

WPF 基础知识

Windows Presentation Foundation (WPF) 是下一代显示系统，用于生成能带给用户震撼视觉体验的 Windows 客户端应用程序。使用 WPF，您可以创建广泛的独立应用程序以及浏览器承载的应用程序。

WPF 的核心是一个与分辨率无关并且基于向量的呈现引擎，旨在利用现代图形硬件的优势。WPF 通过一整套应用程序开发功能扩展了这个核心，这些功能包括 可扩展应用程序标记语言 (XAML)、控件、数据绑定、布局、二维和三维图形、动画、样式、模板、文档、媒体、文本和版式。WPF 包含在 Microsoft .NET Framework 中，使您能够生成融入了 .NET Framework 类库的其他元素的应用程序。

为了支持某些更强大的 WPF 功能并简化编程体验，WPF 包括了更多编程构造，这些编程构造增强了属性和事件：依赖项属性和路由事件。有关依赖项属性的更多信息，请参见依赖项属性概述。有关路由事件的更多信息，请参见路由事件概述。

这种外观和行为的分离具有以下优点：

- 1 降低了开发和维护成本，因为外观特定的标记并没有与行为特定的代码紧密耦合。
- 2 开发效率更高，因为设计人员可以在开发人员实现应用程序行为的同时实现应用程序的外观。
- 3 可以使用多种设计工具实现和共享 XAML 标记，以满足应用程序开发参与者的要求：Microsoft Expression Blend 提供了适合设计人员的体验，而 Visual Studio 2005 针对开发人员。
- 4 WPF 应用程序的全球化和本地化大大简化（请参见 WPF 全球化和本地化概述）。

在运行时，WPF 将标记中定义的元素和属性转换为 WPF 类的实例。例如，Window 元素被转换为 Window 类的实例，该类的 Title 属性 (Property) 是 Title 属性 (Attribute) 的值。

注意在 constructor 中 Call: InitializeComponent();

x:Class 属性用于将标记与代码隐藏类相关联。InitializeComponent 是从代码隐藏类的构造函数中调用的，用于将标记中定义的 UI 与代码隐藏类相合并。（生成应用程序时将为您生成 InitializeComponent，因此您不需要手动实现它。）x:Class 和 InitializeComponent 的组合确保您的实现无论何时创建都能得到正确的初始化。

.NET Framework、System.Windows、标记和代码隐藏构成了 WPF 应用程序开发体验的基础

窗口：WPF 对话框：MessageBox、OpenFileDialog、SaveFileDialog 和 PrintDialog。

WPF 提供了以下两个选项作为替代导航宿主：

- Frame，用于承载页面或窗口中可导航内容的孤岛。
- NavigationWindow，用于承载整个窗口中的可导航内容。

启动：StartupUri="MainWindow.xaml" /> 此标记是独立应用程序的应用程序定义，并指示 WPF 创建一个在应用程序启动时自动打开 MainWindow 的 Application 对象。

WPF 控件一览

此处列出了内置的 WPF 控件。

- 按钮: [Button](#) 和 [RepeatButton](#)。
- 对话框: [OpenFileDialog](#)、[PrintDialog](#) 和 [SaveFileDialog](#)。
- 数字墨迹: [InkCanvas](#) 和 [InkPresenter](#)。
- 文档: [DocumentViewer](#)、[FlowDocumentPageViewer](#)、[FlowDocumentReader](#)、[FlowDocumentScrollViewer](#) 和 [StickyNoteControl](#)。
- 输入: [TextBox](#)、[RichTextBox](#) 和 [PasswordBox](#)。
- 布局: [Border](#)、[BulletDecorator](#)、[Canvas](#)、[DockPanel](#)、[Expander](#)、[Grid](#)、[GridView](#)、[GridSplitter](#)、[GroupBox](#)、[Panel](#)、[ResizeGrip](#)、[Separator](#)、[ScrollBar](#)、[ScrollViewer](#)、[StackPanel](#)、[Thumb](#)、[Viewbox](#)、[VirtualizingStackPanel](#)、[Window](#) 和 [WrapPanel](#)。
- 媒体: [Image](#)、[MediaElement](#) 和 [SoundPlayerAction](#)。
- 菜单: [ContextMenu](#)、[Menu](#) 和 [ToolBar](#)。
- 导航: [Frame](#)、[Hyperlink](#)、[Page](#)、[NavigationWindow](#) 和 [TabControl](#)。
- 选择: [CheckBox](#)、[ComboBox](#)、[ListBox](#)、[TreeView](#)、[RadioButton](#) 和 [Slider](#)。
- 用户信息: [AccessText](#)、[Label](#)、[Popup](#)、[ProgressBar](#)、[StatusBar](#)、[TextBlock](#) 和 [ToolTip](#)。

输入和命令 :控件通常检测和响应用户输入。WPF 输入系统使用直接事件和路由事件来支持文本输入、焦点管理和鼠标定位。有关更多信息,请参见[输入概述](#)。

布局系统的基础是相对定位,它提高了适应窗口和显示条件变化的能力。此外,布局系统还管理控件之间的协商以确定布局。协商过程分为两步:第一步,控件向父控件通知它所需的位置和大小;第二步,父控件通知该控件它可以具有多大空间

- [Canvas](#): 子控件提供其自己的布局。
- [DockPanel](#): 子控件与面板的边缘对齐。
- [Grid](#): 子控件按行和列放置。
- [StackPanel](#): 子控件垂直或水平堆叠。
- [VirtualizingStackPanel](#): 子控件被虚拟化,并沿水平或垂直方向排成一行。
- [WrapPanel](#): 子控件按从左到右的顺序放置,如果当前行中的控件数多于该空间所允许的控件数,则换至下一行

由父控件实现的、供子控件使用的属性是一种 WPF 构造,称为“附加属性”

为了简化应用程序开发,WPF 提供了一个数据绑定引擎以自动执行这些步骤。数据绑定引擎的核心单元是 [Binding](#) 类,它的任务是将控件(绑定目标)绑定到数据对象(绑定源)。下图说明了这种关系。



WPF 数据绑定引擎还提供了其他支持，包括验证、排序、筛选和分组。此外，当标准 WPF 控件显示的 UI 不合适时，数据绑定还支持使用数据模板为绑定的数据创建自定义 UI。

WPF 引进了一组广泛的、可伸缩且灵活的图形功能，它们具有以下优点：

- **与分辨率和设备无关的图形。** WPF 图形系统的基本度量单位是与设备无关的像素，它等于一英寸的 1/96，而不管实际的屏幕分辨率是多少，为与分辨率和设备无关的呈现提供了基础。每个与设备无关的像素都会自动缩放，以符合呈现该像素的系统上的每英寸点数 (dpi) 设置。
- **更高的精度。** WPF 坐标系是使用双精度浮点数字测量的，而不是使用单精度浮点数字。转换值和不透明度值也以双精度表示。WPF 还支持广泛的颜色域 (sRGB)，并为管理来自不同颜色空间的输入提供完整的支持。
- **高级图形和动画支持。** WPF 通过为您管理动画场景简化了图形编程；您不需要担心场景处理、呈现循环和双线性内插算法。此外，WPF 还提供了命中测试支持和全面的 alpha 合成支持。
- **硬件加速。** WPF 图形系统利用了图形硬件的优势来最小化 CPU 使用率。

Path 对象可用于绘制闭合或开放形状、多线形状，甚至曲线形状。

Geometry 对象可用于对二维图形数据进行剪裁、命中测试和呈现。

WPF 二维功能的子集包括渐变、位图、绘图、视频绘制、旋转、缩放和扭曲等视觉效果。这些都可以使用画笔完成；下图演示了某些示例。

WPF 动画支持可以使控件变大、旋转、调节和淡化，以产生有趣的页面过渡和更多效果。您可以对大多数 WPF 类（甚至自定义类）进行动画处理。下图演示了一个简单的活动动画。

为了加快高质量的文本呈现，WPF 提供了以下功能：

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

WPF 本身支持使用三种类型的文档：流文档、固定文档和 XML 纸张规范 (XPS) 文档。WPF 还提供了用于创建、查看、管理、批注、打包和打印文档的服务。

XML 纸张规范 (XPS) 文档建立在 WPF 的固定文档基础上。XPS 文档使用基于 XML 的架构进行描述，该架构本质上就是电子纸的分页表示。XPS 是一个开放的、跨平台的文档格式，旨在简化分页文档的创建、共享、打印和存档。XPS 技术的重要功能包括：

打包

WPF [System.IO.Packaging](#) API 允许您的应用程序将数据、内容和资源组织成一个可移植、易于分发和访问的 ZIP 文档。可以包括数字签名以对程序包中包含的项目进行身份验证，并确定签名的项目未被篡改或修改。您还可以使用权限管理对软件包进行加密，以限制对受保护信息的访问。

打印

.NET Framework 包括一个打印子系统，WPF 通过支持更好的打印系统控制对其进行了增强。打印增强功能包括：

- 实时安装远程打印服务器和队列。
- 动态发现打印机功能。
- 动态设置打印机选项。
- 打印作业重新路由和重新排列优先级次序。

内容模型

大多数 WPF 控件的主要目的都是为了显示内容。在 WPF 中，构成控件内容的项目类型和数量被称为控件的“内容模型”。有些控件只能包含一个项目和内容类型；例如，[TextBox](#) 的内容为字符串值，该值被分配给 [Text](#) 属性。

触发器

尽管 XAML 标记的主要目的是实现应用程序的外观，但您仍然可以使用 XAML 实现应用程序行为的某些方面。一个示例就是使用触发器根据用户交互更改应用程序的外观。有关更多信息，请参见[样式设置和模板化](#)中的“触发器”。

数据模板

控件模板使您可以指定控件的外观，数据模板则允许您指定控件内容的外观。数据模板通常用于改进绑定数据的显示方式。下图演示 [ListBox](#) 的默认外观，它被绑定到一个 Task 对象集合，该集合中的每个任务都有一个名称、说明和优先级。

开发人员和设计人员使用样式可以对其产品的特定外观进行标准化。WPF 提供了一个强大的样式模型，其基础是 [Style](#) 元素。下面的示例创建一个样式，该样式将窗口中的每个 [Button](#) 的背景色设置为 [Orange](#)。

资源

一个应用程序中的各控件应共享相同的外观，包括从字体和背景色到控件模板、数据模板和样式的所有方面。您可以使用 WPF 对 用户界面 (UI) 资源的支持将这些资源封装到一个位置，以便于重复使用。

资源范围有多种，包括下面按解析顺序列出的范围：

1. 单个控件（使用继承的 [FrameworkElement...::Resources](#) 属性）。
2. [Window](#) 或 [Page](#)（也使用继承的 [FrameworkElement...::Resources](#) 属性）。
3. [Application](#)（使用 [Application...::Resources](#) 属性）。

范围的多样性使您可以灵活选择定义和共享资源的方式。

作为将资源与特定范围直接关联的一个备用方法，您可以使用单独的 [ResourceDictionary](#)（可以在应用程序的其他部分引用）打包一个或多个资源。例如，下面的示例在资源字典中定义默认背景色。

由于 WPF 的外观由模板定义，因此 WPF 为每个已知 Windows 主题包括了一个模板，

WPF 中的主题和外观都可以使用资源字典非常轻松地进行定义

自定义控件

尽管 WPF 提供了大量自定义项支持，您仍然可能会遇到现有 WPF 控件不能满足应用程序或用户需求的情况。在以下情况下可能会出现这种情形：

- 无法通过自定义现有 WPF 实现的外观来创建您需要的 UI。
- 现有 WPF 实现不支持（或很难支持）您需要的行为。
- **用户控件模型。**从 [UserControl](#) 派生的自定义控件，由其他一个或多个控件组成。
- **控制模型。**从 [Control](#) 派生的自定义控件，用于生成使用模板将其行为和外观相分离的实现，与多数 WPF 控件非常相似。从 [Control](#) 派生使您可以比用户控件更自由地创建自定义 UI，但可能需要投入更多精力。
- **框架元素模型。**从 [FrameworkElement](#) 派生的自定义控件，其外观由自定义呈现逻辑（而不是模板）定义。

WPF 是一种全面的显示技术，用于生成多种类型的具有视觉震撼力的客户端应用程序。本文介绍了 WPF 的关键功能。

下一步为生成 WPF 应用程序！

将数据连接到控件

在此步骤中，您将编写代码来检索从 [HomePage](#) 上的人员列表中选定的当前项，并在实例化过程中将对当前项的引用传递给 [ExpenseReportPage](#) 的构造函数。[ExpenseReportPage](#) 使用已传入的项设置数据上下文，这就是 [ExpenseReportPage.xaml](#) 中定义的控件要绑定的内容。

WPF 3.5 定义了一个新的 XML 命名空间 <http://schemas.microsoft.com/netfx/2007/xaml/presentation>。在使用 WPF 3.5 生成应用程序时，可以使用此命名空间或在 WPF 3.0 中定义的命名空间。

应用程序

应用程序模型已得到下列改进：

- 提供全面的外接程序支持，可以支持独立应用程序和 XAML 浏览器应用程序 (XBAP) 中的非可视化和可视化外接程序。
- XBAP 现在可在 Firefox 中运行。
- 可以在 XBAP 与同一源站点中的 Web 应用程序之间共享 Cookie。
- 为提高工作效率而改进的 XAML IntelliSense 体验。
- 更广泛的本地化支持。

WPF 中的可视化和非可视化外接程序

可扩展的应用程序可以公开它的功能，从而允许其他应用程序与该应用程序集成并扩展其功能。外接程序是应用程序公开其扩展性的一种常见方式。在 .NET Framework 中，外接程序通常是作为动态链接库 (.dll) 打包的程序集。外接程

序由宿主应用程序在运行时动态加载，以便使用和扩展由宿主公开的服务。宿主和外接程序通过已知协定进行交互，该协定通常是由宿主应用程序发布的公共接口。

对 XBAP 的 Firefox 支持

WPF 3.5 的一个插件使得 XBAP 能够从 Firefox 2.0 中运行，WPF 3.0 中没有这个功能。其中的重要功能包括：

- 如果 Firefox 2.0 是默认浏览器，XBAP 将使用这一配置。也就是说，如果 Firefox 2.0 是默认浏览器，XBAP 将不使用 Internet Explorer。
- 运行 Internet Explorer 的 XBAP 所具备的安全功能对于在 Firefox 2.0 中运行的 XBAP 同样可用，其中包括部分信任的安全沙盒。由浏览器提供的其他安全功能因浏览器而异。

Cookie

独立 WPF 应用程序和 XBAP 可以创建、获取和删除会话和持久性 Cookie。在 WPF 3.5 中，可以在 XBAP、Web 服务器和同一源站点中的 HTML 文件之间共享持久性 Cookie。

