WPF 学习笔记

目录

WPF 学习笔记 1
- Application 1
- Dispatcher 3
- Navigation 5
- XAML 9
- DependencyProperty 15
- RoutedEvent 20
- Resource 24
- Binding 31

Silverlight - Hello, World! 68

Application

和 WinForm 类似，WPF 同样需要一个 Application 来统领一些全局的行为和操作，并且每个 Domain 中只能有一个 Application 实例存在。和 WinForm 不同的是 WPF Application 默认由两部分组成：App.xaml 和 App.xaml.cs，这有点类似于 Delphi Form，将定义和行为代码相分离。当然，WebForm 也采用了类似的方式。XAML 从严格意义上说并不是一个纯粹的 XML 格式文件，它更像是一种 DSL，它的所有定义都直接映射成某些代码，只不过具体的翻译工作由编译器完成而已。

下面是一个简单的 App 定义。

```
public partial class App : Application
{
}
```

当你在自动生成的 Project 代码中看到 **paritial** 时，应该下意识去找找 "This code was generated by a tool." …… 不过这次自动生成的代码存放位置更加古怪 —— **obj\Debug\App.g.cs**。

```
public partial class App : System.Windows.Application
{
  [DebuggerNonUserCode]
  public void InitializeComponent()
  {
    this.StartupUri = new System.Uri("Window1.xaml", System.UriKind.Relative);

  }
```

```
  [STAThread]
  [DebuggerNonUserCode]
  public static void Main()
  {
    App app = new App();
    app.InitializeComponent();
    app.Run();
  }
}
```

App.StartupUri 用于设置 MainWindow，App.Run() 启动消息循环。当然，还有那个 STAThread，这意味着 WPF 依旧使用一个 UI Thread 来处理 UI Message。

我们完全可以舍弃自动生成的代码，自己手工写一个 App。

```
public class App : Application
{
  [STAThread]
  private static void Main()
  {
    var app = new App();
    var window = new Window { Title = "WPF" };

    app.Run(window);
  }
}
```

Application 提供了一些实用的属性和方法。

Current: 获取 Domain 中默认的 Application 实例。
MainWindow: 获取主窗口实例。
Windows: 获取所有被实例化的 Window 实例。
ShutdownMode: 指定 Application.Shutdown 方式，包括主窗体关闭，最后一个窗口关闭，以及手工调用 Shutdown()。
Properties: 一个线程安全的全局字典，可用来存储一个公共信息。

Shutdown: 该方法终止 Application Process，可向操作系统返回一个退出码。

我们依然可以使用 Mutex 来阻止运行多个实例。

```
private void Application_Startup(object sender, StartupEventArgs e)
{
  var createdNew = false;
  var name = Assembly.GetEntryAssembly().FullName;
  new Mutex(true, name, out createdNew);
```

```
  if (!createdNew)
  {
    MessageBox.Show("There is already an instance running, Exit!");
    Application.Current.Shutdown();
  }
}
```

当然也可以用 Windows 属性判断窗体是否已经存在。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  var window2 = Application.Current.Windows.OfType<Window>().FirstOrDefault(w
=> w is Window2);
  if (window2 == null) window2 = new Window2();
  window2.Show();
  window2.Activate();
}
```

----------无聊的分割线----------

迟到的笔记总算开始了，算是为 Silverlight 做准备吧。

Dispatcher

WPF 使用一个专用的 UI 线程来完成界面的操作和更新，这个线程会关联一个唯一的 **Dispatcher** 对象，用于调度按优先顺序排列的工作项队列。Application.Run() 实际上就是对 Dispatcher.Run() 的间接调用。

Dispatcher 通过循环来处理工作项队列，这个循环通常被成为 "帧 (DispatcherFrame)"。Dispatcher.Run() 创建并启动这个帧，这也是 Application.Run() 启动消息循环的最终途径。

```
public sealed class Dispatcher
{
  [SecurityCritical, UIPermission(SecurityAction.LinkDemand,
Unrestricted=true)]
  public static void Run()
  {
    PushFrame(new DispatcherFrame());
  }
}
```

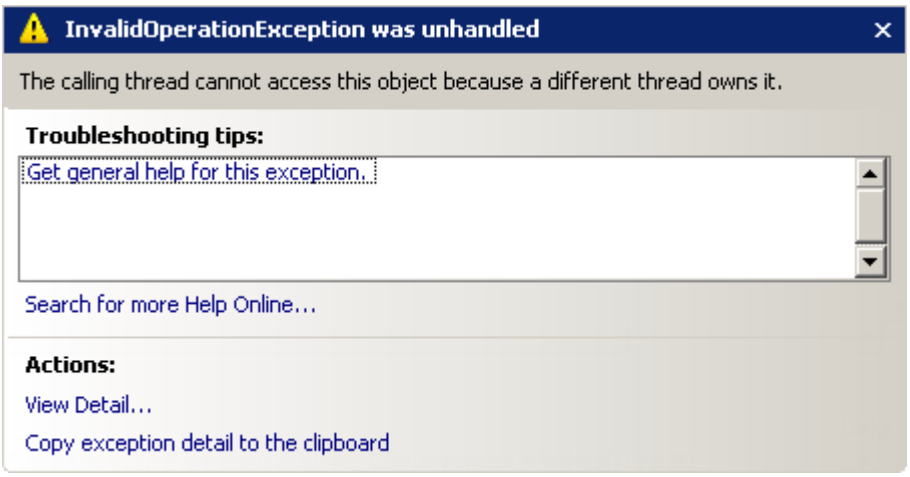
DispatcherFrame 可以嵌套，并通过检查 Continue 属性来决定循环是否继续。我们可以通过调用 Dispatcher.ExitAllFrames() 来终止所有的帧循环，当然这种编程方式并不可取，可能会造成一些意外出现。

与 Dispatcher 调度对象想对应的就是 **DispatcherObject**，在 WPF 中绝大部分控件都继承自

DispatcherObject，甚至包括 Application。这些继承自 DispatcherObject 的对象具有线程关联特征，也就意味着只有创建这些对象实例，且包含了 Dispatcher 的线程(通常指默认 UI 线程)才能直接对其进行更新操作。

当我们尝试从一个非 UI 线程更新一个标签，会看到一个如下的异常。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  new Thread(() => this.label1.Content = DateTime.Now.ToString()).Start();
}
```



按照 DispatcherObject 的限制原则，我们改用 Window.Dispatcher.Invoke() 即可顺利完成这个更新操作。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  new Thread(() =>
  {
    this.Dispatcher.Invoke(DispatcherPriority.Normal,
      new Action(() => this.label1.Content = DateTime.Now.ToString()));
  }).Start();
}
```

如果在其他项目(比如类库)中，我们可以用 Application.Current.Dispatcher.Invoke(...) 完成同样的操作，它们都指向 UI Thread Dispatcher 这个唯一对象。

Dispatcher 还提供了 BeginInvoke 这个异步版本。

```
private void button1_Click(object sender, RoutedEventArgs e)
{
  new Thread(() =>
  {
    Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Normal,
      new Action(() =>
```

```
      {
        Thread.Sleep(3000);
        this.label1.Content = DateTime.Now.ToString();
      }));

    MessageBox.Show("Hi!");
  }).Start();
}
```

凡事都有例外，WPF 还提供了一种继承自 Freezable 的类型，尽管 Freezable 也间接继承自 DispatcherObject，但当这类对象从修改状态变成冻结状态时，它即变成自由线程对象，不在具有线程关联。(有关 Freezable 详情可参考 MSDN)

Navigation

互联网的兴起，造就和培养了一种新的用户交互界面 —— Page & Navigation。无论是前进、后退还是页面，都完全是一个全新的门类，不同于以往的 SDI/MDI。WPF 或者是它的简化版 Silverlight 都不可避免地遵从了这种改良的 B/S 模式，使用 URI 来串接 UI 流程。

NavigationService、Page、Hyperlink、Journal(日志/历史记录) 是 WPF 整个导航体系的核心。NavigationService 提供了类似 IE Host 的控制环境，Journal 可以记录和恢复相关 Page 的状态，我们通常会选用的宿主方式包括：Browser(XBAP) 和 NavigationWindow。

1. NavigationWindow

NavigationWindow 继承自 Window，不知什么原因，我并没有在 VS2008 "New Item..." 中找到相关的条目，只好自己动手将一个 Window 改成 NavigationWindow。

Window1.xaml

```
<NavigationWindow x:Class="Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="300" Width="300"
WindowStartupLocation="CenterScreen"
  Source="Page1.xaml">
</NavigationWindow>
```

Source 属性指定了该窗口的默认页面，当然，我们还要修改一下 Window1.xaml.cs 里的基类。

```
public partial class Window1 : NavigationWindow
{
  public Window1()
  {
    InitializeComponent();
```

```
  }
}
```

创建一个 Page1.xaml，我们就可以像普通 Window 那样添加相关的控件和操作。

2. Hyperlink

超链接应该是我们最熟悉的一种导航方式。

Page1.xaml

```
<Page x:Class="Learn.WPF.Page1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Page1">
  <Grid>
    <TextBlock>
      <Hyperlink NavigateUri="Page2.xaml">Page2</Hyperlink>
    </TextBlock>
  </Grid>
</Page>
```

NavigateUri 相当于 "Html a.href"，当然我们也可以使用 Hyperlink.Click 事件，然后使用 NavigationService 来完成导航操作。

Page1.xaml

```
<Hyperlink Click="Hyperlink_Click">Page2</Hyperlink>
```

Page1.xaml.cs

```
private void Hyperlink_Click(object sender, RoutedEventArgs e)
{
  this.NavigationService.Navigate(new Uri("Page2.xaml", UriKind.Relative));
}
```

Hyperlink 还支持 "test.htm#name" 这样的导航定位方式，滚动页面直到某个特定名称的控件被显示。Hyperlink 的另外一个实用属性是 Command，我们可以使用 NavigationCommands 中创建的一系列静态成员来执行一些常用操作。

```
<Hyperlink Command="NavigationCommands.Refresh">Refresh</Hyperlink>
<Hyperlink Command="NavigationCommands.BrowseBack">BrowseBack</Hyperlink>
<Hyperlink
Command="NavigationCommands.BrowseForward">BrowseForward</Hyperlink>
```

3. NavigationService

很多时候我们都需要使用 NavigationService 代替 Hyperlink.NavigateUri，比如非默认构造的 Page，动态

确定目标页面等等。我们可以使用 Page.NavigationService 或者 NavigationService.GetNavigationService() 获得 NavigationService 的实例引用 (别忘了添加 using System.Windows.Navigation)。

```
public partial class Page1 : Page
{
  private void Hyperlink_Click(object sender, RoutedEventArgs e)
  {
    var page2 = new Page2();
    page2.label1.Content = "Beijing 2008!";

    this.NavigationService.Navigate(page2);
  }
}
```

除了 Navigate()，还可以使用 NavigationService 的两个属性完成导航切换操作。

```
//this.NavigationService.Content = page2;
this.NavigationService.Source = new Uri("Page2.xaml", UriKind.Relative);
```

NavigationService 提供了大量的方法和时间来管理相关导航操作。

日志: AddBackEntry、RemoveBackEntry。
载入: Navigate、Refresh、StopLoading。
切换: GoBack、GoForward。
事件: Navigating(新导航请求时触发，可取消导航)……

我们也可以使用 Application 的相关事件来处理导航过程。

4. Journal

Journal 相当于 WebBrowser.History，它包含两个数据栈用来记录前进和后退页面的显示状态，每个相关 Page 都会对应一个 JournalEntry。日志状态自动恢复仅对单击导航条上前进后退按钮有效。

5. Page

有关 Page 本身的使用并不是本文的内容，我们此处关心的是它在导航过程中的生命周期。在 WPF 中，Page 注定是个短命鬼，无论我们使用导航还是后退按钮都会重新创建 Page 对象实例，然后可能是日志对其恢复显示状态。也就是说日志只是记录了 Page 相关控件的状态数据，而不是 Page 对象引用(默认情况下)。

有两种方式来维持一个 Page 引用。第一种就是我们自己维持一个 Page 引用，比如使用某个类似 Application.Properties 这样的容器。

```
private void Hyperlink_Click(object sender, RoutedEventArgs e)
{
  var page2 = Application.Current.Properties["page2"] as Page2;
  if (page2 == null)
```

```
  {
    page2 = new Page2();
    page2.label1.Content = DateTime.Now.ToString();

    Application.Current.Properties["page2"] = page2;
  }

  this.NavigationService.Navigate(page2);
  //this.NavigationService.Content = page2;
}
```

另外一种就是设置 Page.KeepAlive 属性，这样一来日志会记录该 Page 的引用，当我们使用前进后退按钮时，将不会再次创建该 Page 的对象实例。

```
<Page x:Class="Learn.WPF.Page2"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Page2" Loaded="Page_Loaded"
  KeepAlive="True">

</page>
```

有一点需要注意：该方法仅对前进后退等日志操作有效。如果我们使用 HyperLink.NavigateUri 或 NavigationService.Navigate() 导航时依旧会生成新的页面实例，并可能代替日志中最后一个同类型的对象引用记录。另外，当多个页面存在循环链接时，会导致多个页面实例被日志记录，造成一定的内存浪费。

6. Frame

Frame 的作用和 HTML 中的 IFrame 类似，我们可以用它在一个普通的 Window 或 Page 中嵌套显示其他的 Page。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="300" Width="300">
  <Grid>
    <Frame Source="Page1.xaml"></Frame>
  </Grid>
</Window>
```

默认情况下，Frame 会尝试使用上层页面(Page)或窗体(NavigationWindow)的日志，当然我们也可以使用 JournalOwnership 属性强行让 Frame 使用自己的日志导航。

```
<Frame Source="Page1.xaml" JournalOwnership="OwnsJournal"></Frame>
```

