# .net高级编程

#### 本系列主要包括三个方面内容：

* 第一部分UI部分，重点介绍.net桌面编程框架WPF（Windows Presentation Foundation）
* 第二部分数据访问部分，主要介绍.net的数据库访问技术，包括基础的ADO.net，Linq to Sql，以及.net的OR Mapping框架EF（Entity FrameWork）、
* 第三部分并发和异步，重点介绍TPL、TAP和Rx.net.。

## WPF

UI是绝大多数应用系统必要的组成部分，也是一个应用系统是否成功的重要因素之一。大致来说，现代的UI分成Web应用，主要以Html页面运行在浏览器中；桌面应用，主要运行在PC一类的桌面设备中，是最传统的用户界面模式；终端模式，主要应用于系统的管理配置，不是常见的UI模式；app，严格说也是一种桌面应用，不过更多应用于手机等移动设备；其他UI模式，如语音、AR/VR/MR虚拟技术等。

Web应用来说，传统的模式是服务器端混合模式，由服务器端生产页面，.net对应的框架是Asp.net，包括asp.net WebForm和 asp.net MVC。当前更多的应用模式是前后端分离的模式，前端有Html、CSS、Javascript实现，后端提供服务如Webapi等。

在桌面应用中，微软有着传统的优势和大量的技术积累，.net对应的桌面应用框架有传统的Win Form，以及更现代的WPF，这两种技术框架也正在.net core中获得支持，将会成为跨平台的桌面应用。

App应用，.net对应的技术框架是UWP（Universal Windows Platform）和Xamarian。严格地说，UWP不是简单的app开发技术，从它的名字就可以看出，UWP的目标是跨越Windows平台的各种设备，从手机、平板、Pc以及Xbox和Hololens等。Xamarin则是一种跨平台技术，可以构建跨iOS、Mac OS、Android、Windows平台的应用。

本篇重点介绍WPF技术，WPF技术也是UWP（Windows UI）、Xamarin Form技术的基础。

1. WPF概述
   1. WPF特点

各种UI技术一直在互相借鉴和发展，WPF借鉴了Web页面技术的成功经验，并充分发挥硬件渲染的优势。

相比传统的WinForm，WPF有以下特色：

* 使用标记语言XAML，分离关注点。类似于html+css和JavaScript的关系，XAML负责界面元素的组织和布局，代码负责对用户交互的相应，界面内容的变更等逻辑，从而使得界面设计人员和程序开发者可以分工合作，发挥各自的特长。
* 和Web相似的布局模型。
* 基于页面的导航。
* 使用DirectX技术，充分发挥显卡的能力。
* 对2D、3D、矢量图形的支持。
* 设备分辨率无关性。
* MVVM模式和数据绑定。
* 样式和模板。

|  |
| --- |
| **WPF的设备无关性**  WPF的所有可视化元素都使用设备无关单位度量（device-independent unit），每个设备无关单位被定义为1/96英寸。可视化元素的物理尺寸按如下公式计算：  物理单位尺寸=1/96英寸（设备无关单位尺寸）×系统DPI  这样，就保证了，不管设备的系统DPI怎么改变，实际显示的物理尺寸是不变的。例如一个96×96的按钮，系统DPI为96时，物理单位尺寸为1像素，按钮的物理尺寸为96×96像素，大小为1英寸。如果设备的系统DPI改变为192，则物理单位尺寸为2个像素，按钮就变成192×192像素，尺寸仍为1英寸。 |

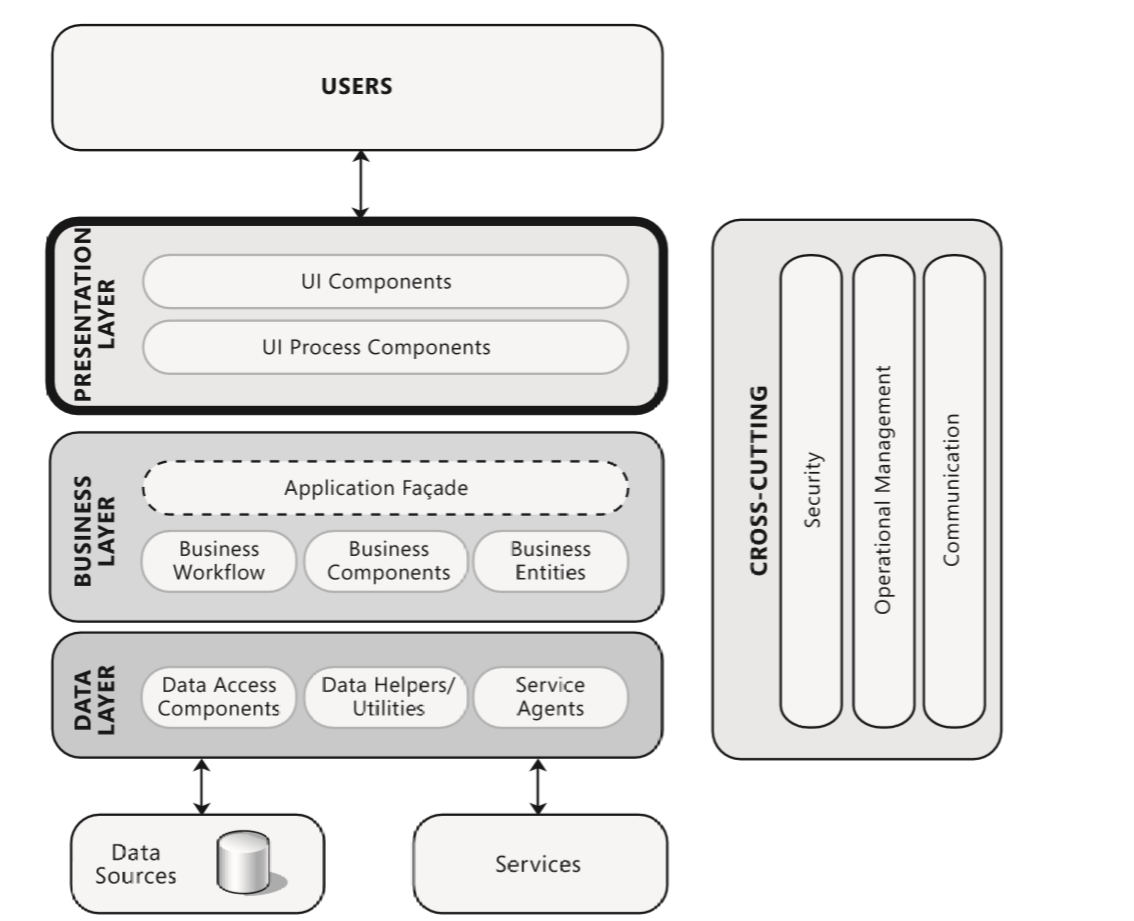
* 1. 用户界面（表示层）考虑的要素[[1]](#footnote-1)

图 1...1 典型的三层架构

表示层通常包含以下部分：

* UI组件：用于呈现信息和接受用户输入的可视化元素。
* 表示逻辑组件：独立于特定的界面实现，定义应用程序的交互逻辑行为和组织。在MVC及其衍生的界面分离实现模式中，由以下两部分组成：Controller、Presenter或者ViewModel；表示层Model，用来封装来自业务层的数据以及实现数据校验等逻辑。

表示层设计首先要考虑的因素是：

* 根据应用需要，选择合适的类型，如Web客户端或者富客户端。
* 根据类型的不同，选取合适的UI技术。
* 使用相关的界面架构设计模式，如MVC、MVP、MVVM。
* 分离关注点。
* 考虑人机交互原则。
* 用户驱动设计为主。

表示层设计需要考虑的主要问题：

* 缓存（Caching）
* 通信（Communication）
* 组合（composition）
* 异常管理（Exception Management）
* 导航（Navigation）
* 用户体验（User Experience）
* 用户界面（User Interface）
* 校验（Validation）
  1. WPF体系结构

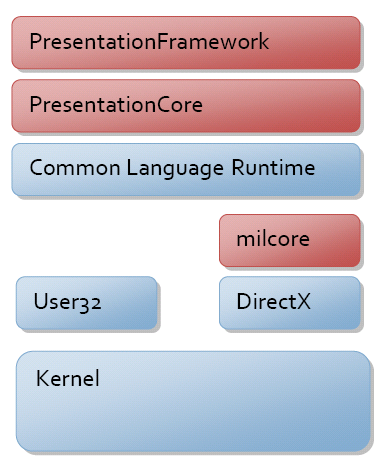


图 1.2 WPF体系结构图

上图描述了WPF的主要组件，图中的红色部分（PresentationFramework、PresentationCore 和 milcore）是 WPF 的主要代码部分。出于性能的考量， milcore 是以非托管代码编写的，目的是实现与 DirectX 的紧密集成。

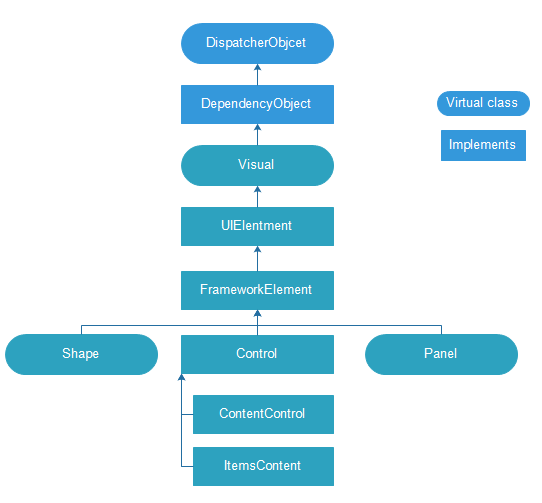


图 1..3 WPF类结构图

1. **System.Threading.DispatcherObject**

绝大多数WPF对象派生自DispatcherObject，DispatcherObject提供了基本的并发和线程处理结构。WPF采用STA模型（Single Thread Affinity），dispatcher是有多个优先队列的基本消息分派系统，继承此类的对象可以验证是否运行在正确的线程上。

1. **System.Windows.DependencyObject**

提供依赖属性和附加属性的支持。

1. **System.Windows.Media.Visual**

Visual类用来构建可视化对象树，每个可视化对象包含绘制指令以及渲染这些指令的元数据。Visual是WPF组合系统的真正入口点，是WPF的两个托管子系统和非托管的milcore的连接点。每个继承自该类的对象都可以在窗口中显示出来。

1. **System**.Windows.UIElement

UIElement 定义核心子系统，包括布局、 输入和事件，还引入了 CommandBindings 的概念。

1. **System.Windows.FrameworkElement**

FrameworkElement引入了将数据绑定和样式，动态资源引用支持，逻辑树概念。

1. **System.Windows.Controls.Control**

Control最重要的功能是模板化。数据模型（属性）、交互模型（命令和事件）及显示模型（模板）之间的划分，可实现对控件的外观和行为的完全自定义。

* 1. Application类、应用程序管理概述

WPF中，应用程序范围的公共功能封装在Application类。 Application类包括以下功能：

* 对应用程序的生存期进行跟踪并与之进行交互。
* 检索和处理命令行参数。
* 检测和响应未经处理的异常。
* 共享应用程序范围的属性和资源。
* 管理独立应用程序中的窗口。
* 跟踪和管理导航。

下表列出了Application中的常见任务以及如何实现它们：

表 1.1

| **任务** | **方法** |
| --- | --- |
| 获取表示当前应用程序的对象 | 使用 [Application.Current](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.current) 属性。 |
| 将启动屏幕添加到应用程序中 | 请参阅[将初始屏幕添加到 WPF 应用程序](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/how-to-add-a-splash-screen-to-a-wpf-application)。 |
| 启动应用程序 | 使用 [Application.Run](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.run) 方法。 |
| 停止应用程序 | 使用[Shutdown](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.shutdown)方法的[Application.Current](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.current)对象。 |
| 从命令行获取参数 | 处理[Application.Startup](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.startup)事件并使用[StartupEventArgs.Args](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.startupeventargs.args)属性。 有关示例，请参阅[Application.Startup](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.startup)事件。 |
| 获取和设置应用程序退出代码 | 设置[ExitEventArgs.ApplicationExitCode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.exiteventargs.applicationexitcode)中的属性[Application.Exit](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.exit)事件处理程序或调用[Shutdown](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.shutdown)方法并传入一个整数。 |
| 检测和响应未经处理的异常 | 处理[DispatcherUnhandledException](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.dispatcherunhandledexception)事件。 |
| 获取和设置应用程序范围的资源 | 使用 [Application.Resources](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.resources) 属性。 |
| 使用应用程序范围的资源字典 | 请参阅[使用应用程序范围的资源字典](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/how-to-use-an-application-scope-resource-dictionary)。 |
| 获取和设置应用程序范围的属性 | 使用 [Application.Properties](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.properties) 属性。 |
| 获取和保存应用程序的状态 | 请参阅[保持和还原应用程序作用域属性在应用程序会话之间](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/persist-and-restore-application-scope-properties)。 |
| 管理非代码数据文件，包括资源文件、内容文件和源站点文件。 | 请参阅[WPF 应用程序资源、 内容和数据文件](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/wpf-application-resource-content-and-data-files)。 |
| 管理独立应用程序中的窗口 | 请参阅 [WPF 窗口概述](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/wpf-windows-overview)。 |
| 跟踪和管理导航 | 请参阅[导航概述](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/navigation-overview)。 |

自定义Application类：

XAML方式：

<Application

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

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

x:Class="SDKSample.App" />

C#代码方式：

using System.Windows;

namespace SDKSample

{

public partial class App : Application { }

}

系统构建是会自动生成main入口点，创建Application实例，调用InitiallizeComponent（）方法，调用Run（）方法启动程序。

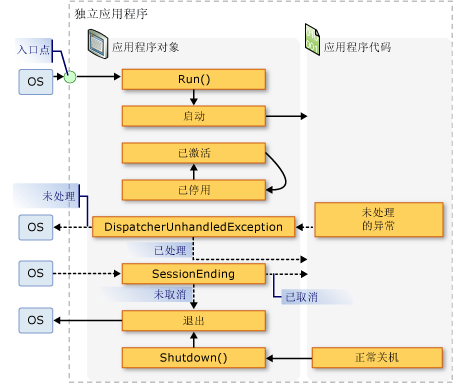


图 1.4 Application生存期事件

获取当前AppDomain的Application对象：

// Get current application

Application current

调用Run（）方法并完成初始化之后，触发Startup事件，可以在此启动窗口，出来命令行参数等，

<Application

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

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

x:Class="SDKSample.App"

Startup="App\_Startup" />

using System.Windows;

namespace SDKSample

{

public partial class App : Application

{

void App\_Startup(object sender, StartupEventArgs e)

{

// Open a window

MainWindow window = new MainWindow();

window.Show();

}

}

}

<Application

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

StartupUri="HomePage.xaml" />

using System;

using System.Windows;

using System.Windows.Navigation;

namespace SDKSample

{

public partial class App : Application

{

void App\_Startup(object sender, StartupEventArgs e)

{

((NavigationWindow)this.MainWindow).Navigate(new Uri("HomePage.xaml", UriKind.Relative));

}

}

}

处理命令行参数：

<Application

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

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

x:Class="SDKSample.App"

Startup="App\_Startup" />

using System.Windows;

namespace SDKSample

{

public partial class App : Application

{

void App\_Startup(object sender, StartupEventArgs e)

{

// Application is running

// Process command line args

bool startMinimized = false;

for (int i = 0; i != e.Args.Length; ++i)

{

if (e.Args[i] == "/StartMinimized")

{

startMinimized = true;

}

}

// Create main application window, starting minimized if specified

MainWindow = new MainWindow();

if (startMinimized)

{

mainWindow.WindowState = WindowState.Minimized;

}

mainWindow.Show();

}

}

}

下面的代码演示如何处理[Activated](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.activated)和[Deactivated](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.deactivated)事件，以确定应用程序是否处于活动状态。

XAML

<Application

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

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

x:Class="SDKSample.App"

StartupUri="MainWindow.xaml"

Activated="App\_Activated"

Deactivated="App\_Deactivated" />

C#

using System;

using System.Windows;

namespace SDKSample

{

public partial class App : Application

{

bool isApplicationActive;

void App\_Activated(object sender, EventArgs e)

{

// Application activated

this.isApplicationActive = true;

}

void App\_Deactivated(object sender, EventArgs e)

{

// Application deactivated

this.isApplicationActive = false;

}

}

}

其他事件的处理详见MSDN文档。

* 1. [演练：我的第一个 WPF 桌面应用程序](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/getting-started/walkthrough-my-first-wpf-desktop-application)
  2. 本篇后续主要探讨的内容
* XAML
* 控件和布局
* 依赖属性和附加属性
* 事件、路由事件、命令
* 样式和行为、触发器
* MVVM和数据绑定
* 资源和模板
* 页面和导航
  1. 小结

本章从宏观上对WPF做了一个简要的介绍，介绍了软件设计中的用户界面设计需要考虑的主要因素；介绍了WPF的体系结构和主要类；较详细介绍了Application类及其生命周期事件；通过动手演练对WPF程序有一个整体的认识。

1. XAML

**本章主要内容：**

* **XAML语法**
* **标记扩展**
* **类型转换器**
* **命名空间**
* **自定义前缀和自定义类型**
* **事件和代码隐藏**
* **元素树**

由于Attribute、Property在中文中没有统一约定的术语加以区分，并且Attribute在Xaml和CLR中有不同的语义，本文没有特别说明，如果上下文中不能明确识别，统一约定使用Attribute表示Xaml的Attribute，使用Property原文代表CLR中对象的Property，使用中文“属性”代表CLR中的Attribute。

* 1. XAML概述

XAML（eXtensible Application Markup Language，可扩展性应用程序标记语言），是一种基于XML的标记性语言。理论上，可以不使用XAML，而使用纯代码方式也能实现WPF应用程序的开发，但这不是开发WPF的正确姿势。在实际的WPF开发中，都是采用XAML和代码结合的方式。作为一种标记性语言，类似于Html，XAML非常适合用于用户界面的描述，和专门的设计工具结合，界面设计人员和编码人员可以分工协作，各自完成自己擅长的工作。使用XAML描述视图，从而和操纵控制逻辑分离，实现良好的界面实现模式（MVVM）。从而方便复用和测试。

XAML本身并不是WPF的一部分，它还可以用于UWP、Xamarin Form的开发，以及WF（工作流）的描述等，也可以用来开发Web应用。

图形用户界面由一系列可视化元素按一定的形式通过布局容器组织在一起，是一种树形结构，但现有的大多数呈现模式，给我们只有平面的观感，XAML基于XML，天生适合描述树形结构，很方便用来描述界面元素之间的关系。XAML的标记扩展，可以方便定义需要的资源，从而达到资源复用的目的。

* 1. XAML语法

XAML遵守XML语法，对一些特殊字符以及空白的处理也遵从XML规则。

|  |
| --- |
| ExpenseItHome.xaml  <Page x:Class="ExpenseIt.ExpenseItHome"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  xmlns:local="clr-namespace:ExpenseIt"  mc:Ignorable="d"  d:DesignHeight="300" d:DesignWidth="300"  Title="ExpenseIt - Home">  <Grid Margin="10,0,10,10">  <Grid.Resources>  <!-- Expense Report Data -->  <XmlDataProvider x:Key="ExpenseDataSource" XPath="Expenses">  <x:XData>  <Expenses xmlns="">  <Person Name="Mike" Department="Legal">  <Expense ExpenseType="Lunch" ExpenseAmount="50" />  <Expense ExpenseType="Transportation" ExpenseAmount="50" />  </Person>  <Person Name="Lisa" Department="Marketing">  <Expense ExpenseType="Document printing"  ExpenseAmount="50"/>  <Expense ExpenseType="Gift" ExpenseAmount="125" />  </Person>  <Person Name="John" Department="Engineering">  <Expense ExpenseType="Magazine subscription"  ExpenseAmount="50"/>  <Expense ExpenseType="New machine" ExpenseAmount="600" />  <Expense ExpenseType="Software" ExpenseAmount="500" />  </Person>  <Person Name="Mary" Department="Finance">  <Expense ExpenseType="Dinner" ExpenseAmount="100" />  </Person>  </Expenses>  </x:XData>  </XmlDataProvider>  <!-- Name item template -->  <DataTemplate x:Key="nameItemTemplate">  <Label Content="{Binding XPath=@Name}"/>  </DataTemplate>  </Grid.Resources>  <Grid.Background>  <ImageBrush ImageSource="watermark.png"/>  </Grid.Background>  <Grid.ColumnDefinitions>  <ColumnDefinition Width="230" />  <ColumnDefinition />  </Grid.ColumnDefinitions>  <Grid.RowDefinitions>  <RowDefinition/>  <RowDefinition Height="Auto"/>  <RowDefinition />  <RowDefinition Height="Auto"/>  </Grid.RowDefinitions>  <!-- People list -->  <Label Style="{StaticResource headerTextStyle}" Grid.Column="1" >  查看费用报告  </Label>  <Border Grid.Row="1" Style="{StaticResource listHeaderStyle}" Grid.Column="1" >  <Label Style="{StaticResource listHeaderTextStyle}">姓名</Label>  </Border>  <ListBox Name="peopleListBox"  ItemsSource="{Binding Source={StaticResource ExpenseDataSource}, XPath=Person}"  ItemTemplate="{StaticResource nameItemTemplate}" Grid.Column="1" Grid.Row="2" />  <!-- View report button -->  <Button Grid.Row="3" Style="{StaticResource buttonStyle}" Click="Button\_Click" Grid.Column="1" >View</Button>    </Grid>  </Page> |

|  |
| --- |
| ExpenseItHome.xam.cs  using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  using System.Windows;  using System.Windows.Controls;  using System.Windows.Data;  using System.Windows.Documents;  using System.Windows.Input;  using System.Windows.Media;  using System.Windows.Media.Imaging;  using System.Windows.Navigation;  using System.Windows.Shapes;  namespace ExpenseIt  {  /// <summary>  /// ExpenseItHome.xaml 的交互逻辑  /// </summary>  public partial class ExpenseItHome : Page  {  public ExpenseItHome()  {  InitializeComponent();  }  private void Button\_Click(object sender, RoutedEventArgs e)  {  // View Expense Report  ExpenseReportPage expenseReportPage = new ExpenseReportPage(this.peopleListBox.SelectedItem);  this.NavigationService.Navigate(expenseReportPage);  }  }  } |

* XAML元素（Element）（或按照Html习惯称之为标签

一个XAML元素对应CLR中的对象。

* Attribute语法

XAML的Attribute对应对象的Property或事件。

* Property元素语法

和Attribute语法具有相同效果，但可以处理一些特殊情况，比如空白的处理，换行的处理。标记的语法以<typeName.propertyName>开始。

|  |
| --- |
| <Button>  <Button.Background>  <SolidColorBrush Color="Blue"/>  </Button.Background>  <Button.Foreground>  <SolidColorBrush Color="Red"/>  </Button.Foreground>  <Button.Content>  This is a button  </Button.Content>  </Button> |

* 集合语法

如果Property是集合类型，那么Property元素下的子元素标记项成为集合中的项。

* XAML内容属性（Content Property）

通常用来简化标记，以便直接嵌套父子关系的元素。内容属性的语法要求在类声明中使用ContentPropertyAttribute属性，一个类只能指定一个Property为XAML内容属性，这样在设置Property时省略property元素。

XAML内容属性的元素必须在其他Property元素之前或之后，不能混合在一起，对集合也是如此。

|  |
| --- |
| <Border>  <TextBox Width="300"/>  </Border>  <!--explicit equivalent-->  <Border>  <Border.Child>  <TextBox Width="300"/>  </Border.Child>  </Border> |

XAML内容属性常和集合语法组合使用。

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  >  <StackPanel>  <Button>Button 1</Button>  <Button>Button 2</Button>  <Button>Button 3</Button>  </StackPanel>  </Page> |

* 1. 标记扩展（Markup Extension）

标记扩展是XAML语言概念，用一对大括号（ {、} ）语法表示标记扩展。

XAMK解析器通常把Attribute语法中的文本字符串解释为基本类型或使用后文将介绍到的类型转换器解释为引用对象，标记扩展则提供了一种引用其他现有对象的方法。如在前面代码示例中：

