WPF插件

程序的功能是动态加载TreeView中的节点，并且当点击TreeView中的节点时，窗体右边的标题栏和下面的网页要动态的刷新改变，把右边的标题栏和下面显示网页的部分整体做一个自定义控件，对于每一个自定义控件独立开发，也就是做成插件，做好一个控件后直接把dll文件放过去就可以了。

下面对于几个关键点详述一下: 首先要有一个统一的接口，供主程序和插件调用，新建一个Class Library项目(Interface)，里面只放了一个接口IEditor.cs

```
 public interface IEditor
 {
        String PluginName
        {
            get;
        }
        UserControl GetControl();
  }
```

建一个插件项目WPF Usercontrol Library(PluginDLL)，类比可以多建几个插件项目

```
public class Plugin:IEditor
{
        public string PluginName
        {
            get
            {
                return "设备和接口";
            }
        }

        public UserControl GetControl()
        {
            return new WebControl1();
        }
}
```

轮到主程序调用了

把插件项目的编译路径设为一个指定的文件夹，然后主程序去这里拿dll.读到dll之后就加载和实例化

```
private void LoadPlugins()
{
           String[] dlls = Directory.GetFiles(AppDomain.CurrentDomain.BaseDirectory+"Plugins",
"*.dll");
            foreach (string dllPath in dlls)
            {
                Assembly assembly = Assembly.LoadFile(dllPath);//加载dll
                Type[] types = assembly.GetExportedTypes();//获得dll的assembly暴露的所有公开类型,
之后从公开类型里找到实现我们接口的类型
                Type typeIEditor = typeof(IEditor);

                for (int i = 0; i < types.Length; i++)
                {
                    if (typeIEditor.IsAssignableFrom(types[i]) && !types[i].IsAbstract)
                    {
                        IEditor editor = (IEditor)Activator.CreateInstance(types[i]);
                        TreeViewItem subitem = new TreeViewItem();
                        subitem.Header = editor.PluginName;
                        MySystem.Items.Add(subitem);
                        subitem.Selected += new RoutedEventHandler(treeItem_Click);
                        subitem.Tag = editor;
                    }
                }
            }
}
```

给节点添加相应的事件，把拿到的插件加入到主程序的页面

```
private void treeItem_Click(object sender, RoutedEventArgs e)
 {
            stackPanel1.Children.Clear();
            TreeViewItem item = sender as TreeViewItem;
            if (item != null)
            {
                if (item.Tag != null)
                {
                    IEditor editor = item.Tag as IEditor;
                    if (editor != null)
                    {
                        // 运行该插件
                        UserControl wnd = editor.GetControl();
                        WrapPanel panel = new WrapPanel();
                        panel.Children.Add(wnd);
                        panel.Width = wnd.Width;
                        panel.Height = wnd.Height;
                        stackPanel1.Height += panel.Height;
                        stackPanel1.Width = (stackPanel1.Width > panel.Width ? stackPanel1.Width
: panel.Width);
                        stackPanel1.Children.Add(panel);
                    }
                }
            }
}
```

需要添加的引用

```
PresentationCore
PresentationFramework
System.Windows
System.Core
System.Data
System.Web
System.Xaml
```

遇到的问题

每次修改插件时，修改完之后需要重新生成一下，然后删除已经加载好插件项目的编译路径，把新生成的插件项目的编译路径重新复制到指定的文件夹，然后运行程序，界面的内容才会改变。

错误

未能在命名空间“System.Windows.Markup”中找到类型名称“IComponentConnector”。

在项目里加 System.Xaml的reference就好了。

WPF中的TreeView控件

首先从工具箱中拖出一个TreeView控件到WPF窗体中，在Items属性可以添加节点。每一个节点就是一个TreeViewItem。如果要实现在一个子节点下继续添加节点，此时每一个TreeViewItem都存在一个Items属性，用于实现一级节点、二级节点等等。

```
<TreeView Height="665"HorizontalAlignment="Left" Margin="2,2,0,0"Name="tvFunctionList"
 VerticalAlignment="Top" Width="174">
   <TreeViewItem Header="系统设置" >
         <TreeViewItem Header="通信配置" Selected="Communication_Config" />
         <TreeViewItem Header="接收器配置" Selected="ReceiverConfig_Click" />
   </TreeViewItem>
</TreeView>
```

如果要为每一个节点添加选中事件，在Selected中绑定事件处理方法即可。此时选中对应节点，那么对应的事件处理程序将会执行了!

