闭包

闭包是一个表达式(通常是一个函数)，可以有任意参数，连同绑定这些参数的环境(它“封闭”了表达式)一起构成。

闭包是 ECMAScript (javascript)最强大的特性之一，但是不理解就无法正确地利用它。然而，创建它们相对容易，甚至无意间就可以创建，创建它们会在造成潜在的有害后果，尤其在一些相对通用的浏览器环境中。为了避免无意间遭遇这些弊端，并利用它们提供的便利，非常有必要理解其机制。这在很大成程度上取决于标识符解析过程中作用域链所扮演的角色，以及对象属性名解析方面的协议。

闭包的简单解释是，ECMAScript 支持内部函数，函数定义和函数表达式位于其它函数体内部。这样就允许这些内部函数访问所有局部变量、(外部)函数的参数和外部函数中声明的(其它)内部函数。当这些内部函数之一可以在其所属的(外部)函数之外被访问时，就形成了一个闭包。这样，在外部函数返回之后它仍然可以执行。这时它仍然可以访问局部变量、参数和外部函数中声明的其它内部函数。当外部函数返回之后，这些局部变量、(外部)函数参数和函数定义(最初)仍然持有它们拥有的值，并且可以与内部函数交互。

不幸的是，正确理解闭包需要理解它们背后的机制，以及相当多的技术细节。虽然在以下描述的开头部分，ECMA 262 指定的一些算法已经被废弃了，但是很多不能省略或者简单解释。熟悉对象属性名解析的个人可以跳过以下小节，并且可以停止阅读此文档然后回去运用它们。

对象属性名决议

CMAScript 承认两种类型的对象,“本地对象”和“宿主对象”("Native Object" and "Host Object")以及一个本地对象的子集，叫做“内置对象”("Built-in Object")(ECMA262 第三版 4.3 节)。本地对象属于语言，宿主对象由环境提供，例如可以是 document 对象、DOM 节点等等。

本地对象是松散的，并且是命名属性的动态包(当涉及到内置对象的子集时，一些实现不是那么动态，但是这通常无关紧要)。对象定义的命名属性将会持有一个值，它可以是到另一个对象的引用(从这个意义上说函数也是对象)，或者是一个原始值：String,Number,Boolean,Null 或者 Undefined。原生类型 Undefined 有点奇怪，利用它可以把一个对象的属性赋值为 Undefined，但是这么做不会从对象上删除这个属性；它保持为一个已定义的命名属性，仅仅持有了一个为 undefined 的值。

以下是一个简单描述，属性值是如何在对象上设置和读取的，尽可能最大限度地省略内部细节。

赋值

通过给这个命名属性指定一个值，可以创建对象的命名属性，或者为已存在的命名属性赋值。例如给定：

```
var objectRef = new Object(); //创建一个普通的 javascript 对象。
```

一个名为"testNumber"的属性可以这样创建：

```
objectRef.testNumber = 5;
```

```
/* 或者*/
objectRef["testNumber"] = 5;
```

在赋值之前，对象没有"testNumber"属性，但是在赋值之后会创建一个。所有后续赋值都没有必要创建属性，仅仅重新设置它的值：

```
objectRef.testNumber = 8;
/* 或者*/
objectRef["testNumber"] = 8;
```

Javascript 对象的属性自身也可以是对象，稍后详述，并且原型(prototype)也可以拥有命名属性。但是它在赋值的时候不起作用。如果赋值的时候，实际的对象没有相应名称的属性，一个拥有此名称的属性将会被创建，并且把值赋给它。如果它已经拥有此属性则重新赋值。

读取值

从对象属性上读取值时原型开始起作用。如果一个对象拥有一个属性，与属性访问器(译者注，指的就是属性访问操作符)的属性名相同，则返回这个属性的值:

```
/*
给一个命名属性赋值
如果在赋值之前对象没有一个对应名称的属性
在赋值之后我们会获得一个
*/
objectRef.testNumber = 8;
/* 从属性上读回这个值*/
var val = objectRef.testNumber;
/* 然后，val 现在持有了个值为 8，就是我们刚才赋给对象命名属性的值。*/
```

但是所有对象都可以有原型，并且原型也是对象，所以接着它们也可以有原型，原型又可以有原型，如此就形成了所谓的原型链。当链条中的对象有一个值为 null 的原型时，原型链终止。Object 构造器的默认原型值为 null，所以：

```
var objectRef = new Object(); //创建一个通用的 JavaScript 对象。
```

创建一个对象的原型为 Object.prototype，这个原型自身的属性为 null。所以 objectRef 的原型链仅仅包含一个对象：Object.prototype。然而：

```
/*
一个构造函数，用来创建 MyObject1 型的对象。
*/
function MyObject1(formalParameter){
/*
给创建的对象赋一个名为 testNumber 的属性
并且将传给构造函数的第一个参数值赋给它
*/
    this.testNumber = formalParameter;
}
/*
一个构造函数，用来创建 MyObject2 型的对象
*/
```

```
function MyObject2(formalParameter){
   /*
    给创建的对象一个名为 testString 的属性
    并且将传给构造函数的第一个参数值赋给它
    */
    this.testString = formalParameter;
}
/*
   下一步操作，把关联到 MyObject2 实例上的默认 prototype
   替换成 MyObject1 的实例，给 MyObject1 的构造函数传递一个值 8，
   这样它的 testNumber 属性将会被设置为这个值：
*/
MyObject2.prototype = new MyObject1( 8 );
/*
    最后，创建 MyObject2 的实例，然后将这个对象的引用赋值给变量 objRef
    给构造函数传递一个字符串作为第一个参数
*/
var objectRef = new MyObject2( "String_Value" );
```

