第 3 章 XAML 界面原理和语法

XAML 是英文 Extensible Application Markup Language 的简写，是用于实例化. NET 对象的标记语言。XAML 是微软技术体系中的 UI 编程语言，在 Windows 10、Windows 8、Windows Phone、Silverlight 和 WPF 技术框架下都可以使用 XAML 的语法来编写程序的界面。在 Windows 10 的普通应用程序(游戏除外)中，也是使用 XAML 来编写程序的界面(HTML5/JS 的开发模式是使用 HTML 和 CSS 的语法来实现界面的编程)，所以对 XAML 语法的理解和掌握是编写 Windows 10 通用应用程序的重要基础。Windows 10 应用程序中的界面是由 xaml 文件组成的，和这些 xaml 文件一一对应起来的是 xaml. cs 文件，这就是微软典型的 Code-Behind 模式的编程方式。xaml 文件的语法类似于 XML 和 HTML 的结合体，这是微软的 XAML 语言特有的语法结构，本章将介绍有关 XAML 方面的语法和原理知识。

3.1 理解 XAML

XAML 是一种声明性标记语言，它简化了为. NET Framework 应用程序创建 UI 的过程，使程序界面编程更加简单和简洁。XAML 直接以程序集中定义的一组特定后备类型表示对象的实例化，就如同其他的基于 XML 的标记语言一样，XAML 大体上也遵循 XML 的语法规则。例如，每个 XAML 元素包含一个名称以及一个或多个属性。XAML 文件中的每个元素代表. NET 中的一个类，并且 XAML 文件中的每个属性代表. NET 类中的一个属性、方法或事件。例如，若要在 Windows 10 的界面上创建一个按钮，可以用下面的 XAML 代码来实现：

```
<Button x:Name = "button 1" BorderThickness = "1" Click = "OnClick1" Content = "按钮" />
```

上面的 XAML 代码中的按钮 Button 实际上是 Windows 10 中的 Windows. UI. Xaml. Controls. Button 类。XAML 的属性是相应类中的相关属性，如上例中的 Name、BorderThickness 实际上是 Button 类中相应的相关属性。在这句 XAML 语句中，还实现了事件处理程序，Click="OnClick1"，即 XAML 支持声明事件处理程序，具体逻辑在其对应的. xaml. cs 的后台代码文件的 OnClick1 方法中。XAML 文件可以映射到一个扩展名为

.xaml.cs 的后台代码文件。这些后台代码文件中的 partial class 包含了 XAML 呈现层可以使用的事件、方法和属性。

编写 XAML 代码时需要注意，声明一个 XAML 元素时，可以用 Name 属性为该元素指定一个名称，这样在 C# 代码里面才可以访问到此元素。这是因为某种类型的元素可能在 XAML 页面上声明多次，但是如果不显式地指明各个元素的 Name 属性，则无法区分哪个是想要操作的元素，也就无法通过 C# 来操作该元素和其中的属性。

XAML 语言有严格的语法标准，所以在编写 XAML 代码的时候需要遵循一些规则。下面是声明一个 XAML 编程必须遵循的 4 大原则：

（1）XAML 是大小写区分的，元素和属性的名称必须严格区分大小写，例如对于 Button 元素来说，其在 XAML 中的声明应该为< Button >，而不是< button >；

（2）所有的属性值，无论它是什么数据类型，都必须包含在双引号中；

（3）所有的元素都必须是封闭的，例如< Button…/>，或者是有一个起始标记和一个结束标记，例如< Button >…</Button >；

（4）最终的 XAML 文件也必须是合适的 XML 文档。

3.2　XAML 语法

可以直接通过 Visual Stuido 或者 Expression Blend 这些开发工具的图形化界面来编辑 Windows 10 的程序界面，这些开发工具很强大，甚至可以不用手工去编写 XAML 的代码就可以完成界面编程。虽然有这些强大工具的支持，但是还是非常有必要去学习和掌握 XAML 的相关语法。掌握好 XAML 的语法才能够更加透彻地理解 Windows 10 界面编程的原理，实现更加复杂的页面编程。下面介绍一些重要的 XAML 语法。

3.2.1　命名空间

前面说了 XAML 里面的元素都是对应着.NET 里面的类，但是只提供类名是不够的。XAML 解析器还需要知道这个类位于哪个.NET 名称空间，这样解析器才能够正确地识别 XAML 的元素。首先来看一个最简单的 Windows 10 界面的 XAML 代码：

```
< Page
    x:Class = " App1.MainPage"
    xmlns = "http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x = "http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local = "using:HelloworldDemo"
    xmlns:d = "http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc = "http://schemas.openxmlformats.org/markup - compatibility/2006"
    mc:Ignorable = "d"
    Background = "{ThemeResource ApplicationPageBackgroundThemeBrush}">
    < Grid x:Name = "LayoutRoot">
    </Grid >
```

```
</Page>
```

上面的代码声明了若干个 XML 命名空间，XAML 文档也是一个完整的 XML 的文档。xmlns 特性是 XML 中的一个特殊特性，它专门用来声明命名空间。一旦声明一个命名空间，在文档中的任何地方都可以使用该命名空间。using:HelloworldDemo 表示引用的是应用程序里的 HelloworldDemo 空间，表示可以在 XAML 里面通过 local 标识符来使用 HelloworldDemo 空间下的控件或者其他类。