<Border Grid.Row="1" Style="{StaticResource listHeaderStyle}" Grid.Column="1" >

<Label Style="{StaticResource listHeaderTextStyle}">姓名</Label>

</Border>

Border和Label的Style Attribute引用了app.xaml中定义在资源中的样式：

|  |
| --- |
| <Application x:Class="ExpenseIt.App"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:local="clr-namespace:ExpenseIt"  StartupUri="MainWindow.xaml">  <Application.Resources>  <!-- Header text style -->  <Style x:Key="headerTextStyle">  <Setter Property="Label.VerticalAlignment" Value="Center"></Setter>  <Setter Property="Label.FontFamily" Value="Trebuchet MS"></Setter>  <Setter Property="Label.FontWeight" Value="Bold"></Setter>  <Setter Property="Label.FontSize" Value="18"></Setter>  <Setter Property="Label.Foreground" Value="#0066cc"></Setter>  </Style>  <!-- Label style -->  <Style x:Key="labelStyle" TargetType="{x:Type Label}">  <Setter Property="VerticalAlignment" Value="Top" />  <Setter Property="HorizontalAlignment" Value="Left" />  <Setter Property="FontWeight" Value="Bold" />  <Setter Property="Margin" Value="0,0,0,5" />  </Style>  <!-- DataGrid header style -->  <Style x:Key="columnHeaderStyle" TargetType="{x:Type DataGridColumnHeader}">  <Setter Property="Height" Value="35" />  <Setter Property="Padding" Value="5" />  <Setter Property="Background" Value="#4E87D4" />  <Setter Property="Foreground" Value="White" />  </Style>  <!-- List header style -->  <Style x:Key="listHeaderStyle" TargetType="{x:Type Border}">  <Setter Property="Height" Value="35" />  <Setter Property="Padding" Value="5" />  <Setter Property="Background" Value="#4E87D4" />  </Style>  <!-- List header text style -->  <Style x:Key="listHeaderTextStyle" TargetType="{x:Type Label}">  <Setter Property="Foreground" Value="White" />  <Setter Property="VerticalAlignment" Value="Center" />  <Setter Property="HorizontalAlignment" Value="Left" />  </Style>  <!-- Button style -->  <Style x:Key="buttonStyle" TargetType="{x:Type Button}">  <Setter Property="Width" Value="125" />  <Setter Property="Height" Value="25" />  <Setter Property="Margin" Value="0,10,0,0" />  <Setter Property="HorizontalAlignment" Value="Right" />  </Style>  </Application.Resources>  </Application> |

Style的值推迟到资源查找时计算。这里的StaticResource代表StaticResourceExtension类，listHeaderTextStyle是该类构造函数的参数，在资源定义中通过x:Key="listHeaderStyle"标识。

**XAML定义的标记扩展，不只应用于WPF**：

|  |  |
| --- | --- |
| x:Type | 为命名类型提供Type对象，常用于样式和模板。  TargetType="{x:Type Button}" |
| X:Static | 生成静态值。  {x:Static SystemColors.ControlBrushKey} |
| X:Null | 设置Property的值为Null，可以用在Attribute或Property元素语法中， |
| X:Array | 对XAML语法创建常规数组提供支持。 |

**WPF专用的标记扩展**：

|  |  |
| --- | --- |
| StaticResource | 用已定义的资源的值设置Property，在XAML加载时计算，运行时不能改变。 |
| DynamicResource | 与静态不同的是，在运行时计算引用的值，这也意味着可以在运行时改变。 |
| Binding | 数据绑定。 |
| RelativeSource | 为绑定提供源信息，在运行时对象树中确定路径， |
| TemplateBinding | 用于控件模板。 |
| ColorConvertedBitmap | 支持相对高级的影像场景。 |
| ComponentResourceKey  ThemeDictionary | 支持资源查找，特别是和自定义控件打包在一起的资源和主题。 |

每个标记扩展都对应一个MarkupExtension的派生类，一般遵循\*Extension的命名规则。标记扩展可以嵌套。

<Setter Property="Background"

Value="{DynamicResource {x:Static SystemColors.ControlBrushKey}}" />

* 1. 类型转换器

XAML的Attribute语法Attribute=Value中，Value只能是字符串，这只能处理一些Property为基本类型或简单类型的情况，对于复杂类型就勉为其难了。要解决这个问题，有两种方法，一种是前文提到的Property元素语法，一种是使用TypeConverter类型转换器，往往能提供一种更简洁的写法。

例如，Thickness结构启用了TypeConverterAttribute，我们就可以用简洁的方式设置Button的Margin Property：

<Button Margin="10,20,10,30" Content="Click me"/>

否则，使用Property元素语法，就要这样：

<Button Content="Click me">

<Button.Margin>

<Thickness Left="10" Top="20" Right="10" Bottom="30"/>

</Button.Margin>

</Button>

自己实现类型转换器，首先要派生TypeConverter的子类，如StringToXXXTypeConverter，然后在类XXX上使用TypeConverterAttribute：

[TypeConverterAttribute(typeof(StringToXXXTypeConverter))]

Public class XXX

{

……

}

* [CanConvertTo](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.typeconverter.canconvertto)
* [CanConvertFrom](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.typeconverter.canconvertfrom)
* [ConvertTo](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.typeconverter.convertto)
* [ConvertFrom](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.typeconverter.convertfrom)

其中最重要的是ConvertFrom，将Attribute的值转换为Property对象；另一个重要的是ConvertTo，转换方向相反。

* 1. XAML命名空间

每个xaml文件有且只有一个根元素，根元素包含xmlns Attribute。

<Page x:Class="WPFApplication1.MainPage"

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

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

xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary">

...

<custom:ExampleClass/>

...

</Page>

第一个是默认命名空间，对应CLR中相应的命名空间，其他命名空间Attribute名字形如 xmlns：xxx，有些是系统为我们生成，有些是我们自己定义的。自定义的命名控件在引用其他程序集时使用。Xmlns:x映射单独的XAML命名空间，将其映射到x:前缀。

* 1. X:前缀、自定义前缀和XAML中的自定义类型

**x:前缀**映射xnlns:x命名空间<http://schemas.microsoft.com/winfx/2006/xaml>，常用的x:前缀有：

|  |  |
| --- | --- |
| x:Key | 设置资源字典中每个资源的唯一键值。 |
| x:Class | 只能用在XAML产品的根元素（相当于WPF入口），用于指定代码隐藏类的类名。  <object x:Class="namespace.classname"...>  ...  </object> |
| x:Name | 唯一标识XAML名字作用域内的XAML元素，运行时代码可以访问该对象。WPF的Name Attribute与此等价。  <StackPanel Name="buttonContainer">  <Button Click="RemoveThis">  Click to remove this button</Button>  </StackPanel>  C#  void RemoveThis(object sender, RoutedEventArgs e)  {  FrameworkElement fe = e.Source as FrameworkElement;  if (buttonContainer.Children.Contains(fe))  {  buttonContainer.Children.Remove(fe);  }  } |
| x:Static | 引用类的静态成员 |
| x:Type | 用类型名创建类型 |

**自定义前缀**在根元素的自定义xmlns定义，使用自定义前缀的元素，XAML解析器会解析成对应命名空间程序集的类型，称之为XAML自定义类型（代码隐藏及其他主程序中的自定义类不需要自定义前缀）。

**自定义类型**用作XAML元素必须满足：

* 自定义类必须时public且有默认公共构造函数。
* 不能是嵌套类。

类要启用TypeConverterAttribute，或者在Property级别声明类型转换器。有时候还需要创建TypeConverter的子类。对于事件，要实现显式的add和remove方法。Property如果要使用Content Property，还需要启用ContentPropertyAttribute。

* 1. 事件和代码隐藏（Code Behind）

事件的Attribute的值对应分部类中的事件处理函数的方法名。XAML也可以用x:code标签内嵌C#代码，但一般不建议这样用。

* 1. 元素树

如果在 XAML 中定义一个应用程序页，然后加载 XAML，将根据标记中元素之间的嵌套关系来创建树结构。WPF中，有两种概念化的对象树：逻辑树和可视化树。

逻辑树定义在WPF框架级别，FrameworkElement或FrameworkContentElement是逻辑树操作的关系最密切的元素，逻辑树中不包含隐式元素，一般使用逻辑树进行资源查找。

可视化树描述的是视觉对象结构，它们的基类是Visual类。为控件编写模板时，重新定义了适用于该控件的可视化树，可视化树展示了可视化细节，路由事件是通过可视化树查找的。

逻辑树和可视化树可借助LogicalTreeHelper和VisualTreeHelper来遍历。

1. 控件和布局

图形用户界面是一系列UI组件按照一定方式组织在一起，UI组件用于呈现信息和接受用户输入，内容应该是UI的核心。WPF在传统UI事件驱动基础上，引入了数据驱动的理念。UI元素可以分成布局和控件，布局主要负责组织控件的排列方式，控件负责呈现数据，接受用户的输入。

WPF的布局借鉴Web页面的布局方式，采用流式布局，使控件的大小和排列方式可以随着窗体的尺寸变化而改变，达到更好的显示效果，WPF引入的内容模型和控件模板，可以让用户方便地改变控件的外观。

* 1. 布局系统

WPF的界面元素采用Box模型，每个FrameworkElement被看作布局中的一个矩形，矩形的尺寸通过计算可用的屏幕空间、限制的尺寸、特定于布局的Property（如Margin、Padding）以及父面板的个别行为而确定。调整父元素的大小特征（如Boarder）会影响子元素的尺寸。FrameworkElement边界框的大小可以用LayoutInformation类的GetLayoutSlot方法获得。

布局是一个递归系统，描述面板元素的子元素集合成员的测量和排列过程，实现对元素的大小调整、定位和绘制。以下是布局系统的过程描述：

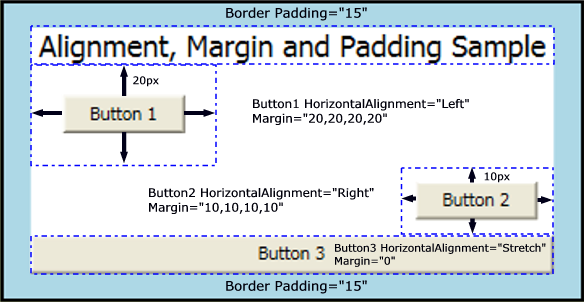
* 子UIElement对象通过首先测量其核心Properties来开始布局过程。
* 计算FrameworkElement和尺寸相关的Properties如Width、Height、Margin等
* 采用面板特定的逻辑，如DockPanel的Dock，StackPanel的Orientation
* 测量所有子对象后，排列或定位其内容
* 绘3制子对象集合的元素
* 如果向子集合中添加元素，调用LayoutTransform或UpdateLayout方法，或者影响测量和排列的依赖属性发生改变，将再次调用布局处理。

**Alignment、Margin 和 Padding**

控件的位置时通过Alignment、Margin和Padding精确控制的。以下代码：

|  |
| --- |
| // Create the application's main Window.  mainWindow = new Window ();  mainWindow.Title = "Margins, Padding and Alignment Sample";  // Add a Border  myBorder = new Border();  myBorder.Background = Brushes.LightBlue;  myBorder.BorderBrush = Brushes.Black;  myBorder.Padding = new Thickness(15);  myBorder.BorderThickness = new Thickness(2);  myStackPanel = new StackPanel();  myStackPanel.Background = Brushes.White;  myStackPanel.HorizontalAlignment = HorizontalAlignment.Center;  myStackPanel.VerticalAlignment = VerticalAlignment.Top;  TextBlock myTextBlock = new TextBlock();  myTextBlock.Margin = new Thickness(5, 0, 5, 0);  myTextBlock.FontSize = 18;  myTextBlock.HorizontalAlignment = HorizontalAlignment.Center;  myTextBlock.Text = "Alignment, Margin and Padding Sample";  Button myButton1 = new Button();  myButton1.HorizontalAlignment = HorizontalAlignment.Left;  myButton1.Margin = new Thickness(20);  myButton1.Content = "Button 1";  Button myButton2 = new Button();  myButton2.HorizontalAlignment = HorizontalAlignment.Right;  myButton2.Margin = new Thickness(10);  myButton2.Content = "Button 2";  Button myButton3 = new Button();  myButton3.HorizontalAlignment = HorizontalAlignment.Stretch;  myButton3.Margin = new Thickness(0);  myButton3.Content = "Button 3";  // Add child elements to the parent StackPanel.  myStackPanel.Children.Add(myTextBlock);  myStackPanel.Children.Add(myButton1);  myStackPanel.Children.Add(myButton2);  myStackPanel.Children.Add(myButton3);  // Add the StackPanel as the lone Child of the Border.  myBorder.Child = myStackPanel;  // Add the Border as the Content of the Parent Window Object.  mainWindow.Content = myBorder;  mainWindow.Show (); |

运行效果如下图：



对齐分HorizonAlignment和VerticalAlignment，分别对应水平对齐和垂直对齐。

**HorizonAlignment**

| **成员** | **描述** |
| --- | --- |
| [Left](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.horizontalalignment#System_Windows_HorizontalAlignment_Left) | 子元素与父元素的已分配布局空间的左端对齐。 |
| [Center](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.horizontalalignment#System_Windows_HorizontalAlignment_Center) | 子元素与父元素的已分配布局空间的中心对齐。 |
| [Right](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.horizontalalignment#System_Windows_HorizontalAlignment_Right) | 子元素与父元素的已分配布局空间的右端对齐。 |
| [Stretch](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.horizontalalignment#System_Windows_HorizontalAlignment_Stretch) （默认值） | 拉伸子元素以填充父元素的已分配布局空间。 显式[Width](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.width)和[Height](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.height)值优先。 |

**VerticalAlignment**

| **成员** | **描述** |
| --- | --- |
| [Top](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.verticalalignment#System_Windows_VerticalAlignment_Top) | 子元素与父元素的已分配布局空间的顶端对齐。 |
| [Center](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.verticalalignment#System_Windows_VerticalAlignment_Center) | 子元素与父元素的已分配布局空间的中心对齐。 |
| [Bottom](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.verticalalignment#System_Windows_VerticalAlignment_Bottom) | 子元素与父元素的已分配布局空间的底端对齐。 |
| [Stretch](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.verticalalignment#System_Windows_VerticalAlignment_Stretch) （默认值） | 拉伸子元素以填充父元素的已分配布局空间。 显式[Width](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.width)和[Height](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.height)值优先。 |

**Margin**设定元素间距，可以指定一个值，代表四个一样的边距，Margin=“20”，也可以指定四个值，Margin=“0，10，15，25”。

**Padding**等于给元素加个边框，放大了元素的尺寸。

**布局性能考量**：

* 尽可能使用满足功能的最高效的面板
* 要注意哪些Property值的改变会影响布局系统的递归更新
* 尽可能用RenderTransform代替LayoutTransform
* 避免UpdateLayout的不必要调用
* 子集合较大时，使用[VirtualizingStackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingstackpanel)而不是普通的StackPanel
* 自上而下构造逻辑树
  1. 面板（Panel）

在WPF中，面板负责UI元素的布局。面板有一个名为Children的内容Property，面板容纳的元素都是Children中的一项，面板负责对它们的大小、维度、位置和排列方式计算渲染。

Panel是所有面板的基类，WPF定义了一套派生类的实现，可支持许多复杂的布局。还可以通过定义Panel的派生类，重写ArrangeOverride和MeasureOverride方法，实现自定义面板。

面板类广泛使用了附加属性（Attached Property），子元素存储由父元素（面板）定义的依赖属性的唯一值，通知面板它们应该如何呈现。

每个面板元素都封装自己的特殊功能，如下表所示。

| **元素名称** | **是否为 UI 面板？** | **描述** |
| --- | --- | --- |
| [Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas) | 是 | 定义一个区域，在其中可以显式定位子元素相对于[Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas)区域的坐标，子元素通过四个附加属性：Top、Left、Right、Bottom来定位。子元素定位可以超出Canvas的范围，如果[ClipToBounds](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.uielement.cliptobounds)设置为true，会导致剪裁。 |
| [DockPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.dockpanel) | 是 | 定义一个区域，可在其中使子元素相互水平或垂直排列。子元素设置附加属性Dock的值为：Top、Left、Right、Bottom来设置停靠位置。默认最后一个子元素将填满剩余未分配位置，可以设置LastChildFill为false来禁止。 |
| [Grid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.grid) | 是 | 定义一个由列和行组成的灵活网格区域。 子元素可以精确地使用[Margin](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.margin)定位。Grid通过RowDefinitions和ColumnDefinition定义行数和列数。多个子元素可以位于同一单元格内，一个子元素也可以跨多个单元格。 |
| [StackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stackpanel) | 是 | 将子元素按顺序排列成水平或垂直排放。 |
| [TabPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.tabpanel) | 否 | 处理中的选项卡按钮的布局[TabControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabcontrol)。 |
| [ToolBarOverflowPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.toolbaroverflowpanel) | 否 | 排列中的内容[ToolBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.toolbar)控件。 |
| [UniformGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.uniformgrid) | 否 | [UniformGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.uniformgrid) 用于与所有相等的单元格大小排列在网格中的子级。 |
| [VirtualizingPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingpanel) | 否 | 为可以“虚拟化”其子级集合的面板提供基类。 |
| [VirtualizingStackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingstackpanel) | 是 | 在水平或垂直方向上将内容排列为一行并使其虚拟化。 |
| [WrapPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.wrappanel) | 是 | 子元素的从左到右依次排列，一行排满后，将内容切换到下一行。顺序从上到下，还是从左到右，具体取决于[Orientation](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.wrappanel.orientation)值。 |

每个派生Panel元素以不同方式处理大小调整约束。

| **Panel 名称** | **x 维度** | **y 维度** |
| --- | --- | --- |
| [Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas) | 按内容约束 | 按内容约束 |
| [DockPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.dockpanel) | 约束 | 约束 |
| [StackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stackpanel) （垂直方向） | 约束 | 按内容约束 |
| [StackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stackpanel) （水平方向） | 按内容约束 | 约束 |
| [Grid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.grid) | 约束 | 在情况下的约束，除非采用自动行列。 |
| [WrapPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.wrappanel) | 按内容约束 | 按内容约束 |

* 1. 控件

WPF沿用传统，把应用程序中的可见对象称为“控件”，但控件并不必然继承于Control类。WPF控件的内容模型，可以方便地在空间中放置数据、图片甚至其他控件等；控制模板允许更改控件的外观而无需创建新的子类。

大多数常用控件和传统用法相似，细节可查阅有关文档。只要清楚控件的功能和应用场景，再通过实践，就可以轻松掌握控件的合理使用。

用户也可以自定义控件，来满足自己特殊的需求。自定义控件涉及依赖属性【路由事件】数据模板等概念，本章中不具体讲述，我们将在学习完有关知识后，通过具体实例，了解如何自定义控件。

**控件的按功能分类：**

布局：

* [Border](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.border)
* [BulletDecorator](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.bulletdecorator)
* [Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas)
* [DockPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.dockpanel)
* [Expander](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.expander)
* [Grid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.grid)
* [GridSplitter](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridsplitter)
* [GroupBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.groupbox)
* [Panel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.panel)
* [ResizeGrip](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.resizegrip)
* [Separator](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.separator)
* [ScrollBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.scrollbar)
* [ScrollViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.scrollviewer)
* [StackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stackpanel)
* [Thumb](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.thumb)
* [Viewbox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.viewbox)
* [VirtualizingStackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingstackpanel)
* [Window](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.window)
* [WrapPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.wrappanel)

按钮：

* [Button](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.button)
* [RepeatButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.repeatbutton)

数据显示：

* [DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)
* [ListView](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listview)
* [TreeView](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.treeview)

日期显示和选择：

* [Calendar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.calendar)
* [DatePicker](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datepicker)

菜单：

* [ContextMenu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contextmenu)
* [Menu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.menu)
* [ToolBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.toolbar)

选择：

* [CheckBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.checkbox)
* [ComboBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.combobox)
* [ListBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listbox)
* [RadioButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.radiobutton)
* [Slider](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.slider)

导航：

* [Frame](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.frame)
* [Hyperlink](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.hyperlink)
* [Page](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.page)
* [NavigationWindow](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationwindow)
* [TabControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabcontrol)

对话框：

* [OpenFileDialog](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.win32.openfiledialog)
* [PrintDialog](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.printdialog)
* [SaveFileDialog](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.win32.savefiledialog)

用户（提示）信息：

* [AccessText](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.accesstext)
* [Label](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.label)
* [Popup](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.popup)
* [ProgressBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.progressbar)
* [StatusBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.statusbar)
* [TextBlock](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textblock)
* [ToolTip](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tooltip)

文档：

* [DocumentViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.documentviewer)
* [FlowDocumentPageViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentpageviewer)
* [FlowDocumentReader](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentreader)
* [FlowDocumentScrollViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentscrollviewer)
* [StickyNoteControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stickynotecontrol)

输入：

* [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox)
* [RichTextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.richtextbox)
* [PasswordBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.passwordbox)

媒体（图像、音频、视频）

* [Image](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.image)
* [MediaElement](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.mediaelement)
* [SoundPlayerAction](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.soundplayeraction)

数字墨迹：

* [InkCanvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.inkcanvas)
* [InkPresenter](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.inkpresenter)

按内容模型分类：

**包括任意内容**

| **包含任意内容的类** | **内容** |
| --- | --- |
| [ContentControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol) | 一个任意对象。 |
| [HeaderedContentControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headeredcontentcontrol) | 一个标头和一个项（两者都是任意对象）。 |
| [ItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.itemscontrol) | 一个任意对象集合。 |
| [HeaderedItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headereditemscontrol) | 一个标头和一个项集合（全部都是任意对象）。 |

包含一个任意对象的控件，它们继承自ContentControl，拥有Content Property：

* [Button](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.button)
* [ButtonBase](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.buttonbase)
* [CheckBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.checkbox)
* [ComboBoxItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.comboboxitem)
* [ContentControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol)
* [Frame](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.frame)
* [GridViewColumnHeader](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumnheader)
* [GroupItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.groupitem)
* [Label](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.label)
* [ListBoxItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listboxitem)
* [ListViewItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listviewitem)
* [NavigationWindow](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationwindow)
* [RadioButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.radiobutton)
* [RepeatButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.repeatbutton)
* [ScrollViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.scrollviewer)
* [StatusBarItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.statusbaritem)
* [ToggleButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.togglebutton)
* [ToolTip](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tooltip)
* [UserControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.usercontrol)
* [Window](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.window)

包含一个标头和一个任意对象的控件，继承自[HeaderedContentControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headeredcontentcontrol)，[HeaderedContentControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headeredcontentcontrol)继承了ContentControl，并定义了Header Property：

* [Expander](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.expander)
* [GroupBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.groupbox)
* [TabItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabitem)

包含一个任意对象集合的控件，继承自ItemsControl（继承自Comtrol），其内容属性为ItemsSource和Items：

* [Menu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.menu)
* [MenuBase](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.menubase)
* [ContextMenu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contextmenu)
* [ComboBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.combobox)
* [ItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.itemscontrol)
* [ListBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listbox)
* [ListView](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listview)
* [TabControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabcontrol)
* [TreeView](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.treeview)
* [Selector](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.selector)
* [StatusBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.statusbar)

包含一个标头和一个任意对象集合的控件，继承自[HeaderedItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headereditemscontrol)，[HeaderedItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headereditemscontrol)继承自了temsControl，定义了Header Property：

* [MenuItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.menuitem)
* [ToolBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.toolbar)
* [TreeViewItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.treeviewitem)

Panel类内容属性是Children，它是UIElement对象的集合：

* [Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas)
* [DockPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.dockpanel)
* [Grid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.grid)
* [TabPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.tabpanel)
* [ToolBarOverflowPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.toolbaroverflowpanel)
* [ToolBarPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.toolbarpanel)
* [UniformGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.uniformgrid)
* [StackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.stackpanel)
* [VirtualizingPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingpanel)
* [VirtualizingStackPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.virtualizingstackpanel)
* [WrapPanel](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.wrappanel)