WPF的Name属性

WPF使用XAML来对界面进行编写，界面与后台逻辑分离。我们也可以写Style、Trigger来实现一些界面效果， 这些都是通过Name来定位控件的，例如Setter.TargetName、Trigger.SourceName和Binding的ElementName等。而这些Name都是通过设置控件的x:Name来定义的，如

但是，XAML中有x:Name和Name这两个属性，究竟它们有什么区别呢？

XAML与Code-Behind

在编写WPF程序时，通常需要分别编写前台XAML代码和后台Code-Behind代码。WPF通过一个partial关键字，将一个类的定义分为两部分：XAML和Code-Behind。其中XAML交给设计师设计，Code-Behind交给程序员写业务逻辑，从而实现分离。

在XAML上写标签，其实与后台写代码是等效的。可以只使用XAML或者只使用Code-Behind来写程序。

示例：

```
<!-- 只使用xaml编写一个窗体  -->
<!-- 只使用一个单独的xaml文件 -->
<Windowx:Class="Cnblog.OnlyXaml"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="OnlyXaml"
       Width="300"
       Height="300">
   <Grid>
       <Button Width="100" Height="100" Click="ButtonClick">Button</Button>
   </Grid>
   x:Code
   private void ButtonClick(object sender, RoutedEventArgs e)
   {
        MessageBox.Show(Button Click);
```

```
    }
    /x:Code
</Window>
```

```
namespace Cnblog
{
    // 只使用Code-Behind编写一个窗体
    // 只使用一个单独的OnlyCode.cs文件
    public class OnlyCode :Window
    {
        public OnlyCode()
        {
            // button
            var button = new Button { Content = "Button",Width = 100,Height = 100};
            button.Click += ButtonClick;
            // grid
            var grid = new Grid();
            grid.Children.Add(button);
            this.Width = 300;
            this.Height = 300;
            this.Title = "OnlyCode";
            this.Content = grid;
        }

        void ButtonClick(object sender, RoutedEventArgs e)
        {
            MessageBox.Show("Button Click");
        }
     }
}
```

上面例子，分别只使用XAML和Code-Behind来定义一个窗体，但是最终的效果都是一样的。

结论：虽然编码方式不一样，但效果是一样的，编译器对XAML进行编译生成BAMP，根据标签创建相应对象。

XAML中x:Name和Name最终效果相同

如果你在xaml中创建一个控件，并同时对x:Name和Name两个属性进行赋值，那么编译器就会提醒你：Name被设置了多次。

```
<!-- 两个Button分别使用x:Name和Name -->
<Windowx:Class="Cnblog.SetName"
```

```
<Windowx:Class="cnblog.SetName"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="SetName"
        Width="300"
        Height="300">
    <StackPanel>
        <Button x:Name="Button1" Loaded="ButtonLoaded"/>
        <Button Name="Button2" Loaded="ButtonLoaded"/>
    </StackPanel>
    x:Code
    private void ButtonLoaded(object sender, RoutedEventArgs e)
    {
        var button = (Button)sender;
        button.Content = button.Name;
    }
    /x:Code
</Window>
```

两者无区别。

结论：XAML中使用Name其实被映射到x:Name，x:Name才是XAML中唯一的标识，所以它们效果相同。

XAML中x:Name与Name并不完全等价。

不是所有类型都可以使用Name，但是任何类型都可以使用x:Name。

只有拥有Name属性，才可以在XAML中使用Name。不同于x:Name，因为这个是附加属性。并且该类型、或者其父类型标记了RuntimeNameProperty特性，才拥有与x:Name一样的效果。