Frame 的另外一个作用就是可以导航到 HTML 页面，我们可以把它当作一个嵌入式 IE WebBrowser 来使用。

```
<Frame Source="http://www.rainsts.net" />
```

7. PageFunction<T>

WPF 提供了一个称之为 PageFunction 的 Page 继承类来实现类似 HTML showModal 的功能。我们可以用它来收集某些数据并返回给调用页，当然这个封装其实非常简单，我们完全可以自己实现，无非是提供一个类似 OnReturn 的方法实现而已。泛型参数 T 表示返回数据类型。

Page1.xaml.cs

```
public partial class Page1 : Page
{
  private void Hyperlink_Click(object sender, RoutedEventArgs e)
  {
    var modal = new PageFunction1();
    modal.Return += (s, ex) => this.label1.Content = ex.Result.ToString();
    this.NavigationService.Navigate(modal);
  }
}
```

PageFunction1.xaml.cs

```
public partial class PageFunction1 : PageFunction<int>
{
  private void button1_Click(object sender, RoutedEventArgs e)
  {
    OnReturn(new ReturnEventArgs<int>(DateTime.Now.Millisecond));
  }
}
```

使用步骤:

(1) 创建 PageFunction<T> 对象实例，当然我们可以使用含参构造传递额外的数据；

(2) 调用 PageFunction<T>.OnReturn() 方法用来返回一个特定的结果包装对象 —— ReturnEventArgs<T>；

(3) 调用者通过订阅 PageFunction<T>.Return 事件获取这个返回结果。

MSDN 中还提到用 OnReturn(null) 来表示 Cancel，  ~~~~ 说实话，个人觉得这个 PageFunction 从命名到执行逻辑都有点别扭，难道仅仅是因为 Page 特殊的实例构造逻辑？我们也可以使用 Application.Properties + Page 来实现一个非关联耦合的 showModel 逻辑，只不过不那么 "标准" 罢了。

有一点需要提醒一下：我们应该及时解除对 FunctionPage<T>.Return 的订阅，我上面的例子和 MSDN 一样偷懒了。

XAML

Microsoft 将 XAML 定义为 "简单"、"通用"、"声明式" 的 "编程语言"。这意味着我们会在更多的地方看到它(比如 Silverlight)，而且它显然比其原始版本 XML (XAML 是一种基于 XML 且遵循 XML 结构规则的语言) 多了更多的逻辑处理手段。如果愿意的话，我们完全可以抛开 XAML 来编写 WPF 程序。只不过这有点类似用记事本开发 .NET 程序的意味，好玩不好用。XAML 的定义模式使得非编程人员可以用 "易懂" 的方式来刻画 UI，并且这种方式我们早已熟悉，比如 WebForm，亦或者是我一直念念不忘的 Delphi Form (偶尔想起而已，其实早将 Object Pascal 忘得精光了)。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
  </Grid>
</Window>
```

这是一个非常简单的 XAML，它定义了一个空白 WPF 窗体(Window)。XAML 对应于 .NET 代码，只不过这个过程由特定的 XAML 编译器和运行时解释器完成。当解释器处理上面这段代码时，相当于：

```
new Window1 { Title = "Window1" };
```

从这里我们可以体会两者的区别，用 XAML 的好处是可以在设计阶段就能看到最终的展现效果，很显然这是美工所需要的。你可以从 VS 命令行输入 "xamlpad.exe"，这样你会看到直观的效果。

作为一种应用于 .NET 平台的 "语言"，XAML 同样支持很多我们所熟悉也是必须的概念。

1. Namespace

XAML 默认将下列 .NET Namespace 映射到 "http://schemas.microsoft.com/winfx/2006/xaml/presentation"：

System.Windows
System.Windows.Automation
System.Windows.Controls
System.Windows.Controls.Primitives
System.Windows.Data
System.Windows.Documents
System.Windows.Forms.Integration
System.Windows.Ink
System.Windows.Input
System.Windows.Media
System.Windows.Media.Animation
System.Windows.Media.Effects
System.Windows.Media.Imaging
System.Windows.Media.Media3D

System.Windows.Media.TextFormatting
System.Windows.Navigation
System.Windows.Shapes

除了这个包含绝大多数 WPF 所需类型的主要命名空间外，还有一个是 XAML 专用的命名空间 (System.Windows.Markup) —— "http://schemas.microsoft.com/winfx/2006/xaml"。使用非默认命名空间的语法有点类似于 C# Namespace Alias， 我们需要添加一个前缀，比如下面示例中的 "x"。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <TextBox x:Name="txtUsername" Background="{x:Null}"></TextBox>
  </Grid>
</Window>
```

我们还可以引入 CLR Namespace。

```
<collections:Hashtable
  xmlns:collections="clr-namespace:System.Collections;assembly=mscorlib"
  xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <sys:Int32 x:Key="key1">1</sys:Int32>
</collections:Hashtable>
```

2. Property

我们可以用下面两种方式来设置 XAML 元素的属性。

方式 1

```
<Label Name="label10" Foreground="Red">Label10</Label>
```

方式 2

```
<Label Name="label11">
  <Label.Content>
    Label11
  </Label.Content>
  <Label.Foreground>
    Blue
  </Label.Foreground>
</Label>
```

WPF 会按下列顺序将 XAML 中的属性字符串转换为实际属性值。

(1) 属性值以大括号开始，或者属性是从 MarkupExtension 派生的元素，则使用标记扩展处理。
(2) 属性用指定的 TypeConverter 声明的，或者使用了转换特性(TypeConverterAttribute)，则提交到类型

转换器。
(3) 尝试基元类型转换，包括枚举名称检查。

```
<Trigger Property="Visibility" Value="Collapsed,Hidden">
  <Setter ... />
</Trigger>
```

3. TypeConverter

WPF 提供了大量的类型转换器，以便将类似下面示例中的 Red 字符串转换城 SystemWindows.Media.Brushes.Red。

```
<Label Name="label10" Foreground="Red"></Label>
```

等价于

```
this.label10.Foreground = System.Windows.Media.Brushes.Red;
```

不过下面的代码更能反应运行期的实际转换行为

```
var typeConverter =
System.ComponentModel.TypeDescriptor.GetConverter(typeof(Brush));
this.label10.Foreground =
(Brush)typeConverter.ConvertFromInvariantString("Red");
```

有关转换器列表，可参考：
ms-help://MS.MSDNQTR.v90.chs/fxref_system/html/35bffd5f-b9aa-1ccd-99fe-b0833551e562.htm

4. MarkupExtension

对 XAML 的一种扩展，以便支持复杂的属性值。这些标记扩展通常继承自 MarkupExtension，并使用大括号包含。WPF 提供了一些常用的标记扩展，诸如 NullExtension、StaticExtension、DynamicResourceExtension、StaticResourceExtension、Binding 等。和 Attribute 规则类似，我们通常可以省略 Extension 这个后缀。需要注意的是某些标记扩展属于 System.Windows.Markup，因此我们需要添加命名空间前缀。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <TextBox Background="{x:Null}"></TextBox>
  </Grid>
</Window>
```

我们可以为标记扩展提供其所需的构造参数。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
  Title="Window1">
  <Grid>
    <Label Content="{x:Static SystemParameters.IconHeight}" />
  </Grid>
</Window>
```

这个例子中，我们将 System.Windows.SystemParameters.IconHeight 值作为参数传递给 "public StaticExtension(string member)" 构造方法，这种参数通常被称作定位参数。而另外一种参数是将特定的值传给标记扩展对象属性，语法上必须指定属性名称，故被称之为命名参数。下面的例子表示将 textBox1.Text 参数绑定到 Label.Content 上，这样当编辑框内容发生变化时，标签内容自动保持同步。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <TextBox Name="textBox1" />
    <Label Content="{Binding ElementName=textBox1, Path=Text}" />
 </Grid>
</Window>
```

标记扩展允许嵌套，并可以引用自身。我们看另外一个例子。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <TextBox Name="textBox2" Width="128" Text="{Binding
RelativeSource={RelativeSource Self}, Path=Width}" />
  </Grid>
</Window>
```

这个例子的意思是将 TextBox.Text 内容绑定为其自身(Self)的高度值(Width)。

标记扩展带来一个问题就是大括号的转义，毕竟很多时候我们需要在内容显示中使用它。解决方法是在前面添加一对额外的大括号。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <Label Content="{}{Hello, World!}" />
  </Grid>
</Window>
```

如果觉得难看，也可以写成下面这样。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <Label>{Hello, World!}</Label>
  </Grid>
</Window>
```

5. Content

XAML 这点和 HTML 非常类似，我们可以将任何内容添加到元素内容项中，这带来更加丰富的 UI 表达能力，再也不像 WinForm 那样 "能做什么，不能做什么"。

```
<Button>
  <Hyperlink>Click</Hyperlink>
</Button>
```

有一点需要注意，内容项并不一定就是 Content。像 ComboBox、ListBox、TabControl 使用 Items 作为内容项。

6. XamlReader & XamlWriter

通常情况下，XAML 在项目编译时会被压缩成 BAML (Binary Application Markup Language) 保存到资源文件中。BAML 只是包含 XAML 的纯格式声明，并没有任何事件之类的执行代码，切记不要和 MSIL 相混淆。XAML 运行期解释器解读 BAML 并生成相应的元素对象。

System.Windows.Markup 命名空间中提供了 XamlReader、XamlWriter 两个类型，允许我们手工操控 XAML 文件。

```
var window = (Window)XamlReader.Parse("<Window
xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"></Window
>");
window.ShowDialog();
```

当然，我们还可以从文件流中读取。

```
using (var stream = new FileStream(@"test.xaml", FileMode.Open))
{
  var window = (Window)XamlReader.Load(stream);

  var button = (Button)window.FindName("btnOK");
  button.Click += (s, ex) => MessageBox.Show("Hello, World!");
```

```
  window.ShowDialog();
}
```

test.xaml

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window2" Height="300" Width="300">
  <Grid>
    <Button x:Name="btnOK">OK</Button>
  </Grid>
</Window>
```

需要注意的是 XamlReader 载入的 XAML 代码不能包含任何类型(x:Class)以及事件代码(x:Code)。

我们可以用 XamlWriter 将一个编译的 BAML 还原成 XAML。

```
var xaml = XamlWriter.Save(new Window2());
MessageBox.Show(xaml);
```

输出:

```
<Window2
  Title="Window2" Width="300" Height="300"
  xmlns="clr-namespace:Learn.WPF;assembly=Learn.WPF"
  xmlns:av="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <av:Grid>
    <av:Button Name="btnOK">OK</av:Button>
  </av:Grid>
</Window2>
```

XAML 的动态载入在使用动态皮肤场景时非常有用，现在只要了解一下即可。

DependencyProperty

依赖属性并不是一种语言层面的 "属性"，而是一种 WPF 提供的 "功能"。它在 CLR Property 的基础上封装了一些内在的行为，使得基于声明式的 XAML 具备更强大的动作操控能力，很显然这比使用程序设计代码编写行为事件要简便和自然得多。

依赖属性的特点：

(1) 使用高效的稀疏存储系统，这意味着在不设置本地值的情况下，所有同类型对象的依赖属性都将共享默认设置，大大节约内存开销。
(2) 依赖属性具备变更通知(Change Notification)能力，当属性值发生变化时，可以通过预先注册的元数据信息触发联动行为。
(3) 依赖属性可以从其在树中的父级继承属性值。

(4) 依赖属性可以依据优先级从多个提供程序中获取最终值。

1. 依赖属性实现

依赖属性的实现很简单：

(1) 所在类型继承自 DependencyObject，几乎所有的 WPF 控件都间接继承自该类型。
(2) 使用 public static 声明一个 DependencyProperty，该字段才是真正的依赖属性 (字段)。
(3) 在静态构造中完成依赖属性的元数据注册，并获取对象引用。
(4) 提供一个依赖属性的实例化包装属性。注意使用 DependencyObject 相关方法作为读取/访问器。

```
public class MyClass : DependencyObject
{
  public static readonly DependencyProperty TestProperty;

  static MyClass()
  {
    TestProperty = DependencyProperty.Register("Test", typeof(string),
typeof(MyClass),
      new PropertyMetadata("Hello, World!", OnTestChanged));
  }

  private static void OnTestChanged(DependencyObject o,
DependencyPropertyChangedEventArgs e)
  {

  }

  public string Test
  {
    get { return (string)GetValue(TestProperty); }
    set { SetValue(TestProperty, value); }
  }
}
```

提示：在 VS2008 中可以使用 "propdp + TAB" 快速生成依赖属性代码。

2. 变更通知

当依赖属性值发生变化时，WPF 会通过预先注册的元数据 (Metadata) 信息完成某些 "关联行为" 调用。这样我们就可以在 XAML 的声明中完成行为控制，比如开始或停止动画。

我们试着将上面我们创建的自定义依赖属性作为源绑定给相关控件。

```
public partial class Window1 : Window
{
```

```
  MyClass o;

  public Window1()
  {
    InitializeComponent();

    o = new MyClass();

    var binding = new Binding("Test") { Source = o, Mode = BindingMode.TwoWay };

    this.textBox1.SetBinding(TextBox.TextProperty, binding);
  }

  private void btnTest_Click(object sender, RoutedEventArgs e)
  {
    o.Test = DateTime.Now.ToString();
  }
}
```

窗体初始化时，textBox1.Text 自动绑定为 MyClass.TestProperty 的默认值 "Hello, World!" (参考 MyClass 静态构造的元数据注册代码)。每当我们单击按钮修改依赖属性(o.Test)时，textBox1.Text 都将自动同步更新，这就是依赖属性的行为方式。当然，我们还应该提供一个完全基于 XAML 的声明方式演示，而不是上面的 C# 代码。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <TextBox x:Name="textBox1" />
    <Label x:Name="label1" Content="{Binding ElementName=textBox1, Path=Text}"
/>
  </Grid>
</Window>
```

label1.Content 绑定到 textBox1.Text 这个依赖属性上，每当我们修改 textBox1.Text 时，label1.Content 都会保持同步修改，这些动作无需我们编写任何代码。很显然，这大大简化了 XAML 的行为控制能力，尤其是对所谓富功能 (Rich functionality) 的控制。