Decorator类在子UIElement上或周围应用视觉效果，它的内容属性是Child，是一个UIElement对象：

* [AdornerDecorator](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.adornerdecorator)
* [Border](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.border)
* [BulletDecorator](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.bulletdecorator)
* [ButtonChrome](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.windows.themes.buttonchrome)
* [ClassicBorderDecorator](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.windows.themes.classicborderdecorator)
* [InkPresenter](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.inkpresenter)
* [ListBoxChrome](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.windows.themes.listboxchrome)
* [SystemDropShadowChrome](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.windows.themes.systemdropshadowchrome)

可让用户输入文本的类：

| **控件** | **文本显示方式** | **内容属性** |
| --- | --- | --- |
| [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox) | 纯文本 | [Text](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox.text) |
| [RichTextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.richtextbox) | 带格式文本 | [Document](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.richtextbox.document) |
| [PasswordBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.passwordbox) | 隐藏文本（字符已屏蔽） | [Password](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.passwordbox.password) |

显示文本的类：

| **控件** | **内容属性** | **内容属性类型** |
| --- | --- | --- |
| TextBlock | Text | String  适合一致格式的文本 |
| Inlines | Inline对象的集合  适合多种格式的文本 |
| [FlowDocumentPageViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentpageviewer) | Document | [IDocumentPaginatorSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.idocumentpaginatorsource) |
| [FlowDocumentReader](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentreader) | Document | [FlowDocument](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.flowdocument) |
| [FlowDocumentScrollViewer](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.flowdocumentscrollviewer) | Document | [FlowDocument](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.flowdocument) |

* 1. ListBox、ListView、DataGrid

应用程序中，我们经常会涉及到集合的操作，WPF提供了一系列列表控件支持此类操作，主要有：ListBox、ComboBox、ListView、TreeView、DataGrid，此节中重点介绍其中的三个。

**ListBox**

ListBox是最简单的列表控件。虽然简单，但结合WPF的内容模型和数据模板，也能实现比较复杂的应用。

<ListBox Name="peopleListBox"

ItemsSource="{Binding Source={StaticResource ExpenseDataSource}, XPath=Person}"

ItemTemplate="{StaticResource nameItemTemplate}" Grid.Column="1" Grid.Row="2" />

ListBox可以通过SelectionMode设置单选或多选模式。

使用SelectedIndex、SelectedItem、SelectedItems获取选中项的索引和内容，SelectionChanged事件监控选择的变化。

可以用以下方法获取特定索引处的ListBoxItem：

<ListBox Margin="10,0,0,5" Name="lb" VerticalAlignment="Top" Grid.Column="0" Grid.Row="2">

<ListBoxItem>Item 0</ListBoxItem>

<ListBoxItem>Item 1</ListBoxItem>

<ListBoxItem>Item 2</ListBoxItem>

<ListBoxItem>Item 3</ListBoxItem>

</ListBox>

private void GetIndex0(object sender, RoutedEventArgs e)

{

ListBoxItem lbi = (ListBoxItem)

(lb.ItemContainerGenerator.ContainerFromIndex(0));

Item.Content = "The contents of the item at index 0 are: " +

(lbi.Content.ToString()) + ".";

}

ListBox包含项目过多时，会影响滚动性能，可以通过设置VirtualizingStackPanel.VirtualizationMode附加属性为VirtualizationMode.Recycling提升性能：

<StackPanel>

<StackPanel.Resources>

<src:LotsOfItems x:Key="data"/>

</StackPanel.Resources>

<ListBox Height="150" ItemsSource="{StaticResource data}"

VirtualizingStackPanel.VirtualizationMode="Recycling" />

</StackPanel>

**ListView**

ListView派生自ListBox，增加了对基于列显示的支持。通过设置View Property，实现View的解耦，从而提供了在不同的布局和视图中显示数据集的能力。

WPF为ListView的View给出了默认的实现GridView，下面的示例演示如何为显示员工信息的ListView控件定义GridView：

<ListView ItemsSource="{Binding Source=

{StaticResource EmployeeInfoDataSource}}">

<ListView.View>

<GridView AllowsColumnReorder="true"

ColumnHeaderToolTip="Employee Information">

<GridViewColumn DisplayMemberBinding=

"{Binding Path=FirstName}"

Header="First Name" Width="100"/>

<GridViewColumn DisplayMemberBinding=

"{Binding Path=LastName}"

Width="100">

<GridViewColumnHeader>Last Name

<GridViewColumnHeader.ContextMenu>

<ContextMenu MenuItem.Click="LastNameCM\_Click"

Name="LastNameCM">

<MenuItem Header="Ascending" />

<MenuItem Header="Descending" />

</ContextMenu>

</GridViewColumnHeader.ContextMenu>

</GridViewColumnHeader>

</GridViewColumn>

<GridViewColumn DisplayMemberBinding=

"{Binding Path=EmployeeNumber}"

Header="Employee No." Width="100"/>

</GridView>

</ListView.View>

</ListView>

显示结果如下：



将数据绑定到ListView：

<ListView ItemsSource="{Binding Source=

{StaticResource EmployeeInfoDataSource}}">

然后将数据集的字段绑定到GridViewColumn：

<GridViewColumn DisplayMemberBinding=

"{Binding Path=FirstName}"

Header="First Name" Width="100"/>

数据模板和样式的设定可以参考后续章节。

GridView列标题的样式和模板相关的Properties：

|  |  |  |  |
| --- | --- | --- | --- |
| **类**  Property | [GridView](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview) | [GridViewColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumn) | [GridViewColumnHeader](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumnheader) |
| **上下文菜单** | [ColumnHeaderContextMenu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview.columnheadercontextmenu) | 不适用 | [ContextMenu](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.contextmenu) |
| **工具提示** | [ColumnHeaderToolTip](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview.columnheadertooltip) | 不适用 | [ToolTip](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.tooltip) |
| **页眉模板**  **（两项同时设定第一项优先）** | [ColumnHeaderTemplate](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview.columnheadertemplate) [ColumnHeaderTemplateSelector](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview.columnheadertemplateselector) | [HeaderTemplate](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumn.headertemplate)  [HeaderTemplateSelector](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumn.headertemplateselector) | [ContentTemplate](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol.contenttemplate)  [ContentTemplateSelector](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol.contenttemplateselector) |
| **样式** | [ColumnHeaderContainerStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridview.columnheadercontainerstyle) | [HeaderContainerStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.gridviewcolumn.headercontainerstyle) | [Style](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.style) |

**DataGrid**

DataGrid是个功能强大的列表控件，可以用来显示和编辑来自各种不同数据源的数据，如数据库、Linq查询或其他任何可绑定的数据源。

Columns Property包含表格中各列的定义，列的类型是DataGridColumn，可以显示文本、控件等各种WPF内容。默认提供以下四种类型：

| **列名称** | **数据显示** |
| --- | --- |
| [DataGridHyperlinkColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridhyperlinkcolumn?view=netframework-4.7.2) | 用于显示 URI 数据。 |
| [DataGridComboBoxColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcomboboxcolumn?view=netframework-4.7.2) | 用于显示枚举数据。 |
| [DataGridTextColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridtextcolumn?view=netframework-4.7.2) | 用于显示文本。 |
| [DataGridCheckBoxColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcheckboxcolumn?view=netframework-4.7.2) | 用于显示布尔数据。 |

还可以使用[DataGridTemplateColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridtemplatecolumn?view=netframework-4.7.2)，用数据模板定义列，充分利用数据模板的强大功能。早前的例子中：

<DataGrid ItemsSource="{Binding XPath=Expense}" ColumnHeaderStyle="{StaticResource columnHeaderStyle}" AutoGenerateColumns="False" RowHeaderWidth="0" >

<DataGrid.Columns>

<DataGridTemplateColumn Header="ExpenseType" CellTemplate="{StaticResource typeItemTemplate}" />

<DataGridTemplateColumn Header="Amount" CellTemplate="{StaticResource amountItemTemplate}" />

</DataGrid.Columns>

</DataGrid>

就使用了数据模板定义列。

数据模板的定义：

<DataTemplate x:Key="typeItemTemplate">

<Label Content="{Binding XPath=@ExpenseType}"/>

</DataTemplate>

<!-- Amount item template -->

<DataTemplate x:Key="amountItemTemplate">

<Label Content="{Binding XPath=@ExpenseAmount}"/>

</DataTemplate>

下表列出了常见的DataGrid任务以及如何操作：

| **方案** | **方法** |
| --- | --- |
| 交替的背景色 | 设置[AlternationIndex](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.itemscontrol.alternationindex)属性设置为 2 或更大，然后[RowBackground](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.rowbackground)和[AlternatingRowBackground](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.alternatingrowbackground)的Brush。 |
| 定义单元格和行选择行为 | 设置 [SelectionMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionmode) 和 [SelectionUnit](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionunit) 属性。 |
| 自定义标头、 单元格和行的可视外观 | 设置[ColumnHeaderStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.columnheaderstyle)， [RowHeaderStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.rowheaderstyle)， [CellStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.cellstyle)，或[RowStyle](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.rowstyle)。 |
| 设置调整大小选项 | 设置[Height](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.height)， [MaxHeight](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.maxheight)， [MinHeight](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.minheight)， [Width](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.width)， [MaxWidth](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.maxwidth)，或[MinWidth](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.minwidth)属性。 有关详细信息，请参阅[DataGrid 控件中调整大小选项](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/sizing-options-in-the-datagrid-control)。 |
| 访问选定项 | 检查[SelectedCells](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectedcells)属性来获取所选单元格和[SelectedItems](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.multiselector.selecteditems)属性来获取所选的行。 有关详细信息，请参阅 [SelectedCells](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectedcells)。 |
| 自定义最终用户交互 | 设置[CanUserAddRows](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuseraddrows)， [CanUserDeleteRows](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserdeleterows)， [CanUserReorderColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserreordercolumns)， [CanUserResizeColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserresizecolumns)， [CanUserResizeRows](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserresizerows)，并[CanUserSortColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canusersortcolumns)属性。 |
| 取消或更改自动生成的列 | 处理[AutoGeneratingColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.autogeneratingcolumn)事件。 |
| 冻结列 | 设置[FrozenColumnCount](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.frozencolumncount)， 并通过设置列的DisplayIndex将冻结的列移动到可滚动列的左侧。 |
| 使用 XML 数据作为数据源 | 参见早先的例子，[ItemsSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.itemscontrol.itemssource)绑定用 XPath 查询的表示的项的集合。 通过绑定数据源的属性设置每个列。 有关示例，请参见 [DataGridTextColumn](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridtextcolumn)。 |

下表列出了MSDN有关操作示例：

| **标题** | **描述** |
| --- | --- |
| [演练：DataGrid 控件中显示的 SQL Server 数据库中的数据](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/walkthrough-display-data-from-a-sql-server-database-in-a-datagrid-control) | 介绍如何设置新的 WPF 项目，添加实体框架元素、 设置源，并显示中的数据[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)。 |
| [如何：将行详细信息添加到 DataGrid 控件](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/how-to-add-row-details-to-a-datagrid-control) | 介绍如何创建用于行详细信息[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)。 |
| [如何：使用 DataGrid 控件实现验证](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/how-to-implement-validation-with-the-datagrid-control) | 描述如何验证中的值[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)单元格和行，以及显示验证反馈。 |
| [DataGrid 控件中的默认键盘和鼠标行为](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/default-keyboard-and-mouse-behavior-in-the-datagrid-control) | 介绍如何与之交互[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)控件中的使用键盘和鼠标。 |
| [如何：分组、 排序和筛选数据的 DataGrid 控件](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/how-to-group-sort-and-filter-data-in-the-datagrid-control) | 描述如何查看中的数据[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)分组、 排序和筛选的数据通过不同的方式。 |
| [DataGrid 控件中的重设大小选项](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/controls/sizing-options-in-the-datagrid-control) | 描述如何控制中的绝对和自动大小调整[DataGrid](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid)。 |

DataGrid控件的鼠标和键盘行为：

默认键盘行为：

| **键或键组合** | **描述** |
| --- | --- |
| 向下键 | 将焦点移到当前单元格的正下方的单元格。 如果焦点位于最后一行中，按向下键没有任何影响。 |
| 向上键 | 将焦点移到当前单元格的正上方的单元格。 如果焦点位于第一行中，按向上键没有任何影响。 |
| 向左键 | 将焦点移到上一个单元格的行中。 如果焦点位于行中的第一个单元中，按向左键没有任何影响。 |
| 向右键 | 将焦点移到下一个单元格的行中。 如果焦点位于行中的最后一个单元中，按向右箭头没有任何影响。 |
| Home | 将焦点移到当前行中的第一个单元。 |
| End | 将焦点移到当前行中的最后一个单元。 |
| PAGE DOWN | 向下翻页 |
| PAGE UP | 向上翻页 |
| Tab | 将焦点移到下一个单元格。如果焦点位于最后一个单元格，将焦点移到tab顺序的下一个控件。  如果当前单元格处于编辑模式并按 tab 键，将引起焦点从当前行移开，对行所做的任何更改会在焦点改变前提交。 |
| Shift+Tab | 和Tab类似，但焦点移动方向向前。 |
| Ctrl+向下键 | 将焦点移到当前列中的最后一个单元格。 |
| Ctrl+向上键 | 将焦点移到当前列中的第一个单元格。 |
| Ctrl+向右键 | 将焦点移到当前行中的最后一个单元。 |
| Ctrl+向左键 | 将焦点移到当前行中的第一个单元。 |
| CTRL + HOME | 将焦点移到控件中的第一个单元。 |
| CTRL + END | 将焦点移到控件中的最后一个单元。 |
| Ctrl+Page Down | 和PageDown相同。 |
| Ctrl+Page Up | 和PageUp相同。 |
| F2 | 如果[DataGrid.IsReadOnly](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.isreadonly)属性是false并且[DataGridColumn.IsReadOnly](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcolumn.isreadonly)属性是false，当前单元格进入编辑模式。 |
| Enter | 提交到当前单元格和行的任何更改并将焦点移到当前单元格的正下方的单元格。 如果焦点位于最后一行中，无需移动焦点。 |
| Esc | 如果控件处于编辑模式下，取消编辑，撤销在控件中所做的任何更改。 如果基础数据源实现[IEditableObject](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.ieditableobject)，第二次按 esc 键取消整个行的编辑模式。 |
| 退格符 | 编辑单元格时，请删除光标位置前的字符。 |
| DELETE | 编辑单元格时，光标位置后删除的字符。 |
| Ctrl+Enter | 提交当前单元格任何更改而无需将焦点移动。 |
| Ctrl+A | 如果[SelectionMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionmode)设置为[Extended](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridselectionmode#System_Windows_Controls_DataGridSelectionMode_Extended)，选中所有行。 |

以下选择键修改多行的选择。

* Shift+向下键
* Shift+向上键
* Shift+Page Down
* Shift+Page Up
* Ctrl+Shift+向下键
* Ctrl+Shift+向上键
* CTRL + SHIFT + HOME
* CTRL + SHIFT + END

下表列出了默认鼠标行为：

| **鼠标操作** | **描述** |
| --- | --- |
| 单击未选定的行 | 将所单击的行作为当前行，单元格作为当前单元格。 |
| 单击当前单元格 | 当前单元格置于编辑模式。 |
| 拖动列标题单元格 | 如果[DataGrid.CanUserReorderColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserreordercolumns)属性是true并且[DataGridColumn.CanUserReorder](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcolumn.canuserreorder)属性是true拖动当前列到新的位置。 |
| 拖动列标题分隔符 | 如果[DataGrid.CanUserResizeColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserresizecolumns)属性是true并[DataGridColumn.CanUserResize](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcolumn.canuserresize)属性是true当前列中，调整列的大小。 |
| 双击列标题分隔符 | 如果[DataGrid.CanUserResizeColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canuserresizecolumns)属性是true并且[DataGridColumn.CanUserResize](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcolumn.canuserresize)属性是true当前列中，自动调整列大小。 |
| 单击列标题单元格 | 如果[DataGrid.CanUserSortColumns](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.canusersortcolumns)属性是true并且[DataGridColumn.CanUserSort](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridcolumn.canusersort)属性是true当前列进行排序的列。  单击已排序的列标题将反转该列的排序方向。  当按下 SHIFT 键单击多个列标题会按单击顺序进行排序。 |
| CTRL + 单击的行 | 如果[SelectionMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionmode)设置为[Extended](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridselectionmode#System_Windows_Controls_DataGridSelectionMode_Extended)，选择非连续多行。  如果行已被选中，则取消选择该行。 |
| 按住 SHIFT 的同时单击的行 | 如果[SelectionMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionmode)设置为[Extended](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridselectionmode#System_Windows_Controls_DataGridSelectionMode_Extended)，选择连续多行。 |
| 单击行组标题 | 展开或折叠组。 |
| 单击左上角的全选按钮 | 如果[SelectionMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagrid.selectionmode)设置为[Extended](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.datagridselectionmode#System_Windows_Controls_DataGridSelectionMode_Extended)，选中所有行。 |

1. 依赖属性和附加属性

前面章节中，我们已经提到过依赖属性和附加属性，尤其是附加属性这一词汇更是在多处涉及。附加属性一般也是特殊的一种依赖属性。WPF在不改变.net的基础设施Property的基础上，，推出了依赖属性的机制，依赖属性是WPF的核心机制之一，无处不在的数据绑定机制，就建立在依赖属性的基础上。除此之外，WPF的许多功能都需要依赖属性的支持。

* 1. 依赖属性概述

.net的Property虽然只是一个语法糖，实质上是get/set方法，但为提供对象的信息隐藏提供了帮助，并且使我们可以用使用字段的方式直观地使用Property，同时，还可以作为虚字段使用，还可以提供诸如数据校验、延迟计算等特性。

在图形用户界面设计中，往往要用到许许多多的控件，这些控件拥有少则数十，多则数百个属性，而这些属性在应用中需要改变和设置的往往是极少数，如果使用传统的Property，意味着大量的存储空间的浪费。为此，WPF推出了依赖属性机制，一方面大大节省了内存开销，另一方面，依赖一词意为着其值可以来自其他对象，通过数据绑定的方式，为WPF的许多机制提供了支持。

依赖属性扩展了Property的功能，代表或支持以下WPF特性：

* [资源](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#resources) 依赖属性的值可以通过引用资源来设置。若要使用动态资源引用，必须使用依赖属性
* [数据绑定](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#data-binding) 依赖属性可以通过数据绑定来引用值
* [样式](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#styles) 样式和模板是使用依赖属性的两个主要场景
* [动画](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#animations)
* [元数据重写](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#metadata-overrides)
* [属性值继承](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#property-value-inheritance)
* [WPF 设计器集成](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/dependency-properties-overview#wpf-designer-integration)

在 WPF 元素定义了许多非依赖属性的属性。一般说来，只有在需要支持至少一个由属性系统启用的方案（数据绑定、样式、动画、默认值支持、继承、附加属性或失效）时，才将属性实现为依赖属性。

* 1. 定义依赖属性

以下代码片段展示了如何定义依赖属性：

C#

Public class AquariumObject：DependencyObject {

public static readonly DependencyProperty AquariumGraphicProperty = DependencyProperty.Register(

"AquariumGraphic",

typeof(Uri),

typeof(AquariumObject),

new FrameworkPropertyMetadata(null,

FrameworkPropertyMetadataOptions.AffectsRender,

new PropertyChangedCallback(OnUriChanged)

)

);

public Uri AquariumGraphic

{

get { return (Uri)GetValue(AquariumGraphicProperty); }

set { SetValue(AquariumGraphicProperty, value); }

}

定义依赖属性有以下几个步骤：

1. 声明依赖属性字段，这里是AquariumGraphicProperty，它必须用public static readonly修饰，字段命名约定是依赖属性名+Property。
2. 创建PropertyMetadata对象，作为下一步注册依赖属性的参数。本例中直接在注册方法调用的第四个参数创建，这一步是可选的，注册时可以使用默认参数。
3. 注册依赖属性。DependencyProperty.Register方法的第一个参数是依赖属性的名称；第二个参数是依赖属性存储的值的类型；第三个参数是依赖属性宿主类的类型；第四个参数是前面创建的PropertyMetadata对象，它是可选的；还可以有第五个参数，它是用于验证的回调函数，也是可选的。这个参数一旦设定，不能修改，只能通过DependencyProperty.OverrideMetadata方法替换。
4. 生成Property包装器，如代码所示。需要注意的是，这里不要包含其他逻辑，因为在XAML解析时会越过包装器。

还可以根据需要定义只读的依赖属性：

C#

internal static readonly DependencyPropertyKey AquariumSizeKey = DependencyProperty.RegisterReadOnly(

"AquariumSize",

typeof(double),

typeof(Aquarium),

new PropertyMetadata(double.NaN)

);

public static readonly DependencyProperty AquariumSizeProperty =

AquariumSizeKey.DependencyProperty;

public double AquariumSize

{

get { return (double)GetValue(AquariumSizeProperty); }

}

和之前的主要区别是：RegisterReadOnly替代Register；返回类型是DependencyPropertyKey；字段不声明为public；Property包装器没有set。

依赖属性字段声明为static，可能会让人迷惑不解。关于这方面的内容，可以参考刘铁猛老师的《深入浅出WPF》，有较详细的讲解，也可以阅读源码，获得更深入细致的理解。

* 1. FrameworkPropertyMetadata

一般使用PropertyMetadata时用的最多的时FrameworkPropertyMetadata，多数情况我们使用其默认行为，通过设定选项标志可以得到特定的行为：

FrameworkPropertyMetadataOptions

|  |  |  |
| --- | --- | --- |
| AffectsArrange | 2 | 更改此依赖属性的值会影响布局组合的排列过程。 |
| AffectsMeasure | 1 | 更改此依赖属性的值会影响布局组合的测量过程。 |
| AffectsParentArrange | 8 | 更改此依赖属性的值会影响父元素上的排列过程。 |
| AffectsParentMeasure | 4 | 更改此依赖属性的值会影响父元素上的测量过程。 |
| AffectsRender | 16 | 更改此依赖属性的值会影响呈现或布局组合的某一方面（不是测量或排列过程）。 |
| BindsTwoWayByDefault | 256 | 此依赖属性上的数据绑定的 [BindingMode](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.bindingmode?view=netframework-4.7.2) 默认为 [TwoWay](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.bindingmode?view=netframework-4.7.2#System_Windows_Data_BindingMode_TwoWay)。 |
| Inherits | 32 | 此依赖属性的值将由子元素继承。 |
| Journal | 1024 | 此依赖属性的值应由导航历史记录或在由URI导航时进行保存或恢复。 |
| None | 0 | 未指定任何选项；依赖属性使用WPF属性系统的默认行为。 |
| NotDataBindable | 128 | 不允许将数据绑定到此依赖属性。 |
| OverridesInheritanceBehavior | 64 | 此依赖属性的值跨越分隔的树以实现属性值继承。 |
| SubPropertiesDoNotAffectRender | 2048 | 此依赖属性值上的子属性不会影响呈现的任何方面。 |

FrameworkPropertyMetadata最完整的构造函数如下：

C#

public FrameworkPropertyMetadata (object defaultValue, System.Windows.FrameworkPropertyMetadataOptions flags, System.Windows.PropertyChangedCallback propertyChangedCallback, System.Windows.CoerceValueCallback coerceValueCallback, bool isAnimationProhibited, System.Windows.Data.UpdateSourceTrigger defaultUpdateSourceTrigger);

示例：

C#

static FrameworkPropertyMetadata fpm =

new FrameworkPropertyMetadata(

Double.NaN,

(FrameworkPropertyMetadataOptions.AffectsRender |

FrameworkPropertyMetadataOptions.BindsTwoWayByDefault),

new PropertyChangedCallback(OnCurrentReadingChanged),

new CoerceValueCallback(CoerceCurrentReading),

true //Animation prohibited

, UpdateSourceTrigger.PropertyChanged

);

* 1. 依赖属性回调和验证

首次注册依赖属性时，可以设定数据校验回调方法（第五个参数）。

C#

public static readonly DependencyProperty CurrentReadingProperty = DependencyProperty.Register(

