特别说明

Javascript 闭包真经

继前阵子写完对象真经后，这篇文章我尝试尽力的去讲透 Js 中的闭包。这里要感谢爱民，爱民的书写得很好，我从中获益良多。不过这次我打算换一种思路来写这篇真经，就是采用提问回答的方式，我下面先提出我要回答的问题，如果读者你都很自信的能够回答上，那么就可以考虑干别的事情去了。如果感觉自己有点把握不准就请给我一步步的寻址吧。：）我保证最后你就会豁然开朗，明白闭包的真谛。问题集：

什么是函数实例?什么是函数引用?什么是闭包?闭包里有什么玩意?函数实例、函数引用和闭包有什么联系?闭包的产生的情形?闭包中的标识符的优先级是什么样的?闭包带来的可见性问题。

什么是函数实例呢?其实在我们平常书写代码的过程中，写的函数就是一段文本，只是对于编译型语言来说会把它编译为确定的二进制代码，并放到确定的内存位置执行，而对于 Js 这样的解释型语言，就会在程序运行的时候把这段文本翻译为计算机能懂的话。那么这里函数代码的文本其实就是这个函数的类，而 Js 引擎解释后的内存中的数据就是这个函数的实例了。所以函数的实例就是 Js 引擎读过这段代码后在内存中产生的一段数据。

什么是函数引用呢?函数引用就是指向刚才所说的那段内存数据的指针。也就是指向某个函数实例的指针。同一份实例可以有多个引用，只要该实例还有指向它的引用，占用的内存就不会被释放。

什么是闭包呢?闭包就是函数实例执行过程中动态产生的一块新的内存里的数据集。这句话有可以表达两层意思：

最初闭包必须由函数实例被调用(也就是函数实例执行)时才会由 Js 引擎动态生成；既然闭包也是一段内存区域，当没有依赖于这个内存中数据的引用时，才会被释放，也就是闭包理论上才会被销毁。记住：闭包是执行期的概念！

那闭包里有什么玩意呢?这里我要引入 Js 引擎在语法分析期就构造的两类结构。执行上下文结构(Context)和调用对象结构(CallObject)。我们还是先看一幅图片：

从上面的图片我们容易看出，这是描述一个函数实例信息的图。Context 结构包含了 myFunc 函数的类型(FUNCTION)、名称(myFunc)、形式参数(x,y,z)和最关键的 CallObject 结构的引用(body 为 CallObject)。而 CallObject 结构就是很详细的记录了这个函数的全部语法分析结构：内部声明的变量表(localDeclVars)、内部声明的具名函数表(localDeclFuncs)以及除去以上内容之外的其它所有代码文本的字符串表示(source)。

好，现在有了这个语法分析期就产生的 CallObject 结构，就好分析闭包里面会有什么了。当一个函数实例被调用时，实际上是从这个原型复制了一份 CallObject 结构到闭包那块空的内存中去。同时会给里面的数据赋值。这里有两个规则，大家要记下：

在函数实例开始执行时，localDeclVars 中的所有值将被置为 undefined；

在函数实例执行结束并退出时，localDeclVars 不被重置，这就是 Js 中函数能够在内部保存数据的特性的原因；这个闭包中的 CallObject 结构中的数据能保留多久取决于是否还有对其的引用存在，所以这里就引出了一个推论，闭包的生存周期不依赖于函数实例。那么对于全局对象呢?是什么样的结构呢?其实和上面的函数实例的基本一样，唯一不同的是，没有 Context，因为它在最顶层了。全局对象里面也有一个 CallObject，只是这个 CallObject 有点特殊，就是 localDeclVars 里面的数据只会初始化一次，且整个 CallObject 里的数据总不被销毁。

除了我讲的复制了一份 CallObject，再往里赋值，闭包其实还包括了一个 upvalue 数组，这里数组里装载的是闭包链上一级闭包中的标识符(localDeclVars 和 localDeclFuncs 及它自身的 upvalue 数组)的引用

函数实例、函数引用和闭包有什么联系呢?要回答这样的问题，就要好好分别理解上面说的这个三个分别指的是什么。然后我想在回答这个问题前，通过代码来感知一些东西。其实有时候人的思维需要一个载体去依靠，也就是我们常说的实践才能出真知。

