学习Javascript闭包（Closure）

作者： [阮一峰](http://www.ruanyifeng.com/)

日期： [2009年8月30日](http://www.ruanyifeng.com/blog/2009/08/)

闭包（closure）是Javascript语言的一个难点，也是它的特色，很多高级应用都要依靠闭包实现。

下面就是我的学习笔记，对于Javascript初学者应该是很有用的。

**一、变量的作用域**

要理解闭包，首先必须理解Javascript特殊的变量作用域。

变量的作用域无非就是两种：全局变量和局部变量。

Javascript语言的特殊之处，就在于函数内部可以直接读取全局变量。

　　var n=999;

　　function f1(){  
　　　　alert(n);  
　　}

　　f1(); // 999

另一方面，在函数外部自然无法读取函数内的局部变量。

　　function f1(){  
　　　　var n=999;  
　　}

　　alert(n); // error

这里有一个地方需要注意，函数内部声明变量的时候，一定要使用var命令。如果不用的话，你实际上声明了一个全局变量！

　　function f1(){  
　　　　n=999;  
　　}

　　f1();

　　alert(n); // 999

**二、如何从外部读取局部变量？**

出于种种原因，我们有时候需要得到函数内的局部变量。但是，前面已经说过了，正常情况下，这是办不到的，只有通过变通方法才能实现。

那就是在函数的内部，再定义一个函数。

　　function f1(){

　　　　var n=999;

　　　　function f2(){  
　　　　　　alert(n); // 999  
　　　　}

　　}

在上面的代码中，函数f2就被包括在函数f1内部，这时f1内部的所有局部变量，对f2都是可见的。但是反过来就不行，f2内部的局部变量，对f1就是不可见的。这就是Javascript语言特有的"链式作用域"结构（chain scope），子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

　　function f1(){

　　　　var n=999;

　　　　function f2(){  
　　　　　　alert(n);   
　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

**三、闭包的概念**

上一节代码中的f2函数，就是闭包。

各种专业文献上的"闭包"（closure）定义非常抽象，很难看懂。我的理解是，闭包就是能够读取其他函数内部变量的函数。

由于在Javascript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成"定义在一个函数内部的函数"。

所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

**四、闭包的用途**

闭包可以用在许多地方。它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

怎么来理解这句话呢？请看下面的代码。

　　function f1(){

　　　　var n=999;

　　　　nAdd=function(){n+=1}

　　　　function f2(){  
　　　　　　alert(n);  
　　　　}

　　　　return f2;

　　}

　　var result=f1();

　　result(); // 999

　　nAdd();

　　result(); // 1000

在这段代码中，result实际上就是闭包f2函数。它一共运行了两次，第一次的值是999，第二次的值是1000。这证明了，函数f1中的局部变量n一直保存在内存中，并没有在f1调用后被自动清除。

为什么会这样呢？原因就在于f1是f2的父函数，而f2被赋给了一个全局变量，这导致f2始终在内存中，而f2的存在依赖于f1，因此f1也始终在内存中，不会在调用结束后，被垃圾回收机制（garbage collection）回收。

这段代码中另一个值得注意的地方，就是"nAdd=function(){n+=1}"这一行，首先在nAdd前面没有使用var关键字，因此nAdd是一个全局变量，而不是局部变量。其次，nAdd的值是一个匿名函数（anonymous function），而这个匿名函数本身也是一个闭包，所以nAdd相当于是一个setter，可以在函数外部对函数内部的局部变量进行操作。

**五、使用闭包的注意点**

1）由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

2）闭包会在父函数外部，改变父函数内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

**六、思考题**

如果你能理解下面两段代码的运行结果，应该就算理解闭包的运行机制了。

代码片段一。

　　var name = "The Window";

　　var object = {  
　　　　name : "My Object",

　　　　getNameFunc : function(){  
　　　　　　return function(){  
　　　　　　　　return this.name;  
　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());

代码片段二。

　　var name = "The Window";

　　var object = {  
　　　　name : "My Object",

　　　　getNameFunc : function(){  
　　　　　　var that = this;  
　　　　　　return function(){  
　　　　　　　　return that.name;  
　　　　　　};

　　　　}

　　};

　　alert(object.getNameFunc()());

（完）

闭包是一个比较抽象的概念,尤其是对js新手来说.书上的解释实在是比较晦涩,对我来说也是一样.

