[未完成的 1](#_Toc96301027)

[第二部分：布局 1](#_Toc96301028)

[Canvas 1](#_Toc96301029)

[Canvas的尺寸 1](#_Toc96301030)

[Canvas子元素的位置和尺寸 1](#_Toc96301031)

[Canvas.ZIndex 1](#_Toc96301032)

[第三部分：数据绑定 1](#_Toc96301033)

[Binding的优势 1](#_Toc96301034)

[Bidning的重要概念 2](#_Toc96301035)

[源属性能够刷新依赖属性的必备条件 2](#_Toc96301036)

[第一种情况：BindingMode.OneTime 2](#_Toc96301037)

[第二种情况：BindingMode.OneWayToSource 2](#_Toc96301038)

[第三种情况：BindingMode.OneWay或Binding.TwoWay 2](#_Toc96301039)

[依赖属性能够刷新源属性的必备条件 3](#_Toc96301040)

[第一种情况：BindingMode.OneTime 3](#_Toc96301041)

[第二种情况: BindingMode.OneWay 3](#_Toc96301042)

[第三种情况:BindingMode.OneWayToSource或BindingMode.TwoWay 3](#_Toc96301043)

[实现一个具有通知能力的数据源对象 3](#_Toc96301044)

[System.CompoentModel.InotifyPropertyChanged 3](#_Toc96301045)

[深入分析PropertyChanged 3](#_Toc96301046)

[NotificationObject 4](#_Toc96301047)

[进阶分析题 4](#_Toc96301048)

[Binding的语法 4](#_Toc96301049)

[绑定的步骤 4](#_Toc96301050)

[XAML实现Binding 4](#_Toc96301051)

[C#实现Binding 4](#_Toc96301052)

[同一依赖对象的不同属性互相绑定 4](#_Toc96301053)

[Source 4](#_Toc96301054)

[Path 5](#_Toc96301055)

[基础概念 5](#_Toc96301056)

[实现具有通知能力的索引器 5](#_Toc96301057)

[多级属性 5](#_Toc96301058)

[索引器 5](#_Toc96301059)

[附加属性 6](#_Toc96301060)

[依赖属性与数据源本身相关联而不是与数据源属性相关联 6](#_Toc96301061)

[TargetNullValue和FallbackValue 6](#_Toc96301062)

[TargetNullValue 6](#_Toc96301063)

[FallBackValue 6](#_Toc96301064)

[经典示例 7](#_Toc96301065)

[值转换器 7](#_Toc96301066)

[Binding.StringFormat 7](#_Toc96301067)

[值转换器 8](#_Toc96301068)

[值转换器和数据触发器的比较 9](#_Toc96301069)

[通过ConverterParameter或自定义属性为值转换器传参 10](#_Toc96301070)

[ConverterParameter 10](#_Toc96301071)

[自定义属性 10](#_Toc96301072)

[支持MarkupExtension的值转换器 10](#_Toc96301073)

[代码示例 10](#_Toc96301074)

[二者比较 11](#_Toc96301075)

[绑定的方向 11](#_Toc96301076)

[依赖属性刷新源属性的时机 11](#_Toc96301077)

[Delay 11](#_Toc96301078)

[Explicit 11](#_Toc96301079)

[IsAsync 12](#_Toc96301080)

[验证和报警 12](#_Toc96301081)

[MultiBinding 17](#_Toc96301082)

[基础概念 17](#_Toc96301083)

[绑定步骤 18](#_Toc96301084)

[例子1：显示人的全名 18](#_Toc96301085)

[例子2：账户和密码框有内容时登录按钮才能响应 18](#_Toc96301086)

[通过BindingOperations操作依赖属性和源属性 19](#_Toc96301087)

[清除依赖属性上的Binding 19](#_Toc96301088)

[BindingExpression的强大功能 19](#_Toc96301089)

[未完成的 21](#_Toc96301090)

# 未完成的

数据源Source专题

BindingGroup

INotifyDataErrorInfo

# 第一部分：WPF App和XAML

# 第二部分：布局

所有的父容器都有Panel.Zindex，来设置子元素的顶级或底级。

子元素尺寸超过容器的大小，显示的时候会被截断，但是其实际大小还是那么大，不会变。

## Canvas

### Canvas的尺寸

Canvas的大小与其子元素完全无关，仅完全取决于

1. 如果未给Canvas显示设置尺寸，则Canvas默认充满其父容器，父容器大小变化，Canvas大小也跟着变化。
2. 如果显示为Canvas设置了尺寸,则Canvas的尺寸便是设置的尺寸，如超过父容器，则会伸展出父容器的范围，覆盖父容器周边的元素，除非设置父容器ClipToBounds = true。
3. 自身的HorizontalAlignment，VerticalAlignment，父容器的HorizontalContentAlignment，VerticalContentAlignment，Canvas的尺寸会为0
4. Canvas的尺寸与子元素的尺寸，数量完全无关，不会受其影响。

### Canvas子元素的位置和尺寸

Canvas画布，没有自己的布局逻辑，其子元素的位置和大小完全程序员自己指定。

位置由附加属性Canvas.Top,Canvas.Left或Canvas.Bottom,Canvas.Right决定；

若未给子元素显示设置Height和Width，子元素大小为0或刚好容纳其内容；若给子元素设置了Height和Width，则其实际尺寸便为设置的尺寸。

Canvas的子元素的尺寸若超过Canvas自身的尺寸，子元素并不会被截断，而是伸出Canvas范围，覆盖到Canvas的周边元素。我们可以设置Canvas.ClipToBounds = true来截断。

Canvas的子元素的尺寸一旦确定，并不会随Canvas尺寸的变化而变化。

### Canvas.ZIndex

Canvas的子元素位置和尺寸在Canvas是自由的，存在相互覆盖的情况。

有两种方法设置哪个子元素在顶层或底层，

1. Opacity透明度 0 - 1和Visibility可见性 true或false
2. 设置子元素的附加属性Panel.Zindex，值大的在顶层

# 第三部分：依赖属性和附加属性

MSDN中有一句话：One of the primary architectural philosophies used in building WPF was a preference for properties over methods or Depenevents.

这句话的意思就是WPF的主要设计思想之一是侧重属性远胜于方法和事件，能用属性解决问题，坚决不用方法和事件。

## WPF依赖属性

两类一树：DependencyProperty & DependencyObject；LogicTree（逻辑树）。

WPF的UI元素是一个很特殊的类，这种类继承DependencyObject，称之为依赖对象。这种类的属性采用DependencyProperty抽象，称之为依赖属性。依赖对象的每一个依赖属性都是一个单实例，依赖属性功能比普通.NET属性强大的多，有专门的读写方法，读写方法是继承自DependencyObject的

public void SetValue(DependencyProperty dp, object value);

public object GetValue(DependencyProperty dp);

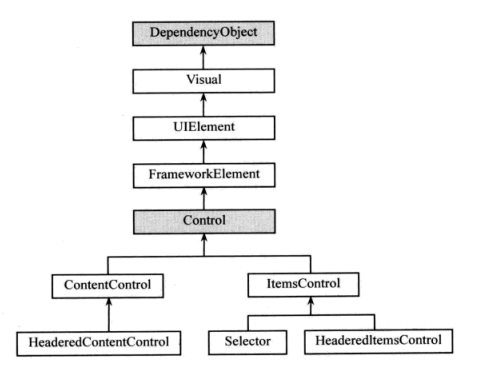
面对一颗UI树，可以认为一堆相同类型的依赖对象从各种单实例依赖属性获取值来展示自己。

## WPF的依赖属性与传统的.NET属性有何不同

1. 依赖属性的值不是每个依赖对象对应一个字段，从字段内存中读取，而是所有的同一个类型的依赖对象从同一个单实例依赖属性实时计算得到（支持绑定，默认值，属性值继承，资源，样式，动画，更改通知等）（DependencyProperty），这完全是以时间换内存的做法。

2. WPF提供了专门的读写依赖属性的方法（DependencyObject）

3. 实现了依赖属性与数据类型（宿主）的解耦，可以动态的为对象扩展(增加)依赖属性（Attached Property）



## 依赖属性的实现原理

定义一个属性

1. 必备的信息：类型，默认值，宿主

2. 非必备信息：合法性检查，属性值改变时触发事件（封装）

CLR

CLR定义属性的方法：

class MyButton

{

public Brush Background { get; set; } = new SolidColorBrush(Colors.Transparent);

}

CLR读写属性的方法：

MyButton myButton = new MyButton();

// 读

Console.WriteLine($"按钮的背景色:{myButton.Background}");

// 写

myButton.Background = new SolidColorBrush(Colors.Red);

WPF有个叫DependencyProperty的类，它用以抽象依赖属性。**为依赖对象抽象类定义一个依赖属性，即是用一个DependencyProperty实例表示一个依赖属性的过程；依赖属性“背后”的类型，默认值，宿主，合法性检验，属性值改变时触发事件都是DependencyProperty的构造参数。**  
**模型：DependencyProperty实例 <=> 类的依赖属性**  
针对类的某个依赖属性DP，该类的所有实例读写依赖属性DP，只能用专门的实例方法读写【表示依赖属性DP的那个DependencyProperty实例】，这也是拥有依赖属性的类必须要继承DependencyObject的原因。

class MyButton : DependencyObject

{

// Using a DependencyProperty as the backing store for Brush. This enables animation, styling, binding, etc...

public static readonly DependencyProperty BrushProperty =

DependencyProperty.Register("Brush", typeof(Brush), typeof(MyButton), new PropertyMetadata(new SolidColorBrush(Colors.Transparent)));

public static void Main()

{

MyButton myButton = new MyButton();

// 读

myButton.GetValue(MyButton.BrushProperty);

// 写

myButton.SetValue(MyButton.BrushProperty, new SolidColorBrush(Colors.Red));

