应用程序的三层结构：数据存储层，数据逻辑层和数据展示层。

在WPF中，展示层和逻辑层之间的沟通使用Data Binding。数据驱动UI。

MVVM模式：Model-View-ViewModel

可以把数据绑定看成是数据的桥梁，两端分别是数据源（Source）和目标（Target）。一般情况下（不绝对），Binding源是逻辑层对象，目标是UI控件层对象。

一个简单的数据绑定的例子：

例：程序binding\_test1

Student.cs

// Copyright 2016.刘珅珅

// author：刘珅珅

// 数据绑定：数据源

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*ComponentModel*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace binding\_test1

{

class Student:*INotifyPropertyChanged*

{

// 属性改变事件

public event *PropertyChangedEventHandler* PropertyChanged;

private string name;

public string Name

{

get { return name; }

set

{

name = value;

if (PropertyChanged != null)

{

PropertyChanged.*Invoke*(this, new *PropertyChangedEventArgs*("Name"));

}

}

}

}

}

MainWindow.xaml

<Window x:Class="binding\_test1.MainWindow"

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

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

Title="MainWindow" Height="350" Width="525">

<StackPanel>

<TextBox x:Name="textBoxName" BorderBrush="Black" Margin="5" />

<Button Content="Add Age" Margin="5" Click="Button\_Click" />

</StackPanel>

</Window>

MainWindow.xaml.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 binding\_test1

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

Student dataSource;

public MainWindow()

{

*InitializeComponent*();

// 准备数据源

dataSource = new Student();

// 准备绑定

*Binding* binding = new *Binding*();

binding.*Source* = dataSource;

// 绑定到数据源的哪个属性

binding.*Path* = new *PropertyPath*("Name");

// 连接数据源和目标

*BindingOperations*.*SetBinding*(textBoxName, *TextBox*.*TextProperty*, binding);

}

private void Button\_Click(object sender, *RoutedEventArgs* e)

{

dataSource.Name += "Name";

}

}

}

在本例中，Binding源的对象具有自动通知Binding自己的属性值已经变化的能力，因为其实现了INotifyPropertyChanged接口并在属性的set语句中激发PropertyChanged事件。

Binding的源与路径

例：程序binding\_test2

MainWindow.xaml

<Window x:Class="binding\_test2.MainWindow"

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

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

Title="MainWindow" Height="350" Width="525">

<StackPanel>

<TextBox x:Name="textBox" Text="{Binding Path=Value, ElementName=slider}" BorderBrush="Black" Margin="5" />

<Slider x:Name="slider" Maximum="100" Minimum="0" Margin="5" />

</StackPanel>

</Window>

MainWindow.xaml.cs

// Copyright 2016.刘珅珅

// author：刘珅珅

// 控件作为Binding源

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 binding\_test2

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

public MainWindow()

{

*InitializeComponent*();

}

}

}

没有“Path”的Binding：Binding源本身就是数据且不需要Path指明

例：程序binding\_test3

MainWindow.xaml

<Window x:Class="binding\_test3.MainWindow"

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

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

xmlns:sys="clr-namespace:System;assembly=mscorlib"

Title="MainWindow" Height="100" Width="220">

<StackPanel>

<StackPanel.Resources>

<sys:String x:Key="myString">

菩提本无树，明镜亦非台。

本来无一物，何处惹尘埃

</sys:String>

</StackPanel.Resources>

<TextBlock x:Name="textBlock" TextWrapping="Wrap"

Text="{Binding Source={StaticResource ResourceKey=myString}}" FontSize="16"

Margin="5" />

</StackPanel>

</Window>

只有一个对象包含数据并能通过属性把数据暴露出来，就能当作Binding的源来使用。

为Binding指定源（Source）的方法：

1. 把普通CLR类型单个对象指定为Source：包括.NET Framework自带类型的对象和用户自定义类型的对象。如果类型实现了INotifyPropertyChanged接口，则可通过在属性的set语句里激发PropertyChanged事件来通知Binding数据已更新。
2. 把普通CLR集合类型对象指定为Source：包括数组、List<T>，ObservableCollection<T>等集合类型。实际工作中，我们经常需要把一个集合作为ItemsControl派生类的数据源来使用，一般把控件的ItemsSource属性使用Binding关联到一个集合对象上。

例：程序binding\_test5

Student.cs

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace binding\_test5

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

}

}

MainWindow.xaml

<Window x:Class="binding\_test5.MainWindow"

