[C#与闭包](https://kb.cnblogs.com/page/111231/)

作者: [jujusharp](http://www.cnblogs.com/jujusharp/)  来源: [博客园](http://www.cnblogs.com/)  发布时间: 2011-11-07 17:54  阅读: 21135 次  推荐: 29   [原文链接](http://www.cnblogs.com/jujusharp/archive/2011/08/04/2127999.html)   [[收藏]](javascript:add_to_wz())

　　首先想说明一点，虽然有这样那样的不好的心态(比如中文技术书)，但总体来说，国内的技术人员还是喜欢分享和教导别人的，这点我的个人感受和之前在园子里看到的朋友的感受恰恰相反。个人认为其实国内很多技术网友都是很热心的，可能因为语言问题同一个技术热点会稍稍落后国外一些，但一些成熟的或者基础的概念都可以找到很细致的中文介绍，特别是关于闭包。因为它的字面解释确实很绕，所以基本所有试图解释这一名词的同学都是尽量用自己认为最通俗易懂的方式来进行讲解。闲话扯远了，这里我就用C#语言来给大家解释下闭包吧。

　　其实要提到闭包，我们还得先提下变量作用域和变量的生命周期。

　　在C#里面，变量作用域有三种，一种是属于类的，我们常称之为field；第二种则属于函数的，我们通常称之为局部变量；还有一种，其实也是属于函数的，不过它的作用范围更小，它只属于函数局部的代码片段，这种同样称之为局部变量。这三种变量的生命周期基本都可以用一句话来说明，每个变量都属于它所寄存的对象，即变量随着其寄存对象生而生和消亡。对应三种作用域我们可以这样说，类里面的变量是随着类的实例化而生，同时伴随着类对象的资源回收而消亡(当然这里不包括非实例化的static和const对象)。而函数(或代码片段)的变量也随着函数(或代码片段)调用开始而生，伴随函数(或代码片段)调用结束而自动由GC释放，它内部变量生命周期满足先进后出的特性。

**那么这里有没有例外呢?**

　　答案是有的，不过在提这点之前，我还需要给各位另外一个名词。都说C#就是MS版本的Java，这话在.NET 1.0可能可以这么说，但自2.0之后C#就可以自豪的说它绝非java了，这里面委托有很大的功劳。如果用过Java和C#的人并且尝试过写WinForm程序时全部手写实现代码的人就会有这样一个感受，同样的click事件，在Java中必须要无端的套个匿名类，但在c#中，你是可以直接将函数名+=到事件之后而不需要显示写上匿名委托的对象类型的。因为编译器会帮你做这部分工作，在3.0和以后的版本之中，微软将委托的用法更是发挥的淋漓精致，无论是简洁的Lamda还是通俗易懂的LINQ，都是源自委托的。

**你可能要问，委托和我们今天要讲的闭包又有什么关系呢?**

　　我们知道，C#, Java和JavaScript, Ruby, Python这些语言不同，在C#和Java的世界里面，原子对象就是类(当然还有struct和基本变量)，而不是很多动态语言中的函数，我们可以实例化一个类，实例化一个变量，但不可以直接new 一个函数。也就是表面上看，我们是没办法像js那样将函数进行实例化和传递的。这也是为什么直到Java 7闭包才被姗姗来迟的加入Java特性中。但对C#来说这些只是表象，我刚学C#的时候，看到最多的解释委托的话就是：委托啊，就相当于C++里面的函数指针啦。这句话虽然笼统，但确实有一定道理，通过委托特别是匿名委托这层对象的包装，我们就可以突破无法将函数当做对象传递的限制了。

　　好像这里还是没讲到闭包和委托的关系，好吧，我太啰嗦了，下面从概念开始讲。

　　闭包其实就是使用的变量已经脱离其作用域，却由于和作用域存在上下文关系，从而可以在当前环境中继续使用其上文环境中所定义的一种函数对象。

　　好拗口，程序员，还是用示例来说明更好理解。

　　首先来个最简单的JavaScript中常常见到的关于闭包的例子：

function f1(){  
　　var n = 999;  
　　return function(){  
　　　 alert(n); // 999  
 return n;  
　　}  
}  
var a = f1();  
alert(a());