在上面的 XAML 代码里面，Grid 元素并没有一个空间引用的前缀。那么 Grid 元素会被解析成哪一个类呢？Grid 类可能是指 Windows. UI. Xaml. Controls. Grid 类，也可能是指位于第三方组件中的 Grid 类，或者在应用程序中定义的 Grid 类等。为了弄清实际上希望使用哪个类，XAML 解析器会检查应用于元素的 XML 命名空间。要弄清楚其运行的原理，先来看一下两个特别的命名空间：

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

这段标记声明了两个命名空间，在创建的所有 Windows 10 的 XAML 文档中都会使用这两个命名空间：

（1）http://schemas. microsoft. com/winfx/2006/xaml/presentation 是 Windows 10 的核心命名空间。它包含了大部分用来构建用户界面的控件类。在该例中，该名称空间的声明没有使用命名空间前缀，所以它成为整个文档的默认命名空间。所以没有前缀的元素都是自动位于这个命名空间下。

（2）http://schemas. microsoft. com/winfx/2006/xaml 是 XAML 命名空间。它包含各种 XAML 实用特性，这些特性可影响文档的解释方式。该名称空间被映射为前缀 x。这意味着可通过在元素名称之前放置名称空间前缀 x 来使用该命名空间，例如上面代码中的 x:Name="LayoutRoot"。

正如在前面看到的，XML 命名空间的名称和任何特定的. NET 名称空间都不匹配。XAML 的创建者选择这种设计的原因有以下两个。

第一个原因是按照约定，XML 命名空间通常是 URI。这些 URI 看起来像是在指定 Web 上的位置，但实际上不是。通过使用 URI 格式的命名空间，不同的 XML 文档格式就会互相区分开来，作为唯一的标识符，表示这是创建在某个特定环境下的 XML 文档，例如 Windows 10 的 XAML 文档就是基于 Windows 10 的. NET 类库的。

另一个原因是 XAML 中使用的 XML 命名空间和. NET 命名空间不是一一对应的，如果一一对应的话，会显著增加 XAML 文档的复杂程度。此处的问题在于，Windows 10 包含了多种命名空间，所有这些命名空间都是以 Windows. UI. Xaml 开头的。如果每个. NET 命名空间都有不同的 XML 命名空间，那就需要为使用的每个控件指定确切的 XML 命名空间，这会使 XAML 文档变得混乱不堪。所以，XAML 将所有这些. NET 命名空间组合到单个 XML 命名空间中。因为在不同的. NET 命名空间中都有一部分. NET 的类，并且所有

这些类的名称都不相同，所以这种设计是可行的。

命名空间信息使得 XAML 解析器可找到正确的类。上面的 Grid 元素没有前缀，那么就会使用默认的空间名，解析器就会到 Windows 10 的 Windows. UI. Xaml 开头的空间下去查找相关的类，最后会查找到 Windows. UI. Xaml. Controls. Grid 类与 Grid 元素匹配起来。

3.2.2　对象元素

XAML 的对象元素是指 XAML 中一个完整的节点，一个 XAML 文件始终只有一个根元素，在 Windows 10 里面通常是 Page 作为根元素，这个根元素就会是当前页面的最顶层的元素。在 XAML 中，除了根元素之外的所有元素都是子元素。根元素只有一个，而子元素理论上可以有无限多个。子元素又可以包含一个或多个子元素，某个元素可以含有子元素的数量由 Windows 10 中具体的类决定。

对象元素语法是一种 XAML 标记语法，它通过声明 XML 元素将 Windows 10 的类或结构实例化。这种语法类似于如 HTML 等其他标记语言的元素语法。对象元素语法以左尖括号(＜)开始，后面紧跟要实例化的类或结构的类型名称。类型名称后面可以有零个或多个空格，对于对象元素还可以声明零个或多个特性，并用一个或多个空格来分隔每个“特性名＝值”这样的属性对。例如定义 Button 的对象元素就有下面的两种实现的语法，两种语法都是正确的：

```
< Button Content = "这是按钮"/>
< Button Content = "这是按钮"></Button >
```

3.2.3　设置属性

XAML 中的属性是也是可以使用多种语法设置的，不同的属性类型也会有不同的设置方式，并不是全部的属性设置都是通用的。总的来说，可以通过下面的 4 种方式来设置对象元素的属性：

(1) 使用属性语法；

(2) 使用属性元素语法；

(3) 使用内容元素语法；

(4) 使用集合语法(通常是隐式集合语法)。

这 4 种设置属性的方法，并不是对所有属性都适用的，有些属性会只适合用一种方式来设置；有一些属性则可以使用多种方式来设置，这取决于属性对象的特性。下面详细地看一下这 4 种属性的设置方法和相关示例：

1. 使用属性语法设置属性

大部分的对象元素都可以使用属性元素语法来设置，这也是最常用的属性设置的语法。使用属性语法设置属性的语法格式如下所示：

```
<objectName propertyName = "propertyValue" .../>
```

或者

```
<objectName propertyName = "propertyValue">
...
</objectName>
```

其中，objectName 是要实例化的对象元素，propertyName 是要对该对象元素设置的属性的名称，propertyValue 是要设置的值。

下面的示例使用属性语法来设置 Rectangle 对象的 Name、Width、Height 和 Fill 属性。

```
<Rectangle Name = "rectangle1" Width = "100" Height = "100" Fill = "Blue" />
```

