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 (scRGB)，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该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

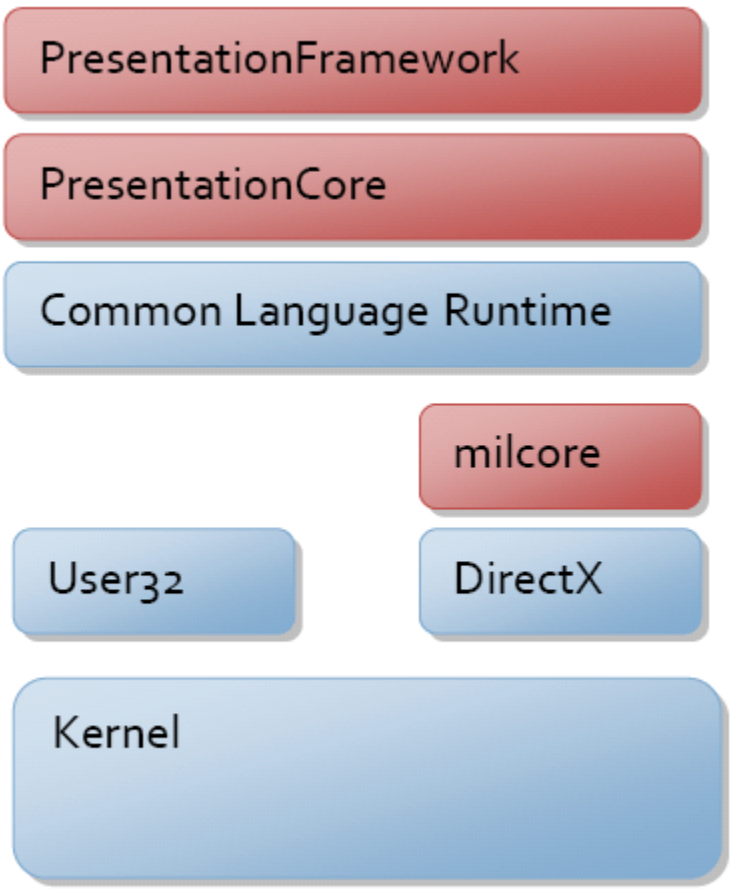
FlowDocumentPageViewer、FlowDocumentScrollViewer 和 FlowDocumentReader 各有一个名为 Selection 的新的公共属性。该属性获取表示文档中选定内容的 TextSelection。

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加属性的概念

  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 创建的对象的代码和调试访问。

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，但如果在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

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 x:Key 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 ResourceDictionary..::.Add 时作为 *key* 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复本

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。这种微妙的路由事件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一个依赖项属性的整体 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 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类行为，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：

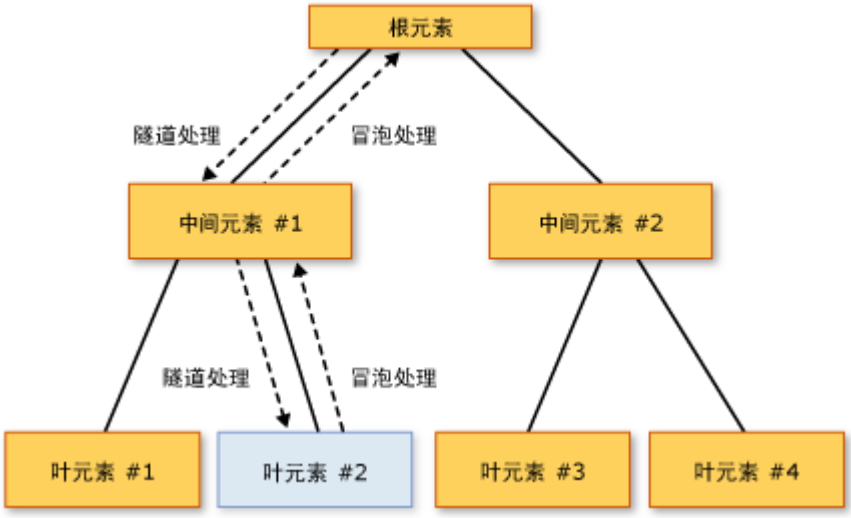
在代码中，不使用适用于一般 CLR 事件的特定于语言的事件语法，而是通过调用 WPF 方法 AddHandler(RoutedEvent, Delegate, Boolean) 来添加处理程序。使用此方法时，请将 *handledEventsToo* 的值指定为 true。
在 EventSetter 中，请将 HandledEventsToo 属性设置为 true。

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="b1SetColor" />
          </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 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 neweEventArgs = new RoutedEventArgs(mySimpleButton.TapEvent);
            RaiseEvent(neweEventArgs);
        }
        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合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 的本机大小属性，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>
```

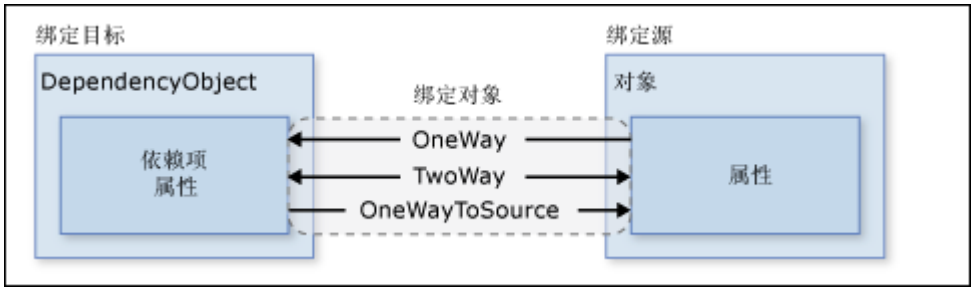
Control:

- 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（TextBox.Text 的默认值）	当 TextBox 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 TextBox
PropertyChanged	当键入到 TextBox 中时	聊天室窗口中的 TextBox 控件
Explicit	当应用程序调用 UpdateSource 时	可编辑窗体中的 TextBox 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"/>
```

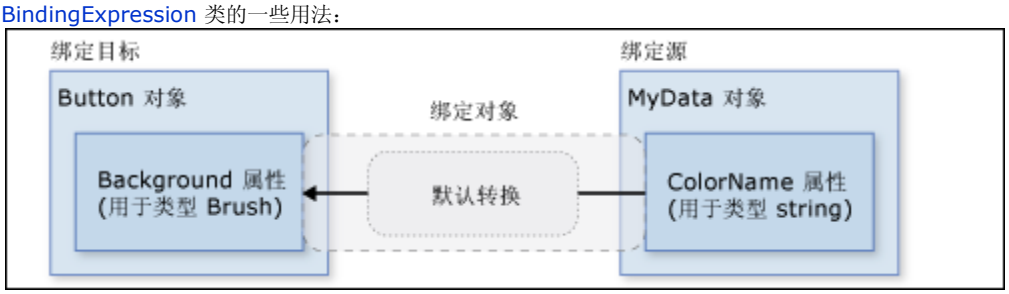
当未指定路径时，默认为绑定到整个对象。换句话说，在此示例中路径已被省略，因为要将 ItemsSource 属性绑定到整个对象。

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);
```

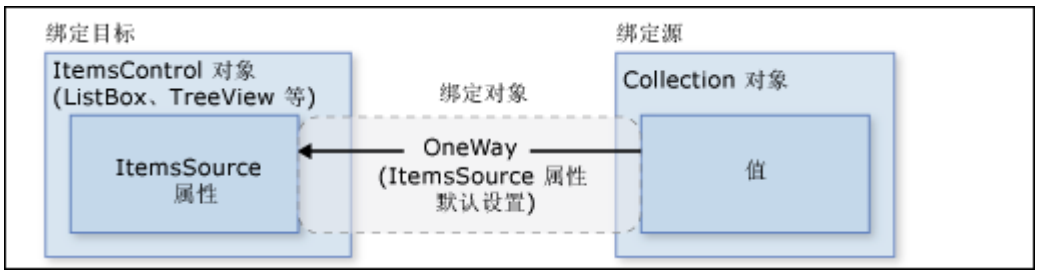
可以通过对数据绑定对象调用 GetBindingExpression 的返回值来获取 BindingExpression 对象。以下主题演示 BindingExpression 类的一些用法：




```
[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 ItemsControl 绑定到集合对象，应使用 ItemsSource 属性。可以将 ItemsSource 属性视为 ItemsControl 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 ItemsControl 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="DescriptionDTDataType" Grid.Row="0" Grid.Column="2"
                Text="{Binding Path=Description}"
                Style="{StaticResource textStyleTextBlock}"/>

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

```
                    <DataTrigger.Value>
                        <src:SpecialFeatures>Color</src:SpecialFeatures>
                    </DataTrigger.Value>
                    <DataTrigger.Setters>
                        <Setter Property="BorderBrush" Value="DodgerBlue" TargetName="border" />
                        <Setter Property="Foreground" Value="Navy" TargetName="descriptionTitle" />
                        <Setter Property="Foreground" Value="Navy" TargetName="currentPriceTitle" />
                        <Setter Property="BorderThickness" Value="3" TargetName="border" />
                        <Setter Property="Padding" Value="5" TargetName="border" />
                    </DataTrigger.Setters>
                </DataTrigger>
                <DataTrigger 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>
```

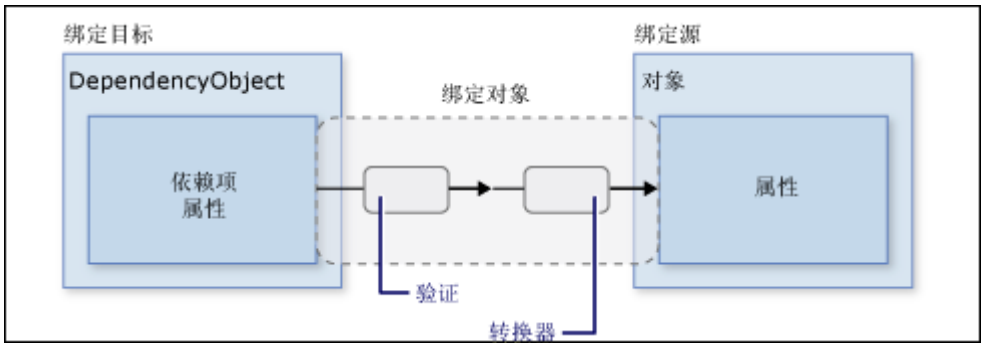
Data validation:

```
<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 Source 属性，而不是 DataContext 属性。有关更多信息，请参见 Source。
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 RelativeSource。
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 Slider 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 Content 绑定到 ListBox 控件的 SelectedValue 属性。有关更多信息，请参见 ElementName

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 仅当源属性发生更改时更新目标属性。
- 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="mycombox" >
            <ComboBoxItem >Green</ComboBoxItem>
            <ComboBoxItem >Red</ComboBoxItem>
            <ComboBoxItem >Yellow</ComboBoxItem>
        </ComboBox>
        <Canvas Width=" 100" Height=" 50" >
            <Canvas.Background >
                <Binding ElementName="mycombox" 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>
```

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合，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 updted:
```

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;
}
```

WPF 知识总结

Visual Tree and Logic Tree:

Attention! 以下各主题为 WPF 中的重要新特性、新概念，掌握了这些内容之后再去阅读其它书籍将会事半功倍。

逻辑树和视觉树（Logical and Visual Tree）

在 WPF 中，用户界面以对象树的形式构建，称作“逻辑树”。

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      Title="关于 WPF 揭秘" SizeToContent="WidthAndHeight"
      Background="OrangeRed">
      <StackPanel>
          <Label FontWeight="Bold" FontSize="20" Foreground="White">
              WPF 揭秘（版本 3.0）
          </Label>
          <Label>(C)2006 SAMS 出版公司</Label>
          <Label>已安装的章节：</Label>
          <ListBox>
              <ListBoxItem>第一章</ListBoxItem>
```

```
            <ListBoxItem>第二章</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>您已经成功注册了本产品。</StatusBar>
         </StackPanel>
</Window>
```

逻辑树并不只存在于由 XAML 创建的 WPF 用户界面中，上例完全可由程序代码写成，同样存在逻辑树。逻辑树是一个很直接的概念，既然这样我们为什么还要关心它呢？这是因为，几乎 WPF 的每个方面（如属性、事件、资源等等）都有依赖逻辑树的行为。例如，属性值有时候会自动沿着逻辑树向下传导，并且可以沿着逻辑树向上或向下触发事件。

上例对应的逻辑树为：

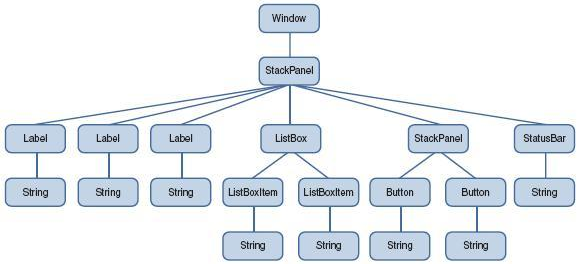


图 1

视觉树的概念与逻辑树相似，它是对逻辑树的基本扩展。视觉树中的节点穿透了它们的核心视觉组件（看图 2，意思是说视觉树的节点夹在逻辑树的节点之间，逻辑树的节点可看作核心视觉组件），并且暴露了视觉实现的细节，而不是将树上的每一个元素当作黑盒子。比如说，尽管 ListBox 逻辑上是一个单独的控件，但是它的**默认视觉实现**（确实是默认的，因为这个视觉实现方式可以修改！）则是由许多 WPF 基本元素构成：一个边框（Border）和两个滚动条（ScrollBar）等。

并不是所有的逻辑树节点都会出现在视觉树中，只有那些从 System.Windows.Media.Visual 或从 System.Windows.Media.Visual3D 派生的元素才会。其他元素和简单的字符串内容都不会出现在视觉树中，因为它们不包含继承的呈现（rendering）行为。

下图表示上例在 Vista 中 Aero 主题下的默认视觉树，其中边框较粗的是在逻辑树上也有的节点。

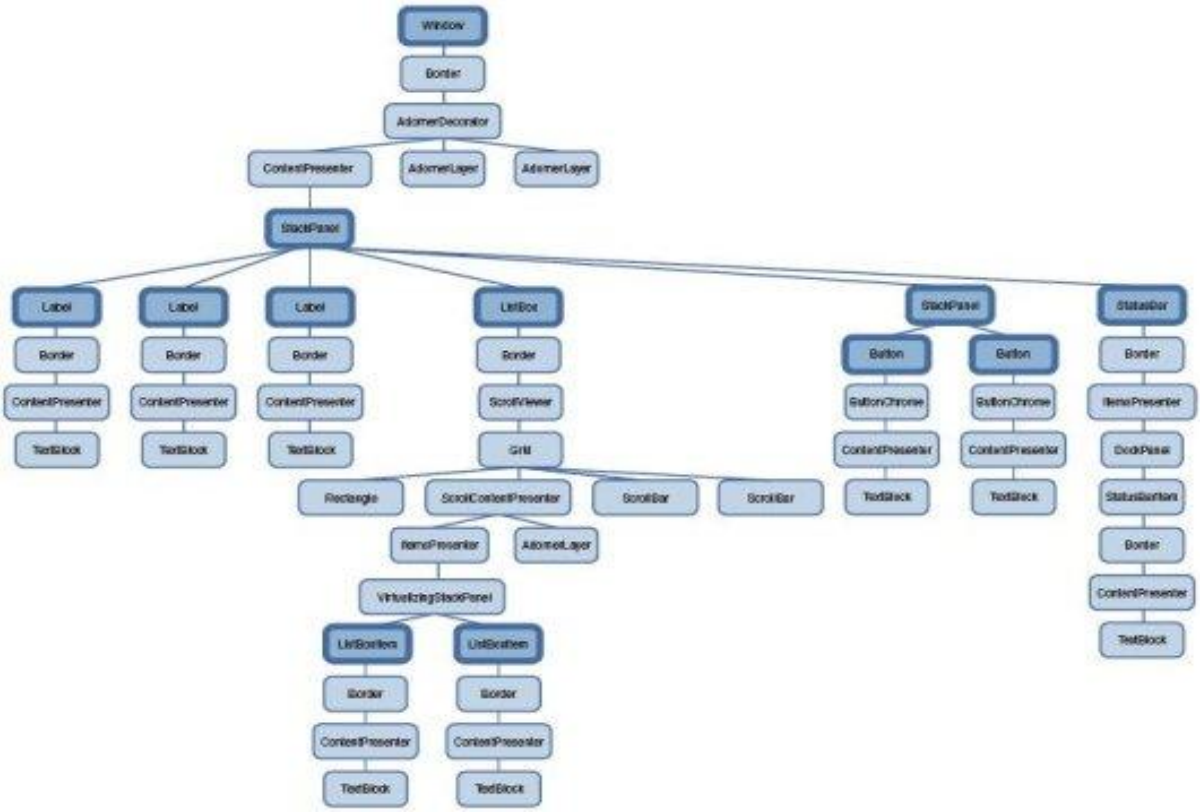


图 2

这张图暴露一些当前不可见的 UI 内部组件，如 ListBox 的两个 ScrollBar 和 Label 的 Border。除此之外，它还揭示了除按钮使用了 ButtonChrome 而不是 Border 之外，Button、Label 和 ListBoxItem 都由共同的元素组成（Border 和 ContentPresenter）。此外，由于不同的默认属性值使得这些控件拥有不同的视觉效果，如 Button 的 Margin 属性默认值是 10，而 Label 的 Margin 属性默认值则为 0。

从 WPF 元素的深度组合可以看出来，视觉树可以是难以想象得复杂。不过尽管视觉树是 WPF 基础结构的本质部分，却并不需要担心，除非我们需要彻底改变控件的风格，或者做一些低级别的绘画操作。另外需要注意到，为 Button 编写依赖于特定视觉树的代码将会损害 WPF 的核心原则之一——外观和逻辑的分离。

2.

我们可以简单地通过 System.Windows.LogicalTreeHelper 和 System.Windows.VisualTreeHelper 实现对逻辑树和视觉树的遍历。如下例：

```
using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;

public partial class AboutDialog : System.Windows.Window
{

    public AboutDialog()
```

```
        {
            InitializeComponent();
            PrintLogicalTree(0, this);
        }

        protected override void OnContentRendered(EventArgs e)
        {
            base.OnContentRendered(e);
            PrintVisualTree(0, this);
        }

        void PrintLogicalTree(int depth, object obj)
        {
            //  按深度打印缩进空白
            Debug.WriteLine(new string(' ', depth) + obj);

            //  有时候叶子节点不是 DependencyObject(如 string)
            if (!(obj is DependencyObject)) return;

            //  递归打印逻辑树的节点
            foreach (object child in LogicalTreeHelper.GetChildren(
                obj as DependencyObject))
            {
                PrintLogicalTree(depth + 1, child);
            }
        }

        void PrintVisualTree(int depth, DependencyObject obj)
        {
            //  按深度打印缩进空白
            Debug.WriteLine(new string(' ', depth) + obj);

            //  递归打印视觉树的节点
            for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
            {
                PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
            }
        }
    }
```

在 AboutDialog 的构造方法中的 InitCompoment 方法之后，可以调用 PrintLogicalTree 遍历逻辑树上的节点。然而对于视觉树，直到 AboutDialog 完成一次布局之前，它都将是空的，因此要在 OnContentRendered 方法中调用 PrintVisualTree 实现遍历。

有时候在这两棵树上的导航操作可由元素自己的实例方法来完成。比如，Visual 类包含三个受保护的成员方法 VisualParent、VisualChordCount 和 GetVisualChild，它们用来检视它的视觉父元素和子元素。比如，Button、Label 等控件的通用基类 FrameworkElement 定义了一个公共的 Parent 属性用来表示逻辑父元素。再比如，特

定的 FramewordElement 子类们以不同的方式暴露了它们的逻辑子元素。如一些类暴露了 Children 集合，其它类（如 Button 和 Label）则暴露了一个 Content 属性，用以表示这个元素只能包含一个逻辑子元素。

图 2 表示的树通常也被称作“元素树”，因为它同时包含了逻辑树元素和特定的视觉树元素。由此，术语“视觉树”用来描述任何仅包含了视觉元素的子树。例如，许多人会说 Window 的**默认**视觉树由一个 Border、一个 AdornerDecorator、两个 AdornerLayers 和一个 ContentPresenter 组成。图 2 中，顶级 StackPannel 通常不被算做 ContentPresenter 的视觉子元素，尽管它可以由 **Visual**TreeHelper 输出。

（WPF 中的这两棵树是一个非常神奇有趣的特色。我们可以看到，现在的控件不再像以前那样需要在一个或多个函数中“重绘”了。从前，如果我们想要在 VC 中制作一个自定义的菜单，需要响应 WM_DRAWITEM 和 WM_MEASUREITEM 两条消息，然后根据参数在响应函数中生画，比如先画背景，然后是左侧的图标，然后是右侧的菜单文本，总之是一个烦人的工作。这也是很多人说的，用 VC 做界面不方便的地方。后来到了 Windows Form，控件已经有了很好的编程模型，但是仍旧没有摆脱重绘的模式。想我曾经自绘了一个仿 IE7 的标签页控件，那叫一个累。不过如今好了，WPF 中的控件是以树形的层次结构叠加而成的，这样使得修改控件的外观变得很方便。比如之前在《XAML 揭秘》中的那个按钮示例，它的中心不是文本，而是一个黑色的方块，这正是因为，组成按钮的 TextBlock 被替换为 Rectangle。使用如下 XAML 代码配合上面的程序代码即可看出。

```
<Window x:Class="AboutDialog"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="关于" SizeToContent="WidthAndHeight"
        Background="OrangeRed" Height="200">
  <StackPanel>
    <Button Content="OK" />
    <Button>
      <Button.Content>
        <Rectangle Height="40" Width="40" Fill="Black" />
      </Button.Content>
    </Button>
  </StackPanel>
</Window>）
```

第一个 Button 输出的视觉树为：

```
System.Windows.Controls.Button: OK
     Microsoft.Windows.Themes.ButtonChrome
          System.Windows.Controls.ContentPresenter
               System.Windows.Controls.TextBlock
```

第二个 Button 输出的视觉树为：

```
     System.Windows.Controls.Button
        Microsoft.Windows.Themes.ButtonChrome
          System.Windows.Controls.ContentPresenter
               System.Windows.Shapes.Rectangle
```

很有趣吧！）

Chapter:依赖属性（Dependency Properties）1

WPF 引入了一种新的属性类型，称作“依赖属性”，它可以用在外观风格、自动化数据绑定以及动画等方面。我们在第一次遇到这个概念的时候，可能会有一些迷惑，因为它把拥有简单字段、属性、方法和事件的.NET 类型弄得有点复杂。但是，当我们理解了它可以用来解决什么样的问题时，就会非常喜欢它。

依赖属性依赖多个能够在任意时刻及时确定属性值的提供器（provider），这些提供器可以是一个不断改变属性值的动画，也可以是一个可以将属性值传递到子元素的父元素。它的最大的特点无疑是它能够提供变更通知（change notification）的能力。

为属性加入这种能力，可以使得我们在 XAML 中直接使用 WPF 的各种丰富功能。WPF“说明性友好”设计的关键就是为了支持对属性的大量使用（这就是说，XAML 就像 XML，可以为元素设置各种各样的属性，“说明性友好”的目的就是追求有效地用利属性），比如 Button 就拥有 96 个公共属性！XAML 可以简单地设置属性而无需任何程序代码，但是如果在依赖属性中没有额外的引擎，那么即使在没有编写代码的情况下想要获得属性值都是困难的。

理解依赖属性的细节通常只对自定义控件开发者来说比较重要。有时候，我们可能仅需要知道风格类和动画类的依赖属性。在开发一段时间 WPF 应用后，我们可能发现，我们甚至希望所有的属性都是依赖属性！

依赖的属性的实现

实际上，依赖属性就是连接了一些 WPF 基础结构的一般.NET 属性。它全部通过 WPF API 完成，对任何.NET 语言（除了 XAML）来说是完全透明的。

例：标准依赖属性的实现（Button 的 IsDefault）

```
public class Button : ButtonBase
{
    //  依赖属性
    public static readonly DependencyProperty IsDefaultProperty;

    static Button()
    {
        //  注册属性
        Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
            typeof(bool), typeof(Button),
            new FrameworkPropertyMetadata(false,
                new PropertyChangedCallback(OnIsDefaultChange)));
        //  ……
    }

    //  .NET 属性包装器（可选的）
    public bool IsDefault
    {
        get { return (bool)GetValue(Button.IsDefaultProperty); }
        set { SetValue(Button.IsDefaultProperty, value); }
    }
```

```
        //  属性变更回调（可选的）
        private static void OnIsDefaultChanged(
            DependencyObject o, DependencyPropertyChangedEventArgs e) { }

