统 API。若要注册属性，可在类的主体内（在类中但在所有成员定义外）调用 `Register` 方法。`Register` 方法调用也会提供标识符字段作为返回值。`Register` 调用在其他成员定义外完成的原因在于，需要使用此返回值分配并创建一个 `DependencyProperty` 类型的 `public static readonly` 字段，作为类的一部分。此字段会作为依赖属性的标识符。

C#

```
public static readonly DependencyProperty AquariumGraphicProperty =
DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
```

## 依赖属性命名约定

必需完全遵循已有的依赖属性命名约定，例外情况除外。

依赖属性本身有一个基本名称（此示例中为“`AquariumGraphic`”），此名称用作 `Register` 的第一个参数。此名称在每个注册类型内必须唯一。通过基类型继承的依赖属性会被视为注册类型的已有部分；无法再次注册已继承属性的名称。但是，即使不继承依赖属性，也有方法可将类添加为依赖属性的所有者；有关详细信息，请参阅[依赖属性元素据](#)。

创建标识符字段时，按注册时的属性名称命名此字段，再加上后缀 `Property`。此字段是依赖属性的标识符，之后会被用作在包装器中进行 `SetValue` 和 `GetValue` 调用的输入，供

任何其他代码通过你自己的代码、经过允许的外部代码、属性系统甚至可能通过 XAML 处理器来访问属性时使用。

### 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。

## 实现“包装器”

包装器实现应在 `get` 实现中调用 `GetValue`，并在 `set` 实现中调用 `SetValue`（为清楚起见，此处也显示了原始注册调用和字段）。

除特殊情况外，包装器实现仅应执行 `GetValue` 和 `SetValue` 操作。其原因请参阅 [XAML 加载和依赖属性](#) 主题。

WPF 类上提供的所有现有公共依赖属性都使用这一简单的包装器实现模型；大多数情况下，依赖属性工作原理的复杂性本质上在于它是属性系统的行为，还是通过其他概念（例如强制转换或通过属性元数据进行的属性更改回调）实现的行为。

C#

```
public static readonly DependencyProperty AquariumGraphicProperty =
DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
public Uri AquariumGraphic
{
    get { return (Uri)GetValue(AquariumGraphicProperty); }
    set { SetValue(AquariumGraphicProperty, value); }
}
```

同样，根据约定，包装器属性名称必须与注册属性的 `Register` 调用选择并指定的第一个参数相同。如果属性不遵从此约定，尽管不一定会禁用所有可能的用法，但你会遇到几个比较突出的问题：

- 样式和模板的某些方面不起作用。

- 大多数工具和设计器必须依赖命名约定，才能正确序列化 XAML 或在每个属性级别提供设计器环境帮助。
- 处理特性值时，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。有关详细信息，请参阅 [XAML 加载和依赖属性](#)。

## 新依赖属性的属性元数据

注册依赖属性时，通过属性系统进行注册会创建一个存储属性特征的元素据对象。如果属性使用 [Register](#) 的简单签名进行注册，则其中许多特征会使用设置的默认值。使用 [Register](#) 的其他签名在注册属性时可以指定需要的元数据。为依赖属性使用的最常见元数据是为其使用默认值。该默认值适用于使用此属性的新实例。

如果要创建一个存在于 [FrameworkElement](#) 派生类上的依赖属性，可使用更专业的元数据类 [FrameworkPropertyMetadata](#)，而不是基本的 [PropertyMetadata](#) 类。

[FrameworkPropertyMetadata](#) 类的构造函数具有数个签名，可在这些签名中同时指定多个元数据特征。若仅需指定默认值，请使用采用 [Object](#) 类型的单个参数的签名。将该对象参数作为属性的特定于类型的默认值进行传递（提供的默认值必须是 [Register](#) 调用中作为 `propertyType` 参数提供的类型）。

对于 [FrameworkPropertyMetadata](#)，还可以为属性指定元数据选项标记。注册后这些标记会转换为属性元素据上的不同属性，并用于将某些条件传送给布局引擎等其他进程。

## 设置合适的元数据标记

- 属性（或属性值的更改）影响用户界面（UI），特别是影响布局系统如何调整页面中元素的大小或呈现方式时，请设置以下一个或多个标记：[AffectsMeasure](#)、[AffectsArrange](#)、[AffectsRender](#)。
  - [AffectsMeasure](#) 指示更改此属性需要更改 UI 呈现，其中包含的对象在父级内可能需要更多或更少的空间。例如，“宽度”属性应该设置此标记。
  - [AffectsArrange](#) 指示更改此属性需要更改 UI 呈现，通常无需在专用空间中进行更改，但会指示该空间内的位置已发生更改。例如，“对齐”属性应该设置此标记。
  - [AffectsRender](#) 指示已发生一些其他更改，这些更改不会影响布局和度量值，但需要其他的呈现方式。更改现有元素的颜色的属性便是一个示例，例如“背景”。
  - 对属性系统或布局回调进行自己的替代实现时，这些标记通常用作元数据中的协议。例如，如果实例的任何属性报告值发生了更改，并且在其元数据中将 [AffectsArrange](#) 设置为 `true`，则可以使用调用 [InvalidateArrange](#) 的 [OnPropertyChanged](#) 回调。

- 超出上述所需大小时，某些属性可能会影响所含父元素的呈现特征。其中一个示例是流文档模型中使用的 [MinOrphanLines](#) 属性，对该属性的更改会更改包含该段落的流文档的整体呈现。使用 [AffectsParentArrange](#) 或 [AffectsParentMeasure](#) 标识自己属性中相似的情况。
- 默认情况下，依赖属性支持数据绑定。在无实际的数据绑定方案或大型对象的数据绑定性能构成问题的情况下，可有意禁用数据绑定。
- 默认情况下，依赖属性的数据绑定 [Mode](#) 默认为 [OneWay](#)。随时可以将每个绑定实例的绑定更改为 [TwoWay](#)；有关详细信息，请参阅[指定绑定的方向](#)。但作为依赖属性的作者，你可选择将属性设置为默认使用 [TwoWay](#) 绑定模式。现有依赖属性的一个示例是 [MenuItem.Is\\_submenuOpen](#)；此属性的应用场景为：[Is\\_submenuOpen](#) 设置逻辑和 [MenuItem](#) 合成与默认的主题样式交互。[Is\\_submenuOpen](#) 属性逻辑以本机方式使用数据绑定，使属性状态与其他状态属性和方法调用保持一致。另一个默认情况下绑定 [TwoWay](#) 的示例属性是 [TextBox.Text](#)。
- 还可以通过设置 [Inherits](#) 标记在自定义依赖属性中启用属性继承。在父元素和子元素具有相同属性的情况下，属性继承非常有用，它可以使子元素将该特定属性值设置为与父元素设置的值相同。可继承属性的一个示例是 [DataContext](#)，此属性用于绑定操作，以便为数据呈现启用重要的主-从方案。通过使 [DataContext](#) 成为可继承属性，所有子元素还会继承该数据上下文。因为使用了属性值继承，你可以在页面或应用程序根目录上指定数据上下文，而无需对所有可能子元素中的绑定重新指定上下文。[DataContext](#) 也是演示继承替代默认值的一个很好的示例，但始终可以在任何特定子元素上对其进行本地设置；有关详细信息，请参阅[对分层数据使用主-从模式](#)。属性值继承确实可能存在性能成本，因此应谨慎使用；有关详细信息，请参阅[属性值继承](#)。
- 设置 [Journal](#) 标记，以指示导航日志服务是否应该检测或使用依赖属性。其中一个示例是 [SelectedIndex](#) 属性；导航日志历史记录时应保留选择控件中选择的任何项。

## 只读依赖项属性

可以定义只读的依赖属性。但是，为何将属性定义为只读的情况略有不同，其过程与在属性系统中注册属性并公开标识符相同。有关详细信息，请参阅[只读依赖属性](#)。

## 集合类型依赖项属性

集合类型依赖属性要考虑一些其他实现问题。有关详细信息，请参阅[集合类型依赖属性](#)。

# 依赖属性安全注意事项

依赖属性应声明为公共属性。 依赖属性标识符字段应声明为公共静态字段。 即使尝试声明其他访问级别（例如受保护），也始终可以通过标识符和属性系统 API 来访问依赖属性。 由于元数据报告或值确定 API 属于属性系统，因此甚至可以访问受保护的标识符字段，例如 [LocalValueEnumerator](#)。 有关详细信息，请参阅[依赖属性的安全性](#)。

## 依赖属性和类构造函数

托管代码编程（通常通过 FxCop 等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方法。 这是因为构造函数可以作为派生的类构造函数的基本初始化来调用，并且可能会在所构造的对象实例不完全初始化状态下通过构造函数输入虚方法。 从已派生自 [DependencyObject](#) 的任何类进行派生时，应注意到属性系统本身会在内部调用和公开虚方法。 这些虚方法属于 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

- [依赖项属性概述](#)
- [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 加载和依赖项属性](#)
- [DependencyObject 的安全构造函数模式](#)

# 依赖项属性元数据

项目 · 2023/02/06

Windows Presentation Foundation (WPF) 属性系统包括一个元数据报告系统，该系统不局限于可以通过反射或常规公共语言运行时 (CLR) 特征报告的关于某个属性的内容。 依赖属性的元数据还可以由定义依赖属性的类来唯一地分配，可以在依赖属性添加到另一个类时进行更改，可以由所有从定义基类继承依赖属性的派生类来明确地重写。

## 先决条件

本主题假定你从 WPF 应用程序的现有依赖属性的使用者角度了解依赖属性。

## 依赖属性元数据的使用方式

依赖属性的元数据作为一个对象存在，可以通过查询该对象来检查依赖属性的特征。 当属性系统处理任何给定的依赖属性时，也会经常访问这些元数据。 依赖属性的元数据对象可以包含以下类型的信息：

- 依赖属性的默认值（如果通过本地值、样式和继承等信息不能确定依赖属性的其他任何值）。有关在为依赖属性赋值时，默认值如何参与属性系统所使用的优先级的完整讨论，请参阅[依赖属性值优先级](#)。
- 对影响每个所有者类型的强制行为或更改通知行为的回叫实现的引用。请注意，这些回叫通常是非公共访问级别定义的，因此，除非实际引用位于允许的访问范围内，否则通常无法从元数据获得这些引用。有关依赖属性回叫的详细信息，请参阅[依赖属性回调和验证](#)。
- 如果所讨论的依赖属性被视为一个 WPF 框架级别的属性，则元数据中可能包含 WPF 框架级别的依赖属性特征，这些特征报告各种服务（如 WPF 框架级别的布局引擎和属性继承逻辑）的信息和状态。有关依赖属性元数据的这一方面的详细信息，请参阅[框架属性元数据](#)。

## 元数据 API

可报告属性系统所使用的大部分元数据信息的类型是 [PropertyMetadata](#) 类。 在向属性系统注册依赖属性时，可以选择指定元数据实例，并且可以为以下附加类型再次指定这些实例：将自身作为所有者添加的类型，或者重写它们从基类依赖属性定义继承的元数据的类型。（如果在注册属性时未指定元数据，则使用该类的默认值来创建默认 [PropertyMetadata](#)。）当针对 [DependencyObject](#) 实例调用各种从依赖属性获取元数据的 [GetMetadata](#) 重载时，所注册的元数据作为 [PropertyMetadata](#) 返回。

随后会从 [PropertyMetadata](#) 类派生，以便为体系结构区域（如 WPF 框架级别的类）提供更具体的元数据。[UIPropertyMetadata](#) 添加动画报告功能，[FrameworkPropertyMetadata](#) 提供上一节中提到的 WPF 框架级别的属性。在注册依赖属性之后，可以将它们注册到这些 [PropertyMetadata](#) 派生类中。在检查元数据之后，可以选择将 [PropertyMetadata](#) 基类型强制转换为派生类，这样就可以检查更具体的属性。

### ① 备注

在本文档中，有时将可以在 [FrameworkPropertyMetadata](#) 中指定的属性特征称为“标志”。在新建要用于注册依赖属性或重写元数据的元数据实例时，将使用按标志枚举 [FrameworkPropertyMetadataOptions](#) 来指定这些值，然后向 [FrameworkPropertyMetadata](#) 构造函数提供枚举的可能连接值。但是，一经构造，这些选项特征就会在 [FrameworkPropertyMetadata](#) 中作为一系列布尔属性而不是构造枚举值公开。使用布尔属性，可以检查每个条件，而不必为了获得感兴趣的信息而向按标志枚举值应用掩码。构造函数使用连接的 [FrameworkPropertyMetadataOptions](#) 使构造函数签名保持合理的长度，而实际构造的元数据公开不同的属性，使元数据的查询变得更加直观。

## 何时重写元数据以及何时派生类

WPF 属性系统已经建立了如下功能：在不必完全重新实现依赖属性的情况下，更改依赖属性的某些特征。这是通过为特定类型上所存在的依赖属性构造不同的属性元数据实例来完成的。请注意，现有的大多数依赖属性都不是虚拟属性，因此，严格地说，只能通过隐藏现有成员来针对继承类“重新实现”依赖属性。

如果尝试对某个类型的依赖属性启用的方案不能通过修改现有依赖属性的特征来完成，则可能有必要创建一个派生类，然后为该派生类声明一个自定义依赖属性。自定义依赖属性与 WPF API 定义的依赖属性具有相同的行为。有关自定义依赖属性的更多详细信息，请参阅[自定义依赖属性](#)。

不能重写的依赖属性的一个显著特征就是它的值类型。如果要继承的依赖属性的行为与所需的行为大体相同，但是要求它具有另一种类型，则必须实现一个自定义依赖属性，可能还需要通过类型转换或其他实现机制在自定义类上链接这些属性。而且，不能替换现有的 [ValidateValueCallback](#)，因为此回叫存在于注册字段本身，而不是存在于它的元数据中。

## 更改现有元数据的方案

如果要处理现有依赖属性的元数据，则更改依赖属性元数据的一种常见方案是更改默认值。更改或添加属性系统回叫是一种更高级的方案。如果所实现的派生类的依赖属性之

间具有不同的相互关系，则你可能希望这样做。让编程模型既支持代码又支持声明性用法的条件之一就是，属性必须能够按任何顺序设置。因此，需要在没有上下文的情况下实时设置任何依赖属性，而且可以不必知道设置顺序（例如，可能在构造函数中找到的顺序）。有关属性系统这一方面的详细信息，请参阅[依赖属性回调和验证](#)。请注意，验证回叫不是元数据的一部分，而是依赖属性标识符的一部分。因此，不能通过重写元数据来更改验证回叫。

在某些情况下，可能还希望在现有的依赖属性上改变 WPF 框架级别的属性元数据选项。这些选项将有关 WPF 框架级别属性的某些已知条件传递到其他 WPF 框架级别的进程，例如布局系统。通常，只有在注册新的依赖属性时，才设置这些选项，但是也可以在调用 [OverrideMetadata](#) 或 [AddOwner](#) 的过程中更改 WPF 框架级别属性的元数据。有关要使用的特定值以及详细信息，请参阅[框架属性元数据](#)。有关应当如何为新注册的依赖属性设置这些选项的详细信息，请参阅[自定义依赖属性](#)。

## 重写元数据

重写元数据的主要目的在于，使你有机会更改各种派生自元数据的行为，这些行为应用于类型上存在的依赖属性。[元数据](#)一节中更详细地介绍了重写元数据的原因。有关详细信息（包括一些代码示例），请参阅[重写依赖属性的元数据](#)。

在注册调用 ([Register](#)) 过程中可以为依赖属性提供属性元数据。但是，在许多情况下，当类继承该依赖属性时，你可能希望为该类提供特定于类型的元数据。为此，可调用 [OverrideMetadata](#) 方法。对于 WPF API 中的示例，[FrameworkElement](#) 类是第一个注册 [Focusable](#) 依赖属性的类型。但是，[Control](#) 类会重写该依赖属性的元数据以提供自己的初始默认值，将它从 `false` 改为 `true`，并以其他方式重用最初的 [Focusable](#) 实现。

当你重写元数据时，系统会合并或替换不同的元数据特征。

- 合并 [PropertyChangedCallback](#)。如果添加一个新的 [PropertyChangedCallback](#)，该回叫则存储在元数据中。如果没有在重写中指定 [PropertyChangedCallback](#)，则 [PropertyChangedCallback](#) 的值会从在元数据中指定它的最近上级提升为一个引用。
- [PropertyChangedCallback](#) 的实际属性系统行为是：层次结构中所有元数据所有者的实现都保留并添加到表中，属性系统的执行顺序是首先调用最深派生类的回叫。
- 替换 [DefaultValue](#)。如果没有在替代中指定 [DefaultValue](#)，[DefaultValue](#) 的值则来自在元数据中指定它的最近上级。
- 会替换 [CoerceValueCallback](#) 实现。如果添加一个新的 [CoerceValueCallback](#)，该回叫则存储在元数据中。如果没有在替代中指定 [CoerceValueCallback](#)，[CoerceValueCallback](#) 的值则会从在元数据中指定它的最近上级提升为一个引用。

- 属性系统的行为是仅调用直接元数据中的 [CoerceValueCallback](#)。而不保留对层次结构中所实现的其他 [CoerceValueCallback](#) 的引用。

此行为由 [Merge](#) 实现，并且可以在派生的元数据类上重写。

## 重写附加属性元数据

在 WPF 中，附加属性作为依赖属性来实现。这意味着它们还具有能够由个别类重写的属性元数据。关于 WPF 中的附加属性的范围，通常需要注意：可以针对任何 [DependencyObject](#) 设置附加属性。因此，任何 [DependencyObject](#) 派生类都可以重写任何附加属性的元数据，就好像它们是在类的实例上设置的一样。可以重写默认值、回叫或 WPF 框架级别的特征报告属性。如果针对类的实例设置了附加属性，则这些重写属性元数据特征将适用。例如，可以重写默认值，这样，只要未以其他方式设置附加属性，重写值就会报告为类实例的附加属性的值。

### ① 备注

[Inherits](#) 属性与附加属性无关。

## 将类作为现有依赖属性的所有者来添加

通过使用 [AddOwner](#) 方法，类可以将自身作为已注册的依赖属性的所有者来添加。这使得该类可以使用最初针对另一个类型注册的依赖属性。添加类通常不是首先将该依赖属性注册为所有者的类型的派生类。实际上，这使类及其派生类可以“继承”依赖属性实现，而不需要最初的所有者类，而且添加类也不必位于同一个实际的类层次结构中。另外，添加类（以及所有派生类）随后可以为最初的依赖属性提供特定于类型的元数据。

添加类除了通过属性系统的实用工具方法将自身添加为所有者以外，还应当在自身声明其他公共成员，以使依赖属性向代码和标记公开，从而完全参与属性系统。就为依赖属性公开对象模型而言，添加现有依赖属性的类与定义新的自定义依赖属性的类具有相同的职责。要公开的第一个此类成员是依赖属性标识符字段。此字段应当是类型为 [DependencyProperty](#) 的 `public static readonly` 字段，此类型将赋予 [AddOwner](#) 调用的返回值。要定义的第二个成员是公共语言运行时 (CLR)“包装器”属性。使用包装器，可以更方便地在代码中操作依赖属性（应当避免每次都调用 [SetValue](#)，而且在包装器本身只能发出该调用一次）。包装器的实现方式与在注册自定义依赖属性时的实现方式完全相同。有关实现依赖属性的详细信息，请参阅[自定义依赖属性](#)和[为依赖属性添加所有者类型](#)。

## AddOwner 和附加属性

对于由所有者类定义为附加属性的依赖属性，可以调用 [AddOwner](#)。这样做的目的通常是为了将以前附加的属性作为非附加依赖属性来公开。随后将 [AddOwner](#) 返回值作为一个要用作依赖属性标识符的 `public static readonly` 字段来公开，并定义相应的“包装器”属性，以便该属性出现在成员表中并支持在类中使用非附加属性。

## 另请参阅

- [PropertyMetadata](#)
- [DependencyObject](#)
- [DependencyProperty](#)
- [GetMetadata](#)
- [依赖项属性概述](#)
- [框架属性元数据](#)

# 依赖项属性回调和验证

项目 · 2023/02/06

本主题介绍如何使用与属性相关的功能（如验证确定、更改属性的有效值时调用的回调）的替代自定义实现，并重写对值确定的外部可能影响来创建依赖属性。本主题还讨论使用这些技术扩展默认属性系统行为所适用的方案。

## 先决条件

本主题假定你了解实现依赖属性的基本方案，以及如何将元数据应用于自定义依赖属性。有关上下文，请参阅[自定义依赖属性](#)和[依赖属性元数据](#)。

## 验证回叫

在首次注册依赖属性时，可以为其分配验证回叫。验证回叫不属于属性元数据，而是[Register](#)方法的直接输入。因此，在为某个依赖属性创建验证回叫后，新实现无法重写该验证回叫。

C#

```
public static readonly DependencyProperty CurrentReadingProperty =
DependencyProperty.Register(
    "CurrentReading",
    typeof(double),
    typeof(Gauge),
    new FrameworkPropertyMetadata(
        Double.NaN,
        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(OnCurrentReadingChanged),
        new CoerceValueCallback(CoerceCurrentReading)
    ),
    new ValidateValueCallback(IsValidReading)
);
public double CurrentReading
{
    get { return (double)GetValue(CurrentReadingProperty); }
    set { SetValue(CurrentReadingProperty, value); }
}
```

实现回叫，以便为其提供对象值。如果提供的值对属性有效，回叫会返回 `true`；否则，回叫返回 `false`。假定按照向属性系统注册的类型，属性的类型是正确的，因此通常不会在回叫内执行类型检查。属性系统可在多种不同操作中使用回叫。其中包括按照默认值进行初始类型初始化、通过调用 [SetValue](#) 进行编程更改或尝试使用提供的新默认值重

写元数据。如果验证回叫是通过其中任何一种操作调用的，并且返回 `false`，则会引发异常。应用程序编写器必须准备处理这些异常。验证回叫常用于验证枚举值，或在属性设置的度量值必须大于等于零时约束整数值或双精度型值。

验证回叫专用作类验证程序，而不用作实例验证程序。回叫参数不会传递设置了要验证的属性的特定 `DependencyObject`。因此，验证回叫不能用来强制执行可能会影响属性值的“依赖项”，其中，某个属性特定于实例的值依赖于其他属性特定于实例的值或运行时状态等因素。

下面是一个非常简单的验证回叫方案的代码示例：验证类型化为 `Double` 基元的属性是否为 `PositiveInfinity` 或 `NegativeInfinity`。

C#

```
public static bool IsValidReading(object value)
{
    Double v = (Double)value;
    return (!v.Equals(Double.NegativeInfinity) &&
    !v.Equals(Double.PositiveInfinity));
}
```

## 强制值回叫和属性更改事件

强制值回叫传递属性的特定 `DependencyObject` 实例，在更改依赖属性的值时属性系统调用的 `PropertyChangedCallback` 实现也是如此。通过将这两种回叫组合使用，可以对元素创建一系列属性，其中，一个属性的更改会对另一个属性实施强制或重新计算。

使用依赖属性链接的典型方案是：具有用户界面驱动属性，其中，元素为最小值和最大值分别保留一个属性，为实际值或当前值保留第三个属性。此时，如果按照当前值超出新的最大值的方式调整最大值，则可能需要强制当前值不超过新的最大值，并对最小值与当前值强制类似的关系。

下面是一个非常简短的代码示例，仅针对阐释此关系的三个依赖属性之一。该示例演示如何注册相关的 \*Reading 属性的最小/最大/当前值集的 `CurrentReading` 属性。它使用上一节中所示的验证。

C#

```
public static readonly DependencyProperty CurrentReadingProperty =
DependencyProperty.Register(
    "CurrentReading",
    typeof(double),
    typeof(Gauge),
    new FrameworkPropertyMetadata(
        Double.NaN,
```

```

        FrameworkPropertyMetadataOptions.AffectsMeasure,
        new PropertyChangedCallback(OnCurrentReadingChanged),
        new CoerceValueCallback(CoerceCurrentReading)
    ),
    new ValidateValueCallback(IsValidReading)
);
public double CurrentReading
{
    get { return (double)GetValue(CurrentReadingProperty); }
    set { SetValue(CurrentReadingProperty, value); }
}

```

当前值的属性更改回叫用于将更改转发到其他依赖属性，方法是显式调用为这些属性注册的强制值回叫：

C#

```

private static void OnCurrentReadingChanged(DependencyObject d,
DependencyPropertyChangedEventArgs e)
{
    d.CoerceValue(MinReadingProperty);
    d.CoerceValue(MaxReadingProperty);
}

```

强制值回叫会检查当前属性可能依赖的属性的值，并在必要时强制当前值：

C#

```

private static object CoerceCurrentReading(DependencyObject d, object value)
{
    Gauge g = (Gauge)d;
    double current = (double)value;
    if (current < g.MinReading) current = g.MinReading;
    if (current > g.MaxReading) current = g.MaxReading;
    return current;
}

```

## ① 备注

不会强制属性的默认值。如果属性值仍然采用其初始默认值，或通过使用 `ClearValue` 清除其他值，则可能存在等于默认值的属性值。

强制值回叫和属性更改回叫属于属性元数据的一部分。因此，可以通过在类型上重写特定依赖属性的元数据来更改对该属性的回叫，因为其所在的类型派生自拥有该依赖属性的类型。

# 高级强制和回叫方案

## 约束和所需的值

属性系统将根据所声明的逻辑使用 [CoerceValueCallback](#) 回叫来强制值，但本地设置的属性的强制值仍然会在内部保留“所需的值”。如果约束基于可能会在应用程序生存期间动态更改的其他属性值，则强制约束也会进行动态更改，并且在给定新约束的情况下约束属性可更改其值以尽可能接近所需的值。如果解除所有约束，该值会成为所需的值。如果多个属性以循环方式相互依赖，则可能会引入某些相当复杂的依赖项方案。例如，在最小/最大/当前值方案中，可以选择将最小值和最大值作为用户可设置的资源。在这种情况下，可能需要强制最大值始终大于最小值，反之亦然。但是，如果该强制处于活动状态，并且最大值强制为最小值，则会将当前值保留在不可设置的状态，因为它依赖于二者，且被约束在这些值之间的范围内，即为零。这样，如果调整最大值或最小值，当前值可能会“沿用”这些值之一，因为仍然存储了当前值所需的值，并且当前值会在放松约束时尝试接近所需的值。

复杂依赖项没有任何技术错误，但是，如果它们需要大量重新计算，则可能会略微降低性能，而且还可能导致用户混淆（如果直接影响 UI）。请小心使用属性更改回叫和强制值回叫，确保尽可能地明确处理所尝试的强制，并且不会“过度约束”。

## 使用 CoerceValue 取消值更改

属性系统会将返回值 [UnsetValue](#) 的任何 [CoerceValueCallback](#) 视为特殊情况。此特殊情况意味着，属性系统应拒绝导致调用 [CoerceValueCallback](#) 的属性更改，并且还应报告该属性以前的任何值。该机制可用于检查异步启动的属性更改对当前对象状态是否仍然有效，如果无效，则可取消这些更改。另一个可能的方案是：可以根据负责所报告的值的属性值确定组件，有选择地取消该值。为此，可以使用回叫中传递的 [DependencyProperty](#) 和属性标识符用作 [GetValueSource](#) 的输入，然后处理 [ValueSource](#)。

## 另请参阅

- [依赖项属性概述](#)
- [依赖属性元数据](#)
- [自定义依赖属性](#)

# 框架属性元数据

项目 · 2023/02/06

对于 WPF 演示 API 和可执行文件中被视为 WPF 框架级的对象元素的属性，将报告框架属性元数据选项。这些系统对框架属性元数据进行查询，以便为特定的元素属性确定特定于功能的特征。

## 先决条件

本主题假设你作为 Windows Presentation Foundation (WPF) 类上现有依赖属性的使用者已经对依赖属性有所了解，并且已阅读了[依赖属性概述](#)。另外，应当已阅读[依赖属性元数据](#)。

## 框架属性元数据传达的信息

框架属性元数据分为以下类别：

- 报告影响某个元素的布局属性 ([AffectsArrange](#)、[AffectsMeasure](#)、[AffectsRender](#))。如果属性对这些相应的方面存在影响并且你还在你的类中实现了 [MeasureOverride](#) / [ArrangeOverride](#) 方法以向布局系统提供特定的呈现行为和信息，则可以在元数据中设置这些标志。通常，如果在属性元数据中有布局属性为 true，则此类实现会在依赖属性中检查属性是否无效，并且只有无效属性才必须请求新的布局处理过程。
- 报告影响元素的父元素的布局属性 ([AffectsParentArrange](#)、[AffectsParentMeasure](#))。默认设置了这些标志的部分示例包括 [FixedPage.Left](#) 和 [Paragraph.KeepWithNext](#)。
- [Inherits](#)。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。

### ① 备注

属性值上下文中的术语“继承”是指特定于依赖属性的情况；它意味着由于存在 WPF 属性系统的 WPF 框架级功能，子元素可以从父元素继承实际的依赖属性值。它与通过派生类型的托管代码类型和成员继承没有直接关系。有关详细信息，请参阅[属性值继承](#)。

- 报告数据绑定特征 ([IsNotDataBindable](#)、[BindsTwoWayByDefault](#))。默认情况下，框架中的依赖属性支持具有单向绑定行为的数据绑定。如果数据绑定没有适用

的方案，则可以禁用数据绑定（因为这些属性应该是灵活且可扩展的，所以在默认的 WPF API 中没有太多此类属性的示例）。可以针对以下属性将绑定设置为具有双向默认值：将控件在其组件部分中的各种行为绑定在一起的属性（例如 [IsSubmenuOpen](#)）；或者双向绑定对用户来说是常见且期望的方案的属性（例如 [Text](#)）。更改与数据绑定相关的元数据只会影响默认值；始终可以根据各个绑定更改此默认值。若要详细了解常规的绑定模式和绑定，请参阅[数据绑定概述](#)。

- 报告支持日记 ([Journal](#)) 的应用程序或服务是否应对属性进行日记记录。对于一般的元素，默认情况下不会启用日记记录功能，但可针对特定用户输入控件有选择性地启用。此属性旨在由日记的 WPF 实现等日记服务读取，并且通常针对用户控件（例如应在各导航步骤中保留的列表中的用户选择）而设置。有关日记的信息，请参阅[导航概述](#)。

## 读取 FrameworkPropertyMetadata

以上链接的各个属性都是 [FrameworkPropertyMetadata](#) 向其直接基类 [UIPropertyMetadata](#) 添加的特定属性。默认情况下，这些属性都为 `false`。在必须了解这些属性值的情况下，针对某一属性的元数据请求应尝试将返回的元数据强制转换为 [FrameworkPropertyMetadata](#)，然后根据需要检查各个属性的值。

## 指定元数据

如果出于将元数据应用到新依赖属性注册的目的而创建新的元数据实例，则可以选择要使用的元数据类：基 [PropertyMetadata](#) 或者某些派生类，例如 [FrameworkPropertyMetadata](#)。通常，应使用 [FrameworkPropertyMetadata](#)，尤其是在你的属性与属性系统和 WPF 功能（例如布局和数据绑定）存在任何交互的情况下。针对更为复杂的方案，还可以从 [FrameworkPropertyMetadata](#) 派生以创建自己的元数据报告类，使其成员承载额外的信息。或者，可以使用 [PropertyMetadata](#) 或 [UIPropertyMetadata](#) 来传达对实现的功能的支持程度。

对于现有属性（[AddOwner](#) 或 [OverrideMetadata](#) 调用），应始终使用原始注册所用的元数据类型来替代。

如果正在创建 [FrameworkPropertyMetadata](#) 实例，可以通过以下两种方式使用传达框架属性特征的特定属性的值来填充该元数据：

1. 使用允许 `flags` 参数的 [FrameworkPropertyMetadata](#) 构造函数签名。此参数应使用 [FrameworkPropertyMetadataOptions](#) 枚举标志的全部所需合并值来填充。
2. 使用无 `flags` 参数的其中一个签名，然后将 [FrameworkPropertyMetadata](#) 上的每个报告布尔属性设置为 `true`，以获得每个所需的特征更改。为此，必须在构造具

有此依赖属性的任何元素之前对这些属性进行设置；布尔属性是可读写属性，以允许此项避免 `flags` 参数的行为，同时仍然可以填充元数据，但元数据必须在属性使用之前进行有效密封。因此，尝试在请求元数据之后设置属性是无效操作。

## 框架属性元数据合并行为

重写框架属性元数据时，会合并或替换不同的元数据特征。

- 会合并 `PropertyChangedCallback`。如果添加一个新的 `PropertyChangedCallback`，该回叫则存储在元数据中。如果没有在替代中指定 `PropertyChangedCallback`，`PropertyChangedCallback` 的值则会从在元数据中指定它的最近上级提升为一个引用。
- `PropertyChangedCallback` 的实际属性系统行为是：层次结构中所有元数据所有者的实现都保留并添加到表中，属性系统的执行顺序是首先调用最深派生类的回叫。继承的回叫仅运行一次，当将这些回叫放置在元数据中的类占有时计数。
- 会替换 `DefaultValue`。如果没有在替代中指定 `PropertyChangedCallback`，`DefaultValue` 的值则来自在元数据中指定它的最近上级。
- 会替换 `CoerceValueCallback` 实现。如果添加一个新的 `CoerceValueCallback`，该回叫则存储在元数据中。如果没有在替代中指定 `CoerceValueCallback`，`CoerceValueCallback` 的值则会从在元数据中指定它的最近上级提升为一个引用。
- 属性系统的行为是仅调用直接元数据中的 `CoerceValueCallback`。不保留对层次结构中其他 `CoerceValueCallback` 实现的引用。
- `FrameworkPropertyMetadataOptions` 枚举的标志组合为按位 OR 运算。如果指定 `FrameworkPropertyMetadataOptions`，则不会替代原始选项。若要更改某个选项，请在 `FrameworkPropertyMetadata` 上设置相应的属性。例如，如果原始 `FrameworkPropertyMetadata` 对象设置 `FrameworkPropertyMetadataOptions.NotDataBindable` 标志，则可以通过将 `FrameworkPropertyMetadata.IsNotDataBindable` 设置为 `false` 来进行更改。

此行为由 `Merge` 实现，并且可以在派生的元数据类上替代。

## 另请参阅

- [GetMetadata](#)
- [依赖属性元数据](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)

# 依赖项属性值优先级

项目 • 2023/02/06

本主题说明 Windows Presentation Foundation (WPF) 属性系统的工作机制如何影响依赖属性的值，并介绍应用于属性有效值的属性系统的各方面所依据的优先级。

## 先决条件

本主题假定你从 WPF 类的现有依赖属性的使用者角度了解依赖属性，并且已阅读[依赖属性概述](#)。若要采用本主题中的示例，还应当了解 WPF 应用程序。

## WPF 属性系统

WPF 属性系统提供一种强大的方法，使得依赖属性的值由多种因素决定，从而实现诸如实时属性验证、后期绑定以及向相关属性发出有关其他属性值发生更改的通知等功能。用来确定依赖属性值的确切顺序和逻辑相当复杂。了解此顺序有助于避免不必要的属性设置，并且还有可能澄清混淆，使你正确了解为何某些影响或预测依赖属性值的尝试最终却没有得出所期望的值。

## 依赖属性可以在多个位置“设置”

在下面的示例 XAML 中，同一个属性 ([Background](#)) 具有三个不同的“设置”操作，它们都可能会影响属性值。

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>
```

这里，你希望应用什么颜色：红色、绿色还是蓝色？

本地属性集在设置时具有最高优先级，动画值和强制除外。如果在本地设置某个值，你可以期待该值优先得到应用，甚至期待其优先级高于任何样式或控件模板。在本示例中，`Background` 在本地设置为“Red”。因此，在此范围内定义的样式（即使是可能会应用于该范围内该类型的所有元素的隐式样式）在向 `Background` 属性提供其值时并没有最高优先级。如果从该 `Button` 实例中删除本地值“Red”，样式将获得优先级，而按钮将从该样式中获得 `Background` 值。在该样式中，触发器具有优先级，因此当鼠标位于按钮上时，按钮为蓝色，其他情况下则为绿色。

## 依赖属性设置优先级列表

下面是属性系统在分配依赖属性的运行时值时所使用的最终顺序。最高优先级最先列出。此列表对[依赖属性概述](#)中的某些一般化内容进行了扩充。

1. **属性系统强制。** 有关强制的详细信息，请参阅本主题后面的[强制、动画和基值](#)。
2. **活动动画或具有 Hold 行为的动画。** 为了获得任何实用效果，属性的动画必须优先于基（未动画）值，即使该值是在本地设置的也是如此。有关详细信息，请参阅本主题后面的[强制、动画和基值](#)。
3. **本地值。** 本地值可以通过“包装器”属性的便利性进行设置，这也相当于在 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。如果使用绑定或资源来设置本地值，则每个值都按照直接设置值的优先级顺序来应用。
4. **TemplatedParent 模板属性。** 如果元素是作为模板（`ControlTemplate` 或 `DataTemplate`）的一部分创建的，则具有 `TemplatedParent`。有关何时应用此原则的详细信息，请参阅本主题后面的 `TemplatedParent`。在模板中，按以下优先级顺序应用：
  - a. 来自 `TemplatedParent` 模板的触发器。
  - b. `TemplatedParent` 模板中的属性集（通常通过 XAML 属性进行设置）。

5. **隐式样式。** 仅应用于 `Style` 属性。`Style` 属性是由任何样式资源通过与其类型匹配的键来填充的。该样式资源必须存在于页面或应用程序中；查找隐式样式资源不会进入到主题中。
6. **样式触发器。** 来自页面或应用程序的样式中的触发器（这些样式可以是显式或隐式样式，但不是来自优先级较低的默认样式）。
7. **模板触发器。** 来自样式中的模板或者直接应用的模板的任何触发器。
8. **样式资源库。** 来自页面或应用程序的样式中的 `Setter` 的值。
9. **默认（主题）样式。** 有关何时应用此样式以及主题样式如何与主题样式中的模板相关的详细信息，请参阅本主题后面的[默认（主题）样式](#)。在默认样式中，按以下优先级顺序应用：
  - a. 主题样式中的活动触发器。
  - b. 主题样式中的资源库。
10. **继承。** 有几个依赖属性从父元素向子元素继承值，因此不需要在应用程序中的每个元素上专门设置这些属性。有关详细信息，请参阅[属性值继承](#)。
11. **来自依赖属性元数据的默认值。** 任何给定的依赖属性都可能有一个默认值，它由该特定属性的属性系统注册来确定。而且，继承依赖属性的派生类可以选择按照类型重写该元数据（包括默认值）。有关详细信息，请参阅[依赖属性元数据](#)。因为继承是在默认值之前检查的，所以对于继承的属性，父元素的默认值优先于子元素。因此，如果任何地方都没有设置可继承的属性，将使用在根元素或父元素中指定的默认值，而不是子元素的默认值。

## TemplatedParent

TemplatedParent 作为一个优先级项并不应用于在标准应用程序标记中直接声明的元素的任何属性。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 TemplatetdParent 概念。当属性系统在 `TemplatedParent` 模板中搜索某个值时，就是在搜索创建该元素的模板。来自 `TemplatedParent` 模板的属性值通常就像在子元素上设置的本地值一样来应用，但是这些属性值的优先级低于本地值，因为模板有可能被共享。有关详细信息，请参阅 [TemplatedParent](#)。

## Style 属性

前面介绍的查找顺序适用于除 `Style` 属性之外的所有可能的依赖属性。`Style` 属性的独特之处在于它不能为自己设置样式，因此优先级项 5 到 8 不适用。此外，建议不要对 `Style`

进行动画处理或强制（对 `Style` 进行动画处理需要一个自定义动画类）。这产生了三种设置 `Style` 属性的方法：

- **显式样式。** `Style` 属性是直接设置的。在大多数情况下，样式不是内联定义的，而是作为资源由显式键进行引用的。在这种情况下，`Style` 属性本身就像本地值（优先级项 3）一样来应用。
- **隐式样式。** `Style` 属性不是直接设置的。但是，`Style` 在某种程度上存在于资源查找序列（页面、应用程序）中，并且使用与要应用样式的类型相匹配的资源键进行键控。在这种情况下，`Style` 属性本身按照在该序列中标识为第 5 项的优先级来应用。通过对 `Style` 属性使用 `DependencyPropertyHelper` 并在结果中查找 `ImplicitStyleReference`，可检测此条件。
- “默认样式”，也称为“主题样式”。`Style` 属性不是直接设置的，并且实际上将作为 `null` 读取，一直到运行时为止。在这种情况下，样式来自作为 WPF 表示引擎一部分的运行时主题评估。

对于主题中不存在的隐式样式，类型必须完全匹配 - `MyButton Button` 派生的类不会对 `Button` 隐式使用样式。

## 默认（主题）样式

WPF 附带的每个控件都有一个默认样式。默认样式可能因主题而异，这就是默认样式有时候称为主题样式的原因。

在默认样式中，对于控件最重要的信息就是其控件模板，它作为其 `Template` 属性的资源库存在于主题样式中。如果默认样式中没有模板，自定义样式中没有自定义模板的控件将根本没有可视化外观。默认样式中的模板为每个控件的可视化外观提供一个基本结构，还定义在模板的可视化树中定义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，而无需完全替换模板。例如，请考虑 `Thumb` 控件（`ScrollBar` 的一个组件）的默认可视化外观。

`Thumb` 具有一些可自定义属性。`Thumb` 的默认模板可创建一个基本结构/可视化树（具有几个嵌套的 `Border` 组件），以创建棱台外观。如果模板中的某个属性要通过 `Thumb` 类公开以进行自定义，则该属性必须通过 `TemplateBinding` 在模板中公开。对于 `Thumb`，这些边界的各种属性都共享对属性（如 `Background` 或 `BorderThickness`）的模板绑定。但是其他某些属性或可视化排列被硬编码到控件模板中，或者绑定到直接来自主题的值，除了替换整个模板外，不能对其进行更改。一般而言，如果属性来自模板化的父元素，并且不是通过模板绑定公开的，则不能通过样式进行调整，因为没有简单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的影响，或受默认值的影响。

主题样式将类型用作其定义中的键。但是，当主题应用于给定的元素实例时，主题将通过检查控件上是否具有 [DefaultStyleKey](#) 属性来查找此类型。这与使用文本类型的隐式样式正好相反。[DefaultStyleKey](#) 的值将由派生类继承，即使实施者未更改该属性也是如此（更改属性的理想方式不是在属性级别将其重写，而是在属性元数据中更改其默认值）。这种间接方式使基类可以为没有样式（或者该样式中没有模板，因此根本没有默认的可视化外观，这一点更为重要）的派生元素定义主题样式。所以，可以从 [Button](#) 派生 [MyButton](#) 并且仍然能够获得 [Button](#) 默认模板。如果你是 [MyButton](#) 的控件作者，而你需要一个不同的行为，则可以为 [MyButton](#) 上的 [DefaultStyleKey](#) 重写依赖属性元数据以返回一个不同的键，然后定义相关主题样式，包括必须与 [MyButton](#) 控件一起打包的 [MyButton](#) 的模板。有关主题、样式和控件创作的更多详细信息，请参阅[控件创作概述](#)。

## 动态资源引用和绑定

动态资源引用和绑定操作遵守其设置位置的优先级。例如，应用于本地值的动态资源作为优先级项 3 进行应用，主题样式中的属性资源库的绑定作为优先级项 9 进行应用，依此类推。由于动态资源引用和绑定必须都能够从应用程序的运行时状态中获得值，这使确定任何给定属性的属性值优先级的实际处理也将扩展到运行时。

严格来讲，动态资源引用并不是属性系统的一部分，但它们的确具有自己的查找顺序，该顺序与上面列出的序列交互。[XAML 资源](#)中对该优先级进行了更全面的介绍。该优先级的基本概括是：元素到页面根元素、应用程序、主题、系统。

动态资源和绑定具有其设置位置的优先级，但是值会延迟。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，该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（没有“文字”本地值）调用 [ClearValue](#)，它们也会被 [ClearValue](#) 调用清除掉。

## SetCurrentValue

[SetCurrentValue](#) 方法是设置属性的另一种方式，但它不具有优先级顺序。相反，使用 [SetCurrentValue](#) 可以更改属性值而不会覆盖以前值的源。每当要设置一个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](#)。例如，如果一个属性是通过触发器设置的，然后通过 [SetCurrentValue](#) 分配了另一个值，则属性系统仍将遵循该触发器，并且在触发器操作发生时该属性将更改。使用 [SetCurrentValue](#) 可以更改属性值而不必为其提供具有更高优先级的源。同样，可以使用 [SetCurrentValue](#) 更改属性值而不覆盖绑定。

## 强制、动画和基值

强制和动画在本 SDK 中都作用于称为“基值”的值。因此，基值是在各项中通过向上计算一直到第 2 项为止而确定的任何值。

对于动画，如果没有为某些行为指定“From”和“To”值，或者动画在完成时故意还原为基值，那么基值将影响动画值。若要了解实际效果，请运行 [From, To, and By Animation Target Values Sample](#)（From、To 和 By 动画目标值示例）。尝试为示例中的矩形高度设置本地值，使初始本地值不同于动画中的任何“From”值。你会注意到动画立即使用“From”值开始，并在开始后替换基值。可以指定动画在完成后返回到在动画前发现的值，具体操作是指定 Stop [FillBehavior](#)。然后，根据正常优先级来确定基值。

多个动画可能应用于一个属性，而每个动画可能是从值优先级中的不同点进行定义的。但是，这些动画的值可能会组合起来，而不仅仅是从较高的优先级开始应用动画。这完全取决于动画的定义方式以及进行动画处理的值类型。有关对属性进行动画处理的详细信息，请参阅[动画概述](#)。

强制在最高级别应用。即使正在运行的动画也会受到值强制的制约。WPF 中的某些现有依赖属性具有内置的强制行为。对于自定义依赖属性，可以在创建该属性时通过编写 [CoerceValueCallback](#) 并作为元数据的一部分传递回调来为其定义强制行为。还可以通过在派生类中重写现有属性的元数据来重写其强制行为。强制与基值的交互使强制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，如果强制约束后来被解除，强制将返回与基值最接近的值，并且一旦所有约束都解除，强制对属性的影响可能会立即停止。有关强制行为的详细信息，请参阅[依赖属性回调和验证](#)。

## 触发器行为

控件常常在主题中将触发器行为定义为其默认样式的一部分。为控件设置本地属性可能会阻止触发器从视觉或行为上响应用户驱动的事件。属性触发器最常用于控件或状态属性，如 [IsSelected](#)。例如，默认情况下，当禁用 [Button](#) 时 ([.IsEnabled](#) 的触发器是 `false`)，主题样式中的 [Foreground](#) 值会导致控件“灰显”。但是，如果设置了本地 [Foreground](#) 值，本地属性集将优先于普通灰显颜色，即使在这个属性触发的方案中也是如此。为具有主题级触发器行为的属性设置值时要倍加小心，并确保不要过度妨碍该控件应有的用户体验。

## ClearValue 和值优先级

[ClearValue](#) 方法为从在元素上设置的依赖属性中清除任何本地应用的值提供了一个有利的途径。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。值优先级中的所有其他参与者仍然有效。只有在本地设置的值才会从优先级序列中删除。例如，如果对同时也由主题样式设置的属性调用 [ClearValue](#)，主题值将作为新值而不是基于元数据的默认值进行应用。如果希望取消过程中的所有属性值参与者，

而将值设置为注册的元数据默认值，则可以通过查询依赖属性元数据最终获得默认值，然后使用该默认值在本地设置属性并调用 [SetValue](#)。

## 另请参阅

- [DependencyObject](#)
- [DependencyProperty](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [依赖属性回调和验证](#)

# 只读依赖项属性

项目 · 2023/02/06

本主题介绍只读依赖属性，包括现有只读依赖属性、创建自定义只读依赖属性的方案和技术。

## 先决条件

本主题假定你了解实现依赖属性的基本方案，以及如何将元数据应用于自定义依赖属性。有关上下文，请参阅[自定义依赖属性](#)和[依赖属性元数据](#)。

## 现有只读依赖属性

Windows Presentation Foundation (WPF) 框架中定义的某些依赖项属性是只读的。指定只读依赖属性的一般原因如下：这些属性应该用于状态确定，但是有多种因素影响该状态，从用户界面设计的角度看，仅将属性设置为该状态并不能达到预期的效果。例如，属性 `IsMouseOver` 实际上只是结合从鼠标输入确定的状态。任何通过避开实际的鼠标输入以编程方式设置此值的尝试都是不可预期的，并将导致不一致。

由于其不可设置性，只读依赖属性不适用于依赖属性通常为其提供一个解决方案（即：数据绑定，直接样式化为某个值、验证、动画和继承）的多种方案。尽管不可设置，但只读依赖属性仍具有一些由属性系统中的依赖属性支持的其他功能。只读依赖属性仍可以用作样式中的属性触发器，这是其他功能中最重要的功能。无法使用常规的公共语言运行时 (CLR) 属性启用触发器；必须使用依赖属性才行。之前提到的 `IsMouseOver` 属性是以下方案的最佳示例：定义控件的样式非常有用；用户将鼠标悬停在控件的一些已定义区域上方时，控件内复合元素的一些可见属性（例如背景、前景）或类似属性将发生更改。只读依赖属性中的更改还可以由属性系统的固有失效进程检测并报告，这实际上是在内部支持属性触发器功能。

## 创建自定义只读依赖属性

请务必阅读上一节中有关只读依赖属性为何不适用于许多典型依赖属性方案的内容。但是如果有适当的方案，可能需要创建自己的只读依赖属性。

创建只读依赖属性的大多数过程与[自定义依赖属性](#)和[实现依赖属性](#)主题中介绍的过程相同。但有三个重要的差异：

- 注册属性时，调用 `RegisterReadOnly` 方法，而不是常规的 `Register` 方法。

- 实现 CLR“包装器”属性时，请确保该包装器也没有设置的实现，以便在公开的公共包装器的只读状态中不存在不一致现象。
- 由只读注册返回的对象是 [DependencyPropertyKey](#) 而不是 [DependencyProperty](#)。仍应将该字段存储为成员，但通常不将其设置为类型的公共成员。

无论你具有什么专用字段或值，可使用你确定的任何逻辑来完全编写对只读依赖属性的支持。但是，在最初或运行时逻辑过程中设置属性的最简单方法是使用属性系统的 API，而不是避开属性系统并直接设置专有支持字段。特别是在存在接受 [DependencyPropertyKey](#) 类型参数的 [SetValue](#) 的签名的情况下更是如此。在应用程序逻辑内以编程方式设置此值的方式和位置，会影响设置访问首次注册依赖属性时创建的 [DependencyPropertyKey](#) 的方式。如果完全在专有类中处理此逻辑，或者如果要求从程序集的其他部分对其进行设置，可以在内部进行设置。一种方法是在相关事件的类事件处理程序内调用 [SetValue](#)，该事件会通知类实例需要更改存储的属性值。另一种方法是通过在注册期间将成对的 [PropertyChangedCallback](#) 和 [CoerceValueCallback](#) 回叫用作这些属性元数据的一部分来将依赖属性关联在一起。

因为 [DependencyPropertyKey](#) 为专有，并且不是由代码外部的属性系统传播的，所以只读依赖属性的设置安全性确实比读写依赖属性高。对于读写依赖属性，标识字段是显式或隐式公用的，因此该属性可广泛设置。有关更多详细信息，请参阅[依赖属性的安全性](#)。

## 另请参阅

- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [样式设置和模板化](#)

# 属性值继承

项目 · 2023/02/06

属性值继承是 Windows Presentation Foundation (WPF) 属性系统的一项功能。属性值继承使元素树中的子元素可以从父元素获取特定属性的值，并继承该值，就如同它是在最近的父元素中任意位置设置的一样。父元素可能也已通过属性值继承获得了其值，因此系统有可能一直递归到页面根。属性值继承不是默认属性系统行为；属性必须用特定的元数据设置来建立，以便使该属性对子元素启动属性值继承。

## 属性值继承是内含继承

此处作为术语的“继承”与类型和常规的面向对象的编程中提到的继承并非完全相同的概念，后者指派生类从其基类继承成员定义。继承的这一含义在 WPF 中也适用：当在代码中用作元素且公开为成员时，各基类中定义的属性将公开为 XAML 派生类的特性。属性值继承则是关于属性值如何基于元素树中的父子关系从一个元素继承到另一个元素。如果在 XAML 标记中定义应用程序时将元素嵌套在其他元素中，则该元素树在此种情况下最直接可见。还可以通过向其他对象的指定集合中添加对象来以编程方式创建对象树，在运行时，属性值继承在完成树中以相同方式工作。

## 属性值继承的实际应用

WPF API 包括几个启用了属性继承的属性。通常，使用这些属性的情况是当涉及到一个属性，每页可以仅对该属性设置一次，但是该属性还是某个基元素类的成员，因此还存在于大多数子元素中。例如，[FlowDirection](#) 属性控制流式内容应当沿哪个方向在页面上呈现和排列。通常做法是在所有子元素中以一致的方式处理文本流概念。如果用户或环境操作因某种原因在元素树的某一层重置了流方向，则流方向通常会在整个树中重置。在将 [FlowDirection](#) 属性设置为继承后，只需在应用程序的元素树中需要演示的每页的层处设置或重置一次该值即可。即使是最初的默认值也将按照这种方式继承。在需有意混用流方向的极罕见情况下，属性值继承模型也仍允许个别元素重置该值。

## 使自定义属性可继承

通过更改自定义属性的元数据，还可以使自己的自定义属性可继承。但是，请注意，将属性指定为可继承需要考虑到性能问题。如果该属性没有已建立的本地值或通过样式、模板或数据绑定获取的值，则可继承的属性会将赋予它的属性值提供给逻辑树中的所有子元素。

为了让属性参与值继承，请按照[注册附加属性](#)中所述创建自定义附加属性。向元数据([FrameworkPropertyMetadata](#)) 注册属性，并在该元数据内的选项设置中指定“Inherits”选

项。同时确保该属性已建立了默认值，因为该值现将继承。尽管已将该属性注册为附加属性，可能还需像对“非附加”依赖项属性一样，为所有者类型上的 get/set 访问创建属性“包装”。然后，可通过对所有者类型或派生类型使用直接属性包装或通过对任意 [DependencyObject](#) 使用附加属性语法，来设置可继承属性。

附加属性在概念上与全局属性类似；可检查任意 [DependencyObject](#) 上的值并获取有效的结果。附加属性的典型方案是针对子元素设置属性值，如果所涉及的属性是一个在树中的每个元素 ([DependencyObject](#)) 上都始终作为附加属性存在的附加属性，则该方案会更为有效。

### 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。如果要在元数据中指定 `Inherits`，请始终使用 `RegisterAttached` 来注册属性。

## 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。在使用 [LogicalTreeHelper](#) 的方法时，可发现这一点反映在了结果中。但是，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](#)），那么属性值继承就可以弥合逻辑树中的这种差异，并且仍可遍历传递所继承的值。

## 另请参阅

- [依赖属性元数据](#)
- [附加属性概述](#)
- [依赖项属性值优先级](#)

# 依赖项属性的安全性

项目 · 2023/02/06

依赖属性通常应当被视为公共属性。 Windows Presentation Foundation (WPF) 属性系统在本质上无法对依赖属性值提供安全保证。

## 包装器和依赖属性的访问和安全性

通常，依赖属性是随“包装器”公共语言运行时 (CLR) 属性一起实现的，这些包装器属性可以简化从实例获取或设置属性的过程。但包装器实际上只是实现与依赖属性交互时所使用的基础 `GetValue` 和 `SetValue` 静态调用的便捷方法。从另外一个角度考虑，这些属性作为恰好由依赖属性（而非私有字段）支持的公共语言运行时 (CLR) 属性公开。应用于包装器的安全机制与属性系统行为以及基础依赖属性的访问并不可比。对包装器提出安全要求仅阻止使用便捷方法，但不会阻止对 `GetValue` 或 `SetValue` 的调用。同样，为包装器设置受保护或私有访问级别也不会提供任何有效的安全性。

如果要编写自己的依赖属性，应当将包装器和 `DependencyProperty` 标识符字段声明为公共成员，以便调用方（由于它的存储是作为依赖属性实现的）不会获得有关该属性的实际访问级别的误导信息。

对于自定义依赖属性，可以将属性注册为只读依赖属性，而且这确实提供了一种有效的方法来防止属性被任何不拥有对该属性的 `DependencyPropertyKey` 的引用的人设置。有关详细信息，请参阅[只读依赖属性](#)。

### ① 备注

不禁止将 `DependencyProperty` 标识符字段声明为私有字段，而且可以放心地使用这种方法来帮助减小自定义类的直接公开命名空间，但是这种属性的“私有”性质不应当被视为与公共语言运行时 (CLR) 语言定义所定义的私有访问级别一样，具体原因将在下一节介绍。

## 依赖属性的属性系统公开

将 `DependencyProperty` 声明为公共级别以外的任何访问级别通常毫无用处而且可能会产生误导作用。该访问级别设置只是防止某人从声明类获得对实例的引用。但是，属性系统有几个方面会返回 `DependencyProperty` 作为一种标识在类实例或派生类实例上存在的特定属性的方式，而且该标识符在 `SetValue` 调用中仍可用，即使最初的静态标识符声明为非公共也是如此。此外，`OnPropertyChanged` 虚拟方法接收任何更改值的现有依赖属

性的信息。此外，[GetLocalValueEnumerator](#) 方法返回具有本地设置值的实例上的任何属性的标识符。

## 验证和安全

下面的安全机制不足以保证安全：向 [ValidateValueCallback](#) 应用一个要求，并期望在该要求未满足时验证失败以防止设置属性。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 set-value 失效。

## 另请参阅

- [自定义依赖属性](#)

# DependencyObject 的安全构造函数模式

项目 · 2023/02/06

通常，类构造函数不应调用诸如虚拟方法或委托等回调，其原因是构造函数可作为派生类的构造函数的基本初始化进行调用。输入该虚拟的操作可能会在任何给定对象的不完全初始化状态下进行。但是，属性系统本身在内部调用并公开回调，作为依赖属性系统的一部分。某些简单操作，例如使用 [SetValue](#) 调用来设置依赖属性值，就可能在决定中的某处包括回调。因此，在构造函数体内设置依赖属性值时应保持谨慎（将类型用作基类可能会导致问题）。存在一种特定的模式，用以实现可避免依赖属性状态和内在回调特定问题的 [DependencyObject](#) 构造函数，此处对其进行了说明。

## 属性系统虚拟方法

在计算用于设置依赖属性值的 [SetValue](#) 调用期间，可能会调用以下虚拟方法或回调：[ValidateValueCallback](#)、[PropertyChangedCallback](#)、[CoerceValueCallback](#)、[OnPropertyChanged](#)。这些虚拟方法或回调中的每一个在扩展 Windows Presentation Foundation (WPF) 属性系统和依赖属性的多样性方面都具有特定的用途。有关如何使用这些虚拟方法对属性值确定进行自定义的详细信息，请参阅[依赖属性回调和验证](#)。

## FXCop 规则强制与属性系统虚方法

如果将 Microsoft 工具 FXCop 用作生成过程的一部分，并从某些调用基构造函数的 WPF 框架类派生，或在派生的类上实现自己的依赖属性，则可能会违反某个 FXCop 规则。此违反事件的名称字符串为：

`DoNotCallOverridableMethodsInConstructors`

此规则是为 FXCop 设置的默认公共规则的一部分。此规则所报告的内容可能为通过依赖属性系统实现的跟踪情况，该系统会最终调用依赖属性系统虚拟方法。即使遵循本主题中介绍并建议使用的构造函数模式，仍可能存在此规则冲突，因此可能需要在 FXCop 规则设置配置中禁用或取消该规则。

## 大多数问题是因派生类导致的，而不是因为使用现有类所致

当按构造顺序使用虚拟方法实现一个类后从该类进行派生时，便会发生此规则所报告的问题。如果密封了类，或者确信不会派生自此类或强制不得派生自此类，则此处介绍的注意事项和触发 FXCop 规则的问题将不适用。但是，如果出于将类用作基类的目的来创作类，例如，如果正创建模板或可扩展的控件库集，则应遵循此处建议的用于构造函数的模式。

# 默认构造函数必须初始化由回叫请求的所有值

由类重写或回叫（来自“属性系统虚拟方法”部分列表中的回叫）使用的任何实例成员必须在类的默认构造函数中进行初始化，即使其中的某些值通过非默认构造函数的参数填充为“真实”值时也是如此。

以下代码示例（和后面的示例）是一个与此规则冲突的伪 C# 示例，其中对问题进行了说明：

C#

```
public class MyClass : DependencyObject
{
    public MyClass() {}
    public MyClass(object toSetWobble)
        : this()
    {
        Wobble = toSetWobble; //this is backed by a DependencyProperty
        _myList = new ArrayList(); // this line should be in the default
        ctor
    }
    public static readonly DependencyProperty WobbleProperty =
        DependencyProperty.Register("Wobble", typeof(object),
        typeof(MyClass));
    public object Wobble
    {
        get { return GetValue(WobbleProperty); }
        set { SetValue(WobbleProperty, value); }
    }
    protected override void
    OnPropertyChanged(DependencyPropertyChangedEventArgs e)
    {
        int count = _myList.Count; // null-reference exception
    }
    private ArrayList _myList;
}
```

当应用程序代码调用 `new MyClass(objectvalue)` 时，这会调用无参数构造函数和基类构造函数。然后它会设置 `Property1 = object1`，后者对所具有的 `MyClass DependencyObject` 调用虚拟方法 `OnPropertyChanged`。重写引用尚未初始化的 `_myList`。

避免这些问题的一个方法是：确保回叫仅使用其他依赖属性，并且每个这样的依赖属性都有一个确立的默认值作为其注册元数据的一部分。

## 安全构造函数模式

若要在将类用作基类的情况下规避未完全实现初始化的风险，请遵循以下模式：

## 用于调用基本初始化的无参数构造函数

实现以下用于调用基本默认值的构造函数：

C#

```
public MyClass : SomeBaseClass {
    public MyClass() : base() {
        // ALL class initialization, including initial defaults for
        // possible values that other ctors specify or that callbacks need.
    }
}
```

## 非默认（方便）构造函数，不与任何基本签名匹配

如果这些构造函数使用参数来设置初始化中的依赖属性，首先应调用自己的类无参数构造函数进行初始化，然后使用参数来设置依赖属性。这些可能是类所定义的依赖属性，也可能是在基类继承的依赖属性，但在这两种情况下都使用以下模式：

C#

```
public MyClass : SomeBaseClass {
    public MyClass(object to SetProperty1) : this() {
        // Class initialization NOT done by default.
        // Then, set properties to values as passed in ctor parameters.
        Property1 = to SetProperty1;
    }
}
```

## 非默认（方便）构造函数，确实与基本签名匹配

不调用具有相同参数设置的基构造函数，而是再次调用自己的类的无参数构造函数。请勿调用基本初始值设定项，而应调用 `this()`。然后使用传递的参数作为用于设置相关属性的值来重现原始构造函数行为。将原始基本构造函数文档用作指导依据，确定要对特定参数设置的属性：

C#

```
public MyClass : SomeBaseClass {
    public MyClass(object to SetProperty1) : this() {
        // Class initialization NOT done by default.
        // Then, set properties to values as passed in ctor parameters.
        Property1 = to SetProperty1;
    }
}
```

```
    }  
}
```

## 必须匹配所有签名

对于基类型具有多个签名的情况，必须在设置更多属性之前特意将所有可能的签名与自己的构造函数的实现相匹配，该实现使用建议的类无参数构造函数调用模式。

## 使用 SetValue 设置依赖属性

如果设置的属性不具有便于属性设置的包装，并使用 [SetValue](#) 设置值，这些模式同样适用。通过构造函数参数对 [SetValue](#) 的调用还应调用类的无参数构造函数以进行初始化。

## 另请参阅

- [自定义依赖属性](#)
- [依赖项属性概述](#)
- [依赖属性的安全性](#)

# 集合类型依赖项属性

项目 · 2023/02/06

本主题就如何实现属性类型为集合类型的依赖属性提供相应指导和建议模式。

## 实现集合类型依赖属性

对于一般的依赖属性，应遵循的实现模式是定义 CLR 属性包装器，其中该属性由 [DependencyProperty](#) 标识符（而非字段或其他构造）提供支持。实现集合类型属性时也应按照此相同模式。但是，当集合中包含的类型本身为 [DependencyObject](#) 或 [Freezable](#) 派生类，则集合类型属性会增加此模式的复杂程度。

## 不使用默认值初始化集合

创建依赖属性时，不要将属性默认值指定为初始字段值。相反，应通过依赖属性元数据指定默认值。如果属性为引用类型，则依赖属性元数据中指定的默认值不是每个实例分别的默认值，而是应用到类型的所有实例的默认值。因此，对于类型的新创建实例，必须小心，不要将集合属性元数据定义的单个静态集合用作工作默认值。相反，必须确保有意将集合值设置为唯一（实例）集合，作为类构造函数逻辑的一部分。否则，你会创建一个不需要的单例类。

请看下面的示例。本示例的以下部分显示了类 [Aquarium](#) 的定义，该类包含具有默认值的缺陷。该类定义了集合类型依赖项属性 [AquariumObjects](#)，该属性使用具有 [FrameworkElement](#) 类型约束的泛型 [List<T>](#) 类型。在对依赖属性的 [Register\(String, Type, Type, PropertyMetadata\)](#) 调用中，元数据将建立默认值作为新的泛型 [List<T>](#)。

### ⚠ 警告

以下代码的行为不正确。

C#

```
public class Fish : FrameworkElement { }
public class Aquarium : DependencyObject {
    private static readonly DependencyPropertyKey
AquariumContentsPropertyKey =
    DependencyProperty.RegisterReadOnly(
        "AquariumContents",
        typeof(List<FrameworkElement>),
        typeof(Aquarium),
        new FrameworkPropertyMetadata(new List<FrameworkElement>())
    );
}
```

```

public static readonly DependencyProperty AquariumContentsProperty =
    AquariumContentsPropertyKey.DependencyProperty;

public List<FrameworkElement> AquariumContents
{
    get { return
        (List<FrameworkElement>)GetValue(AquariumContentsProperty); }
}

// ...
}

```

但是，如果仅如上所示保留代码，该单一列表默认值会对 `Aquarium` 的所有实例共享。如果运行以下测试代码（用于演示如何实例化两个单独的 `Aquarium` 实例并向二者皆添加一个不同的 `Fish`），则其结果可能出乎意料：

C#

```

Aquarium myAq1 = new Aquarium();
Aquarium myAq2 = new Aquarium();
Fish f1 = new Fish();
Fish f2 = new Fish();
myAq1.AquariumContents.Add(f1);
myAq2.AquariumContents.Add(f2);
MessageBox.Show("aq1 contains " + myAq1.AquariumContents.Count.ToString() +
    " things");
MessageBox.Show("aq2 contains " + myAq2.AquariumContents.Count.ToString() +
    " things");

```

每个集合具有两个计数，而不是一个！这是因为，每个 `Aquarium` 将其 `Fish` 添加到默认值集合，此默认值集合因元数据中单个构造函数调用而产生，因此会在所有实例之间共享。而你全然不希望出现这种情况。

为纠正此问题，必须将集合依赖属性值重置为唯一实例，作为类构造函数调用的一部分。因此该属性为只读依赖属性，使用 `SetValue(DependencyPropertyKey, Object)` 方法进行设置，同时使用仅在类内部可访问的 `DependencyPropertyKey`。

C#

```

public Aquarium() : base()
{
    SetValue(AquariumContentsPropertyKey, new List<FrameworkElement>());
}

```

现在，如果再次运行此相同测试代码，则每个 `Aquarium` 会仅支持自己的唯一集合，这样的结果更符合预期。

如果选择集合属性为读写，则此模式会稍有变化。在这种情况下，可以从构造函数调用公共集访问器来进行初始化，这仍会在集合包装器内调用 [SetValue\(DependencyProperty, Object\)](#) 的非键签名，方法是使用公共 [DependencyProperty](#) 标识符。

## 报告集合属性中的绑定值更改

本身为依赖属性的集合属性不会自动将更改报告给其子属性。如果将绑定创建到集合，这可阻止绑定报告更改，从而使某些数据绑定方案无效。但是，如果使用集合类型 [FreezableCollection<T>](#)，则会正常报告集合中所包含元素的子属性更改，绑定也会按预期工作。

若要在依赖对象集合中启用子属性绑定，请将集合属性创建为类型 [FreezableCollection<T>](#)，其中该集合具有针对任何 [DependencyObject](#) 派生类的类型约束。

## 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](#)

# XAML 加载和依赖项属性

项目 • 2023/02/06

在加载二进制 XAML 和处理作为依赖属性的特性时，WPF XAML 处理器的当前 WPF 实现使用依赖属性的属性系统方法。这会有效地跳过属性包装器。实现自定义依赖属性时，必须要考虑到此行为，并应避免在属性包装器中放入除属性系统方法 `GetValue` 和 `SetValue` 以外的任何其他代码。

## 先决条件

本主题假定你已从使用者和创作者角度了解依赖属性，并已阅读[依赖属性概述](#)和[自定义依赖属性](#)。你也应已阅读[WPF 中的 XAML](#)和[XAML 语法详述](#)。

## WPF XAML 加载程序实现和性能

出于实现原因，相较于使用属性包装器和资源库，将属性标识为依赖属性并通过访问属性系统 `SetValue` 方法对其进行设置的计算成本较低。这是因为 XAML 处理器必须仅通过了解由标记结构和多种字符串所指示的类型和成员关系来推断支持代码的整个对象模型。

虽然此类型可通过结合 `xmlns` 和程序集属性来查询，但标识成员、确定支持设置为属性的项和解析属性值支持的类型则需要使用  `PropertyInfo` 的扩展反射。因为给定类型的依赖属性通过属性系统可作为存储表进行访问，因此 XAML 处理器的 WPF 实现使用此表，并推断出可通过使用依赖属性标识符 `ABCProperty` 在所含 `DependencyObject` 派生类型上调用 `SetValue` 来更高效设置任何给定属性 `ABC`。

## 自定义依赖属性的影响

由于进行属性设置的 XAML 处理器行为的当前 WPF 实现会完全跳过包装器，因此对于自定义依赖属性不应将任何其他逻辑放入包装器的设置定义中。如果将此类逻辑放入设置定义，则在 XAML 而不是代码中设置属性时不会执行逻辑。

同样，从 XAML 处理获取属性值的 XAML 处理器的其他方面也会使用 `GetValue` 而不是使用包装器。因此，在超出 `GetValue` 调用的 `get` 定义中，还应避免任何其他实现。

如下示例是包装器的推荐依赖属性定义，其中属性标识符存储为 `public static readonly` 字段，`get` 和 `set` 定义不包含除定义依赖属性支持所需属性系统方法外的任何代码。

C#

```
public static readonly DependencyProperty AquariumGraphicProperty =
DependencyProperty.Register(
    "AquariumGraphic",
    typeof(Uri),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(null,
        FrameworkPropertyMetadataOptions.AffectsRender,
        new PropertyChangedCallback(OnUriChanged)
    )
);
public Uri AquariumGraphic
{
    get { return (Uri)GetValue(AquariumGraphicProperty); }
    set { SetValue(AquariumGraphicProperty, value); }
}
```

## 另请参阅

- [依赖项属性概述](#)
- [WPF 中的 XAML](#)
- [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](#)

# 属性帮助主题

项目 · 2023/02/06

## 本节内容

- [实现依赖属性](#)
- [添加依赖属性的所有者类型](#)
- [注册附加属性](#)
- [重写依赖属性的元数据](#)

## 参考

- [DependencyProperty](#)
- [PropertyMetadata](#)
- [FrameworkPropertyMetadata](#)
- [DependencyObject](#)

## 相关章节

- [属性](#)

# 如何：实现依赖属性

项目 • 2023/02/06

本示例演示如何使用 `DependencyProperty` 字段来支持公共语言运行时 (CLR) 属性，从而定义一个依赖属性。定义自己的属性并需要其支持 Windows Presentation Foundation (WPF) 功能的诸多方面（包括样式、数据绑定、继承、动画和默认值）时，应将其作为依赖属性实现。

## 示例

下面的示例通过调用 `Register` 方法首次注册依赖属性。用于存储依赖属性名称和特征的标识符字段名称必须是作为 `Register` 调用的一部分为依赖属性选择的 `Name`，并追加字符串 `Property`。例如，如果使用 `Location` 的 `Name` 注册一个依赖属性，则为依赖属性定义的标识符字段必须命名为 `LocationProperty`。

在此示例中，依赖属性的名称及其 CLR 访问器均为 `State`；标识符字段为 `StateProperty`；属性类型为 `Boolean`；注册依赖属性的类型为 `MyStateControl`。

如果不遵循此命名模式，则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。

还可为依赖属性指定默认元数据。此示例将 `State` 依赖属性的默认值注册为 `false`。

```
C#  
  
public class MyStateControl : ButtonBase  
{  
    public MyStateControl() : base() { }  
    public Boolean State  
    {  
        get { return (Boolean)this.GetValue(StateProperty); }  
        set { this.SetValue(StateProperty, value); }  
    }  
    public static readonly DependencyProperty StateProperty =  
DependencyProperty.Register(  
    "State", typeof(Boolean), typeof(MyStateControl), new  
PropertyMetadata(false));  
}
```

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

- [依赖项属性概述](#)
- [操作指南主题](#)

# 如何：为依赖项属性添加所有者类型

项目 • 2023/02/06

此示例演示如何将类添加为注册为不同类型的依赖属性的所有者。通过执行此操作，WPF XAML 读取器和属性系统都能够将类识别为属性的附加所有者。选择性地添加为所有者让添加类可以提供特定于类型的元数据。

在以下示例中，`StateProperty` 是 `MyStateControl` 类注册的属性。

`UnrelatedStateControl` 类使用 `AddOwner` 方法将自身添加为 `StateProperty` 的所有者，特别是使用支持在添加类型上存在依赖属性的新元数据的签名。请注意，应该为属性提供公共语言运行时 (CLR) 访问器，类似于[实现依赖属性](#)示例中所示的示例，并在作为所有者添加的类上重新公开依赖属性标识符。

从使用 `GetValue` 或 `SetValue` 的编程访问的角度来看，在没有包装器的情况下，依赖属性仍然可以正常工作。但通常需要将此属性系统行为与 CLR 属性包装器并行执行。包装器可以更轻松地以编程方式设置依赖属性，并可以将属性设置为 XAML 特性。

若要了解如何替代默认元数据，请参阅[替代依赖属性的元数据](#)。

## 示例

C#

```
public class MyStateControl : ButtonBase
{
    public MyStateControl() : base() { }

    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }

    public static readonly DependencyProperty StateProperty =
        DependencyProperty.Register(
            "State", typeof(Boolean), typeof(MyStateControl), new
        PropertyMetadata(false));
}
```

C#

```
public class UnrelatedStateControl : Control
{
    public UnrelatedStateControl() { }

    public static readonly DependencyProperty StateProperty =
        MyStateControl.StateProperty.AddOwner(typeof(UnrelatedStateControl), new
        PropertyMetadata(true));
```

```
public Boolean State
{
    get { return (Boolean)this.GetValue(StateProperty); }
    set { this.SetValue(StateProperty, value); }
}
```

## 另请参阅

- [自定义依赖属性](#)
- [依赖项属性概述](#)

# 如何：注册附加属性

项目 • 2023/02/06

此示例演示如何注册附加属性和提供公共访问器，以便可以在两种 WPF 类型（也实现为依赖属性）中使用该属性。可以在任何 [DependencyObject](#) 类型上使用依赖属性。

## 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。在其他类上使用属性时，提供程序类可以选择为适用的属性提供默认元数据，除非该类会重写元数据。在此示例中，`IsBubbleSource` 属性的默认值设置为 `false`。

附加属性（即使未注册为依赖属性）的提供程序类必须提供遵循 `Set [附加属性名称]` 和 `Get [附加属性名称]` 命名约定的静态 get 和 set 访问器。需要这些访问器目的是生效的 XAML 读取器可以在 XAML 中将属性识别为特性，并解析相应的类型。

C#

```
public static readonly DependencyProperty IsBubbleSourceProperty =
DependencyProperty.RegisterAttached(
    "IsBubbleSource",
    typeof(Boolean),
    typeof(AquariumObject),
    new FrameworkPropertyMetadata(false,
FrameworkPropertyMetadataOptions.AffectsRender)
);
public static void SetIsBubbleSource(UIElement element, Boolean value)
{
    element.SetValue(IsBubbleSourceProperty, value);
}
public static Boolean GetIsBubbleSource(UIElement element)
{
    return (Boolean)element.GetValue(IsBubbleSourceProperty);
}
```

## 另请参阅

- [DependencyProperty](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [操作指南主题](#)

# 如何：重写依赖属性的元数据

项目 • 2023/02/06

此示例演示如何重写来自继承类的默认依赖属性元数据，其方法是调用 [OverrideMetadata](#) 方法并提供特定于类型的元数据。

## 示例

通过定义其 [PropertyMetadata](#)，类可以定义依赖属性的行为，例如其默认值和属性系统回叫。很多依赖属性类都已经将默认元数据创建为其注册过程的一部分。这包含作为 WPF API 一部分的依赖属性。通过其类继承继承依赖属性的类可重写原始的元数据，以便可通过元数据更改的属性的特征与任何特定于子类的要求相匹配。

在依赖属性上重写元数据的操作必须在属性系统使用该属性之前进行，也就是说，在对注册属性的对象的特定实例进行实例化之前进行。必须在将其自身作为 [OverrideMetadata](#) 的 `forType` 参数的类型的静态构造函数内执行对 [OverrideMetadata](#) 的调用。所有者类型的实例存在之后尝试更改元数据不会引发异常，但会在属性系统中导致不一致的行为。此外，每种类型只可以重写一次元数据。以后在同一类型上重写元数据的尝试会引发异常。

在下面示例中，自定义类 `MyAdvancedStateControl` 重写了由带有新属性元数据的 `MyAdvancedStateControl` 为 `StateProperty` 提供的元数据。例如，在新构造的 `MyAdvancedStateControl` 实例上查询 `StateProperty` 时，其默认值现在是 `true`。

C#

```
public class MyStateControl : ButtonBase
{
    public MyStateControl() : base() { }

    public Boolean State
    {
        get { return (Boolean)this.GetValue(StateProperty); }
        set { this.SetValue(StateProperty, value); }
    }

    public static readonly DependencyProperty StateProperty =
        DependencyProperty.Register(
            "State", typeof(Boolean), typeof(MyStateControl), new
        PropertyMetadata(false));
}
```

C#

```
public class MyAdvancedStateControl : MyStateControl
{
```

```
public MyAdvancedStateControl() : base() { }

static MyAdvancedStateControl()
{
    MyStateControl.StateProperty.OverrideMetadata(typeof(MyAdvancedStateControl)
        , new PropertyMetadata(true));
}
```

## 另请参阅

- [DependencyProperty](#)
- [依赖项属性概述](#)
- [自定义依赖属性](#)
- [操作指南主题](#)

# 事件 (WPF)

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 引入了路由事件，这些事件可在应用程序的元素树中调用存在于各个侦听器上的处理程序。

## 本节内容

- [路由事件概述](#)
- [附加事件概述](#)
- [对象生存期事件](#)
- [将路由事件标记为“已处理”和“类处理”](#)
- [预览事件](#)
- [属性更改事件](#)
- [Visual Basic 和 WPF 事件处理](#)
- [弱事件模式](#)
- [操作指南主题](#)

## 参考

[RoutedEventArgs](#)

[EventManager](#)

[RoutingStrategy](#)

## 相关章节

- [WPF 体系结构](#)
- [WPF 中的 XAML](#)
- [基元素](#)
- [元素树和序列化](#)
- [属性](#)
- [输入](#)
- [资源](#)
- [样式设置和模板化](#)
- [WPF 内容模型](#)
- [线程模型](#)

# 路由事件概述

项目 • 2022/09/27

本主题介绍 Windows Presentation Foundation (WPF) 中路由事件的概念。本主题定义路由事件术语、描述路由事件如何通过元素树来路由、概述如何处理路由事件，并介绍如何创建你自己的自定义路由事件。

## 先决条件

本主题假定你对如下内容有基本了解：公共语言运行时 (CLR)、面向对象的编程以及如何将 WPF 元素之间的关系概念化为树。若要理解本主题中的示例，还应当了解 WPF 应用程序或页面。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序和 WPF 中的 XAML](#)。

## 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。此处对这两种定义均进行了说明，因为有的用户认为前者更有用，有的用户认为后者更有用。

**功能定义：**路由事件是一种可以针对元素树中的多个侦听器（而不是仅针对引发该事件的对象）调用处理程序的事件。

**实现定义：**路由事件是一个 CLR 事件，由 [RoutedEventArgs](#) 类的实例提供支持并由 Windows Presentation Foundation (WPF) 事件系统处理。

典型的 WPF 应用程序包含许多元素。无论这些元素是在代码中创建还是在 XAML 中声明，它们存在于彼此关联的元素树关系中。根据事件的定义，事件路由可以按两种方向之一传播，但是通常会在元素树中从源元素向上“浮升”，直到它到达元素树的根（通常是页面或窗口）。如果你以前用过 DHTML 对象模型，则可能会熟悉这个浮升概念。

请思考下面的简单元素树：

XAML

```
<Border Height="50" Width="300" BorderBrush="Gray" BorderThickness="1">
  <StackPanel Background="LightGray" Orientation="Horizontal"
    Button.Click="CommonClickHandler">
    <Button Name="YesButton" Width="Auto" >Yes</Button>
    <Button Name="NoButton" Width="Auto" >No</Button>
    <Button Name="CancelButton" Width="Auto" >Cancel</Button>
  </StackPanel>
</Border>
```

此元素树生成类似如下的内容：



在这个简化的元素树中，[Click](#) 事件的源是某个 [Button](#) 元素，而所单击的 [Button](#) 是有机会处理该事件的第一个元素。但是，如果附加到 [Button](#) 的所有处理程序均未处理该事件，则该事件向上浮升到元素树中的 [Button](#) 父级（即 [StackPanel](#)）。该事件可能会浮升到 [Border](#)，然后会到达元素树的页面根（未显示）。

换言之，此 [Click](#) 事件的事件路由为：

Button-->StackPanel-->Border-->...

## 路由事件的顶级方案

下面简要概述了需运用路由事件概念的方案，以及为什么典型的 CLR 事件不适合这些方案：

**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，所添加的图像不得中断命中测试行为（该行为会使按钮响应其内容的 [Click](#)，即使用户所单击的像素在技术上属于该图像也是如此）。

**单一处理程序附加点：**在 Windows 窗体中，必须多次附加同一个处理程序，才能处理从多个元素引发的事件。借助路由事件，可以只附加该处理程序一次（如上例中所示），并在必要时使用处理程序逻辑来确定该事件的源位置。例如，这可以是前面显示的 XAML 的处理程序：

C#

```
private void CommonClickHandler(object sender, RoutedEventArgs e)
{
    FrameworkElement feSource = e.Source as FrameworkElement;
    switch (feSource.Name)
    {
        case "YesButton":
            // do something here ...
            break;
        case "NoButton":
            // do something ...
            break;
        case "CancelButton":
            // do something ...
            break;
    }
}
```

```
    e.Handled=true;
}
```

**类处理**：路由事件允许使用由类定义的静态处理程序。此类处理程序能够抢在任何附加的实例处理程序之前处理事件。

**引用事件，而不反射**：某些代码和标记技术需要能标识特定事件的方法。路由事件创建 [RoutedEvent](#) 字段作为标识符，从而提供不需要静态反射或运行时反射的可靠的事件标识技术。

## 路由事件的实现方式

路由事件是一个 CLR 事件，它由 [RoutedEvent](#) 类的实例提供支持并向 WPF 事件系统注册。从注册中获取的 [RoutedEvent](#) 实例通常保留为特定类的 `public static readonly` 字段成员，该类注册路由事件并因此“拥有”路由事件。与同名 CLR 事件（有时称为“包装器”事件）的连接是通过替代 CLR 事件的 `add` 和 `remove` 实现来完成的。通常，`add` 和 `remove` 保留为隐式默认值，该默认值使用特定于语言的相应事件语法来添加和删除该事件的处理程序。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。

以下示例演示自定义 `Tap` 路由事件的声明，其中包括注册和公开 [RoutedEvent](#) 标识符字段以及对 `Tap` CLR 事件进行 `add` 和 `remove` 实现。

C#

```
public static readonly RoutedEvent TapEvent =
EventManager.RegisterRoutedEvent(
    "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
    typeof(MyButtonSimple));

// Provide CLR accessors for the event
public event RoutedEventHandler Tap
{
    add { AddHandler(TapEvent, value); }
    remove { RemoveHandler(TapEvent, value); }
}
```

## 路由事件处理程序和 XAML

若要使用 XAML 为事件添加处理程序，可将该事件的名称声明为用作事件侦听器的元素的属性。该属性的值是所实现的处理程序方法的名称，该方法必须存在于代码隐藏文件的分部类中。

XAML

```
<Button Click="b1SetColor">button</Button>
```

用来添加标准 CLR 事件处理程序的 XAML 语法与用来添加路由事件处理程序的语法相同，因为实际上是向底层具有路由事件实现的 CLR 事件包装器添加处理程序。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](#)。

## 路由策略

路由事件使用以下三种路由策略之一：

- **浮升**：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，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不同控件或其他 UI 元素的输入或状态变化。
- **直接**：只有源元素本身才有机会调用处理程序以进行响应。 这与 Windows 窗体用于事件的“路由”相似。 但是，与标准 CLR 事件不同的是，直接路由事件支持类处理（将在下一节中介绍类处理），并且可供 [EventSetter](#) 和 [EventTrigger](#) 使用。
- **隧道**：最初将调用元素树的根处的事件处理程序。 随后，路由事件将朝着路由事件的源节点元素（即引发路由事件的元素）方向，沿路由线路传播到后续的子元素。 合成控件的过程中通常会使用或处理隧道路由事件，通过这种方式，可以有意地禁止复合部件中的事件，或者将其替换为特定于整个控件的事件。 在 WPF 中提供的输入事件通常是以隧道/浮升对实现的。 隧道事件有时又称作预览事件，这是由该对所使用的命名约定决定的。

## 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。

如果使用以下任一建议方案，路由事件的功能将得到充分发挥：在公用根处定义公用处理程序、合成自己的控件或者定义自己的自定义控件类。

路由事件侦听器和路由事件源不必在其层次结构中共享公用事件。 任何 [UIElement](#) 或 [ContentElement](#) 可以是任一路由事件的事件侦听器。 因此，可以将有效的 API 集中可用的全套路由事件用作概念“接口”，应用程序中的不同元素通过这个接口来交换事件信息。 路由事件的这个“接口”概念特别适用于输入事件。

路由事件还可以用于通过元素树进行通信，因为事件的事件数据会保留到路由中的每个元素中。 一个元素可以更改事件数据中的某些内容，该更改将用于路由中的下一个元素。

之所以将任何给定的 WPF 事件作为路由事件实现（而不是作为标准 CLR 事件实现），除了路由方面的原因，还有两个其他原因。如果要实现自己的事件，则可能也需要考虑这些原则：

- 某些 WPF 样式和模板功能（如 [EventSetter](#) 和 [EventTrigger](#)）要求被引用的事件是路由事件。前面提到的事件标识符方案就是这样的。
- 路由事件支持类处理机制，类可以通过该机制来指定静态方法，这些静态方法能够在任何已注册的实例处理程序访问路由事件之前，处理这些路由事件。这在控件设计中非常有用，因为类可以强制执行事件驱动的类行为，以防它们在处理实例上的事件时被意外禁止。

本主题将用单独的章节来讨论上述每个因素。

## 为路由事件添加和实现事件处理程序

若要在 XAML 中添加事件处理程序，只需将事件名称作为属性添加到元素中，并将属性值设置为用来实现相应委托的事件处理程序的名称，如以下示例所示。

XAML

```
<Button Click="b1SetColor">button</Button>
```

`b1SetColor` 是实现的处理程序的名称，该处理程序包含用来处理 `Click` 事件的代码。

`b1SetColor` 必须具有与 [RoutedEventHandler](#) 委托相同的签名，该委托是 `Click` 事件的事件处理程序委托。所有路由事件处理程序委托的第一个参数都指定要向其中添加事件处理程序的元素，第二个参数指定事件的数据。

C#

```
void b1SetColor(object sender, RoutedEventArgs args)
{
    //logic to handle the Click event
}
```

[RoutedEventHandler](#) 是基本的路由事件处理程序委托。对于针对某些控件或方案的专用路由事件，要用于路由事件处理程序的委托还可能会变得更加专用化，以便可以传输专用的事件数据。例如，在常见的输入方案中，可能需要处理 [DragEnter](#) 路由事件。处理程序应实现 [DragEventHandler](#) 委托。通过使用更具针对性的委托，可以处理处理程序中的 [DragEventArgs](#)，并读取 [Data](#) 属性，该属性包含拖动操作的剪贴板有效负载。

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[处理路由事件](#)。

在用代码创建的应用程序中为路由事件添加处理程序非常简单。 路由事件处理程序始终可以通过 helper 方法 `AddHandler` 来添加（对 `add` 的现有支持进行调用也是使用此方法）。但是，现有的 WPF 路由事件通常借助于支持机制来实现 `add` 和 `remove` 逻辑，这些实现允许使用特定于语言的事件语法来添加路由事件的处理程序，特定于语言的事件语法比 helper 方法更直观。 下面是 Helper 方法的示例用法：

```
C#  
  
void MakeButton()  
{  
    Button b2 = new Button();  
    b2.AddHandler(Button.ClickEvent, new RoutedEventHandler(Onb2Click));  
}  
void Onb2Click(object sender, RoutedEventArgs e)  
{  
    //logic to handle the Click event  
}
```

下一个示例演示 C# 运算符语法（Visual Basic 的运算符语法稍有不同，因为它以不同的方法来处理取消引用）：

```
C#  
  
void MakeButton2()  
{  
    Button b2 = new Button();  
    b2.Click += new RoutedEventHandler(Onb2Click2);  
}  
void Onb2Click2(object sender, RoutedEventArgs e)  
{  
    //logic to handle the Click event  
}
```

有关如何在代码中添加事件处理程序的示例，请参阅[使用代码添加事件处理程序](#)。

如果使用的是 Visual Basic，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

所有路由事件共享一个通用事件数据基类 `RoutedEventArgs`。`RoutedEventArgs` 定义了采用布尔值的 `Handled` 属性。`Handled` 属性的目的在于，允许路由中的任何事件处理程序通过将 `Handled` 的值设置为 `true` 来将路由事件标记为“已处理”。处理程序在路由上的某个元素处对共享事件数据进行处理之后，这些数据将再次报告给路由上的每个侦听器。

`Handled` 的值会影响路由事件在沿路由向远处传播过程中的报告或处理方式。在路由事件的事件数据中，如果 `Handled` 为 `true`，则通常不再为该特定事件实例调用用于在其他元素上侦听该路由事件的处理程序。这条规则对以下两类处理程序均适用：在 XAML 中附加的处理程序；由特定于语言的事件处理程序附加语法（如 `+ =` 或 `Handles`）添加的处理程序。对于最常见的处理程序方案，如果通过将 `Handled` 设置为 `true` 以便将事件标记为“已处理”，则将“停止”隧道路由或浮升路由，此外还会“停止”类处理程序在某个路由点处理的所有事件的路由。

但是，侦听器仍可以通过“`handledEventsToo`”机制来运行处理程序，以便在事件数据中的 `Handled` 为 `true` 时响应路由事件。换言之，将事件数据标记为“已处理”并不会真的停止事件路由。只能在代码或 `EventSetter` 中使用 `handledEventsToo` 机制：

- 在代码中，不要使用适用于一般 CLR 事件的特定于语言的事件语法，而是通过调用 WPF 方法 `AddHandler(RoutedEventArgs, Delegate, Boolean)` 来添加处理程序。请将 `handledEventsToo` 的值指定为 `true`。
- 在 `EventSetter` 中，将 `HandledEventsToo` 属性设置为 `true`。

除了 `Handled` 状态在路由事件中生成的行为以外，`Handled` 概念还会影响设计自己的应用程序和编写事件处理程序代码的方式。可以将 `Handled` 概念化为由路由事件公开的简单协议。此协议的具体使用方法由你决定，但是需要按照如下方式对 `Handled` 值的预期使用方式进行概念设计：

- 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”，则说明该路由中前面的其他侦听器已选择了不注册处理程序，或者已注册的处理程序选择不操作事件数据并将 `Handled` 设置为 `true`。（或者，当前侦听器很可能是路由中的第一个点。）当前侦听器上的处理程序现在有三个可能的操作过程：
  - 不执行任何操作；该事件保持未处理状态，该事件将路由到下一个侦听器。
  - 执行代码以响应该事件，但是所执行的操作被视为不足以保证将事件标记为“已处理”。该事件将路由到下一个侦听器。
  - 执行代码以响应该事件。在传递到处理程序的事件数据中将该事件标记为“已处理”，因为所执行的操作被视为不足以保证将该事件标记为“已处理”。该事件仍将路由到下一个侦听器，但是，由于在其事件数据中，`Handled = true`，因此只有 `handledEventsToo` 侦听器才有机会进一步调用处理程序。

这个概念设计通过前面所述的路由行为得到增强：为调用的路由事件附加处理程序变得更难（虽然在代码或样式中依然可执行此操作），即使路由中前面的处理程序已经将 `Handled` 设置为 `true` 也是如此。

有关 [Handled](#)、路由事件的类处理的详细信息，以及针对何时适合将路由事件标记为 [Handled](#) 的建议，请参阅[将路由事件标记为“已处理”和“类处理”](#)。

在应用程序中，相当常见的做法是只针对引发浮升路由事件的对象来处理该事件，而根本不考虑事件的路由特征。但是，在事件数据中将路由事件标记为“已处理”仍是一个不错的选择，因为这样可以防止元素树中位置更高的元素也对同一个路由事件附加了处理程序而出现意外的副作用。

## 类处理程序

如果你定义的类是以某种方式从 [DependencyObject](#) 派生的，那么对于作为类的已声明或已继承事件成员的路由事件，还可以定义和附加一个类处理程序。每当路由事件到达其路由中的元素实例时，都会先调用类处理程序，然后再调用附加到该类某个实例的任何实例侦听器处理程序。

有些 WPF 控件对某些路由事件具有固有的类处理。路由事件可能看起来从未引发过，但实际上正对其进行类处理，如果使用某些技术，路由事件还是可以由实例处理程序进行处理。此外，许多基类和控件会公开可用来替代类处理行为的虚拟方法。若要深入了解如何解决不需要的类处理以及如何在自定义类中定义自己的类处理，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。使用附加事件，可以将特定事件的处理程序添加到任意元素中。处理事件的元素不必定义或继承附加事件，可能引发事件的对象和用来处理实例的目标也都不必将该事件定义为类成员，或将其作为类成员来“拥有”。

WPF 输入系统广泛使用附加事件。但是，几乎所有的附加事件都是通过基本元素转发的。输入事件随后会显示为作为基本元素类成员的等效非附加路由事件。例如，通过针对该 [UIElement](#) 使用 [MouseDown](#)（而不是在 XAML 或代码中处理附加事件语法），可以针对任何给定的 [UIElement](#)，更方便地处理基础附加事件 [Mouse.MouseDown](#)。

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

为子元素所引发的路由事件附加处理程序是另一个语法用法，它与 `typename.eventname` 附加事件语法相似，但它并非严格意义上的附加事件用法。可以向公用父级附加处理程序以利用事件路由，即使公用父级可能没有作为成员的相关路由事件，也是如此。请再次思考下面的示例：

## XAML

```
<Border Height="50" Width="300" BorderBrush="Gray" BorderThickness="1">
  <StackPanel Background="LightGray" Orientation="Horizontal"
    Button.Click="CommonClickHandler">
    <Button Name="YesButton" Width="Auto" >Yes</Button>
    <Button Name="NoButton" Width="Auto" >No</Button>
    <Button Name="CancelButton" Width="Auto" >Cancel</Button>
  </StackPanel>
</Border>
```

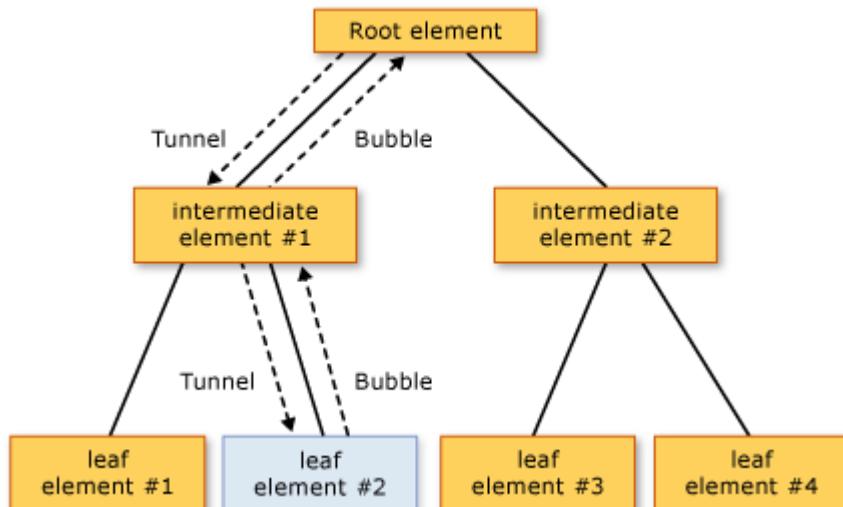
在这里，向其中添加处理程序的父元素侦听器是 `StackPanel`。但是，它正在为已经声明而且将由 `Button` 类（实际上是 `ButtonBase`，但是可以通过继承向 `Button` 提供）引发的路由事件添加处理程序。`Button`“拥有”该事件，但是路由事件系统允许将任何路由事件的处理程序附加到任何 `UIElement` 或 `ContentElement` 实例侦听器，这可能会以其他方式为公共语言运行时 (CLR) 事件附加侦听器。对于这些限定的事件属性名称来说，默认的 `xmlns` 命名空间通常是默认的 WPF `xmlns` 命名空间，但是还可以为自定义路由事件指定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。在 WPF 中，按照约定，隧道路由事件的名称以单词“Preview”为前缀。输入事件通常成对出现，一个是浮升事件，另一个是隧道事件。例如，`KeyDown` 事件和 `PreviewKeyDown` 事件具有相同的签名，前者是浮升输入事件，后者是隧道输入事件。偶尔，输入事件只有浮升版本，或者有可能只有直接路由版本。在文档中，路由事件主题交叉引用具有备用路由策略的类似路由事件（如果存在这类路由事件），托管的引用页面中的相关部分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，使来自输入的单个用户操作（如按鼠标按钮）按顺序引发该对中的两个路由事件。首先引发隧道事件并沿路由传播。然后引发浮升事件并沿路由传播。这两个事件共享同一个事件数据实例，因为用来引发浮升事件的实现类中的 `RaiseEvent` 方法调用会侦听隧道事件中的事件数据，并在新引发的事件中重用它。具有隧道事件处理程序的侦听器首先获得将路由事件标记为“已处理”的机会（首先是类处理程序，然后是实例处理程序）。如果隧道路由中的某个元素将路由事件标记为“已处理”，则会针对浮升事件发送已处理的事件数据，而且将不调用等效的浮升输入事件的附加典型处理程序。已处理的浮升事件看起来好像尚未引发。此处理行为对于控件合成非常有用，因为在此情况下你可能希望所有基于命中测试的输入事件或者所有基于焦点的输入事件都由最终的控件（而不是它的复合部件）报告。作为可支持控件类的代码的一部分，最后一个控件元素靠近合成中的根，因此将有机会首先对隧道事件进行类处理，或者有机会将该路由事件“替换”为更针对控件的事件。

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` 是先后发生的 `PreviewMouseDown` 事件和 `MouseDown` 事件的源：



事件的处理顺序如下所述：

1. 针对根元素处理 `PreviewMouseDown` ( 隧道 )。
2. 针对中间元素 #1 处理 `PreviewMouseDown` ( 隧道 )。
3. 针对源元素 #2 处理 `PreviewMouseDown` ( 隧道 )。
4. 针对源元素 #2 处理 `MouseDown` ( 浮升 )。
5. 针对中间元素 #1 处理 `MouseDown` ( 浮升 )。
6. 针对根元素处理 `MouseDown` ( 浮升 )。

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。在其中调用处理程序的对象是由 `sender` 参数报告的对象。首先在其中引发事件的对象由事件数据中的 `Source` 属性报告。路由事件仍可以由同一个对象引发和处理，在这种情况下，`sender` 和 `Source` 是相同的（事件处理示例列表中的步骤 3 和 4 属于这样的情况）。

由于存在隧道和浮升，因此父元素接收输入事件，其中 `Source` 是其子元素之一。当有必要知道源元素是哪个元素时，可以通过访问 `Source` 属性来标识源元素。

通常，一旦将输入事件标记为 `Handled`，就不会再进一步调用处理程序。通常，一旦调用了用来对输入事件的含义进行特定于应用程序的逻辑处理的处理程序，就应当将输入事件标记为“已处理”。

有关 `Handled` 状态的此通用声明有一个例外，即注册为有意忽略事件数据 `Handled` 状态的输入事件处理程序仍将在其路由中被调用。有关详细信息，请参阅[预览事件](#)或[将路由事件标记为“已处理”和类处理](#)。

通常，隧道事件和浮升事件之间的共享事件数据模型以及先引发隧道事件后引发浮升事件的顺序引发并非适用于所有的路由事件的概念。该行为的实现取决于 WPF 输入设备选择引发和连接输入事件对的具体方式。实现你自己的输入事件是一个高级方案，但是你也可以选择针对自己的输入事件遵循该模型。

一些类选择对某些输入事件进行类处理，其目的通常是重新定义用户驱动的特定输入事件在该控件中的含义并引发新事件。有关详细信息，请参阅[将路由事件标记为“已处理”和类处理](#)。

有关输入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

在样式中，可以通过使用 [EventSetter](#) 在标记中添加一些预先声明的 XAML 事件处理语法。在应用样式时，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](#)。下面是一个示例。请注意，此处引用的 `b1SetColor` 方法位于代码隐藏文件中。

XAML

```
<StackPanel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.EventOvw2"
    Name="dpanel2"
    Initialized="PrimeHandledToo"
>
    <StackPanel.Resources>
        <Style TargetType="{x:Type Button}">
            <EventSetter Event="Click" Handler="b1SetColor"/>
        </Style>
    </StackPanel.Resources>
    <Button>Click me</Button>
    <Button Name="ThisButton" Click="HandleThis">
        Raise event, handle it, use handled=true handler to get it anyway.
    </Button>
</StackPanel>
```

这样做好处在于，样式有可能包含大量可应用于应用程序中任何按钮的其他信息，将 [EventSetter](#) 添加到样式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](#) 还进一步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，只有路由事件可以用于 [EventTrigger](#)。通常将 [EventTrigger](#) 声明为样式的一部分，但是还可以在页面级元素上将 [EventTrigger](#) 声明为 [Triggers](#) 集合的一部分。

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

本主题主要从以下角度讨论路由事件：描述基本概念；就如何以及何时响应各种基元素和控件中已经存在的路由事件提供指南。但是，你可以在自定义类上创建自己的路由事件以及所有必要的支持（如专用的事件数据类和委托）。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](#)

# 附加事件概述

项目 · 2023/02/06

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。通过附加事件的概念，你能够向任意元素（而不是实际定义或继承事件的元素）添加特定事件的处理程序。在这种情况下，对象既不会引发事件，目标处理实例也不会定义或“拥有”事件。

## 先决条件

本主题假定你已阅读[路由事件概述](#)和[WPF 中的 XAML](#)。

## 附加事件语法

附加事件具有一种 XAML 语法和编码模式，后备代码必须使用该语法和编码模式才能支持使用附加事件。

在 XAML 语法中，不仅可以通过事件名称来指定附加事件，还可以通过事件所属类型和事件名称来指定，中间以句点 (.) 分隔。因为事件名称是使用具有其所属类型的名称限定的，所以附加事件语法允许将任何附加事件附加到可以实例化的任何元素。

例如，下面是 XAML 语法，用于附加自定义 `NeedsCleaning` 附加事件的处理程序：

XAML

```
<aqua: Aquarium Name="theAquarium" Height="600" Width="800"
aqua: AquariumFilter.NeedsCleaning="WashMe"/>
```

请注意 `aqua:` 前缀；该前缀在本例中是必需的，因为附加事件是来自自定义映射 `xmlns` 的自定义事件。

## WPF 如何实现附加事件

在 WPF 中，附加事件由 `RoutedEventArgs` 字段支持，并在引发后通过树进行路由。通常，附加事件的源（引发该事件的对象）是系统或服务源，所以运行引发该事件的代码的对象并不是元素树的直接组成部分。

## 附加事件的方案

在 WPF 中，附加事件存在于具有服务级抽象的某些功能区域，例如，对于通过静态 [Mouse](#) 类或 [Validation](#) 类实现的事件。与该服务交互或使用该服务的类可以在附加事件语法中使用该事件，也可以选择将附加事件作为路由事件来呈现，这是类如何集成服务功能的一部分。

尽管 WPF 定义了许多附加事件，但直接使用或处理附加事件的情形却很少。一般情况下，附加事件用于体系结构，但随后即被转发给非附加（使用 CLR 事件“包装器”提供支持）路由事件。

例如，通过针对该 [UIElement](#) 使用 [MouseDown](#)（而不是在 XAML 或代码中处理附加事件语法），可以针对任何给定的 [UIElement](#)，更方便地处理基础附加事件 [Mouse.MouseDown](#)。附加事件用于体系结构，因为它允许进一步扩展输入设备。假设的设备只需引发 [Mouse.MouseDown](#) 即可模拟鼠标输入，而不需要从 [Mouse](#) 派生。但是，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。

## 在 WPF 中处理附加事件

处理附加事件的过程以及将要编写的处理程序代码与路由事件的基本相同。

一般情况下，WPF 附加事件与 WPF 路由事件并没有太大的区别。不同之处在于如何确定事件的源，以及如何通过类将事件作为成员进行公开（这还将影响 XAML 处理程序语法）。

但是，正如前文所述，现有的 WPF 附加事件并不是专门用于在 WPF 中进行处理。更多情况下，该事件用于在复合时，使复合元素能够向父元素报告状态，在这种情况下，事件通常用代码引发，同时依赖于相关父类中的类处理。例如，[Selector](#) 中的项应引发附加的 [Selected](#) 事件，该事件随后由 [Selector](#) 类进行类处理，之后可能由 [Selector](#) 类转换为不同路由事件，[SelectionChanged](#)。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”](#)。

## 将自己的附加事件定义为路由事件

如果从通用 WPF 基类派生，则可以通过在类中包括某些模式方法并使用基类中已经存在的实用工具方法来实现自己的附加事件。

模式如下：

- 具有两个参数的方法 `AddEventNameHandler`。第一个参数是添加事件处理程序的实例。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。

- 具有两个参数的方法 RemoveEventNameHandler。第一个参数是从中删除事件处理程序的实例。第二个参数是要删除的事件处理程序。方法必须是 `public` 和 `static`，没有返回值。

当在元素上声明附加的事件处理程序属性时，`AddEventNameHandler` 访问器方法有助于 XAML 处理。`AddEventNameHandler` 和 `RemoveEventNameHandler` 方法还允许代码访问附加事件的事件处理程序存储。

此常规模式对于框架中的实际实现还不够精确，因为任何给定的 XAML 读取器实现都可能采用不同的方案在支持语言和体系结构中标识基础事件。这是 WPF 将附加事件实现为路由事件的原因之一；用于事件 ([RoutedEvent](#)) 的标识符已由 WPF 事件系统定义。另外，路由一个事件也是对附加事件的 XAML 语言级概念的自然实现扩展。

WPF 附加事件的 `AddEventNameHandler` 实现包括使用路由事件和处理程序作为参数调用 [AddHandler](#)。

此实现策略和路由事件系统一般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，以下代码通过将附加事件声明为路由事件的 WPF 附加事件策略，在所有者类 `Aquarium` 中定义 `NeedsCleaning` 附加事件。

C#

```
public static readonly RoutedEvent NeedsCleaningEvent =
EventManager.RegisterRoutedEvent("NeedsCleaning", RoutingStrategy.Bubble,
typeof(RoutedEventHandler), typeof(AquariumFilter));
public static void AddNeedsCleaningHandler(DependencyObject d,
RoutedEventHandler handler)
{
    UIElement uie = d as UIElement;
    if (uie != null)
    {
        uie.AddHandler(AquariumFilter.NeedsCleaningEvent, handler);
    }
}
public static void RemoveNeedsCleaningHandler(DependencyObject d,
RoutedEventHandler handler)
{
    UIElement uie = d as UIElement;
    if (uie != null)
    {
        uie.RemoveHandler(AquariumFilter.NeedsCleaningEvent, handler);
    }
}
```

请注意，用来确定附加事件标识符字段的方法，[RegisterRoutedEvent](#)，实际上与用来注册非附加路由事件的方法相同。附加事件和路由事件都已注册到集中式内部存储。此事

件存储实现能够考虑到[路由事件概述](#)中介绍的“事件即界面”概念。

## 引发 WPF 附加事件

通常不需要从代码中引发 WPF 定义的现有附加事件。这些事件采用常规“服务”概念模型，而 [InputManager](#) 等服务类则负责引发事件。

但是，如果要基于 WPF 模型（基于 [RoutedEvent](#) 附加事件）定义自定义附加事件，则可以使用 [RaiseEvent](#) 从任何 [UIElement](#) 或 [ContentElement](#) 引发附加事件。引发路由事件（不管是否附加）要求将元素树中的一个特定元素声明为事件源；该事件源被报告为 [RaiseEvent](#) 调用方。服务负责确定将哪个元素报告为元素树中的事件源

## 另请参阅

- [路由事件概述](#)
- [XAML 语法详述](#)
- [XAML 及 WPF 的自定义类](#)

# 对象生存期事件

项目 · 2023/02/06

本主题介绍表示对象生存期（创建、使用和销毁）中的阶段的特定 WPF 事件。

## 先决条件

本主题假定你从 WPF 应用程序的现有依赖属性的使用者角度了解依赖属性。

## 对象生存期事件

Microsoft .NET Framework 托管代码中的所有对象都要经历类似的一系列生命阶段，即创建、使用和销毁。许多对象还包括生命终止阶段，该阶段属于销毁阶段的一部分。WPF 对象（更明确的说，WPF 标识为元素的视觉对象）还具有对象生命的一系列常见阶段。WPF 编程和应用程序模型将这些阶段作为一系列事件公开。WPF 中有四种与生存期事件有关的主要类型对象 - 即常规元素、窗口元素、导航宿主和应用程序对象。窗口和导航宿主也都属于视觉对象（元素）这个较大的组。本主题首先介绍所有元素的通用生存期事件，然后介绍应用于应用程序定义、窗口或导航宿主的更具针对性的生存期事件。

## 元素通用生存期事件

任何 WPF 框架级元素（从 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生的那些对象）都具有三个常见的生命周期事件：[Initialized](#)、[Loaded](#) 和 [Unloaded](#)。

### Initialized

首先引发 [Initialized](#)，它大致对应于通过调用其构造函数进行的对象初始化。由于事件为响应初始化而发生，因此可以确保对象的所有属性均已设置。（异常是表达式用法，例如动态资源或绑定；这些表达式将是未计算的表达式。）由于要求设置所有属性，由标记中定义的嵌套元素引发的顺序 [Initialized](#) 似乎按元素树中最深层元素的顺序进行，然后是指向根的父元素。采用此顺序是鉴于父子关系和包容均为属性，因此在填充属性的子元素完全初始化前，父元素无法报告初始化。

编写处理程序以响应 [Initialized](#) 事件时，必须考虑到无法保证元素树（逻辑树或可视化树）中附加处理程序周围的所有其他元素（尤其是父元素）均已创建。成员变量可能为 null，或者基础绑定还未填充数据源（即使在表达式级别）。

### 已加载

接下来将引发 [Loaded](#)。在最终呈现之前且在布局系统已计算呈现所需的所有值后，将引发 [Loaded](#) 事件。[Loaded](#) 要求包含元素的逻辑树是完整的，并连接到提供 HWND 和呈现图面的演示源。标准数据绑定（绑定到本地源，例如其他属性或直接定义的数据源）将在 [Loaded](#) 之前发生。可能已发生异步数据绑定（外部或动态源），但是根据其异步特性的定义不能保证已发生异步数据绑定。

引发 [Loaded](#) 事件的机制不同于 [Initialized](#)。将以逐个元素的方式引发 [Initialized](#) 事件，无需通过整个元素树直接协调。相反，引发 [Loaded](#) 事件是通过整个元素树（具体而言是逻辑树）协调的结果。当树中所有元素都处于被视为已加载的状态时，将首先在根元素上引发 [Loaded](#) 事件。然后在每个子元素上连续引发 [Loaded](#) 事件。

### ① 备注

这种行为看起来可能类似于路由事件的隧道。但是，未将任何信息在事件之间传送。每个元素始终有机会处理其 [Loaded](#) 事件，并且将事件数据标记为已处理仅影响该元素。

## 已卸载

最后引发 [Unloaded](#)，并由演示源或要删除的可视化父级启动。当引发 [Unloaded](#) 并对其进行处理后，作为事件源父级（由 [Parent](#) 属性确定）的元素或逻辑树或可视树中上游的任何给定元素可能已移位，因此可能无法将数据绑定、资源引用和样式设置为其常规的或最后已知的运行时值。

## 生存期事件应用程序模型元素

基于元素的常见生存期事件构建了以下应用程序模型元素：[Application](#)、[Window](#)、[Page](#)、[NavigationWindow](#) 和 [Frame](#)。这些与特定用途相关的额外事件扩充了常见的生存期事件。将在以下位置详细介绍这些内容：

- [Application](#) 应用程序管理概述。
- [Window](#)：WPF 窗口概述。
- [Page](#)、[NavigationWindow](#) 和 [Frame](#)：导航概述。

## 另请参阅

- [依赖项属性值优先级](#)
- [路由事件概述](#)

# 将路由事件标记为“已处理”和“类处理”

项目 • 2023/02/06

路由事件的处理程序可以在事件数据内将事件标记为已处理。 处理事件将有效地缩短路由。 类处理是一个编程概念，受路由事件支持。 类处理程序有机会在类级别使用处理程序处理特定路由事件，在类的任何实例上存在任何实例处理程序之前调用该处理程序。

## 先决条件

本主题将详细介绍[路由事件概述](#)中引入的概念。

## 何时将事件标记为“已处理”

当在路由事件的事件数据中将 `Handled` 属性的值设置为 `true` 时，这就称为“将事件标记为已处理”。 对于应用程序创作者或者响应现有路由事件或实现新路由事件的控件创建者而言，何时应将路由事件标记为已处理，没有绝对规则。 大多数情形下，路由事件的事件数据中携带的“已处理”概念应当用作一种限定协议，用于自己的应用程序响应 WPF API 中公开的各种路由事件时以及用于任意自定义路由事件。 考虑“已处理”问题的另一种方式为：如果代码以重要且相对完整的方式响应路由事件，则通常应将路由事件标记为已处理。 通常，重要响应不应该超过一个。 所谓重要响应，是指对于单个任意路由事件，都需要单独的处理程序实现。 如果需要多个响应，则应通过在单个处理程序内链接的应用程序逻辑实现必要的代码，而不是使用路由事件系统进行转发。 是否“重要”这一概念比较主观，视应用程序或代码而定。 作为一般性指导原则，一些“重要响应”示例包括：设置焦点、修改公共状态、设置影响视觉表示形式的属性以及引发其他新事件。 非重要响应的示例包括：修改私有状态（无视觉影响，或编程表示形式）、记录事件或查看事件参数并选择不响应该事件。

路由事件系统行为在使用路由事件的已处理状态方面增强了此“重要响应”模型，因为不会调用 XAML 中添加的处理程序或 `AddHandler` 的公用签名来响应事件数据已标记为已处理的路由事件。 必须额外添加参数版本为 `handledEventsToo` 的处理程序 (`AddHandler(RoutedEventArgs, Delegate, Boolean)`) 才能处理由事件路由中的早期参与者标记为已处理的路由事件。

在某些情况下，控件本身会将某些路由事件标记为已处理。 已处理的路由事件表示 WPF 控件作者做出的以下决定：响应路由事件的控件操作是重要的，或者作为控件实现的一部分已完成，并且事件无需进一步处理。 通常，通过为事件添加一个类处理程序，或重写存在于基类上的其中一个类处理程序虚方法，可以完成此操作。 必要时仍然可以应对此事件处理；请参阅本主题后面部分的[通过控件解决事件禁止问题](#)。

# “预览”（隧道）事件与浮升事件和事件处理

预览路由事件是指遵循元素树的隧道路由的事件。命名约定中的“Preview”是指输入事件的一般原则，即预览（隧道）路由事件是在对等的冒泡路由事件之前引发的。另外，具有隧道和冒泡对的输入路由事件具有截然不同的处理逻辑。如果隧道/预览路由事件标记为已由事件侦听器处理，则冒泡路由事件将标记为已处理，然而此时冒泡路由事件的任何侦听器尚未接收到该事件。隧道路由事件和冒泡路由事件在技术层面上是单独的事件，但是它们有意共享相同的事件数据实例以实现此行为。

任意给定的 WPF 类引发其声明的路由事件的方式的内部实现可在隧道路由事件与浮升路由事件之间建立连接，对于成对的输入路由事件也是如此。但是除非该类级实现存在，否则共享命名方案的隧道路由事件与冒泡路由事件之间将没有连接：没有上述实现，它们将是两个完全独立的路由事件，不会顺次引发，也不会共享事件数据。

有关如何在自定义类中实现隧道/冒泡输入路由事件对的详细信息，请参阅[创建自定义路由事件](#)。

## 类处理程序和实例处理程序

路由事件会考虑事件的两种不同类型的侦听器：类侦听器和实例侦听器。类侦听器已存在，因为类型已在其静态构造函数中调用特定的 [EventManager](#)，即 [RegisterClassHandler](#)，或者已从元素基类重写类处理程序虚方法。实例侦听器是特定的类实例/元素，其中该路由事件已通过调用 [AddHandler](#) 附加了一个或多个处理程序。现有 WPF 路由事件调用 [AddHandler](#) 作为事件的公共语言运行时 (CLR) 事件包装器 `add{}` 和 `remove{}` 实现的一部分，这也是通过属性语法附加事件处理程序的简单 XAML 机制的启用方式。因此，即使是简单的 XAML 用法最终也等同于 [AddHandler](#) 调用。

将检查可视化树内的元素是否有注册的处理程序实现。可能会在整个路由中以该路由事件的路由策略类型所固有的顺序调用处理程序。例如，冒泡路由事件将首先调用那些附加到引发该路由事件的同一元素的处理程序。然后，路由事件“冒泡”到下一父元素，以此类推，直到到达应用程序根元素。

从冒泡路由中的根元素的角度看，如果类处理或者更靠近路由事件源的任意元素调用那些将事件参数标记为正在处理的处理程序，那么就不会调用根元素上的处理程序，这样在到达该根元素之前的事件路由会大大缩短。不过，路由并未完全停止，因为可以使用仍应调用处理程序的特殊条件来添加处理程序，即使类处理程序或实例处理程序已将路由事件标记为已处理也是如此。本主题后面部分的[即使在事件标记为已处理时也要添加引发的实例处理程序](#)对此进行了说明。

在比事件路由更深的级别上，还可能有多个类处理程序处理任意给定的类实例。这是因为，路由事件的类处理模型使得类层次结构中的所有可能的类都可以针对每个路由事件注册自己的类处理程序。每个类处理程序都会添加到一个内部存储，当构造应用程序的事

件路由时，所有类处理程序都会添加到该事件路由中。类处理程序按以下顺序添加到路由：最先调用派生程度最高的类处理程序，然后调用每个后续基类中的类处理程序。通常，不会注册类处理程序以便它们也响应标记为已处理的路由事件。因此，此类处理机制可提供以下两个选项之一：

- 派生类可以通过添加一个不将路由事件标记为已处理的处理程序，来补充从基类继承的类处理，因为有时会在派生类处理程序之后调用基类处理程序。
- 派生类可以通过添加一个将路由事件标记为已处理的类处理程序，来替换基类中的类处理。应慎用此方法，因为它可能会更改视觉外观、状态逻辑、输入处理和命令处理等方面原本的基控件设计。

## 按控件基类进行的路由事件类处理

在事件路由中的每个给定元素节点上，类侦听器都有机会在元素的任意实例侦听器之前响应路由事件。为此，类处理程序有时用于禁止某个特定控件类实现不希望继续传播的路由事件，或者用于提供类的一项功能，即对该路由事件进行特殊处理。例如，类可能引发特定于其自身的事件，其中包含有关某个用户输入条件在该特定类的上下文中所代表的意义的更多具体信息。然后，此类实现可能会将较为常规的路由事件标记为已处理。一般会添加类处理程序，这样在共享事件数据标记为已处理后，就不会对路由事件调用该处理程序，但是对于非典型情况，还存在一个 `RegisterClassHandler(Type, RoutedEvent, Delegate, Boolean)` 签名，即使在将路由事件标记为已处理的情况下，该签名也会注册类处理程序。

## 类处理程序虚方法

某些元素（特别是 `UIElement` 等基元素）会公开与其公共路由事件列表对应的空的“On\*Event”和“OnPreview\*Event”虚方法。可以重写这些虚方法以便为该路由事件实现类处理程序。如前所述，基元素类使用 `RegisterClassHandler(Type, RoutedEvent, Delegate, Boolean)` 将这些虚方法注册为每个此类路由事件的类处理程序。因为无需在静态构造函数中针对每个类型进行特殊初始化，所以 `On*Event` 虚方法使得为相关路由事件实现类处理变得简单得多。例如，可以通过替代 `OnDragEnter` 虚方法来在任何 `UIElement` 派生类中添加对 `DragEnter` 事件的类处理。在重写内，可以处理路由事件、引发其他事件、启动可能更改实例的元素属性的特定于类的逻辑或上述操作的任意组合。通常情况下，在此类重写中，即使将事件标记为已处理，也应调用基实现。强烈建议调用基实现，因为虚方法是在基类上。从每个虚方法调用基实现的标准受保护虚拟模式实质上会替换和比较路由事件类处理所固有的类似机制，因此可以在任意给定实例上调用类层次结构中所有类的类处理程序，并从派生程度最高的类的处理程序开始，一直继续到基类处理程序。如果类明确要求更改基类处理逻辑，则应仅忽略基实现调用。在重写代码之前还是之后调用基实现取决于实现的性质。

## 输入事件类处理

类处理程序虚方法均已注册，只有当存在任何共享事件数据尚未标记为已处理的情况时才调用这些方法。另外，仅仅对于输入事件而言，隧道和冒泡版本通常是顺次引发的，并且共享事件数据。对于一对给定的输入事件的类处理程序（其中一个是隧道版本，另一个是冒泡版本），当你可能不希望立即将事件标记为已处理时也需要此项。如果实现隧道类处理虚方法以便将事件标记为已处理，这将会阻止调用冒泡类处理程序（并阻止调用隧道或冒泡事件的任意正常注册的实例处理程序）。

节点上的类处理一经完成，就会考虑实例侦听器。

## 即使在事件标记为已处理时也要添加引发的实例处理程序

`AddHandler` 方法提供一个允许添加处理程序的特定重载，只要事件到达路由中的处理元素，事件系统就会调用该处理程序，即使其他处理程序通过调整事件数据将该事件标记为已处理也是如此。但一般不这样操作。通常情况下，不论事件是在元素树中的什么位置处理，即使需要多个最终结果，都可以通过编写处理程序来调整可能受事件影响的应用程序代码的各个方面。另外，通常情况下，需要响应该事件的实际只有一个元素，并且已发生了适当的应用程序逻辑。但是 `handledEventsToo` 重载适用于某些例外情况，例如元素数或控件复合中的某些其他元素已将事件标记为已处理，但是元素数中较高或较低的其他元素（视路由而定）仍希望调用自己的处理程序。

## 何时将已处理事件标记为未处理

通常，标记为已处理的路由事件不应再标记为未处理（将 `Handled` 重设回 `false`），即便处理 `handledEventsToo` 的处理程序也不应这样做。不过，某些输入事件具有高级别和低级别两种事件表示形式，当在树中的一个位置看到高级别事件，在另一个位置看到低级别事件时，这两种表示形式可以重叠。例如，假设这样一种情况：一个子元素侦听高级别键事件（如 `TextInput`），而父元素侦听低级别事件（如 `KeyDown`）。如果父元素处理低级别事件，则高级别事件甚至可能在子元素中被禁止，尽管直观看来子元素应该具有处理事件的先机。

在上述情形下，可能需要针对低级别事件向父元素和子元素中添加处理程序。子元素处理程序实现可以将低级别事件标记为已处理，但父元素处理程序实现会再次将其设置为未处理，这样树上方的更多元素（以及高级别事件）就可以有机会响应。这种情形应该非常少见。

## 有意对控件复合禁止输入事件

路由事件的类处理主要是用于输入事件和复合控件。按照定义，复合控件是由多个实际控件或控件基类组成的。控件作者通常希望合并每个子组件可能引发的所有可能的输入事件，以便将整个控件报告为单事件源。某些情况下，控件作者可能希望完全禁止来自组件的事件，或者替换上携带更多信息或者指示更具体行为的组件定义的事件。举个任何组件作者都可立即看到的标准示例：Windows Presentation Foundation (WPF) [Button](#) 处理任意鼠标事件的方式，该鼠标事件最终会解析为所有按钮都有的一个直观事件，即 [Click](#) 事件。

[Button](#) 基类 ([ButtonBase](#)) 派生自 [Control](#)，而后者又派生自 [FrameworkElement](#) 和 [UIElement](#)，并且控制输入处理所需的大多数事件基础结构在 [UIElement](#) 级别可用。特别是，[UIElement](#) 处理用于处理鼠标光标在其范围内的命中测试的一般 [Mouse](#) 事件，并针对最常见的按钮操作（例如 [MouseLeftButtonDown](#)）提供不同的事件。[UIElement](#) 还提供了一个空的虚拟 [OnMouseLeftButtonDown](#) 作为 [MouseLeftButtonDown](#) 的预注册类处理程序，并且 [ButtonBase](#) 将替代它。同样，[ButtonBase](#) 使用 [MouseLeftButtonUp](#) 的类处理程序。在传递事件数据的替代中，实现通过将 [Handled](#) 设置为 `true` 来将 [RoutedEventArgs](#) 实例标记为已处理，并且相同的事件数据会沿路由的其余部分传递到其他类处理程序，以及实例处理程序或事件资源库也是如此。此外，[OnMouseLeftButtonUp](#) 替代接下来将引发 [Click](#) 事件。大多数侦听器的最终结果将是 [MouseLeftButtonDown](#) 和 [MouseLeftButtonUp](#) 事件“消失”并由 [Click](#) 代替（这是一个更有意义的事件），因为已知此事件源自真正的按钮，而不是按钮的某个复合部分或其他元素的整体。

## 通过控件解决事件禁止问题

有时，各个控件内的此事件禁止行为可能会干扰应用程序的事件处理逻辑的某些较为常规的意图。例如，如果出于某种原因，应用程序在应用程序根元素中有一个 [MouseLeftButtonDown](#) 处理程序，你会发现任何鼠标单击按钮操作都不会调用根级别的 [MouseLeftButtonDown](#) 或 [MouseLeftButtonUp](#) 处理程序。事件本身实际上会向上冒泡（同样，事件路由未真正结束，但路由事件系统会在事件标记为已处理后更改其处理程序调用行为）。当路由事件到达按钮时，[ButtonBase](#) 类处理会将 [MouseLeftButtonDown](#) 标记为已处理，因为它希望将 [Click](#) 事件替换为更有意义的事件。因此，任何更深层路由上的标准 [MouseLeftButtonDown](#) 处理程序都不会被调用。可以使用两种方法来确保在此情形下会调用处理程序。

第一种技术是使用 [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 的 `handledEventsToo` 签名有意添加处理程序。这种方法的局限性在于用于附加事件处理程序的这种技术只可能来自代码，而不能来自标记。通过 Extensible Application Markup Language (XAML) 将事件处理程序名称指定为事件属性值的简单语法不会启用该行为。

第二种技术仅适用于输入事件，其中路由事件的隧道版本和冒泡版本是配对的。对于这些路由事件，可以改为将处理程序添加到预览/隧道对等路由事件。假如在应用程序的元

素树中的某个上级元素级别附加了 Preview 处理程序，该路由事件将从根开始在路由中传递，所以按钮类处理代码不会截获到该事件。如果使用此方法，则将任意 Preview 事件标记为已处理时一定要谨慎。对于在根元素处对 [PreviewMouseLeftButtonDown](#) 进行处理这一示例，如果在处理程序实现中将事件标记为 [Handled](#)，实际上会抑制 [Click](#) 事件。这通常不是希望的行为。

## 另请参阅

- [EventManager](#)
- [预览事件](#)
- [创建自定义路由事件](#)
- [路由事件概述](#)

# 预览事件

项目 · 2023/02/06

预览事件（也称为隧道事件）是路由事件，其中路由的方向是从应用程序根到引发事件的元素，并在事件数据中报告为源。并非所有事件方案都支持或需要预览事件；本主题介绍存在预览事件的情况，应用程序或组件应如何处理它们，以及可能适合在自定义组件或类中创建预览事件的情况。

## 预览事件和输入

通常处理预览事件时，请谨慎标记事件数据中处理的事件。在引发预览事件的元素（在事件数据中被报告为源的元素）以外的任何元素上处理预览事件会导致不向元素提供处理其发起的事件的机会。有时，这是期望的结果，尤其是当问题元素存在于控件组合内的关系中时。

具体而言，对于输入事件，预览事件还与等效的浮升事件共享事件数据实例。如果使用预览事件类处理程序将输入事件标记为已处理，则不会调用浮升输入事件类处理程序。或者，如果使用预览事件实例处理程序将事件标记为已处理，则通常不会调用浮升事件的处理程序。即使事件被标记为已处理，也可为类处理程序或实例处理程序注册或附加一个要调用的选项，但这种方法并不常用。

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](#)。

## 通过控件解决事件禁止问题

通常使用预览事件的一种情况是对输入事件进行复合控件处理。有时，控件的作者会禁止特定事件源自其控件，或许是为了替换包含更多信息或暗示更具体行为的组件定义事件。例如，Windows Presentation Foundation (WPF) [Button](#) 禁止 [Button](#) 或其复合元素引发 [MouseLeftButtonDown](#) 或 [MouseRightButtonDown](#) 浮升事件，支持捕获鼠标并引发始终由 [Button](#) 自身引发的 [Click](#) 事件。事件及其数据仍沿路由继续，但由于 [Button](#) 将事件数据标记为 [Handled](#)，因此只调用专门指示应在 [handledEventsToo](#) 情况下执行操作的事件的处理程序。如果应用程序根的其他元素仍然希望有机会处理受控件抑制的事件，一种替代方法是在代码中附加处理程序，并将 [handledEventsToo](#) 指定为 [true](#)。但通常更简单的方法是将你处理的路由方向更改为输入事件的预览等效项。例如，如果控件禁止 [MouseLeftButtonDown](#)，请尝试改为附加 [PreviewMouseLeftButtonDown](#) 的处理程序。此方法仅适用于基元素输入事件，例如 [MouseLeftButtonDown](#)。这些输入事件使用隧道/浮升对，引发这两个事件并共享事件数据。

上述每种方法都有副作用或限制。 处理预览事件的副作用是处理当时的事件可能会禁用预期处理浮升事件的处理程序，因此限制是，当事件仍在路由的预览部分时将其标记为已处理通常不是一个好主意。`handledEventsToo` 方法的限制是，不能在 XAML 中将 `handledEventsToo` 处理程序指定为属性，在获取要附加处理程序的元素的对象引用后，必须在代码中注册事件处理程序。

## 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](#)

# 属性更改事件

项目 · 2023/02/06

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。事件本身有时是路由事件，有时是标准公共语言运行时 (CLR) 事件。事件的定义因具体情况而异，因为有些属性更改更适于通过元素树路由，而其他属性更改则通常只与属性发生更改的对象有关。

## 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模式或命名模式显式地标识为属性更改事件。通常，SDK 文档中的事件描述会指出事件是否直接与属性值更改相关，以及是否提供属性与事件之间的交叉引用。

### RoutedPropertyChanged 事件

某些事件使用显式用于属性更改事件的事件数据类型和委托。事件数据类型为 `RoutedEventArgs<T>`，委托为 `RoutedPropertyChangedEventHandler<T>`。事件数据和委托都具有用于在定义处理程序时指定更改属性实际类型的泛型类型参数。事件数据包含两个属性 `OldValue` 和 `NewValue`，它们稍后都将作为事件数据中的类型参数传递。

名称中的“Routed”部分表示属性更改事件注册为路由事件。路由属性更改事件的好处是，如果子元素（控件的复合部分）上的属性更改了值，控件的顶层可以接收到属性更改事件。例如，可以创建一个包含 `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，则可能需要在父控件（而非该部分）上处理此更改。

因为有一个旧值和一个新值，所以很可能使用此事件处理程序作为属性值的验证程序。但是，这不是大多数属性更改事件的设计意图。通常，提供这些值是为了能够在代码的其他逻辑领域处理这些值，但实际上，在事件处理程序内更改值并不明智，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，则有一种更好的机制来跟踪属性更改，并且此机制是内置于 WPF 属性系统中的，即：属性系统回调 `CoerceValueCallback` 和 `PropertyChangedCallback`。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

属性更改事件场景中的另一对类型是 `DependencyPropertyChangedEventArgs` 和 `DependencyPropertyChangedEventHandler`。这些属性更改的事件不会路由；它们是标准 CLR 事件。`DependencyPropertyChangedEventArgs` 是一种罕见的事件数据报告类型，因为它不派生自 `EventArgs`；`DependencyPropertyChangedEventArgs` 是一个结构，而不是一个类。

使用 `DependencyPropertyChangedEventArgs` 和 `DependencyPropertyChangedEventHandler` 的事件比 `RoutedPropertyChanged` 事件稍微常见一些。使用这些类型的事件的一个示例是 `IsMouseCapturedChanged`。

与 `RoutedEventArgs<T>` 一样，`DependencyPropertyChangedEventArgs` 也会报告属性的新旧值。此外，关于如何处理这些值的注意事项也同样适用；通常不建议为响应事件而尝试在发送方再次更改值。

## 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，通过它可以创建基于分配了属性触发器的属性的值的条件行为。

属性触发器的属性必须是依赖项属性。它可以是（且通常是）只读依赖项属性。判断控件公开的依赖项属性是否至少部分设计为属性触发器的一个好方法是查看属性名称是否以“Is”开头。采用此命名规则的属性通常是只读的布尔依赖项属性，其属性的主要作用是报告可能与实时 UI 具有因果关系的控件状态，因此它是一个属性触发器候选。

其中一些属性还具有专用属性更改事件。例如，属性 `IsMouseCaptured` 具有属性更改事件 `IsMouseCapturedChanged`。该属性本身是只读的，其值由输入系统调整，并且输入系统对每次实时更改都引发 `IsMouseCapturedChanged`。

与真正的属性更改事件相比，使用属性触发器来处理属性更改具有一些限制。

属性触发器通过完全匹配逻辑来工作。指定一个属性和一个值，该值指示触发器将执行操作的特定值。例如：`<Setter Property="IsMouseCaptured" Value="true"> ... </Setter>`。由于此限制，大多数属性触发器用法都将针对布尔属性或者采用专用枚举值的属性，其可能值范围是可管理的，足以以为每种情况定义一个触发器。或者属性触发器可能仅仅用于特殊值，例如，当项计数达到零，却没有触发器来处理当属性值再次从零变为其他值（此处可能需要代码事件处理程序，或者当值不为零时再次从触发器状态切换回来的默认行为，而非针对所有情况的触发器）。

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“执行”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` 元素来设置正在应用样式或模板的对象的其他属性。

为了抵消具有各种可能值的属性触发器的“if”条件，通常建议使用 [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，[Trigger](#) 包含的 setter 将具有优先级，而只要触发器条件为 false，则不在 [Trigger](#) 内的 [Setter](#) 就具有优先级。

属性触发器通常适合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。

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](#)
- [依赖项属性概述](#)

# Visual Basic 和 WPF 事件处理

项目 • 2023/02/06

对于 Microsoft Visual Basic .NET 语言（仅），可以使用语言特定的 `Handles` 关键字将事件处理程序与实例关联，而不是通过特性或使用 `AddHandler` 方法附加事件处理程序。但是，用于将处理程序附加到实例的 `Handles` 技术存在一些限制，因为 `Handles` 语法不支持 WPF 事件系统的某些特定路由事件功能。

## 在 WPF 应用程序中使用“Handles”

通过 `Handles` 连接到实例和事件的事件处理程序必须全部在实例的分部类声明中定义，此要求也适用于通过元素上的特性值进行分配的事件处理程序。只能为页面上具有 `Name` 属性值（或声明了 `x:Name 指令`）的元素指定 `Handles`。这是因为 XAML 中的 `Name` 创建实例引用，该实例引用对于支持 `Handles` 语法所需的 `Instance.Event` 引用格式是非常必要的。可用于 `Handles` 且不具有 `Name` 引用的唯一元素是用于定义分部类的根元素实例。

可以通过使用逗号分隔 `Handles` 后面的 `Instance.Event` 引用，向多个元素分配相同的处理程序。

可以使用 `Handles` 将多个处理程序分配给同一 `Instance.Event` 引用。请勿对 `Handles` 引用中特定处理程序的提供顺序赋予任何重要性；应假定可按任何顺序调用处理相同事件的处理程序。

若要删除声明中通过 `Handles` 添加的处理程序，可调用 `RemoveHandler`。

如果要将处理程序附加到实例成员表中用于定义要处理的事件的实例，可使用 `Handles` 附加路由事件的处理程序。对于路由事件，通过 `Handles` 附加的处理程序遵循的路由规则与附加为 XAML 特性或使用 `AddHandler` 公共签名附加的处理程序所遵循的路由规则相同。这意味着如果事件已标记为已处理（事件数据中的 `Handled` 属性为 `True`），则不会调用通过 `Handles` 附加的处理程序来响应该事件实例。可通过路由中另一个元素上的实例处理程序，或通过在路由中当前元素或更早元素上进行类处理，将事件标记为已处理。对于可支持成对的隧道事件/冒泡事件的输入事件，隧道路由可能已将事件对标记为已处理。有关路由事件的详细信息，请参阅[路由事件概述](#)。

## 添加处理程序时的“Handles”限制

`Handles` 无法引用附加事件的处理程序。必须对该附加事件使用 `add` 访问方法，或使用 XAML 中的 `typename.eventname` 事件特性。有关详细信息，请参阅[路由事件概述](#)。

对于路由事件，只能使用 `Handles` 为实例成员表中存在该事件的实例分配处理程序。但是，对于一般的路由事件，父元素可以是来自子元素的事件的侦听器，即使该父元素的成员表中没有此事件也是如此。在特性语法中，这可以通过 `typename.membername` 特性形式来指定，此形式限定哪种类型实际定义要处理的事件。例如，某个父 `Page`（未定义 `Click` 事件）可以通过分配 `Button.Click` 形式的特性处理程序来侦听按钮单击事件。但 `Handles` 不支持 `typename.membername` 形式，因为它必须支持与该形式冲突的 `Instance.Event` 形式。有关详细信息，请参阅[路由事件概述](#)。

`Handles` 无法附加针对标记为已处理的事件而调用的处理程序。必须使用代码并调用 `AddHandler(RoutedEventArgs, Delegate, Boolean)` 的 `handledEventsToo` 重载。

#### ① 备注

在 XAML 中为相同事件指定事件处理程序时，请勿在 Visual Basic 代码中使用 `Handles` 语法。在这种情况下，将调用事件处理程序两次。

## WPF 如何实现“Handles”功能

编译 Extensible Application Markup Language (XAML) 页时，中间文件会声明对页中设置了 `Name` 属性（或声明了 `x:Name 指令`）的每个元素的 `Friend WithEvents` 引用。每个命名实例都可能是可通过 `Handles` 分配给处理程序的元素。

#### ① 备注

在 Visual Studio 中，IntelliSense 可完整显示可用于页中的 `Handles` 引用的全部元素。但是，这可能需要执行一个编译传递，以便中间文件可以填充所有 `Friends` 引用。

## 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](#)
- [WPF 中的 XAML](#)

# 弱事件模式

项目 · 2023/02/06

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) 引入了一种可用于解决此问题的设计模式，即为特定事件提供专用管理器类，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。

## 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。侦听事件的典型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，在 C# 中，语法为：`source.SomeEvent += new SomeEventHandler(MyEventHandler)`。

此技术可创建从事件源到事件侦听器的强引用。通常，为侦听器附加事件处理程序会导致侦听器的对象生存期受源对象生存期影响（除非显式删除事件处理程序）。但在某些情况下，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，而不是受源的生存期影响。每当源对象生存期超出侦听器的对象生存期时，正常的事件模式就会导致内存泄漏：侦听器的存活时间比预期的要长。

弱事件模式旨在解决此内存泄漏问题。每当侦听器需要注册事件时，都可以使用弱事件模式，但侦听器并不知晓事件会在何时注销。每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，有用由 you.）弱事件模式允许侦听器注册和接收事件，而不会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？

主要由控件作者来实现弱事件模式。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，源对象通常完全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

可通过三种方法实现弱事件模式。下表列出了这三种方法，并提供有关何时应使用每种方法的一些指导。

方法	何时实现
现有弱事件管理器类	如果要订阅的事件具有对应的 <a href="#">WeakEventManager</a> ，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 <a href="#">WeakEventManager</a> 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。
泛型弱事件管理器类	当现有 <a href="#">WeakEventManager</a> 不可用，而你想要一种简单的实现方法，且不在意效率问题时，可使用泛型 <a href="#">WeakEventManager&lt;TEventArgs&gt;</a> 。泛型 <a href="#">WeakEventManager&lt;TEventArgs&gt;</a> 效率低于现有的或自定义的弱事件管理器。例如，泛型类需要执行更多反射来发现给定事件名称的事件。此外，使用泛型 <a href="#">WeakEventManager&lt;TEventArgs&gt;</a> 注册事件的代码比使用现有或自定义 <a href="#">WeakEventManager</a> 注册事件的代码更详细。
自定义弱事件管理器类	当现有 <a href="#">WeakEventManager</a> 不可用同时又想要获得最佳效率时，可创建自定义 <a href="#">WeakEventManager</a> 。使用自定义 <a href="#">WeakEventManager</a> 订阅事件会更高效，但会在开始时编写更多代码。

## 方法 何时实现

使用 NuGet 有几个弱事件管理器，许多 WPF 框架也支持此模式（有关示例，请参阅 [Prism 关于松散耦合事件订阅的文档](#)）。

### 第三方弱事件管理器

以下部分介绍如何实现弱事件模式。在本讨论中，要订阅的事件有以下特征：

- 事件名称为 `SomeEvent`。
- 事件由 `EventSource` 类引发。
- 事件处理程序的类型为：`SomeEventEventHandler`（或 `EventHandler<SomeEventEventArgs>`）。
- 事件将 `SomeEventEventArgs` 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](#) 类中的继承层次结构。

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.SomeEvent += new SomeEventEventHandler(OnSomeEvent);
```

将其更改为以下模式：

C#

```
SomeEventWeakEventManager.AddHandler(source, OnSomeEvent);
```

同样，如果代码使用以下模式取消订阅事件：

C#

```
source.SomeEvent -= new SomeEventEventHandler(OnSomeEvent);
```

将其更改为以下模式：

C#

```
SomeEventWeakEventManager.RemoveHandler(source, OnSomeEvent);
```

## 使用泛型弱事件管理器类

1. 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并调用 `AddHandler`，如以下代码所示：

C#

```
WeakEventManager<EventSource, SomeEventArgs>.AddHandler(source,
    "SomeEvent", source_SomeEvent);
```

## 创建自定义弱事件管理器类

1. 将以下类模板复制到项目。

此类继承自 `WeakEventManager` 类。

C#

```
class SomeEventWeakEventManager : WeakEventManager
{
    private SomeEventWeakEventManager()
    {
    }

    /// <summary>
    /// Add a handler for the given source's event.
    /// </summary>
    public static void AddHandler(EventSource source,
        EventHandler<SomeEventArgs>
```

```

handler)
{
    if (source == null)
        throw new ArgumentNullException("source");
    if (handler == null)
        throw new ArgumentNullException("handler");

    CurrentManager.ProtectedAddHandler(source, handler);
}

/// <summary>
/// Remove a handler for the given source's event.
/// </summary>
public static void RemoveHandler(EventSource source,
                                  EventHandler<SomeEventEventArgs>
handler)
{
    if (source == null)
        throw new ArgumentNullException("source");
    if (handler == null)
        throw new ArgumentNullException("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<SomeEventEventArgs>();
}

```

```

/// <summary>
/// Listen to the given source for the event.
/// </summary>
protected override void StartListening(object source)
{
    EventSource typedSource = (EventSource)source;
    typedSource.SomeEvent += new EventHandler<SomeEventArgs>(OnSomeEvent);
}

/// <summary>
/// Stop listening to the given source for the event.
/// </summary>
protected override void StopListening(object source)
{
    EventSource typedSource = (EventSource)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. 使用自己名称替换 `SomeEventWeakEventManager` 名称。
3. 将前面所述的三个名称替换为事件对应的名称。 (`SomeEvent`、`EventSource` 和 `SomeEventArgs` )
4. 将弱事件管理器类的可见性 ( 公共/内部/专用 ) 设置为与其所管理的事件相同的可见性。
5. 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 SomeEventArgs(OnSomeEvent);
```

将其更改为以下模式 :

C#

```
SomeEventWeakEventManager.AddHandler(source, OnSomeEvent);
```

同样，如果代码使用以下模式取消订阅事件：

C#

```
source.SomeEvent -= new SomeEventEventHandler(OnSome);
```

将其更改为以下模式：

C#

```
SomeEventWeakEventManager.RemoveHandler(source, OnSomeEvent);
```

## 另请参阅

- [WeakEventManager](#)
- [IWeakEventListener](#)
- [路由事件概述](#)
- [数据绑定概述](#)

# 事件帮助主题

项目 • 2023/02/06

本部分中的主题介绍如何在 WPF 中使用事件。

## 本节内容

[使用代码添加事件处理程序](#)

[处理路由事件](#)

[创建自定义路由事件](#)

[在事件处理程序中查找源元素](#)

[为路由事件添加类处理](#)

## 参考

[RoutedEventArgs](#)

[EventManager](#)

[RoutingStrategy](#)

## 相关章节

# 如何：使用代码添加事件处理程序

项目 • 2023/02/06

此示例演示如何使用代码向元素添加事件处理程序。

如果要将事件处理程序添加到 XAML 元素，并且已加载包含该元素的标记页面，则必须使用代码添加处理程序。或者，如果你完全使用代码为应用程序构建元素树，而不使用 XAML 声明任何元素，则可以调用特定方法将事件处理程序添加到构造的元素树。

## 示例

以下示例将新的 `Button` 添加到最初在 XAML 中定义的现有页面。代码隐藏文件实现事件处理程序方法，然后将该方法添加为 `Button` 上的新事件处理程序。

C# 示例使用 `+=` 运算符将处理程序分配给事件。这与用于在公共语言运行时 (CLR) 事件处理模型中分配处理程序的运算符相同。Microsoft Visual Basic 不支持将此运算符作为添加事件处理程序的一种方式。相反，它需要以下两种技术之一：

- 使用 `AddHandler` 方法和 `AddressOf` 运算符来引用事件处理程序实现。
- 使用 `Handles` 关键字作为事件处理程序定义的一部分。此处未说明此技术；请参阅 [Visual Basic 和 WPF 事件处理](#)。

XAML

```
<TextBlock Name="text1">Start by clicking the button below</TextBlock>
<Button Name="b1" Click="MakeButton">Make new button and add handler to
it</Button>
```

C#

```
public partial class RoutedEventAddRemoveHandler {
    void MakeButton(object sender, RoutedEventArgs e)
    {
        Button b2 = new Button();
        b2.Content = "New Button";
        // Associate event handler to the button. You can remove the event
        // handler using "-=" syntax rather than "+=".
        b2.Click += new RoutedEventHandler(Onb2Click);
        root.Children.Insert(root.Children.Count, b2);
        DockPanel.SetDock(b2, Dock.Top);
        text1.Text = "Now click the second button...";
        b1.IsEnabled = false;
    }
    void Onb2Click(object sender, RoutedEventArgs e)
```

```
{  
    text1.Text = "New Button (b2) Was Clicked!!";  
}
```

### ① 备注

在最初解析的 XAML 页面中添加事件处理程序要简单得多。在要添加事件处理程序的对象元素中，添加与要处理的事件名称一致的属性。然后将该属性的值指定为你在 XAML 页面的代码隐藏文件中定义的事件处理程序方法的名称。有关更多信息，请参阅 [WPF 中的 XAML 或路由事件概述](#)。

## 另请参阅

- [路由事件概述](#)
- [操作指南主题](#)

# 如何：处理路由事件

项目 • 2023/02/06

本示例演示了浮升事件的工作原理，以及如何编写可处理路由事件数据的处理程序。

## 示例

在 Windows Presentation Foundation (WPF) 中，元素以元素树结构形式排列。父元素可以参与处理最初由元素树中的子元素引发的事件。这都是因为事件路由。

路由事件通常遵循以下两个路由策略之一：浮升和隧道。本示例主要针对浮升事件，并且使用 [ButtonBase.Click](#) 事件演示路由的工作原理。

以下示例创建两个 [Button](#) 控件，并使用 XAML 特性语法将事件处理程序附加到公用父元素（在本示例中为 [StackPanel](#)）。本示例使用特性语法将事件处理程序附加到 [StackPanel](#) 父元素，而不是为每个 [Button](#) 子元素附加各自的事件处理程序。此事件处理模式展示了如何使用事件路由技术来减少附加处理程序的元素数。每个 [Button](#) 的所有浮升事件都通过父元素路由。

请注意，在父 [StackPanel](#) 元素上，指定为特性的 [Click](#) 事件名称可通过命名 [Button](#) 类来进行部分限定。[Button](#) 类是一个 [ButtonBase](#) 派生类，在其成员列表中有 [Click](#) 事件。如果要处理的事件不在附加路由事件处理程序的元素的成员列表中，则有必要使用这种部分限定技术来附加事件处理程序。

XAML

```
<StackPanel
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.RoutedEventHandle"
    Name="dpanel"
    Button.Click="HandleClick">
    <StackPanel.Resources>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Height" Value="20"/>
            <Setter Property="Width" Value="250"/>
            <Setter Property="HorizontalAlignment" Value="Left"/>
        </Style>
    </StackPanel.Resources>
    <Button Name="Button1">Item 1</Button>
    <Button Name="Button2">Item 2</Button>
    <TextBlock Name="results"/>
</StackPanel>
```

下面的示例处理 [Click](#) 事件。 该示例会报告哪个元素处理事件以及哪个元素引发事件。 用户单击任一按钮时都将执行事件处理程序。

C#

```
public partial class RoutedEventHandle : StackPanel
{
    StringBuilder eventstr = new StringBuilder();
    void HandleClick(object sender, RoutedEventArgs args)
    {
        // Get the element that handled the event.
        FrameworkElement fe = (FrameworkElement)sender;
        eventstr.Append("Event handled by element named ");
        eventstr.Append(fe.Name);
        eventstr.Append("\n");

        // Get the element that raised the event.
        FrameworkElement fe2 = (FrameworkElement)args.Source;
        eventstr.Append("Event originated from source element of type ");
        eventstr.Append(args.Source.GetType().ToString());
        eventstr.Append(" with Name ");
        eventstr.Append(fe2.Name);
        eventstr.Append("\n");

        // Get the routing strategy.
        eventstr.Append("Event used routing strategy ");
        eventstr.Append(args.RoutedEventArgs.RoutingStrategy);
        eventstr.Append("\n");

        results.Text = eventstr.ToString();
    }
}
```

## 请参阅

- [RoutedEventArgs](#)
- [输入概述](#)
- [路由事件概述](#)
- [操作指南主题](#)
- [XAML 语法详述](#)

# 如何：创建自定义路由事件

项目 • 2023/02/06

若要使你的自定义事件支持事件路由，需要使用 [RegisterRoutedEvent](#) 方法注册 [RoutedEvent](#)。本示例演示了创建自定义路由事件的基本原理。

## 示例

如以下示例所示，首先使用 [RegisterRoutedEvent](#) 方法注册 [RoutedEvent](#)。按照约定，[RoutedEvent](#) 静态字段名称应以 Event 后缀结尾。在此示例中，事件的名称是 `Tap`，事件的路由策略是 `Bubble`。在注册调用之后，可以为事件提供添加和移除公共语言运行时 (CLR) 事件访问器。

请注意，尽管事件在此特定示例中是通过 `OnTap` 虚拟方法引发的，但引发事件的方式或事件响应更改的方式取决于你的需要。

另请注意，此示例基本上实现了 [Button](#) 的整个子类；该子类构建为单独的程序集，然后在单独的 Windows Presentation Foundation (WPF) 元素上实例化为自定义类。

C#

```
public class MyButtonSimple: Button
{
    // Create a custom routed event by first registering a RoutedEventID
    // This event uses the bubbling routing strategy
    public static readonly RoutedEvent TapEvent =
        EventManager.RegisterRoutedEvent(
            "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler),
            typeof(MyButtonSimple));

    // Provide CLR accessors for the event
    public event RoutedEventHandler Tap
    {
        add { AddHandler(TapEvent, value); }
        remove { RemoveHandler(TapEvent, value); }
    }

    // This method raises the Tap event
    void RaiseTapEvent()
    {
        RoutedEventArgs newEventArgs = new
        RoutedEventArgs(MyButtonSimple.TapEvent);
        RaiseEvent(newEventArgs);
    }
    // For demonstration purposes we raise the event when the MyButtonSimple
    // is clicked
    protected override void OnClick()
```

```
{  
    RaiseTapEvent();  
}  
}
```

## XAML

```
<Window  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"  
    x:Class="SDKSample.RoutedEventCustomApp"  
  
>  
<Window.Resources>  
    <Style TargetType="{x:Type custom:MyButtonSimple}">  
        <Setter Property="Height" Value="20"/>  
        <Setter Property="Width" Value="250"/>  
        <Setter Property="HorizontalAlignment" Value="Left"/>  
        <Setter Property="Background" Value="#808080"/>  
    </Style>  
</Window.Resources>  
<StackPanel Background="LightGray">  
    <custom:MyButtonSimple Name="mybtnsimple" Tap="TapHandler">Click to  
    see Tap custom event work</custom:MyButtonSimple>  
</StackPanel>  
</Window>
```

隧道事件的创建方式相同，但在注册调用中将 [RoutingStrategy](#) 设置为 [Tunnel](#)。按照约定，WPF 中的隧道事件以单词“Preview”开头。

若要查看浮升事件的工作原理示例，请参阅[处理路由事件](#)。

## 另请参阅

- [路由事件概述](#)
- [输入概述](#)
- [控件创作概述](#)

# 如何：在事件处理程序中查找源元素

项目 • 2023/02/06

此示例演示如何在事件处理程序中查找源元素。

## 示例

下面的示例演示在代码隐藏文件中声明的 Click 事件处理程序。当用户单击附加到的按钮时，处理程序将更改属性值。处理程序代码使用事件参数中报告的路由事件数据的 Source 属性来更改 Source 元素上的 Width 属性值。

XAML

```
<Button Click="HandleClick">Button 1</Button>
```

C#

```
void HandleClick(object sender, RoutedEventArgs e)
{
    // You must cast the sender object as a Button element, or at least as
    FrameworkElement, to set Width
    Button srcButton = e.Source as Button;
    srcButton.Width = 200;
}
```

## 另请参阅

- [RoutedEventArgs](#)
- [路由事件概述](#)
- [操作指南主题](#)

# 如何：为路由事件添加类处理

项目 • 2023/02/06

路由事件可以由路由中任何给定节点上的类处理程序或实例处理程序处理。类处理程序首先被调用，并且可以由类实现用于抑制实例处理中的事件，或在基类拥有的事件上引入其他特定于事件的行为。此示例说明了用于实现类处理程序的两种密切相关的技术。

## 示例

此示例使用基于 [Canvas](#) 面板的自定义类。应用程序的基本前提是，自定义类在其子元素上引入行为，包括在调用子元素类或其上的任何实例处理程序之前拦截任何鼠标左键单击操作并将其标记为“已处理”。

[UIElement](#) 类公开了一个虚拟方法，该方法通过简单地重写事件来启用对 [PreviewMouseLeftButtonDown](#) 事件的类处理。如果这样的虚拟方法在你的类层次结构中的某处可用，则这是实现类处理的最简单方法。以下代码显示了派生自 [Canvas](#) 的 “MyEditContainer” 中的 [OnPreviewMouseLeftButtonDown](#) 实现。该实现将事件标记为在参数中已处理，然后添加一些代码，以便为源元素提供基本的可见更改。

C#

```
protected override void  
OnPreviewMouseRightButtonDown(System.Windows.Input.MouseButtonEventArgs e)  
{  
    e.Handled = true; //suppress the click event and other  
leftmousebuttondown responders  
    MyEditContainer ec = (MyEditContainer)e.Source;  
    if (ec.EditState)  
    { ec.EditState = false; }  
    else  
    { ec.EditState = true; }  
    base.OnPreviewMouseRightButtonDown(e);  
}
```

如果基类或该特定方法上没有可用的虚拟方法，则可以使用 [EventManager](#) 类的实用方法 [RegisterClassHandler](#) 直接添加类处理。只能在要添加类处理的类的静态初始化中调用此方法。此示例为 [PreviewMouseLeftButtonDown](#) 添加了另一个处理程序，在这种情况下，注册的类是自定义类。相反，当使用虚拟方法时，注册的类实际上是 [UIElement](#) 基类。在基类和子类各自注册类处理的情况下，将首先调用子类处理程序。应用程序中的行为是，首先此处理程序将显示其消息框，然后将显示虚拟方法处理程序中的视觉更改。

C#

```
static MyEditContainer()
{
    EventManager.RegisterClassHandler(typeof(MyEditContainer),
PreviewMouseRightButtonDownEvent, new
RoutedEventHandler(LocalOnMouseRightButtonDown));
}
internal static void LocalOnMouseRightButtonDown(object sender,
RoutedEventArgs e)
{
    MessageBox.Show("this is invoked before the On* class handler on
UIElement");
    //e.Handled = true; //uncommenting this would cause ONLY the subclass'
class handler to respond
}
```

## 另请参阅

- [EventManager](#)
- [将路由事件标记为“已处理”和“类处理”](#)
- [处理路由事件](#)
- [路由事件概述](#)

# 输入 (WPF)

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 包括对若干输入类型的支持。此输入包括文本、触摸、鼠标、命令、焦点、触摸、拖放和数字墨迹。本部分介绍与 WPF 中的输入相关的主题。

## 本节内容

- [输入概述](#)
- [命令概述](#)
- [焦点概述](#)
- [为控件中的焦点设置样式以及 FocusVisualStyle](#)
- [演练：创建你的第一个触控应用程序](#)
- [操作指南主题](#)
- [数字墨迹](#)

## 参考

- [UIElement](#)
- [FrameworkElement](#)
- [ContentElement](#)
- [FrameworkContentElement](#)
- [Keyboard](#)
- [Mouse](#)
- [FocusManager](#)

## 相关章节

- [控件](#)
- [事件](#)

# 输入概述

项目 · 2023/02/06

WPF 并说明了输入系统的体系结构。

## 输入 API

主要输入 API 公开存在于以下基元素类上：[UIElement](#)、[ContentElement](#)、[FrameworkElement](#) 和 [FrameworkContentElement](#)。有关基元素的详细信息，请参阅[基元素概述](#)。这些类提供有关输入事件（例如按键、鼠标按钮、鼠标滚轮、鼠标移动、焦点管理和鼠标捕获等）的功能。通过将输入 API 放置在基元素上，而不是将所有输入事件视作一项服务，该输入体系结构使输入事件可以由 UI 中的特定对象指明其出处，并支持事件路由方案，从而使得多个元素有机会处理输入事件。许多输入事件都具有与之相关联的一对事件。例如，键盘按下事件与 [KeyDown](#) 和 [PreviewKeyDown](#) 事件相关联。这些事件的区别在于它们如何路由至目标元素。预览事件将元素树从根元素到目标元素向下进行隧道操作。冒泡事件从目标元素到根元素向上进行冒泡操作。WPF 中的事件路由在本概述的后面和[路由事件概述](#)中有更详细的讨论。

## 键盘和鼠标类

除了基元素类上的输入 API 之外，[Keyboard](#) 类和 [Mouse](#) 类还提供了更多 API 来处理键盘和鼠标输入。

[Keyboard](#) 类上的输入 API 示例包括可返回当前按下的 [ModifierKeys](#) 的 [Modifiers](#) 属性和可确定是否按下了特定键的 [IsKeyDown](#) 方法。

以下示例使用 [GetKeyStates](#) 方法来确定 [Key](#) 是否处于按下状态。

C#

```
// Uses the Keyboard.GetKeyStates to determine if a key is down.  
// A bitwise AND operation is used in the comparison.  
// e is an instance of KeyEventArgs.  
if ((Keyboard.GetKeyStates(Key.Return) & KeyStates.Down) > 0)  
{  
    btnNone.Background = Brushes.Red;  
}
```

[Mouse](#) 类上的输入 API 示例包括可获取鼠标中键状态的 [MiddleButton](#) 和可获得鼠标指针当前指向的元素的 [DirectlyOver](#)。

以下示例确定了鼠标的 [LeftButton](#) 是否处于 [Pressed](#) 状态。

C#

```
if (Mouse.LeftButton == MouseButtonState.Pressed)
{
    UpdateSampleResults("Left Button Pressed");
}
```

[Mouse](#) 和 [Keyboard](#) 类在本概述中有更详细的介绍。

## 触笔输入

WPF 集成了对 [Stylus](#) 的支持。 [Stylus](#) 是因 Tablet PC 而变得普及的一种笔输入。 WPF 应用程序可以通过使用鼠标 API 将触笔视为鼠标，但 API 也公开了触笔设备抽象，其使用的模型与键盘和鼠标类似。所有与触笔相关的 API 都包含单词“[Stylus](#)”（触笔）。

由于触笔可充当鼠标，因此仅支持鼠标输入的应用程序仍可以自动获得一定程度的触笔支持。以这种方式使用触笔时，应用程序有能力处理相应的触笔事件，然后处理相应的鼠标事件。此外，通过触笔设备抽象也可以使用墨迹输入等较高级别的服务。有关墨迹输入的详细信息，请参阅[墨迹入门](#)。

## 事件路由

[FrameworkElement](#) 可以在其内容模型中包含其他元素作为子元素，从而形成一个元素树。在 WPF 中，父元素可以通过处理事件来参与定向到其子元素或其他后代的输入。这对于从较小的控件生成控件（该过程称为“控件组合”或“组合”）特别有用。有关元素树以及元素树与事件路由的关系的详细信息，请参阅[WPF 中的树](#)。

事件路由是将事件转发到多个元素的过程，以便使路由中的特定对象或元素可以选择对已由其他元素指明来源的事件提供重要响应（通过处理）。路由事件使用三种路由机制的其中一种：直接、浮升和隧道。在直接路由中，源元素是收到通知的唯一元素，事件不会路由至任何其他元素。但是相对于标准 CLR 事件，直接路由事件仍然提供一些仅针对路由事件而存在的其他功能。浮升操作在元素树中向上进行，首先通知指明了事件来源的第一个元素，然后是父元素等等。隧道操作从元素树的根开始，然后向下进行，以原始的源元素结束。有关路由事件的详细信息，请参阅[路由事件概述](#)。

WPF 输入事件通常成对出现，由一个隧道事件和一个浮升事件组成。隧道事件与冒泡事件的不同之处在于它有“预览”前缀。例如，[PreviewMouseMove](#) 是鼠标移动事件的隧道版本，[MouseMove](#) 是此事件的浮升版本。此事件配对是在元素级别实现的一种约定，不是 WPF 事件系统的固有功能。有关详细信息，请参阅[路由事件概述](#)中的 WPF 输入事件部分。

# 处理输入事件

若要在元素上接收输入，必须将事件处理程序与该特定事件关联。在 XAML 中，这很简单：将事件的名称作为要侦听此事件的元素的特性进行引用。然后，根据委托，将特性的值设置为所定义的事件处理程序的名称。事件处理程序必须用代码（例如 C#）编写，并且可以包含在代码隐藏文件中。

当操作系统报告发生键操作时，如果键盘焦点正处在元素上，则将发生键盘事件。鼠标和触笔事件分别分为两类：报告指针位置相对于元素的变化的事件，和报告设备按钮状态的变化的事件。

## 键盘输入事件示例

以下示例侦听按下向左键的操作。创建了一个具有 [Button](#) 的 [StackPanel](#)。用于侦听按下向左键的事件处理程序附加到了 [Button](#) 实例。

示例的第一部分创建了 [StackPanel](#) 和 [Button](#)，并且附加了 [KeyDown](#) 的事件处理程序。

XAML

```
<StackPanel>
    <Button Background="AliceBlue"
        KeyDown="OnButtonKeyDown"
        Content="Button1"/>
</StackPanel>
```

C#

```
// Create the UI elements.
StackPanel keyboardStackPanel = new StackPanel();
Button keyboardButton1 = new Button();

// Set properties on Buttons.
keyboardButton1.Background = Brushes.AliceBlue;
keyboardButton1.Content = "Button 1";

// Attach Buttons to StackPanel.
keyboardStackPanel.Children.Add(keyboardButton1);

// Attach event handler.
keyboardButton1.KeyDown += new KeyEventHandler(OnButtonKeyDown);
```

第二部分用代码编写，定义了事件处理程序。按下向左键且 [Button](#) 具有键盘焦点时，处理程序会运行，且 [Button](#) 的 [Background](#) 颜色会发生变化。如果按下一个键，但不是向左键，[Button](#) 的 [Background](#) 颜色将变回其初始颜色。

C#

```
private void OnButtonKeyDown(object sender, KeyEventArgs e)
{
    Button source = e.Source as Button;
    if (source != null)
    {
        if (e.Key == Key.Left)
        {
            source.Background = Brushes.LemonChiffon;
        }
        else
        {
            source.Background = Brushes.AliceBlue;
        }
    }
}
```

## 鼠标输入事件示例

在下面的示例中，当鼠标指针进入 Button 时，Button 的 Background 颜色会发生变化。当鼠标离开 Button 时，Background 颜色将还原。

示例的第一部分创建了 StackPanel 和 Button 控件，并将 MouseEnter 和 MouseLeave 事件附加到了 Button。

XAML

```
<StackPanel>
    <Button Background="AliceBlue"
            MouseEnter="OnMouseExampleMouseEnter"
            MouseLeave="OnMosueExampleMouseLeave">Button

    </Button>
</StackPanel>
```

C#

```
// Create the UI elements.
StackPanel mouseMoveStackPanel = new StackPanel();
Button mouseMoveButton = new Button();

// Set properties on Button.
mouseMoveButton.Background = Brushes.AliceBlue;
mouseMoveButton.Content = "Button";

// Attach Buttons to StackPanel.
mouseMoveStackPanel.Children.Add(mouseMoveButton);
```

```
// Attach event handler.  
mouseMoveButton.MouseEnter += new  
MouseEventHandler(OnMouseExampleMouseEnter);  
mouseMoveButton.MouseLeave += new  
MouseEventHandler(OnMosueExampleMouseLeave);
```

该示例的第二部分用代码编写，定义了事件处理程序。当鼠标进入 Button 时，Button 的 `Background` 颜色将变为 `SlateGray`。当鼠标离开 Button 时，Button 的 `Background` 颜色将变回 `AliceBlue`。

C#

```
private void OnMouseExampleMouseEnter(object sender, MouseEventArgs e)  
{  
    // Cast the source of the event to a Button.  
    Button source = e.Source as Button;  
  
    // If source is a Button.  
    if (source != null)  
    {  
        source.Background = Brushes.SlateGray;  
    }  
}
```

C#

```
private void OnMosueExampleMouseLeave(object sender, MouseEventArgs e)  
{  
    // Cast the source of the event to a Button.  
    Button source = e.Source as Button;  
  
    // If source is a Button.  
    if (source != null)  
    {  
        source.Background = Brushes.AliceBlue;  
    }  
}
```

## 文本输入

通过 `TextInput` 事件，你能够借助与设备无关的方式侦听文本输入。键盘是文本输入的主要方式，但通过语音、手写和其他输入设备也可以生成文本输入。

对于键盘输入，WPF 首先发送相应的 `KeyDown/KeyUp` 事件。如果未处理这些事件，并且按键是文本（而不是诸如方向箭头或功能键之类的控制键），则将引发 `TextInput` 事件。`KeyDown/KeyUp` 和 `TextInput` 事件之间并不总是简单的一对一映射，因为多次击键可以生成单个字符的文本输入，而单次击键可以生成多个字符串。对于中文、日文和韩

文等语言尤其如此，这些语言使用输入法编辑器 (IME) 生成由其对应的字母组成的成千上万个可能的字符。

当 WPF 发送 `KeyUp/KeyDown` 事件时，如果击键可能成为 `TextInput` 事件的一部分（例如按下 `ALT+S`），`Key` 将设置为 `Key.System`。这允许 `KeyDown` 事件处理程序中的代码检查是否存在 `Key.System`，如果有，则留给随后引发的 `TextInput` 事件的处理程序处理。在这些情况下，可以使用 `TextCompositionEventArgs` 参数的各种属性来确定原始击键。同样，如果 IME 处于活动状态，则 `Key` 具有值 `Key.ImeProcessed`，且 `ImeProcessedKey` 会提供一个或多个原始击键。

以下示例定义了 `Click` 事件的处理程序和 `KeyDown` 事件的处理程序。

第一段代码或标记创建用户界面。

XAML

```
<StackPanel KeyDown="OnTextInputKeyDown">
    <Button Click="OnTextInputButtonClick"
        Content="Open" />
    <TextBox> . . . </TextBox>
</StackPanel>
```

C#

```
// Create the UI elements.
StackPanel textInputStackPanel = new StackPanel();
Button textInputButton = new Button();
TextBox textInputTextBox = new TextBox();
textInputButton.Content = "Open";

// Attach elements to StackPanel.
textInputStackPanel.Children.Add(textInputButton);
textInputStackPanel.Children.Add(textInputTextBox);

// Attach event handlers.
textInputStackPanel.KeyDown += new KeyEventHandler(OnTextInputKeyDown);
textInputButton.Click += new RoutedEventHandler(OnTextInputButtonClick);
```

第二段代码包含事件处理程序。

C#

```
private void OnTextInputKeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.O && Keyboard.Modifiers == ModifierKeys.Control)
    {
        handle();
        e.Handled = true;
    }
}
```

```
}

private void OnTextInputButtonClick(object sender, RoutedEventArgs e)
{
    handle();
    e.Handled = true;
}

public void handle()
{
    MessageBox.Show("Pretend this opens a file");
}
```

由于输入事件在事件路由中向上浮升，因此不管哪个元素具有键盘焦点，`StackPanel` 都将接收输入。会首先通知 `TextBox` 控件，仅当 `TextBox` 不处理输入时，才会调用 `OnTextInputKeyDown` 处理程序。如果使用了 `PreviewKeyDown` 事件而不是 `KeyDown` 事件，将首先调用 `OnTextInputKeyDown` 处理程序。

在此示例中，处理逻辑写入了两次，分别针对 `CTRL+O` 和按钮的单击事件。使用命令，而不是直接处理输入事件，可简化此过程。本概述和[命令概述](#)中将讨论这些命令。

## 触摸和操作

Windows 7 操作系统中的新硬件和 API 使应用程序能够同时接收来自多个触控的输入。WPF 通过在触摸发生时引发事件，使应用程序能够以类似于响应其他输入（例如鼠标或键盘）的方式来检测和响应触摸设备。

发生触摸时，WPF 将公开两种类型的事件：触摸事件和操作事件。触摸事件提供有关触摸屏上每个手指及其移动的原始数据。操作事件将输入解释为特定操作。本部分将讨论这两种类型的事件。

## 先决条件

需要以下组件才能开发响应触摸的应用程序。

- Visual Studio 2010。
- Windows 7。
- 支持 Windows 触控的设备，如触摸屏。

## 术语

讨论触摸时使用了以下术语。

- **触摸**是 Windows 7 可识别的一种用户输入。通常，将手指放在触敏式屏幕上会触发触摸。请注意，如果设备仅将手指的位置和移动转换为鼠标输入，则笔记本电脑上常用的触摸板等设备不支持触摸。
- **多点触摸**是同时发生在多个点上的触摸。Windows 7 和 WPF 支持多点触摸。WPF 文档中每当论及触摸时，相关概念均适用于多点触摸。
- 当触摸被解释为应用于对象的实际操作时，就发生了**操作**。在 WPF 中，操作事件将输入解释为平移、展开或旋转操作。
- `touch device` 表示产生触摸输入的设备，例如触摸屏上的一根手指。

## 响应触摸的控件

如果以下控件的内容延伸到视图之外，则可以通过在控件上拖动手指来滚动该控件。

- [ComboBox](#)
- [ContextMenu](#)
- [DataGrid](#)
- [ListBox](#)
- [ListView](#)
- [MenuItem](#)
- [TextBox](#)
- [ToolBar](#)
- [TreeView](#)

[ScrollViewer](#) 定义了 [ScrollViewer.PanningMode](#) 附加属性，该属性让你可以指定是水平、垂直、同时以这两种方式还是不以任何一种方式启用触摸移动。

[ScrollViewer.PanningDeceleration](#) 属性指定当用户从触摸屏上抬起手指时滚动速度减慢的速度。[ScrollViewer.PanningRatio](#) 附加属性指定滚动偏移与平移操作偏移的比率。

## 触摸事件

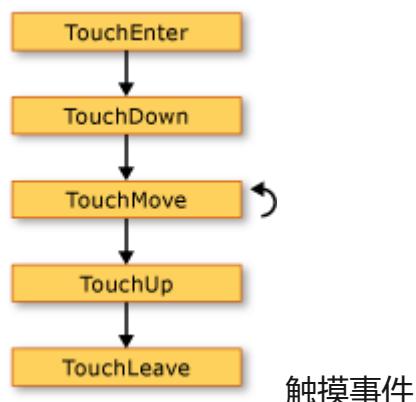
基类 [UIElement](#)、[UIElement3D](#) 和 [ContentElement](#) 定义你可以订阅的事件，以便你的应用程序对触摸做出响应。当应用程序将触摸解释为操作对象以外的其他操作时，触摸事件非常有用。例如，使用户能够以一个或多个手指绘制的应用程序将订阅触摸事件。

所有三个类都定义了以下事件，其行为类似，而无论定义类是什么。

- `TouchDown`
- `TouchMove`
- `TouchUp`
- `TouchEnter`
- `TouchLeave`
- `PreviewTouchDown`
- `PreviewTouchMove`
- `PreviewTouchUp`
- `GotTouchCapture`
- `LostTouchCapture`

像键盘和鼠标事件一样，触摸事件也是路由事件。以 `Preview` 开头的事件是隧道事件，以 `Touch` 开头的事件是冒泡事件。有关路由事件的详细信息，请参阅[路由事件概述](#)。当你处理这些事件时，可以通过调用 `GetTouchPoint` 或 `GetIntermediateTouchPoints` 方法获得输入相对于任何元素的位置。

为了理解触控事件之间的交互，请考虑以下这种情况：用户将一个手指放在元素上，在该元素中移动手指，然后将手指从该元素上移开。下图显示了冒泡事件的执行（为简单起见，省略了隧道事件）。



下列内容描述了上图中的事件顺序。

1. 当用户将手指放在元素上时，会发生一次 `TouchEnter` 事件。
2. 会发生一次 `TouchDown` 事件。
3. 当用户在元素中移动手指时，会发生多次 `TouchMove` 事件。

4. 当用户从元素抬起手指时，会发生一次 [TouchUp](#) 事件。

5. 会发生一次 [TouchLeave](#) 事件。

当使用两根以上的手指时，每根手指都会发生事件。

## 操作事件

对于应用程序支持用户操作对象的情况，[UIElement](#) 类定义了操作事件。与只是报告触摸位置的触摸事件不同，操作事件会报告可采用何种方式解释输入。有三种类型的操作：转换、扩展和旋转。下列内容介绍了如何调用这三种类型的操作。

- 将一根手指放在对象上，并在触摸屏上拖动手指以调用转换操作。此操作通常会移动对象。
- 将两根手指放在物体上，并将手指相互靠拢或分开以调用扩展操作。此操作通常会调整对象的大小。
- 将两根手指放在对象上，并将一个手指围绕另一个手指旋转以调用旋转操作。此操作通常会旋转对象。

多种类型的操作可以同时发生。

使对象响应操作时，可以让对象看起来具有惯性。这样可以使对象模拟真实的世界。例如，在桌子上推一本书时，如果你足够用力，书将在你松手后继续移动。利用 WPF，可以通过在用户的指松开对象后引发操作事件来模拟这种行为。

如需深入了解如何创建使用户可以对对象进行移动、调整大小和旋转的应用程序，请参阅[演练：创建你的第一个触控应用程序](#)。

[UIElement](#) 定义了以下操作事件。

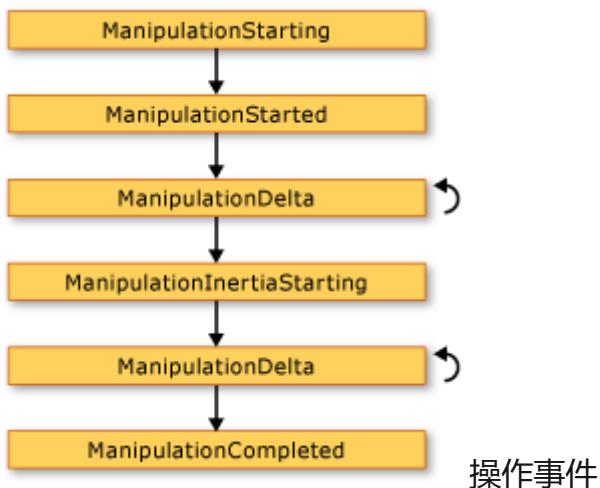
- [ManipulationStarting](#)
- [ManipulationStarted](#)
- [ManipulationDelta](#)
- [ManipulationInertiaStarting](#)
- [ManipulationCompleted](#)
- [ManipulationBoundaryFeedback](#)

默认情况下，[UIElement](#) 不会收到这些操作事件。若要在 [UIElement](#) 上收到操作事件，请将 [UIElement.IsManipulationEnabled](#) 设置为 `true`。

## 操作事件的执行路径

考虑用户“抛出”一个对象的情况。 用户将手指放在对象上，将手指在触摸屏上移动一段短距离，然后在移动的同时抬起手指。 此操作的结果是，该对象将在用户的手指下方移动，并在用户抬起手指后继续移动。

下图显示了操作事件的执行路径和每个事件的重要信息。



下列内容描述了上图中的事件顺序。

1. 当用户将手指放在对象上时，会发生 [ManipulationStarting](#) 事件。此外，此事件还允许你设置 [ManipulationContainer](#) 属性。在后续事件中，操作的位置将相对于 [ManipulationContainer](#)。在除 [ManipulationStarting](#) 之外的事件中，此属性是只读的，因此你只能在发生 [ManipulationStarting](#) 事件时设置此属性。
2. 接下来会发生 [ManipulationStarted](#) 事件。此事件报告操作的原始位置。
3. 当用户的手指在触摸屏上移动时，会发生多次 [ManipulationDelta](#) 事件。  
[ManipulationDeltaEventArgs](#) 类的 [DeltaManipulation](#) 属性报告操作是解释为移动、展开还是平移。这是你执行操作对象的大部分工作的地方。
4. 当用户的手指与对象失去接触时，会发生 [ManipulationInertiaStarting](#) 事件。此事件使你可以指定操作在惯性期间的减速。这样，选择时对象就可以模拟不同的物理空间或特性。例如，假设应用程序有两个表示真实世界中的物品的对象，并且一个物品比另一个物品重。你可以使较重的对象比较轻的对象减速更快。
5. 在发生惯性时，会发生多次 [ManipulationDelta](#) 事件。请注意，当用户的手指在触摸屏上移动并且 WPF 模拟惯性时，将发生此事件。换句话说，在 [ManipulationInertiaStarting](#) 事件之前和之后，会发生 [ManipulationDelta](#)。  
[ManipulationDeltaEventArgs](#).[IsInertial](#) 属性报告在惯性期间是否发生了 [ManipulationDelta](#) 事件，以便你可以检查该属性并根据其值执行不同的操作。
6. 当操作和任何惯性结束时，会发生 [ManipulationCompleted](#) 事件。也就是说，在所有 [ManipulationDelta](#) 事件发生后，会发生 [ManipulationCompleted](#) 事件以指示操

作已完成。

[UIElement](#) 还定义了 [ManipulationBoundaryFeedback](#) 事件。在 [ManipulationDelta](#) 事件中调用 [ReportBoundaryFeedback](#) 方法时，会发生此事件。

[ManipulationBoundaryFeedback](#) 事件使应用程序或组件可以在对象到达边界时提供可视反馈。例如，[Window](#) 类会处理 [ManipulationBoundaryFeedback](#) 事件，以便在到达窗口边缘时使窗口轻微移动。

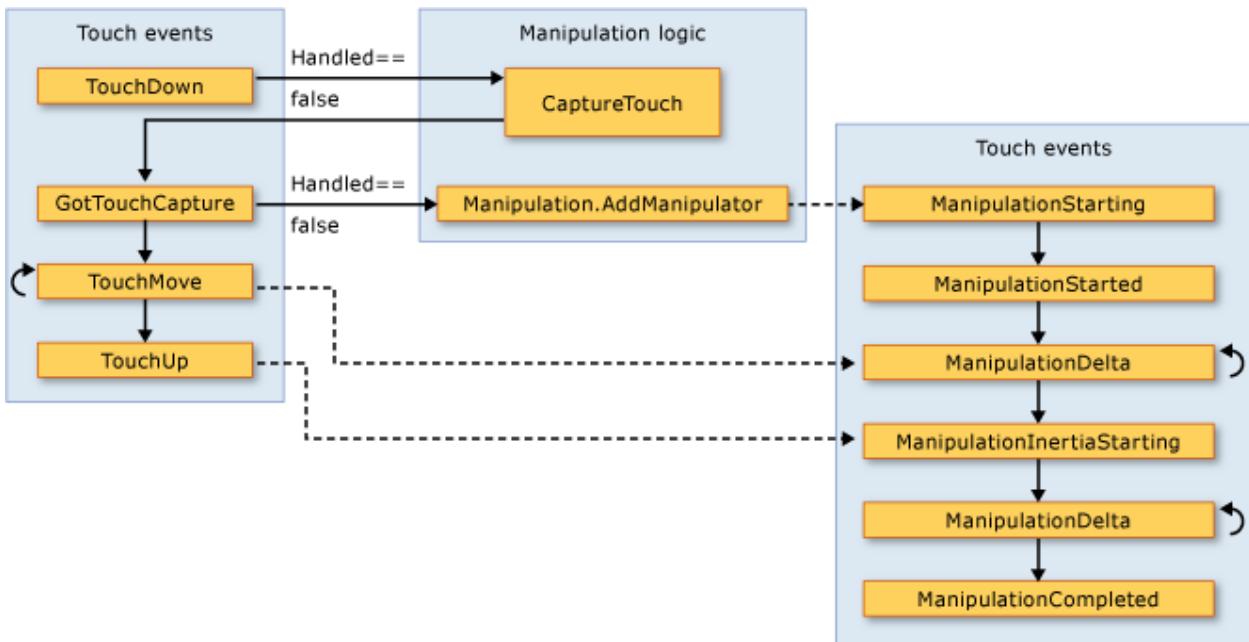
你可以通过对任意操作事件中的事件参数调用 [Cancel](#) 方法来取消操作，[ManipulationBoundaryFeedback](#) 除外。当你调用 [Cancel](#) 时，不再引发操作事件，触摸会发生鼠标事件。下表描述了取消操作的时间与所发生的鼠标事件之间的关系。