"CurrentReading",

typeof(double),

typeof(Gauge),

new FrameworkPropertyMetadata(

Double.NaN,

FrameworkPropertyMetadataOptions.AffectsMeasure,

new PropertyChangedCallback(OnCurrentReadingChanged),

new CoerceValueCallback(CoerceCurrentReading)

),

new ValidateValueCallback(IsValidReading)

);

public double CurrentReading

{

get { return (double)GetValue(CurrentReadingProperty); }

set { SetValue(CurrentReadingProperty, value); }

}

public static bool IsValidReading(object value)

{

Double v = (Double)value;

return (!v.Equals(Double.NegativeInfinity) && !v.Equals(Double.PositiveInfinity));

}

private static void OnCurrentReadingChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)

{

d.CoerceValue(MinReadingProperty);

d.CoerceValue(MaxReadingProperty);

}

private static object CoerceCurrentReading(DependencyObject d, object value)

{

Gauge g = (Gauge)d;

double current = (double)value;

if (current < g.MinReading) current = g.MinReading;

if (current > g.MaxReading) current = g.MaxReading;

return current;

}

上例中验证依赖属性的值是否合法的double值。

注册时使用第三个参数PropertyMetadata，创建时允许指定强制值回调方法和属性变更回调方法。

强制值回调方法检查当前的属性值是否合理，不合理则返回强制的合理值。属性的实际当前值会保存，除非调用ClearValue方法清除。在应用程序生存期，如果约束依赖的值发生变化，强制行为也会动态变化。

系统会首先调用强制值回调，然后调用验证回调，上述两个成功完成后，出发属性值变更回调。

* 1. 依赖属性值的优先级

依赖属性的值可以来自本地设定的值，也可以来自默认值，或者绑定的其他来源的值，也可以设定为从父元素继承的值。它们的优先级顺序如下：

1. 属性系统强制**。**
2. 活动动画或具有Hold行为的动画。
3. 属性的本地值。
4. TemplatedParent 模板属性。在模板中，按以下优先级顺序应用：
5. 从[TemplatedParent](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.templatedparent)模板触发。
6. (通常通过XAML attributes)在[TemplatedParent](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.templatedparent)模板中的属性集。
7. 隐式样式，仅用于Style属性。
8. 样式触发器。
9. 模板触发器。
10. 样式资源库Setter。
11. 默认（主题）样式。在默认样式中，按以下优先级顺序应用：
12. 主题样式中的活动触发器。
13. 主题样式中的资源库。
14. 从父元素继承
15. 来自依赖属性元数据的默认值。
    1. 附加属性

附加属性是XAML定义的概念，旨在用作可在任一对象上设置的一类全局属性。在WPF中，附加属性常被定义为没有常规Property包装器的依赖属性。

附加属性的一个主要用途是允许子元素为父元素中定义的属性指定唯一值。

**定义附加属性**：

C#

public static readonly DependencyProperty IsBubbleSourceProperty = DependencyProperty.RegisterAttached(

"IsBubbleSource",

typeof(Boolean),

typeof(AquariumObject),

new FrameworkPropertyMetadata(false, FrameworkPropertyMetadataOptions.AffectsRender)

);

public static void SetIsBubbleSource(UIElement element, Boolean value)

{

element.SetValue(IsBubbleSourceProperty, value);

}

public static Boolean GetIsBubbleSource(UIElement element)

{

return (Boolean)element.GetValue(IsBubbleSourceProperty);

}

定义附加属性注册时调用RegisterAttached而不是Register方法，包装器也和定义依赖属性不同，不是使用常规的Property，而是显式的GetXXX、SetXXX静态访问器。

1. 事件、路由事件和命令
   1. 事件

.net为我们提供了事件机制，它是一个发布-订阅模式。事件是委托的封装，事件的订阅者可以把事件的处理方法注册到事件发布者中，事件发布者收到相应的Windows消息，就会触发事件，调用订阅者注册的委托方法链。如我们的在WinForm中常见到的：

btn1.Click+=sub1.OnClick;

但在WPF中，仅仅依靠基本的事件模型是不够的。WPF的内容模型为我们提供了灵活的控件组合方式，可以帮助我们方便地合成控件或自定义控件，这时候，可视化树相对逻辑树展示了更多的细节，对控件子元素的点击在逻辑上可能是对父元素的点击，比如：一个按钮可以嵌入图片，点击按钮的图片显然应该解释为对按钮的点击而不是对图片的点击，这时候，利用基本的事件模型处理起来就很繁琐；有时候，多个子元素触发的事件我们需要的是同样的处理方法，采用基本事件处理方式，就需要把事件处理方法附加到每一个事件源上，处理起来也颇为繁琐。

为此，WPF引入了新的事件处理模型-路由事件。路由事件允许源自某个元素的事件由另一个元素触发，从而使得可以在最合适的位置进行事件处理，编写出更灵活，更紧凑的代码。

* 1. 路由事件

可以从功能和实现的角度理解路由事件。

**功能定义**：路由事件是一种不只在引发事件的对象上，而且可以在元素树的多个侦听者上调用事件处理程序的事件。

**实现定义**：路由事件是依靠RoutedEvent的实例支持，由WPF事件系统处理的CLR事件。

路由事件的传播方式有以下三种：

* 冒泡（Bubble）：先调用事件源的处理程序，接着路由事件沿着可视化树向上传播，大多数路由事件采用冒泡传播策略。冒泡传播通常用于报告来自不同子UI元素的输入或状态变化。
* 直接（Direct）：仅源元素本身有机会响应，调用事件处理程序。
* 隧道（Tunneling）：首先调用元素树根的事件处理程序，然后朝着源节点向下传播到后续子元素。隧道路由事件通常作为组合控件的一部分使用或处理，这样，复合部件中的事件可以有意被终止或被替换为整个控件的事件。隧道事件又称为预览事件，WPF的输入事件以通常隧道/冒泡对实现。

通常，如果为某个事件定义了隧道事件，就会有对应的冒泡事件。在这种情况下，隧道事件先被触发，从根元素开始下行至事件源，查找事件处理程序。一旦它被处理或到达事件源，就会触发冒泡事件，从事件源上行，查找处理程序。

WPF的事件系统，采用路由事件取代标准的CLR事件，事件的监听器不需要和事件源在层次结构中共享事件，任何一个UIElement或ContentElement元素都可以作为任意路由事件的侦听器。

路由事件除了上一节我们提到的好处，还可以用于通过元素树通信。除此之外，要设计自己的事件，还要考虑：

* 某些WPF样式和模板化功能，如EventSetter和EventTrigger需要被引用的事件是路由的事件。
* 路由事件支持类处理机制，类可以通过该机制来指定静态方法，这些静态方法能够在任何已注册的实例处理程序访问路由事件之前，处理这些路由事件。

**使用路由事件：**

XAML

<Button Click="b1SetColor">button</Button>

C#

void b1SetColor(object sender, RoutedEventArgs args)

{

//logic to handle the Click event

}

事件名称（Click）作为元素（Button）的Attribute，其值（b1SetColor）是分布类中具有相同签名的方法（RoutedEventHandler委托）。

以下是纯代码形式示例：

C#

void MakeButton2（）

{

Button b2 = new Button();

b2.Click += new RoutedEventHandler(Onb2Click2);

}

void Onb2Click2(object sender, RoutedEventArgs e)

{

//logic to handle the Click event

}

RoutedEventHandler委托的签名第一个参数sender是事件处理程序附加到的对象，sender不一定是事件源，第二个参数RoutedEventArgs包含事件数据，它有四个Properties：

|  |  |
| --- | --- |
| [Handled](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.handled?view=netframework-4.7.2#System_Windows_RoutedEventArgs_Handled) | 允许终止事件的冒泡或隧道过程。如果设置为true，事件就不再传播，并且也不会再为任何元素引发该事件（这其实并不十分准确，见下文**处理挂起事件**） |
| [OriginalSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.originalsource?view=netframework-4.7.2#System_Windows_RoutedEventArgs_OriginalSource) | 在父元素进行任何可能的 [Source](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.source?view=netframework-4.7.2#System_Windows_RoutedEventArgs_Source) 调整之前，真正的最原始事件源。 |
| [RoutedEvent](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.routedevent?view=netframework-4.7.2#System_Windows_RoutedEventArgs_RoutedEvent) | 获取或设置与此 [RoutedEventArgs](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs?view=netframework-4.7.2) 实例关联的 [RoutedEvent](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.routedevent?view=netframework-4.7.2#System_Windows_RoutedEventArgs_RoutedEvent)。 |
| [Source](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.routedeventargs.source?view=netframework-4.7.2#System_Windows_RoutedEventArgs_Source) | 获取或设置对引发事件的对象的引用。 |

**处理挂起事件**

private void OnChildElementMouseDown(object sender,

MouseButtonEventArgs e) {

e.Handled = true;

}

一旦您的处理程序将事件标记为已处理，该事件便不会传给任何其他处理程序。但实际上事件路由仍在起作用，有一种handledEventsToo机制，监听器认可运行事件处理程序相应已标记Handled=true的路由事件：

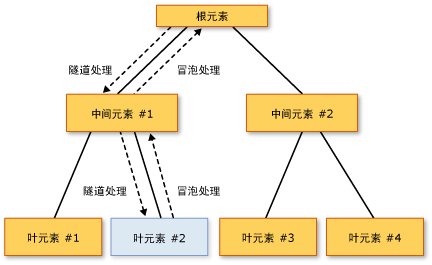
* 在代码中，调用AddHandler (System.Windows.RoutedEvent routedEvent, Delegate handler, bool handledEventsToo)，handledEventsToo参数设置为true。m\_SomeChildElement.AddHandler(UIElement.MouseDownEvent, (RoutedEventHandler)OnMouseDownCallMeAlways,true);
* 在XAML的EventSetter，将handledEventsToo设置为true。

路由事件会考虑两种监听器：类监听器和实例监听器。如果存在类监听器，会先调用类监听器的事件处理程序。

**WPF输入事件**

输入事件是WPF中最常见的路由事件。输入事件经常是隧道事件和冒泡事件成对出现，按约定，隧道事件的名称以Preview打头，例如KeyDown和PreviewKeyDown事件具有相同签名，前者是冒泡事件，后者是隧道事件。如之前所述，来自用户的输入事件，首先引发隧道事件，然后引发冒泡事件。如果隧道路由过程中某个元素设置了Handled=true，已处理的事件数据会发送到冒泡事件，冒泡事件的处理程序将不被调用。

下图中，叶元素#2是PreviewMouseDown和MouseDown的事件源：



事件的处理顺序如下所述：

1. 针对根元素处理 PreviewMouseDown（隧道）。
2. 针对中间元素 #1 处理 PreviewMouseDown（隧道）。
3. 针对源元素 #2 处理 PreviewMouseDown（隧道）。
4. 针对源元素 #2 处理 MouseDown（冒泡）。
5. 针对中间元素 #1 处理 MouseDown（冒泡）。
6. 针对根元素处理 MouseDown（冒泡）。
   1. 自定义路由事件

|  |
| --- |
| public class MyButtonSimple: Button  {  // Create a custom routed event by first registering a RoutedEventID  // This event uses the bubbling routing strategy  public static readonly RoutedEvent TapEvent = EventManager.RegisterRoutedEvent(  "Tap", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(MyButtonSimple));  // Provide CLR accessors for the event  public event RoutedEventHandler Tap  {  add { AddHandler(TapEvent, value); }  remove { RemoveHandler(TapEvent, value); }  }  // This method raises the Tap event  void RaiseTapEvent()  {  RoutedEventArgs newEventArgs = new RoutedEventArgs(MyButtonSimple.TapEvent);  RaiseEvent(newEventArgs);  }  // For demonstration purposes we raise the event when the MyButtonSimple is clicked  protected override void OnClick()  {  RaiseTapEvent();  }  } |

和定义依赖属性有些类似，首先声明一个public static readonly RoutedEvent对象，对象约定名称为路由事件名称+Event（隧道事件要使用Preview前缀），然后用EventManager.RegisterRoutedEvent方法注册，接下来对路由事件做CLR事件封装，最后用RaiseEvent触发事件。

使用自定义路由事件：

XAML

< Window

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

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

xmlns:custom="clr-namespace:SDKSample;assembly=SDKSampleLibrary"

x:Class="SDKSample.RoutedEventCustomApp"

>

<Window.Resources>

<Style TargetType="{x:Type custom:MyButtonSimple}">

<Setter Property="Height" Value="20"/>

<Setter Property="Width" Value="250"/>

<Setter Property="HorizontalAlignment" Value="Left"/>

<Setter Property="Background" Value="#808080"/>

</Style>

</Window.Resources>

<StackPanel Background="LightGray">

<custom:MyButtonSimple Name="mybtnsimple" Tap="TapHandler">Click to see Tap custom event work</custom:MyButtonSimple>

</StackPanel>

</Window>

* 1. 附加事件

XAML定义了称为附加事件的一类事件，通过附加事件的概念，你能够向任意元素（而不是实际定义或继承事件的元素）添加特定事件的处理程序。附加事件也是路由事件。例如，如果您想要 Grid 侦听采用气泡方式通过的 Button.Click 事件，仅需按如下所示进行挂接即可。

<Grid Button.Click="myButton\_Click">

<Button Name="myButton" >Click Me</Button>

</Grid>

在编译时生成的局部类中的最终代码现在如下所示：

((System.Windows.Controls.Grid)(target)).AddHandler(

System.Windows.Controls.Primitives.ButtonBase.ClickEvent,

new System.Windows.RoutedEventHandler(this.myButton\_Click));

一般情况下，WPF 附加事件与 WPF 路由事件并没有太大的区别。 不同之处在于如何确定事件的源，以及如何通过类将事件作为成员进行公开。

通常，附加事件的源（引发该事件的对象）是系统或服务源，所以运行引发该事件的代码的对象并不是元素树的直接组成部分。

尽管 WPF 定义了许多附加事件，但直接使用或处理附加事件的情形却很少。 一般情况下，附加事件用于体系结构，但随后即被转发给非附加（使用 CLR 事件“包装器”提供支持）路由事件。

定义自己的附加事件可参考：

C#

public static readonly RoutedEvent NeedsCleaningEvent = EventManager.RegisterRoutedEvent("NeedsCleaning", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(AquariumFilter));

public static void AddNeedsCleaningHandler(DependencyObject d, RoutedEventHandler handler)

{

UIElement uie = d as UIElement;

if (uie != null)

{

uie.AddHandler(AquariumFilter.NeedsCleaningEvent, handler);

}

}

public static void RemoveNeedsCleaningHandler(DependencyObject d, RoutedEventHandler handler)

{

UIElement uie = d as UIElement;

if (uie != null)

{

uie.RemoveHandler(AquariumFilter.NeedsCleaningEvent, handler);

}

}

* 1. 命令

WPF命令是WPF在语义级别提供输入处理的一种机制。乍一看，路由事件已经可以解决问题。但命令和事件针对的场景和抽象层次还是不一样的。事件是一种订阅发布模式，可以支持发布者和订阅者一对多的关系；而命令常有的场景之一是多个命令的调用者对应一个命令执行逻辑，比如你可以从菜单、上下文菜单、工具栏、快捷键等调用“Save”命令。在多层应用架构中，事件主要是应用在表示层（UI），在MVC及其衍生UI架构，主要表现在View和Controller；而命令是更高级的抽象，通常需要完成某种或某一组业务逻辑，执行的命令逻辑很可能是业务层的而不限于表示层。WPF的命令还可以指示操作是否可用，从而禁用控件来管理用户界面，比如，以剪切对象或文本为例，只有选择了内容才会发生作用。

命令是GoF软件设计模式中行为模式的重要一种，它使用命令接口，把命令的调用者和命令的执行逻辑解耦，命令调用者并不需要知道命令的执行逻辑，从而使程序有更少的耦合和更好的扩展性。

WPF的命令模型可分解为四个重要概念：

* ***命令***：要执行的操作
* ***命令源***：调用命令的对象
* ***命令目标***：在其上执行命令的对象
* ***命令绑定***：把命令逻辑映射到命令的对象

下面示例展示如何使用命令：

|  |
| --- |
| XAML：  <Window x:Class="WCSamples.Window1"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  Title="CloseCommand"  Name="RootWindow"  >  <Window.CommandBindings>  <CommandBinding Command="ApplicationCommands.Close"  Executed="CloseCommandHandler"  CanExecute="CanExecuteHandler"  />  </Window.CommandBindings>  <StackPanel Name="MainStackPanel">  <Button Command="ApplicationCommands.Close"  Content="Close File" />  </StackPanel>  </Window> |
| C#：  // Create ui elements.  StackPanel CloseCmdStackPanel = new StackPanel();  Button CloseCmdButton = new Button();  CloseCmdStackPanel.Children.Add(CloseCmdButton);  // Set Button's properties.  CloseCmdButton.Content = "Close File";  CloseCmdButton.Command = ApplicationCommands.Close;  // Create the CommandBinding.  CommandBinding CloseCommandBinding = new CommandBinding(  ApplicationCommands.Close, CloseCommandHandler, CanExecuteHandler);  // Add the CommandBinding to the root Window.  RootWindow.CommandBindings.Add(CloseCommandBinding); |

示例中，命令源是按钮（CloseCmdButton），命令是ApplicationCommands. Close，通过命令绑定把具体的命令处理过程和命令关联起来。

|  |
| --- |
| // Executed event handler.  private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)  {  // Calls a method to close the file and release resources.  CloseFile();  }  // CanExecute event handler.  private void CanExecuteHandler(object sender, CanExecuteRoutedEventArgs e)  {  // Call a method to determine if there is a file open.  // If there is a file open, then set CanExecute to true.  if (IsFileOpened())  {  e.CanExecute = true;  }  // if there is not a file open, then set CanExecute to false.  else  {  e.CanExecute = false;  }  } |

**命令**是ICommand的实现，Icommand公开了两个方法：Execute和CanExecute以及一个事件：CanExecuteChanged。Execute执行与该命令相关的操作，CanExecute确定是否可以在当前命令目标上执行该命令，CanExecuteChanged事件通知命令源当前命令目标上执行命令的能力可能已改变。

ICommand的WPF实现是路由命令（RoutedCommand），路由命令的的 Execute 和 CanExecute 方法不包含该命令的应用程序逻辑，而是引发通过元素树的隧道路由事件和冒泡路由事件，直到遇到具有 [CommandBinding](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding) 的对象。所以，这里命令绑定的位置很重要。

RoutedCommand 上的 Execute 方法引发命令目标上的 PreviewExecuted 和 Executed 事件。 RoutedCommand 上的 CanExecute 方法引发命令目标上的 CanExecute 和 PreviewCanExecute 事件。

WPF 提供了分布在几个类中的一组常用路由命令：[MediaCommands](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.mediacommands)、[ApplicationCommands](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.applicationcommands)、[NavigationCommands](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.navigationcommands)、[ComponentCommands](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.componentcommands) 和 [EditingCommands](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.documents.editingcommands)，提供诸如 [Cut](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.applicationcommands.cut)、[BrowseBack](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.navigationcommands.browseback)、[BrowseForward](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.navigationcommands.browseforward)、[Play](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.mediacommands.play)、[Stop](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.mediacommands.stop) 和 [Pause](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.mediacommands.pause) 的命令。 这些类仅由 [RoutedCommand](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.routedcommand) 对象构成，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命令源**的例子有MenuItem、Button 和 InputGesture（KeyGesture、MouseGesture）。WPF命令源实现了IcommandSource接口，[ICommandSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.icommandsource) 公开三个属性：[Command](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.icommandsource.command)、[CommandTarget](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.icommandsource.commandtarget) 和 [CommandParameter](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.icommandsource.commandparameter)，[CommandParameter](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.icommandsource.commandparameter) 是用于将信息传递给实现命令的处理程序的用户定义数据类型。

实现 ICommandSource 的 WPF 类是 ButtonBase、MenuItem、Hyperlink 和 InputBinding。

命令源将侦听 CanExecuteChanged 事件。命令源可以使用 [CanExecute](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.routedcommand.canexecute) 方法查询 [RoutedCommand](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.routedcommand) 的当前状态。 如果命令无法执行，命令源可禁用自身。如菜单项命令无法执行时将灰显。

[CommandBinding](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding) 类包含 [Command](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding.command) 属性，及 [PreviewExecuted](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding.previewexecuted)、[Executed](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding.executed)、[PreviewCanExecute](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding.previewcanexecute) 和 [CanExecute](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding.canexecute) 事件。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是否可以在当前命令目标上执行该命令。

上例中具体执行Close命令的处理程序是CloseCommandHandler方法，CanExecuteHandler通过测试文件是否打开，设置CanExecute为true或false。

[CommandBinding](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.input.commandbinding)可以附加到命令目标本身，但更多是附加在命令目标的上级元素。

命令目标可以在命令源显示设置，如果未设置，具有键盘焦点的元素将用作命令目标（如果需要的话）。示例如下：

|  |
| --- |
| <StackPanel>  <Menu>  <MenuItem Command="ApplicationCommands.Paste"  CommandTarget="{Binding ElementName=mainTextBox}" />  </Menu>  <TextBox Name="mainTextBox"/>  </StackPanel> |

对应的C#代码：

|  |
| --- |
| // Creating the UI objects  StackPanel mainStackPanel = new StackPanel();  TextBox pasteTextBox = new TextBox();  Menu stackPanelMenu = new Menu();  MenuItem pasteMenuItem = new MenuItem();    // Adding objects to the panel and the menu  stackPanelMenu.Items.Add(pasteMenuItem);  mainStackPanel.Children.Add(stackPanelMenu);  mainStackPanel.Children.Add(pasteTextBox);  // Setting the command to the Paste command  pasteMenuItem.Command = ApplicationCommands.Paste;  // Setting the command target to the TextBox  pasteMenuItem.CommandTarget = pasteTextBox; |

* 1. 创建自己的命令源

下面的示例展示了如何通过实现ICommandSource接口创建自定义的事件源，本例来自MSDN的示例：[实现 ICommandSource](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/how-to-implement-icommandsource)，创建了Slider的子类并实现了IcommandSource接口。

* 定义类

|  |
| --- |
| public class CommandSlider : Slider, ICommandSource  {  public CommandSlider() : base()  {  } |

* 实现ICommandSource成员（Command、CommandTarget和CommandParameter）。为使用数据绑定，把它们定义为DependencyProperty对象。这里仅显示Command Property：

|  |
| --- |
| // Make Command a dependency property so it can use databinding.  public static readonly DependencyProperty CommandProperty =  DependencyProperty.Register(  "Command",  typeof(ICommand),  typeof(CommandSlider),  new PropertyMetadata((ICommand)null,  new PropertyChangedCallback(CommandChanged)));  public ICommand Command  {  get  {  return (ICommand)GetValue(CommandProperty);  }  set  {  SetValue(CommandProperty, value);  }  } |

* PropertyChangedCallback

|  |
| --- |
| // Command dependency property change callback.  private static void CommandChanged(DependencyObject d,  DependencyPropertyChangedEventArgs e)  {  CommandSlider cs = (CommandSlider)d;  cs.HookUpCommand((ICommand)e.OldValue,(ICommand)e.NewValue);  } |

* 添加和删除命令源相关的命令

|  |
| --- |
| // Add a new command to the Command Property.  private void HookUpCommand(ICommand oldCommand, ICommand newCommand)  {  // If oldCommand is not null, then we need to remove the handlers.  if (oldCommand != null)  {  RemoveCommand(oldCommand, newCommand);  }  AddCommand(oldCommand, newCommand);  }  // Remove an old command from the Command Property.  private void RemoveCommand(ICommand oldCommand, ICommand newCommand)  {  EventHandler handler = CanExecuteChanged;  oldCommand.CanExecuteChanged -= handler;  }  // Add the command.  private void AddCommand(ICommand oldCommand, ICommand newCommand)  {  EventHandler handler = new EventHandler(CanExecuteChanged);  canExecuteChangedHandler = handler;  if (newCommand != null)  {  ewCommand.CanExecuteChanged += canExecuteChangedHandler;  }  } |