例如：会报错，因为SolidColorBrush没有Name属性。只能使用x:Name。

为什么要有x:Name

前面提到，XAML中经常需要通过名字来定位某个控件或对象，而SomeWpfType的Name属性，只是一个DP，可以设置两个控件拥有相同的Name属性。

那么这样就非常不利于定位控件，因为Name不是一个唯一的标识了。

使用对象的引用有两个好处：

1.在特定的范围域内，能够保证它的唯一性；

2.在视图树中查找某个对象时，通过引用对象的名称比查找Name属性更加简单。

x:key和x:name的区别，前者是为xaml中定义的资源文件提供唯一的标识，后者是为xaml中定义的控件元素提供唯一标识。

WPF中的XML

很多时候数据是以XML形式存取的，如果把XML节点先转换为CLR数据类型再应用DataTemplate就麻烦了。DataTemplate很智能，具有直接把XML数据节点当作目标对象的功能-----XML数据中的元素名（标签名）可以

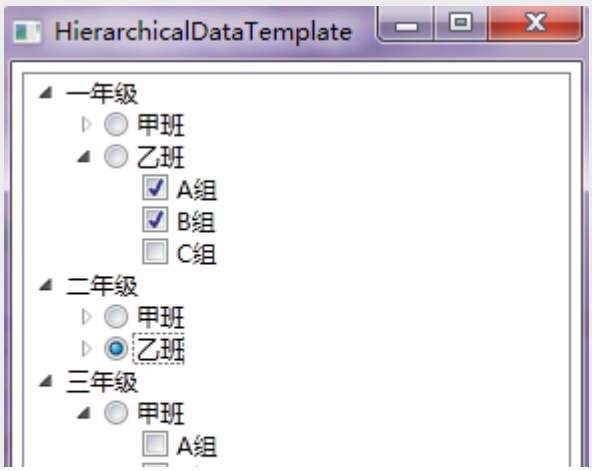
作为DataType，元素的子节点和Attribute可以使用XPath来访问。下面的代码使用XmlDataProvider作为数据源（其XPath指出的必须是一组节点）

XML的优势就是可以方便的表示带有层级的数据，比如：年级----班级----小组或 主菜单---次菜单----三级菜单。同时WPF准备了TreeView和MenuItem控件来显示层级数据。能够帮助层级控件显示层级数据的模板是HierachicalDataTemplate。下面两个常见的例子：

```
<?xml version="1.0"encoding="utf-8" ?>
<Data xmlns="">
 <Grade Name="一年级">
   <Class Name="甲班">
     <Group Name="A组">
     </Group>
     <Group Name="B组">
     </Group>
     <Group Name="C组">
     </Group>
   </Class>
   <Class Name="乙班">
     <Group Name="A组">
     </Group>
     <Group Name="B组">
     </Group>
     <Group Name="C组">
     </Group>
   </Class>
 </Grade>
 <Grade Name="二年级">
   <Class Name="甲班">
     <Group Name="A组">
     </Group>
     <Group Name="B组">
     </Group>
     <Group Name="C组">
     </Group>
   </Class>
   <Class Name="乙班">
     <Group Name="A组">
     </Group>
     <Group Name="B组">
     </Group>
     <Group Name="C组">
     </Group>
   </Class>
 </Grade>
</Data>
```

```
//程序XAML代码如下：
<Windowx:Class="WPFApplication.Window6"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Window6" Height="268"Width="362">
   <Window.Resources>
       <!--数据源-->
       <XmlDataProvider x:Key="ds" Source="XMLStudent.xml" XPath="Data/Grade">
</XmlDataProvider>
       <!--年级模板-->
       <HierarchicalDataTemplate DataType="Grade" ItemsSource="{Binding XPath=Class}">
           <TextBlock Text="{Binding XPath=@Name}"></TextBlock>
       </HierarchicalDataTemplate>
       <!--班级模板-->
       <HierarchicalDataTemplate DataType="Class" ItemsSource="{Binding XPath=Group}">
           <RadioButton Content="{Binding XPath=@Name}" GroupName="gn"></RadioButton>
       </HierarchicalDataTemplate>
       <!--小组模板-->
       <HierarchicalDataTemplate DataType="Group" ItemsSource="{Binding XPath=Student}">
           <CheckBox Content="{Binding XPath=@Name}"></CheckBox>
       </HierarchicalDataTemplate>
   </Window.Resources>
   <Grid>
       <TreeView Margin="5" ItemsSource="{BindingSource={StaticResource ds}}">
       </TreeView>
   </Grid>
</Window>
```