　　这段代码翻译成C#代码就是这样：

public class TCloser  
{  
 public Func<int> T1()  
 {  
 var n = 999;  
 return () =>  
 {  
 Console.WriteLine(n);  
 return n;  
 };  
 }  
}  
  
class Program  
{  
 static void Main()  
 {  
 var a = new TCloser();  
 var b = a.T1();  
 Console.WriteLine(b());  
 }  
}

　　从上面的代码我们不难看到，变量n实际上是属于函数T1的局部变量，它本来生命周期应该是伴随着函数T1的调用结束而被释放掉的，但这里我们却在返回的委托b中仍然能调用它，这里正是闭包所展示出来的威力。因为T1调用返回的匿名委托的代码片段中我们用到了n，而在编译器看来，这些都是合法的，因为返回的委托b和函数T1存在上下文关系，也就是说匿名委托b是允许使用它所在的函数或者类里面的局部变量的，于是编译器通过一系列动作(具体动作我们后面再说)使b中调用的函数T1的局部变量自动闭合，从而使该局部变量满足新的作用范围。

　　因此，如果你看到.NET中的闭包，你就可以像js中那样理解它，由于返回的匿名函数对象是在函数T1中生成的，因此相当于它是属于T1的一个属性。如果你把T1的对象级别往上提升一个层次就很好理解了，这里就相当于T1是一个类，而返回的匿名对象则是T1的一个属性，对属性而言，它可以调用它所寄存的对象T1的任何其他属性或者方法，包括T1寄存的对象TCloser内部的其他属性。如果这个匿名函数会被返回给其他对象调用，那么编译器会自动将匿名函数所用到的方法T1中的局部变量的生命周转期自动提升，并与匿名函数的生命周期相同，这样就称之为闭合。

　　也许你会说，这个返回的委托包含的变量n只是编译器通过某种方式隐藏的对这个委托对象的一个同样对象的赋值吧，那么我们再对比下面两个方法：

public class TCloser  
{  
 public Func<int> T1()  
 {  
 var n = 999;  
 Func<int> result = () =>  
 {  
 return n;  
 };  
 n = 10;  
 return result;  
 }  
  
 public dynamic T2()  
 {  
 var n = 999;  
 dynamic result = new { A = n };  
 n = 10;  
 return result;  
 }  
 static void Main()  
 {  
 var a = new TCloser();  
 var b = a.T1();  
 var c = a.T2();  
 Console.WriteLine(b());  
 Console.WriteLine(c.A);  
 }  
}

　　最后输出结果是什么呢？答案是10和999，因为闭包的特性，这里匿名函数中所使用的变量就是实际T1中的变量，与之相反的是，匿名对象result里面的A只是初始化时被赋予了变量n的值，它并不是n，所以后面n改变之后A并未随之而改变。这正是闭包的魔力所在。

　　你可能会好奇.NET本身并不支持函数对象，那么这样的特性又是从何而来呢？答案是编译器，我们一看IL代码便会明白了。

　　首先我给出C#代码：

public class TCloser  
{  
 public Func<int> T1()  
 {  
 var n = 10;  
 return () =>  
 {  
 return n;  
 };  
 }  
  
 public Func<int> T4()  
 {  
 return () =>  
 {  
 var n = 10;  
 return n;  
 };  
 }  
}

　　这两个返回的匿名函数的唯一区别就是返回的委托中变量n的作用域不一样而已，T1中变量n是属于T1的，而在T4中，n则是属于匿名函数本身的。但我们看看IL代码就会发现这里面的大不同了：

