## 实现具有通知外界自身属性值已变化能力的数据源

不具有通知能力的数据源

class Student

{

public string Name { get; set; }

}

Student stu = new Student(); stu.Name = "毛姝弋"，外界无法得知Name属性的值已发生了变化。

**实现思路**

属性值发生变化的对象是被观察者(ObservableObject)，属性值发生变化是事件(PropertyChangedEvent)，对属性值变化感兴趣的对象是观察者，这显然是一个观察者模式。被观察者提供事件供观察者注册，事件参数(EventArgs)包含发生变化的属性名称和被观察者本身(Sender)，那么观察者的事件处理程序就能获知哪个对象的哪个属性变化成了什么值。

微软实现了System.ComponentModel.INotifyPropertyChanged，只要类继承了此接口便拥有事件PropertyChangedEventHandler，在类想通知外界自身属性发生变化时，调用此事件即可。

**具备通知外界自身属性值已变化能力的数据源**

class Student : INotifyPropertyChanged

{

public string Name { get; set; }

public event PropertyChangedEventHandler PropertyChanged;

protected void RaisePropertyChanged(string propertyName)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

当Student想通知外界自身属性发生变化时，调用RaisePropertyChanged(string propertyName)即可，参数propertyName是想要通知外界值已变化的属性的名称。根据C#事件定义，只能由发布者发布事件，也就是说，事件只能在类内部调用，所以，RaisePropertyChanged(string propertyName)的访问权限必须是protected，以保证RaisePropertyChanged(string propertyName)只能在Student的成员属性或成员方法内被调用。

**常见应用场景**

public static void Main(string[] args)

{

Student stu = new Student() { Name = "Donald Trump" };

Teacher teacher = new Teacher();

stu.PropertyChanged += teacher.HandleStudentPropertyChanged;

for (int i = 0; i < 3; i++)

{

stu.Name = "Elon Musk";

Thread.Sleep(1000);

}

}

class Student : INotifyPropertyChanged

{

private string \_name;

public string Name

{

get { return \_name; }

set

{

\_name = value;

RaisePropertyChanged("Name");

}

}

public event PropertyChangedEventHandler PropertyChanged;

protected void RaisePropertyChanged(string propertyName)

{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

class Teacher

{

public void HandleStudentPropertyChanged(object sender, PropertyChangedEventArgs e)

{

Type type = sender.GetType();

object value = type.GetProperty(e.PropertyName).GetValue(sender);

Console.WriteLine($"{sender}的{e.PropertyName}值变化成{value}.");

}

}

控制台打印

Student的Name值变化成Elon Musk.

Student的Name值变化成Elon Musk.

Student的Name值变化成Elon Musk.

结论：

属性通过setter修改值，故，RaisePropertyChanged(string propertyName)几乎都是紧跟在=value之后，在属性值改变后发布属性值被修改事件。

**数据源可随时发布属性变化通知**

<StackPanel DataContext="{x:Static local:MainWindow.Student}">

<StackPanel Margin="5" Orientation="Horizontal">

<TextBlock Text="Full Name : " />

<TextBlock Text="{Binding FullName}" />

</StackPanel>

<StackPanel Margin="5" Orientation="Horizontal">

<TextBlock Text="First Name : " />

<TextBlock Text="{Binding FirstName}" />

</StackPanel>

<StackPanel Margin="5" Orientation="Horizontal">

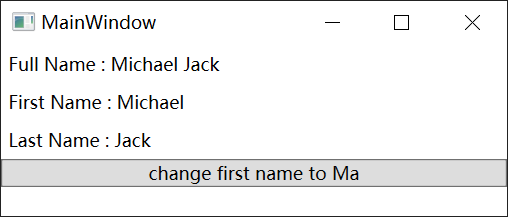
<TextBlock Text="Last Name : " />

<TextBlock Text="{Binding LastName}" />

</StackPanel>

<Button Click="Button\_Click" Content="change first name to Ma" />

</StackPanel>



public partial class MainWindow : Window

{

public static Student Student { get; set; } = new Student()

{

FirstName = "Michael",

LastName = "Jack"

};

public MainWindow()

{

InitializeComponent();