        //  ……
    }
```

静态的 IsDefaultProperty 字段是真正的依赖属性，它是 System.Windows.DependencyProperty 类型的。所有的依赖属性习惯上以 Property 为后缀，且其访问权限是 pulbic 和 static。依赖属性一般由 DependencyProperty.Register 方法创建，它需要一个名称（IsDefault）、一个属性类型（bool）和声明拥有这个属性的类型（Button）。通过该方法的不同重载，我们可以有选择地传递处理自定义属性元数据，也可以编写那些用于处理属性值变更、强制转换（coerce）和验证（validate）的回调方法。Button 在其静态构造方法中调用了 Register 方法的一种重载，并向其提供了依赖属性的默认值（false），并且为处理变更通知（change notification）附加了一个委托。

IsDefault 属性最终通过调用继承自依赖属性基类 System.Windows.DependencyObject 的 GetValue 和 SetValue 方法实现了对 IsDefaultProperty 的访问器。GetValue 返回由 SetValue 最后一次设置的值，或是当 SetValue 从未被调用时，返回属性注册时提供的默认值。IsDefault 属性有时被称作“属性包装器”（property wrapper），它不是必须的，如 Button 的使用者可以直接调用公共方法 GetValue 和 SetValue 来获取以来属性的值。尽管如此，定义一个.NET 属性仍旧十分有用，因为它不但令我们以编程方式读/写字段更加自然，而且还令 XAML 可以直接设置属性。（意思是，如果在元素对象的类型中定义了.NET 属性，则我们在 XAML 中操作元素的属性，就如同在元素对象上操作那个相应的.NET 属性；相反地，如果我们没有定义.NET 属性，那么在 XAML 中也无法简单通过操作元素的属性来调整元素对象的状态，因此最好为依赖属性定义对应的属性包装器。）

在 XAML 在使用依赖属性的时候需要注意，尽管 XAML 编译器在编译时需要属性包装器，但是在运行时，WPF 并不使用属性包装器，而是直接调用底层 GetValue 和 SetValue 方法。因此，在依赖属性对应的包装器中，除了调用 GetValue 和 SetValue 之外，不要包含任何逻辑。所有的 WPF 内建属性包装器都遵循这条规则，因此在我们编写新的包含依赖属性的类型时，也应当这么做。

像上例那样表示一个简单布尔值的属性，表面上看起来有些啰嗦，但由于 GetValue 和 SetValue 内部使用了一个高效的存储结构（意思大概是说，依赖属性通过 Register 方法注册到一个数据结构中，这个结构的存取效率很好，因此通过那两个方法操作依赖属性，相应的开销也会很小），且 IsDefaultProperty 是静态字段，这使得依赖属性的实现相对于一般的.NET 属性而言，由于不用为每个实例都分配内存，从而节省内存的开销。

依赖属性带来的好处不仅仅在内存使用方面，它还集中化并且标准化了许多属性的实现代码，这些属性实现代码有用来检查线程访问的，也有用来引发重绘所含元素的。例如，在元素属性值变更的时候（如 Button 的 Background 属性），元素需要被重绘，这可以通过传递 FrameworkPropertyMatadataOptions.AffectsRender 标志到重载的 DependencyProperty.Register 方法来实现。此外，依赖属性还允许三种重要的特性：变更通知、属性值继承以及对多种提供器的支持。

2 变更通知

在依赖属性值发生变化的时候，WPF 能够自动触发一些依赖这个属性元数据的动作，这些动作可以是重绘相应的元素、更新当前布局以及刷新数据绑定等。由变更通知实现的一个有趣的特性被称作“属性触发器”（property triggers），它允许我们不编写程序代码就可以在属性值发生改变的时候执行自定义动作。例如，我们把鼠标悬停在 About Dialog（参看之前发布的文章）的按钮上时，按钮的文本立刻变成蓝色；移开鼠标，按钮的文本又恢复成黑色。为了实

现这样的想法，我们在没有属性触发器的情况下，可以为每一个按钮附加事件处理器。鼠标悬停的事件是 MouseEnter，鼠标移开的事件是 MouseLeave。

```
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
          MinWidth="75" Margin="10">Help</Button>
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
          MinWidth="75" Margin="10">OK</Button>
```

对应的两个事件处理函数为：

```
//  当鼠标悬停在按钮上时，将按钮的背景颜色改为蓝色
private void Button_MouseEnter(object sender, MouseEventArgs e)
{
     Button b = sender as Button;
     if (b != null) b.Foreground = Brushes.Blue;
}

//  当鼠标离开按钮时，恢复按钮的背景颜色
private void Button_MouseLeave(object sender, MouseEventArgs e)
{
     Button b = sender as Button;
     if (b != null) b.Foreground = Brushes.Black;
}
```

如果使用属性触发器，我们可以大幅简化上面的实现，仅通过 XAML 就可以完成相同的任务：

```
 <Trigger Property="IsMouseOver" Value="True">
<Setter Property="Foreground" Value="Blue" />
 </Trigger>
```

触发器可以作用在按钮的 IsMouseOver 属性上，它的值在 MouseEnter 事件被触发的时候变为 true，在 MouseLeave 被触发的时候变为 false。值得注意的是，我们并不需要在 IsMouseOver 变为 false 时将 Foreground 恢复成黑色，这个工作完全由 WPF 自动完成。

触发器需要附加在每一个按钮上，但不幸的是，由于 WPF 3.0 中的人为限制，我们不能像在 Button 这样的元素上直接应用触发器，而只能将其应用在其 Style 对象内部，如下：

```
<Button MinWidth="75" Margin="10" Content="Help">
    <Button.Style>
        <Style TargetType="{x:Type Button}">
            <Style.Triggers>
                <Trigger Property="IsMouseOver" Value="True">
                    <Setter Property="Foreground" Value="Blue" />
                </Trigger>
            </Style.Triggers>
        </Style>
    </Button.Style>
```

```
        </Button>
```

属性触发器只是 WPF 支持的三类触发器中的一类，其它两类是数据触发器和事件触发器，将会在以后介绍。此外，尽管 FrameworkElement 的 Triggers 属性是一个 TriggerBase 类型（三类触发器的基类）的读/写集合，但其在 WPF 3.0 中只能包含事件触发器，这是由于 WPF 团队来不及完全实现造成的，因此，如果向该集合添加属性触发器或数据触发器，将导致运行时异常。（尚未在 VS2008 中试验这么做是否会出现异常）

（如果我为 Button 同时设置 MouseEnter 事件处理器和 IsMouseOver 的触发器会是什么结果呢？对于上例，修改 XAML 代码如下：

```
        <Button MinWidth="75" Margin="10" Content="Help" MouseEnter="Button_MouseEnter">
            <Button.Style>
                <Style TargetType="{x:Type Button}">
                    <Style.Triggers>
                        <Trigger Property="IsMouseOver" Value="True">
                            <Setter Property="Foreground" Value="Blue" />
                        </Trigger>
                    </Style.Triggers>
                </Style>
            </Button.Style>
    </Button>
```

并为其添加事件处理方法：

```
        private void Button_MouseEnter(object sender, MouseEventArgs e)
        {
        System.Threading.Thread.Sleep(1000);
    }
```

运行后将鼠标放在 Button 上面，可以发现按钮的文本颜色在 1 秒钟后变成蓝色。这说明按钮的事件处理方法先于风格触发器执行。同理，修改上面的事件处理方法如下：

```
        private void Button_MouseEnter(object sender, MouseEventArgs e)
        {
        Button b = sender as Button;
        if (b != null) b.Foreground = Brushes.Red;
        }
```

运行后将鼠标放在按钮上面，可以看到其文本颜色变为红色，且当鼠标从按钮上移开，文本并不变色。如果想要解释这种想象的产生原因，需要了解后面的一些知识。）

3

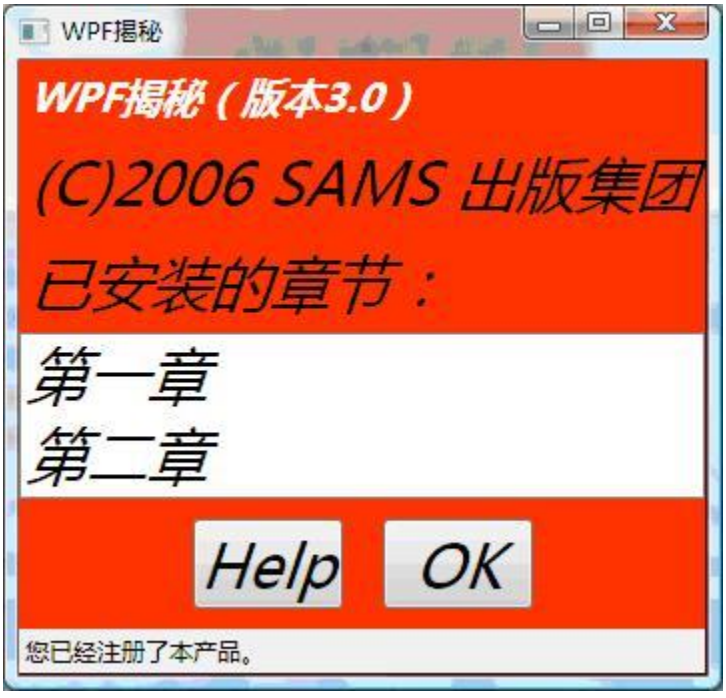
属性值的继承

属性值继承（简称属性继承）与传统的面向对象继承不同，它是指属性值可以沿着元素树向下传递的过程。

例：在 Window 元素上设置属性

```
<Window x:Class="Test.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF 揭秘" SizeToContent="WidthAndHeight"
    FontSize="30" FontStyle="Italic" Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 揭秘（版本 3.0）
        </Label>
        <Label>(C)2006 SAMS 出版集团</Label>
        <Label>已安装的章节：</Label>
        <ListBox>
            <ListBoxItem>第一章</ListBoxItem>
            <ListBoxItem>第二章</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>您已经注册了本产品。</StatusBar>
    </StackPanel>
</Window>
```

下图展示了以显式方式设置 Window 元素的 FontSize 和 FontStyle 依赖属性后，整个窗体的变化情况。



为这两个属性设置的值将会沿着元素树向下传递，并被相应的子元素继承（即将子元素的对应属性也被设定为这个值）。上例中，Button、Label 和 ListBoxItem 都受到了影响，但由于第一个 Label 显式设置了 FontSize，从而其字

体大小未受影响。值得注意的是，StatusBar 中的文本并没有受到这两个值的影响，尽管它与其它控件相同，也包含这两个属性。由于以下两种原因，属性值的继承显得有些微妙：

（1）并不是所有的依赖属性都参与属性值的继承（依赖属性可以通过向 DependencyProperty.Register 传递 FrameworkPropertyMatadataOptions.Inherits 来选择是否参与继承）

（2）可能存在更高优先级的属性设定源（稍后解释）

StatusBar 显示的结果由第二种原因导致的。一些如 StatusBar、Menu 和 Tooltip 控件的内部将它们的字体属性设定为匹配当前系统的设置。这种结果有些令人迷惑，因为这样的控件阻止了继承属性值沿着元素树继续传递。例如，当我们在 StatusBar 中加入一个 Button 作为其逻辑子元素，那么 Button 的 FontSize 和 FontStyle 都将保持默认值，这与处于 StatusBar 之外的那些 Button 不同。

属性值继承本来是用来操作元素树的，但是它也可以用于在其它情境。例如，属性值可以传递到某个 XML 意义上的子元素，而这个子元素并非逻辑树或视觉树的子元素。这些伪子元素可以是某个元素的触发器，也可是任意属性值，只要它是一个从 Freezable 派生的对象就可以。（暂不明）

4

对多种提供器的支持

WPF 包含许多强大的机制，这些机制尝试独立地设置依赖属性值。WPF 需要五个步骤来计算依赖属性的最终值：确定基本值—>计算表达式（如果有）—>应用动画—>强制—>验证。

第一步　　确定基本值

基本值的计算被分解成许多属性值的提供器，下列提供器可以设置大部分依赖属性，按从高到低的优先级排列：

（1）局部值（local value）提供器；
（2）样式触发（style trigger）提供器；
（3）模板触发（template trigger）提供器；
（4）样式设定（style setter）提供器；
（5）主题样式触发（theme style trigger）提供器；
（6）主题样式设定（theme style setter）提供器；
（7）属性值继承（property value inheritance）提供器；
（8）默认值（default value）提供器

我们已经见过一些属性值提供器了，比如属性值继承提供器。从技术上说，局部值提供器是任何调用了 DependencyObject.SetValue 的对象，但是它通常可以被看作一种在 XAML 或程序代码中给普通属性赋值的操作。默认值提供器设定的值是依赖属性在注册时设定的那个值，因此其优先级最低。（详情参见依赖属性（1）对依赖属性实现的介绍）

这个优先级顺序解释了为何 StatusBar 的 FontSize 和 FontStyle 属性没有受属性值继承影响，这是因为 StatusBar 已经被主题样式提供器（第 6 级）设定为匹配系统的值。尽管它的优先级超过了属性值继承提供器（第 7 级），但是我们仍旧可以通过较高优先级的提供器来设定字体属性值。

第二步　　计算表达式

如果来自第一步的值是一个表达式（一种派生自 System.Windows.Expression 的对象），则 WPF 负责执行一个将表达式转换为具体值的特殊计算步骤。在 WPF 3.0 中，表达式仅可是动态资源或数据绑定，未来版本的 WPF 可能会允许其它种类的表达式。

第三步　　应用动画

在一个或多个动画正在运行的时候，它们拥有更改当前属性值（来自第二步的值）的权利。因此，动画比任何其它属性值提供器的优先级都高，这对 WPF 初学者来说有些难度。

第四步　　强制（Coerce）

在所有的属性值提供器完成设定后，WPF 将结果属性值传递到为依赖属性注册的 CoerceValueCallback 委托中。我们可以在这个委托内编写自己的代码，用以重新计算依赖属性值或执行强制转换。例如，WPF 内建控件 ProgressBar 使用这个回调来约束其 Value 依赖属性。这个属性的值应在 Minimum 和 Maximum 之间，但如果输入值小于 Minimum，则其返回 Minimum 值，如果输入值大于 Maximum，则其返回 Maximum 值。

（这个 CoerceValueCallback 在 MSDN 中的说法是“为只要重新计算依赖项属性值或专门请求强制转换时就调用的方法提供一个模板，此回调的实现应检查 baseValue 中的值，并根据该值或其类型确定该值是否需要进一步进行强制转换”。按我自己的理解，这个委托的作用就是为上一步计算好的属性值添加使其合理化的逻辑。

MSDN 为这个回调提供了一个示例（其中 d 是该属性所在的对象，value 是该属性在尝试执行任何强制转换之前的新值）：

```
private static object CoerceButtonColor(DependencyObject d, object value)
{
     ShirtTypes newShirtType = (d as Shirt).ShirtType;
     if (newShirtType == ShirtTypes.Dress || newShirtType == ShirtTypes.Bowling)
     {
         return ButtonColors.Black;
     }
     return ButtonColors.None;
}
```

从中可以看出，依赖属性 ButtonColor 的值依赖于对象 d 的 ShirtType，但并没有使用上一步计算好的值 value。）

第五步　　验证

最终，经过强制转换的值被传递到为依赖属性注册的 ValidateValueCallback 委托中。如果输入值是合法的，则此委托必须返回 true；否则，必须返回 false。返回 false 将导致一个取消整个过程的异常。

（MSDN 中的示例：

```
private static bool ShirtValidateCallback(object value)
{
ShirtTypes sh = (ShirtTypes)value;
return (sh == ShirtTypes.None || sh == ShirtTypes.Bowling || sh == ShirtTypes.Dress |
| sh == ShirtTypes.Rugby || sh == ShirtTypes.Tee);
}
```

有此可见，在这个回调的实现中，应当加入验证属性值的范围的逻辑。同理，在 ProgressBar 为其 double 类型的 ValueProperty 依赖属性实现的验证回调中，判断了属性值是否是合法的 double 数据。）

如果我们想要查看依赖属性值是由那一个提供器设置的，可以使用静态的 DependencyPropertyHelper.GetValue Source 方法来调试。该方法返回一个 ValueSoruce 结构，它包含一个暴露基本值来自于何方的 BaseValueSource 枚举，以及对应 2~4 步的 IsExpression、IsAnimated、IsCoerced 属性。当我们在前面的 StatusBar 实例上为 FontSize 和 FontStyle 调用这个方法后，BaseValueSource 的返回值为 DefaultStyle，这说明它的值来自于主题风格设定提供器（主题风格有时也称作默认风格，对应主题风格触发器就是 DefaultStyleTrigger）。此外切记不要在产品代码中使用这个方法。

在前面通过事件处理器设置按钮文本颜色的示例中，有一个问题值得注意：在 MouseLeave 事件处理器中，按钮的 Foreground 被赋予了局部值 Brushes.Black（局部值提供器）。这与按钮的初始状态不同，其初始状态是由主题风格设定提供器设置的。如果主题变换了，且新主题（或其它优先级较高的提供器）尝试改变 Foreground 的默认值，那么局部值提供器会胜出（因为它的优先级高于主题风格提供器），并将 Foreground 设置为黑色。对于这种情况，我们可能会希望清除这个局部值，以便可以使用主题的默认设定。DependencyObject 提供了一个 ClearValue 方法，可以清除局部值：

```
b.ClearValue(Button.ForegroundProperty);
```

值得注意的是，在 IsMouseOver 上的触发器与此不同，当其处于“未触发”状态时，将会忽略对属性值的计算。

5

附加属性

附加属性一种特殊的依赖属性形式，它可以被附加到任意对象上。

对于之前的 About Dialog 示例，如果我们不想让整个 Window 元素及其子元素都被 FontSize 和 FontStyle 影响，而是希望改变它们仅影响位于第二个 StackPanel 中的 OK 和 Help 两个按钮。我们很自然地会想到把它们从 Window 元素中移动到 StackPanel 元素中，但这样做是不可以的，因为 StackPanel 本身并不包含任何与字体相关的属性。此时，我们可以使用定义在 TextElement 类中的附加属性 FontSize 和 FontStyle 来完成相同的任务：

```
<Window x:Class="Test.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="WPF 揭秘" SizeToContent="WidthAndHeight"
    Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 揭秘（版本 3.0）
        </Label>
        <Label>(C)2006 SAMS 出版集团</Label>
        <Label>已安装的章节：</Label>
        <ListBox>
            <ListBoxItem>第一章</ListBoxItem>
            <ListBoxItem>第二章</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center"
                    TextElement.FontSize="30" TextElement.FontStyle="Italic">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>
            <Button>您已经注册了本产品。</Button>
        </StatusBar>
    </StackPanel>
</Window>
```

TextElement.FontSize 和 TextElement.FontStyle 必须用在 StackPanel 元素上，因为它不包含这两个属性。当 XAML 解析器/编译器遇到这个语法时，需要 TextElement（有时被称作“附加属性提供器”）提供静态的 SetFontSize 和 SetFontStyle 方法。因此，与上例中第二个 StackPanel 等价的 C#代码为：

```
StackPanel panel = new StackPanel();
TextElement.SetFontSize(panel, 30);
TextElement.SetFontStyle(panel, FontStyles.Italic);
panel.Orientation = Orientation.Horizontal;
```

```
panel.HorizontalAlignment = HorizontalAlignment.Center;
Button helpButton = new Button();
helpButton.MinWidth = 75;
helpButton.Margin = new Thickness(10);
helpButton.Content = "Help";
Button okButton = new Button();
okButton.MinWidth = 75;
okButton.Margin = new Thickness(10);
okButton.Content = "OK";
panel.Children.Add(helpButton);
panel.Children.Add(okButton);
```

枚举值如 FontStyles.Italic、Orientation.Horizontal 和 HorizontalAlignment.Center 可以在 XAML 中被分别简单表示为 Italic、Horizontal 和 Center，这要感谢.NET 中的 EnumConverter 类型转换器，它可以将任何大小写不敏感的字符串转换为枚举值。

在内部，SetFontSize 方法调用了 DependencyObject.SetValue 方法，后者一般由依赖属性的访问器（accessor）（就是之前说过的按钮的 IsDefaultProperty 的访问器 IsDefault）调用，但此时却是由当前传入的 DependencyObject 的实例进行调用：

```
public static void SetFontSize(DependencyObject element, double value)
{
element.SetValue(TextElement.FontSizeProperty, value);
}
```

相似地，附加属性也定义了调用 DependencyObject.GetValue 的 GetXXX 方法：

```
public static double GetFontSize(DependencyObject element)
{
return (double)element.GetValue(TextElement.FontSizeProperty);
}
```

附加属性被广泛使用在 WPF 的布局系统中，相关内容放到以后介绍。

Button 等控件继承的 FontSize 和 FontStyle 附加属性并不是由它们自己定义的，而是定义在其基类 Control 中，这十分令人迷惑。实际上，它们真正被定义在看起来与控件并不相关的 TextElement 类当中。在 TextElement 类中，包含下面的注册代码：

```
TextElement.FontSizePropert = DependencyProperty.RegisterAttached(
"FontSize", typeof(double), typeof(TextElement),
new FrameworPropertyMetadata(
    SystemFonts.MessageFontSize,
    FrameworkPropertyMetadataOptions.Inherits |
    FrameworkPropertyMetadataOptions.AffectsRender,
    FrameworkPropertyMetadataOptions.AffectsMeasure),
new ValidateValueCallback(TextElement.IsValidFontSize));
```

与前面讲到的 Button.IsDefaultProperty 的 Register 类似，只不过 RegisterAttached 为附加属性优化了元数据的处理过程。

控件并不注册它们，而是通过调用 AddOwner 来添加一个已注册了的依赖属性：

```
Control.FontSizeProperty = TextElement.FontSizeProperty.AddOwner(
typeof(Control), new FrameworkPropertyMetadata(SystemFonts.MessageFontSize,
FrameworkPropertyMetadata.Inherits));
```

因此，由控件继承所公开的 FontSize、FontStyle 以及其它字体相关的依赖属性都与 TextElement 公开的属性完全一样。所幸在多数类中，公开了附加属性的类（比如包含 GetXXX 和 SetXXX）都与定义了一般依赖属性的类相同，这使得许多混乱得以避免。

与 Windows Form 类似，WPF 中的许多类也定义了一个 System.Object 类型的 Tag 属性，用以存储任意与当前实例相关的数据。与 Tag 属性相比，附加属性是为派生自 DependencyObject 的对象附加自定义数据的一种更加强大、更灵活的机制。附加属性允许为密封类（sealed）的实例添加自定义数据，WPF 中有许多这样的做法，常常被忽略。

尽管在 XAML 中设置附加属性时需要依靠静态的 SetXXX 方法，但是我们在程序代码中可以直接通过 DependencyObject.SetValue 方法完成。这意味着我们可以在程序代码中将任何依赖属性作为附加属性。例如，下例将 ListBox 的 IsTextSearchEnabled 属性附加给了 Button，并为其设置了一个值：

```
okButton.SetValue(ListBox.IsTextSearchEnabledProperty, true);
```

尽管这么做没什么意义，而且它确实对 Button 不起作用，但是我们仍旧可以一种对程序和组件有意义的方式来自由地使用它。

下例展示了一种有趣的元素扩展方式：FrameworkElement 的 Tag 属性是一个依赖属性，因此我们可以把它附加到 GeometryModel3D 的实力上。

```
GeometryModel3D model = new GeometryModel3D();
model.SetValue(FrameworkElement.TagProperty, "自定义数据");
```

这只不过是一种 WPF 提供灵活性的方式，这种方式不需要继承就可以获得新的属性。

6

至此，有关依赖属性的话题就告一段落了，让我们回顾一下有关依赖属性的一些知识。下面提到的许多例子都可以在前面的文章中找到，这里就不在赘述了。

上图画的有点乱，不过我们还是可以从其中看出一些门道：

（1）依赖属性（DependencyProperty）值需要使用依赖对象（DependencyObject）的 GetValue 和 SetValue 来读/写；

（2）WPF 中有众多类派生自 DependencyObject，因此它们都会继承 GetValue 和 SetValue 方法；

（3）从 DependencyObject 派生的类可以定义习惯上是 public static readonly 的 DependencyProperty 对象，并且在其静态构造方法中使用 DependencyProperty.Register 方法向属性系统注册依赖属性；有些依赖属性对于 C#代码来说是多于的，因为总可以使用别的方式来获得同样的属性值，但是它们对于 XAML 来说却是完全必要的；

（4）依赖属性总是伴随一些元数据，这些元数据是该属性在属性系统内的特征。属性系统内部会有某些机制来维护它们，但我们也可以通过某些方法将对应的元数据传入，如 FrameworkPropertyMetadata；

（5）当依赖属性值发生变化的时候，通常会调用 FrameworkPropertyMetadata.PropertyChangedCallback 来处理，处理过程引发相应的事件处理器，程序代码可以通过实现事件处理器来处理变化；或是改变某些依赖属性的值，使得 XAML 可以利用这些它们来实现特殊的效果（如 IsMouseOver 的触发器）；

（6）有些依赖属性的值可以被子元素继承，通常会给依赖属性设置指定了 FrameworkPropertyMetadataOptions.Inherits 的元数据（如看到 Inherits 标志，就使用 LogicalTreeHelper 遍历当前元素节点及其子结点，依次将子元素对象的对应依赖属性设置为与当前元素对象的相同值）；

（7）为了确定依赖属性的值，共有 8 种优先级不同的提供器，属性值继承提供器只是其中一种，因此某些元素的属性没有获得继承值的原因，通常是被比属性值继承提供器更优先的提供器设定过了。依赖属性值的确定过程很大程度上依靠提供器的优先顺序，需要格外注意，否则会出现明明设置了值，却不按预想显示的问题（如之前的 `StatusBar`）。整个这个过程共分 5 步：获得基本值、计算表达式、处理动画、强制和验证。

（8）附加属性有两种用法，

ⅰ）**伪继承**

某个类（通常从 DependencyObject 派生，如 TextElement）定义了一个依赖属性，并且通过 RegisterAttached 方法进行了注册；其它类（如 Control）也定义了一个依赖属性（通常是同名的），并且使用 AddOwner 方法为其添加了所有者。这样就仿佛 Control 是从 TextElement 派生而来的一样，因为它也包含了 TextElement 中的依赖属性，并且这个依赖属性一旦被“伪基类”TextElement 改变，“伪子类”Control 中的副本也会跟着变。不过既然称作“伪”，那么实际上 Control 和 TextElement 就没有继承关系，而仅是通过附加属性联系在一起。

这种用法很有意义，如上图所示，TextElement 定义的附加属性在 Control 中都有与之相对的，且在 Control 的静态构造方法中使用 AddOwner 获得了它的引用。从 Control 派生的所有子类中都继承了由 Control 定义的那个对应的依赖属性。如前例，在 StackPanel 上设置 TextElement.FontSize 会导致其值被改变。由于 StackPanel 的所有子元素继承这个属性值，因此 Control 中 FontSize 的副本会跟着改变，继而改变 Control 子类 Button 等的字体大小的呈现。

ⅱ）**直接附加**

由于 DependencyObject 的 SetValue 和 GetValue 都需要 DependencyProperty 类型的参数，因此可以直接在某 DependencyObject 的对象身上调用这两个方法，并传入任何 DependencyProperty 的对象。

这种用法就是为了给 DependencyObject 的实例附加任意的依赖属性，只是如果附加的属性对实例来说有意义，那就有意义（如之前文章中给 GeomertyModel3D 的实例附加 FrameworkElement.TagProperty 属性），否则就是没意义。

（9）书里提到过依赖属性具有高效的存储结构，不过至于这是种怎样的结构，就不得而知了。尽管可以利用反射器一点点分析出来，不过我想似乎也没有这样的必要。

【以下内容全部都是我的猜想，没有经过验证！】

经过试验，我发现有三种情况都会使得 DependencyProperty 将对应的依赖属性存入那个结构，权且假设它就是个哈希表吧。这三种情况是：

ⅰ）使用 DependencyProperty.Register 及 RegisterAttached 等方法注册依赖属性；

ⅱ）使用 DependencyProperty.AddOwner 方法添加所有者；

ⅲ）使用 DepenedencyObject.SetValue 设定依赖属性值。

第一种显而易见，注册属性的时候需要指定所有者，这样就可以建立如下关系：

Key	Value
所有者	注册的依赖属性

第二种仔细想想也容易看出来：

Key		Value
注册时的所有者	注册的依赖属性	（这是通过 Register 完成的映射）
添加的所有者	同上	（这是由 AddOwner 添加的映射）

当依赖属性发生变化时，属性系统会根据这些映射给每个所有者发送变更通知，如前所述。

第三种需要做个简单的试验，通过定义两个不同的 DependencyObject 来获得：

