**【原创】javascript closures**

偶最近很懒，不想多说什么，用代码来代表偶要说的话...:P  
不过多少人能够完全懂? 这，就不得而知了...^^  
  
A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).   
  
1、对象上下文(The contexts of the objects)  
  
   提示：您可以先修改部分代码再运行  
  
2、函数和运行环境上下文(Functional and execution contexts)  
  
   提示：您可以先修改部分代码再运行  
  
3、构造闭包(Forming closures)  
  
   提示：您可以先修改部分代码再运行  
  
4、异步调用(Call later or call back)  
  
   提示：您可以先修改部分代码再运行  
  
5、对象事件绑定(Associating events with object instance methods)  
  
   提示：您可以先修改部分代码再运行  
  
6、集合遍历器(Collection Iterator)  
  
   提示：您可以先修改部分代码再运行

closure，闭包？  
  
以前看过到一篇文章，没有仔细去理解  
  
写在前面的话：  
试图翻译自 <http://jibbering.com/faq/faq_notes/closures.html>  
文中大量提到《ECMA 262 》，我也没时间读这东西，可能有问题理解有误。希望纠正。  
只译了前边部分，我得理解几天再继续下去。  
英文水平差，凑合看吧。  
国内找了半天没这篇文章中文版，献丑了就。  
读后有种豁然开朗的感觉，清楚了很多javascript的问题。  
  
  
一、Introduction   
Closure (闭包)  
A "closure" is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).   
  
    闭包是ECMAScript(javascript)语言强大的特征之一，如果没有真正的理解它的概念，不可能很好使用它。在一般浏览器环境中，它们很容易被建立，但也会造成比较难理解的代码逻辑。为了避免闭包引起的缺点，利用它所提供的优点，明白它的机制是重要的。javascript语言的闭包很大程度上依靠 scope chains(函数，变量的范围链) 和 javascript对象的灵活的属性机制 实现。  
    闭包简单的解释是，ECMAScript允许inner functions(嵌套函数)：函数可以定义在另外一个函数里面（关于嵌套函数可以看看<javascript权威指南>）。这些内部的函数可以访问outer function（父函数）的local变量，参数，其它内部函数。当内部函数被构造，并可以在函数外被获得（函数当成返回值），这个内部函数被在 outer function返回后被执行（在outer函数外执行），那一个闭包形成了。（简单的理解，function被当成数据类型传递或动态执行）。 inner function还有权利访问 那些outer function（父函数）的local变量，参数，其它内部函数。那些outer function（父函数）的local变量，参数，其它内部函数在outer function返回前就有值，并返回的inner function需要改变这些值。  
  
我估计以下代码就是一个闭包。  
  
  
  
< script >   
var g\_count = 0 ;  
function aaa(p)  { // outer function   
  var outer\_count = 0 ;  
  function innerfun(name,count)  { // outer function中定义的另外一个inner function   
     return  name + ' : ' + count + ' 次; ' ;  
  }   
   return  function()  { // 返回一个匿名的inner函数   
      var inner\_count = 0 ;  
             return  innerfun( ' g\_count ' ,( ++ g\_count)) + innerfun( ' outer\_count ' ,( ++    outer\_count))+innerfun('inner\_count',(++inner\_count))+p;  
  }   
}   
  
  var fun1=aaa("fun1");  
var fun2=aaa("fun2");  
alert(fun1)  
alert(fun1());//这时候才真正执行  
  alert(fun2());  
</script>  
    不幸的，完全明白闭包需要知道它背后的机制和一些技术细节。  
二、The Resolution of Property Names on Objects (javascript对象的属性)  
    ECMAScript认可两类对象，“Native Object”和“Host Object”,Host Object属于Native Object的子类，在(ECMA 262 3rd Ed Section 4.3)中叫"Built-in Object"（内置对象）。Native objects属于语言级别，host objects被环境提供（浏览器等），例如，document objects,DOM nodes等。  
    关于对象属性的存取，数据类型，原型对象prototype的使用，我这就不译了。  
    可以参见我的另一篇文章  
  