}

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

{

Student.FirstName = "Ma";

}

}

public class Student : INotifyPropertyChanged

{

public string FullName

{

get => FirstName + " " + LastName;

}

private string \_firstName;

public string FirstName

{

get { return \_firstName; }

set

{

\_firstName = value;

RaisePropertyChanged("FirstName");

RaisePropertyChanged("FullName");

}

}

private string \_lastName;

public string LastName

{

get { return \_lastName; }

set

{

\_lastName = value;

RaisePropertyChanged("LastName");

RaisePropertyChanged("FullName");

}

}

public event PropertyChangedEventHandler PropertyChanged;

protected void RaisePropertyChanged(string propertyName)

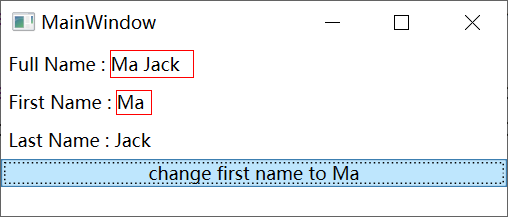
{

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

}

点击按钮后



结论：

属性Full Name的更新通知并不是在自己的setter中发布的，而是放在First Name和Last Name的setter中。改变Fist Name，Full Name在UI的显示也同步更新，所以，***无论属性的值是否真的发生了变化，被观察者都可随时调用RaisePropertyChanged(string propertyName)向外界发布某个属性变化事件***。

***数据源为什么通过接口添加属性变化事件***

WPF通过Binding将数据源和目标控件绑定在一起，目标控件监听数据源属性的值是否发生变化，这显然存在一个目标控件注册数据源属性值变化的事件处理程序操作。Binding检测到数据源有实现INotifyPropertyChanged接口，便会将控件的事件处理程序注册到数据源事件，目标控件是订阅者，数据源是发布者。

## 源属性刷新目标属性

Binding binding = new Binding();

binding.Source = source;

binding.Path = new PropertyPath("A.B.C");

语法规则：

1. 目标属性关联源属性C，C是B的属性，B是A的属性，A是source的属性
2. 但不要求source必须继承INotifyPropertyChanged，因为任何对象都可以作为数据源

**发生以下情况会导致目标属性被刷新：**

情况一：重置Source。

情况二：属性链上被重置值的节点属性的setter调用了PropertyChanged?.Invoke()，即可自动刷新UI，即更新目标属性如

* 重置C

1. B继承了INotifyPropertyChanged

2. C的Setter调用PropertyChanged?.Invoke(this,new PropertyChangedArgs("C"));

* 重置B

1. A继承了INotifyPropertyChanged

2. B的Setter调用PropertyChanged?.Invoke(this,new PropertyChangedArgs("B"));

* 重置A

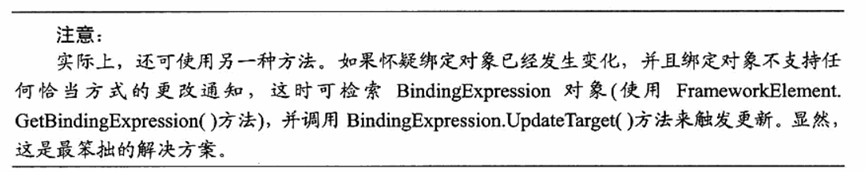
1. source继承了INotifyPropertyChanged

2. A的Setter调用PropertyChanged?.Invoke(this,new PropertyChangedArgs("A"));

换言之，实现通知机制的属性以及它的子属性都可以实现双向通知的绑定，若子属性作为末节点属性，重新赋值属性可通知UI，但重新赋值子属性无法刷新UI；若属性作为末节点属性，重新赋值属性即可刷新UI.

情况三：

利用BindingExpression强制刷新UI，此方法与直接拿到控件赋值(button.Background = new SolidColorBrush(Colors.Yellow);)的区别是，一个明确知道赋予什么值，另外一个驱动Binding管道从后台拿出应该赋予的值。



public BindingExpression GetBindingExpression(DependencyProperty dp);

public override void UpdateTarget();

Code Sample

[link-files\202205\后台刷新前端的条件](link-files/202205/后台刷新前端的条件)