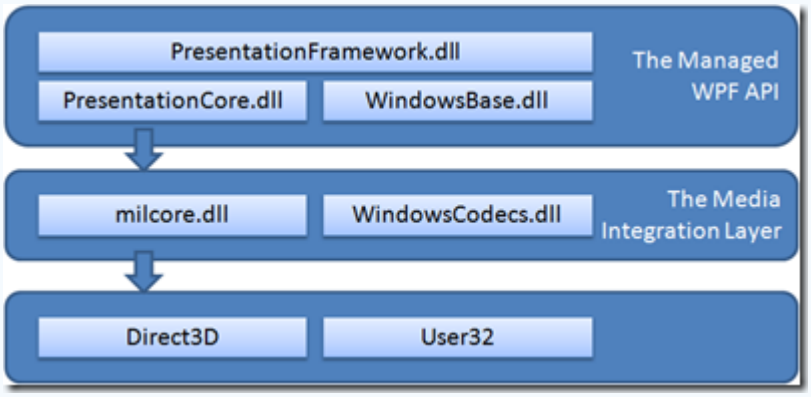
目录

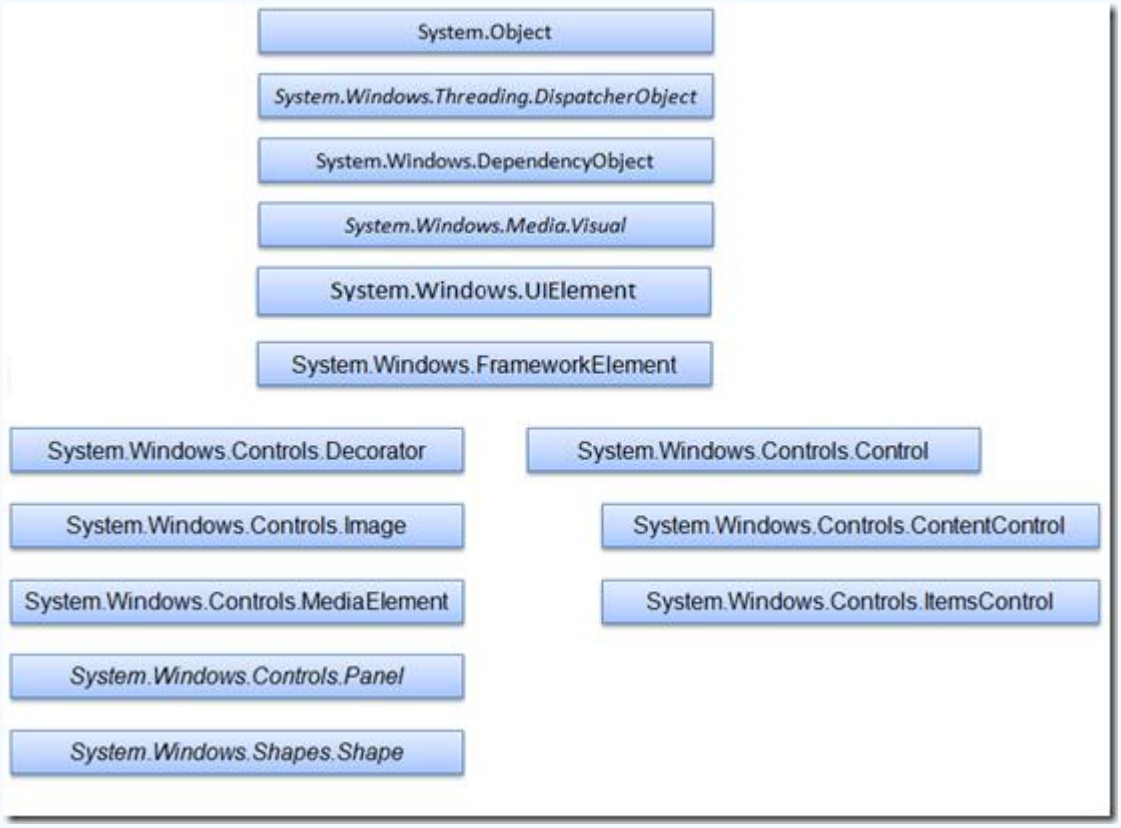
WPF 体系结构

WPF 的基础的体系结构，所引用的 Assembly 如下图所示：



关系图中的 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，可实现高效的硬件和软件呈现。WPF 还要求对内存和执行进行精确控制。milcore 中的组合引擎受性能影响关系大，需要放弃 CLR 的许多优点来提高性能。

在 WPF 中常用的的控件类继承结构如下图所示：



System.Object 类：在.Net 中所有类型的根类型

System.Windows.Threading.DispatcherObject 类：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。

System.Windows.DependencyObject 类：表示一个参与依赖项属性系统的对象。

System.Windows.Media.Visual 类：为 WPF 中的呈现提供支持，其中包括命中测试、坐标转换和边界框计算。

System.Windows.UIElement 类：UIElement 是 WPF 核心级实现的基类，该类建立在 Windows Presentation Foundation (WPF) 元素和基本表示特征基础上。

System.Windows.FrameworkElement 类：为 Windows Presentation Foundation (WPF) 元素提供 WPF 框架级属性集、事件集和方法集。此类表示附带的 WPF 框架级实现，它是基于由 UIElement 定义的 WPF 核心级 API 构建的。

System.Windows.Controls.Control 类：表示 用户界面 (UI) 元素的基类，这些元素使用 ControlTemplate 来定义其外观。

System.Windows.Controls.ContentControl 类：表示包含单项内容的控件。

System.Windows.Controls.ItemsControl 类：表示一个可用于呈现项的集合的控件。

System.Windows.Controls.Decorator 类：提供在单个子元素（如 Border 或 Viewbox）上或周围应用效果的元素的基类。

System.Windows.Controls.Image 类：表示显示图像的控件。

System.Windows.Controls.MediaElement 类：表示包含音频和/或视频的控件。

System.Windows.Controls.Panel 类：为所有 Panel 元素提供基类。使用 Panel 元素在 Windows Presentation Foundation (WPF) 应用程序中放置和排列子对象。

System.Windows.Sharps.Sharp 类：为 Ellipse、Polygon 和 Rectangle 之类的形状元素提供基类。

详细的有关 WPF 的控件类及其使用请参考 MSDN 文档。

WPF 应用程序管理

一、WPF 应用程序由 System.Windows.Application 类进行管理

无内容.

二、创建 WPF 应用程序

创建 WPF 应用程序有两种方式：

1、Visual Studio 和 Expression Blend 默认的方式，使用 App.xaml 文件定义启动应用程序

App.xaml 文件的内容大致如下：

```
1: <Application x:Class="WpfApplicationLifeCycle.App"
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     StartupUri="Window1.xaml">
5:     <Application.Resources>
6:     </Application.Resources>
7: </Application>
```

其中 StartupUri 指定启动的 WPF 窗体

2、可以自己定义类，定义 Main 方法实现对 WPF 应用程序的启动

在项目中添加一个类，类的代码如下，在项目选项中，设定此类为启动项。

```
1: using System;
2: using System.Windows;
3:
4: namespace WpfApplicationLifeCycle
5: {
6:     public class MainClass
7:     {
8:         [STAThread]
9:         static void Main()
10:         {
11:             // 定义 Application 对象
12:             Application app = new Application();
```

```
13:
14:             // 方法一：调用 Run 方法，参数为启动的窗体对象
15:             Window2 win = new Window2();
16:             app.Run(win);
17:
18:             // 方法二:指定Application对象的MainWindow属性为启动窗体，调用无参数的 Run 方法
19:             //Window2 win = new Window2();
20:             //app.MainWindow = win;
21:             //win.Show();           // 此处必须有 win.Show()，否则不能显示窗体
22:             //app.Run();
23:
24:             // 方法三:
25:             //app.StartupUri = new Uri("Window2.xaml", UriKind.Relative);
26:             //app.Run();
27:         }
28:     }
29: }
```

三、应用程序关闭

应用程序关闭时的策略由 **ShutdownMode** 属性指定，其类型为 **System.Windows.ShutdownMode** 枚举类型，其枚举成员有：

OnLastWindowClose（默认值）：当应用程序中的最后一个窗体关闭时或调用 **Application** 对象的 **Shutdown()**方法时，应用程序关闭;

OnMainWindowClose：当主窗体（即启动窗体）关闭时或调用 **Application** 对象的 **Shutdown()**方法时，应用程序关闭。（类似于 **C#**的 **Windows** 应用程序的关闭模式）;

OnExplicitShutdown：只有在调用 **Application** 对象的 **Shutdown()**方法时，应用程序才会关闭;

更改的时候，可以直接在 App.xaml 中更改：

```
<Application x:Class="WpfApplicationLifeCycle.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Window1.xaml"
    ShutdownMode="OnExplicitShutdown">
    <Application.Resources>
    </Application.Resources>
</Application>
```

也可以在代码文件（App.xaml.cs）中更改

```
Application app = new Application();
Window2 win = new Window2();

// 更改关闭模式必须要在调用 app.Run()方法之前
app.ShutdownMode = ShutdownMode.OnExplicitShutdown;
app.Run(win);
```

四、Application 对象的事件

名称	说明
Activated	当应用程序成为前台应用程序时发生。
Deactivated	当应用程序停止作为前台应用程序时发生。
DispatcherUnhandledException	在异常由应用程序引发但未进行处理时发生。
Exit	恰好在应用程序关闭之前发生，且无法取消。
FragmentNavigation	当应用程序中的导航器开始导航至某个内容片断时发生，如果所需片段位于当前内容中，则导航会立即发生；或者，如果所需片段位于不同内容中，则导航会在加载了源 XAML 内容之后发生。
LoadCompleted	在已经加载、分析并开始呈现应用程序中的导航器导航到的内容时发生。
Navigated	在已经找到应用程序中的导航器要导航到的内容时发生，尽管此时该内容可能尚未完成加载。
Navigating	在应用程序中的导航器请求新导航时发生。
NavigationFailed	在应用程序中的导航器在导航到所请求内容时出现错误的情况下发生。
NavigationProgress	在由应用程序中的导航器管理的下载过程中定期发生，以提供导航进度信息。

NavigationStopped	在调用应用程序中的导航器的 StopLoading 方法时发生，或者当导航器在当前导航正在进行期间请求了一个新导航时发生。
SessionEnding	在用户通过注销或关闭操作系统而结束 Windows 会话时发生。
Startup	在调用 Application 对象的 Run 方法时发生。

应用程序的事件处理可以：

1、在 App.xaml 中做事件的绑定，在 App.xaml.cs 文件中添加事件的处理方法

在 App.xaml 文件中：

```
1: <Application x:Class="WpfApplicationLifeCycle.App"
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     StartupUri="Window1.xaml"
5:     Startup="Application_Startup">
6:     <Application.Resources>
7:     </Application.Resources>
8: </Application>
```

在 App.xaml.cs 文件中：

```
1: using System.Windows;
2:
3: namespace WpfApplicationLifeCycle
4: {
5:     /// <summary>
6:     /// Interaction logic for App.xaml
7:     /// </summary>
8:     public partial class App : Application
9:     {
10:         private void Application_Startup(object sender, StartupEventArgs e)
11:         {
12:             // 定义应用程序启动时要处理的内容
13:         }
14:     }
15: }
```

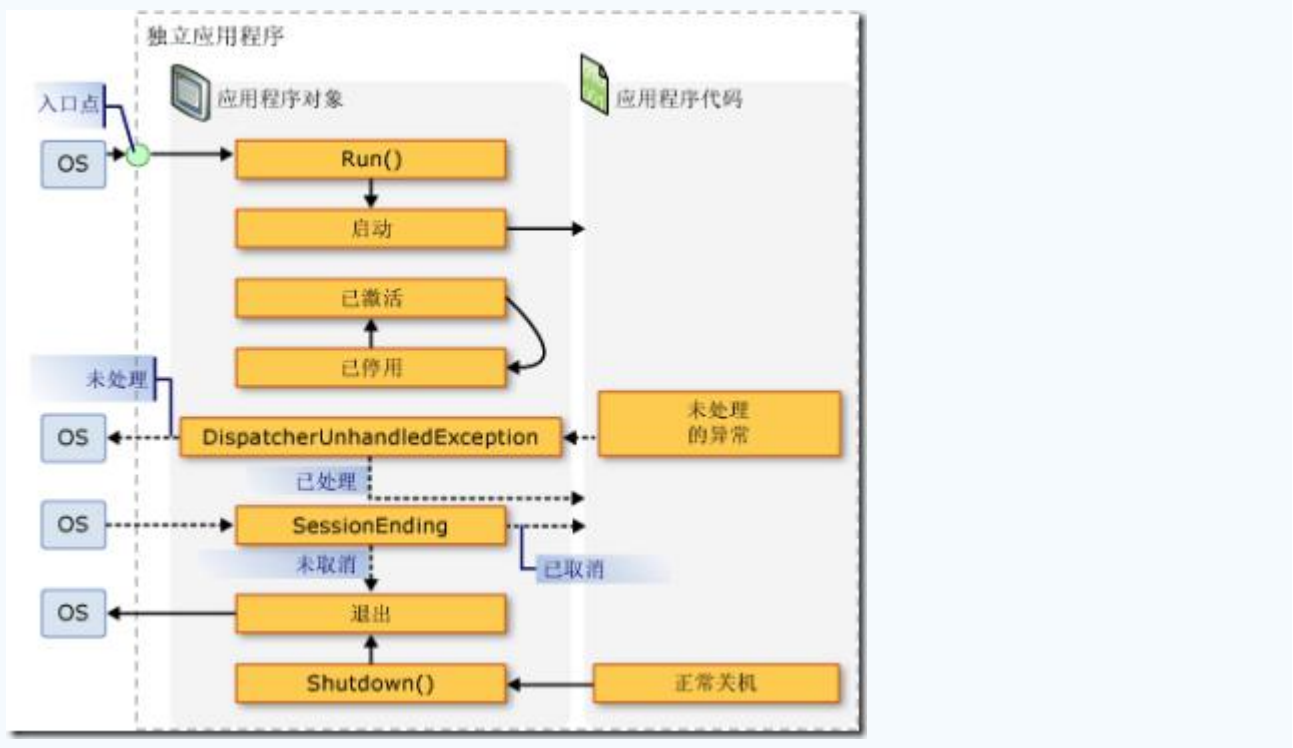
2、在自定义的类中可以做正常的 C#的事件绑定：

```
1: [STAThread]
2: static void Main()
3: {
```

```
4:      // 定义 Application 对象
5:      Application app = new Application();
6:      Window2 win = new Window2();
7:
8:      // 添加事件的绑定
9:      app.Startup += new StartupEventHandler(app_Startup);
10:
11:     app.Run(win);
12: }
13:
14: static void app_Startup(object sender, StartupEventArgs e)
15: {
16:     Window2 win = new Window2();
          win.Show();
          win.button1.Content = "YOU!";
17: }
```

主窗体载入时，会在 Window2 里面定义的 button1 上面就会显示 YOU!