WPF 提供了一种称之为 "触发器(Trigger)" 的机制来配合依赖属性工作，我们用一个简单的属性触发器看看效果。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
```

```
  <Grid>
    <Button x:Name="btnTest">
      <Button.Style>
        <Style TargetType="{x:Type Button}">
          <Style.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
              <Setter Property="Foreground" Value="Red" />
            </Trigger>
          </Style.Triggers>
        </Style>
      </Button.Style>
    </Button>
  </Grid>
</Window>
```

当依赖属性 Button.IsMouseOver 发生变化时 (== True)，将导致触发器执行，设置 Foreground=Red。很显然这比我们处理 MouseEnter 事件要简单得多，关键是 UI 设计人员无需编写代码即可得到所需的效果。除了属性触发器外，WPF 还提供了数据触发器和事件触发器。

3. 属性值继承

此继承非 OOP 上的继承，它的本意是父元素的相关设置会自动传递给所有层次的子元素 (元素可以从其在树中的父级继承依赖项属性的值)。其实很简单也很熟悉，当我们修改窗体父容器控件的字体设置时，所有级别的子控件都将自动使用该字体设置(未做自定义设置)，相信你在 WinForm 中已经使用过了。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" FontSize="20">
  <Grid>
    <TextBox x:Name="textBox1" />
    <Label x:Name="label1" Content="Hello, World!" />
    <Button x:Name="btnTest" Content="Test" />
  </Grid>
</Window>
```

Window.FontSize 设置会影响所有的内部元素字体大小，这就是所谓的属性值继承。当然，一旦子元素提供了显式设置(比如下例中的 label1.FontSize)，这种继承就会被打断。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" FontSize="20">
  <Grid>
    <TextBox x:Name="textBox1" />
    <Label x:Name="label1" Content="Hello, World!" FontSize="10" />
```

```
    <Button x:Name="btnTest" Content="Test" />
  </Grid>
</Window>
```

注意并不是所有的依赖属性都会继承父元素的设置。

4. 多提供程序优先级

WPF 允许我们可以在多个地方设置依赖属性的值，这的确很方便，但也问题也不少。比如下面的例子中，我们在三个地方设置了按钮的背景颜色，只是最后哪个会起作用呢？

```
<Button x:Name="button1" Background="Red">
  <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>
```

确切的答案是 "<Button Background='Red'>" 起作用了，为什么呢？因为 WPF 内在的优先级规则决定了 "本地值" 优先级别最高。

```
本地值 > 样式触发器 > 模板触发器 > 样式设置程序 > 主题样式触发器 > 出题样式设置程序 > 属性值继承 > 默认值
```

这样的一个过程被称之为 "基础值判断"。我们需要特别说名一下，所谓本地值是指我们直接或间接调用了 DependencyObject.SetValue，也就是显示设置了依赖属性的值。我们可以用下面这样的代码清除本地值设置。

```
this.button1.ClearValue(Button.BackgroundProperty);
```

虽然我们获取了基础值，但事情并没有结束，接下来有几个更厉害的选手出场，他们的优先级别更高，依赖属性必须一一过关才算最后确定下来。

```
基础值判断 -> 表达式计算 -> 应用动画 -> 限制(Coerce) -> 验证 -> 最终结果
```

(1) 验证是指我们注册依赖属性所提供的 ValidateValueCallback 委托方法，它最终决定了属性值设置是否有效；

(2) 限制则是注册时提供的 CoerceValueCallback 委托，它负责验证属性值是否在允许的限制范围之内，比如大于等于 9 小于等等 100；

(3) 动画是一种特殊行为，它的优先级高于基础设置也能理解；

(4) 如果依赖属性值是计算表达式 (System.Windows.Expression，比如前面示例中的绑定语法)，那么 WPF

会尝试 "计算" 表达式的结果。
(5) 基础值就是上面提供的那些显示设置，它的优先级比较好确定。

5. 附加属性

附加属性是一种特殊的依赖属性，它看上去颇为古怪，尤其是对我们这些熟悉了面向对象规则的程序员而言。看下面的例子。

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

DockPanel.Dock 是 DockPanel 中定义的依赖属性，但却出现在子元素的声明上，看上去很诡异。

```
<StackPanel TextElement.FontSize="30" TextElement.FontStyle="Blod">
  <Button>Help</Button>
  <Button>OK</Button>
</StackPanel>
```

TextElement.FontSize, TextElement.FontStyle 既不属于 StackPanel，也不属于 Button，但却能完成元素树的字体定义。

附加属性严格来说是一个 XAML 概念，依赖属性是 WPF 概念。附加属性通常用于界面元素的布局设置。这样一种特殊的扩展机制，使得父元素可以将一些自定义设置传递给所有子元素，而无需要求子元素必须具备相同的依赖属性。这种应用方式有点扩展方法的意思，初次接触时的确不太好理解。

RoutedEvent

WPF 的采取了路由事件机制，这样事件可以在可视树上层级传递。要知道 XAML 中控件都是由很多其他元素组合而成，比如我们单击了 Button 内部的 TextBlock 元素，Button 依然可以可以接收到该事件并触发 Button.Click。通常情况下，我们只是关心逻辑树上的事件过程。

我们看看 Button Click 事件的实现。

```
public abstract class ButtonBase : ContentControl, ICommandSource
{
  public static readonly RoutedEvent ClickEvent;

  static ButtonBase()
  {
    ClickEvent = EventManager.RegisterRoutedEvent("Click",
RoutingStrategy.Bubble,
      typeof(RoutedEventHandler), typeof(ButtonBase));
    ......
  }

  public event RoutedEventHandler Click
```

```
  {
    add { base.AddHandler(ClickEvent, value); }
    remove { base.RemoveHandler(ClickEvent, value); }
  }
}
```

看上去很简单，不是吗？和依赖属性有点类似。

注册路由事件时，我们可以选择不同的路由策略。

- **管道传递(Tunneling)**：事件首先在根元素上触发，然后向下层级传递，直到那个最初触发事件的子元素。
- **冒泡(Bubbling)**：事件从最初触发事件的子元素向根元素层级往上传递。
- **直接(Direct)**：事件仅在最初触发事件的子元素上触发。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <Border MouseRightButtonDown="MouseRightButtonDown">
      <StackPanel MouseRightButtonDown="MouseRightButtonDown">
        <Button MouseRightButtonDown="MouseRightButtonDown">Test</Button>
      </StackPanel>
    </Border>
  </Grid>
</Window>
```

Window1.xaml.cs

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

  private void MouseRightButtonDown(object sender, MouseButtonEventArgs e)
  {
    MessageBox.Show((sender as Label).Name);
  }
}
```

在按钮上单击右键后，你会依次看到显示 "Button"、"StackPanel"、"Border" 的三个对话框，显然事件按照冒泡向根元素传递。

有一点需要注意，WPF 路由事件参数有个 Handled 属性标记，一旦被某个程序标记为已处理，事件传递就会终止。测试一下。

```
public partial class Window1 : Window
{
  private void MouseRightButtonDown(object sender, MouseButtonEventArgs e)
  {
    MessageBox.Show(sender.GetType().Name);
    if (sender.GetType().Name == "StackPanel") e.Handled = true;
  }
}
```

很有效，Border.MouseRightButtonDown 不在有效。严格来说，事件并没有被终止，它依然会继续传递个上级或下级的元素，只是 WPF 没有触发事件代码而已。我们可以使用 AddHandler 方法重新注册一个新的事件处理方法，使得可以继续处理被终止的事件(注意：如果事件没有终止，这会导致两次事件处理)。

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

    this.border1.AddHandler(Border.MouseRightButtonDownEvent,
      new MouseButtonEventHandler(MouseRightButtonDown), true);
  }

  private void MouseRightButtonDown(object sender, MouseButtonEventArgs e)
  {
    MessageBox.Show(sender.GetType().Name);
    if (sender.GetType().Name == "StackPanel") e.Handled = true;
  }
}
```

再运行试试，你会发现 Border.MouseRightButtonDown 被触发了。

```
public void AddHandler(
  RoutedEvent routedEvent,
  Delegate handler,
  bool handledEventsToo
)
```

handledEventsToo：如果为 true，则将按以下方式注册处理程序：即使路由事件在其事件数据中标记为已处理，也会调用该处理程序；如果为 false，则使用默认条件注册处理程序，即当路由事件被标记为已处理时，将不调用处理程序。

通常情况下，WPF 控件会在管道事件的名称前添加 Preview 前缀。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <Border PreviewMouseRightButtonDown="MouseRightButtonDown">
      <StackPanel PreviewMouseRightButtonDown="MouseRightButtonDown">
        <Button
PreviewMouseRightButtonDown="MouseRightButtonDown">Test</Button>
      </StackPanel>
    </Border>
  </Grid>
</Window>
```

这回的输出结果正好跟前面的演示反过来，依次是 "Border"、"StackPanel"、"Button"。如果继续保留事件终止代码，那么 Button.PreviewMouseRightButtonDown 就不再被触发。

附加事件

和附加属性类似，WPF 允许我们在一个没有定义事件的元素上处理经管道或冒泡传递的路由事件。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel Button.Click="Click">
      <Button>Test</Button>
    </StackPanel>
  </Grid>
</Window>
```

Window1.xaml.cs

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

  private void Click(object sender, RoutedEventArgs e)
  {
    MessageBox.Show(e.Source.ToString());
```

```
    }
}
```

输出:

System.Windows.Controls.Button:Test

虽然 StackPanel 没有 Click 事件，但我们依然捕获了 Button 的点击事件，还有就是注意附加属性的写法。不要想当然认为该示例表示只有 Button 类型的子元素才会触发 Click 附加事件。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel Button.Click="Click">
      <Button>Test</Button>
      <CheckBox>Check</CheckBox>
    </StackPanel>
  </Grid>
</Window>
```

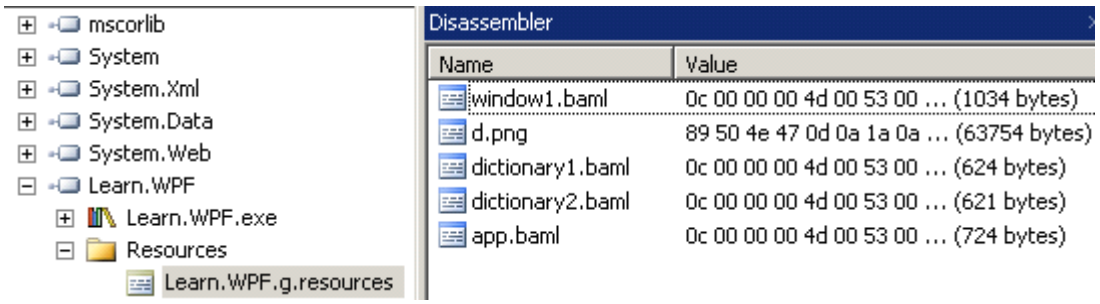
很遗憾地告诉你，CheckBox 一样触发了 Click 附加事件。

Resource

1. 二进制资源

WPF 支持三种方式的二进制资源，这些资源可以非常方便地在 XAML 中使用。

- **Resource**: 将资源嵌入程序集中，和 Embedded Resource 有点像。区别在于 WPF 将相关资源打包到 .Resources 文件，然后再由编译器嵌入到程序集文件中。WPF 默认的 URI 访问方式是不支持 Embedded Resource 的。
- **Content**: 资源不会嵌入到程序集，仅仅在程序集清单中添加一条记录，这和以往我们所熟悉的方式一样。资源文件必须随其他程序集文件一起部署到目标目录。
- **Loose File**: 这类资源通常是运行期动态确定或加入的。



WPF 通过**统一资源标识符(URI)**访问相关资源文件。

(1) 访问 Resource / Content 资源

```
<Image Source="/a.png" />
<Image Source="/xxx/x.png" />
```

(2) 访问松散资源文件(Loose File)

```
<Image Source="pack://siteOfOrigin:,,,/a.png" />
<Image Source="c:\test\a.png" />
```

XAML 编译时需要验证所有资源有效性，因此在使用松散资源文件时必须使用有效的 URI 路径。"pack://" 表示 Package URI，"pack://siteOfOrigin:,,," 表示从部署位置开始，相应的还有 "pack://application:,,," 上面的 Resource 资源全路径应该是 "pack://application:,,,/a.png"，只不过通常情况下我们使用省略写法而已。

```
<Image Source="pack://application:,,,/a.png" />
```

其他的 URI 写法还包括：

```
<Image Source="http://www.qidian.com/images/logo.gif" />
<Image Source="\\server1\share\logo.gif" />
<Image Source="file://c:/test/logo.gif" />
```

(3) 访问其他程序集中的资源文件

提供一个可替换的专用资源 DLL 也是一种常用编程手法，尤其是多语言或者换肤机制。WPF 访问其他程序集资源文件的语法有点古怪。

```
pack://application:,,,/AssemblyReference;Component/ResourceName
```

- 其他程序集的资源必须以 Resource 方式嵌入。
- 不能省略 "/AssemblyReference" 前面的反斜杠。
- Component 是关键字，必须包含在 URI 中。

```
<Image Source="pack://application:,,,/Learn.Library;component/s.gif" />
<Image Source="/Learn.Library;component/s.gif" />
```

2. 逻辑资源

逻辑资源是一些存储在元素 Resources 属性中的对象，这些 "预定义" 的对象被一个或多个子元素所引用或共享。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <ContentControl x:Key="label1">Hello, World!</ContentControl>
    <ImageSource x:Key="image1">/a.png</ImageSource>
  </Window.Resources>
  <Grid>
```