.method public hidebysig instance class [mscorlib]System.Func`1<int32> T1() cil managed{  
 .maxstack 3  
 .locals init (  
 [0] class ConsoleApplication1.TCloser/<>c\_\_DisplayClass1 CS$<>8\_\_locals2,  
 [1] class [mscorlib]System.Func`1<int32> CS$1$0000)  
 L\_0000: newobj instance void ConsoleApplication1.TCloser/<>c\_\_DisplayClass1::.ctor()  
 L\_0005: stloc.0  
 L\_0006: nop  
 L\_0007: ldloc.0  
 L\_0008: ldc.i4.s 10  
 L\_000a: stfld int32 ConsoleApplication1.TCloser/<>c\_\_DisplayClass1::n  
 L\_000f: ldloc.0  
 L\_0010: ldftn instance int32 ConsoleApplication1.TCloser/<>c\_\_DisplayClass1::<T1>b\_\_0()  
 L\_0016: newobj instance void [mscorlib]System.Func`1<int32>::.ctor(object, native int)  
 L\_001b: stloc.1  
 L\_001c: br.s L\_001e  
 L\_001e: ldloc.1  
 L\_001f: ret  
}  
   
.method public hidebysig instance class [mscorlib]System.Func`1<int32> T4() cil managed  
{  
 .maxstack 3  
 .locals init (  
 [0] class [mscorlib]System.Func`1<int32> CS$1$0000)  
 L\_0000: nop  
 L\_0001: ldsfld class [mscorlib]System.Func`1<int32> ConsoleApplication1.TCloser::CS$<>9\_\_CachedAnonymousMethodDelegate4  
 L\_0006: brtrue.s L\_001b  
 L\_0008: ldnull  
 L\_0009: ldftn int32 ConsoleApplication1.TCloser::<T4>b\_\_3()  
 L\_000f: newobj instance void [mscorlib]System.Func`1<int32>::.ctor(object, native int)  
 L\_0014: stsfld class [mscorlib]System.Func`1<int32> ConsoleApplication1.TCloser::CS$<>9\_\_CachedAnonymousMethodDelegate4  
 L\_0019: br.s L\_001b  
 L\_001b: ldsfld class [mscorlib]System.Func`1<int32> ConsoleApplication1.TCloser::CS$<>9\_\_CachedAnonymousMethodDelegate4  
 L\_0020: stloc.0  
 L\_0021: br.s L\_0023  
 L\_0023: ldloc.0  
 L\_0024: ret  
}

　　看IL代码你就会很容易发现其中究竟了，在T1中，函数对返回的匿名委托构造的是一个类，名称为newobj instance void ConsoleApplication1.TCloser/<>c\_\_DisplayClass1::.ctor()，而在T4中，则是仍然是一个普通的Func委托，只不过级别变为类级别了而已。

　　那我们接着看看T1中声明的类c\_\_DisplayClass1是何方神圣：

.class auto ansi sealed nested private beforefieldinit <>c\_\_DisplayClass1  
 extends [mscorlib]System.Object{  
 .custom instance void [mscorlib]System.Runtime.CompilerServices.CompilerGeneratedAttribute::.ctor()  
 .method public hidebysig specialname rtspecialname instance void .ctor() cil managed{}  
 .method public hidebysig instance int32 <T1>b\_\_0() cil managed{}  
 .field public int32 n  
}

　　看到这里想必你已经明白了，在C#中，原来闭包只是编译器玩的花招而已，它仍然没有脱离.NET对象生命周期的规则。它将需要修改作用域的变量直接封装到返回的类中，变成类的一个属性n，从而保证了变量的生命周期不会随函数T1调用结束而结束，因为变量n在这里已经成了返回的类的一个属性了。

　　看到这里我想大家应该大体上了解C#闭包的来龙去脉了吧。C#中，闭包其实和类中其他属性、方法是一样的，它们的原则都是下一层可以畅快的调用上一层定义的各种设定，但上一层则不具备访问下一层设定的能力。即类中方法里的变量可以自由访问类中的所有属性和方法，而闭包又可以访问它的上一层即方法中的各种设定。但类不可以访问方法的局部变量，同理，方法也不可以访问其内部定义的匿名函数所定义的局部变量。