* CanExecuteChanged处理程序

|  |
| --- |
| private void CanExecuteChanged(object sender, EventArgs e)  {  if (this.Command != null)  {  RoutedCommand command = this.Command as RoutedCommand;  // If a RoutedCommand.  if (command != null)  {  if (command.CanExecute(CommandParameter, CommandTarget))  {  this.IsEnabled = true;  }  else  {  this.IsEnabled = false;  }  }  // If a not RoutedCommand.  else  {  if (Command.CanExecute(CommandParameter))  {  this.IsEnabled = true;  }  else  {  this.IsEnabled = false;  }  }  }  } |

* 触发命令，执行命令的Execute方法

|  |
| --- |
| // If Command is defined, moving the slider will invoke the command;  // Otherwise, the slider will behave normally.  protected override void OnValueChanged(double oldValue, double newValue)  {  base.OnValueChanged(oldValue, newValue);  if (this.Command != null)  {  RoutedCommand command = Command as RoutedCommand;  if (command != null)  {  command.Execute(CommandParameter, CommandTarget);  }  else  {  ((ICommand)Command).Execute(CommandParameter);  }  }  } |

这样，自定义的slider事件源就可以像Button之类的事件源一样使用。

* 1. 自定义命令

WPF的路由命令适合用于图形界面，但如果命令与可视化树无关，或者需要执行一组命令，或者提供命令的离实以及Undo、Redo操作，路由命令就有很大局限性，这时候就需要定义自己的命令。

自定义命令其实很简单，利用.net的委托机制，可以很容易地按照GoF的命令模式定义自己的路由，最主要的是要实现ICommand的CanExecute和Execute方法。

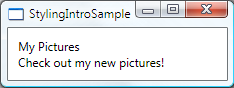
1. 样式、触发器和行为

在我们设计用户界面时，程序的功能只是其中的一部分，还有很重要的一部分可以归于美学的范畴。常言道，人靠衣裳马靠鞍，良好的界面设计是一个应用程序成功不可或缺的要素。在界面设计时，我们往往需要设置控件的颜色、字体、形状、动画等外观表现，设计出与众不同的独特界面，往往需要在不同页面之间保持统一的风格，同一类控件具有类似的表现行为，但不同地方可能要有一些独特的表现行为，而完成这些工作，如果针对每一个控件分别设置其属性等，很明显及意味者繁重的工作，也难以保持一致的风格，修改起来也非常困难。幸运的是，WPF借鉴Web页面设计的样式技术，提供了样式机制，为我们提供了丰富多彩而又方便使用的功能。

* 1. 样式

考虑以下TextBlock元素和它们的默认外观：

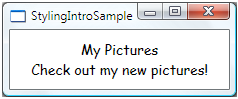
|  |
| --- |
| <TextBlock>My Pictures</TextBlock>  <TextBlock>Check out my new pictures!</TextBlock> |



可以通过设置属性，如设置字体等改变默认外观，但如果所有TextBlock要保持一致的外观，通过在资源中定义Style是可行的方法。

|  |
| --- |
| <Window.Resources>  <!--A Style that affects all TextBlocks-->  <Style TargetType="TextBlock">  <Setter Property="HorizontalAlignment" Value="Center" />  <Setter Property="FontFamily" Value="Comic Sans MS"/>  <Setter Property="FontSize" Value="14"/>  </Style>  </Window.Resources> |

通过把样式的目标TargetType设置为TextBlock，所有的TextBlock都会呈现一致的外观：



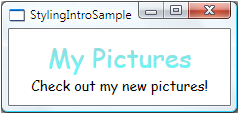
有时候还会希望某些TextBlock再次基础上有一些别的属性，可以在此样式的基础上对其扩展。

|  |
| --- |
| <Window.Resources>  <!--A Style that extends the previous TextBlock Style-->  <!--This is a "named style" with an x:Key of TitleText-->  <Style BasedOn="{StaticResource {x:Type TextBlock}}"  TargetType="TextBlock"  x:Key="TitleText">  <Setter Property="FontSize" Value="26"/>  <Setter Property="Foreground">  <Setter.Value>  <LinearGradientBrush StartPoint="0.5,0" EndPoint="0.5,1">  <LinearGradientBrush.GradientStops>  <GradientStop Offset="0.0" Color="#90DDDD" />  <GradientStop Offset="1.0" Color="#5BFFFF" />  </LinearGradientBrush.GradientStops>  </LinearGradientBrush>  </Setter.Value>  </Setter>  </Style>  </Window.Resources> |

上面的样式扩展使用了x:key

|  |
| --- |
| <TextBlock Style="{StaticResource TitleText}" Name="textblock1">My Pictures</TextBlock>  <TextBlock>Check out my new pictures!</TextBlock> |

效果：



使用这个扩展的样式时，使用静态资源引用设值Style，这里用到了前面定义的x:key。

如果在Style的Setter中多次设置同一个属性，则最后一个优先。

在设置样式时，如果只设置了TargetType而没有设置x:key，则样式应用于所有的TargetType值对应的元素（TextBlock），这里，x:key隐式设置为 {x:Type TextBlock}。如果显示设定了x:key，则该Style不自动应用于所有TargetType指向的元素，要使用该样式，必须在元素上显示地引用x:key设置Style。

用代码设置Style：

textblock1.Style = (Style)(this.Resources["TitleText"]);

还可以用EventSetter在样式中指定事件处理程序。示例如下：

|  |
| --- |
| <StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.EventOvw2"  Name="dpanel2"  Initialized="PrimeHandledToo"  >  <StackPanel.Resources>  <Style TargetType="{x:Type Button}">  <EventSetter Event="Click" Handler="b1SetColor"/>  </Style>  </StackPanel.Resources>  <Button>Click me</Button>  <Button Name="ThisButton" Click="HandleThis">  Raise event, handle it, use handled=true handler to get it anyway.  </Button>  </StackPanel> |

对应的事件处理程序：

|  |
| --- |
| void b1SetColor(object sender, RoutedEventArgs e)  {  Button b = e.Source as Button;  b.Background = new SolidColorBrush(Colors.Azure);  }  void HandleThis(object sender, RoutedEventArgs e)  {  e.Handled=true;  } |

仅Style.Setters支持EventSetter对象。

* 1. 触发器（Triggers）

触发器在属性（Property）值发生改变或触发事件时，设置属性或启动操作。除了Style，还可以在模板中使用Triggers。

属性触发器：设置属性或启动基于属性值的操作的触发器，

下面的示例把ListBox的每一项ListBoxItem设置成半透明，如果被选中，则设置为全透明。

|  |
| --- |
| <Style TargetType="ListBoxItem">  <Setter Property="Opacity" Value="0.5" />  <Setter Property="MaxHeight" Value="75" />  <Style.Triggers>  <Trigger Property="IsSelected" Value="True">  <Setter Property="Opacity" Value="1.0" />  </Trigger>  </Style.Triggers>  </Style> |

另一种触发器时**EventTrigger**，基于事件的发生启动一组动作。

|  |
| --- |
| <EventTrigger RoutedEvent="Mouse.MouseEnter">  <EventTrigger.Actions>  <BeginStoryboard>  <Storyboard>  <DoubleAnimation  Duration="0:0:0.2"  Storyboard.TargetProperty="MaxHeight"  To="90" />  </Storyboard>  </BeginStoryboard>  </EventTrigger.Actions>  </EventTrigger>  <EventTrigger RoutedEvent="Mouse.MouseLeave">  <EventTrigger.Actions>  <BeginStoryboard>  <Storyboard>  <DoubleAnimation  Duration="0:0:1"  Storyboard.TargetProperty="MaxHeight" />  </Storyboard>  </BeginStoryboard>  </EventTrigger.Actions>  </EventTrigger> |

详细信息，参考MSDN[情节提要（Storyboard）概述](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/graphics-multimedia/storyboards-overview)。

MultiTrigger：当一组条件满足时，设置属性值或触发操作。

|  |
| --- |
| <Style.Triggers>  <Trigger Property="IsEnabled" Value="false">  <Setter Property="Background" Value="#EEEEEE" />  </Trigger>  <MultiTrigger>  <MultiTrigger.Conditions>  <Condition Property="HasItems" Value="false" />  <Condition Property="Width" Value="Auto" />  </MultiTrigger.Conditions>  <Setter Property="MinWidth" Value="120"/>  </MultiTrigger>  <MultiTrigger>  <MultiTrigger.Conditions>  <Condition Property="HasItems" Value="false" />  <Condition Property="Height" Value="Auto" />  </MultiTrigger.Conditions>  <Setter Property="MinHeight" Value="95"/>  </MultiTrigger>  </Style.Triggers> |

* 1. 行为（behavior）

样式给我们提供了统一设置控件属性的能力，但我们在界面设计时为追求效果，还常常面临如拖放、缩放、停靠等等与程序功能无关，但又涉及较高的复杂度和大量编码的问题，这类代码往往还是通用的。为解决此类问题，XAML的设计工具Blend推出了被称为“行为”（Behavior)的机制。

行为不是WPF基础部分，它时Blend的一个扩展。Behavior可以赋予控件新的行为能力，它通过附加属性为控件注入一些新的功能。要使用Behavior，需要引入Interactions库。例如要使用MouseDragElementBehavior，需要以下几步步骤：

1. 添加Interactions库的引用。主要添加如下两个DLL：Microsoft.Expression.Interactions.dll和System.Windows.Interactivity.dll。
2. 添加如下名字空间

xmlns:i=<http://schemas.microsoft.com/expression/2010/interactivity>

xmlns:ei="http://schemas.microsoft.com/expression/2010/interactions"

1. 在控件中添加MouseDragElementBehavior

|  |
| --- |
| <Image Source="2.jpg" >  <i:Interaction.Behaviors>  <ei:MouseDragElementBehavior/>  </i:Interaction.Behaviors>  </Image> |

一般来说，我们应该使用别人创建好的Behavior，而不需要自己重新创建。

创建自己的行为，要继承自Behavior<T>，覆盖OnAttached方法和OnDetaching方法。通过AssociatedObject属性访问放置行为的元素，并关联事件处理程序。OnDetaching移除事件处理程序。

下面示例展示了自定义在Canvas上拖动元素的行为：

|  |
| --- |
| public class DragInCanvasBehavior : Behavior<UIElement>  {  //元素父节点  private Canvas canvas;  //标识是否进入拖动  private bool isDraging = false;  //按下鼠标时的坐标(用于计算要移动的位置)  private Point mouseOffset;  /// <summary>  /// 附加行为后  /// </summary>  protected override void OnAttached()  {  base.OnAttached();  //添加鼠标事件(AssociatedObject也就是当前应用此Behavior的元素)  this.AssociatedObject.MouseLeftButtonDown+= AssociatedObject\_MouseLeftButtonDown;  this.AssociatedObject.MouseMove+=AssociatedObject\_MouseMove;  this.AssociatedObject.MouseLeftButtonUp+= AssociatedObject\_MouseLeftButtonUp;  }    /// <summary>  /// 分离行为  /// </summary>  protected override void OnDetaching()  {  base. OnDetaching();  //移除鼠标事件  this.AssociatedObject.MouseLeftButtonDown-= AssociatedObject\_MouseLeftButtonDown;  this.AssociatedObject.MouseMove-= AssociatedObject\_MouseMove;  this.AssociatedObject.MouseLeftButtonUp-= AssociatedObject\_MouseLeftButtonUp;  }  private void AssociatedObject\_MouseLeftButtonUp(object sender, System.Windows.Input.MouseButtonEventArgs e)  {  //释放拖动状态  isDraging = false;  }  private void AssociatedObject\_MouseMove(object sender, System.Windows.Input.MouseEventArgs e)  {  //如果进入拖动状态  if (isDraging)  {  //得到新的位置  Point newPoint = e.GetPosition(canvas);  //旧的坐标加上新坐标和旧坐标的差  mouseOffset.X += newPoint.X - mouseOffset.X;  mouseOffset.Y += newPoint.Y - mouseOffset.Y;  //设置元素的Left和Top，之所以要用X(Y)减去Width(Height)，主要是为了使鼠标在元素中心  Canvas.SetLeft(this.AssociatedObject, mouseOffset.X-(this.AssociatedObject as FrameworkElement).ActualWidth/2);  Canvas.SetTop(this.AssociatedObject, mouseOffset.Y - (this.AssociatedObject as FrameworkElement).ActualHeight/2);  }  }  private void AssociatedObject\_MouseLeftButtonDown(object sender, System.Windows.Input.MouseButtonEventArgs e)  {  //将元素的父节点元素赋值为Canvas(之所以使用Canvas，是因为Canvas容易动态布局)  if (canvas == null)  canvas= (Canvas)VisualTreeHelper.GetParent(this.AssociatedObject);  //进入拖动状态  isDraging = true;  //获得初始位置  mouseOffset = e.GetPosition(this.AssociatedObject);  this.AssociatedObject.CaptureMouse();  }  } |

使用DragInCanvasBehavior：

|  |
| --- |
| <Window x:Class="WpfApplication2.MainWindow"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:i="clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"  xmlns:behavior="clr-namespace:CustomBehaviorsLibrary;assembly=CustomBehaviorsLibrary"  Title="MainWindow" Height="350" Width="525">  <Canvas Background="LightBlue">  <Rectangle Height="50" Width="50" Fill="Green" >  <i:Interaction.Behaviors>  <behavior:DragInCanvasBehavior></behavior:DragInCanvasBehavior>  </i:Interaction.Behaviors>  </Rectangle>  </Canvas>  </Window> |

这样，就可以用鼠标拖动矩形。

1. 数据绑定和MVVM
   1. MVVM模式

衡量一个软件是否成功，第一个标准是可用性，它能否解决我们面临的问题；第二个标准就是适应变化的能力。有许多评价软件的指标，比如可用性、可维护性、灵活性、可扩展性等等。要实现这些目标，一个很重要的也可以说是本质的原则是模块化，软件有高内聚低耦合的模块组合而成。在软件开发并不算太长的历史中，人们为了开发出好的软件和提高软件开发效率，前赴后继探索总结出了一系列程序设计思想方法和原则。一个好的软件设计应该遵循SOLID原则：SRP（The Single Responsibility Principle）、OCP（The Open Closed Principle）、LSP（The Liskov Substitution Principle）、DIP（The Dependency Inversion Principle）、ISP（The Interface Segregation Principle）

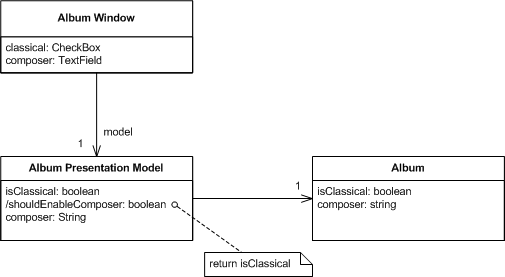
不是解决任何问题都需要从头做起，人们往往乐于重复使用以前使用过的解决问题的方案，重复自己或他人成功的经验。GoF的《设计模式：可复用的面向对象软件的基础》在软件行业掀起了模式旋风。时至今日，理解和应用设计模式基本上成了软件人员的必备技能。在前端程序开发设计中，一个很重要的架构模式是MVC模式，随着软件开发技术的演化，MVC衍生出了MVP、MVVM等模式。

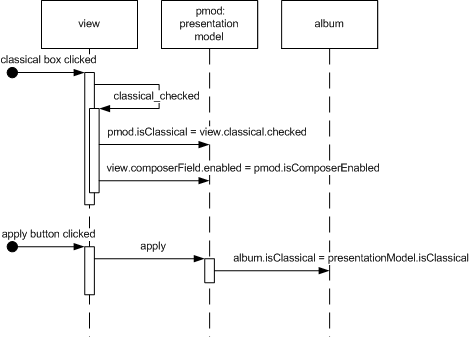
最初的MVC模式主要解决的问题是同一个数据有多种视图展示的场景，MVC模式把应用解耦为Model、View、Controller三部分，View负责数据的呈现，Model表示业务数据，Controller负责处理用户的交互。View和Model之间通过订阅/发布模式，Model的变化会通知View自动变化，View和Controller之间通过策略模式，响应用户的输入，三者之间形成三角关系，互相之间都有关联，也就造成了一定的复杂性。在MVC演进过程中，逐步把Controller置于中间位置，解除了View和Model之间的直接联系，View和Model之间要通过Controller来做中介。这样发展的结果就是MVP模式（Model、View、Presenter）。

但不管改良后的MVC还是MVP，都有View和Controller（Presenter）交互过于频繁，容易把业务逻辑放入Controller（Presenter），造成Controller过于庞大问题。

模式领域的专家Martin Fowler提出了表示模型（Presentation Model）的概念。表示模型代表独立于界面中GUI控件的表示层的状态和行为。表示模型把视图的状态和行为抽取出来，放进作为表示层一部分的模型类中。表示模型和领域层协同，为view提供接口，view把状态存储在表示模型或和表示模型同步状态。表示模型可以认为成独立于GUI框架的抽象的view。通过这样的隔离，使得程序不但更容易扩展，更重要的是，使得难以测试的GUI变得容易测试。

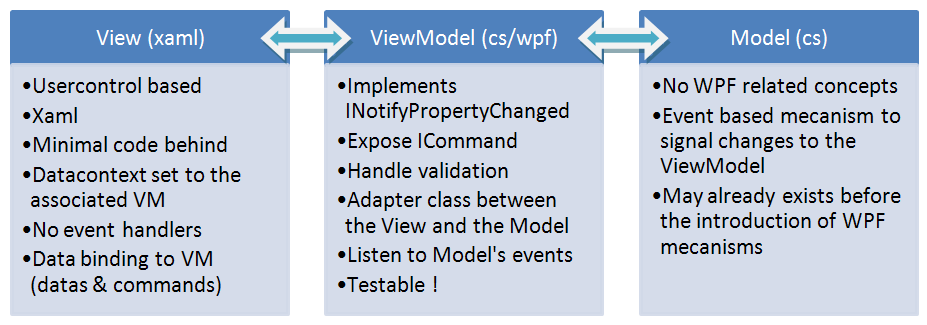
Martin Fowler用一个点击checkbox的例子说明了其工作过程。





以上类图和顺序图清晰地描述了点击选中CheckBox，表示模型和视图以及模型的分工协作过程。表示模型类似于MVP中的Presenter，但由于view中的状态和行为移入表示模型中，这样，view只负责界面的渲染绘制工作，而把状态的维护交给了表示模型，简化了view。通过对表示模型的测试，就可以解决用户界面的逻辑测试问题。

MVVM是Model、View、ViewModel的缩写，这里的ViewModel是Martin Fowle的表示模型在WPF中的术语和实现。借助于数据绑定，WPF极大简化了VM的实现。



* View把DataContext设置成相关的ViewModel，这样很容易绑定到VM的Property上。
* VM调用适当的model方法把它的更新映射到Model上。
* VM公开ICommand类型的Property，View中用命令绑定之，以便执行命令。
* Model通过触发事件通知VM数据的变化。
  1. 数据绑定的基本概念

WPF拥有大量技术特性与生俱来地支持数据绑定，数据绑定是MVVM的主要技术支撑。

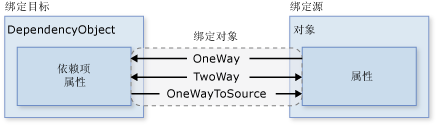
数据绑定在应用程序的视图和业务逻辑之间架起桥梁。如果正确设置了绑定，并且数据提供适当的通知，数据的变更会自动反映到绑定该数据的元素上；反过来，如果元素的值发生改变，绑定的数据也会自动更新。WPF中，元素的可以绑定到CLR对象、XML数据等。

不论要绑定什么元素，也不论数据源的特性是什么，每个绑定都始终遵循下图所示的模型：



每个绑定有以下四个组件：绑定目标对象、目标对象的属性（Target Property）、绑定源、绑定源中要使用的值的路径。目标对象的属性必须为依赖属性，大多数UIElement的属性（Property）是依赖属性，除了只读的，默认都支持数据绑定。绑定源不限于自定义的CLR对象，还可以是XML形式的数据。

在数据绑定中，有三种数据流向：



* OneWay：源的变更导致目标属性更新，适合于控件是只读的情形。
* TwoWay：双向变更，源的变更会导致目标属性变更，反过来目标属性变更也会引起源变更。适用于可编辑的窗体。
* OneWayToSource：和OneWay相反，目标属性改变引发源的更新。

还有一种OneTime绑定，顾名思义，只在开始时用源值初始化目标属性，后续变化不起作用。如果在生命周期内，源值不会发生改变，这是一种相对OneWay更好的选择，意味着更少的开销。

如果是TwoWay或OneWayToSource绑定，目标属性的变化要更新源。但究竟什么时机更新，是由UpdateSourceTrigger决定的。如果它的值是PropertyChanged，会立即更新源；如果值是LostFocus，则在目标属性失去焦点时更新源。

类似于Mode属性，不同的依赖属性的UpdateSourceTrigger有不同的默认值，大多数默认值为PropertyChanged，Text是LostFocus。

下表以TextBox为例展示每种UpdateSourceTrigge值的情形：

| **UpdateSourceTrigger 值** | **源值何时进行更新** | **文本框的示例方案** |
| --- | --- | --- |
| LostFocus (默认为[TextBox.Text](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox.text)) | 当 TextBox 控件失去焦点时 | 一个[TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox)验证逻辑与该键相关联 （请参阅数据验证部分） |
| PropertyChanged | 如您将键入 [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox) | [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox) 聊天室窗口中的控件 |
| Explicit | 当应用程序调用 [UpdateSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.bindingexpression.updatesource) | [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox) （仅当用户单击提交按钮时，更新源值） 的可编辑窗体中的控件 |

* 1. 简单绑定示例

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:src="clr-namespace:SDKSample"  SizeToContent="WidthAndHeight"  Title="Simple Data Binding Sample">  <Window.Resources>  <src:Person x:Key="myDataSource" PersonName="Joe"/>  <Style TargetType="{x:Type Label}">  <Setter Property="DockPanel.Dock" Value="Top"/>  <Setter Property="FontSize" Value="12"/>  </Style>  <Style TargetType="{x:Type TextBox}">  <Setter Property="Width" Value="100"/>  <Setter Property="Height" Value="25"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  </Style>  <Style TargetType="{x:Type TextBlock}">  <Setter Property="Width" Value="100"/>  <Setter Property="Height" Value="25"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  <Setter Property="Padding" Value="3"/>  </Style>  </Window.Resources>  <Border Margin="5" BorderBrush="Aqua" BorderThickness="1" Padding="8" CornerRadius="3">  <DockPanel Width="200" Height="100" Margin="35">  <Label>Enter a Name:</Label>  <TextBox>  <TextBox.Text>  <Binding Source="{StaticResource myDataSource}" Path="PersonName"  UpdateSourceTrigger="PropertyChanged"/>  </TextBox.Text>  </TextBox>    <Label>The name you entered:</Label>  <TextBlock Text="{Binding Source={StaticResource myDataSource}, Path=PersonName}"/>  </DockPanel>  </Border>  </Window> |

本例中，绑定数据源是Person，来自SDKSample命名空间，有一个PersonName Property，值为“Joe”，数据源在资源中定义并实例化，并指定了x:Key。TextBlock控件绑定到数据源的PersonName属性，显示值Joe。