```
GeometryModel3D model1 = new GeometryModel3D(),
model2 = new GeometryModel3D();
model1.SetValue(FrameworkElement.TagProperty, "model1");
model2.SetValue(FrameworkElement.TagProperty, "model2");
Debug.WirteLine(model.GetValue(FrameworkElement.TagProperty) as String);
Debug.WirteLine(model.GetValue(FrameworkElement.TagProperty) as String);
```

控制台分别输出字符串 model1 和 model2。由此可以看出来，那个结构存储了两个对象与同一个属性的映射，比如

Key	Value
model1	FrameworkElement.TagProperty 的副本 1
model2	FrameworkElement.TagProperty 的副本 2

这样只要在 model1 上调用 GetValue，那么必然获得附加在 model1 身上的依赖属性值，并且不会因其与 model2 附加的是相同的属性，就会获得 model2 的属性值。

一言以蔽之，WPF 中的依赖属性非常多，它们在对 XAML 来说至关重要。

7.

如何定义就不具体说了。主要说说

1. 元数据 (MetaData)
2. DependencyProperty value 设置步骤

1. 元数据：

依赖项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项属性的特征。当属性系统处理任何给定的依赖项属性时，也会经常访问这些元数据。

依赖属性的缺省值定义，回调等都需要在注册时定义的元数据。看看元数据的构造函数(最全的那个)

```
public PropertyMetadata(
    Object defaultValue,
    PropertyChangedCallback propertyChangedCallback,
    CoerceValueCallback coerceValueCallback
)
```

defaultValue	依赖属性的缺省值
PropertyChangedCallback	当属性有效值变化时，属性系统会调用该 delegrate
coerceValueCallback	当属性系统对调用该依赖属性的 CoerceValue 时，都会调用该 delegrate。在调用时，你可以强制返回另外一个值

2. 设置步骤

下图说明了 WPF 在处理 DependencyProperty 时的 5 个步骤：

Step 1: 确定 Base Value

Base value 有一个设置的顺序，下面表示设置的优先级 高-->低：

1. Local Value
2. Style triggers
3. Template triggers
4. Style Setters
5. Theme style triggers
6. Theme style setters
7. Property value inheritance
8. Default value.

Step 2: Evaluate

如果 Step1 设置的 Value 是个表达式(Value 是 System.Windows.Expression)，WPF 需要先计算该表达式。

Step 3: Apply Animations

如果该 DependencyProperty 被 animation 影响，此 Animation 产生的 Value 会冲掉它本身的 value，其实这就是 WPF 中动画最优先原理。

Step 4: Coerce

DependencyProperty 的最后一次机会去改变 value。使得设置的 value 可以满足逻辑。

Step 5: Validate

在注册时可以选择是否进行验证：

```
public sealed class DependencyProperty
{
public static DependencyProperty Register(string name, Type propertyType, Type ownerType, PropertyMetadat
a typeMetadata, ValidateValueCallback validateValueCallback);
}
```

8.

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。例如：

```
<DockPanel>
  <CheckBox DockPanel.Dock="Top">Hello</CheckBox>
</DockPanel>
```

Dock 不是 CheckBox 的属性，而是定义在 DockPanel 中的。

用代码使用：

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

如何创建附加属性

1. 声明一个 DependencyProperty 类型的 public static readonly 字段，将附加属性定义为一个依赖项属性。

2. 使用 **RegisterAttached** 方法的返回值来定义此字段。例如：

```
    public class OwerClass：DependencyObject
    {

        public static string GetAttachedPropertyName(DependencyObject obj)
        {
            return (string)obj.GetValue(AttachedPropertyNameProperty);
        }

        public static void SetAttachedPropertyName(DependencyObject obj, string value)
        {
            obj.SetValue(AttachedPropertyNameProperty, value);
```

```
    }

    public static readonly DependencyProperty AttachedPropertyNameProperty =
        DependencyProperty.RegisterAttached("AttachedPropertyName", typeof(string), typeof(OwerClass), new UIPropertyMetadata(0));
}
```

小提示：
可以利用 VS2008 智能提示：在 class 里面输入 propa，然后按 Tab 自动生成基本内容:)

9.

WPF 引入了一种新的属性：Dependency 属性。Dependency 属性的应用贯串在整个 WPF 当中。Dependency 属性根据多个提供对象来决定它的值。并且是及时更新的。提供对象可以是动画，不断地改变它的值。也可以是父元素，它的属性值被继承到子元素。毫无疑问，Dependency 属性最大的特点就是内建的变化通知功能。提供 Dependency 属性功能主要是为了直接从声明标记提供丰富的功能。WPF 声明的友好设计的关键是大量的使用属性。如果没有 Dependency 属性，我们将不得不编写大量的代码。关于 WPF 的 Dependency 属性，我们将重点研究如下三个方面：

1、变化通知功能：属性的值被改变后，通知界面进行更新。

2、属性值的继承功能：子元素将继承父元素中对应属性名的值。

3、支持多个提供对象：我们可以通过多种方式来设置 Dependency 属性的值。

在上面的实现代码中，System.Windows.DependencyProperty 类表示的静态字段 IsDefaultProperty 才是真正的 Dependency 属性。为了方便，所有的 Dependency 属性都是公有、静态的，并且还有属性后缀。通常创建 Dependency 属性可用静态方法 DependencyProperty.Register。参数的属性名称、类型、使用这个属性的类。并且可以根据重载的方法提供其他的通知事件处理和默认值等等。这些相关的信息可参考 FrameworkPropertyMetadata 类的多个重载构造函数。

最后，实现了一个.NET 属性，其中调用了从 System.Windows.DependencyObject 继承的 GetValue、SetValue 方法。所有具有 Dependency 属性的类都肯定会继承这个类。GetValue 方法返回最后一次设置的属性值，如果还没有调用一次 SetValue，返回的将是 Register 方法所注册的默认值。而且，这种.NET 样式的属性封装是可选的，因为 GetValue/SetValue 方法本身是公有的。我们可以直接调用这两个函数，但是这样的封装使代码更可读。

虽然这种实现方式比较麻烦，但是，由于 GetValue/SetValue 方法使用了一种高效的小型存储系统，以及真正的 Dependency 属性是静态字段（不是实例字段），Dependency 属性的这种实现可以大大的减少每个属性实例的存储空间。想象一下，如果 Button 有 50 个属性，并且全部是非静态的实例字段，那么每个 Button 实例都含有这

样 50 个属性的空间，这就存在很大的空间浪费。除了节省空间，Dependency 属性的实现还集中、并且标准化了属性的线程访问检查、提示元素重新提交等等。

10.

现在我们给上面的 Window 添加一个状态栏。XAML 代码如下：

```
<Window ......>
    <StackPanel>
        <Label>LabelText</Lable>
        <StatusBar>This is a Statusbar</StatusBar>
    </StackPanel>
</Window>
```

这时你会发现：虽然 StatusBar 支持这个 FontSize 这个属性，它也是 Window 的子元素，但是它的字体大小却没有变化。为什么呢？因为并不是所有的元素都支持属性值继承。还存在如下两种例外的情况：

1、部分 Dependency 属性在用 Register 注册时可以指定 Inherits 为不可继承。

2、如果有其他更高优先级方法设置了其他的值。（关于优先级的介绍且看下面分解。）

部分控件如 StatusBar、Menu 和 Tooptip 内部设置它们的字体属性值以匹配当前系统的设置。这样用户通过控制面板可以修改它们的外观。这种方法存在一个问题：StatusBar 等截获了从父元素继承来的属性，并且不影响其子元素。比如，如果我们在 StatusBar 中添加了一个 Button。这个 Button 的字体属性会因为 StatusBar 的截断没没有改变，将保留其默认值。

附加说明：属性值继承的最初设计只适用于元素 Tree，现在已经进行多个方面的扩展。比如，值可以传递下级看起来像 Children，但在逻辑或者视觉 Tree 中并不是 Children 的某些元素。这些伪装的子元素可以是触发器、属性的值，只要它是从 Freezable 继承的对象。对于这些内容没有很好的文档说明。我们只需要能使用就行不必过多关心。

11.

【支持多个提供对象】

WPT 提供了独立的、强大的机制来设置 Dependency 属性的值。由于支持多个提供对象，如果没有很好的机制来处理这些提供对象，那么 Dependency 属性的值将是不可预测的、系统也将变得混乱。Dependency 属性的值取决于这些提供对象，它们以一定的顺序和优先级排列。

4、强制值

在处理完所有的提供对象后，WPF 将最终的属性值传递到 CoerceValueCallback 委派。如果 Dependency 属性在注册时提供了这样的委派，那么就应该根据自定义逻辑返回一个新的值。比如 ProgressBar，当所有提供对象最后所提供的值超出了其定义的最大、最小值范围时，ProgressBar 将利用这个 CoerceValueCallback 委派限制在这个范围之内。

5、值验证

最后，前缀的强制值将传递给 ValidateValueCallback 委派，如果 Dependency 属性注册了这个委派。当值有效时委派必须返回 True，否则返回 False。返回 False 将抛出异常，终止整个进程。

附加说明：如果我们不知道给定的 Dependency 属性的值来源于何处，可以调用静态的 DependencyPropertyHelper.GetValueSource 方法。它作为调试时的辅助工具，有时能给我们提供帮助。方法会返回一个 ValueSource 结构。ValueSource 结构中的属性成员 BaseValueSource、IsExpression、IsAnimated、IsCoerced 分别表示了前面列出的八个提供对象的相应类型。注意：请不要在最后的发布产品中使用这个方法，因为在将来版本的 WPF 中可能有不同的行为。只应该将其作为调试工具。

12.

从这里的代码可以看出，Attached 属性并不神秘。只是调用方法把元素和不相关的属性关联起来。而 SetFontSize 实现也比较简单。它只是调用了 Dependency 属性访问函数所调用的 DependencyObject.SetValue 方法。注意调用的对象是传入的 DependencyObject，而不是当前的实例：

```
public static void SetFontSize(DependencyObject element, double value)
{
    element.SetValue(TextElement.FontSizeProperty, value);
}
```

同样地，Attached 属性也定义了对应的 GetXXX 函数。它调用的 DependencyObject.GetValue 方法：

```
public static double GetFontSize(DependencyObject element)
{
    return (double)element.GetValue(TextElement.FontSizeProperty);
}
```

与普通的 Dependency 属性一样，这些 GetXXX 和 SetXXX 方法除了实现对 GetValue 和 SetValue 的调用，不能做任何其他额外的工作。

其实，在 WPF 应用中，Attached 属性更多的用来控制 UI 的布局。除了前面的 StackPanel，还有 Grid 等等。

补充说明：上面的代码还有一个问题需要说明。我们设置 StackPanel 的字体属性时用的是 TextElement 元素。为什么不用其他的元素 Control、Button 呢？

这个问题的关键之处在于 Dependency 属性的注册方法。我曾在 Dependency 属性[1]做过简单的说明。我们看看 Element 的 FontSizeProperty 属性的注册代码：

```
TextElement.FontSizeProperty = DependencyProperty.RegisterAttached(
    “FontSize”, typeof(double), typeof(TextElement), new FrameworkPropertyMetadata(
    SystemFonts.MessageFontSize, FrameworkPropertyMetadataOptions.Inherits |
    FrameworkPropertyMetadataOptions.AffectsRender |
    FrameworkPropertyMetadataOptions.AffectsMeasure),
    new ValidateValueCallback(TextElement.IsValidFontSize));
```

这里与我们前面的 IsDefault 属性类似，只是 RisterAttached 方法优化了 Attached 属性需要的属性元数据的处理过程。

另一方面，Control 的 FontSize 属性是在 TextElement 元素已经注册的属性之上调用 AddOwner 方法，获取一个完全相同的实例引用：

```
Control.FontSizeProperty = TextElement.FontSizeProperty.AddOwner(
    typeof(Control), new FrameworkPropertyMetadata(SystemFonts.MessageFontSize,
    FrameworkPropertyMetadataOptions.Inherits));
```

所以，在实现 Attached 属性时我们使用的是 TextElement，而不是 Control 等等。

13.

依赖项属性

Windows Presentation Foundation (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。

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。

下面汇集了在本软件开发工具包 (SDK) 文档中，在讨论依赖项属性时所使用的术语：

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。

CLR"包装"：属性的实际 get 和 set 实现。这些实现通过在 GetValue 和 SetValue 调用中使用依赖项属性标识符来合并此标识符，从而使用 WPF 属性系统为属性提供支持。

附加属性
附加属性是可扩展应用程序标记语言 (XAML) 定义的一个概念。
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。
一个主要方案是，使子元素可以将其在 UI 中的表示方式通知给父级；

在 Windows Presentation Foundation (WPF) 中，WPF 类型上存在的大多数附加属性都实现为依赖项属性。
附加属性是一个 XAML 概念，而依赖项属性则是一个 WPF 概念。
因为 WPF 附加属性是依赖项属性，所以它们支持依赖项属性概念，例如，属性元数据以及这些属性元数据中的默认值。

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，或者该值将会被其他对象使用。
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所属类型报告公用信息。
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。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一个依赖项属性的整体 WPF 模型，则需要从 DependencyObject 派生。

通过声明一个 DependencyProperty 类型的 public static readonly 字段将附加属性定义为一个依赖项属性。

通过使用 RegisterAttached 方法的返回值来定义此字段。

为了遵循命名标识字段及其所表示的属性的已建立 WPF 模式，字段名必须与附加属性名一致，并附加字符串 Property。

附加属性提供程序还必须提供静态的 Get 属性名 和 Set 属性名 方法作为附加属性访问器，否则会导致属性系统无法使用您的附加属性。

14.

依赖属性

什么时候需要定义依赖属性

1〉 如果希望属性可以在 Style 中设定
2〉 如果希望属性可以数据绑定
3〉 如果希望属性可以由动态资源(DynamicResource)设定
4〉 如果希望从父元素那里获得值
5〉 如果希望属性可以设置动画
6〉 如果希望属性在被修改时能够检测值得有效性，修改后回调
7〉 如果希望使用已有的依赖属性（Metadata override）

功能：

1〉 资源
2〉 数据邦定
3〉 风格
4〉 动画
5〉 Metadata 重载
6〉 属性值继承

依赖属性的实现

注册

如果希望属性成为依赖属性，你必须注册该属性到一个由属性系统维持的表中，并为其指定一个唯一标示。

示例：

```
public static readonly DependencyProperty AquariumGraphicProperty = DependencyProperty.Register(
 "AquariumGraphic",
 typeof(Uri),
 typeof(AquariumObject),
 new FrameworkPropertyMetadata(null,
     FrameworkPropertyMetadataOptions.AffectsRender,
     new PropertyChangedCallback(OnUriChanged)
 )
);
// Wrapper
public Uri AquariumGraphic
{
 get { return (Uri)GetValue(AquariumGraphicProperty); }
 set { SetValue(AquariumGraphicProperty, value); }
}
```

Wrapper 的 Identity 与 Register 的第一个参数最好一致。

Property Metadata

当注册依赖属性时，将通过属性系统创建一个 metadata object 来保存属性特征。可以在其中设置属性的缺省值，回调函数等。

有两种 Metadata

1> FrameworkPropertyMetadata

一般用在继承自 FrameworkElement 的类中，它提供比 PropertyMetadata 更多的选项来初始化属性，示例如上。

2> PropertyMetadata

一般初始化属性的缺省值和回调函数

示例如下：

```
public DateTime Time
 {
    get
    {
       return (DateTime)GetValue(TimeProperty);
    }
    private set
    {
       SetValue(TimeProperty, value);
    }
 }
 public static DependencyProperty TimeProperty = DependencyProperty.Register(
    "Time",
    typeof(DateTime),
    typeof(MyClock),
  new PropertyMetadata(DateTime.Now, new PropertyChangedCallback(OnDateTimeInvalidated)));

 private static void OnDateTimeInvalidated(DependencyObject d,
DependencyPropertyChangedEventArgs e)
 {
    MyClock clock = (MyClock)d;

    DateTime oldValue = (DateTime)e.OldValue;
    DateTime newValue = (DateTime)e.NewValue;

    clock.OnDateTimeChanged(oldValue, newValue);

 }
```

Chapter: Routed Event

路由事件（Routed Event）

就像 WPF 在.NET 属性之上添加了许多基础结构一样，它也在.NET 事件之上添加了许多基础结构。路由事件是一种与树上元素协同工作的事件，当它被触发后，能够沿着逻辑树和视觉树上/下传递，触发每个子元素的对应事件，不需要任何自定义的代码。

事件路由帮助大多数应用程序屏蔽视觉树的细节，是 WPF 中“元素组合”得以成功的关键。例如，Button 公开了处理低级 MouseLeftButtonDown 和 KeyDown 的 Click 事件。当用户在标准按钮上单击了鼠标左键，实际上他们是与 ButtonChrome 或 TextBlock 视觉子元素进行交互的。由于事件沿着**视觉树**向上传递，所以 Button 最终收到这个事件并且能够处理它。相似地，对于之前（在 XAML 揭秘中）录像机风格的 Stop 按钮，用户可能直接是在 Rectangle 逻辑子元素上单击了鼠标左键。由于事件沿着**逻辑树**向上传递，Button 也将会收到这个事件并且可以处理它。

因此，我们可以在像 Button 这样的元素内嵌入任意复杂的内容，或者赋予它任意复杂的视觉树。尽管如此，在任意内部元素上单击鼠标左键的行为都将导致 Click 事件被父元素 Button 触发。试想如果没有路由事件，那么实现这样的功能将不得不编写不少代码。

路由事件的行为和实现与依赖属性很相似，让我们先从路由事件的实现开始。

路由事件的实现

大多数情况下，路由事件与普通的.NET 事件看起来非常相似。与依赖属性相同，没有.NET 语言（除了 XAML）知道路由事件的存在，对其的支持则完全基于 WPF API。

下例展示了 Button 实现 Click 路由事件的方法，不过事实上 Click 事件是由 Button 的基类 ButtonBase 实现的，不过这并不影响我们的讨论。

```
public class Button : ButtonBase
{
    //  路由事件
    public static readonly RoutedEvent ClickEvent;

    static Button()
    {
        //  注册事件
        Button.ClickEvent = EventManager.RegisterRoutedEvent("Click",
            RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Button));
        //  ...
    }

    //  事件包装器（可选）
    public event RoutedEventHandler Click
    {
        add { AddHandler(Button.ClickEvent, value); }
        remove { RemoveHandler(Button.ClickEvent, value); }
    }

    protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
    {
        //  ...
        //  触发事件
        RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this));
        //  ...
    }
}
```

路由事件的实现与依赖属性的实现十分相似。它在实现类型中是一个 RoutedEvent 类型的对象，习惯上被 public static readonly 修饰，且拥有 Event 后缀。EventManager 的 RegisterRoutedEvent 方法用来向事件系统进行注册新的路由事件，并且仅在实现类型中可选的“事件包装器”（如 Click 事件）中调用 AddHandler 和 RemoveHandler 方法来添加和移除它。这两个方法不是从 DependencyObject 继承而来的，而是来自 System.Windows.UIElement（Button 等的高层基类），负责在适当的路由事件身上附加或移除事件处理器（委托）。在 OnMouseLeftButtonDown 内部，UIElement 定义的 RaiseEvent 方法被调用，传入参数分别为对应的路由事件对象和事件源的引用。

路由策略和事件处理器

注册的时候，每个路由事件都会选择一种路由策略。路由策略是事件触发时沿元素树传递的方式，WPF 的事件系统一共包含三种事件路由策略：

（1）**下降式**（tunneling）：事件首先在根元素上被触发，继而沿着元素树向下触发子元素的相应事件，直到到达事件源子元素（或直到事件处理方法将这个事件标记为已处理）为止。

（2）**上升式**（bubbling）：事件首先在事件源元素上被触发，继而沿着元素树向上触发父元素的相应事件，直到达到根元素（或直到事件处理方法将这个事件标记为已处理）为止。

（3）**直接式**（direct）：事件**只能**在事件源元素上被触发。这与普通的.NET 事件类似，但它可以参与路由事件的特殊机制，如“事件触发器”（event trigger）。

路由事件的处理方法拥有和一般.NET 事件类似的签名。第一个参数是 System.Object 类型的事件源对象 sender，它保存着附加了该处理函数的元素对象的引用；第二个参数是从 RoutedEventArgs（派生自 System.EventArgs）派生的类的实例 e。RoutedEventArgs 公开了四个有用的属性：

（1）Source：位于**逻辑树**上的事件源元素（如 Button）。

（2）OriginalSource：位于**视觉树**上的事件源元素（如 Button 的 ButtonChrome 或 TextBlock）。

（3）Handled：一个布尔变量，设置为 true 表示事件已被处理，它可以精确地中止上升式和下降式路由事件的传递。

（4）RoutedEvent：真正的路由事件对象（如 Button.ClickEvent ），它对鉴别相同处理函数处理不同路由事件很有帮助。

Source 和 OriginalSource 的区别仅在于处理物理事件上，如鼠标事件等。对于那些与视觉树上元素不需要有直接关系的事件，Source 和 OriginalSource 引用的是相同的对象，比如 Click 事件（因为用户可以通过键盘触发它，如在拥有焦点的按钮上按下空格键）。

2 路由事件实战

UIElement 类为键盘、鼠标和手写输入设备定义了许多路由事件，它们之中大多数是上升式（bubbling）事件，但其中也有许多事件拥有一个下降式的副本。下降式事件可以很容易识别出来，因为根据习惯，它们都包含 Preview 前缀，并且习惯上在其上升式副本被触发之前立即得到触发。例如，PreviewMouseMove 是一个下降式事件，它在 MouseMove 上升式事件之前被触发。

隐藏在这一对事件之后的思想是给予元素一种有效取消或修改即将发生的事件的机会。一般地，WPF 内建元素仅对上升式事件起作用（在同时定义了上升式和下降式事件的时候），即确保下降式事件的“preview”前缀名副其实。例如，设想我们想要实现一个限制输入内容为一个确定的模式或正字表达式的文本框。如果我们处理 TextBox 的 KeyDown 事件，我们最好的做法是将已经显示在 TextBox 中的文本清除。但是如果我们处理 TextBox 的 PreviewKeyDown 事件，那么我们可以将它标记为“已处理”，这样不仅终止了事件向下传递，而且终止了 KeyDown 事件触发的事件源向上传递。对于这种情形，TextBox 永远都不会接收到 KeyDown 的通知，继而当前的字符永远不会被显示出来。

下面的例子描述一个简单的上升式事件的用法：

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="AboutDialog" MouseRightButtonDown="Window_MouseRightButtonDown"
    Title="WPF 揭秘" SizeToContent="WidthAndHeight" Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 揭秘（版本 3.0）
        </Label>
        <Label>(C)2006 SAMS 出版集团</Label>
        <Label>安装的章节：</Label>
        <ListBox>
            <ListBoxItem>第一章</ListBoxItem>
            <ListBoxItem>第二章</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>您已经成功注册了本产品。</StatusBar>
    </StackPanel>
</Window>

using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Controls;

public partial class AboutDialog : Window
{
    public AboutDialog()
    {
        InitializeComponent();
    }

    private void AboutDialog_MouseRightButtonDown(object sender, MouseButtonEventArgs
e)
    {
        this.Title = "Source = " + e.Source.GetType().Name + ", OriginalSource = " +
            e.OriginalSource.GetType().Name + " @ " + e.Timestamp;

        Control source = e.Source as Control;

        if (source.BorderThickness != new Thickness(5))
        {
            source.BorderThickness = new Thickness(5);
            source.BorderBrush = Brushes.Black;
```

```
            }
            else source.BorderThickness = new Thickness(0);
        }
    }
```

无论何时右键单击事件上传到 Window 对象，AboutDialog_MouseRightButtonDown 事件处理方法会执行两个动作：（1）在 Window 的标题栏显示事件的有关信息；（2）添加（随后移除）一个粗黑边框围绕在右键单击的逻辑树元素外面。值得注意的是，当我们右键单击 Label 时，它的 Source 是当前 Label 的引用，而其 OriginalSource 则是其视觉子元素 TextBlock 的引用。

如果我们运行这个示例并且在任意元素上单击右键，我们将会注意到两种有趣的行为：

（1）当我们右键单击任意 ListBoxItem 的时候，Window 都不会接收到 MouseRightButtonDown 事件。这是因为 ListBoxItem 内部处理了这个事件，这与 MouseLeftButton 事件实现了项目选择一样，都是终止了事件上传。

