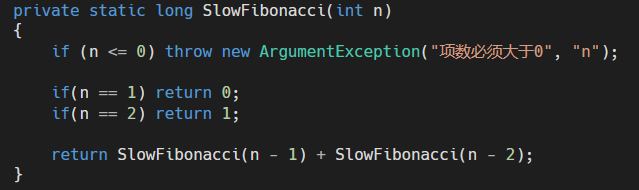
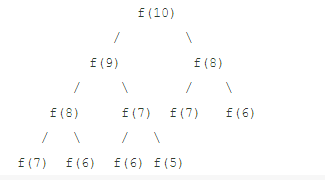
## C#里的闭包——从斐波那契数列的例子说起

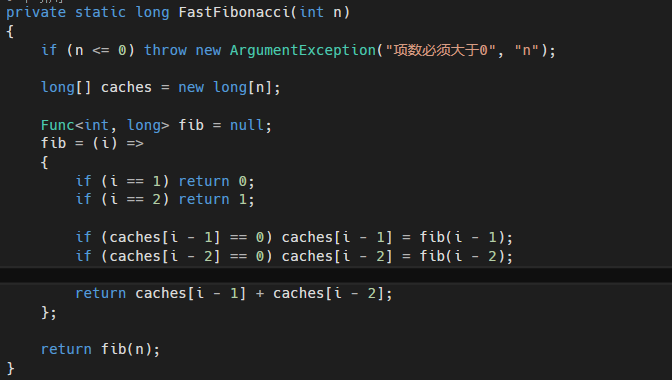
斐波那契数列对很多码农来说应该不陌生：F(1)=0，F(2)=1，F(n)=F(n-1)+F(n-2) (n为大于等于3的自然数)。如果用C#来编写一段代码，求任意第n项的值，很容易想到采用递归的方式：



递归代码很直接，意思也很明白，唯一的缺点是效率低。在我的机子上，当n=35的时候基本还可以秒出，n=40的时候已经明显感觉到慢，45以上就卡在那边不动了。这个低效版本的时间复杂度是O(2^n)。下面是计算F(10)的部分示意图：

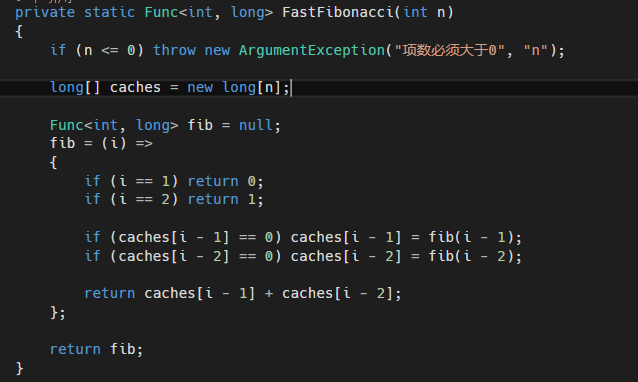


可见有大量的重复计算过程，而且重复计算量随着n呈指数增长。所以较好的解决办法是避免重新计算那些已经计算过的值，要将他们缓存起来，下面是另一个快速版本的实现：



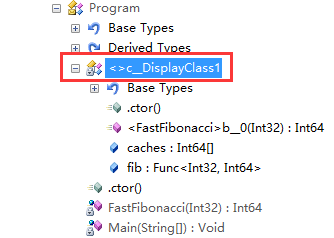
这个版本的关键代码在于fib这个Lambda表达式（匿名函数）里，他将所有已经计算过的值缓存在局部的caches数组里，每项只需计算一次。这个版本的时间复杂度是O(n)。看起来是典型的空间换时间的做法，但仔细想，这个版本在空间上的效率应该也比之前的递归版本要高，因为免去了大量的函数递归调用，如果n比较大，递归版本很可能会导致栈溢出。

上面举这个例子并不是要讨论斐波那契的数列的最优算法什么的（网上还有一种O(logn)的算法），而是为了引出：C#里的函数闭包。因为我觉得这个例子很优雅地在C#里利用了函数闭包。fib是个闭包，它保留了外层作用于里的caches变量。其实上面的代码还可以这样写：

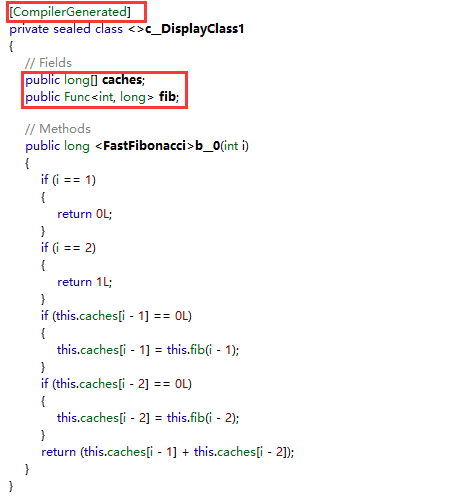


这时候计算F(n)的值需要这样调用：FastFibonacci(n)()，这样就更有函数式编程的味道了，而且也更容易看出fib是个闭包。闭包只是个通用的概念，并不是某个语言所特有的，只是各种语言有各自的实现方式而已。JavaScript通过作用域链实现，Java通过内部类实现，那么C#又通过什么方式实现？