}

}

为什么BrushProperty是static？

因为所有实例读取属性时，需要的是同一个DependencyProperty实例。假设我们不搞成static，每次实例化都会Register一次，返回的不是同一个DP实例。即使Register内部实现了只会注册和实例化一次，那么static更合理，保证整个应用程序只有一个栈中的引用指向DP实例，通过类名即可拿到DP实例。

### 定义依赖属性

public static DependencyProperty Register(string name, Type propertyType, Type ownerType, PropertyMetadata typeMetadata, ValidateValueCallback validateValueCallback);

* name CLR属性包装器的名称
* propertyType 此依赖属性的“背后”类型
* ownerType 拥有此依赖属性的类型
* validateValueCallback 回调函数，为依赖属性赋值时，会调用此函数检验新值的合法性
* PropertyMetadata是FrameworkPropertyMetadata的基类，FrameworkPropertyMetadata的一些关键成员如下：
* DefaultValue 属性的默认值
* AffectsArrange,AffectsMeasure,AffectsParentArrange,AffectsParentMeasure 默认值为false，设置为true时，属性值变化，会重新布局该元素。
* AffectsRender 默认值为false，设置为true时，属性值变化，会重新绘制该元素。
* Inherits 默认值为false，设置为true时，此属性能支持UI树继承。
* CoerceValueCallback 回调函数，在验证依赖属性之前尝试纠正属性值。返回DependencyProperty.Unset表示拒绝更新值。
* PropertyChangedCallback,回调函数属性值发生变化时被调用.
* BindsTwoWayByDefault，如果设置成True，进行数据绑定时默认Mode是双向绑定而不是单向绑定，但是可以在创建绑定时显示设置单双向。

### CoerceValueCallback，ValidateValueCallback PropertyChangedCallback

public delegate object CoerceValueCallback(DependencyObject d, object baseValue);

public delegate bool ValidateValueCallback(object value);

public delegate void PropertyChangedCallback(DependencyObject d, DependencyPropertyChangedEventArgs e);

当为依赖属性赋值时，首先，执行CoerceValueCallback，主要目的是矫正属性值，返回一个Object,可以返回DependencyProperty.UnsetValue拒绝本次赋值。然后，执行ValidateValueCallback，true则接受本次赋值，false，则拒绝本次赋值。最后，调用PropertyChangedCallback。

**ValidateValueCallback和CoerceValueCallback的区别和关系**

CoerceValueCallback和ValidateValueCallback都是用于校验属性的值，前者可以访问到依赖对象的其他属性，侧重于检查新值是否与其他属性兼容，不兼容时，通常会进行矫正返回一个其他的合理值，也可以返回DependencyProperty.UnsetValue拒绝本次赋值。后者侧重于捕获明显错误的新值，比如Margin不能为负数。  
ValidateValueCallback的实参是CoerceValueCallback的返回值，CoerceValueCallback能获知对象的其他的属性值，而ValidateValueCallback只能知道被修改的属性即将被赋予的值。

### 读写依赖属性

MyButton myButton = new MyButton();

// 读

myButton.GetValue(MyButton.BrushProperty);

// 写

myButton.SetValue(MyButton.BrushProperty, new SolidColorBrush(Colors.Red));

### CLR属性包装器

我们可以利用GetValue或SetValue读写实例的依赖属性，但是我们也可以简单封装一下提供等价的CLR属性，我们通过CLR属性读写依赖属性。

public Brush Brush

{

get { return (Brush)GetValue(BrushProperty); }

set { SetValue(BrushProperty, value); }

}

为什么要提供CLR属性包装器？

可以在XAML中通过CLR属性操作依赖属性，如绑定，赋值，资源等。

依赖对象可以作为数据源，依赖属性可以作为Path。

依赖属性是天生合格的数据源，属性值变化后具有通知能力。

注意

不要在CLR属性包装器的访问器中，添加属性值的合理性逻辑检查或属性值改变要触发的事件等任何其他代码，因为用户仍旧可以绕过CLR属性包装器利用GetValue和SetValue读写依赖属性。

### 深入理解CorceValueCallback

public class Gauge : Control

{

//注册CurrentReading依赖属性，并添加PropertyChanged、CoerceValue、ValidateValue的回调委托

public static readonly DependencyProperty CurrentReadingProperty = DependencyProperty.Register(

"CurrentReading",

typeof(double),

typeof(Gauge),

new FrameworkPropertyMetadata(

Double.NaN,

FrameworkPropertyMetadataOptions.None,

new PropertyChangedCallback(OnCurrentReadingChanged),

new CoerceValueCallback(CoerceCurrentReading)

),

new ValidateValueCallback(IsValidReading)

);

//注册MaxReading依赖属性，并添加PropertyChanged、CoerceValue、ValidateValue的回调委托

public static readonly DependencyProperty MaxReadingProperty = DependencyProperty.Register(

"MaxReading",

typeof(double),

typeof(Gauge),

new FrameworkPropertyMetadata(

double.NaN,

FrameworkPropertyMetadataOptions.None,

new PropertyChangedCallback(OnMaxReadingChanged),

new CoerceValueCallback(CoerceMaxReading)

),

new ValidateValueCallback(IsValidReading)

);

//注册MinReading依赖属性，并添加PropertyChanged、CoerceValue、ValidateValue的回调委托

public static readonly DependencyProperty MinReadingProperty = DependencyProperty.Register(

"MinReading",

typeof(double),

typeof(Gauge),

new FrameworkPropertyMetadata(

double.NaN,

FrameworkPropertyMetadataOptions.None,

new PropertyChangedCallback(OnMinReadingChanged),

new CoerceValueCallback(CoerceMinReading)

),

new ValidateValueCallback(IsValidReading));

public Gauge() : base()

{

}

//属性包装器，通过它来暴露CurrentReading的值

public double CurrentReading

{

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

set { SetValue(CurrentReadingProperty, value); }

}

//属性包装器，通过它来暴露MaxReading的值

public double MaxReading

{

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

set { SetValue(MaxReadingProperty, value); }

}

//属性包装器，通过它来暴露MinReading的值

public double MinReading

{

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

set { SetValue(MinReadingProperty, value); }

}

//验证value是否有效，如果返回True表示验证通过，否则会提示异常

public static bool IsValidReading(object value)

{

Double v = (Double)value;

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

}

//在CoerceCurrentReading加入强制判断赋值

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;

}

//在CoerceMaxReading加入强制判断赋值

private static object CoerceMaxReading(DependencyObject d, object value)

{

Gauge g = (Gauge)d;

double max = (double)value;

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

return max;

}

//在CoerceMinReading加入强制判断赋值

private static object CoerceMinReading(DependencyObject d, object value)

{

Gauge g = (Gauge)d;

double min = (double)value;

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

return min;

}

//当CurrentReading值改变的时候，调用MinReading和MaxReading的CoerceValue回调委托

private static void OnCurrentReadingChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)

{

d.CoerceValue(MinReadingProperty);

d.CoerceValue(MaxReadingProperty);

}

//当MaxReading值改变的时候，调用MinReading和CurrentReading的CoerceValue回调委托

private static void OnMaxReadingChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)

{

d.CoerceValue(MinReadingProperty);

d.CoerceValue(CurrentReadingProperty);

}

//当OnMinReading值改变的时候，调用CurrentReading和MaxReading的CoerceValue回调委托

private static void OnMinReadingChanged(DependencyObject d, DependencyPropertyChangedEventArgs e)

{

d.CoerceValue(MaxReadingProperty);

d.CoerceValue(CurrentReadingProperty);

}

}

<Window x:Class="WpfApp3.MainWindow"

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

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

xmlns:local="clr-namespace:WpfApp3"

WindowStartupLocation="CenterScreen"

Title="Callback\_Validation\_DPs" Height="400" Width="400">

<StackPanel Orientation="Vertical">

<local:Gauge x:Name="gauge1" MaxReading="100" MinReading="0" Height="20"/>

<StackPanel Orientation="Horizontal" Height="60">

<Label Content="当前值为 : "/>

<Label Background="Yellow" BorderBrush="Black" BorderThickness="1"

IsEnabled="False" Content="{Binding ElementName=gauge1, Path=CurrentReading}" Height="25" VerticalAlignment="Top" />

</StackPanel>

<Button x:Name="btnSetBelowMin" Content="设置为 -100"

Click="btnSetBelowMin\_Click"/>

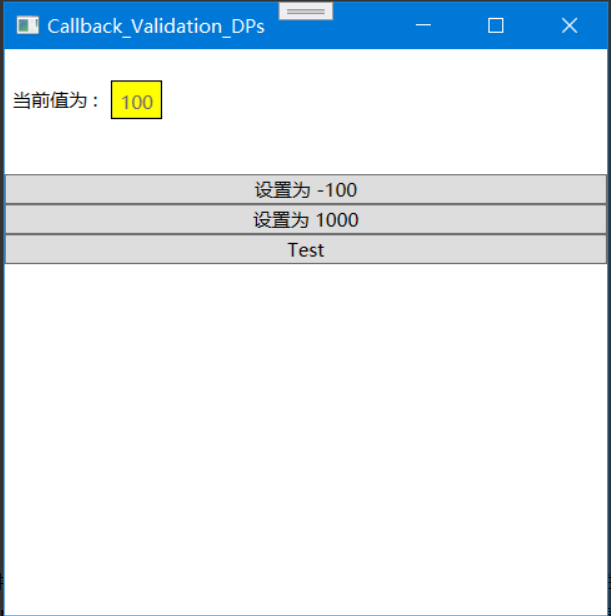
<Button x:Name="btnSetAboveMax" Content="设置为 1000"