XAML 分析器会把上面的代码解析成为 C# 的类，当然也可以直接用 C# 的代码来实现对象元素和它的属性设置，下面的 C# 代码可以实现等效的效果：

```
Rectangle rectangle1 = new Rectangle();
rectangle1.Width = 100.0;
rectangle1.Height = 100.0;
rectangle1.Fill = new SolidColorBrush(Colors.Blue);
```

2. 使用属性元素语法设置属性

属性元素语法是指把属性当作一个独立的元素来进行设置，如果要用属性元素语法设置属性，该属性的值也必须是一个 XAML 的对象元素。使用属性元素语法，就需要为要设置的属性创建 XML 元素。这些元素的形式为< Object. Property >，Object 表示对象元素，Property 表示对象元素的属性。在标准的 XML 中，这些元素会被视为在名称中有一个点的元素，但是使用 XAML 时，元素名称中的点将该元素标识为属性元素，表示 Property 是 Object 的属性。

下面用伪代码来表示属性元素语法设置属性的写法。在下面的代码中，Property 表示要设置属性的名称，PropertyValueAsObjectElement 表示声明一个新的 XAML 的对象元素，其值类型是该属性的值。

```
<Object>
    <Object. property>
    PropertyValueAsObjectElement
    </ Object. property>
</Object>
```

下面的示例使用属性元素语法来设置 Rectangle 对象元素的 Fill 属性。因为 Rectangle 对象的 Fill 属性的值其实就是一个 SolidColorBrush 类型的值，所以可以声明一个 SolidColorBrush 的对象来表示 Fill 属性的值。在 SolidColorBrush 中，Color 使用属性语法来设置。

```
<Rectangle Name = "rectangle1" Width = "100" Height = "100">
```

```
    <Rectangle.Fill>
        <SolidColorBrush Color="Blue"/>
    </Rectangle.Fill>
</Rectangle>
```

如果使用属性语法来编写上面的代码，则等同于下面的代码：

```
<Rectangle Name="rectangle1" Width="100" Height="100" Fill=" Blue ">
</Rectangle>
```

3. 使用内容语法设置属性

使用内容语法设置属性是指直接在对象元素节点内容中设置该属性，忽略该属性的属性元素，相当于把该属性的值看作是当前对象元素的内容。这个内容可以是一个或者多个对象元素，这些对象元素也是完全独立的 UI 控件。内容语法和属性元素语法的区别就是，内容语法比属性元素语法少了 Object.Property 的属性表示。内容语法需要较为特殊的属性才能支持，例如常见的 Child 属性、Content 属性都可以使用内容语法进行设置。下面的示例使用内容语法设置 Border 的 Content 属性：

```
<Border>
    <Button Content="按钮"/>
</Border>
```

如果内容属性也支持“松散”的对象模型，在此模型中，属性类型为 Object 类型或者 String 类型，则可以使用内容语法将纯字符串作为内容放入开始对象标记与结束对象标记之间。下面的示例使用内容语法直接用字符串来设置 TextBlock 的 Text 属性：

```
<TextBlock>你好!</TextBlock>
```

4. 使用集合语法设置属性

上面讨论的属性都是非集合的值，如果属性的值是一个集合，就需要使用集合语法设置该属性。使用集合语法来设置属性是一种比较特殊的设置方式，使用这种方式的元素通常都是支持一个属性元素的集合。可以使用 C# 代码的 Add 方法来添加更多的集合元素。使用集合语法设置元素实际上是向对象集合中添加属性项，如下所示：

```
<Rectangle Width="200" Height="150">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0.0" Color="Coral" />
                    <GradientStop Offset="1.0" Color="Green" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Rectangle.Fill>
```

```
</Rectangle>
```

不过,对于采用集合的属性而言,XAML 分析器可根据集合所属的属性隐式知道集合的后备类型。因此,可以省略集合本身的对象元素,上面的代码页可以省略掉 GradientStopCollection,如下所示:

```
<Rectangle Width="200" Height="150">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <LinearGradientBrush.GradientStops>
                <GradientStop Offset="0.0" Color="Coral" />
                <GradientStop Offset="1.0" Color="Green" />
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </Rectangle.Fill>
  </Rectangle>
```

另外,有一些属性不但是集合属性,同时还是内容属性。前面示例中以及许多其他属性中使用的 GradientStops 属性就是这种情况。在这些语法中,也可以省略属性元素,如下所示:

```
<Rectangle Width="200" Height="150">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Offset="0.0" Color="Coral" />
            <GradientStop Offset="1.0" Color="Green" />
        </LinearGradientBrush>
     </Rectangle.Fill>
</Rectangle>
```

集合属性语法最常见于布局控件中,例如 Grid、StackPanel 等布局控件的属性设置。下面的示例演示 StackPanel 面板的属性设置,分别为显示的属性设置和最简洁的属性设置。

显示的 StackPanel 的属性设置:

```
<StackPanel>
    <StackPanel.Children>
        <TextBlock>Hello</TextBlock>
        <TextBlock>World</TextBlock>
    </StackPanel.Children>
</StackPanel>
```

最简洁的 StackPanel 的属性设置:

```
<StackPanel>
    <TextBlock>Hello</TextBlock>
    <TextBlock>World</TextBlock>
```

```
</StackPanel>
```