被变量 objectRef 引用的 MyObject2 实例拥有一个原型链。链中的第一个对象是 MyObject1 的实例，他被创建并赋值给 MyObject2 构造函数的 prorotype 属性。MyObject1 的实例有一个原型，此对象由(引擎)实现分配给函数 MyObject1 的 prototype 属性。这个对象(译者注：指分配给 MyObject1 的 prototype 属性的对象)有一个原型，就是默认 Object 的原型，对应 Object.prototype 所引用的对象。Object.prototype 是一个值为 null 的原型，所以原型链到这一点结束。

当一个属性访问器试图从变量 objectRef 所引用的对象上读取一个命名属性时，整个原型链都会加入处理过程。在简单的情况下：

```
var val = objectRef.testString;
```

objectRef 所引用的 MyObject2 的实例有一个名为”testString”的属性，所以它就是这个属性的值，设置为”String_Value”，它被赋值给变量 val。然而：

```
var val = objectRef.testNumber;
```

不能从 MyObject2 实例自身读取命名属性，因为它没有这个属性，但是变量 val 的值被设置为 8 而不是 undefined，因为在对象自身上查找对应的属性失败之后，解释器会检查对象的原型。它的原型是 MyObject1 的实例，它被创建时有一个属性名为”testNumber”，并且把值 8 赋给了这个属性，所以属性访问器算出的值为 8。MyObject1 和 MyObject2 都没有定义 toString 方法，但是如果一个属性访问器试图从 objectRef 上读取 toString 属性的值：

```
var val = objectRef.toString;
```

变量 val 被赋值为一个函数的引用。这个函数是 Object.ptorotype 的 toString 属性，它被返回是因为检查了 objectRef 的原型，当发现 objectRef 没有”toString”属性，而是引用一个对象，所以当发现原型中也缺少此属性时，则继续查找原型的原型。它的原型是 Object.prorotype，确实有一个 toString 方法，所以 val 就被赋值为一个引用，指向返回的函数对象。

最后：

```
var val = objectRef.madeUpProperty;
```

返回 undefined，因为处理过程检查了整个原型链，发现没有哪个对象有一个名为

"madeUpPeoperty"的属性，它最终获得了 Object.prototype 的值，这是一个 null 值，然后处理过程返回一个 undefined。

对命名属性的读取操作会返回第一个找到的值，这个值可以在对象上或者在它的原型链上。给对象的命名属性赋值时，如果对象上没有对应的属性存在，将会在对象自身创建一个属性。

这意味着，如果一个值被分配为 objectRef.testNumber=3，一个”testNumber”属性将会在 MyObject2 的实例自身上被创建，随后所有读取这个属性的操作将会获得在对象上设置的值。不再需要检查原型链来解析属性访问器，但是 MyObject1 的实例，”testNumber”属性被赋值为 8 不会变。objectRef 对象上分配的对应属性遮盖了其原型链上的对应属性。

注意：ECMAScript 为内部 Object 类型定义了一个内部[[prototype]]属性。这个属性不能直接通过脚本访问，但是它是一个引用对象原型链的对象链，同时内部[[prototype]]属性在属性访问解析中被使用。存在一个公共的 prototype 属性与内部[[prototype]]属性相关联，允许对它进行赋值、定义和操作。这两者之间关系的细节在 ECMA 262 (第三版)中有描述，这已经超越了本文的讨论范围。

标识符解析，执行环境和作用域链

执行环境

执行环境是 ECMSScript 规范使用的一个抽象概念(ECMA 262 第三版)，用来定义 ECMAScript 实现所必须的行为。规范中没有提到关于执行环境应该如何实现的任何事情，但是执行环境有关联的属性，它引用了规范所定义的结构，所以它们可以被设想(甚至实现)为拥有属性的对象，虽然不是公共属性。

所有 JavaScript 代码都是在一个执行环境中被执行的。全局代码(内嵌代码，一般作为 js 文件或者 HTML 页面加载)在全局执行环境中执行，每次函数调用(可以是构造器)都有一个分配的执行环境。用 eval 函数执行的代码也有一个独特的执行环境，但是因为 eval 从来不会被 JavaScript 程序员经常使用，所以这里不讨论它。执行环境的特定的细节可以在 ECMA 262 的 10.2(第三版)节找到。

当一个 JavaScript 函数被调用的时候，它进入一个执行环境，如果其它函数被调用(或者在相同的函数上递归)，一个新的执行环境被创建，在函数调用过程中进入此环境。当这个被调用的函数返回之后会返回先前的执行环境。这样，运行中的 javascript 代码就形成了一个**执行环境栈**。

当一个执行环境被创建时，很多事情按照规定的顺序发生。首先，在一个函数的执行环境中，一个“活动”对象被创建。活动对象是另一个规范机制。它可以被看做一个对象，因为它最终拥有可访问的命名属性，但是它不是一个普通对象，因为它没有原型(prototype)(至少不是一个已定义的 prototype)，并且它不能直接被 javascript 代码引用。

为调用函数创建执行环境的下一步是创建一个 arguments 对象，这是一个类似数组的对象，拥有以整数值为下标的成员，它和调用函数时传递的参数按次序对应。arguments 对象还有 length 和 callee 两个属性(和这里的讨论无关，参见细节描述)。**活动对象**会被创建一个名为”arguments”的属性，它会被赋值为一个引用，指向 arguments 对象。