在其中调用取消的事件	针对已经发生的输入发生的鼠标事件
<a href="#">ManipulationStarting</a> 和 <a href="#">ManipulationStarted</a>	鼠标按下事件。
<a href="#">ManipulationDelta</a>	鼠标按下和鼠标移动事件。
<a href="#">ManipulationInertiaStarting</a> 和 <a href="#">ManipulationCompleted</a>	鼠标按下、鼠标移动和鼠标弹起事件。

请注意，如果你在操作处于惯性期间调用 [Cancel](#)，则该方法返回 `false`，并且输入不会引发鼠标事件。

## 触摸事件和操作事件之间的关系

[UIElement](#) 可以始终收到触摸事件。当 [IsManipulationEnabled](#) 属性设置为 `true` 时，[UIElement](#) 会同时收到触摸和操作事件。如果未处理 [TouchDown](#) 事件（即 [Handled](#) 属性为 `false`），则操作逻辑会将触摸捕获到元素并生成操作事件。如果 [Handled](#) 属性在 [TouchDown](#) 事件中设置为 `true`，则操作逻辑不会生成操作事件。下图显示了触摸事件和操作事件之间的关系。



## 触摸和操作事件

下列内容描述了上图中所示的触摸事件和操作事件之间的关系。

- 当第一个触摸设备在 `UIElement` 上生成 `TouchDown` 事件时，操作逻辑将调用 `CaptureTouch` 方法，该方法会生成 `GotTouchCapture` 事件。
- 当发生 `GotTouchCapture` 时，操作逻辑将调用 `Manipulation.AddManipulator` 方法，该方法会生成 `ManipulationStarting` 事件。
- 当发生 `TouchMove` 事件时，操作逻辑将生成在 `ManipulationInertiaStarting` 事件之前发生的 `ManipulationDelta` 事件。
- 当元素上的最后一个触摸设备引发 `TouchUp` 事件时，操作逻辑将生成 `ManipulationInertiaStarting` 事件。

## 侧重点

在 WPF 中，有两个与焦点有关的主要概念：键盘焦点和逻辑焦点。

## 键盘焦点

键盘焦点指当前正在接收键盘输入的元素。在整个桌面上，只能有一个具有键盘焦点的元素。在 WPF 中，具有键盘焦点的元素会将 `IsKeyboardFocused` 设置为 `true`。静态 `Keyboard` 方法 `FocusedElement` 返回当前具有键盘焦点的元素。

可以通过 Tab 键移到某个元素或通过在特定元素（如 `TextBox`）上单击鼠标来获取键盘焦点。也可以使用 `Keyboard` 类的 `Focus` 方法以编程方式获取键盘焦点。`Focus` 尝试为指定元素提供键盘焦点。`Focus` 返回的元素是当前具有键盘焦点的元素。

为使元素获得键盘焦点，必须将 `Focusable` 属性和 `IsVisible` 属性设置为 `true`。某些类（例如 `Panel`）默认将 `Focusable` 设置为 `false`；因此，如果希望该元素能够获得焦点，必须将此属性设置为 `true`。

以下示例使用 `Focus` 将键盘焦点设置在 `Button` 上。`Loaded` 事件处理程序是在应用程序中设置初始焦点的推荐位置。

C#

```
private void OnLoaded(object sender, RoutedEventArgs e)
{
    // Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton);
}
```

有关键盘焦点的详细信息，请参阅[焦点概述](#)。

## 逻辑焦点

逻辑焦点是指焦点范围内的 `FocusManager.FocusedElement`。一个应用程序中可以有多个具有逻辑焦点的元素，但在一个特定的焦点范围内只能有一个具有逻辑焦点的元素。

焦点范围是一个容器元素，用于跟踪其范围内的 `FocusedElement`。焦点离开焦点范围时，焦点元素会失去键盘焦点，但保留逻辑焦点。焦点返回到焦点范围时，焦点元素会再次获得键盘焦点。这使得键盘焦点可在多个焦点范围之间切换，但确保了焦点返回到焦点范围时，焦点范围中的焦点元素仍为焦点元素。

通过将 `FocusManager` 附加属性 `IsFocusScope` 设置为 `true`，或者通过在代码中使用 `SetIsFocusScope` 方法设置该附加属性，可将元素转换为 Extensible Application Markup Language (XAML) 中的焦点范围。

以下示例通过设置 `IsFocusScope` 附加属性将 `StackPanel` 转换为焦点范围。

XAML

```
<StackPanel Name="focusScope1"
            FocusManager.IsFocusScope="True"
            Height="200" Width="200">
    <Button Name="button1" Height="50" Width="50"/>
    <Button Name="button2" Height="50" Width="50"/>
</StackPanel>
```

C#

```
StackPanel focusScope2 = new StackPanel();
FocusManager.SetIsFocusScope(focusScope2, true);
```

WPF 中默认为焦点范围的类是 [Window](#)、[Menu](#)、[ToolBar](#) 和 [ContextMenu](#)。

具有键盘焦点的元素还具有它所属的焦点范围的逻辑焦点；因此，在 [Keyboard](#) 类或基元素类上使用 [Focus](#) 方法将焦点设置在一个元素上将尝试赋予该元素键盘焦点和逻辑焦点。

要确定焦点范围中的焦点元素，请使用 [GetFocusedElement](#)。 若要更改焦点范围的焦点元素，请使用 [SetFocusedElement](#)。

有关逻辑焦点的详细信息，请参阅[焦点概述](#)。

## 鼠标位置

WPF 输入 API 提供了与坐标空间有关的有用信息。例如，坐标  $(0,0)$  为左上角坐标，但该坐标是树中那一个元素的左上角坐标？是属于输入目标的元素？是在其上附加事件处理程序的元素？还是其他内容？为了避免混淆，WPF 输入 API 要求，在处理通过鼠标获取的坐标时，应指定参考框架。[GetPosition](#) 方法返回鼠标指针相对于指定元素的坐标。

## 鼠标捕获

鼠标设备专门保留称为鼠标捕获的模式特征。鼠标捕获用于在拖放操作开始时保持转换的输入状态，从而不一定发生涉及鼠标指针的标称屏幕位置的其他操作。拖动过程中，未终止拖放时用户无法单击，这使得大多数鼠标悬停提示在拖动来源拥有鼠标捕获时是不合适的。输入系统公开了可确定鼠标捕获状态的 API 以及可强制在特定元素上捕获鼠标或清除鼠标捕获状态的 API。有关拖放操作的详细信息，请参阅[拖放概述](#)。

## 命令

使用命令，输入处理可以更多地在语义级别（而不是在设备输入级别）进行。命令是简单的指令，如 [Cut](#)、[Copy](#)、[Paste](#) 或 [Open](#)。命令可用于集中命令逻辑。可从 [Menu](#)、在 [ToolBar](#) 上或者通过键盘快捷方式使用相同的命令。命令还提供一种机制，用于在命令不可用时禁用控件。

[RoutedCommand](#) 是  [ICommand](#) 的 WPF 实现。当执行 [RoutedCommand](#) 时，将在命令目标上引发 [PreviewExecuted](#) 和 [Executed](#) 事件，这会像其他输入一样，在元素树中发生隧道操作和浮升操作。如果未设置命令目标，则具有键盘焦点的元素将成为命令目标。执行命令的逻辑将附加到 [CommandBinding](#)。当 [Executed](#) 事件到达该特定命令的 [CommandBinding](#) 时，将调用 [CommandBinding](#) 上的 [ExecutedRoutedEventHandler](#)。此处理程序执行该命令的操作。

有关命令的详细信息，请参阅[命令概述](#)。

WPF 提供由 [ApplicationCommands](#)、[MediaCommands](#)、[ComponentCommands](#)、[NavigationCommands](#) 和 [EditingCommands](#) 组成的常用命令库，你也可以定义自己的命令。

以下示例显示了如何设置 [MenuItem](#)，以便在单击时它将调用 [TextBox](#) 上的 [Paste](#) 命令，假定 [TextBox](#) 具有键盘焦点。

XAML

```
<StackPanel>
  <Menu>
    <MenuItem Command="ApplicationCommands.Paste" />
  </Menu>
  <TextBox />
</StackPanel>
```

C#

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

有关 WPF 中的命令的详细信息，请参阅[命令概述](#)。

## 输入系统和基元素

输入事件（例如由 [Mouse](#)、[Keyboard](#) 和 [Stylus](#) 类定义的附加事件）由输入系统引发，并基于在运行时命中测试可视化树来注入到对象模型中的某个特定位置。

[Mouse](#)、[Keyboard](#) 和 [Stylus](#) 定义为附加事件的每个事件也会由基元素类 [UIElement](#) 和 [ContentElement](#) 重新公开为新路由事件。基元素路由事件由处理原始附加事件并重用事件数据的类生成。

当输入事件通过其基元素输入事件实现与特定源元素相关联时，可以通过基于逻辑和可视化树对象的组合的事件路由的其余部分进行路由，并由应用程序代码进行处理。通常，使用 [UIElement](#) 和 [ContentElement](#) 上的路由事件处理这些与设备有关的输入事件更为方便，因为可以使用 XAML 中和代码中更直观的事件处理程序语法。你可以选择处理发起进程的附加事件，但将会面临几个问题：附加事件可能会被基元素类处理标记为已处理，并且你需要使用访问器方法（而不是真正的事件语法）才能为附加事件附加处理程序。

## 后续步骤

现在有多种方法来处理 WPF 中的输入。你还应该对 WPF 使用的各种类型的输入事件和路由事件机制有进一步的了解。

也可以获取更详细说明 WPF 框架元素和事件路由的详细资源。有关详细信息，请参阅以下概述：[命令概述](#)、[焦点概述](#)、[基元素概述](#)、[WPF 中的树](#)和[路由事件概述](#)。

## 另请参阅

- [焦点概述](#)
- [命令概述](#)
- [路由事件概述](#)
- [基元素概述](#)
- [属性](#)

# 命令概述

项目 · 2022/09/27

命令是 Windows Presentation Foundation (WPF) 中的一种输入机制，与设备输入相比，它提供的输入处理更侧重于语义级别。示例命令如许多应用程序均具有的“复制”、“剪切”和“粘贴”操作。

本概述定义 WPF 中有哪些命令、哪些类属于命令模型以及如何在应用程序中使用和创建命令。

本主题包含以下各节：

- [什么是命令？](#)
- [WPF 中的简单命令示例](#)
- [WPF 命令中的四个主要概念](#)
- [命令库](#)
- [创建自定义命令](#)

## 什么是命令

命令具有多个用途。第一个用途是分隔语义和从执行命令的逻辑调用命令的对象。这可使多个不同的源调用同一命令逻辑，并且可针对不同目标自定义命令逻辑。例如，许多应用程序中均有的编辑操作“复制”、“剪切”和“粘贴”若通过使用命令来实现，那么可通过使用不同的用户操作来调用它们。应用程序可允许用户通过单击按钮、选择菜单中的项或使用组合键（例如 Ctrl+X）来剪切所选对象或文本。通过使用命令，可将每种类型的用户操作绑定到相同逻辑。

命令的另一用途是指示操作是否可用。继续以剪切对象或文本为例，此操作只有在选择了内容时才会发生作用。如果用户在未选择任何内容的情况下尝试剪切对象或文本，则不会发生任何操作。为了向用户指示这一点，许多应用程序通过禁用按钮和菜单项来告知用户是否可以执行某操作。命令可以通过实现 [CanExecute](#) 方法来指示操作是否可行。按钮可以订阅 [CanExecuteChanged](#) 事件，如果 [CanExecute](#) 返回 `false` 则禁用，如果 [CanExecute](#) 返回 `true` 则启用。

虽然命令的语义在应用程序和类之间可保持一致，但操作的逻辑特定于操作所针对的特定对象。组合键 Ctrl+X 调用文本类、图像类和 Web 浏览器中的“剪切”命令，但执行“剪切”操作的实际逻辑由执行剪切的应用程序定义。[RoutedCommand](#) 使客户端实现逻辑。文

本对象可将所选文本剪切到剪贴板，而图像对象则剪切所选图像。 应用程序处理 [Executed](#) 事件时可访问命令的目标，并根据目标的类型采取相应操作。

## WPF 中的简单命令示例

使用 WPF 中命令的最简单的方式是使用某一个命令库类中预定义的 [RoutedCommand](#) 使用具有命令处理本机支持的控件，以及使用具有命令调用本机支持的控件。[Paste](#) 命令是 [ApplicationCommands](#) 类中的预定义命令之一。[TextBox](#) 控件含有用于处理 [Paste](#) 命令的内置逻辑。[MenuItem](#) 类具有调用命令的本机支持。

以下示例显示了如何设置 [MenuItem](#)，以便在单击时它将调用 [TextBox](#) 上的 [Paste](#) 命令，假定 [TextBox](#) 具有键盘焦点。

XAML

```
<StackPanel>
  <Menu>
    <MenuItem Command="ApplicationCommands.Paste" />
  </Menu>
  <TextBox />
</StackPanel>
```

C#

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

## WPF 命令中的四个主要概念

WPF 中的路由命令模型可分解为四个主要概念：命令、命令源、命令目标和命令绑定：

- 命令是要执行的操作。

- 命令源是调用命令的对象。
- 命令目标是在其上执行命令的对象。
- 命令绑定是将命令逻辑映射到命令的对象。

在前面的示例中，`Paste` 命令是命令，`MenuItem` 是命令源，`TextBox` 是命令目标，命令绑定由 `TextBox` 控件提供。值得注意的是，`CommandBinding` 并不总是由作为命令目标类的控件提供。通常，`CommandBinding` 必须由应用程序开发者创建，否则 `CommandBinding` 可能会附加到命令目标的上级元素。

## 命令

WPF 中的命令是通过实现 `ICommand` 接口创建的。`ICommand` 公开了两种方法 `Execute` 和 `CanExecute`，以及一个事件 `CanExecuteChanged`。`Execute` 执行与该命令关联的操作。`CanExecute` 确定是否可以在当前命令目标上执行该命令。如果集中管理命令操作的命令管理器检测到命令源中存在一个可能使已引发命令无效但尚未由命令绑定执行的更改，则会引发 `CanExecuteChanged`。`ICommand` 的 WPF 实现是 `RoutedCommand` 类，并且是本概述的重点。

WPF 中输入的主要源是鼠标、键盘、墨迹和路由命令。面向设备程度更高的输入使用 `RoutedEventArgs` 通知应用程序页中的对象输入事件已发生。`RoutedCommand` 也不例外。`RoutedCommand` 的 `Execute` 和 `CanExecute` 方法不包含该命令的应用程序逻辑，而是引发通过元素树通行和浮升的路由事件，直到遇到具有 `CommandBinding` 的对象。`CommandBinding` 包含这些事件的处理程序，命令正是由这些处理程序执行。有关 WPF 中事件路由的详细信息，请参阅[路由事件概述](#)。

`RoutedCommand` 上的 `Execute` 方法引发命令目标上的 `PreviewExecuted` 和 `Executed` 事件。`RoutedCommand` 上的 `CanExecute` 方法引发命令目标上的 `CanExecute` 和 `PreviewCanExecute` 事件。这些事件通过元素树通行和浮升，直到遇到一个具有针对该特定命令的 `CommandBinding` 的对象。

WPF 提供了分布在几个类中的一组常用路由命令：`MediaCommands`、`ApplicationCommands`、`NavigationCommands`、`ComponentCommands` 和 `EditingCommands`。这些类仅由 `RoutedCommand` 对象构成，而不包含命令的实现逻辑。实现逻辑由在其上执行命令的对象负责。

## 命令源

命令源是调用命令的对象。命令源的示例有 `MenuItem`、`Button` 和 `KeyGesture`。

WPF 中的命令源通常实现 `ICommandSource` 接口。

`ICommandSource` 公开三个属性：`Command`、`CommandTarget` 和 `CommandParameter`：

- `Command` 是在调用命令源时执行的命令。
- `CommandTarget` 是要执行命令的对象。值得注意的是，在 WPF 中，仅当  `ICommand` 为 `RoutedCommand` 时，`ICommandSource` 上的 `CommandTarget` 属性才适用。如果在 `ICommandSource` 上设置 `CommandTarget` 并且相应的命令不是 `RoutedCommand`，则忽略命令目标。如果未设置 `CommandTarget`，则具有键盘焦点的元素将成为命令目标。
- `CommandParameter` 是用于将信息传递给实现命令的处理程序的用户定义数据类型。

实现 `ICommandSource` 的 WPF 类是 `ButtonBase`、`MenuItem`、`Hyperlink` 和 `InputBinding`。单击 `ButtonBase`、`MenuItem` 和 `Hyperlink` 时，调用一个命令，当执行与其关联的 `InputGesture` 时，`InputBinding` 调用命令。

以下示例显示如何将 `ContextMenu` 中的 `MenuItem` 用作 `Properties` 命令的命令源。

XAML

```
<StackPanel>
  <StackPanel.ContextMenu>
    <ContextMenu>
      <MenuItem Command="ApplicationCommands.Properties" />
    </ContextMenu>
  </StackPanel.ContextMenu>
</StackPanel>
```

C#

```
StackPanel cmdSourcePanel = new StackPanel();
ContextMenu cmdSourceContextMenu = new ContextMenu();
MenuItem cmdSourceMenuItem = new MenuItem();

// Add ContextMenu to the StackPanel.
cmdSourcePanel.ContextMenu = cmdSourceContextMenu;
cmdSourcePanel.ContextMenu.Items.Add(cmdSourceMenuItem);

// Associate Command with MenuItem.
cmdSourceMenuItem.Command = ApplicationCommands.Properties;
```

通常，命令源将侦听 `CanExecuteChanged` 事件。此事件通知命令源在当前命令目标上执行命令的能力可能已发生更改。命令源可以使用 `CanExecute` 方法查询 `RoutedCommand` 的当前状态。如果命令无法执行，命令源可禁用自身。此情况的一个示例是 `MenuItem`，在命令无法执行时，它自身将灰显。

`InputGesture` 可以用作命令源。 WPF 中的两种输入笔势是 `KeyGesture` 和 `MouseGesture`。可以将 `KeyGesture` 视为键盘快捷方式，例如 `Ctrl+C`。`KeyGesture` 由一个 `Key` 和一组 `ModifierKeys` 组成。`MouseGesture` 由 `MouseAction` 和一组可选的 `ModifierKeys` 组成。

为了将 `InputGesture` 用作命令源，它必须与一个命令相关联。可通过几种方式来实现此目的。其中一种方法是使用 `InputBinding`。

以下示例演示如何在 `KeyGesture` 和 `RoutedCommand` 之间创建 `KeyBinding`。

XAML

```
<Window.InputBindings>
  <KeyBinding Key="B"
    Modifiers="Control"
    Command="ApplicationCommands.Open" />
</Window.InputBindings>
```

C#

```
KeyGesture OpenKeyGesture = new KeyGesture(
  Key.B,
  ModifierKeys.Control);

KeyBinding OpenCmdKeybinding = new KeyBinding(
  ApplicationCommands.Open,
  OpenKeyGesture);

this.InputBindings.Add(OpenCmdKeybinding);
```

将 `InputGesture` 关联到 `RoutedCommand` 的另一种方法是将 `InputGesture` 添加到 `RoutedCommand` 上的 `InputGestureCollection`。

以下示例演示如何将 `KeyGesture` 添加到 `RoutedCommand` 的 `InputGestureCollection` 中。

C#

```
KeyGesture OpenCmdKeyGesture = new KeyGesture(
  Key.B,
  ModifierKeys.Control);

ApplicationCommands.Open.InputGestures.Add(OpenCmdKeyGesture);
```

## CommandBinding

[CommandBinding](#) 将命令与实现该命令的事件处理程序相关联。

[CommandBinding](#) 类包含 [Command](#) 属性，及 [PreviewExecuted](#)、[Executed](#)、[PreviewCanExecute](#) 和 [CanExecute](#) 事件。

[Command](#) 是与 [CommandBinding](#) 关联的命令。附加到 [PreviewExecuted](#) 和 [Executed](#) 事件的事件处理程序实现命令逻辑。附加到 [PreviewCanExecute](#) 和 [CanExecute](#) 事件的事件处理程序确定是否可以在当前命令目标上执行该命令。

以下示例演示如何在应用程序的根 [Window](#) 上创建 [CommandBinding](#)。

[CommandBinding](#) 将 [Open](#) 命令与 [Executed](#) 和 [CanExecute](#) 处理程序关联。

XAML

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
        Executed="OpenCmdExecuted"
        CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>
```

C#

```
// Creating CommandBinding and attaching an Executed and CanExecute handler
CommandBinding OpenCmdBinding = new CommandBinding(
    ApplicationCommands.Open,
    OpenCmdExecuted,
    OpenCmdCanExecute);

this.CommandBindings.Add(OpenCmdBinding);
```

接下来，创建了 [ExecutedRoutedEventHandler](#) 和 [CanExecuteRoutedEventHandler](#)。

[ExecutedRoutedEventHandler](#) 打开了显示字符串的 [MessageBox](#)，该字符串表示已执行此命令。[CanExecuteRoutedEventHandler](#) 将 [CanExecute](#) 属性设置为 `true`。

C#

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    String command, targetobj;
    command = ((RoutedCommand)e.Command).Name;
    targetobj = ((FrameworkElement)target).Name;
    MessageBox.Show("The " + command + " command has been invoked on target
object " + targetobj);
}
```

C#

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

将 `CommandBinding` 附加到特定对象，例如应用程序或控件的根 `Window`。

`CommandBinding` 附加到的对象定义了绑定的范围。例如，附加到命令目标的上级元素的 `CommandBinding` 可以通过 `Executed` 事件到达，但无法到达附加到命令目标的下级元素的 `CommandBinding`。其直接原因在于 `RoutedEvent` 从引发事件的对象通行和浮升的方式。

在某些情况下，`CommandBinding` 会附加到命令目标本身，例如 `TextBox` 类及 `Cut`、`Copy` 和 `Paste` 命令。然而，很多时候将 `CommandBinding` 附加到命令目标的上级元素（例如主要 `Window` 或应用程序对象）会更加方便，尤其是在同一 `CommandBinding` 可用于多个命令目标时。这是在创建命令基础结构时需要考虑的设计决策。

## 命令目标

命令目标是在其上执行命令的元素。关于 `RoutedCommand`，命令目标是 `Executed` 和 `CanExecute` 的路由开始的元素。如前所述，在 WPF 中，仅当 `ICommand` 为 `RoutedCommand` 时，`ICommandSource` 上的 `CommandTarget` 属性才适用。如果在 `ICommandSource` 上设置 `CommandTarget` 并且相应的命令不是 `RoutedCommand`，则忽略命令目标。

命令源可以显式设置命令目标。如果未定义命令目标，则具有键盘焦点的元素将用作命令目标。将具有键盘焦点的元素用作命令目标的一个好处在于，这样可使应用程序开发者能够使用同一命令源在多个目标上调用命令，而无需跟踪命令目标。例如，如果 `MenuItem` 在具有 `TextBox` 控件和 `PasswordBox` 控件的应用程序中调用“Paste”命令，则目标可以是 `TextBox` 或 `PasswordBox`，具体取决于哪个控件具有键盘焦点。

以下示例演示如何在标记和代码隐藏中显式设置命令目标。

XAML

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste"
                  CommandTarget="{Binding ElementName=mainTextBox}" />
    </Menu>
    <TextBox Name="mainTextBox" />
</StackPanel>
```

C#

```
// Creating the UI objects
StackPanel mainStackPanel = new StackPanel();
TextBox pasteTextBox = new TextBox();
Menu stackPanelMenu = new Menu();
MenuItem pasteMenuItem = new MenuItem();

// Adding objects to the panel and the menu
stackPanelMenu.Items.Add(pasteMenuItem);
mainStackPanel.Children.Add(stackPanelMenu);
mainStackPanel.Children.Add(pasteTextBox);

// Setting the command to the Paste command
pasteMenuItem.Command = ApplicationCommands.Paste;

// Setting the command target to the TextBox
pasteMenuItem.CommandTarget = pasteTextBox;
```

## CommandManager

[CommandManager](#) 提供许多与命令相关的函数。它提供了一组静态方法，用于在特定元素中添加和删除 [PreviewExecuted](#)、[Executed](#)、[PreviewCanExecute](#) 和 [CanExecute](#) 事件处理程序。它提供了将 [CommandBinding](#) 和 [InputBinding](#) 对象注册到特定类的方法。[CommandManager](#) 还通过 [RequerySuggested](#) 事件提供了一种方法，用于在应引发 [CanExecuteChanged](#) 事件时通知命令。

[InvalidateRequerySuggested](#) 方法强制 [CommandManager](#) 引发 [RequerySuggested](#) 事件。这在应禁用/启用命令的情况下非常有用，但对于 [CommandManager](#) 可识别的情况，则不太有用。

## 命令库

WPF 提供一组预定义命令。命令库包括以下类：[ApplicationCommands](#)、[NavigationCommands](#)、[MediaCommands](#)、[EditingCommands](#) 和 [ComponentCommands](#)。这些类提供诸如 [Cut](#)、[BrowseBack](#)、[BrowseForward](#)、[Play](#)、[Stop](#) 和 [Pause](#) 的命令。

许多这些命令都包含一组默认输入绑定。例如，如果指定应用程序处理复制命令，则可自动获取键盘绑定“CTRL+C”。此外，还可获得其他输入设备的绑定，例如 Tablet PC 笔势和语音信息。

使用 WPF 引用各种命令库中的命令时，WPF XAML 处理器在加载时执行引用。

## 创建自定义命令

如果命令库类中的命令不能满足需要，你可以创建自己的命令。 可通过两种方式创建自定义命令。 第一种方式是从头开始并实现  [ICommand](#) 接口。 另一种更常见的方法是创建  [RoutedCommand](#) 或  [RoutedUICommand](#)。

有关创建自定义  [RoutedCommand](#) 的示例，请参阅 [创建自定义 RoutedCommand 示例](#)。

## 请参阅

- [RoutedCommand](#)
- [CommandBinding](#)
- [InputBinding](#)
- [CommandManager](#)
- [输入概述](#)
- [路由事件概述](#)
- [实现 ICommandSource](#)
- [如何：将命令添加到菜单项](#)
- [创建自定义 RoutedCommand 示例](#)

# 焦点概述

项目 · 2023/02/06

在 WPF 中，有两个与焦点有关的主要概念：键盘焦点和逻辑焦点。键盘焦点指接收键盘输入的元素，而逻辑焦点指焦点范围中具有焦点的元素。本概述详细介绍了这些概念。对于创建具有多个可获取焦点的区域的复杂应用程序来说，理解这些概念之间的区别非常重要。

参与焦点管理的主要类是 [Keyboard](#) 类、[FocusManager](#) 类和基本元素类，例如 [UIElement](#) 和 [ContentElement](#)。有关基元素的详细信息，请参阅[基元素概述](#)。

[Keyboard](#) 类主要与键盘焦点相关，而 [FocusManager](#) 主要与逻辑焦点相关，但这种区别不是绝对的。具有键盘焦点的元素也具有逻辑焦点，但具有逻辑焦点的元素不一定具有键盘焦点。使用 [Keyboard](#) 类来设置具有键盘焦点的元素时，这一点就很明显，因为它还在元素上设置逻辑焦点。

## 键盘焦点

键盘焦点指当前正在接收键盘输入的元素。在整个桌面上，只能有一个具有键盘焦点的元素。在 WPF 中，具有键盘焦点的元素会将 [IsKeyboardFocused](#) 设置为 `true`。  
[Keyboard](#) 类的静态属性 [FocusedElement](#) 获取当前具有键盘焦点的元素。

为使元素获得键盘焦点，必须将基元素的 [Focusable](#) 和 [isVisible](#) 属性设置为 `true`。某些类（例如 [Panel](#) 基类）默认将 [Focusable](#) 设置为 `false`；因此，如果要此类元素能够获得键盘焦点，必须将 [Focusable](#) 设置为 `true`。

可通过用户与 UI 交互（例如，按 Tab 键导航到某个元素或者在某些元素上单击鼠标）来获取键盘焦点。也可以使用 [Keyboard](#) 类的 [Focus](#) 方法以编程方式获取键盘焦点。[Focus](#) 方法尝试为指定元素提供键盘焦点。返回的元素是具有键盘焦点的元素，如果旧的或新的焦点对象阻止请求，则具有键盘焦点的元素可能不是请求的元素。

以下示例使用 [Focus](#) 方法将键盘焦点设置在 [Button](#) 上。

C#

```
private void OnLoaded(object sender, RoutedEventArgs e)
{
    // Sets keyboard focus on the first Button in the sample.
    Keyboard.Focus(firstButton);
}
```

基元素类的 `IsKeyboardFocused` 属性获取一个值，该值指示元素是否具有键盘焦点。基元素类的 `IsKeyboardFocusWithin` 属性获取一个值，该值指示元素或其任何一个视觉子元素是否具有键盘焦点。

如果在应用程序启动时设置初始焦点，接收焦点的元素必须位于应用程序加载的初始窗口的可视化树中，并且该元素必须将 `Focusable` 和 `Visible` 设置为 `true`。`Loaded` 事件处理程序是设置初始焦点的推荐位置。还可以通过调用 `Invoke` 或 `BeginInvoke` 来使用 `Dispatcher` 回叫。

## 逻辑焦点

逻辑焦点是指焦点范围内的 `FocusManager.FocusedElement`。焦点范围是一个元素，用于跟踪其范围内的 `FocusedElement`。键盘焦点离开焦点范围时，焦点元素会失去键盘焦点，但保留逻辑焦点。键盘焦点返回到焦点范围时，焦点元素会再次获得键盘焦点。这使得键盘焦点可在多个焦点范围之间切换，但确保了焦点返回到焦点范围时，焦点范围中的焦点元素重新获得键盘焦点。

一个应用程序中可以有多个具有逻辑焦点的元素，但在一个特定的焦点范围内只能有一个具有逻辑焦点的元素。

具有键盘焦点的元素还具有其所属焦点范围的逻辑焦点。

通过将 `FocusManager` 附加属性 `IsFocusScope` 设置为 `true`，可将元素转换为 Extensible Application Markup Language (XAML) 中的焦点范围。在代码中，可通过调用 `SetIsFocusScope` 将元素转换为焦点范围。

以下示例通过设置 `IsFocusScope` 附加属性将 `StackPanel` 转换为焦点范围。

XAML

```
<StackPanel Name="focusScope1"
    FocusManager.IsFocusScope="True"
    Height="200" Width="200">
    <Button Name="button1" Height="50" Width="50"/>
    <Button Name="button2" Height="50" Width="50"/>
</StackPanel>
```

C#

```
StackPanel focusScope2 = new StackPanel();
FocusManager.SetIsFocusScope(focusScope2, true);
```

`GetFocusScope` 返回指定元素的焦点范围。

WPF 中默认为焦点范围的类是 [Window](#)、[MenuItem](#)、[ToolBar](#) 和 [ContextMenu](#)。

[GetFocusedElement](#) 获取指定焦点范围的焦点元素。 [SetFocusedElement](#) 设置指定焦点范围中的焦点元素。[SetFocusedElement](#) 通常用于设置初始焦点元素。

以下示例在焦点范围内设置焦点元素并获取焦点范围的焦点元素。

C#

```
// Sets the focused element in focusScope1  
// focusScope1 is a StackPanel.  
FocusManager.SetFocusedElement(focusScope1, button2);  
  
// Gets the focused element for focusScope 1  
IInputElement focusedElement = FocusManager.GetFocusedElement(focusScope1);
```

## 键盘导航

按下导航键之一时，[KeyboardNavigation](#) 类负责实现默认键盘焦点导航。 导航键包括：[Tab](#)、[Shift+Tab](#)、[Ctrl+Tab](#)、[Ctrl+Shift+Tab](#)、向上键、向下键、向左键和向右键。

可以通过设置附加的 [KeyboardNavigation](#) 属性 [TabNavigation](#)、[ControlTabNavigation](#) 和 [DirectionalNavigation](#) 来更改导航容器的导航行为。 这些属性的类型为 [KeyboardNavigationMode](#)，可能的值为 [Continue](#)、[Local](#)、[Contained](#)、[Cycle](#)、[Once](#) 和 [None](#)。 默认值为 [Continue](#)，表示该元素不是导航容器。

以下示例创建具有多个 [MenuItem](#) 对象的 [Menu](#)。[TabNavigation](#) 附加属性在 [Menu](#) 上设置为 [Cycle](#)。 使用 [Tab](#) 键在 [Menu](#) 中改变焦点时，焦点会从每个元素上移过，到达最后一个元素后会返回第一个元素。

XAML

```
<Menu KeyboardNavigation.TabNavigation="Cycle">  
  <MenuItem Header="Menu Item 1" />  
  <MenuItem Header="Menu Item 2" />  
  <MenuItem Header="Menu Item 3" />  
  <MenuItem Header="Menu Item 4" />  
</Menu>
```

C#

```
Menu navigationMenu = new Menu();  
MenuItem item1 = new MenuItem();  
MenuItem item2 = new MenuItem();  
MenuItem item3 = new MenuItem();  
MenuItem item4 = new MenuItem();
```

```
navigationMenu.Items.Add(item1);
navigationMenu.Items.Add(item2);
navigationMenu.Items.Add(item3);
navigationMenu.Items.Add(item4);

KeyboardNavigation.SetTabNavigation(navigationMenu,
    KeyboardNavigationMode.Cycle);
```

## 以编程方式导航焦点

与焦点结合使用的其他 API 是 [MoveFocus](#) 和 [PredictFocus](#)。

[MoveFocus](#) 将焦点转移到应用程序中的下一个元素。 [TraversalRequest](#) 用于指定方向。传递给 [MoveFocus](#) 的 [FocusNavigationDirection](#) 指定焦点可移动的各个方向，例如 [First](#)、[Last](#)、[Up](#) 和 [Down](#)。

以下示例使用 [MoveFocus](#) 更改焦点元素。

C#

```
// Creating a FocusNavigationDirection object and setting it to a
// local field that contains the direction selected.
FocusNavigationDirection focusDirection = _focusMoveValue;

// MoveFocus takes a TraversalRequest as its argument.
TraversalRequest request = new TraversalRequest(focusDirection);

// Gets the element with keyboard focus.
UIElement elementWithFocus = Keyboard.FocusedElement as UIElement;

// Change keyboard focus.
if (elementWithFocus != null)
{
    elementWithFocus.MoveFocus(request);
}
```

[PredictFocus](#) 返回在焦点更改时接收焦点的对象。 目前，[PredictFocus](#) 仅支持 [Up](#)、[Down](#)、[Left](#) 和 [Right](#)。

## 焦点事件

与键盘焦点相关的事件是 [PreviewGotKeyboardFocus](#)、[GotKeyboardFocus](#) 和 [PreviewLostKeyboardFocus](#)、[LostKeyboardFocus](#)。 这些事件定义为 [Keyboard](#) 类上的附加事件，但更多地作为基元素类上的等效路由事件来访问。 有关事件的详细信息，请参阅[路由事件概述](#)。

元素获得键盘焦点时会引发 [GotKeyboardFocus](#)。 元素丢失键盘焦点时会引发 [LostKeyboardFocus](#)。 如果已处理 [PreviewGotKeyboardFocus](#) 事件或 [PreviewLostKeyboardFocusEvent](#) 事件并将 [Handled](#) 设置为 `true`，则焦点不会改变。

以下示例将 [GotKeyboardFocus](#) 和 [LostKeyboardFocus](#) 事件处理程序附加到 [TextBox](#)。

XAML

```
<Border BorderBrush="Black" BorderThickness="1"
        Width="200" Height="100" Margin="5">
    <StackPanel>
        <Label HorizontalAlignment="Center" Content="Type Text In This TextBox"
    />
        <TextBox Width="175"
                Height="50"
                Margin="5"
                TextWrapping="Wrap"
                HorizontalAlignment="Center"
                VerticalScrollBarVisibility="Auto"
                GotKeyboardFocus="TextBoxGotKeyboardFocus"
                LostKeyboardFocus="TextBoxLostKeyboardFocus"
                KeyDown="SourceTextKeyDown"/>
    </StackPanel>
</Border>
```

[TextBox](#) 获得键盘焦点时，[TextBox](#) 的 [Background](#) 属性更改为 [LightBlue](#)。

C#

```
private void TextBoxGotKeyboardFocus(object sender,
KeyboardFocusChangedEventArgs e)
{
    TextBox source = e.Source as TextBox;

    if (source != null)
    {
        // Change the TextBox color when it obtains focus.
        source.Background = Brushes.LightBlue;

        // Clear the TextBox.
        source.Clear();
    }
}
```

[TextBox](#) 丢失键盘焦点时，[TextBox](#) 的 [Background](#) 属性变回白色。

C#

```
private void TextBoxLostKeyboardFocus(object sender,
KeyboardFocusChangedEventArgs e)
```

```
{  
    TextBox source = e.Source as TextBox;  
  
    if (source != null)  
    {  
        // Change the TextBox color when it loses focus.  
        source.Background = Brushes.White;  
  
        // Set the hit counter back to zero and updates the display.  
        this.ResetCounter();  
    }  
}
```

与逻辑焦点相关的事件是 [GotFocus](#) 和 [LostFocus](#)。这些事件在 [FocusManager](#) 上定义为附加事件，但 [FocusManager](#) 不公开 CLR 事件包装器。[UIElement](#) 和 [ContentElement](#) 可以更方便地公开这些事件。

## 请参阅

- [FocusManager](#)
- [UIElement](#)
- [ContentElement](#)
- [输入概述](#)
- [基元素概述](#)

# 为控件中的焦点设置样式以及 FocusVisualStyle

项目 · 2023/02/06

Windows Presentation Foundation (WPF) 提供两种用于在控件接收键盘焦点时更改其视觉外观的并行机制。第一种机制是对应用于控件的样式或模板中的属性（如 `IsKeyboardFocused`）使用属性 setter。第二种机制是将一个单独的样式作为 `FocusVisualStyle` 属性的值提供；“焦点视觉样式”为绘制于控件顶部的装饰器创建一个单独的可视化树，而不是通过替换来更改控件或其他 UI 元素的可视化树。本主题讨论上述每一种机制的适用情况。

## 焦点视觉样式的用途

焦点视觉样式功能提供一种通用“对象模型”，用于基于任何 UI 元素的键盘导航来引入视觉用户反馈。即使未向控件应用新模板，或者不知道具体的模板组合，这也是可能的。

但是，正因为焦点视觉样式功能可以在不知道控件模板的情况下工作，所以必须限制可针对使用焦点视觉样式的控件显示的视觉反馈。此功能实际执行的操作是在控件通过模板进行呈现来创建可视化树时在其上覆盖另一可视化树（装饰器）。使用一个填写 `FocusVisualStyle` 属性的样式来定义这一单独的可视化树。

## 默认焦点视觉样式行为

焦点视觉样式仅在焦点操作由键盘启动时才起作用。任何鼠标操作或者通过编程实现的焦点更改都会禁用焦点视觉样式模式。有关焦点模式间区别的详细信息，请参阅[焦点概述](#)。

控件的主题包括默认焦点视觉样式行为，该焦点视觉样式成为主题中所有控件的焦点视觉样式。该主题样式由静态键 `FocusVisualStyleKey` 的值来标识。当在应用程序级声明自己的焦点视觉样式时，将替换主题中的这一默认样式行为。或者，如果要定义整个主题，那么应同样使用这个键来为整个主题的默认行为定义样式。

在主题中，默认焦点视觉样式通常非常简单。下面是一个近似的焦点视觉样式：

XAML

```
<Style x:Key="{x:Static SystemParameters.FocusVisualStyleKey}">
  <Setter Property="Control.Template">
    <Setter.Value>
      <ControlTemplate>
        <Rectangle StrokeThickness="1"
```

```
        Stroke="Black"
        StrokeDashArray="1 2"
        SnapsToDevicePixels="true"/>
    </ControlTemplate>
</Setter.Value>
</Setter>
</Style>
```

## 何时使用焦点视觉样式

从概念上来说，应用于控件的焦点视觉样式的外观在所有控件上应该是一致的。确保一致性的方法是仅在创作整个主题时才更改焦点视觉样式，这样主题中定义的每个控件都要么获得完全相同的焦点视觉样式，要么获得在视觉上与控件具有相关性的某一样式的变体。或者，可能使用相同的样式（或类似的样式）来设置页面或 UI 中每个可通过键盘获得焦点的元素的样式。

在单个不属于主题一部分的控件样式上设置 [FocusVisualStyle](#) 并非焦点视觉样式的预期用法。这是因为控件间的不一致视觉行为会使用户对键盘焦点产生混乱的感觉。如果打算为键盘焦点设计特定于控件的行为，并且希望这些行为在主题中存在不一致，那么一种更好的方法是在样式中对个别输入状态属性（如 [IsFocused](#) 或 [IsKeyboardFocused](#)）使用触发器。

焦点视觉样式专门作用于键盘焦点。就其本身而言，焦点视觉样式是一种辅助功能。如果希望针对任何类型的焦点来更改 UI（无论是通过鼠标、键盘还是编程方式），那么不应使用焦点视觉样式，而应在样式或模板中使用 [setter](#) 和触发器，它们根据常规焦点属性（如 [IsFocused](#) 或 [IsKeyboardFocusWithin](#)）的值发生作用。

## 如何创建焦点视觉样式

为焦点视觉样式创建的样式应始终具有 [Control](#) 的 [TargetType](#)。样式应主要包含 [ControlTemplate](#)。不必将目标类型指定为将焦点视觉样式分配给 [FocusVisualStyle](#) 的类型。

因为目标类型始终是 [Control](#)，所以必须通过使用所有控件共有的属性（使用 [Control](#) 类及其基类的属性）来设置样式。应创建将正确地用作 UI 元素的覆盖层、且不会隐藏控件的功能区域的模板。通常，这意味着视觉反馈应显示在控件边距之外，或者显示为临时的或不显眼的效果（这样就不会阻止应用焦点视觉样式的控件上的命中测试）。可以在可用于确定覆盖模板的大小和位置的模板绑定中使用的属性包括 [ActualHeight](#)、[ActualWidth](#)、[Margin](#) 和 [Padding](#)。

## 使用焦点视觉样式的替代方法

对于不适合使用焦点视觉样式的情况（因为仅设置单个控件的样式或希望对控件模板有更多的控制），存在许多其他可用的属性和技术，可用来创建响应焦点更改的视觉行为。

[样式设置和模板化](#)中对触发器、setter 以及事件 setter 进行了详细介绍。 [路由事件概述](#) 中对路由事件处理进行了介绍。

## IsKeyboardFocused

如果对键盘焦点有特别兴趣，那么可将 `IsKeyboardFocused` 依赖属性用于属性 `Trigger`。对于定义专用于单个控件且可能与其他控件的键盘焦点行为在视觉上不匹配的键盘焦点行为，一个更适合的方法是使用样式或模板中的属性触发器。

另一个类似的依赖属性是 `IsKeyboardFocusWithin`，如果希望从视觉上显示出键盘焦点在控件组成中的位置或控件功能区域中的位置，那么可能适合使用此属性。例如，可以放置一个 `IsKeyboardFocusWithin` 触发器，以便某个包含若干控件的面板以不同的外观显示，即使将键盘焦点的位置精确到面板中的某个元素上，也可呈现出不同的外观。

还可使用事件 `GotKeyboardFocus` 和 `LostKeyboardFocus`（以及其 Preview 等效项）。可以将这些事件用作 `EventSetter` 的基础，也可以通过代码隐藏的方式为这些事件编写处理程序。

## 其他焦点属性

如果希望导致更改焦点的所有可能原因都产生一个视觉行为，那么应让 setter 或触发器基于 `IsFocused` 依赖属性，或者基于用于 `EventSetter` 的 `GotFocus` 或 `LostFocus` 事件。

## 请参阅

- [FocusVisualStyle](#)
- [样式设置和模板化](#)
- [焦点概述](#)
- [输入概述](#)

# 演练：创建你的第一个触控应用程序

项目 • 2023/02/06

WPF 使应用程序能够响应触控。例如，可以通过在触控敏感型设备（如触摸屏）上使用一根或多根手指与应用程序交互。本演练创建了一个应用程序，使用户能够使用触控来移动、旋转单个对象或重设其大小。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio。
- 一种接受触控输入的设备，例如支持 Windows Touch 的触摸屏。

此外，应对如何在 WPF 中创建应用程序，尤其是如何订阅和处理事件有一个基本的了解。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。

## 创建应用程序

### 创建应用程序

1. 在 Visual Basic 或 Visual C# 中创建名为 `BasicManipulation` 的新 WPF 应用程序项目。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。
2. 将 `MainWindow.xaml` 的内容替换为以下 XAML。

此标记创建一个简单的应用程序，该应用程序在 `Canvas` 上包含一个红色 `Rectangle`。`Rectangle` 的 `IsManipulationEnabled` 属性设置为 `true`，以便接收操作事件。应用程序订阅 `ManipulationStarting`、`ManipulationDelta` 和 `ManipulationInertiaStarting` 事件。这些事件包含用于在用户操作 `Rectangle` 时移动它的逻辑。

XAML

```
<Window x:Class="BasicManipulation.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Move, Size, and Rotate the Square"
        WindowState="Maximized"
        ManipulationStarting="Window_ManipulationStarting"
        ManipulationDelta="Window_ManipulationDelta"
```

```

        ManipulationInertiaStarting="Window_InertiaStarting">
<Window.Resources>

    <!--The movement, rotation, and size of the Rectangle is
        specified by its RenderTransform.-->
    <MatrixTransform x:Key="InitialMatrixTransform">
        <MatrixTransform.Matrix>
            <Matrix OffsetX="200" OffsetY="200"/>
        </MatrixTransform.Matrix>
    </MatrixTransform>

</Window.Resources>

<Canvas>
    <Rectangle Fill="Red" Name="manRect"
        Width="200" Height="200"
        RenderTransform="{StaticResource
InitialMatrixTransform}"
        IsManipulationEnabled="true" />
</Canvas>
</Window>

```

3. 如果使用的是 Visual Basic，则在 MainWindow.xaml 的第一行中，将 `x:Class="BasicManipulation.MainWindow"` 替换为 `x:Class="MainWindow"`。

4. 在 `MainWindow` 类中，添加以下 `ManipulationStarting` 事件处理程序。

当 WPF 检测到触控输入开始操作对象时，将发生 `ManipulationStarting` 事件。代码通过设置 `ManipulationContainer` 属性来指定操作的位置应相对于 `Window`。

C#

```

void Window_ManipulationStarting(object sender,
ManipulationStartingEventArgs e)
{
    e.ManipulationContainer = this;
    e.Handled = true;
}

```

5. 在 `MainWindow` 类中，添加以下 `ManipulationDelta` 事件处理程序。

当触控输入改变位置时，会发生 `ManipulationDelta` 事件，该事件可在操作期间多次发生。该事件也可以在手指抬起后发生。例如，如果用户在屏幕上拖动手指，则 `ManipulationDelta` 事件会在手指移动时发生多次。当用户从屏幕上抬起手指时，`ManipulationDelta` 事件会不断发生以模拟惯性。

该代码将 `DeltaManipulation` 应用于 `Rectangle` 的 `RenderTransform` 以便在用户移动触控输入时移动它。它还会检查当事件在惯性期间发生时，`Rectangle` 是否超出了

Window 的边界。如果是，应用程序会调用 ManipulationDeltaEventArgs.Complete 方法来结束操作。

C#

```
void Window_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    // Get the Rectangle and its RenderTransform matrix.
    Rectangle rectToMove = e.OriginalSource as Rectangle;
    Matrix rectMatrix =
        ((MatrixTransform)rectToMove.RenderTransform).Matrix;

    // Rotate the Rectangle.
    rectMatrix.RotateAt(e.DeltaManipulation.Rotation,
                        e.ManipulationOrigin.X,
                        e.ManipulationOrigin.Y);

    // Resize the Rectangle. Keep it square
    // so use only the X value of Scale.
    rectMatrix.ScaleAt(e.DeltaManipulation.Scale.X,
                       e.DeltaManipulation.Scale.X,
                       e.ManipulationOrigin.X,
                       e.ManipulationOrigin.Y);

    // Move the Rectangle.
    rectMatrix.Translate(e.DeltaManipulation.Translation.X,
                         e.DeltaManipulation.Translation.Y);

    // Apply the changes to the Rectangle.
    rectToMove.RenderTransform = new MatrixTransform(rectMatrix);

    Rect containingRect =
        new
    Rect(((FrameworkElement)e.ManipulationContainer).RenderSize);

    Rect shapeBounds =
        rectToMove.RenderTransform.TransformBounds(
            new Rect(rectToMove.RenderSize));

    // Check if the rectangle is completely in the window.
    // If it is not and inertia is occurring, stop the manipulation.
    if (e.IsInertial && !containingRect.Contains(shapeBounds))
    {
        e.Complete();
    }

    e.Handled = true;
}
```

6. 在 `MainWindow` 类中，添加以下 `ManipulationInertiaStarting` 事件处理程序。

当用户从屏幕上抬起所有手指时，会发生 `ManipulationInertiaStarting` 事件。代码设置矩形移动、扩展和旋转的初始速度和减速度。

C#

```
void Window_InertiaStarting(object sender,
    ManipulationInertiaStartingEventArgs e)
{
    // Decrease the velocity of the Rectangle's movement by
    // 10 inches per second every second.
    // (10 inches * 96 pixels per inch / 1000ms^2)
    e.TranslationBehavior.DesiredDeceleration = 10.0 * 96.0 / (1000.0 *
    1000.0);

    // Decrease the velocity of the Rectangle's resizing by
    // 0.1 inches per second every second.
    // (0.1 inches * 96 pixels per inch / (1000ms^2))
    e.ExpansionBehavior.DesiredDeceleration = 0.1 * 96 / (1000.0 *
    1000.0);

    // Decrease the velocity of the Rectangle's rotation rate by
    // 2 rotations per second every second.
    // (2 * 360 degrees / (1000ms^2))
    e.RotationBehavior.DesiredDeceleration = 720 / (1000.0 * 1000.0);

    e.Handled = true;
}
```

## 7. 生成并运行该项目。

应会在窗口中看到一个红色方块。

# 测试应用程序

若要测试应用程序，请尝试以下操作。请注意，可以同时执行以下多项操作。

- 若要移动 `Rectangle`，请将手指放在 `Rectangle` 上，然后在屏幕上移动手指。
- 若要重设 `Rectangle` 的大小，请将两根手指放在 `Rectangle` 上，然后将手指相互靠拢或分开。
- 若要旋转 `Rectangle`，请将两根手指放在 `Rectangle` 上并同向转动手指。

若要引起惯性，请在执行之前的操作时将手指从屏幕上快速抬起。`Rectangle` 将继续移动、重设大小或旋转几秒钟，然后才停止。

## 另请参阅

- [UIElement.ManipulationStarting](#)
- [UIElement.ManipulationDelta](#)
- [UIElement.ManipulationInertiaStarting](#)

# 输入和命令帮助主题

项目 • 2023/02/06

本节中的主题介绍了如何使用 Windows Presentation Foundation (WPF) 中的输入和命令基础结构。

## 本节内容

- [启用命令](#)
- [更改光标类型](#)
- [使用焦点事件更改元素的颜色](#)
- [向控件应用 FocusVisualStyle](#)
- [检测何时按下 Enter 键](#)
- [使用事件创建变换效果](#)
- [使对象跟随鼠标指针移动](#)
- [创建 RoutedCommand](#)
- [实现 ICommandSource](#)
- [将命令挂钩到不支持命令的控件](#)
- [将命令挂钩到支持命令的控件](#)

## 参考

- [UIElement](#)
- [FrameworkElement](#)
- [ContentElement](#)
- [FrameworkContentElement](#)
- [Keyboard](#)
- [Mouse](#)
- [FocusManager](#)

## 相关章节

# 如何：启用命令

项目 • 2023/02/06

以下示例演示如何在 Windows Presentation Foundation (WPF) 中使用命令。示例演示如何将 `RoutedCommand` 关联到 `Button`、创建 `CommandBinding` 以及创建实现 `RoutedCommand` 的事件处理程序。有关命令的详细信息，请参阅[命令概述](#)。

## 示例

第一部分代码创建了用户界面 (UI)，界面由 `Button` 和 `StackPanel` 组成，并且创建了将命令处理程序与 `RoutedCommand` 关联的 `CommandBinding`。

`Button` 的 `Command` 属性与 `Close` 命令相关联。

将 `CommandBinding` 添加到根 `Window` 的 `CommandBindingCollection`。`Executed` 和 `CanExecute` 事件处理程序附加到此绑定，并与 `Close` 命令相关联。

没有 `CommandBinding` 就没有命令逻辑，只有调用命令的机制。单击 `Button` 时，将对命令目标引发 `PreviewExecutedRoutedEventArgs`，随后引发 `ExecutedRoutedEventArgs`。这些事件会遍历元素树，为该特定命令寻找 `CommandBinding`。值得注意的是，由于 `RoutedEventArgs` 通过元素树使用 tunnel 和 bubble 机制，因此必须注意放置 `CommandBinding` 的位置。如果 `CommandBinding` 位于命令目标的同级节点或位于不在 `RoutedEventArgs` 路由上的另一个节点上，则不会访问 `CommandBinding`。

XAML

```
<Window x:Class="WCSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="CloseCommand"
    Name="RootWindow"
    >
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Close"
            Executed="CloseCommandHandler"
            CanExecute="CanExecuteHandler"
            />
    </Window.CommandBindings>
    <StackPanel Name="MainStackPanel">
        <Button Command="ApplicationCommands.Close"
            Content="Close File" />
    </StackPanel>
</Window>
```

C#

```

// Create ui elements.
StackPanel CloseCmdStackPanel = new StackPanel();
Button CloseCmdButton = new Button();
CloseCmdStackPanel.Children.Add(CloseCmdButton);

// Set Button's properties.
CloseCmdButton.Content = "Close File";
CloseCmdButton.Command = ApplicationCommands.Close;

// Create the CommandBinding.
CommandBinding CloseCommandBinding = new CommandBinding(
    ApplicationCommands.Close, CloseCommandHandler, CanExecuteHandler);

// Add the CommandBinding to the root Window.
RootWindow.CommandBindings.Add(CloseCommandBinding);

```

下一部代码分实现了 [Executed](#) 和 [CanExecute](#) 事件处理程序。

[Executed](#) 处理程序调用了一个方法来关闭打开的文件。 [CanExecute](#) 处理程序调用了一个方法来确定文件是否打开。 如果文件处于打开状态，则将 [CanExecute](#) 设置为 `true`，否则，设置为 `false`。

C#

```

// Executed event handler.
private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
    // Calls a method to close the file and release resources.
    CloseFile();
}

// CanExecute event handler.
private void CanExecuteHandler(object sender, CanExecuteRoutedEventArgs e)
{
    // Call a method to determine if there is a file open.
    // If there is a file open, then set CanExecute to true.
    if (IsFileOpened())
    {
        e.CanExecute = true;
    }
    // if there is not a file open, then set CanExecute to false.
    else
    {
        e.CanExecute = false;
    }
}

```

## 另请参阅

- 命令概述

# 如何：更改光标类型

项目 • 2023/02/06

此示例演示如何为特定元素和应用程序更改鼠标指针的 [Cursor](#)。

此示例包含 Extensible Application Markup Language (XAML) 文件和代码隐藏文件。

## 示例

创建用户界面，它包括一个 [ComboBox](#) 来选择所需的 [Cursor](#)，一对 [RadioButton](#) 对象来确定光标更改是仅应用于单个元素还是应用于整个应用程序，以及一个 [Border](#) 作为新光标应用于的元素。

XAML

```
<StackPanel>
    <Border Width="300">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center">
            <StackPanel Margin="10">
                <Label HorizontalAlignment="Left">Cursor Type</Label>
                <ComboBox Width="100"
                          SelectionChanged="CursorTypeChanged"
                          HorizontalAlignment="Left"
                          Name="CursorSelector">
                    <ComboBoxItem Content="AppStarting" />
                    <ComboBoxItem Content="ArrowCD" />
                    <ComboBoxItem Content="Arrow" />
                    <ComboBoxItem Content="Cross" />
                    <ComboBoxItem Content="HandCursor" />
                    <ComboBoxItem Content="Help" />
                    <ComboBoxItem Content="IBeam" />
                    <ComboBoxItem Content="No" />
                    <ComboBoxItem Content="None" />
                    <ComboBoxItem Content="Pen" />
                    <ComboBoxItem Content="ScrollSE" />
                    <ComboBoxItem Content="ScrollWE" />
                    <ComboBoxItem Content="SizeAll" />
                    <ComboBoxItem Content="SizeNESW" />
                    <ComboBoxItem Content="SizeNS" />
                    <ComboBoxItem Content="SizeNWSE" />
                    <ComboBoxItem Content="SizeWE" />
                    <ComboBoxItem Content="UpArrow" />
                    <ComboBoxItem Content="WaitCursor" />
                    <ComboBoxItem Content="Custom" />
                </ComboBox>
            </StackPanel>
        </StackPanel>
    </Border>
    <!-- The user can select different cursor types using this ComboBox -->
```

```

<StackPanel Margin="10">
    <Label HorizontalAlignment="Left">Scope of Cursor</Label>
    <StackPanel>
        <RadioButton Name="rbScopeElement" IsChecked="True"
                    Checked="CursorScopeSelected">Display Area
                    Only</RadioButton>
        <RadioButton Name="rbScopeApplication"
                    Checked="CursorScopeSelected">Entire
                    Application</RadioButton>
    </StackPanel>
</StackPanel>
</Border>
<!-- When the mouse pointer is over this Border -->
<!-- the selected cursor type is shown -->
<Border Name="DisplayArea" Height="250" Width="400"
        Margin="20" Background="AliceBlue">
    <Label HorizontalAlignment="Center">
        Move Mouse Pointer Over This Area
    </Label>
</Border>
</StackPanel>

```

下面的代码创建了一个 `SelectionChanged` 事件处理程序，当在 `ComboBox` 中更改光标类型时调用该处理程序。 `switch` 语句筛选光标名称并在名为 `DisplayArea` 的 `Border` 上设置 `Cursor` 属性。

如果光标更改设置为“整个应用程序”，则 `OverrideCursor` 属性设置为 `Border` 控件的 `Cursor` 属性。 这会强制整个应用程序更改光标。

C#

```

private void CursorTypeChanged(object sender, SelectionChangedEventArgs e)
{
    ComboBox source = e.Source as ComboBox;

    if (source != null)
    {
        ComboBoxItem selectedCursor = source.SelectedItem as ComboBoxItem;

        // Changing the cursor of the Border control
        // by setting the Cursor property
        switch (selectedCursor.Content.ToString())
        {
            case "AppStarting":
                DisplayArea.Cursor = Cursors.AppStarting;
                break;
            case "ArrowCD":
                DisplayArea.Cursor = Cursors.ArrowCD;
                break;
            case "Arrow":
                DisplayArea.Cursor = Cursors.Arrow;
                break;
        }
    }
}

```

```
        break;
    case "Cross":
        DisplayArea.Cursor = Cursors.Cross;
        break;
    case "HandCursor":
        DisplayArea.Cursor = Cursors.Hand;
        break;
    case "Help":
        DisplayArea.Cursor = Cursors.Help;
        break;
    case "IBeam":
        DisplayArea.Cursor = Cursors.IBeam;
        break;
    case "No":
        DisplayArea.Cursor = Cursors.No;
        break;
    case "None":
        DisplayArea.Cursor = Cursors.None;
        break;
    case "Pen":
        DisplayArea.Cursor = Cursors.Pen;
        break;
    case "ScrollSE":
        DisplayArea.Cursor = Cursors.ScrollSE;
        break;
    case "ScrollWE":
        DisplayArea.Cursor = Cursors.ScrollWE;
        break;
    case "SizeAll":
        DisplayArea.Cursor = Cursors.SizeAll;
        break;
    case "SizeNESW":
        DisplayArea.Cursor = Cursors.SizeNESW;
        break;
    case "SizeNS":
        DisplayArea.Cursor = Cursors.SizeNS;
        break;
    case "SizeNWSE":
        DisplayArea.Cursor = Cursors.SizeNWSE;
        break;
    case "SizeWE":
        DisplayArea.Cursor = Cursors.SizeWE;
        break;
    case "UpArrow":
        DisplayArea.Cursor = Cursors.UpArrow;
        break;
    case "WaitCursor":
        DisplayArea.Cursor = Cursors.Wait;
        break;
    case "Custom":
        DisplayArea.Cursor = CustomCursor;
        break;
    default:
        break;
}
```

```
// If the cursor scope is set to the entire application
// Use OverrideCursor to force the cursor for all elements
if (cursorScopeElementOnly == false)
{
    Mouse.OverrideCursor = DisplayArea.Cursor;
}
}
```

## 请参阅

- [输入概述](#)

# 如何：使用焦点事件更改元素的颜色

项目 • 2023/02/06

此示例演示如何在元素使用 `GotFocus` 和 `LostFocus` 事件获取和失去焦点时更改元素的颜色。

此示例包含 Extensible Application Markup Language (XAML) 文件和代码隐藏文件。

## 示例

以下 XAML 创建用户界面，该用户界面包含两个 `Button` 对象，并将 `GotFocus` 和 `LostFocus` 事件的事件处理程序附加到 `Button` 对象。

XAML

```
<StackPanel>
  <StackPanel.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="Height" Value="20"/>
      <Setter Property="Width" Value="250"/>
      <Setter Property="HorizontalAlignment" Value="Left"/>
    </Style>
  </StackPanel.Resources>
  <Button
    GotFocus="OnGotFocusHandler"
    LostFocus="OnLostFocusHandler">Click Or Tab To Give Keyboard
Focus</Button>
  <Button
    GotFocus="OnGotFocusHandler"
    LostFocus="OnLostFocusHandler">Click Or Tab To Give Keyborad
Focus</Button>
</StackPanel>
```

下面的代码隐藏文件创建 `GotFocus` 和 `LostFocus` 事件处理程序。当 `Button` 获得键盘焦点时，`Button` 的 `Background` 变为红色。当 `Button` 失去键盘焦点时，`Button` 的 `Background` 变回白色。

C#

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

  // Raised when Button gains focus.
```

```
// Changes the color of the Button to Red.  
private void OnGotFocusHandler(object sender, RoutedEventArgs e)  
{  
    Button tb = e.Source as Button;  
    tb.Background = Brushes.Red;  
}  
// Raised when Button losses focus.  
// Changes the color of the Button back to white.  
private void OnLostFocusHandler(object sender, RoutedEventArgs e)  
{  
    Button tb = e.Source as Button;  
    tb.Background = Brushes.White;  
}  
}
```

## 请参阅

- [输入概述](#)

# 如何：对控件应用 FocusVisualStyle

项目 • 2023/02/06

此示例演示了如何使用 [FocusVisualStyle](#) 属性在资源中创建焦点视觉样式并将该样式应用于控件。

## 示例

以下示例定义的样式创建了其他控件组合，此组合仅在控件在用户界面 (UI) 中使用键盘设定为焦点时才适用。这是通过使用 [ControlTemplate](#) 定义样式，然后在设置 [FocusVisualStyle](#) 属性时将该样式引用为资源来实现的。

一个类似于边框的外部矩形将放置在矩形区域之外。除非另有修改，否则调整样式大小时将使用应用焦点视觉样式的矩形控件的 [ActualHeight](#) 和 [ActualWidth](#)。此示例将 [Margin](#) 设置为负值，以使边框略微超出焦点控件。

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Page.Resources>
        <Style x:Key="MyFocusVisual">
            <Setter Property="Control.Template">
                <Setter.Value>
                    <ControlTemplate>
                        <Rectangle Margin="-2" StrokeThickness="1" Stroke="Red"
                        StrokeDashArray="1 2"/>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>
    <StackPanel Background="Ivory" Orientation="Horizontal">
        <Canvas Width="10"/>
        <Button Width="100" Height="30" FocusVisualStyle="{DynamicResource
        MyFocusVisual}">
            Focus Here</Button>
        <Canvas Width="100"/>
        <Button Width="100" Height="30" FocusVisualStyle="{DynamicResource
        MyFocusVisual}">
            Focus Here</Button>
    </StackPanel>
</Page>
```

[FocusVisualStyle](#) 可添加到任何控件模板样式（来自显式样式或主题样式）；仍然可以通过使用 [ControlTemplate](#) 并将该样式设置为 [Style](#) 属性来创建控件的主要样式。

主题或 UI 中使用的焦点视觉样式应一致，而不是针对每个可聚焦的元素使用不同的视觉样式。有关详细信息，请参阅[为控件中的焦点设置样式以及 FocusVisualStyle](#)。

## 请参阅

- [FocusVisualStyle](#)
- [样式设置和模板化](#)
- [为控件中的焦点设置样式以及 FocusVisualStyle](#)

# 如何：检测何时按下 Enter 键

项目 • 2023/02/06

此示例演示如何检测何时按下键盘上的 [Enter](#) 键。

此示例包含 Extensible Application Markup Language (XAML) 文件和代码隐藏文件。

## 示例

用户按下 [TextBox](#) 中的 [Enter](#) 键后，文本框中的输入将显示在用户界面 (UI) 的另一个区域中。

以下 XAML 创建了用户界面，该用户界面由一个 [StackPanel](#)、一个 [TextBlock](#) 和一个 [TextBox](#) 组成。

XAML

```
<StackPanel>
    <TextBlock Width="300" Height="20">
        Type some text into the TextBox and press the Enter key.
    </TextBlock>
    <TextBox Width="300" Height="30" Name="textBox1"
        KeyDown="OnKeyDownHandler"/>
    <TextBlock Width="300" Height="100" Name="textBlock1"/>
</StackPanel>
```

以下代码隐藏文件创建了 [KeyDown](#) 事件处理程序。如果按下的键是 [Enter](#) 键，则会在 [TextBlock](#) 中显示消息。

C#

```
private void OnKeyDownHandler(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Return)
    {
        textBlock1.Text = "You Entered: " + textBox1.Text;
    }
}
```

## 请参阅

- [输入概述](#)
- [路由事件概述](#)

# 如何：使用事件创建翻转效果

项目 • 2023/02/06

本示例演示如何更改鼠标指针进入和离开由元素占用的区域时的元素颜色。

此示例包含 Extensible Application Markup Language (XAML) 文件和代码隐藏文件。

## ① 备注

此示例演示如何使用事件，但实现此相同效果的建议方法是在样式中使用 Trigger。有关详细信息，请参阅[样式设置和模板化](#)。

## 示例

以下 XAML 创建用户界面，该界面上包括 TextBlock 周围的 Border，并将 MouseEnter 和 MouseLeave 事件处理程序附加到 Border。

XAML

```
<StackPanel>
    <Border MouseEnter="OnMouseEnterHandler"
            MouseLeave="OnMouseLeaveHandler"
            Name="border1" Margin="10"
            BorderThickness="1"
            BorderBrush="Black"
            VerticalAlignment="Center"
            Width="300" Height="100">
        <Label Margin="10" FontSize="14"
              HorizontalAlignment="Center">Move Cursor Over Me</Label>
    </Border>
</StackPanel>
```

下面的代码隐藏文件创建 MouseEnter 和 MouseLeave 事件处理程序。当鼠标指针进入 Border 时，Border 的背景将变为红色。当鼠标指针离开 Border 时，Border 的背景将变回白色。

C#

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

```
// raised when mouse cursor enters the area occupied by the element
void OnMouseEnterHandler(object sender, MouseEventArgs e)
{
    border1.Background = Brushes.Red;
}

// raised when mouse cursor leaves the area occupied by the element
void OnMouseLeaveHandler(object sender, MouseEventArgs e)
{
    border1.Background = Brushes.White;
}
}
```

# 如何：使对象跟随鼠标指针移动

项目 • 2023/02/06

此示例演示如何在鼠标指针在屏幕上移动时更改对象的尺寸。

该示例包括一个用户界面 (UI) 和一个用于创建事件处理程序的代码隐藏文件。

## 示例

以下 XAML 创建了 UI，它由 StackPanel 内的 Ellipse 组成，并附加了 MouseMove 事件的事件处理程序。

XAML

```
<Window x:Class="WCSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="mouseMoveWithPointer"
    Height="400"
    Width="500"
    >
<Canvas MouseMove="MouseMoveHandler"
    Background="LemonChiffon">
    <Ellipse Name="ellipse" Fill="LightBlue"
        Width="100" Height="100"/>
</Canvas>
</Window>
```

以下代码隐藏文件创建了 MouseMove 事件处理程序。当鼠标指针移动时，Ellipse 的高度和宽度会随之增加和减少。

C#

```
// raised when the mouse pointer moves.
// Expands the dimensions of an Ellipse when the mouse moves.
private void MouseMoveHandler(object sender, MouseEventArgs e)
{
    // Get the x and y coordinates of the mouse pointer.
    System.Windows.Point position = e.GetPosition(this);
    double pX = position.X;
    double pY = position.Y;

    // Sets the Height/Width of the circle to the mouse coordinates.
    ellipse.Width = pX;
    ellipse.Height = pY;
}
```

# 请参阅

- [输入概述](#)

# 如何：创建 RoutedCommand

项目 • 2023/02/06

此示例演示如何创建自定义 [RoutedCommand](#)，以及如何通过创建 [ExecutedRoutedEventHandler](#) 和 [CanExecuteRoutedEventArgs](#) 并将它们附加到 [CommandBinding](#) 来实现自定义命令。有关命令的详细信息，请参阅[命令概述](#)。

## 示例

创建 [RoutedCommand](#) 的第一步是定义命令并将其实例化。

C#

```
public static RoutedCommand CustomRoutedCommand = new RoutedCommand();
```

为了在应用程序中使用命令，必须创建定义命令所执行的操作的事件处理程序

C#

```
private void ExecutedCustomCommand(object sender,
    ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Custom Command Executed");
}
```

C#

```
// CanExecuteRoutedEventHandler that only returns true if
// the source is a control.
private void CanExecuteCustomCommand(object sender,
    CanExecuteRoutedEventArgs e)
{
    Control target = e.Source as Control;

    if(target != null)
    {
        e.CanExecute = true;
    }
    else
    {
        e.CanExecute = false;
    }
}
```

接下来，创建一个 [CommandBinding](#)，它将命令与事件处理程序关联起来。[CommandBinding](#) 是在特定对象上创建的。此对象定义了元素树中 [CommandBinding](#) 的范围

#### XAML

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:custom="clr-namespace:SDKSamples"
    Height="600" Width="800"
    >
<Window.CommandBindings>
    <CommandBinding Command="{x:Static custom:Window1.CustomRoutedCommand}"
        Executed="ExecutedCustomCommand"
        CanExecute="CanExecuteCustomCommand" />
</Window.CommandBindings>
```

#### C#

```
CommandBinding customCommandBinding = new CommandBinding(
    CustomRoutedCommand, ExecutedCustomCommand, CanExecuteCustomCommand);

// attach CommandBinding to root window
this.CommandBindings.Add(customCommandBinding);
```

最后一步是调用命令。调用命令的一种方法是将其与 [ICommandSource](#) 关联，例如 [Button](#)。

#### XAML

```
<StackPanel>
    <Button Command="{x:Static custom:Window1.CustomRoutedCommand}"
        Content="CustomRoutedCommand"/>
</StackPanel>
```

#### C#

```
// create the ui
StackPanel CustomCommandStackPanel = new StackPanel();
Button CustomCommandButton = new Button();
CustomCommandStackPanel.Children.Add(CustomCommandButton);

CustomCommandButton.Command = CustomRoutedCommand;
```

单击按钮时，将调用自定义 [RoutedCommand](#) 上的 [Execute](#) 方法。[RoutedCommand](#) 引发 [PreviewExecuted](#) 和 [Executed](#) 路由事件。这些事件会遍历元素树，查找此特定命令的

CommandBinding。如果找到 CommandBinding，则调用与 CommandBinding 关联的 ExecutedRoutedEventHandler。

## 另请参阅

- [RoutedCommand](#)
- [命令概述](#)

# 如何：实现 ICommandSource

项目 • 2022/09/27

此示例演示如何通过实现 [ICommandSource](#) 创建命令源。 命令源是知道如何调用命令的对象。[ICommandSource](#) 接口公开三个成员：

- [Command](#)：将调用的命令。
- [CommandParameter](#)：从命令源传递到处理命令的方法的用户定义数据类型。
- [CommandTarget](#)：在其上执行该命令的对象。

在此示例中，创建的类将继承自 [Slider](#) 控件并实现 [ICommandSource](#) 接口。

## 示例

WPF 提供了几个实现 [ICommandSource](#) 的类，如 [Button](#)、[MenuItem](#) 和 [Hyperlink](#)。 命令源定义如何调用命令。 这些类在被单击时调用命令，并且仅在它们的 [Command](#) 属性被设置时成为命令源。

在此示例中，当滑块移动时，或者更准确地说，当 [Value](#) 属性更改时，你将调用该命令。

下面是类定义：

C#

```
public class CommandSlider : Slider, ICommandSource
{
    public CommandSlider() : base()
    {
    }
}
```

下一步是实现 [ICommandSource](#) 成员。 在此示例中，属性作为 [DependencyProperty](#) 对象实现。 这使属性能够使用数据绑定。 有关 [DependencyProperty](#) 类的详细信息，请参阅[依赖属性概述](#)。 有关数据绑定的详细信息，请参阅[数据绑定概述](#)。

此处仅显示 [Command](#) 属性。

C#

```
// Make Command a dependency property so it can use databinding.
public static readonly DependencyProperty CommandProperty =
    DependencyProperty.Register(
        "Command",
        typeof(ICommand),
        typeof(CommandSlider),
        new PropertyMetadata((ICommand)null,
```

```
        new PropertyChangedCallback(CommandChanged)));  
  
    public ICommand Command  
{  
    get  
    {  
        return (ICommand)GetValue(CommandProperty);  
    }  
    set  
    {  
        SetValue(CommandProperty, value);  
    }  
}
```

下面是 [DependencyProperty](#) 更改回调：

C#

```
// Command dependency property change callback.  
private static void CommandChanged(DependencyObject d,  
    DependencyPropertyChangedEventArgs e)  
{  
    CommandSlider cs = (CommandSlider)d;  
    cs.HookUpCommand((ICommand)e.OldValue, (ICommand)e.NewValue);  
}
```

下一步是添加和删除与命令源关联的命令。添加新命令时，不能只是覆盖 [Command](#) 属性，因为必须先删除与上一个命令关联的事件处理程序（如果有）。

C#

```
// Add a new command to the Command Property.  
private void HookUpCommand(ICommand oldCommand, ICommand newCommand)  
{  
    // If oldCommand is not null, then we need to remove the handlers.  
    if (oldCommand != null)  
    {  
        RemoveCommand(oldCommand, newCommand);  
    }  
    AddCommand(oldCommand, newCommand);  
}  
  
// Remove an old command from the Command Property.  
private void RemoveCommand(ICommand oldCommand, ICommand newCommand)  
{  
    EventHandler handler = CanExecuteChanged;  
    oldCommand.CanExecuteChanged -= handler;  
}  
  
// Add the command.  
private void AddCommand(ICommand oldCommand, ICommand newCommand)
```

```
{  
    EventHandler handler = new EventHandler(CanExecuteChanged);  
    canExecuteChangedHandler = handler;  
    if (newCommand != null)  
    {  
        newCommand.CanExecuteChanged += canExecuteChangedHandler;  
    }  
}
```

下一步是为 [CanExecuteChanged](#) 处理程序创建逻辑。

[CanExecuteChanged](#) 事件通知命令源在当前命令目标上执行命令的能力可能已发生更改。当命令源收到此事件时，它通常对命令调用 [CanExecute](#) 方法。如果命令不能在当前命令目标上执行，命令源通常会禁用自身。如果命令能在当前命令目标上执行，命令源通常会启用自身。

C#

```
private void CanExecuteChanged(object sender, EventArgs e)  
{  
  
    if (this.Command != null)  
    {  
        RoutedCommand command = this.Command as RoutedCommand;  
  
        // If a RoutedCommand.  
        if (command != null)  
        {  
            if (command.CanExecute(CommandParameter, CommandTarget))  
            {  
                this.IsEnabled = true;  
            }  
            else  
            {  
                this.IsEnabled = false;  
            }  
        }  
        // If a not RoutedCommand.  
        else  
        {  
            if (Command.CanExecute(CommandParameter))  
            {  
                this.IsEnabled = true;  
            }  
            else  
            {  
                this.IsEnabled = false;  
            }  
        }  
    }  
}
```

最后一步是 Execute 方法。如果命令是一个 RoutedCommand，则会调用 RoutedCommandExecute 方法；否则调用 ICommandExecute 方法。

C#

```
// If Command is defined, moving the slider will invoke the command;
// Otherwise, the slider will behave normally.
protected override void OnValueChanged(double oldValue, double newValue)
{
    base.OnValueChanged(oldValue, newValue);

    if (this.Command != null)
    {
        RoutedCommand command = Command as RoutedCommand;

        if (command != null)
        {
            command.Execute(CommandParameter, CommandTarget);
        }
        else
        {
            ((ICommand)Command).Execute(CommandParameter);
        }
    }
}
```

## 另请参阅

- [ICommandSource](#)
- [ICommand](#)
- [RoutedCommand](#)
- [命令概述](#)

# 如何：将命令挂钩到不支持命令的控件

项目 • 2023/02/06

以下示例演示如何将 [RoutedCommand](#) 挂钩到不含对该命令的内置支持的 [Control](#)。有关将命令挂钩到多个源的完整示例，请参阅[创建自定义 RoutedCommand 示例](#)示例。

## 示例

Windows Presentation Foundation (WPF) 提供了应用程序程序员经常遇到的常见命令库。构成命令库的类为：[ApplicationCommands](#)、[ComponentCommands](#)、[NavigationCommands](#)、[MediaCommands](#) 和 [EditingCommands](#)。

构成这些类的静态 [RoutedCommand](#) 对象不提供命令逻辑。命令的逻辑通过 [CommandBinding](#) 与命令相关联。WPF 中的许多控件都为命令库中的某些命令提供内置支持。例如，[TextBox](#) 支持许多应用程序编辑命令，例如 [Paste](#)、[Copy](#)、[Cut](#)、[Redo](#) 和 [Undo](#)。应用程序开发人员不必执行任何特殊操作即可使命令适用于这些控件。如果命令执行时，[TextBox](#) 是命令目标，它将使用内置于控件中的 [CommandBinding](#) 处理该命令。

下面演示了如何将 [Button](#) 用作 [Open](#) 命令的命令源。创建一个将指定的 [CanExecuteRoutedEventHandler](#) 和 [CanExecuteRoutedEventArgs](#) 与 [RoutedCommand](#) 相关联的 [CommandBinding](#)。

首先，创建命令源。将 [Button](#) 用作命令源。

XAML

```
<Button Command="ApplicationCommands.Open" Name="MyButton"
        Height="50" Width="200">
    Open (KeyBindings: Ctrl+R, Ctrl+0)
</Button>
```

C#

```
// Button used to invoke the command
Button CommandButton = new Button();
CommandButton.Command = ApplicationCommands.Open;
CommandButton.Content = "Open (KeyBindings: Ctrl-R, Ctrl-0)";
MainStackPanel.Children.Add(CommandButton);
```

接下来，创建 [ExecutedRoutedEventHandler](#) 和 [CanExecuteRoutedEventArgs](#)。[ExecutedRoutedEventArgs](#) 只需打开 [MessageBox](#) 即可表示已执行该命令。

`CanExecuteRoutedEventHandler` 将 `CanExecute` 属性设置为 `true`。通常，`can execute` 处理程序将执行更可靠的检查，确定是否可以在当前命令目标上执行该命令。

C#

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    String command, targetobj;
    command = ((RoutedCommand)e.Command).Name;
    targetobj = ((FrameworkElement)target).Name;
    MessageBox.Show("The " + command + " command has been invoked on target
object " + targetobj);
}
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = true;
}
```

最后，在将路由事件处理程序与 `Open` 命令相关联的应用程序的根 `Window` 上创建 `CommandBinding`。

XAML

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
                    Executed="OpenCmdExecuted"
                    CanExecute="OpenCmdCanExecute"/>
</Window.CommandBindings>
```

C#

```
// Creating CommandBinding and attaching an Executed and CanExecute handler
CommandBinding OpenCmdBinding = new CommandBinding(
    ApplicationCommands.Open,
    OpenCmdExecuted,
    OpenCmdCanExecute);

this.CommandBindings.Add(OpenCmdBinding);
```

## 另请参阅

- [命令概述](#)
- [将命令挂钩到支持命令的控件](#)

# 如何：将命令挂钩到支持命令的控件

项目 • 2023/02/06

以下示例显示如何将 [RoutedCommand](#) 挂钩到含对该命令的内置支持的 [Control](#)。有关将命令挂钩到多个源的完整示例，请参阅[创建自定义 RoutedCommand 示例](#)示例。

## 示例

Windows Presentation Foundation (WPF) 提供了应用程序程序员经常遇到的常见命令库。构成命令库的类为：[ApplicationCommands](#)、[ComponentCommands](#)、[NavigationCommands](#)、[MediaCommands](#) 和 [EditingCommands](#)。

构成这些类的静态 [RoutedCommand](#) 对象不提供命令逻辑。命令的逻辑通过 [CommandBinding](#) 与命令相关联。某些控件的部分命令具有内置的 [CommandBindings](#)。这种机制可使命令的语义保持不变，而实际实现可以更改。例如，[TextBox](#) 对 [Paste](#) 命令的处理方式与专门支持图像的控件对其的处理方式不同，后者的基本思路是通过粘贴让内容保持不变。命令无法提供命令逻辑，但是控件或应用程序必须提供命令逻辑。

WPF 中的许多控件都为命令库中的某些命令提供内置支持。例如，[TextBox](#) 支持许多应用程序编辑命令，例如 [Paste](#)、[Copy](#)、[Cut](#)、[Redo](#) 和 [Undo](#)。应用程序开发人员不必执行任何特殊操作即可使命令适用于这些控件。如果命令执行时，[TextBox](#) 是命令目标，它将使用内置于控件中的 [CommandBinding](#) 处理该命令。

以下说明如何将 [MenuItem](#) 用作 [Paste](#) 命令的命令源，其中 [TextBox](#) 是命令的目标。定义 [TextBox](#) 如何执行粘贴的所有逻辑都已内置于 [TextBox](#) 控件中。

已创建 [MenuItem](#)，并将其 [Command](#) 属性设置为 [Paste](#) 命令。[CommandTarget](#) 未显式设置为 [TextBox](#) 对象。未设置 [CommandTarget](#) 时，该命令的目标是具有键盘焦点的元素。如果具有键盘焦点的元素不支持 [Paste](#) 命令，或当前无法执行粘贴命令（例如，剪贴板为空），则 [MenuItem](#) 为灰显。

XAML

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MenuItemCommandTask"
>
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <MenuItem Command="ApplicationCommands.Paste" Width="75" />
    </Menu>
    <TextBox BorderBrush="Black" BorderThickness="2" Margin="25"
```

```
        TextWrapping="Wrap">
    The MenuItem will not be enabled until
    this TextBox gets keyboard focus
</TextBox>
</DockPanel>
</Window>
```

C#

```
// Window1 constructor
public Window1()
{
    InitializeComponent();

    // Instantiating UIElements.
    DockPanel mainPanel = new DockPanel();
    Menu mainMenu = new Menu();
    MenuItem pasteMenuItem = new MenuItem();
    TextBox mainTextBox = new TextBox();

    // Associating the MenuItem with the Paste command.
    pasteMenuItem.Command = ApplicationCommands.Paste;

    // Setting properties on the TextBox.
    mainTextBox.Text =
        "The MenuItem will not be enabled until this TextBox receives
    keyboard focus.";
    mainTextBox.Margin = new Thickness(25);
    mainTextBox.BorderBrush = Brushes.Black;
    mainTextBox.BorderThickness = new Thickness(2);
    mainTextBox.TextWrapping = TextWrapping.Wrap;

    // Attaching UIElements to the Window.
    this.AddChild(mainPanel);
    mainMenu.Items.Add(pasteMenuItem);
    mainPanel.Children.Add(mainMenu);
    mainPanel.Children.Add(mainTextBox);

    // Defining DockPanel layout.
    DockPanel.SetDock(mainMenu, Dock.Top);
    DockPanel.SetDock(mainTextBox, Dock.Bottom);
}
```

## 另请参阅

- 命令概述
- 将命令挂钩到不支持命令的控件

# 数字墨迹

项目 • 2023/02/06

本节讨论 WPF 中数字墨迹的使用。 数字墨迹过去只能在平板电脑 SDK 中找到，现在也在核心 Windows Presentation Foundation 提供。 这意味着你现在可以使用 Windows Presentation Foundation 的强大功能开发成熟的平板电脑应用程序。

## 本节内容

[概述](#)

[操作指南主题](#)

## 相关章节

[Windows Presentation Foundation](#)

# 数字墨迹概述

项目 • 2023/02/06

## 本节内容

- 墨迹入门
- 收集墨迹
- 手写识别
- 存储墨迹
- 墨迹对象模型：Windows 窗体和 COM 与 WPF
- 高级墨迹处理

# WPF 中的墨迹入门

项目 • 2022/10/05

Windows Presentation Foundation (WPF) 具有墨迹功能，可以轻松将数字墨迹合并到应用中。

## 先决条件

若要使用以下示例，请先安装 [Visual Studio](#)。它还有助于了解如何编写基本的 WPF 应用。有关 WPF 入门的帮助，请参阅 [演练：我的第一个 WPF 桌面应用程序](#)。

## 快速启动

本部分有助于编写收集墨迹的简单 WPF 应用程序。

### 是否有墨迹？

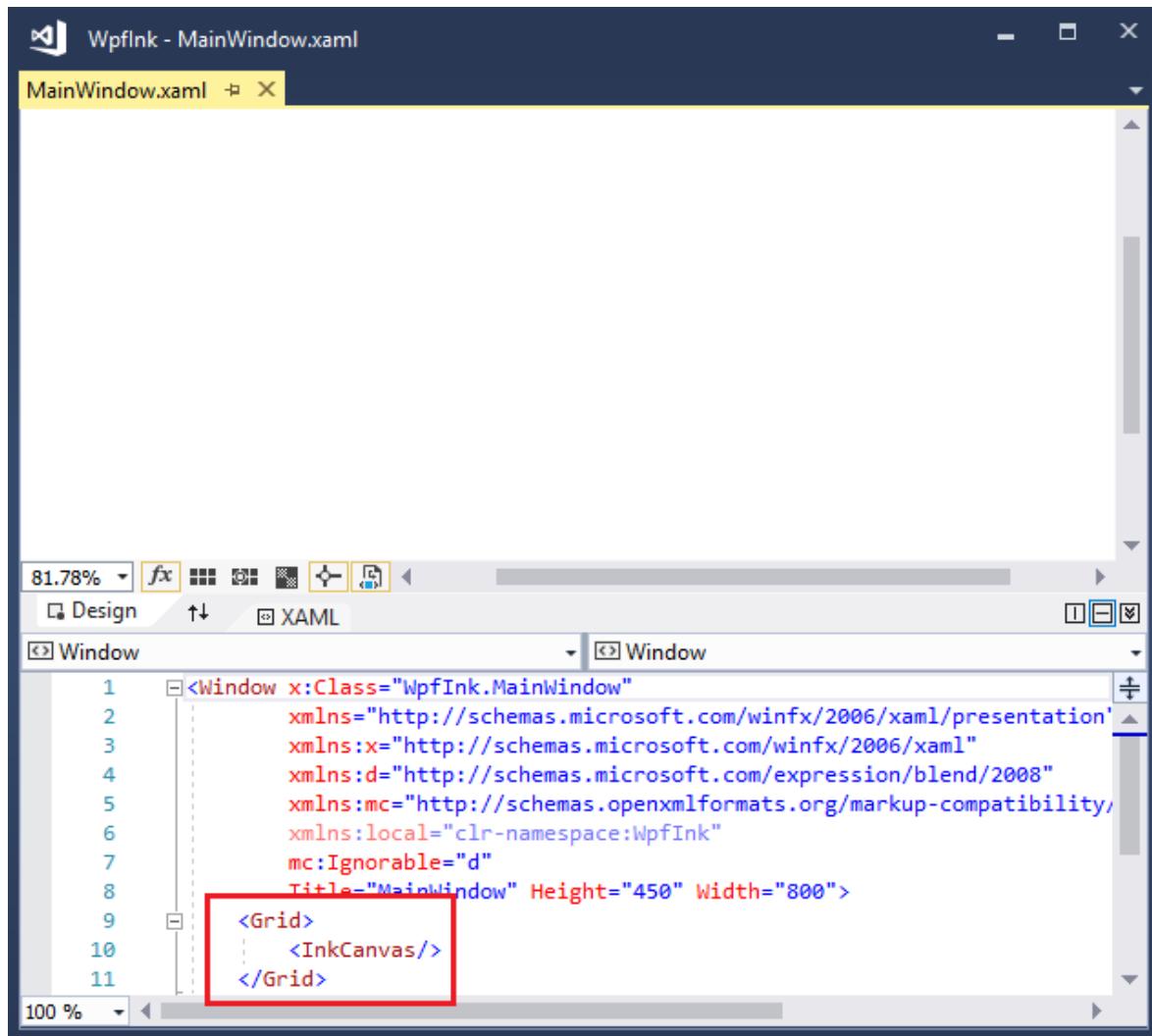
创建支持墨迹的 WPF 应用：

1. 打开 Visual Studio。
2. 创建新的 WPF 应用。

在“新建项目”对话框中，展开“已安装”>“Visual C#”或“Visual Basic”>“Windows 桌面”类别。然后选择“WPF 应用(.NET Framework)”应用模板。输入一个名称，然后选择“确定”。

Visual Studio 将创建工作，并在设计器中打开 MainWindow.xaml。

3. 在 `<Grid>` 标记之间键入 `<InkCanvas/>`。



4. 按 F5 在调试器中启动应用程序。
5. 使用触笔或鼠标，在窗口中写下“hello world”。

你仅击键 12 次便通过墨迹编写了“hello world”应用程序！

## 丰富你的应用

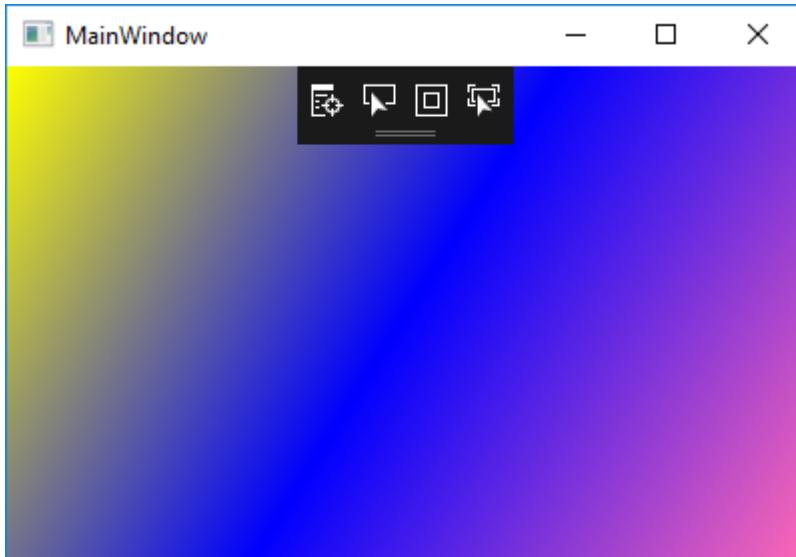
让我们利用一些 WPF 功能。将开始窗口标记和结束窗口标记之间的所有内容替换为以下标记<>：

```
XAML

<Page>
    <InkCanvas Name="myInkCanvas" MouseRightButtonUp="RightMouseUpHandler">
        <InkCanvas.Background>
            <LinearGradientBrush>
                <GradientStop Color="Yellow" Offset="0.0" />
                <GradientStop Color="Blue" Offset="0.5" />
                <GradientStop Color="HotPink" Offset="1.0" />
            </LinearGradientBrush>
        </InkCanvas.Background>
    </InkCanvas>
</Page>
```

```
</InkCanvas>  
</Page>
```

此 XAML 会在着墨迹式图面上创建渐变画笔背景。



## 在 XAML 后面添加一些代码

虽然 XAML 使设计用户界面变得非常简单，但任何实际应用程序都需要添加代码来处理事件。下面是一个简单的示例，其中放大了墨迹以响应鼠标右键单击。

1. 在 XAML 中设置 `MouseRightButtonUp` 处理程序：

```
XAML  
  
<InkCanvas Name="myInkCanvas" MouseRightButtonUp="RightMouseUpHandler">
```

2. 在“解决方案资源管理器”中，展开 `MainWindow.xaml` 并打开代码隐藏文件（`MainWindow.xaml.cs` 或 `MainWindow.xaml.vb`）。添加以下事件处理程序代码：

```
C#  
  
private void RightMouseUpHandler(object sender,  
System.Windows.Input.MouseEventArgs e)  
{  
    Matrix m = new Matrix();  
    m.Scale(1.1d, 1.1d);  
    ((InkCanvas)sender).Strokes.Transform(m, true);  
}
```

3. 运行应用。添加一些墨迹，然后右键单击鼠标或使用触笔按住（等效操作）。

每次单击鼠标右键时，都会放大显示内容。

## 使用程序代码而不是 XAML

可以通过程序代码访问所有 WPF 功能。按照以下步骤为 WPF 创建一个完全不使用任何 XAML 的“Hello Ink World”应用程序。

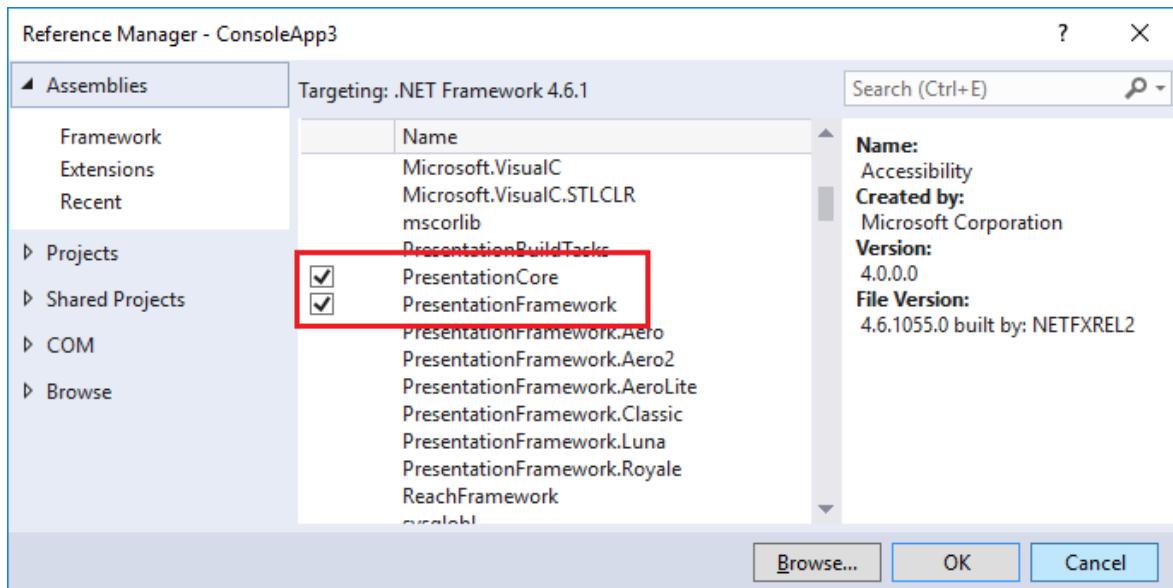
1. 在 Visual Studio 中创建新控制台应用程序项目。

在“新建项目”对话框中，展开“已安装”>“Visual C#”或“Visual Basic”>“Windows 桌面”类别。然后选择“控制台应用(.NET Framework)”应用模板。输入一个名称，然后选择“确定”。

2. 将以下代码粘贴到 Program.cs 或 Program.vb 文件中：

```
C#  
  
using System;  
using System.Windows;  
using System.Windows.Controls;  
class Program : Application  
{  
    Window win;  
    InkCanvas ic;  
  
    protected override void OnStartup(StartupEventArgs args)  
    {  
        base.OnStartup(args);  
        win = new Window();  
        ic = new InkCanvas();  
        win.Content = ic;  
        win.Show();  
    }  
  
    [STAThread]  
    static void Main(string[] args)  
    {  
        new Program().Run();  
    }  
}
```

3. 右键单击“解决方案资源管理器”中的“引用”，然后选择“添加引用”，以添加对 PresentationCore、PresentationFramework 和 WindowsBase 程序集的引用。



4. 按 F5 生成应用程序。

## 另请参阅

- [数字墨迹](#)
- [收集墨迹](#)
- [手写识别](#)
- [存储墨迹](#)

# 收集墨迹

项目 • 2022/09/27

Windows Presentation Foundation 平台会收集数字墨迹，这是其功能中的核心部分之一。本主题讨论在 Windows Presentation Foundation (WPF) 中收集墨迹的方法。

## 先决条件

若要使用以下示例，必须先安装 Visual Studio 和 Windows SDK。还应了解如何编写 WPF 的应用程序。有关 WPF 入门的详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。

## 使用 InkCanvas 元素

`System.Windows.Controls.InkCanvas` 元素提供在 WPF 中收集墨迹的最简单方式。使用 `InkCanvas` 元素可接收和显示墨迹输入。通常使用触笔（与数字化仪交互产生墨迹笔画）输入墨迹。此外，还可以使用鼠标代替触笔。创建的笔画表示为 `Stroke` 对象，它们可以通过编程方式和用户输入方式进行操作。`InkCanvas` 允许用户选择、修改或删除现有的 `Stroke`。

通过使用 XAML，可以轻松设置墨迹收集，就像将 `InkCanvas` 元素添加到树中一样轻松。以下示例向 Visual Studio 中创建的默认 WPF 项目添加一个 `InkCanvas`：

XAML

```
<Grid>
    <InkCanvas/>
</Grid>
```

`InkCanvas` 元素也可以包含子元素，这样便可以将墨迹批注功能添加到几乎所有类型的 XAML 元素。例如，要将墨迹功能添加到文本元素，只需使其成为 `InkCanvas` 的子元素即可：

XAML

```
<InkCanvas>
    <TextBlock>Show text here.</TextBlock>
</InkCanvas>
```

同样，可轻松添加图像墨迹标记支持：

## XAML

```
<InkCanvas>
    <Image Source="myPicture.jpg"/>
</InkCanvas>
```

## InkCollection 模式

InkCanvas 通过其 [EditMode](#) 属性提供对各种输入模式的支持。

## 操作墨迹

InkCanvas 支持许多墨迹编辑操作。例如，InkCanvas 支持笔后端清除功能，且不需使用其他代码将该功能添加到该元素。

## 选择

设置选择模式与将 [InkCanvasEditingStyle](#) 属性设置为 Select 一样容易。

下面的代码根据 [CheckBox](#) 的值来设置编辑模式：

### C#

```
// Set the selection mode based on a checkbox
if ((bool)cbSelectionMode.IsChecked)
{
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Select;
}
else
{
    theInkCanvas.EditingMode = InkCanvasEditingStyle.Ink;
}
```

## DrawingAttributes

使用 [DrawingAttributes](#) 属性更改墨迹笔划的外观。例如，[DrawingAttributes](#) 的 [Color](#) 成员设置呈现的 [Stroke](#) 的颜色。

下面的示例将所选笔划的颜色更改为红色：

### C#

```
// Get the selected strokes from the InkCanvas
StrokeCollection selection = theInkCanvas.GetSelectedStrokes();
```

```
// Check to see if any strokes are actually selected
if (selection.Count > 0)
{
    // Change the color of each stroke in the collection to red
    foreach (System.Windows.Ink.Stroke stroke in selection)
    {
        stroke.DrawingAttributes.Color = System.Windows.Media.Colors.Red;
    }
}
```

## DefaultDrawingAttributes

`DefaultDrawingAttributes` 属性可提供对在 `InkCanvas` 中创建的笔画的高度、宽度和颜色等属性的访问。更改 `DefaultDrawingAttributes` 之后，以后输入到 `InkCanvas` 中的所有笔划都将使用新的属性值来呈现。

除了在代码隐藏文件中修改 `DefaultDrawingAttributes` 外，还可以使用 XAML 语法指定 `DefaultDrawingAttributes` 属性。

下一个示例演示如何设置 `Color` 属性。若要使用此代码，请在 Visual Studio 中创建名为“HelloInkCanvas”的新 WPF 项目。将 `MainWindow.xaml` 中的代码替换为以下代码：

### XAML

```
<Window x:Class="HelloInkCanvas.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:Ink="clr-namespace:System.Windows.Ink;assembly=PresentationCore"
    Title="Hello, InkCanvas!" Height="300" Width="300"
    >
<Grid>
    <InkCanvas Name="inkCanvas1" Background="Ivory">
        <InkCanvas.DefaultDrawingAttributes>
            <Ink:DrawingAttributes xmlns:ink="system-windows-ink" Color="Red"
Width="5" />
        </InkCanvas.DefaultDrawingAttributes>
    </InkCanvas>

    <!-- This stack panel of buttons is a sibling to InkCanvas (not a child)
but overlapping it,
        higher in z-order, so that ink is collected and rendered behind -->
    <StackPanel Name="buttonBar" VerticalAlignment="Top" Height="26"
Orientation="Horizontal" Margin="5">
        <Button Click="Ink">Ink</Button>
        <Button Click="Highlight">Highlight</Button>
        <Button Click="EraseStroke">EraseStroke</Button>
        <Button Click="Select">Select</Button>
    </StackPanel>
</Grid>
</Window>
```

然后，将以下按钮事件处理程序添加到 MainWindow 类中的代码隐藏文件：

```
C#  
  
// Set the EditingMode to ink input.  
private void Ink(object sender, RoutedEventArgs e)  
{  
    inkCanvas1.EditingMode = InkCanvasEditingMode.Ink;  
  
    // Set the DefaultDrawingAttributes for a red pen.  
    inkCanvas1.DefaultDrawingAttributes.Color = Colors.Red;  
    inkCanvas1.DefaultDrawingAttributes.IsHighlighter = false;  
    inkCanvas1.DefaultDrawingAttributes.Height = 2;  
}  
  
// Set the EditingMode to highlighter input.  
private void Highlight(object sender, RoutedEventArgs e)  
{  
    inkCanvas1.EditingMode = InkCanvasEditingMode.Ink;  
  
    // Set the DefaultDrawingAttributes for a highlighter pen.  
    inkCanvas1.DefaultDrawingAttributes.Color = Colors.Yellow;  
    inkCanvas1.DefaultDrawingAttributes.IsHighlighter = true;  
    inkCanvas1.DefaultDrawingAttributes.Height = 25;  
}  
  
// Set the EditingMode to erase by stroke.  
private void EraseStroke(object sender, RoutedEventArgs e)  
{  
    inkCanvas1.EditingMode = InkCanvasEditingMode.EraseByStroke;  
}  
  
// Set the EditingMode to selection.  
private void Select(object sender, RoutedEventArgs e)  
{  
    inkCanvas1.EditingMode = InkCanvasEditingMode.Select;  
}
```

复制此代码后，在 Visual Studio 中按 F5，以在调试器中运行该程序。

请注意 StackPanel 是如何将按钮置于 InkCanvas 顶部的。如果尝试在这些按钮顶部使用墨迹，InkCanvas 将收集墨迹并在按钮后面呈现墨迹。这是因为这些按钮是 InkCanvas 的同级，而不是子级。此外，这些按钮的 Z 顺序较高，所以墨迹呈现在其后面。

## 另请参阅

- [DrawingAttributes](#)
- [DefaultDrawingAttributes](#)

- System.Windows.Ink

# 手写识别

项目 · 2022/09/27

本节介绍了识别基础知识，因为这与 WPF 平台中数字墨迹有关。

## 识别解决方案

以下示例演示如何使用 [Microsoft.Ink.InkCollector](#) 类识别墨迹。

### ① 备注

此示例要求在系统上安装手写识别器。

在 Visual Studio 中创建一个名为 `InkRecognition` 的新 WPF 应用程序项目。用下列 XAML 代码替换 `Window1.xaml` 文件的内容。此代码呈现应用程序的用户界面。

XAML

```
<Window x:Class="InkRecognition.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="InkRecognition"
    >
<Canvas Name="theRootCanvas">
    <Border
        Background="White"
        BorderBrush="Black"
        BorderThickness="2"
        Height="300"
        Width="300"
        Canvas.Top="10"
        Canvas.Left="10">
        <InkCanvas Name="theInkCanvas"></InkCanvas>
    </Border>
    <TextBox Name="textBox1"
        Height="25"
        Width="225"
        Canvas.Top="325"
        Canvas.Left="10"></TextBox>
    <Button
        Height="25"
        Width="75"
        Canvas.Top="325"
        Canvas.Left="235"
        Click="buttonClick">Recognize</Button>
    <Button x:Name="btnClear" Content="Clear Canvas" Canvas.Left="10"
        Canvas.Top="367" Width="75" Click="btnClear_Click"/>
```

```
</Canvas>
</Window>
```

添加对 Microsoft 墨迹程序集和 Microsoft.Ink.dll 的引用，可在 \Program Files\Common Files\Microsoft Shared\Ink 中找到这些内容。用下列代码替换代码隐藏文件的内容。

C#

```
using System.Windows;
using Microsoft.Ink;
using System.IO;

namespace InkRecognition
{
    /// <summary>
    /// Interaction logic for Window1.xaml
    /// </summary>

    public partial class Window1 : Window
    {

        public Window1()
        {
            InitializeComponent();
        }

        // Recognizes handwriting by using RecognizerContext
        private void buttonClick(object sender, RoutedEventArgs e)
        {
            using (MemoryStream ms = new MemoryStream())
            {
                theInkCanvas.Strokes.Save(ms);
                var myInkCollector = new InkCollector();
                var ink = new Ink();
                ink.Load(ms.ToArray());

                using (RecognizerContext context = new RecognizerContext())
                {
                    if (ink.Strokes.Count > 0)
                    {
                        context.Strokes = ink.Strokes;
                        RecognitionStatus status;

                        var result = context.Recognize(out status);

                        if (status == RecognitionStatus.NoError)
                            textBox1.Text = result.TopString;
                        else
                            MessageBox.Show("Recognition failed");
                    }
                    else
                    {
                        MessageBox.Show("No stroke detected");
                    }
                }
            }
        }
    }
}
```

```
        }
    }

    private void btnClear_Click(object sender, RoutedEventArgs e) {
        theInkCanvas.Strokes.Clear();
    }
}
```

## 另请参阅

- [Microsoft.Ink.InkCollector](#)

# 存储墨迹

项目 • 2023/02/06

[Save](#) 方法支持将墨迹存储为墨迹序列化格式 (ISF)。 [StrokeCollection](#) 类的构造函数为读取墨迹数据提供支持。

## 墨迹存储和检索

本部分讨论如何在 WPF 平台中存储和检索墨迹。

以下示例实现一个按钮单击事件处理程序，它向用户显示“文件保存”对话框，并从 [InkCanvas](#) 输出墨迹保存到文件中。

C#

```
private void buttonSaveAsClick(object sender, RoutedEventArgs e)
{
    SaveFileDialog saveFileDialog1 = new SaveFileDialog();
    saveFileDialog1.Filter = "isf files (*.isf)|*.isf";

    if (saveFileDialog1.ShowDialog() == true)
    {
        FileStream fs = new FileStream(saveFileDialog1.FileName,
                                         FileMode.Create);
        theInkCanvas.Strokes.Save(fs);
        fs.Close();
    }
}
```

以下示例实现一个按钮单击事件处理程序，它向用户显示“文件打开”对话框，并读取来自文件的墨迹到 [InkCanvas](#) 元素中。

C#

```
private void buttonLoadClick(object sender, RoutedEventArgs e)
{
    OpenFileDialog openFileDialog1 = new OpenFileDialog();
    openFileDialog1.Filter = "isf files (*.isf)|*.isf";

    if (openFileDialog1.ShowDialog() == true)
    {
        FileStream fs = new FileStream(openFileDialog1.FileName,
                                         FileMode.Open);
        theInkCanvas.Strokes = new StrokeCollection(fs);
        fs.Close();
    }
}
```

---

## 另请参阅

- [InkCanvas](#)
- [Windows Presentation Foundation](#)

# 墨迹对象模型：Windows 窗体和 COM 与 WPF

项目 • 2022/09/27

基本上有三个平台支持数字墨迹：Tablet PC Windows 窗体平台、Tablet PC COM 平台和 Windows Presentation Foundation (WPF) 平台。Windows 窗体和 COM 平台具有类似的对象模型，但 WPF 平台的对象模型大不相同。本主题简要讨论存在的差异，以便使用过某一种对象模型的开发人员能够更好地了解另一种对象模型。

## 在应用程序中启用墨迹

这三个平台都提供对象和控件，使应用程序能够接收来自触笔的输入。Windows 窗体和 COM 平台附带 [Microsoft.Ink.InkPicture](#)、[Microsoft.Ink.InkEdit](#)、[Microsoft.Ink.InkOverlay](#) 和 [Microsoft.Ink.InkCollector](#) 类。[Microsoft.Ink.InkPicture](#) 和 [Microsoft.Ink.InkEdit](#) 是可添加到应用程序以收集墨迹的控件。[Microsoft.Ink.InkOverlay](#) 和 [Microsoft.Ink.InkCollector](#) 可以附加到现有窗口、启用了墨迹的窗口和自定义控件。

WPF 平台包括 [InkCanvas](#) 控件。可以向应用程序添加一个 [InkCanvas](#)，然后立即开始收集墨迹。有了 [InkCanvas](#)，用户可以复制、选择墨迹并调整其大小。你可向 [InkCanvas](#) 添加其他控件，用户也可在这些控件上手写。可以通过向其添加 [InkPresenter](#) 并收集其触笔点来创建启用了墨迹的自定义控件。

下表列出了在哪里可以详细了解如何在应用程序中启用墨迹：

要实现的目的...	在 WPF 平台上...	在 Windows 窗体/COM 平台上...
向应用程序添加启用了墨迹的控件	请参阅 <a href="#">墨迹入门</a> 。	请参阅 <a href="#">自动声明窗体示例</a>
在自定义控件上启用墨迹	请参阅 <a href="#">创建墨迹输入控件</a> 。	请参阅 <a href="#">墨迹剪贴板示例</a> 。

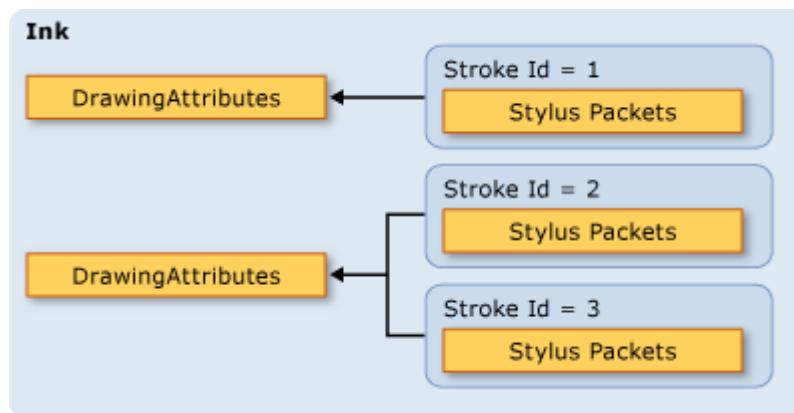
## 墨迹数据

在 Windows 窗体和 COM 平台上，[Microsoft.Ink.InkCollector](#)、[Microsoft.Ink.InkOverlay](#)、[Microsoft.Ink.InkEdit](#) 和 [Microsoft.Ink.InkPicture](#) 都会公开 [Microsoft.Ink.Ink](#) 对象。[Microsoft.Ink.Ink](#) 对象包含一个或多个 [Microsoft.Ink.Stroke](#) 对象的数据，并公开用于管理和操作这些笔划的常用方法和属性。[Microsoft.Ink.Ink](#) 对象管理

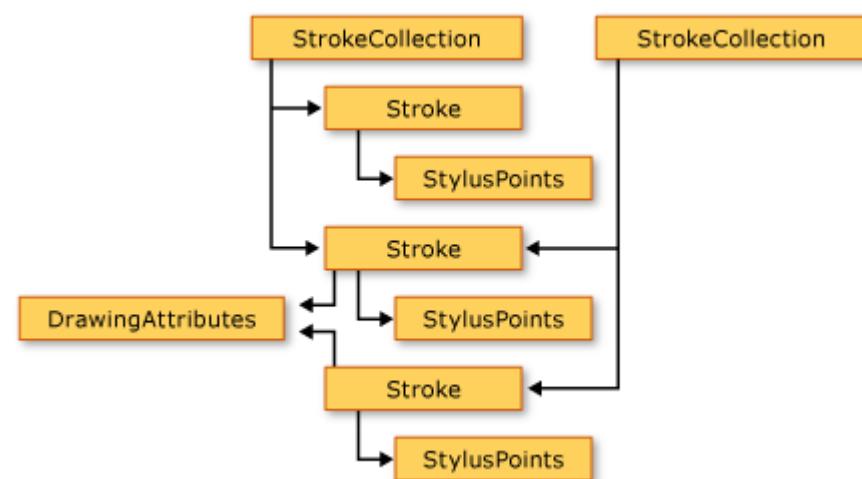
其包含的笔划的生存期；[Microsoft.Ink.Ink](#) 对象创建并删除其拥有的笔划。每个 [Microsoft.Ink.Stroke](#) 都有一个在其父 [Microsoft.Ink.Ink](#) 对象中唯一的标识符。

在 WPF 平台上，[System.Windows.Ink.Stroke](#) 类拥有并管理自己的生存期。一组 [Stroke](#) 对象可以一起收集在 [StrokeCollection](#) 中，它提供常见的墨迹数据管理操作方法，例如命中测试、擦除、转换和序列化墨迹。在任何给定时间，[Stroke](#) 都可以属于零个、一个或多个 [StrokeCollection](#) 对象。[InkCanvas](#) 和 [InkPresenter](#) 包含的不是 [Microsoft.Ink.Ink](#) 对象，而是 [System.Windows.Ink.StrokeCollection](#)。

下面这对插图比较了墨迹数据对象模型。在 Windows 窗体和 COM 平台上，[Microsoft.Ink.Ink](#) 对象约束 [Microsoft.Ink.Stroke](#) 对象的生存期，而触笔数据包属于各个笔划。两个或更多笔划可以引用相同的 [Microsoft.Ink.DrawingAttributes](#) 对象，如下图所示。



在 WPF 上，每个 [System.Windows.Ink.Stroke](#) 都是一个公共语言运行时对象，只要某些内容引用了该对象，该对象就会存在。每个 [Stroke](#) 引用一个 [StylusPointCollection](#) 和 [System.Windows.Ink.DrawingAttributes](#) 对象，这些对象也是公共语言运行时对象。



下表比较了如何在 WPF 平台以及 Windows 窗体和 COM 平台上完成一些常见任务。

任务	Windows Presentation Foundation	Windows 窗体和 COM
----	---------------------------------	-----------------

任务	Windows Presentation Foundation	Windows 窗体和 COM
保存墨迹	Save	Microsoft.Ink.Ink.Save
加载墨迹	使用 <a href="#">StrokeCollection</a> 构造函数创建一个 <a href="#">StrokeCollection</a> 。	Microsoft.Ink.Ink.Load
命中测试	HitTest	Microsoft.Ink.Ink.HitTest
复制墨迹	CopySelection	Microsoft.Ink.Ink.ClipboardCopy
粘贴墨迹	Paste	Microsoft.Ink.Ink.ClipboardPaste
访问笔划集合的自定义属性	AddPropertyData ( 这些属性存储在内部 , 通过 AddPropertyData 、 RemovePropertyData 和 ContainsPropertyData 访问 )	使用 Microsoft.Ink.Ink.ExtendedProperties

## 在平台之间共享墨迹

尽管这些平台对墨迹数据具有不同的对象模型 , 但在平台之间共享墨迹共享数据非常简单。以下示例保存 Windows 窗体应用程序中的墨迹 , 并将墨迹加载到 Windows Presentation Foundation 应用程序中。

C#

```
using Microsoft.Ink;
using System.Drawing;
```

C#

```
/// <summary>
/// Saves the digital ink from a Windows Forms application.
/// </summary>
/// <param name="inkToSave">An Ink object that contains the
/// digital ink.</param>
/// <returns>A MemoryStream containing the digital ink.</returns>
MemoryStream SaveInkInWinforms(Ink inkToSave)
{
    byte[] savedInk = inkToSave.Save();

    return (new MemoryStream(savedInk));
}
```

C#

```
using System.Windows.Ink;
```

C#

```
/// <summary>
/// Loads digital ink into a StrokeCollection, which can be
/// used by a WPF application.
/// </summary>
/// <param name="savedInk">A MemoryStream containing the digital ink.
</param>
public void LoadInkInWPF(MemoryStream inkStream)
{
    strokes = new StrokeCollection(inkStream);
}
```

以下示例保存 Windows Presentation Foundation 应用程序中的墨迹，并将墨迹加载到 Windows窗体应用程序中。

C#

```
using System.Windows.Ink;
```

C#

```
/// <summary>
/// Saves the digital ink from a WPF application.
/// </summary>
/// <param name="inkToSave">A StrokeCollection that contains the
/// digital ink.</param>
/// <returns>A MemoryStream containing the digital ink.</returns>
MemoryStream SaveInkInWPF(StrokeCollection strokesToSave)
{
    MemoryStream savedInk = new MemoryStream();

    strokesToSave.Save(savedInk);

    return savedInk;
}
```

C#

```
using Microsoft.Ink;
using System.Drawing;
```

C#

```
/// <summary>
/// Loads digital ink into a Windows Forms application.
/// </summary>
/// <param name="savedInk">A MemoryStream containing the digital ink.
</param>
public void LoadInkInWinforms(MemoryStream savedInk)
{
    theInk = new Ink();
    theInk.Load(savedInk.ToArray());
}
```

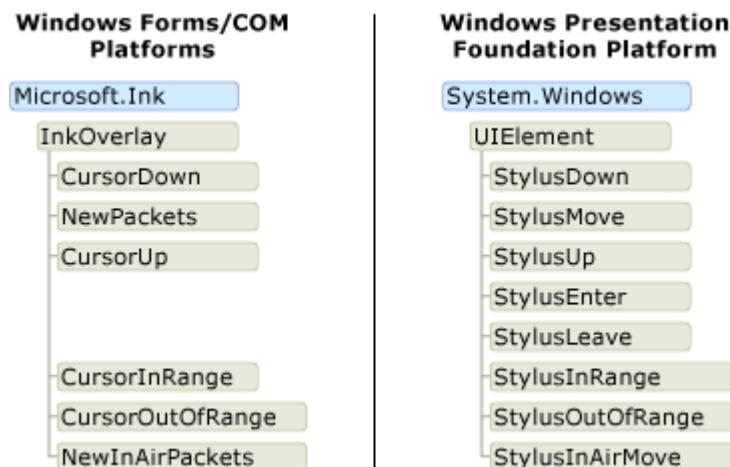
## 来自触笔的事件

当用户输入触笔数据时，Windows 窗体和 COM 平台上的 [Microsoft.Ink.InkOverlay](#)、[Microsoft.Ink.InkCollector](#) 和 [Microsoft.Ink.InkPicture](#) 接收事件。

[Microsoft.Ink.InkOverlay](#) 或 [Microsoft.Ink.InkCollector](#) 附加到窗口或控件，并且可以订阅触笔输入数据引发的事件。发生这些事件的线程取决于这些事件是用笔、鼠标还是以编程方式引发的。有关与这些事件相关的线程处理的详细信息，请参阅[常规线程处理注意事项和可以触发事件的线程](#)。

在 Windows Presentation Foundation 平台上，[UIElement](#) 类包含触笔输入的事件。这意味着每个控件都会公开完整的触笔事件集。触笔事件具有隧道/浮升事件对，并且始终出现在应用程序线程上。有关详细信息，请参阅[路由事件概述](#)。

下图比较了引发触笔事件的类的对象模型。Windows Presentation Foundation 对象模型仅显示浮升事件，而不显示隧道事件对应项。



## 触笔数据

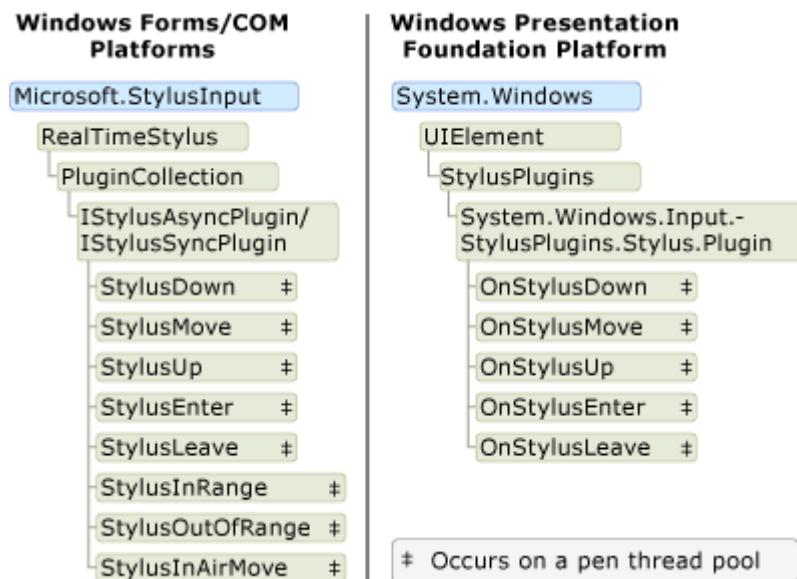
这三个平台都为你提供了截获和操作来自触笔的数据的方法。在 Windows 窗体和 COM 平台上，通过创建 [Microsoft.StylusInput.RealTimeStylus](#)，向其附加窗口或控件并创建实

现 `Microsoft.StylusInput.IStylusSyncPlugin` 或 `Microsoft.StylusInput.IStylusAsyncPlugin` 接口的类来实现此目的。然后将自定义插件添加到 `Microsoft.StylusInput.RealTimeStylus` 的插件集合中。有关此对象模型的详细信息，请参阅 [StylusInput API 的体系结构](#)。

在 WPF 平台上，`UIElement` 类公开一个插件集合，类似于 `Microsoft.StylusInput.RealTimeStylus` 的设计。若要截获笔数据，请创建一个继承自 `StylusPlugIn` 的类，并将对象添加到 `UIElement` 的 `StylusPlugins` 集合中。有关此交互的详细信息，请参阅[截获触笔的输入](#)。

在所有平台上，线程池通过触笔事件接收墨迹数据，并将其发送到应用程序线程。有关 COM 和 Windows 平台上的线程处理的详细信息，请参阅 [StylusInput API 的线程处理注意事项](#)。有关 Windows Presentation 软件上的线程处理的详细信息，请参阅[墨迹线程模型](#)。

下图比较了在触笔线程池上接收触笔数据的类的对象模型。



# 高级墨迹处理

项目 • 2023/02/06

WPF 附带了 [InkCanvas](#)，可以将其放入应用程序中以立即开始收集和显示墨迹。但是，如果 [InkCanvas](#) 控件没有提供足够精细的控件级别，可以通过使用 [System.Windows.Input.StylusPlugIns](#) 自定义自己的墨迹集合和墨迹呈现，以更高级别对控件进行维护。

[System.Windows.Input.StylusPlugIns](#) 类提供了一种机制，用于实现对 [Stylus](#) 输入和动态呈现墨迹的低级别控制。[StylusPlugIn](#) 类提供了一种机制，用于实现自定义行为并将其应用于来自触笔设备的数据流以获得最佳性能。[DynamicRenderer](#) 是一个专门的 [StylusPlugIn](#)，可以通过它自定义实时动态呈现墨迹数据，这意味着 [DynamicRenderer](#) 会在生成 [StylusPoint](#) 数据时立即绘制数字墨迹，所以看起来像是从触笔设备“流出”一样。

## 本节内容

[自定义呈现墨迹](#)

[截获触笔输入](#)

[创建墨迹输入控件](#)

[墨迹线程处理模型](#)

# 自定义呈现墨迹

项目 · 2023/02/06

通过笔划的 [DrawingAttributes](#) 属性可指定笔划的外观，例如笔划的大小、颜色和形状，但有时你可能想要自定义 [DrawingAttributes](#) 不能实现的外观。建议通过使用喷笔、油画和多种其他效果呈现外观，从而自定义墨迹的外观。Windows Presentation Foundation (WPF) 允许通过实现自定义 [DynamicRenderer](#) 和 [Stroke](#) 对象来自定义呈现墨迹。

本主题包含以下小节：

- [体系结构](#)
- [实现动态呈现器](#)
- [实现自定义笔划](#)
- [实现自定义 InkCanvas](#)
- [结论](#)

## 体系结构

墨迹呈现会出现两次：用户将墨迹写入墨迹书写表面时，以及将笔划添加到启用了墨迹的表面之后。用户在数字化器上移动触笔时，[DynamicRenderer](#) 会呈现墨迹，并且将 [Stroke](#) 添加到元素后它会呈现自身。

动态呈现墨迹时需要实现三个类。

1. [DynamicRenderer](#)：实现派生自 [DynamicRenderer](#) 的类。此类是专用的 [StylusPlugIn](#)，可按绘制的原本形式呈现笔划。[DynamicRenderer](#) 在一个单独线程上执行呈现，因此即使在应用程序用户界面 (UI) 线程被阻止时也会出现墨迹书写表面来收集墨迹。有关线程模型的详细信息，请参阅[墨迹线程模型](#)。若要自定义动态呈现笔划，请重写 [OnDraw](#) 方法。
2. **笔划**：实现派生自 [Stroke](#) 的类。此类负责在 [StylusPoint](#) 数据被转换为 [Stroke](#) 对象后静态呈现该数据。重写 [DrawCore](#) 方法是为了确保笔划的静态呈现与动态呈现一致。
3. [InkCanvas](#)：实现派生自 [InkCanvas](#) 的类。将自定义的 [DynamicRenderer](#) 分配给 [DynamicRenderer](#) 属性。重写 [OnStrokeCollected](#) 方法并将自定义笔划添加到 [Strokes](#) 属性。这样可以确保墨迹外观一致。

# 实现动态呈现器

尽管 [DynamicRenderer](#) 类是 WPF 的标准组成部分，但是若要执行更专业的呈现，必须创建派生自 [DynamicRenderer](#) 的自定义动态呈现器，并重写 [OnDraw](#) 方法。

以下示例演示可绘制具有线性渐变画笔效果的自定义的 [DynamicRenderer](#) 墨迹。

C#

```
using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;
```

C#

```
// A StylusPlugin that renders ink with a linear gradient brush effect.
class CustomDynamicRenderer : DynamicRenderer
{
    [ThreadStatic]
    static private Brush brush = null;

    [ThreadStatic]
    static private Pen pen = null;

    private Point prevPoint;

    protected override void OnStylusDown(RawStylusInput rawStylusInput)
    {
        // Allocate memory to store the previous point to draw from.
        prevPoint = new Point(double.NegativeInfinity,
double.NegativeInfinity);
        base.OnStylusDown(rawStylusInput);
    }

    protected override void OnDraw(DrawingContext drawingContext,
                                  StylusPointCollection stylusPoints,
                                  Geometry geometry, Brush fillBrush)
    {
        // Create a new Brush, if necessary.
        brush ??= new LinearGradientBrush(Colors.Red, Colors.Blue, 20d);

        // Create a new Pen, if necessary.
        pen ??= new Pen(brush, 2d);

        // Draw linear gradient ellipses between
        // all the StylusPoints that have come in.
        for (int i = 0; i < stylusPoints.Count; i++)
        {
```

```

        Point pt = (Point)stylusPoints[i];
        Vector v = Point.Subtract(prevPoint, pt);

        // Only draw if we are at least 4 units away
        // from the end of the last ellipse. Otherwise,
        // we're just redrawing and wasting cycles.
        if (v.Length > 4)
        {
            // Set the thickness of the stroke based
            // on how hard the user pressed.
            double radius = stylusPoints[i].PressureFactor * 10d;
            drawingContext.DrawEllipse(brush, pen, pt, radius, radius);
            prevPoint = pt;
        }
    }
}

```

## 实现自定义笔划

实现从 [Stroke](#) 派生的类。此类负责在 [StylusPoint](#) 数据被转换为 [Stroke](#) 对象后呈现该数据。重写 [DrawCore](#) 类进行实际绘制。

通过使用 [AddPropertyData](#) 方法，笔划类还可存储自定义数据。此数据持续存在时会与笔划数据一起存储。

[Stroke](#) 类还可执行命中测试。也可通过重写当前类中的 [HitTest](#) 方法实现自己的命中测试算法。

以下 C# 代码演示了将 [StylusPoint](#) 数据呈现为三维笔划的自定义 [Stroke](#) 类。

C#

```

using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;

```

C#

```

// A class for rendering custom strokes
class CustomStroke : Stroke
{
    Brush brush;
    Pen pen;

    public CustomStroke(StylusPointCollection stylusPoints)

```

```

    : base(stylusPoints)
{
    // Create the Brush and Pen used for drawing.
    brush = new LinearGradientBrush(Colors.Red, Colors.Blue, 20d);
    pen = new Pen(brush, 2d);
}

protected override void DrawCore(DrawingContext drawingContext,
                                DrawingAttributes drawingAttributes)
{
    // Allocate memory to store the previous point to draw from.
    Point prevPoint = new Point(double.NegativeInfinity,
                                double.NegativeInfinity);

    // Draw linear gradient ellipses between
    // all the StylusPoints in the Stroke.
    for (int i = 0; i < this.StylusPoints.Count; i++)
    {
        Point pt = (Point)this.StylusPoints[i];
        Vector v = Point.Subtract(prevPoint, pt);

        // Only draw if we are at least 4 units away
        // from the end of the last ellipse. Otherwise,
        // we're just redrawing and wasting cycles.
        if (v.Length > 4)
        {
            // Set the thickness of the stroke
            // based on how hard the user pressed.
            double radius = this.StylusPoints[i].PressureFactor * 10d;
            drawingContext.DrawEllipse(brush, pen, pt, radius, radius);
            prevPoint = pt;
        }
    }
}
}

```

## 实现自定义 InkCanvas

使用自定义 [DynamicRenderer](#) 和笔划最简单的方式是实现派生自 [InkCanvas](#) 的类并使用这些类。[InkCanvas](#) 具有 [DynamicRenderer](#) 属性，该属性可以指定用户绘制笔划时笔划的呈现方式。

若要在 [InkCanvas](#) 上呈现笔划，请执行以下操作：

- 创建一个派生自 [InkCanvas](#) 的类。
- 将自定义的 [DynamicRenderer](#) 分配给 [InkCanvas.DynamicRenderer](#) 属性。
- 重写 [OnStrokeCollected](#) 方法。使用此方法时，请删除之前添加到 [InkCanvas](#) 的原始笔划。然后创建一个自定义笔划，将其添加到 [Strokes](#) 属性，然后使用新的

[InkCanvasStrokeCollectedEventArgs](#) ( 其中包含此自定义笔划 ) 调用基类。

以下 C# 代码演示了一个自定义 [InkCanvas](#) 类，该类使用了自定义的 [DynamicRenderer](#) 并收集自定义笔划。

C#

```
public class CustomRenderingInkCanvas : InkCanvas
{
    CustomDynamicRenderer customRenderer = new CustomDynamicRenderer();

    public CustomRenderingInkCanvas() : base()
    {
        // Use the custom dynamic renderer on the
        // custom InkCanvas.
        this.DynamicRenderer = customRenderer;
    }

    protected override void
    OnStrokeCollected(InkCanvasStrokeCollectedEventArgs e)
    {
        // Remove the original stroke and add a custom stroke.
        this.Strokes.Remove(e.Stroke);
        CustomStroke customStroke = new CustomStroke(e.Stroke.StylusPoints);
        this.Strokes.Add(customStroke);

        // Pass the custom stroke to base class' OnStrokeCollected method.
        InkCanvasStrokeCollectedEventArgs args =
            new InkCanvasStrokeCollectedEventArgs(customStroke);
        base.OnStrokeCollected(args);
    }
}
```

一个 [InkCanvas](#) 可以有多个 [DynamicRenderer](#)。可以通过将多个 [DynamicRenderer](#) 对象添加到 [StylusPlugIns](#) 属性来将其添加到 [InkCanvas](#)。

## 结论

可以通过派生自己的 [DynamicRenderer](#)、[Stroke](#) 和 [InkCanvas](#) 类来自定义墨迹的外观。将这些类结合使用可以确保用户绘制笔划时和笔划被收集后的笔划外观保持一致。

## 另请参阅

- [高级墨迹处理](#)

# 截获触笔输入

项目 · 2022/10/19

[System.Windows.Input.StylusPlugIns](#) 体系结构提供了一种机制，用于对 [Stylus](#) 输入和数字墨迹 [Stroke](#) 对象的创建实现低级别控制。[StylusPlugIn](#) 类提供了一种机制，用于实现自定义行为并将其应用于来自触笔设备的数据流以获得最佳性能。

本主题包含以下小节：

- [体系结构](#)
- [实现触笔插件](#)
- [将插件添加到 InkCanvas](#)
- [结论](#)

## 体系结构

[StylusPlugIn](#) 是 [StylusInput API](#) 的演化结果，如[访问和处理触笔输入](#)中所述。

每个 [UIElement](#) 的 [StylusPlugIns](#) 属性都是 [StylusPlugInCollection](#)。可以向元素的 [StylusPlugIns](#) 属性添加 [StylusPlugIn](#)，以在生成 [StylusPoint](#) 数据时操作该数据。[StylusPoint](#) 数据包括系统数字化器支持的所有属性，包括 [X](#) 和 [Y](#) 点数据以及 [PressureFactor](#) 数据。

当您向 [StylusPlugIns](#) 属性添加 [StylusPlugIn](#) 时，[StylusPlugIn](#) 对象会直接插入来自 [Stylus](#) 设备的数据流中。插件添加到 [StylusPlugIns](#) 集合的顺序决定了它们接收 [StylusPoint](#) 数据的顺序。例如，如果添加一个将输入限制为特定区域的筛选器插件，然后添加一个在写入内容时识别手势的插件，则识别手势的插件将收到筛选的 [StylusPoint](#) 数据。

## 实现触笔插件

若要实现插件，请从 [StylusPlugIn](#) 中派生类。此类应用于数据流，因为它来自 [Stylus](#)。在此类中，可以修改 [StylusPoint](#) 数据的值。

### ⊗ 注意

如果 [StylusPlugIn](#) 引发或导致异常，应用程序将关闭。如果确定 [StylusPlugIn](#) 不会引发异常，则应彻底测试使用 [StylusPlugIn](#) 的控件且仅使用控件。

以下示例演示了一个插件，该插件可以在 StylusPoint 数据从 Stylus 设备传入时修改该数据中的 X 和 Y 值，从而限制触笔输入。

C#

```
using System;
using System.Windows.Media;
using System.Windows;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Input;
using System.Windows.Ink;
```

C#

```
// A StylusPlugin that restricts the input area.
class FilterPlugin : StylusPlugIn
{
    protected override void OnStylusDown(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusDown(rawStylusInput);

        // Restrict the stylus input.
        Filter(rawStylusInput);
    }

    protected override void OnStylusMove(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusMove(rawStylusInput);

        // Restrict the stylus input.
        Filter(rawStylusInput);
    }

    protected override void OnStylusUp(RawStylusInput rawStylusInput)
    {
        // Call the base class before modifying the data.
        base.OnStylusUp(rawStylusInput);

        // Restrict the stylus input
        Filter(rawStylusInput);
    }

    private void Filter(RawStylusInput rawStylusInput)
    {
        // Get the StylusPoints that have come in.
        StylusPointCollection stylusPoints =
rawStylusInput.GetStylusPoints();

        // Modify the (X,Y) data to move the points
        // inside the acceptable input area, if necessary.
    }
}
```

```

        for (int i = 0; i < stylusPoints.Count; i++)
        {
            StylusPoint sp = stylusPoints[i];
            if (sp.X < 50) sp.X = 50;
            if (sp.X > 250) sp.X = 250;
            if (sp.Y < 50) sp.Y = 50;
            if (sp.Y > 250) sp.Y = 250;
            stylusPoints[i] = sp;
        }

        // Copy the modified StylusPoints back to the RawStylusInput.
        rawStylusInput.SetStylusPoints(stylusPoints);
    }
}

```

## 将插件添加到 InkCanvas

使用自定义插件的最简单方法是实现派生自 InkCanvas 的类并将其添加到 [StylusPlugIns](#) 属性。

以下示例演示筛选墨迹的自定义 [InkCanvas](#)。

C#

```

public class FilterInkCanvas : InkCanvas
{
    FilterPlugin filter = new FilterPlugin();

    public FilterInkCanvas()
        : base()
    {
        this.StylusPlugIns.Add(filter);
    }
}

```

如果向应用程序添加一个 `FilterInkCanvas` 并运行它，你会注意到，在用户完成笔划之前，墨迹不会限制到某个区域。这是因为 `InkCanvas` 的 `DynamicRenderer` 属性是 `StylusPlugIn` 并且已经是 `StylusPlugIns` 集合的一个成员。添加到 `StylusPlugIns` 集合的自定义 `StylusPlugIn` 在 `DynamicRenderer` 接收数据后接收 `StylusPoint` 数据。因此，只有在用户抬起笔以结束笔划后才会筛选 `StylusPoint` 数据。若要在用户绘制墨迹时筛选墨迹，必须在 `DynamicRenderer` 之前插入 `FilterPlugin`。

以下 C# 代码演示了一个自定义 `InkCanvas`，用于在绘制墨迹时筛选墨迹。

C#

```
public class DynamicallyFilteredInkCanvas : InkCanvas
{
    FilterPlugin filter = new FilterPlugin();

    public DynamicallyFilteredInkCanvas()
        : base()
    {
        int dynamicRenderIndex =
            this.StylusPlugIns.IndexOf(this.DynamicRenderer);

        this.StylusPlugIns.Insert(dynamicRenderIndex, filter);
    }
}
```

## 结论

通过派生自己的 `StylusPlugin` 类并将其插入到 `StylusPlugInCollection` 集合中，可以极大地增强数字墨迹的行为。你可以在 `StylusPoint` 数据生成时访问该数据，借此机会自定义 `Stylus` 输入。由于你对 `StylusPoint` 数据的访问权限非常低，因此可以使用应用程序的最佳性能实现墨迹收集和呈现。

## 另请参阅

- [高级墨迹处理](#)
- [访问和处理触笔输入](#)

# 创建墨迹输入控件

项目 · 2022/09/27

可以创建动态和静态呈现墨迹的自定义控件。也就是说，在用户绘制笔划时呈现墨迹，使墨迹看起来像是从触笔中“流出”的一样，并在将墨迹添加到控件（通过触笔、从剪贴板粘贴或从文件加载）后显示墨迹。若要动态呈现墨迹，控件必须使用 [DynamicRenderer](#)。要静态呈现墨迹，必须重写触笔事件方法（[OnStylusDown](#)、[OnStylusMove](#) 和 [OnStylusUp](#)）以收集 [StylusPoint](#) 数据、创建笔触并将它们添加到 [InkPresenter](#)（用于在控件上呈现墨迹）。

本主题包含以下小节：

- [如何：收集触笔点数据并创建墨迹笔划](#)
- [如何：使控件能够接受鼠标输入](#)
- [综合运用](#)
- [使用其他插件和 DynamicRenderers](#)
- [结论](#)

## 如何：收集触笔点数据并创建墨迹笔划

若要创建收集和管理墨迹笔划的控件，请执行以下操作：

1. 从 [Control](#) 或派生自 [Control](#)（例如 [Label](#)）的类之一派生类。

C#

```
using System;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Controls;
using System.Windows;
```

C#

```
class InkControl : Label
{
```

C#

```
}
```

2. 向类添加 [InkPresenter](#)，并将 [Content](#) 属性设置为新的 [InkPresenter](#)。

```
C#
```

```
InkPresenter ip;  
  
public InkControl()  
{  
    // Add an InkPresenter for drawing.  
    ip = new InkPresenter();  
    this.Content = ip;  
}
```

3. 通过调用 [AttachVisuals](#) 方法将 [DynamicRenderer](#) 的 [RootVisual](#) 附加到 [InkPresenter](#)，并将 [DynamicRenderer](#) 添加到 [StylusPlugIns](#) 集合。这允许 [InkPresenter](#) 在控件收集触点数据时显示墨迹。

```
C#
```

```
public InkControl()  
{  
  
    // Add a dynamic renderer that  
    // draws ink as it "flows" from the stylus.  
    dr = new DynamicRenderer();  
    ip.AttachVisuals(dr.RootVisual, dr.DrawingAttributes);  
    this.StylusPlugIns.Add(dr);  
}
```

4. 重写 [OnStylusDown](#) 方法。在此方法中，通过调用 [Capture](#) 来捕获触笔。通过捕获触笔，即使触笔离开控件的边界，控件也会继续接收 [StylusMove](#) 和 [StylusUp](#) 事件。这不是严格强制的，但若要获得良好的用户体验，则始终需要这样做。创建新的 [StylusPointCollection](#)，以收集 [StylusPoint](#) 数据。最后，将初始 [StylusPoint](#) 数据集添加到 [StylusPointCollection](#)。

```
C#
```

```
protected override void OnStylusDown(StylusDownEventArgs e)  
{  
    // Capture the stylus so all stylus input is routed to this  
    // control.  
    Stylus.Capture(this);
```

```
// Allocate memory for the StylusPointsCollection and
// add the StylusPoints that have come in so far.
stylusPoints = new StylusPointCollection();
StylusPointCollection eventPoints =
    e.GetStylusPoints(this, stylusPoints.Description);

stylusPoints.Add(eventPoints);
}
```

5. 重写 [OnStylusMove](#) 方法并将 [StylusPoint](#) 数据添加到之前创建的 [StylusPointCollection](#) 对象。

```
C#
```

```
protected override void OnStylusMove(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);
}
```

6. 重写 [OnStylusUp](#) 方法并使用 [StylusPointCollection](#) 数据创建一个新 [Stroke](#)。将创建的新 [Stroke](#) 添加到 [InkPresenter](#) 的 [Strokes](#) 集合，并释放触笔捕获。

```
C#
```

```
protected override void OnStylusUp(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);

    // Create a new stroke from all the StylusPoints since
    // OnStylusDown.
    Stroke stroke = new Stroke(stylusPoints);
```

```

    // Add the new stroke to the Strokes collection of the
    InkPresenter.
    ip.Strokes.Add(stroke);

    // Clear the StylusPointsCollection.
    stylusPoints = null;

    // Release stylus capture.
    Stylus.Capture(null);
}

```

## 如何：使控件能够接受鼠标输入

如果将上述控件添加到应用程序，运行它并使用鼠标作为输入设备，你会注意到笔划不会持久保留。要在将鼠标用作输入设备时保留笔画，请执行以下操作：

1. 重写 `OnMouseLeftButtonDown` 并创建一个新的 `StylusPointCollection`。获取事件发生时鼠标的位置，使用点数据创建一个 `StylusPoint`，并将 `StylusPoint` 添加到 `StylusPointCollection`。

```

C#

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Start collecting the points.
    stylusPoints = new StylusPointCollection();
    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

```

2. 重写 `OnMouseMove` 方法。获取事件发生时鼠标的位置，并使用点数据创建一个 `StylusPoint`。将 `StylusPoint` 添加到之前创建的 `StylusPointCollection` 对象。

```

C#

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);
}

```

```

// If a stylus generated this event, return.
if (e.StylusDevice != null)
{
    return;
}

// Don't collect points unless the left mouse button
// is down.
if (e.LeftButton == MouseButtonState.Released ||
    stylusPoints == null)
{
    return;
}

Point pt = e.GetPosition(this);
stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

```

3. 重写 `OnMouseLeftButtonUp` 方法。使用 `StylusPointCollection` 数据创建一个新 `Stroke`，并将创建的新 `Stroke` 添加到 `InkPresenter` 的 `Strokes` 集合。

C#

```

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{

    base.OnMouseLeftButtonUp(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    if (stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));

    // Create a stroke and add it to the InkPresenter.
    Stroke stroke = new Stroke(stylusPoints);
    stroke.DrawingAttributes = dr.DrawingAttributes;
    ip.Strokes.Add(stroke);

    stylusPoints = null;
}

```

# 综合运用

以下示例是在用户使用鼠标或笔时收集墨迹的自定义控件。

C#

```
using System;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Controls;
using System.Windows;
```

C#

```
// A control for managing ink input
class InkControl : Label
{
    InkPresenter ip;
    DynamicRenderer dr;

    // The StylusPointsCollection that gathers points
    // before Stroke from is created.
    StylusPointCollection stylusPoints = null;

    public InkControl()
    {
        // Add an InkPresenter for drawing.
        ip = new InkPresenter();
        this.Content = ip;

        // Add a dynamic renderer that
        // draws ink as it "flows" from the stylus.
        dr = new DynamicRenderer();
        ip.AttachVisuals(dr.RootVisual, dr.DrawingAttributes);
        this.StylusPlugIns.Add(dr);
    }

    static InkControl()
    {
        // Allow ink to be drawn only within the bounds of the control.
        Type owner = typeof(InkControl);
        ClipToBoundsProperty.OverrideMetadata(owner,
            new FrameworkPropertyMetadata(true));
    }

    protected override void OnStylusDown(StylusDownEventArgs e)
    {
        // Capture the stylus so all stylus input is routed to this control.
        Stylus.Capture(this);

        // Allocate memory for the StylusPointsCollection and
```

```

        // add the StylusPoints that have come in so far.
        stylusPoints = new StylusPointCollection();
        StylusPointCollection eventPoints =
            e.GetStylusPoints(this, stylusPoints.Description);

        stylusPoints.Add(eventPoints);
    }

protected override void OnStylusMove(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);
}

protected override void OnStylusUp(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    // Add the StylusPoints that have come in since the
    // last call to OnStylusMove.
    StylusPointCollection newStylusPoints =
        e.GetStylusPoints(this, stylusPoints.Description);
    stylusPoints.Add(newStylusPoints);

    // Create a new stroke from all the StylusPoints since OnStylusDown.
    Stroke stroke = new Stroke(stylusPoints);

    // Add the new stroke to the Strokes collection of the InkPresenter.
    ip.Strokes.Add(stroke);

    // Clear the StylusPointsCollection.
    stylusPoints = null;

    // Release stylus capture.
    Stylus.Capture(null);
}

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)

```

```
        {
            return;
        }

        // Start collecting the points.
        stylusPoints = new StylusPointCollection();
        Point pt = e.GetPosition(this);
        stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
    }

protected override void OnMouseMove(MouseEventArgs e)
{

    base.OnMouseMove(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    // Don't collect points unless the left mouse button
    // is down.
    if (e.LeftButton == MouseButtonState.Released ||
        stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));
}

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{

    base.OnMouseLeftButtonUp(e);

    // If a stylus generated this event, return.
    if (e.StylusDevice != null)
    {
        return;
    }

    if (stylusPoints == null)
    {
        return;
    }

    Point pt = e.GetPosition(this);
    stylusPoints.Add(new StylusPoint(pt.X, pt.Y));

    // Create a stroke and add it to the InkPresenter.
    Stroke stroke = new Stroke(stylusPoints);
    stroke.DrawingAttributes = dr.DrawingAttributes;
}
```

```
        ip.Strokes.Add(stroke);

        stylusPoints = null;
    }
}
```

## 使用其他插件和 DynamicRenderers

与 InkCanvas 一样，自定义控件可以具有自定义 StylusPlugIn 和其他 DynamicRenderer 对象。将这些对象添加到 StylusPlugIns 集合。StylusPlugInCollection 中 StylusPlugIn 对象的顺序会影响墨迹的呈现外观。假设你有一个名为 dynamicRenderer 的 DynamicRenderer 和一个名为 translatePlugin 的自定义 StylusPlugIn，用于偏移来自触笔的墨迹。如果 translatePlugin 是 StylusPlugInCollection 中的第一个 StylusPlugIn，而 dynamicRenderer 是第二个，则“流动”的墨迹将随着用户移动笔而偏移。如果 dynamicRenderer 是第一个，而 translatePlugin 是第二个，则在用户抬起笔之前，墨迹不会偏移。

## 结论

可以通过重写触笔事件方法来创建一个收集和呈现墨迹的控件。通过创建自己的控件、派生自己的 StylusPlugIn 类并将其插入 StylusPlugInCollection 中，可以实现几乎任何可使用数字墨迹想象的行为。你可以在 StylusPoint 数据生成时访问该数据，借此机会自定义 Stylus 输入并按照适合应用程序的方式在屏幕上呈现该输入。由于你对 StylusPoint 数据的访问权限非常低，因此可以使用应用程序的最佳性能实现墨迹收集和呈现。

## 另请参阅

- [高级墨迹处理](#)
- [访问和处理触笔输入](#)

# 墨迹线程处理模型

项目 · 2023/02/06

平板电脑上墨迹的好处之一是，它感觉很像使用常规的笔和纸张进行书写。为此，平板电脑笔收集输入数据的速率比鼠标高得多，并在用户书写时呈现墨迹。应用程序的用户界面 (UI) 线程不足以收集笔数据和呈现墨迹，因为它可能会被阻止。为了解决此问题，当用户写入墨迹时，WPF 应用程序会使用另外两个线程。

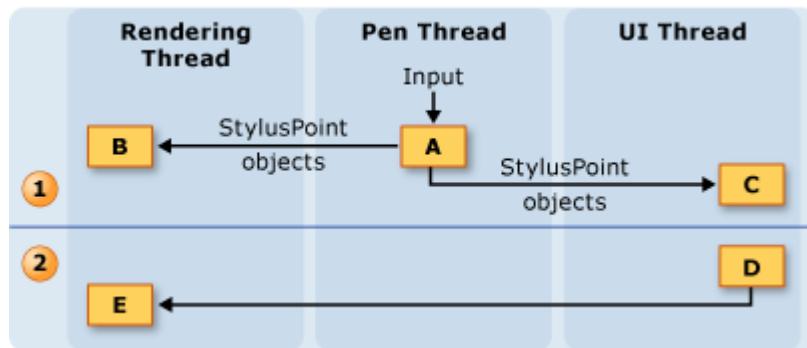
以下列表描述了参与收集和呈现数字墨迹的线程：

- 笔线程 - 从触笔获取输入的线程。（实际上，这是一个线程池，但本主题将其称为笔线程。）
- 应用程序用户界面线程 - 控制应用程序用户界面的线程。
- 动态呈现线程 - 当用户绘制笔划时呈现墨迹的线程。动态呈现线程不同于为应用程序呈现其他 UI 元素的线程，如 Window Presentation Foundation [线程模型](#) 中所述。

无论应用程序使用 [InkCanvas](#) 还是自定义控件（类似于[创建墨迹输入控件](#)中的控件），墨迹模型都是相同的。虽然本主题是从 [InkCanvas](#) 的角度讨论线程，但在创建自定义控件时也适用相同的概念。

## 线程概述

下图演示了用户绘制笔划时的线程模型：



### 1. 用户绘制笔划时发生的操作

- 当用户绘制笔划时，触笔点进入笔线程。包括 [DynamicRenderer](#) 在内的触笔插件接受笔线程上的触笔点，并有机会在 [InkCanvas](#) 接收到它们之前对其进行修改。
- [DynamicRenderer](#) 在动态呈现线程上呈现触笔点。这与上一步同时发生。

c. `InkCanvas` 接收 UI 线程上的触笔点。

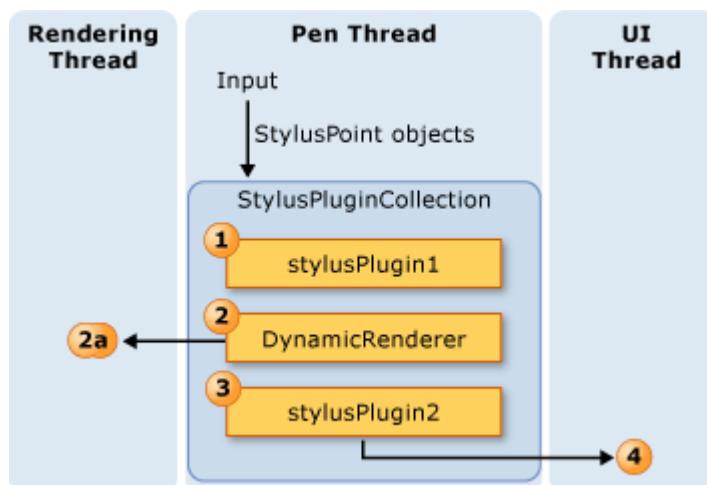
## 2. 用户结束笔划后发生的操作

- a. 当用户完成绘制笔划时，`InkCanvas` 创建一个 `Stroke` 对象并将其添加到 `InkPresenter` 中，后者静态地呈现它。
- b. UI 线程提醒 `DynamicRenderer` 笔画是静态呈现的，因此 `DynamicRenderer` 会删除笔划的可视表示形式。

## 墨迹集合和触笔插件

每个 `UIElement` 都有一个 `StylusPlugInCollection`。 `StylusPlugInCollection` 中的 `StylusPlugIn` 对象接收并可以修改笔线程上的触笔点。 `StylusPlugIn` 对象根据触笔点在 `StylusPlugInCollection` 中的顺序接收触笔点。

下面的图表说明了假设的情况，其中 `UIElement` 的 `StylusPlugins` 集合以该顺序包含 `stylusPlugin1`、`DynamicRenderer` 和 `stylusPlugin2`。



在上图中，发生了以下行为：

1. `StylusPlugin1` 修改 x 和 y 的值。
2. `DynamicRenderer` 接收修改后的触笔点，并在动态呈现线程上呈现它们。
3. `StylusPlugin2` 接收修改后的触笔点，并进一步修改 x 和 y 的值。
4. 应用程序收集触笔点，当用户完成笔划时，静态呈现笔划。

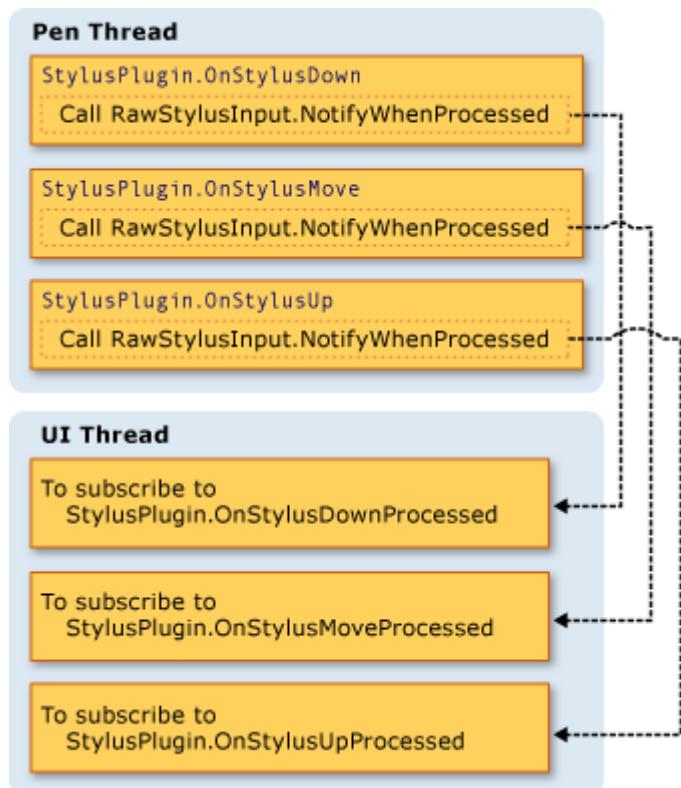
假设 `stylusPlugin1` 将触笔点限制为矩形，且 `stylusPlugin2` 将触笔点转换为右侧。在前面的场景中，`DynamicRenderer` 接受受限制的触笔点，但不接收经过转换的触笔点。当用户绘制笔划时，笔划将呈现在矩形的边界内，但在用户拿起笔之前，笔划似乎不会转换。

# 在 UI 线程上使用触笔插件执行操作

由于无法在笔线程上执行准确的命中测试，因此某些元素偶尔可能会接收用于其他元素的触笔输入。如果需要确保在执行操作之前正确路由输入，请订阅

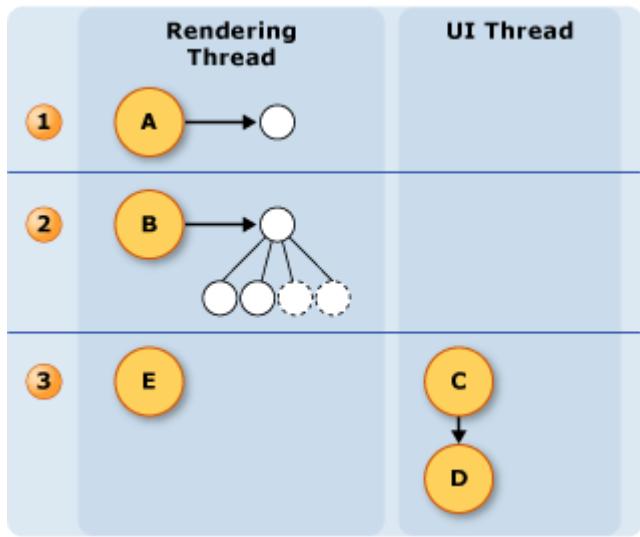
[OnStylusDownProcessed](#)、[OnStylusMoveProcessed](#) 或 [OnStylusUpProcessed](#) 方法并执行操作。这些方法由应用程序线程在执行准确的命中测试后调用。若要订阅这些方法，请调用发生在笔线程上的方法中的 [NotifyWhenProcessed](#) 方法。

下图说明了在 [StylusPlugIn](#) 的触笔事件方面笔线程和 UI 线程之间的关系。



## 呈现墨迹

当用户绘制笔划时，[DynamicRenderer](#) 将墨迹呈现在单独的线程上，这样即使在 UI 线程繁忙时，墨迹也会看上去从笔中“流”出来。当 [DynamicRenderer](#) 收集触笔点时，它在动态呈现线程上构建一个可视化树。当用户完成笔划时，[DynamicRenderer](#) 要求在应用程序执行下一个呈现通道时收到通知。应用程序完成下一个呈现通道后，[DynamicRenderer](#) 将清理其可视化树。下图演示了此过程。



1. 用户开始笔划。
  - a. [DynamicRenderer](#) 创建可视化树。
2. 用户正在绘制笔划。
  - a. [DynamicRenderer](#) 生成可视化树。
3. 用户结束笔划。
  - a. [InkPresenter](#) 将笔划添加到其可视化树中。
  - b. 媒体集成层 (MIL) 静态呈现笔划。
  - c. [DynamicRenderer](#) 清除视觉对象。

# 数字墨迹帮助主题

项目 • 2023/02/06

## 本节内容

- [从自定义控件选择墨迹](#)
- [向墨迹数据添加自定义数据](#)
- [清除自定义控件上的墨迹](#)
- [识别应用程序笔势](#)
- [拖放墨迹](#)
- [将数据绑定到 InkCanvas](#)
- [通过分析提示来分析墨迹](#)
- [旋转墨迹](#)
- [禁用用于 WPF 应用程序的 RealTimeStylus](#)

# 如何：从自定义控件选择墨迹

项目 • 2023/02/06

通过将 [IncrementalLassoHitTester](#) 添加到你的自定义控件，可以启用你的控件，以便用户可使用套索工具选择墨迹，这类似于 [InkCanvas](#) 使用套索选择墨迹的方式。

此示例假设你熟悉如何创建启用了墨迹的自定义控件。要创建自定义控件以接受墨迹输入，请参阅[创建墨迹输入控件](#)。

## 示例

当用户绘制套索时，[IncrementalLassoHitTester](#) 会预测用户完成套索后哪些笔划将位于套索路径的边界内。被确定在套索路径边界内的笔划可以被认为是被选中的。选中的笔划也可以变为未选中状态。例如，如果用户在绘制套索时反转方向，则 [IncrementalLassoHitTester](#) 可能会取消选择某些笔划。

[IncrementalLassoHitTester](#) 引发 [SelectionChanged](#) 事件，这使你的自定义控件能够在用户绘制套索时做出响应。例如，你可以在用户选择和取消选择时更改笔划的外观。

## 管理墨迹模式

如果套索的显示方式与控件上的墨迹不同，则对用户很有帮助。为此，你的自定义控件必须跟踪用户是写入还是选择墨迹。执行此操作的最简单方法是声明具有两个值的枚举：一个用于指示用户正在写入墨迹，一个用于指示用户正在选择墨迹。

C#

```
// Enum that keeps track of whether StrokeCollectionDemo is in ink mode
// or select mode.
public enum InkMode
{
    Ink, Select
}
```

接下来，将两个 [DrawingAttributes](#) 添加到类：一个在用户写入墨迹时使用，一个在用户选择墨迹时使用。在构造函数中，初始化 [DrawingAttributes](#) 并将两个 [AttributeChanged](#) 事件附加到同一个事件处理程序。然后将 [DynamicRenderer](#) 的 [DrawingAttributes](#) 属性设置为墨迹 [DrawingAttributes](#)。

C#

```
DrawingAttributes inkDA;
DrawingAttributes selectDA;
```

C#

```
// In the constructor.
// Selection drawing attributes use dark gray ink.
selectDA = new DrawingAttributes();
selectDA.Color = Colors.DarkGray;

// Ink drawing attributes use default attributes
inkDA = new DrawingAttributes();
inkDA.Width = 5;
inkDA.Height = 5;

inkDA.AttributeChanged += new
PropertyDataChangedEventHandler(DrawingAttributesChanged);
selectDA.AttributeChanged += new
PropertyDataChangedEventHandler(DrawingAttributesChanged);
```

添加一个属性来公开选择模式。当用户更改选择模式时，将 [DynamicRenderer](#) 的 [DrawingAttributes](#) 属性设置为适当的 [DrawingAttributes](#) 对象，然后将 [RootVisual](#) 属性重新附加到 [InkPresenter](#)。

C#

```
// Property to indicate whether the user is inputting or
// selecting ink.
public InkMode Mode
{
    get
    {
        return mode;
    }

    set
    {
        mode = value;

        // Set the DrawingAttributes of the DynamicRenderer
        if (mode == InkMode.Ink)
        {
            renderer.DrawingAttributes = inkDA;
        }
        else
        {
            renderer.DrawingAttributes = selectDA;
        }

        // Reattach the visual of the DynamicRenderer to the InkPresenter.
    }
}
```

```
        presenter.DetachVisuals(renderer.RootVisual);
        presenter.AttachVisuals(renderer.RootVisual,
renderer.DrawingAttributes);
    }
}
```

将 [DrawingAttributes](#) 公开为属性，以便应用程序可以确定墨迹笔划和选择笔划的外观。

C#

```
// Property to allow the user to change the pen's DrawingAttributes.
public DrawingAttributes InkDrawingAttributes
{
    get
    {
        return inkDA;
    }
}

// Property to allow the user to change the Selector's newStroke
DrawingAttributes.
public DrawingAttributes SelectDrawingAttributes
{
    get
    {
        return selectDA;
    }
}
```

当 [DrawingAttributes](#) 对象的属性发生更改时，必须将 [RootVisual](#) 重新附加到 [InkPresenter](#)。在 [AttributeChanged](#) 事件的事件处理程序中，将 [RootVisual](#) 重新附加到 [InkPresenter](#)。

C#

```
void DrawingAttributesChanged(object sender, PropertyChangedEventArgs e)
{
    // Reattach the visual of the DynamicRenderer to the InkPresenter
    // whenever the DrawingAttributes change.
    presenter.DetachVisuals(renderer.RootVisual);
    presenter.AttachVisuals(renderer.RootVisual,
renderer.DrawingAttributes);
}
```

## 使用 IncrementalLassoHitTester

创建并初始化包含所选笔划的 [StrokeCollection](#)。

C#

```
// StylusPointCollection that collects the stylus points from the stylus events.  
StylusPointCollection stylusPoints;
```

当用户开始绘制笔划（墨迹或套索）时，取消选择任何选定的笔划。然后，如果用户正在绘制套索，则通过调用 [GetIncrementalLassoHitTester](#) 创建一个 [IncrementalLassoHitTester](#)，订阅 [SelectionChanged](#) 事件，然后调用 [AddPoints](#)。此代码可以是单独的方法，并可使用 [OnStylusDown](#) 和 [OnMouseDown](#) 方法进行调用。

C#

```
private void InitializeHitTester(StylusPointCollection collectedPoints)  
{  
    // Deselect any selected strokes.  
    foreach (Stroke selectedStroke in selectedStrokes)  
    {  
        selectedStroke.DrawingAttributes.Color = inkDA.Color;  
    }  
    selectedStrokes.Clear();  
  
    if (mode == InkMode.Select)  
    {  
        // Remove the previously drawn lasso, if it exists.  
        if (lassoPath != null)  
        {  
            presenter.Strokes.Remove(lassoPath);  
            lassoPath = null;  
        }  
  
        selectionTester =  
            presenter.Strokes.GetIncrementalLassoHitTester(80);  
        selectionTester.SelectionChanged +=  
            new  
        LassoSelectionChangedEventHandler(selectionTester_SelectionChanged);  
        selectionTester.AddPoints(collectedPoints);  
    }  
}
```

在用户绘制套索时将触控笔的点添加到 [IncrementalLassoHitTester](#)。使用 [OnStylusMove](#)、[OnStylusUp](#)、[OnMouseMove](#) 和 [OnMouseLeftButtonUp](#) 方法调用以下方法。

C#

```
private void AddPointsToHitTester(StylusPointCollection collectedPoints)  
{  
    if (mode == InkMode.Select &&
```

```

        selectionTester != null &&
        selectionTester.IsValid)
    {
        // When the control is selecting strokes, add the
        // stylus packetList to selectionTester.
        selectionTester.AddPoints(collectedPoints);
    }
}

```

处理 `IncrementalLassoHitTester.SelectionChanged` 事件以在用户选择和取消选择笔划时做出响应。`LassoSelectionChangedEventArgs` 类具有 `SelectedStrokes` 和 `DeselectedStrokes` 属性，分别用于获取选中和未选中的笔划。

C#

```

void selectionTester_SelectionChanged(object sender,
    LassoSelectionChangedEventArgs args)
{
    // Change the color of all selected strokes to red.
    foreach (Stroke selectedStroke in args.SelectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = Colors.Red;
        selectedStrokes.Add(selectedStroke);
    }

    // Change the color of all unselected strokes to
    // their original color.
    foreach (Stroke unselectedStroke in args.DeselectedStrokes)
    {
        unselectedStroke.DrawingAttributes.Color = inkDA.Color;
        selectedStrokes.Remove(unselectedStroke);
    }
}

```

当用户绘制完套索后，取消订阅 `SelectionChanged` 事件并调用 `EndHitTesting`。

C#

```

if (mode == InkMode.Select && lassoPath == null)
{
    // Add the lasso to the InkPresenter and add the packetList
    // to selectionTester.
    lassoPath = newStroke;
    lassoPath.DrawingAttributes = selectDA.Clone();
    presenter.Strokes.Add(lassoPath);
    selectionTester.SelectionChanged -= new
    LassoSelectionChangedEventHandler
        (selectionTester_SelectionChanged);
    selectionTester.EndHitTesting();
}

```

# 放在一起后如下所示。

下面的示例是一个自定义控件，使用户能够使用套索选择墨迹。

C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Input;
using System.Windows.Input.StylusPlugIns;
using System.Windows.Ink;

// Enum that keeps track of whether StrokeCollectionDemo is in ink mode
// or select mode.
public enum InkMode
{
    Ink, Select
}

// This control allows the user to input and select ink. When the
// user selects ink, the lasso remains visible until they erase, or clip
// the selected strokes, or clear the selection. When the control is
// in selection mode, strokes that are selected turn red.
public class InkSelector : Label
{
    InkMode mode;

    DrawingAttributes inkDA;
    DrawingAttributes selectDA;

    InkPresenter presenter;
    IncrementalLassoHitTester selectionTester;
    StrokeCollection selectedStrokes = new StrokeCollection();

    // StylusPointCollection that collects the stylus points from the stylus
    events.
    StylusPointCollection stylusPoints;
    // Stroke that represents the lasso.
    Stroke lassoPath;

    DynamicRenderer renderer;

    public InkSelector()
    {
        mode = InkMode.Ink;

        // Use an InkPresenter to display the strokes on the custom control.
        presenter = new InkPresenter();
        this.Content = presenter;

        // In the constructor.
    }
}
```

```

    // Selection drawing attributes use dark gray ink.
    selectDA = new DrawingAttributes();
    selectDA.Color = Colors.DarkGray;

    // Ink drawing attributes use default attributes
    inkDA = new DrawingAttributes();
    inkDA.Width = 5;
    inkDA.Height = 5;

    inkDA.AttributeChanged += new
    PropertyDataChangedEventHandler(DrawingAttributesChanged);
    selectDA.AttributeChanged += new
    PropertyDataChangedEventHandler(DrawingAttributesChanged);

    // Add a DynamicRenderer to the control so ink appears
    // to "flow" from the tablet pen.
    renderer = new DynamicRenderer();
    renderer.DrawingAttributes = inkDA;
    this.StylusPlugIns.Add(renderer);
    presenter.AttachVisuals(renderer.RootVisual,
        renderer.DrawingAttributes);
}

static InkSelector()
{
    // Allow ink to be drawn only within the bounds of the control.
    Type owner = typeof(InkSelector);
    ClipToBoundsProperty.OverrideMetadata(owner,
        new FrameworkPropertyMetadata(true));
}

// Prepare to collect stylus packets. If Mode is set to Select,
// get the IncrementalHitTester from the InkPresenter's newStroke
// StrokeCollection and subscribe to its StrokeHitChanged event.
protected override void OnStylusDown(StylusEventArgs e)
{
    base.OnStylusDown(e);

    Stylus.Capture(this);

    // Create a new StylusPointCollection using the
    StylusPointDescription
    // from the stylus points in the StylusDownEventArgs.
    stylusPoints = new StylusPointCollection();
    StylusPointCollection eventPoints = e.GetStylusPoints(this,
    stylusPoints.Description);

    stylusPoints.Add(eventPoints);

    InitializeHitTester(eventPoints);
}

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);
}

```

```

        Mouse.Capture(this);

        if (e.StylusDevice != null)
        {
            return;
        }

        Point pt = e.GetPosition(this);

        StylusPointCollection collectedPoints = new
StylusPointCollection(new Point[] { pt });

        stylusPoints = new StylusPointCollection();

        stylusPoints.Add(collectedPoints);

        InitializeHitTester(collectedPoints);
    }
    private void InitializeHitTester(StylusPointCollection collectedPoints)
{
    // Deselect any selected strokes.
    foreach (Stroke selectedStroke in selectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = inkDA.Color;
    }
    selectedStrokes.Clear();

    if (mode == InkMode.Select)
    {
        // Remove the previously drawn lasso, if it exists.
        if (lassoPath != null)
        {
            presenter.Strokes.Remove(lassoPath);
            lassoPath = null;
        }

        selectionTester =
            presenter.Strokes.GetIncrementalLassoHitTester(80);
        selectionTester.SelectionChanged +=
            new
LassoSelectionChangedEventHandler(selectionTester_SelectionChanged);
        selectionTester.AddPoints(collectedPoints);
    }
}

// Collect the stylus packets as the stylus moves.
protected override void OnStylusMove(StylusEventArgs e)
{
    if (stylusPoints == null)
    {
        return;
    }

    StylusPointCollection collectedPoints = e.GetStylusPoints(this),

```

```

        stylusPoints.Description);
        stylusPoints.Add(collectedPoints);
        AddPointsToHitTester(collectedPoints);
    }

    protected override void OnMouseMove(MouseEventArgs e)
    {

        base.OnMouseMove(e);

        if (e.StylusDevice != null)
        {
            return;
        }

        if (e.LeftButton == MouseButtonState.Released)
        {
            return;
        }

        stylusPoints ??= new StylusPointCollection();

        Point pt = e.GetPosition(this);

        StylusPointCollection collectedPoints = new
StylusPointCollection(new Point[] { pt });

        stylusPoints.Add(collectedPoints);

        AddPointsToHitTester(collectedPoints);
    }

    private void AddPointsToHitTester(StylusPointCollection collectedPoints)
    {

        if (mode == InkMode.Select &&
            selectionTester != null &&
            selectionTester.IsValid)
        {
            // When the control is selecting strokes, add the
            // stylus packetList to selectionTester.
            selectionTester.AddPoints(collectedPoints);
        }
    }

    // When the user lifts the stylus, create a Stroke from the
    // collected stylus points and add it to the InkPresenter.
    // When the control is selecting strokes, add the
    // point data to the IncrementalHitTester.
    protected override void OnStylusUp(StylusEventArgs e)
    {
        stylusPoints ??= new StylusPointCollection();
        StylusPointCollection collectedPoints =
            e.GetStylusPoints(this, stylusPoints.Description);
    }
}

```

```
        stylusPoints.Add(collectedPoints);
        AddPointsToHitTester(collectedPoints);
        AddStrokeToPresenter();
        stylusPoints = null;

        Stylus.Capture(null);
    }

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{
    base.OnMouseLeftButtonUp(e);

    if (e.StylusDevice != null) return;

    if (stylusPoints == null) stylusPoints = new
    StylusPointCollection();

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new
    StylusPointCollection(new Point[] { pt });

    stylusPoints.Add(collectedPoints);
    AddPointsToHitTester(collectedPoints);
    AddStrokeToPresenter();

    stylusPoints = null;

    Mouse.Capture(null);
}

private void AddStrokeToPresenter()
{
    Stroke newStroke = new Stroke(stylusPoints);

    if (mode == InkMode.Ink)
    {
        // Add the stroke to the InkPresenter.
        newStroke.DrawingAttributes = inkDA.Clone();
        presenter.Strokes.Add(newStroke);
    }

    if (mode == InkMode.Select && lassoPath == null)
    {
        // Add the lasso to the InkPresenter and add the packetList
        // to selectionTester.
        lassoPath = newStroke;
        lassoPath.DrawingAttributes = selectDA.Clone();
        presenter.Strokes.Add(lassoPath);
        selectionTester.SelectionChanged -= new
        LassoSelectionChangedEventHandler
            (selectionTester_SelectionChanged);
        selectionTester.EndHitTesting();
    }
}
```

```

}

void selectionTester_SelectionChanged(object sender,
    LassoSelectionChangedEventArgs args)
{
    // Change the color of all selected strokes to red.
    foreach (Stroke selectedStroke in args.SelectedStrokes)
    {
        selectedStroke.DrawingAttributes.Color = Colors.Red;
        selectedStrokes.Add(selectedStroke);
    }

    // Change the color of all unselected strokes to
    // their original color.
    foreach (Stroke unselectedStroke in args.DeselectedStrokes)
    {
        unselectedStroke.DrawingAttributes.Color = inkDA.Color;
        selectedStrokes.Remove(unselectedStroke);
    }
}

// Property to indicate whether the user is inputting or
// selecting ink.
public InkMode Mode
{
    get
    {
        return mode;
    }

    set
    {
        mode = value;

        // Set the DrawingAttributes of the DynamicRenderer
        if (mode == InkMode.Ink)
        {
            renderer.DrawingAttributes = inkDA;
        }
        else
        {
            renderer.DrawingAttributes = selectDA;
        }

        // Reattach the visual of the DynamicRenderer to the
        // InkPresenter.
        presenter.DetachVisuals(renderer.RootVisual);
        presenter.AttachVisuals(renderer.RootVisual,
        renderer.DrawingAttributes);
    }
}

void DrawingAttributesChanged(object sender,
PropertyDataChangedEventArgs e)
{
    // Reattach the visual of the DynamicRenderer to the InkPresenter
}

```

```
// whenever the DrawingAttributes change.  
presenter.DetachVisuals(renderer.RootVisual);  
presenter.AttachVisuals(renderer.RootVisual,  
renderer.DrawingAttributes);  
}  
  
// Property to allow the user to change the pen's DrawingAttributes.  
public DrawingAttributes InkDrawingAttributes  
{  
    get  
    {  
        return inkDA;  
    }  
}  
  
// Property to allow the user to change the Selector's newStroke  
DrawingAttributes.  
public DrawingAttributes SelectDrawingAttributes  
{  
    get  
    {  
        return selectDA;  
    }  
}  
}
```

## 另请参阅

- [IncrementalLassoHitTester](#)
- [StrokeCollection](#)
- [StylusPointCollection](#)
- [创建墨迹输入控件](#)

# 如何：向墨迹数据添加自定义数据

项目 • 2023/02/06

可向墨迹添加自定义数据，这些数据将在墨迹另存为墨迹序列化格式 (ISF) 时保存。可将自定义数据保存到 [DrawingAttributes](#)、[StrokeCollection](#) 或 [Stroke](#)。如果能够将自定义数据保存到三个对象上，你可决定保存数据的最佳位置。这三个类都使用类似的方法来存储和访问自定义数据。

只有下列类型才能另存为自定义数据：

- [Boolean](#)
- [Boolean\[\]](#)
- [Byte](#)
- [Byte\[\]](#)
- [Char](#)
- [Char\[\]](#)
- [DateTime](#)
- [DateTime\[\]](#)
- [Decimal](#)
- [Decimal\[\]](#)
- [Double](#)
- [Double\[\]](#)
- [Int16](#)
- [Int16\[\]](#)
- [Int32](#)
- [Int32\[\]](#)
- [Int64](#)
- [Int64\[\]](#)
- [Single](#)

- Single[]
- String
- UInt16
- UInt16[]
- UInt32
- UInt32[]
- UInt64
- UInt64[]

## 示例

以下示例演示如何在 [StrokeCollection](#) 中添加和检索自定义数据。

C#

```
Guid timestamp = new Guid("12345678-9012-3456-7890-123456789012");

// Add a timestamp to the StrokeCollection.
private void AddTimestamp()
{
    inkCanvas1.Strokes.AddPropertyData(timestamp, DateTime.Now);
}

// Get the timestamp of the StrokeCollection.
private void GetTimestamp()
{
    if (inkCanvas1.Strokes.ContainsPropertyData(timestamp))
    {
        object date = inkCanvas1.Strokes.GetPropertyData(timestamp);

        if (date is DateTime)
        {
            MessageBox.Show("This StrokeCollection's timestamp is " +
                           ((DateTime)date).ToString());
        }
        else
        {
            MessageBox.Show(
                "The StrokeCollection does not have a timestamp.");
        }
    }
}
```

以下示例将创建一个显示一个 InkCanvas 和两个按钮的应用程序。通过 switchAuthor 按钮，两支笔可以由两位不同的作者使用。 changePenColors 按钮可根据作者更改 InkCanvas 上的每个笔划的颜色。应用程序定义两个 DrawingAttributes 对象，并向每个对象添加一个自定义属性，以指示哪个作者绘制了 Stroke。当用户单击 changePenColors 时，应用程序会根据自定义属性的值更改笔划的外观。

#### XAML

```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Adding Custom Data to Ink" Height="500" Width="700"
    >
<DockPanel Name="root">

    <StackPanel Background="DarkSlateBlue">
        <Button Name="switchAuthor" Click="switchAuthor_click" >
            Switch to student's pen
        </Button>
        <Button Name="changePenColors" Click="changeColor_click" >
            Change the color of the pen ink
        </Button>
    </StackPanel>
    <InkCanvas Name="inkCanvas1">
    </InkCanvas>
</DockPanel>
</Window>
```

#### C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Ink;

/// <summary>
/// Interaction logic for Window1.xaml
/// </summary>

public partial class Window1 : Window
{
    Guid authorGuid = new Guid("12345678-9012-3456-7890-123456789012");
    DrawingAttributes teachersDA = new DrawingAttributes();
    DrawingAttributes studentsDA = new DrawingAttributes();
    string teacher = "teacher";
```

```

string student = "student";
bool useStudentPen = false;

public Window1()
{
    InitializeComponent();

    teachersDA.Color = Colors.Red;
    teachersDA.Width = 5;
    teachersDA.Height = 5;
    teachersDA.AddPropertyData(authorGuid, teacher);

    studentsDA.Color = Colors.Blue;
    studentsDA.Width = 5;
    studentsDA.Height = 5;
    studentsDA.AddPropertyData(authorGuid, student);

    inkCanvas1.DefaultDrawingAttributes = teachersDA;
}

// Switch between using the 'pen' DrawingAttributes and the
// 'highlighter' DrawingAttributes.
void switchAuthor_click(Object sender, RoutedEventArgs e)
{
    useStudentPen = !useStudentPen;

    if (useStudentPen)
    {
        switchAuthor.Content = "Use teacher's pen";
        inkCanvas1.DefaultDrawingAttributes = studentsDA;
    }
    else
    {
        switchAuthor.Content = "Use student's pen";
        inkCanvas1.DefaultDrawingAttributes = teachersDA;
    }
}

// Change the color of the ink that on the InkCanvas that used the pen.
void changeColor_click(Object sender, RoutedEventArgs e)
{
    foreach (Stroke s in inkCanvas1.Strokes)
    {
        if (s.DrawingAttributes.ContainsPropertyData(authorGuid))
        {
            object data =
s.DrawingAttributes.GetPropertyData(authorGuid);

            if ((data is string) && ((string)data == teacher))
            {
                s.DrawingAttributes.Color = Colors.Black;
            }
            if ((data is string) && ((string)data == student))
            {
                s.DrawingAttributes.Color = Colors.Green;
            }
        }
    }
}

```

```
        }
    }
}
}
```

# 如何：清除自定义控件上的墨迹

项目 • 2023/02/06

[IncrementalStrokeHitTester](#) 确定当前绘制的笔划是否与另一个笔划相交。这可用于创建一个使用户能够擦除部分笔划的控件，就像用户在 [EditMode](#) 设置为 [EraseByPoint](#) 时在 [InkCanvas](#) 上所做的那样。

## 示例

以下示例创建一个自定义控件，使用户能够擦除部分笔划。此示例创建一个在初始化时包含墨迹的控件。若要创建收集墨迹的控件，请参阅[创建墨迹输入控件](#)。

C#

```
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Windows.Input;
using System.Windows.Media;
using System.IO;

// This control initializes with ink already on it and allows the
// user to erase the ink with the tablet pen or mouse.
public class InkEraser : Label
{
    IncrementalStrokeHitTester eraseTester;

    InkPresenter presenter;

    string strokesString =
        @"ALwHAXdIEETLgIgERYQBwIAJAFGhAEbAgAkAQUBOBkgMgkA9P8CAekiOkUzCQD4n"
        + "wIBWiA6RTgIAP4DAAAAGh8RAACAPx8JEQAAAAAAAPA/CiUHh/A6N4HR0AivFX8Vs"
        + "IfsiuyLSaIeDSLwabiHm0GgUDi+KZkACjsQh/A9t4IC5VJpfLhaIxxyxXIh7Dncnh"
        + "+6e7qODwoER1PAw8EpGoJAh61IKjCYXBYDA4DAIH67KIHAojb4fwMteBn+RKB"
        + "lziaIfwWTcWePqbM8WgIeeQCDQ0FRvcIAKNA+h8B8XgQHkUbjQTTnGuaZns3l4h"
        + "/Dwt4a0YKBB0A94D6HRCAiGnp5CS8LExMLB1t0gYIAKUBOH8KnXhU71Mold+tcbi"
        + "kChkqu2EtPYxp9bmYCH8HDHg4ZhMIwRyMHH+4Jt8n1eX8c0/D/AkYwxJGiHkkQgM"
        + "Ch9CqcFhMDQCBlAwuR2eAACmgdh/EpF41A6XMUfhMXgMHgVDxBFpRKpZII5EINA"
        + "OA64M+J4Lw1CIoAh/B2x4PS4bQodAopEI5IJBki4waEx2Qy+dy+ayHgleEmmHH8c"
        + "e3MZOCGw5Twd3CwsHAWMCgRAEAgE1KwOHZKBApaGIfxezeL0uN02N8IzwaGEpNIJ"
        + "ZxHnELy0j0GfyuU6Fgmhp1IgIfwYgeDHea1vj0tZgcHgHAYb9hUCgEFgsPm1xnM"
        + "ZkYhsnYJgZeZh4uAgCgnSBl0Jv40AgwCmkgh/GrR41X4dGoRJL9EKra5HKY7IZ3C"
        + "4fj/M06olSoU8kkehUbh8jkMdCH8IJXhAXhMCk8JuN1mNyh0YiEumUwn2wMRxyHw"
        + "2TzWmzeb020zGKxMITwIhnrrjzb44zRhGEKRhCM4zrr6sQKXRWH8kuXkmPj0DiXC"
        + "gcJbC9HZZgkKgUG4bLh3YrwJHAYw2CAh/Cin4Tq7DOZr4BB/AFtdOWW5P2h1Wkzv"
        + "l4+YwqXF8d5fZ7ih51QKbB4LQrlAYDBIDABA4B04nAICApvIIfy4BeXA2DRSrQlL"
        + "oHHsYQ/KMX1sv18rn8Xkcdg+G9NVaUWimUDYk9Ah/BoF4M0YBCqZPYqk8dwLf7hD"
        + "YNBJFLKBNqZTqNubWsh19VoM1reFYZYQEBGUoDAwKEjYuDQKBgICBgcAgIOAg4nI"
        + "80AC1oSh/BFl4Gf/I0t6FXff8F4ToPCzzIPwP4+B+DHmQ0847rfDeCcG8eKh/EZV"
```