（2）右键单击 Button 时，Window 可以接收到 MouseRightButtonDown 事件，但是为按钮设置 Border 属性不会有变化。这是因为按钮的默认视觉树不像 Window、Label、ListBox、ListBoxItem 和 StatusBar，树中根本就没有 Border 元素。

尽管将 RoutedEventArgs 的 Handled 属性设置为 true 似乎终止事件向上或向下传递，但通过程序代码借助 AddHandler 的重载就可以打破这个规则。如下例：

```
public AboutDialog()
{
    InitializeComponent();
    this.AddHandler(Window.MouseRightButtonDownEvent,
        new MouseButtonEventHandler(AboutDialog_MouseRightButtonDown), true);
}
```

将第三个参数设置为 true，AboutDialog_MouseRightButtonDown 事件处理方法就可以在右键单击 ListBoxItem 时为其添加边框了。

我们应当在任何可能的时候避免处理已处理完成的事件，因为事件应该在它发生的处理方法中得到处理。为 Preview 版本的事件附加一个事件处理器是更好的选择。

终止事件上/下传递不切实际，更加正确的说法是，事件上/下传递在路由事件被标记为已处理的时候仍然会继续，但是事件处理器方法在默认情况下只能接收到未处理的事件。

3 附加事件

当树上元素公开了路由事件时，按上升或下降方式传递它是很自然的，但是 WPF 还支持在没有定义路由事件的元素上上下传递另外的路由事件！这要感谢“附加事件”（attached events）表示法。

附加事件的操作非常像附加属性（其上/下传递的方式与附加属性的继承非常类似）。下例再次改变了 AboutDialog：

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="AboutDialog"
        ListBox.SelectionChanged="ListBox_SelectionChanged"
        Button.Click="Button_Click"
        Title="关于 WPF 揭秘" SizeToContent="WidthAndHeight" Background="OrangeRed">
    <StackPanel>
        <Label FontWeight="Bold" FontSize="20" Foreground="White">
            WPF 揭秘（版本 3.0）
        </Label>
        <Label>(C)2006 SAMS 出版集团</Label>
        <Label>已安装的章节：</Label>
        <ListBox>
            <ListBoxItem>第一章</ListBoxItem>
            <ListBoxItem>第二章</ListBoxItem>
        </ListBox>
        <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
            <Button MinWidth="75" Margin="10">Help</Button>
            <Button MinWidth="75" Margin="10">OK</Button>
        </StackPanel>
        <StatusBar>您已经成功注册了本产品。</StatusBar>
    </StackPanel>
</Window>
```

```
public partial class AboutDialog : Window
{
    public AboutDialog()
    {
        InitializeComponent();
    }

    private void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
    {
        if (e.AddedItems.Count > 0)
            MessageBox.Show("您选择了 " + e.AddedItems[0]);
    }

    private void Button_Click(object sender, RoutedEventArgs e)
    {
        MessageBox.Show("您单击了 " + e.Source);
    }
}
```

上例中，AboutDialog 处理了两个不属于它（Window）的事件——ListBox 的上升式事件 SelectionChanged 和 Button 的 Click。这两个路由事件定义在 Window 元素中，因此需要添加所属类的名称作为前缀。

每个路由事件都可以被用作附加事件。在编译时，XAML 编译器看到的 Click 事件是定义在 Button 中的；在运行时，系统直接调用 AddHandler 方法来将这两个事件附加到 Window 对象上。因此，它们等价于下面的 C#代码：

```
    public AboutDialog()
{
        InitializeComponent();
        this.AddHandler(ListBox.SelectionChangedEvent,
                new SelectionChangedEventHandler(ListBox_SelectionChanged));
        this.AddHandler(Button.ClickEvent,
                new RoutedEventHandler(Button_Click));
}
```

由于路由事件携带丰富的信息，因此我们可以在一个通用的事件处理器中处理它们，如下所示：

```
private void GenericHandler(object sender, RoutedEventArgs e)
{
        if (e.RoutedEvent == Button.ClickEvent)
        {
                MessageBox.Show("您单击了 " + e.Source);
        }
        else if (e.RoutedEvent == ListBox.SelectionChangedEvent)
        {
                SelectionChangedEventArgs sce = (SelectionChangedEventArgs)e;
                if (sce.AddedItems.Count > 0)
                        MessageBox.Show("您选择了 " + sce.AddedItems[0]);
        }
}
```

4.

```
<Window xmlns=”http://schemas.microsoft.com/winfx/2006/xaml/presentation”
xmlns:x=”http://schemas.microsoft.com/winfx/2006/xaml”
x:Class=”Window1”
Button.Click=”Button_Click”
<Button Text="TestButton" Width="50" Height="30">
</Window>
```

例子中，因为 Window 本身没有定义 Click 事件，所以我们必须指定 Click 事件属性的名称前缀，也就是定义事件的类名。经过这样的定义后，点击在 Window 中的 TestButton，也会激发属性声明的 Click 事件，调用对应的 Button_Click 方法。

为什么这样的定义可以通过呢？首先编译时，XAML 会看到 Button 类确实定义了一个 Click 的.NET 事件。在运行时，会直接调用 AddHandler 把这两个事件依附到 Window 对应的类当中。所以上面用 XAML 属性声明的事件代码与下面的程序代码等效：

```
public Window1
{
 InitializeComponent();
 this.AddHandler(Button.ClickEvent, new RoutedEventHandler(Button_Click));
}
```

5.

WPF 中的 Event

Routed Events 意指一个事件在一个相互关联的元素组成的树中传递，而非直接传递到一个特定的目标。

它可以在两个方向进行传递，Bubbling, tunneling

事件 Routed 的三种方式

1〉 Bubbling：事件由源头传递到元素树的最顶点

2〉 tunneling：事件由树的顶端传递的源头

3〉 direct：事件不在树中移动，表现方式类同标准的 CLR 事件

如何写 Routed Event

示例：

```
Public static readonly RoutedEvent TapEvent =

EventManager.RegisterRoutedEvent(“Tap”,

RoutingStrategy.Bubble,

typeof(RoutedEventHandle),

typeof(MyButtonSimple)

);
```

```
// Provide CLR accessors for the event

Public event RoutedEventHandle Tap

{

       Add{AddHandler(TapEvent, value);}

       Remove{RemoveHandler(TapEvent, value);}

}
```

如何在 Xaml 关联事件

```
<Button Click=”blSetColor”>button</Button>
```

为什么要使用 Routed Event

对于一组需要交互的控件，可以将父元素作为一个通用的事件监听者，然后使用同一个事件函数来处理交互。

示例如下

```
A grouped button set
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>

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;

}
```

自定义 Event

示例如下：

```
//Invoke 该事件件
protected virtual void OnPipeValueChanged(double oldValue, double newValue)
        {
            RoutedPropertyChangedEventArgs<double> args = new
RoutedPropertyChangedEventArgs<double>(oldValue, newValue);
            args.RoutedEvent = PipeControl.PipeValueChangedEvent;
            RaiseEvent(args);
        }

// 注册该事件件
        public static readonly RoutedEvent PipeValueChangedEvent =
            EventManager.RegisterRoutedEvent("PipeValueChanged",
            RoutingStrategy.Bubble,
            typeof(RoutedPropertyChangedEventHandler<double>),
            typeof(PipeControl));
```

Chapter : Base Element in WPF

主要记录要写 WPF Control，必须掌握的层次结构，明白自己要写的 Control 应该从那层派生更合适 Object

1. DispatcherObject

●·类似于消息分发机制
●·提供对 WPF 线程模型的支持
●·每一个 DispatcherObject 关联一个 Dispatcher
●·

2. DependencyObject

●·可以认为是 WPF 的 Object 基础
●·提供 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。
●·

3. Visual

```
public abstract class Visual : DependencyObject
```

●·Visual 对象是一个核心 WPF 对象，其主要作用是提供呈现支持
●·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。

具体支持的功能包括：

✧·坐标转换：
✧·输出显示：为可视对象呈现持久的序列化绘图内容。
✧·转换：对可视对象执行转换。
✧·剪辑：为可视对象提供剪辑区域支持。
✧·命中测试：确定指定的坐标（点）或几何图形是否包含在可视对象的边界内。
✧·边界框计算：确定可视对象的边框。

在体系结构上，Visual 对象不包含对其他应用程序开发要求/与该对象的呈现密切相关的 WPF 功能的支持，如下所示：

✧·事件处理
✧·布局
✧·样式
✧·数据绑定
✧·全球化

4. UIElement

```
public class UIElement : Visual, IAnimatable, IInputElement
```

●·可以呈现为子元素
●·对基本输入事件和命令的支持，可以对用户输入作出响应（包括通过处理事件路由或命令路由控制输入发送的位置）
●·包含用于确定 UIElement 可能的子元素的大小和位置的逻辑（由布局系统解释时）
●·可以引发遍历逻辑元素树路由的路由事件
●·对动画属性值的基本支持，支持动画系统的某些方面

Note: Visibility 状态影响该元素的所有输入处理。不可见的元素不参与命中测试并且不接收输入事件，即使鼠标放在该元素可见时的边界上也如此。

5. FrameworkElement

```
public class FrameworkElement:UIElement, IFrameworkInputElement, IInputElement,
```

ISupportInitialize

●·其他框架特定的布局特征
●·支持更丰富的有关属性的元数据报告
●·某些输入基类的特定于类的实现及其附加属性或附加事件
●·样式支持
●·更多的动画支持
●·对动态资源引用的支持。
●·对数据绑定的支持。
●·逻辑树的概念
WPF 中的控件，图形操作，动画多媒体，以及 3 D 效果就是由下面 4 个类的派生类来完成的：

1 Control 是创建自定义应用程序控件的基类。可以重写 Control 类所提供的属性，方法，事件等，为自定义控件添加自定义逻辑。 构建 WPF 应用程序页面的 window 类就派生自它.还有 button,TextBox 等控件也派生自他。

2 Shape:WPF 中呈现二维矢量图形的基础类。有 Line、Polyline、Polygon、Path、Rectangle 和 Ellipse 等子类。可从 Shape 类进行派生以实现自定义矢量图形基元。从 Shape 派生是确保这些自定义基元使用 WPF 布局系统的协议的最简单方法。

3 Freezable:WPF 中对动画和多媒体的操作类，基本上都是派生自它，它实现了多种生成深层克隆的方法。

4 Visual3D:提供可视三维对象通用的服务和属性，其中包括命中测试、坐标转换和边界框计算。与 Visual 类一样，只不过是 3 D 的基础类.

Chapter WPF 中的资源）

资源是保存在可执行文件中的一种不可执行数据。通过资源我们可以包含图像、字符串等等几乎是任意类型的数据。如此重要的功能，.NET Framework 当然也是支持的，其中内建有资源创建、定位、打包和部署的工具。在.NET 中可以创建.resx 和.resources 文件。其中.resx 由 XML 项组成。.resx 只是一种中间格式，不能被应用程序直接使用，它必须用工具转换为.resource 格式。

在 WPF 中，资源的含义和处理方式与传统的 Win32 和 Windows Forms 资源有所区别。首先，不需要创建.resx 文件，只需要在工程中指出资源即可，其它所有的工作都由 WPF 完成。其次，WPF 中的资源不再像.NET 中有资源 ID，在 XAML 中引用资源需要使用 Uri。最后，在 WPF 的资源中，几乎可以包含所有的任意 CLR 对象，只要对象有一个默认的构造函数和独立的属性。在 WPF 本身的对象中，可以声明如下四种对象：Style、Brushes、Templates 和 DataSource。

在定义具体的资源之前，我们先考虑如下几个相关的问题：

1、资源的有效范围：在 WPF 中，所有的框架级元素（FrameworkElement 或者 FrameworkContentElement）都有一个 Resource 属性。也就是说。我们可以在所有这类元素的 Resource 子元素中定义属性。在实践中，最常用的三种就是三种根元素所对应的资源：Application、Page 和 Window。顾名思义，在 Application 根元素下定

义的资源将在当前整个应用程序中可见，都可以访问。在 Page 和 Window 中定义的元素只能在对应的 Page 和 Window 中才能访问。

2、资源加载形式：WPF 提供了两种资源类型：Static 资源和 Dynamic 资源。

两种的区别主要有两点：A）、Static 资源在编译时决议，而 Dynamic 资源则是在运行时决议。B）、Static 资源在第一次编译后即确定了相应的对象或者值。此后不能对其进行修改，即使修改成功也是没有任何意义的，因为其它使用资源的对象不会得到通知。Dynamic 资源不同，它只有在运行过程中真正需要时，才会在资源目标中查找。所以我们可以动态的修改 Dynamic 资源。显而易见，Dynamic 资源的运行效率将比 Static 资源低。

3、不管是 Static 资源还是 Dynamic 资源，所有的资源都需要设置 Key 属性:x:Key=”KeyName”。因为 WPF 中的资源没有资源 ID，需要通过资源 Key 来标识以方便以后访问资源。范围资源时我们根据资源的类型使用 StaticResource 或者 DynamicResource 标记扩展。

好了，对 WPF 中的资源所有了解后，我们看一些简单的例子：

```
<Window

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

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

<StackPanel>

    <StackPanel.Resources>

        <SolidColorBrush x:Key="MyBrush" Color="gold"/>

    </StackPanel.Resources>

    <TextBlock Foreground="{StaticResource MyBrush}" Text="Text"/>

</StackPanel>

</Window>
```

在这个例子中，我们在 StackPanel 元素的 Resource 子元素中定义了一个 SolidColorBrush 资源。然后在后面通过 StaticResouce 标记扩展，利用前面的 x:Key 属性访问定义好的资源。

资源除了可以在 XAML 声明外，还可以通过代码进行访问控制。支持 Resource 属性的对象都可以通过 FindResource、以及 Resource.Add 和 Resource.Remove 进行控制：

```
<Window

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

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

<Window.Resouce>

    <SolidColorBrush x:Key="MyBrush" Color="gold"/>

</Window.Resouce>

</Window>
```

我们先在代码 XAML 的 Window.Resource 中定义了一个 MyBrush。在代码中可以如下对其进行访问：

SolidColorBrush brush = this.FindResource("MyBrush") as SolidColorBrush;

如果需要进一步修改或者删除资源时，可如下编码：

this.Resouce.Remove(“MyBrush”);　　//删除 MyBrush 资源

this.Resouce.Add(“MyBrush”);　　//重新动态添加资源

说明：以上三处的 this 引用都是特指我们定义 MyBrush 的元素 Window。读者朋友可根据实际情况修改。

2.

在本系列的之十三中简单介绍了 WPF 中资源的资源。但是，没有给出任何具体的实例，在这个 Post 中将给出一个动态资源的例子，也算是响应 daxian110 的请求。并适当的扩展在前一个 Post 当中没有涉及的知识。

我们先看一个例子程序：

```
<Window x:Class="WindowsApplication1.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="WindowsApplication1" Height="150" Width="100" Loaded="OnLoaded"
   >
   <Canvas>
     <Button Click="OnClick" Canvas.Left="10" Canvas.Top="20"
          Width="80" Height="30" Content="{DynamicResource TestRes1}"/>
     <Button Canvas.Left="10" Canvas.Top="60" Width="80"
          Height="30" Content="{DynamicResource TestRes2}"/>
   </Canvas>
</Window>
```

程序很简单，在窗口中添加了两个按钮，我们需要关注的是其中对 Content 属性。这个属性的作用就是设置按钮的内容。为什么这里的名称不是 Text，而是 Content？如此命名的原因和 WPF 中控件一个非常重要的概念有关：WPF 中几乎任何的控件（也就是 Element）都可以作为一个容器存在。也就是说我们在 Content 属性中可以包含其它任何你想显示的内容。不止是字符串文本。这种抽象的处理使我们可以把所有的内容等同对待，减少了很多处理上的麻烦。在本例子中，Content 属性被和一个 TestRes1 和 TestRes2 关联起来。这个 TestRes 到底是什么呢？这就是动态资源的名称。具体的内容在显示按钮的时候决定。

注意上面 Window 中的 Loaded 属性，通过它我们可以设置一个函数名称，它将 Window 加载完成后被调用。下面就看看如何用代码控制 TestRes：

```
     private void OnLoaded(object sender, RoutedEventArgs e)
{
    string szText1 = "Res Text1";
    this.Resources.Add("TestRes1", szText1);

          string szText2 = "Res Text2";
    this.Resources.Add("TestRes2", szText2);
}
```

OnLoaded 是 Window1 类中的一个成员函数，在这个函数里，我们需要添加资源，因为我们的 XAML 中需要使用 TestRes1 和 TestRes2，运行时如果找不到对应资源，程序将失败。

现在，我们调用 Add 方法添加资源。第一个参数是资源的名称，第二个参数是添加的资源对象。

程序的运行效果如图 1：

图 1

图 2

接下来我们看看修改资源的方法。在上面 XAML 的第一个按钮的 Click 属性中我们指定了一个 OnClick 事件方法。它将在点击按钮时调用，现在我们通过这个事件来修改另一个按钮的 Content 资源：

```
private void OnClick(object sender, RoutedEventArgs e)
{
    string szText = "New Res Text";
    this.Resources.Remove("TestRes2");
    this.Resources.Add("TestRes2", szText);
}
```

OnLoaded 实现同样的简单，先调用 Remove 方法删除已有的 TestRes2 资源，然后重新添加一个新的 TestRes2 资源对象。点击第一个按钮后，下面按钮的文本将自动修改为新的资源对象。运行效果如图 2。

XAML 加载器在分析 XAML 文件时，发现 StaticResource，将会在当前 Element 的资源中查找指定的 Key，如果查找失败，将沿着逻辑树向上查找，直到 Root 元素。如果还没有找到资源，再查找 Application 下定义的资源。在 Application 中定义的资源适用于整个应用程序。类似于全局对象。注意：使用 Static 资源时，不能向前引用。即使偶尔程序运行成功，向前引用的效率将非常低，因为它需要查找所有的 ResourceDictionay。对于这种情况，使用 DynamicResource 将更适合。

另一方面，XAML 加载器发现 DynamicResource 时，将根据当前的属性设置创建一个表达式，直到运行过程中资源需要，才根据表达式从资源中查找相关内容进行计算，返回所需的对象。注意，DynamicResource 的查找于 StaticResource 基本类似，除了在定义了 Style 和 Template 时，会多一个查找目标。具体的细节可参数 MSDN。

4.

在前一个 Post 当中，我从资源编译行为的角度讨论了 WPF 中的资源。但是，不管是 Resource 还是 Content 都是在编译时声明资源。如果我们打破这个限制，不希望指定完全确认的资源地址。WPF 提供了一种类似 IE 地址定位的抽象，它根据应用程序部署的位置决议。

WPF 将应用程序的起源地点进行概念上的抽象。如果我们的应用程序位于 http://yilinglai.cnblogs.com/test dir/test.application。我们应用程序的起源地点是 http://yilinglai.cnblogs.com/testdir/，那么我们就可以在应用程序中这样指定资源位置：

```
<Image Source="pack://siteoforigin:,,,/Images/Test.JPG"/>
```

通过这种包装的 Uri，使用资源的引用更加灵活。那么，这种类似 Internet 应用程序的资源包装 Uri 指定方式有什么优点呢？

1）、应用程序 Assembly 建立后，文件也可被替代。

2）、可以使文件只在需要使才被下载。

3）、编译应用程序时，我们不需要知道文件的内容（或者文件根本不存在）。

4）、某些文件如果被嵌入到应用程序的 Assembly 后，WPF 将不能加载。比如 Frame 中的 HTML 内容，Media 文件。

这里的 pack://其实是一种 URI（Uniform Resource Identifiers）语法格式。pack://<authority><absolute_path>，其中的 authority 部分是一个内嵌的 URI。注意这个 URI 也是遵守 RFC 2396 文档声明的。由于它是被嵌入到 URI 当中，因此一些保留字符必须被忽略。在我们前面的例子中，斜线（”/”）被逗号（”,”）代替。其它的字符如”%”、”?”都必须忽略。

前面例子中的 siteoforigin 可以理解为一种 authority 的特例。WPF 利用它抽象了部署应用程序的原始站点。比如我们的应用程序在 C:\App，而在相同目录下有一个 Test.JPG 文件，访问这个文件我们可以用硬编码 URI file:///c:/App/Test.JPG。另外一种方法就是这种抽象性：pack://siteoforigin:,,/Test.JPG。这种访问方法的便利是不言而喻的！在 XPS 文档规范中，对 URI 有更好的说明。有兴趣朋友可以在此下载。

也许你看到现在对此的理解有些问题。不用太着急，随着你对 WPF 越来越熟悉，会有更多的体会。对于 WPF 的新手（我也是），对于此点不必过度纠缠。因为 WPF 的 Application 类中提供了一些有用的方法：

Application.GetResourceStream (Uri relativeUri);

Application.GetContentStream(Uri relativeUri);

Application.GetRemoteStream (Uri relativeUri);

通过使用这些函数，隐藏了 URI 定位的细节。从这些函数的名称我们可以看出，它们分别对应于我在前面介绍的三种类型：Content、Resource 和 SiteofOrigin。

最后，简单的说明一下另一种使用资源的方式，直接定义资源，不使用任何的属性，具体的用法看例子就明白了：

```
<StackPanel Name="sp1">
   <StackPanel.Resources>
     <Ellipse x:Key="It1" Fill="Red" Width="100" Height="50"/>
     <Ellipse x:Key="It2" Fill="Blue" Width="200" Height="100"/>
   </StackPanel.Resources>
   <StaticResource ResourceKey="It1" />
   <StaticResource ResourceKey="It2" />
</StackPanel>
```

5．DynamicResource 与 StaticResource 的区别

资源的使用

下面的示例在 page 的根元素定义了一个 SolidColorBrush 画刷作为一个资源，并展示如何用它来设置 Page 中子元素的属性

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

</Page.Resources>
 <StackPanel>
   <DockPanel>
        <Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30" Background="{St
aticResource MyBrush}" Margin="40">Button</Button>
            </DockPanel>
 </StackPanel>
</Page>
```

每个框架级别的元素（FrameworkElement 或 FrameworkContentElement）都有一个 Resources 属性，我们可以在任何元素上定义资源，不过习惯上在根元素上定义，如上面 的 xmal 代码中<Page.Resources/>定义资源。

通过属性 x:Key 给每个资源赋予一个唯一的关键字。这样我们就可以在 Xmal 的其它地方通过 Key 值来操作对应的资源了。如下示例，使用资源给元素的属性赋值

```
<Button Background=”{StaticResource MyBrush}” />
```

StaticResource 和 DynamicResource

资源可以被当作 StaticResource 和 DynamicResource 两种类型来引用。

当引用资源时，下面的考虑将影响你是选择 StaticResource 还是 DynamicResource 来它。

1〉 如何为应用程序创建资源(资源是在一个 Page 中，在 APP 范围还是在松散的 Xaml 中或仅仅在程序集中)

2〉 应用程序功能：是否在运行时改变资源

3〉 每个资源引用类型不同的寻找行为

StaticResources

StaticResources 在如下情况下使用比较好

1〉 在资源第一次引用之后无需再修改资源的值

2〉 StaticResource 引用不会基于运行时的行为进行重新计算。比如在重新加载 Page 的时候。

3〉 当需要设置的属性不是 DependencyObject 或 Freezable 类型的时候，需要用 staticResource

4〉 当需要将资源编译到 dll 中，并打包为程序的一部份，或者希望在各应用程序之间共享

5〉 当需要为一个自定义控件创建一个 theme，并 theme 中使用资源，就需要使用 StaticResource。因为 StaticResource 的资源查找行为时可预测的，并且本身包含在 theme 中。而对于 DynamicResource，即使资源是定义在 theme 中，也只能等到运行时确定，导致一些可能意料不到的情况发生。

6〉 当需要使用资源设置大量的依赖属性的时候（dependency property），依赖属性具有属性系统提供的值缓存机制，所以如果能在程序装载时设置依赖属性的值，依赖属性就不需要检查自己的值并返回最后的有效值了。可以获得显示时的好处。

Static resource 查询行为

1〉 查找使用该资源的元素的 resource 字典

2〉 顺逻辑树向上查找父元素的资源字典，直到根节点

3〉 查找 Application 资源

4〉 不支持向前引用。即不能引用在引用点之后才定义的资源

Dynamic Resource

Dynamic resources 一般使用在如下场合

1〉 资源的值依赖一些条件，而该条件直到运行时才能确定。这包括系统资源，或是用户可设置的资源。例如，可以创建引用系统属性诸如 SystemColors,SystemFonts 来设置值，这些属性是动态的，他们的值来自于运行环境和操作系统

2〉 为自定义控件引用或创建 theme style

3〉 希望在程序运行期间调整资源字典的内容

4〉 希望资源可以向前引用

5〉 资源文件很大，希望在运行时加载

6〉 要创建的 style 的值可能来自于其它值，而这些值又依赖于 themes 或用户设置

7〉 当引用资源的元素的父元素有可能在运行期改变，这个时候也需要使用动态资源。因为父元素的改变将导致资源查询的范围。

Dynamic resource 查询行为

1〉 查找使用该资源的元素的 resource 字典

如果元素定义了一个 Style 属性，将查找 Style 中的资源字典

如果元素定义了一个 Template 属性，将查找 FrameworkTemplate 中的资源字典

2〉 顺逻辑树向上查找父元素的资源字典，直到根节点

3〉 查找 Application 资源

4〉 查找当前激活状态下的 theme 资源字典。

5〉 查找系统资源

Dynamic resource 的限制条件

1〉 属性必须是依赖属性，或是 Freezable 的

Chapter WPF 中的 Style

Style 是一种修改属性值是方法。我们可以将其理解为对属性值的批处理。对批处理大家应该不会感到默认。对，通过 Style 我们可以批量修改属性的值。先从一个简单的 Style 例子开始：

```
    <Window x:Class="Viewer3D.WindowSettins"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Viewer3D Settings"
  >
 <Window.Resources>
  <Style TargetType="CheckBox">
   <Setter Property="Height" Value="20"/>
   <Setter Property="Width" Value="50"/>
   <EventSetter Event="Checked" Handler="Checked_Click"/>
   <Setter Property="VerticalAlignment" Value="Center"/>
  </Style>
 </Window.Resources>
</Window>
```