然后为执行环境分配一个作用域。作用域由一组对象列表(或链)组成。每个函数对象都有一个内部[[scope]]属性(稍后我们将涉及更多细节)，它也由一组对象列表(或链)组成。

调用函数时分配给执行环境的作用域(scope)由一个列表组成，这个列表就是被对应函数对象的[[scope]]属性所引用的列表，并且把活动对象添加到链(或者列表顶端)的前端而组成。

然后，使用一个 ECMA262 所指的”可变”(Variable)对象执行“变量初始化”的过程。然而，活动对象被用作可变对象(注意，很重要：**它们是同一个对象**)。可变对象的命名参数是为函数的每一个形参创建的，并且，如果调用函数的参数与这些参数对应，这些参数的值会被赋到属性上(否则，赋值为 undefined)。内部函数定义也被用来创建函数对象，它们也被设置到可变对象的属性上，属性名和定义函数时的函数名对应。变量初始化的最后一步是创建可变对象的命名属性，对应函数中声明的所有局部变量。

在可变对象上创建的，与申明的局部变量对应的属性，在变量初始化时被初始为 undefined，局部变量的初始化实际不会发生，直到执行函数体中代码对相应的表达式时才进行计算。

事实上，拥有 arguments 属性的“活动对象”，和拥有对应到函数局部变量的命名属性的“可变对象”，是同一个对象，这就允许把 arguments 标识符当作函数的局部变量看待。

最后，为使用 this 关键字分配一个值。如果分配的值指向一个对象，那么使用 this 关键为前缀的属性访问器将会引用这个对象的属性。如果分配(内部)的值为 null，那么 this 关键字将会指向全局对象。

全局执行环境做了一些细微不同的处理，因为它没有参数，所以它没有必要定义活动对象去引用它们。全局执行环境确实需要一个作用域，并且它的作用域链仅仅由一个对象构成，就是全局对象。全局执行环境也需要经历变量初始化，它的内部函数是所声明的普通顶级函数，它们组成了 javascript 代码的绝大部分。全局对象被用作可变对象，这就是为什么声明的全局函数成为了全局对象的属性。全局范围内声明的变量也一样。

全局执行环境也使用一个到全局对象的引用作为 this 对象。

作用域链和[[scope]]

调用函数时执行环境中的作用域链，是通过把执行环境中的活动对象/可变对象添加到作用域链顶部而构成，作用域链由函数对象的[[scope]]属性持有，所以，理解[[scope]]属性是如何定义的非常重要。

在 ECMAScript 中，函数是对象，它们在变量初始化时被创建：函数申明、执行函数表达式或者通过调用 Function 构造器。

使用 Function 构造器创建的函数对象，总是把[[scope]]属性指向一个作用域链，其中仅仅包含全局对象。

通过函数声明或者函数表达式创建的函数对象，拥有一个执行环境中的作用域链，被赋值给它们内部的[[scope]]属性，它们是在这个执行环境中被创建的。

全局函数声明最简单的情况如：

```
function exampleFunction(formalParameter){
    ...   // function body code
}
```

在为全局执行环境进行变量初始化时，对应的函数对象被创建。全局执行环境有一个作用域链，它仅仅由全局对象组成。这样，这个被创建的函数对象被全局对象以”exampleFunction”为名称引用，并且被分配了一个内部[[scope]]属性，它引用一个作用域链，其中只包含全局对象。

当一个函数表达式在全局环境中被执行时，分配一个类似的作用域链：

```
    var exampleFuncRef = function(){
        ...   // function body code
    }
```

在这种情况下，在为全局执行环境进行变量初始化的过程中，全局对象的一个命名属性被创建，并且它的一个引用被赋值给全局对象的一个命名属性，但是函数对象没有被创建，直到对赋值表达式求值为止。但是在全局环境中创建函数对象的操作还是发生了，所以创建的函数对象的[[scope]]属性仍然指向了只包含一个全局对象的作用域链。

内部函数声明和以函数对象为结果的表达式在函数内部的执行环境中被创建，所以它们拥有更精细的作用域链。考虑以下代码，定义了一个有内部函数声明的函数，然后执行外部函数：

```
function exampleOuterFunction(formalParameter){
    function exampleInnerFuncitonDec(){
        ... // inner function body
    }
    ...  // the rest of the outer function body.
}
exampleOuterFunction( 5 );
```

对应着外部函数声明的函数对象在全局执行环境的变量初始化过程中被创建，所以它的[[scope]]属性包含只有一个元素的作用域链，其中只有一个全局对象。

当外部代码调用 exampleOuterFunction 时，会为这个函数调用创建一个新的执行环境，同时还有和它一起的活动对象/可变对象。新执行环境中的作用域构成变成了：新的活动对象加上外部函数对象[[scope]]属性(仅包含全剧对象)所引用的作用域链。为新执行环境进行变量初始化导致创建了一个函数对象，它对应内部函数定义，并且这个函数对象的[[scope]]属性被分配为它被创建时执行环境中的作用域值。作用域链包含了活动对象，紧接着是全局对象。

到目前为止，这些都是自动完成的，并且由构建和执行源码(的机制)控制。执行环境中的作用域链定义了所创建的函数对象的[[scope]]属性，同时函数对象的[[scope]]属性为它们的执行环境定义了作用域(和对应的活动对象一起)。但是 ECMAScript 提供了 with 语句用来作为修改作用域链的手段。

with 语句对一个表达式求值，如果这个表达式是一个对象，它将会被添加到当前执行环境的作用域链中(在活动对象/可变对象前面)。with 语句然后执行其它语句(这本身可能是一个语句块)然后恢复执行环境的作用域链为之前的值。