Click="btnSetAboveMax\_Click"/>

<Button x:Name="btnTest" Content="Test" Click="btnTest\_Click"/>

</StackPanel>

</Window>



当相互作用的几个依赖属性其中一个发生变化时，在它的PropertyChangeCallback中调用受它影响的依赖属性的CoerceValue，这样才能保证相互作用关系的正确性。拿此例来说，无论我们对Max，Min，Current的赋值顺序如何，都能保证三者的值的合理性。CorceValueCallback虽然强制纠正了我们要赋给依赖属性的原始值，但是这个原始值仍旧会被WPF保存，在条件成立后，再次调用CorceValueCallback依赖属性的值会从纠正值变成原始值。  
单独调用CorceValueCallback虽然也可以改变依赖属性的值，但不会触发PropertyChangedCallback。

class MyButton : Button

{

public int Demo

{

get { return (int)GetValue(DemoProperty); }

set { SetValue(DemoProperty, value); }

}

// Using a DependencyProperty as the backing store for Demo. This enables animation, styling, binding, etc...

public static readonly DependencyProperty DemoProperty =

DependencyProperty.Register("Demo", typeof(int), typeof(MyButton), new FrameworkPropertyMetadata(2021) {Inherits=true});

}

class MyStackPanel: StackPanel

{

public int Demo

{

get { return (int)GetValue(DemoProperty); }

set { SetValue(DemoProperty, value); }

}

// Using a DependencyProperty as the backing store for Demo. This enables animation, styling, binding, etc...

public static readonly DependencyProperty DemoProperty = MyButton.DemoProperty.AddOwner(typeof(MyStackPanel),new FrameworkPropertyMetadata(0) { Inherits = true});

}

1. inherits必须设置成true
2. AddOwner时可以重新设置元数据
3. PropertyChanged会叠加，DefaultValue会重写

截断原因：

1. Inherits = false
2. 其他更高优先级设置了依赖属性值
3. Frame，StatusBar就像一个包围圈，屏蔽圈外的继承作用

### 获取依赖属性值的途径的优先级

1. CoerceValueCallBack强制更改的值
2. 动画
3. 本地
4. XAML或CS中的直接赋值，包括资源和数据绑定
5. 模板父类
6. 样式触发器（Style Trigger）
7. 模板触发器（Template Trigger）
8. 样式设置器（Style Setter）
9. 模板设置器（Template Setter）
10. 继承值
11. 默认值

### 清除本地值

即使没有设置依赖属性，但是它可能也已经有了数值，来源于动画，样式，数据绑定，继承等。如果为依赖属性赋予了本地值，根据优先级原则，其他的样式，数据绑定，触发器都会被忽略掉。当我们重新想让某个依赖属性不要继续再使用本地值，怎么干掉本地值呢？显然不可能通过另外再赋一个本地值的方式。

从DependencyObject中继承的ClearValue( )，会从依赖属性单实例清除本地值，那么后续所有同种类型的依赖对象实例从单实例读取值时，便会是由数据绑定，样式，动画等得到的。

**public** void ClearValue**(**DependencyProperty dp**);**

myElement**.**ClearValue**(**FrameworkElement**.**MarginProperty**);**

### WPF的属性系统的优势

WPF控件的属性值由实时的逻辑计算得到，而非存储在字段内存，WPF为什么这么做呢？

1. 更改通知
2. 支持传统的读写依赖属性
3. 依赖在其他对象上，即数据绑定
4. UI树的属性值继承
5. 默认值
6. Style，资源，动画
7. 属性值变化时也影响其他属性
8. 节省内存

## 附加属性 (Attached Property)

### 附加属性的本质

附加属性的作用就是让属性与数据类型解耦，我们可以使用附加属性动态扩展类的属性。

### 为什么要有附加属性

放置到DockPanel中的控件，必须具备属性Dock(Dock.Top，Dock.Buttom，Dock.Left，Dock.Right)，才能被DockPanel读到控件的布局信息以正确布局。

放置到Grid中的控件，必须具备属性Row，Column才能被Grid读到控件的布局信息以正确布局。

放置到Canvas中的控件，必须具备属性Canvas.Top，Canvas.Left才能被Canvas读到控件的布局信息以正确布局。

每个控件都可能放到各种布局容器中，那么每个控件必须都必须带有Dock，Row，Column，Canvas.Top等属性。这样做看似没有任何毛病，但是某个控件，如果没有被放置到Grid，那么它的Row，Column就没有什么作用了，只是占有内存。所以，我们可以为对象动态添加属性，在对象需要的时候，添加属性，在对象不需要的时候，不添加即可。

### 定义附加属性

class Person **:** DependencyObject

**{**

**}**

class School**:** DependencyObject

**{**

**public** static int GetGrade**(**DependencyObject obj**)**

**{**

**return** **(**int**)**obj**.**GetValue**(**GradeProperty**);**

**}**

**public** static void SetGrade**(**DependencyObject obj**,** int **value)**

**{**

obj**.**SetValue**(**GradeProperty**,** **value);**

**}**

// Using a DependencyProperty as the backing store for Grade. This enables animation, styling, binding, etc...

**public** static **readonly** DependencyProperty GradeProperty **=**

DependencyProperty**.**RegisterAttached**(**"Grade"**,** **typeof(**int**),** **typeof(**School**),** **new** PropertyMetadata**(**0**));**

**}**

// 使用附加属性

Person person **=** **new** Person**();**

School**.**SetGrade**(**person**,** 7**);**

附加属性与依赖属性没有实质性上的区别，形式上有以下不同：

（1）注册时，附加属性使用RegisterAttached，依赖属性使用Register

（2）依赖属性会提供CLR属性包装器，而附加属性不会提供。因为依赖属性一般是在定义它的类中使用，便于在XAML或后台代码中直接赋值，而附加属性一般不会在定义它的类中应用，虽然它可以应用到任何依赖对象中（包括定义它的类）。

（3）附加属性虽然没有提供CLR属性包装器，但是提供了两个方法GetPropertyName和SetPropertyName，其实这两个方法也可以不提供，我们可以直接

person.SetValue(School.GradeProperty,7);但是有了这两个方法，设置和获取附件属性会比较方便。

（4）确切的说，附加属性与定义它的类几乎无关，任何附加属性可被添加进任何对象，即使对象可能不需要也用不到这个附加属性，虽然定义附加属性的类提供了将一个附加属性添加到某个依赖对象的方法以及读取的方法，但是我们仍旧可以使用依赖对象的GetValue和SetValue使用附加属性。

（5）依赖属性也可以被当作附加属性，附加到一个依赖对象上，如

ComboBox cmb **=** **new** ComboBox**;**

cmb**.**SetValue**(**PasswordBox**.**PasswordCharProperty**,**"\*"**);**

该操作不会影响ComboBox，毕竟ComboBox的内部代码不会去查找它不使用的属性---但是我们可以在外界使用ComboxBox的此附加属性。但是在XAML中，是无法做到将一个依赖属性附加到别的不是定义它的类中的，相当于一个防呆。

### 附加属性的应用

<Grid>

<Button Grid.Column=**"0"** Grid.Row=**"0"**/>

</Grid>

Button button **=** **new** Button**();**

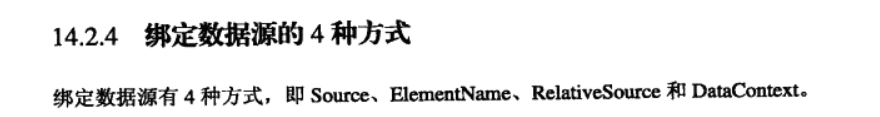
Grid**.**SetRow**(**button**,** 0**);**

Grid**.**SetColumn**(**button**,** 0**);**

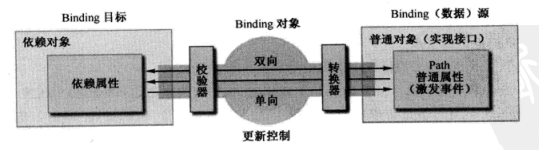
<Button Grid.Column=**"{Binding Path=ActualHeight, RelativeSource={RelativeSource Mode=Self}}"**/>

btn**.**SetBinding**(**Grid**.**Row**,new** Binding**(**"Value"**){**source **=** slider**});**

# 第四部分：数据绑定



## Binding的优势



应用程序一般可以分成以下三层

1. 数据存储层 ：存储方式有数据库,磁盘文件,内存。
2. 数据处理层：也叫逻辑层，业务层；包含读取和写回数据存储层的算法，映射存储文件的.NET类，有关业务逻辑的大量的算法。
3. 数据展示层：展示数据处理层的数据(变量)，收集用户操作反馈到数据处理层

WPF作为一种专门的展示层技术，华丽的外观和动画只是它的表象，更重要的是它在深层次上帮助程序员把工作重心放在逻辑层，让展示层永远处于逻辑层的从属地位，WPF具有这种能力的关键是它引入了DataBinding和DependencyProperty两个概念。

DataBinding在WPF系统中起到的是数据高速公路的作用，是逻辑层和展示层沟通和连接的桥梁，有了这条高速公路，加工好的数据会被自动送达用户界面加以显示，被用户修改过的数据也会自动传回逻辑层，一旦数据被加工好又会被送达用户界面......程序的逻辑层就像一个强有力的引擎不停的运转，用加工好的数据驱动程序的用户界面以文字，图形，动画等形式把数据显示出来，这就是数据驱动UI。

所有与业务逻辑相关的算法都处在数据逻辑层，逻辑层成为一个能够独立运转的，完整的体系，而用户界面层则不含任何代码，完全依赖和从属于数据逻辑层。这样前后端分离，逻辑层不需要界面就可以进行单元测试，而且，UI可以随时换肤但不影响逻辑层。WPF提供了一种界面与内部数据联动的方法=>数据绑定。WPF是一种内部数据与界面联动的技术。