  
三、Identifier Resolution, Execution Contexts and Scope Chains   
1、The Execution Context   
    执行环境上下文（Execution Context）是个抽象的概念，the ECMSScript specification (ECMA 262 3rd edition) to define the behaviour required of ECMAScript implementations。规范没有说 execution contexts 应该怎样实现，但规范中提到execution contexts是个关联属性结构，因此你可以假想为一个有属性的对象，但不是公有的（public）。  
    所有的javascript代码在一个execution context中执行。Global代码(.js文件中)在我叫做globla execution context中执行，每个函数的调用有个专属的execution context。注意，用eval函数执行的代码有个独特的execution context.(原文中说eval函数没常被程序员应用，确实如果掌握闭包使用后，还是可以避免一些eval使用的)。在section 10.2 of ECMA 262 (3rd edition)中详细讲述的execution context.  
    当一个函数被调用，那相应的execution context被建立，如果另外的函数（或同一个函数递归调用），那新的execution context被建立，直到函数return(对于递归调用，execution context是独立的)。因此，javascript代码的执行会建立很多的execution contexts.  
    当一个函数的execution context被建立(javascript中有global和function两种，eval没讨论)，按照顺序，有几个事情要发生。  
   （1）在一个函数的execution context中，一个"Activation"对象被建立（我在其它文章中叫调用对象）。the activation被另外规范解释。你可以把它当成一个对象，因为它有对象的属性，但它不是一般对象，它没有原型对象，并不能被javascript代码直接引用。  
   （2）建立一个arguments对象，它和数组类似，以整数为索引来访问值，表示函数的参数。它有length和callee属性。这个arguments对象被当成activation对象的属性。在函数内可以直接访问得到。  
   （3）下一步，execution context被分配一个 scope属性（scope chain后面讲到，我们可以把scope理解成对象的一个scope属性，值是scope chain）。一个scope由一列对象组成（或叫chain）。每个函数对象也有由chain组成的scope属性。函数的scope＝Activation object+上级对象的scope的属性.（这里的scope可以理解成servlet中的chain,一系列请求组成的链。）  
   （4）Activation object的实例化。Activation object(调用对象)可以看作Variable(变量)。  
function fun(a,b){};fun('p'); a和b会当成调用对象的属性，但函数调用是参数不够，b的值为undefined。如果函数内有inner function，当成属性赋值给调用对象。变量实例化最后把local variables（函数内部声名的变量） 当成调用对象的参数。调用对象的属性 包括函数的参数、内部变量。  
   （5）在函数内，local variables作为调用对象的属性出现，function (a){alert(s);   var s='a';}调用时，s的值是unidefine,直到执行到赋值语句后才有值。  
   （6）arguments属性是以索引标识的参数，它和显示声明的参数是重复的，值也相同。如果local变量的签名和参数相同，那么它们三者一个变化，其它都会相应改变值。见下例  
function a(p){alert( arguments [0]);alert(p);var p=1;alert(p);alert( arguments [0]);};a(0);  
   （7）最后，为this关键字设置值。可能 new Function()的有些疑问。关于this关键字，感觉自己还没有彻底理解。this关键字关联于执行时的作用域，而非定义时的作用域。(The this keyword is relative to the execution context, not the declaration context )  
  
    global execution context 的过程和上面有些不同，它没有arguments也不需要定义Activation object。global execution context也不需要scope chain,因为scope chain只有一个，就是global object.它的变量实例化过程和inner function其实都是根变量和函数,就是global对象的属性。global execution context用this应用global对象，在浏览器中为window.  
  
2、Scope chains and [[scope]]   
    The scope chain of the execution context for a function call is constructed by adding the execution context's Activation/Variable object to the front of the scope chain held in the function object's [[scope]] property。我理解每个函数执行环境都有scope chain，子函数(inner function)的scope chain包括它的父函数的scope chain，如此递归对global对象。  
    在ECMAScript中，函数是个对象，它们可以用function声明，或function表达式声明，或Function构造函数初始化。  
    用Function构造的函数对象一直有个scope属性,指向的scope chain 仅包括 global 对象。  
    用function表达式定义的函数对象，这类函数对象的scope chain被分配到内部的scope 属性。  
  