一个函数声明不会被 with 语句影响，因为它们会在变量初始化的时候导致创建函数对象，但是一个函数表达式可以在一个 with 段中被执行：

```
/* 创建一个全局变量 y，引用一个对象*/
var y = {x:5}; //对象字面值，有一个 x 属性
function exampleFuncWith(){
    var z;
    /*
      将变量 y 引用的对象添加到作用域链的最前端
    */
    with(y){
        /*
            对函数表达式求值
```

```
            创建一个函数对象，将这个函数对象的引用赋值给局部变量 z
        */
        z = function(){
            ... // inner function expression body;
        }
    }
    ...
}

/* 执行 exampleFuncWith 函数*/
exampleFuncWith();
```

当 exampleFuncWith 函数被调用时，结果执行环境拥有一个作用域链，由它的活动对象加上全局对象组成。with 语句执行，当函数表达式被求值时，全局变量 y 引用的对象被添加到这个作用域链前面。由执行函数表达式创建的函数对象被分配了一个[[scope]]属性，它对应于被创建时执行环境的作用域。这个作用域链由对象 y 加上调用外部函数时执行环境中的活动对象，再加上全局对象组成。

当 with 语句中的代码块结束时，执行环境的作用域被恢复(y 对象被删除)，但是函数对象已经在那一点被创建，并且它的[[scope]]属性被赋值成了一个作用域链的引用，它以 y 对象开头。

标识符解析

标识符根据作用域链解析。ECMA 262 把 this 当作了一个关键字而不是标识符，这是不无道理的，因为它总是根据它被使用时执行环境中的 this 值解析，而没有引用作用域链。

标识符解析从作用域链的第一个对象开始。它被检查，用来看看是否有与属性名称对应标识符。因为作用域链是一个对象链，这个检查包含该对象的原型链(如果有)。如果在作用域链中的第一个对象上没有找到对应的值，查找过程在下一个对象上进行。如此等等，直到作用域链中(或者其原型)的一个对象拥有一个属性的名字与标识符对应，或者作用域链耗尽为止。

在标识符上进行的操作与以上描述的访问对象原型的方式一样。在作用域链中确定的拥有对应属性的对象，会替代属性访问器中的对象，并且使用标识符作为这个对象的属性名。全局对象总是位于作用域链的最后。

因为绑定到调用函数时的执行环境将会拥有活动/可变对象位于链的顶部，函数体中使用的标识符将会首先做有效性检测，看看它们是否与形参、内部函数声明的名称或者局部变量相一致。这些将会被解析成活动对象/可变对象的命名属性。

闭包

自动垃圾收集

ECMAScript 使用自动垃圾收集。规范没有规定细节，把它留给具体实现去考虑，一些已知的实现给它们的垃圾收集操作一个很低的优先级。但是总的思路是，如果一个对象变成

孤立的(有没留下到它的引用供执行的代码访问)，它就变成了一个需要进行垃圾收集的变量，在未来的某个时间点它会被销毁，并且它所消耗的所有资源都会被释放并归还给系统，以便重复利用。

这通常是退出执行环境时反生的情况。作用域链、活动/可变对象以及所有在执行环境中创建的对象，包括函数对象，将不再可以访问并且将会变成可被垃圾收集的变量。

构造闭包

通过返回一个函数对象，它在一次函数调用的执行环境中被创建，从这次函数调用中，把这个内部函数的引用赋值给另一个对象的属性，或者直接把这个函数对象的引用分配给，例如一个全局变量、一个全局可见对象的属性，或者赋给在调用外部函数时作为参数传递的对象引用。例如：

```
function exampleClosureForm(arg1, arg2){
    var localVar = 8;
    function exampleReturned(innerArg){
        return ((arg1 + arg2)/(innerArg + localVar));
    }
    /*
        返回一个定义为 exampleReturned 的内部函数引用
    */
    return exampleReturned;
}
var globalVar = exampleClosureForm(2, 4);
```

现在，调用 exampleClosureForm 函数时执行环境中创建的函数对象不能被垃圾收集，因为它被全局变量所引用并且仍然可以访问，它甚至可以使用 globalVar(n)这种方式来执行。

因为函数对象现在被 globalVar 引用，事情变得有一点小复杂，这个函数创建了一个[[scope]]属性，引用了一个属于执行环境的作用域链，包含了活动/可变对象(还有全局对象)，这个函数是在执行环境中被创建的。现在，活动/可变对象也不能被垃圾收集，因为对象被 globalVar 引用，函数执行时将会需要从它的[[scope]]属性中把整个作用域链添加到调用它时所创建执行环境的作用域中去。

一个闭包形成了。内部函数对象拥有任意变量和活动/可变对象，这些对象由环境绑定到函数的作用域链上。

活动/可变对象现在被限制，并被作用域链引用，作用域链被赋值给函数对象内部的[[scope]]属性，函数对象被变量 globalVar 引用。活动/可变对象和它的状态及其属性一起被保存。调用内部函数时，执行环境中的作用域解析将会解析标识符，活动/可变对象相应的命名属性就是这个对象的属性。这些属性值仍然可以读写，即使创建它们的执行环境已经退出。

