[闭包（Closures）](http://www.cnblogs.com/TomXu/archive/2012/01/31/2330252.html)

**介绍**

闭包（closure）。闭包其实大家都已经谈烂了。尽管如此，这里还是要试着从理论角度来讨论下闭包，看看ECMAScript中的闭包内部究竟是如何工作的。

英文原文：http://dmitrysoshnikov.com/ecmascript/chapter-6-closures/

**概论**

在直接讨论ECMAScript闭包之前，还是有必要来看一下函数式编程中一些基本定义。

众所周知，在函数式语言中（ECMAScript也支持这种风格），函数即是数据。就比方说，函数可以赋值给变量，可以当参数传递给其他函数，还可以从函数里返回等等。这类函数有特殊的名字和结构。

**定义**

A functional argument (“Funarg”) — is an argument which value is a function.  
函数式参数（“Funarg”） —— 是指值为函数的参数。

例子：

function exampleFunc(funArg) {  
 funArg();  
}  
  
exampleFunc(function () {  
 alert('funArg');  
});

上述例子中funarg的实际参数其实是传递给exampleFunc的匿名函数。

反过来，接受函数式参数的函数称为高阶函数（high-order function 简称：HOF）。还可以称作：函数式函数或者偏数理或操作符。上述例子中，exampleFunc 就是这样的函数。

此前提到的，函数不仅可以作为参数，还可以作为返回值。这类以函数为返回值的函数称为带函数值的函数（functions with functional value or function valued functions）。

(function functionValued() {  
 return function () {  
 alert('returned function is called');  
 };  
})()();

可以以正常数据形式存在的函数（比方说：当参数传递，接受函数式参数或者以函数值返回）都称作 第一类函数（一般说第一类对象）。在ECMAScript中，所有的函数都是第一类对象。

*函数可以作为正常数据存在（例如：当参数传递，接受函数式参数或者以函数值返回）都称作第一类函数（一般说第一类对象）。*

在ECMAScript中，所有的函数都是第一类对象。

接受自己作为参数的函数，称为自应用函数（auto-applicative function 或者 self-applicative function）：

(function selfApplicative(funArg) {  
  
 if (funArg && funArg === selfApplicative) {  
 alert('self-applicative');  
 return;  
 }  
  
 selfApplicative(selfApplicative);  
  
})();

以自己为返回值的函数称为自复制函数（auto-replicative function 或者 self-replicative function）。通常，“自复制”这个词用在文学作品中：

(function selfReplicative() {  
 return selfReplicative;  
})();

自复制函数的其中一个比较有意思的模式是让仅接受集合的一个项作为参数来接受从而代替接受集合本身。

// 接受集合的函数  
function registerModes(modes) {  
 modes.forEach(registerMode, modes);  
}  
  
// 用法  
registerModes(['roster', 'accounts', 'groups']);  
  
// 自复制函数的声明  
function modes(mode) {  
 registerMode(mode); // 注册一个mode  
 return modes; // 返回函数自身  
}  
  
// 用法，modes链式调用  
modes('roster')('accounts')('groups')  
  
//有点类似：jQueryObject.addClass("a").toggle().removClass("b")

但直接传集合用起来相对来说，比较有效并且直观。

在函数式参数中定义的变量，在“funarg”激活时就能够访问了（因为存储上下文数据的变量对象每次在进入上下文的时候就创建出来了）：

function testFn(funArg) {  
 // funarg激活时, 局部变量localVar可以访问了  
 funArg(10); // 20  
 funArg(20); // 30  
  
}  
  
testFn(function (arg) {  
 var localVar = 10;  
 alert(arg + localVar);  
});

然而，我们从第14章知道，在ECMAScript中，函数是可以封装在父函数中的，并可以使用父函数上下文的变量。这个特性会引发funarg问题。

**Funarg问题**

在[面向堆栈的编程语言](http://en.wikipedia.org/wiki/Stack-oriented_programming_language)中，函数的局部变量都是保存在栈上的，每当函数激活的时候，这些变量和函数参数都会压入到该堆栈上。

当函数返回的时候，这些参数又会从栈中移除。这种模型对将函数作为函数式值使用的时候有很大的限制（比方说，作为返回值从父函数中返回）。绝大部分情况下，问题会出现在当函数有自由变量的时候。

*自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量*

例子：

function testFn() {  
  
 var localVar = 10;  
  
 function innerFn(innerParam) {  
 alert(innerParam + localVar);  
 }  
  
 return innerFn;  
}  
  
var someFn = testFn();  
someFn(20); // 30

上述例子中，对于innerFn函数来说，localVar就属于自由变量。

对于采用面向栈模型来存储局部变量的系统而言，就意味着当testFn函数调用结束后，其局部变量都会从堆栈中移除。这样一来，当从外部对innerFn进行函数调用的时候，就会发生错误（因为localVar变量已经不存在了）。