```
    <Label Content="{StaticResource label1}" />
    <Image Source="{StaticResource image1}" />
  </Grid>
</Window>
```

当然，我们也可以写成下面这种格式。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <ContentControl x:Key="label1">Hello, World!</ContentControl>
    <ImageSource x:Key="image1">/a.png</ImageSource>
  </Window.Resources>
  <Grid>
    <Label>
      <Label.Content>
        <StaticResource ResourceKey="label1" />
      </Label.Content>
    </Label>
    <Image>
      <Image.Source>
        <StaticResource ResourceKey="image1" />
      </Image.Source>
    </Image>
  </Grid>
</Window>
```

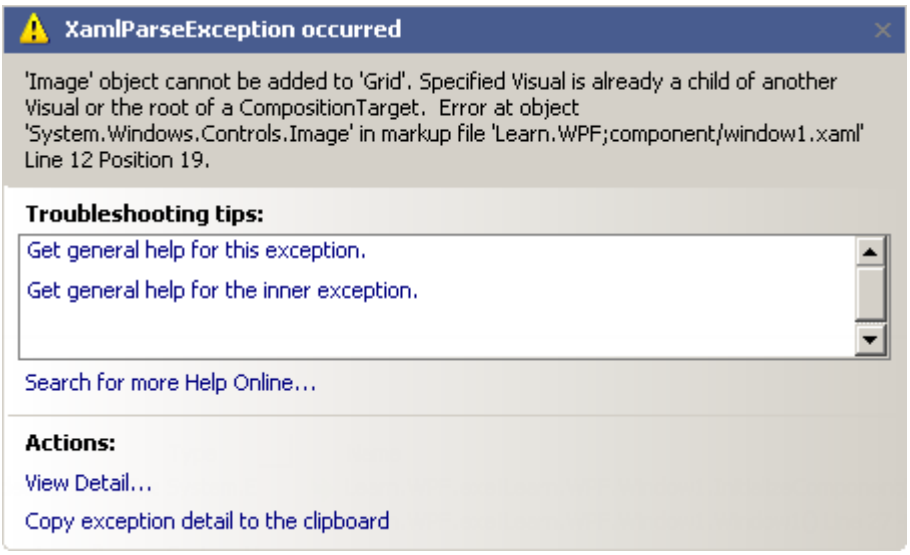
我们甚至可以用逻辑资源定义一个完整的子元素。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <Label x:Key="label1" Content=" Hello, World!" />
    <Image x:Key="image1" Source="/a.png" />
  </Window.Resources>
  <Grid>
    <StaticResource ResourceKey="label1" />
    <StaticResource ResourceKey="image1" />
  </Grid>
</Window>
```

逻辑资源有个限制，那就是这些继承自 Visual 的元素对象只能有一个父元素，也就是说不能在多个位置使用，因为多次引用的是同一个对象实例。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <Image x:Key="image1" Source="/a.png" />
  </Window.Resources>
  <Grid>
    <StaticResource ResourceKey="image1" />
    <StaticResource ResourceKey="image1" />
  </Grid>
</Window>
```

你将看到下面这样一个错误提示。



解决方法就是添加 "x:Shared=False"，这样每次引用都会生成一个新的逻辑资源对象。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <Image x:Key="image1" x:Shared="False" Source="/a.png" />
  </Window.Resources>
  <Grid>
    <StaticResource ResourceKey="image1" />
    <StaticResource ResourceKey="image1" />
```

```
  </Grid>
</Window>
```

资源查找：引用逻辑资源时首先会查找父元素 Resources 集合，如未找到，会逐级检查更上层的父元素。如果直到根元素依然未找到有效的逻辑资源定义，那么 WPF 会检查 Application.Resources (App.xaml 中定义) 和系统属性集合(SystemParamters 等)。每个独立的资源字典中键名不能重复，但在多个不同层级的资源字典中允许重复，离资源引用最近的那个逻辑资源项被优先采纳。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <Label x:Key="label1" Content=" Hello, World!" />
  </Window.Resources>
  <Grid>
    <Grid.Resources>
      <Label x:Key="label1" Content=" Hello, C#!" />
    </Grid.Resources>
    <StackPanel>
      <StaticResource ResourceKey="label1" />
    </StackPanel>
  </Grid>
</Window>
```

将显示 "Hello, C#!"。

逻辑资源的引用方式又分类 "**静态(StaticResource)**" 和 "**动态(DynamicResource)**" 两种方式，区别在于静态引用仅在第一次资源加载时被应用，而动态引用则会在资源被更改时重新应用。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <ContentControl x:Key="label1" x:Shared="False">Hello,
World!</ContentControl>
  </Window.Resources>
  <x:Code>
    private void button1_Click(object sender, RoutedEventArgs e)
    {
      this.Resources["label1"] = "Hello, C#!";
    }
  </x:Code>
  <Grid>
    <StackPanel>
```

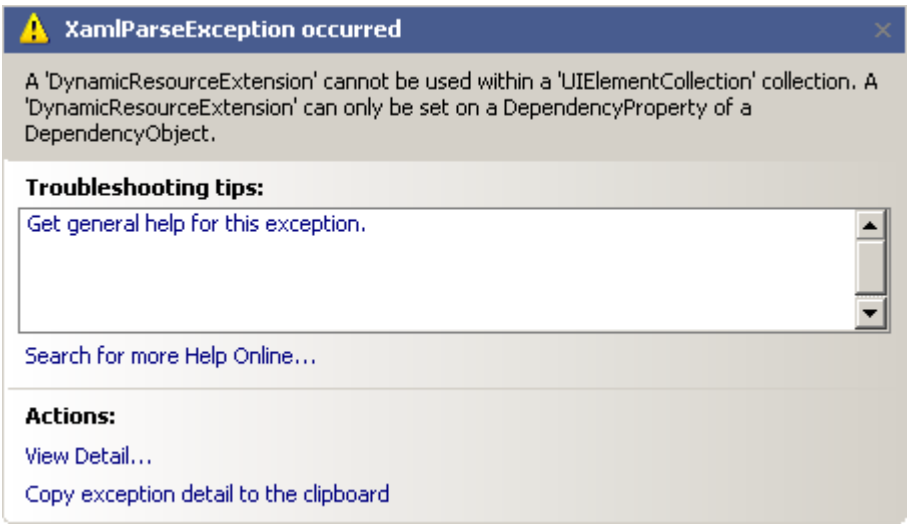
```
      <Label x:Name="label1" Content="{StaticResource label1}" />
      <Label x:Name="label2" Content="{DynamicResource label1}" />
      <Button x:Name="button1" Click="button1_Click">Test</Button>
    </StackPanel>
  </Grid>
</Window>
```

单击按钮后，你会发现 label2 实时反应了资源的修改。

动态资源只能用于设置依赖属性值，因此它不能像静态资源那样引用一个完整的元素对象。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <Label x:Key="label" x:Shared="False" Content="Hello, World!" />
  </Window.Resources>
  <Grid>
    <StackPanel>
      <DynamicResource ResourceKey="label" />
    </StackPanel>
  </Grid>
</Window>
```

这将导致出现下图这样的异常，而静态资源则没有这个问题。

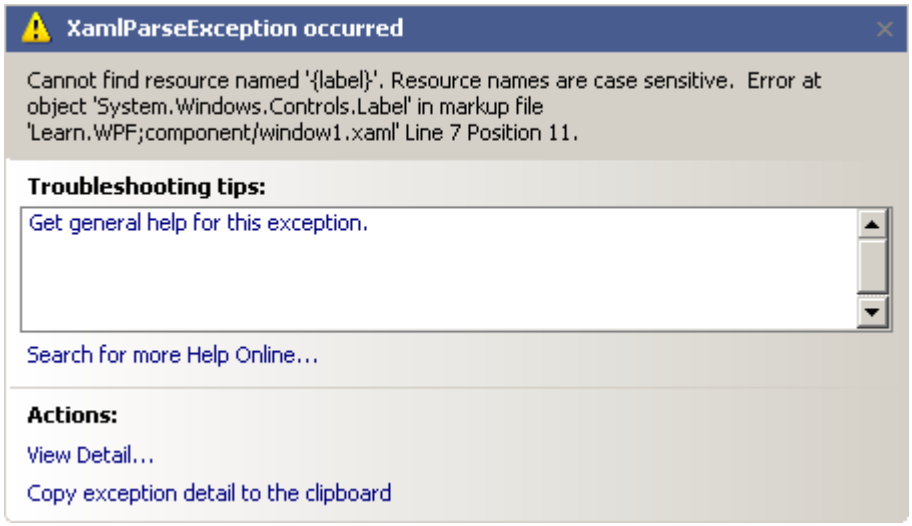


当然，静态资源也有另外一个麻烦，就是不支持前向引用(Forward Reference)，也就是说我们必须先定义资源，然后才能使用静态引用。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel>
      <Label Content="{StaticResource label}" />
    </StackPanel>
  </Grid>
  <Window.Resources>
    <ContentControl x:Key="label">Hello, World!</ContentControl>
  </Window.Resources>
</Window>
```

属于静态资源的异常出现了，当然动态资源是没有这个问题的。



如果有必要的话，我们可以将逻辑资源分散定义在多个 XAML 文件(Resource Dictionary) 中。

Dictionary1.xaml

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ContentControl x:Key="label1">Hello, World!</ContentControl>
</ResourceDictionary>
```

Dictionary2.xaml

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ContentControl x:Key="label2">Hello, C#!</ContentControl>
</ResourceDictionary>
```

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="Dictionary1.xaml" />
        <ResourceDictionary Source="Dictionary2.xaml" />
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <Label Content="{DynamicResource label1}" />
      <Label Content="{DynamicResource label2}" />
    </StackPanel>
  </Grid>
</Window>
```

如果合并的字典中有重复的键值，那么最后加入的资源将优先。比如 Dictionary1.xaml 和 Dictionary2.xaml 都有一个 x:Key="label1" 的资源，那么 Dictionary2.label1 将获胜。

在程序代码中我们可以用 FindResource/TryFindResource 设置静态资源，用 SetResourceReference 设置动态资源。

```
this.labelA.Content = (ContentControl)this.labelA.FindResource("key1"); //
StaticResource
this.labelB.SetResourceReference(Label.ContentProperty, "key2"); //
DynamicResource
```

Binding

1. 绑定简介

WPF 绑定可以在源数据对象和 UI 控件间建立联系，实现单向或双向变更通知，以此实现更好的业务逻辑和 UI 的分离。通常的模式是：**将目标对象(通常是 XAML 元素控件等)的目标属性(必须是依赖属性)通过绑定对象(Binding 对象实例)绑定到数据源(CLR 对象、ADO.NET 数据表、XML 数据等)。**比如我们可以将 TextBox1.Text 绑定到 Personal.Name。

下面的例子中，我们可以观察到如下自动行为。

(1) 单击 btnSet 修改源对象，会发现目标属性 textbox1.Text 自动变更。
(2) 修改 textbox1.Text，单击 btnGet 会发现源对象被自动修改。

Window.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1" Height="276" Width="360"
WindowStartupLocation="CenterScreen">
  <Grid>
    <StackPanel>
      <TextBox x:Name="textbox1" />
      <Button x:Name="btnGet" Content="Get Name" Click="buttonClick" />
      <Button x:Name="btnSet" Content="Set Name" Click="buttonClick" />
    </StackPanel>
  </Grid>
</Window>
```

Windows.xaml.cs

```
class MyData : DependencyObject
{
  public static readonly DependencyProperty NameProperty =
   DependencyProperty.Register("Name", typeof(string), typeof(MyData),
   new UIPropertyMetadata("Hello, World!"));

  public string Name
  {
    get { return (string)GetValue(NameProperty); }
    set { SetValue(NameProperty, value); }
  }
}

public partial class Window1 : Window
{
  MyData data;

  public Window1()
```

```
  {
    InitializeComponent();

    data = new MyData();

    var binding = new Binding("Name") { Source = data };
    this.textbox1.SetBinding(TextBox.TextProperty, binding);
  }

  private void buttonClick(object sender, RoutedEventArgs e)
  {
    if (sender == btnSet)
      data.Name = DateTime.Now.ToString();
    else
      MessageBox.Show(data.Name);
  }
}
```

很显然，这种效果可以让开发人员只关注业务逻辑或者 UI 展示，大大降低了两者之间的代码关联。

我们还可以使用 Binding.Mode 属性来指定绑定变更通知的方向，默认情况下通常是双向绑定。

- **OneWay**: 对数据源进行修改，会自动更新目标属性。而对目标属性的修改则不会影响源对象。
- **TwoWay**: 无论是修改数据源还是目标属性，都会自动更新另一方。
- **OneWayToSource**: 和 OneWay 相反，当修改目标属性时会自动更新数据源，反之则不然。
- **OneTime**: 仅在初始化时修改目标属性。

```
data = new MyData();
var binding = new Binding("Name") { Source = data, Mode = BindingMode.OneWay };

this.textbox1.SetBinding(TextBox.TextProperty, binding);
```

System.Windows.Data.BindingOperations 类提供了绑定所需的全部的操作方法。和 FrameworkElement.SetBinding() 相比，BindingOperations.SetBinding 方法可能更通用些，因为它可以直接使用 DependencyObject 对象。

```
public BindingExpression FrameworkElement.SetBinding(DependencyProperty dp,
BindingBase binding)

public static BindingExpressionBase
BindingOperations.SetBinding(DependencyObject target,
  DependencyProperty dp, BindingBase binding)
```

将上面例子改成 BindingOperations 试试。

```
BindingOperations.SetBinding(this.textbox1, TextBox.TextProperty, new
Binding("Name") { Source = data });
```

我们可以用 ClearBinding 方法解除绑定。