图形

现在，您可以将通过 HTTP 下载的图像缓存到本地 Microsoft Internet Explorer 临时文件缓存中，这样，对该图像的后续请求将来自本地磁盘而非 Internet。根据图像大小的不同，这一功能可以显著改善网络性能。为支持此功能，添加了下面的成员：

- [BitmapImage.....UriCachePolicy](#)
- [BitmapDecoder.....Create\(Uri, BitmapCreateOptions, BitmapCacheOption, RequestCachePolicy\)](#)
- [BitmapFrame.....Create\(Uri, RequestCachePolicy\)](#)
- [BitmapFrame.....Create\(Uri, BitmapCreateOptions, BitmapCacheOption, RequestCachePolicy\)](#)

添加了 [BitmapSource.....DecodeFailed](#) 事件，用以在图像由于文件头损坏而加载失败时向用户发出通知。

数据绑定

数据绑定已得到下列改进：

- 新的调试机制降低了调试数据绑定的难度。
- 数据模型通过提供对 [IDataErrorInfo](#) 接口的支持，可以实现对业务层的验证。另外，验证模型现在还支持使用属性语法来设置验证规则。
- 数据绑定模型现在支持 LINQ 和 XLINQ。

文档

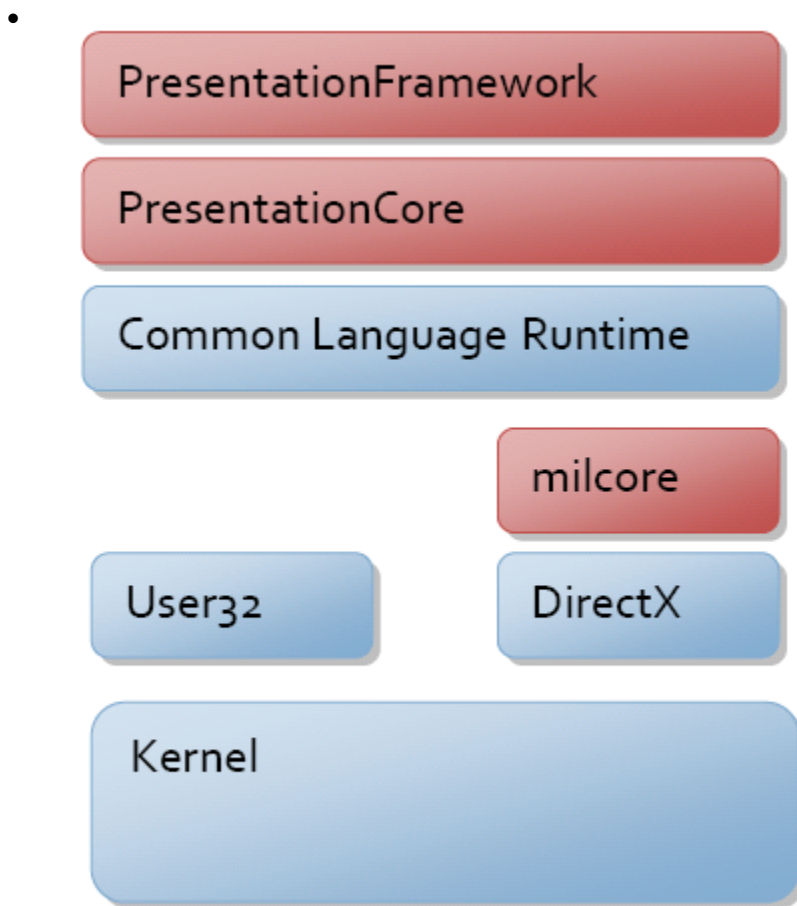
[FlowDocumentPageViewer](#)、[FlowDocumentScrollViewer](#) 和 [FlowDocumentReader](#) 各有一个名为 [Selection](#) 的新的公共属性。该属性获取表示文档中选定内容的 [TextSelection](#)。

应用程序是否具有特定于语言或非特定于语言的资源。例如，您是否为 [Application](#)、[Page](#) 和 [Resource](#) 类型指定了 [UICulture](#) 项目属性或可本地化的元数据？

本主题包括下列各节。

- [System.Object](#)
- [System.Threading.DispatcherObject](#)
- [System.Windows.DependencyObject](#)
- [System.Windows.Media.Visual](#)
- [System.Windows.UIElement](#)
- [System.Windows.FrameworkElement](#)
- [System.Windows.Controls.Control](#)
- [摘要](#)
- [相关主题](#)

- WPF 主要编程模型是通过托管代码公开的。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，可以令开发效率更高并且更加可靠（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 中的所有显示是通过 DirectX 引擎完成的，



- [System.Threading.DispatcherObject](#)

- WPF 中的大多数对象是从 [DispatcherObject](#) 派生的，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。
- [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。属性是声明性的，使您更方便地指定意图而不是操作。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您可以绑定到的更多属性，从而更好地控制应用程序的行为。

WPF 提供一个丰富的属性系统，该属性系统是从 [DependencyObject](#) 类型派生的。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，如果您具有一个会继承的属性（如 [FontSize](#)），当继承该值的元素的父级发生属性更改时，会自动更新系统。

WPF 属性系统的基础是属性表达式的概念。

属性系统还提供属性值的稀疏存储

属性系统的最后一个新功能是附家属性的概念

[Visual](#) 实际上是到 WPF 组合系统的入口点

可视对象和绘制指令的整个树都要进行缓存

System.Windows.UIElement

[UIElement](#) 定义核心子系统，包括 [Layout](#)、[Input](#) 和 [Event](#)。

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 的复杂进程路由到正确的进程和线程。与输入相对应的 User32 消息一旦路由到 WPF，它就会转换为 WPF 原始输入消息，并发送到调度程序。WPF 允许原始输入事件转换为多个实际事件，允许在保证传递到位的情况下在较低的系统级别实现类似“[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，则被称为“冒泡”，如果从根开始向下遍历到目标，它们被称为“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）事件将从目标向上冒泡到根。

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

[FrameworkElement](#) 引入的主要策略是关于应用程序布局。[FrameworkElement](#) 在 [UIElement](#) 引入的基本布局协定之上生成，并增加了布局“插槽”的概念，使布局制作者可以方便地拥有一组面向属性的一致的布局语义。[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。

[FrameworkElement](#) 引入的两个最关键的内容是数据绑定和样式。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
样式实际上是轻量级的数据绑定

System.Windows.Controls.Control

控件的最重要的功能是模板化。

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，使用户可以对控件的外观和行为进行完全自定义。最常见的控件数据模型是内容模型

您就能够创建更丰富的应用程序，这些应用程序在根本上会将数据视为应用程序的核心驱动力。

可扩展应用程序标记语言 (XAML) 语言支持，以便您能够在可扩展应用程序标记语言 (XAML) 标记中创建大部分应用程序 UI。

XAML 简化了为 .NET Framework 编程模型创建 UI 的过程。您可以在声明性 XAML 标记中创建可见的 UI 元素，然后使用代码隐藏文件（通过分部类定义与标记相连接）将 UI 定义与运行时逻辑相分离。

与其他大多数标记语言不同，XAML 直接呈现托管对象的实例化。这种常规设计原则简化了使用 XAML 创建的对象代码和调试访问。

XAML 有一组规则，这些规则将对象元素映射为类或结构，将属性 (Attribute) 映射为属性 (Property) 或事件，并将 XML 命名空间映射为 CLR 命名空间。XAML 元素映射为被引用程序集中定义的 Microsoft .NET 类型，而属性 (Attribute) 则映射为这些类型的成员。

每个实例都是通过调用基础类或结构的默认构造函数并对结果进行存储而创建的。为了可用作 XAML 中的对象元素，该类或结构必须公开一个公共的默认（无参数）构造函数。

```
<Button.Content>
    This is a button
</Button.Content>
```

XAML 的属性 (Property) 元素语法表示了与标记的基本 XML 解释之间的巨大背离。对于 XML，<类型名称,属性> 代表了另一个元素，该元素仅表示一个子元素，而与 *TypeName* 父级之间没有必然的隐含关系。在 XAML 中，<类型名称.Property> 直接表示 Property 是类型名称 的属性（由属性元素内容设置），而绝不会是一个名称相似（碰巧名称中有一个点）但却截然不同的元素。

引用值和标记扩展

标记扩展是一个 XAML 概念。在属性语法中，花括号 { 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属性值视为一个字符串或者可直接转换为文本字符串的值。

WPF 应用程序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 (Property) 不支持对直接对象实例化使用属性 (Attribute) 语法，也可以使用属性 (Attribute) 语法为属性 (Property) 提供引用值

资源只是 WPF 或 XAML 启用的一种标记扩展用法

Typeconverter 的属性值：但是很多 WPF 类型或这些类型的成员扩展了基本字符串属性处理行为，因此更复杂的对象类型的实例可通过字符串指定为属性值

该对象元素的任何 XML 子元素都被当作包含在一个表示该内容属性的隐式属性元素标记中来处理。在标记中，可以省略 XAML 内容属性的属性元素语法。在标记中指定的任何子元素都将成为 XAML 内容属性的值。

XAML 内容属性值必须连续

XAML 处理器和序列化程序将忽略或删除所有无意义的空白，并规范化任何有意义的空白。只有当您在 XAML 内容属性中指定字符串时，才会体现此行为的重要性。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。

一个 XAML 文件只能有一个根元素，这样才能成为格式正确的 XML 文件和有效的 XAML 文件。通常，应选择属于应用程序模型一部分的元素（例如，为页面选择 [Window](#) 或 [Page](#)，为外部字典选择 [ResourceDictionary](#)，或为应用程序定义根选择 [Application](#)）。下面的示例演示 WPF 页面的典型 XAML 文件的根元素，其中的根元素为 [Page](#)。

代码隐藏、事件处理程序和分部类要求

1 分部类必须派生自用作根元素的类的类型。您可以在代码隐藏的分部类定义中将派生留空，但编译的结果会假定页根作为分部类的基类，即使在没有指定的情况下也是如此（因为分部类的标记部分确实将页根指定为基）。

2 编写的事件处理程序必须是 `x:Class` 标识的命名空间中的分部类所定义的实例方法。您不能限定事件处理程序的名称来指示 XAML 处理器在其他类范围中查找该处理程序，也不能将静态方法用作事件处理程序。

3 事件处理程序必须与相应事件的委托匹配。

类要能够实例化为对象元素，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。（托管代码结构隐式支持这样的构造函数。）
- 自定义类不能是嵌套类（嵌套类和其语法中的“点”会干扰其他 WPF 功能，例如附加属性）。
- `x:Class` 可以声明为充当可扩展应用程序标记语言 (XAML) 元素树的根元素并且正在编译（可扩展应用程序标记语言 (XAML) 通过 `Page` 生成操作包括在项目中）的任何元素的属性，也可以声明为已编译应用程序的应用程序定义中的 `Application` 根的属性。在页面根元素或应用程序根元素之外的任何元素上以及在未编译的可扩展应用程序标记语言 (XAML) 文件的任何环境下声明 `x:Class` 都会导致编译时错误。
- 用作 `x:Class` 的类不能是嵌套类。
- 完全可以在没有任何代码隐藏的情况下拥有 XAML 页，从这个角度而言，`x:Class` 是可选的，但是，如果页面声明了事件处理属性值，或者实例化其定义类在代码隐藏类中的自定义元素，那么将最终需要为代码隐藏提供对适当类的 `x:Class` 引用（或 `x:Subclass`）。
- `x:Class` 属性的值必须是一个指定类的完全限定名的字符串。对于简单的应用程序，只要命名空间信息与代码隐藏的构建方式相同（定义从类级别开始），就可以省略命名空间信息。页面或应用程序定义的代码隐藏文件必须在代码文件内，而该代码文件应作为产生已编译应用程序的项目的一部分而包括在该项目中。必须遵循 CLR 类的命名规则；有关详细信息，请参见 [Type Definitions](#)（类型定义）。默认情况下，代码隐藏类必须是 `public` 的，但也可以通过使用 `x:ClassModifier` 属性定义为另一访问级别。
- 请注意，`x:Class` 属性值的此含义是 WPF XAML 实现所特有的。WPF 外部的其他 XAML 实现可能不使用托管代码，因此可能使用不同的类解析公式。

`x:Code` 是在 XAML 中定义的一种指令元素。`x:Code` 指令元素可以包含 [内联编程代码](#)。

自定义类作为 XAML 元素的要求

为了可方便地用作路由事件，CLR 事件应实现显式 `add` 和 `remove` 方法，这两种方法分别添加和移除 CLR 事件签名的处理程序，并将这些处理程序转发到 `AddHandler` 和 `RemoveHandler` 方法

XAML 处理器是指可根据其规范（通过编译或解释）将 XAML 接受为语言、并且可以生成结果基础类以供运行时对象模型使用（也是根据 XAML 规范）的任意程序

当用于提供属性 (`Attribute`) 值时，将标记扩展与 XAML 处理器区分开来的语法就是左右大括号 (`{` 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。

`StaticResource` 通过替换已定义资源的值来为 XAML 属性提供值

DynamicResource 通过将值推迟为对资源的运行时引用来为 XAML 属性提供值

Binding 按应用于元素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，

通过 **TemplateBinding**，控件模板可以使用来自要利用该模板的类的对象模型定义属性中的模板化属性的值

XAML 处理器中的属性处理使用大括号作为标记扩展的指示符。

这些声明之间的关系是：**XAML** 实际上是语言标准，而 **WPF** 是将 **XAML** 作为语言使用的一个实现。**XAML** 语言指定一些为了兼容而假定要实现的语言元素，每个元素都应当能通过针对 **XAML** 命名空间执行的 **XAML** 处理器实现进行访问。**WPF** 实现为其自己的 **API** 保留默认命名空间，为 **XAML** 中需要的标记语法使用单独的映射前缀。按照约定，该前缀是 **x:**，此 **x:** 约定后面是项目模板、示例代码和此 **SDK** 中语言功能的文档。**XAML** 命名空间定义了许多常用功能，这些功能即使对于基本的 **WPF** 应用程序也是必需的。例如，若要通过分部类将任何代码隐藏加入 **XAML** 文件，您必须将该类命名为相关 **XAML** 文件的根元素中的 **x:Class** 属性。或者，在 **XAML** 页中定义的、您希望作为键控资源访问的任何元素应当对相关元素设置了 **x:Key** 属性。

映射到自定义类和程序集：**clr-namespace:** 在包含要作为元素公开的公共类型的程序集中声明的公共语言运行库 (CLR) 命名空间。

assembly= 是指包含部分或全部引用的 CLR 命名空间的程序集。该值通常只是程序集的名称，而不是路径。该程序集的路径必须在生成编译的 **XAML** 的项目文件中以项目引用的形式建立。另外，为了合并版本管理和强名称签名，该值也可以是 [AssemblyName](#) 定义的字符串。