[javascript]function myFunc(){this.doFunc=function(){}}var obj={}; //进入 myFunc，取得 doFunc()的一个实例 myFunc.call(obj)； //套取函数实例的一个引用并赋值给 func var func=obj.doFunc；//再次进入 myFunc，又取得了 doFunc()的一个新实例 myFunc.call(obj)；//比较两次取得的函数实例，结果显示 false，表明是不同的实例 print(func===obj.doFunc)；//显示 false//显示 true，表明两个函数实例的代码文本完全一样 print(func.toString()===obj.doFunc.toString())；//显示 true[/javascript]我两次调用了 myFunc，却发现对于相同的代码文本产生了不同的实例(func 和 obj.doFunc)。这是什么原因呢?事实上是我们每次调用 myFunc 函数时，Js 引擎进入 myFunc 函数，完成赋值操作时必然要解释那段匿名函数代码文本，所以以它为蓝本产生了一个函数实例。此后我们让 obj 对象的 doFunc 成员指向这个实例。多次调用，就多次的进入 myFunc 做赋值，就多次的产生新的同样的代码文本的实例，所以两次 obj.doFunc 成员所指的函数实例是不一样的(但函数实例的静态代码文本完全相同，这里都是一个匿名空函数)。再看一个实例与引用的例子：

[javascript]function MyObject(){}MyObject.prototype.method=function(){}; var obj1=new MyObject(); var obj2=new MyObject(); print(obj1.method===obj2.method); //显示 true[/javascript]这里的 obj1 和 obj2 的方法其实也是对一个函数实例的引用，但怎么就相同呢?其实这样回顾我在 Js 对象真经里说的，prototype 原型是一个对象实例，里面维护了自己的成员表，而 MyObject 的对象实例会有一个指针指向这个成员表，而不是复制一份。所以 obj1.method 和 obj2.method 实质都是执行同一个函数实例的两个引用。再来：

[javascript]var OutFunc=function(){//闭包 1 var MyFunc=function(){}; return function(){//闭包 2 return MyFunc; }}(); //注意这里的一个调用操作 var f1=OutFunc(); var f2=OutFunc(); print(f1===f2); //显示 true[/javascript]这个例子其实玩了一个视觉陷阱。如果你没有注意那个函数调用的()号，就会由前面讲的知识推导出 f1 和 f2 所指的实例不是同一个(尽管它们代码文本都一样)。但这个例子我想讲的其实是关于 upvalue 数组的问题，不会就忘记这个数组了吧。这里其实 OutFunc 最终成了一个匿名函数(function(){return MyFunc})实例的引用，而由于闭包 1 内

的 localDeclFuncs 里的数据有被引用，所以闭包 1 不会被销毁。然后调用这个 OutFunc 函数实例返回的是 MyFunc 函数的实例，但仅有一个，也就是多次调用都返回同一个，其原因在于匿名函数 function(){return MyFunc}实例其实是通过它运行时产生的闭包 2 中的 upvalue 中来找到 MyFunc 的(位于闭包 1 中)，而 MyFunc 在我的那个要你们特别留意的()操作时就产生了，在闭包 1 的 localDeclVars 中记录下来。

前面的函数实例都对应的是一个闭包，这里我想拿出一个函数实例可以对应多个闭包的例子来为等下回答函数实例、函数引用和闭包有什么联系做铺垫。

[javascript]var globalFunc; function myFunc(){//闭包 1 if(globalFunc){globalFunc(); }print('do myFunc: '+str); var str='test'; if(! globalFunc){globalFunc=function(){//闭包 2 print('do globalFunc: '+str); }}return arguments.callee; }myFunc()(); /*输出结果：do myFunc：undefined//第一次执行时显示 do globalFunc：test//第二次执行时显示 do myFunc：undefined//第二次执行时显示*/[/javascript]这个例子有点复杂，请读者耐着性质冷静的分析下。这里的 myFunc 函数实例被调用了两次，但都是同一个 myFunc 的实例。这是因为第一次调用 myFunc 实例后返回了一个它的引用(通过 return arguments.callee)。然后立即被()操作，完成第二次调用。在第一次调用中，str 还没有并赋值，但已经被声明了并被初始化为了 undefined，所以显示 do myFunc：undefined。而接着后面的语句，实现了赋值，这时候 str 的值为"test"，由于 globalFunc 为 undefined，所以赋值为一个匿名函数实例的引用。由于 globalFunc 是全局变量，属于不会销毁的全局闭包。所以这时候在第一次 myFunc 函数实例调用完毕后，全局闭包中的 globalFunc 所指的匿名函数实例引用了闭包 1 中的 str(值为 test)，导致闭包 1 不会被销毁，等第二次 myFunc 函数实例被调用时，产生了新的闭包 3，同时也会先声明变量 str，初始化为 undefined。由于全局变量 globalFunc 有值，所以会直接调用对应的闭包 1 中的那个匿名函数实例，产生了闭包 2，同时闭包 2 通过 upvalue 数组取得了闭包 1 中的 str，闭包 1 中当时为"test"，所以输出了 do globalFunc：test。注意，这里就体现了同一个实例多次调用都要产生新的闭包，且闭包中的数据不是共享的。

