# 属性

## 属性的由来

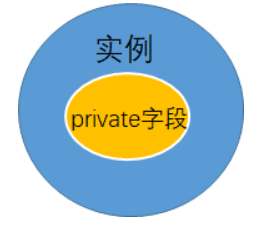
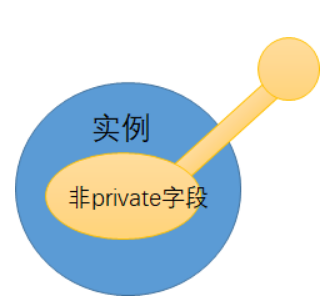
程序的本质是“数据+算法”，或者说是用算法来处理数据以期得到输出结果。在程序中，数据表现为各种各样的变量，算法则表现为各种各样的函数。被封装在类中的变量为字段（field），表示类或者实例的状态；被封装在类里的函数称为方法（Method），表示类或者实例的功能。字段和方法构成了最原始的面向对象封装。

C#中规定，对类有意义的字段和方法使用static关键字修饰，称为静态成员；

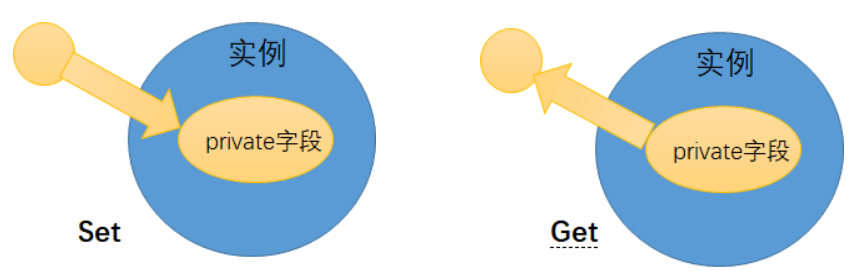
对类的实例有意义的字段和方法不加static关键字，称为非静态成员或实例成员。

静态字段在内存中只有一个拷贝，非静态字段则是每一个实例拥有一个拷贝；无论方法是否是静态的，在内存中只会有一个拷贝，区别只是你能用过类名来访问存放指令的内存还是通过实例名来访问存放之类的内存。

字段被封装在实例里，要么能被外接访问、要么不能。

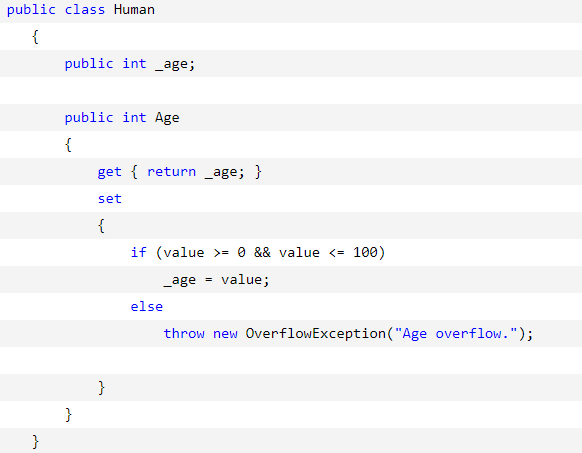
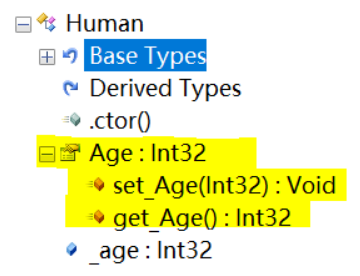
把数据直接暴露给外界是非常不安全的，如果每次写入数据的时候都先判断一下值的有效性又会增加代码冗余性并且违反面向对象要求“高内聚”的原则。一般我们会将字段标记为private，同时使用一对非private的方法来包装它。在这对方法中，一个以Set为前缀负责判断数据的有效性并写入数据，Get负责把字段里的数据读取出来。





安全包装

当.net推出时，微软更进一步把Get/Set这对方法合并称了属性（Property）。

这种.Net Framework中的属性又称为CLR属性（Common Language Runtime）。这样，可以说CLR属性是private字段的安全访问包装，也可以说一个private字段在后台支持(back)一个CLR属性。