五、WPF 应用程序生存周期



WPF 窗体

一、窗体类

在 Visual Studio 和 Expression Blend 中，自定义的窗体均继承 System.Windows.Window 类（类型化窗体）。定义的窗体由两部分组成：

1、XAML 文件

```
1: <Window
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     x:Class="WpfWindow.BasicWindow"
5:     x:Name="Window"
6:     Title="BasicWindow"
7:     Width="300" Height="200">
8:     <Canvas>
9:         <Button x:Name="btnMessage" Width="79" Height="24" Content="OK"
10:         Canvas.Left="172" Canvas.Top="93" Click="btnMessage_Click"/>
11:         <TextBox x:Name="txtValue" Width="215" Height="25"
12:         Canvas.Left="36" Canvas.Top="48" Text="" TextWrapping="Wrap"/>
13:     </Canvas>
14: </Window>
```

2、后台代码文件

```
1: using System;
2: using System.Windows;
3:
4: namespace WpfWindow
5: {
6:     public partial class BasicWindow : Window
7:     {
8:         public BasicWindow()
9:         {
10:             this.InitializeComponent();
11:         }
12:
```

```
  13:         private void btnMessage_Click(object sender, System.Win
dows.RoutedEventArgs e)
  14:         {
  15:             txtValue.Text = "Hello World";
  16:         }
  17:     }
  18: }
```

也可以将后台代码放在 XAML 文件中，上面的例子可以改写为：

```
   1: <Window
   2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen
tation"
   3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:     x:Class="WpfWindow.BasicWindow"
   5:     x:Name="Window"
   6:     Title="BasicWindow"
   7:     Width="300" Height="200">
   8:     <Canvas>
   9:         <Button x:Name="btnMessage" Width="79" Height="24" Cont
ent="OK"
  10:         Canvas.Left="172" Canvas.Top="93" Click="btnMessage_Cli
ck"/>
  11:         <x:Code><![CDATA[
  12:             void btnMessage_Click(object sender, System.Windows.
RoutedEventArgs e)
  13:             {
  14:                 txtValue.Text = "Hello World";
  15:             }
  16:         ]]>
  17:         </x:Code>
  18:         <TextBox x:Name="txtValue" Width="215" Height="25"
  19:         Canvas.Left="36" Canvas.Top="48" Text="" TextWrapping="
Wrap"/>(自动换行的意思)
  20:     </Canvas>
  21: </Window>
```

二、窗体的生存周期

1、显示窗体

构造方法 Show()、ShowDialog()方法：

Show()方法显示非模态窗口，ShowDialog()方法显示模态窗口 ；

Loaded 事件：窗体第一次 Show()或 ShowDialog()时引发的事件，通常在此事件中加载窗体的初始化数据 ；

2、关闭窗体

Close()方法：关闭窗体，并释放窗体的资源 ；

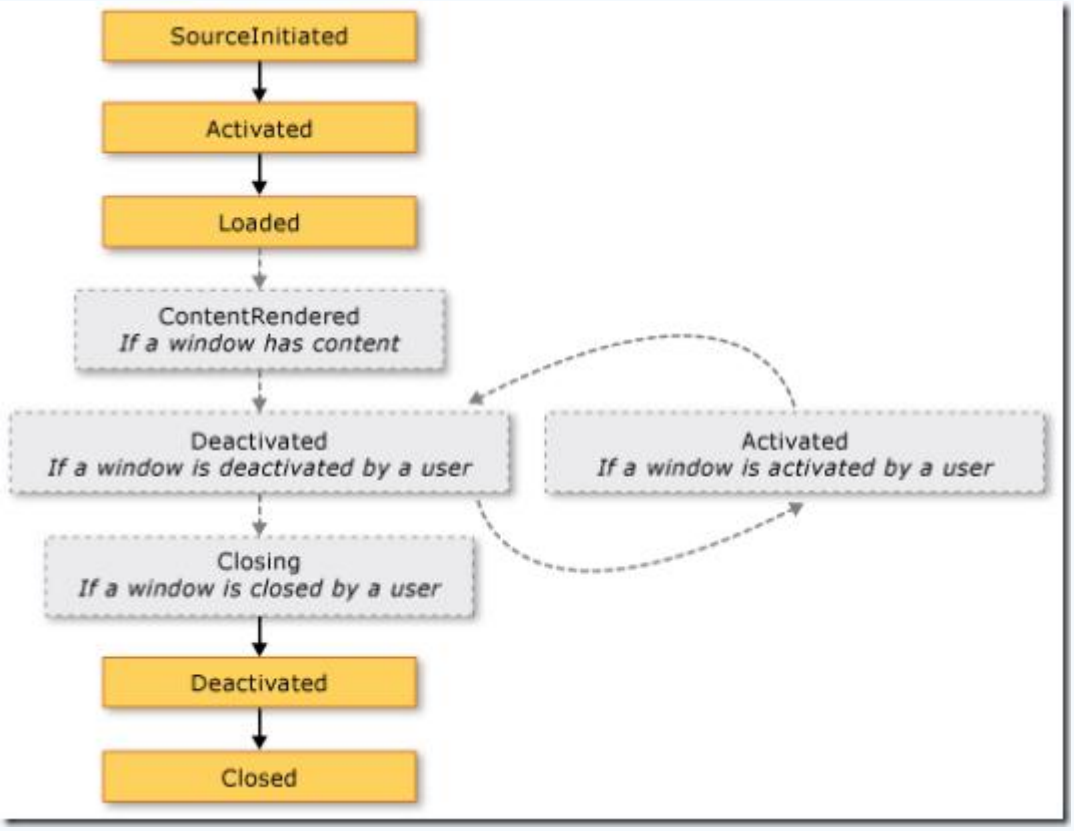
Closing 事件、Closed 事件：关闭时、关闭后引发的事件，通常在 Closing 事件中提示用户是否退出;

3、窗体的激活

Activate()方法：激活某窗体 ；

Activated、Deactivated 事件：当窗体激动、失去焦点时引发的事件 ；

4、窗体的生存周期



示例程序：

在窗体载入时显示当前日期，并开始播放媒体

当窗体失去焦点时暂停播放媒体

当窗体重新获得焦点时继承播放窗体

当点击窗体的关闭按钮时，询问用户是否退出应用程序

XAML 文件：

```
   1: <Window x:Class="WpfWindow.WindowLifeCycle"
   2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:     Title="WindowLifeCycle" Height="200" Width="300"
   5:         Loaded="Window_Loaded"
   6:         Activated="Window_Activated"
   7:         Deactivated="Window_Deactivated"
   8:         Closing="Window_Closing">
   9:     <Canvas>
  10:         <TextBlock Canvas.Right="15" Canvas.Bottom="15" Height="21" Name="txtDate"/>
  11:         <MediaElement Canvas.Left="89" Canvas.Top="12" Height="100" Width="100"
  12:                       Name="myMedia" Source="numbers.wmv"
  13:                       Stretch="Fill" LoadedBehavior="Manual" />
  14:     </Canvas>
  15: </Window>
```

代码文件：（事件的邦定也在代码中了，这与上面的重复了！）

```
   1: using System;
   2: using System.Windows;
   3:
   4: namespace WpfApplication1
{
    /// <summary>
    /// Window1.xaml 的交互逻辑
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
            this.Loaded+=new RoutedEventHandler(Window_Loaded);
            this.Deactivated+=new EventHandler(Window_Deactivated);
            this.Activated += new EventHandler(Window_Activated);
            this.Closing += new
```

```
System.ComponentModel.CancelEventHandler(Window_Closing);
        }
        private bool isPlaying;
        private void Window_Loaded(object sender, RoutedEventArgs e)
        {
            // 窗体加载时，显示当前日期及开始播放媒体
            myMedia.Source = new Uri("D:\\没有你的每一天.wma",
UriKind.Absolute);
            txtDate.Text = DateTime.Now.ToString("yyyy-MM-dd");
              myMedia.Play();
            isPlaying = true;
        }
        private void Window_Activated(object sender, EventArgs e)
          {
             // 如果窗体被激活，则继承播放媒体
              if (!isPlaying)
              {
                  myMedia.Play();
                  isPlaying = true;
              }
          }
        private void Window_Deactivated(object sender, EventArgs e)
          {
               // 如果窗体失去焦点，则暂停播放媒体
               if (isPlaying)
               {
                   myMedia.Pause();
                   isPlaying = false;
               }
          }
         private void Window_Closing(object sender,
System.ComponentModel.CancelEventArgs e)
          {
              // 点击窗体的“关闭”按钮，询问用户是否退出程序
              string message = "Quit the application?";
                string title = "System Information";
                MessageBoxButton button = MessageBoxButton.OKCancel;
```

```
                MessageBoxImage img = MessageBoxImage.Question;
                MessageBoxResult result = MessageBox.Show(
                    message, title, button, img);
                if (result == MessageBoxResult.Cancel)
                {
                    e.Cancel = true;    // 取消退出
                }
            }
        }
    }
```

三、其他窗体相关的属性、方法、事件

WPF 窗体的详细的属性、方法、事件请参考 MSDN，有很多的属性、方法、事件与 Windows 应用程序中 System.Windows.Forms.Form 类相同或近似,其中常用的一些属性、方法、事件有：

窗体边框模式（WindowStyle 属性）和是否允许更改窗体大小（ResizeMode 属性）；

窗体启动位置（属性）和启动状态（WindowState 属性） WindowStartupLocation;

窗体标题（Title 属性）；

始终在最前（TopMost 属性） ；

是否显示在任务栏（ShowInTaskbar）；

四、定义异形窗体

使用异形窗体，可以将窗体的背景设置为透明，边框设置为空，然后利用控件做出异形的窗体，例如：

XAML：

```
1: <Window x:Class="WpfWindow.CustomerWindow"
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="NonRectangularWindowSample" SizeToContent="WidthAndHeight"
5:     MouseLeftButtonDown="NonRectangularWindow_MouseLeftButtonDown"
6:     WindowStyle="None"
7:     AllowsTransparency="True"
8:     Background="Transparent">
9:     <Canvas Width="200" Height="200" >
10:        <Path Stroke="DarkGray" StrokeThickness="2">
11:            <Path.Fill>
12:                <LinearGradientBrush StartPoint="0.2,0" EndPoint="0.8,1" >
13:                    <GradientStop Color="White"  Offset="0"></GradientStop>
14:                    <GradientStop Color="White"  Offset="0.45"></GradientStop>
15:                    <GradientStop Color="LightBlue" Offset="0.9"></GradientStop>
16:                    <GradientStop Color="Gray" Offset="1"></GradientStop>
17:                </LinearGradientBrush>
18:            </Path.Fill>
19:            <Path.Data>
20:                <PathGeometry>
21:                    <PathFigure StartPoint="40,20" IsClosed="True">
22:                        <LineSegment Point="160,20"></LineSegment>
23:                        <ArcSegment Point="180,40" Size="20,20" SweepDirection="Clockwise"></ArcSegment>
```

```
 24:                     <LineSegment Point="180,80"></LineSegmen
t>
 25:                     <ArcSegment Point="160,100" Size="20,20"
SweepDirection="Clockwise"></ArcSegment>
 26:                     <LineSegment Point="90,100"></LineSegmen
t>
 27:                     <LineSegment Point="90,150"></LineSegmen
t>
 28:                     <LineSegment Point="60,100"></LineSegmen
t>
 29:                     <LineSegment Point="40,100"></LineSegmen
t>
 30:                     <ArcSegment Point="20,80" Size="20,20" Sw
eepDirection="Clockwise"></ArcSegment>
 31:                     <LineSegment Point="20,40"></LineSegment>
 32:                     <ArcSegment Point="40,20" Size="20,20" Sw
eepDirection="Clockwise"></ArcSegment>
 33:                 </PathFigure>
 34:             </PathGeometry>
 35:         </Path.Data>
 36:     </Path>
 37:     <Label Width="200" Height="120" FontSize="15" Horizonta
lContentAlignment="Center" VerticalContentAlignment="Center">Drag M
e</Label>
 38:     <Button Canvas.Left="155" Canvas.Top="30" Click="closeB
uttonRectangle_Click">
 39:         <Button.Template>
 40:             <ControlTemplate>
 41:                 <Canvas>
 42:                     <Rectangle Width="15" Height="15" Stroke=
"Black" RadiusX="3" RadiusY="3">
 43:                         <Rectangle.Fill>
 44:                             <SolidColorBrush x:Name="myAnimate
dBrush" Color="Red" />
 45:                         </Rectangle.Fill>
 46:                     </Rectangle>
 47:                     <Line X1="3" Y1="3" X2="12" Y2="12" Stroke
="White" StrokeThickness="2"></Line>
 48:                     <Line X1="12" Y1="3" X2="3" Y2="12" Stroke
="White" StrokeThickness="2"></Line>
 49:                 </Canvas>
 50:             </ControlTemplate>
 51:         </Button.Template>
 52:     </Button>
```

```
    </Canvas>
</Window>
```

代码文件：

```
using System.Windows;
using System.Windows.Input;

namespace WpfWindow
{
    public partial class CustomerWindow : Window
    {
        public CustomerWindow()
        {
            InitializeComponent();
        }

        void NonRectangularWindow_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
        {
            this.DragMove();
        }

        void closeButtonRectangle_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}
```

StackPanel、WrapPanel、DockPanel 容器

一、StackPanel

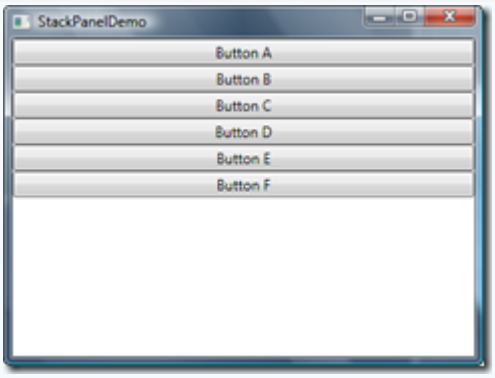
StackPanel 是以堆叠的方式显示其中的控件

1、可以使用 Orientation 属性更改堆叠的顺序

Orientation="Vertical"

默认，由上到下显示各控件。控件在未定义的前提下，宽度为 StackPanel 的宽度，高度自动适应控件中内容的高度

```
1: <StackPanel Orientation="Vertical">
2:     <Button>Button A</Button>
3:     <Button>Button B</Button>
4:     <Button>Button C</Button>
5:     <Button>Button D</Button>
6:     <Button>Button E</Button>
7:     <Button>Button F</Button>
8: </StackPanel>
```

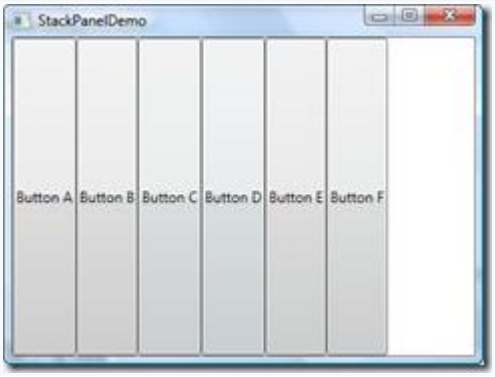


Orientation="Horizontal"

由左到右显示各控件。控件在未定义的前提下，高度为 StackPanel 的高度，宽度自动适应控件中内容的宽度

```
1: <StackPanel Orientation="Horizontal">
2:     <Button>Button A</Button>
3:     <Button>Button B</Button>
4:     <Button>Button C</Button>
5:     <Button>Button D</Button>
6:     <Button>Button E</Button>
```

```
    <Button>Button F</Button>
</StackPanel>
```



2、设置控件的属性，调整控件的显示

Margin 属性

定义控件的外边缘，可以通过以下几种方式来设置
1）Margin=”10”：各边缘均为 10
2）Margin=”10,20,30,40”：设定左、上、右、下各边缘分别为 10、20、30、40
3）使用拆分式方式设定，如上下为 10，左右为 20

```
<Button Content="Button A">
    <Button.Margin>
        <Thickness Top="10" Bottom="10" Left="20" Right="20" />
    </Button.Margin>
</Button>
```

Width、Height 属性
设定控件的宽度和高度，取消自动的宽度和高度

HorizontalAlignment、VerticalAlignment 属性

设定控件的水平或竖直对齐方式，如整体 Orientation="Vertical"的前提下，设置水平对齐为 Left、Right 或 Center，在没有设定宽度的情况下，控件的宽度自动调整

MinWidth、MinHeight、MaxWidth、MaxHeight 属性
在调整窗体大小，同时更改控件大小时，控件宽度、高度可变化的最大值和最小值

二、WrapPanel

以流的形式由左到右，由上到下显示控件，其功能类似于 Java AWT 布局中的 FlowLayout

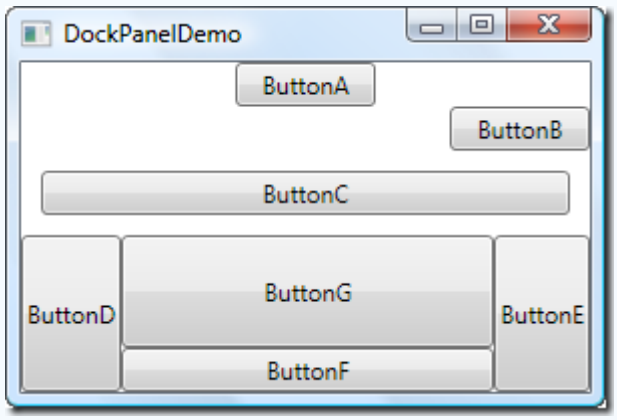
三、DockPanel

以上、下、左、右、中为基本结构的布局方式，类似于 Java AWT 布局中的 BorderLayout。

但与 BorderLayout 不同的是，每一个区域可以同时放置多个控件，在同一区域放置的多个控件采用的布局方式为 StackPanel 方式。

如：

```
   1: <DockPanel >
   2:     <Button Content="ButtonA" Width="70" DockPanel.Dock="Top"
/>
   3:     <Button Content="ButtonB" Width="70" HorizontalAlignment="
Right" DockPanel.Dock="Top" />
   4:     <Button Content="ButtonC" Margin="10" DockPanel.Dock="Top"
 />
   5:     <Button Content="ButtonD" DockPanel.Dock="Left" />
   6:     <Button Content="ButtonE" DockPanel.Dock="Right" />
   7:     <Button Content="ButtonF" DockPanel.Dock="Bottom" />
   8:     <Button Content="ButtonG" />
   9: </DockPanel>
```