在以上例子中，在外部函数返回之后(退出它的执行环境)，活动/可变对象有一个表示形参值、内部函数定义和局部变量的状态。arg1 属性值为 2，arg2 属性值为 4，loclaVar 值为 8，exampleReturned 属性是从外部函数返回的一个指向内部函数对象的引用(为了方便，在后面的讨论中我们将会把这个活动/可变对象称作”ActOuter1”)。

如果 exampleClousureForm 函数再次被调用作：

```
var secondGlobalVar = exampleClosureForm(12, 3);
```

一个新的执行环境将会被创建，还有一个新的活动对象。然后一个新的函数对象将会被返回，拥有它自己唯一的[[scope]]属性，引用一个作用域链，包含来自第二个执行环境的活动对象，arg1 为 12，arg2 为 3(为了方便，在后面的讨论中我们将会把这个活动/可变对象称作”ActOuter2)

通过再次执行 exampleClosureForm，形成了第二个不同的闭包。

通过执行 exampleClosureForm 函数创建了两个函数对象，它们的引用被分别赋值给了全局变量 globalVar 和 secondGlobalVar，函数对象返回表达式((arg1 + arg2)/(innerArg + localVar))。此表达式在四个标识符上进行了大量操作。这些标识符如何被解析对闭包的用法和价值非常重要。

考虑使用 globalVar(2)的形式执行 globalVar 所引用的函数对象。一个新的执行环境被创建，还有一个活动对象(我们将会称它做”ActInner1”)，它将被添加到作用域链顶部，被所执行的函数对象中的[[scope]]属性引用。ActInner1 被赋给了一个名为 innerArg 的属性，然后它的形参和参数值 2 被分配给它。新的执行环境的作用域链目前是：ActInner1->ActOuter1->global object。

根据作用域链，标识符解析已经完成，所以为了返回表达式((arg1 + arg2)/(innerArg + localVar))的值，通过查找与标识符名称一致的属性，这些标识符的值将会被计算，在作用域链中的每个对象上依次进行。

作用域链中的第一个对象就是 ActInner1，它有一个名为 innerArg 的属性其，值为 2。其它 3 个标识符都与 ActOuter1 的命名属性相对应；arg1 为 2，arg2 为 4，localVar 为 8。函数调用将会返回((2+4)/(2+8))。

与被 secondGlobalVar 引用的其它相同函数对象相比较，secondGlobalVar(5)。把这个新执行环境中的活动对象叫做”ActInner2”，作用域链变成了:ActInner2->ActOuter2->global object。ActInner2 返回 innerArg 为 5，ActOuer2 返回 arg1，arg2，localVar 分别 12，3，8。最终返回的值为((12 + 3)/(5 + 8)).

再次执行 secondGlobalVar,一个新的活动对象出现在作用域链的头部,但是 ActOuter2 仍然是作用域链中的下一个对象，并且它的命名属性在解析标识符 arg1，arg2 和 localVar 时会被再次使用。

这就是 ECMAScript 内部函数如何获取、保持、访问它被创建时的那个执行环境中的形参、声明的内部函数和变量的方式。并且就是闭包如何允许这些函数对象以何种形式保持到这些值的引用、读写它们，只要它继续存在。所创建内部函数执行环境中的活动/可变对象，存在于作用域链中，被函数对象的[[scope]]属性引用，直到所有对内部函数的引用都被释放并且变为可以进行垃圾收集为止(与它的作用域链中所有无效对象一起)。

内部函数自己也可以有内部函数,从函数的执行环境中返回的内部函数形成的闭包自己又可以返回内部函数,并形成它自己的闭包。每层嵌套,执行环境都会获得额外的活动对象,此对象来自函数对象被创建的执行环境。ECMAScript 规范要求作用域链是有限的，但是没有限定其长度。具体实现可以做一些实际的额外限制，但是具体的幅度还没有相关报告。迄今为止，内部函数的潜力已经超越了任何实现它们的人的期望。

我们可以用闭包做什么？

奇怪的是，这个问题的答案是任何东西，一切。有人告诉我，闭包使得 ECMAScript 可以模拟任何东西，所以唯一的限制就是想象力和模拟实现的能力了。这有一点深奥，以一些更实际的东西开始可能更好。

例 1：使用函数引 setTimeout

闭包的一个常见用途是在执行函数之前为函数提供参数。例如，把函数用作 setTimeout 函数的第一个参数在 web 浏览器环境很常用。

setTimout 计划调用一个函数(或者一个 javasctipt 源码字符串，但是不在当前上下文中)，作为其第一个参数提供，后面是一个以毫秒为单位的时间间隔(作为其第二个参数)。如果一段代码想使用 setTimeout，它调用 setTimeout 函数并传递一个函数对象的引用作为第一个参数，和一个毫秒时间间隔作为第二个参数，但是函数对象的引用无法为这个计划执行的函数提供参数。

但是，代码可以调用另一个函数，它返回一个内部函数的引用，把这个内部函数的引用传递给 setTimeout 函数。执行内部函数需要使用的参数在调用(外部)函数时传递，这个(外部)函数生成了内部函数。setTimeout 执行这个内部函数而不传递参数，但是内部函数仍然可以访问调用外部函数时所提供的参数，外部函数返回了它(指内部函数)：

```
function callLater(paramA, paramB, paramC){
    /*
        返回一个匿名内部函数，它是通过一个函数表达式创建的
    */
    return (function(){
        /*
            这个内部函数将会被 setTimeout 执行，并且当它被执行时
            它可以读取并操纵传递给外部函数的参数
        */
        paramA[paramB] = paramC;
    });
}

...

/*
    调用这个函数，它将返回一个内部函数对象的引用，这个函数对象
    是在它的执行环境中被创建的 。传递参数传递给外部函数，这些参数
    是最终执行内部函数时需要的。
*/
var functRef = callLater(elStyle, "display", "none");



/*
    调用 setTimeout 函数，将赋值给 functRef 的内部函数引用作为第一个参数传递
*/
hideMenu=setTimeout(functRef, 500);
```