（1）简单的global函数，例如:-  
function exampleFunction(formalParameter){  
    ...   // function body code  
}  
  
在global execution context的变量实例化阶段，the corresponding function object 被创建。global execution context有scope chain，只包含global object.因此，函数对象被分配一个指向global object的 scope属性（ internal [[scope]] property）。  
  
（2）A similar scope chain is assigned when a function expression is executed in the global context:-  
  
var exampleFuncRef = function(){  
    ...   // function body code  
}  
  
这个例子scope chain情形与上类似。有个区别是函数对象在代码执行过程才创建。（见我以前文章）  
（3）inner 函数的情形较为复杂，看下面代码：  
  
function exampleOuterFunction(formalParameter){  
    function exampleInnerFuncitonDec(){  
        ... // inner function body  
    }  
    ...  // the rest of the outer function body.  
}  
exampleOuterFunction( 5 );  
  
    outer函数在global execution context变量实例化阶段被创建，因此它的scope chain只包括global object.  
   当global代码执行到调用exampleOuterFunction时，一个新的execution context被创建，(Activation)调用对象也被创建。这个新的execution context的scope chain由两部分组成，新的调用对象在顶层，outer函数scope chain（只包括global object）在后。新的execution context的变量实例化阶段（outer 函数体内）导致inner函数对象被创建，这个inner函数对象的[[scope]] property 被指向上述的哪个scope chain,也就是调用对象和global object.注意inner function也有调用对象。  
  
  
引用了 <http://wj.cnblogs.com/archive/2006/04/22/381851.html> 回复内的代码  
  
  
<SCRIPT LANGUAGE="JavaScript">  
<!--  
//global 代码实例化阶段，它知道global object.  
function createAClosure()   
{   
//当调用时，调用对象创建，execution context的scope chain 包括调用对象和global   
//object.  
var local = 0;   
return function(){return ++local;}; //这个inner function 的scope //chain持有  
//createAClosure的调用对象，所以也持有local的值  
}   
var c1 = createAClosure(); //调用对象和global object  
var c2 = createAClosure(); //另外一个调用对象和global object  
document.write(c1() + "<br/>");   
document.write(c1() + "<br/>");   
document.write(c1() + "<br/>");   
document.write(c2() + "<br/>");   
document.write(c2() + "<br/>");   
//-->  
</SCRIPT>  
  
以上所有过程自动进行，代码不需要任何设置（造成很多人不知道闭包原因）。  
scope chain 简单看来可以按照下面的代码来描述：  
  
函数体外Execution context 的scope chain  只有 global.  
function fun(){  
函数体内Execution context 的scope chain  fun的调用对象+global  
    function innerfun(){  
      inner函数体内Execution context 的scope chain innerfun的调用对象 + fun的调用对象 + global  
    }  
  
}  
  
但是ECMAScript提供的with表达式会修改scope chain.with表达式，我是能不用就不用了，<javascript权威指南>中也说with会造成性能的集聚下降。原文贴在下面。有时间再仔细研究。  
  
The with statement evaluates an expression and if that expression is an object it is added to the scope chain of the current execution context (in front of the Activation/Variable object). The with statement then executes another statement (that may itself be a block statement) and then restores the execution context's scope chain to what it was before.   
  
A function declaration could not be affected by a with statement as they result in the creation of function objects during variable instantiation, but a function expression can be evaluated inside a with statement:-   
  
/\* create a global variable - y - that refers to an object:- \*/  
var y = {x:5}; // object literal with an - x - property  
function exampleFuncWith(){  
    var z;  
    /\* Add the object referred to by the global variable - y - to the  
       front of he scope chain:-  
    \*/  
    with(y){  
        /\* evaluate a function expression to create a function object  
           and assign a reference to that function object to the local  
           variable - z - :-  
        \*/  
        z = function(){  
            ... // inner function expression body;  
        }  
    }  
    ...   
}  
  
/\* execute the - exampleFuncWith - function:- \*/  
exampleFuncWith();  
When the exampleFuncWith function is called the resulting execution context has a scope chain consisting of its Activation object followed by the global object. The execution of the with statement adds the object referred to by the global variable y to the front of that scope chain during the evaluation of the function expression. The function object created by the evaluation of the function expression is assigned a [[scope]] property that corresponds with the scope of the execution context in which it is created. A scope chain consisting of object y followed by the Activation object from the execution context of the outer function call, followed by the global object.   
  