效果如下图所示：



第二个例子是同一种数据类型的嵌套结构，这种情况下只设计一个HierarchicalDataTemplate就可以了，它会产生自动迭代应用效果。

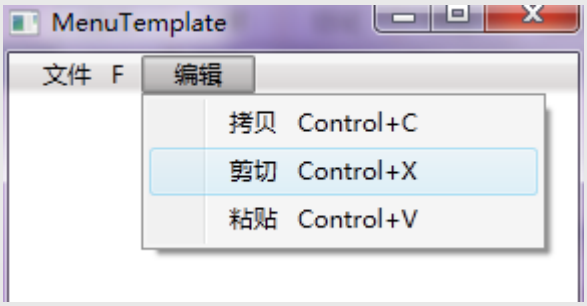
数据依然存放在XML文件中。数据全部是OPeration类型：

```
<?xml version="1.0"encoding="utf-8" ?>
<Data xmlns="">
 <Operation Name="文件" Gesture="F">
   <Operation Name="新建" Gesture="N">
     <Operation Name="项目" Gesture="Ctr+P"/>
     <Operation Name="网站" Gesture="Ctr+W"/>
     <Operation Name="文档" Gesture="Ctr+D"/>
   </Operation>

   <Operation Name="保存" Gesture="S"/>
   <Operation Name="打印" Gesture="P"/>
   <Operation Name="退出" Gesture="X"/>
 </Operation>

 <Operation Name="编辑" Gesture="E">
   <Operation Name="剪切" Gesture="Ctr+X"/>
   <Operation Name="复制" Gesture="Ctr+C"/>
   <Operation Name="粘贴" Gesture="Ctr+V"/>
 </Operation>
</Data>
```

```
//程序XAML代码如下：
<Windowx:Class="WPFApplication.Window7"
       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
       Title="Window7" Height="300" Width="300">
  <Window.Resources>
       <!--数据源-->
    <XmlDataProvider x:Key="ds" Source="MenuXML.xml" XPath="Data/Operation">
</XmlDataProvider>
       <!--Operation模板-->
       <HierarchicalDataTemplate DataType="Operation" ItemsSource="{Binding XPath=Operation}"
>
           <StackPanel Orientation="Horizontal">
                <TextBlockText="{Binding XPath=@Name}" Margin="10,0"></TextBlock>
                <TextBlockText="{Binding XPath=@Gesture}"></TextBlock>
           </StackPanel>
       </HierarchicalDataTemplate>
```

```
  </Window.Resources>
   <StackPanel>
       <Menu ItemsSource="{Binding Source={StaticResourceds}}"></Menu>
   </StackPanel>
</Window>
```

运行效果如下图：



值得一提的是，HierarchicalDataTemplate的作用不是MenuItem的内容而是它的Header。如果对MenuItem的单击事件进行侦听处理，我们就可以从被单击的MenuItem的Header中取出XML数据。

```
//XAML代码如下：
<StackPanelMenuItem.Click="StackPanel_Click">
       <Menu ItemsSource="{Binding Source={StaticResourceds}}"></Menu>
</StackPanel>
```

```
//事件处理代码如下：
private void StackPanel_Click(object sender, RoutedEventArgs e)
{
      MenuItem item = e.OriginalSource as MenuItem;
      XmlElement xe = item.Header as XmlElement;
      MessageBox.Show(xe.Attributes["Name"].Value);
}
```