* 1. 绑定源

绑定源对象是指从其获取数据的对象。WPF数据绑定的绑定源可以是所有CLR对象和XML对象。

| **绑定源** | **描述** |
| --- | --- |
| 公共语言运行时 (CLR) 对象 | 可以绑定到任何 公共语言运行时 (CLR) 对象的公有属性、子属性以及索引器。 绑定引擎使用 CLR 反射来获取属性值。 或者，对象实现[ICustomTypeDescriptor](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.icustomtypedescriptor)或注册了[TypeDescriptionProvider](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.typedescriptionprovider)也适用于绑定引擎。  有关如何实现可用作绑定源的类的详细信息，请参阅本主题后面的[为绑定源实现类](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/data/binding-sources-overview#classes)。 |
| 动态对象 | 可以绑定到实现了[IDynamicMetaObjectProvider](https://docs.microsoft.com/zh-cn/dotnet/api/system.dynamic.idynamicmetaobjectprovider)接口的对象的Property和Indexer。如果可以在代码中访问成员，就可以绑定它。 |
| ADO.NET 对象 | 可以将绑定到ADO.NET对象，如[DataTable](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.datatable)。 ADO.NET [DataView](https://docs.microsoft.com/zh-cn/dotnet/api/system.data.dataview)实现[IBindingList](https://docs.microsoft.com/zh-cn/dotnet/api/system.componentmodel.ibindinglist)接口，它提供了绑定引擎侦听的更改通知。 |
| XML 对象 | 你可以将绑定到并运行XPath查询上的[XmlNode](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.xmlnode)， [XmlDocument](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.xmldocument)，或[XmlElement](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.xmlelement)。 一种便利的访问XML数据的方法是绑定到[XmlDataProvider](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.xmldataprovider)标记的对象。 有关详细信息，请参阅[使用 XMLDataProvider 和 XPath 查询绑定到 XML 数据](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/data/how-to-bind-to-xml-data-using-an-xmldataprovider-and-xpath-queries)。  此外可以绑定到[XElement](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.linq.xelement)或[XDocument](https://docs.microsoft.com/zh-cn/dotnet/api/system.xml.linq.xdocument)，或绑定到LINQ to XML 查询的结果。 使用 LINQ to XML 访问 XML 数据的标记中的绑定源的简便方法是使用[ObjectDataProvider](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.objectdataprovider)对象。 有关详细信息，请参阅[绑定到 XDocument、XElement 或 LINQ for XML 查询结果](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/data/how-to-bind-to-xdocument-xelement-or-linq-for-xml-query-results)。 |
| [DependencyObject](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.dependencyobject) 对象 | 可以绑定到的任何依赖关系属性[DependencyObject](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.dependencyobject)。 有关示例，请参阅[绑定两个控件的属性](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/data/how-to-bind-the-properties-of-two-controls)。 |

下面具体介绍常见的绑定情形：

* 简单绑定：之前的例子给出了简单绑定的用法，使用{binding ……}扩展标记把绑定目标绑定到源。
* 绑定两个控件的属性

下面的示例演示如何将 [Canvas](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.canvas) 的 [Background](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.panel.background) 属性绑定到[ComboBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.combobox) 的 [SelectedItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.selector.selecteditem).[Content](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol.content) 属性。

|  |
| --- |
| <Window  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  Width="460" Height="200"  Title="Binding the Properties of Two Controls">  <Window.Resources>  <Style TargetType="TextBlock">  <Setter Property="FontSize" Value="16"/>  <Setter Property="FontWeight" Value="Bold"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  <Setter Property="HorizontalAlignment" Value="Center"/>  </Style>  <Style TargetType="Canvas">  <Setter Property="Height" Value="50"/>  <Setter Property="Width" Value="50"/>  <Setter Property="Margin" Value="8"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  </Style>  <Style TargetType="ComboBox">  <Setter Property="Width" Value="150"/>  <Setter Property="Margin" Value="8"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  </Style>  </Window.Resources>  <Border Margin="10" BorderBrush="Silver" BorderThickness="3" Padding="8">  <DockPanel>  <TextBlock>Choose a Color:</TextBlock>  <ComboBox Name="myComboBox" SelectedIndex="0">  <ComboBoxItem>Green</ComboBoxItem>  <ComboBoxItem>Blue</ComboBoxItem>  <ComboBoxItem>Red</ComboBoxItem>  </ComboBox>  <Canvas>  <Canvas.Background>  <Binding ElementName="myComboBox" Path="SelectedItem.Content"/>  </Canvas.Background>  </Canvas>  </DockPanel>  </Border>  </Window> |

需要注意的是，绑定目标属性必须是依赖属性。

* 使用DataContext

很多时候，窗体中多个组件要绑定一个对象的不同属性（Properties），这时候就适合设置DataContext为整个对象。这是应用中ui常用的场景。

|  |
| --- |
| <Grid  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.DirectionalBinding"  xmlns:c="clr-namespace:SDKSample"  Name="Page1"  >  <Grid.Resources>  <c:NetIncome x:Key="incomeDataSource"/>  <Style TargetType="{x:Type TextBlock}">  <Setter Property="Padding" Value="8"/>  </Style>  <Style TargetType="{x:Type TextBox}">  <Setter Property="Margin" Value="0,6,0,0"/>  </Style>  </Grid.Resources>  <Grid.DataContext>  <Binding Source="{StaticResource incomeDataSource}"/>  </Grid.DataContext>  <Grid.ColumnDefinitions>  <ColumnDefinition Width="120" />  <ColumnDefinition Width="120" />  <ColumnDefinition Width="400" />  </Grid.ColumnDefinitions>  <Grid.RowDefinitions>  <RowDefinition Height="25"/>  <RowDefinition Height="25"/>  <RowDefinition Height="25"/>  <RowDefinition Height="25"/>  <RowDefinition Height="25"/>  <RowDefinition Height="25"/>  </Grid.RowDefinitions>  <!-- OneTime binding example -->  <Label Grid.Row="0" Grid.Column="0">Total Income:</Label>  <TextBlock Name="IncomeText" Grid.Row="0" Grid.Column="1"  Text="{Binding Path=TotalIncome, Mode=OneTime}"/>  <TextBlock Grid.Row="0" Grid.Column="2">OneTime Binding</TextBlock>  <!-- OneWay binding example -->  <Label Grid.Row="1" Grid.Column="0">Rent</Label>  <TextBlock Grid.Row="1" Grid.Column="1" Name="RentText"  Text="{Binding Path=Rent, Mode=OneWay, NotifyOnTargetUpdated=True}"  TargetUpdated="OnTargetUpdated"/>  <TextBlock Grid.Row="1"  Grid.Column="2">OneWay Binding, with TargetUpdated event handling</TextBlock>  <!-- TwoWay binding example (default for TextBox), with UpdateSourceTrigger=PropertyChanged-->  <Label Grid.Row="2" Grid.Column="0">Food</Label>  <TextBox Name="FoodText" Grid.Row="2" Grid.Column="1"  Text="{Binding Path=Food, UpdateSourceTrigger=PropertyChanged}" />  <TextBlock Grid.Row="2" Grid.Column="2">  TwoWay Binding (TextBox default), Update on PropertyChanged</TextBlock>  <!-- TwoWay binding example (default for TextBox), with  UpdateSourceTrigger=LostFocus (default for TextBox) -->  <Label Grid.Row="3" Grid.Column="0">Miscellaneous</Label>  <TextBox Grid.Row="3" Grid.Column="1" Name="MiscText" Text="{Binding Path=Misc}"/>  <TextBlock Grid.Row="3" Grid.Column="2">  TwoWay Binding (TextBox default), Update on LostFocus (TextBox default)</TextBlock>  <!-- OneWay binding example (which is default for Text) -->  <Label Grid.Row="4" Grid.Column="0">Savings</Label>  <TextBlock Grid.Row="4" Grid.Column="1" Name="SavingsText" Text="{Binding Path=Savings}"/>  <TextBlock Grid.Row="4" Grid.Column="2">OneWay Binding</TextBlock>  <Button Grid.Row="5" Grid.Column="0" Click="OnRentRaise">Raise the Rent!</Button>  <TextBlock Name="infoText" Grid.Row="5" Grid.Column="1"  Grid.ColumnSpan="2"/>  </Grid> |

以下是NetIncome类的定义：

|  |
| --- |
| public class NetIncome : INotifyPropertyChanged  {  private int totalIncome = 5000;  private int rent = 2000;  private int food = 0;  private int misc = 0;  private int savings = 0;  public NetIncome()  {  savings = totalIncome - (rent+food+misc);  }  public int TotalIncome  {  get  {  return totalIncome;  }  set  {  if( TotalIncome != value)  {  totalIncome = value;  OnPropertyChanged("TotalIncome");  }  }  }  public int Rent  {  get  {  return rent;  }  set  {  if( Rent != value)  {  rent = value;  OnPropertyChanged("Rent");  UpdateSavings();  }  }  }  public int Food  {  get  {  return food;  }  set  {  if( Food != value)  {  food = value;  OnPropertyChanged("Food");  UpdateSavings();  }  }  }  public int Misc  {  get  {  return misc;  }  set  {  if( Misc != value)  {  misc = value;  OnPropertyChanged("Misc");  UpdateSavings();  }  }  }  public int Savings  {  get  {  return savings;  }  set  {  if( Savings != value)  {  savings = value;  OnPropertyChanged("Savings");  UpdateSavings();  }  }  }  private void UpdateSavings()  {  Savings = TotalIncome - (Rent+Misc+Food);  if(Savings < 0)  {}  else if(Savings >= 0)  {}  }  public event PropertyChangedEventHandler PropertyChanged;  private void OnPropertyChanged(String info)  {  PropertyChangedEventHandler handler = PropertyChanged;  if (handler !=null)  {  handler(this, new PropertyChangedEventArgs(info));  }  }  } |

在绑定声明中，只需要{binding }，不需要指定source，path用来查找指定的Property。

如果要绑定已在代码中实例化的对象，则需要用编程方式，在代码中设置控件的DataContext属性。

WPF的属性系统的继承功能可以保证子元素从最近的父元素获得DataContext值。如果使用了以下选项，这些选项优先于继承：

| **属性** | **描述** |
| --- | --- |
| [Source](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.source) | 使用此属性将源设置为对象的实例。 如果不需要特定功能（用于建立其中有多个属性继承相同的数据上下文的作用域），则可以使用 [Source](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.source) 属性而不是 DataContext 属性。 有关详细信息，请参阅[Source](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.source)。 |
| [RelativeSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.relativesource) | 当希望指定相对于绑定目标位置的源时，这很有用。 当想要将元素的一个属性绑定到同一元素的另一个属性时，或者如果要在样式或模板中定义绑定，则可能需要使用此属性。 有关详细信息，请参阅[RelativeSource](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.relativesource)。 |
| [ElementName](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.elementname) | 指定一个字符串，用于表示你希望绑定到的元素。 当希望绑定到应用程序上另一个元素的属性时，这很有用。 例如，想用 [Slider](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.slider) 来控制应用程序中另一个控件的高度，或想将控件的 [Content](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.contentcontrol.content) 绑定到 [ListBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listbox) 控件的 [SelectedValue](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.primitives.selector.selectedvalue)属性。 有关详细信息，请参阅 [ElementName](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.elementname)。 |

* 把集合对象绑定到列表控件

需要在XAML中或用代码设置列表控件的ItemsSource属性。

|  |
| --- |
| <Window x:Class="SDKSample.Window1"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:local="clr-namespace:SDKSample"  Title="Binding to a Collection"  SizeToContent="WidthAndHeight">  <Window.Resources>  <local:People x:Key="MyFriends"/>  <Style TargetType="ListBoxItem">  <Setter Property="FontFamily" Value="Verdana"/>  <Setter Property="FontSize" Value="11"/>  <Setter Property="Padding" Value="10"/>  </Style>  <DataTemplate x:Key="DetailTemplate">  <Border Width="300" Height="100" Margin="20"  BorderBrush="Aqua" BorderThickness="1" Padding="8">  <Grid>  <Grid.RowDefinitions>  <RowDefinition/>  <RowDefinition/>  <RowDefinition/>  </Grid.RowDefinitions>  <Grid.ColumnDefinitions>  <ColumnDefinition/>  <ColumnDefinition/>  </Grid.ColumnDefinitions>  <TextBlock Grid.Row="0" Grid.Column="0" Text="First Name:"/>  <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=FirstName}"/>  <TextBlock Grid.Row="1" Grid.Column="0" Text="Last Name:"/>  <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=LastName}"/>  <TextBlock Grid.Row="2" Grid.Column="0" Text="Home Town:"/>  <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=HomeTown}"/>  </Grid>  </Border>  </DataTemplate>  </Window.Resources>  <StackPanel>  <TextBlock FontFamily="Verdana" FontSize="11"  Margin="5,15,0,10" FontWeight="Bold">My Friends:</TextBlock>  <ListBox Width="200" IsSynchronizedWithCurrentItem="True"  ItemsSource="{Binding Source={StaticResource MyFriends}}"/>  <TextBlock FontFamily="Verdana" FontSize="11"  Margin="5,15,0,5" FontWeight="Bold">Information:</TextBlock>  <ContentControl Content="{Binding Source={StaticResource MyFriends}}"  ContentTemplate="{StaticResource DetailTemplate}"/>  </StackPanel>  </Window> |

本例也是主从绑定的示例。ContentControl绑定到集合的当前项，ListBox的IsSynchronizedWithCurrentItem="True"保证当前项和选择项同步。本例还用到了数据模板，事实上，如果不用数据模板，将不会达到我们期望的效果。

ItemsSource 优选使用ObservableCollection<T>，有更好的性能，因为实现了INotifyCollectionChanged接口和INotifyPropertyChanged接口，数据源的改变可以立即通知列表控件随之更新。

* 绑定到 ADO.NET 数据源

先看示例：

|  |
| --- |
| DataSet myDataSet;  private void OnInit(object sender, EventArgs e)  {  string mdbFile = Path.Combine(AppDataPath, "BookData.mdb");  string connString = string.Format(  "Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}", mdbFile);  OleDbConnection conn = new OleDbConnection(connString);  OleDbDataAdapter adapter = new OleDbDataAdapter("SELECT \* FROM BookTable;", conn);  myDataSet = new DataSet();  adapter.Fill(myDataSet, "BookTable");  // myListBox is a ListBox control.  // Set the DataContext of the ListBox to myDataSet  myListBox.DataContext = myDataSet;  } |

使用ADO.net获取数据集DataSet，然后把ListBox列表控件的DataContext设置为DataSet对象。

|  |
| --- |
| <ListBox Name="myListBox" Height="200"  ItemsSource="{Binding Path=BookTable}"  ItemTemplate ="{StaticResource BookItemTemplate}"/> |

接下来，在XAML中设置ListBox的ItemSource属性。

下面是定义数据显示方式的数据模板BookItemTemplate：

|  |
| --- |
| <StackPanel.Resources>  <c:IntColorConverter x:Key="MyConverter"/>  <DataTemplate x:Key="BookItemTemplate">  <Grid>  <Grid.ColumnDefinitions>  <ColumnDefinition Width="250" />  <ColumnDefinition Width="100" />  <ColumnDefinition Width="\*"/>  </Grid.ColumnDefinitions>  <TextBlock Text="{Binding Path=Title}" Grid.Column="0"  FontWeight="Bold" />  <TextBlock Text="{Binding Path=ISBN}" Grid.Column="1" />  <TextBlock Grid.Column="2" Text="{Binding Path=NumPages}"  Background="{Binding Path=NumPages,  Converter={StaticResource MyConverter}}"/>  </Grid>  </DataTemplate>  </StackPanel.Resources> |

IntColorConverter是将int转换为颜色的转换器。

* 使用 XMLDataProvider 和 XPath 查询绑定到 XML 数据

|  |
| --- |
| <StackPanel  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  Background="Cornsilk">  <StackPanel.Resources>  <XmlDataProvider x:Key="InventoryData" XPath="Inventory/Books">  <x:XData>  <Inventory xmlns="">  <Books>  <Book ISBN="0-7356-0562-9" Stock="in" Number="9">  <Title>XML in Action</Title>  <Summary>XML Web Technology</Summary>  </Book>  <Book ISBN="0-7356-1370-2" Stock="in" Number="8">  <Title>Programming Microsoft Windows With C#</Title>  <Summary>C# Programming using the .NET Framework</Summary>  </Book>  <Book ISBN="0-7356-1288-9" Stock="out" Number="7">  <Title>Inside C#</Title>  <Summary>C# Language Programming</Summary>  </Book>  <Book ISBN="0-7356-1377-X" Stock="in" Number="5">  <Title>Introducing Microsoft .NET</Title>  <Summary>Overview of .NET Technology</Summary>  </Book>  <Book ISBN="0-7356-1448-2" Stock="out" Number="4">  <Title>Microsoft C# Language Specifications</Title>  <Summary>The C# language definition</Summary>  </Book>  </Books>  <CDs>  <CD Stock="in" Number="3">  <Title>Classical Collection</Title>  <Summary>Classical Music</Summary>  </CD>  <CD Stock="out" Number="9">  <Title>Jazz Collection</Title>  <Summary>Jazz Music</Summary>  </CD>  </CDs>  </Inventory>  </x:XData>  </XmlDataProvider>  </StackPanel.Resources>  <TextBlock FontSize="18" FontWeight="Bold" Margin="10"  HorizontalAlignment="Center">XML Data Source Sample</TextBlock>  <ListBox  Width="400" Height="300" Background="Honeydew">  <ListBox.ItemsSource>  <Binding Source="{StaticResource InventoryData}"  XPath="\*[@Stock='out'] | \*[@Number>=8 or @Number=3]"/>  </ListBox.ItemsSource>  <!--Alternatively, you can do the following. -->  <!--<ListBox Width="400" Height="300" Background="Honeydew"  ItemsSource="{Binding Source={StaticResource InventoryData},  XPath=\*[@Stock\=\'out\'] | \*[@Number>\=8 or @Number\=3]}">-->  <ListBox.ItemTemplate>  <DataTemplate>  <TextBlock FontSize="12" Foreground="Red">  <TextBlock.Text>  <Binding XPath="Title"/>  </TextBlock.Text>  </TextBlock>  </DataTemplate>  </ListBox.ItemTemplate>  </ListBox>  </StackPanel> |

本例中，XPath起到过滤器的作用。在XmlDataProvider只公开Book项。结果显示Book中Stock=“out”并且3<=number<=8的那些项的Title。

以下是XPath的一些用法：

* XPath="Book[1]" 将返回第一个 Book 元素（“XML in Action”）。 请注意，XPath 索引从 1 而不是从 0 开始。
* XPath="Book[@\*]" 将返回带有任意属性的所有 Book 元素。
* XPath="Book[last()-1]" 将返回倒数第二个 Book 元素（“Introducing Microsoft .NET”）。
* XPath="\*[position()>3]" 将返回除前 3 个元素之外的所有 Book 元素。

XmlDataProvider还支持从外部获取XML数据：

<XmlDataProvider x:Key="BookData" Source="data\bookdata.xml" XPath="Books"/>

或从远程：

<XmlDataProvider x:Key="BookData" Source="http://MyUrl" XPath="Books"/>

* 绑定到 XDocument、XElement 或 LINQ to XML 查询结果

|  |
| --- |
| <StackPanel Name="stacky">  <StackPanel.Resources>  <DataTemplate DataType="{}{http://planetsNS}Planet" >  <StackPanel Orientation="Horizontal">  <TextBlock Width="100" Text="{Binding Path=Element[{http://planetsNS}DiameterKM].Value}" />  <TextBlock Width="100" Text="{Binding Path=Attribute[Name].Value}" />  <TextBlock Text="{Binding Path=Element[{http://planetsNS}Details].Value}" />  </StackPanel>  </DataTemplate>  </StackPanel.Resources> |

数据模板中，数据类型为Planet，它的XML命名空间必须包含在{}中，如果出现在XML标记扩展可能出现的位置，前面哟加上空的{}转义。

|  |
| --- |
| <ItemsControl  ItemsSource="{Binding }" >  </ItemsControl>  </StackPanel> |

下面的 C# 代码调用 Load 并将堆栈面板的数据上下文设置为 http://planetsNS XML 命名空间中名为 SolarSystemPlanets 的元素的所有子元素。

|  |
| --- |
| planetsDoc = XDocument.Load("../../Planets.xml");  stacky.DataContext = planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements(); |

使用Linq to XML：

|  |
| --- |
| stacky.DataContext =  from c in planetsDoc.Element("{http://planetsNS}SolarSystemPlanets").Elements()  orderby Int32.Parse(c.Element("{http://planetsNS}DiameterKM").Value)  select c; |

* 绑定到 LINQ 查询的结果

以下示例创建两个列表框。 第一个列表框包含三个列表项。

|  |
| --- |
| <ListBox SelectionChanged="ListBox\_SelectionChanged"  SelectedIndex="0" Margin="10,0,10,0" >  <ListBoxItem>1</ListBoxItem>  <ListBoxItem>2</ListBoxItem>  <ListBoxItem>3</ListBoxItem>  </ListBox>  <ListBox Width="400" Margin="10" Name="myListBox"  HorizontalContentAlignment="Stretch"  ItemsSource="{Binding}"  ItemTemplate="{StaticResource myTaskTemplate}"/> |

选中第一个列表中的项，将触发ListBox\_SelectionChanged事件：

|  |
| --- |
| private void ListBox\_SelectionChanged(object sender, SelectionChangedEventArgs e)  {  int pri = Int32.Parse(((sender as ListBox).SelectedItem as ListBoxItem).Content.ToString());  this.DataContext = from task in tasks  where task.Priority == pri  select task;  } |

* 绑定到方法

|  |
| --- |
| <Window.Resources>  <ObjectDataProvider ObjectType="{x:Type local:TemperatureScale}"  MethodName="ConvertTemp" x:Key="convertTemp">  <ObjectDataProvider.MethodParameters>  <system:Double>0</system:Double>  <local:TempType>Celsius</local:TempType>  </ObjectDataProvider.MethodParameters>  </ObjectDataProvider>  <local:DoubleToString x:Key="doubleToString" />  </Window.Resources> |

TemperatureScale类有一个方法ConvertTemp，ObjectDataProvider实例化TemperatureScale对象，指定调用方法ConvertTemp和参数。

|  |
| --- |
| <Label Grid.Row="1" HorizontalAlignment="Right">Enter the degree to convert:</Label>  <TextBox Grid.Row="1" Grid.Column="1" Name="tb">  <TextBox.Text>  <Binding Source="{StaticResource convertTemp}" Path="MethodParameters[0]"  BindsDirectlyToSource="true" UpdateSourceTrigger="PropertyChanged"  Converter="{StaticResource doubleToString}">  <Binding.ValidationRules>  <local:InvalidCharacterRule/>  </Binding.ValidationRules>  </Binding>  </TextBox.Text>  </TextBox>  <ComboBox Grid.Row="1" Grid.Column="2"  SelectedValue="{Binding Source={StaticResource convertTemp},  Path=MethodParameters[1], BindsDirectlyToSource=true}">  <local:TempType>Celsius</local:TempType>  <local:TempType>Fahrenheit</local:TempType>  </ComboBox>  <Label Grid.Row="2" HorizontalAlignment="Right">Result:</Label>  <Label Content="{Binding Source={StaticResource convertTemp}}"  Grid.Row="2" Grid.Column="1" Grid.ColumnSpan="2"/> |

TextBox和ComboBox设定ConvertTemp的两个参数，Label显示调用ConvertTemp的结果。

**更多的数据绑定示例可参见MSDN**[操作说明主题](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/data/data-binding-how-to-topics)

* 1. 数据转换

控件显示的值一般是字符串或数值类型，许多情况下，WPF为我们提供了绑定源和目标对象的自动转换，但有时候会遇到我们自己定义的复杂类型，WPF无法为我们自动转换或者自动转换不符合我们的要求，这时候就需要我们实现自己的转换器。

实现自定义转换器需要实现IvalueConverter接口，如下例所示：

|  |
| --- |
| [ValueConversion(typeof(Color), typeof(SolidColorBrush))]  public class ColorBrushConverter : IValueConverter  {  public object Convert(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)  {  Color color = (Color)value;  return new SolidColorBrush(color);  }  public object ConvertBack(object value, Type targetType, object parameter, System.Globalization.CultureInfo culture)  {  return null;  }  } |

之前ADO.net的例子中，我们已经见到如何使用自定义的转换器。

下面是一些需要提供自定义转换器的典型场景：

* 数据应根据区域性以不同方式显示。 例如，可能需要根据在特定区域性中使用的值或标准，实现货币转换器或日历日期/时间转换器。
* 使用的数据不一定会更改属性的文本值，但会更改其他某个值（如图像的源，或显示文本的颜色或样式）。 在这种情况下，可以通过转换可能不合适的属性绑定（如将文本字段绑定到表单元格的 Background 属性）来使用转换器。
* 将多个控件或控件的多个属性绑定到相同数据。 在这种情况下，主绑定可能仅显示文本，而其他绑定则处理特定的显示问题，但仍使用同一绑定作为源信息。
* 使用MultiBinding时，要定义转换器，该转换器要实ImultiValueConverter接口。
  1. 数据验证