3.2.4　附加属性

附加属性是一种特定类型的属性，和普通属性的作用并不一样。这种属性的特殊之处在于，其属性值受到 XAML 中专用属性系统的跟踪和影响。附加属性可用于多个控件，但却在另一个类中定义。在 Windows 10 中，附加属性常用于控件布局。

下面解释附加属性的工作原理。每个控件都有各自固有的属性，例如 Button 空间会有 Content 属性来设置按钮的内容等。当在布局面板中放置控件时，根据容器的类型，控件会获得额外特征，例如在 Canvas 布局面板中放置一个按钮，这个按钮要放在面板的什么位置呢？这时候就需要使用附件属性来解决这样的问题。下面来看一下一个附件属性使用的示例，如下所示：

```
<Canvas>
    <Button Canvas.Left = "50"> Hello </Button>
</Canvas>
```

上面使用了附件属性 Canvas.Left＝"50"表示按钮放置在距离 Canvas 面板左边 50 像素的位置。关于布局的知识后续还有更详细的讲解，这里只是作为一个示例演示附加属性。还需要注意的是，该用法在一定程度上类似于一个静态属性，你可以对任何一个对象元素设置附加属性 Canvas.Left，而不是按名称引用任何实例，当然如果对象元素并不在 Canvas 面板里面，设置 Canvas.Left 附加属性不会产生任何的影响。

3.2.5　标记扩展

标记扩展是一个被广泛使用的 XAML 语言概念。通过 XAML 标记扩展来设定属性值，从而可以让对象元素的属性具备更加灵活和复杂的赋值逻辑。常用的 XAML 标记扩展功能包括以下 5 种：

(1) Binding(绑定)标记扩展，实现在 XAML 载入时，将数据绑定到 XAML 对象；

(2) StaticResource(静态资源)标记扩展，实现引用数据字典(ResourceDictionary)中定义的静态资源；

(3) ThemeResource(主题资源)标记扩展，表示系统内置的静态资源；

(4) TemplateBinding(模板绑定)标记扩展，实现在 XAML 页面中，对象模板绑定调用；

(5) RelativeSource(绑定关联源)标记扩展，实现对特定数据源的绑定。

在语法上，XAML 使用大括号{}来表示扩展。例如，下面这句 XAML：

```
<TextBlock Text = "{Binding Source = {StaticResource myDataSource},Path = PersonName}"/>
```

这里有两处使用了 XAML 扩展，一处是 Binding，另一处是 StaticResource，这种用法又称为嵌套扩展，TextBlock 元素的 Text 属性的值为{}中的结果。当 XAML 编译器看到大

括号{}时,把大括号中的内容解释为 XAML 标记扩展。

XAML 本身也定义了一些内置的标记扩展,这类扩展包括: x:Type、x:Static、x:null 和 x:Array。x:null 是一种最简单的扩展,其作用就是把目标属性设置为 null。x:Type 在 XAML 中取对象的类型,相当于 C# 中的 typeof 操作,这种操作发生在编译的时候。x:Static 是用来把某个对象中的属性或域的值赋给目标对象的相关属性。x:Array 表示一个 .NET 数组,x:Array 元素的子元素都是数组元素,它必须与 x:Type 一起使用,用于定义数组类型。使用 x:null 扩展标记把 TextBlock 的 Background 属性设置为 null,如下所示:

```
<TextBlock Text="你好" Background="{x:null} />
```

3.2.6 事件

大多数 Windows 10 应用程序都是由标记和后台代码组成,在一个项目中,XAML 作为.xaml 文件来编写,然后用 C# 语言来编写后台代码文件。当 XAML 文件被编译时,通过 XAML 页面的根元素的 x:Class 属性指定的命名空间和类来表示每个 XAML 页对应的后台代码的位置。事件是 XAML 中常用的语法,下面来看一下事件的语法实现。

事件在 XAML 中基础语法如下:

```
<objectName eventName="eventHandle"/>
```

例如,使用按钮控件的 Click 事件,响应按钮单击效果,代码如下:

```
<Button Content="按钮" Click="Button_Click_1"/>
```

其中,Button_Click 连接后台代码中的同名事件处理程序:

```
Private void Button_Click_1(object sender, RoutedEventArgs e)
{
    //在这里可以添加事件处理的逻辑程序
}
```

在实际项目开发中,Visual Studio 的 XAML 语法解析器为开发人员提供了智能感知功能,通过该功能可以在 XAML 中方便地调用指定事件,而 Visual Studio 将为对应事件自动生成事件处理函数后台代码。

3.3 XAML 的原理

XAML 是 Windows 10 界面编程的语法,承担了如何显示和布局 Windows 10 程序界面的工作。本节将从应用程序运行原理的角度去讲解 XAML 文件的是如何被程序使用的。

3.3.1 XAML 页面的编译

Windows10 的应用程序项目会通过 Visual Studio 完成 XAML 页面的编译,在程序运

行时会通过直接链接操作加载和解析 XAML，将 XAML 和过程式代码自动连接起来。用户不需要将 XAML 文件和过程式代码融合，只需要把它添加到项目中，并通过 Build 动作来完成编译即可，一般公共样式资源的 XAML 文件都是采用这种方式。但是，如果要编译一个 XAML 文件并将它与过程式代码混合，第一步就是为 XAML 文件的根元素指定一个子类，这可以用 XAML 语言命名空间中的 Class 关键字来完成，Windows 10 的程序页面一般采用这种方式。通常在 Windows 10 项目中新增的 XAML 文件都会自动生成一个对应的 XAML.CS 文件，并且默认地将两个文件关联起来。例如，添加的 XAML 文件如下：

```
< Page
    x:Class = "App1.MainPage"
    … >
    …
</ Page >
```

