最近在网上查阅了不少**Javascript闭包**(closure)相关的资料，写的大多是非常的学术和专业。对于初学者来说别说理解闭包了，就连文字叙述都很难看懂。撰写此文的目的就是用最通俗的文字揭开Javascript闭包的真实面目。

**一、什么是闭包？**

“官方”的解释是：所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

相信很少有人能直接看懂这句话，因为他描述的太学术。我想用如何在Javascript中创建一个闭包来告诉你什么是闭包，因为跳过闭包的创建过程直接理解闭包的定义是非常困难的。看下面这段代码：

1. **function** a(){
2. **var** i=0;
3. **function** b(){
4. alert(++i);
5. }
6. **return** b;
7. }
8. **var** c = a();
9. c();

这段代码有两个特点：

1、函数b嵌套在函数a内部；

2、函数a返回函数b。

这样在执行完var c=a()后，变量c实际上是指向了函数b，再执行c()后就会弹出一个窗口显示i的值(第一次为1)。这段代码其实就创建了一个闭包，为什么？因为函数a外的变量c引用了函数a内的函数b，就是说：

当函数a的内部函数b被函数a外的一个变量引用的时候，就创建了一个闭包。

我猜想你一定还是不理解闭包，因为你不知道闭包有什么作用，下面让我们继续探索。

**二、闭包有什么作用？**

简而言之，闭包的作用就是在a执行完并返回后，闭包使得Javascript的垃圾回收机制GC不会收回a所占用的资源，因为a的内部函数b的执行需要依赖a中的变量。这是对闭包作用的非常直白的描述，不专业也不严谨，但大概意思就是这样，理解闭包需要循序渐进的过程。

在上面的例子中，由于闭包的存在使得函数a返回后，a中的i始终存在，这样每次执行c()，i都是自加1后alert出i的值。

那 么我们来想象另一种情况，如果a返回的不是函数b，情况就完全不同了。因为a执行完后，b没有被返回给a的外界，只是被a所引用，而此时a也只会被b引 用，因此函数a和b互相引用但又不被外界打扰(被外界引用)，函数a和b就会被GC回收。(关于Javascript的垃圾回收机制将在后面详细介绍)

**三、闭包内的微观世界**

如 果要更加深入的了解闭包以及函数a和嵌套函数b的关系，我们需要引入另外几个概念：函数的执行环境(excution context)、活动对象(call object)、作用域(scope)、作用域链(scope chain)。以函数a从定义到执行的过程为例阐述这几个概念。

1、当定义函数a的时候，js解释器会将函数a的作用域链(scope chain)设置为定义a时a所在的“环境”，如果a是一个全局函数，则scope chain中只有window对象。

2、当函数a执行的时候，a会进入相应的执行环境(excution context)。

3、在创建执行环境的过程中，首先会为a添加一个scope属性，即a的作用域，其值就为第1步中的scope chain。即a.scope=a的作用域链。

4、然后执行环境会创建一个活动对象(call object)。活动对象也是一个拥有属性的对象，但它不具有原型而且不能通过JavaScript代码直接访问。创建完活动对象后，把活动对象添加到a的作用域链的最顶端。此时a的作用域链包含了两个对象：a的活动对象和window对象。

5、下一步是在活动对象上添加一个arguments属性，它保存着调用函数a时所传递的参数。

6、最后把所有函数a的形参和内部的函数b的引用也添加到a的活动对象上。在这一步中，完成了函数b的的定义，因此如同第3步，函数b的作用域链被设置为b所被定义的环境，即a的作用域。

到此，整个函数a从定义到执行的步骤就完成了。此时a返回函数b的引用给c，又函数b的作用域链包含了对函数a的活动对象的引用，也就是说b可以访问到a中定义的所有变量和函数。函数b被c引用，函数b又依赖函数a，因此函数a在返回后不会被GC回收。

当函数b执行的时候亦会像以上步骤一样。因此，执行时b的作用域链包含了3个对象：b的活动对象、a的活动对象和window对象，如下图所示：

如图所示，当在函数b中访问一个变量的时候，搜索顺序是先搜索自身的活动对象，如果存在则返回，如果不存在将继续搜索函数a的活动对象，依 次查找，直到找到为止。如果整个作用域链上都无法找到，则返回undefined。如果函数b存在prototype原型对象，则在查找完自身的活动对象 后先查找自身的原型对象，再继续查找。这就是Javascript中的变量查找机制。

**四、闭包的应用场景**

1、保护函数内的变量安全。以最开始的例子为例，函数a中i只有函数b才能访问，而无法通过其他途径访问到，因此保护了i的安全性。

2、在内存中维持一个变量。依然如前例，由于闭包，函数a中i的一直存在于内存中，因此每次执行c()，都会给i自加1。

以上两点是闭包最基本的应用场景，很多经典案例都源于此。

**五、Javascript的垃圾回收机制**

在Javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收。如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。因为函数a被b引用，b又被a外的c引用，这就是为什么函数a执行后不会被回收的原因。