Grid　UniformGrid 容器

一、Grid

Grid 是以表格形式组织控件的一种布局方式，与 Java AWT 中的 GridLayout 类似，但区别在于

WPF 中的 Grid 的每一个单元格中可以放置多个控件，但控件可能会层叠在一起

WPF 中的 Grid 支持单元格的合并，类似于 HTML 中的 table td 中的 row span 和 colspan

Grid 中的行和列可以自定义高度（Height）和宽度（Width）

在设置高度和宽度时可以采用两种写法：

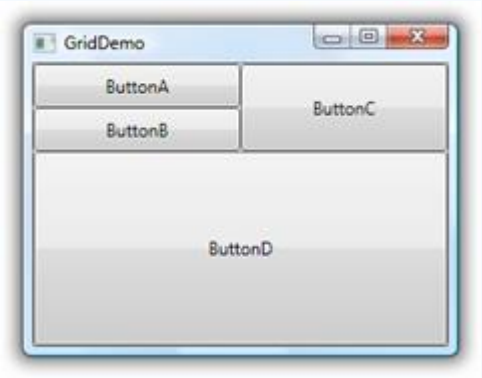
1）Height="60"：不加"星号"表示固定的高度

2）Height="60*"：加"星号"表示"加权"的高度，在调整窗体大小时，此高度或宽度会按窗体大小改变的比例进行缩放

如：

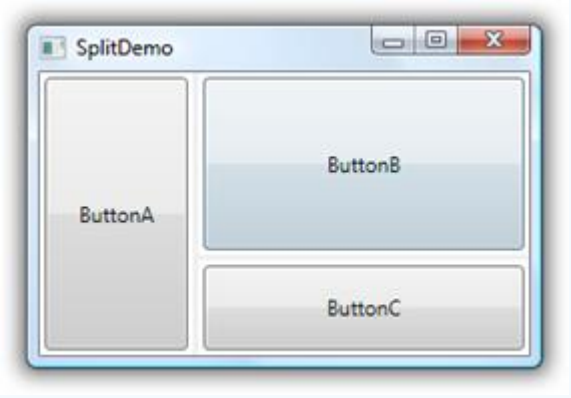
```
 1: <Grid>
 2:     <Grid.RowDefinitions>
 3:         <RowDefinition Height="60" />
 4:         <RowDefinition Height="202*" />
 5:     </Grid.RowDefinitions>
 6:     <Grid.ColumnDefinitions>
 7:         <ColumnDefinition/>
 8:         <ColumnDefinition/>
 9:     </Grid.ColumnDefinitions>
10:     <Button Grid.Column="0" Grid.Row="0" Height="30" VerticalAlignment="Top">ButtonA</Button>
11:     <Button Grid.Column="0" Grid.Row="0" Height="30" VerticalAlignment="Bottom">ButtonB</Button>
12:     <Button Grid.Column="1" Grid.Row="0">ButtonC</Button>
```

```
13:     <Button Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2">B
uttonD</Button>
14: </Grid>
```



二、使用 GridSplit 分割

可以使用 **GridSplit** 控件结合 **Grid** 控件实现类似于 **Windows** 应用程序中 **SplitContainer** 的功能，如下面的应用程序：



要实现以下的功能：

ButtonA 和 **ButtonB**、**ButtonC** 组成的整体，可以左右拖动，改变两者的宽度

ButtonB 和 **ButtonC** 可以上下拖动，改变两者的高度

实现以上功能的 **XAML** 代码如下：

```
1: <Grid>
2:     <Grid.ColumnDefinitions>
3:         <ColumnDefinition Width="88*" />
4:         <ColumnDefinition Width="Auto" />
5:         <ColumnDefinition Width="190*" />
6:     </Grid.ColumnDefinitions>
7:     <Grid.RowDefinitions>
8:         <RowDefinition Height="172*" />
```

```
 9:         <RowDefinition Height="Auto" />
10:         <RowDefinition Height="90*" />
11:     </Grid.RowDefinitions>
12:
13:     <Button Content="ButtonA" Margin="3" Grid.Row="0" Grid.Col
umn="0" Grid.RowSpan="3" />
14:     <Button Content="ButtonB" Margin="3" Grid.Row="0" Grid.Col
umn="2" />
15:     <Button Content="ButtonC" Margin="3" Grid.Row="2" Grid.Col
umn="2" />
16:
17:     <GridSplitter Width="3" HorizontalAlignment="Stre

tch" VerticalAlignment="Stretch"
18:                 Grid.Row="0" Grid.Column="1" Grid.RowSpan="3">
</GridSplitter>
19:     <GridSplitter Height="3" VerticalAlignment="Stretch" Horiz
ontalAlignment="Stretch"
20:                 Grid.Row="1" Grid.Column="2"></GridSplitter>
21: </Grid>
```

其核心想法为：

定义 **3*3** 的表格，其中放置分割线的列（下标为 **1**）和行（下标为 **1**）的宽度和高度设置为 **Auto**

ButtonA 放置在 **Row=0**、**Column=0**、**RowSpan=3** 的单元格中

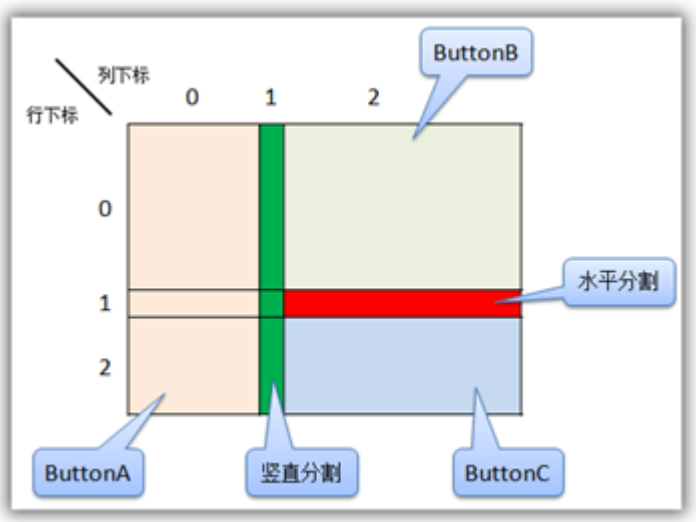
ButtonB 放置在 **Row=0**、**Column=2** 的单元格中

ButtonC 放置在 **Row=2**、**Column=2** 的单元格中

竖直分割线放置在 **Row=0**、**Column=1**、**ColSpan=3** 的单元格中

水平分割线放置在 **Row=1**、**Column=2** 的单元格中

如下图所示：

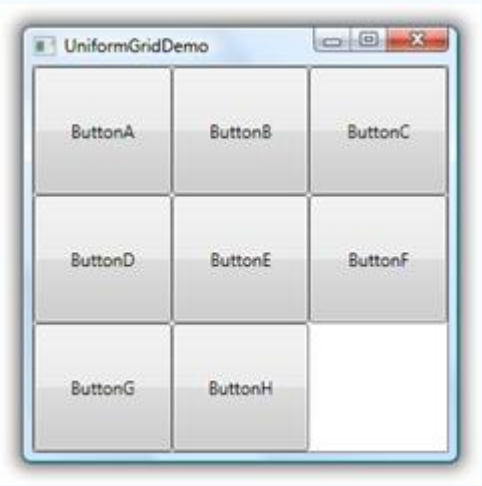


三、UniformGrid

UniformGrid 控件为控件提供了一种简化的网格布局。当控件添加到 **UniformGrid** 时，它们会排列在一个网格模式中，该网格模式会自动调整以使控件之间的距离保持均匀。单元格的数目将进行调整，以适应控件的数目。例如，如果四个控件添加到 **UniformGrid** 中，它们将安排在包含四个单元格的网格中。

如：

```
1: <UniformGrid>
2:     <Button Content="ButtonA" />
3:     <Button Content="ButtonB" />
4:     <Button Content="ButtonC" />
5:     <Button Content="ButtonD" />
6:     <Button Content="ButtonE" />
7:     <Button Content="ButtonF" />
8:     <Button Content="ButtonG" />
9:     <Button Content="ButtonH" />
10: </UniformGrid>
```



在使用 **UniformGrid** 的时候：

各单元格的大小完全相同

单元格的数量取决于放入的控件的数量，且单元格一定是行、列数相同的，即 **1*1、2*2、3*3、4*4...**的单元格分布

Canvas、InkCanvas 布局

一、Canvas

在 WPF 中子元素的绝对定位的布局控件

其子元素使用 Width、Height 定义元素的宽度和高度

使用 Convas.Left（Convas.Right）、Convas.Top（Convas.Bottom）定义与 Convas 容器的相对位置

如果同时存在 Convas.Left 和 Convas.Right、Convas.Top 和 Convas.Bottom，则 Convas.Left、Convas.Top 优先生效

例如：

```
1: <Canvas>
2:     <Button Canvas.Left="10" Canvas.Top="10" Height="23" Width="75">LT</Button>
3:     <Button Canvas.Right="10" Canvas.Top="10" Height="23" Width="75">RT</Button>
4:     <Button Canvas.Left="10" Canvas.Bottom="10" Height="23" Width="75">LB</Button>
5:     <Button Canvas.Right="10" Canvas.Bottom="10" Height="23" Width="75">RB</Button>
6: </Canvas>
```

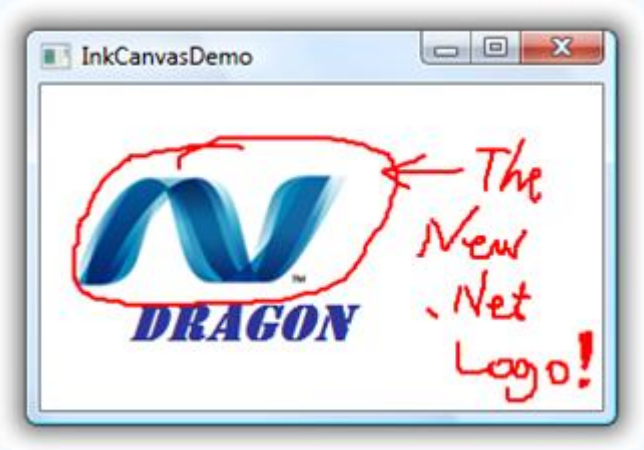
在调整窗体大小时，LT 与左、上距离保持不变；RT 与右、上距离保持不变；LB 与左、下距离保持不变；RB 与右、下距离保持不变。使用 Canvas 不能简单地实现 Windows 应用程序中 Acho 的功能。

二、InkCanvas

在 WPF 中实现允许使用墨迹的控件。如：

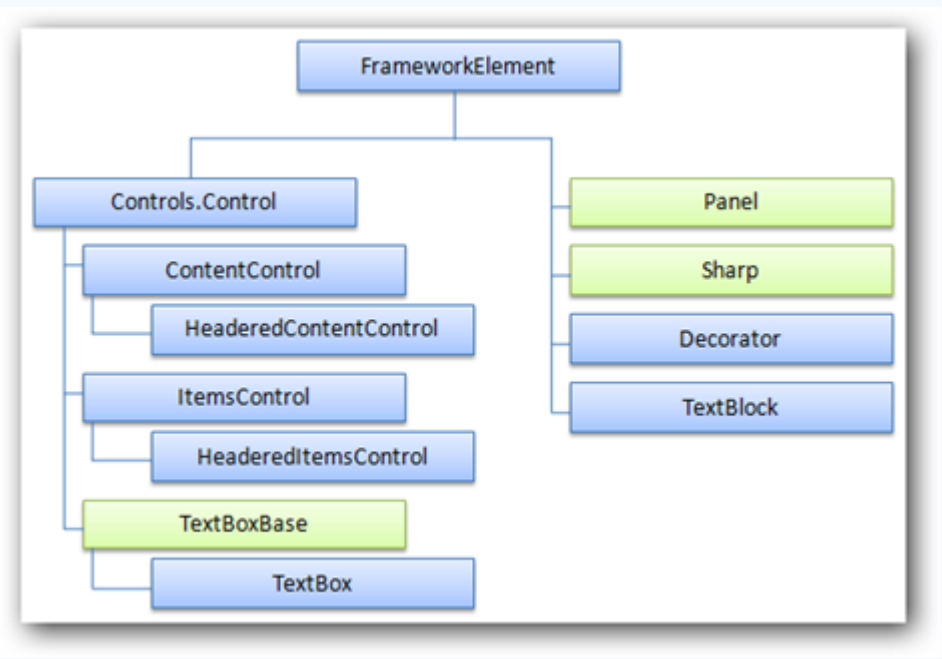
```
1: <Window x:Class="WPFLayoutDemo.InkCanvasDemo"
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
   3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:     Title="InkCanvasDemo" Height="200" Width="300">
   5:     <InkCanvas>
   6:         <InkCanvas.DefaultDrawingAttributes>
   7:             <DrawingAttributes Color="Red" />
   8:         </InkCanvas.DefaultDrawingAttributes>
   9:         <Image Width="155" Height="155" InkCanvas.Left="10" Ink
Canvas.Top="10"
  10:                Source="Logo2.png"/>
  11:     </InkCanvas>
  12: </Window>
```



其他的功能与 Canvas 相近。

WPF 对控件其类型的继承方式如下



（其中绿色表示的类是抽象类，蓝色表示的类是非抽象类）

控件内容模型

System.Windows.Controls.Control 类：表示 用户界面 (UI) 元素的基类，这些元素使用 ControlTemplate 来定义其外观

ContentControl：ContentControl 是一种包含一段内容的 Control

说明	ContentControl 是一种包含一段内容的 Control。
内容属性	Content
内容模型信息	控件内容模型概述
属于此类型系列的类型	Button, ButtonBase, CheckBox, ComboBoxItem, ContentControl, Frame, GridViewColumnHeader, GroupItem, Label, ListBoxItem, ListViewItem, NavigationWindow, RadioButton, RepeatButton, ScrollViewer, StatusBarItem, ToggleButton, ToolTip, UserControl, Window
可包含 ContentControl 类型的类型	ContentControl 类、HeaderedContentControl 类、ItemsControl 类、HeaderedItemsControl 类、Panel 类、Decorator 类和 Adorner 类

子类别	HeaderedContentControl 类

HeaderedContentControl：包含一段内容并具有 Header 的 ContentControl

说明	HeaderedContentControl 是一种包含一段内容并具有 Header 的 ContentControl。
内容属性	Content, Header
内容模型信息	控件内容模型概述
属于此类型系列的类型	Expander, GroupBox, HeaderedContentControl, TabItem
可包含 HeaderedContentControl 类型的类型	ContentControl 类、HeaderedContentControl 类、ItemsControl 类、HeaderedItemsControl 类、Panel 类、Decorator 类和 Adorner 类

ItemsControl：可包含多个项目（例如字符串、对象或其他元素）的 Control

说明	ItemsControl 是一种可包含多个项目（例如字符串、对象或其他元素）的 Control。
内容属性	Items, ItemsSource
主内容类型	多个项目，可以是字符串、对象或其他元素。
内容模型信息	控件内容模型概述
属于此类型系列的类型	Menu, MenuBase, ContextMenu, ComboBox, ItemsControl, ListBox, ListView, TabControl, TreeView, Selector, StatusBar
可包含 ItemsControl 类型的类型	ContentControl 类、HeaderedContentControl 类、ItemsControl 类、HeaderedItemsControl 类、Panel 类、Decorator 类和 Adorner 类
子类别	HeaderedItemsControl 类

HeaderedItemsControl：可包含多个项目（例如字符串、对象或其他元素）并具有标题的 ItemsControl

说明	HeaderedItemsControl 是一种 ItemsControl，可包含多个项目（例如字符串、对象或其他元素）并具有标题。
内容属性	Header, Items, ItemsSource
内容模型信息	控件内容模型概述
属于此类型系列的类型	HeaderedItemsControl, MenuItem, TreeViewItem, ToolBar
可包含 HeaderedItemsControl	通常，MenuItem 对象用作 Menu 元素的子元素；TreeViewItem 对象用作

类型的类型	TreeView 元素的子元素；ToolBar 对象用作 ToolBarTray 的子元素。

Panel 内容模型

System.Windows.Controls.Panel 抽象类：为所有 Panel 元素提供基类。使用 Panel 元素在 Windows Presentation Foundation (WPF) 应用程序中放置和排列子对象。