与 XAML 文件关联起来的 XAML.CS 文件如下：

```
namespace App1
{
    public sealed partial class MainPage : Page
    {
    …
    }
}
```

通常把与 XAML 文件关联的 XAML.CS 文件称为代码隐藏文件。如果添加 XAML 中的任何一个事件处理程序(通过事件特性，如 Button 的 Click 特性)，在 XAML.CS 文件上就会生成事件的处理事件代码。在类定义中有一个 partial 关键字，这个关键字很重要，因为类的实现是分布在多个文件中的。可能用户会觉得不可思议，因为在项目里面只看到 MainPage.xaml.cs 文件定义了 MainPage 类，其实 MainPage 类也在另外一个地方定义了，只是在项目工程里面隐藏了而已。当编译完 Windows 10 的项目时，会在项目的 obj\Debug 文件夹下看到 Visual Studio 创建的以 g.cs 为扩展名的文件，对于每一个 XAML 文件，可以找到一个对应的 g.cs 文件。例如，如果项目中有一个 MainPage.xaml 文件，就会在 obj\Debug 文件夹下找到 MainPage.g.cs 文件。下面来看一下 MainPage.g.cs 文件的结构：

```
using System;
…
namespace App1 {
    public partial class MainPage : global::Windows.UI.Xaml.Controls.Page {

[global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.Windows.UI.Xaml.Build.
Tasks"," 14.0.0.0")]
        (global::Windows.UI.Xaml.Controls.Grid LayoutRoot;
```

```
        …
        private bool _contentLoaded;
        [System.Diagnostics.DebuggerNonUserCodeAttribute()]
        public void InitializeComponent() {
            if (_contentLoaded) {
                return;
            }
            _contentLoaded = true;
            global::Windows.UI.Xaml.Application.LoadComponent(this, new global::System.Uri
("ms - appx:///MainPage.xaml"), global::Windows.UI.Xaml.Controls.Primitives.Component-
ResourceLocation.Application);
            LayoutRoot = (global::Windows.UI.Xaml.Controls.Grid)this.FindName("LayoutRoot");
            …
        }
    }
}
```

从 MainPage.g.cs 文件中可以看到,MainPage 类还在这里定义了一些控件和相关的方法,并且 InitializeComponent()方法里面加载和解析了 MainPage.xaml 文件,MainPage.cs 文件里调用的 InitializeComponent()方法就是在 MainPage.g.cs 文件中定义的。在 xaml 页面中声明的控件,通常会在.g.cs 中生成对应控件的内部字段。实际上这取决于控件是否有 x:Name 属性,只要有这个属性,都会自动调用 FindName 方法,用于把字段和页面控件关联。没有 x:Name 属性,就没有字段。这种关联会有一定的性能浪费,因为在应用载入控件的时候,通过 LoadComponents 方法关联,而 xaml 也是在这个时候动态解析。

在项目的 obj\Debug 文件夹下,还找到了 g.i.cs 为扩展名的文件,对于每一个 XAML 文件,也会找到一个对应的 g.i.cs 文件,并且这些 g.i.cs 文件与对应的 g.cs 文件是基本一样的。这些 g.i.cs 文件又有怎样的含义呢? 其实这些 g.i.cs 文件并不是在编译的时候生成的,而是当创建了 XAML 文件的时候就马上生成,或者修改了 XAML 文件 g.i.cs 文件也会跟着改变,而 g.cs 文件则是必须要成功编译了项目之后才会生成的。文件后缀中的 g 表示 generated 产生的意思,i 表示 intellisense 智能感知的意思,g.i.cs 文件是 XAML 文件对应的智能感知文件,在 vs 中利用 go to definition 功能找 InitializeComponent 方法的实现的时候,进入的就是 g.i.cs 文件的 InitializeComponent 方法里面。

3.3.2 动态加载 XAML

动态加载 XAML 是指在程序运行时通过解析 XAML 格式的字符串或者文件来动态生成 UI 的效果。通常情况下,Windows 10 的界面元素都是通过直接读取 XAML 文件的内容来呈现的,如上一小节讲解的那样通过 XAML 文件和 XAML.CS 文件关联起来编译,这也是默认的 UI 实现的方式,但是在某些时候并不能预先设计好所有的 XAML 元素,而是需要在程序运行的过程中动态地加载 XAML 对象,这时候就需要使用到动态加载 XAML 来实现了。

在应用程序里面动态加载 XAML 需要使用到 XamlReader. Load 方法来实现，XamlReader 类是为分析 XAML 和创建相应的 Windows 10 对象树提供 XAML 处理器引擎，XamlReader. Load 方法可以分析格式良好的 XAML 片段并创建相应的 Windows 10 对象树，然后返回该对象树的根。大部分可以在 XAML 页面上编写的代码，都可以通过动态加载 XAML 的形式来实现，不仅仅是普通的 UI 控件，动画等其他的 XAML 代码一样可以动态加载，如：

```
//一个透明度变化动画的 XAML 代码的字符串
private const string FadeInStoryboard =
@"< Storyboard xmlns = ""http://schemas.microsoft.com/winfx/2006/xaml/presentation"">
    < DoubleAnimation
        Duration = ""0:0:0.2""
        Storyboard.TargetProperty = ""(UIElement.Opacity)""
        To = ""1""/>
</Storyboard >";
//使用 XamlReader.Load 方法加载 XAML 字符串并且解析成动画对象
Storyboard storyboard = XamlReader.Load(FadeInStoryboard) as Storyboard;
```