在IL反编译器中打开编译结果可以看出，在C#代码中的属性的编译结果为两个方法，前面已经知道，再多实例方法也只有一个拷贝，所有CLR属性并不增加内存的负担。同时也说明，属性仅仅是个语法糖衣（Syntax Sugar）。

## 依赖属性

在WPF中，微软将属性进一步升华，推出依赖属性这个概念。MSDN这样定义：

Windows Presentation Foundation (WPF) 提供一组服务，这些服务可用于扩展类型的[属性](https://docs.microsoft.com/zh-cn/dotnet/standard/base-types/common-type-system" \l "Properties)的功能。 这些服务通常统称为 WPF 属性系统。 由 WPF 属性系统支持的属性称为依赖属性。（Windows Presentation Foundation (WPF) provides a set of services that can be used to extend the functionality of a type's [property](https://docs.microsoft.com/zh-cn/dotnet/standard/base-types/common-type-system#Properties). Collectively, these services are typically referred to as the WPF property system. A property that is backed by the WPF property system is known as a dependency property. ）。简言之，就是依赖属性是一种可以自己没有值，并通过使用Binding从数据源获得数值的属性。与传统的CLR属性和面向对象思想相比依赖属性有很多新颖之处：

1. 节省实例对内存的开销。

2. 属性值可以通过Bingding依赖在别的对象之上。

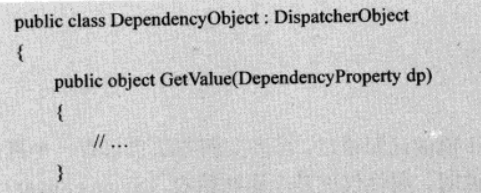
### 2.1 依赖属性对内存的使用方式

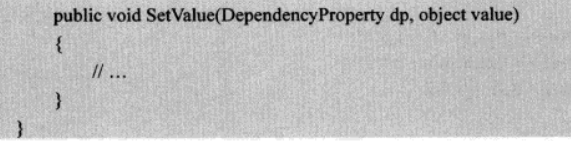
在传统的CLR属性中都包括一个非静态的字段，考虑一个这样的问题，TextBox有138个属性，假设每一个CLR属性都包装一个4字节的字段，如果程序在运行时候创建了10列1000行的一个TextBox列表，则这些字段将占有4\*138\*10\*1000 ，即约5.26M内存，在这些属性中，我们使用最多的也就是Text属性，这就意味着大多数内存都会被浪费掉。

如何避免这种浪费？我们考虑现实世界一个问题：如果一个登山队员，他的装备会很多，包括登山服、登山靴、护目镜、无线电、水、甚至氧气瓶等。如果去登珠峰，这些装备都需要带上，如果去登会稽山呢？氧气瓶不会再背着吧！所有我们实际中是：需要则带，不需要就不带，有必要的时候可以借用(共享经济啊）。

其实，这就是WPF中依赖属性的原理。传统的.NET开发中，一个对象的所占的内存空间在调用new操作的时候就已经决定了，而WPF允许对象在被创建的时候并不包含用于存储数据的空间(即字段所占用的内存)、只保留在需要用到数据时候获得默认值、借用其他对象数据或者实时分配空间的能力。我们称之为依赖对象(Dependency Object），而它这种实时获取数据的能力则依靠依赖属性(Dependency Property）来实现。在WPF开发中，必须使用依赖对象作为依赖属性的宿主，才能形成完整的Binding目标被数据所驱动。

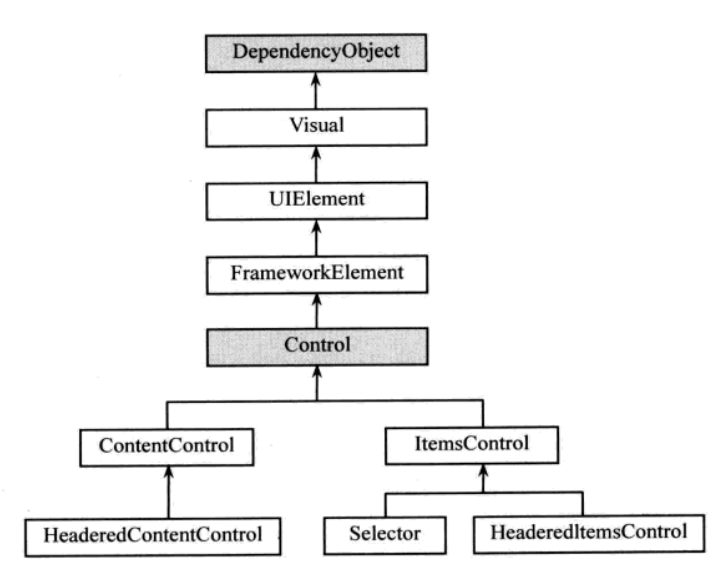
在WPF系统中，依赖对象的概念被DependencyObject类所实现，依赖属性的概念被DependencyPorperty类所实现。DependencyObject具有GetValue和SetValue两个方法：