说明	Panel 是一种 FrameworkElement，它用于定位和排列子对象。
内容属性	Children
主内容类型	一个或多个 UIElement 对象。
内容模型信息	Panel 内容模型概述
属于此类型系列的类型	Canvas, DockPanel, Grid, TabPanel, ToolBarOverflowPanel, StackPanel, ToolBarPanel, UniformGrid, VirtualizingPanel, VirtualizingStackPanel, WrapPanel
可包含 Panel 类型的类型	ContentControl 类、HeaderedContentControl 类、ItemsControl 类、HeaderedItemsControl 类、Panel 类、Decorator 类和 Adorner 类

Sharp 模型

System.Windows.Sharps.Sharp 抽象类：为 Ellipse、Polygon 和 Rectangle 之类的形状元素提供基类

Decorator 模型

System.Windows.Controls.Decorator 类：提供在单个子元素（如 Border 或 Viewbox）上或周围应用效果的元素的基类

说明	Decorator 是一种 FrameworkElement，它将效果应用于单一子 UIElement 之上或周围。
内容属性	Child
主内容类型	单一 UIElement
内容模型信息	Decorator 内容模型概述
属于此类型系列的类型	ButtonChrome, ClassicBorderDecorator, ListBoxChrome, SystemDropShadowChrome, Border, InkPresenter, BulletDecorator, Viewbox, AdornerDecorator
可包含 Decorator 类型的类型	ContentControl 类、HeaderedContentControl 类、ItemsControl 类、HeaderedItemsControl 类、Panel 类、Decorator 类和 Adorner 类

TextBox 和 TextBlock

WPF 控件内容模型

WPF控件内容模型主要指派生于System.Windows.Controls.Control类的各种控件，其主要分为四部分：

ContentControl

HeaderedContendControl

ItemsControl

HeaderedItemsControl

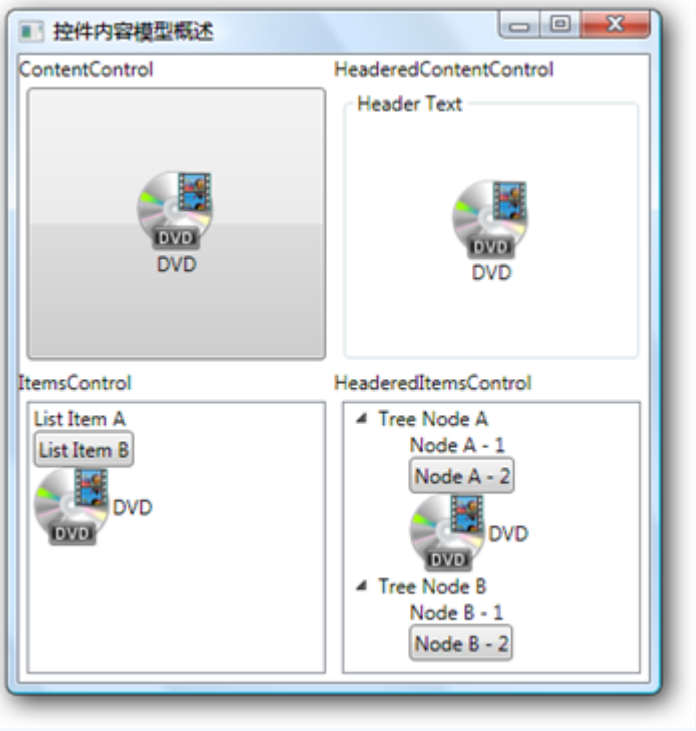
这四个类用作为 WPF 中大多数控件的基类。使用这些内容模型的类可以包含相同类型的内容，并以相同的方式处理该内容；可以放置在某个 ContentControl（或从 ContentControl 继承的类）中的任何类型的对象都可以放置在具有其他三个内容模型中的任何一个的控件中。如：

```
1: <Window x:Class="WPFControlContentModule.winOverView"
2:     xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="控件内容模型概述" Height="300" Width="400">
5:     <Grid>
6:         <Grid.ColumnDefinitions>
7:             <ColumnDefinition />
8:             <ColumnDefinition />
9:         </Grid.ColumnDefinitions>
10:        <Grid.RowDefinitions>
11:            <RowDefinition />
12:            <RowDefinition />
13:        </Grid.RowDefinitions>
14:
15:        <!--ContentControl 示例（Button）-->
16:        <TextBlock Text="ContentControl"  Grid.Row="0" Grid.Column="0"/>
17:        <Button Margin="5,20,5,5" Grid.Row="0" Grid.Column="0" >
18:            <Button.Content>
19:                <StackPanel VerticalAlignment="Center" HorizontalAlignment="Center">
```

```
 20:                     <Image Source="Images/DVD.png" Width="48" Hei
ght="48" />
 21:                     <TextBlock Text="DVD" HorizontalAlignment="Ce
nter" />
 22:                 </StackPanel>
 23:             </Button.Content>
 24:         </Button>
 25:
 26:         <!--HeaderedContentControl 示例（GroupBox）-->
 27:         <TextBlock Text="HeaderedContentControl"  Grid.Row="0"
Grid.Column="1"/>
 28:         <GroupBox  Margin="5,20,5,5" Grid.Row="0" Grid.Column="
1">
 29:             <GroupBox.Header>
 30:                 <TextBlock Text="Header Text" />
 31:             </GroupBox.Header>
 32:             <GroupBox.Content>
 33:                 <StackPanel VerticalAlignment="Center" Horizonta
lAlignment="Center">
 34:                     <Image Source="Images/DVD.png" Width="48" Hei
ght="48" />
 35:                     <TextBlock Text="DVD" HorizontalAlignment="Ce
nter" />
 36:                 </StackPanel>
 37:             </GroupBox.Content>
 38:         </GroupBox>
 39:
 40:         <!--ItemsControl 示例（ListBox）-->
 41:         <TextBlock Text="ItemsControl"  Grid.Row="1" Grid.Colum
n="0"/>
 42:         <ListBox  Margin="5,20,5,5" Grid.Row="1" Grid.Column="0
">
 43:             <ListBox.Items>
 44:                 <TextBlock Text="List Item A" />
 45:                 <Button Content="List Item B" />
 46:                 <StackPanel Orientation="Horizontal" VerticalAli
gnment="Center">
 47:                     <Image Source="Images/DVD.png" Width="48" Hei
ght="48" />
 48:                     <TextBlock Text="DVD" HorizontalAlignment="Ce
nter" VerticalAlignment="Center" />
 49:                 </StackPanel>
```

```
 50:             </ListBox.Items>
 51:         </ListBox>
 52:
 53:         <!--HeaderedItemsControl 示例（TreeView）-->
 54:         <TextBlock Text="HeaderedItemsControl"  Grid.Row="1" Grid.Column="1"/>
 55:         <TreeView Margin="5,20,5,5" Grid.Row="1" Grid.Column="1">
 56:             <TreeViewItem>
 57:                 <TreeViewItem.Header>
 58:                     Tree Node A
 59:                 </TreeViewItem.Header>
 60:                 <TreeViewItem.Items>
 61:                     <TextBlock Text="Node A - 1" />
 62:                     <Button Content="Node A - 2" />
 63:                     <StackPanel Orientation="Horizontal" VerticalAlignment="Center">
 64:                         <Image Source="Images/DVD.png" Width="48" Height="48" />
 65:                         <TextBlock Text="DVD" HorizontalAlignment="Center" VerticalAlignment="Center" />
 66:                     </StackPanel>
 67:                 </TreeViewItem.Items>
 68:             </TreeViewItem>
 69:             <TreeViewItem>
 70:                 <TreeViewItem.Header>
 71:                     Tree Node B
 72:                 </TreeViewItem.Header>
 73:                 <TreeViewItem.Items>
 74:                     <TextBlock Text="Node B - 1" />
 75:                     <Button Content="Node B - 2" />
 76:                 </TreeViewItem.Items>
 77:             </TreeViewItem>
 78:         </TreeView>
 79:     </Grid>
 80: </Window>
```

应用程序执行的结果如下：



一、ContentControl 模型

ContentControl 模型的类型具有一个 Content 属性。Content 属性的类型为 Object，因此，对于您在 ContentControl 中可以放置的内容没有任何限制。可以使用可扩展应用程序标记语言 (XAML) 或代码来设置 Content。

以下控件使用 ContentControl 内容模型：

Button、ButtonBase、CheckBox、ComboBoxItem、ContentControl、Frame、GridViewColumnHeader、GroupItem、Label、ListBoxItem、ListViewItem、NavigationWindow、RadioButton、RepeatButton、ScrollViewer、StatusBarItem、ToggleButton、ToolTip、UserControl、Window

在 Content 中只能放置一个控件（可以放置一个容器，然后再在容器中放置多个控件）。

严格地说，Content 的内容应该放置于<XXX.Content></XXX.Content>内部，但也可以省略此标记。如在按钮中放置一图片可以有以下几种写法：

```
1: <!--方法一-->
2: <Button Margin="5">
3:     <Button.Content>
4:         <Image Source="Images/DVD.png" Width="48" Height="48" />
5:     </Button.Content>
6: </Button>
7:
```

```
<!--方法二-->
<Button Margin="5">
    <Image Source="Images/DVD.png" Width="48" Height="48" />
</Button>

<!--如果是字符串，或者是数组绑定、资源引用还可以-->
<Button Margin="5" Content="Button Text" />
```

另外，还可以使用代码来为 ContentControl 指定相应的 Content 属性，如：

```
TextBlock date = new TextBlock();
date.Text = DateTime.Now.ToString("yyyy-MM-dd");

TextBlock time = new TextBlock();
time.Text = DateTime.Now.ToString("hh:mm:ss");

StackPanel panel = new StackPanel();
panel.Children.Add(date);
panel.Children.Add(time);

btn.Content = panel;
```

二、HeaderedContentControl 模型

HeaderedContentControl 类继承 ContentControl 类，表示带有 Header 的 ContentControl，其除了具有 ContentControl 的 Content 属性外，还具有一个 Header 属性，Header 的类型也是 Object 对象，与 Content 属性的用法类似。

从 HeaderedContentControl 继承的控件有：Expander、GroupBox、TabItem。

如定义一个带有图片和文字标题的 Expander：

```
<Window x:Class="WPFControlContentModule.winHeaderedContentControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Headered Content Control" Height="200" Width="300">
    <Canvas>
        <Expander Margin="5" HorizontalAlignment="Center">
            <Expander.Header>
                <StackPanel Orientation="Horizontal">
                    <Image Source="Images/Users.png" Width="32" Height="32" />
                    <TextBlock Text="User Info" VerticalAlignment="Center" />
```

```
  11:                 </StackPanel>
  12:             </Expander.Header>
  13:             <Expander.Content>
  14:                 <Grid>
  15:                     <Grid.RowDefinitions>
  16:                         <RowDefinition />
  17:                         <RowDefinition />
  18:                         <RowDefinition />
  19:                     </Grid.RowDefinitions>
  20:                     <Grid.ColumnDefinitions>
  21:                         <ColumnDefinition />
  22:                         <ColumnDefinition />
  23:                     </Grid.ColumnDefinitions>
  24:
  25:                     <TextBlock Text="UserName:" HorizontalAlignme
nt="Right"
  26:                                Grid.Row="0" Grid.Column="0" Margin
="3"/>
  27:                     <TextBlock Text="Tom" HorizontalAlignment="Le
ft"
  28:                                Grid.Row="0" Grid.Column="1" Margin
="3"/>
  29:                     <TextBlock Text="Gender:" HorizontalAlignment
="Right"
  30:                                Grid.Row="1" Grid.Column="0" Margin
="3"/>
  31:                     <TextBlock Text="Male" HorizontalAlignment="L
eft"
  32:                                Grid.Row="1" Grid.Column="1" Margin
="3"/>
  33:                     <TextBlock Text="Age:" HorizontalAlignment="R
ight"
  34:                                Grid.Row="2" Grid.Column="0" Margin
="3"/>
  35:                     <TextBlock Text="23" HorizontalAlignment="Lef
t"
  36:                                Grid.Row="2" Grid.Column="1" Margin
="3"/>
  37:                 </Grid>
  38:             </Expander.Content>
  39:         </Expander>
  40:     </Canvas>
  41: </Window>
```

三、ItemsControl 模型

从 ItemsControl 继承的控件包含一个对象集合。 ItemsControl 的一个示例是 ListBox。可以使用 ItemsSource 属性或 Items 属性来填充一个 ItemsControl。

1、使用 ItemSource 属性

使用 ItemSource 属性，需将其绑定到一个实现 IEnumerable 接口的类型的实例上，系统会枚举其成员做为 ItemsControl 的 Item。如在一个 ListBox 中列出系统的所有字体，并在每一项中显示字体的样式（类似于 Office 系列中的字体下拉菜单）：

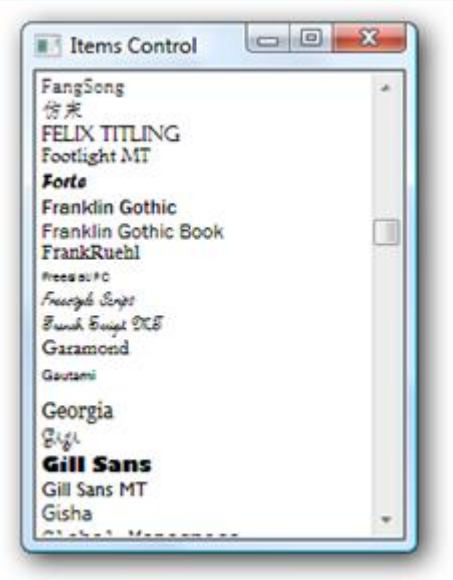
```
<Window x:Class="WPFControlContentModule.winItemsControl"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Items Control" Height="300" Width="300">
    <Grid>
        <ListBox x:Name="lstFont"/>
    </Grid>
</Window>
```

```
using System.Collections.Generic;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Markup;
using System.Windows.Media;

namespace WPFControlContentModule
{
    /// <summary>
    /// Interaction logic for winItemsControl.xaml
    /// </summary>
    public partial class winItemsControl : Window
```

```
14:     {
15:         public winItemsControl()
16:         {
17:             InitializeComponent();
18:
19:             // 设置绑定的数据源
20:             Binding binding = new Binding();
21:               binding.Source = Items;  ? ?
22:
23:             // 绑定
24:             lstFont.SetBinding(ListBox.ItemsSourceProperty, binding);
25:         }
26:
27:         private List<TextBlock> Items
28:         {
29:             get
30:             {
31:                 List<TextBlock> result = new List<TextBlock>();
32:
33:                 // 遍历系统的所有字体
34:                 foreach (FontFamily family in Fonts.SystemFontFamilies)
35:                 {
36:                     foreach (KeyValuePair<XmlLanguage, string> pair in family.FamilyNames)
37:                     {
```

```
38:                         TextBlock t = new TextBlock();
39:                         // 设置字体名称
40:                         t.Text = pair.Value;
41:
42:                         // 设置字体样式
43:                         t.FontFamily = family;
44:                         t.FontSize = 12;
45:
46:                         result.Add(t);
47:                     }
48:                 }
49:
50:                 // 返回一个 TextBlock 的控件对象集合
51:                 return result;
52:             }
53:         }
54:     }
55: }
```



2、使用 Items 属性

随 WPF 附带的每个 ItemsControl 具有一个对应的类，该类代表 ItemsControl 中的一个项。下表列出了随 WPF 附带的 ItemsControl 对象及其相应的项容器。

ItemsControl	项容器
ComboBox	ComboBoxItem

ContextMenu	MenuItem
ListBox	ListBoxItem
ListView	ListViewItem
Menu	MenuItem
StatusBar	StatusBarItem
TabControl	TabItem
TreeView	TreeViewItem

在做相应的项的时候可以有三种做法：

```
<!--方法一-->
<ListBox>
    <ListBox.Items>
        <ListBoxItem>
            Item A
        </ListBoxItem>
        <ListBoxItem>
            <TextBlock Text="Item B" />
        </ListBoxItem>
    </ListBox.Items>
</ListBox>

<!--方法二，省略 ListBox.Items-->
<ListBox>
    <ListBoxItem>
        Item A
    </ListBoxItem>
    <ListBoxItem>
        <TextBlock Text="Item B" />
    </ListBoxItem>
</ListBox>

<!--方法三，省略 ListBoxItem-->
<ListBox>
    <ListBox.Items>
        Item A
```

```
27:         <TextBlock Text="Item B" />
28:     </ListBox.Items>
29: </ListBox>
```