```
+ "4i9eZt8A9nUF8VzxaUe5grl7YrPaHfpRKJNx4yHmUuj1vicwmMBEAjUVgKB61A=";

public InkEraser()
{
    presenter = new InkPresenter();
    this.Content = presenter;

    // Create a StrokeCollection the string and add it to
    StrokeCollectionConverter converter =
        new StrokeCollectionConverter();

    if (converter.CanConvertFrom(typeof(string)))
    {
        StrokeCollection newStrokes =
            converter.ConvertFrom(strokesString) as StrokeCollection;
        presenter.Strokes.Clear();
        presenter.Strokes.Add(newStrokes);
    }
}

protected override void OnStylusDown(StylusEventArgs e)
{
    base.OnStylusDown(e);
    StylusPointCollection points = e.GetStylusPoints(this);

    InitializeEraserHitTester(points);
}

protected override void OnMouseLeftButtonDown(MouseEventArgs e)
{
    base.OnMouseLeftButtonDown(e);

    if (e.StylusDevice != null)
    {
        return;
    }

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new
    StylusPointCollection(new Point[] { pt });

    InitializeEraserHitTester(collectedPoints);
}

// Prepare to collect stylus packets. Get the
// IncrementalHitTester from the InkPresenter's
// StrokeCollection and subscribe to its StrokeHitChanged event.
private void InitializeEraserHitTester(StylusPointCollection points)
{
    EllipseStylusShape eraserTip = new EllipseStylusShape(3, 3, 0);
    eraseTester =
        presenter.Strokes.GetIncrementalStrokeHitTester(eraserTip);
    eraseTester.StrokeHit += new
    StrokeHitEventHandler(eraseTester_StrokeHit);
}
```

```
        eraseTester.AddPoints(points);
    }

protected override void OnStylusMove(StylusEventArgs e)
{
    StylusPointCollection points = e.GetStylusPoints(this);

    AddPointsToEraserHitTester(points);
}

protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);

    if (e.StylusDevice != null)
    {
        return;
    }

    if (e.LeftButton == MouseButtons.Released)
    {
        return;
    }

    Point pt = e.GetPosition(this);

    StylusPointCollection collectedPoints = new
StylusPointCollection(new Point[] { pt });

    AddPointsToEraserHitTester(collectedPoints);
}

// Collect the StylusPackets as the stylus moves.
private void AddPointsToEraserHitTester(StylusPointCollection points)
{
    if (eraseTester.IsValid)
    {
        eraseTester.AddPoints(points);
    }
}

// Unsubscribe from the StrokeHitChanged event when the
// user lifts the stylus.
protected override void OnStylusUp(StylusEventArgs e)
{
    StylusPointCollection points = e.GetStylusPoints(this);

    StopEraseHitTesting(points);
}

protected override void OnMouseLeftButtonUp(MouseEventArgs e)
{
    base.OnMouseLeftButtonUp(e);

    if (e.StylusDevice != null)
```

```

        {
            return;
        }

        Point pt = e.GetPosition(this);

        StylusPointCollection collectedPoints = new
        StylusPointCollection(new Point[] { pt });

        StopEraseHitTesting(collectedPoints);
    }

    private void StopEraseHitTesting(StylusPointCollection points)
    {
        eraseTester.AddPoints(points);
        eraseTester.StrokeHit -= new
            StrokeHitEventHandler(eraseTester_StrokeHit);
        eraseTester.EndHitTesting();
    }

    // When the stylus intersects a stroke, erase that part of
    // the stroke. When the stylus dissects a stroke, the
    // Stroke.Erase method returns a StrokeCollection that contains
    // the two new strokes.
    void eraseTester_StrokeHit(object sender,
        StrokeHitEventArgs args)
    {
        StrokeCollection eraseResult =
            args.GetPointEraseResults();
        StrokeCollection strokesToReplace = new StrokeCollection();
        strokesToReplace.Add(args.HitStroke);

        // Replace the old stroke with the new one.
        if (eraseResult.Count > 0)
        {
            presenter.Strokes.Replace(strokesToReplace, eraseResult);
        }
        else
        {
            presenter.Strokes.Remove(strokesToReplace);
        }
    }
}

```

# 如何：识别应用程序操作

项目 • 2023/02/06

以下示例演示当用户在 InkCanvas 上做 ScratchOut 手势时如何擦除墨迹。本示例假定在 XAML 文件中声明了一个名为 inkCanvas1 的 InkCanvas。

## 示例

C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Collections.ObjectModel;

public partial class Window1 : Window
{
    public Window1()
    {
        InitializeComponent();

        if (inkCanvas1.IsGestureRecognizerAvailable)
        {
            inkCanvas1.EditingMode = InkCanvasEditingStyle.InkAndGesture;
            inkCanvas1.Gesture += new
InkCanvasGestureEventHandler(inkCanvas1_Gesture);
            inkCanvas1.SetEnabledGestures(
                new ApplicationGesture[] { ApplicationGesture.ScratchOut });
        }
    }

    void inkCanvas1_Gesture(object sender, InkCanvasGestureEventArgs e)
    {
        ReadOnlyCollection<GestureRecognitionResult> gestureResults =
            e.GetGestureRecognitionResults();

        // Check the first recognition result for a gesture.
        if ((gestureResults[0].RecognitionConfidence ==
            RecognitionConfidence.Strong) &&
            (gestureResults[0].ApplicationGesture ==
            ApplicationGesture.ScratchOut))
        {
            StrokeCollection hitStrokes = inkCanvas1.Strokes.HitTest(
                e.Strokes.GetBounds(), 10);

            if (hitStrokes.Count > 0)
            {
                inkCanvas1.Strokes.Remove(hitStrokes);
            }
        }
    }
}
```

```
        }  
    }  
}
```

## 另请参阅

- [ApplicationGesture](#)
- [InkCanvas](#)
- [Gesture](#)

# 如何：拖放墨迹

项目 • 2023/02/06

## 示例

以下示例创建一个应用程序，使用户能够将所选笔划从一个 InkCanvas 拖动到另一个。

XAML

```
<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="InkDragDropSample" Height="500" Width="700"
    >
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition/>
        <ColumnDefinition/>
    </Grid.ColumnDefinitions>

    <InkCanvas Name="ic1" AllowDrop="True"
        Grid.Column="0" Grid.Row="0"
        Margin="10,10,10,10" Background="AliceBlue"
        PreviewMouseDown="InkCanvas_PreviewMouseDown"
        Drop="InkCanvas_Drop"/>

    <InkCanvas Name="ic2" AllowDrop="True"
        Grid.Column="1" Grid.Row="0"
        Margin="10,10,10,10" Background="Beige"
        PreviewMouseDown="InkCanvas_PreviewMouseDown"
        Drop="InkCanvas_Drop"/>

    <CheckBox Grid.Row="1"
        Checked="switchToSelect" Unchecked="switchToInk">
        Select Mode
    </CheckBox>
</Grid>
</Window>
```

C#

```
using System;
using System.IO;
using System.Windows;
using System.Windows.Ink;
```

```

using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Input;
using System.Windows.Media;

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

    void InkCanvas_PreviewMouseDown(object sender, MouseEventArgs e)
    {
        InkCanvas ic = (InkCanvas)sender;

        Point pt = e.GetPosition(ic);

        // If the user is moving selected strokes, prepare the strokes to be
        // moved to another InkCanvas.
        if (ic.HitTestSelection(pt) ==
            InkCanvasSelectionHitResult.Selection)
        {
            StrokeCollection selectedStrokes = ic.GetSelectedStrokes();
            StrokeCollection strokesToMove = selectedStrokes.Clone();

            // Remove the offset of the selected strokes so they
            // are positioned when the strokes are dropped.
            Rect inkBounds = strokesToMove.GetBounds();
            TranslateStrokes(strokesToMove, -inkBounds.X, -inkBounds.Y);

            // Perform drag and drop.
            MemoryStream ms = new MemoryStream();
            strokesToMove.Save(ms);
            DataObject dataObject = new DataObject(
                StrokeCollection.InkSerializedFormat, ms);

            DragDropEffects effects =
                DragDrop.DoDragDrop(ic, dataObject,
                    DragDropEffects.Move);

            if ((effects & DragDropEffects.Move) ==
                DragDropEffects.Move)
            {
                // Remove the selected strokes
                // from the current InkCanvas.
                ic.Strokes.Remove(selectedStrokes);
            }
        }
    }

    void InkCanvas_Drop(object sender, DragEventArgs e)
    {
        // Get the strokes that were moved.
    }
}

```

```

InkCanvas ic = (InkCanvas)sender;
MemoryStream ms = (MemoryStream)e.Data.GetData(
    StrokeCollection.InkSerializedFormat);
ms.Position = 0;
StrokeCollection strokes = new StrokeCollection(ms);

// Translate the strokes to the position at which
// they were dropped.
Point pt = e.GetPosition(ic);
TranslateStrokes(strokes, pt.X, pt.Y);

// Add the strokes to the InkCanvas and keep them selected.
ic.Strokes.Add(strokes);
ic.Select(strokes);
}

// Helper method that translates the specified strokes.
void TranslateStrokes(StrokeCollection strokes,
                      double x, double y)
{
    Matrix mat = new Matrix();
    mat.Translate(x, y);
    strokes.Transform(mat, false);
}

void switchToSelect(object sender, RoutedEventArgs e)
{
    ic1.EditingMode = InkCanvasEditingStyle.Select;
    ic2.EditingMode = InkCanvasEditingStyle.Select;
}

void switchToInk(object sender, RoutedEventArgs e)
{
    ic1.EditingMode = InkCanvasEditingStyle.Ink;
    ic2.EditingMode = InkCanvasEditingStyle.Ink;
}
}

```

# 如何：将数据绑定到 InkCanvas

项目 • 2023/02/06

## 示例

以下示例演示如何将 InkCanvas 的 Strokes 属性绑定到另一个 InkCanvas。

XAML

```
<InkCanvas Background="LightGray"
    Canvas.Top="0" Canvas.Left="0"
    Height="400" Width="200" Name="ic"/>

<!-- Bind the Strokes of the second InkCanvas to the first InkCanvas
    and mirror the strokes along the Y axis.-->
<InkCanvas Background="LightBlue"
    Canvas.Top="0" Canvas.Left="200"
    Height="400" Width="200"
    Strokes="{Binding ElementName=ic, Path=Strokes}">
    <InkCanvas.LayoutTransform>
        <ScaleTransform ScaleX="-1" ScaleY="1" />
    </InkCanvas.LayoutTransform>
</InkCanvas>
```

以下示例演示如何将 DefaultDrawingAttributes 属性绑定到数据源。

XAML

```
<Canvas.Resources>
    <!--Define an array containing some DrawingAttributes.-->
    <x:Array x:Key="MyDrawingAttributes" x:Type="{x:Type DrawingAttributes}">
        <DrawingAttributes Color="Black" FitToCurve="true" Width="3"
Height="3"/>
        <DrawingAttributes Color="Blue" FitToCurve="false" Width="5"
Height="5"/>
        <DrawingAttributes Color="Red" FitToCurve="true" Width="7"
Height="7"/>
    </x:Array>

    <!--Create a DataTemplate to display the DrawingAttributes shown above-->
    <DataTemplate DataType="{x:Type DrawingAttributes}" >
        <Border Width="80" Height="{Binding Path=Height}">
            <Border.Background>
                <SolidColorBrush Color="{Binding Path=Color}" />
            </Border.Background>
        </Border>
    </DataTemplate>
</Canvas.Resources>
```

## XAML

```
<!--Bind the InkCanvas' DefaultDrawingAtributes to  
    a Listbox, called lbDrawingAttributes.-->  
<InkCanvas Name="inkCanvas1" Background="LightGreen"  
    Canvas.Top="400" Canvas.Left="0"  
    Height="400" Width="400"  
    DefaultDrawingAttributes="{Binding  
        ElementName=lbDrawingAttributes, Path=SelectedItem}"  
    >  
</InkCanvas>  
  
<!--Use the array, MyDrawingAttributes, to populate a ListBox-->  
<ListBox Name="lbDrawingAttributes"  
    Canvas.Top="400" Canvas.Left="450"  
    Height="100" Width="100"  
    ItemsSource="{StaticResource MyDrawingAttributes}" />
```

以下示例声明 XAML 中的两个 `InkCanvas` 对象，并在这两个对象和其他数据源之间建立数据绑定。第一个名为 `ic` 的 `InkCanvas` 绑定到两个数据源。`ic` 上的 `EditingMode` 和 `DefaultDrawingAttributes` 属性绑定到 `ListBox` 对象，此对象又反过来绑定到 XAML 中定义的数组。第二个 `InkCanvas` 的 `EditMode`、`DefaultDrawingAttributes` 和 `Strokes` 属性绑定到第一个 `InkCanvas`，即 `ic`。

## XAML

```
<Canvas>  
    <Canvas.Resources>  
        <!--Define an array containing the InkEditingStyle Values.-->  
        <x:Array x:Key="MyEditingStyle" x:Type="{x:Type InkCanvasEditingStyle}">  
            <x:Static Member="InkCanvasEditingStyle.Ink"/>  
            <x:Static Member="InkCanvasEditingStyle.Select"/>  
            <x:Static Member="InkCanvasEditingStyle.EraseByPoint"/>  
            <x:Static Member="InkCanvasEditingStyle.EraseByStroke"/>  
        </x:Array>  
  
        <!--Define an array containing some DrawingAttributes.-->  
        <x:Array x:Key="MyDrawingAttributes"  
            x:Type="{x:Type DrawingAttributes}">  
            <DrawingAttributes Color="Black" FitToCurve="true"  
                Width="3" Height="3"/>  
            <DrawingAttributes Color="Blue" FitToCurve="false"  
                Width="5" Height="5"/>  
            <DrawingAttributes Color="Red" FitToCurve="true"  
                Width="7" Height="7"/>  
        </x:Array>  
  
        <!--Create a DataTemplate to display the  
            DrawingAttributes shown above-->  
        <DataTemplate DataType="{x:Type DrawingAttributes}">  
            <Border Width="80" Height="{Binding Path=Height}">
```

```

        <Border.Background>
            <SolidColorBrush Color="{Binding Path=Color}" />
        </Border.Background>
    </Border>
</DataTemplate>
</Canvas.Resources>

<!--Bind the first InkCanvas' DefaultDrawingAtributes to a
Listbox, called lbDrawingAttributes, and its EditingMode to
a ListBox called lbEditMode.-->
<InkCanvas Name="ic" Background="LightGray"
    Canvas.Top="0" Canvas.Left="0"
    Height="400" Width="200"
    DefaultDrawingAttributes="{Binding
        ElementName=lbDrawingAttributes, Path=SelectedItem}"
    EditingMode=
        "{Binding ElementName=lbEditMode, Path=SelectedItem}"
    >
</InkCanvas>

<!--Bind the Strokes, DefaultDrawingAtributes, and, EditingMode properties
of
the second InkCanvas the first InkCanvas.-->
<InkCanvas Background="LightBlue"
    Canvas.Top="0" Canvas.Left="200"
    Height="400" Width="200"
    Strokes="{Binding ElementName=ic, Path=Strokes}"
    DefaultDrawingAttributes="{Binding
        ElementName=ic, Path=DefaultDrawingAttributes}"
    EditingMode="{Binding ElementName=ic, Path=EditMode}">

<InkCanvas.LayoutTransform>
    <ScaleTransform ScaleX="-1" ScaleY="1" />
</InkCanvas.LayoutTransform>

</InkCanvas>

<!--Use the array, MyEditingModes, to populate a ListBox-->
<ListBox Name="lbEditMode"
    Canvas.Top="0" Canvas.Left="450"
    Height="100" Width="100"
    ItemsSource="{StaticResource MyEditingModes}" />

<!--Use the array, MyDrawingAttributes, to populate a ListBox-->
<ListBox Name="lbDrawingAttributes"
    Canvas.Top="150" Canvas.Left="450"
    Height="100" Width="100"
    ItemsSource="{StaticResource MyDrawingAttributes}" />

</Canvas>

```

# 如何：通过分析提示来分析墨迹

项目 • 2022/09/27

`System.Windows.Ink.AnalysisHintNode` 为其附加到的 `System.Windows.Ink.InkAnalyzer` 提供提示。该提示适用于 `System.Windows.Ink.AnalysisHintNode` 的 `System.Windows.Ink.ContextNode.Location` 属性指定的区域，并为墨迹分析器提供额外的上下文以提高识别准确度。`System.Windows.Ink.InkAnalyzer` 在分析从提示区域内获取的墨迹时应用此上下文信息。

## 示例

下面的示例是一个应用程序，该应用程序在接受墨迹输入的窗体上使用多个 `System.Windows.Ink.AnalysisHintNode` 对象。应用程序使用 `System.Windows.Ink.AnalysisHintNode.Factoid` 属性为窗体上的每一项提供上下文信息。应用程序使用后台分析来分析墨迹，并在用户停止添加墨迹 5 秒后清除所有墨迹的窗体。

XAML

```
<Window x:Class="FormAnalyzer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="FormAnalyzer"
    SizeToContent="WidthAndHeight"
>
<StackPanel Orientation="Vertical">
<InkCanvas Name="xaml_writingCanvas" Height="500" Width="840"
    StrokeCollected="RestartAnalysis" >
    <Grid>
        <Grid.Resources>
            <Style TargetType="{x:Type Label}">
                <Setter Property="FontSize" Value="20"/>
                <Setter Property="FontFamily" Value="Arial"/>
            </Style>

            <Style TargetType="{x:Type TextBlock}">
                <Setter Property="FontSize" Value="18"/>
                <Setter Property="VerticalAlignment"
Value="Center"/>
            </Style>
        </Grid.Resources>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="100"/></ColumnDefinition>
            <ColumnDefinition Width="160"/></ColumnDefinition>
            <ColumnDefinition Width="160"/></ColumnDefinition>
            <ColumnDefinition Width="100"/></ColumnDefinition>
```

```

        <ColumnDefinition Width="160"></ColumnDefinition>
        <ColumnDefinition Width="160"></ColumnDefinition>
    </Grid.ColumnDefinitions>

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

    <Label Grid.Row="0" Grid.Column="0">Title</Label>
    <Label Grid.Row="1" Grid.Column="0">Director</Label>
    <Label Grid.Row="2" Grid.Column="0">Starring</Label>
    <Label Grid.Row="3" Grid.Column="0">Rating</Label>
    <Label Grid.Row="3" Grid.Column="3">Year</Label>
    <Label Grid.Row="4" Grid.Column="0">Genre</Label>

    <TextBlock Name="xaml_blockTitle"
        Grid.Row="0" Grid.Column="1"
        Grid.ColumnSpan="5"/>
    <TextBlock Name="xaml_blockDirector"
        Grid.Row="1" Grid.Column="1"
        Grid.ColumnSpan="5"/>
    <TextBlock Name="xaml_blockStarring"
        Grid.Row="2" Grid.Column="1"
        Grid.ColumnSpan="5"/>
    <TextBlock Name="xaml_blockRating"
        Grid.Row="3" Grid.Column="1"
        Grid.ColumnSpan="2"/>
    <TextBlock Name="xaml_blockYear"
        Grid.Row="3" Grid.Column="4"
        Grid.ColumnSpan="2"/>
    <TextBlock Name="xaml_blockGenre"
        Grid.Row="4" Grid.Column="1"
        Grid.ColumnSpan="5"/>

    <Line Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="6"
        StrokeThickness="2" Stroke="Black"
        X1="0" Y1="100" X2="840" Y2="100" />
    <Line Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="6"
        StrokeThickness="2" Stroke="Black"
        X1="0" Y1="100" X2="840" Y2="100" />
    <Line Grid.Row="2" Grid.Column="0" Grid.ColumnSpan="6"
        StrokeThickness="2" Stroke="Black"
        X1="0" Y1="100" X2="840" Y2="100" />
    <Line Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="6"
        StrokeThickness="2" Stroke="Black"
        X1="0" Y1="100" X2="840" Y2="100" />
    <Line Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="6"
        StrokeThickness="2" Stroke="Black"
        X1="420" Y1="0" X2="420" Y2="100" />
</Grid>
</InkCanvas>
```

```
</StackPanel>
</Window>
```

C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Ink;
using System.Windows.Threading;

public partial class FormAnalyzer : Window
{
    private InkAnalyzer analyzer;

    private AnalysisHintNode hintNodeTitle;
    private AnalysisHintNode hintNodeDirector;
    private AnalysisHintNode hintNodeStarring;
    private AnalysisHintNode hintNodeRating;
    private AnalysisHintNode hintNodeYear;
    private AnalysisHintNode hintNodeGenre;

    // Timer that raises an event to
    // clear the InkCanvas.
    private DispatcherTimer strokeRemovalTimer;

    private const int CLEAR_STROKES_DELAY = 5;

    public FormAnalyzer()
    {
        InitializeComponent();
    }

    protected override void OnContentRendered(EventArgs e)
    {
        base.OnContentRendered(e);

        // Initialize the Analyzer.
        analyzer = new InkAnalyzer();
        analyzer.ResultsUpdated +=
            new ResultsUpdatedEventHandler(analyzer_ResultsUpdated);

        // Add analysis hints for each form area.
        // Use the absolute Width and Height of the Grid's
        // RowDefinition and ColumnDefinition properties defined in XAML,
        // to calculate the bounds of the AnalysisHintNode objects.
        hintNodeTitle = analyzer.CreateAnalysisHint(
            new Rect(100, 0, 740, 100));
        hintNodeDirector = analyzer.CreateAnalysisHint(
            new Rect(100, 100, 740, 100));
        hintNodeStarring = analyzer.CreateAnalysisHint(
            new Rect(100, 200, 740, 100));
        hintNodeRating = analyzer.CreateAnalysisHint(
```

```

                new Rect(100, 300, 320, 100));
hintNodeYear = analyzer.CreateAnalysisHint(
                new Rect(520, 300, 320, 100));
hintNodeGenre = analyzer.CreateAnalysisHint(
                new Rect(100, 400, 740, 100));

        //Set the factoids on the hints.
        hintNodeTitle.Factoid = "(!IS_DEFAULT)";
        hintNodeDirector.Factoid = "(!IS_PERSONALNAME_FULLNAME)";
        hintNodeStarring.Factoid = "(!IS_PERSONALNAME_FULLNAME)";
        hintNodeRating.Factoid = "(!IS_DEFAULT)";
        hintNodeYear.Factoid = "(!IS_DATE_YEAR)";
        hintNodeGenre.Factoid = "(!IS_DEFAULT)";
    }

    /// <summary>
    /// InkCanvas.StrokeCollected event handler. Begins
    /// ink analysis and starts the timer to clear the strokes.
    /// If five seconds pass without a Stroke being added,
    /// the strokes on the InkCanvas will be cleared.
    /// </summary>
    /// <param name="sender">InkCanvas that raises the
    /// StrokeCollected event.</param>
    /// <param name="args">Contains the event data.</param>
    private void RestartAnalysis(object sender,
                                InkCanvasStrokeCollectedEventArgs args)
    {

        // If strokeRemovalTimer is enabled, stop it.
        if (strokeRemovalTimer != null && strokeRemovalTimer.IsEnabled)
        {
            strokeRemovalTimer.Stop();
        }

        // Restart the timer to clear the strokes in five seconds
        strokeRemovalTimer = new DispatcherTimer(
            TimeSpan.FromSeconds(CLEAR_STROKES_DELAY),
            DispatcherPriority.Normal,
            ClearCanvas,
            Dispatcher.CurrentDispatcher);

        // Add the new stroke to the InkAnalyzer and
        // begin background analysis.
        analyzer.AddStroke(args.Stroke);
        analyzer.BackgroundAnalyze();
    }

    /// <summary>
    /// Analyzer.ResultsUpdated event handler.
    /// </summary>
    /// <param name="sender">InkAnalyzer that raises the
    /// event.</param>
    /// <param name="e">Event data</param>
    /// <remarks>This method checks each AnalysisHint for
    /// analyzed ink and then populated the TextBlock that

```

```
/// corresponds to the area on the form.</remarks>
void analyzer_ResultsUpdated(object sender, ResultsUpdatedEventArgs e)
{
    string recoText;

    recoText = hintNodeTitle.GetRecognizedString();
    if (recoText != "") xaml_blockTitle.Text = recoText;

    recoText = hintNodeDirector.GetRecognizedString();
    if (recoText != "") xaml_blockDirector.Text = recoText;

    recoText = hintNodeStarring.GetRecognizedString();
    if (recoText != "") xaml_blockStarring.Text = recoText;

    recoText = hintNodeRating.GetRecognizedString();
    if (recoText != "") xaml_blockRating.Text = recoText;

    recoText = hintNodeYear.GetRecognizedString();
    if (recoText != "") xaml_blockYear.Text = recoText;

    recoText = hintNodeGenre.GetRecognizedString();
    if (recoText != "") xaml_blockGenre.Text = recoText;
}

//Clear the canvas, but leave the current strokes in the analyzer.
private void ClearCanvas(object sender, EventArgs args)
{
    strokeRemovalTimer.Stop();

    xaml_writingCanvas.Strokes.Clear();
}
}
```

# 如何：旋转墨迹

项目 • 2023/02/06

## 旋转墨迹的示例

以下示例将墨迹从 `InkCanvas` 复制到包含 `InkPresenter` 的 `Canvas`。当应用程序复制墨迹时，它还会将墨迹顺时针旋转 90 度。

XAML

```
<Canvas>
    <InkCanvas Name="inkCanvas1" Background="LightBlue"
        Height="200" Width="200"
        Canvas.Top="20" Canvas.Left="20" />

    <Border Name="canvas1" Background="LightGreen"
        Height="200" Width="200" ClipToBounds="True"
        Canvas.Top="20" Canvas.Left="240" >
        <InkPresenter Name="inkPresenter1"/>
    </Border>
    <Button Click="button_Click"
        Canvas.Top="240" Canvas.Left="170">
        Copy and Rotate Strokes
    </Button>
</Canvas>
```

C#

```
// Button.Click event handler that rotates the strokes
// and copies them to a Canvas.
private void button_Click(object sender, RoutedEventArgs e)
{
    StrokeCollection copiedStrokes = inkCanvas1.Strokes.Clone();
    Matrix rotatingMatrix = new Matrix();
    double canvasLeft = Canvas.GetLeft(inkCanvas1);
    double canvasTop = Canvas.GetTop(inkCanvas1);
    Point rotatePoint = new Point(canvas1.Width / 2, canvas1.Height / 2);

    rotatingMatrix.RotateAt(90, rotatePoint.X, rotatePoint.Y);
    copiedStrokes.Transform(rotatingMatrix, false);
    inkPresenter1.Strokes = copiedStrokes;
}
```

## 笔划装饰器

以下示例是一个自定义 `Adorner`，它在 `InkPresenter` 上旋转笔划。

C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Controls.Primitives;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Shapes;
using System.Windows.Ink;

public class RotatingStrokesAdorner : Adorner
{
    // The Thumb to drag to rotate the strokes.
    Thumb rotateHandle;

    // The surrounding boarder.
    Path outline;

    VisualCollection visualChildren;

    // The center of the strokes.
    Point center;
    double lastAngle;

    RotateTransform rotation;

    const int HANDLEMARGIN = 10;

    // The bounds of the Strokes;
    Rect strokeBounds = Rect.Empty;

    public RotatingStrokesAdorner(UIElement adornedElement)
        : base(adornedElement)
    {

        visualChildren = new VisualCollection(this);
        rotateHandle = new Thumb();
        rotateHandle.Cursor = Cursors.SizeNWSE;
        rotateHandle.Width = 20;
        rotateHandle.Height = 20;
        rotateHandle.Background = Brushes.Blue;

        rotateHandle.DragDelta += new
        DragDeltaEventHandler(rotateHandle_DragDelta);
        rotateHandle.DragCompleted += new
        DragCompletedEventHandler(rotateHandle_DragCompleted);

        outline = new Path();
        outline.Stroke = Brushes.Blue;
        outline.StrokeThickness = 1;

        visualChildren.Add(outline);
    }

    void rotateHandle_DragDelta(object sender, DragDeltaEventArgs e)
    {
        Point currentCenter = center;
        Point delta = e.Delta;
        lastAngle += Math.Atan2(delta.Y, delta.X);
        center = new Point(currentCenter.X + delta.X * Math.Cos(lastAngle),
                           currentCenter.Y + delta.Y * Math.Sin(lastAngle));
        rotation.Center = center;
    }

    void rotateHandle_DragCompleted(object sender, DragCompletedEventArgs e)
    {
        if (e.HorizontalOffset != 0 || e.VerticalOffset != 0)
        {
            Point currentCenter = center;
            Point delta = e.Delta;
            lastAngle += Math.Atan2(delta.Y, delta.X);
            center = new Point(currentCenter.X + delta.X * Math.Cos(lastAngle),
                               currentCenter.Y + delta.Y * Math.Sin(lastAngle));
            rotation.Center = center;
        }
    }
}
```

```

        visualChildren.Add(rotateHandle);

        strokeBounds = AdornedStrokes.GetBounds();
    }

    /// <summary>
    /// Draw the rotation handle and the outline of
    /// the element.
    /// </summary>
    /// <param name="finalSize">The final area within the
    /// parent that this element should use to arrange
    /// itself and its children.</param>
    /// <returns>The actual size used. </returns>
protected override Size ArrangeOverride(Size finalSize)
{
    if (strokeBounds.IsEmpty)
    {
        return finalSize;
    }

    center = new Point(strokeBounds.X + strokeBounds.Width / 2,
                       strokeBounds.Y + strokeBounds.Height / 2);

    // The rectangle that determines the position of the Thumb.
    Rect handleRect = new Rect(strokeBounds.X,
                               strokeBounds.Y - (strokeBounds.Height / 2 +
                                                  HANDLEMARGIN),
                               strokeBounds.Width, strokeBounds.Height);

    if (rotation != null)
    {
        handleRect.Transform(rotation.Value);
    }

    // Draws the thumb and the rectangle around the strokes.
    rotateHandle.Arrange(handleRect);
    outline.Data = new RectangleGeometry(strokeBounds);
    outline.Arrange(new Rect(finalSize));
    return finalSize;
}

    /// <summary>
    /// Rotates the rectangle representing the
    /// strokes' bounds as the user drags the
    /// Thumb.
    /// </summary>
void rotateHandle_DragDelta(object sender, DragDeltaEventArgs e)
{
    // Find the angle of which to rotate the shape. Use the right
    // triangle that uses the center and the mouse's position
    // as vertices for the hypotenuse.

    Point pos = Mouse.GetPosition(this);

    double deltaX = pos.X - center.X;
}

```

```

        double deltaY = pos.Y - center.Y;

        if (deltaY.Equals(0))
        {

            return;
        }

        double tan = deltaX / deltaY;
        double angle = Math.Atan(tan);

        // Convert to degrees.
        angle = angle * 180 / Math.PI;

        // If the mouse crosses the vertical center,
        // find the complementary angle.
        if (deltaY > 0)
        {
            angle = 180 - Math.Abs(angle);
        }

        // Rotate left if the mouse moves left and right
        // if the mouse moves right.
        if (deltaX < 0)
        {
            angle = -Math.Abs(angle);
        }
        else
        {
            angle = Math.Abs(angle);
        }

        if (Double.IsNaN(angle))
        {
            return;
        }

        // Apply the rotation to the strokes' outline.
        rotation = new RotateTransform(angle, center.X, center.Y);
        outline.RenderTransform = rotation;
    }

    /// <summary>
    /// Rotates the strokes to the same angle as outline.
    /// </summary>
    void rotateHandle_DragCompleted(object sender,
                                    DragCompletedEventArgs e)
    {
        if (rotation == null)
        {
            return;
        }

        // Rotate the strokes to match the new angle.
        Matrix mat = new Matrix();

```

```

        mat.RotateAt(rotation.Angle - lastAngle, center.X, center.Y);
        AdornedStrokes.Transform(mat, true);

        // Save the angle of the last rotation.
        lastAngle = rotation.Angle;

        // Redraw rotateHandle.
        this.InvalidateArrange();
    }

    /// <summary>
    /// Gets the strokes of the adorned element
    /// (in this case, an InkPresenter).
    /// </summary>
    private StrokeCollection AdornedStrokes
    {
        get
        {
            return ((InkPresenter)AdornedElement).Strokes;
        }
    }

    // Override the VisualChildrenCount and
    // GetVisualChild properties to interface with
    // the adorner's visual collection.
    protected override int VisualChildrenCount
    {
        get { return visualChildren.Count; }
    }

    protected override Visual GetVisualChild(int index)
    {
        return visualChildren[index];
    }
}

```

以下示例是一个 Extensible Application Markup Language (XAML) 文件，它定义了 [InkPresenter](#) 并用墨迹填充它。 `Window_Loaded` 事件处理程序将自定义装饰器添加到 [InkPresenter](#)。

#### XAML

```

<Window x:Class="Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Rotating Strokes Adorner" Height="500" Width="500"
    Loaded="Window_Loaded"
    >
    <InkPresenter Name="inkPresenter1" >
        <InkPresenter.Strokes>
            ALMDAwRIEEU1BQE4GSAyCQD0/wIB6SI6RTMJAPifAgFaIDpFOAgA/gMAAACAfxEAAIA/
            HwkRAAAAAAAA8D8K1wE1h/CPd4SB4NA40icCjcClcDj8Lh8DgUSkUmmU6nUmoUuk
        </InkPresenter.Strokes>
    </InkPresenter>

```

```
0ukUCQKVyeHz+rzuly+bz0Rx+BReRQ+RTaRCH8JyXhPbgcPicPh8Pg80h0qk1SoVGrV
Oo0mi0Xi8rm9Xr9Dqc/p87pc/k8XicHic0j1CoVKtVmV1GqUaiUH1Yg8e14akXK7m7T
cSJgQgghEyyM5zx6+PACk4dhPwg/fhCbxY8dp4p2tqnqxyvbP085z1X1aswhvCd94Tq
55DRUGi4+Tk60Ln4KLko0ej06ig5KTioOPCD9L1HmrzNxMRCCc3ec8+fe4AKQBmE/Cw
9+FkPNv10dkrYsWa+acp3Z8er0IT8JaX4S6+FbFilbHNvvPXNJbFqluxghKc5DkwrVF
GEEIJ1w5eLKYAKShuF+Dnr40a8HVHXNPFFFFho8VFkqsMRYuuvJxiF+F9r4Xx8HFiqs
FNcirnweDw9+LvvvixdV0+GhONmlj3wjNOcSCEYTnfLy4oA
</InkPresenter.Strokes>
</InkPresenter>
</Window>
```

C#

```
void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Add the rotating strokes adorner to the InkPresenter.
    adornerLayer = AdornerLayer.GetAdornerLayer(inkPresenter1);
    adorner = new RotatingStrokesAdorner(inkPresenter1);

    adornerLayer.Add(adorner);
}
```

# 禁用用于 WPF 应用程序的 RealTimeStylus

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 具有处理 Windows 7 触控输入的内置支持。该支持通过平板电脑平台的实时触控输入（以 `OnStylusDown`、`OnStylusUp` 和 `OnStylusMove` 事件的形式）实现。Windows 7 还提供多点触控输入作为 Win32 `WM_TOUCH` 窗口消息。这两个 API 在同一 `HWND` 上互斥。通过平板电脑平台启用触控输入（WPF 应用程序的默认设置）会禁用 `WM_TOUCH` 消息。因此，若要使用 `WM_TOUCH` 从 WPF 窗口接收触控消息，必须禁用 WPF 中的内置触笔支持。这适用于托管使用 `WM_TOUCH` 组件的 WPF 窗口等方案。

若要禁用 WPF 侦听触笔输入，请删除 WPF 窗口添加的任何平板电脑支持。

## 示例

以下示例代码演示如何使用反射删除默认的平板电脑平台支持。

C#

```
public static void DisableWPFTabletSupport()
{
    // Get a collection of the tablet devices for this window.
    TabletDeviceCollection devices =
        System.Windows.Input.Tablet.TabletDevices;

    if (devices.Count > 0)
    {
        // Get the Type of InputManager.
        Type inputManagerType = typeof(System.Windows.Input.InputManager);

        // Call the StylusLogic method on the InputManager.Current instance.
        object stylusLogic = inputManagerType.InvokeMember("StylusLogic",
            BindingFlags.GetProperty | BindingFlags.Instance |
            BindingFlags.NonPublic,
            null, InputManager.Current, null);

        if (stylusLogic != null)
        {
            // Get the type of the stylusLogic returned from the call to
            // StylusLogic.
            Type stylusLogicType = stylusLogic.GetType();

            // Loop until there are no more devices to remove.
            while (devices.Count > 0)
            {

```

```
// Remove the first tablet device in the devices collection.
stylusLogicType.InvokeMember("OnTabletRemoved",
    BindingFlags.InvokeMethod | BindingFlags.Instance |
BindingFlags.NonPublic,
    null, stylusLogic, new object[] { (uint)0 });
}
}

}
```

## 另请参阅

- [截获触笔输入](#)

# 拖放

项目 • 2023/02/06

WPF 应用程序以及其他 Windows 应用程序。

## 本节内容

[拖放概述](#)

[数据和数据对象](#)

[演练：在用户控件上启用拖放功能](#)

[操作指南主题](#)

## 参考

[DataFormat](#)

[DataObject](#)

[DragDrop](#)

[DragDropEffects](#)

[DragEventHandler](#)

[TextDataFormat](#)

## 相关章节

# 拖放概述

项目 · 2023/02/06

本主题概述用户界面 (UI) 中的拖放支持，以及如何放置对象。

## WPF 中的拖放支持

拖放操作通常涉及两个参与方：拖动对象所源自的拖动源和接收放置对象的拖放目标。拖动源和放置目标可能是相同应用程序或不同应用程序中的 UI 元素。

可借助拖放操纵的对象的类型和数量是完全任意的。例如，文件、文件夹和内容选择是利用拖放操作操纵的一些常见对象。

拖放操作期间执行的特定操作特定于应用程序，并且通常由上下文而定。例如，将选择的文件从一个文件夹拖动至相同存储设备上的另一个文件夹将默认移动文件；而将文件从通用命名约定 (UNC) 共享拖动至本地文件夹将默认复制文件。

WPF 提供的拖放设施拥有高度的灵活性并可自定义，以便支持各种拖放方案。拖放支持在单个应用程序内或不同应用程序之间操作对象。也完全支持 WPF 应用程序和其它 Windows 应用程序之间的拖放。

在 WPF 中，任何 [UIElement](#) 或 [ContentElement](#) 都可以参与拖放。拖放操作所需的事件和方法是在 [DragDrop](#) 类中定义的。[UIElement](#) 和 [ContentElement](#) 类包含 [DragDrop](#) 附加事件的别名，从而在 [UIElement](#) 或 [ContentElement](#) 作为基元素继承时，这些事件出现在类成员列表中。附加到这些事件的事件处理程序会附加到基础 [DragDrop](#) 附加事件，并接收相同的事件数据实例。有关详细信息，请参阅 [UIElement.Drop](#) 事件。

### ① 重要

在 Internet 区域中 OLE 拖放无效。

## 数据传输

拖放属于广义的数据传输。数据传输包括拖放和复制粘贴操作。拖放操作类似于用于借助系统剪贴板将数据从一个对象或应用程序传输到另一个对象或应用程序的复制粘贴或剪切和粘贴操作。这两种类型的操作均要求：

- 提供数据的源对象。
- 用于临时存储传输的数据的方法。

- 接收数据的目标对象。

在复制粘贴操作中，系统剪贴板用于临时存储传输的数据；在拖放操作中，[DataObject](#) 用于存储数据。从概念上讲，数据对象由一对或多对包含实际数据的 [Object](#) 和对应的数据格式标识符组成。

拖动源通过调用静态 [DragDrop.DoDragDrop](#) 方法和向其传递传输的数据来启动拖放操作。如有必要，[DoDragDrop](#) 方法将使 [DataObject](#) 中的数据自动换行。为了更好地控制数据格式，可将 [DataObject](#) 中的数据换行，然后再将其传递至 [DoDragDrop](#) 方法。拖放目标负责从 [DataObject](#) 中提取数据。有关使用数据对象的详细信息，请参阅[数据和数据对象](#)。

拖放操作的源和目标均为 UI 元素；然而，实际正在传输的数据通常不具有可视表示形式。可以编写代码来提供拖动的数据的可视表示形式（比如当在 Windows 资源管理器中拖动文件时会出现这种情况）。默认情况下，通过更改光标将反馈提供给用户，以便表示拖放操作将对数据产生的影响，例如将移动数据还是复制数据。

## 拖放效果

拖放操作对传输的数据可具有不同的效果。例如，可以复制数据，或者可以移动数据。WPF 定义可用于指定拖放操作效果的 [DragDropEffects](#) 枚举。在拖动源中，可以指定源在 [DoDragDrop](#) 方法中允许的效果。在拖放目标中，可以指定目标在 [Effects](#) 类的 [DragEventArgs](#) 属性中的预期的效果。当拖放目标指定其在 [DragOver](#) 事件中的预期效果时，该信息将被传递回 [GiveFeedback](#) 事件中的拖动源。拖动源则使用此信息通知用户拖放目标想要对数据产生的效果。放置数据时，拖放目标指定其在 [Drop](#) 事件中的实际效果。该信息会作为 [DoDragDrop](#) 方法的返回值传递回拖动源。如果拖放目标返回并不在 [allowedEffects](#) 拖动源列表中的效果，那么将取消拖放操作，且不会进行任何数据传输。

请务必记住，在 WPF 中，[DragDropEffects](#) 值仅用于提供有关拖放操作效果的拖动源和拖放目标之间的通信。拖放操作的实际效果取决于你在应用程序中编写的相应代码。

例如，拖放目标可以指定在其中放置数据的效果是移动数据。然而，若要移动数据，必须将数据添加到目标元素并从源元素中删除数据。源元素可能指示允许移动数据，但是如果没有提供从源元素中删除数据的代码，那么最终结果将为复制但不删除数据。

## 拖放事件

拖放操作支持事件驱动模型。拖动源和拖放目标都使用一组标准的事件来处理拖放操作。下表总结了标准的拖放事件。它们是 [DragDrop](#) 类中的附加事件。有关附加事件的详细信息，请参阅[附加事件概述](#)。

## 拖动源事件

事件	总结
GiveFeedback	此事件在拖放操作期间持续发生，并且使放置源能够向用户提供反馈信息。通常通过更改鼠标指针外观来指示拖放目标允许的效果这一方式来提供这种反馈。这是冒泡事件。
QueryContinueDrag	此事件于拖放操作期间键盘或鼠标按钮状态发生变化时发生，并使放置源能够根据键/按钮状态取消拖放操作。这是冒泡事件。
PreviewGiveFeedback	GiveFeedback 的隧道版本。
PreviewQueryContinueDrag	QueryContinueDrag 的隧道版本。

## 拖放目标事件

事件	总结
DragEnter	将对象拖到拖放目标的边界中时发生此事件。这是冒泡事件。
DragLeave	将对象拖出拖放目标边界时发生此事件。这是冒泡事件。
DragOver	在拖放目标的边界内拖动（移动）对象时会持续发生此事件。这是冒泡事件。
Drop	将对象放置在拖放目标上时发生此事件。这是冒泡事件。
PreviewDragEnter	DragEnter 的隧道版本。
PreviewDragLeave	DragLeave 的隧道版本。
PreviewDragOver	DragOver 的隧道版本。
PreviewDrop	Drop 的隧道版本。

若要处理对象实例的拖放事件，请为上表中所列的事件添加处理程序。若要处理类级别的拖放事件，请重写相应的虚拟 On\* Event 和 On\* PreviewEvent 方法。有关详细信息，请参阅[按控件基类进行的路由事件类处理](#)。

## 实现拖放

UI 元素可以是拖动源、拖放目标或两者均可。若要实现基本拖放，请编写用于启动拖放操作和处理放置的数据的代码。可以通过处理可选拖放事件增强拖放体验。

若要实现基本拖放，将完成以下任务：

- 标识将作为拖动源的元素。 拖动源可以是 [UIElement](#) 或 [ContentElement](#)。
- 在将启动拖放操作的拖动源上创建事件处理程序。 此事件通常是 [MouseMove](#) 事件。
- 在拖动源事件处理程序中，调用 [DoDragDrop](#) 方法启动拖放操作。 在 [DoDragDrop](#) 调用中，指定拖动源、要传输的数据和允许的效果。
- 标识将作为拖放目标的元素。 拖放目标可以是 [UIElement](#) 或 [ContentElement](#)。
- 在拖放目标上，将 [AllowDrop](#) 属性设置为 `true`。
- 在拖放目标中，创建 [Drop](#) 事件处理程序以处理放置的数据。
- 在 [Drop](#) 事件处理程序中，利用 [DragEventArgs](#) 和 [GetDataPresent](#) 方法提取 [GetData](#) 中的数据。
- 在 [Drop](#) 事件处理程序中，使用数据来执行所需的拖放操作。

可以通过创建自定义 [DataObject](#) 和处理可选拖动源和拖放目标事件来增加拖放实现，如以下任务中所示：

- 若要传输自定义数据或多个数据项，请创建一个 [DataObject](#) 以传递至 [DoDragDrop](#) 方法。
- 若要在拖动过程中执行其他操作，请处理拖放目标上的 [DragEnter](#)[DragOver](#) 和 [DragLeave](#) 事件。
- 若要更改鼠标指针外观，请处理拖动源上的 [GiveFeedback](#) 事件。
- 若要更改取消拖放操作的方式，请处理拖动源上的 [QueryContinueDrag](#) 事件。

## 拖放示例

本节介绍如何实现 [Ellipse](#) 元素的拖放。[Ellipse](#) 既是拖动源也是拖放目标。 传输的数据是椭圆形的 [Fill](#) 属性的字符串表示形式。 下面的 XAML 展示 [Ellipse](#) 元素和它处理的拖放相关事件。 有关如何实现拖放的完整步骤，请参阅[演练：对用户控件启用拖放功能](#)。

XAML

```
<Ellipse Height="50" Width="50" Fill="Green"
    MouseMove="ellipse_MouseMove"
    GiveFeedback="ellipse_GiveFeedback"
    AllowDrop="True"
    DragEnter="ellipse_DragEnter" DragLeave="ellipse_DragLeave"
    DragOver="ellipse_DragOver" Drop="ellipse_Drop" />
```

# 使元素作为拖动源

拖动源对象用于：

- 标识拖动发生的时候。
- 启动拖放操作。
- 标识要传输的数据。
- 指定允许拖放操作对传输的数据产生的效果。

拖动源还可能针对允许的操作（移动、复制、无）向用户提供反馈，并且可以根据额外用户输入（如拖动过程中按 ESC 键）取消拖放操作。

你的应用程序负责确定发生拖动的时间，然后通过调用 [DoDragDrop](#) 方法启动拖放操作。通常情况下，这是在按下鼠标按钮的同时，要拖动的元素上发生 [MouseMove](#) 事件时。下面的示例显示了如何从 [Ellipse](#) 元素的 [MouseMove](#) 事件处理程序中启动拖放操作，以将其作为拖动源。传输的数据是椭圆形的 [Fill](#) 属性的字符串表示形式。

C#

```
private void ellipse_MouseMove(object sender, MouseEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null && e.LeftButton == MouseButtonState.Pressed)
    {
        DragDrop.DoDragDrop(ellipse,
                            ellipse.Fill.ToString(),
                            DragDropEffects.Copy);
    }
}
```

在 [MouseMove](#) 事件处理程序中，调用 [DoDragDrop](#) 方法启动拖放操作。[DoDragDrop](#) 方法采用三个参数：

- `dragSource` – 引用作为传输的数据的源的依赖项对象；通常是 [MouseMove](#) 事件的源。
- `data` – 包含传输的数据（包装在 [DataObject](#) 中）的对象。
- `allowedEffects` – 指定拖放操作允许的效果的 [DragDropEffects](#) 枚举值之一。

任何可序列化对象都可以在 `data` 参数中传递。如果数据尚未包装在 [DataObject](#) 中，则它将自动包装在一个新的 [DataObject](#) 中。若要传递多个数据项，必须自行创建 [DataObject](#)，并将其传递到 [DoDragDrop](#) 方法。有关详细信息，请参阅[数据和数据对象](#)。

`allowedEffects` 参数用于指定拖动源允许拖放目标对传输的数据进行什么操作。 拖动源公共值为 `CopyMove` 和 `All`。

### ① 备注

拖放目标也能够指定其对放置的数据的预期效果。 例如，如果拖放目标不能识别要放置的数据类型，则可以通过将其允许的效果设置为 `None` 来拒绝数据。 通常在其 `DragOver` 事件处理程序中进行此操作。

拖动源还可以可选地处理 `GiveFeedback` 和 `QueryContinueDrag` 事件。 这些事件都具有使用的默认处理程序，除非将事件标记为已处理。 通常将忽略这些事件，除非有更改其默认行为的特定需要。

对拖动源进行拖动时，持续引发 `GiveFeedback` 事件。 此事件的默认处理程序会检查拖动源是否在有效放置目标之上。 如果是，它会检查拖放目标的允许的效果。 然后向最终用户提供有关允许的放置效果的反馈。 通常通过将鼠标光标更改为非放置、复制或移动光标实现此操作。 仅在需要使用自定义光标向用户提供反馈时处理此事件。 在处理此事件时，请务必将其标记为“已处理”，以便默认处理程序不会替代你的处理程序。

对拖动源进行拖动时，持续引发 `QueryContinueDrag` 事件。 你可以根据 ESC、SHIFT、CTRL 和 ALT 键以及鼠标按钮的状态处理此事件，以确定结束拖放操作的操作。 此事件的默认处理程序在按下 ESC 键后取消拖放操作，并且在释放鼠标按钮后放置数据。

### ⊗ 注意

在拖放操作过程中，将持续引发这些事件。 因此，应避免事件处理程序中的资源密集型任务。 例如，每次引发 `GiveFeedback` 事件时，请使用缓存的光标，而不是创建新光标。

## 使元素作为拖放目标

作为拖放目标的对象用于：

- 指定其是有效的拖放目标。
- 当它拖动到目标之上时，向拖动源作出响应。
- 检查传输的数据是否是它可以接收的格式。
- 处理已放置的数据。

若要指定一个元素是拖放目标，请将其 `AllowDrop` 属性设置为 `true`。然后，元素中将引发拖放目标事件，以便处理这些事件。在拖放操作期间，拖放目标上将依次发生以下事件：

1. [DragEnter](#)
2. [DragOver](#)
3. [DragLeave 或 Drop](#)

将数据拖到拖放目标的边界中时发生 `DragEnter` 事件。通常，如果适用于你的应用程序，可处理此事件，以便提供拖放操作效果预览。请勿设置 `DragEventArgs.Effects` 事件中的 `DragEnter` 属性，因为在 `DragOver` 事件中该属性将被覆盖。

下面的示例演示 `DragEnter` 元素的 `Ellipse` 事件处理程序。此代码通过保存当前的 `Fill` 画笔预览拖放操作的效果。然后，它使用 `GetDataPresent` 方法来检查是否已将 `DataObject` 拖动到包含可以转换为 `Brush` 的字符串数据的椭圆上方。如果是，则使用 `GetData` 方法提取数据。然后将其转换为 `Brush` 并应用于椭圆。在 `DragLeave` 事件处理程序中还原更改。如果数据无法转换为 `Brush`，则不执行任何操作。

C#

```
private Brush _previousFill = null;
private void ellipse_DragEnter(object sender, DragEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null)
    {
        // Save the current Fill brush so that you can revert back to this
        // value in DragLeave.
        _previousFill = ellipse.Fill;

        // If the DataObject contains string data, extract it.
        if (e.Data.GetDataPresent(DataFormats.StringFormat))
        {
            string dataString =
                (string)e.Data.GetData(DataFormats.StringFormat);

            // If the string can be converted into a Brush, convert it.
            BrushConverter converter = new BrushConverter();
            if (converter.IsValid(dataString))
            {
                Brush newFill =
                    (Brush)converter.ConvertFromString(dataString);
                ellipse.Fill = newFill;
            }
        }
    }
}
```

将数据拖动到拖放目标上方时持续发生 [DragOver](#) 事件。此事件和拖动源上的 [GiveFeedback](#) 事件成对出现。在 [DragOver](#) 事件处理程序中，通常使用 [GetDataPresent](#) 和 [GetData](#) 方法来检查传输的数据是否是拖放目标可以处理的格式。还可以检查是否已按下修改键，这通常指示用户想进行移动操作还是复制操作。执行这些检查后，设置 [DragEventArgs.Effects](#) 属性以通知拖动源放置数据将产生的效果。拖动源收到 [GiveFeedback](#) 事件参数中的此信息，并且可以设置相应的光标以向用户提供反馈。

下面的示例演示 [DragOver](#) 元素的 [Ellipse](#) 事件处理程序。此代码检查是否已将 [DataObject](#) 拖动到包含可以转换为 [Brush](#) 的字符串数据的椭圆上方。如果是，它会将 [DragEventArgs.Effects](#) 属性设置为 [Copy](#)。这将向拖动源指示可以将数据复制到椭圆。如果数据无法转换为 [Brush](#)，则将 [DragEventArgs.Effects](#) 属性设置为 [None](#)。这将向拖动源指示椭圆不是数据的有效拖放目标。

C#

```
private void ellipse_DragOver(object sender, DragEventArgs e)
{
    e.Effects = DragDropEffects.None;

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString =
(string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush, allow copying.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            e.Effects = DragDropEffects.Copy | DragDropEffects.Move;
        }
    }
}
```

将数据拖出目标边界而未放置时，发生 [DragLeave](#) 事件。可以处理此事件，以便撤销在 [DragEnter](#) 事件处理程序中进行的一切操作。

下面的示例演示 [DragLeave](#) 元素的 [Ellipse](#) 事件处理程序。此代码通过将保存的 [Brush](#) 应用到椭圆来撤销 [DragEnter](#) 事件处理程序中执行的预览。

C#

```
private void ellipse_DragLeave(object sender, DragEventArgs e)
{
    Ellipse ellipse = sender as Ellipse;
    if (ellipse != null)
    {
        ellipse.Fill = _previousFill;
```

```
    }  
}
```

数据放置在拖放目标上方时，发生 [Drop](#) 事件；默认情况下，释放鼠标按钮时，发生此事件。在 [Drop](#) 事件处理程序中，使用 [GetData](#) 方法提取 [DataObject](#) 中的传输的数据并执行应用程序所需的任何数据处理。[Drop](#) 事件结束拖放操作。

下面的示例演示 [Drop](#) 元素的 [Ellipse](#) 事件处理程序。此代码应用拖放操作的效果，并且它类似于 [DragEnter](#) 事件处理程序中的代码。它会检查是否将 [DataObject](#) 拖动到包含可以转换为 [Brush](#) 的字符串数据的椭圆上方。如果是，将 [Brush](#) 应用于椭圆。如果数据无法转换为 [Brush](#)，则不执行任何操作。

C#

```
private void ellipse_Drop(object sender, DragEventArgs e)  
{  
    Ellipse ellipse = sender as Ellipse;  
    if (ellipse != null)  
    {  
        // If the DataObject contains string data, extract it.  
        if (e.Data.GetDataPresent(DataFormats.StringFormat))  
        {  
            string dataString =  
(string)e.Data.GetData(DataFormats.StringFormat);  
  
            // If the string can be converted into a Brush,  
            // convert it and apply it to the ellipse.  
            BrushConverter converter = new BrushConverter();  
            if (converter.IsValid(dataString))  
            {  
                Brush newFill =  
(Brush)converter.ConvertFromString(dataString);  
                ellipse.Fill = newFill;  
            }  
        }  
    }  
}
```

## 另请参阅

- [Clipboard](#)
- [演练：在用户控件上启用拖放功能](#)
- [操作指南主题](#)
- [拖放](#)

# 数据和数据对象

项目 · 2023/02/06

作为拖放操作的一部分传输的数据存储在数据对象中。从概念上讲，数据对象由以下一个或多个对组成：

- 包含实际数据的 [Object](#)。
- 相应的数据格式标识符。

数据本身可以包含任何可以表示为基础映像 [Object](#) 的内容。相应数据格式是一个字符串或 [Type](#)，它提供关于数据所采用的格式的提示。数据对象支持托管多个数据/数据格式对，这使得单个数据对象可以提供多种格式的数据。

## 数据对象

所有数据对象必须实现 [IDataObject](#) 接口，该接口提供以下标准方法集，用于启用和促进数据传输。

方法	总结
<a href="#">GetData</a>	检索指定数据格式的数据对象。
<a href="#">GetDataPresent</a>	查看是否具有指定格式的数据，或者数据是否可以转换为指定格式。
<a href="#">GetFormats</a>	返回一个格式列表，此数据对象中的数据以这些格式存储，或可以转换为这些格式。
<a href="#">SetData</a>	在此数据对象中存储指定的数据。

WPF 在 [DataObject](#) 类中提供 [IDataObject](#) 的基本实现。普通的 [DataObject](#) 类足以满足许多常见的数据传输场景。

有多种预定义的格式，例如位图、CSV、文件、HTML、RTF、字符串、文本和音频。有关 WPF 提供的预定义数据格式的信息，请参阅 [DataFormats](#) 类参考主题。

数据对象通常包括在提取数据时自动将存储在一种格式中的数据转换为另一种格式的功能，此功能称为自动转换。查询数据对象中可用的数据格式时，通过调用 [GetFormats\(Boolean\)](#) 或 [GetDataPresent\(String, Boolean\)](#) 方法并将参数指定 `autoConvert` 为 `false` 可以从本机数据格式中筛选出自动转换的数据格式。使用 [SetData\(String, Object, Boolean\)](#) 方法将数据添加到数据对象时，可以通过将参数 `autoConvert` 设置为 `false` 来禁止数据的自动转换。

# 使用数据对象

本部分介绍创建和使用数据对象的常用技术。

## 创建新数据对象

[DataObject](#) 类提供了多个重载构造函数，这些构造函数有助于使用单个数据/数据格式对填充新 [DataObject](#) 实例。

以下示例代码创建一个新的数据对象，并使用重载构造函数 [DataObject](#) 之一 ([DataObject\(String, Object\)](#))，以字符串和指定的数据格式初始化数据对象。在这种情况下，数据格式由字符串指定，[DataFormats](#) 类提供一组预定义的类型字符串。默认情况下，允许自动转换存储的数据。

C#

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.UnicodeText;  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

有关创建数据对象的更多代码示例，请参阅[创建数据对象](#)。

## 以多种格式存储数据

单个数据对象都能以多种格式存储数据。在单个数据对象中有策略地使用多种数据格式，可能会使数据对象被更广泛的放置目标所使用，而不是只能表示单个数据格式。请注意，一般情况下，拖动源必须与潜在放置目标可使用的数据格式无关。

以下示例演示了如何使用 [SetData\(String, Object\)](#) 方法以多种格式将数据添加到数据对象。

C#

```
DataObject dataObject = new DataObject();  
string sourceData = "Some string data to store...";  
  
// Encode the source string into Unicode byte arrays.  
byte[] unicodeText = Encoding.Unicode.GetBytes(sourceData); // UTF-16  
byte[] utf8Text = Encoding.UTF8.GetBytes(sourceData);  
byte[] utf32Text = Encoding.UTF32.GetBytes(sourceData);  
  
// The DataFormats class does not provide data format fields for denoting  
// UTF-32 and UTF-8, which are seldom used in practice; the following  
strings  
// will be used to identify these "custom" data formats.  
string utf32DataFormat = "UTF-32";
```

```
string utf8DataFormat = "UTF-8";

// Store the text in the data object, letting the data object choose
// the data format (which will be DataFormats.Text in this case).
dataObject.SetData(sourceData);
// Store the Unicode text in the data object. Text data can be
// automatically
// converted to Unicode (UTF-16 / UCS-2) format on extraction from the data
// object;
// Therefore, explicitly converting the source text to Unicode is generally
// unnecessary, and
// is done here as an exercise only.
dataObject.SetData(DataFormats.UnicodeText, unicodeText);
// Store the UTF-8 text in the data object...
dataObject.SetData(utf8DataFormat, utf8Text);
// Store the UTF-32 text in the data object...
dataObject.SetData(utf32DataFormat, utf32Text);
```

## 查询数据对象以获取可用格式

由于单个数据对象可以包含任意数量的数据格式，因此数据对象包括用于检索可用数据格式列表的功能。

以下示例代码使用 [GetFormats](#) 重载来获取表示数据对象中可用的所有数据格式的字符串数组（在本机或者通过自动转换）。

C#

```
DataObject dataObject = new DataObject("Some string data to store...");

// Get an array of strings, each string denoting a data format
// that is available in the data object. This overload of GetDataFormats
// returns all available data formats, native and auto-convertible.
string[] dataFormats = dataObject.GetFormats();

// Get the number of data formats present in the data object, including both
// auto-convertible and native data formats.
int numberOfDataFormats = dataFormats.Length;

// To enumerate the resulting array of data formats, and take some action
when
// a particular data format is found, use a code structure similar to the
following.
foreach (string dataFormat in dataFormats)
{
    if (dataFormat == DataFormats.Text)
    {
        // Take some action if/when data in the Text data format is found.
        break;
    }
    else if(dataFormat == DataFormats.StringFormat)
```

```
{  
    // Take some action if/when data in the string data format is found.  
    break;  
}  
}
```

有关查询数据对象以获取可用数据格式的更多代码示例，请参阅[列出数据对象中的数据格式](#)。有关查询数据对象是否存在特定数据格式的示例，请参阅[确定数据对象中是否存在数据格式](#)。

## 从数据对象检索数据

从特定格式的数据对象检索数据只需调用 [GetData](#) 方法之一并指定所需的数据格式。可使用 [GetDataPresent](#) 方法之一检查是否存在特定的数据格式。[GetData](#) 返回 [Object](#) 中的数据。根据数据格式的不同，可以将此对象强制转换为特定于类型的容器。

以下示例代码使用 [GetDataPresent\(String\)](#) 重载来检查指定的数据格式是否可用（在本机或者通过自动转换）。如果指定格式可用，则该示例使用 [GetData\(String\)](#) 方法检索数据。

C#

```
DataObject dataObject = new DataObject("Some string data to store...");  
  
string desiredFormat = DataFormats.UnicodeText;  
byte[] data = null;  
  
// Use the GetDataPresent method to check for the presence of a desired data  
format.  
// This particular overload of GetDataPresent looks for both native and  
auto-convertible  
// data formats.  
if (dataObject.GetDataPresent(desiredFormat))  
{  
    // If the desired data format is present, use one of the GetData methods  
    // to retrieve the  
    // data from the data object.  
    data = dataObject.GetData(desiredFormat) as byte[];  
}
```

有关从数据对象检索数据的代码的更多示例，请参阅[以特定数据格式检索数据](#)。

## 从数据对象中删除数据

无法直接从数据对象中删除数据。要有效地从数据对象中删除数据，请执行以下步骤：

1. 创建一个新的数据对象，该对象仅包含要保留的数据。

2. 将所需数据从旧数据对象“复制”到新数据对象。要复制数据，请使用 `GetData` 方法之一检索包含原始数据的 `Object`，然后使用 `SetData` 方法之一将数据添加到新数据对象中。
3. 将旧数据对象替换为新数据对象。

#### ① 备注

`SetData` 方法仅向数据对象添加数据；即使数据和数据格式与前一个调用完全相同，它们也不会替换数据。对于相同的数据和数据格式，调用 `SetData` 两次将导致数据/数据格式在数据对象中出现两次。

# 演练：在用户控件上启用拖放功能

项目 • 2023/02/06

本演练演示如何创建可在 Windows Presentation Foundation (WPF) 中参与拖放数据传输的自定义用户控件。

在本演练中，将创建一个表示圆形的自定义 WPF [UserControl](#)。你将在该控件上实现可通过拖放进行数据传输的功能。例如，如果从一个圆形控件拖到另一个圆形控件，则会将填充颜色数据从源圆形复制到目标圆形。如果从一个圆形控件拖到 [TextBox](#)，则填充颜色的字符串表示形式将复制到 [TextBox](#)。你还将创建一个小应用程序，该应用程序包含两个面板控件和一个 [TextBox](#)，用以测试拖放功能。你将编写可使面板处理放置的圆形数据的代码，这样就可以将圆形从一个面板的 Children 集合移动或复制到其他面板。

本演练阐释了以下任务：

- 创建自定义用户控件。
- 使用用户控件成为拖动源。
- 使用用户控件成为拖放目标。
- 使面板能够接收用户控件中放置的数据。

## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 创建应用程序项目

在本节中，你将创建应用程序基础结构，其中包括一个具有两个面板和一个 [TextBox](#) 的主页。

1. 在 Visual Basic 或 Visual C# 中创建名为 `DragDropExample` 的新 WPF 应用程序项目。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。
2. 打开 `MainWindow.xaml`。
3. 在开始和结束 [Grid](#) 标记间添加以下标记。

此标记将创建用于测试应用程序的用户界面。

XAML

```
<Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
</Grid.ColumnDefinitions>
<StackPanel Grid.Column="0"
            Background="Beige">
    <TextBox Width="Auto" Margin="2"
             Text="green"/>
</StackPanel>
<StackPanel Grid.Column="1"
            Background="Bisque">
</StackPanel>
```

## 向项目添加新的用户控件

本节将介绍如何向项目添加新的用户控件。

1. 在“项目”菜单中，选择“添加用户控件”。
2. 在“添加新项”对话框中，将名称更改为 `Circle.xaml`，然后单击“添加”。

`Circle.xaml` 及其代码隐藏内容将添加到项目中。

3. 打开 `Circle.xaml`。

此文件将包含用户控件的用户界面元素。

4. 将以下标记添加到根 `Grid`，以创建将蓝色圆形作为其 UI 的简单用户控件。

### XAML

```
<Ellipse x:Name="circleUI"
         Height="100" Width="100"
         Fill="Blue" />
```

5. 打开 `Circle.xaml.cs` 或 `Circle.xaml.vb`。

6. 在 C# 中，在无参数构造函数后面添加以下代码以创建复制构造函数。在 Visual Basic 中，添加以下代码以同时创建无参数构造函数和复制构造函数。

若要允许复制用户控件，需在代码隐藏文件中添加复制构造函数方法。在简化的圆形用户控件中，将只复制该用户控件的填充和大小。

### C#

```
public Circle(Circle c)
{
```

```
InitializeComponent();
this.circleUI.Height = c.circleUI.Height;
this.circleUI.Width = c.circleUI.Height;
this.circleUI.Fill = c.circleUI.Fill;
}
```

## 向主窗口添加用户控件

1. 打开 MainWindow.xaml。
2. 将以下 XAML 添加到开始 `Window` 标记以创建对当前应用程序的 XML 命名空间引用。

XAML

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

3. 在第一个 `StackPanel` 中，添加以下 XAML，以在第一个面板中创建圆形用户控件的两个实例。

XAML

```
<local:Circle Margin="2" />
<local:Circle Margin="2" />
```

此面板的完整 XAML 如下所示。

XAML

```
<StackPanel Grid.Column="0"
           Background="Beige">
    <TextBox Width="Auto" Margin="2"
             Text="green"/>
    <local:Circle Margin="2" />
    <local:Circle Margin="2" />
</StackPanel>
<StackPanel Grid.Column="1"
           Background="Bisque">
</StackPanel>
```

## 在用户控件中实现拖动源事件

在本节中，将替代 `OnMouseMove` 方法并启动拖放操作。

如果已开始拖动（按下鼠标按钮并移动鼠标），则会将要传输的数据打包到 `DataObject` 中。在这种情况下，圆形控件将打包三个数据项：其填充颜色的字符串表示形式、其高度的双精度表示形式以及其自身的副本。

## 启动拖放操作

1. 打开 `Circle.xaml.cs` 或 `Circle.xaml.vb`。
2. 添加以下 `OnMouseMove` 替代，以便为 `MouseMove` 事件提供类处理。

C#

```
protected override void OnMouseMove(MouseEventArgs e)
{
    base.OnMouseMove(e);
    if (e.LeftButton == MouseButtonState.Pressed)
    {
        // Package the data.
        DataObject data = new DataObject();
        data.SetData(DataFormats.StringFormat,
circleUI.Fill.ToString());
        data.SetData("Double", circleUI.Height);
        data.SetData("Object", this);

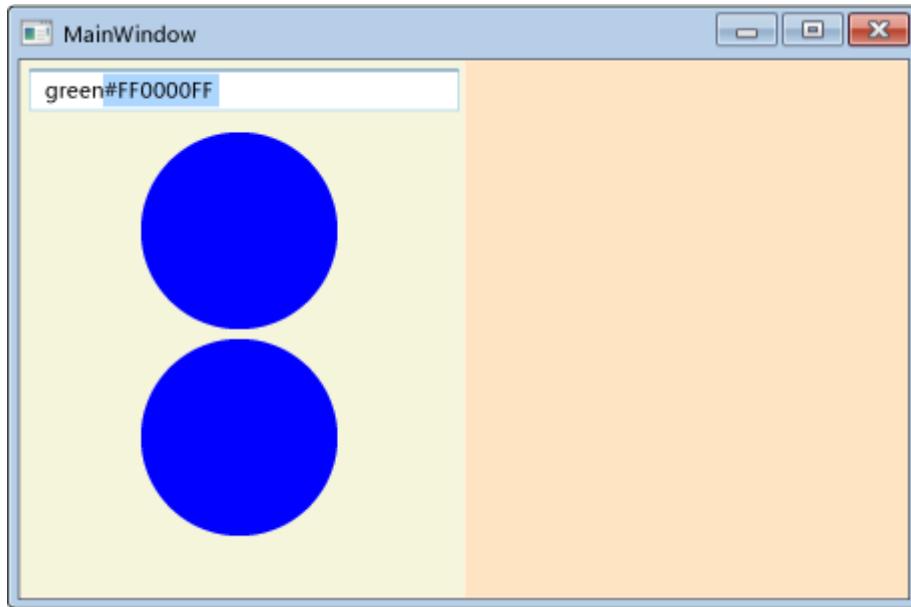
        // Initiate the drag-and-drop operation.
        DragDrop.DoDragDrop(this, data, DragDropEffects.Copy |
DragDropEffects.Move);
    }
}
```

此 `OnMouseMove` 替代执行下列任务：

- 检查移动鼠标时是否按下了鼠标左键。
- 将圆形数据打包到 `DataObject`。在这种情况下，圆形控件将打包三个数据项：其填充颜色的字符串表示形式、其高度的双精度表示形式以及其自身的副本。
- 调用静态 `DragDrop.DoDragDrop` 方法启动拖放操作。向 `DoDragDrop` 方法传递以下三个参数：
  - `dragSource` – 对此控件的引用。
  - `data` – 上一个示例中创建的 `DataObject`。
  - `allowedEffects` – 允许的拖放操作，即 `Copy` 或 `Move`。

3. 按 F5 生成并运行应用程序。

4. 单击一个圆形控件并将其拖到面板、另一个圆形和 TextBox 上。拖到 TextBox 上时，光标会更改，以指示移动。
5. 将圆形拖到 TextBox 上时，按 Ctrl 键。请注意光标是如何更改以指示复制的。
6. 将圆形拖放到 TextBox 上。该圆形填充颜色的字符串表示形式会追加到 TextBox。



默认情况下，光标会在拖放操作过程中更改，以指示放置数据会产生的效果。可通过处理 GiveFeedback 事件并设置不同光标来自定义向用户提供的反馈。

## 向用户提供反馈

1. 打开 Circle.xaml.cs 或 Circle.xaml.vb。
2. 添加以下 OnGiveFeedback 替代，以便为 GiveFeedback 事件提供类处理。

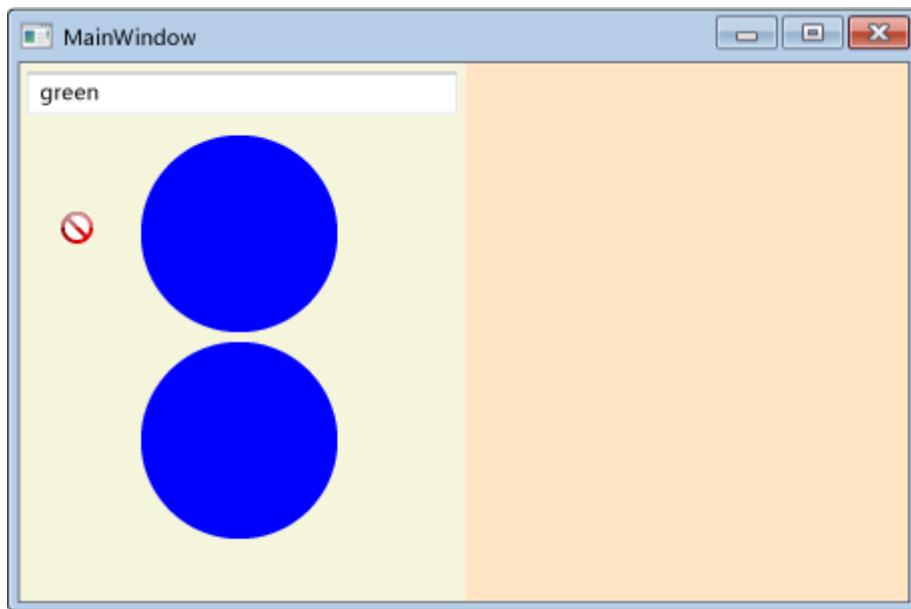
```
C#
protected override void OnGiveFeedback(GiveFeedbackEventArgs e)
{
    base.OnGiveFeedback(e);
    // These Effects values are set in the drop target's
    // DragOver event handler.
    if (e.Effects.HasFlag(DragDropEffects.Copy))
    {
        Mouse.SetCursor(Cursors.Cross);
    }
    else if (e.Effects.HasFlag(DragDropEffects.Move))
    {
        Mouse.SetCursor(Cursors.Pen);
    }
    else
    {
        Mouse.SetCursor(Cursors.No);
    }
}
```

```
        }  
        e.Handled = true;  
    }  
}
```

此 [OnGiveFeedback](#) 替代执行下列任务：

- 检查在拖放目标的 [DragOver](#) 事件处理程序中设置的 [Effects](#) 值。
- 基于 [Effects](#) 值设置自定义光标。该光标旨在向用户提供关于放置数据所产生的效果的可视反馈。

3. 按 F5 生成并运行应用程序。
4. 将一个圆形控件拖到面板、另一个圆形和 [TextBox](#) 上。请注意，现在的光标是在 [OnGiveFeedback](#) 替代中指定的自定义光标。



5. 从 [TextBox](#) 中选择文本 `green`。
6. 将 `green` 文本拖到一个圆形控件上。请注意，将显示默认光标以指示拖放操作效果。反馈光标始终由拖动源设置。

## 在用户控件中实现拖放目标事件

在本节中，你将指定用户控件为拖放目标，替代可使用户控件成为拖放目标的方法，并处理用户控件上放置的数据。

### 使用用户控件成为拖放目标

1. 打开 Circle.xaml。

2. 在开始 `UserControl` 标记中，添加 `AllowDrop` 属性并将其设置为 `true`.

XAML

```
<UserControl x:Class="DragDropWalkthrough.Circle"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:mc="http://schemas.openxmlformats.org/markup-
compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        mc:Ignorable="d"
        d:DesignHeight="300" d:DesignWidth="300"
        AllowDrop="True">
```

当 `AllowDrop` 属性设为 `true` 并且拖动源的数据放置在圆形用户控件上时，就会调用 `OnDrop` 方法。在这种方法中，将处理已放置的数据，并将这些数据应用于圆形。

## 处理已放置的数据

1. 打开 `Circle.xaml.cs` 或 `Circle.xaml.vb`。
2. 添加以下 `OnDrop` 替代，以便为 `Drop` 事件提供类处理。

C#

```
protected override void OnDrop(DragEventArgs e)
{
    base.OnDrop(e);

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString =
(string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush,
        // convert it and apply it to the ellipse.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            Brush newFill =
(Brush)converter.ConvertFromString(dataString);
            circleUI.Fill = newFill;

            // Set Effects to notify the drag source what effect
            // the drag-and-drop operation had.
            // (Copy if CTRL is pressed; otherwise, move.)
            if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
```

```
        {
            e.Effects = DragDropEffects.Copy;
        }
        else
        {
            e.Effects = DragDropEffects.Move;
        }
    }
    e.Handled = true;
}
```

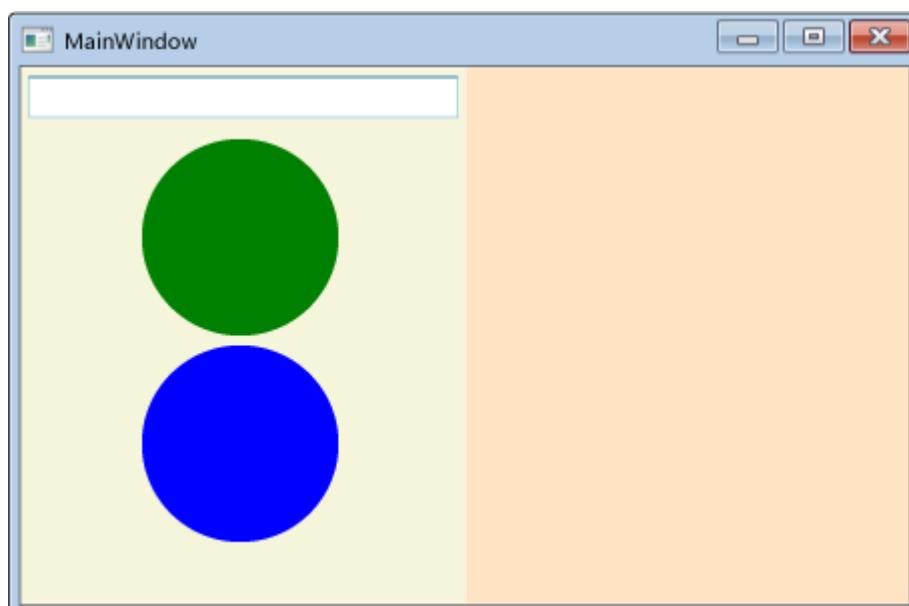
此 [OnDrop](#) 替代执行下列任务：

- 使用 [GetDataPresent](#) 方法检查拖动的数据是否包含字符串对象。
- 如果存在，请使用 [GetData](#) 方法提取字符串数据。
- 使用 [BrushConverter](#) 尝试将字符串转换为 [Brush](#)。
- 如果转换成功，则将画笔应用于提供圆形控件的 UI 的 [Ellipse](#) 的 [Fill](#)。
- 将 [Drop](#) 事件标记为已处理。应将放置事件标记为已处理，这样接收此事件的其他元素才会知道圆形用户控件已处理了该事件。

3. 按 F5 生成并运行应用程序。

4. 在 [TextBox](#) 中选择文本 [green](#)。

5. 将文本拖放到一个圆形控件上。该圆形会从蓝色变为绿色。



6. 在 [green](#) 中键入文本 [TextBox](#)。

7. 在 [TextBox](#) 中选择文本 [gre](#)。

8. 将其拖放到一个圆形控件上。请注意，光标会更改以指示允许放置，但圆形的颜色不会更改，因为 `gre` 不是有效颜色。

9. 从绿色圆形控件拖放到蓝色圆形控件。该圆形会从蓝色变为绿色。请注意，显示的光标取决于 `TextBox` 还是圆形作为拖动源。

若要使某个元素成为拖放目标，只需将 `AllowDrop` 属性设置为 `true` 并处理放置的数据。但是，若要提供更佳的用户体验，还应处理 `DragEnter`、`DragLeave` 和 `DragOver` 事件。在这些事件中，你可以在放置数据前执行检查并向用户提供其他反馈。

将数据拖动到圆形用户控件上时，该控件应通知拖动源它是否可以处理所拖动的数据。如果该控件不知如何处理这些数据，则它应拒绝放置。为此，你将处理 `DragOver` 事件并设置 `Effects` 属性。

## 验证是否允许数据放置

1. 打开 `Circle.xaml.cs` 或 `Circle.xaml.vb`。

2. 添加以下 `OnDragOver` 替代，以便为 `DragOver` 事件提供类处理。

C#

```
protected override void OnDragOver(DragEventArgs e)
{
    base.OnDragOver(e);
    e.Effects = DragDropEffects.None;

    // If the DataObject contains string data, extract it.
    if (e.Data.GetDataPresent(DataFormats.StringFormat))
    {
        string dataString =
(string)e.Data.GetData(DataFormats.StringFormat);

        // If the string can be converted into a Brush, allow copying
        // or moving.
        BrushConverter converter = new BrushConverter();
        if (converter.IsValid(dataString))
        {
            // Set Effects to notify the drag source what effect
            // the drag-and-drop operation will have. These values are
            // used by the drag source's GiveFeedback event handler.
            // (Copy if CTRL is pressed; otherwise, move.)
            if (e.KeyStates.HasFlag(DragDropKeyStates.ControlKey))
            {
                e.Effects = DragDropEffects.Copy;
            }
            else
            {
                e.Effects = DragDropEffects.Move;
            }
        }
    }
}
```

```
        }
    }
    e.Handled = true;
}
```

此 `OnDragOver` 替代执行下列任务：

- 将 `Effects` 属性设置为 `None`。
- 执行在 `OnDrop` 方法中执行的相同检查，以确定圆形用户控件是否可以处理拖动的数据。
- 如果用户控件可以处理数据，将 `Effects` 属性设置为 `Copy` 或 `Move`。

3. 按 F5 生成并运行应用程序。
4. 在 `TextBox` 中选择文本 `gre`。
5. 将文本拖到一个圆形控件上。请注意，光标此时会更改以指示不允许放置，因为 `gre` 不是有效颜色。

可通过应用拖放操作预览进一步增强用户体验。对于圆形用户控件，将替代 `OnDragEnter` 和 `OnDragLeave` 方法。将数据拖动到该控件上时，当前背景 `Fill` 会保存在一个占位符变量中。随后字符串会转换为画笔并应用于提供圆形 UI 的 `Ellipse`。如果将数据拖动出该圆形而没有放置，则原始 `Fill` 值将重新应用于该圆形。

## 预览拖放操作的效果

1. 打开 `Circle.xaml.cs` 或 `Circle.xaml.vb`。
2. 在圆形类中，可声明一个名为 `_previousFill` 的私有 `Brush` 变量，并将其初始化为 `null`。

```
C#
public partial class Circle : UserControl
{
    private Brush _previousFill = null;
```

3. 添加以下 `OnDragEnter` 替代，以便为 `DragEnter` 事件提供类处理。

```
C#
protected override void OnDragEnter(DragEventArgs e)
{
```

```

        base.OnDragEnter(e);
        // Save the current Fill brush so that you can revert back to this
        // value in DragLeave.
        _previousFill = circleUI.Fill;

        // If the DataObject contains string data, extract it.
        if (e.Data.GetDataPresent(DataFormats.StringFormat))
        {
            string dataString =
(string)e.Data.GetData(DataFormats.StringFormat);

            // If the string can be converted into a Brush, convert it.
            BrushConverter converter = new BrushConverter();
            if (converter.IsValid(dataString))
            {
                Brush newFill =
(Brush)converter.ConvertFromString(dataString.ToString());
                circleUI.Fill = newFill;
            }
        }
    }
}

```

此 [OnDragEnter](#) 替代执行下列任务：

- 将 [Ellipse](#) 的 [Fill](#) 属性保存在 [\\_previousFill](#) 变量中。
- 执行在 [OnDrop](#) 方法中执行的相同检查，以确定是否可将数据转换为 [Brush](#)。
- 如果数据可转换为有效的 [Brush](#)，则将其应用于 [Ellipse](#) 的 [Fill](#)。

4. 添加以下 [OnDragLeave](#) 替代，以便为 [DragLeave](#) 事件提供类处理。

C#

```

protected override void OnDragLeave(DragEventArgs e)
{
    base.OnDragLeave(e);
    // Undo the preview that was applied in OnDragEnter.
    circleUI.Fill = _previousFill;
}

```

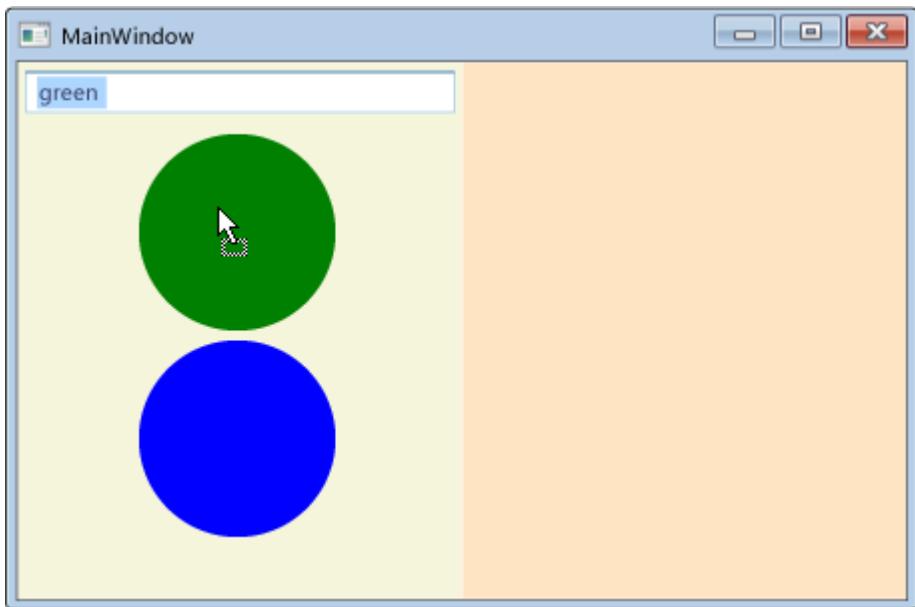
此 [OnDragLeave](#) 替代执行下列任务：

- 将保存在 [\\_previousFill](#) 变量中的 [Brush](#) 应用于提供圆形用户控件 UI 的 [Ellipse](#) 的 [Fill](#)。

5. 按 F5 生成并运行应用程序。

6. 在 [TextBox](#) 中选择文本 [green](#)。

7. 将该文本拖到一个圆形控件上而不放置。 该圆形会从蓝色变为绿色。



8. 将文本拖离该圆形控件。 该圆形会从绿色变回蓝色。

## 使面板能够接收放置的数据

在本节中，让承载圆形用户控件的面板充当日拖动圆形数据的拖放目标。 将实现的代码使你可以将圆形从一个面板移动到另一个面板，或通过在拖放圆形时按住 Ctrl 键来复制圆形控件。

1. 打开 MainWindow.xaml。

2. 如以下 XAML 所示，在每个 StackPanel 控件中，为 DragOver 和 Drop 事件添加处理程序。 将 DragOver 事件处理程序命名为 `panel_DragOver`，并将 Drop 事件处理程序命名为 `panel_Drop`。

默认情况下，面板不会删除目标。 若要启用它们，请将 AllowDrop 属性添加到两个面板，并将值设置为 `true`。

XAML

```
<StackPanel Grid.Column="0"
            Background="Beige"
            AllowDrop="True"
            DragOver="panel_DragOver"
            Drop="panel_Drop">
    <TextBox Width="Auto" Margin="2"
            Text="green"/>
    <local:Circle Margin="2" />
    <local:Circle Margin="2" />
</StackPanel>
<StackPanel Grid.Column="1"
```

```
Background="Bisque"
AllowDrop="True"
DragOver="panel_DragOver"
Drop="panel_Drop">
</StackPanel>
```

3. 打开 MainWindows.xaml.cs 或 MainWindow.xaml.vb。

4. 为 DragOver 事件处理程序添加以下代码。

C#

```
private void panel_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent("Object"))
    {
        // These Effects values are used in the drag source's
        // GiveFeedback event handler to determine which cursor to
        display.
        if (e.KeyStates == DragDropKeyStates.ControlKey)
        {
            e.Effects = DragDropEffects.Copy;
        }
        else
        {
            e.Effects = DragDropEffects.Move;
        }
    }
}
```

此 DragOver 事件处理程序执行以下任务：

- 检查拖动的数据是否包含由圆形用户控件打包在 DataObject 中并且在 DoDragDrop 调用中传递的“对象”数据。
- 如果存在“对象”数据，请检查是否按下了 Ctrl 键。
- 如果按下了 Ctrl 键，则将 Effects 属性设置为 Copy。否则将 Effects 属性设置为 Move。

5. 为 Drop 事件处理程序添加以下代码。

C#

```
private void panel_Drop(object sender, DragEventArgs e)
{
    // If an element in the panel has already handled the drop,
    // the panel should not also handle it.
    if (e.Handled == false)
    {
```

```

        Panel _panel = (Panel)sender;
        UIElement _element = (UIElement)e.Data.GetData("Object");

        if (_panel != null && _element != null)
        {
            // Get the panel that the element currently belongs to,
            // then remove it from that panel and add it the Children
            of
            // the panel that its been dropped on.
            Panel _parent =
            (Panel)VisualTreeHelper.GetParent(_element);

            if (_parent != null)
            {
                if (e.KeyStates == DragDropKeyStates.ControlKey &&
                    e.AllowedEffects.HasFlag(DragDropEffects.Copy))
                {
                    Circle _circle = new Circle((Circle)_element);
                    _panel.Children.Add(_circle);
                    // set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Copy;
                }
                else if
                (e.AllowedEffects.HasFlag(DragDropEffects.Move))
                {
                    _parent.Children.Remove(_element);
                    _panel.Children.Add(_element);
                    // set the value to return to the DoDragDrop call
                    e.Effects = DragDropEffects.Move;
                }
            }
        }
    }
}

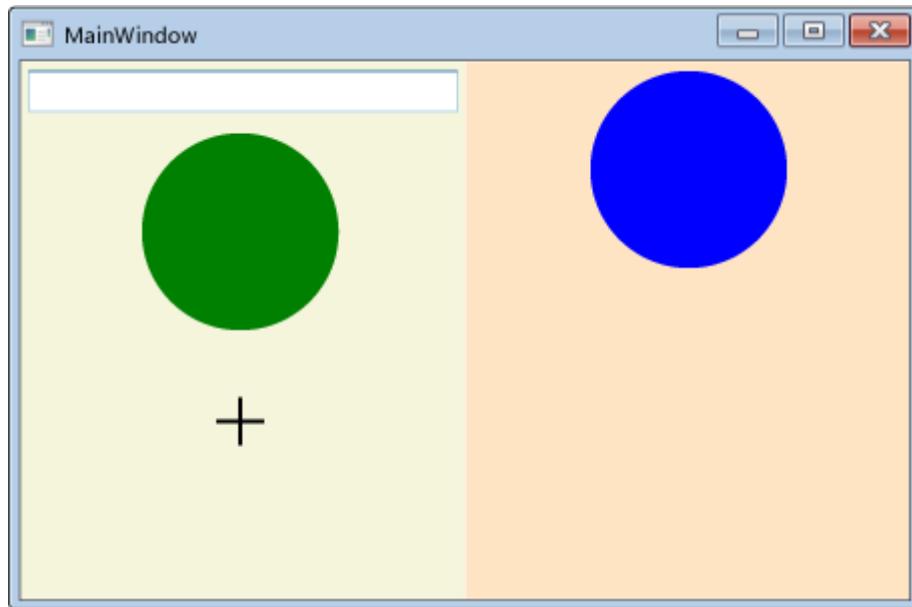
```

此 Drop 事件处理程序执行以下任务：

- 检查是否已处理 Drop 事件。例如，如果将一个圆形放置在处理 Drop 事件的另一个圆形上，则无需让包含该圆形的面板也处理该事件。
- 如果未处理 Drop 事件，请检查是否按下了 Ctrl 键。
- 如果在发生 Drop 时按下了 Ctrl 键，则会创建圆形控件的副本并将其添加到 StackPanel 的 Children 集合。
- 如果未按下 Ctrl 键，则将圆形从其父面板的 Children 集合移动到放置该圆形的面板的 Children 集合。
- 设置 Effects 属性以通知 DoDragDrop 方法是执行了移动还是复制操作。

6. 按 F5 生成并运行应用程序。

7. 从 [TextBox](#) 中选择文本 `green`。
8. 将文本拖放到一个圆形控件上。
9. 将一个圆形控件从左面板拖放到右面板。 该圆形将从左面板的 [Children](#) 集合中移除，并添加到右面板的 [Children](#) 集合中。
10. 在按下 [Ctrl](#) 键的同时，将一个圆形控件从其所在的面板拖放到其他面板。 将复制该圆形并将副本添加到接收面板的 [Children](#) 集合中。



## 另请参阅

- [拖放概述](#)

# 拖放帮助主题

项目 • 2023/02/06

以下示例演示如何使用 Windows Presentation Foundation (WPF) 拖放框架完成常见任务。

## 本节内容

[打开放入 RichTextBox 控件的文件](#)

[创建数据对象](#)

[确定数据格式是否存在于数据对象中](#)

[列出数据对象中的数据格式](#)

[以特定数据格式检索数据](#)

[在数据对象中存储多种数据格式](#)

## 另请参阅

- [拖放概述](#)

# 如何：打开放置在 RichTextBox 控件上的文件

项目 • 2022/09/27

在 Windows Presentation Foundation (WPF) 中，[TextBox](#)、[RichTextBox](#) 和 [FlowDocument](#) 控件都具有内置的拖放功能。此内置功能支持在控件内和控件之间拖放文本。但是，它不支持将文件拖放到控件上来打开文件。这些控件还会将拖放事件标记为已处理。因此，默认情况下，你不能通过添加自己的事件处理程序来提供打开拖放文件的功能。

若要在这些控件中添加对拖放事件的其他处理，请使用 [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 方法来为拖放事件添加事件处理程序。将 `handledEventsToo` 参数设置为 `true`，从而为已标记为由事件路由中的其他元素处理的路由事件调用指定的处理程序。

## 提示

可以通过处理拖放事件的预览版本并将预览事件标记为已处理来取代 [TextBox](#)、[RichTextBox](#) 和 [FlowDocument](#) 的内置拖放功能。但是，这将禁用内置的拖放功能，不建议这样做。

## 示例

下面的示例演示了如何为 [RichTextBox](#) 上的 [DragOver](#) 和 [Drop](#) 事件添加处理程序。此示例使用 [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 方法并将 `handledEventsToo` 参数设置为 `true`，这样一来，事件处理程序将被调用，即使 [RichTextBox](#) 将这些事件标记为已处理也是如此。事件处理程序中的代码添加了打开拖放到 [RichTextBox](#) 上的文本文件的功能。

若要测试此示例，请将文本文件或 RTF 格式文件从 Windows 资源管理器拖到 [RichTextBox](#) 中。该文件将在 [RichTextBox](#) 中打开。如果在放置文件之前按 Shift 键，该文件将作为纯文本打开。

### XAML

```
<RichTextBox x:Name="richTextBox1"  
    AllowDrop="True" />
```

### C#

```
public MainWindow()
{
    InitializeComponent();

    // Add using System.Windows.Controls;
    richTextBox1.AddHandler(RichTextBox.DragOverEvent, new
    DragEventHandler(RichTextBox_DragOver), true);
    richTextBox1.AddHandler(RichTextBox.DropEvent, new
    DragEventHandler(RichTextBox_Drop), true);
}

private void RichTextBox_DragOver(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        e.Effects = DragDropEffects.All;
    }
    else
    {
        e.Effects = DragDropEffects.None;
    }
    e.Handled = false;
}

private void RichTextBox_Drop(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.FileDrop))
    {
        string[] docPath = (string[])e.Data.GetData(DataFormats.FileDrop);

        // By default, open as Rich Text (RTF).
        var dataFormat = DataFormats.Rtf;

        // If the Shift key is pressed, open as plain text.
        if (e.KeyStates == DragDropKeyStates.ShiftKey)
        {
            dataFormat = DataFormats.Text;
        }

        System.Windows.Documents.TextRange range;
        System.IO.FileStream fStream;
        if (System.IO.File.Exists(docPath[0]))
        {
            try
            {
                // Open the document in the RichTextBox.
                range = new
                System.Windows.Documents.TextRange(richTextBox1.Document.ContentStart,
                richTextBox1.Document.ContentEnd);
                fStream = new System.IO.FileStream(docPath[0],
                System.IO FileMode.OpenOrCreate);
                range.Load(fStream, dataFormat);
                fStream.Close();
            }
        }
    }
}
```

```
        catch (System.Exception)
        {
            MessageBox.Show("File could not be opened. Make sure the
file is a text file.");
        }
    }
}
```

# 如何：创建数据对象

项目 • 2023/02/06

以下示例演示了使用 [DataObject](#) 类提供的构造函数来创建数据对象的各种方法。

## DataObject(对象) 构造函数

### 描述

以下示例代码创建一个新的数据对象，并使用重载构造函数 ([DataObject\(Object\)](#))，以初始化带字符串的数据对象。在这种情况下，根据存储的数据类型自动确定适当的数据格式，默认情况下允许自动转换存储的数据。

### 代码

C#

```
string stringData = "Some string data to store...";  
DataObject dataObject = new DataObject(stringData);
```

### 说明

以下示例代码是上面所示代码的精简版本。

### 代码

C#

```
DataObject dataObject = new DataObject("Some string data to store...");
```

## DataObject (字符串、对象) 构造函数

### 描述

以下示例代码创建一个新的数据对象，并使用重载构造函数 ([DataObject\(String, Object\)](#))，以初始化带字符串和指定数据格式的数据对象。在这种情况下，数据格式由字

字符串指定；[DataFormats](#) 类提供一组预定义的类型字符串。 默认情况下，允许自动转换存储的数据。

## 代码

C#

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.UnicodeText;  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

## 说明

以下示例代码是上面所示代码的精简版本。

## 代码

C#

```
DataObject dataObject = new DataObject(DataFormats.UnicodeText, "Some string  
data to store...");
```

## DataObject () 构造函数

## 描述

以下示例代码创建一个新的数据对象，并使用重载构造函数 ([DataObject](#))，以初始化带字符串和指定数据格式的数据对象。在这种情况下，数据格式由 [Type](#) 参数指定。默认情况下，允许自动转换存储的数据。

## 代码

C#

```
string stringData = "Some string data to store...";  
Type dataFormat = stringData.GetType();  
DataObject dataObject = new DataObject(dataFormat, stringData);
```

## 说明

以下示例代码是上面所示代码的精简版本。

## 代码

C#

```
DataObject dataObject = new DataObject("".GetType(), "Some string data to store...");
```

# DataObject (字符串、 对象、 布尔) 构造函数

## 描述

以下示例代码创建一个新的数据对象，并使用重载构造函数 ([DataObject\(String, Object, Boolean\)](#))，以初始化带字符串和指定数据格式的数据对象。在这种情况下，数据格式由字符串指定；[DataFormats](#) 类提供一组预定义的类型字符串。此特定构造函数重载使调用方能够指定是否允许自动转换。

## 代码

C#

```
string stringData = "Some string data to store...";  
string dataFormat = DataFormats.Text;  
bool autoConvert = false;  
DataObject dataObject = new DataObject(dataFormat, stringData, autoConvert);
```

## 说明

以下示例代码是上面所示代码的精简版本。

## 代码

C#

```
DataObject dataObject = new DataObject(DataFormats.Text, "Some string data to store...", false);
```

## 另请参阅

- [IDataObject](#)

# 如何：确定数据格式是否存在于数据对象中

项目 • 2023/02/06

以下示例演示如何使用各种 `GetDataPresent` 方法重载来查询数据对象中是否存在特定数据格式。

## GetDataPresent(String) 重载

### 描述

以下示例代码使用 `GetDataPresent(String)` 重载按描述符字符串查询是否存在特定数据格式。

### 代码

C#

```
DataObject dataObject = new DataObject("Some string data to store...");

// Query for the presence of Text data in the data object, by a data format
descriptor string.
// In this overload of GetDataPresent, the method will return true both for
native data formats
// and when the data can automatically be converted to the specified format.

// In this case, string data is present natively, so GetDataPresent returns
"true".
string textData = null;
if (dataObject.GetDataPresent(DataFormats.StringFormat))
{
    textData = dataObject.GetData(DataFormats.StringFormat) as string;
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, so GetDataPresent returns "true".
byte[] unicodeData = null;
if (dataObject.GetDataPresent(DataFormats.UnicodeText))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}
```

## GetDataPresent(Type) 重载

## 描述

以下示例代码使用 [GetDataPresent\(Type\)](#) 重载按类型查询是否存在特定数据格式。

## 代码

C#

```
DataContract dataObject = new DataObject("Some string data to store...");  
  
// Query for the presence of String data in the data object, by type. In  
this overload  
// of GetDataPresent, the method will return true both for native data  
formats  
// and when the data can automatically be converted to the specified format.  
  
// In this case, the Text data present in the data object can be  
autoconverted  
// to type string (also represented by DataFormats.String), so  
GetDataPresent returns "true".  
string stringData = null;  
if (dataObject.GetDataPresent(typeof(string)))  
{  
    stringData = dataObject.GetData(DataFormats.Text) as string;  
}
```

## GetDataPresent(String, Boolean) 重载

## 描述

以下示例代码使用 [GetDataPresent\(String, Boolean\)](#) 重载按描述符字符串查询数据，并指定如何处理可自动转换的数据格式。

## 代码

C#

```
DataContract dataObject = new DataObject("Some string data to store...");  
  
// Query for the presence of Text data in the data object, by data format  
descriptor string,  
// and specifying whether auto-convertible data formats are acceptable.  
  
// In this case, Text data is present natively, so GetDataPresent returns  
"true".  
string textData = null;
```

```
if (dataObject.GetDataPresent(DataFormats.Text, false /* Auto-convert? */))
{
    textData = dataObject.GetData(DataFormats.Text) as string;
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, but it is not available natively, so GetDataPresent returns
//"false".
byte[] unicodeData = null;
if (dataObject.GetDataPresent(DataFormats.UnicodeText, false /* Auto-
convert? */))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}

// In this case, the Text data in the data object can be autoconverted to
// Unicode text, so GetDataPresent returns "true".
if (dataObject.GetDataPresent(DataFormats.UnicodeText, true /* Auto-convert?
*/))
{
    unicodeData = dataObject.GetData(DataFormats.UnicodeText) as byte[];
}
```

## 另请参阅

- [IDataObject](#)

# 如何：列出数据对象中的数据格式

项目 • 2023/02/06

以下示例演示如何使用 [GetFormats](#) 方法重载来获取表示数据对象中可用的每个数据格式的字符串数组。

## 针对所有数据格式的 GetFormats() 重载

### 描述

以下示例代码使用 [GetFormats](#) 重载来获取表示数据对象中可用的所有数据格式的字符串数组（本机和自动转换）。

### 代码

C#

```
DataObject dataObject = new DataObject("Some string data to store...");

// Get an array of strings, each string denoting a data format
// that is available in the data object. This overload of GetDataFormats
// returns all available data formats, native and auto-convertible.
string[] dataFormats = dataObject.GetFormats();

// Get the number of data formats present in the data object, including both
// auto-convertible and native data formats.
int numberOfDataFormats = dataFormats.Length;

// To enumerate the resulting array of data formats, and take some action
// when
// a particular data format is found, use a code structure similar to the
// following.
foreach (string dataFormat in dataFormats)
{
    if (dataFormat == DataFormats.Text)
    {
        // Take some action if/when data in the Text data format is found.
        break;
    }
    else if(dataFormat == DataFormats.StringFormat)
    {
        // Take some action if/when data in the string data format is found.
        break;
    }
}
```

# 针对本机数据格式的 GetFormats() 重载

## 描述

以下示例代码使用 [GetFormats](#) 重载来获取表示仅数据对象中可用的数据格式的字符串数组（筛选自动转换数据格式）。

## 代码

C#

```
DataObject dataObject = new DataObject("Some string data to store...");  
  
// Get an array of strings, each string denoting a data format  
// that is available in the data object. This overload of GetDataFormats  
// accepts a Boolean parameter indicating whether to include auto-  
// convertible  
// data formats, or only return native data formats.  
string[] dataFormats = dataObject.GetFormats(false /* Include auto-  
convertible? */);  
  
// Get the number of native data formats present in the data object.  
int numberOfDataFormats = dataFormats.Length;  
  
// To enumerate the resulting array of data formats, and take some action  
when  
// a particular data format is found, use a code structure similar to the  
following.  
foreach (string dataFormat in dataFormats)  
{  
    if (dataFormat == DataFormats.Text)  
    {  
        // Take some action if/when data in the Text data format is found.  
        break;  
    }  
}
```

## 另请参阅

- [IDataObject](#)
- [拖放概述](#)

# 如何：以特定数据格式检索数据

项目 • 2023/02/06

以下示例演示如何以指定格式的从数据对象检索数据。

## 使用 `GetDataPresent`（字符串）重载检索数据

### 描述

下面的示例代码使用 `GetDataPresent(String)` 重载首先检查指定的数据格式是否可用（在本机或者通过自动转换）；如果指定格式可用，则该示例使用 `GetData(String)` 方法检索数据。

### 代码

C#

```
 DataObject dataObject = new DataObject("Some string data to store...");

 string desiredFormat = DataFormats.UnicodeText;
 byte[] data = null;

 // Use the GetDataPresent method to check for the presence of a desired data
 // format.
 // This particular overload of GetDataPresent looks for both native and
 // auto-convertible
 // data formats.
 if (dataObject.GetDataPresent(desiredFormat))
 {
    // If the desired data format is present, use one of the GetData methods
    // to retrieve the
    // data from the data object.
    data = dataObject.GetData(desiredFormat) as byte[];
 }
```

## 使用 `GetDataPresent`（字符串、布尔）重载检索数据

### 描述

下面的示例代码使用 [GetDataPresent\(String, Boolean\)](#) 重载首先检查指定的数据格式是否在本机可用（自动转换数据格式被筛选）；如果指定格式可用，则该示例使用 [GetData\(String\)](#) 方法检索数据。

## 代码

C#

```
 DataObject dataObject = new DataObject("Some string data to store...");  
  
 string desiredFormat = DataFormats.UnicodeText;  
 bool noAutoConvert = false;  
 byte[] data = null;  
  
 // Use the GetDataPresent method to check for the presence of a desired data  
 // format.  
 // The autoconvert parameter is set to false to filter out auto-convertible  
 // data formats,  
 // returning true only if the specified data format is available natively.  
 if (dataObject.GetDataPresent(desiredFormat, noAutoConvert))  
 {  
     // If the desired data format is present, use one of the GetData methods  
     // to retrieve the  
     // data from the data object.  
     data = dataObject.GetData(desiredFormat) as byte[];  
 }
```

## 另请参阅

- [IDataObject](#)
- [拖放概述](#)

# 如何：在数据对象中存储多种数据格式

项目 • 2023/02/06

以下示例演示了如何使用 [SetData\(String, Object\)](#) 方法以多种格式将数据添加到数据对象。

## 示例

### 说明

### 代码

C#

```
DataObject dataObject = new DataObject();
string sourceData = "Some string data to store...";

// Encode the source string into Unicode byte arrays.
byte[] unicodeText = Encoding.Unicode.GetBytes(sourceData); // UTF-16
byte[] utf8Text = Encoding.UTF8.GetBytes(sourceData);
byte[] utf32Text = Encoding.UTF32.GetBytes(sourceData);

// The DataFormats class does not provide data format fields for denoting
// UTF-32 and UTF-8, which are seldom used in practice; the following
strings
// will be used to identify these "custom" data formats.
string utf32DataFormat = "UTF-32";
string utf8DataFormat = "UTF-8";

// Store the text in the data object, letting the data object choose
// the data format (which will be DataFormats.Text in this case).
dataObject.SetData(sourceData);
// Store the Unicode text in the data object. Text data can be
automatically
// converted to Unicode (UTF-16 / UCS-2) format on extraction from the data
object;
// Therefore, explicitly converting the source text to Unicode is generally
unnecessary, and
// is done here as an exercise only.
dataObject.SetData(DataFormats.UnicodeText, unicodeText);
// Store the UTF-8 text in the data object...
dataObject.SetData(utf8DataFormat, utf8Text);
// Store the UTF-32 text in the data object...
dataObject.SetData(utf32DataFormat, utf32Text);
```

## 另请参阅

- [IDataObject](#)
- [拖放概述](#)

# 资源 (WPF)

项目 • 2023/02/06

资源是可以在应用程序中的不同位置重复使用的对象。 WPF 支持不同类型的资源。这些资源主要有两种类型：XAML 资源和资源数据文件。 XAML 资源的示例包括画笔和样式。 资源数据文件是应用程序所需的非可执行数据文件。

## 本节内容

[XAML 资源](#)

[WPF 应用程序资源、内容和数据文件](#)

[WPF 中的 Pack URI](#)

## 参考

[ResourceDictionary](#)

[StaticResource 标记扩展](#)

[DynamicResource 标记扩展](#)

[x:Key 指令](#)

## 相关章节

[WPF 中的 XAML](#)

# XAML 资源概述

项目 • 2022/09/27

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。

## 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

```
<Page Name="root"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Page.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="Background" Value="Blue"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="Background" Value="Blue"/>
            <Setter Property="DockPanel.Dock" Value="Top"/>
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
            <Setter Property="FontFamily" Value="Trebuchet MS"/>
            <Setter Property="Margin" Value="0,40,10,10"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="Label">
            <Setter Property="DockPanel.Dock" Value="Right"/>
            <Setter Property="FontSize" Value="8"/>
            <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>
    </Page.Resources>
    <StackPanel>
```

```

<Border Style="{StaticResource PageBackground}">
    <DockPanel>
        <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
        <TextBlock Style="{StaticResource Label}">Label</TextBlock>
        <TextBlock DockPanel.Dock="Top" HorizontalAlignment="Left"
FontSize="36" Foreground="{StaticResource MyBrush}" Text="Text" Margin="20"
/>
        <Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30"
Background="{StaticResource MyBrush}" Margin="40">Button</Button>
        <Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100"
Height="100" Fill="{StaticResource MyBrush}" Margin="40" />
    </DockPanel>
</Border>
</StackPanel>
</Page>

```

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 [Page](#)。

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，即 [Page](#)。如果在 [Page](#) 根元素上定义资源，则 [Page](#) 的逻辑树中的所有元素都可以访问它。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类型的属性的所有属性的值。在前面的示例中，同一 `MyBrush` 资源设置两个不同的属性：[Button](#) 的 [Background](#) 和 [Rectangle](#) 的 [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，表达式会进行计算并提供一个值。

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。
- 要创建的资源字典将编译成 DLL，并将打包为应用的一部分或在应用间共享。
- 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，通常不希望执行动态资源引用查找行为，而是希望执行静态资源引用行为，以确保查找可预测并包含到主题中。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，主题的行为将偏离预期方式。
- 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：

- 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。
- 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。
- 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，则异常会在加载时发生。

## 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[样式设置和模板化](#)。

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">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0.0" Color="AliceBlue"/>
                <GradientStop Offset="1.0" Color="Salmon"/>
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
    <Setter Property="FontSize" Value="18"/>
</Style>
```

该样式确实具有一个键：隐式键 `typeof(System.Windows.Controls.Button)`。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](#)。

## 请参阅

- [ResourceDictionary](#)
- [应用程序资源](#)
- [资源和代码](#)
- [定义和引用资源](#)
- [应用程序管理概述](#)
- [x:Type 标记扩展](#)
- [StaticResource 标记扩展](#)
- [DynamicResource 标记扩展](#)



# 资源和代码

项目 · 2023/02/06

本概述重点介绍 XAML 语法透视的方式，请参阅 [XAML 资源](#)。

## 从代码访问资源

用于识别通过 XAML 定义的资源的键也用于检索特定资源（如果你在代码中请求此资源）。从代码检索资源的最简单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当未找到所请求的键时所发生的情况。[FindResource](#) 引发异常；[TryFindResource](#) 不引发异常，而是返回 `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。资源键通常是字符串，但有时也用作非字符串；有关详细信息，请参阅[将对象用作键](#)部分。通常应将返回的对象强制转换为请求资源时设置的属性所要求的类型。代码资源解析的查找逻辑与动态资源引用 XAML 情况相同。对资源的搜索从调用元素开始，然后继续搜索逻辑树中的后续父元素。如果必要，将继续查找应用程序资源、主题以及系统资源。资源的代码请求将正确地说明资源字典（可能排在从 XAML 加载的资源字典之后）中的运行时更改，也将说明实时系统资源更改。

下面是一段简短的代码示例，该示例通过键查找资源并使用返回的值来设置属性，该示例是作为 [Click](#) 事件处理程序实现的。

C#

```
void SetBGByResource(object sender, RoutedEventArgs e)
{
    Button b = sender as Button;
    b.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。通过访问该属性包含的字典，还可以向现有集合添加新资源、检查集合中是否已经存在给定的键名称以及执行其他字典/集合操作。如果完全通过代码方式编写 WPF 应用程序，则也可以通过代码创建整个集合，为其分配键，然后将完成的集合分配给已建立的元素的 [Resources](#) 属性。这将在下一部分介绍。

可以在任何给定的 [Resources](#) 集合内编制索引（将特定键用作索引），但你应当知道以这种方式访问资源未遵循资源解析的常规运行时规则。你访问的仅是该特定集合。如果在

请求的键处未找到有效的对象，则资源查找将不会遍历范围直至根或应用程序。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。有关如何直接处理资源字典的更多详细信息，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，然后使用对 [ResourceDictionary.Add](#) 的连续调用将所有资源添加到字典中。然后使用创建的 [ResourceDictionary](#) 来设置位于页范围内的元素的 [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，各个 WPF 功能都特意不使用字符串类型来指定键，而是将此参数设置为对象。WPF 样式和主题支持使用按对象对资源进行键控的功能。主题中成为非样式控件的默认样式的样式都将按它们应当应用于的控件的 [Type](#) 来进行键控。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，即使派生类型不具有默认样式，也可以通过反射检测到类型，并将类型用于设置派生类的样式。可以为 WPF 功能中定义的资源指定一个 [Type](#) 键，例如 [ComponentResourceKey Markup Extension](#)。

## 另请参阅

- [XAML 资源](#)
- [样式设置和模板化](#)

# 合并资源字典

项目 · 2023/02/06

已编译的 XAML 应用程序之外的 WPF 应用程序。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。

## 引入合并资源字典

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>
```

请注意，`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。引入合并资源字典的 `ResourceDictionary` 中不能有 `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，则通常应在主要 `ResourceDictionary` 中（而不是合并字典中）定义资源。

## 合并字典的行为

合并字典中的资源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重复出现。

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。默认情况下，任何 `Resources` 属性的初始为空的 `ResourceDictionary` 都具有初始为空的默认 `MergedDictionaries` 集合属性。若要通过代码添加合并字典，可获取对所需主要 `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对象来获取 `ResourceDictionary` 对象。一种方法，用于在具有 `ResourceDictionary` 根的现有 XAML 文件流上加载现有 `ResourceDictionary` 的 `XamlReader.Load` 调用，然后将 `XamlReader.Load` 返回值强制转换为 `ResourceDictionary`。

## 合并资源字典 URI

可通过几种技术来包括合并资源字典，具体由将使用的统一资源标识符 (URI) 格式指示。这些技术可概括地分为两类：作为项目的一部分编译的资源，和不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，因为资源已包括在编译的应用程序中。也可以使用“内容”生成操作，但必须将文件复制到输出目录，还需将相同路径关系中的资源文件部署到可执行文件。

### ① 备注

不要使用“嵌入资源”生成操作。WPF 应用程序支持该生成操作本身，但 `Source` 的解析不会合并 `ResourceManager`，因此不能将单个资源分离到流外。如果还使用 `ResourceManager` 访问资源，则仍可将“嵌套资源”用于其他用途。

还有一种相关技术是使用指向 XAML 文件的 Pack URI，并将其作为“源”进行引用。Pack URI 可启用对引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](#)。

对于不作为项目的一部分编译的资源，URI 将在运行时计算。可使用常见的 URI 传输（如 file: 或 http:）引用资源文件。使用非编译资源方法的缺点是：file: 访问需要额外部署步骤，并且 http: 访问意味着需访问 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和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，通常可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。还有可能通过使用 http 协议的分布式 URI 引用合并资源。

另一种合并字典/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](#)
- [资源和代码](#)
- [WPF 应用程序资源、内容和数据文件](#)

# 资源帮助主题

项目 · 2023/02/06

本节中的主题介绍如何使用 Windows Presentation Foundation (WPF) 资源。

## 本节内容

[定义和引用资源](#)

[使用应用程序资源](#)

[使用 SystemFonts](#)

[使用系统字体键](#)

[使用 SystemParameters](#)

[使用系统参数键](#)

## 参考

[Resources](#)

[SystemColors](#)

[SystemParameters](#)

[SystemFonts](#)

## 相关章节

[XAML 资源](#)

# 如何：定义和引用资源

项目 • 2022/09/27

此示例演示如何使用 Extensible Application Markup Language (XAML) 中的属性定义资源并引用它。

## 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。

`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。`Style` 资源 `PageBackground`、`TitleText` 和 `Label` 每个都针对特定的控件类型。当样式资源由资源键引用并用于设置 XAML 中定义的几个特定控件元素的 `Style` 属性时，样式会在目标控件上设置各种不同的属性。

请注意，`Label` 样式的资源库中的属性之一也引用了之前定义的 `MyBrush` 资源。这是一种常见的技术，但重要的是要记住，资源是按照给定的顺序解析并输入到资源字典中的。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。请确保引用的任何资源在资源集合中的定义早于随后请求该资源的位置。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 [XAML 资源](#)。

XAML

```
<Page Name="root"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Page.Resources>
        <SolidColorBrush x:Key="MyBrush" Color="Gold"/>
        <Style TargetType="Border" x:Key="PageBackground">
            <Setter Property="Background" Value="Blue"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="TitleText">
            <Setter Property="Background" Value="Blue"/>
            <Setter Property="DockPanel.Dock" Value="Top"/>
            <Setter Property="FontSize" Value="18"/>
            <Setter Property="Foreground" Value="#4E87D4"/>
            <Setter Property="FontFamily" Value="Trebuchet MS"/>
            <Setter Property="Margin" Value="0,40,10,10"/>
        </Style>
        <Style TargetType="TextBlock" x:Key="Label">
            <Setter Property="DockPanel.Dock" Value="Right"/>
            <Setter Property="FontSize" Value="8"/>
            <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>
            <Setter Property="FontFamily" Value="Arial"/>
        </Style>
    </Page.Resources>

```

```
<Setter Property="FontWeight" Value="Bold"/>
<Setter Property="Margin" Value="0,3,10,0"/>
</Style>
</Page.Resources>
<StackPanel>
<Border Style="{StaticResource PageBackground}">
<DockPanel>
<TextBlock Style="{StaticResource TitleText}">Title</TextBlock>
<TextBlock Style="{StaticResource Label}">Label</TextBlock>
<TextBlock DockPanel.Dock="Top" HorizontalAlignment="Left"
FontSize="36" Foreground="{StaticResource MyBrush}" Text="Text" Margin="20"
/>
<Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30"
Background="{StaticResource MyBrush}" Margin="40">Button</Button>
<Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100"
Height="100" Fill="{StaticResource MyBrush}" Margin="40" />
</DockPanel>
</Border>
</StackPanel>
</Page>
```

## 另请参阅

- [XAML 资源](#)
- [样式设置和模板化](#)

# 如何：使用应用程序资源

项目 • 2023/02/06

本示例演示如何使用应用程序资源。

## 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="Button" x:Key="GelButton" >
        <Setter Property="Margin" Value="1,2,1,2"/>
        <Setter Property="HorizontalAlignment" Value="Left"/>
        <Setter Property="Template">
            <Setter.Value>
```

XAML

```
        </Setter.Value>
    </Setter>
</Style>
</Application.Resources>
```

下面的示例演示了引用上例中定义的应用程序级资源的 XAML 页。 通过使用 [StaticResource 标记扩展](#)（用于指定所请求资源的唯一资源键）引用该资源。 在当前页中没有找到具有“GelButton”键的资源，所以请求资源的资源查找范围超出当前页，进入已定义的应用程序级资源。

XAML

```
<StackPanel
    Name="root"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Button Height="50" Width="250" Style="{StaticResource GelButton}"
    Content="Button 1" />
    <Button Height="50" Width="250" Style="{StaticResource GelButton}"
    Content="Button 2" />
</StackPanel>
```

## 另请参阅

- [XAML 资源](#)
- [应用程序管理概述](#)
- [操作指南主题](#)

# 如何：使用 SystemFonts

项目 • 2023/02/06

此示例演示如何使用 [SystemFonts](#) 类的静态资源设置按钮样式或自定义按钮。

## 示例

系统资源将系统确定的许多值以资源和属性的形式公开，以帮助创建与系统设置一致的视觉效果。[SystemFonts](#) 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，[CaptionFontFamilyKey](#) 是相应的资源键。

在 XAML 中，可以使用 [SystemFonts](#) 的成员作为静态属性或动态资源引用（静态属性值为资源键）。如果希望字体规格在应用程序运行时自动更新，请使用动态资源引用；否则，请使用静态值引用。

### ① 备注

资源键属性名称后面附有“Key”后缀。

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按钮。此标记示例将 [SystemFonts](#) 值分配给按钮。

#### XAML

```
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3"
    FontSize="{x:Static SystemFonts.IconFontSize}"
    FontWeight="{x:Static SystemFonts.MessageFontWeight}"
    FontFamily="{x:Static SystemFonts.CaptionFontFamily}">
    SystemFonts
</Button>
```

若要在代码中使用 [SystemFonts](#) 的值，不一定要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#

```
Button btncsharp = new Button();
btncsharp.Content = "SystemFonts";
btncsharp.Background = SystemColors.ControlDarkDarkBrush;
btncsharp.FontSize = SystemFonts.IconFontSize;
```

```
btncsharp.FontWeight = SystemFonts.MessageFontWeight;
btncsharp.FontFamily = SystemFonts.CaptionFontFamily;
cv1.Children.Add(btncsharp);
```

## 另请参阅

- [SystemFonts](#)
- [使用系统画笔绘制区域](#)
- [使用 SystemParameters](#)
- [使用系统字体键](#)
- [操作指南主题](#)
- [x:Static 标记扩展](#)
- [XAML 资源](#)
- [DynamicResource 标记扩展](#)

# 如何：使用系统字体键

项目 • 2023/02/06

系统资源将多个系统规格以资源的形式公开，以帮助开发人员创建与系统设置一致的视觉效果。[SystemFonts](#) 是一个包含系统字体值和绑定到这些值的系统字体资源（例如 [CaptionFontFamily](#) 和 [CaptionFontFamilyKey](#)）的类。

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按钮。此 XAML 示例会创建将 [SystemFonts](#) 值分配给按钮的按钮样式。

## 示例

XAML

```
<Style x:Key="SimpleFont" TargetType="{x:Type Button}">
    <Setter Property = "FontSize" Value= "{DynamicResource {x:Static
SystemFonts.IconFontSizeKey}}"/>
    <Setter Property = "FontWeight" Value= "{DynamicResource {x:Static
SystemFonts.MessageFontWeightKey}}"/>
    <Setter Property = "FontFamily" Value= "{DynamicResource {x:Static
SystemFonts.CaptionFontFamilyKey}}"/>
</Style>
```

## 另请参阅

- [使用系统画笔绘制区域](#)
- [使用 SystemParameters](#)
- [使用 SystemFonts](#)

# 如何：使用 SystemParameters

项目 • 2023/02/06

此示例演示如何访问并使用 [SystemParameters](#) 的属性来设置按钮样式或自定义按钮。

## 示例

系统资源将多个基于系统的设置以资源的形式公开，以帮助创建与系统设置一致的视觉效果。[SystemParameters](#) 是一个类，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。

在 XAML 中，可以使用 [SystemParameters](#) 的成员作为静态属性使用或动态资源引用（静态属性值作为资源键）。如果希望基于系统的值在应用程序运行时自动更新，请使用动态资源引用；否则，请使用静态引用。资源键属性名称后面附有后缀 `Key`。

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" Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="5"
    HorizontalAlignment="Left"
    Height="{x:Static SystemParameters.CaptionHeight}"
    Width="{x:Static SystemParameters.IconGridWidth}">
    SystemParameters
</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，但是系统承载的 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#

```
Button btncsharp = new Button();
btncsharp.Content = "SystemParameters";
btncsharp.FontSize = 8;
btncsharp.Background = SystemColors.ControlDarkDarkBrush;
btncsharp.Height = SystemParameters.CaptionHeight;
btncsharp.Width = SystemParameters.IconGridWidth;
cv2.Children.Add(btncsharp);
```

## 另请参阅

- [SystemParameters](#)
- [使用系统画笔绘制区域](#)
- [使用 SystemFonts](#)
- [使用系统参数键](#)
- [操作指南主题](#)

# 如何：使用系统参数键

项目 • 2023/02/06

系统资源将多个系统规格以资源的形式公开，以帮助开发人员创建与系统设置一致的视觉效果。[SystemParameters](#) 是同时包含系统参数值和绑定到该值的资源键的一个类—例如 [FullPrimaryScreenHeight](#) 和 [FullPrimaryScreenHeightKey](#)。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

动态资源具有附加有关键字“键”的属性名称。

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 示例通过为按钮的宽度和高度分配 [SystemParameters](#) 值来设置按钮大小。

## 示例

XAML

```
<Style x:Key="SimpleParam" TargetType="{x:Type Button}">
    <Setter Property = "Height" Value= "{DynamicResource {x:Static
SystemParameters.CaptionHeightKey}}"/>
    <Setter Property = "Width" Value= "{DynamicResource {x:Static
SystemParameters.IconGridWidthKey}}"/>
</Style>
```

## 另请参阅

- [使用系统画笔绘制区域](#)
- [使用 SystemFonts](#)
- [使用 SystemParameters](#)

# 文档

项目 • 2022/09/27

Windows Presentation Foundation (WPF) 提供一组功能丰富的组件，使用这些组件，开发人员能够生成具有高级文档功能和提供更好阅读体验的应用程序。除增强功能和质量外，Windows Presentation Foundation (WPF) 还针对文档打包、安全性和存储提供简单的管理服务。

## 本节内容

[WPF 中的文档](#)

[文档序列化和存储](#)

[批注](#)

[流内容](#)

[版式](#)

[打印和打印系统管理](#)

## 另请参阅

- [DocumentViewer](#)
- [FlowDocument](#)
- [System.Windows.Xps](#)
- [isXPS.exe \( isXPS 合规性工具 \)](#)

# WPF 中的文档

项目 • 2023/02/06

WPF 还对文档显示、打包和安全性提供集成服务。本主题介绍 WPF 文档类型和文档打包。

## 文档类型

WPF 基于文档用途将文档分成两大类别；这些文档类别分别称为“固定文档”和“流文档”。

固定文档适用于需要精确的“所见即所得”(WYSIWYG) 呈现的应用程序，这与所使用的显示器或打印机硬件无关。固定文档的典型用途包括桌面发布、字处理和窗体布局，在这些情况下，遵循原始页面设计非常关键。作为其布局的一部分，固定文档独立于所使用的显示或打印设备来对内容元素进行精确地定位安放。例如，一个固定文档页面在 96 dpi 显示器上显示的效果与在 600 dpi 激光打印机或 4800 dpi 照相排字机上输出的效果是完全一样的。虽然文档质量会根据每台设备的功能达到最优化，但是页面布局在所有情况下都保持不变。

比较而言，流文档旨在优化查看和可读性，因此，当易读性是文档的主要使用要求时，最适合使用流文档。流文档根据运行时变量（例如，窗口大小、设备分辨率和可选的用户首选项）来动态调整和重新排列内容，而不是设置为一个预定义的布局。网页就是流文档的一个简单示例，网页上的页面内容会动态调整格式以适应当前窗口。流文档会基于运行时环境来优化用户的查看和阅读体验。例如，在高分辨率的 19 英寸显示器上或小型 2x3 英寸 PDA 屏幕上，同一流文档会动态调整格式以实现最佳可读性。此外，流文档还有很多内置功能，包括搜索、能够优化可读性的查看模式以及更改字体大小和外观的功能。有关流文档的演示、示例和详细信息，请参阅[流文档概述](#)。

## 文档控件和文本布局

.NET Framework 提供一组预生成的控件，可以简化应用程序内固定文档、流文档和常规文本的使用。固定文档内容的显示是由 [DocumentViewer](#) 控件支持的。流文档内容的显示是由以下三个不同的控件支持的：[FlowDocumentReader](#)、[FlowDocumentPageViewer](#) 和 [FlowDocumentScrollView](#)，它们分别映射到不同的用户方案（请参阅以下部分）。其他 WPF 控件提供简化的布局以支持常规文本的使用（请参阅下面的[用户界面中的文本](#)）。

### 固定文档控件 - DocumentViewer

[DocumentViewer](#) 控件旨在显示 [FixedDocument](#) 内容。[DocumentViewer](#) 控件提供直观的用户界面，为常见操作（包括打印输出、复制到剪贴板、缩放和文本搜索功能）提供内

置支持。此控件通过常见的滚动机制提供对页面内容的访问。像所有 WPF 控件一样，[DocumentViewer](#) 支持完整或部分样式调整，这使得控件可以在视觉效果方面几乎与任何应用程序或环境相集成。

[DocumentViewer](#) 旨在以只读模式显示内容；不支持对内容进行编辑或修改。

## 流文档概述

### ① 备注

有关流文档功能以及如何创建流文档的更多详细信息，请参阅[流文档概述](#)。

流文档内容的显示是由以下三个控件支持的：[FlowDocumentReader](#)、[FlowDocumentPageViewer](#) 和 [FlowDocumentScrollViewer](#)。

### FlowDocumentReader

[FlowDocumentReader](#) 包含使用户能够动态选择各种查看模式的功能，这些查看模式包括单页（一次一页）查看模式、一次两页（书本阅读格式）查看模式和连续滚动（无界限）查看模式。有关这些查看模式的详细信息，请参阅 [FlowDocumentReaderViewingMode](#)。如果不需要在不同查看模式之间动态切换的功能，则可使用 [FlowDocumentPageViewer](#) 和 [FlowDocumentScrollViewer](#)，它们提供了固定使用特定查看模式的轻量级流内容查看器。

### FlowDocumentPageViewer 和 FlowDocumentScrollViewer

[FlowDocumentPageViewer](#) 以一次一页的查看模式显示内容，而 [FlowDocumentScrollViewer](#) 以连续滚动模式显示内容。[FlowDocumentPageViewer](#) 和 [FlowDocumentScrollViewer](#) 都固定使用特定查看模式。相比之下，[FlowDocumentReader](#) 包含的功能使用户能够动态选择各种查看模式（由 [FlowDocumentReaderViewingMode](#) 枚举提供），但代价是需要消耗比 [FlowDocumentPageViewer](#) 或 [FlowDocumentScrollViewer](#) 更多的资源。

默认情况下，总是显示垂直滚动条，而水平滚动条则在需要时显示。[FlowDocumentScrollViewer](#) 的默认 UI 不包括工具栏；不过，可使用 [IsToolBarVisible](#) 属性来启用内置工具栏。

## 用户界面中的文本

除可将文本添加到文档外，文本显然还可以用于应用程序 UI（如窗体）中。 WPF 包括多个用于在屏幕中绘制文本的控件。 每个控件都面向不同的方案，并具有自己的功能和限制列表。 一般而言，当需要有限的文本支持（例如用户界面（UI）中的简短句子）时，应使用 [TextBlock](#) 元素。 当需要最少的文本支持时，可以使用 [Label](#)。 有关详细信息，请参阅 [TextBlock 概述](#)。

## 文档打包

[System.IO.Packaging](#) API 提供一种有效方式，用于在便于访问、可移植和易于分发的单一容器中组织应用程序数据、文档内容和相关资源。 ZIP 文件是 [Package](#) 类型的一个示例，能够将多个对象保存为一个单元。 打包 API 提供一个默认 [ZipPackage](#) 实现，此实现设计为在 XML 和 ZIP 文件体系结构中使用开放式打包约定标准。 WPF 打包 API 使得创建包以及在包内存储和访问对象变得更为简单。 存储在 [Package](#) 中的对象称为 [PackagePart](#)（“部件”）。 包还可包括已签名的数字证书，这些证书可用于标识部件的发行方以及验证包内容是否尚未修改。 包还包括 [PackageRelationship](#) 功能，该功能允许将其他信息添加到包中，或在未实际修改现有部件内容的情况下与特定部件相关联。 包服务还支持 Microsoft Windows Rights Management (RM)。

WPF 包体体系结构用作大量关键技术的基础：

- 符合 XML 纸张规范 (XPS) 的 XPS 文档。
- Microsoft Office“12”开放式 XML 格式文档 (.docx)。
- 用于个人应用程序设计的自定义存储格式。

[XpsDocument](#) 基于打包 API，专为存储 WPF 固定内容文档而设计。[XpsDocument](#) 是自包含文档，可以在查看器中打开、在 [DocumentViewer](#) 控件中显示、路由到打印后台或直接输出到与 XPS 兼容的打印机。

以下部分提供有关 WPF 附带的 [Package](#) 和 [XpsDocument](#) API 的更多信息。

## 包组件

WPF 打包 API 允许将应用程序数据和文档组织成单个可移植单元。 ZIP 文件是最常见的包类型之一，并且是 WPF 附带的默认包类型。[Package](#) 本身是一个抽象类，可从中使用开放式标准 XML 和 ZIP 文件体系结构来实现 [ZipPackage](#)。[Open](#) 方法默认使用 [ZipPackage](#) 来创建和使用 ZIP 文件。 包可以包含三种基本类型的项：

项	描述
<a href="#">PackagePart</a>	应用程序内容、数据、文档和资源文件。

项	描述
PackageDigitalSignature	[X.509 证书] 用于标识、身份验证和验证。
PackageRelationship	与包或特定部件相关的补充信息。

## PackageParts

[PackagePart](#) (“部件”) 是一个抽象类，指存储在 [Package](#) 中的对象。在 ZIP 文件中，包的各个部件与存储在 ZIP 文件中的各个文件相对应。[ZipPackagePart](#) 为存储在 [ZipPackage](#) 中的可序列化对象提供默认实现。与文件系统类似，包中包含的部件存储在分层目录或“文件夹样式”组织中。使用 WPF 打包 API，应用程序可以使用单一 ZIP 文件容器写入、存储和读取多个 [PackagePart](#) 对象。

## PackageDigitalSignatures

为安全起见，[PackageDigitalSignature](#) (“数字签名”) 可以与包内的各部件关联。[PackageDigitalSignature](#) 包含提供以下两个功能的 [509]：

1. 标识部件的发信方并对其进行身份验证。
2. 验证部件是否尚未被修改。

数字签名不会阻止修改部件，但如果该部件已经以任何方式发生改变，则对该数字签名的验证检查将失败。然后应用程序可采取适当的操作（例如，阻止打开部件，或通知用户该部件已修改，是不安全的）。

## PackageRelationships

[PackageRelationship](#) (“关系”) 提供一种机制，用于将其他信息与包或包内的部件关联。关系是一种包级别的设备，可以在未修改实际部件内容的情况下将其他信息与部件关联。在许多情况下，直接向部件内容中插入新数据通常是不可行的：

- 部件及其内容架构的实际类型未知。
- 即使已知，内容架构可能也不会提供添加新信息的方式。
- 部件可能已进行数字签名或加密，不能进行任何修改。

包关系提供一种可检测到的方式，用于添加其他信息并将该信息与各个部件或整个包关联。包关系具有两种主要功能：

1. 定义一个部件与另一个部件之间的依赖关系。

2. 定义添加注释或与部件相关的其他数据的信息关系。

[PackageRelationship](#) 提供一种快速、可检测到的方式来定义依赖关系，并添加与包的一个部件或整个包相关联的其他信息。

## 依赖关系

依赖关系用于描述一个部件对其他部件的依赖性。例如，一个包可能包含一个 HTML 部件，该部件包含一个或多个 `<img>` 图像标记。此图像标记指的是作为包的其他内部部件或外部部件（可通过 Internet 访问）定位的图像。通过创建与 HTML 文件关联的 [PackageRelationship](#)，可以快速轻松地发现和访问从属资源。浏览器或查看器应用程序可以直接访问部件关系，并在无需了解架构或未分析文档的情况下立即开始汇编从属资源。

## 信息关系

与注释或批注类似，[PackageRelationship](#) 还可以用于存储要与部件关联的其他类型的信息，而不必实际修改部件内容本身。

## XPS 文档

XML 纸张规范 (XPS) 文档是一个包，其中包含一个或多个固定文档以及呈现操作所需的所有资源和信息。XPS 是 Windows Vista 的本机后台打印文件格式。[XpsDocument](#) 存储在标准 ZIP 数据集中，可以包含 XML 和二进制组件的组合，比如图像和字体文件。

[PackageRelationships](#) 用于定义内容和完全呈现文档所需的资源之间的依赖关系。

[XpsDocument](#) 设计提供单一的高保真文档解决方案，支持以下多种用途：

- 将固定文档内容和资源读取、写入和存储为单个可移植且易于分发的文件。
- 使用 XPS 查看器应用程序显示文档。
- 以 Windows Vista 的本机后台打印输出格式输出文档。
- 将文档直接路由到与 XPS 兼容的打印机。

## 另请参阅

- [FixedDocument](#)
- [FlowDocument](#)
- [XpsDocument](#)
- [ZipPackage](#)
- [ZipPackagePart](#)

- [PackageRelationship](#)
- [DocumentViewer](#)
- [文本](#)
- [流文档概述](#)
- [打印概述](#)
- [文档序列化和存储](#)

# 文档序列化和存储

项目 • 2022/09/27

Microsoft .NET Framework 为创建和显示高质量的文档提供一个强大环境。增强功能可支持固定文档和流文档以及高级查看控件，这些增强功能与强大的 2D 和 3D 图形功能结合在一起，将 .NET Framework 应用程序提升到新的质量和用户体验水平。.NET Framework 的主要功能是能够灵活管理文档的内存中表示形式，而几乎每个应用程序都需要能够高效保存和加载数据存储中的文档。将文档从内部的内存中表示形式转换为外部数据存储的过程称为序列化。反之，读取数据存储并重新创建原始内存中实例的过程则称为反序列化。

## 关于文档序列化

理论上，对于应用程序来说，从内存中序列化文档和将文档反序列到原来的内存中都是透明的。应用程序调用序列化程序“write”方法来保存文档，而反序列化程序“read”方法则访问数据存储并在内存中重新创建原始实例。对于应用程序来说，只要序列化和反序列化进程将文档重新创建为其原始格式，数据存储的特定格式通常无关紧要。

应用程序通常提供多个序列化选项，用户可以使用这些选项将文档保存到不同的介质或保存为不同格式。例如，应用程序可提供“另存为”选项将文档存储到磁盘文件、数据库或 Web 服务。同样，不同的序列化程序可将文档存储为不同的格式，例如 HTML、RTF、XML、XPS 或第三方格式。对于应用程序，序列化定义了一个接口，该接口可以隔离每个特定序列化程序的实现内部的存储介质的详细信息。除封装存储详细信息这个优点之外，.NET Framework [System.Windows.Documents.Serialization](#) API 还提供多个其他重要功能。

## .NET Framework 3.0 文档序列化程序的功能

- 通过直接访问高级别文档对象（逻辑树和视觉对象），可以有效存储已分页的内容、二维/三维元素、图像、媒体、超链接、注释以及其他支持内容。
- 同步和异步操作。
- 支持具有以下增强功能的插件序列化程序：
  - 供所有 .NET Framework 应用程序使用的系统范围访问。
  - 简单应用程序插件的发现功能。
  - 第三方自定义插件的简单部署、安装和更新。
  - 对自定义运行时设置和选项的用户界面支持。

## XPS 打印路径

Microsoft .NET Framework XPS 打印路径还通过打印输出为编写文档提供一种可扩展机制。 XPS 可以作为文档文件格式，也可以作为 Windows Vista 的本机后台打印格式。 XPS 文档可直接发送到与 XPS 兼容的打印机，而无需转换为中间格式。有关打印路径输出选项和功能的其他信息，请参阅[打印概述](#)。

## 插件序列化程序

[System.Windows.Documents.Serialization](#) API 对以下各项提供支持：与应用程序分开安装、在运行时绑定并使用 [SerializerProvider](#) 发现机制访问的插件序列化程序和链接序列化程序。插件序列化程序提供增强功能，可以简化部署和系统范围的使用。你还可以针对无法在其中访问插件序列化程序的部分可信环境（例如 XAML 浏览器应用程序（XBAP））实现链接序列化程序。链接序列化程序基于 [SerializerWriter](#) 类的派生的实现，将被编译并直接链接到应用程序。插件序列化程序和链接序列化程序都是通过相同的公共方法和事件来运行的，这样可以方便地在同一应用程序中使用其中一种或两种序列化程序。

插件序列化程序对应用程序开发人员很有帮助：它向新的存储设计和文件格式提供扩展性，而无需在生成时对每种潜在的格式进行直接编码。插件序列化程序还通过为自定义或专有文件格式提供部署、安装和更新系统可访问插件的标准方法，使第三方开发人员受益匪浅。

## 使用插件序列化程序

插件序列化程序易于使用。[SerializerProvider](#) 类为系统上安装的每个插件枚举一个 [SerializerDescriptor](#) 对象。[IsLoadable](#) 属性根据当前配置筛选已安装的插件，并验证应用程序是否可以加载和使用序列化程序。[SerializerDescriptor](#) 还提供了其他属性，例如 [DisplayName](#) 和 [DefaultFileExtension](#)，应用程序可以使用它们来提示用户选择可用输出格式的序列化程序。XPS 的默认插件序列化程序随 .NET Framework 一起提供，并且始终被枚举。用户选择输出格式后，使用 [CreateSerializerWriter](#) 方法为特定格式创建 [SerializerWriter](#)。然后可以调用 [SerializerWriter.Write](#) 方法将文档流输出到数据存储。

以下示例说明了在“PlugInFileFilter”属性中使用 [SerializerProvider](#) 方法的应用程序。PlugInFileFilter 枚举已安装的插件，并使用 [SaveFileDialog](#) 的可用文件选项生成筛选器字符串。

C#

```
// ----- PlugInFileFilter -----
/// <summary>
/// Gets a filter string for installed plug-in serializers.</summary>
```

```

/// <remark>
///   PlugInFileFilter is used to set the SaveFileDialog or
///   OpenFileDialog "Filter" property when saving or opening files
///   using plug-in serializers.</remark>
private string PlugInFileFilter
{
    get
    {
        // Create a SerializerProvider for accessing plug-in serializers.
        SerializerProvider serializerProvider = new SerializerProvider();
        string filter = "";

        // For each loadable serializer, add its display
        // name and extension to the filter string.
        foreach (SerializerDescriptor serializerDescriptor in
            serializerProvider.InstalledSerializers)
        {
            if (serializerDescriptor.IsLoadable)
            {
                // After the first, separate entries with a "|".
                if (filter.Length > 0) filter += "|";

                // Add an entry with the plug-in name and extension.
                filter += serializerDescriptor.DisplayName + " (" +
                    serializerDescriptor.DefaultFileExtension + ")|*" +
                    serializerDescriptor.DefaultFileExtension;
            }
        }

        // Return the filter string of installed plug-in serializers.
        return filter;
    }
}

```

用户选择输出文件名之后，下面的示例阐释如何使用 [CreateSerializerWriter](#) 方法以指定格式存储给定文档。

C#

```

// Create a SerializerProvider for accessing plug-in serializers.
SerializerProvider serializerProvider = new SerializerProvider();

// Locate the serializer that matches the fileName extension.
SerializerDescriptor selectedPlugIn = null;
foreach ( SerializerDescriptor serializerDescriptor in
            serializerProvider.InstalledSerializers )
{
    if ( serializerDescriptor.IsLoadable &&
        fileName.EndsWith(serializerDescriptor.DefaultFileExtension) )
    {
        // The plug-in serializer and fileName extensions match.
        selectedPlugIn = serializerDescriptor;
        break; // foreach
    }
}

```

```
// If a match for a plug-in serializer was found,
// use it to output and store the document.
if (selectedPlugIn != null)
{
    Stream package = File.Create(fileName);
    SerializerWriter serializerWriter =
        serializerProvider.CreateSerializerWriter(selectedPlugIn,
                                                    package);
    IDocumentPaginatorSource idoc =
        flowDocument as IDocumentPaginatorSource;
    serializerWriter.Write(idoc.DocumentPaginator, null);
    package.Close();
    return true;
}
```

## 安装插件序列化程序

[SerializerProvider](#) 类为插件序列化程序的发现和访问提供上层应用程序接口。

[SerializerProvider](#) 为应用程序查找和提供系统上已安装并可访问的序列化程序的列表。

已安装序列化程序的具体信息通过注册表设置来定义。 使用 [RegisterSerializer](#) 方法可向注册表添加插件序列化程序；或者如果尚未安装 .NET Framework，插件安装脚本自身可以直接设置注册表值。 [UnregisterSerializer](#) 方法可用于删除以前安装的插件，或者类似地，可以通过卸载脚本重置注册表设置。

## 创建插件序列化程序

插件序列化程序和链接序列化程序使用同一公开的公共方法和事件，并且可同样设计为以同步或异步方式运行。 创建插件序列化程序通常应执行下列三个基本步骤：

1. 首先以链接序列化程序的形式实现和调试序列化程序。 开始时通过创建直接编译和链接到测试应用程序的序列化程序，可以提供对断点以及其他有助于测试的调试服务的完全访问权限。
2. 序列化程序经过全面测试后，添加一个 [ISerializerFactory](#) 接口来创建插件。  
[ISerializerFactory](#) 接口允许完全访问包括逻辑树、[UIElement](#) 对象、[IDocumentPaginatorSource](#) 和 [Visual](#) 元素的所有 .NET Framework 对象。 此外，[ISerializerFactory](#) 提供了链接序列化程序使用的相同同步和异步方法和事件。 由于输出大型文档需要一定时间，因此推荐使用异步操作以维持响应用户交互，并在数据存储出现问题时提供“取消”选项。
3. 创建插件序列化程序之后，实现安装脚本以分发和安装（以及卸载）插件（请参阅上面的“[安装插件序列化程序](#)”）。

## 另请参阅

- [System.Windows.Documents.Serialization](#)
- [XpsDocumentWriter](#)
- [XpsDocument](#)
- [WPF 中的文档](#)
- [打印概述](#)
- [XML 纸张规范](#) ↗

# 批注

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 提供支持批注文档内容的文档查看控件。

## 本节内容

[批注概述](#)

[批注架构](#)

## 参考

[Annotation](#)

[AnnotationService](#)

[DocumentViewer](#)

## 相关章节

[WPF 中的文档](#)

[流文档概述](#)

# 批注概述

项目 · 2022/09/27

在纸质文档上编写说明或注释毫不稀奇，我们几乎认为这是理所当然的。这些说明或注释就是“批注”，我们将其添加到文档，用于标注信息或突出显示兴趣项以供日后参考。

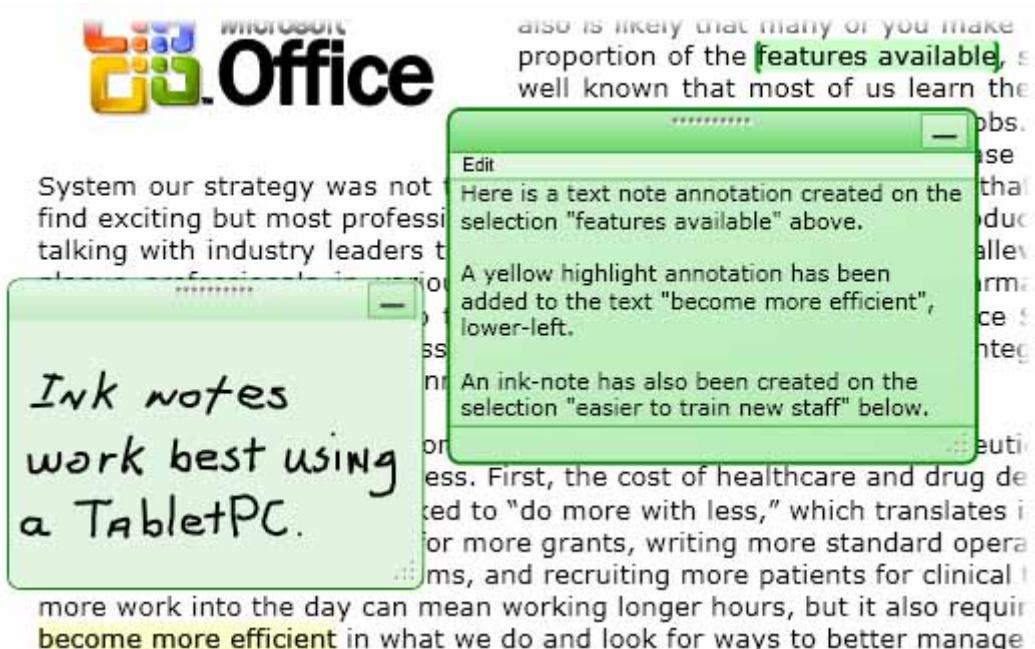
虽然在打印文档上编写注释很简单也很平常，但是就算在所有电子文档上添加个人注释，功能上却通常有很多限制。

本主题介绍几种常见类型的批注，重点介绍便笺和突出显示，并举例说明 Microsoft Annotations Framework 如何通过 Windows Presentation Foundation (WPF) 文档查看控件简化在应用程序中使用这些类型的批注。支持批注的 WPF 文档查看控件包括 [FlowDocumentReader](#) 和 [FlowDocumentScrollViewer](#)，以及派生自 [DocumentViewerBase](#) 的控件，如 [DocumentViewer](#) 和 [FlowDocumentPageViewer](#)。

## 便笺

平常的便笺是将信息写在小块彩纸上，随后将这张彩纸“粘贴”到文档。数字便笺为电子文档提供类似的功能，但灵活性更高，可包括许多其他类型的内容，如键入文本、手写注释（如 Tablet PC“墨迹”笔划）或 Web 链接。

下图显示了突出显示、文本便笺以及墨迹便笺批注的一些示例。



下面的示例演示了可用于在应用程序中启用批注支持的方法。

```
C#
```

```
// ----- StartAnnotations -----  
/// <summary>
```

```

///<summary> Enables annotations and displays all that are viewable.</summary>
private void StartAnnotations()
{
    // If there is no AnnotationService yet, create one.
    if (_annotService == null)
        // docViewer is a document viewing control named in Window1.xaml.
        _annotService = new AnnotationService(docViewer);

    // If the AnnotationService is currently enabled, disable it.
    if (_annotService.IsEnabled == true)
        _annotService.Disable();

    // Open a stream to the file for storing annotations.
    _annotStream = new FileStream(
        _annotStorePath, FileMode.OpenOrCreate, FileAccess.ReadWrite);

    // Create an AnnotationStore using the file stream.
    _annotStore = new XmlStreamStore(_annotStream);

    // Enable the AnnotationService using the new store.
    _annotService.Enable(_annotStore);
} // end:StartAnnotations()

```

## 亮点

当人们在纸质文档上作标记时，往往使用创造性地方法来突出显示兴趣项，例如对于句子中的某些字词，加下划线、高亮显示、圈出，或者将在空白的地方绘制标记或符号。Microsoft Annotations Framework 中的突出显示批注具有类似的功能，用于标记在 WPF 文档查看控件中显示的信息。

下图演示了一个突出显示批注的示例。

**Highlight** The Microsoft Annotation Framework makes it easy to add annotation capabilities to virtually any application.

用户通常以如下方法创建批注：首先选择感兴趣的文本或者项，然后单击右键显示批注选项的 [ContextMenu](#)。在下面的示例中，你可以使用 Extensible Application Markup Language 声明包含路由命令的 [ContextMenu](#)，用户可以访问这些命令来创建和管理批注。

XAML

```

<DocumentViewer.ContextMenu>
    <ContextMenu>
        <MenuItem Command="ApplicationCommands.Copy" />
        <Separator />
        <!-- Add a Highlight annotation to a user selection. -->
        <MenuItem Command="ann:AnnotationService.CreateHighlightCommand"
            Header="Add Highlight" />

```

```

<!-- Add a Text Note annotation to a user selection. -->
<MenuItem Command="ann:AnnotationService.CreateTextStickyNoteCommand"
           Header="Add Text Note" />
<!-- Add an Ink Note annotation to a user selection. -->
<MenuItem Command="ann:AnnotationService.CreateInkStickyNoteCommand"
           Header="Add Ink Note" />
<Separator />
<!-- Remove Highlights from a user selection. -->
<MenuItem Command="ann:AnnotationService.ClearHighlightsCommand"
           Header="Remove Highlights" />
<!-- Remove Text Notes and Ink Notes from a user selection. -->
<MenuItem Command="ann:AnnotationService.DeleteStickyNotesCommand"
           Header="Remove Notes" />
<!-- Remove Highlights, Text Notes, Ink Notes from a selection. -->
<MenuItem Command="ann:AnnotationService.DeleteAnnotationsCommand"
           Header="Remove Highlights &amp; Notes" />
</ContextMenu>
</DocumentViewer.ContextMenu>

```

## 数据锚定

Annotations Framework 将批注与用户选择的数据绑定，而不仅仅是绑定到显示视图中的某个位置。因此，如果文档视图更改（例如，当用户滚动显示窗口或者调整其大小时），批注将仍然跟随它绑定到的所选数据。例如，下图显示了用户在所选文本上做的批注。当文档视图更改时（滚动、调整大小、缩放或者移动），突出显示批注将与最初所选数据一起移动。

**Original** The Microsoft Annotation Framework makes it **easy to add annotation capabilities** to virtually any application.

**Reflowed** The Microsoft Annotation Framework makes it **easy to add annotation capabilities** to virtually any application.

## 匹配批注与批注对象

你可以将批注与对应的批注对象匹配。以具有注释窗格的简单文档读取器应用程序为例。注释窗格可能是一个列表框，用于显示锚定到文档的批注列表的文本。如果用户在列表框中选择一项，应用程序将显示相应的批注对象所锚定到的文档段落。

下面的示例演示如何实现充当注释窗格的此类列表框的事件处理程序。

C#

```

void annotationsListBox_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{

```

```
Annotation comment = (sender as ListBox).SelectedItem as Annotation;
if (comment != null)
{
    // IAnchorInfo info;
    // service is an AnnotationService object
    // comment is an Annotation object
    info = AnnotationHelper.GetAnchorInfo(this.service, comment);
    TextAnchor resolvedAnchor = info.ResolvedAnchor as TextAnchor;
    TextPointer textPointer = (TextPointer)resolvedAnchor.BoundingStart;
    textPointer.Paragraph.BringIntoView();
}
}
```

另一示例方案涉及通过电子邮件在文档读取器之间实现交换批注和便笺的应用程序。凭借此功能，这些应用程序可以将读取器导航到包含要交换的批注的页面。

## 另请参阅

- [DocumentViewerBase](#)
- [DocumentViewer](#)
- [FlowDocumentPageViewer](#)
- [FlowDocumentScrollView](#)
- [FlowDocumentReader](#)
- [IAnchorInfo](#)
- [批注架构](#)
- [ContextMenu 概述](#)
- [命令概述](#)
- [流文档概述](#)
- [如何：将命令添加到菜单项](#)

# 批注架构

项目 • 2022/09/27

本主题介绍 Microsoft Annotations Framework 用来保存和检索用户批注数据的 XML 架构定义 (XSD)。

Annotations Framework 将批注数据从内部表示形式序列化为 XML 格式。 Annotations Framework XSD 架构描述了用于此转换的 XML 格式。 该架构定义独立于实现、可用来在应用程序之间交换批注数据的 XML 格式。

Annotations Framework XML 架构定义由两个子架构组成

- 批注 XML 核心架构 (核心架构)。
- 批注 XML 基本架构 (核心架构)。

核心架构定义 [Annotation](#) 的主 XML 结构。 核心架构中定义的大多数 XML 元素对应于 [System.Windows.Annotations](#) 命名空间中的类型。 核心架构公开应用程序可在其中添加自己的 XML 数据的三个扩展点。 这些扩展点包括 [Authors](#)、[ContentLocatorPart](#) 和“内容”。 ( 内容元素以 [XmlElement](#) 列表的形式提供。 )

本主题中描述的基本架构定义 [Authors](#) 和 [ContentLocatorPart](#) 的扩展，以及初始 Windows Presentation Foundation (WPF) 版本随附的内容类型。

## 批注 XML 核心架构

批注 XML 核心架构定义用于存储 [Annotation](#) 对象的 XML 结构。

```
XML

<xsd:schema elementFormDefault="qualified"
attributeFormDefault="unqualified"
blockDefault="#all"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://schemas.microsoft.com/windows/annotations/2003/11/core"
xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core">

    <!-- The Annotations element groups a number of annotations. -->
    <xsd:element name="Annotations" type="anc:AnnotationsType" />

    <xsd:complexType name="AnnotationsType">
        <xsd:sequence>
            <xsd:element name="Annotation" minOccurs="0" maxOccurs="unbounded">
```

```

                type="anc:AnnotationType" />
            </xsd:sequence>
        </xsd:complexType>

<!-- AnnotationType defines the structure of the Annotation element. -->
<xsd:complexType name="AnnotationType">
    <xsd:sequence>

        <!-- List of 0 or more authors. -->
        <xsd:element name="Authors" minOccurs="0" maxOccurs="1"
            type="anc:AuthorListType" />

        <!-- List of 0 or more anchors. -->
        <xsd:element name="Anchors" minOccurs="0" maxOccurs="1"
            type="anc:ResourceListType" />

        <!-- List of 0 or more cargos. -->
        <xsd:element name="Cargos" minOccurs="0" maxOccurs="1"
            type="anc:ResourceListType" />

    </xsd:sequence>

    <!-- Unique annotation ID. -->
    <xsd:attribute name="Id" type="xsd:string" use="required" />

    <!-- Annotation "Type" is used to map the annotation to an annotation
        component that takes care of the visual representation of the
        annotation. WPF V1 recognizes three annotation types:
        http://schemas.microsoft.com/windows/annotations/2003/11/base:Highlight
        http://schemas.microsoft.com/windows/annotations/2003/11/base:TextStickyNote
        http://schemas.microsoft.com/windows/annotations/2003/11/base:InkStickyNote
    -->
    <xsd:attribute name="Type" type="xsd:QName" use="required" />

    <!-- Time when the annotation was last modified. -->
    <xsd:attribute name="LastModificationTime" use="optional"
        type="xsd:dateTime" />

    <!-- Time when the annotation was created. -->
    <xsd:attribute name="CreationTime" use="optional"
        type="xsd:dateTime" />
</xsd:complexType>

<!-- "Authors" consists of 0 or more elements that represent an author. -->
<xsd:complexType name="AuthorListType">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="anc:Author" />
    </xsd:sequence>
</xsd:complexType>

<!-- The core schema allows any author type. Supported author types
    in version 1 (V1) are described in the base schema. -->
```

```

<xsd:element name="Author" abstract="true" block="extension restriction"/>

<!-- Both annotation anchor and annotation cargo are represented by the
     ResourceType which contains 0 or more "Resource" elements. -->
<xsd:complexType name="ResourceListType">
  <xsd:sequence>
    <xsd:element name="Resource" minOccurs="0" maxOccurs="unbounded"
                 type="anc:ResourceType" />
  </xsd:sequence>
</xsd:complexType>

<!-- Resource groups identification, location
     and/or content of some information. -->
<xsd:complexType name="ResourceType">
  <xsd:choice minOccurs="0" maxOccurs="unbounded" >
    <xsd:choice>
      <xsd:element name="ContentLocator" type="anc:ContentLocatorType" />
      <xsd:element name="ContentLocatorGroup"
type="anc:ContentLocatorGroupType" />
    </xsd:choice>
    <xsd:element ref="anc:Content"/>
  </xsd:choice>

  <!-- Unique resource identifier. -->
  <xsd:attribute name="Id" type="xsd:string" use="required" />

  <!-- Optional resource name. -->
  <xsd:attribute name="Name" type="xsd:string" use="optional" />
</xsd:complexType>

<!-- ContentLocatorGroup contains a set of ContentLocators -->
<xsd:complexType name="ContentLocatorGroupType">
  <xsd:sequence>
    <xsd:element name="ContentLocator" minOccurs="1" maxOccurs="unbounded"
                 type="anc:ContentLocatorType" />
  </xsd:sequence>
</xsd:complexType>

<!-- A ContentLocator describes the location or the identification
     of particular data within some context. The ContentLocator consists
     of one or more ContentLocatorParts. Each ContentLocatorPart needs to
     be successively applied to the context to arrive at the data. What
     "applying", "context", and "data" mean is application dependent.
-->
<xsd:complexType name="ContentLocatorType">
  <xsd:sequence minOccurs="1" maxOccurs="unbounded">
    <xsd:element ref="anc:ContentLocatorPart" />
  </xsd:sequence>
</xsd:complexType>

<!-- A ContentLocatorPart is a set of "Item" elements. Each "Item"
element
     has "Name" and "Value" attributes that define a name/value pair.
     ContentLocatorPart is an abstract type that must be restricted for
each

```

```

        concrete ContentLocatorPart definition. This restriction should
define
        allowed names and values for the concrete ContentLocatorPart type.
That
        way the application can define its own way of locating information.
The
        ContentLocatorPartTypes that are allowed in version 1 (V1) of WPF are
defined in the Annotations Base Schema.

-->
<xsd:element name="ContentLocatorPart" type="anc:ContentLocatorPartType"
              abstract="true" />

<xsd:complexType name="ContentLocatorPartType" abstract="true"
                  block="restriction">
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="Item" type="anc:ItemType" />
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ItemType" abstract="true" >
    <xsd:attribute name="Name" type='xsd:string' use="required" />
    <xsd:attribute name="Value" type='xsd:string' use="optional" />
</xsd:complexType>

<!-- Content describes the underlying content of a resource. This is an
     abstract type that should be redefined for each concrete content type
     through restriction. Allowed content types in WPF version 1 are
     defined in the Annotations Base Schema.

-->
<xsd:element name="Content" abstract="true" block="extension
restriction"/>

</xsd:schema>

```

## 批注 XML 基本架构

基本架构为核心架构中定义的以下三个抽象元素定义 XML 结构：[Authors](#)、[ContentLocatorPart](#) 和 [Contents](#)。

XML

```

<xsd:schema elementFormDefault="qualified"
attributeFormDefault="unqualified"
blockDefault="#all"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://schemas.microsoft.com/windows/annotations/2003/11/ba
se"

xmlns:anb="http://schemas.microsoft.com/windows/annotations/2003/11/base"

```

```

xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core">

<xsd:import schemaLocation="AnnotationCoreV1.xsd"
namespace="http://schemas.microsoft.com/windows/annotations/2003/11/core"/>

<!-- ***** Author ***** -->
<!-- Simple DisplayName Author -->
<xsd:complexType name="StringAuthorType">
  <xsd:simpleContent>
    <xsd:extension base='xsd:string' />
  </xsd:simpleContent>
</xsd:complexType>
<xsd:element name="StringAuthor" type="anb:StringAuthorType"
  substitutionGroup="anc:Author"/>

<!-- ***** LocatorParts ***** -->

<!-- Helper types -->

<!-- CountItemNameType - helper type to define count item -->
<xsd:simpleType name="CountItemNameType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="Count" />
  </xsd:restriction>
</xsd:simpleType>

<!-- NumberType - helper type to define segment count item -->
<xsd:simpleType name="NumberType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- SegmentNameType: helper type to define possible segment name types -->
<xsd:simpleType name="SegmentItemNameType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value="Segment\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- Flow Locator Part -->

<!-- FlowSegmentValueItemType: helper type to define flow segment values -->
<xsd:simpleType name="FlowSegmentItemValueType">
  <xsd:restriction base='xsd:string'>
    <xsd:pattern value=" \d*,\d*" />
  </xsd:restriction>
</xsd:simpleType>

<!-- FlowItemType -->
<xsd:complexType name="FlowItemType" abstract = "true">
  <xsd:complexContent>

```

```

        <xsd:restriction base="anc:ItemType">
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<!-- FlowSegmentItemType --&gt;
&lt;xsd:complexType name="FlowSegmentItemType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anb:FlowItemType"&gt;
            &lt;xsd:attribute name="Name" use="required"
                type="anb:SegmentItemNameType"/&gt;
            &lt;xsd:attribute name="Value" use="required"
                type="anb:FlowSegmentItemValueType"/&gt;
        &lt;/xsd:restriction&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;

<!-- FlowCountItemType --&gt;
&lt;xsd:complexType name="FlowCountItemType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anb:FlowItemType"&gt;
            &lt;xsd:attribute name="Name" type="anb:CountItemNameType"
use="required"/&gt;
            &lt;xsd:attribute name="Value" type="anb:NumberType" use="required"/&gt;
        &lt;/xsd:restriction&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;

<!-- CharacterRangeType is an extension of ContentLocatorPartType that
locates
 *      part of the content within a FlowDocument. CharacterRangeType
contains one
 *      "Item" element with name "Count" and value the number(N) of
"SegmentXX"
 *      elements that this ContentLocatorPart has. It also contains N "Item"
 *      elements with name "SegmentXX" where XX is a number from 0 to N-1.
The
 *      value of each "SegmentXX" element is a string in the form "offset,
length"
 *      which locates one sequence of symbols in the FlowDocument. Example:

 *          &lt;anb:CharacterRange&gt;
 *              &lt;anc:Item Name="Count" Value="2" /&gt;
 *              &lt;anc:Item Name="Segment0" Value="5,10" /&gt;
 *              &lt;anc:Item Name="Segment1" Value="25,2" /&gt;
 *          &lt;/anb:CharacterRange&gt;
--&gt;
&lt;xsd:complexType name="CharacterRangeType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:extension base="anc:ContentLocatorPartType"&gt;
            &lt;xsd:sequence minOccurs="1" maxOccurs="unbounded"&gt;
                &lt;xsd:element name="Item" type="anb:FlowItemType" /&gt;
            &lt;/xsd:sequence&gt;
        &lt;/xsd:extension&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;</pre>

```

```

</xsd:complexType>


<xsd:element name="CharacterRange" type="anb:CharacterRangeType"
    substitutionGroup="anc:ContentLocatorPart"/>

<!-- Fixed LocatorPart --&gt;

<!-- Helper type - FixedItemType --&gt;
&lt;xsd:complexType name="FixedItemType" abstract = "true"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anc:ItemType"&gt;
            &lt;/xsd:restriction&gt;
        &lt;/xsd:complexContent&gt;
    &lt;/xsd:complexType&gt;

<!-- Helper type - FixedCountItemType: ContentLocatorPart items count --&gt;
&lt;xsd:complexType name="FixedCountItemType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anb:FixedItemType"&gt;
            &lt;xsd:attribute name="Name" type="anb:CountItemNameType"
use="required"/&gt;
                &lt;xsd:attribute name="Value" type="anb:NumberType" use="required"/&gt;
            &lt;/xsd:restriction&gt;
        &lt;/xsd:complexContent&gt;
    &lt;/xsd:complexType&gt;

<!-- Helper type -FixedSegmentValue: Defines possible fixed segment values
--&gt;
&lt;xsd:simpleType name="FixedSegmentItemValueType"&gt;
    &lt;xsd:restriction base='xsd:string'&gt;
        &lt;xsd:pattern value="\d*,\d*,\d*,\d*" /&gt;
    &lt;/xsd:restriction&gt;
&lt;/xsd:simpleType&gt;

<!-- Helper type - FixedSegmentItemType --&gt;
&lt;xsd:complexType name="FixedSegmentItemType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anb:FixedItemType"&gt;
            &lt;xsd:attribute name="Name" use="required"
                type="anb:SegmentItemNameType"/&gt;
            &lt;xsd:attribute name="Value" use="required"
                type="anb:FixedSegmentItemValueType "/&gt;
        &lt;/xsd:restriction&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;

<!-- FixedTextRangeType is an extension of ContentLocatorPartType that
locates
    * content within a FixedDocument. It contains one "Item" element with
name
    * "Count" and value the number (N) of "Item" elements with name
"SegmentXX"
    * that this ContentLocatorPart has. FixedTextRange locator part also
    * contains N "Item" elements with one attribute Name="SegmentXX" where
</pre>

```

XX is

- \* a number from 0 to N-1 and one attribute "Value" in the form "X1, Y1, X2,
- \* Y2". Here X1,Y1 are the coordinates of the start symbol in this segment,
- \* X2,Y2 are the coordinates of the end symbol in this segment.

Example:

```

*
*      <anb:FixedTextRange>
*          <anc:Item Name="Count" Value="2" />
*          <anc:Item Name="Segment0" Value="10,5,20,5" />
*          <anc:Item Name="Segment1" Value="25,15, 25,20" />
*      </anb:FixedTextRange>
-->
<xsd:complexType name="FixedTextRangeType">
    <xsd:complexContent>
        <xsd:extension base="anc:ContentLocatorPartType">
            <xsd:sequence minOccurs="1" maxOccurs="unbounded">
                <xsd:element name="Item" type="anb:FixedItemType" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>


<xsd:element name="FixedTextRange" type="anb:FixedTextRangeType"
    substitutionGroup="anc:ContentLocatorPart"/>

<!-- DataId --&gt;

<!-- ValueItemNameType: helper type to define value item --&gt;
&lt;xsd:simpleType name="ValueItemNameType"&gt;
    &lt;xsd:restriction base='xsd:string'&gt;
        &lt;xsd:pattern value="Value" /&gt;
    &lt;/xsd:restriction&gt;
&lt;/xsd:simpleType&gt;

<!-- StringValueItemType --&gt;
&lt;xsd:complexType name="StringValueItemType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:restriction base="anc:ItemType"&gt;
            &lt;xsd:attribute name="Name" type="anb:ValueItemNameType"
use="required"/&gt;
            &lt;xsd:attribute name="Value" type="xsd:string" use="required"/&gt;
        &lt;/xsd:restriction&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;

&lt;xsd:complexType name="StringValueLocatorPartType"&gt;
    &lt;xsd:complexContent&gt;
        &lt;xsd:extension base="anc:ContentLocatorPartType"&gt;
            &lt;xsd:sequence minOccurs="1" maxOccurs="1"&gt;
                &lt;xsd:element name="Item" type="anb:ValueItemType" /&gt;
            &lt;/xsd:sequence&gt;
        &lt;/xsd:extension&gt;
    &lt;/xsd:complexContent&gt;
&lt;/xsd:complexType&gt;</pre>

```

```

        </xsd:complexContent>
    </xsd:complexType>

    <!-- DataId element substitutes ContentLocatorPart and is used to locate a
     *      subtree in the logical tree. Including DataId locator part in a
     *      ContentLocator helps to narrow down the search for a particular
content.
     *      Example of DataId ContentLocatorPart:
     *
     *          <anb:DataId>
     *              <anc:Item Name="Value" Value="FlowDocument" />
     *          </anb:DataId>
-->

<xsd:element name="DataId" type="anb: StringValueLocatorPartType" substitutionGroup="anc:ContentLocatorPart"/>

<!-- PageNumber -->

<!-- NumberValueItemType -->
<xsd:complexType name="NumberValueItemType">
    <xsd:complexContent>
        <xsd:restriction base="anc:ItemType">
            <xsd:attribute name="Name" type="anb:ValueItemNameType" use="required"/>
                <xsd:attribute name="Value" type="anb:NumberType" use="required"/>
            </xsd:restriction>
        </xsd:complexContent>
    </xsd:complexType>

<xsd:complexType name="NumberValueLocatorPartType">
    <xsd:complexContent>
        <xsd:extension base="anc:ContentLocatorPartType">
            <xsd:sequence minOccurs="1" maxOccurs="1">
                <xsd:element name="Item" type="anb:ValueItemType" />
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<!-- PageNumber element substitutes ContentLocatorPart and is used to
locate a
     *      page in a FixedDocument. PageNumber ContentLocatorPart is used in
     *      conjunction with the FixedTextRange ContentLocatorPart and it shows
on with
     *      page are the coordinates defined in the FixedTextRange.
     *      Example of a PageNumber ContentLocatorPart:
     *
     *          <anb:PageNumber>
     *              <anc:Item Name="Value" Value="1" />
     *          </anb:PageNumber>
-->

<xsd:element name="PageNumber" type="anb:NumberValueLocatorPartType" substitutionGroup="anc:ContentLocatorPart"/>
```

```

<!-- ***** Content ***** -->
<!-- Highlight colors - defines highlight color for annotations of type
*   Highlight or normal and active anchor colors for annotations of type
*   TextStickyNote and InkStickyNote.
-->
<xsd:complexType name="ColorsContentType">
  <xsd:attribute name="Background" type='xsd:string' use="required" />
  <xsd:attribute name="ActiveBackground" type='xsd:string' use="optional" />
</xsd:complexType>

<xsd:element name="Colors" type="anb:ColorsContentType"
  substitutionGroup="anc:Content"/>

<!-- RTB Text -contains XAML representing StickyNote Reach Text Box text.
*   Used in annotations of type TextStickyNote. -->
<xsd:complexType name="TextContentType">
  <!-- See XAML schema for RTB content -->
</xsd:complexType>

<xsd:element name="Text" type="anb:TextContentType"
  substitutionGroup="anc:Content"/>

<!-- Ink - contains XAML representing Sticky Note ink.
*   Used in annotations of type InkStickyNote. -->
<xsd:complexType name="InkContentType">
  <!-- See XAML schema for Ink content -->
</xsd:complexType>

<xsd:element name="Ink" type="anb:InkContentType"
  substitutionGroup="anc:Content"/>

<!-- SN Metadata - defines StickyNote attributes as position width,
height,
*   etc. Used in annotations of type TextStickyNote and InkStickyNote. --
-->
<xsd:complexType name="MetadataContentType">
  <xsd:attribute name="Left" type='xsd:decimal' use="optional" />
  <xsd:attribute name="Top" type='xsd:decimal' use="optional" />
  <xsd:attribute name="Width" type='xsd:decimal' use="optional" />
  <xsd:attribute name="Height" type='xsd:decimal' use="optional" />
  <xsd:attribute name="XOffset" type='xsd:decimal' use="optional" />
  <xsd:attribute name="YOffset" type='xsd:decimal' use="optional" />
  <xsd:attribute name="ZOrder" type='xsd:decimal' use="optional" />
</xsd:complexType>

<xsd:element name="Metadata" type="anb:MetadataContentType"
  substitutionGroup="anc:Content"/>

</xsd:schema>

```

## 批注 XmlStreamStore 所生成的示例 XML

下面的 XML 演示批注 `XmlStreamStore` 的输出以及示例文件的组织，该文件包含三个批注：突出显示、文本便笺以及墨迹便笺。

```
XML

<?xml version="1.0" encoding="utf-8"?>
<anc:Annotations

    xmlns:anc="http://schemas.microsoft.com/windows/annotations/2003/11/core"
    xmlns:anb="http://schemas.microsoft.com/windows/annotations/2003/11/base">

        <anc:Annotation Id="d308ea9b-36eb-4cc4-94d0-97634f10f7a2"
            CreationTime="2006-09-13T18:28:51.4465702-07:00"
            LastModificationTime="2006-09-13T18:28:51.4465702-07:00"
            Type="anb:Highlight">
            <anc:Anchors>
                <anc:Resource Id="4f53661b-7328-4673-8e3f-c53f08b9cd94">
                    <anc:ContentLocator>
                        <anb:DataId>
                            <anc:Item Name="Value" Value="FlowDocument" />
                        </anb:DataId>
                        <anb:CharacterRange>
                            <anc:Item Name="Segment0" Value="600,609" />
                            <anc:Item Name="Count" Value="1" />
                        </anb:CharacterRange>
                    </anc:ContentLocator>
                </anc:Resource>
            </anc:Anchors>
        </anc:Annotation>

        <anc:Annotation Id="d7a8d271-387e-4144-9f8b-bc3c97816e5f"
            CreationTime="2006-09-13T18:28:56.7903202-07:00"
            LastModificationTime="2006-09-13T18:28:56.8996952-07:00"
            Type="anb:TextStickyNote">
            <anc:Authors>
                <anb:StringAuthor>Denise Smith</anb:StringAuthor>
            </anc:Authors>

            <anc:Anchors>
                <anc:Resource Id="dab2560e-6ebd-4ad0-80f9-483356a3be0b">
                    <anc:ContentLocator>
                        <anb:DataId>
                            <anc:Item Name="Value" Value="FlowDocument" />
                        </anb:DataId>
                        <anb:CharacterRange>
                            <anc:Item Name="Segment0" Value="787,801" />
                            <anc:Item Name="Count" Value="1" />
                        </anb:CharacterRange>
                    </anc:ContentLocator>
                </anc:Resource>
            </anc:Anchors>

            <anc:Cargos>
```

```

<anc:Resource Id="ea4dbabd-b400-4cf9-8908-5716b410f9e4" Name="Meta
Data">
    <anb:MetaData anb:ZOrder="0" />
</anc:Resource>
</anc:Cargos>
</anc:Annotation>

<anc:Annotation Id="66803c69-b0d7-4cc3-bdff-cacc1955e806"
    CreationTime="2006-09-13T18:29:03.6653202-07:00"
    LastModificationTime="2006-09-13T18:29:03.7121952-07:00"
    Type="anb:InkStickyNote">
    <anc:Authors>
        <anb:StringAuthor>Mike Nash</anb:StringAuthor>
    </anc:Authors>

    <anc:Anchors>
        <anc:Resource Id="52251c53-8eeb-4fd7-b8f3-94e78dfc25fa">
            <anc:ContentLocator>
                <anb:DataId>
                    <anc:Item Name="Value" Value="FlowDocument" />
                </anb:DataId>
                <anb:CharacterRange>
                    <anc:Item Name="Segment0" Value="880,884" />
                    <anc:Item Name="Count" Value="1" />
                </anb:CharacterRange>
            </anc:ContentLocator>
        </anc:Resource>
    </anc:Anchors>

    <anc:Cargos>
        <anc:Resource Id="11e50b97-8d91-4ff9-82c3-16607b2b552b" Name="Meta
Data">
            <anb:MetaData anb:ZOrder="1" />
        </anc:Resource>
    </anc:Cargos>
</anc:Annotation>

</anc:Annotations>

```

## 另请参阅

- [System.Windows.Annotations](#)
- [System.Windows.Annotations.Storage](#)
- [Annotation](#)
- [AnnotationStore](#)
- [XmlStreamStore](#)
- [批注概述](#)

# 流内容

项目 • 2023/02/06

流内容元素为创建适合在 [FlowDocument](#) 中承载的流内容提供了构建块。

## 本节内容

[流文档概述](#)

[TextElement 内容模型概述](#)

[表概述](#)

[操作指南主题](#)

## 参考

[FlowDocument](#)

[Block](#)

[List](#)

[Paragraph](#)

[Section](#)

[Table](#)

[Figure](#)

[Floater](#)

[Hyperlink](#)

[Inline](#)

[Run](#)

[Span](#)

[ListItem](#)

## 相关章节

[WPF 中的文档](#)

# 流文档概述

项目 • 2022/09/27

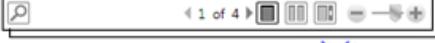
流文档旨在优化查看和可读性。 流文档根据运行时变量（例如，窗口大小、设备分辨率和可选的用户首选项）来动态调整和重新排列内容，而不是设置为一个预定义的布局。 此外，流文档还提供一些高级文档功能，例如分页和分栏。 本主题概述了流文档及其创建方式。

## 什么是流文档？

流文档旨在根据窗口大小、设备分辨率和其他环境变量来“重排内容”。 此外，流文档还具有很多内置功能，包括搜索、能够优化可读性的查看模式以及更改字体大小和外观的功能。 当易读性是文档的主要使用要求时，最适合使用流文档。 相反，固定文档旨在提供静态表示形式。 当源内容的保真度至关重要时，就适合使用固定文档。 有关不同类型文档的详细信息，请参阅 [WPF 中的文档](#)。

下图演示在多个不同大小的窗口中查看同一个示例流文档的情况。 随着显示区域的变化，内容将重新布局，以充分利用可用空间。

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla ligula tortor, dapibus et, facilisis a, aliquet et, diam. Cras convallis. Fusce at urna. Quisque interdum, turpis sed dictum fringilla, magna arcu gravida libero, elementum tincidunt dolor mauris sed erat. Curabitur egestas aliquet nibh. Etiam quis sem in lorem auctor consequat. Suspendisse vitae lorem. Proin eleifend elementum ligula. Donec mattis consectetur erat. Donec fermentum consectetur neque. Suspendisse tempus faucibus magna. Pellentesque sodales velit eget nibh. Phasellus velit magna, malesuada vitae, rhoncus eget, dignissim ut, metus. Praesent pulvinar suscipit diam.



Lore ipsum dolor sit amet, consectetuer adipiscing elit. Nulla ligula tortor, dapibus et, aliquet et, diam. Cras convallis. Fusce at urna. Quisque interdum, turpis sed dictum fringilla, magna arcu gravida libero, elementum tincidunt dolor mauris sed erat. Curabitur egestas aliquet nibh. Etiam quis sem in lorem auctor consequat. Suspendisse vitae lorem. Proin eleifend elementum ligula. Donec mattis consectetuer erat. Donec fermentum consectetuer neque. Suspendisse tempus fauciibus magna. Pellentesque sodales velit eget nibh. Phasellus velit magna, malesuada vitae, rhoncus eget, dignissim ut, metus. Praesent pulvinar suscipit diam.



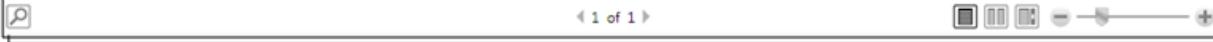
Morbi ut tellus at tellus semper consecetur. Nullam fringilla nonummy justo. Proin rutrum, purus id adipiscing malesuada, enim velit accumsan nisi, pellentesque rhoncus nibh nisi ut magna. Nunc massa nulla, volutpat sit amet, pulvinar ac, scelerisque ac, magna. Nulla facilisi. Integer pharetra felis vitae risus. Nunc aliquet lacus pretium urna varius hendrerit. Duis odio nisl, venenatis at, sollicitudin eu, viverra sed, nibh. Nullam consequat. Duis felis, rutrum viverra, auctor eget, iaculis ut, mi. Integer sagittis pharetra ipsum. Donec lacinia erat, erat nisi. Nullam aliquet. In et sapien. Fusce blandit odio et mi.

Lore ipsum dolor sit amet, consectetur adipiscing elit. Nulla ligula tortor, dapibus et, facilisis a, aliquet et, diam. Cras convallis. Fusce at urna. Quisque interdum, turpis sed dictum fringilla, magna arcu gravida libero, elementum tincidunt dolor mauris sed erat. Curabitur egestas aliquet nibh. Etiam quis sem in lorem auctor consequat. Suspendisse vitae lorem. Proin eleifend elementum ligula. Donec mattis consectetur erat. Donec fermentum consectetur neque. Suspendisse tempus faucibus magna. Pellentesque sodales velit eget nibh. Phasellus velit magna, malesuada vitae, rhoncus eget, dignissim ut, metus. Praesent pulvinar suscipit diam.



