**1.依赖项属性**

依赖项属性是WPF应用程序所特有的机制属性，它不同与普通的.NET属性，是一种特殊属性，实际上，它通常使用普通的.NET属性包装器进行封装。

* 为什么要引入依赖项属性？
* 高效的内存资源利用

在WPF程序中，一个Element元素类型可能拥有成百上千的属性，如Margin、FontSize等，但用户往往只使用其中的少许属性进行赋值配置就能够满足业务要求。

如果采用传统的.NET属性，那么对于创建一个拥有100个普通.NET属性的对象，需要占用足够的内存保存这100个属性值。如果我们只是使用其中的几个属性进行读写，实际上只访问内存中的少许数据块，而其他被占用的内存根本没有利用，这就极大的浪费了内存资源。

而WPF的Control控件元素（继承自Element元素）往往有许多属性，而通常我们只是使用其中的少量属性进行配置。因此，如果采用普通的.NET属性进行包装配置，则在创建大量的对象时将占用大量的内存，而事实上利用的资源只是少数。

因此，WPF引入了依赖项属性。所有依赖项属性都是静态的只读对象，并保存其属性对应字段的默认值。对于一个拥有依赖项属性的类型，编译器在编译时会将所有的依赖项属性对象编译保存，同时保存注册时指定的默认值。而在创建拥有者对象后，假如我们通过属性包装器读取某一个属性值，WPF依赖项属性管理器会查询该对象中依赖项属性对象二次赋值列表List中是否存在该依赖项属性信息：若无，说明该对象的依赖项属性未经过属性包装器二次赋值，因此，WPF依赖项属性管理器会使用静态的依赖项属性对象的默认值；若有，则说明属性被二次赋值，WPF依赖项属性管理器会使用List表中的属性值（实际上，在二次赋值时，WPF依赖项属性管理器会向List表添加该属性值）。

总之，WPF依赖项属性管理器会在属性值被二次赋值时在内存中分配资源保存该属性值，否则，使用依赖项属性的静态默认值。

* 支持数据绑定、动画和日志服务

依赖项属性具有更改通知和动态值识别的功能，是WPF数据绑定、动画以及样式的基础。支持WPF数据绑定、动画的属性包装器必须操作的是依赖项属性。

* 依赖项属性有什么特征？
* 它的命名规则：名称 = 普通属性包装器名称 + Property

如果一个依赖项属性的普通.NET包装器命名为TheName，则该依赖项属性应该命名为TheNameProperty。

* 它是静态的、公有的、只读的DependencyProperty对象

public static readonly DependencyProperty BufferSizeProperty;

* 由于依赖项属性是静态只读的，因此DependencyProperty对象只能在其归属的类的静态构造函数里面赋值

public class MyDependencyObjectClass : DependencyObject

{

public static readonly DependencyProperty BufferSizeProperty;

static MyDependencyObjectClass()

{

BufferSizeProperty = DependencyProperty.Register(...);

}

}

注意，由于DependencyProperty类型没有公有的构造函数，因此只能通过DependencyProperty的公有方法Register进行注册并返回新建的DependencyProperty对象。

* 所属对象必须是直接或者间接继承DependencyObject基类

如上面定义的依赖项属性BufferSizeProperty，它的所属类型MyDependencyObjectClass是直接继承DependencyObject类型。实际上，WPF的所有元素类型Element都是直接或者间接继承DependencyObject。

* 如何自定义依赖项属性？

以下是自定义的依赖项属性例子：

//定义继承DependencyObject基类的类型

public class MyDependencyObjectClass : DependencyObject

{

//定义依赖项属性

public static readonly DependencyProperty BufferSizeProperty;

/静态构造函数

static MyDependencyObjectClass()

{

//注册

BufferSizeProperty = DependencyProperty.Register(

"BufferSize", //属性包装器名称

typeof(long), //属性值类型

typeof(MyDependencyObjectClass), //拥有者类型

//依赖项属性附加功能配置，可选

new FrameworkPropertyMetadata(

(long)100 //默认值（new long()使用该类型的.NET默认值）

),

//验证回调函数，可选

new ValidateValueCallback(IsBufferSizeValid)

);

}

//定义验证回调函数

private static bool IsBufferSizeValid(object value)

{

if (((long)value) < 0)

return false;

return true;

}

//属性包装器

public long BufferSize {

get { return (long)base.GetValue(BufferSizeProperty); }

set { base.SetValue(BufferSizeProperty, value); }

}

}

* 如何使用依赖项属性？

因为依赖项属性封装在属性封装器中，可以通过操作属性封装器间接使用依赖项属性，举例如下：

public MainWindow()