例 2：将函数与对象实例方法关联

有很多其它情况，当一个函数的引用被赋值，然后在未来某时被执行，为执行这个函数提供参数非常有用，并且在执行时提供不是非常合适，但是直到赋值时都无法确知。

一个例子可能是，javascript 对象被设计用来包装与特定 DOM 元素的交互。它有 doOnClick，doMouseOver 和 doMouseOut 几个方法，需要在 DOM 元素触发对应的事件时执行，但是可能有任意数量的 javascript 对象实例被创建并关联到不同 DOM 元素，并且单个对象实例并不知道它们将会如何受雇于实例化它们的代码。对象实例不知道如何全局地引用它们自己，因为它们不知道它们的实例会被赋值给哪个全局变量(如果存在)。

所以问题是，执行一个事件处理函数，它关联了一个特定的 javascript 对象，并知道调用这个对象的哪个方法。

以下例子使用了一个一般性的基于闭包的小函数，它把事件处理函数绑定到对象实例。安排事件处理程序调用对象实例的特定方法，传递事件对象和一个所关联元素的引用给对象的方法，并返回该方法的返回值。

```
/*
    一个通用函数，将一个对象关联到事件处理器。返回的
    内部函数将被用作事件处理器。对象实例以名为 obj 的参数传递
    对象上需要被调用方法的名字以名为 methodName(字符串)的参数传递
*/
function associateObjWithEvent(obj, methodName){
    /*
        返回的内部函数将作为替代，扮演 DOM 元素事件处理器的角色
    */
    return (function(e){
        /*
            一般化的事件对象，在 DOM 标准的浏览器中，它将会被以名为 e 的
            参数传递，对于 IE 事件对象不会被作为参数传递给作为事件处理器
            的内部函数
        */
        e = e||window.event;//译者注：这是兼容 IE 事件对象的一个重要技巧
        /*
            事件处理器调用 obj 对象上的一个方法，其名称被字符串 methodName 持有
            传递一般化的事件以及元素的引用，事件处理器将会以 this 引用它们
            (这能起作用是因为这个内部函数被作为这个元素的一个方法执行，因为它
            已经被作为一个事件处理函数赋值)
        */
        return obj[methodName](e, this);
    });
}

/*
    这个构造函数创建对象，把它们自己关联到 DOM 元素，这些元素
    的 ID 被作为字符串传递给构造器。对象实例需要安排当相应元素触发 onclick，
    onmouseover，onmouseout 事件时，它们对象实例上的相应方法会被调用
*/
function DhtmlObject(elementId){
```

```
    /*
        一个函数被调用，用来获取 DOM 元素的引用(如果没有找到为 null)，
        传递所需元素的 ID 作为参数。返回值将会被赋给局部变量 el
    */
    var el = getElementWithId(elementId);
    /*
        因为 if 语句，el 的值将会被隐含转换成布尔型，所以如果 el 引用一个对象
        结果将会为 true，如果 el 为 null 则结果为 false。所以，接下来的代码块只有
        在 el 变量引用一个 DOM 元素时才会被执行
    */
    if(el){
        /*
            为了给元素的事件处理器赋一个值，该对象调用 associateObjWithEvent 函
            数，指定自己(使用 this 关键字)作为需要在其中调用方法的对象，并且提供
            需要调用方法的名称。associateObjWithEvent 函数将会返回一个内部函数引
            用，它将会被赋值给 DOM 元素的事件处理器。这个内部函数将会调用
            javascript 对象上所需的方法响应事件：
        */
        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.
}
```

这样 DhtmlObject 的所有实例都可以把自己关联到自己感兴趣的 DOM 元素上，而不需要知道它们自己是如何被其它代码调用的任何细节，以及对全局命名空间的影响、与其它 DhtmlObject 实例冒冲突的风险。

例 3：封装相关的功能

闭包可以用来创建额外的作用域，可以用来组合互相关联和依赖的代码，通过这种方式最大限度地减少意外交互的风险。假设一个函数用来创建一个字符串并避免重复的连接操作(和创建许多中间字符串)期望是使用一个数组来顺序存储字符串片段，然后使用 Array.prototype.join 方法输出结果(使用一个空字符串作为它的参数)。数组将会用来作为一个输出缓冲区，但是在函数内部定义它将会导致每次执行函数时都会重复创建，如果数

组中的唯一变量内容将会在每次调用函数时被重新分配，这可能是没有必要的。

一种方法是把数组做成一个全局变量，这样它就可以被重复利用而无需重新创建。但是这样做的后果是，除了全局变量所引用的函数会使用这个数组之外，还存在第二个全局变量引用数组自身。其影响就是降低代码的受控程度，因为，如果它被用在其它地方，它的作者可能不记得既导入函数定义又导入数组定义。这也导致代码不能与其它代码方便地进行交互，因为除了保证函数名在全局命名空间中唯一之外，还必须保证它所依赖的数组在全局命名空间中名称唯一。