　　但是他也是js能力提升中无法绕过的一环,几乎每次面试必问的问题,因为在回答的时候.你的答案的深度,对术语的理解以及js内部解释器的运作方式的描述,都是可以看出你js实际水平的.即使你没答对,也能让考官对你的水平有个评估.那么我先来说说我对js中的闭包的理解.

　　闭包是很多语言都具备的特性,在js中,闭包主要涉及到js的几个其他的特性:作用域链,垃圾(内存)回收机制,函数嵌套,等等.

　　在理解闭包以前.最好能先理解一下作用域链的含义,简单来说,作用域链就是函数在定义的时候创建的,用于寻找使用到的变量的值的一个索引,而他内部的规则是,把函数自身的本地变量放在最前面,把自身的父级函数中的变量放在其次,把再高一级函数中的变量放在更后面,以此类推直至全局对象为止.当函数中需要查询一个变量的值的时候,js解释器会去作用域链去查找,从最前面的本地变量中先找,如果没有找到对应的变量,则到下一级的链上找,一旦找到了变量,则不再继续.如果找到最后也没找到需要的变量,则解释器返回undefined.

　　了解了作用域链,我们再来看看js的内存回收机制,一般来说,一个函数在执行开始的时候,会给其中定义的变量划分内存空间保存,以备后面的语句所用,等到函数执行完毕返回了,这些变量就被认为是无用的了.对应的内存空间也就被回收了.下次再执行此函数的时候,所有的变量又回到最初的状态,重新赋值使用.但是如果这个函数内部又嵌套了另一个函数,而这个函数是有可能在外部被调用到的.并且这个内部函数又使用了外部函数的某些变量的话.这种内存回收机制就会出现问题.如果在外部函数返回后,又直接调用了内部函数,那么内部函数就无法读取到他所需要的外部函数中变量的值了.所以js解释器在遇到函数定义的时候,会自动把函数和他可能使用的变量(包括本地变量和父级和祖先级函数的变量(自由变量))一起保存起来.也就是构建一个闭包,这些变量将不会被内存回收器所回收,只有当内部的函数不可能被调用以后(例如被删除了,或者没有了指针),才会销毁这个闭包,而没有任何一个闭包引用的变量才会被下一次内存回收启动时所回收.

也就是说,有了闭包,嵌套的函数结构才可以运作,这也是符合我们的预期的.然后,闭包还有一些特性,却往往让程序员觉得很难理解.

看看下面一段代码.

[复制代码](javascript:void(0);)

var result=[];

function foo(){

var i= 0;

for (;i<3;i=i+1){

result[i]=function(){

alert(i)

}

}

};

foo();

result[0](); // 3

result[1](); // 3

result[2](); // 3

[复制代码](javascript:void(0);)

这段代码中,程序员希望foo函数中的变量i被内部循环的函数使用,并且能分别获得他们的索引,而实际上,只能获得该变量最后保留的值,也就是说.闭包中所记录的自由变量,只是对这个变量的一个引用,而非变量的值,当这个变量被改变了,闭包里获取到的变量值,也会被改变.

解决的方法之一,是让内部函数在循环创建的时候立即执行,并且捕捉当前的索引值,然后记录在自己的一个本地变量里.然后利用返回函数的方法,重写内部函数,让下一次调用的时候,返回本地变量的值,改进后的代码:

[复制代码](javascript:void(0);)

var result=[];

function foo(){

var i= 0;

for (;i<3;i=i+1){

result[i]=(function(j){

return function(){

alert(j);

};

})(i);

}

};

foo();

result[0](); // 0

result[1](); // 1

result[2](); // 2

[复制代码](javascript:void(0);)

越来越觉得国内没有教书育人的氛围，为了弄懂JS的闭包，我使出了我英语四级吃奶的劲去google上搜寻着有关闭包的解释，当我看到stackoverflow上这一篇解答，我脑中就出现了一句话：就是这货没跑了!   
  
不才译文见下，见笑了。   
  
Peter Mortensen问：   
  
就像老Albert所说的，“如果你不能向一个六岁的孩子解释清楚，那么其实你自己根本就没弄懂。”好吧，我试着向一个27岁的朋友就是JS闭包(JavaScript closure)却彻底失败了。   
  
你们会怎么把它解释给一个充满好奇心的六岁孩子听呢?   
  