{

InitializeComponent();

MyDependencyObjectClass obj0 = new MyDependencyObjectClass();

Obj0.BufferSize = 666;

MessageBox.Show(obj0.BufferSize.ToString()); //新值666

MyDependencyObjectClass obj1 = new MyDependencyObjectClass();

MessageBox.Show(obj1.BufferSize.ToString()); //默认值100

}

* 属性验证回调函数
* CoerceValueCallback委托

CoerceValueCallback委托代理的回调函数定义如下：

private static object fun(DependencyObject d, object value);

可以看出，该回调函数也是静态私有的，接收DependencyObject对象和属性值对象并返回调整后的值对象的新值。而CoerceValueCallback回调函数在元数据对象FrameworkPropertyMetadata的委托CoerceValueCallback进行声明，该声明是可选的。其经典的用法是最大值属性Maximum和最小值属性Minimum的值调整。

* ValidateValueCallback委托

ValidateValueCallback委托代理的回调函数定义如下：

private static bool fun(object value);

可以看出，该回调函数是静态私有的，接收值对象并返回bool值。而ValidateValueCallback回调函数在DependencyProperty.Register注册函数中声明，该声明是可选的：

//注册

BufferSizeProperty = DependencyProperty.Register(

"BufferSize", //属性包装器名称

typeof(long), //属性值类型

typeof(MyDependencyObjectClass), //拥有者类型

//依赖项属性附加功能配置，可选

new FrameworkPropertyMetadata(

(long)100 //默认值（new long()使用该类型的.NET默认值）

),

//验证回调函数，可选

**new ValidateValueCallback(IsBufferSizeValid)**

);

**//定义验证回调函数**

**private static bool IsBufferSizeValid(object value)**

**{**

**if (((long)value) < 0)**

**return false;**

**return true;**

**}**

上面的验证函数表明，若尝试给属性BufferSize赋值负值数值，则会发生运行时错误，因为该值不是有效值。

* PropertyChangedCallback事件

如果CoerceValueCallback回调函数和ValidateValueCallback回调函数验证通过后，将会触发PropertyChangedCallback事件。

PropertyChangedCallback事件代理的回调函数定义如下：

private static void fun(DependencyObject d,

DependencyPropertyChangedEventArgs e);

同样，PropertyChangedCallback回调函数在元数据对象FrameworkPropertyMetadata的中进行声明。

* 共享依赖项属性

上述定义的依赖项属性同样可以被其他的继承自DependencyObject类型的类共享，如下定义：

public class OtherClass : DependencyObject

{

public static readonly DependencyProperty BufferSizeProperty;

static OtherClass()

{

OtherClass.BufferSizeProperty = MyDependencyObjectClass.BufferSizeProperty.AddOwner(typeof(OtherClass));

}

}

这样，类MyDependencyObjectClass和OtherClass将使用同一个依赖项属性BufferSizeProperty。

* 附加属性

附件属性也是DependencyProperty类型，它是一种特殊的依赖项属性，它与后者的主要区别以下三点：

* 注册函数不同

附加属性通过DependencyProperty的RegisterAttached函数注册，而普通的依赖项属性通过Register函数注册。

* 封装方式不同

普通的依赖项属性通过.NET属性包装器进行封装，而附加属性则通过公有的静态函数进行封装。经典的案例是布局面板Grid的Row属性和Column属性，Row的封装在Grid类的内部定义如下：

public static int GetRow(UIElement element)

{

if (element == null)

{

throw new ArgumentNullException(...);

}

return (int)element.GetValue(Grid.RowProperty);

}

public static void SetRow(UIElement element)

{

if (element == null)

{

throw new ArgumentNullException(...);

}

element.SetValue(Grid.RowProperty, value);

}

Row的静态封装函数也指明，函数的命名规则如下：

* Set封装函数名 = Set + 注册时指定的普通属性名称
* Get封装函数名 = Get + 注册时指定的普通属性名称

下面的示例使用代码将元素放到Grid控件中的第一行：

Grid.SetRow(element, 0);

实际上，它调用的是element.SetRow(Grid.RowProperty, 0)。同时我们可以看出，上面的封装定义其实在Grid内部定义的，但其调用时却通过其他Element元素操作附加属性，这看起来有些奇怪，但是实际上其他Element元素和Grid都是继承自DependencyObject类，Element元素的属性操作函数GetValue和SetValue使用基类的方法，不同与依赖项属性，通过其他元素的属性操作函数来操作本体的附加属性，传入的DependencyProperty对象必须指明是哪个类型的，即上面的Grid.RowProperty，而不能写成RowProperty。