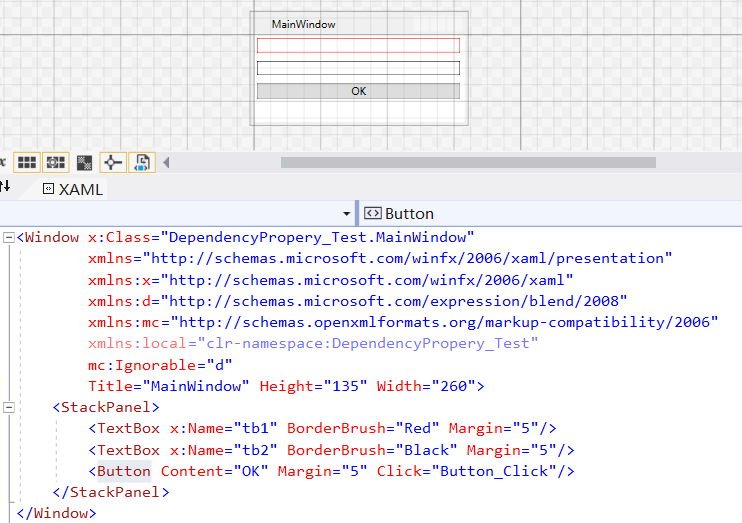
这两个方法都以DependencyProperty对象为参数，GetValue方法通过DependencyProperty对象获取数据；SetValue通过DependencyProperty对象存储值。

DependencyObject是WPF系统中相当底层的一个基类，WPF的所有UI控件都是依赖对象。

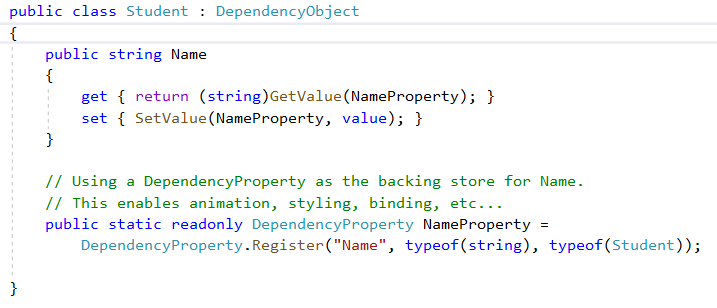


### 2.2声明和使用依赖属性

首先我们准备一个简单的界面：



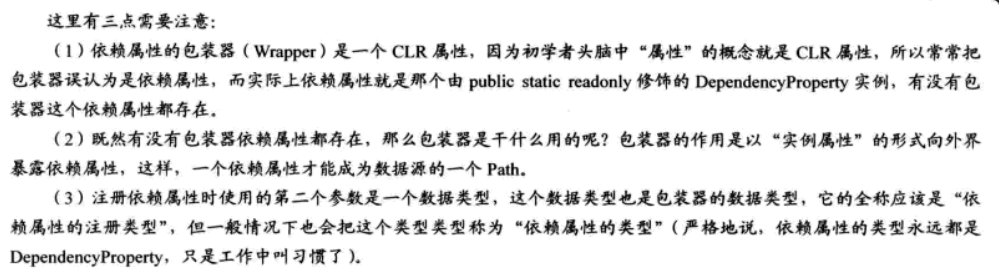
DependencyProperty必须以DependencyObject为宿主，借助它的SetvValue和GetValue方法进行写入和读取。因此自定义的DependencyProperty宿主一定是DependencyObject的派生类。