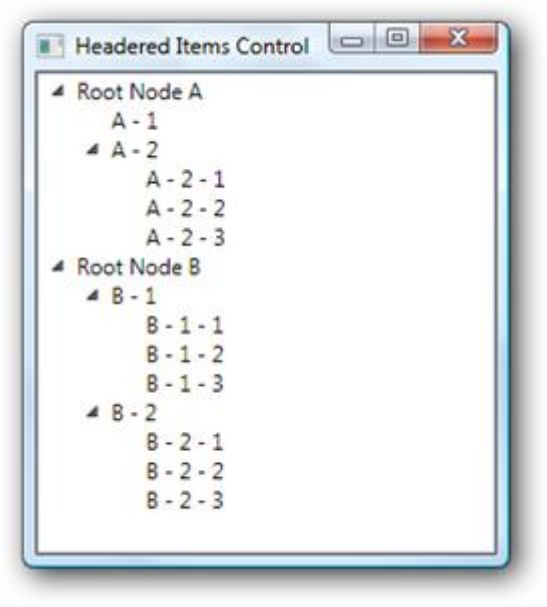
四、HeaderedItemsControl 模型

HeaderedItemsControl 从 ItemsControl 类继承。HeaderedItemsControl 定义 Header 属性，该属性遵从相同的规则，因为 HeaderedContentControl. WPF 的 Header 属性附带三个从 HeaderedItemsControl 继承的控件：MenuItem、ToolBar、TreeViewItem

HeaderedItemsControl 模型可以理解为如下结构：一个 HeaderedItemsControl 包含一个 Items 集合，每一个 Item 包含一个 Header 属性，一个子 Items 集合，以 TreeView 和 TreeViewItem 为例：

```
1: <TreeView>
2:     <TreeView.Items>
3:         <TreeViewItem>
4:             <TreeViewItem.Header>
5:                 <TextBlock Text="Root Node A" />
6:             </TreeViewItem.Header>
7:             <TreeViewItem.Items>
8:                 <TextBlock Text="A - 1" />
9:                 <TreeViewItem>
10:                    <TreeViewItem.Header>
11:                        <TextBlock Text="A - 2" />
12:                    </TreeViewItem.Header>
13:                    <TreeViewItem.Items>
14:                        <TextBlock Text="A - 2 - 1" />
15:                        <TextBlock Text="A - 2 - 2" />
16:                        <TextBlock Text="A - 2 - 3" />
17:                    </TreeViewItem.Items>
18:                </TreeViewItem>
19:            </TreeViewItem.Items>
20:        </TreeViewItem>
21:        <TreeViewItem>
22:            <TreeViewItem.Header>
23:                <TextBlock Text="Root Node B" />
24:            </TreeViewItem.Header>
25:            <TreeViewItem.Items>
26:                <TreeViewItem>
27:                    <TreeViewItem.Header>
28:                        <TextBlock Text="B - 1" />
29:                    </TreeViewItem.Header>
30:                    <TreeViewItem.Items>
```

```
31:                         <TextBlock Text="B - 1 - 1" />
32:                         <TextBlock Text="B - 1 - 2" />
33:                         <TextBlock Text="B - 1 - 3" />
34:                     </TreeViewItem.Items>
35:                 </TreeViewItem>
36:                 <TreeViewItem>
37:                     <TreeViewItem.Header>
38:                         <TextBlock Text="B - 2" />
39:                     </TreeViewItem.Header>
40:                     <TreeViewItem.Items>
41:                         <TextBlock Text="B - 2 - 1" />
42:                         <TextBlock Text="B - 2 - 2" />
43:                         <TextBlock Text="B - 2 - 3" />
44:                     </TreeViewItem.Items>
45:                 </TreeViewItem>
46:             </TreeViewItem.Items>
47:         </TreeViewItem>
48:     </TreeView.Items>
49: </TreeView>
```



Panel　Decorator TextBlock 内容模型

一、Panel 内容模型

Panel 内容模型指从 System.Windows.Controls.Panel 继承的控件，这些控件都是容器，可以在内部承载其他的控件和子容器。Panel 内容模型包含的容器有：

Canvas

DockPanel

Grid

TabPanel

ToolBarOverflowPanel

UniformGrid

StackPanel

ToolBarPanel

VirtualizingPanel

VirtualizingStackPanel

WrapPanel

对于 Panel 模型，其包含一个 Children 属性，表示其所有的子控件和子容器的集合，在 XAML 代码中可以省略<XXX.Children>标记

如：

```
<StackPanel x:Name="mainPanel">
    <StackPanel x:Name="panelA">
        <StackPanel.Children>
            <Button>Button A</Button>
        </StackPanel.Children>
    </StackPanel>
    <Button>Button B</Button>
    <StackPanel x:Name="panelB">
    </StackPanel>
</StackPanel>
```

也可以通过代码，动态添加 Children 中的对象

```
// 定义一个 Button
Button btn = new Button();
btn.Content = "Button C";

```

```
5: // 将 Button 添加到 StackPanel 中
6: panelB.Children.Add(btn);
```

二、Decorator 内容模型

Decorator 内容模型指的是从 System.Windows.Controls.Decorator 类继承的控件，主要是对其中的一个子元素的边缘进行修饰。Decorator 模型的主要控件包含：

AdornerDecorator

Border

InlineUIContainer

BulletDecorator

ButtonChrome

ClassicBorderDecorator

InkPresenter

ListBoxChrome

SystemDropShadowChrome

Viewbox

Decorator 模型包含一个 Child 属性，表示其包含的一个子元素（注意，只能是一个子元素（控件或容器，在容器中可以再添加其他的控件）），Child 属性的 XAML 标记可以省略。

例如，对于一个 TextBox 添加一个边框，使用 XAML 语言定义：

```
1: <StackPanel x:Name="mainPanel">
2:     <Border BorderThickness="5" BorderBrush="DarkBlue" Margin=
"5">
3:         <Border.Child>
4:             <TextBox Text="TextBox Content"/>
5:         </Border.Child>
6:     </Border>
7: </StackPanel>
```

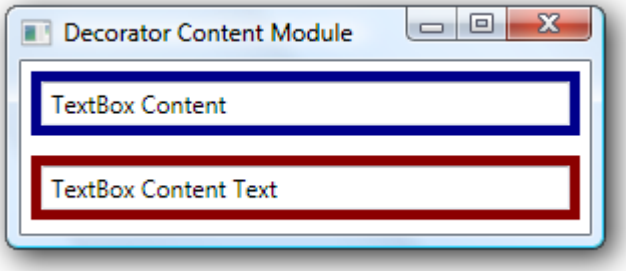
也可以使用代码完成上述功能：

```
1: // 定义一个 Border 对象，并设置其边框的大小，颜色，外边距
2: Border border = new Border();
3: border.BorderThickness = new Thickness(5);
4: border.BorderBrush = new SolidColorBrush(Colors.DarkRed);
5: border.Margin = new Thickness(5);
6:
7: // 定义一个 TextBox 对象
```

```
TextBox textBox = new TextBox();
textBox.Text = "TextBox Content Text";

// 使用 Border 修饰 TextBox 的边框
border.Child = textBox;

// 将 Border 添加到 StackPanel 中
mainPanel.Children.Add(border);
```



三、TextBlock 模型

TextBlock 模型实际上指的就是 System.Windows.Controls.TextBlock 类，它是一个用于显示少量流内容的轻量控件。其中包含一个 InLines 属性，支持 Inline 流内容元素的承载和显示。 支持的元素包括 AnchoredBlock、Bold（粗体字符串）、Hyperlink（超链接，在浏览器支持的模式下有效）、InlineUIContainer（承载其他控件的容器）、Italic（斜体字符串）、LineBreak（换行符）、Run（普通字符串）、Span（可以设置字体、颜色等的 Span） 和 Underline（下划线）。

例如：

```
<StackPanel Orientation="Horizontal">
    <Border BorderThickness="2" Margin="5" BorderBrush="Black
">
        <TextBlock Margin="5" TextWrapping="WrapWithOverflow">
            <TextBlock.Inlines>
                <Bold>
                    <Run>BlockText 控件 XAML 示例</Run>
                </Bold>
                <LineBreak/>
                <Run>TextBlock 支持以下的几种流显示样式：</Run>
                <LineBreak/>
                <Bold>粗体（Bold）</Bold>
```

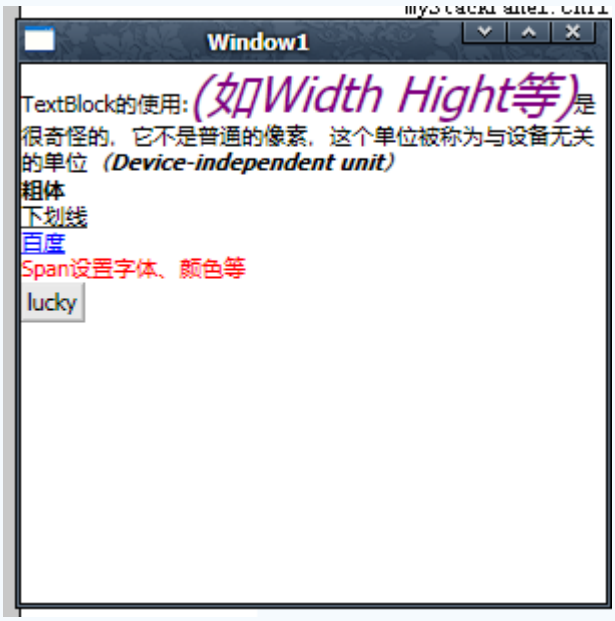
```
12:                 <LineBreak/>
13:                 <Italic>斜体（Italic）</Italic>
14:                 <LineBreak/>
15:                 <Underline>下划线（Underline）</Underline>
16:                 <LineBreak/>
17:                 <Hyperlink NavigateUri="http://www.microsoft.
com">超链接</Hyperlink>
18:                 <LineBreak/>
19:                 <Span Foreground="Red" FontSize="18">Span 设置字体、
颜色等</Span>
20:                 <LineBreak />
21:                 <InlineUIContainer>
22:                     <StackPanel Background="AntiqueWhite" Margin=
"5">
23:                         <TextBlock>Inline UI 容器</TextBlock>
24:                         <Button Content="按钮" Width="80" />
25:                     </StackPanel>
26:                 </InlineUIContainer>
27:             </TextBlock.Inlines>
28:         </TextBlock>
29:     </Border>
30:     <Border BorderThickness="2" Margin="5" BorderBrush="Black
">
31:         <TextBlock Margin="5" TextWrapping="WrapWithOverflow" x:
Name="textBlock">
32:             <TextBlock.Inlines>
33:                 <Bold>
34:                     <Run x:Name="title"></Run>
35:                 </Bold>
36:                 <LineBreak x:Name="line"/>
37:                 <InlineUIContainer x:Name="container">
38:                     <StackPanel Background="AntiqueWhite" Margin=
"5" x:Name="panel">
39:                     </StackPanel>
40:                 </InlineUIContainer>
41:             </TextBlock.Inlines>
```

```
42:         </TextBlock>
43:     </Border>
44: </StackPanel>
```

使用代码操作 Inlines：

```
            TextBlock myTextBlock = new TextBlock();
            myGrid.Children.Add(myTextBlock);
            myTextBlock.Inlines.Add("TextBlock的使用:");
            Italic myItalic = new Italic(new Run("(如Width Hight等)"));
            myItalic.FontSize = 24;
            myItalic.Foreground = Brushes.Purple;
            myTextBlock.Inlines.Add(myItalic);
            myTextBlock.Inlines.Add("是很奇怪的，它不是普通的像素，这个单位被称为与设备无关的单位");
            myTextBlock.TextWrapping = TextWrapping.WrapWithOverflow;
            Bold myBold = new Bold(new Italic(new Run(" (Device-independent unit)")));
            myTextBlock.Inlines.Add(myBold);
            myTextBlock.HorizontalAlignment = HorizontalAlignment.Stretch;
            myTextBlock.Inlines.Add(new LineBreak());
            Bold myBold1 = new Bold(new Run("粗体"));
            myTextBlock.Inlines.Add(myBold1);
            myTextBlock.Inlines.Add(new LineBreak());
            Underline myUnderline = new Underline(new Run("下划线"));
            myTextBlock.Inlines.Add(myUnderline);
            myTextBlock.Inlines.Add(new LineBreak());
            Hyperlink myHyperlink=new Hyperlink(new Run("百度"));
            myHyperlink.NavigateUri = new Uri("http://www.baidu.com");
            myTextBlock.Inlines.Add(myHyperlink);
            myTextBlock.Inlines.Add(new LineBreak());
            Span mySpan = new Span(new Run("Span设置字体、颜色等"));
            mySpan.Foreground = Brushes.Red;//或者改为：mySpan.Foreground=new SolidColorBrush(Colors.Red);
            myTextBlock.Inlines.Add(mySpan);
            myTextBlock.Inlines.Add(new LineBreak());
            InlineUIContainer myInlineUIContainer = new InlineUIContainer();
             myTextBlock.Inlines.Add(myInlineUIContainer);
            StackPanel myStackPanel=new StackPanel();
            myInlineUIContainer.Child = myStackPanel;
            Button myButton = new Button();
            myButton.Content = "lucky";
            myStackPanel.Children.Add(myButton);
```

(代码部分的执行结果)执行结果：



四、TextBox 模型

System.Windows.Controls.TextBox 类，实现的是可编辑的文本框，文本框的内容由字符串类型的 Text 属性指定，并且整个 TextBox 的内容使用共同的（即 TextBox 指定的）样式。

依赖项属性和路由事件

一、依赖项属性（Dependency Property）

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时 (CLR) 属性的功能。这些服务通常统称为 WPF 属性系统。由 WPF 属性系统支持的属性称为依赖项属性。本概述介绍 WPF 属性系统以及依赖项属性的功能，这包括如何在可扩展应用程序标记语言 (XAML) 中和代码中使用现有的依赖项属性。

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。

1、依赖项属性与 CLR 包装属性

以 Button 的 Backgroud 为例，设置或获取其值可以有以下几种方式：

XAML 文件中

```
1: <StackPanel>
2:     <!--
3:         在所生成的代码中，XAML 加载器将 XAML 属性的简单字符串值的
4:         类型转换为 WPF 类型（一种 Color，通过 SolidColorBrush）。
5:     -->
6:     <Button Margin="3" Background="Yellow" Content="Button A"
/>
7:
8:     <!--
9:         使用嵌套元素的方式，设置 Button.Backgroud 的值
10:     -->
11:     <Button Margin="3" Content="Button B" x:Name="btn_ButtonB
">
12:         <Button.Background>
13:             <SolidColorBrush Color="Gold" />
14:         </Button.Background>
15:     </Button>
16:
17:     <!--预留给代码使用的控件-->
```

```
 18:     <Button Margin="3" Content="Button C" x:Name="btn_ButtonC"
/>
 19:     <Button Margin="3" Content="Button D" x:Name="btn_ButtonD"
/>
 20:     <TextBox Margin="3" x:Name="txt_Value1" />
 21:     <TextBox Margin="3" x:Name="txt_Value2" />
 22: </StackPanel>
```

代码文件中：

```
  1: // 通过包装的属性设置按钮的背景颜色
  2: btn_ButtonC.Background = new SolidColorBrush(Colors.Red);
  3:
  4: // 通过依赖性属性的 SetValue 设置按钮的背景颜色
  5: SolidColorBrush brush = new SolidColorBrush(Colors.Blue);
  6: btn_ButtonD.SetValue(
  7:     Button.BackgroundProperty, brush);
  8:
  9: // 通过包装的属性获取 ButtonB 的背景颜色
 10: SolidColorBrush b_Brush1 = (SolidColorBrush) (btn_ButtonB.Back
ground);
 11: txt_Value1.Text = b_Brush1.Color.ToString();
 12:
 13: // 通过依赖性属性的 GetValue 获取 ButtonB 的背景颜色
 14: SolidColorBrush b_Brush2 = (SolidColorBrush) (btn_ButtonB.GetV
alue(
 15:     Button.BackgroundProperty));
 16: txt_Value2.Text = b_Brush2.Color.ToString();
```