请注意，分隔 **clr-namespace** 标记和其值的字符是冒号 (:)，而分隔 **assembly** 标记和其值的字符是等号 (=)。这两个标记之间使用的字符是分号。例如：

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

唯一标识对象元素，以便于从代码隐藏或通用代码中访问实例化的元素。**x:Name** 一旦应用于支持编程模型，便可被视为与由构造函数返回的用于保存对象引用的变量等效。

x:Name 无法应用于某些范围。例如，[ResourceDictionary](#) 中的项不能有名称，因为它们已有作为唯一标识符的 **x:Key** 属性。

因为在 **WPF** 命名空间为几个重要基类（如 [FrameworkElement/FrameworkContentElement](#)）指定的 **Name** 依赖项属性也具有此用途。仍然有一些常见的 **XAML** 以及框架方案需要在不使用 **Name** 属性的情况下通过代码访问元素，这种情况在某些动画和演示图板支持类中最为突出。例如，您应当在时间线以及在 **XAML** 中创建的转换上指定 **x:Name**，前提是您计划在代码中引用它们。

如果 **Name** 定义为元素的一个属性，则 **Name** 和 **x:Name** 可互换使用，但如果同一元素上同时指定了两者，将会产生错误。

x:Static 标记扩展：

引用以符合公共语言规范 (CLS) 的方式定义的任何静态的按值代码实体。引用的属性在加载

XAML 页的余下部分之前计算，可用于以 **XAML** 提供

```
<object>
  <object.property>
    <x:Static Member="prefix:typeName.staticMemberName" .../>
  </object.property>
</object>
```

引用的代码实体必须是下面的某一项：


- 常量
- 静态属性
- 字段
- 枚举值
- `x:Subclass` 用法主要针对不支持分部类声明的语言。

派生类中的事件处理程序必须是 `internal override`（在 `Microsoft Visual Basic .NET` 中必须是 `Friend Overrides`），才能重写编译期间在中间类中创建的处理程序的存根。否则，派生类实现将隐藏中间类实现，并且中间类处理程序无法被调用。

`x:type` 本质上是 `C#` 中的 `typeof()` 运算符或 `Microsoft Visual Basic .NET` 中的 `GetType` 运算符的等效标记扩展。

`{}` 转义序列用来对属性语法中用于标记扩展的 `{` 和 `}` 进行转义。严格来说，转义序列本身并不是标记扩展，

WPF 命名空间 XAML 扩展

 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](#) 属性

DynamicResource 标记扩展

key	所请求的资源的键。如果资源是在标记中创建的，则这个键最初是由 <code>x:key</code> 属性分配的；如果资源是在代码中创建的，则这个键是在调用 <code>ResourceDictionary....Add</code> 时作为 <code>key</code> 参数提供的。
-----	--

`DynamicResource` 将在初始编译过程中创建一个临时表达式，因而会将资源查找延迟到实际需要所请求的资源值来构造对象时才执行。这可能是在加载 XAML 页之后。将基于键搜索在所有活动的资源字典中查找资源值（从当前页范围开始），并且资源值将取代编译期间的占位符表达式。

Windows Presentation Foundation (WPF) 中的大部分类都从四个类派生而来，这四个类在 SDK 文档中常常被称为基元素类。这些类包括 [UIElement](#)、[FrameworkElement](#)、[ContentElement](#) 和 [FrameworkContentElement](#)。[DependencyObject](#) 也是一个相关类，因为它是 [UIElement](#) 和 [ContentElement](#) 的通用基类。

[UIElement](#) 和 [ContentElement](#) 都是从 [DependencyObject](#) 派生而来，但途径略有不同。此级别上的拆分涉及到 [UIElement](#) 或 [ContentElement](#) 如何在用户界面上使用，以及它们在应用程序起到什么作用。[UIElement](#) 在其类层次结构中也有 [Visual](#)，该类为 Windows Presentation Foundation (WPF) 公开较低级别的图形支持。[Visual](#) 通过定义独立的矩形屏幕区域来提供呈现框架。实际上，[UIElement](#) 适用于支持大型数据模型的元素，这些元素用于在可以称为矩形屏幕区域的区域内进行呈现和布局，在该区域内，内容模型特意设置得更加开放，以允许不同的元素进行组合。[ContentElement](#) 不是从 [Visual](#) 派生的；它的模型由其他对象（例如，阅读器或查看器，用来解释元素并生成完整的 [Visual](#) 供 Windows Presentation Foundation (WPF) 使用）来使用 [ContentElement](#)。某些 [UIElement](#) 类可用作内容宿主：它们为一个或多个 [ContentElement](#) 类（如 [DocumentViewer](#)）提供宿主和呈现。[ContentElement](#) 用作以下元素的基类：所具有的对象模型较小，并且多用于寻址可能宿主在 [UIElement](#) 中的文本、信息或文档内容。

创建用于扩展 WPF 的自定义类的最实用方法是从某个 WPF 类中派生，这样您可以通过现有的类层次结构获得尽可能多的所需功能。本节列出了三个最重要的元素类附带的功能，以帮助您决定要从哪个类进行派生。

如果您要实现控件（这的确是从 WPF 类派生的更常见的原因之一），您可能需要从以下类中派生：实际控件、控件系列基类或至少是 [Control](#) 基类

[DependencyObject](#) 派生的类，则将继承以下功能：

- [GetValue](#) 和 [SetValue](#) 支持以及一般的属性系统支持。
- 使用依赖项属性以及作为依赖项属性实现的附加属性的能力。

从 [UIElement](#) 派生的类，则除了能够继承 [DependencyObject](#) 提供的功能外，还将继承以下功能：

对动画属性值的基本支持。有关更多信息，请参见[动画概述](#)。

对基本输入事件和命令的支持。有关更多信息，请参见[输入概述](#)和[命令概述](#)。

可以重写以便为布局系统提供信息的虚方法。

[FrameworkElement](#) 派生的类，则除了能够继承 [UIElement](#) 提供的功能外，还将继承以下功能：

- 对样式设置和演示图板的支持。有关更多信息，请参见 [Style](#) 和[演示图板概述](#)。
- 对数据绑定的支持。有关更多信息，请参见[数据绑定概述](#)。
- 对动态资源引用的支持。有关更多信息，请参见[资源概述](#)。
- 对属性值继承以及元数据中有助于向框架服务报告属性的相关情况（如数据绑定、样式或布局的框架实现）的其他标志的支持。有关更多信息，请参见[框架属性元数据](#)。
- 逻辑树的概念。有关更多信息，请参见 [WPF 中的树](#)。

对布局系统的实际 WPF 框架级实现的支持，

[ContentElement](#) 派生的类，则除了能够继承 [DependencyObject](#) 提供的功能外，还将继承以下功能：

- 对动画的支持。有关更多信息，请参见[动画概述](#)。
- 对基本输入事件和命令的支持。有关更多信息

[FrameworkContentElement](#) 派生的类，则除了能够继承 [ContentElement](#) 提供的功能外，还将获得以下功能：

- 对样式设置和演示图板的支持。有关更多信息，请参见 [Style](#) 和 [动画概述](#)。
- 对数据绑定的支持。有关更多信息，请参见 [数据绑定概述](#)。
- 对动态资源引用的支持。有关更多信息，请参见 [资源概述](#)。
- 对属性值继承以及元数据中有助于向框架服务报告属性情况（如数据绑定、样式或布局的框架实现）的其他标志的支持。有关更多信息，请参见 [框架属性元数据](#)。
- 您不会继承对布局系统修改（如 [ArrangeOverride](#)）的访问权限。布局系统实现只在 [FrameworkElement](#) 上提供。但是，您会继承 [OnPropertyChanged](#) 重写（可以检测影响布局的属性更改并将这些更改报告给任何内容宿主）。

DispatcherObject

[DispatcherObject](#) 为 WPF 线程模型提供支持，并允许为 WPF 应用程序创建的所有对象与 [Dispatcher](#) 相关联。即使您不从 [UIElement](#)、[DependencyObject](#) 或 [Visual](#) 派生，也应考虑从 [DispatcherObject](#) 派生，以获得此线程模型支持

Visual

[Visual](#) 实现二维对象在近似矩形的区域中通常需要具有可视化表示的概念。[Visual](#) 的实际呈现发生在其他类中（不是独立的），但是 [Visual](#) 类提供了一个由各种级别的呈现处理使用的已知类型。

[Freezable](#) 对象的示例包括画笔、钢笔、变换、几何图形和动画。[Freezable](#) 提供了一个 [Changed](#) 事件以将对对象所做的任何修改通知给观察程序。冻结 [Freezable](#) 可以改进其性能，因为它不再需要因更改通知而消耗资源。冻结的

[Freezable](#) 也可以在线程之间共享，而解冻的 [Freezable](#) 则不能。

并不是每个 [Freezable](#) 对象都可以冻结。若要避免引发 [InvalidOperationException](#)，请在尝试冻结 [Freezable](#) 对象之前，查看其 [CanFreeze](#) 属性的值，以确定它是否可以被冻结。

当不再需要修改某个 [Freezable](#) 时，冻结它可以改进性能。如果您在该示例中冻结画笔，则图形系统将不再需要监视它的更改情况。图形系统还可以进行其他优化，因为它知道画笔不会更改。

```
SolidColorBrush a = new SolidColorBrush(Colors.Yellow );
    if (a.CanFreeze)
    {
        a.Freeze();
    }
```

如果下列任一情况属实，则**无法**冻结 [Freezable](#)：

- 它有动画或数据绑定的属性。
- 它有由动态资源设置的属性（有关动态资源的更多信息，请参见 [资源概述](#)）。
- 它包含无法冻结的 [Freezable](#) 子对象。

为了避免引发此异常，可以使用 [IsFrozen](#) 方法来确定 [Freezable](#) 是否处于冻结状态。

```
if (myBrush.IsFrozen) // Evaluates to true.
```

在前面的代码示例中，使用 [Clone](#) 方法对一个冻结的对象创建了可修改副本。下一节将更详细地讨论克隆操作。

从标记冻结。

若要冻结在标记中声明的 [Freezable](#) 对象，请使用 `PresentationOptions:Freeze` 属性。在下面的示例中，将一个 [SolidColorBrush](#) 声明为页资源，并冻结它。随后将它用于设置按钮的背景。

```
<Page.Resources>
```

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

若要使用 `Freeze` 属性，必须映射到表示选项命名空间：

<http://schemas.microsoft.com/winfx/2006/xaml/presentation/options>。 `PresentationOptions` 是用于映射该命名空间的推荐前缀：

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

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

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

```
mc:Ignorable="PresentationOptions"
```

[Freezable](#) 一旦冻结，便不能再修改或解冻；不过，您可以使用 [Clone](#) 或 [CloneCurrentValue](#) 方法创建一个解冻的副本

从 [Freezable](#) 派生的类可以获取以下功能。

- 特殊的状态：只读（冻结）状态和可写状态。
- 线程安全：冻结的 [Freezable](#) 可以在线程之间共享。
- 详细的更改通知：与其他 [DependencyObject](#) 不同，[Freezable](#) 对象会在子属性值更改时提供更改通知。
- 轻松克隆：[Freezable](#) 类已经实现了多种生成深层复本的方法。

每个 [Freezable](#) 子类都必须重写 [CreateInstanceCore](#) 方法。如果您的类对于其所有数据都使用依赖项属性，则您的工作已完成。

[FrameworkElement](#) 类公开一些用于精确定位子元素的属性。本主题论述其中四个最重要的属性：[HorizontalAlignment](#)、[Margin](#)、[Padding](#) 和 [VerticalAlignment](#)。务必要了解这些属性的作用，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。

在一个元素上显式设置的 [Height](#) 和 [Width](#) 属性优先于 [Stretch](#) 属性值。如果尝试设置 [Height](#)、[Width](#) 以及 [Stretch](#) 的 [HorizontalAlignment](#) 值，将会导致 [Stretch](#) 请求被忽略。

[Padding](#) 在大多数方面类似于 [Margin](#)。[Padding](#) 属性只会在少数类上公开，主要是为了方便起见而公开：[Block](#)、[Border](#)、[Control](#) 和 [TextBlock](#) 是公开 [Padding](#) 属性的类的示例。[Padding](#) 属性可将子元素的有效大小增大指定的 [Thickness](#) 值。

元素树和序列化:WPF 编程元素彼此之间通常以某种形式的树关系存在。例如，在 XAML 中创建的应用程序 UI 可以被概念化为一个元素树。可以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 中的序列化涉及到保存这两个树和应用程序的状态并将状态写入文件（通常以 XAML 形式）。

WPF 中主要的树结构是元素树。如果使用 XAML 创建应用程序页，则将基于标记中元素的嵌套关系创建树结构。如果使用代码创建应用程序，则将基于为属性（实现给定元素的内容模型）指定属性值的方式创建树结构。在 Windows Presentation Foundation (WPF) 中，处理和使用概念说明元素树的方法实际上有两种：即逻辑树和可视化树。逻辑树与可视化树之间的区别并不始终很重要，但在某些 WPF 子系统中它们可能会偶尔导致问题，并影响您对标记或代码的选择。

逻辑树

在 WPF 中，可使用属性向元素中添加内容。例如，使用 `ListBox` 控件的 `Items` 属性可向该控件中添加项。通过此方式，可将项放置到 `ListBox` 控件的 `ItemCollection` 中。若要向 `DockPanel` 中添加元素，可使用其 `Children` 属性。此时，将向 `DockPanel` 的 `UIElementCollection` 中添加元素

逻辑树用途

逻辑树的存在用途是使内容模型可以容易地循环访问其可能包含的子元素，从而可以对内容模型进行扩展。此外，逻辑树还为某些通知提供了框架，例如当加载逻辑树中的所有元素时。

此外，在 `Resources` 集合的逻辑树中首先向上查找初始请求元素，然后再查找父元素，这样可以解析资源引用。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

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，保留原始值的父元素以及继承该父元素的子元素都必须是 `FrameworkElement` 或 `FrameworkContentElement`，并且它们都必须属于某个逻辑树的一部分。但是，允许父元素的逻辑树与子元素的逻辑树相互独立，这样可以通过一个不在逻辑树中的中介可视元素使属性值继承永续进行。若要使属性值继承在这样的界限中以一致的方式工作，必须将继承属性注册为附加属性。通过帮助器类实用工具方法无法完全预测属性继承确切使用的树，即使在运行时也一样。有关更多信息，请参见 [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原因想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。作为常规 WPF 应用程序编程一部分的可视化树的一个公开情况是，**路由事件的事件路由大多数情况下遍历可视化树，而不是逻辑树**。这种微妙的事件路由行为可能不会很明显，除非您是控件作者。在可视化树中路由使得在可视化级别实现组合的控件能够处理事件或创建事件 `setter`。