而且，上述例子在面向栈实现模型中，要想将innerFn以返回值返回根本是不可能的。因为它也是testFn函数的局部变量，也会随着testFn的返回而移除。

还有一个问题是当系统采用动态作用域，函数作为函数参数使用的时候有关。

看如下例子（伪代码）：

var z = 10;  
  
function foo() {  
 alert(z);  
}  
  
foo(); // 10 – 使用静态和动态作用域的时候  
  
(function () {  
  
 var z = 20;  
 foo(); // 10 – 使用静态作用域, 20 – 使用动态作用域  
  
})();  
  
// 将foo作为参数的时候是一样的  
(function (funArg) {  
  
 var z = 30;  
 funArg(); // 10 – 静态作用域, 30 – 动态作用域  
  
})(foo);

我们看到，采用动态作用域，变量（标识符）的系统是通过变量动态栈来管理的。因此，自由变量是在当前活跃的动态链中查询的，而不是在函数创建的时候保存起来的静态作用域链中查询的。

这样就会产生冲突。比方说，即使Z仍然存在（与之前从栈中移除变量的例子相反），还是会有这样一个问题： 在不同的函数调用中，Z的值到底取哪个呢（从哪个上下文，哪个作用域中查询）？

上述描述的就是两类funarg问题 —— 取决于是否将函数以返回值返回（第一类问题）以及是否将函数当函数参数使用（第二类问题）。

为了解决上述问题，就引入了 闭包的概念。

**闭包**

闭包是代码块和创建该代码块的上下文中数据的结合。

让我们来看下面这个例子（伪代码）：

var x = 20;  
  
function foo() {  
 alert(x); // 自由变量"x" == 20  
}  
  
// 为foo闭包  
fooClosure = {  
 call: foo // 引用到function  
 lexicalEnvironment: {x: 20} // 搜索上下文的上下文  
};

上述例子中，“fooClosure”部分是伪代码。对应的，在ECMAScript中，“foo”函数已经有了一个内部属性——创建该函数上下文的作用域链。

“lexical”通常是省略的。上述例子中是为了强调在闭包创建的同时，上下文的数据就会保存起来。当下次调用该函数的时候，自由变量就可以在保存的（闭包）上下文中找到了，正如上述代码所示，变量“z”的值总是10。

定义中我们使用的比较广义的词 —— “代码块”，然而，通常（在ECMAScript中）会使用我们经常用到的函数。当然了，并不是所有对闭包的实现都会将闭包和函数绑在一起，比方说，在Ruby语言中，闭包就有可能是： 一个过程对象（procedure object）, 一个lambda表达式或者是代码块。

对于要实现将局部变量在上下文销毁后仍然保存下来，基于栈的实现显然是不适用的（因为与基于栈的结构相矛盾）。因此在这种情况下，上层作用域的闭包数据是通过 动态分配内存的方式来实现的（基于“堆”的实现），配合使用垃圾回收器（garbage collector简称GC）和 引用计数（reference counting）。这种实现方式比基于栈的实现性能要低，然而，任何一种实现总是可以优化的： 可以分析函数是否使用了自由变量，函数式参数或者函数式值，然后根据情况来决定 —— 是将数据存放在堆栈中还是堆中。

**ECMAScript闭包的实现**

讨论完理论部分，接下来让我们来介绍下ECMAScript中闭包究竟是如何实现的。这里还是有必要再次强调下：ECMAScript只使用[静态（词法）作用域](http://en.wikipedia.org/wiki/Scope_(programming)#Lexical_scoping)（而诸如Perl这样的语言，既可以使用静态作用域也可以使用动态作用域进行变量声明）。

var x = 10;  
  
function foo() {  
 alert(x);  
}  
  
(function (funArg) {  
  
 var x = 20;  
  
 // 变量"x"在(lexical)上下文中静态保存的，在该函数创建的时候就保存了  
 funArg(); // 10, 而不是20  
  
})(foo);

技术上说，创建该函数的父级上下文的数据是保存在函数的内部属性 [[Scope]]中的。如果你还不了解什么是[[Scope]]，该章节对[[Scope]]作了非常详细的介绍。如果你对[[Scope]]和作用域链的知识完全理解了的话，那对闭包也就完全理解了。

根据函数创建的算法，我们看到 在ECMAScript中，所有的函数都是闭包，因为它们都是在创建的时候就保存了上层上下文的作用域链（除开异常的情况） （不管这个函数后续是否会激活 —— [[Scope]]在函数创建的时候就有了）：

var x = 10;  
  
function foo() {  
 alert(x);  
}  
  
// foo是闭包  
foo: <FunctionObject> = {  
 [[Call]]: <code block of foo>,  
 [[Scope]]: [  
 global: {  
 x: 10  
 }  
 ],  
 ... // 其它属性  
};