When the block statement associated with the with statement terminates the scope of the execution context is restored (the y object is removed), but the function object has been created at that point and its [[scope]] property assigned a reference to a scope chain with the y object at its head.   
  
3、Identifier Resolution   
   关于这部分我决定不按照原文直译。Identifier Resolution是一个过程，而不是具体的概念，我举个例子可能就明白了。  
  
<SCRIPT LANGUAGE="JavaScript">  
<!--  
var s\_global='global';//scope chain {global} 中   
var s\_outer='global';//scope chain {global} 中   
var s\_inner='global';//scope chain {global} 中   
function outerfun(){//scope chain {global} 中   
    var s\_outer='outer';//scope chain  {outerfun调用对象，global}  
pf('outer代码开始');  
pf(s\_global);//global   
    pf(s\_outer);//outerfun调用对象  
    pf(s\_inner);//global   
  
function innerfun(){////scope chain  {outerfun调用对象，global}  
    var s\_inner='inner';//scope chain  {innerfun调用对象，outerfun调用对象，global}  
pf('inner代码开始');  
pf(s\_global);//global   
    pf(s\_outer);//outerfun调用对象  
    pf(s\_inner);//innerfun调用对象  
}  
return innerfun;  
}  
function pf(msg){document.writeln('</br>'+msg);};  
pf('global代码开始');  
pf(s\_global);//global   
pf(s\_outer);//global   
pf(s\_inner);//global   
  
var a=outerfun();  
a();  
pf('第二个函数开始------------------------');  
var b=outerfun();  
b();  
//-->  
</SCRIPT>  
  
其实Identifier Resolution就是属性查找的过程。 先从scope chain 的第一个对象开始找，如果找不到再从scope chain的第二个对象找， global对象始终是scope chain 的最后一个对象，如果global object中也找不到属性，那为undefined.  
有两个注意点：  
   如果可能，这个查找过程会对对象的prototype（原型对象）查找。先找实例属性，再找原型属性。见我的其它文章。  
   在函数内，这个函数的调用对象包括的参数，local变量，inner函数等。  
  
  
Closures  
  
1、自动的垃圾回收  
   ECMAScript有自动的垃圾回收机制。与java类似。但是规范也没有对该机制详细定义，而是让浏览器等规范实现厂家来实现，各种浏览器实现不一样，垃圾回收的算法也不同。好象ie的实现会出现内存溢出问题。对我我们来说注意到这点就够了，后面会提到如何避免ie的这个bug.  
   关于上篇提到的execution context，调用对象，参数，scope chain 等等都需要内存，垃圾回收机制会在适当时候释放内存。  
2、闭包如何形成  
   通俗的说，当一个(outer)函数的返回类型是(inner)函数类型时，这个被返回的inner函数斥又outer函数的scope chain，这时候闭包形成。  如下例：  
  
function exampleClosureForm(arg1, arg2){  
    var localVar = 8;  
    function exampleReturned(innerArg){  
        return ((arg1 + arg2)/(innerArg + localVar));  
    }  
    /\* return a reference to the inner function defined as -  
       exampleReturned -:-  
    \*/  
    return exampleReturned;  
}  
var globalVar = exampleClosureForm(2, 4);  
  
  现在exampleClosureForm(2, 4)返回的inner函数不能被垃圾回收，因为它被变量globalVar持有，并可执行globalVar(n)。  
  但是内部的原理没有表面上那么简单。现在globalVar是个函数对象，它的[[scope]] property 指向一个scope chain,而这个scope chain 包括   exampleClosureForm函数的调用对象+global对象。所以垃圾回收不能回收这部分内存。  
  一个闭包形成了。inner函数对象有自己的变量，也可以访问exampleClosureForm函数调用过程中的参数，local变量等。  
  在上面的例子中，globalVar(n)执行时,在通过调用对象可以访问到exampleClosureForm(2, 4)执行过程中的参数，local变量等。arg1 = 2，arg2 = 4 ，localVar=8，这些属性都通过调用对象"ActOuter1"可以得到。  
  
如果增加以下代码，又返回另外一个inner 函数。  
var secondGlobalVar = exampleClosureForm(12, 3);  
exampleClosureForm(12, 3)会引起新的调用对象创建，我们定义为ActOuter2。这个过程中，arg1 = 12，arg2 = 3 ，localVar=8。第二个闭包形成了.  
  