## Bidning的重要概念

1. 目标对象必须是依赖对象，继承DependenceObject.
2. 目标对象的依赖属性才可以设置Binding。
3. 任何类型的对象都可以做数据源，包括控件
4. 任何属性都可以做源属性
5. 依赖属性可以与数据源或数据源的属性或数据源的多级属性关联。
6. 属性指有参属性(索引器)和无参属性，二者都可以作为源属性。
7. 一个依赖属性可以关联多个源属性，这些源属性可以来自不同的数据源。
8. 依赖对象是天生的数据源，它的依赖属性改变了会自动通知目标对象
9. 非依赖对象类型的对象做数据源，如果想通知目标对象源属性改变了，则该对象必须继承INotifyPropertyChanged接口。
10. 一个依赖属性只能设置一次Binding，多次设置，先前的Binding会自动清除
11. 一个源属性做很多Binding的Source，即可以与多个依赖属性关联。
12. 依赖属性未找到源属性，或者数据源或源属性的值为NULL，或者依赖属性和数据源属性类型不一致而不含值转换器，或者值转换器返回的类型与绑定的属性类型不一致，或者属性的set访问器抛出异常，都会被WPF忽略，此时目标属性的值是默认值或者继承而来的值。
13. 值转换器抛出异常，会导致应用程序崩溃。我们应该总是保证值转换器不抛出异常
14. 调试时，如果依赖属性未找到源属性，可以在output窗口查看发现。类似：System.Windows.Data Error: 40 : BindingExpression path error: 'XXXXXXXXXX' property not found on 'object' ''Int32' (HashCode=1)'. BindingExpression:Path=XXXXXXXXXX; DataItem='Int32' (HashCode=1); target element is 'TextBlock' (Name=''); target property is 'Text' (type 'String')

## 数据更新的方向

Binding.Mode = BindingMode.OneTime

单向，目标属性被源属性更新，程序启动时，双方的值是源属性的初始值。源属性只更新一次目标属性，后续值发生变化不会再更新目标属性，二者再无关系。

Binding.Mode = BindingMode.OneWay

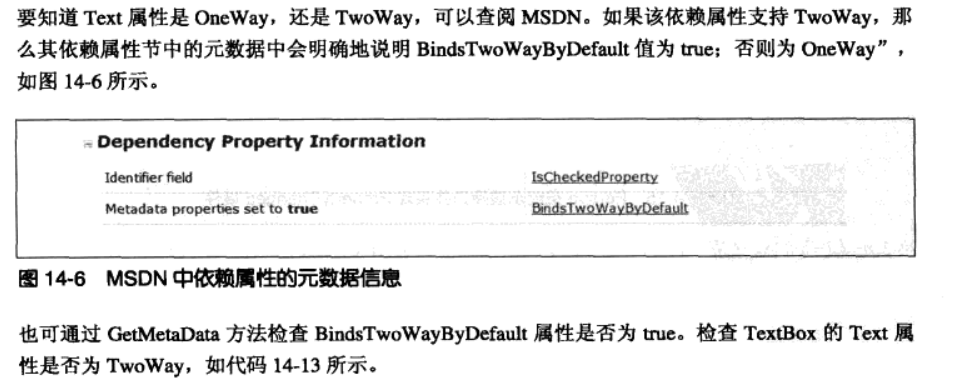
单向，目标属性被源属性更新，程序启动时，双方的值是源属性的初始值

Binding.Mode = BindingMode.OneWayToSource

单向，源属性被目标属性更新，程序启动时，双方的值是目标属性的初始值

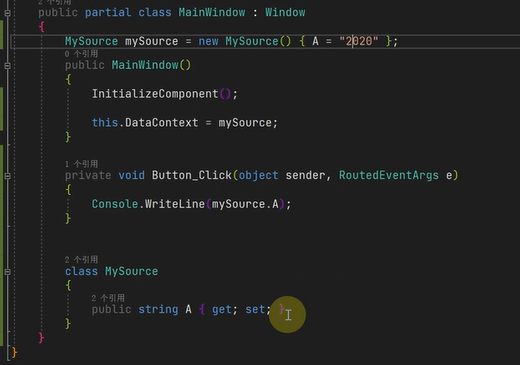
Binding.Mode = BindingMode.TwoWay

双向，源属性和目标属性互相更新，程序启动时，双方是源属性的初始值。

* OneTime，OneWay OneWayToSource，TwoWay的性能开销依次变大，刚好够用最好。
* 显示为每一个Binding指定Mode是个好习惯。
* 如果Mode不支持源更新目标的方向(OneWayToSource)，那么源属性则无法更新目标属性，即使Mode支持，但如果源不是依赖对象且未实现INotifyProeprtyChanged，也无法更新目标属性。
* 使用BindingExpression.UpdateTarget()和BidningExpression.UpdateSource()会强制更新一次目标属性或源属性，而不管Mode和INotifyProeprtyChanged。
* 

下面的Demo,验证OneWayToSource时，程序启动时，源属性和目标属性值一致且是目标属性的值。





程序启动后，文本框显示空字符串，同时属性源属性A也是空字符串而不是2020.

## Binding的语法

### 绑定的步骤

Binding是一个对象，先与数据源和数据源属性打包在一起，此时的Binding就像一个在后台自由流窜的随时可能变化的值，谁(UI元素)需要谁就可以抓过来用于给自己关联(赋值)。

Binding一端是数据源的属性，另一端是目标控件的依赖属性。

### XAML实现Binding

<TextBox x:Name="tbxName" Text="{Binding Path=Name, Mode=OneWay}"/>

### C#实现Binding

Binding bind = new Binding();

bind.Source = stu;

bind.Path = new PropertyPath("Name");

bind.Mode = BindingMode.OneWay;

BindingOperations.SetBinding(tbxName, TextBox.TextProperty, bind);

或

tbxName.SetBinding(TextBlock.TextProperty, bind);

WPF在继承链的FrameworkElement和FrameworkContentElement层次，为类增添了SetBinding()方法，内部调用的相当于全局的方法BindingOperations.SetBinding（）。

public BindingExpressionBase SetBinding(DependencyProperty dp, BindingBase binding)

{

BindingOperations.SetBinding(this,dp,binding);

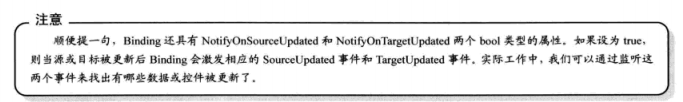
}

### 同一依赖对象的不同属性互相绑定

**public** BindingExpression SetBinding**(**DependencyProperty dp**,** string path**);**

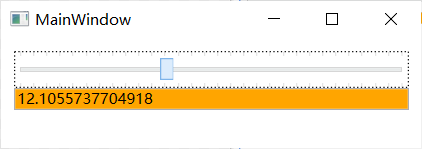
textBlock**.**SetBinding**(**TextBlock**.**TextProperty**,**”Background”**);**

### SourceUpdated和TargetUpdated



## Source

### ElementName



<Slider x:Name=**"slider"** Value=**"{Binding Path=Text, ElementName=tbk, Mode=TwoWay}"** Maximum=**"30"** Minimum=**"1"** TickFrequency=**"1"** TickPlacement=**"Both"**/>

<TextBox x:Name=**"tbk"** Text=**"1"** Background=**"Orange"**/>

在XAML中设置控件的x:Name=“xxx“，有两个作用，作用1控件的Name属性是xxx，作用2控件的引用变量名是xxx。我们也可以只设置Name=”xxx“,但是这样不会存在一个名称是xxx的引用变量。

Binding数据源是UI元素时，可以在使用Name属性也可以使用引用变量索引控件。在C#中可以Binding.Source=xxx（引用变量）【适用于x:Name=xxx，不适用Name=xxx】，也可以Bidning.ElementName=xxx(Name属性) 【适用于x:Name=xxx和Name=xxx】. 但是在XAML中，显然只能{Binding ElementName=xxx(Name属性)} 【适用于x:Name=xxx和Name=xxx】。

### Source

一般用来指定不可见的对象为数据源。

### RelativeResource

<TextBlock Width="120" Background="Orange" Text="{Binding Path=Width, RelativeSource={RelativeSource Mode=Self}}"/>



<StackPanel>

<StackPanel.Resources>

<Style TargetType="{x:Type Button}">

<Setter Property="Background" Value="Green"/>

<Setter Property="Template">

<Setter.Value>

<ControlTemplate TargetType="{x:Type Button}">

<Grid>

<Ellipse>

<Ellipse.Fill>

<SolidColorBrush Color="{Binding Path=Background.Color,RelativeSource={RelativeSource TemplatedParent}}"/>

</Ellipse.Fill>

</Ellipse>

</Grid>

</ControlTemplate>

</Setter.Value>

</Setter>

</Style>

</StackPanel.Resources>

<Button Width="120" Height="20" Background="Orange"/>

</StackPanel>

常用于数据模板和控件模板

### DataContext

Binding未指明Source时，会把依赖对象的DataContext作为Source。

当没有为依赖对象的DataContext指定值时，会根据UI树继承父容器和父控件的DataContext，如果指定值，则不会再从父容器和父控件继承。默认情况下，所有依赖对象的DataContext的值为NULL.

