### 我来说说JavaScript闭包

js里的闭包其实是一个函数（对象），这个函数对象包含（或者说保留了）其被创建时外层函数作用里的变量。本质上js的每个函数都是闭包，因为语言机制本身就允许内层函数访问外层函数作用域里的变量，比如下面的代码：

function counter(start){

return function(){return start++}

}

counter函数返回一个计数器，每次调用的时候会返回一个递增的值，这个计数器就是一个闭包，其中他保留了外层函数里的start变量。

像这样的代码在js里太常见了，以至于我们有时候会忽视闭包的一些陷阱。再看下面的代码：

var counters = [];

var start;

for(var i = 1; i <= 10; i++){

start = i;

var counter = function(){return start++}

counters.push(counter);

}

这段代码的本意是通过循环来产生10个计数器，分别从1～10开始计数，但是结果并不是期望的那样，每个计数器其实都是从10开始计数，正确的方式是使用之前的counter函数来产生计数器：

var counters = [];

for(var i = 1; i <= 10; i++){

var counter = counter(i);

counters.push(counter);

}

理解js闭包，归根结底需要理解js是如何解析变量(标识符)的。和一些静态语言(c,java,c#)通过栈来访问变量不同，js是通过“作用域链”来访问(解析)变量的，可以把它理解成一个个对象形成的链表，每个对象以属性的方式存储了一些可以供函数访问的变量。作用域链的头部对象由当前执行函数体内定义的变量构成的，尾部是全局对象。每个js函数被创建的时候，它会被绑定一个作用域链，如果是在父函数里创建的，那么作用域链就是父函数当前执行时的作用域链，如果函数是在全局环境下创建的，那么就是全局对象。在函数执行的时候，会在该函数已绑定作用域链的头部添加一个对象，这个对象用于存储在当前函数体声明的变量。当函数要解析一个标识符的时候，它会在作用域链的头部对象(就是当前函数的作用域)开始查找，依次往作用域链的尾部(全局对象)，直到找到为止。

回过头看上面通过循环来产生计数器的错误版本。首先循环代码是在全局环境下执行的，所以在循环里创建的计数器函数被绑定的作用域链就是全局对象，此时作用域链上有且仅有一个全局对象，全局对象上此时可以访问的变量除了自身有的属性外，准确地说还有4个：counters, start, i,counter(因为变量提升的关系，这四个变量最终都成为全局对象的属性)。可以看到循环结束后所有计数器绑定的作用域链都是一样的。如果这时候调用第一个计数器counters[0]，此时函数执行，js引擎会在之前绑定的作用域链的头部创建一个新的对象用来存储函数体内定义的变量，只是在这个例子中函数体内并没有定义任何变量，所以新创建的对象其实是个空对象。但是计数器的执行还是需要解析标识符的，就是start。通过前面的分析，计数器执行的时候作用域链上其实有两个对象：一个头部的空对象，一个尾部的全局对象，在头部的空对象上并不能找到名为start的属性，所以往后去全局对象上去找，可以找到start属性。那么此时全局对象上start属性的值是多少？答案是10。同理可知，所有其他的计数器在首次调时返回的值都是10(从10开始计数)，我们甚至可以断定这些计数器其实是相互影响的，就是说调用其中一个计数器，会使其他计数器在下次调用时产生不连续的值，因为他们操作的是同一个start！而通过第二种方法返回的计数器之所以正确是因为这个时候计数器绑定的作用域链是不同的，每个计数器创建的时候绑定了一个拥有2个对象的作用域链：从头至尾依次是counter函数执行时的头部对象，以及全局对象。而这个头部对象存储了一个变量i，关键在于每次调用counter函数时这个头部对象都是新创建的，此时i是递增的，计数器正是查找到了这个i。

js函数和一些静态语言比如C#里的函数的最大区别是，js的函数是动态创建的，C#是静态编译的(排除一些动态代码生成技术)。另外js函数和普通的对象没有什么区别，唯一的不同是它是可执行的。既然是动态创建的可执行对象，那么它的执行环境也必定要和创建时的执行环境相关，而不是静态的，这可能也是js采用作用域链的形式来访问变量的原因吧。