Morbi ut tellus at tellus semper consecetetuer. Nullam fringilla nonummy justo. Proin rutrum, purus id adipiscing malesuada, enim velit accumsan nisi, pellentesque rhoncus nibh nisi ut magna. Nunc massa nulla, volutpat sit amet, pulvinar ac, scelerisque ac, magna. Nulla facilisi. Integer pharetra felis vitae risus. Nunc aliquet lacus pretium urna varius hendrerit. Duis odio nisl, venenatis at, sollicitudin eu, viverra sed, nibh. Nullam consequat. Duis felis felis, rutrum viverra, auctor eget, iaculis ut, mi. Integer sagittis pharetra ipsum. Donec lacinia scelerisque nisl. Nullam aliquet. In et sapien. Fusce blandit odio et mi.

Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Suspendisse laoreet condimentum purus. Etiam vel enim. Suspendisse at dolor. Pellentesque gravida. Aenean convallis. Vivamus diam. Aenean lorem libero, suscipit vel, tempor eu, fermentum aliquam, metus. Quisque volutpat urna at augue. Nam placerat. In viverra congue tellus. Proin auctor. Fusce nisl lorem, dapibus vitae, porta sed, malesuada a, lacus. Phasellus mollis elementum enim. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam vitae urna vel velit volutpat tempor. Quisque ac dolor. Suspendisse potenti. Nunc nec tortor. Curabitur pharetra pellentesque diam.



如上图所示，流内容可包括多个组成部分，包括段落、列表、图像等等。这些组成部分对应于标记中的元素和程序代码中的对象。稍后，我们将在本概述的[与流相关的类](#)部分中详细介绍这些类。现在，我们提供一个简单的代码示例，该示例将创建一个由包含部分粗体文本的段落和列表组成的流文档。

XAML

```
<!-- This simple flow document includes a paragraph with some  
    bold text in it and a list. -->  
<FlowDocumentReader  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
    <FlowDocument>
```

```

<Paragraph>
    <Bold>Some bold text in the paragraph.</Bold>
    Some text that is not bold.
</Paragraph>

<List>
    <ListItem>
        <Paragraph>ListItem 1</Paragraph>
    </ListItem>
    <ListItem>
        <Paragraph>ListItem 2</Paragraph>
    </ListItem>
    <ListItem>
        <Paragraph>ListItem 3</Paragraph>
    </ListItem>
</List>

</FlowDocument>
</FlowDocumentReader>

```

C#

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class SimpleFlowExample : Page
    {
        public SimpleFlowExample()
        {

            Paragraph myParagraph = new Paragraph();

            // Add some Bold text to the paragraph
            myParagraph.Inlines.Add(new Bold(new Run("Some bold text in the
paragraph.")));

            // Add some plain text to the paragraph
            myParagraph.Inlines.Add(new Run(" Some text that is not
bold."));

            // Create a List and populate with three list items.
            List myList = new List();

            // First create paragraphs to go into the list item.
            Paragraph paragraphListItem1 = new Paragraph(new Run("ListItem
1"));
            Paragraph paragraphListItem2 = new Paragraph(new Run("ListItem
2"));
            Paragraph paragraphListItem3 = new Paragraph(new Run("ListItem
3"));

```

```

3"));

        // Add ListItems with paragraphs in them.
        myList.ListItems.Add(new ListItem(paragraphListItem1));
        myList.ListItems.Add(new ListItem(paragraphListItem2));
        myList.ListItems.Add(new ListItem(paragraphListItem3));

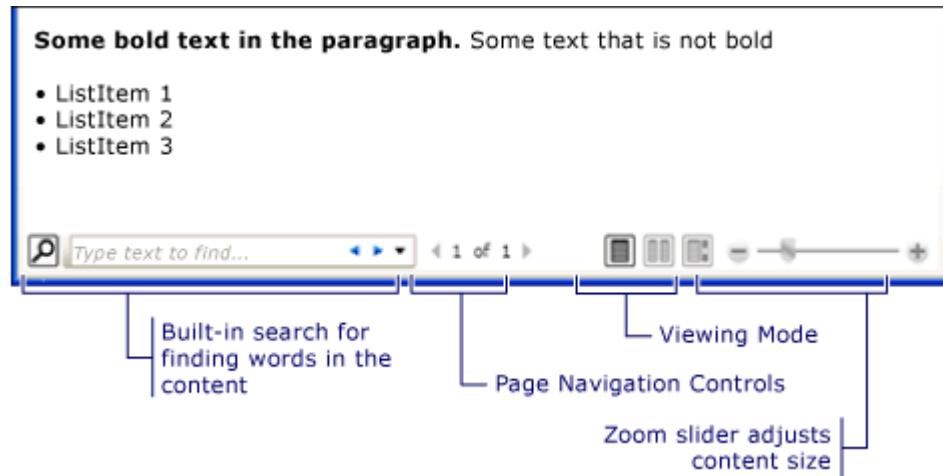
        // Create a FlowDocument with the paragraph and list.
        FlowDocument myFlowDocument = new FlowDocument();
        myFlowDocument.Blocks.Add(myParagraph);
        myFlowDocument.Blocks.Add(myList);

        // Add the FlowDocument to a FlowDocumentReader Control
        FlowDocumentReader myFlowDocumentReader = new
        FlowDocumentReader();
        myFlowDocumentReader.Document = myFlowDocument;

        this.Content = myFlowDocumentReader;
    }
}
}
}

```

下图显示了此代码片段。



在此示例中，**FlowDocumentReader** 控件用于托管流内容。有关流内容托管控件的详细信息，请参阅[流文档类型](#)。**Paragraph**、**List**、**ListItem** 和 **Bold** 元素用于根据其在标记中的顺序来控制内容格式。例如，**Bold** 元素只涵盖该段落中的一部分文本；因此，只有这一部分文本是粗体。如果使用过 HTML，你就会了解这一点。

正如上图中突出显示的那样，流文档中有多个内置功能：

- **搜索**：允许用户对整个文档执行全文搜索。
- **查看模式**：用户可选择喜欢的查看模式，包括单页（一次一页）查看模式、一次两页（书本阅读格式）查看模式和连续滚动（无界限）查看模式。有关这些查看模式的详细信息，请参阅 [FlowDocumentReaderViewingMode](#)。

- **页面导航控件**：如果文档的查看模式使用页面，则页面导航控件包括一个用于跳转到下一页（向下键）或上一页（向上键）的按钮，以及显示当前页码和总页数的指示器。也可使用键盘上的箭头键来实现翻页操作。
- **缩放**：缩放控件可使用户通过单击加号或减号按钮来相应地增大或减小缩放级别。缩放控件还包括一个用于调整缩放级别的滑块。有关详细信息，请参阅 [Zoom](#)。

这些功能可根据用于托管流内容的控件进行修改。下一节介绍了各种控件。

## 流文档类型

流文档内容的显示和外观依赖于用于托管流内容的对象。有 4 个支持查看流内容的控件：[FlowDocumentReader](#)、[FlowDocumentPageViewer](#)、[RichTextBox](#) 和 [FlowDocumentScrollView](#)。下面简要介绍了这些控件。

### ① 备注

需使用 [FlowDocument](#) 来直接托管流内容，因此所有这些查看控件都使用 [FlowDocument](#) 来启用流内容托管。

## FlowDocumentReader

[FlowDocumentReader](#) 包含使用户能够动态选择各种查看模式的功能，这些查看模式包括单页（一次一页）查看模式、一次两页（书本阅读格式）查看模式和连续滚动（无界限）查看模式。有关这些查看模式的详细信息，请参阅 [FlowDocumentReaderViewingMode](#)。如果不需要在不同查看模式之间动态切换的功能，则可使用 [FlowDocumentPageViewer](#) 和 [FlowDocumentScrollView](#)，它们提供了固定使用特定查看模式的轻量级流内容查看器。

## FlowDocumentPageViewer 和 FlowDocumentScrollView

[FlowDocumentPageViewer](#) 以一次一页的查看模式显示内容，而 [FlowDocumentScrollView](#) 以连续滚动模式显示内容。[FlowDocumentPageViewer](#) 和 [FlowDocumentScrollView](#) 都固定使用特定查看模式。相比之下，[FlowDocumentReader](#) 包含的功能使用户能够动态选择各种查看模式（由 [FlowDocumentReaderViewingMode](#) 枚举提供），但代价是需要消耗比 [FlowDocumentPageViewer](#) 或 [FlowDocumentScrollView](#) 更多的资源。

默认情况下，总是显示垂直滚动条，而水平滚动条则在需要时显示。[FlowDocumentScrollView](#) 的默认 UI 不包括工具栏；不过，可使用 [IsToolBarVisible](#) 属性来启用内置工具栏。

## RichTextBox

若要允许用户编辑流内容，请使用 [RichTextBox](#)。例如，如果希望创建一个允许用户处理表、斜体和粗体格式等内容的编辑器，则应使用 [RichTextBox](#)。有关详细信息，请参阅 [RichTextBox 概述](#)。

### ① 备注

[RichTextBox](#) 内部的流内容行为与其他控件中包含的流内容行为并不完全相同。例如，[RichTextBox](#) 中没有列，因此没有自动调整大小行为。另外，在 [RichTextBox](#) 中不能使用通常内置在流内容中的功能，例如搜索、查看模式、页面导航和缩放。

## 创建流内容

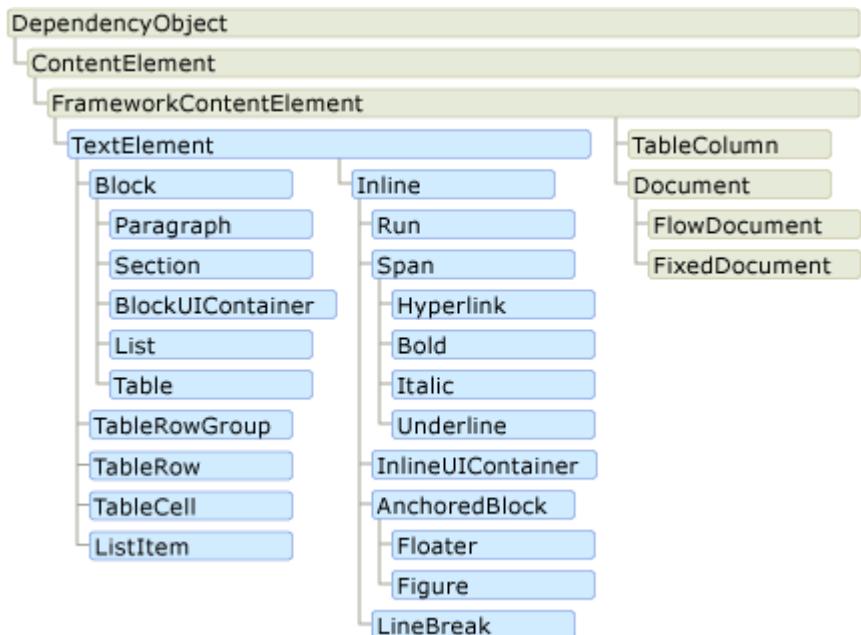
流内容可能很复杂并包含各种元素，包括文本、图像、表甚至像控件这样的 [UIElement](#) 派生类。若要了解如何创建复杂流内容，掌握下列知识点非常关键：

- **与流相关的类**：流内容中使用的每个类都有特定用途。此外，了解各种流类之间的层次关系有助于了解其使用方式。例如，从 [Block](#) 类派生的类用于包含其他对象，而从 [Inline](#) 派生的类包含显示的对象。
- **内容架构**：流文档可能需要大量嵌套元素。内容架构指定了元素之间可能存在的父/子关系。

以下各节详细介绍了上述各个方面。

## 与流相关的类

下图演示了流内容中最常使用的对象：



根据流内容的用途，可分为两个重要类别：

1. **Block 派生类**：也称为“Block 内容元素”，或简称为“Block 元素”。继承自 [Block](#) 的元素可用于将元素分组到一个公用父级下，或将公用属性应用于某个组。
2. **Inline 派生类**：也称为“Inline 内容元素”，或简称为“Inline 元素”。继承自 [Inline](#) 的元素要么包含在 Block 元素中，要么包含在另一个 Inline 元素中。Inline 元素通常用作在屏幕上呈现的内容的直接容器。例如，[Paragraph](#) ( Block 元素 ) 可包含 [Run](#) ( Inline 元素 )，而 [Run](#) 实际包含在屏幕上呈现的文本。

下面简要介绍了这两个类别中的每个类。

## Block 派生类

### Paragraph

[Paragraph](#) 常用于将内容分组到一个段落中。 Paragraph 的最简单且最常见的用途是创建文本段落。

XAML
<pre> &lt;FlowDocument   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"&gt;   &lt;Paragraph&gt;     Some paragraph text.   &lt;/Paragraph&gt; &lt;/FlowDocument&gt;   </pre>

C#

```

using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class ParagraphExample : Page
    {
        public ParagraphExample()
        {

            // Create paragraph with some text.
            Paragraph myParagraph = new Paragraph();
            myParagraph.Inlines.Add(new Run("Some paragraph text."));

            // Create a FlowDocument and add the paragraph to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}

```

不过，也可以包含其他 `Inline` 派生元素，如下所示。

## 节

`Section` 只用于包含其他 `Block` 派生元素。 它不会向其中包含的元素应用任何默认格式。但是，为 `Section` 设置的任何属性值都适用于其子元素。 使用节能够以编程方式循环访问其子集合。`Section` 的使用方式类似于 HTML 中的 `<DIV>` 标记。

以下示例在一个 `Section` 下定义了 3 个段落。该节具有 `Background` 属性值 `Red`，因此段落的背景色也是红色。

### XAML

```

<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <!-- By default, Section applies no formatting to elements contained
        within it. However, in this example, the section has a Background
        property value of "Red", therefore, the three paragraphs (the block)
        inside the section also have a red background. -->
    <Section Background="Red">
        <Paragraph>
            Paragraph 1
        </Paragraph>
        <Paragraph>
            Paragraph 2
        </Paragraph>
    </Section>

```

```
</Paragraph>
<Paragraph>
    Paragraph 3
</Paragraph>
</Section>
</FlowDocument>
```

C#

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class SectionExample : Page
    {
        public SectionExample()
        {

            // Create three paragraphs
            Paragraph myParagraph1 = new Paragraph(new Run("Paragraph 1"));
            Paragraph myParagraph2 = new Paragraph(new Run("Paragraph 2"));
            Paragraph myParagraph3 = new Paragraph(new Run("Paragraph 3"));

            // Create a Section and add the three paragraphs to it.
            Section mySection = new Section();
            mySection.Background = Brushes.Red;

            mySection.Blocks.Add(myParagraph1);
            mySection.Blocks.Add(myParagraph2);
            mySection.Blocks.Add(myParagraph3);

            // Create a FlowDocument and add the section to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(mySection);

            this.Content = myFlowDocument;
        }
    }
}
```

## BlockUIContainer

BlockUIContainer 使 [UIElement](#) 元素（即 [Button](#)）能够嵌入到 Block 派生的流内容中。

[InlineUIContainer](#)（见下文）用于在 [Inline](#) 派生的流内容中嵌入 [UIElement](#) 元素。

BlockUIContainer 和 [InlineUIContainer](#) 很重要，因为除非 [UIElement](#) 包含在这两个元素之一中，否则没有其他办法在流内容中使用它。

以下示例演示如何使用 `BlockUIContainer` 元素在流内容中托管 `UIElement` 对象。

#### XAML

```
<FlowDocument ColumnWidth="400">
    <Section Background="GhostWhite">
        <Paragraph>
            A UIElement element may be embedded directly in flow content
            by enclosing it in a BlockUIContainer element.
        </Paragraph>
        <BlockUIContainer>
            <Button>Click me!</Button>
        </BlockUIContainer>
        <Paragraph>
            The BlockUIContainer element may host no more than one top-level
            UIElement. However, other UIElements may be nested within the
            UIElement contained by an BlockUIContainer element. For example,
            a StackPanel can be used to host multiple UIElement elements within
            a BlockUIContainer element.
        </Paragraph>
        <BlockUIContainer>
            <StackPanel>
                <Label Foreground="Blue">Choose a value:</Label>
                <ComboBox>
                    <ComboBoxItem IsSelected="True">a</ComboBoxItem>
                    <ComboBoxItem>b</ComboBoxItem>
                    <ComboBoxItem>c</ComboBoxItem>
                </ComboBox>
                <Label Foreground ="Red">Choose a value:</Label>
                <StackPanel>
                    <RadioButton>x</RadioButton>
                    <RadioButton>y</RadioButton>
                    <RadioButton>z</RadioButton>
                </StackPanel>
                <Label>Enter a value:</Label>
                <TextBox>
                    A text editor embedded in flow content.
                </TextBox>
            </StackPanel>
        </BlockUIContainer>
    </Section>
</FlowDocument>
```

下图显示了此示例的呈现效果：

A UIElement element may be embedded directly in flow content by enclosing it in a BlockUIContainer element.

Click me!

The BlockUIContainer element may host no more than one top-level UIElement. However, other UIElements may be nested within the UIElement contained by an BlockUIContainer element. For example, a StackPanel can be used to host multiple UIElement elements within a BlockUIContainer element.

Choose a value:

a

Choose a value:

- x
- y
- z

Enter a value:

A text editor embedded in flow content.

## 列表

List 用于创建项目符号列表或编号列表。将 MarkerStyle 属性设置为 TextMarkerStyle 枚举值可确定列表的样式。下例演示了如何创建简单列表。

XAML

```
<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <List>
        <ListItem>
            <Paragraph>
                List Item 1
            </Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>
                List Item 2
            </Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>
                List Item 3
            </Paragraph>
        </ListItem>
    </List>
</FlowDocument>
```

C#

```

using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class ListExample : Page
    {
        public ListExample()
        {

            // Create three paragraphs
            Paragraph myParagraph1 = new Paragraph(new Run("List Item 1"));
            Paragraph myParagraph2 = new Paragraph(new Run("List Item 2"));
            Paragraph myParagraph3 = new Paragraph(new Run("List Item 3"));

            // Create the ListItem elements for the List and add the
            // paragraphs to them.
            ListItem myListItem1 = new ListItem();
            myListItem1.Blocks.Add(myParagraph1);
            ListItem myListItem2 = new ListItem();
            myListItem2.Blocks.Add(myParagraph2);
            ListItem myListItem3 = new ListItem();
            myListItem3.Blocks.Add(myParagraph3);

            // Create a List and add the three ListItems to it.
            List myList = new List();

            myList.ListItems.Add(myListItem1);
            myList.ListItems.Add(myListItem2);
            myList.ListItems.Add(myListItem3);

            // Create a FlowDocument and add the section to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myList);

            this.Content = myFlowDocument;
        }
    }
}

```

## ① 备注

List 是唯一一个使用 ListItemCollection 来管理子元素的流元素。

## 表

[Table](#) 用于创建表。 [Table](#) 与 [Grid](#) 元素类似，但是前者具有更多功能，因此需要更大的资源开销。因为 [Grid](#) 是一个 [UIElement](#)，所以除非它包含在 [BlockUIContainer](#) 或 [InlineUIContainer](#) 中，否则不能在流内容中使用。有关 [Table](#) 的详细信息，请参阅[表概述](#)。

## Inline 派生类

### Run

[Run](#) 用于包含无格式文本。你可能会看到 [Run](#) 对象广泛用于流内容。但是，在标记中，无需显式使用 [Run](#) 元素。使用代码创建或操作流文档时，需使用 [Run](#)。例如，在下面的标记中，第一个 [Paragraph](#) 显式指定了 [Run](#) 元素，而第二个却没有。这两个段落生成相同的输出。

#### XAML

```
<Paragraph>
    <Run>Paragraph that explicitly uses the Run element.</Run>
</Paragraph>

<Paragraph>
    This Paragraph omits the Run element in markup. It renders
    the same as a Paragraph with Run used explicitly.
</Paragraph>
```

#### ① 备注

从 .NET Framework 4 开始，[Run](#) 对象的 [Text](#) 属性为依赖属性。可将 [Text](#) 属性绑定到数据源，例如 [TextBlock](#)。[Text](#) 属性完全支持单向绑定。[Text](#) 属性还支持双向绑定，[RichTextBox](#) 除外。有关示例，请参见 [Run.Text](#)。

### 跨越

[Span](#) 将其他内联内容元素组合到一起。对于 [Span](#) 元素中的内容，不应用任何固有呈现。但是，从 [Span](#) 继承的元素（包括 [Hyperlink](#)、[Bold](#)、[Italic](#) 和 [Underline](#)）会向文本应用格式设置。

下面是 [Span](#) 的一个示例，它用于包含内联内容，包括文本、一个 [Bold](#) 元素和一个 [Button](#)。

#### XAML

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

```

<Paragraph>
    Text before the Span. <Span Background="Red">Text within the Span is
    red and <Bold>this text is inside the Span-derived element Bold.</Bold>
    A Span can contain more than text, it can contain any inline content.
    For
        example, it can contain a
        <InlineUIContainer>
            <Button>Button</Button>
        </InlineUIContainer>
        or other UIElement, a Floater, a Figure, etc.</Span>
    </Paragraph>

</FlowDocument>

```

下面的屏幕截图显示了此示例的呈现效果。

Text before the Span. Text within the Span is red and **this text is inside the Span-derived element Bold.** A Span can contain more than text, it can contain any inline content. For example, it can contain a **Button** or other UIElement, a Floater, a Figure, etc.

## InlineUIContainer

InlineUIContainer 使 UIElement 元素（即 Button 这样的控件）能够嵌入到 Inline 内容元素中。此元素是与上述 BlockUIContainer 等效的 Inline 元素。以下示例使用 InlineUIContainer 将 Button 以内联方式插入 Paragraph 中。

### XAML

```

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

    <Paragraph>
        Text to precede the button...

        <!-- Set the BaselineAlignment property to "Bottom"
            so that the Button aligns properly with the text. -->
        <InlineUIContainer BaselineAlignment="Bottom">
            <Button>Button</Button>
        </InlineUIContainer>
        Text to follow the button...
    </Paragraph>

</FlowDocument>

```

### C#

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class InlineUIContainerExample : Page
    {
        public InlineUIContainerExample()
        {
            Run run1 = new Run(" Text to precede the button... ");
            Run run2 = new Run(" Text to follow the button... ");

            // Create a new button to be hosted in the paragraph.
            Button myButton = new Button();
            myButton.Content = "Click me!";

            // Create a new InlineUIContainer to contain the Button.
            InlineUIContainer myInlineUIContainer = new InlineUIContainer();

            // Set the BaselineAlignment property to "Bottom" so that the
            // Button aligns properly with the text.
            myInlineUIContainer.BaselineAlignment =
BaselineAlignment.Bottom;

            // Assign the button as the UI container's child.
            myInlineUIContainer.Child = myButton;

            // Create the paragraph and add content to it.
            Paragraph myParagraph = new Paragraph();
            myParagraph.Inlines.Add(run1);
            myParagraph.Inlines.Add(myInlineUIContainer);
            myParagraph.Inlines.Add(run2);

            // Create a FlowDocument and add the paragraph to it.
            FlowDocument myFlowDocument = new FlowDocument();
            myFlowDocument.Blocks.Add(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}
```

## ① 备注

不需要在标记中显式使用 `InlineUIContainer`。如果将其省略，编译代码时仍将创建一个 `InlineUIContainer`。

## Figure 和 Floater

通过 [Figure](#) 和 [Floater](#)，可使用位置属性在流文档中嵌入内容，这些属性可独立于主内容流进行自定义。 [Figure](#) 或 [Floater](#) 元素常用于突出显示或强调内容的某些部分，托管主内容流中的支持图像或其他内容，或用于注入松散相关的内容，例如广告。

以下示例演示如何将 [Figure](#) 嵌入文本段落中。

XAML

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

    <Paragraph>
        <Figure
            Width="300" Height="100"
            Background="GhostWhite" HorizontalAnchor="PageLeft" >
            <Paragraph FontStyle="Italic" Background="Beige"
Foreground="DarkGreen" >
                A Figure embeds content into flow content with placement properties
                that can be customized independently from the primary content flow
            </Paragraph>
        </Figure>
        Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam
        nonummy
        nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut
        wisi
        enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
        lobortis
        nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
    </Paragraph>

</FlowDocument>
```

C#

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class FigureExample : Page
    {
        public FigureExample()
        {

            // Create strings to use as content.
            string strFigure = "A Figure embeds content into flow content";
        }
    }
}
```

```

with" +
                    " placement properties that can be
customized" +
                    " independently from the primary content
flow";
        string strOther = "Lorem ipsum dolor sit amet, consectetur
adipiscing" +
                    " elit, sed diam nonummy nibh euismod
tincidunt ut laoreet" +
                    " dolore magna aliquam erat volutpat. Ut wisi
enim ad" +
                    " minim veniam, quis nostrud exerci tation
ullamcorper" +
                    " suscipit lobortis nisl ut aliquip ex ea
commodo consequat." +
                    " Duis autem vel eum iriure.";

        // Create a Figure and assign content and layout properties to
it.
        Figure myFigure = new Figure();
        myFigure.Width = new FigureLength(300);
        myFigure.Height = new FigureLength(100);
        myFigure.Background = Brushes.GhostWhite;
        myFigure.HorizontalAnchor = FigureHorizontalAnchor.PageLeft;
        Paragraph myFigureParagraph = new Paragraph(new Run(strFigure));
        myFigureParagraph.FontStyle = FontStyles.Italic;
        myFigureParagraph.Background = Brushes.Beige;
        myFigureParagraph.Foreground = Brushes.DarkGreen;
        myFigure.Blocks.Add(myFigureParagraph);

        // Create the paragraph and add content to it.
        Paragraph myParagraph = new Paragraph();
        myParagraph.Inlines.Add(myFigure);
        myParagraph.Inlines.Add(new Run(strOther));

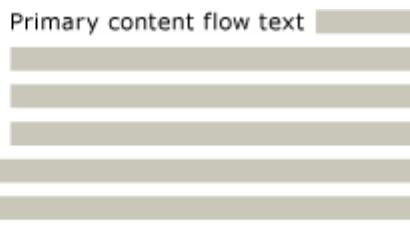
        // Create a FlowDocument and add the paragraph to it.
        FlowDocument myFlowDocument = new FlowDocument();
        myFlowDocument.Blocks.Add(myParagraph);

        this.Content = myFlowDocument;
    }
}
}

```

下图显示了此示例的呈现效果。

*A Figure embeds content into flow content with placement properties that can be customized independently from the primary content flow*



[Figure](#) 和 [Floater](#) 在多个方面存在差异，并用于不同的方案。

#### Figure :

- 可定位：可设置其水平和垂直定位点，以便相对于页面、内容、栏或段落进行停靠。还可使用其 [HorizontalOffset](#) 和 [VerticalOffset](#) 属性指定任意偏移量。
- 可将其大小调整为列大小的几倍：可将 [Figure](#) 的高度和宽度设置为页面、内容或列的高度或宽度的倍数。请注意，对于页面和内容，倍数不能大于 1。例如，可将 [Figure](#) 的宽度设置为“页面的 0.5 倍”、“内容的 0.25 倍”或“列的 2 倍”。还可将高度和宽度设置为绝对像素值。
- 不分页：如果 [Figure](#) 中的内容无法容纳在 [Figure](#) 内部，它会呈现能够容纳的内容部分，而其余内容将丢失。

#### Floater :

- 无法定位，可在能够为其提供空间的任何位置呈现。不能设置偏移量或锚定 [Floater](#)。
- 不能将其大小调整为列大小的几倍：默认情况下，[Floater](#) 的大小为 1 个列大小。它有一个可设置为绝对像素值的 [Width](#) 属性，但是如果此值大于 1 个列宽，则将其忽略并将浮动对象的大小调整为 1 个列大小。可通过设置正确的像素宽度将其大小调整为小于 1 个列宽，但调整大小与列无关，因此“0.5 倍列宽”并不是 [Floater](#) 宽度的有效表达。[Floater](#) 没有高度属性，无法设置其高度；其高度取决于内容
- [Floater](#) 分页：如果指定宽度的内容超出了 1 个列高，则浮动对象会断开并显示到下一列、下一页等。

[Figure](#) 适合放置希望控制其大小和位置的独立内容，并且可以确信内容适合指定的大小。[Floater](#) 适合放置流动更加自由的内容，其流动方式与主页内容类似，但与主页内容相分离。

#### LineBreak

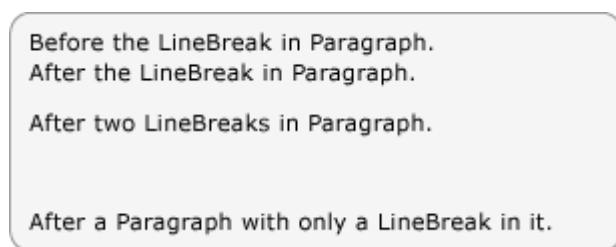
[LineBreak](#) 导致在流内容中发生换行。以下示例演示了 [LineBreak](#) 的用法。

##### XAML

```
<FlowDocument  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">  
    <Paragraph>  
        Before the LineBreak in Paragraph.  
        <LineBreak />  
        After the LineBreak in Paragraph.  
        <LineBreak/><LineBreak/>
```

```
After two LineBreaks in Paragraph.  
</Paragraph>  
  
<Paragraph>  
    <LineBreak/>  
</Paragraph>  
  
<Paragraph>  
    After a Paragraph with only a LineBreak in it.  
</Paragraph>  
</FlowDocument>
```

下面的屏幕截图显示了此示例的呈现效果。



Before the LineBreak in Paragraph.  
After the LineBreak in Paragraph.  
After two LineBreaks in Paragraph.  
  
After a Paragraph with only a LineBreak in it.

## 流集合元素

在上面的多个示例中，[BlockCollection](#) 和 [InlineCollection](#) 用于以编程方式构造流内容。例如，若要向 [Paragraph](#) 添加元素，可使用以下语法：

```
C#  
  
myParagraph.Inlines.Add(new Run("Some text"));
```

这会将 [Run](#) 添加到 [Paragraph](#) 的 [InlineCollection](#)。这与标记中的 [Paragraph](#) 内部包含的隐式 [Run](#) 相同：

```
XML  
  
<Paragraph>  
    Some Text  
</Paragraph>
```

作为使用 [BlockCollection](#) 的示例，以下示例创建了一个新的 [Section](#)，然后使用 [Add](#) 方法将一个新的 [Paragraph](#) 添加到 [Section](#) 内容中。

```
C#  
  
Section secx = new Section();  
secx.Blocks.Add(new Paragraph(new Run("A bit of text content...")));
```

除向流集合中添加项之外，还可以移除项。以下示例将删除 `Span` 中的最后一个 `Inline` 元素。

C#

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

以下示例将从 `Span` 中清除所有内容（`Inline` 元素）。

C#

```
spanx.Inlines.Clear();
```

以编程方式使用流内容时，可能会广泛使用这些集合。

流元素是使用 `InlineCollection` (`Inline`) 还是 `BlockCollection` (`Block`) 来包含其子元素取决于父级可以包含的子元素类型（`Block` 或 `Inline`）。下一节中的内容架构中概述了流内容元素的包容规则。

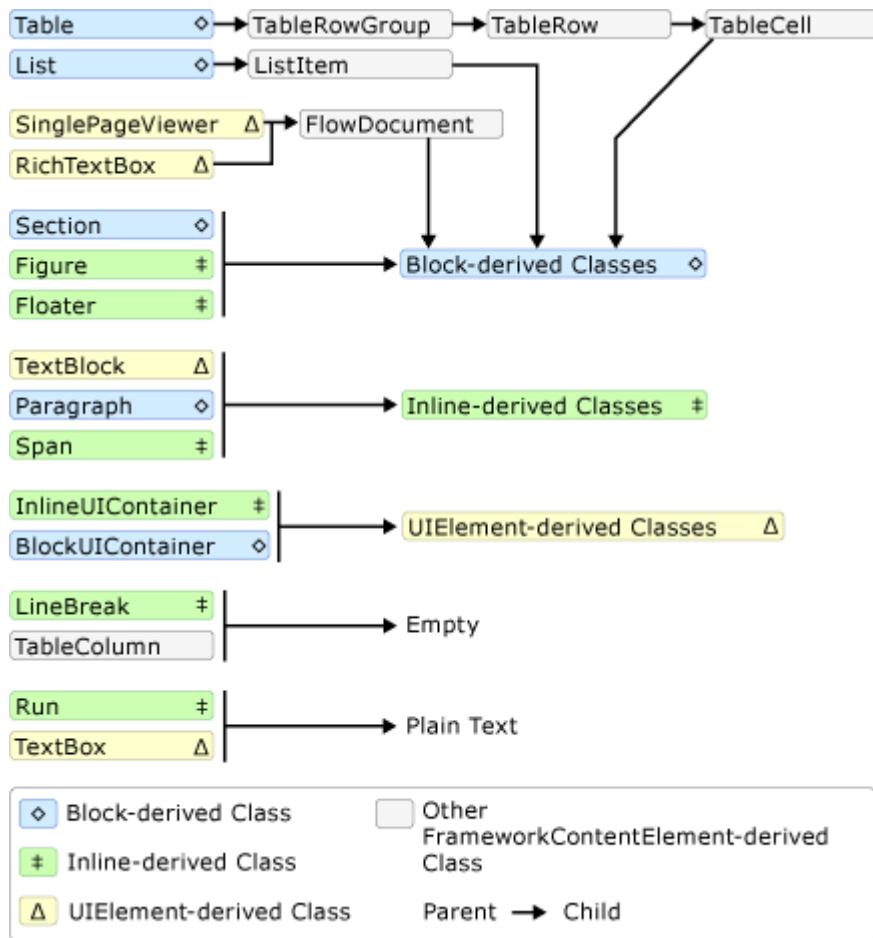
### ① 备注

还有第三种类型的集合可用于流内容，即 `ListItemCollection`，但此集合仅用于 `List`。此外，还有几个可用于 `Table` 的集合。有关详细信息，请参阅[表概述](#)。

## 内容架构

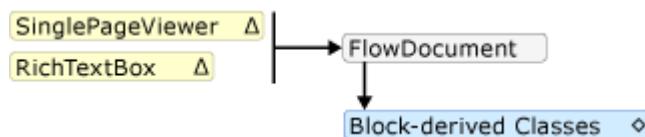
不同流内容元素的数量是如此之多，因此了解某个元素可包含的子元素类型非常困难。

下面的关系图概述了流元素的包容规则。箭头表示可能存在的父/子关系。



如上面的关系图所示，元素可以具有的子元素不一定通过该元素是 Block 元素还是 Inline 元素来确定。例如，**Span** ( Inline 元素 ) 只能具有 Inline 子元素，而 **Figure** ( 也是 Inline 元素 ) 只能具有 Block 子元素。因此，关系图可用于快速确定哪些元素可以包含在其他元素中。例如，可使用关系图来确定如何构造 **RichTextBox** 的流内容。

1. **RichTextBox** 必须包含 **FlowDocument**，而后者又必须包含 **Block** 派生对象。下面是上述关系图中的对应部分。

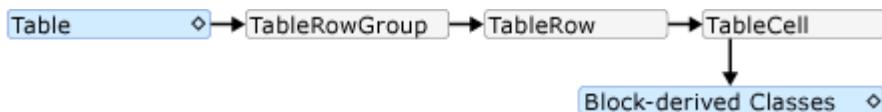


到此为止，标记可能类似于所示内容。

XAML
<pre> &lt;RichTextBox&gt;   &lt;FlowDocument&gt;     &lt;!-- One or more Block-derived object... --&gt;   &lt;/FlowDocument&gt; &lt;/RichTextBox&gt;   </pre>

2. 按照该关系图，存在多个可从中进行选择的 Block 元素，包括 **Paragraph**、**Section**、**Table**、**List** 和 **BlockUIContainer** ( 请参阅上面的 Block 派生类 )。假设需要一个 **Table**。

按照上面的关系图，Table 包含一个 TableRowGroup，后者包含 TableRow 元素，这些元素又包含 TableCell 元素，而这些元素包含一个 Block 派生对象。下面是取自上述关系图中 Table 的对应部分。

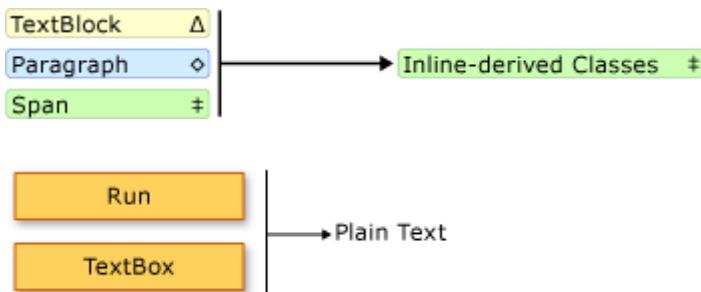


下面是对应的标记。

XAML

```
<RichTextBox>
  <FlowDocument>
    <Table>
      <TableRowGroup>
        <TableRow>
          <TableCell>
            <!-- One or more Block-derived object... -->
          </TableCell>
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
</RichTextBox>
```

3. 同样，TableCell 下需要一个或多个 Block 元素。为简单起见，在单元格内部放置一些文本。可以使用带有 Run 元素的 Paragraph 来实现该操作。下面是该关系图中的对应部分，它显示 Paragraph 可以包含 Inline 元素，而 Run ( Inline 元素 ) 只能包含纯文本。



下面是标记中的完整示例。

XAML

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <RichTextBox>
    <FlowDocument>
      <!-- Normally a table would have multiple rows and multiple
           cells but this code is for demonstration purposes.-->
      <Table>
```

```
<TableRowGroup>
  <TableRow>
    <TableCell>
      <Paragraph>

        <!-- The schema does not actually require
            explicit use of the Run tag in markup. It
            is only included here for clarity. -->
        <Run>Paragraph in a Table Cell.</Run>
      </Paragraph>
    </TableCell>
  </TableRow>
</TableRowGroup>
</Table>

</FlowDocument>
</RichTextBox>
</Page>
```

## 自定义文本

通常，文本是流文档中最普遍的内容类型。 尽管上面介绍的对象可用于控制文本呈现方式的大多数方面，但本文还介绍了其他一些自定义文本的方法。

## 文本修饰

使用文本修饰，可向文本应用下划线、上划线、基线和删除线效果（请参见下图）。 这些修饰是使用 [TextDecorations](#) 属性添加的，该属性由许多对象公开，其中包括 [Inline](#)、[Paragraph](#)、[TextBlock](#) 和 [TextBox](#)。

以下示例演示如何设置 [TextDecorations](#) 的 [Paragraph](#) 属性。

XAML

```
<FlowDocument ColumnWidth="200">
  <Paragraph TextDecorations="Strikethrough">
    This text will render with the strikethrough effect.
  </Paragraph>
</FlowDocument>
```

C#

```
Paragraph parx = new Paragraph(new Run("This text will render with the
strikethrough effect."));
parx.TextDecorations = TextDecorations.Strikethrough;
```

下图显示了此示例的呈现效果。

This text will render with the default strikethrough effect.

下面各图分别显示了上划线、基线和下划线修饰的呈现效果。

This text will render with the default overline effect.

This text will render with the default baseline effect.

This text will render with the default underline effect.

## 版式

`Typography` 属性由大多数与流相关的内容公开，其中包括 `TextElement`、`FlowDocument`、`TextBlock` 和 `TextBox`。此属性用于控制文本的版式特征/变体（即小型大写字母或大型大写字母、设置上标和下标等）。

以下示例将 `Paragraph` 作为示例元素，演示如何设置 `Typography` 属性。

XAML

```
<Paragraph
    TextAlignment="Left"
    FontSize="18"
    FontFamily="Palatino Linotype"
    Typography.NumeralStyle="OldStyle"
    Typography.Fraction="Stacked"
    Typography.Variants="Inferior"
>
<Run>
    This text has some altered typography characteristics. Note
    that use of an open type font is necessary for most typographic
    properties to be effective.
</Run>
<LineBreak/><LineBreak/>
<Run>
    0123456789 10 11 12 13
</Run>
<LineBreak/><LineBreak/>
<Run>
    1/2 2/3 3/4
</Run>
</Paragraph>
```

下图显示了此示例的呈现效果。

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

$\frac{1}{2}$   $\frac{2}{3}$   $\frac{3}{4}$

与此相反，下图显示了一个具有默认版式属性的类似示例的呈现效果。

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

1/2 2/3 3/4

以下示例演示如何以编程方式设置 **Typography** 属性。

C#

```
Paragraph par = new Paragraph();

Run runText = new Run(
    "This text has some altered typography characteristics. Note" +
    "that use of an open type font is necessary for most typographic" +
    "properties to be effective.");
Run runNumerals = new Run("0123456789 10 11 12 13");
Run runFractions = new Run("1/2 2/3 3/4");

par.Inlines.Add(runText);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runNumerals);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runFractions);

par.TextAlignment = TextAlignment.Left;
par.FontSize = 18;
par.FontFamily = new FontFamily("Palatino Linotype");

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle;
par.Typography.Fraction = FontFraction.Stacked;
par.Typography.Variants = FontVariants.Inferior;
```

有关版式的详细信息，请参阅 [WPF 中的版式](#)。

## 另请参阅

- [文本](#)
- [WPF 中的版式](#)
- [操作指南主题](#)
- [TextElement 内容模型概述](#)
- [RichTextBox 概述](#)
- [WPF 中的文档](#)
- [表概述](#)
- [批注概述](#)

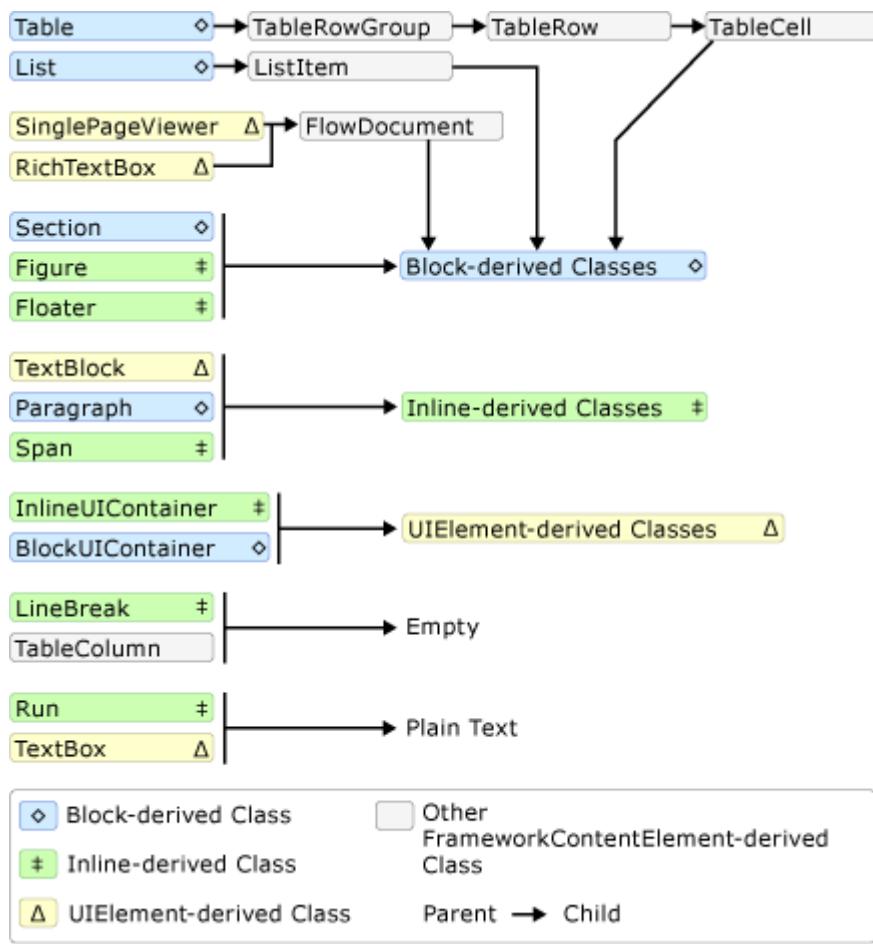
# TextElement 内容模型概述

项目 • 2023/02/06

本内容模型概述介绍 [TextElement](#) 的支持内容。 [Paragraph](#) 类是 [TextElement](#) 的一种类型。 内容模型描述哪些对象/元素可包含在其他对象/元素中。 本概述概括了派生自 [TextElement](#) 的对象所使用的内容模型。 有关详细信息，请参阅[流文档概述](#)。

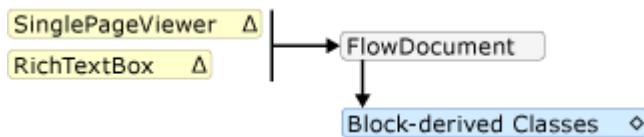
## 内容模型图

下图概括了派生自 [TextElement](#) 的类的内容模型，以及其他非 [TextElement](#) 类是如何与该模型相适应的。



如上面的关系图所示，元素可以具有的子元素不一定通过某个类是派生自 [Block](#) 类还是 [Inline](#) 类来确定。例如，[Span](#)（从 [Inline](#) 派生的类）只能具有 [Inline](#) 子元素，但是 [Figure](#)（也是从 [Inline](#) 派生的类）只能具有 [Block](#) 子元素。因此，关系图可用于快速确定哪些元素可以包含在其他元素中。例如，可使用关系图来确定如何构造 [RichTextBox](#) 的流内容。

1. [RichTextBox](#) 必须包含 [FlowDocument](#)，而后者又必须包含 [Block](#) 派生的对象。以下是上述关系图中的相应部分。



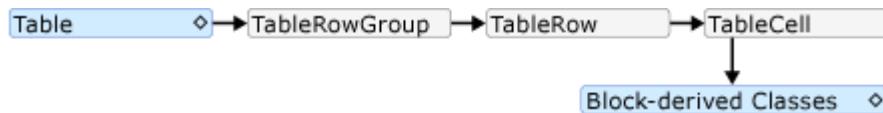
到此为止，标记可能类似于所示内容。

XAML

```

<RichTextBox>
  <FlowDocument>
    <!-- One or more Block-derived object... -->
  </FlowDocument>
</RichTextBox>
  
```

- 按照该关系图，存在多个可以从中进行选择的 **Block** 元素，包括 **Paragraph**、**Section**、**Table**、**List** 和 **BlockUIContainer**（请参阅上图中从 **Block** 派生的类）。假设需要一个 **Table**。按照上述关系图，**Table** 包含一个 **TableRowGroup**，后者包含多个 **TableRow** 元素，这些元素又包含多个 **TableCell** 元素，而这些元素包含一个从 **Block** 派生的对象。下面是取自上述关系图的 **Table** 的对应部分。



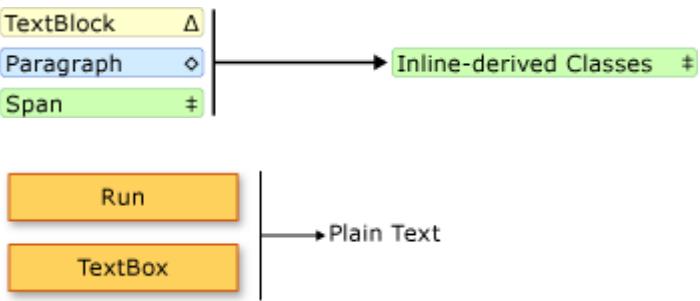
下面是相应的标记。

XAML

```

<RichTextBox>
  <FlowDocument>
    <Table>
      <TableRowGroup>
        <TableRow>
          <TableCell>
            <!-- One or more Block-derived object... -->
          </TableCell>
        </TableRow>
      </TableRowGroup>
    </Table>
  </FlowDocument>
</RichTextBox>
  
```

- 同样，**TableCell** 下面需要一个或多个 **Block** 元素。为简单起见，在单元格内部放置一些文本。可以使用带有 **Run** 元素的 **Paragraph** 来实现该操作。以下该关系图中的对应部分，其中显示，**Paragraph** 可以包含一个 **Inline** 元素，而 **Run**（一个 **Inline** 元素）只能包含纯文本。



下面是标记中的完整示例。

### XAML

```

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

            <!-- Normally a table would have multiple rows and multiple
                 cells but this code is for demonstration purposes.-->
            <Table>
                <TableRowGroup>
                    <TableRow>
                        <TableCell>
                            <Paragraph>

                                <!-- The schema does not actually require
                                     explicit use of the Run tag in markup. It
                                     is only included here for clarity. -->
                                <Run>Paragraph in a Table Cell.</Run>
                            </Paragraph>
                        </TableCell>
                    </TableRow>
                </TableRowGroup>
            </Table>

            </FlowDocument>
        </RichTextBox>
    </Page>

```

## 以编程方式处理 TextElement 内容

TextElement 的内容由集合组成，因此可以通过处理这些集合以编程方式处理 TextElement 对象的内容。从 TextElement 派生的类使用以下三个不同的集合：

- **InlineCollection**：表示 Inline 元素的集合。 InlineCollection 定义 Paragraph、Span 和 TextBlock 元素允许的子内容。
- **BlockCollection**：表示 Block 元素的集合。 BlockCollection 定义 FlowDocument、Section、ListItem、TableCell、Floater 和 Figure 元素允许的子内容。

- [ListItemCollection](#)：一个流内容元素，表示有序或无序的 [List](#) 中的一个特定内容项。

可以从这些集合操作、添加或移除项，方法是分别使用 [Inlines](#)、[Blocks](#) 和 [ListItems](#) 属性来实现。下面的示例演示如何使用 [Inlines](#) 属性来操作 [Span](#) 的内容。

### ① 备注

表格使用多个集合来操作其内容，但这里不对其进行说明。有关详细信息，请参阅 [表概述](#)。

以下示例创建新的 [Span](#) 对象，然后使用 [Add](#) 方法将两个文本运行添加为 [Span](#) 的内容子级。

C#

```
Span spanx = new Span();
spanx.Inlines.Add(new Run("A bit of text content..."));
spanx.Inlines.Add(new Run("A bit more text content..."));
```

下面的示例创建一个新的 [Run](#) 元素，并将其插入到 [Span](#) 的开头。

C#

```
Run runx = new Run("Text to insert...");
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx);
```

下面的示例删除 [Span](#) 中的最后一个 [Inline](#) 元素。

C#

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

以下示例将从 [Span](#) 中清除所有内容 ([Inline](#) 元素)。

C#

```
spanx.Inlines.Clear();
```

## 共享此内容模型的类型

下面的类型继承自 [TextElement](#) 类，可用来显示本概述中介绍的内容。

[Bold](#), [Figure](#), [Floater](#), [Hyperlink](#), [InlineUIContainer](#), [Italic](#), [LineBreak](#), [List](#), [ListItem](#), [Paragraph](#), [Run](#), [Section](#), [Span](#), [Table](#), [Underline](#).

请注意，此列表中仅包括与 Windows SDK 一起分发的非抽象类型。可以使用继承自 [TextElement](#) 的其他类型。

## 可包含 TextElement 对象的类型

请参阅 [WPF 内容模型](#)。

## 另请参阅

- 通过 [Blocks 属性操作 FlowDocument](#)
- 通过 [Blocks 属性操作流内容元素](#)
- 通过 [Blocks 属性操作 FlowDocument](#)
- 通过 [Columns 属性操作表的列](#)
- 通过 [RowGroups 属性操作表的行组](#)

# 表概述

项目 · 2023/02/06

[Table](#) 是一个块级元素，支持对流文档内容的基于网格的表示形式。此元素极具灵活性，因此很有用，但也因此显得更加复杂，从而不容易理解和正确使用。

本主题包含以下各节：

- [表基础](#)
- [表与网格有什么区别？](#)
- [基本表结构](#)
- [表包容](#)
- [行分组](#)
- [背景呈现优先级](#)
- [跨行或列](#)
- [使用代码生成表](#)
- [相关主题]

## 表基础

### 表与网格有什么区别？

[Table](#) 和 [Grid](#) 有一些共同的功能，但每个都有其各自最适合的场景。[Table](#) 设计为在流内容内使用（有关流内容的详细信息，请参阅[流文档概述](#)）。网格最适合在表单中（主要在流内容以外的任意位置）使用。在 [FlowDocument](#) 中，[Table](#) 支持分页、列重排和内容选择等流内容行为，而 [Grid](#) 不支持。另一方面，[Grid](#) 最适合在 [FlowDocument](#) 之外使用，其原因有多种，例如 [Grid](#) 基于行和列索引添加元素，而 [Table](#) 不是。[Grid](#) 元素支持对子内容进行分层，从而允许多个元素共存于单个“单元格”内，而 [Table](#) 不支持分层。[Grid](#) 的子元素可相对于其“单元格”边界区域进行绝对定位。[Table](#) 不支持此功能。最后，[Grid](#) 所需的资源比 [Table](#) 更少，因此请考虑使用 [Grid](#) 来提高性能。

### 基本表结构

[Table](#) 提供了一个基于网格的表示，其中包含列（由  [TableColumn](#) 元素表示）和行（由  [TableRow](#) 元素表示）。  [TableColumn](#) 元素不承载内容，只是定义列和列的特征。 [TableRow](#) 元素必须包含在  [TableRowGroup](#) 元素中，后者定义表的一组行。  [TableCell](#) 元素（包含表格要呈现的实际内容）必须包含在  [TableRow](#) 元素中。  [TableCell](#) 只能包含派生自  [Block](#) 的元素。 包括  [TableCell](#) 的有效子元素。

- [BlockUIContainer](#)
- [List](#)
- [Paragraph](#)
- [Section](#)
- [Table](#)

### ① 备注

[TableCell](#) 元素不可直接承载文本内容。有关流内容元素（例如  [TableCell](#)）的包含规则的详细信息，请参阅 [流文档概述](#)。

### ① 备注

[Table](#) 类似于  [Grid](#) 元素，但具有更多功能，因此所需的资源会更多。

以下示例使用 XAML 定义一个简单的 2 行 3 列的表。

#### XAML

```
<!--  
    Table is a Block element, and as such must be hosted in a container  
    for Block elements. FlowDocument provides such a container.  
-->  
<FlowDocument>  
<Table>  
<!--  
    This table has 3 columns, each described by a TableColumn  
    element nested in a Table.Columns collection element.  
-->  
<Table.Columns>  
    <TableColumn />  
    <TableColumn />  
    <TableColumn />  
</Table.Columns>  
<!--  
    This table includes a single TableRowGroup which hosts 2 rows,  
    each described by a TableRow element.  
-->
```

```

-->
<TableRowGroup>
  <!--
    Each of the 2 TableRow elements hosts 3 cells, described by
    TableCell elements.
  -->
  <TableRow>
    <TableCell>
      <!--
        TableCell elements may only host elements derived from Block.
        In this example, Paragraph elements serve as the ultimate content
        containers for the cells in this table.
      -->
      <Paragraph>Cell at Row 1 Column 1</Paragraph>
    </TableCell>
    <TableCell>
      <Paragraph>Cell at Row 1 Column 2</Paragraph>
    </TableCell>
    <TableCell>
      <Paragraph>Cell at Row 1 Column 3</Paragraph>
    </TableCell>
  </TableRow>
  <TableRow>
    <TableCell>
      <Paragraph>Cell at Row 2 Column 1</Paragraph>
    </TableCell>
    <TableCell>
      <Paragraph>Cell at Row 2 Column 2</Paragraph>
    </TableCell>
    <TableCell>
      <Paragraph>Cell at Row 2 Column 3</Paragraph>
    </TableCell>
  </TableRow>
</TableRowGroup>
</Table>
</FlowDocument>

```

下图显示了此示例的呈现效果。

Cell at Row 1 Column 1	Cell at Row 1 Column 2	Cell at Row 1 Column 3
Cell at Row 2 Column 1	Cell at Row 2 Column 2	Cell at Row 2 Column 3

## 表包容

Table 派生自 Block 元素，并遵守 Block 级别元素的通用规则。 Table 元素可包含在以下任意元素中：

- FlowDocument

- TableCell
- ListBoxItem
- ListViewItem
- Section
- Floater
- Figure

## 行分组

`TableRowGroup` 元素提供一种对表中的行进行任意分组的方式；表中的每行必须属于一个行分组。行分组中的行通常具有相同的用途，并可作为一个组来设置样式。行分组的一个常见使用方式是用于将特定用途的行（如标题行、标头和页脚行）与表中所含的主要内容分隔开来。

以下示例使用 XAML 定义具有带样式的标头行和页脚行的表。

XAML

```

<Table>
    <Table.Resources>
        <!-- Style for header/footer rows. -->
        <Style x:Key="headerFooterRowStyle" TargetType="{x:Type TableRowGroup}">
            <Setter Property="FontWeight" Value="DemiBold"/>
            <Setter Property="FontSize" Value="16"/>
            <Setter Property="Background" Value="LightGray"/>
        </Style>

        <!-- Style for data rows. -->
        <Style x:Key="dataRowStyle" TargetType="{x:Type TableRowGroup}">
            <Setter Property="FontSize" Value="12"/>
            <Setter Property="FontStyle" Value="Italic"/>
        </Style>
    </Table.Resources>

    <Table.Columns>
        < TableColumn /> < TableColumn /> < TableColumn /> < TableColumn />
    </Table.Columns>

    <!-- This TableRowGroup hosts a header row for the table. -->
    < TableRowGroup Style="{StaticResource headerFooterRowStyle}">
        < TableRow>
            < TableCell />
            < TableCell >< Paragraph >Gizmos</ Paragraph >< /TableCell >
            < TableCell >< Paragraph >Thingamajigs</ Paragraph >< /TableCell >
            < TableCell >< Paragraph >Doohickies</ Paragraph >< /TableCell >
        < /TableRow>
    < / TableRowGroup>
< /Table>

```

```

        </TableRow>
    </TableRowGroup>

    <!-- This TableRowGroup hosts the main data rows for the table. -->
    <TableRowGroup Style="{StaticResource dataRowStyle}">
        <TableRow>
            <TableCell><Paragraph Foreground="Blue">Blue</Paragraph></TableCell>
            <TableCell><Paragraph>1</Paragraph></TableCell>
            <TableCell><Paragraph>2</Paragraph></TableCell>
            <TableCell><Paragraph>3</Paragraph> </TableCell>
        </TableRow>
        <TableRow>
            <TableCell><Paragraph Foreground="Red">Red</Paragraph></TableCell>
            <TableCell><Paragraph>1</Paragraph></TableCell>
            <TableCell><Paragraph>2</Paragraph></TableCell>
            <TableCell><Paragraph>3</Paragraph></TableCell>
        </TableRow>
        <TableRow>
            <TableCell><Paragraph Foreground="Green">Green</Paragraph></TableCell>
            <TableCell><Paragraph>1</Paragraph></TableCell>
            <TableCell><Paragraph>2</Paragraph></TableCell>
            <TableCell><Paragraph>3</Paragraph></TableCell>
        </TableRow>
    </TableRowGroup>

    <!-- This TableRowGroup hosts a footer row for the table. -->
    <TableRowGroup Style="{StaticResource headerFooterRowStyle}">
        <TableRow>
            <TableCell><Paragraph>Totals</Paragraph></TableCell>
            <TableCell><Paragraph>3</Paragraph></TableCell>
            <TableCell><Paragraph>6</Paragraph></TableCell>
            <TableCell>
                <Table></Table>
            </TableCell>
        </TableRow>
    </TableRowGroup>
</Table>

```

下图显示了此示例的呈现效果。

	<b>Gizmos</b>	<b>Thingamajigs</b>	<b>Doohickies</b>
<i>Blue</i>	1	2	3
<i>Red</i>	1	2	3
<i>Green</i>	1	2	3
<b>Totals</b>	<b>3</b>	<b>6</b>	<b>9</b>

## 背景呈现优先级

表元素以下列顺序呈现（按 Z 顺序从最低到最高排列）。此顺序不能更改。例如，对于这些元素，没有可用于替换此已有顺序的“Z 顺序”属性。

1. Table

2. TableColumn

3. TableRowGroup

4. TableRow

5. TableCell

请参考以下示例，该示例对表内每个元素的背景色进行定义。

XAML

```
<Table Background="Yellow">
    <Table.Columns>
        < TableColumn />
        < TableColumn Background="LightGreen" />
        < TableColumn />
    </Table.Columns>
    < TableRowGroup >
        < TableRow >
            < TableCell />< TableCell />< TableCell />
        </ TableRow >
    </ TableRowGroup >
    < TableRowGroup Background="Tan" >
        < TableRow >
            < TableCell />< TableCell />< TableCell />
        </ TableRow >
    </ TableRowGroup >
    < TableRow Background="LightBlue" >
        < TableCell />< TableCell Background="Purple" />< TableCell />
    </ TableRow >
    < TableRow >
        < TableCell />< TableCell />< TableCell />
    </ TableRow >
    < TableRowGroup >
        < TableRow >
            < TableCell />< TableCell />< TableCell />
        </ TableRow >
    </ TableRowGroup >
    < TableRowGroup >
        < TableRow >
            < TableCell />< TableCell />< TableCell />
        </ TableRow >
    </ TableRowGroup >
</Table>
```

下图显示了此示例的呈现方式（仅显示背景色）。



## 跨行或列

通过分别使用 [RowSpan](#) 或 [ColumnSpan](#) 属性，可以将表格单元格配置为跨越多行或多列。

请参考以下示例，该示例中有一个跨三列的单元格。

XAML

```
<Table>
    <Table.Columns>
        < TableColumn />
        < TableColumn />
        < TableColumn />
    </Table.Columns>

    < TableRowGroup>
        < TableRow>
            < TableCell ColumnSpan="3" Background="Cyan">
                < Paragraph>This cell spans all three columns.</ Paragraph>
            </ TableCell>
        </ TableRow>
        < TableRow>
            < TableCell Background="LightGray">< Paragraph>Cell 1</ Paragraph>
        </ TableCell>
            < TableCell Background="LightGray">< Paragraph>Cell 2</ Paragraph>
        </ TableCell>
            < TableCell Background="LightGray">< Paragraph>Cell 3</ Paragraph>
        </ TableCell>
        </ TableRow>
    </ TableRowGroup>
</ Table>
```

下图显示了此示例的呈现效果。

This cell spans all three columns.		
Cell 1	Cell 2	Cell 3

## 使用代码生成表

以下示例演示如何以编程方式创建 [Table](#) 并填充内容。表格内容分为五行（由 [RowGroups](#) 对象中包含的 [TableRow](#) 对象表示）和六列（由 [TableColumn](#) 对象表示）。各行用于不同的显示目的，其中，标题行用于显示整个表的标题，标头行用于描述表中的数据列，而页脚行则包含摘要信息。请注意，“标题”行、“标头”行和“页脚”行并非表格所固有的，它们只是具有不同特征的行。表单元格包含实际内容，可以包含文本、图像或几乎任何其他用户界面 (UI) 元素。

首先，创建一个 [FlowDocument](#) 来承载 [Table](#)，然后创建一个新的 [Table](#) 并将其添加到 [FlowDocument](#) 的内容中。

C#

```
// Create the parent FlowDocument...
flowDoc = new FlowDocument();

// Create the Table...
table1 = new Table();
// ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1);

// Set some global formatting properties for the table.
table1.CellSpacing = 10;
table1.Background = Brushes.White;
```

接下来，创建六个 [TableColumn](#) 对象并将其添加到表的 [Columns](#) 集合中，然后应用一些格式。

### ① 备注

请注意，表格的 [Columns](#) 集合使用标准的从零开始的索引。

C#

```
// Create 6 columns and add them to the table's Columns collection.
int numberOfColumns = 6;
for (int x = 0; x < numberOfColumns; x++)
{
    table1.Columns.Add(new TableColumn());

    // Set alternating background colors for the middle columns.
    if(x%2 == 0)
        table1.Columns[x].Background = Brushes.Beige;
    else
        table1.Columns[x].Background = Brushes.LightSteelBlue;
}
```

接下来，创建一个标题行，并将其添加到表中，同时应用某些格式设置。标题行包含一个单元格，该单元格跨表中的全部六列。

C#

```
// Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup());
```

```

// Add the first (title) row.
table1.RowGroups[0].Rows.Add(new TableRow());

// Alias the current working row for easy reference.
TableRow currentRow = table1.RowGroups[0].Rows[0];

// Global formatting for the title row.
currentRow.Background = Brushes.Silver;
currentRow.FontSize = 40;
currentRow.FontWeight = System.Windows.FontWeights.Bold;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales
Project"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;

```

接下来，创建一个标头行并将其添加到表中，同时创建标头行中的单元格并填充其内容。

C#

```

// Add the second (header) row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[1];

// Global formatting for the header row.
currentRow.FontSize = 18;
currentRow.FontWeight = FontWeights.Bold;

// Add cells with content to the second row.
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))));

```

接下来，创建一个数据行并将其添加到表中，同时创建此行中的单元格并填充其内容。生成此行的过程与生成标头行的过程类似，只是应用的格式设置略有不同。

C#

```

// Add the third row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[2];

// Global formatting for the row.
currentRow.FontSize = 12;
currentRow.FontWeight = FontWeights.Normal;

// Add cells with content to the third row.

```

```
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))));
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))));

// Bold the first cell.
currentRow.Cells[0].FontWeight = FontWeights.Bold;
```

最后，创建、添加脚注行并设置其格式。与标题行类似，脚注包含的单元格的跨度为表中的全部六列。

C#

```
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[3];

// Global formatting for the footer row.
currentRow.Background = Brushes.LightGray;
currentRow.FontSize = 18;
currentRow.FontWeight = System.Windows.FontWeights.Normal;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004
Revenue: $810,000"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;
```

## 另请参阅

- [流文档概述](#)
- [使用 XAML 定义表](#)
- [WPF 中的文档](#)
- [使用流内容元素](#)

# 流内容元素帮助主题

项目 • 2023/02/06

本节中的主题介绍如何使用各种流内容元素和相关功能完成常见任务。

## 本节内容

[调整段落间的间距](#)

[以编程方式生成表](#)

[以编程方式更改内容的 FlowDirection](#)

[以编程方式更改 TextWrapping 属性](#)

[使用 XAML 定义表](#)

[修改文本的版式](#)

[启用文本修整](#)

[以编程方式将元素插入文本](#)

[通过 Blocks 属性操作流内容元素](#)

[通过 Inlines 属性操作流内容元素](#)

[通过 Blocks 属性操作 FlowDocument](#)

[通过 Columns 属性操作表的列](#)

[通过 RowGroups 属性操作表的行组](#)

[使用流内容元素](#)

[使用 FlowDocument 列分隔特性](#)

## 参考

[FlowDocument](#)

[Block](#)

[Inline](#)

## 相关章节

[WPF 中的文档](#)

# 如何：调整段落间的间距

项目 • 2023/02/06

此示例演示如何在流内容中调整或消除段落之间的间距。

在流内容中，段落之间出现的额外空间是在这些段落上设置边距的结果；因此，段落之间的间距可以通过调整这些段落的边距来控制。若要消除两个段落之间的额外间距，请将段落的边距设置为“0”。若要在整个 [FlowDocument](#) 中实现段落之间的统一间距，请使用样式设置 [FlowDocument](#) 中所有段落的统一边距值。

请务必注意，相邻两段的边距会“折叠”到较大的边距，而不是叠加。因此，如果两个相邻段落的边距分别为 20 像素和 40 像素，则最终段落之间的间距将为 40 像素，即两个边距值中较大的一个。

## 示例

下面的示例使用样式将 [FlowDocument](#) 中的所有 [Paragraph](#) 元素的边距设置为“0”，这有效地消除了 [FlowDocument](#) 中段落之间的额外间距。

XAML

```
<FlowDocument>
    <FlowDocument.Resources>
        <!-- This style is used to set the margins for all paragraphs in the
FlowDocument to 0. --&gt;
        &lt;Style TargetType="{x:Type Paragraph}"&gt;
            &lt;Setter Property="Margin" Value="0"/&gt;
        &lt;/Style&gt;
    &lt;/FlowDocument.Resources&gt;

    &lt;Paragraph&gt;
        Spacing between paragraphs is caused by margins set on the paragraphs.
        Two adjacent margins
        will "collapse" to the larger of the two margin widths, rather than
        doubling up.
    &lt;/Paragraph&gt;

    &lt;Paragraph&gt;
        To eliminate extra spacing between two paragraphs, just set the
        paragraph margins to 0.
    &lt;/Paragraph&gt;
&lt;/FlowDocument&gt;</pre>
```

# 如何：以编程方式生成表

项目 • 2023/02/06

以下示例演示如何以编程方式创建 [Table](#) 并填充内容。表格内容分为五行（由 [RowGroups](#) 对象中包含的 [TableRow](#) 对象表示）和六列（由 [TableColumn](#) 对象表示）。各行用于不同的显示目的，其中，标题行用于显示整个表的标题，标头行用于描述表中的数据列，而页脚行则包含摘要信息。请注意，“标题”行、“标头”行和“页脚”行并非表格所固有的，它们只是具有不同特征的行。表单元格包含实际内容，可以包含文本、图像或几乎任何其他用户界面 (UI) 元素。

## 创建表

首先，创建一个 [FlowDocument](#) 来承载 [Table](#)，然后创建一个新的 [Table](#) 并将其添加到 [FlowDocument](#) 的内容中。

C#

```
// Create the parent FlowDocument...
flowDoc = new FlowDocument();

// Create the Table...
table1 = new Table();
// ...and add it to the FlowDocument Blocks collection.
flowDoc.Blocks.Add(table1);

// Set some global formatting properties for the table.
table1.CellSpacing = 10;
table1.Background = Brushes.White;
```

## 添加列

接下来，创建六个 [TableColumn](#) 对象并将其添加到表的 [Columns](#) 集合中，然后应用一些格式。

### ① 备注

请注意，表格的 [Columns](#) 集合使用标准的从零开始的索引。

C#

```
// Create 6 columns and add them to the table's Columns collection.
int numberOfColumns = 6;
```

```
for (int x = 0; x < numberOfRowsColumns; x++)
{
    table1.Columns.Add(new TableColumn());

    // Set alternating background colors for the middle columns.
    if(x%2 == 0)
        table1.Columns[x].Background = Brushes.Beige;
    else
        table1.Columns[x].Background = Brushes.LightSteelBlue;
}
```

## 添加标题行

接下来，创建一个标题行，并将其添加到表中，同时应用某些格式设置。标题行包含一个单元格，该单元格跨表中的全部六列。

C#

```
// Create and add an empty TableRowGroup to hold the table's Rows.
table1.RowGroups.Add(new TableRowGroup());

// Add the first (title) row.
table1.RowGroups[0].Rows.Add(new TableRow());

// Alias the current working row for easy reference.
TableRow currentRow = table1.RowGroups[0].Rows[0];

// Global formatting for the title row.
currentRow.Background = Brushes.Silver;
currentRow.FontSize = 40;
currentRow.FontWeight = System.Windows.FontWeights.Bold;

// Add the header row with content,
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("2004 Sales
Project"))));
// and set the row to span all 6 columns.
currentRow.Cells[0].ColumnSpan = 6;
```

## 添加标头行

接下来，创建一个标头行并将其添加到表中，同时创建标头行中的单元格并填充其内容。

C#

```
// Add the second (header) row.
table1.RowGroups[0].Rows.Add(new TableRow());
currentRow = table1.RowGroups[0].Rows[1];
```

```
// Global formatting for the header row.  
currentRow.FontSize = 18;  
currentRow.FontWeight = FontWeights.Bold;  
  
// Add cells with content to the second row.  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Product"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 1"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 2"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 3"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Quarter 4"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("TOTAL"))));
```

## 添加行

接下来，创建一个数据行并将其添加到表中，同时创建此行中的单元格并填充其内容。生成此行的过程与生成标头行的过程类似，只是应用的格式设置略有不同。

C#

```
// Add the third row.  
table1.RowGroups[0].Rows.Add(new TableRow());  
currentRow = table1.RowGroups[0].Rows[2];  
  
// Global formatting for the row.  
currentRow.FontSize = 12;  
currentRow.FontWeight = FontWeights.Normal;  
  
// Add cells with content to the third row.  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Widgets"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$50,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$55,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$60,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$65,000"))));  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("$230,000"))));  
  
// Bold the first cell.  
currentRow.Cells[0].FontWeight = FontWeights.Bold;
```

## 添加页脚行

最后，创建、添加脚注行并设置其格式。与标题行类似，脚注包含的单元格的跨度为表中的全部六列。

C#

```
table1.RowGroups[0].Rows.Add(new TableRow());  
currentRow = table1.RowGroups[0].Rows[3];
```

```
// Global formatting for the footer row.  
currentRow.Background = Brushes.LightGray;  
currentRow.FontSize = 18;  
currentRow.FontWeight = System.Windows.FontWeights.Normal;  
  
// Add the header row with content,  
currentRow.Cells.Add(new TableCell(new Paragraph(new Run("Projected 2004  
Revenue: $810,000"))));  
// and set the row to span all 6 columns.  
currentRow.Cells[0].ColumnSpan = 6;
```

## 另请参阅

- [表概述](#)

# 如何：以编程方式更改内容的 FlowDirection

项目 • 2023/02/06

此示例演示如何以编程方式更改 [FlowDocumentReader](#) 的 [FlowDirection](#) 属性。

## 创建按钮元素

创建两个 [Button](#) 元素，每个元素都代表 [FlowDirection](#) 的一个可能值。单击某个按钮时，关联的属性值会应用于名为 `tf1` 的 [FlowDocumentReader](#) 的内容。该属性值还会写入一个名为 `txt1` 的 [TextBlock](#)。

XAML

```
<StackPanel DockPanel.Dock="Top" Orientation="Horizontal" Margin="0,0,0,10">
    <Button Click="LR">LeftToRight</Button>
    <Button Click="RL">RightToLeft</Button>
</StackPanel>

<TextBlock Name="txt1" DockPanel.Dock="Bottom" Margin="0,50,0,0"/>

<FlowDocumentReader>
    <FlowDocument FontFamily="Arial" Name="tf1">
        <Paragraph>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit,
            sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam,
            quis nostrud exerci tation ullamcorper suscipit lobortis nisl
            ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
        </Paragraph>
        <Paragraph>
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
            aliquam erat volutpat. Ut wisi enim ad minim veniam, quis
            nostrud exerci tation ullamcorper suscipit lobortis nisl ut
            aliquip ex ea commodo consequat. Duis autem vel eum iriure.
            Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed
            diam nonummy nibh euismod tincidunt ut laoreet dolore magna
```

```
aliquam erat volutpat. Ut wisi enim ad minim veniam, quis  
nostrud exerci tation ullamcorper suscipit lobortis nisl ut  
aliquip ex ea commodo consequat. Duis autem vel eum iriure.  
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed  
diam nonummy nibh euismod tincidunt ut laoreet dolore magna  
aliquam erat volutpat. Ut wisi enim ad minim veniam, quis  
nostrud exerci tation ullamcorper suscipit lobortis nisl ut  
aliquip ex ea commodo consequat. Duis autem vel eum iriure.  
</Paragraph>  
</FlowDocument>  
</FlowDocumentReader>
```

## C# 代码示例

与上述按钮单击关联的事件在 C# 代码隐藏文件中进行处理。

C#

```
private void LR(object sender, RoutedEventArgs e)  
{  
    tf1.FlowDirection = FlowDirection.LeftToRight;  
    txt1.Text = "FlowDirection is now " + tf1.FlowDirection;  
}  
private void RL(object sender, RoutedEventArgs e)  
{  
    tf1.FlowDirection = FlowDirection.RightToLeft;  
    txt1.Text = "FlowDirection is now " + tf1.FlowDirection;  
}
```

# 如何：以编程方式更改 TextWrapping 属性

项目 • 2023/02/06

## 示例

下面的代码示例演示如何以编程方式更改 `TextWrapping` 属性的值。

三个 `Button` 元素放置在 XAML 的 `StackPanel` 元素内。 `Button` 的每个 `Click` 事件对应于代码中的一个事件处理程序。 单击按钮时，事件处理程序使用的名称与其将应用于 `txt2` 的 `TextWrapping` 值相同。 此外，还会更新 `txt1` ( XAML 中未显示的 `TextBlock` ) 中的文本，以反映属性中的变化。

XAML

```
<StackPanel Orientation="Horizontal" Margin="0,0,0,20">
    <Button Name="btn1" Background="Silver" Width="100"
    Click="Wrap">Wrap</Button>
    <Button Name="btn2" Background="Silver" Width="100"
    Click="NoWrap">NoWrap</Button>
    <Button Name="btn4" Background="Silver" Width="100"
    Click="WrapWithOverflow">WrapWithOverflow</Button>
</StackPanel>

<TextBlock Name="txt2" TextWrapping="Wrap" Margin="0,0,0,20"
Foreground="Black">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit. Lorem ipsum
    dolor sit amet,
    consectetur adipiscing elit. Lorem ipsum dolor sit amet, consectetur
    adipiscing elit.
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
</TextBlock>
```

C#

```
private void Wrap(object sender, RoutedEventArgs e)
{
    txt2.TextWrapping = System.Windows.TextWrapping.Wrap;
    txt1.Text = "The TextWrap property is currently set to Wrap.";
}

private void NoWrap(object sender, RoutedEventArgs e)
{
    txt2.TextWrapping = System.Windows.TextWrapping.NoWrap;
    txt1.Text = "The TextWrap property is currently set to NoWrap.";
}

private void WrapWithOverflow(object sender, RoutedEventArgs e)
```

```
{  
    txt2.TextWrapping = System.Windows.TextWrapping.WrapWithOverflow;  
    txt1.Text = "The TextWrap property is currently set to  
WrapWithOverflow.";  
}
```

## 另请参阅

- [TextWrapping](#)
- [TextWrapping](#)

# 如何：使用 XAML 定义表

项目 • 2023/02/06

以下示例演示如何使用 Extensible Application Markup Language (XAML) 定义 Table。此示例表有 4 列（由 TableColumn 元素表示）和若干行（由 TableRow 元素表示），包含数据以及标题、表头和页脚信息。行必须包含在 TableRowGroup 元素中。表中的每一行都由一个或多个单元格组成（由 TableCell 元素表示）。表单元格中的内容必须包含在 Block 元素中；在这种情况下，应使用 Paragraph 元素。该表还在页脚行中包含一个超链接（由 Hyperlink 元素表示）。

## 示例

XAML

```
<FlowDocumentReader>
<FlowDocument>

    <Table CellSpacing="5">

        <Table.Columns>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
            <TableColumn/>
        </Table.Columns>

        <TableRowGroup>

            <!-- Title row for the table. -->
            <TableRow Background="SkyBlue">
                <TableCell ColumnSpan="4" TextAlignment="Center">
                    <Paragraph FontSize="24pt" FontWeight="Bold">Planetary
Information</Paragraph>
                </TableCell>
            </TableRow>

            <!-- Header row for the table. -->
            <TableRow Background="LightGoldenrodYellow">
                <TableCell><Paragraph FontSize="14pt"
FontWeight="Bold">Planet</Paragraph></TableCell>
                <TableCell><Paragraph FontSize="14pt" FontWeight="Bold">Mean
Distance from Sun</Paragraph></TableCell>
                <TableCell><Paragraph FontSize="14pt" FontWeight="Bold">Mean
Diameter</Paragraph></TableCell>
                <TableCell><Paragraph FontSize="14pt"
FontWeight="Bold">Approximate Mass</Paragraph></TableCell>
            </TableRow>
        </TableRowGroup>
    </Table>
</FlowDocument>
</FlowDocumentReader>
```

```

<!-- Sub-title row for the inner planets. -->
<TableRow>
    <TableCell ColumnSpan="4"><Paragraph FontSize="14pt"
FontWeight="Bold">The Inner Planets</Paragraph></TableCell>
</TableRow>

<!-- Four data rows for the inner planets. -->
<TableRow>
    <TableCell><Paragraph>Mercury</Paragraph></TableCell>
    <TableCell><Paragraph>57,910,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>4,880 km</Paragraph></TableCell>
    <TableCell><Paragraph>3.30e23 kg</Paragraph></TableCell>
</TableRow>
<TableRow Background="lightgray">
    <TableCell><Paragraph>Venus</Paragraph></TableCell>
    <TableCell><Paragraph>108,200,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>12,103.6 km</Paragraph></TableCell>
    <TableCell><Paragraph>4.869e24 kg</Paragraph></TableCell>
</TableRow>
<TableRow>
    <TableCell><Paragraph>Earth</Paragraph></TableCell>
    <TableCell><Paragraph>149,600,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>12,756.3 km</Paragraph></TableCell>
    <TableCell><Paragraph>5.972e24 kg</Paragraph></TableCell>
</TableRow>
<TableRow Background="lightgray">
    <TableCell><Paragraph>Mars</Paragraph></TableCell>
    <TableCell><Paragraph>227,940,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>6,794 km</Paragraph></TableCell>
    <TableCell><Paragraph>6.4219e23 kg</Paragraph></TableCell>
</TableRow>

<!-- Sub-title row for the outer planets. -->
<TableRow>
    <TableCell ColumnSpan="4"><Paragraph FontSize="14pt"
FontWeight="Bold">The Major Outer Planets</Paragraph></TableCell>
</TableRow>

<!-- Four data rows for the major outer planets. -->
<TableRow>
    <TableCell><Paragraph>Jupiter</Paragraph></TableCell>
    <TableCell><Paragraph>778,330,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>142,984 km</Paragraph></TableCell>
    <TableCell><Paragraph>1.900e27 kg</Paragraph></TableCell>
</TableRow>
<TableRow Background="lightgray">
    <TableCell><Paragraph>Saturn</Paragraph></TableCell>
    <TableCell><Paragraph>1,429,400,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>120,536 km</Paragraph></TableCell>
    <TableCell><Paragraph>5.68e26 kg</Paragraph></TableCell>
</TableRow>
<TableRow>
    <TableCell><Paragraph>Uranus</Paragraph></TableCell>
    <TableCell><Paragraph>2,870,990,000 km</Paragraph></TableCell>
    <TableCell><Paragraph>51,118 km</Paragraph></TableCell>

```

```

        <TableCell><Paragraph>8.683e25 kg</Paragraph></TableCell>
    </TableRow>
    <TableRow Background="lightgray">
        <TableCell><Paragraph>Neptune</Paragraph></TableCell>
        <TableCell><Paragraph>4,504,000,000 km</Paragraph></TableCell>
        <TableCell><Paragraph>49,532 km</Paragraph></TableCell>
        <TableCell><Paragraph>1.0247e26 kg</Paragraph></TableCell>
    </TableRow>

    <!-- Footer row for the table. -->
    <TableRow>
        <TableCell ColumnSpan="4"><Paragraph FontSize="10pt" FontStyle="Italic">
            Information from the
            <Hyperlink
                NavigateUri="http://encarta.msn.com/encnet/refpages/artcenter.aspx">Encarta</Hyperlink>
            web site.
        </Paragraph></TableCell>
    </TableRow>

    </TableRowGroup>
</Table>
</FlowDocument>
</FlowDocumentReader>

```

下图显示了此示例中定义的表如何呈现：

Planetary Information			
Planet	Mean Distance from Sun	Mean Diameter	Approximate Mass
<b>The Inner Planets</b>			
Mercury	57,910,000 km	4,880 km	3.30e23 kg
Venus	108,200,000 km	12,103.6 km	4.869e24 kg
Earth	149,600,000 km	12,756.3 km	5.972e24 kg
Mars	227,940,000 km	6,794 km	6.4219e23 kg
<b>The Major Outer Planets</b>			
Jupiter	778,330,000 km	142,984 km	1.900e27 kg
Saturn	1,429,400,000 km	120,536 km	5.68e26 kg
Uranus	2,870,990,000 km	51,118 km	8.683e25 kg

# 如何：修改文本的版式

项目 • 2022/09/27

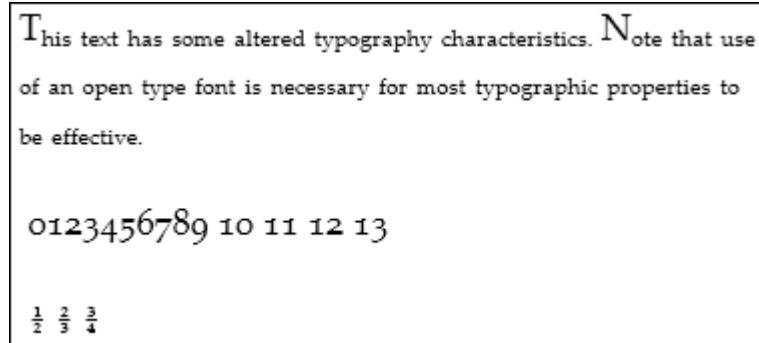
以下示例将 [Paragraph](#) 作为示例元素，演示如何设置 [Typography](#) 属性。

## 示例 1：演示如何在屏幕上呈现文本的更改版式属性和默认版式属性

XAML

```
<Paragraph
    TextAlignment="Left"
    FontSize="18"
    FontFamily="Palatino Linotype"
    Typography.NumeralStyle="OldStyle"
    Typography.Fraction="Stacked"
    Typography.Variants="Inferior"
>
<Run>
    This text has some altered typography characteristics. Note
    that use of an open type font is necessary for most typographic
    properties to be effective.
</Run>
<LineBreak/><LineBreak/>
<Run>
    0123456789 10 11 12 13
</Run>
<LineBreak/><LineBreak/>
<Run>
    1/2 2/3 3/4
</Run>
</Paragraph>
```

下图显示了此示例的呈现效果。



与此相反，下图显示了一个具有默认版式属性的类似示例的呈现效果。

This text has some altered typography characteristics. Note that use of an open type font is necessary for most typographic properties to be effective.

0123456789 10 11 12 13

1/2 2/3 3/4

## 示例 2：演示如何以编程方式设置文本的版式属性

下面的示例演示如何以编程方式设置 **Typography** 属性。

C#

```
Paragraph par = new Paragraph();

Run runText = new Run(
    "This text has some altered typography characteristics. Note" +
    "that use of an open type font is necessary for most typographic" +
    "properties to be effective.");
Run runNumerals = new Run("0123456789 10 11 12 13");
Run runFractions = new Run("1/2 2/3 3/4");

par.Inlines.Add(runText);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runNumerals);
par.Inlines.Add(new LineBreak());
par.Inlines.Add(new LineBreak());
par.Inlines.Add(runFractions);

par.TextAlignment = TextAlignment.Left;
par.FontSize = 18;
par.FontFamily = new FontFamily("Palatino Linotype");

par.Typography.NumeralStyle = FontNumeralStyle.OldStyle;
par.Typography.Fraction = FontFraction.Stacked;
par.Typography.Variants = FontVariants.Inferior;
```

## 另请参阅

- [流文档概述](#)

# 如何：启用文本修整

项目 • 2022/09/27

此示例演示了 [TextTrimming 枚举](#) 中可用值的用法和效果。

## 示例

下面的示例利用 [TextTrimming 特性集](#) 定义 [TextBlock](#) 元素。

XAML

```
<TextBlock
    Name="myTextBlock"
    Margin="20" Background="LightGoldenrodYellow"
    TextTrimming="WordEllipsis" TextWrapping="NoWrap"
    FontSize="14"
>
    One<LineBreak/>
    two two<LineBreak/>
    Three Three Three<LineBreak/>
    four four four four<LineBreak/>
    Five Five Five Five Five<LineBreak/>
    six six six six six six<LineBreak/>
    Seven Seven Seven Seven Seven Seven Seven
</TextBlock>
```

下面的内容演示如何在代码中设置相应的 [TextTrimming 属性](#)。

C#

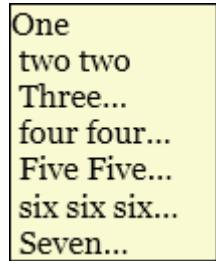
```
myTextBlock.TextTrimming = TextTrimming.CharacterEllipsis;
```

目前有三个可用于剪裁文本的选项：[CharacterEllipsis](#)、[WordEllipsis](#) 和 [None](#)。

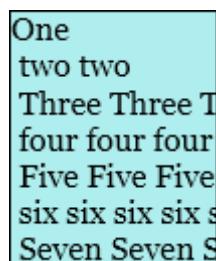
当 [TextTrimming](#) 设置为 [CharacterEllipsis](#) 时，将对文本进行剪裁，并在最靠近剪裁边缘的字符处使用省略号填充。此设置旨在修整文本以使其更适应修整边界，但可能会导致某些单词仅部分修整。下图显示了在类似于上述定义的 [TextBlock](#) 中此设置的效果。

```
One
two two
Three Thre...
four four fo...
Five Five Fi...
six six six si...
Seven Seve...
```

当 [TextTrimming](#) 设置为 `WordEllipsis` 时，将对文本进行剪裁，并在最靠近剪裁边缘的第一个完整的单词结尾处使用省略号填充。此设置不会导致单词部分剪裁，但是剪裁文本时不如 `CharacterEllipsis` 设置那样靠近剪裁边缘。下图显示了在上述定义的 [TextBlock](#) 中此设置的效果。



当 [TextTrimming](#) 设置为 `None` 时，不会执行文本剪裁。在这种情况下，只会将文本裁切到父文本容器的边界。下图显示了在类似于上述定义的 [TextBlock](#) 中此设置的效果。



# 如何：以编程方式将元素插入文本

项目 • 2023/02/06

下面示例显示如何使用两个 `TextPointer` 对象来指定文本中要应用 `Span` 元素的范围。

## 示例

C#

```
using System;
using System.Windows;
using System.Windows.Media;
using System.Windows.Controls;
using System.Windows.Documents;

namespace SDKSample
{
    public partial class InsertInlineIntoTextExample : Page
    {
        public InsertInlineIntoTextExample()
        {

            // Create a paragraph with a short sentence
            Paragraph myParagraph = new Paragraph(new Run("Neptune has 72
times Earth's volume..."));

            // Create two TextPointers that will specify the text range the
            Span will cover
            TextPointer myTextPointer1 =
myParagraph.ContentStart.GetPositionAtOffset(10);
            TextPointer myTextPointer2 =
myParagraph.ContentEnd.GetPositionAtOffset(-5);

            // Create a Span that covers the range between the two
            TextPointers.
            Span mySpan = new Span(myTextPointer1, myTextPointer2);
            mySpan.Background = Brushes.Red;

            // Create a FlowDocument with the paragraph as its initial
            content.
            FlowDocument myFlowDocument = new FlowDocument(myParagraph);

            this.Content = myFlowDocument;
        }
    }
}
```

下图显示此示例。

**Neptune has 72 times Earth's volume...**

## 另请参阅

- [流文档概述](#)

# 如何：通过 Blocks 属性操作流内容元素

项目 • 2023/02/06

这些示例演示了可通过“Blocks”属性对流内容元素执行的一些更常见的操作。此属性用于添加和删除来自 [BlockCollection](#) 的项。具有“Blocks”属性的流内容元素包括：

- [Figure](#)
- [Floater](#)
- [ListItem](#)
- [Section](#)
- [TableCell](#)

这些示例恰好使用 [Section](#) 作为流内容元素，但是这些技术适用于托管流内容元素集合的所有元素。

## 创建一个新的“部分”

下面的示例创建一个新的 [Section](#)，然后使用“添加”方法向“部分”内容添加一个新的段落。

C#

```
Section secx = new Section();
secx.Blocks.Add(new Paragraph(new Run("A bit of text content...")));
```

## 创建新的 Paragraph 元素

下面的示例创建一个新的 [Paragraph](#) 元素，并将其插入到 [Section](#) 的开头。

C#

```
Paragraph parx = new Paragraph(new Run("Text to insert..."));
secx.Blocks.InsertBefore(secx.Blocks.FirstBlock, parx);
```

## 获取“部分”中的顶级 Block 元素

下面的示例获取 [Section](#) 中包含的顶级 [Block](#) 元素数。

C#

```
int countTopLevelBlocks = secx.Blocks.Count;
```

## 删除“部分”中的最后一个 Block 元素

下面的示例删除 [Section](#) 中的最后一个 [Block](#) 元素。

C#

```
secx.Blocks.Remove(secx.Blocks.LastBlock);
```

## 清除“部分”中的所有 Block 元素内容

下面的示例清除 [Section](#) 中的所有内容（[Block](#) 元素）。

C#

```
secx.Blocks.Clear();
```

## 另请参阅

- [BlockCollection](#)
- [InlineCollection](#)
- [ListItemCollection](#)
- [流文档概述](#)
- [通过 RowGroups 属性操作表的行组](#)
- [通过 Columns 属性操作表的列](#)
- [通过 RowGroups 属性操作表的行组](#)

# 如何：通过 Inlines 属性操作流内容元素

项目 • 2023/02/06

这些示例演示可以针对内联流内容元素（和此类元素的容器，例如 [TextBlock](#)）通过 [Inlines](#) 属性执行的一些较常见操作。此属性用于添加和删除来自 [InlineCollection](#) 的项。具有“[Inlines](#)”属性的流内容元素包括：

- [Bold](#)
- [Hyperlink](#)
- [Italic](#)
- [Paragraph](#)
- [Span](#)
- [Underline](#)

这些示例恰好使用 [Span](#) 作为流内容元素，但是这些技术适用于承载 [InlineCollection](#) 集合的所有元素或控件。

## 创建新的 Span 对象

以下示例创建新的 [Span](#) 对象，然后使用 [Add](#) 方法将两个文本运行添加为 [Span](#) 的内容子级。

C#

```
Span spanx = new Span();
spanx.Inlines.Add(new Run("A bit of text content..."));
spanx.Inlines.Add(new Run("A bit more text content..."));
```

## 创建新的 Run 元素

下面的示例创建一个新的 [Run](#) 元素，并将其插入到 [Span](#) 的开头。

C#

```
Run runx = new Run("Text to insert...");
spanx.Inlines.InsertBefore(spanx.Inlines.FirstInline, runx);
```

## 获取 Span 中的顶级 Inline 元素

下面的示例获取 [Span](#) 中包含的顶级 [Inline](#) 元素数。

C#

```
int countTopLevelInlines = spanx.Inlines.Count;
```

## 获取 Span 中的最后一个 Inline 元素

下面的示例删除 [Span](#) 中的最后一个 [Inline](#) 元素。

C#

```
spanx.Inlines.Remove(spanx.Inlines.LastInline);
```

## 清除 Span 中的所有 Inline 元素内容

下面的示例清除 [Span](#) 中的所有内容 ([Inline](#) 元素)。

C#

```
spanx.Inlines.Clear();
```

## 另请参阅

- [BlockCollection](#)
- [InlineCollection](#)
- [ListItemCollection](#)
- [流文档概述](#)
- [通过 Blocks 属性操作 FlowDocument](#)
- [通过 Columns 属性操作表的列](#)
- [通过 RowGroups 属性操作表的行组](#)

# 如何：通过 Blocks 属性操作 FlowDocument

项目 • 2023/02/06

这些示例演示可以通过 [Blocks 属性](#)对 [FlowDocument](#) 执行的一些更常见的操作。

## 创建新的 FlowDocument

以下示例创建一个新的 [FlowDocument](#)，然后将一个新的 [Paragraph](#) 元素追加到 [FlowDocument](#)。

C#

```
FlowDocument flowDoc = new FlowDocument(new Paragraph(new Run("A bit of text content...")));
flowDoc.Blocks.Add(new Paragraph(new Run("Text to append...")));
```

## 创建新的 Paragraph 元素

下面的示例创建一个新的 [Paragraph](#) 元素，并将其插入到 [FlowDocument](#) 的开头。

C#

```
Paragraph p = new Paragraph(new Run("Text to insert..."));
flowDoc.Blocks.InsertBefore(flowDoc.Blocks.FirstBlock, p);
```

## 获取顶级 Block 元素

下面的示例获取 [FlowDocument](#) 中包含的顶级 [Block](#) 元素数。

C#

```
int countTopLevelBlocks = flowDoc.Blocks.Count;
```

## 删除最后一个 Block 元素

下面的示例删除 [FlowDocument](#) 中的最后一个 [Block](#) 元素。

C#

```
flowDoc.Blocks.Remove(flowDoc.Blocks.LastBlock);
```

## 清除所有 Block 内容

下面的示例清除 [FlowDocument](#) 中的所有内容（[Block](#) 元素）。

C#

```
flowDoc.Blocks.Clear();
```

## 另请参阅

- [通过 RowGroups 属性操作表的行组](#)
- [通过 Columns 属性操作表的列](#)
- [通过 RowGroups 属性操作表的行组](#)

# 如何：通过 Columns 属性操作表列

项目 • 2023/02/06

此示例演示了可通过 [Columns](#) 属性对表列执行的一些更常见的操作。

## 创建新表

下面的示例将创建一个新表，然后使用 [Add](#) 方法将列添加到表的 [Columns](#) 集合中。

C#

```
Table tbl = new Table();
int columnsToAdd = 4;
for (int x = 0; x < columnsToAdd; x++)
    tbl.Columns.Add(new TableColumn());
```

## 插入新的 TableColumn

下面的示例将插入一个新的 [TableColumn](#)。 新列在索引位置 0 处插入，使其成为表中的第一列。

### ① 备注

[TableColumnCollection](#) 集合使用从零开始的标准索引。

C#

```
tbl.Columns.Insert(0, new TableColumn());
```

## 访问 TableColumnCollection 中的属性

下面的示例访问 [TableColumnCollection](#) 集合中列的一些任意属性，该示例通过索引引用特定列。

C#

```
tbl.Columns[0].Width = new GridLength(20);
tbl.Columns[1].Background = Brushes.AliceBlue;
tbl.Columns[2].Width = new GridLength(20);
tbl.Columns[3].Background = Brushes.AliceBlue;
```

# 获取表中的列数

以下示例获取表当前托管的列数。

C#

```
int columns = tbl.Columns.Count;
```

## 按引用删除列

以下示例按引用删除特定列。

C#

```
tbl.Columns.Remove(tbl.Columns[3]);
```

## 按索引删除列

以下示例按索引删除特定列。

C#

```
tbl.Columns.RemoveAt(2);
```

## 删除所有列

下面的示例从表的列集合中删除所有列。

C#

```
tbl.Columns.Clear();
```

## 另请参阅

- [表概述](#)
- [使用 XAML 定义表](#)
- [以编程方式生成表](#)
- [通过 RowGroups 属性操作表的行组](#)
- [通过 Blocks 属性操作 FlowDocument](#)

- 通过 RowGroups 属性操作表的行组

# 如何：通过 RowGroups 属性操作表的行组

项目 • 2023/02/06

此示例演示了可通过 [RowGroups](#) 属性对表行组执行的一些更常见的操作。

## 使用 Add 方法创建一个新表

下面的示例将创建一个新表，然后使用 [Add](#) 方法将列添加到表的 [RowGroups](#) 集合中。

C#

```
Table tbl = new Table();
int rowGroupsToAdd = 4;
for (int x = 0; x < rowGroupsToAdd; x++)
    tbl.RowGroups.Add(new TableRowGroup());
```

## 插入新 TableRowGroup

下面的示例将插入一个新的 [TableRowGroup](#)。 新列在索引位置 0 处插入，使其成为表中的第一行组。

### ① 备注

[TableRowGroupCollection](#) 集合使用从零开始的标准索引。

C#

```
tbl.RowGroups.Insert(0, new TableRowGroup());
```

## 向 TableRowGroup 添加行

以下示例将几行添加到表中的特定 [TableRowGroup](#)（由索引指定）。

C#

```
int rowsToAdd = 10;
for (int x = 0; x < rowsToAdd; x++)
    tbl.RowGroups[0].Rows.Add(new TableRow());
```

# 访问第一行组中的行属性

下面的示例访问表中第一个行组中行的一些任意属性。

C#

```
// Alias the working TableRowGroup for ease in referencing.  
TableRowGroup trg = tbl.RowGroups[0];  
trg.Rows[0].Background = Brushes.CornflowerBlue;  
trg.Rows[1].FontSize = 24;  
trg.Rows[2].ToolTip = "This row's tooltip";
```

# 向 TableRow 添加单元格

以下示例将几个单元格添加到表中的特定 [TableRow](#) (由索引指定)。

C#

```
int cellsToAdd = 10;  
for (int x = 0; x < cellsToAdd; x++)  
    tbl.RowGroups[0].Rows[0].Cells.Add(new TableCell(new Paragraph(new  
Run("Cell " + (x + 1)))));
```

# 访问第一行组中单元格的方法和属性

以下示例访问第一行组中第一行中单元格的一些任意方法和属性。

C#

```
// Alias the working for for ease in referencing.  
TableRow row = tbl.RowGroups[0].Rows[0];  
row.Cells[0].Background = Brushes.PapayaWhip;  
row.Cells[1].FontStyle = FontStyles.Italic;  
// This call clears all of the content from this cell.  
row.Cells[2].Blocks.Clear();
```

# 获取表中的 TableRowGroup 元素数

以下示例返回表托管的 [TableRowGroup](#) 元素的数量。

C#

```
int rowGroups = tbl.RowGroups.Count;
```

## 按引用删除行组

以下示例按引用删除特定行组。

C#

```
tbl.RowGroups.Remove(tbl.RowGroups[0]);
```

## 按索引删除行组

以下示例按索引删除特定行组。

C#

```
tbl.RowGroups.RemoveAt(0);
```

## 从表的行组集合中删除所有行组

以下示例从表的行组集合中删除所有行组。

C#

```
tbl.RowGroups.Clear();
```

## 另请参阅

- [如何：通过 Inlines 属性操作流内容元素](#)
- [通过 Blocks 属性操作 FlowDocument](#)
- [通过 Columns 属性操作表的列](#)

# 如何：使用流内容元素

项目 · 2023/02/06

下面的示例演示各种流内容元素及其相关联特性的声明性用法。 演示的元素和特性包括：

- [Bold](#) 元素
- [BreakPageBefore](#) 特性
- [FontSize](#) 特性
- [Italic](#) 元素
- [LineBreak](#) 元素
- [List](#) 元素
- [ListItem](#) 元素
- [Paragraph](#) 元素
- [Run](#) 元素
- [Section](#) 元素
- [Span](#) 元素
- [Variants](#) 特性（上标和下标）
- [Underline](#) 元素

## 示例

XAML

```
<FlowDocument
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
    <Paragraph FontSize="18">Flow Format Example</Paragraph>
    <Paragraph>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam
        nonummy nibh euismod tincidunt ut labore dolore magna aliquam erat volutpat.
        Ut wisi
```

```
    enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis
    nisl ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
</Paragraph>
<Paragraph>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed diam
nonummy nibh
    euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut
wisi enim
    ad minim veniam, quis nostrud exerci tation ullamcorper suscipit
lobortis nisl
    ut aliquip ex ea commodo consequat. Duis autem vel eum iriure.
</Paragraph>

<Paragraph FontSize="18">More flow elements</Paragraph>
<Paragraph FontSize="15">Inline, font type and weight, and a
List</Paragraph>

<List>
    <ListItem><Paragraph>ListItem 1</Paragraph></ListItem>
    <ListItem><Paragraph>ListItem 2</Paragraph></ListItem>
    <ListItem><Paragraph>ListItem 3</Paragraph></ListItem>
    <ListItem><Paragraph>ListItem 4</Paragraph></ListItem>
    <ListItem><Paragraph>ListItem 5</Paragraph></ListItem>
</List>

<Paragraph><Bold>Bolded</Bold></Paragraph>
<Paragraph><Underline>Underlined</Underline></Paragraph>
<Paragraph><Bold><Underline>Bolded and Underlined</Underline></Bold>
</Paragraph>
<Paragraph><Italic>Italic</Italic></Paragraph>

<Paragraph><Span>The Span element, no inherent rendering</Span>
</Paragraph>
<Paragraph><Run>The Run element, no inherent rendering</Run></Paragraph>

<Paragraph FontSize="15">Subscript, Superscript</Paragraph>

<Paragraph>
    <Run Typography.Variants="Superscript">This text is Superscripted.
</Run> This text isn't.
</Paragraph>
<Paragraph>
    <Run Typography.Variants="Subscript">This text is Subscripted.</Run>
This text isn't.
</Paragraph>
<Paragraph>
    If a font does not support a particular form (such as Superscript) a
default font form will be displayed.
</Paragraph>

<Paragraph FontSize="15">Blocks, breaks, paragraph</Paragraph>

<Section><Paragraph>A block section of text</Paragraph></Section>
<Section><Paragraph>Another block section of text</Paragraph></Section>
```

```
<Paragraph><LineBreak/></Paragraph>
<Section><Paragraph>... and another section, preceded by a
LineBreak</Paragraph></Section>

<Section BreakPageBefore="True"/>
<Section><Paragraph>... and another section, preceded by a
PageBreak</Paragraph></Section>

<Paragraph>Finally, a paragraph. Note the break between this paragraph
...</Paragraph>
<Paragraph TextIndent="25">... and this paragraph, and also the left
indentation.</Paragraph>

<Paragraph><LineBreak/></Paragraph>

</FlowDocument>
```

# 如何：使用 FlowDocument 列分隔特性

项目 • 2023/02/06

此示例演示如何使用 [FlowDocument](#) 的列分隔功能。

## 示例

下面的示例定义 [FlowDocument](#) 并设置 [ColumnGap](#)、[ColumnRuleBrush](#) 和 [ColumnRuleWidth](#) 属性。 [FlowDocument](#) 包含示例内容的单个段落。

XAML

```
<FlowDocumentReader>
  <FlowDocument
    ColumnGap="20.0"
    ColumnRuleBrush="DodgerBlue"
    ColumnRuleWidth="5.0"
    ColumnWidth="140.0"
  >
    <Paragraph Background="AntiqueWhite" TextAlignment="Left">
      This paragraph has the background set to antique white to make its
      boundaries obvious.

      The column gap is the space between columns; this FlowDocument will
      have a column gap of 20 device-independend pixels. The column rule
      is a vertical line drawn in the column gap, and is used to visually
      separate columns; this FlowDocument a Dodger-blue column rule that
      is 5 pixels wide.

      The column rule and column gap both take space between columns. In
      this case, a column gap width of 20 plus a column rule of width of 5
      results in the space between columns being 25 pixels wide, 5 pixels
      for the column rule, and 10 pixels of column gap on either side of the
      column rule.
    </Paragraph>
  </FlowDocument>
</FlowDocumentReader>
```

下图显示 [ColumnGap](#)、[ColumnRuleBrush](#) 和 [ColumnRuleWidth](#) 属性在呈现的 [FlowDocument](#) 中的效果。

This paragraph has the background set to antique white to make its boundaries obvious. The column gap is the space between columns; this FlowDocument will have a column gap of 20 device-independent pixels. The column rule is a vertical line drawn

in the column gap, and is used to visually separate columns; this FlowDocument a Dodger-blue column rule that is 5 pixels wide. The column rule and column gap both take space between columns. In this case, a column gap width of 20 plus a column rule of

width of 5 results in the space between columns being 25 pixels wide, 5 pixels for the column rule, and 10 pixels of column gap on either side of the column rule.



# 版式

项目 · 2023/02/06

WPF 通过 Microsoft ClearType 呈现，该技术增强了文本的清晰度和可读性。 WPF 还支持 OpenType 字体，这些字体提供了 TrueType® 格式定义的字体之外的其他功能。

## 本节内容

[WPF 中的版式](#)

[ClearType 概述](#)

[ClearType 注册表设置](#)

[绘制格式化文本](#)

[高级文本格式设置](#)

[字体](#)

[标志符号](#)

[操作指南主题](#)

## 另请参阅

- [Typography](#)
- [WPF 中的文档](#)
- [OpenType 字体功能](#)
- [优化 WPF 应用程序性能](#)

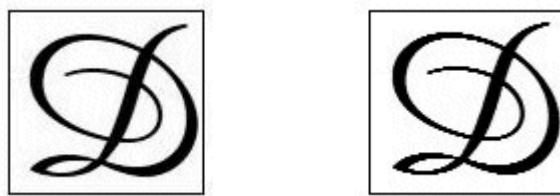
# WPF 中的版式

项目 · 2022/09/27

本主题介绍 WPF 的主要版式功能。这些功能包括改进的文本呈现质量和性能、OpenType 版式支持、增强的国际文本、增强的字体支持和新的文本应用程序编程接口 (Api.)。

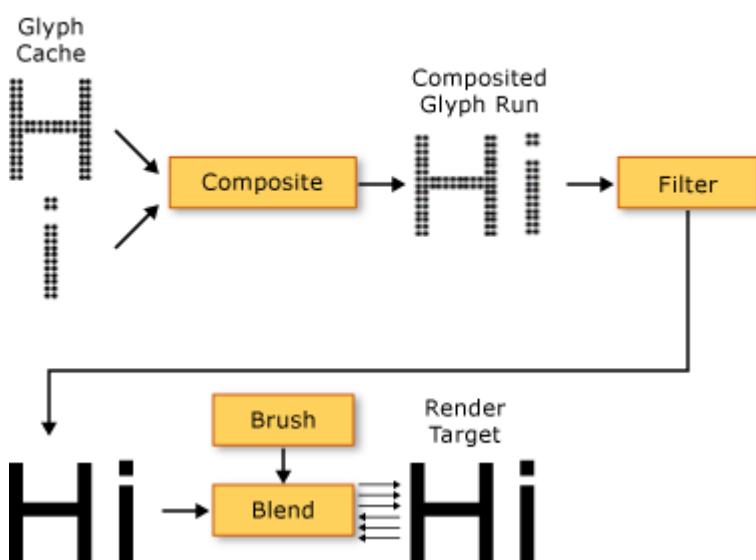
## 改进的文本质量和性能

WPF 中的文本使用 Microsoft ClearType 呈现，从而增强了文本的清晰度和可读性。ClearType 是由 Microsoft 开发的一种软件技术，可提高现有 lcd 上的文本可读性 (Crystal 显示屏)，如笔记本电脑屏幕、Pocket PC 屏幕和平板显示器。ClearType 使用子像素呈现，这允许通过在像素的小数部分对齐字符，以更高的保真度将文本显示为真正的形状。超高的分辨率增加了文本显示中细节的清晰度，使其更便于长时间阅读。WPF 中 ClearType 的另一个改进是 y 方向抗锯齿，这会平滑文本字符中浅曲线的顶部和底部。有关 ClearType 功能的更多详细信息，请参阅 [ClearType 概述](#)。



采用 ClearType y 向抗锯齿的文本

如果你的计算机满足所需硬件的最低级别要求，则可以在 WPF 中对整个文本呈现管道进行硬件加速。不能使用硬件执行加速的呈现会退回软件呈现。硬件加速会影响文本呈现管道的所有阶段（从存储单个标志符号、将标志符号组合到标志符号运行、应用效果），以将 ClearType 混合算法应用于最终显示的输出。有关硬件加速的详细信息，请参阅[图形呈现层](#)。



此外，动画文本（无论是按字符还是按字形）充分利用了由 WPF 启用的图形硬件功能。因此，可生成平滑的文本动画。

## 丰富的版式

OpenType 字体格式是 TrueType® 字体格式的扩展。 OpenType 字体格式由 Microsoft 和 Adobe 共同开发，提供丰富的高级版式功能。[Typography](#) 对象公开了 OpenType 字体的许多高级功能，如样式替代和花体。 Windows SDK 提供了一组结合了丰富功能的示例 OpenType 字体，如 Pericles 和 Pescadero 字体。有关详细信息，请参阅[示例 OpenType 字体包](#)。

Pericles OpenType 字体包含其他一些标志符号，它们提供标准字形集的样式备用项。以下文本显示样式备用字形。

A N C I E N T G R E E K M Y T H O L O G Y

花体是使用精美修饰的装饰性字形，通常与书法相关。以下文本显示 Pescadero 字体的标准和花体字形。

A B C D E F G H I J K L M N  
A B C D E F G H I J K L M N

有关 OpenType 功能的更多详细信息，请参阅[Opentype 字体功能](#)。

## 增强的国际文本支持

WPF 提供以下功能，从而提供增强的国际文本支持：

- 使用自适应测量功能，在所有书写系统中实现自动行距调整。
- 对国际文本的广泛支持。有关详细信息，请参阅[WPF 的全球化](#)。
- 根据不同的语言进行分行、连字和对齐。

## 增强的字体支持

WPF 提供以下功能，从而提供增强的字体支持：

- 所有文本均采用 Unicode。字体行为和选择不再需要字符集或代码页。

- 字体行为与全局设置（如系统区域设置）无关。
- 分隔 `FontWeight`、`FontStretch` 和 `FontStyle` 类型用于定义 `FontFamily`。这比 Win32 编程提供了更大的灵活性，其中斜体和粗体的布尔组合用于定义字体系列。
- 在处理书写方向（横向与纵向）时不受字体名称的影响。
- 使用复合字体技术在可移植 XML 文件中链接字体和字体回退。使用复合字体可以构造全面的多语言字体。复合字体还提供一种可避免显示缺失字形的机制。有关详细信息，请参阅类中 `FontFamily` 的备注。
- 使用一组单语言字体，根据复合字体生成国际字体。在开发多语言字体时，该功能可节省资源成本。
- 在文档中嵌入复合字体，从而能够提供文档可移植性。有关详细信息，请参阅类中 `FontFamily` 的备注。

## 新的文本应用程序编程接口 (API)

WPF 提供了多个文本 API，供开发人员在其应用程序中包含文本时使用。这些 API 分为三个类别：

- **布局和用户界面。** 图形用户界面 (GUI) 的常见文本控件。
- **轻量文本绘制。** 可直接在对象上绘制文本。
- **高级文本格式设置。** 可实现自定义文本引擎。

## 布局和用户界面

在功能的最高级别，文本 API 提供应用程序中的常用 UI 元素，并提供一种简单的方式来呈现文本和与之交互。控件（如和 `PasswordBox`）`RichTextBox` 可实现更高级或专用文本处理。和类（`TextRange` 如、`TextSelection` 和 `TextPointer`）启用有用的文本操作。这些 UI 控件提供了属性（`FontFamily``FontSize` 如、和 `FontStyle`），使您能够控制用于呈现文本的字体。

## 使用位图效果、转换和文本效果

WPF 允许您通过使用位图效果、转换和文本效果等功能来创建直观的文本使用。下面的示例演示了应用于文本的典型类型的投影效果。

# Shadow Text

下面的示例演示了应用于文本的投影效果和噪音。

# Shadow Text

下面的示例演示了应用于文本的外发光效果。

# Shadow Text

以下示例显示了应用于文本的模糊效果。

# Shadow Text

下面的示例演示沿 X 轴放大 150% 得到第二行文本，沿 Y 轴放大 150% 得到第三行文本。

Scaled Text

**Scaled Text**

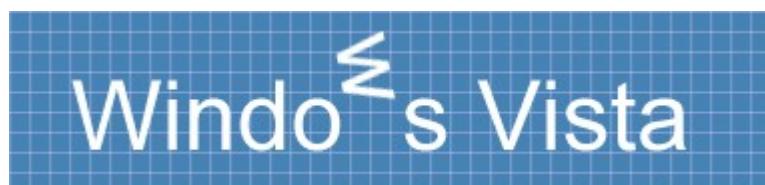
Scaled Text

以下示例演示沿 X 轴倾斜的文本。

*Skewed Text*

**Skewed Text**

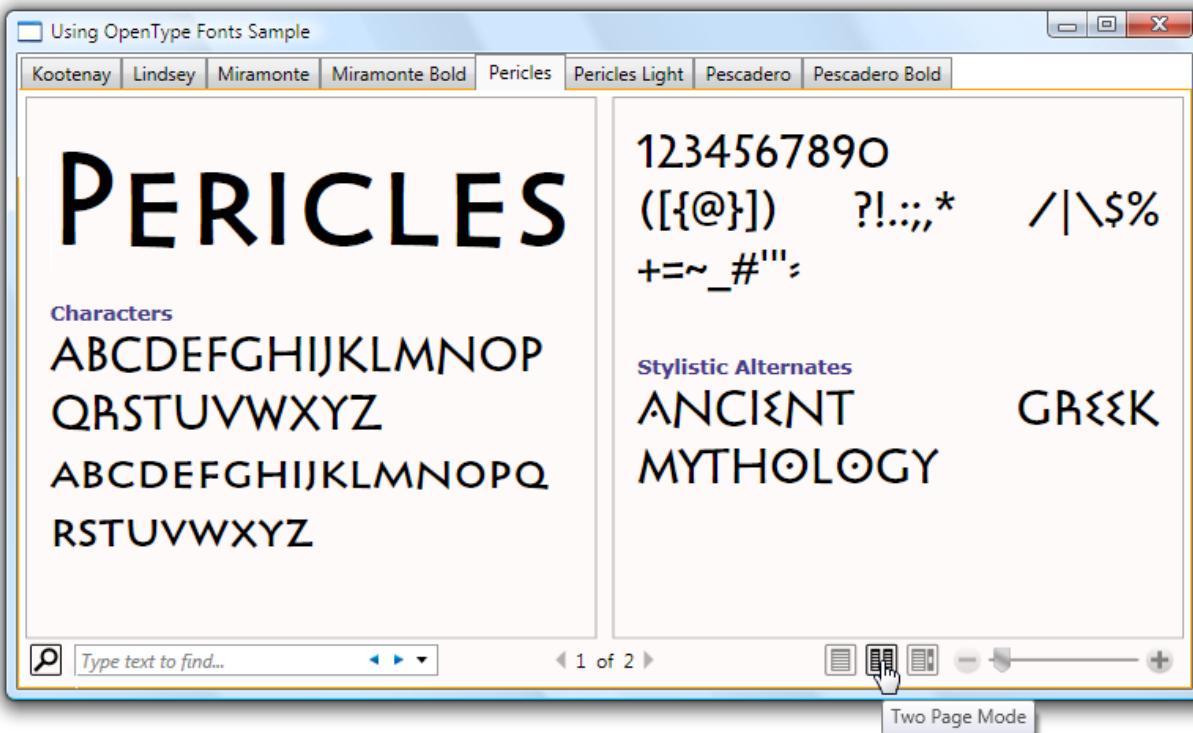
[TextEffect](#)对象是一个帮助器对象，它允许您将文本视为文本字符串中的一个或多个字符组。下面的示例演示发生旋转的单个字符。每个字符都将以 1 秒为间隔单独旋转。



## 使用流文档

除了常见的 UI 控件外，UI 控件。

下面的示例演示了中 `FlowDocumentReader` 托管的文本内容，它提供了搜索、导航、分页和内容缩放支持。



有关详细信息，请参阅 [WPF 中的文档](#)。

## 轻量文本绘制

使用 `DrawText` 对象的方法 `DrawingContext` 可以将文本直接绘制到 WPF 对象。若要使用此方法，请创建 `FormattedText` 对象。使用该对象可以绘制多行文本，可对文本中的每个字符单独设置格式。对象的功能 `FormattedText` 包含 Windows API 中 `DrawText` 标志的许多功能。此外，`FormattedText` 对象还包含省略号支持等功能，其中在文本超出其边界时显示省略号。下面的示例演示应用多种格式的文本，其中第二个和第三个单词应用了线性渐变。

**Lorem ipsum**  
**dolor** sit amet,  
*consectetur*  
*adipisicing elit,*  
sed do eiusmod...

您可以将格式化文本 [Geometry](#) 转换为对象，从而允许您创建其他类型的视觉对象。例如，可以根据文本字符串的轮廓创建 [Geometry](#) 对象。

# Spectrum Outline

以下示例说明了几种通过修改已转换文本的笔划、填充和突出显示来创建悦目的视觉效果的方法。

*Fancy Outlined Text*

BUTTERFLIES

WILDFIRE

有关对象的详细信息 [FormattedText](#)，请参阅 [绘制格式化文本](#)。

## 高级文本格式设置

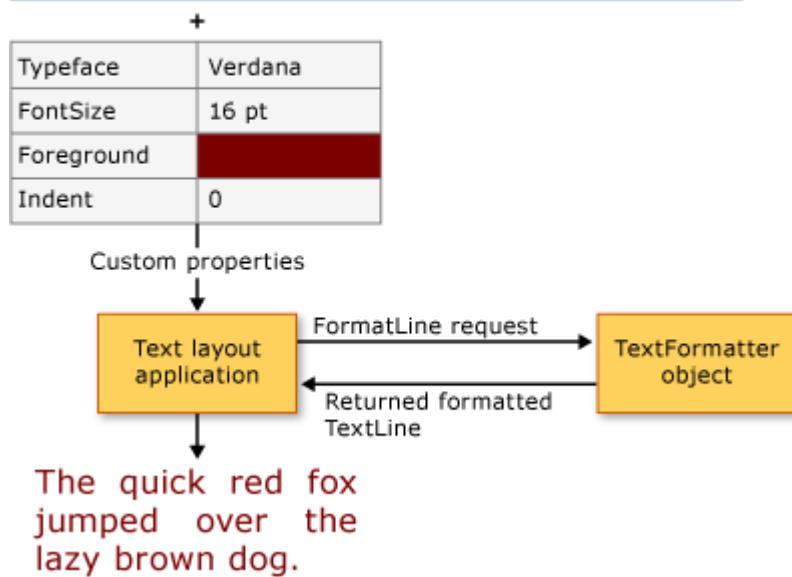
在文本 API 的最高级别的层面上，WPF 提供了使用 [TextFormatter](#) 对象和命名空间中 [System.Windows.Media.TextFormatting](#) 的其他类型创建自定义文本布局的功能。

[TextFormatter](#) 和关联的类允许您实现自定义文本布局，该布局支持您自己定义的字符格式、段落样式、换行符规则和其他用于国际化文本的布局功能。在极少数情况下，您可能希望重写 WPF 文本布局支持的默认实现。但是，如果创建的是文本编辑控件或应用程序，则可能需要不同于默认 WPF 实现的实现。

与传统的文本 API 不同，[TextFormatter](#) 通过一组回调方法与文本布局客户端进行交互。它要求客户端在类的实现 [TextSource](#) 中提供这些方法。下图说明了客户端应用程序和 [TextFormatter](#) 之间的文本布局交互。

Text to display

The quick red fox jumped over the lazy brown dog.



有关创建自定义文本布局的详细信息，请参阅[高级文本格式设置](#)。

## 另请参阅

- [FormattedText](#)
- [TextFormatter](#)
- [ClearType 概述](#)
- [OpenType 字体功能](#)
- [绘制格式化文本](#)
- [高级文本格式设置](#)
- [文本](#)
- [Microsoft 板式](#)

# ClearType 概述

项目 • 2023/02/06

本主题概述了 Windows Presentation Foundation (WPF) 中的 Microsoft ClearType 技术。

## 技术概述

ClearType 是一种由 Microsoft 开发的软件技术，可提高现有 LCD ( 液晶显示器，如笔记本电脑屏幕、Pocket PC 屏幕和平板显示器 ) 上文本的可读性。 ClearType 在工作时访问 LCD 屏幕中每个像素的各个垂直色条元素。在 ClearType 之前，计算机可以显示的最小级别的细节是一个像素，但是有了在 LCD 显示器上运行的 ClearType，我们现在显示的文本特征可以小到一个像素宽度的一部分。超高的分辨率增加了文本显示中细节的清晰度，使其更便于长时间阅读。

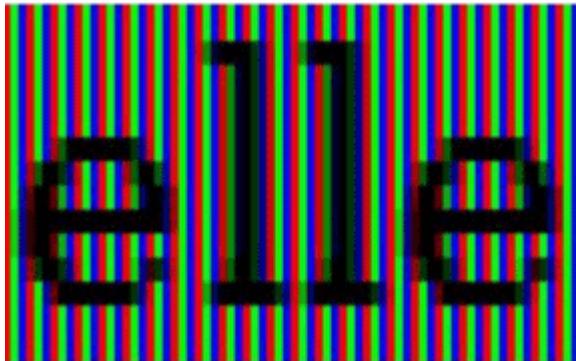
Windows Presentation Foundation (WPF) 中提供的 ClearType 是最新一代的 ClearType，该版本相对于 Microsoft Windows 图形设备接口 (GDI) 中提供的版本有了若干项改进。

## 子像素定位

与之前的 ClearType 版本相比，该版本的一项重大改进是使用了子像素定位。与 GDI 中提供的 ClearType 实现不同，Windows Presentation Foundation (WPF) 中提供的 ClearType 允许字形在像素内启动，而不仅仅是像素的起始边界。由于在定位字形时的这种超高的分辨率，字形的间距和比例更加精确和一致。

以下两个示例演示了使用子像素定位时，字形如何从任意子像素边界处开始。左侧示例使用的是较早版本的 ClearType 呈现器，该呈现器没有采用子像素定位。右侧示例使用的是新版本的 ClearType 呈现器，该呈现器使用了子像素定位。请注意右侧图像中的每个 e 和 l 的呈现方式稍有不同，因为每一个字母都开始于一个不同的子像素。在屏幕上以正常尺寸查看文本时，由于字形图像的高对比度，这种差异并不明显。因为 ClearType 中采用了复杂的颜色筛选，所以才能看出此差异。

Earlier version of ClearType



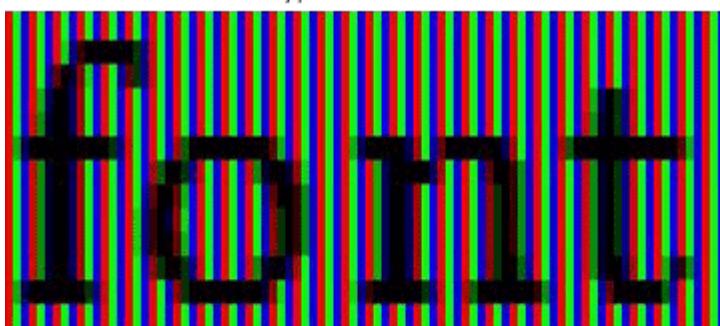
Later version of ClearType



使用较早和较晚版本的 ClearType 显示的文本

以下两个示例将较早的 ClearType 呈现器与新版本的 ClearType 呈现器的输出进行比较。右侧显示的子像素定位显著改善了屏幕上的字体间距，尤其是在较小尺寸处，子像素与整个像素之间的差异在很大程度上代表了字形宽度。请注意，第二张图中字母之间的间距更均匀。子像素定位不断提升文本屏幕的整体外观效果，代表了 ClearType 技术的重大进步。

Earlier version of ClearType



Later version of ClearType

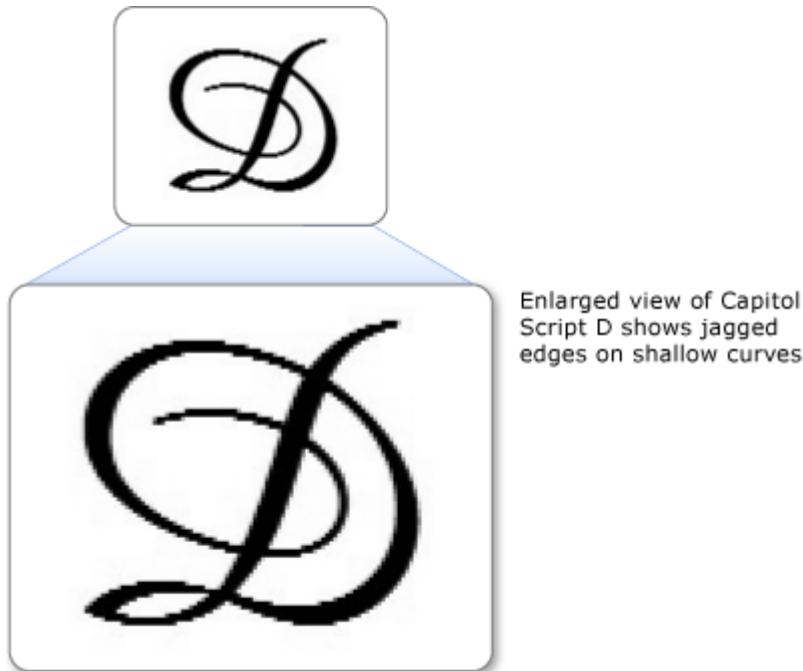


使用较早和较晚版本的 ClearType 显示的文本

## Y 方向抗锯齿功能

Windows Presentation Foundation (WPF) 中 ClearType 的另一个改进是 y 方向抗锯齿功能。 GDI 中的 ClearType 没有 y 方向抗锯齿功能，它能在 x 轴方向提供更好的分辨率，但在 y 轴方向不行。在平缓曲线的顶部和底部，锯齿状边缘会降低其可读性。

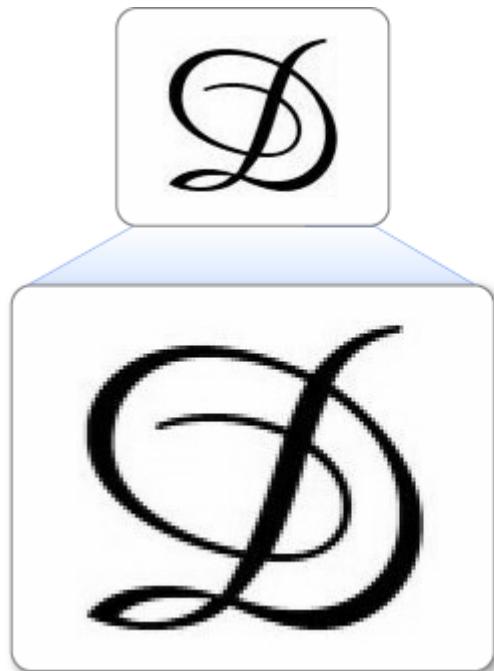
以下示例演示了没有 y 方向抗锯齿功能的效果。在这个示例中，字母顶部和底部的锯齿状边缘非常明显。



平缓曲线上有粗糙边缘的文本

Windows Presentation Foundation (WPF) 中的 ClearType 提供 y 方向的抗锯齿功能，可以使任何锯齿状边缘变得平滑。这对于提高东亚语言的可读性特别重要，在东亚语言中，表意文字的水平和垂直平缓曲线几乎同样多。

以下示例演示了 y 方向抗锯齿功能的效果。在这个示例中，字母顶部和底部对曲线较为平滑。



ClearType with y-direction antialiasing smooths out the jagged edges on shallow curves and other y-direction features

的文本

采用 ClearType y 向抗锯齿的文本

## 硬件加速

Windows Presentation Foundation (WPF) 中的 ClearType 可以利用硬件加速来提高性能，并减少 CPU 负载和系统内存需求。通过使用像素着色器和图形卡的视频内存，ClearType 可以更快呈现文本，特别是使用动画时。

Windows Presentation Foundation (WPF) 中的 ClearType 不会修改系统范围的 ClearType 设置。在 Windows 中禁用 ClearType 会将 Windows Presentation Foundation (WPF) 抗锯齿功能设置为灰度模式。此外，Windows Presentation Foundation (WPF) 中的 ClearType 不会修改 [ClearType Tuner PowerToy](#) 的设置。

Windows Presentation Foundation (WPF) 体系结构设计决策之一是让不依赖于分辨率的布局更好地支持较高分辨率的 DPI 显示器，这种显示器正在日益普及。此决策的后果是 Windows Presentation Foundation (WPF) 不支持某些东亚字体中抗锯齿的文本呈现或位图，因为它们都依赖于分辨率。

## 其他信息

[ClearType 信息](#)

[ClearType 调谐器 PowerToy](#)

## 另请参阅

- [ClearType 注册表设置](#)



# ClearType 注册表设置

项目 • 2023/02/06

本主题概述 WPF 应用程序使用的 Microsoft ClearType 注册表设置。

## 技术概述

将文本呈现给显示设备的 WPF 应用程序使用 ClearType 功能来增强阅读体验。 ClearType 是一种由 Microsoft 开发的软件技术，可提高现有 LCD（液晶显示器，如笔记本电脑屏幕、Pocket PC 屏幕和平板显示器）上文本的可读性。 ClearType 在工作时访问 LCD 屏幕中每个像素的各个垂直色条元素。 有关 ClearType 的详细信息，请参阅 [ClearType 概述](#)。

使用 ClearType 呈现的文本在各种显示设备上查看时可能会明显不同。 例如，少数显示器按蓝色、绿色、红色的顺序实现色条元素，而不是更常见的红色、绿色、蓝色 (RGB) 顺序。

色彩敏感度级别不同的个人在查看时，使用 ClearType 呈现的文本表现得也会明星不同。某些人能够比别人更擅长发觉颜色的微小差异。

针对每种情况，需要修改 ClearType 功能，以便为每个人提供最佳阅读体验。

## 注册表设置

WPF 指定了 4 个用于控制 ClearType 功能的注册表设置：

设置	说明
ClearType 级别	描述 ClearType 色彩清晰度的级别。
伽马级别	描述显示设备的像素颜色组件的级别。
像素结构	描述显示设备的像素排列。
文本对比度级别	描述显示文本的对比度级别。

知道如何引用已标识的 WPF ClearType 注册表设置的外部配置实用工具可以访问这些设置。 还可以直接使用 Windows 注册表编辑器来访问这些值，从而创建或修改这些设置。

如果未设置 WPF ClearType 注册表设置（这是默认状态），则 WPF 应用程序将查询 Windows 系统参数信息以进行字体平滑设置。

① 备注

有关枚举显示设备名称的信息，请参阅 SystemParametersInfo Win32 功能。

## ClearType 级别

借助 ClearType 级别，你可以根据个人的色彩敏感度和感知度来调整文本的呈现。对于某些人而言，在最高级别使用 ClearType 的文本呈现不会产生最佳阅读体验。

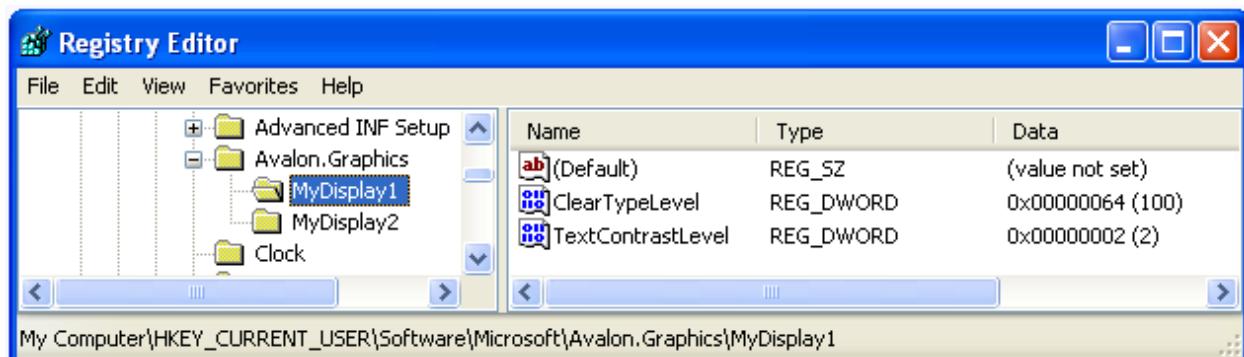
ClearType 级别是一个介于 0 到 100 之间的整数值。默认级别为 100，表示 ClearType 使用显示设备的色条元素的最大能力。但是，ClearType 级别为 0 会将文本呈现为灰度级。通过将 ClearType 级别设置为 0 到 100 之间的某个数字，可以创建适合个人色彩敏感度的中间级别。

## 注册表设置

ClearType 级别的注册表设置位置是对应于特定显示设备名称的单个用户设置：

HKEY\_CURRENT\_USER\Software\Microsoft\Avalon.Graphics\<displayName>

对于用户的每个显示设备名称，定义了 `ClearTypeLevel` DWORD 值。以下屏幕截图显示了 ClearType 级别的注册表编辑器设置。



### ① 备注

WPF 应用程序用两种模式中的任一种呈现文本：使用和不使用 ClearType。不使用 ClearType 呈现文本时，称为灰度呈现。

## 伽玛级别

伽玛级别指的是像素值和亮度之间的非线性关系。伽玛级别设置应对应于显示设备的物理特性；否则呈现的输出中可能会出现失真。例如，文本可能显得太宽或太窄，或者字形的垂直干线的边缘上可能会出现彩色条纹。

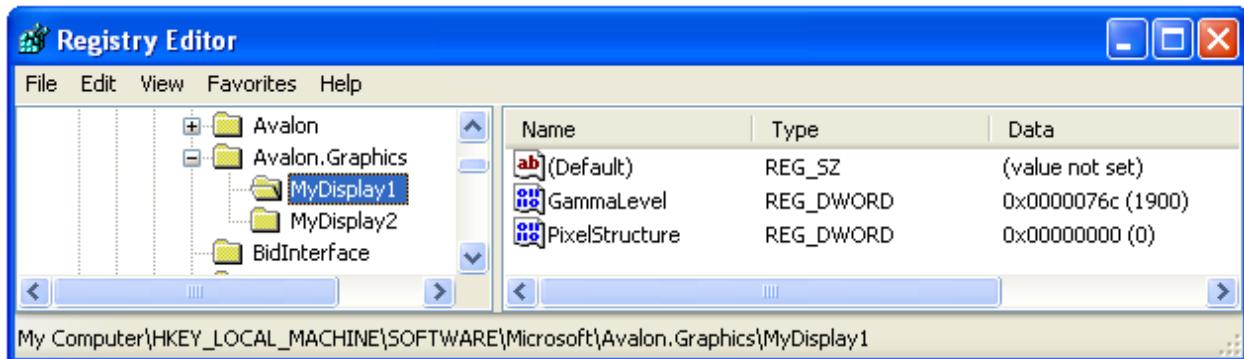
伽马级别是一个介于 1000 到 2200 之间的整数值。默认级别为 1900。

## 注册表设置

伽马级别的注册表设置位置是对应于特定显示设备名称的本地计算机设置：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Avalon.Graphics\<displayName>

对于用户的每个显示设备名称，定义了 `GammaLevel` DWORD 值。以下屏幕快照显示了伽马级别的注册表编辑器设置。



## 像素结构

像素结构描述构成显示设备的像素的类型。像素结构定义为三种类型之一：

类型	值	说明
平面	0	显示设备没有像素结构。这意味着每种颜色的光源均匀分布在像素区域上，称为灰度呈现。这是标准显示设备的工作方式。ClearType 从不应用于呈现的文本。
RGB	1	显示设备的像素由三种色条按以下顺序构成：红色、绿色和蓝色。ClearType 应用于呈现的文本。
BGR	2	显示设备的像素由三种色条按以下顺序构成：蓝色、绿色和红色。ClearType 应用于呈现的文本。注意该顺序与 RGB 类型相反。

像素结构对应于 0 到 2 之间的一个整数值。默认级别为 0，表示平面像素结构。

### ① 备注

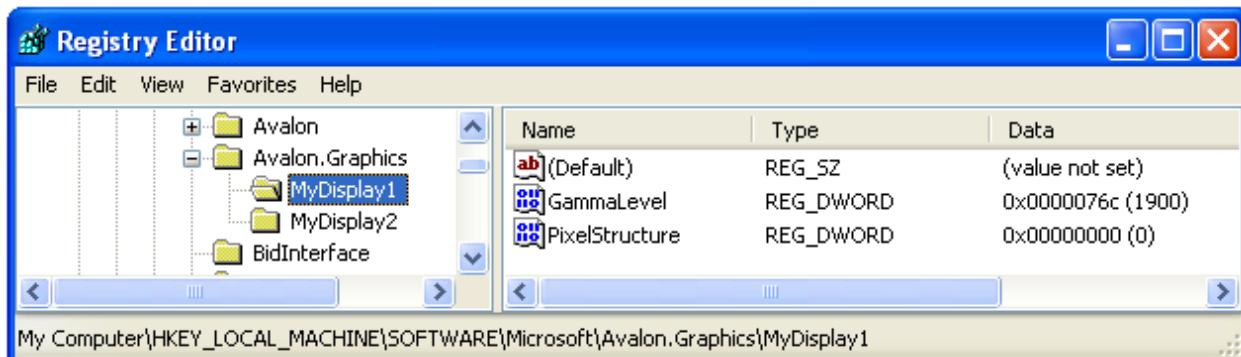
有关枚举显示设备名称的信息，请参阅 `EnumDisplayDevices` Win32 功能。

## 注册表设置

像素结构的注册表设置位置是对应于特定显示设备名称的本地计算机设置：

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Avalon.Graphics\<displayName>

对于用户的每个显示设备名称，定义了 PixelStructure DWORD 值。以下屏幕快照显示了像素结构的注册表编辑器设置。



## 文本对比度级别

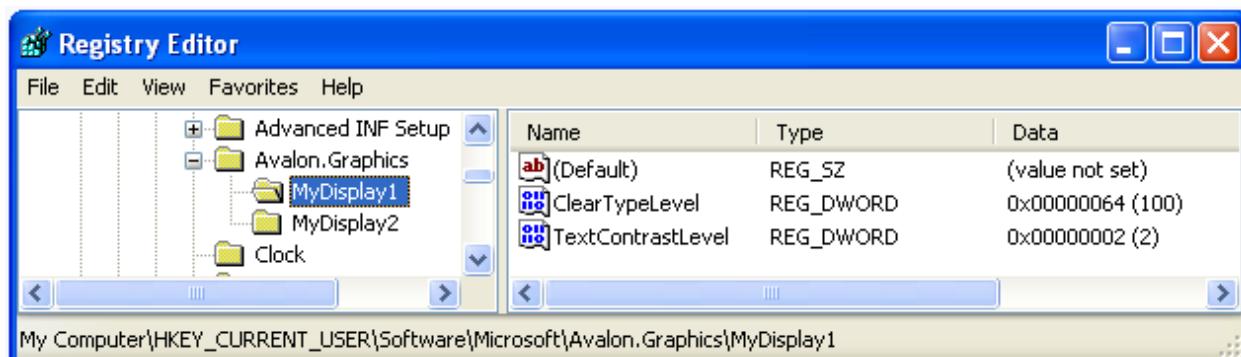
通过文本对比度级别可以根据字形的主体宽度来调整文本的呈现。文本对比度级别是一个 0 到 6 之间的整数值 - 整数值越大，主体就越宽。默认级别为 1。

## 注册表设置

文本对比度级别的注册表设置位置是对应于特定显示设备名称的单个用户设置：

HKEY\_CURRENT\_USER\Software\Microsoft\Avalon.Graphics\<displayName>

对于用户的每个显示设备名称，定义了 TextContrastLevel DWORD 值。以下屏幕快照显示了文本对比度级别的注册表编辑器设置。



## 另请参阅

- [ClearType 概述](#)
- [ClearType 抗锯齿](#)



# 绘制格式化文本

项目 · 2022/09/27

本主题概述了 [FormattedText](#) 对象的功能。此对象为在 Windows Presentation Foundation (WPF) 应用程序中绘制文本提供低级别控制。

## 技术概述

使用 [FormattedText](#) 对象可以绘制多行文本，可对文本中的每个字符单独设置格式。下例演示应用了多种格式的文本。

**Lo**rem **i**psum  
**d**olor sit amet,  
*co*nsectetur  
*ad*ipisicing *el*it,  
sed do eiusmod...

### ① 备注

对于从 Win32 API 迁移的开发人员，[Win32 迁移](#)一节中的表列出了 Win32 DrawText 标志和 Windows Presentation Foundation (WPF) 中的近似等效项。

## 使用格式化文本的原因

WPF 包括多个用于在屏幕中绘制文本的控件。每个控件都面向不同的方案，并具有自己的功能和限制列表。一般而言，当需要有限的文本支持（例如用户界面 (UI) 中的简短句子）时，应使用 [TextBlock](#) 元素。当需要最少的文本支持时，可以使用 [Label](#)。有关详细信息，请参阅 [WPF 中的文档](#)。

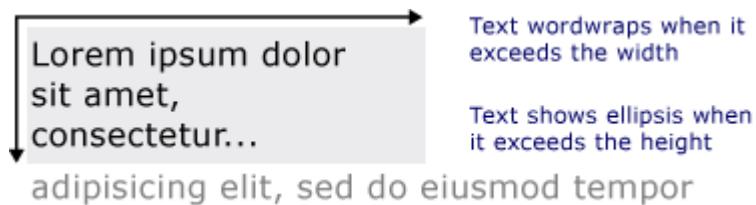
[FormattedText](#) 对象提供的文本格式设置功能比 Windows Presentation Foundation (WPF) 文本控件提供的相应功能更强大，并且在要将文本用作装饰元素时很有用。有关详细信息，请参阅下一节[将格式化文本转换为几何图形](#)。

此外，对于创建面向文本的 [DrawingVisual](#) 派生的对象，[FormattedText](#) 对象非常有用。[DrawingVisual](#) 是一个轻量绘图类，用于绘制形状、图像或文本。有关详细信息，请参阅 [使用 DrawingVisuals 执行测试示例](#)。

# 使用 FormattedText 对象

若要创建格式化文本，请调用 [FormattedText 构造函数](#)来创建 [FormattedText 对象](#)。 创建初始格式化文本字符串后，便可应用某一范围的格式样式。

使用 [MaxTextWidth 属性](#)将文本限制在特定宽度。 文本将自动换行，避免超过指定宽度。 使用 [MaxTextHeight 属性](#)将文本限制在特定高度。 超过指定高度的文本将显示一个省略号“...”。



可向一个或多个字符应用多种格式样式。例如，可以同时调用 [SetFontStyle](#) 和 [SetForegroundBrush](#) 方法来更改文本中前五个字符的格式。

以下代码示例创建一个 [FormattedText 对象](#)，然后向文本应用多种格式化样式。

C#

```
protected override void OnRender(DrawingContext drawingContext)
{
    string testString = "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor";

    // Create the initial formatted text string.
    FormattedText formattedText = new FormattedText(
        testString,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface("Verdana"),
        32,
        Brushes.Black);

    // Set a maximum width and height. If the text overflows these values,
    // an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300;
    formattedText.MaxTextHeight = 240;

    // Use a larger font size beginning at the first (zero-based) character
    // and continuing for 5 characters.
    // The font size is calculated in terms of points -- not as device-
    // independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);

    // Use a Bold font weight beginning at the 6th character and continuing
    // for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11);
```

```
// Use a linear gradient brush beginning at the 6th character and
// continuing for 11 characters.
formattedText.SetForegroundBrush(
    new LinearGradientBrush(
        Colors.Orange,
        Colors.Teal,
        90.0),
    6, 11);

// Use an Italic font style beginning at the 28th character and
// continuing for 28 characters.
formattedText.SetFontStyle(FontStyles.Italic, 28, 28);

// Draw the formatted text string to the DrawingContext of the control.
drawingContext.DrawText(formattedText, new Point(10, 0));
}
```

## 字号度量单位

如同 Windows Presentation Foundation (WPF) 应用程序中的其他文本对象，[FormattedText](#) 对象使用与设备无关的像素作为度量单位。但是，大多数 Win32 应用程序使用分为度量单位。如果要在 Windows Presentation Foundation (WPF) 应用程序中使用以分为单位的显示文本，则需要将与设备无关的单位（每单位 1/96 英寸）转换为分。以下代码示例演示如何执行此转换。

C#

```
// The font size is calculated in terms of points -- not as device-
// independent pixels.
formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);
```

## 将格式化文本转换为几何图形

可将格式化文本转换为 [Geometry](#) 对象，这样便可以创建其他类型的悦目文本。例如，可基于文本字符串的轮廓创建 [Geometry](#) 对象。

Spectrum Outline

以下示例说明了几种通过修改已转换文本的笔划、填充和突出显示来创建悦目的视觉效果的方法。

# *Fancy Outlined Text*

**BUTTERFLIES**

**WILDFIRE**

将文本转换为 [Geometry](#) 对象时，它不再是字符的集合，也就是说不能修改文本字符串中的字符。但是，可修改已转换文本的笔划和填充属性，以此来影响该文本的外观。笔划是指已转换文本的轮廓；填充是指已转换文本的轮廓的内部区域。有关详细信息，请参阅[创建轮廓文本](#)。

还可将格式化文本转换为 [PathGeometry](#) 对象，并使用此对象突出显示文本。例如，可将动画应用于 [PathGeometry](#) 对象，使此动画沿着格式化文本的轮廓显示。

以下示例显示已转换为 [PathGeometry](#) 对象的格式化文本。经过动画处理的椭圆会沿着所呈现文本的笔划路径显示。

**Hello World!**

沿着文本路径几何图形运动的球

有关详细信息，请参阅[如何：为文本创建 PathGeometry 动画](#)。

将格式化文本转换为 [PathGeometry](#) 对象后，可为其创建其他有趣的用法。例如，可剪辑视频，以便在格式化文本中显示。



## Win32 迁移

[FormattedText](#) 用于绘制文本的功能与 Win32 DrawText 函数的功能相似。对于从 Win32 API 迁移的开发人员，下表列出了 Win32 DrawText 标志和 Windows Presentation Foundation (WPF) 中的近似等效项。

DrawText 标志	WPF 等效项	备注
DT_BOTTOM	<a href="#">Height</a>	使用 <a href="#">Height</a> 属性计算相应的 Win32 DrawText "y" 位置。
DT_CALCRECT	<a href="#">Height</a> , <a href="#">Width</a>	使用 <a href="#">Height</a> 和 <a href="#">Width</a> 属性计算输出矩形。
DT_CENTER	<a href="#">TextAlignment</a>	使用值设置为 <a href="#">Center</a> 的 <a href="#">TextAlignment</a> 属性。
DT_EDITCONTROL	无	不需要。间距宽度和最后一行的呈现与框架编辑控件中的相同。
DT_END_ELLIPSIS	<a href="#">Trimming</a>	使用值为 <a href="#">CharacterEllipsis</a> 的 <a href="#">Trimming</a> 属性。  使用 <a href="#">WordEllipsis</a> 来获取带有 DT_WORD_ELLIPSIS 尾部省略号的 Win32 DT_END_ELLIPSIS；在这种情况下，省略号字符仅出现在一行容不下的字词中。
DT_EXPAND_TABS	无	不需要。制表符自动扩展为在每 4 个 em 后停止，这大约为 8 个与语言无关的字符的宽度。
DT_EXTERNALLEADING	无	不需要。行距中始终包括外部间隙。使用 <a href="#">LineHeight</a> 属性创建用户定义的行距。
DT_HIDEPREFIX	无	不支持。在构造 <a href="#">FormattedText</a> 对象之前，从字符串中删除“&”。

DrawText 标志	WPF 等效项	备注
DT_LEFT	TextAlignment	这是默认文本对齐方式。 使用值设置为 <a href="#">Left</a> 的 <a href="#">TextAlignment</a> 属性。 ( 仅限 WPF )
DT_MODIFYSTRING	无	不支持。
DT_NOCLIP	VisualClip	剪辑不会自动发生。 如果要剪切文本，请使用 <a href="#">VisualClip</a> 属性。
DT_NOFULLWIDTHCHARBREAK	无	不支持。
DT_NOPREFIX	无	不需要。 字符串中的“&”字符始终作为正常字符处理。
DT_PATHELLIPSIS	无	使用值为 <a href="#">WordEllipsis</a> 的 <a href="#">Trimming</a> 属性。
DT_PREFIX	无	不支持。 如果想在文本 ( 例如快捷键或链接 ) 中使用下划线，请使用 <a href="#">SetTextDecorations</a> 方法。
DT_PREFIXONLY	无	不支持。
DT_RIGHT	TextAlignment	使用值设置为 <a href="#">Right</a> 的 <a href="#">TextAlignment</a> 属性。 ( 仅限 WPF )
DT_RTLREADING	FlowDirection	将 <a href="#">FlowDirection</a> 属性设置为 <a href="#">RightToLeft</a> 。
DT_SINGLELINE	无	不需要。 <a href="#">FormattedText</a> 对象表现为单行控件，除非设置了 <a href="#">MaxTextWidth</a> 属性或文本包含回车/换行 (CR/LF) 字符。
DT_TABSTOP	无	不支持用户定义的制表位位置。
DT_TOP	Height	不需要。 上对齐是默认设置。 其他垂直定位值可通过使用 <a href="#">Height</a> 属性计算相应的 Win32 DrawText“y”位置来定义。
DT_VCENTER	Height	使用 <a href="#">Height</a> 属性计算相应的 Win32 DrawText“y”位置。
DT_WORDBREAK	无	不需要。 使用 <a href="#">FormattedText</a> 对象会自动发生断字。 无法禁用它。
DT_WORD_ELLIPSIS	Trimming	使用值为 <a href="#">WordEllipsis</a> 的 <a href="#">Trimming</a> 属性。

## 另请参阅

- [FormattedText](#)
- [WPF 中的文档](#)

- WPF 中的版式
- 创建空心文本
- 如何：为文本创建 PathGeometry 动画

# 高级文本格式设置

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 提供了一组可靠的 API，用于在应用程序中包含文本。布局和用户界面 (UI) API，例如 [TextBlock](#)，为文本展示提供最常用和常规用途的元素。绘图 API，例如 [GlyphRunDrawing](#) 和 [FormattedText](#)，为在绘图中包含格式化文本提供方法。在最高级别，WPF 提供了一个可扩展的文本格式引擎，用于控制文本展示的各个方面，如文本存储管理、文本运行格式管理和嵌入对象管理。

本主题介绍了 WPF 文本格式化。它重点介绍了客户端实现和 WPF 文本格式化引擎的使用。

## ① 备注

本文档中的所有代码示例都可以在[高级文本格式化示例](#)中找到。

## 先决条件

本主题假定你熟悉用于文本展示的较高级别 API。大部分用户方案都不需要在本主题中所讨论的高级文本格式化 API。有关不同文本 API 的介绍，请参阅 [WPF 中的文档](#)。

## 高级文本格式设置

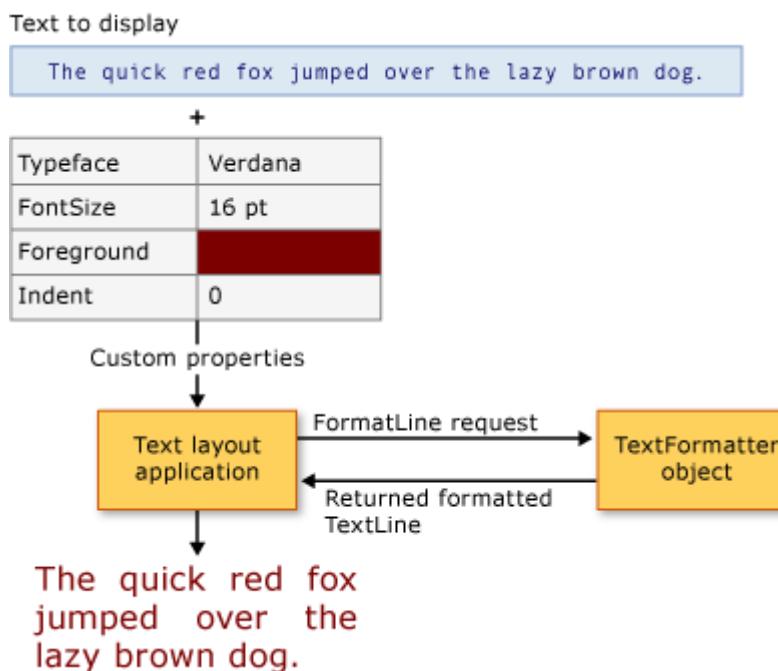
WPF 中的文本布局和 UI 控件提供了格式化属性，通过这些属性，可轻松地在应用程序中加入格式化文本。这些控件会公开许多用于处理文本呈现（包括字样、大小和颜色）的属性。一般情况下，这些控件可以处理应用程序中的大部分文本呈现。但是，某些高级方案要求控制文本存储和文本呈现。为此，WPF 提供了一个可扩展的文本格式化引擎。

WPF 中的高级文本格式化功能包括文本格式化引擎、文本存储、文本运行和格式化属性。文本格式化引擎 [TextFormatter](#) 创建用于展示的文本行。这是通过启动行格式化进程并调用文本格式化程序的 [FormatLine](#) 来实现的。文本格式化程序通过调用文本存储的 [GetTextRun](#) 方法从文本存储中检索文本运行。随后 [TextRun](#) 对象由文本格式化程序形成 [TextLine](#) 对象，并提供给应用程序以进行检查或显示。

## 使用文本格式化程序

[TextFormatter](#) 是 WPF 文本格式化引擎，提供格式化和中断文本行服务。文本格式化程序可处理各种文本字符格式和段落样式，并提供对国际文本布局的支持。

与传统文本 API 不同，[TextFormatter](#) 通过一组回调方法与文本布局客户端交互。它要求客户端在 [TextSource](#) 类的实现中提供这些方法。下图说明了客户端应用程序和 [TextFormatter](#) 之间的文本布局交互。



该文本格式化程序用于从文本存储中检索已格式化的文本行，即 [TextSource](#) 的实现。它通过首先使用 [Create](#) 方法创建文本格式化程序的实例完成。此方法可创建一个文本格式化程序实例，并设置最大行高值和行宽值。创建文本格式化程序的实例后，立刻通过调用 [FormatLine](#) 方法开始行创建进程。[TextFormatter](#) 回叫到文本源以检索文本和格式化参数，该参数用于运行构成行的文本。

下例演示文本存储的格式设置过程。[TextFormatter](#) 对象用于检索来自文本存储中的文本行，然后格式化文本行以绘制到 [DrawingContext](#)。

C#

```
// Create a DrawingGroup object for storing formatted text.
textDest = new DrawingGroup();
DrawingContext dc = textDest.Open();

// Update the text store.
_textStore.Text = textToFormat.Text;
_textStore.FontRendering = _currentRendering;

// Create a TextFormatter object.
TextFormatter formatter = TextFormatter.Create();

// Format each line of text from the text store and draw it.
while (textStorePosition < _textStore.Text.Length)
{
    // Create a textline from the text store using the TextFormatter object.
    using (TextLine myTextLine = formatter.FormatLine(
        _textStore,
        textStorePosition,
```

```

    96*6,
    new GenericTextParagraphProperties(_currentRendering),
    null))
{
    // Draw the formatted text into the drawing context.
    myTextLine.Draw(dc, linePosition, InvertAxes.None);

    // Update the index position in the text store.
    textStorePosition += myTextLine.Length;

    // Update the line position coordinate for the displayed line.
    linePosition.Y += myTextLine.Height;
}
}

// Persist the drawn text content.
dc.Close();

// Display the formatted text in the DrawingGroup object.
myDrawingBrush.Drawing = textDest;

```

## 实现客户端文本存储

扩展文本格式引擎时，需要实现和管理文本存储的各个方面。该任务不是无关紧要的。文本存储负责跟踪文本运行属性、段落属性、嵌入对象和其他类似内容。它还提供具有 [TextRun](#) 对象的文本格式化程序，该文本格式化程序用于创建 [TextLine](#) 对象。

若要处理文本存储的虚拟化，该文本存储必须派生自 [TextSource](#)。[TextSource](#) 定义了文本格式化程序用于从文本存储中检索文本运行的方法。[GetTextRun](#) 文本格式化程序用于检索在行格式化中使用的文本运行的方法。对 [GetTextRun](#) 的调用由文本格式化程序反复生成，直到出现下列条件之一：

- 返回一个 [TextEndOfLine](#) 或一个子类。
- 文本运行的累积宽度超出在创建文本格式化程序的调用或对文本格式化程序的 [FormatLine](#) 方法的调用中指定的最大行宽。
- 返回 Unicode 换行序列，如“CF”、“LF”或“CRLF”。

## 提供文本运行

文本格式设置过程的核心是文本格式化程序和文本存储之间的交互。[TextSource](#) 的实现提供具有 [TextRun](#) 对象的文本格式化程序，以及用于格式化文本运行的属性。此交互通过 [GetTextRun](#) 方法处理，该方法由文本格式化程序调用。

下表显示了一些预定义的 [TextRun](#) 对象。

TextRun 类型	使用情况
TextCharacters	专用文本运行，用于将字符标志符号的表示形式传回给文本格式化程序。
TextEmbeddedObject	专用文本运行，用于提供其中的度量、命中测试和绘制将作为整体执行的内容，例如文本中的按钮或图像。
TextEndOfLine	专用文本运行，用于对行尾进行标记。
TextEndOfParagraph	专用文本运行，用于对段落结尾进行标记。
TextEndOfSegment	用于对段尾进行标记的专用文本运行，例如结束受前一次 <a href="#">TextModifier</a> 运行影响的作用域。
TextHidden	专用文本运行，用于对一系列隐藏字符进行标记。
TextModifier	专用文本运行，用于在其范围内修改文本运行属性。此作用域扩展到下一个匹配的 <a href="#">TextEndOfSegment</a> 文本运行，或下一个 <a href="#">TextEndOfParagraph</a> 。

任何预定义的 [TextRun](#) 对象都可以被子类化。这样，文本源便可为文本格式化程序提供包含自定义数据的文本运行。

下面的示例演示了 [GetTextRun](#) 方法。该文本存储返回 [TextRun](#) 对象给文本格式化程序用以处理。

C#

```
// Used by the TextFormatter object to retrieve a run of text from the text
source.
public override TextRun GetTextRun(int textSourceCharacterIndex)
{
    // Make sure text source index is in bounds.
    if (textSourceCharacterIndex < 0)
        throw new ArgumentOutOfRangeException("textSourceCharacterIndex",
"Value must be greater than 0.");
    if (textSourceCharacterIndex >= _text.Length)
    {
        return new TextEndOfParagraph(1);
    }

    // Create TextCharacters using the current font rendering properties.
    if (textSourceCharacterIndex < _text.Length)
    {
        return new TextCharacters(
            _text,
            textSourceCharacterIndex,
            _text.Length - textSourceCharacterIndex,
            new GenericTextRunProperties(_currentRendering));
    }

    // Return an end-of-paragraph if no more text source.
```

```
    return new TextEndOfParagraph(1);  
}
```

## ① 备注

在此示例中，文本存储为所有文本提供了相同的文本属性。高级文本存储需要实现各自的范围管理，以便单个字符具有不同的属性。

## 指定格式设置属性

[TextRun](#) 对象通过使用文本存储提供的属性进行格式化。这些属性分为两种类型，[TextParagraphProperties](#) 和 [TextRunProperties](#)。[TextParagraphProperties](#) 处理段落包含属性，例如 [TextAlignment](#) 和 [FlowDirection](#)。[TextRunProperties](#) 是每个在段落内运行的文本可能不同的属性，例如前景画笔 [Typeface](#)，和字号。要实现自定义段落和自定义文本运行属性类型，应用程序必须创建分别从 [TextParagraphProperties](#) 和 [TextRunProperties](#) 派生出来的类。

## 另请参阅

- [WPF 中的版式](#)
- [WPF 中的文档](#)

# 字体 (WPF)

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 包括对使用 OpenType 字体的丰富文本表示的支持。 Windows SDK 中包含一个 OpenType 字体示例包。

## 本节内容

[OpenType 字体功能](#)

[将字体与应用程序一起打包](#)

[示例 OpenType 字体包](#)

[操作指南主题](#)

## 另请参阅

- [FontStyle](#)
- [SystemFonts](#)
- [WPF 中的文档](#)
- [WPF 中的版式](#)

# OpenType 字体功能

项目 • 2022/10/19

本主题概述了 Windows Presentation Foundation (WPF) 中 OpenType 字体技术的一些主要功能。

## OpenType 字体格式

OpenType 字体格式是 TrueType® 字体格式的扩展，增加了对 PostScript 字体数据的支持。OpenType 字体格式由 Microsoft 和 Adobe Corporation 联合开发。无论字体包含 TrueType 边框还是 CFF (PostScript) 边框，OpenType 字体和支持 OpenType 字体的操作系统服务都向用户提供一种简单的字体安装和使用方式。

OpenType 字体格式解决了以下开发人员挑战：

- 更广泛的多平台支持。
- 更出色的国际字符集支持。
- 更优的字体数据保护。
- 更小的文件大小，让字体发布更加高效。
- 更广泛的高级版式控件支持。

### ① 备注

Windows SDK 包含一组可用于 Windows Presentation Foundation (WPF) 应用程序的示例 OpenType 字体。这些字体提供本主题余下部分所述的大多数功能。有关详细信息，请参阅[示例 OpenType 字体包](#)。

有关 OpenType 字体格式的详细信息，请参阅 [OpenType 规范](#)。

## 高级版式扩展

高级版式表格（OpenType 布局表格）扩展了具有 TrueType 或 CFF 边框的字体的功能。OpenType 布局字体包含一些其他信息，可扩展字体功能以支持高质量国际版式。大多数 OpenType 字体仅体现全部可用 OpenType 功能的一部分。OpenType 字体提供以下功能。

- 字符与字形之间的丰富映射，可支持连字、定位格式、备用项以及其他字体替换功能。

- 支持二维定位和字形附加。
- 字体中包含的显式脚本和语言信息，使文本处理应用程序可相应调整其行为。

在 OpenType 规范的“[字体文件表格](#)”部分中对 OpenType 布局表格进行了更详细的介绍。

此概述的其余部分介绍了一些直观有趣的 OpenType 功能的广度和灵活性（这些功能由 [Typography 对象的属性公开](#)）。有关此对象的详细信息，请参阅[版式类](#)。

## 变量

变量用于呈现不同的版式风格，例如上标和下标。

### 上标和下标

通过 [Variants](#) 属性可以设置 OpenType 字体的上标和下标值。

以下文本显示 Palatino Linotype 字体的上标。

2<sup>3</sup> 14<sup>th</sup>

以下标记示例演示如何使用 [Typography 对象的属性](#)定义 Palatino Linotype 字体的上标。

#### XAML

```
<Paragraph FontFamily="Palatino Linotype">
    2<Run Typography.Variants="Superscript">3</Run>
    14<Run Typography.Variants="Superscript">th</Run>
</Paragraph>
```

以下文本显示 Palatino Linotype 字体的下标。

H<sub>2</sub>O Footnote<sub>4</sub>

以下标记示例演示如何使用 [Typography 对象的属性](#)定义 Palatino Linotype 字体的下标。

#### XAML

```
<Paragraph FontFamily="Palatino Linotype">
    H<Run Typography.Variants="Subscript">2</Run>O
    Footnote<Run Typography.Variants="Subscript">4</Run>
</Paragraph>
```

## 上标和下标的修饰用法

也可使用上标和下标来营造混合大小写文本的修饰效果。以下文本显示 Palatino Linotype 字体的上标和下标文本。注意，大写字母不受影响。

Chapter One

Chapter One

以下标记示例演示如何使用 [Typography](#) 对象的属性定义字体的上标和下标。

### XAML

```
<Paragraph FontFamily="Palatino Linotype" Typography.Variants="Superscript">
    Chapter One
</Paragraph>
<Paragraph FontFamily="Palatino Linotype" Typography.Variants="Subscript">
    Chapter One
</Paragraph>
```

## 大写字母

大写字母是一组以大写样式字形呈现文本的版式形式。通常情况下，当以全大写呈现文本时，字母之间的间距可能看起来很小，字母的权重和比例看起来会很大。OpenType 支持多种大写字母的样式格式，包括小体大写字母、小号大写字母、标题和大写字母间距。通过这些样式格式可控制大写字母的外观。

以下文本显示 Pescadero 字体的标准大写字母，其后接样式为“SmallCaps”和“AllSmallCaps”的字母。本例中，对所有三个单词均使用相同的字体大小。

CAPITALS CAPITALS CAPITALS

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pescadero 字体的大写字母。使用“SmallCaps”格式时会忽略任何前导大写字母。

### XAML

```
<Paragraph FontFamily="Pescadero" FontSize="48">
    <Run>CAPITALS</Run>
    <Run Typography.Capitals="SmallCaps">Capitals</Run>
    <Run Typography.Capitals="AllSmallCaps">Capitals</Run>
</Paragraph>
```

## 标题大写字母

标题大写字母权重和比例更小，外观比普通大写字母更加雅致。标题大写字母通常用于作为标题的大号字体中。以下文本显示 Pescadero 字体的普通大写字母和标题大写字母。请注意第二行文本的宽度更窄。

CHAPTER ONE  
CHAPTER ONE

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pescadero 字体的标题大写字母。

### XAML

```
<Paragraph FontFamily="Pescadero">
    <Run Typography.Capitals="Titling">chapter one</Run>
</Paragraph>
```

## 大写字母间距

大写字母间距功能让你可以在使用全大写字母文本时提供更宽的间距。大写字母通常设计为与小写字母混合使用。大写字母和小写字母之间看起来比较美观的间距在使用全大写字母时可能会显得过紧。以下文本显示 Pescadero 字体的普通间距和大写字母间距。

CHAPTER ONE  
CHAPTER ONE

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pescadero 字体的大写字母间距。

### XAML

```
<Paragraph FontFamily="Pescadero">
    <Run Typography.CapitalSpacing="True">CHAPTER ONE</Run>
</Paragraph>
```

## 连字

连字是为使文本更具可读性或更加美观而由两个或更多字形形成的一个单一字形。

OpenType 字体支持四种类型的连字：

- **标准连字**。旨在增强可读性。标准连字包括“fi”、“fl”和“ff”。
- **上下文连字**。旨在通过在组成连字的字符之间提供更好的联结行为来增强可读性。
- **自由连字**。重在修饰性，并非专为可读性而设计。
- **历史连字**。重在历史性，并非专为可读性而设计。

以下文本显示 Pericles 字体的标准连字字形。

FI FL TH TT TV TW TY VT WT YT

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pericles 字体的标准连字字形。

#### XAML

```
<Paragraph FontFamily="Pericles" Typography.StandardLigatures="True">
    <Run Typography.StylisticAlternates="1">FI</Run>
    <Run Typography.StylisticAlternates="1">FL</Run>
    <Run Typography.StylisticAlternates="1">TH</Run>
    <Run Typography.StylisticAlternates="1">TT</Run>
    <Run Typography.StylisticAlternates="1">TV</Run>
    <Run Typography.StylisticAlternates="1">TW</Run>
    <Run Typography.StylisticAlternates="1">TY</Run>
    <Run Typography.StylisticAlternates="1">VT</Run>
    <Run Typography.StylisticAlternates="1">WT</Run>
    <Run Typography.StylisticAlternates="1">YT</Run>
</Paragraph>
```

以下文本显示 Pericles 字体的自由连字字形。

CO LA LE LI LL LO LU

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pericles 字体的自由连字字形。

#### XAML

```
<Paragraph FontFamily="Pericles" Typography.DiscretionaryLigatures="True">
    <Run Typography.StylisticAlternates="1">CO</Run>
    <Run Typography.StylisticAlternates="1">LA</Run>
    <Run Typography.StylisticAlternates="1">LE</Run>
    <Run Typography.StylisticAlternates="1">LI</Run>
    <Run Typography.StylisticAlternates="1">LL</Run>
    <Run Typography.StylisticAlternates="1">LO</Run>
    <Run Typography.StylisticAlternates="1">LU</Run>
</Paragraph>
```

默认情况下，Windows Presentation Foundation (WPF) 中的 OpenType 字体启用标准连字。例如，如果使用 Palatino Linotype 字体，则标准连字“fi”、“ff”和“fl”显示为组合字符字形。请注意，每个标准连字的字符对之间彼此相连。

fi ff fl

但是，可禁用标准连字功能，从而使“ff”等标准连字显示为两个单独的字形，而不显示为一个组合字符字形。

fi ff fl

以下标记示例演示如何使用 [Typography](#) 对象的属性禁用 Palatino Linotype 字体的标准连字字形。

XAML

```
<!-- Set standard ligatures to false in order to disable feature. -->
<Paragraph Typography.StandardLigatures="False" FontFamily="Palatino
Linotype" FontSize="72">
    fi ff fl
</Paragraph>
```

## 花体

花体是使用精美修饰的装饰性字形，通常与书法相关。以下文本显示 Pescadero 字体的标准和花体字形。

A B C D E F G H I J K L M N  
A**B** C**D**E**F** G**H**I**J**K**L**M**N**

标准字形文本

花体通常用作事件公告等简短文章中的修饰元素。以下文本使用花体强调事件名称的大写字母。

Wishing you a  
Happy New Year!  
文本

以下标记示例演示如何使用 [Typography](#) 对象的属性定义字体花体。

#### XAML

```
<Paragraph FontFamily="Pescadero" TextBlock.TextAlignment="Center">
    Wishing you a<LineBreak/>
    <Run Typography.StandardSwashes="1" FontSize="36">Happy New Year!</Run>
</Paragraph>
```

## 连接形式花体

花体字形的某些组合可能导致文本外观欠佳，例如相邻字母的下行处出现重叠。通过连接形式花体，可使用能生成更佳外观的替代花体字形。以下文本显示同一单词应用连接形式花体前后的外观。

Lyon Lyon

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pescadero 字体的连接形式花体。

#### XAML

```
<Paragraph FontFamily="Pescadero" Typography.StandardSwashes="1">
    Lyon <Run Typography.ContextualSwashes="1">L</Run>yon
</Paragraph>
```

## 备用项

备用项是可替代标准字形的字形。OpenType 字体（例如以下示例中使用的 Pericles 字体）可包含用于塑造不同文本外观的备用字形。以下文本显示 Pericles 字体的标准字形。

ANCIENT GREEK MYTHOLOGY

Pericles OpenType 字体包含其他字形，可为标准自行集提供样式备用项。以下文本显示样式备用字形。

ANCIENT GREEK MYTHOLOGY 的文本

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pericles 字体的样式备用字形。

### XAML

```
<Paragraph FontFamily="Pericles">
    <Run Typography.StylisticAlternates="1">A</Run>NCIENT
    GR<Run Typography.StylisticAlternates="1">EE</Run>K
    MYTH<Run Typography.StylisticAlternates="1">O</Run>LOGY
</Paragraph>
```

以下文本显示 Pericles 字体的几种其他样式备用字形。



以下标记示例演示如何定义其他样式备用字形。

### XAML

```
<Paragraph FontFamily="Pericles">
    <Run Typography.StylisticAlternates="1">A</Run>
    <Run Typography.StylisticAlternates="2">A</Run>
    <Run Typography.StylisticAlternates="3">A</Run>
    <Run Typography.StylisticAlternates="1">C</Run>
    <Run Typography.StylisticAlternates="1">E</Run>
    <Run Typography.StylisticAlternates="1">G</Run>
    <Run Typography.StylisticAlternates="1">O</Run>
    <Run Typography.StylisticAlternates="1">Q</Run>
    <Run Typography.StylisticAlternates="1">R</Run>
    <Run Typography.StylisticAlternates="2">R</Run>
    <Run Typography.StylisticAlternates="1">S</Run>
    <Run Typography.StylisticAlternates="1">Y</Run>
</Paragraph>
```

## 随机备用连接形式

随机备用连接形式为单个字符提供多种备用字形。 实现脚本类型字体时，此功能可通过使用一组随机选择的外观稍有差异的字形来模拟手写内容。 以下文本使用 Lindsey 字体的随机备用连接形式。 请注意字母“a”外观稍有变化



以下标记示例演示如何使用 `Typography` 对象的属性定义 Lindsey 字体的随机备用连接形式。

### XAML

```
<TextBlock FontFamily="Lindsey">
    <Run Typography.ContextualAlternates="True">
```

```
a banana in a cabana  
</Run>  
</TextBlock>
```

## 历史形式

历史形式指过去常见的版式约定。以下文本使用 Palatino Linotype 字体的一种历史字形形式显示短语“Boston, Massachusetts”。

Boston, Maffachufettf 文本

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Palatino Linotype 字体的历史形式。

XAML

```
<Paragraph FontFamily="Palatino Linotype">  
    <Run Typography.HistoricalForms="True">Boston, Massachusetts</Run>  
</Paragraph>
```

## 数字样式

OpenType 字体支持多种可用于文本中数值的功能。

### 分数

OpenType 字体支持多种分数样式，包括横式分数和竖式分数。

以下文本显示 Palatino Linotype 字体的分数样式。

1/8 1/4 3/8 1/2 5/8 3/4 7/8

$\frac{1}{8}$   $\frac{1}{4}$   $\frac{3}{8}$   $\frac{1}{2}$   $\frac{5}{8}$   $\frac{3}{4}$   $\frac{7}{8}$  的文本

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Palatino Linotype 字体的分数样式。

XAML

```
<Paragraph FontFamily="Palatino Linotype" Typography.Fraction="Slashed">  
    1/8 1/4 3/8 1/2 5/8 3/4 7/8  
</Paragraph>  
<Paragraph FontFamily="Palatino Linotype" Typography.Fraction="Stacked">
```

1/8 1/4 3/8 1/2 5/8 3/4 7/8  
</Paragraph>

## 旧样式数字

OpenType 字体支持旧样式数字格式。此格式对于显示不再是标准样式的数字非常有用。以下文本以 Palatino Linotype 字体的标准和旧样式数字格式显示 18 世纪日期。

July 4, 1776      July 4, 1776

以下文本显示 Palatino Linotype 字体的标准数字，后跟旧样式数字。

1234567890 1234567890

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Palatino Linotype 字体的旧样式数字。

XAML

```
<Paragraph FontFamily="Palatino Linotype">
  <Run Typography.NumeralStyle="Normal">1234567890</Run>
  <Run Typography.NumeralStyle="OldStyle">1234567890</Run>
</Paragraph>
```

## 比例数字和表格式数字

OpenType 字体支持比例数字和表格式数字功能，可在使用数字时控制宽度对齐。比例数字将每个数字视为具有不同的宽度—“1”窄于“5”。表格式数字被视为宽度相等的数字，因此它们可垂直对齐，从而增强财务类型信息的可读性。

以下文本使用 Miramonte 字体显示第一列中的两个表格式数字。请注意数字“5”和“1”之间的宽度差异。第二列显示相同的两个数值，并通过使用表格式数字功能调整其宽度。

550,689      550,689  
114,131      114,131 的文本

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Miramonte 字体的比例数字和表格式数字。

XAML

```
<TextBlock FontFamily="Miramonte">
  <Run Typography.NumeralAlignment="Proportional">114,131</Run>
</TextBlock>
<TextBlock FontFamily="Miramonte">
  <Run Typography.NumeralAlignment="Tabular">114,131</Run>
</TextBlock>
```

## 斜线零

OpenType 字体支持斜线零数字格式来强调字母“O”和数字“0”之间的差异。 斜线零数字通常用于财务和商务信息中的标识符。

以下文本显示使用 Miramonte 字体的订单标识符。 第一行使用标准数字。 第二行使用斜线零数字，以便更易于与大写字母“O”进行区分。

Order #0048-OTC-390  
Order #0048-OTC-390

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Miramonte 字体的斜线零数字。

### XAML

```
<Paragraph FontFamily="Miramonte">
  <Run>Order #0048-OTC-390</Run>
  <LineBreak/>
  <Run Typography.SlashedZero="True">Order #0048-OTC-390</Run>
</Paragraph>
```

## 版式类

[Typography](#) 对象公开 OpenType 字体支持的功能集。 通过在标记中设置 [Typography](#) 的属性，可轻松创作使用 OpenType 功能的文档。

以下文本显示 Pescadero 字体的标准大写字母，其后接样式为“SmallCaps”和“AllSmallCaps”的字母。 本例中，对所有三个单词均使用相同的字体大小。

CAPITALS CAPITALS CAPITALS

以下标记示例演示如何使用 [Typography](#) 对象的属性定义 Pescadero 字体的大写字母。 使用“SmallCaps”格式时会忽略任何前导大写字母。

## XAML

```
<Paragraph FontFamily="Pescadero" FontSize="48">
    <Run>CAPITALS</Run>
    <Run Typography.Capitals="SmallCaps">Capitals</Run>
    <Run Typography.Capitals="AllSmallCaps">Capitals</Run>
</Paragraph>
```

以下代码示例完成与先前的标记事例相同任务。

## C#

```
MyParagraph.FontFamily = new FontFamily("Pescadero");
MyParagraph.FontSize = 48;

Run run_1 = new Run("CAPITALS ");
MyParagraph.Inlines.Add(run_1);

Run run_2 = new Run("Capitals ");
run_2.Typography.Capitals = FontCapitals.SmallCaps;
MyParagraph.Inlines.Add(run_2);

Run run_3 = new Run("Capitals");
run_3.Typography.Capitals = FontCapitals.AllSmallCaps;
MyParagraph.Inlines.Add(run_3);

MyParagraph.Inlines.Add(new LineBreak());
```

## 版式类属性

下表列出了 **Typography** 对象的属性、值和默认设置。

properties	值	默认值
AnnotationAlternates	数值 - 字节	0
Capitals	AllPetiteCaps   AllSmallCaps   Normal   PetiteCaps   SmallCaps   Titling   Unicase	FontCapitals.Normal
CapitalSpacing	Boolean	false
CaseSensitiveForms	Boolean	false
ContextualAlternates	Boolean	true
ContextualLigatures	Boolean	true
ContextualSwashes	数值 - 字节	0

properties	值	默认值
DiscretionaryLigatures	Boolean	false
EastAsianExpertForms	Boolean	false
EastAsianLanguage	HojoKanji   Jis04   Jis78   Jis83   Jis90   Nlckanji   Normal   Simplified   Traditional   TraditionalNames	FontEastAsianLanguage.Normal
EastAsianWidths	Full   Half   Normal   Proportional   Quarter   Third	FontEastAsianWidths.Normal
Fraction	Normal   Slashed   Stacked	FontFraction.Normal
HistoricalForms	Boolean	false
HistoricalLigatures	Boolean	false
Kerning	Boolean	true
MathematicalGreek	Boolean	false
NumericalAlignment	Normal   Proportional   Tabular	FontNumericalAlignment.Normal
NumericalStyle	Boolean	FontNumericalStyle.Normal
SlashedZero	Boolean	false
StandardLigatures	Boolean	true
StandardSwashes	数值 - 字节	0
StylisticAlternates	数值 - 字节	0
StylisticSet1	Boolean	false
StylisticSet2	Boolean	false
StylisticSet3	Boolean	false
StylisticSet4	Boolean	false
StylisticSet5	Boolean	false
StylisticSet6	Boolean	false
StylisticSet7	Boolean	false
StylisticSet8	Boolean	false
StylisticSet9	Boolean	false

properties	值	默认值
StylisticSet10	Boolean	false
StylisticSet11	Boolean	false
StylisticSet12	Boolean	false
StylisticSet13	Boolean	false
StylisticSet14	Boolean	false
StylisticSet15	Boolean	false
StylisticSet16	Boolean	false
StylisticSet17	Boolean	false
StylisticSet18	Boolean	false
StylisticSet19	Boolean	false
StylisticSet20	Boolean	false
Variants	Inferior   Normal   Ordinal   Ruby   Subscript   Superscript	FontVariants.Normal

## 另请参阅

- [Typography](#)
- [OpenType 规范](#)
- [WPF 中的版式](#)
- [示例 OpenType 字体包](#)
- [将字体与应用程序一起打包](#)

# 将字体与应用程序一起打包

项目 • 2022/09/27

本主题概述如何将字体随 Windows Presentation Foundation (WPF) 应用程序一起打包。

## ① 备注

与大多数软件类型一样，字体文件也采用许可模式，而不是出售。用来控制字体使用的许可证因供应商而异，但一般说来，大多数许可证，包括那些涵盖随应用程序和 Microsoft 提供的字体 Windows 的许可证，都不允许将字体嵌入应用程序中或以其他方式重新发布。因此，开发人员有责任确保自己具备必要的许可权，可以在应用程序中嵌入相应的字体或者以其他方式重新分布。

## 字体打包简介

可以轻松地将字体作为资源打包在 WPF 应用程序中，以显示用户界面文本和基于文本的其他类型的内容。字体可以与应用程序的程序集文件分开，也可以嵌入到这些程序集文件中。还可以创建纯资源字体库，以供应用程序引用。

OpenType 和 TrueType® 字体包含类型标志 `fsType`，指示字体的字体嵌入许可权。但是，这个类型标志仅引用存储在文档中的嵌入字体，而不引用嵌入到应用程序中的字体。可以通过创建 `GlyphTypeface` 对象并引用其 `EmbeddingRights` 属性来检索字体的字体嵌入权。有关 `fsType` 标志的详细信息，请参考 [OpenType 规范](#) 的“OS/2 and Windows Metrics”( OS/2 和 Windows 规范 )一节。

[Microsoft 版式](#)网站包括联系信息，这些信息可帮助查找特定的字体供应商或者查找自定义作品的字体供应商。

## 将字体作为内容项添加

可以将字体作为项目内容项添加到应用程序中，这些项目内容项与应用程序的程序集文件是分开的。这意味着内容项不会作为程序集中的资源嵌入。以下项目文件示例演示如何定义内容项。

XML

```
<Project DefaultTargets="Build"
      xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- Other project build settings ... -->
  <ItemGroup>
```

```
<Content Include="Peric.ttf" />
<Content Include="Pericl.ttf" />
</ItemGroup>
</Project>
```

为了确保应用程序可以在运行时使用字体，这些字体必须能够从应用程序的部署目录中访问。 使用应用程序项目文件中的 `<CopyToOutputDirectory>` 元素，可以在生成过程中将字体自动复制到应用程序部署目录中。 以下项目文件示例演示如何将字体复制到部署目录中。