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

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

Title="MainWindow" Height="260" Width="300">

<StackPanel x:Name="stackPanel" Background="LightBlue">

<TextBlock Text="Student ID:" FontWeight="Bold" Margin="5" />

<TextBox x:Name="textBoxId" Margin="5" />

<TextBlock Text="Student List" FontWeight="Bold" Margin="5" />

<ListBox x:Name="listBoxStudents" Height="120" Margin="5">

<ListBox.ItemTemplate>

<DataTemplate>

<StackPanel Orientation="Horizontal">

<TextBlock Text="{Binding Path=Id}" Width="30" />

<TextBlock Text="{Binding Path=Name}" Width="60" />

<TextBlock Text="{Binding Path=Age}" Width="30" />

</StackPanel>

</DataTemplate>

</ListBox.ItemTemplate>

</ListBox>

</StackPanel>

</Window>

MainWindow.xaml.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 binding\_test5

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

public MainWindow()

{

*InitializeComponent*();

// 准备数据源

*List*<Student> studentList = new *List*<Student>()

{

new Student() {Id = 0, Name = "Tim", Age = 29},

new Student() {Id = 1, Name = "Tom", Age = 28},

new Student() {Id = 2, Name = "Kyle", Age = 27},

new Student() {Id = 3, Name = "Tony", Age = 26},

new Student() {Id = 4, Name = "Vina", Age = 25},

new Student() {Id = 5, Name = "Mike", Age = 24}

};

// 为ListBox设置Binding

listBoxStudents.*ItemsSource* = studentList;

// 为ID的TextBox设置Binding

*Binding* binding = new *Binding*("SelectedItem.Id") { *Source* = listBoxStudents };

textBoxId.*SetBinding*(*TextBox*.*TextProperty*, binding);

}

}

}

1. 把ADO.NET数据对象指定为Source：包括DataTable和DataView等对象。
2. 使用XmlDataProvider把XML数据指定为Source：一些WPF控件如TreeView和Menu，可以把树状结构的XML数据作为源。

例：程序binding\_test6

MainWindow.xaml

<Window x:Class="binding\_test6.MainWindow"

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

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

Title="MainWindow" Height="205" Width="240">

<StackPanel Background="LightBlue">

<ListView x:Name="listViewStudents" Height="130" Margin="5">

<ListView.View>

<GridView>

<GridViewColumn Header="Id" Width="80"

DisplayMemberBinding="{Binding XPath=@Id}" />

<GridViewColumn Header="Name" Width="120"

DisplayMemberBinding="{Binding XPath=Name}" />

</GridView>

</ListView.View>

</ListView>

<Button Content="Load" Click="Button\_Click" Height="25" Margin="5,0" />

</StackPanel>

</Window>

MainWindow.xaml.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*;

using *System*.*Xml*;

namespace binding\_test6

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

public MainWindow()

{

*InitializeComponent*();

}

private void Button\_Click(object sender, *RoutedEventArgs* e)

{

*XmlDocument* doc = new *XmlDocument*();

doc.*Load*(@"D:\study\CSharp-study\binding\_test6\RawData.xml");

*XmlDataProvider* xdp = new *XmlDataProvider*();

xdp.*Document* = doc;

// 使用XPath选择需要暴露的数据

// 现在是需要暴露一组Student

xdp.*XPath* = @"/StudentList/Student";

listViewStudents.*DataContext* = xdp;

// 绑定没有指定源，使用DataContext作为源

listViewStudents.*SetBinding*(*ListView*.*ItemsSourceProperty*, new *Binding*());

}

}

}

TreeView结构的xml文档

例：程序binding\_test7

MainWindow.xaml

<Window x:Class="binding\_test7.MainWindow"