一旦拿到了数据，使用数据去驱动什么样的逻辑完全由你来决定了。比如可以维护一个CommandHelper类，根据拿到的数据来决定执行什么RoutedCommand。

WPF中的XAML与C#的语法

XAML的语法与C#的语法并不是完全对称的，最明显的就是在设置Binding的时候。 XAML： 这句XAML对应的C#应该是： listBox1.SetBinding(ListBox.ItemsSourceProperty, new Binding(...)); 为了让Binding更详细，往往把它拎出来写，成为这样：

```
Binding b = new Binding("Path");
b.Source = XXXX;
listBox1.SetBinding(ListBox.ItemsSourceProperty, b);
```

向窗体动态添加控件

同样看上去是窗体，WinForm编程对应的类是Form，WPF编程对应的类是Window。虽然在运行时（run time）它们都是Windows API用CreateWindowEx函数创建出来的Window Class，但在它们还是.NET类的时候，却有着巨大的区别——特别是体现在内部控件的组织形式上。 WinForm WinForm窗体里的按钮、文本框等供用户操作的对象称为“控件”（controls）。这些控件可以分为两类，一类是非容器控件，这类控件的内部结构是固定不变的，比如一个Button内部只能是一串文字（还可以设置Button的背景图片），如果你想在Button内显示一个图标后跟上一串文字，要么你写一个自定义控件（派生自Button、再来点儿GDI+的技术）、要么你把文字写在图片上整个作为Button的背景图片；另一类是容器控件，它的内部可以装一些其他控件，这类控件的一个特点就是有一个Controls属性，这是一个ControlCollection。 你可以把Form也看成是一个容器控件。动态添加控件也就是向容器控件内添加控件——方法就是先声明一个与控件类型相对应的变量、为它创建一个控件类型的实例、把这个实例初始化好之后再调用Contols.Add方法，把这个变量添加进容器控件就好了。看起来大概是这样：

```
private void button1_Click(object sender, EventArgs e)
{
      Button button = new Button();
      button.Size = new Size(80, 20);
      button.Text = "OK";
      groupBox1.Controls.Add(button);
}
```

WinForm的控件组织是“平面化”的，也就是说，在一个容器控件内，它们处在同一个ControlCollection内、不再有包含关系（除非它是一个容器控件）。WinForm窗体在包含容器控件时，可以称之为一棵以容器控件为结点的、逻辑上的树，WPF里也有“树”，WPF几乎一切事件消息路由都依赖于这棵树。 要想向Controls里添加一个初始化完备的控件，你几乎总要声明一个变量——在WPF里就不用这么做，因为XAML本身的树状结构，再加上新版本.NET支持“对象初始化”语法，使代码变得非常简单。

WPF UI元素与WinForm控件区别

WPF窗体里的按钮、文本框等UI组件称为“元素”（Element），更确切地说是“UI元素”（与UIElement类对应）。UI元素这个词里隐含了一点，那就是：它一定是“可视”的（Visual），不然怎么让用户去使用呢？ WPF UI元素与WinForm控件最大的不同就是WPF UI元素不再以“容器”和“非容器”作为区分，而是以“内容元素”和“非内容元素”来区分。所谓内容元素，就是说它有一个名为Content的属性—它是Object类型的！Object类是所有.NET类的父类，对于一个内容元素来说，你随便往它里面装什么都可以！装一个UI元素可以、装一组UI元素也可以——把这组元素组合在一个集合里就好了。 如果内容元素的内容仍然是内容元素呢？一棵真正的、可视化的“树”就形成了。这就是WPF中的Visual Tree，WPF窗体上的事件消息也是沿着这棵可视化树传递的——消息经过每个可视化树上的结点（UIElement）时称为“路由”（Route），可以对消息进行处理和控制。