   下面考虑返回的inner函数执行过程。如globalVar(2)。新的execution context、调用对象（ActInner）被创建。现在的scope chain是  ActInner1->ActOuter1->global object.  函数返回是 ((2 + 4)/(2 + 8)).  
    如果是secondGlobalVar(5)被执行情况是什么呢？现在的scope chain是ActInner2-> ActOuter2-> global object.函数返回是 ((12 + 3)/(5 + 8)).  
  
    通过比较，这两个inner函数互不干扰的执行。如果嵌套更多的函数的话，与上面所诉类似。明白的javascript的闭包，从这个方面可能就能体会到它比java等慢n个数量级的原因。  
3、闭包能做什么（例子）  
（1）  
function callLater(paramA, paramB, paramC){  
    return (function(){  
        paramA[paramB] = paramC;  
    });  
}  
var functRef = callLater(elStyle, "display", "none");  
hideMenu=setTimeout(functRef, 500);  
想象我们做颜色渐变，或者动画的时候吧。上面提供的函数多幽雅。  
（2）  
function associateObjWithEvent(obj, methodName){  
    return (function(e){  
        e = e||window.event;  
        return obj[methodName](e, this);  
    });  
}  
function DhtmlObject(elementId){  
    var el = getElementWithId(elementId);  
    if(el){  
        el.onclick = associateObjWithEvent(this, "doOnClick");  
        el.onmouseover = associateObjWithEvent(this, "doMouseOver");  
        el.onmouseout = associateObjWithEvent(this, "doMouseOut");  
    }  
}  
DhtmlObject.prototype.doOnClick = function(event, element){  
    ... // doOnClick method body.  
}  
DhtmlObject.prototype.doMouseOver = function(event, element){  
    ... // doMouseOver method body.  
}  
DhtmlObject.prototype.doMouseOut = function(event, element){  
    ... // doMouseOut method body.  
}  
......  
  
  
又一种注册事件的方法。我觉得作者的这种实现可谓精妙。大大的开阔了我的思路。我们可以为我们的UI事件绑定到对象上，可以很好的重用代码。另外比起prototype.js的时间注册来说简单点。  
（３）  
var getImgInPositionedDivHtml = (function(){  
    var buffAr = [  
        '<div id="',  
        '',   //index 1, DIV ID attribute  
        '" style="position:absolute;top:',  
        '',   //index 3, DIV top position  
        'px;left:',  
        '',   //index 5, DIV left position  
        'px;width:',  
        '',   //index 7, DIV width  
        'px;height:',  
        '',   //index 9, DIV height  
        'px;overflow:hidden;\"><img src=\"',  
        '',   //index 11, IMG URL  
        '\" width=\"',  
        '',   //index 13, IMG width  
        '\" height=\"',  
        '',   //index 15, IMG height  
        '\" alt=\"',  
        '',   //index 17, IMG alt text  
        '\"><\/div>'  
    ];  
    return (function(url, id, width, height, top, left, altText){  
        buffAr[1] = id;  
        buffAr[3] = top;  
        buffAr[5] = left;  
        buffAr[13] = (buffAr[7] = width);  
        buffAr[15] = (buffAr[9] = height);  
        buffAr[11] = url;  
        buffAr[17] = altText;  
        return buffAr.join('');  
    }); //:End of inner function expression.  
})();  
  
这种匿名函数的调用在dojo中见过，现在再看，感觉不一样。  
  
以上是原作者的例子，我抄过来的。下次我准备深入研究一下闭包能给我们开发js类库提供什么更好的思路。感觉现在很多人对闭包了解不多，经过这段时间的思考，利用javascript中的闭包，代码偶合性会更低。