前端应用程序中，用户输入的数据一般都要加以验证，以确保用户输入了预期信息。验证逻辑基于类型、范围、格式或其他应用程序特定的要求。

WPF的数据绑定模型可以将验证规则和绑定对象关联起来。

|  |
| --- |
| <TextBox Name="StartPriceEntryForm" Grid.Row="2" Grid.Column="1"  Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5">  <TextBox.Text>  <Binding Path="StartPrice" UpdateSourceTrigger="PropertyChanged">  <Binding.ValidationRules>  <ExceptionValidationRule />  </Binding.ValidationRules>  </Binding>  </TextBox.Text>  </TextBox> |

本例中为TextBox设置了ExceptionValidationRule验证规则，StartPrice为整型，如果用户输入无法转换为整数，就会引发异常，导致绑定无效。可以设置ValidatesOnExceptions="True"代替显示设置。

WPF内置两种ValidationRules，除了ExceptionValidationRule，还有DataErrorValidationRule，检查由实现了IDataErrorInfo接口的绑定源引发的错误，它可以显式设置，也可以设置ValidationOnDataErrors="True"来代替。

绑定源实现IdataErrorInfo接口：

|  |
| --- |
| public class Person : IDataErrorInfo  {  private int age;  public int Age  {  get { return age; }  set { age = value; }  }  public string Error  {  get  {  return null;  }  }  public string this[string name]  {  get  {  string result = null;  if (name == "Age")  {  if (this.age < 0 || this.age > 150)  {  result = "Age must not be less than 0 or greater than 150.";  }  }  return result;  }  }  } |

还可以创建自己的验证规则：

|  |
| --- |
| class FutureDateRule : ValidationRule  {  public override ValidationResult Validate(object value, CultureInfo cultureInfo)  {  DateTime date;  try  {  date = DateTime.Parse(value.ToString());  }  catch (FormatException)  {  return new ValidationResult(false, "Value is not a valid date.");  }  if (DateTime.Now.Date > date)  {  return new ValidationResult(false, "Please enter a date in the future.");  }  else  {  return ValidationResult.ValidResult;  }  }  } |

使用自定义规则如下所示：

|  |
| --- |
| <TextBox Name="StartDateEntryForm" Grid.Row="3" Grid.Column="1"  Validation.ErrorTemplate="{StaticResource validationTemplate}"  Style="{StaticResource textStyleTextBox}" Margin="8,5,0,5">  <TextBox.Text>  <Binding Path="StartDate" UpdateSourceTrigger="PropertyChanged"  Converter="{StaticResource dateConverter}" >  <Binding.ValidationRules>  <src:FutureDateRule />  </Binding.ValidationRules>  </Binding>  </TextBox.Text>  </TextBox> |

验证出现错误时，我们一般希望能有错误信息提示。WPF默认会在出错的控件加上红色边框。也可以定义自己的ErrorTemplate：

|  |
| --- |
| <ControlTemplate x:Key="validationTemplate">  <DockPanel>  <TextBlock Foreground="Red" FontSize="20">!</TextBlock>  <AdornedElementPlaceholder/>  </DockPanel>  </ControlTemplate> |

AdornedElementPlaceholder元素指定要装饰的控件应放置的位置。

还可以创建ToolTip显示错误信息：

|  |
| --- |
| <Style x:Key="textStyleTextBox" TargetType="TextBox">  <Setter Property="Foreground" Value="#333333" />  <Setter Property="MaxLength" Value="40" />  <Setter Property="Width" Value="392" />  <Style.Triggers>  <Trigger Property="Validation.HasError" Value="true">  <Setter Property="ToolTip"  Value="{Binding RelativeSource={RelativeSource Self},  Path=(Validation.Errors)[0].ErrorContent}"/>  </Trigger>  </Style.Triggers>  </Style> |

更多验证示例可以参考MSDN。

* 1. MultiBinding

有时候，一个绑定目标显示的值可能来自多个源，或根据条件做出不同的选择，这时候就需要用到MultiBinding。MultiBinding需要使用自定义的实现ImultiValueConverter接口的转换器。示例如下：

|  |
| --- |
| <Window x:Class="MultiBinding.MainWindow"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  xmlns:local="clr-namespace:MultiBinding"  mc:Ignorable="d"  Title="MainWindow" Height="350" Width="525">  <Window.Resources>  <local:NameList x:Key="NameListData"/>  <local:NameConverter x:Key="MyNameConverter"/>  <DataTemplate x:Key="NameItemTemplate">  <TextBlock>  <TextBlock.Text>  <MultiBinding Converter="{StaticResource MyNameConverter}">  <Binding Path="FirstName"/>  <Binding Path="LastName"/>  </MultiBinding>  </TextBlock.Text>  </TextBlock>  </DataTemplate>  <Style TargetType="{x:Type TextBlock}">  <Setter Property="Width" Value="120"/>  <Setter Property="Background" Value="Silver"/>  <Setter Property="HorizontalAlignment" Value="Center"/>  </Style>  </Window.Resources>  <StackPanel>  <TextBlock FontSize="18" FontWeight="Bold" Margin="10"  Background="White" Width="Auto">MultiBinding Sample</TextBlock>  <ListBox Width="200"  ItemsSource="{Binding Source={StaticResource NameListData}}"  ItemTemplate="{StaticResource NameItemTemplate}"  IsSynchronizedWithCurrentItem="True"/>  <TextBlock Padding="0,20,0,0" FontSize="11" Background="White">Normal Format:</TextBlock>  <TextBlock Name="textBox1" DataContext="{StaticResource NameListData}">  <TextBlock.Text>  <MultiBinding Converter="{StaticResource MyNameConverter}"  ConverterParameter="FormatNormal">  <Binding Path="FirstName"/>  <Binding Path="LastName"/>  </MultiBinding>  </TextBlock.Text>  </TextBlock>  <TextBlock Padding="0,20,0,0" FontSize="11" Background="White">Last Name First Format:</TextBlock>  <TextBlock Name="textBox2" DataContext="{StaticResource NameListData}">  <TextBlock.Text>  <MultiBinding Converter="{StaticResource MyNameConverter}"  ConverterParameter="FormatLastFirst">  <Binding Path="FirstName"/>  <Binding Path="LastName"/>  </MultiBinding>  </TextBlock.Text>  </TextBlock>  </StackPanel>  </Window> |

转换器定义如下：

|  |
| --- |
| public class NameConverter : IMultiValueConverter  {  public object Convert(object[] values, Type targetType, object parameter, CultureInfo culture)  {  string name;  switch ((string) parameter)  {  case "FormatLastFirst":  name = values[1] + ", " + values[0];  break;  default:  name = values[0] + " " + values[1];  break;  }  return name;  }  public object[] ConvertBack(object value, Type[] targetTypes, object parameter, CultureInfo culture)  {  var splitValues = ((string) value).Split(' ');  return splitValues;  }  } |

* 1. 在代码中创建绑定及其他

还可以在代码中创建绑定。下面示例中TheConverter时一个转换器：

|  |
| --- |
| private void OnPageLoaded(object sender, EventArgs e)  {  // Make a new source, to grab a new timestamp  MyData myChangedData = new MyData();    // Create a new binding  // TheDate is a property of type DateTime on MyData class  Binding myNewBindDef = new Binding("TheDate");    myNewBindDef.Mode = BindingMode.OneWay;  myNewBindDef.Source = myChangedData;  myNewBindDef.Converter = TheConverter;  myNewBindDef.ConverterCulture = new CultureInfo("en-US");    // myDatetext is a TextBlock object that is the binding target object  BindingOperations.SetBinding(myDateText, TextBlock.TextProperty, myNewBindDef);  BindingOperations.SetBinding(myDateText, TextBlock.ForegroundProperty, myNewBindDef);  lbChooseCulture.SelectedIndex = 0;  } |

FrameworkElement或FrameworkContentElement对象可以直接调用SetBinding方法，而不是BindingOperations.SetBinding。

BindingOperations.SetBinding用法示例如下：

|  |
| --- |
| //make a new source  MyData myDataObject = new MyData(DateTime.Now);  Binding myBinding = new Binding("MyDataProperty");  myBinding.Source = myDataObject;  BindingOperations.SetBinding(myText, TextBlock.TextProperty, myBinding); |

如果绑定方向是OneWay或TwoWay，一般建议绑定源实现INotifyPropertyChanged接口，以确保使用中的数据保持最新（集合派生自ObservableCollection<T>）：

|  |
| --- |
| public class MyData : INotifyPropertyChanged  {  private string myDataProperty;  public MyData() { }  public MyData(DateTime dateTime)  {  myDataProperty = "Last bound time was " + dateTime.ToLongTimeString();  }  public String MyDataProperty  {  get { return myDataProperty; }  set  {  myDataProperty = value;  OnPropertyChanged("MyDataProperty");  }  }  public event PropertyChangedEventHandler PropertyChanged;    private void OnPropertyChanged(string info)  {  PropertyChangedEventHandler handler = PropertyChanged;  if (handler != null)  {  handler(this, new PropertyChangedEventArgs(info));  }  }  } |

XAML中绑定的Path在之前例子中出现很多，常见用法是指定源对象或其属性。用/和属性名遍历集合的属性：Path=/Offices/ManagerName；Path=/指定集合视图的当前项。

转义机制：

* 在索引器 ([ ]) 内部，脱字符号 (^) 用于对下一个字符进行转义。
* 如果您设置[Path](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.data.binding.path)在 XAML，您还需要转义 （使用 XML 实体） 对 XML 语言定义的某些字符：
  + 使用 & 对字符“&”进行转义。
  + 使用 > 对结束标记“>”进行转义。
* 此外，如果在属性中使用标记扩展语法描述整个绑定，需要（使用反斜杠 \）对 WPF 标记扩展分析器专用的字符进行转义：
  + 反斜杠 (\) 本身是转义字符。
  + 等号 (=) 将属性名与属性值分隔开。
  + 逗号 (,) 用于分隔属性。
  + 右大括号 (}) 是标记扩展的结尾。

数据绑定如果遇到大量数据，如大的集合数据，可能会有性能问题，可参考[优化性能：数据绑定](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/advanced/optimizing-performance-data-binding)。

1. XAML资源

资源是应用程序中可重复使用的对象。WPF除了可以支持传统的应用程序资源，如资源文件、内容文件以及用Pack Uri标识的运行是加载的资源，还支持XAML资源。本章重点讲述XAML资源，要了解其他资源可以参考[WPF 应用程序资源、内容和数据文件](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/wpf-application-resource-content-and-data-files)和[WPF 中的 Pack URI](https://docs.microsoft.com/zh-cn/dotnet/framework/wpf/app-development/pack-uris-in-wpf)。

* 1. 定义XAML资源

下面示例在页面根元素定义了SolidColorBrush资源，并随后引用：

|  |
| --- |
| <Page Name="root"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  >  <Page.Resources>  <SolidColorBrush x:Key="MyBrush" Color="Gold"/>  <Style TargetType="Border" x:Key="PageBackground">  <Setter Property="Background" Value="Blue"/>  </Style>  <Style TargetType="TextBlock" x:Key="TitleText">  <Setter Property="Background" Value="Blue"/>  <Setter Property="DockPanel.Dock" Value="Top"/>  <Setter Property="FontSize" Value="18"/>  <Setter Property="Foreground" Value="#4E87D4"/>  <Setter Property="FontFamily" Value="Trebuchet MS"/>  <Setter Property="Margin" Value="0,40,10,10"/>  </Style>  <Style TargetType="TextBlock" x:Key="Label">  <Setter Property="DockPanel.Dock" Value="Right"/>  <Setter Property="FontSize" Value="8"/>  <Setter Property="Foreground" Value="{StaticResource MyBrush}"/>  <Setter Property="FontFamily" Value="Arial"/>  <Setter Property="FontWeight" Value="Bold"/>  <Setter Property="Margin" Value="0,3,10,0"/>  </Style>  </Page.Resources>  <StackPanel>  <Border Style="{StaticResource PageBackground}">  <DockPanel>  <TextBlock Style="{StaticResource TitleText}">Title</TextBlock>  <TextBlock Style="{StaticResource Label}">Label</TextBlock>  <TextBlock DockPanel.Dock="Top" HorizontalAlignment="Left" FontSize="36" Foreground="{StaticResource MyBrush}" Text="Text" Margin="20" />  <Button DockPanel.Dock="Top" HorizontalAlignment="Left" Height="30" Background="{StaticResource MyBrush}" Margin="40">Button</Button>  <Ellipse DockPanel.Dock="Top" HorizontalAlignment="Left" Width="100" Height="100" Fill="{StaticResource MyBrush}" Margin="40" />  </DockPanel>  </Border>  </StackPanel>  </Page> |

每个框架级元素（FrameworkElement或FrameworkContentElement类）都有一个Resources属性，这是定义资源的地方（作为资源字典ResourcesDictionary）。任意一个元素都可以定义资源，但通常把它们定义在根元素上。Application上也可以定义资源，应用程序的其他任何部分都可以访问Application级资源。

资源字典中的每个资源都必须有唯一键，用x:Key标识，它通常是字符串，但可以使用扩展标记将其设置为其他对象类型。但并不是每个资源都需要显示的x:Key，如之前介绍过的样式，如果不显示指定x:Key，则样式作用域所有TargetType指定的类型，它实际上由一个隐式的x:Key，值为TargetType指向的类型。

样式、数据模板等一般都定义在样式字典中。

* 1. 引用资源

在XAML中，使用StaticResource和DynamicResource标记扩展引用资源。静态资源引用在加载是时处理，动态资源引用使用Lazy加载方式，先计算一个表达式，直到真正使用时才加载。静态资源引用必须先定义才能引用，动态资源引用则无此限制。在运行过程中，所引用资源发生变更，动态引用会随之变化，而静态引用则值使用最初加载时的资源值。

**静态资源查找行为**：

1. 在所设置属性的元素的资源字典中查找。
2. 向上查找逻辑树，在父元素中查找资源字典，一直到达根元素，查到时为止。
3. 查找Application的资源字典。

**动态资源查找行为**：

1. 在所设置属性的元素的资源字典中查找。
2. 向上查找逻辑树，在父元素中查找资源字典，一直到达根元素，查到时为止。
3. 查找Application的资源字典。
4. 为当前活动主题检查主题（Theme）资源字典中。如果运行时主题发生改变，则会重新计算。
5. 检查系统资源。

动态资源引用由一些限制，必须满足下列条件之一：

* 要设置属性必须是FrameworkElement或FrameworkContentElement元素的属性，属性必须基于依赖属性。
* 引用的值是StyleSetter中设定的。
* [Freezable](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.freezable)属性
  1. 使用代码创建和引用资源

|  |
| --- |
| void SetBGByResource(object sender, RoutedEventArgs e)  {  Button b = sender as Button;  b.Background = (Brush)this.FindResource("RainbowBrush");  } |

引用资源调用FindResource方法或TryFindResource方法，后者不会引发异常。另一种访问资源的方法是[SetResourceReference](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.frameworkelement.setresourcereference)，还可以直接把资源作为字典的属性来访问。

使用代码创建资源，要先创建ResourceDictionary实例，然后用ResourceDictionary.Add添加资源。

* 1. 合并资源字典

WPF提供合并资源字典功能，可以把资源定义成单独的xaml文件，在应用程序中共享资源。

|  |
| --- |
| <Page.Resources>  <ResourceDictionary>  <ResourceDictionary.MergedDictionaries>  <ResourceDictionary Source="myresourcedictionary.xaml"/>  <ResourceDictionary Source="myresourcedictionary2.xaml"/>  </ResourceDictionary.MergedDictionaries>  </ResourceDictionary>  </Page.Resources> |

1. 模板

WPF控件模型的扩展性极大减少了创建新控件的需要，但某些情况下，我们还会有创建新控件的需要。以前，如果要扩展一个控件的外观和行为，往往需要创建新的控件，继承现有控件并重写控件的渲染方法。在WPF中，我们可以利用WPF的内容模型、样式、模板和触发器，轻松地达到同样的目标，扩展现有控件的外观、行为、数据呈现方式，或组合生成新的控件。

本章主要介绍WPF的模板。WPF的模板提供了丰富灵活的功能，可以说是WPF最强大特性之一。WPF的模板依其功能用途分三种：数据模板（DataTemplate）、控件模板（ControlTemplate）和面板模板（ItemsPanelTemplate），它们都继承自FrameworkTemplate抽象类。

* 1. 控件模板（ControlTemplate）

控件模板用来指定可在多个实例之间共享的可视化结构和行为。

|  |
| --- |
| <Style TargetType="Button">  <!--Set to true to not get any properties from the themes.-->  <Setter Property="OverridesDefaultStyle" Value="True"/>  <Setter Property="Template">  <Setter.Value>  <ControlTemplate TargetType="Button">  <Grid>  <Ellipse Fill="{TemplateBinding Background}"/>  <ContentPresenter HorizontalAlignment="Center"  VerticalAlignment="Center"/>  </Grid>  </ControlTemplate>  </Setter.Value>  </Setter>  </Style> |

本例的ControlTemplate把Button控件的外观由矩形改变为椭圆形。

WPF中，控件通常是一个组合FrameworkElement对象。当创建控件模板时，是把多个FrameworkElement对象组合成单个控件。控件模板必须有一个FrameworkElement对象作为根元素，下面包含其他一些其他FrameworkElement对象，组合构成控件的可视化结构。

下面示例展示了控件哎不同的状态下显示不同的外观行为：

|  |
| --- |
| <StackPanel>  <StackPanel.Resources>  <Style TargetType="Button" x:Key="newTemplate">  <!--Set the Background, Foreground, FontSize, Width,  Height, Margin, and Template properties for  the Button.-->  <Setter Property="Background" Value="Navy"/>  <Setter Property="Foreground" Value="White"/>  <Setter Property="FontSize" Value="14"/>  <Setter Property="Width" Value="100"/>  <Setter Property="Height" Value="40"/>  <Setter Property="Margin" Value="10"/>  <Setter Property="HorizontalContentAlignment" Value="Center"/>  <Setter Property="VerticalContentAlignment" Value="Center"/>  <Setter Property="Template">  <Setter.Value>  <ControlTemplate TargetType="Button">  <Border x:Name="RootElement">  <VisualStateManager.VisualStateGroups>  <!--Define the states and transitions for the common states.  The states in the VisualStateGroup are mutually exclusive to  each other.-->  <VisualStateGroup Name="CommonStates">  <!--Define the VisualTransitions that can be used when the control  transitions between VisualStates that are defined in the  VisualStatGroup.-->  <VisualStateGroup.Transitions>  <!--Take one hundredth of a second to transition to the  Pressed state.-->  <VisualTransition To="Pressed"  GeneratedDuration="0:0:0.01" />  <!--Take one half second to trasition to the MouseOver state.-->  <VisualTransition To="MouseOver"  GeneratedDuration="0:0:0.5" />  <!--Take one hundredth of a second to transition from the  Pressed state to the MouseOver state.-->  <VisualTransition From="Pressed" To="MouseOver"  GeneratedDuration="0:0:0.01" />  <!--Take one and a half seconds to transition from the  MouseOver state to the Normal state.  Have the SolidColorBrush, BorderBrush, fade to blue,  then to yellow, and then to black in that time.-->  <VisualTransition From="MouseOver" To="Normal"  GeneratedDuration="0:0:1.5">  <Storyboard>  <ColorAnimationUsingKeyFrames  Storyboard.TargetProperty="Color"  Storyboard.TargetName="BorderBrush"  FillBehavior="HoldEnd" >  <ColorAnimationUsingKeyFrames.KeyFrames>  <LinearColorKeyFrame Value="Blue"  KeyTime="0:0:0.5" />  <LinearColorKeyFrame Value="Yellow"  KeyTime="0:0:1" />  <LinearColorKeyFrame Value="Black"  KeyTime="0:0:1.5" />  </ColorAnimationUsingKeyFrames.KeyFrames>  </ColorAnimationUsingKeyFrames>  </Storyboard>  </VisualTransition>  </VisualStateGroup.Transitions>  <!--The Normal state is the state the button is in  when it is not in another state from this VisualStateGroup.  There is no special visual behavior for this state, but  the VisualState must be defined in order for the button  to return to its initial state.-->  <VisualState x:Name="Normal" />  <!--Change the border of the button to red when the  mouse is over the button.-->  <VisualState x:Name="MouseOver">  <Storyboard>  <ColorAnimation Storyboard.TargetName="BorderBrush"  Storyboard.TargetProperty="Color"  To="Red" />  </Storyboard>  </VisualState>  <!--Change the border of the button to Transparent when the  button is pressed.-->  <VisualState x:Name="Pressed">  <Storyboard >  <ColorAnimation Storyboard.TargetName="BorderBrush"  Storyboard.TargetProperty="Color"  To="Transparent"  />  </Storyboard>  </VisualState>  <!--Show the DisabledRect when the IsEnabled property on  the button is false.-->  <VisualState x:Name="Disabled">  <Storyboard>  <DoubleAnimation Storyboard.TargetName="DisabledRect"  Storyboard.TargetProperty="Opacity"  To="1" Duration="0" />  </Storyboard>  </VisualState>  </VisualStateGroup>  <!--Define the states and transitions for the focus states.  The states in the VisualStateGroup are mutually exclusive to  each other.-->  <VisualStateGroup x:Name="FocusStates">  <!--Define the VisualStates in this VistualStateGroup.-->  <VisualState x:Name="Focused">  <Storyboard>  <ObjectAnimationUsingKeyFrames  Storyboard.TargetName="FocusVisual"  Storyboard.TargetProperty="Visibility" Duration  ="0">    <DiscreteObjectKeyFrame KeyTime="0">  <DiscreteObjectKeyFrame.Value>  <Visibility>Visible</Visibility>  </DiscreteObjectKeyFrame.Value>  </DiscreteObjectKeyFrame>  </ObjectAnimationUsingKeyFrames>  </Storyboard>  </VisualState>  <VisualState x:Name="Unfocused">  <Storyboard>  <ObjectAnimationUsingKeyFrames  Storyboard.TargetName="FocusVisual"  Storyboard.TargetProperty="Visibility"  Duration="0">    <DiscreteObjectKeyFrame KeyTime="0">  <DiscreteObjectKeyFrame.Value>  <Visibility>Collapsed</Visibility>  </DiscreteObjectKeyFrame.Value>  </DiscreteObjectKeyFrame>  </ObjectAnimationUsingKeyFrames>  </Storyboard>  </VisualState>  </VisualStateGroup>  </VisualStateManager.VisualStateGroups>  <!--Create the SolidColorBrush for the Background  as an object elemment and give it a name so  it can be referred to elsewhere in the control template.-->  <Border.Background>  <SolidColorBrush x:Name="BorderBrush" Color="Black"/>  </Border.Background>  <!--Create a border that has a different color by adding smaller grid.  The background of this grid is specified by the button's Background  property.-->  <Grid Background="{TemplateBinding Background}" Margin="4">  <!--Create a Rectangle that indicates that the  Button has focus.-->  <Rectangle Name="FocusVisual"  Visibility="Collapsed" Margin="2"  Stroke="{TemplateBinding Foreground}"  StrokeThickness="1"  StrokeDashArray="1.5 1.5"/>  <!--Use a ContentPresenter to display the Content of  the Button.-->  <ContentPresenter  HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"  VerticalAlignment="{TemplateBinding VerticalContentAlignment}"  Margin="4,5,4,4" />  <!--Create a rectangle that causes the button to appear  grayed out when it is disabled.-->  <Rectangle x:Name="DisabledRect"  Fill="#A5FFFFFF"  Opacity="0" IsHitTestVisible="false" />  </Grid>  </Border>  </ControlTemplate>  </Setter.Value>  </Setter>  </Style>  </StackPanel.Resources>  <Button Style="{StaticResource newTemplate}"  Content="Button1"/>  <Button Style="{StaticResource newTemplate}"  Background="Purple"  Content="Button2" />  </StackPanel> |