常用场景：

1. 当UI上的多个控件的数据源是同一个对象时，可以将对象设置成高层次级别的DataContext
2. 因为DataContext是public的，当窗体A内部的某个控件是private的，可以设置控件为窗体A的DataContext，那么窗体A的内部控件便可以暴露出来做Source

<StackPanel>

<StackPanel.DataContext>

<sys:Int32>2022</sys:Int32>

</StackPanel.DataContext>

<TextBlock Text="{Binding}"/>

<TextBlock Text="{Binding}"/>

<TextBlock Text="{Binding}"/>

</StackPanel>



### ObjectDataProvider

## Path

### 基础概念

* 任何属性(无参属性和有参属性)都可以指定为path。无参属性时，为Path指定属性名即可；有参属性时，我们并不知道属性名，WPF假定参数为属性名来识别有参属性，有参属性的参数有多少种，那么就可以认为有多少个有参属性。
* 索引器便是有参属性，所以索引器可以作为path，集合的某个元素也是索引器Indexer，所以集合中的某个元素可以作为path
* Path可以设置成数据源的某个属性，也可以设置成这个数据源的属性的属性… 即多级属性，绑定多级属性的写法：无参属性加 . 即可，有参属性加[常量参数]或使用public PropertyPath(string path, params object[] pathParameters);以变量形式为有参属性传参即可。
* 利用Binding.IndexerName可以通知依赖属性数据源的某个有参属性发生了变化。
* 附加属性
* 无path的绑定，即绑定源本身而非源属性

示例：TextBlock显示TextBox的内容或内容的某个字符或内容长度。

<TextBox Name="Input" Height="30" Background="BlueViolet"/>

<TextBlock Height="30" Text="{Binding Path=Text, ElementName=Input}" Background="Orange"/>

<TextBox Name="Input" Height="30" Background="BlueViolet"/>

<TextBlock Height="30" Text="{Binding Path=Text[0], ElementName=Input}" Background="Orange"/>

<TextBox Name="Input" Height="30" Background="BlueViolet"/>

<TextBlock Height="30" Text="{Binding Path=Text.Length, ElementName=Input}" Background="Orange"/>

### 多级属性

示例一

class City

{

public string Name { get; set; }

}

class Province

{

public string Name { get; set; }

public IList<City> Cities { get; set; }

}

class Country

{

public string Name { get; set; }

public IList<Province> Provinces { get; set; }

}

City city1 = new City() { Name = "珠海" };

City city2 = new City() { Name = "郑州" };

Province province1 = new Province() { Name = "广东", Cities = new City[] { city1 } };

Province province2 = new Province() { Name = "河南", Cities = new City[] { city2 } };

Country country = new Country() { Name = "中华人民共和国", Provinces = new Province[] { province1, province2 } };

this.DataContext = country;

在XAML中实现绑定

<TextBlock Text="{Binding Path=Provinces[1].Cities[0].Name }" Height="30" Background="Gray"/>

也可以使用C#代码实现绑定

txt.SetBinding(TextBox.TextProperty, new Binding("Provinces[0].Cities[0].Name ") { Mode = BindingMode.OneWay });

### 实现具有通知能力的索引器

数据源属性变化时，可以通知UI。有参属性变化时，利用Binding.IndexerName自动得到相应的属性名来通知目标对象.

class MyDataSource : INotifyPropertyChanged

{

Dictionary<string, string> dic1 = new Dictionary<string, string>() { { "A", "a" }, { "B", "b" }, { "C", "c" } };

Dictionary<int, string> dic2 = new Dictionary<int, string>() { { 1, "one" }, { 2, "two" }, { 3, "three" } };

public string this[string key1, int key2]

{

get

{

return dic1[key1] + dic2[key2];

}

set

{

string temp = dic1[key1];

dic1[key1] = dic2[key2];

dic2[key2] = temp;

PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(Binding.IndexerName));

}

}

public event PropertyChangedEventHandler PropertyChanged;

}

### 索引器

#### 绑定时，字面值作为有参属性的参数

参数间用逗号隔开，在XAML中逗号需要转义。

<TextBlock x:Name="tbk" Text="{Binding Path=[A\,1]}"/>

当一个类有多个参数数量相同的索引器，可以在传参时加类型，以做区分，如Path="[(sys:Int32)42,(sys:Int32)24]"

#### 绑定时，变量作为有参属性的参数

只能用C#代码实现绑定。

0和1要加(),（0）（1）相当于占位符，表示参数。

string key1 = "A";

int key2 = 1;

MyDataSource s = new MyDataSource();

Binding bind = new Binding();

bind.Source = s;

bind.Path = new PropertyPath("[(0),(1)]",key1,key2);

tbk.SetBinding(TextBlock.TextProperty, bind);

s[key1, key2] = "";

### 附加属性

我们绑定的源属性几乎都是在数据源类中定义的，但是有时候我们会绑定到一个数据源不存在的属性，按常理说，如果数据源不存在该属性，则依赖属性找不到源属性，便无绑定。但是如果我们用 (属性) ，即括号括住属性这种形式来强调该属性数据源确实不存在，但这个属性是附加属性，被别人给予，相当于数据源定义了该属性。

附加属性既可以绑定，也可以被绑定。

<Grid>

<Grid ShowGridLines="True">

<Grid.ColumnDefinitions>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

<ColumnDefinition/>

</Grid.ColumnDefinitions>

<!--附加属性做依赖属性-->

<Button Name="btnRed" Background="Red" Grid.Column="{Binding}" />

<!--附加属性做源属性-->

<Button Background="Green" Height="30" Grid.Column="{Binding Path=(Grid.Column), ElementName=btnRed}"/>

</Grid>

</Grid>

this.DataContext = 2;

### 依赖属性与源本身相关联而不是与源属性相关联

如果Binding不显示设置path或path = . ，则依赖属性不与数据源的任何属性相绑定，而是直接与数据源本身相关联。如果没有为Binding设置StringFormat或Converter，且数据源类型与依赖属性类型一致，那么相当于直接将数据源对象赋值给依赖属性。

#### 基元类型作为path

我们将基元类型设置成数据源时，就可以不指定path，因为也确实无法指定path。

在XAML可以省略 . ,但是在C#中，必须显示指定path为 .

<Grid>

<Grid.Resources>

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

大漠孤烟直

</sys:String>

</Grid.Resources>

<TextBlock Text="{Binding Source={StaticResource res}}"/>

</Grid>

Binding bind = new Binding();

bind.Source = "大漠孤烟直";

bind.Path = new PropertyPath(".");

txt.SetBinding(TextBlock.TextProperty, bind);



#### path实现动态赋值

依赖属性与数据源时刻保持相等，数据源改变，依赖属性的值也跟着改变。

我们经常看到这样形式的设置列表控件的数据源。列表控件的DataContext便是真正的ItemsSouce，且DataContext的值改变，ItemsSource也会跟着改变。

<ComboBox x:Name = “cbx” ItemsSource="{Binding}"/>

cbx.DataContext = new List<string>(){“1”,”2”,”3”};

## TargetNullValue和FallbackValue

TargetNullValue和FallbackValue是针对依赖属性而存在的两个属性。

### TargetNullValue

（Source不为null，且Source类中含有Path指定的属性），依赖属性找到Binding的path指定的源属性，但是发现属性的值为null，此时，依赖属性会将TargetNullValue作为自己的值。

### FallBackValue

1. 依赖属性找不到Binding的path指定的源属性（Source为null或Source不为null但不含有Path指定的属性），依赖属性的值会是继承值或默认的元数据值。但是如果为Binding的FallBackValue指定了值，依赖属性的值会是FallBackValue。
2. （Source不为null，且Source类中含有Path指定的属性）依赖属性成功的找到了path指定的属性，但是因为依赖属性的类型和源属性的值不一致且无值转换器，或者值转换器发生异常，验证器发生异常，get访问器出现异常等其他原因，导致源属性无法为依赖属性提供一个值，此时依赖属性也会以继承值或默认的元数据值或FallBackValue作为自己的值。

验证demo1：Image控件显示TargetNullValue

<Window.Resources>

<BitmapImage x:Key="targetNull" UriSource="C:\Users\Administrator\Desktop\targetNull.jpg"/>

<BitmapImage x:Key="fallBack" UriSource="C:\Users\Administrator\Desktop\fallBack.jpg"/>

</Window.Resources>

<StackPanel>

<Image Source="{Binding BitmapImage, TargetNullValue={StaticResource targetNull}}"/>

</StackPanel>

public partial class MainWindow : Window

{

MySource MySource = new MySource();

public MainWindow()

{

InitializeComponent();

this.DataContext = MySource;

}

}

class MySource

{

public BitmapImage BitmapImage { get; set; }

}

验证Demo2：Image控件显示FallBackValue

<Window.Resources>

<BitmapImage x:Key="targetNull" UriSource="C:\Users\Administrator\Desktop\targetNull.jpg"/>

<BitmapImage x:Key="fallBack" UriSource="C:\Users\Administrator\Desktop\fallBack.jpg"/>

</Window.Resources>

<StackPanel>

<Image Source="{Binding BitmapImage, TargetNullValue={StaticResource targetNull}}"/>

</StackPanel>

public partial class MainWindow : Window

{

MySource MySource = new MySource();

public MainWindow()

{

InitializeComponent();

this.DataContext = MySource;

}

}

class MySource

{

}

### 经典示例

<Window.Resources>

<sys:String x:Key="targetNull">TargetNull</sys:String>

<sys:String x:Key="fallBack">FallBack</sys:String>

</Window.Resources>

<Grid>

<TextBlock Width="200" Height="50" Background="Orange" Text="{Binding TargetNullValue={StaticResource targetNull}, FallbackValue={StaticResource fallBack}}"/>

</Grid>

后台CS中不添加任何代码，即不设置DataContext，文本框会显示TargetValue，因为，Text绑定在数据源本身了，而数据源又是null。

如果我们添加this.DataContext = 1，文本框则会显示1，而不是FallBackValue。

启动程序，UI会显示武林外传.jpg。数据源为null，path是数据源本身，即path等于null，所以显示TargetNullValue。

我们在后台增加一行代码：

this.DataContext = 1;

再次启动程序，UI上不显示任何图片。因为此时的数据源是1，未找到path指定的属性，而不是找到path指定的属性但path的值为null，所以不再显示TargetNullValue。

<Window.Resources>

<BitmapImage x:Key="default" UriSource="C:\Users\Administrator\Desktop\武林外传.jpg"/>

</Window.Resources>

<Grid>

<Image Source="{Binding FallbackValue={StaticResource default}}"/>

</Grid>

启动程序，UI不显示任何图片。

我们在后台增加一行代码：

this.DataContext = 1;