树、内容元素和内容宿主

内容元素（从 `ContentElement` 派生的类）不是可视化树的一部分；内容元素不从 `Visual` 继承并且没有可视化表示形式。若要完全在 UI 中显示，则必须在既是 `Visual`，也是逻辑树元素（通常是 `FrameworkElement`）的内容宿主中承载 `ContentElement`。您可以使用概念说明，内容宿主有点类似于内容的“浏览器”，它选择要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，即使承载内容不是真实可视化树的一部分时也将如此。这样做是必要的，以便 `ContentElement` 可以为路由到除其自身之外的任何元素的路由事件提供来源。

`LogicalTreeHelper` 类为逻辑树遍历提供 `GetChildren`、`GetParent` 和 `FindLogicalNode` 方法。在大多数情况下

资源和树

资源查找基本上遍历逻辑树。不在逻辑树中的对象可以引用资源，但查找将从该对象连接到逻辑树的位置开始。仅逻辑树节点可以有包含 [ResourceDictionary](#) 的 `Resources` 属性，因此这意味着，遍历可视化树来查找资源没有好处。

但是，资源查找也可以超出直接逻辑树。

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。

[Save](#) 的序列化输出是独立的：序列化的所有内容都包含在单个 XAML 页面中，该页面具有单个根元素而且没有除 `URI` 以外的外部引用。例如，如果您的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

因为序列化是独立的并局限于逻辑树，所以没有工具可用于存储事件处理程序

应注意 **Windows Presentation Foundation (WPF)** 类的对象初始化的几个方面特意不在调用类构造函数时所执行的代码的某一部分实现。这种情况对于控件类尤为突出，该控件的大部分可视化表示都不是由构造函数定义的，而是由控件的模板定义的。模板可能来自于各种源，但是最常见的情况是来自于主题样式。模板实际上是后期绑定的；只有在相应的控件已准备好应用布局时，才会将所需的模板附加到该控件上。

如果针对其设置属性的元素是 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生类，则您可以调用 [BeginInit](#) 和 [EndInit](#) 的类版本，而不是强制转换为 [ISupportInitialize](#)。

WPF 提供了一组服务，这些服务可用于扩展公共语言运行库 (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项属性。本概述介绍 WPF 属性系统以及依赖项属性的功能

依赖项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。这些其他输入可以包括系统属性（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/演示图板）、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，可以通过实现依赖项属性来提供独立验证、默认值、监视其他属性的更改的回调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项属性元数据（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。

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项属性的基类。

- **依赖项属性：**一个由 [DependencyProperty](#) 支持的属性。
- **依赖项属性标识符：**一个 [DependencyProperty](#) 实例，在注册依赖项属性时作为返回值获得，之后将存储为一个类成员。在与 WPF 属性系统交互的许多 API 中，此标识符用作一个参数。

属性以及支持它的 [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。

```
<Button.Background>
    <ImageBrush ImageSource="wavy.jpg"/>
</Button.Background>
</Button>
```

在代码中设置依赖项属性值通常只是调用由 CLR“包装”公开的 `set` 实现。获取属性值实质上也是在调用 `get`“包装”实现：

- 依赖项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。每个这样的功能通常都表示或支持整套 WPF 功能中的特定功能
[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](#)
- `<DockPanel.Resources>`
- `<SolidColorBrush x:Key="MyBrush" Color="Gold"/>`

- `</DockPanel.Resources>`

在定义了某个资源之后，可以引用该资源并使用它来提供属性值：

```
<Button Background="{DynamicResource MyBrush}" Content="I am gold" />
```

```
//StaticResource
```

```
<Style x:Key="GreenButtonStyle">
  <Setter Property="Control.Background" Value="Green"/>
</Style>
//用
<Button Style="{StaticResource GreenButtonStyle}">I am green!</Button>
```

资源被视为本地值，这意味着，如果您设置另一个本地值，该资源引用将被消除

绑定被视为本地值，这意味着，如果您设置另一个本地值，该绑定将被消除

```
<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>
```

属性值继承

元素可以从其在树中的父级继承依赖项属性的值。

说明：

属性值继承行为并未针对所有的依赖项属性在全局启用，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

作用一类 Button 的 property：

```

<StackPanel>
    <StackPanel.Resources>
        <Style 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>

```

为什么存在依赖项属性优先级？

通常，您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，通常将很难使用样式或元素）。因此，来自样式的值的操作优先级将低于本地设置的值。

附加属性是一种类型的属性，它支持 XAML 中的专用语法。附加属性通常与公共语言运行库 (CLR) 属性不具有 1:1 对应关系，而且不一定是依赖项属性。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。附加属性旨在用作可在任何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”的一种特殊形式的依赖项属性。

。附加属性是一个 XAML 概念，而依赖项属性则是一个 WPF 概念。

附加属性的一个用途是允许不同的子元素为实际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将如何在用户界面 (UI) 中呈现。一个示例是 `DockPanel.Dock` 属性。`DockPanel.Dock` 属性创建为附加属性，因为它将在 `DockPanel` 中包含的元素上设置，而不是在 `DockPanel` 本身设置。`DockPanel` 类定义名为 `DockProperty` 的静态 `DependencyProperty` 字段，然后将 `GetDock` 和 `SetDock` 方法作为该附加属性的公共访问器提供。

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。之后，该类型将在内部逻辑中循环访问其子元素，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个服务。其他类型为该附加属性设置值。之后，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。

如果您希望对属性启用属性值继承，则应使用附加属性，而不是非附加的依赖项属性。有关详细信息

```

DockPanel myDockPanel = new DockPanel();
CheckBox myCheckBox = new CheckBox();
myCheckBox.Content = "Hello";
myDockPanel.Children.Add(myCheckBox);
DockPanel.SetDock(myCheckBox, Dock.Top);

```

何时创建附加属性

当确实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，您可能会创建附加属性。这种情况的最常见方案是布局。现有布局属性的示例有 `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。

前面已提到，如果您希望使用属性值继承，应注册为一个附加属性。

如何创建附加属性

如果您的类将附加属性严格定义为用于其他类型，那么该类不必从 `DependencyObject` 派生。但是，如果您遵循使附加属性同时也是依赖项属性的整体 WPF 模型，则需要从 `DependencyObject` 派生。

```

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);
}

```

[DependencyProperty](#) 和 [DependencyObject](#) 配合，提供了 WPF 中基本的数据存储、访问和通知的机制。也正是因为这两个东西的存在，使得 XAML，Binding，Animation 都成为可能。

```

public static readonly DependencyProperty CounterProperty =
    DependencyProperty.Register(
        "PropertyName", // 属性名
        typeof(int),     // 属性的类型
        typeof(MyButtonSimple), // 属性所在的类型
        new PropertyMetadata(0) // 属性的默认值
    )

```

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。

什么是依赖项属性？

您可以启用本应为公共语言运行库 (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，

依赖项属性只能由 [DependencyObject](#) 类型使用，WPF 中的所有依赖项属性（大多数附加属性除外）也是 CLR 属性

在类体中定义依赖项属性是典型的实现，但是也可以在类静态构造函数中定义依赖项属性。如果您需要多行代码来初始化依赖项属性，则此方法可能会很有意义。

如果要创建在 [FrameworkElement](#) 的派生类上存在的依赖项属性，则可以使用更专用的元数据类型 [FrameworkPropertyMetadata](#)，而不是 [PropertyMetadata](#) 基类。

WPF 属性系统提供了一个强大的方法，使得依赖项属性的值由多种因素决定，从而实现了诸如实时属性验证、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

本地属性集在设置时具有最高优先级，动画值和强制转换除外。如果您在本地设置某个值，一定会希望该值能优先得到应用，甚至希望其优先级高于任何样式或控件模板

```

<Button >
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Setter Property="Background" Value="Green"/>
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Background" Value="Blue" />

```

```
        </Trigger>
    </Style.Triggers>
</Style>
</Button.Style>
Click
</Button>
```

依赖项属性设置优先级列表

1. **属性系统强制转换。** 有关强制转换的详细信息，
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情况下也将如此。有关详细信息，
3. **本地值。** 本地值可以通过“包装”属性 (Property) 的便利性进行设置，这也相当于在 XAML 中设置属性 (Attribute) 或属性 (Property) 元素，或者使用特定实例的属性调用 [SetValue](#) API。如果您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，请参见本主题后面的 [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 (Property) 集。（通常通过 XAML 属性 (Attribute) 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上的样式中的触发器。（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 Setter。** 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，
 - a. 主题样式中的活动触发器。
 - b. 主题样式中的 [Setter](#)。
10. **继承。** 有几个依赖项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。
11. **来自依赖项属性元数据的默认值。** 任何给定的依赖项属性都具有一个默认值，它由该特定属性的属性系统注册来确定。而且，继承依赖项属性的派生类具有按照类型重写该元数据（包括默认值）的选项。有关更多信息，。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。

WPF 附带的每个控件都有一个默认样式，在默认样式中，对于控件最重要的信息就是其控件模板，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。

[ClearValue](#) 方法为从在元素上设置的依赖项属性中清除任何本地应用的值提供了一个有利的途径

属性值继承

属性值继承是包容继承

父元素还可以通过属性值继承来获得其值，因此系统有可能一直递归到页面根元素。属性值继承不是属性系统的默认行为；属性必须用特定的元数据设置来建立，以便使该属性能够对子元素启动属性值继承。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。通过更改自定义属性的元数据，还可以使您自己的自定义属性可继承。

附加属性的典型方案是针对子元素设置属性值，如果所讨论的属性是附加属性

属性继承通过遍历元素树来工作。此树通常与逻辑树平行。跨树边界继承属性值(属性值继承就可以弥合逻辑树中的这种间隙，而且仍可以传递所继承的值)

为了纠正此问题，在类构造函数调用中，必须将集合依赖项属性值重设为唯一的实例

当您定义自己的属性并需要它们支持 **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，应将其实现为依赖项属性。

设计依赖项属性：

```
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));
}
```

设计附加属性：

```
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);
}
```

可以从功能或实现的角度来考虑路由事件。此处对这两种定义均进行了说明，因为用户当中有的认为前者更有用，而有的则认为后者更有用。

路由事件：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，可以由 **RoutedEvent** 类的实例提供支持并由 **Windows Presentation Foundation (WPF)** 事件系统来处理。

路由事件的顶级方案

下面简要概述了需运用路由事件的方案，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对图像内容的 [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，才能处理可能是从多个元素引发的事件。路由事件使您可以只附加该处理程序一次（像上例中那样），并在必要时使用处理程序逻辑来确定该事件源自何处。例如，这可以是前面显示的 XAML 的处理程序：

类处理：路由事件允许使用由类定义的静态处理程序。这个类处理程序能够抢在任何附加的实例处理程序之前来处理事件。

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，以此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。

路由事件使用以下三个路由策略之一：

- **冒泡：**针对事件源调用事件处理程序。路由事件随后会路由到后续的父元素，直到到达元素树的根。大多数路由事件都使用冒泡路由策略。冒泡路由事件通常用来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（类处理将在下一节中介绍）而且可以由 [EventSetter](#) 和 [EventTrigger](#) 使用。
- **隧道：**最初将在元素树的根处调用事件处理程序。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。在合成控件的过程中通常会使用或处理隧道路由事件，这样，就可以有意地禁止显示复合部件中的事件，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/冒泡对实现的。隧道事件有时又称作 **Preview** 事件，这是由隧道/冒泡对所使用的命名约定决定的。

如果您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您自己的自定义控件类。路由事件还可以用来通过元素树进行通信，因为事件的事件数据会永存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项内容，该更改将对于路由中的下一个元素可用。

- 某些 WPF 样式和模板功能（如 [EventSetter](#) 和 [EventTrigger](#)）要求所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，类可以凭借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，因为您的类可以强制执行事件驱动的行为，以防它们在处理实例上的事件时被意外禁止。
- `<Button Click="b1SetColor">button</Button>`
- **b1SetColor** 是所实现的处理程序的名称，该处理程序中包含用来处理 [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。

- **entHandler** 是基本的路由事件处理程序委托

-
- ```
void MakeButton()
```
- ```
{
```
- ```
 Button b2 = new Button();
```
- ```
    b2.AddHandler(Button.ClickEvent, new RoutedEventHandler(Onb2Click));
```
- ```
}
```

```

void MakeButton2()
{
 Button b2 = new Button();
 b2.Click += new RoutedEventHandler(Onb2Click2);
}

```

## “已处理”概念

所有的路由事件都共享一个公用的事件数据基类 [RoutedEventArgs](#)。[RoutedEventArgs](#) 定义了一个采用布尔值的 [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”。处理程序在路由路径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路径上的每个侦听器。

[Handled](#) 的值影响路由事件在沿路由线路向远处传播时的报告或处理方式。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负责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；由语言特定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，如果将 [Handled](#) 设置为 [true](#)，以此将事件标记为“已处理”，则将“停止”隧道路由或冒泡路由，同时，类处理程序在某个路由点处处理的所有事件的路由也将“停止”。

但是，侦听器仍可以凭借“[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。您只能在代码或 [EventSetter](#) 中使用 [handledEventsToo](#) 机制：

在代码中，不使用适用于一般 CLR 事件的特定于语言的事件语法，而是通过调用 WPF 方法 [AddHandler\(RoutedEvent, Delegate, Boolean\)](#) 来添加处理程序。使用此方法时，请将 [handledEventsToo](#) 的值指定为 [true](#)。

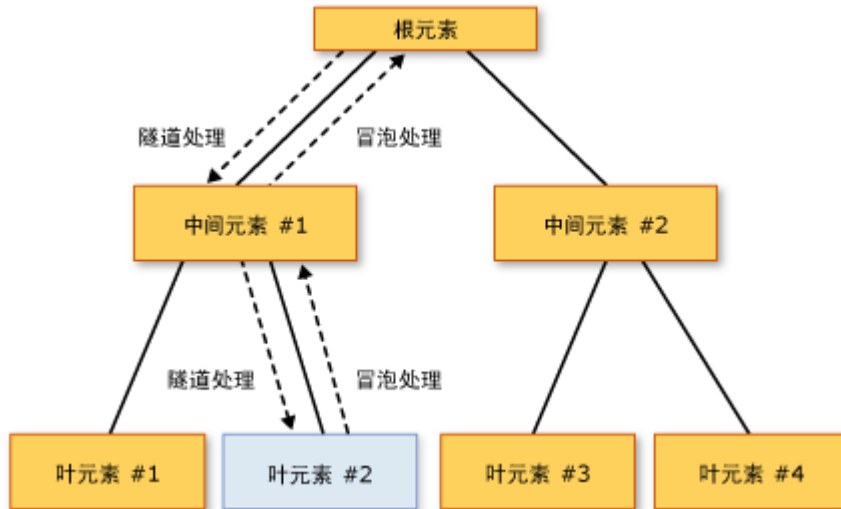
在 [EventSetter](#) 中，请将 [HandledEventsToo](#) 属性设置为 [true](#)。