**执行上下文**

执行上下文是ECMAScript规范（ECMA 262 3rd edition）中定义的有关ECMAScript实现的相关行为要求。虽然规范没有具体规定执行上下文的实现，但是根据规范中对其所定义的数据结构（包含一些列相关是属性），因此执行上下文可以理解为（甚至按照）对象的方式来实现，尽管其包含的属性不是公共的。

所有的JavaScript代码都是在执行上下文中所执行的。全局代码（包含已执行的行内代码，JS文件，或者内潜入HTML的JS代码）都是在全局执行上下文中执行的，并且对于每一个函数（包含构造函数）都有一个与之相关联的执行上下文。使用eval函数执行的代码也是包含一个独特的执行上下文，但是由于其在JavaScript中的特殊性，这里就不再多考虑了。有关执行上下文的具体细则规范可以在ECMA 262 (3rd edition) 10.2节中找到。

当一个JavaScript函数被调用的时候，它就进入到一个执行上下文中。如果其中又有其他函数被调用（或者自身的递归）都会创建一个新的执行上下文，然后函数调用进入到那个执行上下文中，直到那个函数执行结束（译者注：这不就类似[盗梦空间](http://movie.douban.com/subject/3541415/)中的进入一层层梦境，直到该层梦境的时间结束，由此看来[盗梦空间](http://movie.douban.com/subject/3541415/)真是值得广大程序员好好研究的佳片啊=，=||）。因此，由于这种现象就在代码执行中形成了一个执行上下文堆栈。

当一个执行上下文被创建的时候会依次经历以下几个阶段。首先，在函数的执行上下文中创建一个Activation（激活）对象（貌似[犀牛书](http://book.douban.com/subject/2228378/)中就是这样翻译的~）。Activation对象实际是规范中又一机制。由于它拥有一些可访问的具名属性，因此可以将它作为一个对象来看待，但比较特殊的是，它没有原型对象（至少没有一个定义的原型对象），同时Activation对象也不能在代码中直接引用。

Activation对象创建完毕后，下一步就是为函数调用创建arguments对象，众所周知，arguments对象是一个类数组的对象，使用整数对其成员索引，且其排列顺序与函数调用传递的参数顺序一致。同时arguments对象还拥有length和callee两个成员（译者注：实际还应该有一个caller成员，不过这个成员已经建议弃用了），不过这里与主题无关就不做详细讨论了。这时候会在Activation对象上创建一个名为"arguments"的属性，且指向arguments对象。

接下来，执行上下文需要给作用域赋值。这里的作用域就是包含一些列的对象，也就是我们常说的作用域链，注意对“链”的理解。每一个函数对象在内部都拥有一个[[scope]]的属性（稍候会详细介绍），它也包含一个链式的对象。那么执行上下文的作用域主要包括Activation对象，且其处于作用域链的最顶端，之后就是函数对象对应的[[scope]]对象。

下一步称为变量实例化（variable instantiation），这时候使用到一个对象，在ECMA 262中称为Variable对象。然后Activation对象实际就是当作Variable对象使用（这里需要注意的是，其实两者指的是同一个对象）。在这个阶段里，会为函数的每一个形参在Variable对象上创建一个具名属性，如果函数调用传入的参数与形参一致的话，那么就会将参数的值逐一赋值给Variable对象上的那些属性（否则的话赋值为undefined）。之后对于调用函数的内部函数声明，同样首先会给这些函数创建函数对象，之后以函数名称为Variable对象的属性名，然后添加对应的函数对象引用。变量实例化的最后一个阶段就是处理调用函数的具备变量。同样的，将所有调用函数内的具备变量声明添加到Variable对象中。（译者注：此时的Variable对象就完整地包含了形参，内嵌函数以及局部变量）

这里需要注意的是在变量实例化阶段中，为Variable对象添加局部变量时，所赋值的初值均为undefined（译者注：因为我们知道JS引擎会将以var声明的具备变量提前解析，这就是刚刚所提到的变量实例化阶段所做的事儿，因此提前之后它们的初值都是undefined），而真正的局部变量实例化（赋值）直到在执行代码体的表达式时才会完成。

事实上，带有arguments的Activation对象与带有函数局部变量的Variable对象就是同一个对象，因此我们也可以把arguments标识符当作一个函数局部对象。

最后一个步骤就是对this关键字赋值。如果this关键字的值为一个对象的话，那么以this关键字为前缀的属性访问均是指向那个对象；但如果this关键字赋值为null的话，那么this关键字就指向全局对象。

对于全局执行上下文来说，处理的方式会有些许差异，这主要是因为其没有参数，因此就不需要定义Activation对象。同样的，全局执行上下文需要作用域，且它的作用域只包含一个对象——全局对象；全局执行上下文也需要经历变量实例化，它内部的函数自然就成为了最顶层的函数声明，包含了大量JavaScript代码；全局对象被当作Variable对象使用，因此其中的变量声明、函数声明自然就成为了全局对象的成员（译者注：这也就是为什么我们建议在全局代码中用一个匿名函数包裹内部代码，防止全局对象的污染）。

同时，全局执行上下文中的this对象指向的是全局对象。

最后以一张图片来结尾。