WPF的路由事件

创建一个WPF应用程序，代码如下：

```
<Window x:Class="Wpfceshi.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" MouseDown="Window_MouseDown"  >
    <Grid MouseDown="Grid_MouseDown" x:Name="grid">
        <Button Height="30" Width="100" Content="点击我" MouseDown="Button_MouseDown"/>
```

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

```
using System.Windows;
using System.Windows.Input;
namespace Wpfceshi
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }

        private void Window_MouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Window被点击");
        }

        private void Grid_MouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Grid被点击");
        }

        private void Button_MouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Button被点击");
        }
    }
}
```

调试运行，鼠标右键点击按钮，会依次弹出三个对话框。 为什么点击按钮，Grid和Window也会引发事件呢？其实这就是路由事件的机制，引发的事件由源元素逐级传到上层的元素，Button—>Grid—>Window，这样就导致这几个元素都接收到了事件。

那么如何让Grid和Window不处理这个事件呢？

只需要在Button_MouseDown这个方法中加上e.Handled = true; 这样就表示事件已经被处理，其他元素不需要再处理这个事件了。

```
private void Button_MouseDown(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show("Button被点击");
```

```
    e.Handled = true;
}
```

这时如果需要Grid也参与处理这个事件该怎么做呢？只需要给他AddHandler即可。

修改代码如下

```
public Window1()
{
    InitializeComponent();
    grid.AddHandler(Grid.MouseDownEvent, new RoutedEventHandler(Grid_MouseDown1), true);
}

//再加上这个方法
private void Grid_MouseDown1(object sender, RoutedEventArgs e)
{
     MessageBox.Show("Grid被点击");
}
```

总结：

气泡事件最为常见，它表示事件从源元素扩散（传播）到可视树，直到它被处理或到达根元素。这样您就可以针对源元素的上方层级对象处理事件。例如，您可向嵌入的 Grid 元素附加一个 Button.Click 处理程序，而不是直接将其附加到按钮本身。气泡事件有指示其操作的名称（例如，MouseDown）。

隧道事件采用另一种方式，从根元素开始，向下遍历元素树，直到被处理或到达事件的源元素。这样上游元素就可以在事件到达源元素之前先行截取并进行处理。根据命名惯例，隧道事件带有前缀 Preview（例如 PreviewMouseDown）。

直接事件类似 .NET Framework 中的正常事件。该事件唯一可能的处理程序是与其挂接的委托。

对于隧道事件

```
<Window x:Class="Wpfceshi.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" PreviewMouseDown="Window_PreviewMouseDown">
    <Grid PreviewMouseDown="Grid_PreviewMouseDown" x:Name="grid">
        <Button Height="30" Width="100" Content="点击我"
PreviewMouseDown="Button_PreviewMouseDown"/>
    </Grid>
</Window>
```

```
using System.Windows;
using System.Windows.Input;
namespace Wpfceshi
```

```
{
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
        private void Button_PreviewMouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Button被点击");
        }
        private void Grid_PreviewMouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Grid被点击");
        }
        private void Window_PreviewMouseDown(object sender, MouseButtonEventArgs e)
        {
            MessageBox.Show("Window被点击");
        }
    }
}
```

可以看到，隧道事件的传递刚好与气泡事件相反。

Click Me

挂接事件的 XAML 声明就象 XAML 中的属性分配，但结果是针对指定事件处理程序的对象产生一个正常的事件挂接。此挂接实际上出现在编译时生成的窗口局部类中。要查看这一挂接，转到类的构造 函数，右键单击 InitializeComponent 方法调用，然后从上下文菜单中选择“转到定义”。编辑器将显示生成的代码文件（其命名约定为 .i.g.cs 或 .i.g.vb），其中包括在编译时正常生成的代码。在显示的局部类中向下滚动到 Connect 方法，您会看到下面的内容：

this.myButton.Click += new System.Windows.RoutedEventHandler(this.myButton_Click);