## 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。正在处理该事件的元素不必定义或继承附加事件，可能引发这个特定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 输入系统广泛地使用附加事件。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等效的、作为基本元素类成员的非附加路由事件。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](#)。

这两个事件会共享同一个事件数据实例，因为用来引发冒泡事件的实现类中的 [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（先是类处理程序，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”，则会针对冒泡事件发送已经处理的事件数据，而且将不调用为等效的冒泡输入事件附加的典型处理程序。已处理的冒泡事件看起来好像尚未引发过。此处理行为对于控件合成非常有用，因为此时您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链中的根，因此将有机会首先对隧道事件进行类处理，或许还有机会将该路由事件“替换”为更特定于控件的事件。



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（冒泡）。
5. 针对中间元素 1 处理 MouseDown（冒泡）。
6. 针对根元素处理 MouseDown（冒泡）。

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是由事件数据中的 *Source* 属性报告的。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 就是这样的情况）。

通常，隧道事件和冒泡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冒泡事件等概念并非对于所有的路由事件都适用。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。实现自己的输入事件是一个高级方案，但是您也可以选择针对自己的输入事件遵循该模型。

```
<StackPanel >
 <StackPanel.Resources >
 <Style TargetType="{x:Type Button}" >
 <EventSetter Event="Click" Handler="blSetColor" />
 </Style>
 </StackPanel.Resources>
 <Button >click me</Button>
 <Button Click="HandleThis" >buton make</Button>
</StackPanel>
```

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会导致它启动现有演示图板相比，*EventTrigger* 的好处在于，*EventTrigger* 对演示图板及其运行时行为提供更好的控制

## 附加事件概述

可扩展应用程序标记语言 (XAML) 定义了一个语言组件和称为“附加事件”的事件类型。附加事件的概念允许您针对特定事件为任意元素（而不是为实际定义或继承该事件的元素）添加处理程序。在这种情况下，对象既不会引发该事件，目标处理实例也不会定义或“拥有”该事件。

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支持附加事件的使用，在 WPF 中，附加事件由 [RoutedEvent](#) 字段来支持，并在引发后通过元素树进行路由。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。

Microsoft .NET Framework 托管代码中的所有对象都要经历类似的一系列的生命阶段，即创造、使用和析构，在 WPF 中有四种与生存期事件有关的主要类型的对象：即常规元素、窗口元素、导航宿主和应用程序对象。任何 WPF 框架级元素（从 [FrameworkElement](#) 或 [FrameworkContentElement](#) 派生的那些对象）都有三种通用的生存期事件：[Initialized](#)、[Loaded](#) 和 [Unloaded](#)。

引发 [Loaded](#) 事件的机制不同于 [Initialized](#)。将逐个元素引发 [Initialized](#) 事件，而无需通过整个元素树直接协调。相反，引发 [Loaded](#) 事件是在整个元素树内协调的结果（特别是逻辑树）。当树中所有元素都处于被视为已加载状态中时，将首先在根元素上引发 [Loaded](#) 事件。然后在每个子级元素上连续引发 [Loaded](#) 事件。

构建于元素的通用生存期事件上的为以下应用程序模型元素：[Application](#)、[Window](#)、[Page](#)、[NavigationWindow](#) 和 [Frame](#)。这些元素使用与其特定用途关联的附加事件扩展了通用生存期事件

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，该处理程序在类的任何实例上的任何实例处理程序之前调用

另一个要考虑“已处理”问题的情形是，如果您的代码以重要且相对完整的方式响应路由事件，则您通常应将路由事件标记为已处理

[AddHandler\(RoutedEvent, Delegate, Boolean\)](#)，在某些情况下，控件本身会将某些路由事件标记为已处理。已处理的路由事件代表 WPF 控件作者这样的决定：即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，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虚拟类处理程序之一，可以完成此操作

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

隧道路由事件与冒泡路由事件之间的连接是由给定的任意 WPF 类引发自己的已声明路由事件的方式的内部实现来完成的，对于成对的输入路由事件也是如此。但是除非这一类级实现存在，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。

路由事件的类处理主要是用于输入事件和复合控件

当您通常处理预览事件时，应谨慎地在事件数据中将事件标记为已处理。在引发预览事件的元素（在事件数据中报告为源的元素）之外的任何元素上处理该事件都有这样的后果：使得元素没有机会处理源自于它的事件。有时这是希望的结果，尤其当该元素存在于控件的复合关系内时。

特别是对于输入事件而言，预览事件还与对等的冒泡事件共享事件数据实例。如果您使用预览事件类处理程序将输入事件标记为已处理，将不会调用冒泡输入事件类处理程序。或者，如果您使用预览事件实例处理程序将事件标记为已处理，则通常不会调用冒泡事件的类处理程序。

通常使用预览事件的一种情形是复合控件的输入事件处理。

## 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。该事件数据类型是 [RoutedPropertyChangedEventArgs<Of <\(T\)>>](#)，委托是 [RoutedPropertyChangedEventHandler<Of <\(T\)>>](#)。事件数据和委托都具有用于在您定义处理程序时指

定更改属性的实际类型的泛型参数。事件数据包含两个属性，即 `OldValue` 和 `NewValue`，之后它们均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一个路由事件。路由属性更改事件的好处是，如果子元素（控件的组成部分）的属性值更改，控件的顶级也可以接收到属性更改事件。例如，您可能创建一个合并 `RangeBase` 控件（如 `Slider`）的控件。如果滑块部分的 `Value` 属性值更改，则您可能需要在父控件（而不是在该部分）处理此更改。

### 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。该事件数据类型是 `RoutedPropertyChangedEventArgs<Of <(T>)>>`，委托是 `RoutedPropertyChangedEventHandler<Of <(T>)>>`。事件数据和委托都具有用于在您定义处理程序时指定更改属性的实际类型的泛型参数。事件数据包含两个属性，即 `OldValue` 和 `NewValue`，之后它们均作为事件数据中的类型参数传递。

### DependencyPropertyChanged 事件

属于属性更改事件方案一部分的另一对类型是 `DependencyPropertyChangedEventArgs` 和 `DependencyPropertyChangedEventHandler`。这些属性更改的事件不会路由；它们是标准的 CLR 事件。`DependencyPropertyChangedEventArgs` 不是普通的事件数据报告类型，因为它不是派生自 `EventArgs`；`DependencyPropertyChangedEventArgs` 是一个结构，而不是一个类。

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部创建的，使用它，您可以创建基于分配有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一个依赖项属性。属性的主要方案是报告控件状态，可能与实时 UI 具有因果关系，并因此是一个属性触发器候选项。

其中的一些属性还具有专用属性更改事件。例如，属性 `IsMouseCaptured` 具有一个属性更改事件 `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`。

为了补偿具有各种可能值的属性触发器的“if”条件，通常最好使用 `Setter` 将该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` 将具有优先权，而只要触发器条件为 `false`，则不在 `Trigger` 内部的 `Setter` 就具有优先权。

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。

如何实现 WeakEvent 模式？

实现 WeakEvent 模式由三个方面组成：

- 从 `WeakEventManager` 类派生一个管理器。
- 在任何想要注册弱事件的侦听器的类上实现 `IWeakEventListener` 接口，而不生成源的强引用。
- 注册侦听器时，对于想要侦听器使用该模式的事件，不要使用该事件的常规的 `add` 和 `remove` 访问器，请在该事件的专用 WPF) 中的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，这是由于存在事件路由。用 `WeakEventManager` 中改用“`AddListener`”和“`RemoveListener`”实现。

下面的示例使用 XAML 属性语法向公用的父元素（在本示例中为 `StackPanel`）附加事件处理程序。本示例使用属性语法向 `StackPanel` 父元素附加事件处理程序，而不是为每个 `Button` 子元素都附加一个事件处理程序。这个事件处理模式演示了如何使用事件路由技术来减少需要附加处理程序的元素数量。每个 `Button` 的所有冒泡事件都通过父元素进行路由。

若要使您的自定义事件支持事件路由，需要使用 `RegisterRoutedEvent` 方法注册 `RoutedEvent`。本示例演示创建自定义路由事件的基本原理。

自定义路由事件：

```
public partial class mySimpleButton : Button
{
 //public static readonly DependencyProperty IsSpinningProperty = DependencyProperty.Register();
```

```

 public static readonly RoutedEvent TapEvent =EventManager.RegisterRoutedEvent("Tap",
RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(mySimpleButton));
 public event RoutedEventHandler Tap
 {
 add { AddHandler(TapEvent, value); }
 remove { RemoveHandler(TapEvent, value); }
 }
 public void RaiseTapEvent()
 {
 RoutedEventArgs newEventArgs = new RoutedEventArgs(mySimpleButton.TapEvent);
 RaiseEvent(newEventArgs);
 }
 protected override void OnClick()
 {
 RaiseTapEvent();
 }
 }
}

```

**WPF** 子系统为输入提供了一个全新的功能强大的 **API**。命令是 (**WPF**) 中的输入机制，它提供的输入处理比设备输入具有更高的语义级别，(**WPF**) 子系统提供了一个功能强大的 **API**，用于获取来自各种设备（包括鼠标、键盘和手写笔）的输入。

。许多输入事件都有一对与其关联的事件。例如，键按下事件与 **KeyDown** 和 **PreviewKeyDown** 事件关联。这些事件之间的差别是它们路由到目标元素的方式。预览事件在元素树中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。

下面的示例使用 **GetKeyStates** 方法确定 **Key** 是否处于按下状态。

```

if ((Keyboard.GetKeyStates(Key.Return) & KeyStates.Down) > 0)
{
 btnNone.Background = Brushes.Red;
}

```

下面的示例确定鼠标上的 **LeftButton** 是否处于 **Pressed** 状态。

```

if (Mouse.LeftButton == MouseButtonState.Pressed)
{
 UpdateSampleResults("Left Button Pressed");
}

```

路由事件使用三种路由机制之一：直接、冒泡和隧道。在直接路由中，源元素是唯一得到通知的元素，该事件不会路由到任何其他元素。但是，相对于标准 **CLR** 事件，直接路由事件仍提供了一些其他仅对于路由事件才存在的功能。冒泡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。

由于输入事件在事件路由中向上冒泡，因此不管哪个元素具有键盘焦点，**StackPanel** 都将接收输入。**TextBox** 控件首先得到通知，而只有在 **TextBox** 未处理输入时才会调用 **OnTextInputKeyDown** 处理程序。如果使用 **PreviewKeyDown** 事件而不是 **KeyDown** 事件，则将首先调用 **OnTextInputKeyDown** 处理程序。

在此示例中，处理逻辑写入了两次，一次针对 **Ctrl+O**，另一次针对按钮的单击事件。使用命令，而不是直接处理输入事件，可简化此过程。

为了使元素能够获取键盘焦点，**Focusable** 属性和 **IsVisible** 属性必须设置为 **true**。某些类（如 **Panel**）默认情况下将 **Focusable** 设置为 **false**；因此，如果您希望该元素能够获取焦点，则必须将此属性设置为 **true**。



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。WPF 资源，可以通过一种简单的方法来重用通常定义的对象和值。

确保在请求资源之前已在资源集中对该资源进行了定义。如有必要，您可以使用 [DynamicResource](#) 标记扩展在运行时引用资源，这样可以绕过严格的资源引用创建顺序，但应注意这种 [DynamicResource](#) 技术会对性能产生一定的负面影响

```
<SolidColorBrush x:Key="MyBrush" Color="Gold"/>
<Style TargetType="Border" x:Key="PageBackground">
 <Setter Property="Background" Value="Blue"/>
</Style>
```

引用：<Style TargetType="TextBlock" x:Key="Label">  
    <Setter Property="DockPanel.Dock" Value="Right"/>  
    <Setter Property="FontSize" Value="8"/>  
    <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>  
</Style>

引用：

```
<Style TargetType="Button" x:Key="GelButton" >
 <Setter Property="Margin" Value="1, 2, 1, 2"/>
 <Setter Property="HorizontalAlignment" Value="Left"/>
 <Setter Property="Template">
 <Setter.Value>

 </Setter.Value>
 </Setter>
</Style>
```

SystemFonts:

```
<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>
```

系统资源将许多系统度量作为资源公开，以帮助开发人员创建与系统设置一致的可视元素。[SystemFonts](#) 是一个类，它包含系统字体值以及绑定到这些值的系统字体资源。例如，[CaptionFontFamily](#) 和 [CaptionFontFamilyKey](#)。

系统字体规格可以用作静态或动态资源。如果您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。

引用 DynamicResource:

```
<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>
```

系统资源会将多个基于系统的设置作为资源进行显示，以帮助您创建与系统设置协调一致的视觉效果。[SystemParameters](#) 是一个类，其中既包含系统参数值属性，又包含绑定到这些值的资源键。例如，[FullPrimaryScreenHeight](#) 是 [SystemParameters](#) 属性值，[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用法或动态资源引用（静态属性值为资源键）。如果您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。资源键的属性名称后面附有 **Key** 后缀。

```
<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>
```

查找 template 中资源：

```
<Style TargetType="{x:Type Button}">
 <Setter Property="Template">
 <Setter.Value>
 <ControlTemplate TargetType="{x:Type Button}">
 <Grid Margin="5" Name="grid">
 <Ellipse Stroke="DarkBlue" StrokeThickness="2">
 <Ellipse.Fill>
 <RadialGradientBrush Center="0.3,0.2" RadiusX="0.5" RadiusY="0.5">
 <GradientStop Color="Azure" Offset="0.1" />
 <GradientStop Color="CornflowerBlue" Offset="1.1" />
 </RadialGradientBrush>
 </Ellipse.Fill>
 </Ellipse>
 <ContentPresenter Name="content" Margin="10"
 HorizontalAlignment="Center" VerticalAlignment="Center"/>
 </Grid>
 </ControlTemplate>
 </Setter.Value>
 </Setter>
</Style>
```

Code;

```
Grid myGrid = (Grid)myButton1.Template.FindName("grid",myButton1);
MessageBox.Show("the width is "+myGrid.GetValue (Grid.ActualHeightProperty));
```

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。布局系统为 [Children](#) 集合的每个成员完成两个处理过程：[测量处理过程](#)和[排列处理过程](#)，每个子 [Panel](#) 均提供自己的 [MeasureOverride](#) 和 [ArrangeOverride](#) 方法，以实现自己特定的布局行为。不论何时调用布局系统，都会发生以下系列事件。

1. 子 [UIElement](#) 通过首先测量它的核心属性来开始布局过程。
2. 计算在 [FrameworkElement](#) 上定义的大小调整属性，例如 [Width](#)、[Height](#) 和 [Margin](#)。
3. 应用 [Panel](#) 特定逻辑，例如 [Dock](#) 方向或堆栈 [Orientation](#)。
4. 测量所有子级后排列内容。
5. [Children](#) 集合绘制到屏幕。
6. 如果其他 [Children](#) 添加到集合、应用 [LayoutTransform](#) 或调用 [UpdateLayout](#) 方法，会再次调用此过程。