　　这正是C#中的闭包，它通过超越Java语言的委托打下了闭包的第一步基础，随后又通过各种语法糖和编译器来实现如今在.NET世界全面开花的Lamda和LINQ，也使得我们能够编写出更加简洁优雅的代码。

　　附：后面是吐槽，与上文无关，大家可以略过，这篇文章其实两年之前在给同事讲C#闭包的时候就有想法整理出来和大家分享了，不过因为生活，工作，或许主要还是自己太懒的原因而拖着没动笔，到今天早上看到园友抱怨国内教书育人的氛围才最终决定利用晚上时间把它整理，然后放出来。我个人认为国内技术圈子的氛围尚可，虽然仍然有很多浮躁和易怒在圈子里徘徊，但我们想想国内IT人的生存空间就容易理解了。每天最理想的情况朝9晚6的干活，晚上加班，周末加班这些都是常事，而对我们而言，只要想写出一些经过细细思考的东西都至少需要2个小时以上，而且最好中间不要有人来打扰，这也就注定我们在白天工作时候很难完全有时间静下来组织语言，刨掉这些时间，留给我们自己的生活时间又有多少呢？所以我每次看到有园友发表帖子的时间是晚上1点，2点甚至更晚，都毫不意外，

　　我们并非专业写手，也不像国外IT人那样有充足的闲暇时光可以钻研自己的最爱，我们赚着他们的零头，买着比他们本子价格更贵的笔记本，担着比他们更高房价的压力来生活，这样的生活条件下我们这些可爱的社区(不仅限于cnblogs, javaeye, phpchina等)Geek们仍然如此活跃和热情，你还能抱怨什么呢？你要知道你看到的每篇文章(如果是工作人士的话)都是他们晚上从9点写到12点的生活点滴啊。

　　所以，以后不要抱怨国内IT氛围吧，相对这个社会其他各行各业的浮躁，我觉得我们的IT圈子已经是很乐于分享的一个群体了。而且除了因为“天下武功，源自欧美，滞后于英语国家”的缘故，我们有些技术确实要晚些才能跟上国外社区的脚步，但对于一些基础知识的解释，已经有很多中文的文章解释得很不错了。像我以前在理解闭包的时候， javaeye上看到的一大堆，像WIKI，像阮一峰的文章，我个人认为对中文用户是足够了。当然，这只是我个人的观点，大家不必较劲。

　　最后一点抱怨就是国内大大小小的抄袭网站，我想这也是影响我们中文用户查询资料的一个重要因素吧。以前曾经尝试过在baidu, google上搜索自己的文章，但结果相当令人失望，那些抄袭的网站从来都不在乎内容，因为这些可以通过抄来解决，而且不必带原文链接，不必表明作者。好像东西就是他们自己的一样，他们唯一在乎的就是SEO。这也导致我在使用google搜索的时候时常看到同一篇文章出现在某一页的所有搜索结果中，当然，网址是千奇百怪，实在让人无奈。有些网站即使标明了出处和作者，但用心略有险恶的不是给的链接，而是文字。而且,在这些抄袭者中，最让我感到悲哀的是大名鼎鼎的败毒文库，我至少看到不下5篇败毒文库里的文章是来自JE或者CSDN的，但在文库里面只有个文档，你看不到任何作者提示或者原文链接。也许有人会说，这也可能是作者自己上传的呀，但我个人认为这种可能性太小了，以国内IT人的风格，对败毒即使谈不上厌恶，也很少有主动去巴巴的。试想，一个国内最大的互联网公司都不尊重IT人的劳动(但愿我是错的吧)，你又能对其他人说什么呢? 同样看看国外，就我看到的DZone, WindowsPhoneGeek等，每个都是很明确的给出原文的链接，基本上我很少看到有引用别人文章不给原文链接的文章的。而正是这些不尊重作者劳动的网站对国内互联网资料搜索造成大量的垃圾信息。

**引用：**

　　维基百科： <http://zh.wikipedia.org/wiki/%E9%97%AD%E5%8C%85_%28%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6%29>  
　　博客园： <http://www.cnblogs.com/frankfang/archive/2011/08/03/2125663.html>