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

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

Title="MainWindow" Height="210" Width="260">

<Window.Resources>

<XmlDataProvider x:Key="xdp" XPath="FileSystem/Folder">

<x:XData>

<FileSystem xmlns="">

<Folder Name="Books">

<Folder Name="Programming">

<Folder Name="Windows">

<Folder Name="WPF" />

<Folder Name="MFC" />

<Folder Name="Delphi" />

</Folder>

</Folder>

</Folder>

</FileSystem>

</x:XData>

</XmlDataProvider>

</Window.Resources>

<Grid>

<TreeView ItemsSource="{Binding Source={StaticResource xdp}}">

<TreeView.ItemTemplate>

<HierarchicalDataTemplate ItemsSource="{Binding XPath=Folder}">

<TextBlock Text="{Binding XPath=@Name}" />

</HierarchicalDataTemplate>

</TreeView.ItemTemplate>

</TreeView>

</Grid>

</Window>

1. 把依赖对象（Dependency Object）指定为Source：依赖对象不仅可以作为Binding的目标，同时也可以作为Binding的源。这样可以形成Binding链。依赖对象中的依赖属性可以作为Binding的Path。
2. 把容器的DataContext（WPF控件都具备这个属性）指定为Source（WPF Data Binding的默认行为）：当我们明确知道将从哪个属性获取数据，但具体把哪个对象作为Binding源不能确定。这时，我们可以建立一个Binding，只设置Path，不设置Source，让这个Binding自己去寻找Source，Binding会自动把控件的DataContext当作Source（它会沿着控件树一层一层向外找，直到找到带有Path指定属性的对象为止）。

例：程序binding\_test4

Student.cs

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace binding\_test4

{

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int Age { get; set; }

}

}

MainWindow.xaml

<Window x:Class="binding\_test4.MainWindow"

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

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

xmlns:local="clr-namespace:binding\_test4"

Title="MainWindow" Height="135" Width="300">

<StackPanel Background="LightBlue">

<StackPanel.DataContext>

<local:Student Id="6" Age="29" Name="Tim" />

</StackPanel.DataContext>

<Grid>

<StackPanel>

<TextBox Text="{Binding Path=Id}" Margin="5" />

<TextBox Text="{Binding Path=Name}" Margin="5" />

<TextBox Text="{Binding Path=Age}" Margin="5" />

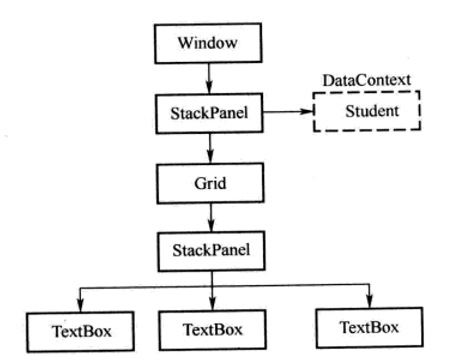
</StackPanel>

</Grid>

</StackPanel>

</Window>

本例中的UI的布局如图所示：



3个TextBox的Binding就会自动向UI元素树的上层去寻找可用的DataContext。

1. 通过ElementName指定Source：XAML无法访问对象，只能使用对象的Name属性来找到对象。
2. 通过Binding的RelativeSource属性相对地指定Source：当控件需要关注自己的、自己容器的或者自己内部元素的某个值就需要使用这种办法。
3. 把ObjectDataProvider对象指定为Source：当数据源的数据不是通过属性而是通过方法暴露给外界的时候，我们可以使用这两种对象来包装数据源再指定为Source。

使用ObjectDataProvider包装对象

例：程序bindint\_test8

MainWindow.xaml

<Window x:Class="binding\_test8.MainWindow"

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

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

Title="MainWindow" Height="135" Width="300">

<StackPanel Background="LightBlue">

<TextBox x:Name="textBoxArg1" Margin="5" />

<TextBox x:Name="textBoxArg2" Margin="5" />

<TextBox x:Name="textBoxResult" Margin="5" />

</StackPanel>

</Window>

Calculator.cs

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Linq*;

using *System*.*Text*;

using *System*.*Threading*.*Tasks*;

namespace binding\_test8

{

class Calculator

{

public string Add(string arg1, string arg2)

{

double x = 0;

double y = 0;

double z = 0;

if (double.*TryParse*(arg1, out x) && double.*TryParse*(arg2, out y))

{

z = x + y;

return z.*ToString*();

}

return "Input Error";

}

}

}

MainWindow.xaml.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 binding\_test8

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

public MainWindow()

{

*InitializeComponent*();

SetBinding();

}

private void SetBinding()

{

// 创建并配置ObjectDataProvider对象

*ObjectDataProvider* odp = new *ObjectDataProvider*();

// 被包装对象

odp.*ObjectInstance* = new Calculator();

odp.*MethodName* = "Add";

odp.*MethodParameters*.Add("0");

odp.*MethodParameters*.Add("0");

// 以ObjectDataProvider对象为Source创建Binding

*Binding* bindingToArg1 = new *Binding*("MethodParameters[0]")