当呈现 [Window](#) 对象的内容时，会自动调用布局系统。为了显示内容，窗口的 [Content](#) 必须定义根 [Panel](#)

首先，将计算 `UIElement` 的本地大小属性，如 `Clip` 和 `Visibility`。这将生成一个名为 `constraintSize` 的传递给 `MeasureCore` 的值。

其次，会处理在 `FrameworkElement` 上定义的框架属性，这将影响 `constraintSize` 的值。这些属性旨在描述基础 `UIElement` 的大小调整特性，例如其 `Height`、`Width`、`Margin` 和 `Style`。上述每个属性均可能改变显示元素所必需的空间。然后，将用 `constraintSize` 作为一个参数调用 `MeasureOverride`。

此排列过程将以调用 `Arrange` 方法开始。在排列处理过程期间，父 `Panel` 元素生成一个代表子级边界的矩形。该值会传递给 `ArrangeCore` 方法以便进行处理。

`ArrangeCore` 方法计算子级的 `DesiredSize`，计算可能影响该元素呈现大小的任何其他边距，并生成 `arrangeSize`（作为参数传递给 `Panel` 的 `ArrangeOverride`）。`ArrangeOverride` 生成子级的 `finalSize`，最后，`ArrangeCore` 方法执行偏移属性（例如边距和对齐方式）的最终计算，并将子级放在其布局槽内。子级无需（且通常不会）填充整个分配空间。然后，控件返回到父 `Panel`，至此布局过程完成。

`LayoutTransform` 可能是影响用户界面 (UI) 内容的非常有用的方式。不过，如果转换的效果无需对其他元素的位置施加影响，则最好改为使用 `RenderTransform`，因为 `RenderTransform` 不会调用布局系统。`LayoutTransform` 会应用其转换，并强制对帐户执行递归布局更新，以获取受影响元素的新位置。

WPF) 提供了丰富的控件库，这些控件支持 用户界面 (UI) 开发、文档查看和序列化数字墨迹。

类不必从 `Control` 类继承，即可具有可见外观。从 `Control` 类继承的类包含一个 `ControlTemplate`，允许控件的使用方在无需创建新子类的情况下根本改变控件的外观。

渐变色：

```
<Button.Background>
 <LinearGradientBrush StartPoint="0, 0.5"
 EndPoint="1, 0.5">
 <GradientStop Color="Green" Offset="0.0" />
 <GradientStop Color="White" Offset="0.9" />
 </LinearGradientBrush>
</Button.Background>
```

Coding:

```
LinearGradientBrush buttonBrush = new LinearGradientBrush();
buttonBrush.StartPoint = new Point(0, 0.5);
buttonBrush.EndPoint = new Point(1, 0.5);
buttonBrush.GradientStops.Add(new GradientStop(Colors.Green, 0));
buttonBrush.GradientStops.Add(new GradientStop(Colors.White, 0.9));

submit.Background = buttonBrush;
submit.FontSize = 14;
submit.FontWeight = FontWeights.Bold;
```

从 `Control` 类继承的类具有 `ControlTemplate`，它用于定义 `Control` 的结构和外观。`Control` 的 `Template` 属性是公共的，因此您可以为 `Control` 指定非默认 `ControlTemplate`。通常，您可以为 `Control` 指定新的 `ControlTemplate`（而不是从控件继承）以自定义 `Control` 的外观。

ControlTemplate:

```
<Style TargetType="Button">
 <Setter Property="Template">
 <Setter.Value>
 <ControlTemplate TargetType="Button">
 <Border x:Name="Border">
```

```

CornerRadius="20"
BorderThickness="1"
BorderBrush="Black">
 <Border.Background>
 <LinearGradientBrush StartPoint="0,0.5"
 EndPoint="1,0.5">
 <GradientStop Color="{Binding Background.Color,
RelativeSource={RelativeSource TemplatedParent}}"
 Offset="0.0" />
 <GradientStop Color="White" Offset="0.9" />
 </LinearGradientBrush>
 </Border.Background>
</ContentPresenter

Margin="2"
HorizontalAlignment="Center"
VerticalAlignment="Center"
RecognizesAccessKey="True"/>
</Border>
<ControlTemplate.Triggers>
 <!--Change the appearance of
the button when the user clicks it.-->
 <Trigger Property="IsPressed" Value="true">
 <Setter TargetName="Border" Property="Background">
 <Setter.Value>
 <LinearGradientBrush StartPoint="0,0.5"
 EndPoint="1,0.5">
 <GradientStop Color="{Binding Background.Color,
RelativeSource={RelativeSource TemplatedParent}}"
 Offset="0.0" />
 <GradientStop Color="DarkSlateGray" Offset="0.9" />
 </LinearGradientBrush>
 </Setter.Value>
 </Setter>
 </Trigger>

</ControlTemplate.Triggers>
</ControlTemplate>
</Setter.Value>
</Setter>
</Style>

```

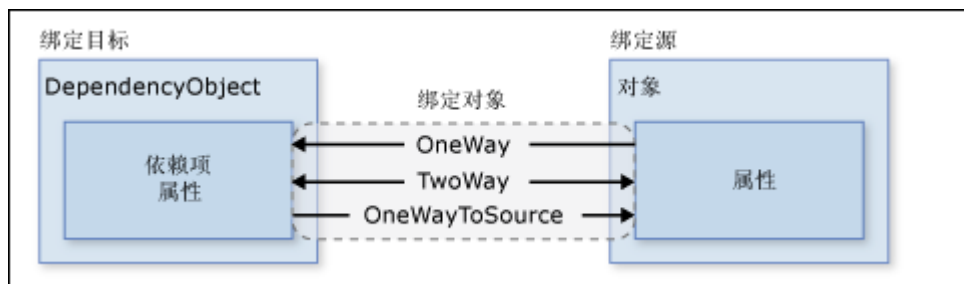
Control:

- **ContentControl** - 从此类继承的类的部分示例包括有 **Label**、**Button** 和 **ToolTip**。
- **ItemsControl** - 从此类继承的类的部分示例包括有 **ListBox**、**Menu** 和 **StatusBar**。
- **HeaderedContentControl** - 从此类继承的类的部分示例包括有 **TabItem**、**GroupBox** 和 **Expander**。
- **HeaderedItemsControl** - 从此类继承的类的部分示例包括有 **MenuItem**、**TreeViewItem** 和 **ToolBar**。

WPF) 数据绑定为应用程序提供了一种表示数据和与数据交互的简单而又一致的方法

Databinding:

数据绑定是在应用程序 UI 与业务逻辑之间建立连接的过程。如果绑定具有正确设置并且数据提供正确通知，则当数据更改其值时，绑定到数据的元素会自动反映更改。数据绑定可能还意味着如果元素中数据的外部表现形式发生更改，则基础数据可以自动更新以反映更改。例如，如果用户编辑 `TextBox` 元素中的值，则基础数据值会自动更新以反映该更改。



数据绑定实质上是绑定目标与绑定源之间的桥梁

目标属性必须为依赖项属性。大多数 `UIElement` 属性都是依赖项属性，而大多数依赖项属性（除了只读属性）默认情况下都支持数据绑定。（只有 `DependencyObject` 类型可以定义依赖项属性，所有 `UIElement` 都派生自 `DependencyObject`。）

请务必记住一点：当您建立绑定时，您是将绑定目标绑定到 绑定源。例如，如果您要使用数据绑定在一个 `ListBox` 中显示一些基础 XML 数据，就是将 `ListBox` 绑定到 XML 数据。

- **OneWay** 绑定导致对源属性的更改会自动更新目标属性，但是对目标属性的更改不会传播回源属性。此绑定类型适用于绑定的控件为隐式只读控件的情况。例如，您可能绑定到如股票行情自动收录器这样的源，或许目标属性没有用于进行更改的控件接口（如表的数据绑定背景色）。如果无需监视目标属性的更改，则使用 **OneWay** 绑定模式可避免 **TwoWay** 绑定模式的系统开销。
- **TwoWay** 绑定导致对源属性的更改会自动更新目标属性，而对目标属性的更改也会自动更新源属性。此绑定类型适用于可编辑窗体或其他完全交互式 UI 方案。大多数属性都默认为 **OneWay** 绑定，但是一些依赖项属性（通常为用户可编辑的控件的属性，如 `TextBox` 的 `Text` 属性和 `CheckBox` 的 `IsChecked` 属性）默认为 **TwoWay** 绑定。确定依赖项属性在默认情况下是单向绑定还是双向绑定的一种编程方式是使用 `GetMetadata` 获取属性的属性元数据，然后检查 `BindsTwoWayByDefault` 属性的布尔值。
- **OneWayToSource** 与 **OneWay** 绑定相反：它在目标属性更改时更新源属性。一个示例方案是您只需要从 UI 重新计算源值的情况。

请注意，若要检测源更改（适用于 **OneWay** 和 **TwoWay** 绑定），则源必须实现一种合适的属性更改通知机制（如 `INotifyPropertyChanged`）。

但是，源值是在您编辑文本的同时进行更新，还是在您结束编辑文本并将鼠标指针从文本框移走后才进行更新呢？绑定的 `UpdateSourceTrigger` 属性确定触发源更新的原因。下图中右箭头的点演示 `UpdateSourceTrigger` 属性的角色：

如果 `UpdateSourceTrigger` 值为 `PropertyChanged`，则 **TwoWay** 或 **OneWayToSource** 绑定的右箭头指向的值会在目标属性更改时立刻进行更新。但是，如果 `UpdateSourceTrigger` 值为 `LostFocus`，则仅当目标属性失去焦点时，该值才会使用新值进行更新。

UpdateSourceTrigger 值	源值何时进行更新	文本框的示例方案
LostFocus ( <code>TextBox.Text</code> 的默认值 )	当 <code>TextBox</code> 控件失去焦点时	一个与验证逻辑 ( 请参见 “验证逻辑” 一节 ) 关联的 <code>TextBox</code>
PropertyChanged	当键入到 <code>TextBox</code> 中时	聊天室窗口中的 <code>TextBox</code> 控件
Explicit	当应用程序调用 <code>UpdateSource</code> 时	可编辑窗体中的 <code>TextBox</code> 控件 ( 仅当用户单击提交按钮时才更新源值 )

```
<DockPanel xmlns:c="clr-namespace:SDKSample">
 <DockPanel.Resources>
 <c:MyData x:Key="myDataSource"/>
 </DockPanel.Resources>
 <DockPanel.DataContext>
 <Binding Source="{StaticResource myDataSource}"/>
 </DockPanel.DataContext>
 <Button Background="{Binding Path=ColorName}"
 Width="150" Height="30">I am bound to be RED!</Button>
</DockPanel>
```

Like the example above:

```
<DockPanel.Resources>
 <c:MyData x:Key="myDataSource"/>
</DockPanel.Resources>
<Button Width="150" Height="30"
 Background="{Binding Source={StaticResource myDataSource},
 Path=ColorName}">I am bound to be RED!</Button>
```

除了在元素上直接设置 `DataContext` 属性、从上级继承 `DataContext` 值 ( 如第一个示例中的按钮 ) 、通过设置 `Binding` 上的 `Source` 属性来显式指定绑定源 ( 如最后一个示例中的按钮 ) ，还可以使用 `ElementName` 属性或 `RelativeSource` 属性指定绑定源。当绑定到应用程序中的其他元素时 ( 例如在使用滑块调整按钮的宽度时 ) ，`ElementName` 属性是很有用的。当在 `ControlTemplate` 或 `Style` 中指定绑定时，`RelativeSource` 属性是很有用的

请注意，虽然我们已强调要使用的值的 `Path` 是绑定的四个必需组件之一，但在要绑定到整个对象的情况下，要使用的值会与绑定源对象相同。在这些情况下，不指定 `Path` 比较合适

```
<ListBox ItemsSource="{Binding}"
 IsSynchronizedWithCurrentItem="true"/>
```

当未指定路径时，默认为绑定到整个对象。换句话说，在此示例中路径已被省略，因为要将 `ItemsSource` 属性绑定到整个对象。

相关类 `BindingExpression` 是维持源与目标之间的连接的基础对象。一

例如，请看下面的示例，其中 `myDataObject` 是 `MyData` 类的实例，`myBinding` 是源 `Binding` 对象，`MyData` 类是包含一个名为 `MyDataProperty` 的字符串属性的已定义类。此示例将 `mytext` (`TextBlock` 的实例) 的文本内容绑定到 `MyDataProperty`。

```
//make a new source
MyData myDataObject = new MyData(DateTime.Now);
Binding myBinding = new Binding("MyDataProperty");
myBinding.Source = myDataObject;
myText.SetBinding(TextBlock.TextProperty, myBinding);
```

可以通过对数据绑定对象调用 `GetBindingExpression` 的返回值来获取 `BindingExpression` 对象。以下主题演示 `BindingExpression` 类的一些用法：



```
[ValueConversion(typeof(Color), typeof(SolidColorBrush))]
public class ColorBrushConverter : IValueConverter
{
 public object Convert(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 Color color = (Color)value;
 return new SolidColorBrush(color);
 }

 public object ConvertBack(object value, Type targetType, object parameter,
 System.Globalization.CultureInfo culture)
 {
 return null;
 }
}
```

Binding to the itemcontrols:



正如此图中所示，若要将 `ItemControl` 绑定到集合对象，应使用 `ItemsSource` 属性。可以将 `ItemsSource` 属性视为 `ItemControl` 的内容。请注意，绑定是 `OneWay`，因为 `ItemsSource` 属性默认情况下支持 `OneWay` 绑定。

一旦 `ItemControl` 绑定到数据集合，您可能希望对数据进行排序、筛选或分组。若要执行此操作，可以使用集合视图，这是实现 `ICollectionView` 接口的类

### 什么是集合视图？

可以将集合视图视为位于绑定源集合顶部的层，您可以通过它使用排序、筛选和分组查询来导航和显示源集合，所有这些操作都无需操作基础源集合本身。如果源集合实现了 `INotifyCollectionChanged` 接口，则 `CollectionChanged` 事件引发的更改将传播到视图

将视图用作绑定源并不是创建和使用集合视图的唯一方式

Sorting:

```

 private void AddSorting(object sender, RoutedEventArgs args)
 {

 listingDataView.SortDescriptions.Add(new SortDescription("Category",
ListSortDirection.Ascending));
 listingDataView.SortDescriptions.Add(
new SortDescription("StartDate", ListSortDirection.Ascending));
 }