如果使用的是现有属性，则上述操作通常不是必需的（使用包装会更方便，并能够更好地向开发人员工具公开属性）。但是在某些情况下适合直接调用 API。

2、使用由依赖项属性提供的属性功能

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

3、自定义依赖项属性及重写依赖项属性

对于自定义依赖项属性，其所在的类型必须直接或间接继承 System.Windows.DependencyObject 类，依赖项属性是通过调用 Register 方法（或 RegisterReadOnly，自定义的只读的依赖项属性）在 WPF 属性系统中注册，并通过 DependencyProperty 标识符字段备份的属性。依赖项属性只能由 DependencyObject 类型使用，但 DependencyObject 在 WPF 类层次结构中的级别很高，因此，WPF 中的大多数可用类都支持依赖项属性。在对依赖项属性及 CLR 包装属性命名时必须满足：CLR 包装属性名+Property＝依赖项属性名。

例如：在某 DependencyObject 类的子类中定义：

```
1: // 定义并注册依赖项属性
2: public static readonly DependencyProperty AquariumGraphicProperty =
3:     DependencyProperty.Register(
4:         "AquariumGraphic",                  // 要注册的依赖项对象的名称
5:         typeof(Uri),                        // 属性的类型
6:         typeof(AquariumObject),             // 正注册依赖项对象的所有者类型
7:         new FrameworkPropertyMetadata(  // 依赖项对象的属性元数据
8:             null,
9:             FrameworkPropertyMetadataOptions.AffectsRender,
10:            new PropertyChangedCallback(OnUriChanged)
11:        )
12:    );
13:
14: // 定义 CLR 包装属性
15: public Uri AquariumGraphic
16: {
17:     get { return (Uri)GetValue(AquariumGraphicProperty); }
```

```
 18:     set { SetValue(AquariumGraphicProperty, value); }
 19: }
```

二、路由事件（RoutedEvent）

先看以下的应用程序：

```
  1: <StackPanel>
  2:     <StackPanel Background="Blue" Margin="3" x:Name="panel_Blu
e">
  3:         <Button Margin="3" Content="Button A" x:Name="btn_B1" C
lick="btn_B_Click" />
  4:         <Button Margin="3" Content="Button B" x:Name="btn_B2" C
lick="btn_B_Click" />
  5:     </StackPanel>
  6:     <StackPanel Background="Green" Margin="3" x:Name="panel_Gr
een" ButtonBase.Click="panel_G_Click">
  7:         <Button Margin="3" Content="Button A" x:Name="btn_G1"
/>
  8:         <Button Margin="3" Content="Button B" x:Name="btn_G2"
/>
  9:     </StackPanel>
 10:     <StackPanel Background="Red" Margin="3" x:Name="panel_Red"
 ButtonBase.Click="panel_R_Click">
 11:         <Button Margin="3" Content="Button A" x:Name="btn_R1" C
lick="btn_R1_Click" />
 12:         <Button Margin="3" Content="Button B" x:Name="btn_R2" C
lick="btn_R2_Click" />
 13:     </StackPanel>
 14: </StackPanel>
```

```
  1: private void btn_B_Click(object sender, RoutedEventArgs e)
  2: {
  3:     MessageBox.Show(
  4:         "Click Blue Panel Button !",
  5:         "System Infomation",
  6:         MessageBoxButton.OK,
  7:         MessageBoxImage.Information);
  8:
  9:     MessageBox.Show(
 10:         string.Format("sender Type is {0}.", sender.GetType().N
ame));
 11:
 12:     Button sourceButton = (Button) (sender);
```

```
  13:     MessageBox.Show(
  14:         string.Format("Source Button is {0} !", sourceButton.Name),
  15:         "System Infomation",
  16:         MessageBoxButton.OK,
  17:         MessageBoxImage.Information);
  18: }
  19:
  20: private void panel_G_Click(object sender, RoutedEventArgs e)
  21: {
  22:     MessageBox.Show(
  23:         "Click Green Panel Button !",
  24:         "System Infomation",
  25:         MessageBoxButton.OK,
  26:         MessageBoxImage.Information);
  27:
  28:     MessageBox.Show(
  29:         string.Format("sender Type is {0}.", sender.GetType().Name));
  30:
  31:     MessageBox.Show(
  32:         string.Format("e.Source Type is {0}.", e.Source.GetType().Name));
  33:
  34:     Button sourceButton = (Button)(e.Source);
  35:     MessageBox.Show(
  36:         string.Format("Source Button is {0} !", sourceButton.Name),
  37:         "System Infomation",
  38:         MessageBoxButton.OK,
  39:         MessageBoxImage.Information);
  40: }
  41:
  42: private void panel_R_Click(object sender, RoutedEventArgs e)
  43: {
  44:     MessageBox.Show(
  45:         "Click Red Panel Button !",
  46:         "System Infomation",
  47:         MessageBoxButton.OK,
  48:         MessageBoxImage.Information);
  49: }
  50:
  51: private void btn_R1_Click(object sender, RoutedEventArgs e)
  52: {
```

```
53:     MessageBox.Show(
54:         "Click Red Panel Button A !",
55:         "System Infomation",
56:         MessageBoxButton.OK,
57:         MessageBoxImage.Information);
58: }
59:
60: private void btn_R2_Click(object sender, RoutedEventArgs e)
61: {
62:     MessageBox.Show(
63:         "Click Red Panel Button B !",
64:         "System Infomation",
65:         MessageBoxButton.OK,
66:         MessageBoxImage.Information);
67:
68:     e.Handled = true;
69: }
```

1、在 panel_Blue 中定义的三个按钮的 Click 事件属于“类事件”，即在类型对象中声明事件的绑定。此时事件响应方法中的 sender 指的就是由哪个对象引发的事件

2、在 panel_Green 中定义 ButtonBase.Click="xxx"，将其容器内所有的 ButtonBase 类型及其子类型的事件，统一绑定到一个事件处理方法上，统一处理。此时事件响应方法中的 sender 指的是 panel_Green 对象，而 e.Source 指的是引发 ButtonBase.Click 的某个按钮

3、路由事件的处理模型常用的有两种：

冒泡事件：由子控件位次向父容器传递，大部分的路由事件都是冒泡事件

隧道事件：由父容器位次向其子容器、控件传递，一般 PreXXX 事件属性隧道事件

直接

4、使用路由事件响应方法中的e.Handled = true;意味着此事件已经被处理，将不再传递，默认 e.Handled 的值为 false，意味着此路由事件还未处理完整，事件将依据其模型继续向下处理（即执行其他的事件处理方法）

键盘输入、鼠标输入、焦点处理

一、键盘类和键盘事件

WPF 提供了基础的键盘类（System.Input.Keyboard 类），该类提供与键盘相关的事件、方法和属性，这些事件、方法和属性提供有关键盘状态的信息。Keyboard 的事件也通过 UIElement 等 XAML 基元素类的事件向外提供。

对于键盘操作，其常用的事件有两组：

KeyDown 事件和 PreviewKeyDown 事件：处理键盘键按下 ；

KeyUp 事件和 PreviewKeyUp 事件：处理键盘键抬起 ；

其中 KeyDown 和 KeyUp 事件属于冒泡路由事件，而 PreviewKeyDown 和 PreviewKeyup 属于隧道路由事件。

为了使元素能够接收键盘输入，该元素必须可获得焦点。默认情况下，大多数 UIElement 派生对象都可获得焦点。如果不是这样，则要使元素可获得焦点，请将基元素上的 Focusable 属性设置为 true。像 StackPanel 和 Canvas 这样的 Panel 类将 Focusable 的默认值设置为 false。因此，对要获取键盘焦点的这些对象而言，必须将 Focusable 设置为 true。

例如：在笔者的 Notebook 中有“静音”、“增大音量”、“减小音量”这三个快捷键，在一个应用程序的窗体上处理这三个键的点击可以：

```
1: <Window x:Class="InputCommandAndFocus.Window1"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="Window1" Height="300" Width="480"
5:     Focusable="True" PreviewKeyDown="Window_PreviewKeyDown">
6:     <Canvas>
7:         <!-- ... -->
8:     </Canvas>
9: </Window>
```

```
1: private void Window_PreviewKeyDown(object sender, KeyEventArgs
e)
2: {
3:     if (e.Key == Key.VolumeMute)
4:     {
5:         // 按下“静音”键
6:         txtMessage.Text = "Mute";
7:         e.Handled = true;
8:     }
9:     else if (e.Key == Key.VolumeUp)
10:    {
11:        // 按下“增大音量”键
12:        txtMessage.Text = "Up";
13:        e.Handled = true;
14:    }
15:    else if (e.Key == Key.VolumeDown)
16:    {
17:        // 按下“减小音量”键
18:        txtMessage.Text = "Down";
19:        e.Handled = true;
20:    }
21: }
```

二、鼠标类和鼠标事件

WPF 提供的 System.Input.Mouse 类提供与鼠标相关的事件、方法和属性，这些事件、方法和属性提供有关鼠标状态的信息。与 Keyboard 类类似，其事件也通过 UIElement 等基元素向外提供。

其事件主要有以下几组(每个事件均包含 XXX 冒泡路由事件和 PreviewXXX 隧道路由事件)

MouseDown、MouseUp 事件：处理鼠标键的按下与抬起 ；

MouseEnter、MouseLeave、MouseMove：处理鼠标进入、离开控件及在控件上移动

MouseWheel：处理鼠标滚轮滚动 ；

另外，对于鼠标位置的捕获，使用 Mouse 类的 GetPosition 方法，其参数是一个 UIElement，表示其鼠标位置基于哪一个控件的坐标系。

例如，对于一个矩形图形，设置其鼠标的各种事件：

```
1: <Rectangle Canvas.Left="246" Canvas.Top="46" Height="118"
2:           Name="mainRectangle" Stroke="Black" Width="200"
Fill="White"
3:           MouseEnter="mainRectangle_MouseEnter"
MouseLeave="mainRectangle_MouseLeave"
4:           MouseMove="mainRectangle_MouseMove"
MouseDown="mainRectangle_MouseDown"
5:           MouseWheel="mainRectangle_MouseWheel"/>
```

```
1: private void mainRectangle_MouseEnter(object sender,
MouseEventArgs e)
2: {
3:     // 鼠标进入控件时，控件的颜色为红色
4:     mainRectangle.Fill = new SolidColorBrush(Colors.Red);
5: }
6:
7: private void mainRectangle_MouseLeave(object sender,
MouseEventArgs e)
8: {
9:     // 鼠标离开控件时，控件的颜色为红色
10:    mainRectangle.Fill = new SolidColorBrush(Colors.White);
11: }
```

```
12:
13: private void mainRectangle_MouseMove(object sender,
MouseEventArgs e)
14: {
15:     // 获取基于 Rectangle 的鼠标的坐标
16:     Point pointBaseRectangle =
Mouse.GetPosition(mainRectangle);
17:     txtMessage.Text = string.Format(
18:         "Mouse Position (Base the Rectangle) is ({0},{1})",
19:         pointBaseRectangle.X, pointBaseRectangle.Y);
20:
21:     txtMessage.Text += "\r\n";
22:
23:     // 获取基于窗体的鼠标的坐标
24:     Point pointBaseWindow = Mouse.GetPosition(this);
25:     txtMessage.Text += string.Format(
26:         "Mouse Position (Base the Window) is ({0},{1})",
27:         pointBaseWindow.X, pointBaseWindow.Y);
28: }
29:
30: private void mainRectangle_MouseDown(object sender,
MouseButtonEventArgs e)
31: {
32:     // 获取点出的鼠标的按钮
33:     MouseButton button = e.ChangedButton;
34:
35:     txtMessage.Text += "\r\n";
36:     txtMessage.Text += string.Format(
37:         " Mouse Button is {0}", button.ToString());
```

```
 38: }
 39:
 40: private void mainRectangle_MouseWheel(object sender,
MouseWheelEventArgs e)
 41: {
 42:     if (e.Delta > 0)
 43:     {
 44:         // 如果向上推动滚轮，图形的宽度增加
 45:         rectangle1.Width++;
 46:     }
 47:
 48:     if (e.Delta < 0)
 49:     {
 50:         // 如果向下推动滚轮，图形的宽度减小
 51:         rectangle1.Width--;
 52:     }
 53: }
```

三、焦点处理

在 WPF 中，有两个与焦点有关的主要概念：键盘焦点和逻辑焦点。 键盘焦点指接收键盘输入的元素，而逻辑焦点指焦点范围中具有焦点的元素。

1、键盘焦点：

键盘焦点指当前正在接收键盘输入的元素。 在整个桌面上，只能有一个具有键盘焦点的元素。 在 WPF 中，具有键盘焦点的元素会将 IsKeyboardFocused 设置为 true。 Keyboard 类的静态属性 FocusedElement 获取当前具有键盘焦点的元素。

为了使元素能够获取键盘焦点，基元素的 Focusable 和 IsVisible 属性必须设置为 true。 有些类（如 Panel 基类）默认情况下将 Focusable 设置为 false；因此，如果您希望此类元素能够获取键盘焦点，必须将 Focusable 设置为 true。

可以通过用户与 UI 交互（例如，按 Tab 键定位到某个元素或者在某些元素上单击鼠标）来获取键盘焦点。 还可以通过使用 Keyboard 类的 Focus 方法，以编程方式获取键盘焦点。 Focus 方法尝试将键盘焦点给予指定的元素。 返回的元素是具有键盘焦点的元素，如果有旧的或新的焦点对象阻止请求，则具有键盘焦点的元素可能不是所请求的元素。

2、逻辑焦点

逻辑焦点指焦点范围中的 FocusManager..:::.FocusedElement。 焦点范围是一个跟踪其范围内的 FocusedElement 的元素。 当键盘焦点离开焦点范围时，焦点元素会失去键盘焦点，但保留逻辑焦点。当键盘焦点返回到焦点范围时，焦点元素会再次获得键盘焦点。这使得键盘焦点可以在多个焦点范围之间切换，但确保了在焦点返回到焦点范围时，焦点范围中的焦点元素再次获得键盘焦点。

一个应用程序中可以有多个具有逻辑焦点的元素，但在一个特定的焦点范围中只能有一个具有逻辑焦点的元素。GetFocusScope 返回指定元素的焦点范围。WPF 中默认情况下即为焦点范围的类有 Window、MenuItem、ToolBar 和 ontextMenu。GetFocusedElement 获取指定焦点范围的焦点元素。SetFocusedElement 设置指定焦点范围中的焦点元素。SetFocusedElement 通常用于设置初始焦点元素。

3、键盘导航

当按下导航键之一时，KeyboardNavigation 类将负责实现默认键盘焦点导航。 导航键有：Tab、Shift+Tab、Ctrl+Tab、Ctrl+Shift+Tab、向上键、向下键、向左键和向右键。可以通过设置附加的 KeyboardNavigation 属性 TabNavigation、ControlTabNavigation 和 DirectionalNavigation 来更改导航容器的导航行为。这些属性是 KeyboardNavigationMode 类型，可能值有 Continue、Local、Contained、Cycle、Once 以及 None。 默认值是 Continue，这意味着元素不是导航容器。

4、焦点事件

与键盘焦点相关的事件有 PreviewGotKeyboardFocus、GotKeyboardFocus、PreviewLostKeyboardFocus 以及 LostKeyboardFocus。 这些事件定义为 Keyboard 类的附加事件，但更便于作为基元素类上的等效路由事件来访问。当元素获取键盘焦点时，会引发 GotKeyboardFocus。当元素失去键盘焦点时，会引发 LostKeyboardFocus。 如

果处理了 PreviewGotKeyboardFocus 事件或 PreviewLostKeyboardFocusEvent 事件，并且 Handled 设置为 true，则焦点将不会改变。