这个例子总的来说反应了几个问题：

Js 中同一个实例可能拥有多个闭包；Js 中函数实例与闭包的生存周期是分别管理的；Js 中函数实例被调用，总是会产生一个新的闭包，但上次调用产生的闭包是否已经销毁取决于那个闭包中是否有被其他闭包引用的数据。讲到这里，是时候揭晓谜团了。函数实例可以有多个函数引用，而只要存在函数实例的引用，该实例就不会被销毁。而闭包是函数实例被调用时产生的，但不一定随着调用结束就销毁，一个函数实例可以同时拥有多个闭包。

闭包有些什么产生的情形呢?其实细分可以分为：

全局闭包；具名函数实例产生的闭包；匿名函数实例产生的闭包；通过 new Function(bodystr)产生的函数实例的闭包；通过 with 语句所指示的对象的闭包。关于后面 3 种大家先不要急，我会在后面的闭包带来的可见性问题中举例说明。

闭包中的标识符的优先级是什么样的呢?要了解优先级，就要先说说闭包中有什么标识符。完整地说，函数实例闭包内的标识符系统包括：

thislocalDeclVars 函数实例的形式参数 argumentslocalDeclFuncs 我们先看一个例子：

[javascript]function foo1(arguments){print(typeof arguments); }foo1(1000); //显示 number function arguments(){print(typeof arguments); }arguments(); //显示 object function foo2(foo2){print(foo2); }foo2('hi'); //显示 hi[/javascript]从上面我们不难分析出，形式参数优先于 arguments(由 foo1 知)，内置对象 arguments 优先于函数名(由 arguments()知)，最后一个反应了形式参数优于函数名。其实有前面两个也说明了这点形式参数优于 arguments 优于函数名。再看一个例子：

[javascript]function foo(str){var str; print(str); }foo('test'); //显示 test function foo2(str){var str='not test'; print(str); }foo2('test'); //显示 not test[/javascript]这个例子可以得出一个结论：

当形式参数名与未赋值的局部变量名重复时，取形式参数；当形式参数名与有值的局部变量名重复时，取局部变量值。而 this 关键字，我们不能用它去做函数名，也不能作为形式参数，所以没有冲突，可以理解为它是优先级最高的。

闭包带来的可见性问题，这不是一个提问，而是一个对于闭包理解的实践。之所以这样说，是因为其实我们遇到的很多标识符可见性的问题，其实和闭包息息相关。比如内部函数可以访问外部函数中的标识符，完全是由于内部函数实例的闭包通过其 upvalue 数组来获得外部函数实例闭包中的数据的。可见性覆盖的实质就是在内部函数实例闭包中能找到对应的标识符，就不会去通过 upvalue 数组寻找上一级闭包里的标识符了。还有比如我们如果在一个函数里面不用 var 关键字来声明一个标识，就会隐式的在全局声明一个这样的标识。其实这就是因为在函数实例的所有闭包中找不到对应的标识，一直到了全局闭包中，也没有，所以 Js 引擎就在全局闭包(这个闭包有点特殊)声明了一个这样的标识来作为对于你的代码的一种容错，所以最好不要这样去用，容易污染全局闭包的标识符系统，要知道全局闭包是不会销毁的。

这里我想补充讲一个很重要的话题，就是闭包在闭包链中的位置怎么确定?

全局闭包没什么好说的，必然是最顶层的；对于具名函数实例产生的闭包，其实由该具名函数实例的静态语法作用域决定，也就是 Js 引擎做语法分析时，它处于什么语法作用域，那以后产生的闭包，在闭包链中也要按这个顺序，因为函数实例被执行时也会在这个位置去执行。也就是如果它在代码文本里表现为一个函数里的函数，那将来它的闭包就被外面函数实例产生的闭包包裹；注意：对于 SpiderMonkey 有点不同，在 with 语句里面的函数闭包链隶属于 with 语句打开的闭包。这在后面会特别举例。匿名函数实例的闭包位置由匿名函数直接量的创建位置决定，动态的加入闭包链中，而与它是否执行过无关，因为将来要产生闭包时，匿名函数直接量还是会回到创建位置执行；通过 new Function(bodystr)产生的函数实例的闭包很有意思，它不管在哪里创建，总是直接紧紧的隶属于全局闭包，也就是其 upvalue 数组里的数据是全局闭包中的数据；通过 with 语句所指示的对象的闭包位置由该 with 语句执行时的具体位置决定，动态的加入闭包链中。我们来看一些例子，用来说明上面的结论。