第一感觉你可能会奇怪，为什么 Style 在资源里呢？我个人直接将理解为“批处理”的缘故。因此 Style 是修改多个对象的属性值，它不从属于单独的元素对象。另一个疑惑的问题是 Style 没有设置 x:Key 属性。这是一个非常关键的设置。如果我们设置了 Style 的 x:Key 属性，相当于在当前 Window 是资源中定义了一个名称为 x:Key 设定值的 Style 对象。记住定义的效果相当于对象。如果没有设置 x;Key，那么这个 Style 将对属于这个 Window 中所有 CheckBox 生效。这就起到了批处理的效果。

首先设定的是 Style 的 TargetType 属性，它表示我们希望修改的目标类型。然后定义一个 Setters 的集合。每个 Setter 都表示修改的一个属性或者事件。Property 设置属性名称，Value 设置属性值。Event 设置事件名称，Handler 设置事件的响应函数名称。只要你在 Resource 做了类似的定义，在此 Window 中所使用的任何 ChekcBox 都会默认这些属性值。是不是很方便呢？我们在此定义一次，可以节省很多代码。

也许你还会问：这样的统一修改属性太武断、霸道了吧！也许是的。我们只修改部分 Element 的属性值，而希望对某些特殊的 Element 做特殊处理。这样的需求 WPF 当然也是支持的。看看下面的代码：

```
<Style BasedOn="{StaticResource {x:Type CheckBox}}"
   TargetType="CheckBox"
   x:Key="WiderCheckBox">
  <Setter Property="Width" Value="70"/>
</Style>
```

WPT 通过 BasedOn 对这种特殊的 Style 提供了支持。很明显，BasedOn 的意思是我们当前的 Style 基于在资源的 CheckBox。这里又看到了 x;Key 扩展标记。因为我们需要的是一个特例，一个特殊的 Style 对象。为了以后引用这个 Style，我们需要 x:Key 的标识作用。其它的代码与前面类似。

定义后，引用这个特殊 Style 的 CheckBox 的代码是这样的：

```
<CheckBox Style="{StaticResource WiderCheckBox}">Win</CheckBox>
```

你已经看到，我们在 CheckBox 中指定了 Style 属性，并引用前面的 StaticResource 标记。

Chapter: ControlTemplate [1]

通过前面的介绍，我们已经知道 WPF 支持用 Style Setters 修改控件的属性值，以改变控件的外观。我们知道，WPF 的任何控件都有视觉树和逻辑树。但是 Style 有它自己的局限性：它只能修改控件已有树型结构的属性，不能修改控件的树型层次结构本身。而在实际运用中，我们常常需要对控件进行更高级的自定义。此时，可以需要使用 ControlTemplate 才能实现。

在 WPF 中，ControlTemplate 用来定义控件的外观。我们可以为控件定义新的 ControlTemplate 来实现控件结构和外观的修改。同样，我们先看一个例子：

```
<Style TargetType="Button">
  <Setter Property="OverridesDefaultStyle" Value="True"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="Button">
        <Grid>
          <Ellipse Fill="{TemplateBinding Background}"/>
          <ContentPresenter HorizontalAlignment="Center"
                            VerticalAlignment="Center"/>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

从例子代码我们可以看出，ControlTemplate 含有模板的语义。也就是说它影响的应该是多个控件。而这个功能恰好可以利用 Style 实现。所以，在理解了 Style 之后，这样的代码应该不会感到陌生。首先把 OverridesDefaultStyle 设置为 True，表示这个控件不使用当前 Themes 的任何属性。然后用 Setters 修改控件的 Template 属性。我们定义了一个新的 ControlTemplate 来设置新的值。

同样地，ControlTemplate 也使用 TargetType 属性，其意义与 Style 的 TargetType 一样。它的 x:Key 属性也是如此。然后，由一个 Grid 来表示控件的视觉内容。其中的 TemplateBinding 与 Binding 类似，表示当前 Ellipse 的显示颜色与 Button 的 Background 属性保持同步。TemplateBinding 可以理解为 Binding 在模板中的特例。而另一个 ContentPresenter 与 WPF 的基本控件类型有关，一种是 ContentControl，一个是 ItemControl。在上面的例子中定义的是基于 ContentControl 的 Button。所以使用 ContentPresenter 来表示内容的显示。

WPF 中每个预定义的控件都有一个默认的模板，因此，在我们学习自定义模板（也就是自定义控件）之前，可以先熟悉了解 WPF 的默认模板。为了便于查看模板的树形结构层次，我们可以将模板输出为 XML 文件格式，这样能有助于理解。

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;
settings.IndentChars = new string(' ', 4);
settings.NewLineOnAttributes = true;
StringBuilder strbuild = new StringBuilder();
XmlWriter xmlwrite = XmlWriter.Create(strbuild, settings);
XamlWriter.Save(ctrl.Template, xmlwrite);
```

这里的 ctrl 是一个实例化的 Control 类。并且 Control 需要已经显示在屏幕上，否则 Control.Template 可能为 NULL。

2.

前面关于 ControlTempalte 的 Post 当中，只说明了如何定义的外观。如果对于很复杂的自定义控件，通常我们还需要在 ControlTemplate 使用 Resource。很显然，Resource 的目的是便于实现元素的重用。另外，我们的自定义模板通常是在 XAML 中完成的，因为用代码实现是非常烦琐的。对于小的应用程序，这个 ControlTemplate 一般直接定义在 XAML 的根元素。对于大的应用程序，通常应该定义在专门的资源 XAML 文件中，根元素是 Resource Dictionary。

不管定义在什么地方，除了前面用 Style 定义外观，以及用 Resource 实现元素重用外，ControlTemplate 包括一个 Trigger 元素，它描述在控件属性发生变化时控件的外观如何变化。比如自定义 Button 时需要考虑鼠标在 Button 上移动时控件的外观。Trigger 元素也是可选的，比如文本标签元素，它一般不包括 Trigger。

在 ControlTemplate 中使用资源很简单，与其他元素中的资源一样：

```
<ControlTemplate x:Key="templateThermometer" TargetType="{x:Type ProgressBar}">
   <ControlTemplate.Resources>
      <RadialGradientBrush x:Key="brushBowl"
                           GradientOrigin="0.3 0.3">
         <GradientStop Offset="0" Color="Pink" />
         <GradientStop Offset="1" Color="Red" />
      </RadialGradientBrush>
  </ControlTemplate.Resources>
  <!-- 忽略其他相关内容-->
</ControlTemplate>
```

接下来是 Trigger 的使用。利用 Trigger 对象，我们可以接收到属性变化或者事件发生，并据此做出适当的响应。Trigger 本身也是支持多种类型的，下面是一个属性 Trigger 的例子：

```
      <Style TargetType="ListBoxItem">
  <Setter Property="Opacity" Value="0.5" />
  <Style.Triggers>
    <Trigger Property="IsSelected" Value="True">
      <Setter Property="Opacity" Value="1.0" />
      <!--其他的 Setters->
```

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

这段代码设置 ListBoxItem 的 Opacity 属性的默认值为 0.5。但是，在 IsSelected 属性为 True 时，ListBoxItem 的 Opacity 属性值为 1。从上面的代码还可以看出，在满足一个条件后，可以触发多个行为（定义多个 Setters）。同样地，上面的 Triggers 也是一个集合，也可以添加多个 Trigger。

注意上面的多个 Trigger 是相互独立的，不会互相影响。另一种情况是需要满足多个条件时才触发某种行为。为此，WPF 提供了 MultiTrigger 以满足这种需求。比如：

```
      <Style TargetType="{x:Type Button}">
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition Property="IsMouseOver" Value="True" />
        <Condition Property="Content" Value="{x:Null}" />
      </MultiTrigger.Conditions>
      <Setter Property="Background" Value="Yellow" />
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

这就表示只有 IsMouseOver 为 True、Content 为 NULL 的时候才将 Background 设置为 Yellow。

以上的 Trigger 都是基于元素属性的。对于鼠标移动等事件的处理，WPF 有专门的 EventTrigger。但因 EventTrigger 多数时候是和 Storyboard 配合使用的。因此，我将在后面介绍动画的时候详细说明 EventTrigger。

另一方面，现在所讨论的 Trigger 都是基于属性的值或者事件的。WPF 还支持另一种 Trigger：DataTrigger。显然，这种数据 Trigger 用于数据发生变化时，也就是说触发条件的属性是绑定数据的。类似地，数据 Trigger 也支持多个条件：MultiDataTrigger。他们的基于用法和前面的 Trigger 类似。

下一个 Post 我将分析 Windows SDK 中 ControlTemplateExamples 的例子，这个例子涉及了很多的控件模板使用。

3.

在实际应用中，ControlTemplate 是一个非常重要的功能。它帮助我们快速实现很 Cool 的自定义控件。下面我以 Windows Vista SDK 中的例子 ControlTemplateExamples 为基础，简单地分析 ControlTemplate 的使用。这个例子工程非常丰富，几乎包含了所有的标准控件。所以，在实现自定义控件时，可以先参考这样进行适当的学习研究。

首先是 App.xaml 文件，这里它把 Application.StartupUri 属性设置为 Window1.xaml。然后把工程目录 Resource 下所有的控件 xaml 文件都合成为了应用程序范围内的资源。

```
      <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Resources\Shared.xaml" />
        <!-- 这里省略 -->
        <ResourceDictionary Source="Resources\NavigationWindow.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
</Application.Resources>
```

这样的用法很有借鉴意义。在 WPF 中实现 Skin 框架也随之变得非常简单。值需要动态使用不同的 XAML 文件即可。然后是 Window1.xaml 文件。它里面几乎把所有的控件都显示了一遍。没有什么多说的。重点看 Resource 目录下的自定义控件文件。这里的控件太多，不可能每个都说说。我只挑选其中的 Button.xaml 为例：

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

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

     <!-- Focus Visual -->

     <Style x:Key="ButtonFocusVisual">
  <Setter Property="Control.Template">
   <Setter.Value>
    <ControlTemplate>
     <Border>
      <Rectangle Margin="5" StrokeThickness="3"
      Stroke="#60000000" StrokeDashArray="4 2"/>
     </Border>
    </ControlTemplate>
   </Setter.Value>
  </Setter>
 </Style>
 <!--..............-->
</ResourceDictionary>
```

因为这个 XAML 文件作为资源使用，所以其根元素是 ResourceDictionary，而不再是 Window/Application 等等。同时，资源文件也可以相互的嵌套，比如上面的包含的 Shared.xaml 文件。然后定义了一个 Style，注意这里的目标类型为 Control.Template，也就是针对所有的控件模板有效，所以 Style 添加了一个 x:Key 属性。这样就阻止 Style 适用于当前的所有控件。我们必须显式的引用这个 Style。相关内容，可以参考我前面的 Style 文章。

另一个需要说明的是<ControlTemplate>的子元素，可以是任何的 VisualTree。比如这里的 Border，也可以是 Grid 等等。好了，现在定义了一个名为 ButtonFocusVisual 的模板，下面只需要引用它即可。

```
    <Style TargetType="Button">
  <!--.............-->
  <Setter Property="FocusVisualStyle" Value="{StaticResource ButtonFocusVisual}"/>
  <!--.............-->
  <Setter Property="Template">
   <Setter.Value>
    <ControlTemplate TargetType="Button">

        <Border x:Name="Border" ...../>

        <ControlTemplate.Triggers>
      <Trigger Property="IsKeyboardFocused" Value="true">
       <Setter TargetName="Border" Property="BorderBrush" Value="{StaticResource DefaultedB
orderBrush}" />
      </Trigger>
     </ControlTemplate.Triggers>

       </ControlTemplate>
   </Setter.Value>
```

```
    </Setter>
</Style>
```

这是真正影响控件外观的代码。因为在定义 Style 的时候没有指定具体的 x:Key，所以将影响所有的 Button。如你所见，在 FocusVisualStyle 这个属性（类型是 Style）上我们用资源方式引用了前面定义的命名 Style：Button FocusVisual。接下来是定义 Template，并为其子元素 Border 定义了一个名称。然后就是 ControlTemplate 的触发器。在 IsKeyboardFocused 属性满足条件的情况下，我们把 Border（注意这个 Border 不是类型，而是具体的某个对象）的 BorderBrush 修改为另一个静态资源。结合前面的 Post，理解也就不难了。

最后，我们还会发现一个有趣的问题：这个例子虽然是 ControlTempalte，但工程名称却是 SimpleStyle，从这一点我们也可以看出：Style 和 Template 通常是配合使用才能真正的实现丰富的自定义功能。

WPF 性能优化点:

在建立漂亮 UI 的同时，我们还需要关注应用程序的性能，WPF 尤其如此。下面从 MS 的文档中总结出了一些有用的性能优化点。在实际编写的过程中，可以参考。这个 Post 非完全原创，是根据一些文档总结出来的。

1、建立逻辑树的时候，尽量考虑从父结点到子结点的顺序构建。因为当逻辑树的一个结点发生变化时（比如添加或删除），它的父结点和所有的子结点都会激发 Invalidation。我们应该避免不必要的 Invalidation。

2、当我们在列表（比如 ListBox）显示了一个 CLR 对象列表（比如 List）时，如果想在修改 List 对象后，List Box 也动态的反映这种变化。此时，我们应该使用动态的 ObservableCollection 对象绑定。而不是直接的更新 Item Source。两者的区别在于直接更新 ItemSource 会使 WPF 抛弃 ListBox 已有的所有数据，然后全部重新从 List 加载。而使用 ObservableCollection 可以避免这种先全部删除再重载的过程，效率更高。

3、在使用数据绑定的过程中，如果绑定的数据源是一个 CLR 对象，属性也是一个 CLR 属性，那么在绑定的时候对象 CLR 对象所实现的机制不同，绑定的效率也不同。

A、数据源是一个 CLR 对象，属性也是一个 CLR 属性。对象通过 TypeDescriptor/PropertyChanged 模式实现通知功能。此时绑定引擎用 TypeDescriptor 来反射源对象。效率最低。
B、数据源是一个 CLR 对象，属性也是一个 CLR 属性。对象通过 INotifyPropertyChanged 实现通知功能。此时绑定引擎直接反射源对象。效率稍微提高。
C、数据源是一个 DependencyObject，而且属性是一个 DependencyProperty。此时不需要反射，直接绑定。效率最高。

4、访问 CLR 对象和 CLR 属性的效率会比访问 DependencyObject/DependencyProperty 高。注意这里指的是访问，不要和前面的绑定混淆了。但是，把属性注册为 DependencyProperty 会有很多的优点：比如继承、数据绑定和 Style。所以有时候我们可以在实现 DependencyProperty 的时候，利用缓存机制来加速访问速度：看下面的缓存例子：

```
    public static readonly DependencyProperty MagicStringProperty =
  DependencyProperty.Register("MagicString", typeof(string), typeof(MyButton), new PropertyMetada
ta(new PropertyInvalidatedCallback(OnMagicStringPropertyInvalidated),new GetValueOverride(MagicS
tringGetValueCallback)));

     private static void OnMagicStringPropertyInvalidated(DependencyObject d)
{
  // 将缓存的数据标识为无效
  ((MyButton)d)._magicStringValid = false;
}

     private static object MagicStringGetValueCallback(DependencyObject d)
{
  // 调用缓存的访问器来获取值
  return ((MyButton)d).MagicString;
}
```

```
        // 私有的 CLR 访问器和本地缓存
  public string MagicString
  {
    get
    {
      // 在当前值无效时，获取最新的值保存起来
      if (!_magicStringValid)
      {
        _magicString = (string)GetValueBase(MagicStringProperty);
        _magicStringValid = true;
      }

            return _magicString;
    }
    set
    {
      SetValue(MagicStringProperty, value);
    }
  }

        private string _magicString;
  private bool _magicStringValid;
```

另外，因为注册的 DependencyProperty 在默认是不可继承的，如果需要继承特性，也会降低 DependencyProroperty 值刷新的效率。注册 DependencyProperty 属性时，应该把 DefaultValue 传递给 Register 方法的参数来实现默认值的设置，而不是在构造函数中设置。

5、使用元素 TextFlow 和 TextBlock 时，如果不需要 TextFlow 的某些特性，就应该考虑使用 TextBlock，因为它的效率更高。

6、在 TextBlock 中显式的使用 Run 命令比不使用 Run 命名的代码要高。

7、在 TextFlow 中使用 UIElement（比如 TextBlock）所需的代价要比使用 TextElement（比如 Run）的代价高。

8、把 Label（标签）元素的 ContentProperty 和一个字符串（String）绑定的效率要比把字符串和 TextBlock 的 Text 属性绑定的效率低。因为 Label 在更新字符串是会丢弃原来的字符串，全部重新显示内容。

9、在 TextBlock 块使用 HyperLinks 时，把多个 HyperLinks 组合在一起效率会更高。看下面的两种写法，后一种效率高。

A、

```
<TextBlock Width="600" >
  <Hyperlink TextDecorations="None">MSN Home</Hyperlink>
</TextBlock>
<TextBlock Width="600" >
  <Hyperlink TextDecorations="None">My MSN</Hyperlink>
</TextBlock>
```

B、

```
<TextBlock Width="600" >
  <Hyperlink TextDecorations="None">MSN Home</Hyperlink>
  <Hyperlink TextDecorations="None">My MSN</Hyperlink>
</TextBlock>
```

10、任与上面 TextDecorations 有关，显示超链接的时候，尽量只在 IsMouseOver 为 True 的时候显示下划线，一直显示下划线的代码高很多。

11、在自定义控件，尽量不要在控件的 ResourceDictionary 定义资源，而应该放在 Window 或者 Application 级。因为放在控件中会使每个实例都保留一份资源的拷贝。

12、如果多个元素使用相同的 Brush 时，应该考虑在资源定义 Brush，让他们共享一个 Brush 实例。

13、如果需要修改元素的 Opacity 属性，最后修改一个 Brush 的属性，然后用这个 Brush 来填充元素。因为直接修改元素的 Opacity 会迫使系统创建一个临时的 Surface。

14、在系统中使用大型的 3D Surface 时，如果不需要 Surface 的 HitTest 功能，请关闭它。因为默认的 HitTest 会占用大量的 CPU 时间进行计算。UIElement 有应该 IsHitTestVisible 属性可以用来关闭 HitTest 功能。

Chapter:WPF 的数据处理 [1]

数据绑定，这是 WPF 提供的一个真正的优点。除了可以用在传统的绑定环境中，数据绑定已经被扩展应用到控件属性上。学习应用数据绑定，也能真正的体现 XAML 的好处。到底什么是数据绑定呢？也许你从字面上已经理解的很不错了。通过数据绑定，我们在应用程序 UI 和程序逻辑之间建立了一种联系。正常建立绑定后，在数据的值发生改变后，绑定到数据的元素将自动更新、体现出数据的变化。

同样，我们先看几个相关的知识点：

1、DataContext 属性。设置 DataContext 属性，其实就是指定数据上下文。那么数据上下文又是什么呢？又是一个新的概念：数据上下文允许元素从它的父元素继承数据绑定的数据源。很简单，在某个元素的 DataContext 中指定的值，那么在这个元素的子元素也可以使用。注意，如果我们修改了 FrameworkElement 或者 FrameworkContentElement 元素的 DataContext 属性，那么元素将不再继承 DataContext 值。也就是说新设置的属性值将覆盖父元素的设置。如何设置 DataContext 属性，稍后说明。

2、数据源的种类。也许，WPF 提供的数据绑定只是实现了一项普通的功能而已，但是，WPF 中所支持的多种数据源使得它的数据绑定功能将更加强大。现在，WPF 支持如下四种绑定源：

（1）、任意的 CLR 对象：数据源可以是 CLR 对象的属性、子属性以及 Indexers。对于这种类型的绑定源，WPF 采用两种方式来获取属性值：A）、反射（Reflection）；B）、CustomTypeDescriptor，如果对象实现了 ICustomTypeDescriptor，绑定将使用这个接口来获取属性值。

（2）、XML 结点：数据源可以是 XML 文件片断。也可以是 XMLDataProvider 提供的整个 XML 文件。

（3）、ADO.NET 数据表。我对 ADO.NET 的了解不够，在此不做过多评论。

（4）、Dependency 对象。绑定源可以是其它 DependencyObject 的 DependencyProperty 属性。

3、数据绑定的方式：（1）、OneWay，单一方向的绑定，只有在数据源发生变化后才会更新绑定目标。（2）、TwoWay，双向绑定，绑定的两端任何一端发生变化，都将通知另一端。（3）、OneTime，只绑定一次。绑定完成后任何一端的变化都不会通知对方。

在上面的第二点我介绍了数据源的种类，注意这里的概念和下面要说明的指定数据源的方式的区别。目前，指定数据源有三种方式，我们可以通过任何一种方式来指定上述的任何一种数据源：

（1）、通过 Source 标记。我们可以在使用 Binding 使用 Source 标记显式指定数据源。

（2）、通过 ElementName 标记。这个 ElementName 指定了一个已知的对象名称，将使用它作为绑定数据源。

（3）、通过 RelativeRource 标记。这个标记将在后面说明 ControlTemplate 和 Style 时再进行说明。

现在我们说明了很多和数据源相关的内容。但是再绑定的时候，我们还需要指定绑定对象的属性名称。所以 WPT 提供了一个 Path 标记。它被用来指定数据源的属性。也即是数据源将在数据源对象的 Path 所指定的属性上寻找属性值。

现在，理论的东西讲了一堆，我将在后面用一些简单的例子进行说明。

2.

在上一个 Post 当中，我叙述了 WPF 中的数据绑定相关的一堆理论知识。现在，我们将对其中的某些方面通过实例做进一步的分析。

在介绍 WPF 数据绑定源的种类时，第一种就是任意的 CLR 对象。这里需要注意的是 WPF 虽然支持任意的 CLR 对象，但是一个普通的 CLR 对象类却不行。我们还需要在 CLR 对象类上实现一种变化通知机制。

WPF 把这种通知机制封装在了 INotifyPropertyChanged 接口当中。我们的 CLR 对象类只要实现了这个接口，它就具有了通知客户的能力，通常是在属性改变后通知绑定的目标。

下面是一个简单的例子，实现了一个支持通知功能的 Camera 类：

```
using System;
using System.ComponentModel;
using System.Windows.Media.Media3D;
namespace LYLTEST
{
   public class Camera : INotifyPropertyChanged
   {
      private PerspectiveCamera m_Camera;
      public event PropertyChangedEventHandler PropertyChanged;
      public Camera()
      {
         m_Camera = new PerspectiveCamera();
      }
      private void NotifyPropertyChanged(String info)
      {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(info));
        }
```

```
        }
         public PerspectiveCamera CameraProp
        {
           get { return m_Camera; }
           set
           {
              if (value != m_Camera)
              {
                 this.m_Camera = value;
                 NotifyPropertyChanged("CameraProp");
              }
           }
        }
    }
}
```

这一段代码很简单，首先引入类中使用的 INotifyPropertyChanged 和 PerspectiveCamera 需要的名字空间。这里与普通 CLR 类的区别在于首先有一个公有的 PropertyChangedEventHandler 事件类型。然后我们在.NET 属性包装 CameraProp 判断属性是否发生了变化，如果是，则用当前是属性名称字符串“CameraProp”调用另一个私有函数 NotifyPropertyChanged。由它根据属性的名称构造一个 PropertyChangedEventArgs 对象，并完成对 PropertyChanged 的调用。它才是属性变化时真正应该调用的一个通知事件。

最后一点，如果我们需要通知所以的属性都发生了变化，则将上面的属性字符串“CameraProp”用参数 NULL 替代即可。

3.