使用 XamlReader. Load 方法动态加载 XAML 对 XAML 的字符串是有一定的要求的，这些“格式良好的 XAML 片段”必须要符合以下要求：

（1）XAML 内容字符串必须定义单个根元素，使用 XamlReader. Load 创建的内容只能赋予一个 Windows 10 对象，它们是一对一的关系。

（2）内容字符串 XAML 必须是格式良好的 XML，并且必须是可分析的 XAML。

（3）所需的根元素还必须指定某一默认的 XML 命名空间值。这通常是命名空间 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"。

下面给出动态加载 XAML 的示例：演示了使用 XamlReader. Load 方法加载 XAML 字符串生成一个按钮和加载 XAML 文件生成一个矩形。

代码清单 3-1：动态加载 XAML（源代码：第 3 章\Examples_3_1）

MainPage. xaml 文件主要代码

```
-------------------------------------------------------------------------
    < Grid x:Name = "ContentPanel" Grid.Row = "1" Margin = "12,0,12,0">
        < StackPanel x:Name = "sp_show">
            < Button x:Name = "bt_addXAML" Content = "加载 XAML 按钮" Click = "bt_addXAML_
Click"></Button >
        </StackPanel >
    </Grid >
```

MainPage. xaml. cs 文件主要代码

```
-------------------------------------------------------------------------
        //加载 XAML 按钮
        private void bt_addXAML_Click(object sender, RoutedEventArgs e)
```

```
        {
            //注意 XAML 字符串里面的命名空间"http://schemas.microsoft.com/winfx/2006/
            //xaml/presentation" 不能少
            string buttonXAML = "<Button xmlns='http://schemas.microsoft.com/winfx/2006/
xaml/presentation'" +
                  " Content=\"加载 XAML 文件\" Foreground=\"Red\"></Button>";
            Button btnRed = (Button)XamlReader.Load(buttonXAML);
            btnRed.Click += btnRed_Click;
            sp_show.Children.Add(btnRed);
        }
        //已加载的 XAML 按钮关联的事件
        async void btnRed_Click(object sender, RoutedEventArgs e)
        {
            string xaml = string.Empty;
            //加载程序的 Rectangle.xaml 文件
            StorageFile fileRead = await Windows.ApplicationModel.Package.Current.Install-
edLocation.GetFileAsync("Rectangle.xaml");
            //读取文件的内容
            xaml = await FileIO.ReadTextAsync(fileRead);
            //加载 Rectangle
            Rectangle rectangle = (Rectangle)XamlReader.Load(xaml);
            sp_show.Children.Add(rectangle);
        }
```

添加 Rectangle.xaml 文件的时候，Build Action 属性默认值为 Page，表示是程序编译的页面，这时候需要手动把 Rectangle.xaml 文件的 Build Action 属性设置为 Content，表示该 xaml 文件是作为 Content 资源来使用。

Rectangle.xaml 文件代码：动态加载到程序里面的 XAML 文件

```
-------------------------------------------------------------------------
<Rectangle xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          Height="100" Width="200">
    <Rectangle.Fill>
        <LinearGradientBrush>
            <GradientStop Color="Black" Offset="0"/>
            <GradientStop Color="Red" Offset="0.5"/>
            <GradientStop Color="Black" Offset="1"/>
        </LinearGradientBrush>
    </Rectangle.Fill>
</Rectangle>
```

程序的运行效果如图 3.1 所示。

图 3.1　动态加载 XAML

3.4　XAML 的树结构

XAML 是界面编程语言，用来呈现用户界面，它具有层次化的特性，它的元素的组成就是一种树的结构类型。XAML 编程元素之间通常以某种形式的“树”关系存在，在 XAML 中创建的应用程序 UI 可以抽象化为一个对象树，也称为元素树，可以进一步将对象树分为两个离散但有时会并行的树：逻辑树和可视化树。逻辑树是根据父控件和子控件来构造而成的，在路由事件中将会按照这样的一种层次结构来触发。可视化树是 XAML 中可视化控件及其子控件组成的一个树形的控件元素结构图，可视化树在控件编程中使用很广泛。

3.4.1　可视化树

在讲解可视化树的概念之前，先来了解一下对象树。对象树是指在 XAML 中创建和存在的对象彼此关联起来的一棵树，对象树里面的对象都是基于对象具有属性这一原则，意思就是说某个 XAML 的元素具有属性，这个元素就是对象树里面的一个对象。在很多情况下对象的属性的值是另一个对象，而此对象也具有属性，这个属性的值就是对象的一个子对象节点。对象树具有分支，因为其中某些属性是集合属性并具有多个对象；并且，对象树具有根，因为体系结构最终必须引用单个对象，而该对象是与对象树之外的概念之间的连接点。

可视化树中包含应用程序的用户界面所使用的所有可视化元素，并通过了树形的数据结构按照父子元素的规则来把这些可视化元素排列起来。可视化树概念指的是较大的对象树在经过编辑或筛选后的表示形式。所应用的筛选器是在可视化树中只存在具有呈现含义的对象。具有呈现含义的对象并不是指一定要显示出来在界面的对象，那些集成在控件里