这一局部类是在编译时从 XAML 中生成的，其中包含那些需要设计时编译的 XAML 元素。大部分 XAML 最终都会成为编译后程序集中嵌入了二进制的资源，在运行时会与二进制标记表示的已编译代码合并。

如果看一下窗口的代码隐藏，您会发现 Click 处理程序如下所示：

private void myButton_Click(object sender, RoutedEventArgs e) { }

它看起来就象任何其他 .NET 事件挂接一样——您有一个显式声明的委托，它挂接到一个对象事件且委托指向某个处理方法。使用路由事件的唯一标记是 Click 事件的事件参数类型，即 RoutedEventArgs。

WebBrowser

承载并在HTML文档间导航。在WPF托管代码和HTML脚本间启用互操作性。 WebBrowser.Navigate方法(Uri) 异步导航到位于指定Uri处的文档。 命名空间：System.Windows.Controls 程序集:PresentationFramework(在PresentationFramework.dll中) 语法：public void Navigate(Uri source)

```
string szTmp = “http://192.168.0.11/sample.htm“;
Uri uri = new Uri(szTmp);
CamWeb.Navigate(uri);
```

WebBrowser.Document属性 获取表示所承载的HTML页的文档对象 。MSHTML 是微软的窗口操作系统（Windows）搭载的网页浏览器—Internet Explorer的排版引擎的名称，（又称为Trident）。 MSHTML是微软公司的一个COM组件，该组件封装了HTML语言中的所有元素及其属性，通过其提供的标准接口，可以访问指定网页的所有元素。 MSHTML提供了丰富的HTML文档接口，有IHTMLDocument、IHTMLDocument2、...、IHTMLDocument7等7种。其中，IHTMLDocument只有一个Script属性，是管理页面脚本用的；IHTMLDocument2接口跟C#的HtmlDocument类（即通过Web Browser控件直接获得的Document属性）很相似；IHTMLDocument3是跟Visual Basic 6.0里的文档对象相似的一个接口，基本上可以用到的方法都在其中。MSHTML还提供封装了对HTML元素完整操作的IHTMLElement接口，通过IHTMLElement，可以准确地判断HTML元素节点的类型，还可以获取HTML元素节点的所有属性。

实际应用

WebBrowser控件不停的导航一个.html文件，直到导航成功。 但是如果想要导航的.html文件刚开始不存在，一段时候后才存在的话就有些棘手。 可以采用延时，但是.html产生的时间是不确定的，这样就无法确定延时的秒数。所以可以采用以下方法： 在XAML文件中加入：

```
<WebBrowser Name="Web" Width="640" Height="480"LoadCompleted="Web_LoadCompleted" />1
```

在.CS文件的初始化中加入：

```
string szTmp = "http://192.168.0.11/sample2.htm";
Uri uri = new Uri(szTmp);
Web.Navigate(uri);
```

在.CS文件中实现 LoadCompleted事件：

```
private void Web_LoadCompleted(object sender, NavigationEventArgs e)
{
    ((sender as WebBrowser).Document as mshtml.HTMLDocumentEvents_Event).oncontextmenu += new
  mshtml.HTMLDocumentEvents_oncontextmenuEventHandler(ExtendFrameControl_oncontextmenu);
    mshtml.HTMLDocument dom = (mshtml.HTMLDocument)Web.Document; //定义HTML
    dom.documentElement.style.overflow = "hidden"; //隐藏浏览器的滚动条
    dom.body.setAttribute("scroll", "no"); //禁用浏览器的滚动条
    if (!dom.body.innerHTML.Contains("123456"))
    {
        string szTmp = "http://192.168.0.11/sample2.htm";
        Uri uri = new Uri(szTmp);
        Web.Navigate(uri);
    }
}
```