最近比较忙些，好多天没有写 WPF 了。今天，我们继续回到前面的话题：WPF 中的数据处理。前面讲过，通过实现 INotifyPropertyChanged，我们可以改变使任意的 CLR 对象支持 WPF 的绑定源。但是，INotifyPropertyChanged 通常只应用在单个的类属性上。在现实应用中，我们还会遇到另外一种情况：我们需要监视某一堆的数据是否发生变化。也就是说我们绑定的数据源不再是一个单独数据对象。比如，绑定源是一个数据表时，我们希望在表中任何一条数据发生变化就能得到通知。（这里暂不考虑 WPF 绑定对 ADO.NET 的支持。）

WPF 提供了一个 ObservableCollection 类，它实现了一个暴露了 INotifyPropertyChanged 的数据集合。也就是说我们不需要自己对每个单独的数据实现 INotifyPropertyChanged 结构。我们先看看如何实现一个简单的绑定数据集合。

```
    namespace NSLYL
{
   public class LYLDataObj
   {
       public LYLDataObj(string name, string description)
      {
         this.name = name;
         this.description = description;
      }

            public string Name
      {
         get { return name; }
         set { name = value; }
      }

            public string Description
      {
         get { return description; }
         set { description = value; }
      }

      private string name;
      private string description;
   }


   public class LYLDataObjCol : ObservableCollection<LYLDataObj>
   {
      public LYLDataObjCol()
      {
         this.Add(new LYLDataObj("Microsot", "Operating System"));
         this.Add(new LYLDataObj("Google", "Search"));
      }
   }
}
```

代码很简单，基本上就是这样的一个模板。然后，我们就可以把 LYLDataObjCol 绑定到一个需要多项数据的 Element 之上，比如 ListBox、ComboBox 等等。

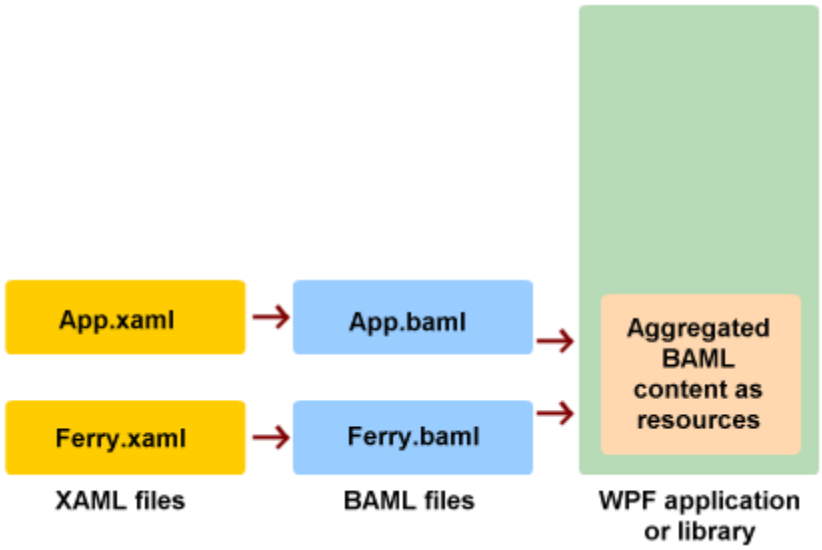
```
<ListBox ItemsSource="{StaticResource dataObj}" .../>
```

绑定之后，只要我的 LYLDataObjCol 对象发送了变化，ListBox、ComboBox 的数据也会有对应的变化。

到现在，我们已经知道在绑定的时候有两种指定数据源的方式：1、DataContext，关于它我们在这个 Post 有简单介绍。2、直接用 Binding 类的 Source 属性。那么，我们在使用的时候如何区别呢？首先，Source 的优先级比 DataContext 高，只有 Source 不存在，或者在当前 Source 到不到需要的属性时才会查找 DataContext。除此之外，这两者没有真正的区别，只是建议使用 Source，它能有助于我们调试应用程序。因为通过它可以明确的得到 Source 的信息。而 DataContext 支持一种继承。可以在父 Element 指定 Source 源。这同时也成为了 DataContext 的一个优点：如果多个 Element 需要绑定同一个 Source 源，那么我们只需要在一个地方指定 DataContext，就可以在其子 Element 使用。

Chapter :XAML

XAML 被编译为 BAML（Binary Application Markup Language）文件。通常，BAML 文件比 XAML 更小，编译后的 BAML 都是 Pre-tokenized 的，这样在运行时能更快速的加载、分析 XAML 等等。这些 BAML 文件被以资源的形式嵌入到 Assembly 当中。同时生成相应的代码（文件名称是**.g.cs 或者**.g.vb），这些代码根据 XAML 元素分别生成命名的 Attribute 字段。以及加载 BAML 的构造函数。



最后，关于 XAML 的优点，我附上一点翻译过来的条款，可能更直观：

XAML 除了有标记语言、XML 的优点外，还有如下一些优点：

用 XAML 设计 UI 更简单

XAML 比其他的 UI 设计技术所需编码更少。

XAML 设计的 UI 方便转移、方便在其他环境提交。比如在 Web 或 Windows Client。

用 XAML 设计动态 UI 非常容易

XAML 给 UI 设计人员带来新的革命，现在所有的设计人员不再需要.NET 开发的知识同样可以设计 UI。在不远的将来，终端用户可以看到更漂亮的 UI。

2.XAML 的名字空间

在前一篇文章中，指出 xmlns 的作用是设置 XML 文件的命名空间。类似的，xmlns:x 的作用也是指定命名空间。这里为什么是 x 而不是其他的，我们可以简单的理解为其只是 MS 的一个命名而已，没有任何特殊的意义，当然，为了避免和它的冲突，我们定义自己的命名空间的时候不能是 x。

而另一个 x:Class 的作用就是支持当前 Window 所对应的类，前面已经说过每个 XAML 元素都是一个 CLR 类型，这里的 x:Class 是 Window 的一个属性，属性的内容指出当前的窗口类是 FirstXAML 名字空间下的 Windows1。为什么需要类，而不全部用 XAML 实现？XAML 的主要作用还是编写 UI 部分，我们仍然需要用代码对程序逻辑进行更深层次的控制。

好了，这是两个最基本的名字空间。同样地，名字空间也可以自定义，并且这个自定义会给我们带来很大的方便。我们定义如下的一个类：

```
namespace DataBind4Image
{
   public class GroupData
   {
      //具体的细节忽略
    }
}
```

如果想在 XAML 文件中使用这个 GroupData 类对象，我们就可以通过自定义的名字空间引入这个类：

```
xmlns:local="clr-namespace:DataBind4Image"
```

这里的后缀 local 只是一个标识，你可以设置为任何你喜欢的唯一标识。通过这个引入定义我们就可以在 XAML 文件中用 local 来标识 DataBind4Image 当中的任何类。访问 GroupData 类时只需要加上 local 就可以识别了：

```
<local:DrawingGroupData/>
```

利用名字空间，除了可以引入我们定义的当前工程的类，还可以引入任何的 Assembly。直接看例子是最简单的：

```
<Window x:Class="WindowsApplication1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=System"
  >
  <ListBox>
    <sys:String>One</sys:String>
  </ListBox>
</Window>
```

例子当中引入.NET 的 System Assembly，通过它我们就可以直接使用 System 的任何类。利用这种类似的方式，我们可以在 XAML 中使用几乎所有的 DOTNET 框架类。

最后说明一下在 XAML 中 inline 嵌入程序逻辑处理代码的情况。利用<CDATA[…]]>关键字引入处理代码。这种情况在实际当中不太合适，我们不应该采用 UI 和逻辑混合的方式。详细的解释可以参数 Windows SDK 文档。

```
<![CDATA[
void Clicked(object sender, RoutedEventArgs e)
{
    button1.Content = "Hello World";
}
]]></x:Code>
```

前面提到过每个 XAML 元素表示一个.NET CLR 类。多数的 XAML 元素都是从 System.Windows.UIElement, System.Windows.FrameworkElement, System.Windows.FrameworkContentElement 和 System.Windows.ContentElement 继承。没有任何的 XAML 元素与.NET CLR 的抽象类对应。但是很多元素都有一个抽象类的派生类对应。

通常有如下四种通用的 XAML 元素：

Root 元素：Windows 和 Page 是最常用的根元素。这些元素位于 XAML 文件的根元素，并包含其他元素。

Panel 元素：帮助布置 UI 位置。常用的是 StackPanel, DockPanel, Grid 和 Canvas。

Control 元素：定义 XAML 文件的控件类型。允许添加控件并自定义。

Document 元素：帮助实现文档提交。主要分为 Inline 和 Block 元素组，帮助设计的外观类似文档。一些有名的 Inline 元素有 Bold，LineBreak, Italic。Block 元素有 Paragraph, List, Block, Figure 和 Table。

XAML 元素的属性与.NET 类对象的属性类似，XAML 的面向对象特征使得它的行为与之前的 HTML 类似。每个属性（实际上是类属性）继承了父元素的属性或者重载（如果重新设置了属性）。

3.

Visual tree and logic tree

```
<Window ......>
  <StackPanel>
   <Label>LabelText</Lable>
  </StackPanel>
</Window>
```

在这样一个简单 UI 中，Window 是一个根结点，它有一个子结点 StackPanel。而 StackPanel 有一个子结点 Label。注意 Label 下还有一个子结点 string（LabelText），它同时也是一个叶子结点。这就构成了窗口的一个逻辑树。逻辑树始终存在于 WPF 的 UI 中，不管 UI 是用 XAML 编写还是用代码编写。WPF 的每个方面（属性、事件、资源等等）都是依赖于逻辑树的。

视觉树基本上是逻辑树的一种扩展。逻辑树的每个结点都被分解为它们的核心视觉组件。逻辑树的结点对我们而言基本是一个黑盒。而视觉树不同，它暴露了视觉的实现细节。下面是 Visual Tree 结构就表示了上面四行 XAML 代码的视觉树结构：

并不是所有的逻辑树结点都可以扩展为视觉树结点。只有从 System.Windows.Media.Visual 和 System.Windows.Media.Visual3D 继承的元素才能被视觉树包含。其他的元素不能包含是因为它们本身没有自己的提交（Rendering）行为。

在 Windows Vista SDK Tools 当中的 XamlPad 提供查看 Visual Tree 的功能。需要注意的是 XamlPad 目前只能查看以 Page 为根元素，并且去掉了 SizeToContent 属性的 XAML 文档。如下图所示：

注意图中工具栏特别标记的地方。我们可以看到 Visual Tree 确实比较复杂，其中还包含有很多的不可见元素，比如 ContentPresenter。Visual Tree 虽然复杂，但是在一般情况下，我们不需要过多地关注它。我们在从根本上改变控件的风格、外观时，需要注意 Visual Tree 的使用，因为在这种情况下我们通常会改变控件的视觉逻辑。

WPF 中还提供了遍历逻辑树和视觉树的辅助类：System.Windows.LogicalTreeHelper 和 System.Windows.Media.VisualTreeHelper。注意遍历的位置，逻辑树可以在类的构造函数中遍历。但是，视觉树必须在经过至少一次的布局后才能形成。所以它不能在构造函数遍历。通常是在 OnContentRendered 进行，这个函数为在布局发生后被调用。

其实每个 Tree 结点元素本身也包含了遍历的方法。比如，Visual 类包含了三个保护成员方法 VisualParent、VisualChildrenCount、GetVisualChild。通过它们可以访问 Visual 的父元素和子元素。而对于 FrameworkElement，它通常定义了一个公共的 Parent 属性表示其逻辑父元素。特定的 FrameworkElement 子类用不同的方式暴露了它的逻辑子元素。比如部分子元素是 Children Collection，有是有时 Content 属性，Content 属性强制元素只能有一个逻辑子元素。

Chapter：（WPF 与 Win32）

说明：这里的 Win32 特指 Vista 操作系统之前的所有图形系统：GDI、GDI+、Direct3D。

GDI 是当今应用程序的主流图形库，GDI 图形系统已经形成了很多年。它提供了 2D 图形和文本功能，以及受限的图像处理功能。虽然在一些图形卡上支持部分 GDI 的加速，但是与当今主流的 Direct3D 加速相比还是很弱小。GDI+开始出现是在 2001 年，它引入了 2D 图形的反走样，浮点数坐标，渐变以及单个象素的 Alpha 支持，还支持多种图像格式。但是，GDI+没有任何的加速功能（全部是用软件实现）。

当前版本的 WPF 中，对一些 Win32 功能还没有很好的支持，比如 WMF/EMF 文件，单个象素宽度的线条等等。对于这些需求还需要使用 GDI/GDI+来实现。

在 Windows Vista 中，GDI 和 GDI+仍然支持，它们与 WPF 并行存在，但是基本上没有任何功能性的改进。对 GDI 和 GDI+的改进主要集中在安全性和客户相关问题上。WPF 的所有提交都不依赖于 GDI 和 GDI+，而是 Direct3D。并且所有的 Primitive 都是通过 Direct3D 的本地接口实现的。还记得我前面随笔中提到过的 Milcore 吗？它就是和 Direct3D 交互的非托管代码组件。由于 WPF 的大部分代码都是以托管代码的形式存在的，所以 WPF 中有

很多托管、非托管的交互。当然，在一些图形卡不支持 WPF 所需要的功能时，WPF 也提供了稍微低效的软件实现，以此来支持在某些 PC 上运行 WPF 应用程序。

在 Windows Vista 中，Direct3D 的关键改进就是引入了新的显示驱动模型。VDDM 驱动模型虚拟化了显卡上的资源（主要是显示内存），提供了一个调度程序，因此多个基于 Direct3D 的应用程序可以共享显卡（比如 WPF 应用程序和基于 WPF 的 Windows Vista 桌面窗口管理）。VDDM 的健壮性、稳定性也得到了提高，大量的驱动操作从内核（Kernel）模式移动到了用户（User）模式，这样提高了安全性，也简化了显示驱动的开发过程。

在 Windows Vista 中存在两个版本的 Direct3D：Direct3D 9 和 Direct3D 10。WPF 依赖于 Direct3D 9，这样能更广泛的解决兼容性问题。另外一个非常重要的原因就是为 Vista 的服务器版本提高方便，因为服务器版本的 Vista 对显卡和 Direct3D 基本上没有任何的要求。同时 WPF 也支持 Direct3D 10。Direct3D 10 依赖与 VDDM，只能在 Windows Vista 上使用。由于 Windows XP 没有 VDDM，虽然 Microsoft 做了很大的努力来改善 XP 中 Direct3D 9 相关驱动，提高内容的显示质量，但是由于 XP 中没有对显卡资源的虚拟化，强制所有的应用程序都用软件提交。

WPF 对某些多媒体的功能支持还需要依赖老的技术，比如 DirectShow。当我们进行音频视频的捕捉或者其它任务时，只能直接用 DirectShow 实现，然后再用 HwndHost 嵌入到 WPF 内容当中。

利用类似的技术，我们可以在 WPF 应用程序中显示自定义格式的内容。通过提供自定义的 DirectShow CODEC，然后用 Media 元素实现和 WPF 内容毫无限制的集成。

另外，WPF 对 XPS 等文档的打印输出也得到了极大的改善。XPS 文档本身的规范也极大的提高了其打印的质量，XPS 文档的规范可以参考 MSDN 的资料。除了打印，Vista 操作系统中对远程的改进也部分依赖于 WPF，比如有远程协助、远程桌面和终端服务等等。它们的实现过程是通过发送一系列的“远程”命名到客户端，客户根据自己 PC 的性能和命名进行显示，这样显示的质量能得到极大的提高。

在 WPF 中，对 Direct3D 进行各种封装。当然，如果你本身对 Direct3D/OpenGL 很熟悉，也可以直接在 WPF 中使用。封装后的 Direct3D 更容易使用。并且在 Web 应用程序（XBAP）也可以使用 Direct3D。在 WPF 中使用的 Direct3D，没有直接用非托管代码控制所拥有的灵活性，也不能直接对硬件进行底层控制。

WPF 中所有的提交都是矢量形式的，我们可以对图像或窗口进行任意级的放缩，而图像的质量不会有任何的损耗。

Chapter：XAML 中的类型转换

在前面关于 XAML 的 Post 当中，简单说明了 XAML 如果引入自定义名称空间。还提到过 XAML 基本上也是一种对象初始化语言。XAML 编译器根据 XAML 创建对象然后设置对象的值。比如：

```
<Button Width=”100”/>
```

很明显，我们设置 Button 的宽度属性值为 100。但是，这个“100”的字符串为什么可以表示宽度数值呢？在 XAML 中的所有属性值都是用文本字符串来描述，而它们的目标值可以是 double 等等。WPF 如何将这些字符串转换为目标类型？答案是类型转换器（TypeConverter）。WPF 之所以知道使用 Double 类型是因为在 FrameworkElement 类中的 WidthProperty 字段被标记了一个 TypeConverterAttribute，这样就可以知道在类型转换时使用何种类型转换器。TypeConverter 是一种转换类型的标准方法。.NET 运行时已经为标准的内建类型提供了相应的 TypeConverter。所以我们可以用字符串值指定元素的属性。

然而并不是所有的属性都标记了一个 TypeConverterAttribute。这种情况下，WPF 将根据属性值的目标类型，比如 Brush，来判断使用的类型转换器。虽然属性本身没有指定 TypeConverterAttribute，但是目标类型 Brush 自己标记了一个 TypeConverterAttribute 来指定它的类型转换器：BrushConverter。所以在转换这种属性时将自动使用目标值类型的 BrushConverter 将文本字符串类型的属性值转换为 Brush 类型。

类型转换器对开发人员有什么作用呢？通过它我们可以实现自定义的类型转换。下面一个例子演示了如何从 Color 类型转换为 SolidColorBrush。

```
[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;
   }
}
```

然后我们可以在资源中定义一个 ColorBrushConverter 实例（src 是一个自定义命名空间，引入了 ColorBrushConverter 类所在的 Assembly）。

```
<Application.Resources>
  <src:ColorBrushConverter x:Key="ColorToBrush"/>
</Application.Resources>
```

最后使用这个自定义的类型转换器：

```
<DataTemplate DataType="{x:Type Color}">
  <Rectangle Height="25" Width="25" Fill="{Binding Converter={StaticResource ColorToBrush}}"/>
</DataTemplate>
```

Chapter: XAML 的标记兼容性（Markup Compaibility）

继续 XAML 的话题，在前一个 Post 当中简单介绍了 XAML 的类型转换器（TypeConverters）。这次介绍一些 XAML 标记兼容性（Markup Compatibility）的相关内容。

利用 XAML 标记兼容性实现更加强大的注释功能

写过 XAML 的朋友应该都知道：在 XAML 中可以通过<!--****-->标记来实现注释。但是，利用 XAML 标记兼容性，还提供了其它更加强大的注释功能。

```
<Window
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
   xmlns:c="Comment"
   mc:Ignorable="c">
  <Canvas>
   <Button c:Width="100" Height="50">Hello</Button>
  </Canvas>
</Window>
```

看见了 Width 前面的 c 前缀吗？它的作用就是注释掉 Width 属性。是不是感觉比标记注释的方法简单。而且这个 c 前面不但可以应用在属性上，也可以直接应用在实例上，如下：

```
<Window
  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  xmlns:mc=http://schemas.openxmlformats.org/markup-compatibility/2006
  xmlns:c="Comment"
  mc:Ignorable="c">
 <Canvas>
  <c:Button Width="100" Height="50">Hello</c:Button>
 </Canvas>
</Window>
```

上面的代码就全部注释掉了 Button 实例。当然，这种方法不建议在最后的发布 XAML 文档中出现。只适合在 XAML 文档的开发过程中使用。

XAML 标记的向后兼容性

XAML 支持 XAML 文档的向前和向后兼容性。为了帮助说明 XAML 标记的向后兼容性，我们看一个自定义的 View 类，其中定义了一个 Color 类型的颜色属性 Color_Prop。

```
public class CLYLView
{
 Color _color;
public Color Color_Prop { get { return _color; } set { _color = value; } }
}
```

很简单，在 XAML 中，我们可以如下使用这个 CLYLView 类：

```
<CLYLView Color="Red" xmlns="… assembly-V1-uri…">
```

注意其中的 xmlns="… assembly-V1-uri…"，这就是一个所谓的 XmlnsCompatibleWith 属性。通过它我们指定了包含 CLYLView 的特定 Assembly。

现在，我们向 V2 版本的 CLYLView 添加了一个 Content 属性。如下所示：

```
public class CLYLView
{
 Color _color;
Content _content;
public Color Color_Prop { get { return _color; } set { _color = value; } }
public Content Content_Prop { get { return _content; } set { _content = value; } }

}
```

现在我们可以这样使用 V2 版本的 CLYLView 实例：

```
<CLYLView Color="Red" Content="Unknown" xmlns="... assembly-v2-uri…"/>
```

但是，我们仍然希望在 V2 版本的 CLYLView 支持 V1 版本。满足这种需求，我们可以用 XmlnsCompatableWith 声明一个新的 Assembly 与老的 Assembly 兼容。XAML 加载器看到了 XmlnsCompatableWith 属性，就会把默认地把所有对 V1 的引用处理为 V2 的引用。

向后兼容最大的一个好处就是：当我们只有新版的 Assembly 时，所有对老版 Assembly 的引用仍然是可读的，不会出现任何的错误。

Chapter: （XAML 的向前兼容性）

前一个 Post 当中，我们简单介绍了 XAML 的向后兼容性，以及利用标记兼容性实现注释的功能。现在，我们接着讨论 XAML 的向前兼容性问题。

同样地，我们用一个简单的例子来帮助说明 XAML 的向前兼容性。假设有一个自定义的 CLYLButton，实现了一个 Light 属性。在 V1 版本它的默认属性值是 Blue（蓝光）。在 V2 版本中支持属性值 Green（绿光）。假设我们在程序中利用 Light 属性实现了绿光效果。但是，如果恰好目标机器上的 V2 版本意外地被替换为了 V1 版本。此时，程序的行为应该怎么样呢？崩溃，不，我们希望它在没有 V2 的情况下能利用 V1 版本的默认值实现蓝光效果。如何实现且看 XAML 标记的向前兼容性。向前兼容性表示通过标记兼容性名字空间的 Ignorable 属性标识元素、属性和类，使它们可以动态的支持向前版本。

```
<CLYLButton V2:Light="Green"
xmlns="...assembly-v1-uri..."
xmlns:V2="...assembly-V2-uri..."
xmlns:mc=http://schemas.micrsoft.com/winfx/2006/markup-compatibility
mc:Ignorable="V2" />
```

这就利用了标记兼容性名字空间的 Ignorable 属性。mc:Ignorable=”V2”表示所有用 V2 前缀关联的名字空间中元素或者属性都是可以忽略的。如果现在只有 V1 版本的 CLYLButton，上面的代码就被 XAML 加载器解释为：

```
<CLYLButton Light=”Blue” xmlns=”… assembly-V1-uri …”/>
```

如果现在有 V2 版本的 CLYLButton，上面的代码将被 XAML 加载器解释为：

```
<CLYLButton Light=”Green” xmlns=”… assembly-V2-uri …”/>
```

XMAL 标记兼容性除了可应用在属性上，还可以应用在元素之上。仍然通过例子进行说明，定义如下的一个类：

```
[ContentProperty("Buttons")]
public class CElement {
    List<CLYLButton> _buttons = new List<CLYLButton>();
    public List<CLYLButton> Buttons { get { return _buttons; }
}
```

关于 ContentProperty 的用法可以参考 MSDN 文档 ContentPropertyAttribute Class

同样，我们可以如下编写 XAML 代码，使其可以同时兼容两个版本的 CElement。

```
<CElement mc:Ignorable="V2"
xmln="...assembly-v1-uri..."
xmlns:V2="...assembly-V2-uri..."
xmlns:mc="http://schemas.micrsoft.com/winfx/2006/markup-compatibility">
<CLYLButton Light="Blue" />
<V2:CLYLButton Light="Green"/>
</CElement>
```

这样，如果加载器有 V2 版本，则 Green 属性值生效。如果没有则被忽略。类似地，我们还可以完全自动地处理名字空间的类：

```
<CElement mc:Ignorable="v2"
xmln="...assembly-v1-uri..."
xmlns:V2="...assembly-v2-uri..."
xmlns:mc="http://schemas.micrsoft.com/winfx/2006/markup-compatibility">
<V2:Favor/>
</CElement>
```

加载时，如果没有 V2 版本存在，Favor 类实例同样将被忽略。

在 Markup Compatibility 中，除了有前面介绍的 Comment、Ignorable 属性修饰外，另一个有趣的就是 AlternateContent。利用 AlternateContent，我们能方便的实现可选内容。比如，我们的程序使用了 V2 版本 Assembly 的 CLYLButton 类，但是，如果没有找到这个 Assembly，那么它对应的内容自动用另一个指定版本 V1 替换，而不是兼容性体现的忽略。看下面的例子：

```
<CElement mc:Ignorable="v2"
xmln="...assembly-v1-uri..."
xmlns:v2="...assembly-v2-uri..."
xmlns:mc="http://schemas.micrsoft.com/winfx/2006/markup-compatibility">
<mc:AlternateContent>
    <mc:Choice Requires="V2">
        <CLYLButton Light="Green" Shape="Dog" />
```

```
            <V2:Favor/>
          </mc:Choice>
          <mc:Fallback>
            <CLYLButton Light="Blue"/>
          </mc:Fallback>
      </mc:AlternateContent>
</CElement>
```

这一段 XAML 代码在有 V1 版本的 Assembly 时将被视为：

```
<CElement xmln="...assembly-v1-uri...">
 <CLYLButton Light="Blue"/>
</CElement>
```

如果有 V2 版本的 Assembly，编译的结果如下：

```
<CElement xmln="...assembly-v1-uri...">
 <CLYLButton Light="Green"/>
<Favor/>
</CElement>
```

Chpater : *数据触发器(Data Trigger)*

数据触发器与属性触发器几乎一样,只是数据触发器可以由任何.NET 属性触发,而不仅仅是依赖属性.为了使用数据触发器,可向 Triggers 集合中添加 DataTrigger 对象然后指定属性/值对.同时可以用 Binding 来指定相关属性而不仅仅是属性名.

以下示例通过 binding 指定当自己的值为 disabled 的时候将自己禁用. 注意:Text 属性不是依赖属性.

```
<StackPanel Width="200">
   <StaticPanel.Resources>
      <Style TargetType="{x:Type TextBox}">
         <Style.Triggers>
            <DataTrigger Binding="{Binding RelativeResource=
                                  {RelativeResource Self}, Path=Text}"
                        value = "disabled">
              <Setter Property="IsEnabled" Value="false" />
            /DataTrigger>
         </Style.Triggers>
         <Setter Property="Background" Value=
                "{Binding RelativeResource={RelativeResource Self}, Path=Text}" />
      </Style>
   </StackPanel.Resources>
```

```
    <TextBox Margin="3" />
</StackPanel>
```

事件触发器(Event Trigger)

当一个已选择的事件产生时事件触发器会被激活.这个事件由触发器的 RouteEvent 属性指定,他在 Action 集合中包含一个或多个动作(从抽象类 TriggerAction 继承来的对象).