面并不直接显示的对象也是可视化树里面的元素。

下面来看一下 XAML 代码所包含的可视化树的元素：

```
<StackPanel>
    <TextBox></TextBox>
    <Button Content="确定"></Button>
</StackPanel>
```

从上面的代码中可以看到一个 StackPanel 控件里面包含了两个子控件：一个是 TextBox 控件；一个是 Button 控件。从逻辑树的角度去看，这棵逻辑树只是包含了三个元素——StackPanel、TextBox 和 Button，StackPanel 为父节点，TextBox 和 Button 为子节点。从可视化树的角度去看，里面的元素要比逻辑树的元素要多得多。可视化树的层次结构关系图如图 3.2 所示。

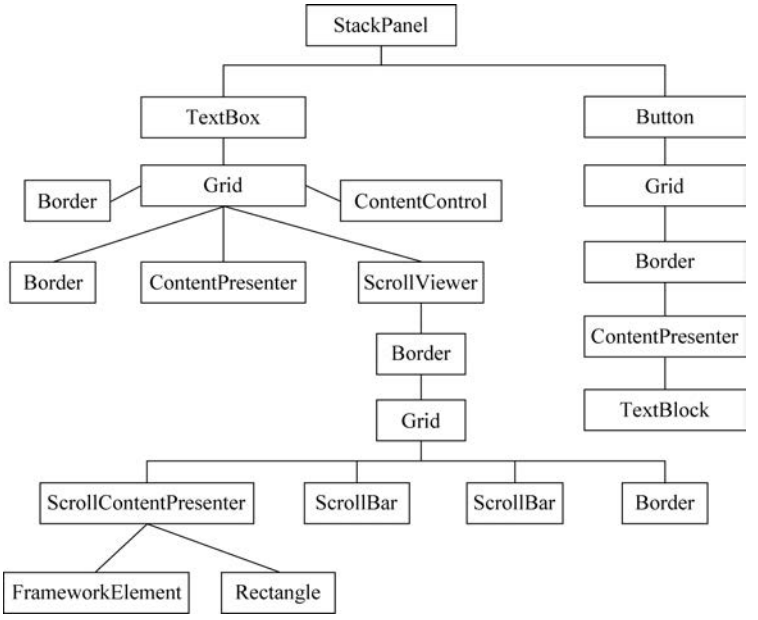

图 3.2　可视化树的层次结构关系图

通过可视化树，可以确定 Windows 10 可视化对象和绘图对象的呈现顺序。将从位于可视化树中最顶层节点的可视化元素根开始遍历，然后按照从左到右的顺序遍历可视化元素根的子级。如果某个可视化元素有子级，则将先遍历该可视化元素的子级，然后再遍历其同级。这意味着子可视化元素的内容先于该可视化元素本身的内容而呈现。

3.4.2　VisualTreeHelper 类

可视化树针对 XAML 的显示操作是在应用程序内部使用，但是在特定情形下，了解有关可视化树的一些信息通常很重要，例如编写或替换控制模板或在运行时分析控件的结构

或部分。对于这些情形，Windows 10 提供了 VisualTreeHelper 类，它通过一种方式检查可视化树，这种方式比通过对象特定的父属性和子属性来实现更加便捷和高效。

VisualTreeHelper 类是一个静态帮助器类，它提供了一个在可视化对象级别编程的低级功能，该类在非常特殊的方案（如开发高性能自定义控件）中非常有用；在大多数情况下，该类给一些更高级的 Windows 10 控件（如 ListBox）提供更大的灵活性。

VisualTreeHelper 类提供了用来枚举可视化树成员的功能。可以在运行时对可视化树执行操作，并且可以遍历到模板部件，这是一种可用来检查模板组成情况的有用手段。此外，可以检查可能通过数据绑定填充的子集合，或者是应用程序代码可能无法全部了解运行时对象树的完整本质的子集合。若要检索父级，请调用 GetParent 方法。若要检索可视化对象的子级或直接子代，请调用 GetChild 方法，此方法返回父级在指定索引处的子对象。VisualTreeHelper 类的 4 个常用的静态方法如表 3.1 所示。

表 3.1　VisualTreeHelper 类的常用静态方法

方　法	说　明
FindElementsInHostCoordinates	检索一组对象，这些对象位于某一对象的坐标空间的指定点或矩形内
GetChild	使用提供的索引，通过检查可视化树获取所提供对象的特定子对象
GetChildrenCount	返回在可视化树中在某一对象的子集合中存在的子级的数目
GetParent	返回可视化树中某一对象的父对象

3.4.3　遍历可视化树

下面给出遍历可视化树的示例，演示使用 VisualTreeHelper 类来遍历 XAML 元素的可视化树。

代码清单 3-2：遍历可视化树（源代码：第 3 章\Examples_3_2）

MainPage.xaml 文件主要代码

```
<Grid x:Name = "ContentPanel" Grid.Row = "1" Margin = "12,0,12,0">
    <StackPanel x:Name = "stackPanel">
        <TextBox ></TextBox >
        <Button Content = "遍历" Click = "Button_Click_1"></Button >
    </StackPanel >
</Grid >
```