如我们所说，为了优化目的，当一个函数没有使用自由变量的话，实现可能不保存在副作用域链里。不过，在ECMA-262-3规范里任何都没说。因此，正常来说，所有的参数都是在创建阶段保存在[[Scope]]属性里的。

有些实现中，允许对闭包作用域直接进行访问。比如Rhino，针对函数的[[Scope]]属性，对应有一个非标准的 \_\_parent\_\_属性，在第12章中作过介绍：

var global = this;  
var x = 10;  
  
var foo = (function () {  
  
 var y = 20;  
  
 return function () {  
 alert(y);  
 };  
  
})();  
  
foo(); // 20  
alert(foo.\_\_parent\_\_.y); // 20  
  
foo.\_\_parent\_\_.y = 30;  
foo(); // 30  
  
// 可以通过作用域链移动到顶部  
alert(foo.\_\_parent\_\_.\_\_parent\_\_ === global); // true  
alert(foo.\_\_parent\_\_.\_\_parent\_\_.x); // 10

**所有对象都引用一个[[Scope]]**

这里还要注意的是：在ECMAScript中，同一个父上下文中创建的闭包是共用一个[[Scope]]属性的。也就是说，某个闭包对其中[[Scope]]的变量做修改会影响到其他闭包对其变量的读取：

*这就是说：所有的内部函数都共享同一个父作用域*

var firstClosure;  
var secondClosure;  
  
function foo() {  
  
 var x = 1;  
  
 firstClosure = function () { return ++x; };  
 secondClosure = function () { return --x; };  
  
 x = 2; // 影响 AO["x"], 在2个闭包公有的[[Scope]]中  
  
 alert(firstClosure()); // 3, 通过第一个闭包的[[Scope]]  
}  
  
foo();  
  
alert(firstClosure()); // 4  
alert(secondClosure()); // 3

关于这个功能有一个非常普遍的错误认识，开发人员在循环语句里创建函数（内部进行计数）的时候经常得不到预期的结果，而期望是每个函数都有自己的值。

var data = [];  
  
for (var k = 0; k < 3; k++) {  
 data[k] = function () {  
 alert(k);  
 };  
}  
  
data[0](); // 3, 而不是0  
data[1](); // 3, 而不是1  
data[2](); // 3, 而不是2

上述例子就证明了 —— 同一个上下文中创建的闭包是共用一个[[Scope]]属性的。因此上层上下文中的变量“k”是可以很容易就被改变的。

activeContext.Scope = [  
 ... // 其它变量对象  
 {data: [...], k: 3} // 活动对象  
];  
  
data[0].[[Scope]] === Scope;  
data[1].[[Scope]] === Scope;  
data[2].[[Scope]] === Scope;

这样一来，在函数激活的时候，最终使用到的k就已经变成了3了。如下所示，创建一个闭包就可以解决这个问题了：

var data = [];  
  
for (var k = 0; k < 3; k++) {  
 data[k] = (function \_helper(x) {  
 return function () {  
 alert(x);  
 };  
 })(k); // 传入"k"值  
}  
  
// 现在结果是正确的了  
data[0](); // 0  
data[1](); // 1  
data[2](); // 2

让我们来看看上述代码都发生了什么？函数“\_helper”创建出来之后，通过传入参数“k”激活。其返回值也是个函数，该函数保存在对应的数组元素中。这种技术产生了如下效果： 在函数激活时，每次“\_helper”都会创建一个新的变量对象，其中含有参数“x”，“x”的值就是传递进来的“k”的值。这样一来，返回的函数的[[Scope]]就成了如下所示：

data[0].[[Scope]] === [  
 ... // 其它变量对象  
 父级上下文中的活动对象AO: {data: [...], k: 3},  
 \_helper上下文中的活动对象AO: {x: 0}  
];  
  
data[1].[[Scope]] === [  
 ... // 其它变量对象  
 父级上下文中的活动对象AO: {data: [...], k: 3},  
 \_helper上下文中的活动对象AO: {x: 1}  
];  
  
data[2].[[Scope]] === [  
 ... // 其它变量对象  
 父级上下文中的活动对象AO: {data: [...], k: 3},  
 \_helper上下文中的活动对象AO: {x: 2}  
];

我们看到，这时函数的[[Scope]]属性就有了真正想要的值了，为了达到这样的目的，我们不得不在[[Scope]]中创建额外的变量对象。要注意的是，在返回的函数中，如果要获取“k”的值，那么该值还是会是3。

顺便提下，大量介绍JavaScript的文章都认为只有额外创建的函数才是闭包，这种说法是错误的。实践得出，这种方式是最有效的，然而，从理论角度来说，在ECMAScript中所有的函数都是闭包。