```
private void buttonClick(object sender, RoutedEventArgs e)
{
  if (sender == btnSet)
    data.Name = DateTime.Now.ToString();
  else if (sender == btnClear)
    BindingOperations.ClearBinding(this.textbox1, TextBox.TextProperty);
  else
    MessageBox.Show(data.Name);
}
```

单击 btnClear 后，你会发现绑定自动变更失效。

2. 在 XAML 中使用绑定

在 XAML 中我们不能使用 SetBinding，而必须改用扩展标记 Binding (注意没有 Extension 后缀)。该扩展标记可将目标属性绑定到静态资源(StaticResource)或者其他 XAML 元素(包括目标元素自身)上。

(1) 绑定到其他元素

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel>
      <TextBox x:Name="textbox1" />
      <Label x:Name="label1" Content="{Binding ElementName=textbox1,
Path=Text}" />
    </StackPanel>
  </Grid>
</Window>
```

(2) 绑定到静态资源

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>
    <ContentControl x:Key="text">Hello, World!</ContentControl>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <Label x:Name="label1" Content="{Binding Source={StaticResource text}}"
```

```
/>
    </StackPanel>
  </Grid>
</Window>
```

System.Windows.Data.Binding.Source 并不是一个依赖属性，因此我们无法将其绑定到一个动态资源(DynamicResouce)上。

(3) 绑定到自身

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel>
      <Label x:Name="label1" Content="{Binding RelativeSource={RelativeSource
Self}, Path=Name}" />
    </StackPanel>
  </Grid>
</Window>
```

(4) 绑定到指定类型的父元素

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid x:Name="Grid1">
    <StackPanel>
      <Label x:Name="label1" Content="{Binding RelativeSource={RelativeSource
FindAncestor,
        AncestorType={x:Type Grid}}, Path=Name}" />
    </StackPanel>
  </Grid>
</Window>
```

3. 绑定到普通对象

WPF 允许将任何 .NET 对象作为数据绑定源。

```
class Data
{
  public string Name { get; set; }
}

public partial class Window1 : Window
```

```
{
  public Window1()
  {
    InitializeComponent();

    var data = new Data { Name = "Q.yuhen" };
    BindingOperations.SetBinding(this.textbox1, TextBox.TextProperty, new
Binding("Name") { Source = data });
  }
}
```

不过有个问题，就是当我们修改数据源时，目标属性会因为无法接收变更通知而自动更新。要解决这个问题，我们需要让数据源对象实现 System.ComponentModel.INotifyPropertyChanged 接口。

```
public delegate void PropertyChangedEventHandler(object sender,
PropertyChangedEventArgs e);

public interface INotifyPropertyChanged
{
  // Events
  event PropertyChangedEventHandler PropertyChanged;
}
```

我们试着修改一下上面例子中的 Data 类型。

```
class Data : INotifyPropertyChanged
{
  private string name;
  public event PropertyChangedEventHandler PropertyChanged;

  public string Name
  {
    get
    {
      return name;
    }
    set
    {
      if (value != name)
      {
        name = value;

        if (PropertyChanged != null)
        {
          PropertyChanged(this, new PropertyChangedEventArgs("Name"));
        }
```

```
      }
    }
  }
}

public partial class Window1 : Window
{
  Data data;

  public Window1()
  {
    InitializeComponent();

    data = new Data { Name = "Q.yuhen" };
    BindingOperations.SetBinding(this.textbox1, TextBox.TextProperty,
      new Binding("Name") { Source = data });
  }

  protected void ButtonClick(object sender, RoutedEventArgs e)
  {
    if (sender == button1)
      data.Name = DateTime.Now.ToString();
    else
      MessageBox.Show(data.Name);
  }
}
```

好了，现在支持双向变更自动更新了。

4. 绑定到集合

在实际开发中，我们通常是将一个集合数据对象 (比如数据表) 绑定到一个 DataGrid 或者 ListBox 列表控件上，这时候我们就需要使用到集合绑定方式。WPF 特意为我们实现了一个 System.Collections.ObjectModel.ObservableCollection<T> 泛型集合，省却了我们写具备变更通知功能集合代码的时间。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Grid>
    <StackPanel>
      <ListBox x:Name="listbox1"></ListBox>
```

```
    </StackPanel>
  </Grid>
</Window>
```

Window1.xaml.cs

```
public class Personal
{
  public string Name { get; set; }
  public int Age { get; set; }
  public string Sex { get; set; }
}

public class PersonalList : ObservableCollection<Personal>
{
}

public partial class Window1 : Window
{
  public Window1()
  {
    InitializeComponent();

    var list = new PersonalList
    {
      new Personal { Name = "Tom", Age = 10, Sex = "Male" },
      new Personal { Name = "Mary", Age = 15, Sex = "Female" },
      new Personal { Name = "Jack", Age = 12, Sex = "Male" },
    };

    var binding = new Binding{ Source = list };
    this.listbox1.SetBinding(ListBox.ItemsSourceProperty, binding);
    this.listbox1.DisplayMemberPath = "Name";
  }
}
```

注意使用 DisplayMemberPath 属性指定 ListBoxItem 的内容，否则会调用 ToString() 来显示结果。

当然，我们也可以直接绑定逻辑资源中的列表数据。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
```

```
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <ListBox x:Name="listbox2"
        ItemsSource="{Binding Source={StaticResource personals}}"
        DisplayMemberPath="Name">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

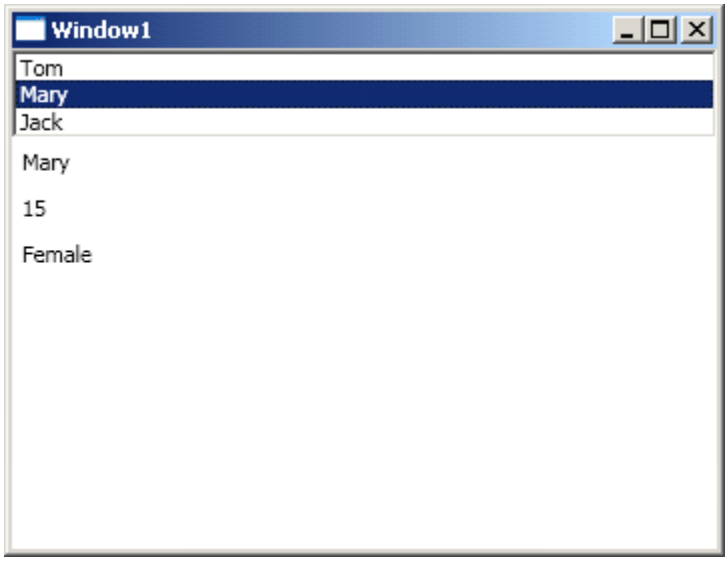
为了使用 Personal 和 PersonalList，我们引入了一个 CLR Namespace。

在主从结构 (Master-Detail) 显示中，我们通常要实现 "选择项跟踪" 功能，也就是说当我们选中主表的某个记录时，其他细节控件要同步刷新该记录的细节内容。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <ListBox x:Name="listbox1"
        ItemsSource="{Binding Source={StaticResource personals}}"
        DisplayMemberPath="Name"
        IsSynchronizedWithCurrentItem="True">
      </ListBox>

      <Label x:Name="lblName" Content="{Binding Source={StaticResource
personals}, Path=Name}" />
```

```
      <Label x:Name="lblAge" Content="{Binding Source={StaticResource
personals}, Path=Age}"/>
      <Label x:Name="lblSex" Content="{Binding Source={StaticResource
personals}, Path=Sex}"/>
    </StackPanel>
  </Grid>
</Window>
```

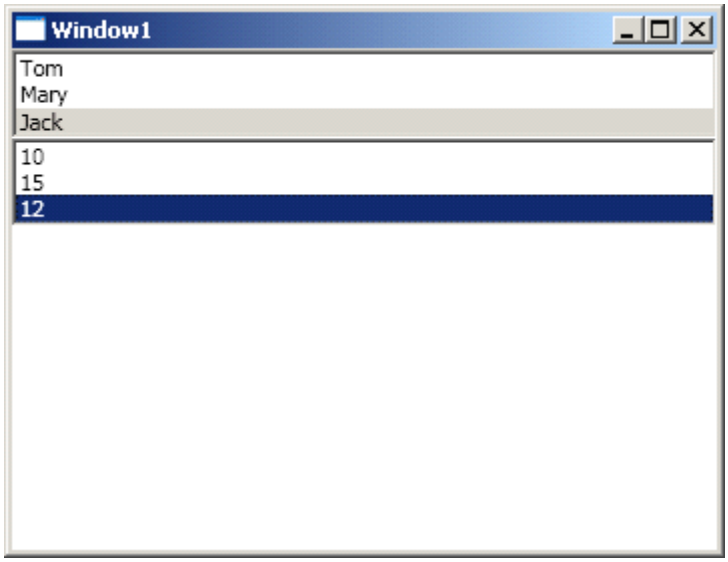
我们将 ListBox.IsSynchronizedWithCurrentItem 置为 True，这样当我们改变 ListBox 选择项时，下面三个标签会自动同步变更为被选中记录的信息。



当然，我们还可以使用多个 ListBox，就像真正的主从表显示那样相互影响。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel>
      <ListBox x:Name="listbox1"
        ItemsSource="{Binding Source={StaticResource personals}}"
        DisplayMemberPath="Name"
```

```
        IsSynchronizedWithCurrentItem="True">
      </ListBox>

      <ListBox x:Name="listbox2"
        ItemsSource="{Binding Source={StaticResource personals}}"
        DisplayMemberPath="Age"
        IsSynchronizedWithCurrentItem="True">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```



有一点需要说明，IsSynchronizedWithCurrentItem 不支持多项选择同步。

5. DataContext 共享源

在上面的例子中，我们需要将同一资源绑定到多个 UI 元素上，很显然到处写 "{Binding Source={StaticResource personals}}" 是件很繁琐且不利于修改的做法。WPF 提供了一个称之为 "数据上下文 (DataContext)" 的东西让我们可以在多个元素上共享一个源对象，只需将其放到父元素 DataContext 属性即可。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
```

```
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1"
        ItemsSource="{Binding}"
        DisplayMemberPath="Name"
        IsSynchronizedWithCurrentItem="True">
      </ListBox>

      <Label x:Name="lblName" Content="{Binding Path=Name}" />
      <Label x:Name="lblAge" Content="{Binding Path=Age}"/>
      <Label x:Name="lblSex" Content="{Binding Path=Sex}"/>
    </StackPanel>
  </Grid>
</Window>
```

当我们不给 Binding 扩展标志指定 Source 属性时，它会自动寻找上级父元素的数据上下文。

当然，我们也可以在代码中做同样的事情。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Grid>
    <StackPanel x:Name="stackPanel1">
      <ListBox x:Name="listbox1"
        ItemsSource="{Binding}"
        DisplayMemberPath="Name"
        IsSynchronizedWithCurrentItem="True">
      </ListBox>

      <Label x:Name="lblName" Content="{Binding Path=Name}" />
      <Label x:Name="lblAge" Content="{Binding Path=Age}"/>
      <Label x:Name="lblSex" Content="{Binding Path=Sex}"/>
    </StackPanel>
  </Grid>
</Window>
```

Window1.xaml.cs

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

    var list = new PersonalList {
     new Personal{Name="Tom", Age=10, Sex="Male"},
     new Personal{Name="Mary", Age=15, Sex="Female"},
     new Personal{Name="Jack", Age=12, Sex="Male"},
    };

    this.stackPanel1.DataContext = list;
  }
}
```

从上面的例子中，我们可以看出使用 DataContext 使得数据和 UI 分离更加灵活，因为 DataContext 可以跨越多级父元素。比如我们可以直接将数据源设置为 Window.DataContext。

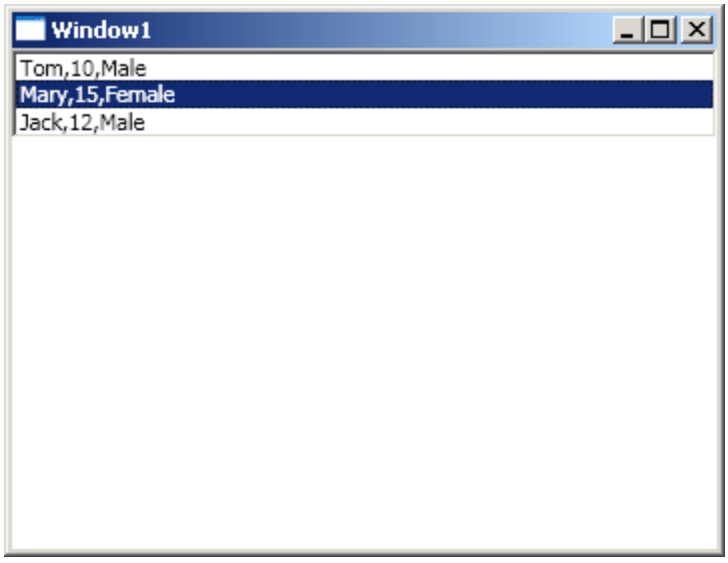
```
public partial class Window1 : Window
{
  public Window1()
  {
    ... ...

    this.DataContext = list;
  }
}
```

6. 数据模板

数据模板为展示数据提供了极大的灵活性，我们继续以前面的例子来看看它的能力。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
```

```
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">

        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>, </TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>

      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```



很显然，利用 ListBox.ItemTemplate.DataTemplate 我们可以用更复杂和更丰富的手段显示数据源对象，不再仅仅是通过 Path 显示某单个属性。不过通常情况下，我们会像 HTML/CSS 那样将数据模板定义到资源中，以便在多个地方重复使用。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
```

```
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <DataTemplate x:Key="myData">
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Age}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Sex}" />
      </StackPanel>
    </DataTemplate>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}"
        ItemTemplate="{StaticResource myData}">

      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

当然，我们也可以使用 CSS 那样的选择器来指定模板的应用类型，而不是在 ListBox 上设置 ItemTemplate 属性。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <DataTemplate DataType="{x:Type my:Personal}">
```

```
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Age}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Sex}" />
      </StackPanel>
    </DataTemplate>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