MainPage.xaml.cs 文件主要代码

```
string visulTreeStr = "";
//单击事件，弹出 XAML 页面里面 StackPanel 控件的可视化树的所有对象
private async void Button_Click_1(object sender, RoutedEventArgs e)
{
    visulTreeStr = "";
    GetChildType(stackPanel);
```

```
        MessageDialog messageDialog = new MessageDialog(visulTreeStr);
        await messageDialog.ShowAsync();
    }
    //获取某个 XAML 元素的可视化对象的递归方法
    public void GetChildType(DependencyObject reference)
    {
        //获取子对象的个数
        int count = VisualTreeHelper.GetChildrenCount(reference);
        //如果子对象的个数不为 0 将继续递归调用
        if (count > 0)
        {
            for (int i = 0; i <= VisualTreeHelper.GetChildrenCount(reference) - 1; i++)
            {
                //获取当前节点的子对象
                var child = VisualTreeHelper.GetChild(reference, i);
                //获取子对象的类型
                visulTreeStr += child.GetType().ToString() + count + " ";
                //递归调用
                GetChildType(child);
            }
        }
```

程序的运行效果如图 3.3 和图 3.4 所示。

图 3.3　测试页面首页

图 3.4　弹出的可视化树对象的类型

3.5　框架和页面

在一个 Windows 10 的应用程序里面包含一个框架多个页面，框架相当于是应用程序的最外层的一个容器，然后这个容器里面包含了很多个页面，而这些页面都是存在于导航堆栈上。本节将介绍框架和页面的相关知识以及如何在程序开发中使用这种框架和页面的结构。

3.5.1　框架页面结构

Windows 10 应用程序平台提供了框架和页面类，框架类为 Frame，页面类为 Page。Windows 10 应用程序的框架是一个顶级容器控件，该控件可托管 Page，Page 页面又包含应用程序中不同部分的内容，也就是程序界面 UI 的内容。在 Windows 10 里面可以创建所需的任何数目的页面，以便在应用程序中展现内容，然后从框架导航到这些页面。图 3.5 展示了应用程序可能具有的框架和页面层次结构。

图 3.5　框架和页面层次结构图

在一个 Windows 10 应用程序的项目里面，可以看到 App.xaml.cs 页面包含下面的代码：

```
Frame frame = Window.Current.Content as Frame;
```

Window 类的单例对象的 Content 属性就是当前 Windows 10 应用程序最顶层的元素，也就是应用程序的主框架，一个 Windows 10 应用程序只有一个主框架。Window.Current.Content 属性的值是与应用关联的 Frame，每个应用都有一个 Frame，当用户导航到该页面时，导航框架会将应用的每个页面或 Page 的实例设置为框架的 Content。

在 Frame 类里面也提供了一些与应用程序全局相关的属性和方法，例如属性 CanGoBack 表示在当前应用程序中是否可以返回到上一个页面，如果是 false 则表示当前已经是在应用程序的首页了。同时，在 Frame 类里面的 Navigate 方法也很常用，用于导航到新的页面。在第 2 章的 BackButtonDemo 项目里面，实现从 MainPage.xaml 页面跳转到 BlankPage1.xaml 页面的时候就使用了 Frame 类的 Navigate 方法，实现的代码如下所示：

```
this.Frame.Navigate(typeof(BlankPage1));
```

3.5.2　页面导航

Windows 10 应用页面的互相跳转的逻辑是用一个堆栈结构的容器来管理这些页面。应用的导航历史记录是一种后进先出的堆栈结构。该结构还称为后退堆栈，因为它包含的一组页面在后退导航的堆栈结构中，可以将该堆栈看成一叠盘子，添加到该堆栈的最后一个

盘子将是可以移除的第一个盘子，如果想要移除中间的某个碟子，必须把这个碟子上面的碟子先移走。当前最新的页面会被添加到此堆栈的顶部，此操作称为推送操作。删除堆栈顶部项的操作称为弹出操作，通过从堆栈顶部一次删除一个页面，当然也可以检索堆栈中的某些内容。图 3.6 展示了 Windows 10 页面堆栈的概念。

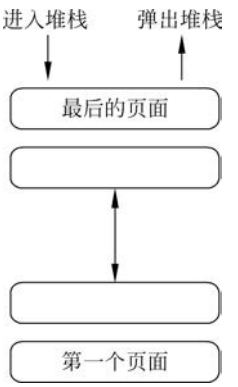

图 3.6　页面堆栈

当应用中的页面调用 Navigate 时，当前页面会被放到后退堆栈上，并且系统将创建并显示目标页的新实例。当在应用的页面之间进行导航时，系统会将多个条目添加到此堆栈。当页面调用 GoBack 时，将放弃当前页面，并将堆栈顶部的页面从后退堆栈中弹出并进行显示。此后退导航会继续弹出并显示，直到堆栈中不再有条目。通常，手机里面的应用程序会使用物理返回键来实现 GoBack 的操作，这时候就需要在项目里面实现物理返回键的事件来进行处理，例如第 2 章的 BackButtonDemo 项目就实现了这样的交互操作。

还可以使用 Frame 类 BackStack 属性获取后退堆栈的项目，后退堆栈的项目为 PageStackEntry 类的对象，PageStackEntry 类表示后退或前进导航历史记录中的一个条目。通过 PageStackEntry 类的 Type 属性和 Parameter 属性可以知道导航过来的 Page 对象的类型和参数。