本例中，控件模板的根元素是Border，显式命名为“RootElement”；其子元素为Grid；ContentPresenter显示按钮的内容。

比较扩展{TemplateBinding ……}把控件模板元素的属性绑定到控件定义的公共属性。模板绑定时，目标属性和源属性名称可以不同，但必须有相同的类型。

控件模板必须使用属性继承或模板绑定来使用属性值。以下是从Control类继承的可视化属性在模板中的使用方式：

| **属性** | **使用方法** |
| --- | --- |
| [Background](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.background) | 模板绑定 |
| [BorderThickness](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.borderthickness) | 模板绑定 |
| [BorderBrush](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.borderbrush) | 模板绑定 |
| [FontFamily](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.fontfamily) | 属性继承或模板绑定 |
| [FontSize](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.fontsize) | 属性继承或模板绑定 |
| [FontStretch](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.fontstretch) | 属性继承或模板绑定 |
| [FontWeight](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.fontweight) | 属性继承或模板绑定 |
| [Foreground](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.foreground) | 属性继承或模板绑定 |
| [HorizontalContentAlignment](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.horizontalcontentalignment) | 模板绑定 |
| [Padding](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.padding) | 模板绑定 |
| [VerticalContentAlignment](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.control.verticalcontentalignment) | 模板绑定 |

如果控件模板中有ContentPresenter，它将自动绑定到ContentTemplate和Content属性；同样，ItemsPresenter将自动绑定到Items和ItemsPresenter属性。

尽管控件模板不会改变控件的功能，但它可以改变控件的可视化行为。可视化行为描述控件处于某种状态下的外观。可以用VisualState指定控件处于某种状态下的外观。VisualState对象包含在VisualStateGroup对象中，VisualStateGroup添加到VisualStateGroups附加属性中，VisualStateGroups必须位于控件模板的根元素。

每一组VisualStateGroup中的VisualState是互斥的。本例把按钮的状态分为CommonStates和FocusStates两组，按钮的Normal、MouseOver、Pressed、Disabled状态属于CommonStates组，Focused和UnFocused属于FocusStates状态组。

VisualTransition指定了从一个状态到另一个状态平滑转换的动画效果。

* 1. 查找由 ControlTemplate 生成的元素

下面是一个简单的Button类的控件模板：

|  |
| --- |
| <Style TargetType="{x:Type Button}">  <Setter Property="Template">  <Setter.Value>  <ControlTemplate TargetType="{x:Type Button}">  <Grid Margin="5" Name="grid">  <Ellipse Stroke="DarkBlue" StrokeThickness="2">  <Ellipse.Fill>  <RadialGradientBrush Center="0.3,0.2" RadiusX="0.5" RadiusY="0.5">  <GradientStop Color="Azure" Offset="0.1" />  <GradientStop Color="CornflowerBlue" Offset="1.1" />  </RadialGradientBrush>  </Ellipse.Fill>  </Ellipse>  <ContentPresenter Name="content" Margin="10"  HorizontalAlignment="Center" VerticalAlignment="Center"/>  </Grid>  </ControlTemplate>  </Setter.Value>  </Setter>  </Style> |

下面代码演示了如何查找模板内的元素：

|  |
| --- |
| // Finding the grid that is generated by the ControlTemplate of the Button  Grid gridInTemplate = (Grid)myButton1.Template.FindName("grid", myButton1);  // Do something to the ControlTemplate-generated grid  MessageBox.Show("The actual width of the grid in the ControlTemplate: "  + gridInTemplate.GetValue(Grid.ActualWidthProperty).ToString()); |

* 1. 布局模板ItemsPanelTemplate

ItemsPanelTemplate为列表控件的项设置布局。比如ListBox默认纵向显示，使用ItemsPanelTemplate可以改变让它横向显示。

|  |
| --- |
| <Style TargetType="{x:Type ListBox}">  <Setter Property="ItemsPanel">  <Setter.Value>  <ItemsPanelTemplate>  <StackPanel Orientation="Horizontal"  VerticalAlignment="Center"  HorizontalAlignment="Center"/>  </ItemsPanelTemplate>  </Setter.Value>  </Setter>  <Setter Property="Template">  <Setter.Value>  <ControlTemplate TargetType="{x:Type ListBox}">  <Border CornerRadius="5"  Background="{TemplateBinding ListBox.Background}">  <ScrollViewer HorizontalScrollBarVisibility="Auto">  <ItemsPresenter/>  </ScrollViewer>  </Border>  </ControlTemplate>  </Setter.Value>  </Setter>  </Style> |

* 1. 数据模板（DataTemplate）

数据模板用来描述数据的可视化结构，它作用于数据对象，数据以模板指定的呈现方式在控件上显示数据。

DataTemplate可以简单地以内联方式设定控件的内容模板属性，如下所示：

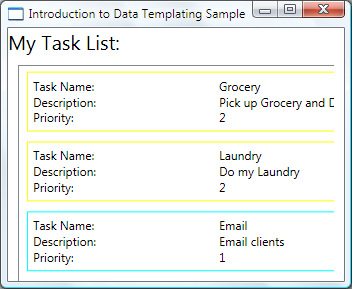
|  |
| --- |
| <ListBox Width="400" Margin="10"  ItemsSource="{Binding Source={StaticResource myTodoList}}">  <ListBox.ItemTemplate>  <DataTemplate>  <StackPanel>  <TextBlock Text="{Binding Path=TaskName}" />  <TextBlock Text="{Binding Path=Description}"/>  <TextBlock Text="{Binding Path=Priority}"/>  </StackPanel>  </DataTemplate>  </ListBox.ItemTemplate>  </ListBox> |

但更多时候是在资源中定义它。本例的基础数据是一个CLR对象的集合，如果绑定到XML，语法稍有不同，要使用Xpath，参见数据绑定部分。

完整的数据模板示例：

|  |
| --- |
| <Window x:Class="DataTemplatingIntro.MainWindow"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"  xmlns:local="clr-namespace:DataTemplatingIntro"  mc:Ignorable="d"  Title="MainWindow" Height="350" Width="525" SizeToContent="WidthAndHeight">  <Window.Resources>  <local:Tasks x:Key="MyTodoList"/>  <local:TaskListDataTemplateSelector x:Key="MyDataTemplateSelector"/>  <DataTemplate x:Key="ImportantTaskTemplate">  <DataTemplate.Resources>  <Style TargetType="TextBlock">  <Setter Property="FontSize" Value="20"/>  </Style>  </DataTemplate.Resources>  <Border Name="border" BorderBrush="Red" BorderThickness="1"  Padding="5" Margin="5">  <DockPanel HorizontalAlignment="Center">  <TextBlock Text="{Binding Path=Description}" />  <TextBlock>!</TextBlock>  </DockPanel>  </Border>  </DataTemplate>  <DataTemplate x:Key="MyTaskTemplate">  <Border Name="border" BorderBrush="Aqua" BorderThickness="1"  Padding="5" Margin="5">  <Grid>  <Grid.RowDefinitions>  <RowDefinition/>  <RowDefinition/>  <RowDefinition/>  </Grid.RowDefinitions>  <Grid.ColumnDefinitions>  <ColumnDefinition />  <ColumnDefinition />  </Grid.ColumnDefinitions>  <TextBlock Grid.Row="0" Grid.Column="0" Text="Task Name:"/>  <TextBlock Grid.Row="0" Grid.Column="1" Text="{Binding Path=TaskName}" />  <TextBlock Grid.Row="1" Grid.Column="0" Text="Description:"/>  <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding Path=Description}"/>  <TextBlock Grid.Row="2" Grid.Column="0" Text="Priority:"/>  <TextBlock Grid.Row="2" Grid.Column="1" Text="{Binding Path=Priority}"/>  </Grid>  </Border>  <DataTemplate.Triggers>  <DataTrigger Binding="{Binding Path=TaskType}">  <DataTrigger.Value>  <local:TaskType>Home</local:TaskType>  </DataTrigger.Value>  <Setter TargetName="border" Property="BorderBrush" Value="Yellow"/>  </DataTrigger>  </DataTemplate.Triggers>  </DataTemplate>  </Window.Resources>  <StackPanel>  <TextBlock FontSize="20" Text="My Task List:"/>  <ListBox Width="400" Margin="10"  ItemsSource="{Binding Source={StaticResource MyTodoList}}"  ItemTemplateSelector="{StaticResource MyDataTemplateSelector}"  HorizontalContentAlignment="Stretch"  IsSynchronizedWithCurrentItem="True"/>  <TextBlock FontSize="20" Text="Information:"/>  <ContentControl Content="{Binding Source={StaticResource MyTodoList}}"  ContentTemplate="{StaticResource MyTaskTemplate}"/>  </StackPanel>  </Window> |

显示效果如下：



HorizontalContentAlignment="Stretch"每项宽度充满整个空间。

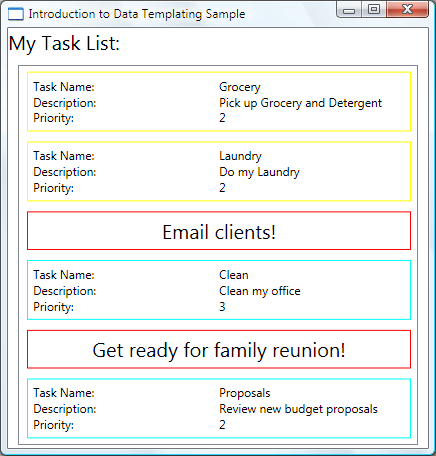
数据模板的触发器DataTrigger把Home类任务边框显示成黄色。

**数据模板选择器（DataTemplateSelector）**：

DataTrigger在数据对象匹配指定的值时，应用设定的属性或执行操作。但对一些复杂的条件匹配就无能为力，缺乏灵活性。DataTemplateSelector可以提供自己的逻辑对数据分类。本例中选择器的逻辑基于任务的Priority：

|  |
| --- |
| using System.Windows;  using System.Windows.Controls;  namespace SDKSample  {  public class TaskListDataTemplateSelector : DataTemplateSelector  {  public override DataTemplate  SelectTemplate(object item, DependencyObject container)  {  FrameworkElement element = container as FrameworkElement;  if (element != null && item != null && item is Task)  {  Task taskitem = item as Task;  if (taskitem.Priority == 1)  return  element.FindResource("importantTaskTemplate") as DataTemplate;  else  return  element.FindResource("myTaskTemplate") as DataTemplate;  }  return null;  }  }  } |

根据Priority选择不同的数据模板，效果如下：



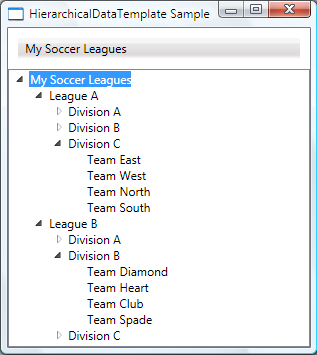
**对层级数据的支持**

前面例子展示对单个数据集合的支持，但实际应用中，我们经常会遇到数据集合是层级的嵌套结构，这时候就需要使用[HierarchicalDataTemplate](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.hierarchicaldatatemplate)，HierarchicalDataTemplate配合 [HeaderedItemsControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.headereditemscontrol)使用，用于诸如TreeView空间、Menu空间这样的树形控件的场景。

|  |
| --- |
| <Window x:Class="SDKSample.Window1"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  Title="HierarchicalDataTemplate Sample"  xmlns:src="clr-namespace:SDKSample">  <DockPanel>  <DockPanel.Resources>  <src:ListLeagueList x:Key="MyList"/>  <HierarchicalDataTemplate DataType = "{x:Type src:League}"  ItemsSource = "{Binding Path=Divisions}">  <TextBlock Text="{Binding Path=Name}"/>  </HierarchicalDataTemplate>  <HierarchicalDataTemplate DataType = "{x:Type src:Division}"  ItemsSource = "{Binding Path=Teams}">  <TextBlock Text="{Binding Path=Name}"/>  </HierarchicalDataTemplate>  <DataTemplate DataType="{x:Type src:Team}">  <TextBlock Text="{Binding Path=Name}"/>  </DataTemplate>  </DockPanel.Resources>  <Menu Name="menu1" DockPanel.Dock="Top" Margin="10,10,10,10">  <MenuItem Header="My Soccer Leagues"  ItemsSource="{Binding Source={StaticResource MyList}}" />  </Menu>  <TreeView>  <TreeViewItem ItemsSource="{Binding Source={StaticResource MyList}}" Header="My Soccer Leagues" />  </TreeView>  </DockPanel>  </Window> |

本例中，ListLeagueList 是 League 对象的列表。 每个 League 对象都有一个 Name 和 Division 对象的集合。 每个 Division 都有一个 Name 和 Team 对象的集合，并且每个 Team 对象都有一个 Name。

显示效果如下：



* 1. 查找由 DataTemplate 生成的元素

一个ListBox：

|  |
| --- |
| <ListBox Name="myListBox" ItemTemplate="{StaticResource myDataTemplate}"  IsSynchronizedWithCurrentItem="True">  <ListBox.ItemsSource>  <Binding Source="{StaticResource InventoryData}" XPath="Books/Book"/>  </ListBox.ItemsSource>  </ListBox> |

数据模板定义：

|  |
| --- |
| <DataTemplate x:Key="myDataTemplate">  <TextBlock Name="textBlock" FontSize="14" Foreground="Blue">  <TextBlock.Text>  <Binding XPath="Title"/>  </TextBlock.Text>  </TextBlock>  </DataTemplate> |

要查找数据模板中的TextBlock元素，因为数据模板是挂接在元素的ContentPresenter下，所以本例中要先获取ListBoxItem，然后在ListBoxItem中查找ContentPresenter，从它的ContentTemplate属性得到DataTemplate，然后调用FindName方法查找TextBlock：

|  |
| --- |
| // Getting the currently selected ListBoxItem  // Note that the ListBox must have  // IsSynchronizedWithCurrentItem set to True for this to work  ListBoxItem myListBoxItem =  (ListBoxItem)(myListBox.ItemContainerGenerator.ContainerFromItem(myListBox.Items.CurrentItem));  // Getting the ContentPresenter of myListBoxItem  ContentPresenter myContentPresenter= FindVisualChild<ContentPresenter>(myListBoxItem);  // Finding textBlock from the DataTemplate that is set on that ContentPresenter  DataTemplate myDataTemplate = myContentPresenter.ContentTemplate;  TextBlock myTextBlock = (TextBlock)myDataTemplate.FindName("textBlock", myContentPresenter);  // Do something to the DataTemplate-generated TextBlock  MessageBox.Show("The text of the TextBlock of the selected list item: "  + myTextBlock.Text); |

辅助方法FindVisualChild如下：

|  |
| --- |
| private childItem FindVisualChild<childItem>(DependencyObject obj)  where childItem : DependencyObject  {  for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)  {  DependencyObject child = VisualTreeHelper.GetChild(obj, i);  if (child != null && child is childItem)  return (childItem)child;  else  {  childItem childOfChild = FindVisualChild<childItem>(child);  if (childOfChild != null)  return childOfChild;  }  }  return null;  } |

1. 页面和导航

页面（Page）是为了导航对内容的封装，可以使用超链接用声明的方式从一个页面导航到另一个页面，也可以使用NavigationService以编程方式实现导航。WPF使用日志（Journal）记载导航的历史踪迹，以支持导航回退。

页面（Page）、页面超链接（PageHyperlink）、导航服务（NavigationService）和日志组成WPF提供的导航支持的核心。

* 1. 实现页面

虽然WPF支持导航到多个内容类型，但最常用最方便的方法还是用Page包装内容，Page实现了导航特定的功能。

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.HomePage">  Hello, from the XBAP HomePage!  </Page> |

隐藏代码（Code-behind）：

|  |
| --- |
| using System.Windows.Controls;  namespace SDKSample  {  public partial class HomePage : Page  {  public HomePage()  {  InitializeComponent();  }  }  } |

在XAML中，Page元素必须包含x:Class属性，其值为代码隐藏的分部类的类名，该类派生自Page类，在构造器中要调用InitializeComponent方法。

* 1. 配置起始页

按照约定，一般WPF应用程序入口时app.xaml，app.xaml的根元素是Application，通过设置StartupUri指定程序启动时加载的页面：

|  |
| --- |
| <Application  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.App"  StartupUri=" HomePage.xaml" /> |

Code-behind：

|  |
| --- |
| using System.Windows;  namespace SDKSample  {  public partial class App : Application { }  } |

* 1. 配置宿主窗口的标题、高度和宽度

设置WindowTitle、WindowWidth和WindowHeight属性：

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.HomePage"  WindowTitle="Page Title"  WindowWidth="500"  WindowHeight="200">  Hello, from the XBAP HomePage!  </Page> |

* 1. 超链接导航

最简单的导航到页面的方法就是在XAML中使用Hyperlink标记：

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  WindowTitle="Page With Hyperlink"  WindowWidth="250"  WindowHeight="250">  <Hyperlink NavigateUri="UriOfPageToNavigateTo.xaml">  Navigate to Another Page  </Hyperlink>  </Page> |

* 1. 片段（Fragment）导航

和Html类似，还可以导航到页面的某个片段：

具有片段的页面：

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  WindowTitle="Page With Fragments" >  <!-- Content Fragment called "Fragment1" -->  <TextBlock Name="Fragment1">  Ea vel dignissim te aliquam facilisis ...  </TextBlock>  </Page> |

片段URI格式：PageURI # ElementName

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  WindowTitle="Page That Navigates To Fragment" >  <Hyperlink NavigateUri="PageWithFragments.xaml#Fragment1">  Navigate To pack Fragment  </Hyperlink>  </Page> |

* 1. 导航服务

真正定位并加载页面的工作是由NavigationService执行的。可以使用NavigationService**以编程的方式实现导航**。

要导航到的页面：

|  |
| --- |
| <Page  x:Class="SDKSample.PageWithNonDefaultConstructor"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  Title="PageWithNonDefaultConstructor">    <!-- Content goes here -->    </Page> |

Code-behind：

|  |
| --- |
| using System.Windows.Controls;  namespace SDKSample  {  public partial class PageWithNonDefaultConstructor : Page  {  public PageWithNonDefaultConstructor(string message)  {  InitializeComponent();  this.Content = message;  }  }  } |

起始页面：

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.NSNavigationPage">  <Hyperlink Click="hyperlink\_Click">  Navigate to Page with Non-Default Constructor  </Hyperlink>  </Page> |

Code-behind：

|  |
| --- |
| using System.Windows;  using System.Windows.Controls;  using System.Windows.Navigation;  namespace SDKSample  {  public partial class NSNavigationPage : Page  {  public NSNavigationPage()  {  InitializeComponent();  }  void hyperlink\_Click(object sender, RoutedEventArgs e)  {  // Instantiate the page to navigate to  PageWithNonDefaultConstructor page = new PageWithNonDefaultConstructor("Hello!");  // Navigate to the page, using the NavigationService  this.NavigationService.Navigate(page);  }  }  } |

**导航到PackUri**

页面

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.NSUriNavigationPage">  <Hyperlink Click="hyperlink\_Click">Navigate to Page by Pack URI</Hyperlink>  </Page> |

代码：

|  |
| --- |
| using System;  using System.Windows;  using System.Windows.Controls;  using System.Windows.Navigation;  namespace SDKSample  {  public partial class NSUriNavigationPage : Page  {  public NSUriNavigationPage()  {  InitializeComponent();  }  void hyperlink\_Click(object sender, RoutedEventArgs e)  {  // Create a pack URI  Uri uri = new Uri("AnotherPage.xaml", UriKind.Relative);  // Get the navigation service that was used to  // navigate to this page, and navigate to  // AnotherPage.xaml  this.NavigationService.Navigate(uri);  }  }  } |

**刷新当前页**

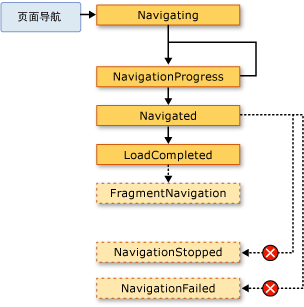
|  |
| --- |
| void hyperlink\_Click(object sender, RoutedEventArgs e)  {  // Force WPF to download this page again  this.NavigationService.Refresh();  }  }  } |

* 1. 导航生存期

可以使用以下NavigationService事件来跟踪和影响导航：

* [Navigating](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.navigating)。 请求新导航时发生。 可用于取消导航。
* [NavigationProgress](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.navigationprogress)。 在下载过程中定期发生，用于提供导航进度信息。
* [Navigated](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.navigated)。 已定位并下载页时发生。
* [NavigationStopped](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.navigationstopped)。 停止导航时发生 (通过调用[StopLoading](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.stoploading))，或当前导航正在进行时在请求新导航。
* [NavigationFailed](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.navigationfailed)。 在导航到所需内容的同时遇到错误时发生。
* [LoadCompleted](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.loadcompleted)。 导航到的内容已加载和分析，并开始呈现时发生。
* [FragmentNavigation](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.navigation.navigationservice.fragmentnavigation)。 导航到内容片段开始时发生，具体如何发生如下所述：
  + 立即，如果所需片段位于当前内容中。
  + 源内容加载之后，如果所需片段在不同内容中。

引发导航的事件顺序如下图：



一般情况下，Page不关心这类事件，这些事件会由Application引发：

* [Application.Navigating](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.navigating)
* [Application.NavigationProgress](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.navigationprogress)
* [Application.Navigated](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.navigated)
* [Application.NavigationFailed](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.navigationfailed)
* [Application.NavigationStopped](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.navigationstopped)
* [Application.LoadCompleted](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.loadcompleted)
* [Application.FragmentNavigation](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.application.fragmentnavigation)

每次NavigationService引发事件时，Application类引发相应事件。 Frame 和NavigationWindow提供相同的事件来检测其各自的范围内的导航。

* 1. 使用日志记住导航历史

可以使用标记方式实现导航后退、前进：

<Hyperlink Command="NavigationCommands.BrowseBack">Back</Hyperlink>

<Hyperlink Command

="NavigationCommands.BrowseForward">Forward</Hyperlink>

也可以使用编程方式，调用NavigationService的GoBack、GoForward、CanGoBack、CanGoForward方法。

* 1. 保留导航历史记录的内容状态

日志可以记住跨页面导航的数据，包括控件数据。日志的每条Page记录扮演了对应Page状态的临时容器的角色。

导航时，发生了以下步骤：

1. 添加当前页面记录到日志后退栈中。
2. 页面状态存储到添加的页面记录中。
3. 导航到新页面。

当页面后退时：

1. 用后退栈顶部页面记录示例化Page。
2. 用保存在日志的页面状态刷新Page。
3. 页面后退导航。

WPF 在页面中使用以下控件时将自动使用这种支持:

* [CheckBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.checkbox)
* [ComboBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.combobox)
* [Expander](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.expander)
* [Frame](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.frame)
* [ListBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listbox)
* [ListBoxItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.listboxitem)
* [MenuItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.menuitem)
* [ProgressBar](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.progressbar)
* [RadioButton](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.radiobutton)
* [Slider](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.slider)
* [TabControl](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabcontrol)
* [TabItem](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.tabitem)
* [TextBox](https://docs.microsoft.com/zh-cn/dotnet/api/system.windows.controls.textbox)

未列出的控件需要自己编码支持。

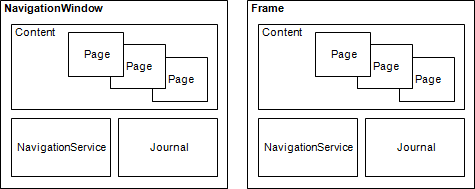
* 1. NavigationWindow和Frame

一般使用NavigationWindow作为应用程序的主窗体。 NavigationWindow和Frame可以通过设置Source属性来导航：

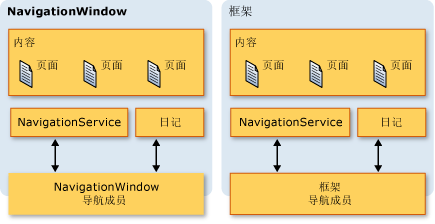
|  |
| --- |
| <NavigationWindow  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  x:Class="SDKSample.MainWindow"  Source="HomePage.xaml"/> |

|  |
| --- |
| <Page  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  WindowTitle="Page that Hosts a Frame"  WindowWidth="250"  WindowHeight="250">  <Frame Source="FramePage1.xaml" />  </Page> |

NavigationWindow和Frame被称为导航宿主类。



除了NavigationService和日志，导航宿主为它们实现了相同的NavigationService实现的成员：



1. 本节内容来自Microsoft Application Architecture Guide 2ndEdition [↑](#footnote-ref-1)