WPF 命令

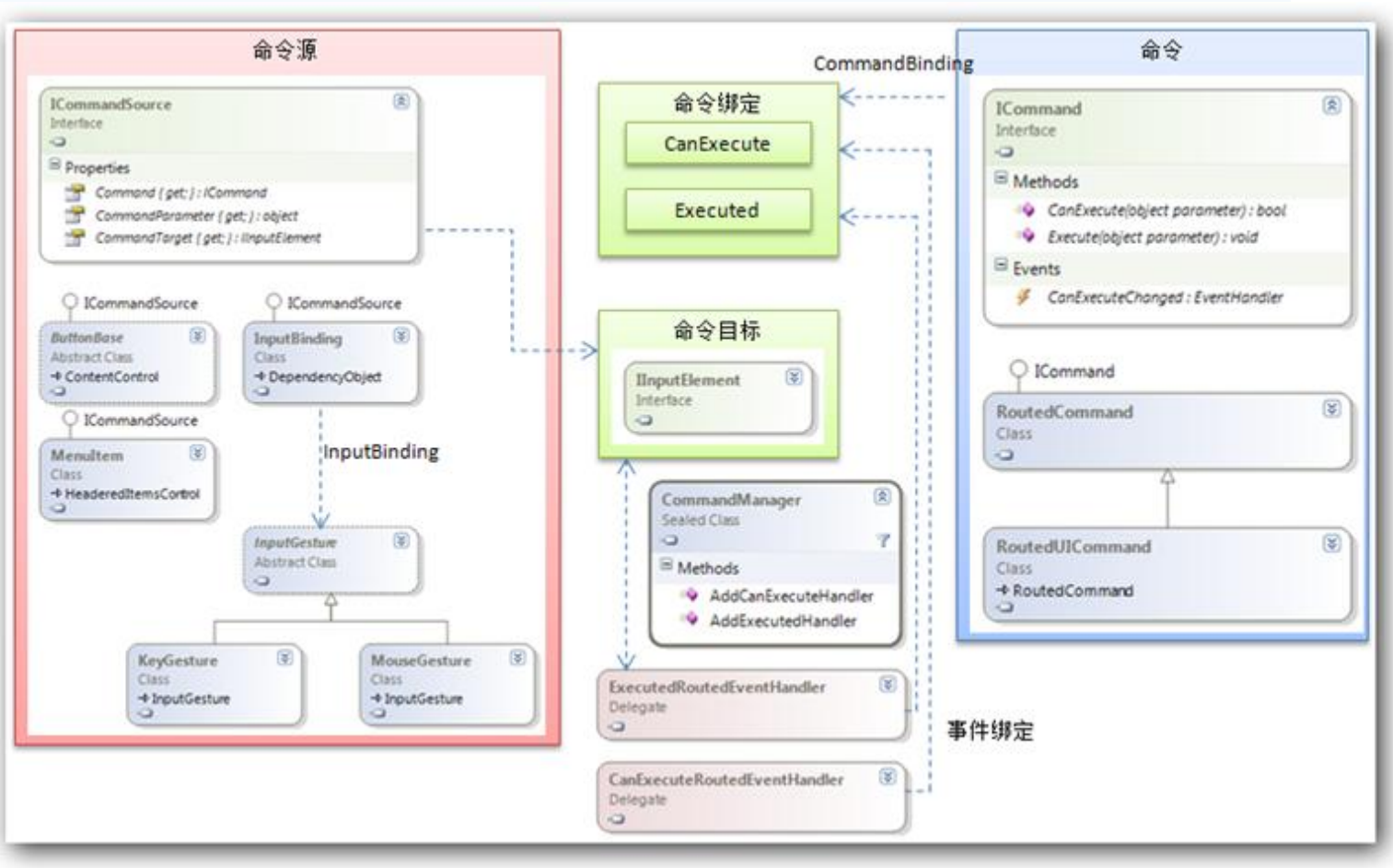
WPF 中的命令路由与事件路由是两个很让初学者头痛的概念，对于命令路由可以理解为，系统（WPF）定义了一系列的操作，在应用程序中可以直接使用。例如，定义一系列菜单，执行对窗体中文本框的复制、剪切、粘贴操作，简单地可以这样做：

```
1: <Grid>
2:     <Grid.RowDefinitions>
3:         <RowDefinition Height="23" />
4:         <RowDefinition />
5:     </Grid.RowDefinitions>
6:     <Menu Grid.Row="0" Grid.Column="0">
7:         <MenuItem Header="Edit">
8:             <MenuItem x:Name="menuCopy" Header="Copy"
9:                       Command="ApplicationCommands.Copy" />
10:            <MenuItem x:Name="menuCut" Header="Cut"
11:                      Command="ApplicationCommands.Cut" />
12:            <MenuItem x:Name="menuPaste" Header="Paste"
13:                      Command="ApplicationCommands.Paste" />
14:        </MenuItem>
15:    </Menu>
16:    <TextBox Grid.Row="1" Grid.Column="0" x:Name="mainText"
17:             TextWrapping="Wrap" AcceptsReturn="True" />
18: </Grid>
```

WPF 中的路由命令模型可以分为四个主要概念：命令、命令源、命令目标以及命令绑定： 命令是要执行的操作。在本例中命令为 ApplicationCommands.Copy、Cut、Paste

命令源是调用命令的对象。 在本例中命令源为三个 MenuItem 控件命令目标是在其上执行命令的对象。 在本例中命令目标是 mainText 这个 TextBox 文本框

命令绑定是将命令逻辑映射到命令的对象。 在本例中命令绑定到系统定义的对于文本框的“复制”、“剪切”、“粘贴”操作、其四者的关系如下图所示：



一、命令：

WPF 中的命令是通过实现 ICommand 接口来创建的。ICommand 公开两个方法（Execute 和 CanExecute）和一个事件 (CanExecuteChanged)。Execute 执行与命令关联的操作。CanExecute 确定是否可以在当前命令目标上执行命令。如果集中管理命令操作的命令管理器检测到命令源中发生了更改，此更改可能使得已引发但尚未由命令绑定执行的命令无效，则将引发 CanExecuteChanged。ICommand 的 WPF 实现是 RoutedCommand 类。RoutedCommand 上的 Execute 方法在命令目标上引发 PreviewExecuted 和 Executed 事件。RoutedCommand 上的 CanExecute 方法在命令目标上引发 CanExecute 和 PreviewCanExecute 事件。这些事件沿元素树以隧道和冒泡形式传递，直到遇到具有该特定命令的 CommandBinding 的对象。 WPF 提供了一组常用的路由命令，这组命令分布在几个类中：MediaCommands、ApplicationCommands、NavigationCommands、ComponentCommands 和

EditingCommands。这些类仅包含 RoutedCommand 对象，而不包含命令的实现逻辑。实现逻辑由在其上执行命令的对象负责。

WPF 已封装的命令类有：

命令类	示例命令
ApplicationCommands	Close、Cut、Copy、Paste、Save、Print
NavigationCommands	BrowseForward、BrowseBack、Zoom、Search
EditingCommands	AlignXXX、MoveXXX、SelectXXX
MediaCommands	Play、Pause、NextTrack、IncreaseVolume、Record、Stop
ComponentCommands	MoveXXX、SelectXXX、ScrollXXX、ExtendSelectionXXX

XXX 代表操作的集合，例如 MoveNext 和 MovePrevious。其中 ApplicationCommands 为默认的命令类，引用其中的命令时可以省略 ApplicationCommands。

二、命令源

命令源是调用命令的对象。例如，MenuItem、Button 和 KeyGesture 就是命令源。

WPF 中的命令源通常实现 ICommandSource 接口。

ICommandSource 公开三个属性：Command、CommandTarget 和 CommandParameter：

Command 是在调用命令源时执行的命令。

CommandTarget 是要在其上执行命令的对象。值得注意的是，在 WPF 中，ICommandSource 上的 CommandTarget 属性只有在 ICommand 是 RoutedCommand 时才适用。如果在 ICommandSource 上设置了 CommandTarget，而对应的命令不是 RoutedCommand，将会忽略命令目标。如果未设置 CommandTarget，则具有键盘焦点的元素将是命令目标。

CommandParameter 是用户定义的数据类型，用于将信息传递到实现命令的处理程序。

实现 ICommandSource 的 WPF 类包括：ButtonBase、MenuItem、Hyperlink 以及 InputBinding。ButtonBase、MenuItem 和 Hyperlink 在被单击时调用命令，InputBinding 在与之关联的 InputGesture 执行时调用命令。 ButtonBase 等直接使用控件的 Command 属性绑定命令：

```
1: <Button Command="ApplicationCommands.Copy" />
```

而 InputBinding 使用 KeyBinding 或 MouseBinding 绑定特定的输入手势到某一命令上：例如在 Window 上注册 Ctrl+F2 快捷键到 ApplicationCommands.Open 上：

```
1: <KeyBinding Command="ApplicationCommands.Open"
2:             Key="F2" Modifiers="Control" />
```

三、命令目标

命令目标是在其上执行命令的元素。对于 RoutedCommand 而言，命令目标是 Executed 和 CanExecute 的路由的起始元素。前面已提到，在 WPF 中，ICommandSource 上的 CommandTarget 属性只有在 ICommand 是一个 RoutedCommand 时才适用。如果在 ICommandSource 上设置了 CommandTarget，而对应的命令不是 RoutedCommand，将会忽略命令目标。命令源可以显式设置命令目标。如果未定义命令目标，则具有键盘焦点的元素将用作命令目标。将具有键盘焦点的元素用作命令目标的一个好处是，应用程序开发人员可以使用同一个命令源在多个目标上调用命令，而不必跟踪命令目标。例如，如果 MenuItem 在具有一个 TextBox 控件和一个 PasswordBox 控件的应用程序中调用“粘贴”命令，则目标既可以是 TextBox，也可以是 PasswordBox，具体取决于哪个控件具有键盘焦点。

如将 Paste 命令设置到 mainText 控件上：

```
1: <MenuItem x:Name="menuPaste" Header="Paste"
2:           Command="ApplicationCommands.Paste"
3:           CommandTarget="{Binding ElementName=txtMain}"/>
```

四、命令绑定

CommandBinding 将一个命令与实现该命令的事件处理程序关联。CommandBinding 类包含一个 Command 属性以及 PreviewExecuted、Executed、PreviewCanExecute 和 CanExecute 事件。

Command 是 CommandBinding 要与之关联的命令。附加到 PreviewExecuted 和 Executed 事件的事件处理程序实现命令逻辑。附加到 PreviewCanExecute 和 CanExecute 事件的事件处理程序确定命令是否可以在当前命令目标上执行。

CanExecute 事件和 PreviewCanExecute 事件，通过其 EventArgs 参数中的 CanExecute 属性，设置其命令是否可以执行，并且系统会自动的和命令目标的某些特定属性进行绑定，例如 Button、MenuItem 等在 CanExecute 属性的值设为 False 时，会“灰化”，不可用

Executed 事件和 PreviewExecuted 事件的代码，是执行命令的真正代码。 例如，将 ApplicationCommans.Save 绑定到菜单栏的 Save 菜单项中，并在文本框中没有任何文本时不可用：

```
1: <Window x:Class="InputCommandAndFocus.WinCommandDemo"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="WinCommandDemo" Height="300" Width="300"
Focusable="True">
5:     <Window.InputBindings>
6:         <KeyBinding Command="ApplicationCommands.Save"
7:                     Key="F3" Modifiers="Control" />
8:     </Window.InputBindings>
9:     <Window.CommandBindings>
10:         <CommandBinding Command="ApplicationCommands.Save"
```

```
 11:
CanExecute="CommandBinding_Save_CanExecute"
 12:                     Executed="CommandBinding_Save_Executed" />
 13:     </Window.CommandBindings>
 14:     <Grid>
 15:         <Grid.RowDefinitions>
 16:             <RowDefinition Height="23" />
 17:             <RowDefinition />
 18:         </Grid.RowDefinitions>
 19:         <Menu Grid.Row="0" Grid.Column="0">
 20:             <MenuItem Header="File">
 21:                 <MenuItem x:Name="menuSave" Header="Save"
 22:                     Command="ApplicationCommands.Save" />
 23:             </MenuItem>
 24:         </Menu>
 25:         <TextBox Grid.Row="1" Grid.Column="0" x:Name="mainText"
 26:             TextWrapping="Wrap" AcceptsReturn="True" />
 27:     </Grid>
 28: </Window>
```

```
  1: private void CommandBinding_Save_CanExecute(object sender,
CanExecuteRoutedEventArgs e)
  2: {
  3:     if (mainText.Text == string.Empty)
  4:     {
  5:         // 如果文本框中没有任何文本，则不可以保存
  6:         e.CanExecute = false;
  7:     }
  8:     else
  9:     {
```

```
10:         e.CanExecute = true;
11:     }
12: }
13:
14: private void CommandBinding_Save_Executed(object sender,
ExecutedRoutedEventArgs e)
15: {
16:     // 保存文件对话框
17:     SaveFileDialog save = new SaveFileDialog();
18:     save.Filter = "文本文件|*.txt|所有文件|*.*";
19:
20:     bool? result = save.ShowDialog();
21:     if (result.Value)
22:     {
23:         // 执行保存文件操作
24:     }
25: }
```

WPF 资源

一、什么是资源

通常使用 WPF 资源作为重用通常定义的对象和值的简单方法。例如定义一种可以复用的单色的 Brush 对象，按钮的背景及矩形的填充颜色均使用此 Brush：

```
1: <Window x:Class="WPFResource.WinBasicResource"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="Basic Resource" Height="200" Width="300">
5:     <Window.Resources>
6:         <SolidColorBrush x:Key="myBrush" Color="Gold" />
7:     </Window.Resources>
8:     <StackPanel>
9:         <Button Margin="5" Content="Sample Button"
Background="{StaticResource myBrush}" />
10:         <Rectangle Margin="5" Width="100" Height="100"
Fill="{StaticResource myBrush}" />
11:     </StackPanel>
12: </Window>
```

在 WPF 中资源通常用作“样式”（Style）、样式模板、数据模板等。

二、资源的定义及 XAML 中引用

资源可以定义在以下几个位置：

1 应用程序级资源

定义在 App.xaml 文件中，作为整个应用程序共享的资源存在 ;在 App.xaml 文件中定义：

```
1: <Application x:Class="WPFResource.App"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     StartupUri="Window1.xaml">
5:     <Application.Resources>
6:         <SolidColorBrush Color="Gold" x:Key="myGoldBrush" />
7:     </Application.Resources>
8: </Application>
```

在 ApplicationResourceDemo.xaml 文件（窗体）中使用 App.xaml 中定义的 Resource

```
1: <Window x:Class="WPFResource.ApplicationResourceDemo"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="Application Resource Demo" Height="300" Width="300">
5:     <StackPanel>
6:         <Button Margin="5" Background="{StaticResource
myGoldBrush}">Sample Button</Button>
7:     </StackPanel>
8: </Window>
```

2 窗体级资源

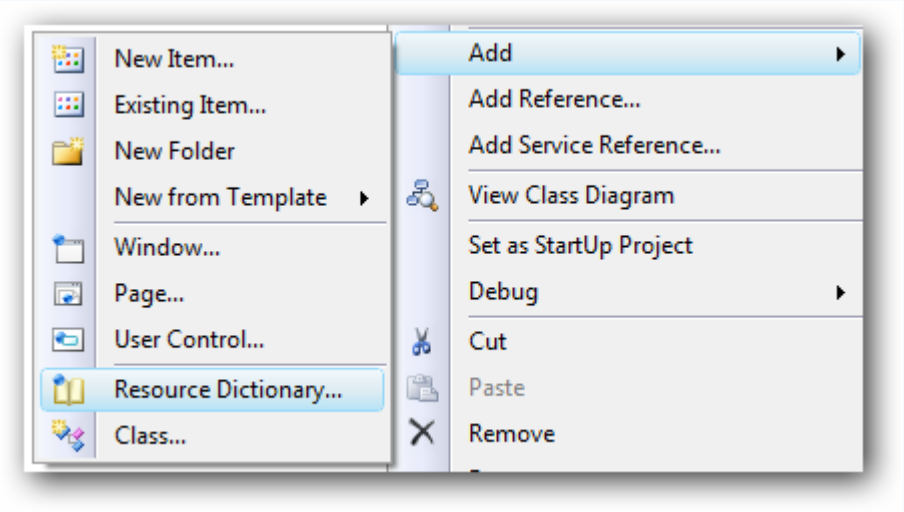
定义在 Window 或 Page 中，作为一个窗体或页面共享的资源存在

```
1: <Window x:Class="WPFResource.WindowResourceDemo"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="WindowResourceDemo" Height="300" Width="300">
5:     <Window.Resources>
```

```
    <SolidColorBrush x:Key="myRedBrush" Color="Red" />
  </Window.Resources>
  <StackPanel>
    <Button Margin="5" Background="{StaticResource
myRedBrush}">Sample Button</Button>
  </StackPanel>
</Window>
```

3 文件级资源

定义在资源字典的 XAML 文件中，再引用 在 Visual Studio 的 WPF 应用程序项目中，添加"资源字典（Resource Dictionary）"类型的项



在其 XAML 文件中定义：

```
<ResourceDictionary
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <SolidColorBrush x:Key="myWhiteBrush" Color="White" />
</ResourceDictionary>
```

在 FileResourceDemo.xaml 文件（窗体）中，将其注册为窗体级的资源，并引用

```
<Window x:Class="WPFResource.FileResourceDemo"
```

```
   2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:     Title="FileResourceDemo" Height="300" Width="300">
   5:     <Window.Resources>
   6:         <ResourceDictionary Source="MyResourceDictionary.xaml"
/>
   7:     </Window.Resources>
   8:     <StackPanel>
   9:         <Button Margin="5" Background="{StaticResource
myWhiteBrush}">Sample Button</Button>
  10:     </StackPanel>
  11: </Window>
```