再次启动程序，UI会显示FallbackValue。

## 值转换器

依赖属性和数据源属性相关联，即根据对方的属性，进行逻辑转换，赋值自己的属性。这个逻辑转换通过值转换器或格式化器来实现的。

Binding两侧是属性，数据在Binding上流通时，Binding可以对其进行加工。Binding.StringFormat可以针对数字，日期，字符串简单的格式化，复杂的逻辑转换需要Binding.Converter.另外，StringFormat只能处理数据源流向目标对象的数据，如果是OneWayToSource或者TwoWay则只能使用Converter。

### Binding.StringFormat

{0}表示源属性，加:格式化字符串即可。如{0:xxxxx}

如果是MultiBinding，可能出现StringFormat = {}{0:C3},{1:F0},{2:P}

格式化字符串后面的带的数字通常用来表示保留几位小数，如0是取整，1是保留1位小数。

StringFormat适合OneTime和OneWay两种模式，用来处理数据源流向目标对象的数据，反之则不能。

列表控件的ItemStringFormat可以为每一个子项实现StringFormat。

<!-- DateTime -->

<TextBlock Height="50" Background="Gray" Text="{Binding StringFormat={}{0:yyyy年MM月dd日 HH:mm:ss}, Source={x:Static local:MainWindow.DateTime}, Path=.}"/>

<!—货币 -->

<TextBlock Height="50" Background="Yellow" Text="{Binding StringFormat={}{0:C3}, Source={x:Static local:MainWindow.Price}}"/>

<!—百分比 -->

<TextBlock Height="50" Background="BlueViolet" Text="{Binding StringFormat={}{0:P0}, Source={x:Static local:MainWindow.Percentage}}"/>

<!—数字 -->

<TextBlock Height="50" Background="BlueViolet" Text="{Binding StringFormat={}{0:F1}, Source={x:Static local:MainWindow.Number}}"/>

<!—列表控件也可以”一键”格式化-->

<ComboBox x:Name="cbx" DisplayMemberPath="." ItemStringFormat="单价:{0:C}" SelectedIndex="0"/>

public static DateTime DateTime = DateTime.Now;

public static double Price = 9d;

public static double Percentage = 0.171;

public static int Number = 3;



### 值转换器

#### 实现一个类型转换器的步骤

1. 继承System.Windows.Data.IValueConverter

2. 实现接口方法Convert和ConvertBack

3. 在XAML中资源字典添加一个转换器实例，利用静态资源为Binding的Converter属性赋值

#### Sample1: 整型总秒数与时分秒字符串

* 转换器

Convert方法的第一个参数是数据源属性的值，第二个参数指该转换器寄宿的Binding实际绑定的依赖属性的类型，也是我们的Convert方法应当返回的类型！第三个参数是为方法提供的额外信息，该参数来自Binding的ConverterParameter属性，我们为Binding的ConverterParameter属性赋值，该Binding在数据转换调用Convert或ConvertBack方法时，会取ConverterParameter作为方法的第三个参数的实参。最后一个参数是当前线程的文化环境。

value，targetType，culture这三个参数是当数据流经Binding时，Binding通过自己的Converter属性调用Convert或ConvetBack方法时，根据Binding两侧的属性的类型自动传入的。只有parameter这个参数可以让开发者自己决定提供什么参数。所以，同一个值转换器实例，赋值给不同的Binding，它的Convert和Converter方法运行时获得的参数可能不同，这取决于Binding连接的是何种类型的依赖属性和源属性。

ConvertBack方法的第一个参数是依赖属性的值，第二个参数指要返回的类型，即数据源属性的类型，第三个参数同上。

class Number2Timespan : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

return TimeSpan.FromSeconds((int)value).ToString();

}

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

{

return TimeSpan.Parse(value.ToString()).TotalSeconds;

}

}

* 添加资源

<Window.Resources>

<local:Number2Timespan x:Key="ntConv"/>

</Window.Resources>

* 使用

<TextBlock Text="{Binding Converter={StaticResource ntConv}}"/>

#### Sample2 : 利用转换器属性或ConverterParameter复用转换器

分数分成3个等级：优秀，良好，差。优秀成绩文本为绿色，良好成绩文本为黄色，差成绩文本为红色。

class Score2Background : IValueConverter

{

public double GoodScore { get; set; }

public double PassingScore { get; set; }

public double TerribleScore { get; set; }

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

double score = (double)value;

if (score >= GoodScore)

{

return new SolidColorBrush(Colors.Green);

}

if (score >= PassingScore)

{

return new SolidColorBrush(Colors.Yellow);

}

return new SolidColorBrush(Colors.Red);

}

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

{

throw new NotImplementedException();

}

}

<local:Score2Background x:Key="score" GoodScore="90" PassingScore="80" TerribleScore="70" />

<TextBlock Background="{Binding Converter={StaticResource score}}"/>

class Score2Background : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

double[] scores = parameter as double[];

double score = (double)value;

if (score >= scores[0])

{

return new SolidColorBrush(Colors.Green);

}

if (score >= scores[1])

{

return new SolidColorBrush(Colors.Yellow);

}

return new SolidColorBrush(Colors.Red);

}

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

{

throw new NotImplementedException();

}

}

<x:Array Type="{x:Type System:Double}" x:Key="args">

<System:Double>90</System:Double>

<System:Double>80</System:Double>

<System:Double>70</System:Double>

</x:Array>

<TextBlock Text="{Binding ConverterParameter={StaticResource args}}"/>

### 值转换器和数据触发器的比较

数据触发器的优点是不用编写代码，缺点是不能指定高于或低于某个条件设置依赖属性，而值转换器就比较灵活。例子：显示价格的文本框，高于某个价格，文本框的背景颜色显示为红色，其他情况为灰色。

数据库中的文件路径得到UI图。

### 通过ConverterParameter或自定义属性为值转换器传参

Sample：分数分成3个等级：优秀，良好，差劲。优秀成绩文本框背景色为绿色，良好成绩文本框背景色为黄色，差劲成绩文本框背景为红色。

### ConverterParameter

class Score2Brush : IValueConverter

{

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

double[] scores = parameter as double[];

double score = System.Convert.ToDouble(value);

if(score >= scores[0])

{

return new SolidColorBrush(Colors.Green);

}

else if(score >= scores[1])

{

return new SolidColorBrush(Colors.Yellow);

}

else

{

return new SolidColorBrush(Colors.Red);

}

}

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

{

throw new NotImplementedException();

}

}

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

<Window.Resources>

<local:Score2Brush x:Key="sb"/>

</Window.Resources>

<Grid>

<TextBlock Text="{Binding}">

<TextBlock.Background>

<Binding Converter="{StaticResource sb}">

<Binding.ConverterParameter>

<x:Array Type="{x:Type sys:Double}">

<sys:Double>90</sys:Double>

<sys:Double>60</sys:Double>

</x:Array>

</Binding.ConverterParameter>

</Binding>

</TextBlock.Background>

</TextBlock>

</Grid>

### 自定义属性

class Score2Brush : IValueConverter

{

public double ExcellentScore { get; set; }

public double GoodScore { get; set; }

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

double score = System.Convert.ToDouble(value);

if(score >=ExcellentScore)

{

return new SolidColorBrush(Colors.Green);

}

else if(score >= GoodScore)

{

return new SolidColorBrush(Colors.Yellow);

}

else

{

return new SolidColorBrush(Colors.Red);

}

}

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

{

throw new NotImplementedException();

}

}

<Window.Resources>

<local:Score2Brush x:Key="sb" ExcellentScore="90" GoodScore="60"/>

</Window.Resources>

### 支持MarkupExtension的值转换器

### 代码示例

public class MyConverter : MarkupExtension, IValueConverter

{

public MyConverter(int propExample)

{

this.PropExample = propExample;

}

public int PropExample { get; set; }

public object Convert(object value, Type targetType, object parameter, CultureInfo culture)

{

return this.PropExample;

}

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

{

throw new NotImplementedException();

}

public override object ProvideValue(IServiceProvider serviceProvider)

{

return this;

}

}

<TextBlock Text="{Binding Converter={local:MyConverter 2020,PropExample=2021}}"/>

public class MyMultiValueConverter : MarkupExtension, IMultiValueConverter

{

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

{

throw new NotImplementedException();

}

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

{

throw new NotImplementedException();

}

public override object ProvideValue(IServiceProvider serviceProvider)

{

return this;

}

}

### 多值转换器

使用MultiBinding时，有多个数据源属性，可以继承IMultiConverter实现值转换器，也可以使用StringFormat="{}{0},{1}"这种格式，{0}表示第一个源属性，{1}表示第二个源属性。

### 二者比较

将值转换器在资源字典中实例化，再利用StaticResource赋值给Binding的Converter属性，这样在应用程序中多个Binding的Converter可以重复利用这个转换器实例资源，应用程序中始终只有一个转换器实例。花括号实例化值转换器的方法，会导致应用程序中存在多个完全相同的值转换器实例，这是不必要的。综上，如果一个转换器类只应用于一个或少量的Binding的Converter属性，那么我们可以牺牲内存来换来编程的方便，否则，还是用资源字典形式比较好。

## 绑定的方向

1. OneTime

源属性是初始值，依赖属性从源属性拿值，成功拿到的话，二者值相同。否则依赖属性是默认值。

可通过重新设置数据源或BindingExpression.UpdateTarget( )刷新依赖属性

1. OneWay

源属性是初始值，依赖属性从源属性拿值，成功拿到的话，二者值相同。否则依赖属性是默认值。

1. TwoWay

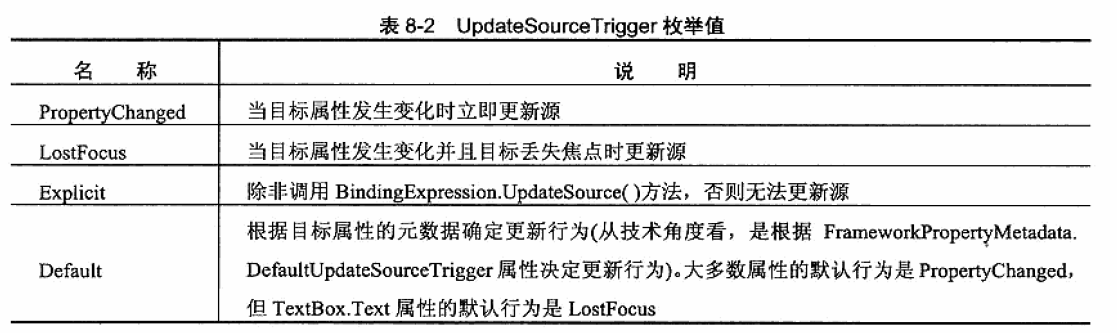
源属性是初始值，依赖属性从源属性拿值，成功拿到的话，二者值相同。否则依赖属性是默认值。