#### XML

```
<ItemGroup>
<Content Include="Peric.ttf">
<CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</Content>
<Content Include="Pericl.ttf">
<CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
</Content>
</ItemGroup>
```

以下代码示例演示如何将应用程序的字体作为内容项来引用，所引用的内容项必须与应用程序的程序集文件位于同一个目录中。

#### XAML

```
<TextBlock FontFamily="./#Pericles Light">
Aegean Sea
</TextBlock>
```

## 将字体作为资源项添加

可以将字体作为项目资源项添加到应用程序中，这些项目资源项会嵌入到应用程序的程序集文件中。 对资源使用单独的子目录有助于对应用程序的项目文件进行整理。 以下项目文件示例演示如何在单独的子目录中将字体定义为资源项。

#### XML

```
<Project DefaultTargets="Build"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<!-- Other project build settings ... --&gt;

&lt;ItemGroup&gt;
&lt;Resource Include="resources\Peric.ttf" /&gt;
&lt;Resource Include="resources\Pericl.ttf" /&gt;
&lt;/ItemGroup&gt;
&lt;/Project&gt;</pre>
```

## ① 备注

将字体作为资源添加到应用程序中时，请确保设置的是 `<Resource>` 元素，而不是应用程序项目文件中的 `<EmbeddedResource>` 元素。不支持将 `<EmbeddedResource>` 元素用于生成操作。

以下标记示例演示如何引用应用程序的字体资源。

XAML

```
<TextBlock FontFamily=".//resources/#Pericles Light">  
    Aegean Sea  
</TextBlock>
```

## 在代码中引用字体资源项

为了在代码中引用字体资源项，必须提供由两部分组成的字体资源引用：基本统一资源标识符 (URI) 和字体位置引用。这些值用作 `FontFamily` 方法的参数。以下代码示例演示如何在名为 `resources` 的项目子目录中引用应用程序的字体资源。

C#

```
// The font resource reference includes the base URI reference (application  
directory level),  
// and a relative URI reference.  
myTextBlock.FontFamily = new FontFamily(new Uri("pack://application:,,,/"),  
    ".//resources/#Pericles Light");
```

基本统一资源标识符 (URI) 可以包括字体资源所在的应用程序子目录。在这种情况下，字体位置引用无需指定目录，但必须包括前缀“`./`”，该前缀指示字体资源位于由基本统一资源标识符 (URI) 指定的相同目录中。以下代码示例演示另一种引用字体资源项的方法，该示例与上面的代码示例等效。

C#

```
// The base URI reference can include an application subdirectory.  
myTextBlock.FontFamily = new FontFamily(new  
    Uri("pack://application:,,,/resources/"), "./#Pericles Light");
```

## 在同一应用程序子目录中引用字体

可以将应用程序内容和资源文件放在应用程序项目中由用户定义的同一子目录中。以下项目文件示例显示在同一子目录中定义的内容页和字体资源。

XAML

```
<ItemGroup>
  <Page Include="pages\HomePage.xaml" />
</ItemGroup>
<ItemGroup>
  <Resource Include="pages\Peric.ttf" />
  <Resource Include="pages\PericL.ttf" />
</ItemGroup>
```

由于应用程序内容和字体位于同一子目录中，因此字体引用是相对于应用程序内容的。以下示例演示当字体与应用程序位于同一目录中时，如何引用应用程序的字体资源。

XAML

```
<TextBlock FontFamily=".#/Pericles Light">
  Aegean Sea
</TextBlock>
```

C#

```
// The font resource reference includes the base Uri (application directory
// level),
// and the file resource location, which is relative to the base Uri.
myTextBlock.FontFamily = new FontFamily(new Uri("pack://application:,,,/",
"/pages/#Pericles Light"));
```

## 在应用程序中枚举字体

若要在应用程序中将字体作为资源项来枚举，请使用 [GetFontFamilies](#) 或 [GetTypefaces](#) 方法。以下示例演示如何使用 [GetFontFamilies](#) 方法从应用程序字体所在的位置返回 [FontFamily](#) 对象的集合。在本示例中，应用程序包含一个名为“resources”的子目录。

C#

```
foreach (FontFamily fontFamily in Fonts.GetFontFamilies(new
Uri("pack://application:,,,/"), "./resources/"))
{
  // Perform action.
}
```

以下示例演示如何使用 [GetTypefaces](#) 方法从应用程序字体所在的位置返回 [Typeface](#) 对象的集合。在本示例中，应用程序包含一个名为“resources”的子目录。

C#

```
foreach (Typeface typeface in Fonts.GetTypefaces(new Uri("pack://application:,,,/", "./resources/")))
{
    // Perform action.
}
```

## 创建字体资源库

可以创建仅包含字体的纯资源库，这种类型的库项目中没有任何代码。 创建纯资源库是将资源与使用它们的应用程序代码分开的一种常用方法。 这还使得库程序集可以包括在多个应用程序项目中。 以下项目文件示例演示纯资源库项目的关键部分。

XML

```
<PropertyGroup>
    <AssemblyName>FontLibrary</AssemblyName>
    <OutputType>library</OutputType>
    ...
</PropertyGroup>
...
<ItemGroup>
    <Resource Include="Kooten.ttf" />
    <Resource Include="Pesca.ttf" />
</ItemGroup>
```

## 引用资源库中的字体

若要在应用程序中引用资源库中的字体，必须将库程序集的名称作为字体引用的前缀。 在本例中，字体资源程序集是“FontLibrary”。 若要将程序集名称与程序集内的引用分开，请使用“;”字符。 添加后跟字体名引用的“Component”关键字即完成字体库资源的完整引用。 以下代码示例演示如何引用资源库程序集内的字体。

XAML

```
<Run FontFamily="/FontLibrary;Component/#Kootenay" FontSize="36">
    ABCDEFGHIJKLMNOPQRSTUVWXYZ
</Run>
```

### ① 备注

本 SDK 包含一组 OpenType 字体示例，可以在 WPF 应用程序中使用这些字体。 这些字体是在纯资源库中定义的。 有关详细信息，请参阅[示例 OpenType 字体包](#)。

# 字体的使用限制

下表介绍在 WPF 应用程序中对字体进行打包和使用时的一些限制：

- **字体嵌入权限位**：WPF 应用程序不检查或实施任何字体嵌入权限位。有关详细信息，请参阅 [Introduction\\_to\\_Packing\\_Fonts](#) 一节。
- **源字体站点**：WPF 应用程序不允许对 http 或 ftp 统一资源标识符 (URI) 进行字体引用。
- **使用 pack: 表示法的绝对 URI**：WPF 应用程序不允许将“pack:”用作字体的绝对统一资源标识符 URI 引用的一部分，以编程方式创建 [FontFamily](#) 对象。例如，“`pack://application:,,,/resources/#Pericles_Light`”是无效的字体引用。
- **自动嵌入字体**：在设计时，不支持搜索应用程序对字体的使用情况，而且不支持自动在应用程序的资源中嵌入字体。
- **字体子集**：WPF 应用程序不支持为非固定文档创建字体子集。
- 如果存在不正确的引用，应用程序将回退到使用可用字体。

## 另请参阅

- [Typography](#)
- [FontFamily](#)
- [Microsoft Typography: Links, News, and Contacts](#) ( Microsoft 版式：链接、新闻和联系人 )
- [OpenType 规范](#)
- [OpenType 字体功能](#)
- [示例 OpenType 字体包](#)

# 示例 OpenType 字体包

项目 • 2023/02/06

本主题概述随 Windows SDK 一起分发的示例 OpenType 字体。这些示例字体支持可供 Windows Presentation Foundation (WPF) 应用程序使用的扩展的 OpenType 功能。

## OpenType 字体包中的字体

Windows SDK 提供一组可用来创建 Windows Presentation Foundation (WPF) 应用程序的示例 OpenType 字体。这些示例字体在 Ascender Corporation 的许可下提供。它们仅实现由 OpenType 格式所定义的全部功能的一部分。下表列出了示例 OpenType 字体的名称。

名称	文件
Kootenay	Kooten.ttf
Lindsey	Linds.ttf
Miramonte	Miramo.ttf
Miramonte Bold	Miramob.ttf
Pericles	Peric.ttf
Pericles Light	Pericl.ttf
Pescadero	Pesca.ttf
Pescadero Bold	Pescab.ttf

下图展示了示例 OpenType 字体。

Kootenay

Lindsey

Miramonte

**Miramonte Bold**

PERICLES

PERICLES LIGHT

Pescadero

**Pescadero Bold**

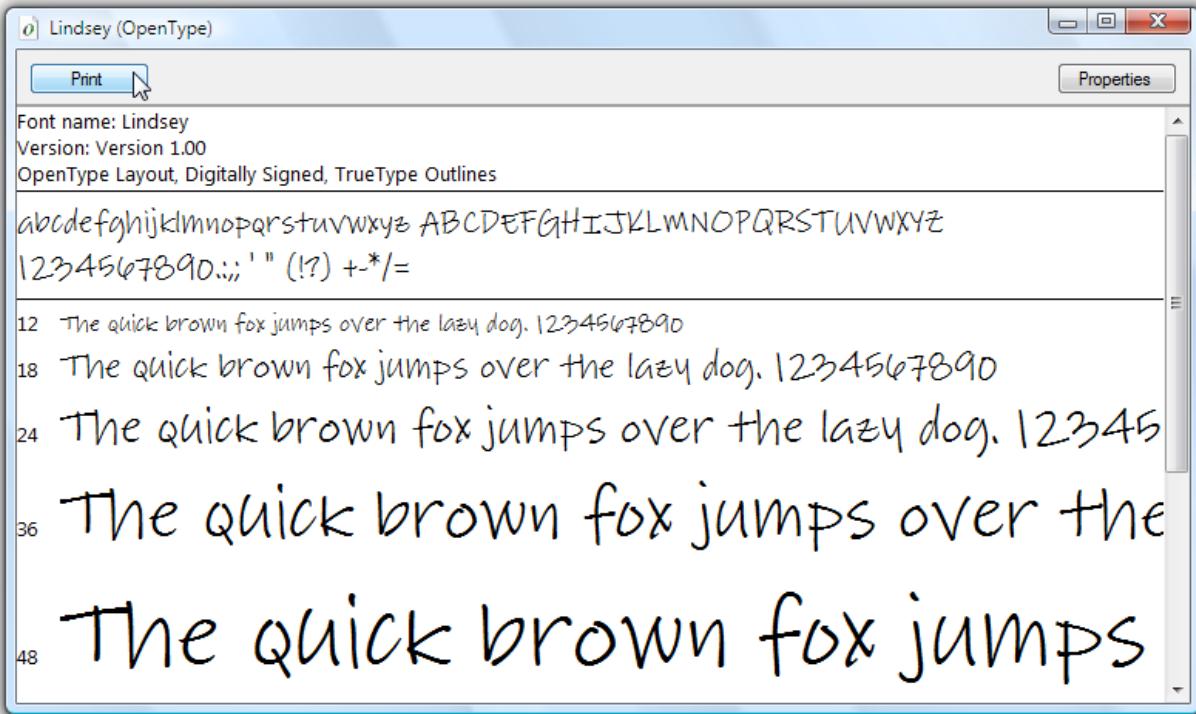
这些示例字体在 Ascender Corporation 的许可下提供。 Ascender 是一家高级字体产品提供商。 若要对示例字体的扩展版本或自定义版本进行授权，请参阅 [Ascender Corporation 的网站](#)。

① 备注

开发人员有责任确保自己具备必要的许可权，可以在应用程序中嵌入相应的字体或者以其他方式重新分布。

## 安装字体

可以选择将示例 OpenType 字体安装到默认 Windows 字体目录：\WINDOWS\Fonts。 使用“字体”控制面板安装字体。 在计算机上安装这些字体后，引用默认 Windows 字体的所有应用程序都可访问这些字体。 可以通过双击字体文件以多种字体大小来显示具有代表性的字符集。 下面的屏幕截图显示 Lindsey 字体文件 Linds.ttf。



显示 Lindsey 字体

## 使用字体

可以通过两种方法在应用程序中使用字体。可以将字体作为项目内容项添加到应用程序中，这些项目内容项不会作为资源嵌入到程序集中。此外，还可以将字体作为嵌入到应用程序程序集文件中的项目资源项添加到应用程序中。有关详细信息，请参阅[将字体与应用程序一起打包](#)。

## 另请参阅

- [Typography](#)
- [OpenType 字体功能](#)
- [将字体与应用程序一起打包](#)

# 字体帮助主题

项目 • 2023/02/06

本节中的主题演示如何使用 Windows Presentation Foundation (WPF) 附带的字体功能。

## 本节内容

[枚举系统字体](#)

[使用 FontSizeConverter 类](#)

## 另请参阅

- [FontStyle](#)
- [SystemFonts](#)
- [WPF 中的文档](#)
- [WPF 中的版式](#)

# 如何：枚举系统字体

项目 • 2023/02/06

## 示例

下面的示例演示如何枚举系统字体集合中的字体。 [SystemFontFamilies](#) 内的每个 [FontFamily](#) 的字体系列名称会作为项添加到组合框。

C#

```
public void FillFontComboBox(ComboBox comboBoxFonts)
{
    // Enumerate the current set of system fonts,
    // and fill the combo box with the names of the fonts.
    foreach (FontFamily fontFamily in Fonts.SystemFontFamilies)
    {
        // FontFamily.Source contains the font family name.
        comboBoxFonts.Items.Add(fontFamily.Source);
    }

    comboBoxFonts.SelectedIndex = 0;
}
```

如果同一体字系列的多个版本位于同一目录中，则 Windows Presentation Foundation (WPF) 字体枚举返回最新版本的字体。如果版本信息不提供分辨率，则会返回具有最新时间戳的字体。如果时间戳信息相同，则返回按字母顺序排列为第一个的字体文件。

# 如何：使用 FontSizeConverter 类

项目 • 2023/02/06

## 示例

此示例演示如何创建 [FontSizeConverter](#) 的实例并用它更改字体大小。

如单独的 Extensible Application Markup Language (XAML) 文件中所述，该示例定义了一个名为 `changeSize` 的自定义方法，它将 [ListBoxItem](#) 的内容转换为 [Double](#) 的实例，随后转换为 [String](#)。此方法将 [ListBoxItem](#) 传递给 [FontSizeConverter](#) 对象，该对象将 [ListBoxItem](#) 的 [Content](#) 转换为 [Double](#) 的实例。然后，此值作为 [TextBlock](#) 元素的 [FontSize](#) 属性的值传回。

此示例还定义称为 `changeFamily` 的另一个自定义方法。此方法将 [ListBoxItem](#) 的 [Content](#) 转换为 [String](#)，然后将该值传递给 [TextBlock](#) 元素的 [FontFamily](#) 属性。

此示例不运行。

C#

```
private void changeSize(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li = ((sender as ListBox).SelectedItem as ListBoxItem);
    FontSizeConverter myFontSizeConverter = new FontSizeConverter();
    text1.FontSize =
    (Double)myFontSizeConverter.ConvertFromString(li.Content.ToString());
}

private void changeFamily(object sender, SelectionChangedEventArgs args)
{
    ListBoxItem li2 = ((sender as ListBox).SelectedItem as ListBoxItem);
    text1.FontFamily = new
    System.Windows.Media.FontFamily(li2.Content.ToString());
}
```

## 另请参阅

- [FontSizeConverter](#)

# 标志符号

项目 • 2023/02/06

字形是要在屏幕上绘制的字符的低级描述。 Windows Presentation Foundation (WPF) 为希望在格式化后截取和保留文本的客户提供对字形的直接访问。

## 本节内容

[GlyphRun 对象和 Glyphs 元素简介](#)

[如何：使用 Glyphs 绘制文本](#)

## 另请参阅

- [GlyphRun](#)
- [DrawText](#)
- [Glyphs](#)
- [WPF 中的文档](#)
- [WPF 中的版式](#)

# GlyphRun 对象和 Glyphs 元素简介

项目 • 2023/02/06

本主题介绍 [GlyphRun 对象](#) 和 [Glyphs 元素](#)。

## GlyphRun 简介

Windows Presentation Foundation (WPF) 提供高级文本支持，包括可直接访问 [Glyphs](#) 的字形级标记，以便客户拦截和保留格式化后的文本。这些功能为以下每种方案中不同的文本呈现要求提供关键支持。

1. 固定格式文档的屏幕显示。
2. 打印方案。
  - Extensible Application Markup Language (XAML) 作为设备打印机语言。
  - Microsoft XPS 文档编写器。
  - 以前的打印机驱动程序，从 Win32 应用程序输出为固定格式。
  - 打印后台处理格式。
3. 固定格式的文档演示，包括以前版本的 Windows 客户端和其他计算设备。

### ① 备注

[Glyphs](#) 和 [GlyphRun](#) 专为固定格式的文档演示和打印方案而设计。有关 UI 方案，请参阅 [WPF 中的版式](#)。

## GlyphRun 对象

[GlyphRun](#) 对象表示单一大小且具有单一呈现样式的单个字体的单个字面的一系列字形。

[GlyphRun](#) 包括字体细节（如字形 [Indices](#)）和各字形位置。它还包括运行生成自的原始 Unicode 代码点、字符到字形缓冲偏移映射信息，以及每字符和每字形标志。

[GlyphRun](#) 具有相应的高级别 [FrameworkElement](#)、[Glyphs](#)。可以在元素树和 XAML 标记中使用 [Glyphs](#) 来表示 [GlyphRun](#) 输出。

## Glyphs 元素

[Glyphs](#) 元素表示 XAML 中 [GlyphRun](#) 的输出。以下标记语法用于描述 [Glyphs](#) 元素。

#### XAML

```
<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString      = "Hello World!"
            Fill               = "Black"
            OriginX            = "100"
            OriginY            = "200"
        />

    </StackPanel>
</Page>
```

以下属性定义对应于示例标记中的前四个特性。

属性	说明
FontUri	指定资源标识符：文件名、Web 统一资源标识符 (URI)，或者应用程序 .exe 或容器中的资源引用。
FontRenderingEmSize	以绘图画面单位指定字号（默认值为 .96 英寸）。
StyleSimulations	指定粗体和斜体样式的标志。
BidiLevel	指定双向布局级别。偶数和零值表示从左到右布局；奇数值表示从右到左布局。

## Indices 属性

[Indices](#) 属性是字形规范的字符串。在一系列字形形成单个群集的情况下，群集中第一个字形的规范之前会跟有一个规范，说明组合了多少个字形和多少个代码点来形成群集。

[Indices](#) 属性在一个字符串中收集以下属性。

- 字形索引
- 字形步进宽度

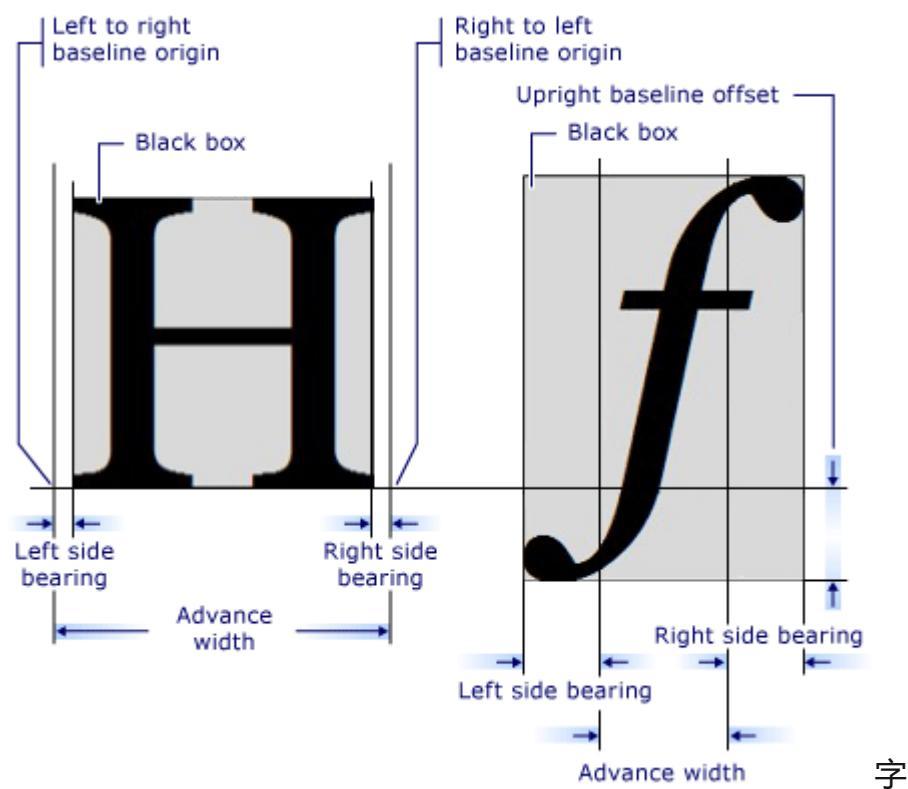
- 组合字形附加矢量
- 从代码点映射到字形的群集
- 字形标志

每个字形规范具有以下形式。

```
[GlyphIndex][,[Advance][,[uOffset][,[vOffset]][,[Flags]]]]]
```

## 字形度量值

每个字形定义了指定它与其他 [Glyphs](#) 的对齐方式的度量值。以下图形定义两种不同字形字符的各种排版品质。



## Glyphs 标记

以下代码示例演示如何使用 XAML 中 [Glyphs](#) 元素的各种属性。

XAML

```
<!-- The example shows how to use different property settings of Glyphs
objects. -->
<Canvas
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Background="PowderBlue"
>
```

```

<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations = "ItalicSimulation"
  UnicodeString    = "Hello World!"
  Fill              = "SteelBlue"
  OriginX           = "50"
  OriginY           = "75"
/>

<!-- "Hello World!" with default kerning -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
  FontRenderingEmSize = "36"
  UnicodeString    = "Hello World!"
  Fill              = "Maroon"
  OriginX           = "50"
  OriginY           = "150"
/>

<!-- "Hello World!" with explicit character widths for proportional font -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
  FontRenderingEmSize = "36"
  UnicodeString    = "Hello World!"
  Indices           = ",80;,80;,80;,80;,80;,80;,80;,80;,80;,80;,80"
  Fill              = "Maroon"
  OriginX           = "50"
  OriginY           = "225"
/>

<!-- "Hello World!" with fixed-width font -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\COUR.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations = "BoldSimulation"
  UnicodeString    = "Hello World!"
  Fill              = "Maroon"
  OriginX           = "50"
  OriginY           = "300"
/>

<!-- "Open file" without "fi" ligature -->
<Glyphs
  FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
  FontRenderingEmSize = "36"
  StyleSimulations = "BoldSimulation"
  UnicodeString    = "Open file"
  Fill              = "SlateGray"
  OriginX           = "400"
  OriginY           = "75"
/>

<!-- "Open file" with "fi" ligature -->

```

```
<Glyphs
    FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
    FontRenderingEmSize = "36"
    StyleSimulations   = "BoldSimulation"
    UnicodeString      = "Open file"
    Indices            = ";;;;;(2:1)191"
    Fill               = "SlateGray"
    OriginX           = "400"
    OriginY           = "150"
/>

</Canvas>
```

## 另请参阅

- [WPF 中的版式](#)
- [WPF 中的文档](#)
- [文本](#)

# 使用 Glyphs 绘制文本

项目 • 2023/02/06

本主题说明如何使用低级别 **Glyphs** 对象来显示 Extensible Application Markup Language (XAML) 文本。

## 示例

以下示例演示如何定义 XAML 中 **Glyphs** 对象的属性。示例假定本地计算机上的 C:\WINDOWS\Fonts 文件夹中安装了 Arial、Courier New 和 Times New Roman 字体。

XAML

```
<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString      = "Hello World!"
            Fill               = "Black"
            OriginX            = "100"
            OriginY            = "200"
        />

    </StackPanel>
</Page>
```

此示例演示如何定义 XAML 中 **Glyphs** 对象的其他属性。

XAML

```
<!-- The example shows how to use different property settings of Glyphs
objects. -->
<Canvas
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Background="PowderBlue"
    >

    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
```

```

        FontRenderingEmSize = "36"
        StyleSimulations   = "ItalicSimulation"
        UnicodeString      = "Hello World! "
        Fill               = "SteelBlue"
        OriginX            = "50"
        OriginY            = "75"
    />

    <!-- "Hello World!" with default kerning -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
        FontRenderingEmSize = "36"
        UnicodeString     = "Hello World! "
        Fill              = "Maroon"
        OriginX           = "50"
        OriginY           = "150"
    />

    <!-- "Hello World!" with explicit character widths for proportional font -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\ARIAL.TTF"
        FontRenderingEmSize = "36"
        UnicodeString     = "Hello World! "
        Indices           = ",80;,80;,80;,80;,80;,80;,80;,80;,80;,80;,80;,80"
        Fill              = "Maroon"
        OriginX           = "50"
        OriginY           = "225"
    />

    <!-- "Hello World!" with fixed-width font -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\COUR.TTF"
        FontRenderingEmSize = "36"
        StyleSimulations   = "BoldSimulation"
        UnicodeString      = "Hello World! "
        Fill               = "Maroon"
        OriginX            = "50"
        OriginY            = "300"
    />

    <!-- "Open file" without "fi" ligature -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
        FontRenderingEmSize = "36"
        StyleSimulations   = "BoldSimulation"
        UnicodeString      = "Open file"
        Fill               = "SlateGray"
        OriginX            = "400"
        OriginY            = "75"
    />

    <!-- "Open file" with "fi" ligature -->
    <Glyphs
        FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
        FontRenderingEmSize = "36"

```

```
StyleSimulations      = "BoldSimulation"
UnicodeString        = "Open file"
Indices              = ";;;;;(2:1)191"
Fill                 = "SlateGray"
OriginX              = "400"
OriginY              = "150"
/>

</Canvas>
```

## 另请参阅

- [WPF 中的版式](#)

# 版式帮助主题

项目 • 2023/02/06

本部分中的主题介绍如何使用 Windows Presentation Foundation (WPF) 支持在应用程序中丰富地呈现文本。

## 本节内容

[创建文本修饰](#)

[指定是否为超链接添加下划线](#)

[向文本应用转换](#)

[向文本应用动画](#)

[创建有阴影的文本](#)

[创建空心文本](#)

[向控件的背景绘制文本](#)

[向视觉对象绘制文本](#)

[在 XAML 中使用特殊字符](#)

## 另请参阅

- [Typography](#)
- [WPF 中的文档](#)
- [OpenType 字体功能](#)

# 如何：创建文本修饰

项目 • 2023/02/06

[TextDecoration](#) 对象是可以添加到文本中的视觉装饰。有四种类型的文本修饰：下划线、基线、删除线和上划线。以下示例显示了文本修饰相对于文本的位置。



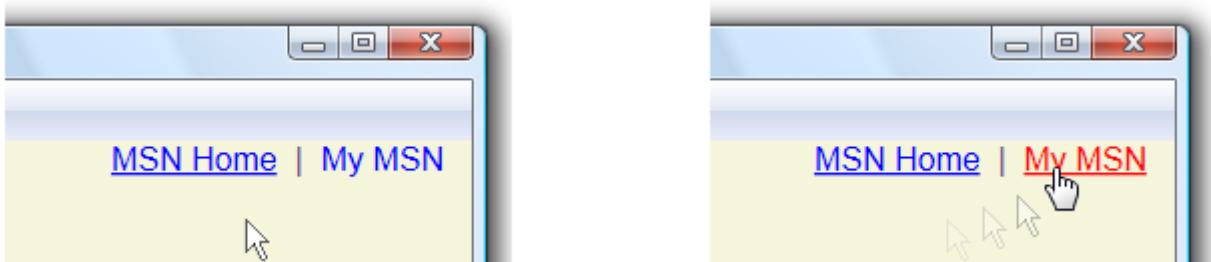
若要向文本添加文本修饰，请创建 [TextDecoration](#) 对象并修改其属性。使用 [Location](#) 属性指定显示文本修饰的位置，例如下划线。使用 [Pen](#) 属性指定文本修饰的外观，例如纯色填充或渐变颜色。如果未为 [Pen](#) 属性指定值，修饰将默认为与文本相同颜色。定义 [TextDecoration](#) 对象后，将其添加到所需文本对象的 [TextDecorations](#) 集合。

以下示例显示了一个文本修饰，该修饰采用线性渐变画笔和虚线笔样式。

**emphasis**

[Hyperlink](#) 对象为内联级流内容元素，允许承载流内容中的超链接。默认情况下，[Hyperlink](#) 使用 [TextDecoration](#) 对象显示下划线。实例化 [TextDecoration](#) 对象可能会占用大量性能，尤其是在拥有许多 [Hyperlink](#) 对象的情况下。如果大范围使用 [Hyperlink](#) 元素，请考虑仅在触发事件时显示下划线，例如 [MouseEnter](#) 事件。

在以下示例中，“我的 MSN”链接的下划线是动态的，仅在触发 [MouseEnter](#) 事件时才显示。



有关详细信息，请参阅[指定是否为超链接添加下划线](#)。

## 示例

在下面的代码示例中，下划线文本修饰使用的是默认字体值。

### C#

```
// Use the default font values for the strikethrough text decoration.  
private void SetDefaultStrikethrough()  
{  
    // Set the underline decoration directly to the text block.  
    TextBlock1.TextDecorations = TextDecorations.Strikethrough;  
}
```

### XAML

```
<!-- Use the default font values for the strikethrough text decoration. -->  
<TextBlock  
    TextDecorations="Strikethrough"  
    FontSize="36" >  
    The quick red fox  
</TextBlock>
```

在下面的代码示例中，下划线文本修饰是使用纯色画笔创建的。

### C#

```
// Use a Red pen for the underline text decoration.  
private void SetRedUnderline()  
{  
    // Create an underline text decoration. Default is underline.  
    TextDecoration myUnderline = new TextDecoration();  
  
    // Create a solid color brush pen for the text decoration.  
    myUnderline.Pen = new Pen(Brushes.Red, 1);  
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended;  
  
    // Set the underline decoration to a TextDecorationCollection and add it  
    // to the text block.  
    TextDecorationCollection myCollection = new TextDecorationCollection();  
    myCollection.Add(myUnderline);  
    TextBlock2.TextDecorations = myCollection;  
}
```

### XAML

```
<!-- Use a Red pen for the underline text decoration -->  
<TextBlock  
    FontSize="36" >  
    jumps over  
<TextBlock.TextDecorations>  
    <TextDecorationCollection>  
        <TextDecoration  
            PenThicknessUnit="FontRecommended">  
            <TextDecoration.Pen>
```

```
<Pen Brush="Red" Thickness="1" />
</TextDecoration.Pen>
</TextDecoration>
</TextDecorationCollection>
</TextBlock.TextDecorations>
</TextBlock>
```

在下面的代码示例中，下划线文本修饰是使用虚线笔的线性渐变画笔创建的。

C#

```
// Use a linear gradient pen for the underline text decoration.
private void SetLinearGradientUnderline()
{
    // Create an underline text decoration. Default is underline.
    TextDecoration myUnderline = new TextDecoration();

    // Create a linear gradient pen for the text decoration.
    Pen myPen = new Pen();
    myPen.Brush = new LinearGradientBrush(Colors.Yellow, Colors.Red, new
    Point(0, 0.5), new Point(1, 0.5));
    myPen.Opacity = 0.5;
    myPen.Thickness = 1.5;
    myPen.DashStyle = DashStyles.Dash;
    myUnderline.Pen = myPen;
    myUnderline.PenThicknessUnit = TextDecorationUnit.FontRecommended;

    // Set the underline decoration to a TextDecorationCollection and add it
    // to the text block.
    TextDecorationCollection myCollection = new TextDecorationCollection();
    myCollection.Add(myUnderline);
    TextBlock3.TextDecorations = myCollection;
}
```

XAML

```
<!-- Use a linear gradient pen for the underline text decoration. -->
<TextBlock FontSize="36">the lazy brown dog.
<TextBlock.TextDecorations>
    <TextDecorationCollection>
        <TextDecoration
            PenThicknessUnit="FontRecommended">
            <TextDecoration.Pen>
                <Pen Thickness="1.5">
                    <Pen.Brush>
                        <LinearGradientBrush Opacity="0.5"
                            StartPoint="0,0.5" EndPoint="1,0.5">
                            <LinearGradientBrush.GradientStops>
                                <GradientStop Color="Yellow" Offset="0" />
                                <GradientStop Color="Red" Offset="1" />
                            </LinearGradientBrush.GradientStops>
                        </LinearGradientBrush>
                    </Pen.Brush>
                </Pen>
            </TextDecoration.Pen>
        </TextDecoration>
    </TextDecorationCollection>
</TextBlock.TextDecorations>
</TextBlock>
```

```
</Pen.Brush>
<Pen.DashStyle>
    <DashStyle Dashes="2"/>
</Pen.DashStyle>
</Pen>
</TextDecoration.Pen>
</TextDecoration>
</TextDecorationCollection>
</TextBlock.TextDecorations>
</TextBlock>
```

## 另请参阅

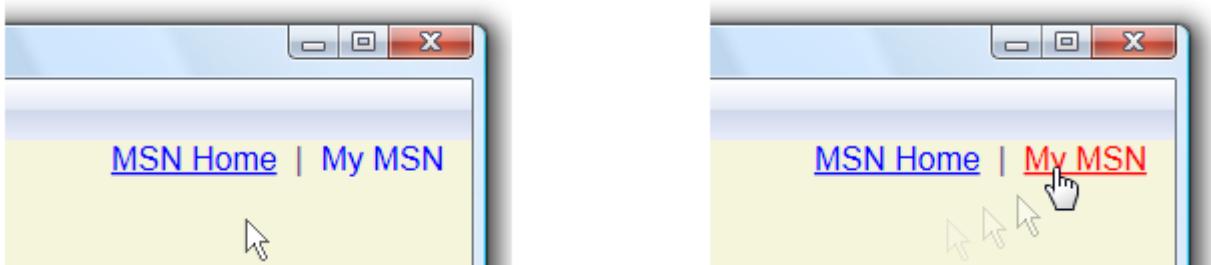
- [TextDecoration](#)
- [Hyperlink](#)
- [指定是否为超链接添加下划线](#)

# 如何：指定是否为超链接添加下划线

项目 • 2023/02/06

Hyperlink 对象为内联级别的流内容元素，允许承载流内容中的超链接。默认情况下，Hyperlink 使用 TextDecoration 对象显示下划线。实例化 TextDecoration 对象可能会占用大量性能，尤其是在拥有许多 Hyperlink 对象的情况下。如果大范围使用 Hyperlink 元素，请考虑仅在触发事件时显示下划线，例如 MouseEnter 事件。

在以下示例中，“我的 MSN”链接的下划线是动态的，即仅在触发 MouseEnter 事件时才显示。



## 示例

以下标记示例演示使用和不使用下划线定义的 Hyperlink：

XAML

```
<!-- Hyperlink with default underline. -->
<Hyperlink NavigateUri="http://www.msn.com">
    MSN Home
</Hyperlink>

<Run Text=" | " />

<!-- Hyperlink with no underline. -->
<Hyperlink Name="myHyperlink" TextDecorations="None"
    MouseEnter="OnMouseEnter"
    MouseLeave="OnMouseLeave"
    NavigateUri="http://www.msn.com">
    My MSN
</Hyperlink>
```

以下代码示例显示如何在 MouseEnter 事件上为 Hyperlink 创建下划线，并在 MouseLeave 事件上将其删除。

C#

```
// Display the underline on only the MouseEnter event.  
private void OnMouseEnter(object sender, EventArgs e)  
{  
    myHyperlink.TextDecorations = TextDecorations.Underline;  
}  
  
// Remove the underline on the MouseLeave event.  
private void OnMouseLeave(object sender, EventArgs e)  
{  
    myHyperlink.TextDecorations = null;  
}
```

## 另请参阅

- [TextDecoration](#)
- [Hyperlink](#)
- [优化 WPF 应用程序性能](#)
- [创建文本修饰](#)

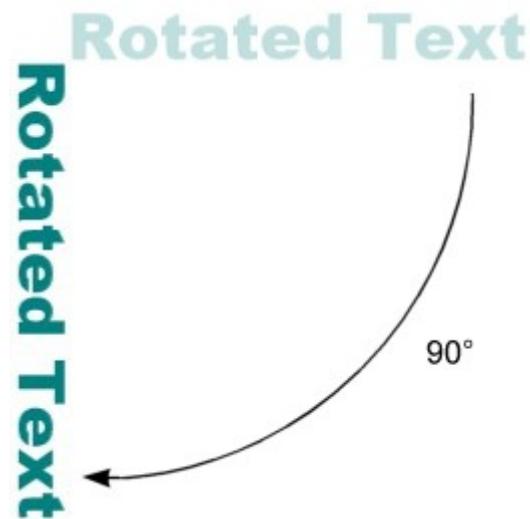
# 如何：向文本应用变换

项目 • 2023/02/06

应用变换可以改变应用程序中文本的显示。 下面的示例使用不同类型的呈现变换来影响 [TextBlock](#) 控件中文本的显示。

## 示例

下面的示例演示了在 x-y 二维平面中文本围绕一个特定点进行旋转。



以下代码示例使用 [RotateTransform](#) 旋转文本。 如果 [Angle](#) 值为 90，则元素会按顺时针方向旋转 90 度。

XAML

```
<!-- Rotate the text 90 degrees using a RotateTransform. -->
<TextBlock FontFamily="Arial Black" FontSize="64" Foreground="Moccasin"
Margin ="80, 10, 0, 0">
    Text Transforms
    <TextBlock.RenderTransform>
        <RotateTransform Angle="90" />
    </TextBlock.RenderTransform>
</TextBlock>
```

下面的示例演示沿 X 轴放大 150% 得到第二行文本，沿 Y 轴放大 150% 得到第三行文本。

## Scaled Text

# Scaled Text

## Scaled Text

下面的代码示例使用 [ScaleTransform](#) 从文本原始大小对文本进行缩放。

### XAML

```
<!-- Scale the text using a ScaleTransform. -->
<TextBlock
    Name="textblockScaleMaster"
    FontSize="32"
    Foreground="SteelBlue"
    Text="Scaled Text"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="0">
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="SteelBlue"
    Text="{Binding Path=Text, ElementName=textblockScaleMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="1">
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="1.5" ScaleY="1.0" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="SteelBlue"
    Text="{Binding Path=Text, ElementName=textblockScaleMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="2">
    <TextBlock.RenderTransform>
        <ScaleTransform ScaleX="1.0" ScaleY="1.5" />
    </TextBlock.RenderTransform>
</TextBlock>
```

### ① 备注

放大文本不同于增大文本字号。字号的计算相互独立，以便针对不同字号提供最佳分辨率。而缩放后的文本将按比例保持原始的文本大小。

以下示例演示沿 X 轴倾斜的文本。

# *Skewed Text*

## Skewed Text

下面的代码示例使用 [SkewTransform](#) 来扭曲文本。 扭曲（也称为倾斜）是一种以非均匀方式拉伸坐标空间的变换。 在本示例中，两个文本字符串沿 x 坐标扭曲了 -30° 和 30°。

### XAML

```
<!-- Skew the text using a SkewTransform. -->
<TextBlock
    Name="textblockSkewMaster"
    FontSize="32"
    FontWeight="Bold"
    Foreground="Maroon"
    Text="Skewed Text"
    Margin="125, 0, 0, 0"
    Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <SkewTransform AngleX="-30" AngleY="0" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="Maroon"
    Text="{Binding Path=Text, ElementName=textblockSkewMaster}"
    Margin="100, 0, 0, 0"
    Grid.Column="0" Grid.Row="1">
    <TextBlock.RenderTransform>
        <SkewTransform AngleX="30" AngleY="0" />
    </TextBlock.RenderTransform>
</TextBlock>
```

下面的示例演示沿 x 轴和 y 轴平移或移动的文本。

# Translated Text

以下代码示例使用 [TranslateTransform](#) 偏移文本。 在本示例中，主要文本下方略微偏移的文本副本营造出了阴影效果。

### XAML

```
<!-- Skew the text using a TranslateTransform. -->
<TextBlock
    FontSize="32"
    FontWeight="Bold"
    Foreground="Black"
    Text="{Binding Path=Text, ElementName=textblockTranslateMaster}"
    Margin="100, 0, 0, 0">
```

```
Grid.Column="0" Grid.Row="0">
<TextBlock.RenderTransform>
  <TranslateTransform X="2" Y="2" />
</TextBlock.RenderTransform>
</TextBlock>
<TextBlock
  Name="textblockTranslateMaster"
  FontSize="32"
  FontWeight="Bold"
  Foreground="Coral"
  Text="Translated Text"
  Margin="100, 0, 0, 0"
  Grid.Column="0" Grid.Row="0"/>
```

## ① 备注

DropShadowBitmapEffect 提供了个多种丰富的功能来产生阴影效果。有关详细信息，请参阅[创建具有阴影的文本](#)。

## 另请参阅

- [向文本应用动画](#)

# 如何：向文本应用动画

项目 • 2023/02/06

动画可以更改应用程序中文本的显示和外观。 下面的示例使用不同类型的动画来影响 [TextBlock](#) 控件中文本的显示。

## 示例

以下示例使用 [DoubleAnimation](#) 来对文本块的宽度设置动画。 宽度值从文本块的宽度更改为 0，持续时间为 10 秒，然后再改回其宽度值并继续。 这种动画会创建一个擦除效果。

XAML

```
<TextBlock
    Name="MyWipedText"
    Margin="20"
    Width="480" Height="100" FontSize="48" FontWeight="Bold"
    Foreground="Maroon">
    This is wiped text

    <!-- Animates the text block's width. -->
    <TextBlock.Triggers>
        <EventTrigger RoutedEvent="TextBlock.Loaded">
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation
                        Storyboard.TargetName="MyWipedText"
                        Storyboard.TargetProperty="(TextBlock.Width)"
                        To="0.0" Duration="0:0:10"
                        AutoReverse="True" RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </TextBlock.Triggers>
</TextBlock>
```

以下示例使用 [DoubleAnimation](#) 来对文本块的不透明度设置动画。 不透明度值从 1.0 更改为 0，持续时间为 5 秒，然后再改回其不透明度值并继续。

XAML

```
<TextBlock
    Name="MyFadingText"
    Margin="20"
    Width="640" Height="100" FontSize="48" FontWeight="Bold"
    Foreground="Maroon">
```

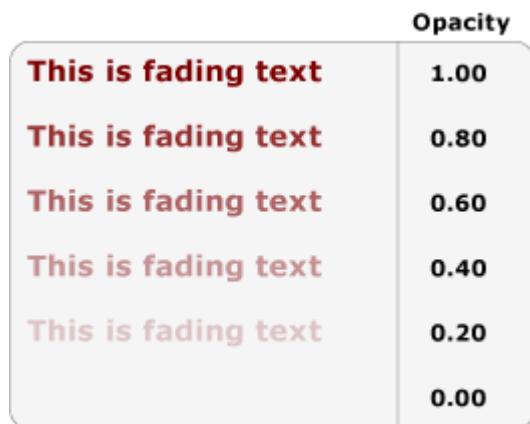
```

This is fading text

<!-- Animates the text block's opacity. -->
<TextBlock.Triggers>
  <EventTrigger RoutedEvent="TextBlock.Loaded">
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation
          Storyboard.TargetName="MyFadingText"
          Storyboard.TargetProperty="(TextBlock.Opacity)"
          From="1.0" To="0.0" Duration="0:0:5"
          AutoReverse="True" RepeatBehavior="Forever" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</TextBlock.Triggers>
</TextBlock>

```

以下示意图显示 `TextBlock` 控件的效果，该控件在 `Duration` 定义的 5 秒间隔期间，将其不透明度从 `1.00` 更改为 `0.00`。



以下示例使用 `ColorAnimation` 来对文本块的前景色设置动画。前景色值从一种颜色更改为另一种颜色，持续时间为 5 秒，然后返回到原来的颜色值并继续。

#### XAML

```

<TextBlock
  Name="MyChangingColorText"
  Margin="20"
  Width="640" Height="100" FontSize="48" FontWeight="Bold">
  This is changing color text
  <TextBlock.Foreground>
    <SolidColorBrush x:Name="MySolidColorBrush" Color="Maroon" />
  </TextBlock.Foreground>

  <!-- Animates the text block's color. -->
  <TextBlock.Triggers>
    <EventTrigger RoutedEvent="TextBlock.Loaded">
      <BeginStoryboard>
        <Storyboard>

```

```
<ColorAnimation  
    Storyboard.TargetName="MySolidColorBrush"  
    Storyboard.TargetProperty="Color"  
    From="DarkOrange" To="SteelBlue" Duration="0:0:5"  
    AutoReverse="True" RepeatBehavior="Forever" />  
</Storyboard>  
</BeginStoryboard>  
</EventTrigger>  
</TextBlock.Triggers>  
</TextBlock>
```

以下示例使用 `DoubleAnimation` 旋转文本块。 文本块旋转一圈，持续时间为 20 秒，然后继续重复该旋转。

#### XAML

```
<TextBlock  
    Name="MyRotatingText"  
    Margin="20"  
    Width="640" Height="100" FontSize="48" FontWeight="Bold" Foreground="Teal"  
    >  
    This is rotating text  
    <TextBlock.RenderTransform>  
        <RotateTransform x:Name="MyRotateTransform" Angle="0" CenterX="230"  
            CenterY="25"/>  
    </TextBlock.RenderTransform>  
  
    <!-- Animates the text block's rotation. -->  
    <TextBlock.Triggers>  
        <EventTrigger RoutedEvent="TextBlock.Loaded">  
            <BeginStoryboard>  
                <Storyboard>  
                    <DoubleAnimation  
                        Storyboard.TargetName="MyRotateTransform"  
                        Storyboard.TargetProperty="(RotateTransform.Angle)"  
                        From="0.0" To="360" Duration="0:0:10"  
                        RepeatBehavior="Forever" />  
                </Storyboard>  
            </BeginStoryboard>  
        </EventTrigger>  
    </TextBlock.Triggers>  
</TextBlock>
```

## 另请参阅

- [动画概述](#)

# 如何：创建具有阴影的文本

项目 • 2023/02/06

本节中的示例演示如何为所显示的文本创建阴影效果。

## 示例

[DropShadowEffect](#) 对象允许为 Windows Presentation Foundation (WPF) 对象创建各种投影效果。以下示例显示了应用于文本的投影效果。在本例中，阴影是柔影，这意味着阴影颜色模糊化。

# Shadow Text

可以通过设置 [ShadowDepth](#) 属性来控制阴影的宽度。值 4.0 指示阴影宽度为 4 像素。可通过修改 [BlurRadius](#) 属性来控制阴影的柔和度或模糊程度。值 0.0 指示无模糊。以下代码示例演示如何创建柔影。

XAML

```
<!-- Soft single shadow. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Teal">
    <TextBlock.Effect>
        <DropShadowEffect
            ShadowDepth="4"
            Direction="330"
            Color="Black"
            Opacity="0.5"
            BlurRadius="4" />
    </TextBlock.Effect>
</TextBlock>
```

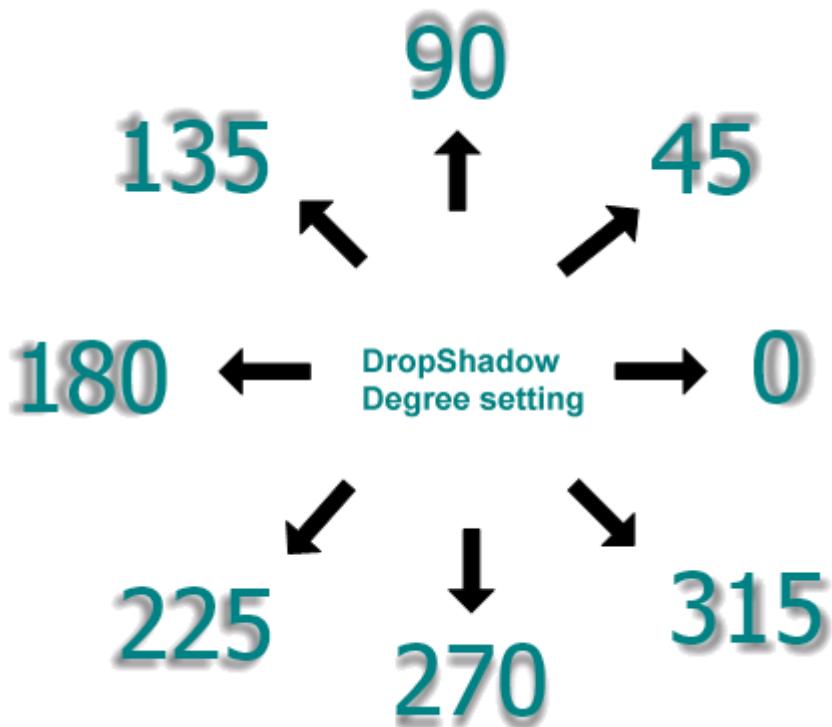
### ① 备注

这些阴影效果不通过 Windows Presentation Foundation (WPF) 文本呈现管道。因此，在使用这些效果时，将禁用 ClearType。

以下示例显示了应用于文本的硬投影效果。在此例中，阴影不模糊。

# Shadow Text

可以通过将 `BlurRadius` 属性设置为 `0.0` ( 指示不使用模糊 ) 来创建硬阴影。可通过修改 `Direction` 属性来控制阴影的方向。将此属性的方向值设置为介于 `0` 和 `360` 之间的度数。下图显示了 `Direction` 属性设置的方向值。



以下代码示例演示如何创建硬阴影。

XAML

```
<!-- Hard single shadow. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Maroon">
    <TextBlock.Effect>
        <DropShadowEffect
            ShadowDepth="6"
            Direction="135"
            Color="Maroon"
            Opacity="0.35"
            BlurRadius="0.0" />
    </TextBlock.Effect>
</TextBlock>
```

## 使用模糊效果

[BlurBitmapEffect](#) 可用于创建可放置在文本对象之后的类似于阴影的效果。应用于文本的模糊位图效果在各个方向上均匀地对文本进行模糊化。

以下示例显示了应用于文本的模糊效果。

# Shadow Text

以下代码示例演示如何创建模糊效果。

XAML

```
<!-- Shadow effect by creating a blur. -->
<TextBlock
    Text="Shadow Text"
    Foreground="Green"
    Grid.Column="0" Grid.Row="0" >
    <TextBlock.Effect>
        <BlurEffect
            Radius="8.0"
            KernelType="Box"/>
    </TextBlock.Effect>
</TextBlock>
<TextBlock
    Text="Shadow Text"
    Foreground="Maroon"
    Grid.Column="0" Grid.Row="0" />
```

# 使用转换变换

[TranslateTransform](#) 可用于创建可放置在文本对象之后的类似于阴影的效果。

以下代码示例使用 [TranslateTransform](#) 偏移文本。在本示例中，主要文本下方略微偏移的文本副本营造出了阴影效果。

# Shadow Text

以下代码示例演示如何创建转换以实现阴影效果。

XAML

```
<!-- Shadow effect by creating a transform. -->
<TextBlock
    Foreground="Black"
    Text="Shadow Text"
```

```
    Grid.Column="0" Grid.Row="0">
    <TextBlock.RenderTransform>
        <TranslateTransform X="3" Y="3" />
    </TextBlock.RenderTransform>
</TextBlock>
<TextBlock
    Foreground="Coral"
    Text="Shadow Text"
    Grid.Column="0" Grid.Row="0">
</TextBlock>
```

# 如何：创建空心文字

项目 • 2023/02/06

大多数情况下，在 Windows Presentation Foundation (WPF) 应用程序中向文本字符串添加装饰时，使用的是离散字符集合或字形集合的文本。例如，可以创建线性渐变画笔并将其应用于 `TextBox` 对象的 `Foreground` 属性。显示或编辑文本框时，线性渐变画笔会自动应用于文本字符串中的当前字符集。

## Spectrum Foreground

但是，你也可以将文本转换为 `Geometry` 对象，从而创建其他类型的视觉富文本。例如，可基于文本字符串的轮廓创建 `Geometry` 对象。

## Spectrum Outline

将文本转换为 `Geometry` 对象时，它不再是字符的集合，也就是说不能修改文本字符串中的字符。但是，可修改已转换文本的笔划和填充属性，以此来影响该文本的外观。笔划是指已转换文本的轮廓；填充是指已转换文本的轮廓的内部区域。

以下示例说明了几种通过修改已转换文本的笔划和填充来创建视觉效果的方法。

*Fancy Outlined Text*

BUTTERFLIES

也可以修改已转换文本的边界框矩形或突出显示。以下示例说明了通过修改已转换文本的笔划和突出显示来创建视觉效果的方法。

WILDFIRE

## 示例

使用 [FormattedText](#) 对象是将文本转换为 [Geometry](#) 对象的关键。 创建此对象后，可以使用 [BuildGeometry](#) 和 [BuildHighlightGeometry](#) 方法将文本转换为 [Geometry](#) 对象。 第一个方法返回格式化文本的几何形状；第二个方法返回格式化文本边界框的几何形状。以下代码示例介绍如何创建 [FormattedText](#) 对象并检索格式化文本及其边界框的几何形状。

C#

```
/// <summary>
/// Create the outline geometry based on the formatted text.
/// </summary>
public void CreateText()
{
    System.Windows.FontStyle fontStyle = FontStyles.Normal;
    FontWeight fontWeight = FontWeights.Medium;

    if (Bold == true) fontWeight = FontWeights.Bold;
    if (Italic == true) fontStyle = FontStyles.Italic;

    // Create the formatted text based on the properties set.
    FormattedText formattedText = new FormattedText(
        Text,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface(
            Font,
            fontStyle,
            fontWeight,
            FontStretches.Normal),
        FontSize,
        System.Windows.Media.Brushes.Black // This brush does not matter
since we use the geometry of the text.
    );

    // Build the geometry object that represents the text.
    _textGeometry = formattedText.BuildGeometry(new System.Windows.Point(0,
0));

    // Build the geometry object that represents the text highlight.
    if (Highlight == true)
    {
        _textHighLightGeometry = formattedText.BuildHighlightGeometry(new
System.Windows.Point(0, 0));
    }
}
```

为显示检索到的 [Geometry](#) 对象，需访问显示已转换文本的对象的 [DrawingContext](#)。 在这些代码示例中，通过创建从支持用户定义呈现的类派生的自定义控件对象来实现此访问。

若要在自定义控件中显示 Geometry 对象，请重写 [OnRender](#) 方法。重写的方法应使用 [DrawGeometry](#) 方法来绘制 Geometry 对象。

C#

```
/// <summary>
/// OnRender override draws the geometry of the text and optional highlight.
/// </summary>
/// <param name="drawingContext">Drawing context of the OutlineText control.
</param>
protected override void OnRender(DrawingContext drawingContext)
{
    // Draw the outline based on the properties that are set.
    drawingContext.DrawGeometry(Fill, new System.Windows.Media.Pen(Stroke,
StrokeThickness), _textGeometry);

    // Draw the text highlight based on the properties that are set.
    if (Highlight == true)
    {
        drawingContext.DrawGeometry(null, new
System.Windows.Media.Pen(Stroke, StrokeThickness), _textHighLightGeometry);
    }
}
```

有关示例自定义用户控件对象的源，请参阅适用于 C# 的 [OutlineTextControl.cs](#) 和适用于 Visual Basic 的 [OutlineTextControl.vb](#)。

## 另请参阅

- [绘制格式化文本](#)

# 如何：将文本绘制到控件的背景上

项目 • 2023/02/06

可以直接将文本绘制到控件的背景上，方法是将文本字符串转换为 `FormattedText` 对象，然后将该对象绘制到控件的 `DrawingContext`。还可以使用这种方法绘制到派生自 `Panel` 的对象（如 `Canvas` 和 `StackPanel`）的背景上。



具有自定义文本背景的控件的示例

## 示例

若要绘制到某个控件的背景上，请新建一个新的 `DrawingBrush` 对象，并将转换后的文本绘制到该对象的 `DrawingContext`。然后，将新的 `DrawingBrush` 分配给控件的背景属性。

以下代码示例演示如何创建 `FormattedText` 对象并绘制到 `Label` 和 `Button` 对象的背景上。

C#

```
// Handle the WindowLoaded event for the window.
private void WindowLoaded(object sender, EventArgs e)
{
    // Update the background property of the label and button.
    myLabel.Background = new DrawingBrush(DrawMyText("My Custom Label"));
    myButton.Background = new DrawingBrush(DrawMyText("Display Text"));
}

// Convert the text string to a geometry and draw it to the control's
// DrawingContext.
private Drawing DrawMyText(string textString)
{
    // Create a new DrawingGroup of the control.
    DrawingGroup drawingGroup = new DrawingGroup();

    // Open the DrawingGroup in order to access the DrawingContext.
    using (DrawingContext drawingContext = drawingGroup.Open())
    {
        // Create the formatted text based on the properties set.
        FormattedText formattedText = new FormattedText(
            textString,
            CultureInfo.GetCultureInfo("en-us"),
            FlowDirection.LeftToRight,
            new Typeface("Comic Sans MS Bold"),
            48,
```

```
System.Windows.Media.Brushes.Black // This brush does not matter
since we use the geometry of the text.
);

// Build the geometry object that represents the text.
Geometry textGeometry = formattedText.BuildGeometry(new
System.Windows.Point(20, 0));

// Draw a rounded rectangle under the text that is slightly larger
than the text.

drawingContext.DrawRoundedRectangle(System.Windows.Media.Brushes.PapayaWhip,
null, new Rect(new System.Windows.Size(formattedText.Width + 50,
formattedText.Height + 5)), 5.0, 5.0);

// Draw the outline based on the properties that are set.
drawingContext.DrawGeometry(System.Windows.Media.Brushes.Gold, new
System.Windows.Media.Pen(System.Windows.Media.Brushes.Maroon, 1.5),
textGeometry);

// Return the updated DrawingGroup content to be used by the
control.
return drawingGroup;
}
}
```

## 另请参阅

- [FormattedText](#)
- [绘制格式化文本](#)

# 如何：将文本绘制到 Visual 中

项目 • 2023/02/06

以下示例演示如何使用 [DrawingContext](#) 对象将文本绘制到某个 [DrawingVisual](#)。通过调用 [RenderOpen](#) 对象的 [DrawingVisual](#) 方法来返回绘图上下文。你可以将图形和文本绘制到绘图上下文中。

若要将文本绘制到绘图上下文中，请使用 [DrawText](#) 对象的 [DrawingContext](#) 方法。完成将内容绘制到绘图上下文中这一步操作后，调用 [Close](#) 方法来关闭绘图上下文并保存内容。

## 示例

C#

```
// Create a DrawingVisual that contains text.
private DrawingVisual CreateDrawingVisualText()
{
    // Create an instance of a DrawingVisual.
    DrawingVisual drawingVisual = new DrawingVisual();

    // Retrieve the DrawingContext from the DrawingVisual.
    DrawingContext drawingContext = drawingVisual.RenderOpen();

    // Draw a formatted text string into the DrawingContext.
    drawingContext.DrawText(
        new FormattedText("Click Me!",
            CultureInfo.GetCultureInfo("en-us"),
            FlowDirection.LeftToRight,
            new Typeface("Verdana"),
            36, System.Windows.Media.Brushes.Black),
        new System.Windows.Point(200, 116));

    // Close the DrawingContext to persist changes to the DrawingVisual.
    drawingContext.Close();

    return drawingVisual;
}
```

### ① 备注

有关从中提取上述代码示例的完整代码示例，请参阅[使用 DrawingVisual 的命中测试示例](#)。

# 如何：在 XAML 中使用特殊字符

项目 • 2023/02/06

Visual Studio 中创建的标记文件将自动保存为 Unicode UTF-8 文件格式，这意味着大部分特殊字符（如重音符号）已正确编码。但是，有一组常用特殊字符的处理方式不同。这些特殊字符采用[万维网联合会 \(W3C\) XML 标准](#)进行编码。

下表显示这组特殊字符的编码语法：

字符	语法	说明
<	&lt;	小于符号。
>	&gt;	大于符号。
&	&amp;	& 符号。
"	&quot;	双引号。
'	&apos;	单引号。

## ① 备注

如果使用文本编辑器（如 Windows 记事本）创建标记文件，必须将文件保存为 Unicode UTF-8 文件格式才能保留所有已编码的特殊符号。

以下示例演示创建标记时如何在文本中使用特殊字符。

## 示例

XAML

```
<!-- Display special characters that require special encoding: < > & " -->
<TextBlock>
    &lt;      <!-- Less than symbol -->
    &gt;      <!-- Greater than symbol -->
    &amp;     <!-- Ampersand symbol -->
    &quot;    <!-- Double quote symbol -->
</TextBlock>

<!-- Display miscellaneous special characters -->
<TextBlock>
    Cæsar   <!-- AE diphthong symbol -->
    © 2006   <!-- Copyright symbol -->
    Español <!-- Tilde symbol -->
```

```
¥      <!-- Yen symbol -->  
</TextBlock>
```

# 打印和打印系统管理

项目 • 2023/02/06

Windows Vista 和 Microsoft .NET Framework 引入了一种新的打印路径（Microsoft Windows 图形设备接口 (GDI) 打印的替代方案）和一系列广泛扩展的打印系统管理 API。

## 本节内容

### 打印概述

讨论新的打印路径和 API。

### 操作指南主题

一系列文章，介绍如何使用新的打印路径和 API。

## 另请参阅

- [System.Printing](#)
- [System.Printing.IndexedProperties](#)
- [System.Printing.Interop](#)
- [WPF 中的文档](#)
- [XPS 文档](#)

# 打印概述

项目 • 2023/02/06

借助 Microsoft .NET Framework，使用 Windows Presentation Foundation 的应用程序开发人员 (WPF) 具有一组丰富的新打印和打印系统管理 API。对于 Windows Vista，其中一些打印系统增强功能也可供创建 Windows 窗体应用程序的开发人员和使用非托管代码的开发人员使用。此新功能的核心是新的 XML 纸张规范 (XPS) 文件格式和 XPS 打印路径。

本主题包含以下各节：

## 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、开放打包约定 (OPC) 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 的其他信息，请参阅 [XPS 文档](#)。

[以编程方式打印 XPS 文件](#)中演示了几种使用 WPF 打印基于 XPS 的内容的技术。在查看本主题中的内容时，参考这些示例会很有帮助。（非托管代码开发人员应参阅 [MXDC\\_ESCAPE 函数](#) 的文档。Windows 窗体开发人员必须使用 [System.Drawing.Printing](#) 命名空间中的 API，该命名空间不支持完整的 XPS 打印路径，但支持混合 GDI 到 XPS 打印路径。请参阅下面的“打印路径体系结构”。）

## XPS 打印路径

XML 纸张规范 (XPS) 打印路径是一项新的 Windows 功能，它重新定义了在 Windows 应用程序中处理打印的方式。因为 XPS 可以替代文档表示语言（如 RTF）、后台打印程序格式（如 WMF）和页面描述语言（如 PCL 或 Postscript）；新的打印路径保持从应用程序发布到打印驱动程序或设备中的最终处理的 XPS 格式。

XPS 打印路径基于 XPS 打印机驱动程序模型 (XPSDrv) 构建，它为开发人员提供了多项优势，例如“所见即所得”(WYSIWYG) 打印、改进的颜色支持以及显着改进的打印性能。（有关 XPSDrv 的详细信息，请参阅 [Windows 驱动程序开发工具包](#)。）

XPS 文档的后台打印程序的操作与以前版本的 Windows 基本相同。但是，除了现有的 GDI 打印路径之外，它已得到增强以支持 XPS 打印路径。新的打印路径本机使用 XPS 后台打印文件。虽然为以前版本的 Windows 编写的用户模式打印机驱动程序将继续工作，但需要 XPS 打印机驱动程序 (XPSDrv) 才能使用 XPS 打印路径。

XPS 打印路径具有巨大优势，其中包括：

- WYSIWYG 打印支持
- 对高级颜色配置文件的本机支持（包括 32 位/通道 (bpc)、CMYK、已命名的颜色、n 墨迹）以及对透明和渐变的本机支持。
- 改进了 .NET Framework 和基于 Win32 的应用程序的打印性能。
- 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 以及用于用户界面、配置和作业提交的单一入口点。对于高级场景，完全添加了对 UI 的额外支持）、同步或异步打印以及批量打印功能。这两个选项在完全或部分信任模式下都提供打印支持。

XPS 的设计考虑了可扩展性。通过使用可扩展性框架，可以以模块化方式将特性和功能添加到 XPS。扩展性功能包括：

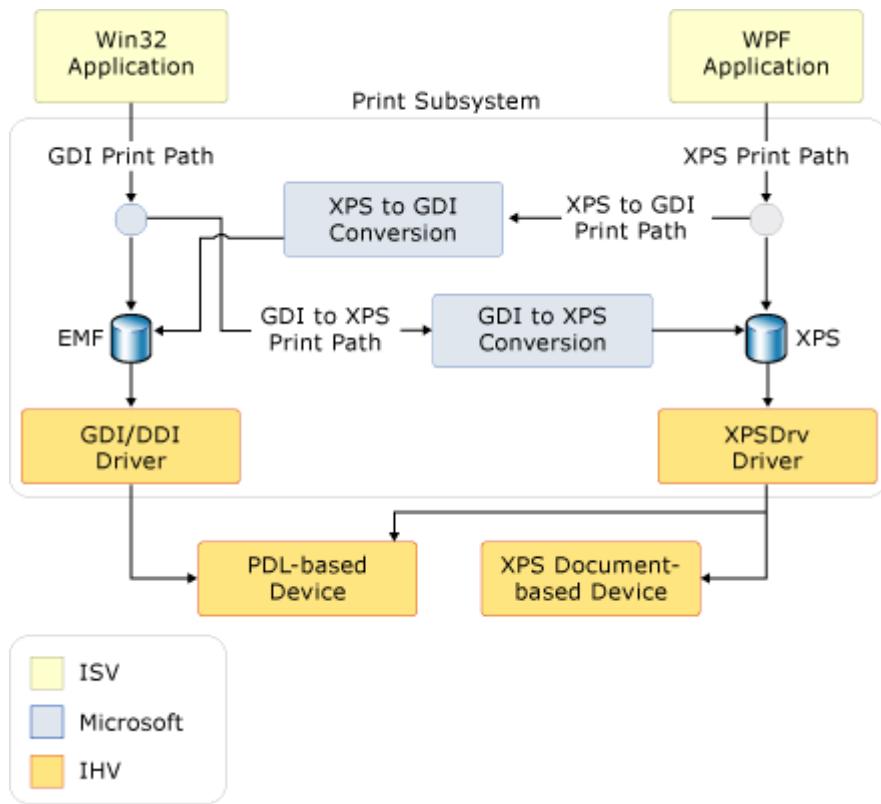
- 打印架构。公共架构将定期进行更新，并可以实现设备功能的迅速扩展。（请参阅下面的 [PrintTicket 和 PrintCapabilities](#)。）
- 可扩展筛选器管道。XPS 打印机驱动程序 (XPSDrv) 过滤器管道旨在实现 XPS 文档的直接和可扩展打印。有关详细信息，请参阅 [XPSDrv 打印机驱动程序](#)。

## 打印路径体系结构

虽然 Win32 和 .NET Framework 应用程序都支持 XPS，但 Win32 和 Windows 窗体应用程序使用 GDI 到 XPS 的转换，以便为 XPS 打印机驱动程序 (XPSDrv) 创建 XPS 格式的内容。这些应用程序不需要使用 XPS 打印路径，并且可以继续使用基于增强型元文件 (EMF) 的打印。但是，大多数 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。

为了使 Win32 和 Windows 窗体应用程序能够使用基于 XPSDrv 的打印机，XPS 打印机驱动程序 (XPSDrv) 支持将 GDI 转换为 XPS 格式。XPSDrv 模型还提供了 XPS 到 GDI 格式的转换器，以便 Win32 应用程序可以打印 XPS 文档。对于 WPF 应用程序，只要写入操作的目标打印队列没有 XPSDrv 驱动程序，就会由 [XpsDocumentWriter](#) 类的 [Write](#) 和 [WriteAsync](#) 方法自动完成 XPS 到 GDI 格式的转换。（Windows 窗体应用程序无法打印 XPS 文档）

下图描述了打印子系统并定义了 Microsoft 提供的部分，以及软件和硬件供应商定义的部分：



## 基本 XPS 打印

WPF 定义了基本 API 和高级 API。对于那些不需要大量打印定制或访问完整 XPS 功能集的应用程序，可以使用基本打印支持。基本打印支持通过打印对话框控件公开，该控件需要最少的配置并具有熟悉的 UI。使用这种简化的打印模型时可以使用许多 XPS 功能。

### PrintDialog

[System.Windows.Controls.PrintDialog](#) 控件为 UI、配置和 XPS 作业提交提供了一个入口点。若要了解如何实例化和使用该控件，请参阅[调用打印对话框](#)。

## 高级 XPS 打印

要访问完整的 XPS 功能集，必须使用高级打印 API。下面更详细地描述了几个相关的 API。有关 XPS 打印路径 API 的完整列表，请参阅 [System.Windows.Xps](#) 和 [System.Printing](#) 命名空间参考。

### PrintTicket 和 PrintCapabilities

[PrintTicket](#) 和 [PrintCapabilities](#) 类是高级 XPS 功能的基础。这两种类型的对象都是面向打印的功能（如排序规则、双面打印、装订等）的 XML 格式结构。这些结构由打印架构定义。[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。

下面的示例演示如何使用代码查询打印机的 PrintCapabilities 和创建 PrintTicket。

C#

```
// ----- GetPrintTicketFromPrinter -----
/// <summary>
///   Returns a PrintTicket based on the current default printer.</summary>
/// <returns>
///   A PrintTicket for the current local default printer.</returns>
private PrintTicket GetPrintTicketFromPrinter()
{
    PrintQueue printQueue = null;

    LocalPrintServer localPrintServer = new LocalPrintServer();

    // Retrieving collection of local printer on user machine
    PrintQueueCollection localPrinterCollection =
        localPrintServer.GetPrintQueues();

    System.Collections.IEnumerator localPrinterEnumerator =
        localPrinterCollection.GetEnumerator();

    if (localPrinterEnumerator.MoveNext())
    {
        // Get PrintQueue from first available printer
        printQueue = (PrintQueue)localPrinterEnumerator.Current;
    }
    else
    {
        // No printer exist, return null PrintTicket
        return null;
    }

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

    PrintCapabilities printCapabilites = printQueue.GetPrintCapabilities();

    // Modify PrintTicket
    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;
    }

    return printTicket;
}// end:GetPrintTicketFromPrinter()

```

## PrintServer 和 PrintQueue

[PrintServer](#) 类表示一个网络打印服务器，而 [PrintQueue](#) 类则表示一台打印机以及与其关联的输出作业队列。结合使用这些，可以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](#)。 [AddJob](#) 方法用于将新的打印作业插入队列。

下面的示例演示如何使用代码创建 [LocalPrintServer](#) 和访问其默认的 [PrintQueue](#)。

C#

```

// ----- GetPrintXpsDocumentWriter() -----
/// <summary>
/// Returns an XpsDocumentWriter for the default print queue.</summary>
/// <returns>
/// An XpsDocumentWriter for the default print queue.</returns>
private XpsDocumentWriter GetPrintXpsDocumentWriter()
{
    // Create a local print server
    LocalPrintServer ps = new LocalPrintServer();

    // Get the default print queue
    PrintQueue pq = ps.DefaultPrintQueue;

    // Get an XpsDocumentWriter for the default print queue
    XpsDocumentWriter xpsdw = PrintQueue.CreateXpsDocumentWriter(pq);
    return xpsdw;
}// end:GetPrintXpsDocumentWriter()

```

## XpsDocumentWriter

[XpsDocumentWriter](#) 及其许多 [Write](#) 和 [WriteAsync](#) 方法用于将 XPS 文档写入 [PrintQueue](#)。例如，[Write\(FixedPage, PrintTicket\)](#) 方法用于同步输出 XPS 文档和 [PrintTicket](#)。[WriteAsync\(FixedDocument, PrintTicket\)](#) 方法用于异步输出 XPS 文档和 [PrintTicket](#)。

下面的示例演示如何使用代码创建 [XpsDocumentWriter](#)。

C#

```
// ----- GetPrintXpsDocumentWriter() -----
/// <summary>
///   Returns an XpsDocumentWriter for the default print queue.</summary>
/// <returns>
///   An XpsDocumentWriter for the default print queue.</returns>
private XpsDocumentWriter GetPrintXpsDocumentWriter()
{
    // Create a local print server
    LocalPrintServer ps = new LocalPrintServer();

    // Get the default print queue
    PrintQueue pq = ps.DefaultPrintQueue;

    // Get an XpsDocumentWriter for the default print queue
    XpsDocumentWriter xpsdw = PrintQueue.CreateXpsDocumentWriter(pq);
    return xpsdw;
}// end:GetPrintXpsDocumentWriter()
```

AddJob 方法还提供打印方法。请参阅[以编程方式打印 XPS 文件](#)，以获取详细信息。

## GDI 打印路径

虽然 WPF 应用程序本身支持 XPS 打印路径，但 Win32 和 Windows 窗体应用程序也可以利用一些 XPS 功能。XPS 打印机驱动程序 (XPSDrv) 可以将基于 GDI 的输出转换为 XPS 格式。对于高级方案，使用 [Microsoft XPS 文档转换器 \(MXDC\)](#) 时支持自定义内容转换。同样，WPF 应用程序也可以通过调用 [XpsDocumentWriter](#) 类的 [Write](#) 或 [WriteAsync](#) 方法之一并将非 XpsDrv 打印机指定为目标打印队列来输出到 GDI 打印路径。

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 转换选项的其他参考资料，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 用作本机打印后台处理格式，从而提高打印后台处理程序的效率。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。

EMF 是一种封闭格式，它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。与 EMF 不同的是，XPS 后台打印格式呈现实际的文档，而无需在输出到基于 XPS 的打印机驱动程序 (XPSDrv) 时进行进一步的转译。这些驱动程序可以用这种格式直接对数据进

行操作。此功能消除了在使用 EMF 文件和基于 GDI 的打印驱动程序时所需的数据和颜色空间转换。

使用以 XPS 打印机驱动程序 (XPSDrv) 为目标的 XPS 文档时，后台打印文件的大小通常会比它们的 EMF 等效文件小；但也有例外：

- 相当复杂、分为多层或者编写效率低下的向量图形可能比同一图形的位图图形更大。
- 出于屏幕显示的目的，XPS 文件中嵌入了设备字体以及基于计算机的字体，而 GDI 打印后台文件未嵌入设备字体。这两种字体都划分了子集（请参见下面的内容），而且打印机驱动程序可以在将文件传输给打印机之前删除这些设备字体。

可通过几种机制来减小后台打印文件的大小：

- **字体子集划分。** 只有在实际文档中使用的字符才会存储在 XPS 文件中。
- **高级图形支持。** 对透明度和渐变基元的本机支持避免了 XPS 文档中内容的光栅化。
- **公共资源的识别。** 将多次使用的资源（如表示公司徽标的图像）视为共享资源，并且只加载一次。
- **ZIP 压缩。** 所有 XPS 文档都使用 ZIP 压缩。

## 另请参阅

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

# 打印帮助主题

项目 • 2023/02/06

本节中的主题演示如何使用 Windows Presentation Foundation (WPF) 附带的打印和打印系统管理功能，以及新的 XML 纸张规范 (XPS) 打印路径。

## 本节内容

### [调用打印对话框](#)

说明如何使用 XAML 标记声明 Microsoft Windows 打印对话框对象，以及如何使用代码从 Windows Presentation Foundation (WPF) 应用程序中调用对话框。

### [克隆打印机](#)

说明如何安装其属性与现有打印队列的属性完全相同的另一个打印队列。

### [诊断有问题的打印作业](#)

说明如何使用打印队列和打印作业的属性诊断未打印的打印作业。

### [确定此时是否可以打印一项打印作业](#)

说明如何使用打印队列和打印作业的属性以编程方式确定一天中的什么时间可以打印作业。

### [枚举打印队列的子集](#)

说明如何生成具有特定特征的打印机列表。

### [在不使用反射的情况下获取打印系统对象属性](#)

说明如何在运行时发现打印系统对象的属性及其类型。

### [以编程方式打印 XPS 文件](#)

说明如何快速打印 XML 纸张规范 (XPS) 文件，而无需使用用户界面 (UI)。

### [远程调查打印机的状态](#)

说明如何创建实用工具以调查打印机，从而发现遇到卡纸或其他问题的打印机。

### [验证和合并 PrintTicket](#)

说明如何检查打印票证是否有效，以及它是否未请求打印机不支持的任何内容。

## 另请参阅

- [System.Printing](#)
- [System.Printing.IndexedProperties](#)
- [System.Printing.Interop](#)

- 打印概述
- WPF 中的文档
- XPS 文档

# 如何：调用打印对话框

项目 • 2023/02/06

若要提供从应用程序进行打印的功能，只需创建并打开 [PrintDialog](#) 对象即可。

## 示例

[PrintDialog](#) 控件为 UI、配置和 XPS 作业提交提供单一入口点。该控件易于使用，可以使用 Extensible Application Markup Language (XAML) 标记或代码进行实例化。下面的示例演示如何在代码中实例化和打开控件，以及如何从中打印。它还演示如何确保对话框将为用户提供设置特定页面范围的选项。示例代码假定 C: 驱动器的根目录中有一个 FixedDocumentSequence.xps 文件。

C#

```
private void InvokePrint(object sender, RoutedEventArgs e)
{
    // Create the print dialog object and set options
    PrintDialog pDialog = new PrintDialog();
    pDialog.PageRangeSelection = PageRangeSelection.AllPages;
    pDialog.UserPageRangeEnabled = true;

    // Display the dialog. This returns true if the user presses the
    // Print button.
    Nullable<Boolean> print = pDialog.ShowDialog();
    if (print == true)
    {
        XpsDocument xpsDocument = new
XpsDocument("C:\\\\FixedDocumentSequence.xps", FileAccess.ReadWrite);
        FixedDocumentSequence fixedDocSeq =
xpsDocument.GetFixedDocumentSequence();
        pDialog.PrintDocument(fixedDocSeq.DocumentPaginator, "Test print
job");
    }
}
```

打开对话框后，用户将能够从计算机上安装的打印机中进行选择。他们还可以选择 [Microsoft XPS 文档编写器](#) 来创建 XML 纸规范 (XPS) 文件，而不是打印。

### ① 备注

本主题中讨论的 WPF 的 `System.Windows.Controls.PrintDialog` 控件不应与 Windows 窗体的 `System.Windows.Forms.PrintDialog` 组件混淆。

严格地说，可在没有打开对话框的情况下使用 [PrintDocument](#) 方法。从这个意义上说，该控件可用作不可见的打印组件。但是出于性能原因，最好使用 [AddJob](#) 方法或 [XpsDocumentWriter](#) 的 [Write](#) 和 [WriteAsync](#) 方法之一。有关此内容的详细信息，请参阅 [以编程方式打印 XPS 文件](#)。

## 另请参阅

- [PrintDialog](#)
- [WPF 中的文档](#)
- [打印概述](#)
- [Microsoft XPS 文档编写器](#)

# 如何：克隆打印机

项目 • 2023/02/06

大多数企业有时会购买多台同一型号的打印机。通常，这些打印机都安装了几乎相同的配置设置。安装每台打印机既费时又容易出错。使用 Microsoft .NET Framework 公开的 `System.Printing.IndexedProperties` 命名空间和 `InstallPrintQueue` 类可以立即安装从现有打印队列克隆的任意数量的附加打印队列。

## 示例

在下面的示例中，从现有打印队列克隆第二个打印队列。第二个队列与第一个队列的不同之处仅在于名称、位置、端口和共享状态。执行此操作的主要步骤如下。

1. 为将要克隆的现有打印机创建 `PrintQueue` 对象。
2. 从 `PrintQueue` 的 `PropertiesCollection` 中创建 `PrintPropertyDictionary`。此字典中每个条目的 `Value` 属性是从 `PrintProperty` 派生的类型之一的对象。可通过两种方式在此字典中设置条目的值。

- 使用字典的 `Remove` 和 `Add` 方法删除条目，然后使用所需的值重新添加它。
- 使用字典的  `SetProperty` 方法。

下面的示例对这两种方式进行说明。

3. 创建 `PrintBooleanProperty` 对象并将其 `Name` 设置为“`IsShared`”以及将其 `Value` 设置为 `true`。
4. 使用 `PrintBooleanProperty` 对象作为 `PrintPropertyDictionary` 的“`IsShared`”条目的值。
5. 创建 `PrintStringProperty` 对象并将其 `Name` 设置为“`ShareName`”以及将其 `Value` 设置为相应的 `String`。
6. 使用 `PrintStringProperty` 对象作为 `PrintPropertyDictionary` 的“`ShareName`”条目的值。
7. 创建另一个 `PrintStringProperty` 对象并将其 `Name` 设置为“`Location`”以及将其 `Value` 设置为相应的 `String`。
8. 使用第二个 `PrintStringProperty` 对象作为 `PrintPropertyDictionary` 的“`Location`”条目的值。
9. 创建一个 `String` 数组。每一项都是服务器上的端口的名称。

## 10. 使用 `InstallPrintQueue` 安装具有新值的新打印机。

下面是一个示例。

C#

```
LocalPrintServer myLocalPrintServer = new LocalPrintServer(PrintSystemDesiredAccess.AdministrateServer);
PrintQueue sourcePrintQueue = myLocalPrintServer.DefaultPrintQueue;
PrintPropertyDictionary myPrintProperties =
sourcePrintQueue.PropertiesCollection;

// Share the new printer using Remove/Add methods
PrintBooleanProperty shared = new PrintBooleanProperty("IsShared", true);
myPrintProperties.Remove("IsShared");
myPrintProperties.Add("IsShared", shared);

// Give the new printer its share name using SetProperty method
PrintStringProperty theShareName = new PrintStringProperty("ShareName",
"\\" + "Son of " + sourcePrintQueue.Name + "\\");
myPrintProperties SetProperty("ShareName", theShareName);

// Specify the physical location of the new printer using Remove/Add methods
PrintStringProperty theLocation = new PrintStringProperty("Location", "the
supply room");
myPrintProperties.Remove("Location");
myPrintProperties.Add("Location", theLocation);

// Specify the port for the new printer
String[] port = new String[] { "COM1:" };

// Install the new printer on the local print server
PrintQueue clonedPrinter = myLocalPrintServer.InstallPrintQueue("My clone of
" + sourcePrintQueue.Name, "Xerox WCP 35 PS", port, "WinPrint",
myPrintProperties);
myLocalPrintServer.Commit();

// Report outcome
Console.WriteLine("{0} in {1} has been installed and shared as {2}",
clonedPrinter.Name, clonedPrinter.Location, clonedPrinter.ShareName);
Console.WriteLine("Press Return to continue ...");
Console.ReadLine();
```

## 另请参阅

- [System.Printing.IndexedProperties](#)
- [PrintPropertyDictionary](#)
- [LocalPrintServer](#)
- [PrintQueue](#)
- [DictionaryEntry](#)

- WPF 中的文档
- 打印概述

# 如何：诊断有问题的打印作业

项目 • 2023/02/06

网络管理员经常接收到有关打印作业无法打印或打印速度慢的用户投诉。 Microsoft .NET Framework 的 API 中公开的丰富打印作业属性集提供对打印作业执行快速远程诊断的方法。

## 示例

以下是创建此类实用程序的主要步骤。

1. 标识用户投诉的打印作业。 用户通常无法准确完成此操作。 他们可能不知道打印服务器或打印机的名称。 他们描述打印机的位置可能与设置其 [Location](#) 属性时所用的术语不同。 这样的话，一个好办法是生成用户当前所提交作业的列表。 如果存在多个作业，则可通过用户和打印系统管理员之间的通信来查明出现问题的作业。 子步骤如下。
  - a. 获取所有打印服务器的列表。
  - b. 循环访问服务器以查询其打印队列。
  - c. 在每一轮服务器循环访问过程中，循环访问所有服务器的队列，以查询其作业
  - d. 在每一轮服务器循环访问过程中，循环访问其作业，并收集与投诉用户已提交的作业相关的标识信息。
2. 当已识别有问题的打印作业时，检查相关属性以查明可能的问题。 例如，作业是否处于错误状态，或服务于队列的打印机是否在打印该作业之前处于脱机状态？

以下代码是一系列代码示例。 第一个代码示例包含针对打印队列的循环访问操作。（上述步骤 1c.）变量 `myPrintQueues` 是 [PrintQueueCollection](#) 当前打印服务器的对象。

代码示例首先使用 [PrintQueue.Refresh](#) 刷新当前打印队列对象。 这可确保该对象的属性能够准确表示它所表示的物理打印机的状态。 然后，该应用程序通过使用 [GetPrintJobInfoCollection](#) 获取打印队列中当前打印作业的集合。

接下来应用程序在 [PrintSystemJobInfo](#) 集合内循环，并将每个 [Submitter](#) 属性与抱怨用户的别名进行比较。 如果属性和别名相匹配，则应用程序会将有关该作业的标识信息添加到将呈现的字符串。（`userName` 和 `jobList` 变量在应用程序早期进行初始化。）

C#

```

foreach (PrintQueue pq in myPrintQueues)
{
    pq.Refresh();
    PrintJobInfoCollection jobs = pq.GetPrintJobInfoCollection();
    foreach (PrintSystemJobInfo job in jobs)
    {
        // Since the user may not be able to articulate which job is
        // problematic,
        // present information about each job the user has submitted.
        if (job.Submitter == userName)
        {
            atLeastOne = true;
            jobList = jobList + "\nServer:" + line;
            jobList = jobList + "\n\tQueue:" + pq.Name;
            jobList = jobList + "\n\tLocation:" + pq.Location;
            jobList = jobList + "\n\t\tJob: " + job.JobName + " ID: " +
            job.JobIdentifier;
        }
    } // end for each print job
} // end for each print queue

```

下代码示例在步骤 2 提取应用程序。 (请参阅 above.) 已识别有问题的作业，应用程序会提示输入将识别它的信息。 它从此信息中创建 [PrintServer](#)、[PrintQueue](#) 和 [PrintSystemJobInfo](#) 对象。

此时，应用程序包含一个分支结构，该结构对应于检查打印作业状态的两种方法：

- 可读取类型为 [PrintJobStatus](#) 的 [JobStatus](#) 属性的标记。
- 可读取每个相关属性，如 [IsBlocked](#) 和 [IsInError](#)。

此示例将演示这两种方法，以便预先提示用户要使用哪种方法，且如果用户想要使用 [JobStatus](#) 属性的标记，则将收到回复“y”。 请参阅以下有关这两种方法的详细信息。 最后，该应用程序使用一种名为 [ReportQueueAndJobAvailability](#) 的方法来报告是否可以在一天的此时打印该作业。 确定此时是否可以打印一项打印作业中就此方法进行了讨论。

C#

```

// When the problematic print job has been identified, enter information
// about it.
Console.Write("\nEnter the print server hosting the job (including leading
slashes \\\\): " +
"\n(press Return for the current computer \\\\{}): ",
Environment.MachineName);
String pServer = Console.ReadLine();
if (pServer == "")
{
    pServer = "\\\\" + Environment.MachineName;
}

```

```

Console.WriteLine("\nEnter the print queue hosting the job: ");
String pQueue = Console.ReadLine();
Console.WriteLine("\nEnter the job ID: ");
Int16 jobID = Convert.ToInt16(Console.ReadLine());

// Create objects to represent the server, queue, and print job.
PrintServer hostingServer = new PrintServer(pServer,
PrintSystemDesiredAccess.AdministrateServer);
PrintQueue hostingQueue = new PrintQueue(hostingServer, pQueue,
PrintSystemDesiredAccess.AdministratePrinter);
PrintSystemJobInfo theJob = hostingQueue.GetJob(jobID);

if (useAttributesResponse == "Y")
{
    TroubleSpotter.SpotTroubleUsingJobAttributes(theJob);
    // TroubleSpotter class is defined in the complete example.
}
else
{
    TroubleSpotter.SpotTroubleUsingProperties(theJob);
}

TroubleSpotter.ReportQueueAndJobAvailability(theJob);

```

若要使用 [JobStatus](#) 属性的标记检查打印作业状态，请检查每个相关标记以查看是否已设置。 检查是否在一组位标志中设置了一个位的标准方法是执行一个逻辑 AND 运算，其中将该组标志作为一个操作数，将标志本身作为另一操作数。 由于该标志本身仅设置一个位，因此逻辑 AND 的结果至多为设置了该相同位。 若要查明事实是否如此，只需将逻辑 AND 的结果与标志本身进行比较。 有关详细信息，请参阅[PrintJobStatus](#)，&[运算符 \(C# 引用\)](#)，和 [FlagsAttribute](#)。

对于已设置了其位的每个属性，代码会将此报告给控制台屏幕，有时还会建议如何响应。  
( 下面讨论了作业或队列暂停时所调用的 [HandlePausedJob](#) 方法。 )

C#

```

// Check for possible trouble states of a print job using the flags of the
JobStatus property
internal static void SpotTroubleUsingJobAttributes(PrintSystemJobInfo
theJob)
{
    if (((theJob.JobStatus & PrintJobStatus.Blocked) ==
PrintJobStatus.Blocked)
    {
        Console.WriteLine("The job is blocked.");
    }
    if (((theJob.JobStatus & PrintJobStatus.Completed) ==
PrintJobStatus.Completed)
        ||
        ((theJob.JobStatus & PrintJobStatus.Printed) ==
PrintJobStatus.Printed))

```

```

{
    Console.WriteLine("The job has finished. Have user recheck all
output bins and be sure the correct printer is being checked.");
}
if (((theJob.JobStatus & PrintJobStatus.Deleted) ==
PrintJobStatus.Deleted)
    ||
    ((theJob.JobStatus & PrintJobStatus.Deleting) ==
PrintJobStatus.Deleting))
{
    Console.WriteLine("The user or someone with administration rights to
the queue has deleted the job. It must be resubmitted.");
}
if ((theJob.JobStatus & PrintJobStatus.Error) == PrintJobStatus.Error)
{
    Console.WriteLine("The job has errored.");
}
if ((theJob.JobStatus & PrintJobStatus.Offline) ==
PrintJobStatus.Offline)
{
    Console.WriteLine("The printer is offline. Have user put it online
with printer front panel.");
}
if ((theJob.JobStatus & PrintJobStatus.PaperOut) ==
PrintJobStatus.PaperOut)
{
    Console.WriteLine("The printer is out of paper of the size required
by the job. Have user add paper.");
}

if (((theJob.JobStatus & PrintJobStatus.Paused) ==
PrintJobStatus.Paused)
    ||
    ((theJob.HostingPrintQueue.QueueStatus & PrintQueueStatus.Paused) ==
PrintQueueStatus.Paused))
{
    HandlePausedJob(theJob);
    //HandlePausedJob is defined in the complete example.
}

if ((theJob.JobStatus & PrintJobStatus.Printing) ==
PrintJobStatus.Printing)
{
    Console.WriteLine("The job is printing now.");
}
if ((theJob.JobStatus & PrintJobStatus.Spooling) ==
PrintJobStatus.Spooling)
{
    Console.WriteLine("The job is spooling now.");
}
if ((theJob.JobStatus & PrintJobStatus.UserIntervention) ==
PrintJobStatus.UserIntervention)
{
    Console.WriteLine("The printer needs human intervention.");
}

```

```
    }
}//end SpotTroubleUsingJobAttributes
```

若要使用单独的属性检查打印作业状态，只需读取每个属性，如果属性为 `true`，则报告给控制台屏幕，并建议一种可能的响应方式。（下面讨论了作业或队列暂停时所调用的 `HandlePausedJob` 方法。）

C#

```
// Check for possible trouble states of a print job using its properties
internal static void SpotTroubleUsingProperties(PrintSystemJobInfo theJob)
{
    if (theJob.IsBlocked)
    {
        Console.WriteLine("The job is blocked.");
    }
    if (theJob.IsCompleted || theJob.IsPrinted)
    {
        Console.WriteLine("The job has finished. Have user recheck all
output bins and be sure the correct printer is being checked.");
    }
    if (theJob.IsDeleted || theJob.IsDeleting)
    {
        Console.WriteLine("The user or someone with administration rights to
the queue has deleted the job. It must be resubmitted.");
    }
    if (theJob.IsInError)
    {
        Console.WriteLine("The job has errored.");
    }
    if (theJob.Offline)
    {
        Console.WriteLine("The printer is offline. Have user put it online
with printer front panel.");
    }
    if (theJob.PaperOut)
    {
        Console.WriteLine("The printer is out of paper of the size required
by the job. Have user add paper.");
    }

    if (theJob.Paused || theJob.HostingPrintQueue.Paused)
    {
        HandlePausedJob(theJob);
        //HandlePausedJob is defined in the complete example.
    }

    if (theJob.Printing)
    {
        Console.WriteLine("The job is printing now.");
    }
    if (theJob.Spooling)
```

```

{
    Console.WriteLine("The job is spooling now.");
}
if (theJob.IsUserInterventionRequired)
{
    Console.WriteLine("The printer needs human intervention.");
}
}//end SpotTroubleUsingProperties

```

使用 `HandlePausedJob` 方法，应用程序的用户能够远程恢复暂停的作业。由于可能存在一个充分的暂停打印队列的理由，因此该方法会首先提示用户是否确实要恢复该作业。如果答案是“y”，则调用 `PrintQueue.Resume` 方法。

接下来，系统会提示用户是否确实要继续执行作业，这么做是考虑到这个作业可能是独立于队列中的其他作业被单独暂停的。(Compare `PrintQueue.IsPaused` 和 `PrintSystemJobInfo.IsPaused`.) 如果答案为“Y”，则 `PrintSystemJobInfo.Resume` 调用;否则 `Cancel` 调用。

C#

```

internal static void HandlePausedJob(PrintSystemJobInfo theJob)
{
    // If there's no good reason for the queue to be paused, resume it and
    // give user choice to resume or cancel the job.
    Console.WriteLine("The user or someone with administrative rights to the
queue" +
        "\nhas paused the job or queue." +
        "\nResume the queue? (Has no effect if queue is not paused.)" +
        "\nEnter \"Y\" to resume, otherwise press return: ");
    String resume = Console.ReadLine();
    if (resume == "Y")
    {
        theJob.HostingPrintQueue.Resume();

        // It is possible the job is also paused. Find out how the user
wants to handle that.
        Console.WriteLine("Does user want to resume print job or cancel it?"
+
            "\nEnter \"Y\" to resume (any other key cancels the print job):");
    };
    String userDecision = Console.ReadLine();
    if (userDecision == "Y")
    {
        theJob.Resume();
    }
    else
    {
        theJob.Cancel();
    }
}//end if the queue should be resumed
}//end HandlePausedJob

```

---

## 另请参阅

- [PrintJobStatus](#)
- [PrintSystemJobInfo](#)
- [FlagsAttribute](#)
- [PrintQueue](#)
- [&运算符 \( C# 引用 \)](#)
- [WPF 中的文档](#)
- [打印概述](#)

# 如何：确定此时是否可以打印一项打印作业

项目 • 2023/02/06

打印队列并非始终每天 24 小时都可用。 打印队列具有开始和结束时间属性，可以设置这些属性以使队列在一天中的某些时间不可用。 例如，此功能可用于在下午 5 点之后保留一台打印机供某个部门专用。 该部门将使用与其他部门不同的队列来维护打印机。 其他部门的队列将设置为在下午 5 点之后不可用，而优先部门的队列可以设置为随时可用。

此外，打印作业本身可以设置为只能在指定的时间范围内打印。

Microsoft .NET Framework 的 API 中公开的 [PrintQueue](#) 和 [PrintSystemJobInfo](#) 类提供了一种远程检查给定打印作业当前是否可以在给定队列上打印的方法。

## 示例

以下是一个可以诊断打印作业问题的示例。

此类函数有两个主要步骤，如下所示。

1. 读取 [PrintQueue](#) 的 [StartTimeOfDay](#) 和 [UntilTimeOfDay](#) 属性以确定当前时间是否在它们之间。
2. 读取 [PrintSystemJobInfo](#) 的 [StartTimeOfDay](#) 和 [UntilTimeOfDay](#) 属性以确定当前时间是否在它们之间。

但是由于这些属性不是 [DateTime](#) 对象这一事实而产生了复杂性。 相反，它们是 [Int32](#) 对象，将一天中的时间表示为自午夜以来的分钟数。 此外，这不是当前时区的午夜，而是 UTC（协调世界时）的午夜。

第一个代码示例演示了静态方法 [ReportQueueAndJobAvailability](#)，它传递了一个 [PrintSystemJobInfo](#) 并调用帮助程序方法来确定作业是否可以在当前时间打印，如果不能，则确定何时可以打印。 请注意，没有将 [PrintQueue](#) 传递给该方法。 这是因为 [PrintSystemJobInfo](#) 在其 [HostingPrintQueue](#) 属性中包含对队列的引用。

从属方法包括重载的 [ReportAvailabilityAtThisTime](#) 方法，该方法可以采用 [PrintQueue](#) 或 [PrintSystemJobInfo](#) 作为参数。 还有一个 [TimeConverter.ConvertToLocalHumanReadableTime](#)。 下面将讨论所有这些方法。

[ReportQueueAndJobAvailability](#) 方法首先检查队列或打印作业此时是否不可用。 如果其中任一项不可用，它将检查队列是否不可用。 如果队列不可用，则该方法将报告此事实以及队列再次可用的时间。 然后它会检查作业，如果作业不可用，它会报告下一个可以

打印的时间跨度。最后，该方法报告作业可以打印的最早时间。即后面两项时间中较晚的一项。

- 打印队列下次可用的时间。
- 打印作业下次可用的时间。

报告一天中的时间时，还会调用 [ToShortTimeString](#) 方法，因为此方法会禁止显示输出中的年、月和日。不能将打印队列或打印作业的可用性限制为特定的年、月或日。

C#

```
internal static void ReportQueueAndJobAvailability(PrintSystemJobInfo
theJob)
{
    if (!(ReportAvailabilityAtThisTime(theJob.HostingPrintQueue) &&
ReportAvailabilityAtThisTime(theJob)))
    {
        if (!ReportAvailabilityAtThisTime(theJob.HostingPrintQueue))
        {
            Console.WriteLine("\nThat queue is not available at this time of
day." +
                "\nJobs in the queue will start printing again at {0}",

TimeConverter.ConvertToLocalHumanReadableTime(theJob.HostingPrintQueue.Start
TimeOfDay).ToString("t"));
            // TimeConverter class is defined in the complete sample
        }

        if (!ReportAvailabilityAtThisTime(theJob))
        {
            Console.WriteLine("\nThat job is set to print only between {0}
and {1}",

TimeConverter.ConvertToLocalHumanReadableTime(theJob.StartTimeOfDay).ToString("t"),
TimeConverter.ConvertToLocalHumanReadableTime(theJob.UntilTimeOfDay).ToString("t"));

        }
        Console.WriteLine("\nThe job will begin printing as soon as it
reaches the top of the queue after:");
        if (theJob.StartTimeOfDay > theJob.HostingPrintQueue.StartTimeOfDay)
        {

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.Start
TimeOfDay).ToString("t"));
        }
        else
        {

Console.WriteLine(TimeConverter.ConvertToLocalHumanReadableTime(theJob.Hosti
ngPrintQueue.StartTimeOfDay).ToString("t"));
        }
    }
}
```

```
    }
}//end if at least one is not available
}//end ReportQueueAndJobAvailability
```

ReportAvailabilityAtThisTime 方法的两个重载除了传递给它们的类型外是相同的，因此下面仅介绍 PrintQueue 版本。

### ① 备注

“这些方法除了类型外均相同”这一事实引出了一个问题：为什么示例不创建泛型方法 ReportAvailabilityAtThisTime<T>。原因是此类方法必须限制为具有该方法调用的 StartTimeOfDay 和 UntilTimeOfDay 属性的类，但泛型方法只能限制为单个类，而继承树中 PrintQueue 和 PrintSystemJobInfo 唯一共有的类是 PrintSystemObject，它没有此类属性。

ReportAvailabilityAtThisTime 方法（在下面的代码示例中显示）首先将 Boolean sentinel 变量初始化为 true。如果队列不可用，它将被重置为 false。

接下来，该方法检查开始时间和“截至”时间是否相同。如果是，则队列始终可用，因此该方法返回 true。

如果队列并非始终可用，则该方法使用静态 UtcNow 属性以 DateTime 对象的形式获取当前时间。（我们不需要本地时间，因为 StartTimeOfDay 和 UntilTimeOfDay 属性本身就是 UTC 时间。）

但是，这两个属性不是 DateTime 对象。它们是 Int32，将时间表示为 UTC 午夜后的分钟数。因此，我们必须将 DateTime 对象转换为午夜之后的分钟数。完成后，该方法只检查“现在”是否在队列的开始时间和“截至”时间之间，如果“现在”不在这两个时间之间，则将标记设置为 false，并返回 sentinel。

C#

```
private static Boolean ReportAvailabilityAtThisTime(PrintQueue pq)
{
    Boolean available = true;
    if (pq.StartTimeOfDay != pq.UntilTimeOfDay) // If the printer is not
available 24 hours a day
    {
        DateTime utcNow = DateTime.UtcNow;
        Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) +
utcNow.TimeOfDay.Minutes;

        // If now is not within the range of available times . . .
        if (!((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight) &&
(utcNowAsMinutesAfterMidnight < pq.UntilTimeOfDay)))
            available = false;
    }
    return available;
}
```

```
        {
            available = false;
        }
    }
    return available;
}//end ReportAvailabilityAtThisTime
```

TimeConverter.ConvertToLocalHumanReadableTime 方法（在下面的代码示例中显示）不使用 Microsoft .NET Framework 引入的任何方法，因此讨论很简短。该方法有一个双重转换任务：它必须采用表示午夜之后分钟数的整数并将其转换为人类可读的时间，并且必须将其转换为本地时间。为此，它首先创建一个设置为 UTC 午夜的 `DateTime` 对象，然后使用 `AddMinutes` 方法添加传递给该方法的分钟数。这将返回一个新的 `DateTime`，表示传递给该方法的原始时间。`ToLocalTime` 方法随后将其转换为本地时间。

C#

```
class TimeConverter
{
    // Convert time as minutes past UTC midnight into human readable time in
    local time zone.
    internal static DateTime ConvertToLocalHumanReadableTime(Int32
timeInMinutesAfterUTCMidnight)
    {
        // Construct a UTC midnight object.
        // Must start with current date so that the local Daylight Savings
        system, if any, will be taken into account.
        DateTime utcNow = DateTime.UtcNow;
        DateTime utcMidnight = new DateTime(utcNow.Year, utcNow.Month,
        utcNow.Day, 0, 0, 0, DateTimeKind.Utc);

        // Add the minutes passed into the method in order to get the
        intended UTC time.
        Double minutesAfterUTCMidnight =
(Double)timeInMinutesAfterUTCMidnight;
        DateTime utcTime = utcMidnight.AddMinutes(minutesAfterUTCMidnight);

        // Convert to local time.
        DateTime localTime = utcTime.ToLocalTime();

        return localTime;
    }// end ConvertToLocalHumanReadableTime
}//end TimeConverter class
```

## 另请参阅

- [DateTime](#)
- [PrintSystemJobInfo](#)
- [PrintQueue](#)

- WPF 中的文档
- 打印概述

# 如何：枚举打印队列的子集

项目 • 2023/02/06

管理公司范围内的打印机集的信息技术 (IT) 专业人员面临的一个常见情况是生成具有某些特征的打印机列表。此功能由 [PrintServer](#) 对象的 [GetPrintQueues](#) 方法和 [EnumeratedPrintQueueTypes](#) 枚举提供。

## 示例

在下面的示例中，代码首先创建一个标志数组，这些标志指定我们要列出的打印队列的特征。在此示例中，我们将查找在打印服务器上本地安装并共享的打印队列。[EnumeratedPrintQueueTypes](#) 枚举提供了许多其他可能性。

然后，该代码会创建一个 [LocalPrintServer](#) 对象，这是一个派生自 [PrintServer](#) 的类。本地打印服务器是运行应用程序的计算机。

最后一个重要步骤是将数组传递给 [GetPrintQueues](#) 方法。

最后，会向用户显示结果。

C#

```
// Specify that the list will contain only the print queues that are
// installed as local and are shared
EnumeratedPrintQueueTypes[] enumerationFlags =
{EnumeratedPrintQueueTypes.Local,
 EnumeratedPrintQueueTypes.Shared};

LocalPrintServer printServer = new LocalPrintServer();

//Use the enumerationFlags to filter out unwanted print queues
PrintQueueCollection printQueuesOnLocalServer =
printServer.GetPrintQueues(enumerationFlags);

Console.WriteLine("These are your shared, local print queues:\n\n");

foreach (PrintQueue printer in printQueuesOnLocalServer)
{
    Console.WriteLine("\tThe shared printer " + printer.Name + " is located
at " + printer.Location + "\n");
}
Console.WriteLine("Press enter to continue.");
Console.ReadLine();
```

可以通过让单步执行每个打印队列的 `foreach` 循环进行进一步筛选来扩展此示例。例如，可以通过让循环调用每个打印队列的 [GetPrintCapabilities](#) 方法来筛选出不支持双面打印的打印机，并测试返回值是否存在双面打印。

## 另请参阅

- [GetPrintQueues](#)
- [PrintServer](#)
- [LocalPrintServer](#)
- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [GetPrintCapabilities](#)
- [WPF 中的文档](#)
- [打印概述](#)
- [Microsoft XPS 文档编写器](#)

# 如何：在不使用反射的情况下获得打印系统对象属性

项目 • 2023/02/06

使用反射逐项列出对象上的属性（以及这些属性的类型）可能会降低应用程序性能。命名空间 [System.Printing.IndexedProperties](#) 提供了一种在不使用反射的情况下获取此信息的方法。

## 示例

执行此操作的步骤如下所示。

1. 创建类型的实例。在下面的示例中，类型是 Microsoft .NET Framework 附带的 [PrintQueue](#) 类型，但几乎相同的代码应该适用于从 [PrintSystemObject](#) 派生的类型。
2. 从类型的 [PropertiesCollection](#) 创建 [PrintPropertyDictionary](#)。此字典中每个条目的 [Value](#) 属性是从 [PrintProperty](#) 派生的类型之一的对象。
3. 枚举字典的成员。对于每个成员，请执行以下操作。
4. 将每个条目的值向上转换为 [PrintProperty](#)，并使用它来创建 [PrintProperty](#) 对象。
5. 获取每个 [PrintProperty](#) 对象的 [Value](#) 类型。

C#

```
// Enumerate the properties, and their types, of a queue without using
Reflection
LocalPrintServer localPrintServer = new LocalPrintServer();
PrintQueue defaultPrintQueue = LocalPrintServer.GetDefaultPrintQueue();

PrintPropertyDictionary printQueueProperties =
defaultPrintQueue.PropertiesCollection;

Console.WriteLine("These are the properties, and their types, of {0}, a
{1}", defaultPrintQueue.Name, defaultPrintQueue.GetType().ToString() + "\n");

foreach (DictionaryEntry entry in printQueueProperties)
{
    PrintProperty property = (PrintProperty)entry.Value;

    if (property.Value != null)
    {
        Console.WriteLine(property.Name + "\t(Type: {0})",
{0} : property.Value.GetType().Name);
    }
}
```

```
property.Value.GetType().ToString());
    }
}
Console.WriteLine("\n\nPress Return to continue...");
Console.ReadLine();
```

## 另请参阅

- [PrintProperty](#)
- [PrintSystemObject](#)
- [System.Printing.IndexedProperties](#)
- [PrintPropertyDictionary](#)
- [LocalPrintServer](#)
- [PrintQueue](#)
- [DictionaryEntry](#)
- [WPF 中的文档](#)
- [打印概述](#)

# 如何：以编程方式打印 XPS 文件

项目 • 2022/09/27

可以使用 [AddJob](#) 方法的一个重载来打印 XML 纸张规范 (XPS) 文件，而根本无需打开 [PrintDialog](#) 或任何用户界面 (UI) ( 从原理上讲 )。

还还可以使用多种 [XpsDocumentWriter.Write](#) 和 [XpsDocumentWriter.WriteAsync](#) 方法打印 XPS 文件。有关详细信息，请参阅[打印 XPS 文档](#)。

打印 XPS 的另一种方法是使用 [PrintDialog.PrintDocument](#) 或 [PrintDialog.PrintVisual](#) 方法。请参阅[调用打印对话框](#)。

## 示例

使用三参数 [AddJob\(String, String, Boolean\)](#) 方法的主要步骤如下。以下示例提供了详细信息。

1. 确定打印机是否是 XPSDrv 打印机。有关 XPSDrv 的详细信息，请参阅[打印概述](#)。
2. 如果打印机不是 XPSDrv 打印机，将线程的单元设置为单线程。
3. 实例化打印服务器并打印队列对象。
4. 调用该方法，指定作业的名称、要打印的文件和一个 [Boolean](#) 标志，该标志指示该打印机是否是 XPSDrv 打印机。

以下示例演示如何以批处理方式打印目录中的所有 XPS 文件。尽管应用程序会提示用户指定目录，但三参数 [AddJob\(String, String, Boolean\)](#) 方法不需要用户界面 (UI)。它可用于具有 XPS 文件名的任何代码路径和可以传递到该方法的路径。

只要 [Boolean](#) 参数为 `false` ( 使用非 XPSDrv 打印机时，该参数必须为此值 )，[AddJob](#) 的三参数 [AddJob\(String, String, Boolean\)](#) 重载必须在单线程单元中运行。但是，.NET 的默认单元状态为多线程。由于本示例假定使用非 XPSDrv 打印机，因此此默认值必须为相反值。

有两种可用于更改此默认值的方法。一种方法是在应用程序的 `Main` 方法 ( 通常为 `"static void Main(string[] args)"` ) 的第一行正上方添加 [STAThreadAttribute](#) ( 即 `"[System.STAThreadAttribute()]"` ) 即可。但是，许多应用程序要求 `Main` 方法具有多线程单元状态，因此存在第二种方法：将对 [AddJob\(String, String, Boolean\)](#) 的调用放在单独的线程中，该线程的单元状态通过 [SetApartmentState](#) 设置为 [STA](#)。以下示例使用第二种方法。

因此，该示例先实例化 `Thread` 对象，并向其传递 `PrintXPS` 方法，以用作 `ThreadStart` 参数。`(PrintXPS` 方法稍后在 `example.`) 下一个线程设置为单个线程单元。`Main` 方法的唯一剩余代码会启动新线程。

该示例的内容主要关于 `static BatchXPSPrinter.PrintXPS` 方法。创建打印服务器和队列后，该方法会提示用户提供包含 XPS 文件的目录。在验证存在该目录且其中存在 \*.xps 文件之后，该方法会将每个此类文件添加到打印队列。该示例假定打印机不是 XPSDrv 打印机，因此将向 `AddJob(String, String, Boolean)` 方法的最后一个参数传递 `false`。出于此原因，该方法先验证文件中的 XPS 标记，然后再尝试将其转换为打印机的页面描述语言。如果验证失败，会引发异常。该示例代码将捕获该异常，并通知用户相关信息，然后继续处理下一 XPS 文件。

C#

```
class Program
{
    [System.MTAThreadAttribute()] // Added for clarity, but this line is
    redundant because MTA is the default.
    static void Main(string[] args)
    {
        // Create the secondary thread and pass the printing method for
        // the constructor's ThreadStart delegate parameter. The
        BatchXPSPrinter
        // class is defined below.
        Thread printingThread = new Thread(BatchXPSPrinter.PrintXPS);

        // Set the thread that will use PrintQueue.AddJob to single
        // threading.
        printingThread.SetApartmentState(ApartmentState.STA);

        // Start the printing thread. The method passed to the Thread
        // constructor will execute.
        printingThread.Start();
    }//end Main
}//end Program class

public class BatchXPSPrinter
{
    public static void PrintXPS()
    {
        // Create print server and print queue.
        LocalPrintServer localPrintServer = new LocalPrintServer();
        PrintQueue defaultPrintQueue =
        LocalPrintServer.GetDefaultPrintQueue();

        // Prompt user to identify the directory, and then create the
        // directory object.
        Console.Write("Enter the directory containing the XPS files: ");
        String directoryPath = Console.ReadLine();
        DirectoryInfo dir = new DirectoryInfo(directoryPath);
```

```

        // If the user mistyped, end the thread and return to the Main
        thread.

        if (!dir.Exists)
        {
            Console.WriteLine("There is no such directory.");
        }
        else
        {
            // If there are no XPS files in the directory, end the thread
            // and return to the Main thread.
            if (dir.GetFiles("*.xps").Length == 0)
            {
                Console.WriteLine("There are no XPS files in the
directory.");
            }
            else
            {
                Console.WriteLine("\nJobs will now be added to the print
queue.");

                Console.WriteLine("If the queue is not paused and the
printer is working, jobs will begin printing.");

                // Batch process all XPS files in the directory.
                foreach (FileInfo f in dir.GetFiles("*.xps"))
                {
                    String nextFile = directoryPath + "\\\" + f.Name;
                    Console.WriteLine("Adding {0} to queue.", nextFile);

                    try
                    {
                        // Print the Xps file while providing XPS validation
                        and progress notifications.
                        PrintSystemJobInfo xpsPrintJob =
defaultPrintQueue.AddJob(f.Name, nextFile, false);
                    }
                    catch (PrintJobException e)
                    {
                        Console.WriteLine("\n\t{0} could not be added to the
print queue.", f.Name);
                        if (e.InnerException.Message == "File contains
corrupted data.")
                        {
                            Console.WriteLine("\tIt is not a valid XPS file.
Use the isXPS Conformance Tool to debug it.");
                        }
                        Console.WriteLine("\tContinuing with next XPS
file.\n");
                    }
                } // end for each XPS file
            } //end if there are no XPS files in the directory
        } //end if the directory does not exist

        Console.WriteLine("Press Enter to end program.");
        Console.ReadLine();

```

```
 } // end PrintXPS method  
 } // end BatchXPSPrinter class
```

如果使用 XPSDrv 打印机，则可将最后一个参数设置为 `true`。在这种情况下，由于 XPS 是打印机的页面描述语言，该方法会将文件发送到打印机，而不会对其进行验证或将其转换为另一种页面描述语言。如果在设计时不确定应用程序是否会使用 XPSDrv 打印机，可以修改应用程序，使其根据所发现的内容读取 `IsXpsDevice` 属性和分支。

由于发布 Windows Vista 和 Microsoft .NET Framework 后，最初存在几个可立即使用的 XPSDrv 打印机，可能需要将非 XPSDrv 打印机伪装为 XPSDrv 打印机。为此，请将 Pipelineconfig.xml 添加到运行应用程序的计算机的注册表项中的以下文件列表：

HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Environments\Windows NT x86\Drivers\Version-3\<PseudoXPSPrinter>\DependentFiles

其中 `<PseudoXPSPrinter>` 是任一打印队列。然后必须重新启动计算机。

此伪装允许用户将 `true` 传递为 `AddJob(String, String, Boolean)` 的最后一个参数，而不会引发异常，但由于 `<PseudoXPSPrinter>` 并不是真正的 XPSDrv 打印机，所以仅会打印垃圾内容。

### ① 备注

为简单起见，以上示例通过测试是否存在 `*.xps` 扩展名来确定文件是否为 XPS。但是，XPS 文件不需要具有此扩展名。`isXPS.exe` (`isXPS 合规性工具`) 是一种测试文件是否具有 XPS 有效性的方法。

## 另请参阅

- [PrintQueue](#)
- [AddJob](#)
- [ApartmentState](#)
- [STAThreadAttribute](#)
- [XPS 文档](#)
- [打印 XPS 文档](#)
- [托管和非托管线程处理](#)
- [isXPS.exe \( isXPS 合规性工具 \)](#)
- [WPF 中的文档](#)
- [打印概述](#)

# 如何：远程调查打印机的状态

项目 • 2023/02/06

在大中型公司，在任何给定时间里，都可能发生由于卡纸、纸张用完或某些其他有问题而导致多台打印机无法工作的情况。Microsoft .NET Framework 的 API 中公开的一组丰富的打印机属性提供一种方法，用于快速调查打印机状态。

## 示例

以下是创建此类实用程序的主要步骤。

1. 获取所有打印服务器的列表。
2. 循环访问服务器以查询其打印队列。
3. 在每一轮服务器循环访问过程中，循环访问所有服务器的队列并读取每个属性，这些属性可能指示队列当前不在工作。

以下代码是一系列代码段。为简单起见，本示例假定存在通过 CRLF 分隔的打印服务器列表。变量 `fileOfPrintServers` 是该文件的 `StreamReader` 对象。由于每个服务器名称是在单独的一行上，因此对 `ReadLine` 的任何调用将获取下一服务器的名称并将 `StreamReader` 的光标移动到下一行的开头。

在外部循环中，代码会创建最新打印服务器的 `PrintServer` 对象，并指定应用程序需具有服务器的管理权限。

### ① 备注

如果有很多服务器，可以通过使用 `PrintServer(String, String[], PrintSystemDesiredAccess)` 构造函数来提高性能，这些构造函数仅初始化需要的属性。

然后，该示例使用 `GetPrintQueues` 来创建所有服务器队列的集合，并开始对其进行循环访问。此内部循环包含一个分支结构，该结构对应于检查打印机状态的两种方法：

- 可以读取类型为 `PrintQueueStatus` 的 `QueueStatus` 属性的标志。
- 可以读取每个相关属性，例如 `IsOutOfPaper` 和 `IsPaperJammed`。

此示例会演示这两种方法，系统会预先提示用户要使用哪种方法，且如果用户想要使用 `QueueStatus` 属性的标志，可回复“y”。请参阅以下有关这两种方法的详细信息。

最后，会向用户显示结果。

C#

```
// Survey queue status for every queue on every print server
String line;
String statusReport = "\n\nAny problem states are indicated below:\n\n";
while ((line = fileOfPrintServers.ReadLine()) != null)
{
    PrintServer myPS = new PrintServer(line,
PrintSystemDesiredAccess.AdministrateServer);
    PrintQueueCollection myPrintQueues = myPS.GetPrintQueues();
    statusReport = statusReport + "\n" + line;
    foreach (PrintQueue pq in myPrintQueues)
    {
        pq.Refresh();
        statusReport = statusReport + "\n\t" + pq.Name + ":";

        if (useAttributesResponse == "y")
        {
            TroubleSpotter.SpotTroubleUsingQueueAttributes(ref
statusReport, pq);
            // TroubleSpotter class is defined in the complete example.
        }
        else
        {
            TroubleSpotter.SpotTroubleUsingProperties(ref statusReport,
pq);
        }
    }
    // end for each print queue
} // end while list of print servers is not yet exhausted

fileOfPrintServers.Close();
Console.WriteLine(statusReport);
Console.WriteLine("\nPress Return to continue.");
Console.ReadLine();
```

若要使用 [QueueStatus](#) 属性的标志检查打印机状态，请检查每个相关标志以查看是否对其进行了设置。 检查是否在一组位标志中设置了一个位的标准方法是执行一个逻辑 AND 运算，其中将该组标志作为一个操作数，将标志本身作为另一操作数。 由于该标志本身仅设置一个位，因此逻辑 AND 的结果至多为设置了该相同位。 若要查明事实是否如此，只需将逻辑 AND 的结果与标志本身进行比较。 有关详细信息，请参阅 [PrintQueueStatus、& 运算符 \(C# 参考\)](#) 和 [FlagsAttribute](#)。

对于已设置了其位的各个特性，代码会将一条通知添加到将向用户显示的最终报告中。  
( 下面会讨论代码结束时调用的 [ReportAvailabilityAtThisTime](#) 方法。 )

C#

```
// Check for possible trouble states of a printer using the flags of the
QueueStatus property
internal static void SpotTroubleUsingQueueAttributes(ref String
statusReport, PrintQueue pq)
{
    if ((pq.QueueStatus & PrintQueueStatus.PaperProblem) ==
PrintQueueStatus.PaperProblem)
    {
        statusReport = statusReport + "Has a paper problem. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.NoToner) ==
PrintQueueStatus.NoToner)
    {
        statusReport = statusReport + "Is out of toner. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.DoorOpen) ==
PrintQueueStatus.DoorOpen)
    {
        statusReport = statusReport + "Has an open door. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.Error) == PrintQueueStatus.Error)
    {
        statusReport = statusReport + "Is in an error state. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.NotAvailable) ==
PrintQueueStatus.NotAvailable)
    {
        statusReport = statusReport + "Is not available. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.Offline) ==
PrintQueueStatus.Offline)
    {
        statusReport = statusReport + "Is off line. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.OutOfMemory) ==
PrintQueueStatus.OutOfMemory)
    {
        statusReport = statusReport + "Is out of memory. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.PaperOut) ==
PrintQueueStatus.PaperOut)
    {
        statusReport = statusReport + "Is out of paper. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.OutputBinFull) ==
PrintQueueStatus.OutputBinFull)
    {
        statusReport = statusReport + "Has a full output bin. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.PaperJam) ==
PrintQueueStatus.PaperJam)
    {
        statusReport = statusReport + "Has a paper jam. ";
    }
}
```

```

    if ((pq.QueueStatus & PrintQueueStatus.Paused) ==
PrintQueueStatus.Paused)
    {
        statusReport = statusReport + "Is paused. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.TonerLow) ==
PrintQueueStatus.TonerLow)
    {
        statusReport = statusReport + "Is low on toner. ";
    }
    if ((pq.QueueStatus & PrintQueueStatus.UserIntervention) ==
PrintQueueStatus.UserIntervention)
    {
        statusReport = statusReport + "Needs user intervention. ";
    }

    // Check if queue is even available at this time of day
    // The method below is defined in the complete example.
    ReportAvailabilityAtThisTime(ref statusReport, pq);
}

```

若要使用各个属性检查打印机状态，只需读取各个属性，并将注释添加到最终报告，如果属性为 `true`，将向用户显示最终报告。（下面会讨论代码结束时调用的 `ReportAvailabilityAtThisTime` 方法。）

C#

```

// Check for possible trouble states of a printer using its properties
internal static void SpotTroubleUsingProperties(ref String statusReport,
PrintQueue pq)
{
    if (pq.HasPaperProblem)
    {
        statusReport = statusReport + "Has a paper problem. ";
    }
    if (!(pq.HasToner))
    {
        statusReport = statusReport + "Is out of toner. ";
    }
    if (pq.IsDoorOpened)
    {
        statusReport = statusReport + "Has an open door. ";
    }
    if (pq.IsInError)
    {
        statusReport = statusReport + "Is in an error state. ";
    }
    if (pq.IsNotNullAvailable)
    {
        statusReport = statusReport + "Is not available. ";
    }
    if (pq.IsOffline)

```

```

{
    statusReport = statusReport + "Is off line. ";
}
if (pq.OutOfMemory)
{
    statusReport = statusReport + "Is out of memory. ";
}
if (pq.OutOfPaper)
{
    statusReport = statusReport + "Is out of paper. ";
}
if (pq.OutputBinFull)
{
    statusReport = statusReport + "Has a full output bin. ";
}
if (pq.PaperJammed)
{
    statusReport = statusReport + "Has a paper jam. ";
}
if (pq.Paused)
{
    statusReport = statusReport + "Is paused. ";
}
if (pq.TonerLow)
{
    statusReport = statusReport + "Is low on toner. ";
}
if (pq.NeedUserIntervention)
{
    statusReport = statusReport + "Needs user intervention. ";
}

// Check if queue is even available at this time of day
// The following method is defined in the complete example.
ReportAvailabilityAtThisTime(ref statusReport, pq);
}//end SpotTroubleUsingProperties

```

创建了 ReportAvailabilityAtThisTime 方法，以应对需要确定队列在一天的当前时间是否可用的情况。

如果 StartTimeOfDay 和 UntilTimeOfDay 属性相等，该方法将不会执行任何操作；因为在这种情况下，打印机始终可用。如果它们不同，则该方法将获取当前时间，该时间必须随后转换为午夜过后的总分钟数，因为 StartTimeOfDay 和 UntilTimeOfDay 属性是表示午夜后分钟数的 Int32，而不是 DateTime 对象。最后，该方法会检查当前时间是否介于开始时间和“截至”时间之间。

C#

```

private static void ReportAvailabilityAtThisTime(ref String statusReport,
PrintQueue pq)
{

```

```
    if (pq.StartTimeOfDay != pq.UntilTimeOfDay) // If the printer is not
available 24 hours a day
    {
DateTime utcNow = DateTime.UtcNow;
Int32 utcNowAsMinutesAfterMidnight = (utcNow.TimeOfDay.Hours * 60) +
utcNow.TimeOfDay.Minutes;

    // If now is not within the range of available times . . .
    if (!((pq.StartTimeOfDay < utcNowAsMinutesAfterMidnight)
&&
(utcNowAsMinutesAfterMidnight < pq.UntilTimeOfDay)))
{
    statusReport = statusReport + " Is not available at this time of
day. ";
}
}
}
```

## 另请参阅

- [StartTimeOfDay](#)
- [UntilTimeOfDay](#)
- [DateTime](#)
- [PrintQueueStatus](#)
- [FlagsAttribute](#)
- [GetPrintQueues](#)
- [PrintServer](#)
- [LocalPrintServer](#)
- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [&运算符 \( C# 引用 \)](#)
- [WPF 中的文档](#)
- [打印概述](#)

# 如何：验证和合并 PrintTicket

项目 • 2023/02/06

Microsoft Windows 打印架构包括灵活且可扩展的 PrintCapabilities 和 PrintTicket 元素。前一个元素逐条列出打印设备的功能，后一个指定设备应如何使用这些功能来处理特定文档序列、单个文档或单个页面。

支持打印的应用程序的典型任务序列应如下。

1. 确定打印机的功能。
2. 配置一个 PrintTicket，用来使用这些功能。
3. 验证 PrintTicket。

本文介绍如何执行此操作。

## 示例

在下面的简单示例中，我们仅对打印机是否支持双工（即双面打印）感兴趣。主要步骤如下。

1. 使用 GetPrintCapabilities 方法获取 PrintCapabilities 对象。
2. 测试是否有你需要的功能。在下面的示例中，我们测试 PrintCapabilities 对象的 DuplexingCapability 属性是否有以下功能：在一张纸的两面进行打印，并沿纸张的长边“翻页”。DuplexingCapability 是集合，因此我们使用 Contains 的 ReadOnlyCollection<T> 方法。

### ① 备注

此步骤不是绝对必需的。下面使用的 MergeAndValidatePrintTicket 方法将根据打印机的功能检查 PrintTicket 中的每个请求。如果打印机不支持请求的功能，打印机驱动程序将在方法返回的 PrintTicket 中用备用请求作为替代。

3. 如果打印机支持双工，示例代码创建要求双工的 PrintTicket。但应用程序不指定 PrintTicket 元素中可用的每个可能的打印机设置。那样做会在程序员和程序时间两个方面产生浪费。相反，代码只设置双工请求，然后将此 PrintTicket 与已完全配置且验证的现有 PrintTicket 合并，在本例中为用户的默认 PrintTicket。
4. 相应的，该示例调用 MergeAndValidatePrintTicket 方法将新的最小 PrintTicket 与用户的默认 PrintTicket 合并。这会返回一个 ValidationResult 属性，它包含

[PrintTicket](#) 作为其属性之一。

5. 然后，该示例测试新的 [PrintTicket](#) 请求双工。如果是这样，该示例接下来将它设为用户的新的默认打印工单。如果漏掉了上面的步骤 2，并且打印机不支持沿长边双工，测试会得出 `false`。（请参阅上面的说明。）

6. 最后一个重要步骤是使用 [Commit](#) 方法将更改提交到 [PrintQueue](#) 的 [UserPrintTicket](#) 属性。

C#

```
/// <summary>
/// Changes the user-default PrintTicket setting of the specified print
queue.
/// </summary>
/// <param name="queue">the printer whose user-default PrintTicket setting
needs to be changed</param>
static private void ChangePrintTicketSetting(PrintQueue queue)
{
    //
    // Obtain the printer's PrintCapabilities so we can determine whether or
not
    // duplexing printing is supported by the printer.
    //
    PrintCapabilities printcap = queue.GetPrintCapabilities();

    //
    // The printer's duplexing capability is returned as a read-only
collection of duplexing options
    // that can be supported by the printer. If the collection returned
contains the duplexing
    // option we want to set, it means the duplexing option we want to set
is supported by the printer,
    // so we can make the user-default PrintTicket setting change.
    //
    if (printcap.DuplexingCapability.Contains(Duplexing.TwoSidedLongEdge))
    {
        //
        // To change the user-default PrintTicket, we can first create a
delta PrintTicket with
        // the new duplexing setting.
        //
        PrintTicket deltaTicket = new PrintTicket();
        deltaTicket.Duplexing = Duplexing.TwoSidedLongEdge;

        //
        // Then merge the delta PrintTicket onto the printer's current user-
default PrintTicket,
        // and validate the merged PrintTicket to get the new PrintTicket we
want to set as the
        // printer's new user-default PrintTicket.
        //
        ValidationResult result =
```

```

queue.MergeAndValidatePrintTicket(queue.UserPrintTicket, deltaTicket);

    //
    // The duplexing option we want to set could be constrained by other
    PrintTicket settings
        //
        // or device settings. We can check the validated merged PrintTicket
        to see whether the
            //
            // the validation process has kept the duplexing option we want to
            set unchanged.
        //
        if (result.ValidatedPrintTicket.Duplexing ==
Duplexing.TwoSidedLongEdge)
    {
        //
        // Set the printer's user-default PrintTicket and commit the set
        operation.
        //
        queue.UserPrintTicket = result.ValidatedPrintTicket;
        queue.Commit();
        Console.WriteLine("PrintTicket new duplexing setting is set on
'{0}'.", queue.FullName);
    }
    else
    {
        //
        // The duplexing option we want to set has been changed by the
        validation process
            //
            // when it was resolving setting constraints.
        //
        Console.WriteLine("PrintTicket new duplexing setting is
constrained on '{0}'.", queue.FullName);
    }
}
else
{
    //
    // If the printer doesn't support the duplexing option we want to
    set, skip it.
    //
    Console.WriteLine("PrintTicket new duplexing setting is not
supported on '{0}'.", queue.FullName);
}
}

```

这样，你就能快速地测试此示例，下面显示了该示例的其余部分。 创建项目和命名空间，然后将本文中的两个代码片段粘贴到命名空间块中。

C#

```

/// <summary>
/// Displays the correct command line syntax to run this sample program.
/// </summary>
static private void DisplayUsage()

```

```

{
    Console.WriteLine();
    Console.WriteLine("Usage #1: printticket.exe -l \"<printer_name>\"");
    Console.WriteLine("      Run program on the specified local printer");
    Console.WriteLine();
    Console.WriteLine("      Quotation marks may be omitted if there are no
spaces in printer_name.");
    Console.WriteLine();
    Console.WriteLine("Usage #2: printticket.exe -r \"\\\\\\<server_name>\\\
<printer_name>\"");
    Console.WriteLine("      Run program on the specified network printer");
    Console.WriteLine();
    Console.WriteLine("      Quotation marks may be omitted if there are no
spaces in server_name or printer_name.");
    Console.WriteLine();
    Console.WriteLine("Usage #3: printticket.exe -a");
    Console.WriteLine("      Run program on all installed printers");
    Console.WriteLine();
}

[STAThread]
static public void Main(string[] args)
{
    try
    {
        if ((args.Length == 1) && (args[0] == "-a"))
        {
            //
            // Change PrintTicket setting for all local and network printer
connections.
            //
            LocalPrintServer server = new LocalPrintServer();

            EnumeratedPrintQueueTypes[] queue_types =
{EnumeratedPrintQueueTypes.Local,

EnumeratedPrintQueueTypes.Connections};

            //
            // Enumerate through all the printers.
            //
            foreach (PrintQueue queue in server.GetPrintQueues(queue_types))
            {
                //
                // Change the PrintTicket setting queue by queue.
                //
                ChangePrintTicketSetting(queue);
            }
        }//end if -a

        else if ((args.Length == 2) && (args[0] == "-l"))
        {
            //
            // Change PrintTicket setting only for the specified local
printer.

```

```

        //
        LocalPrintServer server = new LocalPrintServer();
        PrintQueue queue = new PrintQueue(server, args[1]);
        ChangePrintTicketSetting(queue);
    }//end if -l

    else if ((args.Length == 2) && (args[0] == "-r"))
    {
        //
        // Change PrintTicket setting only for the specified remote
        printer.
        //
        String serverName = args[1].Remove(args[1].LastIndexOf(@"\")); 
        String printerName = args[1].Remove(0,
        args[1].LastIndexOf(@"\")+1);
        PrintServer ps = new PrintServer(serverName);
        PrintQueue queue = new PrintQueue(ps, printerName);
        ChangePrintTicketSetting(queue);
    }//end if -r

    else
    {
        //
        // Unrecognized command line.
        // Show user the correct command line syntax to run this sample
        program.
        //
        DisplayUsage();
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
    Console.WriteLine(e.StackTrace);

    //
    // Show inner exception information if it's provided.
    //
    if (e.InnerException != null)
    {
        Console.WriteLine("--- Inner Exception ---");
        Console.WriteLine(e.InnerException.Message);
        Console.WriteLine(e.InnerException.StackTrace);
    }
}
finally
{
    Console.WriteLine("Press Return to continue...");
    Console.ReadLine();
}
}//end Main

```

## 另请参阅

- [PrintCapabilities](#)
- [PrintTicket](#)
- [GetPrintQueues](#)
- [PrintServer](#)
- [EnumeratedPrintQueueTypes](#)
- [PrintQueue](#)
- [GetPrintCapabilities](#)
- [WPF 中的文档](#)
- [打印概述](#)
- [打印架构](#)

# 全球化和本地化

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 为开发全球通用的应用程序提供了广泛的支持。

## 本节内容

[WPF 全球化和本地化概述](#)

[WPF 的全球化](#)

[使用自动布局概述](#)

[本地化特性和注释](#)

[WPF 中的双向功能概述](#)

[操作指南主题](#)

## 参考

[System.Globalization](#)

[FlowDirection](#)

[NeutralResourcesLanguageAttribute](#)

[XAML 中 xml:lang 的处理](#)

## 相关章节

# WPF 全球化和本地化概述

项目 • 2022/09/27

当你将自己的产品限制为只能通过一种语言使用时，便将潜在的客户群限制为全球 75 亿人口中的一小部分。如果想让自己的应用程序被全球用户所接受，那么对产品进行经济而有效的本地化将是赢得更多客户的最好、最经济的方法。

本概述文章介绍 WPF 中的全球化和本地化提供的全球化设计功能，包括自动布局、附属程序集以及本地化属性和注释。

本地化是针对应用程序所支持的特定区域性将应用程序资源转换为本地化版本的过程。在 WPF 中进行本地化时，可使用 [System.Windows.Markup.Localizer](#) 命名空间中的 API。这些 API 支持 [LocBaml 工具示例](#) 命令行工具。有关如何生成和使用 LocBaml 的信息，请参阅[对应用程序进行本地化](#)。

## 在 WPF 中进行全球化和本地化的最佳做法

按照本部分提供的与 UI 设计和本地化相关的提示操作，可以最大限度地利用 WPF 内置的大部分全球化和本地化功能。

### WPF UI 设计的最佳做法

设计 UI 时，请考虑实施下列最佳做法：

- 使用代码编写 UI。使用 XAML 创建 UI 时，应通过内置的本地化 API 公开 UI。
- 避免使用绝对位置和固定大小对内容进行布局；改用相对位置或自动调整大小。
  - 使用 [SizeToContent](#)，并将宽度和高度设置为 `Auto`。
  - 避免使用 [Canvas](#) 对 UI 进行布局。
  - 使用 [Grid](#) 及其大小共享功能。
- 在边距中提供额外的空间，因为本地化文本通常需要更多空间。额外空间为可能会延伸的字符预留了余地。
- 对 [TextBlock](#) 启用 [TextWrapping](#)，以避免剪裁。
- 设置 `xml:lang` 特性。此特性描述特定元素及其子元素的区域性。此属性的值可更改 WPF 中几种功能的行为。例如，它可以更改断字、拼写检查、数字替换、复杂脚本成型和字体回退的行为。有关设置 [XAML 中 xml:lang 的处理](#)的详细信息，请参阅 [WPF 的全球化](#)。

- 创建自定义的复合字体，以更好地控制用于不同语言的字体。默认情况下，WPF 使用 Windows\Fonts 目录中的 GlobalUserInterface.composite 字体。
- 当创建的导航应用程序可能在以从右到左的格式显示文本的区域性中进行本地化时，请显式设置每页的 `FlowDirection`，以确保该页不从 `NavigationWindow` 继承 `FlowDirection`。
- 当创建在浏览器之外托管的独立导航应用程序时，请将初始应用程序的 `StartupUri` 设置为 `NavigationWindow` 而不是页面（例如 `<Application StartupUri="NavigationWindow.xaml">`）。此设计使你可以更改窗口和导航栏的 `FlowDirection`。有关详细信息和示例，请参阅 [Globalization Homepage Sample](#)（全球化主页示例）。

## WPF 本地化的最佳做法

对基于 WPF 的应用程序进行本地化时，请考虑实施下列最佳做法：

- 使用本地化注释为本地化人员提供额外的上下文。
- 使用本地化特性控制本地化，而不是选择性地省略元素的 `Uid` 属性。有关详细信息，请参阅[本地化特性和注释](#)。
- 使用 `msbuild -t:updateuid` 和 `-t:checkuid` 在 UI 中添加和检查 `Uid` 属性，该任务通常很繁琐且不太准确。
  - 开始进行本地化之后，请勿编辑或更改 `Uid` 属性。
  - 请勿使用重复的 `Uid` 属性（使用“复制并粘贴”命令时，请记住此提示）。
  - 在 `AssemblyInfo.*` 中设置 `UltimateResourceFallback` 位置，以指定合适的回退语言（例如 `[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.Satellite)]`）。

如果决定通过在项目文件中省略 `<UICulture>` 标记，在主程序集中添加源语言，请将 `UltimateResourceFallback` 位置设置为主程序集而不是附属程序集（例如 `[assembly: NeutralResourcesLanguage("en-US", UltimateResourceFallbackLocation.MainAssembly)]`）。

## 对 WPF 应用程序进行本地化

对 WPF 应用程序进行本地化时，有多种选择。例如，可以将应用程序中的可本地化资源绑定到 XML 文件，在 resx 表中存储可本地化文本，或者让本地化人员使用 Extensible

Application Markup Language (XAML) 文件。本部分介绍使用 XAML 的 BAML 形式的本地化工作流，这种工作流提供以下几个好处：

- 可以在生成之后进行本地化。
- 可以从较旧版本 XAML 的 BAML 形式更新到本地化的较新版本 XAML 的 BAML 形式，以便在开发的同时进行本地化。
- 因为 XAML 的 BAML 形式是 XAML 的已编译形式，所以可以在编译时验证原始源元素和语义。

## 本地化生成过程

开发 WPF 应用程序时，本地化的生成过程如下：

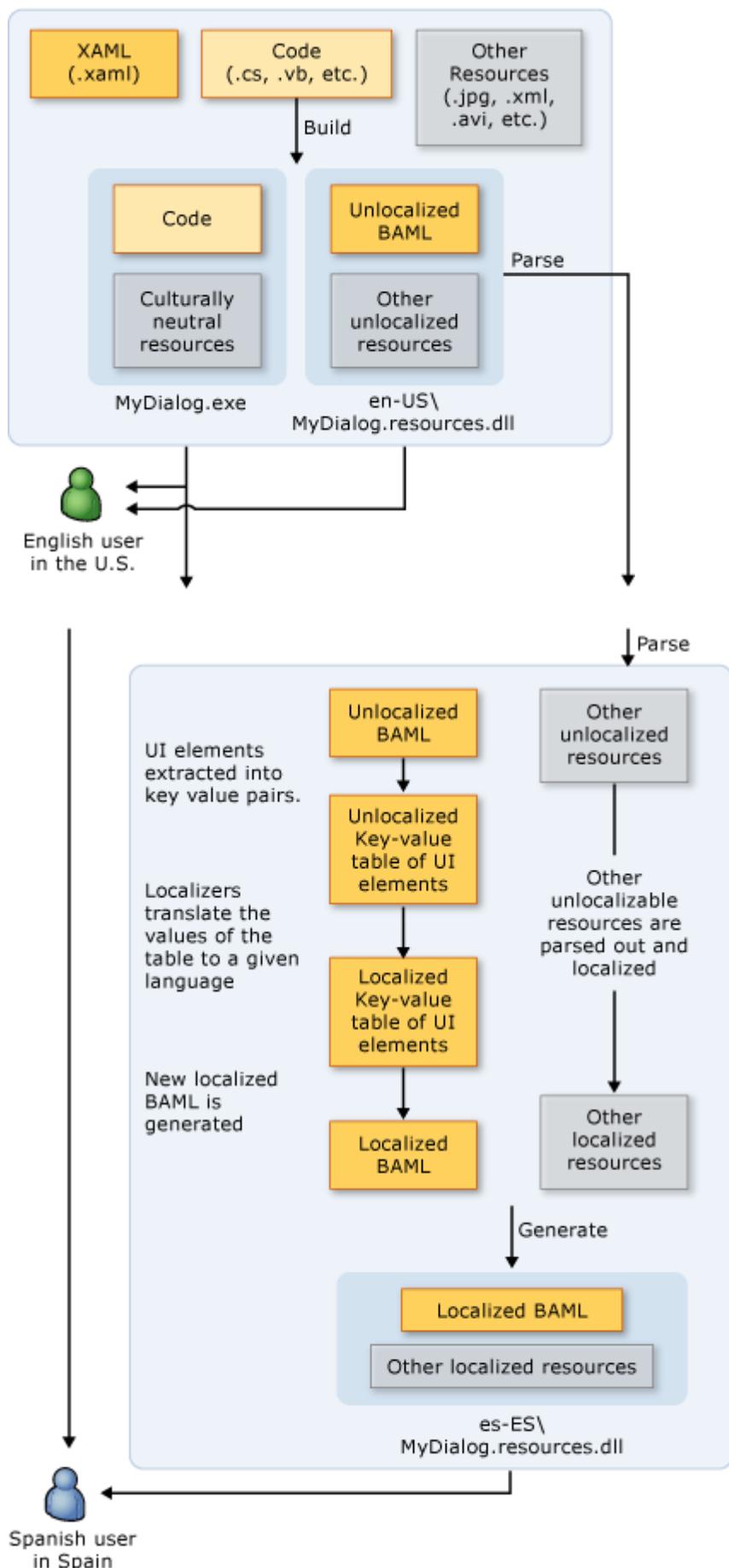
- 开发人员创建 WPF 应用程序并将其全球化。在项目文件中，开发人员设置 `<UICulture>en-US</UICulture>`，以便在编译应用程序时生成一个中性语言的主程序集。此程序集具有一个附属 `.resources.dll` 文件，其中包含所有可本地化的资源。因为本地化 API 支持从主程序集进行提取，所以可选择在主程序集中保留源语言。
- 将文件编译到生成中时，会将 XAML 转换为 XAML 的 BAML 形式。将向说英语的客户发布非特定区域性的 `MyDialog.exe` 和区域性相关的（英语）`MyDialog.resources.dll` 文件。

## 本地化工作流

本地化过程在生成未本地化的 `MyDialog.resources.dll` 文件之后开始。使用 `System.Windows.Markup.Localizer` 下的 API 将原始 XAML 中的 UI 元素和属性从 XAML 的 BAML 形式提取为键值对。本地化人员使用键/值对来对应用程序进行本地化。在本地化完成之后，可以从新值生成一个新的 `.resource.dll`。

键值对的键是在本地化人员开始进行本地化之后，开发人员放置在原始 UI 中的 `x:Uid` 值，你可以将开发更改与已完成的本地化工作进行合并，使损失的翻译工作降至最少。

下图显示了一个基于 XAML 的 BAML 形式的典型本地化工作流。此关系图假设开发人员用英语编写应用程序。开发人员创建 WPF 应用程序并将其全球化。在项目文件中，开发人员设置 `<UICulture>en-US</UICulture>`，以便在生成时会生成一个中性语言的主程序集，该程序集具有一个包含所有可本地化资源的附属 `.resources.dll`。或者，因为 WPF 本地化 API 支持从主程序集进行提取，所以还可以保留主程序集中的源语言。生成过程结束之后，XAML 会编译为 BAML。将向说英语的客户提供非特定区域性的 `MyDialog.exe.resources.dll`。



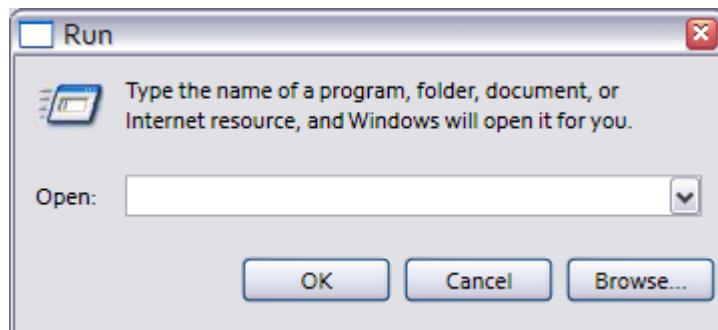
## WPF 本地化示例

本部分包含几个本地化应用程序示例，有助于你了解如何生成和本地化 WPF 应用程序。

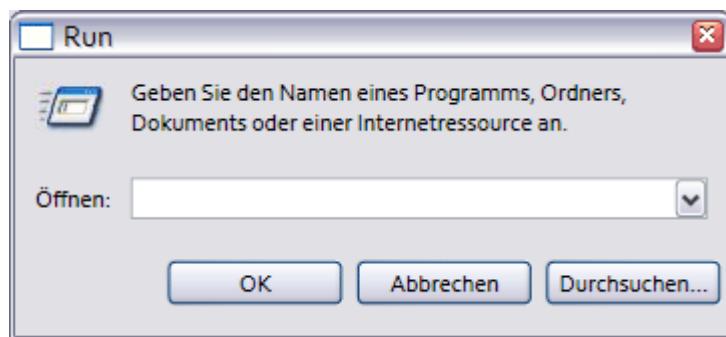
# “运行”对话框示例

下图显示“运行”对话框示例的输出。

**英语：**



**德语：**



## 设计全球化“运行”对话框

此示例使用 WPF 和 XAML 生成一个“运行”对话框。此对话框与 Microsoft Windows“开始”菜单中提供的“运行”对话框等效。

生成全球化对话框的一些要点包括：

### 自动布局

在 *Window1.xaml* 中：

```
<Window SizeToContent="WidthAndHeight">
```

以前的 Window 属性会根据内容大小自动调整窗口大小。此属性可以防止窗口切断在本地化之后大小增加的内容；它还可以在内容由于本地化而大小减小时删除不必要的空格。

```
<Grid x:Uid="Grid_1">
```

为了使 WPF 本地化 API 正确运行，需要使用 *Uid* 属性。

它们由具有较旧的 UI 本地化的 UI 使用。通过在命令行界面中运行 `msbuild -t:updateuid RunDialog.csproj`，可以添加 *Uid* 属性。因为手动添加 *Uid* 属性通常比较费

时并且准确性较差，所以建议使用此方法来添加这些属性。可以通过运行 `msbuild -t:checkuid RunDialog.csproj` 来检查是否正确设置了 `Uid` 属性。

使用 `Grid` 控件可以构造 UI，这是一个有用的控件，它可以利用位于每个单元格中的 UI 元素中的自动布局，适应本地化过程中大小的增加和减小。

#### XAML

```
<Grid.ColumnDefinitions>
<ColumnDefinition x:Uid="ColumnDefinition_1" />
<ColumnDefinition x:Uid="ColumnDefinition_2" />
```

放置 `Open:` 标签和 `ComboBox` 的前两列占据 UI 总宽度的 10%。

#### XAML

```
<ColumnDefinition x:Uid="ColumnDefinition_3" SharedSizeGroup="Buttons" />
<ColumnDefinition x:Uid="ColumnDefinition_4" SharedSizeGroup="Buttons" />
<ColumnDefinition x:Uid="ColumnDefinition_5" SharedSizeGroup="Buttons" />
</Grid.ColumnDefinitions>
```

请注意，该示例使用 `Grid` 的共享调整大小功能。最后三列通过将自身放置在相同的 `SharedSizeGroup` 中利用此功能。正如属性名称所示，此属性允许不同的列采用相同大小。因此，在将“Browse...”本地化为更长的字符串“Durchsuchen...”时，所有按钮的宽度都会增加，而不是显示一个小小的“OK”按钮和一个大得不相称的“Durchsuchen...”按钮。

#### xml:lang

```
xml:lang="en-US"
```

请注意放置在 UI 根元素中的 [XAML 中的 `xml:lang` 处理](#)。此属性描述给定元素及其子元素的区域性。WPF 中的多项功能都使用此值，在本地化过程中应对此值进行相应的更改。此值会更改在断字以及对字词进行拼写检查时所使用的字典。它还会影响数字的显示以及字体回退系统选择所用字体的方式。最后，该属性会影响数值的显示方式，形成在复杂脚本中编写文本的方式。默认值为“en-US”。

## 生成附属资源程序集

在 `.csproj` 中：

编辑 `.csproj` 文件并将以下标记添加到无条件 `<PropertyGroup>`：

```
<UICulture>en-US</UICulture>
```

请注意，增加了 `UICulture` 值。如果将此属性设置为有效的 `CultureInfo` 值（例如，“en-US”），生成项目时会生成一个包含所有可本地化资源的附属程序集。

```
<Resource Include="RunIcon.JPG">  
    <Localizable>False</Localizable>  
</Resource>
```

不需要对 RunIcon.JPG 进行本地化，因为它对所有区域性都应显示相同的外观。

Localizable 设置为 false，以使其保留在中性语言主程序集而不是附属程序集中。所有不可编译资源的默认值是 Localizable 设置为 true。

## 本地化“运行”对话框

Parse

生成应用程序之后，对其进行本地化的第一步是将可本地化的资源从附属程序集中分析出来。本主题使用 [LocBaml 工具示例](#) 上的示例 LocBaml 工具。请注意，LocBaml 只是一个示例工具，用于帮助你了解有关如何生成适合你的本地化过程的本地化工具的入门知识。使用 LocBaml 可以运行下面的命令进行分析：LocBaml /parse RunDialog.resources.dll /out:，从而生成“RunDialog.resources.dll.CSV”文件。

## 本地化

使用你喜欢的支持 Unicode 的 CSV 编辑器来编辑此文件。筛选掉本地化类别为“None”的所有项。应看到下面的项：

资源键	本地化类别	值
Button_1:System.Windows.Controls.Button.\$Content	Button	OK
Button_2:System.Windows.Controls.Button.\$Content	Button	取消
Button_3:System.Windows.Controls.Button.\$Content	Button	浏览...
ComboBox_1:System.Windows.Controls.ComboBox.\$Content	ComboBox	
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	文本	Windows 将根据您所输入的名称，为您打开相应的程序、文件夹、文档或 Internet 资源。
TextBlock_2:System.Windows.Controls.TextBlock.\$Content	文本	未解决：
Window_1:System.Windows.Window.Title	标题	运行

将该应用程序本地化为德语版本需要进行下面的翻译：

资源键	本地化类别	值
Button_1:System.Windows.Controls.Button.\$Content	Button	OK
Button_2:System.Windows.Controls.Button.\$Content	Button	Abbrechen
Button_3:System.Windows.Controls.Button.\$Content	Button	Durchsuchen...
ComboBox_1:System.Windows.Controls.ComboBox.\$Content	ComboBox	
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	文本	Geben Sie den Namen eines Programms, Ordners, Dokuments oder einer Internetressource an.
TextBlock_2:System.Windows.Controls.TextBlock.\$Content	文本	打开：
Window_1:System.Windows.Window.Title	标题	运行

## Generate

本地化的最后一步涉及创建新进行本地化的附属程序集。 可以使用下面的 LocBaml 命令完成此操作：

```
LocBaml.exe /generate RunDialog.resources.dll /trans:RunDialog.resources.dll.CSV
/out: . /cul:de-DE
```

在德语版 Windows 上，如果此 resources.dll 放置在主程序集旁边的 de-DE 文件夹中，则会自动加载此资源而不是 en-US 文件夹中的资源。 如果没有德语版的 Windows 来测试这种情况，请将区域性设置为你所使用的 Windows 的任何区域性（例如，en-US），并替换原始的资源 DLL。

## 附属资源加载

MyDialog.exe	en-US\MyDialog.resources.dll	de-DE\MyDialog.resources.dll
代码	原始英语版 BAML	本地化的 BAML
非特定区域性资源	其他英语资源	已本地化为德语的其他资源

.NET 根据应用程序的 `Thread.CurrentUICulture` 自动选择要加载的附属资源程序集。 其默认值为你的 Windows OS 的区域性。 如果使用德语版 Windows，则会加载 de-DE\MyDialog.resources.dll 文件。 如果使用英语版 Windows，则会加载 en-US\MyDialog.resources.dll 文件。 通过在项目的 AssemblyInfo 文件中指定

`NeutralResourcesLanguage` 属性，可以设置应用程序的最终回退资源。例如，如果指定：

```
[assembly: NeutralResourcesLanguage("en-US",
UltimateResourceFallbackLocation.Satellite)]
```

并且 de-DE\MyDialog.resources.dll 和 de\MyDialog.resources.dll 都不可用，则德语版 Windows 将使用 en-US\MyDialog.resources.dll 文件。

## Microsoft 沙特阿拉伯主页

下图显示了英语和阿拉伯语主页。有关生成这些图形的完整示例，请参阅 [Globalization Homepage Sample](#)（全球化主页示例）。

英语：



阿拉伯语：



## 设计 Microsoft 全球主页

Microsoft 沙特阿拉伯网站的这个实体模型说明了针对从右向左布局语言提供的全球化功能。希伯来语和阿拉伯语等语言具有从右到左的阅读顺序，因此 UI 的布局方式通常必须与英语等从左到右的语言截然不同。从从左到右布局语言本地化到从右到左布局语言可能相当困难，反之亦然。WPF 可使此类本地化工作变得更容易。

### FlowDirection

*Homepage.xaml:*

```

XAML

<Page x:Uid="Page_1" x:Class="MicrosoftSaudiArabiaHomepage.Homepage"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      FlowDirection="LeftToRight"
      Localization.Comments="FlowDirection(This FlowDirection controls the
actual content of the homepage)"
      xml:lang="en-US">

```

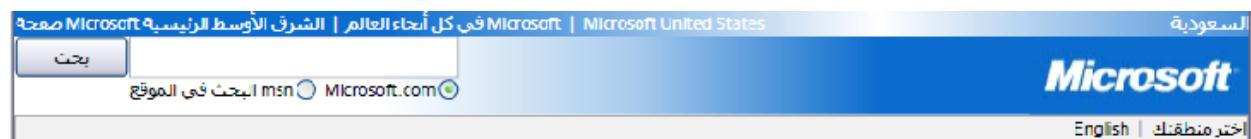
请注意 [Page](#) 的 [FlowDirection](#) 属性。将此属性更改为 [RightToLeft](#) 将更改 [Page](#) 及其子元素的 [FlowDirection](#)，以使此 UI 的布局翻转成从右到左布局，从而满足阿拉伯语用户的需求。可以通过在任何元素上指定显式的 [FlowDirection](#) 来替代继承行为。[FlowDirection](#) 属性对所有 [FrameworkElement](#) 或文档相关元素都可用，并且具有隐式值 [LeftToRight](#)。

请注意，当根 [FlowDirection](#) 发生更改时，甚至连背景渐变画笔都进行了正确的翻转：

[FlowDirection="LeftToRight"](#)



[FlowDirection="RightToLeft"](#)



## 避免对面板和控件使用固定维度

浏览 [Homepage.xaml](#) 时会注意到，除了为顶部 [DockPanel](#) 的整个 UI 指定的固定宽度和高度之外，没有其他固定维度。请勿使用固定尺寸，以防裁剪可能比源文本长的本地化文本。WPF 面板和控件将根据其包含的内容自动调整大小。大多数控件还具有最小和最大维度，设置这些维度可以加强控制（例如 [MinWidth= "20"](#)）。通过 [Grid](#)，还可以使用 ["\\*\\*"](#) 设置相对宽度和高度（例如 [Width= "0.25\\*\\*"](#)），或使用其单元格大小共享功能。

## 本地化注释

在很多情况下，内容可能不太明确，难以翻译。开发人员或设计人员可通过本地化注释为本地化人员提供额外的上下文和注释。例如，下面的 [Localization.Comments](#) 阐明了字符“|”的用法。

```
XAML

<TextBlock
    x:Uid="TextBlock_2"
    DockPanel.Dock="Right"
    Foreground="White"
    Margin="5,0,5,0"
    Localization.Comments="$Content(This character is used as a decorative
rule.)">
    |
</TextBlock>
```

此注释与 [TextBlock\\_1](#) 的内容相关联，并且在使用 [LocBaml](#) 工具（请参阅[对应用程序进行本地化](#)）时，可以在输出 [.csv](#) 文件的 [TextBlock\\_1](#) 行的第六列看到此注释：

资源键	类别	可读性	可修改性	评论	值
TextBlock_1:System.Windows.Controls.TextBlock.\$Content	文本	TRUE	TRUE	此字符串用作装饰性规则。	

使用下面的语法可以将注释放置在任何元素的内容或属性上：

XAML

```
<TextBlock
    x:Uid="TextBlock_1"
    DockPanel.Dock="Right"
    Foreground="White"
    Margin="5,0,5,0"
    Localization.Comments="$Content(This is a comment on the TextBlock's
content.)"
    Margin(This is a comment on the TextBlock's Margin property.)>
|
</TextBlock>
```

## 本地化特性

通常，开发人员或本地化经理需要控制本地化人员能够阅读和修改的内容。例如，可能不希望本地化人员翻译你公司的名称或法律用语。WPF 提供了一些特性，使用这些特性可以设置元素的内容或属性（本地化工具可以使用这些内容或属性锁定、隐藏元素或对元素进行排序）的可读性、可修改性和类别。有关详细信息，请参阅 [Attributes](#)。此示例中 LocBaml 工具仅输出这些特性的值。WPF 控件对这些特性都使用默认值，但可以替代这些默认值。例如，以下示例替代 `TextBlock_1` 的默认本地化特性，并将其内容设置为可供本地化人员阅读，但不能被其修改。

XAML

```
<TextBlock
    x:Uid="TextBlock_1"
    Localization.Attributes=
    "$Content(Readable Unmodifiable)">
    Microsoft Corporation
</TextBlock>
```

除了可读性和可修改性之外，WPF 还提供了常见 UI 类别 ([LocalizationCategory](#)) 的枚举，使用该枚举可以为本地化人员提供更多上下文。平台控件的 WPF 默认类别也可以在 XAML 中替代：

XAML

```
<TextBlock x:Uid="TextBlock_2">
<TextBlock.ToolTip>
<TextBlock
x:Uid="TextBlock_3"
Localization.Attributes=
"$Content(ToolTip Readable Unmodifiable)">
Microsoft Corporation
</TextBlock>
</TextBlock.ToolTip>
Windows Vista
</TextBlock>
```

WPF 提供的默认本地化特性还可以通过代码进行替代，因此可以正确地为自定义控件设置合适的默认值。例如：

C#

```
[Localizability(Readability = Readability.Readable,
Modifiability=Modifiability.Unmodifiable, LocalizationCategory.None)]
public class CorporateLogo : TextBlock
{
    // ...
}
```

在 XAML 中设置的实例特定特性优先于在自定义控件的代码中设置的值。有关特性和注释的详细信息，请参阅[本地化特性和注释](#)。

## 字体回退和复合字体

如果指定一种不支持给定代码点范围的字体，WPF 将使用 Windows\Fonts 目录中的 Global User Interface.compositefont 自动回退到支持该范围的字体。复合字体与任何其他字体的用法相同，通过设置元素的 `FontFamily`（例如 `FontFamily="Global User Interface"`）可以显式使用复合字体。通过创建你自己的复合字体并指定针对特定代码点范围和语言所使用的字体，可以指定你自己的字体回退首选项。

有关复合字体的详细信息，请参阅 [FontFamily](#)。

## 本地化 Microsoft 主页

可以按照“运行”对话框示例中的相同步骤对此应用程序进行本地化。在 [Globalization Homepage Sample](#)（全球化主页示例）中可以找到本地化的阿拉伯语 .csv 文件。

# WPF 的全球化

项目 • 2023/02/06

本主题介绍了编写面向全球市场的 Windows Presentation Foundation (WPF) 应用程序时应注意的问题。 [System.Globalization](#) 命名空间的 .NET 中定义了全球化编程元素。

## XAML 全球化

Extensible Application Markup Language (XAML) 基于 XML，并利用在 XML 规范中定义的全球化支持。以下各节介绍了一些应了解的 XML 功能。

### 字符引用

字符引用使用十进制或十六进制的形式指定其表示的特定 Unicode 字符的 UTF16 代码单元。下例演示 COPTIC CAPITAL LETTER HORI 或“ڦ”十进制字符引用：

XAML

```
&#1000;
```

下例演示十六进制字符引用。请注意，十六进制数字的前面具有 x。

XAML

```
&x3E8;
```

### 编码

XAML 支持的编码有 ASCII、Unicode UTF-16 和 UTF-8。编码语句位于 XAML 文档的开头。如果不存在编码特性，并且没有任何字节顺序，则分析器默认为 UTF-8。UTF-8 和 UTF-16 都是首选编码。不支持 UTF-7。以下示例演示如何在 XAML 文件中指定 UTF-8 编码。

XAML

```
?xml encoding="UTF-8"?
```

### 语言特性

XAML 使用 `xml:lang` 来表示元素的语言特性。 若要使用 `CultureInfo` 类，语言特性值应为 `CultureInfo` 预定义的区域性名称之一。`xml:lang` 在元素树中可继承（可按 XML 规则继承，但并不一定这样继承，因为存在依赖属性继承）；在未明确赋值的情况下，其默认值为空字符串。

语言特性对指定方言非常有用。例如，法语在法国、魁北克、比利时和瑞士的拼写、词汇和发音各不相同。此外，中文、日语和朝鲜语虽然在 Unicode 中具有共同的码位，但表意文字字形不同，并使用完全不同的字体。

以下 Extensible Application Markup Language (XAML) 示例使用 `fr-CA` 语言特性指定加拿大法语。

#### XAML

```
<TextBlock xml:lang="fr-CA">Découvrir la France</TextBlock>
```

## Unicode

WPF 应用程序可使用 `StringInfo` 操作字符串，而无需知道其是否具有代理项对或组合字符。

# 使用 XAML 设计国际用户界面

本节说明编写应用程序时应考虑的用户界面 (UI) 功能。

## 国际化文本

WPF 包括用于所有 Microsoft .NET Framework 支持的书写系统的内置处理。

目前支持以下脚本：

- 阿拉伯语
- 孟加拉语
- 梵语
- 西里尔语
- 希腊语
- 古吉拉特语
- 果鲁穆奇语

- 希伯来语
- 表意文字脚本
- 卡纳达语
- 老挝语
- 拉丁语
- 马拉雅拉姆语
- 蒙古语
- 奥里亚语
- 叙利亚语
- 泰米尔语
- 泰卢固语
- 塔安那文
- 泰语\*
- 藏语

\*此版本支持显示和编辑泰语文本，但不支持断词。

目前不支持以下脚本：

- 高棉语
- 古朝鲜语
- 缅甸
- 僧伽罗语

所有书写系统引擎都支持 OpenType 字体。OpenType 字体可包括 OpenType 布局表格，字体创建者通过此布局表格能够更好地设计国际化的高端版式字体。OpenType 字体布局表格包含有关字形替换、字形位置、对齐方式、基线位置的信息，以便文本处理应用程序改进文本布局。

通过 OpenType 字体，可使用 Unicode 编码来处理大型字形集。这种编码享有广泛的国际支持，并支持版式字形变体。

WPF 文本呈现使用 Microsoft ClearType 子像素技术，该技术支持分辨率独立性。这极大地提高了可读性，并为所有脚本提供了支持高质量杂志样式文档的功能。

## 国际化布局

WPF 提供一种支持水平、双向和垂直布局的简便方式。在呈现框架中，[FlowDirection](#) 属性可用于定义布局。流方向模式包括：

- *LeftToRight* - 适用于拉丁语和东亚语等语言的水平布局。
- *RightToLeft* - 适用于阿拉伯语和希伯来语等语言的双向布局。

## 开发可本地化的应用程序

编写供全球使用的应用程序时，应牢记应用程序必须可本地化。以下主题指出了若干注意事项。

### 多语言用户界面

多语言用户界面 (MUI) 是用于在 UI 中切换语言的 Microsoft 支持。WPF 应用程序使用程序集模型来支持 MUI。一个应用程序包含非特定语言程序集和与语言相关的附属资源程序集。入口点是主程序集中的托管 .EXE。WPF 资源加载程序采用 Framework 的资源管理器来支持资源查找和回退。多个语言附属程序集使用同一个主程序集。加载的资源程序集取决于当前线程的 [CurrentUICulture](#)。

### 可本地化的用户界面

UI。UI 并使用 C# 等编程语言响应用户交互。

从资源方面来看，UI 属于资源元素范畴，因此，其最终分发格式必须可本地化，以支持国际语言。由于 XAML 无法处理事件，因此，许多 XAML 应用程序都包含用于执行此操作的代码块。有关详细信息，请参阅 [WPF 中的 XAML](#)。将 XAML 文件标记为 XAML 的 BAML 形式时，会剥除代码并将其编译为不同的二进制文件。BAML 形式的 XAML 文件、图像以及其他类型的托管资源对象将嵌入附属资源程序集中，该程序集可本地化为其他语言，如果不需要进行本地化，以上各项就会嵌入主程序集中。

#### ① 备注

WPF 应用程序支持所有 FrameworkCLR 资源，包括字符串表、图像等。

# 生成可本地化的应用程序

本地化是指调整 UI 以适应不同的区域性。若要使 WPF 应用程序可本地化，开发人员需要将所有可本地化的资源生成一个资源程序集。该资源程序集本地化为不同语言，代码隐藏功能使用资源管理 API 进行加载。WPF 应用程序所需的文件之一是项目文件 (.proj)。应用程序中使用的所有资源都应包括在项目文件中。以下 .csproj 文件示例演示如何执行此操作。

XML

```
<Resource Include="data\picture1.jpg"/>
<EmbeddedResource Include="data\stringtable.en-US.resx"/>
```

若要在应用程序中使用资源，请实例化 [ResourceManager](#) 并加载要使用的资源。下面的示例演示如何执行此操作。

C#

```
void OnClick(object sender, RoutedEventArgs e)
{
    ResourceManager rm = new ResourceManager ("MySampleApp.data.stringtable",
        Assembly.GetExecutingAssembly());
    Text1.Text = rm.GetString("Message");
}
```

# 在本地化的应用程序中使用 ClickOnce

ClickOnce 是 Visual Studio 2005 附带的新 Windows 窗体部署技术。通过该技术可安装应用程序和升级 Web 应用程序。对使用 ClickOnce 部署的应用程序进行本地化后，只能在本地化的区域性中查看该应用程序。例如，如果将已部署的应用程序本地化为日语，则只能在日语版 Microsoft Windows 上查看该应用程序，而不能在英语版 Windows 上查看。由于日语用户经常运行英语版本的 Windows，因此这常常会出现问题。

此问题的解决方案是设置非特定语言回退特性。应用程序开发人员可选择从主程序集中删除资源，并指定可在特定区域性对应的附属程序集中找到该资源。若要控制此过程，请使用 [NeutralResourcesLanguageAttribute](#)。[NeutralResourcesLanguageAttribute](#) 类的构造函数包含两个签名，其中一个签名使用 [UltimateResourceFallbackLocation](#) 参数指定 [ResourceManager](#) 应在主程序集还是附属程序集中提取回退资源。下面的示例演示如何使用此特性。对于最终回退位置，该代码会导致 [ResourceManager](#) 在当前执行的程序集的目录的“de”子目录中查找资源。

C#

```
[assembly: NeutralResourcesLanguageAttribute(  
    "de" , UltimateResourceFallbackLocation.Satellite)]
```

## 另请参阅

- [WPF 全球化和本地化概述](#)

# 使用自动布局概述

项目 • 2022/09/27

本主题介绍适用于开发人员的、有关如何编写 WPF 应用程设计的指导原则。

## 使用自动布局的优点

由于 WPF 演示系统非常强大、灵活，可以利用它布局应用程序中的元素，这些元素可进行调整以适应不同语言的要求。下面列出自动布局的部分优点。

- UI 可通过任意语言正常显示。
- 减少了文本转换完后重新调整控件位置和大小的需要。
- 减少了重新调整窗口大小的需要。
- UI 布局可通过任意语言正确呈现。
- 本地化过程可缩减为与进行字符串转换差不多。

## 自动布局和控件

利用自动布局，应用程序可以自动调整控件大小。例如，控件可按字符串长度相应改变。利用此功能，本地化人员可转换字符串，而无需再调整控件大小以适应转换后的文本。下面的示例创建一个带有英文内容的按钮。

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Panel1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">My name is Hope.</Button>
    </DockPanel>
</Window>
```

在此示例中，只需更改文本即可生成一个西班牙文按钮。例如，

XAML

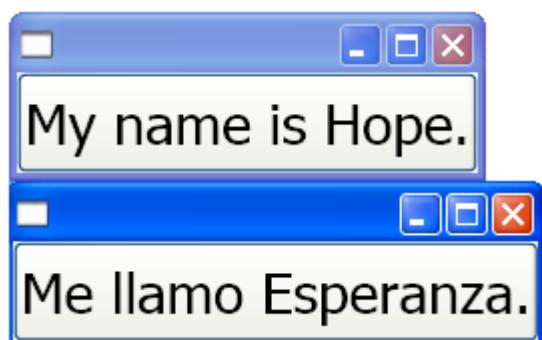
```

<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Panel1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">Me llamo Esperanza.</Button>
    </DockPanel>
</Window>

```

下图显示代码示例的输出：



## 自动布局和编码标准

使用自动布局方法需要一组用于生成可完全本地化的 UI 的编码及设计标准和规则。下列准则可帮助完成自动布局编码。

### 不使用绝对位置

- 不使用 `Canvas`，因为它会以绝对方式定位元素。
- 使用 `DockPanel`、`StackPanel` 和 `Grid` 定位控件。

有关各种面板类型的介绍，请参阅[面板概述](#)。

### 不设定固定的窗口大小

- 请使用 `Window.SizeToContent`。例如：

XAML
<pre> &lt;StackPanel     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" </pre>

```
x:Class="GridLoc.Panel"
```

```
>
```

## 添加 FlowDirection

- 将 `FlowDirection` 添加到应用程序的根元素中。

WPF 提供一种支持水平、双向和垂直布局的简便方式。在呈现框架中，`FlowDirection` 属性可用于定义布局。流方向模式包括：

- `FlowDirection.LeftToRight` (LrTb) - 适用于拉丁语和东亚语等语言的水平布局。
- `FlowDirection.RightToLeft` (RlTb) - 适用于阿拉伯语和希伯来语等语言的双向布局。

## 使用复合字体而不是实体字体

- 使用复合字体，则无需本地化 `FontFamily` 属性。
- 开发人员可以使用以下字体之一，也可以创建自己的字体。
  - Global User Interface
  - Global San Serif
  - Global Serif

## 添加 `xml:lang`

- 在 UI 的根元素中添加 `xml:lang` 特性，如用于英文应用程序的 `xml:lang="en-US"`。
- 因为复合字体使用 `xml:lang` 来确定要使用的字体，请将此属性设置为支持多语言方案。

# 自动布局和网格

`Grid` 元素对于自动布局很有用，因为开发人员可以利用它来定位元素。`Grid` 控件可以使用列和行排列方式在其子元素中分发可用空间。UI 元素可跨多个单元格，并且可在网格内再设网格。由于可以通过网格创建和定位复杂的 UI，因此网格很有用。下面的示例演示使用网格来定位某些按钮和文本。请注意，单元格的高度和宽度设置为 `Auto`；因此，包含带图像按钮的单元格会调整为适应该图像。

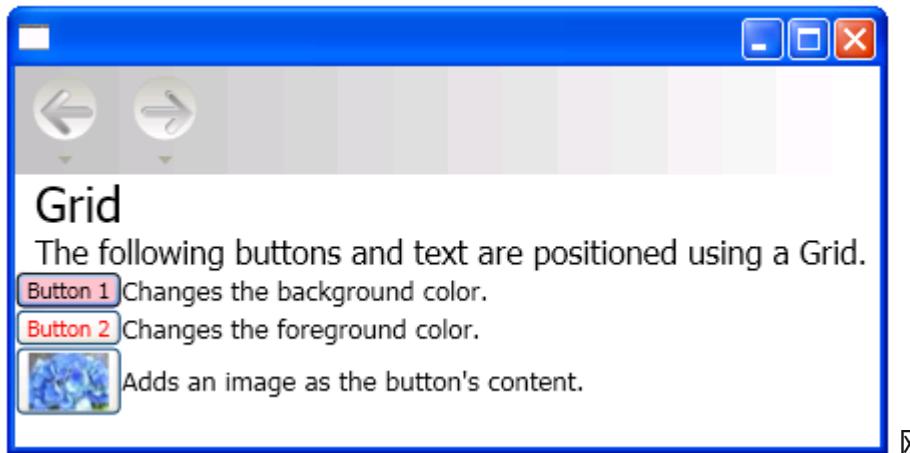
### XAML

```
<Grid Name="grid" ShowGridLines ="false">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
```

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

<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="0" FontSize="24">Grid
</TextBlock>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="1" FontSize="12" Grid.ColumnSpan="2">The following buttons and text are positioned using a Grid.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="2" Background="Pink" BorderBrush="Black" BorderThickness="10">Button 1
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="2" FontSize="12" VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the background color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3" Foreground="Red">Button 2
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="3" FontSize="12" VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the foreground color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="4">
    <Image Source="data\flower.jpg"/>
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="4" FontSize="12" VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Adds an image as the button's content.
</TextBlock>
</Grid>
```

下图显示了以上代码生成的网格。



网格

## 使用 IsSharedSizeScope 属性的自动布局和网格

Grid 可用于在可本地化的应用程序中创建可根据内容进行调整的控件。不过，有时可能希望控件无论包含什么内容都可以保持特定大小。例如，对于“确定”、“取消”和“浏览”按钮，可能不希望按钮根据内容调整大小。在这种情况下，`Grid.IsSharedSizeScope` 附加属性对于在多个网格元素中共享同样的大小很有用。下面的示例演示如何在多个 Grid 元素中共享列和行的大小数据。

XAML

```
<StackPanel Orientation="Horizontal" DockPanel.Dock="Top">
    <Button Click="setTrue" Margin="0,0,10,10">Set
        IsSharedSizeScope="True"</Button>
    <Button Click="setFalse" Margin="0,0,10,10">Set
        IsSharedSizeScope="False"</Button>
</StackPanel>

<StackPanel Orientation="Horizontal" DockPanel.Dock="Top">

    <Grid ShowGridLines="True" Margin="0,0,10,0">
        <Grid.ColumnDefinitions>
            <ColumnDefinition SharedSizeGroup="FirstColumn"/>
            <ColumnDefinition SharedSizeGroup="SecondColumn"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" SharedSizeGroup="FirstRow"/>
        </Grid.RowDefinitions>

        <Rectangle Fill="Silver" Grid.Column="0" Grid.Row="0" Width="200"
Height="100"/>
        <Rectangle Fill="Blue" Grid.Column="1" Grid.Row="0" Width="150"
Height="100"/>

        <TextBlock Grid.Column="0" Grid.Row="0" FontWeight="Bold">First
Column</TextBlock>
        <TextBlock Grid.Column="1" Grid.Row="0" FontWeight="Bold">Second
Column</TextBlock>
    </Grid>
```

```
<Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition SharedSizeGroup="FirstColumn"/>
        <ColumnDefinition SharedSizeGroup="SecondColumn"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" SharedSizeGroup="FirstRow"/>
    </Grid.RowDefinitions>

    <Rectangle Fill="Silver" Grid.Column="0" Grid.Row="0"/>
    <Rectangle Fill="Blue" Grid.Column="1" Grid.Row="0"/>

    <TextBlock Grid.Column="0" Grid.Row="0" FontWeight="Bold">First
Column</TextBlock>
    <TextBlock Grid.Column="1" Grid.Row="0" FontWeight="Bold">Second
Column</TextBlock>
</Grid>

</StackPanel>

<TextBlock Margin="10" DockPanel.Dock="Top" Name="txt1"/>
```

## ① 备注

有关完整的代码示例，请参阅[在网格之间共享大小调整属性](#)。

## 另请参阅

- [WPF 的全球化](#)
- [使用自动布局创建按钮](#)
- [使用网格进行自动布局](#)

# 本地化特性和注释

项目 · 2023/02/06

WPF 本地化 API 用于指示哪些资源要进行本地化。任意形式的注释是应用程序作者希望包含的任何信息。

## 添加本地化注释

如果标记应用程序作者对 XAML 中的特定元素具有一定要求（如对文本长度、字体系列或字号进行约束），则他们可以借助于 XAML 代码中的注释将此信息传达给本地化人员。下面是用来向源代码中添加注释的过程：

1. 应用程序开发人员向 XAML 源代码中添加本地化注释。
2. 在生成过程中，可以在 .proj 文件中指定是将任意形式的本地化注释留在程序集中、去掉部分注释还是去掉所有注释。去掉的注释放在一个单独的文件中。可以使用 `LocalizationDirectivesToLocFile` 标记指定你的选项，例如：

```
<LocalizationDirectivesToLocFile>value</LocalizationDirectivesToLocFile>
```
3. 可以分配的值包括：
  - **None** - 注释和特性都保留在程序集中，不会生成单独的文件。
  - **CommentsOnly** - 仅从程序集中去掉注释，并将它们放在单独的 LocFile 中。
  - **All** - 从程序集中去掉注释和特性，并将它们两者都放在单独的 LocFile 中。
4. 从 BAML 中提取可本地化资源时，可本地化性特性将由 BAML 本地化 API 保留。
5. 以后可以将仅包含任意形式的注释的本地化注释文件合并到本地化过程中。

以下示例演示如何将本地化注释添加到 XAML 文件中。

```
<TextBlock x:Id = "text01"
    FontFamily = "Microsoft Sans Serif"
    FontSize = "12"
    Localization.Attributes = "$Content (Unmodifiable Readable Text)
        FontFamily (Unmodifiable Readable)"
    Localization.Comments = "$Content (Trademark)">
```

```
FontSize (Trademark font size)" >
```

```
Microsoft
```

```
</TextBlock>
```

在上一示例中，Localization.Attributes 部分包含本地化特性，Localization.Comments 部分包含任意形式的注释。下表显示了特性和注释及其对于本地化人员的含义。

本地化特性	含义
\$Content ( 不可修改的可读文本 )	无法修改 TextBlock 元素的内容。本地化人员不能更改单词“Microsoft”。内容对本地化人员可见（可读）。内容为文本类别。
FontFamily ( 不可修改但可读 )	无法更改 TextBlock 元素的字体系列属性，但该属性对本地化人员可见。

本地化任意形式的注释	含义
\$Content ( 商标 )	应用程序作者告知本地化人员 TextBlock 元素中的内容是商标。
FontSize ( 商标字号 )	应用程序作者指示字号属性应遵循标准商标大小。

## 可本地化特性

Localization.Attributes 中的信息包含对的列表：目标值名称和关联的可本地化值。目标名称可以是属性名称或特殊 \$Content 名称。如果是属性名称，则目标值为属性的值。如果是 \$Content，则目标值为元素的内容。

有三种类型的属性：

- **类别。** 它指定是否可以从本地化工具修改值。请参阅 [Category](#)。
- **Readability。** 它指定本地化工具是否可读取（和显示）值。请参阅 [Readability](#)。
- **Modifiability。** 它指定本地化工具是否允许修改值。请参阅 [Modifiability](#)。

这些特性可以按照由空格分开的任何顺序进行指定。如果指定了重复特性，则最后一个特性将替代前面的特性。例如，Localization.Attributes = "Unmodifiable Modifiable" 会将 Modifiability 设置为 Modifiable，因为最后一个值是 Modifiable。

Modifiability 和 Readability 的名称已经揭示了自身的含义。“Category”特性提供的预定义类别可帮助本地化人员翻译文本。类别（如 Text、Label 和 Title）为本地化人员提供了有关如何翻译文本的信息。另外，还有一些特殊类别：None、Inherit、Ignore 和 NeverLocalize。

下表显示了这些特殊类别的含义。

Category	含义
无	没有为目标值定义类别。
继承	目标值从其父级继承其类别。
忽略	在本地化过程中将忽略目标值。 Ignore 仅影响当前值。 它不会影响子节点。
NeverLocalize	无法对当前值进行本地化。 此类别由元素的子级继承。

## 本地化注释

Localization.Comments 包含与目标值有关的任意形式的字符串。 应用程序开发人员可以添加一些信息，以便为本地化人员提供有关如何翻译应用程序文本的提示。 注释格式可以是由“()”括起来的任何字符串。 使用“\”转义字符。

## 另请参阅

- [WPF 的全球化](#)
- [使用自动布局创建按钮](#)
- [使用网格进行自动布局](#)
- [对应用程序进行本地化](#)

# WPF 中的双向功能概述

项目 • 2022/09/27

与其他任何开发平台不同，WPF 具有许多支持双向内容快速开发的功能，例如，同一文档中混合了从左到右和从右到左的数据。同时，WPF 也为需要双向功能的用户（如阿拉伯语和希伯来语用户）带来了绝佳的体验。

以下各节结合一些示例阐释了如何获得双向内容的最佳显示效果，并对许多双向功能进行了说明。大多数示例使用的是 XAML，但你可以轻松地将这些概念应用于 C# 或 Microsoft Visual Basic 代码。

## FlowDirection

[FlowDirection](#) 是在 WPF 应用程序中定义内容流方向的基本属性。此属性可设置为以下两个枚举值之一：[LeftToRight](#) 或 [RightToLeft](#)。此属性可用于从 [FrameworkElement](#) 继承的所有 WPF 元素。

以下示例将设置 [TextBox](#) 元素的流方向。

### 从左向右的流方向

XAML

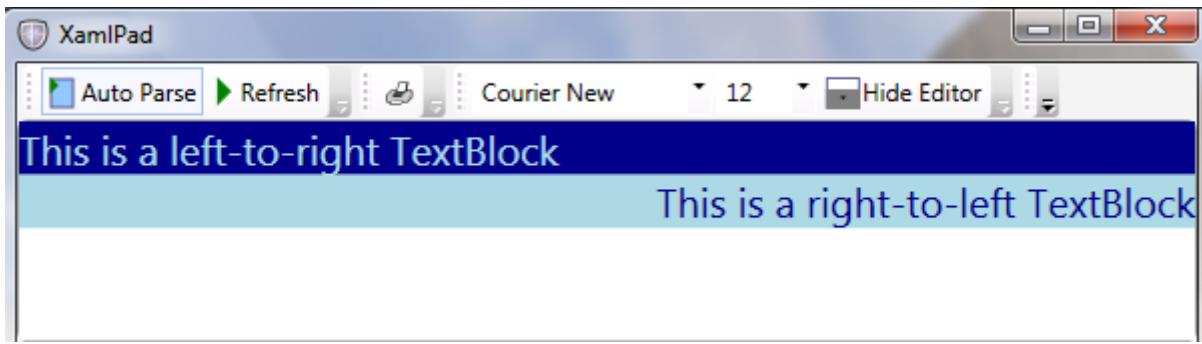
```
<TextBlock Background="DarkBlue" Foreground="LightBlue"
FontSize="20" FlowDirection="LeftToRight">
    This is a left-to-right TextBlock
</TextBlock>
```

### 从右向左的流方向

XAML

```
<TextBlock Background="LightBlue" Foreground="DarkBlue"
FontSize="20" FlowDirection="RightToLeft">
    This is a right-to-left TextBlock
</TextBlock>
```

下图显示了前面代码的呈现方式。



用户界面 (UI) 树内的元素将从其容器继承 `FlowDirection`。在下面的示例中，`TextBlock` 位于 `Grid` 中，而后者位于 `Window` 中。为 `Window` 设置 `FlowDirection` 意味着还将为 `Grid` 和 `TextBlock` 设置该属性。

以下示例演示如何设置 `FlowDirection`。

#### XAML

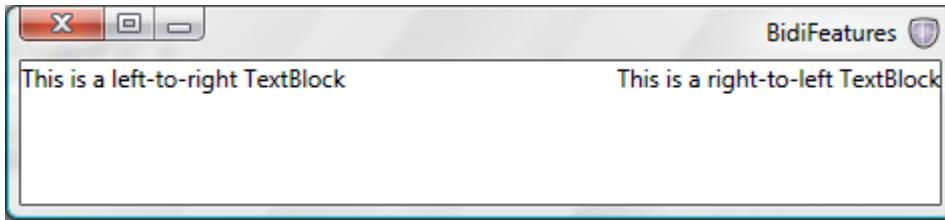
```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="FlowDirectionApp.Window1"
    Title="BidiFeatures" Height="200" Width="700"
    FlowDirection="RightToLeft">

    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition/>
            <ColumnDefinition/>
        </Grid.ColumnDefinitions>
        <TextBlock Grid.Column="0" >
            This is a right-to-left TextBlock
        </TextBlock>

        <TextBlock Grid.Column="1" FlowDirection="LeftToRight">
            This is a left-to-right TextBlock
        </TextBlock>
    </Grid>
</Window>
```

顶级 `Window` 包含 `RightToLeftFlowDirection`，因此其中包含的所有元素也都继承同一 `FlowDirection`。对于要替代指定的 `FlowDirection` 的元素，必须添加显式方向更改，如上一示例中的第二个 `TextBlock`，该控件更改为 `LeftToRight`。如果未定义 `FlowDirection`，将应用默认的 `LeftToRight`。

下图显示了上一个示例的输出：



## FlowDocument

HTML、Win32 和 Java 等许多开发平台都对双向内容开发提供特殊支持。标记语言（如 HTML）为内容编写器提供了在任意所需方向显示文本时必需的标记，如 HTML 4.0 标记、采用“rtl”或“ltr”作为值的“dir”等。此标记类似于 [FlowDirection](#) 属性，但 [FlowDirection](#) 属性使用了一种更高级的方法来布局文本内容，并可用于除文本以外的内容。

在 UI 中，它是一种可以托管文本、表、图像和其他元素的组合的元素。以下各节中的示例均使用此元素。

可通过多种方式将文本添加到 [FlowDocument](#)。其中一种简单方法就是使用 [Paragraph](#)，它是用于对文本等内容进行分组的块级元素。为了将文本添加到内联级元素，这些示例使用了 [Span](#) 和 [Run](#)。[Span](#) 是用于对其他内联元素进行分组的内联级流内容元素，而 [Run](#) 是用于包含一系列无格式文本的内联级流内容元素。一个 [Span](#) 可以包含多个 [Run](#) 元素。

第一个文档示例包含的文档具有很多个网络共享名，例如 `\server1\folder\file.ext`。无论此网络链接是包含在阿拉伯语文档还是英语文档中，建议始终以相同的方式显示它。下图演示了如何使用 [Span](#) 元素，并显示了阿拉伯语 [RightToLeft](#) 文档中的链接：

The screenshot shows the XamlPad application interface. The title bar says "XamlPad". The toolbar includes "Auto Parse", "Refresh", "Courier New", "12", "Hide Editor", and "100%". The main area displays XAML code for a page with a flow direction of "RightToLeft". Inside a paragraph, there is a span element with a "RightToLeft" flow direction containing Arabic text: "ستجد الملف هنا:\\server1\\filename\\filename1.txt ثم باقى النص!". Below the code, a status bar says "Done. Markup saved to \"C:\Program Files\XamlPad\XamlPad\_Saved.xaml\"."