下边的示例展示了在 Button 的 Click 事件被触发时执行 DoubleAnimation 动作.

```
<Button>OK
    <Button.Triggers>
      <EventTrigger RouteEvent="Button.Click">
        <EventTrigger.Actions>
          <BeginStoryboard>
            <Storyboard TargetProperty="width">
              <DoubleAnimation From="50" To="100"
                                    Duration="0:0:5" AutoReverse="True" />
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
</Button>
```

Complie of XAML:

XAML 编译 通常包括三项事情：将一个 XAML 文件转换为一种特殊的二进制格式（BAML:Binary Application Markup Language）；将转换好的二进制资源嵌入到正在被创建的程序集中；然后执行链接操作将 XAML 和过程式代码自动连接起来。

每个 XAML 都会在编译过程中产生过程式代码，这是在编译过程中动态生成的，但这些过程代码仅仅是"粘合代码"Glue Code,类似于在运行时加载和解析松散 XAML 文件时所需要的代码（和普通的 Win Form 自动生成的代码用途类似），你可以在.g.cs(.g.vb)文件中找到这些代码。

同样你可以在 XAML 中混合使用过程代码（Code inside），当 XAML 编译后 X：code 元素中的部分将会放到部分类.g.cs 文件中。

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml>
<Button Click=”button_Click”>OK</Button>
<x:Code><![CDATA[
Void button_Click(object sender, RouteEventArgs e)
{
this.Close();
}
]]></x:Code>
</Window>
```

Chapter:

WPF 的体系结构上

这里主要讨论一下 WPF 的类结构。包含了 WPF 大多数主要子系统，描述了他们之间是怎样进行交互的，同时也对 WPF 体系为什么要选择这样或那样的方式来实现 WPF 的某些部分做一个大概的陈述

本文主要包含如下话题：

1〉 System.Object

2〉 System.Threading.DispatcherObject

3〉 System.Windows.DependencyObject

4〉 System.Windows.Media.Visual

5〉 System.Windows.UIElement

6〉 System.Windows.FrameworkElement

7〉 System.Windows.Controls.Control

8〉 Summary

9〉 Other topics

System.Object

WPF 编程模型主要通过托管代码实施。在 WPF 设计早期，关于在何处区分托管组件和非托管组件一直是一个争论不休的话题。CLR 提供了一系列的特性是个整个开发更具效率且获得更好的健壮性能，然而，鱼和熊掌不能兼得，这是以其他方面的代价换来的。

下图展示了构成 WPF 主要的组件。图的红色区域是 WPF 的主要组成。其中只有 milcore 是非托管模块，这主要是为了更好的将 DirectX 整合到 WPF 中。其一，WPF 中所有的显示由 DirectX 引擎处理，这将允许高效的硬件和软件渲染。其二，WPF 要求对内存和执行体良好的控制。其三，milcore 对显示极为敏感，不得不放弃许多 CLR 提供的好处来获得更好的性能

System.Threading.DispatcherObject

WPF 中有许多类继承自 DispatcherObject，DispatcherObject 提供了处理同步和并发的基本构造。WPF 建立在 dispatcher 提供的消息系统之上。它的工作方式很像 win32 中的消息泵；事实上，WPF 的 dispatcher 使用 User32 消息来实现线程间调用的。

当讨论 WPF 中的并发时，有两个核心概念需要弄清楚—dispatcher(分发器)thread affinity(线程亲缘性)

线程亲缘性发生在当一个组件要使用当前正在执行线程的变量来存储状态的时候。通用的方式是使用线程局部存储（TLS）来存储这些状态。线程亲缘性要求 EXE 的每个逻辑线程都只属于存在于 OS 中的物理线程。

System.Windows.DependencyObject

构建 WPF 的一个主要的体系哲学是属性优先于方法和事件。属性是声名性的并且允许你更方便的指定你想要的意图。它也支持模型驱动，数据驱动，UI 显示。它(前文哲学)允许更多的属性通过绑定实对应用程序的更加方便的控制。

为使更多的系统用属性来驱动，一个比 CLR 提供的属性系统更加丰富的属性系统是必须的。一个简单的例子就是 change notification（更改通知）。为了实现双向绑定。需要绑定双方都实现 change notification。为使行为关联到属性值，需要在属性值改变的时候获得通知。Microsoft.NET Framework 提供了一个接口:INotifyPropertyChange。通过它一个对象可以发布它自己的 change notification。

WPF 提供了丰富的属性系统，继承自 DependencyObject。它的一个基础是属性表达式。属性系统提供对属性的稀疏保存。

Animation:

```
<Viewbox
         Margin="10"
         OpacityMask="#99000000">
        <Button/>

    </Viewbox>
```

Databinding

1. 何为数据绑定

在 WPF 技术中控件基类（FrameworkElement、FrameworkContentElement）中 DataContext 属性实现了绑定机制，在 XAML 中也支持此机制。当一个控件的 DataContext 发生变化时，其子控件的 DataContext 也会继承父控件的 DataContext（前提是这个子控件没有另外赋值）。子控件的属性获取数据源中的数据，支持 XAML 通过

Binding 标记获取数据源中的值。数据源更新时刷新其每个子控件中的数据更新，实现一呼百应的效果！

Binding 绑定标记

XAML 处理器支持绑定机制的语法，以{Binding PropertyName=Value}格式出现。其中 PropertyName 基本分为“数据源指定标记、读取数据源标记、附加选项标记”，Value 可以使用其它标记获取特殊的值。绑定表达式中可以使用数据源标记、读取数据源标记、附加选项标记组合使用，也可以使用父控件绑定数据源，子控件继承父控件的数据源并进行绑定。

语法

代码:

```
<TextBox Name="theTextBox" />
<TextBlock Text="{Binding ElementName=theTextBox, Path=Text}"/>
同于
<TextBox Name="theTextBox" />
<TextBlock>
                <TextBlock.Text>
                      <Binding ElementName="theTextBox" Path="Text" />
                </TextBlock.Text>
</TextBlock>
```

数据源指定标记

Source 属性：通过其它扩展标记制定任何类型的对象实例为数据源，可以使用 StaticResource 等标记设置。例如：“{Binding Source={StaticResource xKeyElement}, XPath=//item}”，xKeyElement 是 XMLDataProvider 的对象，意思是获取以 xKeyElement 中所有以 item 为标签节点的数据为数据源。

ElementName 属性：制定当前 XAML 文档中任何以(x:Name)名称为 ElementName 值的对象为数据源。例如：“{Binding ElementName=ListBox1, Path=SelectedItem}”，意思是把 ListBox1 的 SelectedItem 属性为当前控件的数据源。

RelativeSource 属性：相对数据源，使用 RelativeSource 可以制定与自身相关联的对象为数据源。例如：“RelativeSource={RelativeSource Self}}”，读者注意“中间的是 RelativeSource{}是一个扩展标记，而外面的 RelativeSource={}是 Binding 标记的属性”。这段实例代码的意思是获取自身为数据源。

获取数据源标记

Path 属性：于“数据源指定标记”一起使用，获取数据源中的成员（属性）。如果数据源对象继承了 ICustomeTypeDescriptor 接口，将会从接口中获取属性值，否则使用类反射获取。例如：Text="{Binding ElementName=ListBox1, Path=Items[0].Text}"，其中 ElementName 设置数据源，获取 ListBox1.Items[0].Text 并赋予给 Text 属性。

XPath 属性：于“数据源指定标记”一起使用，通过 XPath 检索 XML 数据源内容。

```
<ListBox
    Width="400" Height="300" Background="Honeydew">
            <ListBox.ItemsSource>
                <Binding Source="{StaticResource InventoryData}"
```

```
            XPath="*[@Stock='out'] | *[@Number>=8 or @Number=3]"/>
        </ListBox.ItemsSource>

        <ListBox.ItemTemplate>
            <DataTemplate>
                <TextBlock FontSize="12" Foreground="Red">
      <TextBlock.Text>
        <Binding XPath="Title"/>
      </TextBlock.Text>
    </TextBlock>
            </DataTemplate>
        </ListBox.ItemTemplate>
</ListBox>
```

WPF 教程 - WPF 数据绑定机制详解_.Net 开发论坛 文章来源：IT 人才网(http://www.ad0.cn)

附加选项标记

Model 属性：BindingMode 枚举值，设置数据绑定的依赖机制。

OneWay 仅当源属性发生更改时更新目标属性。

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。

默认为双向绑定。例如：{Binding Pa IT 人才网(http://it.ad0.cn) th=Value, Mode=OneTime}，当数据源更新时当前控件重新绑定 www.ad0.cn

2.

WPF 中的数据绑定

到目前为止，很多人都知道使用 Windows® Presentation Foundation (WPF) 可以轻松地设计强大的用户界面。但是您可能并不知道它还提供了强大的数据绑定功能。使用 WPF，可以通过利用 Microsoft® .NET Framework 代码、XAML 或两者的组合进行数据操作。您可以绑定控件、公共属性、XML 或对象，从而使数据绑定比以前更快捷、灵活和简单。所以，让我们来看一下如何开始将控件绑定到您所选的数据源中。

数据绑定细节

要使用 WPF 数据绑定功能，您必须始终要有目标和源。绑定的目标可以是从 DependencyProperty 派生而来的任何可访问属性或元素，例如 TextBox 控件的 Text 属性。绑定的源可以是任何公共属性，包括其他控件、公共语言运行库 (CLR) 对象、XAML 元素、ADO.NET Dataset、XML 片段等的属性。为了帮助您正确实现绑定，WPF 包含了两个特殊的提供程序：XmlDataProvider 和 ObjectDataProvider。

现在让我们看一下 WPF 数据绑定技术的工作原理，我将列举一些实用的示例来说明它们的用法。

创建简单的绑定

首先，我们来看一个简单的示例，该示例说明了如何将 TextBlock 的 Text 属性绑定到 ListBox 的选定项。图 1 中的代码显示的是声明了六个 ListBoxItem 的 ListBox。该代码示例中的第二个 TextBlock 具有名为 Text（使用 XML 子元素 <TextBlock.Text> 在 XAML

属性元素语法中指定）的属性，它将包含 TextBlock 的文本。Text 属性声明了通过 <Binding> 标记与 ListBox 选定项的绑定。Binding 标记的 ElementName 属性指示 TextBlock 的 Text 属性要与其绑定的控件的名称。Path 属性指示我们将绑定到的元素（在本例中是 ListBox）的属性。此代码产生的结果是，如果从 ListBox 选择了一种颜色，该颜色的名称则会在 TextBlock 中显示。

图 1 基本但详细的控件绑定

```
<StackPanel>
    <TextBlock Width="248" Height="24" Text="Colors:"
        TextWrapping="Wrap"/>
    <ListBox x:Name="lbColor" Width="248" Height="56">
        <ListBoxItem Content="Blue"/>
        <ListBoxItem Content="Green"/>
        <ListBoxItem Content="Yellow"/>
        <ListBoxItem Content="Red"/>
        <ListBoxItem Content="Purple"/>
        <ListBoxItem Content="Orange"/>
    </ListBox>
    <TextBlock Width="248" Height="24" Text="You selected color:" />
    <TextBlock Width="248" Height="24">
        <TextBlock.Text>
            <Binding ElementName="lbColor" Path="SelectedItem.Content"/>
        </TextBlock.Text>
    </TextBlock>
</StackPanel>
```

为了使用简单的语法来进行数据绑定，可以对图 1 中列出的代码稍加修改。例如，我们用下列代码段替代 TextBlock 的 <Binding> 标记：

```
<TextBlock Width="248" Height="24"
    Text="{Binding ElementName=lbColor,
    Path=SelectedItem.Content}" />
```

这种语法称为属性语法，它压缩了 TextBlock 的 Text 属性内部的数据绑定代码。基本上，Binding 标记会连同它的属性一起被归入大括号内。

绑定模式

我可以继续以上述示例为例，将 TextBlock 的背景色绑定到在 ListBox 中选择的颜色。以下代码可将 Background 属性添加到 TextBlock 中，并使用该属性的绑定语法将其绑定到 ListBox 中选定项的值：

```
<TextBlock Width="248" Height="24"
Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,
    Mode=OneWay}"
x:Name="tbSelectedColor"
Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,
    Mode=OneWay}"/>
```

如果用户在 ListBox 中选择了一种颜色，那么该颜色的名称就会出现在 TextBlock 中，并且 TextBlock 的背景色会变为选定的颜色（请参见图 2）。

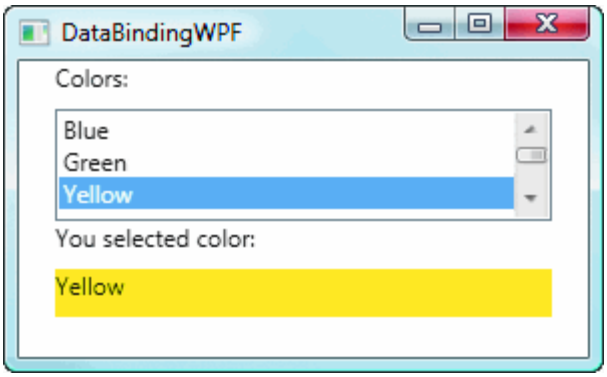


图 2 将一个源绑定到两个目标

请注意前一个示例中将 Mode 属性设为 OneWay 的语句。Mode 属性用于定义绑定模式，它将决定数据如何在源和目标之间流动。除 OneWay 之外，还有另外三种绑定模式：OneTime、OneWayToSource 和 TwoWay。

正如前面的代码段中所示，使用 OneWay 绑定时，每当源发生变化，数据就会从源流向目标。尽管我在示例中显式指定了此绑定模式，但其实 OneWay 绑定是 TextBlock 的 Text 属性的默认绑定模式，无需对其指定。和 OneWay 绑定一样，OneTime 绑定也会将数据从源发送到目标；但是，仅当启动了应用程序或 DataContext 发生更改时才会如此操作，因此，它不会侦听源中的更改通知。与 OneWay 和 OneTime 绑定不同，OneWayToSource 绑定会将数据从目标发送到源。最后，TwoWay 绑定会将源数据发送到目标，但如果目标属性的值发生变化，则会将它们发回给源。

在上述示例中，我使用了 OneWay 绑定，因为我希望只要 ListBox 选择发生变化，就将源（选定的 ListBoxItem）发送到 TextBlock。我不希望 TextBlock 的更改再回到 ListBox。当然，用户无法编辑 TextBlock。如果我想使用 TwoWay 绑定，可以将 TextBox 添加到此代码中，将其文本和背景色绑定到 ListBox，并将 Mode 属性设为 TwoWay。用户在 ListBox 中选择一种颜色后，该颜色就会显示在 TextBox 中，并且其背景色会相应变化。如果该用户在 TextBox 中键入了一种颜色（例如蓝绿色），ListBox 中的颜色名称就会更新（从目标到源），反过来，因为 ListBox 已经更新，所以此新值就会被发送到绑定到 ListBox 的 SelectedItem 属性的所有元素。这意味着 TextBlock 也会更新其颜色，并且将其文本值设置为该新的颜色（请参见图 3）。

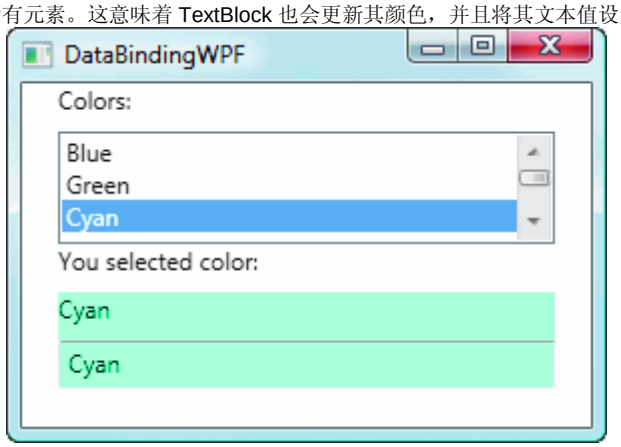


图 3 运行中的 TwoWay 绑定

下面是我刚才用来将 TextBlock (OneWay) 和 TextBox (TwoWay) 绑定到 ListBox 的代码：

```
<TextBlock Width="248" Height="24"
    Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,
    Mode=OneWay}" x:Name="tbSelectedColor"
    Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,
    Mode=OneWay}"/>

<TextBox Width="248" Height="24"
```

```
Text="{Binding ElementName=lbColor, Path=SelectedItem.Content,
Mode=TwoWay}" x:Name="txtSelectedColor"
Background="{Binding ElementName=lbColor, Path=SelectedItem.Content,
Mode=OneWay}"/>
```

如果我将 TwoWay 模式改回到 OneWay，用户则可以编辑 TextBox 中的颜色，且不会导致更改过的值被发回给 ListBox。

选择合适的绑定模式非常重要。当我想向用户显示只读数据时，我通常会采用 OneWay 模式。当我希望用户可以更改控件中的数据，并且让该变化能在数据源（DataSet、对象、XML 或其他绑定控件）中体现出来时，我会使用 TwoWay 绑定。如果想让用户在数据源不将其数据绑定到目标的情况下更改数据源，我发现 OneWayToSource 是个不错的选择。我曾经接到一个任务，要求在只读控件中显示与加载屏幕时一样的数据状态。在这个任务中，我使用了 OneTime 绑定。通过使用 OneTime 绑定，一系列只读控件均被绑定到了数据，并且当用户与表单交互且数据源的值发生更改时，绑定控件仍保持不变。这为用户提供了一种比较所发生更改的方法。此外，当源没有实现 INotifyPropertyChanged 时，OneTime 绑定也是一个不错的选择。

绑定的时间

在上述示例中，TextBox 允许 TwoWay 绑定到在 ListBox 中选定的 ListBoxItem。这会使数据在 TextBox 失去焦点时从 TextBox 流回 ListBox。为了改变导致将数据发送回源的这种情况，可以为 UpdateSourceTrigger 指定值，它是用于定义何时更新源的绑定属性。可以为 UpdateSourceTrigger 设置三个值：Explicit、LostFocus 和 PropertyChanged。

如果将 UpdateSourceTrigger 设置为 Explicit，则不会更新源，除非从代码调用 BindingExpression.UpdateSource 方法。LostFocus 设置（TextBox 控件的默认值）指示源在目标控件失去焦点时才会更新。PropertyChanged 值指示目标会在目标控件的绑定属性每次发生更改时更新源。如果您想指示绑定的时间，该设置非常有用。

绑定到 XML

绑定到数据源（例如 XML）和对象同样也非常方便。图 4 显示了 XmlDataProvider 的示例，其中包含将用作数据源的颜色的嵌入式列表。XmlDataProvider 可用来绑定到 XML 文档或片断，该文档或片段既可以嵌入在 XmlDataProvider 标记中，也可以位于外部位置引用的文件中。

嵌入式 XML 内容必须置于 XmlDataProvider 内部的 <x:XData> 标记中，如图 4 所示。必须为 XmlDataProvider 提供 x:Key 值，以便数据绑定目标可对其进行引用。请注意，XPath 属性设置为“/colors”。此属性定义了将用作数据源的 XML 内容的级别。当绑定到可能包含在文件或数据库中的较大 XML 结构，且想要绑定的数据不是根元素时，这一属性会变得非常有用。

XmlDataProvider 是可放置到特定上下文资源内部的一种资源。如图 4 所示，已将 XmlDataProvider 定义为 StackPanel 上下文中的资源。这意味着 XmlDataProvider 将可用于该 StackPanel 内部的所有内容。设置资源的上下文有助于限制数据源向合适的区域公开。这使您可以在页面内分别为控件和支持资源创建定义明确的独立区域，从而提高可读性。

图 4 XmlDataProvider

```
<StackPanel>
<StackPanel.Resources>
<XmlDataProvider x:Key="MoreColors" XPath="/colors">
    <x:XData>
        <colors xmlns="">
            <color name="pink"/>
            <color name="white"/>
            <color name="black"/>
            <color name="cyan"/>
            <color name="gray"/>
            <color name="magenta"/>
        </colors>
```

```
        </x:XData>
    </XmlDataProvider>
```

绑定到资源的语法与绑定到元素的语法略有不同。绑定到控件时，可以设置绑定的 ElementName 和 Path 属性。但是绑定到资源时，需要设置 Source 属性，由于我们是绑定到 XmlDataProvider，所以还要设置绑定的 XPath 属性。例如，下列代码可将 ListBox 的项绑定到 MoreColors 资源。将 Source 属性设置为资源，并将其指定为名为 MoreColors 的 StaticResource。XPath 属性指示项会绑定到 XML 数据源中 <color> 元素的名称属性：

```
    <ListBox x:Name="lbColor" Width="248" Height="56"
        IsSynchronizedWithCurrentItem="True"
        ItemsSource="{Binding Source={StaticResource MoreColors},
        XPath=color/@name}">
    </ListBox>
```

我在本例中指定了 StaticResource，因为 XML 不会发生更改。如果数据源发生了更改，则不会将这些更改发送到目标。DynamicResource 设置则表示相反的情况，即会将数据源的更改发送到目标。当引用系统主题、全球化语言或字体时，该设置非常有用。DynamicResource 将允许这些类型的设置被传送到与其动态绑定的所有 UI 元素中。

XmlDataProvider 也可以指向 XML 内容的外部源。例如，我有一个名为 colors.xml 的文件，其中包含我希望 ListBox 绑定到的颜色列表。我只需要将第二个 XmlDataProvider 资源添加到 StackPanel，并将其引向 XML 文件即可。请注意，我将 Source 属性设置为了 XML 文件的名称，并将 x:Key 设置为 Colors：

```
    <XmlDataProvider x:Key="Colors" Source="Colors.xml" XPath="/colors"/>
```

两个 XmlDataProvider 在同一个 StackPanel 中都作为资源而存在。我可以使 ListBox 将其本身绑定到这个新的资源，方法是更改 StaticResource 设置的名称：

```
    <ListBox x:Name="lbColor" Width="248" Height="56"
        IsSynchronizedWithCurrentItem="True"
        ItemsSource="{Binding Source={StaticResource Colors},
        XPath=color/@name}">
    </ListBox>
```

对象绑定和 DataTemplates

虽然 XmlDataProvider 对 XML 非常有用，但是当您想绑定到对象或对象列表时，可以创建 ObjectDataProvider 作为资源。ObjectDataProvider 的 ObjectType 指定将提供数据绑定源的对象，而 MethodName 则指示为获得数据而需调用的方法。例如，假设我有一个名为 PersonService 的类，该类使用一种名为 GetPersonList 的方法来返回列表 <Person>，那么 ObjectDataProvider 可能会如下所示：

```
    <StackPanel.Resources>
    <ObjectDataProvider x:Key="persons"
    ObjectType="{x:Type svc:PersonService}"
    MethodName="GetPersonList"></ObjectDataProvider>
    </StackPanel.Resources>
```

如果您想进行更全面的了解，本栏随附的代码中还包含了 PersonService 和 Person 类以及其他示例代码。

ObjectDataProvider 还可以使用许多其他属性。ConstructionParameters 属性允许您将参数传递给要调用的类的构造函数。此外，可以使用 MethodParameters 属性来指定参数，同时还可以使用 ObjectInstance 属性来指定现有的对象实例作为源。

如果希望异步检索数据，可以将 ObjectDataProvider 的 IsAsynchronous 属性设为 true。这样，用户将可以在等待数据填充绑定到 ObjectDataProvider 的源的目标控件时与屏幕进行交互。

在添加 ObjectDataProvider 时，必须限定数据源类的命名空间。在本例中，我必须将 xmlns 属性添加到 <Window> 标记中，以便 svc 快捷方式符合要求，并指示正确的命名空间：

```
    xmlns:svc="clr-namespace:DataBindingWPF"
```

既然数据源已通过 ObjectDataProvider 定义，接着我想将 ListBox 控件中的项绑定到此数据。我想在每个 ListBoxItem 中显示两行文本。第一行将以粗体显示 Person 实例的 FullName 属性，第二行将显示该实例的 Title 和 City。在 XAML 中，通过使用 DataTemplate，这是很容易实现的，DataTemplate 允许您定义可重用的数据可视化策略。

图 5 显示了完整的 XAML，其中将 DataTemplate 定义为在我指定的布局中显示 Person 信息。我设置了 DataTemplate 的 DataType 属性，以指示 DataTemplate 将会引用 Person 类类型。我没有在 DataTemplate 中指定真正的绑定，因为我会在 ListBox 控件中指定。通过省略绑定源，可对作用域内的当前 DataContext 执行绑定。

图 5 对象绑定

```
<Window x:Class="DataBindingWPF.ObjectBinding"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:svc="clr-namespace:DataBindingWPF"
Title="DataBindingWPF" Height="300" Width="300">
<StackPanel>
    <StackPanel.Resources>
        <ObjectDataProvider x:Key="persons"
          ObjectType="{x:Type svc:PersonService}"
          MethodName="GetPersonList" ></ObjectDataProvider>
            <DataTemplate x:Key="personLayout" DataType="Person">
                <StackPanel Orientation="Vertical">
                    <TextBlock Text="{Binding Path=FullName}"
                            FontWeight="Bold" Foreground="Blue">
                    </TextBlock>
                <StackPanel Orientation="Horizontal">
                    <TextBlock Text="{Binding Path=Title}"></TextBlock>
                    <TextBlock Text=", "></TextBlock>
                    <TextBlock Text="{Binding Path=City}"></TextBlock>
                </StackPanel>
            </StackPanel>
        </DataTemplate>
    </StackPanel.Resources>
    <TextBlock></TextBlock>
    <ListBox x:Name="lbPersons"
            ItemsSource="{Binding Source={StaticResource persons}}"
            ItemTemplate="{DynamicResource personLayout}"
            IsSynchronizedWithCurrentItem="True"/>
</StackPanel>
</Window>
```

在图 5 中，我将 ListBox 的 ItemsSource 属性设置为绑定到人员资源，以便我可以将数据绑定到 ListBox，而不用对它进行格式化。通过将 ItemTemplate 属性设置为 PersonLayout 资源（即 DataTemplate 的键名），可以正确显示数据。最终结果的屏幕如图 6 所示。

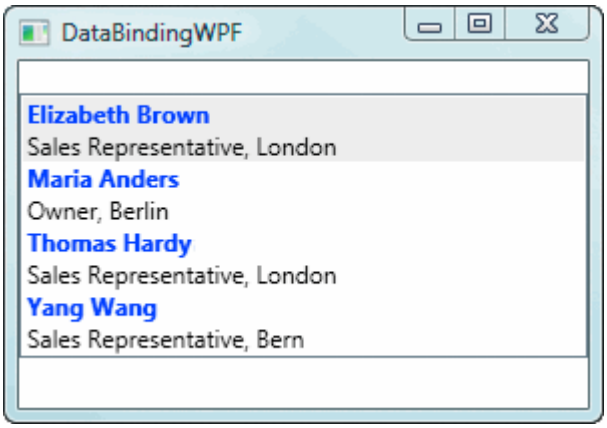


图 6 使用 DataTemplate

对数据进行排序

如果想以特定的方式对数据进行排序，可以绑定到 CollectionViewSource，而不是直接绑定到 ObjectDataProvider。CollectionViewSource 则会成为数据源，并充当截取 ObjectDataProvider 中的数据的媒介，并提供排序、分组和筛选功能，然后将它传送到目标。

接着显示的 CollectionViewSource 将其 Source 属性设置为 ObjectDataProvider（人员）的资源名称。然后我通过指示排序依据的属性及其方向定义了数据的排序顺序：