1. OneWayToSource

依赖属性是初始值，然后源属性从依赖属性拿值，源属性的初始值会被覆盖。

TwoWay的性能开销最大，OneTime的性能开销最小，总是为Binding显示指定合适的Mode值是个好习惯。

## 依赖属性刷新源属性的时机



PropertyChanged时，目标的依赖属性被更改时，实时触发数据源更新；优点是用户可以实时观察效果，缺点以TextBox举例，用户想输入5个字符，但是每输入一个字符便会触发一次数据源更新，其中有4次没必要的更新增加了系统性能开销。解决办法有两个，设置Delay属性或设置更新时机是LostFocus或Explicit。

### 延迟更新数据源Delay

<Slider Value=**"{Binding Path=Text, ElementName=tbk, UpdateSourceTrigger=PropertyChanged,Delay=500}"** Maximum=**"30"** Minimum=**"1"** TickFrequency=**"1"** TickPlacement=**"Both"**/>

<TextBox x:Name=**"tbk"** Text=**"1"**/>

Binding的Delay属性，会将该Binding的目标的属性改变信息延迟指定毫秒后通知到数据源，这会降低性能开销，尤其适用于数据源属性变化时执行处理器密集型操作。请注意，Delay是延迟依赖属性改变时的通知！数据源的属性改变还是会立刻通知到目标。

### 完全控制更新数据源的时机Explicit

<Slider x:Name=**"slider"** Value=**"{Binding Path=Text, ElementName=tbk, UpdateSourceTrigger=Explicit, Mode=TwoWay, Delay=500}"** Maximum=**"30"** Minimum=**"1"** TickFrequency=**"1"** TickPlacement=**"Both"**/>

<TextBox x:Name=**"tbk"** Text=**"1"**/>

<Button x:Name=**"btnSunmitTargetAlter"** MinHeight=**"20"** Content=**"提交(更新数据源属性)"** Click=**"BtnSunmitTargetAlter\_Click"**/>

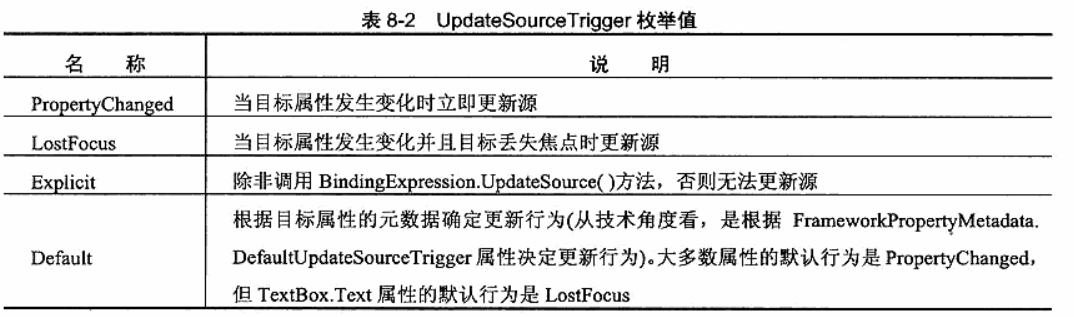
**private** void BtnSunmitTargetAlter\_Click**(object** sender**,** RoutedEventArgs e**)**

**{**

BindingExpression be **=** BindingOperations**.**GetBindingExpression**(**slider**,** Slider**.**ValueProperty**);**

be**.**UpdateSource**();**

**}**



Binding.UpdateSourceTrigger有4个枚举值，Explicit，只有程序员调用BindingExpression.UpdateSource( )，目标属性的变化才会通知数据源。一般我们有个Apply(应用)按钮，点击按钮调用更新方法，先获取依赖属性的BindingExpression再去更新。

### IsAsync

如果源属性的 getter比较耗时，比如从web请求得到属性值，可以使用Binding. IsAsync 设置为 true 可以避免阻塞UI。

## MultiBinding

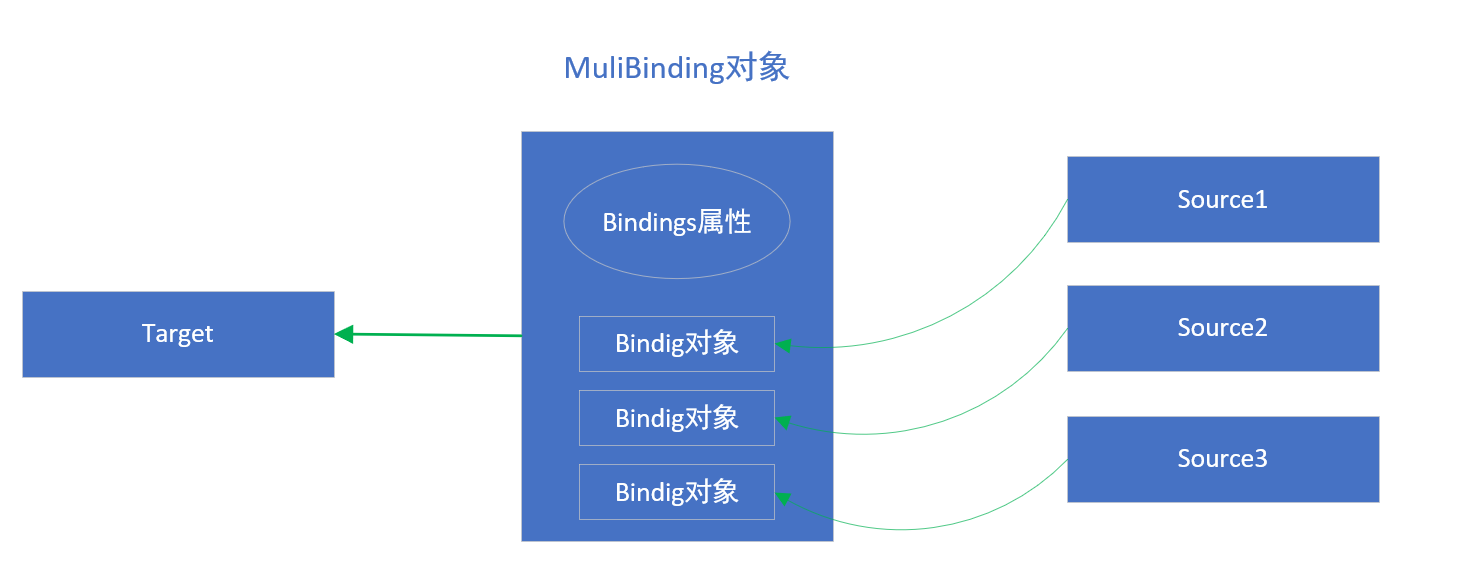
### 基础概念

一个依赖属性的值与多个源属性关联，这些源属性可以来自同一个数据源，也可以来自不同的数据源。

MultiBinding的Converter继承自IMultiValueConverter.

MultiBinding对加入它的Collection<BindingBase> Bindings的子级Binding的顺序是敏感的，顺序直接决定IMultiValueConverter的object[] Values的元素。

BindingBase是Binding和MultiBinding的基类。



### 绑定步骤

1. 指明所有的源属性和数据源
2. 指定MultiBinding的转换器（StringFormat或Converter）

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

### 例子1：显示人的全名

XAML

<TextBlock>

<TextBlock.Text>

<MultiBinding StringFormat="{}{0},{1}">

<Binding Path="FirstName"/>

<Binding Path="LastName"/>

</MultiBinding>

</TextBlock.Text>

</TextBlock>

this.DataContext = new Person() { FirstName = "Li", LastName = "Liuwei" };

### 例子2：账户和密码框有内容时登录按钮才能响应

UI

<StackPanel>

<TextBox x:Name="tbxId"/>

<TextBox x:Name="tbxPwd"/>

<Button x:Name="btnSubmit" Content="登录"/>

</StackPanel>

C#代码实现绑定

MultiBinding mb = new MultiBinding();

mb.Bindings.Add(new Binding("Text") { ElementName = tbxId.Name });

mb.Bindings.Add(new Binding("Text") { ElementName = tbxPwd.Name });

mb.Converter = new LogonConverter();

btnSubmit.SetBinding(Button.IsEnabledProperty, mb);

转换器

class LogonConverter : IMultiValueConverter

{

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

{

string id = values[0] as string;

string pwd = values[1] as string;

if(string.IsNullOrEmpty(id) || string.IsNullOrEmpty(pwd))

{

return false;

}

else

{

return true;

}

}

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

{

throw new NotImplementedException();

}

}

## 通过BindingOperations操作依赖属性和源属性

### 清除依赖属性上的Binding

清除目标对象上的某个依赖属性上的Binding

public static void ClearBinding(DependencyObject target, DependencyProperty dp);【BindingOperations】

清除目标对象上所有依赖属性上的Binding

public static void ClearAllBindings(DependencyObject target); 【BindingOperations】

### BindingExpression的强大功能

**获取指定的目标控件的指定的依赖属性的BindingExpression**

public static BindingExpression GetBindingExpression(DependencyObject target, DependencyProperty dp);

BindingExpression bindingExpression = BindingOperations.GetBindingExpression(tbk, TextBlock.TextProperty);