4 对象（控件）级资源

定义在某个 ContentControl 中，作为其子容器、子控件共享的资源 在 Button 中定义一个资源，供 Button 内的 Content 控件使用

```
   1: <Window x:Class="WPFResource.ControlResourceDemo"
   2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   4:     Title="ControlResourceDemo" Height="300" Width="300">
   5:     <StackPanel>
   6:         <Button Margin="5">
   7:             <Button.Resources>
   8:                 <SolidColorBrush x:Key="myGreenBrush"
Color="Green" />
   9:             </Button.Resources>
  10:             <Button.Content>
```

```
11:                 <TextBlock Text="Sample Text"
Background="{StaticResource myGreenBrush}" />
12:             </Button.Content>
13:         </Button>
14:     </StackPanel>
15: </Window>
```

三、XAML 解析资源的顺序

在 XAML 中解析资源按照由引用资源的控件向外层容器依次调用资源。例如在在应用程序级别、窗体级别及对象级别分为定义 x:Key 相的同资源：

在 App.xaml 文件中：

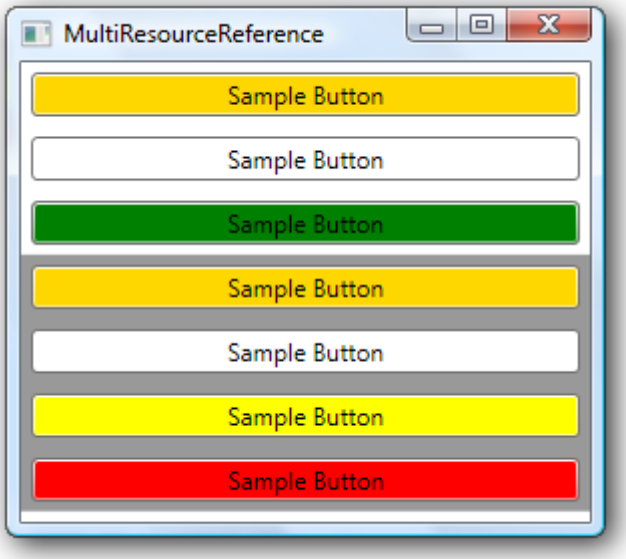
```
1: <Application x:Class="WPFResource.App"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     StartupUri="MultiResourceReference.xaml">
5:     <Application.Resources>
6:         <!-- 应用程序级资源 -->
7:         <SolidColorBrush Color="Gold" x:Key="myGoldBrush" />
8:         <SolidColorBrush Color="Blue" x:Key="myBrush" />
9:     </Application.Resources>
10: </Application>
```

在窗体的 XAML 文件中：

```
1: <Window x:Class="WPFResource.MultiResourceReference"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="MultiResourceReference" Height="265" Width="300">
5:     <Window.Resources>
```

```
 6:         <!-- 窗体级资源 -->
 7:         <SolidColorBrush Color="White" x:Key="myWhiteBrush" />
 8:         <SolidColorBrush Color="Green" x:Key="myBrush" />
 9:     </Window.Resources>
10:     <StackPanel>
11:         <!-- 使用应用程序级定义的资源 -->
12:         <Button Margin="5" Content="Sample Button"
Background="{StaticResource myGoldBrush}" />
13:
14:         <!-- 使用窗体级定义的资源 -->
15:         <Button Margin="5" Content="Sample Button"
Background="{StaticResource myWhiteBrush}" />
16:
17:         <!-- 窗体级资源的值覆盖应用程序级资源的值 -->
18:         <Button Margin="5" Content="Sample Button"
Background="{StaticResource myBrush}" />
19:
20:         <StackPanel Background="#FF999999">
21:             <StackPanel.Resources>
22:                 <!-- 对象级资源 -->
23:                 <SolidColorBrush Color="Yellow"
x:Key="myYellowBrush" />
24:                 <SolidColorBrush Color="Red" x:Key="myBrush" />
25:             </StackPanel.Resources>
26:
27:             <!-- 使用应用程序级定义的资源 -->
28:             <Button Margin="5" Content="Sample Button"
Background="{StaticResource myGoldBrush}" />
```

```
 29:
 30:             <!-- 使用窗体级定义的资源 -->
 31:             <Button Margin="5" Content="Sample Button"
Background="{StaticResource myWhiteBrush}" />
 32:
 33:             <!-- 使用对象级定义的资源 -->
 34:             <Button Margin="5" Content="Sample Button"
Background="{StaticResource myYellowBrush}" />
 35:
 36:             <!-- 使用对象级定义的资源覆盖窗体级、应用程序级定义的资源 -->
 37:             <Button Margin="5" Content="Sample Button"
Background="{StaticResource myBrush}" />
 38:         </StackPanel>
 39:     </StackPanel>
 40: </Window>
```



四、静态资源（StaticResource）和动态资源（DynamicResource）

资源可以作为静态资源或动态资源进行引用。这是通过使用 StaticResource 标记扩展或 DynamicResource 标记扩展完成的。

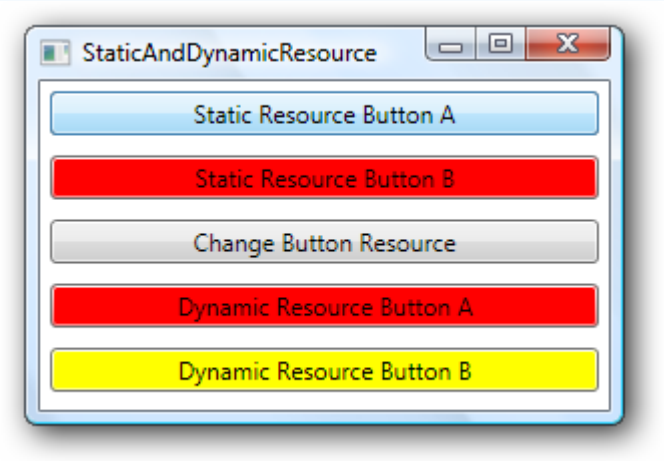
通常来说，不需要在运行时更改的资源使用静态资源；而需要在运行时更改的资源使用动态资源。动态资源需要使用的系统开销大于静态资源的系统开销。例如以下的例子：

```
1: <Window x:Class="WPFResource.StaticAndDynamicResource"
2:
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3:     xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4:     Title="StaticAndDynamicResource" Height="200" Width="300">
5:     <Window.Resources>
6:         <SolidColorBrush x:Key="ButtonBrush" Color="Red" />
7:     </Window.Resources>
8:
9:     <StackPanel>
10:         <Button Margin="5" Content="Static Resource Button A"
Background="{StaticResource ButtonBrush}" />
11:         <Button Margin="5" Content="Static Resource Button B"
Background="{StaticResource ButtonBrush}">
12:             <Button.Resources>
13:                 <SolidColorBrush x:Key="ButtonBrush"
Color="Yellow" />
14:             </Button.Resources>
15:         </Button>
16:         <Button Margin="5" Content="Change Button Resource"
Click="Button_Click" />
17:         <Button Margin="5" Content="Dynamic Resource Button A"
Background="{DynamicResource ButtonBrush}" />
```

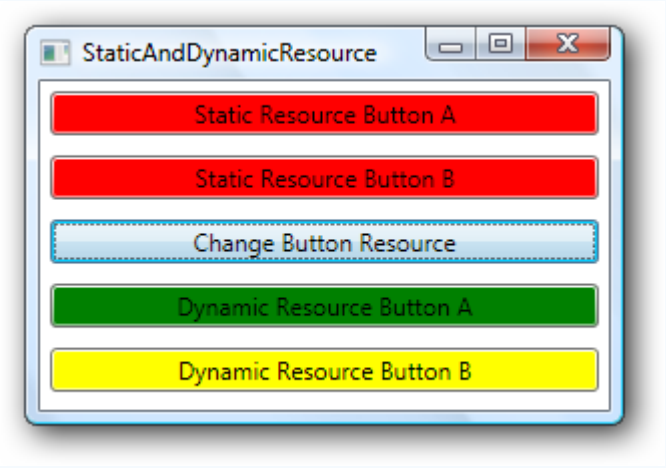
```
 18:         <Button Margin="5" Content="Dynamic Resource Button B"
Background="{DynamicResource ButtonBrush}">
 19:             <Button.Resources>
 20:                 <SolidColorBrush x:Key="ButtonBrush"
Color="Yellow" />
 21:             </Button.Resources>
 22:         </Button>
 23:     </StackPanel>
 24: </Window>
```

```
  1: private void Button_Click(object sender, RoutedEventArgs e)
  2: {
  3:     SolidColorBrush brush = new SolidColorBrush(Colors.Green);
  4:     this.Resources["ButtonBrush"] = brush;
  5: }
```

以上的例子在运行时显示如下：



而点击“Change Button Resource”按钮后，显示的结果为：



从程序执行的结果来看，我们可以得到如下的结论：

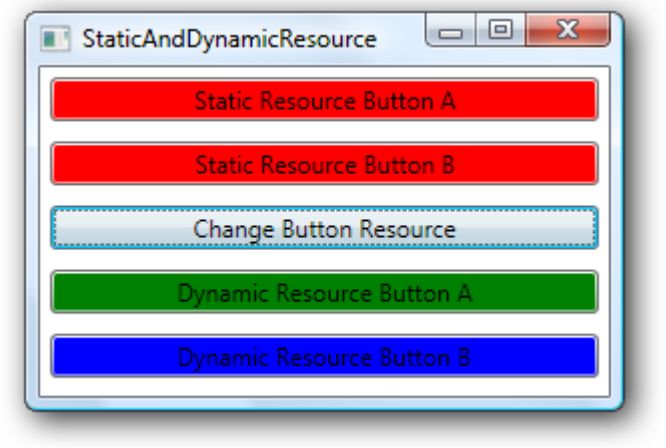
静态资源引用是从控件所在的容器开始依次向上查找的，而动态资源的引用是从控件开始向上查找的（即控件的资源覆盖其父容器的同名资源）

更改资源时，动态引用的控件样式发生变化（即"Dynamic Resource Button A"发生变化）

如果要更改"Dynamic Resource Button B"的背景，需要在按钮的事件中添加以下代码（将"Dynamic Resource Button B"的控件的 x:Name 设置为"btn4"）

```
1: SolidColorBrush brushB = new SolidColorBrush(Colors.Blue);
2: this.btn4.Resources["ButtonBrush"] = brushB;
```

执行的结果如下：



静态资源引用最适合于以下情况：

您的应用程序设计几乎将所有的应用程序资源集中到页或应用程序级别的资源字典中。静态资源引用不会基于运行时行为（例如重新加载页）进行重新求值，因此，根据您的资源和应用程序设计避免大量不必要的动态资源引用，这样可以提高性能。

您正在设置不在 DependencyObject 或 Freezable 上的属性的值。

您正在创建将编译为 DLL 并打包为应用程序的一部分或在应用程序之间共享的资源字典。

您正在为自定义控件创建一个主题，并定义在主题中使用的资源。对于这种情况，通常不需要动态资源引用查找行为，而需要静态资源引用行为，以使该查找可预测并且独立于该主题。使用动态资源引用时，即使是主题中的引用也会直到运行时才进行求值，并且在应用主题时，某个本地元素有可能会重新定义您的主题试图引用的键，并且本地元素在查找中会位于主题本身之前。如果发生该情况，主题将不会按预期方式运行。

您正在使用资源来设置大量依赖项属性。依赖项属性具有由属性系统启用的有效值缓存功能，因此，如果您为可以在加载时求值的依赖项属性提供值，该依赖项属性将不必查看重新求值的表达式，并且可以返回最后一个有效值。该方法具有性能优势。

您需要为所有使用者更改基础资源，或者需要通过使用 x:Shared 属性为每个使用者维护独立的可写实例。

动态资源最适合于以下情况：

资源的值取决于直到运行时才知道的情况。这包括系统资源，或用户可设置的资源。例如，您可以创建引用由 SystemColors、SystemFonts 或 SystemParameters 公开的系统属性的 setter 值。这些值是真正动态的，因为它们最终来自于用户和操作系统的运行时环境。您还可以使用可以更改的应用程序级别的主题，在此情况下，页级别的资源访问还必须捕获更改。您正在为自定义控件创建或引用主题样式。您希望在应用程序生存期调整 ResourceDictionary 的内容。您有一个存在依存关系的复杂资源结构，在这种情况下，可能需要前向引用。静态资源引用不支持前向引用，但动态资源引用支持，因为资源直到运行时才需要进行求值，因此，前向引用不是一个相关概念。

从编译或工作集角度来说，您引用的资源特别大，并且加载页时可能无法立即使用该资源。静态资源引用始终在加载页时从 XAML 加载；而动态资源引用直到实际使用时才会加

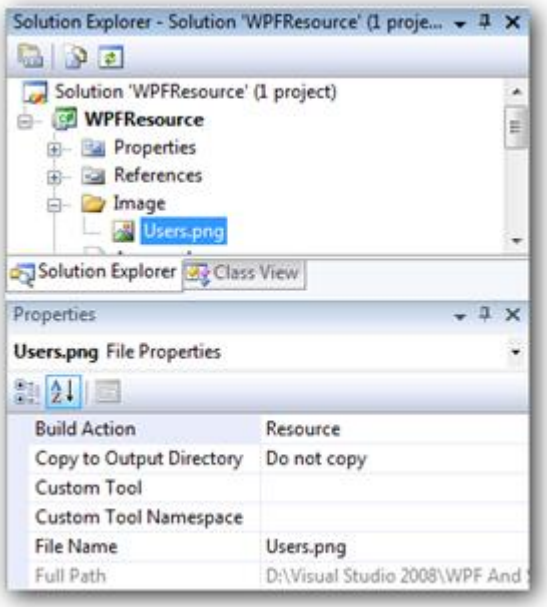
载。您要创建的样式的 setter 值可能来自受主题或其他用户设置影响的其他值。您正在将资源应用到元素，而在应用程序生存期中可能会在逻辑树中重新设置该元素的父级。更改此父级还可能会更改资源查找范围，因此，如果您希望基于新范围对重新设置了父级的元素的资源进行重新求值，请始终使用动态资源引用。

五、不同类型的资源

1、程序集资源。

这种常见于将图片设定到程序集中，做为程序集的资源。

程序集资源在定义时，将文件复制到解决方案-项目所在的目录或其子目录中，并将文件的属性中的 Build Action 设置为 Resource。（注意，WPF 不支持项目属性中的资源）



然后在 XAML 文件中使用如 Image 的 Source 属性，指定到此文件：

```
1: <StackPanel Background="#FFC7DAFF">
2:     <TextBlock Margin="5" Text="Assembly Resource" />
3:     <Image Margin="5" Source="Image/Users.png" Width="32"
Height="32" />
4:     <Button Margin="5">
5:         <Button.Content>
6:             <StackPanel Orientation="Horizontal">
```

```
   7:                 <Image Margin="5" Source="Image/Users.png"
Width="32" Height="32" />
   8:                 <TextBlock Text="My Button"
VerticalAlignment="Center" />
   9:             </StackPanel>
  10:         </Button.Content>
  11:     </Button>
  12: </StackPanel>
```

此项目编译后，在 Assembly 中将封装该图片文件。此种方法适用于较小的资源。

2、对象资源

除刚刚我们使用的图片做为程序集资源外，前面例子中所使用的资源均是对象资源。系统对于对象资源使用 ResouceDictionary 这个字典集合处理，其 Key 对应即 x:Key 声明的键，Value 对应资源。我们前面使用的都是 SolidColorBrush 对象，再例如使用字符串及 ImageBrush 对象做为资源：

```
   1: <StackPanel Background="#FFE6FDC8">
   2:     <StackPanel.Resources>
   3:         <c:String x:Key="strMessage">My String Value.</c:String>
   4:         <ImageBrush x:Key="imgBrush"
ImageSource="Image/Users.png"
   5:                     ViewportUnits="Absolute" Viewport="0 0 32 32"/>
   6:     </StackPanel.Resources>
   7:     <TextBlock Margin="5" Text="Object Resource" />
   8:     <TextBox Margin="5" Text="{StaticResource strMessage}" />
   9:     <Button Margin="5" Height="32"
HorizontalContentAlignment="Right"
  10:             Content="{StaticResource strMessage}"
  11:             Background="{StaticResource imgBrush}" Width="125" />
  12: </StackPanel>
```

程序执行结果为：