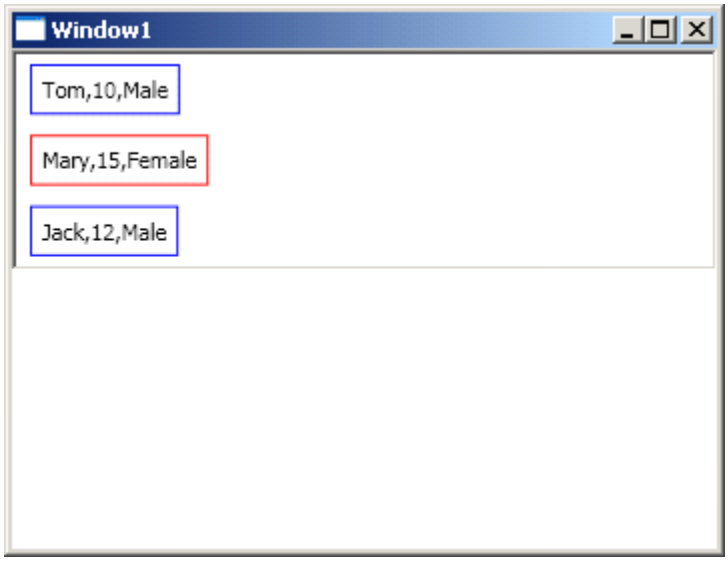
通过 DataType="{x:Type my:Personal}" 我们就可以让所有使用 Personal 对象的地方自动使用这个模板设置。

利用数据模板，我们还可以做出复杂的效果来，比如根据 Personal.Sex 来显示一个不同颜色的边框。类似的做法在 WinForm 似乎很麻烦，现在只需做些简单的设置即可。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <DataTemplate DataType="{x:Type my:Personal}">
      <Border x:Name="border1" BorderBrush="Red" BorderThickness="1"
Padding="5" Margin="5">
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Path=Name}" />
          <TextBlock>,</TextBlock>
          <TextBlock Text="{Binding Path=Age}" />
          <TextBlock>,</TextBlock>
          <TextBlock Text="{Binding Path=Sex}" />
```

```
        </StackPanel>
      </Border>

      <DataTemplate.Triggers>
        <DataTrigger Binding="{Binding Path=Sex}">
          <DataTrigger.Value>Male</DataTrigger.Value>
          <Setter TargetName="border1" Property="BorderBrush" Value="Blue" />

        </DataTrigger>
      </DataTemplate.Triggers>
    </DataTemplate>

  </Window.Resources>

  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

首先，我们在数据模板中增加了一个 Border，默认颜色是红色。然后利用触发器，当 Personal.Sex == Male 时，将边框颜色设置为蓝色。



7. 值转换器

某些时候，我们需要对绑定的源值进行类型或者显示格式转换，那么可以采用值转换器达到这个目的。比如我们可以将上面的 Sex 转换成 "男"、"女" 来显示。

Window1.xaml.cs

```
public class SexConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
  {
    return value.ToString() == "Male" ? "男" : "女";
  }

  public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

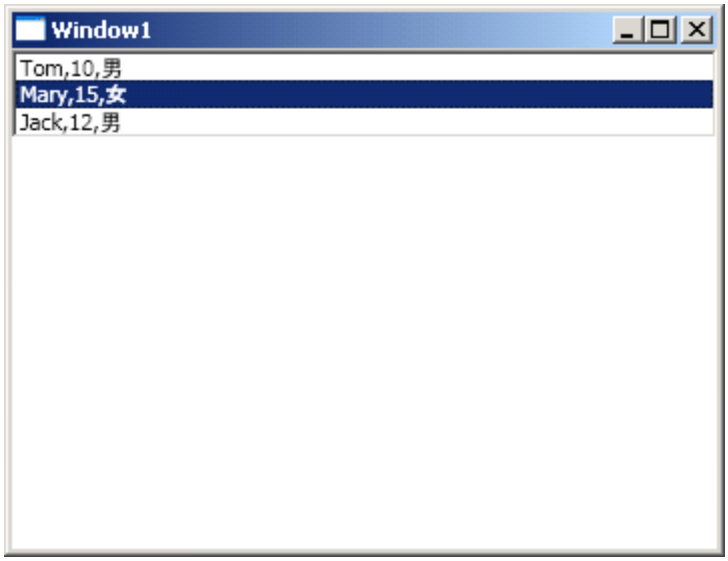
转换器很简单，只需实现 IValueConverter 接口即可。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <my:SexConverter x:Key="sexConverter" />

    <DataTemplate DataType="{x:Type my:Personal}">
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Path=Name}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Age}" />
        <TextBlock>,</TextBlock>
        <TextBlock Text="{Binding Path=Sex, Converter={StaticResource
sexConverter}}" />
      </StackPanel>
    </DataTemplate>
```

```
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

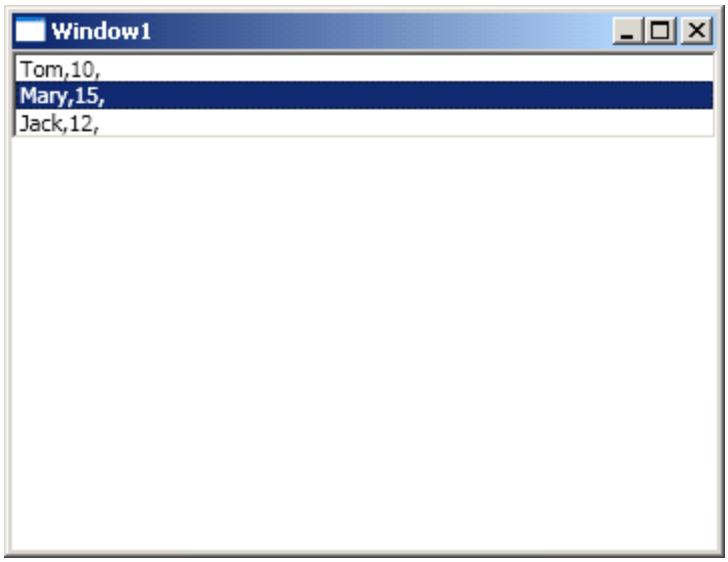
首先在资源中创建一个转换器实例，然后在数据模板中使用 Binding.Converter 来指定转换器实例即可。看看最终效果。



Convert() 有个有趣的的返回结果 "Binding.DoNothing"，它的意思是 "暂时取消该绑定"。

```
public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
{
  //return value.ToString() == "Male" ? "男" : "女";
  return Binding.DoNothing;
}
```

注意，DoNothing 和 null 并不是一回事，null 是个有效返回值。



接下来，我们试着将 Sex 转换成 Brush 类型，以便显示不同的颜色。

Window1.xaml.cs

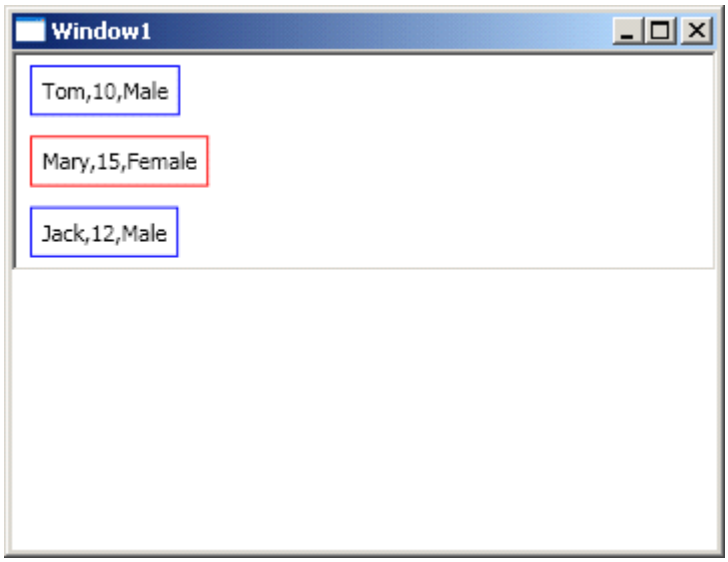
```
public class SexToBrushConverter : IValueConverter
{
  public object Convert(object value, Type targetType, object parameter,
CultureInfo culture)
  {
    return value.ToString() == "Male" ? Brushes.Blue : Brushes.Red;
  }

  public object ConvertBack(object value, Type targetType, object parameter,
CultureInfo culture)
  {
    throw new NotImplementedException();
  }
}
```

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1" Height="276" Width="360"
WindowStartupLocation="CenterScreen">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="10" Sex="Male" />
```

```
      <my:Personal Name="Mary" Age="15" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <my:SexToBrushConverter x:Key="sexToBrushConverter" />

    <DataTemplate DataType="{x:Type my:Personal}">
      <Border x:Name="border1"
          BorderBrush="{Binding Path=Sex, Converter={StaticResource
sexToBrushConverter}}"
          BorderThickness="1" Padding="5" Margin="5">

        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding Path=Name}" />
          <TextBlock>,</TextBlock>
          <TextBlock Text="{Binding Path=Age}" />
          <TextBlock>,</TextBlock>
          <TextBlock Text="{Binding Path=Sex}" />
        </StackPanel>

      </Border>
    </DataTemplate>
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

注意 Border.BorderBrush 属性中的转换器用法，结果表明转换器达到了上面例子中数据模板触发器同样的效果。



8. 集合视图

当绑定到一个集合对象时，WPF 总是默认提供一个视图 (CollectionViewSource)。视图会关联到源集合上，并自动将相关的操作在目标对象上显示出来。

(1) 排序

向 CollectionViewSource.SortDescriptions 属性中插入一个或多个排序条件 (SortDescription) 即可实现单个或多个条件排序。

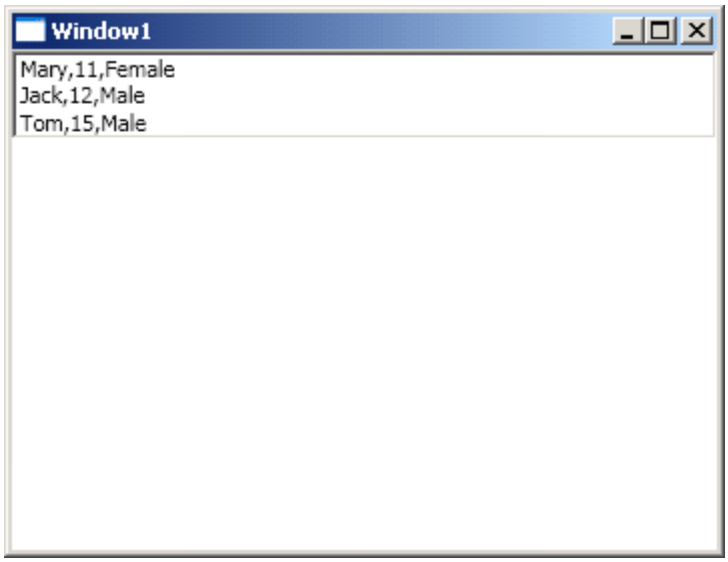
Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="15" Sex="Male" />
      <my:Personal Name="Mary" Age="11" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
```

```
            <TextBlock>,</TextBlock>
            <TextBlock Text="{Binding Path=Age}" />
            <TextBlock>,</TextBlock>
            <TextBlock Text="{Binding Path=Sex}" />
          </StackPanel>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </StackPanel>
 </Grid>
</Window>
```

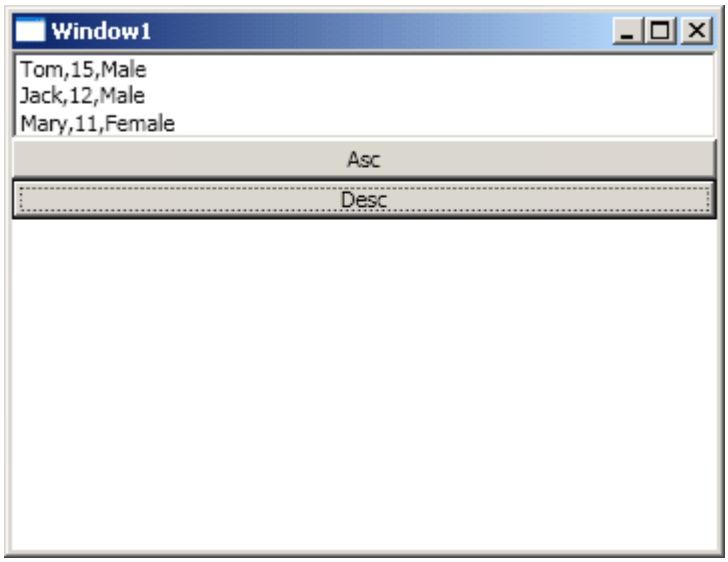
Window1.xaml.cs

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

    var personals = this.FindResource("personals");
    var view = CollectionViewSource.GetDefaultView(personals);
    view.SortDescriptions.Add(new SortDescription("Age",
ListSortDirection.Ascending));
  }
}
```



对 CollectionViewSource.SortDescriptions 的修改会直接反应在界面显示上。