闭包允许把这个缓冲区数组绑定(并整齐地包装)到依赖它的函数上，同时把缓冲区数组的属性名保持在全局命名空间之外，从而避免了命名冲突和意外交互的风险。

这里的技巧是创建一个额外的执行环境，通过执行一个内联的函数表达式，让这个函数表达式返回一个内部函数，它将被外部代码使用。这个操作只执行一次，所以数组只会被创建一次，但是却可以被依赖它的函数重复使用。

以下代码创建了一个会返回 HTML 字符串的函数，其中大部分是常量，但是这些常量字符序列需要插入调用函数时所提供的变量信息。

从对一个函数表达式的内联执行，会返回一个内部函数的引用，并且赋值给一个全局变量，这样它就可以被当作全局函数调用。缓冲区数组定义为外部函数表达式的局部变量。它没有被暴露在全局命名空间中，并且没有必要被重复创建，无论使用它的函数何时被调用。

```
/*
    全局变量 getImgInPositionedDivHtml 被声明并赋值为内部函数表达式的引用，
    这个内部函数是对外部函数表达式的一次调用产生的。
    这个内部函数返回一个 HTML 字符串，描述了一个绝对定位的 DIV，包装在 IMG
    元素的周围，使得所有变量的属性值都被以调用函数时的参数提供：
*/
var getImgInPositionedDivHtml = (function(){
    /*
        buffAr 数组被赋值给外部函数表达式的一个局部变量。
        它仅仅被创建一次，并且这个数组实例对内部函数有效，
        所以每次执行这个内部函数时它都有效。
        空字符串当前被用来作占位符，它们将会被内部函数插入到数组中
        (译者注：指它们将会被实际值替换)：
    */
    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>'
    ];
    /*
        返回内部函数的引用，它是一个函数表达式执行的结果。
        每次调用 getImgInPositionedDivHtml( ... )时，这个内部函数
        对象将会被调用：
    */
    return (function(url, id, width, height, top, left, altText){
        /*
            将各个参数赋值到缓冲区数组中对应的位置：
        */
        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;
        /*
            把数组中的每个元素以空字符串连接到一起，返回
            这个字符串 (这和仅仅把元素连接到一起是一样的):
        */
        return buffAr.join('');
    }); //内部函数表达式结束
})();
/*外部函数表达式的内联调用*/
```

如果一个函数依赖一个(或者多个)其它函数，并且这些其它的函数不想直接受雇于任何其它代码，那么可以使用同样的技术把这些函数组织到需要被公开暴露的函数中。把一个复杂的多个函数处理的过程制作成便于携带的封装代码单元。

其它例子

闭包最著名的应用之一可能是 Douglas Crockford 的《在 ECMAScript 对象中模拟私有实例变量的技术》。它可以被扩展到各种基于作用域嵌套结构的访问控制/可见性控制中，包括《为 ECMAScript 对象模拟私有静态成员》。

闭包可能的应用无穷无尽，理解它们是如何运作的可能是认识它们用途的最好指南。

意外的闭包

把任何函数渲染到在创建它们的函数体之外可以访问会形成一个闭包。这导致闭包非常容易被创建，并且后果之一就是不熟悉闭包作为语言特性的 javascript 作者可能会注意到内部函数的各种作用并且运用内部函数，但是没有考虑结果、没有注意到闭包被创建或者这样做的影响是什么。

意外地创建闭包可能会产生有害的副作用，就像后文对 IE 内存泄漏问题的描述中所述，但它们也可以影响代码效率。这不是闭包自身的问题，小心地利用它们确实可以有助于创建高效的代码。这就是使用内部函数对效率的影响。

常见的情况就是内部函数被用做 DOM 元素的事件处理函数。例如，以下代码可能被用来为链接元素添加一个 onclick 处理函数：

```
/*
    定义全局变量，它持有添加到链接 href 上的值，它作为以下函数的查询字符串：
*/
var quantaty = 5;
/*
    当一个链接被传递给这个函数时(调用函数时作为 linkRef 参数)，一个 onclick 事件处
    理器就被添加到了链接上，这将把全局变量 quantaty 的值添加到链接的 href 属性上作
    为查询字符串，然后返回 true，这样链接将会导航到 href 指定的资源，此时所分配的
    查询字符串将会被包含在内：
*/
function addGlobalQueryOnClick(linkRef){
    /*
        如果 linkRef 参数的类型可以被转换成 true(如果它引用了一个对象)：
    */
    if(linkRef){
        /*
            执行一个函数表达式，给 link 元素的 onclick 处理器分配一个
            引用，引用执行函数表达式时创建的函数对象：
        */
        linkRef.onclick = function(){
            /*
                这个内部函数表达式把查询字符串添加到元素的 href 属性上，
                此函数被绑定为一个事件处理器：
            */
            this.href += ('?quantaty='+escape(quantaty));
            return true;
        };
    }
}
```

无论 addGlobalQueryOnClick 函数何时被调用，一个新的内部函数都会被创建(并且通过它的赋值产生一个闭包)。从效率角度考虑，如果 addGlobalQueryOnClick 函数仅仅调用一次两次，这影响不大，但是如果这个函数被大规模应用，将会创建大量独立的函数对象(每