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:
                \\server1\filename\filename1.txt
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

Done. Markup saved to "C:\Program Files\XamlPad\XamlPad\_Saved.xaml".

由于文本是 [RightToLeft](#)，因此所有特殊字符（如“＼”）都按从右到左的顺序分隔文本。这样会导致链接显示顺序不正确，因此，若要解决此问题，必须嵌入文本以保留按 [LeftToRight](#) 显示的单独的 [Run](#)。除了为每种语言提供单独的 [Run](#) 之外，还可使用一种更好的方法来解决此问题，即，将不常用的英语文本嵌入到较大的阿拉伯语 [Span](#) 中。

下图通过使用 [Span](#) 元素中嵌入的 [Run](#) 元素说明了这一点：

The screenshot shows the XamlPad application interface. The title bar says "XamlPad". The toolbar includes "Auto Parse", "Refresh", "Courier New", "12", "Hide Editor", and "100%". The main area displays XAML code for a page with a flow direction of "RightToLeft". Inside a paragraph, there is a span element with a "RightToLeft" flow direction containing a run element with a "LeftToRight" flow direction, which contains the Arabic text: "ستجد الملف هنا:\\server1\\filename\\filename1.txt ثم باقى النص!". Below the code, a status bar says "Done. Markup saved to \"C:\Program Files\XamlPad\XamlPad\_Saved.xaml\"."

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:
                <Run FlowDirection="LeftToRight">\\server1\filename\filename1.txt</Run>
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

Done. Markup saved to "C:\Program Files\XamlPad\XamlPad\_Saved.xaml".

以下示例演示如何在文档中使用 [Run](#) 和 [Span](#) 元素。

## XAML

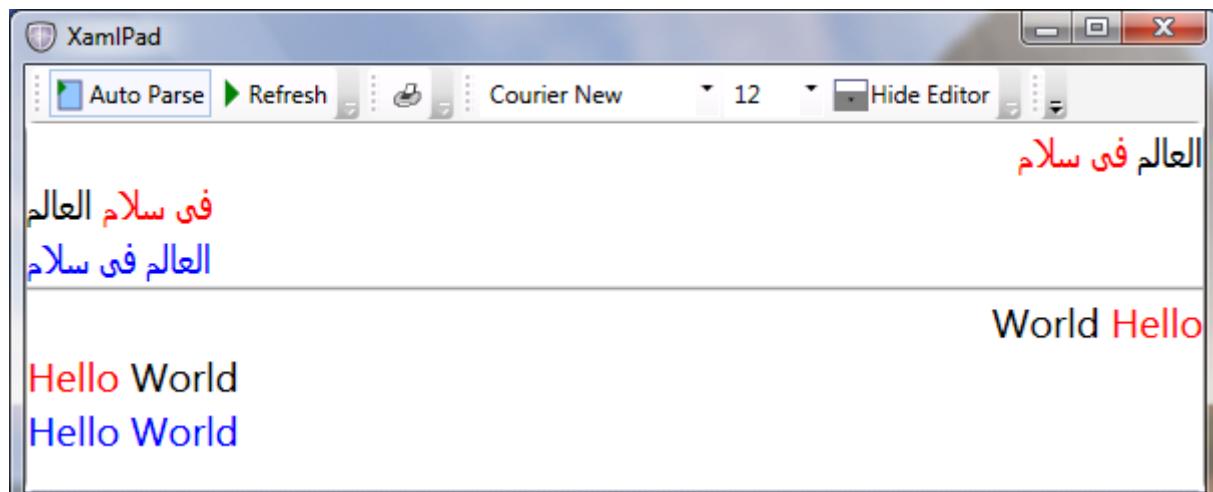
```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">

    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" >
                ستجد الملف هنا:
                <Run FlowDirection="LeftToRight">
                    \\server1\filename\filename1.txt</Run>
                ثم باقى النص!
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

## Span 元素

Span 元素用作具有不同流方向的文本之间的边界分隔符。甚至具有相同流方向的 Span 元素也被视为具有不同的双向范围，这意味着 Span 元素按容器的 FlowDirection 进行排序，只有 Span 元素内的内容才遵循 Span 的 FlowDirection。

下图显示了几个 TextBlock 元素的流方向。



以下示例演示如何使用 Span 和 Run 元素生成上图中显示的结果。

## XAML

```
<Page xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
    <StackPanel>
        <TextBlock FontSize="20" FlowDirection="RightToLeft">
            <Run FlowDirection="LeftToRight">العالم فى سلام</Run>
        </TextBlock>
    </StackPanel>
</Page>
```

```

        <Run FlowDirection="LeftToRight" Foreground="Red" >فی سلام</Run>
    </TextBlock>

    <TextBlock FontSize="20" FlowDirection="LeftToRight">
        <Run FlowDirection="RightToLeft">العالمن</Run>
        <Run FlowDirection="RightToLeft" Foreground="Red" >فی سلام</Run>
    </TextBlock>

    <TextBlock FontSize="20" Foreground="Blue">العالمن فی سلام</TextBlock>

    <Separator/>

    <TextBlock FontSize="20" FlowDirection="RightToLeft">
        <Span Foreground="Red" FlowDirection="LeftToRight">Hello</Span>
        <Span FlowDirection="LeftToRight">World</Span>
    </TextBlock>

    <TextBlock FontSize="20" FlowDirection="LeftToRight">
        <Span Foreground="Red" FlowDirection="RightToLeft">Hello</Span>
        <Span FlowDirection="RightToLeft">World</Span>
    </TextBlock>

    <TextBlock FontSize="20" Foreground="Blue">Hello World</TextBlock>

</StackPanel>

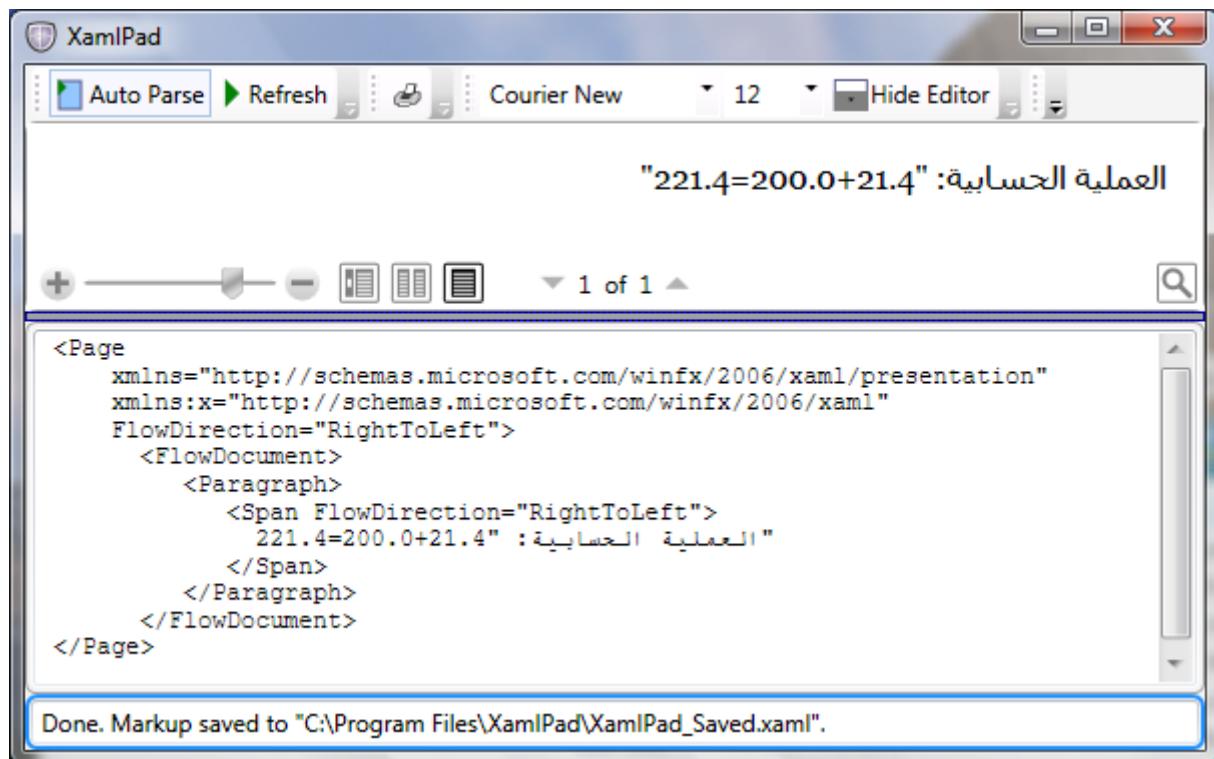
</Page>

```

在此示例的 `TextBlock` 元素中，`Span` 元素根据其父级的 `FlowDirection` 进行布局，但每个 `Span` 元素中的文本根据其自己的 `FlowDirection` 流动。这适用于拉丁语和阿拉伯语，也适用于任何其他语言。

## 添加 `xml:lang`

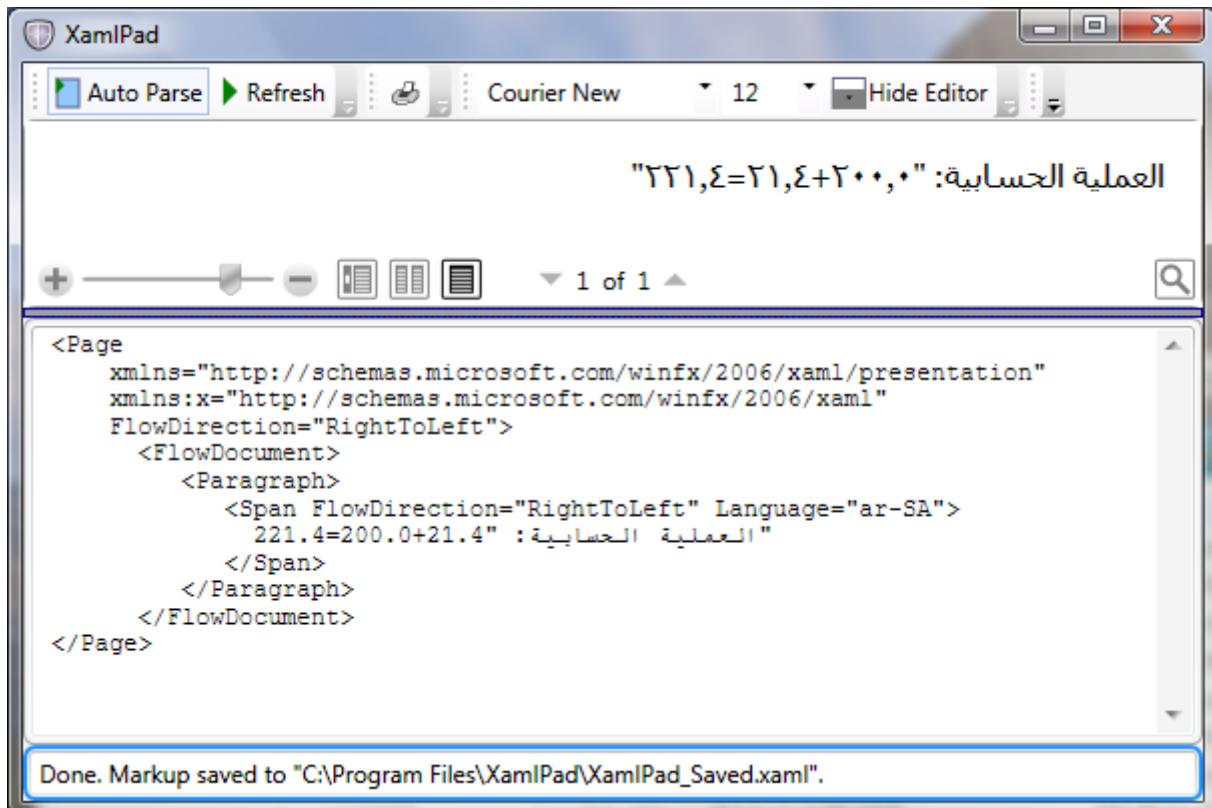
下图显示了另一个示例，该示例使用数字和算术表达式，如 "200.0+21.4=221.4"。请注意，仅设置了 `FlowDirection`。



此应用程序的用户将对输出感到失望，即使 `FlowDirection` 正确，但数字的形状不是阿拉伯语数字应有的形状。

XAML 元素可以包括 XML 属性 (`xml:lang`)，该属性用于定义每个元素的语言。XAML 还支持 XML 语言原则，即，应用于树中父元素的 `xml:lang` 值将用于子元素。在上面的示例中，由于未对 `Run` 元素或其任意顶级元素设置语言，因此使用了默认的 `xml:lang`（对于 XAML 为 `en-US`）。Windows Presentation Foundation (WPF) 的内部数字整理算法选择对应语言（在本示例中为英语）中的数字。若要正确呈现阿拉伯语数字，需要设置 `xml:lang`。

下图演示添加了 `xml:lang` 的示例。



以下示例将向应用程序添加 `xml:lang`。

XAML

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    FlowDirection="RightToLeft">
    <FlowDocument>
        <Paragraph>
            <Span FlowDirection="RightToLeft" Language="ar-SA">
                221.4=200.0+21.4"
            </Span>
        </Paragraph>
    </FlowDocument>
</Page>
```

请注意，根据目标区域的不同，许多语言具有不同的 `xml:lang` 值，例如，`"ar-SA"` 和 `"ar-EG"` 表示阿拉伯语的两个变体。上述示例说明你需要同时定义 `xml:lang` 和 `FlowDirection` 值。

## 非文本元素的 `FlowDirection`

`FlowDirection` 不仅定义文本在文本元素中的流动方式，还定义几乎所有其他 UI 元素的流方向。下图显示了一个 `ToolBar`，该控件使用水平 `LinearGradientBrush` 以从左到右渐变绘制其背景。

The screenshot shows the XamlPad application interface. At the top, there's a toolbar with buttons for Auto Parse, Refresh, and Hide Editor. The font is set to Courier New at size 12, and the zoom is at 100%. Below the toolbar, a toolbar control is displayed with four buttons labeled Button1, Button2, Button3, and Button4. The background of the toolbar is defined by a LinearGradientBrush with four gradient stops: DarkRed, DarkBlue, LightBlue, and White. The XAML code for this control is shown in the main editor area:

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

<ToolBar>
    <ToolBar.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,1">
            <LinearGradientBrush.GradientStops>
                <GradientStop Color="DarkRed" Offset="0" />
                <GradientStop Color="DarkBlue" Offset="0.3" />
                <GradientStop Color="LightBlue" Offset="0.6" />
                <GradientStop Color="White" Offset="1" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </ToolBar.Background>

    <Button FontSize="12" Foreground="White">Button1</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button2</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button3</Button>
    <Rectangle Width="20"/>
    <Button FontSize="12" Foreground="White">Button4</Button>
    <Rectangle Width="20"/>
</ToolBar>

</Page>
```

At the bottom of the editor, a message says "Done. Markup saved to 'C:\Program Files\XamlPad\XamlPad\_Saved.xaml'".

将 `FlowDirection` 设置为 `RightToLeft` 后，不仅 `ToolBar` 按钮是从右到左排列的，甚至 `LinearGradientBrush` 也将其偏移量重新调整为从右到左流动。

下图显示了 `LinearGradientBrush` 的重新排列。

The screenshot shows the XamlPad application interface. The toolbar control now has its `FlowDirection` property set to `RightToLeft`. As a result, the four buttons are arranged from right to left: Button4, Button3, Button2, and Button1. The background of the toolbar is defined by a `LinearGradientBrush` with four gradient stops: White, LightBlue, DarkBlue, and DarkRed. The XAML code for this control is shown in the main editor area:

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

<ToolBar FlowDirection="RightToLeft">
    <ToolBar.Background>
```

以下示例绘制了一个 `RightToLeftToolBar`。若要从左到右绘制，请删除 `ToolBar` 的 `FlowDirection` 属性。

The screenshot shows the XamlPad application interface with a tab labeled "XAML". The XAML code for the `RightToLeftToolBar` is displayed:

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

<ToolBar FlowDirection="RightToLeft" Height="50" DockPanel.Dock="Top">
    <ToolBar.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,1">
```

```
<LinearGradientBrush.GradientStops>
  <GradientStop Color="DarkRed" Offset="0" />
  <GradientStop Color="DarkBlue" Offset="0.3" />
  <GradientStop Color="LightBlue" Offset="0.6" />
  <GradientStop Color="White" Offset="1" />
</LinearGradientBrush.GradientStops>
</LinearGradientBrush>
</ToolBar.Background>

<Button FontSize="12" Foreground="White">Button1</Button>
<Rectangle Width="20"/>
<Button FontSize="12" Foreground="White">Button2</Button>
<Rectangle Width="20"/>
<Button FontSize="12" Foreground="White">Button3</Button>
<Rectangle Width="20"/>
<Button FontSize="12" Foreground="White">Button4</Button>
<Rectangle Width="20"/>
</ToolBar>
</Page>
```

## FlowDirection 异常

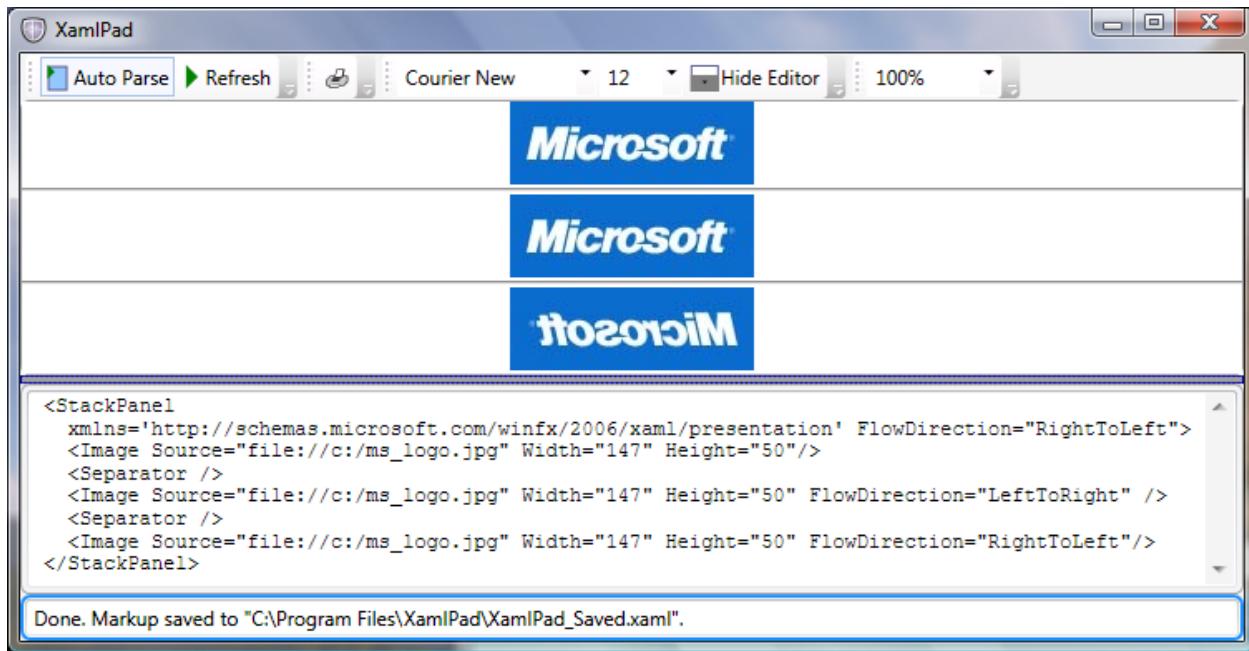
在少数情况下，[FlowDirection](#) 会不按预期方式运行。 本部分介绍其中两种异常。

### 图像

[Image](#) 表示用于显示图像的控件。 在 XAML 中，该控件可与 [Source](#) 属性一起使用，后者用于定义要显示的 [Image](#) 的统一资源标识符 (URI)。

与其他 UI 元素不同，[Image](#) 不会从容器继承 [FlowDirection](#)。 但是，如果将 [FlowDirection](#) 显式设置为 [RightToLeft](#)，则会以水平翻转方式显示 [Image](#)。 这可作为一种便捷功能提供给双向内容的开发人员，因为在某些情况下，水平翻转图像会达到所需的效果。

下图显示翻转后的 [Image](#)。



以下示例演示 `Image` 未能从包含它的 `StackPanel` 中继承 `FlowDirection`。

### ① 备注

C:\ 驱动器上必须具有名为 ms\_logo.jpg 的文件才能运行此示例。

#### XAML

```
<StackPanel  
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'  
    FlowDirection="RightToLeft">  
  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50"/>  
    <Separator Height="10"/>  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50" FlowDirection="LeftToRight" />  
    <Separator Height="10"/>  
    <Image Source="file:///c:/ms_logo.jpg"  
        Width="147" Height="50" FlowDirection="RightToLeft"/>  
</StackPanel>
```

### ① 备注

下载文件中包含的内容是 `ms_logo.jpg` 文件。该代码假定 `.jpg` 文件不在项目中，而是位于 C:\ 驱动器中的某个位置。必须将 `.jpg` 从项目文件复制到 C:\ 驱动器或更改代码才能在项目内查找该文件。为此，请将 `Source="file:///c:/ms_logo.jpg"` 更改为 `Source="ms_logo.jpg"`。

## 路径

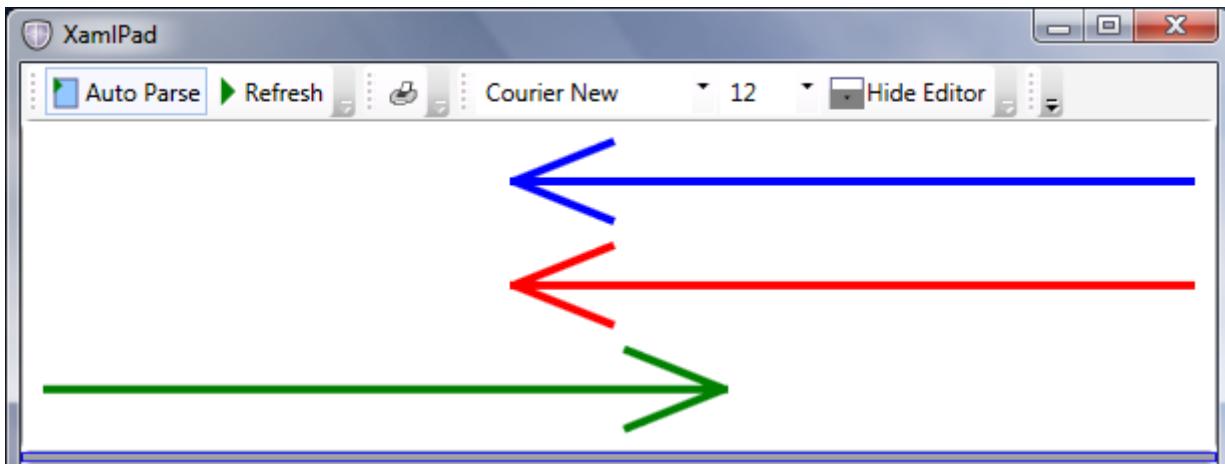
除 [Image](#) 外，另一个值得关注的元素是 [Path](#)。 [Path](#) 是可用于绘制一系列连接的直线和曲线的对象。就其 [FlowDirection](#) 而言，它的行为方式类似于 [Image](#)，例如它的 [RightToLeftFlowDirection](#) 是其 [LeftToRight](#) 的水平镜像。但是，与 [Image](#) 不同的是，[Path](#) 从容器中继承其 [FlowDirection](#)，因此用户无需显式指定它。

下面的示例使用 3 条线绘制简单的箭头。第一个箭头从 [StackPanel](#) 继承 [RightToLeft](#) 流方向，以便从右侧的根测量其起点和终点。第二个箭头具有显式的 [RightToLeftFlowDirection](#)，也从右侧开始。但第三个箭头的起始根位于左侧。有关绘图的详细信息，请参阅 [LineGeometry](#) 和 [GeometryGroup](#)。

### XAML

```
<StackPanel  
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'  
    FlowDirection="RightToLeft">  
  
    <Path Stroke="Blue" StrokeThickness="4">  
        <Path.Data>  
            <GeometryGroup >  
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />  
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />  
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />  
            </GeometryGroup>  
        </Path.Data>  
    </Path>  
  
    <Path Stroke="Red" StrokeThickness="4" FlowDirection="RightToLeft">  
        <Path.Data>  
            <GeometryGroup >  
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />  
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />  
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />  
            </GeometryGroup>  
        </Path.Data>  
    </Path>  
  
    <Path Stroke="Green" StrokeThickness="4" FlowDirection="LeftToRight">  
        <Path.Data>  
            <GeometryGroup >  
                <LineGeometry StartPoint="300,10" EndPoint="350,30" />  
                <LineGeometry StartPoint="10,30" EndPoint="352,30" />  
                <LineGeometry StartPoint="300,50" EndPoint="350,30" />  
            </GeometryGroup>  
        </Path.Data>  
    </Path>  
</StackPanel>
```

下图显示了上一个示例的输出，其中使用 [Path](#) 元素绘制了箭头：



[Image](#) 和 [Path](#) 是在容器内按特定方向布局 UI 元素的两个示例，[FlowDirection](#) 可与 [InkPresenter](#)（在图面上呈现墨迹）、[LinearGradientBrush](#) 和 [RadialGradientBrush](#) 等元素一起使用。每当模拟从左到右行为的内容需要从右到左行为（反之亦然）时，Windows Presentation Foundation (WPF) 就会提供该功能。

## 数字替换

一直以来，Windows 始终通过以下方式支持数字替换：允许对相同数字使用不同区域性形状的表示形式，但同时使这些数字的内部存储形式在不同区域设置之间保持统一；例如，数字虽然以其常见的十六进制值（如 0x40、0x41）存储，但却根据所选的语言进行显示。

这使应用程序无需将数值从一种语言转换为另一种语言，就可对它们进行处理；例如，用户可以在本地化的阿拉伯语 Windows 中打开 Microsoft Excel 电子表格，并会看到阿拉伯语形状的数字，但在欧洲版 Windows 中打开它时，会看到相同数字的欧洲表示形式。这对其他符号（如逗号分隔符和百分比符号）来说也是必需的，因为在同一文档中它们通常随数字一起出现。

Windows Presentation Foundation (WPF) 沿承了这一传统，并为此功能提供了进一步支持，以允许用户更好地控制何时以及如何使用替换。虽然此功能适用于任何语言，但它对双向内容尤其有用；由于应用程序可能会在各种区域性下运行，因此针对特定语言来设置数字形状通常是应用程序开发人员所面临的难题。

控制 Windows Presentation Foundation (WPF) 中的数字替换方式的核心属性是 [Substitution](#) 依赖属性。[NumberSubstitution](#) 类指定如何显示文本中的数字。它有三个定义其行为的公共属性。下面概括了其中的每个属性：

### CultureSource :

此属性指定如何确定数字的区域性。它使用三个 [NumberCultureSource](#) 枚举值之一。

- [Override](#) : 数字区域性是 [CultureOverride](#) 属性的区域性。

- Text：数字区域性是文本运行的区域性。在标记中，这将是 `xml:lang`，或其别名 `Language` 属性（`Language` 或 `Language`）。此外，它还是派生自 `FrameworkContentElement` 的类的默认值。这种类包括 `System.Windows.Documents.Paragraph`、`System.Windows.Documents.Table`、`System.Windows.Documents.TableCell` 等。
- User：数字区域性是当前线程的区域性。该属性是 `FrameworkElement` 的所有子类（如 `Page`、`Window` 和 `TextBlock`）的默认值。

#### CultureOverride：

仅当 `CultureSource` 属性设置为 `Override` 时才使用 `CultureOverride` 属性，否则将忽略。该属性指定数字区域性。默认值 `null` 被解释为 `en-US`。

#### Substitution：

此属性指定要执行的数字替换类型。它使用以下 `NumberSubstitutionMethod` 枚举值之一：

- `AsCulture`：替换方法根据数字区域性的 `NumberFormatInfo.DigitSubstitution` 属性决定。这是默认值。
- `Context`：如果数字区域性为阿拉伯语或波斯语区域性，则指定数字取决于上下文。
- `European`：数字始终呈现为欧洲数字。
- `NativeNational`：使用数字区域性的民族数字（由区域性的 `NumberFormat` 指定）呈现数字。
- `Traditional`：使用数字区域性的传统数字呈现数字。对于大多数区域性，这与 `NativeNational` 相同。但是，`NativeNational` 对某些阿拉伯语区域性会产生拉丁数字，而此值对所有阿拉伯语区域性产生阿拉伯数字。

这些值对双向内容开发人员意味着什么？在大多数情况下，开发人员可能只需要定义 `FlowDirection` 和每个文本 UI 元素的语言，例如 `Language="ar-SA"` 和 `NumberSubstitution` 逻辑负责根据正确的 UI 显示数字。以下示例演示如何在阿拉伯语版 Windows 中运行的 Windows Presentation Foundation (WPF) 应用程序中使用阿拉伯语和英语数字。

#### XAML

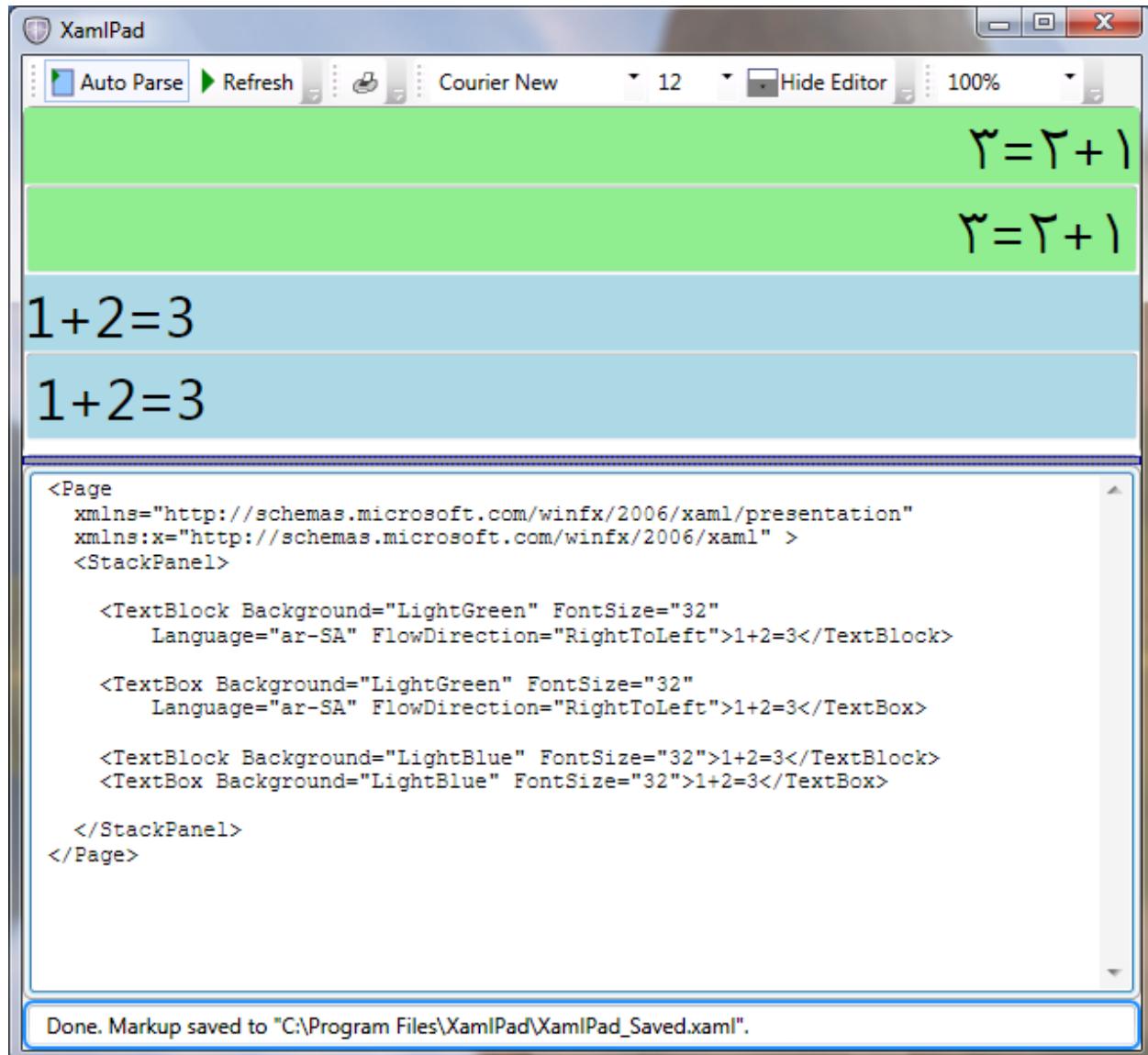
```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" >
    <StackPanel>
        <TextBlock Background="LightGreen" FontSize="32"
```

```

        Language="ar-SA" FlowDirection="RightToLeft">1+2=3</TextBlock>
    <TextBox Background="LightGreen" FontSize="32"
        Language="ar-SA" FlowDirection="RightToLeft">1+2=3</TextBox>
    <TextBlock Background="LightBlue" FontSize="32">1+2=3</TextBlock>
    <TextBlock Background="LightBlue" FontSize="32">1+2=3</TextBlock>
</StackPanel>
</Page>

```

下图演示了在阿拉伯语版的 Windows 中运行并显示阿拉伯语和英语数字时上一示例的输出：



在此示例中，[FlowDirection](#) 很重要，因为将 [FlowDirection](#) 改为设置成 [LeftToRight](#) 时会产生欧洲数字。以下各节介绍如何在整个文档内统一显示数字。如果未在阿拉伯语版 Windows 上运行此示例，所有数字都将显示为欧洲数字。

## 定义替换规则

在实际的应用程序中，可能需要以编程方式设置语言。例如，希望将 `xml:lang` 属性设置为系统 UI 所用的相同属性，或根据应用程序具体状态更改语言。

如果希望根据应用程序状态进行更改，请利用 Windows Presentation Foundation (WPF) 提供的其他功能。

首先，设置应用程序组件的 `NumberSubstitution.CultureSource="Text"`。 使用此设置可确保对于将“User”用作默认值的文本元素（如 `TextBlock`），设置不会来自 UI。

例如：

XAML

```
<TextBlock  
    Name="text1" NumberSubstitution.CultureSource="Text">  
    1234+5679=6913  
</TextBlock>
```

例如，在对应的 C# 代码中，将 `Language` 属性设置为 `"ar-SA"`。

C#

```
text1.Language = System.Windows.Markup.XmlLanguage.GetLanguage("ar-SA");
```

如果需要将 `Language` 属性设置为当前用户的 UI 语言，请使用下面的代码。

C#

```
text1.Language =  
System.Windows.Markup.XmlLanguage.GetLanguage(System.Globalization.CultureIn  
fo.CurrentCulture.IetfLanguageTag);
```

`CultureInfo.CurrentCulture` 表示当前线程在运行时所用的当前区域性。

最后的 XAML 示例应与下面的示例类似。

XAML

```
<Page x:Class="WindowsApplication.Window1"  
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
      Title="Code Sample" Height="300" Width="300"  
>  
    <StackPanel>  
      <TextBlock Language="ar-SA"  
                FlowDirection="RightToLeft">3=2+1 : عربى  
      </TextBlock>  
      <TextBlock Language="ar-SA"  
                FlowDirection="RightToLeft"  
                NumberSubstitution.Substitution="European">3=2+1 : عربى  
      </TextBlock>
```

```
</StackPanel>
</Page>
```

最后的 C# 示例应与下面的示例类似。

```
C#
namespace BidiTest
{
    public partial class Window1 : Window
    {

        public Window1()
        {
            InitializeComponent();

            string currentLanguage =
System.Globalization.CultureInfo.CurrentCulture.IetfLanguageTag;

            text1.Language =
System.Windows.Markup.XmlLanguage.GetLanguage(currentLanguage);

            if (currentLanguage.ToLower().StartsWith("ar"))
            {
                text1.FlowDirection = FlowDirection.RightToLeft;
            }
            else
            {
                text1.FlowDirection = FlowDirection.LeftToRight;
            }
        }
    }
}
```

下图显示了任一编程语言的窗口外观，该窗口显示阿拉伯语数字：



## 使用 Substitution 属性

Windows Presentation Foundation (WPF) 中的数字替换方式同时取决于文本元素的语言及其 `FlowDirection`。如果 `FlowDirection` 为从左到右，则会呈现欧洲数字。但是，如果数字的前面为阿拉伯语文本或者具有设置为“ar”的语言，且 `FlowDirection` 为 `RightToLeft`，则会改为呈现阿拉伯语数字。

但在某些情况下，建议创建统一的应用程序，例如适用于所有用户的欧洲数字。或者在具有特定 `Style` 的 `Table` 单元格中呈现阿拉伯语数字。执行此操作的一种简单方法是使用

## Substitution 属性。

在下面的示例中，第一个 `TextBlock` 未设置 `Substitution` 属性，因此算法将按预期方式显示阿拉伯语数字。但是，在第二个 `TextBlock` 中，替换设置为欧洲，替代了默认的阿拉伯语数字替换，此时将显示欧洲数字。

### XAML

```
<Page x:Class="WindowsApplication.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Code Sample" Height="300" Width="300"
>
    <StackPanel>
        <TextBlock Language="ar-SA"
            FlowDirection="RightToLeft">3=2+1 : عربى
        </TextBlock>
        <TextBlock Language="ar-SA"
            FlowDirection="RightToLeft"
            NumberSubstitution.Substitution="European">3=2+1 : عربى
        </TextBlock>
    </StackPanel>
</Page>
```

# 如何：对应用程序进行本地化

项目 • 2022/09/27

本教程介绍如何通过使用 LocBaml 工具创建本地化应用程序。

## ① 备注

LocBaml 工具不是可直接用于生产的应用程序。 它表示为一个示例，该示例使用某些本地化 API 并演示编写一个本地化工具的方法。

## 概述

本文提供逐步实现本地化应用程序的方法。首先，你要准备应用程序，以便可以提取要翻译的文本。文本被翻译之后，需要将翻译后的文本合并到原始应用程序的新副本中。

## 创建一个简单的应用程序

在此步骤中，你将准备要用于本地化的应用。Windows Presentation Foundation (WPF) 示例中提供了一个 HelloApp 示例，将用于本讨论中的代码示例。如果要使用此示例，请从 [LocBaml 工具示例](#) 下载 Extensible Application Markup Language (XAML) 文件。

1. 将应用程序开发到想要开始进行本地化的位置。
2. 在项目文件中指定开发语言，以便 MSBuild 生成主程序集和附属程序集（具有 .resources.dll 扩展的文件）以包含非特定语言资源。HelloApp 示例中的项目文件是 HelloApp.csproj。在该文件中，你将找到标识如下的开发语言：

```
<UICulture>en-US</UICulture>
```

3. 将 Uid 添加到 XAML 文件。Uid 用于跟踪对文件的更改并标识必须翻译的项。要将 Uid 添加到文件，请在项目文件上运行 updateuid：

```
msbuild -t:updateuid helloapp.csproj
```

若要验证没有缺少或重复的 Uid，请运行 checkuid：

```
msbuild -t:checkuid helloapp.csproj
```

运行 updateuid 之后，文件应包含 Uid。例如，在 HelloApp 的 Pane1.xaml 文件中，你应能找到下列内容：

XAML

```
<StackPanel x:Uid="StackPanel_1">
  <TextBlock x:Uid="TextBlock_1">Hello World</TextBlock>
  <TextBlock x:Uid="TextBlock_2">Goodbye World</TextBlock>
</StackPanel>
```

## 创建非特定语言资源附属程序集

将应用程序配置为生成非特定语言资源附属程序集后，将生成应用程序。这会生成主应用程序程序集，以及 LocBaml 本地化所需的非特定语言资源附属程序集。

若要生成应用程序，请执行以下操作：

1. 编译 HelloApp 以创建动态链接库 (DLL)：

```
msbuild helloapp.csproj
```

2. 新创建的主应用程序程序集 HelloApp.exe 创建在下列文件夹中：C:\HelloApp\Bin\Debug

3. 新创建的非特定语言资源附属程序集 HelloApp.resources.dll 创建在下列文件夹中：  
C:\HelloApp\Bin\Debug\en-US

## 生成 LocBaml 工具

1. 生成 LocBaml 所需的所有文件都位于 WPF 示例中。从 [LocBaml 工具示例](#) 下载 C# 文件。

2. 从命令行运行项目文件 (locbaml.csproj) 来生成该工具：

```
msbuild locbaml.csproj
```

3. 转到 Bin\Release 目录以查找新创建的可执行文件 (locbaml.exe)。示例：

C:\LocBaml\Bin\Release\locbaml.exe

4. 运行 LocBaml 时可指定下列选项。

选项	描述
<code>parse</code> 或 <code>-p</code>	分析 Baml、资源或 DLL 文件以生成 .csv 或 .txt 文件。
<code>generate</code> 或 <code>-g</code>	通过使用翻译后的文件生成本地化的二进制文件。
<code>out</code> 或 <code>-o</code> {filedirectory}	输出文件名。
<code>culture</code> 或 <code>-cul</code> {culture}	输出程序集的区域设置。
<code>translation</code> 或 <code>-trans</code> {translation.csv}	翻译后的文件或已本地化的文件。
<code>asmpath</code> 或 <code>-asmpath</code> {filedirectory}	如果你的 XAML 代码包含自定义控件，则必须为自定义控件程序集提供 <code>asmpath</code> 。
<code>nologo</code>	显示没有徽标或版权信息。
<code>verbose</code>	显示详细模式信息。

### ① 备注

如果在运行该工具时需要选项列表，则输入 `LocBaml.exe`，然后按 Enter。

## 使用 LocBaml 分析文件

由于已创建 LocBaml 工具，就可使用它来分析 HelloApp.resources.dll，从而提取将进行本地化的文本内容。

1. 将 LocBaml.exe 复制到应用程序的 bin\debug 文件夹，这也是创建主应用程序程序集的位置。

2. 若要分析附属程序集文件并将输出存储为 .csv 文件，请使用下列命令：

```
LocBaml.exe /parse HelloApp.resources.dll /out:Hello.csv
```

#### ① 备注

如果输入文件 HelloApp.resources.dll 不在 LocBaml.exe 所在的同一目录中，请移动其中一个文件以使两个文件都位于同一目录中。

3. 当运行 LocBaml 来分析文件时，输出包含由逗号（.csv 文件）或制表符（.txt 文件）分隔的七个字段。下面显示了 HelloApp.resources.dll 的已分析的 .csv 文件：

已分析的 .csv 文件
HelloApp.g.en-US.resources:window1.baml,Stack1:System.Windows.Controls.StackPanel.\$Content,Ignore, FALSE, FALSE,,#Text1;#Text2;
HelloApp.g.en-US.resources:window1.baml,Text1:System.Windows.Controls.TextBlock.\$Content,None,TRUE, TRUE,,Hello World
HelloApp.g.en-US.resources:window1.baml,Text2:System.Windows.Controls.TextBlock.\$Content,None,TRUE, TRUE,,Goodbye World

这七个字段是：

- BAML 名称。** 与源语言附属程序集相关的 BAML 资源的名称。
- 资源键。** 本地化的资源标识符。
- 类别。** 值类型。请参阅[本地化属性和注释](#)。
- Readability。** 值是否可以由本地化人员读取。请参阅[本地化属性和注释](#)。
- Modifiability。** 值是否可以由本地化人员修改。请参阅[本地化属性和注释](#)。
- 注释。** 值的附加说明，用于确定值被本地化的方式。请参阅[本地化属性和注释](#)。
- 值。** 要翻译为所需区域性设置的文本值。

下表显示了这些字段映射到 .csv 文件的分隔值的方式：

BAML 名称	资源键	类别	可读性	可修改性	注释	值
HelloApp.g.en-US.resources:window1.baml	Stack1:System.Windows.Controls.StackPanel.\$Content	忽略	FALSE	FALSE	#Text1;#Text2	
HelloApp.g.en-US.resources:window1.baml	Text1:System.Windows.Controls.TextBlock.\$Content	无	TRUE	TRUE	Hello World	
HelloApp.g.en-US.resources:window1.baml	Text2:System.Windows.Controls.TextBlock.\$Content	无	TRUE	TRUE	Goodbye World	

请注意，所有“注释”字段的值不包含任何值；如果字段没有值，则为空。此外，请注意第一行中的项既不可读也不可修改，并且拥有“Ignore”作为其“类别”值，这些都指示该值不可本地化。

4. 为了便于发现已分析文件中的可本地化项（特别是在大型文件中），可以通过“类别”、“可读性”和“可修改性”对这些项进行排序或筛选。例如，你可以筛选出不可读且不可修改的值。

# 翻译可本地化的内容

使用任何你可用的工具翻译提取的内容。执行此操作的一个好办法是将这些资源写入 .csv 文件，并在 Microsoft Excel 中查看它们，对最后一列（值）作出翻译更改。

## 使用 LocBaml 生成新的 .resources.dll 文件

通过使用 LocBaml 分析 HelloApp.resources.dll 而标识的内容已被翻译，且必须合并回原始应用程序。使用 `generate` 或 `-g` 选项生成一个新的 .resources.dll 文件。

1. 使用下列语法来生成新的 HelloApp.resources.dll 文件。将区域性标记为 zh-CN (/cul:zh-CN)。

```
LocBaml.exe /generate HelloApp.resources.dll /trans:Hello.csv /out:c:\ /cul:en-US
```

### ① 备注

如果输入文件 Hello.csv 与可执行文件 LocBaml.exe 不在同一目录中，请移动其中一个文件以使两个文件都位于同一目录中。

2. 使用新创建的 HelloApp.resources.dll 文件替换 C:\HelloApp\Bin\Debug\en-US\HelloApp.resources.dll 目录中的旧 HelloApp.resources.dll 文件。
3. 应在你的应用程序中将“Hello World”和“Goodbye World”翻译过来。
4. 若要翻译到不同的区域性设置，请使用目标语言的区域设置。下列示例演示了如何翻译为加拿大法语：

```
LocBaml.exe /generate HelloApp.resources.dll /trans:Hellofr-CA.csv /out:c:\ /cul:fr-CA
```

5. 在主应用程序程序集所在的程序集，创建一个新的特定于区域性的文件夹，以容纳新的附属程序集。对于加拿大法语，该文件夹将为 fr-CA。
6. 将生成的附属程序集复制到新建文件夹。
7. 若要测试新的附属程序集，你需要更改应用程序将在其下运行的区域性设置。可以通过两种方法执行此操作：
  - 更改操作系统的区域设置。
  - 在你的应用程序中，将下列代码添加到 App.xaml.cs 中：

XAML

```
<Application  
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
    x:Class="SDKSample.App"  
    x:Uid="Application_1"  
    StartupUri="Window1.xaml">  
</Application>
```

C#

```
using System.Windows;  
using System.Globalization;  
using System.Threading;
```

```
namespace SDKSample
{
    public partial class App : Application
    {
        public App()
        {
            // Change culture under which this application runs
            CultureInfo ci = new CultureInfo("fr-CA");
            Thread.CurrentThread.CurrentCulture = ci;
            Thread.CurrentThread.CurrentUICulture = ci;
        }
    }
}
```

## 使用 LocBaml 的一些提示

- 所有定义自定义控件的依赖程序集必须复制到 LocBaml 的本地目录，或安装到 GAC。这是必要的，因为本地化 API 在读取二进制 XAML (BAML) 时必须具有对依赖程序集的访问权限。
- 如果主程序集已签名，则生成的资源 DLL 也必须签名以进行加载。
- 本地化的资源 DLL 的版本需与主程序集进行同步。

## 另请参阅

- [WPF 的全球化](#)
- [使用自动布局概述](#)

# 如何：使用自动布局创建按钮

项目 • 2023/02/06

本示例介绍如何使用自动布局方法在可本地化的应用程序中创建按钮。

已根据所需调整对 UI 本地化进行了调整。现在，使用 Windows Presentation Foundation (WPF) 的功能，你可以对元素进行设计以减少必需的调整工作。这种编写可更轻松地调整大小和重新定位的应用程序的方法称为 `automatic layout`。

## 示例

以下两个 Extensible Application Markup Language (XAML) 示例创建了两个可实例化按钮的应用程序：一个使用英文文本，另一个使用西班牙文本。请注意，除文本之外，代码都一样；按钮会配合文字进行调整。

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Pane1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

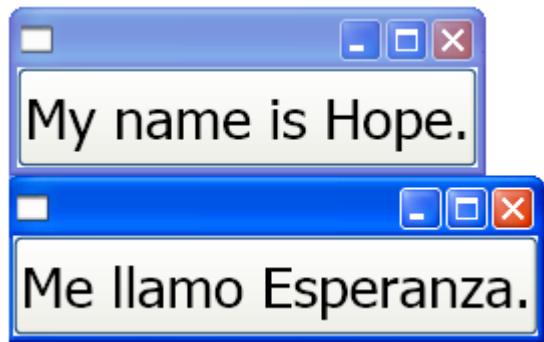
    <DockPanel>
        <Button FontSize="28" Height="50">My name is Hope.</Button>
    </DockPanel>
</Window>
```

XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ButtonLoc.Pane1"
    Name="myWindow"
    SizeToContent="WidthAndHeight"
    >

    <DockPanel>
        <Button FontSize="28" Height="50">Me llamo Esperanza.</Button>
    </DockPanel>
</Window>
```

下图显示可自动调整大小的按钮的代码示例的输出：



## 另请参阅

- [使用自动布局概述](#)
- [使用网格进行自动布局](#)

# 如何：使用网格进行自动布局

项目 • 2023/02/06

本示例介绍如何通过自动布局方法使用网格来创建可本地化的应用程序。

UI 的本地化已针对所需的调整进行了调整。现在，使用 Windows Presentation Foundation (WPF) 的功能，你可以对元素进行设计以减少必需的调整工作。这种编写可以更方便地调整大小和重新定位的应用程序的方法称为 `auto layout`。

以下 Extensible Application Markup Language (XAML) 示例演示如何使用网格来定位某些按钮和文本。请注意，单元格的高度和宽度设置为 `Auto`；因此，包含带图像按钮的单元格会调整为适应该图像。`Grid` 元素可根据内容进行调整，因此采用自动化布局方式设计可本地化的应用程序时，此元素非常有用。

## 示例

下面的示例演示如何使用网格。

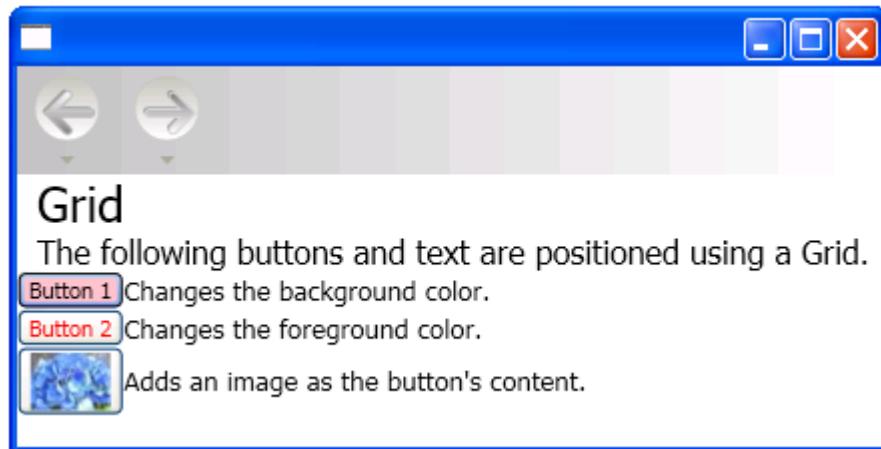
XAML

```
<Grid Name="grid" ShowGridLines ="false">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="Auto"/>
<ColumnDefinition Width="*"/>
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
<RowDefinition Height="Auto"/>
</Grid.RowDefinitions>

<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="0"
FontSize="24">Grid
</TextBlock>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="1" FontSize="12"
Grid.ColumnSpan="2">The following buttons and text are positioned using
a Grid.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="2" Background="Pink"
BorderBrush="Black" BorderThickness="10">Button 1
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="2" FontSize="12"
VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the
background
color.
```

```
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="3" Foreground="Red">
    Button 2
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="3" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Sets the
foreground
color.
</TextBlock>
<Button Margin="10, 10, 5, 5" Grid.Column="0" Grid.Row="4">
    <Image Source="data\flower.jpg"/></Image>
</Button>
<TextBlock Margin="10, 10, 5, 5" Grid.Column="1" Grid.Row="4" FontSize="12"
    VerticalAlignment="Center" TextWrapping="WrapWithOverflow">Adds an image
as
the button's content.
</TextBlock>
</Grid>
```

下图显示了代码示例的输出。



网格

## 另请参阅

- [使用自动布局概述](#)
- [使用自动布局创建按钮](#)

# 如何：使用 ResourceDictionary 来管理可本地化的字符串资源

项目 • 2023/02/06

本示例演示如何使用 [ResourceDictionary](#) 来包装 Windows Presentation Foundation (WPF) 应用程序的可本地化字符串资源。

## 使用 ResourceDictionary 来管理可本地化的字符串资源

1. 创建一个 [ResourceDictionary](#)，其中包含要本地化的字符串。以下代码展示了一个示例。

XAML

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:system="clr-namespace:System;assembly=mscorlib">

    <!-- String resource that can be localized -->
    <system:String x:Key="localizedMessage">en-US Message</system:String>

</ResourceDictionary>
```

此代码从 mscorelib.dll 中的 [System](#) 命名空间定义了一个类型为 [String](#) 的字符串资源 `localizedMessage`。

2. 使用以下代码将 [ResourceDictionary](#) 添加到应用程序中。

XAML

```
<Application.Resources>
    <ResourceDictionary>
        <ResourceDictionary.MergedDictionaries>
            <ResourceDictionary Source="StringResources.xaml" />
        </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

3. 通过如下所示的 Extensible Application Markup Language (XAML) 来使用标记中的字符串资源。

XAML

```
<!-- Declarative use of string resource from StringResources.xaml  
resource dictionary -->  
<TextBox DockPanel.Dock="Top" Text="{StaticResource localizedMessage}"  
/>
```

4. 通过如下所示的代码来使用代码隐藏文件中的字符串资源。

C#

```
// Programmatic use of string resource from StringResources.xaml  
resource dictionary  
string localizedMessage =  
(string)Application.Current.FindResource("localizedMessage");  
MessageBox.Show(localizedMessage);
```

5. 对应用程序进行本地化。有关详细信息，请参阅[对应用程序进行本地化](#)。

# 如何：使用可本地化的应用中的资源

项目 • 2022/10/19

本地化是指调整用户界面以适应不同的区域性。为此，必须翻译诸如标题、描述和列表框项等文本。为简化翻译过程，要翻译的项需收集到资源文件中。请参阅[对应用程序进行本地化](#)，了解如何创建要进行本地化的资源文件。若要使 WPF 应用程序可本地化，开发人员需要将所有可本地化的资源生成一个资源程序集。该资源程序集本地化为不同语言，代码隐藏功能使用资源管理 API 进行加载。

## 示例

WPF 应用程序所需的文件之一是项目文件 (.proj)。应用程序中使用的所有资源都应包括在项目文件中。以下 XAML 示例说明了这一点。

XAML

```
<Resource Include="data\picture1.jpg"/>
<EmbeddedResource Include="data\stringtable.en-US.restext"/>
```

若要在应用程序中使用资源，请实例化 [ResourceManager](#) 并加载要使用的资源。下面的 C# 代码演示如何执行此操作。

C#

```
void OnClick(object sender, RoutedEventArgs e)
{
    ResourceManager rm = new ResourceManager ("MySampleApp.data.stringtable",
        Assembly.GetExecutingAssembly());
    Text1.Text = rm.GetString("Message");
}
```

## 另请参阅

- [对应用进行本地化](#)

# Layout

项目 · 2022/09/27

本主题介绍 Windows Presentation Foundation (WPF) 布局系统。了解布局计算发生的方式和时间对于在 WPF 中创建用户界面非常重要。

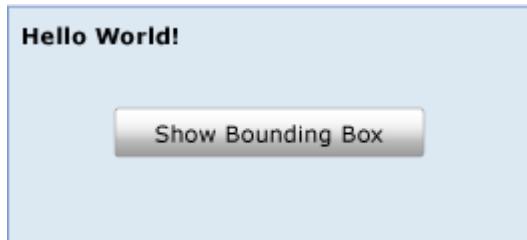
本主题包含以下各节：

- [元素边界框](#)
- [布局系统](#)
- [测量和排列子元素](#)
- [面板元素和自定义布局行为](#)
- [布局性能注意事项](#)
- [子像素渲染和布局舍入](#)
- [后续步骤](#)

## 元素边界框

在 WPF 中构思布局时，了解环绕所有元素的边界框非常重要。布局系统使用的每个 [FrameworkElement](#) 都可以被视为嵌入到布局中的矩形。[LayoutInformation](#) 类返回元素的布局分配或布局槽的边界。矩形的大小是通过计算可用屏幕空间、任何约束的大小、特定于布局的属性（如边距和填充）以及父 [Panel](#) 元素的个别行为来确定的。处理此数据时，布局系统能够计算特定 [Panel](#) 的所有子级的位置。请务必记住，调整在父元素（如 [Border](#)）上定义的特性的大小会影响其子级。

下图显示了一个简单的布局。



可以通过使用以下 XAML 实现此布局。

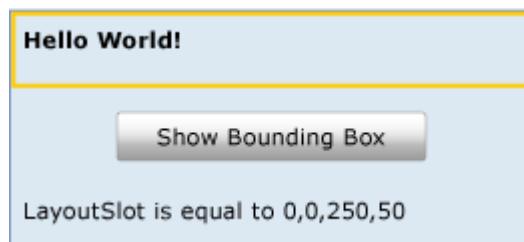
XAML

```

<Grid Name="myGrid" Background="LightSteelBlue" Height="150">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="250"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition />
        <RowDefinition />
        <RowDefinition />
    </Grid.RowDefinitions>
    <TextBlock Name="txt1" Margin="5" FontSize="16" FontFamily="Verdana"
        Grid.Column="0" Grid.Row="0">Hello World!</TextBlock>
    <Button Click="getLayoutSlot1" Width="125" Height="25" Grid.Column="0"
        Grid.Row="1">Show Bounding Box</Button>
    <TextBlock Name="txt2" Grid.Column="1" Grid.Row="2"/>
</Grid>

```

单个 [TextBlock](#) 元素托管在 [Grid](#) 中。 虽然文本仅填充第一列的左上角，但 [TextBlock](#) 的分配空间实际上要大得多。 可以使用 [GetLayoutSlot](#) 方法检索任何 [FrameworkElement](#) 的边界框。 下图显示 [TextBlock](#) 元素的边界框。



如黄色矩形所示，[TextBlock](#) 元素的分配空间实际上远大于所显示的空间。 由于还有其他元素添加到 [Grid](#) 中，这种分配可能会收缩或扩展，具体取决于所添加元素的类型和大小。

使用 [GetLayoutSlot](#) 方法将 [TextBlock](#) 的布局槽转换为 [Path](#)。 此方法可用于显示元素的边界框。

C#

```

private void getLayoutSlot1(object sender, System.Windows.RoutedEventArgs e)
{
    RectangleGeometry myRectangleGeometry = new RectangleGeometry();
    myRectangleGeometry.Rect = LayoutInformation.GetLayoutSlot(txt1);
    Path myPath = new Path();
    myPath.Data = myRectangleGeometry;
    myPath.Stroke = Brushes.LightGoldenrodYellow;
    myPath.StrokeThickness = 5;
    Grid.SetColumn(myPath, 0);
    Grid.SetRow(myPath, 0);
    myGrid.Children.Add(myPath);
    txt2.Text = "LayoutSlot is equal to " +
    LayoutInformation.GetLayoutSlot(txt1).ToString();
}

```

# 布局系统

简单地说，布局是一个递归系统，实现对元素进行大小调整、定位和绘制。更具体地说，布局描述测量和排列 `Panel` 元素的 `Children` 集合的成员的过程。布局是一个密集的过程。`Children` 集合越大，必须进行的计算次数就越多。根据拥有该集合的 `Panel` 元素所定义的布局行为，还可能会增加复杂性。相对简单的 `Panel`（如 `Canvas`）可以比更复杂的 `Panel`（如 `Grid`）具有更好的性能。

每当子 `UIElement` 改变其位置时，布局系统都可能触发一个新的传递。因此，了解哪些事件会调用布局系统就很重要，因为不必要的调用可能导致应用程序性能变差。下面描述调用布局系统时发生的过程。

1. 子 `UIElement` 通过首先测量其核心属性来开始布局过程。
2. 计算 `FrameworkElement` 上定义的大小调整属性，例如 `Width`、`Height` 和 `Margin`。
3. 应用 `Panel` 特定的逻辑，例如 `Dock` 方向或堆叠 `Orientation`。
4. 测量所有子级后排列内容。
5. 在屏幕上绘制 `Children` 集合。
6. 如果向集合添加了其他 `Children`，应用了 `LayoutTransform`，或者调用了 `UpdateLayout` 方法，则会再次调用该过程。

以下各节更详细地定义了此过程及其调用方式。

## 测量和排列子元素

布局系统为 `Children` 集合的每个成员完成两个过程：一个测量过程和一个排列过程。每个子 `Panel` 都提供自己的 `MeasureOverride` 和 `ArrangeOverride` 方法来实现自己的特定布局行为。

在测量过程中，会计算 `Children` 集合的每个成员。此过程从调用 `Measure` 方法开始。需要在父 `Panel` 元素的实现中调用此方法，而不必为要出现的布局显式调用该方法。

首先，计算 `UIElement` 的本机大小属性，例如 `Clip` 和 `Visibility`。这将生成一个名为 `constraintSize` 的值，该值将传递给 `MeasureCore`。

其次，处理在 `FrameworkElement` 上定义的框架属性，这会影响 `constraintSize` 的值。这些属性通常描述基础 `UIElement` 的大小调整特性，例如其 `Height`、`Width`、`Margin` 和

[Style](#)。其中每个属性都可以更改显示元素所需的空间。然后使用 `constraintSize` 作为参数调用 [MeasureOverride](#)。

### ① 备注

`Height` 和 `Width` 属性与 `ActualHeight` 和 `ActualWidth` 属性有所不同。例如，`ActualHeight` 属性是基于其他高度输入和布局系统的计算值。该值是由布局系统本身基于实际呈现的传递设置的，因此可能稍微小于属性（例如作为输入更改基础的 `Height`）的设置值。

因为 `ActualHeight` 是计算所得的值，所以你应该知道，由于布局系统各种操作的结果，该值可能有多个或递增的报告的更改。布局系统可能会计算子元素所需的测量空间、父元素的约束等。

测量过程的最终目标是让子元素确定其 `DesiredSize`，这发生在 [MeasureCore](#) 调用期间。[Measure](#) 存储 `DesiredSize` 值，供在内容排列过程中使用。

排列过程从调用 [Arrange](#) 方法开始。在排列过程中，父 [Panel](#) 元素会生成一个表示子元素边界的矩形。该值将传递给 [ArrangeCore](#) 方法进行处理。

[ArrangeCore](#) 方法计算子元素的 `DesiredSize`，并且计算可能影响元素呈现大小的任何其他边距。[ArrangeCore](#) 生成一个 `arrangeSize`，后者作为参数传递给 [Panel](#) 的 [ArrangeOverride](#) 方法。[ArrangeOverride](#) 生成子元素的 `finalSize`。最后，[ArrangeCore](#) 方法执行偏移量属性（如边距和对齐）的最终计算，并将子元素放在其布局槽内。子元素不需要（并且通常不）填充整个分配空间。然后将控件返回给父级 [Panel](#)，布局过程即告完成。

## 面板元素和自定义布局行为

WPF 包含一组派生自 [Panel](#) 的元素。这些 [Panel](#) 元素支持许多复杂的布局。例如，使用 [StackPanel](#) 元素可以轻松实现堆叠元素，而使用 [Canvas](#) 可实现更复杂和自由流动的布局。

下表汇总了可用的布局 [Panel](#) 元素。

面板名称	说明
<a href="#">Canvas</a>	定义一个区域，可在其中通过相对于 <a href="#">Canvas</a> 区域的坐标显式定位子元素。
<a href="#">DockPanel</a>	定义一个区域，从中可以按相对位置水平或垂直排列各个子元素。
<a href="#">Grid</a>	定义由列和行组成的灵活的网格区域。

面板名称	说明
StackPanel	将子元素排列成水平或垂直的一行。
VirtualizingPanel	为虚拟化其子数据集合的 <a href="#">Panel</a> 元素提供一个框架。 这是一个抽象类。
WrapPanel	按从左到右的顺序位置定位子元素，在包含框的边缘处将内容切换到下一行。 排序顺序是从上到下还是从右到左，取决于 <a href="#">Orientation</a> 属性的值。

对于需要使用任何预定义的 [Panel](#) 元素都无法实现的布局的应用程序，可以通过继承 [Panel](#) 并替代 [MeasureOverride](#) 和 [ArrangeOverride](#) 方法来实现自定义布局行为。

## 布局性能注意事项

布局是一个递归过程。[Children](#) 集合中的每个子元素都会在每次调用布局系统期间得到处理。因此，应避免在不必要时触发布局系统。以下注意事项有助于实现更好的性能。

- 应注意哪些属性值更改会强制执行布局系统的递归更新。

如果依赖属性的值可能导致布局系统被初始化，则会使用公共标志对该依赖属性进行标记。[AffectsMeasure](#) 和 [AffectsArrange](#) 提供了有用的线索，说明哪些属性值更改会强制执行布局系统的递归更新。一般来说，任何可能影响元素边界框大小的属性都应将 [AffectsMeasure](#) 标志设置为 `True`。有关详细信息，请参阅[依赖项属性概述](#)。

- 如果可能，请使用 [RenderTransform](#) 而不是 [LayoutTransform](#)。

[LayoutTransform](#) 是一种影响用户界面 (UI) 内容的非常有用的方式。但是，如果转换效果不需要影响其他元素的位置，则最好改用 [RenderTransform](#)，因为 [RenderTransform](#) 不会调用布局系统。[LayoutTransform](#) 会应用其转换，并强制执行递归布局更新以获得受影响元素的新位置。

- 避免对 [UpdateLayout](#) 进行不必要的调用。

[UpdateLayout](#) 方法会强制执行递归布局更新，通常是不必要的。除非你确定需要进行完整更新，否则请依赖布局系统为你调用此方法。

- 在处理大型 [Children](#) 集合时，请考虑使用 [VirtualizingStackPanel](#) 而不是常规的 [StackPanel](#)。

通过虚拟化子集合，[VirtualizingStackPanel](#) 仅在内存中保留当前位于父级视区内的对象。因此，在大多数情况下，性能得到显著提高。

## 子像素渲染和布局舍入

WPF 图形系统使用与设备无关的单元来使分辨率和设备独立。 每个与设备无关的像素都会随着系统的每英寸点数 (dpi) 设置自动进行缩放。 这为 WPF 应用程序提供了不同 dpi 设置的适当缩放，并使应用程序自动感知 dpi。

但是，这种 dpi 无关性可能由于抗锯齿而呈现出不规则的边缘。 这些伪影通常被视为模糊或半透明边缘，当边缘的位置落在设备像素的中间而不是设备像素之间时，就可能出现。 布局系统提供了一种通过布局倒圆对此进行调整的方法。 布局舍入是布局系统在布局传递中舍入任何非整数像素值的情况。

默认情况下禁用布局舍入。 若要启用布局舍入，请在任何 [FrameworkElement](#) 上将 [UseLayoutRounding](#) 属性设置为 `true`。 因为它是一个依赖属性，所以该值将传播到可视化树中的所有子级。 若要为整个 UI 启用布局舍入，请在根容器上将 [UseLayoutRounding](#) 设置为 `true`。 有关示例，请参见 [UseLayoutRounding](#)。

## 后续步骤

了解元素的测量和排列方式是了解布局的第一步。 有关可用 [Panel](#) 元素的详细信息，请参阅[面板概述](#)。 若要更好地了解可能影响布局的各种定位属性，请参阅[对齐、边距和填充概述](#)。 当你准备好将其全部放在一个轻量级应用程序中时，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。

## 另请参阅

- [FrameworkElement](#)
- [UIElement](#)
- [面板概述](#)
- [Alignment、Margin 和 Padding 概述](#)
- [布局和示例](#)

# 迁移和互操作性

项目 • 2022/10/19

此页面包含文档链接，这些文档讨论如何在 Windows Presentation Foundation (WPF) 应用程序和其他类型的 Microsoft Windows 应用程序之间实现互操作。

## 本节内容

[从 WPF 迁移到 System.Xaml 的类型](#)

[WPF 和 Windows 窗体互操作](#)

[WPF 和 Win32 互操作](#)

[WPF 和 Direct3D9 互操作](#)

## 参考

术语	定义
<a href="#">WindowsFormsHost</a>	可用于将 Windows 窗体控件作为 WPF 页面元素来托管的元素。
<a href="#">ElementHost</a>	可用于托管 Windows Presentation Foundation (WPF) 控件的 Windows 窗体控件。
<a href="#">HwndSource</a>	在 Win32 应用程序中托管 WPF 区域。
<a href="#">HwndHost</a>	<a href="#">WindowsFormsHost</a> 的基类，定义所有基于 HWND 的技术在由 WPF 应用程序托管时使用的一些基本功能。 创建该类的子类以在 WPF 应用程序中托管 Win32 窗口。
<a href="#">BrowserInteropHelper</a>	用于报告由浏览器托管的 WPF 应用程序的浏览器环境状况的帮助程序类。

## 相关章节

# 从 WPF 迁移到 System.Xaml 的类型

项目 • 2022/09/27

在 .NET Framework 3.5 和 .NET Framework 3.0 中，Windows Presentation Foundation (WPF) 和 Windows Workflow Foundation 都包含 XAML 语言实现。为 WPF XAML 实现提供扩展性的很多公共类型都存在于 WindowsBase、PresentationCore 和 PresentationFramework 程序集中。同样，为 Windows Workflow Foundation XAML 提供扩展性的公共类型存在于 System.Workflow.ComponentModel 程序集中。在 .NET Framework 4 中，一些与 XAML 相关的类型已迁移到 System.Xaml 程序集。XAML 语言服务的一个常见 .NET Framework 实现支持最初由特定框架的 XAML 实现定义的很多 XAML 扩展性方案，但是现在这些方案都属于整体 .NET Framework 4 XAML 语言支持的一部分。本文列出了迁移的类型并讨论了与迁移有关的问题。

## 程序集和命名空间

在 .NET Framework 3.5 和 .NET Framework 3.0 中，WPF 为支持 XAML 而实现的类型通常位于 [System.Windows.Markup](#) 命名空间中。大多数这些类型都位于 WindowsBase 程序集中。

在 .NET Framework 4 中，新增了一个 [System.Xaml](#) 命名空间和 System.Xaml 程序集。最初为 WPF XAML 实现的很多类型现在作为任何 XAML 实现的扩展性点或服务提供。为了使其可用于更普通的方案，将类型从其最初所在的 WPF 程序集类型转发到 System.Xaml 程序集。这可启用 XAML 扩展性方案而无需包含其他框架（如 WPF 和 Windows Presentation Foundation）的程序集。

对于已迁移的类型，大多数类型仍位于 [System.Windows.Markup](#) 命名空间中。这是做部分是为了避免中断每个文件上现有实现中的 CLR 命名空间映射。因此，.NET Framework 4 中的 [System.Windows.Markup](#) 命名空间包含混合的常规 XAML 语言支持类型（来自 System.Xaml 程序集）和特定于 WPF XAML 实现的类型（来自 WindowsBase 和其他 WPF 程序集）。在版本 4 的 WPF 程序集中，已迁移到 System.Xaml，但以前存在于 WPF 程序集中的任何类型都具有类型转发支持。

## 工作流 XAML 支持类型

Windows Workflow Foundation 还提供 XAML 支持类型，并且在很多情况下它们具有与 WPF 等效项相同的短名称。以下是 Windows Workflow Foundation XAML 支持类型的列表：

- [ContentPropertyAttribute](#)
- [RuntimeNamePropertyAttribute](#)

- [XmlAttributePrefixAttribute](#)

这些支持类型仍然存在于 .NET Framework 4 的 Windows Workflow Foundation 程序集中，并且仍可用于特定的 Windows Workflow Foundation 应用程序；但是，它们不应由不使用 Windows Workflow Foundation 的应用程序或框架引用。

## MarkupExtension

在 .NET Framework 3.5 和 .NET Framework 3.0 中，WPF 的 [MarkupExtension](#) 类位于 WindowsBase 程序集中。Windows Workflow Foundation 的 Parallel 类 [MarkupExtension](#) 位于 System.Workflow.ComponentModel 程序集中。在 .NET Framework 4 中，[MarkupExtension](#) 类被迁移到 System.Xaml 程序集中。在 .NET Framework 4 中，[MarkupExtension](#) 适用于使用 .NET XAML 服务的任何 XAML 扩展性方案，而不仅仅用于在特定框架上生成的方案。在可能的情况下，特定的框架或架构中的用户代码也应在 XAML 扩展的 [MarkupExtension](#) 类上生成。

## MarkupExtension 支持服务类

WPF 的 .NET Framework 3.5 和 .NET Framework 3.0 提供了多种服务，这些服务可用于 [MarkupExtension](#) 实施者和 [TypeConverter](#) 实现以支持 XAML 中的类型/属性用法。这些服务如下所示：

- [IProvideValueTarget](#)
- [IUriContext](#)
- [IXamlTypeResolver](#)

### ① 备注

来自 .NET Framework 3.5 且与标记扩展有关的另一个服务是 [IReceiveMarkupExtension](#) 接口。[IReceiveMarkupExtension](#) 未迁移并且对于 .NET Framework 4 标记为 [Obsolete]。以前使用 [IReceiveMarkupExtension](#) 的方案应改用 [XamlSetMarkupExtensionAttribute](#) 特性化回调。[AcceptedMarkupExtensionExpressionTypeAttribute](#) 也标记为 [Obsolete]。

## XAML 语言功能

WPF 的多个 XAML 语言功能和组件以前存在于 PresentationFramework 程序集中。这些功能和组件作为 [MarkupExtension](#) 子类实现以在 XAML 标记中公开标记扩展用法。在

.NET Framework 4 中，它们存在于 System.Xaml 程序集中，以便不包含 WPF 程序集的项目可以使用这些 XAML 语言级别的功能。WPF 对 .NET Framework 4 应用程序使用这些相同的实现。与本主题中列出的其他情况一样，这些支持类型继续存在于 System.Windows.Markup 命名空间中，以避免中断以前的引用。

下表包含在 System.Xaml 中定义的 XAML 功能支持类的列表。

XAML 语言功能	使用情况
ArrayExtension	<x:Array ...>
NullExtension	{x:Null}
StaticExtension	{x:Static ...}
TypeExtension	{x:Type ...}

虽然 System.Xaml 可能没有特定支持类，但用于处理 XAML 语言的语言功能的常规逻辑现在位于 System.Xaml 及其实现的 XAML 读取器和 XAML 编写器中。例如，`x:TypeArguments` 是一个由 System.Xaml 实现中的 XAML 读取器和 XAML 编写器处理的特性；该特性可在 XAML 节点流中说明，可在默认的（基于 CLR 的）XAML 架构上下文中进行处理，具有 XAML 类型系统表示形式，等等。有关 XAML 参考文档的详细信息，请参阅 [XAML 服务](#)。

## ValueSerializer 和支持类

[ValueSerializer](#) 类支持到字符串的类型转换，尤其对于序列化可能需要输出中有多个模式或节点的 XAML 序列化情况。在 .NET Framework 3.5 和 .NET Framework 3.0 中，WPF 的 [ValueSerializer](#) 位于 WindowsBase 程序集中。在 .NET Framework 4 中，[ValueSerializer](#) 类位于 System.Xaml 中，并且适用于任何 XAML 扩展性方案，而不仅仅用于在 WPF 上生成的方案。[IValueSerializerContext](#)（支持服务）和 [DateTimeValueSerializer](#)（特定子类）也会被迁移到 System.Xaml。

## 与 XAML 相关的属性

WPF XAML 包含多个特性，这些特性可应用于 CLR 类型以指示有关它们的 XAML 行为的某些内容。以下是 .NET Framework 3.5 和 .NET Framework 3.0 的 WPF 程序集中存在的特性列表。在 .NET Framework 4 中，这些特性迁移到 System.Xaml 中。

- [AmbientAttribute](#)
- [ContentPropertyAttribute](#)

- ContentWrapperAttribute
- DependsOnAttribute
- MarkupExtensionReturnTypeAttribute
- NameScopePropertyAttribute
- RootNamespaceAttribute
- RuntimeNamePropertyAttribute
- TrimSurroundingWhitespaceAttribute
- ValueSerializerAttribute
- WhitespaceSignificantCollectionAttribute
- XmlLangPropertyAttribute
- XmlNsCompatibleWithAttribute
- XmlNsDefinitionAttribute
- XmlNsPrefixAttribute

## 其他类

在 .NET Framework 3.5 和 .NET Framework 3.0 中，[IComponentConnector](#) 接口存在于 WindowsBase 中，但在 .NET Framework 4 中，该接口存在于 System.Xaml 中。  
[IComponentConnector](#) 主要适用于工具支持和 XAML 标记编译器。

在 .NET Framework 3.5 和 .NET Framework 3.0 中，[INamespace](#) 接口存在于 WindowsBase 中，但在 .NET Framework 4 中，该接口存在于 System.Xaml 中。  
[INamespace](#) 定义用于 XAML 名称范围的基本操作。

## 存在于 WPF 和 System.Xaml 中具有共享名称的与 XAML 相关的类

在 .NET Framework 4 中，以下类既存在于 WPF 程序集中，也存在于 System.Xaml 程序集中：

XamlReader

XamlWriter

## XamlParseException

WPF 实现位于 [System.Windows.Markup](#) 命名空间和 PresentationFramework 程序集中。 System.Xaml 实现位于 [System.Xaml](#) 命名空间中。 如果你使用的是 WPF 类型或是从 WPF 类型派生的，则通常应使用 [XamlReader](#) 和 [XamlWriter](#) 的 WPF 实现，而不是 System.Xaml 实现。 有关详细信息，请参阅 [System.Windows.Markup.XamlReader](#) 和 [System.Windows.Markup.XamlWriter](#) 中的“备注”。

如果要包括对 WPF 程序集和 System.Xaml 的应用，并且还要对 `include` 和 [System.Windows.Markup](#) 命名空间使用 [System.Xaml](#) 语句，则可能需要完全限定对这些 API 的调用，以便清楚无误地解析类型。

# WPF 和 Windows 窗体互操作

项目 • 2023/02/06

WPF 和 Windows 窗体提供两种用于创建应用程序接口的不同体系结构。

`System.Windows.Forms.Integration` 命名空间提供实现常见互操作方案的类。 实现互操作功能的两个关键类是 `WindowsFormsHost` 和 `ElementHost`。 本主题介绍支持哪些互操作方案以及不支持哪些互操作方案。

## ① 备注

关于混合控件方案，需要考虑一些特殊因素。 混合控件将一种技术中的控件嵌套于另一种技术中的控件。 这也称为嵌套互操作。 多级混合控件具有多个级别的混合控件嵌套。 多级嵌套互操作的一个示例是包含 WPF 控件的 Windows 窗体控件，前者又包含另一个 Windows 窗体控件。 不支持多级混合控件。

## 在 WPF 中承载 Windows 窗体控件

当 WPF 控件承载一个 Windows 窗体控件时，支持下列互操作方案：

- WPF 控件可以使用 XAML 承载一个或多个 Windows 窗体控件。
- 它可以使用代码承载一个或多个 Windows 窗体控件。
- 它可以承载包含其他 Windows 窗体控件的 Windows 窗体容器控件。
- 它可以承载具有 WPF 母版和 Windows 窗体详细信息的母版/详细信息窗体。
- 它可以承载具有 Windows 窗体母版和 WPF 详细信息的母版/详细信息窗体。
- 它可以承载一个或多个 ActiveX 控件。
- 它可以承载一个或多个复合控件。
- 它可以使用 Extensible Application Markup Language (XAML) 承载混合控件。
- 它可以使用代码承载混合控件。

## 布局支持

下面列出了当 `WindowsFormsHost` 元素尝试将其承载的 Windows 窗体控件集成到 WPF 布局系统时的已知限制。

- 某些情况下，不能调整 Windows 窗体控件的大小，或者大小只能调整为特定尺寸。例如，Windows 窗体 [ComboBox](#) 控件仅支持由控件的字号定义的单一高度。在 WPF 动态布局中，假定元素可以垂直拉伸，则承载的 [ComboBox](#) 控件不会按预期拉伸。
- 不能旋转或倾斜 Windows 窗体控件。例如，将用户界面旋转 90 度时，所承载的 Windows 窗体控件将保持其垂直位置。
- 大多数情况下，Windows 窗体控件不支持按比例缩放。尽管该控件的整体尺寸将会缩放，但其子控件和组件元素可能不会按预期调整大小。此限制取决于每个 Windows 窗体控件支持缩放的程度。
- 在 WPF 用户界面中，可以更改元素的 z 顺序以控制重叠行为。由于承载的 Windows 窗体控件是在单独的 HWND 中绘制的，所以始终在 WPF 元素之上绘制它。
- Windows 窗体控件支持基于字号的自动缩放。在 WPF 用户界面中，更改字号不会改变整个布局的大小，但是可动态调整单个元素的大小。

## 环境属性

WPF 控件的某些环境属性具有 Windows 窗体等效项。这些环境属性传播到所承载的 Windows 窗体控件，并作为 [WindowsFormsHost](#) 控件上的公共属性公开。[WindowsFormsHost](#) 控件将每个 WPF 环境属性转换成它的 Windows 窗体等效项。

有关详细信息，请参阅 [Windows 窗体和 WPF 属性映射](#)。

## 行为

下表介绍互操作行为。

行为	支持	不支持
透明度	Windows 窗体控件呈现支持透明度。父 WPF 控件的背景可以成为所承载的 Windows 窗体控件的背景。	某些 Windows 窗体控件不支持透明度。例如， <a href="#">TextBox</a> 和 <a href="#">ComboBox</a> 控件在由 WPF 承载时不会是透明的。

行为	支持	不支持
Tab 键次序	<p>所承载的 Windows 窗体控件的 Tab 键顺序与这些控件承载于基于 Windows 窗体的应用程序中时是相同的。</p> <p>使用 Tab 键和 Shift+Tab 键从 WPF 控件切换到 Windows 窗体控件将按常规方式工作。</p> <p>当用户按 Tab 键在多个控件之间切换时，<a href="#">TabStop</a> 属性值为 <code>false</code> 的 Windows 窗体控件不会获得焦点。</p> <ul style="list-style-type: none"> <li>- 每个 <a href="#">WindowsFormsHost</a> 控件都有一个 <a href="#">TabIndex</a> 值，该值确定该 <a href="#">WindowsFormsHost</a> 控件何时获得焦点。</li> <li>- 包含在 <a href="#">WindowsFormsHost</a> 容器内的 Windows 窗体控件遵循 <a href="#">TabIndex</a> 属性指定的顺序。从最后一个 Tab 键索引使用 Tab 键切换会将焦点置于下一个 WPF 控件上（若存在）。如果不存在其他可聚焦的 WPF 控件，则按 Tab 键将返回到 Tab 键顺序中的第一个 Windows 窗体控件。</li> <li>- <a href="#">WindowsFormsHost</a> 内部控件的 <a href="#">TabIndex</a> 值与 <a href="#">WindowsFormsHost</a> 控件中包含的同级 Windows 窗体控件相关。</li> <li>- 按 Tab 键遵循特定于控件的行为。例如，在 <a href="#">AcceptsTab</a> 属性值为 <code>true</code> 的 <a href="#">TextBox</a> 控件中按 TAB 键会在文本框中输入一个制表符，而不会移动焦点。</li> </ul>	不适用。

行为	支持	不支持
使用箭头键导航	<p>- 在 <a href="#">WindowsFormsHost</a> 控件中使用箭头键进行导航与在普通的 Windows 窗体容器控件中相同：向上键和向左键选择上一个控件，向下键和向右键选择下一个控件。</p> <p>- 从 <a href="#">WindowsFormsHost</a> 控件中包含的第一个控件按向上键和向左键与 Shift+Tab 键盘快捷方式所执行的操作相同。如果存在可聚焦的 WPF 控件，则焦点将移到 <a href="#">WindowsFormsHost</a> 控件的外部。此行为与标准 <a href="#">ContainerControl</a> 行为的不同之处是不会包装最后一个控件。如果不存在其他可聚焦的 WPF 控件，则焦点将返回到 Tab 键顺序中的最后一个 Windows 窗体控件。</p> <p>- 从 <a href="#">WindowsFormsHost</a> 控件中包含的最后一个控件按向下键和向右键与 Tab 键所执行的操作相同。如果存在可聚焦的 WPF 控件，则焦点将移到 <a href="#">WindowsFormsHost</a> 控件的外部。此行为与标准 <a href="#">ContainerControl</a> 行为的不同之处是不会包装第一个控件。如果不存在其他可聚焦的 WPF 控件，则焦点将返回到 Tab 键顺序中的第一个 Windows 窗体控件。</p>	不适用。
加速键	加速键按常规方式工作，但“不支持”列中注明的情况除外。	跨多种技术的重复加速键与普通重复加速键的工作方式不同。如果加速键跨多种技术复制，并且至少一个在 Windows 窗体控件上，另一个在 WPF 控件上，Windows 窗体控件将始终收到此加速键。当按重复加速键时，焦点不会在控件之间切换。

行为	支持	不支持
快捷键	快捷键按常规方式工作，但“不支持”列中注明的情况除外。	<ul style="list-style-type: none"> <li>- 在预处理阶段处理的 Windows 窗体快捷键总是优先于 WPF 快捷键。例如，如果有一个定义了 Ctrl+S 快捷键的 <a href="#">ToolStrip</a> 控件，并且存在绑定到 Ctrl+S 的 WPF 命令，则无论焦点在何处，总是首先调用 <a href="#">ToolStrip</a> 控件处理程序。</li> <li>- 由 <a href="#">KeyDown</a> 事件处理的 Windows 窗体快捷键在 WPF 中最后处理。可通过替代 Windows 窗体控件的 <a href="#">IsInputKey</a> 方法或处理 <a href="#">PreviewKeyDown</a> 事件来防止此行为。从 <a href="#">IsInputKey</a> 方法返回 <code>true</code>，或在 <a href="#">PreviewKeyDown</a> 事件处理程序中将 <a href="#">PreviewKeyDownEventArgs.IsInputKey</a> 属性的值设置为 <code>true</code>。</li> </ul>
AcceptsReturn、AcceptsTab 以及其他特定于控件的行为	更改默认键盘行为的属性按常规方式工作，前提是 Windows 窗体控件替代 <a href="#">IsInputKey</a> 方法以返回 <code>true</code> 。	通过处理 <a href="#">KeyDown</a> 事件来更改默认键盘行为的 Windows 窗体控件在 WPF 主机控件中最后处理。因为最后处理这些控件，所以它们可能产生意外行为。
Enter 和 Leave 事件	如果焦点未转到包含的 <a href="#">ElementHost</a> 控件，则在一个 <a href="#">WindowsFormsHost</a> 控件中更改焦点时，将会像通常那样引发 Enter 和 Leave 事件。	<p>发生以下焦点更改时，不会引发 Enter 和 Leave 事件：</p> <ul style="list-style-type: none"> <li>- 从 <a href="#">WindowsFormsHost</a> 控件的内部到外部。</li> <li>- 从 <a href="#">WindowsFormsHost</a> 控件的外部到内部。</li> <li>- 在 <a href="#">WindowsFormsHost</a> 控件外部。</li> <li>- 从 <a href="#">WindowsFormsHost</a> 控件中承载的 Windows 窗体控件到同一 <a href="#">WindowsFormsHost</a> 中承载的 <a href="#">ElementHost</a> 控件。</li> </ul>
多线程处理	支持所有类型的多线程处理。	Windows 窗体和 WPF 技术均采用单线程并发模型。调试期间，从其他线程调用框架对象会引发一个异常，以强制实施此要求。
安全性	所有互操作方案都需要完全信任。	部分信任情况下，不允许任何互操作方案。

行为	支持	不支持
可访问性	支持所有辅助功能方案。当辅助技术产品用于同时包含 Windows 窗体和 WPF 控件的混合应用程序时，可正常工作。	不适用。
剪贴板	所有剪贴板操作按常规方式工作。这包括 Windows 窗体与 WPF 控件之间的剪切和粘贴。	不适用。
拖放功能	所有拖放操作按常规方式工作。这包括 Windows 窗体和 WPF 控件之间的操作。	不适用。

## 在 Windows 窗体中承载 WPF 控件

当 Windows 窗体控件承载一个 WPF 控件时，支持下列互操作方案：

- 使用代码承载一个或多个 WPF 控件。
- 将属性表与一个或多个承载的 WPF 控件关联。
- 在窗体中承载一个或多个 WPF 页。
- 启动 WPF 窗口。
- 承载具有 Windows 窗体母版和 WPF 详细信息的母版/详细信息窗体。
- 承载具有 WPF 母版和 Windows 窗体详细信息的母版/详细信息窗体。
- 承载自定义 WPF 控件。
- 承载混合控件。

## 环境属性

Windows 窗体控件的某些环境属性具有 WPF 等效项。这些环境属性传播到所承载的 WPF 控件，并作为 [ElementHost](#) 控件上的公共属性公开。[ElementHost](#) 控件将每个 Windows 窗体环境属性转换成它的 WPF 等效项。

有关详细信息，请参阅 [Windows 窗体和 WPF 属性映射](#)。

## 行为

下表介绍互操作行为。

行 为	支持	不支持
透 明 度	WPF 控件呈现支持透明度。 父 Windows 窗体控件的背景可以成为所承载的 WPF 控件的背景。	不适用。
多 线 程 处 理	支持所有类型的多线程处理。	Windows 窗体和 WPF 技术均采用单线程并发模型。 调试期间，从其他线程调用框架对象会引发一个异常，以强制实施此要求。
安 全 性	所有互操作方案都需要完全信任。	部分信任情况下，不允许任何互操作方案。
可 访 问 性	支持所有辅助功能方案。 当辅助技术产品用于同时包含 Windows 窗体和 WPF 控件的混合应用程序时，可正常工作。	不适用。
剪 贴 板	所有剪贴板操作按常规方式工作。 这包括 Windows 窗体与 WPF 控件之间的剪切和粘贴。	不适用。
拖 放 功 能	所有拖放操作按常规方式工作。 这包括 Windows 窗体和 WPF 控件之间的操作。	不适用。

## 请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [演练：在 WPF 中承载 Windows 窗体控件](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)
- [Windows 窗体和 WPF 属性映射](#)

# Windows 窗体和 WPF 互操作性输入体系结构

项目 • 2023/02/06

WPF 与 Windows 窗体之间的互操作要求这两种技术都具有适当的键盘输入处理。本主题介绍这些技术如何实现键盘和消息处理，以便在混合应用程序中实现平滑互操作。

本主题包含以下小节：

- 无模式窗体和对话框
- WindowsFormsHost 键盘和消息处理
- ElementHost 键盘和消息处理

## 无模式窗体和对话框

在 [WindowsFormsHost](#) 元素上调用 [EnableWindowsFormsInterop](#) 方法以从基于 WPF 的应用程序打开无模式窗体或对话框。

在 [ElementHost](#) 控件上调用 [EnableModelessKeyboardInterop](#) 方法，以在基于 Windows 窗体的应用程序中打开无模式 WPF 页面。

## WindowsFormsHost 键盘和消息处理

当由基于 WPF 的应用程序承载时，Windows 窗体键盘和消息处理包括以下内容：

- [WindowsFormsHost](#) 类从 WPF 消息循环中获取消息，该循环由 [ComponentDispatcher](#) 类实现。
- [WindowsFormsHost](#) 类创建一个代理 Windows 窗体消息循环，以确保进行普通的 Windows 窗体键盘处理。
- [WindowsFormsHost](#) 类实现 [IKeyboardInputSink](#) 接口以协调焦点管理与 WPF。
- [WindowsFormsHost](#) 控件自行注册并启动其消息循环。

以下部分更详细地描述了该过程的这些部分。

### 从 WPF 消息循环中获取消息

`ComponentDispatcher` 类实现 WPF 的消息循环管理器。 `ComponentDispatcher` 类提供挂钩，使外部客户端能够在 WPF 处理消息之前对其进行筛选。

互操作实现处理 `ComponentDispatcher.ThreadFilterMessage` 事件，使 Windows 窗体控件能够在 WPF 控件之前处理消息。

## 代理 Windows 窗体消息循环

默认情况下，`System.Windows.Forms.Application` 类包含 Windows 窗体应用程序的主要消息循环。在互操作期间，Windows 窗体消息循环不处理消息。因此，必须重现此逻辑。`ComponentDispatcher.ThreadFilterMessage` 事件的处理程序执行以下步骤：

1. 使用 `IMessageFilter` 接口筛选消息。
2. 调用 `Control.PreProcessMessage` 方法。
3. 如果需要，转换并调度消息。
4. 如果没有其他控件处理消息，则将消息传递给承载控件。

## IKeyboardInputSink 实现

代理消息循环处理键盘管理。因此，`IKeyboardInputSink.TabInto` 方法是唯一需要在 `WindowsFormsHost` 类中实现的 `IKeyboardInputSink` 成员。

默认情况下，`HwndHost` 类为其 `IKeyboardInputSink.TabInto` 实现返回 `false`。这可以防止从 WPF 控件到 Windows 窗体控件的 Tab 键输入。

`IKeyboardInputSink.TabInto` 方法的 `WindowsFormsHost` 实现执行以下步骤：

1. 查找 `WindowsFormsHost` 控件包含并且可以接收焦点的第一个或最后一个 Windows 窗体控件。控件选项取决于遍历信息。
2. 将焦点设置为控件并返回 `true`。
3. 如果没有控件可以接收焦点，则返回 `false`。

## WindowsFormsHost 注册

创建 `WindowsFormsHost` 控件的窗口句柄时，`WindowsFormsHost` 控件调用一个内部静态方法，该方法为消息循环注册其存在。

在注册期间，`WindowsFormsHost` 控件检查消息循环。如果尚未启动消息循环，则创建 `ComponentDispatcher.ThreadFilterMessage` 事件处理程序。当附加

`ComponentDispatcher.ThreadFilterMessage` 事件处理程序时，消息循环被认为正在运行。

当窗口句柄被销毁时，`WindowsFormsHost` 控件将自己从注册中移除。

## ElementHost 键盘和消息处理

当由 Windows 窗体应用程序承载时，WPF 键盘和消息处理包括以下内容：

- `HwndSource`、`IKeyboardInputSink` 和 `IKeyboardInputSite` 接口实现。
- Tab 键和箭头键。
- 命令键和对话框键。
- Windows 窗体快捷键处理。

以下部分更详细地描述了这些部分。

## 接口实现

在 Windows 窗体中，键盘消息被路由到具有焦点的控件的窗口句柄。在 `ElementHost` 控件中，这些消息被路由到托管元素。为此，`ElementHost` 控件提供了一个 `HwndSource` 实例。如果 `ElementHost` 控件具有焦点，则 `HwndSource` 实例会路由大多数键盘输入，以便 WPF `InputManager` 类可以处理它。

`HwndSource` 类实现 `IKeyboardInputSink` 和 `IKeyboardInputSite` 接口。

键盘互操作依赖于实现 `OnNoMoreTabStops` 方法来处理将焦点移出托管元素的 Tab 键和箭头键输入。

## Tab 键和箭头键

Windows 窗体选择逻辑映射到 `IKeyboardInputSink.TabInto` 和 `OnNoMoreTabStops` 方法以实现 Tab 键和箭头键导航。重写 `Select` 方法可完成此映射。

## 命令键和对话框键

为了让 WPF 第一次有机会处理命令键和对话框键，Windows 窗体命令预处理连接到 `TranslateAccelerator` 方法。重写 `Control.ProcessCmdKey` 方法可连接这两种技术。

使用 `TranslateAccelerator` 方法，托管元素可以处理任何键消息，例如 WM\_KEYDOWN、WM\_KEYUP、WM\_SYSKEYDOWN 或 WM\_SYSKEYUP，包括命令键，例如 Tab、Enter、

Esc 和箭头键。如果未处理键消息，则将其向上发送到 Windows 窗体上次层次结构以进行处理。

## 快捷键处理

若要正确处理快捷键，Windows 窗体快捷键处理必须连接到 WPF [AccessKeyManager](#) 类。此外，所有 WM\_CHAR 消息必须正确路由到托管元素。

由于 [TranslateChar](#) 方法的默认 [HwndSource](#) 实现返回 `false`，因此使用以下逻辑处理 WM\_CHAR 消息：

- 重写 [Control.IsInputChar](#) 方法以确保将所有 WM\_CHAR 消息转发到托管元素。
- 如果按下 Alt 键，则消息为 WM\_SYSCHAR。Windows 窗体不通过 [IsInputChar](#) 方法预处理此消息。因此，将重写 [ProcessMnemonic](#) 方法以查询 WPF [AccessKeyManager](#)，获取已注册的快捷键。如果找到已注册的快捷键，[AccessKeyManager](#) 会对其进行处理。
- 如果未按下 Alt 键，WPF [InputManager](#) 类将处理未处理的输入。如果输入是快捷键，[AccessKeyManager](#) 会对其进行处理。为未处理的 WM\_CHAR 消息处理 [PostProcessInput](#) 事件。

当用户按下 Alt 键时，快捷键视觉提示会显示在整个窗体上。为了支持这种行为，活动窗体上的所有 [ElementHost](#) 控件都会接收 WM\_SYSKEYDOWN 消息，而不管哪个控件具有焦点。

消息仅发送到活动窗体中的 [ElementHost](#) 控件。

## 另请参阅

- [EnableWindowsFormsInterop](#)
- [EnableModelessKeyboardInterop](#)
- [ElementHost](#)
- [WindowsFormsHost](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)
- [WPF 和 Win32 互操作](#)

# WindowsFormsHost 元素的布局注意事项

项目 · 2023/02/06

本主题介绍 [WindowsFormsHost](#) 元素如何与 WPF 布局系统交互。

WPF 和 Windows 窗体支持在窗体或页面上调整和定位元素的不同但相似的逻辑。在 WPF 中创建托管 Windows 窗体控件的混合用户界面 (UI) 时，[WindowsFormsHost](#) 元素集成了两种布局方案。

## WPF 与 Windows 窗体之间的布局差异

WPF 使用与分辨率无关的布局。所有 WPF 布局尺寸均使用与设备无关的像素指定。设备独立像素大小为 96 英寸且与分辨率无关，因此无论是渲染到 72 dpi 的显示器还是渲染到 19,200 dpi 的打印机，你都可以获得类似的结果。

WPF 也基于动态布局。这意味着 UI 元素根据其内容、其父布局容器和可用的屏幕大小在窗体或页面上自行排列。动态布局通过在它们包含的字符串更改长度时自动调整 UI 元素的大小和位置来促进本地化。

Windows 窗体中的布局依赖于设备并且更有可能是静态的。通常，Windows 窗体控件使用以硬件像素指定的尺寸绝对定位在窗体上。但是，Windows 窗体确实支持一些动态布局功能，如下表所述。

布局功能	描述
自动调整大小	某些 Windows 窗体控件会调整自身大小以正确显示其内容。有关详细信息，请参阅 <a href="#">AutoSize 属性概述</a> 。
定位和停靠	Windows 窗体控件支持基于父容器的定位和大小调整。有关详细信息，请参阅 <a href="#">Control.Anchor</a> 和 <a href="#">Control.Dock</a> 。
自动缩放	容器控件根据输出设备的分辨率或默认容器字体的大小（以像素为单位）调整自身及其子项的大小。有关详细信息，请参阅 <a href="#">Windows 窗体中的自动缩放</a> 。
布局容器	<a href="#">FlowLayoutPanel</a> 和 <a href="#">TableLayoutPanel</a> 控件根据其内容排列其子控件并调整大小。

## 布局限制

通常，Windows 窗体控件无法在 WPF 中做到最大程度的缩放和转换。下面列出了当 [WindowsFormsHost](#) 元素尝试将其托管的 Windows 窗体控件集成到 WPF 布局系统时的已知限制。

- 某些情况下，不能调整 Windows 窗体控件的大小，或者大小只能调整为特定尺寸。例如，Windows 窗体 [ComboBox](#) 控件仅支持由控件的字号定义的单一高度。在 WPF 动态布局中，如果元素可以垂直拉伸，则托管的 [ComboBox](#) 控件不会按预期拉伸。
- 不能旋转或倾斜 Windows 窗体控件。如果应用倾斜或旋转转换，[WindowsFormsHost](#) 元素将引发 [LayoutError](#) 事件。如果不处理 [LayoutError](#) 事件，将引发一个 [InvalidOperationException](#) 事件。
- 大多数情况下，Windows 窗体控件不支持按比例缩放。尽管该控件的整体尺寸将会缩放，但其子控件和组件元素可能不会按预期调整大小。此限制取决于每个 Windows 窗体控件支持缩放的程度。此外，不能将 Windows 窗体控件缩放到 0 像素的大小。
- Windows 窗体控件支持自动缩放，在这种情况下，窗体将根据字号自动调整自身及其控件的大小。在 WPF 用户界面中，更改字号不会改变整个布局的大小，但是可动态调整单个元素的大小。

## Z 顺序

在 WPF 用户界面中，可以更改元素的 z 顺序以控制重叠行为。由于承载的 Windows 窗体控件是在单独的 HWND 中绘制的，所以始终在 WPF 元素之上绘制它。

托管 Windows 窗体控件也可在任何 [Adorner](#) 元素之上进行绘制。

## 布局行为

以下部分描述了在 WPF 中托管 Windows 窗体控件时布局行为的特定方面。

### 缩放、单位转换和设备独立

每当 [WindowsFormsHost](#) 元素执行涉及 WPF 和 Windows 窗体尺寸的操作时，都会涉及两个坐标系：WPF 的设备独立像素和 Windows 窗体的硬件像素。因此，必须应用适当的单位和缩放转换才能实现一致的布局。

坐标系之间的转换取决于当前设备分辨率以及应用于 [WindowsFormsHost](#) 元素或其上级的任何布局或呈现转换。

如果输出设备为 96 dpi 且未将缩放应用于 [WindowsFormsHost](#) 元素，则一个设备独立像素等于一个硬件像素。

所有其他情况都需要坐标系缩放。不调整托管控件的大小。相反，[WindowsFormsHost](#) 元素会尝试缩放托管控件及其所有子控件。由于 Windows 窗体只是部分支持缩放，因此 [WindowsFormsHost](#) 元素只能缩放到特定控件支持的程度。

替代 [ScaleChild](#) 方法以提供托管 Windows 窗体控件的自定义缩放行为。

除缩放之外，[WindowsFormsHost](#) 元素还处理舍入和溢出情况，如下表所述。

转换 描述	
舍入	WPF 设备独立像素尺寸指定为 <code>double</code> ，Windows 窗体硬件像素尺寸指定为 <code>int</code> 。如果基于 <code>double</code> 的尺寸转换为基于 <code>int</code> 的尺寸，则 <a href="#">WindowsFormsHost</a> 元素会使用标准舍入，因此小于 0.5 的小数值会向下舍入为 0。
溢出	当 <a href="#">WindowsFormsHost</a> 元素从 <code>double</code> 值转换为 <code>int</code> 值时，可能会发生溢出。大于 <code>.MaxValue</code> 的值设置为 <code>.MaxValue</code> 。

## 与布局相关的属性

控制 Windows 窗体控件和 WPF 元素中的布局行为的属性由 [WindowsFormsHost](#) 元素适当地映射。有关详细信息，请参阅 [Windows 窗体和 WPF 属性映射](#)。

## 托管控件中的布局更改

托管 Windows 窗体控件中的布局更改将传播到 WPF 以触发布局更新。

[WindowsFormsHost](#) 上的 [InvalidateMeasure](#) 方法确保托管控件中的布局更改促使 WPF 布局引擎运行。

## 连续调整大小的 Windows 窗体控件

支持连续缩放的 Windows 窗体控件与 WPF 布局系统完全交互。[WindowsFormsHost](#) 元素像照常使用 [MeasureOverride](#) 和 [ArrangeOverride](#) 方法来调整和排列托管的 Windows 窗体控件。

## 大小调整算法

[WindowsFormsHost](#) 元素使用以下过程调整托管控件的大小：

1. WindowsFormsHost 元素替代 MeasureOverride 和 ArrangeOverride 方法。
2. 为了确定托管控件的大小，MeasureOverride 方法调用托管控件的 GetPreferredSize 方法，其中包含从传递给 MeasureOverride 方法的约束转换而来的约束。
3. ArrangeOverride 方法尝试将托管控件设置为给定大小约束。
4. 如果托管控件的 Size 属性与指定的约束匹配，则托管控件的大小就是约束大小。

如果 Size 属性与指定的约束不匹配，则托管控件不支持连续大小调整。例如，MonthCalendar 控件仅允许离散大小。此控件的允许大小由高度和宽度的整数（表示月数）组成。在这种情况下，WindowsFormsHost 元素的行为如下所示：

- 如果 Size 属性返回的大小大于指定的约束，则 WindowsFormsHost 元素会剪切托管控件。高度和宽度是分开处理的，因此可以从任意方向剪裁托管控件。
- 如果 Size 属性返回的大小小于指定的约束，则 WindowsFormsHost 接受此大小值并将该值返回给 WPF 布局系统。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [演练：在 WPF 中排列 Windows 窗体控件](#)
- [有关在 WPF 中排列 Windows 窗体控件的示例](#)
- [Windows 窗体和 WPF 属性映射](#)
- [迁移和互操作性](#)

# Windows 窗体控件和等效的 WPF 控件

项目 • 2022/09/27

许多 Windows 窗体控件具有等效的 WPF 控件，但某些 Windows 窗体控件在 WPF 中没有等效控件。本主题比较两种技术提供的控件类型。

你始终可以使用互操作来托管在基于 WPF 的应用程序中没有等效项的 Windows 窗体控件。

下表显示具有等效 WPF 控件功能的 Windows 窗体控件和组件。

Windows 窗体控件	WPF 等效控件	注解
BindingNavigator	没有等效控件。	
BindingSource	CollectionViewSource	
Button	Button	
CheckBox	CheckBox	
CheckedListBox	ListBox 及其构成。	
ColorDialog	没有等效控件。	
ComboBox	ComboBox	ComboBox 不支持自动完成。
ContextMenuStrip	ContextMenu	
DataGridView	DataGrid	
DateTimePicker	DatePicker	
DomainUpDown	TextBox 和两个 RepeatButton 控件。	
ErrorProvider	没有等效控件。	
FlowLayoutPanel	WrapPanel 或 StackPanel	
FolderBrowserDialog	没有等效控件。	
FontDialog	没有等效控件。	
Form	Window	Window 不支持子窗口。
GroupBox	GroupBox	
HelpProvider	没有等效控件。	无 F1 帮助。“这是什么”帮助被工具提示取代。
HScrollBar	ScrollBar	滚动内置于容器控件中。
ImageList	没有等效控件。	
Label	Label	
LinkLabel	没有等效控件。	可以使用 <a href="#">Hyperlink</a> 类在流内容中托管超链接。
ListBox	ListBox	
ListView	ListView	ListView 控件提供只读详细信息视图。

Windows 窗体控件	WPF 等效控件	注解
MaskedTextBox	没有等效控件。	
MenuStrip	Menu	Menu 控件样式可以近似于 System.Windows.Forms.ToolStripProfessionalRenderer 类的行为和外观。
MonthCalendar	Calendar	
NotifyIcon	没有等效控件。	
NumericUpDown	TextBox 和两个 RepeatButton 控件。	
OpenFileDialog	OpenFileDialog	OpenFileDialog 类是环绕 Win32 控件的 WPF 包装器。
PageSetupDialog	没有等效控件。	
Panel	Canvas	
PictureBox	Image	
PrintDialog	PrintDialog	
PrintDocument	没有等效控件。	
PrintPreviewControl	DocumentViewer	
PrintPreviewDialog	没有等效控件。	
ProgressBar	ProgressBar	
PropertyGrid	没有等效控件。	
RadioButton	RadioButton	
RichTextBox	RichTextBox	
SaveFileDialog	SaveFileDialog	SaveFileDialog 类是环绕 Win32 控件的 WPF 包装器。
ScrollableControl	ScrollViewer	
SoundPlayer	MediaPlayer	
SplitContainer	GridSplitter	
StatusStrip	StatusBar	
TabControl	TabControl	
TableLayoutPanel	Grid	
TextBox	TextBox	
Timer	DispatcherTimer	
ToolStrip	ToolBar	
ToolStripContainer	ToolBar 及其构成。	
ToolStripDropDown	ToolBar 及其构成。	

Windows 窗体控件	WPF 等效控件	注解
ToolStripDropDownMenu	ToolBar 及其构成。	
ToolStripPanel	ToolBar 及其构成。	
ToolTip	ToolTip	
TrackBar	Slider	
TreeView	TreeView	
UserControl	UserControl	
VScrollBar	ScrollBar	滚动内置于容器控件中。
WebBrowser	Frame, System.Windows.Controls.WebBrowser	Frame 控件可托管 HTML 页面。 从 .NET Framework 3.5 SP1 开始 , System.Windows.Controls.WebBrowser 控件可托管 HTML 页面并支持 Frame 控件。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [适用于 Windows 窗体开发人员的 WPF 设计器](#)
- [演练 : 在 WPF 中承载 Windows 窗体控件](#)
- [演练 : 在 Windows 窗体中承载 WPF 复合控件](#)
- [迁移和互操作性](#)

# Windows 窗体和 WPF 属性映射

项目 • 2023/02/06

Windows 窗体和 WPF 技术具有两种相似但不同的属性模型。 属性映射支持两种体系结构之间的互操作，并提供以下功能：

- 可以轻松地将主机环境中的相关属性更改映射到托管控件或元素。
- 提供映射最常用属性的默认处理。
- 允许轻松删除、替代或扩展默认属性。
- 确保自动检测主机上的属性值更改并将其转换为托管控件或元素。

## ① 备注

属性更改事件不会传播到主机控件或元素层次结构中。如果属性的本地值因直接设置、样式、继承、数据绑定或其他更改属性值的机制而没有更改，则不会执行属性转换。

使用 [WindowsFormsHost](#) 元素上的 [PropertyMap](#) 属性和 [ElementHost](#) 控件上的 [PropertyMap](#) 属性来访问属性映射。

## 使用 WindowsFormsHost 元素进行属性映射

[WindowsFormsHost](#) 元素使用以下转换表将默认 WPF 属性转换为 Windows 窗体等效项。

Windows Presentation Foundation 托管	Windows 窗体	互操作行为
Background (System.Windows.Media.Brush)	BackColor (System.Drawing.Color)	<p><a href="#">WindowsFormsHost</a> 元素设置托管控件的 <a href="#">BackColor</a> 属性和 <a href="#">BackgroundImage</a> 属性。 使用以下规则执行映射：</p> <p>- 如果 <a href="#">Background</a> 为纯色，则将被转换并用于设置托管控件的 <a href="#">BackColor</a> 属性。<a href="#">BackColor</a> 属性未在托管控件上设置，因为托管控件可以继承 <a href="#">BackColor</a> 属性的值。 注意：托管控件不支持透明度。 分配给 <a href="#">BackColor</a> 的任何颜色都必须完全不透明，Alpha 值为 0xFF。</p> <p>- 如果 <a href="#">Background</a> 不是纯色，<a href="#">WindowsFormsHost</a> 控件会从 <a href="#">Background</a> 属性创建位图。<a href="#">WindowsFormsHost</a> 控件将此位图分配给托管控件的 <a href="#">BackgroundImage</a> 属性。 这提供了类似于透明度的效果。 注意：可以替代此行为，也可以删除 <a href="#">Background</a> 属性映射。</p>
Cursor	Cursor	<p>如果尚未重新分配默认映射，则 <a href="#">WindowsFormsHost</a> 控件会遍历其上级层次结构，直到找到设置了 <a href="#">Cursor</a> 属性的上级。 此值将转换为最接近的对应 Windows 窗体游标。</p> <p>如果 <a href="#">ForceCursor</a> 属性的默认映射尚未重新分配，则遍历将在 <a href="#">ForceCursor</a> 设置为 <code>true</code> 的第一个上级处停止。</p>
FlowDirection (System.Windows.FlowDirection)	RightToLeft (System.Windows.Forms.RightToLeft)	<p><a href="#">LeftToRight</a> 映射到 <a href="#">No</a>。</p> <p><a href="#">RightToLeft</a> 映射到 <a href="#">Yes</a>。</p> <p><a href="#">Inherit</a> 未映射。</p> <p><a href="#">FlowDirection.RightToLeft</a> 映射到 <a href="#">RightToLeft.Yes</a>。</p>

Windows Presentation Foundation 托管	Windows 窗体	互操作行为
FontStyle	托管控件的 <a href="#">System.Drawing.Font</a> 上的 <a href="#">Style</a>	WPF 属性集将转换为相应的 <a href="#">Font</a> 。当其中一个属性发生更改时，将创建一个新的 <a href="#">Font</a> 。对于 <a href="#">Normal</a> : <a href="#">Italic</a> 已禁用。对于 <a href="#">Italic</a> 或 <a href="#">Oblique</a> : <a href="#">Italic</a> 已启用。
FontWeight	托管控件的 <a href="#">System.Drawing.Font</a> 上的 <a href="#">Style</a>	WPF 属性集将转换为相应的 <a href="#">Font</a> 。当其中一个属性发生更改时，将创建一个新的 <a href="#">Font</a> 。对于 <a href="#">Black</a> 、 <a href="#">Bold</a> 、 <a href="#">DemiBold</a> 、 <a href="#">ExtraBold</a> 、 <a href="#">Heavy</a> 、 <a href="#">Medium</a> 、 <a href="#">SemiBold</a> 或 <a href="#">UltraBold</a> : <a href="#">Bold</a> 已启用。对于 <a href="#">ExtraLight</a> 、 <a href="#">Light</a> 、 <a href="#">Normal</a> 、 <a href="#">Regular</a> 、 <a href="#">Thin</a> 或 <a href="#">UltraLight</a> : <a href="#">Bold</a> 已启用。
FontFamily	<a href="#">Font</a>	WPF 属性集将转换为相应的 <a href="#">Font</a> 。当其中一个属性发生更改时，将创建一个新的 <a href="#">Font</a> 。托管的 Windows 窗体控件根据字号调整大小。
FontSize	( <a href="#">System.Drawing.Font</a> )	
FontStretch		WPF 中的字号表示为 1/96 英寸，Windows 窗体中的字号表示为 1/72 英寸。相应的转换如下：
FontStyle		Windows Forms 窗体字号 = WPF 字号 * 72.0 / 96.0。
FontWeight		
Foreground ( <a href="#">System.Windows.Media.Brush</a> )	<a href="#">ForeColor</a> ( <a href="#">System.Drawing.Color</a> )	使用以下规则执行 <a href="#">Foreground</a> 属性映射：  - 如果 <a href="#">Foreground</a> 为 <a href="#">SolidColorBrush</a> ，则将 <a href="#">Color</a> 用于 <a href="#">ForeColor</a> 。 - 如果 <a href="#">Foreground</a> 为 <a href="#">GradientBrush</a> ，请将具有最低偏移值的 <a href="#">GradientStop</a> 的颜色用于 <a href="#">ForeColor</a> 。 - 对于任何其他 <a href="#">Brush</a> 类型，保留 <a href="#">ForeColor</a> 不变。这意味着使用默认值。
IsEnabled	<a href="#">Enabled</a>	设置 <a href="#">IsEnabled</a> 时， <a href="#">WindowsFormsHost</a> 元素设置托管控件的 <a href="#">Enabled</a> 属性。
Padding	<a href="#">Padding</a>	将托管 Windows 窗体控件的 <a href="#">Padding</a> 属性的所有四个值都设置为相同的 <a href="#">Thickness</a> 值。  - 大于 <a href="#">MaxValue</a> 的值设置为 <a href="#">MaxValue</a> 。 - 小于 <a href="#">MinValue</a> 的值设置为 <a href="#">MinValue</a> 。
Visibility	<a href="#">Visible</a>	- <a href="#">Visible</a> 映射到 <a href="#">Visible = true</a> 。托管 Windows 窗体控件可见。建议不要将托管控件的 <a href="#">Visible</a> 属性显式设置为 <a href="#">false</a> 。 - <a href="#">Collapsed</a> 映射到 <a href="#">Visible = true</a> 或 <a href="#">false</a> 。托管 Windows 窗体控件不会绘制，并且其区域会折叠。 - <a href="#">Hidden</a> ：托管 Windows 窗体控件占用布局中的空间，但不可见。在本例中， <a href="#">Visible</a> 属性设置为 <a href="#">true</a> 。建议不要将托管控件的 <a href="#">Visible</a> 属性显式设置为 <a href="#">false</a> 。

[WindowsFormsHost](#) 元素完全支持容器元素上的附加属性。

有关详细信息，请参阅[演练：使用 WindowsFormsHost 元素映射属性](#)。

## 对父属性的更新

对大多数父属性的更改会导致向托管子控件发出通知。以下列表描述了在属性值发生更改时不会发出通知的属性。

- [Background](#)
- [Cursor](#)
- [ForceCursor](#)

- [Visibility](#)

例如，如果更改 [WindowsFormsHost](#) 元素的 [Background](#) 属性的值，则托管控件的 [BackColor](#) 属性不会更改。

## 使用 ElementHost 控件进行属性映射

以下属性提供内置的更改通知。 映射这些属性时，请勿调用 [OnPropertyChanged](#) 方法：

- AutoSize
- BackColor
- BackgroundImage
- BackgroundImageLayout
- BindingContext
- CausesValidation
- ContextMenu
- ContextMenuStrip
- 游标
- 靠接
- 已启用
- 字体
- ForeColor
- 位置
- Margin
- 填充
- Parent
- 区域
- RightToLeft
- 大小
- TabIndex
- TabStop
- 文本
- 可见

[ElementHost](#) 控件使用以下转换表将默认 Windows 窗体属性转换为 WPF 等效项。

有关详细信息，请参阅[演练：使用 ElementHost 控件映射属性](#)。

Windows 窗体托管	Windows Presentation Foundation	互操作行为
BackColor (System.Drawing.Color)	Background (System.Windows.Media.Brush) 托管元素上	设置此属性将强制使用 <code>ImageBrush</code> 重画。如果 <code>BackColorTransparent</code> 属性设置为 <code>false</code> （默认值），则此 <code>ImageBrush</code> 基于 <code>ElementHost</code> 控件的外观，包括其 <code>BackColor</code> 、 <code>BackgroundImage</code> 、 <code>BackgroundImageLayout</code> 属性以及任何附加的画图处理程序。  如果 <code>BackColorTransparent</code> 属性设置为 <code>true</code> ，则 <code>ImageBrush</code> 基于 <code>ElementHost</code> 控件的父控件的外观，包括父控件的 <code>BackColor</code> 、 <code>BackgroundImage</code> 、 <code>BackgroundImageLayout</code> 属性以及任何附加的画图处理程序。
BackgroundImage (System.Drawing.Image)	Background (System.Windows.Media.Brush) 托管元素上	设置此属性会导致 <code>BackColor</code> 映射所述的相同行为。
BackgroundImageLayout	Background (System.Windows.Media.Brush) 托管元素上	设置此属性会导致 <code>BackColor</code> 映射所述的相同行为。
Cursor (System.Windows.Forms.Cursor)	Cursor (System.Windows.Input.Cursor)	Windows 窗体标准游标将转换为相应的 WPF 标准游标。如果 Windows 窗体不是标准游标，则会分配默认值。
Enabled	IsEnabled	设置 <code>Enabled</code> 时， <code>ElementHost</code> 控件设置托管元素的 <code>IsEnabled</code> 属性。
Font (System.Drawing.Font)	FontFamily FontSize FontStretch FontStyle FontWeight	<code>Font</code> 值将转换为相应的 WPF 字体属性集。
Bold	托管元素的 <code>FontWeight</code>	如果 <code>Bold</code> 为 <code>true</code> ，则 <code>FontWeight</code> 设置为 <code>Bold</code> 。  如果 <code>Bold</code> 为 <code>false</code> ，则 <code>FontWeight</code> 设置为 <code>Normal</code> 。
Italic	托管元素的 <code>FontStyle</code>	如果 <code>Italic</code> 为 <code>true</code> ，则 <code>FontStyle</code> 设置为 <code>Italic</code> 。  如果 <code>Italic</code> 为 <code>false</code> ，则 <code>FontStyle</code> 设置为 <code>Normal</code> 。
Strikeout	托管元素的 <code>TextDecorations</code>	仅在托管 <code>TextBlock</code> 控件时适用。
Underline	托管元素的 <code>TextDecorations</code>	仅在托管 <code>TextBlock</code> 控件时适用。
RightToLeft (System.Windows.Forms.RightToLeft)	FlowDirection (FlowDirection)	<code>No</code> 映射到 <code>LeftToRight</code> 。  <code>Yes</code> 映射到 <code>RightToLeft</code> 。
Visible	Visibility	<code>ElementHost</code> 控件使用以下规则设置托管元素的 <code>Visibility</code> 属性：  - <code>Visible = true</code> 映射到 <code>Visible</code> 。 - <code>Visible = false</code> 映射到 <code>Hidden</code> 。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [WPF 和 Win32 互操作](#)
- [WPF 和 Windows 窗体互操作](#)
- [演练：使用 WindowsFormsHost 元素映射属性](#)
- [演练：使用 ElementHost 控件映射属性](#)

# 混合应用程序疑难解答

项目 · 2022/09/27

本主题列出了在创作同时使用 WPF 和 Windows 窗体技术的混合应用程序时可能发生的一些常见问题。

## 重叠控件

控件可能不按预期的方式重叠。 Windows 窗体为每个控件使用单独的 HWND。 WPF 为一个页面上的所有内容使用一个 HWND。 这一实现差异会导致意外的重叠行为。

WPF 中托管的 Windows 窗体控件始终显示在 WPF 内容顶部。

[ElementHost](#) 控件中托管的 WPF 内容以 [ElementHost](#) 控件的 z 顺序显示。 可以重叠 [ElementHost](#) 控件，但托管的 WPF 内容不能组合或交互。

## 子属性

[WindowsFormsHost](#) 和 [ElementHost](#) 类只能托管单个子控件或元素。 若要承载多个控件或元素，则必须使用容器作为子内容。 例如，可以向 [System.Windows.Forms.Panel](#) 控件添加 Windows 窗体按钮和复选框控件，然后将该面板分配给 [WindowsFormsHost](#) 控件的 [Child](#) 属性。 但是，不能将按钮和复选框控件分别添加到相同的 [WindowsFormsHost](#) 控件。

## 扩展

WPF 和 Windows 窗体具有不同的缩放模型。 某些 WPF 缩放变换对于 Windows 窗体控件是有意义的，但其他变换是无意义的。 例如，将 Windows 窗体控件缩放到 0 是可行的，但如果尝试将同一控件重新缩放回非零值，该控件的大小仍然为 0。 有关详细信息，请参阅 [WindowsFormsHost 元素的布局注意事项](#)。

## 适配器

在使用 [WindowsFormsHost](#) 和 [ElementHost](#) 类时可能会发生混乱，因为它们包含一个隐藏容器。[WindowsFormsHost](#) 和 [ElementHost](#) 类都具有一个名为“适配器”的隐藏容器，用来托管内容。 对于 [WindowsFormsHost](#) 元素，适配器派生自 [System.Windows.Forms.ContainerControl](#) 类。 对于 [ElementHost](#) 控件，适配器派生自 [DockPanel](#) 元素。 如果你发现其他互操作主题中提到了适配器，那么所讨论的就是此容器。

## 嵌套

不支持在 [ElementHost](#) 控件内嵌套 [WindowsFormsHost](#) 元素。也不支持在 [WindowsFormsHost](#) 元素内嵌套 [ElementHost](#) 控件。

## 侧重点

焦点在 WPF 和 Windows 窗体中的工作方式是不同的，这意味着混合应用程序中可能发生焦点问题。例如，如果 [WindowsFormsHost](#) 元素内部具有焦点，那么，当最小化并还原页面，或者显示模式对话框时，[WindowsFormsHost](#) 元素内部的焦点可能会丢失。

[WindowsFormsHost](#) 元素仍然具有焦点，但该元素内部的控件可能不具有焦点。

焦点还会影响数据验证。验证在 [WindowsFormsHost](#) 元素中有效，但当按 Tab 离开 [WindowsFormsHost](#) 元素或在两个不同的 [WindowsFormsHost](#) 元素之间切换时，验证无效。

## 属性映射

某些属性映射需要先进行大量的转译，才能将 WPF 和 Windows 窗体技术之间不同的实现桥接起来。属性映射可使代码对字体、颜色和其他属性的更改做出反应。通常，属性映射的工作方式是侦听 *PropertyChanged* 事件或 *OnPropertyChanged* 调用，然后在子控件或其适配器上设置适当的属性。有关详细信息，请参阅 [Windows 窗体和 WPF 属性映射](#)。

## 所承载内容上的布局相关属性

分配 [WindowsFormsHost.Child](#) 或 [ElementHost.Child](#) 属性后，会自动设置托管内容上的几个与布局相关的属性。更改这些内容属性可能会导致意外的布局行为。

停靠托管内容以填充 [WindowsFormsHost](#) 和 [ElementHost](#) 父级。为了启用此填充行为，在设置子属性时，将设置多个属性。下表列出了由 [ElementHost](#) 和 [WindowsFormsHost](#) 类设置的内容属性。

宿主类	内容属性
-----	------

宿主类	内容属性
ElementHost	Height Width Margin VerticalAlignment HorizontalAlignment
WindowsFormsHost	Margin Dock AutoSize Location MaximumSize

请勿在所承载的内容上直接设置这些属性。有关详细信息，请参阅 [WindowsFormsHost 元素的布局注意事项](#)。

## 导航应用程序

导航应用程序不能保持用户状态。在导航应用程序中使用 [WindowsFormsHost](#) 元素时，该元素将重新创建其控件。当用户通过导航操作离开托管 [WindowsFormsHost](#) 元素的页面，然后又返回到该页面时，将发生重新创建子控件的操作。用户已经键入的所有内容都将丢失。

## 消息循环互操作

在使用 Windows 窗体消息循环时，可能无法按照预期方式处理消息。

[EnableWindowsFormsInterop](#) 方法由 [WindowsFormsHost](#) 构造函数调用。此方法将向 WPF 消息循环中添加消息筛选器。如果 [System.Windows.Forms.Control](#) 是消息的目标，则此筛选器会调用 [Control.PreProcessMessage](#) 方法并转换/调度该消息。

如果在使用 [Application.Run](#) 的 Windows 窗体消息循环中显示 [Window](#)，则除非调用 [EnableModelessKeyboardInterop](#) 方法，否则不能键入任何内容。

[EnableModelessKeyboardInterop](#) 方法采用 [Window](#) 并添加 [System.Windows.Forms.IMessageFilter](#)，将与密钥相关的消息重新路由到 WPF 消息循环。有关详细信息，请参阅 [Windows 窗体和 WPF 互操作性输入体系结构](#)。

# 不透明度和分层

[HwndHost](#) 类不支持分层。这意味着在 [WindowsFormsHost](#) 元素上设置 [Opacity](#) 属性不起作用，并且不会与将 [AllowsTransparency](#) 设置为 `true` 的其他 WPF 窗口产生混合效果。

## 释放

未正确释放类可能会泄漏资源。在混合应用程序中，请确保释放 [WindowsFormsHost](#) 和 [ElementHost](#) 类，否则可能会泄漏资源。Windows 窗体在其非模式 [Form](#) 父级关闭时释放 [ElementHost](#) 控件。WPF 会在应用程序关闭时释放 [WindowsFormsHost](#) 元素。可以在 Windows 窗体消息循环中的 [Window](#) 中显示 [WindowsFormsHost](#) 元素。在这种情况下，代码可能不会收到应用程序正在关闭的通知。

## 启用视觉样式

可能无法启用 Windows 窗体控件上的 Microsoft Windows XP 视觉样式。

[Application.EnableVisualStyles](#) 方法在 Windows 窗体应用程序的模板中调用。尽管默认情况下不会调用此方法，但如果使用 Visual Studio 创建项目，并且 Comctl32.dll 版本 6.0 可用，将会获得控件的 Microsoft Windows XP 视觉样式。在线程上创建句柄之前，必须先调用 [EnableVisualStyles](#) 方法。有关详细信息，请参阅[如何：在混合应用程序中启用视觉样式](#)。

## 授权控件

授权的 Windows 窗体控件会在消息框中向用户显示许可信息，对于混合应用程序，这可能会导致意外行为。某些授权控件会显示一个对话框来响应创建句柄的操作。例如，授权控件可能会通知用户需要许可证，或者用户还可以试用该控件三次。

[WindowsFormsHost](#) 元素派生自 [HwndHost](#) 类，子控件的句柄将在 [BuildWindowCore](#) 方法内创建。[HwndHost](#) 类不允许在 [BuildWindowCore](#) 方法中处理消息，但显示对话框会导致发送消息。要启用此许可方案，请先调用控件上的 [Control.CreateControl](#) 方法，然后再将其分配为 [WindowsFormsHost](#) 元素的子级。

## WPF 设计器

可以使用适用于 Visual Studio 的 WPF 设计器设计 WPF 内容。以下部分列出了在使用 WPF 设计器创作混合应用程序时可能发生的一些常见问题。

## 在设计时忽略 BackColorTransparent

[BackColorTransparent](#) 属性在设计时可能无法按预期工作。

如果 WPF 控件不在可见的父级上，WPF 运行时会忽略 [BackColorTransparent](#) 值。 可能忽略 [BackColorTransparent](#) 的原因是，[ElementHost](#) 对象是在单独的 [AppDomain](#) 中创建的。 但在运行应用程序时，[BackColorTransparent](#) 会按预期工作。

## 删除 obj 文件夹时出现设计时错误列表

如果删除 obj 文件夹，会出现设计时错误列表。

使用 [ElementHost](#) 进行设计时，Windows 窗体设计器将使用项目的 obj 文件夹内 Debug 或 Release 文件夹中生成的文件。 如果删除这些文件，将出现设计时错误列表。 若要解决此问题，请重新生成项目。 有关详细信息，请参阅 [Windows 窗体设计器中的设计时错误](#)。

## ElementHost 和 IME

[ElementHost](#) 中托管的 WPF 控件当前不支持 [ImeMode](#) 属性。 托管的控件将忽略对 [ImeMode](#) 所做的更改。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [WPF 设计器中的互操作性](#)
- [Windows 窗体和 WPF 互操作性输入体系结构](#)
- [如何：在混合应用程序中启用视觉对象样式](#)
- [WindowsFormsHost 元素的布局注意事项](#)
- [Windows 窗体和 WPF 属性映射](#)
- [Windows 窗体设计器中的设计时错误](#)
- [迁移和互操作性](#)

# 演练：在 WPF 中承载 Windows 窗体控件

项目 • 2022/09/27

WPF 提供了许多具有丰富功能集的控件。但是，你有时可能希望在 WPF 页上使用 Windows 窗体控件。例如，你可能已经为现有的 Windows 窗体控件花费了大量金钱，或者你的 Windows 窗体控件只能提供特殊的功能。

本演练演示如何使用代码在 WPF 页上承载 Windows 窗体 [System.Windows.Forms.MaskedTextBox 控件](#)。

有关本演练中介绍的任务的完整代码列表，请参阅[在 WPF 示例中承载 Windows 窗体控件](#)。

## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 承载 Windows 窗体控件

### 承载 MaskedTextBox 控件

1. 创建名为 `HostingWfInWpf` 的 WPF 应用程序项目。
2. 添加对下列程序集的引用。
  - WindowsFormsIntegration
  - System.Windows.Forms
3. 在 WPF 设计器中打开 `MainWindow.xaml`。
4. 将 `Grid` 元素命名为 `grid1`。

```
XAML  
<Grid Name="grid1">  
</Grid>
```

5. 在设计视图或 XAML 视图中，选择 `Window` 元素。
6. 在“属性”窗口中，单击“事件”选项卡。

7. 双击“CellDoubleClick”事件 [Loaded](#)。

8. 插入以下代码以处理 [Loaded](#) 事件。

```
C#  
  
private void Window_Loaded(object sender, RoutedEventArgs e)  
{  
    // Create the interop host control.  
    System.Windows.Forms.Integration.WindowsFormsHost host =  
        new System.Windows.Forms.Integration.WindowsFormsHost();  
  
    // Create the MaskedTextBox control.  
    MaskedTextBox mtbDate = new MaskedTextBox("00/00/0000");  
  
    // Assign the MaskedTextBox control as the host control's child.  
    host.Child = mtbDate;  
  
    // Add the interop host control to the Grid  
    // control's collection of child controls.  
    this.grid1.Children.Add(host);  
}
```

9. 在文件的顶部，添加以下 [Imports](#) 或 [using](#) 语句。

```
C#  
  
using System.Windows.Forms;
```

10. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：使用 XAML 在 WPF 中承载 Windows 窗体控件](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)
- [Windows 窗体控件和等效的 WPF 控件](#)
- [在 WPF 示例中承载 Windows 窗体控件](#)

# 演练：使用 XAML 在 WPF 中承载 Windows 窗体控件

项目 • 2023/02/06

WPF 提供了许多具有丰富功能集的控件。但是，你有时可能希望在 WPF 页上使用 Windows 窗体控件。例如，你可能已经为现有的 Windows 窗体控件花费了大量金钱，或者你的 Windows 窗体控件只能提供特殊的功能。

本演练演示如何使用 XAML 在 WPF 页面上托管 Windows 窗体 [System.Windows.Forms.MaskedTextBox 控件](#)。

有关本演练中介绍的任务的完整代码列表，请参阅[使用 XAML 在 WPF 中承载 Windows 窗体控件的示例](#)。

## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 承载 Windows 窗体控件

### 承载 MaskedTextBox 控件

1. 创建名为 `HostingWfInWpfWithXaml` 的 WPF 应用程序项目。
2. 添加对下列程序集的引用。
  - WindowsFormsIntegration
  - System.Windows.Forms
3. 在 WPF 设计器中打开 `MainWindow.xaml`。
4. 在 `Window` 元素中，添加以下命名空间映射。`wf` 命名空间映射建立对包含 Windows 窗体控件的程序集的引用。

XAML

```
xmlns:wf="clr-  
namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. 在 `Grid` 元素中，添加以下 XAML。

MaskedTextBox 控件创建为 WindowsFormsHost 控件的子级。

#### XAML

```
<Grid>

    <WindowsFormsHost>
        <wf:MaskedTextBox x:Name="mtbDate" Mask="00/00/0000"/>
    </WindowsFormsHost>

</Grid>
```

6. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 WPF 中承载 Windows 窗体控件](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)
- [Windows 窗体控件和等效的 WPF 控件](#)
- [使用 XAML 在 WPF 中承载 Windows 窗体控件的示例](#)

# 演练：在 WPF 中托管 Windows 窗体复合控件

项目 • 2022/09/27

WPF 应用程序，而不是从头开始重写它。 最常见的情况是当你拥有现有的 Windows 窗体控件时。 在某些情况下，你甚至可能无法使用这些控件的源代码。 WPF 提供了一个用于承载 WPF 应用程序中的此类控件的简单过程。 例如，你可以将 WPF 用于大部分编程，同时承载专用 [DataGridView](#) 控件。

本演练将引导你完成承载 Windows 窗体复合控件以在 WPF 应用程序中执行数据输入的应用程序。 复合控件打包在一个 DLL 中。 此常规步骤可扩展到更复杂的应用程序和控件。 本演练旨在实现与[演练：在 Window 窗体中承载 WPF 复合控件](#)几乎完全相同的外观和功能。 主要区别在于承载方案是相反的。

本演练分为两个部分。 第一部分简要介绍 Windows 窗体复合控件的实现。 第二部分详细讨论了如何在 WPF 应用程序中承载复合控件、从控件接收事件以及访问控件的某些属性。

本演练涉及以下任务：

- 实现 Windows 窗体复合控件。
- 实现 WPF 主机应用程序。

有关本演练中介绍的任务的完整代码列表，请参阅[在 WPF 示例中承载 Windows 窗体复合控件](#)。

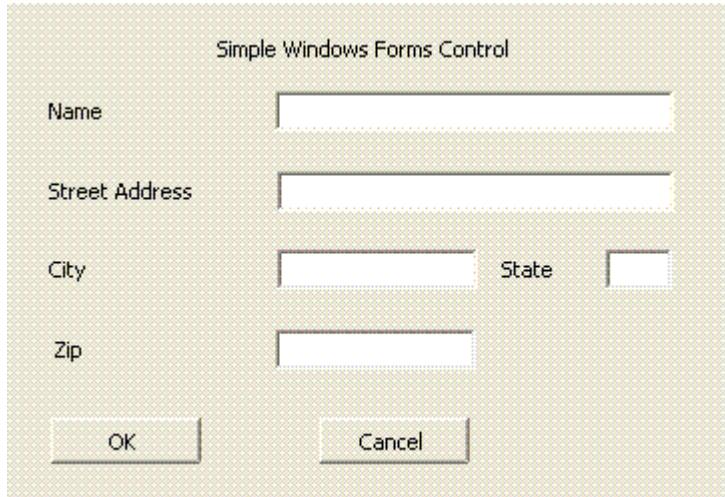
## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 实现 Windows 窗体复合控件

本示例中所用的 Windows 复合控件是一种简单的数据输入形式。 此窗体需要用户名和地址，然后使用自定义事件将该信息返回到主机。 下图显示呈现的控件。

下图显示了一个 Windows 窗体复合控件：



## 创建项目

启动项目：

1. 启动 Visual Studio，然后打开“新建项目”对话框。
2. 在“窗口类别”中，选择“Windows 窗体控件库”模板。
3. 将新项目命名为 `MyControls`。
4. 对于位置，指定可以方便命名的顶层文件夹，如 `WpfHostingWindowsFormsControl`。随后，将主机应用程序放在此文件夹中。
5. 单击“确定”以创建该项目。默认项目包含一个名为 `UserControl1` 的控件。
6. 在解决方案资源管理器中，将 `UserControl1` 重命名为 `MyControl1`。

项目应具有对以下系统 DLL 的引用。如果默认不包含其中任何 DLL，则将它们添加到项目中。

- 系统
- `System.Data`
- `System.Drawing`
- `System.Windows.Forms`
- `System.Xml`

## 将控件添加到窗体

向窗体添加控件：

- 在设计器中打开 `MyControl1`。

在窗体上添加五个 `Label` 控件及其对应的 `TextBox` 控件，它们的大小和排列方式如上图所示。在此示例中，控件 `TextBox` 命名为：

- `txtName`
- `txtAddress`
- `txtCity`
- `txtState`
- `txtZip`

添加两个标记为“确认”和“取消”的 `Button` 控件。在此示例中，按钮名称分别是 `btnOK` 和 `btnCancel`。

## 实现支持代码

在代码视图中打开窗体。控件通过引发自定义 `OnButtonClick` 事件将收集的数据返回到其宿主。数据包含在事件自变量对象中。以下代码演示事件和委托声明。

将以下代码添加到 `MyControl1` 类。

```
C#  
  
public delegate void MyControlEventHandler(object sender, MyControlEvents args);  
public event MyControlEventHandler OnButtonClick;
```

`MyControlEvents` 类包含要返回到宿主的信息。

将以下类添加到窗体中。

```
C#  
  
public class MyControlEvents : EventArgs  
{  
    private string _Name;  
    private string _StreetAddress;  
    private string _City;  
    private string _State;  
    private string _Zip;  
    private bool _IsOK;  
  
    public MyControlEvents(bool result,
```

```

        string name,
        string address,
        string city,
        string state,
        string zip)
{
    _IsOK = result;
    _Name = name;
    _StreetAddress = address;
    _City = city;
    _State = state;
    _Zip = zip;
}

public string MyName
{
    get { return _Name; }
    set { _Name = value; }
}
public string MyStreetAddress
{
    get { return _StreetAddress; }
    set { _StreetAddress = value; }
}
public string MyCity
{
    get { return _City; }
    set { _City = value; }
}
public string MyState
{
    get { return _State; }
    set { _State = value; }
}
public string MyZip
{
    get { return _Zip; }
    set { _Zip = value; }
}
public bool IsOK
{
    get { return _IsOK; }
    set { _IsOK = value; }
}
}

```

用户单击“确定”或“取消”按钮时，`Click` 事件处理程序将创建包含该数据并引发 `OnButtonClick` 事件的 `MyControlEvents` 对象。两个处理程序的唯一区别是事件自变量的 `isOk` 属性。此属性使主机可以确定单击的按钮。“确认”按钮设置为 `true`，“取消”按钮设置为 `false`。以下代码演示两个按钮处理程序。

将以下代码添加到 `MyControl1` 类。

C#

```
private void btnOK_Click(object sender, System.EventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(true,
        txtName.Text,
        txtAddress.Text,
        txtCity.Text,
        txtState.Text,
        txtZip.Text);
    OnButtonClick(this, retvals);
}

private void btnCancel_Click(object sender, System.EventArgs e)
{
    MyControlEventArgs retvals = new MyControlEventArgs(false,
        txtName.Text,
        txtAddress.Text,
        txtCity.Text,
        txtState.Text,
        txtZip.Text);
    OnButtonClick(this, retvals);
}
```

## 赋予程序集强名称并生成程序集

要让 WPF 应用程序引用此程序集，它必须具有强名称。 若要创建强名称，请使用 Sn.exe 创建密钥文件并将其添加到项目中。

1. 打开 Visual Studio 命令提示。为此，请单击“开始”菜单，然后选择“所有程序”>“Microsoft Visual Studio 2010”>“Visual Studio Tools”>“Visual Studio 命令提示符”。这将启动包含自定义环境变量的控制台窗口。
2. 在命令提示符下，使用 cd 命令转到项目文件夹。
3. 通过运行以下命令生成名为 MyControls.snk 的密钥文件。

控制台

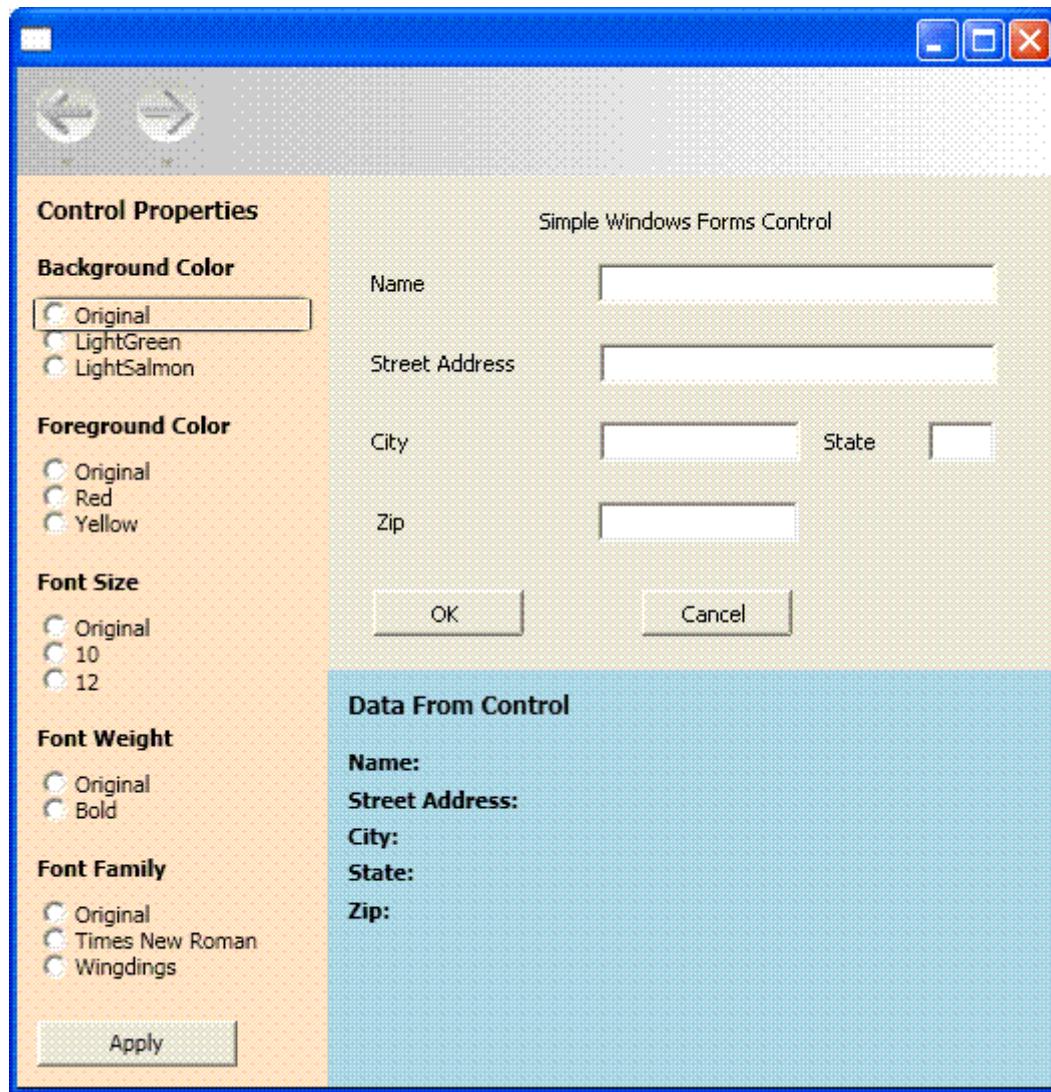
```
Sn.exe -k MyControls.snk
```

4. 若要在项目中包括密钥文件，请在解决方案资源管理器中右键单击项目名称，然后单击“属性”。在“项目设计器”中，单击“签名”选项卡，选中“签名程序集”复选框，然后浏览到密钥文件。
5. 生成解决方案。生成将产生一个名为 MyControls.dll 的 DLL。

# 实现 WPF 主机应用程序

WPF 宿主应用程序使用 [WindowsFormsHost](#) 控件来承载 `MyControl1`。该应用程序处理 `OnButtonClick` 事件以接收来自控件的数据。它还有一组选项按钮，使你能够在 WPF 应用程序中更改某些控件属性。下图显示已完成的应用程序。

下图显示了完整的应用程序，包括 WPF 应用程序中嵌入的控件：



## 创建项目

启动项目：

1. 打开 Visual Studio，然后选择“新建项目”。
2. 在“窗口”类别中，选择 WPF 应用程序模板。
3. 将新项目命名为 `WpfHost`。
4. 对于位置，指定包含 `MyControls` 项目的同一顶层文件夹。

## 5. 单击“确定”以创建该项目。

你还需要添加对包含 MyControl1 和其他程序集的 DLL 的引用。

1. 右键单击解决方案资源管理器中的项目名称，然后选择“添加引用”。
2. 单击“浏览”选项卡，然后浏览到包含 MyControls.dll 的文件夹。在本演练中，此文件夹位于 MyControls\bin\Debug。
3. 选择 MyControls.dll，然后单击“确定”。
4. 添加对 WindowsFormsIntegration 程序集的引用，该程序集名为 WindowsFormsIntegration.dll。

## 实现基本布局

宿主应用程序的用户界面 (UI) 在 MainWindow.xaml 中实现。此文件包含 Extensible Application Markup Language (XAML) 标记，用于定义布局，并承载 Windows 窗体控件。该应用程序分为三个区域：

- “控件属性”面板，其中包含一组选项按钮，你可以使用这些按钮修改所承载控件的各种属性。
- “来自控件的数据”面板，其中包含多个用于显示从所承载控件返回的数据的 **TextBlock** 元素。
- 所承载控件本身。

基本布局如下面的 XAML 中所示。此示例中省略了承载 MyControl1 所需的标记，但将在后面进行讨论。

将 MainWindow.xaml 中的 XAML 替换为以下内容。如果使用 Visual Basic，请将类更改为 `x:Class="MainWindow"`。

### XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WpfHost.MainWindow"
    xmlns:mcl="clr-namespace:MyControls;assembly=MyControls"
    Loaded="Init">
    <DockPanel>
        <DockPanel.Resources>
            <Style x:Key="inlineText" TargetType="{x:Type Inline}">
                <Setter Property="FontWeight" Value="Normal"/>
            </Style>
            <Style x:Key="titleText" TargetType="{x:Type TextBlock}">
                <Setter Property="DockPanel.Dock" Value="Top"/>
            </Style>
        </DockPanel.Resources>
    </DockPanel>
</Window>
```

```

        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Margin" Value="10,5,10,0"/>
    </Style>
</DockPanel.Resources>

<StackPanel Orientation="Vertical"
            DockPanel.Dock="Left"
            Background="Bisque"
            Width="250">

    <TextBlock Margin="10,10,10,10"
               FontWeight="Bold"
               FontSize="12">Control Properties</TextBlock>
    <TextBlock Style="{StaticResource titleText}">Background
Color</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnOriginalBackColor"
                    IsChecked="True"
                    Click="BackColorChanged">Original</RadioButton>
        <RadioButton Name="rdbtnBackGreen"
                    Click="BackColorChanged">LightGreen</RadioButton>
        <RadioButton Name="rdbtnBackSalmon"
                    Click="BackColorChanged">LightSalmon</RadioButton>
    </StackPanel>

    <TextBlock Style="{StaticResource titleText}">Foreground
Color</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnOriginalForeColor"
                    IsChecked="True"
                    Click="ForeColorChanged">Original</RadioButton>
        <RadioButton Name="rdbtnForeRed"
                    Click="ForeColorChanged">Red</RadioButton>
        <RadioButton Name="rdbtnForeYellow"
                    Click="ForeColorChanged">Yellow</RadioButton>
    </StackPanel>

    <TextBlock Style="{StaticResource titleText}">Font Family</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnOriginalFamily"
                    IsChecked="True"
                    Click="FontChanged">Original</RadioButton>
        <RadioButton Name="rdbtnTimes"
                    Click="FontChanged">Times New Roman</RadioButton>
        <RadioButton Name="rdbtnWingdings"
                    Click="FontChanged">Wingdings</RadioButton>
    </StackPanel>

    <TextBlock Style="{StaticResource titleText}">Font Size</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnOriginalSize"
                    IsChecked="True"
                    Click="FontSizeChanged">Original</RadioButton>
        <RadioButton Name="rdbtnTen"
                    Click="FontSizeChanged">10</RadioButton>
    </StackPanel>

```

```

        <RadioButton Name="rdbtnTwelve"
                     Click="FontSizeChanged">12</RadioButton>
    </StackPanel>

    <TextBlock Style="{StaticResource titleText}">Font Style</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnNormalStyle"
                     IsChecked="True"
                     Click="StyleChanged">Original</RadioButton>
        <RadioButton Name="rdbtnItalic"
                     Click="StyleChanged">Italic</RadioButton>
    </StackPanel>

    <TextBlock Style="{StaticResource titleText}">Font Weight</TextBlock>
    <StackPanel Margin="10,10,10,10">
        <RadioButton Name="rdbtnOriginalWeight"
                     IsChecked="True"
                     Click="WeightChanged">
            Original
        </RadioButton>
        <RadioButton Name="rdbtnBold"
                     Click="WeightChanged">Bold</RadioButton>
    </StackPanel>
</StackPanel>

<WindowsFormsHost Name="wfh"
                  DockPanel.Dock="Top"
                  Height="300">
    <mcl:MyControl1 Name="mc"/>
</WindowsFormsHost>

<StackPanel Orientation="Vertical"
            Height="Auto"
            Background="LightBlue">
    <TextBlock Margin="10,10,10,10"
              FontWeight="Bold"
              FontSize="12">Data From Control</TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Name: <Span Name="txtName" Style="{StaticResource inlineText}" />
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Street Address: <Span Name="txtAddress" Style="{StaticResource
        inlineText}" />
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        City: <Span Name="txtCity" Style="{StaticResource inlineText}" />
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        State: <Span Name="txtState" Style="{StaticResource inlineText}" />
    </TextBlock>
    <TextBlock Style="{StaticResource titleText}">
        Zip: <Span Name="txtZip" Style="{StaticResource inlineText}" />
    </TextBlock>
</StackPanel>

```

```
</DockPanel>  
</Window>
```

第一个 `StackPanel` 元素包含几组 `RadioButton` 控件，使你能够修改所承载控件的各种默认属性。其后是承载 `MyControl1` 的 `WindowsFormsHost` 元素。最后一个 `StackPanel` 元素包含几个 `TextBlock` 元素，这些元素显示所承载空间返回的数据。元素的顺序以及 `Dock` 和 `Height` 属性设置将所承载控件嵌入到窗口中，没有间隙或失真。

## 承载控件

此前 XAML 的以下编辑版本侧重于承载 `MyControl1` 所需的元素。

XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
        x:Class="WpfHost.MainWindow"  
        xmlns:mcl="clr-namespace:MyControls;assembly=MyControls"  
        Loaded="Init">
```

XAML

```
<WindowsFormsHost Name="wfh"  
                  DockPanel.Dock="Top"  
                  Height="300">  
    <mcl:MyControl1 Name="mc"/>  
</WindowsFormsHost>
```

`xmlns` 命名空间映射属性创建对包含所承载控件的 `MyControls` 命名空间的引用。此映射使你能够将 XAML 中的 `MyControl1` 表示为 `<mcl:MyControl1>`。

XAML 中的两个元素处理承载：

- `WindowsFormsHost` 表示使你能够在 WPF 应用程序中承载 Windows 窗体控件的 `WindowsFormsHost` 元素。
- 代表 `MyControl1` 的 `mcl:MyControl1` 被添加到 `WindowsFormsHost` 元素的子集合中。因此，此 Windows 窗体控件呈现为 WPF 窗口的一部分，你可以从应用程序与控件进行通信。

## 实现代码隐藏文件

代码隐藏文件 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 包含实现上一节中讨论的 UI 功能的过程代码。主要任务有：

- 将事件处理程序附加到 `MyControl1` 的 `OnButtonClick` 事件中。
- 根据选项按钮集合的设置方式修改 `MyControl1` 的各种属性。
- 显示控件收集的数据。

## 初始化应用程序

初始化代码包含在窗口的 `Loaded` 事件的事件处理程序中，并将事件处理程序附加到控件的 `OnButtonClick` 事件。

在 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 中，将以下代码添加到 `MainWindow` 类。

C#

```
private Application app;
private Window myWindow;
FontWeight initFontWeight;
Double initFontSize;
FontStyle initFontStyle;
SolidColorBrush initBackBrush;
SolidColorBrush initForeBrush;
FontFamily initFontFamily;
bool UIIsReady = false;

private void Init(object sender, EventArgs e)
{
    app = System.Windows.Application.Current;
    myWindow = (Window)app.MainWindow;
    myWindow.SizeToContent = SizeToContent.WidthAndHeight;
    wfh.TabIndex = 10;
    initFontSize = wfh.FontSize;
    initFontWeight = wfh.FontWeight;
    initFontFamily = wfh.FontFamily;
    initFontStyle = wfh.FontStyle;
    initBackBrush = (SolidColorBrush)wfh.Background;
    initForeBrush = (SolidColorBrush)wfh.Foreground;
    (wfh.Child as MyControl1).OnButtonClick += new
    MyControl1.MyControlEventHandler(Pane1_OnButtonClick);
    UIIsReady = true;
}
```

由于前面讨论的 XAML 将 `MyControl1` 添加到 `WindowsFormsHost` 元素的子元素集合中，因此你可以强制转换 `WindowsFormsHost` 元素的 `Child` 以获取对 `MyControl1` 的引用。可以使用该引用将事件处理程序附加到 `OnButtonClick`。

除了提供对控件本身的引用之外，[WindowsFormsHost](#) 还公开了一些控件的属性，可以从应用程序中对其进行操作。 初始化代码将这些值分配给私有全局变量，以供稍后在应用程序中使用。

这样便可轻松访问 `MyControls` DLL 中的类型，将以下 `Imports` 或 `using` 语句添加到文件顶部。

```
C#
```

```
using MyControls;
```

## 处理 OnButtonClick 事件

当用户单击控件的任一按钮时，`MyControl1` 会引发 `OnButtonClick` 事件。

将以下代码添加到 `MainWindow` 类。

```
C#
```

```
//Handle button clicks on the Windows Form control
private void Panel1_OnButtonClick(object sender, MyControlEventArgs args)
{
    txtName.Inlines.Clear();
    txtAddress.Inlines.Clear();
    txtCity.Inlines.Clear();
    txtState.Inlines.Clear();
    txtZip.Inlines.Clear();

    if (args.IsOK)
    {
        txtName.Inlines.Add(" " + args.MyName );
        txtAddress.Inlines.Add(" " + args.MyStreetAddress );
        txtCity.Inlines.Add(" " + args.MyCity );
        txtState.Inlines.Add(" " + args.MyState );
        txtZip.Inlines.Add(" " + args.MyZip );
    }
}
```

文本框中的数据将打包到 `MyControlEventArgs` 对象。如果用户单击“确定”按钮，事件处理程序将提取该数据并在 `MyControl1` 下的面板中显示。

## 修改控件属性

[WindowsFormsHost](#) 元素公开了一些所承载控件的默认属性。因此，可以更改空间外观，以更贴合应用程序的样式。位于左侧面板中的选项按钮组使用户能够修改多个颜色

和字体属性。每组按钮都有一个用于 Click 事件的处理程序，该处理程序检测用户的选项按钮选择并更改控件上的相应属性。

将以下代码添加到 `MainWindow` 类。

C#

```
private void BackColorChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnBackGreen)
        wfh.Background = new SolidColorBrush(Colors.LightGreen);
    else if (sender == rdbtnBackSalmon)
        wfh.Background = new SolidColorBrush(Colors.LightSalmon);
    else if (UIIsReady == true)
        wfh.Background = initBackBrush;
}

private void ForeColorChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnForeRed)
        wfh.Foreground = new SolidColorBrush(Colors.Red);
    else if (sender == rdbtnForeYellow)
        wfh.Foreground = new SolidColorBrush(Colors.Yellow);
    else if (UIIsReady == true)
        wfh.Foreground = initForeBrush;
}

private void FontChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnTimes)
        wfh.FontFamily = new FontFamily("Times New Roman");
    else if (sender == rdbtnWingdings)
        wfh.FontFamily = new FontFamily("Wingdings");
    else if (UIIsReady == true)
        wfh.FontFamily = initFontFamily;
}

private void FontSizeChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnTen)
        wfh.FontSize = 10;
    else if (sender == rdbtnTwelve)
        wfh.FontSize = 12;
    else if (UIIsReady == true)
        wfh.FontSize = initFontSize;
}

private void StyleChanged(object sender, RoutedEventArgs e)
{
    if (sender == rdbtnItalic)
        wfh.FontStyle = FontStyles.Italic;
    else if (UIIsReady == true)
        wfh.FontStyle = initFontStyle;
}

private void WeightChanged(object sender, RoutedEventArgs e)
```

```
{  
    if (sender == rdbtnBold)  
        wfh.FontWeight = FontWeights.Bold;  
    else if (UIIsReady == true)  
        wfh.FontWeight = initFontWeight;  
}
```

生成并运行应用程序。在 Windows 窗体复合控件中添加一些文本，然后单击“确定”。文本将显示在标签中。单击不同的单选按钮查看在控件上的效果。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 WPF 中承载 Windows 窗体控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)

# 演练：在 WPF 中承载 ActiveX 控件

项目 • 2023/02/06

要改进与浏览器的交互，可以在基于 WPF 的应用程序中使用 Microsoft ActiveX 控件。本演练演示如何将 Microsoft Windows Media Player 承载为 WPF 页面上的控件。

本演练涉及以下任务：

- 创建项目。
- 创建 ActiveX 控件。
- 在 WPF 页面上承载 ActiveX 控件。

完成本演练后，你将了解如何在基于 WPF 的应用程序中使用 Microsoft ActiveX 控件。

## 先决条件

您需要满足以下条件才能完成本演练：

- Microsoft Windows Media Player 安装在安装了 Visual Studio 的计算机上。
- Visual Studio 2010。

## 创建项目

### 创建并设置项目

1. 创建名为 `HostingAxInWpf` 的 WPF 应用程序项目。
2. 将 Windows 窗体控件库项目添加到解决方案，并将项目命名为 `WmpAxLib`。
3. 在 `WmpAxLib` 项目中，添加对名为 `wmp.dll` 的 Windows Media Player 程序集的引用。
4. 打开“工具箱”。
5. 右键单击工具箱，然后单击“选择项”。
6. 单击“COM 组件”选项卡，选择“Windows Media Player”控件，然后单击“确定”。

Windows Media Player 控件将添加到“工具箱”中。

7. 在解决方案资源管理器中，右键单击“UserControl1”文件，然后单击“重命名”。
8. 根据语言将名称更改为 `WmpAxControl.vb` 或 `WmpAxControl.cs`。
9. 如果系统提示重命名所有引用，请单击“是”。

## 创建 ActiveX 控件

将控件添加到设计图面时，Visual Studio 会自动为 Microsoft ActiveX 控件生成 `AxHost` 包装类。以下过程创建名为 `AxInterop.WMPLib.dll` 的托管程序集。

### 创建 ActiveX 控件

1. 在 Windows 窗体设计器中打开 `WmpAxControl.vb` 或 `WmpAxControl.cs`。
2. 在“工具箱”中，将 Windows Media Player 控件添加到设计图面。
3. 在“属性”窗口中，将 Windows Media Player 控件的 `Dock` 属性的值设置为 `Fill`。
4. 生成 `WmpAxLib` 控件库项目。

## 在 WPF 页面上承载 ActiveX 控件

### 承载 ActiveX 控件

1. 在 `HostingAxInWpf` 项目中，添加对生成的 ActiveX 互操作性程序集的引用。  
此程序集名为 `AxInterop.WMPLib.dll`，并在你导入 Windows Media Player 控件时添加到 `WmpAxLib` 项目的 `Debug` 文件夹中。
2. 添加对名为 `WindowsFormsIntegration.dll` 的 `WindowsFormsIntegration` 程序集的引用。
3. 添加对名为 `System.Windows.Forms.dll` 的 Windows 窗体程序集的引用。
4. 在 WPF 设计器中打开 `MainWindow.xaml`。
5. 将 `Grid` 元素命名为 `grid1`。

```
XAML

<Grid Name="grid1">
</Grid>
```

6. 在设计视图或 XAML 视图中，选择 [Window](#) 元素。

7. 在“属性”窗口中，单击“事件”选项卡。

8. 双击“CellDoubleClick”事件[Loaded](#)。

9. 插入以下代码以处理 [Loaded](#) 事件。

此代码创建 [WindowsFormsHost](#) 控件的实例并将 [AxWindowsMediaPlayer](#) 控件的实例添加为其子项。

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Create the interop host control.
    System.Windows.Forms.Integration.WindowsFormsHost host =
        new System.Windows.Forms.Integration.WindowsFormsHost();

    // Create the ActiveX control.
    WmpAxLib.AxWindowsMediaPlayer axWmp = new
    WmpAxLib.AxWindowsMediaPlayer();

    // Assign the ActiveX control as the host control's child.
    host.Child = axWmp;

    // Add the interop host control to the Grid
    // control's collection of child controls.
    this.grid1.Children.Add(host);

    // Play a .wav file with the ActiveX control.
    axWmp.URL = @"C:\Windows\Media\tada.wav";
}
```

10. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)

# 如何：在混合应用程序中启用视觉对象样式

项目 • 2022/09/27

本主题介绍如何在基于 WPF 的应用程序中承载的 Windows 窗体控件上启用视觉样式。

如果应用程序调用了 [EnableVisualStyles](#) 方法，则大多数 Windows 窗体控件将自动使用视觉样式。有关详细信息，请参阅[使用视觉样式呈现控件](#)。

有关本主题所涉及任务的完整代码列表，请参阅[在混合应用程序中启用视觉样式示例](#)。

## 启用 Windows 窗体视觉样式

### 启用 Windows 窗体视觉样式

1. 创建名为 `HostingWfWithVisualStyles` 的 WPF 应用程序项目。
2. 在解决方案资源管理器中，添加对下列程序集的引用。
  - `WindowsFormsIntegration`
  - `System.Windows.Forms`
3. 在工具箱中，双击 `Grid` 图标以在设计图面上放置 `Grid` 元素。
4. 在“属性”窗口中，将 `Height` 和 `Width` 属性的值设置为“自动”。
5. 在设计视图或 XAML 视图中，选择 `Window`。
6. 在“属性”窗口中，单击“事件”选项卡。
7. 双击“CellDoubleClick”事件[Loaded](#)。
8. 在 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 中，插入以下代码来处理 `Loaded` 事件。

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Comment out the following line to disable visual
    // styles for the hosted Windows Forms control.
    System.Windows.Forms.Application.EnableVisualStyles();
```

```
// Create a WindowsFormsHost element to host  
// the Windows Forms control.  
System.Windows.Forms.Integration.WindowsFormsHost host =  
    new System.Windows.Forms.Integration.WindowsFormsHost();  
  
// Create a Windows Forms tab control.  
System.Windows.Forms.TabControl tc = new  
System.Windows.Forms.TabControl();  
tc.TabPages.Add("Tab1");  
tc.TabPages.Add("Tab2");  
  
// Assign the Windows Forms tab control as the hosted control.  
host.Child = tc;  
  
// Assign the host element to the parent Grid element.  
this.grid1.Children.Add(host);  
}
```

9. 按 F5 生成并运行应用程序。

Windows 窗体控件具有了视觉样式。

## 禁用 Windows 窗体视觉样式

若要禁用视觉样式，只需移除对 [EnableVisualStyles](#) 方法的调用。

### 禁用 Windows 窗体视觉样式

1. 在代码编辑器中打开 MainWindow.xaml.vb 或 MainWindow.xaml.cs。
2. 注释掉对 [EnableVisualStyles](#) 方法的调用。
3. 按 F5 生成并运行应用程序。

Windows 窗体控件具有了默认系统样式。

## 另请参阅

- [EnableVisualStyles](#)
- [System.Windows.Forms.VisualStyles](#)
- [WindowsFormsHost](#)
- [使用视觉样式呈现控件](#)
- [演练：在 WPF 中承载 Windows 窗体控件](#)

# 演练：在 WPF 中排列 Windows 窗体控件

项目 • 2022/09/27

本演练演示如何使用 WPF 布局功能在混合应用程序中排列 Windows 窗体控件。

本演练涉及以下任务：

- 创建项目。
- 使用默认布局设置。
- 根据内容调整大小。
- 使用绝对定位。
- 显式指定大小。
- 设置布局属性。
- 了解 Z 顺序限制。
- 停靠。
- 设置可见性。
- 承载不拉伸的控件。
- 缩放。
- 旋转。
- 设置填充和边距。
- 使用动态布局容器。

有关本演练中介绍的任务的完整代码列表，请参阅[有关在 WPF 中排列 Windows 窗体控件的示例](#)。

完成本演练后，可获得对基于 WPF 的应用程序中的 Windows 窗体布局功能的一定了解。

## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 创建项目

若要创建和设置项目，请执行以下步骤：

1. 创建名为 `WpfLayoutHostingWf` 的 WPF 应用程序项目。
2. 在解决方案资源管理器中，添加对下列程序集的引用：
  - WindowsFormsIntegration

- System.Windows.Forms
- System.Drawing

3. 双击 MainWindow.xaml，在 XAML 视图中将其打开。

4. 在 [Window](#) 元素中，添加以下 Windows 窗体命名空间映射。

```
XAML  
  
xmlns:wf="clr-  
namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. 在 [Grid](#) 元素中，将 [ShowGridLines](#) 属性设置为 `true`，并定义五行三列。

```
XAML  
  
<Grid ShowGridLines="true">  
  <Grid.RowDefinitions>  
    <RowDefinition/>  
    <RowDefinition/>  
    <RowDefinition/>  
    <RowDefinition/>  
    <RowDefinition/>  
  </Grid.RowDefinitions>  
  
  <Grid.ColumnDefinitions>  
    <ColumnDefinition/>  
    <ColumnDefinition/>  
    <ColumnDefinition/>  
  </Grid.ColumnDefinitions>
```

## 使用默认布局设置

默认情况下，[WindowsFormsHost](#) 元素处理托管的 Windows 窗体控件的布局。

若要使用默认布局设置，请执行以下步骤：

1. 将以下 XAML 复制到 [Grid](#) 元素中：

```
XAML  
  
<!-- Default layout. -->  
<Canvas Grid.Row="0" Grid.Column="0">  
  <WindowsFormsHost Background="Yellow">  
    <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>  
  </WindowsFormsHost>  
</Canvas>
```

2. 按 F5 生成并运行应用程序。 Windows 窗体 System.Windows.Forms.Button 控件随即出现在 Canvas 中。 托管控件根据其内容调整大小，WindowsFormsHost 元素根据托管控件调整大小。

## 按内容调整大小

WindowsFormsHost 元素确保按控件内容调整托管控件的大小，以正确显示其内容。

若要按内容调整大小，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

```
XAML

<!-- Sizing to content. -->
<Canvas Grid.Row="1" Grid.Column="0">
    <WindowsFormsHost Background="Orange">
        <wf:Button Text="Windows Forms control with more content"
FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>

<Canvas Grid.Row="2" Grid.Column="0">
    <WindowsFormsHost FontSize="24" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

2. 按 F5 生成并运行应用程序。 两个新按钮控件会调整大小以正确显示较长文本字符串和较大字号，WindowsFormsHost 元素会根据托管控件调整大小。

## 使用绝对定位

可以使用绝对定位将 WindowsFormsHost 元素置于用户界面 (UI) 中的任何位置。

若要使用绝对定位，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

```
XAML

<!-- Absolute positioning. -->
<Canvas Grid.Row="3" Grid.Column="0">
    <WindowsFormsHost Canvas.Top="20" Canvas.Left="20"
Background="Yellow">
        <wf:Button Text="Windows Forms control with absolute positioning"
FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

```
</WindowsFormsHost>  
</Canvas>
```

2. 按 F5 生成并运行应用程序。 WindowsFormsHost 元素置于网格单元格中距顶部 20 个像素、距左侧 20 个像素的位置。

## 显式指定大小

可以使用 Width 和 Height 属性指定 WindowsFormsHost 元素的大小。

若要显式指定大小，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

XAML

```
<!-- Explicit sizing. -->  
<Canvas Grid.Row="4" Grid.Column="0">  
    <WindowsFormsHost Width="50" Height="70" Background="Yellow">  
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>  
    </WindowsFormsHost>  
</Canvas>
```

2. 按 F5 生成并运行应用程序。 WindowsFormsHost 元素的大小设为宽 50 像素、高 70 像素，这比默认布局设置小。 将相应重新排列 Windows 窗体控件的内容。

## 设置布局属性

应始终使用 WindowsFormsHost 元素的属性在托管控件上设置与布局相关的属性。 直接对承载控件设置布局属性会产生意外结果。

在 XAML 中对托管控件设置与布局相关的属性将不起作用。

若要查看对托管控件设置属性的效果，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

XAML

```
<!-- Setting hosted control properties directly. -->  
<Canvas Grid.Row="0" Grid.Column="1">  
    <WindowsFormsHost Width="160" Height="50" Background="Yellow">  
        <wf:Button Name="button1" Click="button1_Click" Text="Click me"  
            FlatStyle="Flat" BackColor="Green"/>  
    </WindowsFormsHost>  
</Canvas>
```

2. 在“解决方案资源管理器”中，双击 MainWindow.xaml.vb 或 MainWindow.xaml.cs 以在代码编辑器中打开它。
3. 将以下代码复制到 `MainWindow` 类定义中：

```
C#
```

```
private void button1_Click(object sender, EventArgs e )  
{  
    System.Windows.Forms.Button b = sender as  
System.Windows.Forms.Button;  
  
    b.Top = 20;  
    b.Left = 20;  
}
```

4. 按 F5 生成并运行应用程序。
5. 单击“单击我”按钮。`button1_Click` 事件处理程序设置托管控件的 `Top` 和 `Left` 属性。这会导致在 `WindowsFormsHost` 元素中重新定位托管控件。宿主保持相同的屏幕区域，但承载控件被剪裁。相反，托管控件应始终填充 `WindowsFormsHost` 元素。

## 了解 Z 顺序限制

可见的 `WindowsFormsHost` 元素始终绘制在其他 WPF 元素的顶部，且不受 Z 顺序的影响。若要查看此 Z 顺序行为，请执行以下操作：

1. 将以下 XAML 复制到 `Grid` 元素中：

```
XAML
```

```
<!-- Z-order demonstration. -->  
<Canvas Grid.Row="1" Grid.Column="1">  
    <WindowsFormsHost Canvas.Top="20" Canvas.Left="20"  
    Background="Yellow">  
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>  
    </WindowsFormsHost>  
    <Label Content="A WPF label" FontSize="24"/>  
</Canvas>
```

2. 按 F5 生成并运行应用程序。`WindowsFormsHost` 元素绘于标签元素上方。

## 停靠

[WindowsFormsHost](#) 元素支持 WPF 停靠。设置 [Dock](#) 附加属性以停靠 [DockPanel](#) 元素中的托管控件。

若要停靠托管控件，请执行以下步骤：

1. 将以下 XAML 复制到 [Grid](#) 元素中：

#### XAML

```
<!-- Docking a WindowsFormsHost element. -->
<DockPanel LastChildFill="false" Grid.Row="2" Grid.Column="1">
    <WindowsFormsHost DockPanel.Dock="Right" Canvas.Top="20"
    Canvas.Left="20" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
</DockPanel>
```

2. 按 F5 生成并运行应用程序。[WindowsFormsHost](#) 元素停靠在 [DockPanel](#) 元素的右侧。

## 设置可见性

通过在 [WindowsFormsHost](#) 元素上设置 [Visibility](#) 属性，可以使 Windows 窗体控件不可见或使其折叠。当控件不可见时，控件不会显示，但会占据布局空间。当控件处于折叠状态时，控件不会显示，也不会占据布局空间。

若要设置托管控件的可见性，请执行以下步骤：

1. 将以下 XAML 复制到 [Grid](#) 元素中：

#### XAML

```
<!-- Setting Visibility to hidden and collapsed. -->
<StackPanel Grid.Row="3" Grid.Column="1">
    <Button Name="button2" Click="button2_Click" Content="Click to make
    invisible" Background="OrangeRed"/>
    <WindowsFormsHost Name="host1" Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>
    <Button Name="button3" Click="button3_Click" Content="Click to
    collapse" Background="OrangeRed"/>
</StackPanel>
```

2. 在 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 中，将以下代码复制到类定义：

#### C#

```
private void button2_Click(object sender, EventArgs e)
{
    this.host1.Visibility = Visibility.Hidden;
}

private void button3_Click(object sender, EventArgs e)
{
    this.host1.Visibility = Visibility.Collapsed;
}
```

3. 按 F5 生成并运行应用程序。

4. 单击“单击以隐藏”按钮使 [WindowsFormsHost](#) 元素不可见。

5. 单击“单击以折叠”按钮使 [WindowsFormsHost](#) 元素在布局中完全隐藏。当 Windows 窗体控件处于折叠状态时，周围的元素会重新排列以占据其空间。

## 承载不拉伸的控件

一些 Windows 窗体控件具有固定大小，不会拉伸以填充布局中的可用空间。例如，[MonthCalendar](#) 控件在固定的空间中显示月份。

若要托管不拉伸的控件，请执行以下步骤：

1. 将以下 XAML 复制到 [Grid](#) 元素中：

XAML

```
<!-- Hosting a control that does not stretch. -->
<!-- The MonthCalendar has a discrete size. -->
<StackPanel Grid.Row="4" Grid.Column="1">
    <Label Content="A WPF element" Background="OrangeRed"/>
    <WindowsFormsHost Background="Yellow">
        <wf:MonthCalendar/>
    </WindowsFormsHost>
    <Label Content="Another WPF element" Background="OrangeRed"/>
</StackPanel>
```

2. 按 F5 生成并运行应用程序。[WindowsFormsHost](#) 元素处于网格行正中，但不会拉伸以填充可用空间。如果窗口足够大，可能会看到托管 [MonthCalendar](#) 控件显示两个或更多个月份，但这些月份会在一行中居中显示。WPF 布局引擎使不能通过调整大小来填充可用空间的元素居中显示。

## 扩展

与 WPF 元素不同，大多数 Windows 窗体控件不以连续方式缩放。 若要提供自定义缩放，请重写 [WindowsFormsHost.ScaleChild](#) 方法。

若要使用默认行为缩放托管控件，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

```
XAML

<!-- Scaling transformation. -->
<StackPanel Grid.Row="0" Grid.Column="2">

    <StackPanel.RenderTransform>
        <ScaleTransform CenterX="0" CenterY="0" ScaleX="0.5" ScaleY="0.5" />
    </StackPanel.RenderTransform>

    <Label Content="A WPF UIElement" Background="OrangeRed"/>

    <WindowsFormsHost Background="Yellow">
        <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
    </WindowsFormsHost>

    <Label Content="Another WPF UIElement" Background="OrangeRed"/>

</StackPanel>
```

2. 按 F5 生成并运行应用程序。 承载控件及其周围元素按 0.5 的比例进行缩放。 但是，承载控件的字体不缩放。

## 旋转

与 WPF 元素不同，Windows 窗体控件不支持旋转。 应用旋转转换时，[WindowsFormsHost](#) 元素不与其他 WPF 元素一起旋转。 180 度以外的任何旋转值都会引发 [LayoutError](#) 事件。

若要查看混合应用程序中的旋转效果，请执行以下步骤：

1. 将以下 XAML 复制到 Grid 元素中：

```
XAML

<!-- Rotation transformation. -->
<StackPanel Grid.Row="1" Grid.Column="2">

    <StackPanel.RenderTransform>
        <RotateTransform CenterX="200" CenterY="50" Angle="180" />
    </StackPanel.RenderTransform>
```

```
<Label Content="A WPF element" Background="OrangeRed"/>

<WindowsFormsHost Background="Yellow">
    <wf:Button Text="Windows Forms control" FlatStyle="Flat"/>
</WindowsFormsHost>

<Label Content="Another WPF element" Background="OrangeRed"/>

</StackPanel>
```

- 按 F5 生成并运行应用程序。承载控件不旋转，但是它周围的元素旋转 180 度。可能必须调整窗口大小才能看到这些元素。

## 设置填充和边距

WPF 布局中的填充和边距类似于 Windows 窗体中的填充和边距。只需在 `WindowsFormsHost` 元素上设置 `Padding` 和 `Margin` 属性即可。

若要为托管控件设置填充和边距，请执行以下步骤：

- 将以下 XAML 复制到 `Grid` 元素中：

XAML

```
<!-- Padding. -->
<Canvas Grid.Row="2" Grid.Column="2">
    <WindowsFormsHost Padding="0, 20, 0, 0" Background="Yellow">
        <wf:Button Text="Windows Forms control with padding"
FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

XAML

```
<!-- Margin. -->
<Canvas Grid.Row="3" Grid.Column="2">
    <WindowsFormsHost Margin="20, 20, 0, 0" Background="Yellow">
        <wf:Button Text="Windows Forms control with margin"
FlatStyle="Flat"/>
    </WindowsFormsHost>
</Canvas>
```

- 按 F5 生成并运行应用程序。填充和边距设置应用于托管的 Windows 窗体控件的方式与 Windows 窗体中的应用方式相同。

## 使用动态布局容器

Windows 窗体提供两个动态布局容器，即 `FlowLayoutPanel` 和 `TableLayoutPanel`。还可以在 WPF 布局中使用这些容器。

若要使用动态布局容器，请执行以下步骤：

1. 将以下 XAML 复制到 `Grid` 元素中：

XAML

```
<!-- Flow layout. -->
<DockPanel Grid.Row="4" Grid.Column="2">
    <WindowsFormsHost Name="flowLayoutHost" Background="Yellow">
        <wf:FlowLayoutPanel/>
    </WindowsFormsHost>
</DockPanel>
```