CLR属性包装

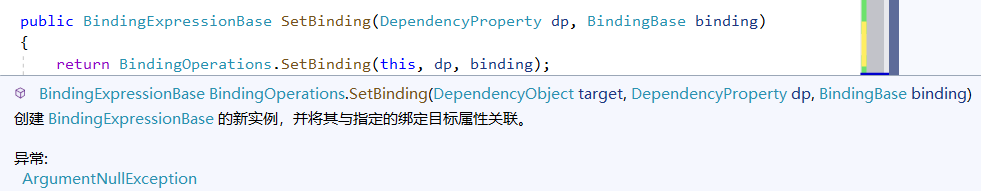
使用DependencyProperty.Register方法生成

变量由public static readonly修饰

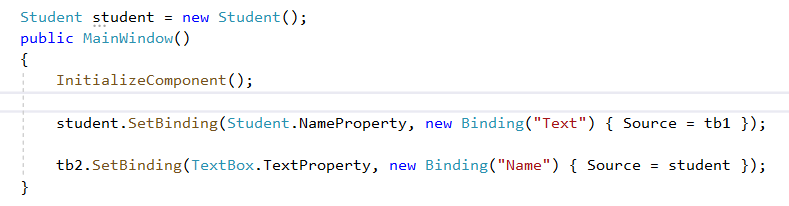


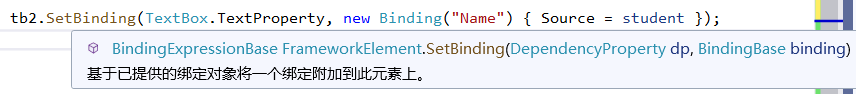
如果不关心底层的实现，下游的程序员在使用依赖属性时与使用单纯的CLR属性感觉别无二致。我们知道，依赖对象可以通过Binding依赖在其他对象上，即依赖对象是作为数据的目标而存在的。当依赖属性添加了CLR属性包装后，就相当于为依赖对象准备了用于暴露数据的Binding Path，现在的依赖对象已经具有了扮演数据源和数据目标双重角色的能力。注意：尽管Student类没有实现INotifyPropertyChanged接口，当属性的值发生改变时与之关联的Binding对象依然可以得到通知，依赖属性默认带有这样的功能，天生就是合格的数据源。