```
protected void ButtonClick(object sender, RoutedEventArgs e)
{
  var personals = this.FindResource("personals");
  var view = CollectionViewSource.GetDefaultView(personals);
  var direction = sender == btnDesc ? ListSortDirection.Descending :
ListSortDirection.Ascending;

  view.SortDescriptions.Clear();
  view.SortDescriptions.Add(new SortDescription("Age", direction));
}
```



当然，我们可以直接在 XAML 中设置，而不是编写程序代码。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  xmlns:model="clr-namespace:System.ComponentModel;assembly=WindowsBase"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="15" Sex="Male" />
      <my:Personal Name="Mary" Age="11" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <CollectionViewSource x:Key="cvs" Source="{StaticResource personals}">
      <CollectionViewSource.SortDescriptions>
        <model:SortDescription PropertyName="Age" />
```

```
        <model:SortDescription PropertyName="Sex" Direction="Descending" />
      </CollectionViewSource.SortDescriptions>
    </CollectionViewSource>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource cvs}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

需要注意的地方包括：

- 引入了 xmlns:model="clr-namespace:System.ComponentModel;assembly=WindowsBase" 命名空间。
- 使用 CollectionViewSource 在资源中定义视图排序条件，注意使用 Source 属性绑定到 personals 资源。
- 目标对象数据源(DataContext)绑定到视图(cvs)而不是数据源(personals)。

(2) 分组

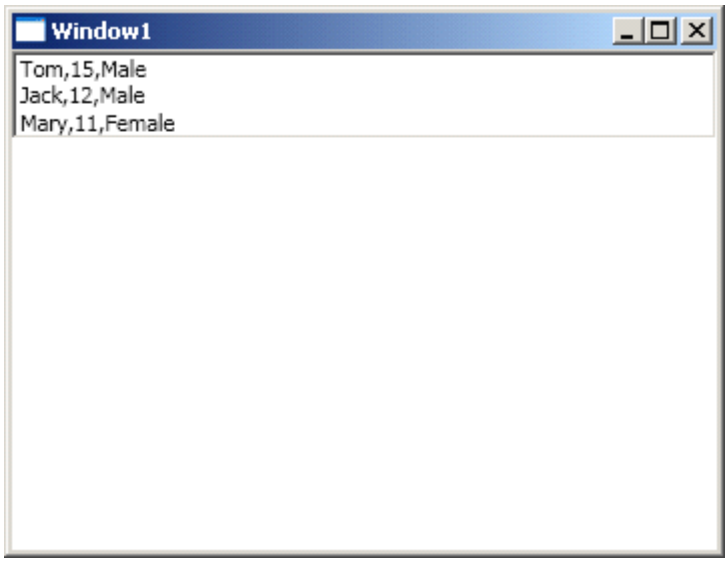
CollectionViewSource.GroupDescriptions 属性用来控制对数据源进行分组。

Window1.xaml.cs

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

    var personals = this.FindResource("personals");
    var view = CollectionViewSource.GetDefaultView(personals);
```

```
    view.GroupDescriptions.Add(new PropertyGroupDescription("Sex"));
  }
}
```

按性别进行分组，只是输出结果没啥直观效果。



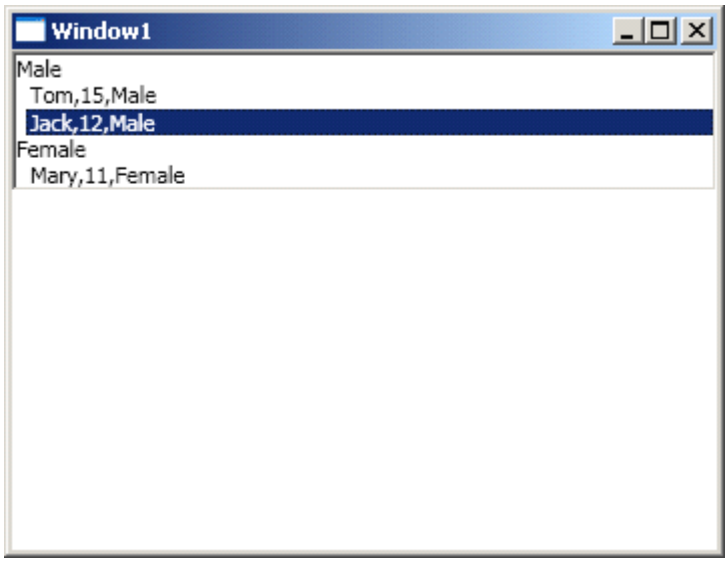
要看到效果，我们还必须为 ListBox 添加一个 GroupStyle，基于一贯偷懒的理由，我们可以直接使用系统内置的 GroupStyle.Default。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="15" Sex="Male" />
      <my:Personal Name="Mary" Age="11" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">

        <ListBox.GroupStyle>
          <x:Static Member="GroupStyle.Default"/>
        </ListBox.GroupStyle>
```

```
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

这回看上去好多了。



当然，GroupDescriptions 同样也可以写在 XAML 里。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  xmlns:data="clr-namespace:System.Windows.Data;assembly=PresentationFramewo
rk"
  Title="Window1">
```

```
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="15" Sex="Male" />
      <my:Personal Name="Mary" Age="11" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
    </my:PersonalList>

    <CollectionViewSource x:Key="cvs" Source="{StaticResource personals}">
      <CollectionViewSource.GroupDescriptions>
        <data:PropertyGroupDescription PropertyName="Sex" />
      </CollectionViewSource.GroupDescriptions>
    </CollectionViewSource>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource cvs}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
        <ListBox.GroupStyle>
          <x:Static Member="GroupStyle.Default"/>
        </ListBox.GroupStyle>
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

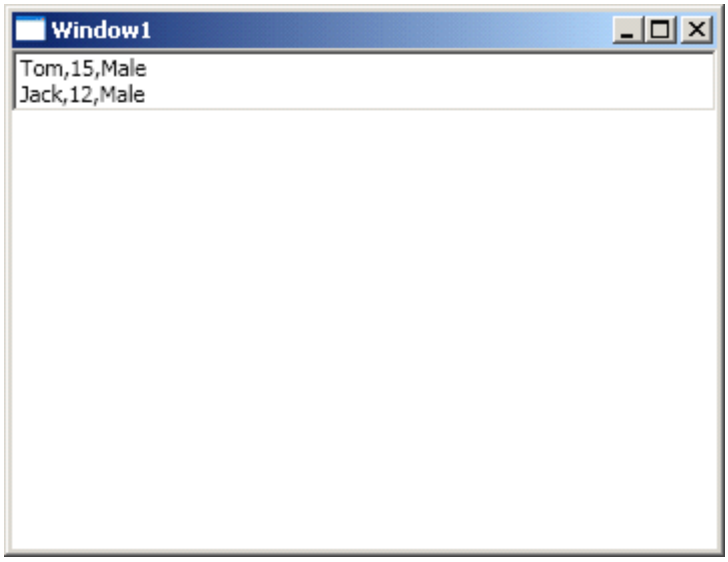
注意引入 xmlns:data="clr-namespace:System.Windows.Data;assembly=PresentationFramework" 命名空间。

(3) 过滤

利用 CollectionViewSource.Filter 委托属性，我们可以对数据源做出过滤处理。比如过滤掉全部女性。

Window1.xaml.cs

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

    var personals = this.FindResource("personals");
    var view = CollectionViewSource.GetDefaultView(personals);

    view.Filter = o =>
    {
      return (o as Personal).Sex != Sex.Female;
    };
  }
}
```



(MSDN 文档好像对不上)

(4) 导航

这个功能很常用，尤其是在数据库系统开发中。不过需要注意的是我们必须确保 IsSynchronizedWithCurrentItem = true。

Window1.xaml

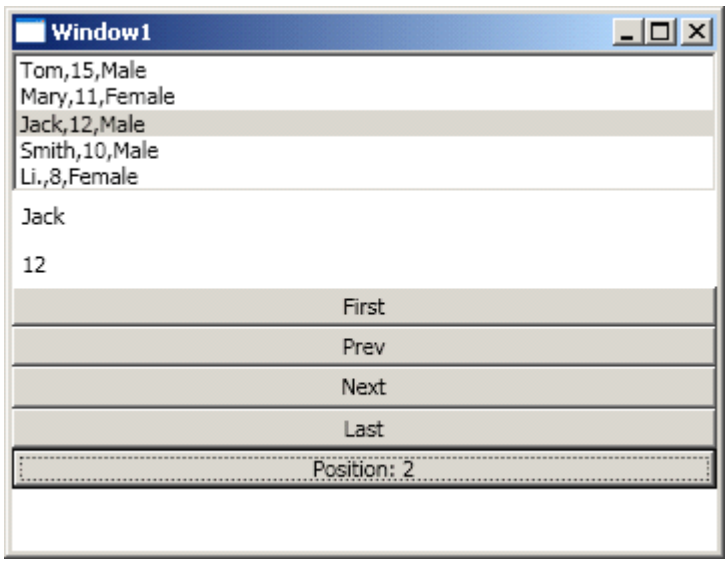
```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
```

```
  Title="Window1">
  <Window.Resources>
    <my:PersonalList x:Key="personals" >
      <my:Personal Name="Tom" Age="15" Sex="Male" />
      <my:Personal Name="Mary" Age="11" Sex="Female" />
      <my:Personal Name="Jack" Age="12" Sex="Male" />
      <my:Personal Name="Smith" Age="10" Sex="Male" />
      <my:Personal Name="Li." Age="8" Sex="Female" />
    </my:PersonalList>
  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}"
IsSynchronizedWithCurrentItem="True" >
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>

      <Label Content="{Binding /Name}" />
      <Label Content="{Binding /Age}"/>

      <Button x:Name="btnFirst" Click="ButtonClick" Content="First" />
      <Button x:Name="btnPrev" Click="ButtonClick" Content="Prev" />
      <Button x:Name="btnNext" Click="ButtonClick" Content="Next" />
      <Button x:Name="btnLast" Click="ButtonClick" Content="Last" />
      <Button x:Name="btnPostion" Click="ButtonClick" Content="Position: 2"
Tag="2" />
    </StackPanel>
  </Grid>
</Window>
```

Window1.xaml.cs

```
public partial class Window1 : Window
{
  public Window1()
```

```
  {
    InitializeComponent();
  }

  protected void ButtonClick(object sender, RoutedEventArgs e)
  {
    var personals = this.FindResource("personals") as PersonalList;
    var view = CollectionViewSource.GetDefaultView(personals);

    if (sender == btnFirst)
      view.MoveCurrentToFirst();
    else if (sender == btnPrev && view.CurrentPosition > 0)
      view.MoveCurrentToPrevious();
    else if (sender == btnNext && view.CurrentPosition < personals.Count - 1)

      view.MoveCurrentToNext();
    else if (sender == btnLast)
      view.MoveCurrentToLast();
    else if (sender == btnPostion)
      view.MoveCurrentToPosition(Convert.ToInt32(btnPostion.Tag));
  }
}
```



这很有趣，或许你也注意到了 Label 的 Binding 语法。

`<Label Content="{Binding /}" />` 表示绑定到当前选择项。

`<Label Content="{Binding /Name}" />` 表示绑定到当前选择项的 Name 属性。

10. 数据提供程序

(1) XmlDataProvider

XmlDataProvider 允许我们直接将 XML 数据作为数据源，我们将前面章节的例子改成 XML 数据岛试试，注意此时我们已经不需要在代码中定义 Personal、PersonalList 类型。

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>

    <XmlDataProvider x:Key="personals" XPath="Personals">
      <x:XData>
        <Personals xmlns="">
          <Personal Name="Tom" Age="15" Sex="Male" />
          <Personal Name="Mary" Age="11" Sex="Female" />
          <Personal Name="Jack" Age="12" Sex="Male" />
        </Personals>
      </x:XData>
    </XmlDataProvider>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding XPath=*}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding XPath=@Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

在资源中定义 XML 数据岛，注意 "Personals xmlns" 不能省略，另外采用 XPath 进行了绑定操作 (XPath 的语法可参考 MSDN 文档)。除了使用数据岛，我们还以使用 XML 数据文件。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  Title="Window1">
  <Window.Resources>

    <XmlDataProvider x:Key="personals"
Source="pack://siteOfOrigin:,,,/Personals.xml"
      XPath="Personals" />

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding XPath=*}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding XPath=@Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

Personals.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Personals xmlns="">
  <Personal Name="Tom" Age="15" Sex="Male" />
  <Personal Name="Mary" Age="11" Sex="Female" />
  <Personal Name="Jack" Age="12" Sex="Male" />
</Personals>
```

在 Source 属性中指定 XML Uri。

当然，我们也可以在程序代码中通过 XmlDocument 来控制 XML 数据源。

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Window1">
  <Window.Resources>

    <XmlDataProvider x:Key="personals" />

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding XPath=*}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding XPath=@Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding XPath=@Sex}" />
            </StackPanel>
          </DataTemplate>
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

Window1.xaml.cs

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

 var xml = new XmlDocument();
    xml.Load("Personals.xml");

    var provider = this.FindResource("personals") as XmlDataProvider;
    provider.Document = xml;
    provider.XPath = "Personals";
  }
}
```

逻辑代码只需修改 XmlDocument 即可自动同步显示到界面上。

```
protected void ButtonClick(object sender, RoutedEventArgs e)
{
  var provider = this.FindResource("personals") as XmlDataProvider;
  var xml = provider.Document;

  var mary = xml.SelectSingleNode("Personals/Personal[@Name=\"Mary\"]") as
XmlElement;
  var age = Convert.ToInt32(mary.Attributes["Age"].Value);

  mary.Attributes["Age"].Value = (++age).ToString();
}
```

- 如果设置了 Source 属性，则放弃所有内联 XML 数据；如果设置了 Document 属性，则清除 Source 属性并放弃所有内联 XML 数据。
- 设置以下属性将隐式导致此 XmlDataProvider 对象刷新：Source、Document、XmlNamespaceManager 和 XPath。
- 在更改多个导致刷新的属性时，建议使用 DeferRefresh。

(2) ObjectDataProvider

ObjectDataProvider 比我们直接绑定对象有如下三个好处：

- 可以在 XAML 申明中使用构造参数。
- 绑定到源对象的方法上。
- 支持异步数据绑定。

我们先看看构造参数的使用。

Window1.xaml.cs

```
enum Sex
{
  Male,
  Female
}

class Personal
{
  public string Name { get; private set; }
  public int Age { get; private set; }
  public Sex Sex { get; private set; }

  public Personal(string name, int age, Sex sex)
  {
    this.Name = name;
    this.Age = age;
    this.Sex = sex;
```

```
  }
}

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

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  Title="Window1">
  <Window.Resources>