2. 在 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 中，将以下代码复制到类定义：

C#

```
private void InitializeFlowLayoutPanel()
{
    System.Windows.Forms.FlowLayoutPanel flp =
        this.flowLayoutPanelHost.Child as
    System.Windows.Forms.FlowLayoutPanel;

    flp.WrapContents = true;

    const int numButtons = 6;

    for (int i = 0; i < numButtons; i++)
    {
        System.Windows.Forms.Button b = new
    System.Windows.Forms.Button();
        b.Text = "Button";
        b.BackColor = System.Drawing.Color.AliceBlue;
        b.FlatStyle = System.Windows.Forms.FlatStyle.Flat;

        flp.Controls.Add(b);
    }
}
```

3. 将调用添加到构造函数中的 `InitializeFlowLayoutPanel` 方法：

C#

```
public MainWindow()
{
    InitializeComponent();
```

```
    this.InitializeFlowLayoutPanel();  
}
```

4. 按 F5 生成并运行应用程序。 WindowsFormsHost 元素填充 DockPanel，而 FlowLayoutPanel 按默认的 FlowDirection 排列其子控件。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [WindowsFormsHost 元素的布局注意事项](#)
- [有关在 WPF 中排列 Windows 窗体控件的示例 ↴](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)

# 演练：在混合应用程序中绑定到数据

项目 • 2022/09/27

无论使用的是 Windows 窗体还是 WPF，都必须将数据源绑定到控件，以便向用户提供对基础数据的访问权限。本演练显示如何在同时包括 Windows 窗体和 WPF 控件的混合应用程序中使用数据绑定。

本演练涉及以下任务：

- 创建项目。
- 定义数据模板。
- 指定窗体布局。
- 指定数据绑定。
- 使用互操作功能显示数据。
- 向项目添加数据源。
- 绑定到数据源。

有关本演练所示任务的完整代码列表，请参阅[混合应用程序中的数据绑定示例](#)。

完成本演练后，你将对混合应用程序中的数据绑定功能有所了解。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio。
- 访问在 Microsoft SQL Server 上运行的 Northwind 示例数据库。

## 创建项目

### 创建并设置项目

1. 创建名为 `WPFWithWFAndDatabinding` 的 WPF 应用程序项目。
2. 在解决方案资源管理器中，添加对下列程序集的引用。
  - WindowsFormsIntegration

- System.Windows.Forms

3. 在 WPF 设计器中打开 MainWindow.xaml。

4. 在 [Window](#) 元素中，添加以下 Windows 窗体命名空间映射。

XAML

```
xmlns:wf="clr-  
namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

5. 通过分配 [Name](#) 属性将默认 [Grid](#) 元素命名为 `mainGrid`。

XAML

```
<Grid x:Name="mainGrid">
```

## 定义数据模板

客户的主列表显示在 [ListBox](#) 控件中。 以下代码示例定义了一个名为 `ListItemsTemplate` 的 [DataTemplate](#) 对象，该对象控制 [ListBox](#) 控件的可视化树。 将此 [DataTemplate](#) 分配给 [ListBox](#) 控件的 [ItemTemplate](#) 属性。

## 定义数据模板

- 将以下 XAML 复制到 [Grid](#) 元素的声明中。

XAML

```
<Grid.Resources>  
    <DataTemplate x:Key="ListItemsTemplate">  
        <TextBlock Text="{Binding Path=ContactName}" />  
    </DataTemplate>  
</Grid.Resources>
```

## 指定窗体布局

窗体的布局由一个三行三列的网格来定义。[Label](#) 控件旨在标识 Customers 表中的每一列。

## 设置网格布局

- 将以下 XAML 复制到 Grid 元素的声明中。

```
XAML

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

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

## 设置 Label 控件

- 将以下 XAML 复制到 Grid 元素的声明中。

```
XAML

<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="1">
    <Label Margin="20,38,5,2">First Name:</Label>
    <Label Margin="20,0,5,2">Company Name:</Label>
    <Label Margin="20,0,5,2">Phone:</Label>
    <Label Margin="20,0,5,2">Address:</Label>
    <Label Margin="20,0,5,2">City:</Label>
    <Label Margin="20,0,5,2">Region:</Label>
    <Label Margin="20,0,5,2">Postal Code:</Label>
</StackPanel>
```

## 指定数据绑定

客户的主列表显示在 [ListBox](#) 控件中。附加的 [ListItemsTemplate](#) 将 [TextBlock](#) 控件绑定到数据库中的 [ContactName](#) 字段。

每个客户记录的详细信息都显示在多个 [TextBox](#) 控件中。

## 指定数据绑定

- 将以下 XAML 复制到 Grid 元素的声明中。

[Binding](#) 类将 [TextBox](#) 控件绑定到数据库中的相应字段。

```
XAML
```

```

<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="0">
    <Label Margin="20,5,5,0">List of Customers:</Label>
    <ListBox x:Name="listBox1" Height="200" Width="200"
    HorizontalAlignment="Left"
        ItemTemplate="{StaticResource ListItemsTemplate}"
    IsSynchronizedWithCurrentItem="True" Margin="20,5,5,5"/>
</StackPanel>

<StackPanel Orientation="Vertical" Grid.Row="0" Grid.Column="2">
    <TextBox Margin="5,38,5,2" Width="200" Text="{Binding
    Path=ContactName}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding
    Path=CompanyName}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=Phone}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=Address}"/>
    <TextBox Margin="5,0,5,2" Width="200" Text="{Binding Path=City}"/>
    <TextBox Margin="5,0,5,2" Width="30" HorizontalAlignment="Left"
    Text="{Binding Path=Region}"/>
    <TextBox Margin="5,0,5,2" Width="50" HorizontalAlignment="Left"
    Text="{Binding Path=PostalCode}"/>
</StackPanel>

```

## 使用互操作功能显示数据

与所选客户对应的订单显示在名为 `dataGridView1` 的 `System.Windows.Forms.DataGridView` 控件中。 `dataGridView1` 控件将绑定到代码隐藏文件中的数据源。`WindowsFormsHost` 控件是此 Windows 窗体控件的父级。

## 在 DataGridView 控件中显示数据

- 将以下 XAML 复制到 `Grid` 元素的声明中。

XAML

```

<WindowsFormsHost Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3"
Margin="20,5,5,5" Height="300">
    <wf:DataGridView x:Name="dataGridView1"/>
</WindowsFormsHost>

```

## 向项目添加数据源

借助 Visual Studio，你可轻松地将数据源添加到项目中。此过程将向项目添加强类型数据集。还添加了其他支持类，例如每个所选表适用的表达配器。

## 添加数据源

1. 从“数据”菜单中，选择“添加新的数据源”。
2. 在“数据资源配置向导”中，使用数据集创建与 Northwind 数据库的连接。有关详细信息，请参阅 [How to: Connect to Data in a Database](#)。
3. 在“数据资源配置向导”提示你时，请将连接字符串另存为 `NorthwindConnectionString`。
4. 当系统提示选择数据库对象时，选择 `Customers` 和 `Orders` 表，然后将生成的数据集命名为 `NorthwindDataSet`。

## 绑定到数据源

`System.Windows.Forms.BindingSource` 组件为应用程序的数据源提供统一的接口。绑定到数据源是在代码隐藏文件中实现的。

## 绑定到数据源

1. 打开名为 `MainWindow.xaml.vb` 或 `MainWindow.xaml.cs` 的代码隐藏文件。
2. 将以下代码复制到 `MainWindow` 类定义中。

此代码声明连接到数据库的 `BindingSource` 组件和关联的帮助程序类。

C#

```
private System.Windows.Forms.BindingSource nwBindingSource;
private NorthwindDataSet nwDataSet;
private NorthwindDataSetTableAdapters.CustomersTableAdapter
customersTableAdapter =
    new NorthwindDataSetTableAdapters.CustomersTableAdapter();
private NorthwindDataSetTableAdapters.OrdersTableAdapter
ordersTableAdapter =
    new NorthwindDataSetTableAdapters.OrdersTableAdapter();
```

3. 将以下代码复制到构造函数中。

此代码创建并初始化 `BindingSource` 组件。

C#

```
public MainWindow()
{
    InitializeComponent();
```

```

// Create a DataSet for the Customers data.
this.nwDataSet = new NorthwindDataSet();
this.nwDataSet.DataSetName = "nwDataSet";

// Create a BindingSource for the Customers data.
this.nwBindingSource = new System.Windows.Forms.BindingSource();
this.nwBindingSource.DataMember = "Customers";
this.nwBindingSource.DataSource = this.nwDataSet;
}

```

4. 打开 MainWindow.xaml。
5. 在设计视图或 XAML 视图中，选择 **Window** 元素。
6. 在“属性”窗口中，单击“事件”选项卡。
7. 双击“CellDoubleClick”事件**Loaded**。
8. 将下面的代码复制到 **Loaded** 事件处理程序。

此代码将分配 **BindingSource** 组件作为数据上下文并填充 **Customers** 和 **Orders** 适配器对象。

C#

```

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Fill the Customers table adapter with data.
    this.customersTableAdapter.ClearBeforeFill = true;
    this.customersTableAdapter.Fill(this.nwDataSet.Customers);

    // Fill the Orders table adapter with data.
    this.ordersTableAdapter.Fill(this.nwDataSet.Orders);

    // Assign the BindingSource to
    // the data context of the main grid.
    this.mainGrid.DataContext = this.nwBindingSource;

    // Assign the BindingSource to
    // the data source of the list box.
    this.listBox1.ItemsSource = this.nwBindingSource;

    // Because this is a master/details form, the DataGridView
    // requires the foreign key relating the tables.
    this.dataGridView1.DataSource = this.nwBindingSource;
    this.dataGridView1.DataMember = "FK_Orders_Customers";

    // Handle the currency management aspect of the data models.
    // Attach an event handler to detect when the current item
    // changes via the WPF ListBox. This event handler synchronizes
    // the list collection with the BindingSource.
}

```

```
//  
  
BindingListCollectionView cv = CollectionViewSource.GetDefaultView(  
    this.nwBindingSource) as BindingListCollectionView;  
  
cv.CurrentChanged += new EventHandler(WPF_CurrentChanged);  
}
```

9. 将以下代码复制到 `MainWindow` 类定义中。

此方法处理 `CurrentChanged` 事件并更新数据绑定的当前项。

C#

```
// This event handler updates the current item  
// of the data binding.  
void WPF_CurrentChanged(object sender, EventArgs e)  
{  
    BindingListCollectionView cv = sender as BindingListCollectionView;  
    this.nwBindingSource.Position = cv.CurrentPosition;  
}
```

10. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [混合应用程序中的数据绑定示例](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)

# 演练：在 Windows 窗体中承载三维 WPF 复合控件

项目 • 2022/09/27

本演练演示如何使用 [ElementHost](#) 控件创建 WPF 复合控件并在 Windows 窗体控件和窗体中承载该控件。

在本演练中，你将实现一个包含两个子控件的 WPF [UserControl](#)。[UserControl](#) 显示三维 (3D) 圆锥。使用 WPF 呈现三维对象比使用 Windows 窗体要容易得多。因此，承载 WPF [UserControl](#) 类似在 Windows 窗体中创建 3D 图形是有意义的。

本演练涉及以下任务：

- 创建 WPF [UserControl](#)。
- 创建 Windows 窗体宿主项目。
- 承载 WPF [UserControl](#)。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio 2017

## 创建 UserControl

1. 创建一个名为 `HostingWpfUserControlInWf` 的 WPF 用户控件库项目。
2. 在 WPF 设计器中打开 `UserControl1.xaml`。
3. 将生成的代码替换为以下代码：

```
XAML

<UserControl x:Class="HostingWpfUserControlInWf.UserControl1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>

    <Grid>
        <!-- Place a Label control at the top of the view. -->
        <Label>
```

```
        HorizontalAlignment="Center"
        TextBlock.TextAlignment="Center"
        FontSize="20"
        Foreground="Red"
        Content="Model: Cone"/>

    <!-- Viewport3D is the rendering surface. -->
    <Viewport3D Name="myViewport" >

        <!-- Add a camera. -->
        <Viewport3D.Camera>
            <PerspectiveCamera
                FarPlaneDistance="20"
                LookDirection="0,0,1"
                UpDirection="0,1,0"
                NearPlaneDistance="1"
                Position="0,0,-3"
                FieldOfView="45" />
        </Viewport3D.Camera>

        <!-- Add models. -->
        <Viewport3D.Children>

            <ModelVisual3D>
                <ModelVisual3D.Content>

                    <Model3DGroup >
                        <Model3DGroup.Children>

                            <!-- Lights, MeshGeometry3D and DiffuseMaterial
                                objects are added to the ModelVisual3D. -->
                            <DirectionalLight Color="#FFFFFF"
                                Direction="3,-4,5" />

                            <!-- Define a red cone. -->
                            <GeometryModel3D>

                                <GeometryModel3D.Geometry>
                                    <MeshGeometry3D
                                        Positions="0.293893 -0.5 0.404509  0.475528 -0.5 0.154509  0 0.5 0
0.475528 -0.5 0.154509  0 0.5 0  0 0.5 0  0.475528 -0.5 0.154509
0.475528 -0.5 -0.154509  0 0.5 0  0.475528 -0.5 -0.154509  0 0.5 0  0
0.5 0  0.475528 -0.5 -0.154509  0.293893 -0.5 -0.404509  0 0.5 0
0.293893 -0.5 -0.404509  0 0.5 0  0 0.5 0  0.293893 -0.5 -0.404509  0
-0.5 -0.5  0 0.5 0  0 -0.5 -0.5  0 0.5 0  0 0.5 0  0 -0.5 -0.5
-0.293893 -0.5 -0.404509  0 0.5 0  -0.293893 -0.5 -0.404509  0 0.5 0
0.5 0  -0.293893 -0.5 -0.404509  -0.475528 -0.5 -0.154509  0 0.5 0
-0.475528 -0.5 -0.154509  0 0.5 0  0 0.5 0  -0.475528 -0.5 -0.154509
-0.475528 -0.5 0.154509  0 0.5 0  -0.475528 -0.5 0.154509  0 0.5 0
0.5 0  -0.475528 -0.5 0.154509  -0.293892 -0.5 0.404509  0 0.5 0
-0.293892 -0.5 0.404509  0 0.5 0  0 0.5 0  -0.293892 -0.5 0.404509
-0.5 0.5  0 0.5 0  0 -0.5 0.5  0 0.5 0  0 0.5 0  0 -0.5 0.5
-0.5 0.404509  0 0.5 0  0.293893 -0.5 0.404509  0 0.5 0  0 0.5 0 "
                                Normals="0.7236065,0.4472139,0.5257313
0.2763934,0.4472138,0.8506507  0.5308242,0.4294462,0.7306172
```

```

0.2763934,0.4472138,0.8506507 0,0.4294458,0.9030925
0.5308242,0.4294462,0.7306172 0.2763934,0.4472138,0.8506507
-0.2763934,0.4472138,0.8506507 0,0.4294458,0.9030925
-0.2763934,0.4472138,0.8506507 -0.5308242,0.4294462,0.7306172
0,0.4294458,0.9030925 -0.2763934,0.4472138,0.8506507
-0.7236065,0.4472139,0.5257313 -0.5308242,0.4294462,0.7306172
-0.7236065,0.4472139,0.5257313 -0.858892,0.429446,0.279071
-0.5308242,0.4294462,0.7306172 -0.7236065,0.4472139,0.5257313
-0.8944269,0.4472139,0 -0.858892,0.429446,0.279071
-0.8944269,0.4472139,0 -0.858892,0.429446,-0.279071
-0.858892,0.429446,0.279071 -0.8944269,0.4472139,0
-0.7236065,0.4472139,-0.5257313 -0.858892,0.429446,-0.279071
-0.7236065,0.4472139,-0.5257313 -0.5308242,0.4294462,-0.7306172
-0.858892,0.429446,-0.279071 -0.7236065,0.4472139,-0.5257313
-0.2763934,0.4472138,-0.8506507 -0.5308242,0.4294462,-0.7306172
-0.2763934,0.4472138,-0.8506507 0,0.4294458,-0.9030925
-0.5308242,0.4294462,-0.7306172 -0.2763934,0.4472138,-0.8506507
0.2763934,0.4472138,-0.8506507 0,0.4294458,-0.9030925
0.2763934,0.4472138,-0.8506507 0.5308249,0.4294459,-0.7306169
0,0.4294458,-0.9030925 0.2763934,0.4472138,-0.8506507
0.7236068,0.4472141,-0.5257306 0.5308249,0.4294459,-0.7306169
0.7236068,0.4472141,-0.5257306 0.8588922,0.4294461,-0.27907
0.5308249,0.4294459,-0.7306169 0.7236068,0.4472141,-0.5257306
0.8944269,0.4472139,0 0.8588922,0.4294461,-0.27907
0.8944269,0.4472139,0 0.858892,0.429446,0.279071
0.8588922,0.4294461,-0.27907 0.8944269,0.4472139,0
0.7236065,0.4472139,0.5257313 0.858892,0.429446,0.279071
0.7236065,0.4472139,0.5257313 0.5308242,0.4294462,0.7306172
0.858892,0.429446,0.279071 " TriangleIndices="0 1 2
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 " />
    </GeometryModel3D.Geometry>

    <GeometryModel3D.Material>
        <DiffuseMaterial>
            <DiffuseMaterial.Brush>
                <SolidColorBrush
                    Color="Red"
                    Opacity="1.0"/>
            </DiffuseMaterial.Brush>
        </DiffuseMaterial>
    </GeometryModel3D.Material>

    </GeometryModel3D>

    </Model3DGroup.Children>
</Model3DGroup>

</ModelVisual3D.Content>

</ModelVisual3D>

</Viewport3D.Children>

```

```
</Viewport3D>
</Grid>

</UserControl>
```

此代码定义一个包含两个子控件的 [System.Windows.Controls.UserControl](#)。第一个子控件是 [System.Windows.Controls.Label](#) 控件；第二个子控件是用于显示 3D 圆锥的 [Viewport3D](#) 控件。

## 创建宿主项目

1. 将名为 [WpfUserControlHost](#) 的 Windows 窗体应用 (.NET Framework) 项目添加到解决方案中。
2. 在“解决方案资源管理器”中，添加对名为 WindowsFormsIntegration.dll 的 WindowsFormsIntegration 程序集的引用。
3. 添加对以下 WPF 程序集的引用：
  - PresentationCore
  - PresentationFramework
  - WindowsBase
4. 添加对 [HostingWpfUserControlInWf](#) 项目的引用。
5. 在“解决方案资源管理器”中，将 [WpfUserControlHost](#) 项目设置为启动项目。

## 承载 UserControl

1. 在 Windows 窗体设计器中，打开 Form1。
2. 在“属性”窗口中，单击“事件”，然后双击 [Load](#) 事件以创建事件处理程序。

新生成的 [Form1\\_Load](#) 事件处理程序将打开代码编辑器。

3. 将 Form1.cs 中的代码替换为以下代码。

[Form1\\_Load](#) 事件处理程序创建 [UserControl1](#) 实例并将其添加到 [ElementHost](#) 控件的子控件集合中。该 [ElementHost](#) 控件将被添加到窗体的子控件集合中。

C#

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

using System.Windows.Forms.Integration;

namespace WpfUserControlHost
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            // Create the ElementHost control for hosting the
            // WPF UserControl.
            ElementHost host = new ElementHost();
            host.Dock = DockStyle.Fill;

            // Create the WPF UserControl.
            HostingWpfUserControlInWf.UserControl1 uc =
                new HostingWpfUserControlInWf.UserControl1();

            // Assign the WPF UserControl to the ElementHost control's
            // Child property.
            host.Child = uc;

            // Add the ElementHost control to the form's
            // collection of child controls.
            this.Controls.Add(host);
        }
    }
}

```

4. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)

- 在 Windows 窗体中承载 WPF 复合控件示例 ↴

# 演练：在 Windows 窗体中承载 WPF 复合控件

项目 • 2023/02/06

WPF 并非从头开始重写它。一种常见情况是希望在 Windows 窗体应用程序内嵌入使用 WPF 实现的一个或多个控件。有关自定义 WPF 控件的详细信息，请参阅[控件自定义](#)。

本讲解将分步介绍承载 WPF 复合控件的应用程序，以在 Windows 窗体应用程序中执行数据输入。复合控件打包在一个 DLL 中。此常规步骤可扩展到更复杂的应用程序和控件。本讲解旨在实现与[讲解：在 WPF 中承载 Windows 窗体复合控件](#)中几乎完全相同的外观和功能。主要区别在于承载方案是相反的。

本演练分为两个部分。第一部分简要介绍了 WPF 复合控件的实现。第二部分详细讨论了如何在 Windows 窗体应用程序中承载复合控件、接收来自控件的事件以及访问控件的某些属性。

本演练涉及以下任务：

- 实现 WPF 复合控件。
- 实现 Windows 窗体主机应用程序。

有关本讲解所涉及任务的完整代码列表，请参阅[在 Windows 窗体中承载 WPF 复合控件示例](#)。

## 先决条件

若要完成本演练，必须具有 Visual Studio。

## 实现 WPF 复合控件

此示例中使用的 WPF 复合控件是一种简单地数据输入窗体，用于获取用户的姓名和地址。当用户单击两个按钮的其中一个以指示任务已完成时，该控件会引发将该信息返回给主机的自定义事件。下图显示呈现的控件。

下图显示了 WPF 复合控件：



## 创建项目

启动项目：

1. 启动 Visual Studio，然后打开“新建项目”对话框。
2. 在“Visual C# 和 Windows”类别中，选择“WPF 用户控件库”模板。
3. 将新项目命名为 `MyControls`。
4. 对于位置，指定可以方便命名的顶层文件夹，如 `WindowsFormsHostingWpfControl`。随后，将主机应用程序放在此文件夹中。
5. 单击“确定”以创建该项目。默认项目包含一个名为 `UserControl1` 的控件。
6. 在解决方案资源管理器中，将 `UserControl1` 重命名为 `MyControl1`。

项目应具有对以下系统 DLL 的引用。如果默认未包含其中任何 DLL，请将它们添加到项目中。

- `PresentationCore`
- `PresentationFramework`
- 系统
- `WindowsBase`

## 创建用户界面

UI 由五个 `TextBox` 元素组成。每个 `TextBox` 元素具有关联的 `TextBlock` 元素用作标签。底部有两个 `Button` 元素，“确定”和“取消”。当用户单击任一按钮时，该控件会引发将信息返回给主机的自定义事件。

## 基本布局

Grid 元素中包含各种 UI 元素。 可以使用 Grid 以排列复合控件的内容，方法与在 HTML 中使用 Table 元素大致相同。 WPF 还具有 Table 元素，但 Grid 更轻量级，更适合于简单的布局任务。

以下 XAML 演示基本布局。此 XAML 通过指定 Grid 元素中的列号和行号来定义控件的整体结构。

在 MyControl1.xaml 中，将现有 XAML 替换为以下 XAML。

XAML

```
<Grid xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      x:Class="MyControls.MyControl1"
      Background="#DCDCDC"
      Width="375"
      Height="250"
      Name="rootElement"
      Loaded="Init">

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

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

```

XAML

```
</Grid>
```

## 向 Grid 添加 TextBlock 和 TextBox 元素

通过将元素的 RowProperty 和 ColumnProperty 特性设置到适当的行号和列号，可以在网格中放置 UI 元素。请记住行号和列号是从零开始的。通过设置元素 ColumnSpanProperty 属性，可以让其跨越多个列。有关 Grid 元素的更多信息，请参阅 [创建网格元素](#)。

以下 XAML 显示复合控件的 `TextBox` 和 `TextBlock` 元素及其 `RowProperty` 和 `ColumnProperty` 特性，设置这些属性是为了在网格中正确放置元素。

在 MyControl1.xaml 中，在 `Grid` 元素内添加以下 XAML。

#### XAML

```
<TextBlock Grid.Column="0"
    Grid.Row="0"
    Grid.ColumnSpan="4"
    Margin="10,5,10,0"
    HorizontalAlignment="Center"
    Style="{StaticResource titleText}">Simple WPF Control</TextBlock>

<TextBlock Grid.Column="0"
    Grid.Row="1"
    Style="{StaticResource inlineText}"
    Name="nameLabel">Name</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="1"
    Grid.ColumnSpan="3"
    Name="txtName"/>

<TextBlock Grid.Column="0"
    Grid.Row="2"
    Style="{StaticResource inlineText}"
    Name="addressLabel">Street Address</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="2"
    Grid.ColumnSpan="3"
    Name="txtAddress"/>

<TextBlock Grid.Column="0"
    Grid.Row="3"
    Style="{StaticResource inlineText}"
    Name="cityLabel">City</TextBlock>
<TextBox Grid.Column="1"
    Grid.Row="3"
    Width="100"
    Name="txtCity"/>

<TextBlock Grid.Column="2"
    Grid.Row="3"
    Style="{StaticResource inlineText}"
    Name="stateLabel">State</TextBlock>
<TextBox Grid.Column="3"
    Grid.Row="3"
    Width="50"
    Name="txtState"/>

<TextBlock Grid.Column="0"
    Grid.Row="4"
    Style="{StaticResource inlineText}"
```

```
Name="zipLabel">Zip</TextBlock>
<TextBox Grid.Column="1"
        Grid.Row="4"
        Width="100"
        Name="txtZip"/>
```

## 设置 UI 元素的样式

数据输入窗体上的许多元素外观相似，这意味着它们对于其若干属性具有相同的设置。前一个 XAML 会使用 [Style](#) 元素为元素的类定义标准属性设置，而不是分别设置每个元素的特性。这种方法可以降低控件的复杂性，并使你能够通过单个样式特性更改多个元素的外观。

[Style](#) 元素包含着 [Grid](#) 元素的 [Resources](#) 属性中，因此可用于控件中的所有元素。如果命名了某样式，可以将 [Style](#) 元素集添加到样式的名称，将其应用到某个元素。未命名的样式将成为该元素的默认样式。有关样式的详细信息，请参阅[样式设置和模板化](#)。

以下 XAML 演示了复合控件的 [Style](#) 元素。若要查看样式如何应用于元素，请参阅前一个 XAML。例如，最后 [TextBlock](#) 元素具有 `inlineText` 样式，且最后 [TextBox](#) 元素使用默认样式。

在 MyControl1.xaml 中，在 [Grid](#) 开始元素后添加以下 XAML。

XAML

```
<Grid.Resources>
    <Style x:Key="inlineText" TargetType="{x:Type TextBlock}">
        <Setter Property="Margin" Value="10,5,10,0"/>
        <Setter Property="FontWeight" Value="Normal"/>
        <Setter Property="FontSize" Value="12"/>
    </Style>
    <Style x:Key="titleText" TargetType="{x:Type TextBlock}">
        <Setter Property="DockPanel.Dock" Value="Top"/>
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="FontSize" Value="14"/>
        <Setter Property="Margin" Value="10,5,10,0"/>
    </Style>
    <Style TargetType="{x:Type Button}">
        <Setter Property="Margin" Value="10,5,10,0"/>
        <Setter Property="Width" Value="60"/>
    </Style>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Margin" Value="10,5,10,0"/>
    </Style>
</Grid.Resources>
```

## 添加“确定”和“取消”按钮

复合控件上的最后两个元素是“确定”和“取消”`Button`元素，它们占用了`Grid`最后一行的前两列。这些元素使用一个公共事件处理程序，`ButtonClicked`，和在前一个 XAML 中定义的默认 `Button` 样式。

在 `MyControl1.xaml` 中，在最后 `TextBox` 元素后添加以下 XAML。复合控件的 XAML 部分现已完成。

#### XAML

```
<Button Grid.Row="5"
        Grid.Column="0"
        Name="btnOK"
        Click="ButtonClicked">OK</Button>
<Button Grid.Row="5"
        Grid.Column="1"
        Name="btnCancel"
        Click="ButtonClicked">Cancel</Button>
```

## 实现代码隐藏文件

代码隐藏文件 `MyControl1.xaml.cs` 实现了三个基本任务：

1. 处理用户单击其中一个按钮时发生的事情。
2. 从 `TextBox` 元素检索数据，并将其打包在自定义事件参数对象中。
3. 引发自定义 `OnButtonClick` 事件，该事件会通知主机用户已完成，并将数据传递回主机。

该控件还公开多个可用来更改外观的颜色和字体属性。与用于托管 Windows 窗体控件的 `WindowsFormsHost` 类不同，`ElementHost` 类只公开控件 `Background` 的属性。为了保持此代码示例与[演练：在 WPF 中托管 Windows 窗体复合控件](#)中所讨论示例之间的相似性，该控件会直接公开其余属性。

## 代码隐藏文件的基本结构

该代码隐藏文件由单个命名空间 `MyControls` 组成，它包含以下两个类：`MyControl1` 和 `MyControlEventArgs`。

#### C#

```
namespace MyControls
{
    public partial class MyControl1 : Grid
    {
        //...
```

```
    }
    public class MyControlEventArgs : EventArgs
    {
        //...
    }
}
```

第一个类 `MyControl1` 是一个分部类，其中包含实现 `MyControl1.xaml` 中定义的 UI 功能的代码。当解析 `MyControl1.xaml` 时，XAML 会转换为相同的分部类，且这两个分部类会合并形成编译的控件。出于此原因，代码隐藏文件中的类名必须与分配给 `MyControl1.xaml` 的类名相匹配，并且它必须继承自控件的根元素。第二个类 `MyControlEventArgs` 是一个事件参数类，用于将数据发送回主机。

打开 `MyControl1.xaml.cs`。更改现有的类声明，使其具有以下名称并继承自 `Grid`。

C#

```
public partial class MyControl1 : Grid
```

## 初始化控件

下面的代码实现几个基本任务：

- 声明一个专用事件 `OnButtonClick` 及其关联的委托 `MyControlEventHandler`。
- 创建几个存储用户数据的私有全局变量。此数据通过相应的属性公开。
- 为控件的 `Loaded` 事件实现处理程序 `Init`。此处理程序通过向全局变量分配 `MyControl1.xaml` 中定义的值来对它们进行初始化。为此，它使用分配给典型 `TextBlock` 元素 `nameLabel` 的 `Name`，以访问该元素的属性设置。

删除现有的构造函数并将以下代码添加到 `MyControl1` 类。

C#

```
public delegate void MyControlEventHandler(object sender, MyControlEventArgs args);
public event MyControlEventHandler OnButtonClick;
private FontWeight _fontWeight;
private double _fontSize;
private FontFamily _fontFamily;
private FontStyle _fontStyle;
private SolidColorBrush _foreground;
private SolidColorBrush _background;

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

```
//They all have the same style, so use nameLabel to set initial values.  
_fontWeight = nameLabel.FontWeight;  
_fontSize = nameLabel.FontSize;  
_fontFamily = nameLabel.FontFamily;  
_fontStyle = nameLabel.FontStyle;  
_foreground = (SolidColorBrush)nameLabel.Foreground;  
_background = (SolidColorBrush)rootElement.Background;  
}
```

## 处理按钮的单击事件

用户通过单击“确定”按钮或“取消”按钮指示数据输入任务已完成。这两个按钮使用相同的 `Click` 事件处理程序 `ButtonClicked`。这两个按钮有一个名称：`btnOK` 或 `btnCancel`，使处理程序能够通过检查 `sender` 参数的值来确定单击了哪个按钮。该处理程序执行以下任务：

- 创建一个 `MyControlEventArgs` 对象，用以包含来自 `TextBox` 元素的数据。
- 如果用户单击了“取消”按钮，则将 `MyControlEventArgs` 对象的 `IsOK` 属性设置为 `false`。
- 引发 `OnButtonClick` 事件，向主机指示用户已完成，并且传回所收集的数据。

在 `Init` 方法后，将以下代码添加到 `MyControl1` 类。

C#

```
private void ButtonClicked(object sender, RoutedEventArgs e)  
{  
    MyControlEventArgs retvals = new MyControlEventArgs(true,  
                                         txtName.Text,  
                                         txtAddress.Text,  
                                         txtCity.Text,  
                                         txtState.Text,  
                                         txtZip.Text);  
  
    if (sender == btnCancel)  
    {  
        retvals.IsOK = false;  
    }  
    if (OnButtonClick != null)  
        OnButtonClick(this, retvals);  
}
```

## 创建属性

类的其余部分只是公开对应于前面所述的全局变量的属性。当属性更改时，set 访问器会通过更改对应的元素属性并更新基础全局变量来修改控件的外观。

将以下代码添加到 MyControl1 类。

```
C#  
  
public FontWeight MyControl_FontWeight  
{  
    get { return _fontWeight; }  
    set  
    {  
        _fontWeight = value;  
        nameLabel.FontWeight = value;  
        addressLabel.FontWeight = value;  
        cityLabel.FontWeight = value;  
        stateLabel.FontWeight = value;  
        zipLabel.FontWeight = value;  
    }  
}  
public double MyControl_FontSize  
{  
    get { return _fontSize; }  
    set  
    {  
        _fontSize = value;  
        nameLabel.FontSize = value;  
        addressLabel.FontSize = value;  
        cityLabel.FontSize = value;  
        stateLabel.FontSize = value;  
        zipLabel.FontSize = value;  
    }  
}  
public FontStyle MyControl_FontStyle  
{  
    get { return _fontStyle; }  
    set  
    {  
        _fontStyle = value;  
        nameLabel.FontStyle = value;  
        addressLabel.FontStyle = value;  
        cityLabel.FontStyle = value;  
        stateLabel.FontStyle = value;  
        zipLabel.FontStyle = value;  
    }  
}  
public FontFamily MyControl_FontFamily  
{  
    get { return _fontFamily; }  
    set  
    {  
        _fontFamily = value;  
        nameLabel.FontFamily = value;
```

```

        addressLabel.FontFamily = value;
        cityLabel.FontFamily = value;
        stateLabel.FontFamily = value;
        zipLabel.FontFamily = value;
    }
}

public SolidColorBrush MyControl_Background
{
    get { return _background; }
    set
    {
        _background = value;
        rootElement.Background = value;
    }
}
public SolidColorBrush MyControl_Foreground
{
    get { return _foreground; }
    set
    {
        _foreground = value;
        nameLabel.Foreground = value;
        addressLabel.Foreground = value;
        cityLabel.Foreground = value;
        stateLabel.Foreground = value;
        zipLabel.Foreground = value;
    }
}

```

## 将数据发送回主机

文件中的最后一个组件是 `MyControlEventArgs` 类，该类用于将收集的数据发送回主机。

将以下代码添加到 `MyControls` 命名空间。该实现非常简单明了，因而不再进一步讨论。

C#

```

public class MyControlEventArgs : EventArgs
{
    private string _Name;
    private string _StreetAddress;
    private string _City;
    private string _State;
    private string _Zip;
    private bool _IsOK;

    public MyControlEventArgs(bool result,
                            string name,
                            string address,
                            string city,
                            string state,

```

```

        string zip)
{
    _IsOK = result;
    _Name = name;
    _StreetAddress = address;
    _City = city;
    _State = state;
    _Zip = zip;
}

public string MyName
{
    get { return _Name; }
    set { _Name = value; }
}
public string MyStreetAddress
{
    get { return _StreetAddress; }
    set { _StreetAddress = value; }
}
public string MyCity
{
    get { return _City; }
    set { _City = value; }
}
public string MyState
{
    get { return _State; }
    set { _State = value; }
}
public string MyZip
{
    get { return _Zip; }
    set { _Zip = value; }
}
public bool IsOK
{
    get { return _IsOK; }
    set { _IsOK = value; }
}
}

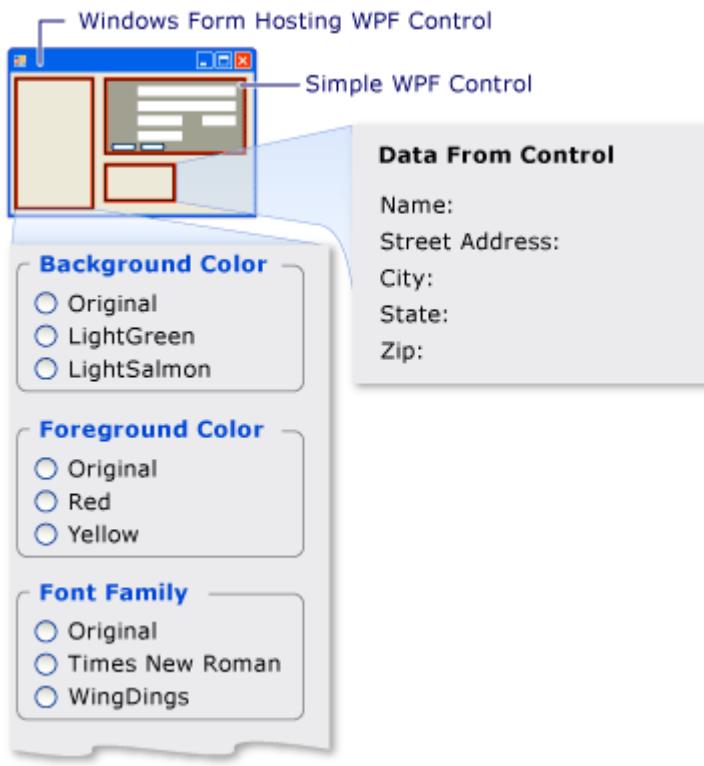
```

生成解决方案。生成将产生一个名为 MyControls.dll 的 DLL。

## 实现 Windows 窗体主机应用程序

Windows 窗体主机应用程序使用 [ElementHost](#) 对象来托管 WPF 复合控件。该应用程序处理 `OnButtonClick` 事件以接收来自复合控件的数据。该应用程序还具有一组选项按钮，可用于修改控件外观。下图显示应用程序。

下图显示托管在 Windows 窗体应用程序中的 WPF 复合控件



## 创建项目

启动项目：

1. 启动 Visual Studio，然后打开“新建项目”对话框。
2. 在“Visual C#”和“Windows”类别中，选择“Windows 窗体应用程序”模板。
3. 将新项目命名为 `WFHost`。
4. 对于位置，指定包含 MyControls 项目的同一顶层文件夹。
5. 单击“确定”以创建该项目。

还需要添加对 DLL 的引用，包含 `MyControl1` 和其他程序集。

1. 在解决方案资源管理器中右键单击项目名称，然后选择“添加引用”。
2. 单击“浏览”选项卡，然后浏览到包含 `MyControls.dll` 的文件夹。在本演练中，此文件夹位于 `MyControls\bin\Debug`。
3. 选择 `MyControls.dll`，然后单击“确定”。
4. 添加对下列程序集的引用。
  - `PresentationCore`
  - `PresentationFramework`

- System.Xaml
- WindowsBase
- WindowsFormsIntegration

## 实现应用程序的用户界面

Windows 窗体应用程序的 UI 包含若干个与 WPF 复合控件进行交互的控件。

1. 在 Windows 窗体设计器中打开 Form1。
2. 放大窗体以适应控件。
3. 在窗体右上角，添加 [System.Windows.Forms.Panel](#) 控件以保留 WPF 复合控件。
4. 向窗体添加以下 [System.Windows.Forms.GroupBox](#) 控件。

名称	文本
groupBox1	背景色
groupBox2	前景色
groupBox3	字号
groupBox4	字体系列
groupBox5	字形
groupBox6	字体粗细
groupBox7	来自控件的数据

5. 添加以下 [System.Windows.Forms.RadioButton](#) 控件到 [System.Windows.Forms.GroupBox](#) 控件。

GroupBox	名称	文本
groupBox1	radioBackgroundOriginal	原始
groupBox1	radioBackgroundLightGreen	LightGreen
groupBox1	radioBackgroundLightSalmon	LightSalmon
groupBox2	radioForegroundOriginal	原始
groupBox2	radioForegroundRed	Red

GroupBox	名称	文本
groupBox2	radioForegroundYellow	黄色
groupBox3	radioSizeOriginal	原始
groupBox3	radioSizeTen	10
groupBox3	radioSizeTwelve	12
groupBox4	radioFamilyOriginal	原始
groupBox4	radioFamilyTimes	Times New Roman
groupBox4	radioFamilyWingDings	WingDings
groupBox5	radioStyleOriginal	普通
groupBox5	radioStyleItalic	斜体
groupBox6	radioWeightOriginal	原始
groupBox6	radioWeightBold	加粗

6. 添加以下 [System.Windows.Forms.Label](#) 控件到最后的 [System.Windows.Forms.GroupBox](#)。这些控件显示 WPF 复合控件返回的数据。

GroupBox	名称	文本
groupBox7	lblName	姓名：
groupBox7	lblAddress	街道地址：
groupBox7	lblCity	市/县：
groupBox7	lblState	状态：
groupBox7	lblZip	邮政编码：

## 初始化窗体

通常在窗体的 [Load](#) 事件处理程序中实现托管代码。以下代码显示 [Load](#) 事件处理程序，这是 WPF 复合控件的 [Loaded](#) 事件的处理程序，以及稍后将使用的几个全局变量的声明。

在 Windows 窗体设计器中，双击窗体以创建 [Load](#) 事件处理程序。在 Form1.cs 顶部，添加以下 `using` 语句。

C#

```
using System.Windows;
using System.Windows.Forms.Integration;
using System.Windows.Media;
```

将现有 `Form1` 类的内容替换为以下代码。

C#

```
private ElementHost ctrlHost;
private MyControls.MyControl1 wpfAddressCtrl;
System.Windows.FontWeight initFontWeight;
double initFontSize;
System.Windows.FontStyle initFontStyle;
System.Windows.Media.SolidColorBrush initBackBrush;
System.Windows.Media.SolidColorBrush initForeBrush;
System.Windows.Media.FontFamily initFontFamily;

public Form1()
{
    InitializeComponent();
}

private void Form1_Load(object sender, EventArgs e)
{
    ctrlHost = new ElementHost();
    ctrlHost.Dock = DockStyle.Fill;
    panel1.Controls.Add(ctrlHost);
    wpfAddressCtrl = new MyControls.MyControl1();
    wpfAddressCtrl.InitializeComponent();
    ctrlHost.Child = wpfAddressCtrl;

    wpfAddressCtrl.OnButtonClick += 
        new MyControls.MyControl1.MyControlEventHandler(
            avAddressCtrl_OnButtonClick);
    wpfAddressCtrl.Loaded += new RoutedEventHandler(
        avAddressCtrl_Loaded);
}

void avAddressCtrl_Loaded(object sender, EventArgs e)
{
    initBackBrush = (SolidColorBrush)wpfAddressCtrl.MyControl_Background;
    initForeBrush = wpfAddressCtrl.MyControl_Foreground;
    initFontFamily = wpfAddressCtrl.MyControl_FontFamily;
    initFontSize = wpfAddressCtrl.MyControl_FontSize;
    initFontWeight = wpfAddressCtrl.MyControl_FontWeight;
    initFontStyle = wpfAddressCtrl.MyControl_FontStyle;
}
```

上述代码中的 `Form1_Load` 方法演示了托管 WPF 控件的常规步骤：

1. 创建新 `ElementHost` 对象。

- 将控件的 Dock 属性设置为 DockStyle.Fill。
- 将 ElementHost 控件添加到 Panel 控件的 Controls 集合。
- 创建 WPF 控件的实例。
- 通过将复合控件分配给 ElementHost 控件的 Child 属性在窗体上托管该复合控件。

Form1\_Load 方法中的剩余两行将处理程序附加到两个控件事件：

- 当用户单击“确定”或“取消”按钮时，OnButtonClick 是该复合控件触发的自定义事件。处理该事件可获取用户的响应并收集用户指定的任何数据。
- Loaded 是在完全加载 WPF 控件时引发的标准事件。此处使用该事件是因为本示例需要使用控件中的属性初始化几个全局变量。在窗体的 Load 事件时期，该控件不会完全加载，并且这些值仍设置为 null。需要等到该控件的 Loaded 事件发生之后，才能访问这些属性。

Loaded 事件处理程序显示在前面的代码中。下一部分将讨论 OnButtonClick 处理程序。

## 处理 OnButtonClick

OnButtonClick 事件会在用户单击“确定”或“取消”按钮时发生。

事件处理程序将检查事件参数的 IsOK 字段，以确定单击了哪个按钮。lblData 变量对应于前面讨论过的 Label 控件。如果用户单击“确定”按钮，则来自该控件的 TextBox 控件会分配给对应的 Label 控件。如果用户单击“取消”按钮，则将 Text 值设置为默认字符串。

将以下按钮单击事件处理程序代码添加到 Form1 类。

```
C#  
  
void avAddressCtrl_OnButtonClick(  
    object sender,  
    MyControls.MyControl1.MyControlEvents args)  
{  
    if (args.IsOK)  
    {  
        lblAddress.Text = "Street Address: " + args.MyStreetAddress;  
        lblCity.Text = "City: " + args.MyCity;  
        lblName.Text = "Name: " + args.MyName;  
        lblState.Text = "State: " + args.MyState;  
        lblZip.Text = "Zip: " + args.MyZip;  
    }  
    else  
    {  
        lblAddress.Text = "Street Address: ";  
        lblCity.Text = "City: ";  
    }  
}
```

```
        lblName.Text = "Name: ";
        lblState.Text = "State: ";
        lblZip.Text = "Zip: ";
    }
}
```

生成并运行应用程序。在 WPF 复合控件中添加一些文本，然后单击“确定”。文本将显示在标签中。此时，尚未添加代码来处理单选按钮。

## 修改控件的外观

窗体上的 [RadioButton](#) 控件使用户能够更改 WPF 复合控件的前景色和背景色以及一些字体属性。背景色由 [ElementHost](#) 对象公开。其余属性作为控件的自定义属性公开。

双击窗体上的每个 [RadioButton](#) 控件以创建 [CheckedChanged](#) 事件处理程序。将 [CheckedChanged](#) 事件处理程序替换为以下代码。

C#

```
private void radioBackgroundOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = initBackBrush;
}

private void radioBackgroundLightGreen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = new
SolidColorBrush(Colors.LightGreen);
}

private void radioBackgroundLightSalmon_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Background = new
SolidColorBrush(Colors.LightSalmon);
}

private void radioForegroundOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = initForeBrush;
}

private void radioForegroundRed_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new
System.Windows.Media.SolidColorBrush(Colors.Red);
}
```

```
private void radioForegroundYellow_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_Foreground = new
System.Windows.Media.SolidColorBrush(Colors.Yellow);
}

private void radioFamilyOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = initFontFamily;
}

private void radioFamilyTimes_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new
System.Windows.Media.FontFamily("Times New Roman");
}

private void radioFamilyWingDings_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontFamily = new
System.Windows.Media.FontFamily("WingDings");
}

private void radioSizeOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = initFontSize;
}

private void radioSizeTen_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 10;
}

private void radioSizeTwelve_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontSize = 12;
}

private void radioStyleOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = initFontStyle;
}

private void radioStyleItalic_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontStyle = System.Windows.FontStyles.Italic;
}

private void radioWeightOriginal_CheckedChanged(object sender, EventArgs e)
{
    wpfAddressCtrl.MyControl_FontWeight = initFontWeight;
}

private void radioWeightBold_CheckedChanged(object sender, EventArgs e)
```

```
{  
    wpfAddressCtrl.MyControl_FontWeight = FontWeights.Bold;  
}
```

生成并运行应用程序。 单击不同的单选按钮来查看在 WPF 复合控件上的效果。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 WPF 中托管 Windows 窗体复合控件](#)
- [演练：在 Windows 窗体中承载三维 WPF 复合控件](#)

# 演练：使用 ElementHost 控件映射属性

项目 • 2022/09/27

本演练演示如何使用 [PropertyMap](#) 属性将 Windows 窗体属性映射到托管 WPF 元素上的相应属性。

本演练涉及以下任务：

- 创建项目。
- 定义新的属性映射。
- 删除默认属性映射。
- 扩展默认属性映射。

完成后，你将能够将 Windows 窗体属性映射到托管元素上相应的 WPF 属性。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio 2017

## 创建项目

### 创建项目

1. 创建名为 `PropertyMappingWithElementHost` 的 Windows 窗体应用项目。
2. 在解决方案资源管理器中，添加对下列 WPF 程序集的引用。
  - PresentationCore
  - PresentationFramework
  - WindowsBase
  - WindowsFormsIntegration
3. 将以下代码复制到 `Form1` 代码文件的顶部。

C#

```
using System.Windows;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Forms.Integration;
```

4. 在 Windows 窗体设计器中打开 `Form1`。 双击窗体，为 `Load` 事件添加事件处理程序。
5. 返回到 Windows 窗体设计器，并为窗体的 `Resize` 事件添加事件处理程序。 有关详细信息，请参阅[如何：使用设计器创建事件处理程序](#)。
6. 在 `Form1` 类中声明 `ElementHost` 字段。

```
C#  
ElementHost elemHost = null;
```

## 定义新的属性映射

`ElementHost` 控件提供了若干默认的属性映射。在 `ElementHost` 控件的 `PropertyMap` 上调用 `Add` 方法来添加新的属性映射。

## 定义新的属性映射

1. 将以下代码复制到 `Form1` 类的定义中。

```
C#  
  
// The AddMarginMapping method adds a new property mapping  
// for the Margin property.  
private void AddMarginMapping()  
{  
    elemHost.PropertyMap.Add(  
        "Margin",  
        new PropertyTranslator(OnMarginChange));  
}  
  
// The OnMarginChange method implements the mapping  
// from the Windows Forms Margin property to the  
// Windows Presentation Foundation Margin property.  
//  
// The provided Padding value is used to construct  
// a Thickness value for the hosted element's Margin  
// property.  
private void OnMarginChange(object h, String propertyName, object  
value)
```

```
{  
    ElementHost host = h as ElementHost;  
    Padding p = (Padding)value;  
    System.Windows.Controls.Button wpfButton =  
        host.Child as System.Windows.Controls.Button;  
  
    Thickness t = new Thickness(p.Left, p.Top, p.Right, p.Bottom );  
  
    wpfButton.Margin = t;  
}
```

AddMarginMapping 方法将为 Margin 属性添加新映射。

OnMarginChange 方法将 Margin 属性转换为 WPF Margin 属性。

2. 将以下代码复制到 Form1 类的定义中。

```
C#  
  
// The AddRegionMapping method assigns a custom  
// mapping for the Region property.  
private void AddRegionMapping()  
{  
    elemHost.PropertyMap.Add(  
        "Region",  
        new PropertyTranslator(OnRegionChange));  
}  
  
// The OnRegionChange method assigns an EllipseGeometry to  
// the hosted element's Clip property.  
private void OnRegionChange(  
    object h,  
    String propertyName,  
    object value)  
{  
    ElementHost host = h as ElementHost;  
    System.Windows.Controls.Button wpfButton =  
        host.Child as System.Windows.Controls.Button;  
  
    wpfButton.Clip = new EllipseGeometry(new Rect(  
        0,  
        0,  
        wpfButton.ActualWidth,  
        wpfButton.ActualHeight));  
}  
  
// The Form1_Resize method handles the form's Resize event.  
// It calls the OnRegionChange method explicitly to  
// assign a new clipping geometry to the hosted element.  
private void Form1_Resize(object sender, EventArgs e)  
{  
    this.OnRegionChange(elemHost, "Region", null);  
}
```

`AddRegionMapping` 方法将为 `Region` 属性添加新映射。

`OnRegionChange` 方法将 `Region` 属性转换为 WPF `Clip` 属性。

`Form1_Resize` 方法处理窗体的 `Resize` 事件并调整剪切区域的大小以适应托管元素。

## 删除默认属性映射

在 `ElementHost` 控件的 `PropertyMap` 上调用 `Remove` 方法来删除默认属性映射。

### 删除默认属性映射

- 将以下代码复制到 `Form1` 类的定义中。

C#

```
// The RemoveCursorMapping method deletes the default
// mapping for the Cursor property.
private void RemoveCursorMapping()
{
    elemHost.PropertyMap.Remove("Cursor");
}
```

`RemoveCursorMapping` 方法删除 `Cursor` 属性的默认映射。

## 扩展默认属性映射

可以使用默认属性映射，并使用自己的映射对其进行扩展。

### 扩展默认属性映射

- 将以下代码复制到 `Form1` 类的定义中。

C#

```
// The ExtendBackColorMapping method adds a property
// translator if a mapping already exists.
private void ExtendBackColorMapping()
{
    if (elemHost.PropertyMap["BackColor"] != null)
    {
        elemHost.PropertyMap["BackColor"] +=
            new PropertyTranslator(OnBackColorChange);
```

```

    }

    // The OnBackColorChange method assigns a specific image
    // to the hosted element's Background property.
    private void OnBackColorChange(object h, String propertyName, object
        value)
    {
        ElementHost host = h as ElementHost;
        System.Windows.Controls.Button wpfButton =
            host.Child as System.Windows.Controls.Button;

        ImageBrush b = new ImageBrush(new BitmapImage(
            new Uri(@"file:///C:\WINDOWS\Santa Fe Stucco.bmp")));
        wpfButton.Background = b;
    }
}

```

`ExtendBackColorMapping` 方法将自定义属性转换器添加到现有的 `BackColor` 属性映射。

`OnBackColorChange` 方法将特定图像分配给托管控件的 `Background` 属性。在应用默认属性映射后会调用 `OnBackColorChange` 方法。

## 初始化属性映射

1. 将以下代码复制到 `Form1` 类的定义中。

```

C#

private void Form1_Load(object sender, EventArgs e)
{
    // Create the ElementHost control.
    elemHost = new ElementHost();
    elemHost.Dock = DockStyle.Fill;
    this.Controls.Add(elemHost);

    // Create a Windows Presentation Foundation Button element
    // and assign it as the ElementHost control's child.
    System.Windows.Controls.Button wpfButton = new
    System.Windows.Controls.Button();
    wpfButton.Content = "Windows Presentation Foundation Button";
    elemHost.Child = wpfButton;

    // Map the Margin property.
    this.AddMarginMapping();

    // Remove the mapping for the Cursor property.
    this.RemoveCursorMapping();

    // Add a mapping for the Region property.
}

```

```
this.AddRegionMapping();

// Add another mapping for the BackColor property.
this.ExtendBackColorMapping();

// Cause the OnMarginChange delegate to be called.
elemHost.Margin = new Padding(23, 23, 23, 23);

// Cause the OnRegionChange delegate to be called.
elemHost.Region = new Region();

// Cause the OnBackColorChange delegate to be called.
elemHost.BackColor = System.Drawing.Color.AliceBlue;
}
```

`Form1_Load` 方法处理 `Load` 事件并执行以下初始化。

- 创建 WPF `Button` 元素。
- 调用先前在演练中定义的方法来设置属性映射。
- 将初始值分配给映射的属性。

2. 按 F5 生成并运行应用程序。

## 另请参阅

- [ElementHost.PropertyMap](#)
- [WindowsFormsHost.PropertyMap](#)
- [WindowsFormsHost](#)
- [Windows 窗体和 WPF 属性映射](#)
- [在 Visual Studio 中设计 XAML](#)
- [演练：在 Windows 窗体中承载 WPF 复合控件](#)

# 演练：使用 WindowsFormsHost 元素映射属性

项目 · 2022/09/27

本演练演示如何使用 [PropertyMap 属性](#) 将 WPF 属性映射到托管 Windows 窗体控件上的相应属性。

本演练涉及以下任务：

- 创建项目。
- 定义应用程序布局。
- 定义新的属性映射。
- 删除默认属性映射。
- 替换默认属性映射。
- 扩展默认属性映射。

完成后，你将能够将 WPF 属性映射到托管 Windows 窗体控件上的相应属性。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio 2017

## 创建并设置项目

1. 创建一个名为 `PropertyMappingWithWfhSample` 的 WPF 应用项目。
2. 在解决方案资源管理器中，添加对 WindowsFormsIntegration 程序集的引用，该程序集名为 `WindowsFormsIntegration.dll`。
3. 在解决方案资源管理器中，添加对 `System.Drawing` 和 `System.Windows.Forms` 程序集的引用。

## 定义应用程序布局

基于 WPF 的应用程序使用 [WindowsFormsHost 元素](#)托管 Windows 窗体控件。

# 定义应用程序布局

1. 在 WPF 设计器中打开 Window1.xaml。
2. 用下面的代码替换现有代码。

XAML

```
<Window x:Class="PropertyMappingWithWfh.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="PropertyMappingWithWfh" Height="300" Width="300"
    Loaded="WindowLoaded">
    <DockPanel Name="panel1" LastChildFill="True">
        <WindowsFormsHost Name="wfHost" DockPanel.Dock="Left"
SizeChanged="Window1_SizeChanged" FontSize="20" />
    </DockPanel>
</Window>
```

3. 在代码编辑器中打开 Window1.xaml.cs。
4. 在该文件顶部导入以下命名空间。

C#

```
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
using System.Windows.Forms.Integration;
```

## 定义新的属性映射

WindowsFormsHost 元素提供了若干默认的属性映射。在 WindowsFormsHost 元素的 PropertyMap 上调用 Add 方法添加新的属性映射。

## 定义新的属性映射

- 将以下代码复制到 Window1 类的定义中。

C#

```
// The AddClipMapping method adds a custom
// mapping for the Clip property.
private void AddClipMapping()
{
    wfHost.PropertyMap.Add(
        "Clip",
```

```

        new PropertyTranslator(OnClipChange));
    }

    // The OnClipChange method assigns an elliptical clipping
    // region to the hosted control's Region property.
    private void OnClipChange(object h, String propertyName, object value)
    {
        WindowsFormsHost host = h as WindowsFormsHost;
        System.Windows.Forms.CheckBox cb = host.Child as
System.Windows.Forms.CheckBox;

        if (cb != null)
        {
            cb.Region = this.CreateClipRegion();
        }
    }

    // The Window1_SizeChanged method handles the window's
    // SizeChanged event. It calls the OnClipChange method explicitly
    // to assign a new clipping region to the hosted control.
    private void Window1_SizeChanged(object sender, SizeChangedEventArgs e)
    {
        this.OnClipChange(wfHost, "Clip", null);
    }

    // The CreateClipRegion method creates a Region from an
    // elliptical GraphicsPath.
    private Region CreateClipRegion()
    {
        GraphicsPath path = new GraphicsPath();

        path.StartFigure();

        path.AddEllipse(new System.Drawing.Rectangle(
            0,
            0,
            (int)wfHost.ActualWidth,
            (int)wfHost.ActualHeight ) );

        path.CloseFigure();

        return( new Region(path) );
    }
}

```

AddClipMapping 方法添加 Clip 属性的新映射。

OnClipChange 方法将 Clip 属性转换为 Windows FormsRegion 属性。

Window1\_SizeChanged 方法处理窗口的 SizeChanged 事件并调整剪切区域的大小以适应应用程序窗口。

# 删除默认属性映射

在 WindowsFormsHost 元素的 PropertyMap 上调用 Remove 方法删除默认属性映射。

## 删除默认属性映射

- 将以下代码复制到 Window1 类的定义中。

C#

```
// The RemoveCursorMapping method deletes the default
// mapping for the Cursor property.
private void RemoveCursorMapping()
{
    wfHost.PropertyMap.Remove("Cursor");
}
```

RemoveCursorMapping 方法删除 Cursor 属性的默认映射。

## 替换默认属性映射

通过删除默认映射并在 WindowsFormsHost 元素的 PropertyMap 上调用 Add 方法可以替换默认属性映射。

## 替换默认属性映射

- 将以下代码复制到 Window1 类的定义中。

C#

```
// The ReplaceFlowDirectionMapping method replaces the
// default mapping for the FlowDirection property.
private void ReplaceFlowDirectionMapping()
{
    wfHost.PropertyMap.Remove("FlowDirection");

    wfHost.PropertyMap.Add(
        "FlowDirection",
        new PropertyTranslator(OnFlowDirectionChange));
}

// The OnFlowDirectionChange method translates a
// Windows Presentation Foundation FlowDirection value
// to a Windows Forms RightToLeft value and assigns
// the result to the hosted control's RightToLeft property.
private void OnFlowDirectionChange(object h, String propertyName,
```

```

object value)
{
    WindowsFormsHost host = h as WindowsFormsHost;
    System.Windows.FlowDirection fd =
    (System.Windows.FlowDirection)value;
    System.Windows.Forms.CheckBox cb = host.Child as
    System.Windows.Forms.CheckBox;

    cb.RightToLeft = (fd == System.Windows.FlowDirection.RightToLeft) ?
    RightToLeft.Yes : RightToLeft.No;
}

// The cb_CheckedChanged method handles the hosted control's
// CheckedChanged event. If the Checked property is true,
// the flow direction is set to RightToLeft, otherwise it is
// set to LeftToRight.
private void cb_CheckedChanged(object sender, EventArgs e)
{
    System.Windows.Forms.CheckBox cb = sender as
    System.Windows.Forms.CheckBox;

    wfHost.FlowDirection = (cb.CheckState == CheckState.Checked) ?
        System.Windows.FlowDirection.RightToLeft :
        System.Windows.FlowDirection.LeftToRight;
}

```

`ReplaceFlowDirectionMapping` 方法替换 `FlowDirection` 属性的默认映射。

`OnFlowDirectionChange` 方法将 `FlowDirection` 属性转换为 Windows Forms`RightToLeft` 属性。

`cb_CheckedChanged` 方法处理 `CheckBox` 控件上的 `CheckedChanged` 事件。它根据 `CheckState` 属性的值分配 `FlowDirection` 属性

## 扩展默认属性映射

可以使用默认属性映射，并使用自己的映射对其进行扩展。

## 扩展默认属性映射

- 将以下代码复制到 `Window1` 类的定义中。

C#

```

// The ExtendBackgroundMapping method adds a property
// translator if a mapping already exists.
private void ExtendBackgroundMapping()

```

```

{
    if (wfHost.PropertyMap["Background"] != null)
    {
        wfHost.PropertyMap["Background"] += new
PropertyTranslator(OnBackgroundChange);
    }
}

// The OnBackgroundChange method assigns a specific image
// to the hosted control's BackgroundImage property.
private void OnBackgroundChange(object h, String propertyName, object
value)
{
    WindowsFormsHost host = h as WindowsFormsHost;
    System.Windows.Forms.CheckBox cb = host.Child as
System.Windows.Forms.CheckBox;
    ImageBrush b = value as ImageBrush;

    if (b != null)
    {
        cb.BackgroundImage = new
System.Drawing.Bitmap(@"C:\WINDOWS\Santa Fe Stucco.bmp");
    }
}

```

`ExtendBackgroundMapping` 方法将自定义属性转换器添加到现有的 `Background` 属性映射。

`OnBackgroundChange` 方法将特定图像分配给托管控件的 `BackgroundImage` 属性。  
应用默认属性映射后调用 `OnBackgroundChange` 方法。

## 初始化属性映射

通过调用 `Loaded` 事件处理程序中前面所述的方法来设置属性映射。

## 初始化属性映射

1. 将以下代码复制到 `Window1` 类的定义中。

C#

```

// The WindowLoaded method handles the Loaded event.
// It enables Windows Forms visual styles, creates
// a Windows Forms checkbox control, and assigns the
// control as the child of the WindowsFormsHost element.
// This method also modifies property mappings on the
// WindowsFormsHost element.
private void WindowLoaded(object sender, RoutedEventArgs e)
{

```

```

System.Windows.Forms.Application.EnableVisualStyles();

// Create a Windows Forms checkbox control and assign
// it as the WindowsFormsHost element's child.
System.Windows.Forms.CheckBox cb = new
System.Windows.Forms.CheckBox();
cb.Text = "Windows Forms checkbox";
cb.Dock = DockStyle.Fill;
cb.TextAlign = ContentAlignment.MiddleCenter;
cb.CheckedChanged += new EventHandler(cb_CheckedChanged);
wfHost.Child = cb;

// Replace the default mapping for the FlowDirection property.
this.ReplaceFlowDirectionMapping();

// Remove the mapping for the Cursor property.
this.RemoveCursorMapping();

// Add the mapping for the Clip property.
this.AddClipMapping();

// Add another mapping for the Background property.
this.ExtendBackgroundMapping();

// Cause the OnFlowDirectionChange delegate to be called.
wfHost.FlowDirection = System.Windows.FlowDirection.LeftToRight;

// Cause the OnClipChange delegate to be called.
wfHost.Clip = new RectangleGeometry();

// Cause the OnBackgroundChange delegate to be called.
wfHost.Background = new ImageBrush();
}

```

`WindowLoaded` 方法处理 `Loaded` 事件并执行以下初始化。

- 创建 Windows Forms`CheckBox` 控件。
- 调用先前在演练中定义的方法来设置属性映射。
- 将初始值分配给映射的属性。

2. 按 F5 生成并运行应用程序。单击复选框可查看 `FlowDirection` 映射的效果。单击复选框时，布局会反转其左右方向。

## 另请参阅

- [WindowsFormsHost.PropertyMap](#)
- [ElementHost.PropertyMap](#)
- [WindowsFormsHost](#)

- Windows 窗体和 WPF 属性映射
- 在 Visual Studio 中设计 XAML
- 演练：在 WPF 中承载 Windows 窗体控件

# 演练：本地化混合应用程序

项目 • 2022/09/27

本演练介绍如何本地化基于 Windows 窗体的混合应用程序中的 WPF 元素。

本演练涉及以下任务：

- 创建 Windows 窗体宿主项目。
- 添加可本地化的内容。
- 启用本地化。
- 分配资源标识符。
- 使用 LocBaml 工具生成附属程序集。

有关本演练所涉及任务的完整代码列表，请参阅[本地化混合应用程序示例](#)。

完成后，你将拥有一个本地化的混合应用程序。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio 2017

## 创建 Windows 窗体宿主项目

第一步是创建 Windows 窗体应用程序项目，并添加一个包含要本地化的内容的 WPF 元素。

## 创建宿主项目

1. 创建一个名为 `LocalizingWpfInWf` 的 WPF 应用项目。（“文件”>“新建”>“项目”>“Visual C#”或“Visual Basic”>“经典桌面”>“WPF 应用程序”）。
2. 将名为 `SimpleControl` 的 `WPFLocalizedUserControl` 元素添加到项目中。
3. 使用 `ElementHost` 控件在窗体上放置一个 `SimpleControl` 元素。有关详细信息，请参阅[演练：在 Windows 窗体中托管三维 WPF 复合控件](#)。

# 添加可本地化的内容

接下来，将添加 Windows 窗体标签控件，并将 WPF 元素的内容设置为可本地化的字符串。

## 添加可本地化的内容

1. 在解决方案资源管理器中双击 SimpleControl.xaml，以在 WPF 设计器中打开它。
2. 使用以下代码设置 [Button](#) 控件的内容。

XAML

```
<UserControl x:Class="LocalizingWpfInWf.SimpleControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >

    <Canvas>
        <Button Content="Hello"/>
    </Canvas>
</UserControl>
```

3. 在解决方案资源管理器中双击 Form1，以在 Windows 窗体设计器中打开它。
4. 打开工具箱，然后双击“标签”以将标签控件添加到窗体。 将其 [Text](#) 属性的值设置为 "Hello"。
5. 按 F5 生成并运行应用程序。

SimpleControl 元素和标签控件都显示文本“Hello”。

## 启用本地化

Windows 窗体设计器提供用于在附属程序集中启用本地化的设置。

## 启用本地化

1. 在解决方案资源管理器中双击 Form1.cs，以在 Windows 窗体设计器中打开它。
2. 在“属性”窗口中，将窗体的 Localizable 属性的值设置为 [true](#)。
3. 在“属性”窗口中，将“语言”属性的值设置为“西班牙语(西班牙)”。
4. 在 Windows 窗体设计器中，选择标签控件。

5. 在属性窗口中，将 `Text` 属性的值设置为 "Hola"。

一个名为 Form1.es-ES.resx 的新资源文件会添加到项目中。

6. 在解决方案资源管理器中，右键单击 Form1.cs，然后单击“查看代码”以在代码编辑器中打开它。

7. 在调用 `InitializeComponent` 之前，将以下代码复制到 `Form1` 构造函数中。

C#

```
public Form1()
{
    System.Threading.Thread.CurrentThread.CurrentCulture = new
    System.Globalization.CultureInfo("es-ES");

    InitializeComponent();
}
```

8. 在解决方案资源管理器中，右键单击 LocalizingWpfInWf，然后单击“卸载项目”。

项目名称将标记为“(不可用)”。

9. 右键单击 LocalizingWpfInWf，然后单击“编辑 LocalizingWpfInWf.csproj”。

项目文件将在代码编辑器中打开。

10. 将以下行复制到项目文件的第一个 `PropertyGroup` 中。

XML

```
<UICulture>en-US</UICulture>
```

11. 保存并关闭项目文件。

12. 在解决方案资源管理器中，右键单击 LocalizingWpfInWf，然后单击“重载项目”。

## 分配资源标识符

可以通过使用资源标识符将可本地化内容映射到资源程序集。 指定选项时 `updateuid`，MSBuild.exe 应用程序会自动分配资源标识符。

## 分配资源标识符

1. 从“开始”菜单，打开“Visual Studio 开发人员命令提示”。

2. 使用以下命令将资源标识符分配到可本地化内容。

```
控制台
```

```
msbuild -t:updateuid LocalizingWpfInWF.csproj
```

3. 在解决方案资源管理器中双击 SimpleControl.xaml，以在代码编辑器中打开它。 将看到 msbuild 命令已将 `Uid` 特性添加到所有元素。 这有助于通过分配资源标识符进行本地化。

```
XAML
```

```
<UserControl x:Uid="UserControl_1"
x:Class="LocalizingWpfInWF.SimpleControl"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>

<Canvas x:Uid="Canvas_1">
    <Button x:Uid="Button_1" Content="Hello"/>
</Canvas>
</UserControl>
```

4. 按 F6 以生成解决方案。

## 使用 LocBaml 生成附属程序集

已本地化的内容存储在仅资源附属程序集中。 使用命令行工具 LocBaml.exe 可为 WPF 内容生成本地化程序集。

### 生成附属程序集

1. 将 LocBaml.exe 复制到项目的 obj\Debug 文件夹中。 有关详细信息，请参阅[对应用程序进行本地化](#)。
2. 在“命令提示符”窗口中，使用以下命令将资源字符串提取到临时文件中。

```
控制台
```

```
LocBaml /parse LocalizingWpfInWF.g.en-US.resources /out:temp.csv
```

3. 使用 Visual Studio 或其他文本编辑器打开 temp.csv 文件。 将字符串 "Hello" 替换为其西班牙语翻译 "Hola"。

4. 保存 temp.csv 文件。

5. 使用以下命令生成本地化资源文件。

控制台

```
LocBaml /generate /trans:temp.csv LocalizingWpfInWf.g.en-US.resources  
/out:.. /cul:es-ES
```

将在 obj\Debug 文件夹中创建 LocalizingWpfInWf.g.es-ES.resources 文件。

6. 使用以下命令生成本地化附属程序集。

控制台

```
A1.exe /out:LocalizingWpfInWf.resources.dll /culture:es-ES  
/embed:LocalizingWpfInWf.Form1.es-ES.resources  
/embed:LocalizingWpfInWf.g.es-ES.resources
```

将在 obj\Debug 文件夹中创建 LocalizingWpfInWf.resources.dll 文件。

7. 将 LocalizingWpfInWf.resources.dll 文件复制到项目的 bin\Debug\es-ES 文件夹中。  
替换现有文件。

8. 运行 LocalizingWpfInWf.exe , 它位于项目的 bin\Debug 文件夹中。 不要重新生成  
应用程序 , 否则附属程序集将被覆盖。

应用程序显示本地化后的字符串 , 而不是英语字符串。

## 另请参阅

- [ElementHost](#)
- [WindowsFormsHost](#)
- [对应用程序进行本地化](#)
- [演练 : 本地化 Windows 窗体](#)
- [在 Visual Studio 中设计 XAML](#)

# WPF 和 Win32 互操作

项目 • 2022/09/27

本主题概述如何互操作 WPF 和 Win32 代码。 Windows Presentation Foundation (WPF) 提供了用于创建应用程序的丰富环境。但是，如果对 Win32 代码投入很大，重复使用其中部分代码可能更有效。

## WPF 和 Win32 互操作基础知识

WPF 和 Win32 代码间的互操作存在两种基本方法。

- 在 Win32 窗口中托管 WPF 内容。通过此方法，可在标准 Win32 窗口和应用程序的框架内使用 WPF 的高级图形功能。
- 在 WPF 内容中托管 Win32 窗口。通过此方法，可在其他 WPF 内容的上下文中使用现有的自定义 Win32 控件，并可跨边界传递数据。

本主题中对上述每种方法进行了概念性介绍。有关在 Win32 中托管 WPF 的更具有代码针对性的说明，请参阅[演练：在 Win32 中托管 WPF 内容](#)。有关在 WPF 中托管 Win32 的更具代码针对性的说明，请参阅[演练：在 WPF 中托管 Win32 控件](#)。

## WPF 互操作项目

虽然 WPF API 为托管代码，但大多数现有 Win32 程序是以非托管 C++ 编写而成。无法从真正的非托管程序调用 WPF API。但是，通过使用 Microsoft Visual C++ 编译器的 `/clr` 选项，可创建托管与非托管混合的程序，在此程序中可无缝混合托管和非托管 API 调用。

存在一个项目级问题，即，无法将 Extensible Application Markup Language (XAML) 文件编译到 C++ 项目中。可通过一些项目分离技术对此进行弥补。

- 创建一个包含所有 XAML 页面的 C# DLL 作为编译程序集，然后使 C++ 可执行文件包含此 DLL 作为引用。
- 为 WPF 内容创建一个 C# 可执行文件，然后使其引用包含 Win32 内容的 C++ DLL。
- 运行时使用 [Load](#) 加载任何 XAML，而不是编译 XAML。
- 切勿在代码中使用 WPF，从 [Application](#) 构建元素树。

请使用最适合你的方法。

## ① 备注

如果之前未使用过 C++/CLI，你可能会在互操作代码示例中看到诸如 `gcnew` 和 `nullptr` 之类的一些“新”关键字。这些关键字取代旧版双下划线语法 (`__gc`)，为 C++ 中的托管代码提供了更加自然的语法。若要详细了解 C++/CLI 托管功能，请参阅适用于运行时平台的组件扩展。

## WPF 如何使用 Hwnd

若要充分利用 WPF“HWND 互操作”，需要了解 WPF 如何使用 HWND。对于任何 HWND，无法将 WPF 呈现与 DirectX 呈现或 GDI / GDI+ 呈现混合。这具有许多影响。首先，若要混合这些绘制模型，必须创建互操作解决方案，并对选择使用的每个绘制模型使用互操作的指定段。此外，绘制行为会为互操作解决方案可实现的操作创建一个“空域”限制。有关“空域”概念的详细信息，请参见[技术区概述](#)主题。

屏幕上的所有 WPF 元素最终受 HWND 支持。创建 WPF [Window](#) 时，WPF 会创建一个顶级 HWND，并使用 [HwndSource](#) 将 [Window](#) 及其 WPF 内容放入 HWND 内。应用程序中的其余 WPF 内容共享单个 HWND。菜单、组合框下拉列表和其他弹出窗口例外。这些元素会创建自己的顶级窗口，因此 WPF 菜单可能超出所在窗口 HWND 的边缘。使用 [HwndHost](#) 将 HWND 放入 WPF 内时，WPF 会告知 Win32 如何相对于 WPF [Window](#) HWND 放置新的子 HWND。

与之相关的 HWND 概念是每个 HWND 内及其之间的透明度。[技术区概述](#)主题中对此也有相关介绍。

## 在 Microsoft Win32 窗口中承载 WPF 内容

在 Win32 窗口中托管 WPF 的关键在于 [HwndSource](#) 类。此类在 Win32 窗口中包装 WPF 内容，以便在单个应用程序中将 WPF 内容并入 WPF。

1. 将 WPF 内容（内容根元素）作为托管类实现。通常，该类继承自可以包含多个子元素和/或用作根元素的类之一，例如 [DockPanel](#) 或 [Page](#)。在后续步骤中，此类称为 WPF 内容类，此类的实例称为 WPF 内容对象。
2. 使用 C++/CLI 实现 Windows 应用程序。若要从现有的非托管 C++ 应用程序开始，通常可以更改项目设置将 `/clr` 编译器标记包括在内，以此允许应用程序调用托管代码（本主题未介绍支持 `/clr` 编译可能需要的内容的完整范围）。
3. 将线程处理模型设置为单线程单元 (STA)。WPF 使用此线程模型。
4. 处理窗口过程中的 WM\_CREATE 通知。

5. 在处理程序（或处理程序调用的函数）中，执行以下操作：
  - a. 创建一个新的 `HwndSource` 对象，将父窗口 `HWND` 用作其 `parent` 参数。
  - b. 创建 WPF 内容类的一个实例。
  - c. 向 `HwndSource` 的 `RootVisual` 属性分配对 WPF 内容对象的引用。
  - d. `HwndSource` 对象的 `Handle` 属性包含窗口句柄 (`HWND`)。要获取可在应用程序的非托管部分中使用的 `HWND`，需将 `Handle.ToPointer()` 强制装换为 `HWND`。
6. 实现一个托管类，该类包含一个用于保存对 WPF 内容对象的引用的静态字段。通过此类可从 Win32 代码获取对 WPF 内容对象的引用，更重要的是，该类可防止意外对 `HwndSource` 进行垃圾回收。
7. 通过将处理程序附加到一个或多个 WPF 内容对象事件，接收来自 WPF 内容对象的通知。
8. 通过使用存储在静态字段中的引用来设置属性、调用方法等，与 WPF 内容对象进行通信。

### ① 备注

如果生成一个单独的程序集然后对其进行引用，对于步骤 1，可使用内容类的默认分部类在 XAML 中完成部分或全部 WPF 内容类定义。虽然在将 XAML 编译到程序集的过程中，通常包含 `Application` 对象，但最后不会在互操作过程中使用此 `Application`，而是仅使用由应用程序引用的 XAML 文件的一个或多个根类并引用其分部类。该过程的其余部分基本与上述相似。

**演练：**在 Win32 中承载 WPF 内容主题中对这些每个步骤通过代码进行了说明。

## 在 WPF 中承载 Microsoft Win32 窗口

在其他 WPF 内容中托管 Win32 窗口的关键在于 `HwndHost` 类。该类在可添加到 WPF 元素树的 WPF 元素中包装窗口。`HwndHost` 也支持 API，使你可执行诸如处理托管窗口的消息之类的任务。基本过程：

1. 为 WPF 应用程序创建一个元素树（可通过代码或标记）。在元素树中找到一个合适的许可点，在元素树中可将 `HwndHost` 实现添加为子元素。剩余步骤中，此元素称为保留元素。
2. 从 `HwndHost` 派生，创建一个包含 Win32 内容的对象。

3. 在此主机类中，替代 [HwndHost](#) 方法 [BuildWindowCore](#)。 返回承载窗口的 [HWND](#)。 可能需要将实际控件包装为返回窗口的子窗口；在托管窗口中包装控件为 WPF 内容从控件接收通知提供了一种简单的方式。此方法有助于更正一些有关托管控件边界处消息处理的 Win32 问题。
4. 替代 [HwndHost](#) 方法 [DestroyWindowCore](#) 和 [WndProc](#)。这样做的目的是处理清除和删除对承载内容的引用，尤其是在已创建对非托管对象的引用的情况下。
5. 在代码隐藏文件中，创建控件承载类的一个实例，并使其成为保留元素的子元素。通常会使用事件处理程序（如 [Loaded](#)）或使用分部类构造函数。但也可通过运行时行为添加互操作内容。
6. 处理选择的窗口消息，例如控件通知。方法有两种。两种方法提供对消息流的相同访问权限，因此你的选择很大程度上取决于编程简便性。
  - 在 [HwndHost](#) 方法 [WndProc](#) 的替代中，实现所有消息（不仅仅是关闭消息）的消息处理。
  - 通过处理 [MessageHook](#) 事件，使托管 WPF 元素处理消息。对发送到承载窗口的主窗口过程的每个消息都会引发该事件。
  - 无法使用 [WndProc](#) 处理来自进程外窗口的消息。
7. 通过使用平台调用来调用非托管 [SendMessage](#) 函数，与承载窗口通信。

按照这些步骤，创建一个处理鼠标输入的应用程序。通过实现 [IKeyboardInputSink](#) 接口，可为托管窗口添加 Tab 键支持。

演练：在 [WPF 中承载 Win32 控件](#) 主题中对这些每个步骤通过代码进行了说明。

## WPF 内部的 Hwnd

可将 [HwndHost](#) 视为一个特殊控件。（从技术上讲，[HwndHost](#) 是 [FrameworkElement](#) 派生类，而不是 [Control](#) 派生类，但出于互操作的目的，它可以被视为控件。）[HwndHost](#) 抽象托管内容的基础 Win32 性质，使 WPF 的其余部分将托管内容视为另一个类似控件的对象，该对象应呈现和处理输入。虽然基于基础 [HWND](#) 可支持项的限制，在输出（绘图和图形）和输入（鼠标和键盘）方面存在重大差异，但 [HwndHost](#) 的行为方式通常与其他所有 WPF [FrameworkElement](#) 类似。

## 输出行为的显著差异

- [FrameworkElement](#)（即 [HwndHost](#) 基类）具有众多表示 UI 更改的属性。例如属性 [FrameworkElement.FlowDirection](#)，该属性会更改该元素（作为父级）内元素的布局。但是，这些属性大多数未映射到可能的 Win32 等效项（即使这类等效项可能

存在）。过多这些属性及其含义具有过高的绘制技术针对性，这使得映射并不可行。因此，在 `HwndHost` 上设置 `FlowDirection` 等属性毫无作用。

- `HwndHost` 无法旋转、缩放、倾斜或受 `Transform` 影响。
- `HwndHost` 不支持 `Opacity` 属性（`alpha` 值混合处理）。如果 `HwndHost` 内的内容执行包含 `alpha` 信息的 `System.Drawing` 操作，虽然这本身不是冲突，但 `HwndHost` 作为整体仅支持不透明度 = 1.0 (100%)。
- `HwndHost` 出现在同一顶级窗口中的其他 WPF 元素之上。但是，`ToolTip` 或 `ContextMenu` 生成的菜单是一个单独的顶级窗口，因此将对 `HwndHost` 采取正确的行为。
- `HwndHost` 不遵从其父级 `UIElement` 的剪切区域。如果试图将 `HwndHost` 类放入滚动区域或 `Canvas`，这可能是个问题。

## 输入行为的显著差异

- 通常而言，虽然输入设备作用域在 `HwndHost` 托管的 Win32 区域内，但是输入事件会直接转到 Win32。
- 尽管鼠标位于 `HwndHost` 上方，但是应用程序不会接收 WPF 鼠标事件，且 WPF 属性 `IsMouseOver` 的值为 `false`。
- 尽管键盘焦点位于 `HwndHost`，但是应用程序不会接收 WPF 键盘事件，且 WPF 属性 `IsKeyboardFocusWithin` 的值为 `false`。
- 焦点位于 `HwndHost` 内并转至 `HwndHost` 内的另一控件时，应用程序不会接收 WPF 事件 `GotFocus` 或 `LostFocus`。
- 相关触笔属性和事件相似，且触笔位于 `HwndHost` 上方时不会报告信息。

## Tab 键、助记键和加速键

通过 `IKeyboardInputSink` 和 `IKeyboardInputSite` 接口，可为 WPF 和 Win32 混合应用程序创建无缝键盘体验：

- Win32 和 WPF 组件之间的 Tab 键
- 焦点位于 Win32 组件内和 WPF 组件内时皆起作用的助记键和加速键。

虽然 `HwndHost` 和 `HwndSource` 类都提供 `IKeyboardInputSink` 的实现，但在更高级的方案中，它们可能无法处理你需要的所有输入消息。替换为适当方法，获取所需的键盘行为。

接口仅对 WPF 和 Win32 区域间的转换过程中发生的事件提供支持。在 Win32 区域内，Tab 键行为完全受 Tab 键的 Win32 实现逻辑（若有）控制。

## 另请参阅

- [HwndHost](#)
- [HwndSource](#)
- [System.Windows.Interop](#)
- [演练：在 WPF 中承载 Win32 控件](#)
- [演练：在 Win32 中承载 WPF 内容](#)

# 技术区概述

项目 • 2023/02/06

如果在应用程序中使用多种呈现技术（例如 WPF、Win32 或 DirectX），则这些呈现技术必须共享公共顶级窗口中的呈现区域。本主题介绍可能会对 WPF 互操作应用程序的呈现和输入造成影响的问题。

## 区域

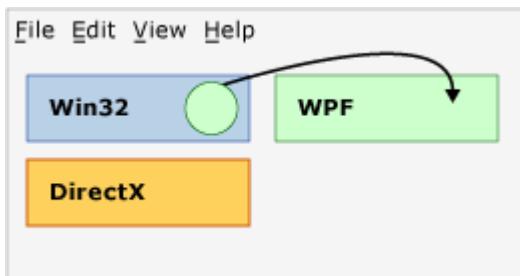
在顶级窗口内，可以这么想：每个包含互操作应用程序某个技术的 HWND 都具有自己的区域（也称为“空域”）。窗口内的每个像素都只属于一个特定 HWND，这构成了该 HWND 的区域。（严格地说，如果有多个 WPF HWND，便会有多个 WPF 区域，但为了便于讨论，可以假定只有一个区域。）区域暗含了这样一层含义：应用程序生存期内尝试在该像素之上呈现的所有层或其他窗口都必须是同一呈现级技术的一部分。尝试在 Win32 上方呈现 WPF 像素会导致意外结果，应尽量通过互操作 API 禁止这种尝试。

## 区域示例

下图显示一个混合使用 Win32、DirectX 和 WPF 的应用程序。每种技术都使用属于自己的且互不重叠的一组像素，因此不存在区域问题。

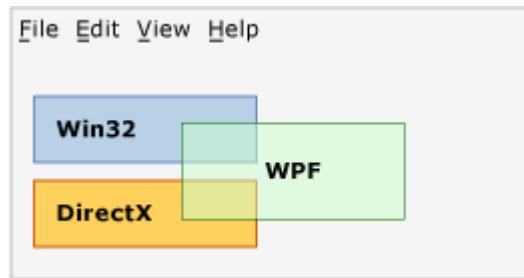


假设此应用程序使用鼠标指针位置来创建想要在这三个区域中任一区域上方呈现的动画。无论动画本身采用哪一种技术，该技术都会与其他两种技术的区域发生冲突。下图演示在一个 Win32 区域上呈现 WPF 圆形的尝试。



如果尝试在不同技术间使用透明度/Alpha 混合，也会发生冲突。在下图中，WPF 框与 Win32 和 DirectX 区域存在冲突。因为该 WPF 框中的像素是半透明的，所以它们必须由

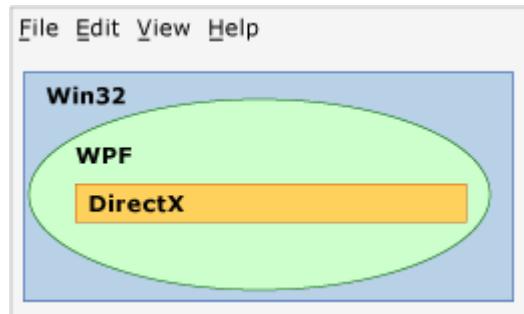
DirectX 和 WPF 共同拥有，但这是不可能的。因此，这是另一种冲突情况，且不可生成。



前面三个示例使用矩形区域，但也可以使用其他形状。例如，区域可以具有一个孔。下图显示了一个带有矩形孔的 Win32 区域，其大小为 WPF 和 DirectX 区域的总大小。



区域也可以完全不是矩形，或可以是可由 Win32 HRGN（区域）描述的任何形状。



## 透明度和顶级窗口

Windows 中的窗口管理器实际上仅处理 Win32 HWND。因此，每个 WPF [Window](#) 都是 HWND。 [Window](#) HWND 必须遵守适用于任何 HWND 的通用规则。在该 HWND 内，WPF 代码可以执行整个 WPF API 支持的任何操作。但是，为实现与桌面上其他 HWND 的交互，WPF 必须遵循 Win32 处理和呈现规则。WPF 通过使用 Win32 API 来支持非矩形窗口，HRGN 用于非矩形窗口，分层窗口用于每像素 Alpha。

不支持常量 Alpha 和颜色键。Win32 分层窗口的功能因平台而异。

分层窗口可通过指定要应用于窗口中每个像素的 Alpha 值来使整个窗口呈现为半透明状。（Win32 实际上支持每像素 Alpha，但这在实际的程序中很难应用，因为在此模式下需要自行绘制任何子 HWND，包括对话框和下拉列表）。

WPF 支持 HRGN；但是，对于此功能，没有相应的托管 API。可以使用平台调用和 [HwndSource](#) 来调用相关的 Win32 API。有关详细信息，请参阅[从托管代码调用本机函数](#)。

WPF 分层窗口在不同操作系统上具有不同的功能。这是因为 WPF 使用 DirectX 进行呈现，而分层窗口主要用于 GDI 呈现，而非 DirectX 呈现。

- WPF 支持硬件加速的分层窗口。
- WPF 不支持透明度颜色键，因为 WPF 无法保证准确呈现所请求的颜色，尤其当呈现采用了硬件加速时更是如此。

## 另请参阅

- [WPF 和 Win32 互操作](#)
- [演练：在 Win32 中承载 WPF 时钟](#)
- [在 WPF 中承载 Win32 内容](#)

# 在 Win32 和 WPF 之间共享消息循环

项目 · 2023/02/06

本主题介绍如何通过使用 [Dispatcher](#) 中的现有消息循环公开，或通过在互操作代码的 Win32 端创建单独的消息循环，为 Windows Presentation Foundation (WPF) 互操作实现消息循环。

## ComponentDispatcher 和消息循环

互操作和键盘事件支持的正常方案是实现 [IKeyboardInputSink](#) 或从已实现 [IKeyboardInputSink](#) 的类创建子类，例如 [HwndSource](#) 或 [HwndHost](#)。但是，键盘接收器支持并不能满足跨互操作边界发送和接收消息时可能遇到的所有可能的消息循环需求。为帮助规范应用程序消息循环体系结构，Windows Presentation Foundation (WPF) 提供了 [ComponentDispatcher](#) 类，该类定义了消息循环应遵循的简单协议。

[ComponentDispatcher](#) 是公开多个成员的静态类。每个方法的范围都隐式绑定到调用线程。消息循环必须在关键时刻调用其中一些 API（如下一部分定义）。

[ComponentDispatcher](#) 提供其他组件（例如键盘接收器）可侦听的事件。[Dispatcher](#) 类以适当顺序调用所有适当的 [ComponentDispatcher](#) 方法。如果要实现自己的消息循环，则你的代码负责以类似方式调用 [ComponentDispatcher](#) 方法。

在线程上调用 [ComponentDispatcher](#) 方法将仅调用在该线程上注册的事件处理程序。

## 编写消息循环

下面是编写自己的消息循环时将使用的 [ComponentDispatcher](#) 成员清单：

- [PushModal](#)：你的消息循环应调用此成员以指示该线程是模式线程。
- [PopModal](#)：你的消息循环应调用此成员以指示线程已还原为非模式。
- [Raiseldle](#)：你的消息循环应调用此成员以指示 [ComponentDispatcher](#) 应引发 [ThreadIdle](#) 事件。如果 [IsThreadModal](#) 为 `true`，[ComponentDispatcher](#) 将不会引发 [ThreadIdle](#)，但消息循环可以选择调用 [Raiseldle](#)，即使 [ComponentDispatcher](#) 无法在模式状态下响应它。
- [RaiseThreadMessage](#)：你的消息循环应调用此成员以指示新消息可用。返回值指示 [ComponentDispatcher](#) 事件的侦听器是否处理了该消息。如果 [RaiseThreadMessage](#) 返回 `true`（表示已处理），调度程序不应对消息执行任何进

一步操作。如果返回值为 `false`，调度程序应调用 Win32 函数 `TranslateMessage`，然后调用 `DispatchMessage`。

## 使用 ComponentDispatcher 和现有消息处理

下面是依赖固有 WPF 消息循环时将使用的 `ComponentDispatcher` 成员清单。

- `IsThreadModal`：返回应用程序是否进入模式状态（例如，模式消息循环已推送）。  
`ComponentDispatcher` 可跟踪此状态，因为该类维持消息循环中 `PushModal` 和 `PopModal` 调用的计数。
- `ThreadFilterMessage` 和 `ThreadPreprocessMessage` 事件遵循委托调用的标准规则。委托以未指定的顺序调用，即使第一个委托将消息标记为已处理，也会调用所有委托。
- `ThreadIdle`：指示执行空闲处理的适当且有效的时间（该线程没有其他挂起的消息）。如果线程为模式线程，则不会引发 `ThreadIdle`。
- `ThreadFilterMessage`：针对消息泵处理的所有消息引发。
- `ThreadPreprocessMessage`：针对 `ThreadFilterMessage` 期间未处理的所有消息引发。

如果在 `ThreadFilterMessage` 事件或 `ThreadPreprocessMessage` 事件后，消息被视为已处理，则事件数据中通过引用传递的 `handled` 参数为 `true`。如果 `handled` 为 `true`，事件处理程序应忽略消息，因为这意味着其他处理程序先处理了消息。这两个事件的事件处理程序可能会修改消息。调度程序应调度修改后的消息，而不是原始未更改的消息。  
`ThreadPreprocessMessage` 传递到所有侦听器，但体系结构意图是，只有包含消息指向的 HWND 的顶级窗口应调用代码以响应消息。

## HwndSource 如何处理 ComponentDispatcher 事件

如果 `HwndSource` 是顶级窗口（没有父 HWND），它将注册到 `ComponentDispatcher`。如果引发 `ThreadPreprocessMessage`，并且消息适用于 `HwndSource` 或子窗口，则 `HwndSource` 调用其 `IKeyboardInputSink.TranslateAccelerator`、`TranslateChar`、`OnMnemonic` 键盘接收器序列。

如果 `HwndSource` 不是顶级窗口（具有父 HWND），则不会进行任何处理。只有顶级窗口才能执行处理，并且在任何互操作方案中，预期会有一个具有键盘接收器支持的顶级窗口。

如果在未先调用适当的键盘接收器方法的情况下调用 [HwndSource](#) 上的 [WndProc](#)，你的应用程序将收到更高级别的键盘事件，例如 [KeyDown](#)。但是，不会调用任何键盘接收器方法，这将避开所需的键盘输入模型功能，例如访问键支持。这可能是因为消息循环未正确通知 [ComponentDispatcher](#) 上的相关线程，或者父 HWND 未调用正确的键盘接收器响应。

如果使用 [AddHook](#) 方法为发往键盘接收器的消息添加了挂钩，则该消息可能不会发送到 HWND。消息可能已在消息泵级别直接处理，而不提交到 [DispatchMessage](#) 函数。

## 另请参阅

- [ComponentDispatcher](#)
- [IKeyboardInputSink](#)
- [WPF 和 Win32 互操作](#)
- [线程模型](#)
- [输入概述](#)

# 在 WPF 中承载 Win32 内容

项目 • 2022/09/27

## 先决条件

请参阅 [WPF 和 Win32 互操作](#)。

## Windows Presentation Framework 中 Win32 的演练 (HwndHost)

要在 WPF 应用程序中重用 Win32 内容，请使用 `HwndHost`，它是使 `HWND` 看起来像 WPF 内容的控件。与 `HwndSource` 相似，`HwndHost` 易于使用：从 `HwndHost` 派生并实现 `BuildWindowCore` 和 `DestroyWindowCore` 方法，然后实例化 `HwndHost` 派生类并将其放置在 WPF 应用程序中。

如果 Win32 逻辑已打包为控件，那么 `BuildWindowCore` 实现只不过是对 `CreateWindow` 的调用。例如，要在 C++ 中创建一个 Win32 LISTBOX 控件：

C++

```
virtual HandleRef BuildWindowCore(HandleRef hwndParent) override {
    HWND handle = CreateWindowEx(0, L"LISTBOX",
        L"This is a Win32 listbox",
        WS_CHILD | WS_VISIBLE | LBS_NOTIFY
        | WS_VSCROLL | WS_BORDER,
        0, 0, // x, y
        30, 70, // height, width
        (HWND) hwndParent.Handle.ToPointer(), // parent hwnd
        0, // hmenu
        0, // hinstance
        0); // lparam

    return HandleRef(this, IntPtr(handle));
}

virtual void DestroyWindowCore(HandleRef hwnd) override {
    // HwndHost will dispose the hwnd for us
}
```

但是假设 Win32 代码不是完全独立的？如果是这样，可以创建一个 Win32 对话框并将其内容嵌入到更大的 WPF 应用程序中。该示例在 Visual Studio 和 C++ 中显示了这一点，但也可以使用不同的语言或在命令行中执行此操作。

从一个简单的对话框开始，它被编译成一个 C++ DLL 项目。

接下来，将对话框引入到更大的 WPF 应用程序中：

- 将 DLL 编译为托管 (/clr)
- 将对话框转换为控件
- 使用 `BuildWindowCore` 和 `DestroyWindowCore` 方法定义 `HwndHost` 的派生类
- 重写 `TranslateAccelerator` 方法来处理对话框键
- 重写 `TabInto` 方法以支持 Tabbing
- 重写 `OnMnemonic` 方法以支持助记符
- 实例化 `HwndHost` 子类并将其放在正确的 WPF 元素下

## 将对话框转换为控件

可以使用 WS\_CHILD 和 DS\_CONTROL 样式将对话框转换为子 HWND。进入定义对话框的资源文件 (.rc)，找到对话框定义的开头：

```
text

IDD_DIALOG1 DIALOGEX 0, 0, 303, 121
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION |
WS_SYSMENU
```

将第二行更改为：

```
text

STYLE DS_SETFONT | WS_CHILD | WS_BORDER | DS_CONTROL
```

该操作并没有完全打包成一个自包含的控件；仍然需要调用 `IsDialogMessage()` 以便 Win32 可以处理某些消息，但控件更改确实提供了将这些控件放入另一个 HWND 的直接方式。

## 子类 HwndHost

导入下列命名空间：

```
C++
```

```
namespace ManagedCpp
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Input;
    using namespace System::Windows::Media;
    using namespace System::Runtime::InteropServices;
```

然后创建 `HwndHost` 的派生类并重写 `BuildWindowCore` 和 `DestroyWindowCore` 方法：

C++

```
public ref class MyHwndHost : public HwndHost, IKeyboardInputSink {
private:
    HWND dialog;

protected:
    virtual HandleRef BuildWindowCore(HandleRef hwndParent) override {
        InitializeGlobals();
        dialog = CreateDialog(hInstance,
            MAKEINTRESOURCE(IDD_DIALOG1),
            (HWND) hwndParent.Handle.ToPointer(),
            (DLGPROC) About);
        return HandleRef(this, IntPtr(dialog));
    }

    virtual void DestroyWindowCore(HandleRef hwnd) override {
        // hwnd will be disposed for us
    }
}
```

在这里，使用 `CreateDialog` 创建真正是控件的对话框。由于这是在 DLL 中首先调用的方法之一，因此还应该通过调用稍后定义的名为 `InitializeGlobals()` 的函数来进行一些标准的 Win32 初始化：

C++

```
bool initialized = false;
void InitializeGlobals() {
    if (initialized) return;
    initialized = true;

    // TODO: Place code here.
    MSG msg;
    HACCEL hAccelTable;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_TYPICALWIN32DIALOG, szWindowClass,
```

```
MAX_LOADSTRING);  
MyRegisterClass(hInstance);
```

## 重写 TranslateAccelerator 方法以处理对话框键

如果现在运行此示例，将获得一个显示的对话框控件，但它将忽略使对话框成为功能对话框的所有键盘处理。现在应重写 `TranslateAccelerator` 实现（它来自 `IKeyboardInputSink`，一个由 `HwndHost` 实现的接口）。当应用程序接收到 `WM_KEYDOWN` 和 `WM_SYSKEYDOWN` 时调用此方法。

C++

```
#undef TranslateAccelerator  
virtual bool TranslateAccelerator(System::Windows::Interop::MSG%  
msg,  
ModifierKeys modifiers) override  
{  
    ::MSG m = ConvertMessage(msg);  
  
    // Win32's IsDialogMessage() will handle most of our tabbing,  
but doesn't know  
    // what to do when it reaches the last tab stop  
    if (m.message == WM_KEYDOWN && m.wParam == VK_TAB) {  
        HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);  
        HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);  
        TraversalRequest^ request = nullptr;  
  
        if (GetKeyState(VK_SHIFT) && GetFocus() == firstTabStop) {  
            // this code should work, but there's a bug with interop  
shift-tab in current builds  
            request = gcnew  
TraversalRequest(FocusNavigationDirection::Last);  
        }  
        else if (!GetKeyState(VK_SHIFT) && GetFocus() ==  
lastTabStop) {  
            request = gcnew  
TraversalRequest(FocusNavigationDirection::Next);  
        }  
  
        if (request != nullptr)  
            return ((IKeyboardInputSink^) this)->KeyboardInputSite-  
>OnNoMoreTabStops(request);  
    }  
  
    // Only call IsDialogMessage for keys it will do something with.  
    if (msg.message == WM_SYSKEYDOWN || msg.message == WM_KEYDOWN) {  
        switch (m.wParam) {  
            case VK_TAB:  
            case VK_LEFT:  
            case VK_UP:
```

```

        case VK_RIGHT:
        case VK_DOWN:
        case VK_EXECUTE:
        case VK_RETURN:
        case VK_ESCAPE:
        case VK_CANCEL:
            IsDialogMessage(dialog, &m);
            // IsDialogMessage should be called
ProcessDialogMessage -- // it processes messages without ever really telling
you // if it handled a specific message or not
return true;
    }
}

return false; // not a key we handled
}

```

这是一大段代码，所因此可以使用一些更详细的说明。首先，使用 C++ 和 C++ 宏的代码；需要注意已经有一个名为 `TranslateAccelerator` 的宏，它在 `winuser.h` 中定义：

C++

```
#define TranslateAccelerator TranslateAcceleratorW
```

因此，请确保定义 `TranslateAccelerator` 方法而不是 `TranslateAcceleratorW` 方法。

同样，有非托管的 `winuser.h` `MSG` 和托管的 `Microsoft::Win32::MSG` 结构。可以使用 C++ `::` 运算符消除两者之间的歧义。

C++

```

virtual bool TranslateAccelerator(System::Windows::Interop::MSG% msg,
ModifierKeys modifiers) override
{
    ::MSG m = ConvertMessage(msg);
}
```

两个 `MSG` 具有相同的数据，但有时使用非托管定义更容易，因此在此示例中可以定义明显的转换例程：

C++

```

::MSG ConvertMessage(System::Windows::Interop::MSG% msg) {
    ::MSG m;
    m.hwnd = (HWND) msg.hwnd.ToPointer();
    m.lParam = (LPARAM) msg.lParam.ToPointer();
    m.message = msg.message;
```

```

m.wParam = (WPARAM) msg.wParam.ToPointer();

m.time = msg.time;

POINT pt;
pt.x = msg.pt_x;
pt.y = msg.pt_y;
m.pt = pt;

return m;
}

```

返回到 `TranslateAccelerator`。基本原则是调用 Win32 函数 `IsDialogMessage` 来做尽可能多的工作，但 `IsDialogMessage` 无权访问对话之外的任何内容。作为对话周围的用户选项卡，当 Tabbing 在我们的对话中运行最后一个控件时，需要通过调用 `IKeyboardInputSite::OnNoMoreStops` 将焦点设置为 WPF 部分。

C++

```

// Win32's IsDialogMessage() will handle most of the tabbing, but doesn't
// know
// what to do when it reaches the last tab stop
if (m.message == WM_KEYDOWN && m.wParam == VK_TAB) {
    HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
    HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);
    TraversalRequest^ request = nullptr;

    if (GetKeyState(VK_SHIFT) && GetFocus() == firstTabStop) {
        request = gcnew TraversalRequest(FocusNavigationDirection::Last);
    }
    else if (!GetKeyState(VK_SHIFT) && GetFocus() == lastTabStop) { {
        request = gcnew TraversalRequest(FocusNavigationDirection::Next);
    }

    if (request != nullptr)
        return ((IKeyboardInputSink^) this)->KeyboardInputSite-
>OnNoMoreTabStops(request);
}

```

最后，调用 `IsDialogMessage`。但是 `TranslateAccelerator` 方法的职责之一是告诉 WPF 是否处理了击键。如果未处理，输入事件可以通过应用程序的其余部分进行隧道化和气泡处理。在这里，将揭示键盘消息处理的一个习惯以及 Win32 中输入体系结构的性质。不幸的是，`IsDialogMessage` 不会以任何方式返回它是否处理特定击键的信息。更糟糕的是，它会在它不应处理的击键上调用 `DispatchMessage()`！因此，必须对 `IsDialogMessage` 进行逆向工程，并且只为已知其将处理的键调用它：

C++

```

// Only call IsDialogMessage for keys it will do something with.
if (msg.message == WM_SYSKEYDOWN || msg.message == WM_KEYDOWN) {
    switch (m.wParam) {
        case VK_TAB:
        case VK_LEFT:
        case VK_UP:
        case VK_RIGHT:
        case VK_DOWN:
        case VK_EXECUTE:
        case VK_RETURN:
        case VK_ESCAPE:
        case VK_CANCEL:
            IsDialogMessage(dialog, &m);
            // IsDialogMessage should be called ProcessDialogMessage --
            // it processes messages without ever really telling you
            // if it handled a specific message or not
            return true;
    }
}

```

## 重写 TabInto 方法以支持 Tabbing

既已实现 `TranslateAccelerator`，用户便可以在对话框内使用 Tab 键并从它跳出进入更大的 WPF 应用程序。但是用户不能按 Tab 键返回对话框。为解决这个问题，重写 `TabInto`：

C++

```

public:
    virtual bool TabInto(TraversalRequest^ request) override {
        if (request->FocusNavigationDirection ==
FocusNavigationDirection::Last) {
            HWND lastTabStop = GetDlgItem(dialog, IDCANCEL);
            SetFocus(lastTabStop);
        }
        else {
            HWND firstTabStop = GetDlgItem(dialog, IDC_EDIT1);
            SetFocus(firstTabStop);
        }
        return true;
    }
}

```

`TraversalRequest` 参数指示 Tab 操作是 Tab 还是 Shift+Tab。

## 重写 OnMnemonic 方法以支持助记符

键盘处理几乎完成了，但缺少一件事 – 助记符不起作用。如果用户按下 Alt-F，焦点不会跳转到“名字：“编辑框。因此，重写 `OnMnemonic` 方法：

C++

```
virtual bool OnMnemonic(System::Windows::Interop::MSG% msg, ModifierKeys modifiers) override {
    ::MSG m = ConvertMessage(msg);

    // If it's one of our mnemonics, set focus to the appropriate hwnd
    if (msg.message == WM_SYSCHAR && GetKeyState(VK_MENU /*alt*/) < 0) {
        int dialogitem = 9999;
        switch (m.wParam) {
            case 's': dialogitem = IDOK; break;
            case 'c': dialogitem = IDCANCEL; break;
            case 'f': dialogitem = IDC_EDIT1; break;
            case 'l': dialogitem = IDC_EDIT2; break;
            case 'p': dialogitem = IDC_EDIT3; break;
            case 'a': dialogitem = IDC_EDIT4; break;
            case 'i': dialogitem = IDC_EDIT5; break;
            case 't': dialogitem = IDC_EDIT6; break;
            case 'z': dialogitem = IDC_EDIT7; break;
        }
        if (dialogitem != 9999) {
            HWND hwnd = GetDlgItem(dialog, dialogitem);
            SetFocus(hwnd);
            return true;
        }
    }
    return false; // key unhandled
};
```

为什么不在此处调用 `IsDialogMessage`？遇到与以前相同的问题，需要能够通知 WPF 代码是否处理了击键，而 `IsDialogMessage` 无法执行此操作。还有第二个问题，因为如果焦点 HWND 不在对话框内，`IsDialogMessage` 将拒绝处理助记符。

## 实例化 HwndHost 派生类

最后，既然所有键和选项卡支持都已到位，可以将 `HwndHost` 放入更大的 WPF 应用程序中。如果主应用程序是用 XAML 编写的，将其放置在正确位置的最简单方法是在要放置 `HwndHost` 的位置留下一个空的 `Border` 元素。此处，创建了一个名为 `insertHwndHostHere` 的 `Border`：

XAML

```
<Window x:Class="WPFApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Windows Presentation Framework Application"
    Loaded="Window1_Loaded"
    >
    <StackPanel>
```

```
<Button Content="WPF button"/>
<Border Name="insertHwndHostHere" Height="200" Width="500"/>
<Button Content="WPF button"/>
</StackPanel>
</Window>
```

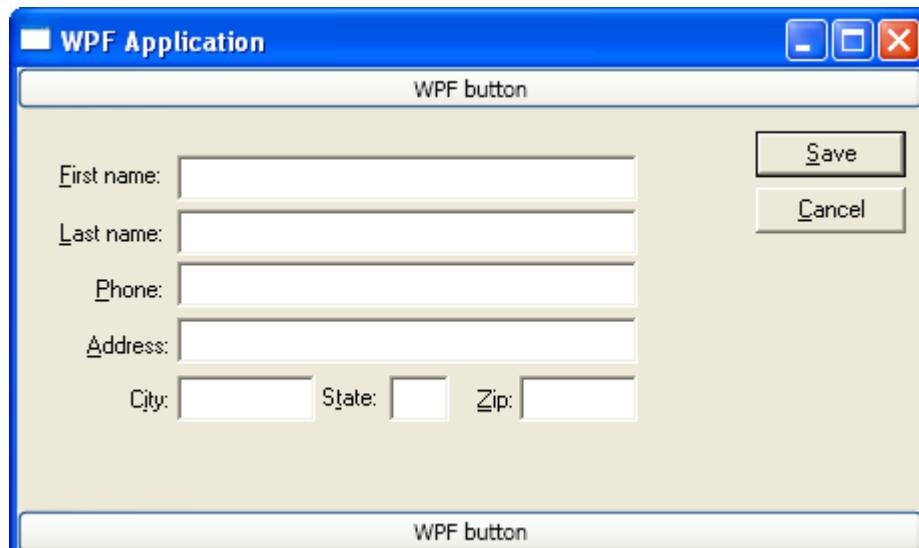
之后就是在代码序列中找到一个合适的位置来实例化 `HwndHost` 并将其连接到 `Border`。在本例中，将把它放置在 `Window` 派生类的构造函数中：

C#

```
public partial class Window1 : Window {
    public Window1() {
    }

    void Window1_Loaded(object sender, RoutedEventArgs e) {
        HwndHost host = new ManagedCpp.MyHwndHost();
        insertHwndHostHere.Child = host;
    }
}
```

这为你提供：



## 另请参阅

- [WPF 和 Win32 互操作](#)

# 演练：在 WPF 中承载 Win32 控件

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 提供了用于创建应用程序的丰富环境。但是，当你对 Win32 代码有大量投入时，在 WPF 应用程序中重复使用至少某些代码（而不是彻底替代）可能更有效。WPF 提供了一个简单的机制，用于在 WPF 页面上承载 Win32 窗口。

本主题介绍用于承载 Win32 列表框控件的应用程序，即[在 WPF 中承载 Win32 ListBox 控件示例](#)。可将此常规步骤扩展到承载任何 Win32 窗口。

## 要求

本主题假定你已基本熟悉 WPF 和 Windows API 编程。有关 WPF 编程的基本介绍，请参阅[入门](#)。有关 Windows API 编程的介绍，请参考有关该主题的众多书籍，尤其是 Charles Petzold 所著的“Programming Windows”。

由于此主题随附的示例是用 C# 实现的，因此它使用 Platform Invocation Services (PInvoke) 来访问 Windows API。稍微熟悉 PInvoke 将有所帮助，但并非必备条件。

### ① 备注

本主题包括来自相关示例的一些代码示例。但是，出于可读性考虑，不包括完整的示例代码。若要获取或查看完整代码，请访问[在 WPF 中承载 Win32 ListBox 控件示例](#)。

## 基本过程

本部分概述了在 WPF 页面中承载 Win32 窗口的基本步骤。其余各节介绍了每个步骤的详细内容。

基本的承载步骤如下：

1. 实现一个 WPF 页面以承载窗口。一种方法是创建 [Border](#) 元素，以便为所承载的窗口保留该页的一部分。
2. 实现一个类以承载继承自 [HwndHost](#) 的控件。
3. 在该类中，替代 [HwndHost](#) 类成员 [BuildWindowCore](#)。

4. 将所承载的窗口创建为包含 WPF 页面的窗口的子窗口。尽管传统的 WPF 编程不需要显式利用承载页，但需要指出的是，承载页是一个带有句柄 (HWND) 的窗口。你通过 [BuildWindowCore](#) 方法的 `hwndParent` 参数接收页面 HWND。应将所承载的窗口创建为此 HWND 的子窗口。
5. 在创建宿主窗口之后，返回所承载的窗口的 HWND。如果想要承载一个或多个 Win32 控件，通常需要将宿主窗口创建为该 HWND 的子窗口，并使这些控件成为该宿主窗口的子窗口。通过将控件包装在宿主窗口中，可以提供一种让 WPF 页从这些控件接收通知的简单方式，该方式可以处理一些与跨 HWND 边界的通知有关的特定 Win32 问题。
6. 处理发送到宿主窗口的选定消息，例如，来自子控件的通知。可通过两种方式来执行此操作。
  - 如果希望在承载类中处理消息，可以替代 [HwndHost](#) 类的 [WndProc](#) 方法。
  - 如果希望让 WPF 处理消息，可以在代码隐藏中处理 [HwndHost](#) 类 [MessageHook](#) 事件。对于所承载的窗口收到的每条消息，都将发生此事件。如果选择此选项，仍然必须替代 [WndProc](#)，但只需提供最小实现。
7. 替代 [HwndHost](#) 的 [DestroyWindowCore](#) 和 [WndProc](#) 方法。必须替代这些方法才能履行 [HwndHost](#) 协议，但只需提供最小实现。
8. 在代码隐藏文件中，创建控件承载类的一个实例，并使其成为用于承载窗口的 [Border](#) 元素的子元素。
9. 通过向所承载的窗口发送 Microsoft Windows 消息以及处理来自其子窗口的消息（例如由控件发送的通知），与该窗口进行通信。

## 实现页面布局

承载 [ListBox](#) 控件的 WPF 页面的布局由两个区域组成。页面左侧承载了多个 WPF 控件，这些控件提供了用户界面 (UI)，使你可以操作 Win32 控件。页面右上角具有一个正方形区域，用于放置所承载的 [ListBox](#) 控件。

用于实现此操作的代码十分简单。根元素是具有两个子元素的 [DockPanel](#)。第一个是承载 [ListBox](#) 控件的 [Border](#) 元素。它在该页的右上角占据了一个大小为 200x200 的正方形。第二个是包含一组 WPF 控件的 [StackPanel](#) 元素，这些控件显示信息，并使你可以通过设置已公开的互操作属性来操作 [ListBox](#) 控件。对于 [StackPanel](#) 的每个子元素，请参阅关于所用的各种元素的参考资料，了解关于这些元素是什么或者它们有哪些功能的详细信息。下面的代码示例中列出了这些元素，但这里不对对其进行说明（基本互操作模型不需要它们中的任何一个，提供它们的目的是为该示例增加一些交互性）。

## XAML

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="WPF_Hosting_Win32_Control.HostWindow"
    Name="mainWindow"
    Loaded="On_UIReady">

    <DockPanel Background="LightGreen">
        <Border Name="ControlHostElement"
            Width="200"
            Height="200"
            HorizontalAlignment="Right"
            VerticalAlignment="Top"
            BorderBrush="LightGray"
            BorderThickness="3"
            DockPanel.Dock="Right"/>
        <StackPanel>
            <Label HorizontalAlignment="Center"
                Margin="0,10,0,0"
                FontSize="14"
                FontWeight="Bold">Control the Control</Label>
            <TextBlock Margin="10,10,10,10" >Selected Text: <TextBlock
                Name="selectedText"/></TextBlock>
            <TextBlock Margin="10,10,10,10" >Number of Items: <TextBlock
                Name="numItems"/></TextBlock>

            <Line X1="0" X2="200"
                Stroke="LightYellow"
                StrokeThickness="2"
                HorizontalAlignment="Center"
                Margin="0,20,0,0"/>

            <Label HorizontalAlignment="Center"
                Margin="10,10,10,10">Append an Item to the List</Label>
            <StackPanel Orientation="Horizontal">
                <Label HorizontalAlignment="Left"
                    Margin="10,10,10,10">Item Text</Label>
                <TextBox HorizontalAlignment="Left"
                    Name="txtAppend"
                    Width="200"
                    Margin="10,10,10,10"></TextBox>
            </StackPanel>

            <Button HorizontalAlignment="Left"
                Click="AppendText"
                Width="75"
                Margin="10,10,10,10">Append</Button>

            <Line X1="0" X2="200"
                Stroke="LightYellow"
                StrokeThickness="2"
                HorizontalAlignment="Center"
                Margin="0,20,0,0"/>
        </StackPanel>
    </DockPanel>

```

```
Margin="0,20,0,0"/>

<Label HorizontalAlignment="Center"
Margin="10,10,10,10">Delete the Selected Item</Label>

<Button Click="DeleteText"
Width="125"
Margin="10,10,10,10"
HorizontalAlignment="Left">Delete</Button>
</StackPanel>
</DockPanel>
</Window>
```

## 实现类以承载 Microsoft Win32 控件

此示例的核心是实际承载控件的类，即 ControlHost.cs。它继承自 [HwndHost](#)。构造函数接受两个参数，即 height 和 width，它们分别对应于承载 ListBox 控件的 Border 元素的高度和宽度。这些值将在以后用于确保控件的大小与 Border 元素相匹配。

C#

```
public class ControlHost : HwndHost
{
    IntPtr hwndControl;
    IntPtr hwndHost;
    int hostHeight, hostWidth;

    public ControlHost(double height, double width)
    {
        hostHeight = (int)height;
        hostWidth = (int)width;
    }
}
```

还有一组常量。这些常量主要取自 Winuser.h，它们使你可以在调用 Win32 函数时使用约定名称。

C#

```
internal const int
WS_CHILD = 0x40000000,
WS_VISIBLE = 0x10000000,
LBS_NOTIFY = 0x00000001,
HOST_ID = 0x00000002,
LISTBOX_ID = 0x00000001,
WS_VSCROLL = 0x00200000,
WS_BORDER = 0x00800000;
```

# 替代 BuildWindowCore 以创建 Microsoft Win32 窗口

替代此方法以创建将由页面承载的 Win32 窗口，并在窗口与页面之间建立连接。由于此示例涉及承载 ListBox 控件，因此将创建两个窗口。第一个窗口是由 WPF 页面实际承载的窗口。ListBox 控件被创建为该窗口的子窗口。

采用此方法的原因是为了简化接收来自控件的通知的过程。[HwndHost](#) 类使你能够处理发送到它所承载的窗口中的消息。如果直接承载 Win32 控件，你将收到发送到该控件内部消息循环的消息。你可以显示控件并向其发送消息，但不会收到该控件发送到其父窗口的通知。这意味着你没有办法检测用户何时与该控件进行交互，以及其他更多方面。可改为创建一个宿主窗口，并使控件成为该窗口的子窗口。这样，你便能够处理宿主窗口的消息，包括由该控件发送给它的通知。由于宿主窗口只是控件的一个简单包装器而已，为了方便起见，以后将把该包称为 ListBox 控件。

## 创建宿主窗口和 ListBox 控件

可以使用 [PInvoke](#) 通过创建和注册窗口类等来创建控件的宿主窗口。但是，更简单的一种方法是使用预定义的“静态”窗口类创建窗口。这将为你提供所需窗口步骤，以接收来自控件的通知，但需要完成最少的编码工作。

控件的 [HWND](#) 通过一个只读属性公开，以便宿主页面可使用它向控件发送消息。

```
C#  
  
public IntPtr hwndListBox  
{  
    get { return hwndControl; }  
}
```

ListBox 控件被创建为宿主窗口的子窗口。两个窗口的高度和宽度被设置为传递给构造函数的值（前面已讨论）。这可确保宿主窗口和控件的大小与页面上的保留区域相同。在创建窗口之后，该示例将返回一个 [HandleRef](#) 对象，其中包含宿主窗口的 [HWND](#)。

```
C#  
  
protected override HandleRef BuildWindowCore(HandleRef hwndParent)  
{  
    hwndControl = IntPtr.Zero;  
    hwndHost = IntPtr.Zero;  
  
    hwndHost = CreateWindowEx(0, "static", "",  
                             WS_CHILD | WS_VISIBLE,  
                             0, 0,  
                             hostWidth, hostHeight,  
                             hwndParent.Handle,  
                             (IntPtr)HOST_ID,
```

```

        IntPtr.Zero,
    );

hwndControl = CreateWindowEx(0, "listbox", "",
    WS_CHILD | WS_VISIBLE | LBS_NOTIFY
    | WS_VSCROLL | WS_BORDER,
    0, 0,
    hostWidth, hostHeight,
    hwndHost,
    (IntPtr) LISTBOX_ID,
    IntPtr.Zero,
    0);

return new HandleRef(this, hwndHost);
}

```

C#

```

//PInvoke declarations
[DllImport("user32.dll", EntryPoint = "CreateWindowEx", CharSet =
CharSet.Unicode)]
internal static extern IntPtr CreateWindowEx(int dwExStyle,
                                             string lpszClassName,
                                             string lpszWindowName,
                                             int style,
                                             int x, int y,
                                             int width, int height,
                                             IntPtr hwndParent,
                                             IntPtr hMenu,
                                             IntPtr hInst,

[MarshalAs(UnmanagedType.AsAny)] object pvParam);

```

## 实现 DestroyWindow 和 WndProc

除 `BuildWindowCore` 外，还必须替代 `HwndHost` 的 `WndProc` 和 `DestroyWindowCore` 方法。在本示例中，控件的消息由 `MessageHook` 处理程序处理，因此 `WndProc` 和 `DestroyWindowCore` 的实现是极少的。在 `WndProc` 的情况下，将 `handled` 设为 `false`，以指示消息未处理，且返回 0。对于 `DestroyWindowCore`，只需销毁该窗口。

C#

```

protected override IntPtr WndProc(IntPtr hwnd, int msg, IntPtr wParam,
IntPtr lParam, ref bool handled)
{
    handled = false;
    return IntPtr.Zero;
}

```

```
protected override void DestroyWindowCore(HandleRef hwnd)
{
    DestroyWindow(hwnd.Handle);
}
```

C#

```
[DllImport("user32.dll", EntryPoint = "DestroyWindow", CharSet =
CharSet.Unicode)]
internal static extern bool DestroyWindow(IntPtr hwnd);
```

## 在页面上承载控件

若要在页面上承载控件，首先需要创建 `ControlHost` 类的新实例。将包含控件 (`ControlHostElement`) 的边框元素的高度和宽度传递给 `ControlHost` 构造函数。这可确保 `ListBox` 具有正确的大小。然后通过将 `ControlHost` 对象分配给主机 `Border` 的 `Child` 属性来承载页面上的控件。

此示例将一个处理程序附加到 `ControlHost` 的 `MessageHook` 事件，以接收来自控件的消息。对于每条发送到宿主窗口的消息，都将引发此事件。在这种情况下，这些消息是发送到包装实际 `ListBox` 控件的窗口的消息，其中包括来自控件的通知。此示例将调用 `SendMessage` 以从控件获取信息并修改其内容。下一节将详细介绍页面如何与控件通信。

### ① 备注

请注意，`SendMessage` 有两个 `PInvoke` 声明。这是必需的，因为其中一个声明使用 `wParam` 参数传递字符串，而另一个声明使用该参数传递整数。对于每个签名，都需要一个单独的声明，以确保正确封送数据。

C#

```
public partial class HostWindow : Window
{
    int selectedItem;
    IntPtr hwndListBox;
    ControlHost listControl;
    Application app;
    Window myWindow;
    int itemCount;

    private void On_UIReady(object sender, EventArgs e)
    {
        app = System.Windows.Application.Current;
```

```

myWindow = app.MainWindow;
myWindow.SizeToContent = SizeToContent.WidthAndHeight;
listControl = new ControlHost(ControlHostElement.ActualHeight,
ControlHostElement.ActualWidth);
ControlHostElement.Child = listControl;
listControl.MessageHook += new HwndSourceHook(ControlMsgFilter);
hwndListBox = listControl(hwndListBox);
for (int i = 0; i < 15; i++) //populate listbox
{
    string itemText = "Item" + i.ToString();
    SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, itemText);
}
itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero,
IntPtr.Zero);
numItems.Text = "" + itemCount.ToString();
}

```

C#

```

private IntPtr ControlMsgFilter(IntPtr hwnd, int msg, IntPtr wParam, IntPtr
lParam, ref bool handled)
{
    int textLength;

    handled = false;
    if (msg == WM_COMMAND)
    {
        switch ((uint)wParam.ToInt32() >> 16 & 0xFFFF) //extract the HIWORD
        {
            case LBN_SELCHANGE : //Get the item text and display it
                selectedItem = SendMessage(listControl(hwndListBox, LB_GETCURSEL,
IntPtr.Zero, IntPtr.Zero);
                textLength = SendMessage(listControl(hwndListBox, LB_GETTEXTLEN,
IntPtr.Zero, IntPtr.Zero);
                StringBuilder itemText = new StringBuilder();
                SendMessage(hwndListBox, LB_GETTEXT, selectedItem, itemText);
                selectedText.Text = itemText.ToString();
                handled = true;
                break;
        }
    }
    return IntPtr.Zero;
}
internal const int
LBN_SELCHANGE = 0x00000001,
WM_COMMAND = 0x00000111,
LB_GETCURSEL = 0x00000188,
LB_GETTEXTLEN = 0x0000018A,
LB_ADDSTRING = 0x00000180,
LB_GETTEXT = 0x00000189,
LB_DELETESTRING = 0x00000182,
LB_GETCOUNT = 0x0000018B;

```

```
[DllImport("user32.dll", EntryPoint = "SendMessage", CharSet =
    CharSet.Unicode)]
internal static extern int SendMessage(IntPtr hwnd,
    int msg,
    IntPtr wParam,
    IntPtr lParam);

[DllImport("user32.dll", EntryPoint = "SendMessage", CharSet =
    CharSet.Unicode)]
internal static extern int SendMessage(IntPtr hwnd,
    int msg,
    int wParam,
    [MarshalAs(UnmanagedType.LPWStr)]
StringBuilder lParam);

[DllImport("user32.dll", EntryPoint = "SendMessage", CharSet =
    CharSet.Unicode)]
internal static extern IntPtr SendMessage(IntPtr hwnd,
    int msg,
    IntPtr wParam,
    String lParam);
```

## 在控件和页面之间的实现通信

可以通过向控件发送 Windows 消息来操作控件。当用户与控件交互时，控件会通过向其宿主窗口发送通知来通知你。[在 WPF 中承载 Win32 ListBox 控件](#)示例包含一个 UI，其中提供了演示这一机制的工作方式的几个示例：

- 向列表中追加项。
- 从列表中删除选定项
- 显示当前选定项的文本。
- 显示列表中的项数。

还可以通过单击列表中的项来选择它，就像在常用 Win32 应用程序中一样。每当用户通过选择、添加或追加项来更改列表框的状态时，都将更新所显示的数据。

若要追加项，请向列表框发送 [LB\\_ADDSTRING 消息](#)。若要删除项，请发送 [LB\\_GETCURSEL](#) 获取当前选定项的索引，然后发送 [LB\\_DELETESTRING](#) 以删除项。此示例还将发送 [LB\\_GETCOUNT](#)，并使用返回值更新显示项数的显示器。[SendMessage](#) 的这两个实例都使用上一节中所述的 PInvoke 声明之一。

```

private void AppendText(object sender, EventArgs args)
{
    if (!string.IsNullOrEmpty(txtAppend.Text))
    {
        SendMessage(hwndListBox, LB_ADDSTRING, IntPtr.Zero, txtAppend.Text);
    }
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero,
    IntPtr.Zero);
    numItems.Text = "" + itemCount.ToString();
}
private void DeleteText(object sender, EventArgs args)
{
    selectedItem = SendMessage(listControl(hwndListBox, LB_GETCURSEL,
    IntPtr.Zero, IntPtr.Zero);
    if (selectedItem != -1) //check for selected item
    {
        SendMessage(hwndListBox, LB_DELETESTRING, (IntPtr)selectedItem,
        IntPtr.Zero);
    }
    itemCount = SendMessage(hwndListBox, LB_GETCOUNT, IntPtr.Zero,
    IntPtr.Zero);
    numItems.Text = "" + itemCount.ToString();
}

```

当用户选择项或更改其选择时，控件通过向承载窗口发送 [WM\\_COMMAND 消息](#)（这会引发页面的 [MessageHook 事件](#)），通知承载窗口。处理程序将接收到与宿主窗口的主窗口过程相同的信息。它还将传递对布尔值 `handled` 的引用。将 `handled` 设为 `true` 可以指示你已经处理该消息并且无需进行其他处理。

发送 [WM\\_COMMAND](#) 的原因多种多样，因此必须检查通知 ID 以确定它是否是想要处理的事件。该 ID 包含在 `wParam` 参数的高位字中。示例使用按位运算符来提取 ID。如果用户已选择或更改其选择，该 ID 将为 [LBN\\_SELCHANGE](#)。

收到 [LBN\\_SELCHANGE](#) 后，该示例将通过向控件发送 [LB\\_GETCURSEL 消息](#) 获取关于选定项的索引。如需获取文本，首先创建 [StringBuilder](#)。然后，向控件发送一条 [LB\\_GETTEXT 消息](#)。将空 [StringBuilder](#) 对象作为 `wParam` 参数传递。`SendMessage` 返回时，[StringBuilder](#) 将包含所选项的文本。`SendMessage` 的这一用法还需要另一个 `Invoke` 声明。

最后，将 `handled` 设为 `true` 以指示该消息已得到处理。

## 另请参阅

- [HwndHost](#)
- [WPF 和 Win32 互操作](#)
- [演练：我的第一个 WPF 桌面应用程序](#)



# 演练：在 Win32 中承载 WPF 内容

项目 • 2023/02/06

向应用程序添加 WPF 功能，而不是重写原始代码。WPF 提供了一个简单的机制，用于在 Win32 窗口中承载 WPF 内容。

本教程介绍如何编写示例应用程序，在 [Win32 窗口示例中承载 WPF 内容](#)，该应用程序可在 Win32 窗口中承载 WPF 内容。你可以扩展此示例，使其可承载任何 Win32 窗口。由于涉及混合托管代码和非托管代码，应用程序是使用 C++/CLI 编写的。

## 要求

本教程假定你已基本熟悉 WPF 和 Win32 编程。有关 WPF 编程的基本介绍，请参阅[入门](#)。有关 Win32 编程的简介，可参考有关该主题的众多书籍，尤其是 Charles Petzold 所著的 Programming Windows（《Windows 编程》）。

由于此教程随附的示例是使用 C++/CLI 实现的，因而本教程假定你熟悉使用 C++ 进行 Windows API 编程，并且了解托管代码编程。熟悉 C++/CLI 将有所帮助，但并非必备条件。

### ① 备注

本教程包括来自相关示例的一些代码示例。但是，出于可读性考虑，不包括完整的示例代码。有关完整的示例代码，请参阅[在 Win32 窗口示例中承载 WPF 内容](#)。

## 基本过程

本节概括介绍了用于在 Win32 窗口中承载 WPF 内容的基本过程。其余章节说明每个步骤的详细内容。

在 Win32 窗口中承载 WPF 内容的关键是 [HwndSource](#) 类。此类在 Win32 窗口中包装 WPF 内容，从而使其可在单个应用程序中并入你的 WPF 中。

1. 将你的 WPF 内容作为托管类实现。
2. 使用 C++/CLI 实现 Windows 应用程序。如果从现有应用程序和非托管 C++ 代码开始，你通常可以通过更改项目设置以包括 `/clr` 编译器标志来使其可以调用托管代码。
3. 将线程处理模型设置为单线程单元 (STA)。

4. 在窗口过程中处理 `WM_CREATE` 通知，并执行以下任务：
  - a. 创建一个新的 `HwndSource` 对象，将父窗口用作其 `parent` 参数。
  - b. 创建 WPF 内容类的一个实例。
  - c. 向 `HwndSource` 的 `RootVisual` 属性分配对 WPF 内容对象的引用。
  - d. 获取该内容的 `HWND`。`Handle` 对象的 `HwndSource` 属性包含窗口句柄 (`HWND`)。要获取可在应用程序的非托管部分中使用的 `HWND`，需将 `Handle.ToPointer()` 强制装换为 `HWND`。
5. 实现一个托管类，该类包含一个用于保存对 WPF 内容的引用的静态字段。该类使你可以从 Win32 代码获取对 WPF 内容的引用。
6. 向静态字段分配 WPF 内容。
7. 通过将处理程序附加到一个或多个 WPF 事件来从 WPF 内容接收通知。
8. 通过使用存储在静态字段中用于设置属性等的引用来与 WPF 内容通信。

① 备注

你也可以使用 WPF 内容。但是，你必须将其作为动态链接库 (DLL) 单独编译并从 Win32 应用程序引用 DLL。该过程的其余部分与上述相似。

## 实现主机应用程序

本节介绍如何在基本的 Win32 应用程序中托管 WPF 内容。该内容本身是在 C++/CLI 中作为托管类实现的。大多数情况下，它是简单的 WPF 编程。在[实现 WPF 内容](#)中对内容实现的主要方面进行了探讨。

- [基本应用程序](#)
- [承载 WPF 内容](#)
- [保存对 WPF 内容的引用](#)
- [与 WPF 内容通信](#)

## 基本应用程序

主机应用程序的起点是创建 Visual Studio 2005 模板。

1. 打开 Visual Studio 2005，然后从“文件”菜单中选择“新建项目”。
2. 在 Visual C++ 项目类型列表中选择“Win32”。如果默认语言不是 C++，你将在“其他语言”下找到这些项目类型。
3. 选择“Win32 项目”模板，为项目分配一个名称，然后单击“确定”以启动“Win32 应用程序向导”。
4. 接受向导的默认设置，然后单击“完成”以启动项目。

该模板将创建一个基本的 Win32 应用程序，包括：

- 应用程序的入口点。
- 一个窗口和关联的窗口过程 (WndProc)。
- 带有“文件”和“帮助”标题的菜单。“文件”菜单具有用于关闭应用程序的“退出”项。“帮助”菜单具有用于启动一个简单对话框的“关于”项。

在开始编写用于承载 WPF 内容的代码之前，需要对基本模板做两项修改。

第一项是将项目作为托管代码进行编译。默认情况下，项目将作为非托管代码进行编译。但是，由于 WPF 是在托管代码中实现的，因此必须将项目进行相应编译。

1. 在“解决方案资源管理器”中右键单击项目名，然后从上下文菜单中选择“属性”以启动“属性页”对话框。
2. 从左窗格中的树视图中选择“配置属性”。
3. 从右窗格中的“项目默认值”列表中选择“公共语言运行时”支持。
4. 从下拉列表框中选择“公共语言运行时支持(/clr)”。

### ① 备注

此编译器标志使你能够在应用程序中使用托管代码，但非托管代码将仍将像以前一样进行编译。

WPF 使用单线程单元 (STA) 线程处理模型。为了正常使用 WPF 内容代码，必须通过对入口点应用特性，从而将应用程序的线程模型设置为 STA。

C++

```
[System::STAThreadAttribute] //Needs to be an STA thread to play nicely with
WPF
int APIENTRY _tWinMain(HINSTANCE hInstance,
                      HINSTANCE hPrevInstance,
```

```
    LPTSTR    lpCmdLine,
    int       nCmdShow)
{
```

## 承载 WPF 内容

WPF 内容是简单的地址条目应用程序。它包含若干个 `TextBox` 控件，这些控件用于获得用户名称、地址等。还有两个 `Button` 控件，即“确定”和“取消”。用户单击“确定”时，该按钮的 `Click` 事件处理程序将收集来自 `TextBox` 控件的数据，将其分配给相应的属性，并引发自定义事件 `OnButtonClicked`。当用户单击“取消”时，该处理程序只引发 `OnButtonClicked`。`OnButtonClicked` 的事件参数对象包含布尔型字段，用于指示被单击的按钮。

用于托管 WPF 内容的代码在主机窗口上的 `WM_CREATE` 通知的处理程序中实现。

C++

```
case WM_CREATE :
    GetClientRect(hWnd, &rect);
    wpfHwnd = GetHwnd(hWnd, rect.right-375, 0, 375, 250);
    CreateDataDisplay(hWnd, 275, rect.right-375, 375);
    CreateRadioButtons(hWnd);
break;
```

`GetHwnd` 方法使用大小和位置信息以及父窗口句柄并返回托管的 WPF 内容的窗口句柄。

### ① 备注

无法对 `#using` 命名空间使用 `System::Windows::Interop` 指令。如果进行此操作，将造成该命名空间中的 `MSG` 结构和在 `winuser.h` 中声明的 `MSG` 结构之间的名称冲突。必须使用完全限定名来访问该命名空间的内容。

C++

```
HWND GetHwnd(HWND parent, int x, int y, int width, int height)
{
    System::Windows::Interop::HwndSourceParameters^ sourceParams = gcnew
    System::Windows::Interop::HwndSourceParameters(
        "hi" // NAME
    );
    sourceParams->PositionX = x;
    sourceParams->PositionY = y;
    sourceParams->Height = height;
    sourceParams->Width = width;
    sourceParams->ParentWindow = IntPtr(parent);
```

```

sourceParams->WindowStyle = WS_VISIBLE | WS_CHILD; // style
System::Windows::Interop::HwndSource^ source = gcnew
System::Windows::Interop::HwndSource(*sourceParams);
WPFPAGE ^myPage = gcnew WPFPAGE(width, height);
//Assign a reference to the WPF page and a set of UI properties to a set
of static properties in a class
//that is designed for that purpose.
WPFPAGEHOST::hostedPage = myPage;
WPFPAGEHOST::initBackBrush = myPage->Background;
WPFPAGEHOST::initFontFamily = myPage->DefaultFontFamily;
WPFPAGEHOST::initFontSize = myPage->DefaultFontSize;
WPFPAGEHOST::initFontStyle = myPage->DefaultFontStyle;
WPFPAGEHOST::initFontWeight = myPage->DefaultFontWeight;
WPFPAGEHOST::initForeBrush = myPage->DefaultForeBrush;
myPage->OnButtonClicked += gcnew
WPFPAGE::ButtonClickHandler(WPFPAGE::OnButtonClicked);
source->RootVisual = myPage;
return (HWND) source->Handle.ToPointer();
}

```

无法直接在应用程序窗口中托管 WPF 内容。从而，首先创建 [HwndSource](#) 对象以包装 WPF 内容。此对象基本上是一个专门用于托管 WPF 内容的窗口。通过将 [HwndSource](#) 对象创建为 Win32 窗口（应用程序的一部分）的子级而将其托管于父窗口中。

[HwndSource](#) 构造函数参数所包含的信息与创建 Win32 子窗口时要传递给 [CreateWindow](#) 的信息基本相同。

接下来，创建 WPF 内容对象的实例。在此情况下，通过使用 C++/CLI，将 WPF 内容作为单独的类 [WPFPAGE](#) 实现。还可以将 WPF 内容作为 DLL 实现 WPF 内容。可以向项目添加对 DLL 的引用，并使用该引用创建 WPF 内容的实例。

通过向 [HwndSource](#) 的 [RootVisual](#) 属性分配一个对 WPF 内容的引用，在子窗口中显示 WPF 内容。

代码的下一行将一个事件处理程序 [WPFPAGE::OnButtonClicked](#) 附加到 WPF 内容 [OnButtonClicked](#) 事件。用户单击“确定”或“取消”按钮时，该处理程序将被调用。有关对此事件处理程序的深入探讨，请参阅[与 WPF 内容通信](#)。

代码显示的最后一行返回与 [HwndSource](#) 对象关联的窗口句柄 (HWND)。可以从 Win32 代码使用此句柄，将消息发送到托管窗口，尽管该示例并未执行这一操作。每当 [HwndSource](#) 对象收到一条消息时就会引发一个事件。若要处理消息，请调用 [AddHook](#) 方法来附加消息处理程序，然后在此处理程序中处理消息。

## 保存对 WPF 内容的引用

对于许多应用程序，你将需要稍后与 WPF 内容进行通信。例如，可能需要修改 WPF 内容属性，或可能让 [HwndSource](#) 对象托管不同的 WPF 内容。执行此操作需要对

`HwndSource` 对象或 WPF 内容的引用。 `HwndSource` 对象及其关联 WPF 内容保留在内存中，直到销毁窗口句柄。但是，一旦从窗口过程返回，分配给 `HwndSource` 对象的变量就将超出范围。用来处理 Win32 应用程序的这一问题的惯用方法是使用静态或全局变量。遗憾的是，你无法向这些类型的变量分配托管对象。可以向全局或静态变量分配与 `HwndSource` 对象关联的窗口句柄，但这一操作不会提供对对象本身的访问。

针对这一问题最简单的解决方案是实现一个托管类，该类包含一组静态字段，这些字段保存对需要访问的任何托管对象的引用。此示例使用 `WPFPAGEHOST` 类来保存对 WPF 内容的引用，以及对该内容某些属性的初始值的引用（用户以后可能会对这些属性进行更改）。这在标头中进行定义。

C++

```
public ref class WPFPAGEHOST
{
public:
    WPFPAGEHOST();
    static WPFPAGE^ hostedPage;
    //initial property settings
    static System::Windows::Media::Brush^ initBackBrush;
    static System::Windows::Media::Brush^ initForeBrush;
    static System::Windows::Media::FontFamily^ initFontFamily;
    static System::Windows::FontStyle initFontSize;
    static System::Windows::FontWeight initFontWeight;
    static double initFontSize;
};
```

`GetHwnd` 函数的后半部分将值分配给这些字段以供以后使用（在 `myPage` 仍处于范围内时）。

## 与 WPF 内容通信

与 UI 的通信有两种类型，UI 使用户可以更改各种 WPF 内容属性，例如背景色或默认字体大小。

如上所述，用户单击任一按钮时，WPF 内容将引发 `OnButtonClicked` 事件。该应用程序将一个处理程序附加到此事件以接收这些通知。如果单击“确定”按钮，该处理程序将从 WPF 内容获得用户信息，并将其显示在一组静态控件中。

C++

```
void WPFPAGECLICKED(Object ^sender, MyPageEventArgs ^args)
{
    if(args->IsOK) //display data if OK button was clicked
    {
        WPFPAGE ^myPage = WPFPAGEHOST::hostedPage;
        LPCWSTR userName = (LPCWSTR)
```

```

    InteropServices::Marshal::StringToHGlobalAuto( "Name: " + myPage-
>EnteredName).ToPointer();
        SetWindowText(nameLabel, userName);
        LPCWSTR userAddress = (LPCWSTR)
    InteropServices::Marshal::StringToHGlobalAuto( "Address: " + myPage-
>EnteredAddress).ToPointer();
        SetWindowText(addressLabel, userAddress);
        LPCWSTR userCity = (LPCWSTR)
    InteropServices::Marshal::StringToHGlobalAuto("City: " + myPage-
>EnteredCity).ToPointer();
        SetWindowText(cityLabel, userCity);
        LPCWSTR userState = (LPCWSTR)
    InteropServices::Marshal::StringToHGlobalAuto("State: " + myPage-
>EnteredState).ToPointer();
        SetWindowText(stateLabel, userState);
        LPCWSTR userZip = (LPCWSTR)
    InteropServices::Marshal::StringToHGlobalAuto("Zip: " + myPage-
>EnteredZip).ToPointer();
        SetWindowText(zipLabel, userZip);
    }
    else
    {
        SetWindowText(nameLabel, L"Name: ");
        SetWindowText(addressLabel, L"Address: ");
        SetWindowText(cityLabel, L"City: ");
        SetWindowText(stateLabel, L"State: ");
        SetWindowText(zipLabel, L"Zip: ");
    }
}

```

该处理程序从 WPF 内容接收到一个自定义事件参数对象 `MyPageEventArgs`。如果单击“确定”按钮，对象的 `IsOK` 属性被设置为 `true`；如果单击“取消”按钮，该属性被设置为 `false`。

如果单击“确定”按钮，该处理程序将从容器类获取对 WPF 内容的引用。然后它会收集由关联的 WPF 内容属性保存的用户信息，并使用静态控件在父窗口上显示该信息。由于 WPF 内容数据的形式为托管字符串，因此必须对其进行封送处理以供 Win32 控件使用。如果单击“取消”按钮，则处理程序将清除静态控件中的数据。

应用程序 UI 提供一组单选按钮，允许用户修改 WPF 内容的背景色和若干与字体相关的属性。下面的示例是应用程序的窗口过程 (WndProc) 的一段摘录及其消息处理功能，通过该功能可设置不同消息上的各种属性，包括背景色。其他内容与此类似，将不进行展示。请查看完整示例了解详细信息和上下文。

C++

```

case WM_COMMAND:
    wmId    = LOWORD(wParam);
    wmEvent = HIWORD(wParam);

```

```

switch (wmId)
{
//Menu selections
    case IDM_ABOUT:
        DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
        break;
    case IDM_EXIT:
        DestroyWindow(hWnd);
        break;
//RadioButtons
    case IDC_ORIGINALBACKGROUND :
        WPFPAGEHOST::hostedPage->Background = WPFPAGEHOST::initBackBrush;
        break;
    case IDC_LIGHTGREENBACKGROUND :
        WPFPAGEHOST::hostedPage->Background = gcnew
SolidColorBrush(Colors::LightGreen);
        break;
    case IDC_LIGHTSALMONBACKGROUND :
        WPFPAGEHOST::hostedPage->Background = gcnew
SolidColorBrush(Colors::LightSalmon);
        break;
}

```

若要设置背景色，请从 `WPFPAGEHOST` 获取对 WPF 内容 (`hostedPage`) 的引用并将背景色属性设置为适当的颜色。此示例使用三种颜色选项：原始颜色、浅绿色或浅橙红色。原始背景色作为静态字段存储在 `WPFPAGEHOST` 类中。若要设置其他两种颜色，请创建一个新的 `SolidColorBrush` 对象，并从 `Colors` 对象向构造函数传递一个静态颜色值。

## 实现 WPF 页

可以托管和使用 WPF 内容而无需任何有关实际实现的知识。如果 WPF 内容已打包到一个单独的 DLL 中，那么它可能是用任何公共语言运行时 (CLR) 语言构建的。以下是在该示例中使用的 C++/CLI 实现的简短演练。本节包含下列子节。

- 布局
- 将数据返回到主机窗口
- 设置 WPF 属性

## Layout

WPF 内容中的 UI 元素由五个 `TextBox` 控件（以及关联的 `Label` 控件）构成：名称、地址、城市、州和邮编。还有两个 `Button` 控件，即“确定”和“取消”

WPF 内容在 `WPFPAGE` 类中实现。布局通过 `Grid` 布局元素进行处理。该类继承自实际使其成为 WPF 内容根元素的 `Grid`。

WPF 内容构造函数采用所需的宽度和高度，并相应地调整 [Grid](#) 的大小。然后，它通过创建一组 [ColumnDefinition](#) 和 [RowDefinition](#) 对象并将它们分别添加到 [Grid](#) 对象基 [ColumnDefinitions](#) 和 [RowDefinitions](#) 集合来定义基本布局。这将定义具有 5 行和 7 列的网格，该网格具有单元内容定义的维度。

C++

```
WPFPage::WPFPage(int allottedWidth, int allottedHeight)
{
    array<ColumnDefinition ^> ^ columnDef = gcnew array<ColumnDefinition ^> (4);
    array<RowDefinition ^> ^ rowDef = gcnew array<RowDefinition ^> (6);

    this->Height = allottedHeight;
    this->Width = allottedWidth;
    this->Background = gcnew SolidColorBrush(Colors::LightGray);

    //Set up the Grid's row and column definitions
    for(int i=0; i<4; i++)
    {
        columnDef[i] = gcnew ColumnDefinition();
        columnDef[i]->Width = GridLength(1, GridUnitType::Auto);
        this->ColumnDefinitions->Add(columnDef[i]);
    }
    for(int i=0; i<6; i++)
    {
        rowDef[i] = gcnew RowDefinition();
        rowDef[i]->Height = GridLength(1, GridUnitType::Auto);
        this->RowDefinitions->Add(rowDef[i]);
    }
}
```

接下来，此构造函数将 UI 元素添加到 [Grid](#)。第一个元素是标题文本，这是一个在网格首行居中的 [Label](#) 控件。

C++

```
//Add the title
titleText = gcnew Label();
titleText->Content = "Simple WPF Control";
titleText->HorizontalAlignment =
System::Windows::HorizontalAlignment::Center;
titleText->Margin = Thickness(10, 5, 10, 0);
titleText->FontWeight = FontWeights::Bold;
titleText->FontSize = 14;
Grid::SetColumn(titleText, 0);
Grid::SetRow(titleText, 0);
Grid::SetColumnSpan(titleText, 4);
this->Children->Add(titleText);
```

下一行包含名称 **Label** 控制及其关联的 **TextBox** 控件。由于对每个标签/文本框对使用同代码，因而该代码被置于一对专用方法内并用于所有五个标签/文本框对。该方法创建相应的控件，并调用 **Grid** 类静态 **SetColumn** 和 **SetRow** 方法，以将控件置于相应单元格内。创建控件后，该示例对 **Add** 的 **Children** 属性调用 **Grid** 方法，以便将控件添加到网格。用于添加剩余标签/文本框对的代码相似。请参阅示例代码了解详细信息。

C++

```
//Add the Name Label and TextBox
nameLabel = CreateLabel(0, 1, "Name");
this->Children->Add(nameLabel);
nameTextBox = CreateTextBox(1, 1, 3);
this->Children->Add(nameTextBox);
```

这两种方法的实现如下所示：

C++

```
Label ^WPFPage::CreateLabel(int column, int row, String ^ text)
{
    Label ^ newLabel = gcnew Label();
    newLabel->Content = text;
    newLabel->Margin = Thickness(10, 5, 10, 0);
    newLabel->FontWeight = FontWeights::Normal;
    newLabel->FontSize = 12;
    Grid::SetColumn(newLabel, column);
    Grid::SetRow(newLabel, row);
    return newLabel;
}
TextBox ^WPFPage::CreateTextBox(int column, int row, int span)
{
    TextBox ^newTextBox = gcnew TextBox();
    newTextBox->Margin = Thickness(10, 5, 10, 0);
    Grid::SetColumn(newTextBox, column);
    Grid::SetRow(newTextBox, row);
    Grid::SetColumnSpan(newTextBox, span);
    return newTextBox;
}
```

最后，该示例添加“确定”和“取消”按钮，然后将一个事件处理程序附加到它们的 **Click** 事件。

C++

```
//Add the Buttons and attach event handlers
okButton = CreateButton(0, 5, "OK");
cancelButton = CreateButton(1, 5, "Cancel");
this->Children->Add(okButton);
this->Children->Add(cancelButton);
```

```
okButton->Click += gcnew RoutedEventHandler(this, &WPFPAGE::ButtonClicked);
cancelButton->Click += gcnew RoutedEventHandler(this,
&WPFPAGE::ButtonClicked);
```

## 将数据返回到主机窗口

单击任一按钮后，将引发其 `Click` 事件。 主机窗口只需将处理程序附加到这些事件，然后直接从 `TextBox` 控件获取数据。 此示例使用的是不太直接的方法。 它处理 WPF 内容内的 `Click`，然后引发自定义事件 `OnButtonClicked` 来通知 WPF 内容。 这使 WPF 内容能够在通知主机之前执行某些参数验证。 该处理程序从 `TextBox` 控件获取文本，然后将其分配给公共属性（主机可从其中检索信息）。

WPFPAGE.h 中的事件声明：

C++

```
public:
    delegate void ButtonClickHandler(Object ^, MyPageEventArgs ^);
    WPFPAGE();
    WPFPAGE(int height, int width);
    event ButtonClickHandler ^OnButtonClicked;
```

WPFPAGE.cpp 中的 `Click` 事件处理程序：

C++

```
void WPFPAGE::ButtonClicked(Object ^sender, RoutedEventArgs ^args)
{
    //TODO: validate input data
    bool okClicked = true;
    if(sender == cancelButton)
        okClicked = false;
    EnteredName = nameTextBox->Text;
    EnteredAddress = addressTextBox->Text;
    EnteredCity = cityTextBox->Text;
    EnteredState = stateTextBox->Text;
    EnteredZip = zipTextBox->Text;
    OnButtonClicked(this, gcnew MyPageEventArgs(okClicked));
}
```

## 设置 WPF 属性

Win32 主机使用户可以更改若干 WPF 内容属性。 就 Win32 端而言，这只是更改属性的问题。 WPF 内容类中的实现从某种意义上来说更加复杂，因为不存在控制所有控件的字体的单个全局属性。 相反，每个控件的相应属性均是在属性的 set 访问器中进行更改

的。以下示例显示 `DefaultFontFamily` 属性的代码。设置属性将调用专用方法，该方法依次又将为各控件设置 `FontFamily` 属性。

从 `WPFFPage.h`：

```
C++  
  
property FontFamily^ DefaultFontFamily  
{  
    FontFamily^ get() {return _defaultFontFamily;}  
    void set(FontFamily^ value) {SetFontFamily(value);}  
};
```

从 `WPFFPage.cpp`：

```
C++  
  
void WPFFPage::SetFontFamily(FontFamily^ newFontFamily)  
{  
    _defaultFontFamily = newFontFamily;  
    titleText->FontFamily = newFontFamily;  
    nameLabel->FontFamily = newFontFamily;  
    addressLabel->FontFamily = newFontFamily;  
    cityLabel->FontFamily = newFontFamily;  
    stateLabel->FontFamily = newFontFamily;  
    zipLabel->FontFamily = newFontFamily;  
}
```

## 另请参阅

- [HwndSource](#)
- [WPF 和 Win32 互操作](#)

# 演练：在 Win32 中托管 WPF 时钟

项目 • 2022/09/27

若要将 WPF 置于 Win32 应用程序内，请使用 [HwndSource](#)，它提供包含 WPF 内容的 HWND。首先创建 [HwndSource](#)，为其赋予类似于 CreateWindow 的参数。然后，告知 [HwndSource](#) 你希望置于其中的 WPF 内容。最后，从 [HwndSource](#) 中获取 HWND。本演练说明如何在重新实现操作系统“日期和时间属性”对话框的 Win32 应用程序内创建混合 WPF。

## 先决条件

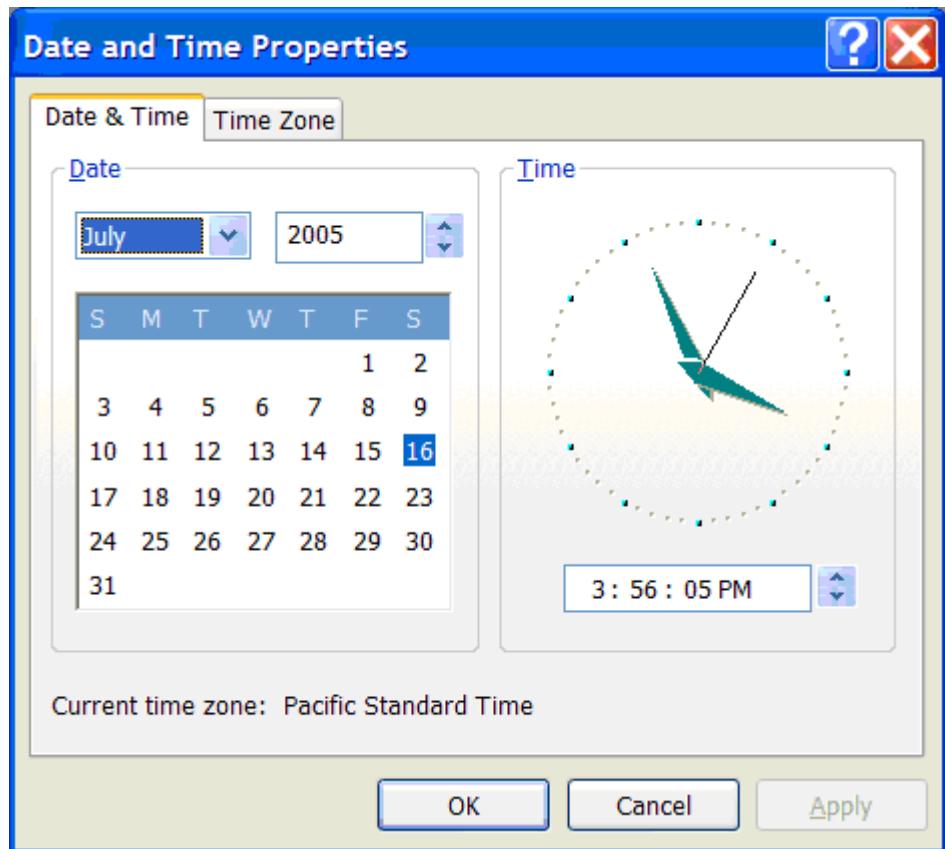
请参阅 [WPF 和 Win32 互操作](#)。

## 如何使用本教程

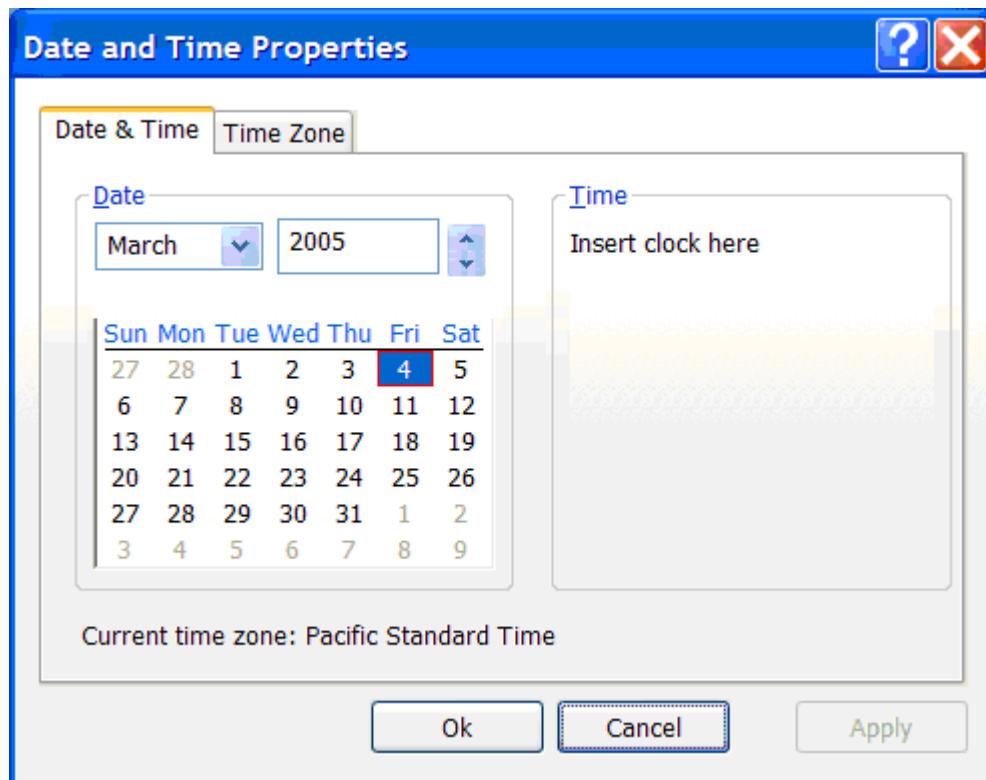
本教程重点介绍生成互操作应用程序的重要步骤。本教程采用 [Win32 时钟互操作示例](#) 示例，但该示例反映了最终产品。本教程将介绍如何开始使用现有的 Win32 项目（也许是一个预先存在的项目）并向应用程序添加承载的 WPF。可以将最终产品与 [Win32 时钟互操作示例](#) 进行比较。

## Win32 中的 Windows Presentation Framework 的演练 (HwndSource)

下图显示了本教程的预期最终产品：



可以通过在 Visual Studio 中创建 C++ Win32 项目来重新创建此对话框，并使用对话框编辑器创建以下内容：



( 不需要使用 Visual Studio 来使用 `HwndSource`，不需要使用 C++ 编写 Win32 程序，但这是执行此操作的一种相当典型的方法，并且非常适合用于分布教程说明 )。

需要实现五个特定的子步骤才能将 WPF 时钟放入对话框中：

1. 通过更改 Visual Studio 中的项目设置，使你的 Win32 项目能够调用托管代码 (/clr)。
2. 在单独的 DLL 中创建 WPF[Page](#)。
3. 将 WPF[Page](#) 放在 [HwndSource](#) 中。
4. 使用 [Handle](#) 属性获取该 [Page](#) 的 HWND。
5. 使用 Win32 决定在较大的 Win32 应用程序中放置 HWND 的位置

## /clr

第一步是将此非托管 Win32 项目转换成一个可以调用托管代码的项目。使用 /clr 编译器选项，它将链接到要使用的所需 DLL，然后调整 Main 方法以便与 WPF 一起使用。

若要实现在 C++ 项目中使用托管代码：右键单击 win32clock 项目，然后选择“属性”。在“常规”属性页（默认值）上，将公共语言运行时支持更改为 `/clr`。

接下来，添加 WPF 所需的对 DLL 的引用：PresentationCore.dll、PresentationFramework.dll、System.dll、WindowsBase.dll、UIAutomationProvider.dll 和 UIAutomationTypes.dll。（以下说明假定操作系统安装在 C: 驱动器上。）

1. 右键单击 win32clock 项目，然后选择“引用...”，并在该对话框中：
2. 右键单击 win32clock 项目，然后选择“引用...”。
3. 依次单击“添加新引用”、“浏览”选项卡，输入 `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationCore.dll`，然后单击“确定”。
4. 对 PresentationFramework.dll 重复相同步骤：`C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationFramework.dll`。
5. 对 WindowsBase.dll 重复相同步骤：`C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\WindowsBase.dll`。
6. 对 UIAutomationTypes.dll 重复相同步骤：`C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\UIAutomationTypes.dll`。
7. 对 UIAutomationProvider.dll 重复相同步骤：`C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\UIAutomationProvider.dll`。
8. 单击“添加新引用”，选择 System.dll，然后单击“确定”。
9. 单击“确定”退出用于添加引用的 win32clock 属性页。

最后，将 `STAThreadAttribute` 添加到 `_tWinMain` 方法以便与 WPF 一起使用：

C++

```
[System::STAThreadAttribute]
int APIENTRY _tWinMain(HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPTSTR    lpCmdLine,
                        int       nCmdShow)
```

此属性告诉公共语言运行时 (CLR)，当它初始化组件对象模型 (COM) 时，它应使用单线程单元模型 (STA)，这是 WPF ( 和 Windows 窗体 ) 所必需的。

## 创建 Windows Presentation Framework 页

接下来，创建一个定义 `WPFPage` 的 DLL。将 `WPFPage` 创建为独立应用程序，并以这种方式编写和调试 WPF 部分通常是最简单的方法。完成后，通过右键单击该项目，单击“属性”转到“应用程序”，并将输出类型更改为 Windows 类库，可以将该项目转换为 DLL。

然后，可以将 WPF dll 项目与 Win32 项目（包含两个项目的一个解决方案）组合 - 右键单击解决方案，选择“添加\现有项目”。

若要使用 Win32 项目中的 WPF dll，你需要添加引用：

1. 右键单击 `win32clock` 项目，然后选择“引用...”。
2. 单击“添加新引用”。
3. 单击“项目”选项卡。选择 `WPFClock`，单击“确定”。
4. 单击“确定”退出用于添加引用的 `win32clock` 属性页。

## HwndSource

接下来，使用 `HwndSource` 使 `WPFPage` 看起来像 HWND。将此代码块添加到 C++ 文件中：

C++

```
namespace ManagedCode
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Media;
```

```

HWND GetHwnd(HWND parent, int x, int y, int width, int height) {
    HwndSource^ source = gcnew HwndSource(
        0, // class style
        WS_VISIBLE | WS_CHILD, // style
        0, // exstyle
        x, y, width, height,
        "hi", // NAME
        IntPtr(parent) // parent window
    );

    UIElement^ page = gcnew WPFClock::Clock();
    source->RootVisual = page;
    return (HWND) source->Handle.ToPointer();
}
}
}

```

这是一长段代码，可以作一些解释。第一部分是各种子句，无需完全限定所有调用：

C++

```

namespace ManagedCode
{
    using namespace System;
    using namespace System::Windows;
    using namespace System::Windows::Interop;
    using namespace System::Windows::Media;

```

然后，定义一个创建 WPF 内容的函数，在其周围放置一个 `HwndSource`，并返回 `HWND`：

C++

```

HWND GetHwnd(HWND parent, int x, int y, int width, int height) {

```

首先创建 `HwndSource`，其参数类似于 `CreateWindow`：

C++

```

HwndSource^ source = gcnew HwndSource(
    0, // class style
    WS_VISIBLE | WS_CHILD, // style
    0, // exstyle
    x, y, width, height,
    "hi", // NAME
    IntPtr(parent) // parent window
);

```

然后通过调用其构造函数创建 WPF 内容类：

C++

```
UIElement^ page = gcnew WPFClock::Clock();
```

然后将页面连接到 [HwndSource](#)：

C++

```
source->RootVisual = page;
```

在最后一行中，返回 [HwndSource](#) 的 HWND：

C++

```
return (HWND) source->Handle.ToPointer();
```

## 放置 Hwnd

现在你拥有包含 WPF 时钟的 HWND，需要将该 HWND 放在 Win32 对话框中。如果你知道在何处放置 HWND，则只需将该大小和位置传递给之前定义的 [GetHwnd](#) 函数。但是，由于已使用资源文件来定义对话框，因此你不完全确定任何 HWND 的放置位置。你可以使用 Visual Studio 对话框编辑器将 Win32 STATIC 控件放在希望放置时钟的位置（“在此处插入时钟”），并使用它来定位 WPF 时钟。

在处理 WM\_INITDIALOG 的地方，你可以使用 [GetDlgItem](#) 检索占位符 STATIC 的 HWND：

C++

```
HWND placeholder = GetDlgItem(hDlg, IDC_CLOCK);
```

然后，计算该占位符 STATIC 的大小和位置，以便可以将 WPF 时钟放置在该位置中：

RECT 矩形；

C++

```
GetWindowRect	placeholder, &rectangle);
int width = rectangle.right - rectangle.left;
int height = rectangle.bottom - rectangle.top;
POINT point;
point.x = rectangle.left;
```

```
point.y = rectangle.top;
result = MapWindowPoints(NULL, hDlg, &point, 1);
```

然后，你隐藏占位符 STATIC：

C++

```
ShowWindow	placeholder, SW_HIDE);
```

在该位置创建 WPF 时钟 HWND：

C++

```
HWND clock = ManagedCode::GetHwnd(hDlg, point.x, point.y, width, height);
```

为了使教程变得有趣，并且生成真正的 WPF 时钟，你将需要在此时创建 WPF 时钟控件。你可以在标记中执行大部分操作，代码隐藏中只有几个事件处理程序。由于本教程是关于互操作，而不是关于控件设计的，所以在此处提供 WPF 时钟的完整代码以作为代码块，无需另外说明如何构建以及每个部分的含义。随意尝试此代码来更改控件的外观或功能。

此处为标记：

XAML

```
<Page x:Class="WPFClock.Clock"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    >
    <Grid>
        <Grid.Background>
            <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                <GradientStop Color="#fcfcfe" Offset="0" />
                <GradientStop Color="#f6f4f0" Offset="1.0" />
            </LinearGradientBrush>
        </Grid.Background>

        <Grid Name="PodClock" HorizontalAlignment="Center"
VerticalAlignment="Center">
            <Grid.Resources>
                <Storyboard x:Key="sb">
                    <DoubleAnimation From="0" To="360" Duration="12:00:00"
RepeatBehavior="Forever"
                        Storyboard.TargetName="HourHand"
                        Storyboard.TargetProperty="
(Rectangle.RenderTransform).(RotateTransform.Angle)" />
                    <DoubleAnimation From="0" To="360" Duration="01:00:00"
RepeatBehavior="Forever"
```

```

        Storyboard.TargetName="MinuteHand"
        Storyboard.TargetProperty="
    (Rectangle.RenderTransform).(RotateTransform.Angle)"
    />
    <DoubleAnimation From="0" To="360" Duration="0:1:00"
RepeatBehavior="Forever"
        Storyboard.TargetName="SecondHand"
        Storyboard.TargetProperty="
    (Rectangle.RenderTransform).(RotateTransform.Angle)"
    />
</Storyboard>
</Grid.Resources>

<Ellipse Width="108" Height="108" StrokeThickness="3">
    <Ellipse.Stroke>
        <LinearGradientBrush>
            <GradientStop Color="LightBlue" Offset="0" />
            <GradientStop Color="DarkBlue" Offset="1" />
        </LinearGradientBrush>
    </Ellipse.Stroke>
</Ellipse>
<Ellipse VerticalAlignment="Center" HorizontalAlignment="Center"
Width="104" Height="104" Fill="LightBlue" StrokeThickness="3">
    <Ellipse.Stroke>
        <LinearGradientBrush>
            <GradientStop Color="DarkBlue" Offset="0" />
            <GradientStop Color="LightBlue" Offset="1" />
        </LinearGradientBrush>
    </Ellipse.Stroke>
</Ellipse>
<Border BorderThickness="1" BorderBrush="Black"
Background="White" Margin="20" HorizontalAlignment="Right"
VerticalAlignment="Center">
    <TextBlock Name="MonthDay" Text="{Binding}" />
</Border>
<Canvas Width="102" Height="102">
    <Ellipse Width="8" Height="8" Fill="Black" Canvas.Top="46"
Canvas.Left="46" />
    <Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
Width="4" Height="8">
        <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46" Angle="0"
/>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
        <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46" Angle="30"
/>
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
        <Rectangle.RenderTransform>

```

```
        <RotateTransform CenterX="2" CenterY="46" Angle="60"
/>
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
Width="4" Height="8">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46" Angle="90"
/>
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="120" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="150" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
Width="4" Height="8">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="180" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="210" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="240" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="48" Fill="Black"
Width="4" Height="8">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="2" CenterY="46"
Angle="270" />
    </Rectangle.RenderTransform>
</Rectangle>
<Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
```

```

        <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46"
Angle="300" />
        </Rectangle.RenderTransform>
    </Rectangle>
    <Rectangle Canvas.Top="5" Canvas.Left="49" Fill="Black"
Width="2" Height="6">
        <Rectangle.RenderTransform>
            <RotateTransform CenterX="2" CenterY="46"
Angle="330" />
        </Rectangle.RenderTransform>
    </Rectangle>

    <Rectangle x:Name="HourHand" Canvas.Top="21"
Canvas.Left="48"
Fill="Black" Width="4" Height="30">
    <Rectangle.RenderTransform>
        <RotateTransform x:Name="HourHand2" CenterX="2"
CenterY="30" />
    </Rectangle.RenderTransform>
</Rectangle>
    <Rectangle x:Name="MinuteHand" Canvas.Top="6"
Canvas.Left="49"
Fill="Black" Width="2" Height="45">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="1" CenterY="45" />
    </Rectangle.RenderTransform>
</Rectangle>
    <Rectangle x:Name="SecondHand" Canvas.Top="4"
Canvas.Left="49"
Fill="Red" Width="1" Height="47">
    <Rectangle.RenderTransform>
        <RotateTransform CenterX="0.5" CenterY="47" />
    </Rectangle.RenderTransform>
</Rectangle>
</Canvas>
</Grid>
</Grid>
</Page>

```

以下是附带的代码隐藏：

```

C#
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Shapes;

```

```

using System.Windows.Threading;

namespace WPFClock
{
    /// <summary>
    /// Interaction logic for Clock.xaml
    /// </summary>
    public partial class Clock : Page
    {
        private DispatcherTimer _dayTimer;

        public Clock()
        {
            InitializeComponent();
            this.Loaded += new RoutedEventHandler(Clock_Loaded);
        }

        void Clock_Loaded(object sender, RoutedEventArgs e) {
            // set the datacontext to be today's date
            DateTime now = DateTime.Now;
            DataContext = now.Day.ToString();

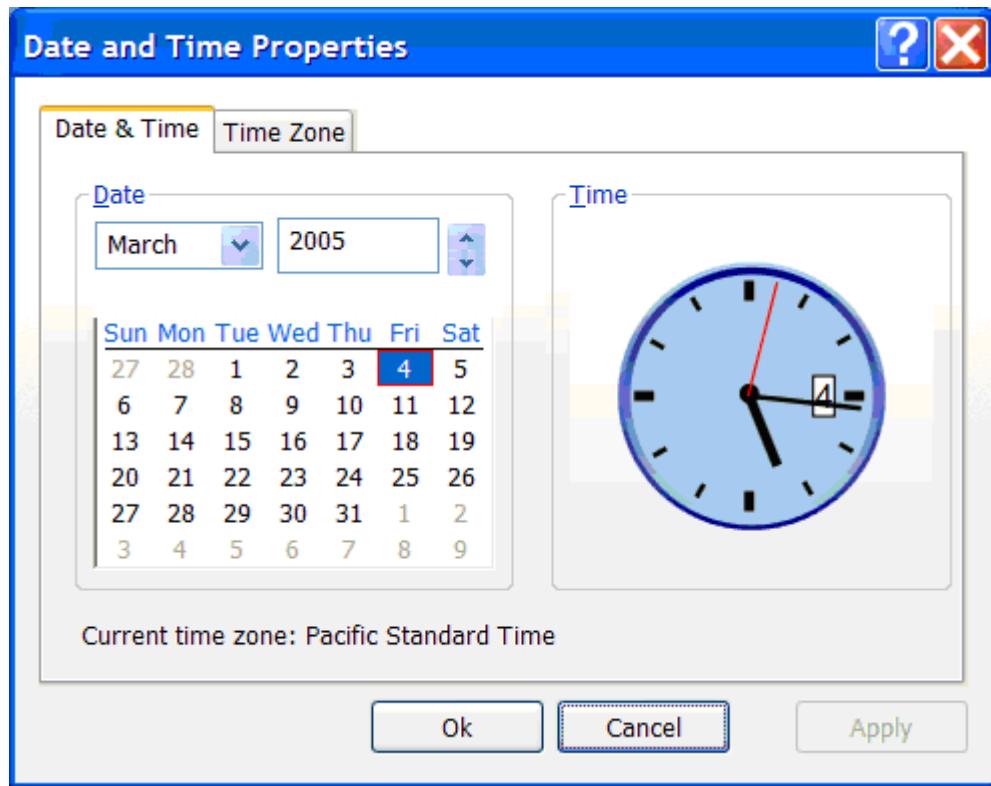
            // then set up a timer to fire at the start of tomorrow, so that
            // we can update
            // the datacontext
            _dayTimer = new DispatcherTimer();
            _dayTimer.Interval = new TimeSpan(1, 0, 0, 0) - now.TimeOfDay;
            _dayTimer.Tick += new EventHandler(OnDayChange);
            _dayTimer.Start();

            // finally, seek the timeline, which assumes a beginning at
            // midnight, to the appropriate
            // offset
            Storyboard sb = (Storyboard)PodClock.FindResource("sb");
            sb.Begin(PodClock, HandoffBehavior.SnapshotAndReplace, true);
            sb.Seek(PodClock, now.TimeOfDay, TimeSeekOrigin.BeginTime);
        }

        private void OnDayChange(object sender, EventArgs e)
        {
            // date has changed, update the datacontext to reflect today's
            // date
            DateTime now = DateTime.Now;
            DataContext = now.Day.ToString();
            _dayTimer.Interval = new TimeSpan(1, 0, 0, 0);
        }
    }
}

```

最终结果如下所示：



若要将最终结果与生成此屏幕快照的代码进行比较，请参阅 [Win32 时钟互操作示例](#)。

## 另请参阅

- [HwndSource](#)
- [WPF 和 Win32 互操作](#)
- [Win32 时钟互操作示例](#)

# WPF 和 Direct3D9 互操作

项目 • 2023/02/06

你可以在 Windows Presentation Foundation (WPF) 应用程序中包含 Direct3D9 内容。本主题介绍如何创建 Direct3D9 内容，以便它有效地与 WPF 互操作。

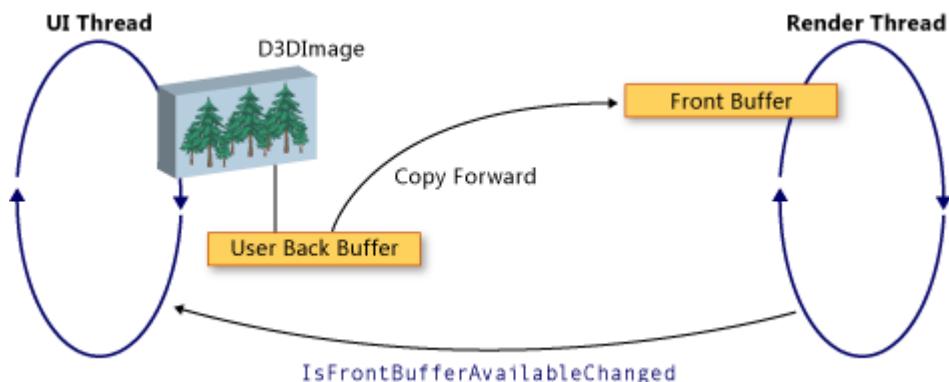
## ① 备注

在 WPF 中使用 Direct3D9 内容时，还需要考虑性能问题。有关如何优化性能的详细信息，请参阅 [Direct3D9 和 WPF 互操作性的性能注意事项](#)。

## 显示缓冲区

`D3DImage` 类管理两个显示缓冲区，称为后台缓冲区和前台缓冲区。后台缓冲区是 Direct3D9 图面。调用 [Unlock](#) 方法时，对后台缓冲区的更改会向前复制到前台缓冲区。

下图显示了后台缓冲区和前台缓冲区之间的关系。



## Direct3D9 设备创建

若要呈现 Direct3D9 内容，必须创建 Direct3D9 设备。有两个 Direct3D9 对象可用于创建设备，即 `IDirect3D9` 和 `IDirect3D9Ex`。可使用这些对象分别创建 `IDirect3DDevice9` 和 `IDirect3DDevice9Ex` 设备。

通过调用以下方法之一创建设备。

- `IDirect3D9 * Direct3DCreate9(UINT SDKVersion);`
- `HRESULT Direct3DCreate9Ex(UINT SDKVersion, IDirect3D9Ex **ppD3D);`

在 Windows Vista 或更高版本的操作系统上，对配置为使用 Windows 显示驱动程序模型 (WDDM) 的显示器使用 `Direct3DCREATE9EX` 方法。对任何其他平台使用 `Direct3DCREATE9` 方法。

## Direct3DCREATE9EX 方法的可用性

d3d9.dll 仅在 Windows Vista 或更高版本的操作系统上提供 `Direct3DCREATE9EX` 方法。如果在 Windows XP 上直接链接该函数，应用程序将无法加载。若要确定是否支持 `Direct3DCREATE9EX` 方法，请加载 DLL 并查找 proc 地址。以下代码演示如何测试 `Direct3DCREATE9EX` 方法。有关完整代码示例，请参阅[演练：创建在 WPF 中承载的 Direct3D9 内容](#)。

C++

```
HRESULT
CRendererManager::EnsureD3DObjects()
{
    HRESULT hr = S_OK;

    HMODULE hD3D = NULL;
    if (!m_pD3D)
    {
        hD3D = LoadLibrary(TEXT("d3d9.dll"));
        DIRECT3DCREATE9EXFUNCTION pfnCreate9Ex =
(DIRECT3DCREATE9EXFUNCTION)GetProcAddress(hD3D, "Direct3DCREATE9EX");
        if (pfnCreate9Ex)
        {
            IFC((*pfnCreate9Ex)(D3D_SDK_VERSION, &m_pD3DE));
            IFC(m_pD3DE->QueryInterface(__uuidof(IDirect3D9),
reinterpret_cast<void **>(&m_pD3D)));
        }
        else
        {
            m_pD3D = Direct3DCREATE9(D3D_SDK_VERSION);
            if (!m_pD3D)
            {
                IFC(E_FAIL);
            }
        }
    }

    m_cAdapters = m_pD3D->GetAdapterCount();
}

Cleanup:
if (hD3D)
{
    FreeLibrary(hD3D);
}
```

```
    return hr;
}
```

## HWND 创建

创建设备需要 HWND。通常，需要创建一个虚拟 HWND 供 Direct3D9 使用。以下代码示例演示如何创建虚拟 HWND。

C++

```
HRESULT CRendererManager::EnsureHwnd()
{
    HRESULT hr = S_OK;

    if (!m_hwnd)
    {
        WNDCLASS wndclass;

        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfWndProc = DefWindowProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = NULL;
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = szAppName;

        if (!RegisterClass(&wndclass))
        {
            IFC(E_FAIL);
        }

        m_hwnd = CreateWindow(szAppName,
                              TEXT("D3DImageSample"),
                              WS_OVERLAPPEDWINDOW,
                              0, // Initial X
                              0, // Initial Y
                              0, // Width
                              0, // Height
                              NULL,
                              NULL,
                              NULL,
                              NULL);
    }

    Cleanup:
    return hr;
}
```

## 呈现参数

创建设备还需要有 `D3DPRESENT_PARAMETERS` 结构，但只有几个参数很重要。选择这些参数是为了最小化内存占用。

将 `BackBufferHeight` 和 `BackBufferWidth` 字段设置为 1。将其设置为 0 会导致其设置为 `HWND` 的维度。

始终设置 `D3DCREATE_MULTITHREADED` 和 `D3DCREATE_FPU_PRESERVE` 标记，以防止损坏 Direct3D9 所用的内存，并防止 Direct3D9 更改 FPU 设置。

以下代码演示如何初始化 `D3DPRESENT_PARAMETERS` 结构。

C++

```
HRESULT
CRenderer::Init(IDirect3D *pD3D, IDirect3DEx *pD3DEx, HWND hwnd, UINT uAdapter)
{
    HRESULT hr = S_OK;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
    d3dpp.BackBufferHeight = 1;
    d3dpp.BackBufferWidth = 1;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;

    D3DCAPS9 caps;
    DWORD dwVertexProcessing;
    IFC(pD3D->GetDeviceCaps(uAdapter, D3DDEVTYPE_HAL, &caps));
    if ((caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) ==
        D3DDEVCAPS_HWTRANSFORMANDLIGHT)
    {
        dwVertexProcessing = D3DCREATE_HARDWARE_VERTEXPROCESSING;
    }
    else
    {
        dwVertexProcessing = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
    }

    if (pD3DEx)
    {
        IDirect3DDevice9Ex *pd3dDevice = NULL;
        IFC(pD3DEx->CreateDeviceEx(
            uAdapter,
            D3DDEVTYPE_HAL,
            hwnd,
            dwVertexProcessing | D3DCREATE_MULTITHREADED |
            D3DCREATE_FPU_PRESERVE,
```

```

        &d3dpp,
        NULL,
        &m_pd3dDeviceEx
    ));

    IFC(m_pd3dDeviceEx->QueryInterface(__uuidof(IDirect3DDevice9),
reinterpret_cast<void**>(&m_pd3dDevice)));
}
else
{
    assert(pD3D);

    IFC(pD3D->CreateDevice(
        uAdapter,
        D3DDEVTYPE_HAL,
        hwnd,
        dwVertexProcessing | D3DCREATE_MULTITHREADED |
D3DCREATE_FPU_PRESERVE,
        &d3dpp,
        &m_pd3dDevice
    ));
}

Cleanup:
    return hr;
}

```

## 创建后台缓冲区呈现目标

若要在 [D3DImage](#) 中显示 Direct3D9 内容，请创建一个 Direct3D9 图面并通过调用 [SetBackBuffer](#) 方法来分配它。

## 验证适配器支持

在创建图面之前，请验证所有适配器是否都支持所需的图面属性。即使只呈现到一个适配器，WPF 窗口也可能显示在系统中的任何适配器上。应始终编写处理多适配器配置的 Direct3D9 代码，并且应检查所有适配器是否提供支持，因为 WPF 可能在可用适配器之间移动图面。

以下代码示例演示如何检查系统上的所有适配器是否支持 Direct3D9。

C++

```

HRESULT
CRendererManager::TestSurfaceSettings()
{
    HRESULT hr = S_OK;

    D3DFORMAT fmt = m_fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8;

```

```

//  

// We test all adapters because we potentially use all adapters.  

// But even if this sample only rendered to the default adapter, you  

// should check all adapters because WPF may move your surface to  

// another adapter for you!  