**强制更新数据源或目标控件。**

数据源未实现通知机制，可以使用此方法定时刷新界面。Binding.UpdateSourceTrigger=Explicit可以完全控制更新数据源的时机。

其实，无论Binding的属性怎么设置，只要调用方法，都会更新一次目标和源。

bindingExpression.UpdateSource();

bindingExpression.UpdateTarget();

// 依赖对象的信息

Console.WriteLine(bindingExpression.Target.GetType().Name);

// 依赖属性信息

Console.WriteLine(bindingExpression.TargetProperty.Name);

// 顶级数据源的信息

Console.WriteLine(bindingExpression.DataItem.GetType().Name);

// 直接数据源的信息(如果绑定的是属性的属性，则是直接数据源，它是顶级数据源的属性)

Console.WriteLine(bindingExpression.ResolvedSource.GetType().Name);

// 数据源属性的信息

Console.WriteLine(bindingExpression.ResolvedSourcePropertyName);

// 获取Binding

Binding binding = bindingExpression.ParentBinding;

public static MultiBindingExpression GetMultiBindingExpression(DependencyObject target, DependencyProperty dp);

### BindingOperations的强大功能

BindingOperations.ClearBinding(目标控件,依赖属性)

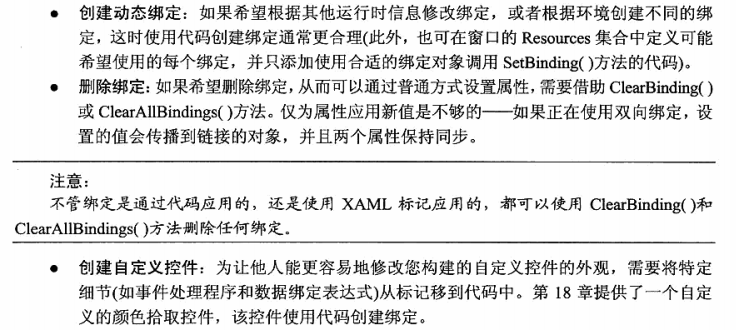
BindingOperations.ClearAllBindings(目标控件)

Binding binding = BindingOperations.GetBinding(目标控件，具有绑定表达式的依赖属性)

Binding.ElementName 目标控件本身

Binding.Path.PropertyPath 数据源属性的值

Binding.Path.Path 数据源属性的名称



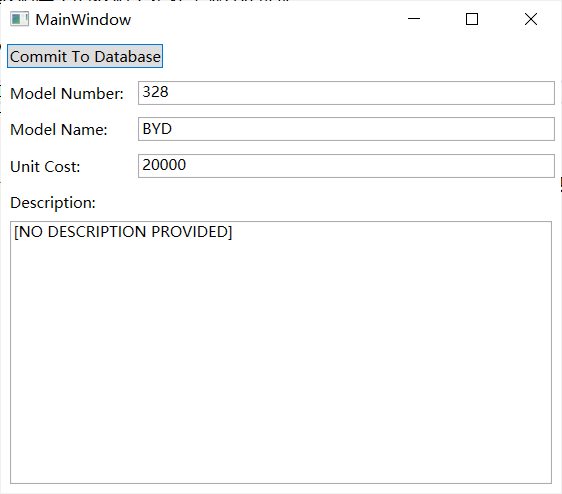
## 零零碎碎

UI上的元素之间可以建立绑定，使UI元素间联动，构建出更精彩的UI效果。

未找到绑定的Path或不存在源等绑定错误，WPF不会报错也不会有任何提示。我们可以在调试时查看Visual Studio的Output窗口，会输出所有的绑定错误。

System.Windows.Data Error: 40 : BindingExpression path error: 'Tet' property not found on 'object' ''TextBox' (Name='tbk')'. BindingExpression:Path=Tet; DataItem='TextBox' (Name='tbk'); target element is 'Slider' (Name='slider'); target property is 'Value' (type 'Double')

## 绑定到集合对象



<DockPanel>

<StackPanel DockPanel.Dock=**"Top"** Orientation=**"Horizontal"** Margin=**"5"**>

<Button x:Name=**"btnCommit"** Content=**"Commit To Database"** IsDefault=**"True"** Click=**"BtnCommit\_Click"**/>

</StackPanel>

<Grid x:Name=**"gridProductDetails"**>

<Grid.ColumnDefinitions>

<ColumnDefinition Width=**"auto"**/>

<ColumnDefinition Width=**"\*"**/>

</Grid.ColumnDefinitions>

<Grid.RowDefinitions>

<RowDefinition Height=**"auto"**/>

<RowDefinition Height=**"auto"**/>

<RowDefinition Height=**"auto"**/>

<RowDefinition Height=**"auto"**/>

<RowDefinition Height=**"\*"**/>

</Grid.RowDefinitions>

<TextBlock Margin=**"7"** Text=**"Model Number:"**/>

<TextBox Margin=**"5"** Grid.Column=**"1"** Text=**"{Binding Path=ModelNumber}"**/>

<TextBlock Margin=**"7"** Grid.Row=**"1"** Text=**"Model Name:"**/>

<TextBox Margin=**"5"** Grid.Row=**"1"** Grid.Column=**"1"** Text=**"{Binding Path=ModelName}"**/>

<TextBlock Margin=**"7"** Grid.Row=**"2"** Text=**"Unit Cost:"**/>

<TextBox Margin=**"5"** Grid.Row=**"2"** Grid.Column=**"1"** Text=**"{Binding Path=UnitCost}"**/>

<TextBlock Margin=**"7,7,7,0"** Grid.Row=**"3"** Text=**"Description:"**/>

<TextBox Margin=**"7"** Grid.Row=**"4"** Grid.Column=**"0"** Grid.ColumnSpan=**"2"** TextWrapping=**"Wrap"**

Text=**"{Binding Path=Description, TargetNullValue=[NO DESCRIPTION PROVIDED]}"**/>

</Grid>

</DockPanel>

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

this.gridProductDetails.DataContext = App.StoreDB.GetPtoduct(1);

}

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

{

FocusManager.SetFocusedElement(this, sender as IInputElement);

Product p = gridProductDetails.DataContext as Product;

App.StoreDB.UpdateProduct(p);

}

}

表单输入-按钮提交型的交互，点击提交按钮，焦点会转移到按钮，在表单最后一次输入的数据会被更新到数据源，提交到数据库中的值就是在表单中最后一次输入的值。但是，如果界面的初始化状态提交按钮的IsDefault=True，用户在表单中输入完毕，按回车键触发按钮进行提交，由于这不会将焦点从表单转移到按钮，那么用户在表单中输入的最后一次值，不会更新到数据源，那么提交到数据库中的数据不是表单显示的值。我们可以在提交按钮的click事件处理程序中强制改变焦点来避免这种现象。

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

**{**

FocusManager**.**SetFocusedElement**(this,** sender **as** IInputElement**);**

Product p **=** gridProductDetails**.**DataContext **as** Product**;**

App**.**StoreDB**.**UpdateProduct**(**p**);**

**}**

IsDefault是指响应Enter键，IsCancel是指响应ESC键，这与焦点是不同的东西。焦点随着鼠标的点击在跟着变，但IsDefault只取决于Tab键。

继承ItemsControl的类用于显示集合的控件，如ListBox，ListView，DataGrid，ComboBox，TreeView，Menu。ListBox，ListView，DataGrid，ComboBox显示各元素地位平等的集合，TreeView，Menu显示元素层次化的集合。

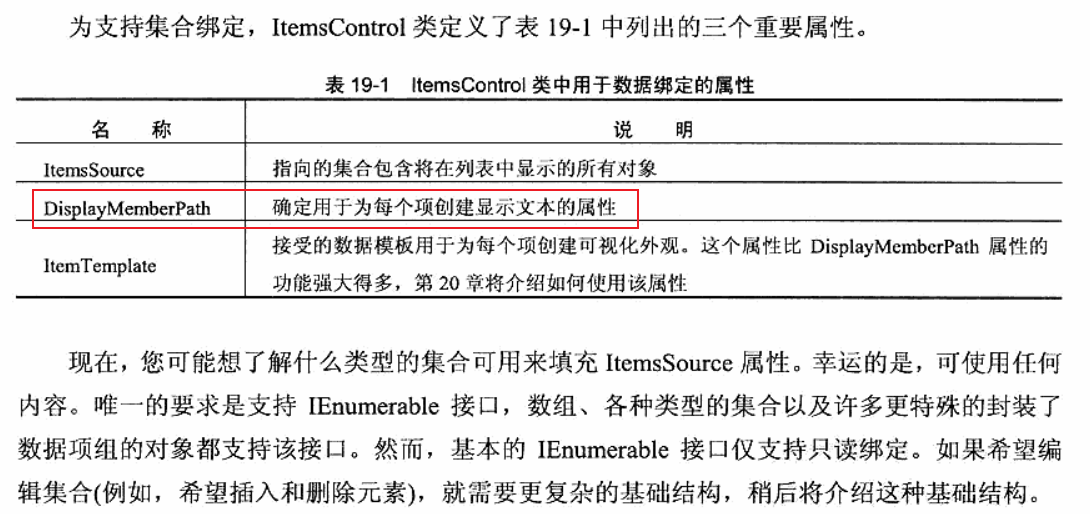
ItemsControl的重要属性

ItemsSource，用于设置显示的集合，集合只要支持IEnumerable接口即可。但仅实现IEnumerable接口的集合仅能只读绑定，若想插入和删除元素，还需实现其他接口。

DisplayMemberPath，指向元素的某个属性，用于设置每个元素在列表控件中呈现的文字。

ItemTemplate，定制数据模板，功能比DisplayMemberPath强大的多。

SelectedItem,选中的集合元素。



# 第五部分：路由事件和附加事件

# 第六部分：命令

# 第七部分：数据模板和控件模板

# 第八部分：资源和主题

# 第九部分：绘图和动画

# 第十部分：控件分类