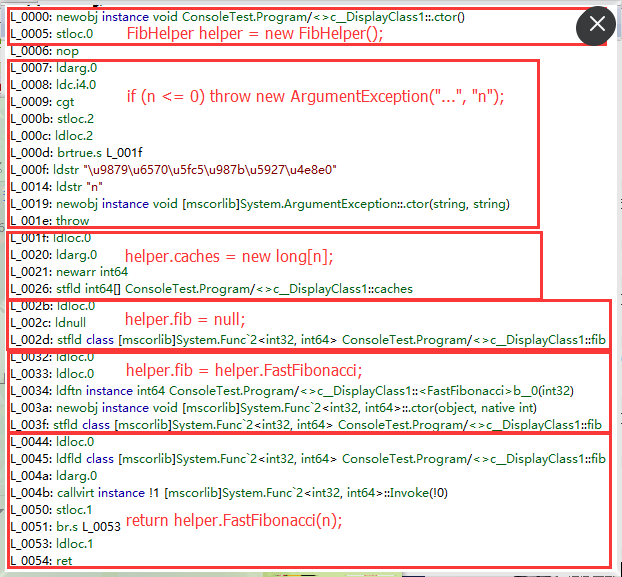
熟悉C#的.Netter应该知道在C#里Lambda表达式在编译后最终会成为类的某个方法。既然编译的时候会将Lambda表达式里的代码“挪到”另外的方法，那在上面的代码，fib里的代码又是如何访问在当前方法里的caches变量的？C#编译器又对我们的代码动了什么手脚。通过.Net Reflector反编译上面的FastFibonacci代码：



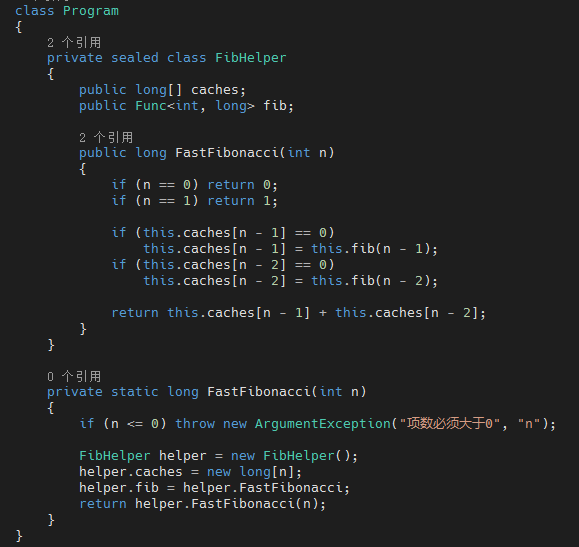
可以看到除了我们自己写的FastFibonacci方法外，还有一个由编译器生成的名字奇怪的内部类（红框）。它的代码如下：



这个类除了有两个caches和fib公共字段之外，还有一段熟悉的代码，它的逻辑和之前的fib闭包里的代码基本相差无几，其实这个方法和我们自己写的FastFibonacci作用是一样的。为了描述方便，下面把这个类叫做FibHelper，编译器生成的方法叫FastFibonacci。接下去是看我们自己写的FastFibonacci代码，由于.Net Reflector在反编译时尽最大可能还原了源代码，所以如果直接看C#代码，FastFibonacci的反编译代码基本没有变化。所以我们得直接看IL代码：



截图里的红字部分是翻译的IL代码，结合上面的FibHelper类，结果已经很清晰了。整理后的代码等价于：



所以说，C#里的闭包是通过内部类的方式实现的：通过内部类，将闭包变量的作用扩大。并且C#里的闭包只是个语法糖而已，CLR并不了解闭包。之所以要用内部类，可能是因为内部类支持访问外部类的私有成员，这样闭包也就同样可以访问类的私有变量了。回过头看前面的斐波那契额数列，其实我们完全可以自己用内部类实现个高效版本，没必要通过什么闭包。但这段代码好就好在优雅地利用了闭包，比反编译出来的代码可读性好多了。

总结：对于C#来说，闭包是个鲜有涉及的概念，可能是因为C#并不是一门专为函数式编程设计的语言。但是C#有委托，它使我们可以把函数当作一个普通对象来看待，这样就在一定程度上支持了函数式编程，而闭包在函数式编程里算是很重要的概念了。C#通过语法糖的方式实现闭包看起来稍微有点别扭，但是有时候对代码的可读性还是很有帮助的，很多时候我们在方法里写个Lambda表达式都意识不到自己在用闭包，比如这个斐波那契数列的例子。