然而，上述提到的方法并不是唯一的方法。通过其他方式也可以获得正确的“k”的值，如下所示：

var data = [];  
  
for (var k = 0; k < 3; k++) {  
 (data[k] = function () {  
 alert(arguments.callee.x);  
 }).x = k; // 将k作为函数的一个属性  
}  
  
// 结果也是对的  
data[0](); // 0  
data[1](); // 1  
data[2](); // 2

**Funarg和return**

另外一个特性是从闭包中返回。在ECMAScript中，闭包中的返回语句会将控制流返回给调用上下文（调用者）。而在其他语言中，比如，Ruby，有很多中形式的闭包，相应的处理闭包返回也都不同，下面几种方式都是可能的：可能直接返回给调用者，或者在某些情况下——直接从上下文退出。

ECMAScript标准的退出行为如下：

function getElement() {  
  
 [1, 2, 3].forEach(function (element) {  
  
 if (element % 2 == 0) {  
 // 返回给函数"forEach"函数  
 // 而不是返回给getElement函数  
 alert('found: ' + element); // found: 2  
 return element;  
 }  
  
 });  
  
 return null;  
}

然而，在ECMAScript中通过try catch可以实现如下效果：

var $break = {};  
  
function getElement() {  
  
 try {  
  
 [1, 2, 3].forEach(function (element) {  
  
 if (element % 2 == 0) {  
 // // 从getElement中"返回"  
 alert('found: ' + element); // found: 2  
 $break.data = element;  
 throw $break;  
 }  
  
 });  
  
 } catch (e) {  
 if (e == $break) {  
 return $break.data;  
 }  
 }  
  
 return null;  
}  
  
alert(getElement()); // 2

**理论版本**

这里说明一下，开发人员经常错误将闭包简化理解成从父上下文中返回内部函数，甚至理解成只有匿名函数才能是闭包。

再说一下，因为作用域链，使得所有的函数都是闭包（与函数类型无关： 匿名函数，FE，NFE，FD都是闭包）。

这里只有一类函数除外，那就是通过Function构造器创建的函数，因为其[[Scope]]只包含全局对象。

为了更好的澄清该问题，我们对ECMAScript中的闭包给出2个正确的版本定义：

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域。
2. 从实践角度：以下函数才算是闭包：
   1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）
   2. 在代码中引用了自由变量

**闭包用法实战**

实际使用的时候，闭包可以创建出非常优雅的设计，允许对funarg上定义的多种计算方式进行定制。如下就是数组排序的例子，它接受一个排序条件函数作为参数：

[1, 2, 3].sort(function (a, b) {  
 ... // 排序条件  
});

同样的例子还有，数组的map方法是根据函数中定义的条件将原数组映射到一个新的数组中：

[1, 2, 3].map(function (element) {  
 return element \* 2;  
}); // [2, 4, 6]

使用函数式参数，可以很方便的实现一个搜索方法，并且可以支持无限制的搜索条件：

someCollection.find(function (element) {  
 return element.someProperty == 'searchCondition';  
});

还有应用函数，比如常见的forEach方法，将函数应用到每个数组元素：

[1, 2, 3].forEach(function (element) {  
 if (element % 2 != 0) {  
 alert(element);  
 }  
}); // 1, 3

顺便提下，函数对象的 apply 和 call方法，在函数式编程中也可以用作应用函数。 apply和call已经在讨论“this”的时候介绍过了；这里，我们将它们看作是应用函数 —— 应用到参数中的函数（在apply中是参数列表，在call中是独立的参数）：

(function () {  
 alert([].join.call(arguments, ';')); // 1;2;3  
}).apply(this, [1, 2, 3]);

闭包还有另外一个非常重要的应用 —— 延迟调用：

var a = 10;  
setTimeout(function () {  
 alert(a); // 10, after one second  
}, 1000);

还有回调函数

//...  
var x = 10;  
// only for example  
xmlHttpRequestObject.onreadystatechange = function () {  
 // 当数据就绪的时候，才会调用;  
 // 这里，不论是在哪个上下文中创建  
 // 此时变量“x”的值已经存在了  
 alert(x); // 10  
};  
//...

还可以创建封装的作用域来隐藏辅助对象：

var foo = {};  
  
// 初始化  
(function (object) {  
  
 var x = 10;  
  
 object.getX = function \_getX() {  
 return x;  
 };  
  
})(foo);  
  
alert(foo.getX()); // 获得闭包 "x" – 10

**总结**

本文介绍了更多关于ECMAScript-262-3的理论知识，而我认为，这些基础的理论有助于理解ECMAScript中闭包的概念。如果有任何问题，我回在评论里回复大家。

**其它参考**

* [Javascript Closures (by Richard Cornford)](http://jibbering.com/faq/notes/closures/)
* [Funarg problem](http://en.wikipedia.org/wiki/Funarg_problem)
* [Closures](http://en.wikipedia.org/wiki/Closure_%28computer_science%29)