[javascript]var value='global value'; function myFunc(){var value='local value'; var foo=new Function('print(value)'); foo(); }myFunc(); //显示 global value var obj={}; var events={m1: 'clicked',m2: 'changed'}; for(e in events){obj[e]=function(){print(events[e]); }}obj.m1(); //显示相同 obj.m2(); //显示相同 var obj={}; var events={m1: 'clicked',m2: 'changed'}; for(e in events){obj[e]=new Function('print(events["'+e+'"])'); }obj.m1(); //显示 clicked obj.m2(); //显示 changed[/javascript]上面一大段代码显然可以分为 3 部分。第一部分说明了通过 new Function(bodystr)产生的函数实例的闭包在闭包链中的位置总是直接隶属于全局闭包，而不是 myFunc 的闭包。第二段显示相同的原因在于调用 m1、m2 时，它们各自的闭包都要引用全局闭包中的 e，这时 e 已经是 events 数组 for 迭代中的最后一个元素的索引了。第三段只用了 new Function 代替了原来的 function，就发生了变化，那是因为 Function 构造器传入的都是字符串，不会将来引用全局闭包的 e 了。

如果我们把闭包的可见性理解为闭包的 upvalue 数组和闭包内的标识符系统，那一般函数实例的闭包和通过 with 语句所指示的对象的闭包在前者上是一致的，而在后者上处理就不一样了。原因在于：通过 with 语句所指示的对象的闭包只有对象成员名可访问，而没有 this、函数形式参数、自动构建赋值的内置对象 arguments 以及 localDeclFuncs，而对于该闭包中的 var 声明变量的效果，就具体的引擎实现会有些差异，我觉得应该避免使用这种代码的写法，所以这里就不具体讨论了。看一些与 with 语句有关的代码：

[javascript]var x; with(x={x1: 'x1',x2: 'x2'}){x.x3='x3'; //通过全局变量 x 访问匿名对象 for(var iin x){print(x[i]); }}function self(x){return x.self=x; }with(self({x1: 'x1',x2: 'x2'})){self.x3='x3'; //通过匿名对象自身的成员 self 访问对象自身 for(var iin self){if(i! ='self')print(self[i]); }}[/javascript]上面的代码里注释都写得很明白了，我就不多说了，我们现在来看一个严重的问题：

[javascript]var obj={v: 10}; var v=1000; with(obj){function foo(){v*=3; }foo(); }/*with(obj){obj.foo=function(){v*=3; }obj.foo()

；}alert(obj.v)；//显示 30 alert(v)；//显示 1000*/alert(obj.v)；//显示 10 alert(v)；//显示 3000[/javascript]这段代码在 ie8、safari3.2、opera9 和 chrome 1.0.154.48 中都表示为 10 3000，但在 Firefox3.0.7 中就变为了 30 1000，这个就很特别了，也许是 bug 问题。这里就违背了我前面说的具名函数实例闭包在闭包链中位置的一般情况，即语法分析期就决定了，视 with 语句于透明。其实从代码的第一印象上说，似乎就是给 obj 对象的属性 v 乘了 3，而不是全局变量 v。其实在除了 Firefox3.0.7(其他版本的 FF 没有测试过)，foo 函数的闭包位置在语法分析期就决定了，直接隶属于全局闭包，而 with 语句打开的 obj 对象的闭包执行的位置在决定了它也直接隶属于全局闭包，所以就出现了并列的情况，那么 foo 函数里面的乘 3 自然就无法访问 with 闭包里的对象 obj.v 了，所以显示为 10 3000。如果想显示为 30 和 1000 我们完全可以利用匿名函数实例闭包位置的动态特性，就是直接创建量的位置决定闭包在闭包链中的位置。所以上面我注释的代码就可以很好的显示 30 和 1000，且 Firefox 也没有问题。注意匿名函数实例闭包位置与其是否执行过无关。下面代码说明这一点：

[javascript]function foo(){function foo2(){//foo2 的闭包 var msg='hello'；m=function(varName){//直接创建量赋值给 m 时，就决定了这个匿名函数实例【将来】的闭包隶属于 foo2 的闭包，所以可以通过 upvalue 数组访问 msg return eval(varName)；}}foo2()；var aFormatStr='the value is: ${msg}'；var m；var rx=/${(.*?)}/g;
print(aFormatStr.replace(rx,function([message],varName){return m(varName)；}))}foo()；//显示 hello[/javascript]好了，我要讲的讲完了，总之，闭包是执行期的概念。如果大家对于闭包还有什么问题或者对于本文有何高见，都欢迎给我留言、评论或者直接 email me。

[转摘：作者：志刚 hh 发表于 2011-03-22 17：16 原文链接

评论：0 查看评论发表评论

最新新闻：

- 比尔盖茨智斗垃圾邮件，返世界 13 亿千瓦时电力(2011-03-22 17：06)

- Opera Mini 6 和 Opera Mobile 11 上市(2011-03-22 16：54)
- 亚马逊 Appstore 将上线首推 3800 款应用(2011-03-22 16：53)
- 变革：联想发力移动互联网(2011-03-22 16：49)
- Firefox 4RC for Android/Maemo 发布(2011-03-22 16：40)

编辑推荐：程序员那些悲催的事儿

网站导航：博客园首页我的园子新闻闪存小组博问知识库