//  

for (UINT i = 0; i < m_cAdapters; ++i)  

{  

    // Can we get HW rendering?  

    IFC(m_pD3D->CheckDeviceType(  

        i,  

        D3DDEVTYPE_HAL,  

        D3DFMT_X8R8G8B8,  

        fmt,  

        TRUE  

    ));  

    // Is the format okay?  

    IFC(m_pD3D->CheckDeviceFormat(  

        i,  

        D3DDEVTYPE_HAL,  

        D3DFMT_X8R8G8B8,  

        D3DUSAGE_RENDERTARGET | D3DUSAGE_DYNAMIC, // We'll use dynamic  

when on XP  

        D3DRTYPE_SURFACE,  

        fmt  

    ));  

    // D3DImage only allows multisampling on 9Ex devices. If we can't  

    // multisample, overwrite the desired number of samples with 0.  

    if (m_pD3DEx && m_uNumSamples > 1)  

    {  

        assert(m_uNumSamples <= 16);  

        if (FAILED(m_pD3D->CheckDeviceMultiSampleType(  

            i,  

            D3DDEVTYPE_HAL,  

            fmt,  

            TRUE,  

            static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),  

            NULL  

        )))  

        {  

            m_uNumSamples = 0;  

        }
    }
else
{
    m_uNumSamples = 0;
}
}
}


```

Cleanup:

```
    return hr;
}
```

## 创建图面

在创建图面之前，请验证设备功能是否支持目标操作系统上的良好性能。有关详细信息，请参阅 [Direct3D9 和 WPF 互操作性的性能注意事项](#)。

验证设备功能后，即可创建图面。以下代码示例演示如何创建呈现目标。

C++

```
HRESULT CRenderer::CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha, UINT m_uNumSamples)
{
    HRESULT hr = S_OK;

    SAFE_RELEASE(m_pd3dRTS);

    IFC(m_pd3dDevice->CreateRenderTarget(
        uWidth,
        uHeight,
        fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8,
        static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
        0,
        m_pd3dDeviceEx ? FALSE : TRUE, // Lockable RT required for good XP
perf
        &m_pd3dRTS,
        NULL
    ));

    IFC(m_pd3dDevice->SetRenderTarget(0, m_pd3dRTS));

    Cleanup:
    return hr;
}
```

## WDDM

在配置为使用 WDDM 的 Windows Vista 及更高版本的操作系统上，可以创建呈现目标纹理并将 0 级图面传递给 [SetBackBuffer](#) 方法。不建议在 Windows XP 上使用此方法，因为你无法创建可锁定的呈现目标纹理，并且性能会降低。

## 处理设备状态

`D3DImage` 类管理两个显示缓冲区，称为后台缓冲区和前台缓冲区。后台缓冲区是 Direct3D 图面。调用 `Unlock` 方法时，对后台缓冲区的更改会向前复制到前台缓冲区，并在硬件上显示。有时，前台缓冲区会变得不可用。导致其不可用的原因可能包括屏幕锁定、全屏独占 Direct3D 应用程序、用户切换或其他系统活动。发生这种情况时，WPF 应用程序会通过处理 `IsFrontBufferAvailableChanged` 事件收到通知。应用程序如何响应变得不可用的前台缓冲区取决于 WPF 是否能够回退到软件呈现。`SetBackBuffer` 方法有一个重载，该重载使用的参数指定了 WPF 是否回退到软件呈现。

调用 `SetBackBuffer(D3DResourceType, IntPtr)` 重载或调用

`SetBackBuffer(D3DResourceType, IntPtr, Boolean)` 重载并将 `enableSoftwareFallback` 参数设置为 `false` 时，呈现系统会在前台缓冲区不可用且不显示任何内容时释放其对后台缓冲区的引用。当前台缓冲区再次可用时，呈现系统会引发 `IsFrontBufferAvailableChanged` 事件以通知 WPF 应用程序。你可以为 `IsFrontBufferAvailableChanged` 事件创建事件处理程序，以使用有效的 Direct3D 图面重新开始呈现。若要重新开始呈现，必须调用 `SetBackBuffer`。

调用 `SetBackBuffer(D3DResourceType, IntPtr, Boolean)` 重载并将

`enableSoftwareFallback` 参数设置为 `true` 时，呈现系统会在前台缓冲区不可用时保留其对后台缓冲区的引用，因此当前台缓冲区再次可用时无需调用 `SetBackBuffer`。

启用软件呈现后，可能会出现用户设备不可用的情况，但呈现系统会保留对 Direct3D 图面的引用。若要检查 Direct3D9 设备是否不可用，请调用 `TestCooperativeLevel` 方法。

若要检查 Direct3D9Ex 设备，请调用 `CheckDeviceState` 方法，因为

`TestCooperativeLevel` 方法已被弃用，并且始终返回 `success`。如果用户设备不可用，请调用 `SetBackBuffer` 以释放 WPF 对后台缓冲区的引用。如果需要重置设备，请调用 `SetBackBuffer` 并将 `backBuffer` 参数设置为 `null`，然后再次调用 `SetBackBuffer` 并将 `backBuffer` 设置为有效的 Direct3D 图面。

仅当实现多适配器支持时，才调用 `Reset` 方法从无效设备中恢复。否则，释放所有 Direct3D9 接口并完全重新创建。如果适配器布局已更改，则在更改之前创建的 Direct3D9 对象不会更新。

## 处理调整大小

如果 `D3DImage` 以其本机大小以外的分辨率显示，则会根据当前 `BitmapScalingMode` 对其进行缩放，但 `Fant` 会被替换为 `Bilinear`。

如果需要更高的保真度，则必须在 `D3DImage` 的容器更改大小时创建一个新图面。

有三种可能的方法来处理调整大小。

- 参与布局系统，并在大小发生变化时创建新的图面。不要创建太多图面，因为这样可能会耗尽视频内存或使内存碎片化。
- 等到固定时间段内未发生调整大小事件时，再创建新的图面。
- 创建一个 `DispatcherTimer`，用于每秒检查几次容器维度。

## 多显示器优化

当呈现系统将 `D3DImage` 移动到另一个显示器时，性能会显著降低。

在 WDDM 上，只要显示器在同一个视频卡上并且你使用 `IDirect3DCreate9Ex`，性能就不会降低。如果显示器位于不同的视频卡上，性能会降低。在 Windows XP 上，性能始终会降低。

当 `D3DImage` 移动到另一个显示器时，可以在相应的适配器上创建新图面，以恢复良好的性能。

为避免性能损失，可专门为多显示器编写代码。以下列表介绍了一种编写多显示器代码的方法。

1. 使用 `Visual.ProjectToScreen` 方法在屏幕空间中查找 `D3DImage` 的一个点。
2. 使用 `MonitorFromPoint` GDI 方法查找正在显示该点的显示器。
3. 使用 `IDirect3D9::GetAdapterMonitor` 方法查找显示器所在的 Direct3D9 适配器。
4. 如果适配器与具有后台缓冲区的适配器不同，则在新显示器上创建一个新的后台缓冲区，并将其分配给 `D3DImage` 后台缓冲区。

### ① 备注

如果 `D3DImage` 跨显示器，性能会很慢，除非 WDDM 和 `IDirect3D9Ex` 在同一个适配器上。在这种情况下，无法提高性能。

以下代码示例演示如何查找当前显示器。

C++

```
void
CRendererManager::SetAdapter(POINT screenSpacePoint)
{
    CleanupInvalidDevices();

    //
```

```

    // After CleanupInvalidDevices, we may not have any D3D objects. Rather
    // than
    // recreate them here, ignore the adapter update and wait for render to
    // recreate.
    //

    if (m_pD3D && m_rgRenderers)
    {
        HMONITOR hMon = MonitorFromPoint(screenSpacePoint,
MONITOR_DEFAULTTONULL);

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            if (hMon == m_pD3D->GetAdapterMonitor(i))
            {
                m_pCurrentRenderer = m_rgRenderers[i];
                break;
            }
        }
    }
}

```

当 `D3DImage` 容器的大小或位置发生变化时更新显示器，或者使用每秒更新几次的 `DispatcherTimer` 更新显示器。

## WPF 软件呈现

在以下情况下，WPF 会在软件中的 UI 线程上同步呈现。

- 打印
- `BitmapEffect`
- `RenderTargetBitmap`

出现其中一种情况时，呈现系统会调用 `CopyBackBuffer` 方法将硬件缓冲区复制到软件。默认实现使用图面调用 `GetRenderTargetData` 方法。此调用发生在锁定/解锁模式之外，因此它可能会失败。在这种情况下，`CopyBackBuffer` 方法会返回 `null` 并且不显示任何图像。

你可以替代 `CopyBackBuffer` 方法，调用基本实现，如果它返回 `null`，你可以返回占位符 `BitmapSource`。

你还可以实现自己的软件呈现，而不是调用基本实现。

### ① 备注

如果 WPF 在软件中完全呈现，则不会显示 D3DImage，因为 WPF 没有前台缓冲区。

## 另请参阅

- [D3DImage](#)
- [Direct3D9 和 WPF 互操作性的性能注意事项](#)
- [演练：创建在 WPF 中承载的 Direct3D9 内容](#)
- [演练：在 WPF 中承载 Direct3D9 内容](#)

# Direct3D9 和 WPF 互操作性的性能注意事项

项目 · 2023/02/06

可以使用 [D3DImage](#) 类托管 Direct3D 内容。 托管 Direct3D 内容可能会影响应用程序的性能。 本主题介绍了在 Windows Presentation Foundation (WPF) 应用程序中托管 Direct3D 内容时优化性能的最佳做法。 这些最佳做法包括如何使用 [D3DImage](#) 以及在使用 Windows Vista、Windows XP 和多监视器显示时的最佳做法。

## ① 备注

有关演示这些最佳做法的代码示例，请参阅 [WPF 和 Direct3D9 互操作](#)。

## 谨慎使用 D3DImage

[D3DImage](#) 实例中托管的 Direct3D 内容的呈现速度不如纯 Direct3D 应用程序中的快。 复制图面并刷新命令缓冲区可能代价高昂。 随着 [D3DImage](#) 实例数的增加，会发生更多刷新，并且性能会下降。 因此，应谨慎使用 [D3DImage](#)。

## Windows Vista 上的最佳做法

对于具有配置为使用 Windows 显示驱动程序模型 (WDDM) 的显示器的 Windows Vista，若要实现最佳性能，请在 `IDirect3DDevice9Ex` 设备上创建 Direct3D 图面。 这将启用图面共享。 视频卡必须支持 Windows Vista 上的

`D3DDEVCAPS2_CAN_STRETCHRECT_FROM_TEXTURES` 和 `D3DCAPS2_CANSHARERESOURCE` 驱动程序功能。 任何其他设置都会使得通过软件复制图面，这将大大降低性能。

## ① 备注

如果 Windows Vista 具有配置为使用 Windows XP 显示驱动程序模型 (XDDM) 的显示器，则无论设置如何，都会始终通过软件复制图面。 借助适当的设置和视频卡，当你使用 WDDM 时，Windows Vista 的性能会更好，因为图面复制是在硬件中执行的。

## Windows XP 上的最佳做法

为了在使用 Windows XP 显示驱动程序模型 (XDDM) 的 Windows XP 上获得最佳性能，请创建一个在调用 `IDirect3DSurface9::GetDC` 方法时可正常运行的可锁定图面。在内部，`BitBlt` 方法在硬件中跨设备传输图面。`GetDC` 方法始终适用于 XRGB 图面。但是，如果客户端计算机运行的是带有 SP3 或 SP2 的 Windows XP，并且客户端还具有分层窗口功能的修补程序，则此方法仅适用于 ARGB 图面。视频卡必须支持 `D3DDEVCAPS2_CAN_STRETCHRECT_FROM_TEXTURES` 驱动程序功能。

16 位桌面显示深度会显着降低性能。建议使用 32 位桌面。

如果要针对 Windows Vista 和 Windows XP 进行开发，请测试 Windows XP 上的性能。Windows XP 上的视频内存不足是一个问题。此外，由于需要额外的视频内存副本，Windows XP 上的 `D3DImage` 使用的视频内存和带宽比 Windows Vista WDDM 更多。因此，对于相同的视频硬件，可以预料在 Windows XP 上的性能会比在 Windows Vista 上的性能差。

### ① 备注

XDDM 在 Windows XP 和 Windows Vista 上均可用；但是，WDDM 仅在 Windows Vista 上可用。

## 常规最佳做法

创建设备时，请使用 `D3DCREATE_MULTITHREADED` 创建标志。这会降低性能，但 WPF 呈现系统会从另一个线程调用此设备上的方法。请务必正确遵循锁定协议，这样就不会有两个线程同时访问设备。

如果在 WPF 托管线程上执行呈现，强烈建议使用 `D3DCREATE_FPU_PRESERVE` 创建标志创建设备。如果没有此设置，D3D 呈现会降低 WPF 双精度运算的准确性并引入呈现问题。

平铺 `D3DImage` 很快，除非在无硬件支持的情况下平铺非 pow2 图面，或者平铺包含 `D3DImage` 的 `DrawingBrush` 或 `VisualBrush`。

## 多监视器显示的最佳做法

如果使用的计算机具有多个监视器，则应遵循前面所述的最佳做法。多监视器配置还有一些其他性能注意事项。

创建后台缓冲区时，它是在特定设备和适配器上创建的，但 WPF 可能在任何适配器上显示前台缓冲区。跨适配器复制以更新前台缓冲区的开销可能非常大。在配置为将 WDDM 与多个视频卡和 `IDirect3DDevice9Ex` 设备配合使用的 Windows Vista 上，如果前台缓冲区位于不同的适配器上但仍使用同一个视频卡，则不会造成性能损失。但是，在

Windows XP 和具有多个视频卡的 XDDM 上，当前台缓冲区与后台缓冲区显示在不同的适配器上时，会有显著的性能损失。有关详细信息，请参阅 [WPF 和 Direct3D9 互操作](#)。

## 性能摘要

下表显示了作为操作系统、像素格式和画面锁定功能的前台缓冲区更新的性能。假定前台缓冲区和后台缓冲区位于同一适配器上。根据适配器配置，硬件更新通常比软件更新快得多。

画面像素格式	Windows Vista、 WDDM 和 9Ex	其他 Windows Vista 配置	Windows XP SP3 或 SP2 w/ 修补程序	Windows XP SP2
D3DFMT_X8R8G8B8 ( 不可锁定 )	硬件更新	软件更新	软件更新	软件更新
D3DFMT_X8R8G8B8 ( 可锁定 )	硬件更新	软件更新	硬件更新	硬件更新
D3DFMT_A8R8G8B8 ( 不可锁定 )	硬件更新	软件更新	软件更新	软件更新
D3DFMT_A8R8G8B8 ( 可锁定 )	硬件更新	软件更新	硬件更新	软件更新

## 另请参阅

- [D3DImage](#)
- [WPF 和 Direct3D9 互操作](#)
- [演练：创建在 WPF 中承载的 Direct3D9 内容](#)
- [演练：在 WPF 中承载 Direct3D9 内容](#)

# 演练：创建在 WPF 中承载的 Direct3D9 内容

项目 • 2023/02/06

本演练演示了如何创建适用于在 Windows Presentation Foundation (WPF) 应用程序中托管的 Direct3D9 内容。有关在 WPF 应用程序中托管 Direct3D9 内容的详细信息，请参阅 [WPF 和 Direct3D9 互操作](#)。

在本演练中，您将执行下列任务：

- 创建 Direct3D9 项目。
- 配置 Direct3D9 项目以在 WPF 应用程序中托管。

完成后，你将有一个 DLL，其中包含用于 WPF 应用程序的 Direct3D9 内容。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio 2010。
- DirectX SDK 9 或更高版本。

## 创建 Direct3D9 项目

第一步是创建和配置 Direct3D9 项目。

### 要创建 Direct3D9 项目

1. 用 C++ 创建新的 Win32 项目，命名为 `D3DContent`。

Win32 应用程序向导随即打开并显示欢迎屏幕。

2. 单击“下一步”。

显示“应用程序设置”屏幕。

3. 在“应用程序类型”部分，选择“DLL”选项。

4. 单击“完成”。

生成 D3DContent 项目。

5. 在解决方案资源管理器中，右键单击 D3DContent 项目，然后选择“属性”。

将打开“D3DContent 属性页”对话框。

6. 选择“C/C++”节点。

7. 在“其他包含目录”字段中，指定 DirectX 包含文件夹的位置。此文件夹的默认位置为 %ProgramFiles%\Microsoft DirectX SDK (version)\Include。

8. 双击“链接器”节点以将其展开。

9. 在“其他库目录”字段中，指定 DirectX 库文件夹的位置。此文件夹的默认位置为 %ProgramFiles%\Microsoft DirectX SDK (version)\Lib\x86。

10. 选择“输入”节点。

11. 在“其他依赖项”字段中，添加 `d3d9.lib` 和 `d3dx9.lib` 文件。

12. 在解决方案资源管理器中，将新的模块定义文件 (.def) `D3DContent.def` 添加到项目。

## 创建 Direct3D9 内容

若要获得最佳性能，Direct3D9 内容必须使用特定设置。以下代码演示了如何创建具有最佳性能特征的 Direct3D9 图面。有关详细信息，请参阅 [Direct3D9 和 WPF 互操作性的性能考虑](#)。

### 要创建 Direct3D9 内容

1. 使用解决方案资源管理器，将三个 C++ 类添加到具有以下名称的项目。

`CRenderer` ( 具有虚拟析构函数 )

`CRendererManager`

`CTriangleRenderer`

2. 在代码编辑器中打开 `Renderer.h`，并用以下代码替换自动生成的代码。

C++

```
#pragma once
```

```

class CRenderer
{
public:
    virtual ~CRenderer();

    HRESULT CheckDeviceState();
    HRESULT CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha,
    UINT m_uNumSamples);

    virtual HRESULT Render() = 0;

    IDirect3DSurface9 *GetSurfaceNoRef() { return m_pd3dRTS; }

protected:
    CRenderer();

    virtual HRESULT Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND
hwnd, UINT uAdapter);

    IDirect3DDevice9    *m_pd3dDevice;
    IDirect3DDevice9Ex *m_pd3dDeviceEx;

    IDirect3DSurface9 *m_pd3dRTS;

};

```

3. 在代码编辑器中打开 Renderer.cpp，并用以下代码替换自动生成的代码。

```

C++

//+-----
-----
// 
//  CRenderer
//
//      An abstract base class that creates a device and a target
// render
//      surface. Derive from this class and override Init() and
// Render()
//      to do your own rendering. See CTriangleRenderer for an example.

//-----
-----

#include "StdAfx.h"

//+-----
-----
// 
//  Member:
//      CRenderer ctor
//
```

```
//-----  
//  
CRenderer::CRenderer() : m_pd3dDevice(NULL), m_pd3dDeviceEx(NULL),  
m_pd3dRTS(NULL)  
{  
}  
  
//+-----  
//  
// Member:  
//      CRenderer dtor  
//  
//-----  
  
CRenderer::~CRenderer()  
{  
    SAFE_RELEASE(m_pd3dDevice);  
    SAFE_RELEASE(m_pd3dDeviceEx);  
    SAFE_RELEASE(m_pd3dRTS);  
}  
  
//+-----  
//  
// Member:  
//      CRenderer::CheckDeviceState  
//  
// Synopsis:  
//      Returns the status of the device. 9Ex devices are a special  
case because  
//      TestCooperativeLevel() has been deprecated in 9Ex.  
//  
//-----  
  
HRESULT  
CRenderer::CheckDeviceState()  
{  
    if (m_pd3dDeviceEx)  
    {  
        return m_pd3dDeviceEx->CheckDeviceState(NULL);  
    }  
    else if (m_pd3dDevice)  
    {  
        return m_pd3dDevice->TestCooperativeLevel();  
    }  
    else  
    {  
        return D3DERR_DEVICELOST;  
    }  
}  
  
//+-----  
//
```

```

// Member:
//     CRenderer::CreateSurface
//
// Synopsis:
//     Creates and sets the render target
//
//-----+
-----+
HRESULT
CRenderer::CreateSurface(UINT uWidth, UINT uHeight, bool fUseAlpha,
UINT m_uNumSamples)
{
    HRESULT hr = S_OK;

    SAFE_RELEASE(m_pd3dRTS);

    IFC(m_pd3dDevice->CreateRenderTarget(
        uWidth,
        uHeight,
        fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8,
        static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
        0,
        m_pd3dDeviceEx ? FALSE : TRUE, // Lockable RT required for
good XP perf
        &m_pd3dRTS,
        NULL
    ));

    IFC(m_pd3dDevice->SetRenderTarget(0, m_pd3dRTS));

Cleanup:
    return hr;
}

//+-----+
-----+
// Member:
//     CRenderer::Init
//
// Synopsis:
//     Creates the device
//
//-----+
-----+
HRESULT
CRenderer::Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND hwnd, UINT
uAdapter)
{
    HRESULT hr = S_OK;

    D3DPRESENT_PARAMETERS d3dpp;
    ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
}

```

```

d3dpp.BackBufferHeight = 1;
d3dpp.BackBufferWidth = 1;
d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;

D3DCAPS9 caps;
DWORD dwVertexProcessing;
IFC(pD3D->GetDeviceCaps(uAdapter, D3DDEVTYPE_HAL, &caps));
if ((caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT) ==
D3DDEVCAPS_HWTRANSFORMANDLIGHT)
{
    dwVertexProcessing = D3DCREATE_HARDWARE_VERTEXPROCESSING;
}
else
{
    dwVertexProcessing = D3DCREATE_SOFTWARE_VERTEXPROCESSING;
}

if (pD3DEx)
{
    IDirect3DDevice9Ex *pd3dDevice = NULL;
    IFC(pD3DEx->CreateDeviceEx(
        uAdapter,
        D3DDEVTYPE_HAL,
        hwnd,
        dwVertexProcessing | D3DCREATE_MULTITHREADED |
D3DCREATE_FPU_PRESERVE,
        &d3dpp,
        NULL,
        &m_pd3dDeviceEx
    ));

    IFC(m_pd3dDeviceEx->QueryInterface(__uuidof(IDirect3DDevice9),
reinterpret_cast<void**>(&m_pd3dDevice)));
}
else
{
    assert(pD3D);

    IFC(pD3D->CreateDevice(
        uAdapter,
        D3DDEVTYPE_HAL,
        hwnd,
        dwVertexProcessing | D3DCREATE_MULTITHREADED |
D3DCREATE_FPU_PRESERVE,
        &d3dpp,
        &m_pd3dDevice
    ));
}

Cleanup:
    return hr;
}

```

4. 在代码编辑器中打开 RendererManager.h，并用以下代码替换自动生成的代码。

C++

```
#pragma once

class CRenderer;

class CRendererManager
{
public:
    static HRESULT Create(CRendererManager **ppManager);
    ~CRendererManager();

    HRESULT EnsureDevices();

    void SetSize(UINT uWidth, UINT uHeight);
    void SetAlpha(bool fUseAlpha);
    void SetNumDesiredSamples(UINT uNumSamples);
    void SetAdapter(POINT screenSpacePoint);

    HRESULT GetBackBufferNoRef(IDirect3DSurface9 **ppSurface);

    HRESULT Render();

private:
    CRendererManager();

    void CleanupInvalidDevices();
    HRESULT EnsureRenderers();
    HRESULT EnsureHWND();
    HRESULT EnsureD3DObjects();
    HRESULT TestSurfaceSettings();
    void DestroyResources();

    IDirect3D      *m_pD3D;
    IDirect3D9Ex  *m_pD3DEx;

    UINT m_cAdapters;
    CRenderer **m_rgRenderers;
    CRenderer *m_pCurrentRenderer;

    HWND m_hwnd;

    UINT m_uWidth;
    UINT m_uHeight;
    UINT m_uNumSamples;
    bool m_fUseAlpha;
    bool m_fSurfaceSettingsChanged;
};

};
```

5. 在代码编辑器中打开 RendererManager.cpp , 并用以下代码替换自动生成的代码。

C++

```
//+-----  
//  
//  CRendererManager  
//  
//      Manages the list of CRenderers. Managed code pinvoke into this  
//      class  
//      and this class forwards to the appropriate CRenderer.  
//  
//-----  
  
#include "StdAfx.h"  
  
const static TCHAR szAppName[] = TEXT("D3DImageSample");  
typedef HRESULT (WINAPI *DIRECT3DCREATE9EXFUNCTION)(UINT SDKVersion,  
IDirect3D9Ex**);  
  
//+-----  
//  
//  Member:  
//      CRendererManager ctor  
//  
//-----  
  
CRendererManager::CRendererManager()  
:  
    m_pD3D(NULL),  
    m_pD3DEx(NULL),  
    m_cAdapters(0),  
    m_hwnd(NULL),  
    m_pCurrentRenderer(NULL),  
    m_rgRenderers(NULL),  
    m_uWidth(1024),  
    m_uHeight(1024),  
    m_uNumSamples(0),  
    m_fUseAlpha(false),  
    m_fSurfaceSettingsChanged(true)  
{  
  
}  
  
//+-----  
//  
//  Member:  
//      CRendererManager dtor  
//  
//-----  
  
CRendererManager::~CRendererManager()  
{
```

```

DestroyResources();

    if (m_hwnd)
    {
        DestroyWindow(m_hwnd);
        UnregisterClass(szAppName, NULL);
    }
}

//+-----
//-----  

// Member:  

//     CRendererManager::Create  

//  

// Synopsis:  

//     Creates the manager  

//  

//-----  

//-----  

HRESULT
CRendererManager::Create(CRendererManager **ppManager)
{
    HRESULT hr = S_OK;

    *ppManager = new CRendererManager();
    IFCOOM(*ppManager);

    Cleanup:
        return hr;
}

//+-----
//-----  

// Member:  

//     CRendererManager::EnsureRenderers  

//  

// Synopsis:  

//     Makes sure the CRenderer objects exist  

//  

//-----  

//-----  

HRESULT
CRendererManager::EnsureRenderers()
{
    HRESULT hr = S_OK;

    if (!m_rgRenderers)
    {
        IFC(EnsureHWND());

        assert(m_cAdapters);
        m_rgRenderers = new CRenderer*[m_cAdapters];
        IFCOOM(m_rgRenderers);
    }
}

```

```

        ZeroMemory(m_rgRenderers, m_cAdapters *
sizeof(m_rgRenderers[0]));

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            IFC(CTriangleRenderer::Create(m_pD3D, m_pD3DEx, m_hwnd, i,
&m_rgRenderers[i]));
        }

        // Default to the default adapter
        m_pCurrentRenderer = m_rgRenderers[0];
    }

Cleanup:
    return hr;
}

//+-----+
//-----+
// Member:
//     CRendererManager::EnsureHWND
//-----+
// Synopsis:
//     Makes sure an HWND exists if we need it
//-----+
//-----+
HRESULT
CRendererManager::EnsureHWND()
{
    HRESULT hr = S_OK;

    if (!m_hwnd)
    {
        WNDCLASS wndclass;

        wndclass.style = CS_HREDRAW | CS_VREDRAW;
        wndclass.lpfnWndProc = DefWindowProc;
        wndclass.cbClsExtra = 0;
        wndclass.cbWndExtra = 0;
        wndclass.hInstance = NULL;
        wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
        wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
        wndclass.hbrBackground = (HBRUSH) GetStockObject (WHITE_BRUSH);
        wndclass.lpszMenuName = NULL;
        wndclass.lpszClassName = szAppName;

        if (!RegisterClass(&wndclass))
        {
            IFC(E_FAIL);
        }

        m_hwnd = CreateWindow(szAppName,
                            TEXT("D3DImageSample"),

```

```

        WS_OVERLAPPEDWINDOW,
        0,                      // Initial X
        0,                      // Initial Y
        0,                      // Width
        0,                      // Height
        NULL,
        NULL,
        NULL,
        NULL);
    }

Cleanup:
    return hr;
}

//+-----+
//-----+
// Member:
//     CRendererManager::EnsureD3DObjects
//
// Synopsis:
//     Makes sure the D3D objects exist
//
//-----+
//-----+
HRESULT
CRendererManager::EnsureD3DObjects()
{
    HRESULT hr = S_OK;

    HMODULE hD3D = NULL;
    if (!m_pD3D)
    {
        hD3D = LoadLibrary(TEXT("d3d9.dll"));
        DIRECT3DCREATE9EXFUNCTION pfnCreate9Ex =
(DIRECT3DCREATE9EXFUNCTION)GetProcAddress(hD3D, "Direct3DCreate9Ex");
        if (pfnCreate9Ex)
        {
            IFC((*pfnCreate9Ex)(D3D_SDK_VERSION, &m_pD3DEx));
            IFC(m_pD3DEx->QueryInterface(__uuidof(IDirect3D9),
reinterpret_cast<void **>(&m_pD3D)));
        }
        else
        {
            m_pD3D = Direct3DCreate9(D3D_SDK_VERSION);
            if (!m_pD3D)
            {
                IFC(E_FAIL);
            }
        }
    }

    m_cAdapters = m_pD3D->GetAdapterCount();
}

```

```

Cleanup:
    if (hD3D)
    {
        FreeLibrary(hD3D);
    }

    return hr;
}

//+-----
//-----  

// Member:  

//     CRendererManager::CleanupInvalidDevices  

//  

// Synopsis:  

//     Checks to see if any devices are bad and if so, deletes all  

resources  

//  

//     We could delete resources and wait for D3DERR_DEVICENOTRESET  

and reset  

//     the devices, but if the device is lost because of an adapter  

order  

//     change then our existing D3D objects would have stale adapter  

//     information. We'll delete everything to be safe rather than  

sorry.  

//  

//-----  

//-----  

void  

CRendererManager::CleanupInvalidDevices()
{
    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        if (FAILED(m_rgRenderers[i]->CheckDeviceState()))
        {
            DestroyResources();
            break;
        }
    }
}

//+-----
//-----  

// Member:  

//     CRendererManager::GetBackBufferNoRef  

//  

// Synopsis:  

//     Returns the surface of the current renderer without adding a  

reference  

//  

//     This can return NULL if we're in a bad device state.  

//-----  

//-----
```

```

-----
HRESULT
CRenderManager::GetBackBufferNoRef(IDirect3DSurface9 **ppSurface)
{
    HRESULT hr = S_OK;

    // Make sure we at least return NULL
    *ppSurface = NULL;

    CleanupInvalidDevices();

    IFC(EnsureD3DObjects());

    //
    // Even if we never render to another adapter, this sample creates
    devices
    // and resources on each one. This is a potential waste of video
    memory,
    // but it guarantees that we won't have any problems (e.g. out of
    video
    // memory) when switching to render on another adapter. In your own
    code
    // you may choose to delay creation but you'll need to handle the
    issues
    // that come with it.
    //

    IFC(EnsureRenderers());

    if (m_fSurfaceSettingsChanged)
    {
        if (FAILED(TestSurfaceSettings()))
        {
            IFC(E_FAIL);
        }

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            IFC(m_rgRenderers[i]->CreateSurface(m_uWidth, m_uHeight,
m_fUseAlpha, m_uNumSamples));
        }

        m_fSurfaceSettingsChanged = false;
    }

    if (m_pCurrentRenderer)
    {
        *ppSurface = m_pCurrentRenderer->GetSurfaceNoRef();
    }
}

Cleanup:
// If we failed because of a bad device, ignore the failure for now
and
// we'll clean up and try again next time.
if (hr == D3DERR_DEVICELOST)

```

```

    {
        hr = S_OK;
    }

    return hr;
}

//+-----
-----  

//  

// Member:  

//     CRendererManager::TestSurfaceSettings  

//  

// Synopsis:  

//     Checks to see if our current surface settings are allowed on  

all  

//     adapters.  

//  

//-----  

-----  

HRESULT  

CRendererManager::TestSurfaceSettings()  

{
    HRESULT hr = S_OK;

    D3DFORMAT fmt = m_fUseAlpha ? D3DFMT_A8R8G8B8 : D3DFMT_X8R8G8B8;

    //
    // We test all adapters because because we potentially use all
adapters.
    // But even if this sample only rendered to the default adapter,
you
    // should check all adapters because WPF may move your surface to
    // another adapter for you!
    //

    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        // Can we get HW rendering?
        IFC(m_pD3D->CheckDeviceType(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            fmt,
            TRUE
        ));

        // Is the format okay?
        IFC(m_pD3D->CheckDeviceFormat(
            i,
            D3DDEVTYPE_HAL,
            D3DFMT_X8R8G8B8,
            D3DUSAGE_RENDERTARGET | D3DUSAGE_DYNAMIC, // We'll use
dynamic when on XP
            D3DRTYPE_SURFACE,

```

```

        fmt
    ));

    // D3DImage only allows multisampling on 9Ex devices. If we
    can't
    // multisample, overwrite the desired number of samples with 0.
    if (m_pD3DEx && m_uNumSamples > 1)
    {
        assert(m_uNumSamples <= 16);

        if (FAILED(m_pD3D->CheckDeviceMultiSampleType(
            i,
            D3DDEVTYPE_HAL,
            fmt,
            TRUE,
            static_cast<D3DMULTISAMPLE_TYPE>(m_uNumSamples),
            NULL
            )))
        {
            m_uNumSamples = 0;
        }
    }
    else
    {
        m_uNumSamples = 0;
    }
}

Cleanup:
    return hr;
}

//+-----+
//-----+
// Member:
//     CRendererManager::DestroyResources
//
// Synopsis:
//     Delete all D3D resources
//
//-----+
//-----+
void
CRendererManager::DestroyResources()
{
    SAFE_RELEASE(m_pD3D);
    SAFE_RELEASE(m_pD3DEx);

    for (UINT i = 0; i < m_cAdapters; ++i)
    {
        delete m_rgRenderers[i];
    }
    delete [] m_rgRenderers;
    m_rgRenderers = NULL;
}

```

```
    m_pCurrentRenderer = NULL;
    m_cAdapters = 0;

    m_fSurfaceSettingsChanged = true;
}

//+-----
//-----  

//  

// Member:  

//      CRendererManager::SetSize  

//  

// Synopsis:  

//      Update the size of the surface. Next render will create a new  

//      surface.  

//  

//-----  

//-----  

void
CRendererManager::SetSize(UINT uWidth, UINT uHeight)
{
    if (uWidth != m_uWidth || uHeight != m_uHeight)
    {
        m_uWidth = uWidth;
        m_uHeight = uHeight;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----
//-----  

//  

// Member:  

//      CRendererManager::SetAlpha  

//  

// Synopsis:  

//      Update the format of the surface. Next render will create a new  

//      surface.  

//  

//-----  

//-----  

void
CRendererManager::SetAlpha(bool fUseAlpha)
{
    if (fUseAlpha != m_fUseAlpha)
    {
        m_fUseAlpha = fUseAlpha;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----
//-----  

//
```

```

// Member:
//     CRendererManager::SetNumDesiredSamples
//
// Synopsis:
//     Update the MSAA settings of the surface. Next render will
//     create a
//     new surface.
//
//-----
-----  

void
CRendererManager::SetNumDesiredSamples(UINT uNumSamples)
{
    if (m_uNumSamples != uNumSamples)
    {
        m_uNumSamples = uNumSamples;
        m_fSurfaceSettingsChanged = true;
    }
}

//+-----  

-----  

// Member:
//     CRendererManager::SetAdapter
//
// Synopsis:
//     Update the current renderer. Next render will use the new
//     renderer.
//
//-----
-----  

void
CRendererManager::SetAdapter(POINT screenSpacePoint)
{
    CleanupInvalidDevices();

    //
    // After CleanupInvalidDevices, we may not have any D3D objects.
    Rather than
        // recreate them here, ignore the adapter update and wait for
        render to recreate.
    //

    if (m_pD3D && m_rgRenderers)
    {
        HMONITOR hMon = MonitorFromPoint(screenSpacePoint,
MONITOR_DEFAULTTONULL);

        for (UINT i = 0; i < m_cAdapters; ++i)
        {
            if (hMon == m_pD3D->GetAdapterMonitor(i))
            {
                m_pCurrentRenderer = m_rgRenderers[i];
                break;
            }
        }
    }
}

```

```

        }
    }
}

//+-----+
// Member:
//      CRendererManager::Render
//
// Synopsis:
//      Forward to the current renderer
//
//-----+
HRESULT
CRendererManager::Render()
{
    return m_pCurrentRenderer ? m_pCurrentRenderer->Render() : S_OK;
}

```

6. 在代码编辑器中打开 TriangleRenderer.h，并用以下代码替换自动生成的代码。

```

C++

#pragma once

class CTriangleRenderer : public CRenderer
{
public:
    static HRESULT Create(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEX, HWND hwnd, UINT uAdapter, CRenderer **ppRenderer);
    ~CTriangleRenderer();

    HRESULT Render();

protected:
    HRESULT Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEX, HWND hwnd, UINT uAdapter);

private:
    CTriangleRenderer();
    IDirect3DVertexBuffer9 *m_pd3dVB;
};

```

7. 在代码编辑器中打开 TriangleRenderer.cpp，并用以下代码替换自动生成的代码。

```
C++
```

```
//+-----  
----  
//  
//  CTriangleRenderer  
//  
//      Subclass of CRenderer that renders a single, spinning triangle  
//  
//-----  
----  
  
#include "StdAfx.h"  
  
struct CUSTOMVERTEX  
{  
    FLOAT x, y, z;  
    DWORD color;  
};  
  
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)  
  
//+-----  
----  
//  
//  Member:  
//      CTriangleRenderer ctor  
//  
//-----  
----  
CTriangleRenderer::CTriangleRenderer() : CRenderer(), m_pd3dVB(NULL)  
{  
}  
  
//+-----  
----  
//  
//  Member:  
//      CTriangleRenderer dtor  
//  
//-----  
----  
CTriangleRenderer::~CTriangleRenderer()  
{  
    SAFE_RELEASE(m_pd3dVB);  
}  
  
//+-----  
----  
//  
//  Member:  
//      CTriangleRenderer::Create  
//  
//  Synopsis:  
//      Creates the renderer
```

```

//-
//-----+
-----+
HRESULT
CTriangleRenderer::Create(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND
hwnd, UINT uAdapter, CRenderer **ppRenderer)
{
    HRESULT hr = S_OK;

    CTriangleRenderer *pRenderer = new CTriangleRenderer();
    IFCOOM(pRenderer);

    IFC(pRenderer->Init(pD3D, pD3DEx, hwnd, uAdapter));

    *ppRenderer = pRenderer;
    pRenderer = NULL;

Cleanup:
    delete pRenderer;

    return hr;
}

//+-----+
-----+
//  Member:
//      CTriangleRenderer::Init
//
//  Synopsis:
//      Override of CRenderer::Init that calls base to create the
device and
//      then creates the CTriangleRenderer-specific resources
//
//-----+
-----+
HRESULT
CTriangleRenderer::Init(IDirect3D9 *pD3D, IDirect3D9Ex *pD3DEx, HWND
hwnd, UINT uAdapter)
{
    HRESULT hr = S_OK;
    D3DXMATRIXA16 matView, matProj;
    D3DXVECTOR3 vEyePt(0.0f, 0.0f, -5.0f);
    D3DXVECTOR3 vLookatPt(0.0f, 0.0f, 0.0f);
    D3DXVECTOR3 vUpVec(0.0f, 1.0f, 0.0f);

    // Call base to create the device and render target
    IFC(CRenderer::Init(pD3D, pD3DEx, hwnd, uAdapter));

    // Set up the VB
    CUSTOMVERTEX vertices[] =
    {
        { -1.0f, -1.0f, 0.0f, 0xffff0000, }, // x, y, z, color
        { 1.0f, -1.0f, 0.0f, 0xff00ff00, },
        { 0.0f, 1.0f, 0.0f, 0xff00ffff, },
    }
}

```

```

};

IFC(m_pd3dDevice->CreateVertexBuffer(sizeof(vertices), 0,
D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, &m_pd3dVB, NULL));

void *pVertices;
IFC(m_pd3dVB->Lock(0, sizeof(vertices), &pVertices, 0));
memcpy(pVertices, vertices, sizeof(vertices));
m_pd3dVB->Unlock();

// Set up the camera
D3DXMatrixLookAtLH(&matView, &vEyePt, &vLookatPt, &vUpVec);
IFC(m_pd3dDevice->SetTransform(D3DTS_VIEW, &matView));
D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI / 4, 1.0f, 1.0f,
100.0f);
IFC(m_pd3dDevice->SetTransform(D3DTS_PROJECTION, &matProj));

// Set up the global state
IFC(m_pd3dDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE));
IFC(m_pd3dDevice->SetRenderState(D3DRS_LIGHTING, FALSE));
IFC(m_pd3dDevice->SetStreamSource(0, m_pd3dVB, 0,
sizeof(CUSTOMVERTEX)));
IFC(m_pd3dDevice->SetFVF(D3DFVF_CUSTOMVERTEX));

Cleanup:
    return hr;
}

//+-----
//-----  

//  

// Member:  

//     CTriangleRenderer::Render  

//  

// Synopsis:  

//     Renders the rotating triangle  

//-----  

//-----  

HRESULT
CTriangleRenderer::Render()
{
    HRESULT hr = S_OK;
    D3DXMATRIXA16 matWorld;

    IFC(m_pd3dDevice->BeginScene());
    IFC(m_pd3dDevice->Clear(
        0,
        NULL,
        D3DCLEAR_TARGET,
        D3DCOLOR_ARGB(128, 0, 0, 128), // NOTE: Premultiplied alpha!
        1.0f,
        0
    ));
}

```

```

// Set up the rotation
UINT iTime = GetTickCount() % 1000;
FLOAT fAngle = iTime * (2.0f * D3DX_PI) / 1000.0f;
D3DXMatrixRotationY(&matWorld, fAngle);
IFC(m_pd3dDevice->SetTransform(D3DTS_WORLD, &matWorld));

IFC(m_pd3dDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1));

IFC(m_pd3dDevice->EndScene());

Cleanup:
    return hr;
}

```

8. 在代码编辑器中打开 stdafx.h，并用以下代码替换自动生成的代码。

C++

```

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#pragma once

#define WIN32_LEAN_AND_MEAN           // Exclude rarely-used stuff
from Windows headers
// Windows Header Files:
#include <windows.h>

#include <d3d9.h>
#include <d3dx9.h>

#include <assert.h>

#include "RendererManager.h"
#include "Renderer.h"
#include "TriangleRenderer.h"

#define IFC(x) { hr = (x); if (FAILED(hr)) goto Cleanup; }
#define IFCOOM(x) { if ((x) == NULL) { hr = E_OUTOFMEMORY; IFC(hr); } }
#define SAFE_RELEASE(x) { if (x) { x->Release(); x = NULL; } }


```

9. 在代码编辑器中打开 dllmain.cpp，并用以下代码替换自动生成的代码。

C++

```

// dllmain.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"

BOOL APIENTRY DllMain( HMODULE hModule,

```

```

        DWORD ul_reason_for_call,
        LPVOID lpReserved
    )
{
    switch (ul_reason_for_call)
    {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}

static CRendererManager *pManager = NULL;

static HRESULT EnsureRendererManager()
{
    return pManager ? S_OK : CRendererManager::Create(&pManager);
}

extern "C" HRESULT WINAPI SetSize(UINT uWidth, UINT uHeight)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetSize(uWidth, uHeight);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI SetAlpha(BOOL fUseAlpha)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetAlpha(!fUseAlpha);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI SetNumDesiredSamples(UINT uNumSamples)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetNumDesiredSamples(uNumSamples);

Cleanup:

```

```

    return hr;
}

extern "C" HRESULT WINAPI SetAdapter(POINT screenSpacePoint)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    pManager->SetAdapter(screenSpacePoint);

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI GetBackBufferNoRef(IDirect3DSurface9
**ppSurface)
{
    HRESULT hr = S_OK;

    IFC(EnsureRendererManager());

    IFC(pManager->GetBackBufferNoRef(ppSurface));

Cleanup:
    return hr;
}

extern "C" HRESULT WINAPI Render()
{
    assert(pManager);

    return pManager->Render();
}

extern "C" void WINAPI Destroy()
{
    delete pManager;
    pManager = NULL;
}

```

10. 在代码编辑器中打开 D3DContent.def。

11. 使用以下代码替换自动生成的代码。

C++

```

LIBRARY "D3DContent"

EXPORTS

SetSize
SetAlpha

```

```
SetNumDesiredSamples
```

```
SetAdapter
```

```
GetBackBufferNoRef
```

```
Render
```

```
Destroy
```

12. 生成项目。

## 后续步骤

- 在 WPF 应用程序中托管 Direct3D9 内容。有关详细信息，请参阅[演练：在 WPF 中托管 Direct3D9 内容](#)。

## 另请参阅

- [D3DImage](#)
- [Direct3D9 和 WPF 互操作性的性能注意事项](#)
- [演练：在 WPF 中承载 Direct3D9 内容](#)

# 演练：在 WPF 中承载 Direct3D9 内容

项目 • 2022/09/27

本演练演示如何在 Windows Presentation Foundation (WPF) 应用程序中托管 Direct3D9 内容。

在本演练中，您将执行下列任务：

- 创建一个 WPF 项目来托管 Direct3D9 内容。
- 导入 Direct3D9 内容。
- 使用 [D3DImage](#) 类显示 Direct3D9 内容。

完成后，你将了解如何在 WPF 应用程序中托管 Direct3D9 内容。

## 先决条件

您需要满足以下条件才能完成本演练：

- Visual Studio。
- DirectX SDK 9 或更高版本。
- 包含 WPF 兼容格式的 Direct3D9 内容的 DLL。有关详细信息，请参阅 [WPF 和 Direct3D9 互操作](#)和[演练：创建在 WPF 中托管的 Direct3D9 内容](#)。

## 创建 WPF 项目

第一步是为 WPF 应用程序创建项目。

## 创建 WPF 项目

在 Visual C# 中创建一个名为 `D3DHost` 的 WPF 应用程序项目。有关详细信息，请参阅[演练：我的第一个 WPF 桌面应用程序](#)。

`MainWindow.xaml` 在 WPF 设计器中随即打开。

## 导入 Direct3D9 内容

使用 `DllImport` 属性从非托管 DLL 导入 Direct3D9 内容。

# 导入 Direct3D9 内容

1. 在代码编辑器中打开 MainWindow.xaml.cs。
2. 使用以下代码替换自动生成的代码。

```
C#  
  
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Data;  
using System.Windows.Documents;  
using System.Windows.Input;  
using System.Windows.Interop;  
using System.Windows.Media;  
using System.Windows.Media.Imaging;  
using System.Windows.Navigation;  
using System.Windows.Shapes;  
using System.Windows.Threading;  
using System.Runtime.InteropServices;  
using System.Security.Permissions;  
  
namespace D3DHost  
{  
    public partial class MainWindow : Window  
    {  
        public MainWindow()  
        {  
            InitializeComponent();  
  
            // Set up the initial state for the D3DImage.  
            HRESULT.Check(SetSize(512, 512));  
            HRESULT.Check(SetAlpha(false));  
            HRESULT.Check(SetNumDesiredSamples(4));  
  
            //  
            // Optional: Subscribing to the  
IsFrontBufferAvailableChanged event.  
            //  
            // If you don't render every frame (e.g. you only render in  
            // reaction to a button click), you should subscribe to the  
            // IsFrontBufferAvailableChanged event to be notified when  
rendered content  
            // is no longer being displayed. This event also notifies  
you when  
            // the D3DImage is capable of being displayed again.  
  
            // For example, in the button click case, if you don't  
render again when
```

```
// the IsFrontBufferAvailable property is set to true, your
// D3DImage won't display anything until the next button
click.
//
// Because this application renders every frame, there is
no need to
// handle the IsFrontBufferAvailableChanged event.
//
CompositionTarget.Rendering += new
EventHandler(CompositionTarget_Rendering);

//
// Optional: Multi-adapter optimization
//
// The surface is created initially on a particular
adapter.
// If the WPF window is dragged to another adapter, WPF
// ensures that the D3DImage still shows up on the new
// adapter.
//
// This process is slow on Windows XP.
//
// Performance is better on Vista with a 9Ex device. It's
only
// slow when the D3DImage crosses a video-card boundary.
//
// To work around this issue, you can move your surface
when
// the D3DImage is displayed on another adapter. To
// determine when that is the case, transform a point on
the
// D3DImage into screen space and find out which adapter
// contains that screen space point.
//
// When your D3DImage straddles two adapters, nothing
// can be done, because one will be updating slowly.
//
_adapterTimer = new DispatcherTimer();
_adapterTimer.Tick += new EventHandler(AdapterTimer_Tick);
_adapterTimer.Interval = new TimeSpan(0, 0, 0, 0, 500);
_adapterTimer.Start();

//
// Optional: Surface resizing
//
// The D3DImage is scaled when WPF renders it at a size
// different from the natural size of the surface. If the
// D3DImage is scaled up significantly, image quality
// degrades.
//
// To avoid this, you can either create a very large
// texture initially, or you can create new surfaces as
// the size changes. Below is a very simple example of
// how to do the latter.
//
```

```

        // By creating a timer at Render priority, you are
guaranteed
            // that new surfaces are created while the element
            // is still being arranged. A 200 ms interval gives
            // a good balance between image quality and performance.
            // You must be careful not to create new surfaces too
            // frequently. Frequently allocating a new surface may
            // fragment or exhaust video memory. This issue is more
            // significant on XDDM than it is on WDDM, because WDDM
            // can page out video memory.
            //
            // Another approach is deriving from the Image class,
            // participating in layout by overriding the
ArrangeOverride method, and
                // updating size in the overriden method. Performance will
degrade
                    // if you resize too frequently.
                    //
                    // Blurry D3DImages can still occur due to subpixel
                    // alignments.
                    //
_sizeTimer = new
DispatcherTimer(DispatcherPriority.Render);
    _sizeTimer.Tick += new EventHandler(SizeTimer_Tick);
    _sizeTimer.Interval = new TimeSpan(0, 0, 0, 0, 200);
    _sizeTimer.Start();
}

~MainWindow()
{
    Destroy();
}

void AdapterTimer_Tick(object sender, EventArgs e)
{
    POINT p = new POINT(imgelt.PointToScreen(new Point(0, 0)));

    HRESULT.Check(SetAdapter(p));
}

void SizeTimer_Tick(object sender, EventArgs e)
{
    // The following code does not account for
RenderTransforms.
        // To handle that case, you must transform up to the root
and
        // check the size there.

        // Given that the D3DImage is at 96.0 DPI, its Width and
Height
            // properties will always be integers. ActualWidth/Height
            // may not be integers, so they are cast to integers.
            uint actualWidth = (uint)imgelt.ActualWidth;
            uint actualHeight = (uint)imgelt.ActualHeight;
            if ((actualWidth > 0 && actualHeight > 0) &&

```

```

        (actualWidth != (uint)d3dimg.Width || actualHeight !=
(uint)d3dimg.Height))
    {
        HRESULT.Check(SetSize(actualWidth, actualHeight));
    }
}

void CompositionTarget_Rendering(object sender, EventArgs e)
{
    RenderingEventArgs args = (RenderingEventArgs)e;

    // It's possible for Rendering to call back twice in the
    same frame
    // so only render when we haven't already rendered in this
    frame.
    if (d3dimg.IsFrontBufferAvailable && _lastRender !=
args.RenderingTime)
    {
        IntPtr pSurface = IntPtr.Zero;
        HRESULT.Check(GetBackBufferNoRef(out pSurface));
        if (pSurface != IntPtr.Zero)
        {
            d3dimg.Lock();
            // Repeatedly calling SetBackBuffer with the same
IntPtr is
            // a no-op. There is no performance penalty.

d3dimg.SetBackBuffer(D3DResourceType.IDirect3DSurface9, pSurface);
            HRESULT.Check(Render());
            d3dimg.AddDirtyRect(new Int32Rect(0, 0,
d3dimg.PixelWidth, d3dimg.PixelHeight));
            d3dimg.Unlock();

            _lastRender = args.RenderingTime;
        }
    }
}

DispatcherTimer _sizeTimer;
DispatcherTimer _adapterTimer;
TimeSpan _lastRender;

// Import the methods exported by the unmanaged Direct3D
content.

[DllImport("D3DCode.dll")]
static extern int GetBackBufferNoRef(IntPtr pSurface);

[DllImport("D3DCode.dll")]
static extern int SetSize(uint width, uint height);

[DllImport("D3DCode.dll")]
static extern int SetAlpha(bool useAlpha);

[DllImport("D3DCode.dll")]

```

```

        static extern int SetNumDesiredSamples(uint numSamples);

        [StructLayout(LayoutKind.Sequential)]
        struct POINT
        {
            public POINT(Point p)
            {
                x = (int)p.X;
                y = (int)p.Y;
            }

            public int x;
            public int y;
        }

        [DllImport("D3DCode.dll")]
        static extern int SetAdapter(POINT screenSpacePoint);

        [DllImport("D3DCode.dll")]
        static extern int Render();

        [DllImport("D3DCode.dll")]
        static extern void Destroy();
    }

    public static class HRESULT
    {
        public static void Check(int hr)
        {
            Marshal.ThrowExceptionForHR(hr);
        }
    }
}

```

## 托管 Direct3D9 内容

最后，使用 [D3DImage](#) 类来托管 Direct3D9 内容。

## 托管 Direct3D9 内容

1. 在 MainWindow.xaml 中，将自动生成的 XAML 替换为以下 XAML。

XAML

```

<Window x:Class="D3DHost.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:i="clr-
    namespace:System.Windows.Interop;assembly=PresentationCore"
    Title="MainWindow" Height="300" Width="300"

```

```
Background="PaleGoldenrod">
<Grid>
    <Image x:Name="imgelt">
        <Image.Source>
            <i:D3DImage x:Name="d3dimg" />
        </Image.Source>
    </Image>
</Grid>
</Window>
```

2. 生成项目。
3. 将包含 Direct3D9 内容的 DLL 复制到 bin/Debug 文件夹。
4. 按 F5 运行项目。

Direct3D9 内容显示在 WPF 应用程序中。

## 另请参阅

- [D3DImage](#)
- [Direct3D9 和 WPF 互操作性的性能注意事项](#)

# 性能

项目 · 2023/02/06

实现最佳应用程序性能需要在应用程序设计中预先计划，并了解 WPF 应用程序的最佳做法。

## 本节内容

图形呈现层

优化 WPF 应用程序性能

演练：在 WPF 应用程序中缓存应用程序数据

## 参考

[RenderCapability](#)

[Tier](#)

[PresentationTraceSources](#)

## 请参阅

- [布局](#)
- [动画提示和技巧](#)

# 图形呈现层

项目 · 2022/09/27

呈现层为运行 WPF 应用程序的设备定义图形硬件功能和性能级别。

## 图形硬件

对呈现层级别影响最大的图形硬件功能包括：

- **视频 RAM** - 图形硬件中的视频内存量决定了可用于合成图形的缓冲区大小和数量。
- **像素着色器** - 像素着色器是基于像素计算效果的图形处理功能。 每个显示帧可能有数百万像素需要处理，具体取决于显示图形的分辨率。
- **顶点着色器** - 顶点着色器是对对象的顶点数据执行数学运算的图形处理功能。
- **多纹理支持** - 多纹理支持是指对 3D 图形对象执行混合操作期间应用两个或更多个不同纹理的功能。 多纹理支持的程度取决于图形硬件中的多纹理单元数。

## 呈现层定义

图形硬件的功能决定了 WPF 应用程序的呈现功能。 WPF 系统定义了 3 个呈现层：

- **呈现层 0** - 无图形硬件加速。 所有图形功能都使用软件加速。 DirectX 版本级别低于 9.0。
- **呈现层 1** - 某些图形功能使用图形硬件加速。 DirectX 版本级别高于或等于 9.0。
- **呈现层 2** - 大多数图形功能都使用图形硬件加速。 DirectX 版本级别高于或等于 9.0。

`RenderCapability.Tier` 属性可用于在应用程序运行时检索呈现层。 使用呈现层可确定设备是否支持某些硬件加速图形功能。 然后，应用程序就可以在运行时根据设备支持的呈现层采用不同的代码路径。

## 呈现层 0

呈现层的值为 0 意味着设备上的应用程序没有图形硬件加速可用。 在这一层次级别，应假设所有图形都由软件呈现，未采用硬件加速。 该层的功能对应于低于 9.0 的 DirectX 版本。

# 呈现层 1 与呈现层 2

## ① 备注

从 .NET Framework 4 开始，呈现层 1 进行了重新定义，只包含支持 DirectX 9.0 或更高版本的图形硬件。支持 DirectX 7 或 8 的图形硬件现定义为呈现层 0。

呈现层的值为 1 或 2 意味着，如果必要的系统资源可用并且尚未耗尽，则 WPF 的大部分图形功能会使用硬件加速。这对应于高于或等于 9.0 的 DirectX 版本。

下表显示呈现层 1 和呈现层 2 的图形硬件需求差异：

Feature	第 1 层	第 2 层
DirectX 版本	必须高于或等于 9.0。	必须高于或等于 9.0。
视频 RAM	必须大于或等于 60 MB。	必须大于或等于 120 MB。
像素着色器	版本级别必须高于或等于 2.0。	版本级别必须高于或等于 2.0。
顶点着色器	没有要求。	版本级别必须高于或等于 2.0。
多纹理单元	没有要求。	单元数必须大于或等于 4。

以下功能对呈现层 1 和呈现层 2 采用硬件加速：

Feature	备注
2D 呈现	支持大多数 2D 呈现。
3D 光栅化	支持大多数 3D 光栅化。
3D 各向异性筛选	WPF 在呈现 3D 内容时尝试使用各向异性筛选。各向异性筛选是指改善离相机较远且与相机角度较大的图面上纹理的图像质量。
3D MIP 映射	WPF 在呈现 3D 内容时尝试使用 MIP 映射。纹理占据 <a href="#">Viewport3D</a> 中的较小视图区域时，MIP 映射可改进纹理呈现的质量。
径向渐变	如果支持，请避免在大型对象上使用 <a href="#">RadialGradientBrush</a> 。
3D 光照计算	WPF 执行每个顶点的光照，这意味着必须在应用于网格的每个材料的每个顶点计算光照强度。
文本呈现	子像素字体呈现使用图形硬件上可用的像素着色器。

以下功能仅对呈现层 2 采用硬件加速：

## Feature 备注

3D 抗锯齿 只有支持 Windows 显示驱动程序模型 (WDDM) 的操作系统 ( 如 Windows Vista 和 Windows 7 ) 才支持 3D 抗锯齿。

以下功能未采用硬件加速：

Feature	备注
打印内容	所有打印内容都使用 WPF 软件管道呈现。
使用 <code>RenderTargetBitmap</code> 的栅格化内容	使用 <code>RenderTargetBitmap</code> 的 <code>Render</code> 方法呈现的任何内容。
使用 <code>TileBrush</code> 的平 铺内容	<code>TileBrush</code> 的 <code>TileMode</code> 属性设置为 <code>Tile</code> 的任何平铺内容。
超过图形硬件最大纹理 大小的图面	对大多数图形硬件而言，大型图面是指达到 2048x2048 或 4096x4096 像素大小的图面。
视频 RAM 要求超过 图形硬件内存的任何 操作	可使用 Windows SDK 中的 <a href="#">WPF 性能套件</a> 包含的分析器工具来监视应用程序视频 RAM 的使用情况。
分层窗口	分层窗口允许 WPF 应用程序将内容呈现到非矩形窗口中的屏幕。在支持 Windows 显示驱动程序模型 (WDDM) 的操作系统 ( 如 Windows Vista 和 Windows 7 ) 上，分层窗口采用硬件加速。在 Windows XP 等其他系统上，分层窗口是通过软件来呈现的，未采用硬件加速。

在 WPF 中，可通过设置以下 `Window` 属性来启用分层窗口：

- `WindowStyle = None`
- `AllowsTransparency = true`
- `Background = Transparent`

## 其他资源

以下资源可帮助你分析 WPF 应用程序的性能特征。

## 图形呈现注册表设置

WPF 提供了四个注册表设置来控制 WPF 呈现：

设置	说明
禁用硬件加速选项	指定是否应启用硬件加速。

设置	说明
<b>最大多重采样值</b>	指定用于消除 3D 内容锯齿的多重采样级别。
<b>必需的视频驱动程序日期设置</b>	指定系统是否对 2004 年 11 月之前发布的驱动程序禁用硬件加速。
<b>使用参考光栅器选项</b>	指定 WPF 是否应使用参考光栅器。

知道如何引用 WPF 注册表设置的任何外部配置实用工具都可以访问这些设置。 还可以直接使用 Windows 注册表编辑器来访问这些值，从而创建或修改这些设置。 有关详细信息，请参阅[图形呈现注册表设置](#)。

## WPF 性能分析工具

WPF 提供了一套性能分析工具，此工具可帮助分析应用程序的运行时行为，并确定可应用的性能优化类型。 下表列出了 Windows SDK 工具中包括的性能分析工具，WPF 性能套件：

工具	说明
分析器	用于分析呈现行为。
可视化探查器	用于按可视化树中的元素分析 WPF 服务（如布局和事件处理）的使用。

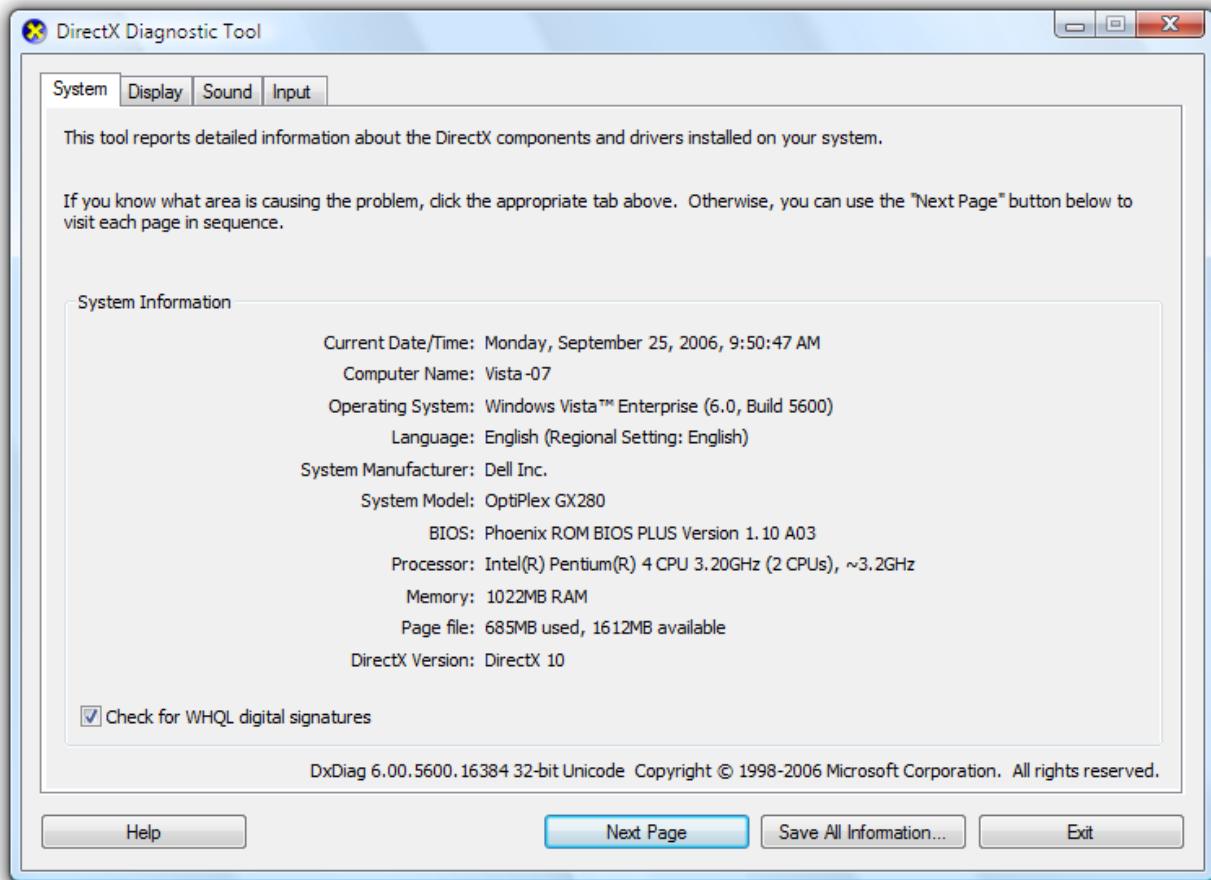
WPF 性能套件提供丰富的性能数据的图形视图。 有关 WPF 性能工具的详细信息，请参阅[WPF 性能套件](#)。

## DirectX 诊断工具

DirectX 诊断工具 Dxdiag.exe 专门用于帮助你解决 DirectX 相关问题。 DirectX 诊断工具的默认安装文件夹是：

`~\Windows\System32`

运行 DirectX 诊断工具时，主窗口中包含一组可用于显示和诊断 DirectX 相关信息的选项卡。 例如，“系统”选项卡提供有关计算机的系统信息，并指定安装在计算机上的 DirectX 版本。



"DirectX 诊断工具"主窗口

## 另请参阅

- [RenderCapability](#)
- [RenderOptions](#)
- [优化 WPF 应用程序性能](#)
- [WPF 性能套件](#)
- [图形呈现注册表设置](#)
- [动画提示和技巧](#)

# 优化 WPF 应用程序性能

项目 • 2023/02/06

本部分旨在作为 WPF 的参考，你应该首先熟悉这两个平台。本部分假定你具备这两者的应用知识，是为知识储备足以启动和运行其应用程序的程序员编写的。

## ① 备注

本部分提供的性能数据基于在具有 512 RAM 和 ATI Radeon 9700 图形卡的 2.8 GHz 电脑上运行的 WPF 应用程序。

## 本节内容

[规划应用程序性能](#)

[利用硬件](#)

[布局和示例](#)

[二维图形和图像处理](#)

[对象行为](#)

[应用程序资源](#)

[文本](#)

[数据绑定](#)

[控件](#)

[其他性能建议](#)

[应用程序启动时间](#)

## 另请参阅

- [RenderOptions](#)
- [RenderCapability](#)
- [图形呈现层](#)
- [WPF 图形呈现疑难解答](#)
- [布局](#)

- WPF 中的树
- Drawing 对象概述
- 使用 DrawingVisual 对象
- 依赖项属性概述
- Freezable 对象概述
- XAML 资源
- WPF 中的文档
- 绘制格式化文本
- WPF 中的版式
- 数据绑定概述
- 导航概述
- 动画提示和技巧
- 演练：在 WPF 应用程序中缓存应用程序数据

# 规划应用程序性能

项目 • 2023/02/06

能否成功实现性能目标取决于如何制定性能策略。规划是开发任何产品的第一阶段。本主题介绍一些非常简单的规则，用于开发良好的性能策略。

## 对各种场景进行考虑

场景可以帮助你专注于应用程序的关键组件。场景通常派生自客户，以及竞争性产品。始终研究你的客户，找出真正让他们对你的产品和竞争对手的产品感到兴奋的原因。客户的反馈可以帮助你确定应用程序的主要场景。例如，如果正在设计一个将在启动时使用的组件，那么该组件很可能只在应用程序启动时调用一次。启动时间即成为关键场景。关键场景的其他示例可能是动画序列所需的帧速率，或应用程序允许的最大工作集。

## 定义目标

目标有助于确定应用程序的执行速度是快还是慢。应为所有场景定义目标。你定义的所有性能目标都应基于客户的期望。在应用程序开发周期的早期设置性能目标可能很困难，因为那时仍然有许多未解决的问题。然而，最好设置初始目标，后续再加以修订，总比根本没有目标要好。

## 了解平台

在应用程序开发周期中始终保持测量、调查、改进/更正的周期。从开发周期的开始到结束，都需要在可靠、稳定的环境中测量应用程序的性能。应避免由外部因素引起的可变性。例如，在测试性能时，应禁用防病毒或任何自动更新（如 SMS），以免影响性能测试结果。测量应用程序的性能后，便需要确定哪些更改可以带来最大的改进。修改应用程序后，请重新开始此循环。

## 使性能优化成为一个迭代过程

你应该知道你将使用的每个功能的相对成本。例如，就计算资源而言，Microsoft .NET Framework 中反射的使用通常是性能密集型的，因此你需要明智地使用它。这并不意味着要避免使用反射，只是应该小心地平衡应用程序的性能需求和所使用功能的性能需求。

## 构建图形丰富性

创建可扩展方法以实现 WPF 应用程序性能的一项关键技术是构建图形丰富性和复杂性。始终从使用性能密集程度最低的资源开始，以实现场景目标。一旦实现了这些目标，就可以通过使用更多性能密集型功能来构建图形丰富性，并始终牢记场景目标。请记住，WPF 是一个非常丰富的平台，提供非常丰富的图形功能。不加思索地使用性能密集型功能会对整体应用程序性能产生负面影响。

WPF 控件本质上是可扩展的，允许对其外观进行广泛自定义，同时不改变其控件行为。通过利用样式、数据模板和控件模板，可以创建并逐步发展可自定义的用户界面 (UI)，以适应性能要求。

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [利用硬件](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [文本](#)
- [数据绑定](#)
- [其他性能建议](#)

# 优化性能：利用硬件

项目 · 2023/02/06

WPF 的内部体系结构有两个呈现管道，硬件和软件。本主题提供有关这些呈现管道的信息，以帮助做出有关应用程序性能优化的决策。

## 硬件呈现管道

决定 WPF 性能的最重要因素之一是它受呈现限制 - 必须呈现的像素越多，性能成本就越高。但是，可以卸载到图形处理单元 (GPU) 的呈现越多，可以获得的性能优势就越多。WPF 应用程序硬件呈现管道充分利用了支持最低 Microsoft DirectX 7.0 版的硬件上的 Microsoft DirectX 功能。支持 Microsoft DirectX 7.0 版和 PixelShader 2.0+ 功能的硬件可以进一步获得优化。

## 软件呈现管道

WPF 软件呈现管道完全受 CPU 限制。WPF 利用 CPU 中的 SSE 和 SSE2 指令集来实现优化的、功能齐全的软件光栅器。在无法使用硬件呈现管道呈现应用程序功能的任何时候，回退到软件都是无缝的。

在软件模式下呈现时遇到的最大性能问题与填充率有关，填充率定义为正在呈现的像素数。如果担心软件呈现模式下的性能，请尽量减少重绘像素的次数。例如，如果有一个蓝色背景的应用程序，然后在其上呈现一个稍微透明的图像，则将在应用程序中呈现所有像素两次。因此，与只有蓝色背景相比，使用图像呈现应用程序所花费的时间将增加一倍。

## 图形呈现层

预测应用程序将在其上运行的硬件配置可能非常困难。但是，可能需要考虑一种设计，允许应用程序在不同硬件上运行时无缝切换功能，以便充分利用每种不同的硬件配置。

为了实现这一点，WPF 提供了在运行时确定系统图形功能的功能。图形功能是通过将视频卡归类为三个呈现功能层之一来确定的。WPF 公开了一个 API，允许应用程序查询呈现功能层。然后，应用程序就可以在运行时根据硬件支持的呈现层采用不同的代码路径。

对呈现层级别影响最大的图形硬件功能包括：

- **视频 RAM** - 图形硬件中的视频内存量决定了可用于合成图形的缓冲区大小和数量。

- **像素着色器** - 像素着色器是基于像素计算效果的图形处理功能。每个显示帧可能有数百万像素需要处理，具体取决于显示图形的分辨率。
- **顶点着色器** - 顶点着色器是对对象的顶点数据执行数学运算的图形处理功能。
- **多纹理支持** - 多纹理支持是指对 3D 图形对象执行混合操作期间应用两个或更多个不同纹理的功能。多纹理支持的程度取决于图形硬件中的多纹理单元数。

像素着色器、顶点着色器和多纹理功能用于定义特定的 DirectX 版本级别，而这些级别又被用来定义 WPF 中的不同呈现层。

图形硬件的功能决定了 WPF 应用程序的呈现功能。WPF 系统定义了 3 个呈现层：

- **呈现层 0** - 无图形硬件加速。DirectX 版本级别低于 7.0。
- **呈现层 1** - 部分图形硬件加速。DirectX 版本级别高于或等于 7.0，并且低于 9.0。
- **呈现层 2** - 大多数图形功能都使用图形硬件加速。DirectX 版本级别高于或等于 9.0。

有关 WPF 呈现层的详细信息，请参阅[图形呈现层](#)。

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [规划应用程序性能](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [文本](#)
- [数据绑定](#)
- [其他性能建议](#)

# 优化性能：布局和设计

项目 · 2023/02/06

WPF 应用程序的设计可能会在计算布局和验证对象引用时产生不必要的开销，从而影响性能。对象的构造会影响应用程序的性能特征，在运行时更是如此。

本主题提供这些方面的性能改进建议。

## Layout

“布局过程”一词描述了测量和排列 [Panel](#)（派生对象的子级集合）成员的过程，然后在屏幕上绘制它们。布局处理过程是一个数学密集型过程，即：集合中的子级数目越多，所需的计算量就越大。例如，每当集合中的子 [UIElement](#) 对象改变其位置时，它有可能触发布局系统的一个新的过程。由于对象特征与布局行为之间的关系非常紧密，因此有必要了解可以调用布局系统的事件类型。应用程序将尽可能减少不必要的布局处理过程调用，从而改善性能。

布局系统对集合中的每个子成员都完成两个处理过程：测量处理过程和排列处理过程。每个子对象提供其自身对 [Measure](#) 和 [Arrange](#) 方法的重写实现，从而提供其自身特定的布局行为。简单地说，布局是一个递归系统，实现在屏幕上对元素进行大小调整、定位和绘制。

- 子 [UIElement](#) 对象通过首先测量其核心属性来开始布局过程。
- 与大小（如 [Width](#)、[Height](#) 和 [Margin](#)）相关的对象的 [FrameworkElement](#) 属性将进行计算。
- [Panel](#)（应用特定逻辑），例如 [DockPanel](#) 的 [Dock](#) 属性，或 [StackPanel](#) 的 [Orientation](#) 属性。
- 在测量所有的子对象后，将排列或定位内容。
- 将子对象集合绘制到屏幕上。

如果发生下列任一操作，将再次调用布局处理过程：

- 向集合中添加了一个子对象。
- [LayoutTransform](#) 将应用于子对象。
- [UpdateLayout](#) 方法用于调用子对象。
- 使用影响测量或排列过程的元数据进行标记的依赖属性的值发生更改时。

# 尽可能使用最高效的面板

布局过程的复杂性直接取决于 [Panel](#) ( 使用的派生元素 ) 的布局行为。例如，[Grid](#) 或 [StackPanel](#) 控件提供比 [Canvas](#) 控件更多的功能。功能大幅度改进的代价是性能成本的大幅度提高。但是，如果不需要 [Grid](#) 控件提供的功能，则应使用成本较低的其他控件，如 [Canvas](#) 或自定义面板。

有关详细信息，请参阅[面板概述](#)。

## 更新而不替换 RenderTransform

可以更新 [Transform](#)，而不是用 [RenderTransform](#) 属性的值加以替代。在涉及动画的方案中，尤其是这样。通过更新现有的 [Transform](#)，可以避免启动不必要的布局计算。

## 从上到下生成树

在逻辑树中添加或删除节点时，会在该节点的父级及其所有子级上引起属性失效。因此，应始终遵循从上到下的构造模式，以避免由于在经过验证的节点中出现不必要的失效而付出代价。下表显示了自上而下构建树与自下而上构建树的执行速度差异，其中树有 150 级深，并且每一级都有单一的 [TextBlock](#) 和 [DockPanel](#)。

操作	构建树 (以毫秒为单位)	呈现 - 包括生成树 (以毫秒为单位)
从下到上	366	454
从上到下	11	96

以下代码示例演示如何从上到下创建树。

C#

```
private void OnBuildTreeTopDown(object sender, RoutedEventArgs e)
{
    TextBlock textBlock = new TextBlock();
    textBlock.Text = "Default";

    DockPanel parentPanel = new DockPanel();
    DockPanel childPanel;

    myCanvas.Children.Add(parentPanel);
    myCanvas.Children.Add(textBlock);

    for (int i = 0; i < 150; i++)
    {
        textBlock = new TextBlock();
        textBlock.Text = "Default";
        parentPanel.Children.Add(textBlock);
```

```
    childPanel = new DockPanel();
    parentPanel.Children.Add(childPanel);
    parentPanel = childPanel;
}
}
```

有关逻辑树的详细信息，请参阅 [WPF 中的树](#)。

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [规划应用程序性能](#)
- [利用硬件](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [文本](#)
- [数据绑定](#)
- [其他性能建议](#)
- [布局](#)

# 优化性能：二维图形和图像处理

项目 · 2023/02/06

WPF 提供了多种可按应用程序要求进行优化的 2D 图形和图像处理功能。本主题提供有关这些方面性能优化的信息。

## 绘图和形状

WPF 提供 [Drawing](#) 和 [Shape](#) 对象来表示图形绘制内容。但是，[Drawing](#) 对象是比 [Shape](#) 对象更为简单的构造，且提供更好的性能特征。

通过 [Shape](#) 可将图形形状绘制到屏幕。因为其派生自 [FrameworkElement](#) 类，所以 [Shape](#) 对象可用于面板和大多数控件内。

WPF 为图形和绘制服务提供多层访问。在顶层，[Shape](#) 对象易于使用，且提供了布局和事件处理等众多实用功能。WPF 提供了许多现成可用的形状对象。所有形状对象都继承自 [Shape](#) 类。可用的形状对象包括 [Ellipse](#)、[Line](#)、[Path](#)、[Polygon](#)、[Polyline](#) 和 [Rectangle](#)。

另一方面，[Drawing](#) 对象不是派生自 [FrameworkElement](#) 类，且对于绘制形状、图像和文本提供了更轻量的实现。

有四种类型的 [Drawing](#) 对象：

- [GeometryDrawing](#) 绘制形状。
- [ImageDrawing](#) 绘制图像。
- [GlyphRunDrawing](#) 绘制文本。
- [DrawingGroup](#) 绘制其他图形。使用图形组将其他图形合并到单个复合图形。

[GeometryDrawing](#) 对象用于绘制几何内容。[Geometry](#) 类及其派生的具体类（例如 [CombinedGeometry](#)、[EllipseGeometry](#) 和 [PathGeometry](#)）提供绘制 2D 图形的方式以及命中测试和剪裁支持。几何对象可用于定义控件的区域或定义应用于图像的剪裁区域等。几何对象可以是简单区域（例如矩形和圆形），或者是由两个或多个几何对象创建的复合区域。通过合并 [PathSegment](#) 派生对象（例如 [ArcSegment](#)、[BezierSegment](#) 和 [QuadraticBezierSegment](#)），可创建更复杂的几何区域。

表面上看，[Geometry](#) 类和 [Shape](#) 类非常相似。二者都用于绘制 2D 图形，且都具有相似的具体派生类，例如 [EllipseGeometry](#) 和 [Ellipse](#)。但是，这两个类集之间存在重大区别。一方面，[Geometry](#) 类缺少 [Shape](#) 类的部分功能，例如绘制自身的功能。若要绘制几何对象，必须使用另一个类（例如 [DrawingContext](#)、[Drawing](#) 或 [Path](#)（值得注意的是 [Path](#)

是一个 Shape ) ) 来执行绘制操作。 填充、笔划和笔划粗细等绘制属性位于绘制几何对象的类上，而形状对象中包含这些属性。 可以这样来理解这种差别：几何对象定义区域（例如圆形），而形状对象定义区域、定义如何填充区域和设置区域边框并参与布局系统。

由于 Shape 对象派生自 FrameworkElement 类，因此使用这些对象会使应用程序的内存消耗显著增加。 如果对于图形内容确实不需要 FrameworkElement 功能，请考虑使用更轻量的 Drawing 对象。

有关 Drawing 对象的详细信息，请参阅 [Drawing 对象概述](#)。

## StreamGeometry 对象

在创建几何形状时，StreamGeometry 对象是对 PathGeometry 的轻量替代。 需要描述复杂几何时使用 StreamGeometry。 StreamGeometry 非常适合用于处理许多 PathGeometry 对象，且与使用众多单独的 PathGeometry 对象相比，其性能更佳。

以下示例使用特性语法在 XAML 中创建一个三角形 StreamGeometry。

XAML

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

        <Path Data="F0 M10,100 L100,100 100,50Z"
              StrokeThickness="1" Stroke="Black"/>

    </StackPanel>
</Page>
```

有关 StreamGeometry 对象的详细信息，请参阅[使用 StreamGeometry 创建形状](#)。

## DrawingVisual 对象

DrawingVisual 对象是一个轻量绘图类，用于绘制形状、图像或文本。 此类之所以为轻量类是因为它不提供布局或事件处理，从而性能得以提升。 因此，绘图非常适用于背景和剪贴画。 有关详细信息，请参阅[使用 DrawingVisual 对象](#)。

## 图像

WPF 图像处理大幅改进了 Windows 先前版本的图像处理功能。 显示位图或在公共控件上使用图像等图像处理功能以前主要由 Microsoft Windows 图形设备接口 (GDI) 或

Microsoft Windows GDI+ 应用程序编程接口 (API) 处理。这些 API 提供基线图像处理功能，但缺少编解码器扩展性支持和高保真图像支持等功能。WPF 图像处理 API 已经过重新设计，克服了 GDI 和 GDI+ 的缺点，提供一组新的 API，用于在应用程序内显示和使用图像。

使用图像时，为使性能更佳，请考虑以下建议：

- 如果应用程序要求显示缩略图，请考虑创建一个缩略版图像。默认情况下，WPF 会加载图像并将其解码到最大尺寸。如果仅需要图像的缩略版本，WPF 会先将图像解码为完整尺寸，然后将其缩放为缩略图大小，这个过程是多余的。若要避免这个多余的过程，可请求 WPF 将图像解码到缩略图大小，或者请求 WPF 加载缩略图大小的图像。
- 请务必将图像解码到所需大小而不是默认大小。如上所述，请求 WPF 将图像解码为所需大小而不是默认的最大尺寸。这样不仅会减小应用程序的工作集，还会减慢其执行速度。
- 如果可能，请将多个图像合并到单个图像中，例如由多个图像构成的胶卷条。
- 有关详细信息，请参阅 [图像概述](#)。

## BitmapScalingMode

对任何位图缩放进行动画处理时，默认的高质量图像重采样算法有时可能由于消耗过多系统资源导致帧速率下降，继而导致动画明显变慢。通过将 [RenderOptions](#) 对象的 [BitmapScalingMode](#) 属性设置为 [LowQuality](#)，缩放位图时可创建更为流畅的动画。处理图像时，[LowQuality](#) 模式会通知 WPF 绘制引擎从质量优化算法切换至速度优化算法。

下面的示例演示如何为图像对象设置 [BitmapScalingMode](#)。

C#

```
// Set the bitmap scaling mode for the image to render faster.  
RenderOptions.SetBitmapScalingMode(MyImage, BitmapScalingMode.LowQuality);
```

## CachingHint

默认情况下，WPF 不会缓存 [TileBrush](#) 对象的绘制内容，例如 [DrawingBrush](#) 和 [VisualBrush](#)。在静态方案中，内容和场景中 [TileBrush](#) 的使用皆不会发生改变，这样具有一定意义，因为可节省视频内存。以非静态方式使用具有静态内容的 [TileBrush](#) 时（例如静态 [DrawingBrush](#) 或 [VisualBrush](#) 映射到旋转 3D 对象的图面时），这样做的意义不大。WPF 的默认行为是对每个帧重新绘制 [DrawingBrush](#) 或 [VisualBrush](#) 的整个内容（即使内容未改变也是如此）。

通过将 `RenderOptions` 对象的 `CachingHint` 属性设置为 `Cache`，可使用缓存版平铺画笔对象来提升性能。

`CacheInvalidateThresholdMinimum` 和 `CacheInvalidateThresholdMaximum` 属性值为相对大小值，这些值决定由于比例更改而应何时重新生成 `TileBrush` 对象。例如，通过将 `CacheInvalidateThresholdMaximum` 属性设置为 2.0，`TileBrush` 的缓存仅需在其大小超过当前缓存大小两倍时重新生成。

如下示例演示如何对 `DrawingBrush` 使用缓存提示选项。

C#

```
DrawingBrush drawingBrush = new DrawingBrush();

// Set the caching hint option for the brush.
RenderOptions.SetCachingHint(drawingBrush, CachingHint.Cache);

// Set the minimum and maximum relative sizes for regenerating the tiled
brush.
// The tiled brush will be regenerated and re-cached when its size is
// 0.5x or 2x of the current cached size.
RenderOptions.SetCacheInvalidateThresholdMinimum(drawingBrush, 0.5);
RenderOptions.SetCacheInvalidateThresholdMaximum(drawingBrush, 2.0);
```

## 另请参阅

- 优化 WPF 应用程序性能
- 规划应用程序性能
- 利用硬件
- 布局和示例
- 对象行为
- 应用程序资源
- 文本
- 数据绑定
- 其他性能建议
- 动画提示和技巧

# 优化性能：对象行为

项目 · 2022/10/19

了解 WPF 对象的内部行为有助于在功能和性能之间做出适当的取舍。

## 不删除对象的事件处理程序可能会使对象保持活动状态

对象传递给其事件的委托是对该对象的有效引用。因此，事件处理程序可以使对象保持活动状态的时间超过预期时间。当对已注册为侦听对象事件的对象执行清理时，在释放对象前删除委托是非常必要的。将不需要的对象保持为活动状态会增加应用程序的内存使用量。在对象为逻辑树或可视化树的根时更是如此。

WPF 为事件引入了弱事件监听器模式，在很难跟踪源和监听器之间的对象生存期关系时，这种模式特别有用。某些现有 WPF 事件使用此模式。如果要实现具有自定义事件的对象，此模式可能会有用。有关详细信息，请参阅[弱事件模式](#)。

有若干工具（如 CLR 探查器和工作集查看器）可以提供有关指定进程的内存使用量的信息。CLR 探查器包括分配配置文件的许多非常有用的视图，其中包括已分配类型的直方图、分配和调用关系图、显示各代垃圾回收及上述回收之后托管堆的生成状态的时间线，以及显示每个方法分配和程序集加载的调用树。有关更多信息，请参阅[性能](#)。

## 依赖属性和对象

通常，访问 [DependencyObject](#) 的依赖属性的速度不会慢于访问 CLR 属性的速度。虽然设置属性值会产生一些小的性能开销，但是获取值与从 CLR 属性获取值的速度一样快。抵消一些小的性能开销是因为依赖属性支持可靠的功能，如数据绑定、动画、继承和样式设置。有关详细信息，请参阅[依赖项属性概述](#)。

## DependencyProperty 优化

在应用程序中定义依赖属性时请务必谨慎。如果 [DependencyProperty](#) 仅影响呈现类型元数据选项，而不影响其他元数据选项（如 [AffectsMeasure](#)），则应通替代其元数据来对其进行同样的标记。有关替代或获取属性元数据的详细信息，请参阅[依赖属性元数据](#)。

如果并非所有属性更改都会影响测量、排列和呈现，则通过属性更改处理程序手动使测量、排列和呈现阶段无效的做法可能会更高效。例如，你可能决定仅在值大于设置限制

时才重新呈现背景。在这种情况下，值超过设置限制时，属性更改处理程序仅会使呈现无效。

## 将 DependencyProperty 设置为可继承会影响性能

默认情况下，注册的依赖属性是不可继承的。但可以显式地将所有属性设置为可继承。尽管这是一个有用的功能，但是将属性转换为可继承会影响性能，因为会增加属性无效的时长。

## 谨慎使用 RegisterClassHandler

调用 [RegisterClassHandler](#) 会允许保存实例状态，但请注意，每个实例上都会调用处理程序，这将导致性能问题。应用程序要求保存实例状态时，仅使用 [RegisterClassHandler](#)。

## 在注册过程中为 DependencyProperty 设置默认值

创建要求默认值的 [DependencyProperty](#) 时，请将传递的默认元数据用作 [DependencyProperty](#) 的 [Register](#) 方法的参数来设置此值。请使用此技术，而不要在构造函数中或在元素的每个实例上设置属性值。

## 使用 Register 设置 PropertyMetadata 值

创建 [DependencyProperty](#) 时，可以选择使用 [Register](#) 或 [OverrideMetadata](#) 方法设置 [PropertyMetadata](#)。尽管对象可能有一个静态构造函数来调用 [OverrideMetadata](#)，但这不是最佳方案，并且会影响性能。为了获得最佳性能，建议在调用过程中将 [PropertyMetadata](#) 设置为 [Register](#)。

## Freezable 对象

[Freezable](#) 是一种特殊的对象类型，具有两个状态：解冻和冻结。尽可能冻结对象会改进应用程序性能，并缩小其工作集。有关详细信息，请参阅 [Freezable 对象概述](#)。

每个 [Freezable](#) 都有一个 [Changed](#) 事件，只要发生更改就会引发该事件。不过，更改通知会降低应用程序的性能。

考虑以下示例，其中每个 [Rectangle](#) 使用相同的 [Brush](#) 对象：

C#

```
rectangle_1.Fill = myBrush;  
rectangle_2.Fill = myBrush;  
rectangle_3.Fill = myBrush;
```

```
// ...
rectangle_10.Fill = myBrush;
```

默认情况下，WPF 为 `SolidColorBrush` 对象的 `Changed` 事件提供事件处理程序，以便使 `Rectangle` 对象的 `Fill` 属性无效。在这种情况下，每当 `SolidColorBrush` 不得不引发其 `Changed` 事件时，必须为每个 `Rectangle` 调用回调函数，这些回调函数调用的累积将导致性能严重下降。此外，此时添加和删除处理程序将会严重影响性能，这是因为应用程序将不得不遍历整个列表才能执行此操作。如果应用程序方案从未更改 `SolidColorBrush`，你将为不必要的维护 `Changed` 事件处理程序付出代价。

冻结 `Freezable` 会改进其性能，因为不再需要因维护更改通知而消耗资源。下表显示简单的 `SolidColorBrush` 在其 `IsFrozen` 属性设置为 `true` 时的大小，并列出该属性未设置为该值时的情况以供对比。这假设将一个画笔应用于十个 `Rectangle` 对象的 `Fill` 属性。

State	大小
已冻结的 <code>SolidColorBrush</code>	212 字节
非冻结的 <code>SolidColorBrush</code>	972 字节

以下代码示例演示了此概念：

```
C#
Brush frozenBrush = new SolidColorBrush(Colors.Blue);
frozenBrush.Freeze();
Brush nonFrozenBrush = new SolidColorBrush(Colors.Blue);

for (int i = 0; i < 10; i++)
{
    // Create a Rectangle using a non-frozen Brush.
    Rectangle rectangleNonFrozen = new Rectangle();
    rectangleNonFrozen.Fill = nonFrozenBrush;

    // Create a Rectangle using a frozen Brush.
    Rectangle rectangleFrozen = new Rectangle();
    rectangleFrozen.Fill = frozenBrush;
}
```

## 解冻的可冻结对象的已更改处理程序可以使对象保持活动状态

对象传递给 `Freezable` 对象的 `Changed` 事件的委托是对该对象的有效引用。因此，`Changed` 事件处理程序可以使对象保持活动状态的时间超过预期时间。当对已注册为侦

听 [Freezable](#) 对象的 [Changed](#) 事件的对象执行清理时，在释放对象前删除委托是非常必要的。

WPF 还会在内部挂钩 [Changed](#) 事件。例如，所有值为 [Freezable](#) 的依赖属性将自动侦听 [Changed](#) 事件。采用 [Brush](#) 的 [Fill](#) 属性说明了此概念。

C#

```
Brush myBrush = new SolidColorBrush(Colors.Red);
Rectangle myRectangle = new Rectangle();
myRectangle.Fill = myBrush;
```

将 `myBrush` 赋值给 `myRectangle.Fill` 时，指向回 [Rectangle](#) 对象的委托将添加到 [SolidColorBrush](#) 对象的 [Changed](#) 事件。这意味着以下代码实际上不会使 `myRect` 可以进行垃圾回收：

C#

```
myRectangle = null;
```

在这种情况下，`myBrush` 仍会使 `myRectangle` 保持活动状态，并在其引发 [Changed](#) 事件时对其进行调用。请注意，将 `myBrush` 赋值给新 [Rectangle](#) 的 [Fill](#) 属性时，仅将另一个事件处理程序添加到 `myBrush`。

清理这些类型的对象的建议方式是从 [Fill](#) 属性删除 [Brush](#)，这将进而删除 [Changed](#) 事件处理程序。

C#

```
myRectangle.Fill = null;
myRectangle = null;
```

## 用户界面虚拟化

WPF 还提供 [StackPanel](#) 元素的一个变体，用于自动“虚拟化”数据绑定子级内容。在此上下文中，“虚拟化”一词指的是以下技术：通过此技术，从较多的数据项中生成一个对象子集，具体取决于屏幕上哪些项可见。如果在指定时刻只有少量 UI 元素位于屏幕上，则此时生成大量 UI 元素需要占用大量内存和处理器。[VirtualizingStackPanel](#)（通过 [VirtualizingPanel](#) 提供的功能）计算可见项，并与来自 [ItemsControl](#)（如 [ListBox](#) 或 [ListView](#)）的 [ItemContainerGenerator](#) 配合使用，以便仅为可见项创建元素。

作为性能优化的结果，将仅生成这些项的视觉对象，或如果它们在屏幕上是可见的，则保持活动状态。这些视觉对象不再位于控件的可视区域时，则可能已被删除。请勿将此与

数据虚拟化发生混淆，数据虚拟化中的数据对象不会全部出现在本地集合中 - 而是根据需要进行流式处理。

下表显示了将 5000 个 [TextBlock](#) 元素添加到 [StackPanel](#) 和 [VirtualizingStackPanel](#) 并使其呈现所需的运行时间。在此情形中，测量值是指从将文本字符串附加到 [ItemsControl](#) 对象的 [ItemsSource](#) 属性至面板元素显示此文本字符串之间的时间。

主机面板	呈现时间 (ms)
<a href="#">StackPanel</a>	3210
<a href="#">VirtualizingStackPanel</a>	46

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [规划应用程序性能](#)
- [利用硬件](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [应用程序资源](#)
- [文本](#)
- [数据绑定](#)
- [其他性能建议](#)

# 优化性能：应用程序资源

项目 · 2023/02/06

WPF 允许共享应用程序资源，以便支持跨类似类型元素的一致性外观或行为。本主题在此区域中提供了一些建议，可帮助你提高应用程序的性能。

有关资源的详细信息，请参阅 [XAML 资源](#)。

## 共享资源

如果应用程序使用自定义控件并在 [ResourceDictionary](#)（或 XAML 资源节点）中定义资源，则建议在 [Application](#) 或 [Window](#) 定义资源，也可以在自定义控件的默认主题中定义资源。在自定义控件的 [ResourceDictionary](#) 中定义资源会对该控件的每个实例的性能产生影响。例如，如果将性能密集型画笔操作定义为自定义控件的资源定义和许多自定义控件实例的一部分，则应用程序的工作集将显著增加。

为了说明这点，请考虑下列情况。假设你正在使用 WPF 开发卡片游戏。对于大多数卡片游戏，你需要 52 张具有 52 张不同牌面的卡片。你决定实施卡片自定义控件，并在卡片自定义控件的资源定义 52 支画笔（每个画笔代表一个牌面）。在主应用程序中，你最初创建此卡片自定义控件的 52 个实例。卡片自定义控件的每个实例会生成 [Brush](#) 对象的 52 个实例，从而在应用程序中总共提供  $52 * 52$  [Brush](#) 个对象。通过将画笔从卡片自定义控件资源移出到 [Application](#) 或 [Window](#) 对象级别，或在自定义控件的默认主题中定义它们，可以减少应用程序的工作集，因为你现在正在卡片控件的 52 个实例之间共享 52 个画笔。

## 共享画笔而无需复制

如果有多个元素使用相同的 [Brush](#) 对象，请将画笔定义为资源并引用该元素，而不是在 XAML 中定义画笔内联。此方法将创建一个实例并重复使用它，而在 XAML 中定义画笔内联会为每个元素创建一个新实例。

下面的标记示例演示了此方法：

XAML

```
<StackPanel.Resources>
    <LinearGradientBrush x:Key="myBrush" StartPoint="0,0.5" EndPoint="1,0.5"
        Opacity="0.5">
        <LinearGradientBrush.GradientStops>
            <GradientStopCollection>
                <GradientStop Color="GoldenRod" Offset="0" />
                <GradientStop Color="White" Offset="1" />
            </GradientStopCollection>
```

```

        </LinearGradientBrush.GradientStops>
    </LinearGradientBrush>
</StackPanel.Resources>

<!-- Non-shared Brush object. -->
<Label>
    Label 1
    <Label.Background>
        <LinearGradientBrush StartPoint="0,0.5" EndPoint="1,0.5" Opacity="0.5">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Color="GoldenRod" Offset="0" />
                    <GradientStop Color="White" Offset="1" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Label.Background>
</Label>

<!-- Shared Brush object. -->
<Label Background="{StaticResource myBrush}">Label 2</Label>
<Label Background="{StaticResource myBrush}">Label 3</Label>

```

## 尽可能使用静态资源

静态资源通过查找对已定义资源的引用，为任何 XAML 属性提供值。查找该资源的行为类似于编译时查找。

另一方面，动态资源将在初始编译期间创建一个临时表达式，从而延迟查找资源，直到实际需要所请求的资源值时才构建对象。查找该资源的行为类似于运行时查找，这会产生性能影响。在应用程序中尽可能使用静态资源，仅在必要时使用动态资源。

以下标记示例演示了这两种类型资源的用法：

### XAML

```

<StackPanel.Resources>
    <SolidColorBrush x:Key="myBrush" Color="Teal"/>
</StackPanel.Resources>

<!-- StaticResource reference -->
<Label Foreground="{StaticResource myBrush}">Label 1</Label>

<!-- DynamicResource reference -->
<Label Foreground="{DynamicResource {x:Static SystemColors.ControlBrushKey}}">Label 2</Label>

```

## 另请参阅

- 优化 WPF 应用程序性能
- 规划应用程序性能
- 利用硬件
- 布局和示例
- 二维图形和图像处理
- 对象行为
- 文本
- 数据绑定
- 其他性能建议

# 优化性能：文本

项目 · 2022/10/19

WPF 支持通过使用功能丰富的用户界面 (UI) 控件来呈现文本内容。通常可以将文本呈现分为三层：

1. 直接使用 [Glyphs](#) 和 [GlyphRun](#) 对象。
2. 使用 [FormattedText](#) 对象。
3. 使用高级控件，如 [TextBlock](#) 和 [FlowDocument](#) 对象。

本主题提供文本呈现性能方面的建议。

## 字形级别的呈现文本

Windows Presentation Foundation (WPF) 为希望在格式化后截取和保留文本的客户提供高级文本支持，包括可直接访问 [Glyphs](#) 的字形级标记。这些功能为以下每种方案中不同的文本呈现要求提供关键支持。

- 固定格式文档的屏幕显示。
  - Extensible Application Markup Language (XAML) 作为设备打印机语言。
  - Microsoft XPS 文档编写器。
  - 以前的打印机驱动程序，从 Win32 应用程序输出为固定格式。
  - 打印后台处理格式。
- 固定格式的文档演示，包括以前版本的 Windows 客户端和其他计算设备。

### ① 备注

[Glyphs](#) 和 [GlyphRun](#) 专为固定格式的文档演示和打印方案而设计。有关 UI 方案，请参阅 [WPF 中的版式](#)。

以下示例演示如何在 XAML 中定义 [Glyphs](#) 对象的属性。示例假定本地计算机上的 C:\WINDOWS\Fonts 文件夹中安装了 Arial、Courier New 和 Times New Roman 字体。

XAML

```
<!-- The example shows how to use a Glyphs object. -->
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
>

    <StackPanel Background="PowderBlue">

        <Glyphs
            FontUri          = "C:\WINDOWS\Fonts\TIMES.TTF"
            FontRenderingEmSize = "100"
            StyleSimulations   = "BoldSimulation"
            UnicodeString      = "Hello World!"
            Fill               = "Black"
            OriginX            = "100"
            OriginY            = "200"
        />

    </StackPanel>
</Page>
```

## 使用 DrawGlyphRun

如果具有自定义控件，并且要呈现字形，请使用 [DrawGlyphRun](#) 方法。

通过使用 [FormattedText](#) 对象，WPF 还为自定义文本格式设置提供较低级别服务。在 Windows Presentation Foundation (WPF) 中呈现文本最高效的方法是使用 [Glyphs](#) 和 [GlyphRun](#) 生成字形级别的文本内容。但是，高效的代价是丢失了丰富的文本格式设置的使用便捷性，这些文本格式设置是 Windows Presentation Foundation (WPF) 控件的内置功能，如 [TextBlock](#) 和 [FlowDocument](#)。

## FormattedText 对象

使用 [FormattedText](#) 对象可以绘制多行文本，可对文本中的每个字符单独设置格式。有关详细信息，请参阅[绘制格式化文本](#)。

若要创建格式化文本，请调用 [FormattedText](#) 构造函数来创建 [FormattedText](#) 对象。创建初始格式化文本字符串后，便可应用某一范围的格式样式。如果应用程序要实现其自身的布局，则相对于使用 [TextBlock](#) 等控件而言，[FormattedText](#) 对象是更好的选择。有关 [FormattedText](#) 对象的详细信息，请参阅[绘制格式化文本](#)。

[FormattedText](#) 对象提供低级别的文本格式设置功能。可向一个或多个字符应用多种格式样式。例如，可以同时调用 [SetFontStyle](#) 和 [SetForegroundBrush](#) 方法来更改文本中前五个字符的格式。

以下代码示例创建 `FormattedText` 对象并对其进行呈现。

C#

```
protected override void OnRender(DrawingContext drawingContext)
{
    string testString = "Lorem ipsum dolor sit amet, consectetur adipisicing
elit, sed do eiusmod tempor";

    // Create the initial formatted text string.
    FormattedText formattedText = new FormattedText(
        testString,
        CultureInfo.GetCultureInfo("en-us"),
        FlowDirection.LeftToRight,
        new Typeface("Verdana"),
        32,
        Brushes.Black);

    // Set a maximum width and height. If the text overflows these values,
    // an ellipsis "..." appears.
    formattedText.MaxTextWidth = 300;
    formattedText.MaxTextHeight = 240;

    // Use a larger font size beginning at the first (zero-based) character
    // and continuing for 5 characters.
    // The font size is calculated in terms of points -- not as device-
    // independent pixels.
    formattedText.SetFontSize(36 * (96.0 / 72.0), 0, 5);

    // Use a Bold font weight beginning at the 6th character and continuing
    // for 11 characters.
    formattedText.SetFontWeight(FontWeights.Bold, 6, 11);

    // Use a linear gradient brush beginning at the 6th character and
    // continuing for 11 characters.
    formattedText.SetForegroundBrush(
        new LinearGradientBrush(
            Colors.Orange,
            Colors.Teal,
            90.0),
        6, 11);

    // Use an Italic font style beginning at the 28th character and
    // continuing for 28 characters.
    formattedText.SetFontStyle(FontStyles.Italic, 28, 28);

    // Draw the formatted text string to the DrawingContext of the control.
    drawingContext.DrawText(formattedText, new Point(10, 0));
}
```

## FlowDocument、TextBlock 和 Label 控件

WPF 包括多个用于在屏幕中绘制文本的控件。 每个控件都面向不同的方案，并具有自己的功能和限制列表。

## FlowDocument 对性能的影响比 TextBlock 或 Label 大

通常，当需要支持有限的文本时应使用 [TextBlock](#) 元素，例如需要句子简短的用户界面 (UI)。 当需要支持最少文本时，可以使用 [Label](#)。[FlowDocument](#) 元素是可重流动文档的容器，该文档支持丰富的内容呈现，因此，相对于使用 [TextBlock](#) 或 [Label](#) 控件，该元素具有更好的性能效果。

有关 [FlowDocument](#) 的详细信息，请参阅[流文档概述](#)。

## 避免在 FlowDocument 中使用 TextBlock

[TextBlock](#) 元素派生自 [UIElement](#)。[Run](#) 元素派生自 [TextElement](#)，后者的使用成本比 [UIElement](#) 派生的对象低。 如果可能，请使用 [Run](#) 而不是 [TextBlock](#) 在 [FlowDocument](#) 中显示文本内容。

以下标记示例演示了在 [FlowDocument](#) 中设置文本内容的两种方法：

XAML

```
<FlowDocument>

    <!-- Text content within a Run (more efficient). -->
    <Paragraph>
        <Run>Line one</Run>
    </Paragraph>

    <!-- Text content within a TextBlock (less efficient). -->
    <Paragraph>
        <TextBlock>Line two</TextBlock>
    </Paragraph>

</FlowDocument>
```

## 避免使用 Run 来设置文本属性

一般情况下，相对于完全不使用显式 [Run](#) 对象，在 [TextBlock](#) 中使用 [Run](#) 占用的资源更多。 如果使用 [Run](#) 来设置文本属性，请改为直接在 [TextBlock](#) 上设置这些属性。

以下标记示例演示了设置文本属性的两种方法，在本例中为设置 [FontWeight](#) 属性：

XAML

```

<!-- Run is used to set text properties. -->
<TextBlock>
    <Run FontWeight="Bold">Hello, world</Run>
</TextBlock>

<!-- TextBlock is used to set text properties, which is more efficient. -->
<TextBlock FontWeight="Bold">
    Hello, world
</TextBlock>

```

下表演示显示 1000 个 `TextBlock` 对象（使用和不使用显式 `Run`）的成本。

TextBlock 类型	创建时间 (ms)	呈现时间 (ms)
运行设置文本属性	146	540
TextBlock 设置文本属性	43	453

## 避免对 `Label.Content` 属性进行数据绑定

想象你有一个经常从 `String` 源更新的 `Label` 对象。将 `Label` 元素的 `Content` 属性数据绑定到 `String` 源对象时，可能会遇到性能不佳的情况。每次更新源 `String` 时，将丢弃旧的 `String` 对象，并重新创建新的 `String`，因为 `String` 对象是不可变的，不能对其进行修改。而这又会导致 `Label` 对象的 `ContentPresenter` 丢弃其旧的内容，并重新生成新的内容，以显示新的 `String`。

此问题的解决方法很简单。如果 `Label` 未设置为自定义 `ContentTemplate` 值，请将 `Label` 替换为 `TextBlock` 并将其 `Text` 属性数据绑定到源字符串。

数据绑定属性	更新时间 (ms)
<code>Label.Content</code>	835
<code>TextBlock.Text</code>	242

## Hyperlink

`Hyperlink` 对象为内联级流内容元素，允许承载流内容中的超链接。

## 在一个 `TextBlock` 对象中合并超链接

通过在同一 `TextBlock` 中将多个 `Hyperlink` 元素组合在一起，可以优化它们的使用。这有助于最小化在应用程序中创建的对象的数量。例如，可显示多个超链接，如下所示：

以下标记示例演示多个用来显示超链接的 [TextBlock](#) 元素：

XAML

```
<!-- Hyperlinks in separate TextBlocks. -->
<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://www.msn.com">MSN
Home</Hyperlink>
</TextBlock>

<TextBlock Text=" | "/>

<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://my.msn.com">My
MSN</Hyperlink>
</TextBlock>
```

以下标记示例演示了一种更有效的超链接显示方式，这次使用单个 [TextBlock](#)：

XAML

```
<!-- Hyperlinks combined in the same TextBlock. -->
<TextBlock>
    <Hyperlink TextDecorations="None" NavigateUri="http://www.msn.com">MSN
Home</Hyperlink>

    <Run Text=" | " />

    <Hyperlink TextDecorations="None" NavigateUri="http://my.msn.com">My
MSN</Hyperlink>
</TextBlock>
```

## 仅在 [MouseEnter](#) 事件的超链接中显示下划线

[TextDecoration](#) 对象是可以添加到文本中的视觉装饰；但是会占用实例化的资源。如果大范围使用 [Hyperlink](#) 元素，请考虑仅在触发事件时显示下划线，例如 [MouseEnter](#) 事件。有关详细信息，请参阅[指定是否为超链接添加下划线](#)。

下图显示了 [MouseEnter](#) 事件如何触发带下划线的超链接：



以下标记示例演示使用和不使用下划线定义的 [Hyperlink](#)：

#### XAML

```
<!-- Hyperlink with default underline. -->
<Hyperlink NavigateUri="http://www.msn.com">
    MSN Home
</Hyperlink>

<Run Text=" | " />

<!-- Hyperlink with no underline. -->
<Hyperlink Name="myHyperlink" TextDecorations="None"
    MouseEnter="OnMouseEnter"
    MouseLeave="OnMouseLeave"
    NavigateUri="http://www.msn.com">
    My MSN
</Hyperlink>
```

下表演示显示 1000 个 [Hyperlink](#) 元素（使用和不使用下划线）的性能成本。

超链接	创建时间 (ms)	呈现时间 (ms)
使用下划线	289	1130
不使用下划线	299	776

## 文本格式设置功能

WPF 提供丰富的文本格式设置服务，如自动断字。这些服务可能会影响应用程序性能，应仅在需要时使用。

## 避免不必要的使用断字

自动断字功能查找文本行的连字符断点，并允许在 [TextBlock](#) 和 [FlowDocument](#) 对象的行中添加中断位置。默认禁用这些对象中的自动断字功能。通过将该对象的 [IsHyphenationEnabled](#) 属性设置为 `true`，可以启用此功能。但是，启用此功能会导致 WPF 启动组件对象模型 (COM) 互操作性，这可能会影响应用程序的性能。除非需要，否则不建议使用自动断字功能。

## 谨慎使用图形

[Figure](#) 元素代表在某页内容中绝对可定位的一部分流内容。在某些情况下，如果 [Figure](#) 的位置与已布局的内容发生冲突，则可能会导致整个页面自动重新格式化。可以通过将

[Figure](#) 元素彼此相邻分组或在固定页面大小方案中将其声明在内容顶部附近，来最大限度地减少不必要的重新格式化的可能性。

## 最佳段落

[FlowDocument](#) 对象的最佳段落功能对段落进行布局，以便尽可能均匀地分布空白区域。默认禁用最佳段落功能。你可以通过将对象的 [IsOptimalParagraphEnabled](#) 属性设置为 `true` 启用此功能。但是，启用此功能会影响应用程序的性能。除非需要，否则不建议使用最佳段落功能。

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [规划应用程序性能](#)
- [利用硬件](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [数据绑定](#)
- [其他性能建议](#)

# 优化性能：数据绑定

项目 · 2023/02/06

Windows Presentation Foundation (WPF) 数据绑定功能提供了一种简单一致的方法来呈现应用程序和与数据交互。元素能够以 CLR 对象和 XML 的形式绑定到各种数据源中的数据。

本主题提供数据绑定性能方面的建议。

## 如何解析数据绑定引用

在讨论数据绑定性能问题前，有必要了解 Windows Presentation Foundation (WPF) 数据绑定引擎如何解析用于绑定的对象引用。

Windows Presentation Foundation (WPF) 数据绑定的源可以是任意 CLR 对象。可绑定到 CLR 对象的属性、子属性或索引器。绑定引用是通过使用 Microsoft .NET Framework 反射或 [ICustomTypeDescriptor](#) 解析的。以下为解析用于绑定的对象引用的三种方法。

第一种方法涉及到使用反射。在这种情况下，会使用  [PropertyInfo](#) 对象来发现属性的特性并提供对属性元数据的访问。使用 [ICustomTypeDescriptor](#) 接口时，数据绑定引擎使用此接口来访问属性值。在对象没有一组静态属性的情况下，[ICustomTypeDescriptor](#) 接口尤为有用。

可以通过实现 [INotifyPropertyChanged](#) 接口或使用与 [TypeDescriptor](#) 关联的更改通知来提供属性更改通知。但是，实现属性更改通知的首选策略是使用 [INotifyPropertyChanged](#)。

如果源对象是 CLR 对象且源属性是 CLR 属性，则 Windows Presentation Foundation (WPF) 数据绑定引擎需要先在源对象上使用反射来获取 [TypeDescriptor](#)，然后再查询 [PropertyDescriptor](#)。从性能角度看，此反射操作序列可能非常耗时。

解析对象引用的第二种方法涉及到实现 [INotifyPropertyChanged](#) 接口的 CLR 源对象以及作为 CLR 属性的源属性。这种情况下，数据绑定引擎直接在源类型上使用反射，并获取所需的属性。这依然不是最佳方法，但相较于第一种方法，这种方法在工作集要求方面的成本低。

解析对象引用的第三种方法涉及到作为 [DependencyObject](#) 的源对象以及作为 [DependencyProperty](#) 的源属性。这种情况下，数据绑定引擎无需使用反射。属性引擎和数据绑定引擎共同独立地解析属性引用。这是解析用于数据绑定的对象引用的最佳方法。

下表比较了使用这三种方法对一千个 [TextBlock](#) 元素的 [Text](#) 属性进行数据绑定的速度。

绑定 TextBlock 的文本属性	绑定时间 (ms)	呈现时间 -- 包括绑定 (ms)
绑定到 CLR 对象的属性	115	314
绑定到实现 <a href="#">INotifyPropertyChanged</a> 的 CLR 对象的属性	115	305
绑定到 <a href="#">DependencyObject</a> 的 <a href="#">DependencyProperty</a> 。	90	263

## 绑定到大型 CLR 对象

数据绑定到具有数千个属性的单个 CLR 对象时会出现明显的性能影响。 可以通过将单个对象分成具有较少属性的多个 CLR 对象来最大限度地降低此影响。 下表列出了数据绑定到单个大型 CLR 对象和多个较小对象的绑定时间和呈现时间。

数据绑定 1000 个 TextBlock 对象	绑定时间 (ms)	呈现时间 -- 包括绑定 (ms)
绑定到具有 1000 个属性的 CLR 对象	950	1200
绑定到 1000 个具有 1 个属性的 CLR 对象	115	314

## 绑定到 ItemsSource

假设你有一个包含员工列表的 CLR [List<T>](#) 对象，并且你想在 [ListBox](#) 中显示此列表。 若要在这两个对象之间创建对应关系，应将员工列表绑定到 [ListBox](#) 的 [ItemsSource](#) 属性。 不妨再假设有一个加入组的新员工。 你可能会想，若要将这名新员工插入到绑定的 [ListBox](#) 值中，仅需将这名新员工添加到员工列表即可，数据绑定引擎应该会自动识别此更改。 这种看法并不正确；实际上，更改不会自动反映在 [ListBox](#) 中。 这是因为 CLR [List<T>](#) 对象不会自动引发集合更改事件。 若要使 [ListBox](#) 接受更改，你需重新创建员工列表并将它重新附加到 [ListBox](#) 的 [ItemsSource](#) 属性。 此解决方案尽管起作用，但是会随之产生很大的性能影响。 每当将 [ListBox](#) 的 [ItemsSource](#) 重新分配到新对象时，[ListBox](#) 会首先抛弃它先前的项，然后重新生成它的整个列表。 如果 [ListBox](#) 映射到复杂的 [DataTemplate](#)，则性能影响会被放大。

针对此问题的一个非常高效的解决方案是使员工列表成为 [ObservableCollection<T>](#)。  
[ObservableCollection<T>](#) 对象会引发数据绑定引擎可接收的更改通知。 此事件从 [ItemsControl](#) 添加或删除项，无需重新生成整个列表。

下表显示了添加一个新项时更新 [ListBox](#)（其中 UI 虚拟化处于关闭状态）所需的时间。 第一行中的数字表示 CLR [List<T>](#) 对象绑定到 [ListBox](#) 元素的 [ItemsSource](#) 的用时。 第二行中的数字表示 [ObservableCollection<T>](#) 绑定到 [ListBox](#) 元素的 [ItemsSource](#) 的用时。 请注意，使用 [ObservableCollection<T>](#) 数据绑定策略可节省大量时间。

数据绑定 ItemsSource	1 个项的更新时间 (ms)
绑定到 CLR <code>List&lt;T&gt;</code> 对象	1656
绑定到 <code>ObservableCollection&lt;T&gt;</code>	20

## 将 `IList` 绑定到 `ItemsControl` 而非 `IEnumerable`

如果可以在将 `IList<T>` 还是 `IEnumerable` 绑定到 `ItemsControl` 对象之间进行选择 , 请选择 `IList<T>` 对象。 将 `IEnumerable` 绑定到 `ItemsControl` 会强制 WPF 创建一个包装器 `IList<T>` 对象 , 这意味着性能会因第二个对象不必要的开销而受到影响。

## 请勿仅为数据绑定而将 CLR 对象转换为 XML。

WPF 使你可以数据绑定到 XML 内容 ; 但是 , 数据绑定到 XML 内容的速度比数据绑定到 CLR 对象的速度更慢。 如果目的仅在于数据绑定 , 请勿将 CLE 对象数据转换为 XML。

## 另请参阅

- [优化 WPF 应用程序性能](#)
- [规划应用程序性能](#)
- [利用硬件](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [文本](#)
- [其他性能建议](#)
- [数据绑定概述](#)
- [演练 : 在 WPF 应用程序中缓存应用程序数据](#)

# 优化性能：控件

项目 · 2022/09/27

Windows Presentation Foundation (WPF) 包含大多数 Windows 应用程序中使用的许多常见用户界面 (UI) 组件。本主题包含提高 UI 性能的技术。

## 显示大型数据集

[ListView](#) 和 [ComboBox](#) 等 WPF 控件用于显示应用程序中的项列表。如果要显示的列表较大，则应用程序性能可能受到影响。这是因为标准布局系统会为每个与列表控件关联的项创建布局容器，并计算其布局大小和位置。通常，无需同时显示所有项，而是显示子集，然后用户滚动浏览列表。这种情况下，使用 UI 虚拟化具有意义，这意味着生成项容器，且项的关联布局计算会推迟，直到项可见时。

UI 虚拟化是列表控件的一个重要方面。不应混淆 UI 虚拟化和数据虚拟化。UI 虚拟化在内存中仅存储可见项，但在数据绑定方案中会存储内存中的整个数据结构。相反，数据虚拟化仅存储内存中屏幕上可见的数据项。

默认情况下，对于 [ListView](#) 和 [ListBox](#) 控件，在其列表项绑定到数据时会启用 UI 虚拟化。通过将 [VirtualizingStackPanel.IsVirtualizing](#) 附加属性设置为 `true` 可启用 [TreeView](#) 虚拟化。若要对派生自 [ItemsControl](#) 的自定义控件或使用 [StackPanel](#) 类的现有控件（例如 [ComboBox](#)）启用 UI 虚拟化，可将 [ItemsPanel](#) 设置为 [VirtualizingStackPanel](#)，并将 [IsVirtualizing](#) 设置为 `true`。遗憾的是，可对没有实现虚拟化的控件禁用 UI 虚拟化。以下为禁用 UI 虚拟化的条件列表。

- 项容器直接添加到 [ItemsControl](#)。例如，如果应用程序显式将 [ListBoxItem](#) 对象添加到 [ListBox](#)，则 [ListBox](#) 不会虚拟化 [ListBoxItem](#) 对象。
- [ItemsControl](#) 中的项容器类型不同。例如，使用 [Separator](#) 对象的 [Menu](#) 无法实现项回收，其原因在于 [Menu](#) 包含 [Separator](#) 和 [MenuItem](#) 类型的对象。
- 将 [CanContentScroll](#) 设置为 `false`。
- 将 [IsVirtualizing](#) 设置为 `false`。

虚拟化项容器时需要考虑到一个重要因素，即是否具有与属于项的项容器相关联的其他状态信息。这种情况下，必须保存其他状态。例如，你可能具有 [Expander](#) 控件中所含的一个项，且 [IsExpanded](#) 状态绑定到项的容器而非项自身。对新项重复使用此容器时，[IsExpanded](#) 的当前值会用于新项。此外，旧项会丢失正确的 [IsExpanded](#) 值。

当前，无任何 WPF 控件提供对数据虚拟化的内置支持。

## 容器回收

对继承自 [ItemsControl](#) 的控件而言，.NET Framework 3.5 SP1 中增添的 UI 虚拟化优化是容器回收，此优化还可提高滚动性能。使用 UI 虚拟化的 [ItemsControl](#) 在填充时，会为每个滚动到视图的项创建项容器并销毁每个滚动到视图外的项的项容器。容器回收可使控件对不同的数据项重复使用现有项容器，从而在用户滚动 [ItemsControl](#) 时不会不断创建和销毁项容器。通过将 [VirtualizationMode](#) 附加属性设置为 [Recycling](#)，可选择启用项回收。

任何支持虚拟化的 [ItemsControl](#) 皆可使用容器回收。有关如何在 [ListBox](#) 上启用容器回收的示例，请参阅[提高 ListBox 的滚动性能](#)。

## 支持双向虚拟化

[VirtualizingStackPanel](#) 提供内置的单向（水平或垂直）UI 虚拟化支持。若要对控件使用双向虚拟化，必须实现一个扩展 [VirtualizingStackPanel](#) 类的自定义面板。

[VirtualizingStackPanel](#) 类公开一些虚拟方法，例如 [OnViewportSizeChanged](#)、[LineUp](#)、[PageUp](#) 和 [MouseWheelUp](#)。通过这些虚拟方法可检测列表可见部分中的更改并相应进行处理。

## 优化模板

可视化树包含应用程序中所有可视元素。除直接创建的对象外，它还包括由于模板扩展而产生的对象。例如，创建 [Button](#) 时，也会获取可视化树中的 [ClassicBorderDecorator](#) 和 [ContentPresenter](#) 对象。如果尚未优化模板控件，则要在可视化树中创建大量不必要的额外对象。有关可视化树的详细信息，请参见 [WPF 图形呈现概述](#)。

## 延迟滚动

默认情况下，用户拖动滚动条上的滚动块时，内容视图会不断更新。如果控件中滚动较慢，请考虑使用延迟滚动。在延迟滚动中，仅在用户释放滚动块时才会更新内容。

若要实现延迟滚动，请将 [IsDeferredScrollingEnabled](#) 属性设置为 `true`。

[IsDeferredScrollingEnabled](#) 为附加属性，可在 [ScrollViewer](#) 以及其控件模板中具有 [ScrollViewer](#) 的任何控件上进行设置。

## 实现性能功能的控件

下表列出了显示数据的常见控件及其性能功能支持。有关如何启用这些功能的信息，请参阅先前章节。

控制	虚拟化	容器回收	延迟滚动
ComboBox	可启用	可启用	可启用
ContextMenu	可启用	可启用	可启用
DocumentViewer	不可用	不可用	可启用
ListBox	默认	可启用	可启用
ListView	默认	可启用	可启用
TreeView	可启用	可启用	可启用
ToolBar	不可用	不可用	可启用

### ① 备注

有关如何在 TreeView 上启用虚拟化和容器回收的示例，请参阅[提高 TreeView 的性能](#)。

## 另请参阅

- [布局](#)
- [布局和示例](#)
- [数据绑定](#)
- [控件](#)
- [样式设置和模板化](#)
- [演练：在 WPF 应用程序中缓存应用程序数据](#)

# 优化性能：其他建议

项目 · 2023/02/06

本主题提供优化 WPF 应用程序性能这一节中各主题内容之外的性能改进建议。

本主题包含以下各节：

- 画笔的不透明度与元素的不透明度
- 导航到对象
- 对大型 3D 图面进行命中测试
- CompositionTarget.Rendering 事件
- 避免使用 ScrollBarVisibility=Auto
- 配置字体缓存服务以缩短启动时间

## 画笔的不透明度与元素的不透明度

使用 Brush 设置元素的 Fill 或 Stroke 时，最好设置 Brush.Opacity 值而不是设置元素的 Opacity 属性。修改元素的 Opacity 属性可能会导致 WPF 创建一个临时表面。

## 导航到对象

NavigationWindow 对象派生自 Window 并通过内容导航支持扩展它，主要是通过聚合 NavigationService 和日志。可以通过指定统一资源标识符 (URI) 或对象来更新 NavigationWindow 的工作区。以下示例演示了这两种方法：

C#

```
private void buttonGoToUri(object sender, RoutedEventArgs args)
{
    navWindow.Source = new Uri("NewPage.xaml", UriKind.RelativeOrAbsolute);
}

private void buttonGoNewObject(object sender, RoutedEventArgs args)
{
    NewPage nextPage = new NewPage();
    nextPage.InitializeComponent();
    navWindow.Content = nextPage;
}
```

每个 [NavigationWindow](#) 对象都有一个日志，用于记录用户在该窗口中的导航历史记录。日志的作用之一是允许用户回溯他们执行的步骤。

使用统一资源标识符 (URI) 进行导航时，日志仅存储统一资源标识符 (URI) 引用。这意味着，每次重新访问该页时都会动态地重新构造该页，根据页面的复杂程度，此过程可能会非常耗时。在这种情况下，虽然占用的日志存储较少，但用于重建该页的时间可能会较长。

使用对象进行导航时，日志会存储对象的整个可视化树。这意味着，每次重新访问该页时，无需重新构造即可立即呈现该页。在这种情况下，虽然占用的日志存储较多，但重建页面所用的时间较短。

使用 [NavigationWindow](#) 对象时，需要记住日志支持如何影响应用程序的性能。有关详细信息，请参阅[导航概述](#)。

## 对大型 3D 图面进行命中测试

就 CPU 消耗而言，对大型 3D 图面进行命中测试是一项非常占用资源的操作。3D 图面显示动画效果时更是如此。如果不需要对这些图面进行命中测试，请禁用命中测试。派生自 [UIElement](#) 的对象可以通过将 [IsHitTestVisible](#) 属性设置为 `false` 来禁用命中测试。

## CompositionTarget.Rendering 事件

[CompositionTarget.Rendering](#) 事件导致 WPF 持续进行动画处理。使用此事件时，应尽可能将其分离。

## 避免使用 ScrollBarVisibility=Auto

尽可能避免对 [HorizontalScrollBarVisibility](#) 和 [VerticalScrollBarVisibility](#) 属性使用 [ScrollBarVisibility.Auto](#) 值。这些属性针对 [RichTextBox](#)、[ScrollViewer](#) 和 [TextBox](#) 对象定义，并作为 [ListBox](#) 对象的附加属性。而是将 [ScrollBarVisibility](#) 设置为 [Disabled](#)、[Hidden](#) 或 [Visible](#)。

[Auto](#) 值适用于空间有限且仅在必要时才应显示滚动条的情况。例如，与包含数百行文本的 [TextBox](#) 相比，将此 [ScrollBarVisibility](#) 值用于包含 30 项的 [ListBox](#) 可能很有用。

## 配置字体缓存服务以缩短启动时间

WPF 字体缓存服务在 WPF 应用程序之间共享字体数据。如果该服务尚未运行，则你运行的第一个 WPF 应用程序将启动该服务。如果你使用的是 Windows Vista，可以将

“Windows Presentation Foundation (WPF) 字体缓存 3.0.0.0”服务从“手动”（默认）设置为“自动（延迟启动）”，以减少 WPF 应用程序的初始启动时间。

## 另请参阅

- [规划应用程序性能](#)
- [利用硬件](#)
- [布局和示例](#)
- [二维图形和图像处理](#)
- [对象行为](#)
- [应用程序资源](#)
- [文本](#)
- [数据绑定](#)
- [动画提示和技巧](#)

# 应用程序启动时间

项目 · 2023/02/06

启动 WPF 应用程序所需的时间可能存在极大差异。本主题介绍用于减少 Windows Presentation Foundation (WPF) 应用程序假设启动时间和实际启动时间的各种技巧。

## 了解冷启动和热启动

冷启动发生在系统重启后第一次启动应用程序时，或启动应用程序、将其关闭，然后在很长一段时间后再次启动应用程序时。应用程序启动时，如果所需的页面（代码、静态数据、注册表等）不在 Windows 内存管理器的待机列表中，会发生页面错误。需要磁盘访问权限，以便将这些页面加载到内存中。

当已将主要公共语言运行时 (CLR) 组件的大多数页面加载到内存中时，则发生热启动，这样可节省宝贵的磁盘访问时间。这就是为什么再次运行托管的应用程序时，该程序的启动速度更快的原因。

## 实现初始屏幕

为应对在启动应用程序后到显示第一个 UI 期间出现重大的、不可避免的延迟的情况，请使用“初始屏幕”优化假设的启动时间。通过此方法，在用户启动应用程序后，几乎可以立即显示图像。当应用程序准备好显示其第一个 UI 时，初始屏幕将淡化。从 .NET Framework 3.5 SP1 开始，可以使用 [SplashScreen](#) 类来实现初始屏幕。有关详细信息，请参阅[将初始屏幕添加到 WPF 应用程序](#)。

还可以通过使用本机 Win32 图形来实现自己的初始屏幕。在调用 [Run](#) 方法之前显示实现。

## 分析启动代码

确定冷启动慢的原因。可能与磁盘 I/O 有关，但这并非唯一的原因。一般情况下，应将外部资源（例如网络、Web 服务或磁盘）的使用率最小化。

在测试之前，验证没有其他正在运行的应用程序或服务会使用托管代码或 WPF 代码。

重新启动后，立即启动 WPF 应用程序，并决定用于显示的时间。如果应用程序的所有后续启动（热启动）相较之下快很多，则冷启动问题很可能是 I/O 所致。

如果应用程序的冷启动问题与 I/O 无关，则冷启动慢的原因很可能是应用程序在启动时会执行一些耗时较长的初始化进程或计算、等待某些事件完成或需要大量的 JIT 编译。以下

章节对其中某些情况进行了详细介绍。

## 优化模块加载

使用工具（例如进程资源管理器 (Procexp.exe) 和 Tlist.exe）来确定应用程序加载的模块。命令 `Tlist <pid>` 显示由某一进程加载的所有模块。

例如，如果没有连接到 Web，并且发现已加载 System.Web.dll，则应用程序中存在引用此程序集的模块。检查以确定该引用是必要的。

如果应用程序具有多个模块，则请将其合并为单个模块。此方法要求较少的 CLR 程序集加载开销。较少的程序集还意味着 CLR 维护较少的状态。

## 延迟初始化操作

请考虑将初始化代码延迟至主应用程序窗口呈现之后。

请注意，可能会在类构造函数内执行初始化，如果初始化代码引用其他类，则可能会导致级联效应（执行许多类构造函数）。

## 避免应用程序配置

请考虑避免应用程序配置。例如，如果某一应用程序具有简单的配置要求，并且具有严格的启动时间目标，则注册表项或简单的 INI 文件可能是更快的启动替代方法。

## 利用 GAC

如果未在全局程序集缓存 (GAC) 中安装程序集，则将出现由强命名程序集的哈希验证和 Ngen 映像验证（如果计算机上该程序集的本机映像可用）导致的延迟。对于安装到 GAC 中的所有程序集，会跳过强命名验证。有关详细信息，请参阅 [Gacutil.exe \(全局程序集缓存工具\)](#)。

## 使用 Ngen.exe

请考虑在应用程序上使用本机映像生成器 (Ngen.exe)。使用 Ngen.exe 意味着通过减少 CPU 消耗来实现更多的磁盘访问，因为由 Ngen.exe 生成的本机映像可能会比 MSIL 映像要大。

若要减少热启动时间，则应在应用程序上始终使用 Ngen.exe，因为这可以避免应用程序代码的 JIT 编译的 CPU 成本。

在某些冷启动方案中，使用 Ngen.exe 也会有所帮助。这是因为不需要加载 JIT 编译器 (mscorjit.dll)。

同时具有 Ngen 和 JIT 模块可能会导致最差的效果。这是因为必须加载 mscojit.dll，且当 JIT 编译器处理代码时，当编译器读取程序集的元数据时，必须访问 Ngen 映像中的许多页面。

## Ngen 和 ClickOnce

计划用于部署应用程序的方法在加载期间也会造成影响。ClickOnce 应用程序部署不支持 Ngen。如果决定对应用程序使用 Ngen.exe，则需要使用其他部署机制，如 Windows Installer。

有关详细信息，请参阅 [Ngen.exe（本机映像生成器）](#)。

## 基址重置和 DLL 地址冲突

如果使用 Ngen.exe，请注意，本机映像加载到内存中时可能会发生基址重置。如果未在 DLL 首选基址加载该 DLL（由于已分配该地址范围），则 Windows 加载程序将在另一地址加载该 DLL，这可能是一个耗时的操作。

可以使用虚拟地址转储 (Vadump.exe) 工具来检查是否存在其中所有页面都为私有的模块。如果是这种情况，则可能已将该模块的基址重置到另一地址。因此，不能共享其页面。

有关如何设置基址的详细信息，请参阅 [Ngen.exe（本机映像生成器）](#)。

## 优化验证码

验证码验证会增加启动时间。验证码签名的程序集必须通过证书颁发机构 (CA) 验证。此验证可能需要较长时间，因为它可能会要求多次连接到网络，以便下载当前证书吊销列表。它还确保受信任的根路径上存在完整有效的证书链。在加载程序集时，这可能会导致几秒钟的延迟。

请考虑在客户端计算机上安装 CA 证书，或尽可能避免使用验证码。如果知道应用程序不需要发布服务器证据，则无需支付签名验证的费用。

从 .NET Framework 3.5 开始，存在一个允许绕过验证码验证的配置选项。为执行此操作，将以下设置添加到 app.exe.config 文件：

XML

```
<configuration>
  <runtime>
    <generatePublisherEvidence enabled="false"/>
  </runtime>
</configuration>
```

有关详细信息，请参阅 [<generatePublisherEvidence> 元素](#)。

## 在 Windows Vista 上比较性能

Windows Vista 中的内存管理器具有一种名为 SuperFetch 的技术。 SuperFetch 可随时间推移分析内存使用情况模式，从而确定特定用户的最优内存内容。 它会持续工作，不间断地维护此内容。

这种方法不同于 Windows XP 中使用的预提取技术，后者将数据预先加载到内存，而不分析使用情况模式。 随着时间的推移，如果用户在 Windows Vista 上频繁使用 WPF 应用程序，则可能改善应用程序的冷启动时间。

## 有效使用 AppDomains

如果可能，将程序集加载到非特定于域的代码区域，以确保本机映像（如果存在）会在应用程序中创建的所有 AppDomain 中使用。

为获得最佳性能，通过减少跨域调用强制实施高效跨域通信。 如果可能，请使用不带参数或具有基元类型参数的调用。

## 使用 NeutralResourcesLanguage 特性

使用 [NeutralResourcesLanguageAttribute](#) 指定 [ResourceManager](#) 的非特定区域性。 此方法可避免程序集查找失败。

## 将 BinaryFormatter 类用于序列化

如果必须使用序列化，请使用 [BinaryFormatter](#) 类，而不是 [XmlSerializer](#) 类。 在 mscorelib.dll 程序集的基类库 (BCL) 中实现 [BinaryFormatter](#) 类。 在 System.Xml.dll 程序集中实现 [XmlSerializer](#)，这可能是额外要加载的 DLL。

如果必须使用 [XmlSerializer](#) 类，则可以通过预生成序列化程序集获得更好的性能。

## 将 ClickOnce 配置为在启动后检查更新

如果应用程序使用 ClickOnce，请通过将 ClickOnce 配置为在应用程序启动后检查部署站点是否有更新，从而避免在启动时访问网络。

如果使用 XAML 浏览器应用程序 (XBAP) 模型，请记住，即使 XBAP 已存在于 ClickOnce 缓存中，ClickOnce 也会检查部署站点是否存在更新。有关详细信息，请参阅 [ClickOnce Security and Deployment](#)。

## 将 PresentationFontCache 服务配置为自动启动

在重新启动后，要运行的第一个 WPF 应用程序是 PresentationFontCache 服务。该服务会缓存系统字体、改进字体访问，并提高整体性能。在启动服务时会产生开销，某些受控环境中也存在开销，请考虑将服务配置为在系统重启时自动启动。

## 以编程方式设置数据绑定

请考虑在 [OnActivated](#) 方法中以编程方式设置主窗口的 [DataContext](#)，而不是使用 XAML 以声明的方式来进行设置。

## 另请参阅

- [SplashScreen](#)
- [AppDomain](#)
- [NeutralResourcesLanguageAttribute](#)
- [ResourceManager](#)
- 向 WPF 应用程序添加初始屏幕
- [Ngen.exe \( 本机映像生成器 \)](#)
- [<generatePublisherEvidence> 元素](#)

# 演练：在 WPF 应用程序中缓存应用程序数据

项目 • 2023/02/06

缓存可以将数据存储在内存中以便快速访问。再次访问数据时，应用程序可以从缓存获取数据，而不是从原始源检索数据。这可改善性能和可伸缩性。此外，数据源暂时不可用时，缓存可提供数据。

.NET Framework 提供了使你能够在 .NET Framework 应用程序中使用缓存的类。这些类位于 [System.Runtime.Caching](#) 命名空间中。

## ① 备注

`System.Runtime.Caching` 是 .NET Framework 4 中的新命名空间。此命名空间使所有 .NET Framework 应用程序都可以使用缓存。旧版 .NET Framework 仅在 `System.Web` 命名空间中提供缓存，因此，需要 ASP.NET 类上的一个依赖项。

本演练介绍如何使用 .NET Framework 中提供的缓存功能作为 Windows Presentation Foundation (WPF) 应用程序的一部分。在本演练中，你将缓存文本文件的内容。

本演练演示以下任务：

- 创建 WPF 应用程序项目。
- 添加对 .NET Framework 4 的引用。
- 初始化缓存。
- 添加包含文本文件内容的缓存条目。
- 为缓存条目提供逐出策略。
- 监视缓存文件的路径，并就受监视项的变化通知缓存实例。

## 先决条件

若要完成本演练，你将需要：

- Visual Studio 2010。
- 包含少量文本的文本文件。（你将在消息框中显示文本文件的内容。）演练中演示的代码假定你正在处理以下文件：

```
c:\cache\cacheText.txt
```

不过，你可以使用任何文本文件并对本演练中的代码稍做更改。

## 创建 WPF 应用程序项目

首先创建一个 WPF 应用程序项目。

### 创建 WPF 应用程序

1. 启动 Visual Studio。
2. 在“文件”菜单中，单击“新建”，然后单击“新建项目”。  
显示“新项目”对话框。
3. 在“已安装的模板”下，选择要使用的编程语言（Visual Basic 或 Visual C#）。
4. 在“新建项目”对话框中，选择“WPF 应用程序”。

#### ① 备注

如果未显示“WPF 应用程序”模板，请确保以支持 WPF 的 .NET Framework 版本为目标。在“新建项目”对话框中，从列表中选择 .NET Framework 4。

5. 在“名称”文本框中，输入项目的名称。例如，可以输入 WPFCaching。
6. 选择“为解决方案创建目录”复选框。
7. 单击“确定”。

WPF 设计器会在“设计”视图中打开并显示 MainWindow.xaml 文件。Visual Studio 会创建“我的项目”文件夹、Application.xaml 文件和 MainWindow.xaml 文件。

## 以 .NET Framework 为目标并添加对缓存程序集的引用

默认情况下，WPF 应用程序以 .NET Framework 4 客户端配置文件为目标。若要在 WPF 应用程序中使用 [System.Runtime.Caching](#) 命名空间，应用程序必须以 .NET Framework 4（而不是 .NET Framework 4 客户端配置文件）为目标，并且必须包含对该命名空间的引用。

因此，下一步是更改 .NET Framework 目标并添加对 `System.Runtime.Caching` 命名空间的引用。

### ① 备注

更改 .NET Framework 目标的过程在 Visual Basic 项目和 Visual C# 项目中有所不同。

## 在 Visual Basic 中更改目标 .NET Framework

1. 在解决方案资源管理器中，右键单击项目名称，然后单击“属性”。

随即显示应用程序的属性窗口。

2. 单击“编译”选项卡。

3. 在窗口底部，单击“高级编译选项...”。

随即显示“高级编译器设置”对话框。

4. 在“目标框架(所有配置)”列表中，选择“.NET Framework 4”。（请勿选择“.NET Framework 4 客户端配置文件”。）

5. 单击“确定”。

随即显示“目标框架更改”对话框。

6. 在“目标框架更改”对话框中，单击“是”。

项目将关闭，然后重新打开。

7. 按照以下步骤添加对缓存程序集的引用：

a. 在解决方案资源管理器中，右键单击项目名称，然后单击“添加引用”。

b. 选择“.NET”选项卡，选择 `System.Runtime.Caching`，然后单击“确定”。

## 在 Visual C# 项目中更改目标 .NET Framework

1. 在解决方案资源管理器中，右键单击项目名称，然后单击“属性”。

随即显示应用程序的属性窗口。

2. 单击“应用程序”选项卡。

3. 在“目标框架”列表中，选择“.NET Framework 4”。（请勿选择“.NET Framework 4 客户端配置文件”。）
4. 按照以下步骤添加对缓存程序集的引用：
  - a. 右键单击“引用”文件夹，然后单击“添加引用”。
  - b. 选择“.NET”选项卡，选择 `System.Runtime.Caching`，然后单击“确定”。

## 向 WPF 窗口添加按钮

接下来，添加按钮控件并为按钮的 `click` 事件创建事件处理程序。稍后将添加代码，以便在单击按钮时，缓存并显示文本文件的内容。

### 添加按钮控件

1. 在解决方案资源管理器中，双击 `MainWindow.xaml` 文件将其打开。
2. 从工具箱的“常用 WPF 控件”下，将 `Button` 控件拖到 `MainWindow` 窗口中。
3. 在“属性”窗口中，将 `Button` 控件的 `Content` 设置为“获取缓存”。

## 初始化缓存并缓存条目

接下来，添加代码以执行以下任务：

- 创建缓存类实例，即实例化新的 `MemoryCache` 对象。
- 指定缓存使用 `HostFileChangeMonitor` 对象来监视文本文件中的更改。
- 读取文本文件，并将其内容缓存为缓存条目。
- 显示缓存文本文件的内容。

### 创建缓存对象

1. 双击刚刚添加的按钮，以便在 `MainWindow.xaml.cs` 或 `MainWindow.xaml.vb` 文件中创建事件处理程序。
2. 在文件顶部（类声明之前），添加以下 `Imports` (Visual Basic) 或 `using` (C#) 语句：

C#

```
using System.Runtime.Caching;
using System.IO;
```

3. 在事件处理程序中，添加以下代码以实例化缓存对象：

C#

```
ObjectCache cache = MemoryCache.Default;
```

[ObjectCache](#) 类是提供内存中对象缓存的内置类。

4. 添加以下代码以读取名为 `filecontents` 的缓存条目的内容：

C#

```
string fileContents = cache["filecontents"] as string;
```

5. 添加以下代码以检查名为 `filecontents` 的缓存条目是否存在：

C#

```
if (fileContents == null)
{
}
```

如果指定的缓存条目不存在，则必须读取文本文件并将其作为缓存条目添加到缓存中。

6. 在 `if/then` 块中，添加以下代码以创建一个新的 [CacheItemPolicy](#) 对象，该对象指定缓存条目在 10 秒后过期。

C#

```
CacheItemPolicy policy = new CacheItemPolicy();
policy.AbsoluteExpiration = DateTimeOffset.Now.AddSeconds(10.0);
```

如果未提供逐出或过期信息，则默认为 [InfiniteAbsoluteExpiration](#)，这意味着缓存条目永远不会仅基于绝对时间过期。相反，缓存条目仅在存在内存压力时才会过期。作为最佳做法，应始终显式提供绝对过期或可调过期。

7. 在 `if/then` 块内上一步添加的代码后面，添加以下代码以创建要监视的文件路径集合，并将文本文件的路径添加到集合中：

C#

```
List<string> filePaths = new List<string>();
filePaths.Add("c:\\cache\\cacheText.txt");
```

### ① 备注

如果要使用的文本文件不是 `c:\cache\cacheText.txt`，请指定要使用的文本文件的路径。

- 在上一步添加的代码后面，添加以下代码以将新的 `HostFileChangeMonitor` 对象添加到缓存条目的更改监视器集合中：

C#

```
policy.ChangeMonitors.Add(new HostFileChangeMonitor(filePaths));
```

`HostFileChangeMonitor` 对象监视文本文件的路径，并在发生更改时通知缓存。在此示例中，如果文件内容发生更改，缓存条目将过期。

- 在上一步添加的代码后面，添加以下代码以读取文本文件的内容：

C#

```
fileContents = File.ReadAllText("c:\\cache\\cacheText.txt") + "\\n" +
DateTime.Now;
```

系统添加了日期和时间时间戳，以便你能够看到缓存条目何时过期。

- 在上一步添加的代码后面，添加以下代码以将文件的内容作为 `CachedItem` 实例插入缓存对象：

C#

```
cache.Set("filecontents", fileContents, policy);
```

可通过将之前创建的 `CachedItemPolicy` 对象作为参数传递，指定有关应如何逐出缓存条目的信息。

- 在 `if/then` 块后面添加以下代码，以在消息框中显示缓存的文件内容：

C#

```
MessageBox.Show(fileContents);
```

12. 在“生成”菜单中，单击“生成 WPF Caching”以生成项目。

## 在 WPF 应用程序中测试缓存

现在可以测试此应用程序。

### 在 WPF 应用程序中测试缓存

1. 按 Ctrl+F5 运行应用程序。

随即显示 `MainWindow` 窗口。

2. 单击“获取缓存”。

来自文本文件的缓存内容显示在消息框中。请注意文件上的时间戳。

3. 关闭消息框，然后再次单击“获取缓存”。

时间戳不变。这表明显示的是缓存内容。

4. 等待 10 秒或更长时间，然后再次单击“获取缓存”。

这次显示一个新的时间戳。这表明该策略让缓存条目过期并显示新的缓存内容。

5. 在文本编辑器中，打开创建的文本文件。暂时不要进行任何更改。

6. 关闭消息框，然后再次单击“获取缓存”。

再次注意时间戳。

7. 对文本文件进行更改，然后保存文件。

8. 关闭消息框，然后再次单击“获取缓存”。

此消息框包含来自文本文件的更新内容和新时间戳。这表明主机文件更改监视器在你更改文件时立即逐出缓存条目，即使绝对超时期限尚未到期也是如此。

#### (!) 备注

你可以将逐出时间增加到 20 秒或更长时间，以便有更多时间对文件进行更改。

## 代码示例

完成本演练后，你创建的项目的代码将类似于以下示例。

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Runtime.Caching;
using System.IO;

namespace WPFCaching
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, RoutedEventArgs e)
        {

            ObjectCache cache = MemoryCache.Default;
            string fileContents = cache["filecontents"] as string;

            if (fileContents == null)
            {
                CacheItemPolicy policy = new CacheItemPolicy();
                policy.AbsoluteExpiration =
                    DateTimeOffset.Now.AddSeconds(10.0);

                List<string> filePaths = new List<string>();
                filePaths.Add("c:\\cache\\cacheText.txt");

                policy.ChangeMonitors.Add(new
                    HostFileChangeMonitor(filePaths));

                // Fetch the file contents.
                fileContents = File.ReadAllText("c:\\cache\\cacheText.txt")
                + "\\n" + DateTimeOffset.Now.ToString();
            }
        }
    }
}
```

```
        cache.Set("filecontents", fileContents, policy);
    }
    MessageBox.Show(fileContents);
}
}
```

## 另请参阅

- [MemoryCache](#)
- [ObjectCache](#)
- [System.Runtime.Caching](#)
- [.NET Framework 应用程序中的缓存](#)

# 线程处理模型

项目 · 2023/02/06

WPF 开发人员无需编写需要使用多个线程的界面。由于多线程程序既复杂又难以调试，因此当存在单线程解决方案时，应避免使用多线程程序。

但是，无论构建得多好，没有任何 UI 框架能为每种问题都提供单线程解决方案。WPF 虽然在这方面有近乎完美的表现，但某些情况下，仍需要使用多线程来改进用户界面 (UI) 响应能力或应用程序性能。基于上文所述的背景材料，本文对上述情况进行探讨，然后通过对一些低级别的细节进行讨论作出总结。

## ① 备注

本主题介绍使用 `BeginInvoke` 方法进行异步调用的线程处理。你也可以通过调用 `InvokeAsync` 方法来执行异步调用，该方法使用 `Action` 或 `Func<TResult>` 作为参数。`InvokeAsync` 方法返回具有 `Task` 属性的 `DispatcherOperation` 或 `DispatcherOperation<TResult>`。可以将 `await` 关键字与 `DispatcherOperation` 或相关联的 `Task` 配合使用。如果需要同步等待 `DispatcherOperation` 或 `DispatcherOperation<TResult>` 返回的 `Task`，请调用 `DispatcherOperationWait` 扩展方法。调用 `Task.Wait` 将导致死锁。有关使用 `Task` 执行异步操作的详细信息，请参阅[基于任务的异步编程](#)。`Invoke` 方法还具有以 `Action` 或 `Func<TResult>` 为参数的重载。可使用 `Invoke` 方法，通过传入 `Action` 或 `Func<TResult>` 委托来执行同步调用。

## 概述和调度程序

通常为 UI。当 UI 线程接收输入、处理事件、绘制屏幕和运行应用程序代码时，呈现线程通过隐藏方式在后台高效运行。大多数应用程序使用单个 UI 线程，不过在某些情况下，最好使用多个线程。我们将稍后通过示例对此进行讨论。

UI 线程在称为 `Dispatcher` 的对象内对工作项进行排队。`Dispatcher` 基于优先级选择工作项，并运行每一个工作项直到完成。每个 UI 线程必须具有至少一个 `Dispatcher`，且每个 `Dispatcher` 都可精确地在一个线程中执行工作项。

若要生成响应迅速、用户友好的应用程序，诀窍在于通过保持工作项小型化来最大化 `Dispatcher` 吞吐量。这样一来，工作项就不会停滞在 `Dispatcher` 队列中，因等待处理而过时。输入和响应间任何可察觉的延迟都会让用户不满。

那么，UI 线程如何自由地处理 `Dispatcher` 队列中的项？大型操作完成后，它可以将其结果报告回 UI 线程以进行显示。

传统而言，Windows 允许 UI 元素仅由创造它们的线程访问。这意味着，负责长时间运行任务的后台线程无法在任务完成时更新文本框。Windows 这么做的目的是确保 UI 组件的完整性。如果在绘制过程中后台线程更新了列表框的内容，则此列表框看起来可能会很奇怪。

WPF 具有内置互相排斥机制，此机制能强制执行这种协调。WPF 中的大多数类都派生自 [DispatcherObject](#)。构造时，[DispatcherObject](#) 会存储对 [Dispatcher](#)（它链接到当前正在运行的线程）的引用。实际上，[DispatcherObject](#) 与创建它的线程相关联。在程序执行期间，[DispatcherObject](#) 可以调用它的公共 [VerifyAccess](#) 方法。[VerifyAccess](#) 检查与当前线程相关联的 [Dispatcher](#)，并将其与构造期间存储的 [Dispatcher](#) 引用相比较。如果它们不匹配，[VerifyAccess](#) 会引发异常。系统会在属于 [DispatcherObject](#) 的每个方法的开头调用 [VerifyAccess](#)。

如果可以修改 UI 的线程只有一个，后台线程将如何与用户进行交互？后台线程可请求 UI 线程代表自己来执行操作。它通过向 UI 线程的 [Dispatcher](#) 注册工作项来实现此目的。[Dispatcher](#) 类为注册工作项提供两种方法：[Invoke](#) 和 [BeginInvoke](#)。这两种方法都计划一个用于执行的委托。[Invoke](#) 是一个同步调用，也就是说，在 UI 线程真正执行完委托之前，它不会返回。[BeginInvoke](#) 是异步的，它会立即返回。

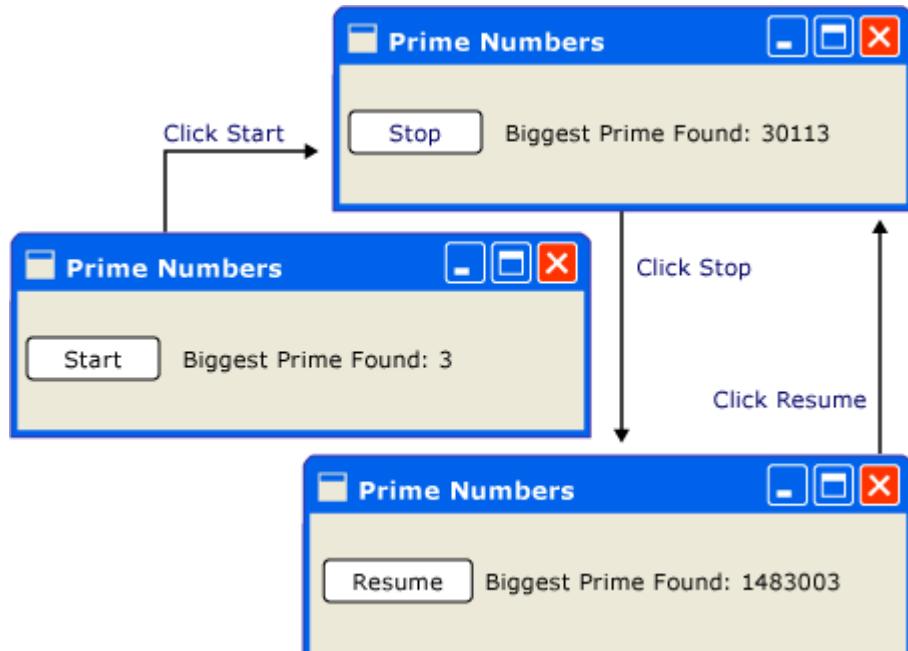
[Dispatcher](#) 按优先级对其队列中的元素排序。向 [Dispatcher](#) 队列添加元素时，可以指定十个级别。这些优先级均在 [DispatcherPriority](#) 枚举中维护。有关 [DispatcherPriority](#) 级别的详细信息可以在 Windows SDK 文档中找到。

## 实际线程：示例

### 具有长时间运行计算的单线程应用程序

在等待由响应用户交互而生成的事件时，大多数图形用户界面 (GUI) 在大多数时间处于空闲状态。通过精心编程，可建设性地使用这些空闲时间，且不会影响 UI 的响应能力。UI 线程。这意味着，必须确保定期返回 [Dispatcher](#)，以便在过时之前处理挂起的输入事件。

请考虑以下示例：



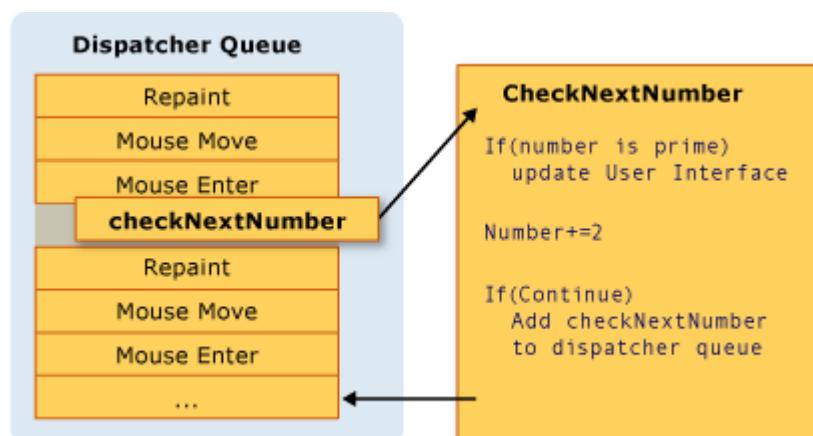
这个简单的应用程序从 3 开始向上计数以搜索质数。 用户单击“开始”按钮时，开始执行搜索。 当程序查找到一个质数时，它将根据其发现内容更新用户界面。 用户可随时停止搜索。

尽管十分简单，但对质数的搜索可以永远持续下去，这会带来一些问题。如果在按钮的单击事件处理程序中处理整个搜索，UI 线程将永远没有机会处理其他事件。UI 将无法响应输入，也无法处理消息。它将永远不会重绘，也永远不会响应按钮单击。

可以在单独的线程中搜索质数，但这样的话，我们需要处理一些同步问题。通过单线程方法，可以直接更新列出所找到的最大质数的标签。

如果将计算任务分解为可管理的多个区块，则可以定期返回 [Dispatcher](#)，并处理事件。WPF 就有机会重绘和处理输入。

划分计算和事件处理之间的处理时间的最佳方式是从 [Dispatcher](#) 管理计算。通过使用 [BeginInvoke](#) 方法，可以在从中绘制 UI 事件的同一队列中计划质数检查。在我们的示例中，一次仅计划一个质数检查。完成质数检查后，立即计划下一个检查。仅当处理挂起的 UI 事件后，此检查才会继续。



Microsoft Word 通过此机制完成拼写检查。 拼写检查是在后台利用 UI 线程的空闲时间完成的。 我们来看一看代码。

下列示例显示了创建用户界面的 XAML。

#### XAML

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Prime Numbers" Width="260" Height="75"
    >
    <StackPanel Orientation="Horizontal" VerticalAlignment="Center" >
        <Button Content="Start"
            Click="StartOrStop"
            Name="startStopButton"
            Margin="5,0,5,0"
            />
        <TextBlock Margin="10,5,0,0">Biggest Prime Found:</TextBlock>
        <TextBlock Name="bigPrime" Margin="4,5,0,0">3</TextBlock>
    </StackPanel>
</Window>
```

以下示例显示了代码隐藏。

#### C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        public delegate void NextPrimeDelegate();

        //Current number to check
        private long num = 3;

        private bool continueCalculating = false;

        public Window1() : base()
        {
            InitializeComponent();
        }

        private void StartOrStop(object sender, EventArgs e)
        {
            if (continueCalculating)
```

```
        {
            continueCalculating = false;
            startStopButton.Content = "Resume";
        }
        else
        {
            continueCalculating = true;
            startStopButton.Content = "Stop";
            startStopButton.Dispatcher.BeginInvoke(
                DispatcherPriority.Normal,
                new NextPrimeDelegate(CheckNextNumber));
        }
    }

    public void CheckNextNumber()
    {
        // Reset flag.
        NotAPrime = false;

        for (long i = 3; i <= Math.Sqrt(num); i++)
        {
            if (num % i == 0)
            {
                // Set not a prime flag to true.
                NotAPrime = true;
                break;
            }
        }

        // If a prime number.
        if (!NotAPrime)
        {
            bigPrime.Text = num.ToString();
        }

        num += 2;
        if (continueCalculating)
        {
            startStopButton.Dispatcher.BeginInvoke(
                System.Windows.Threading.DispatcherPriority.SystemIdle,
                new NextPrimeDelegate(this.CheckNextNumber));
        }
    }

    private bool NotAPrime = false;
}
}
```

以下示例显示了 [Button](#) 的事件处理程序。

C#

```

private void StartOrStop(object sender, EventArgs e)
{
    if (continueCalculating)
    {
        continueCalculating = false;
        startStopButton.Content = "Resume";
    }
    else
    {
        continueCalculating = true;
        startStopButton.Content = "Stop";
        startStopButton.Dispatcher.BeginInvoke(
            DispatcherPriority.Normal,
            new NextPrimeDelegate(CheckNextNumber));
    }
}

```

除更新 [Button](#) 上的文本外，此处理程序还负责通过向 [Dispatcher](#) 队列添加委托，计划首个质数检查。在此事件处理程序完成其工作后一段时间，[Dispatcher](#) 将选择执行此委托。

如前文所述，[BeginInvoke](#) 是用于计划委托执行的 [Dispatcher](#) 成员。在这种情况下，选择 [SystemIdle](#) 优先级。仅当没有要处理的重要事件时，[Dispatcher](#) 才会执行此委托。UI 响应能力比数字检查更重要。我们还传递了一个表示数字检查例程的新委托。

C#

```

public void CheckNextNumber()
{
    // Reset flag.
    NotAPrime = false;

    for (long i = 3; i <= Math.Sqrt(num); i++)
    {
        if (num % i == 0)
        {
            // Set not a prime flag to true.
            NotAPrime = true;
            break;
        }
    }

    // If a prime number.
    if (!NotAPrime)
    {
        bigPrime.Text = num.ToString();
    }

    num += 2;
    if (continueCalculating)
    {

```

```

        startStopButton.Dispatcher.BeginInvoke(
            System.Windows.Threading.DispatcherPriority.SystemIdle,
            new NextPrimeDelegate(this.CheckNextNumber));
    }

    private bool NotAPrime = false;
}

```

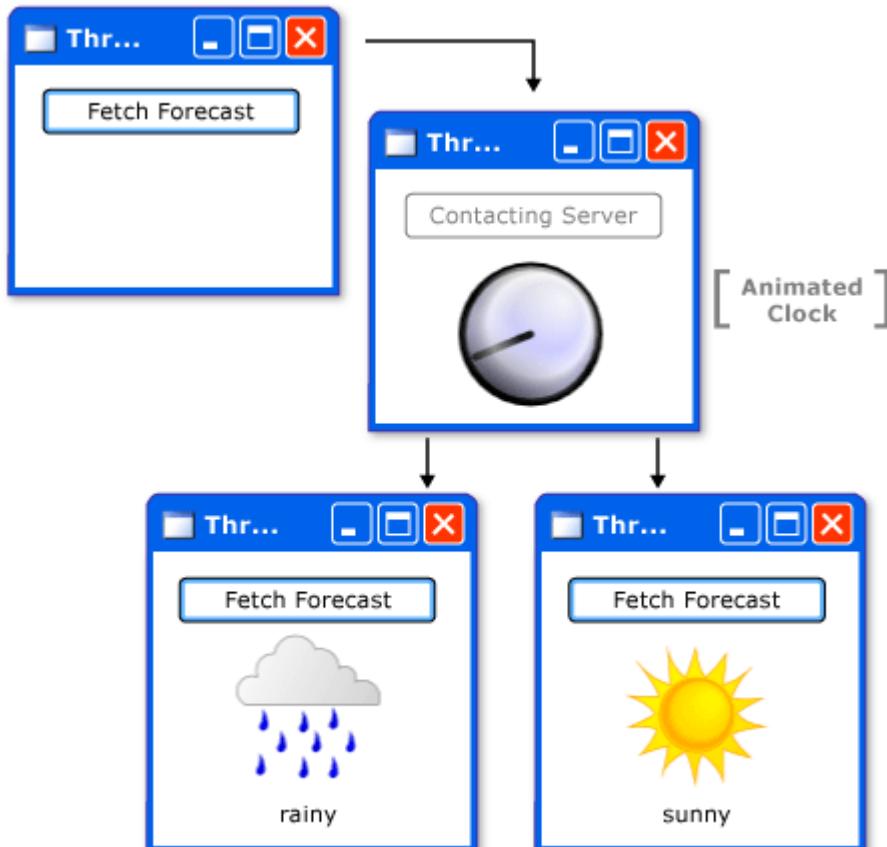
此方法检查下一个奇数是否是质数。如果是质数，此方法将直接更新 `bigPrimeTextBlock`，以反映此发现。可以如此操作的原因是，该计算发生在用于创建组件的相同线程中。如果选择使用单独的线程来进行计算，将必须使用更复杂的同步机制，并在 UI 线程中执行更新。我们将在下一步中演示这种情况。

有关此示例的完整源代码，请参阅[具有长时间运行计算的单线程应用程序示例](#)

## 使用后台线程处理阻塞操作

在图形应用程序中处理阻塞操作可能很困难。我们不希望从事件处理程序调用阻塞方法，因为应用程序可能看上去冻结。可以使用单独的线程来处理这些操作，但在完成操作后，必须使用 UI 线程进行同步，因为无法从工作线程直接修改 GUI。可使用 `Invoke` 或 `BeginInvoke` 将委托插入 UI 线程的 `Dispatcher`。最终，将通过可修改 UI 元素的权限来执行这些委托。

在本例中，我们模拟了一个检索天气预报的远程过程调用。我们使用单独的工作线程来执行此调用，并且计划在调用完成时在 UI 线程的 `Dispatcher` 中更新方法。



C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Media;
using System.Windows.Media.Animation;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {
        // Delegates to be used in placking jobs onto the Dispatcher.
        private delegate void NoArgDelegate();
        private delegate void OneArgDelegate(String arg);

        // Storyboards for the animations.
        private Storyboard showClockFaceStoryboard;
        private Storyboard hideClockFaceStoryboard;
        private Storyboard showWeatherImageStoryboard;
        private Storyboard hideWeatherImageStoryboard;

        public Window1(): base()
        {
            InitializeComponent();
        }

        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // Load the storyboard resources.
            showClockFaceStoryboard =
                (Storyboard)this.Resources["ShowClockFaceStoryboard"];
            hideClockFaceStoryboard =
                (Storyboard)this.Resources["HideClockFaceStoryboard"];
            showWeatherImageStoryboard =
                (Storyboard)this.Resources["ShowWeatherImageStoryboard"];
            hideWeatherImageStoryboard =
                (Storyboard)this.Resources["HideWeatherImageStoryboard"];
        }

        private void ForecastButtonHandler(object sender, RoutedEventArgs e)
        {
            // Change the status image and start the rotation animation.
            fetchButton.IsEnabled = false;
            fetchButton.Content = "Contacting Server";
            weatherText.Text = "";
            hideWeatherImageStoryboard.Begin(this);

            // Start fetching the weather forecast asynchronously.
            NoArgDelegate fetcher = new NoArgDelegate(
```

```

        this.FetchWeatherFromServer);

    fetcher.BeginInvoke(null, null);
}

private void FetchWeatherFromServer()
{
    // Simulate the delay from network access.
    Thread.Sleep(4000);

    // Tried and true method for weather forecasting - random
numbers.

    Random rand = new Random();
    String weather;

    if (rand.Next(2) == 0)
    {
        weather = "rainy";
    }
    else
    {
        weather = "sunny";
    }

    // Schedule the update function in the UI thread.
    tomorrowsWeather.Dispatcher.BeginInvoke(
        System.Windows.Threading.DispatcherPriority.Normal,
        new OneArgDelegate(UpdateUserInterface),
        weather);
}

private void UpdateUserInterface(String weather)
{
    //Set the weather image
    if (weather == "sunny")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "SunnyImageSource"];
    }
    else if (weather == "rainy")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "RainingImageSource"];
    }

    //Stop clock animation
    showClockFaceStoryboard.Stop(this);
    hideClockFaceStoryboard.Begin(this);

    //Update UI text
    fetchButton.IsEnabled = true;
    fetchButton.Content = "Fetch Forecast";
    weatherText.Text = weather;
}

```

```

private void HideClockFaceStoryboard_Completed(object sender,
    EventArgs args)
{
    showWeatherImageStoryboard.Begin(this);
}

private void HideWeatherImageStoryboard_Completed(object sender,
    EventArgs args)
{
    showClockFaceStoryboard.Begin(this, true);
}
}
}

```

以下是一些需要注意的详细信息。

- 创建按钮处理程序

C#

```

private void ForecastButtonHandler(object sender, RoutedEventArgs e)
{
    // Change the status image and start the rotation animation.
    fetchButton.IsEnabled = false;
    fetchButton.Content = "Contacting Server";
    weatherText.Text = "";
    hideWeatherImageStoryboard.Begin(this);

    // Start fetching the weather forecast asynchronously.
    NoArgDelegate fetcher = new NoArgDelegate(
        this.FetchWeatherFromServer);

    fetcher.BeginInvoke(null, null);
}

```

单击按钮时，会显示时钟绘图并开始对其进行动画处理。禁用该按钮。调用新线程中的 `FetchWeatherFromServer` 方法，然后返回，这允许 `Dispatcher` 在我们等待收集天气预报信息时处理事件。

- 获取天气

C#

```

private void FetchWeatherFromServer()
{
    // Simulate the delay from network access.
    Thread.Sleep(4000);

    // Tried and true method for weather forecasting - random numbers.
    Random rand = new Random();
    String weather;
}

```

```

    if (rand.Next(2) == 0)
    {
        weather = "rainy";
    }
    else
    {
        weather = "sunny";
    }

    // Schedule the update function in the UI thread.
    tomorrowsWeather.Dispatcher.BeginInvoke(
        System.Windows.Threading.DispatcherPriority.Normal,
        new OneArgDelegate(UpdateUserInterface),
        weather);
}

```

为简便起见，本例中没有任何网络代码。通过使新线程进入休眠状态四秒钟，模拟网络访问的延迟。此时，原始 UI 线程仍在运行并对事件作出响应。为了对此进行演示，我们让动画保持运行状态，最小化和最大化按钮也继续工作。

延迟结束且我们随机选择了天气预报后，就可以报告回 UI 线程。若要执行此操作，需要使用 UI 线程的 [Dispatcher](#)，在该线程中计划对 [UpdateUserInterface](#) 的调用。将描述天气的字符串传递给此计划方法调用。

- 更新 UI

```

C#

private void UpdateUserInterface(String weather)
{
    //Set the weather image
    if (weather == "sunny")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "SunnyImageSource"];
    }
    else if (weather == "rainy")
    {
        weatherIndicatorImage.Source = (ImageSource)this.Resources[
            "RainingImageSource"];
    }

    //Stop clock animation
    showClockFaceStoryboard.Stop(this);
    hideClockFaceStoryboard.Begin(this);

    //Update UI text
    fetchButton.IsEnabled = true;
    fetchButton.Content = "Fetch Forecast";
}

```

```
    weatherText.Text = weather;
}
```

当 UI 线程中的 `Dispatcher` 有空时，它将执行计划的对 `UpdateUserInterface` 的调用。此方法停止时钟动画，并选择一张映像用于描述天气。它将显示此映像，并还原“获取预报”按钮。

## 多窗口、多线程

某些 WPF 应用程序要求多个顶层窗口。通过单个线程/`Dispatcher` 组合来管理多个窗口是完全可以接受的，但有时多线程可以做得更好。尤其当这些窗口中的某一个将有可能要独占线程时，更是如此。

Windows 资源管理器以这种方式工作。每个新资源管理器窗口都属于原始进程，但它是在独立线程的控件下创建的。

通过使用 `WPFFrame` 控件可以显示网页。我们可以轻松创建一个简单的 Internet Explorer 替代项。让我们从一个重要功能开始：打开新资源管理器窗口的能力。当用户单击“新建窗口”按钮时，我们将在单独的线程中启动窗口的副本。这样一来，在其中一个窗口中的长时间运行或阻塞操作将不会锁定其他窗口。

在实际情况下，Web 浏览器模型自身拥有复杂的线程模型。由于大多数读者都熟悉它，所以我们选择它。

以下示例显示了代码。

XAML

```
<Window x:Class="SDKSamples.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MultiBrowse"
    Height="600"
    Width="800"
    Loaded="OnLoaded"
    >
<StackPanel Name="Stack" Orientation="Vertical">
    <StackPanel Orientation="Horizontal">
        <Button Content="New Window"
            Click="NewWindowHandler" />
        <TextBox Name="newLocation"
            Width="500" />
        <Button Content="GO!"
            Click="Browse" />
    </StackPanel>
    <Frame Name="placeHolder"
        Width="800"
```

```
        Height="550"></Frame>
    </StackPanel>
</Window>
```

C#

```
using System;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Threading;
using System.Threading;

namespace SDKSamples
{
    public partial class Window1 : Window
    {

        public Window1() : base()
        {
            InitializeComponent();
        }

        private void OnLoaded(object sender, RoutedEventArgs e)
        {
            placeHolder.Source = new Uri("http://www.msn.com");
        }

        private void Browse(object sender, RoutedEventArgs e)
        {
            placeHolder.Source = new Uri(newLocation.Text);
        }

        private void NewWindowHandler(object sender, RoutedEventArgs e)
        {
            Thread newWindowThread = new Thread(new
ThreadStart(ThreadStartingPoint));
            newWindowThread.SetApartmentState(ApartmentState.STA);
            newWindowThread.IsBackground = true;
            newWindowThread.Start();
        }

        private void ThreadStartingPoint()
        {
            Window1 tempWindow = new Window1();
            tempWindow.Show();
            System.Windows.Threading.Dispatcher.Run();
        }
    }
}
```

此代码中的以下线程段对我们来说是最有趣的：

C#

```
private void NewWindowHandler(object sender, RoutedEventArgs e)
{
    Thread newWindowThread = new Thread(new
    ThreadStart(ThreadStartingPoint));
    newWindowThread.SetApartmentState(ApartmentState.STA);
    newWindowThread.IsBackground = true;
    newWindowThread.Start();
}
```

当单击“新建窗口”按钮时，将调用该方法。 它创建了一个新线程，并以异步方式启动。

C#

```
private void ThreadStartingPoint()
{
    Window1 tempWindow = new Window1();
    tempWindow.Show();
    System.Windows.Threading.Dispatcher.Run();
}
```

此方法是新线程的起点。 我们在此线程的控件下创建了一个新窗口。 WPF 自动创建新的 [Dispatcher](#) 来管理新线程。 若要使窗口功能化，我们要做的是启动 [Dispatcher](#)。

## 技术详细信息和疑难点

### 使用线程处理编写组件

Microsoft .NET Framework 开发人员指南介绍了组件如何向其客户端公开异步行为的模式（请参阅[基于事件的异步模式概述](#)）。例如，假设我们要将 `FetchWeatherFromServer` 方法打包到可重用的非图形组件。遵循标准 Microsoft .NET Framework 模式，它将如下所示。

C#

```
public class WeatherComponent : Component
{
    //gets weather: Synchronous
    public string GetWeather()
    {
        string weather = "";

        //predict the weather

        return weather;
    }
}
```

```

//get weather: Asynchronous
public void GetWeatherAsync()
{
    //get the weather
}

public event GetWeatherCompletedEventHandler GetWeatherCompleted;
}

public class GetWeatherCompletedEventArgs : AsyncCompletedEventArgs
{
    public GetWeatherCompletedEventArgs(Exception error, bool canceled,
        object userState, string weather)
        :
        base(error, canceled, userState)
    {
        _weather = weather;
    }

    public string Weather
    {
        get { return _weather; }
    }
    private string _weather;
}

public delegate void GetWeatherCompletedEventHandler(object sender,
    GetWeatherCompletedEventArgs e);

```

`GetWeatherAsync` 将使用上述的技术之一（如创建后台线程）来以异步方式工作，而非阻止调用线程。

此模式最重要的部分之一是在调用 `MethodNameAsync` 方法开始工作的同一线程上调用 `MethodNameCompleted` 方法。可通过存储 `CurrentDispatcher` 使用 WPF 轻松完成此操作，但非图形组件只能在 WPF 应用程序中使用，不能在 Windows 窗体或 ASP.NET 程序中使用。

`DispatcherSynchronizationContext` 类可满足此需求 - 将其视作与其他 UI 框架配合使用的 `Dispatcher` 的简化版本。

C#

```

public class WeatherComponent2 : Component
{
    public string GetWeather()
    {
        return fetchWeatherFromServer();
    }

    private DispatcherSynchronizationContext requestingContext = null;

```

```

public void GetWeatherAsync()
{
    if (requestingContext != null)
        throw new InvalidOperationException("This component can only
handle 1 async request at a time");

    requestingContext =
(DispatcherSynchronizationContext)DispatcherSynchronizationContext.Current;

    NoArgDelegate fetcher = new
NoArgDelegate(this.fetchWeatherFromServer);

    // Launch thread
    fetcher.BeginInvoke(null, null);
}

private void RaiseEvent(GetWeatherCompletedEventArgs e)
{
    if (GetWeatherCompleted != null)
        GetWeatherCompleted(this, e);
}

private string fetchWeatherFromServer()
{
    // do stuff
    string weather = "";

    GetWeatherCompletedEventArgs e =
        new GetWeatherCompletedEventArgs(null, false, null, weather);

    SendOrPostCallback callback = new SendOrPostCallback(DoEvent);
    requestingContext.Post(callback, e);
    requestingContext = null;

    return e.Weather;
}

private void DoEvent(object e)
{
    //do stuff
}

public event GetWeatherCompletedEventHandler GetWeatherCompleted;
public delegate string NoArgDelegate();
}

```

## 嵌套泵

有时无法完全锁定 UI 线程。让我们考虑一下 `MessageBox` 类的 `Show` 方法。在用户单击“确定”按钮之前，`Show` 不会返回。但是，它却会创建一个窗口，该窗口为了获得交互性而必须具有消息循环。在等待用户单击“确定”时，原始应用程序窗口将不会响应用户

的输入。但是，它将继续处理绘制消息。当被覆盖和被显示时，原始窗口将重绘其本身。



一些线程必须负责消息框窗口。WPF 可以为消息框窗口创建新线程，但此线程无法在原始窗口中绘制禁用的元素（请回忆之前所讨论的互相排斥）。WPF 使用嵌套消息处理系统。[Dispatcher](#) 类包括一个名为 [PushFrame](#) 的特殊方法，它存储应用程序的当前执行点，然后启动一个新的消息循环。当嵌套消息循环结束后，将在原始 [PushFrame](#) 调用之后继续执行。

在此情况下，[PushFrame](#) 将在调用 [MessageBox.Show](#) 时维护程序上下文，并且它将启动一个新的消息循环，用于重绘后台窗口，并处理对消息框窗口的输入。当用户单击“确定”并清除弹出窗口时，嵌套循环将退出，并在调用 [Show](#) 后继续控制。

## 过时的路由事件

引发事件时，WPF 中的路由事件系统会通知整个树。

XAML

```
<Canvas MouseLeftButtonDown="handler1"
        Width="100"
        Height="100"
        >
    <Ellipse Width="50"
              Height="50"
              Fill="Blue"
              Canvas.Left="30"
              Canvas.Top="50"
              MouseLeftButtonDown="handler2"
            />
</Canvas>
```

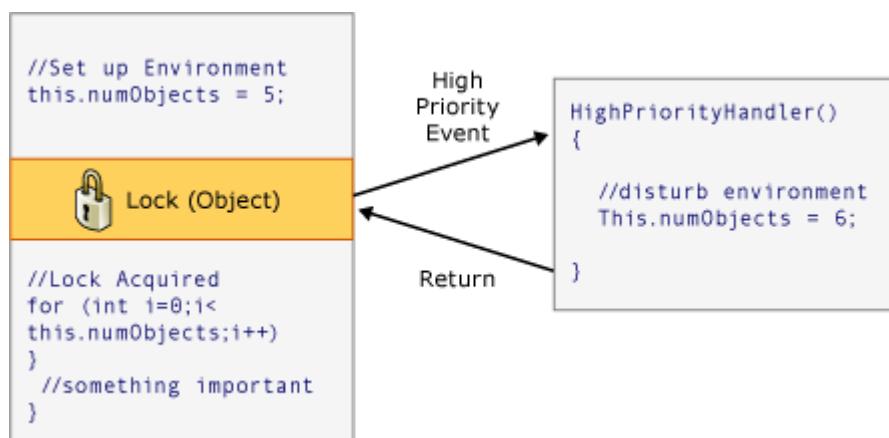
在椭圆形上按下鼠标左键时，将执行 `handler2`。`handler2` 完成后，事件将传递到 [Canvas](#) 对象，后者使用 `handler1` 对其进行处理。仅当 `handler2` 没有显式标记事件对象为已处理时，才会发生这种情况。

`handler2` 有可能花费大量时间来处理此事件。 `handler2` 有可能使用 `PushFrame` 来启动嵌套消息循环，并在数小时内不会返回任何内容。如果在此消息循环完成时，`handler2` 尚未将事件标记为已处理，该事件将沿树向上传递（即使它很旧）。

## 重新进入和锁定

公共语言运行时 (CLR) 的锁定机制与人们所设想的完全不同；可能有人以为在请求锁定时，线程将完全停止操作。实际上，该线程将继续接收和处理高优先级的消息。这样有助于防止死锁，并使接口最低限度地响应，但这样做有可能引入细微 bug。绝大多数时间里，你无需知晓有关这点的任何情况，但在极少数情况下（通常涉及 Win32 窗口消息或 COM STA 组件），可能需要知道这一点。

大部分接口在生成过程中并未考虑线程安全问题，这是因为开发人员在开发过程中假定 UI 绝不会由一个以上的线程访问。在此情况下，该单个线程可能在意外情况下更改环境，造成不良影响，这些影响应由 `DispatcherObject` 互相排斥机制来解决。请看下面的伪代码：



大多数情况下这都没有问题，但在某些时候 WPF 中的异常重入确实会造成严重问题。因此在某些关键时刻，WPF 调用 `DisableProcessing`，这会更改该线程的锁定指令，以使用 WPF 无重入锁定，而非常规 CLR 锁定。

那么，为何 CLR 团队选择这种行为？它与 COM STA 对象和完成线程有关。在对一个对象进行垃圾回收时，其 `Finalize` 方法运行在专用终结器线程之上，而非 UI 线程上。这其中就存在问题，因为在 UI 线程上创建的 COM STA 对象只能在 UI 线程上释放。CLR 相当于 `BeginInvoke`（在此例中使用 Win32 的 `SendMessage`）。但如果 UI 线程正忙，终结器线程被停止，COM STA 对象无法被释放，这将造成严重的内存泄漏。因此，CLR 团队通过严格的调用，使锁定以这种方式工作。

WPF 的任务是在不重新引入内存泄漏的情况下，避免异常的重入，因此我们不阻止各个位置的重入。

## 另请参阅

- 具有长时间运行计算的单线程应用程序示例 ↴

# WPF 非托管 API 参考

项目 • 2023/02/06

Windows Presentation Foundation (WPF) 库公开了许多仅供内部使用的非托管函数。它们不应通过用户代码调用。

## 本节内容

[Activate 函数](#)

[CreateDispatchSTAForwarder 函数](#)

[Deactivate 函数](#)

[ForwardTranslateAccelerator 函数](#)

[LoadFromHistory 函数](#)

[ProcessUnhandledException 函数](#)

[SaveToHistory 函数](#)

[SetFakeActiveWindow 函数](#)

## 另请参阅

- [高级](#)

# Activate 函数 ( WPF 非托管 API 参考 )

项目 • 2022/09/27

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
void Activate(
    const ActivateParameters* pParameters,
    __deref_out_ecount(1) LPUNKNOWN* ppInner,
);
```

## 参数

pParameters

指向窗口激活参数的指针。

ppInner

指向包含指向 IOleDocument 对象的指针的单元素缓冲区的地址的指针。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)

# CreateIDispatchSTAForwarder 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于线程和窗口管理。

## 语法

C++

```
HRESULT CreateIDispatchSTAForwarder(
    __in IDispatch *pDispatchDelegate,
    __deref_out IDispatch **ppForwarder
)
```

## 参数

### 属性值/返回值

pDispatchDelegate

指向 `IDispatch` 接口的指针。

ppForwarder

指向 `IDispatch` 接口地址的指针。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)

# Deactivate 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
void Deactivate()
```

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)

# ForwardTranslateAccelerator 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
HRESULT ForwardTranslateAccelerator(
    MSG* pMsg,
    VARIANT_BOOL appUnhandled
)
```

## 参数

pMsg

指向消息的指针。

appUnhandled

`true` 当应用有机会处理输入消息，但尚未处理它时；否则 `false`。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- WPF 非托管 API 参考

# LoadFromHistory 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
HRESULT LoadFromHistory_export(
    IStream* pHistoryStream,
    IBindCtx* pBindCtx
)
```

## 参数

pHistoryStream

指向历史记录信息流的指针。

pBindCtx

指向绑定上下文的指针。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- WPF 非托管 API 参考

# ProcessUnhandledException 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于异常处理。

## 语法

C++

```
void __stdcall ProcessUnhandledException(
    _in_ecount(1) BSTR errorMsg
)
```

## 参数

errorMsg

错误消息。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)

# SaveToHistory 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
HRESULT SaveToHistory(
    __in_ecount(1) IStream* pHistoryStream
)
```

## 参数

pHistoryStream

指向历史记录流的指针。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：

在 .NET Framework 3.0 和 3.5 中：PresentationHostDLL.dll

在 .NET Framework 4 及更高版本中：PresentationHost\_v0400.dll

.NET Framework 版本：自 3.0 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)

# SetFakeActiveWindow 函数 ( WPF 非托管 API 参考 )

项目 • 2023/02/06

此 API 支持 Windows Presentation Foundation (WPF) 基础结构，不应在代码中直接使用。

由 Windows Presentation Foundation (WPF) 基础结构用于窗口管理。

## 语法

C++

```
void __stdcall SetFakeActiveWindow(  
    HWND hwnd  
)
```

## 参数

hwnd

一个窗口句柄。

## 要求

平台：请参阅 [.NET Framework 系统要求](#)。

DLL：PresentationHost\_v0400.dll

.NET Framework 版本：自 4 起可用

## 另请参阅

- [WPF 非托管 API 参考](#)