修改代码 如下：

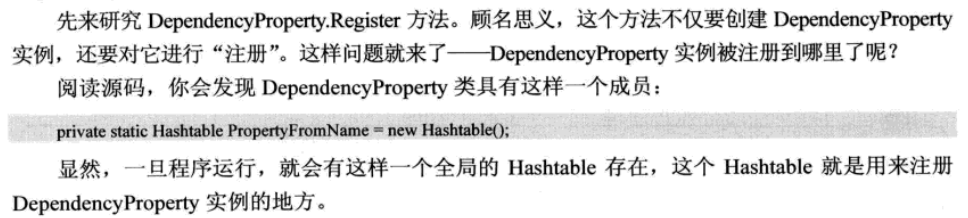


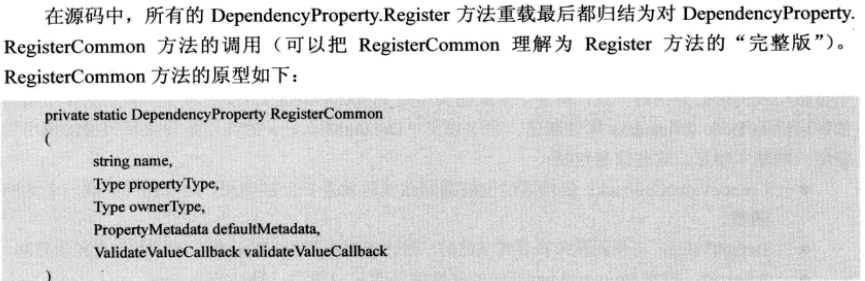


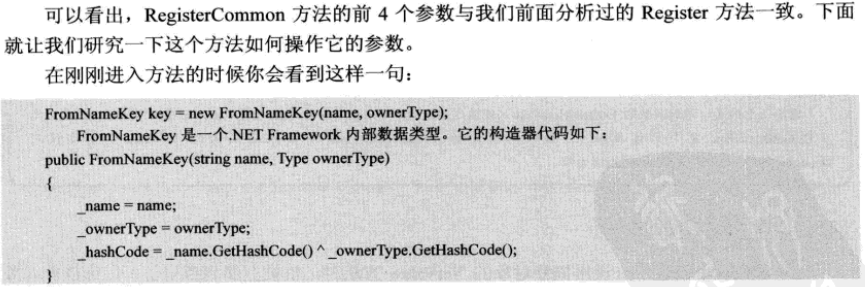
### 2.3依赖属性值存取的秘密

调用依赖对象的SetValue方法时，值被存储到哪里了？因为依赖对象的依赖属性是一个static对象，所以值不可能是保存到这个对象里，不然几百个实例都进行赋值时，到底应该保存哪个、丢掉哪个？显然，WPF有一套机制来存取依赖属性的值。下面就让我们来解析一下。

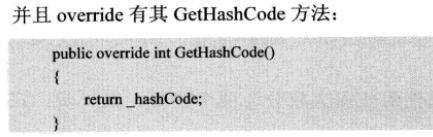
依赖属性的使用大致分为两个步骤：第一步，在DependencyObject派生类中声明public static readonly修饰的DependencyProperty成员变量，并使用DependencyProperty.Register方法获得DependencyProperty实例；第二步，使用DependencyObject 的SetValue和GetValue方法、借助DependencyProperty实例来存取值。因为我们要重点分析的就是DependencyPorperty.Register方法和DependencyObject的SetValue和GetValue方法。



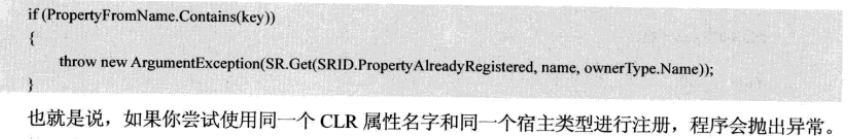




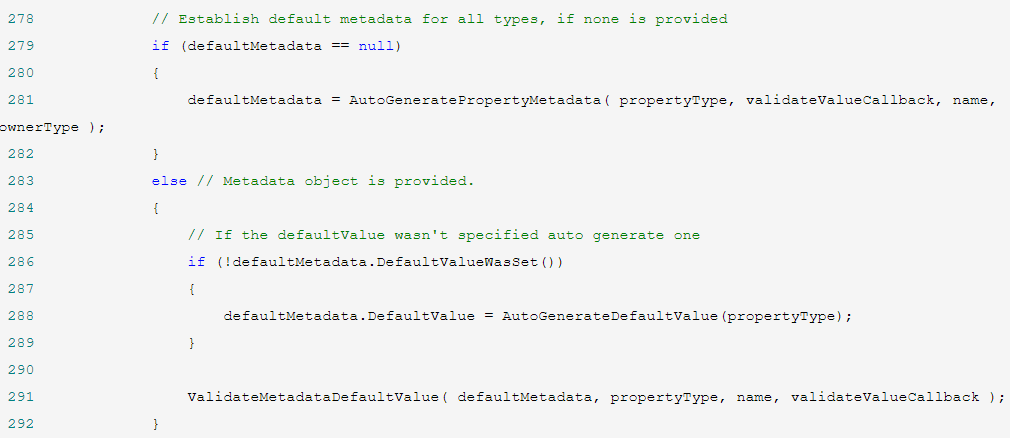
GetHashCode，



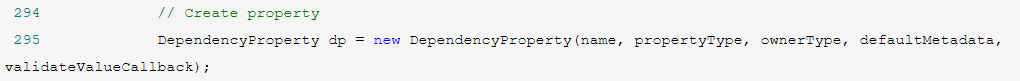
FromNameKey对象，即变量key的hashcode实际上是RegisterCommon第一个参数（CLR属性名字字符串）的hashcode与第3个参数（宿主类型）的hashcode做异或运算得来的。这样操作，每对“CLR属性名—宿主类型”所决定的DependencyProperty实例就是唯一的。所以，RegisterCommon方法里会发现这样的代码：