    <ObjectDataProvider x:Key="personal" ObjectType="{x:Type my:Personal}">
      <ObjectDataProvider.ConstructorParameters>
        <sys:String>Tom</sys:String>
        <sys:Int32>15</sys:Int32>
        <my:Sex>Male</my:Sex>
      </ObjectDataProvider.ConstructorParameters>
    </ObjectDataProvider>

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personal}">
      <Label Content="{Binding Name}" />
      <Label Content="{Binding Age}" />
      <Label Content="{Binding Sex}" />
    </StackPanel>
  </Grid>
</Window>
```

接下来，我们尝试绑定到一个方法上。

Window1.xaml.cs

```
class PersonalList : ObservableCollection<Personal>
{
  public PersonalList GetPersonals()
```

```
  {
    this.Add(new Personal("Tom", 15, Sex.Male));
    this.Add(new Personal("Mary", 11, Sex.Female));
    this.Add(new Personal("Jack", 13, Sex.Male));

    return this;
  }
}

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

Window1.xaml

```
<Window x:Class="Learn.WPF.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:my="clr-namespace:Learn.WPF"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  Title="Window1">
  <Window.Resources>

    <ObjectDataProvider x:Key="personals" ObjectType="{x:Type
my:PersonalList}"
      MethodName="GetPersonals" />

  </Window.Resources>
  <Grid>
    <StackPanel DataContext="{StaticResource personals}">
      <ListBox x:Name="listbox1" ItemsSource="{Binding}">
        <ListBox.ItemTemplate>
          <DataTemplate>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Age}" />
              <TextBlock>,</TextBlock>
              <TextBlock Text="{Binding Path=Sex}" />
            </StackPanel>
          </DataTemplate>
```

```
        </ListBox.ItemTemplate>
      </ListBox>
    </StackPanel>
  </Grid>
</Window>
```

和构造方法参数一样，我们也可以向方法提供参数。

Window1.xaml.cs

```
class PersonalList : ObservableCollection<Personal>
{
  public IEnumerable<Personal> GetPersonals(int top)
  {
    this.Add(new Personal("Tom", 15, Sex.Male));
    this.Add(new Personal("Mary", 11, Sex.Female));
    this.Add(new Personal("Jack", 13, Sex.Male));

    return this.Take(top);
  }
}
```

Window1.xaml

```
<ObjectDataProvider x:Key="personals" ObjectType="{x:Type my:PersonalList}"
MethodName="GetPersonals">
  <ObjectDataProvider.MethodParameters>
    <sys:Int32>2</sys:Int32>
  </ObjectDataProvider.MethodParameters>
</ObjectDataProvider>
```

Silverlight - Hello, World!

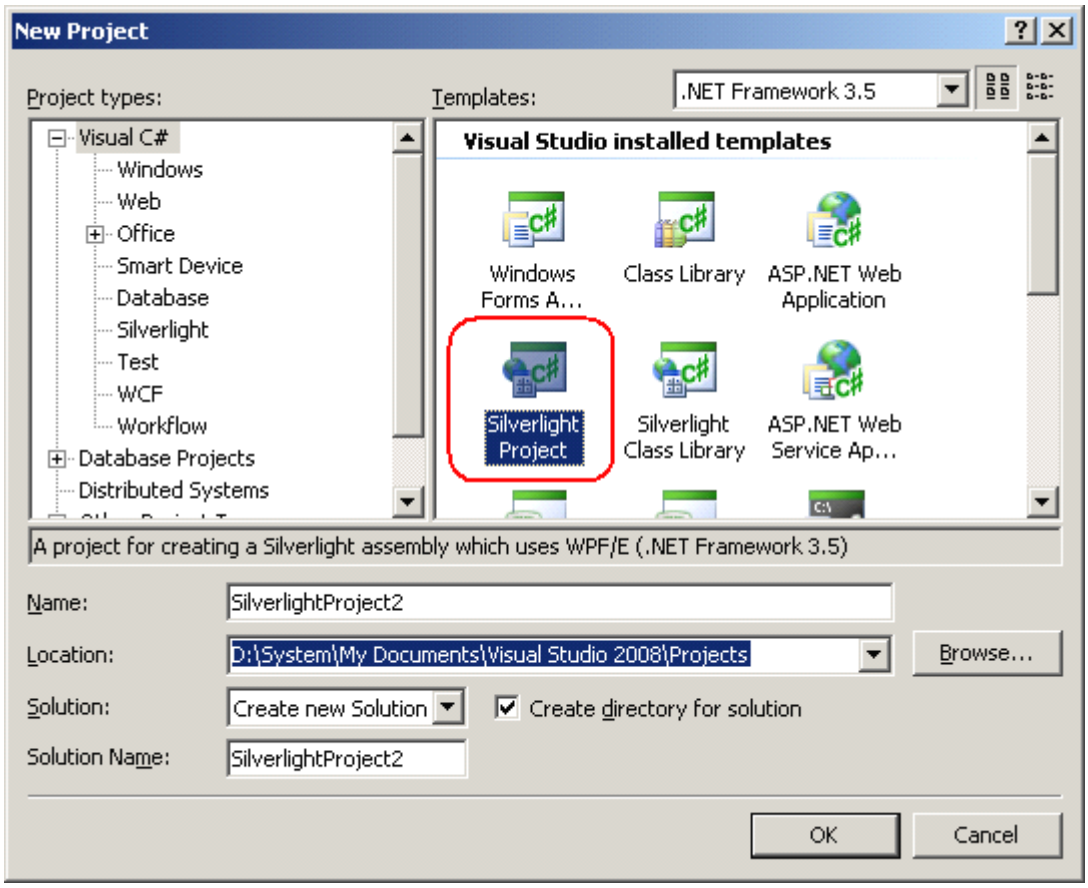
Silverlight 热闹好长时间了，只是我对 UI 这块的东西没啥感觉，所以迟迟不曾接触。今天不知哪根筋出了问题，居然还是绕到这个东东上..... 写个 "Hello, World!" 试试看吧，毕竟对新技术的关注也算是我的本职工作。

首先要下载安装的东西包括：

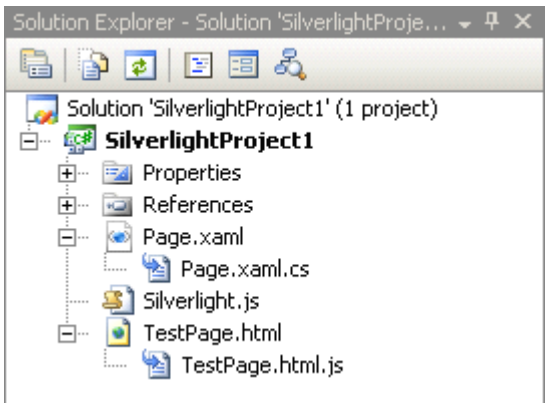
Microsoft Visual Studio 2008 Beta2
Microsoft Silverlight Tools Alpha for Visual Studio 2008 Beta 2
Microsoft Silverlight 1.1 Alpha Software Development Kit (可选，包含文档和演示)
Microsoft Silverlight 1.1 Alpha (用来在浏览器显示 Silverlight 的插件，相当于 Flash.ocx)

1. 安装 Silverlight 1.1 Alpha 和 Silverlight Tools。

2. 在 VS2008 中创建 Silverlight Project。



3. 创建完成以后，会在 Solution Explorer 看到如下一些文件。



- TestPage.html：用来显示 Silverlight Control 的页面。
- TestPage.html.js：包含用来创建 Silverlight Control 对象的 JavaScipt Function。
- Silverlight.js：由 TestPage.html.js 调用。
- Page.xaml：我们要工作的地方，感觉有点像 ASP.NET UserControl。

4. 打开 Page.xaml，加入一个 TextBlock 用来显示 "Hello, World!"。

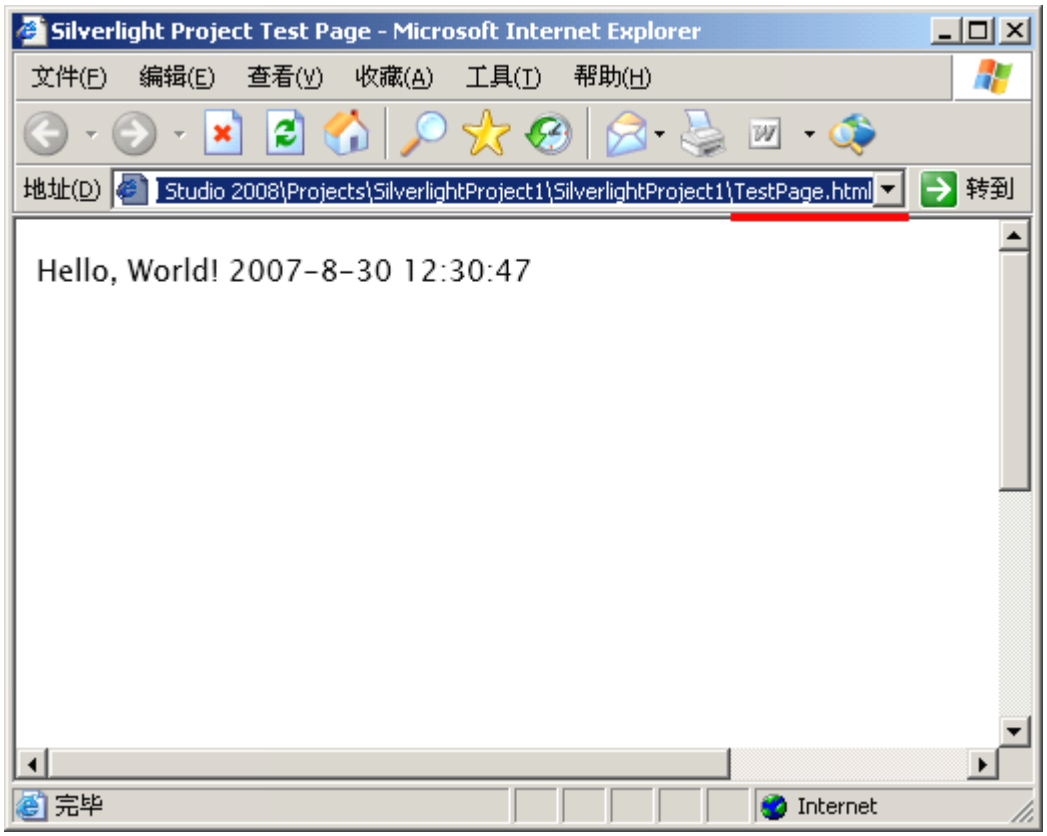
Silverlight.js | TestPage.html.js | TestPage.html | Page.xaml.cs | **Page.xaml**

```
<Canvas x:Name="parentCanvas"
        xmlns="http://schemas.microsoft.com/client/2007"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Loaded="Page_Loaded"
        x:Class="SilverlightProject1.Page;assembly=ClientBin/SilverlightProject1.dll"
        Width="640"
        Height="480"
        Background="White"
        >
    <TextBlock Name="Text1"></TextBlock>
</Canvas>
```

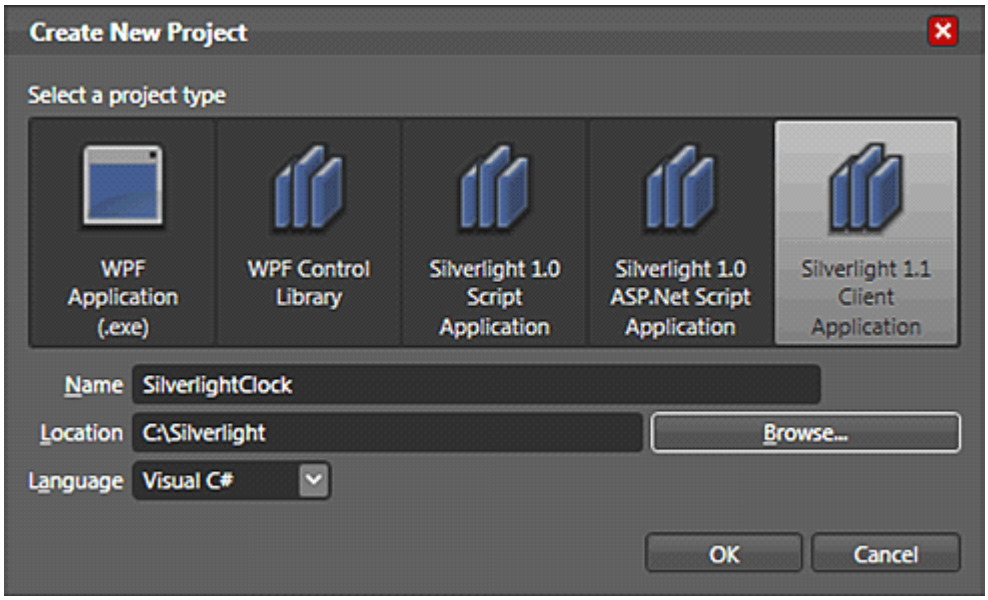
5. 按 F7 打开 Page.xaml.cs，加入代码。

```
namespace SilverlightProject1
{
    public partial class Page : Canvas
    {
        public void Page_Loaded(object o, EventArgs e)
        {
            // Required to initialize variables
            InitializeComponent();

            (this.FindName("Text1") as TextBlock).Text =
                "Hello, World! " + DateTime.Now.ToString();
        }
    }
}
```

6. 将 TestPage.html 设为 Start Page，按 F5 看看效果。



OK~~~ 感觉还行。不过要创建好看的界面，还得用专业点的工具 —— Microsoft Expression Blend，否则手工写 XAML 会死人的。



相关资源

Silverlight

Microsoft Silverlight Dev Center

Silverlight Downloads

Silverlight 1.1 Alpha

Microsoft Silverlight Tools Alpha for Visual Studio 2008 Beta 2

Microsoft Silverlight 1.1 Alpha Software Development Kit (SDK)

Micorosft Expression