次执行内部函数表达式都会创建一个)。

以上代码没有利用的一个事实是，内部函数在被创建的函数之外可以访问(或者说输出闭包)。结果是，通过单独定义一个函数作为事件事件处理器，然后分配一个引用给这个函数的事件处理器的属性，可以达到同样的效果。只有一个函数对象会被创建，并且使用事件处理函数的所有元素都会共享这唯一一个函数的引用：

```
/*
定义全局变量，它持有添加到链接 href 上的值，它作为以下函数的查询字符串：
*/
var quantaty = 5;
/*
    当一个链接被传递给这个函数时(调用函数时作为 linkRef 参数),一个 onclick 事件处
    理器就被添加到了链接上，这将把全局变量 quantaty 的值添加到链接的 href 属性上
    作为查询字符串，然后返回 true，这样链接将会导航到 href 指定的资源，此时分 配
    的查询字符串将会被包含在内：
*/
function addGlobalQueryOnClick(linkRef){
    /*如果 linkRef 参数的类型可以被转换成 true(如果它引用了一个对象)：*/
    if(linkRef){
        /*
            执行一个函数表达式，给 link 元素的 onclick 处理器分配一个
            引用，引用执行函数表达式时创建的函数对象：
        */
        linkRef.onclick = forAddQueryOnClick;
    }
}
/*
    一个全局函数声明，这个函数被设计用来作为链接元素的事件处理器，添加
    一个全局变量的值给元素的 href 属性作为一个事件处理器：
*/
function forAddQueryOnClick(){
    this.href += ('?quantaty='+escape(quantaty));
    return true;
}
```

因为第一个版本中的内部函数没有利用它自己产生的闭包，不使用内部函数会更有效，并且这样不会重复创建本质上相同函数对象。

一个类似的考虑适用于对象的构造函数。这种情况并不少见，参见以下的代码框架：

```
function ExampleConst(param){
    /*
    通过执行函数表达式创建对象的方法，并将结果函数对象的引用赋值给
    所创建对象的属性:
    */
    this.method1 = function(){
        ... // method body.
    };
```

```
    this.method2 = function(){
        ... // method body.
    };
    this.method3 = function(){
        ... // method body.
    };
    /* 将构造函数的参数赋值给对象的属性：*/
    this.publicProp = param;
}
```

每次这个构造函数被用来创建一个对象，使用 new ExampleConst(n)时，一组新的函数对象就被创建用来作为它的方法。所以，对象创建得越多，随着它们一起被创建的函数对象就越多。

Douglas Crockford 的在 javascript 对象上模拟私有成员的技术利用了返回闭包的形式，把内部函数的引用赋给公共属性，这个属性是所创建的对象在构造函数内部创建的。但是如果一个对象的方法没有利用闭包，初始化每个对象时会创建多个函数对象，这使得初始化过程变慢，并且将会消耗更多的资源用来容纳这些创建的额外函数对象。

在这种情况下，创建函数对象一次，然后把它们的引用赋值给构造函数的 prototype 会更有效，这样它们可以被使用这个构造函数创建的所有对象共享：

```
function ExampleConst(param){
    /*将构造函数的参数赋值给对象的一个属性：*/
    this.publicProp = param;
}
/*
    通过执行函数表达式创建对象的方法，并将结果函数对象的引用赋值
    给所构造函数的 prototype 属性:
*/
ExampleConst.prototype.method1 = function(){
    ... // method body.
};
ExampleConst.prototype.method2 = function(){
    ... // method body.
};
ExampleConst.prototype.method3 = function(){
    ... // method body.
};
```

IE 中的内存泄漏问题

IE 浏览器(验证了版本 4 到 6(目前写作时是版本 6))在其垃圾收集系统中存在一个错误，它阻止了对 ECMAScript 和一些宿主对象的垃圾收集，如果这些宿主对象是形成“循环引用”的一部分。出问题的宿主对象是 DOM 节点(包括 document 对象以及它的后代)和 ActiveX 对象。如果形成的循环引用中存在它们中的一个或者多个，则这些对象都不会被释放直到浏览器被关闭，并且它们消耗的内存系统都无法使用直到浏览器关闭。

循环引用是两个或者多个对象互相应用，用一种互相连接并能回到起点的方式引用。例如对象 1 有一个属性引用对象 2，对象 2 有一个属性引用对象 3，然后对象 3 有一个属性回到对象 1。对纯粹的 ECMAScript 对象来说，只要没有其它对象引用对象 1，2 或者 3 中的其中之一，它们互相引用对方的事实就能被确定，并且它们对垃圾收集有效。但是在 IE 中，如果这些对象中的任意一个恰好是 DOM 节点或者 ActiveX 对象，则垃圾收集器无法发现它们之间的这种循环关系是和系统的其它部分相隔离的并释放它们。事实上它们都驻留在内存中直到浏览器被关闭。

闭包极其容易形成循环引用。如果一个函数对象形成了的一个闭包，例如，被分配为一个 DOM 节点的事件处理器，到这个节点的引用就被分配给了执行环境中的活动对象/可变对象之一，然后循环引用就形成了。DOM_Node.onevent -> function_object.[[scope]] -> scope_chain -> Activation_object.nodeRef -> DOM_Node。这非常容易做到，浏览一个在每个页面的一段代码中形成了这种引用关系的站点可能会消耗掉系统的大部分内存(可能是全部)。

可以小心避免形成循环引用，实在无法避免时可以采取补救措施，例如使用 IE 的 onunload 事件，将事件处理函数的引用赋值为 null。认识这个问题并理解闭包(及其机制)是在 IE 中避免此问题的关键所在。

comp.lang.javascript FAQ notes T.O.C.

- Written by Richard Cornford. March 2004.
- With corrections and suggestions by:-
    - Martin Honnen.
    - Yann-Erwan Perio (Yep).
    - Lasse Reichstein Nielsen. (definition of closure)
    - Mike Scirocco.
    - Dr John Stockton.
    - Garrett Smith.