接下来，RegisterCommon检测是否提供了PropertyMetadate，如果没有提供则准备默认的一个PropertyMetadate。



当所有原料都准备妥当、没有问题后，DependencyProperty的实例被创建出来：

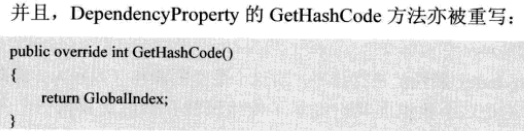
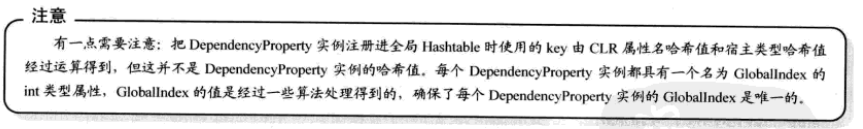


并且被注册进Hashtable中（Hashtable会自动调用key的GetHashcode方法获取其hashcode）：



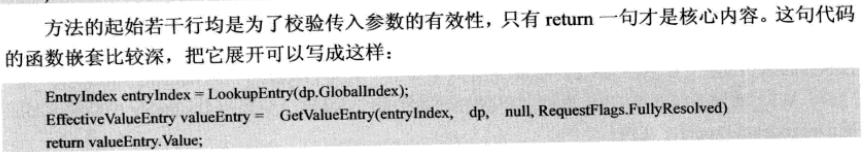
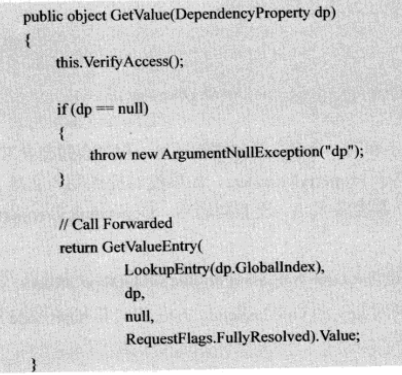
读到这里，我们可以用一句话概括DependencyProperty对象的创建与注册，那就是：创建一个DependencyProperty实例并用它的CLR属性名和宿主类型名生成hashcode，最后把hashcode和DependencyProperty实例作为Key-Value对存入全局的、名为PropertyFromName的Hashtable中。这样，WPF属性系统通过CLR属性名和宿主类型名就可以从这个全局的Hashtable中检索出对应的DependencyProperty实例。

最后，生成的DependencyProperty实例被当作返回值。



GlobalIndex属性值也就是DependencyProperty实例的哈希值，通过这个值就可以直接检索到某个DependencyProperty实例。

至此，一个DependencyProperty实例被创建并注册进一个全局的Hashtable中，下面就要使用DependencyObject的SetValue和GetValue，借助这个DependencyProperty实例保存和读取值。下面我们先看相对简单的GetValue方法，它的代码如下：