{

*Source* = odp,

*BindsDirectlyToSource* = true,

*UpdateSourceTrigger* = *UpdateSourceTrigger*.*PropertyChanged*

};

*Binding* bindingToArg2 = new *Binding*("MethodParameters[1]")

{

*Source* = odp,

// 把从UI元素收集到的数据写入其直接Source（即ObjectDataProvider），

// 而不是被ObjectDataProvider包装着的Calculator对象

*BindsDirectlyToSource* = true,

*UpdateSourceTrigger* = *UpdateSourceTrigger*.*PropertyChanged*

};

// 数据源本身代表数据，使用"."作为Path

*Binding* bindingToResult = new *Binding*(".") { *Source* = odp };

// 将Binding关联到UI元素

textBoxArg1.*SetBinding*(*TextBox*.*TextProperty*, bindingToArg1);

textBoxArg2.*SetBinding*(*TextBox*.*TextProperty*, bindingToArg2);

textBoxResult.*SetBinding*(*TextBox*.*TextProperty*, bindingToResult);

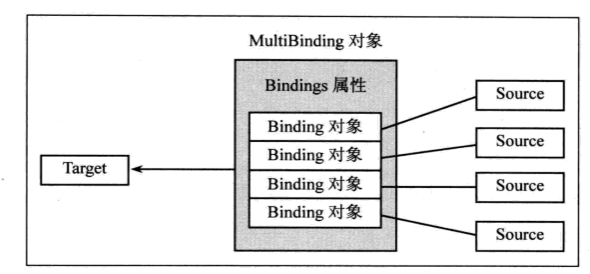
}

}

}

1. 把使用LINQ检索得到的数据对象作为Binding的源。

多路绑定：MultiBinding



例：程序binding\_test9

第一、二个TextBox输入用户名，要求内容一致。

第三、四个TextBox输入E-mail，要求内容一致。

当TextBox的全部符合要求的时候，Button可用。

MainWindow.xaml

<Window x:Class="binding\_test9.MainWindow"

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

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

Title="MainWindow" Height="185" Width="300">

<StackPanel Background="LightBlue">

<TextBox x:Name="textBox1" Height="23" Margin="5" />

<TextBox x:Name="textBox2" Height="23" Margin="5,0" />

<TextBox x:Name="textBox3" Height="23" Margin="5" />

<TextBox x:Name="textBox4" Height="23" Margin="5,0" />

<Button x:Name="button" Content="Submit" Width="80" Margin="5" />

</StackPanel>

</Window>

LogonMultiBindingConver.cs

using *System*;

using *System*.*Collections*.*Generic*;

using *System*.*Globalization*;

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 binding\_test9

{

class LogonMultiBindingConvert:*IMultiValueConverter*

{

public object Convert(object[] values, *Type* targetType, object parameter, *CultureInfo* culture)

{

if (!values.*Cast*<string>().*Any*(text => string.*IsNullOrEmpty*(text))

&& values[0].*ToString*() == values[1].*ToString*()

&& values[2].*ToString*() == values[3].*ToString*())

{

return true;

}

return false;

}

public object[] ConvertBack(object value, *Type*[] targetType, object parameter, *CultureInfo* culture)

{

throw new *NotImplementedException*();

}

}

}

MainWindow.xaml.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 binding\_test9

{

/// <summary>

/// MainWindow.xaml 的交互逻辑

/// </summary>

public partial class MainWindow : *Window*

{

public MainWindow()

{

*InitializeComponent*();

SetMultiBinding();

}

private void SetMultiBinding()

{

// 准备基础Binding

*Binding* b1 = new *Binding*("Text") { *Source* = textBox1 };

*Binding* b2 = new *Binding*("Text") { *Source* = textBox2 };

*Binding* b3 = new *Binding*("Text") { *Source* = textBox3 };

*Binding* b4 = new *Binding*("Text") { *Source* = textBox4 };

// 准备MultiBinding

*MultiBinding* mb = new *MultiBinding*() { *Mode* = *BindingMode*.*OneWay* };

// MultiBinding对Add子Binding的顺序是敏感的

mb.*Bindings*.*Add*(b1);

mb.*Bindings*.*Add*(b2);

mb.*Bindings*.*Add*(b3);

mb.*Bindings*.*Add*(b4);

// 数据转换

mb.*Converter* = new LogonMultiBindingConvert();

// 将Button与MultiBinding对象关联

button.*SetBinding*(*Button*.*IsEnabledProperty*, mb);

}

}

}