```

Datatemplate:

```

<DataTemplate DataType="{x:Type src:AuctionItem}">
 <Border BorderThickness="1" BorderBrush="Gray"
 Padding="7" Name="border" Margin="3" Width="500">
 <Grid>
 <Grid.RowDefinitions>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
 <ColumnDefinition Width="20"/>
 <ColumnDefinition Width="86"/>
 <ColumnDefinition Width="*/>
 </Grid.ColumnDefinitions>

 <Polygon Grid.Row="0" Grid.Column="0" Grid.RowSpan="4"
 Fill="Yellow" Stroke="Black" StrokeThickness="1"
 StrokeLineJoin="Round" Width="20" Height="20"
 Stretch="Fill"
 Points="9,2 11,7 17,7 12,10 14,15 9,12 4,15 6,10 1,7 7,7"
 Visibility="Hidden" Name="star"/>

 <TextBlock Grid.Row="0" Grid.Column="1" Margin="0,0,8,0"
 Name="descriptionTitle"
 Style="{StaticResource smallTitleStyle}">Description:</TextBlock>
 <TextBlock Name="DescriptionDTData Type" Grid.Row="0" Grid.Column="2"
 Text="{Binding Path=Description}"
 Style="{StaticResource textStyleTextBlock}"/>

 <TextBlock Grid.Row="1" Grid.Column="1" Margin="0,0,8,0"
 Name="currentPriceTitle"
 Style="{StaticResource smallTitleStyle}">Current Price:</TextBlock>
 <StackPanel Grid.Row="1" Grid.Column="2" Orientation="Horizontal">
 <TextBlock Text="$" Style="{StaticResource textStyleTextBlock}"/>
 <TextBlock Name="CurrentPriceDTData Type"
 Text="{Binding Path=CurrentPrice}"
 Style="{StaticResource textStyleTextBlock}"/>
 </StackPanel>
 </Grid>
 </Border>
 <DataTemplate.Triggers>
 <DataTrigger Binding="{Binding Path=SpecialFeatures}">

```



```

 <DataTrigger.Value>
 <src:SpecialFeatures>Color</src:SpecialFeatures>
 </DataTrigger.Value>
 <DataTrigger.Setters>
 <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
 <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
 <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
 <Setter Property="BorderThickness" Value="3" TargetName="border" />
 <Setter Property="Padding" Value="5" TargetName="border" />
 </DataTrigger.Setters>
 </DataTrigger>
 <DataTrigger Binding="{Binding Path=SpecialFeatures}">
 <DataTrigger.Value>
 <src:SpecialFeatures>Highlight</src:SpecialFeatures>
 </DataTrigger.Value>
 <Setter Property="BorderBrush" Value="Orange" TargetName="border" />
 <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
 <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
 <Setter Property="Visibility" Value="Visible" TargetName="star" />
 <Setter Property="BorderThickness" Value="3" TargetName="border" />
 <Setter Property="Padding" Value="5" TargetName="border" />
 </DataTrigger>
</DataTemplate.Triggers>
</DataTemplate>

```

Data validation:

```

<TextBox Name="StartPriceEntryForm" Grid.Row="2" Grid.Column="1"
 Style="{StaticResource textStyleTextBox}" Margin="8, 5, 0, 5">
 <TextBox.Text>
 <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">
 <Binding.ValidationRules>
 <ExceptionValidationRule />
 </Binding.ValidationRules>
 </Binding>
 </TextBox.Text>
</TextBox>

```

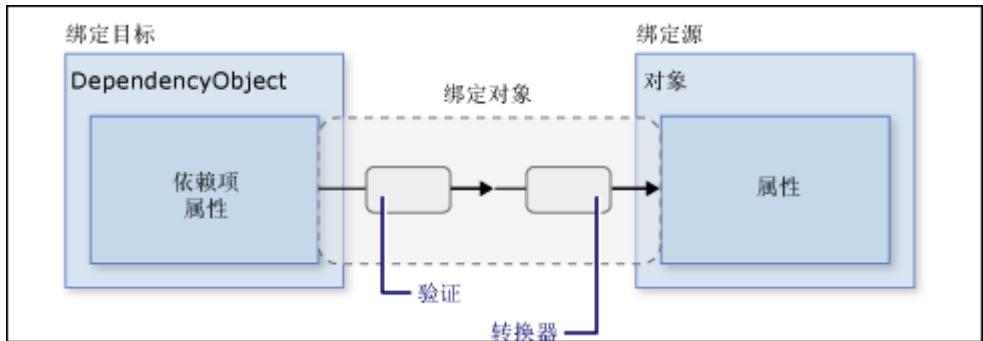
Style:

```

<Style x:Key="textStyleTextBox" TargetType="TextBox">
 <Setter Property="Foreground" Value="#333333" />
 <Setter Property="MaxLength" Value="40" />
 <Setter Property="Width" Value="392" />
 <Style.Triggers>
 <Trigger Property="Validation.HasError" Value="true">
 <Setter Property="ToolTip"
 Value="{Binding RelativeSource={RelativeSource Self},
 Path=(Validation.Errors)[0].ErrorContent}" />
 </Trigger>
 </Style.Triggers>

```

</Style>  
Validation:



Simple databinding:

```
<Window
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:src="clr-namespace:SDKSample"
 SizeToContent="WidthAndHeight"
 Title="Simple Data Binding Sample">

 <Window.Resources>
 <src:Person x:Key="myDataSource" PersonName="Joe"/>
 </Window.Resources>
 <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}" />

</Window>
```

Equals:

```
<ObjectDataProvider x:Key="myperon" ObjectType="{x:Type src:Person}">
 <ObjectDataProvider.ConstructorParameters>
 <System:String>Joe</System:String>
 </ObjectDataProvider.ConstructorParameters>
</ObjectDataProvider>
```

属性	说明
Source	可以使用此属性来将源设置为对象的实例。如果您不需要建立范围（在此范围内若干属性继承同一数据上下文）的功能，您可以使用 <b>Source</b> 属性，而不是 <b>DataContext</b> 属性。有关更多信息，请参见 <b>Source</b> 。
RelativeSource	如果您希望指定相对于绑定目标位置的源，这是有用的。当您想要将元素的一个属性绑定到同一元素的另一个属性时，或者如果您正在样式或模板中定义绑定，您可能需要使用此属性。有关更多信息，请参见 <b>RelativeSource</b> 。
ElementName	您指定一个表示您希望绑定到的元素的字符串。当您希望绑定到应用程序中的另一个元素的属性时，这是有用的。例如，如果您希望使用 <b>Slider</b> 控制应用程序中另一个控件的高度，或者如果您希望将控件的 <b>Content</b> 绑定到 <b>ListBox</b> 控件的 <b>SelectedValue</b> 属性。有关更多信息，请参见 <b>ElementName</b>

UpdateresourceTrigger:

```

<Label>Enter a Name:</Label>
 <TextBox>
 <TextBox.Text>
 <Binding Source="{StaticResource myDataSource}" Path="Name"
 UpdateSourceTrigger="PropertyChanged"/>
 </TextBox.Text>
 </TextBox>

 <Label>The name you entered:</Label>
 <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=Name}"/>

```

如果将 **UpdateSourceTrigger** 值设置为 **Explicit**，则仅当应用程序调用 **UpdateSource** 方法时，该源值才会发生更改。下面的示例演示如何为 **itemNameTextBox** 调用 **UpdateSource**：

必须显示调用：**BindingExpression.UpdateSource()**。

```

BindingExpression be =myText . GetBindingExpression(TextBox.TextProperty);
be.UpdateSource();

```

集合绑定源对象要求：

- 实现 **IEnumerable** 接口支持集合元素遍历
- 实现 **INotifyCollectionChanged** 支持动态更新通知
- Utility: WPF 提供有方便的 **ObservableCollection** 类

绑定目标要求：

- 继承 **ItemsControl** 类，并实现 **ItemsSource** 与 UI 界

面的交互逻辑，成为一个集合控件

Databinding:

```

myData data=new myData("hello");
Binding bind = new Binding();
TextBox mytextbox= new TextBox();
bind.Source = data;
bind.Mode =BindingMode.OneWay ; //twoway, onewaytoSource, onetime, default.
bind.UpdateSourceTrigger = UpdateSourceTrigger.PropertyChanged;
bb.SetBinding(TextBox.TextProperty, bind);

```

使用XAML存储的资源数据将以序列化的形式存储在BAML文件中，然后在需要绑定时再重建对象。使用代码存储的资源对象与普通数据应用类似

WPF: compile proceee, C#-->.DLL, and XAML→BAML(二进制处理), 并不是所有的XAML都可以compile成BAML, 有些要同C# code 进行协同处理。

WPF资源范围

- WPF将资源划归到一些不同类型的范围中：
- 系统级资源(System)
  - 应用程序级资源(Application)
    - 窗体级资源(Window)
      - 元素级资源(Element)

- 页面级自由 (Page)
- 元素级资源 (Element)
- 系统级的资源在整个系统范围可见，应用程序级的资源在应用程序范围可见，依次类推。
- 可以使用XAML声明绑定资源，也可以使用 FindResource 或 TryFindResource 在代码中查找资源

```
this.Resources.Add("mybrush",new SolidColorBrush (Colors .Red));
<SolidColorBrush x:Key="mybrush" Color="Red" />
```

- WPF根据资源的绑定/辨析规则，将资源的绑定/辨析分为两类：静态资源与动态资源。注意：资源本身没有静态和动态之区别，静态与动态的区别在于资源的绑定与辨析。
  - 对于静态资源，WPF在第一次绑定时辨析，以后每次绑定使用第一次辨析得到的值；对于动态资源，WPF在每次绑定时都进行一次全新的辨析。
- Dynamicresource：一个表达式，每次运行都要重新计算一下。

### 静态资源

- 由于以二进制序列化形式存储，静态资源的效率高，比较节省内存和绑定时间。
- 但是由于是第一次绑定时辨析，灵活性差，程序初始化资源一经确定，不能再改变。
- 比较适合常规的页面级或程序级的资源处理。

### 动态资源：

- 由于每次绑定时都进行一次全新的辨析，也就是创建全新的对象（同时将先前的对象丢弃为垃圾），所以动态资源比较浪费内存和绑定时间。
- 但是由于是每次绑定时都进行辨析，灵活性高，程序在运行期可以任意更改资源。
- 比较适合常规的系统级或用户频繁存取的资源处理

```
<Button Background="{DynamicResource mybrush}" Width="50" Height="20" Click="buttonClick" >
<Button Background="{StaticResource mybrush}" Width="50" Height="20" Click="buttonClick" >
void buttonClick(object sender, RoutedEventArgs e)
{
 this.Resources["mybrush"] = new SolidColorBrush(Colors.Green);
 //if staticResource , then button's background will not change.
 //if nynamicResource ,then change.
}
```

Style:

- WPF 通过Style的方式简化了组件属性（特别是针对一系列组件的应用）的表达，大大提升了业务逻辑和UI分离的灵活性。
- 声明性编程使得WPF Style支持非常灵活的具体应用编程。
  - WPF Style通常作为一种资源以声明的方式存储在XAML文件中。也可以使用程序代码来在资源中动态添加/删除/更改Style。

### Style的绑定规则

- 使用TargetType可以指定Style应用的控件类型。指定TargetType后，在当前Style的作用范围内，所有该类型的控件都将自动应用该Style。
- 除了TargetType之外，还可以使x:Key= "MyButton" 来根据资源绑定规则来为某一特定控件指定特定的Style。
- Style通常作为一种资源形式来存储，因此它的绑定规则也符合WPF资源的绑定范围规则。

### Style的扩展

- 可以使用BasedOn属性来扩展某一Style，扩展意味着在继承中有所改写。

```
<Style TargetType="Button" >
 <Setter Property="Foreground" Value="Red"/>
 <Setter Property="FontSize" Value="14"/>
</Style>
<Style BasedOn="{StaticResource {x:Type Button}}"
```

```

TargetType="Button" x:Key="MyButton">
 <Setter Property="Foreground" Value="Green"/>
 <Setter Property="Background" Value="Yellow"/>
 <Setter Property="FontSize" Value="20"/>
</Style>

```

### Template:

WPF Style 只能更改WPF组件结构中的属性值，但是不能更改组件结构本身。如果需要灵活地定制组件的结构，需要使用WPF模板。

- WPF模板有数据模板和控件模板。数据模板可以将数据以另外一种不同的表现形式，实际上是对数据实施一种批次处理。控件模板可以改变WPF控件的结构和表现形式，从而实现更为灵活的控件功能与形式。

### 数据模板

- DateType指定了只要遇到该类型（Photo）的对象，则使用当前的DateTemplate进行改变

```

<Window.Resources>
 <DataTemplate DataType="{x:Type local:Photo}">
 <Border Margin="3">
 <Image Source="{Binding Source}"/>
 </Border>
 </DataTemplate>
</Window.Resources> >

```

### 控件模板

- TargetType指定了要应用该模板的控件类型，Template制定了这是在更改一个控件的缺省模板。

```

<Style TargetType="Button">
 <Setter Property="OverridesDefaultStyle" Value="True"/>
 <Setter Property="Template">

```

### UpdateSourceTrigger:

```

<TextBox x:Name="mytext" >
 <TextBox.Text >
 <Binding Source="{StaticResource myperson}" Path="PersonName"
UpdateSourceTrigger="Explicit" Mode="TwoWay" />
 </TextBox.Text>
</TextBox>

```

### Code:

```

void buttonClick(object sender, RoutedEventArgs e)
{
 //mytext is defined in XAML, which is a instance of a textbox.
 BindingExpression be = mytext.GetBindingExpression(TextBox.TextProperty);
 be.UpdateSource();
}

```

UpdateSourceTrigger 属性用于处理源更新，因此仅适用于 TwoWay 或 OneWay 绑定。若要使 TwoWay 和 OneWay 绑定生效，源对象需要提供属性更改通知

- 无论是目标属性还是源属性，只要发生了更改，TwoWay 就会更新目标属性或源属性。
- OneWay 仅当源属性发生改变时更新目标属性。
- OneTime 仅当应用程序启动时或 DataContext 进行更改时更新目标属性。
- OneWayToSource 在目标属性更改时更新源属性。
- Default: 使用目标属性的默认 Mode 值。

Databinding:

<Window.Resources >

```
<src:Person x:Key="myperson" FirstName="zhou" LastName="wang" HomeTown="zheng" />
<DataTemplate x:Key="DetailTemplate">
 <Border Width="300" Height="100" Margin="20"
 BorderBrush="Aqua" BorderThickness="1" Padding="8">
 <Grid>
 <Grid.RowDefinitions>
 <RowDefinition/>
 <RowDefinition/>
 <RowDefinition/>
 </Grid.RowDefinitions>
 <Grid.ColumnDefinitions>
 <ColumnDefinition/>
 <ColumnDefinition/>
 </Grid.ColumnDefinitions>
 <TextBlock Grid.Row="0" Grid.Column="0" Text="First Name:"/>
 <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=FirstName}" />
 <TextBlock Grid.Row="1" Grid.Column="0" Text="Last Name:"/>
 <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=LastName}" />
 <TextBlock Grid.Row="2" Grid.Column="0" Text="Home Town:"/>
 <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=HomeTown}" />
 </Grid>
 </Border>
</DataTemplate>
</Window.Resources>
<StackPanel>
 <TextBlock FontFamily="Verdana" FontSize="11"
 Margin="5,15,0,10" FontWeight="Bold">My Friends:</TextBlock>

 <ListBox Width="200" IsSynchronizedWithCurrentItem="True"
 ItemsSource="{Binding Source={StaticResource myperson}}"/>

 <TextBlock FontFamily="Verdana" FontSize="11"
 Margin="5,15,0,5" FontWeight="Bold">Information:</TextBlock>

 <ContentControl Content="{Binding Source={StaticResource myperson}}"
 ContentTemplate="{StaticResource DetailTemplate}" />
</StackPanel>
```

Note:

1. **ListBox** 和 **ContentControl** 绑定到同一个源。两个绑定的 **Path** 属性均未指定，因为两个控件都绑定到整个集合对象。
2. 为此，必须将 **IsSynchronizedWithCurrentItem** 属性设置为 **true**。设置此属性可以确保选定项始终设置为 **CurrentItem**。或者，如果 **ListBox** 从 **CollectionViewSource** 获取数据则它将自动同步选定内容和当前内容。

Binding to the enum:

<Window.Resources >

```
<ObjectDataProvider MethodName="GetValues"
 ObjectType="{x:Type sys:Enum}"
 x:Key="AlignmentValues">
```

```

 <ObjectDataProvider.MethodParameters>
 <x:Type TypeName="HorizontalAlignment" />
 </ObjectDataProvider.MethodParameters>
 </ObjectDataProvider>

</Window.Resources>
<Border Margin="10" BorderBrush="Aqua"
 BorderThickness="3" Padding="8">
 <StackPanel Width="300">
 <TextBlock>Choose the HorizontalAlignment value of the Button:</TextBlock>
 <ListBox Name="myComboBox" SelectedIndex="0" Margin="8" ItemsSource="{Binding
Source={StaticResource AlignmentValues}}"/>
 <Button Content="Click Me!" HorizontalAlignment="{Binding ElementName=myComboBox,
Path=SelectedItem}"/>
 </StackPanel>
</Border>

```

Binding the property of two controls:

```

<StackPanel >
 <ComboBox SelectedIndex=" 0" Height="30" Width=" 50" Name="mycomboBox" >
 <ComboBoxItem >Green</ComboBoxItem>
 <ComboBoxItem >Red</ComboBoxItem>
 <ComboBoxItem >Yellow</ComboBoxItem>
 </ComboBox>
 <Canvas Width=" 100" Height=" 50" >
 <Canvas.Background >
 <Binding ElementName="mycomboBox" Path="SelectedItem.Content" />
 </Canvas.Background>
 </Canvas>
</StackPanel>

```

**Note:**绑定目标属性（在本示例中是 **Background** 属性）必须是一个依赖项属性

## 如何：实现绑定验证

此示例演示如何基于自定义的验证规则，使用 **ErrorTemplate** 和样式触发器来提供可视反馈，以便在输入无效值时向用户发出通知。

```

<TextBox Name="textBox1" Width="50" FontSize="15"
 Validation.ErrorTemplate="{StaticResource validationTemplate}"
 Style="{StaticResource textBoxInError}"
 Grid.Row="1" Grid.Column="1" Margin="2">
 <TextBox.Text>
 <Binding Path="Age" Source="{StaticResource ods}"
 UpdateSourceTrigger="PropertyChanged" >
 <Binding.ValidationRules>
 <c:AgeRangeRule Min="21" Max="130"/>
 </Binding.ValidationRules>
 </Binding>
 </TextBox.Text>
</TextBox>

```

```
public override ValidationResult Validate(object value, CultureInfo cultureInfo)
```



```

{
 int age = 0;

 try
 {
 if (((string)value).Length > 0)
 age = Int32.Parse((String)value);
 }
 catch (Exception e)
 {
 return new ValidationResult(false, "Illegal characters or " + e.Message);
 }

 if ((age < Min) || (age > Max))
 {
 return new ValidationResult(false,
 "Please enter an age in the range: " + Min + " - " + Max + ".");
 }
 else
 {
 return new ValidationResult(true, null);
 }
}

```

下面的示例演示了自定义的 **ControlTemplate** `validationTemplate`，它用于创建一个红色感叹号，以通知用户验证错误。控件模板用于重新定义控件的外观。

```

<ControlTemplate x:Key="validationTemplate">
 <DockPanel>
 <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
 <AdornedElementPlaceholder/>
 </DockPanel>
</ControlTemplate>

```

如下面的示例中所示，显示错误消息的 **ToolTip** 是使用名为 `textBoxInError` 的样式创建的。如果 `HasError` 的值是 `true`，则触发器将当前 `TextBox` 的工具提示设置为其第一个验证错误。`RelativeSource` 设置为 `Self`，以引用当前元素。

```

<Style x:Key="textBoxInError" TargetType="{x:Type TextBox}">
 <Style.Triggers>
 <Trigger Property="Validation.HasError" Value="true">
 <Setter Property="ToolTip"
 Value="{Binding RelativeSource={x:Static RelativeSource.Self},
 Path=(Validation.Errors)[0].ErrorContent}" />
 </Trigger>
 </Style.Triggers>
</Style>

```

**Error binding:**

```

<TextBox Style="{StaticResource textBoxInError}">
 <TextBox.Text>
 <Binding Path="Age" Source="{StaticResource data}"
 ValidatesOnExceptions="True"
 UpdateSourceTrigger="PropertyChanged">
 <Binding.ValidationRules>
 <DataErrorValidationRule/>
 </Binding.ValidationRules>
 </Binding>
 </TextBox.Text>
</TextBox>

```

```
</TextBox.Text>
</TextBox>
```

#### Get binding instance:

您可以执行以下操作以获取 **Binding** 对象:

```
// textBox3 is an instance of a TextBox
// the TextProperty is the data-bound dependency property
```

```
Binding myBinding = BindingOperations.GetBinding(textBox3, TextBox.TextProperty);
```

如果您的绑定是 **MultiBinding**, 请使用 **BindingOperations.GetMultiBinding**。如果它是 **PriorityBinding**, 请使用 **BindingOperations.GetPriorityBinding**。如果您不确定目标属性是使用 **Binding**、**MultiBinding** 还是 **PriorityBinding** 绑定的, 则可以使用 **BindingOperations.GetBindingBase**。

使用视图可以用不同的方式查看同一数据集, 具体取决于排序或筛选方式。此外, 视图还提供了当前记录指针概念, 并可移动该指针。本示例演示如何创建视图对象

Get the view :

```
mycollectionview=(CollectionView) CollectionViewSource.GetDefaultView();
```

```
<StackPanel Name="mypanel" >
<StackPanel.DataContext>
 <Binding Source="{StaticResource myDataSource}" />
</StackPanel.DataContext>
```

```
</StackPanel>
```

然后, 您可以应用筛选器, 如下例所示。在本示例中, **myCollectionView** 是一个 **ListCollectionView** 对象。

```
myCollectionView.Filter = new Predicate<object>(Contains);
```

若要撤消筛选, 可以将 **Filter** 属性设置为 **null**:

```
myCollectionView.Filter = null;
```

如果您的视图对象来自 **CollectionViewSource** 对象, 则可以通过为 **Filter** 事件设置事件处理程序来应用筛选逻辑。在下面的示例中, **listingDataView** 是 **CollectionViewSource** 的一个实例。

```
listingDataView.Filter += new FilterEventHandler(ShowOnlyBargainsFilter);
```

```
private void ShowOnlyBargainsFilter(object sender, FilterEventArgs e)
{
 AuctionItem product = e.Item as AuctionItem;
 if (product != null)
 {
 // Filter out products with price 25 or above
 if (product.CurrentPrice < 25)
 {
 e.Accepted = true;
 }
 else
 {
 e.Accepted = false;
 }
 }
}
```

This class implements **INotifyPropertyChanged**:

```
public class Person : INotifyPropertyChanged
{
 private string name;
 // Declare the event
 public event PropertyChangedEventHandler PropertyChanged;

 public Person()
```

```

 {
 }

 public Person(string value)
 {
 this.name = value;
 }

 public string PersonName
 {
 get { return name; }
 set
 {
 name = value;
 // Call OnPropertyChanged whenever the property is updated
 OnPropertyChanged("PersonName");
 }
 }

 // Create the OnPropertyChanged method to raise the event
 protected void OnPropertyChanged(string name)
 {
 PropertyChangedEventHandler handler = PropertyChanged;
 if (handler != null)
 {
 handler(this, new PropertyChangedEventArgs(name));
 }
 }
}

```

创建和绑定到 **ObservableCollection**

```

public class nameList : ObservableCollection<PersonName>
{
 public nameList () : Base()
 {
 add (new PersonName("a", "b"));
 add (new PersonName("a", "b"));
 add (new PersonName("a", "b"));
 }
}

```

如何：绑定到方法

在本示例中，**TemperatureScale** 是一个类，它有一个方法 **ConvertTemp**，该方法将接收两个参数（一个是 **double** 类型，另一个是 **enum** 类型 **TempType**），并将给定值从一个温标转换为另一个温标。在下面的示例中，**ObjectDataProvider** 用于实例化 **TemperatureScale** 对象。将使用两个指定参数调用 **ConvertTemp** 方法。

```

<Window.Resources>
 <ObjectDataProvider ObjectType="{x:Type local:TemperatureScale}"
 MethodName="ConvertTemp" x:Key="convertTemp">
 <ObjectDataProvider.MethodParameters>
 <system:Double>0</system:Double>
 <local:TempType>Celsius</local:TempType>
 </ObjectDataProvider.MethodParameters>
 </ObjectDataProvider>

 <local:DoubleToString x:Key="doubleToString" />

```

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

方法可以作为资源使用，因此您可绑定到其结果。在以下示例中，**TextBox** 的 **Text** 属性和 **ComboBox** 的 **SelectedValue** 绑定到方法的两个参数。用户可借此指定要转换到的温度以及要转换自的温标。请注意，**BindsDirectlyToSource** 设置为 **true**，因为我们要绑定到 **ObjectDataProvider** 实例的 **MethodParameters** 属性，而不是绑定到由 **ObjectDataProvider** 包装的对象（**TemperatureScale** 对象）的属性。

```
<Label Grid.Row="1" HorizontalAlignment="Right">Enter the degree to convert:</Label>
<TextBox Grid.Row="1" Grid.Column="1" Name="tb">
 <TextBox.Text>
 <Binding Source="{StaticResource convertTemp}" Path="MethodParameters[0]"
 BindsDirectlyToSource="true" UpdateSourceTrigger="PropertyChanged"
 Converter="{StaticResource doubleToString}">
 <Binding.ValidationRules>
 <local:InvalidCharacterRule/>
 </Binding.ValidationRules>
 </Binding>
</TextBox.Text>
</TextBox>
<ComboBox Grid.Row="1" Grid.Column="2" SelectedValue="{Binding Source={StaticResource convertTemp},
 Path=MethodParameters[1], BindsDirectlyToSource=true}">
 <local:TempType>Celsius</local:TempType>
 <local:TempType>Fahrenheit</local:TempType>
</ComboBox>
<Label Grid.Row="2" HorizontalAlignment="Right">Result:</Label>
<Label Content="{Binding Source={StaticResource convertTemp}}"
 Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"/>
```

Set the notify when Binding is updated:

将绑定中的 **NotifyOnTargetUpdated** 或 **NotifyOnSourceUpdated** 属性（或两者）设置为 **true**。您提供的用于侦听此事件的处理程序必须直接附加到您希望收到更改通知的元素，或者如果您希望在上下文中的任何内容发生变化时得到通知，则附加到整个数据上下文。

```
<TextBlock Grid.Row="1" Grid.Column="1" Name="RentText"
 Text="{Binding Path=Rent, Mode=OneWay, NotifyOnTargetUpdated=True}"
 TargetUpdated="OnTargetUpdated"/>
```

若要从对象的个别属性清除绑定，请按下例所示调用 **ClearBinding**。下面的示例会从 **mytext** 的 **TextProperty** 中移除绑定，前者是一个 **TextBlock** 对象。

```
BindingOperations.ClearBinding(myText, TextBlock.TextProperty);
```

查找由 **DataTemplate** 生成的元素：

```
<ListBox Name="myListBox" ItemTemplate="{StaticResource myDataTemplate}"
 IsSynchronizedWithCurrentItem="True">
 <ListBox.ItemsSource>
 <Binding Source="{StaticResource InventoryData}" XPath="Books/Book"/>
 </ListBox.ItemsSource>
</ListBox>
//

<DataTemplate x:Key="myDataTemplate">
 <TextBlock Name="textBlock" FontSize="14" Foreground="Blue">
 <TextBlock.Text>
 <Binding XPath="Title"/>
 </TextBlock.Text>
</TextBlock>
</DataTemplate>
```

如果要检索由某个 [ListBoxItem](#) 的 [DataTemplate](#) 生成的 [TextBlock](#) 元素，您需要获得 [ListBoxItem](#)，在该 [ListBoxItem](#) 内查找 [ContentPresenter](#)，然后对在该 [ContentPresenter](#) 上设置的 [DataTemplate](#) 调用 [FindName](#)。下面的示例演示如何执行这些步骤。出于演示的目的，本示例创建一个消息框，用于显示由 [DataTemplate](#) 生成的文本块的文本内容。

```
ListBoxItem myListBoxItem =
(ListBoxItem) (myListBox.ItemContainerGenerator.ContainerFromItem(myListBox.Items.CurrentItem));
ContentPresenter myContentPresenter = FindVisualChild<ContentPresenter>(myListBoxItem);

DataTemplate myDataTemplate = myContentPresenter.ContentTemplate;
TextBlock myTextBlock = (TextBlock)myDataTemplate.FindName("textBlock", myContentPresenter);

// Do something to the DataTemplate-generated TextBlock
MessageBox.Show("The text of the TextBlock of the selected list item: " + myTextBlock.Text);
```

Using the visualTreeHelper to find the datatemplate element.

```
private childItem FindVisualChild<childItem>(DependencyObject obj)
 where childItem : DependencyObject
{
 for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
 {
 DependencyObject child = VisualTreeHelper.GetChild(obj, i);
 if (child != null && child is childItem)
 return (childItem)child;
 else
 {
 childItem childOfChild = FindVisualChild<childItem>(child);
 if (childOfChild != null)
 return childOfChild;
 }
 }
 return null;
}
```