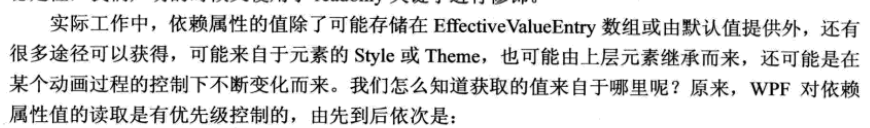


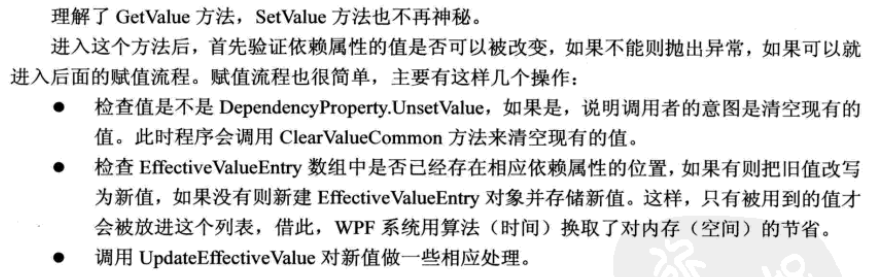
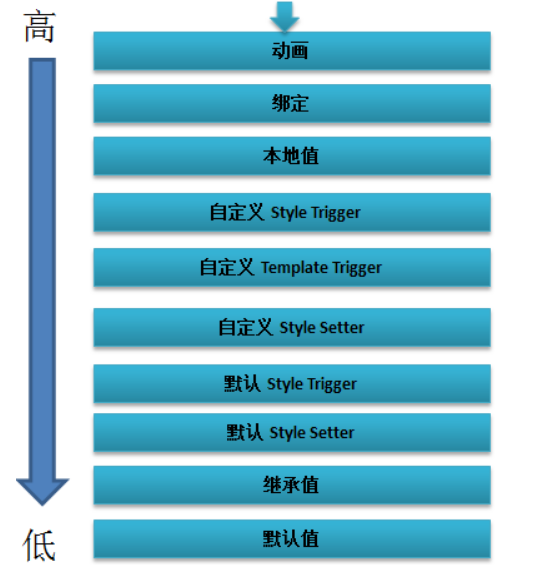
在这几句代码中屡次出现了Entry这个词，Entry是“入口”的意思。WPF的依赖系统在存放值的时候会把每一个有效值存放在一个“小房间”里，每个小房间都有自己的入口---检索算法只要找到这个入口、走进入口就能拿到依赖属性的值。这里说的小房间就是EffectiveValueEntry类的实例。EffectiveValueEntry的所有构造器都有包含一个DependencyProperty类型的参数，也可以这样说，每一EffectiveValueEntry都关联着一个DependencyProperty。EffectiveValueEntry类具有一个名为PropertyIndex的属性，这个属性的值实际上就是与之关联的DependencyProperty的GlobalIndex属性值。

在DependencyObject类的源码中可以找到这样的一个成员变量：

internal [EffectiveValueEntry](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://WindowsBase:4.0.0.0:31bf3856ad364e35/System.Windows.EffectiveValueEntry)[] [**EffectiveValues**](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://WindowsBase:4.0.0.0:31bf3856ad364e35/System.Windows.DependencyObject/property:EffectiveValues:System.Windows.EffectiveValueEntry%5b%5d);

这个数组依每个成员的PropertyIndex属性值进行排序，对这个数组的操作（如插入、删除和排序等）由专门的算法来完成。正是这个数组向我们提示了依赖属性存储值的秘密---每个DependencyObject实例都自带一个[EffectiveValueEntry](http://127.0.0.1/roeder/dotnet/Default.aspx?Target=code://WindowsBase:4.0.0.0:31bf3856ad364e35/System.Windows.EffectiveValueEntry)类型数组，当某个依赖属性的值要被读取时算法就会从这个数组中去检索值，如果数组中没有包含这个值，算法会返回依赖属性的默认值（这个值由依赖属性的DefaultMetadata来提供）。至此，我们明白了一件事情，那就是被static关键字所修饰的依赖属性值对象其作用是用来检索真正的属性值而不是存储值；被用来做检索键的值实际上是依赖属性的GlobalIndex属性（本质是其hashcode，而hashcode又由其CLR包装器名和宿主类型名共同决定），为了保证GlobalIndex属性值的稳定性，我们声明的时候又使用了readonly关键字进行修饰。

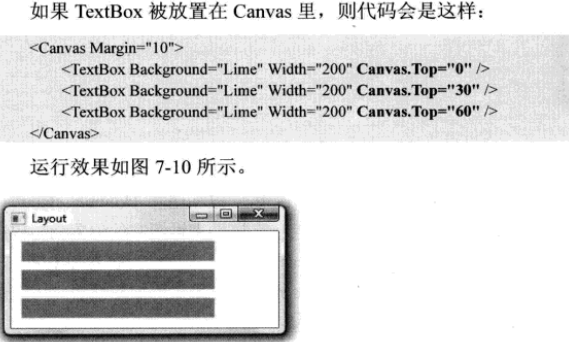




DependencyObject和DependencyProperty两个类是WPF属性系统的核心，本小节主要讲述它们之间的关系以及依赖属性值设置、读取的简要流程。Wpf系统的设计概念，即以public static类型的变量作为标记并以这个标记为索引进行对象的存储、访问、修改、删除等操作，这样的理念在传统的.NET开发体系中是不曾出现的，它是WPF体系的创新并广泛应用（路由事件、命令系统等）。