```
<CollectionViewSource x:Key="personView"
    Source="{Binding Source={StaticResource persons}}">
    <CollectionViewSource.SortDescriptions>
    <ComponentModel:SortDescription

    PropertyName="City"
    Direction="Ascending" />
    <ComponentModel:SortDescription
    PropertyName="FullName"
    Direction="Descending" />
    </CollectionViewSource.SortDescriptions>
</CollectionViewSource>
```

DataContext 可用来将容器控件内部的所有控件都绑定到数据源中。当您拥有多个控件，而且这些控件全部使用同一个绑定源时，这非常有用。如果为每个控件都指定了绑定源，那么代码可能会重复。相反，可以将容器控件的 DataContext 设置为绑定源，然后只需从所包含的控件中省略 Source 属性即可。例如，下面是一系列显式绑定到同一个绑定源的 TextBlock：

```
<StackPanel>
<TextBlock Text="{Binding Source={StaticResource personView},
    Path=FullName}"></TextBlock>
<TextBlock Text="{Binding Source={StaticResource personView},
    Path=Title}"></TextBlock>
<TextBlock Text="{Binding Source={StaticResource personView},
    Path=City}"></TextBlock>
</StackPanel>
```

以下是绑定到 DataContext 的三个相同的 TextBox，在此处，DataContext 反过来引用了该控件的 StackPanel 容器：

```
<StackPanel DataContext="{Binding Source={StaticResource personView}}" >
<TextBlock Text="{Binding Path=FullName}"></TextBlock>
<TextBlock Text="{Binding Path=Title}"></TextBlock>
<TextBlock Text="{Binding Path=City}"></TextBlock>
</StackPanel>
```

如果该容器没有定义 DataContext，那么它会继续查找下一个外部嵌套容器，直到它找到当前的 DataContext 为止。

欢迎试用和反馈

WPF 数据绑定具有很大的灵活性，并且能够控制可被绑定的数据类型以及它的控制和显示方式。有了这么多的功能和选择，我相信您一定跃跃欲试。

Chpater:

wpf 数据绑定，INotifyPropertyChanged vs DependencyProperty （—）
2008-06-05 12:21
忙了两天，做了这么两个示例，如果大家只想使用它，直接拷贝他们就好了，代码如下：】
DependencyProperty,用 vs2008 新建一个 wpf 项目，直接修改 cs 和 xaml 文件如下。
window1.xaml.cs:（修改或者添加代码如下）

```
private void windowMain_Loaded(object sender, RoutedEventArgs e)
    {
      lbText = 4;
    }

    public static readonly DependencyProperty CurrentIndexProperty = DependencyProperty.Register(
      "lbText", typeof(int), typeof(Window1));

    public int lbText
    {
      get
      {
        return (int)GetValue(CurrentIndexProperty);
      }
      set
      {
        SetValue(CurrentIndexProperty, value);
      }
    }

    private void button1_Click(object sender, RoutedEventArgs e)
    {
      lbText++;
    }
```

window1.xaml:

```
<window ...
xmlns:src="clr-namespace:WpfApplication25"
    Name="winMain"
    Loaded="windowMain_Loaded">
<Grid>
<Label Content="{Binding ElementName=winMain,Path=lbText}" Margin="48,15,73,0" x:Name="MyLabel" Height="58"
VerticalAlignment="Top" Background="Wheat" />
    <Button Margin="73,121,76,76" Name="button1" Click="button1_Click">Button</Button>
</Grid>
```

这样就好了，接下来看 INotifyPropertyChanged 的，同样新建一个 wpf 工程。
window1.xaml.cs:

```
private void Window_Loaded(object sender, RoutedEventArgs e)
      {
         UIShow show = (UIShow)FindResource("myUIshow");
         ++show.btnName;
         show.lbStatus += show.btnName.ToString();
      }

      private void button1_Click(object sender, RoutedEventArgs e)
      {
         UIShow show = (UIShow)FindResource("myUIshow");
         ++show.btnName;
         show.lbStatus += show.btnName.ToString();
      }

      private void button2_Click(object sender, RoutedEventArgs e)
      {
         TheTaskState = "dsfkasldf";
      }
   }

   public class UIShow : INotifyPropertyChanged
   {
      // INotifyPropertyChanged Members
      public event PropertyChangedEventHandler PropertyChanged;
      protected void Notify(string propName)
      {
         if (this.PropertyChanged != null)
         {
            PropertyChanged(this, new PropertyChangedEventArgs(propName));
         }
      }

      string lbstatus;
      public string lbStatus
      {
         get { return this.lbstatus; }
         set
         {
            if (this.lbstatus == value) { return; }
            this.lbstatus = value;
            Notify("lbStatus");
         }
      }

      int btnname;
      public int btnName
      {
         get { return this.btnname; }
         set
         {
            if (this.btnname == value) { return; }
            this.btnname = value;
            Notify("btnName");
         }
      }

      public UIShow() { }
      public UIShow(int name, string status)
      {
         this.lbstatus = status;
```

```
        this.btnname = name;
    }
```

window1.xaml:

```
<window ...
xmlns:local ="clr-namespace:WpfAppbinding2"
Loaded="Window_Loaded">
<Window.Resources>
    <local:UIShow x:Key="myUIshow" btnName="0" lbStatus="无聊状态"/>
  </Window.Resources>
<Grid Name="myGrid" DataContext="{ StaticResource myUIshow}">
    <TextBlock Height="31" HorizontalAlignment="Left" Margin="44,31,0,0" Name="textBlock1" VerticalAlignment="Top"
Width="79">Name:</TextBlock>
    <TextBlock Height="29" HorizontalAlignment="Left" Margin="42,85,0,0" Name="textBlock2" VerticalAlignment="Top"
Width="80" >Age:</TextBlock>
    <TextBox Height="31" Text="{Binding Path=lbStatus}" Margin="94,18,149,0" Name="textBox1"
VerticalAlignment="Top" />
    <TextBox Height="31" Text="{Binding Path=btnName}" Margin="83,72,160,0" Name="textBox2"
VerticalAlignment="Top" />
    <Button Margin="66,158,158,157" Name="button1" Click="button1_Click">Button</Button>
    <Button Height="55" HorizontalAlignment="Right" Margin="0,0,25,116" Name="button2" VerticalAlignment="Bottom"
Width="120" Click="button2_Click">Button</Button>
    <Label Height="48" HorizontalAlignment="Right" Margin="0,95,20,0" x:Name="label_TaskState" Content="{Binding
ElementName=windowMain,Path=TheTaskState}" VerticalAlignment="Top" Width="110"/>
  </Grid>
```

.如何与当前页面的元素(如需要获取一个窗体的属性,那么本元素则需要向上寻找)

```
<Window x:Class="WpfApplication.Window5"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Window1" Height="300" Width="150" xmlns:src="clr-namespace:WpfApplication" >

        <Button Content="{Binding Path=Width,RelativeSource={RelativeSource AncestorType={x:Type
src:Window5}}}" ></Button>

</Window>
```

Binding:

首先是绑定一个 xml 文件数据到 TreeView

xml 文件如下：

```
<Library>

 <Feed FeedID="msdn" FeedURL="http://msdn.microsoft.com/rss.xml"></Feed>

</Library>
```

xaml 内容：

```
<XmlDataProvider x:Key="rssResource" XPath="Library" Source="RssResource.xml"></XmlDataProvider>
```

```
<HierarchicalDataTemplate DataType="Feed">
   <TextBlock FontSize="12" Text="{Binding XPath=FeedID}" />
</HierarchicalDataTemplate>

<TreeView Grid.Column="0" Grid.Row="2" Grid.RowSpan="2"
          Width="200"
          Margin="5"
          Style="{StaticResource TreeViewStyle}"
          ItemsSource="{Binding Source={StaticResource rssResource},XPath=Feed}"
          Name="rssResourceTreeview"
          SelectedValuePath="FeedURL" SelectedItemChanged="rssResourceTreeview_SelectedItemChanged">
</TreeView>
```

运行，TreeView 为空，不显示任何东西。于是找原因，后来怀疑 xml 的 Attribute 不能解析为属性。修改 xml 如下：

```
<Library>
 <Feed>
  <FeedID>msdn</FeedID>
  <FeedURL>http://msdn.microsoft.com/rss.xml</FeedURL>
 </Feed>
 </Library>
```

ok，显示了 msdn 字样在里面。

第二个绑定一个 ObservableCollection<T>泛型集合到 ListBox

xaml 内容如下：

```
<Style x:Key="ListBoxStyle" TargetType="ListBox">
        <Setter Property="Template">
          <Setter.Value>
            <ControlTemplate TargetType="ListBox">
              <Border CornerRadius="6"
                      BorderBrush="{StaticResource BorderBrush}"
```

```
                    BorderThickness="1">
                  <ScrollViewer
                    ScrollViewer.VerticalScrollBarVisibility="Auto"
                    ScrollViewer.HorizontalScrollBarVisibility="Auto">
                  </ScrollViewer>
                </Border>
              </ControlTemplate>
            </Setter.Value>
          </Setter>
        </Style>

<DataTemplate x:Key="ServiceInfoItemTemplate">
          <Grid>
            <Grid.ColumnDefinitions>
              <ColumnDefinition Width="250" />
              <ColumnDefinition Width="100" />
              <ColumnDefinition Width="*"/>
            </Grid.ColumnDefinitions>
            <TextBlock Text="{Binding Path=Title}" Grid.Column="0" />
            <TextBlock Text="{Binding Path=PubDate}" Grid.Column="1" />
            <TextBlock Text="{Binding Path=Author}" Grid.Column="2" />
          </Grid>
        </DataTemplate>

 <ListBox Grid.Column="1" Grid.Row="2" Name="titleListBox"
            Height="200"
            Margin="5"
            Style="{StaticResource ListBoxStyle}"
            ItemsSource="{Binding}"
            ItemTemplate="{StaticResource ServiceInfoItemTemplate}"
            SelectedValuePath="Link"
```

/>

再次不显示数据，跟踪程序，发现 ListBox 的 DataContext 是有 99 条数据在里面的，于是怀疑 ItemTemplate 有错误，看半天不象有问题，再次怀疑 Style 不正确，去掉 Style 试试，数据可以显示出来，去看 Style，发现少了一个 <ItemsPresenter />标记，怪不得不显示数据，添加进去，运行，ok，显示了。

3.

我想绑定主窗体的大小，这样做的
xaml:

```
<Window ...
xmlns rc="clr-namespace:WpfApplication25"
      Name="winMain"
Left="{Binding ElementName=winMain,Path=winLeft}"
>
```

cs:

```
public static readonly DependencyProperty winleftProperty = DependencyProperty.Register(
        "winLeft", typeof(int), typeof(Window1));
    public int winLeft
    {
        get
        {
            return (int)GetValue(winleftProperty);
        }
        set
        {
            SetValue(winleftProperty, value);
        }
    }

    private void button1_Click(object sender, RoutedEventArgs e)
    {
        winLeft--;
    }
```

写了这个例子，想实现点击一下那个 button1，窗体就向左移动 1.

运行时候也可以实现，可是我发现一个问题，就是当我用其他方式移动窗体后，这个方法就不再起作用了。比如用鼠标拖着标题栏移动一下，然后再点击 button1，窗体就不会自动移动了。这个问题该怎么解决呢

默认的绑定方式是 BindingMode.Default, 对于 Window.Left, BindingMode.Default 应该是 BindingMode.OneWay. 就是你的数据源发生变化会更新绑定目标, 但反过来不会.
所以你直接改成 BindingMode.TwoWay 就好了, 可以这样改:

```
Code Snippet
Left="{Binding ElementName=winMain,Path=winLeft, Mode=TwoWay}"
```

Example:

学习 WPF: 创建数据绑定目录树

如果使用了 WPF 而不使用数据绑定(手工在界面和数据间进行同步),总会感觉不值.但是大部分讨论 WPF 数据绑定的文章,主题大多集中在 ListBox 这样平坦的数据集合上,讲如何绑定层次结构数据的比较少,这里我就通过一个简单的显示磁盘目录树的例子来展示如何完成这样的任务.

第一步,当然是定义要绑定的数据类型了.

在目录树这个例子中,每个 TreeViewItem 要显示的数据可以用 System.IO.DirectoryInfo 来表示,但是这样做有一个麻烦:DirectoryInfo 只能通过 GetDirectories()方法来获取子目录,但是 WPF 里的数据绑定则更倾向于使用属性在数据间导航,所以为了更方便地使用数据绑定,我们最好还是自定义一个类来完成这样的工作:

```
using System.Collections.Generic;
using System.IO;

namespace WpfApplication1
{

    class BindDirectory
    {
        public BindDirectory(string directoryPath)
        {
            //正规化目录路径,确保 Path 以'\\'结尾
            directoryPath = directoryPath.TrimEnd('\\');
            Path = directoryPath + '\\';

            //计算出目录名称(不包含路径)
            int indexLastSlash = directoryPath.LastIndexOf('\\');
            if (indexLastSlash >= 0)
            {
                Name = directoryPath.Substring(indexLastSlash + 1);
            }
            else
            {
                Name = directoryPath;
            }
        }

        public string Name
        {
            get;
            private set;
        }

        public string Path
        {
            get;
            private set;
```

```
    }

    public IEnumerable<BindDirectory> Directories
    {
      get
      {
        //延迟加载
        if (directories == null)
        {
          directories = new List<BindDirectory>();
          foreach (string d in Directory.GetDirectories(Path))
          {
            directories.Add(new BindDirectory(d));
          }
        }
        return directories;
      }

    }
    List<BindDirectory> directories;
  }
}
```

这个类所作的工作很简单,就是正规化目录路径,获取目录名称,以及延迟加载子目录(以提升性能)的列表,我们的界面也只要求它具有这些功能就行了.

第二步,创建一个数据提供类(DataProvider)

我们可以在 Window 的代码里设置界面的 DataContext,ItemsSource 等属性来让界面显示指定的数据,也可以构造一个专门提供数据的类,完全在界面(XAML)里指定,这里使用的是第二种方法:

```
using System.Collections.Generic;
using System.IO;

namespace WpfApplication1
{
  class BindDirectoryList：List<BindDirectory>
  {
    public BindDirectoryList()
    {
      foreach (var drive in DriveInfo.GetDrives())
      {
```

```
            Add(new BindDirectory(drive.RootDirectory.FullName));
        }
    }
}
```

这个类就更简单了,仅仅是在创建的时候加载所有的磁盘的根目录.

第三步,设计用户界面

只需要在 Window 中添加一个 TreeView,然后修改几行代码,就能轻松地显示我们的数据了:

```
<!--xml:sample 这一行用来引入我们自己代码的命名空间-->
<Window x:Class="WpfApplication1.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sample="clr-namespace:WpfApplication1"
    Title="Window1" Height="300" Width="300">
    <Window.Resources>
        <!--引入我们自己的数据提供对象-->
        <ObjectDataProvider x:Key="drives" ObjectType="{x:Type sample:BindDirectoryList}" />
        <!--设置如何显示数据,以及如何获取下一级数据的列表-->
        <HierarchicalDataTemplate x:Key="itemTemplate" DataType="{x:Type sample:BindDirectory}
" ItemsSource="{Binding Directories}">
            <TextBlock Text="{Binding Name}" />
        </HierarchicalDataTemplate>
    </Window.Resources>
    <TreeView ItemsSource="{Binding Source={StaticResource drives}}"
          ItemTemplate="{StaticResource itemTemplate}" >
    </TreeView>
</Window>
```

这里我们在 XAML 里定义了一个 drives 对象,它的类型为 BindDirectoryList,创建时会自动加载磁盘的根目录;
我们还定义了一个针对 BindDirectory 类型的层次型数据模板 itemsTemplate,指定了要获取此类型的数据的子数据需要通过 Directories 属性,并且告诉 WPF 用一个 TextBlock 来显示它的名称.
最后,我们设置一下 TreeView 的 ItemsSource 和 ItemTemplate 就完成工作了.

WPF 数据绑定(2) 绑定到 XML

```
<Window.Resources>

    <XmlDataProvider x:Key="XML">//自己手动写的

        <x:XData>
```

```
            <Colors>
                <Color>红</Color>
                <Color>黄</Color>
                <Color>蓝</Color>
            </Colors>
        </x:XData>
    </XmlDataProvider>
    <XmlDataProvider x:Key="ttxml" Source="tt.xml">//绑定到外部XML文件
    </XmlDataProvider>
</Window.Resources>
<StackPanel>
    <ListBox Height="69" ItemsSource="{Binding Source={StaticResource XML},XPath=/Colors/Color}"
Width="191" />
    <ListBox Height="100" Margin="10" ItemsSource="{Binding Source={StaticResource
ttxml},XPath=/Data/ff}"/>
</StackPanel>
```

WPF数据绑定(1) 绑定到控件

```
<StackPanel HorizontalAlignment="Left" Height="274" Width="286">
    <TextBox Width="100" Name="TextBox1" Text="1111" Height="30"/>

    <TextBox Width="100" Height="30" Margin="10" Text="{Binding
ElementName=TextBox1,Path=Text,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}"/>
</StackPanel>
```

WPF常见问题—>

```
<Border Background="LightGray" CornerRadius="40" Padding="10"  Height="140" HorizontalAlignment="Center"
VerticalAlignment="center" BorderBrush="Black" BorderThickness="1">
            <TextBlock VerticalAlignment="center">
                <Hyperlink NavigateUri="http://www.baidu.com/">BrawDraw.Com Online</Hyperlink>
            </TextBlock>
        </Border>
```

1. 如何设置链接?

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

```
chemas.microsoft.com/winfx/2006/xaml" >
    <Border Background="LightGray" CornerRadius="40" Padding="10"  Height="140" Horizontalalignment="Center" VerticalAlignment="center" BorderBrush="Black" BorderThickness="1">
        <TextBlock VerticalAlignment="center"><Hyperlink NavigateUri="http://www.zpxp.com/">BrawDraw.Com Online</Hyperlink></TextBlock>
    </Border>
</Grid>
```

(1)注意这句:<Hyperlink NavigateUri="http://www.zpxp.com/">BrawDraw.Com Online</Hyperlink>

2. 如何画圆角矩形?

注意上面(1)Border 标记中的 CornerRadius="40",它用来指定圆角矩形的圆角半径

3. 怎样象 HTML 中的 CSS 一样设置样式?

使用 Style - Setter 方式:

```
<Style TargetType="{x:Type Rectangle}">
 <Setter Property="Rectangle.RadiusX" Value="10"/>
 <Setter Property="Rectangle.RadiusY" Value="10"/>
</Style>
```

TargetType 目标类型,Setter 进行对 Property 属性赋于 Value 指定的值. [关于此问题,可以参见我的另一篇文章:<使用 WPF 创建炫亮按钮> http://blog.csdn.net/johnsuna/archive/2007/08/07/1729039.aspx 第 3 点:使用 Application.Resources 设置按钮属性（类似 CSS 样式单）]

4. 排列类似表格或单元格之类的使用什么?

使用 Grid 标记,比如排日历表

5. 如何画直线?

类似:<Rectangle Fill="Black" RadiusX="0" RadiusY="0" Height="1" Margin="0,20,0,0"/>

6. 如何使用底层 API 进行图形图像绘制而不是 XAML?

首先，由于 WPF 中不象 GDI+中有 Graphics 对象,因此你无法使用 Graphics 进行绘图了，取而代之的是：DrawingContext；类似的，GDI+中的 OnPaint 已被 OnRender 取代。其次，UIElement 有一个 OnRendar 方法，它的定义是： protected virtual void OnRender (DrawingContext drawingContext)

但我们不能直接调用 OnRender 方法，也不能直接创建 DrawingContext 实例，但可以利用 DrawingGroup.Open 和 DrawingVisual.RenderOpen。

这里举两个例子：（1）自定义绘制 Canvas:(2)保存图片到文件。更多详情见这篇文章："WPF 中，如何使用图像 API 进行图形图像绘制而不是 XAML? "

7.**如何在 WPF 中嵌入 Flash(ActiveX):** http://blogs.msdn.com/jijia/archive/2007/06/07/wpf-flash-activex.aspx

8. **在 WPF 下如何获取显示屏幕的高度和宽度，就像 winform 下的 System.Windows.Forms.Screen.PrimaryScreen.Bounds.Width（Height）一样？**

double h = SystemParameters.PrimaryScreenHeight; double w = SystemParameters.PrimaryScreenWidth;

以下是 XAML 代码,它将屏幕宽度绑定到 Button 的宽度： <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.PrimaryScreenWidth}"> SystemParameters </Button>

9. 怎样制作一个带三角形的且有文字的自定义控件？就象下图一样： 

很简单，你只需要将 TextBlock，Polygon 用<StackPanel>组合之后放入 Button 标签内即可！ <StackPanel> <Button Height="50" Width="50"> <StackPanel> <TextBlock>Play</TextBlock> <Polygon Points="0,0 0,26 17,13" Fill="Black" /> </StackPanel> </Button> </StackPanel> 够简单的吧？

10. WPF 内如何加载.net1.1 或 2.0 的控件? 你可以这样加载.net1.1 或 2.0 的控件到 WPF 的 XAML 中: <Window x:Class="HostingWfInWpf.Window1" xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:wf="clr-namespa

ce:System.Windows.Forms;assembly=System.Windows.Forms" Title="HostingWfInWpf" > <Grid> <WindowsFormsHost> <wf:MaskedTextBox x:Name="mtbDate" Mask="00/00/0000"/> </WindowsFormsHost> </Grid> </Window> 同理，你也可以加载 Forms 中你的自定义控件： <WindowsFormsHost> <wf:BrawDrawGraphControl x:Name="brawdrawControl1" Width="500" Height="300" /> </WindowsFormsHost> 所不同的是，你需要在 xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms" 将 clr-namespace 和 assembly 设置为相应的值。

如果是 C#代码呢?将上面的 XAML 转成相应的 C#代码即可。别告诉我你不会哟，如果不会，你就需要加深 WPF 基础知识的学习。加油！

11. 如何防止 WPF 控件的遮盖而无法解发相关事件？ 在 WPF 中，控件是可以树型嵌套的，例如: <Grid Name =" mainGrid" > <Canvas Name ="mainCanvas" Background="Green"/> </Grid> 这时如果 mainGrid 容器里有个 Button, 因为 mainCanvas 的遮盖而无法触发 Click 事件（即使 mainCanvas 背景为透明也不能触发）。此时，你需要将背景色设为 null，这样就可以穿透 mainCanvas 容器的遮盖了。也有一个前提条件，那就是 mainCanvas 中没有控件遮盖 mainGrid 的 Button。

12. WPF 与 GDI+中的颜色有区别吗？ 你可以参见我的这篇文章。《GDI+与 WPF 中的颜色简析》http://blog.csdn.net/johnsuna/archive/2007/08/27/1761061.aspx

13. WPF 中 InkCanvas(墨水面板)用法？ 参见周银辉的这篇文章 http://www.cnblogs.com/zhouyinhui/archive/2007/08/08/841569.html，赞一下，写得不错！

14. 控制 WPF 图片的缩放绘图时的质量? 使用 BitmapScalingMode 枚举: HighQuality：高质量的缩放，比 Bilinear 缩放方式要慢。 LowQuality：使用 Bilinear 缩放方式,输出的图像质量相对较低。 Unspecified：使用默认的缩放方式。

15. WINFORM 中如何加入 FLASH？又如何在 WPF 中插入 FLASH？ (1)启动 vs2005, 新建一个 C#应用程序 (2)在工具箱里添加一个 flash 组件,flash.ocx，它是 flash 的容器，把它拖到 Form 的内容区，并将之改名为 mainFlash (3)双击 Form 窗体加入以下代码： string swfFileName = @"test.swf";//这里你需要根据你的 SWF 位置进行调整 private void MainForm_Load(object sender, System.EventArgs e) { swfFileName = Application.StartupPath + swfFileName; //根据需要进行调整 mainFlash.LoadMovie(0, swfFileName); }

如果你 FLASH 中有什么事件处理（比如点击按钮），那么你可以再找到 mainFlash 的事件 FSCommand，加入下面的代码： private void mainFlash_FSCommand(object sender, AxShockwaveFlashObjects._IShockwaveFlashEvents_FSCommandEvent e) { MessageBox.Show("Command:" + e.command.ToString() + "\r\n" + "Args:" + e.args .ToString()); } 按 F5 或 Ctrl+F5 运行程序，我们会发现 FLASH 已正常运行。如有上述

之点击按钮，你可以点击 flash 上的按钮，第一行显示 FLASH 的命令，第二行显示参数。 **如何在 WPF 中插入 FLASH？** 见这里：http://blogs.msdn.com/jijia/archive/2007/06/07/wpf-flash-activex.aspx

16. Creating Images from Raw Pixel Data in WPF

```
double dpi = 96;
int width = 128;
int height = 128;
byte[] pixelData = new byte[width*height];
for (int y = 0; y < height; ++y)
{
    int yIndex = y * width;
    for (int x = 0; x < width; ++x)
    {
        pixelData[x + yIndex] = (byte) (x + y);
    }
}
BitmapSource bmpSource = BitmapSource.Create(width, height, dpi, dpi,
    PixelFormats.Gray8, null, pixelData, width);
```