注：我看过StackOverflow上给出的示例，但根本没用。   
  
Ali的回答：   
  
当function里嵌套function时，内部的function可以访问外部function里的变量。

复制代码代码如下:

function foo(x) {   
var tmp = 3;   
function bar(y) {   
alert(x + y + (++tmp));   
}   
bar(10);   
}   
foo(2)

不管执行多少次，都会alert 16，因为bar能访问foo的参数x，也能访问foo的变量tmp。   
  
但，这还不是闭包。当你return的是内部function时，就是一个闭包。内部function会close-over外部function的变量直到内部function结束。

复制代码代码如下:

function foo(x) {   
var tmp = 3;   
return function (y) {   
alert(x + y + (++tmp));   
}   
}   
var bar = foo(2); // bar 现在是一个闭包   
bar(10);

上面的脚本最终也会alert 16，因为虽然bar不直接处于foo的内部作用域，但bar还是能访问x和tmp。   
  
但是，由于tmp仍存在与bar闭包的内部，所以它还是会自加1，而且你每次调用bar时它都会自加1.   
  
(考虑到六岁这个限制：我们其实可以建立不止一个闭包方法，比如return它们的数组，也可以把它们设置为全局变量。它们全都指向相同的x和相同的tmp，而不是各自有一份副本。)   
  
注：现在来整点儿七岁的内容。   
  
上面的x是一个字面值(值传递)，和JS里其他的字面值一样，当调用foo时，实参x的值被复制了一份，复制的那一份作为了foo的参数x。   
  
那么问题来了，JS里处理object时是用到引用传递的，那么，你调用foo时传递一个object，foo函数return的闭包也会引用最初那个object!

复制代码代码如下:

function foo(x) {   
var tmp = 3;   
return function (y) {   
alert(x + y + tmp);   
x.memb = x.memb ? x.memb + 1 : 1;   
alert(x.memb);   
}   
}   
var age = new Number(2);   
var bar = foo(age); // bar 现在是一个引用了age的闭包   
bar(10);

不出我们意料，每次运行bar(10)，x.memb都会自加1。但需要注意的是x每次都指向同一个object变量——age，运行两次bar(10)后，age.memb会变成2.   
  
这和HTML对象的内存泄漏有关，呃，不过貌似超出了答题的范围。   
  
JohnMerlino 对Ali说：   
  
这里有一个不用return关键字的闭包例子：

复制代码代码如下:

function closureExample(obj, text, timedelay) {   
setTimeout(function() {   
document.getElementById(objID).innerHTML = text;   
}, timedelay);   
}   
closureExample(‘myDiv', ‘Closure is created', 500);

深夜1:37 John Pick这样回答：   
  
JS里的function能访问它们的：   
  
1. 参数   
  
2. 局部变量或函数   
  
3. 外部变量(环境变量?)，包括   
  
3.1 全局变量，包括DOM   
  
3.2 外部函数的变量或函数。   
  
如果一个函数访问了它的外部变量，那么它就是一个闭包。   
  
注意，外部函数不是必需的。通过访问外部变量，一个闭包可以维持(keep alive)这些变量。在内部函数和外部函数的例子中，外部函数可以创建局部变量，并且最终退出;但是，如果任何一个或多个内部函数在它退出后却没有退出，那么内部函数就维持了外部函数的局部数据。   
  
一个典型的例子就是全局变量的使用。   
  
mykhal这样回答：   
  
Wikipedia对闭包的定义是这样的：   
  
In computer science, a closure is a function together with a referencing environment for the nonlocal names (free variables) of that function.   
  
从技术上来讲，在JS中，每个function都是闭包，因为它总是能访问在它外部定义的数据。   
  
Since scope-defining construction in Javascript is a function, not a code block like in many other languages, what we usually mean by closure in Javascript is a fuction working with nonlocal variables defined in already executed surrounding function.   
  
闭包经常用于创建含有隐藏数据的函数(但并不总是这样)。

复制代码代码如下:

var db = (function() {   
// 创建一个隐藏的object, 这个object持有一些数据   
// 从外部是不能访问这个object的   
var data = {};   
// 创建一个函数, 这个函数提供一些访问data的数据的方法   
return function(key, val) {   
if (val === undefined) { return data[key] } // get   
else { return data[key] = val } // set   
}   
// 我们可以调用这个匿名方法   
// 返回这个内部函数，它是一个闭包   
})();   
db('x'); // 返回 undefined   
db('x', 1); // 设置data['x']为1   
db('x'); // 返回 1   
// 我们不可能访问data这个object本身   
// 但是我们可以设置它的成员

看了这么多外国大牛的解答，不知道你懂还是不懂，反正我是懂了。