## 附加属性（Attached Properties）

附加属性是说一个属性本来不属于某个对象，但由于某种需求而被后来附加上。也就是把对象放入一个特定环境后，对象才具有的属性（表现出来就是被环境赋予的属性），称为附加属性。

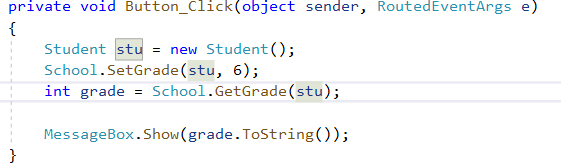


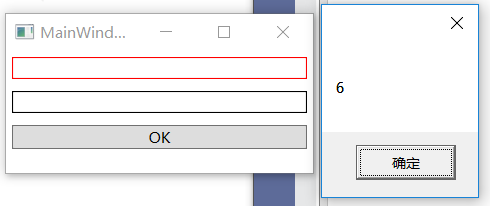
附加属性的作用就是将属性与数据类型（宿主）解耦，让数据类型的设计更加灵活。附加属性的本质是依赖属性，二者仅在注册和包装器上有一点区别。

人放在学校里会获取年级和班级两个属性，说明年级和班级两个属性是由学校附加给人的，因此这两个属性的真实所有者（宿主）应该是学校。我们准备一个名为School的类，并让它继承DependencyObject类，如下所示：

值会被保存到Student实例的EffectiveValueEntry数组里，只是用于在数组里检索值的依赖属性（即附加属性）并不是以Student类为宿主而是寄宿在School类里。没有关系的，因为CLR属性名和宿主类型名只是用来生成hashcode和GlobalIndex的。

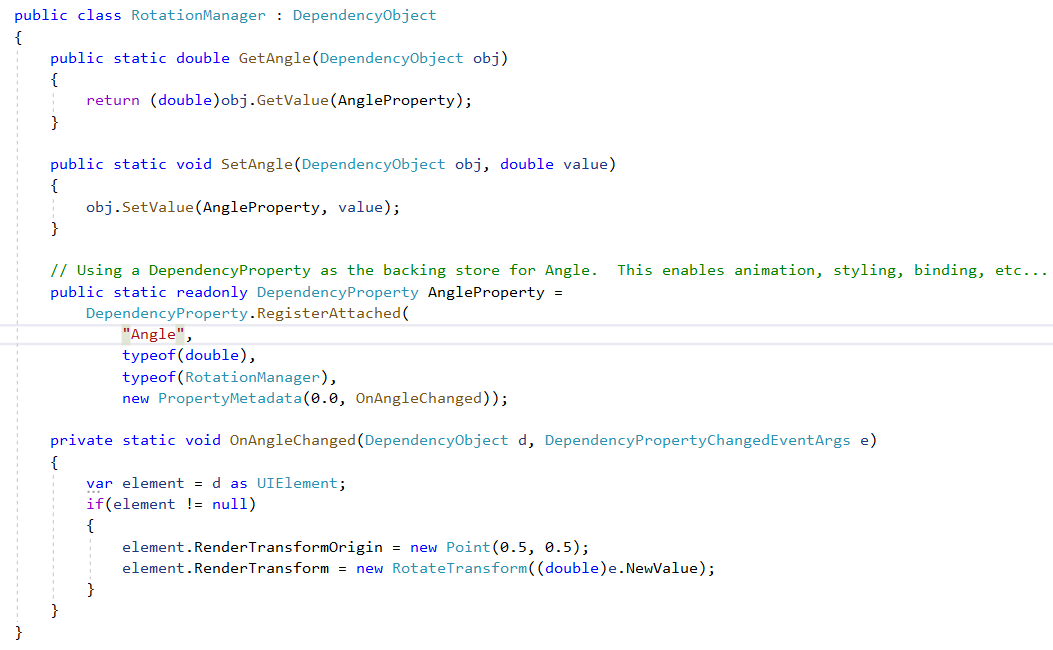
附加属性的包装器与依赖属性的不同，依赖属性使用CLR属性对GetValue和SetValue两个方法进行包装，附加属性则是使用两个方法分别进行包装。





附加属性的一个简单实例：

定义一个旋转类，实现控件角度的转动。



前端实现：

